The Open Network

作者：Nikolai　Durov

简体中文翻译:　Kojhliang, Kenetic Capital

2024年2月8日

摘要

本论文旨在首次详述The Open Network (TON) 以及相关的区块链、P2P (点对点)、分布式储存及服务托管技术。为了确保本文件的篇幅在合理的范畴内，我们主要针对TON平台中具有独特性和关键性的功能进行探讨，这些功能对于实现设定的目标十分重要。

绪论

The Open Network (TON)是一个快速、安全且可扩展的区块链和网络项目，它能够处理每秒数百万笔的交易，对使用者和服务提供者都非常友好。我们希望它能够承载所有目前提议的和设想的合理应用程序。可以将TON视为一个巨大的分布式超级电脑，或者更恰当地说，一个巨大的超级服务器，旨在托管和提供各种服务。

本文并未详尽描述所有细节。并且在开发和测试阶段，部分具体内容有可能已发生变更。

1.TON组件的简要描述 9

2.TON区块链 11

2.1 二维区块链集合的TON区块链 11

2.1.1 区块链类型列表 11

2.1.2 无限分片范式 12

2.1.3 消息,超立方体路由 13

2.1.4 主链，工作链和分片链的数量 13

2.1.5 工作链可以是虚拟区块链，而不是真正的区块链 13

2.1.6 工作链的识别 14

2.1.7 创建和激活新的工作链 14

2.1.8 识别分片链 14

2.1.9 识别账户链 14

2.1.10 分片链的动态拆分和合并（参照 2.7.） 14

2.1.11 基础工作链或工作链0 15

2.1.12 区块出块时间 15

2.1.13 通过主链使工作链和分片链紧密耦合 15

2.1.14 主链的区块哈希值作为一个全局变量 15

2.1.15 验证节点出新块的过程（参照 2.6.） 16

2.1.16 主链的分叉 16

2.1.17 修正无效分片链区块 17

2.1.18 Ton coins 和多货币工作链 18

2.1.19 消息传递和价值转移 18

2.1.20 Ton虚拟机 18

2.1.21 可配置的参数 20

2.2 区块链概述 21

2.2.1 区块链的定义 21

2.2.2 Ton区块链与一般区块链的相关性 21

2.2.3 区块链实例和区块链类型 21

2.2.4 依赖类型理论（Dependent type theory）， Coq and TL 22

2.2.5 TL，类型语言 22

2.2.6 区块和交易作为状态转换运算符 22

2.2.7 区块链序列号 23

2.2.8 区块哈希 24

2.2.9 哈希假设 24

2.2.10 哈希在Ton区块链的应用 25

2.3 区块链状态、帐户和哈希值映射 25

2.3.1 账户ID 25

2.3.2 主要组成部分： HashMaps 25

2.3.3 例子：Ton账户余额 26

2.3.4 例子：一个智能合约的持久存储 26

2.3.5 例子：所有智能合约的持久存储 26

2.3.6 Hashmap 类型 26

2.3.7 将Hashmap定义为Patricia树 27

2.3.8 MPT树（Merkle-Patricia trees） 28

2.3.9 重新计算Merkle树的哈希值 29

2.3.10 默克尔证明（Merkle proofs） 29

2.3.11 对于像TON 这样的多链系统，默克尔证明的重要性 29
2.3.12 TON VM的特点 30
2.3.13 Ton智能合约的持久存储 30
2.3.14 TVM Cells 31
2.3.15 任意代数类型值的一般化Merkle证明 31
2.3.16 支持对Ton VM数据结构进行分片 31
2.3.17 为持久存储支付费用 32
2.3.18 本地和全局智能合约以及智能合约实例 33
2.3.19 限制智能合约的拆分 33
2.3.20 账户/智能合约状态 34
2.4 分片链间的消息 35
2.4.1 消息、帐户和交易 概览 35
2.4.2 使用参与者模型（Actor model）的账户 35
2.4.3 消息接收者 35
2.4.4 消息发送者 35
2.4.5 消息的附加值 36
2.4.6 外部消息：来自区块链外部的消息 36
2.4.7 日志消息 37
2.4.8 与链下服务和外部区块链交互 37
2.4.9 消息体 37
2.4.10 Gas 限制和其他工作链 / VM 特定参数 37
2.4.11 创建新消息：智能合约和交易 37
2.4.12 传递信息 38
2.4.13 传递消息是一种交易 38
2.4.14 同一智能合约实例之间的消息 38
2.4.15 发送给任一个智能合约实例的消息：通配符地址 38
2.4.16 输入队列不存在 39
2.4.17 输出队列 39
2.4.18 可靠，快速的链间消息传递 39
2.4.19 超立方体路由：为确保消息传递的慢速路径 39
2.4.20 即时超立方体路由:消息的“快速路径” 40
2.4.21 从邻近的分片链的输出队列收集输入消息 41
2.4.22 从输出队列中删除消息 41
2.4.23 防止重复传递消息 41
2.4.24 转发用于其他分片链的消息 41
2.4.25 转发和保留消息的费用 41
2.4.26 发送消息到主链 42
2.4.27 同一个分片链中的帐户之间的消息 42
2.5 全局分片链状态：BAG OF CELLS哲学 42
2.5.1 分片链状态是账户链状态的集合 43
2.5.2 拆分和合并分片链状态 43
2.5.3 账户链状态 43
2.5.4 全局工作链状态 43
2.5.5 低层级角度：“bag of cells” 44
2.5.6 分片链区块表示为“bag of cells” 45
2.5.7 更新对象表示为“bag of cells” 45

2.5.8 更新账户的状态 *45*
2.5.9 更新区块 *45*
2.5.10 默克尔证明（Merkle proof）表示为"bag of cells" *46*
2.5.11 Merkle 证明作为全节点的查询响应 *46*
2.5.12 基于默克尔证明进行状态更新 *46*
2.5.13 在分片链区块中更新账户状态 *47*
2.5.14 所有的东西都是一个「bag of cells」的设计哲学 *47*
2.5.15 Ton区块链的区块头 *47*
2.6 创建和验证新区块 48
2.6.1 验证节点 *48*
2.6.2 验证节点代替矿工 *48*
2.6.3 提名者（Nominators） 和矿池（mining pools） *49*
2.6.4 渔夫(Fisherman):通过指出别人的错误来获得金钱 *49*
2.6.5 核对者(Collator):通过给验证者提议新区块来获得奖励 *50*
2.6.6 核对者和验证者：获取为用户在区块中包含用户交易的奖励 *50*
2.6.7 选举全局验证者集合 *50*
2.6.8 选举验证者任务组 *52*
2.6.9 在每个任务组上轮换优先级顺序 *52*
2.6.10 分片链候选区块的传播 *52*
2.6.11 验证候选区块 *54*
2.6.12 下一个候选区块的选举 *54*
2.6.13 验证节点必须保留它们已经签名的区块 *54*
2.6.14 传播新分片链的区块头和签名给所有验证者 *54*
2.6.15 生成新主链区块 *55*
2.6.16 验证节点必须同步主链的状态 *55*
2.6.17 分片链区块的生成和传播是并行的 *55*
2.6.18 减轻区块被私自保留而不发布的风险 *56*
2.6.19 主链区块比分片链区块更晚生成 *56*
2.6.20 慢验证节点会获得更低的奖励 *56*
2.6.21 验证者节点的签名"深度"（ Depth） *56*
2.6.22 验证节点只负责签名区块的相对有效性 *57*
2.6.23 签名主链区块 *57*
2.6.24 验证节点的总数 *57*
2.6.25 系统的去中心化 *58*
2.6.26 区块的相对可靠性 *59*
2.6.27 提高区块链可靠性 *59*
2.6.28 区块的递归可靠性 *59*
2.6.29 Proof-of-Stake算法对于轻节点的重要性 *60*
2.7 分裂和合并分片链 60
2.7.1 分片配置 *60*
2.7.2 最近的分片配置和状态 *61*
2.7.3 宣布并执行分片配置中的更改 *61*
2.7.4 新分片链的验证者任务组 *61*
2.7.5 在原任务组的工作期间限制分裂/合并的操作 *62*
2.7.6 决定执行分裂操作的必要条件 *62*
2.7.7 执行分裂操作 *62*
2.7.8 决定执行合并操作的必要条件 *63*

2.7.9 执行合并操作 *63*
2.8 区块链项目的分类 64
2.8.1 区块链项目的分类 *64*
2.8.2 单链区块链和多链区块链项目 *65*
2.8.3 创建和验证区块:工作量证明（POW）vs.权益证明（POS） *65*
2.8.4 Proof-of-Stake的不同算法： DPOS vs. BFT. *66*
2.8.5 DPOS 和 BFT PoS 的对比 *67*
2.8.6 支持在交易里执行图灵完备的智能合约代码 *68*
2.8.7 多链系统的分类 *69*
2.8.8 区块链类型：同构区块链和异构系统 *69*
2.8.9 同构-异构混合系统 *69*
2.8.10 具有相同的规则的异构系统工作链：联盟 *70*
2.8.11 内部主链和外部主链 *70*
2.8.12 分片支持 *70*
2.8.13 动态和静态分片 *71*
2.8.14 区块链之间的交互:松散耦合和紧密耦合系统 *71*
2.8.15 简化区块链项目分类 *72*
2.8.16 改变区块链项目“基因组”(genome）的复杂性 *73*
2.8.17 TON 区块链的"基因组" *73*
2.9 与其他区块链项目的比较 74
2.9.1 Bitcoin [12]; https://bitcoin.org/ *74*
2.9.2 Ethereum [2]; https://ethereum.org/ *75*
2.9.3 NXT; https://nxtplatform.org/ *75*
2.9.4 Tezos; https://www.tezos.com/ *75*
2.9.5 Casper *75*
2.9.6 BitShares [8]; https://bitshares.org *75*
2.9.7 EOS [5]; https://eos.io. *76*
2.9.8 PolkaDot [17]; https://polkadot.io. *76*
2.9.9 Universa; https://universa.io *76*
2.9.10 Plasma; https://plasma.io *77*
2.9.11 垂直领域区块链项目 *77*
2.9.12 TON 区块链 *78*
2.9.13 在区块链上构建Facebook是否可行? *78*
3. TON网络协议 80
3.1 抽象数据报网络层（ANDL） 80
3.1.1 抽象网络地址 *81*
3.1.2 低层级的网络，UDP实现 *81*
3.1.3 基于UDP的简单ANDL例子 *81*
3.1.4 安全性较低的方式，发件人的地址为明文 *82*
3.1.5 通道和通道标识符 *82*
3.1.6 通道作为隧道标识符 *82*
3.1.7 零通道和引导问题 *83*
3.1.8 基于ADNL 的 类TCP 流协议 *83*

3.1.9 RLDP，或 基于ADNL上的可靠大数据报协议.....83
3.2 TON DHT: 类KADEMLIA 的分布式哈希表.....84
3.2.1 Ton DHT的key.....84
3.2.2 DHT的值.....84
3.2.3 DHT 的节点，半永久网络身份.....85
3.2.4 Kademlia 距离.....85
3.2.5 类 Kademlia 的 DHTs 和 TON DHT.....85
3.2.6 Kademlia路由表.....85
3.2.7 Kademlia 网络查询.....86
3.2.8 引导一个Kademlia节点.....87
3.2.9 存储值到TON DHT.....87
3.2.10 TON DHT中的分布式“流追踪器”和“网络兴趣组”.....87
3.2.11 Fall-back keys.....88
3.2.12 查找服务.....88
3.2.13 定位Ton区块链的账户所有者.....88
3.2.14 定位抽象地址.....89
3.3 覆盖网络和多播消息.....89
3.3.1 覆盖网络.....90
3.3.2 TON的覆盖网络.....90
3.3.3 私有和公共覆盖网络.....90
3.3.4 中心化和去中心化覆盖网络.....91
3.3.5 加入一个覆盖网络.....91
3.3.6 定位覆盖网络中的一个成员.....91
3.3.7 定位覆盖网络中的更多成员，创建链接.....91
3.3.8 维护邻居列表.....92
3.3.9 覆盖网络是一个随机子图.....92
3.3.10 Ton覆盖网络为降低延迟而做的优化.....93
3.3.11 覆盖网络中的八卦协议（Gossip protocol）.....93
3.3.12 将覆盖网络作为一个广播域.....93
3.3.13 更复杂的广播协议.....93
3.3.14 检查覆盖网络的连接性.....94
3.3.15 流式广播协议.....94
3.3.16 基于现有的覆盖网络构建新的覆盖网络.....95
3.3.17 覆盖网络中的覆盖网络.....95

4 TON服务与应用程序.....96

4.1 TON服务实施策略.....96
4.1.1 应用和服务.....96
4.1.2 应用的位置:链上，链外或混合.....96
4.1.3 中心化和去中心化应用程序。.....97
4.1.4 纯链上应用程序：Dapps.....97
4.1.5 纯网络服务：Ton网站和Ton服务.....97
4.1.6 Telegram作为Ton服务，基于RLDP的MTProto协议.....98
4.1.7 混合服务：部分链下，部分链上.....98
4.1.8 保持文件在链下，TON 存储.....98
4.1.9 去中心化的混合服务：雾服务.....99
4.1.10 “雾计算”平台.....99

4.1.11 TON 代理是一个雾服务 *99*
4.1.12 TON 支付是一个雾服务 *100*
4.2 连接用户和服务提供者 100
4.2.1 连接到TON支付 *100*
4.2.2 上传文件到TON存储 *100*
4.2.3 链上，链下和混合型的注册表 *100*
4.2.4 在侧链中创建注册表 *101*
4.2.5 在工作链中创建注册表 *101*
4.3 访问TON服务 101
4.3.1 TON DNS:一种链上的分层域名服务 *102*
4.3.2 TON DNS 用例 *102*
4.3.3 TON DNS 智能合约 *102*
4.3.4 TON DNS 记录 *102*
4.3.5 注册现有域名的子域名 *103*
4.3.6 从DNS智能合约获取数据 *103*
4.3.7 解析TON DNS 域名 *103*
4.3.8 轻节点解析TON DNS 域名 *103*
4.3.9 专用Ton DNS 服务器 *104*
4.3.10 访问智能合约中的数据 *104*
4.3.11 智能合约的get方法 *104*
4.3.12 尝试执行智能合约的get方法 *105*
4.3.13 使用TL方案描述智能合约接口 *105*
4.3.14 智能合约的公共接口 *105*
4.3.15 用户与智能合约的交互 *106*
4.3.16 用户与Ton服务的交互 *106*
4.3.17 通过Ton DNS查询用户接口 *106*
4.3.18 链上和链下服务之间的区别 *107*
4.3.19 轻钱包应用和Ton浏览器可内置到Telegram客户端 *107*
4.3.20 支付HTTP接口的Ton网站 *107*
4.3.21 Ton超链接 *107*
4.3.22 Ton超链接可附带参数 *107*
4.3.23 POST操作 *108*
4.3.24 TON WWW *108*
4.3.25 TON WWW的优点 *108*
5. TON支付 109
5.1 支付通道 109
5.1.1 支付通道的思路 *109*
5.1.2 无需信任的支付通道 *110*
5.1.3 简单的双向同步无需信任的支付通道 *110*
5.1.4 同步支付通道：一个带有两个验证者的简单虚拟区块链 *111*
5.1.5 异步支付通道：一个带有两个工作链的虚拟区块链 *112*
5.1.6 单向支付通道 *112*
5.1.7 更复杂的支付通道：承诺（Promises） *112*
5.1.8 复杂的支付通道的智能合约面临的挑战 *113*
5.19 Ton VM支持“智能”支付通道 *113*
5.1.10 在已存在的支付通道创建简单的支付通道 *114*

5.2 支付通道网络，或「闪电网络」 114
5.2.1 支付通道的限制 *115*
5.2.2 支付通道网络 *115*
5.2.3 支付通道网络概述 *115*
5.2.4 链式资金转账 *115*
5.2.5 支付通道链中的虚拟支付通道 *116*
5.2.6 在闪电网络中寻找路径 *116*
5.2.7 优化 *117*
5.2.8 结论 *117*
结论 **118**
参考 **119**
附录 TON COIN，又名GRAM **121**
A.1 细分和术语 121
A.2 用于表示GAS 价格的小单位 122
A.3 初始供应，挖矿奖励和通货膨胀 122
A.4 TON COIND的初始价格 123
A.4.1 指数式定价的加密货币 *123*
A.4.2 前n个币的总价 *123*
A.4.3 接下来Δn 个币的总价 *123*
A.4.4 总值T可以购买的币的数量 *124*
A.4.5 Ton coin的市场价格 *124*
A.4.6 回购Ton coin *124*
A.4.7 以更高价格售卖Ton coin *125*
A.5 未分配的TON COIN的使用 125
A.5.1 一些未分配的Ton coin会给予开发者 *125*
A.5.2 Ton基金会需要一定的Ton coin用作运营目的 *126*
A.5.3 从储备金中提取预先安排的金额 *126*
A.6 批量销售TON COIN 127

1.TON组件的简要描述

The Open Network (TON)包含了以下组件：

- 一个灵活的多区块链平台(TON 区块链;参考第2章)，能够处理每秒数百万笔交易，拥有图灵完备的智能合约、可升级的区块链、支持多加密货币资产转移、支持微支付通道和链下支付网络。TON 区块链提供了一些独特的新功能，例如「自我修复」垂直区块链机制(参考2.1.17)和超立方体路由（Instant Hypercube Routing (参考2.4.20)），使其具有快速、可靠、可扩展和一致性的特点。

- 一个点对点网络(TON P2P 网络,或简称TON 网络;参考第4章)，用于访问 TON 区块链、发送候选交易，以及只接收客户端感兴趣的区块链的部分更新（例如， 与客户端的账户和智能合约相关的部分），同时支持任意的分布式服务，无论服务是否与区块链相关。

- 一种分布式文件储存技术(TON 存储); (参考5.1.8)，通过TON 网络访问， TON 区块链使用Ton 存储 保存区块和状态数据（快照） 的存档副本，但也可以用于储存平台上的使用者或其他服务的任意文件，使用类似于torrent 的访问技术。

- 一个网络代理/匿名层(TON 代理); (参考5.1.11和4.0.6)，类似于IIP (Invisible Internet Project)，如果有必要（例如，从拥有大量加密货币的账户提交交易的节点，或者希望隐藏其确切IP 地址和地理位置，以对抗DDoS 攻击的高风险区块链验证节点），可以用于隐藏TON 网络节点的身份和IP 地址。

- 一个类似Kademlia 的分布式哈希值表(TON DHT;参考4.1)，用作TON 存储的流追踪器（「torrent tracker」(参考4.1.10)），作为TON Proxy的输入隧道定位器(「input tunnel locator」(参考 4.1.14)），以及作为TON Services的服务定位器(参考4.1.12)。

- 一个提供任意服务的平台(TON 服务;参考第4章)，可通过TON 网络和TON 代理访问，具有正式的界面(参考4.3.14)，使得浏览器或智能手机应用程序可以与其交互。这些正式界面和持久的服务入口点，可以在TON 区块链中发布(参考4.3.17)；在给定的任何时刻，提供服务的实际节点可以通过TON DHT，从TON区块链发布的信息开始查找相关信息(参考3.2.12)。服务平台还支持在TON 区块链中创建智能合约，以方便客户使用 (参考4.1.7)。

- TON DNS (参考4.3.1)，用于为账户、智能合约、服务和网络节点分配易读的名称。

- TON 支付 (参考第5章)，一个用于微支付、微支付通道和微支付通道网络的平台。它可用于快速的链下资金转移，以及使用TON服务提供的服务进行支付。

- TON 可以轻松集成第三方消息和社交网络应用程序，从而使区块链技术和分布式服务，可被普通使用者访问和应用(参考4.3.24)，而不仅仅是少数早期的加密货币采用者使用。我们将在另一个项目中，Telegram Messenger (参考4.3.19)，提供这样的集成例子。

虽然TON区块链 是TON项目的核心，其他的组件可能被只被视为辅助角色，但这些组件其实也具有很多实用的功能。通过结合使用这些组件，将会创建出比仅使用TON 区块链更多样化的应用程序(参考2.9.13和4.1)

2.TON区块链

我们从The Open Network (TON) 区块链开始描述，这是该项目的核心组件。我们这里的方法是「由上而下」的：我们首先给出整体的一般描述， 然后提供每个组件的更多细节。

为了简单起见，我们在此谈论的TON 区块链，虽然原则上这种区块链协议可能有多个独立运行的实例（例如，由于硬分叉的结果），我们只考虑其中之一。

2.1 二维区块链集合的Ton区块链

TON 区块链 实际上是区块链的集合（甚至是区块链的区块链集合，或称为2-区块链---这一点将在2.1.17中进一步说明），因为目前没有区块链项目能够达到我们每秒处理数百万交易的目标，现在的标准仅能每秒处理数十次交易。

2.1.1 区块链类型列表

包括以下几种区块链类型：

- 主链：唯一的主区块链，该区块链包含有关协议的一般信息、其参数的当前值、验证节点集合和他们的股份、 当前活跃的工作链(workchains) 及其分片（shards） ，以及最重要的，所有workchains 和分片链的最近区块的哈希值集合。

- 工作链：数个(最多2的32次方)的工作链，实际上是这系统的“工作马”（workhorses），包含资金转移和智能合约的交易。不同的工作链可能有不同的规则，意味着有不同的帐户地址格式、不同的交易格式、不同的智能合约，不同的虚拟机(VMs)、不同的基础加密货币等等。然而，它们都必须满足某些基本的互操作性标准，以确保不同的工作链可以进行相对简单的交互。在这方面，TON区块链是可兼容不同链的（heterogeneous）（参考2.8.8），类似于EOS（参考2.9.7）和PolkaDot（参考2.9.8）项目。

- 分片链：每个工作链会进一步细分为多达2的60次方的分片区块链，它们具有与工作链本身相同的规则和区块格式，但只对帐户的某个子集负责，这取决于帐户地址的几个首位(最重要的位)。换句话说，这系统内建了一种分片(分片）的形式（参考2.8.12）。因为所有这些分片链共享通用的区块格式和规则，TON 区块链在这方面是可以说是同种类的（homogeneous（参考2.8.8）），这与Ethereum 的某个扩展建议相似。https://github.com/ethereum/wiki/wiki/Sharding--FAQ

- 垂直区块链：分片链（和主链）中的每个区块实际上不只是一个区块，而是一个小区块链。通常，这个小区块链或垂直区块链只包含一个区块， 然后我们可能会认为这只是分片链的相对应区块（在这种情况下也称为 水平区块链）。但是，如果需要修正不正确的分片链区块，新的区块将被提交到垂直区块链，包含无效水平区块链区块的替代品，或一个block difference，即只包含该区块先前版本中需要更改的部分的描述。这是一个TON特有的机制，用于替换检测无效区块，而不会真正地分叉所有涉及的分片链；这将在2.1.17中详细解释。目前，我们只需指出，每个分片链（和主链）不是一个常规的区块链，而是一个区块链集合的区块链，或者是一个二维的区块链，或只是一个2-区块链。

2.1.2 无限分片范式

几乎所有的区块链分片提案都是「自上而下」：首先想像一个单一的区块链，然后讨论如何将它分割成几个互动的分片链以提高效能和达到可扩展性。

TON 的分片方法是「自下而上」，如下所述。想像分片已被极端化，以至于每个分片链中只剩下一个帐户或智能合约。然后我们有大量的「帐户链」，每个链描述只有一个帐户的状态和状态过渡，并向彼此发送具有价值的消息以传输价值和信息。当然，拥有数亿的区块链是不切实际的，每个链中的更新（即新的区块） 通常出现得相对较少。为了更有效地实施它们，我们将这些「帐户链」组合成「分片链」，以便分片链的每个区块基本上是已分配给此分片的帐户链的区块的集合。因此，「帐户链」只在「分片链」内部拥有纯粹的虚拟或逻辑存在。我们称这种观点为「无限分片范式」。它解释了TON 区块链的许多设计决策。

2.1.3 消息,超立方体路由

无限分片范式告诉我们将每个帐户（或智能合约）视为它自己的分片链中。然后，一个帐户可能影响另一帐户的状态的唯一方式是向它发送一个「消息」（这是所谓的 Actor 模型(译者注：在计算机科学中，参与者模式（英语：Actor model）是一种并行运算上的模型。"参与者"是一种程序上的抽象概念，被视为并行运算的基本单元：当一个参与者接收到一则讯息，它可以做出一些决策、建立更多的参与者、传送更多的讯息、决定要如何回答接下来的讯息。参与者模式在 1973 年于 Carl Hewitt、Peter Bishop 及 Richard Steiger 的论文中提出。参与者模型推崇的哲学是"一切皆是参与者"，这与面向对象编程的"一切皆是对象"类似，但是面向对象编程通常是顺序执行的，而参与者模型是并行执行的。https://en.wikipedia.org/wiki/Actor_model）的特殊实例，其中帐户作为 Actors；参照 2.4.2）。因此，帐户间（和分片链间，因为源帐户和目的地帐户，一般来说，位于不同的分片链中）的消息系统对于像 TON 区块链这样的可扩展系统非常重要。事实上，TON 区块链的一个新特性，称为「即时超立方体路由」（参照 2.4.20），使它能够将消息从一个分片链的区块传递和处理到目的地分片链的下一个区块，不考虑系统中的分片链总数。

2.1.4 主链，工作链和分片链的数量

Ton区块链中只有一个主链(masterchain)。但是，此系统可以容纳高达2的32次方数量的工作链(workchains)，每个工作链都可细分为高达2的60次方数量的分片链(分片链s)。

2.1.5 工作链可以是虚拟区块链，而不是真正的区块链

由于工作链通常被细分为分片链，工作链的存在是「虚拟的」，这意味着它不是一个真正的区块链，如2.2. 1下提供的一般定义所描述，而只是一组分片链的集合。当只有一个分片链对应到一个工作链时，这个独特的分片链可能与工作链相同，这样它在某个时间点变成一个「真正的」区块链，进而与常规的单一区块链设计有相似性。然而，无限分片范式(参照 2.1.2)告诉我们这种相似性确实是表面的：能够将潜在的大量「帐户链」暂时分组到一个区块链只是一个巧合。

2.1.6 工作链的识别

每一个工作链都由其数字或工作链标识符（workchain_id : uint32）来识别，它只是一个无符号的32位整数。工作链是由主链中的特殊交易所创建，定义（先前未使用的）工作链识别符和工作链的正式描述，至少足以让此工作链与其他工作链互动以及对此工作链的区块进行表面验证。

2.1.7 创建和激活新的工作链

新的工作链可以由社区中的任何成员创建，只要他们准备支付发布新工作链的正式规范所需的（高额）主链交易费用。但是，为了使新的工作链变得活跃，需要三分之二的验证节点达成共识，因为他们需要升级他们的软件以处理新工作链的区块，并通过特殊的主链交易表示他们准备好与新的工作链一起工作。对新工作链的创建感兴趣的一方可能会提供某些激励，让验证节点透过智能合约分发的某些奖励来支持新的工作链。

2.1.8 识别分片链

每个分片链(分片链) 都由一对(w, s) = (workchain_id, shard_prefix)来识别，其中workchain_id : uint32识别相应的工作链(workchain)，而shard_prefix : $2^{0\ldots60}$是一个最长为60的位元串，定义此分片链所负责的帐户子集。换句话说，所有以shard_prefix开头的帐户account_id （即，具有shard_prefix作为最重要位元）都将被分配到这个分片链。

2.1.9 识别账户链

回忆一下，帐户链(account-chains) 只有虚拟存在(参照 2.1.2)。然而，它们有一个自然的识别符 即， (workchain_id, account_id) 因为任何帐户链都包含关于恰好一个帐户（无论是简单帐户还是智能合约 这里的区别不重要）的状态和更新的信息。

2.1.10 分片链的动态拆分和合并（参照 2.7.）

一个较不复杂的系统可能使用静态分片，例如，使用account_id的前八位来选择256个预定义的分片之一。

TON 区块链的一个重要特点是它实现了动态分片，这意味着分片的数量不是固定的。相反，如果满足某些正式条件（基本上，如果原始分片上的交易负载在很长的时间内都足够高），分片(w, s) 可以自动细分为分片(w, s.0)和(w, s.1)。相反，如果负载在一段时间内保持得太低，分片(w, s.0) 和(w, s.1) 可以自动合并回分片(w, s)。

因此一开始，工作链w 只创建了一个分片(w, ∅)。稍后，当这变得必要时， 它被细分为更多的分片(参照 2.7.6 and 2.7.8)。

2.1.11 基础工作链或工作链0

虽然可以定义高达2的32次方的工作链(workchains) 并有其特定的规则和交易，但我们一开始只定义一个， 即workchain_id = 0。这个工作链被称为工作链0（Workchain Zero ）或基础工作链， 它是用于操作TON smartcontracts和转移Toncoin，也称为Grams (参照附录 A)。大多数应用可能只需要使用到工作链0。基础工作链的分片链会被称为基础分片链。

2.1.12 区块出块时间

我们预计每个分片链和主链大约每五秒会生成一个新的区块。这将导致相对较小的交易确认时间。所有分片链的新区块大约同时生成；主链的新区块大约在一秒后生成，因为它必须包含所有分片链的最新区块的哈希值。

2.1.13 通过主链使工作链和分片链紧密耦合

一旦分片链的区块的哈希值被合并到主链的区块中，该分片链区块及其所有祖先都被认为是「正式化的（canonical）（译者注：表示这些区块已经被正式确认）」，这意味着它们可以被所有分片链的后续区块引用为固定且不可变的内容。实际上，每个新的分片链区块都包含最近的主链区块的哈希值，并且从该主链区块引用的所有分片链区块在新区块中都被认为是不可变的。

从本质上讲，这意味着在分片链区块中提交的交易或消息可以在其他分片链的下一个区块中安全地使用，而不需要等待。例如，在转发消息或基于之前的交易采取其他操作之前，需要二十次确认（即在同一区块链中在原始区块之后生成的二十个区块），这在大多数建议的「松散连接」系统中很常见(参照 2.8.14)，如EOS。我们相信，这种能力在提交后的五秒内在其他分片链中使用交易和消息是我们这种「紧密连接」系统能够提供前所未有的性能的原因之一(参照 2.8.12 and 2.8.14)。

2.1.14 主链的区块哈希值作为一个全局变量

根据2.1.13，最后一个主链区块的哈希值完全确定了外部观察者的整体系统状态。人们不需要单独监视所有分片链的状态。

2.1.15 验证节点出新块的过程（参照 2.6.）

TON 区块链使用Proof-of-Stake (PoS) 方法在分片链和主链中生成新的区块。这意味着有一组，例如，多达几百个的验证节点（验证节点是一种特殊的节点，透过特殊的主链交易存放stakes（大量的Toncoin）以符合生成和验证新区块的资格）。

然后，小部分的验证节点被分配给每一个分片(w, s)，这是以确定的伪随机方式进行的，大约每1024个区块会改变一次。这些验证节点通过从客户端收集合适的建议事务到新的有效区块块候选者，并提议就下一个区块的内容达成共识。对于每个区块，验证者之间有一个伪随机选择的顺序，以确定谁的候选区块在每一轮中有最高的优先顺序被提高。

验证节点和其他节点检查所提议的候选区块的有效性；如果验证节点签署了一个无效的候选区块，它可能会自动被惩罚，失去部分或全部的stake，或被暂停进入验证节点集合一段时间。之后，验证节点应达成对下一个区块的选择的共识，本质上是BFT (Byzantine Fault Tolerant; 参照 2.8.4)共识协议的高效变体，类似于PBFT [4] 或Honey Badger BFT [11] 。如果达成共识，将创建新的区块，验证节点会在创建这个新区块里包含的所有交易的交易费用上进行分割（译者注：举例：假如有10个验证节点，这个区块包含100条交易，总交易费为4 ton，则这4 ton会分配给这10个验证节点），再加上一些新创建的（“铸造的”）币（译者注：即Toncoin）。

每个验证节点可以被选举参与多个验证节点子集；在这种情况下，它将同时运行所有的验证和共识算法。

在生成所有新的分片链区块之后或出块超时后，将生成一个新的主链区块，包括所有分片链的最新区块的哈希值。这是通过所有验证节点的BFT 共识完成的。（实际上，三分之二的stake就足够达成共识，但系统会尽可能收集更多的stake授权签名。）

TON PoS方法及其经济模型的更多细节提供在2.6章。

2.1.16 主链的分叉

我们紧密耦合的方法带来的一个复杂性是，切换到主链的不同分叉，几乎必然需要在一些分片链中切换到另一个分叉。另一方面，只要主链中没有分叉，分片链中的分叉是不可能的，因为分片链的替代分叉中的没有区块可以通过将其哈希值纳入主链区块而变得被正式确认。

一般的规则是，如果主链区块B′是B 的上一个区块，B′包括Hash(B′w,s)和分片链区块B′w,s，而B包括Hash(Bw,s)，那么B′必须是Bw,s的前辈；否则，主链区块B就是无效的。

我们预计主链分叉将会很少，几乎不存在，因为在由TON区块链所采用的BFT范畴中，它们只能在大部分验证节点行为不正确的情况下发生（参见2.6.1和2.6 .15），且这将意味着违反者将承担重大的stake 损失。因此，分片链中不会存在真正的分叉。相反，如

