Telegram Open Network （TON）

中文版白皮书 v0.91

作者：Dr. Nikolai Durov

翻译：吴泰（交流微信：toozWu）

摘要

本文主要提供 Telegram Open Network（TON）和相关区块链、点对点、分布式存储和托管服务技术的描述。篇幅有限，将重点描述 TON 平台自定义和独特的功能，它对实现 TON 的既定目标非常重要。

介绍

Telegram Open Newtwork（TON）是一个快速、安全且可扩展的区块链项目。特殊情况下，它可达百万 TPS。它对普通用户与服务提供商友好。TON 的目标是能够容纳当前提出和计划中所有的应用。你可以认为 TON 是一个巨大的分布式超级计算机，或者更确切地说是一个巨大的“超级服务器”，旨在托管和提供各种服务。本文提供参考，不作为最终实现细节的允诺。在开发和测试阶段，一些细节可能会变化。

2019 年 6 月 **15** 日

1 TON 概述

Telegram Open Network（TON）是由以下组件组成：

- **灵活的多条链区块链平台**（TON Blockchain 或称 TON 区块链；参见第 2 章），能够达到百万 TPS，具有图灵完备智能合约，规则可升级的形式化区块链，支持多类型加密货币价值交换，支持小额支付通道和链下支付网络。TON 区块链提供了一些新颖和独特的功能，例如“自我修复”的垂直区块链机制（参见 2.1.17）和即时超立方体路由（Instant Hypercube Routing；参见 2.4.20），使其同时兼具快速，可靠，可扩展和一致性。

- **点对点网络**（TON P2P Network，或称 TON Network，或称 TON 网络；参见第 3 章），用于访问 TON 区块链，发送交易申请，以及接收用户感兴趣的区块链更新（例如，与客户账户有关的智能合约更新），但它也可以支持任意分布式服务，无论是否是区块链上的。

- **分布式文件存储技**（TON Storage 或称 TON 存储；参见 4.1.8），可通过 TON 网络访问，被 TON 区块链用于存储区块和状态数据（快照）的存档副本，也可用于存储用户或者平台上运行的其他使用流技术服务的任意文件。

- **网络代理/ 匿名层**（TON Proxy 或称 TON 代理；参见 4.1.11 和 3.1.6），类似于 I^2P （Invisible Internet Project；隐形网计划）用于在必要时隐藏 TON 网络节点的身份和 IP 地址（例如，包含大量加密货币的账户进行交易所用的节点，或者 stake 了巨量代币的验证节点希望隐藏其确切的 IP 地址和地理位置防止 DDoS 攻击。

- **类似 Kademlia 的分布式哈希表**（TON DHT；参见 3.2），用作 TON 存储的“流追踪器”（torrent tracker；参见 3.2.10）或者用做 TON 代理的“输入隧道定位器”（input tunnel locator；参见 3.2.14 ），并作为 TON 服务的服务定位器（参见 3.2.12）。

- **支持任意服务的平台**（TON Services 或称 TON 服务；参见第 4 章），可以通过 TON 网络和 TON 代理访问，类似浏览器或智能手机应用程序可以与统一的形式化接口交互（参见 4.3.14）。这些形式化接口可以在 TON 区块链中发布（参见 4.3.17）；在 TON 区块链中发布的信息（参见 3.2.12），可以通过 TON DHT 查找在任何给定时刻提供服务的实际节点。服务可以通过 TON 区块链中创建智能合约的方式来进行保证（参见 4.1.7）。

- **TON DNS**（参见 4.3.1），一种为帐户，智能合约，服务和网络节点分配可读性强的名称的服务。

- **TON Payments**（参见第 5 章），小额支付，小额支付通道和小额支付通道网络的平台。它可用于快速的链下价值交换，将付费给由 TON Services 支持的服务。

- TON 将允许并很容易把第三方的通信与社交网络 APP 集成，从而使区块链技术和分布式服务普及到普通用户（参见 4.3.24），而不仅仅供少数早期的加密货币使用。我们将在另一个项目 Telegram Messenger 中提供这种集成的示例（参见 4.3.19）。
- 虽然 TON Blockchain 是 TON Project 的核心，而其他组件可能被视为 TON Blockchain 的支持角色，但它们本身就具有有用且有趣的功能。结合起来，它们允许平台承载比仅使用 TON Blockchain（参见 2.9.13 和 4.1）更多样化的应用程序。

2 TON Blockchain

我们首先介绍 Telegram Open Network（TON）Blockchain，这是项目的核心组成部分。白皮书打算“自上而下”：首先给出整体项目的描述，然后提供每个组件的更多细节。

为简单起见，我们在这里谈论 TON Blockchain，尽管原则上该区块链协议的几个实例可以独立运行。

2.1 TON Blockchain 作为 2-Blochain 的集合及定义

TON Blockchain 实际上是区块链的集合（甚至是“区块链中的区块链”的集合，或者说是 2D 区块链；这将在后面的 2.1.17 中阐明），因为没有哪个单链区块链项目可以实现百万 TPS 交易吞吐量，它们仅能达到每秒数十次的交易速度。

2.1.1 TON 区块链类型列表。包含以下这几类区块链：

- 独特的 master blockchain，或简称主链（masterchain），用来存储协议、基础参数的设置等基本信息；还包含验证者和它们的 stake 记录；用户的资产；当前活跃的工作链（workchains）和它们的 “分片”（shards），；及最重要的是所有工作链（workchains）和分片链（shardchains）里的最近生成的 Hash 值。

- 主要的 work blockchains（最多 2^{32}），或简称 workchains（工作链），功能包含价值交换和智能合约。不同的工作链承担不同的角色，意味着不同格式的帐户地址，不同的交易格式，不同的智能合约虚拟机（VMs），甚至不同的加密货币结算方式都有不同的工作链来处理。但它们都必须满足统一的基本交互标准，以使不同工作链之间的交互成为可能且相对简单。在这方面，TON Blockchain 是异构的（heterogeneous；参见 2.8.8），类似于 EOS（参见 2.9.7）和 PolkaDot（参见 2.9.8）。

- 工作链（workchains）又被细分为最多 2^{64} 次方个分片链（shard blockchains；或简称为 shardchains），分片链继承工作链的生成规则和区块结构，一个工作链负责一种类型的帐户子集，具体取决于其帐户地址的第一个字节（一般情况下）。换句话说，分片链是在系统中规定的（参见 2.8.12）。因为所有这些分片链具有统一的生成规则和区块结构，所以 TON 区块链是同构的（homogeneous；参见 2.8.8），这也与以太坊之前的 “分片” 扩容方案类似。

- 上的每个区块（其实主链上也是）不仅仅只是一个区块，它可能是一个小型区块链。通常情况下，这种 “区块的区块链” 或者 “垂直区块链” 都只包含一个块，然后我们可能会认为这只是 shardchain 的相应块（在这种情况下也称为 “水平区块链”）。但是，如果有必要修复不正确的 shardchain 块，则会将新块提交到 “垂直区块链” 中，其中包含无效 “水平区块链” 块的替换或 “不同的块”，仅包含对这些块的描述需要更改此块的先前版本的部分内容。这是一种特定于 TON 的机制，用于替换检测到的无效块，而不

会涉及所有分支链的真正分支；它将在 2.1.17 中更详细地解释。现在，我们只是说每个 shardchain （和主链）不是传统的区块链，而是链中链或 blockchain 中的 blockchain，或者只是一个 2-blockchain。

2.1.2 无限分片范式（Infinite Sharding Paradigm）。

几乎所有的区块链分片提议都是 “自上而下” 的：首先想象一个单链区块链，然后讨论如何将其拆分成几个交互的分片链，以提高性能并实现可扩展性。

TON 的分片方法是 “自下而上”，解释如下。

想象一下，分片已经发挥到极致，因此每个 shardchain 中只保留一个帐户或智能合约。然后我们有大量的 “account- chains（账户链）”，每个账户链描述一个账户的状态和状态转换，并相互发送价值信息以传输价值和信息。

当然，有数以亿计的区块链是不切实际的，其中每个区块链通常很少出现更新（即新区块）。为了更有效地实现它们，我们将这些 “account- chains（账户链）” 分组为 “shardchains”，以便 shardchain 的每个块基本上都是已分配给此分片的帐户链块的集合。因此，“帐户链” 在 “shardchains” 中仅具有纯粹的虚拟或逻辑存在。

我们将这种观点称为无限分片范式。它解释了 TON 区块链的许多设计初衷。

2.1.3 消息。

即时超立方体路由（Instant Hypercube Routing）。Infinite Sharding Paradigm 将我们每个帐户（或智能合约）视为自己的 shardchain。那么 A 帐户可能影响 B 帐户状态的唯一方法就是向它发送一条消息（这是所谓的 Actor 模型的特殊实例，帐户为 Actors ；参见 2.4.2）。因此帐户（和分片链之间的消息系统，因为源和目标帐户通常在不同的分片链中）对于可扩展系统（如 TON Blockchain）是至关重要的。实际上 TON Blockchain 的这个新功能，我们称为即时超立方体路由（Instant Hypercube Routing；参见 2.4.20），使其能够将在一个 shardchain 块中创建的消息传递和处理到目标 shardchain 的下一个块中，而不用管系统中 shardchains 的总数。

2.1.4 主链，工作链和分片链的数量。

TON Blockchain 只有一个主链。但该系统最多可容纳2^{32}个工作链，每个工作链最多可细分成=2^{60}个分片链。

2.15 工作链可以是虚拟区块链，而不是真正的区块链。

因为工作链通常被细分为分片链，所以工作链的存在是“虚拟的”，不是真正的区块链，这意味着它不是下面 2.2.1 中提供的一般意义上的真正区块链，而只是一个 shardchains 的集合。当只有一个 shardchain 对应一个工作链时，这个独特的 shardchain 可以用工作链识别，在这种情况下，工作链成为“真正的”区块链，至少在一段时间内，它与传统单链区块链的设计非常趋同。然而无限分片

范式（参见 2.1.2）告诉我们，这种相似性只是表面：重要的是潜在大量的“account- chains（账户链）”可以组成一个区块链。

2.1.6 工作链的识别。

每个工作链都有序号或工作链的专属标识符（*workchain_id*：$uint_{32}$），这是一个无符号的 32 位整数。工作链由主链中的特殊事件创建，定义（先前未使用过的）工作链的标识符和工作链的描述，使该工作链与其他工作链可以交互及验证。

2.1.7 创建和激活新的工作链。

基本上任何社区成员都可以创建新的工作链，只需支付发布新工作链所需的（高）主链交易费用。但为了使新的工作链更积极，它需要得到三分之二验证者的共识，因为他们也需要升级软件来处理新的工作链的块，并通过一笔特殊交易来表明它们已经准好用新的工作链了。对激活新工作链感兴趣的一方可能会为验证者提供一些激励，通过智能合约分发的一些奖励来支持新的工作链。

2.1.8 鉴定分片链。

每个 shardchain 由一对（w，s）=（*workchainid*，*shardprefix*）标识，其中 *workchain_id*：$uint_{32}$标识相应的工作链，*shard_prefix*：$2^{(0\ldots60)}$是一个长度最多为 2 的 60 次方的字符串，来定义此分片链负责的帐户子集。即，所有具有以 *shard_prefix* 开头的 *account_id* 的帐户（即将 *shard_prefix* 作为最高有效位）将被分配给该 shardchain。

2.1.9 识别帐户链。

回想一下，“*account- chains*（账户链）”只是虚拟存在（参见 2.1.2）。但是，它们有一个自然的标识符——即（*workchainid*，*account_id*），因为任何帐户链里都包含有关状态信息和与客户账户有关的智能合约更新（简单帐户或智能合约——在这里的区别并不重要）。

2.1.10 shardchains 的动态拆分和合并（参照 2.7）。

不那么复杂的系统可以使用静态分片——例如，通过使用 *account_id* 的前八位来选择 256 个预定义分片中的一个。

TON 区块链的一个重要特征是它实现了动态分片，这意味着分片数量不固定。相反，如果满足某些条件，则分片（w，s）可以自动细分为分片（w，$s.0$）和（w，$s.1$）（实际上如果原始分片负载长时间过高/拥堵时间过长，则延长）。反过来如果负载在一段时间内都很低，则分片（w，$s.0$）和（w，$s.1$）可以自动合并回分片（w，s）。也就是说一开始只为工作链 w 创建了一个分片（w，$\emptyset$）。之后，如果有必要，它会细分为更多分片（参见 2.7.6 和 2.7.8）。

2.1.11 Basic workchain 或者 Workchain Zero。

虽然使用特定的规则和事务可以定义多达 2^{32} 工作链，但我们最初只定义了一个工作链，其中 *workchain_id = 0*。这个称为 Workchain Zero 或 Basic workchain 的工作链是用于处理 TON 智能合约和转移 *TON coins* 的工作链，也称为 *Grams*。大多数应用程序只需要 Workchain Zero。 Basic workchain 的 shardchains 将被称为 basic shardchains。

2.1.12 出块时间。我们期望每五秒在 shardchain 和 masterchain 中生成一个新块。

这将会有非常短的交易确认时间。所有 shardchain 的新块几乎同时生成；而 masterchain 将在大约一秒后生成一个块，因为它包含所有 shardchains 出的最新块的哈希值。

2.1.13 通过 masterchain 使 workchains 和 shardchains 紧密耦合。

一旦 shardchain 出的块的哈希值被合并到主链的块中，这个 shardchain 的块和它的父块都会被认为是 “一致的”，这意味着之后所有 shardchains 出的块都会引用它。实际上，每个新的 shardchain 块都包含最新 masterchain 的哈希值，并且从该 masterchain 块引用的所有 shardchain 块被新块视为不可变的。

也就是说，这意味着在 shardchain 块中提交的事件或消息可以安全地用在其他 shardchains 的下一个块中，而不需要等待。比如你在发送交易或采取其他行动之前，距离上一笔交易需要系统的 20 个确认（即，在原始块之后生成的 20 个块），这在大多数 “松散耦合” 系统（参见 2.8.14）里是常见的，例如 EOS。而我们预计你提交后仅仅五秒就后能在其他分片链中发送事件和消息，这种能力是我们认为我们的 “紧密耦合” 系统能够提供前所未有的性能的原因之一（参见 2.8.12 和 2.8.14）。

2.1.14 Masterchain （主链）出的块的哈希是一个全局量。

根据 2.1.13，整体来看，最后一个主链块的哈希值确定了系统的全局状态。 而无需分别监视其他 shardchains 的状态。

2.1.15 验证者（Validators）出新块；参照 2.6。

TON 区块链使用 Proof-of-Stake（PoS）在 shardchains 和 masterchain 中生成新块。这意味着会存在几百个验证者（数量待定）——而出块节点需要抵达代币（TON coins），才有资格成为验证者和生成新块。

然后，以确定的伪随机方式将小部分验证者分配给每个分片 (w, s)，大约每 1024 个块改变一次。这些验证者通过从客户端收集合适的建议事务到新的有效块候选者，并建议就下一个 shardchain 块的内容达成共识。对于每个块，验证者都存在伪随机选择的顺序，以确定待出块的优先级。

其他验证者和节点会检查提议的块的有效性；如果验证者签署了无效区，则会收到部分或全部 staketoken 的自动惩罚，或者在一段时间内不许继续做验证

者。在此之后，验证者应该就下一个区块的选择达成共识，主要通过 BFT（拜占庭容错；参见 2.8.4）的优化版，类似于 pBFT 或者 Honey Badger BFT。如果达成共识，则创建新块，验证者会获得该块的交易费用和挖矿奖励。

每个验证者可以选择参与由其他几个验证者组成的组合；这种情况下，所有验证和共识算法都会同时运行。

新的 shardchain 块生成或出块超时后，masterchain 都将会生成新的块，它包括所有 shardchains 最新块的哈希值。这是由验证者通过 BFT 共识完成的。

有关 TON PoS 的方法及其经济模型的更多详细信息，请参见第 2.6 节。

2.1.16 masterchain （主链）分叉。

由于我们是紧密耦合，它带来的复杂性是，主链如果分叉，那么一定会有一些分片链跟随分叉。

另一方面，只要 masterchain 中没有分叉，shardchain 就不可能分叉，因为 shardchain 的分叉链中的任何块都不能通过将它们组合哈希而与 masterchain 块保持“一致性”。

规则是，如果 masterchain 上的块B'是 B 的上一个区块，则B'的 Hash 包括分片链 Hash（$B_{w,s}^`$）；B 的 Hash 包括分片链（$B_{w,s}$）；那么B'必须是 B 的上一个区块，否则将无效。

我们预计 masterchain 几乎不会分叉，因为在 TON Blockchain 采用的 BFT 共识，只有在大多数验证者行为不正确的情况下才可能发生这种情况（参见 2.6.1 和 2.6.15），这将意味着违法者将遭受重大惩罚。 因此可以遇见 shardchains 中不会有真正的分叉。 相反如果检测到无效的 shardchain 块，它会通过 2-blockchain 的“垂直区块链”机制（参见 2.1.17）进行纠正就可以实现这一目标，而不会出现分叉的“水平区块链”（即 shardchain）。 同样的机制也可用于修复 masterchain 块中的非致命错误。

2.1.17 更正无效的 shardchain 块。

通常情况下，出块节点只能只提交有效的 shardchain 块，因为分配给 shardchain 的验证者必须满足三分之二的拜占庭共识才能提交新块。但在此之前，系统必须先检测之前提交的无效块并校正。

当然，一旦找到无效的 shardchain 块——由验证者（不一定是负责这个 shardchain 的）或 “渔夫”（fisherman；系统内的一种节点，可以 stake 然后提出有关块有效性的问题；参见 2.6.4）——当无效索赔及其证据被提交到主链时，签署无效区块的验证者将因失去部分 stake 或暂停作为出块节点而受到惩罚（后一种措施面对攻击者窃取其他良性验证者的私钥签名这种情况时很重要）。但这还不够，因为系统的整体状态（TON Blockchain）由于先前提交的 shardchain 块无效而证明是无效的。必须使用较新的有效版本替换此无效块。

大多数系统都会 “回滚” 至 shardchain 块的无效块之前最后一个块来实现此目的，最后一个块不受从其他每个 shardchains 中的无效块传播的消息的影响。并

从这些块开始分叉。这种方法的缺点是大量的其他正确和提交的事务突然回滚，并且不清楚它们是否稍后将被包括在内。

TON 区块链通过使每个 shardchain 和 masterchain（“水平区块链”）的每个“block” 自身成为一个小区块链（“垂直区块链”）来解决这个问题，包含不同版本的 “block” 或它们的 “差异””。通常，垂直区块链只包含一个区块，而 shardchain 看起来像一个经典的区块链。然而，一旦确认块的无效并将其提交到主链块中，则允许无效块的 “垂直区块链” 在垂直方向上由新块增长，替换或编辑无效块。新块由相关 shardchain 的当前验证者组合来生成。

新的 “垂直” 块有效的规则非常严格。特别是如果无效块中包含的虚拟 “帐户链块”（参见 2.1.2）本身有效，则必须保持新的垂直块不变。一旦在无效块之上提交了新的 “垂直” 块，其哈希就会发布在新的主链块中（或者更确切地说是最初发布的位于原始主链块上方的新 “垂直” 块中其中无效的 shardchain 块的哈希值），并且更改被进一步传播到指向该块的先前版本的任何 shardchain 块（例如，那些已从不正确的块接收消息的那些）。这是通过在垂直区块链中为以前引用 “不正确” 区块的所有区块提交新的 “垂直” 区块来解决的；新的垂直块将引用最新（已更正）的版本。同样，严格的规则禁止更改未受影响的帐户链（即，接收与先前版本相同的消息）。通过这种方式，修复不正确的块会产生 “涟漪”，最终传播到所有受影响的分片链的最新块；这些变化反映在新的 “垂直”masterchain 块也是如此。

一旦 “历史重写” 的涟漪到达最新的块，新的 shardchain 块仅在一个版本中生成，仅作为最近块版本的继承者。这意味着它们将从一开始就包含对正确（最近）垂直块的引用。

主链状态隐含地定义了将每个 “垂直” 区块链的第一个块的哈希值转换为其最新版本的哈希值的映射。这使客户能够验证和定位任何 “垂直区块链” 它的第一个（通常是唯一的）块的哈希值。

2.1.18 TON coins 和 multi-currency workchains（多货币工作链）。

TON Blockchain 最多支持 2^{32} 种不同的“cryptocurrencies”, “coins”, 或者 “tokens”，由 32 个字节的 *currency_id* 作为区分。可以通过 masterchain 中的特殊事务添加新的加密货币。 每个工作链都有一个基础的加密货币，并且可以有几个额外的加密货币。

有一种特殊的加密货币，*currency_id = 0*，即 TON coin，也被称为 $Gram$。它是 Workchain Zero 的基本加密货币。它还用于交易费和验证者的权益证明。

原则上，其他工作链可能会收取其他代币作为交易费用。这种情况，应该提供一些智能合约，将这些交易费用自动转换为 Grams。

2.1.19 消息的传递与价值转移。

属于相同或不同工作链的分片链可以相互发送消息。虽然允许的消息的格式取决于接收工作链和接收帐户（智能合约），但有一些共同的字段让工作链间消

息传递成为可能。特别是每个事件可以附加一些价值，以一定数量的 *Grams*（TON coin）或其他被接收工作链声明为可接受的加密货币。

这种消息传递的最简单形式是从一个（通常不是智能合约）账户到另一个账户的价值转移。

2.1.20 TON 虚拟机。

TON 虚拟机（也称为 TON VM 或 TVM）是用于在 masterchain 和 basic workchain 中执行智能合约代码的虚拟机。其他 workchains 可以与 TVM 一起使用其他虚拟机或代替 TVM。在这里我们列出了它的一些功能。它们将在 2.3.12、2.3.14 中进一步讨论。

- TVM 将所有数据表示为（TVM）cells（细胞）的集合（参见 2.3.14）。每个 cell 最多包含 128 字节的数据，最多包含 4 个对其他单元的引用。由于“一切都是一袋细胞” 的理念（参见 2.5.14），这使得 TVM 能够处理与 TON Blockchain 相关的所有数据，包括块和区块链全局状态（如有必要）。
- TVM 可以处理任意代数数据类型的值（参见 2.3.12），以 TVM 细胞里的 Merkle trees 或 DAG 为代表。但它只对 cells 有效，对代数数据类型未知。
- TVM 内置了对哈希图的支持（参见 2.3.7）。
- TVM 是一台堆栈机器。 它的堆栈保持 64 位整数或 cell 的引用。
- 支持 64 位，128 位和 256 位算术。 所有 n 位算术运算有三种形式：无符号整数，有符号整数和 2^n 整数（后一种情况下不进行自动溢出检测）。
- TVM 具有从 n 位到 m 位的无符号和有符号整数转换，对于所有 $0 \le m$，$n \le 256$，具有溢出检查。
- 默认情况下，所有算术运算都执行溢出检查，大大简化了智能合约的开发。
- TVM 具有 “乘法 - 后移” 和 “移位 - 后除” 算术运算，其中间值以较大的整数类型计算； 这简化了实现定点算术的过程。
- TVM 支持二进制位串和字节字符串。
- 支持对一些预定义曲线（包括 Curve25519）的 256 位椭圆曲线加密（ECC）。
- 还存在对某些椭圆曲线上的 Weil 配对的支持，这有利于快速快速实现 zk-SNARK。
- 支持流行的哈希函数，包括 SHA256
- TVM 可以 Merkle 证明（参见 5.1.9）。

- TVM 为 “大型” 或 “全局” 的智能合约提供支持。 这种智能合约必须意识到分片（参见 2.3.18 和 2.3.16）。 而本地智能合约不用知道是否分片。
- TVM 支持闭包。
- 可在 TVM 内轻松实现 “spineless tagless G-machine（避免创建图计算机上的中间节点）”。

除了 “TVM 组件” 之外，还可以为 TVM 设计几种高级语言。 所有这些语言都将具有静态类型，并将支持代数数据类型。 我们设想了以下可能性：

- 类似 Java 的命令式语言，每个智能合约类似于一个单独的类。
- 一种懒惰的函数式语言（想想 Haskell）。
- 热切的功能语言（想想 ML）。

2.1.21 可配置的参数。

TON Blockchain 的一个重要特征是它的许多参数都是可配置的。 这意味着它们是 masterchain 状态的一部分，并且可以通过 masterchain 中的某些特殊提案/投票/结果事务进行更改，而无需任何硬分叉。 改变这些参数需要收集三分之二的验证者投票和所有其他参与投票过程以支持该提案的参与者的一半以上的投票。

2.2 区块链概述

2.2.1 一般区块链定义。

一般来说，任何（真）区块链都是一个区块序列，每个区块 B 包含上一个区块的 blk-prev(B) 的引用（通常通过将上一个区块的哈希值包含到当前区块的区块头）和交易列表。每笔交易都描述了全局区块链状态的一些转换；按顺序应用区块中列出的交易来计算从旧状态开始的新状态，这是前一个区块出来之后的状态。

2.2.2 TON Blockchain 的相关性。

回想一下，TON Blockchain 不是真正的区块链，而是 2D blockchain 的集合（即区块链的区块链；参见 2.1.1），因此上述内容并不直接适用于它。 但我们从真正的区块链的这些一般性开始，将它们用作我们更复杂结构的构建块。

2.2.3 区块链实例和区块链字段。

人们经常使用区块链这个词来表示一般的区块链字段及其特定的区块链实例，它们被定义为满足某些条件的区块序列。 例如，2.2.1 指的是区块链实例。

以这种方式，区块链类型通常是块的列表（即，有限序列）类型 $Block^*$的“子类型”，由满足某些兼容性和有效性条件的那些块序列组成：

$$Blockchain \subset Block^{*} \qquad (1)$$

定义区块链的更好方法是说定义区块链的更好方法是说 $Blockchainisadependentcoupletype$，由 $consistingofcouples（B，v）$组成，第一个组成部分 B:$Block^{*}$是类型（$Block^{*}$（即一列块），第二个组成部分 v:$isValidBc(B)$ 是 B 的有效性的证明或见证。这样显示，

$$Blockchain \equiv \sum_{(B:Block*)} i\,sValidBc(B) \qquad (2)$$

2.2.3 依赖型理论（Dependent type theory），Coq and TL。

请注意，我们在这里使用（Martin-Löf）依赖型理论，类似于 Coq3 证明助手中使用的理论。依赖类型理论的简化版本也用于 TL（类型语言；Telegram 定义的一种语言），将在 TON Blockchain 的正式规范中使用它来描述所有数据结构的序列化以及块，事务等的布局。

实际上，依赖类型理论（Dependent type theory）给出了一个可用的形式化证明，它是如何证明（或者序列），比如需要为某些区块提供无效证明时，这会变得很方便。

2.2.5 TL 或者叫 Type Language。

由于 TL（类型语言）将用于 TON blocks，事务和网络数据报的正式规范，因此需要进行简短的讨论。

TL 是一种适用于从属代数类型描述的语言，允许使用数字（自然）和类型参数。通过几个构造函数描述每种类型。每个构造函数都有一个（人类可读的）标识符和一个名称，它是一个位串（默认情况下为 32 位整数）。除此之外，构造函数的定义包含字段列表及其类型。

构造函数和类型定义的集合称为 $TL-scheme$。它通常保存在一个或多个带有后缀 .tl 的文件中。

$TL-scheme$ 的一个重要特征是它们确定了一种明确的序列化和反序列化定义的代数类型值（或对象）的方法。也就是说，当需要将值序列化为字节流时，首先序列化用于此值的构造函数的名称。递归计算每个字段的序列化如下。适用于将任意对象序列化为 32 位整数序列的 TL 的先前版本的描述可在以下位置获得 https://core.Teleggram.org/MTProto/TL 。正在开发一种名为 TL-B 的 TL 新版本，用于描述 TON 项目使用的对象的序列化。这个新版本可以将对象序列化为字节流甚至比特流（不仅仅是 32 位整数），并且支持将序列化到 TVM cells 中（参见 2.3.14）。TL-B 的描述将是 TON 区块链的正式规范的一部分。

2.2.6 区块和事务作为状态转换运算符。

通常，任何类型的区块链 *Blockchain* 都具有关联的全局状态（类型）*State* 和事务（类型）*Transaction*。 区块链的语义在很大程度上取决于事务应用程序功能：

$$ev - trans' : Transaction \times State \rightarrow State^{?} \qquad (3)$$

Here $X^{?}$ denotes $MAYBEX$，the result of applying the MAYBE monad to type *X*. This is similar to our use of X^{*} for *LISTX*. Essentially, a value of type $X^{?}$ either a value of type *X* or a special value ⊥ indicating the absence of an actual value (think about a null pointer). In our case, we use $State^{?}$ instead of *State* as the result type because a transaction may be invalid if invoked from certain original states (think about attempting to withdraw from an account more money than it is actually there).

