比原链CTO姚骋鼎力推荐。

从基础架构到代码实践，深度解析公链技术原理和创新设计，包括应用层、网络层、共识层、智能合约层，提升区块链底层技术实现的神秘面纱。

GO语言
公链开发实战

郑东旭 杨明珠 潘登瑜 崔萌 ◎编著

机械工业出版社
China Machine Press

比原链CTO陈维我力推荐。

从基础架构到代码实践，深度解析公链技术原理和功能设计，包括应用层、内核层、网络层、智能合约层等，揭开区块链底层技术实现的神秘面纱。

GO语言
公链开发实战

邓东旭 杨明辉 潘晓瑜 郭萌 ◎编著

机械工业出版社
China Machine Press

版权信息

书名：GO语言公链开发实战
作者：郑东旭 等
排版：小暑暑
出版社：机械工业出版社
出版时间：2019-06-01
ISBN：9787111629870

推荐序一

我与本书作者同在中国较早的一个公有区块链开源技术社区，同样学习和研究区块链，书中的区块链实例也来源于此，我想我可以简单说几句。

区块链最早源于极客社区点对点电子现金论文，时至今日已有十年。早年的开发应用及技术著作均围绕电子现金类的应用展开。即便有链以及应用，也多是介绍联盟链的应用，较少涉及公链类应用。想要深入学习比较困难，当时业界开发类书籍仅有一本《Mastering Bitcoin》为指路明灯。我早年也曾应《程序员》杂志约稿写过一篇文章《区块链产品三定律》，其中罗列了当时市面流行的六七种应用案例，但总体而言，大环境不完善，产业并不繁荣，产品及技术方面的学习颇为不便，国内从事开发、研究的人员也相当稀少，仅二三十人，中文技术类图书也极度短缺。

技术的可贵之处在于，不仅展现了人类的精湛设计，而且提供了解决问题后所带来的改变。区块链之所以备受瞩目，是因为它创造了一种新的范式，能连接资产服务所涉及的各参与方，能够打破数据孤岛，提高安全性，增强风险控制能力，保护隐私，降低交易成本的同时带来收益。

在资本和产业的热捧中，已出现了各种区块链应用。我们很高兴看到本书作者团队以比原链为实例，完成了这本GO语言开发实战类书籍。本书是我见过的为数不多的剖析区块链技术面面俱到，并兼具深度的专业著作，从公链的整体架构开始，到接口，再到内核，从外

及里一步步揭示公链的技术原理。在揭示这些技术原理的过程中，作者不满足于浅尝辄止，而是深入到参数解析，使本书除了供知识学习之外，更成为一本实操的参考书籍。最为难能可贵的是，在解析区块链技术的同时，本书对公链构建过程中使用的一些其他技术也有涉及，将区块链的来龙去脉都说得非常清楚，而不是仅仅关注区块链本身。本书虽然以比原链为蓝本，但实际上所用技术在区块链中也都大致通用。

希望本书能帮助更多对区块链感兴趣的读者、开发者、技术人员进入这片新天地。

段新星，比原链创始人
2019年元旦

推荐序二

2017年，我带领团队创办北京邮电大学平行汇区块链实验室。两年来，实验室一直致力区块链技术研究，积极参与行业内交流与技术分享，与政府、企业齐心合力推进区块链技术更好更快落地。目前，区块链的发展也更加理性，已广泛用于产品溯源、版权验证、数据共享、IOT控制等众多领域。借助区块链技术去中心化、不可篡改的特点，未来大数据流通和广泛数字应用场景的安全性、可靠性、便利性都将得到显著改善。

区块链作为一种新兴的技术，目前仍有很多可供人们施展才能的方向。其中的公链技术，是去中心化思想的集大成者。且不谈比特币（一种虚拟货币）的发展是以公链为载体，目前所熟知的以太坊平台等也是基于公链技术。公有链技术重在强调如何通过严谨的密码学算法，为无先验的节点提供可信的服务，恰似聚散沙为磐石。虽然公链技术始终是区块链技术爱好者的一片热土，但是对新入门的开发人员而言，并无体系成熟、翔实可靠的开发指导，初学者往往淹没在各种问答式的经验帖中，无助于形成完备的技术体系。

本书是市面上不多见的体系完备之作，在兼顾将公链核心技术讲通透的同时，不囿于细节，致力于呈现给读者区块链的全局脉络。

北京邮电大学副教授，硕士生导师
陈萍
2019年春节

前言

2008年由中本聪第一次提出了区块链的概念，在随后的几年中，区块链成为了电子货币比特币的核心组成部分：作为所有交易的公共账簿。2017年笔者的很多朋友已经在关注区块链技术领域，笔者也在各种技术峰会上分享过多次区块链技术实现细节，在线上也组织了几个区块链技术群。笔者发现有相当多的朋友询问如何深入学习区块链实现技术，但目前市面上很多的资料都仅介绍区块链上的某部分技术，比如加密货币交易、智能合约开发等，并没有完整介绍公链的技术实现。在一次技术峰会演讲后与北京邮电大学区块链实验室的老师交流，受到陈萍老师的鼓励，想到编写一本系统性介绍公链开发的书籍，对学习区块链的初学者会有帮助，于是便开始组织本书的写作。

本书的目标是引导读者全面了解区块链技术实现原理，笔者也一直坚信，了解某一系统最直接的方式就是研读它的源码，所以本书并不是只介绍区块链技术，而是深入分析其背后的实现原理。通过阅读本书，读者可以全面地了解一条公链的技术实现。本书基于比原链的源代码进行分析，比原链是一个开源的有智能合约功能的公共区块链平台，是国内优秀的公链，目前比原链的代码量不多，而且源码结构清晰，特别适合初学者学习。

本书主要内容包括：

第1章介绍公链设计架构，使读者能够宏观地了解区块链技术架构。

第2章介绍比原链相关的交互工具，包括交互工具的操作及代码实现。

第3章介绍比原链的核心进程bytomd，包括启动过程中的初始化等操作。

第4章介绍API Server实现及原理。详解HTTP请求的完整生命周期，并介绍区块链相关的API接口设计。

第5章和第6章详细介绍区块链核心部分，包括区块、区块链、交易的核心数据结构，以及UTXO模型、隔离见证、交易脚本、验证等概念的实现。

第7章和第8章详细讲解比原链智能合约以及智能合约在BVM虚拟机上运行的过程。

第9章介绍区块链钱包的基本概念，包括密钥、账户、资产管理、交易管理等，以及钱包的备份和恢复方式。

第10章详解区块链P2P分布式网络实现原理，以及Kademlia结构化网络算法的实现。

第11章介绍数据持久化存储，以及区块与交易的缓存和存储过程。

第12章和第13章详解PoW与PoS共识机制以及挖矿相关的概念和流程。

第14章介绍区块链技术未来的发展趋势，我们相信区块链能够为人类做出重大贡献。

本书适合区块链开发者、Go语言开发者阅读。由于时间与水平比较有限，我们在编写本书时也难免会出现一些纰漏和错误。读者可以随时通过邮箱weilandeshanhuhai@126.com与我们联系，希望和大家一起学习与讨论区块链技术。

本书在写作过程中得到很多人的帮助，特别是郜策宇、陆志亚、王庆华、朱益祺、阳胜、林浩宇，在此深表感谢。尤其感谢比原链技术团队设计了这样一个优秀的公链，给区块链社区做出了贡献。

郑东旭

2019年3月14日

第1章

公链设计架构

1.1 概述

区块链技术起源于2008年中本聪的论文《比特币：一种点对点电子现金系统》，区块链诞生自中本聪的比特币。

区块链是一个分布式账本，一种通过去中心化、去信任的方式集体维护一个可靠数据库分布式账本是一种在网络成员之间共享、复制和同步的数据库，记录网络参与者之间的交易，比如资产或数据的交换。

区块链分类如下。

· 公链：无官方组织及管理机构，无中心服务器。参与的节点按照系统规则自由地接入网络，节点间基于共识机制开展工作。

· 私链：建立在某个企业内部，系统运作规则根据企业要求进行设定，读写权限仅限于少数节点，但仍保留着区块链的真实性和部分去中心化特性。

· 联盟链：若干个机构联合发起，介于公链和私链之间，兼容部分去中心化的特性。

本书基于国内优秀项目比原链（Bytom），为读者展开公链技术的完整实现。如果说比特币代表区块链1.0时代，以太坊拥有图灵完备性代表的是区块链2.0时代的话，比原链则基于UTXO模型支持了更丰富的

功能（图灵完备的智能合约、多资产管理、Tensority新型的PoW共识算法等），其代表的是区块链2.5时代。比原链是一个开源项目，整个项目基于GO语言开发，代码托管于GitHub上（https://github.com/Bytom/bytom）。

本书基于比原链的1.0.5版本源码进行分析。读者不用纠结本书为何不使用比特币或以太坊作为示例，所谓“有道无术，术尚可求也，有术无道，止于术”，作者认为大部分区块链技术实现都是相似的。目前主要在共识算法（PoW、PoS）和模型（UTXO或Account模型）方面有所不同。比原链作为国内优秀的公链，代码量并不多，而且清晰的源码结构使得程序员和链圈爱好者的学习成本也不高。我们从中可以学到很多东西，如GO语言程序设计及应用、公链设计架构、公链运行原理等。

本章主要内容包括：

- 比原链的总体架构。
- 比原链架构内部各模块功能。
- 比原链编译部署及应用。

1.2 公链总体架构

比原链（Bytom Blockchain或者Bytom）是一个开源的有智能合约功能的公共区块链平台。比原链公链设计架构如图1-1所示。

1.3 比原链各模块功能

我们将从图1-1所示的比原链总架构图中抽离出各个模块，逐一分析及阐述。

1.3.1 用户交互层

比原链的用户交互层如图1-2所示。

1.bytomcli客户端

bytomcli是用户与bytomd进程在命令行下建立通信的RPC客户端。在已经部署比原链的机器上，用户能够使用bytomcli可执行文件发起对比原链的多个管理请求。

bytomcli发送相应的请求，请求由bytomd进程接收并处理。bytomcli的一次完整生命周期结束。

2.bytom-dashboard

bytom-dashboard与bytomcli功能类似，都是发送请求与bytomd进程建立通信。用户可通过Web页面与bytomd进程进行更为友好的交互通信。

在已经部署比原链机器上，会默认开启bytom-dashboard功能，无须再手动部署bytom-dashboard。实际上通过传入的参数用户可以决定是否开启或关闭bytom-dashboard功能。如传入--web.closed，则可以关闭该功能。项目源码地址：https://github.com/Bytom/bytom-dashboard。

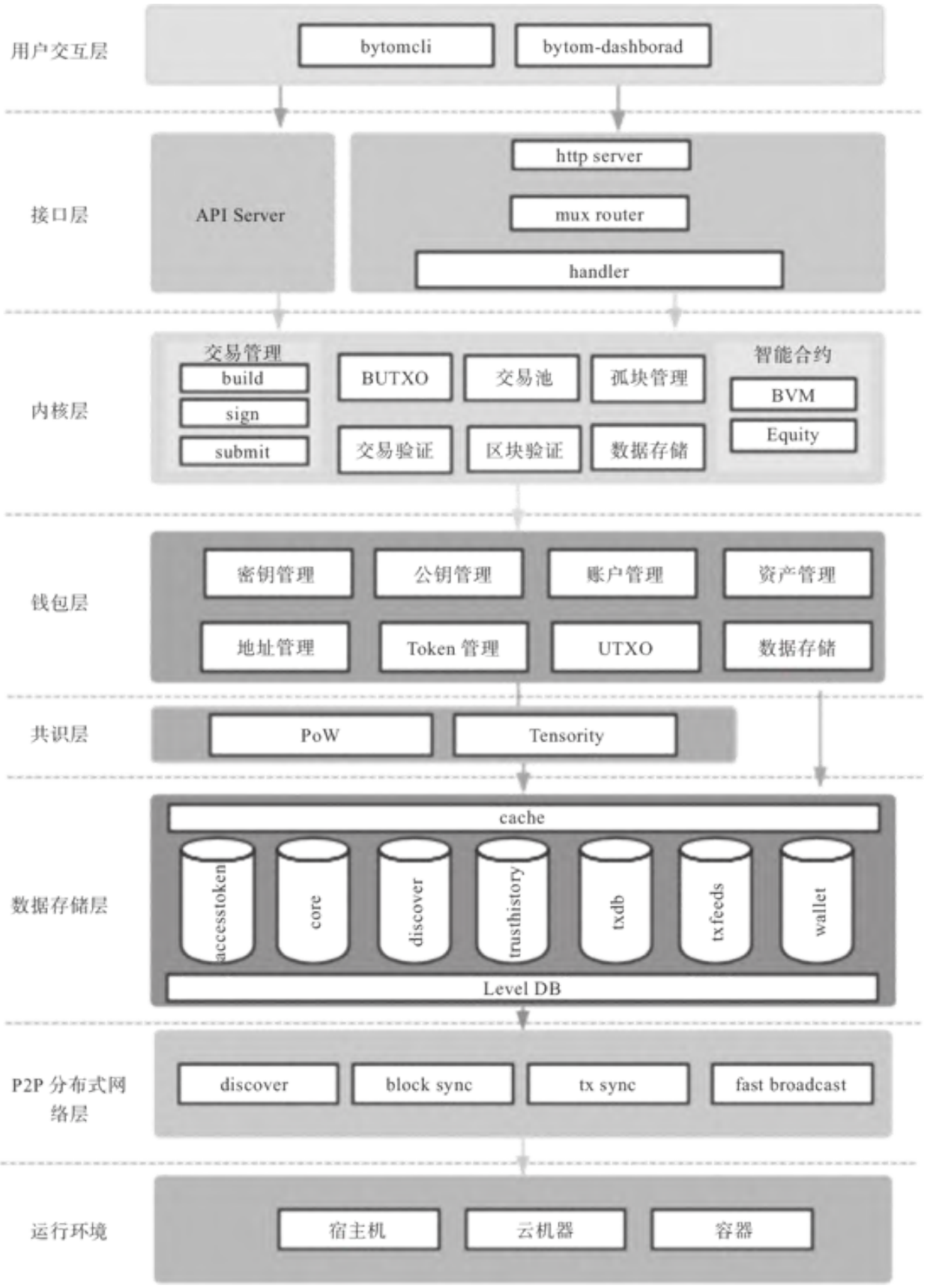


图1-1　比原链架构

图1-2　比原链架构之用户交互层

1.3.2 接口层

比原链接口层架构如图1-3所示。

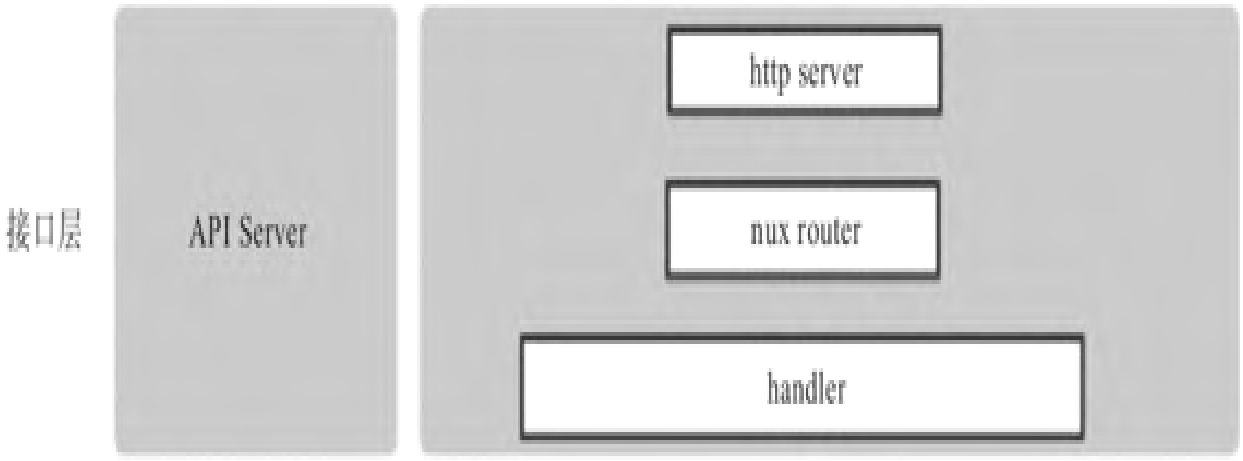


图1-3 比原链架构之接口层

API Server是比原链中非常重要的一个功能，在比原链的架构中专门服务于bytomcli和dashboard，它的功能是接收、处理并响应与用户和矿池相关的请求。默认启动9888端口。API Server的总之主要功能如下：

- 接收并处理用户或矿池发送的请求。
- 管理交易，包括打包、签名、提交等接口操作。
- 管理本地钱包接口。
- 管理本地P2P节点信息接口。
- 管理本地矿工挖矿操作接口等。

API Server服务过程中，在监听地址listener上接收bytomcli或bytom-dashboard的请求访问，对每一个请求，API Server均会创建一个新的goroutine来处理请求。首先，API Server读取请求内容，解析请求；其次，匹配相应的路由项；再次，调用路由项的Handler回调函数来处理；最后，Handler处理完请求之后给bytomcli响应该请求。

1.3.3 内核层

1. 区块和交易管理

内核层包括区块和交易管理、智能合约、虚拟机。

比原链内核层架构如图1-4所示。

内核层是比原链中最重要的功能，代码量大约占总量的54%。

区块链的基本结构是一个链表。链表由一个个区块串联组成。一个区块链中包含成千上万个区块，而一个区块内又包含一个或多个交易。在比原链内核层有一个重要的功能是对区块和交易进行管理。

图1-4 比原链架构之内核层

当网络中的某个节点接收到一个新的有效区块时，节点会验证新区块。当新的区块并未在现有的主链中找到它的父区块，这个新区块会进入孤块管理中等待父区块。如果从现有的主链中找到了父区块，则将其加入到主链。

当网络中的某个节点接收到一笔交易时，节点会验证交易的合法性。验证成功后，该笔交易放入交易池等待矿工打包。一笔交易从发送到完成的整个生命周期需要经过如下过程：

1）A通过钱包向B发出一笔交易，交易金额为100比原币（BTM）。

2）该笔交易被广播到P2P网络中。

3）矿工收到交易信息，验证交易合法性。

4）打包交易，将多个交易组成一个新区块。

5）新区块加入到一个已经存在的区块链中。

6）交易完成，成为区块链的一部分。

2. 智能合约

从传统意义上来说，合约就是现实生活中的合同。区块链中的智能合约是一种旨在以数字化的方式让验证合约谈判或履行合约规则更加便捷的计算机协议。智能合约本质上是一段运行在虚拟机上的“程序代码”，可以在没有第三方信任机构的情况下执行可信交易。

智能合约具有两个特性：可追踪性和不可逆转性。

智能合约是比原链中最核心、也是最重要的部分。在后面章节中，我们会详细介绍智能合约模型（主流模型：UTXO模型、账户模型）、运行原理，以及BVM虚拟机工作机制。我们还将深入代码，了解区块链上智能合约如何在没有第三方信任机构的情况下进行可信交易。

3. 虚拟机

比原链虚拟机（Bytom Virtual Machine，BVM）是建立在区块链上的代码运行环境，其主要作用是处理比原链系统内的智能合约。BVM是比原链中非常重要的部分，在智能合约存储、执行和验证过程中担当着重要角色。

BVM用Equity语言来编写智能合约。比原链是一个点对点的网络，每一个节点都运行着BVM，并执行相同的指令。BVM是在沙盒中运行，和区块链主链完全分开。

1.3.4 钱包层

比原链钱包层架构如图1-5所示。

图1-5 比原链架构之钱包层

钱包可以类比于我们日常生活中的保险箱，我们关心保险箱的开门方式（密钥）和其中保存的财产（UTXO）。比原链钱包层主要负责保存密钥、管理地址、维护UTXO信息，并处理交易的生成、签名，对外提供钱包、交易相关的接口。

比原链的交易发送分为三步：

1）Build：根据交易的输入和输出，构造交易数据。

2）Sign：使用私钥对每个交易输入进行签名。

3）Submit：将交易提交到网络进行广播，等待打包。

1.3.5 共识层

比原链共识层架构如图1-6所示。

图1-6 比原链架构之共识层

共识层用于实现全网数据的一致性，区块链是去中心化账本，需要全网对账本达成共识。共识层通过验证区块和交易，保证新的区块在所有节点上以相同的方式产生。简单说，共识机制就是通过某种方式竞争“记账权”，得到记账权的节点可以将自己生成的区块追加到现有区块链的尾部，其他节点可以根据相同的规则，验证并接受这些区块，丢弃那些无法通过验证的区块。

常见的共识机制有工作量证明（Proof-of-Work，PoW）、股权证明（Proof-of-Stake，PoS）等。

PoW共识机制利用复杂的数学难题作为共识机制，目前一般使用“hash函数的计算结果小于特定的值”。由于hash函数的特性，不可能通过函数值来反向计算自变量，所以必须用枚举的方式进行计算，直到找出符合要求的hash值。这一过程需要进行大量运算。PoW的复杂性保证了任何人都需要付出大量的运算来产生新的块，如果要篡改已有的区块，则需要付出的算力比网络上其他节点的总和都大。PoW优缺点对比如表1-1所示。

表1-1 PoW优缺点对比

PoW 的优点	PoW 的缺点
1. 算法简单，容易实现 2. 破坏共识需要付出极大的成本	1. 消耗大量资源，造成资源浪费 2. 运算过程复杂，导致区块间隔较大 3. 随着 ASIC 的发展，算力集中于少数用户

PoS是另一种共识机制，这种方式要求节点将一部分加密货币锁定，并根据数量和锁定的时长等因素来分配记账权。PoS一般不需要大量计算，所以比PoW更加迅速和高效。PoS优缺点对比如表1-2所示。

表1-2　PoS优缺点对比

PoS 的优点	PoS 的缺点
1. 节能，不需要大量计算 2. 去中心化，所有持币人不需要投入硬件成本，都可以参与 PoS 共识	1. 造成数字货币聚集，导致“贫富不均” 2. 数字货币来于 ICO，早期用户容易囤积

目前还有少量加密数字货币采用其他共识机制，但PoW和PoS是共识机制的主流。由于比原链的特性，结合比原链崇尚“计算即权力”（一种已确定的利益分配的方式。只要计算力高或拥有更多的算力，那就拥有了某些控制权）的主张，需要在多节点上达成较强的共识，对全局一致性、去中心化要求较高，需要在一定程度上牺牲效率，所以比原链选择了PoW作为公链的共识机制。

1.3.6 数据存储层

比原链数据存储层架构如图1-7所示。

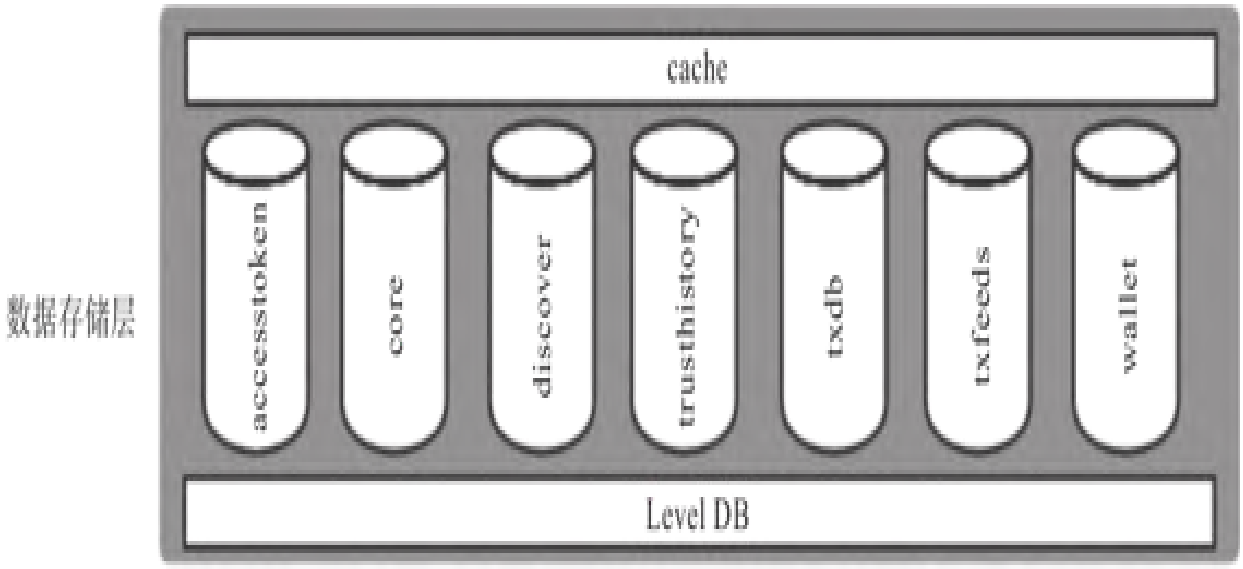


图1-7 比原链架构之数据存储层

比原链在数据存储层上存储所有链上地址、资产交易等信息。数据存储层分为两部分；第一部分为缓存，大部分查询首先从缓存中进行，以减少对磁盘的IO压力；第二部分为持久化存储，当缓存中查询不到数据时，转而从持久化存储中读取，并添加一份到缓存中。

比原链默认使用LevelDB数据库作为持久化存储。LevelDB是Google开发的非常高效的链值数据库。LevelDB是单进程服务，不能同时有多个进程对一个数据库进行读写，同一时间只能有一个进程或一个进程以多并发的方式进行读写。

默认情况下，数据存储在--home参数下的data目录。以Darwin（即MacOS）平台为例，默认数据存储在$HOME/Library/Bytom/data。

数据库包括：

· accesstoken.db：存储token信息（钱包访问控制权限）。

· core.db：存储核心数据库。存储主链相关数据，包括块信息、交易信息、资产信息等。

· discover.db：存储分布式网络中端到端的节点信息。

· trusthistory.db：存储分布式网络中端到端的恶意节点信息。

· txfeeds.db：存储目前比原链代码版本未使用该功能，暂不介绍。

· wallet.db：存储本地钱包数据库。存储用户、资产、交易、UTXO等信息。

1.3.7 P2P分布式网络

比原链分布式网络层的架构如图1-8所示。

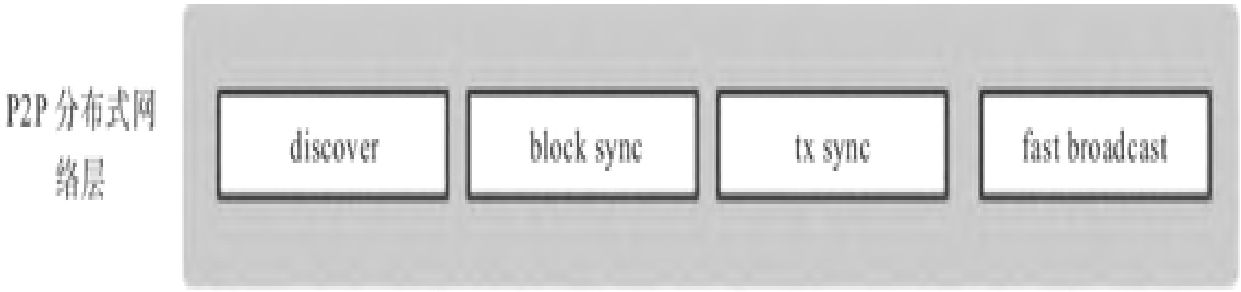


图1-8 比原链架构之P2P分布式网络

比原链作为一个去中心化的分布式系统，其底层个体间的通信机制对整个系统的稳定运行十分重要。个体间的数据同步、状态更新都依赖于整个网络中每个个体之间的通信机制。比原链的网络通信基于P2P通信协议，又根据自身业务的特殊性做了特别的设计。比原链的P2P分布式网络，主要分为四大部分：节点发现、区块同步、交易同步和快速广播。

1. 节点发现

P2P节点发现主要解决新加入网络的节点如何连接到区块链网络中。新加入的节点能够快速地被网络中其他的节点感知；同时节点自己也能够获得其他节点的信息，与其通信交换数据。比原链中的节点发现使用的是Kademlia算法实现的节点发现机制。Kademlia算法实现是一个结构化的P2P覆盖网络，每个节点都有一个全网唯一的标识，称为Node ID。Node ID被分散地存储在各个节点上。Kademlia将发现的节点存储到k桶（k-bucket）中，每个节点只连接距离自己最近的n个节点。以此形成的网络拓扑结构如下图1-9所示。

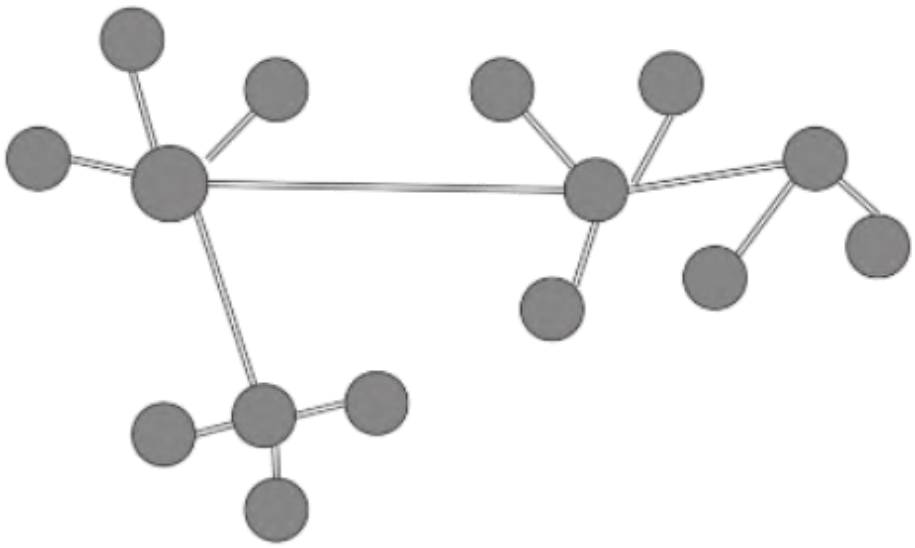


图1-9　P2P网络拓扑架构

2. 区块同步

区块链是一种去中心化的分布式记账系统，全节点需要保存完整的区块信息。当一个节点加入到网络之后，需要做的第一件事情就是同步区块，构建完整的区块链。新加入的节点需要与网络中其他节点同步，从高度为1的区块开始到当前全网的最高高度，这样才能构建成一条完整的区块链。

节点启用快速功能时，同步节点每次最多能够同步1～128个区块（当前高度到下一个checkpoint高度的区块）信息。除了快速同步算法之外，还有普通同步算法。节点主要使用普通同步算法同步网络中较新的区块信息。普通同步算法主要为了保证节点中的区块信息不落后，能够及时同步新挖掘的区块。

普通同步和快速同步的区别是快速同步使用get-headers批量获取区块，使用checkpoint的验证来避免PoW工作量验证，从而能极大提高速度；普通同步一次只能获取一个区块。

快速同步和普通同步可以解决不同场景的需求。快速同步使节点能够快速地重建区块链信息，快速地加入到网络中，这是通过新块广播实现的。在不满足快速同步条件时，则用普通同步，每次同步时请求一个区块。

3. 交易同步

比原链网络为了保证交易的安全，每一笔交易达成后，节点需要将交易信息同步到网络中其他的节点，这个过程称作交易同步。交易同步验证的目的是保证安全性，当交易同步到网络中的其他的节点时，节点会验证交易是否合法，只有当交易合法，节点才会将交易写入自己的交易池中。另一方面，同步交易可以将交易同步到更多的节点，交易可以更快地被打包到区块中。

4. 快速广播

为了保证交易信息能够及时被确认和提交到主链上，比原链提供了快速广播。对于接收到的新区块和新交易信息，网络中的节点会快速广播到当前节点已知的其他节点。

1.4　编译部署及应用

比原链的安装方式有多种。本书从源码分析的角度带领读者了解架构，所以使用源码编译的方式来介绍安装过程。

1. 源码编译部署

1）下载源码：

```
$ git clone https://github.com/Bytom/bytom.git
$GOPATH/src/github.com/bytom
```

2）切换至1.0.5版本：

```
$ cd $GOPATH/src/github.com/bytom
$ git fetch origin v1.0.5
$ git checkout v1.0.5
```

3）编译源码：

```
$ make bytomd
$ make bytomcli
```

4）初始化：

```
$ cd ./cmd/bytomd
$ ./bytomd init --chain_id mainnet
```

目前比原链支持三种网络，使用chain_id进行区分，如下所示：

- mainnet：主网。
- testnet：也称wisdom，测试网。
- solonet：单机模式。

5）启动bytomd进程：

```
$ ./bytomd node

$ ps -ef|grep bytomd
  501 52318   449   0  2:00PM ttys000    0:00.85 ./bytomd node

$ ./bytomcli net-info
{
  "current_block": 36714,
  "highest_block": 36714,
  "listening": true,
  "mining": false,
  "network_id": "wisdom",
  "peer_count": 10,
  "syncing": false,
  "version": "1.0.5+2bc2396a"
}
```

当我们执行ps-ef命令看到bytomd进程时，说明进程已经处于运行状态。使用bytomcli获取节点状态信息，可以看到我们已经成功地运行了bytomd进程。

bytomd进程第一次启动后，默认不会开启挖矿功能。此时会从P2P网络种子节点中获取与之相邻的peer节点，建立握手连接并同步区块。我们将在第10章深入分析P2P网络底层工作原理。

2. 源码目录结构

比原链的源码目录如下所示：

```
$ tree -L 1
.
├── accesstoken         Token管理
├── account             账户管理
├── api                 API Server接口管理
├── asset               资产管理
├── blockchain          交易打包、签名、查询等
├── cmd                 main入口文件
├── common              公共库
├── config               节点配置文件
├── consensus           共识相关模块
├── crypto              加密库
├── dashboard           dashboard页面管理
├── database            数据库管理
├── docs                文档
├── encoding            协议相关的编解码库
├── env                 环境变量管理
├── equity              智能合约语言编译器
├── errors              错误及异常管理
├── math                数学计算相关库
├── metrics             metrics指标库，用于采集API Server请求相关指标
├── mining              挖矿模块
├── net                 API Server使用的HTTP基础库
├── netsync             网络同步管理
├── node                当前节点管理模块，环境初始化等
├── p2p                 分布式网络管理模块
├── protocol            核心数据结构，包含块、交易、bvm虚拟机等
├── test                单元测试
├── testutil            单元测试工具包
├── util                工具包
├── vendor              第三方库
├── version             版本
└── wallet              钱包管理
```

3. 开启挖矿模式

开启挖矿模式的命令如下：

```
$ ./bytomcli set-mining true
```

在默认情况下比原链的挖矿模式是关闭状态。开启挖矿模式有两种方式，第一种方式，使用bytomcli命令行交互，将mining参数设置为true，此时bytomcli会通过RPC协议与bytomd进程交互并启用挖矿模式。关闭挖矿模式则指定set-mining参数为false。第二种方式，使用dashboard页面启用挖矿参数，在这里请读者自行学习dashboard。

4.其他语言SDK简介

比原链技术社区提供了不同语言的SDK，如下所示：

PHP SDK：https://github.com/lxlxw/bytom-php-sdk

Java SDK：https://github.com/chainworld/java-bytom

Java SDK：https://github.com/successli/Bytom-Java-SDK

Python SDK：https://github.com/Bytom-Community/python-bytom

Node SDK：https://github.com/Bytom/node-sdk

1.5 本章小结

本章对比原链的总体架构分层进行了分析，通过比原链的总架构图可以看出一条完整公链的技术架构。然后通过安装部署，介绍了比原链的基础知识。

通过对公链架构的学习，可以从顶层宏观角度对公链设计、功能与价值等诸多方面进行全面了解。

第2章

交互工具

2.1 概述

bytomcli和dashboard是比原链提供的与bytomd进程交互的工具（基于RPC协议）。bytomcli是命令行客户端，dashboard是Web图形界面。dashboard相比bytomcli使用体验更友好，用户可随意选择。

通过交互工具可以完成与token、账户、交易、钱包、挖矿等相关的管理操作。

本章中，我们对bytomcli与bytomd进程的交互过程原理和代码进行分析，并对dashboard做一些简介，主要内容包括：

- bytomcli交互工具介绍。
- dashboard使用方法。

2.2 bytomcli交互工具

2.2.1 bytomcli命令flag参数

bytomcli的使用方式分为两种：一种是后面接flags参数，bytomcli[flags]，bytomcli使用-h参数查看命令的帮助文档；另一种是后面接子命令，bytomcli[command]。所有的命令选项都可以通过执行-h或--help获得指定命令的帮助信息。bytomcli的帮助信息、示例相当详细，简单易懂。建议大家使用帮助信息。

```
$ ./bytomcli -h
Bytomcli is a commond line client for bytom core (a.k.a.
bytomd)

Usage:
  bytomcli [flags]
  bytomcli [command]

Available Commands:
  build-transaction            创建一笔交易
  check-access-token           校验access token 是否合法
  create-access-token          创建 accesss token
  create-account               创建账户
  create-account-receiver      创建账户地址
  create-asset                 创建资产
  create-key                   创建key
  create-transaction-feed      创建交易feed过滤器
  decode-program               将程序解码为指令和数据
  decode-raw-transaction       解码原始交易
  delete-access-token          删除access token
  delete-account               删除账户
  delete-key                   删除key
  delete-transaction-feed      删除交易feed 过滤器
```

estimate-transaction-gas	估算交易费
gas-rate	显示当前交易费比例
get-asset	通过资产id获得资产详情
get-block	根据区块哈希值或高度获取完成区块信息
get-block-count	获取最新区块的数量
get-block-hash	获取最新区块的哈希值
get-block-header	根据区块哈希值或高度获取完成区块的头部信息
get-difficulty	获取最新区块的难度值
get-hash-rate	获取最新区块的nonce值
get-transaction	根据交易hash值获取交易信息
get-transaction-feed	通过别名获取交易feed
get-unconfirmed-transaction	根据交易哈希值获取未确认的交易信息
help	显示帮助信息
is-mining	判断客户端是否开启挖矿功能
list-access-tokens	显示access token信息
list-accounts	显示账户信息
list-addresses	显示账户地址信息
list-assets	显示资产信息
list-balances	显示账户余额
list-keys	显示存在的key信息
list-pubkeys	显示账户公钥
list-transaction-feeds	显示所有的交易feed
list-transactions	显示交易信息
list-unconfirmed-transactions	显示未确认交易的哈希值
list-unspent-outputs	显示账户未使用的输出
net-info	显示当前网络信息
rescan-wallet	重新扫描块信息到钱包
reset-key-password	重置key 密钥
set-mining	开始/停止挖矿
sign-message	对消息进行签名
sign-transaction	使用账户密钥对交易进行签名
submit-transaction	提交签名的交易
update-asset-alias	更新资产别名
update-transaction-feed	更新交易feed
validate-address	校验交易地址
verify-message	校验消息签名
version	显示客户端版本信息
wallet-info	显示钱包信息

以上介绍了bytomcli命令行工具的详细参数。下面，将以一个实际的用例介绍如何使用bytomcli命令行工具。

2.2.2 使用bytomcli查看节点状态信息

使用net-info参数可以查看bytomd节点运行的网络状态。首先，使用bytomcli net-info-h查看net-info的帮助信息。命令执行如下：

```
$ ./bytomcli net-info -h
Print the summary of network
Usage:
  bytomcli net-info [flags]
Flags:
  -h, --help   help for net-info
```

由输出的帮助信息可知，net-info命令仅支持一个可选的flags参数，flags参数列表又仅包含-h参数，可用来查看net-info命令的帮助信息。所以net-info命令的用法是十分简单的，使用bytomcli net-info即可查看节点的网络状态：

```
$ ./bytomcli net-info
{
  "current_block": 36714,
  "highest_block": 36714,
  "listening": true,
  "mining": false,
  "network_id": "mainnet",
  "peer_count": 10,
  "syncing": false,
  "version": "1.0.5+2bc2396a"
}
```

net-info参数返回数据如下所示。

· current_block：当前节点的当前区块高度为36714。

· highest_block：网络中最高区块高度为36714，表明节点已经拥有完整的区块数据。

· listening：当前节点处于监听状态。

· mining：当前节点是否启用挖矿功能。

· network_id：标识节点连接的网络类型（mainnet为正式主网，wisdom为测试网络，solonet为单节点网络）。

· peer_count：节点当前连接的其他节点个数。

· syncing：当前节点是否正在同步区块数据。

· version：节点的版本号。

2.2.3 bytomcli运行案例

本小节我们以net-info命令为例对bytomcli的源码进行剖析，其他命令参数与net-info参数执行的过程大同小异。

bytomcli相关代码文件结构如下所示：

```
$ tree cmd/bytomcli/
cmd/bytomcli/
├── bytomcli
├── commands              commands目录存放bytomcli所有参数的实现
│   ├── accesstoken.go          token参数相关
│   ├── account.go            账户参数相关
│   ├── asset.go            资产参数相关
│   ├── block.go            块参数相关
│   ├── bytomcli.go           bytomcli flag解析相关
│   ├── key.go              key参数相关
│   ├── mining.go           挖矿参数相关
│   ├── net.go              节点状态参数相关
│   ├── program.go          合约参数相关
│   ├── template.go           交易模板参数相关
│   ├── transaction.go            交易参数相关
│   ├── txfeed.go           目前版本该功能未使用
│   ├── util.go               bytomcli相关输出结构体
│   ├── version.go          节点版本参数相关
│   └── wallet.go           本地钱包参数相关
└── main.go               bytomcli入口函数
```

1.Cobra库介绍

bytomcli命令行工具是基于Cobra实现的。Cobra既可以用来创建强大的现代CLI应用程序库，也可以用来生成应用和程序文件。很多知名的开源软件都使用Cobra实现其CLI部分，例如kubernetes、docker、etcd等。

Cobra基于三个基本的概念——commands、arguments和flags，实现对命令参数的解析和行为控制。commands是应用程序的中心，应用程序支持的每个交互都包含在命令中，命令可以具有子命令并可选地运行子命令。flags是修改命令行为的方法，Cobra支持POSIX标准和GO

的flags包，并可以同时作用于父命令和子命令的参数，也支持只在父命令或者子命令有效的参数。

如何使用Cobra库来构建自己的CLI程序呢？下面我们简单介绍Cobra库的使用。

（1）安装Cobra库

Cobra是非常容易使用的，使用go get来安装最新版本的库。Cobra库相对比较大，安装它可能需要花费一些时间。安装完成后，在GOPATH/bin目录下应该有已经编译好的Cobra程序。

```
$ go get -v github.com/spf13/cobra/cobra
```

（2）使用cobra命令生成应用程序

假设现在我们要开发一个基于CLI的命令程序，名字为demo。首先打开cmd，切换到GOPATH的src目录下，执行如下命令：

```
$ cobra init demo
```

在src目录下会生成一个demo的文件夹，如下：

```
$ tree demo/
demo/
├── LICENSE
├── cmd
│   └── root.go
└── main.go
```

如果此时应用程序不需要子命令，那么Cobra生成应用程序的操作就结束了。这里我们先实现一个没有子命令的CLI程序，之后再为程序添加子命令。

接下来继续为demo设计功能。我们打开Cobra自动生成的main.go文件查看，如下：

```
package main

import "demo/cmd"

func main() {
    cmd.Execute()
}
```

main函数执行cmd包的Execute()方法，打开cmd包下的root.go文件查看，发现里面进行了一些初始化操作，并提供了Execute接口。其实，Cobra自动生成的root.go文件中有很多初始化操作是不需要的，其中viper是Cobra集成的配置文件读取的库，这里不需要使用，可以注释掉，如不注释掉则生成的应用程序会大10M左右。

在demo下面新建一个imp包，imp.go内容如下：

```
package imp

import(
    "fmt"
)

func Show(name string, age int) {
    fmt.Printf("My Name is %s, My age is %d\n", name, age)
}
```

在imp.go文件中，Show函数接收两个参数——name和age，使用fmt打印出来。此时，整个demo项目目录结构如下：

```
$ tree demo/
demo/
├── LICENSE
├── cmd
│   └── root.go
├── imp

. .
```

```
│   ├── imp.go
└── main.go
```

Cobra的所有命令都是通过cobra.Command结构体实现的。为了实现demo功能，我们需要修改RootCmd。修改demo/cmd/root.go文件，如下所示：

```
var RootCmd = &cobra.Command{
    Use:   "demo",
    Short: "A test demo",
    Long:  `Demo is a test appcation for print things`,
    // Uncomment the following line if your bare application
    // has an action associated with it:
    Run: func(cmd *cobra.Command, args []string) {
        if len(name) == 0 {
            cmd.Help()
            return
        }
        imp.Show(name, age)
    },
}
```

虽然我们已经定义了command结构，也能通过Execute接口调用RootCmd定义的回调方法。但是若要实现demo的功能还需从命令行解析传入的参数，这部分应该如何实现？这部分可以在cmd包的init方法中实现。init()函数会在每个包完成初始化后自动执行，并且执行优先级比main函数高。init()函数通常被用来对变量进行初始化操作。这里我们使用init()函数在main函数执行之前解析命令行参数。修改demo/cmd/root.go文件，如下所示：

```
var (
    name string
    age  int
)

func init() {
    RootCmd.Flags().StringVarP(&name, "name", "n", "", "person's
name")
    RootCmd.Flags().IntVarP(&age, "age", "a", 0, "person's age")
}
```

至此，demo的功能已经实现了，我们编译运行一下看看实际效果，如下所示：

```
$ go run main.go
Usage:
  demo [flags]

Flags:
  -a, --age int          person's age
      --config string    config file (default is
$HOME/.demo.yaml)
  -h, --help             help for demo
  -n, --name string      person's name
  -t, --toggle           Help message for toggle
```

如果我们想实现一个带有子命令的CLI程序，只需要再执行cobra add为程序新增子命令。下面我们为demo程序新增一个server的子命令，如下所示：

```
$ cd demo
$ cobra add server
```

在demo目录下生成了一个cmd/server.go文件，如下所示：

```
$ tree demo/
demo/
├── LICENSE
├── cmd
│   ├── root.go
│   └── server.go
├── imp
│   └── imp.go
└── main.go
```

接下来配置server子命令，此操作与前文中修改root.go文件类似效果如下：

```
$ go run main.go
Usage:
  demo [flags]
  demo [command]

Available Commands:
  help        Help about any command
  server      A brief description of your command

Flags:
  -a, --age int          person's age
      --config string    config file (default is
$HOME/.demo.yaml)
  -h, --help             help for demo
  -n, --name string      person's name
  -t, --toggle           Help message for toggle

Use "demo [command] --help" for more information about a
command.
```

---

2. bytomcli运行流程与原理

对Cobra库有了了解之后，我们再来学习bytomcli的代码时会感觉比较容易，其运行原理如图2-1所示。同上一节中的demo程序一样，bytomcli在main函数中使用cmd.Execute()来启动应用程序，cmd.Execute()是调用commands包的Execute()方法。在mian.go中引入commands包的时候，给它起了一个别名cmd。

```
cmd/bytomcli/main.go
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    cmd.Execute()
}
```

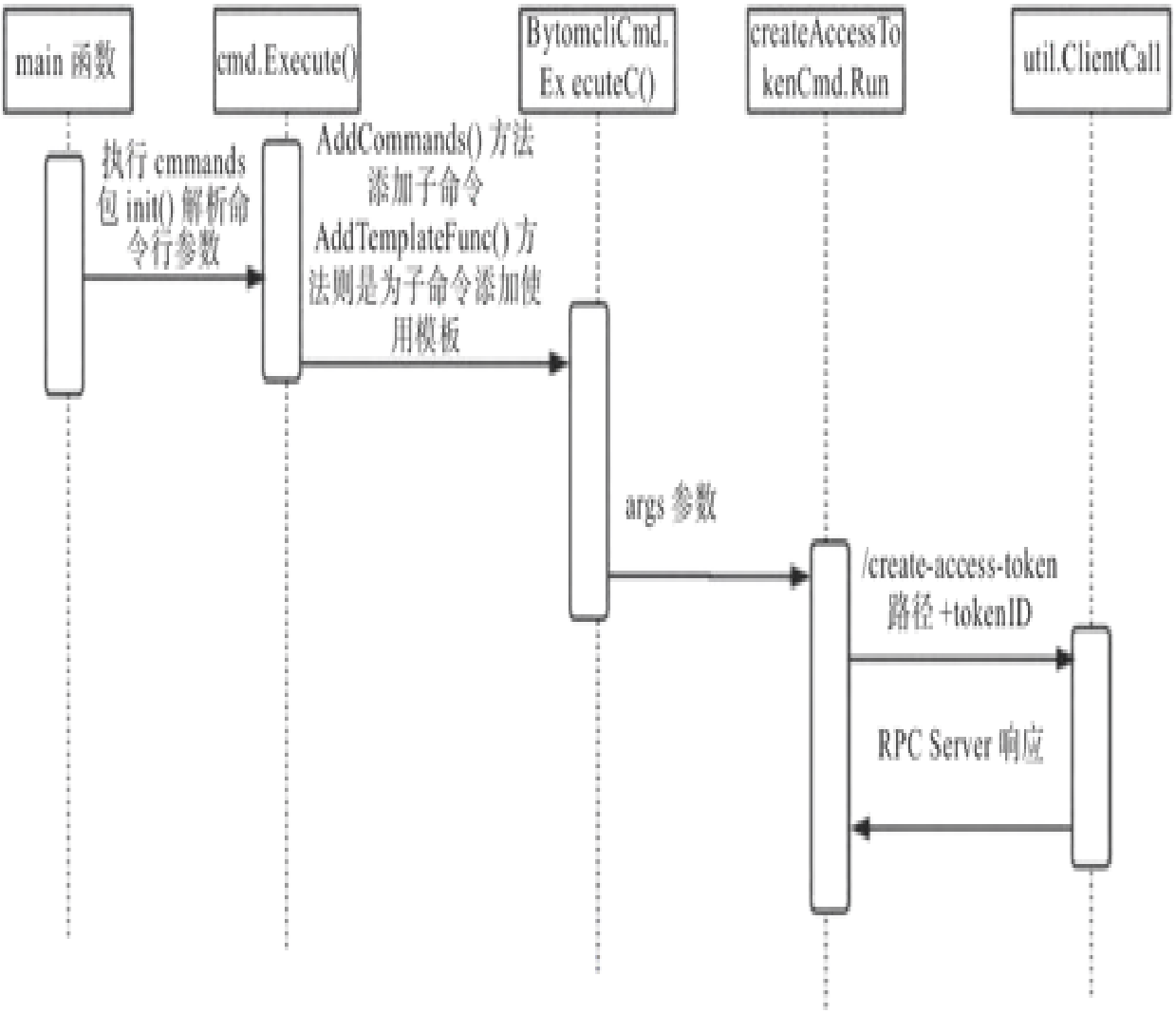


图2-1　bytomcli运行原理

当程序调用commands包的Execute()方法时，会先执行commands包中所有的init()方法，这些init()主要用来解析命令行参数。当执行完commands包中所有的init()方法之后，才会执行Execute()。在Execute()方法中，首先执行AddCommands()方法，该方法主要用来给BytomcliCmd命令添加子命令；而AddTemplateFunc()方法则是为子命令添加使用模板的。在解析flags参数、添加子命令和添加使用模板这些初始化操作完成之后，才真正进入命令的执行过程。在BytomcliCmd.ExecuteC()方法中，Cobra库会根据用户输入的命令跳转到相应的子命令，同时执行命令定义的Run方法。

```
cmd/bytomcli/commands/bytomcli.go
func Execute() {

    AddCommands()
    AddTemplateFunc()

    if _, err := BytomcliCmd.ExecuteC(); err != nil {
        os.Exit(util.ErrLocalExe)
    }
}
```

下面我们以create-access-token命令为例，看看这个命令的Run方法都执行了哪些操作。首先会取args参数的第一位作为tokenID，然后调用util包下的ClientCall方法请求/create-access-token路径，并将tokenID传入，创建token。

```
bytom/cmd/bytomcli/commands/accesstoken.go
var createAccessTokenCmd = &cobra.Command{
    Use:   "create-access-token <tokenID>",
    Short: "Create a new access token",
    Args:  cobra.ExactArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
        var token accessToken
        token.ID = args[0]

        data, exitCode := util.ClientCall("/create-access-
token", &token)
        if exitCode != util.Success {
            os.Exit(exitCode)
        }
        printJSON(data)
    },
}
```

ClientCall方法封装了一个RPC的client，根据用户传入的路径和参数发送请求，并解析RPC server返回的内容。

```
bytom/util/util.go
func ClientCall(path string, req ...interface{}) (interface{},
int) {
```

```
    var response = &api.Response{}
    var request interface{}

    if req != nil {
        request = req[0]
    }

    client := MustRPCClient()
    client.Call(context.Background(), path, request, response)

    switch response.Status {
    case api.FAIL:
        jww.ERROR.Println(response.Msg)
        return nil, ErrRemote
    case "":
        jww.ERROR.Println("Unable to connect to the bytomd")
        return nil, ErrConnect
    }

    return response.Data, Success
}
```

至此，create-access-token命令执行完成，其生命周期终止。

2.3 dashboard交互工具

在比原链中，dashboard是单独一个项目，项目源码地址：https://github.com/Bytom/bytom-dashboard。可能会有读者疑惑，既然dashboard是一个单独的项目，为什么在启动bytomd进程的时候会启用dashboard服务呢？这是因为dashboard编译后被硬编码到bytomd中，源码在dashboard/dashboard.go中。而在API Server下的api/api.go，我们可以看到引用handle信息如下：

```
mux := http.NewServeMux()
mux.Handle("/dashboard/", http.StripPrefix("/dashboard/",
static.Handler{
        Assets:  dashboard.Files,
        Default: "index.html",
}))
```

