基于分组的智能合约并行化研究

Research on Parallelization of Grouping-based Smart Contract's Execution

工程领域：计算机技术

作者姓名：王正凯

指导教师：侯庆志 副教授

企业导师：马昊伯 高级工程师

天津大学智能与计算学部

二零一八年十一月

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作和取得的研究成果，除了文中特别加以标注和致谢之处外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得**天津大学**或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文作者签名：王正凯　签字日期：2018 年 12月 10日

学位论文版权使用授权书

本学位论文作者完全了解**天津大学**有关保留、使用学位论文的规定。特授权**天津大学**可以将学位论文的全部或部分内容编入有关数据库进行检索，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：王正凯　导师签名：

签字日期：2018 年 12月 10 日　签字日期：2018 年 12月 10 日

摘　要

区块链的本质是一个所有参与节点共同维护，公开透明的分布式账本，但并非某个单独的用户可以对它随意修改，只有通过共识机制选出的区块链节点才可对其更改。具有去中心化，不可篡改，数据加密等特性的区块链为智能合约提供了一个良好的运行环境。

目前智能合约的执行效率严重阻碍了它在许多领域的应用。智能合约是通过区块链交易来实现调用的，因而提升智能合约的执行效率也即提高区块链交易的处理速度。为了提升交易的处理效率，论文提出了交易分组并行化策略，将非冲突交易并行处理。策略中的智能合约平台是基于 AElf 的，它的智能合约是支持元数据的。在处理交易时，根据元数据对智能合约函数占用的资源进行标记，之后根据资源标记的结果构建资源的并查集，再通过函数（交易）占用的某个资源查询交易所在的并查集子集，根据所在子集的 ID 将交易放入对应分组中，实现所有交易的分组，然后再根据并发度将交易分组结果进行合并，最终将结果交由 Akka.Net 实现的 Actor 模型执行。实验证明，通过使用交易分组并行化策略，交易的处理速度得到了很大提升。

交易分组并行化策略对计算机性能有一定的要求，因为单机的并发度太低，在一定程度上会限制策略的效率，为此，论文在交易分组并行化策略基础上引入集群，这样便可以提高策略的并发度，减少交易分组合并的机率，使得 Actor 处理更少的交易，提高策略的效率。经实验证明，基于 Actor 集群的交易分组并行化策略，交易的处理速度再度得到了提升。

关键词： 区块链，智能合约，分组，并行化，Actor，集群

ABSTRACT

The essence of the blockchain is a distributed public ledger. The ledger is open to anyone, but not a single user can modify it at will. Only the blockchain nodes selected by the consensus mechanism can change. The decentralization, immutability, security of blockchain provide a good execution environment for smart contracts.

The current execution efficiency of smart contracts has seriously hampered its application in many fields. Smart contracts are invoked through blockchain transactions, thereby increasing the efficiency of smart contract execution is equivalent to increasing the processing speed of blockchain transactions. In order to improve the processing efficiency of transactions, the paper proposes a transaction grouping parallelization strategy to process non-conflicting transactions in parallel. The smart contract platform in the strategy is based on AElf, and its smart contract is to support metadata. When processing the transaction, the strategy marks the resources accessed by the smart contract function according to the metadata, then builds the union-find set of resources according to the result of the marked resources, and then finds the subset of union-find set in which transaction lies in. Every transaction in the transactions set need be set into the corresponding group according to the ID of the subset. And then the result of the transaction grouping is merged according to the concurrency level, and finally the result is submitted to the Actor model implemented by Akka.Net to execute. Experimental results show that the transaction processing speed is greatly improved by using the transaction grouping parallelization strategy.

The transaction grouping parallelization strategy has certain requirements on the performance of the computer, because the concurrency level of the single machine is limited, which will limit the efficiency of the strategy to some extent. For this reason, the paper introduces the cluster on the basis of the transaction grouping parallelization strategy, so that it can improve the concurrency level to reduce the probability of transaction group merging, which allows the Actor to process fewer transactions and improve the efficiency of the strategy. It has been proved by experiments that the processing speed of transactions is improved again when using the actor cluster based transaction grouping parallelization strategy.

KEY WORDS： Blockchain, Smart Contract, Grouping, Parallelization, Actor, Cluster

目 录

第1章 绪论

1.1 研究背景及意义

2008 年 11 月 1 日，一篇名为《Bitcoin: A Peer-to-Peer Electronic Cash System》的论文描述了比特币（Bitcoin）的原型及基本理论。次年 Bitcoin 诞生，它的发展历经坎坷，负面言论比比皆是，但区块链作为其底层技术，逐渐引起了各界的关注，目前许多领域的专家，金融机构，创业公司，甚至一些国家机构，纷纷致力于区块链技术的研究和应用。区块链技术被认为是继大型机、个人电脑、互联网、移动 / 社交网络之后的第五次颠覆式革命[2]。

区块链是基于 Internet 的一种以分布式形式存储数据，点对点传输，利用共识机制来保证参与节点数据一致性，并采用加密算法来保证安全性的新型应用技术，该技术致力于解决“信任”，数据存储等问题。狭义上讲，它是在一个数据区块中保存上一个数据区块的摘要信息，从而达到相连的目的，并通过密码学来保证数据的不可随意篡改和伪造。广义上讲，它是利用类似链表的数据结构存储与验证数据、利用共识机制选择区块链节点来生成和更新数据、使用成熟的加密技术确保数据传输和访问的安全性，并利用由自动化脚本代码组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算范式[3]。

智能合约[4]是区块链 2.0 阶段出现的一项关键技术，它是一种使用计算机语言取代现有法律语言记录各项条款的合约。智能合约作为以数字形式存在的协议，管理着所有合约参与者事先约定的数字资产，可以由事先约定的参与者来调用。智能合约的出现，使得传统的合同可以通过代码的形式来代替。随着区块链技术和智能合约的不断发展和成熟，未来可能有一部分法律合同将由智能合约所取代，有可能实现“代码即法律”[5]。

基于区块链的智能合约的创建和执行可分为三步：首先，所有参与者共同协商合约内容，并通过代码的形式体现出来；其次，将其部署到区块链上，合约会通过 Peer-to-Peer 网络传输给每个区块链节点；最后，由于某件事情的发生，会导致智能合约被某位参与者调用执行，这个过程只有成功或失败，并且这个过程是公开透明的且不可篡改的。

相比传统合约，基于区块链的智能合约有三大优势：一是提高了执行效率。以往传统合约审批环节时间跨度大、流程复杂，而智能合约将参与者约定的所有规则写入合约代码之中，在执行过程中，会自动进行验证条件是否满足，无需审

批，自动化水平极高，大大减少了合约的构建和执行时间。二是智能合约本身的特性所决定的固有公平性[6]。智能合约是通过计算机语言写入区块链中的，并通过区块链实现去中心化，因而合约的存储，读取，执行都具有不可逆性和不可篡改性，所有参与者共同来维护与管理这个合约，并且有关合约的信息都是公开透明的。三是智能合约的易用性和高可用性。虽然各个智能合约平台对智能合约的实现并不完全相同，但通过它们的官方文档和案例可以轻松编写安全并符合要求的智能合约，并且区块链网络是分布式的，所以基于区块链的智能合约是高可用的，不用担心单个节点的宕机而导致智能合约不可用。

智能合约的执行效率虽然高于传统合约的执行效率，但它的执行效率还是有很高的提升空间的。合约的执行在区块中是通过节点发起交易来实现的，因而合约的执行效率直接体现在区块链的交易吞吐量（TPS，Transactions Per Second）上。目前 Bitcoin 的 TPS 峰值才为 7 笔交易每秒，以太坊（Ethereum）[7]也才每秒 10 笔交易左右的峰值，这远远达不到目前金融等相关应用的需求。如果合约的执行拖慢了交易的执行，将会导致区块链出现拥堵，造成大量交易的堆积，影响区块链的整体 TPS，因而提升智能合约的执行效率具有重要意义。

1.2 国内外研究现状

目前区块链 TPS 的提升是区块链项目方和研究机构的研究热点之一，因而也有很多研究成果。很多项目方和机构都是通过将现有成熟的技术与区块链进行结合，达到提升区块链 TPS 的目的。区块链快速发展的这几年，也出现了很多提高 TPS 的方案，比如区块扩容，减小区块产生间隔，降低去中心化程度，更换共识机制等。区块容量的提升意味着单个区块内可以承载更多的交易信息，但区块容量的上升，对区块链全网节点的网络传输速度有了更高的要求，如果大多数节点达不到网络要求，就会出现区块数据不同步，可能会导致全球区块链网络连锁状态下的暂时分裂。减小区块产生间隔也可以提升区块链吞吐量，但可能会带来网络安全性降低的问题。采用伪去中心化的方式，如 EOS[8]，可以加快全网共识速度，再通过改善共识机制，可以有效提升区块链的 TPS，但伪去中心同时也降低了区块链的安全性。最近比较有效提升区块链 TPS 的方式主要有两种：分片（Sharding）和交易处理并行化。Ethereum 和 Zilliqa[9]是采用分片技术的代表。

2017 年 8 月，Bitcoin 进行了硬分叉[10]，比特币现金（Bitcoin Cash），这个硬分叉对的区块进行了扩容，原来 Bitcoin 的区块大小为 1 M，而在 Bitcoin Cash 中区块大小变为 8 M，从而提高了每个区块的交易容量，区块链整体 TPS 有了显

著提升。而其他使用了与 Bitcoin 类似数据结构的加密货币网络，也必将伴随着交易量的增加而逐渐需要考虑区块扩容问题。Ethereum 为了解决区块扩容问题而设计的一种名为 Sharding 的技术方案。Sharding 的大致设计思路为：将区块链网络中的每个区块变成一个子区块链，子区块链中可以包含若干个（目前为 100 个）打包了交易数据的 Collation[11]（校验块，与平常的区块有所不同），这些 Collation 整体构成主链上的一个区块；这些 Collation 中所包含的交易数据也是全部由共识机制所选择出来的矿工节点所打包生成的，本质上和现有区块链中的普通区块没有太大区别，所以不需要额外的验证来增加安全性。通过这样的方式，区块容量就扩大为原来的 100 倍；而且这种设计便于将来的进一步扩展。Ethereum 2.0 将其网络分为两层，现有的 Ethereum 网络为上层，即主链（Main Chain），保持不变；新的分片链（Shard Chain）为底层，主要用来处理和验证各自链中的交易。为了更好地提高每个分片链的共识效率，Ethereum 2.0 将会把现有的 POW（Proof of Work）共识机制升级为 POW 与 POS（Proof of Stake）混合的共识机制，即 Casper 共识算法（CFFG，Casper the Friendly Finality Gadget），最终可能将共识机制升级为完全的 POS，即 CTFG（Casper the Friendly Ghost）。Ethereum 的 Sharding 是在项目上线之后提出的，而 Zilliqa 在创立之初便采用 Sharding，因而它的整体架构便是为了 Sharding 而设计的。其分片原理和架构与 Google 提出的 Map-Reduce 编程模型非常类似。Zilliqa 的 Sharding 技术分为网络分片、交易分片和计算分片，其中网络分片最重要，因为交易分片和计算分片是建立再网络分片基础之上的。网络分片将 Zilliqa 网路分割成包含部分节点的网络群组，这种群组被称为分片。Zilliqa 采用的共识机制为 POW 和 PBFT（Practical Byzantine Fault Tolerance）[12]，严格意义上来说，Zilliqa 在验证交易时使用的共识机制并不是 POW，PBFT 才是在分片中进行交易验证所使用的共识机制。因为通过 PBFT 共识机制使分片内达成共识的时间复杂度与分片中节点个数成平方的关系，所以，Zilliqa 通过改进 PBFT 共识机制的验证过程，使得时间复杂度与节点个数成线性关系，提高了分片内的共识效率，从而使得在节点个数大于 600 的分片中通过 PBFT 可以快速达到共识。

与 Sharding 相比，通过并行计算的方式来提升 TPS 的研究成果相对较少。Thomas Dickerson，Paul Gazzillo[13]等人提出一种基于 Software Transactional Memory 的方法，可以使得 Miner 和 Validator 并行地执行智能合约。首先，Miner “推测性”地并行化执行智能合约，即允许非冲突合约并发地执行，并且找出区块交易集合的一个并发的序列化调度；然后 Validator 根据 Miner 生成的调度并发地执行智能合约。Miner 在执行智能合约的同时使用日志记录数据行为的锁顺序，通过日志生成 Happens-Before 图，对该图进行拓扑排序便可得到一个并发调

度。Validator 为每个数据行为创建一个 Fork-Join 任务，在执行该任务前，会查询它的所有前置任务，等它的前置任务全部执行完，它才会执行。根据 Benchmark 结果，Miner 执行智能合约的效率是原来非并行情形下的 1.33 倍，Validator 的处理效率是原来的 1.69 倍。Serge，Hobor[14]等人展示了智能合约的多交易行为与共享内存并发问题的相似性。Bocchino[15]等人调查了很多确定性地重放并发调度的技术。Ethereum 是目前应用最多的智能合约平台，最近，Ethereum 上的一个项目 Plasma[16]尝试通过分片技术提升 TPS。Plasma 具有很好的扩展性，可以扩展到每秒数十亿的状态更新，致使区块链能替代全球大量的分布式金融应用。通过 Plasma 编写的智能合约能通过网络交易手续费自主地执行，这主要依赖于底层区块链来强制执行交易状态的转换。Plasma 将状态空间分割为多个子空间，在每个子空间中运行一条子链，通过树形结构构成一个包含多条子链的区块链生态系统。Garcia-Molina[17]等人提出了多版本并发控制机制，通过该机制也可以实现智能合约的并发执行，因为数据库事务与区块链交易非常类似。许多 STM（Software Transactional Memory）[18,19]技术对智能合约来说也适用，因为智能合约的执行也具有事务性。Ghosh[20]等人提出了一种新的非阻塞并发控制机制 PSTM（Proposed Software Transactional Memory），它是基于 STM 并使用多版本时间戳来实现并发控制的。该机制在遇到冲突时避免终止事务，如果两个事务的写操作在出现冲突时，不会终止其中的某个事务，而是将事务加入事务操作的变量的 Update Chain 中去。因而多个写事务可以通过构建变量的 Cascading Chain 并发地执行而不用终止或回滚任何事务。PSTM 不需要额外的 Contention Manager，因为事务自身可以解决它们所面临的冲突。与 PSTM 类似的方法是 An Zhang[21]等人提出的 MVTO（Multiversion Transaction Ordering），它被用于实现智能合约的并发执行。MVTO 主要分为两个步骤：首先 Miner 通过任意的并发控制技术去找到区块交易的一个冲突可串行化的调度；然后 Validator 通过并发和确定性地重放可串行化调度来验证区块，即执行区块中交易所包含的智能合约调用。通过 MVTO 致使区块验证速率是原来的 2.5 倍。MVTO 与 PSTM 与不同的是，PSTM 是在运行时构建 Update Chain，而 MVTO 在运行之前便构建好了 Write Chain。

1.3 本文的创新工作

绝大多数现有的区块链对区块中的交易采取串行处理的方式，不只是矿工节点，所有的验证节点也都进行同样的处理，这样的处理效率是很低的，而且浪费资源。针对现有区块链交易处理存在的问题，论文首先提出了交易分组并行化策

略（TGP，Transaction Grouping Parallelization），然后在该策略的基础上，引入集群，提出基于 Actor 集群的交易分组并行化策略（ACGP，Actor Cluster Based Transaction Grouping Parallelization），以达到提高交易处理效率的目的。主要工作及创新如下：

（1）现有区块链为了保证安全性，防止因并行处理交易而可能产生数据不一致的问题，均采用了串行处理区块中交易的方式，这种方式的低效导致区块链的 TPS 无法得到有效提高，也致使区块链无法在一些对系统吞吐量要求很高的领域得到应用。论文第三章提出的交易分组并行化策略，在交易执行前，根据交易中调用的智能合约地址和函数名称获取函数的元数据，根据元数据便可得知交易的资源占用情况，然后根据资源占用情况构建资源占用的并查集，再根据交易所在的并查集将交易进行分组，冲突的交易将会分至同一个交易组，如果分组结果中交易组数不大于区块链节点的 CPU 的核心数，则直接并行化处理各个冲突交易组，否则需要进行分组合并再并行化处理合并后的冲突交易组。

（2）区块链节点机器的性能毕竟有限，而且现有的某些区块链的节点已经开始采用集群来运行，所以可以通过集群的高性能提高并发度，减少交易分组合并的机率，最大限度地利用集群的性能，从而提升区块链的 TPS。基于这样的思想，论文第四章提出基于 Actor 集群的交易分组并行化策略，由于集群具有更多的 CPU 核心数，算法的并发度将可以变得更高，从而减少交易分组合并的机率，最终每个 CPU 核心将执行更少的交易，从而减少交易的处理时间，达到提高区块链 TPS 的目的。

1.4 论文的组织结构

论文分 5 章进行描述说明，主体结构如下：

第 1 章，绪论。首先对课题的学术理论背景和领域发展状况进行阐述，之后具体地介绍论文的研究目标、意义、主要工作以及创新点。

第 2 章，相关理论研究。介绍区块链基础概念，智能合约简介，部署流程以及执行原理，动态连通性问题，以及 Actor 模型。

第 3 章，交易分组并行化策略。智能合约在编写的时候为合约状态变量与函数加上元数据，在部署时，提取元数据并将其与合约代码一起存储，在执行区块时，解析所有交易所调用合约函数的元数据，然后进行资源标记，构建交易资源占用并查集，再根据交易的某个资源查询它所在的并查集子集，将其分组，按照这样的过程，将所有交易进行分组，之后再根据并发度进行分组合并，最后将交易交给 Akka.Net 实现的 Actor 模型执行。

第 4 章，基于 Actor 集群的交易分组并行化策略。在第 3 章提出的交易分组并行化策略基础上，引入 Actor 集群，提高组间交易的执行效率，从而提高交易分组并行化策略的效率。