果检测到一个无效的分片链区块，将通过2-区块链 的「垂直区块链」机制进行修正（参见2.1.17），这可以在不分叉「水平区块链」（即，分片链）的情况下实现此目标。同样的机制也可以用来修正主链区块中的非致命性错误。

2.1.17 修正无效分片链区块

通常，只有有效的分片链区块会被提交，因为分配给分片链的验证节点在新区块可以被提交之前必须达到三分之二的拜占庭共识。在这之前，系统必须检测先前提交的区块是否存在无效区块，然后对其进行修正。

当然，一旦找到一个无效的分片链区块不论是由一个验证节点（不一定分配到这个分片链）还是一个渔夫节点（「fisherman」）（渔夫节点可以是系统的任何节点，前提是它已经质押了一定的ton coin以便对区块有效性提出质疑；参见2.6.4）。当无效性的区块和证明被提交到主链式，那些已经签署无效区块的验证节点将被惩罚， 部分或全部扣除他们的stake， 并且/或者 暂时将这些验证节点移出验证节点集合（后一种的措施，对于防范攻击者窃取本质上是好的验证节点的私钥进行攻击很重要）。

然而，这还不够，因为由于先前提交的无效分片链区块，系统（TON 区块链）的整体状态结果是无效的。这个无效区块必须被一个新的有效版本替换。

大多数系统会通过「回滚」到当前分片链中的无效区块之前的最后一个区块和其他分片链中没有受到无效区块影响的最后一个区块， 并从这些区块创建一个新的分叉来实现这一点。这种方法的缺点是，大量本来正确且已提交的交易突然被回滚，且不清楚它们是否会在稍后再次被包含到区块中。

TON 区块链通过使每个分片链和主链的每个「区块」（「水平区块链」） 本身都成为一个小型区块链（「垂直区块链」），包含这个「区块」的不同版本，或其「差异」来解决这个问题。通常，垂直区块链只包含一个区块，而分片链看起来像一个经典的区块链。但是，一旦一个区块的无效性被确认并提交到主链区块中，该无效区块的「垂直区块链」就被允许在垂直方向上增加一个新区块，替换或编辑无效区块。这个新区块是由当前问题分片链的验证节点子集生成的。

使得一个新的"垂直"区块成为一个有效区块的规则非常严格。特别是，如果无效区块中包含的虚拟"帐户链区块"（参见2.1.2）本身是有效的，则新的垂直区块必须保持这部分不变。

一旦在无效区块上方提交了新的"垂直"区块，它的哈希值就会在新的主链区块中公布（或者更正确地说，在原始主链区块上方的新"垂直"区块中公布，该区块中最初发布了无效分片区块的哈希值），并且进一步将更改传播到任何参照此区块的分片链区块（例如，那些从不正确的区块接收消息的区块）。这可以通过在先前参照"不正确" 区块的所有区块

的垂直区块链中提交新的“垂直”区块来进行修正；新的垂直区块将参照最新（已更正）的版本。同样，严格的规则禁止更改未受到实际影响的帐户链（即，与前一版本中收到的消息相同的帐户链）。通过这种方式，修正不正确的区块产生“涟漪”，最终传播到所有受影响的分片链的最新区块；这些更改也反映在新的“垂直”主链 区块中。

一旦“历史重写”的涟漪到达最新区块，新的分片链区块仅以一个版本生成，仅作为最新区块版本的后继者。这意味着它们将从一开始就包含对正确（最新）的垂直区块的引用。

主链状态隐含地定义了一个映射，将每个“垂直”区块链的第一个区块的哈希值转换为其最新版本的哈希值。这使客户端可以通过其第一个（通常是唯一的）区块的哈希值识别和定位任何垂直区块链。

2.1.18 Ton coins 和多货币工作链

TON 区块链支持多达2的32次方种不同的“加密货币”，通过32 位的currency_id加以区分。新的加密货币可以通过主链中的特殊交易来添加。每个工作链都有一种基础加密货币，并且可以有几种额外的其他加密货币。

有一种特殊的加密货币，currency_id = 0，即TON币，也称为Ton coin （参见附录A）。这是工作链零的基础加密货币。它也用于交易费和验证节点 的权益股份。

原则上，其他工作链可能会以其他代币收取交易费。在这种情况下，应该提供一些智能合约，用于将这些交易费自动转换为Toncoin。

2.1.19 消息传递和价值转移

属于相同或不同工作链的分片链可以互相传送消息。尽管允许的消息的确切形式取决于接收工作链和接收帐户（智能合约），但有一些共同的栏位使跨工作链的消息成为可能。特别是，每条消息可能会附带有一些值，表示一定数量的 Toncoin（Toncoin） 并且/或者附加一些其他注册的加密货币，前提是它们被接收工作链宣布为可接受的加密货币。这种消息的最简单形式是从一个（通常不是智能合约）帐户到另一个帐户的价值转移。

2.1.20 Ton虚拟机

TON Virtual Machine，也缩写为TON VM或TVM，是用于在主链和基础工作链中执

行智能合约程序代码的虚拟机器。其他工作链可能使用其他虚拟机器，与TVM一起使用或直接替代TVM。我们列出了一些特性。它们将在2.3.12、2.3.14和其他地方进一步讨论。

- TVM 将所有资料表示为一系列的(TVM) 细胞（cells）的集合 (参考2.3.14)。每一个cell包含最多128 个资料位元组，以及最多4个指向其他cells的参考。基于"一切皆为cell"的理念(参考2.5.14)，这使得TVM能够处理与TON区块链相关的所有资料，包括区块和区块链全局状态（如果有需要）。

- TVM 可以将任意数据类型的值(参考2.3.12)，表示为TVM cells的树或有向非循环图（DAG）。然后，实际上TVM对处理的数据类型的存在是不知情的； 它只是处理cells。

- TVM 内建支持hashmaps (参考2.3.7)。

- TVM 是一个堆栈机器（译者注：类似EVM的堆栈先进先出处理结构）。它的堆栈保存64位整数或cell 参考。

- 支持64-bit, 128-bit 和256-bit 的算术运算。所有n-bit 的算术操作都有三种形式：用于无符号整数，用于有符号整数，以及对于模2的n次方的整数（后者不自动检查溢出）。

- TVM 有从n-bit 转换到m-bit 的无符号和有符号整数，对所有$0 \leq m, n \leq 256$，并具有溢出检查。

- 所有算术操作预设执行溢出检查，这大大简化了智能合约的开发。（译者注：EVM的solidity 0.8版本后也默认支持算术运算溢出检查）

- TVM 有"乘然后移位"和"移位然后除"的算术操作，中间值以更大的整数类型计算；这简化了定点算术的实现过程。

- TVM 提供对位串（bit strings）和字节串（byte strings）的支持。

- 提供对某些预定义曲线算法的支持，包括Curve25519 的256-bit 椭圆曲线加密(ECC)的支持。

- 支持某些椭圆曲线上的Weil配对/映射功能，这对于快速实现zk-SNARKs（zk-SNARKs是实现零知识证明的一种经典方案）很有帮助。

- 支持包括sha256在内的常用的哈希值函数。

- TVM 可以处理默克尔证明（Merkle proof）(参考6.1.9)。

- TVM 提供对"大型"或"全局"智能合约的支持。这类智能合约必须能正确处理分片(参考2.3.18和2.3.16)。常规（本地）智能合约的实现可以是与分片无关的。

- TVM 支持闭包。

- "无脊标签 G-机器"[13]（spineless tagless G-machine ，避免创建图计算机的中间节点） 可以在 TVM 内部轻松实现。

除了"TVM 组件"，还可以为TVM 设计几种高级语言用以编写智能合约。所有这些语言都将具有静态类型，并支持代数数据类型。我们设想以下可能性：

- 类似Java 的命令式语言，每个智能合约都像一个独立的类。

- 一种惰性求值的编程语言(类似Haskell)（译者注：惰性求值功能编程语言具有延迟求值和按需调用特性，特别适用于函数式编程语言，参考 https://en.wikipedia.org/wiki/Lazy_evaluatio）

- 一种热切求职的编程语言(类似ML)（译者注：热切求值编程语言会立即求值，大多数编程语言采用这种策略，参考：https://www.quora.com/What-is-the-difference-between-lazy-evaluation-and-eager-evaluation-Why-does-Haskell-use-lazy-evaluation）

2.1.21 可配置的参数

TON 区块链的一个重要特性是它有许多可配置的参数。这意味着这些可配置的参数

是主链状态的一部分，且可以通过主链中的某些特殊的提议/投票/结果交易来更改，而不需要硬分叉。更改这些参数需要收集三分之二的验证节点投票，以及超过一半的想要参与投票过程的所有其他参与者的投票。

2.2 区块链概述

（译者注：对于熟悉常见的区块链的原理的读者，2.2这一章节可忽略不看）

2.2.1 区块链的定义

一般来说，任何(真实的)区块链都是一系列的区块，每个区块B都包含一个引用至前一个区块的blk-prev(B) （通常是将前一个区块的哈希值包含在当前区块的区块头中）， 以及一个交易的列表。每笔交易都描述了全局区块链状态的某种转换; 一个区块中列出的交易是按顺序应用的，从旧状态开始计算新状态，旧状态是指在前一个区块通过计算后得到的结果状态。

2.2.2 Ton区块链与一般区块链的相关性

请注意，TON 区块链并不是真正的区块链， 而是一种2维的区块链的集合（即，区块链的区块链集合; 参考2.1.1），所以上述内容2.2.1并不直接适用于它。然而，我们从真正的区块链的这些一般性开始，用它们作为我们建构更复杂区块链的基础。

2.2.3 区块链实例和区块链类型

人们经常使用区块链（「区块链」）一词来表示一般的区块链类型和其特定的区块链实例，具体定义为满足某些条件的区块序列。例如， 2.2.1引用的区块链实例。

一个区块链通常是由区块组成的列表（有限序列），由那些满足某些相容性和有效性条件的区块序列组成:

$$\text{区块链} \subset Block^* \qquad (1)$$

更好的定义方式会将区块链说成是一种紧密组合的类型（dependent couple type），由couple (B, v) 组成，第一部分B: $Block^*$表示区块列表，第二部分v : isValidBc(B)表示B的有效性证明。由此，

$$\text{区块链} \equiv \Sigma_{(B:Block^*)} \text{ isValidBc(B)} \qquad (2)$$

我们在这里使用的组合类型表示法主要借用自[16]。

2.2.4 依赖类型理论（Dependent type theory），Coq and TL

注意，我们在此使用的是（Martin-Löf）依赖类型理论，类似于Coq(https://coq.inria.fr)证明助手中使用的。依赖类型理论的简化版本也用于TL (Type Language)（https://core.telegram.org/mtproto/TL），将在TON 区块链的正式规范中使用，描述所有数据结构的序列化以及区块、交易等的布局。

事实上，依赖类型理论提供了一种有用的形式化证明表达。当需要提供某个区块的无效性证明时，这种形式的证明（或它们的序列化）可能会变得很有用。

2.2.5 TL，类型语言

由于TL（Type Language）将被用于TON 区块、交易和网络数据包的正式规范，因此值得简短地讨论。

TL 是一种适用于描述依赖代数类型的语言，允许具有数字（自然数）和类型参数。每个类型都通过几个构造器（constructors）来描述。每个构造器都有一个（人类可读的）识别码和一个名字，这是一个位串（bit string）（默认为32 位整数）。除此之外，构造器的定义包含一个字段列表和与之对应的类型。

构造器和类型定义的集合被称为TL体系（TL-scheme）。它通常保存在一个或多个带有.tl 后缀的文件中。

TL-schemes 的一个重要特性是，它们确定了序列化和反序列化定义的代数类型的值（或对象）的方法。具体来说，当一个值需要被序列化为字节流（stream of bytes）时，首先序列化用于此值的构造器的名称。随后对每个字段进行递归计算序列化。

一个适合序列化任意对象为32位整数序列的TL 的先前版本的描述， 可以在https://core.telegram.org/mtproto/TL上找到。一个新版本的TL，名为TL-B，正在开发中，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（byte of stream），甚至是位流（bit of stream）（而不仅仅是32 位整数），并提供将其序列化为TVM cell树的支持（参照 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 区块和交易作为状态转换运算符

通常，任何区块链(类型) 都有一个关联的全局状态(类型) ，以及一个交易(类型) T。区块链在很大程度上是由交易应用函数所决定的：

$$\text{ev\_trans}' : \text{Transaction} \times \text{State} \rightarrow \textit{State}^{?} \quad (3)$$

这里的$X^{?}$表示类型X抽象的结果。这与我们使用X^{*}表示List X 类似。本质上，类型$X^{?}$的值要么是类型X的值，要么是一个特殊值⊥表示没有一个实际的值(想想空指针)。在我们的例子，我们使用$\textit{State}^{?}$而不是State作为结果类型，因为某个交易从某些原始状态呼叫可能是无效的(想想从账户中提款的金额超过实际存在的金额的情况) 。我们可能更偏好ev_trans'的柯里化版本：

$$\text{ev\_trans} : \text{Transaction} \rightarrow \text{State} \rightarrow \textit{State}^{?} \quad (4)$$

因为一个区块本质上是交易的列表，所以区块的评估函数

$$\text{ev\_block} : \text{Block} \rightarrow \text{State} \rightarrow \textit{State}^{?} \quad (5)$$

可以从ev_trans中衍生出来。它接受一个区块B : Block和前一个区块链状态s : State (可能包括前一个区块的哈希值) 并计算下一个区块链状态s' = ev_block(B)(s) : State，它要么是一个真正的状态，要么是一个特殊值⊥表示下一状态无法被计算(也就是说，从给定的起始状态评估时该区块是无效的。（例如，该区块包含试图从一个空帐户扣款的交易)。

2.2.7 区块链序列号

每个在区块链中的区块B使用其序列号blk-seqno(B) 来表示，从第一个区块开始为零，并在过渡到下一个区块时区块号加一。更正式地表示为，

$$\text{blk-seqno}(B) = \text{blk-seqno}(\text{ blk-prev}(B)\ ) + 1 \quad (6)$$

请注意，区块序列号在有分叉的情况下**无法唯一表示一个区块。**

2.2.8 区块哈希

表示区块B的另一种方式是通过其哈希值blk-hash(B)， 这个值实际上是区块B的头部内容的哈希值（但是，区块的头部内容通常包含依赖于区块B的所有内容的哈希值）。假设所使用的哈希值函数没有发生碰撞（或至少它们是非常不可能的），一个区块可以由其哈希值唯一识别。

2.2.9 哈希假设

在对区块链算法进行正式分析时，我们假设使用的k-bit 哈希值函数（Hash : $Bytes^*$ → 2^k）没有发生碰撞:

$$\text{Hash}(s) = \text{Hash}(s') \Rightarrow s = s' \text{ 对于任何} s, s' \in Bytes^* \quad (7)$$

这里的Bytes = {0 . . . 255} = 2^8是指字节类型，或所有字节值的集合， 而$Bytes^*$是指一种具有任意（有限）字节列表的类型或集合；而2 = {0, 1} 是位类型（bit type），和2^k是所有k-bit 序列的集合（即，k-bit 数字）。

当然，(7) 在数学上是不可能的，因为从一个无穷集合到一个有限集合的映射不能是单映射。一个更严格的假设是

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

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)。

最后总结:为了使(8)的概率陈述足够严格，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$p^l - p^{l+1}$。 然后(8)应该被理解为条件概率的极限

P（Hash(s) = Hash(s')|$s \neq s'$），当 p 从趋 近于 1 时。

2.2.10 哈希在Ton区块链的应用

我们目前为 TON 区块链使用 256-bit 的 sha256 哈希值。如果它被证明比预期要差，未来可以使用另一个哈希值函数取代。选择使用哪种哈希函数是作为参数可配置的，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、帐户和哈希值映射

通过上一章的定义我们可以发现，任何区块链都需要定义一个全局状态，而每个区块和每个交易定义了这个全局状态的转换过程。接下来我们开始描述TON 区块链使用的全局状态。(译者注：对于已经熟悉常见区块链工作原理的读者，2.3.1至2.3.10章节可以忽略不看，直接从2.3.11章节开始阅读。)

2.3.1 账户ID

TON 区块链（包括主链和工作链零）所使用的基本帐户ID是256- bit 整数，假设这个账户ID是针对特定椭圆曲线的256-bit 椭圆曲线密码学(ECC ）的公钥。即，

$$\text{account\_id} : \text{Account} = uint_{256} = 2^{256} \quad (9)$$

这里的Account是帐户的类型，而account_id : Account是类型Account的特定变量。

其他工作链可以使用其他帐户ID 格式，无论是256-bit 还是其他格式。例如，可以使用等于ECC公钥的sha256的比特币风格的帐户ID。

但是，帐户ID 的位长度 L 必须在工作链的创建期间（在主链中）确定， 且必须至少为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

TON 区块链的状态的主要组成部分是HashMaps。在某些情况下，我们考虑（部分定义的）“maps”： $h : 2^n \to 2^m$。 更一般地说，我们可能对于复合类型X 的哈希值映射 $h : 2^n \to X$ 更感兴趣。 但是，源（或索引）的类型几乎总是2^n。

有时，X的默认值为空，对应的 HashMaps： $h : 2^n \to X$ 将被初始化为它的默认值 $i \to$ empty。

2.3.3 例子：Ton账户余额

一个重要的例子是TON 帐户余额。它是一个 hashmap：

$$\text{balance} : \text{Account} \rightarrow uint_{128} \quad (10)$$

左边的Account表示账户地址（2^{256}*位*），$uint_{128}$ 表示Toncoin (TON 币）的余额（2^{128} *位*）。此哈希值映射的预设值为零，这意味着最初（在处理第一个区块之前）所有帐户的余额都是零。

2.3.4 例子：一个智能合约的持久存储

另一个例子是智能合约的持久储存， 可以（非常接近地）表示为一个hashmap：

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

此哈希值映射的预设值也为零，这意味着未初始化的持久储存的cells的内容为零。

2.3.5 例子：所有智能合约的持久存储

因为我们有多于一个的智能合约，以account_id区分，每个合约都有其独立的持久储存，所以我们需要定义这样一个哈希值映射：

$$\text{Storage} : \text{Account} \rightarrow (2^{256} \rightarrow 2^{256}) \quad (12)$$

其中Account表示智能合约的account_id。

2.3.6 Hashmap 类型

Hashmap不仅仅是一个抽象的（部分定义的）函数$2^n \rightarrow X$；它还有具有特定的表示方式。因此，假设我们有一个特定的hashmap类型：

Hashmap(n, X) : Type　(13)

对应2^n →X的数据编码结构。我们也可以写成：

Hashmap(n : nat)(X : Type) : Type　(14)

或

Hashmap : nat → Type → Type　(15)

我们总是可以将h : Hashmap(n, X) 转换为一个映射函数hget(h) : $2^n \to X^?$。之后，我们通常使用 h[i] 代替hget(h)(i)，即

h[i] :≡ hget(h)(i) : $X^?$ 对于任何 i : 2^n , h : Hashmap(n, X)　(16)

2.3.7 将Hashmap定义为Patricia树

从逻辑上讲，我们可能会定义Hashmap(n, X) 为一个深度为n 的（不完整的）二进制树， 其边的标签为0 和1 ，而叶子中的值类型为X。描述相同结构的另一种方式是作为长度等于n 的二进制字符串的(按位) Trie树。(译者注：Trie树又叫字典树或单词查找树，是一种计算机常用的数据结构，https://en.wikipedia.org/wiki/Trie)。

在实际应用中，我们更倾向于使用这种Trie数的紧凑表示，通过压缩每个只有一个子节点的顶点及其父节点。得到的结果称为Patricia tree或binary radix tree。每个中间顶点现在都有确切的两个子节点，由两个非空的二进制字符串标记，左子节点开始为零，右子节点开始为一。

换句话说，在Patricia 树中有两种类型的（非根）节点：

- Leaf(x)，包含类型为X 的值x。
- $\mathrm{Node}(L, s_l, r, s_r)$,其中 L 是左子节点或子树的（引用）， s_l 是连接此顶点到其左子节点的边的位字符串标签（始终以 0 开头）， r 是右子树，s_r 是到右子节点的边的位字符串标签（始终以 1 开头）。

还需要第三种节点类型，只在Patricia 树的根上使用一次，但也是必要的：

- Root(n, s_0, t)，其中n是Hashmap(n, X) 的索引位字符串的公共长度， s_0是所有索引位字符串的公共前缀，t 是指向叶（Leaf） 或节点（Node） 的引用。

如果我们想让Patricia 树为空，则会使用第四种类型的（根）节点：

- EmptyRoot(n)，其中n 是所有索引位字符串的公共长度。

我们通过以下方式定义Patricia 树的高度：

height(Leaf(x)) = 0 (17)

height($\mathbf{Node(L, s_l, r, s_r)}$) = height(L) + len($\mathbf{s_l}$) = height(r) + len($\mathbf{s_r}$) (18)

height(Root(n, s_0, t)) = len(s_0) + height(t) = n (19)

其中（18）和（19）两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 树来表示类型为Hashmap(n, X) 的值。

如果树中有N 个叶子（即，我们的哈希值映射包含N 个值），则刚好会有N −1 个中间顶点。插入一个新值总是涉及通过在中间插入一个新顶点来分割一个现有的边，并添加一个新叶子作为这个新顶点的另一个子 节点。从哈希值映射中删除一个值做的恰恰相反：叶子和它的父节点被删除，并且父节点的父节点和其另一个子节点直接连接。

2.3.8 MPT树（Merkle-Patricia trees）

（译者注：和比特币、以太坊等区块链使用的MPT树类似）

当使用区块链时，我们希望能够比较Patricia 树（即，哈希值映射）及其子树，并将它们缩减为单一的哈希值。实现此目的的经典方法是通过Merkle 树实现。本质上，我们希望描述一种利用哈希值函数Hash（为二进制字符串定义）对类型为Hashmap(n, X) 的对象h进行哈希值的方法，只要我们知道如何计算对象x : X 的哈希值Hash(x)（例如，通过将哈希值函数Hash 应用于对象x 的二进制序列化）。我们会如下地定义Hash(h)：

Hash(Leaf(x)) := Hash(x) (20)

Hash($\mathbf{Node(L, s_l, r, s_r)}$) := Hash(Hash(L). Hash(r). code($\mathbf{s_l}$). code($\mathbf{s_r}$)) (21)

Hash(Root(n, s_0 t)) := Hash(code(n). code(s_0). Hash(t)) (22)

在此，s.t 表示(位) 字符串s 和t 的连接，而code(s) 是所有位字符串s的前缀码。例如，可以通过10 来编码0，通过11 来编码1，并通过0 来编码字符串的结尾。（可以证明对于大约一半的Patricia 树的边标签（具有随机或连续索引）来说，这种编码是最优的。其余的边标签可能会很长（即，几乎有256 位）。因此，边标签的几乎最优编码是使用上述码，对于「短」位字符串使用前缀0，然后编码1，然后是包含位字符串s 的长度l = |s| 的九位，然后是s的l 位，用于「长」位字符串（其中l ≥ 10）。）

我们稍后会看到（参见 2.3.12和2.3.14），这是针对任意（依赖型）代数类型的值的递归定义的哈希值的（稍微调整的）版本。

2.3.9 重新计算Merkle树的哈希值

这种递归定义Hash(h) 的方法，称为Merkle tree hash，具有以下优点：如果与每个节点h'一起储存Hash(h') （结果在结构上被称为Merkle tree，或在我们的例子中，称为Merkle–Patricia tree），则当元素被添加到哈希值映射、从哈希值映射中删除或更改哈希值映射时，最多只需要重新计算n个哈希值。因此，如果将全局区块链状态表示为适当的Merkle 树哈希值，则在每次交易后，重新计算此状态哈希值就变得很容易。

2.3.10 默克尔证明（Merkle proofs）

根据2.2.9章中描述的哈希假设(7)， 可以构造一个证明，对于一个给定的值z ，z=Hash(h), h : Hashmap(n, X)，存在某些i : 2^n和x : X 使得hget(h)(i)= x。这样的证明将包括从对应于i 的叶子到根的Merkle–Patricia 树中的路径，还包含路径上出现的所有节点的所有兄弟节点的哈希值。

这样，一个轻节点（light node）（轻节点是一种不会跟踪分片链完整状态的节点；相反，它保留了最少的信息，如最近几个区块的哈希值，并在需要检查完整状态的某些部分时，依赖于从其他全节点（Full node）获得的所需的信息），只知道某些hashmap h的Hash(h) 值（例如，智能合约的持久储存或全局区块链状态），它可能会从全节点（全节点是一种跟踪分片链的完整最新状态的节点）请求不仅仅是值x = h[i] = hget(h) (i)，还会获取伴随着从已知值Hash(h)开始的Merkle 证明的这样一个值。然后，在假设(7) 下，轻量节点可以自己检查x 确实是h[i] 的正确值。

在某些情况下，客户端可能希望获得值 y = Hash(x)= Hash(h[i])，例如，如果x 本身非常大（例如，是一个hashmap）。然后，可以提供(i, y) 的Merkle 证明。如果x 也是一个hashmap，那么可以从全节点获得从y = Hash(x) 开始的第二个Merkle 证明，以提供值x[j] = h[i][j] 或仅其哈希值。

2.3.11 对于像TON 这样的多链系统，默克尔证明的重要性

请注意， 一个节点通常不会是TON 环境中存在的所有分片链的全节点。它通常只是某些分片链的全节点。例如，包含其自己的帐户，它感兴趣的智能合约，或者该节点已被指派为其验证节点的那些分片链。对于其他分片链，它必须是一个轻节点，否则储存、计算和网络带宽的需求量将十分巨大。这意味着这样的节点不能直接检查关于其他分片链状态的有效性；它必须依赖于从那些分片链的全节点去获得的Merkle 证明。除非哈希假设(7) 失败（即，找到一个哈希值碰撞），使用这样的方式同样安全。

2.3.12 TON VM的特点

TON VM 或TVM (Telegram Virtual Machine)，用于在主链和工作链0中运行智能合约，与EVM (Ethereum Virtual Machine) 的常见设计有很大的不同：它不仅仅支持256位整数，实际上它支持（几乎）任意的「纪录」、「结构」或「和积类型（sum-product types）」，使其更适合执行用高级语言编写的程序代码。实际上，TVM 使用的是带有标签的数据类型，这与Prolog 或Erlang 的实现中使用的不太相同。

可以首先想像，TVM 智能合约的状态不仅仅是一个hashmap(256,2^{256})或Hashmap $2^{256} \rightarrow 2^{256}$。对于Hashmap(256, X)， 其中X是具有几个构造器的类型，使其除了256位整数之外，还能储存其他数据结构，尤其是其他的Hashmap(256,X)类型。这意味着TVM （持久或临时）储存的一个cells，或者一个在TVM智能合约程序代码中的变量或数组元素，可能不仅包含一个整数，还包含一个全新的hashmap。当然，这意味着一个cells不仅仅包含256位，还包含，例如，一个8位的标签，描述如何解释这256位的数据。

事实上，值不需要一定是256位的。 TVM 使用的值格式由原始字节和对其他结构的引用组成，这些引用以任意顺序混合，并在合适的位置插入一些描述字节，以便能够区分指针和原始数据（例如，字符串或整数）； 请参见2.3.14。

这种原始值格式可以用来实现任意的和积代数类型。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「构造器」（从高级语言的角度看），然后是其他「字段」或「构造器参数」，由原始字节和对其他结构的引用组成，具体取决于选择的构造器（参考2.2.5）。然而，TVM 并不知道构造器及其参数之间的对应关系；因此需要使用某些字节明确描述出来。（这两个描述字节，在任何TVM cells 中都存在，仅描述引用总数和原始字节总数；引用总是放在所有原始字节之前或之后。）

Merkle 树哈希值被扩展到任意这样的结构：要计算这样一个结构的哈希值， 所有的引用都被递归地替换为被引用对象的哈希值，然后计算结果字节串（包括描述字节）的哈希值。通过这种方式，对hashmaps 的Merkle 树哈希值，如2.3.8所述，只是用于类型Hashmap(n, X) 的两个构造器的任意代数数据类型的哈希值的特殊情况。（实际上，Leaf和Node是辅助类型HashmapAux(n, X) 的构造器。类型Hashmap(n, X) 有构造器 Root 和EmptyRoot， 其中Root 包含类型HashmapAux(n, X) 的值。）

2.3.13 Ton智能合约的持久存储

TON 智能合约的持久性储存主要由其「全局变量」组成，这些变量在调用智能合约

之间保持不变。因此，它只是一个「产品（product type，译者注：所谓产品类型，就是一种组合的机构类型 https://en.wikipedia.org/wiki/Product_type）」、「元组」或「记录」类型，由每一个全局变量的正确类型的字段组成。如果全局变量太多，因为TON cells 大小的全局限制，它们不能都放入一个TON cells。在这种情况下，它们被分割成几个记录并组织成一棵树，基本上变成了「产品的产品」或「产品的产品的产品」类型，而不仅仅是一个产品类型。

2.3.14 TVM Cells

最终，TON VM 通过在(TVM) cells集合中保存所有数据。每个cells 首先包含两个描述符字节，表示此cells 中有多少原始数据字节（最多128）以及有多少对其他cells 的引用（最多四个）。然后是这些原始数据字节和引用。每个cells 只被引用一次，所以我们可能已经在每个cells 中包括了对其「父级」的引用（唯一引用此cells 的cells）。但这种引用不必是显式引用。

通过这种方式，TON 智能合约的持久数据储存cells 被组织成一棵树（逻辑上；在2.5.5中描述的「bag of cells」表示法会识别所有重复的cells，当序列化时，将此树转换为一个有向无环图（dag）），智能合约描述中保留了对这棵树的根的引用。如果需要，可以递归计算这整个持久储存的Merkle 树哈希值，从叶子开始，然后简单地将一个cells中的所有引用替换为所引用的cells的递归计算的哈希值，然后计算所得字节串的哈希值。

2.3.15 任意代数类型值的一般化Merkle证明

由于TON VM 通过使用(TVM) cells 组成的树来表示任意代数类型的值，且每个cells 都有一个明确定义的（递归计算的）Merkle哈希值，实际上依赖于此cells 为根的整个子树，我们可以为任意代数 类型的值（的部分） 提供「一般化的Merkle 证明」，旨在证明具有已知Merkle哈希值的树的某个子树，具有某个特定值或特定哈希值。其中2.3.10描述的默克尔证明方法，只考虑了x[i] = y 的Merkle 证明。

2.3.16 支持对Ton VM数据结构进行分片

（译者注：这是Ton的一个比较关键的特性,相当于实现了传统关系型数据库对大数量

的表的进行自动水平切分。）

我们刚刚概述了如何在不过于复杂的情况下，使得TON VM 支持高级智能合约语言中的任意代数数据类型。然而，对于大型（或全局）智能合约的分片，需要TON VM 级别上的特殊支持。为此，系统中增加了hashmap 类型的特殊版本，相当于一个「映射」Account → X。这个「映射」可能看起来等同于Hashmap(m, X)，其中Account = 2^m。但是，当一个分片被分成两个，或两个分片被合并时，这样的hashmaps 会自动被分成两个，或合并回来，并保留只属于相应分片的键。

2.3.17 为持久存储支付费用

（译者注：这是Ton与其他常见区块链的一个重要的不同点，在Ton上部署完智能合约后，会持续为智能合约的持久存储收取费用。而像其他EVM兼容的区块链，只需支持一次部署智能合约的Gas fee，后续无论智能合约存储了多少数据，都不会再向智能合约收取费用。）

TON 区块链的一个值得注意的特点是从智能合约中扣除用于储存其持久数据（例如区块链的全局状态）的费用。它的工作原理如下：

每个区块都宣布两种费率，以区块链的主要货币（通常是 Toncoin）设定以下价格：保存一个cells在持久储存中的价格，以及在持久储存的某个cells 中保持一个字节的价格。因为每个账户使用的cells和字节的总数据作为其状态的一部分储存，所以通过将这些数字乘以在区块头中宣布的两个费率，我们可以计算出从账户余额中需要扣除的费用，以继续保存其数据在前一个区块和当前区块之间。

然而，对于每个普通账户和智能合约在每个区块中的持久储存使用的费用并不是每次都收取的；而是在账户数据中储存上次收取此费用的区块的序列号，并且当对账户进行任何操作时（例如，转移资金或接收并由智能合约处理一条消息），在执行任何进一步的操作之前，从账户余额中扣除自上次这样的支付以来的所有区块的储存使用支付。如果账户的余额在此之后变为负数，则该账户将被销毁。（译者注：这种收费模式相当于每次有交易发生时再计算需要收取的存储费用，另外，如果账户的余额不足以支付存储费用，最新的Ton区块链并不会立刻销毁账户，而是进入冻结状态，只要转入足够支付存储费用的Ton coin就可以解冻该账户。）

为了创建一个「简单」的账户，工作链可能会宣称每个账户的一些原始数据字节是「免费的」（即，这些数据不参与持久储存的费用支付），只保持它们在一或两种加密货币的余额，以避免持续地需要支付存储费用。