2.3.1 使用dashboard发送一笔交易

在浏览器中输入http://127.0.0.1:9888/，打开dashboard页面。页面左下方显示当前节点同步区块的进度。点击右上角“新建交易”如图2-2所示。

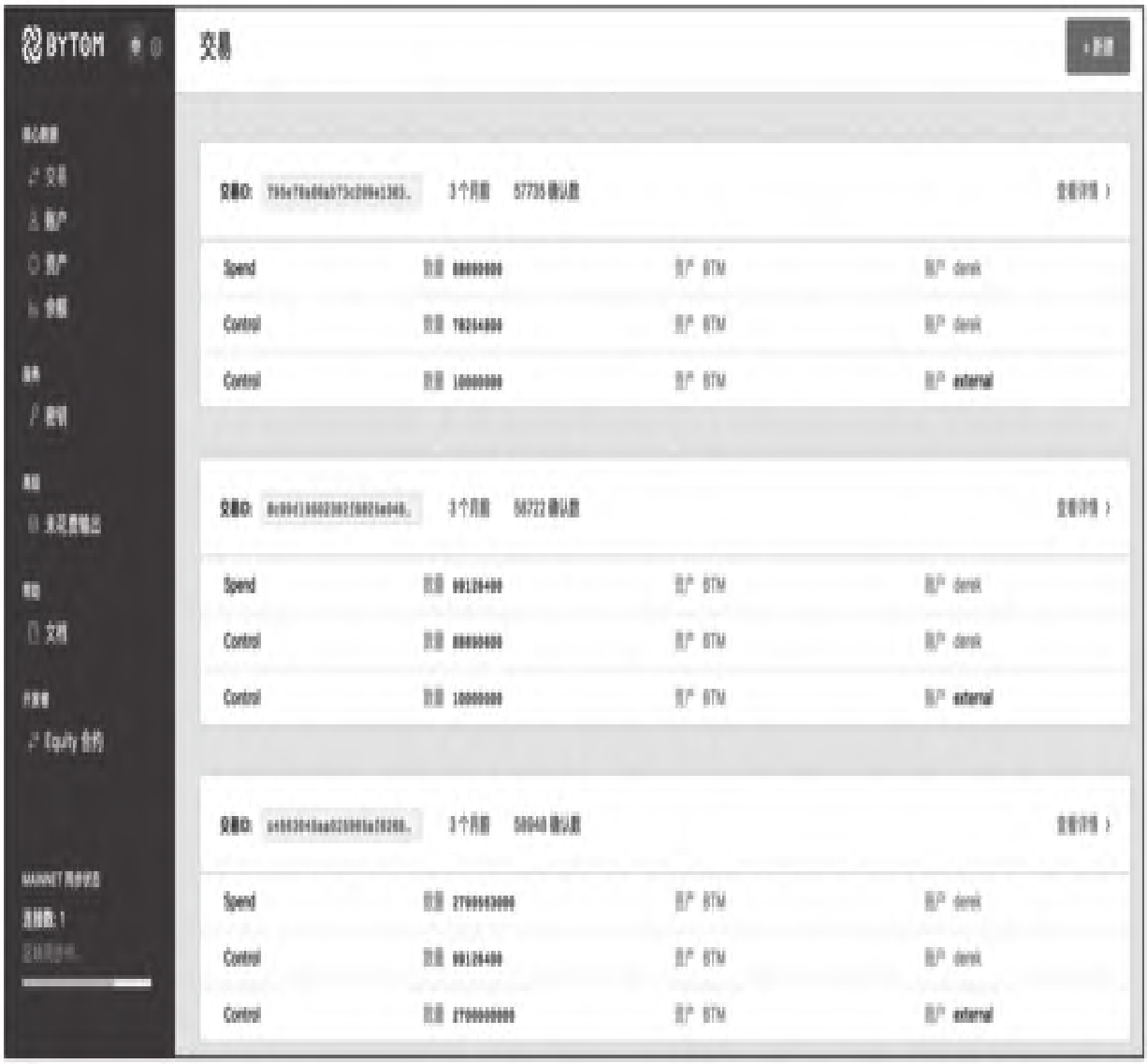


图2-2 dashboard示意图

在比原链中，新建交易分为两种，一种是简单交易，一种是高级交易。本节中使用简单交易发送一笔交易。高级交易会在后面章节进

行详细讲解。

如图2-3所示，我们从本地钱包derek账户下的BTM资产往bm1q5p9d4gelfm4cc3zq3slj7vh2njx23ma2cf866j地址中打入了8 254 800Neu（诺），其中Gas的手续费约为500 000Neu（诺）。然后输入钱包密码并提交交易。得到结果如图2-4所示。

交易ID为3ebe89ddb64f3a7ef1742e...目前正在往主网中广播，待到交易确认则说明这笔交易最终成功。一般认为，一笔交易经过6个区块的验证后此交易无法逆转。

2.3.2 使用dashboard开启挖矿模式

在第1章介绍编译部署及应用时，介绍了如何使用bytomcli命令行交互的方式开启节点的挖矿模式，同样，我们也可以使用dashboard来启用或关停挖矿模式。

如图2-5所示，点击dashboard左上角齿轮按钮，点击核心状态，显示节点的配置信息，如图2-6所示。默认情况下挖矿选项是关闭状态，在这里我们可以启用节点的挖矿模式。

图2-3　dashboard“新建交易”页面

图2-4　dashboard交易结果

图2-5　dashboard示意图

图2-6　dashboard示意图

2.4 本章小结

本章对用户交互层做了详细介绍，对bytomcli命令行和dashboard进行了分析，对Cobra库的使用做了详细的介绍，并对Cobra如何生成CLI下flag的流程进行阐述。本章从源码的角度深入分析了bytomcli与bytomd的RPC交互过程。最后以dashboard发送一笔交易为例，展示了dashboard的使用方法。

第3章

守护进程的初始化与运行

3.1 概述

节点初始化是节点首次使用时，根据用户传入的参数进行设置，并根据参数进行网络、数据库、本地区块链以及P2P分布式网络等模块的初始化，使得节点能够正常运行。节点初始化由bytomd守护进程执行，在初次运行时一次性完成。

本章主要内容：

· bytomd守护进程初始化流程。

· 守护进程初始化的具体实现，包括网络、数据库、本地区块链初始化等。

· 守护进程启动流程和停止流程。

3.2 bytomd守护进程初始化流程及命令参数

守护进程是一种特殊进程，启动后一直在后台运行，只有当触发特定的信号时，才会执行退出操作。比原链的守护进程是bytomd，初始化流程如图3-1所示。

在编写命令行程序时，通常需要对命令参数进行解析。不同语言一般都会提供解析命令行参数的方法或库，以方便程序员使用。在GO语言标准库中提供了flag包，方便进行命令行解析。

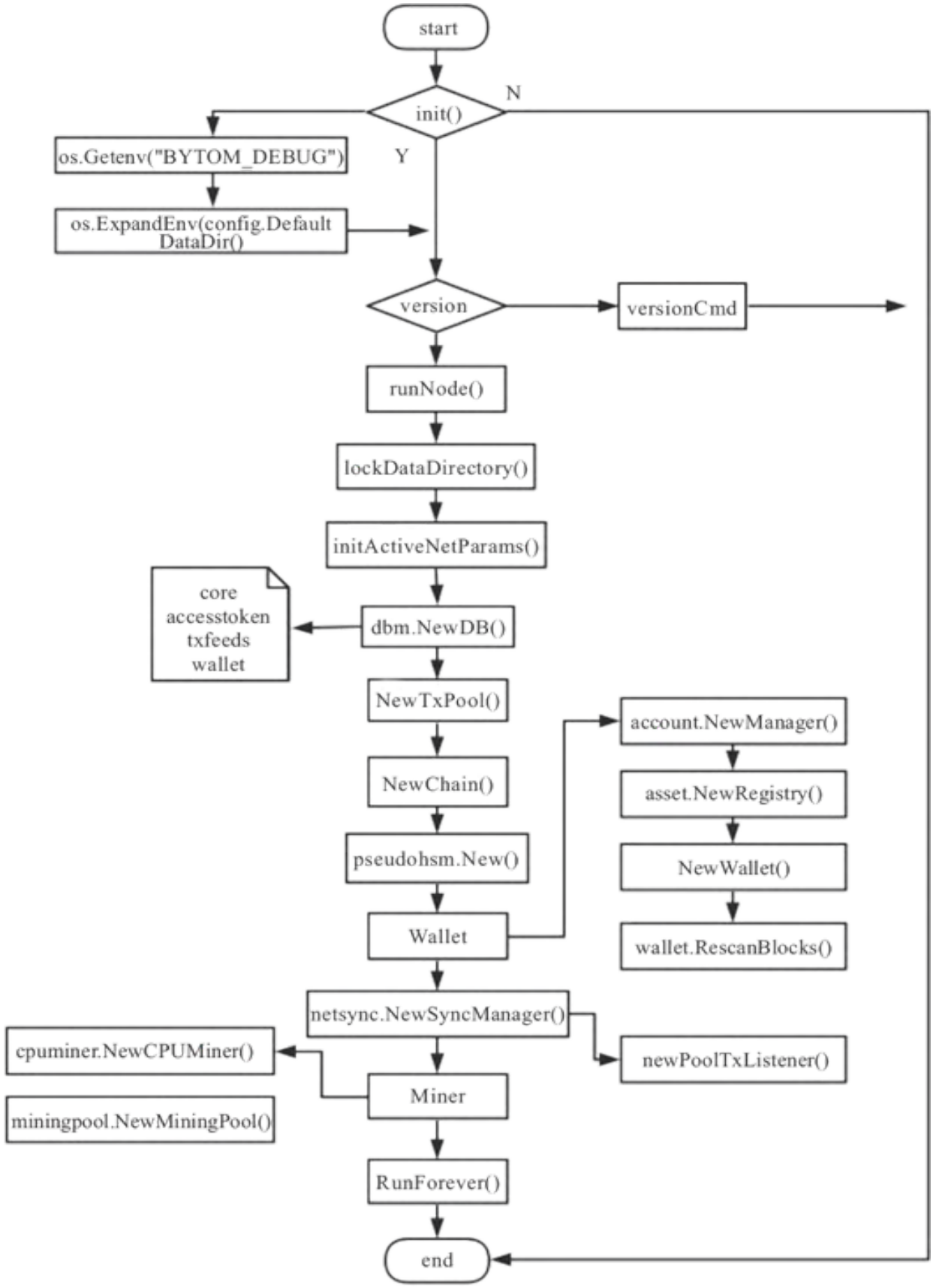


图3-1　bytomd守护进程初始化流程

bytomd进程支持的传参如下：

```
$ ./bytomd -h
Multiple asset management.

Usage:
  bytomd [command]

Available Commands:
  help         显示帮助信息
  init         初始化网络类型
  node         运行bytomd节点
  version      显示版本信息

coral[coralmac] bytomd $ ./bytomd node -h
Run the bytomd

Usage:
  bytomd node [flags]

Flags:
      --auth.disable                关闭API Server Auth验证功能，默
认为true
      --chain_id string             指定网络类型
  -h, --help                        显示帮助信息
      --log_file string              指定日志输出文件
      --log_level string            指定日志输出级别
      --mining                      开启挖矿模式，默认为false
      --p2p.dial_timeout int        设置p2p节点连接超时时间，默认3s
      --p2p.handshake_timeout int   设置p2p节点握手超时时间，默认30s
      --p2p.laddr string            设置p2p节点监听地址，默
认"tcp://0.0.0.0:46656"
      --p2p.max_num_peers int       设置p2p节点最大连接节点，默认50
      --p2p.pex                     设置p2p节点信息交换功能，默认
true
      --p2p.seeds string            设置p2p节点的seeds种子节点
      --p2p.skip_upnp               设置是否使用upnp协议功能。默认
false
      --prof_laddr string           指定pprof地址进入pprof调试模式，
默认不指定则不进入
           pprof调试功能
      --simd.enable                 是否启用simd，用于Tenaority
```

CPU指令优化

--vault_mode	运行vault模式，无网络模式
--wallet.disable	关闭本地钱包功能，默认开启
--wallet.rescan	重载本地钱包信息
--web.closed	自动打开dashboard功能，默认为true

Global Flags:

--home string	指定bytomd运行的home目录，存储配置、keystore、数据的目录
--trace	启用trace功能，出错时显示出错栈信息

3.3 bytomd守护进程的初始化实现

bytomd守护进程的Cobra流程与bytomcli过程非常相似，所以在此略去，后续主要对bytomd守护进程重要内容进行深入分析。在这里我们看一下bytomd预处理过程中使用到的代码文件结构，命令如下：

```
$ tree cmd/bytomd/
cmd/bytomd/
├── bytomd
├── commands
│   ├── init.go          节点网络初始化相关
│   ├── root.go          节点root目录相关
│   ├── run_node.go      节点守护进程相关
│   └── version.go     节点版本参数相关
├── main.go          bytomd入口函数
```

bytomd守护进程启动时，会根据不同的命令行flag参数，初始化不同的模块，最终以守护进程的方式运行。有关bytomd守护进程所有的运行工作都在node.NewNode（config）的具体实现中。下面介绍具体的实现过程。

3.3.1 Node对象

```
node/node.go
type Node struct {
    cmn.BaseService

    // config
    config *cfg.Config

    syncManager *netsync.SyncManager

    wallet       *w.Wallet
    accessTokens *accesstoken.CredentialStore
    api          *api.API
    chain        *protocol.Chain
    txfeed       *txfeed.Tracker
    cpuMiner     *cpuminer.CPUMiner
    miningPool   *miningpool.MiningPool
    miningEnable bool
}
```

Node对象说明如下。

- cmn.BaseService：服务管理。
- config：当前节点的全局配置。
- syncManager：区块和交易同步管理。
- wallet：本地钱包管理。
- accessTokens：token管理，用户访问凭证。
- api：API Server接口服务。
- chain：本地区块链管理对象。
- txfeed：当前版本中该功能未使用。

· cpuMiner：CPU挖矿管理对象。

· miningPool：矿池管理对象。

· miningEnable：是否启用挖矿模式。

node.NewNode（config）整个过程是为了创建Node对象，Node对象是整个bytomd所有模块运行的基础。

cmn.BaseService是tendermint框架的一个服务管理模块，在这里我们可以把Node作为一个服务，对该服务进行OnStart/OnStop/IsRunning等操作管理。tendermint框架可以保证这些操作不会被重复执行多次。

3.3.2 配置初始化

在执行node.NewNode（config）之前，config的默认配置就已经定义好了。在深入node.NewNode（config）分析之前，我们需要先了解默认配置都有哪些。

```
cmd/bytomd/commands/root.go
config = cfg.DefaultConfig()
```

首先，bytomd守护进程声明一个config全局变量，表示整个bytomd守护进程的配置信息。进程启动时config对象被赋予一个默认的配置参数。

```
config/config.go
func DefaultConfig() *Config {
    return &Config{
        BaseConfig: DefaultBaseConfig(),
        P2P:        DefaultP2PConfig(),
        Wallet:     DefaultWalletConfig(),
        Auth:       DefaultRPCAuthConfig(),
        Web:        DefaultWebConfig(),
        Simd:       DefaultSimdConfig(),
    }
}
```

默认的配置参数分有6个，每个针对不同的模块。下面对配置进行说明。我们将默认参数归纳为三块：Base基础配置、P2P网络配置、其他配置。

1.Base基础配置

BaseConfig用于配置bytomd节点所需的基础参数，包括数据目录、日志、监听地址等相关参数。

```
config/config.go
type BaseConfig struct {
```

```
    RootDir string `mapstructure:"home"`        // 存储配置、keystore、数据的root目录
    ChainID string `mapstructure:"chain_id"`    // 网络类型，可选mainnet主网、testnet
      测试网、solonet
    LogLevel string `mapstructure:"log_level"` // 日志输出级别
    PrivateKey string `mapstructure:"private_key"`  // 私钥，用于共识协议的验证
    Moniker string `mapstructure:"moniker"`     // 节点名称，默认情况下为anonymous
    ProfListenAddress string `mapstructure:"prof_laddr"`
    // pprof监听地址，用于查看节点运行时goorutine、栈信息、CPU占用等信息。默认不开启
    FastSync bool `mapstructure:"fast_sync"`    // 块的快速同步机制，默认开启
    Mining bool `mapstructure:"mining"`         // 是否启用节点挖矿，默认不开启
    FilterPeers bool `mapstructure:"filter_peers"` // 目前版本该参数未使用
    TxIndex string `mapstructure:"tx_index"`    // 目前版本该参数未使用
    DBBackend string `mapstructure:"db_backend"` // 比原链数据存储引擎，默认为LevelDB
    DBPath string `mapstructure:"db_dir"` // 数据存储目录，默认为--root参数下的data目录
    KeysPath string `mapstructure:"keys_dir"`  // keystore密钥存储。默认路径为--root
     参数下的keystore目录
    HsmUrl string `mapstructure:"hsm_url"`      // 目前版本该参数未使用
    ApiAddress string `mapstructure:"api_addr"`// API Server本地监听的地址，默认为
 0.0.0.0:9888
    VaultMode bool `mapstructure:"vault_mode"` // vault无网络模式
    Time time.Time                               // 目前版本该参数未使用
    LogFile string `mapstructure:"log_file"`    // 文件日志输出路径
}
```

部分参数从配置文件中获取默认值，比如ApiAddress参数，它的tag是api_addr。我们可以从config/toml.go中获取默认值：

```
config/toml.go
var defaultConfigTmpl = `# This is a TOML config file.
fast_sync = true
db_backend = "leveldb"
api_addr = "0.0.0.0:9888"
`
var mainNetConfigTmpl = `chain_id = "mainnet"
[p2p]
laddr = "tcp://0.0.0.0:46657"
seeds =
"45.79.213.28:46657,198.74.61.131:46657,212.111.41.245:46657,47
.100.214.154:46657,47.100.109.199:46657,47.100.105.165:46657"
`
```

2. P2P网络配置

P2PConfig用于配置bytomd P2P通信协议中使用的参数，包括本机监听端口、通信节点超时、地址簿等相关参数。

```
config/config.go
type P2PConfig struct {
    RootDir          string `mapstructure:"home"`    // 跟
BaseConfig中的RootDir相同
    ListenAddress    string `mapstructure:"laddr"`   // P2P分布式
网络监听的端口，
用于节点之间的相互通信。默认tcp://0.0.0.0:46656
    Seeds            string `mapstructure:"seeds"`   // 种子节点
    SkipUPNP         bool   `mapstructure:"skip_upnp"` // 是否不
使用upnp功能，默认为false
    AddrBook         string `mapstructure:"addr_book_file"`  //
p2p地址簿路径，用于存储已知的peer节点
    AddrBookStrict   bool   `mapstructure:"addr_book_strict"`
// 目前版本该参数未使用
    PexReactor       bool   `mapstructure:"pex"`     // 目前版本该
参数未使用
    MaxNumPeers      int    `mapstructure:"max_num_peers"`// 最
大节点连接数，默认为50
    HandshakeTimeout int    `mapstructure:"handshake_timeout"`
// 节点连接握手超时时间，默认30s
    DialTimeout      int    `mapstructure:"dial_timeout"` // 节
```

```
点连接超时时间，默认3s
}
```

注意，在比特币中，节点会采用DNS的方式来询问种子节点，进而查询到其他节点的IP地址。而在比原链中，种子节点是IP地址，一般会硬编码到代码里。技术细节我们会在后面的第10章详细讲解。

3. 其他配置

WalletConfig用于配置bytomd本地钱包使用的参数，包括是否启用本地钱包和更新等相关参数。

```
config/config.go
type WalletConfig struct {
    Disable bool `mapstructure:"disable"` // 是否启用本地钱包，默认值为false
    Rescan  bool `mapstructure:"rescan"`  // 重新扫描钱包信息
}

type RPCAuthConfig struct {
    Disable bool `mapstructure:"disable"` // 是否启用API Server的验证机制，默认值为false
}

type WebConfig struct {
    Closed bool `mapstructure:"closed"`   // 是否启用bytom-dashboard，默认值为false
}

type SimdConfig struct {
    Enable bool `mapstructure:"enable"`   // 是否启用Tensority CPU指令优化
}
```

在bytomd守护进程声明config=DefaultConfig()之后，init()函数实现了config对象中各属性的赋值。具体实现代码如下：

```
cmd/bytomd/commands/run_node.go
func init() {
```

```
    runNodeCmd.Flags().String("prof_laddr",
config.ProfListenAddress, "Use http to profile bytomd
programs")
    runNodeCmd.Flags().Bool("mining", config.Mining, "Enable
mining")

    runNodeCmd.Flags().Bool("simd.enable", config.Simd.Enable,
"Enable SIMD mechan for tensority")

    runNodeCmd.Flags().Bool("auth.disable",
config.Auth.Disable, "Disable rpc access authenticate")

    runNodeCmd.Flags().Bool("wallet.disable",
config.Wallet.Disable, "Disable wallet")
    runNodeCmd.Flags().Bool("wallet.rescan",
config.Wallet.Rescan, "Rescan wallet")
    runNodeCmd.Flags().Bool("vault_mode", config.VaultMode,
"Run in the offline enviroment")
    runNodeCmd.Flags().Bool("web.closed", config.Web.Closed,
"Lanch web browser or not")
    runNodeCmd.Flags().String("chain_id", config.ChainID,
"Select network type")

    // log level
    runNodeCmd.Flags().String("log_level", config.LogLevel,
"Select log level(debug, info, warn, error or fatal")

    // p2p flags
    runNodeCmd.Flags().String("p2p.laddr",
config.P2P.ListenAddress, "Node listen address. (0.0.0.0:0
means any interface, any port)")
    runNodeCmd.Flags().String("p2p.seeds", config.P2P.Seeds,
"Comma delimited host:port seed nodes")
    runNodeCmd.Flags().Bool("p2p.skip_upnp",
config.P2P.SkipUPNP, "Skip UPNP configuration")
    runNodeCmd.Flags().Bool("p2p.pex", config.P2P.PexReactor,
"Enable Peer-Exchange ")
    runNodeCmd.Flags().Int("p2p.max_num_peers",
config.P2P.MaxNumPeers, "Set max num peers")
    runNodeCmd.Flags().Int("p2p.handshake_timeout",
config.P2P.HandshakeTimeout, "Set handshake timeout")
    runNodeCmd.Flags().Int("p2p.dial_timeout",
config.P2P.DialTimeout, "Set dial timeout")

    // log flags
    runNodeCmd.Flags().String("log_file", config.LogFile, "Log
output file")
```

```
    RootCmd.AddCommand(runNodeCmd)
}
```

在init()函数中定义了很多不同类型的flag参数，并将flag的参数值绑定到config对象上，比如：

```
runNodeCmd.Flags().Bool("mining", config.Mining, "Enable
mining")
```

这条语句的含义为：

- 定义一个Bool类型的flag参数。
- 该flag的名称为mining。
- 该flag的赋值对象为config.Mining。
- 该flag的描述信息为Enable mining。

至此，bytomd守护进程所需要的配置信息初始化完毕，程序运行真正进入初始阶段。下面对此进行深入分析。

3.3.3 创建文件锁

在比原链中，一份数据目录（--root参数指定）只能同时由一个bytomd守护进程读写，因为LevelDB高性能键值数据库是单进程模式，如果多个进程同时读写一份数据，会造成数据不一致的情况。因此，需要使用文件锁可以保证同一时间一个进程读写一份数据目录，代码如下：

```
node/node.go
if err := lockDataDirectory(config); err != nil {
    cmn.Exit("Error: " + err.Error())
}

func lockDataDirectory(config *cfg.Config) error {
    _, _, err := flock.New(filepath.Join(config.RootDir,
"LOCK"))
    if err != nil {
        return errors.New("datadir already used by another
process")
    }
    return nil
}
```

bytomd启动时，lockDataDirectory函数使用flock在RootDir目录下创建一个LOCK文件。如果bytomd进程在一个文件的inode上加了锁，那么再次启动bytomd进程则会对errors.New中的内容报错并退出进程。flock的作用是检测进程是否已经存在。

flock主要有3种操作类型。

· LOCK_SH：共享锁，多个进程使用同一把锁用于读锁。

· LOCK_EX：排他锁，同时只允许一个进程使用，一般用于写锁。

· LOCK_UN：释放锁。

如果深入研究flock包的函数，我们可以看到，这里使用了LOCK_EX锁，即同时只允许一个进程使用，代码示例如下：

```
vendor/github.com/prometheus/prometheus/util/flock/flock_unix.go
func (l *unixLock) set(lock bool) error {
    how := syscall.LOCK_UN
    if lock {
        how = syscall.LOCK_EX
    }
    return syscall.Flock(int(l.f.Fd()), how|syscall.LOCK_NB)
}
```

3.3.4 初始化网络类型

比原链的三种网络模式，分别是mainnet主网、testnet测试网和solonet单机模式。

```
node/node.go
initActiveNetParams(config)

func initActiveNetParams(config *cfg.Config) {
    var exist bool
    consensus.ActiveNetParams, exist =
consensus.NetParams[config.ChainID]
    if !exist {
        cmn.Exit(cmn.Fmt("chain_id[%v] don't exist",
config.ChainID))
    }
}
```

其中initActiveNetParams函数根据用户传入的chain_id，初始化网络类型。consensus.ActiveNetParams对象保存了当前使用的网络模式。在比原链代码中会经常引用consensus.ActiveNetParams对象，用来识别当前节点连接的网络类型。

```
consensus/general.go
var ActiveNetParams = MainNetParams

var NetParams = map[string]Params{
    "mainnet": MainNetParams, // 主网
    "wisdom":  TestNetParams, // 测试网
    "solonet": SoloNetParams, // 单机模式
}

var MainNetParams = Params{
    Name:            "main",
    Bech32HRPSegwit: "bm",
    Checkpoints: []Checkpoint{
        {10000, bc.NewHash([32]byte{0x93, 0xe1, 0xeb, 0x78,
0x21, 0xd2, 0xb4, 0xad, 0x0f, 0x5b, 0x1c, 0xea, 0x82, 0xe8,
0x43, 0xad, 0x8c, 0x09, 0x9a, 0xb6, 0x5d, 0x8f, 0x70, 0xc5,
0x84, 0xca, 0xa2, 0xdd, 0xf1, 0x74, 0x65, 0x2c})},
```

```
        {20000, bc.NewHash([32]byte{0x7d, 0x38, 0x61, 0xf3,
0x2c, 0xc0, 0x03, 0x81, 0xbb, 0xcd, 0x9a, 0x37, 0x6f, 0x10,
0x5d, 0xfe, 0x6f, 0xfe, 0x2d, 0xa5, 0xea, 0x88, 0xa5, 0xe3,
0x42, 0xed, 0xa1, 0x17, 0x9b, 0xa8, 0x0b, 0x7c})},
        {30000, bc.NewHash([32]byte{0x32, 0x36, 0x06, 0xd4,
0x27, 0x2e, 0x35, 0x24, 0x46, 0x26, 0x7b, 0xe0, 0xfa, 0x48,
0x10, 0xa4, 0x3b, 0xb2, 0x40, 0xf1, 0x09, 0x51, 0x5b, 0x22,
0x9f, 0xf3, 0xc3, 0x83, 0x28, 0xaa, 0x4a, 0x00})},
        {40000, bc.NewHash([32]byte{0x7f, 0xe2, 0xde, 0x11,
0x21, 0xf3, 0xa9, 0xa0, 0xee, 0x60, 0x8d, 0x7d, 0x4b, 0xea,
0xcc, 0x33, 0xfe, 0x41, 0x25, 0xdc, 0x2f, 0x26, 0xc2, 0xf2,
0x9c, 0x07, 0x17, 0xf9, 0xe4, 0x4f, 0x9d, 0x46})},
        {50000, bc.NewHash([32]byte{0x5e, 0xfb, 0xdf, 0xf5,
0x35, 0x38, 0xa6, 0x0b, 0x75, 0x32, 0x02, 0x61, 0x83, 0x54,
0x34, 0xff, 0x3e, 0x82, 0x2e, 0xf8, 0x64, 0xae, 0x2d, 0xc7,
0x6c, 0x9d, 0x5e, 0xbd, 0xa3, 0xd4, 0x50, 0xcf})},
        {62000, bc.NewHash([32]byte{0xd7, 0x39, 0x8f, 0x23,
0x57, 0xf9, 0x4c, 0xa0, 0x28, 0xa7, 0x00, 0x2b, 0x53, 0x9e,
0x51, 0x2d, 0x3e, 0xca, 0xc9, 0x22, 0x59, 0xfc, 0xd0, 0x3f,
0x67, 0x1a, 0x0a, 0xb1, 0x02, 0xbf, 0x2b, 0x03})},
    },
}
```

ActiveNetParams默认使用主网。MainNetParams中的参数说明如下。

· Bech32HRPSegwit：隔离见证，是一种协议升级，我们会在后面第5章讲解。

· Checkpoints：检查点，指定一个高度，以及与这个高度相匹配的hash值，用于快速同步时验证区块的正确性。通常在主网升级时，会将历史的块信息硬编码在Checkpoints中。

Checkpoints检查点有两种作用：第一是防止分叉，如果有人试图从检查点之前的区块进行分叉，当前节点不会接受这个分叉；也用于保护网络不受全网51%的算力攻击，因为攻击者不可能逆转检查点之前的交易。第二是用于节点间的快速同步，我们将在第10章中详细讲解。

3.3.5 初始化数据库（持久化存储）

创建一条公链，需要将链上的所有数据（包含块信息、交易信息等）存储在本地键值数据库中。在比原链中使用LevelDB来存储链上数据，代码如下：

```
node/node.go
coreDB := dbm.NewDB("core", config.DBBackend, config.DBDir())
store := leveldb.NewStore(coreDB)

database/leveldb/store.go
type Store struct {
    db    dbm.DB
    cache blockCache
}
```

dbm使用tendermint框架的db管理库。dbm.NewDB返回一个DB对象，DB对象提供了数据库接口和许多方法实现，包括使用内存映射、文件系统目录结构、GO中LevelDB等的实现。

dbm.NewDB返回一个DB对象，需要传入三个参数：db的名称，db使用的键值数据库（默认为LevelDB），db数据存储的路径。leveldb.NewStore函数返回一个Store对象，即比原链对LevelDB进行了封装，在LevelDB的基础上增加了区块缓存（cache）、区块验证、区块状态、区块查询等功能。

3.3.6 初始化交易池

当交易被广播到网络中并且被矿工接收到时，矿工会将接收到的交易加入到本地的TxPool交易池中，TxPool对象的作用是管理本地交易池。交易池相当于一个缓冲区，它并不是无限大。默认情况下比原链中交易池最大可以存储10 000笔交易。如果超出这个阈值，则会返回"transaction pool reach the max number"提示。

```
node/node.go
txPool := protocol.NewTxPool()

protocol/txpool.go
func NewTxPool() *TxPool {
    return &TxPool{
        lastUpdated: time.Now().Unix(),
        pool:        make(map[bc.Hash]*TxDesc),
        utxo:        make(map[bc.Hash]bc.Hash),
        errCache:    lru.New(maxCachedErrTxs),
        msgCh:       make(chan *TxPoolMsg, maxMsgChSize),
    }
}
```

protocol.NewTxPool()返回一个TxPool实例对象。此处我们只介绍交易池初始化部分，交易池实现原理的代码将在6.10节中深入剖析。

3.3.7 创建一条本地区块链

当节点第一次启动时，判断本地持久化存储的状态，当状态为初始化时会初始化本地的区块链。区块链的第一个区块（创世区块）会被加入到区块高度为0的地方。代码如下：

```
node/node.go
chain, err := protocol.NewChain(store, txPool)
if err != nil {
    cmn.Exit(cmn.Fmt("Failed to create chain structure: %v",
err))
}
```

---

protocol.NewChain返回一个Chain对象，NewChain需要接收两个参数：Store区块链的存储对象，TxPool交易池。Chain对象管理着比原链的整个区块链条。代码如下：

```
protocol/protocol.go
func NewChain(store Store, txPool *TxPool) (*Chain, error) {
    c := &Chain{
        orphanManage:   NewOrphanManage(),
        txPool:         txPool,
        store:          store,
        processBlockCh: make(chan *processBlockMsg,
maxProcessBlockChSize),
    }
    c.cond.L = new(sync.Mutex)

    storeStatus := store.GetStoreStatus()
    if storeStatus == nil {
        if err := c.initChainStatus(); err != nil {
            return nil, err
        }
        storeStatus = store.GetStoreStatus()
    }

    var err error
    if c.index, err = store.LoadBlockIndex(); err != nil {
        return nil, err
    }
```

```
    c.bestNode = c.index.GetNode(storeStatus.Hash)
    c.index.SetMainChain(c.bestNode)
    go c.blockProcesser()
    return c, nil
}
```

NewChain函数的执行可分为下面几个步骤：

1）实例化Chain对象。

2）store.GetStoreStatus获取本地区块链的存储状态，如果状态为nil则说明区块链未被初始化。执行initChainStatus初始化本地区块链，该函数初始化创世区块（第一个区块）并添加到本地链上。

3）store.LoadBlockIndex加载块索引，从数据库中读取所有Block Header信息并缓存在内存中，目的是加速访问区块头信息。

4）c.index.SetMainChain，设置当前节点已同步的最新区块。

5）go c.blockProcesser()，启动一个go rutine，用于更新本地区块链上的区块信息。

3.3.8 初始化本地钱包

默认情况下比原链节点会启用本地钱包功能。代码实例如下：

```
node/node.go
hsm, err := pseudohsm.New(config.KeysDir())
if err != nil {
    cmn.Exit(cmn.Fmt("initialize HSM failed: %v", err))
}

if !config.Wallet.Disable {
    walletDB := dbm.NewDB("wallet", config.DBBackend,
config.DBDir())
    accounts = account.NewManager(walletDB, chain)
    assets = asset.NewRegistry(walletDB, chain)
    wallet, err = w.NewWallet(walletDB, accounts, assets, hsm,
chain)
    if err != nil {
        log.WithField("error", err).Error("init NewWallet")
    }

    // trigger rescan wallet
    if config.Wallet.Rescan {
        wallet.RescanBlocks()
    }
}
```

在比原链的节点启动时，上述代码流程主要逻辑为：

1）创建加密机hsm对象，hsm对象管理keystore文件，该文件是存储私钥的一种格式（JSON）。keystore是一串代码，本质上是加密后的私钥，需配合钱包的密码来使用。

2）创建钱包数据库。

3）创建账户管理对象。

4）创建资产管理对象。

5）实例化Wallet对象。

6）RescanBlocks扫描本地所有区块，触发钱包更新操作。

3.3.9 初始化网络同步管理

P2P通信模块主要由SyncManager管理，SyncManager负责节点业务层信息的同步工作，即区块和交易信息的同步。代码如下：

```
node/node.go
const (
    maxNewBlockChSize = 1024
)

newBlockCh := make(chan *bc.Hash, maxNewBlockChSize)
syncManager, _ := netsync.NewSyncManager(config, chain, txPool,
newBlockCh)
go newPoolTxListener(txPool, syncManager, wallet)
```

主要参数说明如下：

· newBlockCh：通道用于新挖掘出的区块进行快速广播给其他节点。通道大小为1024。

· netsync.NewSyncManager：实例化syncManager同步管理对象，它管理节点与节点之间的区块、交易信息同步。

· newPoolTxListenner：启动一个goroutine，监听交易池中的交易，将交易发送给syncManager同步管理对象或本地钱包。

详细实现机制将在第10章进行讲解。

3.3.10　初始化Pprof性能分析工具

Pprof是GO语言标准库中自带的性能分析工具。用于内存分析、CPU分析、代码追踪等，还可以生成性能分析图表。（详细参考https://golang.org/pkg/net/http/pprof/）。在比原链中默认不启用该功能，可以使用--prof_laddr参数启动代码性能分析功能，代码示例如下：

```
node/node.go
profileHost := config.ProfListenAddress
if profileHost != "" {
    go func() {
        http.ListenAndServe(profileHost, nil)
    }()
}
```

3.3.11 初始化CPU挖矿功能

在比原链节点源码中，只提供了CPU设备的挖矿功能，以目前全网的算力来看，CPU设备挖矿几乎挖不到BTM币了。目前主流的挖矿设备，有比特大陆定制的挖矿芯片或各大矿池使用GPU设备挖矿。挖矿和矿池细节将在第13章中详细解读。代码实例如下：

```
node/node.go
node.cpuMiner = cpuminer.NewCPUMiner(chain, accounts, txPool,
newBlockCh)
node.miningPool = miningpool.NewMiningPool(chain, accounts,
txPool, newBlockCh)

if config.Simd.Enable {
    tensority.UseSIMD = true
}
```

其中，simd参数用于Tenaority CPU指令的优化。

3.4 bytomd守护进程的启动方式和停止方式

我们在GO语言下实现守护进程的方式一般是，监听标准的SIGTERM信号。在监听到SIGTERM信号后，进程处于阻塞状态，以实现守护进程。只有当进程收到来自外部的SIGTERM信号时，进程则处于非阻塞状态，实现进程退出。Linux信号参考http://man7.org/linux/man-pages/man7/signal.7.html。代码实例如下：

```
cmd/bytomd/commands/run_node.go
func runNode(cmd *cobra.Command, args []string) error {
    // ...
    n.RunForever()
    // ...
}

node/node.go
func (n *Node) RunForever() {
    // Sleep forever and then...
    cmn.TrapSignal(func() {
        n.Stop()
    })
}

vendor/github.com/tendermint/tmlibs/common/os.go
func TrapSignal(cb func()) {
    c := make(chan os.Signal, 1)
    signal.Notify(c, os.Interrupt, syscall.SIGTERM)
    go func() {
        for sig := range c {
            fmt.Printf("captured %v, exiting...\n", sig)
            if cb != nil {
                cb()
            }
            os.Exit(1)
```

```
        }
    }()
    select {}
}
```

---

signal.Notify监听中断和Term信号。启用goroutine取c对象，select进入阻塞状态。当进程接收到Term信号则通知c对象，执行os.Exit退出守护进程。

发送Term信号有两种方式：一种是执行命令kill-15 pid；另一种是进程运行在前台。

当守护进程接收到Term信号后就停止运行，在其退出之前需要做扫尾工作，如退出挖矿模式，退出P2P同步功能等。代码示例如下：

```
node/node.go
func (n *Node) OnStop() {
    n.BaseService.OnStop()
    if n.miningEnable {
        n.cpuMiner.Stop()
    }
    if !n.config.VaultMode {
        n.syncManager.Stop()
    }
}
```

3.5 本章小结

本章从源码的角度分析了bytomd启动过程中的Node对象创建和初始化，以及总结bytomd实现的逻辑。

第4章

接口层

4.1 概述

在比原链中，API Server是重要组件，是一个监听请求、处理请求、响应请求的服务端。其主要功能包括：接收用户发送的请求，按照相应的路由规则分发请求，最终将请求处理后的结果返回给用户。

本章将从源码角度分析比原链中API Server的工作流程。内容如下：

· 使用GO语言实现一个简易的HTTP Server，让读者了解HTTP服务的创建过程。

· API Server创建HTTP服务的过程。

· HTTP请求的完整生命周期。

· 比原链的接口，调用工具等。

4.2 实现一个简易HTTP Server

GO语言的标准库net/http提供了与HTTP编程相关的接口，封装了内部TCP连接和报文解析等功能，使用者只需要http.request和http.ResponseWriter两个对象交互就足够了。我们实现了handler回收函数，请求通过参数传递进来，然后根据请求的数据做处理，把结果写到Response中。下面我们用GO语言实现一个简易的HTTP服务，代码如下：

```
package main

import (
    "net"
    "net/http"
)

func sayHello(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello, World!"))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", sayHello)

    server := &http.Server{
        Handler: mux,
    }

    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        panic(err)
    }

    err = server.Serve(listener)
```

```
    if err != nil {
        panic(err)
    }
}
```

当运行上面的HTTP服务后，请求本地的8080端口，HTTP服务会返回“Hello，World”信息。

HTTP服务的创建主要由三部分组成：

· 实例化http.NewServeMux()得到mux路由。为mux.Handle添加多个有效的router路由项。每一个路由项由HTTP请求方法（GET、POST、PUT、DELET）、URL和Handler回调函数组成。

· 监听本地的8080端口。

· 将监听地址作为参数，最终执行Serve（listener）开始服务于外部请求。

4.3 API Server创建HTTP服务

4.3.1 创建API对象

API对象负责整个API Server的管理工作，在运行API Server之前，首先要对其初始化。代码如下：

```
node/node.go
func (n *Node) initAndstartAPI Server() {
     n.api = api.NewAPI(n.syncManager, n.wallet, n.txfeed,
n.cpuMiner, n.miningPool, n.chain, n.config, n.accessTokens)

    listenAddr := env.String("LISTEN", n.config.ApiAddress)
    env.Parse()
    n.api.StartServer(*listenAddr)
}

api/api.go
func NewAPI(sync *netsync.SyncManager, wallet *wallet.Wallet,
txfeeds *txfeed.Tracker, cpuMiner *cpuminer.CPUMiner,
miningPool *miningpool.MiningPool, chain *protocol.Chain,
config *cfg.Config, token *accesstoken.CredentialStore) *API {
    api := &API{
        sync:          sync,
        wallet:        wallet,
        chain:         chain,
        accessTokens:  token,
        txFeedTracker: txfeeds,
        cpuMiner:      cpuMiner,
        miningPool:    miningPool,
    }
    api.buildHandler()
    api.initServer(config)
```

```
    return api
}
```

---

api.NewAPI初始化API对象。API Server管理的事情很多，所以参数也相对较多，例如：

· listenAddr：本地端口，如果系统没有设置LISTEN环境变量，则使用config.ApiAddress配置地址，默认为9888。

· n.api.StartServer：监听本地的地址端口，启动HTTP服务。

NewAPI函数有三个操作：

· 实例化API对象。

· api.buildHandler添加router路由项。

· api.initServer实例化http.Server，配置auth验证等。

4.3.2 创建路由项

匹配router路由项的目的是将HTTP请求交给对应的回调函数。比原链代码中有很多路由项，这里只介绍与账号相关的handler，其他的handler大同小异。代码如下：

```
func (a *API) buildHandler() {
    walletEnable := false
    m := http.NewServeMux()

    // ...
    m.Handle("/net-info", jsonHandler(a.getNetInfo))
    // ...

    handler := latencyHandler(m, walletEnable)
    handler = maxBytesHandler(handler) // TODO(tessr): consider
moving this to non-core specific mux
    handler = webAssetsHandler(handler)
    handler = gzip.Handler{Handler: handler}
    // ...
}
```

我们使用Golang标准库http.NewServeMux()创建一个router，提供请求的路由分发功能。一条router由url和对应的handle回调函数组成。当我们请求的url匹配到/net-info时，API Server会执行a.getNetInfo回调函数，并将用户的传参带过去。

额外的handler处理包括：

· latencyHandler：当请求找不到网址路径时，将请求重定向到"/error"错误路径。

· maxBytesHandler：限制请求的字节大小，一个请求不允许超过10MB。

· webAssetsHandler：添加dashboard和equity的路由项，dashboard和equity页面代码被硬编码到bytomd中，路径分别在dashboard/dashboard.go和equity/equity.go。

· gzip.Handler：启用gzip数据压缩，需指定gzip.BestSpeed压缩级别（默认level 1）。最高可指定level 9。随着压缩级别增加，CPU耗时也会增加，可根据HTTP服务接收的数据量调整该压缩级别。

4.3.3 实例化http.Server

http.Server是一个HTTP服务对象，是运行HTTP服务的基础。代码如下：

```
func (a *API) initServer(config *cfg.Config) {
    // ...

    coreHandler.wg.Add(1)
    mux := http.NewServeMux()
    mux.Handle("/", &coreHandler)

    handler = mux
    if config.Auth.Disable == false {
        handler = AuthHandler(handler, a.accessTokens)
    }
    handler = RedirectHandler(handler)

    secureheader.DefaultConfig.PermitClearLoopback = true
    secureheader.DefaultConfig.HTTPSRedirect = false
    secureheader.DefaultConfig.Next = handler

    a.server = &http.Server{
        Handler:      secureheader.DefaultConfig,
        ReadTimeout:  httpReadTimeout,
        WriteTimeout: httpWriteTimeout,
        TLSNextProto: map[string]func(*http.Server, *tls.Conn,
http.Handler){},
    }

    coreHandler.Set(a)
}
```

实例化http.Server分为以下几步：

1）AuthHandler：启用auth验证功能，用户打开dashboard时需要输入access token进行验证登录。默认开启auth功能。

2）RedirectHandler：重定向，当用户输入"/"时，默认跳转到"/dashboard/"，并设置状态为302。

3）http.Server：设置读写超时时间，并实例化http.Server。

到目前为止，所有的handle已经被注册到HTTP服务器中，并实例化http.Server服务对象，下面操作监听端口，启动HTTP服务。

4.3.4 启动API Server

通过Golang标准库net.listen方法，监听本地的地址端口。由于HTTP服务是一个持久运行的服务，我们启动一个goroutine专门运行HTTP服务。当运行a.server.Serve没有任何报错时，可以看到服务器上启动的9888端口，此时API Server已经处于等待接收用户请求的状态，每当接收到请求后，API Server会单独启用一个goroutine来处理该请求。代码如下：

```
api/api.go
func (a *API) StartServer(address string) {
    log.WithField("api address:", address).Info("Rpc listen")
    listener, err := net.Listen("tcp", address)
    if err != nil {
        cmn.Exit(cmn.Fmt("Failed to register tcp port: %v",
err))
    }

    go func() {
        if err := a.server.Serve(listener); err != nil {
            log.WithField("error", errors.Wrap(err,
"Serve")).Error("Rpc server")
        }
    }()
}
```

server.Serve的代码流程如下：

1）a.server.Serve内部启动一个for循环，在循环体中接收accept请求。

2）对每个请求实例化一个conn，并开启一个goroutine为这个请求执行handle。

3）c.readRequest（ctx）读取每个请求的内容。

4）根据request选择handler，并且进入到这个handler的ServeHTTP。

5）调用w.finishRequest()写入conn缓冲区的数据，关闭TCP连接，结束请求。

4.3.5 接收并响应请求

当我们使用curl命令发起一个HTTP请求，获取当前节点运行的网络状态，API Server响应给我们如下数据：

```
$ curl -s http://localhost:9888/net-info|jq .
{
  "status": "success",
  "data": {
    "listening": true,
    "syncing": false,
    "mining": false,
    "peer_count": 0,
    "current_block": 63304,
    "highest_block": 63304,
    "network_id": "mainnet",
    "version": "1.0.5+2bc2396a"
  }
}
```

API Server的处理过程如下：

```
api/api.go
m.Handle("/net-info", jsonHandler(a.getNetInfo))
```

API Server解析HTTP头，匹配到path路径/net-info，跳转至a.getNetInfo回调函数。代码如下：

```
api/nodeinfo.go
func (a *API) getNetInfo() Response {
    return NewSuccessResponse(a.GetNodeInfo())
}
```

GetNetInfo函数通过P2P、cpuMinner等对象获取信息。getNetInfo函数将NetInfo结构体序列化至JSON格式，并通过实例化Response返回给用户。代码如下：

```
api/api.go
type Response struct {
    Status      string      `json:"status,omitempty"`
    Code        string      `json:"code,omitempty"`
    Msg         string      `json:"msg,omitempty"`
    ErrorDetail string      `json:"error_detail,omitempty"`
    Data        interface{} `json:"data,omitempty"`
}

func NewSuccessResponse(data interface{}) Response {
    return Response{Status: SUCCESS, Data: data}
}
```

NewSuccessResponse实例化Response，并设置HTTP状态码为200。Response结构描述如下：

- Status：状态码字符描述。
- Code：状态码。
- Msg：状态码对应的字符描述。
- ErrorDetail：错误信息，请求成功时，错误信息为空。
- Data：响应用户的数据。

4.4 HTTP请求的完整生命周期

API Server服务提供HTTP短连接（非持久连接）服务请求。客户端和服务端进行一次HTTP请求/响应之后，就关闭连接。下一次的HTTP请求/响应操作则需要重新建立连接。一次HTTP请求的完整生命周期如图4-1所示。

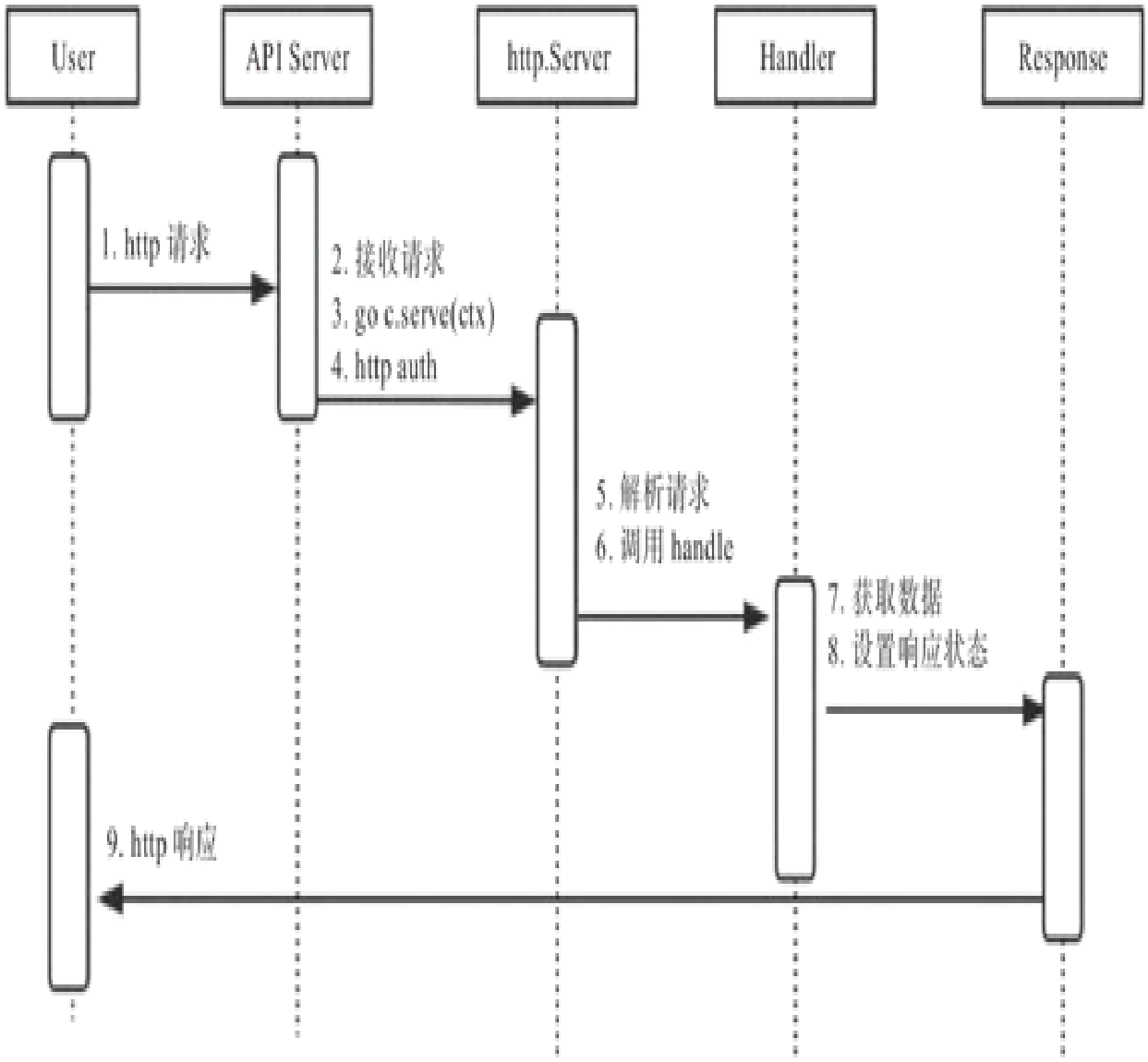


图4-1　HTTP请求的完整生命周期

HTTP请求的完整生命周期过程如下：

1）用户向API Server发出HTTP请求。

2）API Server接收到用户发出的请求。

3）启用goroutine处理接收到的请求。

4）验证请求内容中的auth信息。

5）解析请求内容。

6）调用路由项对应的handle回调函数。

7）获取handle的数据信息。

8）设置响应状态码。

9）响应用户的请求。

4.5 比原链API接口描述

比原链中有很多接口，下面分类介绍。

1. 钱包相关的接口

接口名称	说明
/create-account	创建账户
/list-accounts	显示所有账户
/delete-account	删除账户
/create-account-receiver	创建账户地址
/list-addresses	显示所有地址
/validate-address	验证地址
/list-pubkeys	显示账户公钥
/get-mining-address	获取挖矿地址
/set-mining-address	设置挖矿地址
/get-coinbase-arbitrary	获取 coinbase 交易备注信息
/set-coinbase-arbitrary	设置 coinbase 交易备注信息
/create-asset	创建资产
/update-asset-alias	更新资产别名
/get-asset	显示资产信息
/list-assets	显示所有资产
/create-key	创建 key
/list-keys	显示所有 key
/delete-key	删除 key
/reset-key-password	重置 key 密码
/check-key-password	检查 key 密码
/sign-message	对消息进行签名
/build-transaction	创建一笔交易
/sign-transaction	使用账户密钥对交易进行签名
/get-transaction	根据交易 hash 值获取交易信息
/list-transactions	显示交易信息
/list-balances	显示余额
/list-unspent-outputs	显示账户未使用的输出
/backup-wallet	备份钱包
/restore-wallet	恢复钱包
/rescan-wallet	重新扫描交易信息到钱包
/wallet-info	显示钱包信息