我们可能更喜欢 *ev_trans'* 的这个版本：

$$ev - trans : Transaction \rightarrow State \rightarrow Stat^{?} \qquad (4)$$

因为一个块本质上是一个事务列表，所以块是评价函数

$$ev - block : Block \rightarrow State \rightarrow State^{?} \qquad (5)$$

可以从 ev*trans 派生。 它需要一个块 B：Block 和前一个区块链状态 s：State（可能包括前一个块的哈希值）并计算下一个区块链状态 $s'* = *ev*block(B)(s) : State$，它是真实状态或特殊值⊥，表示无法计算下一个状态（即如果从给定的起始状态进行评估，则该块无效——例如，该块包括试图发起从空账户中扣款的交易）。

2.2.7 Block sequence numbers 区块序列号。

区块链中的每个块 B 可以通过 *sequence number* blk-seqno(B) 来引用，从第一个块的零开始，并且每当传递到下一个块时递增 1。 正式的公式：

$$blk - seqno(B) = blk - seqnoblk - prev(B) + 1 \qquad (6)$$

请注意，序列号在有分叉存在时会出现多个。

2.2.8 Block hashes 区块哈希。

引用区块 B 的另一种方式是通过其哈希值 *blk-seqno(B)*，其实际上是区块 B 块头的哈希值（然而区块的头部通常包含区块 B 的所有内容的哈希值）。假设所

使用的哈希函数没有冲突（或者至少它们是非常不可能的），则通过其哈希值唯一地标识区块。

2.2.9 Hash assumption 哈希假设。

在区块链算法的形式分析期间，我们假设 k-bit 哈希函数 HASH 没有冲突：

$Bytes^* \to 2^k$ used：

$$Hash(s) = Hash(s') \Rightarrow s = s' for any s, s' \in Bytes^* \qquad (7)$$

Here $Bytes$ = {0...255} = 2^8 次方个字节类型，或者是所有字节值的集合，$Bytes^*$ 是任意（有限）字节列表的类型或集合； 而 2 = {0,1} 是位类型，2^K 次方是所有 k-bit 的序列（即 k-bit 的编号）的集合（或实际有的类型）。

当然，（7）在数学上是不可能的，因为从无限集到有限集的映射不能是单射的。 一个更严格的假设是：

$$\forall s, s' : s \neq s', P(Hash(s) = Hash(s')) = 2^{-k} \qquad (8)$$

但这对于证明来说并不方便。 对于某些小 ε（例如，ε=10^{-18}），如果（8）在 2^{-k}N<ε 的证明中最多使用 N 次，我们可以推理如果（7）为真，只要我们接受失败概率 ε（即最终结论将是真实的，概率至少为 1-ε）。

最后评论：为了使（8）的概率陈述真正严格，必须在所有字节序列的设置 $Bytes^*$ 上引入概率分布。 这样做的一种方法是假设所有相同长度的字节序列是等概率的，并且设置观察长度 l 的序列的概率等于某些 p→1- 的 $p^l - pl + 1$ 。然后（8）应该被理解为条件概率的极限 $PHash(s) = Hash(s') | s \neq s'$ 当 p 从趋近于 1 时。

2.2.10 Hash 作用于 TON Blockchain。

我们暂时在 TON Blockchain 中使用 256 位 SHA256 哈希函数。 如果结果比预期的要差，那么将来可以用另一个哈希函数来替换它。 哈希函数的选择是协议的可配置参数，因此可以在不硬分叉的情况下进行更改，如 2.1.21 中所述。

2.3 区块链状态，账户和哈希表

我们在上面已经注意到，全局状态由每一个区块链来决定，每个块和每个事务都定义了全局状态的变化。这里我们描述 TON 区块链使用的全局状态。

2.3.1 帐户 ID。

TON 区块链使用的基本帐户 ID——或者至少是其 mastercvhain 和 Workchain Zero——使用的基本帐户 ID 是 256 位整数，假定为特定椭圆曲线的 256 位椭圆曲线加密（ECC）的公钥。将通过这种方式，

$$account - id: Account = uint^{256} = 2^{256} \qquad (9)$$

此处 Account 是帐户类型，而 account_id 是帐户类型的特定变量。

其他工作链可以使用其他帐户 ID 格式，256 位或其他。 例如，可以使用比特币风格的帐户 ID，等于 ECC 公钥的 SHA256。

但是，在创建工作链（在主链中）期间必须修复帐户 ID 的位 l ，并且它必须至少为 64，因为 account_id 的前 64 位用于分片和消息路由。

2.3.2 主要组成部分：Hashmaps（哈希表）。

TON Blockchain 状态的主要组成部分是哈希表。 在某些情况下，我们考虑（部分定义）“map”h：2^n-> 2^m ，或者通俗地说，我们可能对 hashmaps h 更感兴趣：2^n ->X 表示复合型 X。但是源（或索引）类型几乎总是 2^n。有时，我们的“default value”为空：X, and the hashmap h : $2^n \rightarrow$ X is “initialized” by its “default value” $i \rightarrow empty$.

2.3.3 示例：TON 帐户余额。

TON 账户余额给出了一个重要的例子。 这是一个 hashmap

$$balance: Account \rightarrow uint^{128} \qquad (10)$$

mapping Account = 2^{256} into a Gram (TON coin) balance of type $uint_{128}$ = 2^{128}. 此哈希表的默认值为零，这意味着最初（在处理第一个块之前）所有帐户的余额为零。

2.3.4 示例：智能合约的永久存储。

另一个例子是智能合约永久存储，它可以（非常近似地）表示为哈希表：

$$storage: 2^{256} \rightarrow 2^{256} \qquad (11)$$

此哈希表的默认值也为零，假设未初始化的永久存储 cells 个数为零。

2.3.5 示例：永久存储所有的智能合约。

因为我们有多个智能合约，由 account_id 来区分，每个都要独立的永久存储，所以必须有一个 hashmap

$$Storage: Account \to (2^{256} \to 2^{256}) \qquad (12)$$

将智能合约的 account_id 映射到其永久存储里。

2.3.6 Hashmap 的类型。

Hashmap 不仅仅是一个抽象的（部分定义的）函数 $2^n \to X$； 它有一个特定的代表。 因此，我们假设我们有一个特殊的 hashmap 类型

$$Hashmap(n, X): Type \qquad (13)$$

对应于编码（部分）地图 2^n -> X 的数据结构。我们也可以写为

$$Hashmap(n: nat)(X: Type): Type \qquad (14)$$

或者

$$Hashmap: nat \to Type \to Type \qquad (15)$$

我们也可以翻译为 $h: Hashmap(n, X)$ into a map *hget(h)* : $2^n \to X$?
之后，我们通常写 h[i] 以替代 hget(h)(i)：

$$h[i]: \equiv hget(h)(i): X^? foranyi: 2^n, h: Hashmap(n, X) \qquad (16)$$

2.3.7 把哈希表的类型当作 Patricia 树。

从逻辑上讲，可以将 Hashmap（n，X）定义为深度为 n 的（不完整）binary tree，边缘标签为 0 和 1，叶片中的值为 X。描述相同结构的另一种方式是对于长度等于 n 的二进制串的（按位）trie。

在实践中，我们更喜欢使用此 trie 的紧凑表示，通过压缩每个顶点只有一个子节点与其父节点。 结果我们称为 Patricia 树或者二进制基数树。

换句话说，Patricia 树中有两种类型的（非根）节点：

- Leaf(x), containing value x of type X.

- Node(l, sl, r, sr), where l is the (reference to the) left child or subtree, sl is the bitstring labeling the edge connecting this vertex to its left child (always beginning with 0), r is the right subtree, and sr is the bitstring labeling the edge to the right child (always beginning with 1).

第三种类型的节点，仅在 Patricia 树的根部使用一次，也是必要的：

- Root(*n,s0,t*), where n is the common length of index bitstrings of Hashmap(*n,X*), s0 is the common prefix of all index bitstrings, and t is a reference to a Leaf or a Node.

如果我们想让 Patricia 树为空，将使用第四种类型的（根）节点：

- EmptyRoot(n)，其中 n 是所有索引位串的公共长度。 我们定义 Patricia 树的高度

- $$height(Leaf(x)) = 0 \qquad (17)$$

$$height(Node(l, sl, r, sr)) = height(l) + len(sl) = height(r) + len(sr) \qquad (18)$$

$$height(Root(n, s0, t)) = len(s0) + height(t) = n \qquad (19)$$

最后两个公式中每个公式中的最后两个表达式必须相等。 我们使用高度为 n 的 Patricia 树来表示 Hashmap（n，X）类型的值。

如果树中有 N 个分支（即我们的哈希表包含 N 个值），则恰好有 N-1 个中间顶点。 插入新值总是涉及通过在中间插入新顶点并添加新叶作为此新顶点的另一个子来分割现有边。 从哈希表中删除值则相反：删除了一个分支及其父级，并且父级的父级和另一个子级直接链接。

2.3.8 Merkle-Patricia 树。

使用区块链时，我们希望能够比较 Patricia 树（即哈希表）及其子树，方法是将它们减少为单个哈希值。 Merkle 树给出了实现这一目标的经典方法。 本质上，我们想要描述一种哈希 Hashmap(*n*，*X*)类型的对象 *h* 的方法，借助为二进制字符串定义的哈希函数 Hash，前提是我们知道如何计算对象的哈希 Hash（x）x：X（ 例如，通过将哈希函数 Hash 应用于对象 x）的二进制序列化。

可以递归地定义 Hash（h），如下所示：

$$Hash(Leaf(x)) := Hash(x) \qquad (20)$$

$$Hash(Node(l, sl, r, sr)) := Hash(Hash(l).Hash(r).code(sl).code(sr)) \qquad (21)$$

$$Hash(Root(n, s0, t)) := Hash(code(n).code(s0).Hash(t)) \qquad (22)$$

这里 s 和 t 表示（位）字符串 s 和 t 的串联，并且代码（s）是所有位串 s 的前缀代码。

稍后我们将看到（参见 2.3.12 和 2.3.14）这是一个（稍微调整过的）版本的递归定义的哈希值，用于任意（从属）类型的值。

2.3.9 重新计算 Merkle 树的哈希值。

这种递归定义 Hash（h）的方式称为 Merkle 树哈希，其优点是，如果一个人明确地将 Hash（h'）与每个节点 h' 一起存储（产生一个称为 Merkle 树的结构，或者，在我们的例子中，一个 Merkle-Patricia 树），当在 hashmap 中添加，删除或更改元素时，只需要重新计算最多 n 个哈希值。

这样，如果通过合适的 Merkle 树哈希表示全局区块链状态，则在每次事务后很容易重新计算此状态哈希。

2.3.10 Merkle 证明。

在所选哈希函数 Hash 的 “注入” 假设（7）下，可以构造一个证明，对于 Hash（h）的给定值 z，h：Hashmap（n，X），一个具有 hget（h）（i）= x 对于某些 i:2^n 和 x:X。这样的证明将包括 Merkle-Patricia 树中从对应于 i 到根的分支中的路径，由所有节点的所有兄弟的哈希增强在这条道路上。

以这种方式，仅知道一些哈希表 h 的 Hash（h）的值的轻节点（例如，智能合约永久存储或全局区块链状态）可以从完整节点请求不仅值 x = h [i] = hget（h）（i），但这样的值以及从已知值 Hash（h）开始的 Merkle 证明。然后，在假设（7）下，工作节点可以检查自身 x 确实是 h [i] 的正确值。

在一些情况下，客户端可能想要获得值 y = Hash（x）= Hash（h [i]）而不是——例如，如果 x 本身非常大（例如，哈希表本身）。然后可以提供（i，y）的 Merkle 证明。如果 x 也是哈希函数，则可以从完整节点获得从 y = Hash（x）开始的第二 Merkle 证明，以提供值 x[j] = h [i] [j] 或仅其哈希。

2.3.11 对于诸如 TON 的多链系统，Merkle 证明的重要性。

请注意，节点通常不能是 TON 环境中存在的所有分片链的完整节点。 它通常只包含一些完整的节点——例如，那些包含自己的帐户，它感兴趣的智能合约，或者这个节点被指定为验证者的那些。 对于其他 shardchains，它必须是一个轻型节点——否则存储，计算和网络带宽要求很高。

这意味着这样的节点不能直接检查关于其他 shard 链状态； 它必须依赖从完整节点获得的那些分片链的 Merkle 证明，这与自身检查一样安全，除非（7）失败（即发现哈希冲突）。获得从 y = Hash（x）开始的第二 Merkle 证明，以提供值 x[j] = h [i] [j] 或仅其哈希。

2.3.12 TON VM 的特点。

用于在主链和工作链零中运行智能合约的 TON VM 或 TVM（Telegram 虚拟机）与受 EVM（以太坊虚拟机）启发的传统设计有很大不同：它不仅适用于 256 位整数，但实际上（几乎）任意 “记录”，“结构” 或 “和产品类型”，使其更适合执行用高级（特别是功能）语言编写的代码。 从本质上讲，TVM 使用标记数据类型，与 Prolog 或 Erlang 的实现中使用的数据类型还不同。

人们可能首先想到的是，TVM 智能合约的状态不仅仅是一个 hashmap 2^{256} -> 2^{256}，或者 Hashmap(2^{256} , 2^{256})，但是（作为第一步）Hashmap（256，X），其中 X 是具有多个构造函数的类型，使其能够存储除 256 位整数之外的其他结构的数据，包括其他数据结构，特别是 Hashmap（256，X）。 这意味着 TVM（永久或临时）存储的单元——或 TVM 智能合约代码中的变量或数组元素——不仅可以包含整数，而且可以包含全新的哈希表。 当然，这意味着一个单元不仅包含 256 位，而且还包含一个 8 位标签，描述了如何解释这 256 位。

实际上，值不需要精确为 256 位。TVM 使用的值格式包括一系列原始字节和对其他结构的引用，以任意顺序混合，一些描述符字节插入适当的位置，以便能够区分指针与原始数据（例如，字符串或整数）；参照 2.3.14。

该原始值格式可用于实现任意和积——代数类型。在这种情况下，该值首先包含一个原始字节，描述正在使用的 “构造函数”（从高级语言的角度来看），然后是其他 “字段” 或 “构造函数参数”，包括原始字节和引用取决于所选择的构造函数（参见 2.2.5）。但是，TVM 对构造函数与其参数之间的对应关系一无所知；字节和引用的混合由某些描述符字节显式描述。

Merkle 树哈希扩展到任意这样的结构：为了计算这种结构的哈希，所有引用都被递归的对象哈斯替换，然后计算得到的字节串（包括描述符字节）的哈希。

通过这种方式，2.3.8 中描述的针对哈希图的 Merkle 树哈希，只是对任意（依赖）代数数据类型进行哈希的特殊情况，应用于具有两个构造函数的 Hashmap（n，X）类型。

2.3.13 永久存储 TON 智能合约。

永久存储 TON 智能合约基本上包括其在智能合约调用之间保留的 “全局变量”。 因此，它只是一个 “产品”，“元组” 或 “记录” 类型，由正确类型的字段组成，每个字段对应一个全局变量。 如果全局变量太多，则由于对 TON 单元大小的全局限制，它们不能适合一个 TON cell。 在这种情况下，它们被分成若干记录并组织成树，基本上成为 “产品的产品” 或 “产品的产品” 类型，而不仅仅是产品类型。

2.3.14 TVM cell。

最终，TON VM 将所有数据保存在（TVM）cell 的集合中。每个 cell 首先包含两个描述符字节，指示如何该单元中存在多少字节的原始数据（最多 128 个）以及存在多少个对其他单元的引用（最多四个）。然后是这些原始数据字节和

引用。每个 cell 只引用一次，因此我们可能在每个 cell 中包含对其 “父” 的引用（引用此单元的唯一 cell）。但是，此引用不必明确。

以这种方式，TON 智能合约的永久数据存储单元被组织成树 10，其中参考智能合约描述中保存的该树的根。如有必要，从分支开始递归计算整个永久存储器的 Merkle 树哈希，然后简单地用递归计算的引用单元的哈希值替换单元中的所有引用，并随后计算由此获得的字符串的哈希值。

2.3.15 任意代数类型值的广义 Merkle 证明。

因为 TON VM 通过由（TVM）单元组成的树来表示任意代数类型的值，并且每个单元具有明确定义的（递归计算的）Merkle 哈希，实际上取决于根据该单元格生成的整个子树， 我们可以为任意代数类型的（部分）值提供 “广义 Merkle 证明”，旨在证明具有已知 Merkle 哈希的树的某个子树采用特定值或具有特定哈希的值。 这概括了 2.3.10 的方法，其中只考虑了 x [i] = y 的 Merkle 证明。

2.3.16 支持 TON VM 数据结构中的分片。

我们刚刚概述了 TON VM 如何在不过度复杂的情况下支持高级智能合约语言中的任意（相关）代数数据类型。 但是，大型（或全局）智能合约的分片需要在 TON VM 提供特殊支持。 为此，系统中添加了特殊版本的 hashmap 类型，相当于 “map”Account → X。 这个 “map” 似乎等同于 Hashmap（m，X），其中 Account = 2^m。 但是，当分片分为两个或两个分片合并时，这些哈希表会自动拆分为两个或合并回来。

2.3.17 为永久存储支付。

TON 区块链的一个值得注意的特征是从智能合约中支付的用于存储其永久数据的支付（即，用于扩大区块链的总状态）。 它的工作原理如下：

每个块声明两个速率，以区块链的主要货币（通常是 Gram）提案：将一个 cell 保留在永久存储中的价格，以及在永久存储的某个单元中保留一个原始字节的价格。 每个帐户使用的单元格和字节总数的统计信息存储为其状态的一部分，因此通过将这些数字乘以块标题中声明的两个费率，我们可以计算要从帐户余额中扣除的付款以便保留它在前一个块和当前块之间的数据。

但是，对于每个帐户和每个块中的智能合约，不都会为永久存储使用付款； 相反，该支付最后被执行的块的序列号存储在帐户数据中，当该帐户进行任何动作时（例如，通过智能合约接收和处理价值转移或消息）， 自动执行任何操作之前会从帐户余额中扣除上一次此类付款以来所有块的存储费用。 如果此后帐户的余额将变为负数，则该帐户将被销毁。

workchain 可以声明每个帐户的一些原始数据字节是 “免费” 的（即，不参与永久性存储支付），以便制作 “简单” 帐户，这些帐户仅保留其在一个或两个加密货币中的余额，而不用不断付款。

请注意，如果没有人向帐户发送任何消息，则不会收集其永久存储费用，并且它可以一直存在。 但是，任何人都可以发送一条空消息来销毁这样的帐户。 从要销毁的帐户的余额的一小部分会提供给这样的消息的发送者。 我们预计验证者将乐意销毁此类无力偿债账户，只是为了减少全局区块链的规模，避免在没有补偿的情况下保留大量数据。

为保持永久性数据而收集的付款分布在 shardchain 或 masterchain 的验证者之间（与后一种情况下的质押成比例）。

2.3.18 本地和全局智能合约；智能合约实例。

智能合约通常只存在于一个分片中，根据智能合约的 *account_id* 选择，类似于“普通” 账户。对于大多数应用来说，这通常就足够了。然而，一些 “高负荷” 的智能合约可能希望在某些工作链的每个 shardchain 中都有一个 “实例”。为了实现这一点，他们必须将他们的创建事务传播到所有分片链中，例如，通过将此事务提交到工作链的 “root”shardchain（w，∅）11 并支付大额佣金。

此操作有效地在每个分片中创建智能合约的实例，并具有单独的余额。最初，在创建事务中传输的余额只是通过在 shard（*w*，*s*）中给出实例 $2^{-|s|}$来分配，作为总余额的一部分。当分片分成两个子分片时，所有全局智能合约实例的余额分成两半；当两个分片合并时，余额会加在一起。

在某些情况下，拆分/合并全局智能合约的实例可能涉及（延迟）执行这些智能合约的特殊方法。默认情况下，如上所述拆分和合并余额，并且一些特殊的 “帐户索引” 哈希图也会自动拆分和合并（参见 2.3.16）。

2.3.19 限制智能合约的分割。

全局智能合约可能会在创建时限制其拆分深度 d，以便使永久存储费用更具可预测性。 这意味着，如果 shardchain（w，s）与| s | ≥d 分为两部分，两个新的 shardchains 中只有一个继承了智能合约的一个实例。 确定性地选择此 shardchain：每个全局智能合约都有一些 “account_*id*”，它本质上是其创建事务的哈希值，并且其实例具有相同的 *account_id*，其中第一个≤d 位替换为落入正确分片所需的适当值。 此 account_id 选择在拆分后将哪个分片继承智能合约实例。

2.3.20 账户 / 智能合约状态。 我们可以总结以上所有内容，得出结论：帐户或智能合约状态包含以下内容：

- 区块链主要货币的余额
- 区块链的其他货币余额
- 智能合约代码（或其哈希）
- 智能合约永久性数据（或其 Merkle 哈希）
- 统计永久存储单元数和使用的原始字节数

- 收集智能合约永久存储的付款时的最后一次（实际上是主链块号）
- 传输货币并从此帐户发送所需的公钥（可选；默认情况下等于 *account_id* 本身）。 在某些情况下，可以在此处找到更复杂的签名检查代码，类似于比特币交易输出所做的； *然后 account_id* 将等同于此代码的哈希值。

我们还需要在帐户状态或其他一些帐户索引的 hashmap 中保留以下数据：

- 帐户的输出消息队列（参见 2.4.17）
- 最近发送的消息（哈希）的集合（参见 2.4.23）

并非所有这些都是每个帐户真正需要的；例如，智能合约代码仅适用于智能合约，但不适用于 “简单” 账户。 此外，虽然任何账户必须具有主要货币的非零余额（例如，基本工作链的主链和分片链的 TON coin），但其他货币的余额可能为零。 为了避免保留未使用的数据，定义了一个 sum-product 类型（取决于工作链）（在工作链的创建期间），它使用不同的标记字节（例如，TL 构造函数；参见 2.2.5）来区分不同的 “ 建设者 “使用。 最终，帐户状态本身被保存为 TVM 永久存储的 cells 集合。

2.4 分片链之间的通信

TON 区块链的一个重要组成部分是区块链之间的消息传递系统。这些区块链可以是相同工作链的 shardchains，也可以是不同工作链的 shardchains。

2.4.1 消息，帐户和交易：鸟瞰系统。

消息从一个帐户发送到另一个帐户。 每个事务包括一个接收一条消息的帐户，根据某些规则更改其状态，以及向其他帐户生成多个（可能是一个或零个）新消息。 每条消息都会生成并接收（传递）一次。

这意味着消息在系统中起着至关重要的作用，与帐户（智能合约）相当。 从无限分片范式（参见 2.1.2）的角度来看，每个帐户都在其独立的 “帐户链” 中，并且它影响其他帐户状态的唯一方法是发送消息。

2.4.2 作为流程或参与者的帐户；Actor 模型。

有人可能会将帐户（和智能合约）视为 “流程” 或 “参与者”，它们能够处理传入的消息，更改其内部状态并生成一些出站消息。 这与所谓的 Actor 模型密切相关，在 Erlang 等语言中使用（然而，Erlang 中的 actor 通常称为 “进程”）。 由于处理入站消息的结果也允许现有参与者创建新的参与者（即，智能合约），因此与 Actor 模型的对应基本上是完整的。

2.4.3 消息收件人。

任何消息都有其收件人，其特征是目标工作链标识符 w（默认情况下假定与发起的 shardchain 相同）和收件人帐户 account_id。 *account_id* 的确切格式（即字节数字）取决于 w； 但是，分片始终由其首位（最重要的）64 字节确定。

2.4.4 消息发件人。

在大多数情况下，消息具有发件人，再次由（w'，account_id'）对表示。 如果存在，则它位于消息收件人和消息值之后。 有时，发件人不重要或者是区块链之外的人（即，不是智能合约），在这种情况下，该字段不存在。

请注意，Actor 模型不要求消息具有隐式发送方。 相反，消息可以包含对应该发送请求答案的 Actor 的引用； 通常它与发件人一致。 但是，在加密货币（拜占庭）环境中的消息中具有不可伪造发送者签名字段是有用的。

2.4.5 消息值。

消息的另一个重要特征是其附加值，由源和目标工作链支持的一个或多个加密货币。 消息的值在消息接收者之后立即显示； 它本质上是（currency_id，value）对的列表。

请注意，“简单” 帐户之间的 “简单” 值转移只是空（无操作）消息，并附加了一些值。 另一方面，稍微复杂的消息体可能包含简单的文本或二进制注释（例如，关于支付的目的）。

2.4.6 外部消息，或 “来自任何地方的消息”。

一些消息 “无处不在” 地进入系统——也就是说，它们不是由位于区块链中的账户（智能合约或非智能合约）生成的。当用户想要将一些资金从她控制的账户转移到其他账户时，就会出现最典型的例子。在这种情况下，用户将 “来自任何地方的消息” 发送到她自己的帐户，请求它生成带有指定值的接收帐户的消息。如果此消息已正确签名，则她的帐户会收到该消息并生成所需的出站消息。

实际上，人们可能会将 “简单” 帐户视为具有预定义代码的智能合约的特例。这个智能合约只收到一种消息。这样的入站消息必须包含由于传递（处理）入站消息而生成的出站消息列表以及签名。智能合约检查签名，如果正确，则生成所需的消息。

当然，“不知从哪里传来的消息” 与正常信息之间存在差异，因为 “不知从哪里传来的消息” 不能承担价值，因此他们不能自己支付他们的 “Gas”（即他们的处理）。相反，他们暂时执行了一个小的 Gas limit，甚至建议包括在一个新的 shardchain 块；如果执行失败（签名不正确），则 “来自任何地方的消息” 被认为是不正确的并被丢弃。如果执行在小 Gas Limit 内没有失败，则消息可以包含在新的 shardchain 块中并完全处理，从接收者的帐户中消耗的 Gas（处理能力）的支付。 “不知从哪里传来的消息” 还可以定义一些交易费用，如要求交易费用从接收方账户扣除，用来支付给验证者 gas。

在这个意义上，“不知从哪里传来的消息” 或 “外部消息” 是来自其他区块链系统（例如，比特币和以太坊）的交易候选者。

2.4.7 记录消息，或“无处消息”。

类似地，有时可以生成特殊消息并将其路由到特定的分片链，而不是将其传递给其接收者，而是记录以便接收有关所讨论的分片的更新的任何人都可以容易地观察到。 这些记录的消息可以在用户的控制台中输出，或者在离线服务器上触发某些脚本的执行。 从这个意义上说，它们代表了“区块链超级计算机”的外部“输出”，就像“不知从哪里传来的消息”代表“区块链超级计算机”的外部“输入”一样。

2.4.8 与链下服务和外部区块链互动。

这些外部输入和输出消息可用于与链下服务和其他（外部）区块链（如比特币或以太网）进行交互。 有人可能会在 TON Block-chain 内部创建 token 或加密货币，这些货币链与比特币，Ethers 或以太网区块链中定义的任何 ERC-20 token 挂钩，并使用“不知从哪里传来的消息”和“无处不在的消息”，由脚本生成和处理驻留在某些第三方链下服务器上，以实现 TON Blockchain 与这些外部区块链之间的交互。

2.4.9 消息正文。

消息正文只是一个字节序列，其含义仅由接收工作链和 / 或智能合约确定。 对于使用 TON VM 的区块链，这可以是通过 Send（）操作自动生成的任何 TVM cell 的串行化。 简单地通过用所引用的 cell 递归地替换 TON VM cell 中的所有引用来获得这种序列化。 最终，出现一串原始字节，通常由 4 字节“消息类型”或“消息构造函数”预先设置，用于选择接收智能合约。

另一种选择是使用 TL 序列化对象（参见 2.2.5）作为消息体。 这对于不同工作链之间的通信尤其有用，也许某个工作链不使用 TON VM。

2.4.10 Gas Limit 和其他工作链 / VM 特定参数。

有时，消息需要携带有关 Gas Limit，gas price，交易费用以及依赖于接收工作链的类似值的信息，并且仅与接收工作链相关，但对于原始工作链而言并非必要。 这些参数包含在消息体中或之前，有时（取决于工作链）具有特殊的 4 字节前缀，表明它们的存在（可以通过 TL 方案定义；参见 2.2.5）。

2.4.11 创建消息：智能合约和交易。

这两个新消息的来源。 大多数消息是在智能合约执行期间（通过 TON VM 中的 Send（）操作）创建的，此时调用某个智能合约来处理传入消息。 或者，消息可能来自外部，作为“外部消息”或“不知从哪里传来的消息”（参见 2.4.6）

