Vite: 一个异步高性能的通用去中心化应用平台

刘春明 王东 伍鸣
charles@vite.org daniel@loopring.org woo@vite.org

2018 年 4 月 13 日

Abstract

摘要。

1 简介

1.1 定义

一个通用去中心化应用平台，可以支持一组智能合约，每个智能合约都是一个拥有独立状态以及不同操作逻辑的交易状态机，它们之间可以通过消息传递的方式进行通信。

从整体上，这个系统是一个交易状态机。系统的状态 $\boldsymbol{s} \in \boldsymbol{S}$ 也称为世界状态，是由每一个独立账户的状态构成的。一个能够引起世界状态改变的事件称为交易。更形式化的定义如下：

Definition 1.1 (交易状态机) 一个交易状态机是一个四元组：$(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{g}, \boldsymbol{\delta})$，其中，$\boldsymbol{T}$ 是交易的集合，$\boldsymbol{S}$ 是状态的集合，$\boldsymbol{g} \in \boldsymbol{S}$ 为初始状态，也称为“创世块”，$\boldsymbol{\delta} : \boldsymbol{S} \times \boldsymbol{T} \to \boldsymbol{S}$ 是状态转移函数。

这种交易状态机的语义是一个离散迁移系统，定义如下：

Definition 1.2 (交易状态机的语义) 一个交易状态机 $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{s_0}, \boldsymbol{\delta})$ 的语义是一个离散迁移系统：$(\boldsymbol{S}, \boldsymbol{s_0}, \to)$。$\to \in \boldsymbol{S} \times \boldsymbol{S}$ 是迁移关系，

同时，去中心化应用平台是一个分布式系统，具有最终一致性。通过某种共识算法，节点间可以就最终的状态达成一致。在现实场景中，智能合约的状态中保存的是一个去中心化应用的全部数据，体积比较大，无法在节点间传输。因此，节点间需要通过传递一组交易，来达成最终状态的一致。我们将这样的一组交易组织成某种特定的数据结构，通常称之为账本。

Definition 1.3 (账本) 账本是由一组交易构成的，具有递归构造的抽象数据类型，定义如下：

$$\begin{cases} l = \Gamma(T_t) \\ l = l_1 + l_2 \end{cases}$$

其中，$T_t \in 2^T$，表示一组交易，$\Gamma \in 2^T \to L$，表示通过一组交易构造一个账本的函数，L 是账本的集合，$+ : L \times L \to L$，表示将两个子账本合并成一个账本的操作。

需要注意的是，在此类系统中，账本通常用来表示一组交易，而不是一个状态。在比特币 [1] 和以太坊 [2] 中，账本是一个区块链结构，其中交易是全局有序的。修改账本中的一个交易，需要重新构造账本中的一个子账本，从而提高了篡改交易的成本。

根据同一组交易，可以构造出不同的有效账本，但它们所表示的交易顺序不同，因此可能会导致系统进入不同的状态。当这种情况发生时，通常称账本发生了“分叉”。

Definition 1.4 (分叉) $T_t, T_t' \in 2^T$，$l = l_1 + \Gamma_1(T_t)$，$l' = l_1 + \Gamma_2(T_t')$，且 $l \neq l'$，则称 l 和 l' 是分叉账本。

根据交易状态机的语义，我们可以很容易的证明，从一个初始状态开始，如果账本不分叉，则每个节点最终会进入相同的状态。那么，如果接收到分叉的账本，就一定进入不同的状态吗？这取决于账本中交易的内在逻辑，以及账本如何组织交易之间的偏序。现实中，经常会出现一些满足交换律的交易，却因为账本设计的问题，频繁的引

起分叉。当系统从一个初始状态出发，接收两个分叉的账本，最终进入同一状态，我们称这两个账本为伪分叉账本。

Definition 1.5 (伪分叉) *有初始状态 $s_0 \in S$，账本 $l_1, l_2 \in L$，$s_0 \xrightarrow{l_1} s_1, s_0 \xrightarrow{l_2} s_2$。若 $l_1 \neq l_2$，且 $s_1 = s_2$，则称这两个账本 l_1, l_2 为伪分叉 (false fork) 账本。*

一个设计良好的账本，应该尽量降低伪分叉发生的概率。

当分叉发生时，每个节点都需要从多个分叉的账本中选择一个，为确保状态的一致性，这些节点需要采用同一个算法完成选择，这个算法称为共识算法。

Definition 1.6 (共识算法) *共识算法是一个函数，它接收一个账本的集合，返回其中唯一一个账本：*

$$\Phi : 2^L \to L$$

共识算法是系统设计的一个重要内容，一个好的共识算法应该具有较高的相交速度 (high convergence speed)，减少共识在不同分叉间摇摆，并对恶意攻击具有较高的防范能力。

1.2 当前进展

以太坊 [3] 率先实现了这样一个系统。在以太坊的设计中，世界状态的定义是：$S = \Sigma^A$，是由每个账户 $a \in A$ 与该账户的状态 $\sigma_a \in \Sigma$ 构成的映射。因此，以太坊的状态机中任何一个状态都是全局的，这表示一个节点在每个时刻都可以获取任何一个账户的状态。

以太坊的状态转移函数 δ，是由一组程序代码来定义的，每组代码被称为一个智能合约。以太坊定义了一个图灵完备的虚拟机，称为 EVM，运行在其上的指令集称为 EVM 代码。用户可以通过一种语法类似于 javascript 的程序语言 Solidity 来开发智能合约，并编译成 EVM 代码，部署到以太坊上 [4]。一旦智能合约部署成功，就相当于定义了该合约账户 a 收到一个交易后的状态转移函数 δ_a。EVM 目前在此类平台中被广泛使用，但它也存在一些问题。例如，缺少库函数支持，安全性问题突出等。

以太坊的账本结构是区块链 [1]，区块链由区块构成，每一个区块中包含一组交易的列表，后一个区块引用前一个区块的 hash，构成一个链状结构。

$$\Gamma(\{t_1, t_2, ... | t_1, t_2, ... \in T\}) = (..., (t_1, t_2, ...)) \quad (1)$$

这个结构的最大好处是有效的防止交易被篡改，但由于它维护的是所有交易的全序，任何两个交易交换顺序，都会生成一个新账本，也造成这种结构具有较高的分叉概率。事实上，在这个定义下，交易状态机的状态空间被看作一棵树：初始状态是根节点，不同的交易顺序代表不同的路径，叶子节点是最终状态。现实的情况下，大量叶子节点的状态是相同的，这就造成了大量的伪分叉。

以太坊的共识算法 Φ 称为 PoW，该算法率先在比特币协议中提出 [1]。PoW 算法依赖于某个易于验证但难于求解的数学问题，例如，根据一个 hash 函数 $h : N \to N$，求解 x，使 $h(T + x) \geq d$，d 是一个给定的数，称为难度，T 是区块中包含的交易列表的二进制表示。在区块链的每个区块中，都会包含这类问题的一个解。将全部区块的难度加起来，就是一个区块链账本的总难度：

$$D(l) = D(\sum_i l_i) = \sum_i D(l_i) \quad (2)$$

因此，在从分叉中选择正确账本的时候，只要选择总难度最高的分叉即可：

$$\Phi(l_1, l_2, ..., l_n) = l_m \text{ where } D(l_m) = max(D(l_i)) \quad (3)$$

PoW 共识算法具有较好的安全性，迄今为止在比特币和以太坊中运行得很好。但这个算法有两个主要问题，第一是求解数学难题需要消耗大量计算资源，造成能源浪费；第二是该算法相交速度较慢，因而影响了系统整体的吞吐。目前，以太坊整体的 TPS 只有 15 左右，完全无法满足去中心化应用的需求。