请注意，如果没有人给一个账户发送任何消息，那么这个账户的需要支付的持久储存费用不会被收集（即使账户中有一定的余额），并且它可以无限期地存在。然而，任何人

都可以发送，例如，一条空消息来销毁这样的账户。可以给发送这样一条消息的人提供一个小的激励，从要被销毁的账户的原始余额中收取部分资金。然而， 我们预计验证节点会销毁这样的无资金账户，为了减少全局区块链状态的大小，避免保留大量的数据而不得到任何补偿或费用。（译者注：因为验证节点存储越大量的数据，理论上需要支付更多的费用，无论是升级本地的硬件设别还是购买更多的云服务器资源。）

收集到的账户的持久存储费用会分配给分片链或主链的验证节点，对于主链验证节点，获得的费用按照验证节点质押的Ton coin比例进行分成。

2.3.18 本地和全局智能合约以及智能合约实例

一个智能合约通常只存在于一个分片中，根据智能合约的account_id进行选择， 与「普通」账户类似。这通常对大多数应用程序来说都是足够的。然而，一些「高负载」的智能合约可能希望在某个工作链的每个分片链中都有一个「实例」。为了实现这一点，它们必须将它们的创建交易传播到所有的分片链中，例如，通过将此交易提交到工作链w 的「根」分片链(w, ∅) 中（一个更昂贵的选择是在主链中发布这样一个「全局」智能合约），并支付一大笔费用。（因为需要广播到所有的分片链，因此，它的费用一定非常昂贵。）

这个动作会在每个分片中创建了智能合约的实例，并具有单独的余额。最开始，创建交易中传输的余额，会平均地给每个分片(w, s) 的分配的总余额*$2^{-|s|}$。（译者注：分片（w，s），根据2.1.8的定义，每个分片链(分片链) 都由一对(w, s) = (workchain_id, shard_prefix)来识别，其中shard_prefix：$2^{0\ldots60}$是一个最长为60的位元串，因此每个分片链平均分到的余额比例等于$1/2^{s}$，即$2^{-|s|}$）。当一个分片分裂成两个子分片时，所有全局智能合约的实例的余额都分裂为一半；当两个分片合并时，余额会相应加在一起。

在某些情况下，分裂/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，余额按照上述方式分裂和合并，一些特殊的「账户索引」的 hashmaps 也是自动分裂和合并的（参见2.3.16）。

2.3.19 限制智能合约的拆分

一个全局智能合约可以在其创建时限制其分裂深度d，以使持久储存费用更具可预测性。这意味着，如果分片链(w, s) 满足$|s| \geq d$ 被分裂成两个，两个新分片链中，只有一个继承了智能合约的实例。（译者注：限制合约的最大分裂深度，本质上限制了合约的最大可分裂出的实例数，当已到达最大合约实例数时，这时在任一个合约实例所在的分片链里，再继续分片更多分片链的话，也只会有一个分片链会继承这个合约实例，其他分片链不会产生更多的合约实例。）选择哪个分片链继承智能合约实例是具有确定性的：每个全

局智能合约都有一个「account_id」，本质上是其创建交易的哈希值，然后它的实例地址除了「account_id」，还需要在前面加上≤ d 位的适当的内容，以表示不同的分片。（译者注：假设d=2，则类似这种表示方式：01:ContractHash，02:ContractHash）这个account_id决定了分裂后哪个分片将继承智能合约实例。）

2.3.20 账户/智能合约状态

我们可以总结以上所有内容， 得出账户或智能合约的状态包括以下内容：

- 区块链的基础加密货币的余额
- 区块链其他加密货币的余额
- 智能合约的程序代码（或其哈希值）
- 智能合约的持久性数据（或其Merkle 哈希值）
- 持久性储存cells 和字节使用数量的统计
- 上次收取智能合约持久性储存费用的时间（实际上，是主链区块号）
- 发送加密货币和从此账户发送消息所需的公钥（可选；默认等于account_id本身）。在某些情况下，更为复杂的签名检查程序代码可能位于此处，与比特币交易输出类似；然后， account_id将等于此程序代码的哈希值。

我们还需要在某处保存以下数据，无论是在账户状态中还是在某个其他的账户索引的hashmap中：

- 账户的输出消息队列（参见2.4.17）
- 最近传递的消息的（哈希值的）集合（参见2.4.23）

并不是每个账户都需要所有这些内容；例如，只有智能合约才需要智能合约程序代码这部分内容，而「简单」的账户则不需要。此外，尽管任何账户必须在主要加密货币中有一个非零余额（例如，基础工作链的主链和分片链的Toncoin），但在其他加密货币中可能有零余额。为了避免保留未使用的数据，（在工作链的创建期间）定义了一个和积类型(sum-product type)（取决于工作链），它使用不同的标签字节（例如，TL 构造器；参见2.2.5）来区分使用的不同「构造器」。最终，账户状态本身被保存为TVM持久储存的cells集合。

2.4 分片链间的消息

TON 区块链的一个重要组件是区块链间的消息系统。这些区块链可以是同一工作链的分片链，或者是不同工作链的分片链。

2.4.1 消息、帐户和交易 概览

消息从一个账户发送到另一个账户。每一个交易的过程包括：一个账户接收一个消息，然后根据某些规则改变其状态，并生成到其他账户的多个（可能是一个或零个）新消息。每条消息生成并接收（传递）一次。

这意味着消息在系统中扮演了基本的角色，与账户或智能合约的角色相当。从无穷分片范式的角度看（参见2.1.2），每个账户都位于其独立的「账户链」中，并且它唯一可以影响其他账户的状态的方式是通过发送消息。

2.4.2 使用参与者模型（Actor model）的账户

可以将账户（和智能合约）视为「进程」或「Actor」，它们能够处理进入的消息、改变其内部状态，并因此生成一些出站消息。这与所谓的Actor model十分类似， 该模型在Erlang之类的语言中使用（但是，Erlang 中的角色通常称为「进程」）。由于新的 Actors（即，智能合约）也允许由现有Actor作为处理入站消息的结果来创建，因此与 Actor model 的对应基本上是完整的。

2.4.3 消息接收者

任何消息都有其接收者，使用目标工作链识别符w（默认情况下与原始分片链相同）和接收账户account_id进行描述。account_id的确切格式（即，位数（number of bits））取决于w；但是，分片始终由它的前（最重要的）64 位的内容确定。

2.4.4 消息发送者

在大多数情况下，消息都有一个发送者，使用(w'，account_id') 进行描述。如果存在消息发送者，它位于消息接收者和消息值之后。有时，发送者并不重要或者来自区块链之外的消息（即，不是智能合约）， 在这种情况下，此字段可以不存在。

值得注意的是，Actor model 并不要求消息有一个隐式发送者。相反，消息可能包含

回复发送请求的Actor的引用，它通常与发送者一致。但是，在加密货币（Byzantine）环境中（译者注：这里指可能存在恶意节点或十分复杂的环境），在消息中有一个明确的不可伪造的发送者字段是很有用的。

2.4.5 消息的附加值

消息的另一个重要特性是其附加的值，它是源工作链和目标工作链均支持的一种或多种加密货币。消息的值在消息接收者之后的开头处指示；它实际上是一系列的(currency_id, value)对。（译者注：这和以太坊发送交易附带eth token是类似。不同点是Ton附加的是一个map列表，比如像[{ton:3.5},{jusdt:20.5}]）

请注意，普通账户之间的「简单」值转移（译者注：及Ton或者其他加密货币转移）只是带有某些附加值的空（无操作）消息。另一方面，略为复杂的消息主体可能包含一个简单的文本或二进制评论（例如，关于付款的目的）。

2.4.6 外部消息：来自区块链外部的消息

有些消息是并清楚消息来自哪里的，也就是它们并非由区块链中的账户（无论是否是智能合约）生成的。最典型的例子是当用户希望从她控制的账户转移一些资金到另一个账户时。在这种情况下，用户通过一个客户端发送一个消息到她自己的[普通]账户（智能合约），要求它生成一个发送给接收账户的消息，并带有指定的值。如果此消息被正确签名，她的账户就会接收到它并生成所需的出站消息。

实际上，「普通」账户可以认为是带有预定义程序代码的智能合约的特例。这种智能合约只接收一种类型的消息。这种入站消息必须包含处理入站消息而生成的出站消息列表，以及一个签名。智能合约会检查消息的签名，如果它是正确的，则生成所需的消息。

当然，来自区块链外部的消息和普通消息之间有所不同，来自区块链外部的消息不能承担处理费用，所以它们不能为自己的「gas」（即它们的处理过程）支付费用。另外，它们在被提议包含在新的分片链区块中之前，会先尝试执行一个小的gas 限制；如果执行失败（签名不正确），则来自区块链外部的消息被视为不正确并被丢弃。如果执行在小的gas 限制内不失败，则消息可能被包含在新的分片链区块中并完全处理，从接收者的账户中扣除所消耗的gas（处理这个过程）的费用。 除了支付给验证节点的gas费用之外，来自区块链外部的消息也可以定义一些额外的交易费用来支付给验证节点，这些费用从接收者的账户中扣除。

在这个意义上，来自区块链外部的消息起到了在其他区块链系统中使用的交易候选人的作用（例如，比特币和以太坊）。

2.4.7 日志消息

有时可以生成一种的特殊消息，然后路由到特定的分片，不是为了发送消息给其接受者，而是为了记录日志消息，以便其他人接收到有关该分片的更新时，都可以轻松监测到相关的消息内容。这些记录的消息可以在用户的控制台中输出，或触发某个链下的服务器的某个脚本的执行。在这种意义上，它们代表了区块链系统的外部「输出」，正如来自区块链外部的消息代表了区块链系统的外部「输入」。（译者注：这种日志消息和以太坊的EVM的event事件是类似的）

2.4.8 与链下服务和外部区块链交互

这些外部输入和输出消息可以用于与链下服务和其他外部区块链进行交互， 就像比特币或以太坊一样。人们可以在TON 区块链中创建与比特币、以太坊和在以太坊区块链中定义的任何ERC-20代币，主要通过某些第三方链下服务上的脚本生成和处理来自外部区块链的消息和日志消息，以实现TON 区块链与这些外部区块链之间的必要互动。

2.4.9 消息体

消息体基本上就是一系列的字节，其含义由接收的工作链和/或智能合约决定。对于使用TON VM 的区块链，可以是通过Send() 操作自动生成的任何TVM cell 的序列化结果。这种序列化是通过递归地替换TON VM cell 中的所有引用来获得的。最终，会出现一个字节字符串，通常在前面加上一个4字节的「消息类型」或「消息建构器」，用于选择要调用的智能合约的正确方法。

另一种方法是使用TL-序列化对象（参见2.2.5）作 为消息主体。这对于不同工作链之间的通信可能尤其有用，因为其中的一个工作链不一定使用TON VM。

2.4.10 Gas 限制和其他工作链 / VM 特定参数

有时消息需要携带有关gas 限制、gas 价格、交易费用和类似的值的信息，携带哪些信息主要取决于接收消息的工作链，并且只与接收的工作链有关，与原始发送消息的工作链无关。这些参数包含在消息主体中或之前，有时（取决于工作链）通过特殊的4字节前缀表示它们的存在（可以由TL-方案定义；参见2.2.5）。

2.4.11 创建新消息: 智能合约和交易

新消息的来源有两个。当智能合约被调用来处理入站消息时，大多数消息是在智能合约执行期间创建的（通过TON VM 中的Send() 操作）。或者，消息可能来自外部，作为

来自区块链外部的消息（参见2.4.6）。 （上述描述只有对于基础工作链及其分片链上是有效的；其他工作链可能提供其他创建消息的方法。）

2.4.12 传递信息

当一条消息到达包含其目的账户的分片链时，它被「传递」到其目标账户。（作为一种退化情况，这个分片链可能与原始的分片链重合，例如，如果我们正在内部工作的工作链尚未被分裂。）接下来会发生什么取决于工作链；从外部观点看，重要的是这样的消息永远不会从这个分片链被进一步转发。

对于基础工作链的分片链，传递的内容包含添加到接收账户余额的消息值（扣除任何gas费用），并可能之后会调用接收智能合约的一个用于处理该消息的方法，如果接收账户是一个智能合约。事实上，一个智能合约只有一个进入点用于处理所有传入消息，并且它必须通过查看它们的前几个位元组来区分不同类型的消息（例如，包含TL 建构器的前四个位元组；参见2.2.5）。

2.4.13 传递消息是一种交易

因为消息的交付更改了账户或智能合约的状态，所以它在接收的分片链中是一个特殊的交易，并被明确地注册为这样的交易。本质上，所有TON 区块链交易都包括将一个入站消息传递给其接收账户（智能合约），如果忽略一些次要技术细节的话。

2.4.14 同一智能合约实例之间的消息

回忆一下，一个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分片链中）或全局的（即，在所有的分片中都有实例，或至少在所有深度为d 的分片中；参见2.3.18）。全局智能合约的实例可能需要交换特殊消息以在彼此之间传递信息和转移资金。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 发送给任一个智能合约实例的消息：通配符地址

有时候一条消息（例如，客户端请求）需要被发送给全局智能合约的任意一个实例，通常是最近的一个（如果有一个驻留在与发件人相同的分片链中，它是明显的候选人）。做到这一点的一种方法是使用「通配符收件人地址」，其中目标account_id的前d位可以取任意值。在具体应用中，人们通常会将这d位设置为与发件人的account_id中的相同值。

2.4.16 输入队列不存在

由区块链接收的所有消息（通常是分片链；有时是主链） 或基本上是驻留在某个分片链 内的「账户链」 都立即被传递（即， 由接收账户处理）。因此，没有像「输入队列」这样的设计。相反，如果由于对区块总大小和gas使用的限制，不是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积累。

2.4.17 输出队列

从无限分片范式（Infinite Sharding Paradigm） 的角度看(参照 2.1.2)，每个账户链（即，每个账户）都有其自己的输出队列， 包括由它生成但尚未传递给其接收者的所有消息。当然，账户链只是一个虚拟的存在；它们被分组到分片链 中，而分片链有一个输出「队列」，由属于该分片链的所有账户的输出队列联合组成。

这个分片链的输出「队列」只对消息确保部分顺序。即，消息的顺序必须遵循区块产生的顺序，并且对于账户和目的地相同的任何消息，必须按照它们的生成顺序传递。（译者注：即最终执行的顺序可能和原始的顺序不同，只需要确保按照以上所描述的顺序执行即可。）

2.4.18 可靠，快速的链间消息传递

对于像TON 这样的可扩展、支持多区块链系统的项目，能够在不同的分片链之间转发和传递消息是十分重要的(参照 2.1.3)，即使系统中有数百万个消息。消息应该被可靠地（即，消息不应该丢失或传递多次）和快速地传递。 TON 区块链通过使用两种「消息路由」机制的组合来实现这一目标。

2.4.19 超立方体路由：为确保消息传递的慢速路径

TON 区块链 使用「超立方体路由」作为一种缓慢，但安全可靠的方法，去从一个分片链传递消息到另一个分片链，如有必要，还会使用几个中间的分片链进行转发。否则，任何给定的分片链的验证节点 都需要追踪所有其他分片链 的状态（输出队列），随着分片链 总数的增加，这将需要更多的计算能力和网络带宽，从而限制了 系统的可扩展性。因此，无法直接从任何分片传递消息到其他每一个分片。因此，每个分片只与不同于自己的(w, s) 分片标识符（一个十六进制数字）的分片进行「连接」(参照 2.1.8)。

这样，所有的分片链将构成一个「超立方体」图，消息沿着这个超立方体的边缘移动。如果消息被发送到与当前的分片不同的分片，当前的分片标识符的十六进制数字被替换为目标分片的相应数字，并使用所得的标识符作为消息转发到最邻近的目标。（这不一

定是用于计算超立方体路由的下一跳的算法的最终版本。特别是，十六进制数 字可能会被r-位群组替换，其中r是一个可配置的参数，不一定等于四。）

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的区块的验证节点，必须收集和处理「邻近」分片链 的输出队列中的消息， 否则将失去他们的stake 。这样，任何消息都可以预期最终会达到其最终目的地；消息不能在过程中丢失或传递两次。

请注意，超立方体路由带来了一些额外的延迟和开销，因为必须通过几个中间的分片链 转发消息。然而，这些中间的分片链 的数量增长非常缓慢， 作为分片链 总数N 的对数log N（更确切地说，log16 N−1）。例如，如果N ≈ 250，最多 只有一次中间的转发；对于N ≈ 4000 的分片链， 最多有两个。四次中间的转发，我们可以支持多达一百万的分片链。我们认为为系统的基本无限的可扩展性支付的这个代价是非常小的。事实上，我们甚至不必支付这个费用。

2.4.20 即时超立方体路由:消息的“快速路径”

TON 区块链的一个新特性是它引入了一个「快速路径」的传递机制，从一个分片链转发消息到任何其他分片链，在大多数情况下都可以完全绕过2.4.19中的「慢速」超立方体路由，并在最终目的地分片链的下一个区块中传递消息。

该思路如下。在「慢速」的超立方体路由中，消息在超立方体的边缘上（在网络中）传递，但经过每个中间顶点都会被延迟（大约五秒钟），用来提交提交到相应的分片链 ，然后再继续传递到其他节点。

为了避免不必要的延迟，可以沿着超立方体的边缘转发消息和一个适当的Merkle 证明，而不用等待将其提交到中间的分片链。实际上，网络消息应该从原始分片的「任务组」的验证节点（参照 2.6.8） 转发到目的地分片的「任务组」的指定区块生产者（参照 2.6.9）；这样就可以直接完成，而不必沿着超立方体的边缘进行转发消息。当这条带有Merkle 证明的消息到达目的地分片链的验证节点（更确切地说，是collators；参照 2.6.5）时，他们可以立即将其提交到一个新的区块，而不用等待消息完成沿着「慢路径」的旅程。然后一个带有Merkle 证明的确认消息沿着超立方体的边缘发回，并且可以通过提交一个特殊的交易来停止消息沿着「慢路径」转发的过程。

请注意，这种「即时交付」机制并未取代在2.4.19中描述的「慢速」但不会丢失消息的机制。我们仍然需要有「慢路径」算法，因为如果使用「快速路径」算法，当验证节点丢失区块或简单地决定不将「快速路径」消息提交到区块链的新区块时，系统因无法感知而无法对验证节点做出惩罚。 （然而，验证节点有一定的动机只采用「快速路径」方法，因为他们将能够收集与消息相关的所有转发费用，这些费用尚未在「慢路径」转发消息过程中被消耗。）

因此，两种消息转发方法是并行运行的，只有当一个带有「快路径」机制的正确证明被提交到一个中间的分片链时，「慢路径」机制才会被中止。（事实上，我们可能会暂时或永久地禁用「快路径」机制，这时系统仍将继续正常工作，尽管速度会慢很多。）

2.4.21 从邻近的分片链的输出队列收集输入消息

当为分片链提议一个新的区块时，邻近（在2.4.19的路由超立方体意义上）的分片链的一些输出消息被包含在新的区块中作为「输入」消息，并立即被传递。关于这些邻近的分片链的输出消息必须以哪种顺序进行处理，有一定的规则。基本上， 一个「较旧」的消息（来自较旧的主链区块的分片链区块）必须在任何「较新」的消息之前传递；对于来自同一邻近分片链的消息，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中的消息已被相邻的分片链传递，则会通过一个特殊交易将该消息从输出队列中删除。

2.4.23 防止重复传递消息

为了防止从相邻的分片链的输出队列中重复传递消息，每个分片链（更确切地说，其中的每个账户链）将最近交付的消息的集合（或仅其哈希值）保存作为其状态的一部分。当观察到已交付的消息已经从其源相邻分片链（参见2.4.22）的输出队列中被删除时，会在最近交付的消息的集合中相应地删除这些消息。

2.4.24 转发用于其他分片链的消息

超立方体路由（参见2.4.19）有时出站消息并不是传递给目的收件人的分片链，而是转发给位于到目的地的超立方体路径上的相邻分片链。在这种情况下，消息将包括将入站消息移动到出站队列这样的内容。这在区块中将明确地反映为一个特殊的转发交易。本质上，这看起来就像该消息已被分片链内的某人接收，并且生成了一个具有相同内容的消息作为结果。

2.4.25 转发和保留消息的费用

转发交易实际上消耗了一些Gas（取决于被转发的消息的大小），代表此分片链的验

证节点将从被转发的消息的值中扣除了一笔Gas费用。此转发费用通常远小于当消息最终交付给其收件人时所确定的Gas费用，即使该消息由于超立方体路由而被转发了多次。此外，只要某个分片链的输出队列中保留有消息，它就是分片链的全局状态的一部分，因此特殊交易也可能收取因长时间保留全局数据的费用。

2.4.26 发送消息到主链

消息可以直接从任何分片链发送到主链，反之亦然。但是，发送消息到主链以及在主链中处理消息的Gas价格相当高，因此只有在真正需要时才会使用此功能。 例如， 由验证节点来存入他们质押的Ton coin。在某些情况下，可能会定义发送到主链的消息的最低存款（附加值），只有当接收方认为该消息是"有效"的时候才会退还。

消息不能自动通过主链路由。带有workchain_id = −1 的消息（其中−1 是表示主链的特殊workchain_id）不能转发给主链。

原则上，人们可以在主链内部创建一个消息转发智能合约，但使用它的价格将是非常高昂的。

2.4.27 同一个分片链中的帐户之间的消息

在某些情况下，某个分片链中的账户生成的消息，目标是同一分片链中的另一账户。例如，这可能发生在一个新的未分裂成多个分片链的工作链中，因为这时的负载还不高。

这样的消息可能会在分片链的输出队列中累积，然后，在后续的区块中作为入站消息进行处理。在大多数情况下，有可能在区块本身内就实现了这些消息的传递。

为了实现这一点，对包含在分片链区块中的所有交易进行部分排序， 并严格按照此部分顺序处理交易（每个交易都会向某个账户传递消息）。（译者注：参照2.4.17）

在这种情况下，消息主体不会被复制两次。相反，原始的交易和处理中的交易都引用消息的共享副本。

2.5 全局分片链状态：Bag of Cells哲学

现在我们准备描述TON 区块链或至少是基础工作链的分片链的全局状态。
我们从"高层级" 或"逻辑"描述开始，这个描述可以表述成全局状态是分片链状态（ShardChainState）代数类型的值。

2.5.1 分片链状态是账户链状态的集合

根据无限分片模型（参见2.1.2），任何分片链只是虚拟"账户链"的（临时）集合，每个账户链只包含一个账户。这意味着，本质上，全局分片链状态必须是一个hashmap

$$\text{ShardChainState} := (\text{Account} \to \text{AccountState}) \quad (23)$$

正如如我们之前讨论的分片(w, s)的状态（参见2.1.8），其中hashmap的所有索引account_id必须以前缀s开始。

实际上，我们可能希望将AccountState分成几个部分（例如，保持账户输出消息队列的独立，以简化相邻分片链的检查），并在分片链状态内部拥有几个hashmap (Account $\to$ $AccountStatePart_i$)。我们还可能向分片链S状态添加少量"全局"或"必要"的参数，（例如，属于此分片的所有账户的总余额，或所有输出队列中的消息总数）。

从"逻辑"（"高级"）的角度来看，(23) 是分片链全局状态的一个很好的初步描述。更进一步的话，可以使用TL方案（参见2.2.5）的帮助来进行代数类型AccountState和 分片链State的正式描述。

2.5.2 拆分和合并分片链状态

请注意，无限分片模型描述的分片链状态(23) 显示了当分片被分裂或合并时，该状态应如何被处理。实际上，通过使用hashmap的相关操作，这些状态的变换将变得非常简单。

2.5.3 账户链状态

账户链状态只是一个账户的状态， 由类型AccountState描述。通常它具有在2.3.20中列出的全部或某些字段，具体取决于使用的构造器。

2.5.4 全局工作链状态

与(23) 类似，我们可以使用相同的公式定义全局的工作链状态，但account_id 可以取任何值，不仅仅是属于一个分片的值。在这种情况下，也适用于2.5.1中所做的类似备注：我们可能想要将此哈希值映射分裂成几个哈希值映射，我们可能想要添加一些"必要"的参数，如总余额。

本质上，全局的工作链状态必须与分片链状态类型一致，因 为如果这个工作链的所

有现有分片链突然合并成一个， 我们会得到的就是这个分片链状态。

2.5.5 低层级角度：“bag of cells”

存在一个「低层级」描述关于账户链或分片链 状态，这与上述的「高阶」描述是互补的。此描述相当重要，因为它事实上是非常通用的，为储存、序列化和通过网络传递TON 区块链计划所有的数据（包括区块、分片链 states、smart-contract storage、Merkle proofs 等）提供了一个通用的表达。同时，通过实现这样的普遍「低阶」描述，我们就可以集中注意力仅考虑「高阶」描述。

回想一下，TVM 用一棵「TVM cells」树，或简称为「cells」（参见2.3.14和2.2.5）来表示任意代数类型的值（例如， (23)中的分片链State）。

每个这样的cell都由两个描述字节组成，一个是定义了某些标志和值$0 \le b \le 128$，一个是表示原始bytes 的数量$0 \le c \le 4$，这是指向其他cells的引用数量。然后紧跟的内容是b个bytes 和c 个cell 引用。（如果需要经常访问储存在cell 树中的所有资料的Merkle proofs，则应该使用$b + ch \approx 2(h + r)$ 的cells 来最小化平均Merkle proof 大小，其中$h = 32$ 是hash 在bytes中的大小，而$r \approx 4$ 是cell引用的字节大小。换句话说，一个cell 应该包含两个引用和一些字节，或包含一个引用和大约36字节，或包含完全没有引用但是有72字节。）

cell引用的确切格式取决于它的实现，以及cell是否位于RAM、磁盘、 网络封包（network package）、区块等。一个有用的抽象模型是想像所有cells 都存放在内容可寻址的的存储器中，cell 的地址等于其(sha256) hash。回想一下，cell 的(Merkle) hash 正是通过将其子cell的引用替换为它们（递归计算的）哈希值，然后执行哈希函数计算生成的字节串。

这样，如果我们使用cell的哈希值来引用cells（例如，在其他cells的内部描述中），系统稍微简化，且cell的哈希值开始与代表它的字节串的哈希值一致。

现在我们看到，VM可以表示的任何对象，包括全局分片链状态，都可以表示为一个「bag of cells」。即，一个cells的集合以及指向这个集合的「root」引用（例如，通过计算hash值）。请注意，重复的cells 从此描述中被删除了（「bag of cells」只是一个cells的集合，而不是多组含有cell的集合），所以bag of cells可以抽象成一颗树来表示，最后实际上可以用一个有向无环图(dag)来表示。

我们甚至可以在磁盘上使用B-tree 或B+-tree 来保存这个状态，包含所有相关的cells（也许还有一些附加数据，如子树高度或引用计数器），并由cell的哈希值进行索引。但是，这个想法的简单实现会导致一个智能合约的状态被分散在磁盘文件的不同部分，这是我们想要避免的。（更好的实现方法是，如果智能合约的状态很小，就将其保存为序列化

的字符串，如果很大， 则保存在另一个B-tree中；然后代表区块链状态的顶层结构将是一个B-tree，其叶子节点被允许包含对其他B-tree 的引用。）

现在我们将详细解释TON 区块链使用的几乎所有对象如何可以表示为「bag of cells」，从而展示这种方法的普遍性。

2.5.6 分片链区块表示为“bag of cells”

分片链 block 本身也可以用代数类型来描述，并储存为「bag of cells」。然后可以通过简单地连接表示「bag of cells」中每个cell 的字节串（以任意顺序）来获得区块的简单二进制表示。这种表示可以进一步改进和优化，例如，在区块的开头提供所有cells 的偏移量列表，并在可能的情况下用32位索引替换对其他cells 的hash引用。然而，我们应该认识到区块本质上也是一个「bag of cells」，所有其他技术细节只是次要的优化和实现。

2.5.7 更新对象表示为“bag of cells”

想像我们有一些以「bag of cells」来表示的旧版本的对象，我们想要表示同一对象的新版本，假设这个新版本与上一个版本只有很少的差别。一个简单的表示方法是将新状态表示为具有自己root的另一个「bag of cells」，并从中删除所有在旧版本中出现的cells，剩下的「bag of cells」基本上是对象的更新。每个拥有此对象的旧版本和更新的人都可以计算新版本，只需合并两个bag of cells，并删除旧的root（减少其引用次数，并在引用次数变为零时释放该cell）。

2.5.8 更新账户的状态

特别地，对账户的状态、分片链的全局状态或任何hashmap的更新都可使用在2.5.7中描述的想法进行表示。这意味着当我们接收到一个新的分片链区块（即「bag of cells」）时，我们不只是单独解释这个「bag of cells」，而是将其与代表分片链先前状态的「bag of cells 」结合。从这个意义上说，每个区块可能都「包含」区块链的整体状态。

2.5.9 更新区块

回忆一下，区块本身就是一个「bag of cells」， 所以，如果需要编辑区块，则可以类似地将「block update」定义为「bag of cells」，并在存在该 区块 的先前版本的「bag of cells」的情境下进行解释。这大致上是在2.1.17中讨论的「垂直区块链」背后的想法。

2.5.10 默克尔证明（Merkle proof）表示为"bag of cells"

注意，一个通用的Merkle证明: 例如，从一个已知的值Hash(x) = h 开始，声称x[i] = y（参见2.3.10和2.3.15），这其实也可以表示为「bag of cells」。具体来说，只需要提供一组cells子集， 对应从x : Hashmap(n, X) 的根到其所需的具有索引 i : 2^n和值y : X 的叶子的路径。在此证明中，不位于此路径上的这些cells 的子项的引用将保持「未解决」，由cell哈希值表示。还可以同时提供，例如，x[i] = y 和x[i'] = y' 的Merkle 证明，通过在「bag of cells」中包括位于从x 的根，到对应于索引 i 和i'的叶子的两条路径的cells的集合来提供证明。

2.5.11 Merkle 证明作为全节点的查询响应

实质上， 拥有分片链（或账户链）状态完整副本的全节点，可以在被轻节点（例如，运行TON 区块链 客户端轻版本的网络节点）请求时提供Merkle 证明，使接收者能够仅使用此Merkle 证明中提供的cells执行一些简单的查询，而不需要额外的帮助。轻节点可以将其查询以序列化格式发送给全节点，并接收正确的答案和Merkle 证明或仅仅是Merkle 证明， 因为请求者应该能够仅使用Merkle 证明中包含的cells来计算答案。这个Merkle 证明将仅由一个「bag of cells」组成，只包含属于分片链状态的那些cells，在执行轻节点的查询时由全节点给出。此方法尤其可用于执行智能合约的「get queries」的相关方法（参见5.3.12）。

2.5.12 基于默克尔证明进行状态更新

回想一下（参见2.5.7），我们可以透过一个更新来描述从旧值x : X 到新值x' : X 的状态变化，这只是一个「bag of cells」，其中包含那些位于表示新值x'的子树中的cells，但不包含位于表示旧值x的子树中的cells，因为假设接收者已经拥有旧值x 及其所有的cells副本。

然而，如果接收者并没有x 的完整复本，而只知道其（Merkle）hash h = Hash(x)，它将无法检查更新的有效性（即更新中的所有「悬空」cell 引用确实指向存在于x 的树中

的cells）。我们希望能够拥有一种可验证性的更新，并加上对旧状态中所有引用cells存在的Merkle证明。这样， 就算只知道h = Hash(x) 的任何人都能够检查更新的有效性，并自行计算新的h′ = Hash(x′)。

因为我们的Merkle证明本身就是「bags of cells」（参见2.5.10），可以将这样的「增强更新」构造为一个「bag of cells」，其中包含x 的旧根值、其某些子孙节点以及从x的根到它们的路径，以及x′的新根和其所有不属于x 的子孙节点。

2.5.13 在分片链区块中更新账户状态

特别地，在分片链区块中的账户状态更新应按照2.5.12中讨论的方式进行更新。否则，某人可能提交一个包含无效状态更新的区块，这个区块引用旧状态中不存在的cell；证明此区块的无效性将是困难的（挑战者如何证明cell不是先前状态的一部分？）。

现在，如果包含在block 中的所有状态更新都是「增强更新」的，它们的有效性可以轻松检查，并且其无效性也可以轻松显示为违反（通用）Merkle 哈希值的递归定义特性。

2.5.14 所有的东西都是一个「bag of cells」的设计哲学

前面的讨论表明， 我们在TON 区块链或网络中需要储存或传输的所有内容都可以表示为「bag of cells」。这是TON 区块链设计哲学的重要部分。一旦解释了「bag of cells」方法并定义了一些「bags of cells」的「低阶」序列化，就可以在抽象代数数据类型的高层次上定义所有内容（如区块格式、分片链和账户状态等）。

「一切都是bag of cells」哲学的统一效果大大简化了看似不相关的服务的实现；有关涉及支付通道的示例，请参见6.1.9。

2.5.15 Ton区块链的区块头

通常，区块链 中的区块会从一个小型的header开始，其中包含前一个区块的hash、创建时间、区块中所有交易的树的Merkle hash等。然后，区块的hash 被定义为这个小型区块头的hash。因为区块头最终取决于区块中包含的所有数据，所以无法在不改变其hash的情况下修改区块。

TON 区块链的区块使用的"bag of cells"方法，没有指定区块header。相反，区块hash 被定义为区块的根cell 的(Merkle) hash。 因此，区块的顶部（根）cell 可能被视为此区块的小型"header"。