第 5 章，总结与展望。总结论文所做工作，展望可能的后续研究工作，并讨论其他研究的可能性。

第2章 相关理论研究

2.1 区块链

区块链（Blockchain）首次出现在公众的视野中，是源于 2009 年初上线的 Bitcoin 项目。区块链实际上是记账问题从古演变至今的分布式场景下的必然结果。区块链目前在金融行业有很多应用落地，如跨境支付，与此同时，很多区块链项目方与机构也在将区块链引入其他行业。一般认为，区块链具有如下三个特性[22]：

- 分布式容错性：区块链分布式网络具有高可用性，允许部分节点出现异常状态；
- 不可篡改性：经过共识后的数据会永久存在，不可被修改或删除；
- 隐私保护性：区块链中大量的密码学保证了数据的安全性与隐私性。

区块链基本原理并不是很复杂。首先得理解三个基本概念：

- 交易（Transaction）：交易是对区块链数据的操作，它会导致区块链状态的改变，比如它会添加一条转账记录，调用一次智能合约，通过合约调用更改区块链的状态；
- 区块（Block）：一个区块中包含很多交易，它表示对当前区块链状态的一次共识；
- 链（Chain）：每个区块中包含前一个区块的摘要信息，从而构成一条完整区块链，包含了整个区块链的状态信息。

如果把区块链看作一个状态机，每个交易的执行都会导致一次状态改变，最终生成的区块表示所有节点对区块中所包含的状态改变的共识。

区块链的底层结构类似一个线性的链表（如图 2-1 所示），链表的每个元素为一个区块，链表元素之间是通过后继区块包含前置区块的哈希（Hash）值进行链接的。每个区块以及其中的交易可以通过计算哈希值的方式进行快速校验。区块中每次新添加的区块都需要区块链网络节点的共识。

对于区块链的工作过程，这里以 Bitcoin 为例进行阐述。首先，用户通过 Bitcoin 钱包发起一笔交易，之后该交易会通过 P2P 网络广播到其他 Bitcoin 节点，并进入节点的交易池等待确认（交易的产生与广播过程如图 2-2 所示）。与此同时，网络中的矿工节点会将待确认的交易打包到一起，再加上当前最新区块的哈希值，时间戳等信息，组成一个区块，然后选择一个随机数添加

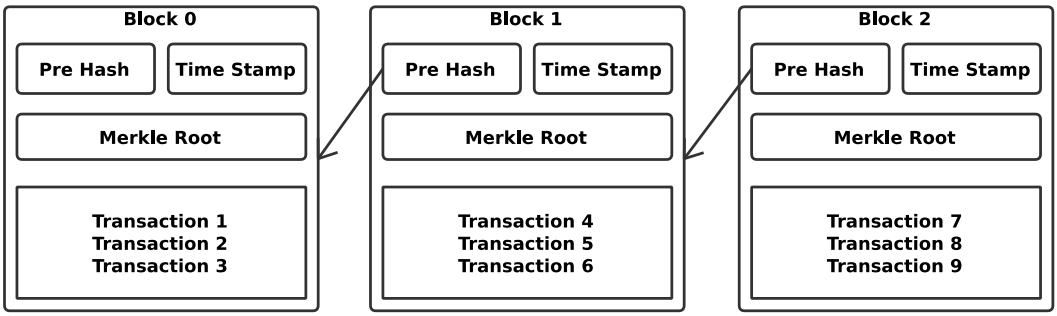

图 2-1 区块链数据结构

到区块中使得区块的哈希值满足一定条件。寻找这个随机数的过程就被称为POW。

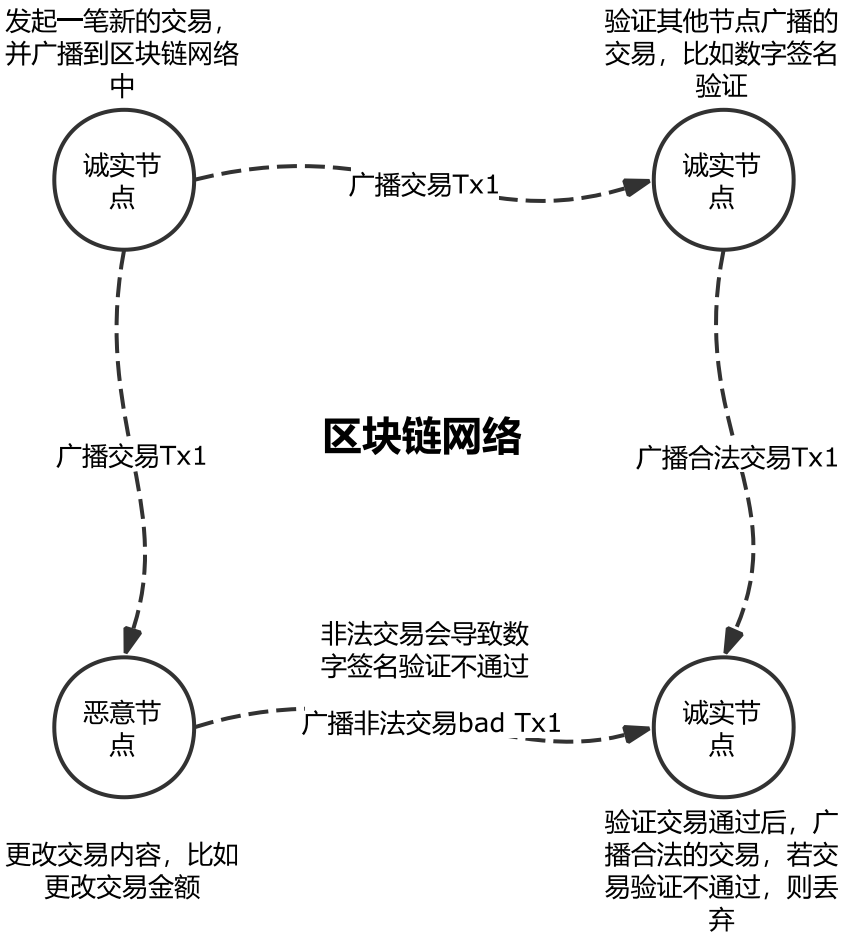

图 2-2 交易的产生与广播过程

一旦矿工节点找到满足条件的随机数之后，则该节点打包的区块便成为候选区块，节点需要将其广播到网络中去，其他节点在收到候选区块后，会停止寻找随机数并且去验证候选区块，如验证通过，便将其添加至自己的区块链中去，否则丢弃区块。当网络中大部分节点都验证通过了该候选区块，那么该区块就被整个网络所接受，区块中交易也得到确认，至此网络的一次共识结束。区块的产生过程如图 2-3 所示。

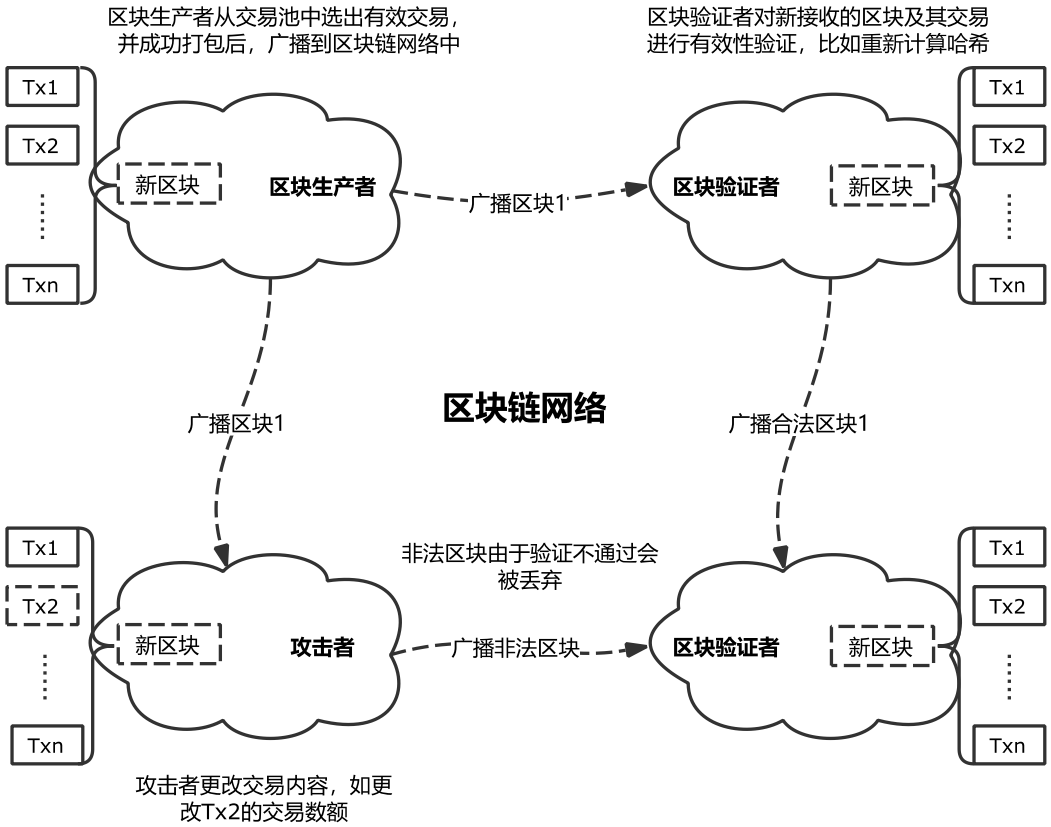

图 2-3 区块的产生与广播过程

Bitcoin 的共识机制 POW 是基于算力的，因为想要让哈希值满足一定条件，目前没有捷径，没有快速的启发式算法，只能通过暴力去挨个尝试，随着暴力尝试的次数越多，暴力求解成功的概率也就越大。通过调整哈希值的限制条件，可以控制区块产生的时间，Bitcoin 通过对其调整，使得区块每 10 分钟左右产生一个。Bitcoin 网络中的节点当然不会免费地去寻找哈希值，它们也是有收益的，一个区块的产生必定伴随着 Bitcoin 的产生，产生的 Bitcoin 会到达产生该区块的节点账户中去，除此之外，区块中的每笔交易的手续费也会归该节点所有。Bitcoin 网络中的节点是来去自如的，Bitcoin 网路允许存在恶意节点，因为它不一定能对网络构成威胁，它可以不承认其他节点产生的区块。但是 Bitcoin 网络中的大部分节点都是“诚实”节点，它们默认都只承认最长的一条链结构。只要网络中

有一半节点是“诚实”节点，那么最长的链将很大概率上成为最终被全网承认的链，也即合法链。随着时间的增加，最长的链称为合法链的概率会越来越大。比如，一笔交易在经过 6 个确认之后，也即在该交易所在的区块之后又产生了 5 个区块，即便有为数一半的节点联合起来更改这笔交易，其概率为$\frac{1}{2}^{6} \approx 0.016$，换言之，就是交易在经过 6 个确认之后，将变得不可更改。当然如果整个网络中大多数节点联合起来进行恶意攻击，整个系统将完全崩溃，不过要做到这一点需要付出很大代价，与进行恶意攻击所获得的收益相比，往往是得不偿失的。

2.2 智能合约

2.2.1 智能合约模型

智能合约（Smart Contract）的概念率先由 Nick Szabo 于 1994 年提出，它是一种计算机协议，目的是以数字形式促进、验证或强制合约的协商与执行。当时，智能合约并没有在实际领域中得到应用，因为缺乏可信任的运行环境，Bitcoin 诞生以后，其底层区块链技术进入人们的视野，它天生可以为智能合约提供可信的执行环境。Vitalik Buterin 等人率先抓住了区块链可以与智能合约结合的机遇，发布了白皮书《Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform》，并一直致力于 Ethereum 的开发与优化，努力将其打造成最优的智能合约平台。

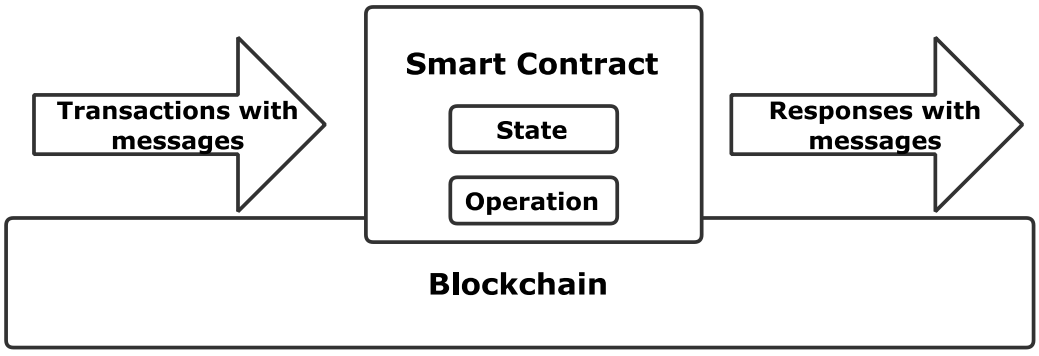

图 2-4 智能合约模型

智能合约不仅仅是一个通过响应事件可自动执行的计算机程序，它自己同时也是一个区块链系统的参与者。它可以对接收的消息进行响应，它可以接收和存储价值，也可以对外发送消息和价值。基于区块链的智能合约类似一个可被信任的第三方，可以临时保管资产，它只会执行事先规定的操作。下图便是一个智能

合约模型：智能合约被部署在区块链上，它可以维护自己的状态，控制自己所管理的数字资产，可以对接收的消息进行响应。智能合约模型如图 2-4 所示。

简单地说，智能合约就是传统合约用计算机语言实现的数字化版本，它是运行在区块链上的计算机程序，可以在满足一定条件时执行。智能合约在编写完成后，一旦部署到区块链上，就不可更改。

2.2.2 智能合约的部署与执行

智能合约是函数操作和数据的集合，可以部署在像 Ethereum，Neo[23]，AElf[24]等智能合约平台上运行。不同的智能合约平台的智能合约执行机制实现并不相同，有的智能合约平台实现了类似 JVM（Java Virtual Machine）的

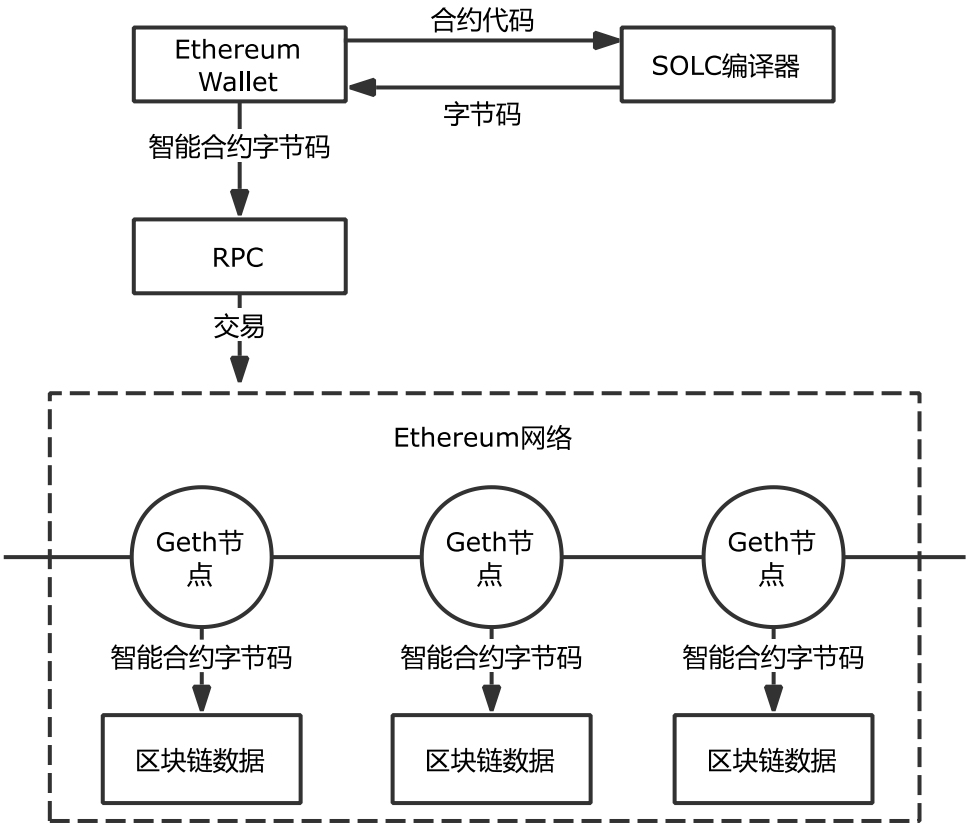

图 2-5 基于 Ethereum 的智能合约部署流程

虚拟机，如 EVM（Ethereum Virtual Machine），智能合约编译后得到的字节码由智能合约虚拟机来解释执行；另一类智能合约平台则利用一些高级语言的反射机制来实现智能合约的执行，如 AElf。智能合约在运行时，它的某些操作是受