1.3 改进方向

在以太坊诞生之后，以太社区和其他一些同类项目开始从不同方向对系统加以改进。从系统的抽象模型来看，可以改进的方向主要包括以下几个：

- 改进系统状态 $\boldsymbol{S}$
- 改进状态迁移函数 $\boldsymbol{\delta}$
- 改进账本结构 $\boldsymbol{\Gamma}$
- 改进共识算法 $\boldsymbol{\Phi}$

1.3.1 改进系统状态

对系统状态的主要改进思路是将全局的世界状态局部化，每个节点不再关心全部交易和状态转移，只维护整个状态机的一个子集。这样集合 S 和集合 T 的势都大为缩减，从而提高了系统扩展性。此类系统包括：Cosmos[5]，Aelf[6]，PChain 等。

从本质上讲，此类基于侧链的方案牺牲了状态的全局性，以换取系统的扩展性。这使得每个运行在其上的 dApp 的去中心化程度都被削弱——一个智能合约的交易历史不再被全网每一个节点保存，而只被一部分节点保存。除此之外，跨合约交互也会成为此类系统的瓶颈。例如，Cosmos 中，不同的 Zone 交互，需要通过一个共同的链 Hub 来完成 [5]。

1.3.2 改进状态迁移函数

一些项目立足于改进 EVM，提供更为丰富的智能合约编程语言。例如，RChain 定义了一种基于 π 演算的智能合约语言 Rholang；NEO 中的智能合约称为 NeoContract，可以用 Java，C# 等主流编程语言开发；EOS 使用 C/C++ 来编程。

1.3.3 改进账本结构

账本结构的改进方向是构造等价类，将多个交易全局有序的线性账本规约为一个只记录部分偏序关系的非线性账本，这种非线性账本结构是一个 DAG(有向无环图)。目前，Byteball[7]，IOTA[8]，Nano[9] 等项目基于 DAG 的账本结构实现了加密货币功能。也有一些项目在尝试利用 DAG 实现智能合约，但迄今为止在这个方向上的改进还在探索中。

1.3.4 改进共识算法

共识算法的改进大部分是为了提高系统的吞吐，主要方向是抑制伪分叉的产生。下面我们讨论伪分叉与哪些因素有关。

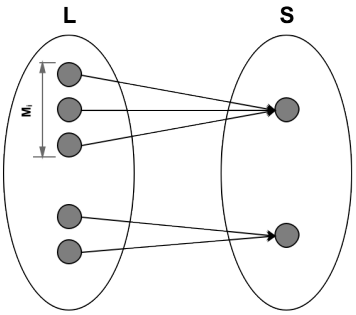

图 1: 伪分叉

如图所示，L 是针对某个交易集合所有可能的分叉账本的集合，S 是以不同顺序执行这一组交易，所能到达的状态的集合。根据定义 1.4，映射 $f : L \to S$ 是一个满射；而根据定义 1.5，这个映射不是单射。下面我们来计算伪分叉的概率：

假设共有 C 个用户有权生产账本，$M = |L|$，$N = |S|$，$M_i = |L_i|$，其中，$L_i = \{l | f(l) = s_i, s_i \in S\}$。则伪分叉概率为：

$$P_{ff} = \sum_{i=1}^{N} \left(\frac{M_i}{M} \right)^C - \frac{1}{M^{C-1}} \tag{4}$$

从这个公式可以看出，为了降低伪分叉概率，可以有两种途径：

- 在账本集合 L 上建立等价关系，对其划分等价类，构造分叉更少的账本
- 限制有权生产账本的用户，从而减少 C

第一种途径是 Vite 设计的重要方向，后文将详细论述；第二种途径现在已被多种算法所采用。在 PoW 算法中，任何用户都有权生产区块；而 PoS 算法将生产区块的权力限制在那些拥有系统权益的用户中；DPoS 算法 [10] 将有权生产区块的用户进一步限制在一组代理节点范围内。

目前，通过改良共识算法，已经产生出一些比较有影响力的项目。例如，Cardano 采用了一种 PoS 算法，称为 Ouroboros，文献 [11] 对该算法相关性质给出了严格证明；EOS 采用了 DPoS 算法，通过快速生产区块，提高系统的吞吐；Qtum[12] 的共识算法也是一种 PoS 算法;RChain 采用的 Casper 算法也是一种 PoS 算法。

还有一些其他项目在共识算法的改进上提出了自己的方案。NEO 采用了一种 BFT 算法，称为 dBFT；Cosmos 采用了一种称为 Tendermint[13] 的算法。

2 账本

2.1 概述

账本的作用主要是为了确定交易之间的顺序，交易的顺序会影响以下两个方面：

- **状态一致性：** 由于系统的状态不是一个 CRDT(Conflict-free replicated data types)[14]，因此，交易不都是可交换的，不同的交易执行顺序可能会导致系统进入不同的状态。
- **hash 有效性：** 账本中，交易会被打包成区块，区块中包含互相引用的 hash。交易的先后顺序会影响账本中 hash 引用的连通性。这种影响的范围越大，篡改交易的成本就越大。这是因为，改变任何一个交易，都必须重建所有直接或间接引用该交易的区块的 hash。

而账本的设计也有两个主要目标：

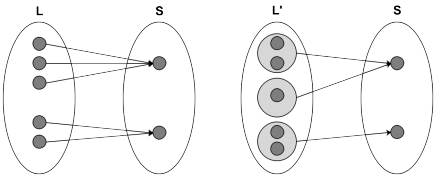

图 2: 账本合并

- **降低伪分叉率：** 如前文讨论，降低伪分叉率可以通过建立等价类，尽量将导致系统进入同一状态的一组账本合并成一个账本来实现。如上图，根据伪分叉率公式可以算得，左图的账本伪分叉率 $P_{ff} = \left(\frac{3}{5}\right)^C + \left(\frac{2}{5}\right)^C - \frac{1}{5^{C-1}}$；而合并账本空间后，右图的伪分叉率为 $P_{ff}{}' = \left(\frac{2}{3}\right)^C + \left(\frac{1}{3}\right)^C - \frac{1}{3^{C-1}}$。可知当 $C > 1$，$P_{ff}{}' < P_{ff}$。也就是说，我们应尽量减小账本中交易之间的偏序关系，允许更多的交易之间的顺序可交换。
- **防篡改：** 当账本 l 中一个交易 t 被修改，账本的两个子账本 $l = l_1 + l_2$ 中，子账本 l_1 不受影响，而子账本 l_2 中的 hash 引用需要重建，以构成一个新的有效账本 $l' = l_1 + l_2{}'$。受影响的子账本 $l_2 = \Gamma(T_2), T_2 = \{x | x \in T, x > t\}$。由此可见，想提高篡改交易的成本，需要在账本中尽量多的维护交易之间的偏序关系，以扩大篡改的影响范围 $|T_2|$。