2. token验证相关的接口

接口名称	说明
/create-access-token	创建 token
/list-access-tokens	显示所有 token
/delete-access-token	删除指定 token
/check-access-token	验证 token

3. 交易相关的接口

接口名称	说明
/submit-transaction	提交签名的交易
/estimate-transaction-gas	估算交易的手续费
/get-unconfirmed-transaction	获取指定的未确认的交易
/list-unconfirmed-transactions	显示所有未确认的交易
/decode-raw-transaction	解码原始交易

4. 块信息相关的接口

接口名称	说明
/get-block	获取某高度的区块信息，输入参数为指定高度
/get-block-hash	获取指定高度区块的哈希值
/get-block-header	获取指定高度区块的区块头
/get-block-count	获取区块数量
/get-difficulty	获取当前网络的标准难度
/get-hash-rate	获取当前网络的算力

5. 矿池相关的接口

接口名称	说明
/get-work	从网络节点获取最新的待计算区块，内容为 Varint 编码压缩形式
/get-work-json	从网络节点获取最新的待计算区块，内容为 JSON 结构化的形式
/submit-work	向网络节点提交计算结果，以 Varint 编码压缩形式
/submit-work-json	向网络节点提交计算结果，以 JSON 结构化的形式
/is-mining	本地节点是否已启用挖矿功能
/set-mining	本地节点启用 / 关闭挖矿功能

6. 合约相关的接口

接口名称	说明
/verify-message	验证已签名的消息
/decode-program	将程序解码为指令和数据
/compile	编译合约

7. P2P网络相关的接口

接口名称	说明
/net-info	显示当前网络信息
/list-peers	显示当前节点连接的对等节点
/disconnect-peer	断开指定的对等节点连接
/connect-peer	连接指定的对等节点

4.6 API接口调用工具

在公链开发中，我们需要对API接口进行调用。调用工具有很多，本节推荐使用两种方式：curl命令或Postman图形工具，在这里以create-key接口为例演示。

4.6.1 使用curl命令行调用API接口

curl命令是利用URL语法在命令行方式下工作的开源文件传输工具。curl命令发出网络请求，然后得到并提取数据，显示在“标准输出”中。

我们使用curl命令请求访问create-key接口，创建私钥，并返回密钥信息。代码如下：

```
$ curl -X POST http://localhost:9888/create-key -d '{ "alias"
:"user1" , "password":"123456"}'

{"alias":"user1",
"xpub":"e441f31e16276902d305ab5eb6fb686ec3954a59c00712c5ee7565c
93f16589b75ea5e11aa09122962ff7bd04c9dfff5027ca10ac9bea647229348
50f6954f56",
"file":"C:\\Users\\Mac\\AppData\\Roaming\\Bytom\\keystore\\UTC-
-2018-09-21T11-51-52.401546400Z--e35c76e1-0f6c-4431-bb0f-
eca37206d885"}}
```

curl命令参数如下：

- -X：指定请求方式，包括GET、POST、DELETE等。
- -d：使用POST方式向API Server发送数据。

其他参数请读者自行学习。

API接口返回字段如下：

- alias：别名，在本例中是user1。
- xpub：公钥。
- file：密钥文件的存储路径。

4.6.2 使用Postman调用API接口

Postman是一个图形化接口请求工具，可以帮助我们更方便地模拟各种请求（GET、POST及其他方式）来调用接口。

请读者下载Postman（官网下载：https://www.getpostman.com/apps）。

使用Postman调用API接口，选择POST请求方式，如图4-2所示。输入http://127.0.0.1:9888/create-key；在Body标签下的raw选择Text格式，输入：{"alias": "user0", "password": "123456"}；点击send按钮，返回与上述curl下create-key格式相同的内容：

图4-2　postman工具

在Postman下调用其他接口同理，此处不再赘述。

4.7 比原链HTTP错误码一览

比原链与HTTP相关的错误码在bytom/api/errors.go中，简单说明如下。

- 0xx：API错误。
- 1xx：网络错误。
- 2xx：签名相关的错误。
- 7xx：交易相关的错误。
- 72x-73x：构建交易错误。
- 73x-75x：验证交易错误。
- 76x-78x：虚拟机错误。
- 8xx：HSM相关错误。

4.8 本章小结

本章介绍了如何使用GO语言标准库实现一个简易的HTTP Server。其中包括HTTP服务的工作原理，接收、处理、响应等过程。本章以图文的方式介绍了一个HTTP请求的完整生命周期。最后介绍了公链如何提供HTTP相关的接口。比原链API接口中已经包含了大部分公链所具备的接口，相比其他公链，比原链扩展了其他接口，比如矿池相关接口、token验证相关接口等。

第5章

内核层：区块与区块链

5.1 概述

区块链是由包含交易信息的区块按照时间先后顺序依次连接起来的数据结构。这种数据结构是一个形象的链表结构，所有区块有序地连接在同一条区块链上，每个区块通过一个哈希指针指向前一个区块。哈希指针其实是前一个区块头进行SHA256哈希得到的，通过这个哈希值，可以唯一地识别一个区块。把每个区块连接到其区块头中前一个区块哈希值代表的区块后面，就可以构建一条完整的区块链。

本章主要内容包括：

- 区块和区块链数据结构。
- 创世区块数据结构及初始化。
- 区块验证、区块上链以及区块难度的计算方式。
- 孤块管理。

5.2 区块

区块是区块链中数据存储的最小单元。下面介绍区块的数据结构、区块头、区块标识符、创世区块等概念，并介绍区块的基本操作。

5.2.1 区块的数据结构

首先说明一下，在比原链中，数据结构的定义是分层的：

· types层：原始数据结构层，用于在节点与节点之间相互传输。

· bc层：虚拟机计算层，专门用于提高交易验证的性能。

区块（block）由区块头（Header）和区块体（Body）组成，区块头记录了区块的元数据信息，区块体包含了打包的所有交易信息。我们可以把区块形象地类比成一本账薄，交易就是记录在账薄中的一笔笔转账记录，区块头则是账本的扉页，记录了账本的交易概括信息。区块的数据结构如下所示：

```
protocol/bc/block.go
type Block struct {
    *BlockHeader
    ID           Hash
    Transactions []*Tx
}
```

区块结构如表5-1所示。

表5-1 区块结构

字段	字节数	说明
BlockHeader	变长	区块头
ID	32	区块头ID，当前区块的唯一标识
Transactions	变长	交易记录

5.2.2 区块头的数据结构

区块头主要由三大部分组成：区块链的连接信息、挖矿信息、区块交易信息。区块链的连接信息是指区块中引用的父区块哈希值，用于将区块与区块链中前一区块相连接。挖矿信息主要包括Timestamp、Nonce和Bits。挖矿时，矿机可以不断地修改Nonce值，计算满足Bits要求的区块头。区块交易信息主要是区块中交易的默克尔树（Merkle Tree）。比原链区块头保存了两个默克尔树：由区块中所有交易哈希值组成的交易信息的默克尔树；由区块中所有交易验证结果组成的交易结果的默克尔树。区块头的数据结构如下所示：

```
protocol/bc/bc.pb.go
type BlockHeader struct {
    Version               uint64
    Height                uint64
    PreviousBlockId       *Hash
    Timestamp             uint64
    TransactionsRoot      *Hash
    TransactionStatusHash *Hash
    Nonce                 uint64
    Bits                  uint64
    TransactionStatus     *TransactionStatus
}
```

区块头结构如表5-2所示。

表5-2　区块头结构

字段	字节数	说明
version	8	区块版本号
height	8	区块高度，当前区块与创世区块的距离
previousBlockId	32	父区块哈希值，区块引用的父区块头的哈希值
timestamp	8	时间戳，该区块产生的近似时间（精确到秒的 Unix 时间戳）
transactionsRoot	32	merkle 根，区块中包含的所有交易的哈希值生成
transactionStatusHash	32	交易状态 merkle 根，区块中所有交易验证结果的哈希值生成
nonce	8	工作量证明算法的计数器
bits	8	难度，区块工作量证明算法的难度目标
transactionStatus	变长	交易状态，每笔交易验证后的状态

5.2.3 区块标识符

区块标识符是区块的唯一标识。根据区块标识符，我们可以唯一确定一个区块。有两种区块标识符：区块头哈希值和区块高度。

区块头哈希值通过使用SHA256算法对区块头进行二次哈希得到32位哈希值。“区块头哈希值”也称为“区块哈希值”。SHA256算法是数字签名标准中定义的数字签名算法，主要用于计算消息的摘要信息。对于任意长度的消息，经过SHA256算法都会生成一个长度为32字节的摘要数据。接收端收到消息时，可以根据摘要信息验证数据是否发生篡改。为什么我们根据摘要信息就可以判断消息是否发生改变？这主要是由SHA256算法的性质决定的。

SHA256算法有如下特性：

- 性质1：任何人都不能从摘要信息中复原原始消息。

- 性质2：两个不同的消息不会产生相同的消息摘要。

由于SHA256算法的性质2，我们也将生成的消息摘要称为消息指纹。因此，每个区块头的哈希值是不同的，我们就可以通过区块头的哈希值唯一确定一个区块。

还有一点需要注意，区块头哈希值并不保存在当前区块的数据结构中。那我们又是如何通过区块头哈希值检索区块的呢？总不能从最高区块开始遍历所有区块，查找哪个区块头的PreviousBlockHash等于要检索的区块。实际上，当节点从网络中同步一个区块的时候，节点会计算区块头哈希值，区块头哈希值会作为key、区块数据作为value存储到LevelDB中。这样我们就可以通过区块头哈希值快速检索到对应的区块。

除区块头哈希值可以唯一确定一个区块外，在某些情景下，通过区块高度也可以唯一确定一个区块。为什么是“有些情景下”才能唯一确定呢？主要原因是存在区块软分叉。我们知道当有多个区块同时被矿工发现时，这时产生的多个区块都会引用同一个父区块，虽然它们的区块头哈希值是不同的，但是它们此时的高度是相同的。这时我们通过区块高度是无法唯一确定一个区块的。区块软分叉是一个临时

状态，当分叉数达到一定阈值时，共识算法会比较每条分叉的工作量，保留总工作量最多的那条分叉。我们通常认为，区块高度比当前最高区块高度小6的区块不存在软分叉。

5.2.4 创世区块

区块链中的第一个区块创称为创世区块，它是区块链里所有区块的共同祖先。一般情况下创世区块被硬编码到代码核心中，每一个节点都始于同一个创世区块，这能确保创世区块不会被修改。每个节点都把创世区块作为区块链的首区块，从而构建了一个安全可信的区块链。

可以通过比原链命令行工具bytomcli获取创世区块的信息，命令执行如下：

```
$ ./bytomcli get-block 0

{
  "bits": 2161727821137910500,
  "difficulty": "15154807",
  "hash":
"a75483474799ea1aa6bb910a1a5025b4372bf20bef20f246a2c2dc5e12e8a0
53",
  "height": 0,
  "nonce": 9253507043297,
  "previous_block_hash":
"0000000000000000000000000000000000000000000000000000000000000
00",
  "size": 546,
  "timestamp": 1524549600,
  "transaction_merkle_root":
"58e45ceb675a0b3d7ad3ab9d4288048789de8194e9766b26d8f42fdb624d43
90",
  "transaction_status_hash":
"c9c377e5192668bc0a367e4a4764f11e7c725ecced1d7b6a492974fab1b6d5
bc",
  "transactions": [
  // ...
  ],
  "version": 1
}
```

创世区块说明：

· bits：目标难度值，只有当挖矿时计算的哈希值小于或等于该值，矿工节点才能打包交易构建区块。创世区块的难度值为2161727821137910500。创世区块不是矿工节点挖出来的，而是一个硬编码的区块。所以该难度值没有任何意义，理论上是可以随便写的。

· difficulty：难度值，由bits值计算出来，我们只看bits值是很难判断某次挖矿的难度的，而通过难度值可以直观地比较出两个区块在挖矿时的难度。

· height：区块高度，创世区块高度为0，区块高度从0开始计数。

· nonce：随机数，挖矿时矿工节点反复尝试不同的nonce值来生成小于等于bits的区块头哈希值。

· previous_block_hash：当前区块的父区块哈希值，由于创世区块是第一个块，它是没有父区块的，因此在硬编码时，给它一个默认的区块哈希值为000000...000000

· timestamp：区块生成时间。创世区块的时间戳为1524549600，标准时间为2018-04-24 14：00：00，这个时间也是比原链主网上线的时间。

· transaction_merkle_root：创世区块的merkle树根节点，根据交易生成不同的merkle树根节点，用于验证当前区块的交易是否完整。

· transaction_status_hash：区块交易状态哈希值。

· transactions：当前块中的交易信息。

5.2.5 生成创世区块

当我们启动一个节点的时候，节点会先尝试从本地LevelDB数据库中加载区块信息。如果节点没有从LevelDB中获取到区块，则节点会认为自己是一个全新的节点。此时，节点会根据硬编码的创世区块信息初始化本地链。硬编码的创世区块信息如下所示：

```
config/genesis.go
func mainNetGenesisBlock() *types.Block {
    tx := genesisTx()
    txStatus := bc.NewTransactionStatus()
    if err := txStatus.SetStatus(0, false); err != nil {
        log.Panicf(err.Error())
    }
    txStatusHash, err :=
types.TxStatusMerkleRoot(txStatus.VerifyStatus)
    // ...

    merkleRoot, err := types.TxMerkleRoot([]*bc.Tx{tx.Tx})
    // ...
    block := &types.Block{
        BlockHeader: types.BlockHeader{
            Version:   1,
            Height:    0,
            Nonce:     9253507043297,
            Timestamp: 1524549600,
            Bits:      2161727821137910632,
            BlockCommitment: types.BlockCommitment{
                TransactionsMerkleRoot: merkleRoot,
                TransactionStatusHash:  txStatusHash,
            },
        },
        Transactions: []*types.Tx{tx},
    }
    return block
}
```

mainNetGenesisBlock方法用来生成创世区块。在mainNetGenesisBlock方法中，首先通过genesisTx方法构造一笔Coinbase交易（Coinbase交易的含义参见6.3.2节）。Coinbase交易输入中附加信息为“Information is power.--Jan/11/2013.Computing

is power.--Apr/24/2018.”，这是为了纪念计算机天才Aaron Swartz，纪念他致力于网络信息开放的创新精神。Coinbase交易的输出为1407000412500000000/1e8=1407000412.5，大约为14亿BTM币。这些币的锁定脚本为00148c9d063ff74ee6d9ffa88d83aeb038068366c4c4。这些币是为官方预留的，部分用于与交易所兑换代币。

构造完Coinbase交易后，节点将Coinbase交易的验证状态设置为成功。这里可能会造成误解，明明传入的值是false，怎么会设置为成功呢。这是因为比原链中使用StatusFail记录交易状态，记录交易是否失败。设置为false则说明交易是成功的。之后，节点根据交易信息生成默克尔树根节点，构造区块结构。

5.2.6 区块验证

区块验证是保证区块链不产生分叉的重要手段。如果没有区块验证过程，则在同步区块的过程中节点间会产生较多的分叉。我们知道分叉会对链上资金和财产安全造成极大的威胁。

一般会在四种情况下对区块进行验证：

· 挖矿节点在成功挖掘到一个区块并向链上提交区块时，节点会先验证区块是否合法。

· 用户通过API接口向节点提交区块到区块链时，节点会验证区块是否合法。

· 同步区块时，节点收到其他节点同步过来的区块，节点会先验证同步的区块是否合法，如果合法则将其加入到本地链。

· 矿池中的节点向矿池提交工作时，矿池会验证矿机提交的区块。

以太坊的区块验证策略比较复杂，每次节点在同步区块时，机器都会嗡嗡作响。相比之下，比原链提供了一种既安全又简单快速的验证方法。代码示例如下：

```
protocol/block.go
func (c *Chain) processBlock(block *types.Block) (bool, error)
{
   blockHash := block.Hash()
   if c.BlockExist(&blockHash) {
      log.WithFields(log.Fields{"hash": blockHash.String(),
"height": block.Height}).Info("block has been processed")
      return c.orphanManage.BlockExist(&blockHash), nil
   }
   if parent := c.index.GetNode(&block.PreviousBlockHash);
parent == nil {
      c.orphanManage.Add(block)
      return true, nil
   }
   if err := c.saveBlock(block); err != nil {
```

```
        return false, err
    }
    bestBlock := c.saveSubBlock(block)
    bestBlockHash := bestBlock.Hash()
    bestNode := c.index.GetNode(&bestBlockHash)
    if bestNode.Parent == c.bestNode {
        log.Debug("append block to the end of mainchain")
        return false, c.connectBlock(bestBlock)
    }
    if bestNode.Height > c.bestNode.Height &&
bestNode.WorkSum.Cmp(c.bestNode.WorkSum) >= 0 {
        log.Debug("start to reorganize chain")
        return false, c.reorganizeChain(bestNode)
    }
    return false, nil
}
```

---

processBlock函数处理过程如下：

1）计算区块的哈希值，根据区块哈希值，查看本地链或孤块池中是否已经存在该区块。如果本地已经存在了相同的区块，则将区块直接丢弃。

2）根据区块中PreviousBlockHash从本地链中查找父区块。如果本地链中没有其父区块，则将区块加入到孤块池中。

3）如果本地链中存在父区块，则将区块加入本地链。但是区块在加入本地链时，需要对区块中的交易信息和元数据进行验证。

4）查找孤块池中是否存在可以上链的孤块。如果存在，则将孤块池中的孤块依次上链，并得到此时链上最高块bestNode。

5）如果bestNode的父区块等于本次操作之前链上的最高区块c.betNode，则将bestNode连接到c.betNode之后。否则说明bestNode并不在主链上，而是被添加到了侧链上。

6）如果bestNode被添加到侧链上，其高度大于主链的高度，工作量之和超过了主链的工作量，节点会重组区块链。

在步骤3）中，区块存储到链上时，需要对区块头的内容进行验证，代码示例如下：

```
protocol/validation/block.go
func ValidateBlockHeader(b *bc.Block, parent *state.BlockNode)
error {
   if b.Version < parent.Version {
      // ...
   }
   if b.Height != parent.Height+1 {
      // ...
   }
   if b.Bits != parent.CalcNextBits() {
      // ...
   }
   if parent.Hash != *b.PreviousBlockId {
      // ...
   }
   if err := checkBlockTime(b, parent); err != nil {
      // ...
   }
   if !difficulty.CheckProofOfWork(&b.ID,
parent.CalcNextSeed(), b.BlockHeader.Bits) {
      // ...
   }
   return nil
}
```

区块头验证主要包括以下6部分：

· 当前区块的版本号要与父区块的版本号相同。

· 当前区块的高度只能比父区块的高度大1。

· 当前区块的难度目标要等于根据父区块高度计算出的下一个区块的难度目标。

· 当前区块的父区块的哈希值要等于父区块的哈希值。

· 当前区块的时间戳不能比当前时间延后一小时，并且不能早于之前11个区块时间戳的中位数。

· 验证区块的哈希值是否等于给定的难度目标。

5.2.7 计算下一个区块的难度目标

根据父区块高度计算下一个区块的难度目标，算法如下：

```
protocol/state/blockindex.go
func (node *BlockNode) CalcNextBits() uint64 {
   if node.Height%consensus.BlocksPerRetarget != 0 ||
node.Height == 0 {
      return node.Bits
   }

   compareNode := node.Parent
   for compareNode.Height%consensus.BlocksPerRetarget != 0 {
      compareNode = compareNode.Parent
   }
   return
difficulty.CalcNextRequiredDifficulty(node.BlockHeader(),
compareNode.BlockHeader())
}
```

算法描述：如果父区块对consensus.BlocksPerRetarget（2016）取模不等于0，并且父区块的高度也不等于0，则下一个区块的难度与父区块的难度相同。否则进行难度动态调整（retarget），更多关于难度动态调整方法请参考第12章中的内容。

5.2.8 孤块管理

当节点收到了一个有效的区块，而在现有的主链中却未找到它的父区块，那么这个区块称为“孤块”。接收到的孤块会存储在孤块池中，直到它们的父区块被节点收到。节点一旦收到了父区块，就会将孤块从孤块池中取出，并且连接到它的父区块，让它作为区块链的一部分。

当两个或多个区块在很短的时间间隔内被挖出来，节点可能会以不同的顺序接收到它们，这时候孤块现象就会出现。我们假设有三个高度分别为100、101、102的块，分别以102、101、100的顺序被节点接收。此时节点将102、101放入到孤块管理缓存池中，等待它们的父块。当高度为100的区块被同步进来时，会经过验证区块和交易，然后存储到区块链上。这时会对孤块缓存池进行递归查询，根据高度为100的区块找到101的区块并存储到区块链上，再根据高度为101的区块找到102的区块并存储到区块链上。

孤块结构如下所示：

```
protocol/orphan_manage.go
type orphanBlock struct {
   *types.Block
   expiration time.Time
}
```

孤块结构说明如下：

· types.Block：孤块的区块结构。

· Expiration：孤块超时时间，如果一个孤块在Expiration时间内仍然没有上链，则这个孤块将会被丢弃。

孤块管理缓存池结构体如下所示：

```
type OrphanManage struct {
   orphan      map[bc.Hash]*orphanBlock
```

```
    prevOrphans map[bc.Hash][]*bc.Hash
    mtx         sync.RWMutex
}
```

孤块管理缓存池结构说明如下。

· Orphan：用于存储孤块，key为块哈希值，value为孤块结构体。

· prevOrphans：用于存储孤块的父块。

· mtx：互斥锁，保护map结构在多并发读写状态下保持数据一致。

孤块管理器的主要方法如下。

· BlockExist()：判断一个区块是否是孤块。

· Add()：向孤块池中加入新的孤块。

· Delete()：从孤块池中删除孤块。

· Get()：根据区块哈希从孤块池中获取孤块。

· GetPrevOrphans()：从孤块池中获取所有父区块哈希值为hash的孤块。

· orphanExpireWorker()：定时器，定时清理超时孤块。

· orphanExpire()：清理超时孤块的工作方法。

1）添加孤块到孤块池，代码示例如下：

```
func (o *OrphanManage) Add(block *types.Block) {
    blockHash := block.Hash()
    o.mtx.Lock()
    defer o.mtx.Unlock()
    if _, ok := o.orphan[blockHash]; ok {
        return
```

```
    }
    o.orphan[blockHash] = &orphanBlock{block,
time.Now().Add(orphanBlockTTL)}
    o.prevOrphans[block.PreviousBlockHash] =
append(o.prevOrphans[block.PreviousBlockHash], &blockHash)
}
```

添加孤块到孤块管理池的方法中，不仅仅是将孤块加入到孤块管理池中，而且还要在管理器中记录父区块与孤块的对应关系。这一步主要是为了在孤块上链时可以快速找到将当前孤块作为父区块的其他孤块。

2）从孤块池中删除孤块，代码示例如下：

```
func (o *OrphanManage) delete(hash *bc.Hash) {
    block, ok := o.orphan[*hash]
    if !ok {
        return
    }
    delete(o.orphan, *hash)
    prevOrphans, ok :=
o.prevOrphans[block.Block.PreviousBlockHash]
    if !ok || len(prevOrphans) == 1 {
        delete(o.prevOrphans, block.Block.PreviousBlockHash)
        return
    }
    for i, preOrphan := range prevOrphans {
        if preOrphan == hash {
            o.prevOrphans[block.Block.PreviousBlockHash] =
append(prevOrphans[:i], prevOrphans[i+1:]...)
            return
        }
    }
}
```

同样，从孤块池中删除孤块时，不仅需要清除孤块池中的记录，也需要请求孤块管理器中父区块与孤块对应关系的记录。

3）定时清理孤块，代码示例如下：

```
orphanExpireScanInterval = 3 * time.Minute

func (o *OrphanManage) orphanExpireWorker() {
   ticker := time.NewTicker(orphanExpireWorkInterval) //3分钟
   for now := range ticker.C {
      o.orphanExpire(now)
   }
   ticker.Stop()
}

func (o *OrphanManage) orphanExpire(now time.Time) {
   o.mtx.Lock()
   defer o.mtx.Unlock()
   for hash, orphan := range o.orphan {
      if orphan.expiration.Before(now) { //孤块超时时间60分钟
         o.delete(&hash)
      }
   }
}
```

定时清理孤块是为了防止大量无用的孤块长期占用内存，浪费资源。孤块中很可能会存在一些恶意节点构造的恶意块，这些块很可能已用了一个不存在的父区块，如果不清理这些区块，将会导致节点内存暴涨。孤块管理器会使用一个goroutine定时清理超时的孤块，定时器每3分钟执行一次，清理60分钟之前加入孤块池中的孤块。

5.3 区块链

区块链按照时间顺序将区块连接，最终形成一条以时间为序列的链表结构。下面详细介绍区块链的数据结构，以及区块上链、连接、重组等操作原理。

5.3.1 区块链的数据结构

链上操作主要通过定义Chain结构来实现，Chain提供了多种链上操作的方法：链的初始化，获取链上指定高度或者哈希值的区块，获取当前链的高度，验证提交到链上的区块，链上区块的连接以及链重组等操作。Chain结构如下所示：

```
protocol/protocol.go
type Chain struct {
    index          *state.BlockIndex
    orphanManage   *OrphanManage
    txPool         *TxPool
    store          Store
    processBlockCh chan *processBlockMsg

    cond     sync.Cond
    bestNode *state.BlockNode
}
```

Chain结构说明如下。

· index：存储在内存中的区块索引，该索引是一个树形结构，可以快速查找区块。

· orphanManage：孤块管理器。

· txPool：交易池。

· store：区块持久化存储接口。

· processBlockCh：待验证区块的缓存队列。

· cond：GO语言sync包中Cond结构，用于goroutine之间并发控制。

· bestNode：记录当前主链的最高区块。

链上的操作方法是整个比原系统中最重要的一部分，很多重要的方法都在这里实现了，例如交易的验证、区块连接、区块校验等。具体方法如下。

· func（c*Chain）BestBlockHeight()：获取主链的高度。

· func（c*Chain）BestBlockHash()：获取主链上目前高度最高的区块。

· func（c*Chain）BestBlockHeader()：获取主链上目前高度最高的区块头。

· func（c*Chain）InMainChain()：判断区块是否在主链上。

· func（c*Chain）CalcNextSeed()：计算Tensority算法的seed。

· func（c*Chain）CalcNextBits()：根据前父区块计算下一个区块的难度值。

· func（c*Chain）GetTransactionStatus()：获取区块中所有交易的状态。

· func（c*Chain）GetTransactionsUtxo()：获取区块中所有交易引用的UTXO。

· func（c*Chain）ValidateTx()：验证交易。

· func（c*Chain）ProcessBlock()：处理区块，通过校验的区块将被连接到主链上。

5.3.2 区块上链

区块在完成一系列的验证步骤后，就会被存储到链上，本小节中，我们将会介绍区块是怎么存储到链上的。这部分主要由protocol/block.go中的func（c*Chain）saveBlock（block*types.Block）方法实现。saveBlock方法主要通过以下几个步骤将区块上链。

步骤1：将区块存储到本地LevelDB数据库中。

在Chain的saveBlock方法中，在完成区块的校验后，会通过LevelDB提供的SaveBlock接口将区块存储到本地LevelDB数据库中。下面我们看一下这个接口的具体实现，代码示例如下：

```
database/leveldb/store.go
func (s *Store) SaveBlock(block *types.Block, ts
*bc.TransactionStatus) error {
   binaryBlock, err := block.MarshalText()
   // ...
   binaryBlockHeader, err := block.BlockHeader.MarshalText()
   // ...
   binaryTxStatus, err := proto.Marshal(ts)
   // ...
   blockHash := block.Hash()
   batch := s.db.NewBatch()
   batch.Set(calcBlockKey(&blockHash), binaryBlock)
   batch.Set(calcBlockHeaderKey(block.Height, &blockHash),
binaryBlockHeader)
   batch.Set(calcTxStatusKey(&blockHash), binaryTxStatus)
   batch.Write()

   return nil
}
```

在SaveBlock接口中，需要将区块的三种信息存储到LevelDB中：第一个是完整区块的序列化信息。这些信息的key将会以“B：”作为前缀，后接区块哈希值。第二个是区块头的序列化信息。这些信息的key将会以“BH”作为前缀，后接区块高度和区块哈希值。第三个是区

块中所有交易状态的序列化信息。这些信息的key将会以“BTS：”作为前缀，后接区块哈希值。

步骤2：从孤块池中删除已存储到LevelDB数据库中的区块，代码如下：

```
c.orphanManage.Delete(&bcBlock.ID)
```

步骤3：将区块更新到内存中的区块索引上，代码如下：

```
node, err := state.NewBlockNode(&block.BlockHeader, parent)
if err != nil {
   return err
}

c.index.AddNode(node)
```

在内存中，我们使用一个叫BlockIndex的结构进行组织并保存区块信息。在BlockIndex结构中，我们使用index做区块索引，在index中使用区块头哈希值做key。如果我们需要通过区块哈希值搜索区块，可以直接从index中快速搜索。同时我们还使用一个名为blockNode的数组结构存储区块信息。在mainChain中保存无分叉的主链数据，所有区块数据按照区块高度依次排列。当我们需要根据区块高度搜索区块时，可以直接从mainChain中检索。数据结构如下：

```
type BlockIndex struct {
    sync.RWMutex

    index     map[bc.Hash]*BlockNode
    mainChain []*BlockNode
}
```

当我们将一个区块保存到链上时，这个区块会上链，但是不一定会上主链。对通过AddNode方法添加的区块进行索引时，只需更新index中的区块索引信息，此时不会更新mainChain中保存的主链信息。代码如下：

```
func (bi *BlockIndex) AddNode(node *BlockNode) {
  bi.Lock()
  bi.index[node.Hash] = node
  bi.Unlock()
}
```

5.3.3 区块连接

完成区块上链的操作后，节点还需要完成区块的连接。连接工作主要是将区块中所有交易花费的UTXO（UTXO概念参见6.4节）的状态标记为已消费，并将交易从交易池中移除。代码示例如下：

```
protocol/block.go
func (c *Chain) connectBlock(block *types.Block) (err error) {
   bcBlock := types.MapBlock(block)
   if bcBlock.TransactionStatus, err =
c.store.GetTransactionStatus(&bcBlock.ID); err != nil {
      return err
   }
   utxoView := state.NewUtxoViewpoint()
   if err := c.store.GetTransactionsUtxo(utxoView,
bcBlock.Transactions); err != nil {
      return err
   }
   if err := utxoView.ApplyBlock(bcBlock,
bcBlock.TransactionStatus); err != nil {
      return err
   }
   node := c.index.GetNode(&bcBlock.ID)
   if err := c.setState(node, utxoView); err != nil {
      return err
   }
   for _, tx := range block.Transactions {
      c.txPool.RemoveTransaction(&tx.Tx.ID)
   }
   return nil
}
```

首先通过GetTransactionsUtxo方法找到区块中所有交易引用的UTXO，并将其保存到UtxoViewpoint中。UtxoViewpoint可以用来表示一组未花费交易输出的集合。ApplyBlock方法则用来更新UTXO的状态。对于交易输入引用的UTXO，如果满足以下两个条件，则将UTXO的状态更新为已消费：

· 交易状态为成功，并且UTXO的资产类型为BTM资产。

· 如果交易引用的UTXO是一个Coinbase交易的输出（参见6.3.2节），则该Coinbase交易所在的区块高度至少要比现在小100，即Coinbase交易的输出会被阻塞100个区块之后，才能正式解锁使用。

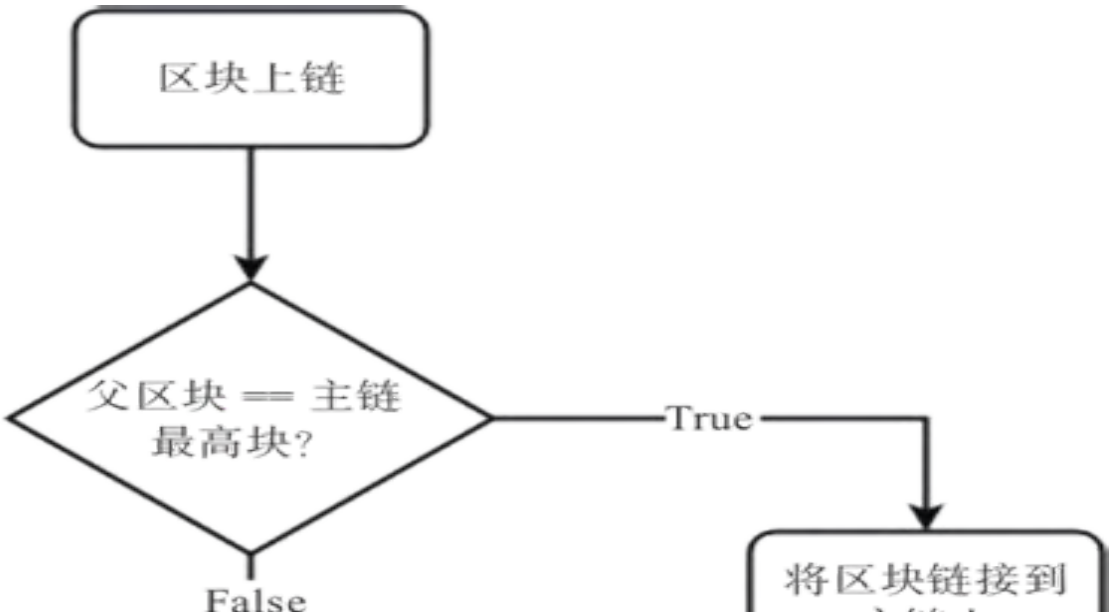
ApplyBlock方法完成UTXO状态更新之后，将区块中Coinbase交易产生的UTXO存储到LevelDB中的UTXO集合中。还需通过setState方法将这些已消费的UTXO更新到LevelDB中的UTXO集合中。

5.3.4 链重组

节点将接收到的区块连接到Chain链上。当新共识规则发布后，没有升级的节点会因不知道新共识规则，而生产不合法的区块，这便会产生临时性分叉。这种临时性分叉可以称为软分叉或侧链（这里的侧链是临时性产生的）。当主链与侧链冲突时，侧链也有可能成为主链，不过这取决于哪个链累积了最多的工作量证明。在processBlock方法中，如果添加区块的父区块不是主链上的尾节点，则将这个区块添加到侧链上，此时，需要判断主链与侧链的工作量哪个更多。如果侧链的工作量比主链的工作量还要多，则需要对区块链进行重组，将侧链变为主链。链重组流程如图5-1所示。

在判断区块链工作量时，需要计算区块链节点工作量总和，算法如下所示：

```
func CalcWork(bits uint64) *big.Int {
   difficultyNum := CompactToBig(bits)
   if difficultyNum.Sign() <= 0 {
      return big.NewInt(0)
   }
   denominator := new(big.Int).Add(difficultyNum, bigOne)
   return new(big.Int).Div(oneLsh256, denominator)
}
```



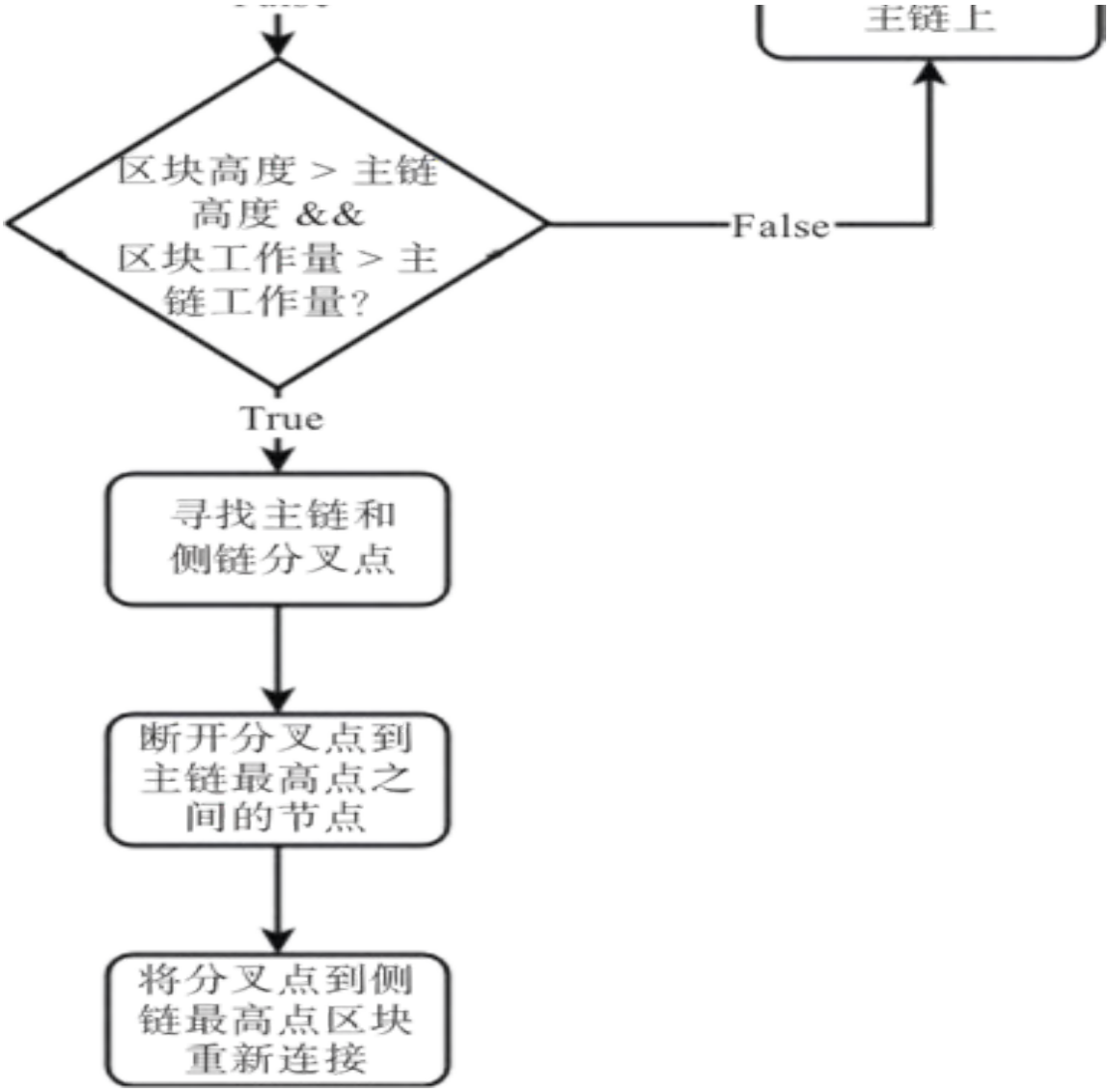


图5-1　链重组流程

节点工作量总和计算公式为：

SumPow = 2^256 / (difficultyNum+1)

如果这个新的侧链的累积工作量总和超过了旧的最佳链，则这个侧链需要成为新的主链。为了实现这一点，需要找到主链和侧链的分叉点，断开形成旧分叉的块与主链的链接，并将形成新链的块连接到从主链开始的主链ancenstor（链分叉的点）。该部分操作主要由reorganizeChain方法实现，步骤如下。

步骤1：区分出需要重组和删除的块。

通过calcReorganizeNodes方法查找到主链和侧链的分叉点，同时记录detachNodes节点和attachNodes节点。detachNodes节点是重组时需要断开与主链连接的节点，attachNodes节点是需要重新添加到主链的节点。代码示例如下：

```
protocol/block.go
func (c *Chain) calcReorganizeNodes(node *state.BlockNode)
([]*state.BlockNode, []*state.BlockNode) {
    var attachNodes []*state.BlockNode
    var detachNodes []*state.BlockNode

    attachNode := node
    for c.index.NodeByHeight(attachNode.Height) != attachNode {
        attachNodes = append([]*state.BlockNode{attachNode},
attachNodes...)
        attachNode = attachNode.Parent
    }

    detachNode := c.bestNode
    for detachNode != attachNode {
        detachNodes = append(detachNodes, detachNode)
        detachNode = detachNode.Parent
    }
    return attachNodes, detachNodes
}
```

calcReorganizeNodes方法首先从侧链最后节点依次向前遍历，通过NodeByHeight方法判断是否遍历到主链的一个节点attachnode（主链的这个节点attachnode就是此时的分叉点）。如果不是，则将遍历的节点加入attachNodes数组中。NodeByHeight方法会根据传入的高度从主链上查找节点并返回。如果传入的高度大于主链目前的高度，则NodeByHeight方法会返回一个空值。之后，该方法从最高节点开始依次遍历主链，将主链上attachnode父节点之后的所有区块加入到detachNodes数组中。

步骤2：处理detachNodes。

遍历detachNodes数组中保存的需要删除的节点，通过DetachTransaction方法将区块中所有交易的状态还原。DetachTransaction方法首先将交易中引用的UTXO状态变为未花费，这样之后的交易就又可以引用这些UTXO；然后将交易产生的OUTPUT状态变为已引用。这样可以防止用户在构建交易的输入中继续引用被重组区块中的交易。通过这两步，就可以将UTXO集的状态恢复到之前的状态。代码示例如下：

```
func (view *UtxoViewpoint) DetachTransaction(tx *bc.Tx,
statusFail bool) error {
    for _, prevout := range tx.SpentOutputIDs {
        spentOutput, err := tx.Output(prevout)
        if err != nil {
            return err
        }
        if statusFail && *spentOutput.Source.Value.AssetId !=
*consensus.BTMAssetID {
            continue
        }

        entry, ok := view.Entries[prevout]
        if ok && !entry.Spent {
            return errors.New("try to revert an unspent utxo")
        }
        if !ok {
            view.Entries[prevout] = storage.NewUtxoEntry(false,
0, false)
            continue
        }
        entry.UnspendOutput()
    }

    for _, id := range tx.TxHeader.ResultIds {
        output, err := tx.Output(*id)
        if err != nil {
            // error due to it's a retirement, utxo doesn't
care this output type so skip it
            continue
        }
        if statusFail && *output.Source.Value.AssetId !=
*consensus.BTMAssetID {
            continue
        }
```

```
            view.Entries[*id] = storage.NewUtxoEntry(false, 0,
true)
        }
        return nil
}
```

步骤3：处理attachNodes。

这步操作和上一步正好相反。在这一步中，节点遍历保存在attachNodes数组中需要变更到主链上的节点。通过ApplyBlock方法重新连接区块上链，ApplyTransaction方法将区块中所有交易的状态锁定。ApplyTransaction方法中，将交易中引用的UTXO状态更新为已消费。同时实例化新的UTXO对象。

步骤4：更新UTXO集合。

将上一步中新实例化的UTXO写入UTXO集合中，同时更新链的状态，将链的最高区块更新为原来侧链的最高区块。

5.3.5 主链的状态

当节点第一次启动时，节点会根据数据库中key为blockStore的内容判断是否初始化主链。初始化主链时，主链状态的高度是0，hash是创世区块。代码示例如下：

```
protocol/store.go
type BlockStoreState struct {
    Height uint64
    Hash   *bc.Hash
}
```

主链状态说明如下：

- Height：当前主链最高的高度。
- Hash：当前主链最高的哈希值。

主链状态是需要持久化到存储器中的，每次节点重启时会加载该值，存储与主链相关的key如下：

```
database/leveldb/store.go
blockStoreKey     = []byte("blockStore")
```

主链状态更新操作的触发时机：

- 当节点第一次启动时。
- 当新的区块添加到主链的末尾。
- 当重组主链时。

5.4 本章小结

本章详细介绍了区块和区块链的核心数据结构，并对区块和区块链的基本操作进行了深入的分析，包括：创世区块的生成、区块校验、区块难度计算、孤块管理、区块上链、区块重组等。通过本章的学习，我们已经对区块链的全貌有了详细了解。

第6章

内核层：交易

6.1 概述

交易是区块链系统中最重要的部分。根据区块链的设计原则，系统中其他部分都是为了确保交易可以生成、能在网络中传播和通过验证，并最终添加到交易总账簿。交易的本质是数据结构，这些数据结构中含有交易参与者价值转移的相关信息。区块链是一本全球复式记账总账簿，每个交易都是区块链上的一个公开记录。区块链的每笔交易都需要一定的费用，用于支付交易执行所需要的计算开销。计算开销一般使用Gas作为基本的计价单位。通过GasPrice与其他货币进行换算。

本章的主要内容如下：

- 交易中的核心数据结构。
- BUTXO模型。
- 交易的流程。
- 交易脚本及交易验证原理。
- 交易费和交易池。
- 默克尔树的原理及应用。

6.2　交易的概念

6.2.1　现实生活中的交易

在现实生活中，我们要完成一笔交易需要经过第三方授信机构（如银行、支付宝、微信等）来完成。如图6-1所示。

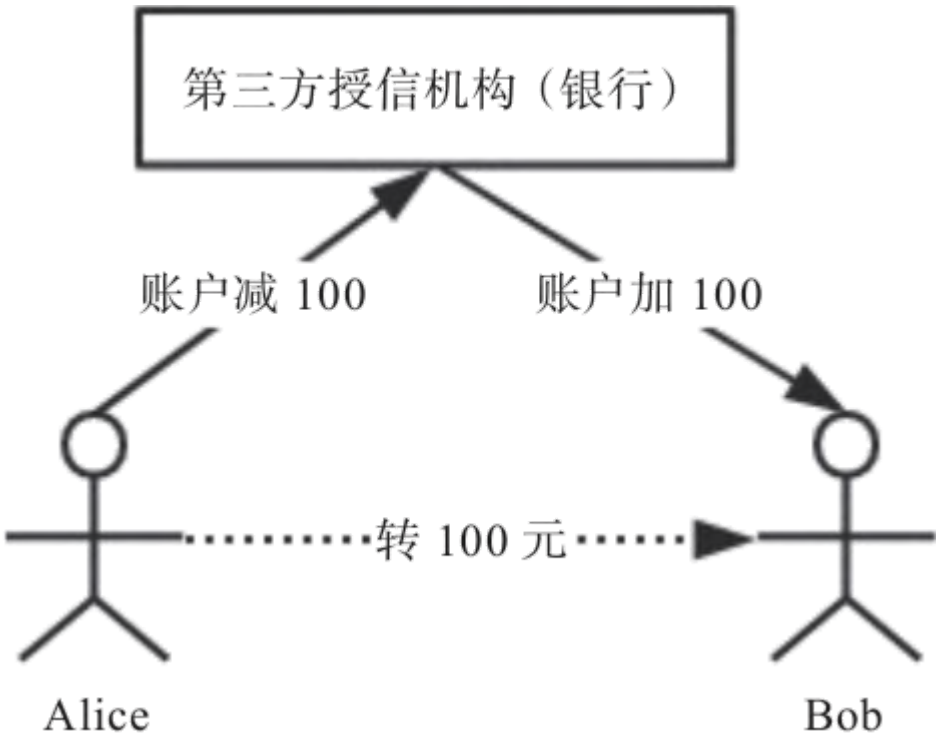


图6-1　现实生活中的交易

图6-1中Alice向Bob转账100元，在转账过程中银行完成了这笔交易。在这笔交易中银行将Alice账户扣除100元，向Bob账户添加100元。要完成上面这笔交易，需具备以下几个必要条件：

· 需要一个中心化的授信机构（如银行、支付宝、微信）。

· Alice和Bob都需要有这个授信机构（银行）的账号，双方必须信任。

· Alice的账号里面要有足够支付这笔交易的钱。

有了上面的必要条件，Alice和Bob之间就可以交易了，而银行负责记录这个交易（也就是记账）。

6.2.2 虚拟世界中的交易

虚拟世界是计算技术与通信技术构筑起来的信息世界，在虚拟世界中，我们不能随意信任一个中心化的机构，这样是不安全的。要想在虚拟世界中安全交易，可以使用区块链这样一个去中心化的网络。区块链类似于传统的银行的角色，但不同于银行这种中心化的机构，区块链网络是一个P2P网络，由分布在全球各个地方的节点组成，每个节点都保存了所有的交易数据，这样就保证了交易的安全，如图6-2所示。

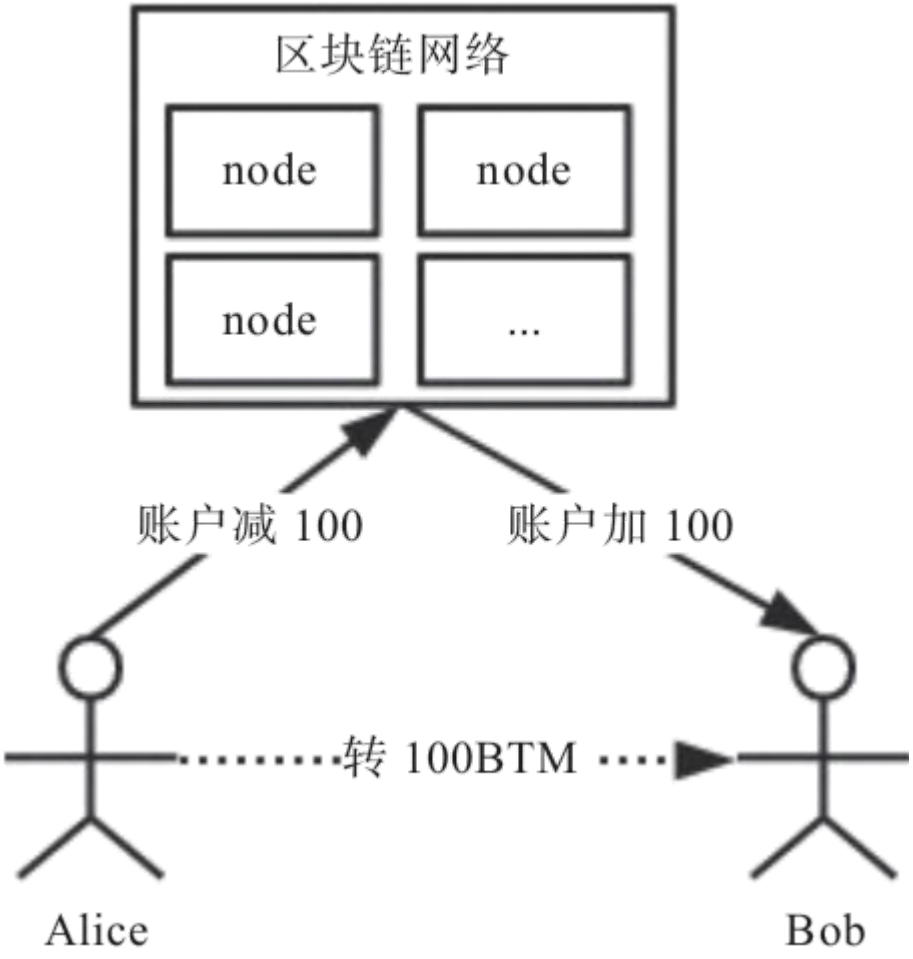


图6-2 虚拟世界中的交易

在区块链网络中，Alice和Bob转账都需要通过分布式的区块链网络来完成。Alice和Bob都有各自的账户。在区块链网络中的账户体系是一对公/私钥（非对称加密），公钥是公开的，相当于银行账户地

址，任何人都可以查看；私钥相当于账户密码，用户只有拥有私钥才能花费交易。在区块链网络中完成一笔交易和现实生活中的交易有些差异：

· 在区块链网络中，密钥丢失或被盗，意味着账号中所有金额丢失。账号无法挂失。

· 所有转账记录是公开的，任何人都有权利查询每个账户产生的交易记录。

6.3 核心数据结构

交易是区块链中最重要的部分。区块链中交易以外的功能都是为了确保交易可以正确地生成、验证、传播以及最终正确打包到区块中。

交易是区块链上一个基本的操作事务，本质是一种含有输入值和输出值的数据结构。这种数据结构记录了交易的参与者和交易的价值转移信息。

比原链中存在两种类型的交易：普通类型的交易和Coinbase类型的交易。普通类型的交易是指由用户发起的转账、发行资产等操作；Coinbase类型的交易是一种没有输入的交易，用来奖励矿工为挖矿所做出的贡献。下面介绍这两种类型的交易数据结构。

6.3.1 普通交易核心数据结构

我们通过比原链区块浏览器（http://www.btmscan.com）查看高度为105449的区块信息，该区块包含两个交易信息，第一个是Coinbase类型的交易，将会在6.3.2节中讲解。第二个交易信息如图6-3所示。