2.4.12 传递信息。

当消息到达包含其目标帐户的分片链时，它将“传递”到其目标帐户。 接下来会发生什么取决于工作链； 从外部的角度来看，重要的是这样的消息永远不能从这个 shardchain 进一步转发。

对于 basic workchain 的 shardchains，交付包括将附带数值（减去任何 gas 支付）添加到接收帐户的余额，如果收款账户是智能合约则根据消息的规则来调用。实际上，智能合约只有一个用于处理所有传入消息的入口点，它必须通过查看它们的前几个字节来区分不同类型的消息（例如，包含 TL 构造函数的前四个字节；参见 2.2.5 ）。

2.4.13 传递消息是一种交易。

因为消息的传递改变了帐户或智能合约的状态，所以它是接收 shardchain 中的特殊事务。忽略细节的话，基本上所有 TON 区块链交易都是向接收账户（智能合约）发送一个入站消息。

2.4.14 同一智能合约实例之间的消息。

回想一下，智能合约可能是本地的（即，像任何普通账户一样驻留在一个 shardchain 中）或全局的（即，在所有分片中具有实例，或者至少在所有分片中具有某个已知深度 d；参见 2.3.18）。 如果需要，全局智能合约的实例可以交换特殊消息以在彼此之间传递信息和价值。 在这种情况下，（不可伪造的）发件人 account_id 变得很重要（参见 2.4.4）。

2.4.15 发送给智能合约任何实例的消息；通配符地址。

有时，消息（例如，客户端请求）需要被传递到全局智能合约的所有实例，通常是最接近的（例如，如果存在与发送者相同的 shardchain 中的一个，则它是明显的候选者）。 一种方法是使用 “通配符收件人地址”，允许目标 account_id 的前 d 位采用任意值。 实际上，通常会将这些 d 位设置为与发送方的 *account_id* 中相同的值。

2.4.16 输入队列不存在。

区块链（通常是 shardchain；有时是主链）接收的所有消息——或者基本上由驻留在某个 shardchain 内的 “帐户链”——立即传递（即，由接收帐户处理）。 因此，没有 “输入队列”。 相反，如果由于区块总大小和 Gas 使用的限制，并非所有发往特定分片链的消息都可以被处理，一些消息只会在原始分片链的输出队列中堆积。

2.4.17 输出队列。

从无限分片范例（参见 2.1.2）的角度来看，每个账户链（即每个账户）都有自己的输出队列，包括它已生成但尚未发送给收件人的所有消息。 当然，账户链只有虚拟存在； 它们被分组为 shardchains，而 shardchain 有一个输出 “queue”，它由属于 shardchain 的所有帐户的输出队列的并集组成。

此 shardchain 输出 “队列” 仅对其成员消息强加部分顺序。 也就是说，必须在后续块中生成的任何消息之前提供在前一个块中生成的消息，并且必须按照它们生成的顺序传送由同一帐户生成并具有相同目的地的任何消息。

2.4.18 可靠，快速的链间消息传递。

对于像 TON 这样的可扩展多区块链项目来说，能够在不同的 shardchains 之间转发和传递消息（参见 2.1.3）是至关重要的，即使系统中有数百万个。 消息应该可靠地传送（即，消息不应丢失或传送不止一次）并且快速传送。 TON 区块链通过结合使用两个 “消息路由” 机制实现了这一目标。

2.4.19 Hypercube routing 超立方体路由：

确保传递的消息的 “慢速路径”。TON 区块链使用 “超立方体路由” 作为一种缓慢但安全可靠的方式，将消息从一个 shardchain 传递到另一个 shardchain，如有必要，可使用多个中间 shardchains 进行传输。否则，任何给定的 shardchain 的验证者都需要追踪所有其他 shardchains 的状态（输出队列），这将需要大量的计算能力和网络带宽，因为 shardchains 的总量增长，反而限制了系统的可扩展性。因此，无法直接从任何分片向其他分片传递消息。相反，每个分片仅 “连接” 到不同于其（w，s）分片标识符的一个十六进制数字的分片（参见 2.1.8）。这样，所有的 shardchains 都构成了一个 “超立方体” 图形，并且消息沿着这个超立方体的边缘传播。

如果将消息发送到与当前分片不同的分片，则当前分片标识符的一个十六进制数字（决定性选择）将被目标分片的相应数字替换，并且所得到的标识符将通过消息发给最近的目标。

超立方体路由的主要优点是区块有效性条件意味着创建分片链块的验证者必须收集并处理来自 “相邻” 分片链的输出队列的消息，以免丢失它们的 stake。 通过这种方式，可以预期任何消息迟早会到达其最终目的地； 消息也不能在传输过程中丢失或接受两次。

请注意，超立方体路由引入了一些额外的延迟和费用，因为必须通过几个中间的 shardchains 转发消息。 然而，这些中间分片链的数量增长非常缓慢，如对数链 N 的总数的对数 log N（更确切的说 [log16 N]-1）。例如，如果 N≈250，则最多只有一个中间跳； 对于 N≈4000 个 shardchains，最多两个。 通过四个中间跃点，我们可以支持多达一百万个分片链。 我们认为这对于系统的基本无限可扩展性来说是一个非常小的代价。

2.4.20 即时超立方体路由：消息的 “快速路径”。

TON 区块链的一个新特点是它引入了一条 “快速路径”，用于将消息从一个 shardchain 转发到任何其他 shardchain，允许在大多数情况下完全绕过 2.4.19 的 “慢” 超立方体路由并将消息传递到 最后一个目的地 shardchain 的下一个区块。

这个想法如下。 在 “慢” 超立方体路由期间，消息沿着超立方体的边缘（在网络中）传播，但是在每个中间顶点处被延迟（大约五秒）以在继续其航行之前被提交到相应的 shardchain 中。

为了避免不必要的延迟，可以使用沿超立方体边缘的合适的 Merkle 证明来中继消息，而无需等待将其提交到中间的 shardchains 中。实际上，网络消息应该从原始分片的 “任务组”（参见 2.6.8）的验证者转发到目的地 “任务组” 的指定分

片区块生成器（参见 2.6.9）。这可以直接完成，而不需要沿着超立方体的边缘。当带有 Merkle 证明的消息到达目的地 shardchain 的验证者（更确切地说，校对者；参见 2.6.5）时，他们可以立即将其提交到新的块中，而无需等待消息完成其沿着 “慢路 “。然后沿着超立方体边缘发送回传送确认以及合适的 Merkle 证据，并且可以通过提交特殊交易来用于沿着 “慢速路径” 停止消息的行进。

请注意，这种 “即时交付” 机制并不能取代 2.4.19 中描述的 “慢速” 但是防止故障的机制。仍然需要 “慢速路径”，因为验证者不会因丢失或仅仅决定不将 “快速路径” 消息提交到其区块链的新块而受到惩罚。

因此，两种消息转发方法并行运行，只有在将 “快速” 机制的成功证明提交到中间分片链时才会中止 “慢” 机制。

2.4.21 从邻居 shardchains 的输出队列收集输入消息。

当提出用于分片链的新块时，邻近的一些输出消息（如 2.4.19 的路由分支所示）分片链作为 “输入” 消息包含在新块中并立即传送（即处理）。 处理这些临近的输出消息的顺序，会存在一些规则。 基本上来说，必须在任何 “较新” 消息之前，先传递 “较旧” 消息（来自指向较旧主链块的分片链的区块）； 对于来自相同相邻 shardchain 的消息，必须遵守 2.4.17 中描述的输出队列的部分顺序。

2.4.22 从输出队列中删除消息。

一旦观察到输出队列消息已经被相邻的 shardchain 传递，它就会被特殊事务从输出队列中显式删除。

2.4.23 防止双重传递消息。

为了防止从相邻分片链的输出队列中获取的消息的传输两次，每个分片链（更确切地说，其中的每个帐户链）将最近传递的消息（或仅仅是它们的哈希）的集合保存为其状态的一部分。当观察到传递的消息由其始发的相邻 shardchain 从输出队列中删除时（参见 2.4.22），它也会从最近传递的消息的集合中删除。

2.4.24 转发用于其他 shardchains 的消息。

超立方体路由（参见 2.4.19）里有时出站消息不会传递到包含预期接收者的 shardchain，而是传递到位于到目的地的超立方体路径上的相邻 shardchain。这种情况下，“传递” 包括将入站消息移动到出站队列。这在区块中具体显示为包含消息本身的特殊转发事件。从本质上讲，这看起来好像是 shardchain 中的某个人收到了消息，并且结果生成了一条相同的消息。

2.4.25 支付转发和保留消息。

转发交易实际上花费了一些 gas（取决于转发的消息的大小），因此从代表该 shardchain 的验证者转发的消息的数值中扣除 gas fee。这种转发支付通常远小于当消息最终传递给其接收者时所支付的 gas 支付，即使该消息由于超立方体

路由而被多次转发。此外，只要消息保存在某个分片链的输出队列中，它就是 shardchain 全局状态的一部分，因此特殊事务也可以收集长时间保存全局数据的支付。

2.4.26 来自主链的消息。

消息可以直接从任何 shardchain 发送到主链，反之亦然。但是，在主链中发送消息和处理消息的 gas price 非常高，因此只有在真正需要时才会使用此功能——例如，验证者可以存入他们的 stake。在一些情况下，可以定义发送到主链的消息的最小 stake（附加值），仅当消息被接收方视为 “有效” 时才返回。

消息无法通过主链自动路由。workchain*id = 1* 的消息（1 是表示主链的特殊 *workchain*id）无法传递给主链。

原则上，可以在主链内创建消息转发智能合约，但使用它的价格会很高。

2.4.27 同一个 shardchain 中的帐户之间的消息。

在某些情况下，消息由属于某个 shardchain 的帐户生成，发往同一个 shardchain 中的另一个帐户。例如，这发生在一个尚未拆分为多个 shardchains 的新工作链中，因为负载此时可知。

此类消息可能会累积在分片链的输出队列中，然后作为后续块中的传入消息进行处理（为此目的，任何分片都被视为临近 shard）。但是，在大多数情况下，可以在原始块本身内传递这些消息。

为了实现这一点，对 shardchain 块中包括的所有事务施加部分顺序，并且关于该部分顺序处理事务（每个事务包括向某个账户传递消息）。特别是，允许事务处理关于该部分顺序的先前事务的一些输出消息。

在这种情况下，消息正文不会被复制两次。相反，始发和处理事务会参考消息的副本。

2.5 分片链的全局状态。 “Bag of Cells” 思想（一袋细胞）

现在我们准备描述 TON Blockchain 的全局状态，或者 basic workchain 的 shardchain。我们从 “高级” 或 “逻辑” 描述开始，其中包括全局状态是代数类型 ShardchainState 的值。

2.5.1 Shardchain 状态作为帐户链状态的集合。

根据 Infinite Sharding Paradigm（参见 2.1.2），任何 shardchain 只是一个（临时）虚拟 “帐户链” 集合，每个只包含一个帐户。这意味着，本质上，全局 shardchain 状态必须是一个 hashmap

$$ShardchainState: = (账户 \rightarrow AccountState) \qquad (23)$$

如果我们正在讨论分片的状态（w，s）（参见 2.1.8），那么所有出现在这个 hashmap 索引中的 account_id 必须以前缀 s 开头。

实际上，我们可能希望将 AccountState 分成几个部分（例如，将帐户输出消息队列分开以简化相邻 shardchains 的验证），并在 ShardchainState 内部有几个哈希映射（Account ➔ FountStateParti）。我们还可以向 ShardchainState 添加少量“全局”或“整数”参数（例如，属于该分片的所有帐户的总余额，或所有输出队列中的消息总数）。从“逻辑”（“高级”）的角度来看，（23）是 shardchain 全局状态看起来的良好的第一步。AccountState 和 ShardchainState 形式描述的字段可以借助 TL 方案（参见 2.2.5）完成。

2.5.2 拆分和合并 shardchain 状态。

请注意，shardchain 状态（23）的无限分片范例描述显示了在分割或合并分片时应如何处理此状态。实际上，这些状态转换结果是使用哈希映射的操作，非常简单。

2.5.3 账户链状态。

（虚拟）帐户链状态只是一个帐户的状态，由 AccountState 字段来描述。 通常它具有 2.3.20 中列出的全部或部分字段，具体取决于所使用的具体构造函数。

2.5.4。 全局工作链状态。

与（23）类似，我们可以通过相同的公式定义全局工作链状态，但允许使用 account_id 获取任何值，而不仅仅是一个分片的值。 类似于 2.5.1 中的注释也适用于这种情况：我们可能希望将此 hashmap 拆分为几个 hashmap，我们可能希望添加一些“整数”参数，例如总余额。

本质上，全局工作链状态必须由 ShardchainState 字段由 shardchain state 给出，因为如果此工作链的所有现有的 shardchains 合并为一个，我们将会获得 shardchain 的状态。

2.5.5 从 Low-level （低级）来看：“Bag of Cells”（一袋细胞）。

帐户链或 shardchain 状态的“低级”描述也是对上面给出的“高级”描述的补充。这个描述非常重要，因为它非常普遍，为通过网络表示，存储，序列化和传输几乎所有 TON 区块链使用的数据提供了通用基础（块，shardchain 状态，智能合约存储， Merkle 证明等）。同时，这种普遍的“低级”描述一旦被理解和实施，就可以使我们只关注“高级”考虑。

回想一下，TVM 通过 TVM 单元树或简称单元（参见 2.3.14 和 2.2.5）表示任意代数类型的值（例如，包括（23）的 ShardchainState）。

任何这样的单元由两个描述符字节组成，定义了某些标志和值 $0 \le b \le 128$，原始字节数，$0 \le c \le 4$，即对其他单元的引用数量。然后是 b 原始字节和 c 单元格引用。

单元引用的确切格式取决于实现以及单元是位于 RAM，磁盘，网络数据包，块中等等。一个有用的抽象模型在于想象所有单元格都保存在内容可寻址的内存中，单元格的地址等于其（sha256）哈希。回想一下，单元格的（Merkle）哈希是通过用它们的（递归计算的）哈希替换对其子单元格的引用并对生成的字节串进行哈希来精确计算的。

以这种方式，如果我们使用单元哈希来引用单元（例如，其他单元的内部描述），则系统稍微简化，并且单元的哈希开始与表示它的字节串的哈希重合。

现在我们看到 TVM 表示的任何对象，包括全局 shardchain 状态，可以表示为“一包单元格”-ie，一组单元格以及对其中一个的“根”引用（例如，通过哈希）。请注意，从此描述中删除了重复的单元格（“包的单元格”是一组单元格，而不是多个单元格的单元格），因此抽象树表示可能实际上成为有向无环图（dag）表示。

有人甚至可能将这种状态保存在 B-或 B + -tree 的磁盘上，包含所有相关单元（可能带有一些额外的数据，如子树高度或参考计数器），由单元格哈希索引。然而，这种想法的天真实施将导致一个智能合约的状态分散在磁盘文件的远端部分，我们宁愿避免 .19 现在我们将详细解释 TON 区块链使用的几乎所有对象如何表示为“细胞袋”，从而证明这种方法的普遍性。

2.5.6 Shardchain 的块作为“Bag of Cells”（一袋细胞）。

分片链的区块本身也可以用代数类型描述，并存储为“一袋细胞”。然后，可以简单地通过以任意顺序连接表示“单元格袋”中的每个单元的字节串来获得块的朴素二进制表示。例如，通过在区块头提供所有单元的偏移列表，并且尽可能将该列表中具有 32 位索引的其他单元的哈希引用替换为该表示，可以改进和优化该表示。然而，人们应该想象一个块本质上是一个“细胞袋”，所有其他技术细节只是次要的优化和实施问题。

2.5.7 更新“Bag of Cells”的对象。

想象一下，我们有一个旧版本的某个对象表示为“细胞袋”，我们想要代表同一个对象的新版本，据说与前一个对象没有太大差别。人们可能只是将新状态表示为具有其自身根的另一个“细胞袋”，并从中移除旧版本中出现的所有细胞。剩下的“细胞袋”本质上是对象的更新。拥有此对象的旧版本和更新的每个人都可以计算新版本，只需将两个单元格联合起来，然后删除旧的根（如果引用计数器变为零，则减少其引用计数器并取消分配单元格）。

2.5.8 更新账户状态。

特别是，可以使用 2.5.7 中描述的思想来表示对帐户状态或分片链的全局状态或任何哈希映射的更新。 这意味着当我们收到一个新的 shardchain 块（这是一个“细胞袋”）时，我们不仅仅是单独解释这个“细胞袋”，而是将它首先与代表前一状态的“细胞袋”结合起来。 分片链。 在这个意义上，每个块可以“包含”区块链的整个状态。

2.5.9 更新块。

回想一下块本身就是一个 “细胞袋”，因此，如果有必要编辑一个块，可以类似地将 “块更新” 定义为 “细胞袋”，在 “袋子” 的情况下进行解释。单元格 “这是该块的先前版本。这大致是 2.1.17 中讨论的 “垂直块” 背后的想法。

2.5.10 Merkle 证明作为 Bag of Cells”（一袋细胞）。

请注意，（广义）Merkle 证明——例如，一个断言 x [i] = y 从已知值 Hash（x）= h（参见 2.3.10 和 2.3.15）开始——也可以表示为 一袋 “细胞”。 也就是说，只需要提供一个单元格的子集，该子集对应于从 x：Hashmap（n，X）的根到其所需分支的路径，其索引为 i:2^n and value y:X。这些单元格的子节点的引用不是说谎在这个路径中，将在此证明中保留 “未解决”，由单元格哈希表示。 也可以通过在 “细胞袋” 中包括位于从根部的两条路径的并集上的 cells 来提供例如 x[i] = y and x[i′] = y′ 的同时 Merkle 证明。

2.5.11 Merkle 证明作为全节点的响应。

实质上，具有 shardchain（或帐户链）状态的完整副本的完整节点可以在轻节点（例如，运行 TON Blockchain 客户端的轻节点）请求时提供 Merkle 证明，从而启用接收器在没有外部帮助的情况下执行一些简单的查询，仅使用此 Merkle 证明中提供的单元格。light 节点可以将序列化格式的查询发送到整个节点，并通过 Merkle 证明或 Merkle 证明接收正确答案，因为请求者应该只能使用 Merkle 证明中包含的单元格来计算答案。这个 Merkle 证明只包含一个 “单元格包”，只包含那些属于 shardchain 状态的单元，这些单元在执行 light 节点查询时已被完整节点访问。这种方法尤其可用于执行智能合约的 “获取查询”（参见 4.3.12）。

2.5.12 用 Merkle 有效证明增强更新和状态更新。

使用 Merkle 有效性证明进行增强更新或状态更新。回想一下（参见 2.5.7）我们可以描述一个对象 statefromanoldvaluex 的变化：X toanewvaluex'：X bymeansofan“update”，它只是一个 “包的单元格”，包含代表新值的子树中的那些单元格 x'，但不在表示旧值 x 的子树中，因为假定接收器具有旧值 x 及其所有单元的副本。

但是，如果接收器没有 x 的完整副本，但只知道它的（Merkle）哈希 h = Hash（x），它将无法检查更新的有效性（即，所有 “dangling” 单元更新中的引用确实引用 x）树中的单元格。人们希望得到 “可验证的” 更新，并通过 Merkle 证明旧状态中所有被引用的细胞的存在。然后任何只知道 h = Hash（x）的人都能够检查更新的有效性并自己计算新的 h'= Hash（x'）。

因为我们的 Merkle 证明本身就是 “细胞袋”（参见 2.5.10），所以可以将这样的增强更新构建为 “细胞袋”，其中包含 x 的旧根，其中的一些后代以及来自 x 的根到它们，以及 x' 的新根和它不属于 x 的所有后代。

2.5.13 Shardchain 块中的账户状态更新。

特别是，应该扩充 shardchain 块中的帐户状态更新，如 2.5.12 中所讨论的那样。 否则，有人可能会提交一个包含无效状态更新的块，指的是旧状态中缺少的单元格； 证明这种阻滞无效将是有问题的（挑战者如何证明一个细胞不属于先前的状态？）。

现在，如果块中包含的所有状态更新都得到了增强，则很容易检查它们的有效性，并且它们的无效性也很容易显示为违反（广义）Merkle 哈希的递归定义属性。

2.5.14 “Bag of Cells” 思想（一袋细胞）

先前的考虑表明，我们需要在 TON 区块链或网络中存储或传输的所有内容都可以表示为 “一袋细胞”。这是 TON 区块链设计理念的重要组成部分。一旦解释了 “Bag of Cells” 方法并定义了 “细胞袋” 的一些 “低级” 序列化，就可以在高层次简单抽象定义出（依赖）代数数据类型的所有内容（块格式，shardchain 和帐户状态等）。“一切都是一小撮细胞” 统一的理念，大大简化了看似无关的服务的实施（比如 5.1.9 举例涉及支付通道）。

2.5.15 TON Blockchains 的区块头。

通常，区块链中的块以小标头开头，包含前一个块的哈希，其创建时间，块中包含的所有事务的树的 Merkle 哈希，等等。然后将块哈希定义为该小块头的哈希。因为块头最终取决于块中包含的所有数据，所以不能在不改变其哈希的情况下改变块。

在 TON Blockchain 块使用的 “单元格袋” 方法中，没有指定的块标题。相反，块哈希被定义为块的根单元的（Merkle）哈希。因此，块的顶部（根）单元可能被认为是该块的小 “标题”。

但是，根单元可能不包含通常从这种标头中预期的所有数据。实质上，需要标头包含 Block 数据字段中定义的一些字段。通常，这些字段将包含在几个单元格中，包括根目录。这些是共同构成所讨论的字段值的 “Merkle 证明” 的单元格。有人可能会在任何其他单元格之前的一开始就坚持要求一个块包含这些 “标题单元格”。然后只需要下载块序列化的前几个字节，以获得所有 “标题单元”，并知道所有预期的字段。

2.6 创建和验证新的区块。

TON 区块链最终由 shardchain 和 masterchain 块组成。 必须通过网络创建，验证这些块并将其传播给所有相关方，以使系统平稳正确地运行。

2.6.1 验证者（Validator）。

新块由特殊的指定节点（称为验证者）创建和验证。基本上，任何希望成为验证者的节点都可以成为一个节点，只要它能够将足够大的桩（在 TON 硬币中，

即 Grams；参见附录 A）存入主链。 Validators 为获得好工作获得了一些 “奖励”，即所有交易（消息）交易，存储和 gas，这些交易（消息）都投入到新生成的区块中，还有一些新铸造的 coins，反映了整个社区的 “gratitude” 保证 TON Blockchain 工作的验证者。该收入按比例分配给所有参与的验证人。

然而，作为验证者是一项高度责任。如果验证者签署了无效区块，则可以通过丢失部分或全部 stake 来处罚，并暂时或永久地从验证人集合中排除。如果验证者不参与创建块，则它不会收到与该块相关的奖励的份额。如果验证者长期放弃创建新块，它可能会丢失部分 stake，并被暂停或永久排除在验证者集之外。

所有这一切都意味着验证者不会 “什么都不做” 就获得金钱。实际上，它必须追踪所有或一些分片链的状态（每个验证者负责验证和创建分片链的某个子集中的新块），执行这些分片链中智能合约所请求的所有计算，接收有关其他的更新 shardchains 等。此活动需要相当大的存储空间，计算能力和网络带宽。

2.6.2 验证者（Validator）代替矿工（Miner）。

回想一下，TON Blockchain 使用的是共识算法是 POS，而不是比特币，当前版本的以太坊以及大多数其他加密货币采用的工作量证明方法。这意味着人们不能通过提供一些证明工作（计算许多其他无用的哈希）来 “挖掘” 一个新的块，并因此获得一些新的硬币。相反，必须成为验证者并花费一个人的计算资源来存储和处理 TON 区块链请求和数据。简而言之，一个人必须是开采新硬币的验证者。在这方面，验证人是新的矿工。

然而，还有其他一些方法来赚取 coin 除了是一个验证者，还有其他一些方法来赚取 coin

2.6.3 提名者（Nominator）和 “mining pools（矿池）”

要成为验证者，需要购买和安装多个高性能服务器并需要维持良好稳定网络连接。这并不像目前开发比特币所需的 ASIC 设备那么昂贵。然而，绝对不能在家用电脑上挖掘新的 TON coin，更不用说智能手机了。

在比特币，以太坊和其他工作量证明加密货币挖掘社区中，有一个挖掘池的概念，其中许多节点，具有不足以自行挖掘新块的计算能力，结合其努力并在之后分享奖励。

权益证明世界中的相应概念是提名者的概念。从本质上讲，这是一个节点借钱来帮助验证者增加其 stake；验证者然后将其奖励的相应份额（或之前商定的一部分——比如 50％）分配给提名者。

通过这种方式，提名者也可以参与 “采矿” 并获得与其愿意为此目的存入的金额成比例的一些奖励。它仅获得验证者奖励的相应份额的一小部分，因为它仅提供 “资本”，但不需要购买计算能力，存储和网络带宽。

但是，如果验证人因无效行为而失去其 stake，则提名人也会失去其 stake。从这个意义上说，提名者分担风险。它必须明智地选择其指定的验证人，否则可

能会赔钱。从这个意义上说，提名者做出加权决定，并用他们的资金对某些验证人进行 “投票”。

另一方面，这个提名或借出系统使人们能够成为验证者，而无需先向 Grams（TON 币）投入大量资金。换句话说，它可以防止那些使大量的 Grams 垄断供应验证者的人。

2.6.4 渔夫（Fisherman）：通过指出别人的错误来获得金钱。

获得一些奖励而不是验证者的另一种方法是成为一名渔夫。基本上，任何节点都可以通过在主链中存入少量存款而成为渔夫。然后，它可以使用特殊的 maschachain 事务来发布（Merkle）先前由验证者签名和发布的一些（通常是 shardchain）块的无效证明。如果其他验证人同意此无效证明，则违规验证人将被处罚（通过丢失部分 stake），并且渔夫获得一些奖励（从违规验证人处没收的一小部分硬币）。之后，必须按照 2.1.17 中的描述更正无效（shardchain）块。纠正无效的主链块可能涉及在先前提交的主链块之上创建 “垂直” 块（参见 2.1.17）；没有必要创建主链的分支。

通常，渔夫需要成为至少一些分片链的完整节点，并通过运行至少一些智能合约的代码来花费一些计算资源。虽然渔夫不需要具有作为验证者的计算能力，但我们认为成为渔夫的自然候选人是准备处理新区块的潜在验证者，但尚未被选为验证者（例如，由于未能存放足够大的桩号）。

2.6.5 收集者（Collator）：通过给验证者提供出新块的建议来获得金钱。

另一种获得一些奖励而不是验证者的方法是成为一名整理者。这是一个节点，它准备并向验证者建议新的 shardchain 块候选，补充（整理）从该 shardchain 的状态和其他（通常是邻近的）shardchains 获取的数据，以及合适的 Merkle 证明。（例如，当某些消息需要从相邻的分片链转发时，这是必要的。）然后验证者可以轻松地检查建议的块候选者的有效性，而无需下载此或其他分片链的完整状态。

因为验证者需要提交新的（整理的）区块候选者以获得一些（“挖掘”）奖励，所以将一部分奖励支付给愿意提供合适的区块候选者的整理者是有意义的。通过这种方式，验证者可以通过将其外包给整理者来摆脱观察相邻分片链状态的必要性。

但是，我们希望在系统的初始部署阶段不会有单独的指定合作者，因为所有验证者都可以自己充当合作者。

2.6.6 收集者或验证者：获取包括用户交易的交易费用。

用户可以向一些收集者或验证者打开小额支付通道，并支付少量硬币以换取在 shardchain 中包含他们的交易。

2.6.7 选举全局验证者。

每个月选出一组 “全局” 验证者（masterchain 每出 2 的 19 次个块选举一次）。这个设置会提前一个知道。

为了成为验证者，节点必须将一些 TON coins（Grams）转移到 masterchain 中，然后将它们作为预备的 stake 发送给系统特殊的智能合约。 与 stake 一起发送的另一个参数是 $l \geq 1$，该节点愿意接受的最大验证负载相对于可能的最小值。l 还有一个全局上限（另一个可配置参数）参数 L，当 L=10 时，我们把 l 称为 L。

然后通过这个智能合约选择全局验证者，选择最多 T 个 stake 最多的候选人并公布它们的身份。最初，验证者的总数是 T=100；我们预计随着负载的增加它可能增长到 1000。T 是一个可配置的参数（参见 2.1.21）。