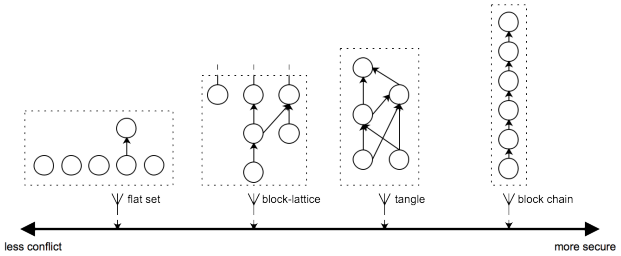

图 3: 账本结构比较

显然，以上两个目标是互相矛盾的，在设计账本结构时必须作出必要的权衡取舍。由于账本维护的是交易之间的偏序，因此它本质上是一个偏序集 (poset)[15]，如果用哈斯图 (Hasse diagram)[16] 来表示，在拓扑上就是一个 DAG。

上图对比了几种常见的账本结构，靠近左侧的账本维护更少的偏序关系，哈斯图显得比较扁平，具有更低的伪分叉率；靠近右侧的账本维护更多的偏序关系，哈斯图比较细长，具有更高的防篡改特性。

图中，最左侧的是一种中心化系统中常见的基于集合的结构，没有任何防篡改特性；最右侧则是典型的区块链账本，具有最好的防篡改特性；而介于二者中间的是两种 DAG 账本，左侧的是 Nano 采用的 block-lattice 账本 [9]，右侧是 IOTA 采用的 tangle 账本 [8]。从特性上看，block-lattice 维护了更少的偏序关系，更适合作为高性能去中心化应用平台的账本结构。但由于它的防篡改特性较差，会产生安全隐患，因此，迄今为止，除了 Nano 采用了该结构之外，还没有其他项目采用。

为了追求高性能，Vite 采用该账本结构。同时，通过引入一种额外的链式结构 Snapshot Chain，并且改进了共识算法，成功的弥补了 block-lattice 安全性方面的不足，后文将详细论述这两个改进。

2.2 前置约束

首先，我们来看一下采用这种账本结构对状态机模型的前置要求。这种结构本质上是将整个状态机看作是一组

独立的状态机的组合，每个账户对应一个独立的状体机，每个交易只影响一个账户的状态。在账本，对所有交易按账户分组，并把同一账户的交易组织成一条链。因此，我们对 Vite 中的状态 S 和交易 T 做出如下限制：

Definition 2.1 (交易的单自由度约束) 系统状态 $s \in S$，是由每个账户的状态 s_i 构成的向量 $s = (s_1, s_2, ..., s_n)$。对于 $\forall t_i \in T$，执行交易 t_i 后，系统状态发生如下转移：$(s_1', ..., s_i', ..., s_n') = \sigma(t_i, (s_1, ..., s_i, ..., s_n))$，需满足：$s_j' = s_j, j \neq i$。该约束称为交易的单自由度约束。

直观上，一个单自由度的交易只会改变一个账户的状态，不会影响系统中其他账户的状态。在状态空间向量所在的多维空间中，执行一次交易，系统状态只会沿平行于某个坐标轴的方向移动。请注意，这个定义要比比特币、以太坊等模型中的交易定义更为严格，比特币中的一个交易，会改变发送者和接收者两个账户的状态；以太坊则可能通过消息调用，改变两个以上的账户的状态。

在这个约束条件下，交易之间的关系得以简化。任何两个交易，要么是正交的，要么是平行的。这为依照账户对交易进行分组提供了条件。下面举一个例子来说明：

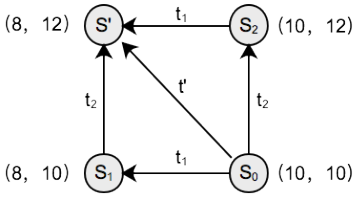

图 4: 单自由度交易和中间状态

如上图所示，假设 Alice 和 Bob 各有 10 元钱，系统的初始状态为 $s_0 = (10, 10)$。当 Alice 想给 Bob 转账 2 元时，在比特币和以太坊的模型中，可以通过一个交易 t'，使系统直接进入最终状态：$s_0 \xrightarrow{t'} s'$。

而在 Vite 的定义中，交易 t' 同时改变了 Alice 和 Bob 两个账户的状态，不符合单自由度的原则。因此，这个交易必须被拆分成两笔交易：

1) 表示 Alice 转出 2 元的交易 t_1

2) 表示 Bob 转入 2 元的交易 t_2。

这样，从初始状态到达最终状态 s' 可以有两条不同的路径 $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ 和 $s_0 \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$。这两条路径分别通过中间状态 s_1 和 s_2，这两个中间状态是最终状态 s' 在两个账户上维度的投影。也就是说，如果只关心其中某一个账户的状态，只需要执行该账户对应的所有交易，而不需要执行其他账户的交易。

下面我们来定义如何将以太坊中的交易，拆分成 Vite 所要求的单自由度交易：

Definition 2.2 (交易分解) 将一个自由度大于 *1* 的交易拆分成一组单自由度交易的过程，称为交易分解。一笔转账交易可以拆分成一个出账交易和一个入账交易；一个合约调用交易可以拆分成一个合约请求交易和一个合约响应交易；每个合约内部的消息调用，可以拆分成一个合约请求交易和一个合约响应交易。

这样，账本中就有两种不同类型的交易，它们被称为“交易对”：

Definition 2.3 (交易对) 一个入账交易或合约请求交易，统称为“请求交易”；一个出账交易或合约响应交易，统称为“响应交易”。一个请求交易和一个对应的响应交易，称为交易对。发起请求交易 t 的账户，记作 $A(t)$；对应的响应交易记作：$\widetilde{t}$，该交易对应的账户，记作 $A(\widetilde{t})$。

根据以上定义，我们可以得出 Vite 中，任何两个交易之间可能存在的关系：

Definition 2.4 (交易的关系) 对于两个交易 t_1 和 t_2，可能存在如下关系：

正交：若 $A(t_1) \neq A(t_2)$，则称两个交易正交，记作 $t_1 \perp t_2$；

平行：若 $A(t_1) = A(t_2)$，则称两个交易平行，记作 $t_1 \parallel t_2$；

因果：若 $t_2 = \widetilde{t_1}$，则称两个交易具有因果关系，记作 $t_1 \rhd t_2$，或者 $t_2 \lhd t_1$。

2.3 账本定义

定义一个账本，就是定义一个偏序集。首先，我们来定义 Vite 中，交易之间的偏序关系：

Definition 2.5 (交易的偏序) 我们用二元关系 $<$ 来表示两个交易的偏序关系，有：

一个响应交易，必须排在一个对应的请求交易之后：$t_1 < t_2 \Leftrightarrow t_1 \rhd t_2$；

一个账户的所有交易，必须严格全局有序：$\forall t_1 \parallel t_2$，必有：$t_1 < t_2$，或 $t_2 < t_1$。

由于建立在交易集合 T 上的偏序关系 $<$ 满足：

- 非自反性 (irreflexive)：$\forall t \in T$，不存在 $t < t$；
- 传递性 (transitive)：$\forall t_1, t_2, t_3 \in T$，若 $t_1 < t_2, t_2 < t_3$，则有 $t_1 < t_3$；
- 非对称性 (asymmetric)：$\forall t_1, t_2 \in T$，若 $t_1 < t_2$，则不存在 $t_2 < t_1$