然而，根cell可能不包含所有的数据，虽然这样的数据通常期望从这样的header 中获得。本质上，人们希望header包含区块数据类型中定义的某些字段。通常， 这些字段将包含在几个cell 中，包括根cell。这些cell 一起构成了有关字段值的"Merkle proof" 。可能有人会坚持在区块中的任何其他cell之前， 在一开始就包含这些"header cells"。然后，只需要下载区块序列化的前几个字节，就可以获得所有的"header cells" ，并获得所有期望的字段内容。

2.6 创建和验证新区块

TON 区块链 最终由分片链 和主链区块组成。为了系统可以顺利且正确地运作，必须创建、验证这些区块，并通过网络将其传播到所有相关方。

2.6.1 验证节点

新的区块 由特定的节点创建和验证，这些节点被称为验证节点。实质 上，任何希望成为验证节点 的节点都可以成为验证节点，前提是它可以在主链中存入足够多的抵押金（以TON 币，即Toncoin； 参考附录 A）。 验证节点如果正常工作可以获得一些「奖励」，这些奖励来自提供到最新区块的所有交易（消息）的储存和Gas费用，以及一些新铸造的币，这反映了社区对验证节点保持Ton区块链正常运作的「感谢」。这笔收入按照所有参与验证节点的抵押金比例分配。

然而，成为验证节点是一项重大的责任。如果验证节点签署了一个无效的区块，它可能会失去部分或全部的抵押金，并且可能会暂时或永久地被排除在验证节点之外。如果验证节点不参与创建区块，它不会收到与该区块 相关的奖励部分。如果验证节点长时间不创建新的区块，它可能会失去部分的抵押金，并被暂停或永久排除在验证节点之外。

这一切都意味着验证节点 不是轻而易举地获得金钱。事实上，它必须追踪所有或某些分片链的状态（每个验证节点 负责验证和创建某一子集分片链中的新区块），执行这些分片链中的智能合约所请求的所有计算，接收其他分片链的更新等等。这项活动需要大量的磁盘空间、计算能力和网络带宽。

2.6.2 验证节点代替矿工

请记住，TON 区块链 使用的是Proof-of-Stake方法，而不是Bitcoin、当前版本的Ethereum 和大多数其他加密货币采用的Proof-of-Work 方法。这意味着人们不能通过提供某些工作证明（计算大量其他无用的hashes）来「挖掘」新的区块，并因此获得一些新

的币。相反，人们必须成为验证节点，并花费自己的计算资源来储存和处理TON 区块链的请求和数据。简而言之，要挖新币，必须成为验证节点。就这一点而言，验证节点就是新的矿工。

但是，除了成为验证节点 之外，还有一些其他方式可以赚取币。

2.6.3 提名者（Nominators） 和矿池（mining pools）

要成为验证节点，通常需要购买和安装几台高性能的服务器，并为它们提供良好的互联网连接。这不像当前挖掘比特币所需的ASIC 设备那么昂贵。但你绝对不能在家用电脑上挖新的TON 币，更不用说智能手机了。在比特币、以太坊和其他Proof-of-Work 加密货币挖掘社区中，有一个名为矿池（mining pools）的概念，其中许多节点因计算能力不足而无法自行挖掘新的区块，所以他们合并力量，并在之后分享奖励。

Proof-of-Stake 世界中的相应概念是提名者（nominator）。实质上，这是一个将其资金借给验证节点以增加节点的抵押金；然后验证节点将其奖励的相应部分（或之前同意的一部分，例如50%）分配给nominator。

通过这种方式，nominator 也可以参与「挖掘」并获得与其存款金额成正比的一些奖励。它只获得验证节点奖励的一部分，因为它只提供了「资本」，但不需要购买计算能力、储存和网络带宽。

但是，如果验证节点 因无效行为而失去其抵押金，nominator 也会失去其部分抵押金。在这种意义上，nominator将分担风险。它必须明智地选择其nominated 验证节点，否则可能会损失资金。在这种意义上，nominator 进行加权决策并使用其资金「投票」支持某些验证节点。另一方面，这种提名或借贷系统使人们能够成为验证节点，而无需首先投入大量金钱购买Toncoin（TON 币）。换句话说，它防止持有大量Toncoin 的人垄断验证节点 的供应。

2.6.4 渔夫(Fisherman):通过指出别人的错误来获得金钱

另一种不成为验证节点而获得一些奖励的方式是成为一名渔夫（fisherman）。实质上，任何节点都可以通过在主链中存入少量资金成为fisherman。然后，它可以使用特殊的主链交易，发布某些由验证节点之前签名和发布的（通常是分片链） 区块的（Merkle）无效性证明。如果其他验证节点同意这个无效性证明，则违规的 验证节点 将被收到惩罚（失去其抵押金的一部分），fisherman 则获得一些奖励（从违规的验证节点中没收一部分币）。之后，如2.1.17中所述，必须更正无效的（分片链） 区块。更正无效的主链 区块， 可能需要在先前提交的主链区块之上创建「垂直」区块（参见2.1.17）；而无需创建主链 的分叉。

通常，一个fisherman 需要成为至少某些分片链的全节点，并花费一些计算资源来运行一些智能合约的程序代码。虽然fisherman不需要像验证节点 那么多的计算能力，但我们认为，一个天生的fisherman 是一个准备处理新区块，但尚未被选为验证节点 的候选者（例如，由于未能存入足够大的抵押金） 。

2.6.5 核对者(Collator):通过给验证者提议新区块来获得奖励

另一种不成为验证节点 但可以获得一些奖励的方法是成为一个核对者（collator）。这是一个节点，它为验证节点准备并提议新的候选分片链区块，并使用从此分片链的状态和其他(通常是相邻的) 分片链中获得的数据进行核对整理和补充，并附带适当的 Merkle 证明。当需要从邻近的分片链转发一些消息时，这种核对和整理操作是必要的。然后，验证节点可以轻松检查所提议的候选区块的有效性，无需下载此或其他分片链的完整状态。

由于验证节点需要提交新的核对者提议的候选区块以获得一些奖励，因此有理由支付一部分奖励给愿意提供合适候选区块的核对者。这样，验证节点可以避免观察邻近分片链的状态，可以将其外包给核对者即可。 然而，我们预期在系统的初始部署阶段不会有单独指定的核对者，因为所有验证节点都将能够为自己充当核对者。

2.6.6 核对者和验证者：获取为用户在区块中包含用户交易的奖励

用户可以向一些核对者和验证者打开微支付（micropayment ）通道，并支付少量的币以换取在分片链中包含他们的交易。

2.6.7 选举全局验证者集合

每月一次（实际上是每2^{19}个主链 区块）选举全局验证节点集合。此集合在一个月前确定并被全局知悉。 要成为验证节点，节点必须将一些TON 币（Toncoin）转入主链， 然后将它们发送到特定的智能合约作为其节点的抵押金。与抵押金一起发送的另一个参数是 $L \geq 1$，这是此节点愿意接受的相对于最小可能值的最大验证负载。还有一个 L的全局上限（另一个可配置参数）L'，比如说是10。

然后，这个智能合约选举全局验证节点 集合，只需选择最大的建议抵押金的前T 个候选者并公布其身份。最初，验证节点 的总数是T = 100； 随着负载的增加，我们预期它将增长到1000。它是一个可配置参数（参见2.1.21）。

每个验证节点的实际抵押金如下计算：如果前T 个提议的抵押金是

$$s_1 \geq s_2 \geq \cdots \geq s_T,$$

那么第 i 个验证节点 的实际抵押金设定为

$$S_i' := \min(s_i, l_i * s_T)$$

这样，$S_i'/S_T' \leq l_i$,所以第 i 个验证节点不会获得超过$l_i \leq$ MAX_L 倍最弱验证节点的负载（因为负载最终与抵押金成正比）。

然后，当选的验证节点可以撤回他们未使用的抵押金部分，$s_i - S_i'$。不成功的验证节点候选人可以撤回他们所有的抵押金。

每个验证节点发布其公开签名key，这个key并不一定等于抵押金来源的帐户的公钥。（对于每次验证节点的选举，生成和使用新的密钥对是有意义的。）

验证节点的抵押金，直到他们被选举的时期结束，再加上一个月，才会被解除冻结。通过这样的措施防止新的争议出现（例如，发现一个由这些验证节点签名的无效区块）。之后，将返还抵押金给验证者，以及在此期间铸造给验证节点的币份额和处理交易的费用。

2.6.8 选举验证者任务组

整体的全局验证节点 集合（其中每个验证节点 都被视为具有与其股份相等的多重性身份- 否则验证节点可能会被诱使承担多个身份并在它们之间划分其股份），只用于验证新的主链 区块。 分片链 的区块只由特定选择的验证节点 子集验证， 这些验证节点是从在2.6.7中描述的选择的全局验证节点 集合中选择的。

为每个分片定义的这些验证节点「子集」或「任务组」，每小时轮换一次（实际上，每2^{10}个主链区块轮换一次），并且它们提前一小时就知道了，因此每个验证节点 都知道它将需要验证哪些分片，并可以为此做准备（例如，通过下载丢失的分片链 数据）。

用于为每个分片(w, s) 选择验证节点 任务组的算法是确定性伪随机的。它使用验证节点 嵌入到每个主链 区块中的伪随机数（通过使用阈值签名生成的共识生成）来创建一个随机种子，然后为每个验证节点计算示例Hash(code(w). code(s).validator_id.rand_seed). 然后按此hash 的值对验证节点 进行排序，并选择第一个验证节点，以便至少具有总验证节点 股份的20/T，并且由至少5个验证节点 组成。

这种选择过程可以由特殊的智能合约完成。在这种情况下，选择算法将可以轻松升级，无需通过2.1.21中提到的投票机制进行硬分叉。迄今为止提到的所有其他「常数」（例如2^{19}、2^{10}、T、20 和5）也都是可配置的参数。

2.6.9 在每个任务组上轮换优先级顺序

在分片任务组的成员上有一个特定的「优先顺序」，取决于先前主链区块的hash 和(分片链) 区块序列号。此顺序是通过生成并排序上述的某些hash 来确定的。

当需要生成新的分片链 区块时， 通常选择用于创建此区块的分片任务组验证节点是根据此轮换「优先顺序」的第一名。如果它未能创建该区块，第二或第三个验证节点可能会创建该区块。本质上，他们都可以提议他们的候选区块，但是由具有最高优先权的验证节点 提议的候选区块应该作为Byzantine Fault Tolerant (BFT) 共识协议的结果获胜。

2.6.10 分片链候选区块的传播

因为分片链 任务组的成员提前一小时就已知，它们的成员可以利用这段时间，使用TON Network 的一般机制(参照 4.2),建立一个专用的「分片验证节点 多播覆盖网络」(shard validators multicast overlay network)。当需要生成一个新的分片链区块时,通常在最近的主链 区块被传播后的一两秒钟，每个人都知道谁有最高的优先权生成下一个区块(参照 2.6.9)。这个验证节点 将创建一个新的核对后的候选区块，无论是自

己还是在核对者的帮助下(参照 2.6.5)。 验证节点 必须检查（验证）此候选区块（尤其是如果它是由某个核对者准备的）并用其（验证节点）私钥签名。然后，使用预先安排的多播覆盖网络，将候选区块传播到任务组的剩余成员(该任务组根据 4.2 中的解释创建自己的私有覆盖网络，然后使用 4.2.15 中描述的**流式**多播协议（streaming multicast protocol ）来传播候选区块)。

真正的 BFT 方式是使用拜占庭多播协议，例如 Honey Badger BFT 中使用的协议[11]： 通过一个(N，2N/3)-擦除码对候选区块进行编码，将结果数据的 1/ N 直接发送到组的每个成员，并期望他们将其属于他们那部分的数据直接多播到组的所有其他成员。

然而，一个更快且更简单的方法（参见4.2.15）是将候选区块分成一系列签名的一千字节区块（「chunks」），通过Reed–Solomon 或Fountain code（如RaptorQ code [9] [14]）扩充它们的序列，并开始传输chunks 到「多播网格」（即覆盖网络）中的邻近节点，期望他们将这些chunks 进一步传播。一旦验证节点 获得足够的chunks 来重建候选区块，它就会签署一个确认收据并通过其近节点将其传播到整个组。然后，它的邻近节点停止向它发送新的chunks，但可能会继续发送这些chunks 的（原始）签名，认为该节点可以通过应用Reed– Solomon 或Fountain code自行生成后续的chunks（拥有所有必要的数据），将它们与签名结合起来，并传播给还没有准备好的邻近节点。

如果在移除所有「坏」节点后「多播网格」（覆盖网络）仍保持连接（回想一下，允许最多三分之一的节点以拜占庭方式坏节点，即以任意恶意方式行为），则此算法将以最快的方式传播候选区块。不仅指定的高优先级区块创建者可以将其候选区块多播到整个组。

不仅具有最高优先级的区块创建者可以广播其候选区块到整个任务组。第二和第三优先级的验证者，也可以立即或在未能从最高优先级验证者接收块候选之后，开始多播它们的候选区块。 但是，通常只有最高优先权的候选区块将被所有（实际上，至少由三分之二的任务组）验证节点 签名并作为新的分片链 区块提交。

2.6.11 验证候选区块

一旦候选区块被验证节点接收，并通过原始验证节点的签名检查，接收验证节点 会检查此候选区块的有效性，执行其中的所有交易并确保其结果与所声称的一致。从其他区块链导入的所有消息都必须在核对数据中有适当的Merkle 证明，否则候选区块将被视为无效（并且，如果此证明被提交到主链，则可能会惩罚已经签署此候选区块的验证节点）。另一方面，如果候选区块被认定为有效，接收验证节点 会签署它并将其签名传播给组中的其他验证节点，可以通过网格多播网络「mesh multicast network」 或直接的网络消息进行传播。

我们想强调，验证节点在检查核对者提交候选区块的有效性时，不需要访问此分片链或相邻分片链的状态。（一个可能的例外是相邻分片链 的输出队列的状态，需要确保如2.4.21描述的那样的小子顺序，而在这种情况下，Merkle 证明的大小可能会变得过大。）这使得验证可以非常快速地进行（不需要磁盘访问），并减轻了验证节点的计算和储存负担（尤其是如果他们愿意接受外部和核对者的帮助来创建候选区块）。

2.6.12 下一个候选区块的选举

一旦候选区块 收集到任务组中拥有至少三分之二的总质押token的验证节点 的有效签名，它就有资格被提交为下一个分片链区块。通过运行一个BFT 协议来达成对所选候选区块 的共识（可能有多个提议），所有「好的」验证节点 都会优先选择该轮中优先级最高的候选区块。运行此协议的结果是，该区块 会被拥有至少三分之二的总质押token的验证节点的签名所确认。这些签名不仅证明了这些待处理的区块的有效性，还证明了它是由BFT 协议选出的。之后， 将区块（不用再附加核对数据）与这些签名组合，以确定的方式序列化，然后通过网络传播给所有相关方。

2.6.13 验证节点必须保留它们已经签名的区块

在验证节点成为任务组的成员期间，以及之后至少一小时（或者2^{10}区块之后），验证节点需要保留它们已签署和提交的区块。如果未能向其他验证节点提供签署的区块，可能会受到惩罚。

2.6.14 传播新分片链的区块头和签名给所有验证者

使用类似于为每个任务组创建的多播网格网络「multicast mesh network」，验证节点将新生成的分片链区块的区块头和签名传播给全局验证节点集合。

2.6.15 生成新主链区块

在所有（或几乎所有） 新的分片链区块生成之后，可以生成一个新的主链区块。这个程序基本上与分片链区块相同（参见2.6.12），不同之处在于所有验证节点（或至少三分之二的验证节点）都必须参与此过程。因为新的分片链区块的区块头和签名会传播给所有的验证节点， 所以每个分片链中最新的区块的hash必须被包含在新的主链区块中。一旦这些hash被提交到主链区块，外部观察者和其他分片链 可以认为新的分片链区块已经提交并且是不可变的（参见2.1.13）。

2.6.16 验证节点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， 所有的验证者都需要追踪主链的状态，而不仅依赖于核对的数据。这一点很重要， 因为任务组的验证节点的知识是从主链的状态中衍生出来的。

2.6.17 分片链区块的生成和传播是并行的

通常，每个验证节点 都是几个分片链 任务组的成员；验证节点作为分片连的成员的数量（基于验证节点 的负载）大约与验证节点质押的Ton coin是比例的。（译者注：即验证节点质押的Ton coin越多，可能会成为更多分片链任务组的成员，执行更多验证和生成分片链区块的工作）。这意味着验证节点会同时运行多个生成新的分片链区块的协议实例。

2.6.18 减轻区块被私自保留而不发布的风险

因为所有验证节点， 只有在只看到新区块的区块头和签名之后，才将这个新的分片链区块的hash 插入到主链中，因此存在一个很小的可能，即生成此区块的验证节点可能会试图避免发布这个新区块的全部内容。这将导致邻近分片链的验证节点无法创建新的区块，因为一旦它的hash 被提交到主链 中，他们必须至少知道新区块的输出消息队列。

为了减轻这种风险，新区块必须从其他一些验证节点（例如，邻近分片链的任务组的三分之二成员）收集签名，证明这些验证节点 确实拥有此区块的副本，并且愿意在需要时将它们发送给其他验证节点。只有在提供这些签名之后，新区块的hash 才可能包含在主链中。

2.6.19 主链区块比分片链区块更晚生成

主链区块大约每五秒生成一次，就像分片链区块一样。但是，虽然所有分片链中的新区块的生成基本上是同时进行的（通常由发布新的主链区块触发），但新的主链区块的生成被系统故意延迟，以预留时间让主链可以包含新生成的分片链区块的hashes。

2.6.20 慢验证节点会获得更低的奖励

如果一个验证节点的处理速度是"慢"的，它可能无法验证新的候选区块，并且无法参与收集提交新区块所需的三分之二的签名的这个过程。在这种情况下，它将收到较低份额的区块奖励。

这将激励验证节点优化他们的硬件、软件和网络连接，以便尽可能快地处理用户交易。

然而，如果验证节点在区块提交之前未签名，但其签名可能包含在接下来的一个或多个区块中，然后这部分奖励（根据已生成多少区块而呈指数下降---例如，如果验证节点延迟k个区块签名，则获得奖励为基础奖励*0.9^{k}）仍将给予此验证节点。

2.6.21 验证者节点的签名"深度"（ Depth）

通常，当验证节点签署一个区块时，该签名只证明了区块的相对有效性：只要这个和其他分片链的所有之前的区块都是有效的，那么这个区块就是有效的。 另外，验证节点不能因为在先前区块中提交了无效数据，就理所当然被认为会受到惩罚。

区块的验证节点的签名有一个称为"深度"（Depth）的整数参数。如果它是非零的，那就意味着验证节点也宣称了之前指定数量的区块的(相对）有效性。通过这种方式，"慢"

或"暂时离线"的验证节点 可以赶上并签署一些已经提交但未经他们签名的区块。然后区块奖励的一部分仍将给予他们（参见2.6.20）。

2.6.22 验证节点只负责签名区块的相对有效性

我们想再次强调，验证节点在分片链 区块B 上的签名只证明了该区块的相对有效性（或者如果签名有"深度"d，也可能是 d 个先前的区块的相对有效性，参见2.6.21；但这不太影响下面的讨论）。换句话说，验证节点 声称分片链 的下一个状态s ′是通过应用在2.2.6中描述的区块评估函数ev_block从先前的状态s 获得的：

$$s' = ev\_block(B)(s) \quad (24)$$

这样，如果原始状态s 被证明为"不正确"（例如，由于先前区块的无效性），那么签署了区块B 的验证节点不能被惩罚。 渔夫（fisherman）（参见2.6.4）只有在发现一个区块是相对无效时才能发起质疑。 PoS 系统作为一个整体努力使每个区块都是相对地有效的，而不是递归地（或绝对地）有效的。然而，注意到，如果区块链中的所有区块都是相对有效的，那么它们所有的和区块链作为一个整体都是绝对有效的；使用对区块链 的长度的数学归纳法可以轻易地证明这一语句。通过这种方式，可轻松验证的区块相对有效性的声明一起证明了整个区块链的更有保证的绝对有效性。

注意，通过签名一个区块B，验证节点 声称该区块给定原始状态 s 是有效的（即，(24)的结果不是值⊥，表示下一个状态不能被计算）。以这种方式， 验证节点必须执行在评估(24)期间访问的原始状态的cell的最小形式检查。

例如，想像一个情境，期望从提交到区块的交易中访问的账户的原始余额的cell 被发现有零个原始字节，而不是期望的8 或16个字节。然后，原始余额不能从cell 中简单地获得，并且在尝试处理该区块时会发生"未处理的异常"。在这种情况下，验证节点 不应该签署这样的区块，否则将受到惩罚。

2.6.23 签名主链区块

主链区块的情况略有不同：签署一个主链 区块，验证节点 不仅声明其相对有效性，而且还声明所有先前区块的相对有效性，直到这个验证节点承担其责任的第一个区块。

2.6.24 验证节点的总数

需要选举的验证节点的总数上限 T （参见2.6.7）， 在迄今为止描述的系统中，不能数量太多，例如，几百或一千， 因为所有验证节点 都预计参与 BFT共识协议来创建每个新的主链区块，并且尚不清楚这样的协议是否可以扩展到数千参与者。更重要的是， 主

链 区块必须收集至少三分之二的所有验证节点（按质押的Ton coin）的签名， 并且这些签名必须包含在新区块中（否则系统中的所有其他节点都没有理由信任新区块而不自己验证它）。如果必须在每个主链 区块中包括超过，例如，一千个验证节点 签名，这将意味着每个主链 区块中有更多的数据，所有全节点都要储存并通过网络传播，以及花费更多的处理能力来检查这些签名（在PoS 系统中， 全节点不需要自己验证区块， 但他们需要检查验证节点的签名）。

虽然将T 限制为一千个验证节点， 对于TON 区块链 的部署的第一阶段似乎足够了，但必须为未来的增长做准备，当分片链的总数变得如此之大，以至于几百个验证节点不足以处理所有的分片链。为此，我们引入了一个额外的可配置参数$T' \leq T$（原本等于T），并且只有前T'名被选举的验证节点（按质押的Ton coin）预计创建和签署新的主链区块。

2.6.25 系统的去中心化

有人可能会怀疑，像TON 区块链 这样的Proof-of-Stake 系统，依赖T ≈ 1000 的验证节点 来创建所有分片链 和主链 区块，是否会变得"太中心化"，与像Bitcoin 或Ethereum 这样的传统Proof-of-Work 区块 链相反，其中每个人（原则上） 都可能开采一个新区块，没有矿工总数的明确上限。

但是，像Bitcoin 和Ethereum 这样的受欢迎的Proof-of-Work 区块链（译者注：目前Etherem也已经转成POS，本文写于Eth2.0之前。）， 目前需要大量的计算能力（高"hash rates"）以成功的概率开采新区块。因此，新区块的开采趋于集中在几个大玩家手中，他们投资大量资金建立充满为开采优化的定制硬件的数据中心；以及几个大型的开采池手中，它们集中并协调了大量的无法自己提供足够"hash rate"的人的计算资源。

因此，到2017 年，超过75％的新Ethereum 或Bitcoin 区块由少于十个矿工产生。实际上，两个最大的Ethereum 矿池共同生产了超过一半的所有新区块！显然，这样的系统比依赖T ≈ 1000 节点生产新区块的系统更为集中。

人们还可能注意到，成为TON区块链验证节点 所需的投资包括：即购买硬件（例如，几个高性能服务器）和质押的Ton coin（如有必要，可以通过nominator pool（类似POW矿池的POS矿池） 轻松收集；参见2.6.3） 比成为成功的独立Bitcoin 或Ethereum 矿工所需的要少。实际上， 2.6.7的参数 L 将迫使nominator 不加入最大的"开采池"（即，积累了最大权益的验证节点），而是寻找目前正在接受nominator 资金的较小的验证节点，甚至创建新的验证节点 ，因为这将使nominator获得更高的挖矿奖励，因为验证节点的矿池包含的nominator越小，每个nominator分配到的挖矿奖励就会约多。通过这种方式，TON Proof-of-Stake 系统实际上鼓励去中心化（创建和使用更多的验证节点）

并惩罚中心化。

2.6.26 区块的相对可靠性

区块的(相对)可靠性简单地说是依赖于已签署此区块的所有验证节点 的总质押Ton coin。换句话说，如果这个区块被证明是无效的，那某些验证节点会失去质押的Ton coin。如果区块的交易转移的资金价值低于为保持这些区块的可靠性而质押的资金的价值，我们可以认为这些交易足够安全。从这个意义上说，相对可靠性是外部观察者可以对特定区块的信任度的衡量指标。

注意，我们谈论的是区块的相对可靠性，因为它保证该区块是有效的前提是，前一个区块和所有其他被引用的分片链区块都是有效的（参见2.6.22）。

一个区块的相对可靠性在提交后可能会增加 例如，当迟来的验证者的签名被添加时（参见2.6.21）。另一方面，如果其中一个验证节点 因为对区块的不当行为而失去部分或全部质押Ton coin时，区块的相对可靠性可能减少。

2.6.27 提高区块链可靠性

提供一定的激励，使验证节点提高区块相对可靠性是很重要的。其中一种方法是为验证节点 分配小量的奖励，鼓励验证节点为其他分片链的区块上添加签名。即使是"即将成为"的验证节点，他们已存入的质押不足以进入按质押排名前T 的验证节点，也因质押不足以被包括在全局验证节点集合中（参见2.6.7），也可能参与这项活动（如果他们同意在失去选举后，继续保留冻结质押的资金，而不是撤回质押的资金）。这样的验证节点 可能同时充当渔夫（fisherman）（参见2.6.4）：如果他们必须检查某些区块的有效性，他们也可以选择报告无效的区块并收集相关奖励。

2.6.28 区块的递归可靠性

人们也可以定义一个区块的递归可靠性（recursive reliability）， 这是其相对可靠性与其引用的所有区块（即，主链区块、先前的分片链 区块和一些相邻分片链的区块）的递归可靠性之间的最小值。换句话说，如果该区块被证明是无效的，无论是因为它本身无效，还是因为它所依赖的某个区块无效，至少有人会因此亏钱（即被没收一定数量质押的Ton coin）。如果人们不确定是否信任区块中的特定交易，人们应该计算这个区块的递归可靠性，而不仅仅是相关靠性。

当计算递归可靠性时，没有必要回溯太远，因为如果我们回溯太远， 我们会看到已经验证节点虽然已经签名，但已解冻并取回质押资金的区块。无论如何，我们不允许验证节点 自动重新考虑那些旧的区块（即，创建超过两个月的区块，如果使用当前的可配置

参数的值），并从中创建分叉或使用"垂直区块链"（参见2.1.17）修正它们，即使它们被证明是无效的。我们假设两个月的时期，已经提供了充足的机会来检查和报告任何无效的区块，因此如果在这段时期内没有人质疑某个区块，那么代表这个区块是正确的，不太可能被质疑。

2.6.29 Proof-of-Stake算法对于轻节点的重要性

TON 区块链 采用Proof- of-Stake 方法的一个重要结果是，TON 区块链 的轻节点（运行轻型客户端软件）不需要为了能够自行检查全节点提供给轻节点的Merkle 证明，而下载所有分片链或主链区块的「区块头」。

实际上，由于最近的分片链区块hash 已包含在主链 区块中，全节点可以轻松提供一个Merkle 证明，从已知的主链 区块的hash 开始，说明给定的分片链 区块是有效的。接下来，轻节点只需要知道主链 的第一个区块（区块里包含已经被宣告的第一组验证节点），这个区块（或至少其hash）可能被内置到客户端软件中，并且每个月后只需一个（包括新当选的验证节点 集合）主链区块即可，因为这个区块已经由上一组验证节点共同签名。从那时起，它可以通过获得最近的几个主链 区块，或至少他们的区块头和验证节点签名，来检查全节点提供的Merkle 证明是否有效。

2.7 分裂和合并分片链

TON 区块链 最具特色和独特的功能之一是，当负载过高时，它能够自动将分片链分裂为两部分，并在负载下降时将它们合并回来（参见2.1.10）。由于其独特性和对整个项目可扩展性的重要性，我们必须详细讨论它。

2.7.1 分片配置

请回忆，在任何给定的时间点，每个工作链w 都被分割成一个或多个分片链(w, s)（参见2.1.8）。这些分片链可以由一棵二进制树的叶子表示，其根为(w, ∅)，且每个非叶子节点(w, s) 都有子节点(w, s.0) 和(w, s.1)。这样，属于工作链w 的每个账户都被分配到确切的一个分片，而知道当前的分片链 配置的每个人，都可以确定这个包含账户account_id的分片(w, s)：因为这是唯一一个带有account_id前缀的二进制字符串的分片。

所谓分片配置，即这个分片binary tree，或给定w 的所有活跃(w, s) 的集合（对应于分片binary tree 的叶子），这个分片设置是主链 状态的一部分， 且对于跟踪主链 的每个人都是有效的。（实际上，分片配置完全由最后的主链 区块确定；这简化了获取分片

配置的访问。）

2.7.2 最近的分片配置和状态

回想一下，最近的分片链 区块的hashes 被包含在每个主链 区块中。这些hashes 被组织成一个分片binary tree（实际上，是每个工作链 的一系列树）。这样，每个主链 区块都包含最近的分片配置。

2.7.3 宣布并执行分片配置中的更改

分片配置可以通过两种方式更改：要么将分片(w, s) 粉裂为两个分片 (w, s.0) 和(w, s.1)，要么将两个分片 (w, s.0) 和(w, s.1) 合并为一个分片(w, s)。

这些分裂/合并操作会在许多区块中被事先宣布（例如，提前在2^6个区块中宣告；这是一个可配置的参数），首先在相应的分片链 区块的「区块头」中宣告，然后在引用这些分片链 区块的主链区块中宣告。这个提前宣告是为了让所有相关方为计划的变更做好准备（例如，建立一个覆盖多播网络（overlay multicast network） 来分发新创建的分片链的新区块，如4.2所述）。然后，首先更改提交到分片链 区块的（区块头），然后传播到主链区块。这样， 主链区块不仅定义了在其创建之前的最近分片配置，而且还定义了下一个即将更改的分片配置。

2.7.4 新分片链的验证者任务组

回忆一下，每个分片，即每个 分片链，通常都被分配一个验证节点 的子集合（一个验证节点 任务组），用于在相应的分片链 中创建和验证新区块（参见2.6.8）。这些任务组成员每隔一段时间（大约一小时）被选择，并且会被提前通知结果（也大约是一小时），当然在此期间是任务组成员是不变的。（只有当某些验证节点 因签署无效的区块而被临时或永久禁止，这样这些验证节点将自动从所有任务组中被排除。）

然后，实际的分片配置可能会在此期间因为分裂/合并操作而发生变化。必须为新创建的分片分配任务组。这是如此完成的：

注意，任何活跃的分片(w, s) 要么是某个唯一确定的原始分片(w, s′) 的后代，意味着s ′是s 的前缀，要么它将是原始分片(w, s′)的子树的根，其中s 将是每个s ′的前缀。在第一种情况下，我们简单地将原始分片(w, s′) 的任务组 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′) 的任务组的集合，这些分片是分片tree中的(w, s) 的后代。

这样，每个活跃的分片(w, s) 都被分配了一个明确定义的验证节点 子集（任务组）。当一个分片分裂时，两个子节点都继承了原始分片的整个任务组。当两个分片被合并时，

它们的任务组也被合并。

任何追踪主链状态的人都可以为每个活跃分片计算验证节点任务组。

2.7.5 在原任务组的工作期间限制分裂/合并的操作

最终，新的分片配置将被考虑在在内，并且新的专用验证节点 子集（任务组）将自动分配给每个分片。在此之前，必须对分裂/合并操作施加某种限制；否则，如果原始分片迅速分裂成2^k个新的分片，则原始任务组可能最终需要同时验证2^k分片链，如果k是一个很大值，将造成很大的负担。

为了解决这个问题，我们可以通过限制活跃分片配置与原始分片配置（用于选择当前负责的验证节点任务组的配置）之间的距离来实现。例如，如果 s' 是 s 的前身（即s'是二进制字符串s的前缀），则可能要求分片树中从活跃分片(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)，则不得超过 2。否则，不允许执行分裂或合并操作。

简单说，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（本质是一定数量的区块）内重新配置。

2.7.6 决定执行分裂操作的必要条件

分片链 的分裂 操作是由某些正式条件触发的（例如，如果连续64 个区块的分片链区块至少有90% 是满的（译者注：即每个区块包含非常多交易，使这个64个区块的90%的区块的区块大小都已经达到最大的区块大小））。这些条件由分片链 任务组进行监控。如果它们得到满足，首先在新的分片链 区块的区块头中包含一个「分裂准备」 标志（并传播到引用这个分片链 区块的主链区块）。然后，在几个区块之后，分片链 区块的区块头中包含「分裂提交」标志（并传播到下一个主链区块）。