2.6.8 选举验证者 “任务组”。

全局验证者（其中每个验证者被认为具有等于其 stake 的多重性——否则验证者可能倾向于假设几个标识并将它们之间的权益分开）仅用于验证新的主链块。shardchain 块仅通过特别选择的验证者子集进行验证，这些验证者子集取自 2.6.7 中所述选择的全局验证者集。

为每个分片定义的这些验证者 “子集” 或 “任务组” 每小时轮换一次（实际上，每 210 个主链块），并且它们是提前一小时知道的，这样每个验证者都知道验证需要哪些分片。并且可以为此做好准备（例如，通过下载丢失的 shardchain 数据）。

用于为每个分片（w，s）选择验证者任务组的算法是确定性伪随机。它使用验证者嵌入的伪随机数到每个主链块（由使用阈值签名的共识生成）来创建随机种子，然后计算例如 Hash（代码(w).code(s).validator*id.rand*seed）每个验证者。然后验证者按此哈希的值进行排序，并选择前几个验证者，以便至少具有验证者总 stake 的 20 / T，并且至少包含 5 个验证者。

这种选择可以通过特殊的智能合约来完成。在这种情况下，选择算法很容易升级，而不需要 2.1.21 中提到的投票机制的硬分叉。到目前为止提到的所有其他 “常量” such as 219, 210, T, 20, and 5 也是可配置的参数。

2.6.9 在每个任务组上轮换优先级顺序。

根据先前主链块和（shardchain）块序列号的哈希，对分片任务组的成员施加了某个 “优先级” 顺序。如上所述，通过生成和排序哈希来确定顺序。

当需要生成新的 shardchain 块时，选择创建此块的分片任务组验证者通常是关于此旋转 “优先级” 顺序的第一个。如果它无法创建块，则第二个或第三个验证者可以执行此操作。从本质上讲，他们都可以建议他们的块候选者，但具有最高优先级的验证者建议的候选者应该作为拜占庭容错（BFT）共识协议的结果而获胜。

2.6.10 shardchain 候选块的传播。

由于分片链任务组成员资格是提前一小时知道的，因此使用 TON 网络的一般机制（参见 3.3），其成员可以利用该时间构建专用的 “分片验证者多播覆盖网络”。当需要生成新的 shardchain 块时——通常在最近的主链块传播后一两秒钟——每个人都知道谁具有最高优先级来生成下一个块（参见 2.6.9）。该验证者将自行创建一个新的整理块候选者，或者在整理器的帮助下（参见 2.6.5）。验证者必须检查（验证）此块候选（特别是如果它已由某个整理器准备）并使用其（验证者）私钥对其进行签名。然后使用预先安排的多播覆盖网络将块候选传播到任务组的其余部分（任务组创建其自己的专用覆盖网络，如 3.3 中所述，然后使用 3.3.15 中描述的流式多播协议的版本来传播块候选者）。

真正的 BFT 方式是使用拜占庭组播协议，例如 Honey Badger BFT 中使用的协议：用（N，2N / 3）——擦除码编码块候选，发送 1 / N 结果数据直接发送给组的每个成员，并期望它们将其部分数据直接组播到组的所有其他成员。

然而，更快更直接的方法（参见 3.3.15）是将块候选者分成一系列有符号的一千字节块（“块”），用 Reed-Solomon 或者一个来增加它们的序列。喷泉代码（例如 RaptorQ 代码），并开始向 “multicast mesh”（即覆盖网络）中的邻居发送块，期望它们进一步传播这些块。一旦验证者获得足够的块以从它们重建块候选者，它就签署确认收据并通过其邻居将其传播到整个组。然后它的邻居停止向它发送新的块，但是可以继续发送这些块的（原始）签名，相信该节点可以通过自己应用 Reed-Solomon 或 Fountain 代码（具有所有必要的数据）来生成后续块。，将它们与签名结合起来，并传播给尚未准备好的相邻块。

如果在删除所有 “坏” 节点后 “多播网格”（覆盖网络）保持连接（回想一下，以拜占庭方式允许多达三分之一的节点是坏的，即以任意恶意方式行事），该算法将尽可能快地传播块候选。

不仅指定的高优先级块创建者可以将其块候选者多播到整个组。优先级的第二和第三验证者可以立即或在未能从最高优先级验证者接收块候选之后开始多播它们的块候选。但是，通常只有具有最大优先级的块候选者将由所有（实际上，至少为任务组的三分之二）验证者签名并作为新的 shardchain 块提交。

2.6.11 验证候选块。

一旦验证者接收到块候选并且检查了其原始验证者的签名，则接收验证者通过执行其中的所有事务并检查它们的结果是否与所声明的一致来检查该块候选的有效性。从其他区块链导入的所有消息必须由整理数据中的合适 Merkle 证明支持，否则块候选被视为无效（并且，如果将此证明提交给主链，则验证者已经签署了此块候选人可能会受到惩罚）。另一方面，如果发现块候选有效，则接收验证者对其进行签名，并通过 “mesh multicast network” 或直接网络消息将其签名传播到该组中的其他验证者。

我们想强调一个验证者不需要访问这个或相邻的 shardchains 的状态，以便检查（整理的）块候选的有效性。这允许验证非常快速地进行（没有磁盘访问），

并且减轻验证者的计算和存储负担（特别是如果他们愿意接受外部整理器的服务来创建块候选者）。

2.6.12 选举下一个候选块。

一旦块候选者收集任务组中验证者的有效性签名的至少三分之二（by stake），它就有资格被提交为下一个 shardchain 块。运行 BFT 协议以对所选择的块候选（可能存在多于一个）进行一致，所有 “好” 验证者优选该轮的具有最高优先级的块候选。作为运行该协议的结果，通过至少三个验证者（by stake）的签名来增强该块。这些签名不仅证明了所讨论的块的有效性，而且证明了它是由 BFT 协议选出的。之后，块（没有整理的数据）与这些签名组合，以确定的方式序列化，并通过网络传播给所有相关方。

2.6.13 验证者必须保留它们签名过的块。

在他们加入任务组并且之后至少一个小时（或者更确切地说是 2 的 10 次方个块）期间，验证者应该保留他们已签署和提交的块。未能向其他验证者提供签名块可能会受到惩罚。

2.6.14 将新 shardchain 块的块头和签名传播到所有验证人。

验证者使用类似于为每个任务组创建的 multicast mesh network，将新生成的 shardchain 块的标头和签名传播到全局验证者集。

2.6.15 masterchain 生成新的块。

在生成所有（或几乎所有）新的 shardchain 块之后，可以生成新的主链块。该过程与 shardchain 块（参见 2.6.12）基本相同，不同之处在于所有验证者（或至少三分之二）都必须参与此过程。因为新的 shardchain 块的头部和签名被传播到所有验证者，所以每个 shardchain 中最新块的哈希可以且必须包含在新的主链块中。一旦这些哈希值被提交到主链块中，外部观察者和其他分片链可能会认为新的 shardchain 块已提交且不可变（参见 2.1.13）。

2.6.16 验证者必须同步主链的状态。

主链和分片链之间一个值得注意的区别是，所有验证者都需要追踪主链状态，而不依赖于整理数据。这很重要，因为验证者任务组的知识来自主链状态。

2.6.17 Shardchain 的块是并行生成和传播的。

通常，每个验证者都是几个 shardchain 任务组的成员；它们的数量（因此验证者上的负载）大约与验证者的 stake 总比成比例。这意味着验证者并行运行新的 shardchain 块生成协议的几个实例。

2.6.18 Mitigation of block retention attacks.

因为验证者的总集合在仅看到其标题和签名之后将新的 shardchain 块的哈希值插入到主链中，所以生成此块的验证者很可能会合谋并试图避免完整地发布新

块。这将导致相邻 shardchains 的验证者无法创建新块，因为一旦将其哈希值提交到主链中，它们必须至少知道新块的输出消息队列。

为了缓解这种情况，新块必须收集来自其他验证者的签名（例如，相邻分片链的任务组加起来三分之二），证明这些验证者确实具有该块的副本并且愿意将它们发送给其他其他验证者。只有在这些签名出现后，新块的哈希才能包含在主链中。

2.6.19 Masterchain 块比 shardchain 块生成的更晚。

Masterchain 块大约每五秒生成一次，shardchain 块也是如此。 然而，虽然所有 shardchains 中的新块的生成基本上同时（通常由新的主链块的释放触发），但是故意延迟生成新的主链块是为了允许包含主链中新生成的 shardchain 块新的哈希值。

2.6.20 慢的验证者可能收到较低的奖励。

如果验证者“慢”，则可能无法验证新的块候选者，并且可以在没有其参与的情况下收集提交新块所需的签名的三分之二。 在这种情况下，它将获得与此块相关的奖励的较低份额。

这为验证者提供了优化其硬件，软件和网络连接的激励，以便尽可能快地处理用户交易。

但是，如果验证者在提交之前未能对块进行签名，则其签名可能包含在下一个块之一中，然后包含在部分奖励中（指数级递减，具体取决于自生成以来已生成的块数，例如，0.9^k 如果验证者迟到 k 块，则仍将给予此验证者。

2.6.21 验证者签名的 “Depth”。

通常，当验证者签署块时，签名仅证明块的相对有效性：如果此块和其他分片链中的所有先前块都有效，则此块有效。 验证者不能因为将前面的块中提交的无效数据视为理所当然而受到惩罚。

但是，块的验证者签名具有称为 “Depth” 的整数参数。 如果它不为零，则意味着验证者也断言指定数量的先前块的（相对）有效性。 这是 “slow” 或 “temporarily offline” 验证者捕获并签署一些未经签名提交的块的方法。 然后仍然会给他们一些块奖励（参见 2.6.20）。

2.6.22 验证者负责签名的 shardchain 块的相对有效性；绝对有效性如下。

我们想再次强调，在 shardchain 块 B 上的验证者签名仅证明该块的相对有效性（或者如果签名具有 “Depth”d，则可能也是 d 先前块的相对有效性，参见 2.6.21；但这并不会影响以下的讨论。换句话说，验证者断言通过应用 2.2.6 中描述的块评估函数 ev_block 从先前状态 s 获得分片链的下一状态 s^i ：

$$s' = ev_b lock(B)(s) \qquad (24)$$

以这种方式，如果原始状态 s 证明是 “incorrect”（例如，由于先前块之一的无效），则不能惩罚签名块 B 的验证者。渔夫（参见 2.6.4）只有在发现相对无效的区块时才应该投诉。 PoS 系统作为一个整体努力使每个块相对有效，而不是递归（或绝对）有效。但是请注意，如果区块链中的所有区块都相对有效，那么所有区块和整个区块链都是绝对有效的；使用区块链长度的数学归纳可以很容易地显示这个陈述。通过这种方式，可以容易地验证块的相对有效性的断言一起证明了整个区块链的绝对有效性。

注意，通过对块 B 进行签名，验证者在给定原始状态 s 的情况下断言该块是有效的（即，（24）的结果不是指示不能计算下一状态的值⊥）。以这种方式，验证者必须对在（24）的评估期间访问的原始状态的单元执行最小的正式检查。

例如，假设预期包含从提交到块中的事务访问的帐户的原始余额的单元格原来具有零个原始字节而不是预期的 8 或 16. 然后原始余额根本无法从单元格，并在尝试处理块时发生 “unhandled exception”。在这种情况下，验证者不应该在受到惩罚的痛苦上签署这样的阻止。

2.6.23 签名 masterchain 的区块。

主链块的情况有所不同：通过签署主链块，验证者不仅断言其相对有效性，还断言所有前面的块的相对有效性，直到该验证者承担其责任时的第一个块（但不再向后）。

2.6.24 验证者的总量。

在目前为止所描述的系统中，要选择的验证者总数的上限 T（参见 2.6.7）不能超过，比如几百或一千，因为所有验证者都应该参与 BFT 共识协议用于创建每个新的主链块，并且不清楚这些协议是否可以扩展到数千个参与者。更重要的是，主链块必须收集所有验证者中至少三分之二的签名（by stake），并且这些签名必须包含在新块中（否则，系统中的所有其他节点都没有理由信任新块而不自己验证它。如果超过，例如，每个主链块中必须包含一千个验证者签名，这将意味着每个主链块中的更多数据，由所有完整节点存储并通过网络传播，并且花费更多的处理能力来检查这些签名（在 PoS 系统中，完整节点不需要自己验证块，但是他们需要检查验证者的签名）。

虽然限制 T 到一千个验证者似乎对 TON 区块链的部署的第一阶段来说已经足够了，但是当 shardchains 的总数变得如此之大以至于几百个验证者不足以处理时，必须为未来的增长做出规定。他们都是。为此，我们引入了一个额外的可配置参数 $T' \leq T$（最初等于 T），并且只有顶部 T' 选举的验证者（by stake）才能创建和签署新的主链块。

2.6.25 权力下放的制度。

人们可能怀疑 TON Blockchain 等证明系统依赖于 $T \approx 1000$ 验证者来创建所有 shardchain 和 masterchain 区块，与传统的工作量证明区块链相比，必然会变得

“过于中心化”。比如 Bitcoin 或 Ethereum，每个人（原则上）都可以挖掘一个新区块，而没有明确的矿工总数上限。

然而，POW 如 Bitcoin 和 Ethereum，目前需要大量的计算能力（高 “哈希率”）来挖掘新块，并且概率很低。因此，出块的往往集中某几个人手中，他们投入大量资金建立研发中心，这些研发中心充满了针对采矿优化的定制设计硬件；算力掌握在几个大型采矿池的手中，这些采矿池集中并协调那些无法自己提供足够 “哈希率” 的人而接入网络。

因此，截至 2017 年，超过 75％的新的以太坊或比特币区块由不到 10 名矿工生产。事实上，两个最大的 Ethereum 矿池共同产生了超过一半的新区块！显然，这样的系统比依赖于 T≈1000 个节点来生成新块的系统更加中心化。

人们可能还会注意到成为 TON Blockchain 验证者所需的投资——即购买硬件（例如，几个高性能服务器）和 stake（如果需要，可以通过一系列提名者轻松收集；参照 2.6.3）——比成为一个成功的 solo Bitcoin 或 Ethereum 矿工所需的要低得多。实际上，2.6.7 的参数 L 将迫使提名者不加入最大的 “采矿池”（即，已经积累了最大 stake 的验证者），而是寻找目前接受来自提名者资金的较小验证者，或者甚至创建新的验证者，因为这将允许验证者的更高比例的 s'i / si——并且还可以使用提名者的 stake，因此从挖掘中获得更大的奖励。通过这种方式，TON 的 POS 实际上鼓励分散（创建和使用更多验证者）并惩罚中心化化。

2.6.26 块的相对可靠性。

块的（相对）可靠性只是已签署此块的所有验证者的总 stake。换句话说，如果这个区块被证明无效，这那某些参与者就会失去部分 stake。如果关注转移价值低于块可靠性的交易，可以认为它们足够安全。从这个意义上讲，相对可靠性是外部观察者在特定区块中可以拥有的信任度量。

请注意，我们说的是块的相对可靠性，因为它保证块是有效的，前提是块和所有其他所指的 shardchains 块都有效（参见 2.6.22）。

块的相对可靠性在提交后可以增长——例如，当添加了迟来的验证者签名时（参见 2.6.21）。另一方面，如果这些验证者中的一个由于其与其他块有关的不当行为而失去部分或全部 stake，则块的相对可靠性会降低。

2.6.27 “Strengthening（强化）” 区块链

重要的是为验证者提供激励，以尽可能地提高块的相对可靠性。 实现此目的的一种方法是向验证者分配一小笔奖励，以便将签名添加到其他分片链的块中。即使是 “would-be” 的有效者，他们已经存入的 stake 不足以通过 stake 进入最高的 T 验证人并被纳入全局验证人（参见 2.6.7），也可能参加这项活动（如果他们同意保留他们的 stake 而不是在失去选举后退出它。 这些潜在的验证者可能会成为渔夫的两倍（参见 2.6.4）：如果他们不得不检查某些区块的有效性，他们也可以选择报告无效区块并收集相关的奖励。

2.6.28 块的递归可靠性。

还可以将块的递归可靠性定义为其相对可靠性的最小值以及它所引用的所有块的递归可靠性（即，主链块，先前的 shardchain 块和相邻的 shardchains 的一些块）。换句话说，如果该块被证明是无效的，或者因为它本身无效或者因为它所依赖的块之一是无效的，那么至少这个数量的钱将被某人丢失。如果真的不确定是否信任块中的特定事务，则应该计算该块的递归可靠性，而不仅仅是相对块的递归可靠性。

在计算递归可靠性时走得太远是没有意义的，因为如果我们看得太远，我们将看到由验证者签署的块，其 stake 已经解冻并撤回。在任何情况下，我们都不允许验证者自动重新考虑那些旧的块（即，如果使用当前的可配置参数值，则在两个月前创建），并创建从它们开始的分叉或使用辅助设备纠正它们“垂直区块链”（参见 2.1.17），即使它们结果无效。我们假设两个月的时间段为检测和报告任何无效块提供了充足的机会，因此如果在此期间没有对块进行质询，则根本不可能受到质疑。

2.6.29 轻节点 POS 的结果。

TON Blockchain 使用的 Proof-of-Stake 方法的一个重要结果是，TON 区块链的轻节点（运行轻节点的客户端）不需要下载所有 shardchain 甚至主链区块的“标题”，以便能够自行检查由完整节点提供给它的 Merkle 证明的有效性作为其查询的答案。

实际上，因为最新的 shardchain 块哈希包含在主链块中，所以完整节点可以容易地提供 Merkle 证明，即给定的 shardchain 块从主链块的已知哈希开始有效。接下来，工作中的节点只需要知道主链的第一个块（其中第一组验证者被公布），其中（或至少其中的哈希）可以内置到客户端软件中，并且仅大约每个月发生一个主链块，其中宣布新选出的验证者集，因为该块将由前一组验证者签署。从那开始，它可以获得几个最新的主链块，或者至少它们的块头和验证者签名，并使用它们作为检查由完整节点提供的 Merkle 证明的基础。

2.7 拆分和合并 Shardchains

TON 区块链最具特色和独特的功能之一是它能够在负载变得过高时自动将分片链分成两部分，并在负载消退时将它们合并（参见 2.1.10）。我们必须详细讨论它，因为它的独特性及其对整个项目可扩展性的重要性。

2.7.1 分片配置。

回想一下，在任何给定的时刻，每个工作链被分成一个或几个 shardchains（w，s）（参见 2.1.8）。

这些分片链可以由 binary tree 的分支表示，具有根（w，∅），并且每个非叶节点（w，s）具有子（w，s.0）和（w，s.1）。 通过这种方式，属于工作链 w 的每个帐户都被分配给一个分片，并且知道当前 shardchain 配置的每个人都可以

确定包含帐户 account_id *的分片（w，s）：它是唯一一个二进制字符串 s 为前缀的分片 of account_id*。

分片配置——即此分片 binary tree，或给定 w 的所有活动（w，s）的集合（对应于分片 binary tree 的分支）——是主链状态的一部分，并且对于每个人都可用追踪主链。

2.7.2 最近的分片配置和状态。

回想一下，最新的 shardchain 块的哈希值包含在每个 masterchain 块中。 这些哈希以分片 binary tree（实际上是树的集合，每个工作链一个）组织。 这样每个主链块都包含最新的分片配置。

2.7.3 宣布并执行分片配置中的更改。

可以通过两种方式更改分片配置：分片（w,s）可以分为两个分片（w,s.0）和（w,s.1），或两个 “兄弟” 分片（w,s.0）和（w,s.1）可以合并为一个分片（w,s）。

这些拆分 / 合并操作被预先公布几个（例如，26； 这是可配置参数）块，首先在相应的 shardchain 块的 “块头” 中，然后在引用这些 shardchain 块的主链块中。所有相关方都需要该预先通知以准备计划的改变（例如，overlay multicast network 以分发新创建的分片链的新块，如 3.3 中所讨论的）。然后提交更改，首先进入 shardchain 块的（头部）（在分割的情况下；对于合并，两个分片链的块应该提交更改），然后传播到主链块。通过这种方式，主链块不仅定义了创建之前的最新分片配置，还定义了下一个立即分片配置。

2.7.4 新 shardchains 的验证者任务组。

回想一下，每个分片（即每个分片链）通常被分配一个验证者子集（验证者任务组），专门用于创建和验证相应分片链中的新块（参见 2.6.8）。这些任务组在一段时间内（大约一小时）被选出，并且提前一段时间（大约一小时）知道，并且在此期间是不可变的 .23
但是，由于拆分 / 合并操作，实际的分片配置可能会在此期间发生更改。必须将任务组分配给新创建的分片。这样做如下：

请注意，任何活动分片（w，s）都将是某些唯一确定的原始分片（w，s'）的后代，这意味着 s' 是 s 的前缀，或者它将是原始分片的子树的根（w，s'），其中 s 将是每个 s' 的前缀。在第一种情况下，我们只需将原始分片（w，s'）的任务组加倍作为新分片（w，s）的任务组。在后一种情况下，新分片（w，s）的任务组将是所有原始分片（w，s'）的任务组的并集，它们是分片树中（w，s）的后代。

通过这种方式，每个活动分片（w，s）都被分配了一个明确定义的验证者子集（任务组）。分割分片时，两个子代都从原始分片继承整个任务组。合并两个分片时，它们的任务组也会合并。

追踪主链状态的任何人都可以为每个活动分片计算验证者任务组。

2.7.5 在原任务组的工作期间限制拆分 / 合并的操作。

最终，将考虑新的分片配置，并且将自动为每个分片分配新的专用验证者子集（任务组）。在此之前，必须对拆分 / 合并操作施加一定的限制；否则，如果原始分片快速分成 2k 个新分片，则原始任务组可能最终同时为大 k 验证 2k 分片链。

这是通过对可以从原始分片配置（用于选择当前负责的验证者任务组的配置）中删除活动分片配置的距离施加限制来实现的。例如，如果 s' 是 s 的前身（即，s'），则可能要求分片树中从活动分片（w，s）到原始分片（w，s'）的距离不得超过 3。如果 s' 是 s 的后继（即 s 是 s' 的前缀），则是二进制字符串 s）的前缀，并且不得超过 2。否则，不允许拆分或合并操作。

粗略地说，人们在给定的验证者任务组的责任期间对分片（例如，三个）或合并（例如，两个）的次数施加限制。除此之外，在通过合并或拆分创建了一个分片之后，它不能在一段时间内重新配置（一定数量的块）。

2.7.6 确定拆分操作的必要性。

分片链的拆分操作由某些形式条件触发（例如，如果连续 64 个块，则分片链块至少满 90％）。这些条件由 shardchain 任务组监视。如果满足它们，首先在新的 shardchain 块的块头中包含 “split preparation” 标志（并且传播到引用该 shardchain 块的 masterchain 块）。然后在几个块之后，“split commit” 标志被包含在 shardchain 块的头部中（并传播到下一个主链块）。

2.7.7 执行拆分操作。

在 shardchain（w，s）的块 B 中包含 “split commit” 标志之后，该 shardchain 中不会有后续块 B'。 相反，将分别创建 shardchains（w，s.0）和（w，s.1）的两个块 B0′ and B1′，两者都将块 B 称为它们的前一个块（并且它们都将通过 标题中的分片标记已被分割）。 下一个主链块将包含新 shardchains 的块 B0' 和 B1' 的哈希值； 不允许包含 shardchain（w，s）的新块 B' 的哈希，因为 “split commit” 事件已经被提交到先前的主链块中。

请注意，两个新的分片链将由与旧验证者相同的验证者任务组验证，因此它们将自动获得其状态的副本。 从无限分片范式（the Infinite Sharding Paradigm）的角度来看，状态分裂操作本身非常简单（参见 2.5.2）。

2.7.8 确定合并操作的必要性。

通过某些形式的条件检测分片链合并操作的必要性（例如，对于 64 个连续块，两个兄弟分片链块的大小之和不超过最大块大小的 60％）。这些正式条件还应考虑这些区块所消耗的总 gas，并将其与当前区块 gas limit 进行比较，否则区块可能会很小，因为有一些计算密集型交易会阻止更多的交易。

这些条件由兄弟分片（w，s.0）和（w，s.1）的验证者任务组监视。请注意，兄弟姐妹必须是超立方体路由的邻居（参见 2.4.19），因此来自任何分片的任务组的验证者将在某种程度上监视兄弟分片。

当满足这些条件时，任何一个验证者子组都可以通过发送特殊消息向另一个建议它们进行合并。然后，它们组合成一个临时的 “合并任务组”，具有组合的成员，能够运行 BFT 一致性算法，并在必要时传播块更新和块候选。

如果他们就合并的必要性和准备情况达成共识，则 “合并准备” 标志将被提交到每个分片链的某些块的标题中，以及兄弟任务组的至少三分之二的验证者的签名（并传播到下一个主链块，以便每个人都可以为即将进行的重新配置做好准备。但是，他们继续为某些预定义数量的块创建单独的 shardchain 块。

2.7.9 执行合并操作。

之后，当来自两个原始任务组的验证者准备好成为合并的 shardchain 的验证者时（这可能涉及从兄弟 shardchain 和状态合并操作的状态转移），它们提交 “merge commit” 标志 在他们的 shardchain 块的标题中（此事件传播到下一个主链块），并停止在单独的 shardchains 中创建新块（一旦出现合并提交标志，则禁止在单独的 shardchains 中创建块）。相反，创建一个合并的 shardchain 块（由两个原始任务组的并集），在其 “header” 中引用它的两个 “preceding blocks”。 这反映在下一个主链块中，它将包含新创建的合并 shardchain 块的哈希值。 之后，合并的任务组继续在合并的 shardchain 中创建块。

2.8 区块链项目的分类

我们将通过将 TON Blockchain 与现有的区块链项目进行比较来结束对 TON Blockchain 的讨论。 然而在此之前，我们必须引入足够多一般的区块链项目分类，基于此分类的特定区块链项目比较将在推 2.9 里展开。

2.8.1 区块链项目的分类。

第一步，我们先建议出区块链的一些分类标准（即区块链项目）。它必须忽略所考虑项目的一些具体和独特的特征，所以任何分类都都会可能不完整和是表面的。 但是我们认为这是必要的，至少提供了区块链领域项目粗略鸟瞰的第一步。

我们考虑的标准清单如下：

- 单区块链与多区块链架构（参见 2.8.2）
- 共识算法：POS 权益证明与 POW 工作量证明（参见 2.8.3）
- 对于 Proof-of-Stake 系统，是否使用了精确的块生成，验证和一致性算法（两个是 DPOS 与 BFT；参见 2.8.4）
- 支持 “任意”（图灵完备）智能合约（参见 2.8.6）多区块链系统有额外的分类标准（参见 2.8.7）：
- 成员区块链成员的类型和规则：同构，异构（参见 2.8.8）；混合（参见 2.8.9）。 联盟（参见 2.8.10）。

- 缺席或存在内部或外部的主链（参见 2.8.11）
- 系统支持分片（参见 2.8.12）。 静态或动态分片（参见 2.8.13）。
- 成员区块链之间的相互作用：松散耦合和紧密耦合系统（参见 2.8.14）

2.8.2 单链区块链和多链区块链项目。

第一个分类标准是系统中区块链的数量。如最古老也是最简单的单链区块链（简称 “单链项目”）；更复杂的项目使用（或者更确切地说，计划使用）多个区块链（“多链项目”）。

Singlechain 项目通常更简单，测试更好；他们经受住了时间的考验。它们的主要缺点是性能低，或者至交易吞吐量低，每秒 10 笔（比特币）到 140 笔（以太坊）不等。一些专门的系统（如 Bitshares）每秒能够处理成千上万的专用事务，代价是要求区块链状态适合内存，并将处理限制在预定义的特殊事务集中，然后由高度优化的代码执行，这些代码用 C ++ 这样的语言编写（它没有 VM）。

多链项目承诺每个人都渴望的可扩展性。它们可以存储更大的总状态和每秒处理更多的事务，代价是使项目更加复杂，并且其实现起来更具挑战性。因此，目前几乎没有还多链项项目在运行，但大多数提议项目都是多链项目。我们相信未来属于多链项目。

2.8.3 创建和验证区块：工作量证明与权益证明。

另一个重要的区别是用于创建和传播新块的算法和协议，检查它们的有效性，如果出现则选择 fork 中的一个。

两种最常见的范例是工作量证明（PoW）和权益证明（PoS）。工作量证明方法通常允许任何节点创建（“mine”）新块（并获得与挖掘块相关的一些奖励），如果它足够幸运地解决其他无用的计算问题（通常涉及在其他竞争者设法做到这一点之前计算大量哈希值。在 forks 的情况下（例如，如果两个节点发布两个其他有效但不同的块跟随前一个），则最长的 fork 获胜。通过这种方式，区块链不可变性的保证是基于生成区块链所花费的工作量（计算资源）：任何想要创建区块链分支的人都需要重新做这项工作来创建替代方案已提交块的版本。为此，需要控制创建新块所花费的总计算能力的 50％以上，否则备用 fork 将成为最长的成功率。