这样，我们就可以用严格偏序集的方式来定义 Vite 的账本：

Definition 2.6 (Vite 账本) *Vite 账本是给定的交易集合 T，以及偏序关系 $<$，所构成的严格偏序集。*

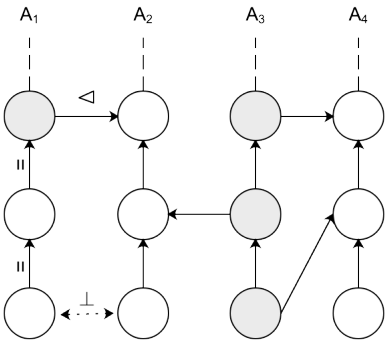

图 5: Vite 账本及交易之间的关系

一个严格偏序集，可以对应到一个 DAG 结构。上面所定义的 Vite 账本，结构类似于 block-lattice。交易分为请求交易和响应交易两种，每个交易对应一个单独的区块，每个账户 A_i 对应一条链，一个交易对中，响应交易引用其对应的请求交易的 hash。

3 快照链

3.1 交易确认

当账本发生分叉时，共识结果可能会在两个分叉账本间摇摆。例如，基于区块链结构的系统，如果节点接收到一个更长的分叉链，就会选择这个新分叉作为共识结果，而原分叉将被废弃，原分叉上的交易也会被回滚。在此类系统中，交易被回滚是一个非常严重的事件，将会导致双花 (double spend)。试想，一个商家接收到一笔付款，提供了商品或服务，之后这笔付款又被撤回，商家可能会因此面临损失。因此，用户在收到一笔付款交易时，需要等待系统对这笔交易进行“确认”，以确保这笔交易被回滚的概率足够低。

Definition 3.1 (交易确认) *当一个交易被回滚的概率小于一个给定的阈值 ϵ 时，称该交易为已确认 (confirmed)。$P_r(t) < \epsilon \Leftrightarrow t$ is **confirmed**。*

交易的确认是一个非常容易被混淆的概念，因为一个交易是否被确认实际上取决于隐含的置信度 $1 - \epsilon$。一个售卖钻石的商家和一个售卖咖啡的商家，在被双花攻击的时候，所蒙受的损失是不同的。因此，前者需要对交易设置更小的 ϵ。这也是比特币中确认数的本质。在比特币中，确认数表示一个交易在区块链中的深度，确认数越大，交易被回滚的概率越低 [1]。因此，商家可以通过设置等待多少确认数，间接设定确认的置信度。

交易被回滚的概率随时间而降低，是由于账本结构中存在 hash 引用关系。如前文所述，当账本的设计具有较好的防篡改特性时，回滚一个交易需要重构该交易所在区块的所有后继区块。随着新的交易被不断加入账本，某个交易的后继节点也越来越多，因此，被篡改的概率也随之下降。

而 block-lattice 结构中，由于交易是按账户分组的，一个交易只会附加到其所属账户的账户链末端，大部分其他账户产生的交易不会自动成为该交易的后继节点。因此，采用这个结构必须合理的设计共识算法，以避免双花的隐患。

Nano 采用了一个基于投票的共识算法 [9]，由一组用户选择的代表节点对交易进行签名，每个代表节点有一个权重，当某个交易获得的签名累计起来有足够多的权重时，就认为该交易被确认。这个算法有以下几个问题：

首先，如果需要更高的确认置信度，则需要提高投票权重的阈值，如果没有足够多的代表节点在线，就无法保证相交速度，有可能一个用户永远也搜集不到确认一个交易所必需的票数；

其次，交易被回滚的概率不随时间递减。这是因为任何时候，推翻一个历史投票的结果，所付出的成本都是一样的。

最后，历史的投票结果并没有被持久化到账本中，只保存在节点的本地存储中。当一个节点从其他节点获取账本时，没有办法可靠的量化一个历史交易被回滚的概率。

从本质上，投票的机制是一个偏中心化的解决方案。我们可以把投票结果看作是对账本状态的一个快照，这个快照会分布式的保存在网络中各个节点的本地存储中。为了拥有和区块链同样的防篡改能力，我们可以将这些快照也组织成链式结构，这就是 Vite 设计的核心之一——快照链 [17]。

3.2 快照链的定义

快照链是 Vite 中最重要的存储结构，它的主要作用是维护 Vite 账本的共识。首先我们给出快照链的定义：

Definition 3.2 (快照块和快照链) *一个快照块存储一个 Vite 账本的状态快照，状态包括：账户的余额、合约状态的 Merkle root，每个账户链中最后一个块的 hash。快照链是由快照块组成的链式结构，后一个快照块引用前一个快照块的 hash。*

一个用户账户的状态包含余额和账户链最后一个块的 hash；一个合约账户的状态，除了包含以上两个字段之外，还包含该账户合约状态的 Merkle root hash。账户的状态的结构如下：

```
struct AccountState {
  // 账户余额
  uint256 balance;
  // 合约状态的Merkle root
  optional uint256 storageRoot;
  // 账户链最后一个交易的hash
  uint256 lastTransaction;
}
```

快照块的结构定义如下：

```
struct SnapshotBlock {
  // 前一个块的hash
  uint256 prevHash;
  // 快照信息
  map<address, AccountState> snapshot;
  // 签名
  uint256 signature;
}
```

快照链中第一个快照块称为“创世快照”，保存的是账本中创世块的快照。

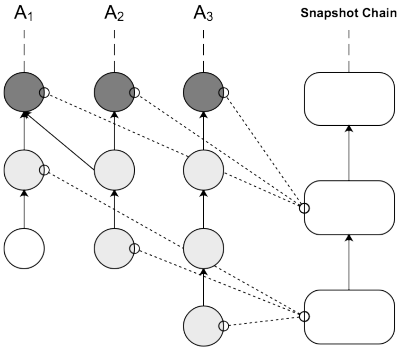

图 6: 快照链

由于快照链中每个快照块都对应于 Vite 账本的唯一分叉，因此，在快照块不发生分叉的情况下，通过快照块，可以确定 Vite 账本的共识结果。

3.3 快照链与交易确认

引入了快照链之后，block-lattice 结构天然的安全性缺陷就得到了弥补。攻击者想生成一个双花交易，除了要重建 Vite 账本中的 hash 引用之外，还需要重建快照链中，首次快照该交易的快照块之后的所有区块，并且需要产生一条更长的快照链。这样，攻击成本将大大提高。

在 Vite 中，交易的确认机制类似于比特币，定义如下：

Definition 3.3 (Vite 交易确认) *在 Vite 中，一个交易被快照链所快照，则成为该交易被确认。第一次快照该交易的快照块的深度，称为交易的确认数。*

在这个定义下，快照链每增长一个区块，此前所有已确认交易的确认数都增加 1，双花攻击成功的概率随着快照链的不断增加而逐渐下降。这样，用户就可以根据具体的场景，通过等待不同的确认数，来定制所需的确认置信度。

快照链本身依赖一个共识算法，如果快照链发生分叉，则选取**最长的**分叉作为合法分叉。当快照链切换到新的分叉时，原有快照信息会被回滚，相当于原来对账本达成的共识被推翻，由新的共识结果取代。因此，快照链是整个系统安全性的基石，需要非常认真的对待。