2.7.7 执行分裂操作

在分片链 (w, s) 的区块B 中包含「分裂提交」标志后，该分片链中将不会有后续的

区块B′。相反， 将创建分片链(w, s.0) 和(w, s.1) 的两个区块B'_0和B'_1，分别引用区块B 作为它们的前一个区块（并且它们都将通过区块头中的标志表示这个分片刚刚被分裂过）。下一个主链区块将包含新分片链区块B'_0和B'_1的哈希值; 且不允许包含分片（w，s)的新块B' 的哈希，因为“分裂提交”标志已经提交到先前的主链区块中。

请注意，两个新的分片链将由与旧的分片链相同的验证节点任务组进行验证，所以它们将自动获得其状态的副本。从无限分片范式的角度来看，状态分裂操作本身相对简单（参见2.5.2）。

2.7.8 决定执行合并操作的必要条件

分片链 的合并操作也需要由某些正式条件触发的(例如，如果连续64个区块的两个兄弟分片链的区块大小总和不超过最大区块大小的60%。（译者注：这样的条件证明两个分片链比较空闲，处理的区块包含的交易数量比较少，因此无需使用两个分片进行处理，合并成一个分片处理即可。）这些正式条件还应考虑这些区块所消耗的总gas，并将其与当前的区块gas 限制进行比较，否则由于有一些计算密集型的交易阻止了更多交易的被包含进区块，区块的大小可能会突然变小。

这些条件由 两个兄弟分片(w, s.0) 和(w, s.1) 的验证节点任务组监控。请注意，兄弟节点在超立方体路由中必然是邻居节点(参考2.4.19)，因此任何分片的任务组 的验证节点都将在某种程度上监控兄弟分片。

当满足这些条件时，验证节点任务子组的任何一个节点可以通过发送特殊消息，提议两个分片合并。然后，它们组合成一个临时的「合并任务组」，具有合并分片的成员资格，能够运行BFT 共识算法，并在必要时传播区块更新和候选区块。

如果它们就合并的必要性和相关情况达成共识，「准备合并」标志将提交到每个分片链的一些区块的区块头，并附带至少三分之二的兄弟任务组 的验证节点 的签名（并被传播到下一个主链区块，以便每个节点都可以为即将到来的新的分片配置做好准备）。然后然后，他们会继续为一些预定义数量的区块创建单独的分片链区块。

2.7.9 执行合并操作

当来自两个原始任务组的验证节点 准备成为合并分片链的验证节点时(这可能涉及从兄弟分片链 的状态转移和状态合并操作)，它们在其分片链的区块的头部提交一个「合并

提交」标志(这一标志会传播到下一个主链区块)，并停止在单独的分片链中创建新的区块(一旦出现合并提交 标志，在单独的分片链中创建区块是被禁止的)。相反，一个合并后的分片链 区块将被创建(由两个原始任务组的联合创建)，并且区块的「区块头」引用了它的两个「之前的区块」（译者注：即第一合并后的分片区块的区块头，会引用合并前的两个分片的区块）。这反映在下一个主链区块中，该区块将包含已合并分片链 的新创建区块的hash。之后，已合并的任务组继续在已合并的分片链 中创建区块。

2.8 区块链项目的分类

我们将通过将TON 区块链与现有和拟议的区块链项目进行比较，来结束我们对TON区块链的简短讨论。但在此之前，我们必须引入一个足够通用的区块链项目分类。基于此分类的特定区块链项目的比较，将被推迟到2.9。

2.8.1 区块链项目的分类

作为第一步，我们提出了一些用于区块链（即， 对于区块链项目）的分类标准。任何这种分类都是有点不完整和表面的，因为它必须忽略正在考虑的项目的一些最具体和独特的特点。但是我们认为这是必要的，至少提供了区块链项目的粗略分类的第一步。

我们考虑的分类标准列表如下所示：

- 单一区块链与多区块链架构(参见 2.8.2)
- 共识算法：Proof-of-Stake 与Proof-of-Work (参见2.8.3)
- 对于Proof-of-Stake 系统，使用的区块生成、验证和共识算法（两个主要选项是DPOS 与BFT；参见2.8.4)
- 对「任意的」(图灵完备Turing-complete) 智能合约的支持(参见2.8.6)

多区块链系统有额外的分类标准(参见2.8.7):

- 成员区块链的类型和规则：同构区块链、异构区块链(参见2.8.8)，混合型区块链(参见2.8.9)。联盟链(参见 2.8.10)

- 有无主链，内部链或外部链(参见2.8.11)·
- 是否原生支持分片(参见2.8.12)。静态或动态分片（参见2.8.13)
- 成员区块链之间的互动：松散耦合型系统和紧密耦合型系统(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的区块链数量。最古老和最简单的区块链项目由一个单一区块链组成（简称「单链项目」）；更复杂的项目使用（或更确切地说，计划使用）多个区块链（简称「多链项目」） 。

单链项目通常更简单且经过更好的测试；它们经受住了时间的考验。它们的主要缺点是低性能，或者至少是交易吞吐量较低，对于通用系统来说，这一数量在十(例如Bitcoin)到不到一百的每秒交易数(例如 Ethereum，译者注：如前所述，目前以太坊已升级成POS，可支持更高的每秒交易数)。一些专用系统（如Bitshares）能够处理每秒数万的专用交易，代理是区块链状态必须进行改造以适合存储在内存中，并将处理限制于一个预定义的特殊交易集，然后由像C++ 这样的语言编写高度优化的程序代码执行（这里没有虚拟机（VMs）执行）。

多链项目则提供了每个人都渴望的可扩展性。他们可能支持更大的总状态和更多的每秒交易数，但代价是使项目变得更为复杂，其实施更具挑战性。因此，已经运行的多链项目很少，但大多数计划开发的项目都是多链的。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（POW）vs.权益证明（POS）

另一个重要的区别是用于创建和传播新区块、检查其有效性，以及在出现多个分支时选择其中之一的算法和协议。

两种最常见的范畴是Proof-of-Work (PoW)和Proof-of-Stake (PoS)。工作量证明（POW）方法通常允许任何节点创建（"挖掘"）一个新区块（并获得与挖掘区块相关的一些奖励），前提是它有幸在其他竞争者成功之前解决一个在其他情况下根本无用的计算问题（通常涉及计算大量的hashes）。在出现分支的情况下（例如，如果两个节点发布两个有效但不同的区块，并且这两个区块都来自相同的上一个区块），则最长的分支会胜出。这样，区块链的不变性保证是基于生成区块链所花费的工作量（计算资源）：任何希望创建此区块链的分支的人都需要重新做这些工作， 以创建已提交区块的替代版本。为此，一个人需要控制超过50% 的创建新区块所花费的总计算能力，否则替代分支变得最长的机会会指数性地降低。

权益证明方法（POS）基于一些特殊节点（验证节点）所做的大量质押（使用加密货币进行质押），以宣传它们已经 检查了一些区块并发现它们是正确的。 验证节点 签署

区块，并因此收到一些小奖励；但是，如果一个验证节点 被发现签署了一个不正确的区块，并且提供了这方面的证据，则其全部或部分质押将被没收。这样，区块链的有效性和不变性保证是由验证节点对区块链有效性的总质押给出的。

从这个角度看，权益证明更为自然，因为它激励验证节点（它们取代了PoW 矿工）执行有用的计算（需要检查或创建新区块，尤其是执行区块中列出的所有交易），而不是计算其他无用的 hashes。这样，验证节点会购买更适合处理用户交易的硬件，以获得与这些交易相关的奖励，从整个系统的角度看，这似乎是一项相当有用的投资。

然而，权益证明系统在实施上有些挑战，因为必须为许多罕见但可能的情况提供支持。例如，一些恶意的验证节点 可能密谋破坏系统以获取利益（例如，通过改变自己的加密货币余额）。这导致了一些非常重要的博弈论问题。

简而言之，权益证明更为自然且更有前景，尤其是对于多区块链项目（因为如果有许多区块链，工作量证明将需要过多的计算资源），但必须更加小心地考虑和实施。大多数目前运行的区块链项目，尤其是最古老的项目（如Bitcoin 和至少是原始的Ethereum），使用工作量证明。

2.8.4 Proof-of-Stake的不同算法： DPOS vs. BFT.

虽然Proof-of-Work 算法彼此非常相似，主要差异在于必须计算以挖掘新区块的hash函数，但Proof-of-Stake 算法有更多的可能性。它们值得作为一个子分类。

本质上，人们必须回答关于Proof-of-Stake 算法的以下问题：

- 谁可以产生（“挖掘”）一个新区块？任何全节点都可以，还是只是验证节点 的（相对地）一个子集的成员？ (大多数PoS 系统要求新区块由数个指定的验证节点 生成并签名。)
- 验证节点 是否通过他们的签名保证区块的有效性，还是所有全节点都期望自己验证所有区块？ (可扩展的PoS 系统必须依赖验证节点的签名，而不是要求所有节点验证所有区块链的所有区块。)
- 是否有一个预先知道的指定生产者来生成下一个区块链区块，这样其他人就不能代替它产生那个区块？
- 新建的区块最初只由一个验证节点（其生产者）签署，还是它必须从一开始就收集大多数验证节点 的签名？

虽然似乎根据这些问题的答案有2^4可能的PoS 算法类别，但在实践中，区别主要归结为两种主要的PoS 方法。事实上，大多数现代的PoS 算法，旨在用于可扩展的多链系统，在前两个问题上的答案是相同的：只有验证节点 可以产生新区块，并且他们保证区

块的有效性，而不要求所有全节点自己检查所有区块的有效性。

至于最后两个问题，它们的答案被证明是高度相关的，基本上只留下了两个基本选项：

- Delegated Proof-of-Stake (DPOS)：每个区块都有一个众所周知的指定生产者；其他人不能生产该区块；新区块最初只由其生成的验证节点 签署。
- Byzantine Fault Tolerant (BFT) PoS 算法：有一个已知的验证节点 子集，其中任何一个都可以提议一个新区块；在多个提议的候选区块中选择下一个区块的选择， 必须在被发布到其他节点之前，由大多数验证节点验证并签名，这是通过Byzantine Fault Tolerant 共识协议的版本来实现的。

2.8.5 DPOS 和 BFT PoS 的对比

BFT 方法的优势是新产生的区块从一开始就有大多数验证节点 的签名证明其有效性。另一个优点是，如果大多数验证节点 正确执行BFT 共识协议，则根本不会出现分叉。但另一方面，BFT 算法往往相当复杂，并且需要更多的时间让 验证节点 子集达成共识。因此，区块不能太频繁地生成。这就是为什么我们预期TON 区块链（从这个分类的角度看是一个BFT 项目）每五秒只生成一个区块。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）， 但如果验证节点 分散在全球各地， 则不会再减少这个时间。

DPOS 算法的优势是相当简单和直接。由于它依赖于预先知道的指定区块生成者，所以可以非常频繁地生成新区块，例如，每两秒一次，或者甚至每秒一次。（有些人甚至声称DPOS 区块生成时间为半秒，但如果验证节点分散在几个地球的不同大陆上，这似乎不太现实。）

然而，DPOS 要求所有节点- 或至少所有验证节点，验证收到的所有区块，因为生成并签署新区块的验证节点 不仅确认了此区块的相对有效性，还确认了它引用的前一个区块的有效性，以及在链中更远的所有区块（也许可以追溯到当前验证节点子集的责任期开始的第一个区块）。当前验证节点 子集上有一个预定的顺序，因此每个区块都有一个指定的生产者（即，预期生成该区块的验证节点）；这些指定的生产者按轮换的方式进行替换。这样，一个区块首先只由其生成的验证节点签名；然后，当挖掘下一个区块时，生产者选择引用此区块而不是它的前一个区块（否则它的区块将位于较短的链中，这可能会在未来失去"最长分叉"的地位），下一个区块的签名本质上也是对前一个区块的附加签名。这样，一个新区块逐渐收集更多验证节点 的签名- 例如，生成下一个区块所需的时间内的二十个签名。一个全节点将需要等待这二十个签名，或者从一个已充分确认的区块开始（例如，回退二十个区块），自己验证区块，然而这可能并不容易。

所以可以看出，DPOS 算法的明显缺点是，只有在挖掘二十个或更多的区块后，一个新区块（以及其内部的交易）才能达到相同的信任水平（"递归可靠性"如2.6.28中所讨论的），而BFT 算法则马上可以提供这种信任水平（例如，二十个签名）。另一个缺点是，DPOS 使用"最长分叉胜出"的方法来切换到其他分叉；如果有一些生产者在我们感兴趣的那个区块之后无法生成后续的区块（或者由于网络分割或复杂的攻击而未能观察到这些区块），这使得很有可能会发生分叉。

我们认为，BFT 方法，虽然更复杂且产生区块的间隔时间比DPOS长，但它更适合用于「紧密耦合」(参考2.8.14)的多链系统。因为其他的区块链可以在看到新区块中的已提交交易（例如，生成给它们的消息）后几乎立即开始行动，而不必等待20 次确认有效性（即，接下来的20 个区块），或者等待接下来的六个区块以确保没有分叉出现，并自己验证新区块（在可扩展的多链系统中，验证其他区块链的区块可能变得过于繁重）。因此，他们可以在保持高度的可靠性和可用性（参考2.8.12）的同时达到可扩展性。

当然，另一方面，对于一个「松散耦合」的多链系统，DPOS 可能是一个好选择，即区块链之间不需要快速交互，例如，如果每个区块链（「工作链」） 代表一个单独的分布式交换，且区块链之间的互动限于将代币（token）从一个工作链到另一个工作链（或者，更准确地说，以接近1 : 1 的兑换比率将一个工作链的代币兑换成另一个工作链的代币）。这就是BitShares 计划和实际上所做的事情，它非常成功地使用了DPOS。

总的来说，虽然DPOS可以生成新的区块并且将交易快速包含进区块（区块间隔时间较短），但这些交易为达到和其他区块链和链下（off-chain ）应用程序所需的信任级别，即作为「已提交」和「不可变」的，比在BFT系统中慢得多， 例如，三十秒而不是五秒。（例如，EOS，迄今为止提出的最好的DPOS 项目之一，承诺45 秒的确认和区块链间的交互延迟（参考[5]，「交易确认」和「区块链间通信的延迟」部分））。

更快的交易包含并不意味着更快的交易承诺（译者注：即确认一个区块是"不可变"的）。如果需要区块链间进行快速交互，这可能会成为一个巨大的问题。在这种情况下，人们必须放弃DPOS，而选择BFT PoS。

2.8.6 支持在交易里执行图灵完备的智能合约代码

区块链项目通常在他们的区块中收集一些交易，然后会以一种被认为有用的方式更改区块链的状态（例如，从一个账户转移一定加密货币金额到另一个账户）。有些区块链项目可能只允许一些特定的预定义交易类型（如从一个账户到另一个账户的值转移，并提供正确的签名）。其他区块链项目可能支持在交易中的执行一些有限形式的脚本。最后，一些区块链项目支持在交易中执行任意复杂的程序代码，使系统（至少原则上）能够支持任

意应用程序，只要系统的性能允许。这通常与「图灵完备虚拟机和脚本语言」相关（意味着可以在任何其他计算语言中编写的任何程序都可以重写，以在区块链内部执行），以及和「智能合约」（存储在区块链 中的程序）相关联。

当然，支持任意智能合约使系统变得真正灵活。另一方面，这种灵活性是有代价的：这些智能合约的程序码必须在某种虚拟机上执行，每次有人想创建或验证一个区块时，都必须对区块中的每个交易执行这个操作。与可以通过在像C++ 这样的语言中，实现优化的预定义处理和不可变的简单交易类型相比，这降低了系统的性能。

最终，对于任何通用区块链项目，似乎都希望支持图灵完整的智能合约;否则，区块链项目的设计者必须预先决定他们的区块链将用于哪些应用程序。实际上，比特币区块链中对智能合约的支持不足，是必须创建新的区块链项目Ethereum 的主要原因。在一个（异构的; 参照 2.8.8） 多链系统中，人们可以通过在一些区块链（即，工作链）中支持图灵完备的智能合约，以及在其他区块链中支持一小组高度优化的交易类型，来「两全其美」。

2.8.7 多链系统的分类

到目前为止，这个分类对单链和多链系统都是有效的。然而，多链系统有更多的分类标准，反映系统中不同的区块链之间的关系。我们现在讨论这些标准。

2.8.8 区块链类型：同构区块链和异构系统

在多链系统中， 所有区块链可能本质上都是相同的类型，并且有相同的规则（即，使用相同的交易格式、相同的虚拟机器执行智能合约程序代码、共享相同的加密货币等），这种相似性被明确地利用，但每个区块链中的数据都是不同的。在这种情况下，我们说该系统是同构的（homogeneous）。否则，不同的区块链（在这种情况下通常被称为工作链）可以有不同的「规则」。然后我们说系统是异构 （heterogeneous）。

2.8.9 同构-异构混合系统

有时我们有一个混合系统，存在多个不同的区块链的类型或规则集合，但其中也有很多区块链具有相同规则。那么这样一个系统就是一个混合的同构-异构系统（heterogeneous-homogeneous system）。正如我们看到的，TON 区块链 是这种系统的唯一例子。

2.8.10 具有相同的规则的异构系统工作链：联盟

在某些情况下，具有相同规则的多个区块链（workchains）可以表现得就像异构系统一样，但它们之间的交互与具有不同规则的区块链之间的交互本质上是相同。即使他们似乎使用「相同的」原生加密货币（如Ton coin），事实上他们使用不同的代币「altcoins」。有时，人们甚至可以使用接近1 : 1 的汇率兑换这些代币。然而，我们认为这并不使系统变得同构化；它仍然是异构的。我们说这样具有相同规则的的异构工作链集合是一个联盟（confederation）。

虽然制作一个异质系统，允许创建具有相同规则的多个工作链（即一个联邦）可能看起来是建立可扩展系统的便宜方法，但这种方法也有很多缺点。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一个大型项目，那么得到的不是一个大型项目，而是该项目的许多小实例。这就像有一个聊天应用程序（或游戏）只允许每个聊天（或游戏）房间有最 多50 名成员，但「扩展」可以在必要时通过创建新房间容纳更多用户。因此使得许多用户可以参加聊天或游戏，但我们能说这样的系统是真的可扩展吗?

2.8.11 内部主链和外部主链

有时，一个多链项目有一个特殊的「主链」（有时被称为「控制区块链」），用于储存系统的整体配置（所有活跃的区块链集合，或更准确地说是工作链）、 当前的验证节点集合（对于一个Proof- of-Stake 系统）等。有时其他的区块链被「绑定」到这个主链，例如通过将它们最新的区块的hash 值提交到它（TON 区块链 也是这么做的）。（译者注：这本质是内部主链。）

在某些情况下，主链是外部的，这意味着它不是该区块链系统的一部分，而是一些已经预先存在的区块链，而这些区块链和原区块链是完全无关的。例如，人们可以尝试使用Ethereum 区块链作为一个外部的主链， 并为此目的在Ethereum 区块链上发布特殊的智能合约（例如，用于选举和惩罚验证节点）。

2.8.12 分片支持

有些区块链项目（或系统）原生支持分片，这意味着这些（必然是同构的；参见2.8.8）区块链被视为单链（从高级角度看）虚拟区块链的分片。例如，人们可以创建256个具有相同规则的分片区块链，并根据其account_id的第一个字节将帐户的状态保存在所选的分片中。

分片是一种很自然的扩展区块链系统的方法，因为，如果正确实施，系统中的用户和智能合约根本不需要知道分片的存在。事实上，当负载过高时，人们经常希望在现有的单

链项目（如Ethereum）中添加分片。

另一种扩展方法是使用在2.8.10中描述的异构工作链 的「联盟」， 允许每个用户在一个或多个选择的工作链中保留帐户信息，并在必要时将资金从一个工作链转移到另一个工作链，本质上进行1 : 1 的代币兑换操作。此方法的缺点已在2.8.10中讨论过。

但是，以快速可靠的方式实现分片并不容易，因为它意味着不同的分片链之间有很多消息。例如，如果帐户在N 个分片之间均匀分布，且唯一的交易类型是从一个帐户到另一个帐户的简单资金转移，那么所有交易的只有一小部分（1/ N）将在单链区块链中执行；几乎所有的（1 – 1/N）交易将涉及两个区块链，需要跨区块链的通信。如果我们希望这些交易快速执行，我们需要一个用于在分片链之间快速传输消息的系统。换句话说，区块链项目需要在2.8.14中描述的意义上是「紧密耦合的」。

2.8.13 动态和静态分片

分片可能是动态（当需要时自动创建额外的分片）或是静态的（有预先定义的分片数量，最好只能通过硬分叉来改变）。大多数分片提议都是静态的；而TON 区块链 使用的是动态分片（参见2.7）。

2.8.14 区块链之间的交互:松散耦合和紧密耦合系统

多区块链项目可以根据区块链之间支持的交互水平进行分类。

最低的支持水平是不同区块链之间完全没有任何互动。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部分，而只是相同区块链协议的单独实例。

下一个支持水平是缺乏对区块链之间的消息传送的任何具体支持，使得原则上可能进行互动，但本质上很尴尬。我们称这样的系统为「松散耦合」；在这些系统中，人们必须像发送消息和转移资金一样，假设它们是属于完全独立的区块链项目的区块链（例如，比特币和以太坊；想像两方希望将储存在比特币区块链中的比特币兑换成储存在以太坊区块链中的Eth）。换句话说，必须在源区块链的区块中包括出站消息（或其生成交易）。然后（或其他某方）必须等待足够的确认（例如，后续区块的给定数量），来确认原始交易已被「提交」并「不可变」，然后再执行外部操作。只有这样，传递到目标区块链的交易消息（也许还带有源交易的引用和Merkle 存在证明）才算正式被提交。。

如果在传输消息之前没有等待足够的时间，或者由于某些其他原因发生了分叉，那么两个区块链的联合状态将被证明是不一致的：一条消息被递送到第二个区块链，而该消息从未在第一个区块链中生成（最终选择了分叉）。

有时会添加一些对消息的相关处理，例如会标准化所有工作链区块中的消息格式和输入和输出消息队列的位置（这在异构系统中尤其有用）。这在某种程度上促进了消息传递，它在概念上与之前的情况没有太大区别， 所以这样的系统仍然是「松散耦合」的。

相反，「紧密耦合」的系统包括特殊的机制，以实现在所有区块链之间的快速传递消息。希望实现的目标是：能够在生成源区块链的区块之后，立即将消息递送到另一个工作链。另一方面，「紧密耦合」的系统还应该在出现分叉的情况下维护整体的一致性。尽管这两个要求乍看之下似乎是矛盾的，但我们相信TON 区块链 使用的机制（将分片链区块 hash 包含到主链区块中；使用「垂直」区块链来修复无效的区块，参见2.1.17；超立方体路由，参见2.4.19；即时超立方体路由，参见2.4.20）使其成为一个「紧密耦合」的系统，也许是迄今为止唯一这样的系统。

当然，建立一个「松散耦合」的系统要简单得多；但是，快速和高效的分片（参见2.8.12）要求系统是「紧密耦合」的。

2.8.15 简化区块链项目分类

我们到目前为止建议的分类方式将所有区块链项目分割成多个类别。然而，我们使用的分类准则在实践中确实具有很大的相关性。这使我们能够提议一种简化的「世代」方式来分类区块链项目，作为对现实的一个大致分类， 并给出一些例子。每一代的最重要的特征都用粗体字显示。

- 第一代：单链，PoW，不支持智能合约。范例：Bitcoin (2009) 和许多其他不那么有趣的模仿者（如Litecoin、Monero 等）。
- 第二代：单链，PoW，支持智能合约。范例：Ethereum (2013; 2015 年部署)

- 第三代：单链，PoS，支持智能合约。范例：Ethereum (2022正式升级成POS)。
- 替代的第三代(3’)：多链，PoS，不支持智能合约，松散耦合。范例： Bitshares (2013–2014; 使用DPOS)。
- 第四代：多链，PoS，支持智能合约，松散耦合。范例：EOS (2017; 使用DPOS)，PolkaDot(2016; 使用BFT)。
- 第五代：多链，使用BFT 的PoS，支持智能合约，紧密耦合，支持分片。范例：TON (2017)。

尽管并非把所有区块链项目都确切地落入这些类别中，但大多数确实如此。

2.8.16 改变区块链项目“基因组”(genome）的复杂性

上述分类定义了区块链项目的「基因组」。这个基因组非常「固定」：一旦项目部署并被许多人使用，就几乎不可能更改它。更改它需要一系列的硬分叉（这需要社区的大部分人批准），即使如此，更改还需要非常保守，以保持向后兼容性（例如，更改虚拟机的语义可能会破坏现有的智能合约） 。另一种方法是创建具有不同规则的新「侧链」，并将它们以某种方式绑定到原始项目的区块链（或区块链）。人们可能使用现有单链项目作为外部主链,但是本质上是一个新的和独立项目。 （例如，Plasma 项目计划使用 Ethereum 区块链作为其（外部）主链；它与Ethereum的其他部分互动不多，且可能由与 Ethereum 项目无关的团队提议和实施。）

我们的结论是，一旦部署了项目的基因组，就很难更改它。即使从PoW开始，并计划在未来用PoS 替换它，也相当复杂。（到2017 年为止，Ethereum 仍在努力从PoW 过渡到结合的PoW+PoS 系统；我们希望它将来会成为一个真正的PoS 系统。）为原始设计并不支持分片的项目添加分片，基本上不可能。（Ethereum 的分片提议可以追溯到2015年；目前还不清楚它们如何被实施和部署，而不会破坏Ethereum 或创建一个本质上独立的平行项目。）实际上，将智能合约的功能添加到一个（即Bitcoin）原始设计不支持这些功能的项目中，也被认为是不太可能的（或至少被Bitcoin 社区的大多数人认为是不太期望的功能），最终导致了一个新的区块链项目，Ethereum。

2.8.17 TON 区块链的"基因组"

因此，如果想建立一个可扩展的区块链系统，必须从一开始就仔细选择其基因组。如果系统在部署时预期在未来会支持一些特定功能，那么它应该从一开始就支持「异构」工作链（可能具有不同的规则）。为了让系统真正可扩展，它必须从一开始就支持分片；只有当系统是「紧密耦合」时，分片才有意义（参考2.8.14），而这些则意味着需要存在主链，快速的区块链间消息系统，BFT PoS 的使用等。

考虑到所有这些含义，为TON 区块链项目做出的大多数设计选择似乎都是自然的，并且几乎是唯一可能的选择。

2.9 与其他区块链项目的比较

我们试图通过比较现有和建议的区块链项目，从而对TON 区块链及其最重要和独特功能进行简单的总结。我们使用在2.8中描述的分类标准以统一的方式，讨论了不同的区块链项目，并建构这样一个「区块链项目地图」。我们将此地图表示为表 1，然后简要地单独讨论几个项目，以指出它们的特性可能不适合一般的区块链方案。

项目	宣告+部署年份	世代	共识算法	智能合约	单链/多链	异构/混合多链	分片	链间交互
Bitcoin	2009	1	POW	不支持	单链	/	/	/
Ethereum	2013,2015	2	POW-> POS	支持	单链	/	/	/
NXT	2014	2+	POS	不支持	单链	/	/	/
Tezos	2017	2+	POS	支持	单链	/	/	/
Casper	2015,2017	3	POW/ POS	支持	单链	/	/	/
BitShares	2013,2014	3'	DPOS	不支持	多链	异构	不支持	松散耦合
EOS	2016,2018	4	DPOS	支持	多链	异构	不支持	松散耦合
PolkaDot	2016，2019	4	POS BFT	支持	多链	异构	不支持	松散耦合
Comos	2017	4	POS BFT	支持	多链	异构	不支持	松散耦合
TON	2017,2018	5	POS BFT	支持	多链	混合	动态分片	紧密耦合

表 1: 一些值得注意的区块链项目的摘要。各列分别为：项目名称；宣告年份和部署年份；世代(参考2.8.15)；共识算法(参考2.8.3和2.8.4)；支持任意程序代码（智能合约；参考2.8.6）；单链/多区块链系统(参考2.8.2)；异构/同构多链系统(参考2.8.8)；支持分片(参考2.8.12)；支持区块链间的交互，松散耦合或紧密紧密(参考2.8.14)。

2.9.1 Bitcoin [12]; https://bitcoin.org/

Bitcoin (2009) 是第一个且最知名的区块链项目。它是典型的第一代区块链项目：它是单一链，使用工作量证明(Proof-of-Work) 与「最长分支胜出」的分支选择算法，并且

没有图灵完整的脚本语言（但是，支持没有循环的简单脚本）。 Bitcoin 的区块链没有帐户的概念；它使用UTXO（未花费的交易输出）模型。

2.9.2 Ethereum [2]; https://ethereum.org/

Ethereum (2015) 是第一个支持图灵完整智能合约的区块链。因此，它是典型的第二代区块链项目，且是其中最受欢迎的。它之前在单一区块链上使用工作量证明，支持智能合约和帐户模型。

2.9.3 NXT; https://nxtplatform.org/

NXT (2014) 是第一个基于权益证明(PoS) 的区块链和货币。它仍然是单链，并且不支持智能合约。

2.9.4 Tezos; https://www.tezos.com/

Tezos (2018 或之后) 是一个提议的基于PoS 的单一区块链项目。我们在这里提到它是因为它的独特功能： 其区块解释函数ev_block（参考2.2.6）不是固定的，而是由一个OCaml 模块决定，该模块可以通过在区块链中提交一个新版本（并为所提议的更改收集一些投票）来升级。通过这种方式，一个人将能够通过首先部署一个「原始」的Tezos 区块链，然后逐步地将区块解释函数更改为所需的方向， 而无需硬分支。

这个想法，尽管引人入胜，但它有一个明显的缺点，那就是它禁止在其他语言（如C++）中的任何优化实现，因此基于Tezos 的区块链注定性能较低。我们认为，可以通过发布所提议的区块解释函数ev_trans的正式规范，获得类似的结果，而不是一个固定的特定实现。

2.9.5 Casper

Casper是Ethereum 的即将到来的PoS 算法；如果在2017 年（或2018 年）的逐步部署成功，它将使Ethereum 变成一个带有智能合约支持的单链PoS 或混合PoW+PoS 系统，将Ethereum 转化为第三代项目。
（https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/）

2.9.6 BitShares [8]; https://bitshares.org

BitShares (2014) 是一个基于分布式区块链交易所的平台。这是一个异构的多区块链DPoS 系统，且不带有智能合约；它透过仅允许一小部分预先定义的专用交易类型来实现高性能，这些交易类型可以在C++ 中有效地实现，前提是改造区块链状态以适合在内存

中存储。这也是第一个使用委托权益证明(Delegated Proof-of-Stake, DPoS) 的区块链项目，至少证明了其可行性。

2.9.7 EOS [5]; https://eos.io.

EOS (2018 或以后) 是一个建议的异构多区块链DPoS 系统，并带有智能合约支持和一些最小的消息传递支持（在2.8.14描述的意义上仍然是松散连接的）。这是先前成功创建了BitShares 和SteemIt 项目的同一团队的尝试，展示了DPoS 共识算法的强大之处。通过为需要它的项目创建专用的工作链（例如，一个分布式交易所可能使用一个支持一组专用的优化交易的工作链，类似于BitShares 所做的那样）以及创建具有相同规则的多个工作链来实现可伸缩性（在2.8.10描述的意义上是联盟）。这种可伸缩性方法的缺点和限制已在本文中讨论过。另见2.8.5、2.8.12和2.8.14，进一步讨论 DPoS、分片、工作链之间的互动及其对区块链系统的可伸缩性的影响。

与此同时，即使一个人不能在区块链内「创建一个Facebook」（参见2.9.13），不论是EOS 还是其他方式，我们认为EOS 可能会成为某些高度专用的弱互动分布式应用程序的便利平台，类似于BitShares（去中心化交易所）和SteemIt（去中心化博客平台）。

2.9.8 PolkaDot [17]; https://polkadot.io

PolkaDot (2019 或之后) 是最佳思考且最详细的提议多链权益证明项目之一；其开发是由Ethereum 的共同创办人之一领导的。此项目是我们地图上与TON 区块链最接近的项目。 （事实上，我们的「fisherman」和「nominator」术语都是归功于PolkaDot 专案。 ）

PolkaDot 是一个异构的松散连接的多链权益证明项目，具有用于生成新区块的拜占庭容错（BFT）共识和主链（可能是外部的，例如Ethereum 区块链）。它还使用超立方体路由，有点像TON的慢速路径版本，如2.4.19所述。

其独特的功能是它不仅可以创建公共的，而且还可以创建私有的区块链。这些私有区块链也将能够与其他公共区块链互动，不论是PolkaDot 还是其他的。

因此，PolkaDot 可能会成为大规模私有区块链的平台，例如，银行联盟可能会使用它来快速相互转账，或者大型公司可能使用它的私有区块链技术用作其他用途。

然而， PolkaDot 不支持分片且区块链之间不是紧密耦合的。这在某种程度上限制了其可伸缩性，与EOS 相似。 （或许稍好一些，因为PolkaDot 使用BFT PoS 而非DPoS。）

2.9.9 Universa; https://universa.io

我们之所以在此提及这个不寻常的区块链项目，是因为它是到目前为止唯一提到与我们的无限分片范式相似的项目（参见2.1.2）。其另一个特点是它通过只有项目的受信任和持有许可的合作伙伴才会被认可为验证节点，因此永远不会提交无效的区块，来绕过所有与Byzantine Fault Tolerance相关的复杂性。这是一个有趣的决定；但是，它本质上使一个区块链项目故意变得集中化，这是区块链项目通常想要避免的（在一个受信任的集中化环境中为什么需要区块链？）。

2.9.10 Plasma; https://plasma.io

Plasma (2019?) 是Ethereum 的另一位共同创办 人的非传统区块链项目。它旨在缓解Ethereum 的某些限制，而不引入分片。本质上，它是一个与Ethereum 分开的项目，引入了一系列的（异构的）工作链，与最高级别的 Ethereum 区块链绑定（作为一个外部主链）。资金可以从层次结构中的任何区块链转移出去（从作为根的Ethereum 区块链开始），并伴随一个要完成的工作描述。然后在子工作链中进行必要的计算（可能需要将原始工作的部分进一步转发到树的下方）， 将结果传递上去，并收集奖励。通过一个（由支付通道payment channel启发的）机制来避免实现这些工作链的一致性和验证的问题，该机制允许用户单方面从一个行为不端的工作链撤回资金到其父工作链（尽管很慢），并将资金和工作重新分配到另一个工作链。

因此，Plasma 可能成为绑定到 Ethereum 区块链的分布式计算平台，就像一个数学协处理器（mathematical co-processor ）。但是，这似乎不像是实现真正的通用可伸缩性的方法。

2.9.11 垂直领域区块链项目

还有一些专用领域的区块链项目， 如FileCoin（一个激励用户为储存愿意支付费用的其他用户的文件提供磁盘空间的系统），Golem（一个基于区块链的平台，用于租用和出借计算能力给专门的应用，如3D-rendering）或SONM（另一个类似的计算能力出借项目）。这类项目在区块链组织的层面上没有引入任何新概念上的事物；相反，它们是特定的区块链应用，可以由运行在通用目的区块链中的智能合约实现，只要它可以提供所需的性能。因此，这类项目可能会使用现有或计划中的区块链项目作为其基础，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； 如果需要在一个联盟背景下工作，为明确的目的定义一系列的工作链 ，那么可以选择EOS 或PolkaDot。

2.9.12 TON 区块链

TON（The Open Network）区块链 （计划于2018年）是我们在本文档中描述的项目。它旨在成为第一个第五代区块链项目 即BFT PoS 多链项目，支持混合同构/异构，支持自定义工作链，具有原生的分片支持，并且紧密耦合（特别是，能够在保持所有分片链的一致状态时，几乎立即转发分片之间的消息）。因此，它将是一个真正可扩展的通用区块链项目，基本上能够容纳可以在区块链中实现的任何应用程序。当与TON的其他组件结合时（参见第一章），其实现可能性会更大。

2.9.13 在区块链上构建Facebook是否可行?