Proof-of-Stake 方法基于由一些特殊节点（验证者）做出的大量 stake（由加密货币提名）来声明他们已经检查（验证）了一些块并且发现它们是正确的。验证者签署块，并为此获得一些小奖励；但是，如果一个验证人被抓到签署了一个不正确的区块，并且出现了证据，那么其部分或全部 stake 将被没收。通过这种方式，区块链的有效性和不变性的保证由验证者对区块链有效性的总 stake 量给出。
利用 Proof-of-Stake 方法更加自然，它激励验证者（取代 PoW 矿工）执行有用的计算（需要检查或创建新的块，特别是通过执行列出的所有交易）块）而不是计算否则无用的哈希。通过这种方式，验证者将购买更适合于处理用户交易

的硬件，以便接收与这些交易相关的奖励，从整个系统的角度来看，这似乎是非常有用的投资。

然而，实施证明制度在实施方面更具挑战性，因为必须提供许多罕见但可能的条件。例如，一些恶意验证者可能合谋破坏系统以获取一些利润（例如，通过改变它们自己的加密货币余额）。这导致了一些非平凡的游戏理论问题。

简而言之，Proof-of-Stake 更自然，更有前途，特别是对于多链项目（因为如果有许多区块链，工作量证明需要大量的计算资源），但必须更仔细地考虑和实施。大多数目前正在运行的区块链项目，尤其是最老的项目（比如 Bitcoin 和最初版本的 Ethereum），都使用了工作量证明。

2.8.4 POS 的演变：DPOS vs BFT

虽然 Proof-of-Work 算法彼此非常相似，并且主要区别在于必须为挖掘新块而计算的哈希函数，但是 Proof-of-Stake 算法有更多可能性。很值得对他们细分一下。

这里必须要回答下关于 Proof-of-Stake 算法的问题：

- 谁可以生成新块——任何完整节点，或仅生成（相对）小验证者子集的成员？（大多数 PoS 系统需要生成新块，并由多个指定验证者之一签名。）
- 验证者是否通过其签名保证块的有效性，或者是否所有完整节点都可以自行验证所有块？（可扩展的 PoS 系统必须依赖验证者签名，而不是要求所有节点验证所有区块链的所有块。）
- 是否有预先知道的下一个区块链区块的指定生产者，以便其他人无法生产该区块？
- 新创建的块最初是仅由一个验证者（其生产者）签署的，还是必须从一开始就收集大多数验证者签名？
  虽然根据这些问题的答案，似乎有 24 种可能的 PoS 算法类别，但实际上的区别归结为两种主要的 PoS 方法。 实际上，设计用于可扩展多链系统的大多数现代 PoS 算法以相同的方式回答前两个问题：只有验证者才能生成新块，并且它们保证块有效性，而不需要所有完整节点检查有效性 他们自己的所有块。

至于最后两个问题，他们的答案结果非常相关，基本上只留下两个基本选项：

- Delegated Proof-of-Stake 委托证明（DPOS）：每个区域都有一个众所周知的指定生产者； 没有人可以产生那块； 新块最初只由其生成验证者签名。
- Byzantine Fault Tolerant 拜占庭容错（BFT）PoS 算法：有一个已知的验证者子集，其中任何一个都可以建议一个新的块； 在被释放到其他节点之前必须由大多数验证者验证和签名的几个建议候选者中的实际下一个块的选择是通过拜占庭容错共识协议的版本来实现的。

2.8.5 DPOS 和 BFT POS 的比较

BFT 方法的优点在于，新生产的块从一开始就具有大多数验证者的签名，证明其有效性。另一个优点是，如果大多数验证者正确执行 BFT 共识协议，则根本不会出现任何分支。另一方面，BFT 算法往往相当复杂，需要更多时间让验证子集达成共识。因此不能经常生成块。这就是为什么我们期望 TON 区块链（从这个分类的角度来看是一个 BFT 项目）每五秒只产生一次块。在实践中，这个间隔可能会减少到 2-3 秒（尽管我们不承诺这一点），但如果验证者遍布全球，则不会更进一步。

DPOS 算法具有非常简单和直接的优点。它可以经常产生新的块——比如每两秒一次，或者甚至每秒一次，因为它依赖于事先已知的指定块生产者。

但是，DPOS 要求所有节点——或至少所有验证者——验证收到的所有块，因为生成和签署新块的验证者不仅确认了该块的相对有效性，而且还确认了它所引用的前一个块的有效性，并且所有块进一步回到链中（可能直到当前验证者子集的责任期的开始）。在当前验证者子集上存在预定顺序，因此对于每个块，存在指定的生成器（即，期望生成该块的验证者）；这些指定的生产者以循环方式轮换。通过这种方式，块首先仅由其生成验证者签名；然后，当下一个区块被开采时，它的生产者选择引用这个区块而不是它的前任之一（否则它的区块将位于较短的链中，这可能会失去未来 “最长的分叉” 竞争），下一个块的签名本质上也是前一个块的附加签名。通过这种方式，新块逐渐收集更多验证者的签名——例如，在生成接下来的 20 个块所需的时间内收集 20 个签名。一个完整的节点要么需要等待这二十个签名，要么自己验证块，从一个充分确认的块（比如二十个块）开始，这可能不是那么容易。

DPOS 算法的明显缺点是，只有在开采了 20 多个块之后，新块（以及提交到其中的事务）才能达到相同的信任级别（“2.6.28 中讨论的” recursive reliability”“）。 BFT 算法，可立即提供此级别的信任（例如，20 个签名）。另一个缺点是 DPOS 使用 “最长叉胜” 方法切换到其他叉子；如果至少一些生产者在我们感兴趣的生产者之后未能生成后续的块（或者由于网络分区或复杂的攻击我们未能观察到这些块），这很可能使得 forks 很可能。

我们认为 BFT 方法虽然比 DPOS 实现更复杂，需要更长的时间间隔，但更适合于 “紧密耦合”（参见 2.8.14）多链系统，因为其他区块链几乎可以开始行动在新的块中看到一个已完成的事务（例如，为它们生成一条消息）之后立即，而不等待 20 次有效性确认（即，接下来的 20 个块），或等待接下来的六个块以确保没有叉子出现并自行验证新块（在可扩展的多链系统中验证其他区块链的块可能变得过高）。因此，它们可以实现可扩展性，同时保持高可靠性和可用性（参见 2.8.12）。

另一方面，DPOS 可能是 “松散耦合” 多链系统的理想选择，其中不需要区块链之间的快速交互——例如，如果每个区块链（“workchain”）代表一个单独的分布链交换和区块链间的交互仅限于将 token 从一个工作链转移到另一个工作链中（或者更确切地说，以一个接近 1：1 的速率将一个山寨币交易到另一个工作链中）。这是在 BitShares 项目中实际完成的，它非常成功地使用了 DPOS。

总而言之，虽然 DPOS 可以生成新的块并且更快地包含交易（块之间的间隔更小），但这些交易达到了在其他区块链和离线应用程序中使用它们所需的信任级别 “已提交” 并且 “不可变” 比 BFT 系统慢得多——比如说，在 30 秒 26 而不是 5 秒。更快的交易包含并不意味着更快的交易承诺。如果需要快速的区块链间交互，这可能会成为一个巨大的问题。在这种情况下，必须放弃 DPOS 并选择 BFT PoS。

2.8.6 支持交易里的图灵完备代码，即允许执行任意智能合约。

区块链项目通常会在其区块中收集一些交易，这会以一种被认为有用的方式改变区块链状态（例如，将一些加密货币从一个账户转移到另一个账户）。 一些区块链项目可能只允许某些特定的预定义分片的事务（例如，从一个帐户到另一个帐户的值传输并提供正确签名）。其他人可能会在交易中支持某种有限形式的脚本。最后，一些区块链支持在事务中执行任意复杂的代码，使系统（至少在原理上）能够支持任意应用程序，只要系统的性能允许。这通常与 “图灵完备的虚拟机和脚本语言”（意味着任何可以用任何其他计算语言编写的程序可以重写以在区块链内执行）和 “智能合约”（这是驻留在区块链中的程序）有关。

当然，对任意智能合约的支持使系统真正具有灵活性。另一方面，这种灵活性需要付出代价：这些智能合约的代码必须在某个虚拟机上执行，并且当有人想要创建或验证块时，必须每次为块中的每个事务执行此操作。与预定义且不可变的简单事务分片集相比，这会降低系统的性能，这可以通过使用诸如 C ++（而不是某些虚拟机）之类的语言来实现它们的处理来优化。

最终，对图灵完备智能合约的支持似乎在任何通用区块链项目中都是可取的；否则，区块链项目的设计者必须事先决定他们的区块链将用于哪些应用程序。事实上，正是比特币不支持智能合约才有了以太坊。

在（异构；参见 2.8.8）多链系统中，一些区块链（即工作链）中支持图灵完全智能合约，一些自定义高度优化过的交易，这样保持 “两全其美”。

2.8.7 多链系统的分类。

到目前为止，该分类对于单链和多链系统都是有效的。 然而，多链系统承认了更多的分类标准，反映了系统中不同区块链之间的关系。 我们现在讨论这些标准。

2.8.8 区块链类型：同构与异构系统。

在多链系统中，所有区块链可以基本上是相同类型并且具有相同的规则（即，使用相同格式的事务，用于执行智能合约代码的相同虚拟机，共享相同的密码保存等等），这种相似性被明确使用，但每个区块链中的数据不同。 在这种情况下，我们说系统是同构的。 否则，不同的区块链（通常是工作链）这种情况）可以有不同的 “rules”。 然后我们说系统是异构的。

2.8.9 同构-异构混合系统。

有时我们有一个混合系统，其中的区块链有几组类型或规则，但是存在许多具有相同规则的区块链，并且这一事实被明确使用。 然后它是一个同构-异构混合系统。 据我们所知 TON Blockchain 是这类唯一的系统。

2.8.10 若干工作链的异构系统具有相同的规则；联盟。

在一些情况下，具有相同规则的若干区块链（工作链）可以存在于异构系统中，但是它们之间的交互与具有不同规则的区块链之间的相互作用相同（即，它们的相似性未被明确利用）。即使他们似乎使用 “the same” 加密货币，他们实际上使用不同的 “altcoins”（加密货币的独立化身）。有时人们甚至可以使用某些机制将这些山寨币转换为接近 1：1 的速率。然而，在我们看来这并不能使系统同构化；它仍然是异构的。我们说这种具有相同规则的异构工作链集合是联盟。

虽然允许用相同规则（即联盟）创建多个工作链的异构系统看起来似乎是构建可扩展系统的廉价方式，但这种方法也有许多缺点。从本质上讲，如果有人在许多具有相同规则的工作链中托管大型项目，她就不会获得大型项目，而是获得该项目的许多小实例。这就像拥有一个聊天应用程序（或游戏），允许在任何聊天（或游戏）房间中最多有 50 个成员，但通过创建新房间来 “缩放” 以在必要时容纳更多用户。因此，很多用户可以参与聊天或游戏，但我们可以说这样的系统真正可扩展吗？

2.8.11 内部或外部存在主链。

存在主链，外部或内部。 有时，多链项目有一个独特的 “主链”（有时称为 “控制区块链”），例如用于存储系统的整体配置（所有活动区块链的集合，或者更确切地说是工作链 ），当前的验证者集（用于证明系统）等等。 有时其他区块链被 “绑定” 到主链上，例如通过将最新块的哈希值存入其中（TON Blockchain 也是如此）。

在某些情况下，主链是外部的，这意味着它不是一个部分该项目，但一些其他预先存在的区块链，最初与新项目的使用完全无关，并且与其无关。 例如，可以尝试使用以太坊区块链作为外部项目的主链，并为此目的将特殊智能合约发布到以太坊区块链中（比如挑出和惩罚验证者）。

2.8.12 分片支持。

一些区块链项目（或系统）本身支持分片，这意味着几个（必然是同构的；参见 2.8.8）区块链被认为是单链（从高级别的角度来看）虚拟区块链的分片。例如，可以创建具有相同规则的 256 个分片区块链（“shardchains”），并根据其 account_id 的第一个字节将帐户的状态保持在一个选定的分片中。

分片是扩展区块链系统的一种自然方法，因为如果它被正确实现，系统中的用户和智能合约根本不需要知道分片链的存在。实际上，当负载变得太高时，人们通常希望为现有的单链项目（例如以太坊）添加分片。

另一种扩展方法是使用 2.8.10 中描述的异构工作链的 “联盟”，允许每个用户将她的帐户保存在她选择的一个或多个工作链中，并将资金从她的帐户转移到另一个工作链中必要时工作链，基本上执行 1：1 的山寨币交换操作。这种方法的缺点已在 2.8.10 中讨论过。

然而，分片在快速可靠的时尚中并不那么容易实现，因为它暗示了不同分片链之间的大量消息。例如，如果账户在 N 个分片之间均匀分配，并且唯一的交易是从一个账户到另一个账户的简单资金转账，那么在单链区块链中只会执行所有交易的一小部分（1 / N）；几乎所有（1——1 / N）交易都涉及两个区块链，需要区块链间通信。如果我们希望这些事务快速，我们需要一个快速系统来在 shardchains 之间传输消息。换句话说，区块链项目需要在 2.8.14 中描述的意义上 “紧密耦合”。

2.8.13 动态和静态分片。

分片可能是动态的（如果在必要时自动创建其他分片）或静态（修改预定义数量的分片，只能通过硬分叉来修改。 大多数分片提案都是静态的）。 TON 区块链使用动态分片（参见 2.7）。

2.8.14 区块链之间的相互作用：松散耦合和紧密耦合的系统。

可以根据组成区块链之间支持的交互级别对多区块链项目进行分类。

最低水平的支持是不同区块链之间没有任何相互作用。我们在这里不考虑这种情况，因为我们宁愿说这些区块链不是一个区块链系统的一部分，而只是相同区块链协议的单独实例。

下一级支持是对区块链之间的消息传递没有任何具体支持，原则上使交互成为可能，但很尴尬。我们称这种系统为 “松散耦合”；在他们中，必须发送消息并在区块链之间传递价值，好像它们是属于完全独立的区块链项目的区块链一样（例如，比特币和以太坊；想象两方想要将比特币区块链中的一些比特币交换到以太网中，保存在以太坊区块链中）。换句话说，必须在源区块链的块中包含出站消息（或其生成事务）。然后，他（或其他一方）必须等待足够的确认（例如给定数量的后续块）以将原始事务视为 “已提交” 和 “不可变”，以便能够执行基于外部操作的外部操作在它的遗嘱上。只有这样，才能提交将消息中继到目标区块链的事务（可能连同引用原始事物的 Merkle 证明）。

如果在传输消息之前没有等待足够长的时间，或者由于其他原因无论如何都发生了分叉，则两个区块链的连接状态变得不一致：消息被传递到从未生成过的第二个区块链中（最终选择的分支）第一个区块链。

有时通过标准化消息的格式以及所有工作链的块中输入和输出消息队列的位置来添加对消息传递的部分支持（这在异构系统中尤其有用）。虽然这在一定程度上促进了消息传递，但它在概念上与先前的情况没有太大的不同，因此这种系统仍然 “松散耦合”。

相比之下，“紧密耦合” 系统包括在所有区块链之间提供快速消息传递的特殊机制。期望的行为是能够在原始区块链的块中生成消息之后立即将消息传递到另

一个工作链。另一方面，“紧密耦合” 系统也可望在货叉的情况下保持整体一致性。虽然这两个要求乍一看似乎是矛盾的，但我们认为 TON 区块链使用的机制（将 shardchain 块哈希包含在主链块中；使用 “垂直” 区块链来修复无效块，参见 2.1.17 ；超立方路由，参见 2.4.19；即时超立方体路由，参见 2.4.20）使其成为 “紧密耦合” 系统，也许是目前唯一的系统。

当然，构建 “松散耦合” 系统要简单得多；然而，快速有效的分片（参见 2.8.12）要求系统 “紧密耦合”。

2.8.15 简化分类。 几代区块链项目。

到目前为止，我们建议的分类将所有区块链项目拆分为大量类。 但是我们使用的分类标准在实践中恰好相关。 通过一些例子，这使我们能够建议区块链项目分类的简化 “generational” 方法，作为对现实的非常粗略的近似。 尚未实施和部署的项目以斜体显示。

- **第一代：单链，PoW，不支持智能合约。** 示例：比特币（2009）和许多其他无趣的模仿者（Litecoin，Monero，...）。
- **第二代：单链，PoW，智能合约支持。** 例如：以太坊（2013 年；2015 年部署），原始形式。
- **第三代：单链，PoS，智能合约支持。** 例如：未来的以太坊（2018 年或更晚）。
- **第三（3'）代：多链，PoS，不支持智能合约，松散耦合。** 示例：Bitshares（2013-2014；使用 DPOS）。
- **第四代：多链，PoS，智能合约支持，松散耦合。** 示例：EOS（2017；使用 DPOS），PolkaDot（2016；使用 BFT）。
- **第五代：多链，PoS 与 BFT，智能合约支持，紧密耦合，分片。** 示例：TON（2017）。

虽然并非所有区块链项目都属于这些类别中的一个，但大多数都属于这些类别。

2.8.16 改变区块链项目 “genome（基因）” 的复杂性。

上述分类定义了区块链项目的 “genome（基因）”。这个基因组是非常 “rigid（僵硬）”：一旦项目部署并且被很多人使用，几乎不可能改变它。一个人需要一系列硬分叉（这需要大多数社区的批准），即使这样，为了保持向后兼容，改变也需要非常保守（例如，改变虚拟机的语义可能打破现有的智能合约）。另一种选择是创建具有不同规则的新 “sidechains 侧链”，并以某种方式将它们绑定到原始项目的区块链（或区块链）。有人可能会使用现有单区块链项目的区块链作为外部主链，用于一个基本上是新的独立项目。

我们的结论是项目的 “genome（基因）” 一旦部署就很难改变。即使从 PoW 开始并计划在未来用 PoS 替换它也是相当复杂的 .。最初设计的项目没有分片而

后再加入，这种支持几乎是不可能的。实际上，将智能合约的支持添加到老项目中（即比特币）被认为是不可能的（或者至少是大多数比特币社区不受欢迎的），并会最终导致创建一个新的区块链项目——以太坊。

2.8.17 TON Blockchain 的 “genome（基因）”

因此，如果想要构建可扩展的区块链系统，必须从一开始就仔细选择其基因组。 如果系统要支持将来在部署时未知的某些其他特定功能，那么它应该从一开始就支持 “异构” 工作链（具有可能不同的规则）。 为了使系统真正可扩展，它必须从一开始就支持分片；只有当系统 “紧密耦合” 时（参见 2.8.14），分片才有意义，因此这反过来意味着存在主链，快速的区块链间消息系统，BFT PoS 的使用等等。

当考虑到所有这些影响时，为 TON 区块链项目做出的大多数设计选择看起来都很自然，这几乎是唯一可能的选择。

2.9 与其他区块链项目的比较

我们通过尝试在包含现有和建议的区块链项目的地图上找到它的位置来结束我们对 TON Blockchain 及其最重要和独特特征的讨论。 我们使用 2.8 中描述的分类标准以统一的方式讨论不同的区块链项目，并构建这样的 “区块链项目图”。我们将此地图表示为表 1，然后分别简要讨论几个项目，以指出它们可能不适合一般方案的特性。

2.9.1 Bitcoin https://bitcoin.org/。

比特币（2009）是第一个也是最着名的区块链项目。 这是一个典型的第一代区块链项目：它是单链的，它使用 “POW” 和 “longest-fork-wins”fork 选择算法，它没有 Turing-complete 脚本语言（但是，支持没有循环的简单脚本）。 Bitcoin 区块链没有帐户的概念；它使用 UTXO（Unspent Transaction Output）模型。

2.9.2 Ethereum https://ethereum.org/。

以太坊（2015）是第一个支持图灵完全智能合约的区块链。 因此，它是典型的第二代项目，也是其中最受欢迎的项目。 它使用了工作量证明。

2.9.3 NXT https://nxtplatform.org/。

NXT（2014）是第一个基于 PoS 的区块链和货币。 它仍然是单链的，没有智能合约支持。

2.9.4 Tezos https://www.tezos.com/。

Tezos（2018 年或更晚）是一个基于 PoS 的单区块链项目。我们在这里提到它是因为它的独特功能：它的块解释函数 ev_block（参见 2.2.6）不是固定的，而是由 OCaml 模块决定的，可以通过将新版本提交到区块链中来升级（并收集一

些对拟议的变更投票）。通过这种方式，人们将能够通过首先部署“vanilla”Tezos 区块链，然后逐步改变所需方向的块解释功能来创建自定义单链项目，而无需任何硬分叉。

这个想法虽然很有趣，却有明显的缺点，它禁止在其他语言（如 C ++）中进行任何优化实现，因此基于 Tezos 的区块链注定会降低性能。我们认为通过发布所提出的块解释函数 ev_trans 的正式规范可能已经获得了类似的结果，而没有修复特定的实现。

2.9.5 Casper

Casper 是即将推出的以太坊 PoS 算法； 如果成功，它将逐步部署，将以太坊改为单链 PoS 或混合 PoW + PoS 系统，并提供智能合约支持，将以太坊转变为第三代项目。

2.9.6 BitShares https://bitshares.org。

BitShares（2014）是基于分布式区块链的交换的平台。 它是一个没有智能合约的异构多块链路 DPoS 系统；假设区块链状态适合内存，它只允许一小组预定义的专用事务分片实现其高性能，这些分片可以在 C++ 中有效实现。 它也是第一个使用委托证明（DPoS）的区块链项目，至少在某些特殊目的下证明了它的可行性。

2.9.7 EOS https://eos.io。

EOS（2018 或更高版本）是一种提议的异构多区块链 DPoS 系统，具有智能合约支持和对消息传递的一些最小支持（在 2.8.14 中描述的意义上仍然松散耦合）。这是以前成功创建 BitShares 和 Steemit 项目的同一团队的尝试，展示了 DPoS 一致性算法的优点。可扩展性将通过为需要它的项目创建专用工作链来实现（例如，分布式交换可能使用支持一组特殊优化专门交易的工作链，类似于 BitShares 所做的那样），并通过创建具有相同规则的多个工作链（在 2.8.10 中描述的类似的联盟）。这种可扩展性方法的缺点和局限性已在 2.8.5,2.8.12 和 2.8.14 有更详细地讨论 DPoS，分片，工作链之间的交互及其对区块链系统可扩展性的影响。

同时，即使一个人无法 “在区块链中创建 Facebook”（参见 2.9.13），EOS 或其他方式，我们认为可能成为一些高度专业、弱互动的分布式 dAPP，类似于 BitShares（分散交换）和 SteemIt（分散式博客平台）。

2.9.8 Polkadot https://polkadot.io/。

PolkaDot（2019 年或更高版本）是最精心设计和最详细的多线投资证明项目之一； 它的发展由以太坊联合创始人之一领导。 该项目与我们 TON Blockchain 的路线图最接近。 （事实上，我们感谢我们对 PolkaDot 项目的 “fishermen” 和 “nominators” 的术语。）
PolkaDot 是一个异构的松耦合多链 Proof-of-Stake 项目，具有拜占庭容错（BFT）共识，用于生成新块和主链（可能是外部的——如以太坊区块链）。它还使用超立方体路由，有点像（2.4.19 中所述）TON 的慢速版本。

它的独特之处在于它不仅可以创建公共区块链，还可以创建私有区块链。 这些私有区块链也可以与其他公共区块链，PolkaDot 进行其他方式的交互。

因此，PolkaDot 可能成为大规模私人区块链的平台，例如，银行财团可以将其用于快速向对方转移资金，或者用于大型公司可能用于私有区块链技术的任何其他用途。

但是，PolkaDot 没有分片支持，也没有紧密耦合。 这有点妨碍了它的可扩展性，这与 EOS 类似。 （PolkaDot 可能更好一点，因为 PolkaDot 使用 BFT PoS 而不是 DPoS。）

2.9.9 Universa https://universa.io。

我们在这里提到这个不寻常的区块链项目的唯一原因是因为它是迄今为止唯一一个通过明确引用类似于我们的无限分片范式（Infinite Sharding Paradigm）的项目（参见 2.1.2）。 它的另一个特点是它绕过了与拜占庭容错相关的所有复杂性，承诺只有项目的可信和许可合作伙伴才会被认可为验证者，因此他们永远不会提交无效的块。 这是一个有趣的决定；然而，它使区块链项目有意中心化，而区块链项目通常要避免中心化。

2.9.10 Plasma https://plasma.io。

Plasma（2019？）是另一个以太坊联合创始人的非常规区块链项目。它的目的是在不引入分片的情况下减轻以太坊的某些局限性。本质上，它是一个与以太坊分开的项目，引入了一个（异构的）工作链的层次结构，它与顶层的以太坊区块链（用作外部主链）绑定在一起。资金可以从层次结构中的任何区块链转移（从作为根的以太坊区块链开始），以及要完成的工作的描述。然后在子工作链中完成必要的计算（可能需要在树下进一步转发原始作业的部分），将结果传递出去，并收集奖励。实现一致性和验证这些工作链的问题被（支付通道启发）机制所规避，允许用户单方面将资金从行为不当的工作链中提取到其父工作链（尽管缓慢），并将其资金和工作重新分配给另一个工作链。

通过这种方式，Plasma 可能成为与以太坊区块链绑定的分布式计算平台，类似于 “mathematical co-processor”。但是，这似乎不是实现真正的通用可伸缩性的一种方法。

2.9.11 垂直领域区块链项目。

还有一些垂直领域区块链项目，例如 FileCoin（一个激励用户提供磁盘空间来存储愿意为其付费的其他用户的文件的系统），Golem（一种基于区块链的租借和借出计算平台）用于专门应用程序（如 3D 渲染）或 SONM（另一个类似的计算能力借贷项目）。这些项目在区块链组织层面上没有引入任何概念上新的东西；相反，它们是特定的区块链应用程序，可以通过在通用区块链中运行的智能合同来实现，前提是它可以提供所需的性能。因此，此类项目可能会使用现有或计划中的区块链项目之一作为其基础，例如 EOS，PolkaDot 或 TON。如果一个项目需要 “真正的” 可扩展性（基于分片），那么最好使用 TON；如

果通过定义一个自己的工作链系列来满足在 “confederated 联盟” 环境中工作的内容，为其目的明确优化，它可能会选择 EOS 或 PolkaDot。。

2.9.12 TON Blockchain。

TON（Telegram Open Network）区块链（计划于 2018 年）是本文档中描述的项目。 它被设计成第一个第五代区块链项目——即 BFT PoS-多链项目，混合同构 / 异构，支持（可分解）自定义工作链，具有本地分片支持，并且紧密耦合（特别是可以保留所有 shardchains 一致性的快速在分片间转发消息。 因此，它将是一个真正可扩展的通用区块链项目，本上可以实现在区块链中实现的任何应用程序。当通过 TON 项目的其他组件进行扩充时，其可能性将进一步扩大。

2.9.13 是否可以 “让 Facebook 上区块链”?