3.4 压缩存储

由于快照链中每一个快照块都需要保存所有账户的状态，耗费的存储空间非常大，因此需要对快照链进行压缩。

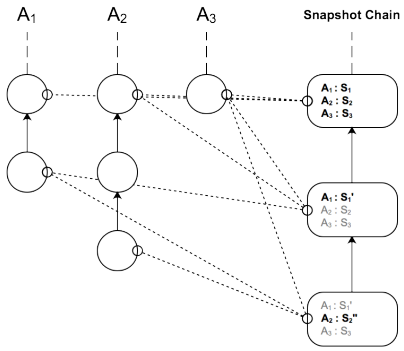

图 7: 压缩前的快照

压缩快照链存储空间的基本思路是利用增量存储：一个快照块只保存相比于前一个快照块发生变化的数据。如果一个账户在两个快照之间没有发生任何交易，则后一个快照块不保存该账户的数据。

要恢复快照信息，可以从前向后依次遍历快照块，将每一个快照块中的数据覆盖到当前的数据上即可。

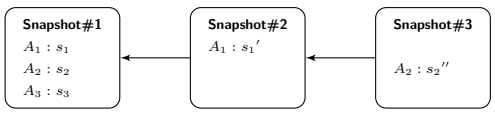

图 8: 压缩后的快照

由于每个快照只保存在快照时刻每个账户最后的状态，不关心中间状态，无论在两个快照块之间，一个账户产生了多少交易，在快照中也只保存一份数据。因此，一个快照块最多占用 $S * A$ 个字节。其中，$S = \text{sizeof}(s_i)$，为每个账户状态占用的字节数，A 是系统总账户数。如果活跃账户与总账户数的平均比例是 a，则压缩率为 $1 - a$。

4 共识

4.1 设计目标

在设计共识协议时，我们需要充分考虑以下因素：

- **性能**。Vite 的首要设计目标是快速，为确保系统具有高吞吐、低延迟的性能表现，我们需要采用具有更高相交速度的共识算法。

- **扩展性**。Vite 是一个公共平台，向所有去中心化应用开放，因此，扩展性也是一个重要的考量因素。

- **安全性**。虽然 Vite 的设计理念不是追求极致的安全性，但仍然需要确保足够的安全性底线，有效防范各类攻击。

对比现存的一些共识算法，PoW 的安全性更好，在恶意节点低于 50% 的情况下可以确保达成共识。但 PoW 的相交速度较慢，无法满足性能要求；PoS 及其变种算法去掉了求解数学难题的步骤，提高了相交速度和单次攻击成本，降低了能源消耗。但 PoS 的扩展性仍然较差，而且 Nothing at Stake 问题 [18] 较难解决；BFT 系列算法在安全性和性能方面有较好表现，但其扩展性是一个问题，通常比较适合于私有链或联盟链；DPoS[10] 系列算法通过限制生成区块的权限，有效降低了伪分叉率，在性能和扩展性方面表现良好。相应的，DPoS 在安全性方面稍有牺牲，需要保证恶意节点数不超过 1/3[19]。

综合来看，DPoS 算法在性能、扩展性方面有比较明显的优势，因此，我们选择 DPoS 作为 Vite 共识协议的基础，并在其基础上进行适当扩展，通过分层共识协议和异步模型，进一步提高平台的整体性能。

4.2 分层共识

Vite 的共识协议称为 HDPOS(Hierarchical Delegated Proof Of Stake)。基本思想是对共识函数 Φ 进行函数分解 (functional decomposition):

$$\begin{aligned}\Phi(l_1, l_2, \ldots, l_n) = \Psi(&\Lambda_1(l_1, l_2, \ldots, l_n),\\ &\Lambda_2(l_1, l_2, \ldots, l_n), \ldots\\ &\Lambda_m(l_1, l_2, \ldots, l_n))\end{aligned} \tag{5}$$

其中，$\Lambda_i : 2^L \to L$，称为局部共识函数，返回的结果称为局部共识；$\Psi : 2^L \to L$，称为全局共识函数，它从一组候选的局部共识中选取一个唯一结果，作为最终的共识结果。

这样拆分之后，整个系统的共识变成了两个独立的过程：

局部共识 生产用户账户或合约账户中的请求交易和响应交易对应的区块，并写入账本；

全局共识 对账本中的数据进行快照，生产快照块。如果账本有分叉，则选择其中一个。

4.3 区块生产权和共识组

那么，谁有权生产账本中的交易区块和快照链中的快照块，采用什么共识算法达成共识呢?

由于 Vite 的账本结构是按账户不同组织成多条账户链的，我们可以方便的按账户的维度来限定账本中区块的生产权，再加上快照块的生产权归属于单独的一组用户。这样，我们就可以将若干条账户链或快照链归入一个共识组，在共识组内部，用统一的方式来生产区块并达成共识。

Definition 4.1 (共识组) *共识组是一个元组 (L, U, Φ, P)，描述了账本的一部分或快照链的共识机制。其中，$L \in A|\{A_s\}$，表示账本中属于该共识组的一条或若干条账户链，或者是快照链；U 表示拥有 L 指定的链上区块生产权的用户;Φ 指定该共识组的共识算法;P 指定共识算法的参数。*

在这样的定义下，用户可以根据需要，灵活的设置共识组，并选择不同的共识参数。下面我们对不同的共识组加以详细说明:

4.3.1 快照共识组

快照链所在的共识组称为快照共识组，这是 Vite 中最重要的一个共识组。快照共识组的共识算法 Φ 对应分层模型中的 Ψ，采用 DPoS 算法。由参数 P 指定代理节点数和出块间隔。

例如，可以采用 25 个代理节点，以 10 秒为间隔生产快照块。这样可以保证交易的确认速度足够快，一个交易达到 6 次确认最多需要等待 60 秒。

4.3.2 私有共识组

私有共识组只适用于账本中交易区块的生产，属于私有共识组的账户链，区块只能由账户的私钥拥有者来生产。默认情况下所有用户账户都属于私有共识组。

私有共识组的最大好处是降低分叉概率。因为只有一个用户有权生产区块，发生分叉的唯一可能是用户主动发起双花攻击，或程序发生错误。

私有共识组的缺点是用户节点必须在线，才能打包交易。这对合约账户来说不太适合，一旦合约拥有者的节点失效，没有其他节点可以代替它生产合约的响应交易，相当于降低了 dApp 的服务可用性。

4.3.3 委托共识组

在委托共识组中，由一组指定的代理节点，通过 DPoS 算法，代替用户打包账户链上的交易。无论是用户账户还是合约账户，都可以加入委托共识组。用户可以设置一组单独的代理节点，建立一个新的委托共识组。Vite 中还有一个默认的委托共识组，来帮助所有未单独建立委托共识组的账户打包交易，这个共识组也称为**公共共识组**。

委托共识组适合于大部分合约账户，因为合约账户的交易大部分都是合约响应交易，相比于用户账户的收款交易，需要更高的可用性和更低的延迟。

4.4 共识的优先级

在 Vite 协议中，全局共识的优先级高于局部共识。当局部共识发生分叉的时候，以全局共识选择的结果为准。也就是说，一旦全局共识选定局部共识的一个分叉作为最终结果，即使未来账本中某个账户链出现更长的分叉，也不会使全局共识结果回滚。