有时候人们声称， 将Facebook 这样规模的社交网络作为分布式应用程序部署在区块链中是可能的。通常会引用一个受喜爱的区块链项目作为这种应用的可能"主机"。

我们不能说这是技术上的不可能。当然，需要一个紧密耦合的区块链项目，具有真正的分片（即TON），以便这种大型应用不会运行得太慢（例如，从一个分片链 中的用户传送消息和更新到另一个分片链中的朋友，且延迟的时间在合理范围）。然而，我们认为这是不需要的，且永远不会被完成，因为价格会过高。

让我们考虑将"将Facebook 上载到区块链"作为一个思考实验；任何其他相似规模的项目也可能作为一个示例。一旦Facebook 被上载到区块链，目前由Facebook 的服务器完成的所有操作，将被序列化为某些区块链的交易（例如，TON 的分片链），并将由这些区块链的所有 验证节点 执行。每项操作都必须执行，比如说，至少20 次，如果我们希望每个区块至少收集20 个验证节点 签名（立即或最终，如在DPOS 系统中）。同样， Facebook 服务器在其磁盘上保留的所有数据将被保留在相应分片链 的所有验证节点 的磁盘上（即，至少有20 份副本）。

因为验证节点 基本上是与Facebook 目前使用的相同服务器（或许是服务器集群，但这不影响此论点的有效性），我们可以看出，将Facebook 运行在区块链 中的总硬件开销至少比传统方式实现高出20 倍。

事实上，开销还会更高，因为区块链的虚拟机比运行优化编译程序代码的"裸CPU" 要慢，且其储存未针对Facebook 特定的问题进行优化。通过为Facebook 设计具有某些特殊交易的特定工作链，可以部分地减轻此问题；这是BitShares 和EOS 实现高性能的方法，也可在TON 区块链 中使用。然而，一般的区块链 设计本身仍然会带来一些额外的限制，例如需要将所有操作注册为区块中的交易，将这些交易组织成Merkle tree，计算和检查它们的Merkle hashes，进一步传播这个区块 等。

因此，保守估计是，为了验证承载该规模社交网络的区块链 项目， 需要的服务器性

能是Facebook 现在使用的服务器的100 倍。有人将不得不为这些服务器付钱，无论是拥有分布式应用的公司（想像一下每个Facebook 页面上有700 条广告，而不是7 条）还是它的用户。无论哪种方式，这在经济上似乎都不可行。

我们认为，不是所有东西都应该上传到区块链中。例如，不必在区块链 中保留用户照片；将这些照片的hashes 注册到区块链 中， 并将照片保存在分布式链下（off- chain） 储存（例如FileCoin 或TON Storage）中， 可能是更好的选择。这就是为什么TON不仅仅是一个区块链 项目，而是围绕TON 区块链为中心的几个组件的集合，如章节1和5所概述的。

3. TON网络协议

任何区块链 项目不仅需要区块格式和区块链 验证规则的规范， 还需要一个用于传播新区块、发送和收集候选交易等的网络协议。换句话说，每个区块链 项目都必须设置一个专门的点对点网络。这个网络必须是点对点的，因为通常希望区块链 项目是分布式的，所以不能依赖于一组集中式的服务器并使用传统的客户端-服务器架构， 例如，传统的网络银行应用程序所做的。即使是轻量级客户端（例如，轻量级加密货币钱包的智能型手机应用程序），它们必须以客户端-服务器的方式连接到全节点，如果先前的节点停止运作，实际上它们可以自由地连接到另一个全节点，只要用于连接到全节点的协议足够标准化。

虽然如Bitcoin或Ethereum这样的单一区块链项目的网络需求可以很容易地被满足（基本上需要构建一个"随机"点对点的覆盖网络，并通过一个gossip 协议传播所有新的区块 和候选交易） ，但像TON 区块链 这样的多区块链 项目则要求更高（例如，人们必须能够订阅只有某些分片链的更新，而不一定是所有的分片）。因此，TON 区块链网络部分和作为整个TON 项目的一部分，至少值得简要的讨论。

另一方面，一旦需要支持TON 区块链的更为复杂的网络协议， 事实证明它们可以很容易地用于不一定与TON 区块链 的直接需求相关的事情上，从而为TON 生态系统中创建新服务提供了更多的可能性和灵活性。

3.1 抽象数据报网络层（ANDL）

建立 TON 网络协议的基石是(TON) 的抽象数据报网络层（Abstract (Datagram) Network Layer）。它使所有节点能够使用 256 位"抽象网络地址"（“abstract network addresses”）表示特定的"网络身份"（“network identities”）， 并仅使用这些 256 位网络地址来识别发件人和收件人并进行通信（作为第一步，首先会互相发送数据报）。尤其是，人们不需要担心 IPv4 或 IPv6 地址、UDP 端号等；它们会被抽象网络地址隐藏。

3.1.1 抽象网络地址

一个抽象网络地址，或称为抽象地址，或简称为地址，是一个256 位整数，基本上等同于256 位ECC公钥。此公钥可以任意生成，因此节点可以根据喜好创建许多不同的网络身份。但是，为了接收（和解密）针对此地址的消息，人们必须知道相对应的私钥。

实际上，地址不是公钥本身；而是一个序列化TL-object (参见2.2.5)的256 位hash（Hash = sha256），这可以根据其构造函数（前四个字节）描述多种类型的公钥和地址。在最简单的情况下，此序列化TL-object 仅由一个4字节的幻数（译者注：幻数，magic number，计算机编程术语，常用于表示标识文件格式或协议的常量数字或独特的值，参考：https://en.wikipedia.org/wiki/Magic_number_(programming) ）和一个256 位椭圆曲线密码学（ECC）公钥组成；在此情况下，地址将等于这36字节结构的hash。然而，人们也可以使用2048 位RSA 钥匙或任何其他公钥密码学方案。

当一个节点获知另一个节点的抽象地址时，它还必须接收其「原像（preimage）」（即序列化TL-object， 其hash值等于该抽象地址）；否则，它将无法加密并发送数据报到该地址。

3.1.2 低层级的网络，UDP实现

从几乎所有TON 网络组件的角度看，唯一存在的就是一个网络（Abstract Datagram Networking Layer），它能够（不可靠地）从一个抽象地址发送数据报到另一个抽象地址。原则上，抽象数据报网络曾（Abstract Datagram Networking Layer (ADNL)） 可以在不同的现有网络技术上实施。但是，我们打算在IPv4/ IPv6 网络（如互联网或内部网）上实施它，并在UDP 不可用时选用TCP 作为备选。

3.1.3 基于UDP的简单ANDL例子

通过UDP 从发送者的抽象地址发送数据报到任何其他抽象地址，最简单情况的实现如下所示。

假设发送者以某种方式知道拥有目标抽象地址的接收者的IP地址和UDP端口，且接收者和发送者都使用从256 位ECC 公钥派生的抽象地址。

在此情况下，发送者简单地通过把ECC 签名（用其私钥完成）和其源地址（或源地址的原像，如果接收者还不知道该原像）附加到要发送的数据报。其数据报结果使用收件人的公钥加密，嵌入到一个UDP 数据报中，并发送到收件人已知的IP和端口。因为UDP 数据报的前256 位包含接收者的抽象地址，所以接收者可以确定应该使用哪个私钥来解密数据报的其余部分。只有在此之后，发送者的身份才会被揭示。

3.1.4 安全性较低的方式，发件人的地址为明文

有时，一个较不安全的方案就足够了，当收件人和发件人的地址在UDP 数据报中直接显示为明文的时候；发件人的私钥和收件人的公钥使用ECDH（椭圆曲线Diffie–Hellman）合并在一起，生成一个256 位共享密钥，之后，该共享密钥与明文部分和一个256位的随机数（nonce），一起用来生成出用于加密的AES 密钥。数据完整性的证明，可以通过在加密前，在明文数据后面添加上原始明文数据的hash值去实现。

这种方法的优点是，如果预计两个地址之间将交换多个数据报，则只需计算一次共享密钥然后将其缓存；然后加密或解密下一个数据报时不再需要执行较耗时的椭圆曲线操作。

3.1.5 通道和通道标识符

在最简单的情况下，携带内嵌TON ADNL 数据报的UDP 数据报的前256 位将等于收件人的地址。这样， 一般来说它们组成了一个通道标识符（channel identifier）。系统有不同类型的通道。其中一些是点对点的；它们由希望在将来交换大量数据的两方创建，并通过交换几个加密数据报（如3.1.3或3.1.4中所描述的那样），运行经典或椭圆曲线Diffie–Hellman（如果需要额外的安全性）来生成一个共享密钥，或者仅由一方生成一个随机共享密钥，并将其发送给另一方。

此后，通道标识符由共享密钥和一些额外数据（例如寄件人和收件人的地址）组合而来，例如通过hashing。该标识符将用作携带加密数据的UDP数据报的前256 位。

3.1.6 通道作为隧道标识符

一般而言，通道或通道标识符仅选择了接收者的已知的处理入站UDP 数据报的方法。如果通道是接收者的抽象地址，则处理的方式如3.1.3或3.1.4中所述；如果通道是3.1.5中讨论的已建立的点对点通道，则处理的方式包括使用共享密钥解密数据报，如本文之前的章节所解释的。

特别地，通道标识符实际上可以选择一个隧道（“tunnel”）,当接收者将接收到的消息转发给其他人时(实际的接收者或另一个代理时)。沿途可能会进行一些加密或解密步骤（这让人想起洋葱网络（“onion routing”） [6] 或者大蒜路由（“garlic routing”）https://geti2p.net/en/docs/how/garlic-routing），并且可能使用另一个通道标识符用于重新加密的转发数据包（例如，点对点通道可以用于将数据包转发给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次上添加对打洞（“tunneling“）和

代理（“proxying”）的支持---与TOR 或 I I P 项目提供的相似- --而不影响所有高层级 TON 网络协议的功能，因为上层协议对底层网络层数据报的处理无感知。这种功能可以被TON 代理服务所利用（参见4.1.11）。

3.1.7 零通道和引导问题

通常，一个TON ADNL 节点会有一些邻居表（“neighbor table“），其中包含有关其他已知节点的信息，例如它们的抽象地址、它们的preimages（即，公钥）及它们的IP 地址和UDP 端口。然后，它使用从这些已知节点获取到的信息，逐步扩展此表格，并有时删除过时的记录。

但是，当一个TON ADNL 节点刚启动时，它可能不知道任何其他节点， 只能知道一个节点的IP 地址和UDP 端口，但不知其抽象地址。例如，如果一个轻型客户端无法访问之前缓存的任何节点和硬编码到软件中的任何节点，并必须要求用户输入节点的IP 地址或DNS 域名，以通过DNS 进行解析。

在这种情况下，节点将发送数据包到该节点的特殊的零通道（“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），所以消息是不加密地传输的。它通常只用于获取接收者的身份（也许是为此目的特别创建的一次性身份），然后开始以更安全的方式通信。

一旦至少知道一个节点，则会发送特殊的查询到已知节点，根据返回的响应获取到更多的信息，从而轻松填充邻居表（“neighbor table“） 和路由表（“路由表”）。

不是所有节点都需要处理发送到zero channel 的数据包，但用于启动轻型客户端的节点应支持此功能。

3.1.8 基于ADNL 的 类TCP 流协议

ADNL，作为基于256位抽象地址的不可靠（小尺寸）数据包协议，可以用作更复杂网络协议的基础。例如，可以构建一个类似TCP 的流协议，使用ADNL 作为IP协议的抽象替代品。但是，TON Project 的大多数组件不需要这样的流协议。

3.1.9 RLDP，或 基于ADNL上的可靠大数据报协议

一个建立在ADNL 上的可靠的任意大小的数据包协议，称为RLDP，用于代替类似

TCP 的协议。这种可靠的数据包协议可以用来，例如，向远程主机发送RPC 查询，并从它们那里接收答案（参见4.1.5）。

3.2 TON DHT: 类Kademlia 的分布式哈希表

TON 分布式哈希表(*TON Distributed Hash Table*, DHT)在 TON 项目的网络部分中扮演了关键角色，用于定位网络中的其他节点。例如，一个想要提交交易到某个分片链 的客户端可能想要找到该分片链 的验证节点 或核对者，或至少某个可以将客户端的交易转发到核对者 的节点。这可以通过在 TON DHT 中查找特定的 key 来完成。 TON DHT 的另一个重要应用是，它只需查找随机 key，或新节点的地址，就可以快速填充新节点的邻居表(参见 4.0.7)。如果节点对其入站数据报使用代理和通道，则将通道标识符和其入口点信息（entry point）（例如，IP 地址和 UDP 端口）发布在 TON DHT 中；然后，所有希望发送数据报到该节点的节点，将首先将从 DHT 获取此联系信息。

TON DHT 属于Kademlia-like分布式哈希表家族[10]。

3.2.1 Ton DHT的key

TON DHT的keys仅仅是一个256 位整数。在大多数情况下， 它们可以作为一个 TL-serialized 对象的sha256 计算出来（参见2.2.5），被称为key 的preimage或key 的描述。在某些情况下，TON Network 节点的抽象地址（参见4.0.1）也可以用作TON DHT 的keys，因为它们也是256 位，而且它们也是TL-serialized 对象的hashes。例如，如果节点不害怕公开其IP 地址，任何知道其抽象地址的人，都可以通过在DHT 中以该IP地址作为key来找到它。

3.2.2 DHT的值

分配给这些256位 keys 的值本质上是有限长度的任意字节串。这样的字节串的解释由对应key 的preimage 所决定；通常查找key 的节点和储存key 的节点都知道。

3.2.3 DHT 的节点，半永久网络身份

存放在DHT 的节点上的TON DHT 的key- value 映射,本质上是TON网络的所有成员。为此，TON Network 的任何节点（除了某些非常轻的节点外），除了3.1.1中描述的任意数量的临时和永久抽象地址外，都至少有一个"半永久地址"，该地址将被识别为TON DHT 的成员。这个半永久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keys。如果节点不想揭露其"真实"身份，则生成一个仅用于参与DHT 的单独抽象地址即可。但是，这个抽象地址必须是公开的，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，我们既有256 位的keys 也有256 位的(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，由以下公式给出

$$dK(x, y) := (x \oplus y) \text{ 解释为一个无符号的256 位整数(25)}$$

其中$x \oplus y$ 表示两个相同长度的两个比特序列按位异或 (XOR)。

Kademlia 距离 在所有256位序列的集合2^{256}上引入了一个度量。特别地，我们可以推出以下几个特性：

$$dK(x, y) = 0 \text{ 当且仅当} x = y$$

$$dK(x, y) = dK(y, x)$$

$$dK(x, z) \leq dK(x, y) + dK(y, z)$$

另一个重要特性是，从x 出发到任何给定距离，只有一个点满足：$dK(x, y) = dK(x, y')$，者意味着$y = y'$。

3.2.5 类 Kademlia 的 DHTs 和 TON DHT

如果一个拥有256 位keys 和256 位节点地址的分布式哈希值表(DHT)， 预期将key K 的值保留在与K 最接近的 s 个Kademlia 节点上(即，与他们的地址到K 的Kademlia 距离最小的s 个节点)，我们称这个DHT为类Kademlia DHT。

这里的s 是一个小参数，比如说，s = 7，用于提高DHT 的可靠性（如果我们只在一个节点上保留key，即与K 最接近的节点，那么如果该节点离线，该key 的值将会丢失）。

根据此定义，TON DHT 是一个类Kademlia DHT。它是在3.1中描述的ADNL 协议上实现的。

3.2.6 Kademlia路由表

任何参与Kademlia-like DHT 的节点，通常都会维护一个Kademlia 路由表。对于TON DHT，它由n = 256 个桶（buckets） 组成，编号从0 到n − 1。第i 个桶将包含一些已知节点的信息（t个"最好"的节点，和可能一些额外的候选节点），这些节点与节点地址a 之间的Kademlia 距离 从2^i到$2^{i+1} - 1$。（如果一个桶中有足够多的节点，比如说，八个子桶，这取决于Kademlia 距离 的前四位。这将加快DHT 查找的速度。）这些信息包括它们的(半永久) 地址、IP 地址和UDP 端口，以及一些可用性信息，如最后一次ping 的时间和延迟。

当一个Kademlia 节点由于某个查询得知其他Kademlia 节点时，它将其包含到其路由表的适当桶中，首先作为候选节点。然后，如果该桶中的一些"最好"的节点失效（例如，长时间不回应ping 查询），它们可以被一些候选节点替换。通过这种方式来保持更新Kademlia 路由表。

Kademlia 路由表 中的新节点也被包含在3.1.7中描述的ADNL 邻居表中。如果Kademlia 路由表 的一个桶 中的 最好的 节点经常被使用，可以建立一个在3.1.5中描述的通道，以方便数据报的加密。

TON DHT 的一个特殊特性是，它试图选择往返延迟最小的节点作为Kademlia 路由表 的桶的"最好"的节点。

3.2.7 Kademlia 网络查询

一个Kademlia 节点通常支持以下网络查询：

- Ping – 检查节点的可用性。
- Store(key, value) – 要求节点保存value作为key 的值。对于TON DHT，Store 查询稍微复杂一些(参考3.2.9)。
- Find_Node(key, L) – 要求节点返回到key节点的L个最接近的Kademlia已知节点（从其Kademlia 路由表中查找）。
- Find_Value(key, L) – 如上所述，但如果节点知道对应的key值，则直接返回该值。

当任何节点想要查找key K 的值时，它首先创建一组s′节点的集合S（对于某些小的s ′值，例如，s ′= 5），这些节点相对于所有已知节点的Kademlia距离最接近K（即，它们是从Kademlia 路由表中取出的）。然后，发送Find_Value 查询给它们中的每一个，并把

响应信息提到的节点包含在S 中。然后， 向S中最接近K 的s'个节点也发送Find_Value查询（如果以前没有这么做），并且该过程不断重复，直到找到该值或集合S停止增长。这是一种寻找相对于Kademlia 距离最接近K 的节点的 束搜索（"beam search"）算法。

如果要设置某个key K 的值，则为s' ≥ s 执行相同的过程，使用Find_Node查询而不是Find_Value，以找到最接近K 的s个节点。之后，向所有这些节点发送Store 查询。

在Kademlia 类似的DHT 的实现中还有一些不太重要的细节（例如，任何节点应该每小时查找一次s个最接近自己的节点，并通过Store 查询的方式重新发布所有储存的key给其他节点）。不过我们暂时忽略这些细节。

3.2.8 引导一个Kademlia节点

当一个Kademlia 节点上线时，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，它识别出最接近自己的s 个节点。它可以从它们那里下载所有已知的(key, value) 对， 以填充DHT表。

3.2.9 存储值到TON DHT

在TON DHT 中储存值与一般的类Kademlia DHT 略有不同。当希望储存一个值时，不仅必须将key K 本身提供给Store 查询，还必须提供其preimage---也就是说，一个TL序列化的字符串（在开头有几个预定义的TL -构造器），其中包含key的描述。此key描述，以及key和value，稍后会由节点一起保存。

key描述 主要描述了储存的对象的类型、所有者（owners）以及在未来更新时的 更新规则。所有者通常由包含在key描述中的公钥识别。如果包含所有者信息，通常只接受由相应的私钥签名的更新。储存的对象的类型 通常只是一个字节字符串。但是，在某些情况下，它可以更为复杂---例如，一个输入隧道描述（input tunnel description）（参考3.1.6），或节点地址的集合。

更新规则也可能不同。在某些情况下，它们仅允许使用新值替换旧值，前提是新值已经由所有者签名（签名必须作为值的一部分保存，以便稍后由其他节点在获得此key的值后进行检查）。在其他情况下，旧值以某种方式影响新值。例如，它可以包含一个序列号，并且仅当新的序列号更大时才覆盖旧值（以防止重放攻击）。

3.2.10 TON DHT中的分布式"流追踪器"和"网络兴趣组"

另一个有趣的情况是当值包含一个节点列表，也许是它们的IP 地址和端口，或者只是它们的抽象地址和更新规则，如果可以确认请求者的身份，更新规则还包含请求者列表。

这种机制可以用于创建一个分布式的流追踪器 （"torrent tracker "），所有对某个

torrent （即，某个文件）感兴趣的节点，可以找到对同一个torrent 感兴趣或已经拥有副本的其他节点。

TON Storage (参考4.1.8)使用此技术来找到拥有所需文件副本的节点（例如， 分片链的状态快照或旧的区块）。然而，其更重要的用途是创建多播子网络（“overlay multicast subnetworks“）和网络兴趣组（“network interest groups”） （参考3.3）。其想法是只有一些节点对特定分片链的更新感兴趣。如果分片链的数量变得非常大，则找到对相同分片感兴趣的节点可能会变得很复杂。这个分布式流追踪器（“distributed torrent tracker”）提供了一种方便的方法来找到这些节点。 另一个选项是从验证节点 那里请求它们，但这不会是一个可扩展的方法， 验证节点 可能选择不回应任何来自未知节点的这种查询。

3.2.11 Fall-back keys

目前描述的大多数key类型在其TL 描述中都有一个额外的32 位整数字段，通常等于零。但是，如果通过hash 该描述获得的key，不能从TON DHT 中检索或更新，则会增加此字段中的值，并进行新的尝试。这样，攻击者即使创建许多靠近被攻击键的抽象地址，并控制相应的DHT 节点，也不能捕获（“capture”）和删剪（“censor”）这个key的内容（即，进行键保留攻击）。

3.2.12 查找服务

一些位于TON网络和基于（TON ADNL）较高级别的协议（在3.1中描述）的服务，可能希望在某处公开服务的抽象地址，以便使用服务的客户知道在哪里找到它们。

但是，将服务的抽象地址发布到TON 区块链中可能不是最好的方法，因为可能需要经常更改抽象地址，并且可能会有多个地址，出于可靠性或负载均衡的目的。

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的key，在TL 描述字符串（参考2.2.5）中指出该公钥为其“owner“，以发布服务的抽象地址的最新列表。这是TON Services 利用的方法之一。

3.2.13 定位Ton区块链的账户所有者

在大多数情况下，TON 区块链账户的所有者不希望与抽象网络地址，特别是IP 地址相关联，因为这可能侵犯他们的隐私。然而， 在某些情况下，TON 区块链账户的所有者可能想要发布一个或多个相关联的抽象地址。

一个典型的情况是TON支付的闪电网络（“lightning network”）中的节点（参考

5.2），这是即时加密货币转账的平台。一个公开的TON支付节点可能不仅想与其他节点建立支付通道，还想发布一个抽象网络地址，然后使用该地址访问已建立的通道进行支付。

一种方法是在创建支付通道的智能合约中包含一个抽象网络地址。更灵活的方法是在智能合约中包括一个公钥，然后像在3.2.12中解释的那样使用DHT。

最自然的方法是使用TON 区块链中账户的同一私钥，来签名和发布关于与该账户相关的抽象地址的TON DHT 中的更新。这几乎与在3.2.12中描述的方式相同；但是，所使用的DHT 键将需要一个特殊的key的描述，只包含account_id本身，等于"账户描述"的sha256，其中包含账户的公钥。该值中包含的签名也将包含账户描述。

通过这种方式，提供了一种定位TON 区块链账户所有者的抽象网络地址的机制。

3.2.14 定位抽象地址

请注意，尽管TON DHT 是在TON ADNL 上实现的， 但TON ADNL 也用它来实现几个目的。

其中最重要的是从其256位抽象地址中，定位节点或节点的联系人数据。这是必要的，因为TON ADNL 应该能够向任意256 位抽象地址发送数据包， 即使没有提供任何其他信息。

为此，只需把256 位抽象地址作为一个key在DHT中进行查找相应的内容。可能会发生以下两种情况，第一种情况:找到使用该地址的节点（即，使用此地址作为公共半永久DHT 地址），在这种情况下，可以获得节点IP 地址和端口；第二种情况：可以查询得到一个由正确的私钥签名的输入隧道描述，在这种情况下，将使用此隧道描述，将ADNL数据包发送到预期的接收者。

请注意，为了使抽象地址"公开"（可以从网络中的任何节点到达），其所有者必须使用它作为半永久 DHT 地址，或在考虑的抽象地址下的DHT key中发布一个输入隧道描述，使用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。另一个选择是简单地发布其IP 地址和UDP 端口。

3.3 覆盖网络和多播消息

在像TON 区块链这样的多区块链系统中，即使是全节点也通常只对获取某些分片链的更新（即新的区块）感兴趣。为此，必须在TON Network 内部构建一个特殊的覆盖（overlay）子网络，基于在3.1中讨论的ADNL 协议， 每个分片链一个。

因此，需要建立任意的覆盖子网络，对希望参与的任何节点开放。这些覆盖网络中将运行基于ADNL 的特殊gossip协议。特别是，这些gossip 协议可用于在这样的子网络内部传播（广播）任意数据。

3.3.1 覆盖网络

一个覆盖（子）网络只是在某个更大的网络内部实现的（虚拟）网络。通常只有较大网络的一些节点参与覆盖子网络，并且只有这些节点之间的一些物理或虚拟"链接"是覆盖子网络的一部分。

这样，如果将网络表示为图（在像ADNL 这样的数据包网络的情况下可能是一个完整的图，其中任何节点都可以轻松地与其他节点通 信）， 则覆盖子网络是此图的子图（subgraph）。

在大多数情况下，会使用大型网络的某些网络协议，来实现覆盖网络。它可以使用与大型网络相同的地址，或使用自定义地址。

3.3.2 TON的覆盖网络

TON 中的覆盖网络基于在3.1中讨论的ADNL 协议构建；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中的一个，作为其在覆盖网络中的地址。

与ADNL 相反，TON 覆盖网络通常不支持向任意其他节点发送数据包。相反，在某些节点之间建立一些"半永久链接"（被称为覆盖网络的"邻居"），并且消息通常沿这些链接转发（即，从一个节点到它的其中一个邻居借点呢）。通过这种方式，TON 覆盖网络是ADNL 网络（完整）图内的（通常不完整的）子图。

TON 覆盖网络中的邻近链路可以使用专用的点对点ADNL 通道来实现（参考3.1.5）。 覆盖网路的每个节点都维护一个邻居列表（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，用于通信的ADNL通道）。

3.3.3 私有和公共覆盖网络

有些覆盖网络是公共的，意味着任何节点都可以随意加入。另一些是私有的，意味着只有某些节点可以被允许（例如，那些可以证明他们作为验证节点的身份的节点。）一些私有覆盖网络甚至可能对"一般公众" 来说是未知的。这些覆盖网络的信息只提供给某些受信任的节点；例如，它可以使用公钥加密，并且只有拥有相应私钥副本的节点才能解密此信息。

3.3.4 中心化和去中心化覆盖网络

有些覆盖网络是中心化控制的，由一个或几个节点或某个众所周知的公钥的所有者控制。其他则是去中心化的，意味着没有特定节点负责它们。

3.3.5 加入一个覆盖网络

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 位的网络标识符，通常等于覆盖网络的描述的sha256，即一个TL-serialized 对象（参考2.2 .5），这可能包含覆盖网络的公钥和可能的抽象地址（或者，抽象地址可能储存在DHT 中，如3.2.12所解释的）、覆盖网络的名称对应的字符串、如果这是与该分片相关的覆盖网络，还有带有TON区块链分片的标识符，等等。

有时从网络标识符开始可以恢复覆盖网络描述，只需在TON DHT 中查找它即可。在其他情况下（例如，对于private 覆盖网络），必须与网络标识符一起获得网络描述。

3.3.6 定位覆盖网络中的一个成员

在一个节点获得它想要加入的覆盖网络的网络标识符和网络描述之后，它必须定位属于该网络的至少一个节点。

这也适用于不想加入覆盖网络，但只是想与其通信的节点；例如，可能有一个专用于为特定分片链 收集和传播候选交易的覆盖网络， 客户端可能想连接到此网络的任何节点以提议一个交易。

用于定位覆盖网络成员的方法，在该网络的描述中定义。有时（尤其是对于私有网络），必须已经知道一个成员节点才能加入。在其他情况下，某些节点的抽象地址包含在网络描述中。一种更灵活的方法是在网络描述中不包含节点抽象地址，只指示负责该网络的中央机构，然后抽象地址可以通过某些由该中央权威签名的DHT 表的key值获得。

最后，真正的去中心化公共的覆盖网络可以使用在3.2.10描述的"分布式流追踪器"机制实现，也使用TON DHT来实现。

3.3.7 定位覆盖网络中的更多成员，创建链接

一旦找到覆盖网络的一个节点，可以向该节点发送一个特殊的查询，要求提供其他成员的列表，例如，被查询节点的邻居节点或随机选择的节点。

这使得加入的成员，通过选择一些新获得的网络节点，并与它们建立连接，能够获得的消息补充节点的覆盖网络 "邻居列表"，（即，专用的ADNL 点对点通道，如3.3.2中所述）。之后，向所有邻居节点发送特殊消息，表示新成员已准备好在覆盖网络中工作。邻

居节点会将它到新成员的连接，补充到自己的“邻居列表”中。

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，或至少是到它们的连接（通道），可能停止响应； 在这种情况下，这些连接必须被标记为“暂停”，必须尝试重新连接到这些邻居，如果这些尝试失败，则必须销毁这些连接。

另一方面，每个节点有时会从随机选择的邻居处，请求它的邻居列表（或其随机选择），并使用获得的信息部分地更新自己的邻居列表，例如：添加一些新发现的节点，移除一些旧的节点。添加或移除节点的依据可以是随机的，或者根据其响应时间和数据包丢失统计数据。

3.3.9 覆盖网络是一个随机子图

这样，覆盖网络在ADNL 网络内部成为一个随机子图。如果每个顶点的度至少为三（即，如果每个节点至少连接到三个邻居），我们可以说这个随机图以接近一的概率是已链接的（connected）。更确切地说，具有n个顶点的随机图是失去连接（disconnected）的概率是指数级小的，如果例如$n \geq 20$，这个概率可以完全忽略。（当然， 对于全局网络分区的情况不适用，因为不同分区的的节点没有机会互相了解。）另一方面，如果n 小于20，只需要求每个顶点至少有，比如说，至少十个邻居，那就已经足够了。（译者注：这段内容主要表达的是覆盖网络与全局网络的节点连接越多，则表示连接程度约紧密，如果覆盖网络的每个节点只有很少的节点与全局网络的节点相连接，那表示这个覆盖网络快与全局网络失去连接了。）