有时人们声称可以在区块链上建立 Facebook 规模的社交网络作为分布式应用程序。通常，这些公链项目被引用可以作为此类应用程序的“host（主机）”。

我们不能说这是技术上的不可能性。当然，需要一个具有真正分片（即 TON）的紧密耦合的区块链项目，以便这样一个大型应用程序不会工作得太慢（例如，将驻留在一个 shardchain 中的用户的消息和更新传递给驻留在另一个 shardchain 中的朋友 shardchain 有合理的延迟）。但是我们认为这不是必需的，也永远不会完成，因为价格过高。

让我们考虑 “将 Facebook 上传到区块链” 作为思想实验；任何其他类似规模的项目也可以作为一个例子。一旦将 Facebook 上传到区块链中，目前由 Facebook 服务器完成的所有操作将被序列化为某些区块链中的交易（例如，TON 的分片链），并且将由这些区块链的所有验证者执行。如果我们希望每个块收集至少 20 个验证者签名（立即或最终，如在 DPOS 系统中），则必须执行每个操作，例如，至少 20 次。类似地 Facebook 的服务器在其磁盘上保存的所有数据将保存在相应分片链的所有验证者的磁盘上（即至少有 20 个副本）。

因为验证者本质上是相同的服务器（或者可能是服务器的集群，但这不会影响这个参数的有效性），因为 Facebook 目前使用的那些服务器，我们看到与运行 Facebook 相关的总硬件费用区块链比以传统方式实施的区块链至少高 20 倍。

事实上，费用仍然会高得多，因为区块链的虚拟机比运行优化的编译代码的 “裸 CPU” 慢，并且其存储未针对特定于 Facebook 的问题进行优化。人们可以通过制作适合 Facebook 的一些特殊交易的特定工作链来部分缓解这个问题；这是 BitShares 和 EOS 实现高性能的方法，也可用于 TON 区块链。然而，一般的区块链设计本身仍会产生一些额外的限制，例如必须将所有操作注册为块中的事务，在 Merkle 树中组织这些事务，计算和检查它们的 Merkle 哈希值，以进行传播等等。

因此，一个保守的估计是，为了验证托管该规模的社交网络的区块链项目，人们需要 100 倍于与 Facebook 现在使用的性能相同的服务器。有些机构将不得不为这些服务器付费，无论是拥有分布式应用程序的公司（想象在每个 Facebook 页面上看到 700 个广告而不是 7 个）或其用户。无论哪种方式，这似乎在经济上都不可行。

我们认为，所有内容都应该上传到区块链中。例如，没有必要将用户照片保留在区块链中；在区块链中注册这些照片的哈希并将照片保存在分布式的链下存储（例如 FileCoin 或 TON 存储）中将是一个更好的主意。这就是为什么 TON 不仅仅是一个区块链项目，而是以 TON 区块链为中心的几个组件（TON P2P 网络，TON 存储，TON 服务）的集合，如第 1 章和第 4 章所述。

3 TON Networking

任何区块链不仅需要区块格式的规则、验证系统的标准，还需要用来传播新块，发送和收集事务候选等的网络协议。换句话说，每个区块链项目都必须建立专门的 P2P 网络。这个网络必须是点对点的，因为区块链项目通常需要分散，因此不能依赖中心化的服务器组并使用传统的客户端——服务器架构，例如，传统的在线银行应用程序。即使轻节点（如手机钱包 APP）必须以类似客户端——服务器的方式连接到全节点，如果其先前链接的全节点发生故障，它还可以再换另一个全节点。

虽然可以很容易地满足单链区块链（如 Bitcoin 或 Ethereum）的网络需求（需要构建一个 “random” 对等覆盖网络，并通过以下方式传播所有新块和事务候选者一个八卦协议），多区块链项目，如 TON Blockchain 的要求更高（例如，可以订阅一些 shardchains 的更新，不一定需要全部订阅）。因此，TON Blockchain 和 TON Project 的 networking 部分将在这里进行讨论。

另一方面，一旦支持了 TON Blockchain 所需的更复杂的网络协议，你就会发现它们也可以用于不一定用到与 TON Blockchain 的直接相关的事物上，它会在 TON 生态里提供更多的可能性和灵活性。

3.1 Abstract （Datagram）Network Layer 摘要（数据报）网络层；简称 ADNL

构建 TON 网络协议的基础是（TON）摘要（Datagram）网络层。 它使所有节点能够假设某些 “网络标识”，由 256 位 “抽象网络地址” 表示，并且可以使用这些 256 位网络地址进行通信（作为第一步发送数据报）以识别发送方和收件方。 人们不必担心 IPv4 或 IPv6 地址，UDP 端口号等；，它们会被抽象网络层隐藏。

3.1.1 抽象网络地址。

抽象网络地址或抽象地址，或简称地址，是 256 位整数，基本上等于 256 位 ECC 公钥。可以任意生成此公钥，从而创建与节点喜欢的一样多的不同网络标识。但是，必须知道相应的私钥才能接收（和解密）用于这种地址的消息。

实际上，地址本身并不是公钥；相反，它是序列化 TL 对象（参见 2.2.5）的 256 位哈希（Hash = sha256），可以根据其构造函数（前四个字节）描述几种类型的公钥和地址。在最简单的情况下，这个序列化的 TL 对象只包含一个 4

字节的幻数和一个 256 位的椭圆曲线加密（ECC）公钥；在这种情况下，地址将等于这个 36 字节结构的哈希。但是，可以使用 2048 位 RSA 密钥或任何其他公钥加密方案。

当节点知晓另一个节点的抽象地址时，它还必须接收其 “preimage”（即序列化的 TL 对象，其哈希等于该抽象地址），否则它将无法加密并将数据报发送到该地址。

3.1.2 较低级别的网络。UDP 实现。

从几乎所有 TON Networking 组件的角度来看，唯一存在的是能够（不可靠）将数据报从一个抽象地址发送到另一个抽象地址的网络（the Abstract Datagram Networking Layer；ADNL）。 原则上，抽象数据报网络层（ADNL）可以在不同的现有网络技术上实现。 但是，我们将在 IPv4 / IPv6 网络（例如 Internet 或 Intranet）中通过 UDP 实现它，如果 UDP 不可用，则使用可选的 TCP 回退。

3.1.3 基于 UDP 简单的 ADNL 的例子

从发送者的抽象地址向任何其他抽象地址（具有已知的原像）发送数据报的最简单情况可以如下实现。

假设发送方以某种方式知道拥有目的地抽象地址的接收方的 IP 地址和 UDP 端口，并且接收方和发送方都使用从 256 位 ECC 公钥派生的抽象地址。

在这种情况下，发送者只是通过其 ECC 签名（使用其私钥完成）及其源地址（或源地址的原像，如果接收者还不知道尚未知道该原像）来增加要发送的数据报。结果使用收件人的公钥加密，嵌入 UDP 数据报并发送到收件人的已知 IP 和端口。由于 UDP 数据报的前 256 位包含收件人的抽象地址，因此收件人可以识别应使用哪个私钥来解密数据报的其余部分。只有在那之后才会发现发件人的身份。

3.1.4 安全性较低的方式，发件人的地址为明文。

有时，当接收方和发送方的地址以明文形式保存在 UDP 数据报中时，安全性较低的方案就足够了； 使用 ECDH（椭圆曲线 Diffie-Hellman）将发送方的私钥和接收方的公钥组合在一起，生成一个 256 位的共享密钥，随后使用该密钥以及随机的 256 位随机数。加密部分用于导出用于加密的 AES 密钥。 例如，可以通过在加密之前将原始明文数据的哈希连接到明文来提供完整性。

这种方法的优点是，如果预计在两个地址之间交换多个 datagram，则只能计算一次共享密钥，然后进行高速缓存； 然后，加密或解密下一个 datagram 将不再需要较慢的椭圆曲线操作。

3.1.5 通道和通道标识符。

在最简单的情况下，携带嵌入式 TON ADNL 数据报的 UDP 数据报的前 256 位将等于接收者的地址。它们构成了通道标识符。有不同类型的通道。其中一些是点对点的；它们是由希望在未来交换大量数据的双方创建的，并通过运行经

典或椭圆曲线 Diffie-Hellman（如果额外的话）交换如 3.1.3 或 3.1.4 中所述加密的几个数据包来生成共享密钥。 安全性是必需的，或者只是由一方产生随机共享密钥并将其发送给另一方。

在此之后，从共享密钥中结合一些附加数据（例如发送者和接收者的地址），例如通过哈希，导出信道标识符，并且该标识符被用作携带用数据加密的数据的 UDP 数据报的前 256 位。

3.1.6 通道作为通道标识符。

通常，“通道” 或 “通道标识符” 仅选择处理接收器已知的入站 UDP 数据报的方式。如果通道是接收方的抽象地址，则按照 3.1.3 或 3.1.4 的规定进行处理；如果通道是 3.1.5 中讨论的已建立的点对点信道，则处理包括借助于共享密钥解密 datagram。

特别地，当直接接收者简单地将所接收的消息转发给其他人——实际接收者或另一个代理时，通道标识符实际上可以选择 “tunnel”。一些加密或解密步骤（让人联想到 “onion routing”[6] 甚至 “garlic routing”31）可能会在此过程中完成，而另一个通道标识符可能会用于重新加密的转发数据包（例如，同行——可以使用对等信道将分组转发到路径上的下一个接收者。

通过这种方式，可以在 TON 抽象数据报网络层的层面上添加一些对 “tunneling” 和 “proxying” 的支持——与 TOR 或 I^2P 项目提供的相似——而不会影响到 TON 抽象数据报网络层的功能。所有更高级别的 TON 网络协议，这些协议都是不可知的。 这是借用了 TON Proxy service（参见 4.1.11）。

3.1.7 零通道和引导问题。

通常，TON ADNL 节点将具有一些 “neighbor table”，其包含关于其他已知节点的信息，例如它们的抽象地址及其预映像（即公钥）及其 IP 地址和 UDP 端口。然后它将通过使用从这些已知节点获取的信息作为特殊查询的答案逐渐扩展此表，并且有时会有修剪的记录。

但当 TON ADNL 节点刚启动时，可能发生它不知道的其他节点，并且只知道节点的 IP 地址和 UDP 端口，而不知道它的抽象地址。例如，如果轻节点无法访问任何先前缓存的节点以及任何硬编码到软件中的节点，并且必须要求用户输入节点的 IP 地址或 DNS 域，则会通过 DNS 发生这种情况。

在这种情况下，节点将数据包发送到相关节点的特殊 “zero channel”。这不需要知道收件人的公钥（但消息仍应包含发件人的身份和签名），因此消息无需加密即可传输。它通常应该仅用于获取接收器的身份（可能是为此目的而创建的一次性身份），然后以更安全的方式开始通信。

一旦知道至少一个节点，就很容易通过更多条目填充 “neighbor table” 和 “routing table”，从发送到已知节点的特殊查询的答案中知道它们。

并非所有节点都需要处理发送到零通道的数据报，但那些用于做轻节点的人应该支持这个功能。

3.1.8 ADNL 上类似 TCP 的流协议。

ADNL 是基于 256 位抽象地址的不可靠（小尺寸）数据报协议，可用作更复杂网络协议的基础。 例如可以使用 ADNL 作为 IP 的抽象替代来构建类似 TCP 的流协议。 但是 TON 项目的大多数组件都不需要这样的流协议。

3.1.9 RLDP，或 ADNL 上的可靠大数据报协议。

使用基于 ADNL 的可靠的任意大小的数据报协议（称为 RLDP）代替类似 TCP 的协议。 例如，可以使用这种可靠的数据报协议将 RPC 查询发送到远程主机并通过它们接收回复（参见 4.1.5）。

3.2 TON DHT：类似 Kademlia 的分布式哈希表

TON 分布式哈希表（DHT）在 TON 项目的网络部分中起着至关重要的作用，用于定位网络中的其他节点。 例如，想要将事务提交到分片链的客户端可能想要找到该 shardchain 的验证者或 collator，或者至少某个可能将客户端的事务中继到 collator 的节点。 这可以通过在 TON DHT 中查找特殊键来完成。 TON DHT 的另一个重要应用是它可以用来快速填充新节点的邻居表（参见 3.1.7），只需查找随机密钥或新节点的地址即可。 如果节点对其入站数据报使用代理和隧道，则它在 TON DHT 中发布隧道标识符及其入口点（例如，IP 地址和 UDP 端口）； 那么希望将数据报发送到该节点的所有节点将首先从 DHT 获得该联系信息。

TON DHT 是类似 Kademlia 的分布式哈希表系列的成员。

3.2.1 覆盖网络。

覆盖（子）网络是一种大型网络里的简单的（网络）。 通常，只有较大网络的一些节点参与覆盖子网，并且这些节点（物理或虚拟）之间只有一些 “链路” 是覆盖子网的一部分。

通过这种方式，如果包含网络被表示为图形（在数据报网络的情况下可以使用完整的图形，例如 ADNL，任何节点都可以轻松地与任何其他节点通信），覆盖子网络是这个的子图 图形。

在大多数情况下，覆盖网络使用基于较大网络的网络协议构建的一些协议来实现。 它可以使用与较大网络相同的地址，或使用自定义地址。

3.2.2 DHT 的值。

分配给这些 256 位密钥的值基本上是有限长度的任意字节串。这种字节串的解释由相应密钥的原像决定；通常查找密钥的节点和存储密钥的节点知道。

3.2.3 DHT 的节点。半永久性网络身份。

TON DHT 的键值映射保留在 DHT 的节点上——基本上是 TON 网络的所有成员。为此，除了 3.1.1 中描述的任何数量的短暂和永久抽象地址之外，TON 网

络的任何节点（可能除了一些非常轻的节点）至少具有一个 “semi-permanent address”，其中将其标识为 TON DHT 的成员。这个 semi-permanent address 或 DHT 地址不应该经常更改，否则其他节点将无法找到他们正在寻找的密钥。如果节点不想显示其 “真实” 标识，则它会生成一个单独的抽象地址，仅用于参与 DHT。但是，此抽象地址必须是公共的，因为它将与节点的 IP 地址和端口相关联。

3.2.4 Kademlia 距离。

现在我们有 256 位密钥和 256 位（半永久）节点地址。 我们在 256 位序列集上引入所谓的 XOR 距离或 Kademlia 距离 dK，由下式给出

$$dK\ (x,y) := \ (x\ \oplus\ y)\ interpreted\ as\ an\ unsigned\ 256-bit\ integer \quad (25)$$

这里 x⊕y 表示相同长度的两个比特序列的按位异或（或 XOR）。
Kademlia 距离在所有 256 位序列的集合 2256 上引入度量。 特别是，当且仅当 dK(x,y) = 0 if and only if x = y, dK(x,y) = dK (y, x), and dK (x, z) ≤ dK (x, y) + dK (y, z). 另一个重要特性是在距 x 的任何给定距离处只有一个点：dK (x, y) = dK (x, y′) implies y = y′。

3.2.5 类似 Kademlia 的 DHTs 和 TON DHT。

我们说具有 256 位密钥和 256 位节点地址的分布式哈希表（DHT）是类似 Kademlia 的 DHT，如果期望将 k Kademlia-最近节点上的密钥 K 的值保持为 K（即，从其地址到 K 的 Kademlia 距离最小的 s 节点。）
这里是一个小参数，比如 s = 7，需要提高 DHT 的可靠性（如果我们只将密钥保存在一个节点上，最接近 K 的那个节点，那个密钥的值如果只有那个节点就会丢失离线）。

根据这个定义，TON DHT 是类似 Kademlia 的 DHT。 它是通过 3.1 中描述的 ADNL 协议实现的。

3.2.6 Kademlia 路由表。

参与 Kademlia-like DHT 的每个节点通常都维护一个 Kademlia 路由表。在 TON DHT 的情况下，它由 n = 256 个桶组成，编号从 0 到 n-1。第 i 个桶将包含关于一些已知节点的信息（固定数量 t 的 “最佳” 节点，可能还有一些额外的候选者）位于距离节点地址 2i to 2i+1 − 1 的 Kademlia 距离。这个信息包括它们的（semi-permanent）地址，IP 地址和 UDP 端口，以及一些可用性信息，如时间和延迟最后一次 ping。

当 Kademlia 节点作为某个查询的结果获知任何其他 Kademlia 节点时，它将其包含在其路由表的合适桶中，首先作为候选者。然后，如果该桶中的一些 “最佳” 节点失败（例如，长时间不响应 ping 查询），则可以用一些候选者替换它们。通过这种方式，Kademlia 路由表保持填充状态。

来自 Kademlia 路由表的新节点也包含在 3.1.7 中描述的 ADNL 邻居表中。如果经常使用来自 Kademlia 路由表的桶中的 “best” 节点，则可以建立 3.1.5 中描述的意义上的信道以便于加密 datagram。

TON DHT 的一个特殊功能是它尝试选择往返延迟最小的节点作为 Kademlia 路由表的桶的 “best” 节点。

3.2.7 （Kademlia 网络查询。）Kademlia 节点通常支持以下网络查询：

- Ping——检查节点可用性。
- Store(key,value)——要求节点将值保持为键值的值。 对于 TON DHT，商店查询稍微复杂一些（参见 3.2.9）。
- Find_Node(key,l)——要求节点将 l Kademlia 最近的已知节点（从其 Kademlia 路由表）返回到 key。
- Find_Value(key, l)——与上面相同，但如果节点知道与键键对应的值，则只返回该值。

当任何节点想要查找键 K 的值时，它首先创建 s' 节点的集合 S（对于 s' 的一些小值，比如 s'= 5），相对于 Kademlia 距离最接近 K. 在所有已知节点中（它们来自 Kademlia 路由表）。 然后向它们中的每一个发送 Find*Value 查询，并且在它们的答案中提到的节点包括在 S 中。如果之前没有这样做，那么来自 S 的 s' 节点（最接近 K）也被发送 Find*Value 查询，并且该过程一直持续到找到值或集合 S 停止增长。 这是关于 Kademlia 距离最接近 K 的节点的一种 “beam search”。

如果要设置某个密钥 K 的值，则对 s'≥s 运行相同的过程，使用 Find*Node 查询而不是 Find*Value 来查找距离 K 最近的节点。之后，将 Store 查询发送给所有这些节点。

在类似 Kademlia 的 DHT 的实现中有一些不太重要的细节（例如，任何节点都应该查找最近的节点，比如每小时一次，并通过 Store 查询重新发布所有存储的密钥）。 我们暂时会忽略它们。

3.2.8 引导 Kademlia 节点。

当 Kademlia 节点联机时，它首先通过查找自己的地址来填充其 Kademlia 路由表。 在此过程中，它标识自身最近的节点。 它可以从它们中下载所有已知的 (key,value) 对，以填充其 DHT 的一部分。

3.2.9 在 TON DHT 中存储值。

在 TON DHT 中存储值与一般的 Kademlia 样 DHT 略有不同。 当有人希望存储一个值时，她必须不仅要向 Store 查询提供密钥 K 本身，还要提供其 preimage- 即 TL 序列化的字符串（开头有几个预定义的 TL 构造函数之一），其中包含 “ 描述 “的关键。 稍后由节点保存此密钥描述以及密钥和值。

密钥描述描述了所存储对象的 “字段”，其 “owner” 以及未来更新时的 “update rules”。所有者通常通过密钥描述中包含的公钥来标识。如果包含它，通常只接受由相应私钥签名的更新。存储对象的 “字段” 通常只是一个字节字符串。但是，在某些情况下，它可能更复杂——例如，输入隧道描述（参见 3.1.6）或节点地址的集合。

“update rules” 也可以不同。在某些情况下，如果新值由所有者签名，它们只允许用新值替换旧值（签名必须保留为值的一部分，以后在获取值后由任何其他节点检查）这个密钥。在其他情况下，旧值会以某种方式影响新值。例如，它可以包含序列号，只有在新序列号较大时才会覆盖旧值（以防止重放攻击）。

3.2.10 TON DHT 中的分布式 “流追踪器” 和 “网络兴趣小组”。

另一个有趣的情况是，当值包含节点列表时——可能包含其 IP 地址和端口，或仅包含其抽象地址——并且 “update rule” 包括将请求者包括在此列表中，前提是她可以确认她的身份。

该机制可以用于创建分布式 “流追踪器”，其中对某个 “流”（文件）感兴趣的所有节点可以找到对相同的流感兴趣或已经具有副本的其他节点。

TON 存储（参见 4.1.8）使用该技术来查找具有所需文件副本的节点（例如，分片链状态或旧块的快照）。但是，更重要的用途是创建 “overlay multicast subnetworks” 和 “网络兴趣小组”（参见 3.3）。这个想法是只有一些节点对特定 shardchain 的更新感兴趣。如果分片链的数量变得非常大，甚至找到对同一分片感兴趣的一个节点可能变得复杂。这种 “分布式流追踪器” 提供了一种查找这些节点的便捷方法。另一种选择是从验证者请求它们，但这不是一种可扩展的方法，验证者可能选择不响应来应对这些未知节点的查询。

3.2.11 后退键。

到目前为止描述的大多数 “key type” 在其 TL 描述中具有额外的 32 位整数字段，通常等于零。 但是，如果无法从 TON DHT 中检索或更新通过哈希描述获得的密钥，则增加该字段中的值，并进行新的尝试。 以这种方式，通过在受到攻击的密钥附近创建许多抽象地址并控制相应的 DHT 节点，不能 “捕获” 和 “审查” 密钥（即执行密钥保留攻击）。

3.2.12 定位服务。

位于 TON 网络中并通过 3.1 中描述的 TON ADNL（基于其构建的更高级别协议）提供的某些服务可能希望在某处发布其抽象地址，以便其客户知道在哪里找到它们。

但是，在 TON 区块链中发布服务的抽象地址可能不是最好的方法，因为抽象地址可能需要经常更改，并且因为为了可靠性或负载平衡目的而提供多个地址是有意义的。

另一种方法是将公钥发布到 TON 区块链中，并使用一个特殊的 DHT 密钥，指示公钥作为 TL 描述字符串中的 “owner”（参见 2.2.5），以发布最新的列表。服务的抽象地址。 这是 TON Services 利用的方法之一。

3.2.13 找到 TON 区块链账户的所有者。

在大多数情况下，TON 区块链帐户的所有者不希望与抽象网络地址相关联，尤其是 IP 地址，因为这可能会侵犯他们的隐私。但是，在某些情况下，TON 区块链帐户的所有者可能希望发布可以联系她的一个或多个抽象地址。

典型的情况是 TON 支付 “lightning network” 中的节点（参见 5.2），即小额加密货币支付平台。公共 TON 支付节点可能不仅希望与其他对等方建立支付信道，而且还要发布可用于在稍后时间联系它以便沿已建立的支付通道的抽象网络地址。

一种选择是在创建支付通道的智能合约中包含抽象网络地址。更灵活的选择是在智能合约中包含公钥，然后按照 3.2.12 中的说明使用 DHT。

最自然的方法是使用控制 TON 区块链中的帐户的相同私钥来签署和发布 TON DHT 中与该帐户关联的抽象地址的更新。这几乎与 3.2.12 中描述的方式相同；但是，使用的 DHT 密钥需要一个特殊的密钥描述，只包含 account_id 本身，等于 “account description” 的 sha256，它包含帐户的公钥。包含在此 DHT 密钥值中的签名也将包含帐户描述。

通过这种方式，建立可以获得用于定位 TON 区块链账户的一些所有者的抽象网络地址的机制。

3.2.14 找到抽象地址。

请注意，TON DHT 虽然通过 TON ADNL 实现，但它本身也被 TON ADNL 用于多种用途。

其中最重要的是从 256 位抽象地址开始定位节点或其联系人数据。这是必要的，因为即使没有提供其他信息，TON ADNL 也应该能够将数据报发送到任意 256 位抽象地址。

为此，将 256 位抽象地址简单地查找为 DHT 中的密钥。找到具有该地址的节点（即，使用该地址作为公共半永久 DHT 地址），在这种情况下，可以知晓其 IP 地址和端口；或者，可以检索输入隧道描述作为所讨论的密钥的值，由正确的私钥签名，在这种情况下，该隧道描述将用于将 ADNL 数据报发送到预期的接收者。

请注意，为了使抽象地址 “public”（可从网络中的任何节点到达），其所有者必须将其用作半永久性 DHT 地址，或者发布（在 DHT 密钥中等于所考虑的抽象地址））输入隧道描述，其另一个公共抽象地址（例如，半永久地址）作为隧道的入口点。另一种选择是简单地发布其 IP 地址和 UDP 端口。

3.3 Overlay Networks（覆盖网络） 和 Multicasting Messages（多播消息）

在像 TON 区块链这样的多区块链系统中，甚至完整节点通常也只对某些分片链获得更新（即新块）感兴趣。 为此，必须在 TON 网络内部建立一个特殊的覆盖（子）网络，在 3.1 中讨论的 ADNL 协议之上，每个 shardchain 一个。

因此，需要构建对任何愿意参与的节点开放的任意覆盖子网。 基于 ADNL 的特殊八卦协议将在这些覆盖网络中运行。 特别是，这些八卦协议可用于在这种子网内传播（广播）任意数据。

3.3.1 覆盖网络。

覆盖（子）网络简单地是在一些较大网络内实现的（虚拟）网络。 通常，只有较大网络的一些节点参与覆盖子网，并且这些节点（物理或虚拟）之间只有一些 “links” 是覆盖子网的一部分。

通过这种方式，如果包含网络被表示为图形（在数据报网络的情况下可以使用完整的图形，例如 ADNL，任何节点都可以轻松地与任何其他节点通信），覆盖子网络是这个的子图 图形。

在大多数情况下，覆盖网络使用基于较大网络的网络协议构建的一些协议来实现。 它可以使用与较大网络相同的地址，或使用自定义地址。

3.3.2 在 TON 中覆盖网络。

TON 中的覆盖网络建立在 3.1 中讨论的 ADNL 协议之上；它们也使用 256 位 ADNL 抽象地址作为覆盖网络中的地址。每个节点通常选择其抽象地址之一作为其在覆盖网络中的地址加倍。

与 ADNL 相比，TON 覆盖网络通常不支持将数据报发送到任意其他节点。相反，在一些节点之间建立一些 “semipermanent links”（相对于所考虑的覆盖网络称为 “neighbors”），并且消息通常沿着这些链路（即从节点到其邻居之一）转发。以这种方式，TON 覆盖网络是 ADNL 网络的（完整）图形内的（通常不是满的）子图。

可以使用专用的对等 ADNL 信道来实现与 TON 覆盖网络中的邻居的链路（参见 3.1.5）。

覆盖网络的每个节点维护一个邻居列表（与覆盖网络相关），包含它们的抽象地址（用于在覆盖网络中识别它们）和一些链路数据（例如，ADNL 信道用于与他们沟通）。

3.3.3 私人和公共覆盖网络。

一些覆盖网络是公共的，这意味着任何节点都可以随意加入它们。 其他是私有的，意味着只允许某些节点被接纳（例如，那些可以证明其身份为验证者的节点）一些私有覆盖网络甚至可以为 “general public” 所知。 关于这种覆盖网络的

信息仅对某些可信节点可用；例如，它可以使用公钥加密，并且只有具有相应私钥副本的节点才能解密此信息。

3.3.4 中央控制的覆盖网络。

一些覆盖网络由一个或多中心化控制，或由一些广为人知的公钥的所有者集中控制。 其他的是分散的，这意味着没有特定的节点负责它们。

3.3.5 加入覆盖网络。

当一个节点想要加入一个覆盖网络时，它首先必须知道它的 256 位网络标识符，通常等于覆盖网络描述的 sha256——一个 TL 序列化对象（参见 2.2.5），它可能包含例如，覆盖网络的中心权威（即其公钥，可能是抽象地址）具有覆盖网络名称的字符串，如果这是与之相关的覆盖网络，则为 TON 区块链分片标识符等等。

有时可以从网络标识符开始恢复覆盖网络描述，只需在 TON DHT 中查找即可。 在其他情况下（例如，对于私有覆盖网络），必须获得网络描述以及网络标识符。

3.3.6 找到覆盖网络的一个成员。

在节点获知其想要加入的覆盖网络的网络标识符和网络描述之后，它必须至少找到属于该网络的一个节点。

对于不想加入覆盖网络但只想与之通信的节点，也需要这样做；例如，可能存在专用于收集和传播特定 shardchain 的事务候选的覆盖网络，并且客户端可能想要连接到该网络的任何节点以建议事务。

用于定位覆盖网络的成员的方法在该网络的描述中定义。有时（特别是对于专用网络），必须已经知道能够加入的成员节点。在其他情况下，一些节点的抽象地址包含在网络描述中。更灵活的方法是在网络描述中仅指示负责网络的中央机构，然后通过由该中央机构签名的某些 DHT 密钥的值来获得抽象地址。

最后，真正分散的公共覆盖网络可以使用 3.2.10 中描述的 “分布式流追踪器” 机制，也可以在 TON DHT 的帮助下实现。

3.3.7 找到覆盖网络的更多成员。

创建链接。 一旦找到覆盖网络的一个节点，就可以向该节点发送特殊查询，请求其他成员的列表，例如，被查询的节点的邻居，或其随机选择。

这使得加入成员能够通过选择一些新知晓的网络节点并建立到它们的链接（即如所概述的专用 ADNL 点对点信道）来填充关于覆盖网络的 “adjacency” 或 “neighbor list”。 在 3.3.2）。 之后，将向所有邻居发送特殊消息，指示新成员已准备好在覆盖网络中工作。 邻居包括他们在邻居列表中的新成员的链接。

3.3.8 维护邻居列表。

覆盖网络节点必须不时更新其邻居列表。 一些邻居，或者至少是他们的链接（频道）可能会停止响应； 在这种情况下，必须将这些链接标记为“suspended”，必须尝试重新连接到此类邻居，并且如果这些尝试失败，则必须销毁链接。