图6-3 高度为105449区块包含的第二个交易信息

由图6-3中可以看出，该交易是一个转账交易，由地址bm1q5wta9jhq8svxvu3pedct29qcvqgq0wrkwh44wa转出31798个BTM。其中，9999.995 BTM转移到地址bm1qvens3huudsjhqysxak5yeudp5evmetly9zna69，1798 BTM转移到地址bm1qvr68qcfjw6lu7euqfr9gky6g9rxgjcgrsjkd8d，……，0.005 BTM作为这笔交易的手续费。这是我们在区块浏览器中看到交易情况，而背后的交易数据结构与我们看到的可能有很大区别。高度为105449的交易JSON信息如下：

```
{

"id":"7d769fb564e5dc4947dab944c33e1b3ec522b8ce1ebdaf365d00edf576
fbe7c1",
    "inputs":[
        {

"address":"bm1q5wta9jhq8svxvu3pedct29qcvqgq0wrkwh44wa",
            "amount":3179800000000,
            "asset_definition":{

            },

"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffff",

"control_program":"0014a397d2cae03c18667221cb70b51418601007b876"
,

"input_id":"cb299ef868d4f7de93c2dfb23b955d283fc769179896f4dcfa0b
34497605e287",

"spent_output_id":"be08e806d5fa7ba234a586fb8ed68a94c0ff79f657a1c
9064db124336acbfa55",
            "type":"spend",
            "witness_arguments":[

"c34b973442fa26ed4d0b6fbdfa0f02de7570139feb7db3111a1fb575fc04971
3fa0b62ca4831e9061d3b833bdab748cf8a604466ae9b77a7a81c7063d17c9b0
1",

"7816e3ff1ae75d18c2f06cadb0891a01b88b5ac328bc3be9b5d9fbbc9c12101
0"
            ]
        }
    ],

"mux_id":"99955b5d3fb339227585279b2b1c3a17a05bd624aa68feedda2892
d26280858a",
    "outputs":[
        {

"address":"bm1qvens3huudsjhqysxak5yeudp5evmetly9zna69",
            "amount":999999500000,
            "asset_definition":{

            },
```

```
"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffff",

"control_program":"0014666708df9c6c25701206eda84cf1a1a659bcafe4"
,

"id":"caa1121750506c0c130d8036887a71c34e18e7ab8a9f8350afcd12e166
d0cb3b",
            "position":0,
            "type":"control"
        },
        {

"address":"bm1qvr68qcfjw6lu7euqfr9gky6g9rxgjcgrsjkd8d",
            "amount":179800000000,
            "asset_definition":{

            },

"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffff",

"control_program":"001460f470613276bfcf678048ca8b134828cc896103"
,

"id":"0b91d5f53a5989e110b82f730685df7d73c04f49593c0feb91b0600c90
f2b662",
            "position":1,
            "type":"control"
        },
        {

"address":"bm1q9e4te09805zd6wkx9k455kz4dg9sde5y7fxtdc",
            "amount":1000000000000,
            "asset_definition":{

            },

"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffff",

"control_program":"00142e6abcbca77d04dd3ac62dab4a58556a0b06e684"
,

"id":"0e6ea801dfd5ac81789250404b603fa3d2677a9c0d8aac455013ea9456
682edd",
            "position":2,
            "type":"control"
```

```
        },
        {

"address":"bm1q3frvfnpa8l6lckuczexj6rw93gfzvh9n4rr9tn",
            "amount":1000000000000,
            "asset_definition":{

            },

"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",

"control_program":"00148a46c4cc3d3ff5fc5b98164d2d0dc58a12265cb3",

"id":"058c30247fa93cc30690ff799aa5a34f11df94ee275b204509f4c4fd2d896125",
            "position":3,
            "type":"control"
        }
    ],
    "size":464,
    "status_fail":false,
    "time_range":0,
    "version":1
}
```

我们使用命令行工具bytomcli通过get-block接口获取第105449区块的信息，可以看到该区块包含的交易哈希值为7d769fb564e5dc4947dab944c33e1b3ec522b8ce1ebdaf365d00edf576fbe7c1。由上面的交易信息可知，一笔交易至少要包含一个inputs和一个outpusts，这是UTXO模型所规定的；还需包含一些标识交易的状态、版本、字节大小和有效时间的字段。

Tx数据结构表示一笔交易，Tx数据结构及其内部包含的其他数据结构如下所示：

```
protocol/bc/types/transaction.go
type Tx struct {
    TxData
    *bc.Tx `json:"-"`
}
```

```
type TxData struct {
    Version uint64
    SerializedSize uint64
    TimeRange uint64
    Inputs []*TxInput
    Outputs []*TxOutput
}
```

交易数据结构说明见表6-1。

表6-1　交易数据结构

字段	字节数	说明
Version	8	版本号
SerializedSize	8	交易总字节数
TimeRange	8	交易过期时间
Input	不定长	交易输入
Output	不定长	交易输出

1. 交易的输出Output

每一笔普通交易都会创造输出，并被记录在BUTXO集合中。除了一些特殊的交易（比如Retire交易），交易的输出都会产生一定数量的可用于支付的BTM。这些BTM可以被接收者所使用。

交易的输出主要包含两部分重要信息：一是交易资产的信息，主要包括资产类型和资产数量；二是确定花费输出的所需条件，这个条件也称为锁定脚本。用户在其他交易中使用该交易产生的BUTXO，必须提供能够解锁该锁定脚本的合法参数。交易输出AnnotatedOutput的数据结构如下：

```
blockchain/query/annotated.go
type AnnotatedOutput struct {
```

```
    Type             string
    OutputID         bc.Hash
    TransactionID    *bc.Hash
    Position         int
    AssetID          bc.AssetID
    AssetAlias       string
    AssetDefinition *json.RawMessage
    Amount           uint64
    AccountID        string
    AccountAlias     string
    ControlProgram   chainjson.HexBytes
    Address          string
}
```

交易输出结构说明见表6-2。

表6-2　交易输出结构

字段	字节数	说明
Type	变长	输出类型，比原链中支持两种类型的输出——control 和 retire。retrie 是销毁资产时产生的输出。除 retire 外，其他的输出都是 control 类型
OutputID	32	输出 ID，由交易的输出哈希后得到
TransactionID	32	交易 ID
Position	8	下标，标识该输出是整个交易中的第几个输出
AssetID	32	输出资产 ID，一个输出只能包含一种资产类型
AssetAlias	变长	输出资产别名
AssetDefinition	变长	输出资产定义信息
Amount	8	输出资产的数量
AccountID	变产	输出账户 ID
AccountAlias	变长	输出账户别名
ControlProgram	变产	输出的锁定脚本，用户在使用该输出时必须提供正确的解锁脚本，才能花费该输出
Address	变长	输出所属用户的地址

AnnotatedOutput其实是一个可以让我们更容易理解的、上层的交易输出结构。可以说，我们通过API接口获取的交易输出数据都是AnnotatedOutput结构。与AnnotatedOutput相比，区块链底层的交易输出结构更加抽象，它剔除了很多冗余信息，例如账户，资产定义等。

区块链底层交易输出数据结构在protocol/bc/type/txoutput.go文件中定义。大部分字段含义相同，读者可自行了解。

2. 交易的输入Input

交易的输入会将BUTXO标记为已消费，并通过解锁脚本提供所有权证明。用户在构建交易时，需要从钱包中寻找足够金额的BUTXO执行支付请求。有时用户使用一个BUTXO就足够支付交易请求。每一个交易输入都会指向一个BUXTO，并提供相应的解锁脚本解锁它。

每一笔普通交易的输出都会引用一个输入，AnnotatedInput数据结构如下：

```
blockchain/query/annotated.go
type AnnotatedInput struct {
    Type            string
    AssetID         bc.AssetID
    AssetAlias      string
    AssetDefinition *json.RawMessage
    Amount          uint64
    IssuanceProgram chainjson.HexBytes
    ControlProgram  chainjson.HexBytes
    Address         string
    SpentOutputID   *bc.Hash
    AccountID       string
    AccountAlias    string
    Arbitrary       chainjson.HexBytes
    InputID         bc.Hash
    WitnessArguments []chainjson.HexBytes
`json:"witness_arguments"`
}
```

交易输入的结构说明见表6-3。

表6-3　交易输入结构

字段	字节数	说明
Type	变长	输入的类型，目前共有三种类型：issue，spend，coinbase
AssetID	32	输入资产的 ID
AssetAlias	变长	输入资产的别名
AssetDefinition	变长	输入资产的定义
Amount	8	输入资产的数量
IssuanceProgram	变长	如果是发行资产（issue）的交易，交易输入的 IssuanceProgram 字段会填充资产的锁定脚本
ControlProgram	变长	除发行资产的交易外，其他所有交易输入引用的 BUTXO 的锁定脚本都会填充到 ControlProgram

（续）

字段	字节数	说明
Address	变长	交易输入账户的地址
SpentOutputID	32	输入引用的 BUTXO 所属输出的 ID
AccountID	变长	交易输入账户的 ID
AccountAlias	变长	交易输入账户别名
Arbitrary	变长	Coinbase 类型交易的输入一般都会附带一些标注信息，填充到 Arbitrary 字段
InputID	32	输入 ID
WitnessArguments	变长	交易见证参数，即交易的解锁脚本

与AnnotatedOutput结构相同，AnnotatedInput也是一个属于上层的交易结构。交易输入的核心数据结构在protocol/bc/type/txinput.go中定义。

6.3.2 Coinbase交易核心数据结构

Coinbase交易也称为“创币交易”“币基交易”。Coinbase交易是每一个区块的第一笔交易，这笔交易用来奖励矿工挖矿做出的贡献。也许很多人都不太理解Coinbase交易为什么又称为“创币交易”，其实这和Coinbase交易对于整个区块链的作用有很大的关系。我们知道任何一种货币都是需要发行的，例如美元和人民币，它们分别是美联储和央行发行的纸币。而像区块链上的数字货币，肯定不能依靠像美联储和央行这种机构印刷发行。区块链上的货币是依靠Coinbase交易发行的。区块链规定矿工每挖出一个区块，系统会奖励矿工一笔“矿工费”。矿工费通过Coinbase交易支付给矿工，Coinbase交易不会引用区块链中已经存在的BUTXO，因此每一笔Coinbase交易都会创建新的币，因此这种交易也称为“创币交易”。

如果从交易的输入和输出角度看，Coinbase是一种特殊的交易类型，这种交易是没有输入的，也不消耗BUTXO。交易的输出通常是指向矿工的钱包地址。

从区块浏览器中查看第105449区块的第一笔交易，这笔交易很明显被标识为挖矿奖励。在其他的一些区块浏览器中，例如“blockmeta.com”，则会将该交易标识为Coinbase交易，并且表明这笔交易没有Inputs，会产生新的币。Coinbase交易信息如图6-4所示。

图6-4 Coinbase交易信息

Coinbase交易的数据结构和普通交易的数据结构在很大一部分上是相同的。使用比原链命令行工具bytomcli可以获取第105449区块的Coinbase交易的信息，示例如下：

```
{

"id":"360754aeed46658be88f5071af4bcae961e334f99018db1325e283aecb
e9360f",
    "inputs":[
        {
            "amount":0,
            "arbitrary":"00313035343439",
            "asset_definition":{

            },

"asset_id":"0000000000000000000000000000000000000000000000000000
0000000000",

"input_id":"7cce7088ae01cd610fd7474f526292b760d38ad5e40ed82d3777
7f27882a2dc0",
            "type":"coinbase",
            "witness_arguments":null
        }
    ],

"mux_id":"4c8f95edfb2fafbbf2209291e63e024306466d39d6cc0d7a59c77f
c067280a5e",
    "outputs":[
        {

"address":"bm1q3yt265592czgh96r0uz63ta8fq40uzu5a8c2h0",
            "amount":41250500000,
            "asset_definition":{

            },

"asset_id":"ffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffff",

"control_program":"00148916ad528556048b97437f05a8afa7482afe0b94"
,

"id":"500444582ea8dd42f7252380f5bd7b4f316762ae08180124ad4e056513
76e567",
            "position":0,
```

```
            "type":"control"
        }
    ],
    "size":82,
    "status_fail":false,
    "time_range":0,
    "version":1
}
```

Coinbase交易的数据结构和普通交易的差别不大，都是由Inputs、Outputs、交易版本、交易字节数大小、交易状态和交易有效期组成。但是，如果仔细观察，Coinbase交易和普通交易在Inputs上还是有很大差别的。

· Amount数值不同：Coinbase交易Inputs的Amount永远都是0，而普通交易中是根据用户实际转账金额填充的。

· asset_id资产类型不同：Coinbase交易Inputs的asset_id永远都是“000000...000000”，普通交易的资产类型会根据用户实际交易的资产类型填充。

· Type交易类型不同：Coinbase交易Inputs的类型被填充为“coinbase”，标识这笔交易是Coinbase交易。普通交易的Inputs的类型会根据实际情况填充。

· Coinbase交易不包含解锁脚本，而是由arbitrary字段代替，该字段可以由挖矿节点自由填充，长度最小2字节，最大100字节。除了开始的几个字节外，矿工可以任意使用Coinbase的其他部分，随意填充任何数据。

在比原链中，Coinbase与普通交易的差别主要体现在TxInput结构，目前共支持三种不同的TypeInput：IssuanceInput、SpendInput和CoinbaseInput：

· IssuanceInput交易是一种发行资产的交易类型。

· SpendInput交易是一种花费UTXO的交易。

· CoinbaseInput是创币交易。

这几种交易的数据结构分别如下所示。

IssuanceInput交易类型数据结构如下：

```
protocol/bc/types/issuance.go
type IssuanceInput struct {
    Nonce  []byte
    Amount uint64

    AssetDefinition []byte
    VMVersion       uint64
    IssuanceProgram []byte
    Arguments       [][]byte
}
```

IssuanceInput交易类型说明见表6-4。

表6-4　IssuanceInput交易类型

字段	字节数	说明
Nonce	8	8字节长的随机数
Amount	8	发行资产的数量
AssetDefinition	变长	资产定义
VMVersion	8	验证资产使用的虚拟机版本
IssuanceProgram	变长	资产的锁定脚本，用户只有提供正确的解锁脚本才能花费资产
Arguments	变长	用来存储见证参数

SpendInput交易类型数据结构如下：

```
protocol/bc/types/spend.go
type SpendInput struct {
    SpendCommitmentSuffix []byte
    Arguments             [][]byte
    SpendCommitment
}
protocol/bc/types/spend_commitment.go
type SpendCommitment struct {
    bc.AssetAmount
    SourceID       bc.Hash
    SourcePosition uint64
    VMVersion      uint64
    ControlProgram []byte
}
protocol/bc/bc.pb.go
type AssetAmount struct {
    AssetId *AssetID
    Amount  uint64
}
```

SpendInput交易类型说明见表6-5。

表6-5　SpendInput交易类型

字段	字节数	说明
SpendCommitmentSuffix	变长	BUTXO承诺后缀，保留字段，目前未使用
Arguments	变长	见证参数
AssetId	32	资产ID
Amount	8	资产数量
SourceID	32	该输入所引用的BUTXO所属输出的ID
SourcePosition	变长	该输入所引用的BUTXO所属输出位置
VMVersion	8	虚拟机版本
ControlProgram	变长	所花费的BUTXO的锁定脚本

CoinbaseInput交易类型数据结构如下：

```
protocol/bc/types/coinbase.go
type CoinbaseInput struct {
    Arbitrary []byte
}
```

其中：Arbitray为变长字节数，用于标注信息。

Coinbase交易的Inputs几乎不含有其他任何资产信息，只含有一个Arbitrary字段，由矿工自己随意填充，也可以为空。

6.3.3 交易Action数据结构

在比原链中，每一笔交易都是由多个交易动作组成的，这些交易动作称为交易Aciton。交易Action主要分为两种：input action和output action。input action代表交易的输入动作，与交易的输入对应；output action代表交易的输出动作，与交易的输出对应。两种Action又包含多种子类型。

input aciton类型包括如下子类型：

- issue：发行资产。
- spend_account：以账户的模式花费UTXO。
- spend_account_unspent_output：直接花费指定的UTXO。

output action类型包括如下子类型：

- control_address：接收方式为地址模式。
- control_program：接收方式为（program）合约模式。
- retire：销毁资产。

1. issue交易类型

issueAction是发行资产时必须使用的一个输入动作。在issueAction中用户需要指定发行资产的种类和发行资产的数量。issueAction的数据结构如下：

```
asset/builder.go
type issueAction struct {
    assets *Registry
    bc.AssetAmount
    Arguments []txbuilder.ContractArgument
}
protocol/bc/bc.pb.go
type AssetAmount struct {
    AssetId *AssetID
```

```
    Amount   uint64
}
```

issue结构说明见表6-6。

表6-6　issue结构

字段	字节数	说明
assets	变长	用于资产管理
AssetId	32	资产ID，调用/create-asset创建新的资产并返回资产的唯一标识ID
Amount	8	发行资产的数量
Arguments	变长	智能合约参数

2. spend_account交易类型

spendAction是转账交易时使用的一个输入动作。比原链提供了两种转账时使用的输入动作，一个是spendAction，另一个是spendUTXOAction。

spendAction是以账户的模式花费BUTXO，即在构建交易的时候，系统会根据用户的账户ID，从BUTXO集合中查找足够数量的BUTXO。使用spendAction转账时，用户需要提供账户ID、资产ID和资产数量。spendAction数据结构如下：

```
account/builder.go
type spendAction struct {
    accounts *Manager
    bc.AssetAmount
    AccountID      string
    UseUnconfirmed bool
}
```

spend_account结构说明见表6-7。

表6-7　spend_account结构

字段	字节数	说明
accounts	变长	用于账户管理
AssetAmount	变长	需要花费的资产ID和资产数量
AccountID	变长	需要花费的资产的账户ID
UseUnconfirmed	1	是否使用未确认交易产生的UTXO

3. spend_account_unspent_output交易类型

spendUTXOAction也是转账交易时使用的一个输入动作。spendUTXOAction是在交易时用户直接花费指定的BUTXO，即转账时用户指定要花费的BUTXO ID。spendUTXOAction数据结构如下：

```
account/builder.go
type spendUTXOAction struct {
    accounts       *Manager
    OutputID       *bc.Hash
    UseUnconfirmed bool
    Arguments      []txbuilder.ContractArgument
}
```

spend_account_unspent_output结构说明见表6-8。

表6-8　spend_account_unspent_output结构

字段	字节数	说明
accounts	变长	用于账户管理
OutputID	32	需要花费的 BUTXO ID
UseUnconfirmed	1	交易是否被确认
Arguments	变长	智能合约参数

4. control_address交易类型

controlAddressAction是一种交易输出动作，用于指定交易输出的接收方式为地址模式。平常我们使用的非智能合约转账交易的输出都为controlAddressAction，非智能合约的转账在转账时都需要提供转账接收方的用户地址。controlAddressAction的数据结构如下：

```
type controlAddressAction struct {
    bc.AssetAmount
    Address string `json:"address"`
}
```

control_address结构说明见表6-9。

表6-9　control_address结构

字段	字节数	说明
AssetAmount	40	接收的资产 ID 和资产数目
Address	变长	接收资产的地址

5. control_programe交易类型

controlProgramAction是交易输出动作中的一种。controlProgramAction指定交易输出的接收方式为合约模式，主要是

为智能合约提供的一种交易输出方式。controlProgramAction的数据结构如下：

```
blockchain/txbuilder/actions.go
type controlProgramAction struct {
    bc.AssetAmount
    Program json.HexBytes
}
```

control_address actiion结构说明见表6-10。

表6-10　control_address actiion结构

字段	字节数	说明
AssetAmount	40	接收的资产ID和资产数量
Program	变长	接收资产的合约脚本

6.retire交易类型

retireAction也是交易输出的一种Action。retireAction是销毁资产必须使用的一种输出动作。用户使用retireAction销毁资产时需要指定销毁资产的数量和资产ID。retireAction数据结构如下：

```
blockchain/txbuilder/actions.go
type retireAction struct {
    bc.AssetAmount
    Arbitrary json.HexBytes
}
```

retire结构说明见表6-11。

表6-11　retire结构

字段	字节数	说明
AssetAmount	40	需要销毁的资产 ID 和资产数量
Arbitrary	变长	标注信息

6.3.4 MUX交易类型

MUX结构是一种多资产链所特有的数据结构。多资产链与比特币这种单资产链相比具有一个显著特征——交易的输入和输出可以包含多种类型的资产。

多种类型的资产在交易中会被拆分组合，构成多种多样的对应关系。例如，一个交易中可能有两个交易的输入分别属于两个不同的资产，而每一个输入又分别对应两个输出。另外一种情况是一个交易可能有同一种资产的两个输入，两个输入又分别对应n个交易输出。为了在交易验证时能够更加简单地抽象出每种资产的支出与花费，在多资产链中提供了一种MUX结构。

我们可以简单将MUX结构理解为交易池。MUX把所有的输入根据不同的资产类型进行归集，再与所有的输出相比较，防止输出和MUX已经归集的输入不匹配，而在逐个验证输出时，不需要根据不同资产去累加输入，直接根据MUX已经归集的输入即可，这样做方便了多资产多输入输出的验证。MUX结构可以防止溢出，如果在输入中恶意填入一些越界的值，比如填充了一笔负值或者超过上限的输入，会被MUX监测出来，从而防止溢出攻击。

MUX数据结构如下：

```
protocol/bc/bc.pb.go
type Mux struct {
    Sources              []*ValueSource
    Program              *Program
    WitnessDestinations []*ValueDestination
    WitnessArguments     [][]byte
}
```

MUX数据结构说明见表6-12。

表6-12 MUX数据结构

字段	字节数	说明
Sources	变长	存储所有 MUX 的输入 Entry，输入 Entry 由交易的输入（IssuanceInput、SpendInput、CoinbaseInput）生成
Program	变长	BVM 验证程序，MUX 结构的验证程序是固定的 bc.Program{VmVersion: 1, Code: []byte{byte(vm.OP_TRUE)}}
WitnessDestinations	变长	存储所有 MUX 的输出 Entry，输出 Entry 由交易的输出（TxOutput）生成
WitnessArguments	变长	见证参数，用于解锁 output 的锁定脚本

这里我们对MUX结构做简单的说明，以方便大家理解后面的交易验证。

WitnessDestination和ValueSource是在bc中定义的，并在交易验证中使用的两个专用名词。ValueSource可以理解为资产源，用来绑定资产输入。例如MUX结构中的资产都是由Issuance Entry、Spend Entry和Coinbase Entry类型导入的，所以MUX结构的ValueSource就应该指向这几种类型。WitnessDestination可以理解为资产输出的方向，它与ValueSource一一对应。

6.4 BUTXO模型

UTXO（unspent transaction outputs，未花费的交易输出）是比特币中最基础的构建单元，也是比特币交易生成及验证的一个核心概念。可以说，如果没有UTXO，就没有比特币的交易。UTXO使比特币的交易构成了一组链式结构，所有合法的比特币交易都可以追溯到向前一个或多个交易的输出，这些链条的源头都是挖矿奖励，末尾则是当前未花费的交易输出。所有的未花费的输出即是整个比特币网络的UTXO。

比原链上的UTXO模型是对比特币中传统的UTXO模型的扩展。这种扩展后的UTXO模型称为BUTXO模型。BUTXO既兼容了比特币的UTXO模型，又实现了高并发和可控匿名。BUTXO模型可以兼容多种类型的比特和原子资产，可以做多元资产管理。如果想深入理解BUTXO模型，就需要先理解比特币的UTXO模型。UTXO模型如图6-5所示。

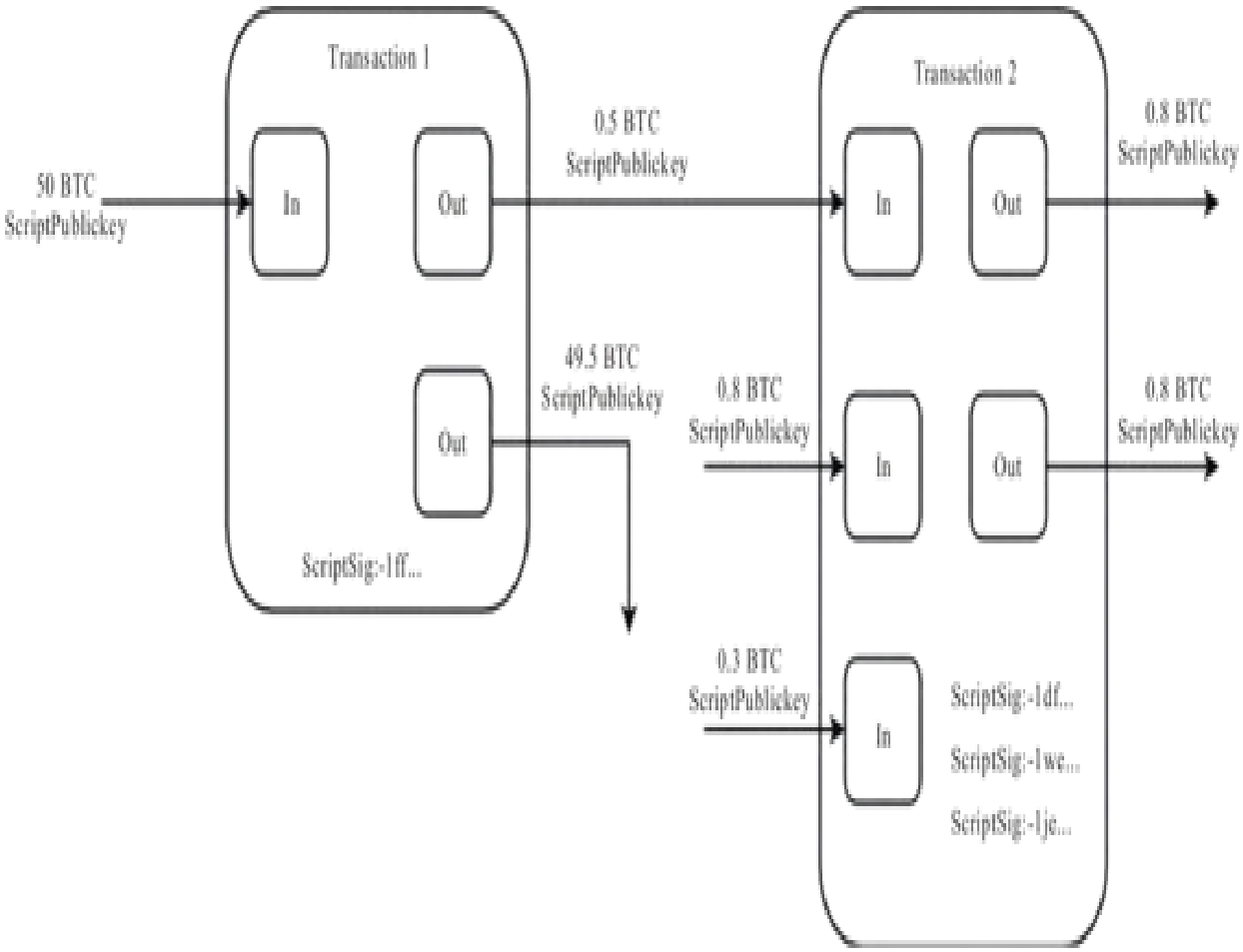


图6-5　UTXO模型

比特币中规定，每一笔新的交易的输入必须是某笔交易未花费的输出，每一笔输入同时也需要上一笔输出所对应的私钥进行签名，整个网络上的节点通过UTXO及签名算法来验证新交易的合法性。这样，节点不需要追溯历史就可以验证新交易的合法性。

在性能方面，由于UTXO是独立的数据记录，并且UTXO模型是一种无状态的模型，因此可以更容易地进行并发处理，以提升区块链交易验证的速度。UTXO模型和现实的货币世界更接近，也更容易理解。但是比特币的UTXO模型的缺陷也十分明显，最显著的一点是，比特币的UTXO模型是非图灵完备的，无法实现一些比较复杂的逻辑，即在比特币上实现智能合约是一件特别困难的事情。因此以太坊为了实现图灵完备的智能合约，选择放弃使用UTXO账户模型，而选用基于Account的账户模型。虽然以太坊的账户模型在实现智能合约上极为方便，但是性能欠佳一直是以太坊无法继续前进的障碍。

6.4.1 BUTXO模型原理

比原链通过扩展比特币的UTXO模型，实现了一种图灵完备的UTXO模型，这种模型既保持了传统UTXO模型在性能上的优势，也改进了传统UTXO模型无法拥有图灵完备性合约的缺陷。

比原链的BUTXO模型与UTXO模型相比，通过添加资产号字段实现了不同资产可以在同一笔交易中转换，如图6-6所示。

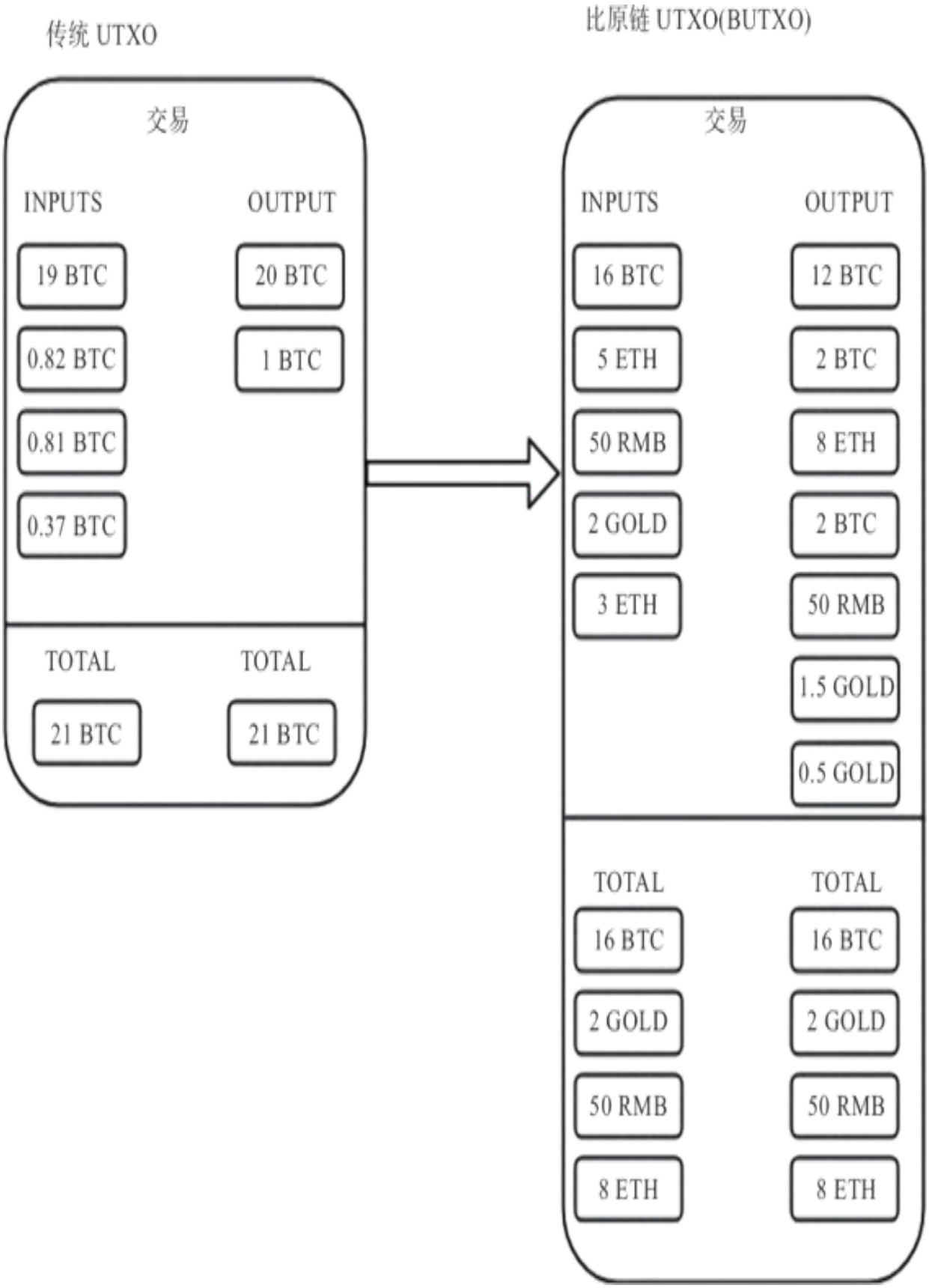


图6-6　UTXO模型与BUTXO模型对比

同时，比原链为了更方便地管理各种资产和UTXO，还引入了以太坊世界状态的概念。

比原链为每一种资产都维护一个全局状态，该全局状态具有快速可查找、不可更改、简单易提供证明等特性。每种资产的所有未花费的交易输出都会在全局BUTXO数据库中保存一个索引计数。每种资产的每一个BUTXO的计数不能超过1，即一个BUTXO只能被一个交易所引用。在并行计算时，可以保持一个output最多只能被一个BVM实例所引用。BVM是比原链实现的一种智能合约的虚拟机模型，其运行机制如图6-7所示。当一笔交易需要验证的时候，都会实例化一个BVM实例。在BVM实例中，各种资产只有在保持全局状态有效和一致的情况下才会更新资产的状态信息。比原链的BVM是可以并行运行的，并且是完全隔离的，每个BVM中智能合约的运行完全不受外界环境的影响（我们将在第8章详细讲解BVM运行原理及机制）。

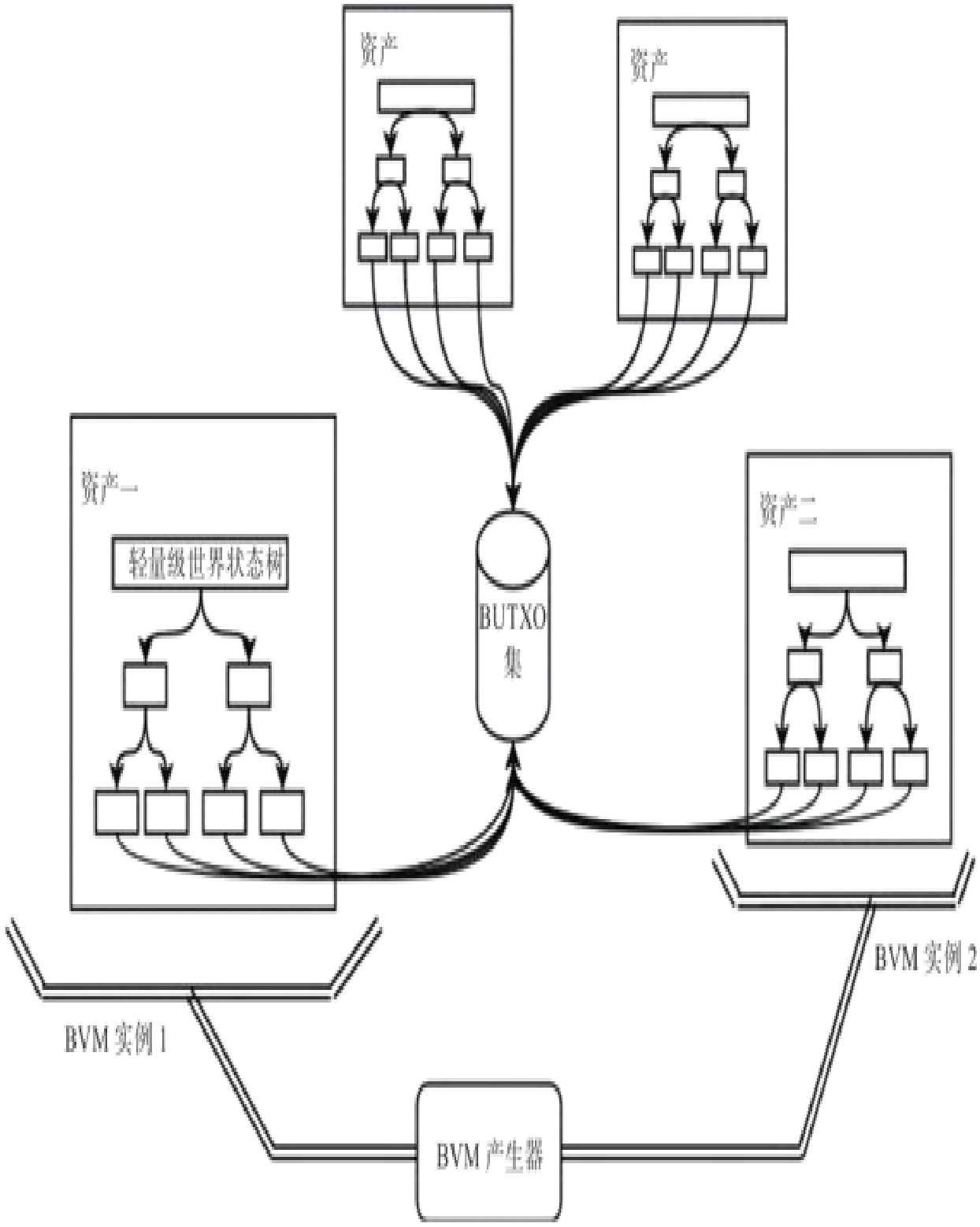


图6-7　BVM运行机制

6.4.2 MUX结构

比原链还在UTXO模型的基础上加入了MUX结构，从而能够在一笔交易中支持多输入和多输出。MUX结构可以理解为一个交易池，如图6-8所示，将一笔交易中的输入放入到MUX中，然后分配成不同的输出。

中间区域表示MUX结构，MUX结构将所有的资产根据不同的种类汇总起来，并根据不同的输出进行分配。在MUX中，一个资产类型对应一个或者多个输入，同时也可以对应一个或者多个不同的输出。MUX结构最重要的好处就是将原本多对多的关系，简化为一次“多对一”和一次“一对多”的关系，从而简化多资产的验证逻辑。因为比原链的多资产是多输入和多输出的，很可能一笔输出是指向之前的多笔输入的，所以比原链的UTXO的结构也不同，简单来说BUTXO可以使用如下公式表示：

BUTXO=HASH（AssertID，Amount，Address，MUX）

我们看到BUTXO架构中包括资产ID、资产数量、地址和MUX结构。

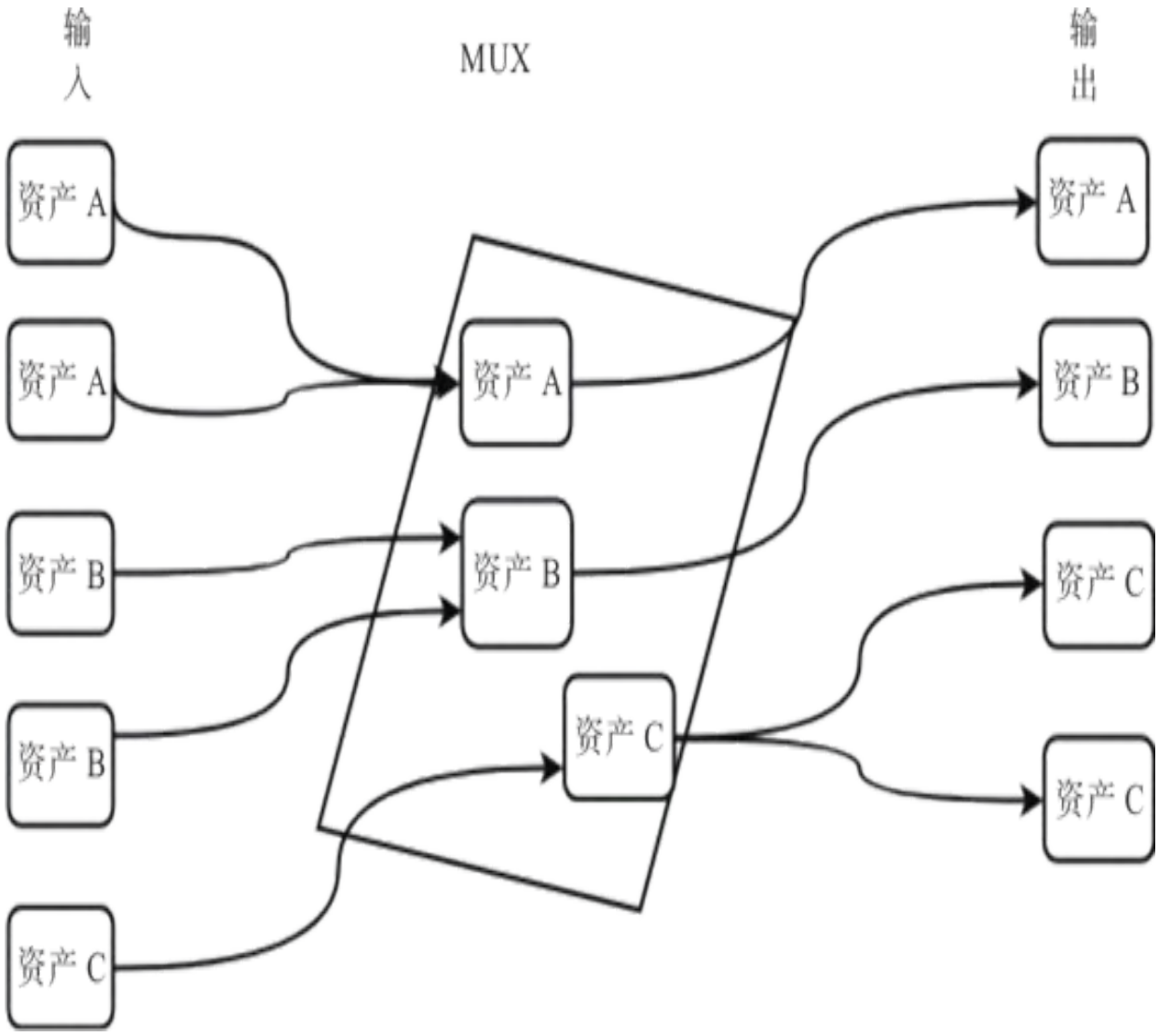


图6-8　MUX结构

6.5 交易的流程

在本节中，我们将详细介绍使用比原链客户端进行链上交易的流程。为了使本节更加贴近生活，以便将区块链交易讲解得更加通俗易懂。我们以一个区块链上的转账交易为例，介绍链上交易的每个流程。

在区块链上完成一笔交易通常包括以下几步：

1）构建交易并估算手续费。

2）签名交易。

3）提交交易。

Alice向Bob购买了一本《GO语言公链开发实战》，这本书价值10元。Bob允许Alice支付数字货币以代替人民币。按照交易当天BTM与RMB的汇率，Alice需要向Bob支付5BTM。Alice如果想成功地向Bob支付5BTM，则Alice需要构建、签名和提交交易。通过钱包添加交易的金额和Bob的地址，点击确认，交易就完成了。

6.5.1 构建交易

首先Alice需要通过API接口/build-transaction，构建一个交易，构建交易的请求参数是一个JSON字符串，该字符串中包含交易的动作、交易有效时间等信息。下面我们构造一个BTM资产的转账交易。交易请求参数如下：

```
{
    "actions":[
        {
            "amount":500000000,
            "type":"spend_account",
            "account_alias":"alice",
            "account_id":"",
            "asset_alias":"BTM",
            "asset_id":""
        },
        {

"address":"tm1qjl0245evamhq8lswfmwruuep8d4ldhx8uxeugs",
            "amount":500000000,
            "type":"control_address",
            "asset_alias":"BTM",
            "asset_id":""
        },
        {
            "amount":10000000,
            "type":"spend_account",
            "account_alias":"alice",
            "account_id":"",
            "asset_alias":"BTM"
        }
    ],
    "ttl":1
}
```

将该交易的请求参数提交到build-transaction接口后，会被接口解析到BuildRequest结构中。BuildRequest结构如下：

```
api/request.go
type BuildRequest struct {
```

```
    Tx         *types.TxData              `json:"base_transaction"`
    Actions    []map[string]interface{}  `json:"actions"`
    TTL        json.Duration              `json:"ttl"`
    TimeRange uint64                     `json:"time_range"`
}
```

BuildRequest结构是交易的请求对象，字段说明如下。

· Tx：交易数据，在build过程中该字段没有使用，请求时置空即可。

· Actions：交易动作，所有的交易都是由一系列的交易动作构成的。资产转出方又称为input，资产接收方则称为output。

· TTL：构建交易的生存时间（单位为毫秒），在TTL指定的时间范围内，已经缓存的UTXO不能用于再一次构建交易。但如果剩余的UTXO足以支付新的交易，那么交易可以正常构建。当请求ttl为0时，系统默认设置为5分钟，即一个UTXO在5分钟内只能被一个交易引用。

· TimeRange：交易有效期，该字段一般填充为区块高度。在到达TimeRange指定的区块时，交易仍然没有被打包到区块中，那么这笔交易会过期失效。0表示不限制过期失效，默认为0。

在上面构造的交易中的，input action类型为spend_account，以账户模式花费5BTM资产，转入接收地址为“tm1qjl0245evamhq8lswfmwruuep8d4ldhx8uxeugs”的账户中。但是还有0.1资产没有output接收，其实这0.1资产是作为交易的手续费，在交易被打包到区块时，转入矿工的账户中。

区块链的每笔交易中至少需要包含一个input和output类型（Coinbase交易除外，因为Coinbase交易是用来产生新币的，因此它不会引用系统已经存在的UTXO）。非BTM资产交易的输入总和和输出总和必须相等，否则交易在构建时会报错，提示交易的输入和输出不平衡。比原链要求交易的手续费必须使用BTM资产结算，交易的手续费等于所有inputs的BTM资产数量减去所有outputs的BTM资产数量。比原链交易的资产单位是Neu（诺），兑换比例为：1 BTM=1000 mBTM=100000000 Neu。

提交的构建交易的JSON字符串，解析为BuildRequest结构后，提交给buildSingle方法构建交易。buildSingle方法将请求的数据校验后，编码组装为交易的核心数据结构TxData struct，然后将TxData赋值给Template struct返回。交易构建模板代码示例如下：

```
blockchain/txbuilder/types.go
type Template struct {
    Transaction         *types.Tx
    SigningInstructions []*SigningInstruction
    AllowAdditional bool
}
blockchain/txbuilder/signing_instruction.go
type SigningInstruction struct {
    Position          uint32
    WitnessComponents []witnessComponent
}

type witnessComponent interface {
    materialize(*[][]byte) error
}
```

Template交易模板结构说明如下。

· Transaction：保存的是与交易相关的信息。Transaction中包含TxData和bc.Tx两个部分，TxData是展示给用户的交易数据，即之前通过bytomcli获取的区块中交易数据的部分。bc.Tx是系统中处理交易数据时用到的缓存结构，这部分对用户是不可见的。

· SigningInstructions，保存的是交易的签名信息。Position记录该签名对于input action中第几个输入有效。WitnessComponents保存的是对input action的签名数据。如果build交易的signatures为null，表示没有签名；如果交易签名成功，则该字段会存有签名信息。

· AllowAdditional，标识是否允许在交易中附加数据。

SigningInstructions该字段是一个interface接口，主要包含3种不同的类型。

1）SignatureWitness

对交易模板Template中交易input action位置的合约program进行哈希，然后对哈希值进行签名。结构示例如下：

```
blockchain/txbuilder/signature_witness.go
type SignatureWitness struct {
    Quorum int `json:"quorum"`
    Keys []keyID `json:"keys"`
    Program chainjson.HexBytes `json:"program"`
    Sigs []chainjson.HexBytes `json:"signatures"`
}
```

SignatureWitness结构说明见表6-13。

表6-13　SignatureWitness结构

字段	字节数	说明
Quorum	8	解锁交易最少需要的签名个数
Keys	变长	包含主公钥 xpub 和派生路径 derivation_path，通过它们可以在签名阶段找到对应的派生私钥 child_xprv，然后使用派生私钥进行签名
Program	变长	签名的数据部分，将 program 的哈希值作为签名数据。如果 program 为空，则会根据当前交易 ID 和对应 action 位置的 InputID 两部分生成一个哈希，然后把它们作为指令数据自动构建一个 program
Sigs	变长	交易的签名。sign-transaction 执行完成后才会有值存在

2）RawTxSigWitness

对交易模板Template的交易ID和对应input action位置的InputID（该字段位于bc.Tx中）进行哈希，然后对哈希值进行签名。结构示例如下：

```
blockchain/txbuilder/rawtxsig_witness.go
type RawTxSigWitness struct {
```

```
    Quorum int `json:"quorum"`
    Keys []keyID `json:"keys"`
    Sigs []chainjson.HexBytes `json:"signatures"`
}
```

字段内容与SignatureWitness的相同，不再赘述。

3）DataWitness

该类型无需签名，验证合约program的附加数据，结构示例如下：

```
blockchain/txbuilder/data_witness.go
type DataWitness chainjson.HexBytes
```

AllowAdditional标识是否允许在交易中附加数据，如果为true，则交易的附加数据会添加到交易中，但是不会影响交易执行的program脚本，对签名结果不会造成影响；如果为false，则整个交易作为一个整体进行签名，任何数据的改变将影响整个交易的签名。

当构造的交易请求成功后，响应如下所示：

```
{
    "status":"success",
    "data":{

"raw_transaction":"070100010160015e28f36f6fc8cb95bb81fd97c20ebbd
10b8bbcf6b00afcaaaab8f14df9e7a6bb77ffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffff8094ebdc0301011600146d79639cc
d7367b0cd22c638a3f813ad3a2220ef010002013dffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffff809dd3e901011600144b8a6
d7390ffd1a7100b2b9de87be5fdcce2f7c400013dffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffff80cab5ee010116001497dea
ad32ceeee03fe0e4edc3e73213b6bf6dcc700",
        "signing_instructions":[
            {
                "position":0,
                "witness_components":[
                    {
                        "type":"raw_tx_signature",
                        "quorum":1,
                        "keys":[
```

```
                                {

"xpub":"821ebb8e08c68469f8a50d6de8ed5cb4dc48ecc8619a34a0528aa281
a3ff86bc29956538dd7629c307ab7a48951997870ce2bdfcd5944b88531b1b0c
c11f1adb",
                                    "derivation_path":[
                                        "0101000000000000000",
                                        "0100000000000000"
                                    ]
                                }
                            ],
                            "signatures":null
                        },
                        {
                            "type":"data",

"value":"c5f3984cd048d6176e873b711ccf25558d0d773e93c8db5741e67d5
b237c49bd"
                        }
                    ]
                }
            ],
            "allow_additional_actions":false
        }
}
```

6.5.2 签名交易

Alice在完成构建交易的操作后，剩下的操作就是签名确认这笔交易。在区块链上交易的签名和我们在银行转账时的签名是不同的。区块链的签名指的是使用密码学的算法对交易生成一个无法抵赖、伪造和篡改的数字签名。

基于UTXO模型进行交易和转账时，转出方只是将这笔资产声明为属于你，如果你想使用这笔资产进行交易，需要在交易时提供一个“解锁脚本”，验证并获取这笔资产。解锁脚本是每一笔交易输入的一部分，而且往往含有一个由用户的私钥生成的数字签名。签名交易的作用就是使用用户的私钥对交易中所有的input生成数字签名，在交易验证时，能够正确地获取这笔资产，作为交易的输入。

通过API接口/sign-transaction，我们可以对一笔构建好的交易进行签名。sign-transaction接受的请求参数如下所示：

```
api/hsm.go
type x struct {
    Password string `json:"password"`
    Txs txbuilder.Template `json:"transaction"`
}
```

password字段是签名的密码，根据密码，节点可以从数据库中解析出用户的私钥，然后用私钥对交易进行签名。Txs是交易的模板信息，模板信息是从build-transaction接口的构建结果中获取的。

SignatureWitness或RawTxSigWitness通过调用自己的Sign方法对交易的input进行签名，签名后的数据保存到各自的Sign数组中。交易签名代码示例如下：

```
blockchain/txbuilder/txbuilder.go
func Sign(ctx context.Context, tpl *Template, auth string,
signFn SignFunc) error {
    for i, sigInst := range tpl.SigningInstructions {
        for j, wc := range sigInst.WitnessComponents {
            switch sw := wc.(type) {
```

```
            case *SignatureWitness:
                err := sw.sign(ctx, tpl, uint32(i), auth,
signFn)
                // ...
            case *RawTxSigWitness:
                err := sw.sign(ctx, tpl, uint32(i), auth,
signFn)
                // ...
            }
        }
    }
    return materializeWitnesses(tpl)
}
```

在交易签名时，会根据input type的不同类型选择不同的签名方式：

· SignatureWitness类型签名：系统会对交易模板Template中交易input action位置的合约program进行哈希，然后对哈希值进行签名。

· RawTxSigWitness类型签名：系统会先对交易模板Template中交易ID和对应input action位置的InputID（该字段位于bc.Tx中）进行哈希，然后对哈希值进行签名。

Sign方法调用SignatureWitness或RawTxSigWitness完成对交易input的签名后，又通过materializeWitnesses方法将见证数据（解锁脚本）赋值到input的Arguments保存。这一部分主要是为方便交易验证过程。交易验证会用到bc层的数据结构，在完成签名后，提前将见证数据（解锁脚本）发送到bc层。

私钥签名过程是，在对交易进行签名时，采用ed25519椭圆曲线加密算法对交易进行签名。ed25519是著名密码学家Daniel J.Bernstein在2006年独立设计的椭圆曲线加密/签名/密钥交换算法，是目前最安全且速度最快的加密和签名算法。代码示例如下：

```
crypto/ed25519/chainkd/chainkd.go
func (xprv XPrv) Sign(msg []byte) []byte {
    return Ed25519InnerSign(xprv.ExpandedPrivateKey(), msg)
}
```

Ed25519InnerSign签名算法的输入是用户的私钥和签名的消息体。用户的私钥是根据用户的公钥从本地钱包查询到的。Ed25519InnerSign算法在签名时用私钥对（消息magic+消息体）的哈希进行签名。

签名成功后，sign-transaction接口的返回数据结构如下：

```
type signTemplateResp struct {
    Tx *txbuilder.Template `json:"transaction"`
    SignComplete bool `json:"sign_complete"`
}
```

Tx是包含签名信息的交易模板，相比build-transaction接口返回的模板，此时的Tx的signatures字段都已经填充了新的值。相应的raw_transaction字段也会因为补充了签名信息而长度变长了许多。SignComplete是一个布尔值，用来表示签名是否完成，如果为true表示签名完成；若为false则表示签名未完成。

如果交易需要使用多签，则一个用户签名完成后，需要将sign-transaction接口返回的数据交给另一个用户，另一个用户使用这些数据再次调用sign-transaction接口对交易进行签名。