3.3.10 Ton覆盖网络为降低延迟而做的优化

TON 覆盖网络会按照以下方法，优化前一方法生成的"随机"网络图。每个节点都尝试保留至少三个最小往返时间的邻居节点，并很少更改这个"快速邻居"列表。同时，它还具有至少其他三个完全随机选择的"慢邻居"节点，以使覆盖网络图始终包含一个随机子图。这是为了保持连接性并防止覆盖网络分裂为几个未连接的区域子网络。另外还选择和保留至少三个"中间速度的邻居"节点，即具有中间速度的往返时间，这个所谓的"中间速度"以某个常数为界进行定义（即，快邻居节点和慢邻居节点的往返时间的函数界定的中间往返时间）。

这样，覆盖网络的图仍然保持足够的随机性以保持连接，但是经过优化后可以实现更低的延迟和更高的吞吐量。

3.3.11 覆盖网络中的八卦协议（Gossip protocol）

在覆盖网络中，经常用来执行所谓的八卦协议（gossip protocols）（译者注：八卦协议广范应用在各种区块链项目的p2p网络中，无论是联盟链还是公链,参考：https://en.wikipedia.org/wiki/Gossip_protocol），它让每个节点只与其邻居互动，但最终会达到某个全局目标。（译者注：即把消息传递到整个网络。）例如，有一些八卦协议用于构建一个(不太大) 覆盖网络的所有成员的大致列表，或者只用每个节点有限的内存，预估(任意大) 覆盖网络的总成员数量（参考[15] , [4.4.3] 或[1] 了解详情)。

3.3.12 将覆盖网络作为一个广播域

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，分别针对不同的使用场景进行了优化。其中最简单的一种就是：接收新的广播消息，并将消息转发给所有未接收过该消息的邻居节点。

3.3.13 更复杂的广播协议

某些应用程序可能需要更为复杂的广播协议。例如，对于广播大数据量的消息，发送收到的消息的hash（或新消息的hash 集合），而不是消息本身给邻居节点，是更合理的。在获得未见过的消息hash后，邻居节点可以请求消息体本身，例如，使用在3.1.9中讨论的可靠的大数据包协议(RLDP) 进行传输。这样，只会从一个邻居节点下载新消息。

3.3.14 检查覆盖网络的连接性

如果覆盖网络中有一个必须长期存在的已知的节点（例如，覆盖网络的“拥有者”或“创建者”），为了确保该节点正常到覆盖网络，我们可以检查覆盖网络的连接性。然后，该节点可以不时地广播包含当前时间、序列号和其签名的短消息。任何其他节点如果在不久之前收到过这样的广播消息，就可以确定它仍然连接到覆盖网络。此协议可以扩展到多个已知节点的情况；例如，它们都会发送这样的广播，为确保连接性，所有其他节点需要从超过一半的已知节点那里接收到这样的广播。

在用于传播特定分片链 的新区块（或仅新区块头）的覆盖网络的情况下，一个节点检查连接性的好方法是，追踪目前为止收到的最新区块。因为一个区块通常每五秒生成一次，如果超过，比如，三十秒都没有收到新的区块，那么节点可能已经从覆盖网络中断开了。

3.3.15 流式广播协议

最后，TON 覆盖网络中还有一个流式广播协议（ streaming broadcast protocol），例如，可以用于在某些分片链的验证节点之间传播候选区块（“分片链 任务组”），当然，为此目的还专门创建了一个私有覆盖网络。相同的协议可以用来将新的分片链区块传播给该分片链的所有全节点。

此协议已在2.6.10中描述过：新的（大）广播消息被分成，比如说，N 个一千字节的片段；这些片段的序列通过像Reed-Solomon或一个喷泉码（例如，RaptorQ码[9] [14]）这样的消除码扩展到M ≥ N 的片段，并且这些M 片段按升序的片段号码顺序流向（传播给）所有邻居节点。参与的节点会收集这些片段，直到它们可以恢复回原始的大消息（为此，必须成功接收至少N 个片段），然后指示其邻居节点停止发送流的新片段，因为现在这些节点已经拥有原始消息的副本，可以自己生成后续的片段。这些节点继续生成流的后续片段，并将它们发送给它们的邻居节点，除非邻居节点反过来指示不再需要这样做。

这样，节点在进一步传播它之前，不需要完整地下载一个大消息。这最大限度地减少了广播延迟，尤其是当与4.2.10中描述的优化相结合时。（译者注：本质上，这有点类似P2P下载文件的原理。）

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一个覆盖网络。因为有可能一个或几个已存在的覆盖网络，预计构建新的覆盖网络的成员会与这些已存在的覆盖网络的成员显著重叠。

一个重要的例子出现在TON 分片链被分成两个，或两个同级的分片链合并为一个（参见2.7）。在第一种情况下，必须为每一个新的分片链 构建用于向全节点传播新区块的覆盖网络；但是，可以预期这些新的覆盖网络中的每一个都包含在原始分片链 的区块传播网络中（并包含大约一半的成员）。在第二种情况下，合并分片链 的新区块的传播的覆盖网络，将大致由与正在合并的两个同级分片链的覆盖网络的成员组成。

在这些情况下，新的覆盖网络的描述可能包含对与一系列相关的现有覆盖网络的显式或隐式的引用。希望加入新的覆盖网络的节点，可以检查它是否已经是这些现有网络中的一个的成员，并询问这些网络中的邻居节点是否也对新网络感兴趣。在得到肯定答案的情况下，可以建立一个到这些邻居节点的新的点对点通道，并且它们可以被包含在新的覆盖网络的邻居列表中。

这种机制并不完全取代在3.3.6和3.3.7中描述的一般机制；相反，两者都是并行运行的，并且用于填充邻居列表。这是为了防止新的覆盖网络意外地分裂成几个未连接的子网络。

3.3.17 覆盖网络中的覆盖网络

另一个有趣的案例出现在TON 支付的实现中。（用于即时链下转移资金的闪电网络（“lightning network”）参见5.2）在这种情况下，首先构建包含所有闪电网络的转发节点的覆盖网络。然而，这些节点中的一些已经在区块链中建立了支付通道；除了通过在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 随机邻居借点外，它们在此覆盖网络中必须始终是邻居节点。这些在支付通道里的邻居节点的永久链接，用于运行特定的闪电网络协议，从而在包含的（几乎始终连接的） 覆盖网络内部有效地创建了一个覆盖子网络（如果出错，则不一定连接）。

4 TON服务与应用程序

我们已经详细讨论了TON 区块链和TON 网络技术。现在我们将解释它们如何可以组合起来创建各种服务和应用程序，并讨论TON 项目本身将提供的一些服务，这些服务可能从一开始就有，或者会在以后的时间再提供。

4.1 TON服务实施策略

我们首先讨论如何在TON 生态系统内实施不同的区块链、网络相关应用程序和服务。首先， 一个简单的分类是有必要的：

4.1.1 应用和服务

我们会将"应用程序"（Application）和"服务"（service）这两个词互换使用。但是，事实上其中有一个微妙且有点模糊的区别：一个"应用程序"通常直接向用户提供一些服务，而一个"服务"通常被其他应用程序和服务所利用。例如，TON 存储（Storage） 是一个服务，因为它是设计成为其他应用程序和服务保存文件的，即使用户也可能直接使用它。之外假设的"在区块链中的Facebook" (参见2.9.13)或Telegram消息传递应用，如果通过TON网络实现(即，作为一个"ton-service实施"；参见4.1.6)，则更像是一个"应用程序"，即使一些"机器人"可能在没有人工干预的情况下会自动访问它。

4.1.2 应用的位置:链上，链外或混合

TON 生态系设计的服务或应用程序，需要在某些地方保存和处理数据。这导致了以下应用程序（和服务）的分类：

- 链上（On-chain）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（Off-chain）应用程序(参见5.1.5)：所有数据和处理都在TON 区块链之外。
- 混合（Mixed）应用程序(参见4.1.7)：有一些数据和处理在TON 区块链中；其余的都通过TON 网络提供的链下服务处理。

4.1.3 中心化和去中心化应用程序。

另一个分类标准是应用程序（或服务）是否依赖于中心化的服务器集群，或者真的是分布式”(参见4.1.9)。所有链上的应用程序都自动地是去中心化和分布式的。链下和混合式的应用程序可能呈现出不同程度的集中化。

现在让我们更详细地考虑上述可能性。

4.1.4 纯链上应用程序：Dapps

其中一种可能的方法，如在4.1.2中提到的，将一个「分布式应用程序」（通常缩写为「dapp」）完全部署在TON 区块链中，作为一个智能合约或一组智能合约。所有数据都保存在这些智能合约的永久状态中，并且所有与该项目的交互都将通过发送到或从这些智能合约接收(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限制。它也有其优点： 这样的分布式应用程式不需要服务器来运行或储存其数据（它运行在「区块链中」---即在验证节点的硬件上），并具有区块链极高的(拜占庭) 可靠性和可访问性。这种分布式应用的开发者不需要购买或租用任何硬件；开发者需要做的只是开发一些软件（即智能合约的程序代码）。之后， 开发者从验证节点那里租用计算能力，并用Toncoin支付，支付的费用要么由开发者自己支付，要么由应用的用户承担。

4.1.5 纯网络服务：Ton网站和Ton服务

另一个极端选择是将服务部署在一些服务器上，并通过在3.1中描述的ADNL协议供用户使用，也许还有一些更高级的协议，如在3.1.9中讨论的RLDP，该协议可以用于以任何自定义的格式向服务发送 RPC 查询并获得这些查询的答案。这样，该服务将完全在链外，几乎不使用TON 区块链，但仍然属于TON 网络中的服务。

TON 区块链可能只用于查找抽象地址或服务地址，如在3.2.12中概述的，也许还可以用于像TON DNS 这样的服务（参见4.3.1），来帮助将类似域名的人类可读字符串翻译成抽象地址。

在某种程度上在ADNL 网络（即TON 网络）类似于隐形互联网项目（Invisible Internet Project（I I P） ），这样的（几乎）纯网络服务类似于所谓的「eep-services」（即， 有一个I I P 地址作为它们的入口点的服务，并通过I I P 网络提供服务给客户）。我们会说，驻留在TON 网络中的这些纯网络服务是Ton服务（「ton-services」）。

一个「eep-service」可以实现把HTTP 作为其客户端-服务器协议；在TON 网络上下文中，「Ton-service」可能只是使用RLDP（参见4.0.9）数据报来传输HTTP 查询和响应。如果它使用TON DNS 允许其抽象地址被查找成一个人类可读的域名，那么就几乎完

美可以与传统网站进行类比了。甚至可以写一个专门的浏览器，或一个在用户的机器上本地运行的特殊代理（「ton- proxy」），该代理接受来自用户使用的普通网络浏览器的任意HTTP查询（一旦代理的本地IP 地址和TCP 端口已经配置在浏览器中），并将这些查询通过TON 网络转发到服务的抽象地址。然后用户将拥有类似于世界万维网（WWW）的浏览体验。

在I I P 生态系中，这样的「eep-services」被称为「eep-sites」。在TON 生态系中也可以轻松创建「ton-sites」。这在某种程度上得到了TON DNS这类服务的帮助，它利用TON 区块链和TON DHT 将(TON) 域名翻译成抽象地址。

4.1.6 Telegram作为Ton服务，基于RLDP的MTProto协议

我们想顺便提到MTProto协议（https://core.telegram.org/mtproto ），这个协议主要用于Telegram（https://telegram.org）客户端-服务器交互，事实上，这个协议可以轻松嵌入到在 3.1.9中讨论的RLDP 协议，从而实质上将Telegram 转化为Ton服务。由于TON 代理技术可以为Ton网络或Ton服务 的终端用户透明地开启，并在RLDP 和ADNL 协议的底层协议级别中实施（参见3.1.6），这将使Telegram 实质上不可封锁。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：部分链下，部分链上

一些服务可能使用混合服务的方式： 大部分处理在链外，但也有一些链上的部分（例如，为了登记服务对用户的义务）。通过这种方式，状态的一部分仍然会被保存在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可以由智能合约进行惩罚。

4.1.8 保持文件在链下，TON 存储

TON Storage 提供了这样的服务的一个例子。在其最简单的形式中，它允许用户在链外储存文件，只在链上保留要储存的文件的哈希值，可能还有一个智能合约，一些其他的参与方同意在给定的时间段内，为预先协商好的费用保存该文件。实际上，文件可以被细分为一些小的片段（例如，1 千字节），通过利用擦除程序代码如Reed–Solomon 或喷泉程序代码，可以为这样的块序列构建一个Merkle tree 哈希值，并且这个Merkle tree 哈希值可能在智能合约中发布，而不是或与文件的通常哈希值一起。这有点让人想起文件在torrent中的储存方式。

储存文件的更简单形式完全是在链下：人们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）。这对于需要经常被访问的受欢迎文件实际上可能工作得更好。但是， 用户好不会得到任何可用性保证。例如，之前假设的「区块链Facebook」（参见2.9.13）， 它选择在这种「torrents」中完全保留其用户的个人资料照片， 可能会冒着失去普通（不是特别受欢迎）的用户的照片的风险，或者至少冒着在很长时间内无法显示这些照片的风险。

大部分在存储在链下的TON 储存技术，通过使用在链上的智能合约来确保储存的文件的可用性， 可能更适合这项任务。

4.1.9 去中心化的混合服务：雾服务

到目前为止， 我们已经讨论了中心化的混合服务和应用程序。虽然它们的链上组件是以去中心化和分布式方式进行处理的，并位于区块链中，但它们的链下组件依赖于某些由服务提供商以常见的中心化方式控制的服务器。并且计算能力不是使用一些专用服务器，而是可能会从其中一家大公司提供的云计算服务中租用。 但是，这不会导致服务的的链下组件的去中心化。

实现服务的链下组件的去中心化方法在于创建一个市场，任何拥有所需硬件并愿意出租其计算能力或磁盘空间的人，都可以向需要它们的人提供服务。

例如，可能存在一个注册表（也可以称为「市场」 或「交易所」），所有有兴趣保存其他用户文件的节点都在那里发布他们的联系信息，以及他们的可用储存容量、可用性政策和价格。需要这些服务的人可能会在那里查找它们，如果另一方同意，则在区块链中创建智能合约并上传文件进行链下储存。这样，像TON存储这样的服务就真正成为去中心化的，因为它不需要依赖任何集中的服务器集群来储存文件。

4.1.10 “雾计算”平台

这种去中心化混合应用的另一个例子是当人们想执行一些特定计算（例如，3D 渲染或训练神经网络），通常需要特定且昂贵的硬件。然后拥有这种设备的人可能会通过类似的「交易所」提供他们的服务，需要这种服务的人则会租用它们，并通过智能合约注册双方的义务。这类似于"雾计算（fog computing）"平台,如Golem (https://golem.network)或SONM (https://sonm.io)所提供的。

4.1.11 TON 代理是一个雾服务

TON Proxy提供了一个雾服务的另一个例子，希望提供ADNL 网络流量的隧道服务的

节点（有偿或无偿）会在市场中注册其服务，需要它们的人可能会根据提供的价格、 延迟和带宽选择其中一个节点。之后，人们可能会使用由TON 支付提供的支付通道来处理这些代理服务的小额支付，例如，每传输128 KB数据就收取一笔付款。

4.1.12 TON 支付是一个雾服务

TON 支付 平台(参照 第5章) 也是这种去中心化混合应用的一个例子。

4.2 连接用户和服务提供者

我们在4.1.9中看到「雾服务」(即混合去中心化服务）通常需要一些市场、交易所或注册表，在那里需要特定服务的人可以遇到提供这些服务的人。

这些市场很可能被实现成链上、链下或混合服务，无论是集中式还是分布式的。

4.2.1 连接到TON支付

例如，如果某人想使用TON 支付(参照第5章)，第一步是找到至少一些现有的「闪电网络」 (参照 5.2)的中转 节点，并与它们建立支付通道，如果它们愿意的话。可以通过「包围（encompassing）」的覆盖网络找到一些节点，该网络应该包含所有的闪电网络的中转节点(参照 3.3.17)。但是，这些节点是否愿意创建新的支付通道还不清楚。因此，需要一个注册表，那些准备创建新连接的节点可以在其中发布它们的联系信息（例如，它们的抽象地址）。

4.2.2 上传文件到TON存储

同样，如果用户想将文件上传到TON 储存，那么用户必须找到一些愿意签署智能合约的节点， 这份智能合约将绑定这些节点需要保留该文件的副本（或任何低于某个大小限制的文件）。因此， 需要一个提供储存文件服务的节点注册表。

4.2.3 链上，链下和混合型的注册表

这种服务提供者注册表可能完全在链上实现，借助一个智能合约来在其永久储存中保

留注册表信息。但是，这将是相当缓慢和昂贵的。使用混合方法更为高效，其中相对较小且很少更改的链上注册表只用于指出某些节点（通过它们的抽象地址，或者通过它们的公钥，如在3.2.12中描述的那样，可以用来查找实际的抽象地址），这些节点提供链下（集中式）的注册服务。

最后，去中心化的、纯链下的方法可能包括一个公共的覆盖网络(参照 3.3)，在那里愿意提供他们的服务的人，或者那些寻求购买某人服务的人，只需广播由他们的私钥签名的报价。如果要提供的服务非常简单，甚至可能不需要广播报价：覆盖网络本身的大致成员关系可以用作愿意提供特定服务的人的「注册表」。然后，需要此服务的客户可以定位(参照 3.3.7)并查询此覆盖网络的一些节点，然后查询其邻居，如果已知的节点还不满足其需求的话。

4.2.4 在侧链中创建注册表

另一种实现去中心化混合注册表的方法是创建一个独立的专用区块链--侧链（「side-chain」），由一组自称为验证节点的人维护，他们在链下（on-chain ）的智能合约中发布他们的身份，并为所有感兴趣的参与方提供网络以访问这个专用的区块链，通过专用覆盖网络收集候选交易和广播块更新（参照 3.3）。然后，这个侧链 的任何全节点都可以维护其自己的共享注册表副本（本质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 在工作链中创建注册表

另一个选择是在TON 区块链内创建一个专用的工作链，专门用于创建注册表、市场和交易所。这可能比使用位于基本工作链中的智能合约更有效和更便宜(参照 2.1.11)。但是， 这仍然比在侧链 中维护注册表更昂贵(参照 4.2.4）。

4.3 访问TON服务

我们已经在4.1中讨论了创建TON生态系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「辅助服务」，包括TON DNS和TON存储（Storage）。

4.3.1 TON DNS:一种链上的分层域名服务

TON DNS 是一个预定义的服务，它使用一系列的智能合约来从人类可读的域名映射到ADNL网络节点、TON区块链账户和智能合约的（256 位）地址。

虽然原则上任何人都可以使用TON 区块链实现这样的服务，但拥有这样一个预定义的服务，并具有一个众所周知的界面，当一个应用程序或服务希望将人类可读的标识符转换为地址时，将其用作默认选择是很有用的。

4.3.2 TON DNS 用例

例如，当用户想将一些加密货币转移给另一个用户或商家的时候，可能更喜欢记住该用户或商家账户的TON DNS 域名，而不是手头保留他们的256位账户识别码，并将它们复制粘贴到他们的轻钱包客户端的收件人字段中。

同样地，TON DNS 可以用来查找智能合约的账户识别码、Ton服务和ton网站的入口点（参照 4.1.5），使得一个专用客户端（「ton-browser」）或一个与专用Ton代理扩展或独立应用程序耦合的常见互联网浏览器，能够为用户提供一个类似WWW 的浏览体验。

4.3.3 TON DNS 智能合约

TON DNS 是通过一系列特殊的(DNS) 智能合约来实现的。每一个DNS 智能合约都负责注册某固定域的子域名。 根DNS智能合约，即TON DNS 系统的一级域名所在的位置，位于主链中。所有希望直接访问TON DNS 数据库的软件，都必须将其帐户标识符硬编码在代码中。

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 。这个hashmap 是作为一个binary Patricia 树来实现的，与在2.3.7中描述的类似，但支持可变长度的（位字符串）bitstrings 作为keys。

4.3.4 TON DNS 记录

至于Ton DNS智能合约里hashmap的值， 它们是由TL- scheme 描述的"TON DNS 记录" (参照 2.2.5)。它们包括一个 魔数，其中之一的支持的选项，然后还可能包括一个帐户标识符、或一个智能合约标识符、或一个抽象网络地址(参照 3.1)，或一个用于查找服务的抽象地址的公钥(参照 3.2.12)，或一个覆盖网络的描述，等等。出了根DNS智能合

约，还可以创建子域名的DNS合约：在这种情况下，该智能合约用于解析其域名的子域名。通过这种方式，可以为不同的域创建单独的注册表，由这些域名的所有者控制。

另外这些DNS记录还可以包含一个到期时间、一个缓存时间（通常非常大，因为在区块链中太经常更新值是昂贵的），并且在大多数情况下引用所问子域名的所有者。所有者有权更改此记录（尤其是所有者字段，从而将域名转移给其他人），并延长域名有效期。

4.3.5 注册现有域名的子域名

为了注册一个现有域的新子域，只需向该域的注册者，即智能合约，发送一条消息，该消息包含要注册的子域名（即key） 、一个预定义格式之一的值、所有者的身份、一个到期日期，以及由该域的所有者决定的某个加密货币金额。子域名是按照先到先得 的原则注册的。

4.3.6 从DNS智能合约获取数据

原则上，对于包含DNS 智能合约的主链或分片链 的任何全节点，只要知道智能合约的持久储存中hashmap 的结构和位置，就可以查找该智能合约数据库中的任何子域。

然而，这种方法只适用于某些DNS 智能合约。如果使用的是非标准的DNS 智能合约，它将彻底失效。

因此，我们使用基于通用智能合约接口的合约和get 方法（general smart contract interfaces和get methods）(参照 4.3.11)去获取数据 。任何DNS 智能合约都必须定义一个带有已知签名的"get method"，用于查找一个键（key）。由于这种方法也适用于其他智能合约，尤其是那些提供链上和混合服务的合约，我们在4.3.11中详细解释了它。

4.3.7 解析TON DNS 域名

一旦任何全节点，无论是独立操作还是代表某个轻客户端，都可以在任何DNS 智能合约的数据库中查找条目，就可以从已知的和固定的根DNS 智能合约(帐户) 标识符递归地解析任意的TON DNS 域名。

例如，如果想要解析A.B.C这个域名， 可以在根域数据库中查找键.C、.B.C和A.B.C。如果第一个key C 没有找到，但第二个key B找到了，且它的值是另一个DNS 智能合约的引用，那么就在该智能合约的数据库中查找A，并获得最终值。

4.3.8 轻节点解析TON DNS 域名

通过这样的方式，对于主链的全节点，以及参与域名查找过程的所有分片链，都可

以在不需要外部帮助的情况下将任何域名解析为其当前值。轻节点可能会要求一个全节点代替它执行这一操作，并返回值以及一个Merkle 证明(参照 2.5.11)。这个Merkle 证明使得轻节点可以验证答案是正确的，因此这样的TON DNS响应不能被恶意拦截器伪造，这与通常的DNS 协议形成了鲜明的对比。因为不能期望任何节点都是对所有分片链的全节点，实际的TON DNS 域名解析将涉及这两种策略的组合。

4.3.9 专用Ton DNS 服务器

人们可以提供一个简单的TON DNS 服务器，它会接收RPC 的"DNS" 查询（例如，通过在3.1中描述的ADNL 或RLDP 协议）， 请求服务器解析给定的域名，如果需要，通过将某些子查询转发到其他（全）节点来处理这些查询，并返回对原始查询的答案，如果有需要，还可以增加Merkle 证明。

这样的DNS 服务器可能会使用在4.2中描述的其中一种方法，向任何其他节点， 特别是轻型客户端提供他们的服务（免费或付费）。请注意， 如果这些服务器被认为是TON DNS 服务的一部分，它们将有效地将其从分布式链上服务转化为分布式混合服务（即雾服务）。

以上是我们对TON DNS 服务的简短概述，这是一个可扩展的链上注册表，用于将TON 区块链和TON网络的实体的抽象地址解析成人类可读的域名。

4.3.10 访问智能合约中的数据

我们已经看到，有时需要访问储存在智能合约中的数据，而不改变其状态。

对于智能合约所在分片链的所有全节点，如果知道了智能合约的详细实现细节，则可以从智能合约的持久储存中提取所有所需的信息。然而，这是一种相当不优雅的实现方式，非常依赖于智能合约的实现。

4.3.11 智能合约的get方法

一个更好的方法是在智能合约中定义一些get方法（methods），即某些类型的入站消息，这些消息不会影响智能合约状态，但会生成一个或多个包含get方法结果的输出消息。通过这种方式，只要知道它实现了一个具有已知签名的get方法（即已知要发送的入站消息的格式以及作为结果接收的出站消息的格式），就可以从智能合约中获取数据。

这种方法更加优雅，并符合面向对象编程（OOP）。然而，到目前为止， 它有一个明显的缺陷：必须实际将交易提交到区块链（将get 消息发送到智能合约），等待它被提交并由验证节点处理，从新块中提取答案，并支付Gas费（即在验证节点的硬件上执行get方法）。这是资源的浪费：get 方法无论如何都不会改变智能合约的状态，所以它们不需要在区块链中执行。

4.3.12 尝试执行智能合约的get方法

我们已经提到(参照 2.4.6)任何全节点都可以尝试执行任智能合约的任何方法（即，将任何消息发送给智能合约），从智能合约的给定状态开始，而无需实际提交相应的交易。全节点可以简单地将智能合约的程序代码加载到TON VM 中，从分片链的全局状态（分片链的所有全节点都知道）初始化其持久储存，并使用入站消息作为其输入参数，执行智能合约程序代码。所创建的输出消息将产生此计算的结果。

通过这种方式，任何全节点都可以评估任意智能合约的任意get 方法， 只要它们的签名（即入站和出站消息的格式）是已知的。该节点可以跟踪在此评估期间访问的分片链状态的cell，并为可能要求全节点这样做的轻型节点创建Merkle 证明，证明所执行的计算的有效性（参照 2.5.11）。

4.3.13 使用TL方案描述智能合约接口

回忆一下，由智能合约实现的方法（即它接受的输入消息）基本上是一些TL 序列化对象，这些对象可以由TL-scheme 描述（参照 2.2.5）。所得到的输出消息也可以由相同的TL-scheme 描述。通过这种方式，智能合约提供给其他帐户和智能合约的接口，可以通过TL-scheme 进行正式化。特别是，可以通过这样一个正式化的智能合约接口，描述智能合约支持的get 方法（的子集）。

4.3.14 智能合约的公共接口

注意，一个正式化的智能合约接口，无论 是以 TL-scheme 的形式（表示为 TL 源文件；参照 2.2.5）还是以序列化形式（TL-schemes 可以表示为 TL 序列化形式;参考 https://core.telegram.org/mtproto/TL-tl），都可以发布在区块链中储存的智能合约

帐户描述的特殊字段中，或者单独发布，如果这个接口将被多次引用。在后一种情况下，支持的公共接口的 hash 可能会被合并到智能合约描述中，而不是接口描述本身。

这样一个公共接口的例子是 DNS 智能合约的接口，它应该至少实现一个用于查找子域的标准 get 方法（参照 4.3.6）。在 DNS 智能合约的标准公共接口中，也可以包括用于注册新子域的标准方法。

4.3.15 用户与智能合约的交互

智能合约的公共接口还有其他好处。例如，当用户请求测试一个智能合约时，一个钱包客户端应用程序可以下载合约的接口信息，显示智能合约支持的公共方法（即，可用的操作）列表，如果在正式接口中提供了一些人类可读的注册，也会一起显示出来。在用户选择其中一种方法后，可以根据TL-scheme 自动生成一个表单，用户将被提示选择所需的所有字段，和要附加到此请求的加密货币（例如，Toncoin）的所需金额。提交此表单将创建一个新的区块链交易，其中包含刚刚组成的消息，这个消息将从用户的区块链帐户发送。

通过这种方式，用户将能够通过填写和提交某些表单，以用户友好的方式，从钱包客户端应用程序与任意智能合约进行交互。

4.3.16 用户与Ton服务的交互

事实证明，Ton服务（即，在TON 网络的服务和基于ADNL和RLDP 协议的查询服务； 参照 第3章和 4.1.5），也可能从由TL-schemes 描述（参照 2.2.5）的公共接口中获益。一个客户端应用程序，例如一个轻型钱包或一个“Ton浏览器”，可能会提示用户选择其中一个方法，并使用由接口定义的参数填写一个表单，这与刚刚在4.3.15中讨论的类似。唯一的区别是，结果的TL-serialized消息不作为区块链中的交易提交；相反，它作为一个RPC 查询发送到“Ton服务” 的抽象地址，并根据正式接口（即，一个TL-scheme）解析和显示此查询的响应。

4.3.17 通过Ton DNS查询用户接口

TON DNS 记录，除了可能包含Ton服务或智能合约帐户标识符的抽象地址外，还可能包含该实体的公共（用户）接口的可选字段描述，或几个支持的接口信息。然后，客户端应用程序（无论是钱包、Ton浏览器还是ton代理）将能够下载接口，并以统一的方式与相应的实体（无论是智能合约还是Ton服务）进行互动。

4.3.18 链上和链下服务之间的区别

这样，对于最终用户，链上、链下和混合服务之间的区别将变得模糊（参照4.1.2）。因为用户只需将所需服务的域名输入到用户ton浏览器或钱包的地址栏中，其余的工作都交由客户端应用程序无缝处理。

4.3.19 轻钱包应用和Ton浏览器可内置到Telegram客户端

在此阶段出现了一个有趣的机会。可以将实现上述功能的轻量钱包和TON浏览器嵌入到Telegram智能手机客户端应用程序中，从而将技术带给超过2亿人。用户可以通过在消息中包含TON URIs（参见4.3.22）来发送指向TON 实体和资源的超链接；如果用户选择点击这些超链接，接收方的Telegram 客户端应用程序将在内部打开它，并开始与选定的实体/服务互动。（译者注：这就是后来Telegram发布的可以和Ton区块链结合的Telegram小程序（类似微信小程序）--Telegram Mini APP（TMA））

4.3.20 支付HTTP接口的Ton网站

一个Ton网站简单地说就是一个支持HTTP接口的Ton服务，也许还有其他一些接口。这些支持信息可以存储在相应的TON DNS 记录中。

4.3.21 Ton超链接

请注意，由Ton网站返回的HTML 页面可能包含Ton超链接（ton-hyperlinks）。Ton超链接可以是Ton网站、智能合约或帐户的引用，它是基于特质的URIs方案（参考4.3.22）而生成的，包含抽象网络地址、帐户标识符或人类可读的TON DNS 域名。然后，当用户选择点击这些超链接时，Ton浏览器可能会根据超链接的内容，检测要使用的界面形式，并显示如4.3.15和4.3.16中所述的用户界面。

4.3.22 Ton超链接可附带参数

Ton超链接可能不仅包含所述服务的(TON) DNS 域名或抽象地址，还包含要调用的方法名称以及部分或所有的参数。这种URI 方案可能如下所示：

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

当用户在Ton浏览器中选择点击这样的链接时，要么立即执行该操作（特别是如果它是匿

名调用的智能合约的get方法），要么显示一个部分填充的表格， 由用户确认和提交（对于支付表单可能需要这样做）。

4.3.23 POST操作

Ton网站可以将它返回的HTML页面嵌入到一些常见的POST表格中，POST 操作通过适当的(TON) URLs 引用Ton网站、Ton服务或智能合约。在这种情况下，一旦用户填写并提交了该自定义表单，就会采取相应的操作，无论是立即还是在确认后。

4.3.24 TON WWW

上述所有内容都将导致在TON 网络中建立了一整个相互参照的实体网络，终端用户可以通过ton浏览器访问它，为用户提供类似WWW的浏览体验。对于终端用户来说，这将最终使区块链应用基本上与他们已经习惯的网站相似。

4.3.25 TON WWW的优点

这种包括链上和链下服务的TON WWW，相比传统服务具有一些优势。例如：

- 支付是集成在系统中的。
- 用户身份可以始终呈现给服务（通过自动生成的交易和生成的RPC 请求上的签名），或者随意隐藏。
- 服务不需要检查和重新检查用户凭证；这些凭证在区块链中发布一次即可。
- 可以很容易地通过TON 代理保留用户网络匿名性，并且所有服务都将无法被封锁。
- 由于ton浏览器可以与TON 支付系统集成，所以很容易进行小额支付。

5. TON支付

我们将简短讨论TON项目的最后一个组件，TON 支付，这是用于(微型) 支付通道和基于闪电网络的资金转移平台。它可以使支付「即时」完成，无需执行以下操作，包括：（将所有交易提交到区块链，支付相关的交易费用（例如，消耗的gas），并等待五秒钟，直到包含所涉交易的区块被确认）。

这样的即时支付的总体开销非常小，所以可以用于微支付。例如，TON 文件储存服务可能会向用户收取每下载128 KB 数据的费用，或者需要付费使用的TON 代理可能需要对每128 KB 的中继流量收取微小的费用。

虽然TON 支付可能会比TON 项目的核心组件较迟发布，但需要从一开始就进行一些考虑。例如，用于执行TON 区块链智能合约的程序代码的TON 虚拟机(TON VM; 参照 2.1.20)，必须支持一些具有Merkle 证明的特殊操作。如果这样的支持在原始设计中不存在，那么在后期加入可能会变得有问题(参照 2.8.16)。但是，我们会看到，TON VM 本身就支持「智能」支付通道(参照 5.1.9)。