另一方面，每个节点有时会从随机选择的邻居请求其邻居列表（或其随机选择），并通过向其添加一些新发现的节点来使用它来部分更新其自己的邻居列表，以及 删除一些旧的，无论是随机还是取决于它们的响应时间和数据报丢失统计信息。

3.3.9 覆盖网络是一个随机子图。

覆盖网络是 ADNL 网络里的随机子图。 如果每个点的度数至少为 3（如果每个节点连接到至少三个邻居），则已知该随机图以几乎等于 1 的概率连接。 更准确地说，n 个顶点断开的随机图的概率是指数小的，如果 n≥20，则可以完全忽略这个概率。（当然，这不适用于全局网络分区的情况， 当不同分片间的节点没有机会互相了解时）在另一方面，如果 n 小于 20，则要求每个顶点具有例如至少十个邻居就足够了。

3.3.10 TON 覆盖网络经过优化，可降低延迟。

TON 覆盖网络优化了由以前的方法生成的 “random” 网络图。每个节点都尝试以最小的往返时间保留至少三个邻居，很少改变这个 “fast neighbors” 列表。同时，它还至少有三个完全随机选择的 “slow neighbors”，因此覆盖网络图总是包含一个随机子图。这是保持连接性并防止将覆盖网络分成几个未连接的区域子网所必需的。还选择并保留至少三个 “intermediate neighbors”，其具有中间往返时间，以一定常数（实际上是快速和慢速邻居的往返时间的函数）为界。

通过这种方式，覆盖网络的图形仍然优化了更低的延迟和更高的吞吐量，以保持足够优秀的连接性。

3.3.11 覆盖网络中的八卦协议。

覆盖网络通常用于运行所谓的八卦协议之一，其实现一些全局目标，同时让每个节点仅与其邻居交互。 例如，有一些八卦协议可以构建一个（not too large）覆盖网络的所有成员的近似列表，或者计算一个（arbitrarily large）覆盖网络的成员数量的估计，仅使用每个节点定量的存储器。

3.3.12 将网络覆盖为广播域。

在覆盖网络中运行的最重要的八卦协议是广播协议（broadcast protocol,），旨在将由网络的任何节点或者可能由指定的发送方节点之一生成的广播消息传播到所有其他节点。

实际上有几种广播协议，针对不同的用例进行了优化。最简单的接收新广播消息并将它们中继到尚未自己发送该消息副本的所有邻居。

3.3.13 更复杂的广播协议。

某些应用可能需要更复杂的广播协议。 例如对于广播大小的广播消息，向邻居发送不是新接收的消息本身，而是发送其哈希（或新消息的哈希集合）是有意义的。 在知道了先前看不见的消息哈希之后，邻居可以自己请求消息，例如使用 3.1.9 中讨论的可靠的大数据报协议（RLDP）进行传输。 这样，新消息将仅从一个邻居下载。

3.3.14 检查覆盖网络的连接性。

如果存在必须在该覆盖网络中的已知节点（例如覆盖网络的 “owner” 或 “creator”），则可以检查覆盖网络的连接性。然后，所讨论的节点不时地广播包含当前时间，序列号及其签名的短消息。任何其他节点可以确定它仍然连接到覆盖网络，如果它不久前已经收到这样的广播。该协议可以扩展到几个众所周知的节点的情况；例如它们都将发送此类广播，并且所有其他节点将期望从超过一半的众所周知的节点接收广播。

在用于传播特定分片链的新块（或仅新块头）的覆盖网络的情况下，节点检查连接性的好方法是追踪到目前为止接收的最新块。因为块通常每五秒生成一次，如果没有收到超过 30 秒的新块，则该节点可能已经与覆盖网络断开连接。

3.3.15 流媒体广播协议。

最后，还有一个用于 TON 覆盖网络的流式广播协议，例如用于在某些 shardchain（“shardchain 任务组”）的验证者中预测块候选者，当然为此目的创建一个私有覆盖网络。 可以使用相同的协议将新的 shardchain 块传播到该 shardchain 的所有完整节点。

该协议已在 2.6.10 中概述：新的（大）广播消息被分成例如 N 个一千字节的块；通过诸如 Reed-Solomon 或喷泉代码（例如，RaptorQ 代码）之类的纠删码将这些块的序列增加到 M≥N 块，并且这些 M 块被流式传输到所有块。 邻居按升序排序。 参与节点收集这些块直到它们可以恢复原始大的消息（一个必须成功接收至少 N 个块），然后指示它们的邻居停止发送流的新块，因为现在这些节点可以生成后续的块，拥有原始消息的副本。这些节点继续生成流的后续块并将它们发送到它们的邻居，除非邻居反过来表明不再需要它。

以这种方式，节点在进一步传播之前不需要完整地下载大消息。这可以最大限度地减少广播延迟，尤其是与 3.3.10 中描述的优化结合使用时。

3.3.16 基于现有的覆盖网络构建新的覆盖网络。

有时人们不想从头开始构建覆盖网络。相反已知一个或多个先前存在的覆盖网络，并且预期新覆盖网络的成员资格与这些覆盖网络的组合成员资格显着重叠。

当一个 TON shardchain 被分成两个，或者两个兄弟的 shardchains 被合并为一个时，就会出现一个重要的例子（参见 2.7）。在第一种情况下，必须为每个新的分支链构建用于将新块传播到完整节点的覆盖网络；然而，可以预期这些新的

覆盖网络中的每一个都包含在原始 shardchain 的块传播网络中（并且包括其大约一半的成员）。在第二种情况下，用于传播合并的 shardchain 的新块的覆盖网络将近似地包括与被合并的两个兄弟串联链相关的两个覆盖网络的成员的并集。

在这种情况下，新覆盖网络的描述可以包含对相关现有覆盖网络列表的显式或隐式引用。希望加入新覆盖网络的节点可以检查它是否已经是这些现有网络之一的成员，并且在这些网络中查询它们的邻居是否也对新网络感兴趣。在肯定答复的情况下，可以为这些邻居建立新的点对点信道，并且它们可以包括在新覆盖网络的邻居列表中。

这种机制并不完全取代 3.3.6 和 3.3.7 中描述的一般机制；相反，它们都是并行运行的，用于填充邻居列表。这是为了防止无意中将新的覆盖网络分成几个未连接的子网。

3.3.17 覆盖网络中的覆盖网络。

另一个有趣的案例是 TON Payments 的实施（即用于即时链下价值转移的“lightning network”；参见 5.2）。 在这种情况下，首先构建包含 “lightning network” 的所有传输节点的覆盖网络。 但是，其中一些节点已在区块链中建立了支付通道；除了 3.3.6,3.3.7 和 3.3.8 中描述的一般覆盖网络算法选择的任何 “random” 邻居之外，它们必须始终是该覆盖网络中的邻居。这些具有已建立的支付信道的邻居的 “permanent links” 用于运行特定的闪电网络协议，从而有效地在包围（最常连接）的覆盖网络内创建覆盖子网（不一定连接，如果出现问题）。

4 TON 服务和应用

我们已经详细讨论了 TON 区块链和 TON 网络技术。 现在我们解释一些可以将它们组合在一起以创建各种服务和应用程序的方法，并讨论 TON 项目本身将从一开始或稍后提供的一些服务。

4.1 TON 服务实施策略

我们首先讨论如何在 TON 生态系统内实现不同的区块链和网络相关的应用程序和服务。 首先，按顺序进行简单分类：

4.1.1 应用和服务。

我们将交替使用 “application 申请” 和 “service 服务”。 但是，存在一种微妙且有些模糊的区别：应用程序通常直接向人类用户提供某些服务，而服务通常由其他应用程序和服务利用。 例如，TON Storage 是一项服务，因为它旨在代表其他应用程序和服务保留文件，即使人类用户也可以直接使用它。 一个假设的 “区块链中的 Facebook”（参见 2.9.13）或 Telegram messenger，如果通过 TON 网络提供（即实施为 “ton-service”；参见 4.1.6），则更像是一个应用程序即使某些 “bots” 可能在没有人为干预的情况下自动访问它。

4.1.2 应用的位置：链上，链外或混合。

为 TON 生态系统设计的服务或应用程序需要保留其数据并将数据处理到某处。这导致以下应用程序（和服务）的分类：

- **链上应用程序**（参见 4.1.4）：所有数据和处理都在 TON 区块链中。
- **链下应用程序**（参见 4.1.5）：所有数据和处理都在 TON 区域链之外，在通过 TON 网络提供的服务器上。
- **混合应用程序**（参见 4.1.7）：部分（但不是全部）数据和处理都在 TON 区块链中； 其余的是通过 TON 网络提供的链下服务器。

4.1.3 中心化：中心化和去中心化或分布式应用程序。

另一个分类标准是应用程序（或服务）是依赖于中心化服务器集群，还是真正 “分布式”（参见 4.1.9）。 所有链上应用程序都自动分散和分发。 链下和混合应用可能表现出不同程度的中心化。现在让我们更详细地考虑上述可能性。

4.1.4 纯粹的 “链上” 应用程序：驻留在区块链中的分布式应用程序或 “dapps”。

4.1.2 中提到的一种可能的方法是在 TON 区块链中完全部署 “分布式应用程序”（通常简称为 “dapp”），作为智能合约或智能合约的集合。所有数据将作为这些智能合约的永久状态的一部分保留，并且与项目的所有交互将通过发送到这些智能合约或从这些智能合约接收的（TON 区块链）消息来完成。

我们已经在 2.9.13 中讨论过这种方法有其缺点和局限性。它也有它的优点：这样的分布式应用程序不需要运行服务器或存储其数据（它在 “区块链” 中运行——即在验证者的硬件上运行），并享有区块链极高（拜占庭）的可靠性和可访问性。这种分布式应用程序的开发人员不需要购买或租用任何硬件；他需要做的就是开发一些软件（即智能合约的代码）。在那之后，她将有效地从验证者那里租用计算能力，并以 Grams 的形式支付它，无论是她自己还是用户买单。

4.1.5 纯粹的网络服务：“ton-sites” 和 “ton-services”。

另一个极端的选择是在某些服务器上部署服务，并通过 3.1 中描述的 ADNL 协议将其提供给用户，也可以使用一些更高级别的协议，如 3.1.9 中讨论的 RLDP，可用于发送 RPC 查询 以任何自定义格式提供服务并获取这些查询的答案。 通过这种方式，该服务将完全链下，并将驻留在 TON 网络中，几乎不使用 TON 区块链。

TON 区块链可能仅用于定位服务的抽象地址或地址，如 3.2.12 中所述，可能借助于诸如 TON DNS（参见 4.3.1）之类的服务来促进域的翻译——像人类可读的字符串到抽象地址。

在某种程度上，ADNL 网络（即 TON 网络）类似于隐形互联网项目（I^2P），这种（几乎）纯粹的网络服务类似于所谓的 “eep-services”（即具有 I^2P -地址作为其入口点，并通过 I^2P 网络提供给客户端）。我们会说，驻留在 TON 网络中的这种纯粹的网络服务是 “ton-services”。

“eep-service” 可以实现 HTTP 作为其客户端——服务器协议；在 TON 网络环境中，“ton-services” 可能只是使用 RLDP（参见 3.1.9）数据报来传输 HTTP 查询和响应。如果它使用 TON DNS 允许通过人类可读的域名查找其抽象地址，那么对网站的类比就变得几乎完美了。甚至可以编写一个专门的浏览器，或者在用户机器上本地运行的特殊代理（“ton-proxy”），从用户使用的普通 Web 浏览器接受任意 HTTP 查询（一旦是本地 IP 地址和 TCP 端口）将代理输入到浏览器的配置中，并通过 TON 网络将这些查询转发到服务的抽象地址。然后，用户将具有类似于万维网（WWW）的浏览体验。

在 I^2P 生态系统中，这种 “eep-services” 被称为 “eep-sites”。人们也可以在 TON 生态系统中轻松创建 “ton-sites”。这在某种程度上得益于 TON DNS 等服务的存在，TON DNS 利用 TON 区块链和 TON DHT 将（TON）域名转换为抽象地址。

4.1.6 Telegram Messenger 作为 ton-service；MTProto 覆盖 RLDP。

顺便提提一嘴，Telegram Messenger 用于客户端——服务器交互的 MTProto 协议可以很容易地嵌入到 3.1.9 中讨论的 RLDP 协议中，从而有效地将电报转换为 ton-service。 由于 TON Proxy 技术可以为 ton-site 或 ton-service 的最终用户透明地开启，实施的程度低于 RLDP 和 ADNL 协议（参见 3.1.6），这将有效的激活 Telegram。当然，其他消息和社交网络服务也可能从这项技术中受益。

4.1.7 混合服务：部分链外，部分链上。

有些服务可能采用混合方法：做大部分链下处理，但也有一些链上部分（例如向用户注册他们的义务，反之亦然）。通过这种方式，部分状态仍将保留在 TON 区块链中（即不可变的公共分类帐），并且服务或其用户的任何不当行为都可能受到智能合同的惩罚。

4.1.8 示例：保持文件在链下；TON Storage。

TON Storage 提供了这种服务的一个例子。在最简单的形式中，它允许用户通过保持链上只存储要存储的文件的哈希值来链接存储文件，并且可能是一个智能合约，其中一些其他方同意在给定时间段内保留有问题的文件预先商定的费用的时间。实际上，文件可以被细分为一些小尺寸（例如，1 千字节）的块，通过诸如 Reed-Solomon 或喷泉代码的擦除代码来增强，可以为增强的块序列构造 Merkle 树哈希。，这个 Merkle 树哈希可以在智能合约中发布，而不是与文件的通常哈希一起发布。这有点让人联想到文件存储在 torrent 中的方式。

一种更简单的存储文件形式是完全链下的：一个可能基本上为新文件创建一个“torrent”，并使用 TON DHT 作为此 torrent 的“分布式 torrent 追踪器”（参见 3.2.10）。对于流文件，这实际上可能非常有效。但是一个人没有获得任何可用性保证。例如，一个假设的“区块链 Facebook”（参见 2.9.13），可能会选择将其用户的个人资料照片完全脱离这种“torrent”，可能会失去普通（不是特别受欢迎的）用户的照片，或至少有可能无法长时间呈现这些照片。 TON Storage 技术主要是链下的，但使用链上智能合约来强制存储文件的可用性，可能更适合此任务。

4.1.9 分散的混合服务，或“雾服务”。

到目前为止，我们已经讨论了中心化混合服务和应用。虽然他们的链上组件以分散和分布的方式处理，但他们的链下组件依赖于服务提供商以通常的中心化方式控制的一些服务器。可以从其中一家大公司提供的云计算服务中租用计算能力，而不是使用某些专用服务器。但是，这不会导致服务的链下组件的分散化。

实现服务的链下组件的分散方法在于创建一个市场，任何拥有所需硬件并愿意租用其计算能力或磁盘空间的人都会向需要它们的人提供服务。

例如，可能存在一个注册表（也可能称为“market”或“exchange”），其中所有对保持其他用户的文件感兴趣的节点都会发布其联系信息，以及它们的可用存储容量，可用性策略和价格。需要这些服务的人可能会在那里查找，如果对方同意，则在区块链中创建智能合约并上传文件以进行离线存储。通过这种方式，像 TON Storage 这样的服务变得真正分散，因为它不需要依赖任何集中的服务器集群来存储文件。

4.1.10 “fog computing” 平台作为分散的混合服务。

当想要执行某些特定计算（例如 3D 渲染或训练神经网络）时，这种分散的混合应用的另一个例子出现，通常需要特定且昂贵的硬件。 那些拥有这种设备的人可能通过类似的 “exchange” 提供他们的服务，那些需要这些服务的人会租用他们，双方的义务通过智能合约登记。 这类似于 “雾计算” 平台，如 Golem（https://golem.network/ ）或 SONM（https://sonm.io/ ），承诺提供。

4.1.11 示例：TON Proxy 是雾服务。

TON 代理提供了雾服务的另一个示例，其中希望提供其服务（有或没有补偿）作为 ADNL 网络流量的隧道的节点可能会注册，而需要它们的节点可能会根据价格，延迟和时间选择其中一个节点。，带宽提供。 之后，可以使用由 TON Payments 提供的支付通道来处理那些代理服务的小额支付，例如每收集 128 KiB 就付一次款。

4.1.12 示例：TON Payments 是一项雾服务。

TON 支付平台（参见 5）也是这种分散的混合应用的一个例子。

4.2 连接用户和服务提供商

我们在 4.1.9 中已经看到，“fog services”（即混合分散服务）通常需要供给双方都在的市场，交易所或注册管理机构。这些市场被分为链上、链下、混合服务、中心化、去中心化等。

4.2.1 示例：连接到 TON Payments。

例如，如果想要使用 TON Payments（参见 5），第一步是找到 “lightning network” 的现有的一些传输节点（参见 5.2），并与它们建立支付通道。借助于 “encompassing” 覆盖网络可以找到一些节点，该覆盖网络应该包含所有传输闪电网络节点（参见 3.3.17）。 但目前尚不清楚这些节点是否愿意创建新的支付通道。 因，需要注册表，其中准备创建新链接的节点可以发布它们的联系信息（例如，它们的抽象地址）。

4.2.2 示例：将文件上传到 TON Storage。

类似地，如果想要将文件上传到 TON Storage，他必须找到一些愿意签署智能合约的节点，以绑定它们以保留该文件的副本（或者任何低于特定大小限制的文件）。 因此，需要提供用于存储文件的服务节点注册表。

4.2.3 链上，混合和链下注册管理机构。

这样的服务提供商注册表可以完全在链上实现，借助智能合约将注册表保留在其永久存储中。然而这将是非常缓慢和昂贵的。混合方法更有效，其中相对较小且很少更改的链上注册表仅用于指出某些节点（通过其抽象地址或其公钥，可用于定位实际的抽象地址，如 3.2.12），提供链下（中心化）注册服务。

最后，分散的纯粹的链下方法可能包括公共覆盖网络（参见 3.3），那些愿意提供服务的人，或那些想要购买某人服务的人，只需通过他们的私钥进行广播他们的优惠。如果要提供的服务非常简单，甚至可能不需要广播要约：覆盖网络本身的近似成员资格可以用作愿意提供特定服务的人的 “注册表”。然后，如果已知的节点尚未准备好满足其需求，需要此服务的客户端可能会定位（参见 3.3.7）并查询此覆盖网络的某些节点，然后查询其邻居。

4.2.4 注册表或在侧链交易。

实施分散式混合注册管理机构的另一种方法是创建一个独立的专业区块链（“侧链”），由其自己的一套验证者维护，他们在链上智能合约中发布其身份并提供网络访问对这个垂直区块链的所有感兴趣的各方，通过专用的覆盖网络收集交易候选者和广播块更新（参见 3.3）。 然后，此侧链的任何完整节点都可以维护自己的共享注册表副本（基本上等于此侧链的全局状态），并处理与此注册表相关的任意查询。

4.2.5 工作链中的注册表或交易。

另一种选择是在 TON Blockchain 内创建专用工作链，专门用于创建注册管理机构，市场和交易所。 与使用基本工作链中的智能合约相比，这可能更有效，更便宜（参见 2.1.11）。 但是，这仍然比在侧链中维护注册管理机构更昂贵（参见 4.2.4）。

4.3 访问 TON Services

我们在 4.1 中讨论了可能用于创建驻留在 TON 生态系统中的新服务和应用程序的不同方法。 现在我们讨论如何访问这些服务，以及 TON 将提供的一些 “helper services”，包括 TON DNS 和 TON 存储。

4.3.1 TON DNS：主要是链上分层域名服务。

TON DNS 是一种预定义的服务，它使用一组智能合约来保存从人类可读域名到 ADNL 网络节点（T6 区块链账户和智能合约）的（256 位）地址的映射。

虽然任何人原则上可以使用 TON 区块链来实现这样的服务，但是当应用程序或服务想要将人类可读标识符转换为地址时，默认情况下使用这种具有众所周知的接口的预定义服务是有用的。

4.3.2 TON DNS 用例。

例如，希望将一些加密货币转移给另一个用户或商家的用户可能更愿意记住该用户或商家的帐户的 TON DNS 域名，而不是保留他们的 256 位帐户标识符和复制粘贴 他们进入轻钱包客户端的收件人字段。

类似地，TON DNS 可用于定位智能合约的帐户标识符或 ton-services 和 ton-sites 的入口点（参见 4.1.5），从而启用专用客户端（“ton-browser”）或通常的互联

网浏览器 结合专门的 TON Proxy 扩展或独立应用程序，为用户提供类似 WWW 的浏览体验。

4.3.3 TON DNS 智能合约。

TON DNS 通过特殊（DNS）智能合约树实现。 每个 DNS 智能合约都负责注册某些固定域的子域。 将保留 TON DNS 系统的第一级域的 “根”DNS 智能合约位于主链中。 其帐户标识符必须硬编码到希望直接访问 TON DNS 数据库的所有软件中。

任何 DNS 智能合约都包含一个 hashmap，将可变长度的以 null 结尾的 UTF-8 字符串映射到它们的 “值” 中。 此哈希图实现为二进制 Patricia 树，类似于 2.3.7 中描述的，但支持可变长度位串作为键。

4.3.4 DNS 哈希映射或 TON DNS 记录的值。

至于这些值，它们是由 TL 方案描述的 “TON DNS 记录”（参见 2.2.5）。它们由一个 “magic number” 组成，选择一个支持的选项，然后选择一个帐户标识符，一个智能合约标识符，一个抽象网络地址（参见 3.1），或一个用于定位抽象地址的公钥服务（参见 3.2.12），或覆盖网络的描述等等。一个重要的案例是另一个 DNS 智能合约：在这种情况下，该智能合约用于解析其域的子域。通过这种方式，可以为不同的域创建单独的注册表，由这些域的所有者控制。

这些记录还可能包含到期时间，缓存时间（通常非常大，因为更新区块链中的值通常是昂贵的），并且在大多数情况下是对相关子域所有者的引用。所有者有权更改此记录（特别是所有者字段，从而将域名转移给其他人的控制权），并延长它。

4.3.5。注册现有域的新子域。

为了注册现有域的新子域，人们只需向智能合约发送消息，智能合约是该域的注册商，包含要注册的子域（即密钥），其中一个是预定义的值之一——由域名所有者确定的罚款格式，所有者身份，到期日期和一定数量的加密货币。

子域名以 “先到先得” 的方式注册。

4.3.6 从 DNS 智能合约中检索数据。

原则上，如果永久存储内的 hashmap 的结构和位置是智能合约，那么包含 DNS 智能合约的主链或 shardchain 的任何完整节点都可以查找该智能合约的数据库中的任何子域。

但这种方法仅适用于某些 DNS 智能合约。 如果使用非标准 DNS 智能合约它将会失败。

相反，使用基于通用智能合约接口和获取方法的方法（参见 4.3.11）。 任何 DNS 智能合约都必须定义带有 “get method” 的 “known signature”，该方法被调用以查找密钥。 由于这种方法对其他智能合约也有意义，特别是那些提供链上和混合服务的合同，我们将在 4.3.11 中详细解释。

4.3.7 翻译 TON DNS 域。

一旦任何完整节点（单独或代表某个轻客户端）可以查找任何 DNS 智能合约的数据库中的条目，就可以从众所周知的固定根开始，任意地翻译任意 TON DNS 域名。 DNS 智能合约（帐户）标识符。

例如，如果想要翻译 A.B.C，则在根域数据库中查找密钥 .C，.B.C 和 A.B.C。如果找不到第一个，但第二个是，并且其值是对另一个 DNS 智能合约的引用，则在该智能合约的数据库中查找 A 并检索最终值。

4.3.8 为轻型节点翻译 TON DNS 域。

通过这种方式，主链的完整节点以及域查找过程中涉及的所有分片链可以在没有外部帮助的情况下将任何域名转换为其当前值。 轻节点可以请求完整节点代表它执行此操作并返回该值以及 Merkle 证明（参见 2.5.11）。 这种 Merkle 证明将使得工作节点能够验证答案是否正确，因此与通常的 DNS 协议相比，这种 TON DNS 响应不能被恶意拦截器 “spoofed”。

因为没有节点可以预期是关于所有 shard 链的完整节点，所以实际的 TON DNS 域转换将涉及这两种策略的组合。

4.3.9 专用的 “TON DNS servers”。

可以提供简单的 “TON DNS servers”，其将接收 RPC“DNS” 查询（例如，通过 3.1 中描述的 ADNL 或 RLDP 协议），请求服务器转换给定域，通过转发一些来处理这些查询必要时子查询到其他（完整）节点，并返回原始查询的答案，如果需要，可以通过 Merkle 证明进行扩充。

这样的 “DNS 服务器” 可以使用 4.2 中描述的方法之一向任何其他节点，尤其是轻客户端提供其服务（免费或不免费）。 请注意，如果这些服务器被认为是 TON DNS servers 的一部分，它将有效地将其从分布式链上服务转换为分布式混合服务（即 “fog service”）。

以上是对 TON DNS 服务的简要概述，TON DNS 服务是 TON 区块链和 TON 网络实体的人类可读域名的可扩展链式注册表。

4.3.10 访问智能合约中保存的数据。

我们已经看到，有时需要访问存储在智能合约中的数据而不改变其状态。

如果知道智能合约实施的细节，就可以从智能合约的永久性存储中提取所有需要的信息，这些信息可供智能合约所在的 shardchain 的所有完整节点使用。但是这是一种非常不优雅的方式做事，很大程度上取决于智能合约的实施。

4.3.11 智能合约的 “Get methods”。

更好的方法是在智能合约中定义一些 get method，即某些分片的入站消息在交付时不会影响智能合约的状态，但会生成一个或多个包含 “result” 的输出消息。得到方法。以这种方式，可以从智能合约获得数据，仅知道它实现具有已

知签名的 get method（即，要发送的入站消息的已知格式和作为结果要接收的出站消息）。

这种方式更加优雅，符合面向对象的编程（OOP）。但是，到目前为止它有一个明显的缺陷：一个必须实际将事务提交到区块链中（将获取消息发送到智能合约），等待它由验证者提交和处理，从新块中提取答案，并且支付 gas（即，用于在验证者的硬件上执行 get 方法）。这是浪费资源：get method 不会改变智能合约的状态，因此不需要在区块链中执行。

4.3.12 暂时执行智能合约的获取方法。

我们已经评论过（参见 2.4.6）任何完整节点都可以暂时执行任何智能合约的任何方法（即向智能合约发送任何消息），从智能合约的给定状态开始，而不实际提交相应的交易。完整节点可以简单地将所考虑的智能合约的代码加载到 TON VM 中，从 shardchain 的全局状态（shardchain 的所有完整节点都知道）初始化其永久存储，并执行智能合约代码。入站消息作为其输入参数。创建的输出消息将产生此计算的结果。

这样，任何完整节点都可以评估任意智能合约的任意获取方法，前提是它们的签名（即入站和出站消息的格式）是已知的。节点可以追踪在该评估期间访问的 shardchain 状态的单元，并创建所执行的计算的有效性的 Merkle 证明，以获得可能已经要求整个节点这样做的工作节点（参见 2.5.11）。

4.3.13 TL 方案中的智能合约接口。

回想一下，智能合约实现的方法（即它接受的输入消息）本质上是一些 TL 序列化的对象，可以通过 TL 方案来描述（参见 2.2.5）。 得到的输出消息也可以用相同的 TL 方案描述。 通过这种方式，智能合约提供给其他账户和智能合约的界面可以通过 TL 方案形式化。

特别是，智能合约支持的 Get methods（的一部分）可以通过这种形式化的智能合约接口来描述。

4.3.14 智能合约的公共接口。

请注意，形式化的智能合约接口，无论是 TL 方案（表示为 TL 源文件；参见 2.2.5）还是序列化形式都可以发布——例如，在特殊领域中 智能合约帐户描述，存储在区块链中，或单独存储，如果此接口将被多次引用。 在后一种情况下，支持的公共接口的哈希可以合并到智能合约描述中而不是接口描述本身。

这种公共接口的一个例子是 DNS 智能合约，它应该实现至少一种用于查找子域的标准 Get methods（参见 4.3.6）。 注册新子域的标准方法也可以包含在 DNS 智能合约的标准公共接口中。

4.3.15 智能合约的用户界面。

智能合约的公共界面的存在也有其他好处。例如，钱包客户端应用程序可以在根据用户的请求检查智能合约时下载这样的接口，并显示智能合约支持的公共

方法列表（即可用动作的列表），可能具有一些人类可读的评论是否在正式界面中提供。在用户选择这些方法之一之后，可以根据 TL 方案自动生成表单，其中将提示用户所选方法所需的所有字段以及所需的加密货币量（例如 Grams）为附在此请求上。提交此表单将创建一个新的区块链交易，其中包含刚刚撰写的消息，该消息来自用户的区块链账户。

通过这种方式，只要这些智能合约已经发布了他们的界面，用户就可以通过填写和提交某些表格，以用户友好的方式与钱包客户端应用程序中的任意智能合约进行交互。

4.3.16 ton-service 的用户界面。

事实证明 “ton-services”（即驻留在 TON 网络中的服务以及通过 ADNL 和 RLDP 协议接受 3 的查询；参见 4.1.5）也可能从 TL 方案描述的公共接口中获益（ 参见 2.2.5）。

客户端应用程序（例如轻型钱包或 “ton-browser”）可能会提示用户选择其中一种方法，并使用界面定义的参数填写表单，类似于刚刚在 4.3.15 中讨论的内容。唯一的区别是生成的 TL 序列化消息不作为区块链中的事务提交；相反它作为 RPC 查询被发送到所讨论的 “ton-service” 的抽象地址，并且根据形式接口（即 TL 方案）解析和显示对该查询的响应。

4.3.17 通过 TON DNS 定位用户界面。

包含 ton 服务的抽象地址或智能合约帐户标识符的 TON DNS 记录还可以包含描述该实体的公共（用户）接口的可选字段，或者几个支持的接口。 然后客户端应用程序（无论是 wallet，ton-browser 还是 ton-proxy）将能够以统一的方式下载界面并与相关实体（无论是智能合约还是一个 ton-service）进行交互。

4.3.18 模糊链上和链外服务之间的区别。

通过这种方式，最终用户模糊了链上，链外和混合服务（参见 4.1.2）之间的区别：她只需将所需服务的域名输入到她的浏览器的地址行中 或钱包，其余部分由客户端应用程序无缝处理。

4.3.19 轻型钱包和 TON entity explorer 实体浏览器可以内置到 Telegram Messenger 客户端中。

在这一点上出现了一个有趣的机会。 实现上述功能的轻型钱包和 TON 实体浏览器可以嵌入到 Telegram Messenger 智能手机客户端应用程序中，从而将该技术带给超过 2 亿人。用户可以通过在消息中包含 TON URIs（参见 4.3.22）来向 TON 实体和资源发送超链接； 如果选择了这样的超链接，将由接收方的电报客户端应用程序在内部打开，并且将开始与所选实体的交互。

4.3.20 “ton-sites” 作为支持 HTTP 接口的 ton-services。

ton-site 只是一个支持 HTTP 接口的服务，可能还有其他一些接口。

可以在相应的 TON DNS 记录中宣布该支持。

4.3.21 Hyperlinks 超链接。

请注意，ton-sites 返回的 HTML 页面可能包含 ton-hyperlinks-即通过特制 URIs 方案引用其他 ton-sites，smart constracts 和 accounts（参见 4.3.22）——包含摘要 网络地址，帐户标识符或人类可读的 TON DNS 域。 然后，当用户选择它时，“ton-browser” 可能会跟随这样的超链接，检测要使用的界面，并显示 4.3.15 和 4.3.16 中概述的用户界面表格。

4.3.22 超链接 URL 可以指定一些参数。

超链接 URL 不仅可以包含（TON）DNS 域或所讨论的服务的抽象地址，还可以包含要调用的方法的名称以及其部分或全部参数。可能的 URI 方案可能如下所示：

ton://<domain>/<method>?<field1>=<value1>&<field2>=. . .