签名失败一般可能是因为签名密码错误。签名失败只需将签名的交易数据用正确的密码重新签名即可，无需再次调用build-transaction接口构建交易。

6.5.3 提交交易

完成签名后，Alice要将交易通过API接口/submit-transaction提交上链，等待6个区块确认后，就可以认为这笔交易成功了。

这里一定要注意，提交交易并不代表交易已经完成，而是交易已经通过了本地验证，加入到本地交易池中。之后会通过交易同步发布到网络中的其他节点。网络中的节点验证交易合法，并通过挖矿过程将交易打包至区块中，才算是交易完成。但是此时交易也存在被篡改的可能。一般交易需要至少6个区块确认之后，才认为是无法篡改的。在比原链中，根据2.5分钟出一个块的频率，一笔交易在提交后大约15分钟后才算是真正确认无法篡改的。

submit-transaction接口的请求参数的数据结构如下所示：

```
api/transact.go
type ins struct {
    Tx types.Tx `json:"raw_transaction"`
}
```

Tx是签名完成之后的交易信息。在调用submit-transaction接口提交交易时，传入已签名后的交易信息。

submit-transaction接口接收到交易数据后，调用txbuilder.FinalizeTx方法验证交易的签名，并在UTXO结果集中标记被引用的UTXO。代码示例如下：

```
blockchain/txbuilder/finalize.go
func FinalizeTx(ctx context.Context, c *protocol.Chain, tx
*types.Tx) error {
    if fee := calculateTxFee(tx); fee >
cfg.CommonConfig.Wallet.MaxTxFee {
        return ErrExtTxFee
    }

    if err := checkTxSighashCommitment(tx); err != nil {
        return err
    }
```

```
    data, err := tx.TxData.MarshalText()
    if err != nil {
        return err
    }
    tx.TxData.SerializedSize = uint64(len(data))
    tx.Tx.SerializedSize = uint64(len(data))
    isOrphan, err := c.ValidateTx(tx)
    if errors.Root(err) == protocol.ErrBadTx {
        return errors.Sub(ErrRejected, err)
    }
    // ...
    if isOrphan {
        return ErrOrphanTx
    }
    return nil
}
```

FinalizeTx运行过程说明：

· 计算交易手续费Gas，一笔交易不能花费超过10BTM的手续费。

· 验证交易中的签名。

· 计算交易中的总字节数。

· 验证交易并添加到交易池中，等待广播至全网。

当交易验证通过后，submit-transaction接口会返回本次交易的ID。submit-transaction接口返回值的数据结构如下所示：

```
api/transact.go
type submitTxResp struct {
    TxID *bc.Hash `json:"tx_id"`
}
```

6.6 隔离见证

隔离见证（Segregated Witness）最早是在比特币中提出的。在中本聪设计的比特币版本中，将交易的签名和交易的内容放到了一起，这种方式在设计多个输入、多重签名的交易时，会导致交易的数据结构极其复杂，不利于交易的扩展。之后有人提出将交易的签名单独拿出来，放到一个Witness的结构中。见证（Witness）通常是指解锁脚本（或ScriptSig）。隔离见证就是比特币的一种结构性调整，旨在将见证数据从一笔交易的解锁脚本字段中移到一个伴随交易的单独见证数据结构中。另外将解锁脚本移动到交易外部，节省了区块空间。

比原链原生支持隔离见证。比原链的区块设计中将数据和见证分离，以实现资产的管理和分布式账本同步控制相分离。实现了更好的可编程性和合约支持，并且为之后的编程预留接口。

比原链中的隔离见证是通过随交易附加一个SigningInstructions结构而实现的，SigningInstructions中保存每个引用的UTXO的解锁脚本。关于SigningInstructions的实现，可参考6.5节。

6.7 交易脚本

交易脚本最初是由比特币提出的一种验证UTXO归属问题的方式。之后其他的数字货币也都采用这种方式来验证数字货币的归属。比原链中交易脚本的设计思想既沿用了比特币的设计思想，又做了相应的改进。比原链在交易脚本上最重要的改进是，比原链原生支持隔离见证（参见上一节）。比原链将交易的见证（即解锁脚本）从交易的输入中抽离出来，单独附加到交易的后面。用户在获取交易信息时，可以选择是否同时获取交易的见证信息。

之前我们在讲解比原链交易的输入和输出时，已经提到了两种脚本，一种是与交易输出相关的锁定脚本，一种是与交易输入相关的解锁脚本。

锁定脚本（locking script）是放置到交易ouput结构中的，使用control_program存储脚本，锁定脚本指定了今后如果需要花费这笔交易必须满足的条件。锁定脚本通常含有一个使用RIPEMD-160算法对公钥哈希后的字符串。因为它含有公钥相关的信息，这个脚本也称为脚本公钥。

解锁脚本（ScriptSig）是一个解决或者满足被锁定脚本在一个输出上设定花费条件的脚本，它允许消费输出。由于比原链采用的是隔离见证机制，解锁脚本又称为见证脚本，并且没有同输入绑定到一起，而是将见证脚本的内容保存在SigningInstruction结构。在SigningInstruction结构中使用Position指定对应的WitnessComponents（见证内容）是哪个输入的见证脚本。

每一个区块链节点都可以通过同时执行锁定脚本和解锁脚本来验证一笔交易的合法性。每一个输入都对应一个见证脚本，如果解锁脚本满足锁定脚本，则输入有效。

比原链中常用的输出脚本如下。

· P2WPKHProgram：支付到公钥哈希（Pay-to-Witness-Public-Key-Hash）。称为单签，占用20个字节。

· P2WSHProgram：支付到脚本哈希（Pay-to-Witness-Script-Hash）。称为多签，占用32个字节。

比原链中额外的锁定脚本如下。

· issuanceProgram：资产上链时的锁定脚本。

· retireProgram：资产销毁时的锁定脚本。

6.7.1 支付到公钥

当我们转账时，接收方的地址填写的是一个短地址时（20个字节），生成交易的output的锁定脚本就是P2WPKHProgram类型。短地址又称为单签地址，即该账户只包含一个扩展公钥。由于这种账户的地址和扩展公钥基本是一一对应的，因此P2WPKH Program又称为支付到公钥地址模式。

下面是一个支付到公钥地址交易的输出：

```
{
    "account_alias": "test2",
    "account_id": "0JUJKI4J00A04",
    "address": "tm1qenntqp82jh0svsnjuhzl4unfc6h827764rzcsm",
    "amount": 100000000,
    "asset_alias": "BTM",
    "asset_definition": {
        "decimals": 8,
        "description": "Bytom Official Issue",
        "name": "BTM",
        "symbol": "BTM"
    },
    "asset_id":
"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
    "control_program":
"0014cce6b004ea95df064272e5c5faf269c6ae757bda",
    "id":
"4f971f5cc14f6bdfd6e4ea78226566f3df769bffc048bdd95db5f3dd3e5749cb",
    "position": 1,
    "type": "control"
}
```

从上面信息可以看到control_program字段，该交易的锁定脚本是0014cce6b004ea95df064272e5c5faf269c6ae757bda，我们可以使用比原链命令行工具bytomcli解码该锁定脚本，命令执行如下：

```
$ ./bytomcli decode-program
0014cce6b004ea95df064272e5c5faf269c6ae757bda
```

```
{
  "instructions": "DUP \nHASH160 \nDATA_20
cce6b004ea95df064272e5c5faf269c6ae757bda\nEQUALVERIFY
\nTXSIGHASH \nSWAP \nCHECKSIG \n"
}
```

由上面的输出我们可以看出P2WPKHProgram类型脚本的格式：

```
DUP HASH160 DATA_20 [一个20字节的哈希值] EQUALVERIFY TXSIGHASH
SWAP CHECKSIG
```

脚本中20字节的哈希值是，使用账户的扩展密钥经Ripemd-160算法加密哈希后得到的。

P2PKHSigProgram虚拟机栈代码示例如下：

```
protocol/vm/vmutil/script.go
func P2PKHSigProgram(pubkeyHash []byte) ([]byte, error) {
    builder := NewBuilder()
    builder.AddOp(vm.OP_DUP)
    builder.AddOp(vm.OP_HASH160)
    builder.AddData(pubkeyHash)
    builder.AddOp(vm.OP_EQUALVERIFY)
    builder.AddOp(vm.OP_TXSIGHASH)
    builder.AddOp(vm.OP_SWAP)
    builder.AddOp(vm.OP_CHECKSIG)
    return builder.Build()
}
```

6.7.2 支付到脚本

我们转账时，当接收方的地址填写的是一个长地址时（32个字节），生成交易输出的锁定脚本是P2WSHProgram类型。长地址又称为多签地址，多签地址的底层需要多个签名来证明所有权，此后才能花费资金。比原链的多重签名采用的是M-N多签名机制。M-N多签名机制，将会使用N个公钥锁定一笔交易，但是如果需要花费这个输出时，用户至少需要提供M个公钥的签名，M-N多签名机制要求M必须要小于等于N。

下面是一个支付到多签地址交易的输出：

```
{
    "account_alias": "tet4",
    "account_id": "0K5DB7NFG0A08",
    "address":
"tm1qj7f7yrdf62h00ppysurw7gfn6n04c3asngprr0q9ln977d4ve48qwdqwtz
",
    "amount": 50000000,
    "asset_alias": "BTM",
    "asset_definition": {
        "decimals": 8,
        "description": "Bytom Official Issue",
        "name": "BTM",
        "symbol": "BTM"
    },
    "asset_id":
"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ff",
    "control_program":
"00209793e20da9d2aef784248706ef2133d4df5c47b09a0231bc05fccbef36
accd4e",
    "id":
"907cd3d6012bc8b0474037e0ae3190cdb50577b66f99287f495e6a2947b1dc
c0",
    "position": 1,
    "type": "control"
}
```

该交易的锁定脚本是：

```
00209793e20da9d2aef784248706ef2133d4df5c47b09a0231bc05fccbef36a
ccd4e
```

我们使用bytomcli解码该锁定脚本。命令执行如下：

```
$ ./bytomcli decode-program
00209793e20da9d2aef784248706ef2133d4df5c47b09a0231bc05fccbef36a
ccd4e
{
  "instructions": "DUP \nSHA3 \nDATA_32
9793e20da9d2aef784248706ef2133d4df5c47b09a0231bc05fccbef36accd4
e\nEQUALVERIFY \nDATA_8 ffffffffffffffff\nSWAP \nFALSE
\nCHECKPREDICATE \n"
}
```

由上输出我们可以看出P2WSHProgram类型脚本的格式如下：

```
DUP(0x 00) SHA3(0x 14) DATA_32 [一个32字节的哈希值] EQUALVERIFY
DATA_8  ffffffffffffffff SWAP FALSE CHECKPREDICATE
```

脚本中32字节的哈希值是多签交易中脚本的哈希值，多签脚本的格式如下：

```
TXSIGHASH PUBKEY1 PUBKEY2 PUBKEY3 M N CHECKMULTISIG
```

其中，M是需要的公钥个数，N是实际提供的公钥个数。

P2WSHProgram虚拟机栈代码示例如下：

```
protocol/vm/vmutil/script.go
func (b *Builder) addP2SPMultiSig(pubkeys []ed25519.PublicKey,
nrequired int) error {
    if err := checkMultiSigParams(int64(nrequired),
int64(len(pubkeys))); err != nil {
        return err
    }
```

```
    b.AddOp(vm.OP_TXSIGHASH)
    for _, p := range pubkeys {
        b.AddData(p)
    }
    b.AddInt64(int64(nrequired))
    b.AddInt64(int64(len(pubkeys)))
    b.AddOp(vm.OP_CHECKMULTISIG)
    return nil
}
```

6.7.3 资产上链

比原链除了支持单签和多签两种常用类型的脚本外，它还额外支持两种锁定脚本，一种是资产上链时产生的issuanceProgram锁定脚本，另一种是资产销毁时产生的retireProgram锁定脚本。

下面是一个发行资产上链交易的input项，issuanceProgram中保存的发行资产的锁定脚本：

```
{
    "amount":100000000,
    "asset_alias":"RMB",
    "asset_definition":{
        "decimals":8,
        "description":{

        },
        "name":"rmb",
        "symobol":"rmb"
    },

"asset_id":"30b52706cc7d8d1e192776f2b4e50114747fc1e8ac3088014601b7ae5ee0811f",

"input_id":"e15c5289c3e27b9c4cb4ac5b5f272d2fda61254a7d54ec7c4506955ebbea238f",

"issuance_program":"ae20986e833b6f84622b132ace40a3606006a70575265566ad103616ddc9e5806c175151ad",
    "type":"issue"
}
```

该交易的锁定脚本是：

```
ae20986e833b6f84622b132ace40a3606006a70575265566ad103616ddc9e5806c175151ad
```

我们使用bytomcli解码该锁定脚本。命令执行如下：

```
$ ./bytomcli decode-program
ae20986e833b6f84622b132ace40a3606006a70575265566ad103616ddc9e58
06c175151ad
{
  "instructions": "TXSIGHASH \nDATA_32
986e833b6f84622b132ace40a3606006a70575265566ad103616ddc9e5806c1
7\n1 01\n1 01\nCHECKMULTISIG \n"
}
```

由上面的输出可以看出issuanceProgram锁定脚本的格式如下：

```
TXSIGHASH DATA_32 [一个32字节的哈希值] n m CHECKMULTISIG
```

issuanceProgram锁定脚本的实质是一个多签脚本，类似上面提到的P2WSH脚本，issuanceProgram脚本同样是addP2SPMultiSig函数生成的，脚本格式为：

```
TXSIGHASH PUBKEY1 PUBKEY2 PUBKEY3 M N CHECKMULTISIG
```

代码示例如下：

```
asset/asset.go
func multisigIssuanceProgram(pubkeys []ed25519.PublicKey,
nrequired int) (program []byte, vmversion uint64, err error) {
    issuanceProg, err := vmutil.P2SPMultiSigProgram(pubkeys,
nrequired)
    if err != nil {
        return nil, 0, err
    }
    builder := vmutil.NewBuilder()
    builder.AddRawBytes(issuanceProg)
    prog, err := builder.Build()
    return prog, 1, err
}
```

6.7.4 资产销毁

retire锁定脚本是销毁资产（RetireProgram）时绑定到输出上的一种锁定脚本。下面是销毁资产时，retire类型的输出：

```
{
    "amount": 100000000,
    "asset_alias": "RMB",
    "asset_definition": {
        "decimals": 8,
        "description": {},
        "name": "rmb",
        "symobol": "rmb"
    },
    "asset_id":
"30b52706cc7d8d1e192776f2b4e50114747fc1e8ac3088014601b7ae5ee081
1f",
    "control_program": "6a",
    "id":
"e1f113b7e18efd9d5de61c839835a415b2390ce34410e3c0349a9ff3da6ab4
6e",
    "position": 0,
    "type": "retire"
}
```

由上面的输出我们可以看出，该retire输出的锁定脚本是6a，对应虚拟机中vm.OP_FAIL指令。我们仍然使用decode-program命令解码锁定脚本，命令执行如下：

```
$ ./bytomcli decode-program 6a
{
  "instructions": "FAIL \n"
}
```

使用decode-program命令解码后的脚本只包含一个FAIL操作。其实retire锁定脚本并不是只有一个FAIL，只是碰巧我们销毁的这笔资产没有arbitrary字段。如果销毁的资产带有arbitrary字段，retire

锁定脚本后面应该还会带有一些DATA操作。retire锁定脚本的格式如下所示：

```
FAIL [DATA_XX] [arbitrary数据]
```

代码示例如下：

```
protocol/vm/vmutil/script.go
func RetireProgram(comment []byte) ([]byte, error) {
    builder := NewBuilder()
    builder.AddOp(vm.OP_FAIL)
    if len(comment) != 0 {
        builder.AddData(comment)
    }
    return builder.Build()
}
```

6.7.5　见证脚本

见证脚本也称为解锁脚本，本书中提到的见证脚本或者是解锁脚本指的是同一种脚本。使用不同的名称是为了适应不同的语义环境，方便理解脚本的含义。

在6.6节中我们讲到，见证脚本是与交易输入一一对应的单独的SigningInstructions结构，见证脚本主要由两部分组成：用户的公钥和用户签名，如图6-9所示。

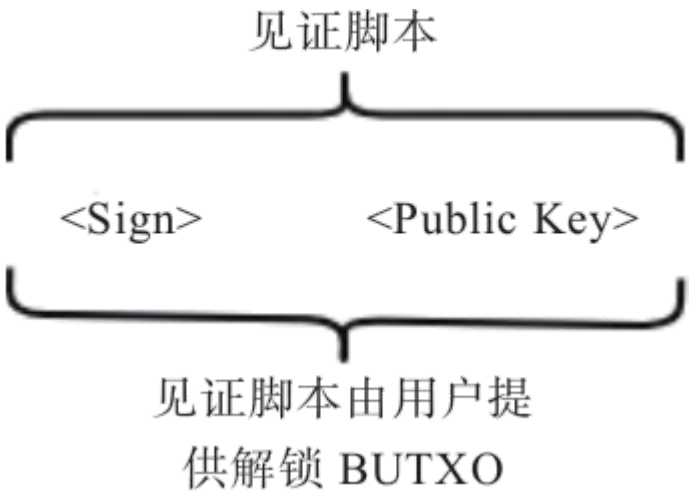


图6-9　见证脚本

1）Public Key（用户公钥）

在交易构建时，系统通过account/builder.go中UtxoToInputs方法生成交易的input和对应的见证数据SigningInstruction。在SigningInstruction中，通过AddWitnessKeys方法将用户的xpubs公钥添加到见证数据中。代码流程如下：

```
func (si *SigningInstruction) AddWitnessKeys(xpubs
[]chainkd.XPub, path [][]byte, quorum int) {
    hexPath := make([]chainjson.HexBytes, 0, len(path))
    for _, p := range path {
        hexPath = append(hexPath, p)
    }
```

```
    keyIDs := make([]keyID, 0, len(xpubs))
    for _, xpub := range xpubs {
        keyIDs = append(keyIDs, keyID{xpub, hexPath})
    }

    sw := &SignatureWitness{
        Quorum: quorum,
        Keys:   keyIDs,
    }
    si.WitnessComponents = append(si.WitnessComponents, sw)
}
```

2）Sign（交易签名）

交易流程中，比原链中一笔完整的交易一共包括三步：交易构建、交易签名、提交交易。其中交易签名时生成的签名就是见证脚本中的Sign。关于Sign生成，可参考6.5.2节。

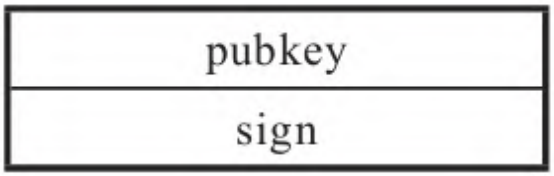
6.7.6 栈语言

解锁脚本一般由用户的公钥和交易的签名组成。将解锁脚本和锁定脚本拼接在一起可以组成一个完整的可被BVM虚拟机执行的程序。图6-10是一个典型的完整交易脚本格式。

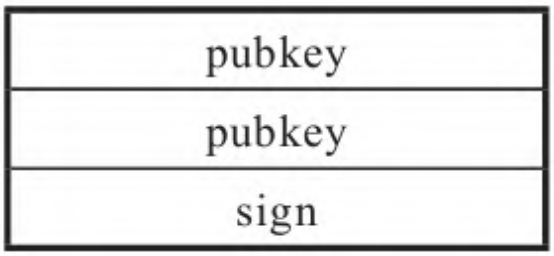
图6-10 比原链脚本格式

脚本的执行类似于逆波兰表达式的基于栈的执行语言，其实我们也可以直接把脚本看做一个逆波兰表达式。对于逆波兰表达式，相信大家应该都不会陌生，在我们学习数据结构时都介绍过这种表达式。逆波兰表达式的执行需要依赖一个数据栈，数据通过入栈和出栈操作完成运算。下面我们简单介绍一下脚本是如何完成验证的。具体的栈实现和操作原理，大家可以参考第8章中BVM虚拟机的内容。

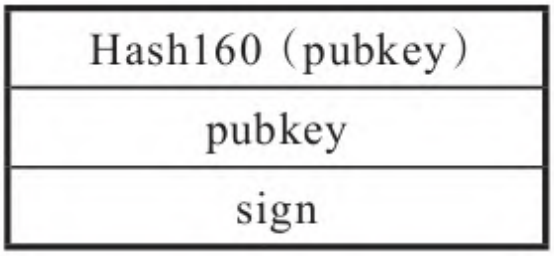
首先我们需要初始化一个空栈，将交易签名sign和公钥pubkey压入栈底。示例如下：



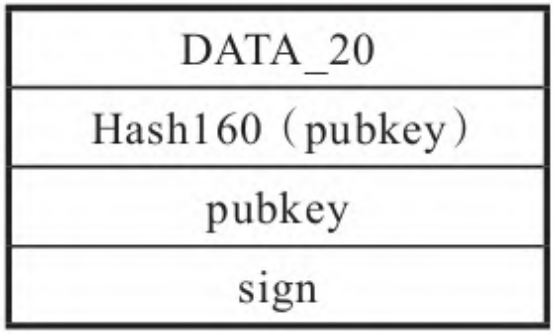
DUP指令操作的意思是将栈顶元素复制一份，此时栈内元素如下所示：



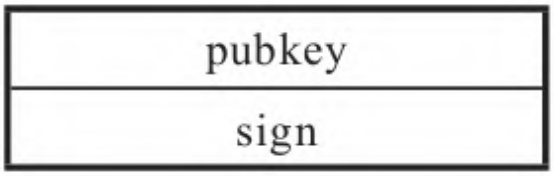
HASH160指令弹出栈顶元素，并对弹出元素执行hash160操作，将操作后的结果压入栈顶。示例如下：



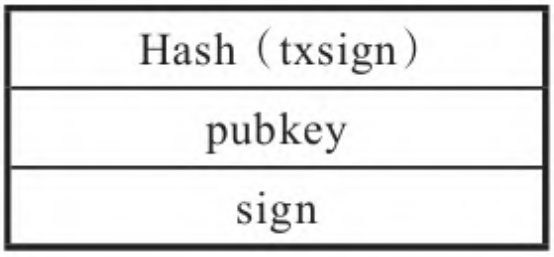
DATA_20[pubkeyHash]将20字节的哈希值压入栈顶。示例如下：



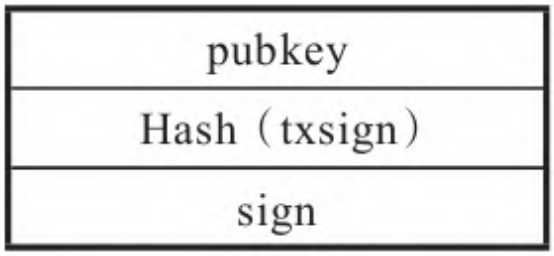
EQUALVERIFY指令弹出栈顶两个元素，验证两个元素是否相等，如果相等继续后面的操作；如果不相等，则运算退出。示例如下：



TXSIGHASH指令计算出交易签名的哈希值后，压入栈顶。示例如下：



SWAP指令交换栈顶两个元素的位置。示例如下：



CHECKSIG指令执行时依次从栈中弹出公钥、交易签名的哈希值和交易签名，验证sign是否是pubkey的有效消息签名。如果CHECKSIG过程没有返回错误，则脚本验证不通过。

6.8 交易验证

交易验证是整个交易环节最重要的一个环节。对于用户来说，交易验证是保证用户财产安全的重要手段。而对于整个比原链来说，交易验证是保证比原链长期稳定运行和持续发展的重要方式。比原链的安全性除了依靠分布式账簿自身的安全性之外，最重要的是能否设计一种安全可靠的交易验证机制。对于比原链这种支持多元比特资产交互协议的区块链，如何在验证的时候处理好多种资产之间的相互转换和相互引用是一件比较困难的事情。比原链设计者做了大量的工作，并且提供了一种目前看来十分安全的交易验证机制。

交易验证主要用在以下几个场景中：

· 用户完成一笔交易的签名时，需要将交易提交到区块链网络中，使交易能够尽快确认。节点在提交交易之前，需要先验证交易，确认交易的合法性。

· 节点收到其他节点广播的交易时，节点需要先验证交易是否合法，合法的交易才会加入节点的交易池。

· 当一个挖矿节点成功计算出符合要求的哈希值后，节点会将交易池中的交易打包到区块中。节点在打包交易的时候需要验证交易的合法性。

· 节点收到其他节点同步的区块时，也需要一一验证区块中包含的交易。

6.8.1 标准交易

什么样的交易是一个标准的交易？在比原链中给出的定义是：当一笔交易中所有output的锁定脚本都是比原链所支持的类型，那么这笔交易就是标准交易。

如比原链中的锁定脚本主要有三种：P2WPKHScript（单签脚本）、P2WSHScript（多签脚本）和Straightforward（销毁资产的脚本）。

标准P2WPKHScript脚本必须是以0014开头的，00对应的是BVM的OP_0操作，该操作会将一个空字符串压入BVM栈。14对应的是OP_DATA_20操作，该操作会将紧随其后的20个数压入BVM栈。

标准的P2WSHScript则必须是以0020开头的，00对应的操作同P2WPKHScript脚本相同，20对应的是OP_DATA_32操作，该操作会将紧随其后的32个数压入BVM栈中。

Straightforward脚本只有两类：一类脚本内容为6a，对应的操作为OP_FAIL，代表这个脚本会无条件的执行失败。还有一类是51，对应的操作为OP_TRUE，该操作会将数字1压入BVM栈。

6.8.2 交易验证流程

在介绍交易验证流程之前，需要先介绍MapTx方法，在第5章我们提到比原链中数据结构的定义是分层的，types层和bc层之间相互转换是通过MapTx方法的。

protocol/bc包与交易相关的数据结构一共有八种。每种数据结构都实现了Entry接口的方法，因此它们都是Entry类型。MapTx方法就是将交易中原来由protocol/bc/type包下数据结构定义的数据转换成以上八种Entry类型，并添加相应的依赖关系。MapTx转换流程如图6-11所示。

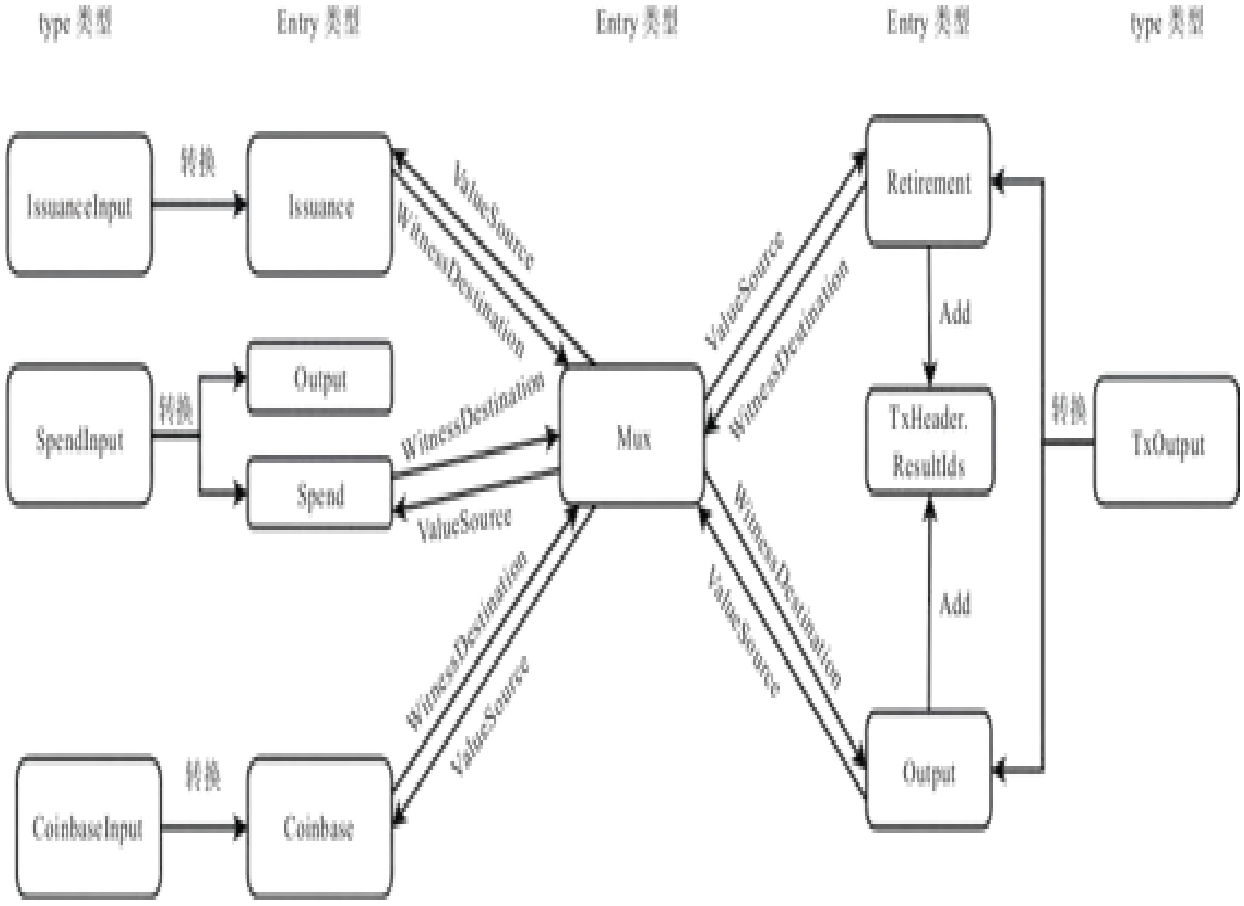


图6-11 MapTx转换流程

了解MapTx转换过程后，接下来介绍validation.ValidateTx交易验证的流程（代码位置：protocol/tx.go和

protocol/validation/tx.go）。

1）根据交易ID查询节点的交易池中是否已经存在相同的交易。如果交易池中已经存在相同的交易，则交易验证不通过，返回ErrCache错误。代码示例如下：

```
if ok := c.txPool.HaveTransaction(&tx.ID); ok {
    return false, c.txPool.GetErrCache(&tx.ID)
}
```

2）验证交易的版本和区块的版本是否对应。目前比原链中交易版本号和区块的版本号都是1。版本号的验证的主要目的是防止在后期比原链出现硬分叉时，导致低版本的交易被错误地打包到高版本的区块中。代码示例如下：

```
if block.Version == 1 && tx.Version != 1 {
    return gasStatus, errors.WithDetailf(ErrTxVersion, "block
version %d, transaction version %d", block.Version, tx.Version)
}
```

3）验证交易序列化后的字节数组的长度。如果序列化后交易信息的长度等于0，说明这个交易在序列化时出现了未知错误。要做到这个交易即使广播出去，其他节点也无法反序列化获取交易的正确内容。代码示例如下：

```
if tx.SerializedSize == 0 {
    return gasStatus, ErrWrongTransactionSize
}
```

4）验证交易的TimeRange，在TimeRange指定的高度，交易仍然没有被打包到区块中，交易会自动作废。如果交易的TimeRange比交易提交时的区块高度还小，那么这笔交易创建时就是一笔过期交易，永远也不会打包到交易中。因此，需要将这种没有实际意义的交易过滤掉。代码示例如下：

```
if err := checkTimeRange(tx, block); err != nil { // TimeRange
>= block.Height
    return gasStatus, err
}
```

5）验证交易是否是一个标准交易。代码示例如下：

```
if err := checkStandardTx(tx); err != nil {
    return gasStatus, err
}
```

6）核心验证流程，即验证交易花费的每一个BUTXO是否合法。交易的核心验证流程由protocol/validation/tx.go文件下的checkValid方法实现。交易验证的两条核心原则是：

· 等价交易：各种资产输入和输出的总量必须相等。

· 权限验证：对于spend类型的输入，用户提供的解锁脚本要正确的。

在6.4.2节介绍Mux结构时，我们提到在交易验证之前，系统会调用MapTx方法将protocol/bc/type包下的交易数据结构转化为protocol/bc包下Entry类型。因此，checkValid方法验证交易其实就是验证每个Entry类型是否满足上面两条原则。说明如下：

· bc.TxHeader Entry验证。bc.TxHeader是checkValid方法验证交易的入口。bc.TxHeader中记录了交易中其他Entry类型的信息，在验证过程中会递归验证其中记录的其他Entry类型。

· bc.Output和bc.Retirement Entry验证。使用checkValidSrc方法验证bc.Output或bc.Retiremen与Mux之间的对应关系是否正确，并且验证资产的数量是否相等。

· bc.Coinbase Entry验证。由于Coinbase交易是创币交易，比原链中的创币交易只能产生BTM币，因此bc.Coinbase验证时，首先需要保证在MapTx时，bc.Coinbase导入到Mux中的资产类型一定是BTM资产，并且

还要检查Coinbase交易的附带标注信息最多不能超过128个字符。之后就是使用checkValidDest保证Coinbase与导入到Mux上的资产数量相等。

· bc.Issuance和bc.Spend Entry验证。Issuance和Spend在交易时需要提供资产所有权证明。因此，交易验证时首先要做的就是验证用户提供的资产所有权证明，即验证比原链中的解锁脚本是否合法。比原链使用BVM虚拟机验证用户的解锁脚本，具体的验证原理，可参考6.7节。

使用BVM虚拟机验证用户的解锁脚本并不是一个免费的操作，需要消耗Gas。因此，用户需要保证在交易时提供了足够的交易费。这里需要说明的一点是，BVM在执行脚本过程中消耗Gas并不是真实地消耗了Gas，只是在数值上做的定义，并没有真正扣除交易费用。在交易打包时，系统才把交易费用转账给打包交易的矿工，这时候交易才真正扣除了Gas。之后，使用checkValidDest保证Issuance或Spend分别与导入到Mux上的资产数量相等。

· bc.Mux Entry验证。在上面四种类型的验证过程中，通过checkValidSrc保证了Mux与交易输出bc.Output、bc.Retirement之间的资产数量相等；通过checkValidDest保证了交易输入bc.Issuance、bc.Spend和bc.Coinbase与Mux上的资产数量相等。checkValidSrc和checkValidDest验证，只是保证了每一输入或输出Mux中对应的Sources和WitnessDestinations之间价值相等，但是无法保证每种资产输入与输出的价值相等。因此Mux验证的主要内容是，保证每种资产输入与输出的资产数量相等。

6.9 交易费

交易费也成为“Gas”。在区块链上任何人都可以读写数据，读取数据是免费的，因为它是一个公开的账本。但是向区块链中写入数据是需要花费一定费用的，在区块链上存储资源是宝贵的，这种操作有助于阻止垃圾内容，并通过支付保护其安全性。网络上的任何节点都可以参与挖矿，这一方式保护了网络。

由于挖矿需要计算能力和电费，所以矿工们的服务需要得到一定的报酬，这也是矿工费的由来。一般情况下，矿工在将交易打包成区块的过程中，会优先打包Gas费用最高的交易，保证矿工的收入最优。如果用户交易时所支付的矿工费非常低，那么这笔交易可能不会被矿工打包，从而造成交易失败。

6.9.1 估算交易手续费

交易在构建完成后，通过estimate-transaction-gas接口估算本次交易的手续费，估算的结构需要重新加入到build-transaction的结果中，然后对交易进行签名和提交。估算手续费结构体如下：

```
consensus/general.go
VMGasRate         = int64(200)
api/transact.go
type EstimateTxGasResp struct {
    TotalNeu   int64
    StorageNeu int64
    VMNeu      int64
}
```

估算手续费字段说明如下。

· VMGasRate：Gas与Neu的兑换比例，1 Gas=200 Neu（诺）。

· TotalNeu：预估的总手续费（单位为neu），该值直接加到build-transaction的BTM资产输入action中。

· StorageNeu：存储交易的手续费。

· VMNeu：运行虚拟机的手续费。

估算手续费的公式为：

totalGas：=totalTxSizeGas+totalP2WPKHGas+totalP2WSHGas

totalTxSizeGas=交易数据长度+签名数据长度×StorageGasRate，StorageGasRate为交易存储的费率1 Gas。

totalP2WPKHGas=单签地址模式的output个数×1419 Gas。

totalP2WSHGas=多签地址模式的output个数×（984×a－72×b－63），

a为WitnessComponents中存储的主公钥的个数，b为账户key的个数。

估算交易费用的源码如下：

```
api/transact.go
func EstimateTxGas(template txbuilder.Template)
(*EstimateTxGasResp, error) {
    data, err := template.Transaction.TxData.MarshalText()
    // ...
    baseTxSize := int64(len(data))

    signSize := estimateSignSize(template.SigningInstructions)

    totalTxSizeGas, ok := checked.MulInt64(baseTxSize+signSize,
consensus.StorageGasRate)
    if !ok {
        return nil, errors.New("calculate txsize gas got a math
error")
    }

    // consume gas for run VM
    totalP2WPKHGas := int64(0)
    totalP2WSHGas := int64(0)
    baseP2WPKHGas := int64(1419)

    for pos, inpID := range template.Transaction.Tx.InputIDs {
        sp, err := template.Transaction.Spend(inpID)
        // ...
        resOut, err :=
template.Transaction.Output(*sp.SpentOutputId)
        // ...

        if segwit.IsP2WPKHScript(resOut.ControlProgram.Code) {
            totalP2WPKHGas += baseP2WPKHGas
        } else if
segwit.IsP2WSHScript(resOut.ControlProgram.Code) {
            sigInst := template.SigningInstructions[pos]
            totalP2WSHGas += estimateP2WSHGas(sigInst)
        }
    }

    // total estimate gas
    totalGas := totalTxSizeGas + totalP2WPKHGas + totalP2WSHGas
```

```
    // rounding totalNeu with base rate 100000
    totalNeu := float64(totalGas*consensus.VMGasRate) /
defaultBaseRate
    roundingNeu := math.Ceil(totalNeu)
    estimateNeu := int64(roundingNeu) * int64(defaultBaseRate)

    return &EstimateTxGasResp{
        TotalNeu:   estimateNeu,
        StorageNeu: totalTxSizeGas * consensus.VMGasRate,
        VMNeu:      (totalP2WPKHGas + totalP2WSHGas) *
consensus.VMGasRate,
    }, nil
}
```

6.9.2 计算交易手续费

计算一笔已经被确认过的交易的费用，公式如下：

交易费用=totalInputBTM－totalOutputBTM

其中：totalInputBTM：遍历交易中的所有input金额相加；totalOutputBTM：遍历交易中的所有output金额相加。

计算一笔交易的矿工费，代码如下：

```
blockchain/txbuilder/finalize.go
func calculateTxFee(tx *types.Tx) (fee uint64) {
    totalInputBTM := uint64(0)
    totalOutputBTM := uint64(0)

    for _, input := range tx.Inputs {
        if input.InputType() != types.CoinbaseInputType &&
input.AssetID() == *consensus.BTMAssetID {
            totalInputBTM += input.Amount()
        }
    }

    for _, output := range tx.Outputs {
        if *output.AssetId == *consensus.BTMAssetID {
            totalOutputBTM += output.Amount
        }
    }

    fee = totalInputBTM - totalOutputBTM
    return
}
```

6.10 交易池

当交易广播到网络中并且被矿工接收到时，矿工会将收到的交易加入到本地的TxPool交易池中，TxPool对象管理本地交易池。交易池相当于一个缓冲区，它并不是无限大。默认情况下在比原链中交易池最大可以存储10000笔交易。如果超出这个值，则会返回"transaction pool reach the max number"的错误。

交易池的数据结构如下：

```
type TxPool struct {
  lastUpdated   int64
  mtx           sync.RWMutex
  store         Store
  pool          map[bc.Hash]*TxDesc
  utxo          map[bc.Hash]*types.Tx
  orphans       map[bc.Hash]*orphanTx
  orphansByPrev map[bc.Hash]map[bc.Hash]*orphanTx
  errCache      *lru.Cache
  msgCh         chan *TxPoolMsg
}
```

交易池数据结构说明如下。

· lastUpdated：上次更新时间，只有在交易池中新增和删除交易时，系统才会更新lastUpdated。

· store：数据库存储接口，提供操作LevelDB的接口。

· pool：存储未确认交易的map结构，key为交易ID。

· utxo：存储未确认交易使用的BUTXO的map结构，key为BUTXO的ID。

· orphans：孤交易。

· orphansByPrev：存储孤交易缺失的BUTXO与孤交易的对应关系。

· errCache：长度1000的LRU缓存，存储未通过验证交易的错误信息。

· msgch：长为1000的消息队列，交易池增删交易时，通知钱包做相应的处理。

在区块链中，无论是本地节点产生的交易和还是同步其他节点的交易，在交易验证完成之后，都会通过ProcessTransaction方法将交易加入到交易池中，等待被节点打包成区块。交易池中的交易分为两类：正常交易和孤交易。与正常交易不同的是，孤交易中消费的BUTXO暂时没有在本地查找到它的引用。在交易同步时，由于网络传输的原因，造成交易到达节点的顺序不一致，就会出现孤交易。

孤交易最长只能在内存中保存10分钟，节点会通过orphanExpireWorker方法每3分钟清理一次过期的孤交易。节点最终只能保存2000条孤交易。如果超出这个值，节点会报“transaction pool reach the max number”错误。

交易池对外提供了如下方法操做其中的交易。

· func（tp*TxPool）AddErrCache()：添加未通过验证交易的错误信息。

· func（tp*TxPool）ExpireOrphan()：删除某个时间之前所有孤交易。

· func（tp*TxPool）GetErrCache()：获取交易验证的错误信息。

· func（tp*TxPool）GetMsgCh()：获取msgch通道。

· func（tp*TxPool）RemoveTransaction()：从交易池中移除交易。

· func（tp*TxPool）GetTransaction()：查询交易池中的一笔交易。

· func（tp*TxPool）GetTransactions()：查询交池中所有交易。

· func（tp*TxPool）IsTransactionInPool()：查询一笔交易是否在交易池中。

· func（tp*TxPool）IsTransactionInErrCache()：查询之前的交易是否存在验证失败的错误。

· func（tp*TxPool）HaveTransaction()：验证交易池中是否有这笔交易的记录，无论验证通过或者失败。

· func（tp*TxPool）ProcessTransaction()：交易验证通过后，将交易加入到交易池中。

6.11 默克尔树

默克尔（Merkle）树是一种典型的二叉树结构，又叫哈希树，是一种用作快速归纳和校验大规模数据完整性的数据结构，由一个根节点、一组中间节点和一组叶子节点组成。在区块链系统出现之前，默克尔树广泛用于文件系统和P2P系统中，图6-12所示。

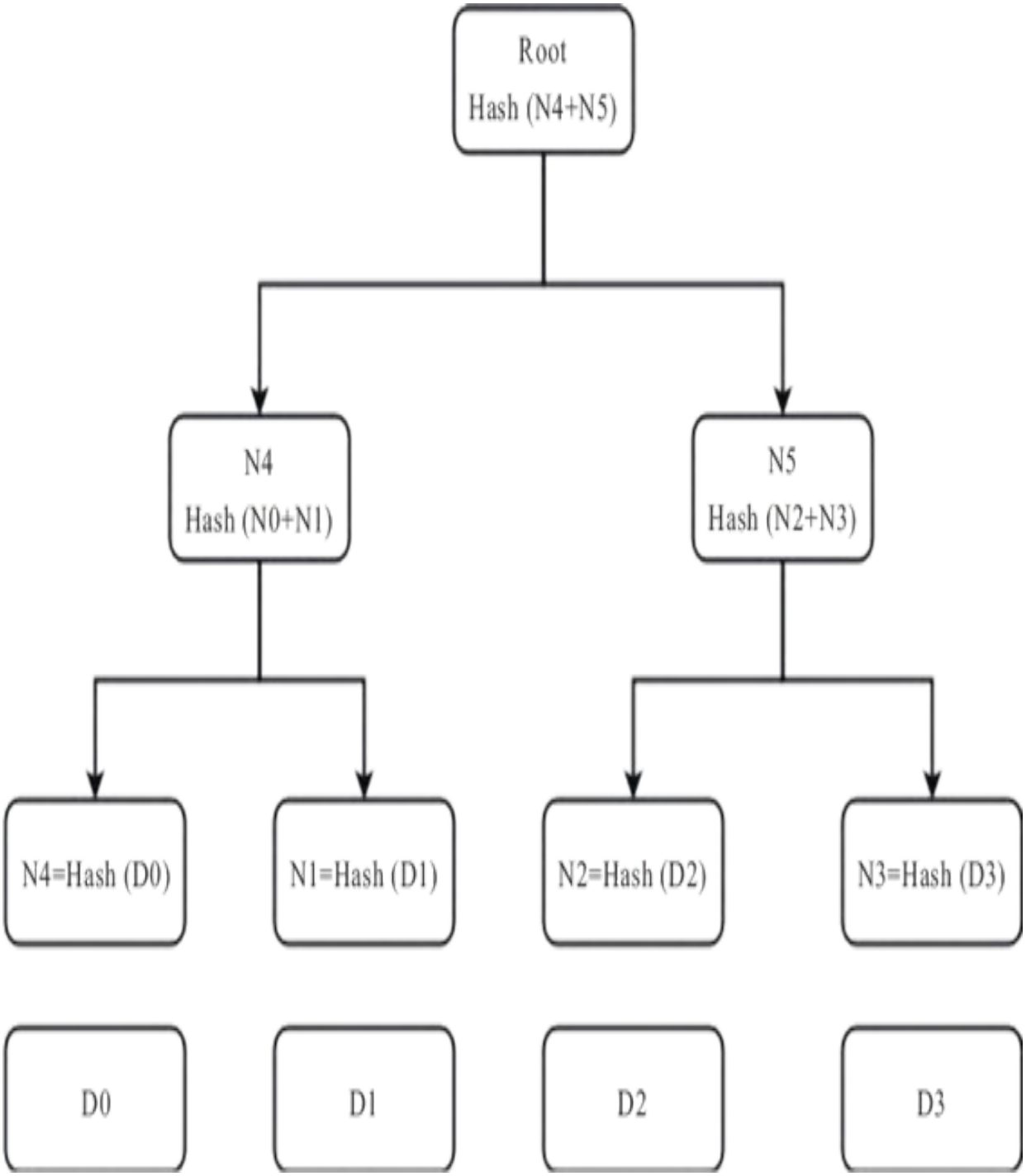


图6-12　默克尔树

默克尔树的主要特点包括：

· 最下面的叶子节点用于存储数据或其哈希值。

· 非叶子节点（包括中间节点和根节点）用于存储其两个子节点内容拼接之后的哈希值。

默克尔树记录哈希值的特点，让它具备了一些独特的性质。例如，底层数据的任何变动，都会传递到其父节点，一层层地沿着路径直达树根，这意味着树根的值实际代表了对底层所有数据的数字摘要。

默克尔树的应用场景非常多，如下：

· 可以用于快速比较大量数据。对每组数据排序后构建默克尔树结构，当两个默克尔树根节点相同时，则意味着两组数据必然相同；否则，必然不同。

· 快速定位出现变更的数据。如图6-12所示，如果D1中的数据被修改，会影响N1，N4和Root节点。因此，一旦发现某个节点的数字发生变化，沿着子节点往下查找最多只需要O（logn）时间即可定位发生改变的数据。

· 零知识证明。证明者能够在不向验证者提供任何有用的信息的情况下，使验证者相信某个论断是正确的。如果某个人需要向他人证明其拥有一组数据中的某个值，又不想暴露其拥有的其他数据，则证明者完全可以将这组数据构建成一棵默克尔树，向验证者公布需要证明的一条路径。

在比原链中，可以使用默克尔树来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，提供了一种校验区块是否存在某交易的高效途径。

默克尔树的数据结构如下：

```
protocol/bc/types/merkle.go
type merkleTreeNode struct {
    hash  bc.Hash
    left  *merkleTreeNode
    right *merkleTreeNode
}
```

默克尔树的数据结构说明如下。

· hash：当前节点的哈希值。

· left：左子树。

· right：右子树。

同时比原链也封装了如下方法操作默克尔树（具体的实现可以参考protocol/bc/types/merkle.go文件中的实现）。

· func merkleRoot()：找到一组节点的默克尔树根节点。

· func interiorMerkleHash()：计算两个节点的merkle哈希值。

· func leafMerkleHash()：计算叶子节点的merkle哈希值。

· func buildMerkleTree()：根据给定的节点数据构造一棵默克尔树。

· func GetTxMerkleTreeProof()：返回一棵默克尔树的证明，通常用来证明一笔交易确实存在于默克尔树中。

· func GetStatusMerkleTreeProof()：返回一棵默克尔树，通常用来证明一笔交易的结果是合法的。

· func getMerkleRootByProof()：根据证明计算默克尔树的根节点。

· func newMerkleTreeNode()：构造一棵默克尔树节点。

· func validateMerkleTreeProof()：验证默克尔树。根据给定节点的哈希值和Flag参数计算merkle树根，只有当计算出的树根等于给定的值，并且构造的默克尔树中包含所有给定的节点，验证结果才为真。

· func ValidateTxMerkleTreeProof()：验证默克尔树中的交易。

· func ValidateStatusMerkleTreeProof()：验证默克尔树的交易的状态。

· func TxStatusMerkleRoot()：根据给定的交易验证结果构造一棵交易状态默克尔树。

· func TxMerkleRoot()：根据给定的交易构造一棵默克尔树。

（1）构建默克尔树

生成一棵完整的默克尔树需要递归地对哈希节点对进行哈希运算，并将新生成的哈希节点插入到默克尔树中，直到只剩一个哈希节点，该节点就是默克尔树的根。在比原链中使用SHA256算法计算节点的哈希值。

构建默克尔树的源码如下：

```
func buildMerkleTree(rawDatas []merkleNode) *merkleTreeNode {
    switch len(rawDatas) {
    case 0:
        return nil
    case 1:
        rawData := rawDatas[0]
        merkleHash := leafMerkleHash(rawData)
        node := newMerkleTreeNode(merkleHash, nil, nil)
        return node
    default:
        k := prevPowerOfTwo(len(rawDatas))
        left := buildMerkleTree(rawDatas[:k])
        right := buildMerkleTree(rawDatas[k:])
        merkleHash := interiorMerkleHash(&left.hash,
&right.hash)
        node := newMerkleTreeNode(merkleHash, left, right)
        return node
    }
}
```

（2）交易验证

为了证明区块中包含某一个特定的交易，一个节点只需要计算log2（N）个32字节的哈希值，形成一条从特定交易到树根的认证路径或者merkle路径即可。随着交易数量的急剧增加，这样的计算量就显得十分重要，因为相对于交易数量的增长，以2为底的交易数量的对数的增长会缓慢许多。这使得比原链节点能够高效地产生一条10个或者12个哈希值（320～384字节）的路径，来证明在一个巨量字节的区块上成千上万交易中的某笔交易的存在。

如图6-12，如果我们要验证一个区块中存在一笔交易N2，则至少需要N3和N4这两个哈希值，由这两个哈希值产生确认路径，再通过计算N5和默克尔树的根节点的哈希值，就可以证明N2包含在默克尔树的根节点中。

默克尔树验证的源码如下：

```
func validateMerkleTreeProof(hashes []*bc.Hash, flags []uint8,
relatedNodes []merkleNode, merkleRoot bc.Hash) bool {
    merkleHashes := list.New()
    for _, relatedNode := range relatedNodes {
        merkleHashes.PushBack(leafMerkleHash(relatedNode))
    }
    hashList := list.New()
    for _, hash := range hashes {
        hashList.PushBack(*hash)
    }
    flagList := list.New()
    for _, flag := range flags {
        flagList.PushBack(flag)
    }
    root := getMerkleRootByProof(hashList, flagList,
merkleHashes)
    return root == merkleRoot && merkleHashes.Len() == 0
}
```

默克尔树广泛用于SPV节点。SPV节点不保存所有交易也不下载整个区块，而是仅仅保存区块头。它们使用认证路径或者merkle路径来验证交易存在于区块中，而不必下载区块中所有交易。

6.12 本章小结

本章详细介绍了区块链的基础——交易，详细阐述了交易的结构，如何通过API接口完成一笔交易，满足哪些规则的交易才是一笔合法的交易。同时也对比原链中的BUTXO模型和交易脚本做了深入细致的分析。

第7章

内核层：智能合约

7.1 概述

在比原链的架构中，智能合约是重要的组成部分。智能合约可以分为UTXO模型和Account模型，由于比原链面向金融体系，从公链设计角度来说采用UTXO模型更适合这类场景。此外，比原链的智能合约还分为创世合约和普通合约，并使用Equity语言进行编写。

本章主要内容如下：

· 智能合约基础知识，对比两种模型下智能合约的优缺点。

· 比原链合约层的设计，对创世合约和普通合约进行作用分析。

· Equity语言介绍，从最基本的数据类型到合约的内置函数，使读者可以逐步掌握合约的书写规则。

· 基于UTXO模型的合约开发实践，从实践的角度介绍合约操作流程，以及此过程中可能存在的问题及注意事项。

7.2 基础知识

7.2.1 智能合约

合约就是有共识的约定，写到区块链并被多方认证。

智能合约是指当达到某时间、某条件或特定情况发生的时候，合约就会被自动触发，然后系统就会自动执行相应的操作，无需人为干预。

最早的智能合约概念由以太坊提出。利用分布式账本的高度安全性和高效执行把程序编写成合约的样式。

7.2.2 图灵完备的智能合约

我们经常看到很多公链项目自称是图灵完备的，包括比原链。那么图灵完备到底是什么样的技术？

图灵完备性（Turing Completeness）是针对一套数据操作规则而言的概念。数据操作规则可以是一门编程语言，也可以是计算机里具体实现了的指令集。当这套规则可以实现图灵机模型里的全部功能时，就称它具有图灵完备性。简单来说，一个语言是否图灵完备，需要看该语言是否支持分支语句、循环语句、递归以及各种算法等，如果都支持则说明该语言是图灵完备的。