限的，它是不能有网络、文件和多线程并发等操作的，这也是为了保证安全才做出的限制。

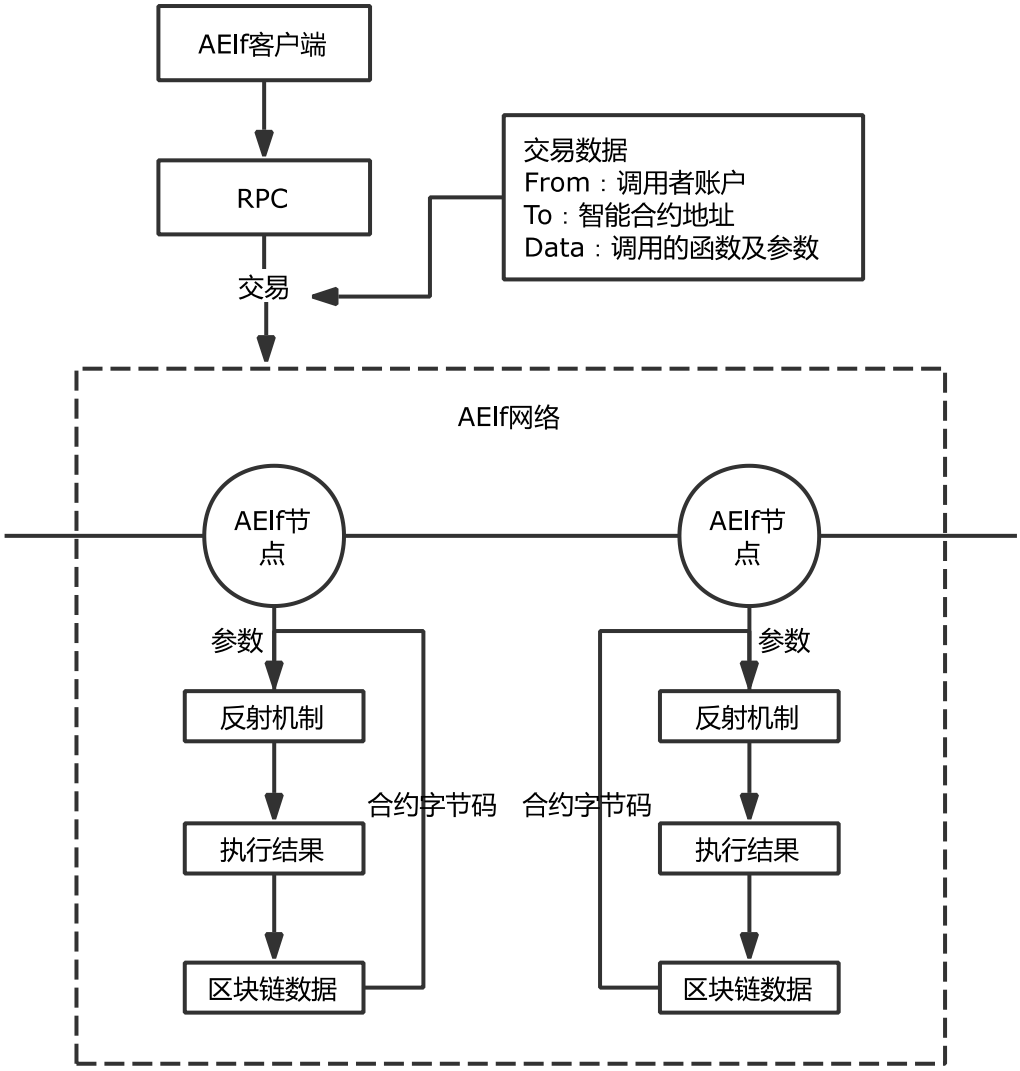

图 2-6 基于 AElf 的智能合约执行流程

由于智能合约平台的实现不同，智能合约的部署也不尽相同。对于 Ethereum，智能合约一般都是由 Solidity[25]编写，编写完成后经 SOLC 编译成字节码，然后通过一笔交易来实现智能合约的部署，交易中包含合约创建者的账户地址，合约字节码等信息，交易成功后会得到合约的地址，它是由创建者的账户地址和创建者账户的 Nonce 作为输入，通过 Kecca-256 加密算法生成的。Ethereum 上智能合约的部署流程如图 2-5 所示。对于 AElf，智能合约由 C#编写，编写完成后经 dotnet 编译后得到 dll 文件，之后将 dll 文件放置指定文件夹下，通过发交易的方式来实

现智能合约的部署。在 AElf 中，部署智能合约也是在调用智能合约，不过调用的这个合约是 AElf 内置的智能合约，通过调用它可以实现智能合约的部署。

智能合约是部署在区块链上的代码，但区块链本身并不执行代码，代码是由智能合约平台来执行的，如 EVM，AElf 的反射机制等。智能合约的执行是由于区块链节点发送交易触发的，其他区块链节点在接收到该交易后会验证交易有效性，如果有效则去区块链数据库中取出合约代码执行，并将执行结果写入区块链数据库。AElf 上智能合约的执行过程如图 2-6 所示。

智能合约的执行时机主要有两个：一个是在矿工节点（Miner）挖矿成功后。矿工节点负责将交易写入区块链，它的工作类似会计，主要就是修改和维护“数字账本”。矿工节点在“挖矿”成功后，即获取记账权之后，便将产生的区块广播到区块链网络中去，与此同时也要将区块加入到自己的区块链中去，此时会去串行化地处理区块中包含的交易及合约代码。另一个执行时机是在验证节点（Validator）接收到区块之后。验证节点可以是全节点但非矿工节点，也可以是矿工节点。验证节点在接收到区块之后，首先会先去验证区块及区块中交易的合法性，如果区块合法，便会去处理区块中交易及智能合约，此时也是串行化的处理。

2.3 动态连通性问题及并查集

2.3.1 动态连通性问题

一个动态连通性问题（Dynamic Connectivity Problem）[26]可表述为：

Given a set of N objects, connect two objects, or ask if two objects are connected(directly or in-directly).

在图论中，动态连通性问题可描述为，对于具有 N 个顶点的图，连通给定的两个节点或判断两个顶点是否在同一个连通分量中。这个问题可通过两个命令实现：Union 和 Connected。

- Union(a, b)用于连接 a 和 b 顶点
- Connected(c, d)用于判断顶点 c 和 d 顶点是否在同一个连通分量内

对于给定的顶点 p，顶点 q 与顶点 r，连通性具有以下特性：

- 自反性：p 与 p 自身是连通的；
- 对称性：如果 p 与 q 是连通的，那么 q 与 p 也是连通的；
- 传递性：如果 p 与 q 是连通的，q 与 r 是连通的，那么 p 与 r 也是连通的。

2.3.2 并查集

为解决连通性问题，并查集（Union-find 或 Disjoint-set）[27]应运而生。并查集是一种树形的数据结构，用于处理一些 Disjoint-Sets 的合并及查询问题。并查集中包含多个元素，每个元素由一个 id，parent 指针，以及 size 和 rank（存在于某些改进的并查集算法中）。并查集中涉及以下几个操作：

- MakeSet：创建只包含一个元素的子集；
- Find：查询某个元素在哪个子集中；
- Union：合并两个元素所在的子集。

实现 MakeSet 操作的伪代码如表 2-1 所示：

表 2-1 MakeSet 操作

算法：MakeSet操作
// 输入：需要添加到并查集中的某一元素x
1. function MakeSet(x)
2. 判断x是否已经存在，不存在算法结束，否则继续；
3. x.parent = x
4. x.rank = 0
5. x.size = 1

表 2-2 Path compression 算法

算法：Path compression算法
// 输入：被查询元素x
// 输出：返回x所在子集的根元素
1. function Find(x)
2. if(x.parent != x)
3. x.parent = Find(x.parent)
4. return x.parent

Find 操作根据被查询元素的 parent 指针链一直遍历到根元素，根元素的 parent 指针是它自己，根元素可以代表一个子集，它可以标识元素属于哪个子集。Find 操作有三种实现方式：Path compression 算法，Path halving[28]算法和 Path splitting[28]算法。Path compression 算法通过在查询元素的过程中将遍历过的元素的 parent 指针都指向根元素，在一次查询过后，树会被扁平化，即树的高度降低。

Path compression 算法不仅会加速后期对查询过程中遍历过的元素的查询，同时会加速引用这些元素的查询。Path halving 会将遍历过的元素的 parent 指针指向其 parent 元素的 parent 元素。Path splitting 算法会像查询路径上的元素的 parent 指针指向其 parent 元素的 parent 元素。Path compression 算法，Path halving 算法以及 Path splitting 算法的伪代码分别如表 2-2，2-3，2-4 所示：

表 2-3 Path halving 算法

算法：Path halving算法
// 输入：被查询元素x
// 输出：返回x所在子集的根元素
1.　function Find(x)
2.　　while(x.parent != x)
3.　　x.parent = x.parent.parent
4.　　x = x.parent
5.　return x

表 2-4 Path splitting 算法

算法：Path splitting算法
// 输入：被查询元素x
// 输出：返回x所在子集的根元素
1.　function Find(x)
2.　　while(x.parent != x)
3.　　t = x.parent
4.　　x.parent = x.parent.parent
5.　　x = t
6.　return x

Union 操作可用 Union(x, y)表示，它表示将 x 与 y 所属的集合进行合并，并且在合并之前会先检查 x 与 y 是否属于同一个集合，即检查 Find(x)与 Find(y)是否相等。通常 Union 操作很简单，只需要将其中一个元素所在集合的根元素设置为另一个元素的子元素即可。不过这样会导致并查集的树形结构的高度变高，为了防止这种情况的发生，通常可以在 Union 操作过程中使用 Rank 或 Size。如果使用 Rank 来进行 Union 操作时，需要将高度较小

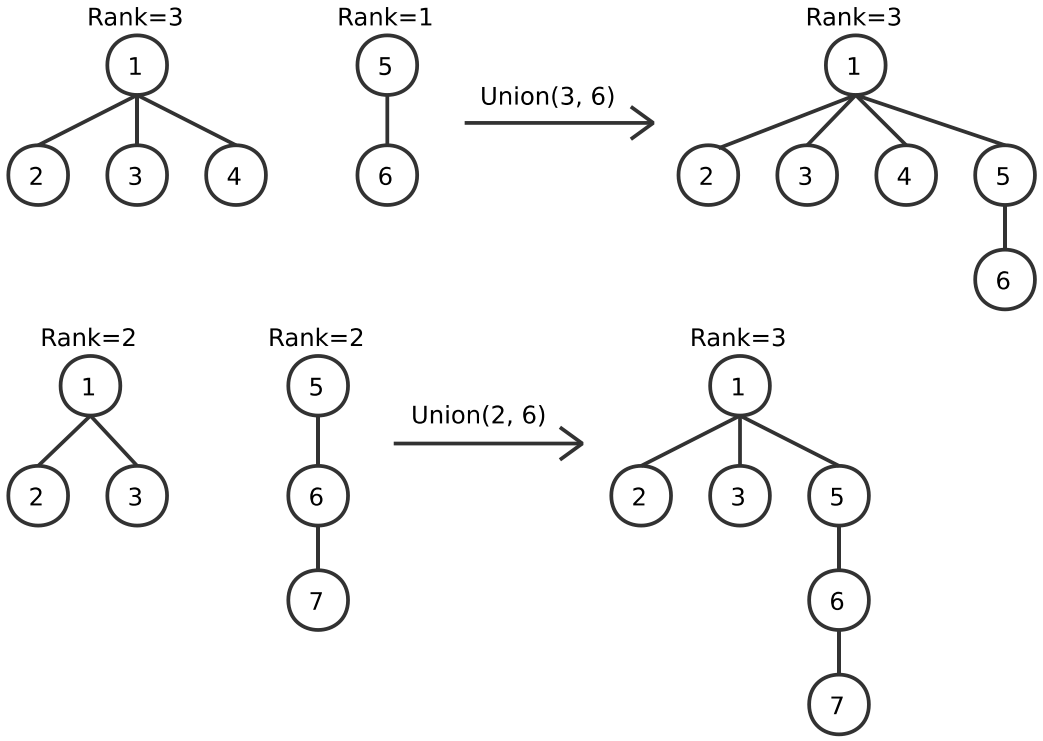

图 2-7 Union by rank 算法

表 2-5 Union by rank 算法

算法：Union by rank算法
// 输入：进行Union的元素x和y
1. function Union(x, y)
2. xRoot = Find(x)
3. yRoot = Find(y)
4. if(xRoot == yRoot)
5. return
6. if(xRoot.rank < yRoot.rank)
7. t = xRoot
8. xRoot = yRoot
9. yRoot = t
10. yRoot.parent = xRoot
11. if(xRoot.rank == yRoot.rank)
12. xRoot.rank = xRoot.rank + 1

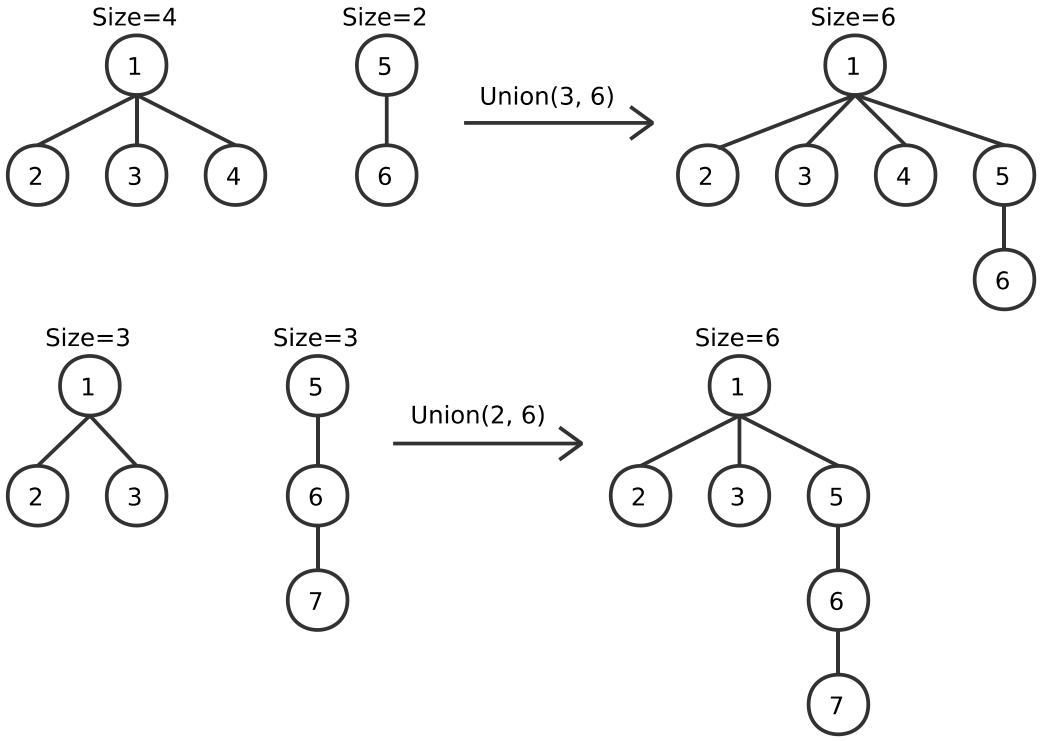

图 2-8 Union by size 算法

表 2-6 Union by size 算法

算法：Union by rank算法
// 输入：进行Union的元素x和y
1.　function Union(x, y)
2.　　xRoot = Find(x)
3.　　yRoot = Find(y)
4.　　if(xRoot == yRoot)
5.　　return
6.　　if(xRoot.size < yRoot.size)
7.　　t = xRoot
8.　　xRoot = yRoot
9.　　yRoot = t
10.　　yRoot.parent = xRoot
11.　　xRoot.size = xRoot.size + yRoot.size

的树附着于高度较大的树的根元素上，通常这样不会使得合并后的树比原来的树更高，除非是合并之前两棵树高度相同，这样会导致合并后的树的高度等于原来

的树的高度加 1。为了实现按 Rank 来进行 Union，集合中的每个元素都需要与一个 Rank 进行关联。最初，集合中只有一个元素的时候，Rank 为 0。如果两个集合具有相同的 Rank，则合并后的集合的 Rank 加 1，否则合并后的集合的 Rank 等于合并前的两个集合中 Rank 较大者。这里的 Rank 并不是并查集中树的高度或深度，因为 Path compression 会随着时间改变树的高度或深度。按 Rank 进行 Union 操作的过程如图 2-7 所示，伪代码如表 2-5 所示。如果使用 Size 来进行 Union 操作时，需要将包含更少元素的树附着于包含更多元素的树的根元素上。按 Size 进行 Union 操作的过程如图 2-8 所示，伪代码如表 2-6 所示。

2.4 Actor 模型

过去十几年摩尔定律[29]一直指导着 CPU 的发展，CPU 单核性能越来越高，但是从最近几年发展来看，摩尔定律已然失效，CPU 的工艺制程和发热稳定性之间难以取舍，增加 CPU 的核心数量成为替代方案，目前单 CPU 多核的家用办公电脑已经很常见，服务器级 CPU 的核心数更是达到了家用电脑 CPU 核心数的 8 倍，甚至是更高，因为现在很多服务器都有多个 CPU。为了能够充分利用多核 CPU 的性能，并发性就应该提前在代码层面被考虑到。通过前人的开发经验来看，Threads 并不是实现并发性的好方法，因为 Threads 往往会带来难以查找追踪的bug，但是技术一直在更新，目前已经有很多其他方法来实现易用的并发性，比如 Actor 模型。

Actor 模型是一种并发模型，在 1973 年于 Carl Hewitt，Peter Bishop 及 Richard Steiger[30]的论文中提出，它与另一种共享数据模型完全相反，Actor 模型中并没有共享数据。所有的计算单元通过消息传递的方式进行合作，这些计算单元称为 Actor。共享数据模型更适合单机多核的并发编程，但是共享数据模型也存在很多问题，在编程过程中也比较复杂。随着多核时代和分布式系统的到来，在并发编程中再使用共享数据模型已经诟病百出，因此人们又重新启用了几十年前就提出的 Actor 模型。Map-Reduce[31]就是一种典型的 Actor 模型，Erlang 在语言层面就对 Actor 模型提供了支持，Scala 也提供了 Actor 模型的支持，但并不是在语言层面的支持，很多高级语言如 Java，C#也提供了实现了 Actor 模型的第三方库。

Actor 模型遵循的哲学是“一切皆是 Actor”，这与面向对象编程的“一切皆对象”类似，但是在面向对象编程中，对象之间通常都是顺序执行的，而在 Actor 模型中，Actor 之间都是可以并行执行的。一个 Actor 指的是一个最基本的计算单元，它能接收一个消息并基于此执行其他的一些逻辑，比如，创建更多的 Actor，发送更多的消息，以及如何响应下一条消息等。消息的发送者与发送的消息是解