当用户在 ton-browser 中选择这样的链接时，该动作是立即形成（特别是如果它是智能合约的获取方法，匿名调用），或显示部分填写的表格，由用户明确确认和提交（这可能是付款表格所必需的）。

4.3.23 POST 动作。

一个 ton-sites 可以嵌入 HTML 页面，它返回一些看似常见的 POST 表单，POST 操作通过合适的（TON）URL 引用 ton-sites，ton-services 或智能合约。在这种情况下，一旦用户填写并提交该自定义表单，就会立即或在明确确认后采取相应的操作。

4.3.24 TON WWW。

所有这些将导致创建整个网络的交叉引用实体，驻留在 TON 网络中，最终用户可通过吨浏览器访问该网络，从而为用户提供类似 WWW 的浏览体验。 对于最终用户，这最终将使区块链应用程序与他们已经习惯的网站基本相似。

4.3.25 TON WWW 的优点。

这种 “链上和链下” 服务的 “TON WWW” 与传统服务相比具有一些优势。 例如，付款本质上集成在系统中。 用户身份可以始终呈现给服务（通过在事务和生成的 RPC 请求上自动生成的签名）或随意隐藏。 服务无需检查和重新检查用户凭据；这些凭证可以一次性发布在区块链中。用户网络匿名可以通过 TON 代理轻松保存，并且所有服务都将是有效的不可阻止的。小额支付也很容易，因为 ton-browser 可以与 TON 支付系统集成。

5 TON Payments

我们将在本文中简要讨论的 TON 项目的最后一个组成部分是 TON Payments，这是（小额）支付通道和 “闪电网络” 价值转移的平台。它将实现 “instant 即时” 支付，而无需将所有交易提交到区块链中，支付相关的交易费用（例如消耗的 gas）并等待五秒钟，直到确认包含有关交易的区块。

这种即时支付的总体开销很小，人们可以将它们用于小额支付。例如，TON file-storing 服务可能会为每 128 个下行数据下载数据向用户收费，或者付费 TON 代理可能需要为每中继 128 KiB 的流量提供一些小额微支付。

虽然 TON Payments 可能会晚于 TON 项目的核心组件发布，但一开始需要考虑一些因素。例如，用于执行 TON 区块链智能合约代码的 TON 虚拟机（TON VM；参见 2.1.20）必须支持 Merkle 校样的一些特殊操作。如果原始设计中不存在此类支持，则在稍后阶段添加它可能会成为问题（参见 2.8.16）。但是，我们将看到 TON VM 自然支持开箱即用的 “smart 智能” 支付通道（参见 5.1.9）。

5.1 支付通道

我们首先讨论点对点支付通道，以及如何在 TON 区块链中实施这些渠道。

5.1.1 支付通道的思路

假设 A 和 B 两方都知道他们将来需要相互支付很多款项。他们不是将每笔付款作为区块链中的交易，而是创建一个共享的 “money pool”（或者可能是一个只有两个帐户的小型私人银行），并为其提供一些资金：A 贡献硬币，B 贡献 b 币。这是通过在区块链中创建一个特殊的智能合约并将钱汇给它来实现的。

在创建 “money pool” 之前，双方同意某个协议。他们将追踪池的状态——即共享池中的余额。最初，状态是（a，b），意思是硬币实际上属于 A，b 硬币属于 B. 然后，如果 A 想要向 B 支付硬币，他们可以简单地同意新状态是（a'，b'）=（a - d，b + d）。之后，如果 B 想要向 A 支付硬币，那么州将成为（''，b''）=（a'+ d'，b' - d'），依此类推。

所有这些池内余额的更新都是完全链下的。当双方决定从池中取出应付资金时，他们会根据池的最终状态这样做。这是通过向智能合约发送特殊消息来实现的，其中包含商定的最终状态（a *，b *）以及 A 和 B 的签名。然后智能合约向 A，b * 发送 * 硬币硬币到 B 和自我毁灭。

此智能合约以及 A 和 B 用于更新池状态的网络协议是 A 和 B 之间的简单支付通道。根据 4.1.2 中描述的分类，它是一种混合服务：部分它的状态存在于区块链（智能合约）中，但其大多数状态更新都是在链外（通过网络协议）执行的。如果一切顺利，双方将能够按照自己的意愿执行尽可能多的付款（唯一的限制是渠道的 “capacity” 不会超支——即，他们在付款渠道中的余额都保持不变非负的），只将两个交易提交到区块链中：一个用于打开（创建）支付通道（智能合约），另一个用于关闭（销毁）它。

5.1.2 无需信任的支付通道。

前面的例子有点不切实际，因为它假设双方都愿意合作，绝不会欺骗以获得某些优势。想象一下，例如，A 会选择不用 '<a 来签署最终余额（a'，b'）。这将使 B 陷入困境。

为了防止这种情况，人们通常会尝试开发无信任的支付通道协议，这些协议不要求各方相互信任，并规定惩罚任何试图作弊的人。

这通常借助签名来实现。支付通道智能合约知道 A 和 B 的公钥，并且如果需要，它可以检查他们的签名。支付通道协议要求各方签署中间状态并将签名发送给彼此。然后，如果其中一方欺骗——例如假装支付通道的某些状态从未存在——可以通过在该状态上显示其签名来证明其不当行为。支付通道智能合约充当 “on-chain arbiter 链上仲裁者”，能够处理双方关于彼此的投诉，并通过没收所有资金并将其授予另一方来惩罚有罪方。

5.1.3 简单的双向同步无信道支付通道。

考虑以下更实际的例子：让支付通道的状态由三元组（δi, i, oi ）描述，其中 i 是状态的序列号（它原来是零，然后当它增加一时） 随后的状态出现），δi 是信道不平衡（意味着 A 和 B 分别拥有 + a+δi 和 b − δi 硬币），并且 oi 是允许产生下一状态（A 或 B）的一方。 在进一步取得进展之前，每个州必须由 A 和 B 签署。

现在，如果 A 想要将 d 硬币转移到支付通道内的 B，并且当前状态是 Si = (δi, i, oi)，其中 oi = A，则它只是创建一个新状态 Si+1 = (δi − d, i + 1, oi+1)，签名，并将其连同其签名一起发送给 B. 然后 B 通过签名并将其签名副本发送给 A 来确认它。之后，双方都拥有新签名的副本，并且可能会发生新的转移。如果 A 想要在具有 oi = B, 的状态 Si 中将硬币转移到 B，则它首先要求 B 提交具有相同不平衡 δi+1 = δi 的后续状态 Si+1，但是具有 oi+1 = A，A 将能够进行转移。

当双方同意关闭支付通道时，他们都将他们的特殊最终签名放在他们认为是最终的州 Sk 上，并通过发送最终状态来调用支付通道智能合约的清洁或双边最终确定方法以及两个最终签名。

如果另一方不同意提供其最终签名，或者仅仅是它停止响应，则可以单方面关闭该渠道。为此，希望这样做的一方将调用单方面的最终确定方法，向智能合约发送其最终状态的版本，最终签名以及具有另一方签名的最新状态。在此之后，智能合约不会立即对收到的最终状态采取行动。相反，它等待一段时间（例如，一天）以使另一方呈现其最终状态的版本。当另一方提交其版本并且结果与已提交的版本兼容时，“真实” 最终状态由智能合约计算并用于相应地分配资金。如果另一方未能将其最终状态的版本呈现给智能合约，则根据所呈现的最终状态的唯一副本重新分配资金。

如果双方中的一方作弊——例如，签署两个不同的作为最终状态，或者通过签署两个不同的下一个状态 S 和 S'，或者通过签署无效的新状态 Si+1 （例如，具有不平衡 δi+1 <-a 或> b）——然后另一方可以提交证明这种不当行为是智能合

约的第三种方法。有罪的一方将立即因完全失去其在支付通道中的份额而受到惩罚。

这种简单的支付通道协议是公平的，即无论是否有另一方的合作，任何一方都可以随时获得应付款，并且如果它试图作弊，可能会失去承诺支付通道的所有资金。

5.1.4 同步支付通道作为一个简单的虚拟区块链，带有两个验证者。

以上简单同步支付信道的示例可以如下重新制作。想象一下，状态序列 S0, S1, ..., Sn 实际上是非常简单的区块链的块序列。该区块链的每个块基本上仅包含区块链的当前状态，并且可能是对前一个区块的引用（即，其哈希）。 A 和 B 双方都充当此区块链的验证者，因此每个区块必须收集其两个签名。区块链的状态 Si 定义了下一个区块的指定生产者 oi，因此 A 和 B 之间没有用于生成下一个区块的竞争。允许生产者 A 创建将资金从 A 转移到 B 的块（即，减少不平衡：δδi+1 ≤ δi)，并且 B 只能将资金从 B 转移到 A（即，增加 δ）。

如果两个验证人就区块链的最终区块（和最终状态）达成一致，则通过收集双方的特殊 “最终” 签名来最终确定，并将其与最终区块一起提交给渠道智能合约进行处理和相应地重新分配资金。

如果验证者签署了无效区块，或者创建了一个分支，或者签署了两个不同的最终区块，则可以通过向智能合约提供其不当行为的证据来惩罚它，该合同充当两个验证者的 “on-chain arbiter 链上仲裁器”。 ；然后，违规方将失去保留在支付通道中的所有资金，这类似于失去其 stake 的验证方。

5.1.5 异步支付通道作为具有两个工作链的虚拟区块链。

5.1.3 中讨论的同步支付通道具有一定的缺点：在另一方确认之前的交易之前，不能开始下一笔交易（支付通道内的汇款）。这可以通过用两个交互虚拟工作链（或者说是 shardchains）的系统替换 5.1.4 中讨论的单链虚拟区块链来解决。

这些工作链中的第一个仅包含 A 的事务，其块只能由 A 生成；它的状态是 Si = (i, φi, j, ψj), 其中 i 是块序列号（即到目前为止由 A 执行的交易或汇款的数量），φi 是从 A 转移到 A 的总量。 B 到目前为止，j 是 A 知道的 B 区块链中最近有效区块的序列号，ψj 是其 j 交易中从 B 转移到 A 的金额。放在第 j 个块上的 B 的签名也应该是这个状态的一部分。也可以包括该工作链的前一个块和另一个工作链的第 j 个块的哈希。如果 i > 0, ψj ≥ 0 且 φi ≥ 0, φi ≥ φi−1 则 Si include φi ≥ 0, φi ≥ φi−1。

类似地，第二个工作链仅包含 B 的事务，其块仅由 B 生成； 其状态为 Tj = (j, ψj , i, φi) ，具有相似的有效性条件。
现在，如果 A 想要将一些钱转移到 B，它只需在其工作链中创建一个新块，签名并发送给 B，而无需等待确认。

支付通道由 A 签署（其版本）区块链的最终状态（具有特殊的 “final signature ”），B 签署其区块链的最终状态，以及将这两个最终状态呈现给清洁终结而最终确定。 支付通道智能合约的方法。 单边最终化也是可能的，但在这

种情况下，智能合约必须等待另一方提交其最终状态的版本，至少在某个宽限期内。

5.1.6 单向支付通道。

如果只有 A 需要向 B 付款（例如，B 是服务提供商，A 是其客户），则可以创建单边支付通道。 从本质上讲，它只是 5.1.5 中描述的第一个没有第二个工作链的工作链。 相反，可以说 5.1.5 中描述的异步支付通道由两个统一支付通道组成，或由同一智能合约管理的 “half-channels 半个通道”。

5.1.7 更复杂的支付通道。

我们将在后面的 5.2.4 中看到，“lightning network”（参见 5.2），通过几个支付通道的链条实现即时汇款，需要从所涉及的支付通道获得更高的复杂程度。

特别是，我们希望能够实施 “promises” 或 “conditional money transfers”：A 同意向 B 发送 c 币，但 B 只有在满足某个条件时才会获得金钱，例如，如果 B 可以用一个已知的 v 值表示一些带有 Hash(u) = v for a known value of v。否则，A 可以在一段时间后收回钱。

这样的承诺可以通过简单的智能合约轻松实现。我们希望承诺和其他类型的有条件汇款可以在链下支付通道中进行，因为它们可以大大简化 “lightning network” 中存在的一系列支付通道的资金转移（参见 5.2.4）。

5.1.4 和 5.1.5 中概述的 “payment channel as a simple blockchain” 图片在这里变得很方便。现在我们考虑一个更复杂的虚拟区块链，其状态包含一系列未实现的 “promises”，以及锁定在此类承诺中的资金数量。这个区块链——或异步情况下的两个工作链——必须通过它们的哈希显式引用前面的块。然而，一般机制仍然相同。

5.1.8 复杂的支付通道智能合约面临的挑战。

请注意，虽然复杂的支付通道的最终状态仍然很小，而且 “clean” 的最终确定很简单（如果双方已就其应付金额达成一致，并且双方已签署协议），但单边定案方法和惩罚欺诈行为的方法需要更加复杂。 实际上，他们必须能够接受 Merkle 证明可能作假，并支持检查区块链上支付通道更复杂的交易是否在正确处理。

换句话说，支付通道智能合约必须能够使用 Merkle 证明，检查其 “哈希有效性”，并且必须包含支付通道（虚拟）的 ev*trans 和 ev*block 功能实现（参见 2.2.6）的 blockchain。

5.1.9 TON VM 支持 “智能” 支付通道。

用于运行 TON 区块链智能合约代码的 TON VM 可以应对执行 “smart” 或复杂支付通道所需的智能合约（参见 5.1.8）。

在这一点上，“everything is a bag of cells 一切都是一袋细胞” 范例（参见 2.5.14）变得非常方便。由于所有块（包括临时支付通道区块链的块）都表示为

单元格袋（并由一些代数数据分片描述），并且同样适用于消息和 Merkle 证明，因此可以轻松地将 Merkle 证明嵌入到发送到付款渠道智能合约的入站消息。将自动检查 Merkle 证明的 “哈希条件”，并且当智能合约访问所呈现的 “Merkle proof” 时，它将使用它，就好像它是相应的代数数据分片的值——尽管不完整，树的一些子树被包含省略子树的 Merkle 哈希的特殊节点替换。然后，智能合约将使用该值，该值可能代表支付通道（虚拟）区块链及其状态的块，并将评估该区块链的 ev_block 功能（参见 2.2.6）。在这个块和以前的状态。然后，计算结束，并且可以将最终状态与块中断言的状态进行比较，或者在尝试访问缺席子树时抛出 “absent node” 异常，指示 Merkle 证明无效。

通过这种方式，使用 TON 区块链智能合约实现智能支付通道区块链的验证码变得非常简单。有人可能会说，TON 虚拟机具有内置支持，可以检查其他简单区块链的有效性。唯一的限制因素是 Merkle 证明的大小要合并到智能合约的入站消息中（即进入交易）。

5.1.10 智能支付通道内的简单支付通道。

我们想讨论在现有支付通道内创建简单（同步或异步）支付通道的可能性。

虽然这看起来有点令人费解，但理解和实施并不比 5.1.7 中讨论的 “promises” 困难得多。基本上，如果出现一些哈希问题的解决方案，而不是承诺向另一方支付 c 币，A 承诺根据某些其他（虚拟）支付通道区块链的最终结算向 B 支付最多 c 币。一般来说，这个其他支付通道区块链甚至不需要在 A 和 B 之间；它可能涉及其他一些方面，比如 C 和 D 将分别将 c 和 d 硬币投入其简单的支付通道。（这种可能性在后面的 5.2.5 中被利用。）

如果包含的支付通道是不对称的，则需要将两个承诺提交到两个工作链中：如果 “internal” 简单支付通道的最终结算产生负的最终不平衡 δ 且 0≤，A 将承诺向 B 支付-δ 个硬币-δ≤c；如果 δ 为正，则 B 必须承诺向 A 支付 δ。另一方面，如果包含支付通道是对称的，这可以通过将单个 “simple payment channel creation” 交易与参数（c，d）一起提交到单个支付通道区块链中来完成（这将冻结 c 硬币属于到 A），然后由 B 进行特殊的 “confirmation transaction”（这将冻结 B 的硬币）。

我们希望内部支付通道非常简单（例如，5.1.3 中讨论的简单同步支付通道），以最小化要提交的 Merkle 证明的大小。在 5.1.7 中描述的意义上，外部支付通道必须是 “smart”。

5.2 支付通道网络或 “闪电网络”

现在我们讨论 TON Payments 的 “Lightning Network”，它可以实现任意两个参与节点间的即时汇款。

5.2.1 付款渠道的限制。

付款渠道对于期望在他们之间进行大量汇款的各方非常有用。 但是如果只需要向特定收件人转账一次或两次，那么与她建立付款渠道将是不切实际的。 除此

之外，这意味着冻结支付通道中的大量资金，并且无论如何都需要至少两个区块链交易。

5.2.2 支付通道网络，或“闪电网络”。

支付通道网络通过支付通道链接实现资金转移，克服了支付通道的限制。如果 A 希望将资金转移到 E，她不需要与 E 建立支付通道。通过几个中间节点（例如，四个支付通道，从 A 到 A）连接 A 和 E 的支付通道就足够了。 B，从 B 到 C，从 C 到 D，从 D 到 E.

5.2.3 支付通道网络概述。

回想一下，支付通道网络，也称为“闪电网络”，由一组参与节点组成，其中一些节点已在它们之间建立了长期支付通道。 我们将在一瞬间看到这些支付通道必须是 5.1.7 意义上的“smart”。 当参与节点 A 想要将钱转移到任何其他参与节点 E 时，她试图找到在支付信道网络内链接 A 到 E 的路径。 当找到这样的路径时，她沿着这条路径进行“chain money transfer”。

5.2.4 “chain money transfer” 连锁货币转账。

假设存在从 A 到 B，从 B 到 C，从 C 到 D，从 D 到 E 的支付通道链。此外，假设 A 想要将 x 个硬币转移到 E.

一种简单的方法是沿着现有的支付通道将 x 币转移到 B，并要求他将钱进一步转发给 C. 但是，为什么 B 不会简单地为自己拿钱也不明显。因此，必须采用更复杂的方法，不要求所有相关方相互信任。

这可以如下实现。 A 生成一个大的随机数 u 并计算其哈希 v = Hash（u）。然后她创建了一个承诺，如果有一个带有哈希 v 的数字 u（参见 5.1.7），在她的支付通道中有 B，则向 B 支付 x 币。这个承诺包含 v，但不是 u，它仍然保密。

之后，B 在其支付通道中创建了与 C 类似的承诺。他并不害怕给出这样的承诺，因为他知道 A 给他的类似承诺的存在。如果 C 曾经提出哈希问题的解决方案来收集 B 承诺的 x 币，那么 B 将立即提交这个解 A 来从 A 收集 x 币。

然后创建 C 到 D 和 D 到 E 的类似承诺。当承诺全部到位时，A 通过将解决方案 u 传达给所有相关方——或者只是向 E 传达来触发转移。

在本文档省略了一些细节。例如，这些承诺必须具有不同的到期时间，并且承诺的金额可能在链条上略有不同（B 可能只承诺 x——ε 个硬币到 C，其中 ε 是预先商定的小的运输费用）。我们暂时忽略这些细节，因为它们对于理解支付通道如何运作以及如何在 TON 中实施这些细节并不太重要。

5.2.5 支付通道链中的虚拟支付通道。

现在假设 A 和 E 期望彼此支付很多钱。他们可能会在区块链中创建一个新的支付通道，但这仍然非常昂贵，因为有些资金会被锁定在这个支付通道中。另一

种选择是对每笔付款使用 5.2.4 中描述的连锁汇款。但是，这将涉及大量网络活动以及所涉及的所有支付通道的虚拟区块链中的大量交易。

另一种方法是在支付通道网络中链接 A 到 E 的链内创建虚拟支付通道。为此，A 和 E 为他们的付款创建（虚拟）区块链，就像他们要在区块链中创建支付通道一样。然而，他们不是在区块链中创建支付通道智能合约，而是要求所有中间支付通道——将 A 到 B，B 到 C 等连接起来——在其中创建简单的支付通道，绑定到 A 创建的虚拟区块链和 E（参见 5.1.10）。换句话说，现在根据 A 和 E 之间的最终结算转移资金的承诺存在于每个中间支付通道内。

如果虚拟支付通道是单向的，那么这种承诺可以很容易地实现，因为最终的不平衡 δ 将是非正的，因此可以在中间支付通道内以与 5.2 中描述的相同的顺序创建简单的支付通道。 0.4。它们的到期时间也可以以相同的方式设置。

如果虚拟支付通道是双向的，情况会稍微复杂一些。在这种情况下，应该将根据最终结算将 δ 硬币转移到两个半承诺中的承诺分开，如 5.1.10 所述：在向前方向上转移 $\delta- = \max(0, -\delta)$ 硬币，以及在向后方向上传递 $\delta+ = \max(0, \delta)$ 。这些半承诺可以在中间支付通道中独立创建，一个半承诺链从 A 到 E，另一个链在相反方向。

5.2.6 寻找闪电网络中的路径。

到目前为止，有一点仍然未被讨论：A 和 E 将如何在支付网络中找到连接它们的路径？如果支付网络不是太大，则可以使用类似 OSPF 的协议：支付网络的所有节点创建覆盖网络（参见 3.3.17），然后每个节点传播所有可用链路（即，参与支付信道））通过八卦协议向其邻居提供信息。最终，所有节点都将拥有参与支付网络的所有支付通道的完整列表，并且能够自己找到最短路径——例如，通过应用 Dijkstra 修改的算法版本来考虑 “所涉及的支付通道的 “能力”（即，可以沿着它们转移的最大金额）。一旦找到候选路径，就可以通过包含完整路径的特殊 ADNL 数据库进行探测，并要求每个中间节点确认所讨论的支付通道的存在，并根据该路径进一步转发该数据。之后，可以构建链，以及链转移协议（参见 5.2.4），或创建虚拟支付可以运行支付通道链中的渠道（参见 5.2.5）。

5.2.7 优化。

可以在此处进行一些优化。例如，只有闪电网络的传输节点需要参与 5.2.6 中讨论的类似 OSPF 的协议。 希望通过闪电网络连接的两个 “分支” 节点将彼此通信他们所连接的传输节点的列表（即，他们已经建立了参与支付网络的支付信道）。 然后，可以检查从一个列表到另一个列表中的传输节点的传输节点的路径，如上面 5.2.6 中所述。

5.2.8 结论。

我们已经概述了 TON 项目的区块链和网络技术如何足以完成创建 TON Payments 的任务，TON Payments 是一个链下即时汇款和小额支付的平台。 该

平台对于驻留在 TON 生态系统中的服务非常有用，使他们可以在需要时轻松收集小额支付。

参考

[1] K. Birman, Reliable Distributed Systems: Technologies, Web Services and Applications, Springer, 2005.

[2] V. Buterin, Ethereum: A next-generation smart contract and de- centralized application platform, https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[3] M. Ben-Or, B. Kelmer, T. Rabin, Asynchronous secure computa- tions with optimal resilience, in Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, p. 183–192. ACM, 1994.

[4] M. Castro, B. Liskov, et al., Practical byzantine fault tolerance, Proceedings of the Third Symposium on Operating Systems Design and Implementation (1999), p. 173–186, available at http://pmg.csail.mit.edu/papers/osdi99.pdf.

[5] EOS.IO, EOS.IO technical white paper, https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md, 2017.

[6] D. Goldschlag, M. Reed, P. Syverson, Onion Routing for Anony- mous and Private Internet Connections, Communications of the ACM, 42, num. 2 (1999), http://www.onion-router.net/Publications/CACM-1999.pdf.

[7] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Transactions on Programming Languages and Systems, 4/3 (1982), p. 382–401.

[8] S. Larimer, The history of BitShares, https://docs.bitshares.org/bitshares/history.html, 2013.

[9] M. Luby, A. Shokrollahi, et al., RaptorQ forward error correction scheme for object delivery, IETF RFC 6330, https://tools.ietf.org/html/rfc6330, 2011.

[10] P. Maymounkov, D. Mazières, Kademlia: A peer-to-peer infor- mation system based on the XOR metric, in IPTPS ’01 revised pa- pers from the First International Workshop on Peer-to-Peer Systems, p. 53–65, available at http://pdos.csail.mit.edu/~petar/papers/ maymounkov-kademlia-lncs.pdf, 2002.

[11] A. Miller, Yu Xia, et al., The honey badger of BFT protocols, Cryptology e-print archive 2016/99, https://eprint.iacr.org/2016/ 199.pdf, 2016.

[12] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/bitcoin.pdf, 2008.

[13] S. Peyton Jones, Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, Journal of Functional Pro- gramming 2 (2), p. 127–202, 1992.

[14] A. Shokrollahi, M. Luby, Raptor Codes, IEEE Transactions on Information Theory 6, no. 3–4 (2006), p. 212–322.

[15] M. van Steen, A. Tanenbaum, Distributed Systems, 3rd ed., 2017.

[16] The Univalent Foundations Program, Homotopy Type Theory: Univalent Foundations of Mathematics, Institute for Advanced Study, 2013, available at https://homotopytypetheory.org/book.

[17] G. Wood, PolkaDot: vision for a heterogeneous multi-chain frame- work, draft 1, https://github.com/w3f/polkadot-white-paper/raw/ master/PolkaDotPaper.pdf, 2016.

申明

转载或商用请保留作者与译者的姓名。

新版白皮书未来将更新在 https://github.com/opeakt/TON-Community，请留意更新。