在实现跨链协议的时候，这个问题需要格外注意。由于某个目标链可能发生回滚，映射该链的中继合约对应的账户链也需要随之回滚。这个时候，如果中继链的局部共识已经被全局共识采纳，是无法完成回滚的，可能会导致中继合约和目标链的数据不一致。

避免这个问题的方法是在共识组参数 P 中设置一个 *delay* 参数，指定快照共识组在局部共识完成后 *delay* 个区块之后，才开始进行快照。这样将大大降低中继合约不一致的概率，但也无法完全避免。在中继合约的代码逻辑中，还需要对目标链回滚的情况做单独处理。

4.5 异步模型

为了进一步提高系统吞吐，我们需要在共识机制上支持更完善的异步模型。

一个交易的生命周期包括：交易发起、交易写入、交易确认。为了提高系统的性能，我们需要将这三个步骤设计成异步模式。这是因为在不同的时刻，用户发起交易的

数量是不同的，而系统处理交易写入和确认交易的速度相对固定。异步模式有利于削平波峰波谷，从而提高系统的整体吞吐能力。

比特币和以太坊等系统的异步模型比较简单：将所有用户发起的交易放入一个未确认交易池中，当矿工将其打包到一个区块中，该交易同时完成了写入和确认，当区块链继续增长时，该交易最终到达预设的确认置信度。

这个异步模型存在两个问题：

- 交易在未确认状态下没有持久化到账本。未确认的交易是不稳定的，没有参与共识，也无法防止交易重复发送；
- 交易的写入和确认之间没有异步机制。交易只有在确认时才写入账本，写入速度受制于确认速度。

Vite 协议建立了更完善的异步模型：首先，将交易拆分成基于“请求-响应”模式的交易对，无论是转账还是合约调用，当一个请求交易被写入账本，则代表该交易被成功发起。另外，交易被写入账本和确认也是异步的。交易可以先被写入 Vite 的 DAG 账本中，不会被确认过程阻塞。交易的确认通过快照链来完成，快照的动作也是异步进行的。

这是一个典型的生产者-消费者模型，在交易的生命周期中，无论上游的生产速率如何变动，下游都可以以恒定的速率消化待处理的交易，从而充分利用平台资源，提高了系统的吞吐。

5 虚拟机

5.1 EVM 兼容性

当前以太坊已经拥有大量开发者，也有不少基于 Solidity 和 EVM 开发的智能合约投入应用。因此，我们决定在 Vite 的虚拟机上提供 EVM 兼容性，大部分 EVM 指令集可以在 Vite 中保持原有语义。但由于 Vite 的账本结构及交易定义与以太坊不同，一些 EVM 指令的语义需要重新定义，例如获取区块信息的一组指令。详细的语义差异可以参考附录 A。

其中，差异最大的是消息调用的语义，下面我们来详细讨论。

5.2 事件驱动

在以太坊的协议中，一个交易或消息，可能会影响多个账户的状态。例如一个合约调用交易，可能会通过消息调用，使多个合约账户状态同时发生变化。这些状态变化要么同时发生，要么就一个都不发生。因此，以太坊中的交易实际上是一种满足 ACID (Atomicity, Consistency, Isolation, Durability)[20] 特性的刚性事务，这也是以太坊缺乏扩展性的一个重要原因。

基于扩展性及性能考虑，Vite 采用了满足 BASE(Basically Available, Soft state, Eventual consistency)[21] 语义的最终一致性方案。具体来说，我们将 Vite 设计成一个事件驱动的架构 (Event-Driven Architecture, EDA)[?]。每个智能合约被看作是一个独立的服务，合约之间可以通过消息通信，但不共享任何状态。

因此，我们需要在 Vite 的 EVM 中，取消跨合约进行同步函数调用的语义，只允许合约间通过消息通信。受影响的 EVM 指令主要是 **CALL**，和 **STATICCALL**。在 Vite EVM 中，这两条指令不会立即执行，也不会返回调用的结果，只会生成一个请求交易写入账本。因此，这个指令在 Vite 中不再有函数调用的语义，而是向一个账户发送消息。

5.3 智能合约语言

以太坊提供了一种图灵完备的编程语言 Solidity，用于开发智能合约。为了支持异步语义，我们对 Solidity 进行了扩展，定义了一组用于消息通信的语法。扩展之后的 Solidity 称为 xSolidity。

xSolidity 将支持 Solidity 的大部分语法，但不再支持合约外的函数调用。开发者可以通过 *message* 关键字来定义消息，并通过 *on* 关键字定义消息处理器 (Message Handler)，从而实现跨合约通信功能。

下面举一个例子，合约 A 需要调用合约 B 中的 add() 方法，根据返回值更新自己的状态。在 Solidity 中，可以通过函数调用的方式来实现，代码如下：

```
pragma solidity ^0.4.0;

contract B {
    function add(uint a, uint b) returns
    (uint ret) {
        return a + b;
```

```
    }
}

contract A {
    uint total;

    function invoker(address addr, uint a,
    uint b) {
        // message call to A.add()
        uint sum = B(addr).add(a, b);
        // use the return value
        if (sum > 10) {
            total += sum;
        }
    }
}
```

而在 xSolidity 中，函数调用代码 `uint sum = B(addr).add(a, b);` 不再有效，取而代之的是，合约 A 和合约 B 通过彼此发送消息来进行异步通信。代码如下：

```
pragma xsolidity ^0.1.0;

contract B {
    message Add(uint a, uint b);
    message Sum(uint sum);

    Add.on {
        // read message
        uint a = msg.data.a;
        uint b = msg.data.b;
        address sender = msg.sender;
        // do things
        uint sum = a + b;
        // send message to return result
        send(sender, Sum(sum));
    }
}

contract A {
    uint total;

    function invoker(address addr, uint a,
    uint b) {
        // message call to B
        send(addr, Add(a, b))
        // you can do anything after sending
        // a message other than using the
        // return value
    }
    Sum.on {
        // get return data from message
        uint sum = msg.data.sum;
        // use the return data
        if (sum > 10) {
            total += sum;
        }
    }
}
```

在代码第一行，通过 `pragma xsolidity 0.1.0;` 定义了源代码是用 xSolidity 编写的，将不能直接使用 Solidity 编译器编译，以避免编译出的 EVM 字节码有不符合预期的语义。Vite 会提供一个专门的编译器，用于编译 xSolidity。这个编译器是部分向前兼容的：如果 Solidity 代码中没有与 Vite 语义冲突的部分，将可以直接通过编译，否则会报错。例如，本地函数调用、向其他账户转账等语法将保持兼容；获取跨合约函数调用的返回值，以及货币单位 `ether` 等将无法通过编译。

合约 A 中，当 `invoker` 函数被调用时，会向合约 B 发送 `Add` 消息，这个过程是异步的，结果不会当场返回。因此，需要在 A 中通过 `on` 关键字定义一个消息处理器，用于接收返回结果并更新状态。