合约的生命周期包括条件触发和限制循环（或递归），后者让合约的生命周期结束，保证系统的每段程序都不会陷入死循环，都会有运行结束的时候。

7.2.3 UTXO模型和Account模型

目前公链中主要有两种保存记录的模型：UTXO模型和Account账户模型。UTXO模型（script），通过script实现了签名验证逻辑，所有的信息都存放在UTXO中，不依赖任何一个对象或状态。代表性的公链有比特币、比原链。Account账户模型，账户余额、合约代码、存储空间、具备图灵完备的虚拟机等，提供运行智能合约的基础设施，状态是面向对象的，有一系列关联。代表性的公链有以太坊、EOS。

UTXO模型中每笔交易会引用之前交易生成的UTXO，然后生成新的UTXO，全局状态即当前所有未花费的UTXO集合。Account账户模型意图创建一个更为通用的协议，该协议支持图灵完备的编程语言，在此协议上用户可以编写智能合约，创建各种去中心化的应用。表7-1对这两种模型进行了对比。

表7-1 UTXO模型与Account模型

	UTXO模型	Account模型
代表性公链	比特币、比原链	以太坊、EOS
状态	无状态	有状态（世界状态）
交易	引用UTXO并产生新的UTXO	触发账户与合约之间的消息交互
program	存储在UTXO中，用于花费UTXO的认证判断	存储在合约账户中，用于改变状态和发送消息
图灵完备	比特币（不支持）、比原链（支持）	以太坊（支持）、EOS（支持）

UTXO模型和Account账户模型两者最大的区别是：无状态和有状态。UTXO模型实现图灵完备的智能合约是困难的。而Account模型就是为了智能合约而实现的，为了易于管理账户，引入了世界状态，每一笔交易都会改变这个世界状态。这与现实世界是相对应的，每一个微小的改变，都会改变这个世界。

1. UTXO模型优缺点

UTXO模型的优点包括：

· 功能分离，节点负责结果的验证；钱包负责UTXO的计算。这在一定程度上减少主链的负担。

· 交易无法重放，交易的先后顺序和依赖关系容易验证。

· 无状态模型，支持并发处理。

· 具有更好的隐私性。

UTXO模型的缺点包括：

· 无法实现一些比较复杂的逻辑，对于需要保存状态的合约实现难度大。

· UTXO模型中，当input较多时，见证脚本会增多。签名本身需要消耗CPU和存储空间。

2. Account模型优缺点

Account模型的优点包括：

· 合约以代码的形式保存在Account中，拥有更好的可编程性。

· 交易的成本极大降低，无需像UTXO那样花费签名验证和存储资源。

Account模型的缺点包括：

· Account模型交易之间没有依赖性，需要解决重放问题。

· 对于实现闪电网络等，用户举证需要更复杂的Proof证明机制，子链向主链进行状态迁移需要更复杂的协议。

7.3 合约层设计

比原链采用基于UTXO模型的智能合约。此外合约还分为两种：创世合约与普通合约。用户能操作的合约为普通合约：

· 创世合约是比原链上一个特殊类型的合约种类，是可以发行并审核智能合约的合约，开发者保留部分权限，例如私钥、作用域等，并具有一定的规范和自动化审计功能，以确保链上资产符合相应的规范和模板进行登记和发布。创世合约的底层实现会调用数据传输层中的发布程序：Asset Insurance Program（负责资产的发行）。

· 普通合约进行资产的交易和分红的设置、认定。此类权限放开，每个合约相当于现实中的一个基金。如果合约中需要开发或引入一种新的资产，需要向创世合约提交请求，审核通过后方能发布到链上。普通合约的底层实现会调用数据传输层中的控制程序：Asset Management Program（负责资产的花费、交换等）。

7.4 智能合约语言

7.4.1 Equity语言

在公链的基础上增加智能合约功能才算是一条完整的公链。智能合约语言有很多种，Equity语言是比原链团队编写的（基于UTXO模型）智能合约语言。这是一种图灵完备、解释型的高级语言，可以方便地对比原链上的资产进行操作，并灵活地融入各种资产业务场景。Equity语言具体特性如下：

· Equity是谓词表述性语言，合约中每条语句的执行要么成功、要么失败。类似于解析型语言，且比其他语言简单。

· Equity是针对UTXO模型的区块链合约编译器，生成的program用于守护UTXO。与账户模型的以太坊合约相比，体积小、安全性高。

· 编译的结果program是根据合约参数变量抽象出虚拟机栈的执行流程，本质是将虚拟机执行的op按照规则组合起来的程序。

· 对编译的program指令执行流程进行检查，避免用户在直接使用指令构造program时出现不合规则的执行流程。

· 对编译的program指令程序进行优化，减少指令的执行开销（Gas消耗），例如“1 ROLL”优化为“SWAP”。

· 合约的读写简单，便于开发者编写和理解。

有些读者可能会有疑问：发布在链上的智能合约代码日积月累，会不会让区块爆炸掉？答案是不会的，因为合约代码不会存在链上，链上只存该合约交易，合约的代码存放在本地。合约代码被编译之后，其本质是指令码program，链上保存的合约指令码program标识了合约的执行流程。

7.4.2 Equity合约组成

智能合约基本组成结构如下：

```
contract  ContractName( parameter ... ) locks value {
clause  ClauseName( parameter ... ) {
    statement
  ...
}
    clause  ClauseName(parameter ... ) requires payments ... {
        statement
        ...
}
    ...
}
```

其中，requires payments是条款的支付需求。合约结构较为简单，首先定义一个合约名（ContractName），带有参数列表（parameter），value资产值，表示合约锁定的资产（即UTXO的资产类型和对应的值）。锁定资产后如何解锁，则需要用到条款函数Clause，通过Clause函数可以定义多个解锁合约的操作，所有Equity合约的解锁都通过Clause函数进行。

1.合约参数类型（parameter）

参数分为三大类，共九种。

Integer整数类型如下。

· Amount：合约中锁定资产的数量，范围$0 \sim 2^{63}-1$。

· Integer：介于$-2^{-63} \sim 2^{63}-1$之间的整数。

Boolean布尔类型如下。

· Boolean：值为true或false。

String字符串类型（为了底层数据格式的统一，均以十六进制字节串形式出现）如下。

· String：普通的字符串，经过ASCII编码得到十六进制字节串。

· Hash：哈希得到的字节串。

· Asset：资产编号。

· PublicKey：公钥。

· Signature：私钥签名后的消息。

· Program：以字节形式表示的程序码。

2. Clause函数语句（statement）

Clause函数语句，共有如下三种语句表达式：

· verify语句：验证表达式的结果是否为真。

· lock语句：将原合约中的value以及支付给条款函数的value锁定至新的合约中。

· unlock语句：解锁合约中锁定的value。

这三种语句表达式的应用将在后面章节中具体介绍。

3. 条款的支付需求（Required payments）

有时候，条款函数（clause）会要求支付一些其他的value才能解锁合约中已有的value。例如，在美元交易换取欧元的时候，就需要支付美元以解锁合约中的欧元。这种情况下，必须在clause中使用requires语法来为所需的value指定名称，并指定其数量和资产类型，形式为：

```
clause ClauseName ( parameters ) requires name : amount of
asset
```

其中：

· name：表示标识符，表示支付的value的名称。

· amount：表示Amount数据类型的表达式（expressions）。

· asset：表示Asset数据类型的表达式。

有些clause需要支付两种或更多种的value才能解锁合约。这时，可以在requires后如下代码：

```
... requires name1 : amount1 of asset1 , name2 : amount2 of
asset2 , ...
```

---

4. 运算表达式（Expression）

表达式有如下多种类型。

· binaryExpr：二元运算表达式，包含：">""<"">=""<=""==""!="^"|"+"-"&"<<">>"%"*"/"等运算符。

· unaryExpr：一元运算表达式，包含"-"~"运算符。

· callExpr：调用运算，可调用："sha3""sha256""size""abs""min""max""checkTxSig""concat""concatpush""below”"above""checkTxMultiSig"等运算。其中除“below”“above”运算以外，其他运算在虚拟机中均有对应的指令操作实现；而“below”“above”则需另外编写代码实现。

· varRef，参数引用。compileRef检查该参数被引用的次数，并调整至栈头。

· integerLiteral：integer类型数据入栈。

· bytesLiteral：byte类型数据入栈。

· booleanLiteral：boolean类型数据入栈。

· listExpr：列表表达式。该类型表达式在此仅返回信息，并不做其他处理。

5. 内置函数（function）

Equity内置了如下函数。

· abs（n）：返回数值n的绝对值。

· min（x，y）：返回数值x和y中最小的一个。

· max（x，y）：返回数值x和y中最大的一个。

· size（s）：返回任意类型的字节大小size。

· concat（s1，s2）：将两个字符串s1和s2连接起来生成新的字符串。

· concatpush（s1，s2）：将两个字符串类型的虚拟机执行操作码s1和s2连接起来（即将s2拼接在s1的后面），然后将它们push到栈中。该操作函数主要用于嵌套合约中。

· below（height）：判断当前区块高度是否低于参数height，如果是则返回true，否则返回false。

· above（height）：判断当前区块高度是否高于参数height，如果是则返回true，否则返回false。

· sha3（s）：返回byte类型字符串参数s的SHA3-256哈希运算结果。

· sha256（s）：返回byte类型字符串参数s的SHA-256哈希运算结果。

· checkTxSig（key，sig）：根据一个PublicKey和一个Signature验证交易的签名是否正确。

· checkTxMultiSig（[key1，key2，...]，[sig1，sig2，...]）：根据多个PublicKey和多个Signature验证交易的多重签名是否正确。

6. Equity合约规则（Rules for contracts）

在Equity语言中只有遵循以下规则的合约才是有效的：

· 标识符不能互相冲突。例如clause参数的名称不能与contract参数的名称相同。（两个不同的clause可以使用相同的参数名，这不构成冲突。）

· 每个contract参数必须至少在一个clause语句中被使用。

· 每个clause参数必须在clause内被使用。

· 每个clause必须使用lock或unlock语句处理合约中锁定的value。

· 每个clause还必须使用lock语句处理所有的支付给clause的value（如果有的话）。

7.5 基于UTXO模型合约开发实战

我们基于UTXO模型，使用Equity语言开发一个币币交易合约，该场景类似于用户在交易所挂单，其他用户与之交易。

通过该合约，用户不通过中心化的机构即可在比原链和其他人进行多种资产的交易。用户可以通过合约锁定部分资产，只有当其他用户打入规定数量的指定资产时，才能解锁这部分资产。资产如图7-1所示。

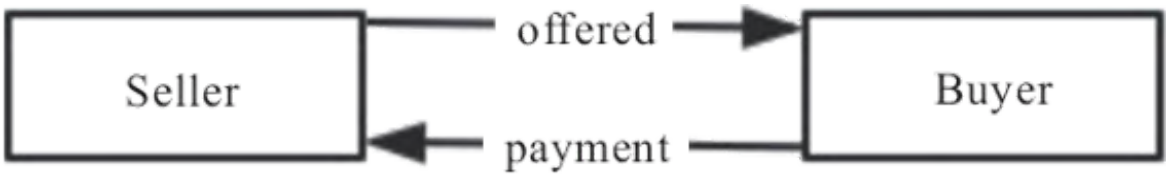


图7-1 资产示意图

Seller发起合约将自己的币locked锁定在合约中，offered即锁定的value资产，payment即Buyer所支付资产（合约所规定的assetRequested）。他的兑换条件是，只要有人将amonutRequested数量的assetRequested资产ID发给Seller，那么他就能获取到Seller锁定在合约中的代币。交易支付后将payment锁定到卖方，将offered锁定到买方。

下面，我们来编写一个交易所挂单合约，合约的发布流程如图7-2所示。

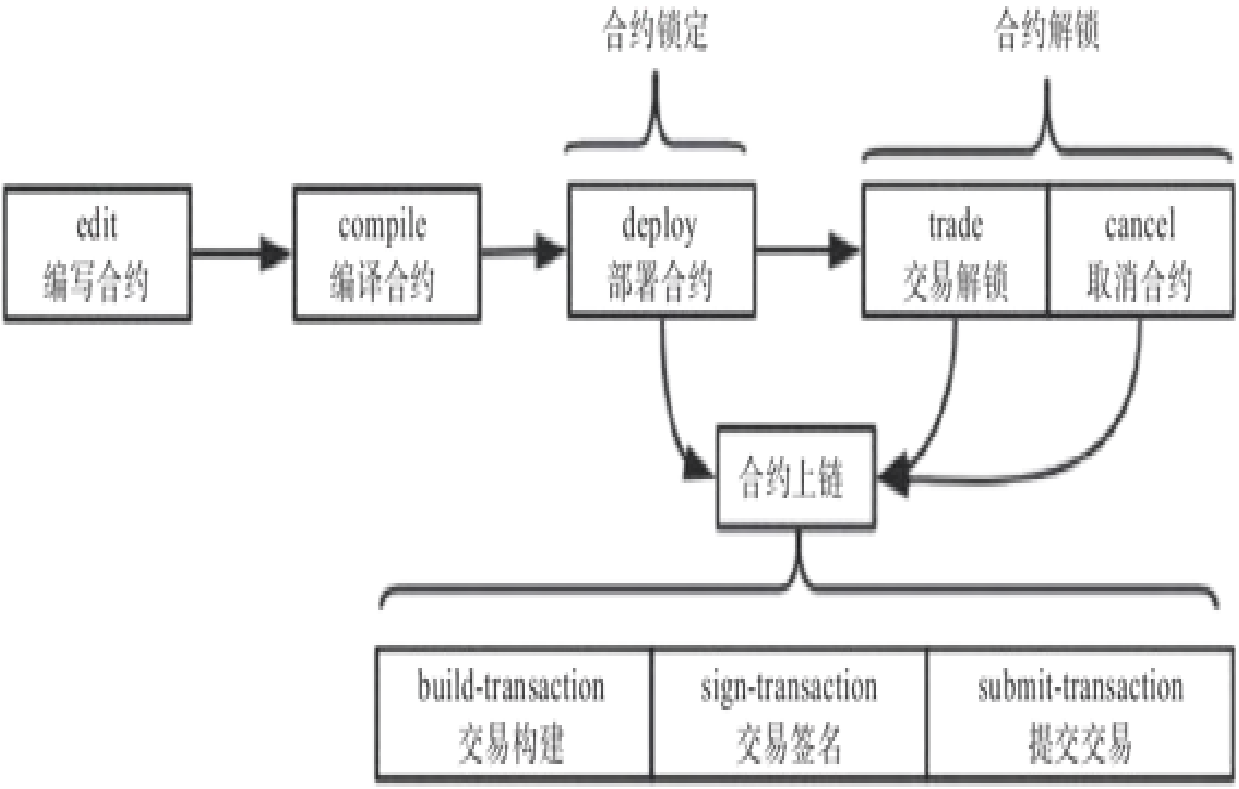


图7-2　合约的发布流程

首先，我们往链上发布两个资产，分别是Seller和Buyer：

资产别名	资产ID
Seller	4e5e65d9be0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020c
Burer	a55e72ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff

7.5.1 编写合约

交易所挂单（币币交易）合约模板如下：

```
contract TradeOffer(assetRequested: Asset,
                    amountRequested: Amount,
                    seller: Program,
                    cancelKey: PublicKey) locks offered {
  clause trade() requires payment: amountRequested of
assetRequested {
    lock payment with seller
    unlock offered
  }
  clause cancel(sellerSig: Signature) {
    verify checkTxSig(cancelKey, sellerSig)
    unlock offered
  }
}
```

在该合约模板中，合约名为TradeOffer，参数说明如下。

· assetRequested：Buyer的资产ID。

· amountRequested：Buyer的资产数量。

· seller：Seller的program，接收资产的合约脚本。

· cancelKey：Seller的公钥，用于取消合约。合约是Seller发起的，也由Seller取消。

合约锁定资产offered，有两个clause函数，即有两种途径解锁合约。

· trade：交易解锁。使用requires语法，支付后将payment锁定到卖方，将offered锁定到买方。此合约中不指定唯一买方，任意用户均可进行交易，使用unlock语句将对应资产锁定至交易用户，合约解锁。

· cancel：取消解锁。合约可以取消，由卖方提供签名，使用verify语句验证卖方签名sellersig和cancelKey是否一致，若一致，则返回true，归还offered资产到卖方，合约解锁。

7.5.2 编译合约

1.创建control_program

program表示接收资产的合约脚本。有两种方式可以创建control_program，第一种方式请求/list-addresses直接查询control_program。第二种方式请求/create-account-receiver新建地址，但会增加地址的个数。本节我们使用第二种方式。

根据/create-account-receiver API接口创建program（返回结果的program和address是一一对应的）。

调用接口：/create-account-receiver

POST传参：

```
{"account_id":"0DCKJ2B2G0A02"}
```

响应结果：

```
{
    "status": "success",
    "data": {
        "control_program":
"00149c31f762ad902dbee9565fad8b74ca4785c0af99",
        "address": "bm1qnsclwc4djqkma62kt7kckax2g7zuptuec052dt"
    }
}
```

字段说明如下。

· control_program：接收资产的合约脚本。如果有人给他人的账号转代币或者其他资产，都是通过control_program转，也就是此合约的seller参数。

· address：该合约下所对应的卖方账户地址。

2. 编译合约

compile接口需指定两个字段：contract为合约代码，args为合约的传参。

币币合约中接收四个参数，我们指定assetRequested参数为Buyer的资产ID。挂单的金额amountRequested为100。其中seller参数是从/create-account-receiver接口获得的control_program数据。cancelKey参数为Seller卖方公钥，可以从/list-pubkeys接口中获取公钥信息。

调用接口：/compile

POST传参：

```
{
    "args":[

{"string":"a55e72ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9
d3f8bd459bff"},
        {"integer":100},

{"string":"00149c31f762ad902dbee9565fad8b74ca4785c0af99"},

{"string":"ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e5762
4164b7d520f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d6
89ef99c36e1fd"}
    ],
    "contract":"contract TradeOffer(assetRequested:
Asset,amountRequested: Amount,seller: Program,cancelKey:
PublicKey) locks offered { clause trade() requires payment:
amountRequested of assetRequested { lock payment with seller
unlock offered } clause cancel(sellerSig: Signature) { verify
checkTxSig(cancelKey, sellerSig) unlock offered } }"
}
```

响应结果：

```
{
    "status": "success",
```

```
    "data": {
        "name": "TradeOffer",
        // ...
        "program":
"40ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d5
20f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c3
6e1fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72
ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a5
47a6413000000007b7b51547ac1631a000000547a547aae7cac00c0",
        // ...
    }
}
```

接口返回success，返回信息中注意记录program项，随后将在创建交易时使用。至此，智能合约编译完成。

注意

在合约上Seller和Buyer交易的是资产数量（amount），BTM本质上作为Gas费用，所以合约锁定的资产ID不能是默认的资产ID（即ffffff...ffffff），若锁定的资产ID为默认，在提交交易时会返回错误“BTM733：非标准交易”。在使用智能合约前，首先要发布其他类型资产。比如我们上面发布了Seller和Buyer的两种资产。如果要想资产数量（amount）和BTM代币挂钩，需要在上层dapp上实现兑换逻辑。

7.5.3 部署合约

合约部署需要经过三个过程：交易构建，交易签名，交易提交到主链。

1.build-transaction

将program传入/build-transaction接口创建一个交易，得到编码后的raw_transaction数据。

POST请求参数中action是一个长度为3的数组，包括：

· action下标为0的元素，表示支付此次交易Gas手续费的账户。手续费自定义为0.02BTM，类型为spend_account。

· action下标为1的元素，表示锁定资产到该合约中的账户，数量自定义为100，asset_id为要锁定资产ID，类型为spend_account。

· action下标为2的元素，表示编译后的合约control_program程序码，类型为control_program。

注意

asset_id都是Seller的资产ID，用于锁定Seller的资产。

调用接口：/build-transaction

POST传参：

```
{
    "base_transaction": null,
    "actions": [
        {
            "account_id": "0DCKJ2B2G0A02",
            "amount": 2000000,
            "asset_id":
"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
```

```
            "type": "spend_account"
        },
        {
            "account_id": "0DCKJ2B2G0A02",
            "amount": 100,
            "asset_id":
"4e5e65d9be0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd02
0c",
            "type": "spend_account"
        },
        {
            "amount": 100,
            "asset_id":
"4e5e65d9be0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd02
0c",
            "control_program":
"40ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d5
20f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c3
6e1fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72
ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a5
47a6413000000007b7b51547ac1631a000000547a547aae7cac00c0",
            "type": "control_program"
        }
    ],
    "ttl": 0,
    "time_range": 0
}
```

响应结果：

```
{
    "status": "success",
    "data": {
        "raw_transaction":
"07010002015f015df371cef0c5a0a17419fa7ff11784e86ef59608821a3faf
e6ed14a39b540ec414ffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffe0e8f011000116001419a45f8834de728571adfffc54
a626ddd3744c6201000160015ef371cef0c5a0a17419fa7ff11784e86ef5960
8821a3fafe6ed14a39b540ec4144e5e65d9be0d5dfc937d70bf6990b7c9d650
4002a1c1d8b902a72176b8dd020cd4cf93ad030101160014801dec8cca6c25a
f20f7f720260ff650c8f10e98010003013cffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffe0dff61001160014b6e3d4c9bb5
346246634fcd0923a82add4f1f7a000013d4e5e65d9be0d5dfc937d70bf6990
b7c9d6504002a1c1d8b902a72176b8dd020cf0ce93ad0301160014eed16e73f
bf70104b40228f403b21368bb4056bd0001bd014e5e65d9be0d5dfc937d70bf
```

```
6990b7c9d6504002a1c1d8b902a72176b8dd020c6401990140ba7babfdbe9f3
0e5f3df47a2ec960629840e360a5daf624e57624164b7d520f32cddf7e748b9
6cac771963b2fa46a30a672a7d9145828b7e3d689ef99c36e1fd1600149c31f
762ad902dbee9565fad8b74ca4785c0af99016420a55e72ad9b9040dbca1d0c
898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a547a6413000000007
b7b51547ac1631a000000547a547aae7cac00c000"
    }
    // ...
}
```

其中，raw_transaction为构建后的交易数据，在下一步签名中需要用到。

2.sign-transaction

接口/sign-transaction对上一步操作返回的raw_transaction进行签名。在请求参数中要求输入password密码。

调用接口：/sign-transaction

POST传参：

```
{
    "password": "xxxxxx",
    "transaction": {
        "raw_transaction":
"07010002015f015df371cef0c5a0a17419fa7ff11784e86ef59608821a3faf
e6ed14a39b540ec414ffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffe0e8f011000116001419a45f8834de728571adfffc54
a626ddd3744c6201000160015ef371cef0c5a0a17419fa7ff11784e86ef5960
8821a3fafe6ed14a39b540ec4144e5e65d9be0d5dfc937d70bf6990b7c9d650
4002a1c1d8b902a72176b8dd020cd4cf93ad030101160014801dec8cca6c25a
f20f7f720260ff650c8f10e98010003013cffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffe0dff61001160014b6e3d4c9bb5
346246634fcd0923a82add4f1f7a000013d4e5e65d9be0d5dfc937d70bf6990
b7c9d6504002a1c1d8b902a72176b8dd020cf0ce93ad0301160014eed16e73f
bf70104b40228f403b21368bb4056bd0001bd014e5e65d9be0d5dfc937d70bf
6990b7c9d6504002a1c1d8b902a72176b8dd020c6401990140ba7babfdbe9f3
0e5f3df47a2ec960629840e360a5daf624e57624164b7d520f32cddf7e748b9
6cac771963b2fa46a30a672a7d9145828b7e3d689ef99c36e1fd1600149c31f
762ad902dbee9565fad8b74ca4785c0af99016420a55e72ad9b9040dbca1d0c
898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a547a6413000000007
b7b51547ac1631a000000547a547aae7cac00c000"
```

```
    }
    // ...
}
```

响应结果：

```
{
    "status": "success",
    "data": {
        "transaction": {
            "raw_transaction":
"07010002015f015df371cef0c5a0a17419fa7ff11784e86ef59608821a3faf
e6ed14a39b540ec414ffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffe0e8f011000116001419a45f8834de728571adfffc54
a626ddd3744c626302400197c8d4101f52ae86f6c624b8884d5282e485ff027
088dc38a60be92461814fe901b542db4662f184fd8e471ba9211b658ea63ca7
50049b30d51464d95e550b2059e7c346a4c6e3fd91d82ce91df637af293b00f
04b1d679e4c3b15689fd9513d0160015ef371cef0c5a0a17419fa7ff11784e8
6ef59608821a3fafe6ed14a39b540ec4144e5e65d9be0d5dfc937d70bf6990b
7c9d6504002a1c1d8b902a72176b8dd020cd4cf93ad030101160014801dec8c
ca6c25af20f7f720260ff650c8f10e98630240a5617d102381bcbb0d4df9726
bdedc67e2c6e37db6bb6298d65ae9a203e817f8c8a2a1f7a74f474a6d41d694
46891a44205936d726db10dd7e68441c13f3b70720bf63624cfaf213d2e817c
0108f91bef49fc6b438231dfcff54e4d67885f6e20503013cfffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffe0dff61001160
014b6e3d4c9bb5346246634fcd0923a82add4f1f7a000013d4e5e65d9be0d5d
fc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020cf0ce93ad03011
60014eed16e73fbf70104b40228f403b21368bb4056bd0001bd014e5e65d9be
0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020c640199014
0ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d520
f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c36e
1fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72ad
9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a547
a6413000000007b7b51547ac1631a000000547a547aae7cac00c000"
            // ...
        }
    }
}
```

其中，raw_transaction为返回签名后的交易信息。我们可以看到，签名后内容变长了，添加上了signatures相关字段。

3. submit-transactions

获取sign-transaction签名成功返回的raw_transaction字段数据，提交交易。

调用接口：/submit-transactions

POST传参：

```
{
  "raw_transaction":
"07010002015f015df371cef0c5a0a17419fa7ff11784e86ef59608821a3faf
e6ed14a39b540ec414ffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffe0e8f011000116001419a45f8834de728571adfffc54
a626ddd3744c626302400197c8d4101f52ae86f6c624b8884d5282e485ff027
088dc38a60be92461814fe901b542db4662f184fd8e471ba9211b658ea63ca7
50049b30d51464d95e550b2059e7c346a4c6e3fd91d82ce91df637af293b00f
04b1d679e4c3b15689fd9513d0160015ef371cef0c5a0a17419fa7ff11784e8
6ef59608821a3fafe6ed14a39b540ec4144e5e65d9be0d5dfc937d70bf6990b
7c9d6504002a1c1d8b902a72176b8dd020cd4cf93ad030101160014801dec8c
ca6c25af20f7f720260ff650c8f10e98630240a5617d102381bcbb0d4df9726
bdedc67e2c6e37db6bb6298d65ae9a203e817f8c8a2a1f7a74f474a6d41d694
46891a44205936d726db10dd7e68441c13f3b70720bf63624cfaf213d2e817c
0108f91bef49fc6b438231dfcff54e4d67885f6e20503013cfffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffe0dff61001160
014b6e3d4c9bb5346246634fcd0923a82add4f1f7a000013d4e5e65d9be0d5d
fc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020cf0ce93ad03011
60014eed16e73fbf70104b40228f403b21368bb4056bd0001bd014e5e65d9be
0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020c640199014
0ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d520
f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c36e
1fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72ad
9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a547
a6413000000007b7b51547ac1631a000000547a547aae7cac00c000"}
```

响应结果：

```
{
    "status": "success",
    "data": {
        "tx_id":
"5c705547d2627e173c517e004c9890fef8f108718a2ff2b2fe8521c2246a8b
74"
```

```
    }
}
```

其中，tx_id为提交成功后返回交易哈希值。可以通过get-transaction获取该交易的详情。

至此，合约部署完成，Seller资产已被锁在合约中，等待解锁合约，解锁成功的人可以获得该资产。

7.5.4　解锁合约

解锁合约（交易解锁/取消合约）有两种方式：用buyer进行交易解锁合约，或用seller取消合约。首先根据7.5.4节得到的tx_id来获得outputID，该id用来解锁合约，此外还需要通过decode-program获取参数偏移量来选择使用哪个clause方式解锁。

1.获取outputID

调用接口/get-transaction。在返回的数据中找到output数组，找到compile编译合约时对应的control_program字段，从而获得outputID。

调用接口：/get-transaction

POST传参：

```
{"tx_id":"5c705547d2627e173c517e004c9890fef8f108718a2ff2b2fe8521c2246a8b74"}
```

响应结果：

```
{
    // ...
    "outputs": [
        {
            "type": "control",
            "id":
"9cea7237d6dd34abf5d2bfdc98a08b8be2dd73fb2a4c4c6b5dd8e91fd587e2df",
            "position": 2,
            "asset_id":
"4e5e65d9be0d5dfc937d70bf6990b7c9d6504002a1c1d8b902a72176b8dd020c",
            "asset_alias": "SELLER",
            "amount": 100,
            "control_program":
"40ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d520f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c3
```

```
6e1fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72
ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a5
47a6413000000007b7b51547ac1631a000000547a547aae7cac00c0"
        }
    // ...
    ]
}
```

结果显示获得了该合约control_program下所对应的outputID：9cea7237d6dd34abf5d2bfdc98a08b8be2dd73fb2a4c4c6b5dd8e91fd587e2df

2. 获取clause偏移量

在我们上面定义的币币合约当中，包含两个clause，在解锁合约的时候需要指定某一个clause来解锁合约。其中clause的选择是由合约代码位置参数偏移量来决定的，通过decode-program可以查看合约对应的偏移量。如果偏移量指定错误则VM执行失败。

该合约中有两个clause，第一个clause偏移量默认为00000000。第二个clause偏移量需要通过/compile接口获取program，再通过/decode-program接口获取clause偏移量。示例如下：

获得program调用接口：/compile

POST传参：

```
{
    "contract":"contract TradeOffer(assetRequested:
Asset,amountRequested: Amount,seller: Program,cancelKey:
PublicKey) locks offered { clause trade() requires payment:
amountRequested of assetRequested { lock payment with seller
unlock offered } clause cancel(sellerSig: Signature) { verify
checkTxSig(cancelKey, sellerSig) unlock offered } }"
}
```

响应结果：

```
{
// ...
    "program":
"547a6413000000007b7b51547ac1631a000000547a547aae7cac",
// ...
}
```

获得clause偏移量调用接口：/decode-program

POST传参：

```
{"program":"547a6413000000007b7b51547ac1631a000000547a547aae7ca
c"}
```

响应结果：

```
{
// ...
    "instructions": "4 04\nROLL \nJUMPIF 13000000\nFALSE \nROT
\nROT \n1 01\n4 04\nROLL \nCHECKOUTPUT \nJUMP 1a000000\n4
04\nROLL \n4 04\nROLL \nTXSIGHASH \nSWAP \nCHECKSIG \n"
// ...
}
```

从decode-program接口返回的数据中可以看到，JUMPIF虚拟机指令返回的值为13000000，该值表示第二个clause的偏移位置。

3. 交易解锁

交易解锁合约也是发起交易，需要重新构建、签名、提交交易。与发布合约不同的是交易解锁不需要编译。在交易解锁合约时build过程与发起合约时的build参数不同。

POST请求参数中action下标为2的元素，参数说明如下。

- output_id：/get-transaction接口返回的outputID。

· argument：合约参数，合约参数中value为clause偏移量。在合约有多个clause时，用来选择某一个clause来执行。合约中每一个clause都对应一个固定的value，这里选取第一个clause解锁方式，偏移量为0，即value为00000000。

· type：指定类型spend_account_unspent_output。

POST请求参数中action下标为3的元素，参数为control_program：解锁合约后资产转入的地址。

调用接口：/build-transaction

POST传参：

```
{
    // ...
    {
        "output_id":
"9cea7237d6dd34abf5d2bfdc98a08b8be2dd73fb2a4c4c6b5dd8e91fd587e2
df",
        "arguments": [
            {
                "type":"data",
                "raw_data":{"value":"00000000"}
            }
        ],
        "type": "spend_account_unspent_output"
    },
    // ...
}
```

响应结果：

```
{
    "status": "success",
    // ...
}
```

接口返回success。此后/sign-transaction和/submit-transaction步骤与发布合约时相同，不再赘述。最终我们在submit提交时正确返回了tx_id，此时合约成功解锁。

4. 取消合约

取消合约由Seller主动取消合约，操作过程同交易解锁相似，仅需要对argument内容进行改动。提供derivation_path（每一个密钥的生成路径）和xpub（可通过/list-pubkeys接口查看），这里选择第二个clause解锁方法，对应参数偏移量value为13000000。

调用接口：/build-transaction

POST传参：

```
{
// ...
    {
        "output_id":
"9cea7237d6dd34abf5d2bfdc98a08b8be2dd73fb2a4c4c6b5dd8e91fd587e2
df",
        "arguments": [
            {
                "type":"raw_tx_signature",
                "raw_data":{
                    "derivation_path":
["0101000000000000000","0100000000000000"],

"xpub":"ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e5762416
4b7d520f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689e
f99c36e1fd"
                }
            },
            {
                "type":"data",
                "raw_data":{"value":"13000000"}
            }
        ],
        "type": "spend_account_unspent_output"
    },
// ...
}
```

响应结果：

```
{
    "status": "success",
    // ...
}
```

此后sign-transaction和submit-transaction与发布合约步骤相同，不再赘述。

7.6 本章小结

对于用户而言，智能合约的使用是必不可少的。同时，基于比原链设计的特性与智能合约相结合，可衍生出多种应用场景，例如，收益权资产管理，非上市公司股权管理，证券化资产管理等。

本章基于UTXO模型下的合约开发，阐述了具体化的合约内容与操作，展现了智能合约的完整操作周期。此外，合约层的另一组成部分——虚拟机，是真正运行智能合约的环境。在下一章节中，我们将详细讲解BVM虚拟机运行原理、源代码执行过程，以及抽象化智能合约的完整运行周期。

第8章

内核层：虚拟机

8.1 概述

区块链上的虚拟机（Virtual Machine）是指建立在去中心化的区块链上的代码运行环境。目前市面上比较主流的是比特币脚本引擎和以太坊虚拟机（Ethereum Virtual Machine，EVM）。EVM基于Account账户模型将智能合约代码以对外完全隔离的方式内部运行，实现了图灵完备的智能合约体系。比特币交易由一套脚本引擎（Script）处理，Script是一种类Forth的、基于栈式模型的、无状态的、非图灵完备的语言。

比原链的虚拟机与比特币类似，最大的区别在于，比原链在基于UTXO模型下实现了图灵完备的智能合约体系。比原链在UTXO模型上支持了runlimit机制来限制陷入死循环。

比原链虚拟机（Bytom Virtual Machine，BVM）主要作用是处理比原链系统内的智能合约。BVM是比原链中非常重要的部分，在智能合约存储、执行和验证过程中担当着重要角色。

BVM使用Equity语言来编写智能合约，比原链是一个点对点的网络，每一个节点都运行着BVM，并执行相同的指令。BVM是在验证交易时运行的，主要是为了验证交易的有效性。

本章主要内容如下：

· 虚拟机特性。

· 虚拟机实现原理。

· BVM操作指令集。

· 智能合约在BVM上的运行过程。

8.2 BVM介绍

我们在第7章中介绍了Equity语言，在比原链中使用Equity语言编写合约代码，通过API接口编译合约代码并生成对应的字节码后，合约以字节码的形式运行在BVM中。合约的运行如图8-1所示。

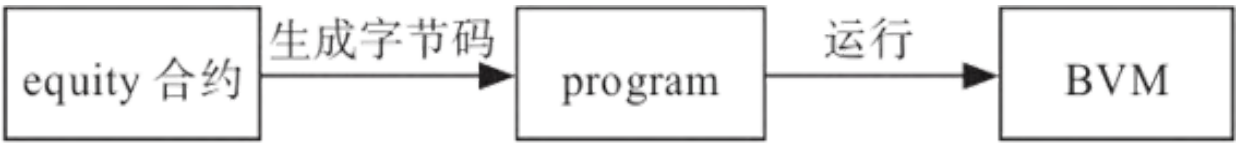


图8-1 合约的运行

8.2.1 虚拟机的栈

首先要明确“栈”的概念。栈是一种先进后出（first-in/last-out，FILO）的数据结构。在比原链中，虚拟机就是一个栈式状态机，任何指令都在该栈中执行。下面举三个简单的例子来理解栈的操作。

1. 入栈操作

栈中自顶向下原有y、x元素，将常数2入栈后，栈中元素为2、y、x，如图8-2所示。

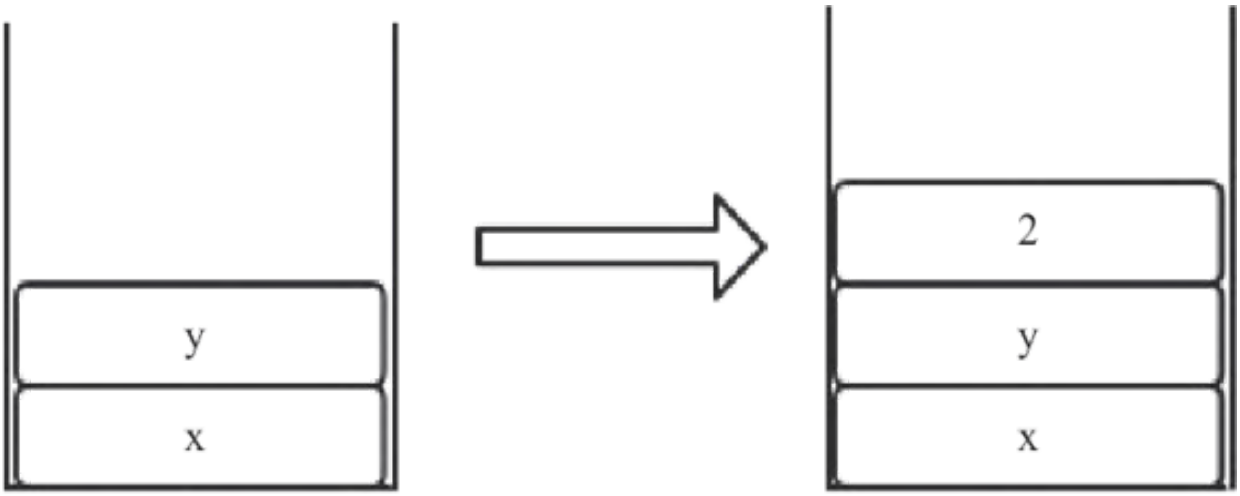


图8-2 入栈操作

2. 出栈操作

栈中自顶向下依次有2、y、x元素，出栈操作即栈顶元素2弹出，栈中元素为y、x，如图8-3所示。

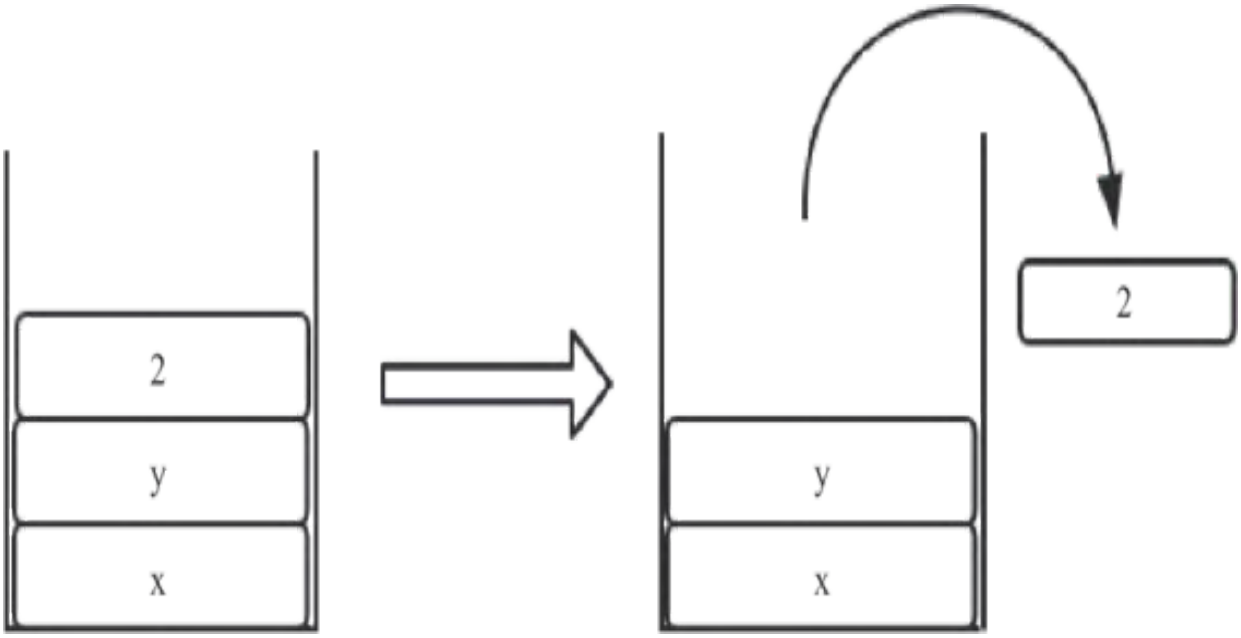


图8-3　出栈操作

3. 执行OP_DUP指令（复制栈顶元素）

栈中自顶向下有2、x元素，执行OP_DUP指令后，栈中元素有2、2、x，如图8-4所示。OP_DUP指令操作是将栈顶元素复制一个，然后压入栈中。

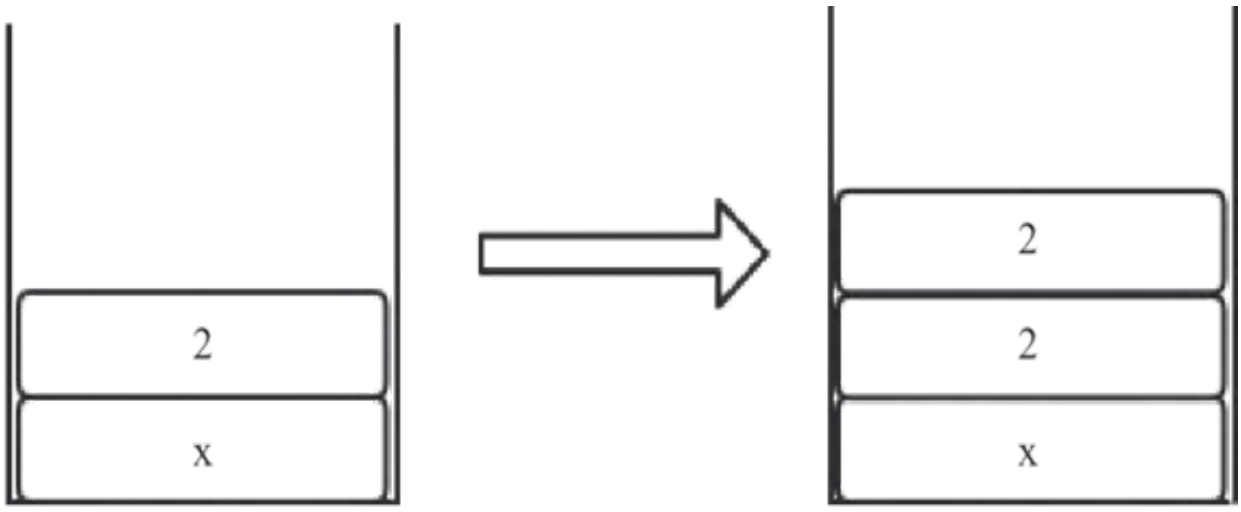


图8-4　执行OP_DUP指令

8.2.2　具有图灵完备性的BVM

在7.2.2节中，我们介绍了什么是图灵完备性的智能合约。一个语言是否是图灵完备的，要看该语言是否支持分支语句、循环语句、递归、各种算法等，如果都支持则说明该语言是图灵完备。合约的生命周期包括：条件触发，限制循环或递归，让合约的生命周期结束。要保证系统的每段程序都不会陷入死循环，都会有运行结束的时候。

防止系统因陷入死循环而导致程序崩溃，在UTXO模型下是比较困难的；而在具有图灵完备的比原链中，可采用runlimit机制来限制陷入死循环，协议允许网络达成共识来设置该runlimit。根据运行消耗来定价每一条指令的runlimit，这非常类似于以太坊的Gas机制，与以太坊不同的是，BVM不会在每笔交易中给runlimit计费。

8.2.3 equity&vm代码结构

equity代码目录如下（此处省略test文件）：

```
$ tree equity/
equity/
├── compiler
│   ├── ast.go                抽象语法树
│   ├── builder.go            生成栈
│   ├── builtins.go           表达式运算
│   ├── checks.go             检查合约内容
│   ├── cmd                   运行文件
│   ├── compile.go            合约编译函数
│   ├── doc.go                描述文档
│   ├── environ.go            数据绑定
│   ├── equitytest
│   │   └── equitytest.go   合约模板
│   ├── optimize.go           优化opcode操作指令
│   ├── parse.go              合约解析函数
│   ├── stack.go              栈操作
│   └── types.go              类型描述
└── equity.go
```

vm代码目录如下（此处省略test文件）：

```
$ tree protocol/vm/
protocol/vm/
├── assemble.go               指令操作与字节码转换
├── bitwise.go                位操作
├── context.go                VM执行上下文
├── control.go                控制流程操作
├── crypto.go                 加密和哈希操作
├── errors.go                 VM错误码
├── introspection.go          VM检查操作
├── numeric.go                数值操作
├── ops.go                    OP指令集及码字
├── pushdata.go               入栈操作
├── splice.go                 字符串处理操作
├── stack.go                  字符串处理操作
```

```
├── types.go                    码字类型
├── vm.go                       VM执行代码
└── vmutil
    ├── builder.go              生成Builder结构体
    ├── script.go               脚本相关函数
```

8.3 virtualMachine对象

virtualMachine是BVM虚拟机对象，所有合约代码执行过程都经过BVM进行解析并运行。

```
protocol/vm/vm.go
type virtualMachine struct {
    context *Context
    program      []byte
    pc, nextPC   uint32
    runLimit     int64
    deferredCost int64
    expansionReserved bool
    data []byte
    depth int
    dataStack [][]byte
    altStack  [][]byte
}
```

virtualMachine对象注解如下。

- context：虚拟机的执行上下文。
- program：程序字节码。
- pc，nextPC：计数器，用来确定program字节码的处理位置。
- runLimit：虚拟机操作总花费限制，防止死循环。
- deferredCost：某次指令操作失败的花费，初始值设定为0。

· expansionReserved：版本扩展预留，目前交易版本值为1。

· data：存储从opcode解析得到的数据。

· depth：记录CHECKPREDICATE所产生子虚拟机的个数。

· dataStack：主栈，存放program数据。

· altStack：辅栈。

virtualMachine对象函数实现注解如下。

· func Verify()：虚拟机运行入口，检验program。

· func falseResult()：检查栈是否为空或栈顶元素是否错误。

· func run()：运行虚拟机，通过step()函数处理program。

· func step()：解析运行program对应字段。

· func push()：对dataStack操作，将数据入栈。

· func pushBool()：调用push()，将Boolean类型数据入栈。

· func pushInt64()：调用push()，将Int64类型数据入栈。

· func pop()：将dataStack栈顶元素弹出，并返回。

· func opInt64()：将dataStack栈顶元素弹出，转化为Int64类型并返回。

· func top()：返回dataStake栈顶元素。

· func applyCost()：判断指令操作花费是否满足runlimit。

· func deferCost()：设定操作失败所需花费。

· func stackCost()：返回某栈所用容量。

8.4 栈实现

栈（stack）功能数据结构如图8-5所示。

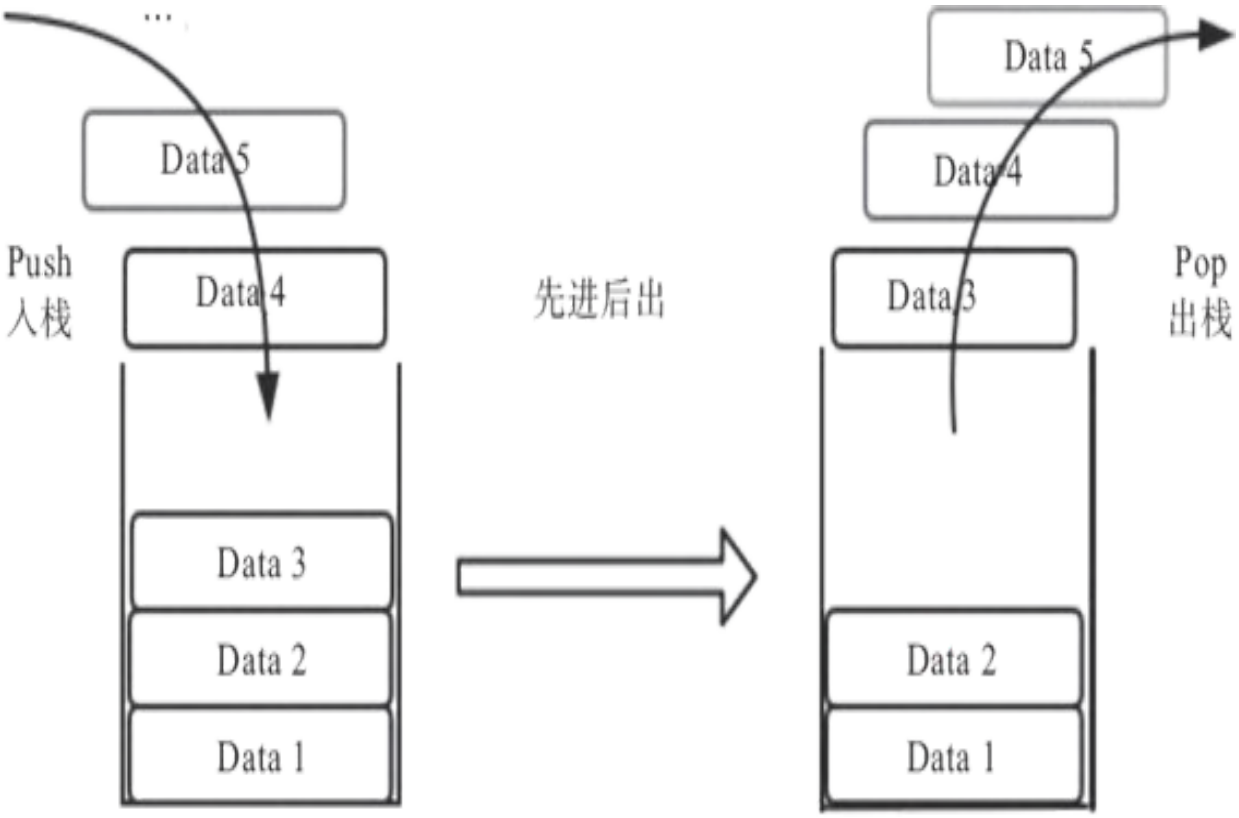


图8-5 栈数据结构

栈是一种线性存储结构，向栈中添加/删除数据时，只能从栈顶进行操作。

栈包括三种基本的操作。

· add：向栈中添加元素。

· top：返回栈顶元素。

· pop：返回并删除栈顶元素。

定义一个栈结构的代码示例如下：

```
equity/compiler/stack.go
type (
    stack struct {
        *stackEntry
    }
    stackEntry struct {
        str  string
        prev *stackEntry
    }
)
```

通过链表方式实现栈，stack结构体包含*stackEntry。在stackEntry结构体中，str为栈顶元素，prev指向前一元素。

栈实现方法如下。

· func isEmpty()：检验栈是否为空。

· func top()：返回栈顶元素。

· func add()：将元素入栈。

· func addFromStack()：将栈other添加到栈stk中，元素顺序不发生改变。

· func drop()：删除栈顶一个元素。

· func dropN()：删除栈顶N个元素。

· func find()：在栈中寻找某一元素。

· func roll()：将栈中第n个元素移动到栈顶。

· func swap()：交换栈顶两元素。

· func dup()：复制栈顶元素并入栈。

· func over()：复制栈顶元素的下一个元素并入栈。

· func pick()：复制栈中第n个元素并入栈。

· func String()：解析字符串，将内容依次入栈。

1. add，将元素入栈

代码如下：

```
func (stk stack) add(str string) stack {
  e := &stackEntry{
     str:  str,
     prev: stk.stackEntry,
  }
  return stack{e}
}
```