耦的，这样就使得异步通信成为了可能。消息接收者之间是通过地址来识别对方的，地址有时也称为 Mailing Address。因此一个 Actor 只能和它拥有对方 Mailing Address 的 Actor 通信，Actor 可以通过它接收到的消息中获取到 Mailing Aaddress，它也可以获取到自己创建的 Actor 的 Mailing Address。在 Actor 模型中，Actor 内部和 Actor 之间都具有并发性，Actor 可以被动态地创建，消息中包含 Actor 的 Mailing Address，Actor 之间的交互是通过异步消息传递来实现的，且消息的传递无所谓顺序。

在一个实现 Actor 模型的系统中，可能会有多个 Actor 同时在运行，但是每个 Actor 对于接收的消息都只能顺序地去处理，这意味着如果一个 Actor 接收到多条消息，它只能一条一条地处理。如果你想要同时处理多条消息，你可能需要创建同样数量的 Actor，每个 Actor 处理一条消息。Actor 在接收到多条消息之后需要将它们存储在某个地方，在 Actor 模型中，Mailbox 就是用来为 Actor 存储消息的，Mailbox 工作原理如图 2-9 所示。

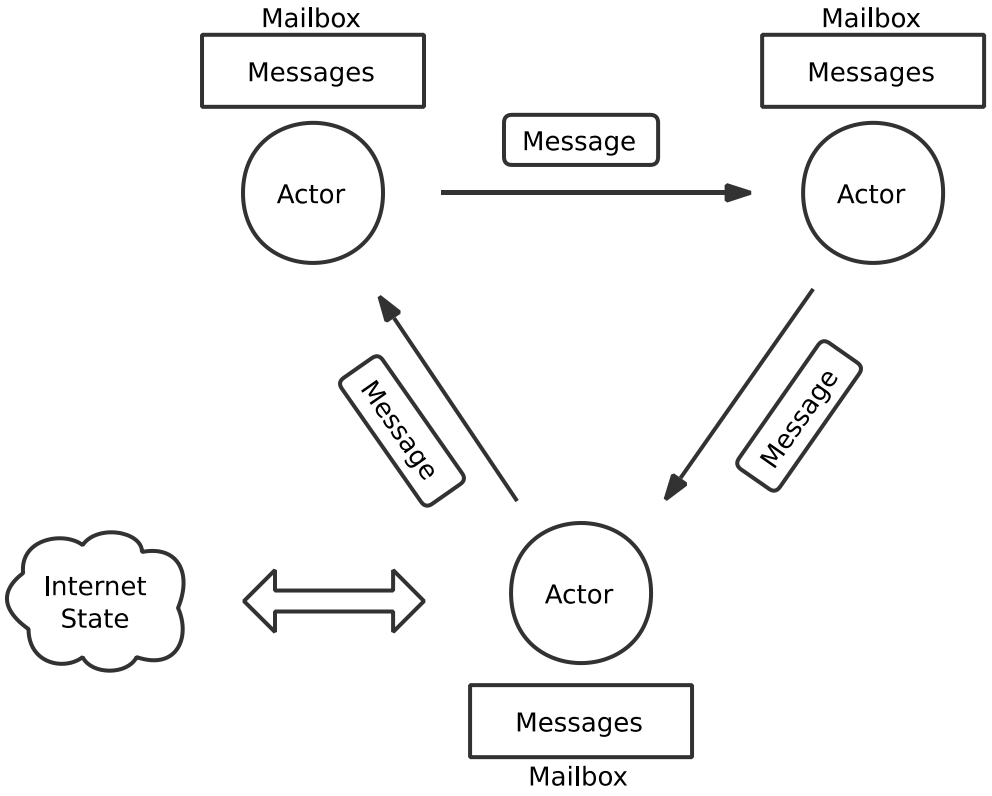

图 2-9 Mailbox 工作原理

Actor 模型具有良好的容错性，每个 Actor 都是独立运行的，Actor 之间是完全隔离的，这意味着一个 Actor 的状态不会影响另一个 Actor 的状态。在 Actor 模型中有一个类似监督者的 Actor，当一个 Actor 的运行状态有异常时，该监督者获取该信息，然后根据异常信息执行相应的逻辑。通过这种方式，可以创建一个可以自我修复的系统，也即，如果一个 Actor 进入了异常状态，监督者可以执行某些逻辑（比如重启 Actor）使异常的 Actor 重新处于正确的状态。Actor 模型还具有良好的分布式特性，Actor 模型中的 Actor 可以运行在不同的节点上，一个发送消息的 Actor 可以运行在一个节点上，接收消息的 Actor 可以运行在另一个节点上。在 Actor 模型中，物理机节点被弱化了，Actor 模型中只有包含 Mailbox、状态和代码的 Actor，只要消息能到达 Actor，Actor 并能做出响应即可。

第3章 交易分组并行化策略

区块链发展至今，TPS 依然是其瓶颈问题，Bitcoin 每秒 7 笔左右的交易量，Ethereum 每秒支持 10 笔左右，这样的 TPS 根本无法与“双十一”购物大促期间的支付宝相比。2017 年 11 月 11 日，支付宝 TPS 峰值达到 25.6 万。因而区块链若想要得到大规模应用，TPS 是必须要解决的问题之一。至今，提升 TPS 的方式比较有效且安全的有两种：分片与交易并行化。分片技术致力于解决区块链的扩展性问题，并通过改变网络验证的方式来提升 TPS。Ethereum 已经推出了分片技术和 Plasma，目前研发已经接近尾声，Ethereum 正逐渐将分片技术移植到现有的链上。除了 Ethereum，其他一些市值排名靠前的区块链项目也致力于将分片技术应用应用到自己的链上。Zilliqa 在 2018 年 4 月的测试网中向世界展示了每秒 2400 笔的交易吞吐量。交易并行化的方式相对来说，研究成果相对较少，几乎没有完整的可商用化的模型。因此，本章提出交易分组并行化策略，在处理交易时，通过对区块中的交易进行分组，将互相不冲突的交易并行化处理，从而提升交易的处理速率以及 TPS。

3.1 Amdahl 定律与区块链交易的并行处理

Amdahl 定律[32]，是由 Gene Amdahl 于 1967 年在 AFIPS Spring Joint Computer Conference 上提出的，它表示串行执行的计算机程序在并行执行后效率的提升。在并行计算中经常使用 Amdahl 定律来预测在使用多处理器时的理论加速。Amdahl 定律的加速比的计算方法如公式(3-1)所示。

$$S_{latency} = \frac{1}{(1-p)+\frac{p}{s}} \tag{3-1}$$

其中$S_{latency}$为执行整个程序或任务的加速比，s为改善系统资源（如增加处理器数量）所带来的部分任务执行的加速比，p为改善系统资源可以带来执行效率提升的那部分任务执行时间占总执行时间的比例。而且

$$\begin{cases} S_{latency} \le \frac{1}{1-p} \\ \lim\limits_{s\to\infty} S_{latency}(s) = \frac{1}{1-p} \end{cases} \tag{3-2}$$

表明随着系统资源的不断改善，理论上执行整个任务的加速比也会随之增加。但不管系统资源改善的程度有多高，理论上的加速比总是会受到无法从改善系统资源而执行效率得到提升的那部分任务的限制。

例如，如果一个程序的执行需要占用处理器 20 小时，但是程序中的某段代码需要占用处理器 1 小时且这段代码不能够并行执行，其余代码均可以并行执行，那么不管有多少处理器执行这个程序，最少执行时间不可能少于那关键的 1 小时。因此理论上的加速比最多为$\frac{1}{1-p}=20$，其中p为可并行时间的占比，也即$p=\frac{19}{20}$。由于这个原因并行计算一般只对于高度并行的程序有作用。

Amdahl 定律是针对的问题的规模是固定的，因此 Amdahl 定律对区块链交易的执行问题是适用的。由于区块链中每个区块的交易大部分是不相关的，根据 Amdahl 定律可知，并行处理交易可以使得区块的执行效率得到快速提升，从而提升区块链的 TPS。

3.2 交易冲突

区块链中的交易类似数据库中的事务，区块链中的交易也具有原子性，交易中涉及的数据操作要么全部执行，要么回滚至交易执行之前的状态。在数据库中，对数据主要有两种操作：读操作与写操作。冲突在数据库定义为，如果两个数据操作在不同的顺序下执行得到不同的结果。在区块链中，交易之间的冲突可定义为，如果两个交易同时执行可能会导致区块链数据不一致的问题，则它们之间是互相冲突的关系。

对于某一简单的 Token 合约，合约中包含一个 Transfer 函数，当账户之间进行转账时将会调用该 Transfer 函数。例如存在三个账户 A，B，C，每个账户的当前余额均为 100，如果此时存在两笔交易$TX_1=\{Transfer\ A\rightarrow B,100\}$，$TX_2=\{Transfer\ B\rightarrow C,200\}$，$TX_1$表示账户 A 向账户 B 转账 100，$TX_2$表示账户 B 向账户 C 转账 200，如果$TX_1$在$TX_2$前执行，则两笔交易都会成功执行，若$TX_2$在$TX_1$之前执行，则$TX_1$会成功执行，$TX_2$会失败。所以当矿工节点按照一定顺序打包交易后，若验证节点按照不同的交易顺序执行区块中交易，将会造成区块链数据不一致的问题，即其他区块链节点的数据与该节点的数据不一致。若是TX_1与TX_2同时执行，会造成区块链数据不一致的问题。

在论文中提出的交易分组并行化策略中，会在交易执行前，分析出交易之间的冲突关系，矿工节点对于冲突的交易将按照自己打包的交易顺序进行串行化处理，而对于非冲突交易进行并行化处理，验证节点对于冲突的交易将按照接收的区块的交易顺序进行串行化处理，并行化处理非冲突的交易。

3.3 智能合约数据分类

交易分组并行化策略的切入点是资源占用隔离，进行交易分组时，先检测交易所占用的资源（状态变量等），之后根据这些资源占用的情况，将占用不同资源的交易分离并行化处理（如图 3-1 所示）。为了实现在交易处理之前实现交易分组，资源占用的检测需要以静态分析的形式进行。

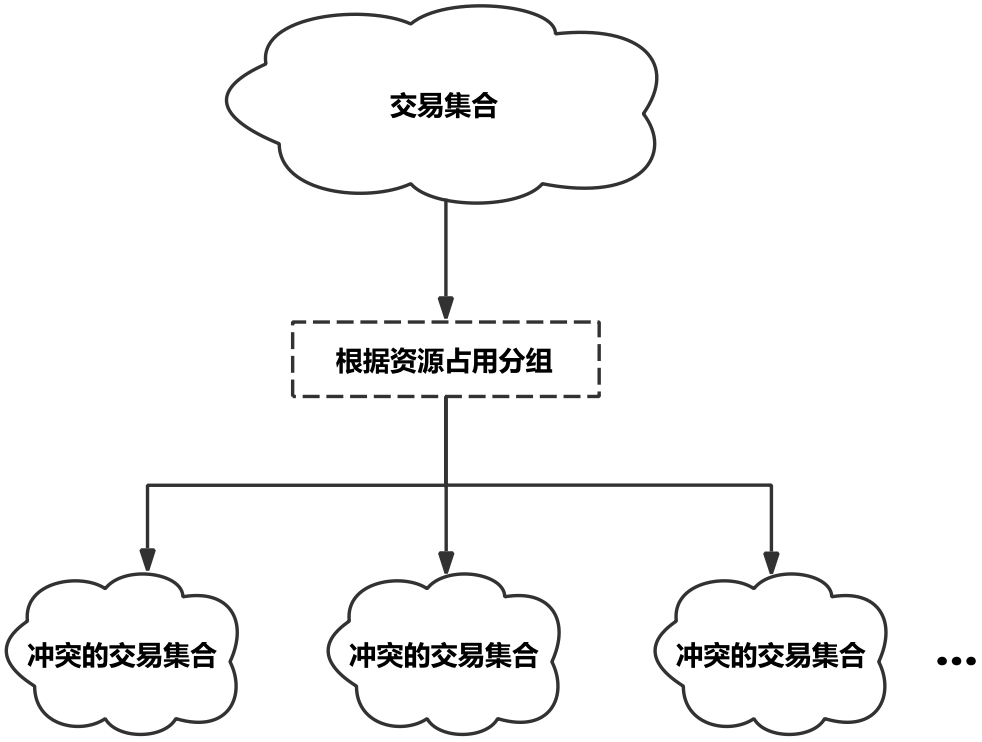

图 3-1 交易分组并行执行模型

资源在智能合约中一般指智能合约的状态变量或数据，在介绍分组策略之前，先将资源数据进行一下分类，将智能合约的状态数据分为三种类型：

（1） 只读的账户间共享数据（Read-only account-sharing data）：这类数据在交易处理过程中不会被更改，例如，区块高度，前置区块的哈希值以及用户在智能合约中定义的只读的常量。

（2） 可读写的账户间共享数据（Read-write account-sharing data）：该类数据可被多个账户进行读写操作，例如一个投票合约，多个投票人可以同时投票给同一个候选人。

（3） 账户相关数据（Account-specific data）：该类数据通常是一个 Key-Value 类型的数据，Key 一般为区块链账户 Address，Value 为与账户 Address

相关的数据。一个账户 Address 只能访问与它自己相关的数据。例如一个 Token 合约，存储账户余额的 Map 类型的变量中存储每个账户 Aaddress 的余额情况，每个账户 Address 只能访问该 Address 的余额，而不能访问其他账户 Address 的余额。

对于只读的账户间共享数据，多个交易可以同时读取这些数据，因为它们是只读的，同时访问它们的多个交易可以并行处理，并不会影响区块链数据的一致性。对于可读写的账户间共享数据，当多个交易访问它们时，如果只是读，并不会出现数据不一致的问题，但写操作会造成数据不一致的问题，由于编写智能合约的语言一般都是图灵完备的，是无法检测是对数据是读操作还是写操作，因而，针对这种情况，只要多个交易同时访问可读写的账户间共享数据，则认为它们是冲突的。对于账户相关的数据，该类数据可分为两种情况，一种是多个交易涉及不同的账户，此时，这些交易可以对账户相关的数据同时读写，可以并行执行；另一种情况是若多个交易涉及相同的账户，此时，这些交易是无法对账户相关的数据同时读写的，因为同时读写会造成数据不一致的问题，所以这些交易也是冲突的，无法并行执行。

这种分类方式的原因是智能合约的业务逻辑都是和账户相关的。换句话说，就是很多智能合约的函数一般都只访问账户相关的数据。在这种情形下，如果不同账户操作账户相关的数据，那么这些交易是可以并行化处理的。

3.4 函数与状态变量的元数据

所谓元数据，就是描述数据的数据[33]。如何描述一个正在运行的程序如何使用，这正是元数据的作用。程序可接受的输入，内部的结构和逻辑处理的方式，程序最终的输出，这些要素之间的内在逻辑关系对用户来说通常是复杂而难以理解的。元数据是在不影响程序原有的功能前提下，提供给用户了解程序的重要资料。

为了获取某个交易所占用的资源集合，需要知道每个智能合约的每个函数的资源占用情况。为了实现这个目的，选择使用函数的元数据。通过函数的元数据可以标记每个函数所占用的资源情况，例如状态数据等。元数据中包含函数的调用集合（Calling set）和资源集合（Resource set）。函数的调用集合表示的是该函数调用了哪些其他函数，调用集合中包含所有的被调用函数的集合；资源集合表示的是该函数使用了数据，不管是读操作还是写操作。这里，使用智能合约地址与资源名称的联合字符串作为资源的名称，它具有唯一性，不管是智能合约里的普通变量，还是集合变量。在 AElf 中，智能合约是使用 C#编写的，由于 C#

是支持元数据的，论文中使用的是 C#中的 Attribute 元数据，因而便可以对智能合约进行元数据静态分析。由于智能合约中主要由两部分组成：状态变量和函数，因而定义两种元数据：状态变量元数据（SmartContractFieldData）和函数元数据（SmartContractFunctionData）。下面以一个 Token 合约为例，叙述 SmartContractFieldData 和 SmartContractFunctionData 如何在智能合约中使用。

SmartContractFieldData 接收两个参数，状态变量的名称和状态变量的类型（也即 3.3 节中定义的数据类型），用 SmartContractFieldData(FiledName, DataAccessMode)来表示。例如 Token 合约中的状态变量 Balances，它的元数据可表示为 SmartContractFieldData(”${this}.Balances”, DataAccess-Mode.AccountSpecific)，其中”${this}.”在合约部署后会被替换为智能合约地址。SmartContractFunctionData 接收三个参数，函数名称、调用的智能合约集合和访问的状态变量集合，用 SmartContractFunctionData(FunctionSignature, CallingSet, LocalResouces)。例如合约中的函数 Transfer，它的功能为从发送者账户发送一定数量的 Token 至接收者账户中，它在代码中使用了 Token 合约的状态变量 Balances，但它并没有调用其他合约，因此它的元数据可表示为 SmartContractFunctionData(”${this}.Transfer”, {}, {”${this}.Balances”})，其中”${this}.”在合约部署后会被替换为智能合约地址。

3.5 交易分组

在提出交易分组并行化策略之前需要一些前置条件，即智能合约需要使用元数据对合约状态变量和函数进行标记，并在部署的时候需要将元数据和合约代码一起存入区块链数据库。上一节已经叙述了如何使用元数据对状态变量和函数标记，这一节讲述在 AElf 上部署智能合约的流程。在 AElf 中智能合约的部署需要发起一笔交易，该交易调用一个 AElf 的 Basic Contract（该合约是在启动区块链的时候就已经写入区块链的创世智能合约），在合约执行时，会去到节点指定目录下获取合约代码，通过反射机制对合约代码进行代码分析检查，比如是否调用了一些禁止的 Libraries（多线程库，网络库），如果代码检查通过，则通过反射机制提取智能合约状态变量和函数的元数据，更新区块链合约调用图，使用智能合约地址替换状态变量与函数元数据中的”${this}.”，根据更新后的状态变量名称元数据与函数名称元数据将它们的元数据存入数据库，最后根据合约地址将合约代码存入数据库。