合约 B 中会监听消息 `Add`，在处理完成后，通过向消息 `Add` 的发送者发送一个 `Sum` 消息来返回结果。

xSolidity 中的消息会被编译成 **CALL** 指令，并生成一个请求交易加入账本。在 Vite 中账本充当了合约之间异步通信的消息中间件，可以确保消息可靠存储并防止重复。同一个合约发往某个合约的多个消息，可以保证 FIFO(First In First Out)；不同合约向同一个合约发送的消息并不保证 FIFO。

需要注意的是，Solidity 中的事件 (Event) 和 xSolidity 中的消息不是同一个概念。事件是通过 EVM 日志，间

接向前端发送通知的。

5.4 标准库

在以太坊上开发智能合约的开发者，经常被 Solidity 缺乏标准库的问题所困扰。例如，路印协议中的环路验证必须在链外执行，其中一个重要原因就是 Solidity 没有提供浮点运算功能，尤其是求浮点数的 n 次方根 [22]。

在 EVM 中，可以通过 **DELEGATECALL** 指令来调用一个预先部署的合约，实现库函数的功能。以太坊也提供了几个预编译合约 (Precompiled Contract)，主要是几个 Hash 运算。但这些功能都太过简单，无法满足复杂的应用需求。

因此，我们将在 xSolidity 中，提供一系列标准库，例如：字符串处理、浮点运算、基本数学运算、容器、排序等。

基于性能考虑，这些标准库将以本地扩展 (Native Extension) 的方式实现，将大部分运算内置到 Vite 本地代码中，只在 EVM 代码中通过 **DELEGATECALL** 指令来完成方法的调用。

标准库可以根据需要不断扩展，但由于整个系统的状态机模型是确定性的，因此无法提供类似于随机数之类的功能。同以太坊类似，我们可以通过快照链的 hash 来模拟伪随机数。

5.5 燃料

燃料 (Gas) 在以太坊中主要有两个作用，一是量化 EVM 代码执行所消耗的计算资源和存储资源，二是确保 EVM 代码停机。根据可计算性理论，图灵机上的停机问题 (Halting Problem) 是一个不可计算问题 [23]。也就是说，无法通过分析 EVM 代码来判断一个智能合约能否在有限次执行后停止。

因此，Vite 中也保留了 EVM 中的 Gas 计算。不过在 Vite 中，没有 gas price 这个概念，用户不是通过支付手续费的方式来购买一次交易所需的燃料，而是通过一种基于配额的模型，来获取计算资源。配额的计算将在后文“经济模型”章节详细讨论。

6 经济模型

6.1 配额

由于 Vite 是一个通用的 dApp 平台，部署在其上的智能合约功能各不相同，每一个不同的智能合约，对吞吐和延迟的需求也不同。即使是同一个智能合约，在不同阶段对性能的要求也不同。

在以太坊的设计中，每一个交易在发起时需要指定一个 gas price，从而与其他交易竞争写入账本的机会。这是一个典型的竞价模型，原则上可以通过价格有效调控供给和需求的平衡。但由于用户在出价之前，很难量化当前的供需情况，也无法预测其他竞争者的出价，很容易发生市场失灵 (market failure)。而且，每次出价所竞争的资源都是针对一个交易的，没有一个按账户维度对资源进行合理配置的协议。

$$TPS_n = \frac{10}{S \cdot (H_n \ - \ H_{n-9} + 1)}$$

7 内置合约

7.1 定时调度

在以太坊中，智能合约是由交易驱动的，合约的执行只能通过用户主动发起一个交易来触发。而在有些应用中，需要一种定时调度功能，通过一个时钟来触发合约的执行。

在以太坊中，这个功能是通过第三方合约来实现的[1]，性能和安全性都无法保障。在 Vite 中，我们将定时调度功能加入到内置合约中。用户可以将自己的调度逻辑注册到定时调度合约中，公共共识组会将快照链作为时钟，根据用户定义的调度逻辑，向目标合约发送请求交易。

xSolidity 中有一个专门的 `Timer` 消息，用户可以在合约代码中，通过 `Timer.on` 来设置自己的调度逻辑。

7.2 路印协议

8 跨链

9 总结

Vite 的特点总结如下:

[1] *Ethereum Alarm Clock* 是一个用于调度其他合约执行的第三方合约，参见 `http://www.ethereum-alarm-clock.com/`

- 特点一。

10 致谢

致谢部分。

参考文献

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[3] Vitalik Buterin. Ethereum: a next generation smart contract and decentralized application platform (2013). *URL {http://ethereum. org/ethereum.html}*, 2017.

[4] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[5] Jae Kwon and Ethan Buchman. Cosmos a network of distributed ledgers. *URL https://cosmos.network/whitepaper*.

[6] Anonymous. aelf - a multi-chain parallel computing blockchain framework. *URL https://grid.hoopox.com/aelf_whitepaper_en.pdf*, 2018.

[7] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. *URL https://byteball.org/Byteball.pdf*.

[8] Serguei Popov. The tangle. *URL https://iota.org/IOTA_Whitepaper.pdf*.

[9] Colin LeMahieu. Raiblocks: A feeless distributed cryptocurrency network. *URL https://raiblocks.net/media/RaiBlocks_Whitepaper___English.pdf*.

[10] Anonymous. Delegated proof-of-stake consensus, a robust and flexible consensus protocol. *URL https://bitshares.org/technology/delegated-proof-of-stake-consensus/*.

[11] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. *URL https://eprint.iacr.org/2017/573.pdf*, 2017.

[12] Dai Patrick, Neil Mahi, Jordan Earls, and Alex Norta. Smart-contract value-transfer protocols on a distributed mobile application platform. *URL https://qtum.org/uploads/files/cf6d69348ca50dd985b60425ccf282f3.pdf*, 2017.

[13] Anonymous. Byzantine consensus algorithm. *URL https://github.com/tendermint/tendermint/wiki/Byzantine-Consensus-Algorithm*.

[14] Shapiro Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. *URL https://hal.inria.fr/inria-00609399v1*, 2011.

[15] Deshpande and Jayant V. On continuity of a partial order. *Proc. Amer. Math. Soc. 19 (1968), 383-386*, 1968.

[16] Weisstein and Eric W. Hasse diagram. *URL http://mathworld.wolfram.com/HasseDiagram.html*.

[17] Chunming Liu. Snapshot chain: An improvement on block-lattice. *URL https://medium.com/@chunming.vite/snapshot-chain-an-improvement-on-block-lattice-561aaabd1a2b.*

[18] Anonymous. Problems. *URL https://github.com/ethereum/wiki/wiki/Problems.*

[19] Dantheman. Dpos consensus algorithm - the missing white paper. *URL https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper.*

[20] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[21] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[22] Daniel Wang, Jay Zhou, Alex Wang, and Matthew Finestone. Loopring: A decentralized token exchange protocol. *URL https://github.com/Loopring/whitepaper/blob/master/en_whitepaper.pdf.*

[23] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, second edition, 2006.