传入栈的元素str和当前栈顶元素的地址作为一个新的stackEntry类型变量e，并返回，实现str元素的入栈。

2. addFromStack，添加另一个栈

addFromStack操作如图8-6所示。

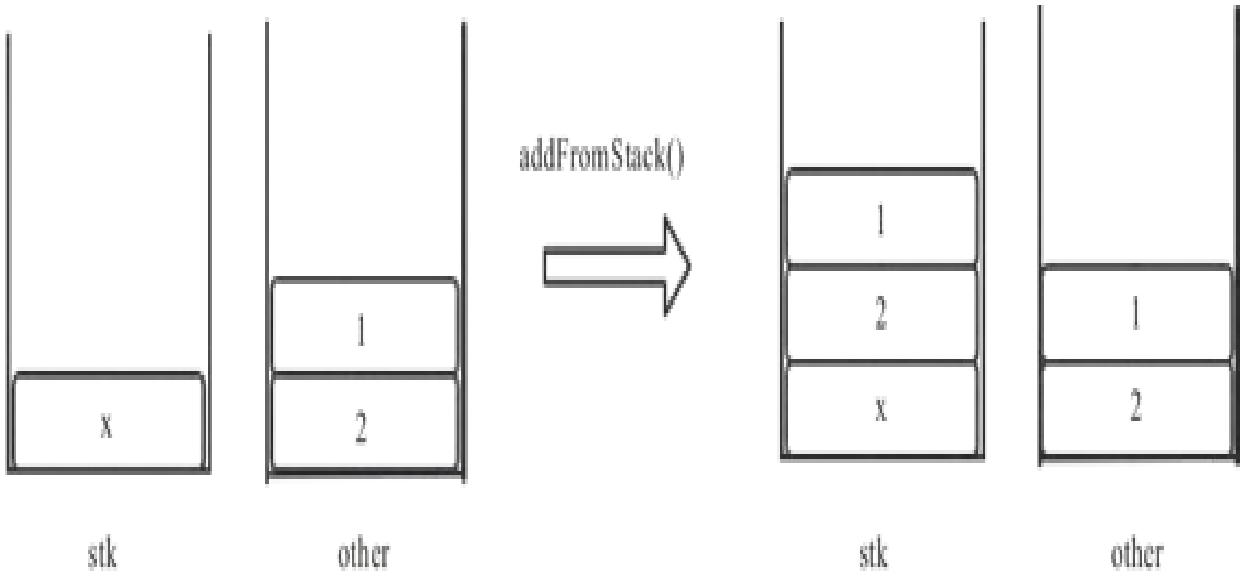


图8-6　addFromStack操作

代码如下：

```
func (stk stack) addFromStack(other stack) stack {
    if other.isEmpty() {
        return stk
    }
    res := stk.addFromStack(other.drop())
    return res.add(other.top())
}
```

该函数为递归调用，调用过程中不断删除other栈的栈顶元素，直至other栈为空，最后从栈底依次添加元素至stk栈。函数作用结果即将other栈复制添加到stk栈中。

8.5 BVM操作指令集

BVM操作指令集包含了很多指令，能支持更丰富的逻辑功能。BVM包含push、pop等基本指令集，同时还包括SHA3，CHECKSIG、CHECKMULTISIG等复杂的数学加密运算。

BVM中所有操作指令定义都在protocol/vm/ops.go文件中。每一个指令对应一个字节码。比如，OP_2 Op=0x52，将数字2入栈，该操作对应字节码为52。再比如，OP_DATA_1 Op=0x01，将一个字节的数据入栈，该操作对应的字节码为01。另外较为复杂的操作指令会有对应的函数实现。

本小节我们以OP_ADD操作指令为例，展示BVM操作指令在虚拟机中执行原理及过程。OP_ADD的操作是返回两个输入值之和，指令示例如下：

```
OP_ADD    Op = 0x93
```

OP_ADD指令在合约编译的程序码中对应93。该指令对应的函数实现如下：

```
OP_ADD:      {OP_ADD, "ADD", opAdd},
```

OP_ADD指令名为“ADD”，通过opAdd()函数实现：

```
protocol/vm/numeric.go
func opAdd(vm *virtualMachine) error {
```

```
    err := vm.applyCost(2)
    if err != nil {
        return err
    }

    y, err := vm.popInt64(true)
    if err != nil {
        return err
    }
    x, err := vm.popInt64(true)
    if err != nil {
        return err
    }

    res, ok := checked.AddInt64(x, y)
    if !ok {
        return ErrRange
    }

    return vm.pushInt64(res, true)
}
```

OP_ADD指令在虚拟机栈式存储下，首先将栈顶两个元素弹出，进行相加后将结果入栈，操作完成。执行流程如下。

1）vm.applyCost：判断runlimit是否满足该指令的花费。

2）vm.popInt64：将栈顶两个元素（y和x）弹出。

3）checked.AddInt64：将两个元素相加。

4）vm.pushInt64：将相加结果入栈，opAdd指令操作完成。

函数中，vm.applyCost判断runlimit是否满足该指令的花费。原理如下：

```
protocol/vm/vm.go
func (vm *virtualMachine) applyCost(n int64) error {
    if n > vm.runLimit {
        vm.runLimit = 0
        return ErrRunLimitExceeded
    }
```

```
    vm.runLimit -= n
    return nil
}
```

虚拟机中使用不同的运算指令所对应的花费不同。比如，这里的OP_ADD一次加操作固定耗费为2，vm.applyCost（2）会先判断runlimit是否满足该花费2。若满足，runlimit将减去2，完成加操作；若不满足则函数返回错误。

8.6 智能合约在BVM上的运行过程

合约进行编译时产生可在虚拟机上运行的程序字节码，在随后的交易中，虚拟机读取程序字节码，转化为对应的操作指令，完成合约交易。本节将展示智能合约在BVM上的运行过程及完整的生命周期，如图8-7所示。

图8-7 智能合约的生命周期

8.6.1 智能合约数据结构

在讲解智能合约在BVM上的运行过程之前，我们看一下智能合约的结构。

1. 合约结构

合约结构如下：

```
equity/compiler/ast.go
type Contract struct {
    Name string `json:"name"`
    Params []*Param `json:"params,omitempty"`
    Clauses []*Clause `json:"clauses"`
    Value string `json:"value"`
    Body chainjson.HexBytes `json:"body_bytecode"`
    Opcodes string `json:"body_opcodes,omitempty"`
    Recursive bool `json:"recursive"`
    Steps []Step `json:"-"`
}
```

合约结构字段说明如下。

· Name：合约名称。

· Params：合约参数列表。

· Clauses：合约条款清单。

· Value：合约锁定资产的名称。

· Body：合约对应的已优化字节码，但不是完整的程序字节码，需要再经过instantiate实例化。

· Opcodes：与body相对应的人类可读的字符串。

· Recursive：合约是否自我调用（递归操作，自己调用自己）。

· Steps：未优化（optimize）的指令列表和指令对应的栈信息。

2. 合约参数结构

参数结构如下：

```
type Param struct {
    Name string `json:"name"`
    Type typeDesc `json:"type"`
    InferredType typeDesc `json:"inferred_type,omitempty"`
}
```

合约参数结构字段说明如下。

· Name：参数名称。

· Type：参数类型。

· InferredType：更精确的参数类型，可为空。

3. 合约条款结构

合约条款结构如下：

```
type Clause struct {
    Name string `json:"name"`
    Params []*Param `json:"params,omitempty"`
    Reqs []*ClauseReq `json:"reqs,omitempty"`
    statements []statement
    BlockHeight []string `json:"blockheight,omitempty"`
    HashCalls []HashCall `json:"hash_calls,omitempty"`
    Values []ValueInfo `json:"values"`
    Contracts []string `json:"contracts,omitempty"`
}
```

合约条款结构字段说明如下。

· Name：条款函数名称。

· Params：条款参数列表。

· Reqs：支付需求列表。

· statements：函数语句。

· BlockHeight：块高度列表。

· HashCalls：条款函数所需的哈希函数和参数列表。

· Values：条款函数解锁或再次锁定的资产列表。

· Contracts：条款函数所调用的合约列表。

8.6.2 合约编译流程与原理

通过调用API的/compile接口，对合约代码进行compiler.Compile()编译，得到Contract.Body后，经过compiler.Instantiate()实例化生成program（合约程序字节码）。代码执行流程如图8-8所示。

仅通过图8-8并无法直观地追溯合约编译的过程，我们还可以借助其他模型（如“公司处理工作的层级关系”）来理解合约编译的流程（如图8-9所示）：公司处理某项工作，首先将工作分配到所需要的部门（API Server接口调用）；进而该部门负责人对整个工作内容进行分析处理（compiler.Compile()作用）；分析完成后将每部分工作交付给对应的项目负责人，项目负责人再对每项工作进行任务拆分（compileContract()作用）；把分拆后的工作交付给小组负责人，小组负责人将任务细化（compileClause()作用），最终具体分配给每一个人去完成（compileExpr()作用）。

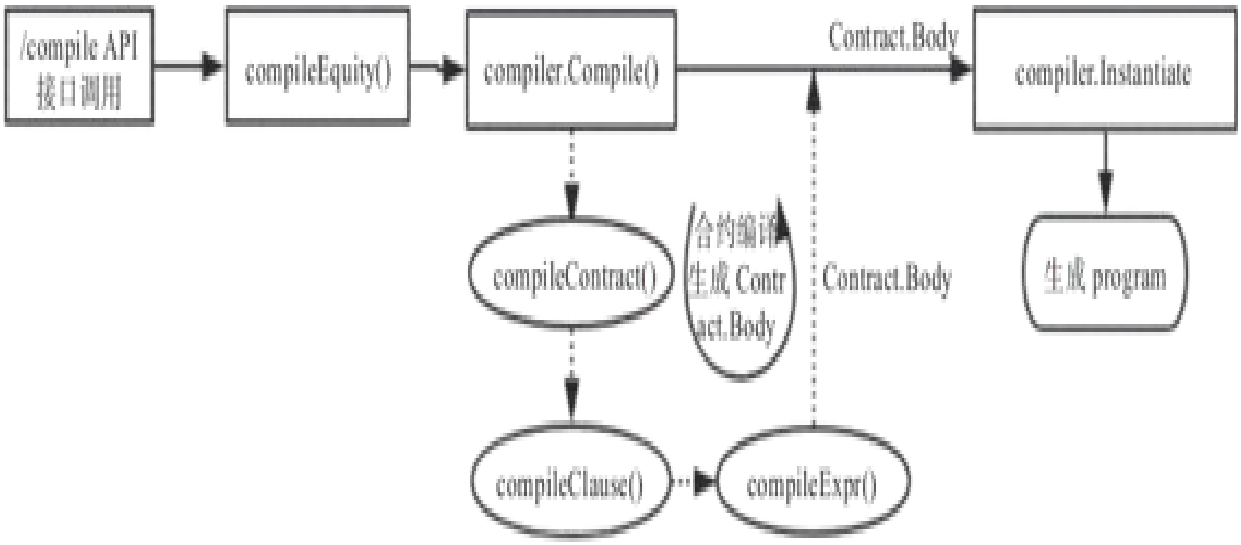


图8-8 合约程序字节码生成流程

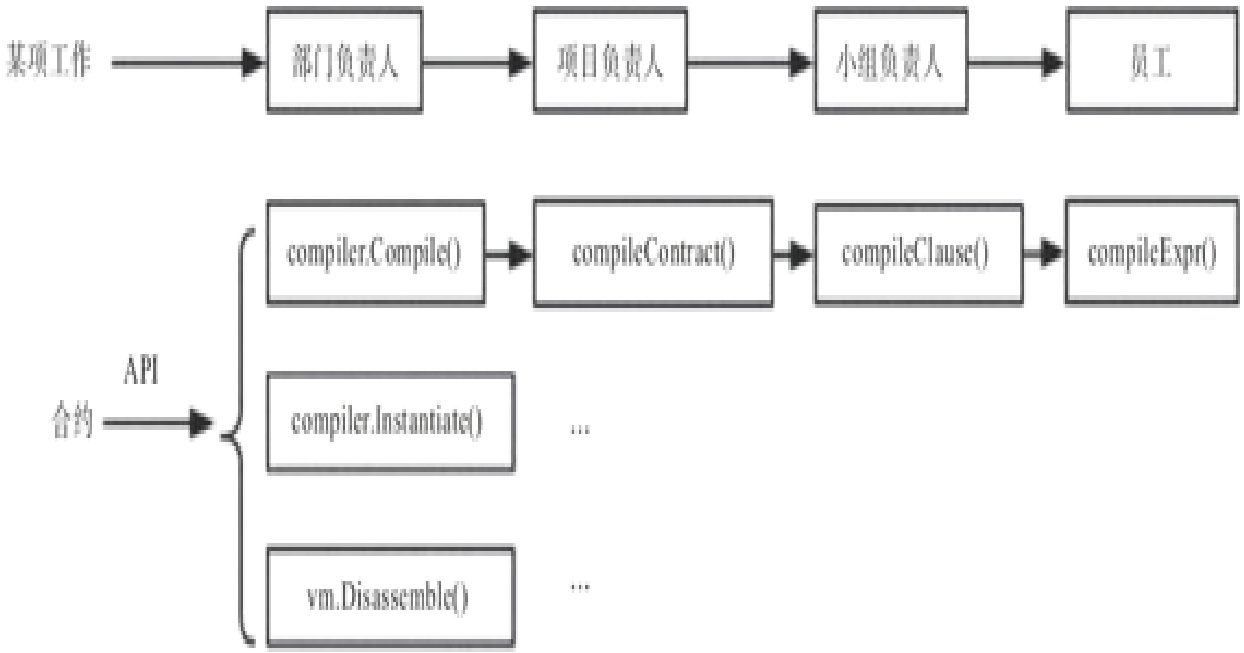


图8-9　合约编译流程类比

工作完成后逐层返回结果，最终交付给部门负责人（compiler.Compile()）；最后需要由其他部门做进一步处理，如compiler.Instantiate()和vm.Disassemble()。

1.API调用

API接口内容已在第4章中详细讲解，这里我们直接进入/compile接口。调用compile接口编译合约，代码如下：

```
api/api.go
m.Handle("/compile", jsonHandler(a.compileEquity))
```

API Server解析HTTP包头，匹配到路由后，跳转至a.compileEquity函数。代码如下：

```
api/compile.go
func (a *API) compileEquity(req compileReq) Response {
    resp, err := compileEquity(req)
    //...
```

```
    return NewSuccessResponse(resp)
}
```

a.compileEquity调用compileEquity合约编译函数。代码如下：

```
api/compile.go
func compileEquity(req compileReq) (compileResp, error) {
    var resp compileResp
    compiled, err :=
compiler.Compile(strings.NewReader(req.Contract))
    // ...

    if req.Args != nil {
        resp.Program, err = compiler.Instantiate(contract.Body,
contract.Params, false, req.Args)
        // ...
        resp.Opcodes, err = vm.Disassemble(resp.Program)
        // ...
    }
    // ...
}
```

代码中调用了三个函数，可以理解为该项工作需要对三个部分依次处理才能完成。

· compiler.Compile()：编译合约得到字节码Contract.Body。

· compiler.Instantiate()：实例化Contract.Body，生成program。

· vm.Disassemble()：将program转化为供人阅读的字符串。

其中，compiler.Compile()处理过程有四层。

第一层：compiler.Compile()处理合约。

调用API，把工作（合约）交付给对应的部门负责人compiler.Compile()。该函数中编译合约代码生成Contract.Body字节码。其过程较为复杂，先要进行预处理，之后调用不同的函数对contract、clause、expr进行编译，即工作的逐层细化。流程如图8-10所示。

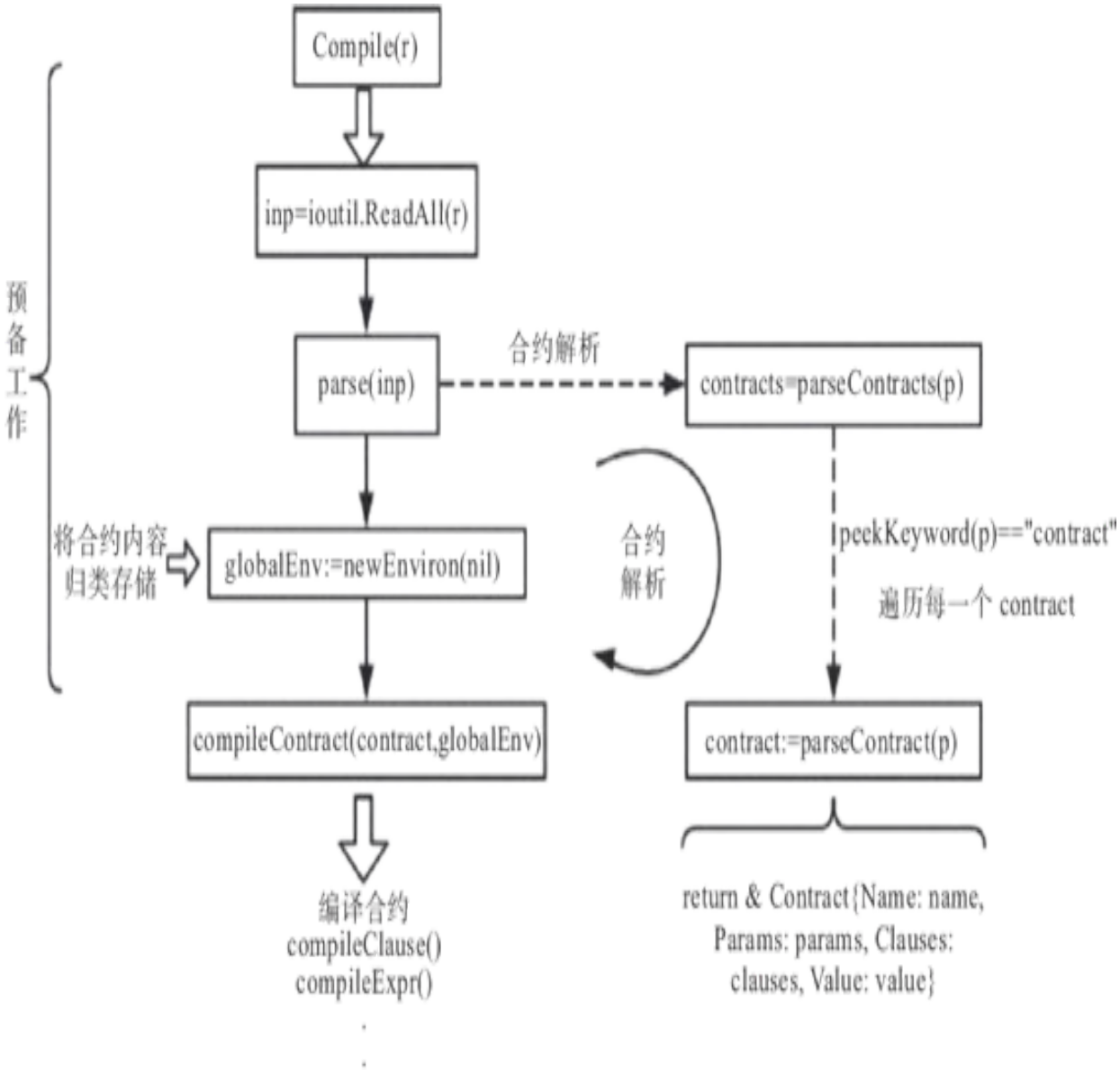


图8-10　Compile处理流程

对合约的预处理工作占据了compiler.Compile()函数的大部分篇幅，完成预处理后接着调用下一层compileContract()继续合约的编译。具体代码如下：

```
equity/compiler/compile.go
func Compile(r io.Reader) ([]*Contract, error) {
    inp, err := ioutil.ReadAll(r)
    // ...
    contracts, err := parse(inp)
```

```
    // ...
    globalEnv := newEnviron(nil)
    for _, k := range keywords {
        globalEnv.add(k, nilType, roleKeyword)
    }
    for _, b := range builtins {
        globalEnv.add(b.name, nilType, roleBuiltin)
    }

    for _, contract := range contracts {
        contract.Recursive = checkRecursive(contract)
    }

    for _, contract := range contracts {
        err = globalEnv.addContract(contract)
        if err != nil {
            return nil, err
        }
    }

    for _, contract := range contracts {
        err = compileContract(contract, globalEnv)
        // ...
    }

    return contracts, nil
}
```

“部门负责人”compiler.Compile()的预处理工作如下。

1）ioutil.ReadAll()：通过I/O接口读取合约代码，代码中可能会包含多个合约。

2）parse()：合约语法解析。将字符串的合约代码解析成Contract对象，参见图8-10，对比Contract结构体的定义，可以看到最后返回的信息与前四项（name、params、clause、value）一一对应，余下项则通过合约编译生成（Body、Opcodes、Recursive、Step）。

3）newEnviron()：实例化Environ对象，将合约内容进行分类，存入该链表结构中。Environ存储合约相关内容，类似单向链表结构，每一个元素指向上一个parent。由于第一次实例化该对象，所以链表头parent=nil。

4）globalEnv.add（roleKeyword）：keywords为"contract""clause"等关键字，将这些内容加入到globalEnv中，对应标记为roleKeyword。

5）globalEnv.add（roleBuiltin）：builtlin为一些运算操作。

6）checkRecursive（contract）：检查合约是否自我调用，结果返回contract.Recursive中。

7）globalEnv.addContract（contract）：将合约添加到globalEnv中。

以上工作完成后，将工作交付给下一层“项目负责人”compileContract（contract，globalEnv）。由于代码中可能存在多个合约，所以此处通过range对contracts中的每一个contract调用compileContract()函数进行编译。

第二层：compileContract编译合约。

compileContract()函数是合约编译过程中的关键部分，我们通过函数分层图具体分析编译思路，如图8-11所示。

compileContract()函数编译合约分为三个步骤：

1）处理合约参数。

2）调用compileClause编译合约条款，内部调用compileExpr编译表达式，最终得到opcodes。

3）将opcodes转化为字节码，得到contract.Body。

代码示例如下：

```
equity/compiler/compile.go
func compileContract(contract *Contract, globalEnv *environ)
error {
        //…
       env := newEnviron(globalEnv)
       for _, p := range contract.Params {
    err = env.add(p.Name, p.Type, roleContractParam)
```

```
    if err != nil {
        return err
    }
}
        //…
      var stk stack
      if len(contract.Clauses) > 1 {
    stk = stk.add("<clause selector>")
     }
        //…
      b := &builder{}
      if len(contract.Clauses) == 1 {
    err = compileClause(b, stk, contract, env,
contract.Clauses[0])
           //…
      }
           //…
            opcodes := optimize(b.opcodes())
    prog, err := vm.Assemble(opcodes)
    if err != nil {
        return err
    }

    contract.Body = prog
    contract.Opcodes = opcodes

    contract.Steps = b.steps()

    return nil
}
```

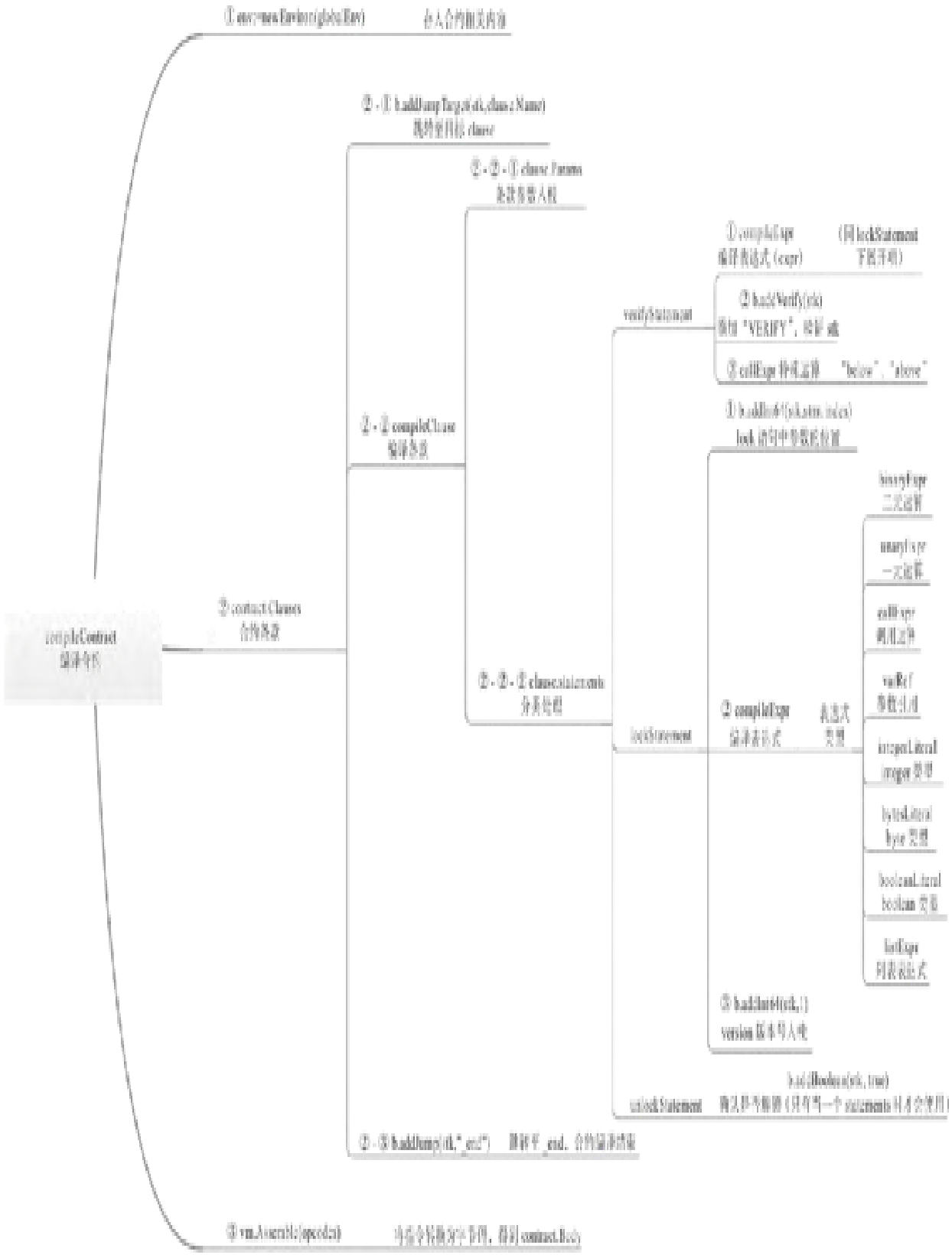


图8-11　Compile编译合约

“项目负责人”compileContract()的预处理工作如下。

1）env：=newEnviron（globalEnv）：以globalEnv为parent，在第二层编译合约函数代码中将每个合约的params、value、clause加入env。

2）var stk stack：合约中会存在多个clause函数，创建一个栈stk，在有多个clause函数时，选择并进行相应操作。此外，stk还用来更新builder中的栈。

3）b：=&builder{}：用于存放合约的指令栈。

builder指令栈结构如下：

```
equity/compiler/builder.go
type builder struct {
    items          []*builderItem
    pendingVerify *builderItem
}

type builderItem struct {
    opcodes string
    stk     stack
}
```

builder类型结构体是合约的指令栈，即采用栈类型数据结构来存储通过合约编译得到的所有opcodes指令和指令对应的栈数据。栈的更新通过上一步所创建的stk实现（var stk stack）。

例如，equity/compiler/builder.go文件中举例指令栈数据如图8-12所示。

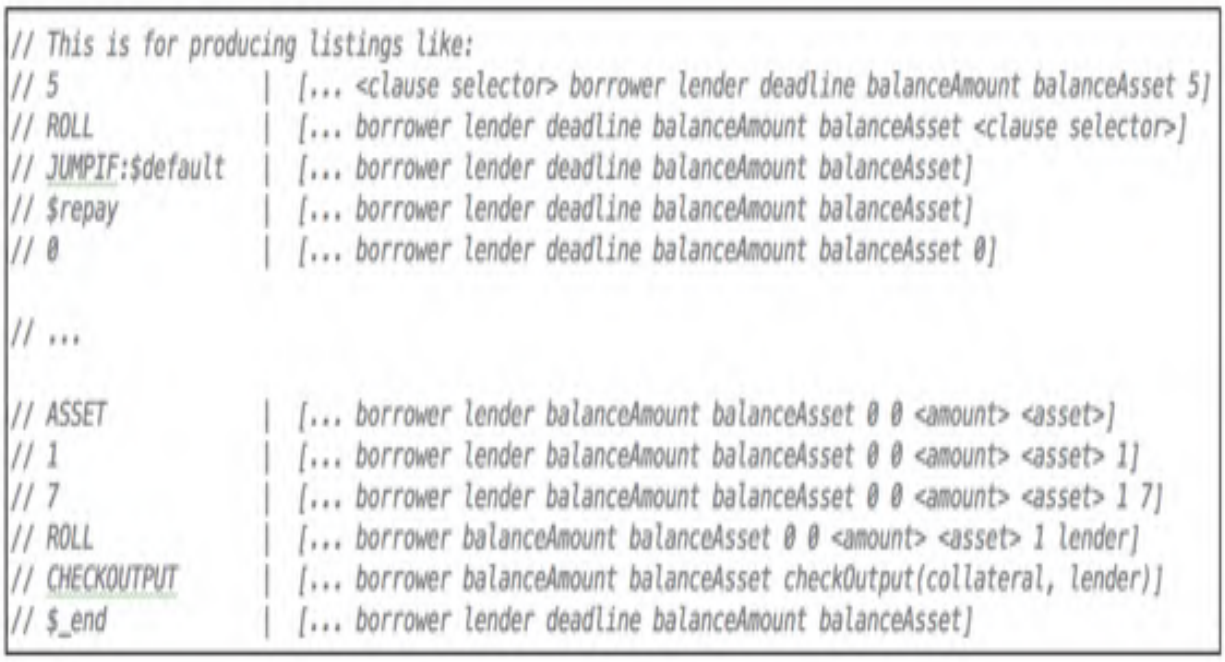


图8-12　builder指令栈信息

图8-12的左侧为指令，右侧为执行指令后栈数据状态。

第三层：compileClause编译条款。

经过第二层的处理，此处的工作可细化为：编译合约中条款函数。compileClause()函数扮演小组负责人的角色，将工作再次细化并分配。代码示例如下：

```
func compileClause(b *builder, contractStk stack, contract
*Contract, env *environ, clause *Clause) error {
    // ...
    env = newEnviron(env)
    for _, p := range clause.Params {
        err = env.add(p.Name, p.Type, roleClauseParam)
    }
    for _, req := range clause.Reqs {
        err = env.add(req.Name, valueType, roleClauseValue)
    }

    var stk stack
    for _, p := range clause.Params {
        stk = stk.add(p.Name)
```

```
    }
    stk = stk.addFromStack(contractStk)

    // ...
    for _, s := range clause.statements {
        switch stmt := s.(type) {
        case *verifyStatement:
            stk, err = compileExpr(b, stk, contract, clause,
env, counts, stmt.expr)
            // ...
            stk = b.addVerify(stk)

            if c, ok := stmt.expr.(*callExpr); ok && len(c.args)
== 1 {
                if b := referencedBuiltin(c.fn); b != nil {
                    switch b.name {
                    case "below":
                        clause.BlockHeight =
append(clause.BlockHeight, c.args[0].String())
                    case "above":
                        clause.BlockHeight =
append(clause.BlockHeight, c.args[0].String())
                    }
                }
            }

        case *lockStatement:
            stk = b.addInt64(stk, stmt.index)
            // ...
            // version
            stk = b.addInt64(stk, 1)

            // prog
            stk, err = compileExpr(b, stk, contract, clause,
env, counts, stmt.
 program)
            // ...

            stk = b.addCheckOutput(stk,
fmt.Sprintf("checkOutput(%s, %s)", stmt.
 locked, stmt.program))
            stk = b.addVerify(stk)

        case *unlockStatement:
            if len(clause.statements) == 1 {
                stk = b.addBoolean(stk, true)
            }
        }
```

```
    }
    // ...
    return nil
}
```

小组负责人“compileClause()”的预处理工作如下。

1）env：=newEnviron（globalEnv）：以上一层env为parent，将clause参数和reqs加入env。

2）创建一个栈stk。

3）对不同类型的statements分别处理，共有verify、lock和unlock三种操作。

· verifyStatement：调用compileExpr()编译boolean类型表达式，编译完成后返回调用b.addVerify（stk），完成指令栈b的更新。

· lockStatement：用于解锁合约资产或条款函数资产。首先将索引Index入栈；若为合约资产，则直接将Amount和Asset入栈；若为条款函数资产，即支付资产，则调用compileExpr()分别对Amount、Asset进行编译并入栈；最后将版本号入栈，再对Program编译并入栈。stk栈如图8-13所示。

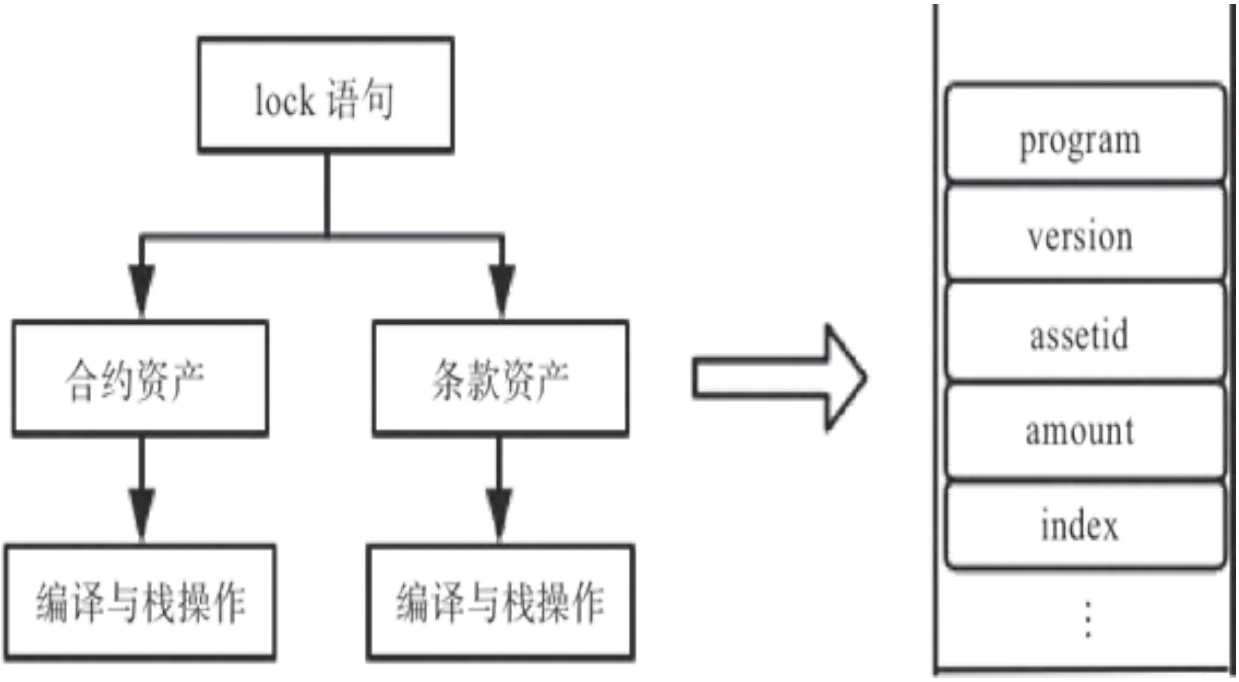


图8-13　lockStatement语句编译

完成以上内容的编译及入栈后，向指令栈b中添加CHECKOUTPUT和VERIFY指令，其中CHECKOUTPUT指令作用将栈中以上5项数据弹出并通过VERIFY指令进行检验。

· unlockStatement：unlock语句仅用于解锁合约资产，所以每个条款函数中最多只有一个unlock语句。此处判断unlock语句个数是否为1，若是则将true入栈。

所有函数最后将调用requireAllValuesDisposedOnce()、typeCheckClause()和requireAll-ParamsUsedInClause()函数，检查是否已完成合约所有内容的编译。

第四层：compileExpr编译表达式。

我们利用compileExpr()来编译条款中的表达式。compileExpr()篇幅较长，此处不一一列出，只例举lockStatement语句。

调用compileExpr()将二元运算表达式拆分为左右两部分分别进行编译，中间加入符号来判定两者关系是否成立，如expr>expr。

通过b.addOps将指令入栈。代码示例如下：

```
func compileExpr(b *builder, stk stack, contract *Contract,
clause *Clause, env *environ, counts map[string]int, expr
expression) (stack, error) {
    switch e := expr.(type) {
        case *binaryExpr:
            stk, err = compileExpr(b, stk, contract, clause,
env, counts, e.left)
            // ...
            stk, err = compileExpr(b, stk, contract, clause,
env, counts, e.right)
            // ...
            lType := e.left.typ(env)
            // ...
            rType := e.right.typ(env)
            // ...
            switch e.op.op {
            case "==", "!=":
                // ...
            }

            stk = b.addOps(stk.dropN(2), e.op.opcodes,
e.String())
        // ...
}
```

至此，我们已从最外部函数compiler.Compile()至最小编译单元compileExpr()，完成了对编译合约的分层梳理。

2. 合约指令栈信息

编译合约过程中会不断地往builder结构中添加合约栈信息，我们以交易所挂单合约为例进行讲解。交易所挂单合约指令信息（未优化）如下：

```
4 ROLL JUMPIF:$cancel $trade 0 2 ROLL 2 ROLL 1 4 ROLL
CHECKOUTPUT JUMP:$_end $cancel 4 ROLL 4 ROLL TXSIGHASH SWAP
CHECKSIG $_end
```

在合约编译的过程中，这些指令都存放在builder的栈中。

那么指令是如何添加到builder中的？例如，调用b.addRoll（stk）方法可以添加“ROLL”指令。方法流程如图8-14所示。

addRoll代码示例如下：

```
func (b *builder) addRoll(stk stack, n int) stack {
  b.addInt64(stk, int64(n))
  return b.add("ROLL", stk.roll(n))
}
```

在函数最后的b.add()方法中，先调用stk.roll()函数完成对栈的操作（将栈中第n个元素移动至栈顶），得到新的栈，作为参数传入b.add()。代码示例如下：

```
func (b *builder) add(opcodes string, newstack stack) stack {
    //…
    item := &builderItem{opcodes: opcodes, stk: newstack}
    if opcodes == "VERIFY" {
        b.pendingVerify = item
    } else {
        b.items = append(b.items, item)
    }
    return newstack
}
```

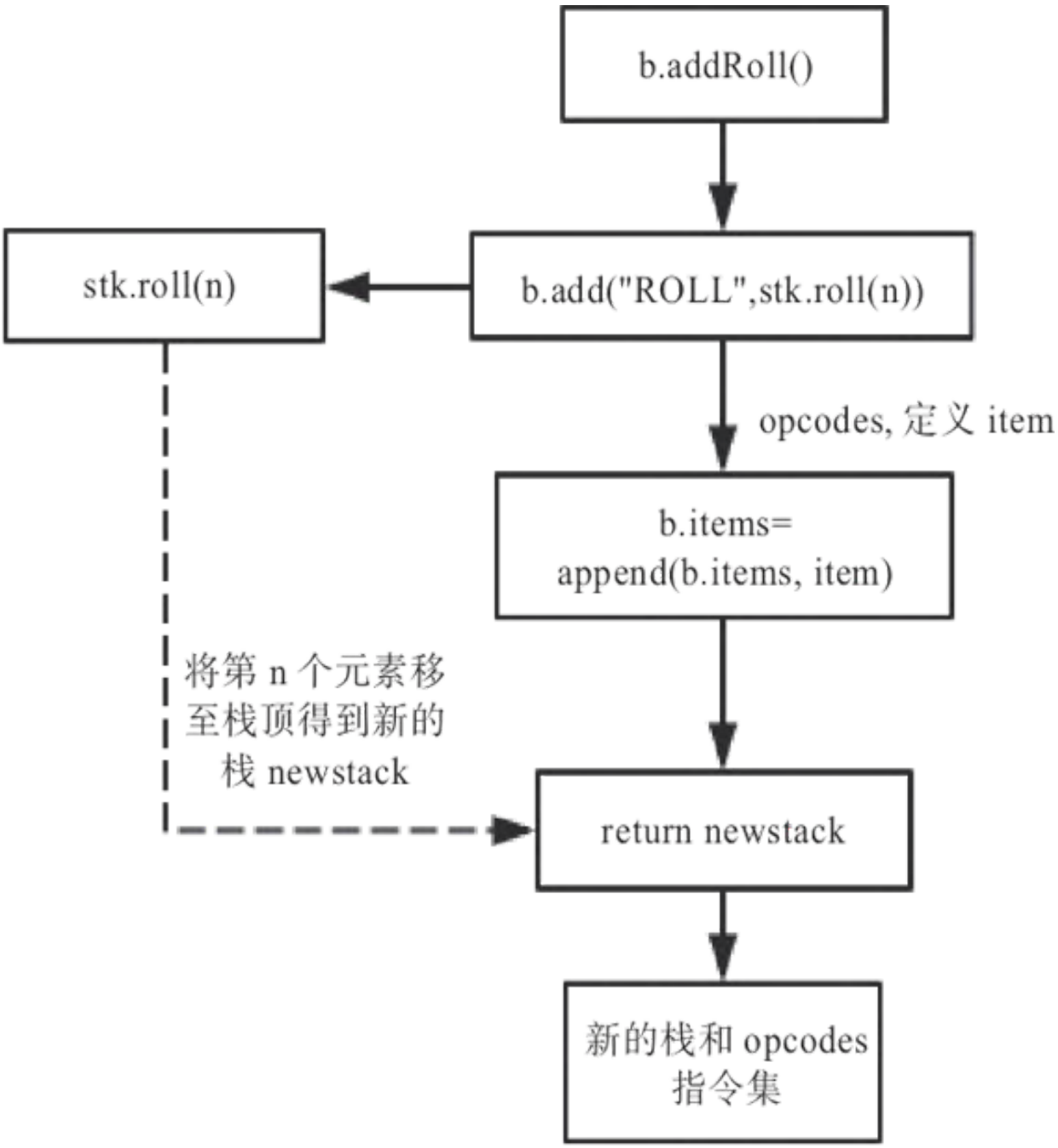


图8-14　b.addRoll（stk）方法流程

将opcodes指令定义为builderItem类型结构体item，并将参数传给item。判断opcodes是否为verify，若是，则标记该项b.pendingVerify=item；若不是，则将该item内容添加至原b中items项后（这里对应的opcodes项会添加“ROLL”），以不断扩充b中的opcodes并更新栈。其他指令同理。

注意

所有方法最终均会调用b.add()以扩充opcodes及更新栈。

3. 虚拟机指令优化

合约代码编译完成后，此时合约的所有操作指令都已存储在builder的opcodes中。根据不同的指令组合进行优化可减少交易所需手续费。代码示例如下：

```
opcodes := optimize(b.opcodes())
```

优化规则定义如下：

```
equity/compiler/optimize.go
var optimizations = []struct {
  before, after string
}{
  {"0 ROLL", ""},
  {"0 PICK", "DUP"},
  {"1 ROLL", "SWAP"},
  {"1 PICK", "OVER"},
  {"2 ROLL", "ROT"},
  {"TRUE VERIFY", ""},
  {"SWAP SWAP", ""},
  {"OVER OVER", "2DUP"},
  {"SWAP OVER", "TUCK"},
  {"DROP DROP", "2DROP"},
  {"SWAP DROP", "NIP"},
  {"5 ROLL 5 ROLL", "2ROT"},
  {"3 PICK 3 PICK", "2OVER"},
  {"3 ROLL 3 ROLL", "2SWAP"},
  {"2 PICK 2 PICK 2 PICK", "3DUP"},
  {"1 ADD", "1ADD"},
  {"1 SUB", "1SUB"},
  {"EQUAL VERIFY", "EQUALVERIFY"},
  {"SWAP TXSIGHASH ROT", "TXSIGHASH SWAP"},
  {"SWAP EQUAL", "EQUAL"},
  {"SWAP EQUALVERIFY", "EQUALVERIFY"},
  {"SWAP ADD", "ADD"},
  {"SWAP BOOLAND", "BOOLAND"},
  {"SWAP BOOLOR", "BOOLOR"},
```

```
  {"SWAP MIN", "MIN"},
  {"SWAP MAX", "MAX"},
  {"DUP 2 PICK EQUAL", "2DUP EQUAL"},
  {"DUP 2 PICK EQUALVERIFY", "2DUP EQUALVERIFY"},
  {"DUP 2 PICK ADD", "2DUP ADD"},
  {"DUP 2 PICK BOOLAND", "2DUP BOOLAND"},
  {"DUP 2 PICK BOOLOR", "2DUP BOOLOR"},
  {"DUP 2 PICK MIN", "2DUP MIN"},
  {"DUP 2 PICK MAX", "2DUP MAX"},
}
```

优化规则中定义前一项为未优化的指令，后一项为优化后的指令。如{"0 ROLL", ""}，ROLL指令的操作为：将栈中第n个元素移动到栈顶。0 ROLL即把栈中第0个元素移动到栈顶，而栈中0位置所代表的就是栈顶元素。该条操作指令等效于没有进行任何操作，所以优化后为空指令“”。

再比如{“0 PICK”，“DUP”}，PICK指令可以将栈中第n个元素复制到栈顶，0 PICK即将栈顶元素复制并压入栈，其显然与DUP指令等效，可将其优化为单个DUP指令。在合约的编译过程中，均会调用该函数对指令进行优化。我们以交易所挂单合约指令为例进行讲解。

指令优化之前，代码如下：

```
4 ROLL JUMPIF:$cancel $trade 0 2 ROLL 2 ROLL 1 4 ROLL
CHECKOUTPUT JUMP:$_end $cancel 4 ROLL 4 ROLL TXSIGHASH SWAP
CHECKSIG $_end
```

指令优化之后，代码如下：

```
4 ROLL JUMPIF:$cancel $trade 0 ROT ROT 1 4 ROLL CHECKOUTPUT
JUMP:$_end $cancel 4 ROLL 4 ROLL TXSIGHASH SWAP CHECKSIG $_end
```

可以看到，优化规则将2 ROLL指令改成ROT指令。

4. 转换字节码

将已优化过的opcodes转化为相应的字节码，代码示例如下：

```
equity/compiler/compile.go
prog, err := vm.Assemble(opcodes)
if err != nil {
    return err
}
contract.Body = prog
```

指令字节码转换函数Assemble()，将传进来的已优化过的opcodes转化为相应的字节码。比如，opcodes为2 3 ADD 5 NUMEQUAL，转化为字节码0x525393559c。过程示例如下：

```
逐字节分析字节码：52 53 93 55 9c
OP_2（将数字2入栈)→52
OP_3（将数字3入栈)→53
OP_ADD （加操作）→93
OP_5（将数字5入栈）→55
OP_NUMEQUAL（NUMEQUAL)→9c
```

显然这里生成的字节码与opcodes一一对应。合约的contract.Body生成过程同理。

至此，compiler.Compile()编译合约并生成contract.Body已全部完成。下一个部门将接手工作——compiler.Instantiate()实例化生成最终的program。

5.生成program字节码

compiler.Instantiate()的功能是，生成合约程序字节码program，运行流程如图8-15所示。

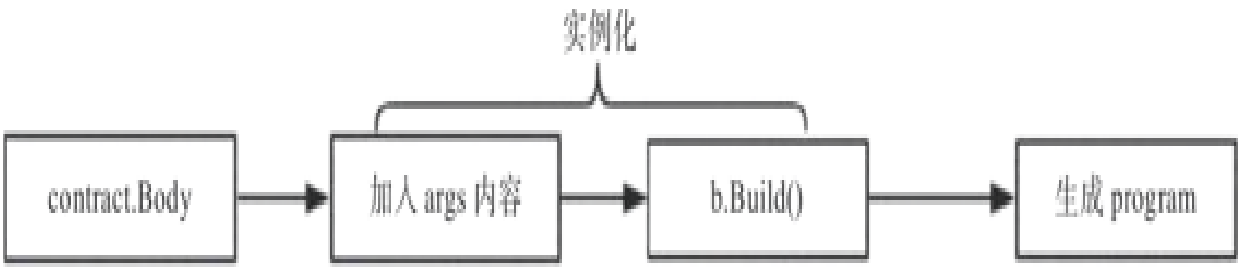


图8-15　compiler.Instantiate()运行流程

生成program字节码，代码示例如下：

```
equity/compiler/compile.go
func Instantiate(body []byte, params []*Param, recursive bool,
args []ContractArg) ([]byte, error) {
    //…
    b := vmutil.NewBuilder()

    for i := len(args) - 1; i >= 0; i-- {
        a := args[i]
        switch {
        case a.B != nil:
            var n int64
            if *a.B {
                n = 1
            }
            b.AddInt64(n)
        case a.I != nil:
            b.AddInt64(*a.I)
        case a.S != nil:
            b.AddData(*a.S)
        }
    }
           //…
    return b.Build()
}
```

“部门负责人”compiler.Instantiate()的预处理工作如下。

1）实例化Builder对象。

2）args参数项，a.I、a.S、a.B分别表示args中类型为Integer、String、Boolean的参数。这里使用for循环将args内容添加到b.program中，如b.AddData（*a.S）。

3）调用b.Build()生成program，返回至resp.Program。

vmutil.NewBuilder结构体如下：

```
protocol/vm/vmutil/builder.go
type Builder struct {
    program     []byte
    jumpCounter int
    jumpAddr map[int]uint32
    jumpPlaceholders map[int][]int
}
```

结构说明如下。

· program：合约程序字节码。

· jumpCounter：跳转计数器。

· jumpAddr：将跳转目标编号映射到其绝对地址。

· jumpPlaceholders：将跳转目标编号映射到必须填充其绝对地址的位置列表。

添加数据，代码如下：

```
func (b *Builder) AddData(data []byte) *Builder {
    b.program = append(b.program, vm.PushdataBytes(data)...)
    return b
}
```

调用vm.PushdataBytes（data）函数将data字段添加至b.program。

build生成program，代码如下：

```
protocol/vm/vmutil/builder.go
func (b *Builder) Build() ([]byte, error) {
    for target, placeholders := range b.jumpPlaceholders {
        addr, ok := b.jumpAddr[target]
        // ...
        for _, placeholder := range placeholders {

binary.LittleEndian.PutUint32(b.program[placeholder:placeholder+
```

```
4], addr)
            }
        }
        return b.program, nil
}
```

合约中有两个跳转类型的指令，如下所示。

· jumpAddr：将跳转目标编号映射到其绝对地址，即目标编号与跳转地址的映射。

· jumpPlaceholders：将跳转目标编号映射到必须填充其绝对地址的位置列表，第一项[int]与jumpAddr中的[int]一致，均为目标编号；第二项int类型数组用来存储program中需要填充地址的位置。

地址的填充由Build()函数实现，最终完成program的编译。至此，合约程序字节码program已经生成。

8.6.3　合约程序字节码示例

我们以交易所挂单合约为例，对所生成的program进行简单分析。POST请求包示例如下：

```
{
    "args":[

{"string":"a55e72ad9b9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d
3f8bd459bff"},
        {"integer":100},

{"string":"00149c31f762ad902dbee9565fad8b74ca4785c0af99"},

{"string":"ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624
164b7d520f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689
ef99c36e1fd"}
    ],
    "contract":"contract TradeOffer(assetRequested:
Asset,amountRequested: Amount,seller: Program,cancelKey:
PublicKey) locks offered { clause trade() requires payment:
amountRequested of assetRequested { lock payment with seller
unlock offered } clause cancel(sellerSig: Signature) { verify
checkTxSig(cancelKey, sellerSig) unlock offered } }"
}
```

生成的program示例如下：

```
40ba7babfdbe9f30e5f3df47a2ec960629840e360a5daf624e57624164b7d520
f32cddf7e748b96cac771963b2fa46a30a672a7d9145828b7e3d689ef99c36e1
fd1600149c31f762ad902dbee9565fad8b74ca4785c0af99016420a55e72ad9b
9040dbca1d0c898ff72b0dfa163d6a9954b5733bb9d3f8bd459bff741a547a64
13000000007b7b51547ac1631a000000547a547aae7cac00c0
```

逐字节分析program，大致分为以下两部分。

1）将args内容由下到上依次入栈，见表8-1。

表8-1　args内容入栈

指令	说明
40	对应 OP_DATA_64 指令，将 64 字节数据入栈
ba7babfdb...99c36e1fd	要入栈的 64 字节数据，对应 cancelKey
16	对应 OP_DATA_22 指令，将 22 字节数据入栈
00149c31f...785c0af99	要入栈的 22 字节数据，对应 seller
01	对应 OP_DATA_1 指令，将 1 字节数据入栈
64	要入栈的 1 字节数据，对应 amountRequested，之前传入的资产总量为 100，即十六进制 64
20	对应 OP_DATA_32 指令，将 32 字节数据入栈
a55e72ad9...8bd459bff	要入栈的 32 字节数据，对应 assetRequested
74	OP_DEPTH，压入数据栈的元素个数

2）Contract.Body内容依次入栈，见表8-2。

表8-2　Contract.Body内容入栈

指令	说明
1a	对应 OP_DATA_26 指令，将 26 字节数据入栈
547a64130...47aae7cac	要入栈的 26 字节数据，是合约编译所产生的对应操作字节码
00	OP_FALSE，把一个字节空串压入栈中
c0	OP_CHECKPREDICATE，调用子虚拟机执行对应的 program，如果执行成功则返回 true，否则返回 false

8.6.4　合约程序字节码的执行

合约交易通过虚拟机执行相应的字节码program来完成。类比生成program时的工作模型，虚拟机执行program相当于公司对某项工作的验收。虚拟机运行流程如图8-16所示。