通过以下三个步骤将单个区块中的交易实现分组：

1. 首先从区块中提取出交易集合，遍历交易集合，根据每个交易调用的合约函数名称和智能合约地址获取函数的元数据，从而可以得到交易占用的资源集合。

2. 根据资源集合构建资源依赖无向图。在这个无向图中，节点为一种资源，如果一个交易中的智能合约函数访问了两种资源，那么这两个资源之间就会有一条边。

3. 根据资源依赖无向图将区块中的交易分配到不同的组中去，每个连通分量所关系到的交易都会被分配到同一个组中。

假设有交易$t_1[Alice, Bob]m_1$和$t_2[Chad, Helen]m_1$，其中m_1是账户相关数据，交易t_1会访问$m_1.Alice$和$m_1.Bob$两个资源，交易t_2会访问m_1.Chad 和$m_1.Helen$两个资源，由于这两个交易之间没有资源冲突，所以可以并行地处理它们。

如果访问同一个可读写的账户间共享数据的交易，它们之间是冲突的，它们会被分配到一组中。因为不管这些交易涉及的账户是什么，它们都会访问可读写的账户间共享数据。下面以一个例子说明策略的执行过程，假设一个区块包含如表 3-1 所示的交易，其中a_i表示可读写的账户间共享数据。

表3-1 某个区块A中的交易集合

交易	账户	元数据中的资源集合
t_1	$[Alice, Bob]$	m_1, a_1
t_2	$[Chad, Helen]$	m_3
t_3	$[Sam, Ean]$	m_3, m_4, a_1
t_4	$[Zack, Chad]$	m_3
t_5	$[Ean, Andy]$	m_2, m_4
t_6	$[John, Chad]$	m_2, a_2
t_7	$[Ada, Ron]$	a_2

分组流程：

1. 针对区块中的每一个交易，按照如下规则标记其访问的资源：

- 如果t_i访问了账户相关的数据m_j，则将$\mathrm{m}_j.Account_1$和$\mathrm{m}_j.Account_2$添加到t_i的资源集合中（假设t_i中相关账户是$Account_1$和$Account_2$）；
- 如果t_i访问了可读写的账户间共享数据a_k，则将a_k添加到t_i的资源集合中；
- 如果t_i访问了只读的账户间共享数据，什么操作也不做，直接跳过。

根据上述规则对上面的区块中的交易进行处理的后结果如表 3-2 所示。

表3-2 某个区块A中的交易集合

交易	标记后的资源集合
t_1	$m_1.Alice, m_1.Bob, a_1$
t_2	$m_3.Chad, m_3.Helen$
t_3	$m_3.Sam, m_3.Ean, m_4.Sam, m_4.Ean, a_1$
t_4	$m_3.Zack, m_3.Chad$
t_5	$m_2.Ean, m_2.Andy, m_4.Ean, m_4.Andy$
t_6	$m_2.John, m_2.Chad, a_2$
t_7	a_2

2. 根据标记的资源集合，进行交易分组。交易分组的结果是要得到一个交易组集合，该集合中每个元素为一个交易组，交易组中的交易均为冲突交易，也即占用相同资源的交易。正如上面有关资源依赖无向图的叙述，在这个无向图中，每个顶点表示的资源，如果两个资源被同一个交易访问，则这两个资源的顶点之间就有一条边。而对于交易来说，如果访问的资源属于同一个连通分量，则它们应该被分到同一个组中。因此根据以上描述，交易分组问题可转化为动态连通性问题，该问题可通过并查集来解决。该动态连通性问题解决过程如下：对于给定的标记后的交易资源占用集合$RS = \{TX_i, i = 1, 2, \dots, n\}$和一个空的交易组集合$RT$，其中$TX_i = \{r_j, j = 1, 2, .., m\}$，$r_j$为$TX_i$占用的某个资源，需要注意的是$TX_i$中的某个资源可能与$TX_j$中的某个资源相同，因为不同交易可能占用相同资源，遍历RS中的资源元素，根据资源是否相同或是否属于同一个交易进行 Union 操作，通过这样的规则实现并查集的构建，构建好的并查集包含多个子集，每个子集都有由一个ID来表示，最后再遍历RS中的交易元素，根据交易占用的某个资源元素查询并查集，找到该资源所在的并查集子集的ID，根据ID查询交易组集合RT，若不存在该ID所对应的交易组，则创建一个包含该交易的交易组，放入RT集合中，并且使得可以通过子集的ID查询得到该交易组，否则直接将交易放入查询到的交易组中。交易分组规则简述如下，根据规则对区块 A 中的交易分组的结果如图 3-2 所示：

- 遍历每个交易标记的资源集合时，创建冲突交易的资源的超集；
- 对于每个交易，如果该交易的某个资源在并查集第 i个超集中，则该交易应该在第i个交易分组中。

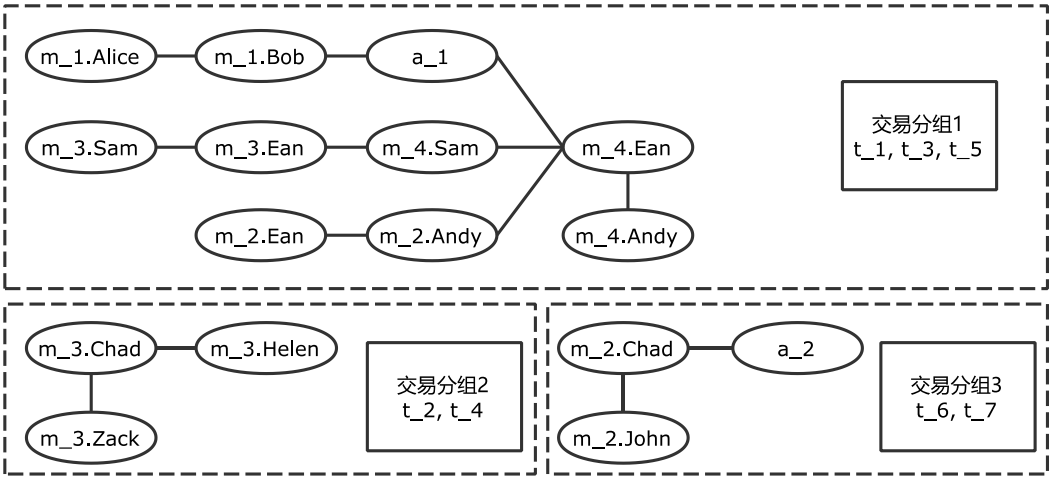

图 3-2 区块 A 的交易分组结果

3.6 分组算法的正确性

假设对于一个交易集合TXS，交易集合中包含TX_1和TX_2，通过分组算法得到两个交易子集（连通分量）CC_1和CC_2，CC_1和CC_2之间是非连通的，则TX_1或TX_2占用的资源不会同时在CC_1和CC_2中。

假设TX_1占用$m_1.Alice$和 $m_1.Bob$两个资源，且TX_1的两个资源分别在CC_1和CC_2中，由于分组算法中定义，如果两个资源被同一个交易访问，则这两个资源的顶点之间就有一条边，那么它们之间是连通的，则CC_1和CC_2是连通的，与假设矛盾，对于TX_2，同理可推出与假设矛盾的结论，则原结论成立。

3.7 交易分组合并

对于任意一组交易，通过上述分组策略，最终得到的交易集合的数量可能会大于区块链节点的 CPU 的核心数，因为对于分组后的交易集合，每一组都会交由一个线程去处理，所以如果结果交易集合的数量大于 CPU 核心数，就会出现多个线程切换，最终会降低交易处理的效率。所以为了避免这种情况的发生，需要对上述策略做出改进，使得最终结果交易集合的数量小于或等于 CPU 核心数。

这里，论文定义一个并发度的概念，也即对于任意交易集合最多可以分多少组。设经上述策略得到的结果交易集合为$UG = \{g_1, g_2, \dots, g_n\}$，其中$g_i$为一个交易集合，集合内的交易均为冲突交易，即它们之间占用了相同的资源。改进策略的目的是要使得$Count(G) \le CL$，且使得$Variance(G)$尽可能小，其中G为最终结果交易集合，CL为并发度，$Count$为统计交易个数的函数，$Variance$为方差函数。

论文中提出两种策略来实现分组合并：最大最小分组合并策略和最小次小分组合并策略，它们的伪码分别如表 3-3 和表 3-4 所示。

表 3-3 最大最小分组合并策略伪码

策略：最大最小分组合并策略
// 输入：未改进分组策略得到的交易集合UG和并发度CL
// 输出：合并后的交易集合G
1. G = {} //初始化结果集合
2. if Count(UG) <= CL
3. return UG
4. SortByDescending(UG) // 根据Count(g_i)降序排序，其中g_i为UG中的元素
5. start = StartIndex(UG), end = EndIndex(UG) // start和end分别指向UG的两端元素
6. while(start <= end)
7. TMP = UG(start) // TMP暂存UG中索引为start的元素
8. while(Count(TMP) + Count(UG(end)) <= Count(UG) / CL && start < end)
9. TMP = Merge(TMP, UG(end)) // 合并TMP与UG(end)
10. end—
11. G.Add(TMP) // 将TMP添加至结果集G中
12. start++
13. if Count(G) > CL
14. SortByDescending(G) // 根据Count(g_i)降序排序，其中g_i为G中的元素
15. while(Count(G) > CL)
16. Min = LastElementOf(G) // 取最后一个元素
17. G.Remove(Min)
18. SubMin = LastElementOf(G) // 取最后一个元素
19. G.Remove(SubMin)
20. TMP = Merge(Min, SubMin) // 合并Min与SubMin
21. G.Insert(TMP) //根据Count(TMP)将TMP插入到合适的位置
22. return G

表 3-4 最小次小分组合并策略伪码

策略：最小次小分组合并策略
// 输入：未改进分组策略得到的交易集合UG和并发度CL
// 输出：合并后的交易集合G
1. G = UG //初始化结果集合
2. if(Count(G) <= CL)
3. return G
4. SortByDescending(G) // 根据Count(g_i)降序排序，其中g_i为G中的元素
5. while(Count(G) > CL)
6. Min = LastElementOf(G) // 取最后一个元素
7. G.Remove(Min)
8. SubMin = LastElementOf(G) // 取最后一个元素
9. G.Remove(SubMin)
10. TMP = Merge(Min, SubMin) // 合并Min与SubMin
11. G.Insert(TMP) //根据Count(TMP)将TMP插入到合适的位置
12. return G

3.8 基于 Actor 模型的交易并行执行

通过上一节分组合并后会得到新的交易集合，这个交易集合中又包含许多交易子集合（交易组），每个子集合中的交易不一定都是冲突的，但冲突的交易一定都在一个交易子集合中。得到这个交易集合后，就可以并行地执行这些交易子集合了，每个交易子集合内的交易串行执行。论文采用 Actor 模型来实现交易执行任务的并行处理，针对每个交易子集合，创建一个 Actor 来执行这个交易子集合中的所有交易。这个过程需要创建多个 Actor，Actor 中任务的分发可以交由 Router 来管理。

Router 是一种特殊的 Actor，它的任务是转发消息给其他的 Actor，这些 Actor 也被称为 Routees。Router 的设计初衷就是用来进行将繁重的任务分发给它的 Routees，这些 Routees 会去真正地执行繁重的任务。在 Akka.Net 中 Router 有两种：Group Router 和 Pool Router。Group Router 并不负责创建与管理它的 Routees，而 Pool Router 本身负责创建与管理它的 Routees。为了有更好的灵活性，论文采用的是 Group Router。Router 如何确定消息转发给它的哪一个 Routee，这就涉及到 Routing Strategies。Routing Strategies 封装了 Router 如何转发消息给它的

Routees 的逻辑。Routing Strategies 有很多种，比如 Broadcast，Random，Round-Robin 等。Broadcast 是指 Router 接收到消息后将消息转发给所有的 Routees，显然它是不符合策略的要求的。Random 是指 Router 接收到消息后将消息随机地转发给它地某个 Routee，它具有随机性，也不太符合要求。Round-Robin 是指 Router 接收到消息后采用轮询的方式转发给 Routees，这对于每个 Routee 都是公平的，能够最大效率地利用 Routee 来处理消息，Round-Robin 的工作原理如图 3-3 所示。

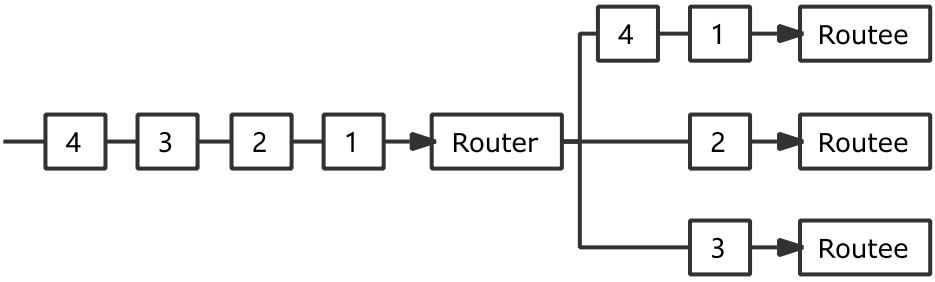

图 3-3 Roun-Robin 工作原理

3.9 交易分组并行化策略

交易分组并行化策略的前置条件是智能合约在编写时使用 SmartContractFieldData 对合约的状态变量进行了标记，即指明变量的名称和变量的访问方式，并使用 SmartContractFunctionData 对函数进行了标记，即指明函数的名称，函数调用其他合约的集合和访问了那些状态变量，同时在部署智能合约时，提取了这些元数据并与智能合约代码一起存入数据库。交易并行化策略伪码如表 3-5 所示，程序流程图如图 3-4 所示。

表 3-5 交易分组并行化策略伪码

策略：交易分组并行化策略

```
// 输入：交易集合TXS和并发度CL
// 输出：合并后的交易集合G
Init RUS // 初始化资源并查集
Init TXFR // 记录每个交易所占的一个资源
Init G // 初始化交易分组集合
for(tx in TXS)
    resources = GetResources(tx) // 获取tx调用的合约函数占用的资源集合
    first = NULL
    for(resource in resources)
        if(RUS.Contains(resource))
            node = RUS.Get(resource)
        else
            node = new Node()
            RUS.Add(resource, node)
        if(first == NULL)
            first == node
            TXFR.Add(tx, resource)
        else
            node.Union(first)
for(tx in TXS)
    first = TXRS.Get(tx)
    nodeId = RUS.Get(first).Find().NodeId
    G.Get(nodeId).Add(tx)
G = Merge(G, CL) // 交易分组合并
Execute(G) // 通过Akka.Net的Actor模型并行化执行分组后的交易
```

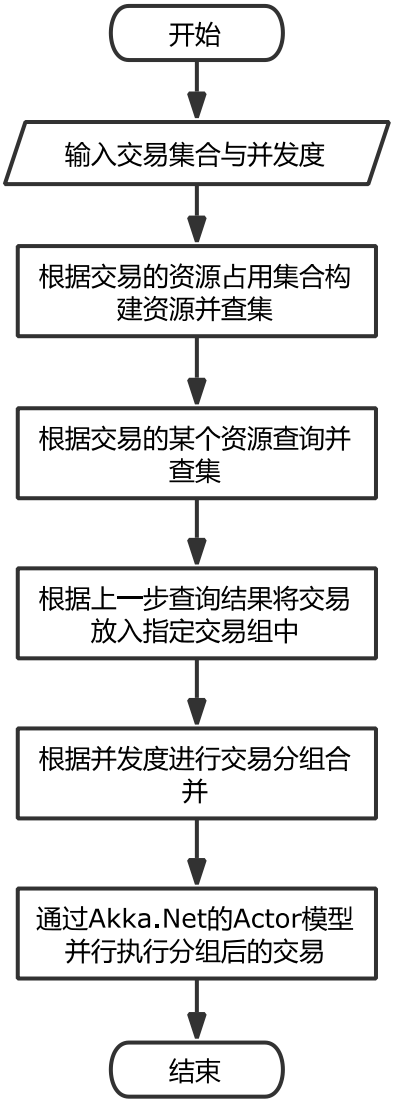

图 3-4 交易分组并行化策略流程图

Step 1：根据交易的资源占用集合构建资源并查集

从交易中提取智能合约的地址和智能合约的函数名称及参数，再通过地址和函数名称获取函数的元数据，根据元数据便可以得到该交易占用的资源集合，即访问了哪些状态变量，然后对资源集合进行标记，构建资源的并查集。

Step 2：根据交易的某个资源查询并查集

根据交易的某个资源查询并查集获取并查集子集的ID。

Step 3：根据上一步查询结果交易放入指定交易组中

通过上一步得到的并查集子集的ID进行分组，具有相同ID的交易放入一组。

Step 4：根据并发度进行交易分组合并

根据并发度和上一步得到的分组后的交易集合使用最大最小策略进行交易分组合并。

Step 5：通过 Akka.Net 的 Actor 模型并行执行分组后的交易

使用 Akka.Net 实现的 Actor 模型，将分组后的交易分成数量等于并发度的执行任务交给 Actor 来执行。

3.10 实验及分析

3.10.1 实验环境描述

硬件环境：

- 处理器：Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
- 硬盘：450G
- 内存：8G

操作系统：Ubuntu 16.04 LTS

编程语言：C#，Shell

编译环境：Rider 2018.2

3.10.2 分组合并策略实验对比

最大最小分组合并策略和最小次小分组合并策略均可以解决分组合并的问题，但是它们合并后结果并不相同，也即最终得到的结果集的方差并不相同。为了选择最优的分组合并策略，该论文进行了一组对比实验。上述两个分组合并策略中并没有使用交易内部数据，而只使用了交易集合的个数，因此这里分组合并策略可以转化为对于给定的一个自然数序列，如何将其中的元素进行合并，使得合并后的自然数序列个数为某个定值且自然数序列的方差尽可能小。这个自然数序列具有很大的随机性，该论文采用随机数序列检测两种分组合并策略的合并效果。实验的评价标准为结果集的方差，方差的计算方法如公式(3-3)所示。

$$Var(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{3-3}$$

其中X为序列，N为序列的个数，x_i为序列的第i个元素，$\mu = \frac{1}{N}\sum_{i}^{N} x_i$为序列的均值。论文随机生成包含 500 个元素的随机数序列且设置不同的并发度，然后通过两种合并策略进行实验对比，两种分组合并策略的实验对比结果如图 3-5 所示。