5.1 支付通道

我们首先讨论点对点支付通道，以及它们如何在TON 区块链中实施。

5.1.1 支付通道的思路

假设有两个参与者，A 和B，他们知道在未来他们需要彼此进行大量的支付。他们不是将每笔付款作为一笔交易提交到区块链，而是创建一个共享的资金池「money pool」（或者也许是一个只有两个帐户的小型私有银行），并向其中投入一些资金：A 投入a个币，B 投入b 个币。这是通过在区块链中创建一个特殊的智能合约，并向其发送资金来实现的。

在创建资金池之前，双方会同意某个特定的协议。他们将跟踪资金池的状态，即他们在共享池中的余额。原始的状态是(a, b)，意味着a 个币实际上属于A，b 个币属于B。然后，如果A 想支付d 个币给B，他们可以简单地同意新的状态是(a′，b′) = (a−d, b+d)。之后，如果，例如，B 想支付d′个币给A，那么状态将变成,(a′′, b′′) = (a′+ d′, b′−d′)，依此类推。

所有这些在资金池内部的余额更新都是完全在链下完成的。当双方决定从资金池中提

取他们应得的资金时，他们会根据资金池的最终状态做相应的操作：首先，向智能合约发送一条特殊的消息，该消息包含经双方商定的最终状态(a *，b *）和双方的签名。然后，智能合约向A发送a *个币，向B发送b *个币，并自毁。

这个智能合约，以及A 和B 用于更新资金池状态的网络协议，是一个简单的在A和B之间的支付通道。根据在4.1.2中描述的分类，它是一个混合服务：它的部分状态位于区块链中（智能合约），但大部分的状态更新是链下的（由网络协议完成）。如果一切顺利，双方将能够互相支付他们想要的金额（唯一的限制是通道的「容量」不能超出资金池的上上限， 也就是说，他们在支付通道中的余额都需要保持非负），并且只需要在区块链中提交两笔交易：一笔用于打开（创建）支付通道（智能合约），另一笔用于关闭（销毁）它。

5.1.2 无需信任的支付通道

先前的例子有些不切实际，因为它假设两个参与者都愿意合作，且永远不会为了某些优势而作弊。例如，想像A 选择不签署最终的余额(a ′, b′)，其中a′ 为< a。这将使B 陷入困境。

了防止这种情况，人们通常试图开发无需信任（trustless）的支付通道协议，这不需要参与者互相信任，且对试图作弊的任何一方进行处罚。

这通常是在签名的帮助下达到的。支付通道智能合约知道A 和B 的公钥，且在需要时可以检查其签名。支付通道协议要求参与者签署中间状态并互相发送签名。然后，如果参与者之一作弊，例如，假装支付通道的某个状态从未存在，可以通过显示该状态的签名来证明其不正当行为。支付通道智能合约充当链上仲裁者（“on-chain arbiter”），能够处理两个参与者关于彼此的投诉，并通过没收有罪一方的所有资金并授予给另一方来惩罚有罪一方。

5.1.3 简单的双向同步无需信任的支付通道

考虑以下更为真实的例子：让支付通道的状态由三元组(δ_i , i, o_i)描述，其中i 是状态的序列号（最初为零， 然后当后续状态出现时增加一），δ_i是channel imbalance（意味着A 和B 分别拥有a + δ_i和b − δ_i的硬币），且o_i是被允许生成下一个状态的参与者（A 或B）。每个状态必须在进一步进展之前由A 和B 签名。

现在，如果A 想要在支付通道内部将d 个硬币转移到B，且当前状态为S_i = (δ_i , i, o_i)，其中o_i = A，那么它只需创建一个新状态S_{i+1} = (δ_i − d, i + 1, o_{i+1})，签名并将其及其签名发送给B。然后，B 通过签名并发送其签名的副本给A来确认。之后，双方都有一

个带有他们两者签名的新状态的副本，且可能会发生新的转账。

如果A 在o_i= B 的S_i状态下想要将硬币转移到B，那么它首先要求B提交具有相同的不平衡（imbalance）δ_{i+1}= δ_i，但o_{i+1}= A的后续状态S_{i+1}。之后，A 将能够进行转账。

当两个参与者同意关闭支付通道时，他们都在他们认为是最后的状态S_k上放置他们特殊的最终签名，并通过将最终状态及两个最终签名发送给支付通道智能合约，来调用合约的清理（clean）或包含两边的销毁方法（two -sided finalization method）。

如果另一方不同意提供其最终签名，或者仅仅是停止回应，则可以单方面关闭该通道。为此，希望这么做的一方将调用单方的销毁（unilateral finalization）方法，将其对于最终状态的版本、其最终签名，以及另一方最近的签名状态，发送到智能合约。在此之后，智能合约不会立即根据接收到的最终状态采取行动。相反，它会等待一段特定的时间（例如，一天），让另一方提交其对于最终状态的版本。当另一方提交其版本并且与已提交版本相容时，智能合约会计算真正的最终状态并相应地分配资金。如果另一方未能向智能合约提交其对于最终状态的版本，那么资金将根据提交的最终状态的唯一副本重新分配。

如果两方之一作弊，例如，签署两个不同的最终的状态，或者签署两个不同的下一个状态S_{i+1}和S'_{i+1}，或者签署一个无效的新状态S_{i+1}（例如，不平衡（imbalance）δ_{i+1}< −a 或> b），那么另一方可以将此行为的证据提交给智能合约的第三种方法。这个方法将会令有罪的一方将立即被处罚，并完全失去其在支付通道中的份额。

这个简单的支付通道协议是公平的，意味着任何一方都可以总是获得其应得的，无论另一方是否合作，并且如果试图欺骗，很可能失去其承诺给支付通道的所有资金。

5.1.4 同步支付通道：一个带有两个验证者的简单虚拟区块链

上面的简单同步支付通道的示例可以这样重新描述。想像一下，状态序列S0、S1、…、Sn实际上是一个非常简单区块链的区块序列。这个区块链的每个区块基本上只包含区块链的当前状态，也许还有对前一个区块（即其hash）的引用。 A 和B 两方都作为此区块链的验证节点， 所以每个区块必须收集他们两人的签名。区块链的状态S_i定义了下一个区块的指定生产者o_i，所以A 和B 在生产下一个区块时没有竞争。生产者A 只能创建将资金从A 转移到B 的区块（减少 不平衡（imbalance）：$\delta_{i+1} \leq \delta_i$），B 则只能将资金从B 转移到A（增加δ）。

如果两个验证节点同意区块链的最后一个区块（和最终状态），则可以通过收集两方的特殊"最终"签名，并与最终区块一起提交到通道智能合约进行处理和相应地重新分配资金。

如果验证节点签署一个无效的区块，或者创建一个分叉，或者签署两个不同的最终区

块，可以通过将其不端行为的证据提交给智能合约，该合约充当两个验证者的 链上仲裁器（"on-chain arbiter "），然后，违规方将失去保留在支付通道中的所有资金，这类似于失去其质押的验证节点。

5.1.5 异步支付通道：一个带有两个工作链的虚拟区块链

先前讨论的同步支付通道在5.1.3中有某些缺点：在前一笔交易被对方确认之前，不能开始下一笔交易（支付通道内的资金转移）。这可以通过将4.1.4中讨论的单一虚拟区块链替换为两个互动的虚拟工作链（或分片链）来解决。

第一个工作链只包含A 的交易，且其区块只能由A 生成；它的状态是 S_i = (i, φ_i, j, ψ_j)，其中i 是区块号（即A 到目前为止执行的交易或资金转移的计数）、φ_i是到目前为止从A 转移到B的总额、j 是A 知道的B的区块链中最近有效的区块号，而ψ_j是在其 j 次交易中从B 转移到A 的金额。 B 放在其第 j 个区块上的签名也应该是此状态的一部分。另外还可能包含这个工作链的前一个区块和另一个工作链的第 j 个区块的hash。S_i的有效性条件包括 $\varphi_i \geq 0$；如果 i > 0， $\varphi_i \geq \varphi_{i-1}$； $\psi_j \geq 0$； $-a \leq \psi_j - \varphi_i \leq b$。

相似地，第二个工作链只包含B 的交易，且其区块只能由B 生成；它的状态是 T_i = (j, ψ_j, i, φ_i)，具有相似的有效性条件。

现在，如果A 想将一些钱转移给B，它只需在其工作链中创建一个新区块，签名，然后发送给B，无需等待确认。

支付通道的最终状态的确认需要：带有A 签名（版本的）的它的区块链的最终状态（带有其特殊的「最终签名」）和 带有B 签名的它的区块链的最终状态，然后将这两个最终状态提交给支付通道智能合约的清理（clean）结算方法来完成的。单方面的终结也是可能的，但在那种情况下，智能合约将必须等待另一方提交其最终状态的版本， 至少要有一些宽限期。

5.1.6 单向支付通道

如果只有A 需要向B 支付（例如，B 是服务提供者，A 是其客户），则可以创建一个单方向的支付通道。本质上，它只是在5.1.5中描述的第一个工作链，没有第二个。反之，可以说在5.1.5中描述的异步支付通道由两个单向支付通道或「半通道」组成， 由相同的智能合约管理。

5.1.7 更复杂的支付通道：承诺（Promises）

我们稍后将在5.2.4看到，「闪电网络」（参见5.2）允许通过多个支付通道的链进行

即时资金转帐，这需要涉及的支付通道具有更高的复杂度。

具体来说，我们希望能够承诺「promises」，或者说「带有条件地转移资金」： A同意发送c 个币给B，但只有在满足某个条件时，B 才会得到钱，例如， 如果B可以提供某个字串 u 并满足Hash(u) = v，B才能得到钱，其中v 的值是已知的。否则， A 可以在一段时间后取回钱。

这样的承诺可以简单地在区块链上通过一个简单的智能合约来实现。但是，我们希望在支付通道中，即链下，能够实现承诺（promises） 和其他类型的条件性资金转移，因为它们大大简化了存在于「闪电网络」中的支付通道链的资金转移（参见5.2. 4）。

通过这样的方式，实现在5.1.4和5.1.5中概述的「作为一个简单区块链的支付通道」，将变得很方便。现在，我们考虑一个更复杂的虚拟区块链，其状态包含一组未实现的承诺，且一定数量的资金锁定在这些承诺中。这个区块链——或异步情况下的两个工作链——必须通过它们的hash显式引用前面的区块。然而，总体机制仍然是一样。

5.1.8 复杂的支付通道的智能合约面临的挑战

请注意， 虽然一个复杂的支付通道的最终状态仍然很小，而且最终的「清理」方法很简单（如果双方都同意他们的应付金额，且双方都已签署他们的协议，那么就没有其他事情要做），但是单方面终结的方法和惩罚欺诈行为的方法需要更为复杂。实际上，它们必须能够接受不当行为的Merkle证明，并检查支付通道区块链的交易是否已被正确处理。

换句话说，支付通道的智能合约必须能够使用Merkle 证明，检查它们的「hash 有效性」，对于支付通道的（虚拟）区块链，并且必须包含检查交易（ev_trans）和检查区块（ev_block）这样的功能实现（参见2.2.6），。

5.19 Ton VM支持“智能”支付通道

TON VM， 用于运行TON 区块链智能合约的程序代码，能够应对执行"智能"或复杂的支付通道所需的智能合约（参见5.1.8） 。

在这一点上，「一切皆为bag of cells」的模式（参见2.5.14）变得非常方便。由于所有的区块（包括临时支付通道区块链的区块）都被表示为bag of cells （并由一些代数数据类型描述），对于消息和Merkle 证明也是如此， Merkle证明可以轻易地嵌入到发送到支付通道智能合约的入站消息中。 Merkle 证明的「hash 条件」将自动被检查，当智能合约访问收到的Merkle 证明时，它将处理这个Merkle 证明，把这个证明当成使用相应代数数据类型的值去处理---尽管它是不完整的，树的一些子树被替换为包含省略子树的Merkle hash 的特殊节点。

智能合约将继续使用这个证明，这个证明要证明的内容可能代表支付通道（虚拟）区

块链的一个区块和其状态，接着执行ev_block函数（参见2.2.6），这个函数将会评估和验证此区块和之前的状态。然后，计算要么完成，并且最终状态可以与区块中的声称的一样，要么在尝试访问缺失的子树时抛出缺失节点「absent node」异常，表示Merkle 证明是无效的。

以这种方式，使用TON 区块链智能合约实现智能支付通道的验证，将变得很直接。可以说，TON虚拟机内建了检查其他简单区块链有效性的支持。唯一的限制因素是要合并到智能合约的入站消息中（即交易）的Merkle 证明的大小。（译者注：即这个Merkle 证明的大小不能太大）

5.1.10 在已存在的支付通道创建简单的支付通道

我们想讨论在现有支付通道内创建一个简单的（同步或非同步）支付通道的可能性。

虽然这可能看起来有点复杂， 但它与5.1.7中讨论的「promises」相比，理解和实现并不困难。本质上，A承诺 根据一些其他（虚拟）支付通道区块链的最终结算，则向B 支付最多c 个硬币，而不是：A承诺如果提供了一些hash问题的解决方案，则支付c个硬币给另一方。一般来说，这个其他的支付通道区块链，甚至不需要在A 和B 之间；它可能涉及一些其他的参与者，例如C 和D，分别希望将c 和d 个硬币承诺到他们的简单支付通道中。 （这个可能性在5.2.5中后来被利用。）

如果包围（encompassing）的支付通道是不对称的，那么需要在两个工作链中提交承诺两个承诺：如果「内部」简单支付通道的最终结算产生了一个与$0 \le -\delta \le c$ 的负最终不平衡（imbalance）值δ ，A 将承诺支付$-\delta$ 个硬币给B；如果δ 是正的，B 将不得不承诺支付δ 个硬币给A。另一方面，如果「包围」的支付通道是对称的， 那么可以通过A 提交一个带有参数(c, d) 的「简单支付通道创建」交易到单一支付通道区块链（这将冻结属于A 的c个硬币），然后由B 提交一个特殊的确认交易」（这将冻结属于B 的d 个硬币）。

我们希望内部支付通道非常简单（例如，在6.1.3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。外部支付通道将必须是5.1.7中描述的那种「智能」支付通道。

5.2 支付通道网络，或「闪电网络」

现在，我们准备讨论TON 支付的「闪电网络」，该网络使得任何两个参与节点之间能够实时进行资金转账。

5.2.1 支付通道的限制

一个支付通道对于那些希望在他们之间有大量的金钱转账的双方是有用的。但是，如果一个人只需要向特定的接收者转账一两次，则与这个人建立支付通道将是不划算的。除其他外，这将意味着在支付通道中冻结大量的金钱，而且还需要至少两次区块链交易。

5.2.2 支付通道网络

支付通道网络通过使金钱沿着链的支付通道进行转账，来克服单一支付通道的局限性。如果A 想要向E 转账，A不需要与E 建立支付通道。只需要拥有一条通过几个中间节点，且能够连接A 和E 的支付通道链就足够了。比如说，四个支付通道：从A 到B，从B到C，从C 到D，以及从D 到E。

5.2.3 支付通道网络概述

回想一下，一个支付通道网络，也被称为「闪电网络」，由一系列参与节点组成，其中一些节点之间建立了长时间存在的支付通道。我们稍后将看到，这些支付通道必须是5.1.7意义上的「智能」支付通道。当一个参与节点A 想要向任何其他参与节点E 转账时，它会尝试在支付通道网络内找到连接A 和E 的路径。当找到这样的路径时，她沿着这条路径进行「链式资金转账」（chain money transfer）。

5.2.4 链式资金转账

假设存在一条从A 到B，从B 到C，从C 到D，以及从D 到E 的支付通道链。进一步假设A 想要转账x 个币给E。

一个简单的方法是沿着现有的支付通道向B 转账x 个币，并要求B将钱进一步转发给C。但是，为什么B 不简单地为自己取走钱呢？因此，必须采用一种更复杂的方法，不需要所有参与方互相信任。

具体实现如下所示。A 生成一个大的随机数u ，并计算其hash 值v = Hash(u)。然后在与B 的支付通道内创建一个承诺，如果出示一个带有hash 值v 的数字u，A将支付x 个硬币给B（参见5.1.7）。这个承诺包含v，但不包含u，所以这仍然是保密的。

之后，B 在他们的支付通道中对C 创建了一个类似的承诺。B不害怕给出这样的承诺，因为他知道A 给了他一个类似的承诺存在。如果C 曾经出示hash问题的解决方案来收集B 承诺的x 个币，那么B 将立即将此解决方案提交给A，以收集来自A 的x 个币。

然后创建C 对D 和D 对E的类似承诺。当所有承诺都到位时，A 通过将解决方案 u

通告给所有参与方来触发转账---或仅通告给E。

这个描述中省略了一些次要的细节。例如，这些承诺必须有不同的到期时间，且沿着链上承诺的金额可能会略有不同（B 可能只承诺x-ε 个硬币给C，其中 ε 是一个事先约定的小中转费）。我们暂时忽略这些细节，因为它们对于理解支付通道如何工作，以及它们如何在TON中实现并不太相关。

5.2.5 支付通道链中的虚拟支付通道

考虑以下情境：假设A 和E 预计会互相进行大量的付款。他们可以在区块链之间创建一个新的支付通道(payment channel)，但这仍然会相当昂贵，因为某些资金会被锁定在这个支付通道中。另一种选择是对于每笔付款使用在5.2.4中描述的链式资金转移(chain money transfers)。但是，这会涉及大量的网络活动，以及所有相关支付通道的大量区块链交易。

另一种方法是在连接A 至E 的支付通道网络中建立一个虚拟支付通道(virtual payment channel)。为此，A 和E 为他们的付款创建一个(虚拟的) 区块链，就像他们要在区块链中创建一个支付通道一样。但是，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---连接A至B、B至C等的通道---在其中创建简单的支付通道，并绑定到A和E创建的虚拟区块链(参考5.1.10)。换句话说，现在每个中间支付通道内，都存在根据A和E之间的最终结算来转移资金的承诺。

如果虚拟支付通道是单向的，那么可以很容易地实施这些承诺，因为最终的失衡值（imbalance）δ 将是非正值，因此可以按照在5.2.4中描述的相同顺序，在中间支付通道中创建简单的支付通道。他们的到期时间也可以以相同的方式设定。

如果虚拟支付通道是双向的，情况就稍微复杂了。在这种情况下，应该将根据最终结算转移δ个硬币的承诺分成两个 半承诺(half-promises)，如5.1.10中所解释的那样：转移 $\delta^- = \max(0, -\delta)$ 个币到前进方向，和转移 $\delta^+ = \max(0, \delta)$ 到反向方向。这些半承诺可以在中间支付通道中独立创建，一个半承诺链在从A到E 的方向，另一个半承诺链在相反的方向。

5.2.6 在闪电网络中寻找路径

关于在支付网络(pay-ment network) 中如何找到连接A 和E 的路径，这一点到目前为止还没有讨论。如果支付网络不是太大，可以使用类似OSPF的协议：所有支付网络的节点创建一个覆盖网络(overlay network)（参考4.2.17），然后每个节点都通过gossip 协议将所有可用链接（即，参与的支付通道）信息传播给其邻居。最终，所有节点都将拥有支付网络中所有参与的支付通道的完整列表，并且将能够自己找到最短的路径---例如，

使用一个修改过的Dijkstra算法版本，以考虑到涉及的支付通道的容量（“容量”表示可以沿着它们传输的最大金额）。一旦找到了一个候选路径，它可以被一个特殊的ADNL数据包探测，该数据包包含完整的路径，并要求每个中间节点确认所涉及的支付通道的存在，并根据路径将此数据包进一步转发。之后，可以构建一条链，并可以进行链式资金转移（参考5.2.4），或在支付通道链中创建虚拟支付通道的协议（参考5.2.5）。

5.2.7 优化

在此处可以进行一些优化。例如，只有闪电网络(lightning network) 的中继节点，需要参与在5.2.6中讨论的类似OSPF 的协议。两个“叶子”节点希望通过闪电网络连接，将他们连接到的中继节点列表（即，他们已建立的参与支付网络的支付通道）通告给对方。然后，可以如上所述（5.2.6），检查从一个列表中的中继节点到另一个列表中的中继节点的路径。

5.2.8 结论

我们已经概述了如何使用TON 项目的区块链和网络技术来创建TON支付，这是一个用于链外即时资金转账和小额支付的平台。对于在TON 生态系中的服务，这个平台可能非常有用，允许它们在需要的时候和地方轻松收集小额支付。

结论

我们已提出一个可扩展的多区块链架构，可以支持大量流行的加密货币和具有用户友好界面的去中心化应用程序。

为了达到必要的可扩展性，我们提出了TON 区块链，一个紧密耦合的多区块链系统（参见 2.8.14），该系统采用了自下而上的分片方法（参见2.8. 12和2.1.2）。为了进一步提高潜在性能，我们引入了2维区块链机制来替换无效的区块（参见2.1.17）以及超立方体路由以实现分片之间的更快通讯（参见2.4.20）。与现有和提议的区块链项目的简短比较后（参见2.8和2.9），突显了这种方法对于寻求每秒处理百万交易的系统的好处。、

在第3章 中描述的TON网络，满足了所提议的多区块链基础设施的网络需求。此网络组件还可与区块链结合使用，以创建一系列仅使用区块链无法实现的应用程序和服务（参见2.9.13）。在第4章中讨论的这些服务包括TON DNS，一个用于将人类可读的识别对象翻译成其地址的服务； TON 存储（Storage），一个用于储存任意文件的分布式平台；TON 代理（Proxy），一个用于匿名化网络访问和访问TON服务的服务；以及TON 支付（参见第5章），这是一个用于在TON 生态系统中进行即时链下资金转移的平台，还可用于小额支付。

TON 基础设施通过专门的轻量客户端钱包、Ton浏览器、桌面和智能手机应用程序，为最终用户提供类似传统浏览器的体验（参见4.3.24），使加密货币支付以及与TON平台上的智能合约和其他服务的互动，对大众用户来说是可访问的。这样的轻量客户端可以集成到Telegram客户端中（参见4.3.19），从而最终将基于区块链的应用程序带给数以亿计的用户。

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.

附录 Ton coin，又名Gram

Toncoin，也被称为Gram (GRM)，它适合TON 区块链的主要加密货币，特别是其主链和基础工作链。它有许多的用途，包括验证节点所需的存款；交易费、gas费支付（即，智能合约消息处理费）和持久储存支付等，通常也以Ton coin 收取。

A.1 细分和术语

一个Ton coin被细分为十亿(10^9）个小的单位，称为nano Ton coin、nTon coin或简单地称为nanos。所有的转账和账户余额都表示为nanos 的非负整数倍。其他单位包括：

- 一个nano、nToncoin或nano Toncoin是最小的单位，等于10^{-9} Ton- coin。
- 一个micro或micro Ton coin等于一千（10^{-3}）nanos。
- 一个milli等于一百万(10^6) nanos，或一个Ton coin 的千分之一（10^{-3}）。
- 一个Toncoin等于十亿(10^9）nanos。
- 一个kilo Ton coin、或kTon coin，等于一千(10^3）Ton coin。
- 一个mega Toncoin、或MToncoin，等于一百万(10^6）Ton coin，或10^{15}nanos。
- 最后，一个giga Toncoin、或GToncoin，等于十亿(10^9）Toncoin，或 10^{18} nanos。

不需要更大的单位，因为Toncoin 的初始供应量将限制为五十亿(5 * 10^9）Toncoin (即，5 Giga Toncoin)。

A.2 用于表示Gas 价格的小单位

如果出现需要更小单位的必要性，将使用等于10^{-16} nanoToncoin 的"specks" 。例如，gas 价格可能以specks 表示。然而，实际要支付的费用，计算为 gas 价格和消耗的gas 量的乘积，将始终四舍五入到最接近的2^{16} specks 的倍数，并表示为nanos 的整数。

A.3 初始供应，挖矿奖励和通货膨胀

Toncoin 的总供应量原本限制为5 Giga Toncoin (即，五十亿的Toncoin 或5 * 10^8 nanos)。

这一供应量将非常缓慢地增加，因为需要奖励给挖掘新的主链和分片链区块的验证节点。这些奖励大约会达到20% (确切的数字可能在未来进行调整) 的验证节点的每年质押的Ton coin，只要验证节点努力履行其职责，签署所有区块，从不离线并且从不签署无效的区块。这样，验证节点将有足够的利润投资更好更快的硬件，以处理不断增加的用户交易量。

在任何给定时刻，验证节点质押的总Ton coin，平均下来，我们预计为最多10%（验证节点质押的最大总量是一个区块链的可配置参数，因此如果有需要，该协议可以强制执行此限制）的Toncoin 总供应量。因此，这将产生一个2% 的每年的通胀率，在35 年内将Toncoin 的总供应量加倍(到十个GigaToncoin)。从本质上讲，这种通胀代表了由社区的所有成员支付给验证节点的款项， 用于保持系统正常运行。另一方面，如果一个验证节点被抓到有不正当行为，其部分或全部的质押将被作为惩罚而扣除，并且其中的较大部分随后将被"销毁"，从而减少Toncoin 的总供应量。这将导致通货紧缩。罚款的一小部分可能会重新分配给验证节点 或提交验证节点不当行为证明的渔夫（"fisherman"）。

A.4 Ton coind的初始价格

第一个要出售的Toncoin 的价格将约等于$0.1 (USD)。每一个接下来要被卖出的Toncoin (由TON 储备，由TON 基金会控制) 都将比前一个价格高十亿分之一。这样，要放入流通的第n个Toncoin大约以以下价格出售：

$$p(n) \approx 0.1 * (1 + 10^{-9})^{n} \text{ USD}, \quad (26)$$

或和其他(加密) 货币的大约相等的金额（由于市场汇率迅速变化），如BTC 或ETH。

A.4.1 指数式定价的加密货币

我们称Toncoin为一种指数式定价的加密货币（exponentially priced cryptocurrency），意思是将要流通的第n个Toncoin 的价格大约为p(n)，由以下公式给出：

$$p(n) = p0 * e^{\alpha n} \quad (27)$$

其中为p0 = 0.1 USD 和α = 10^{-9}。

更精确地说，一旦n 个硬币被放入市场流通，一个新币的小部分dn 价值为p(n) dn 美元。（这里的n 不必然是整数。）

这种加密货币的其他重要参数包括n，流通中的币总数，和N ≥ n，N为总币数。对于Toncoin，N = 5 * 10^{9}。

A.4.2 前n个币的总价

前n个以指数方式定价的，且已进入市场流通的加密货币(例如，Toncoin) 的总价 $T(n) = \int_0^n p(n)\,dn \approx p(0) + p(1) + \cdots + p(n-1)$，可以通过以下方式计算

$$T(n) = p0 * \alpha^{-1}(e^{\alpha n} - 1) \quad (28)$$

A.4.3 接下来Δn 个币的总价

在已经存在n个硬币之后，的Δn 个进入市场流通的币的总价T(n+ Δn) − T(n) 可以通过以下方式计算：

$$T(n + \Delta n) - T(n) = p0 * \alpha^{-1} (e^{\alpha(n+\Delta n)} - e^{\alpha n}) = p(n) * \alpha^{-1} (e^{\alpha \Delta n} - 1) \quad (29)$$

A.4.4 总值T可以购买的币的数量

假设已经有n个币流通，并且有人想花T (美元) 购买新币。新获得的币数量Δn 可以通过将T(n + Δn) − T(n) = T 放入(29)中来计算， 得到

$$\Delta n = \alpha^{-1} \log (1 + T * \alpha / p(n)) . \quad (30)$$

当然，如果T ≪ p(n) α^{-1} ，那么Δn ≈ T/p(n)。

A.4.5 Ton coin的市场价格

当然，如果自由市场价格跌破p(n) := 0.1 * $(1 + 10^{-9})^{n}$，一旦n 个Toncoin 流通，没有人会从TON Reserve 购买新的Toncoin ；他们会选择在自由市场上购买Toncoin，而不增加Toncoin 的总流通量。另一方面，一个Toncoin 的市场价格不能比p(n)高得多，否则从TON Reserve 获得新的Toncoin 就变得更加划算的。这意味着Toncoin 的市场价格不会受到突如其来的剧烈波动(和下跌)；这很重要，因为质押(验证节点的存款) 至少被冻结了一个月，而且gas 价格也不能改变得太快。所以，系统的整体经济稳定性需要一个机制，防止Toncoin 的汇率发生过于剧烈的变化，就像上面描述的那样。

A.4.6 回购Ton coin

当总共有n个Toncoin 流通(即，不保留在TON Reserve 控制的特殊帐户上)，如果Toncoin 的市场价格跌到0.5*p(n) 以下，TON Reserve 保留购回一些 Toncoin 并减少n，即Toncoin 的总流通量的权利。这是为了防止Toncoin 汇率突然下跌。

A.4.7 以更高价格售卖Ton coin

TON Reserve 将根据价格公式(26) 最多卖出一半的 (即，2.5 * 10^9 个Toncoin) 的Toncoin 总供应量。它保留不销售任何剩余的Toncoin的权利，或以高于p(n) 的价格出售它们，但永远不会以较低的价格出售(考虑到汇率快速变化的不确定性)。这里的道理是，一旦至少一半的Toncoin都被卖出，Toncoin 市场的总价值将足够高， 外部力量操纵汇率比在Toncoin 部署初期会变的更困难。

A.5 未分配的Ton coin的使用

TON Reserve 的大部分的未分配Toncoin(大约 5 * 10^9 − n 个Toncoin) (即，那些存在于TON Reserve 的特殊帐户和与之明确关联的其他帐户中的Toncoin，只作为验证节点的质押 (因为TON 基金会本身可能必须在TON区块链的第一个部署阶段提供大部分的验证节点)，和在主链上投票支持或反对有关可配置参数和其他协议更改的提案。这些Ton coin会由TON基金会决定如何分配(即，其创建者---开发团队)。这也意味着在TON 区块链的第一个部署阶段，TON 基金会将拥有大多数的票。希望那时系统会变得更加成熟，不需要太频繁地调整参数。

A.5.1 一些未分配的Ton coin会给予开发者

一个预定数量的（相对较少的）「未分配」Toncoin（例如，200 MegaToncoin，相当于总供应的4%）将在TON 区块链的部署期间转移到由TON基金会控制的特殊账户，然后某些「奖励」可能会从此账户支付给TON 开源软件的开发者，并有最少两年的授权期。

A.5.2 Ton基金会需要一定的Ton coin用作运营目的

请记住，TON 基金会 将收到由TON Reserve 出售的Toncoin 所得的法定货币和加密货币，并将其用于TON 项目的开发和部署。例如，原始的验证节点集合以及TON 存储和TON 代理节点的初始集合可能由TON 基金会进行安装。

虽然这对于项目的快速启动是必要的，但最终目标是使该项目尽可能去中心化。为此，TON 基金会可能需要鼓励安装第三方验证节点以及TON 存储和TON 代理节点。例如，通过支付Ton coin，以鼓励他们储存TON区块链的旧区块或代理所选服务子集的网络流量。此类付款将以Ton-coin进行；因此，TON 基金会将需要大量的Toncoin 用于正常运营的目的。

A.5.3 从储备金中提取预先安排的金额

TON 基金会将从TON Reserve 转账到其账户的一小部分，例如，所有币的10%（即500 MegaToncoin）在Toncoin的初次销售结束后，用于如在A.5.2中概述的目的。最好是与为TON 开发人员提供的资金转账同时完成，如在A.5.1中提到的。

在转移到 TON 基金会和 TON 开发人员之后，TON Reserve 价格 p(n) 的 Toncoin 将立即上升一定量，这一量是提前知道的。例如，如果所有币的 10%被转移用于 TON 基金会的运营，而 4%被转移以鼓励开发者，则流通中的币总数量 n 将立即增加Δn = 7 * 10^8， Toncoin 的价格乘以 $e^{\alpha \Delta n} = e^{0.7} \approx 2$（即，大约 2 倍）。

剩余的「未分配」Toncoin 将由TON Reserve 使用， 如上面的A.5所解释。如果TON 基金会之后还需要更多的Toncoin，它将简单地将其在币销售期间先前获得的部

分资金转换为Toncoin，无论是在自由市场购买Ton coin，还是从TON Reserve 购买Toncoin。为了防止过度中心化， TON 基金会永远不会使其帐户上拥有超过10%的总Toncoin数量（即500 MegaToncoin）。

A.6 批量销售Ton coin

当很多人同时想从TON Reserve 购买大量Toncoin 时，最好不要立即处理他们的订单，因为这将导致结果非常依赖于特定订单的时机和他们的处理顺序。

相反，购买Toncoin 的订单可能在某个预定的时间段（例如，一天或一个月）内收集，然后一次全部处理。如果收到k 个订单，第i个订单价值Ti美元，则总金额T = $T_1 + T_2 + \cdots + T_k$用于根据(30)购买Δn 个新币，并且第i 个订单的发送者分配了这些币的Δn * T_i/T。这样，所有买家都以每Toncoin T/Δn USD 的相同平均价格获得他们的Toncoin。

之后，开始收集新一轮的购买新Toncoin 的订单。

当购买订单的Ton coin总价值变得足够低时，这种「批量销售」系统可能会被一个根据公式(30)从TON Reserve 直接销售Toncoin 的系统所替代。 「批量销售」机制可能在收集TON Project 投资的初始阶段得到广泛使用。