Appendices

附录 A　EVM 指令集

A.0.1　0s: 停止和代数运算指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x00	STOP	0	0	停止代码执行。	语义相同
0x01	ADD	2	1	将两个操作数相加。	语义相同
0x02	MUL	2	1	将两个操作数相乘。	语义相同
0x03	SUB	2	1	将两个操作数相减。	语义相同
0x04	DIV	2	1	将两个操作数整除，如除数为 0，则返回 0。	语义相同
0x05	SDIV	2	1	带符号的整除。	语义相同
0x06	MOD	2	1	求模操作。	语义相同
0x07	SMOD	2	1	带符号的求模操作。	语义相同
0x08	ADDMOD	3	1	先将前两个操作数相加，再与第三个操作数求模。	语义相同
0x09	MULMOD	3	1	先将前两个操作数相乘，再与第三个操作数求模。	语义相同
0x0a	EXP	2	1	求两个操作数的乘方。	语义相同
0x0b	SIGNEXTEND	2	1	符号扩展。	语义相同

A.0.2　10s: 比较和位运算指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x10	LT	2	1	小于比较。	语义相同
0x11	GT	2	1	大于比较。	语义相同
0x12	SLT	2	1	带符号的小于比较。	语义相同
0x13	SGT	2	1	带符号的大于比较。	语义相同
0x14	EQ	2	1	比较是否相等。	语义相同
0x15	ISZERO	1	1	判断是否为 0。	语义相同
0x16	AND	2	1	按位与操作。	语义相同
0x17	OR	2	1	按位或操作。	语义相同
0x18	XOR	2	1	按位异或操作。	语义相同
0x19	NOT	1	1	按位非操作。	语义相同
0x1a	BYTE	2	1	取第二个操作数中的某一个字节。	语义相同

A.0.3　20s: SHA3 指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x20	SHA3	2	1	计算 Keccak-256 哈希。	语义相同

A.0.4　30s: 环境信息指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x30	ADDRESS	0	1	获取当前账户地址。	语义相同
0x31	BALANCE	1	1	获取一个账户的余额。	语义相同。 返回的是账户的 VT 余
0x32	ORIGIN	0	1	获取原始交易发送者地址。	语义相同。 如果在事件回调中使用
0x33	CALLER	0	1	获取直接调用者地址。	语义相同。 事件回调中也可以使用
0x34	CALLVALUE	0	1	获取调用交易中的转账金额。	语义相同
0x35	CALLDATALOAD	1	1	获取本次调用的参数数据。	语义相同
0x36	CALLDATASIZE	0	1	获取本次调用的参数数据大小。	语义相同
0x37	CALLDATACOPY	3	0	将调用参数数据拷贝到内存。	语义相同
0x38	CODESIZE	0	1	获取在当前环境中运行的代码的大小。	语义相同
0x39	CODECOPY	3	0	将当前环境中运行的代码拷贝到内存。	语义相同
0x3a	GASPRICE	0	1	获取当前环境的燃料价格。	语义不同。 永远返回 0。
0x3b	EXTCODESIZE	1	1	获取一个账户的代码大小。	语义相同
0x3c	EXTCODECOPY	4	0	将指定账户的代码拷贝到内存。	语义相同
0x3d	RETURNDATASIZE	0	1	获取前一次调用返回的数据大小。	语义相同
0x3e	RETURNDATACOPY	3	0	将前一次调用返回的数据拷贝到内存。	语义相同

A.0.5　40s: 区块信息指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x40	BLOCKHASH	1	1	获取一个区块的哈希。	语义不同。 返回相应的快照块的哈希。
0x41	COINBASE	0	1	获取所在区块挖坑受益人地址。	语义不同。 永远返回 0。
0x42	TIMESTAMP	0	1	返回所在区块的时间戳。	语义不同。 永远返回 0。
0x43	NUMBER	0	1	返回所在区块编号。	语义不同。 返回响应交易块在该账户链上的编号。
0x44	DIFFICULTY	0	1	返回所在区块的难度。	语义不同。 永远返回 0。
0x45	GASLIMIT	0	1	返回所在区块的燃料限额。	语义不同。 永远返回 0。

A.0.6　50s: 栈、内存、存储、控制流操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x50	POP	1	0	从栈顶弹出一条数据。	语义相同
0x51	MLOAD	1	1	从内存加载一个 word。	语义相同
0x52	MSTORE	2	0	保存一个 word 到内存	语义相同
0x53	MSTORE8	2	0	保存一个字节到内存。	语义相同
0x54	SLOAD	1	1	从存储中加载一个 word。	语义相同
0x55	SSTORE	2	0	保存一个 word 到存储。	语义相同
0x56	JUMP	1	0	跳转指令。	语义相同
0x57	JUMPI	2	0	条件跳转指令。	语义相同
0x58	PC	0	1	获取程序计数器的值。	语义相同
0x59	MSIZE	0	1	获取内存大小。	语义相同
0x5a	GAS	0	1	获取可用燃料数。	语义不同。 永远返回 0。
0x5b	JUMPDEST	0	0	标注一个跳转目的地。	语义相同

A.0.7　60s 和 70s: 压栈操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x60	PUSH1	0	1	将 1 字节对象压入栈顶。	语义相同
0x61	PUSH2	0	1	将 2 字节对象压入栈顶。	语义相同
⋮	⋮	⋮	⋮	⋮	
0x7f	PUSH32	0	1	将 32 字节对象 (整个字) 压入栈顶。	语义相同

A.0.8　80s: 复制操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x80	DUP1	1	2	复制栈中第 1 个对象，并压入栈顶。	语义相同
0x81	DUP2	2	3	复制栈中第 2 个对象，并压入栈顶。	语义相同
⋮	⋮	⋮	⋮	⋮	
0x8f	DUP16	16	17	复制栈中第 16 个对象，并压入栈顶。	语义相同

A.0.9　90s: 交换操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0x90	SWAP1	2	2	交换栈中第 1 个和第 2 个对象。	语义相同
0x91	SWAP2	3	3	交换栈中第 1 个和第 3 个对象。	语义相同
⋮	⋮	⋮	⋮	⋮	
0x9f	SWAP16	17	17	交换栈中第 1 个和第 17 个对象。	语义相同

A.0.10　a0s: 日志操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0xa0	LOG0	2	0	扩展日志记录，不设主题。	语义相同
0xa1	LOG1	3	0	扩展日志记录，1 个主题。	语义相同
⋮	⋮	⋮	⋮	⋮	
0xa4	LOG4	6	0	扩展日志记录，4 个主题。	语义相同

A.0.11 f0s: 系统操作指令集

指令代码	助记词	出栈数	入栈数	原始 EVM 语义	Vite EVM 语义
0xf0	CREATE	3	1	创建一个新合约。	语义相同
0xf1	CALL	7	1	调用另一个合约。	语义不同。 表示向一个账户发送消息， 返回值永远是 0。
0xf2	CALLCODE	7	1	调用另一个合约的代码， 改变本账户状态。	语义相同
0xf3	RETURN	2	0	停止执行并返回数据。	语义相同
0xf4	DELEGATECALL	6	1	调用另一个合约的代码，改变 本账户状态，保留原始交易信息。	语义相同
0xfa	STATICCALL	6	1	调用另一个合约，不允许改变状态。	语义不同。 表示向一个合约发送消息， 不改变目标合约的状态。 永远返回 0。所需要的结果 通过目标合约发送另一个消息
0xfd	REVERT	2	0	停止执行，恢复状态并返回数据。	语义相同 没有返回剩余燃料的语义。
0xfe	INVALID	∅	∅	无效指令。	语义相同
0xff	SELFDESTRUCT	1	0	停止执行，将合约设置为待删除， 返回所有余额。	语义相同