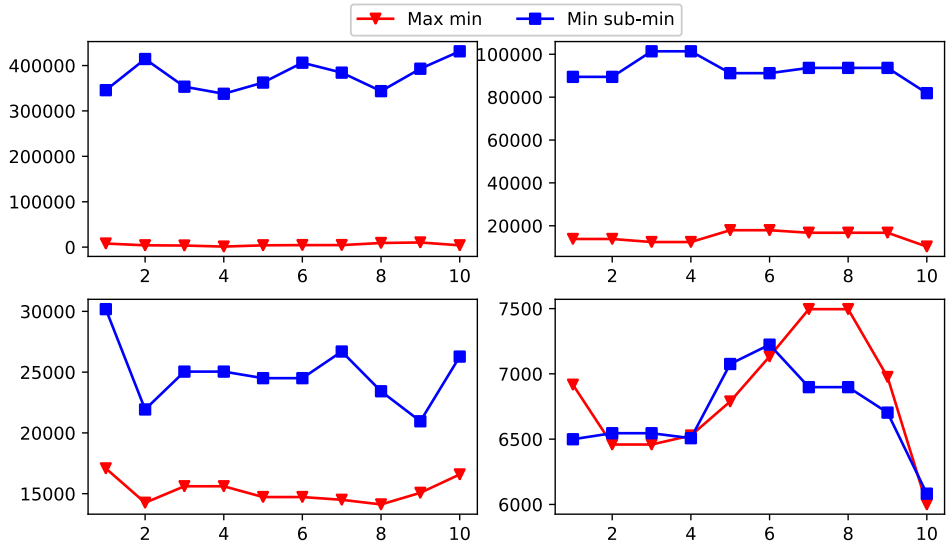

图 3-5 最大最小分组合并策略和最小次小分组合并策略的对比结果

图 3-5 中左上，右上，左下，右下的并发度分别为 8，16，32，64，并且随机数序列包含 500 个元素，纵坐标为方差大小，横坐标为实验次数。从图 3-5 中可以看出，随着并发度的增加，最大最小分组合并策略和最小次小分组合并策略所得结果序列的方差是急剧下降的，且最大最小分组合并策略所得结果序列的方差相比最小次小分组合并策略所得结果序列的方差更小。在并发度较高时，最小次小分组合并策略所得到的结果序列的方差会更小，但随着并发度逐渐趋近于原本序列的个数时，两种合并策略所得结果序列的方差是相同的。为了保证得到的结果序列的方差尽可能小，可采取混合的方式来得到最优结果，即分别计算两种策略结果序列的方差，选取方差最小的结果序列，考虑到论文在实验过程中使用并发度不是很高，论文采用的是最大最小分组合并策略。

3.10.3 分组并行化策略实验分析

Benchmark 是一个模拟程序，主要用于模拟区块的执行过程，也即区块中智能合约的执行过程。在实验中，论文中使用了一个简单的 Token 合约，合约主要用于 Token 的转账，余额查询等 Token 的基础功能。实验的评价标准为交易的处理速度，即 TPS，计算方法如公式(3-4)所示。

$$TPS = \frac{Count(TXS)}{t} \tag{3-4}$$

其中$Count(TXS)$为待处理交易集合的数量，t为处理完所有交易所使用的时间。在下面的实验中会用到一个概念：交易倾斜率（Transaction Tilt Rate）。论文将

其定义为：包含最大数量的冲突交易组所占所有交易数的比例，计算方法如公式(3-5)所示。

$$TTR = \frac{Count(S)}{N}, S \subset TXS \tag{3-5}$$

其中TXS为所有交易的集合，S为冲突交易的集合且其冲突交易的数量是TXS所有冲交易子集中最大的，N为TXS中交易的个数。

论文的实验是基于 AElf 的，由于 AElf 测试网数据并不是很多，论文中的数据采用随机生成的方式，交易账户是随机生成的，冲突交易均为多个账户给同一个账户进行转账 Token。在交易执行过后，都会对交易执行结果进行验证，比如验证交易执行后的账户余额是否符合预期。论文中设置了三组实验来比较交易分组并行化策略的执行效率与交易串行执行策略的执行效率：不同交易数量下的交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比，不同交易分组下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比，以及不同交易倾斜率下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比。

表 3-6 不同交易数量下的交易分组并行化策略的执行效率与交易串行执行策略的执行效

交易数量/TPS	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Serial	1675	1565	1514	1463	1579	1523	1483	1540	1511	1526
TGP	4046	3977	4026	4066	4011	3992	4124	4221	3926	3991

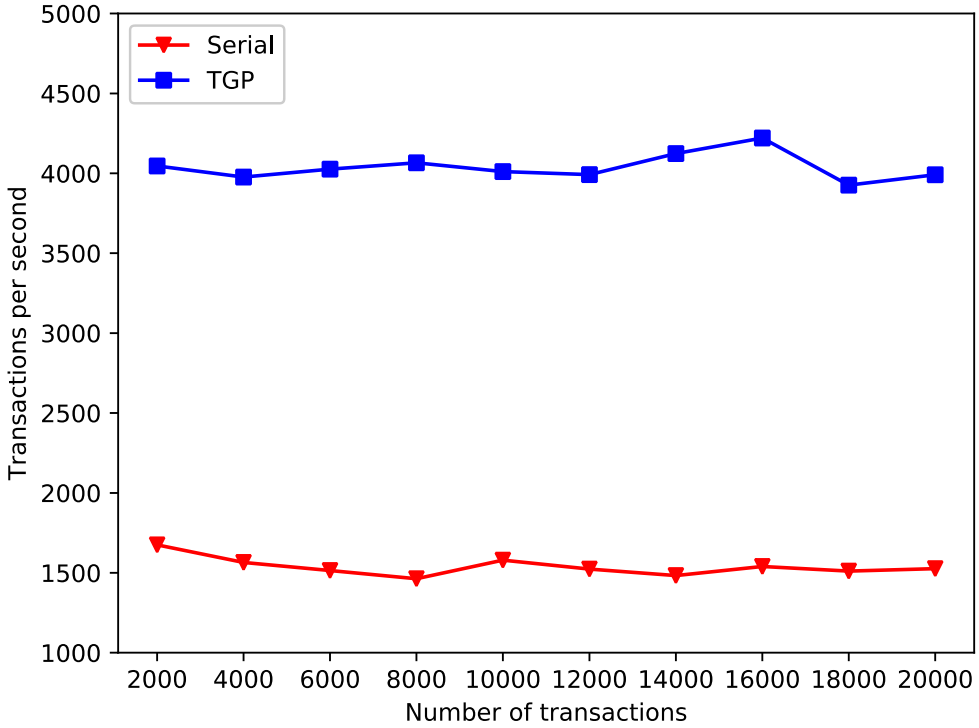

图 3-6 最大最小分组合并策略和最小次小分组合并策略的对比结果

不同交易数量下的交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比实验如表 3-6 和图 3-6 所示，实验中并发度为 8，所以 Actor 模型中的 Router 管理了 8 个 Routee。从表 3-6 和图 3-6 中可以看出，交易串行执行策略在不同交易数量下 TPS 稳定在 1537 左右，而交易分组并行化策略的 TPS 稳定在了 4038 左右，交易分组并行化策略的 TPS 提升至交易串行执行策略 TPS 的 2.6 倍。

表 3-7 不同交易分组下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比

交易组数 /TPS	2	4	6	8	10	12	14	16	18	20
Serial	1579	1579	1579	1579	1579	1579	1579	1579	1579	1579
TGP	3447	4057	4729	4877	5034	4632	4679	4720	4867	5070
Max Group	5000	2500	1666	1250	1000	1666	1428	1250	1110	1000

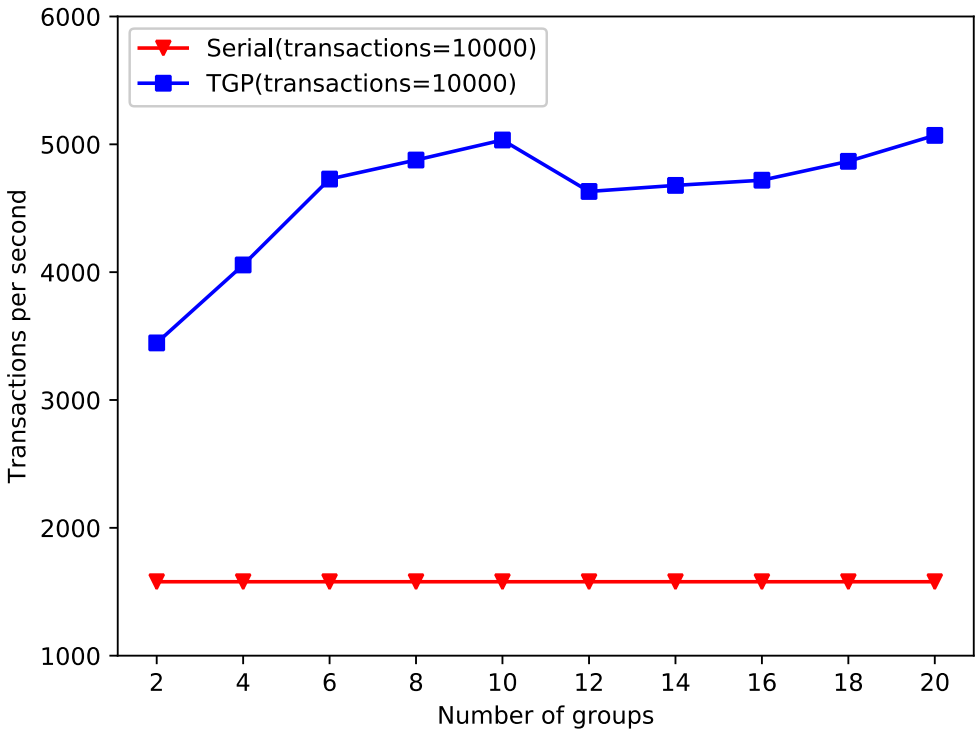

图 3-7 不同交易分组下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比

不同交易分组下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比实验如表 3-7 和图 3-7 所示，实验中并发度为 10，所以 Actor 模型中的 Router 管理了 10 个 Routee。不同交易分组中的“组”指的给定交易集合经交易分组后并未分组合并之前得到的组数，也即冲突交易组的数量。从表 3-7 和图 3-7 中可以看出当冲突交易组数不超过 10 时，说明此时交易经过分组后交易组数不会超过 10，TPS 会随着冲突交易组数的增加而增加。当冲突交易组数超过 10 时，因为交易数量是一定的，此时交易经过分组后会出现交易分组合并，那么结果交易组中交易的数量不相同，也即每个 Routee 执行的交易量并不相同，这会导致交易执行时间相比冲突交易组数小于 10 的情况有所增加，TPS 下降，但随着冲突交易组数的增加，Max Group（分组合并后结果交易集合中包含最多交易的集合的交易数）会减小，这会使得交易执行时间减少，TPS 增加。

表 3-8 不同交易倾斜率下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比

交易倾斜率/TPS	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Serial	1579	1579	1579	1579	1579	1579	1579	1579	1579	1579
TGP	4458	3947	3106	2807	2569	2245	2108	1962	1862	1579

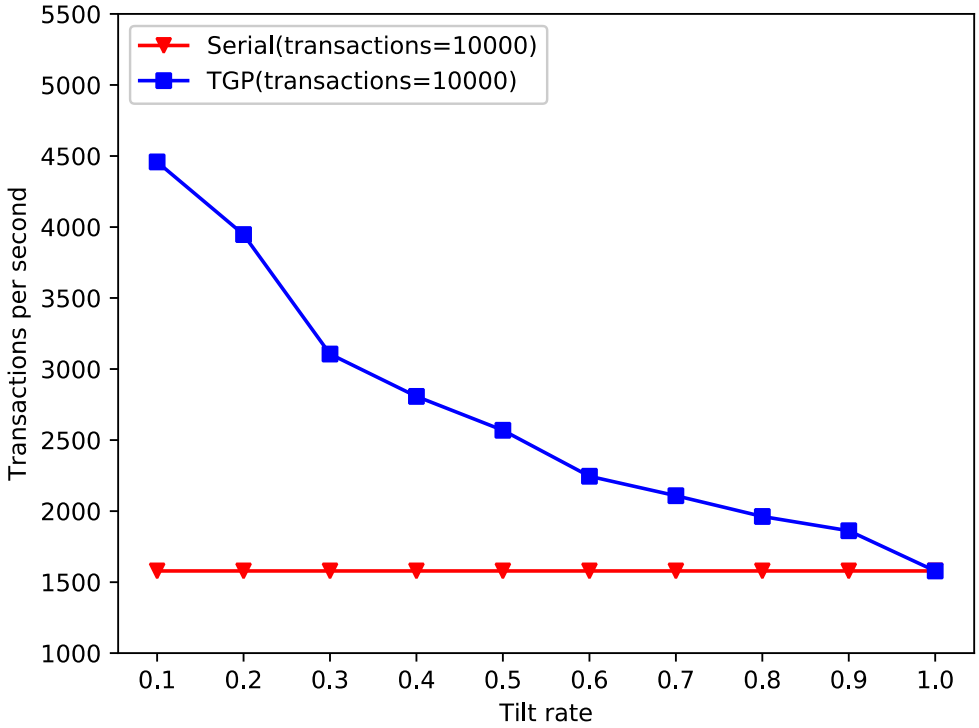

图 3-8 不同交易倾斜率下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比

不同交易倾斜率下交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比实验如表 3-8 和图 3-8 所示，实验中并发度为 8，所以 Actor 模型中的 Router 管理了 8 个 Routee。从表 3-8 和图 3-8 中可以看出，交易串行执行策略在不同交易倾斜率下 TPS 是不变的，因为它对所有的交易都是串行执行的，交易倾斜率对它并没有影响。而交易分组并行化策略的 TPS 随着交易倾斜率的增加，TPS 一直在下降，因为随着交易倾斜率的提升，分组后的某一组交易占比也增大，这将导致处理这一组交易所用时间大大增加，因而策略的整体 TPS 也

随之下降，当交易倾斜率增加至 1 时，说明此时所有的交易都是冲突的，它们经过交易分组后，交易都在同一组，所以与交易串行策略的执行效率是一样的。

3.11 本章小结

现有区块链为了保证安全性，防止因并行处理交易而可能产生数据不一致的问题，均采用了串行处理区块中交易的方式，这种方式的低效导致区块链的 TPS 无法得到有效提高，也致使区块链无法在一些对系统吞吐量要求很高的领域得到应用。本章提出的交易分组并行化策略，在交易执行前，根据交易中调用的智能合约地址和函数名称获取函数的元数据，根据元数据便可得知交易的资源占用情况，然后根据资源占用情况构建资源占用的并查集，再根据交易所在的并查集将交易进行分组，冲突的交易将会分至同一个交易组，如果分组结果中交易组数不大于区块链节点的 CPU 的核心数，则直接并行化处理各个冲突交易组，否则需要进行分组合并再并行化处理合并后的交易组。本章定义了交易冲突与智能合约的数据类型，介绍了元数据在策略中的应用和策略的前置条件，并通过反证法验证了分组策略的正确性，该章还提出了两种分组合并的策略，最后又通过实验比较了两种策略的优越性，本章后面介绍了 Actor 模型在交易并行执行时的应用，最后通过实验验证了交易分组并行化策略对区块链 TPS 的提升。

第4章 基于 Actor 集群的交易分组并行化策略

随着区块链技术的发展，将来区块链会在很多领域得到应用，比如金融领域，这对区块链的 TPS 要求将越来越高。同时随着计算机硬件的快速发展，将来区块链的单个节点有可能将由集群构成，即便是现在，有些公链已经采用超级节点，该超级节点就可能是由集群构成的，所以充分利用集群的性能提升现有区块链交易的执行效率是很有必要的。

4.1 Actor 集群

第三章中的交易分组并行化策略是基于 Actor 模型的，其中的 Actor 模型是使用 Akka.Net 实现的，不过它是基于单机的。单机的 CPU 核心数毕竟有限，假如大量交易经过分组后，结果交易组数远远大于策略设置的并发度，因此需要一个交易分组合并的过程，这将降低交易分组并行化策略的效率。因此，为了提高交易分组并行化策略的效率，可以提高策略的并发度，同时需要提高 CPU 的核心数，也即使用计算机集群。

计算机集群简称集群，是一种计算机系统，它包含许多计算机，这些计算机可以在一个局域网内，也可以不在一个局域网内，它通过一些计算机软件或硬件使得这些分散的计算机协同工作。从某种意义上来说，计算机集群可以被视作是一台计算机，它的内部通讯对外界是透明的。通常可以将一些需要耗费大量计算资源的任务交给集群来做，集群会将任务分散为小型任务，交由集群中的单个节点来快速完成。集群与单机相比，集群不止在处理信息的吞吐量方面有很大提升，它还具有两个特性：可扩展性和高可用性。可扩展性是指集群可以动态地添加节点，从而提升集群的整体性能。高可用性是指集群中某个节点的宕机并不会影响整个集群，集群对外依然是正常工作的。

论文中使用的是 Akka.Net 的集群模块，通过该模块，所有集群中节点自动发现新的节点，并在不需要修改配置的情况下自动删除宕机节点，还可以创建 Actor 集群，通过 Router 来管理。Actor 集群中节点分为两类：种子节点与非种子节点。种子节点是集群形成的联络站，非种子节点最初若要进行通信，首先需要连接种子节点，通过种子节点才能发现其他非种子节点。

4.2 Actor 集群的建立过程

集群最初由两部分组成，种子节点与非种子节点。种子节点的位置对所有非种子节点都是公开的，非种子节点的位置都是未知的，只有在建立集群的过程中才慢慢得知。Actor 集群建立的过程：

1. 非种子节点尝试连接种子节点；

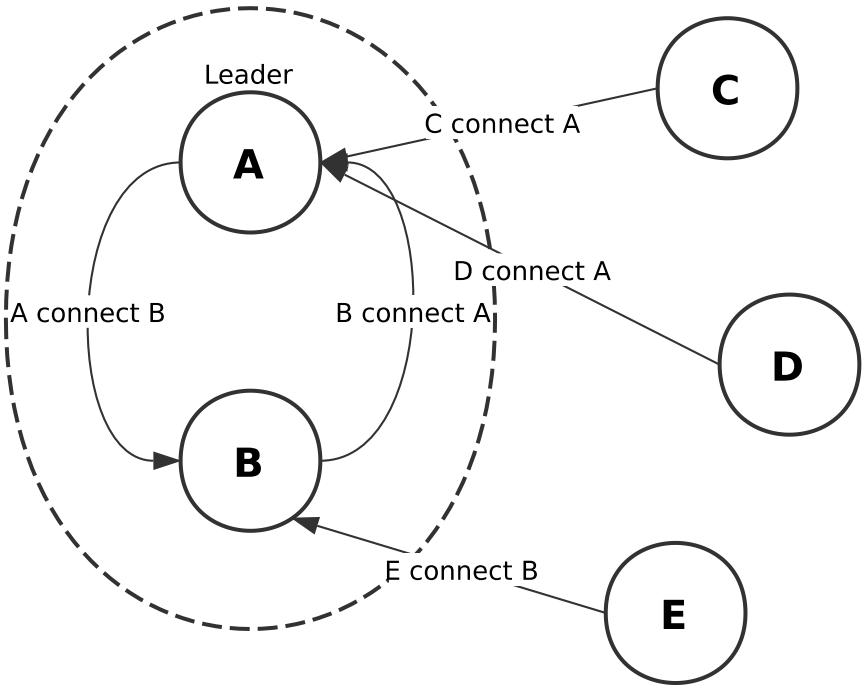

图 4-1 集群建立第一阶段

假设有 5 个节点 A，B，C，D，E，A，B 为种子节点，C，D，E 为非种子节点，初始状态如图 4-1 所示，此时 A，B 节点互相知道对方的信息，而 C，D 节点知道种子 A 节点的信息，E 节点知道 B 节点的信息。

2. Leader 选举，标记已加入集群节点的状态为 Up；

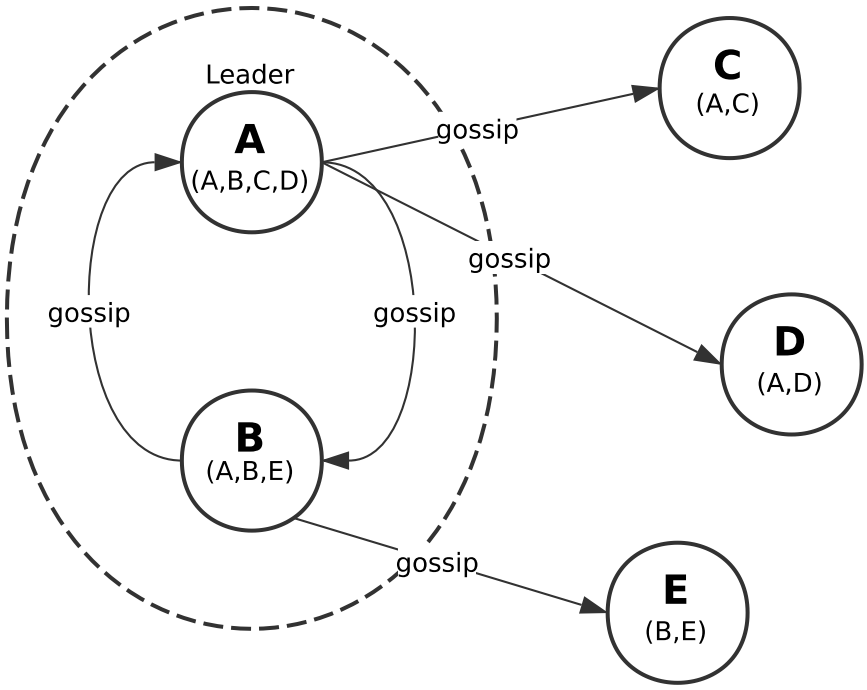

图 4-2 集群建立第二阶段

在集群内部通信开始时，会选举出一个 Leader，由它来标记已加入集群节点的状态为 Up。通过选举策略 A 节点被选举为 Leader，A 节点开始将它所知的 B，C，D 节点标记为 Up 状态，但随着下一阶段 Gossip 的传播，A 节点也会将 E 节点标记为 Up 状态，标记过程如图 4-2 所示。

3. 通过 Gossip 的传播，所有节点已知道所有节点的信息，集群建立完成。

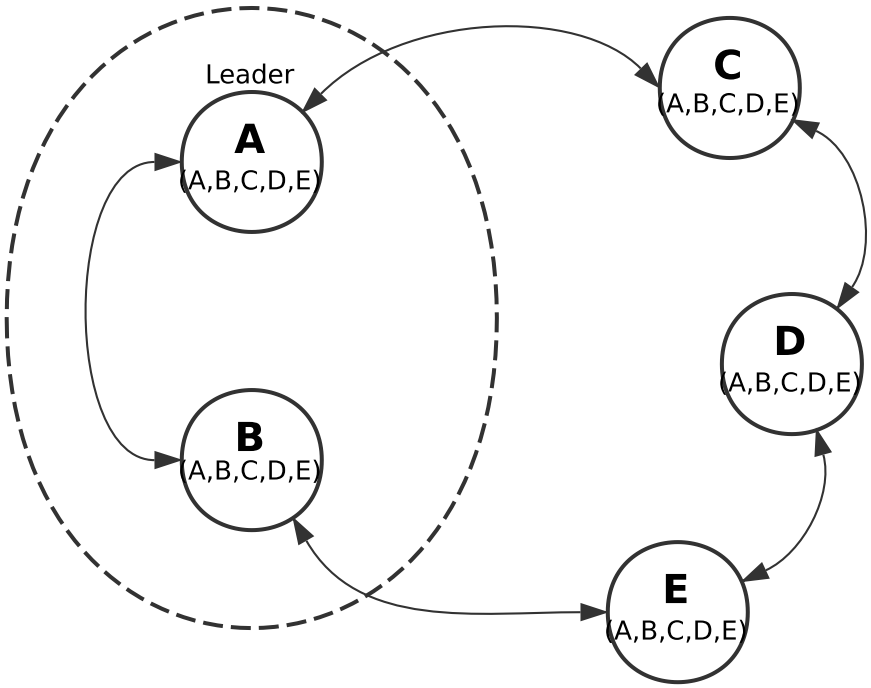

图 4-3 集群建立第三阶段

通过 Gossip，所有节点的信息逐渐共享出去，Leader 也会将所有节点的状态标记为 Up，至此每个节点都与其他节点建立了连接，集群建立完成，现在每个节点都可接收用户自定义的集群操作。集群形成第三阶段如图 4-3 所示。

Actor 集群形成以后具有很好的可靠性，节点之间会不断地互相发送心跳包，如果一个节点错过了足够多地心跳，这将导致 Unreachable Gossip 传播到网络中，Leader 节点会等待该节点恢复正常状态。如果所有节点都收到 Unreachable Gossip，并且同意该节点是 Unreachable，则 Leader 节点可以将该节点标记为 Down，并从集群中移除该节点。

Actor 集群建立之后，便可以通过 Router 管理集群中的 Actors，即 Routees。在此基础之上论文提出基于 Actor 集群的交易并行化策略，首先使用第三章中的交易分组策略将交易进行分组，然后根据并发度进行交易分组合并，最后使用 Actor 集群执行交易。

4.3 基于 Actor 集群的交易分组并行化策略

本章主要解决交易分组并行化策略在单机上的限制，因而需要修改交易执行的环境，将单机环境更改为集群环境。基于 Actor 集群的交易分组并行化策略的流程图如图 4-4 所示。

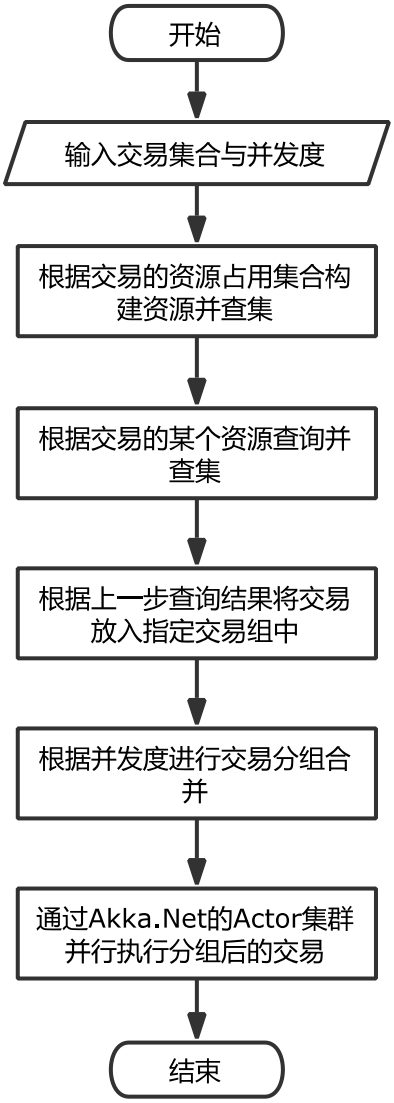

图 4-4 基于 Actor 集群的交易分组并行化策略流程图

Step 1：根据交易的资源占用集合构建资源并查集

从交易中提取智能合约的地址和智能合约的函数名称及参数，再通过地址和函数名称获取函数的元数据，根据元数据便可以得到该交易占用的资源集合，即访问了哪些状态变量，然后对资源集合进行标记，构建资源的并查集。

Step 2：根据交易的某个资源查询并查集

根据交易的某个资源查询并查集获取并查集子集的ID。

Step 3：根据上一步查询结果交易放入指定交易组中

通过上一步得到的并查集子集的ID进行分组，具有相同ID的交易放入一组。

Step 4：根据并发度进行交易分组合并

根据并发度和上一步得到的分组后的交易集合使用最大最小策略进行交易分组合并。

Step 5：通过 Akka.Net 的 Actor 集群并行执行分组后的交易

使用 Akka.Net 构建 Actor 集群，通过 Router 管理 Actor 集群，将分组合并后的交易执行任务交给 Router，由 Router 将执行任务分发给 Actor 集群来执行。

4.4 实验及分析

4.4.1 实验环境描述

硬件环境：

- Docker 容器数量：8
- 处理器：Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
- 硬盘：450G
- 内存：8G

操作系统：Ubuntu 16.04 LTS

编程语言：C#，Shell

编译环境：Rider 2018.2

4.4.2 基于 Actor 集群的分组并行化策略实验分析

论文中设置了三组实验来比较基于 Actor 集群的交易分组并行化策略的执行效率与交易分组并行化策略以及交易串行执行策略的执行效率：不同交易数量下的基于 Actor 集群的交易分组并行化策略的执行效率与交易分组并行化策略以及交易串行执行策略的执行效率的对比，不同交易分组下基于 Actor 集群的分组并行化策略的执行效率与交易串行执行策略的执行效率的对比，以及不同交易倾斜率下基于 Actor 集群的交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比。

表 4-1 不同交易数量下 ACGP，TGP，Serial 执行效率的对比

交易数量/TPS	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Serial	1675	1565	1514	1463	1579	1523	1483	1540	1511	1526
TGP	4046	3977	4026	4066	4011	3992	4124	4221	3926	3991
ACGP	5000	5151	5503	5465	5402	5588	5063	5158	4911	5079

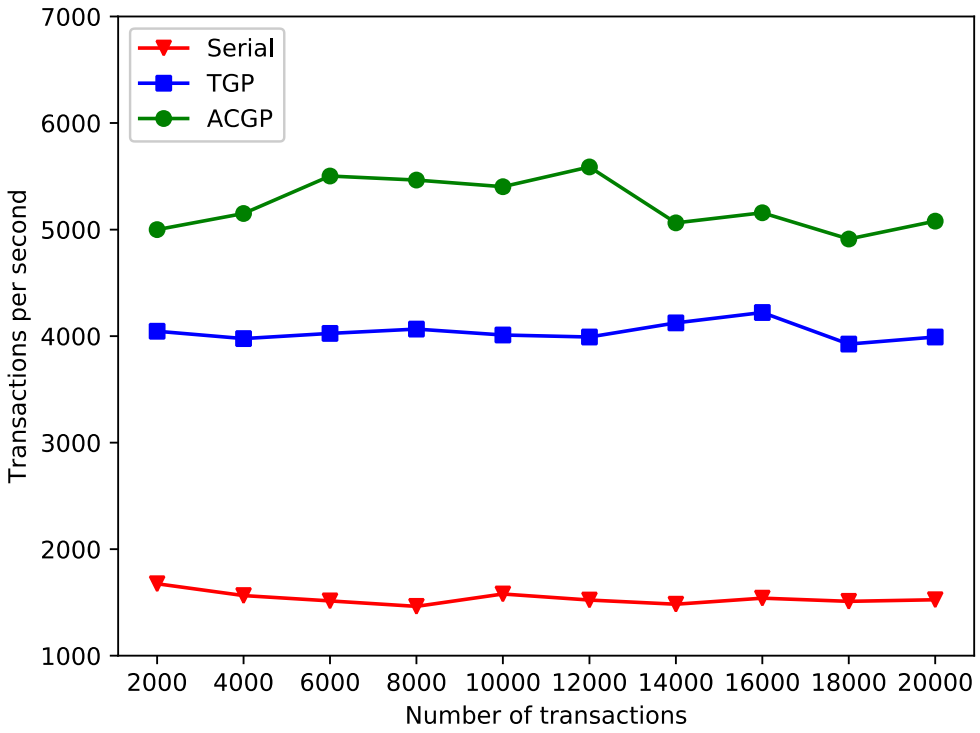

图 4-5 不同交易数量下 ACGP，TGP，Serial 执行效率的对比

不同交易数量下的基于 Actor 集群的交易分组并行化策略的执行效率与交易分组并行化策略以及交易串行执行策略的执行效率的对比实验如表 4-1 和图 4-5 所示，实验中的并发度设置为 64，Router 管理的 Routee 数量为 64。从表 4-1 和图 4-5 中可以看出，交易串行执行策略在不同交易数量下 TPS 稳定在 1537 左右，交易分组并行化策略的 TPS 稳定在了 4038 左右，而基于 Actor 集群的交易分组并行化策略的 TPS 稳定在了 5232 左右，是交易分组并行化策略 TPS 的 1.3 倍，交易串行执行策略 TPS 的 3.4 倍。

表 4-2 不同交易分组下 ACGP，Serial 执行效率的对比

交易组数/TPS	8	16	24	32	40	48	56	64	72	80
Serial	1579	1579	1579	1579	1579	1579	1579	1579	1579	1579
ACGP	4954	4957	4966	4991	5026	5173	5250	5376	4781	4785

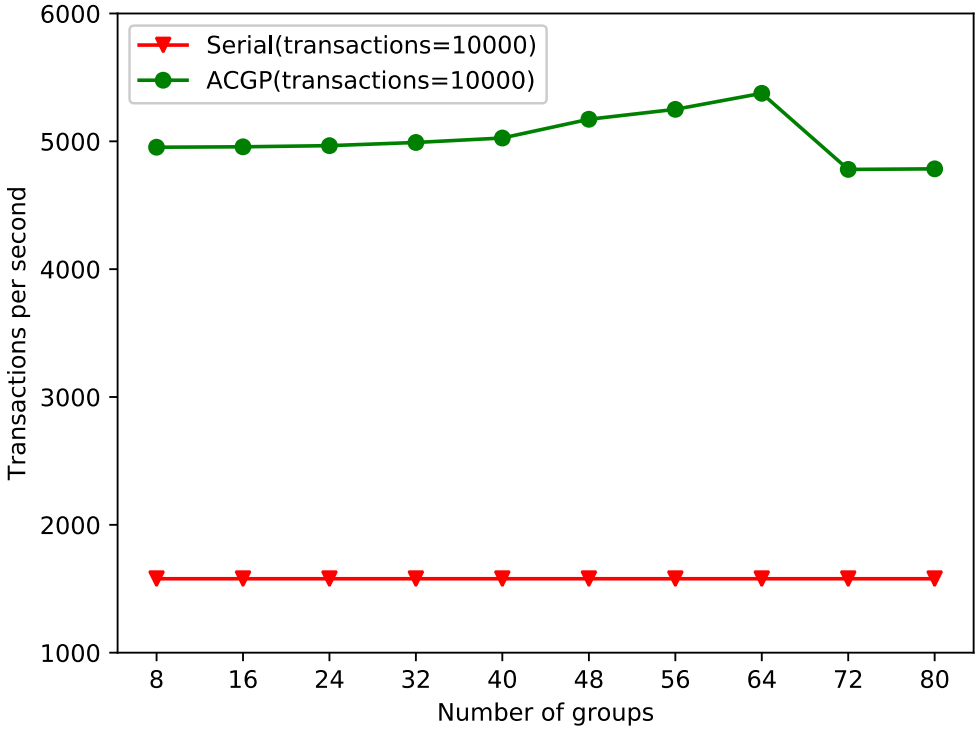

图 4-6 不同交易分组下 ACGP，Serial 执行效率的对比

不同交易分组下基于 Actor 集群的分组并行化策略的执行效率与交易串行执行策略的执行效率的对比实验如表 4-2 和图 4-6 所示，该实验中的并发度为 64，Router 管理的 Routee 数量为 64。从表 4-2 和图 4-6 中可以看出当冲突交易组数不超过 64 时，说明交易经过分组后交易组数不会超过 64，此时，分组结果不需要进行分组合并，TPS 会随着冲突交易组数的增加而增加。当冲突交易组数超过 64 时，此时交易经过分组后会出现交易分组合并，由于交易数量一定，此时会出现结果交易组中交易的数量不相同，也即每个 Routee 执行的交易量并不相同，这会导致交易执行时间相比冲突交易组数小于 64 的情况有所增加，TPS 下降，但随着冲突交易组数的增加，每组执行的交易量降低，这会使得交易执行时间减少，TPS 增加。

表 4-3 不同交易倾斜率下 ACGP，Serial 执行效率的对比

交易倾斜率/TPS	$\frac{1}{64}$	$\frac{8}{64}$	$\frac{16}{64}$	$\frac{24}{64}$	$\frac{32}{64}$	$\frac{40}{64}$	$\frac{48}{64}$	$\frac{56}{64}$	1
Serial	1579	1579	1579	1579	1579	1579	1579	1579	1579
ACGP	5296	4215	3313	3175	2780	2335	2210	1692	1579

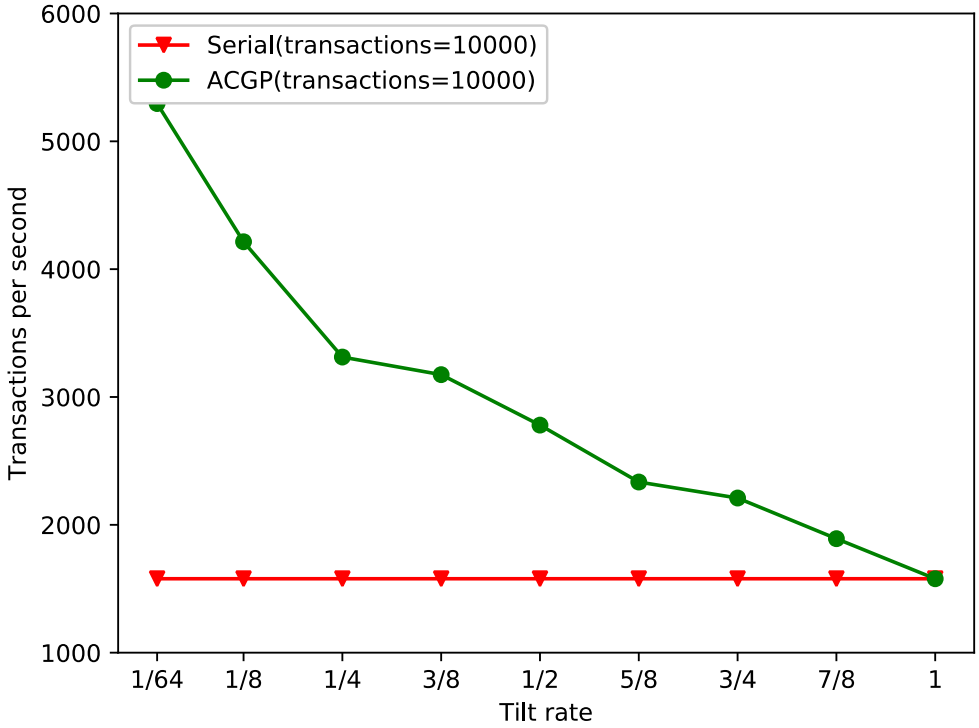

图 4-7 不同交易倾斜率下 ACGP，Serial 执行效率的对比

不同交易倾斜率下基于 Actor 集群的交易分组并行化策略的执行效率与交易串行执行策略的执行效率的对比实验如表 4-3 和图 4-7 所示，该实验中的并发度为 64，Router 管理的 Routee 数量为 64。从表 4-3 和图 4-7 中可以看出，交易串行执行策略在不同交易倾斜率下 TPS 是不变的，因为它对所有的交易都是串行执行的，交易倾斜率对它并没有影响。而基于 Actor 集群的交易分组并行化策略的 TPS 随着交易倾斜率的增加，TPS 一直在下降，因为随着交易倾斜率的提升，分组后的某一组交易占比也增大，这将导致处理这一组交易所用时间大大增加，因而策略的整体 TPS 也随之下降，当交易倾斜率增加至 1 时，说明此时所有的交易都是冲突的，它们经过交易分组后，交易都在同一组，所以与交易串行策略的执行效率是一样的。

4.5 本章小结

本章主要从交易分组并行化策略对区块链 TPS 的提升程度受到单机性能的限制出发，提出基于 Actor 集群的交易分组并行化策略。因为区块链节点机器的性能毕竟有限，而且现有的某些区块链的节点已经开始采用集群来运行，所以可

以通过集群的高性能提高并发度，减少交易分组合并的机率，最大限度地利用集群的性能，从而提升区块链的 TPS。基于这样的思想，论文第四章提出基于 Actor 集群的交易分组并行化策略，由于集群具有更多的 CPU 核心数，策略的并发度将可以变得更高，从而减少交易分组合并的机率，最终每个 CPU 核心将执行更少的交易，从而减少交易的处理时间，达到提高区块链 TPS 的目的。本章还讲述了 Actor 集群的建立过程，以及基于 Actor 集群的并行化策略实现的具体细节，最后通过实验来验证策略对于区块链 TPS 的提升。

第5章 总结与展望

5.1 总结

区块链的本质是一个所有参与节点共同维护，公开透明的分布式账本，但并非某个单独的用户可以对它随意修改，只有通过共识机制选出的区块链节点才可对其更改。区块链的去中心化，不可篡改，数据加密等特性在人与人之间建立了信任桥梁。由于区块链技术的出现，智能合约得到了更好的应用。智能合约要实现其“在没有第三方的情况下进行可信交易，这些交易可追踪且不可逆转”的功能，所以区块链为其提供了一个非常合适的运行环境。一方面，区块链的结构特点决定了智能合约的内容（代码）可追踪且不可篡改，另一方面，区块链上能够加载数字资产，承载价值，使得智能合约有更多的应用场景。

智能合约若想在更多的领域应用，还需提高它的执行效率。智能合约是通过区块链交易来实现调用的，因而提升智能合约的执行效率也即提高区块链交易的处理速度。为了提升交易的处理效率，论文第三章提出交易分组并行化策略，将非冲突交易并行处理。策略中的智能合约平台是基于 AElf 的，它的智能合约是支持元数据的。策略的前置条件是智能合约在编写时需要使用元数据对智能合约状态变量和函数进行标记，标记智能合约状态变量的访问方式，标记函数访问了哪些状态变量，在部署时将智能合约的元数据与合约代码一起存入区块链数据库中。策略在处理交易时，首先提取交易所调用的智能合约地址，智能合约函数及参数，通过合约地址和函数名称获取函数的元数据，然后根据元数据对函数占用的资源进行标记，之后根据资源标记的结果构建资源的并查集，再通过函数（交易）占用的某个资源查询交易在并查集的哪个子集中，根据所在子集的*ID*将交易放入对应分组中，实现所有交易的分组。交易分组后，由于并发度的限制，如果分组结果的交易组数大于了并发度，还需要对分组结果进行交易分组合并，并使得合并后每组交易数量的方差尽可能小。为此引入两个分组合并的策略，最终选取选区方差最小的分组合并结果。分组合并后，将结果交由 Akka.Net 实现的 Actor 模型执行。实验证明，通过使用交易分组并行化策略，交易的处理速度得到了很大提升。

交易分组并行化策略对计算机性能有一定的要求，因为单机的并发度太低，在一定程度上会限制策略的效率，为此，论文第四章在交易分组并行化策略基础上引入集群，这样便可以提高策略的并发度，减少交易分组合并的机率，使得

Actor 处理更少的交易，提高策略的效率。经实验证明，基于 Actor 集群的交易分组并行化策略，交易的处理速度再度得到了提升。

5.2 展望

论文的工作主要围绕智能合约交易分组并行化执行与基于 Actor 集群的交易分组并行化执行两个方面。从前期的工作效果可以看出，论文提出的交易分组并行化策略，通过对冲突交易与非冲突交易进行分组实现了组间交易执行的并行化，提高了交易的执行效率。由于交易分组并行化策略只局限于单机，随着并发度的提升，策略效率将不再提升，在策略基础上引入集群在一定程度上解决了这个问题。后续工作将围绕以下几点进行展开：

（1）交易分组并行化策略在对交易进行分组后，组间交易是可以并行执行的，但组内交易是串行执行的，这在一定程度上也限制了策略的效率，比如，待执行交易如果全是冲突交易，那么分组完成后便只有一组交易，全部串行执行会严重降低执行效率，因此可以考虑使用一些类似数据库事务的并发控制手段来实现组内交易的并发执行，从而提高交易执行效率。

（2）论文使用的分组合并策略得到的结果并不是最优的，论文是在兼顾效率与结果方差而选择的，可以考虑寻找更好的分组合并策略，来提高交易分组并行化策略的整体效率。

（3）交易分组并行化策略在交易执行阶段使用的是 Akka.Net 并发框架，它对任务的调度与执行并不是最优的，因为在加入集群后，调度 Router 的调度效果并不是很好，可以考虑设计一个全新的并发模型，专门针对交易分组并行化策略进行优化，最大程度地提高任务的调度与执行效率，从而提高交易分组并行化策略的效率。

（4）论文中对于元数据的静态分析，需要智能合约编写者配合，若要实现智能合约并行化，需要对合约函数和状态变量进行手动标记，这样增加了出错了机率，可以考虑实现在不借助元数据的情况下，对合约代码直接进行静态分析出资源占用情况，以减少合约出错的机率。

参考文献

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system [EB/OL]. Bitcoin official website, 2008[2018-11-01]. https://bitcoin.org/bitcoin.pdf.

[2] 张秋子. 区块链技术在金融行业的应用构想 [D]. [S. l.]: 浙江大学, 2017.

[3] Swan M. Blockchain: Blueprint for a New Economy [M]. O'Reilly, 2015.

[4] Wikipedia contributors. Smart Contract [EB/OL]. Wikipedia, The Free Encyclopedia,2018-11-19[2018-11-01]. https://en.wikipedia.org/w/index.php?title=Smart_contract&oldid=869532500

[5] 周润, 卢迎. 智能合约对我国合同制度的影响与对策 [J]. 南方金融, 2018 (5).

[6] 夏沅. 区块链智能合约技术应用 [J]. 中国金融, 2018 (6).

[7] Buterin V, et al. A next-generation smart contract and decentralized application platform [EB/OL]. GitHub, 2014[2018-11-01]. https://github.com/ethereum/wiki/wiki/White-Paper.

[8] The EOS Team. The EOS.IO Technical White Paper [EB/OL]. GitHub, 2018-03-01[2018-11-01]. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[9] The Zilliqa Team. The Zilliqa Technical Whitepaper [EB/OL]. Zilliqa official website, 2017-08-01[2018-11-01]. https://docs.zilliqa.com/whitepaper.pdf.

[10] Investopedia contributors. Hard Fork [EB/OL]. Investopedia, 2017-12-19[2018-11-01]. https://www.investopedia.com/terms/h/hard-fork.asp.

[11] James, Ray D, Ryan W-Y, et al. Sharding [EB/OL]. GitHub, 2016-05-1[2018-11-01]. https://github.com/ethereum/sharding/blob/develop/docs/doc.md.

[12] Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery [M]. ACM, 2002.

[13] Dickerson T, Gazzillo P, Herlihy M, et al. Adding Concurrency to Smart Contracts [C]. In ACM Symposium on Principles of Distributed Computing, 2017: 303–312.

[14] Sergey I, Hobor A. A Concurrent Perspective on Smart Contracts [J], 2017: 478–493.

[15] Bocchino Jr R L, Adve V S, Adve S V, et al. Parallel programming must be deterministic by default [C]. In Proceedings of the First USENIX conference on Hot topics in parallelism, 2009: 4–4.

[16] Joseph, Poon V, Buterin. Plasma: Scalable Autonomous Smart Contracts [EB/OL]. Plasma official website, 2017-08-01[2018-11-01]. https://plasma.io/plasma.pdf.

[17] Garcia-Molina H, Widom J, Ullman J D. Database System Implementation [M]. China Machine Press, 2010.

[18] Herlihy M, Luchangco V, Moir M, et al. Software transactional memory for dynamic-sized data structures [C]. In Symposium on Principles of Distributed Computing, 2003: 92–101.

[19] Zhang D, Dechev D. Lock-free Transactions without Rollbacks for Linked Data Structures [C]. In ACM Symposium on Parallelism in Algorithms and Architectures, 2016: 325–336.

[20] Ghosh A, Chaki R, Chaki N. A new concurrency control mechanism for multi-threaded environment using transactional memory [J]. Journal of Supercomputing, 2015, 71 (11): 4095–4115.

[21] Zhang A, Zhang K. Enabling Concurrency on Smart Contracts Using Multiversion Ordering [C]. In Asia-Pacific Web, 2018: 425–439.

[22] Baohua, Yang. 区块链技术指南 [EB/OL]. GitBook, 2016-02-01[2018-11-01]. https://yeasy.gitbooks.io/blockchain_guide/content/.

[23] The NEO Team. The NEO Technical White Paper [EB/OL]. NEO official website, 2016-10-01[2018-11-01]. http://docs.neo.org/zh-cn/whitepaper.html.

[24] The AElf Team. The AElf Technical White Paper [EB/OL]. AElf official website, 2018-06-01[2018-11-01]. https://aelf.io/gridcn/aelf_whitepaper_EN.pdf.

[25] Ethereum contributors. Solidity Documentation [EB/OL]. Solidity website, 2018-09-05 [2018-11-01]. http://solidity.readthedocs.io/en/develop.

[26] Wikipedia contributors. Dynamic Connectivity [EB/OL]. Wikipedia, The Free Encyclopedia, 2018-10-05[2018-11-01]. https://en.wikipedia.org/w/index.php?title=Dynamic_connectivity&oldid=868730681.

[27] Wikipedia contributors. Disjoint Set [EB/OL]. Wikipedia, The Free Encyclopedia, 2018-10-05[2018-11-01]. https://en.wikipedia.org/w/index.php?title=Disjoint_sets&oldid=867647012.

[28] Tarjan R E, Van Leeuwen J. Worst-case analysis of set union algorithms [J]. Journal of the ACM (JACM), 1984, 31 (2): 245–281.

[29] Wikipedia contributos. Moore’s law [EB/OL]. Wikipedia, The Free Encyclopedia, 2018-10-05[2018-11-01]. https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=870722479.

[30] Hewitt, Carl, Bishop, et al. A universal modular ACTOR formalism for artificial intelligence [M]. Springer International Publishing, 1973.

[31] Wikipedia contributors. MapReduce [EB/OL]. Wikipedia, The Free Encyclopedia, 2018-10-01[2018-11-01]. https://en.wikipedia.org/w/index.php?title=MapReduce&oldid=867763840.

[32] Amdahl G M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at I [J]. IEEE Solid-State Circuits Society Newsletter, 2007, 12 (3): 19–20.

[33] Wikipedia contributors. Metadata [EB/OL]. Wikipedia, The Free Encyclopedia, 2018-10-01[2018-11-01]. https://en.wikipedia.org/w/index.php?title=Metadata&oldid=869556134.

发表论文和参加科研情况说明

已申请专利：

[1] 基于情绪的文本分类方法，中国，201710283976.8，2017 年 4 月

致　谢

2018 年剩余的时间越来越少，意味着研究生学业即将结束，这一切似乎来得有点快，三个月前还在实习，一转眼就要毕业了，经历了两年半的洗礼，又给自己的人生增添了几分积淀和沉重。回首过去，不管在科研，实习，还是生活中，总有帮助你，引导你，激励你的人，感谢他们伴你走到今天。

论文课题的研究工作是在我的导师侯庆志副教授的悉心指导下顺利完成的，侯老师总是那么富有干劲，对待科研总是孜孜不倦，治学态度严谨，对工作十分负责认真，不明白的地方总是讲解多次，很耐心，对待学生很负责，修改论文时，总是很认真，逐字推敲和修改，拥有多样的指导方式，为我以后的工作和生活指明了前进方向。

感谢我的指导老师王建荣副教授，对待科研有自己的独到见解，对问题总是能找到突破点，切中要害，治学严谨的作风一直激励着我，让我受益匪浅。生活中的您总是那么和蔼可亲，像一盏明灯照亮我前行的路。

同时也要感谢喻梅老师、于瑞国老师和应翔老师在日常工作中给予我很多指导和帮助，老师们仔细而认真的态度使我受益良多。感谢整个实验室的同学们在我科研工作期间给予我的帮助，尤其是感谢徐天一师兄和高洁师姐、赵满坤学长在求学的道路上指出问题并帮助我解决问题，在日常生活中耐心而风趣地陪伴我成长；感谢同级的多位同学，我从他们身上得到了许多经验，也感谢他们在平时的照顾与陪伴；也感谢师弟师妹在日常上课之余帮助我处理问题，使我意识到了团队合作的重要性。

感谢好扑信息科技有限责任公司的前同事们在技术上给予的支持与帮助，他们耐心的指导使我顺利地完成了实验。

衷心感谢评阅、评审论文和出席论文答辩会的各位专家在百忙之中给予的悉心指导。