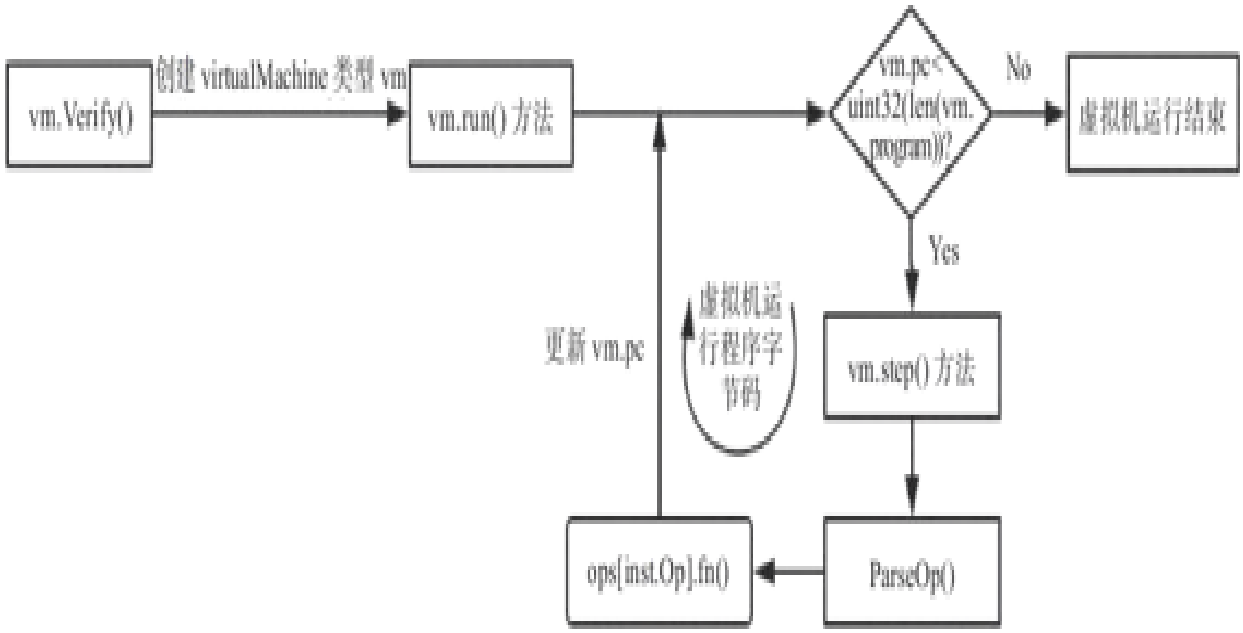


图8-16　虚拟机的运行流程

虚拟机验证，代码如下：

```
protocol/vm/vm.go
func Verify(context *Context, gasLimit int64) (gasLeft int64,
err error) {
    // ...
    if context.VMVersion != 1 {
        return gasLimit, ErrUnsupportedVM
    }

    vm := &virtualMachine{
        expansionReserved: context.TxVersion != nil &&
*context.TxVersion == 1,
        program:           context.Code,
        runLimit:          gasLimit,
        context:           context,
    }
```

```
    args := context.Arguments
    for i, arg := range args {
        err = vm.push(arg, false)
        // ...
    }

    err = vm.run()
    // ...
    return vm.runLimit, wrapErr(err, vm, args)
}
```

虚拟机验证流程如下：

1）验证虚拟机版本。

2）实例化virtualMachine对象。

3）虚拟机上每一步操作都会产生相应的花费，所以虚拟机的运行需要gasLeft参数。

4）对dataStack操作，将数据入栈。

5）调用方法vm.run()运行虚拟机。

虚拟机运行，代码如下：

```
func (vm *virtualMachine) run() error {
    for vm.pc = 0; vm.pc < uint32(len(vm.program)); {
        err := vm.step()
        // ...
    }
    return nil
}
```

vm.pc作为program处理过程中的位置标识，当vm.pc小于program的长度时（即未处理完program），则根据vm.pc循环调用方法vm.step()。代码示例如下：

```
func (vm *virtualMachine) step() error {
    inst, err := ParseOp(vm.program, vm.pc)
    // ...
    vm.nextPC = vm.pc + inst.Len

    // ...
    err = ops[inst.Op].fn(vm)
    // ...
    err = vm.applyCost(vm.deferredCost)
    // ...
    vm.pc = vm.nextPC
    // ...

    return nil
}
```

根据vm.pc，调用ParseOp()函数对指定位置的字节码program进行解析，并返回该字节码所对应的指令操作信息。其中，vm.pc通过vm.nextPC不断地被更新。对应指令操作，主要体现在ops[inst.Op].fn（vm）语句中。

inst为program解析后的指令操作，ops[]内容如下：

```
protocol/vm/ops.go
var (
    ops = [256]opInfo{

        OP_FALSE: {OP_FALSE, "FALSE", opFalse},
        OP_PUSHDATA1: {OP_PUSHDATA1, "PUSHDATA1", opPushdata},
        //…
}
```

调用指令操作inst.Op所对应的函数，完成操作。比如，inst.Op=OP_PUSHDATA1时，通过ops[inst.Op].fn（vm）语句调用OP_PUSHDATA1所对应的函数opPushdata，进而实现该指令操作。

run()函数中循环调用step()函数，step()函数根据vm.pc依次执行program，并对vm.pc进行更新，直至将program执行完毕，虚拟机运行完成。

8.6.5　合约程序字节码的执行示例

结合前面交易所挂单合约例子中的program，生成图8-17右侧第一行栈。注意，合约编译的操作字节码为“547a6413000000007b7b51547ac1631a000000547a547aae7cac”，与opcodes一一对应。

1. 栈信息

栈中有一个clause选择器入栈，第一部分字节码将cancelKey、seller、amountRequested、assetRequested内容依次入栈，即：

```
<--栈底
栈顶-->
[...<clause selector> cancelKey seller amountRequested
assetRequested]
```

2. 指令集

第一部分字节码执行完毕后，进入指令操作的字节码：依次执行左侧每一条指令，右侧表示指令执行后栈的变化。

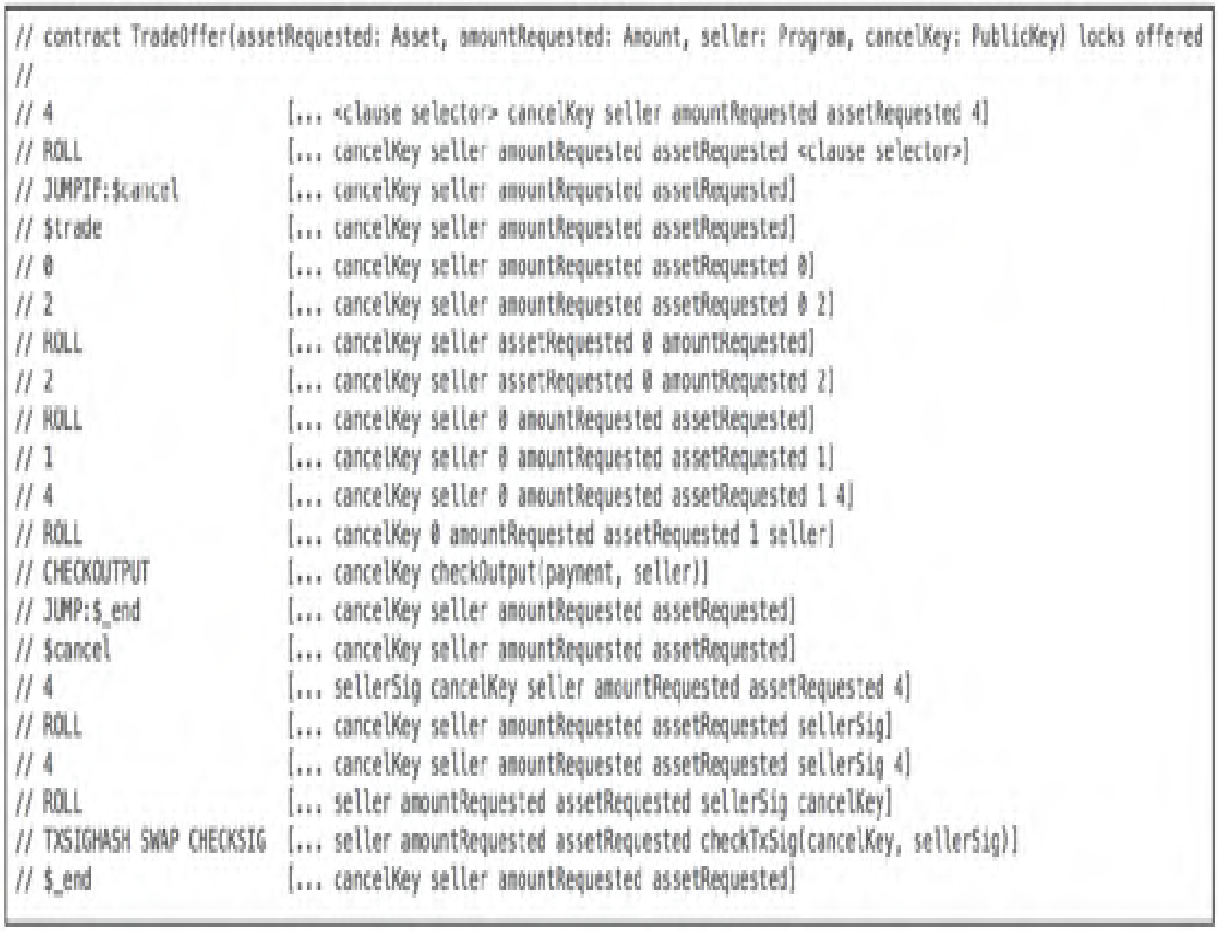


图8-17　交易所挂单合约的指令信息

以4 ROLL指令（将栈中第4个元素移动至栈顶）为例，栈信息如下：

```
//4     [...<clause selector> cancelKey seller amountRequested
assetRequested 4]
//ROLL [...cancelKey seller amountRequested assetRequested
<clause selector>]
```

3. ROLL指令的栈操作

第4个元素<clause selector>移动至栈顶。运行过程如图8-18所示。

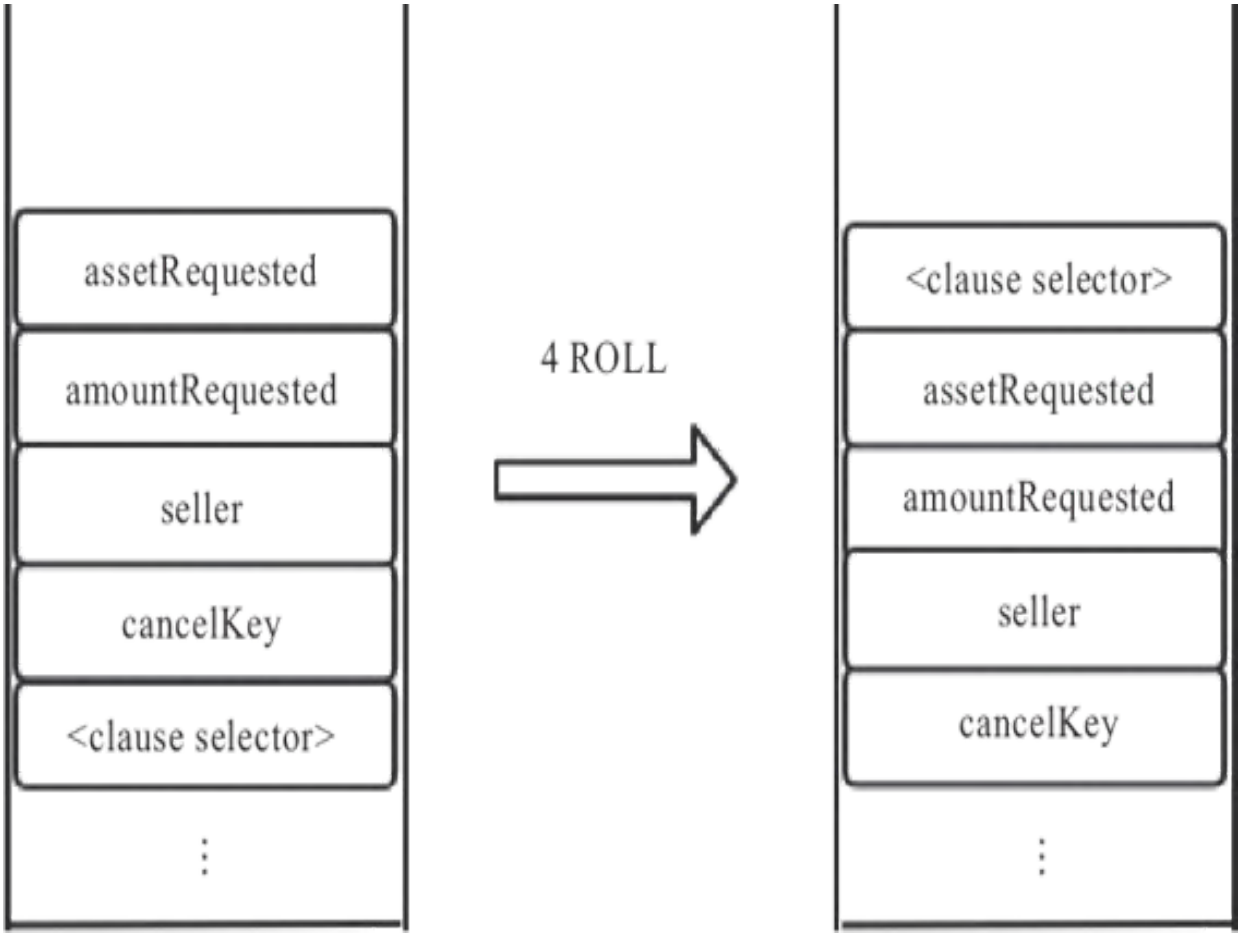


图8-18　4ROLL指令的栈操作

虚拟机运行其他指令同理，直至program执行完毕，此处不再赘述。

8.7 BVM指令集

1.数值操作指令（见表8-3）

表8-3 数值操作指令

关键字	值（十六进制）	描述
OP_1ADD	0x8b	返回栈顶元素加 1
OP_1SUB	0x8c	返回栈顶元素减 1
OP_2MUL	0x8d	返回栈顶元素乘 2
OP_2DIV	0x8e	返回栈顶元素除 2
OP_NEGATE	0x8f	返回栈顶元素符号取反后的结果
OP_ABS	0x90	返回栈顶元素的绝对值
OP_NOT	0x91	判断输入值是否为 0，若为 0，返回 true，否则返回 false
OP_0NOTEQUAL	0x92	判断输入值是否不为 0，若不为 0，返回 true，否则返回 false
OP_ADD	0x93	返回 x,y 两个输入的和，即 x-y（以下输入顺序均为 x,y）
OP_SUB	0x94	返回 x、y 两个输入的差，即 y-x
OP_MUL	0x95	返回 x、y 两个输入的积，即 x*y
OP_DIV	0x96	返回 x、y 两个输入的商，即 y/x
OP_MOD	0x97	返回 x、y 两个输入取余的结果，即 y mod x
OP_LSHIFT	0x98	返回输入 y 左移 x 位的结果，即 y << uint(x)。第一个输入 x 为移位系数，第二个输入 y 为操作数
OP_RSHIFT	0x99	返回输入 y 右移 x 位的结果，即 y >> uint(x)。第一个输入 x 为移位系数，第二个输入 y 为操作数
OP_BOOLAND	0x9a	布尔类型数据与（and）操作，即 x、y 两个输入均不为零时返回 true，否则返回 false
OP_BOOLOR	0x9b	布尔类型数据或（or）操作，即 x、y 两个输入至少有一个不为零时返回 true，否则返回 false
OP_NUMEQUAL	0x9c	判断两个整型输入是否相等，若相等，返回 true，否则返回 false
OP_NUMEQUALVERIFY	0x9d	判断两个整型输入是否相等，若相等，返回 nil（VERIFY 成功），否则返回 VERIFY 失败
OP_NUMNOTEQUAL	0x9e	判断 x、y 两个输入是否相等，若不相等返回 true，否则返回 false
OP_LESSTHAN	0x9f	判断 x、y 两个输入是否满足 y<x，若是，返回 true，否则返回 false
OP_GREATERTHAN	0xa0	判断 x、y 两个输入是否满足 y>x，若是，返回 true，否则返回 false
OP_LESSTHANOREQUAL	0xa1	判断 x、y 两个输入是否满足 y<=x，若是，返回 true，否则返回 false
OP_GREATERTHANOREQUAL	0xa2	判断 x、y 两个输入是否满足 y>=x，若是，返回 true，否则返回 false
OP_MIN	0xa3	返回 x、y 两个输入中较小的一个
OP_MAX	0xa4	返回 x、y 两个输入中较大的一个
OP_WITHIN	0xa5	判断 x、y、z 三个输入是否满足 y <= z <= x，若是，返回 true，否则返回 fale

2. 加密和哈希操作指令（见表8-4）

表8-4　加密和哈希操作指令

关键字	值（十六进制）	描述
OP_SHA256	0xa8	返回输入内容的 sha256 运算结果
OP_SHA3	0xaa	返回输入内容的 sha3 运算结果
OP_HASH160	0xab	返回输入内容的 Ripemd160 运算结果
OP_CHECKSIG	0xac	验证交易的签名，验证成功返回 true，否则返回 false
OP_CHECKMULTISIG	0xad	多个签名时，依次验证每个签名和公钥，若均验证成功，返回 true，否则返回 false
OP_TXSIGHASH	0xae	返回交易签名哈希，即虚拟机 context 的 TxSigHash

3. 虚拟机检查操作指令

这些指令会导致虚拟机立即停止并返回false，参见表8-5。

表8-5　虚拟机检查操作指令

关键字	值（十六进制）	描述
OP_CHECKOUTPUT	0xc1	从数据栈中弹出5项进行检查，依次为：program，version，assetid，amount，index
OP_ASSET	0xc2	返回资产ID（assetid），即虚拟机的context.AssetID。校验context的资产ID，主要有两种情况：1）如果第一个action对象是Issuance，则直接推送value.AssetID；2）如果第一个action对象是Spend，则推送SpentOutput.Source.Value.AssetID
OP_AMOUNT	0xc3	返回资产数量（amount），即虚拟机的context.Amount。校验context的资产数量，主要有两种情况：1）如果第一个action对象是Issuance，则直接推送value.Amount；2）如果第一个action对象是Spend，则推送SpentOutput.Source.Value.Amount
OP_PROGRAM	0xc4	返回接收交易的program，即虚拟机的context.Code
OP_INDEX	0xc9	返回目标接收对象的索引位置，即虚拟机的context.DestPos
OP_ENTRYID	0xca	返回目标接收对象的交易entryID，即虚拟机的context.EntryID
OP_OUTPUTID	0xcb	返回已花费UTXO所生成的交易SpentOutputID，即虚拟机的context.SpentOutputID
OP_BLOCKHEIGHT	0xcd	返回当前的块高度height，即虚拟机的context.BlockHeight

4. 位操作指令（见表8-6）

表8-6　位操作指令

关键字	值（十六进制）	描述
OP_INVERT	0x83	返回输入值按位取反后的结果
OP_AND	0x84	返回 x、y 两个输入按位逻辑与运算的结果
OP_OR	0x85	返回 x、y 两个输入按位逻辑或运算的结果

（续）

关键字	值（十六进制）	描述
OP_XOR	0x86	返回 x、y 两个输入按位逻辑异或运算的结果
OP_EQUAL	0x87	判断 x、y 两个输入是否相等，若相等，返回 true，否则返回 false
OP_EQUALVERIFY	0x88	先执行 OP_EQUAL 操作，后执行 OP_VERIFY 操作。判断 x、y 两个输入是否相等，若相等，返回 nil（VERIFY 成功），否则返回 VERIFY 失败

5. 字符串处理操作指令（见表8-7）

表8-7　字符串处理操作指令

关键字	值（十六进制）	描述
OP_CAT	0x7e	返回 x、y 两个字符串类型输入拼接后的结果，即 append(y, x)
OP_SUBSTR	0x7f	返回从指定偏移量 y 开始截取固定 x 长度的 z 字符串子集，输入值的顺序为 size（x）、offset（y）、str（z）。如 x=3，y=2，z=123456，返回结果为 345
OP_LEFT	0x80	返回截取 y 字符串左边指定 x 长度的子串，输入值的顺序为 size、str。如 x=3，y=bytom，返回结果为 byt
OP_RIGHT	0x81	返回截取 y 字符串右边指定 x 长度的子串，输入值的顺序为 size、str。如 x=2，y=bytom，返回结果为 tom
OP_SIZE	0x82	返回字符串的长度大小，即 len(str)
OP_CATPUSHDATA	0x89	返回 x、y 两个 []byte 数据输入拼接后的结果，即 append(y, x)

6. 栈控制操作指令（见表8-8）

表8-8　栈控制操作指令

关键字	值（十六进制）	描述
OP_TOALTSTACK	0x6b	将主栈的栈顶元素压入辅栈顶部，并从主栈中删除。若主栈为空则执行失败，返回错误
OP_FROMALTSTACK	0x6c	将辅栈的栈顶元素压入主栈顶部，并从辅栈中删除。若辅栈为空则执行失败，返回错误
OP_2DROP	0x6d	删除栈顶 2 个元素
OP_2DUP	0x6e	复制栈顶 2 个元素
OP_3DUP	0x6f	复制栈顶 3 个元素
OP_2OVER	0x70	将栈底的 2 个元素复制到栈顶
OP_2ROT	0x71	将栈中第 5 和第 6 个元素移动到栈顶
OP_2SWAP	0x72	将栈中第 3 和第 6 个元素移动到栈顶
OP_IFDUP	0x73	当数据栈的栈顶不是 false 时，复制栈顶元素并入栈
OP_DEPTH	0x74	将栈中元素的个数压入栈
OP_DROP	0x75	删除栈顶元素
OP_DUP	0x76	复制栈顶元素并入栈

（续）

关键字	值（十六进制）	描述
OP_NIP	0x77	删除栈中的第二个元素
OP_OVER	0x78	将栈中的第二个元素复制到栈顶
OP_PICK	0x79	将栈中第 n 个元素复制到栈顶
OP_ROLL	0x7a	将栈中第 n 个元素移动到栈顶
OP_ROT	0x7b	将栈中第 3 个元素移动到栈顶
OP_SWAP	0x7c	交换栈中的两个元素
OP_TUCK	0x7d	复制栈顶元素并插入到栈中第二个元素之后

7. 控制流程操作指令（见表8-9）

表8-9　控制流程操作指令

关键字	值（十六进制）	描述
OP_JUMP	0x63	无条件跳转到栈的指定位置，将栈顶的4字节小端模式的无符号整数地址赋值给PC，程序将无条件跳转至此处执行，如果该PC值不存在，则执行失败
OP_JUMPIF	0x64	有条件跳转到栈的指定位置，和OP_JUMP的区别在于该操作从数据栈中取出一个boolean值，如果为true才把PC设置到该地址，如果为false，则不进行任何操作
OP_VERIFY	0x69	校验栈的栈顶元素，如果为true，则删除栈顶元素，并返回nil，否则执行失败
OP_FAIL	0x6a	无条件执行失败
OP_CHECKPREDICATE	0xc0	调用子虚拟机执行对应对program，执行成功，则返回true，否则返回false。此外，若虚拟机的运行限制小于256，则立即执行失败

8. 入栈操作指令（见表8-10）

表8-10　入栈操作指令

关键字	值（十六进制）	描述
OP_PUSHDATA1	0x4c	将1个字节的数据长度前缀和对应长度大小的字节数据压入栈，即(prefix + data)，其中，prefix=len(data)，前缀prefix所能表示的范围为(0, 255)bytes，对应的十六进制范围为(0x00, 0xFF)
OP_PUSHDATA2	0x4d	将2个字节的数据长度前缀和对应长度大小的字节数据压入栈，即(prefix + data)，其中，prefix=len(data)，前缀prefix所能表示的范围为(0, 65535)bytes，对应的十六进制范围为(0x0000, 0xFFFF)
OP_PUSHDATA4	0x4e	将4个字节的数据长度前缀和对应长度大小的字节数据压入堆栈，即(prefix + data)，其中prefix=len(data)，而前缀prefix所能表示的范围为(0, 4294967295)bytes，对应的十六进制范围为(0x00000000, 0xFFFFFFFF)
OP_1NEGATE	0x4f	把数字-1压入栈
OP_NOP	0x61	无任何操作

9. 常见关键字（见表8-11）

表8-11　常见关键字

关键字	值（十六进制）	描述
OP_FALSE	0x00	压入一个字节的空字符串到栈中（并非OP_NOP操作，这里会有一个元素被压入栈）
OP_0	0x00	同上
OP_1	0x51	将数字1压入栈
OP_TRUE	0x51	同上
OP_2	0x52	将数字2压入栈
⋮	⋮	将数字…压入栈
OP_15	0x5f	将数字15压入栈
OP_16	0x60	将数字16压入栈

10. 十进制数值操作指令

跟在该类指令后的下一个操作码字节是将要压入栈的数据。

操作码范围为0x00～0x4b，操作是将对应字节的数据压入栈，最小为1个字节（0x01），最大为75个字节（0x4b）（参见表8-12）。

表8-12　十进制数值操作指令

关键字	值（十六进制）	描述
OP_DATA_1	0x01	将1个字节大小的数据压入栈
OP_DATA_2	0x02	将2个字节大小的数据压入栈
⋮	⋮	将…个字节大小的数据压入栈
OP_DATA_74	0x4a	将74个字节大小的数据压入栈
OP_DATA_75	0x4b	将75个字节大小的数据压入栈

8.8 本章小结

虚拟机是一个较抽象的概念，是智能合约的运行环境。本章从虚拟机的角度，对区块链底层指令操作及智能合约的生命周期进行了深层次的剖析，将较为抽象的智能合约运行过程变得具体化，从使用层面进入到机理层面。重点关注虚拟机底层的相关操作指令集、合约编译及转化成对应指令的流程、虚拟机栈信息及program的逐字节执行过程等。本章最后列出了虚拟机的所有指令及其描述，供读者自行操作生成合约字节码。

第9章

钱包层

9.1 概述

钱包作为区块链代币使用中不可或缺的组成部分，充当着个人区块资产管家的角色。比原链钱包层采用UTXO模型，其主要功能是按照Account账户进行账户资产的归集，涉及账户管理、交易管理、资产管理等相关内容。

钱包十分类似于保险柜，有如下几个主要要素：

· 所有者，即谁的保险柜。

· 保险柜内存放什么资产，比如人民币还是美元，BTC还是BTM。

· 资产明细，即记录保险柜内东西的账本。

· 开启保险柜的钥匙或密码，能打开保险柜才能取走其中的东西。

如果要从保险柜里取10万美元，首先要由保险柜所有者通过自己的钥匙打开保险柜，检查保险柜内资产明细，查看是否有10万美元的余额，如果余额充足，从保险柜里取走对应额度资产，并将支出记录到保险柜的资产明细表上，最后关闭保险柜。

本章主要内容如下：

· 钱包的基本概念及密钥。

· 钱包账户管理。

· 钱包资产管理。

· 钱包交易管理。

· 钱包的备份/恢复操作流程。

9.2 钱包对象

Wallet对象是钱包的结构体对象，包括账户管理对象（保险柜的拥有者）、资产管理对象（保险柜里存的资产）、公私钥存储对象（保险柜的密码或钥匙），以及需要获取数据的区块链数据对象（保险柜中的资产账本）。Wallet对象的结构体如下：

```
wallet/wallet.go
type Wallet struct {
  DB          db.DB
  rw          sync.RWMutex
  status      StatusInfo
  AccountMgr *account.Manager
  AssetReg    *asset.Registry
  Hsm         *pseudohsm.HSM
  chain       *protocol.Chain
  rescanCh    chan struct{}
}
```

结构体说明如下：

- DB：数据库对象，用于数据持久化。
- status：钱包节点的状态。
- AccountMgr：账户管理对象。
- AssetReg：资产管理对象。
- Hsm：加密模块对象（公钥私钥对的管理对象）。

· chain：本地主链对象。

· rescanCh：钱包数据更新通知信号。

9.3 密钥管理

介绍密钥管理首先需要了解分层确定性钱包（Hierarchical Deterministic wallet，HD-Wallet）的概念。分层确定性是相对于不确定性而言的，旧的比特币钱包是不确定性钱包，其中私钥的生成是没有规律可言的，每一个公私钥对与其他公私钥对都是没有关系的，出于安全考虑，每增加一个私钥都需要进行钱包的备份，这对钱包的保管与导入都是十分不方便的，一旦丢失其中一把私钥就会丢失一部分财产。

根据比特币核心开发者Gregory Maxwell的原始描述和讨论，Pieter Wuille在2012年2月11日整理完善并提交了BIP32，直到2016年6月15日BIP32才被合并到Bitcoin Core。目前，几乎所有的钱包服务商都整合了该协议，比原链也采用了BIP32。BIP32的核心是通过一颗种子来生产主密钥MasterKey，通过主密钥层层派生，衍化出无穷层级的子孙密钥（可以是私钥也可以是公钥）。只要保存好种子便可在任意环境重建层级结构，这为钱包的使用和备份提供了极大便利。

BIP32的密钥衍生层级如图9-1所示。

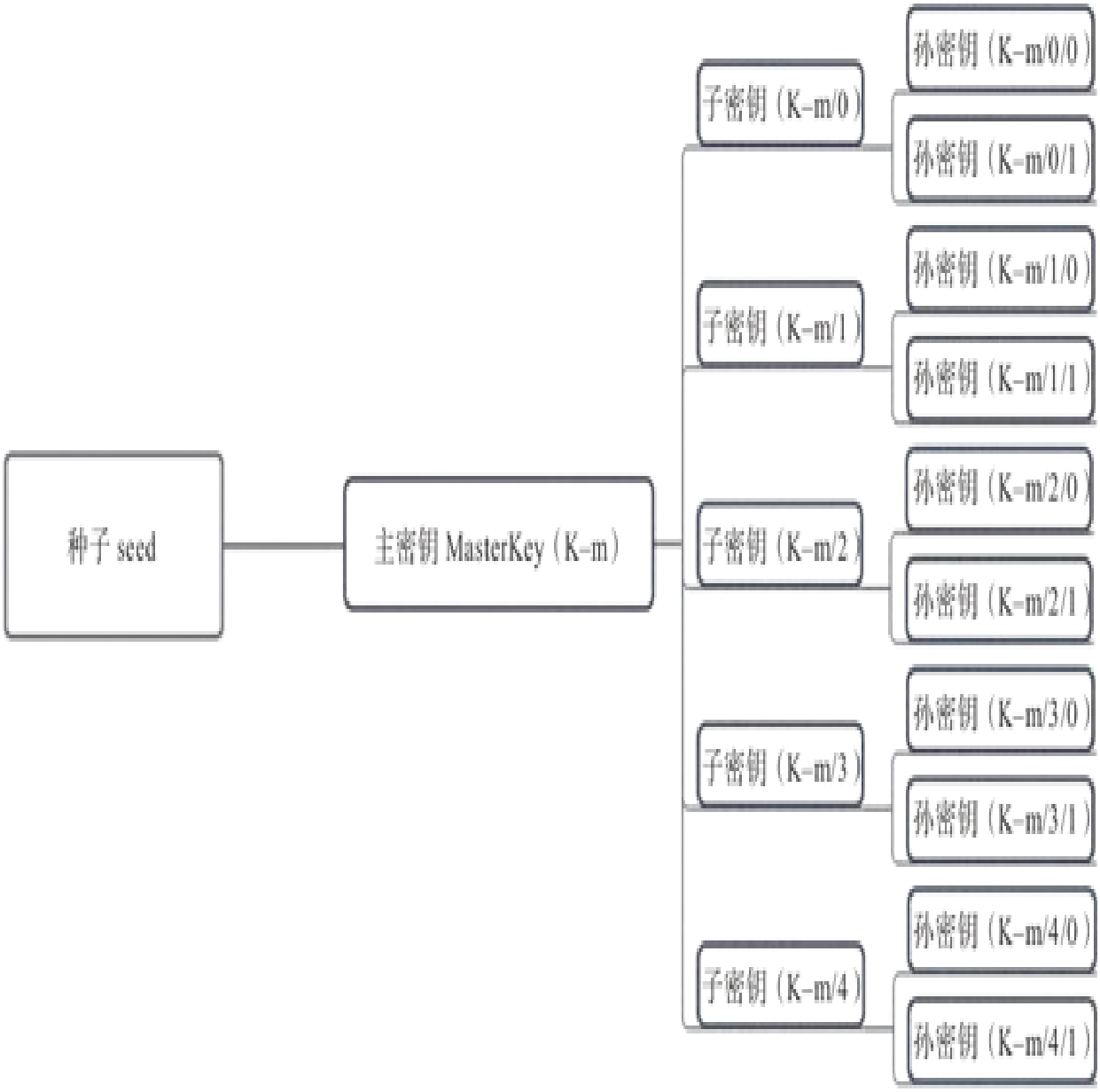


图9-1　BIP32密钥衍生层级结构

为了防止派生过程过度依赖密钥本身，扩展密钥（Extended Key）分别给公钥和私钥延伸出了256位32字节的熵，称为链码。扩展公私钥总长512位合计64字节，前32字节为公/私钥，后32字节为ChainCode链码，链码参与密钥的衍生过程。图9-2所示为一个扩展的公私钥结构。

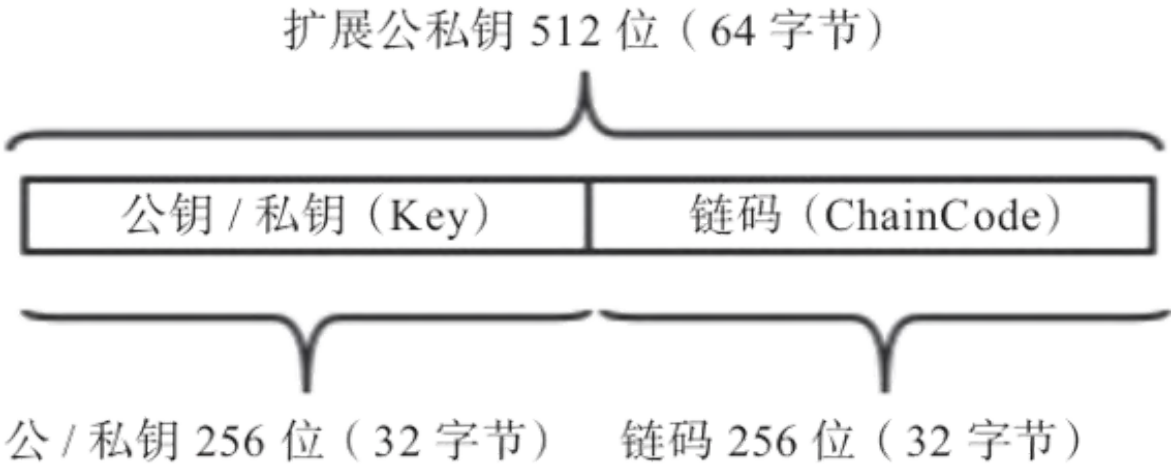


图9-2　扩展公私钥结构

HD层级中节点索引的定位通过Path路径来表示，例如：

· “m/0/0”：表示从主密钥m开始派生的第1个子密钥派生的第1个子密钥。

· “m/1/12”：表示从主密钥m开始派生的第2个子密钥派生的第13个子密钥。

· “m/1’/12”：表示从主密钥m开始派生的第2^31+1个子密钥派生的第13个子密钥。

从例三中可以看到m/1’/12的表示方法，其中1’表示的是硬化或强化子密钥（Hardened Child Key）的概念，每个可扩展密钥拥有2^32个子密钥，前2^31个子密钥称为普通密钥，从0到2^31-1；后2^31个密钥称为硬化或强化子密钥，从2^31到2^32-1；每一个子密钥有一个索引号，普通子密钥为i，强化子密钥的索引号以i’表示，其代表i+2^31。

有了密钥层级定位的方法，当需要给某一层级生成子密钥的时候，只需找到密钥位置，对扩展密钥执行衍生算法即可。

在HD层级模型中一个私钥对应一个公钥，一个私钥对应2^32个子私钥，一个公钥对应2^32个子公钥，父与子是一对多的关系。

9.3.1 密钥对生成

我们通过调用/create-key接口生成密钥信息。代码示例如下：

```
blockchain/pseudohsm/pseudohsm.go
func (h *HSM) XCreate(alias string, auth string) (*XPub, error)
{
    // ...
    if ok := h.cache.hasAlias(normalizedAlias); ok {
        return nil, ErrDuplicateKeyAlias
    }
    xpub, _, err := h.createChainKDKey(auth, normalizedAlias,
false)
    // ...
    h.cache.add(*xpub)
    return xpub, err
}
```

create-key接口接收参数alias（密钥别名）和password（密钥密码），首先查找缓存中是否存在重名公钥，如果不存在重名公钥则通过createChainKDKey函数生成密钥信息，并将公钥加入到缓存中，代码如下：

```
func (h *HSM) createChainKDKey(auth string, alias string, get
bool) (*XPub, bool, error) {
    xprv, xpub, err := chainkd.NewXKeys(nil)
    // ...
    id := uuid.NewRandom()
    key := &XKey{
        ID:      id,
        KeyType: "bytom_kd",
        XPub:    xpub,
        XPrv:    xprv,
        Alias:   alias,
    }
    file := h.keyStore.JoinPath(keyFileName(key.ID.String()))
    if err := h.keyStore.StoreKey(file, key, auth); err != nil
{
        return nil, false, errors.Wrap(err, "storing keys")
    }
    return &XPub{XPub: xpub, Alias: alias, File: file}, true,
```

```
nil
}
```

密钥生成流程如下：

1）chainkd.NewXKeys生成密钥对。

2）随机生成一个uuid串，keystore文件使用UTC时间加上uuid字符串来命名。

3）往keystore文件中写入加密密钥。

以上密钥生成流程主要针对账户的根密钥或主密钥（MasterKey）的生成。

密钥的衍生（即HD钱包模型中密钥层级衍生）主要存在两个维度：父私钥衍生子私钥，父公钥衍生子公钥。父私钥通过父公钥衍生子公钥是可以的，但是父公钥衍生子私钥是行不通的，因为存在巨大的安全隐患。

根据指定路径进行普通子私钥的衍生可以参见：

```
crypto/ed25519/chainkd/chainkd.go#L211
func (xprv XPrv) Derive(path [][]byte) XPrv {}
```

根据指定路径进行普通子公钥的衍生可以参见：

```
crypto/ed25519/chainkd/chainkd.go#L222
 func (xpub XPub) Derive(path [][]byte) XPub {}
```

对于指定路径派生子密钥（无论私钥还是公钥）派生出的都是普通子密钥，如果通过指定字符串派生子密钥，派生公钥与派生私钥是有很大差别的。

普通子公钥通过父公钥即可派生，其对应的子私钥可以通过父私钥派生，参见：

```
crypto/ed25519/chainkd/chainkd.go#L175
func (xpub XPub) Child(sel []byte) (res XPub) {}
```

普通子私钥可以通过父公钥派生，但强化子私钥必须由父私钥派生，参见：

```
crypto/ed25519/chainkd/chainkd.go#L2
func (xprv XPrv) Child(sel []byte, hardened bool) XPrv{}
```

9.3.2 密钥对生成算法

密钥对包含私钥和公钥两个部分，这两者的生成算法是不同的，公链中私钥由于其私人保密的特性要求，其生成往往使用HMAC算法（Hash-based Message Authentication Code，哈希消息验证码）。公钥由于与私钥的一一对应关系，并且需要在开放的网络环境进行传播，所以往往采用椭圆曲线加密算法，从私钥派生公钥。比原链采用的下椭圆加密算法是ed25519。

1.私钥生成的过程

（1）私钥生成

私钥生成使用HMAC-SHA512算法。HMAC是一种特殊的消息摘要算法，与其他单向哈希函数的消息摘要算法的不同之处在于，其算法输入除需要一段消息之外还需要一个密钥。比原链在HMAC实现过程中采用的哈希算法是SHA512。

在生成根私钥的过程中，HMAC-SHA512算法的两个输入参数为：

· 密钥：字符串“Root”。

· 种子：从GO随机数流中读取32字节随机数。

代码示例如下：

```
crypto/ed25519/chainkd/chainkd.go#L35
func RootXPrv(seed []byte) (xprv XPrv) {}
```

（2）子私钥衍生

在子私钥衍生过程中，HMAC-SHA512算法的两个输入参数根据子密钥是强化子私钥还是普通子私钥会有所不同。衍生普通子私钥的2个输入参数是：

· 密钥：父扩展公钥的后32字节，即父扩展公钥中的链码。

· 种子：字母“N”混合父扩展公钥前32字节及用户指定的字符串。

代码示例如下：

```
crypto/ed25519/chainkd/chainkd.go#L79
func (xprv XPrv) nonhardenedChild(sel []byte) (res XPrv) {}
```

（3）强化子私钥衍生

衍生强化子私钥的2个输入参数是：

· 密钥：父扩展私钥的后32字节，即父扩展私钥中的链码。

· 种子：字母“H”混合父扩展私钥前32字节及用户指定字符串。

代码示例如下：

```
/bytom/crypto/ed25519/chainkd/chainkd.go#L69
func (xprv XPrv) hardenedChild(sel []byte) (res XPrv) {}
```

2. 公钥生成的过程

公链中公钥是由私钥派生出来的，派生过程往往通过椭圆曲线加密算法实现，椭圆曲线依赖的数学难题为：k*G=Q。其中，k为正整数；G为椭圆曲线上的一点，称为基点。已知k和G很容易计算Q，已知G和Q很难计算出k，这是椭圆曲线的离散对数问题。目前广泛使用的椭圆曲线算法是由世界著名的密码学家Daniel J. Bernstein于2006年独立设计并发表的25519系列算法，此系列算法的特点是速度快、安全性高、设计开放，目前广泛应用于各类软件与协议。详细的25519加密算法可以参见原版论文《High-speed high-security signatures》，https://ed25519.cr.yp.to/ed25519-20110926.pdf。

ed25519的椭圆曲线方程如下：

$y^2=x^3+486662x^2+x$，$modulo p=2^{255}-19$

比原链的私钥派生公钥的椭圆曲线加密算法使用了ed25519算法，目标扩展公钥长度为64字节，其组成是由前32字节加上扩展私钥的前32字节通过ed25519运算得到，名为公钥。后32字节由扩展私钥的后32字节直接拼接，名为链码。

算法实现代码示例如下：

```
crypto/ed25519/chainkd/chainkd.go
func (xprv XPrv) XPub() (xpub XPub) {
    var scalar ecmath.Scalar
    copy(scalar[:], xprv[:32])

    var P ecmath.Point
    P.ScMulBase(&scalar)
    buf := P.Encode()

    copy(xpub[:32], buf[:])
    copy(xpub[32:], xprv[32:])

    return
}
```

9.3.3 密钥加密存储

用户的密钥生成之后需要进行存储，存储与读取包含两部分：一部分是提取密钥对的密码，另一部分是私钥。为了安全考虑必须对两者都进行加密，并且由于存储的数据在使用的时候还需要解密还原，所以需要选择一种对称加密算法，在存储的同时把加密的初始化场景参数进行持久化。目前公链中对密钥加密常用到的密钥导出函数是PBKDF2，算法的原理是，对密钥和随机加入的salt进行某种单向哈希函数计算，计算结果作为新的salt与密钥继续进行哈希计算，重复多轮之后（数万次）生成最终的密文。针对这一算法的逆向破解往往需要花费几百年甚至更长，所以能够有效保证数据的安全性。如果持有算法所需的密钥和初始化加入的时间salt，经过相同的加密过程可以得到相同的值。

目前公链中对私钥进行加密常用的算法是AES，该算法属于块加密算法，根据其想要得到的密钥长度是16字节还是32字节，可以分为AES128，AES256等，AES的加密模式有很多种，如ECB，CBC，CFB，CTR等，AES中有初始向量IV的概念，针对不同的IV加密结果是不同的，其长度一般与加密块长度一致，比如16字节。

针对提取公私钥对的密码信息的加密存储，比原链中采用的是PBKDF2算法，其中针对加解密性能需要N、P、R三个参数控制，参数N与CPU/Memory性能相关，其取值要求N>1且N为2的幂；P与R两个参数要求P×R小于2的30次幂。比原链中的PBKDF2算法，其输入参数包括：用户指定的密码，由随机数流中读取的32字节salt，N，R，P，DKLen；哈希算法为SHA256。其中，参数R的固定值为8；DkLen（期望得到的密钥长度）值为32；N与P给了两组选择——StandardScryptN=2^18、StandardScryptP=1与LightScryptN=2^12、LightScryptP=6，这两组区别在于不同CPU与内存的配比，第一组以现代的CPU性能加253MB内存可于1s的CPU时间完成，第二组以相同性能的CPU加4MB内存可于100ms的CPU时间内完成。两种组合根据实际情况进行选择，在加密机HSM生成过程中，比原链选择了第二组参数，代码示例/bytom/blockchain/pseudohsm/pseudohsm.go#L40。整个PBKDF2算法加密过程，代码示例如下：

```
crypto/scrypt/scrypt.go#L229
func Key(password, salt []byte, N, r, p, keyLen int) ([]byte,
error) {
    if N <= 1 || N&(N-1) != 0 {
        return nil, errors.New("scrypt: N must be > 1 and a
power of 2")
    }
    if uint64(r)*uint64(p) >= 1<<30 || r > maxInt/128/p || r >
maxInt/256 || N > maxInt/128/r {
        return nil, errors.New("scrypt: parameters are too
large")
    }

    xy := make([]uint32, 64*r)
    v := make([]uint32, 32*N*r)
    b := pbkdf2.Key(password, salt, 1, p*128*r, sha256.New)

    for i := 0; i < p; i++ {
        smix(b[i*128*r:], r, N, v, xy)
    }

    return pbkdf2.Key(password, b, 1, keyLen, sha256.New), nil
}
```

比原链中针对私钥的加密方式采取的是AES128算法CTR模式，加密块为16字节。私钥数据在进行AES-128-CTR加密时有三个输入参数：密钥key由提取公私钥信息的密码经过PBKDF2算法加密后的密文前16个字节组成，加密内容为私钥，初始化向量IV根据块大小取值为16。最终经过加密运算获得密文，代码示例如下：

```
blockchain\pseudohsm\keystore_passphrase.go#L95
encryptKey := derivedKey[:16]
keyBytes := key.XPrv[:]

iv := randentropy.GetEntropyCSPRNG(aes.BlockSize) // 16
cipherText, err := aesCTRXOR(encryptKey, keyBytes, iv)
if err != nil {
    return nil, err
}
```

加密存储最终希望存储的结构为如下的JSON结构：

```
encryptedKeyJSON := encryptedKeyJSON{
    cryptoStruct,
    key.ID.String(),
    key.KeyType,
    version,
    key.Alias,
    hex.EncodeToString(key.XPub[:]),
}
```

其中，字段cryptoStruct的结构如下：

```
cryptoStruct := cryptoJSON{
    Cipher:       "aes-128-ctr",
    CipherText:   hex.EncodeToString(cipherText),
    CipherParams: cipherParamsJSON,
    KDF:          "scrypt",
    KDFParams:    scryptParamsJSON,
    MAC:          hex.EncodeToString(mac),
}
```

cryptoJSON字段说明如下：

· cipherParamsJSON：AES算法的初始IV。

· scryptParamsJSON：PBKDF2算法的初始化数据。

· MAC：PBKDF2算法的加密结果后16字节与AES算法加密结果通过SHA256算法计算得到的哈希值。

对应的解密过程，即通过解析JSON结构获取两种对称密钥的初始参数，重复运算即可得到明文。代码实现如下：

```
blockchain/pseudohsm/keystore_passphrase.go#L130
func DecryptKey(keyjson []byte, auth string) (*XKey, error)
```

9.4 账户管理

在UTXO模型中钱包层是没有账户的概念的（比如比特币），只有私钥和地址，账户其实是上层的抽象。在比原链的设计中，钱包层设计了账户的概念。账户是一种可以通过追踪控制程序（control program）或者地址（address），来确定资产在区块链上所有权而进行抽象的实体。控制程序或地址会出现在每一笔交易的输出中。每个账户可以拥有多把私钥，私钥的不同形式就组成了不同的账户（因为会有多签账户），因此可以把账户理解成私钥的一种表现形式。每个账户会有无限多的地址，地址是由派生公钥生成的，这样可以很好地保护用户的隐私，因为用户的资产分散在不同的地址当中，只追踪某个地址是不可能追溯到用户的所有资产的。

在本地钱包可以存在多个账户，账户并不会保存到区块链上，而是存在于本地钱包中，只有在账户中创建的控制程序、资产（Asset）或地址（Address）才会出现在区块链上。然而，当处理一笔交易时，依附于本地的账户数据可以提供更加强大且对用户更易于理解的查询。Account结构体示例如下：

```
account/accounts.go
type Account struct {
    *signers.Signer
    ID    string `json:"id"`
    Alias string `json:"alias"`
}
```

Account结构体说明如下：

· *signers.Signer：账户签名。

· ID：账户ID。

· Alias：账户别名。

Signer结构体示例如下：

```
blockchain/signers/signers.go
type Signer struct {
    Type     string          `json:"type"`
    XPubs    []chainkd.XPub `json:"xpubs"`
    Quorum   int             `json:"quorum"`
    KeyIndex uint64          `json:"key_index"`
}
```

Signer结构体说明如下。

· Type：类型，包含account与asset两类。

· XPubs：扩展公钥数组，一个账户根据其类型是单签账户还是多签账户，提供一到多个密钥。

· Quorum：签名数，其值大于0且小于XPubs的长度，表示当使用该账户花费一笔资产且需要签名时，至少需要私钥签名的个数。

· KeyIndex：在数据库中的索引。

9.4.1 账户创建

当用户通过API发起新账户申请时，需要对传入参数公钥集合、签名数、别名进行一系列的处理，最终生成一个新账户。账户的具体生成过程如下：

```
account/accounts.go#L119
func (m *Manager) Create(xpubs []chainkd.XPub, quorum int,
alias string) (*Account, error) {
    // ...
    normalizedAlias :=
strings.ToLower(strings.TrimSpace(alias))
    if existed := m.db.Get(aliasKey(normalizedAlias)); existed
!= nil {
        return nil, ErrDuplicateAlias
    }

    signer, err := signers.Create("account", xpubs, quorum,
m.getNextAccountIndex())
    id := signers.IDGenerate()
    // ...

    account := &Account{Signer: signer, ID: id, Alias:
normalizedAlias}
    rawAccount, err := json.Marshal(account)
    // ...

    accountID := Key(id)
    storeBatch := m.db.NewBatch()
    storeBatch.Set(accountID, rawAccount)
    storeBatch.Set(aliasKey(normalizedAlias), []byte(id))
    storeBatch.Write()
    return account, nil
}
```

创建账户流程如下：

1）对账户别名alias字符串进行转小写、去空格、查重等一系列基本操作。

2）通过signers.Create()方法生成账户签名，此方法主要包含对公钥集合排序以及对签名数合法性进行验证等。

3）通过signers.IDGenerate()方法生成新账户ID，该ID是唯一标识。

4）实例化account对象。

5）将账户信息持久化存储。

6）返回account结构，至此新账户生成流程结束。

9.4.2 账户地址

一个账户可以包含多个地址，每一个地址都是用户资产的归集点。用户进行收款与付款操作都需要一个地址来进行资产的接收与转移。地址是资产索引，知道账户所有已用地址就可以知道此账户的资产总额。

账户所有者在新建地址时，根据账户中包含的公钥个数创建出不同的地址类型。地址可分为两类：一类是对公钥进行哈希得到的P2PKH（Pay-to-Public-Key-Hash）地址，由于此账户地址对应的账户中只有一个公钥，花费其中的资产仅需一个私钥签名即可，所以又称“单签地址”。另一类是对支付脚本进行哈希得到的P2SH（Pay-to-Script-Hash），由于其账户地址对应的账户中含有多个公钥集合，花费其中的资产往往需要多个私钥签名才能通过，所以又称“多签地址”。

1. 创建P2PKH单签地址

单签地址P2PKH的创建流程示例如下：

```
bytom/account/accounts.go#L433
func (m *Manager) createP2PKH(account *Account, change bool)
(*CtrlProgram, error) {
    idx := m.getNextContractIndex(account.ID)
    path := signers.Path(account.Signer,
signers.AccountKeySpace, idx)
    derivedXPubs := chainkd.DeriveXPubs(account.XPubs, path)
    derivedPK := derivedXPubs[0].PublicKey()
    pubHash := crypto.Ripemd160(derivedPK)

    address, err := common.NewAddressWitnessPubKeyHash(pubHash,
&consensus.ActiveNetParams)
    // ...

    control, err := vmutil.P2WPKHProgram([]byte(pubHash))
    // ...

    return &CtrlProgram{
        AccountID:      account.ID,
        Address:        address.EncodeAddress(),
        KeyIndex:       idx,
```

```
        ControlProgram: control,
        Change:         change,
    }, nil
}
```

以上代码的主要逻辑为：

1）根据账户ID更新ContractIndex索引自增1，即为新增加的账户地址序号。每创建一个地址都会自增1。

2）根据账户签名、账户密钥空间及新地址索引，获取新公钥生成路径（分层钱包层次）以供新的公钥派生使用（比如当前账户在HD钱包路径中处于第3层，目前有5个地址，这几个地址都是由下层即第4层子密钥通过运算得到，现在要为账户生成一个新地址，那么新地址所处的路径应该是第4层第6个地址）。signers.Path方法的返回值是一个数组，传入多个索引就会返回多个对应路径。

3）根据账户的扩展公钥信息以及新公钥派生路径，派生扩展公钥。

4）由于是单签地址，派生出的子公钥数组长度为1，取此扩展子密钥的前32字节，经ED25519算法得到改进的公钥。

5）针对获取到的公钥，通过ripemd160算法得到新公钥的哈希值，即公钥哈希。

6）方法common.NewAddressWitnessPubKeyHash()根据新公钥哈希和网络参数（根据当前网络MainNet，TestNet，SoloNet，分别生成带有不同前缀的地址bm，tm，sm），生成支持隔离见证的公钥哈希地址P2PKH。

7）根据公钥哈希生成BVM虚拟机执行的脚本控制程序。

8）构建CtrlProgram，其中Address是通过bech32编码的，在比原链中引入bech32编码主要为了支持隔离见证。

2.创建P2SH多签地址

多签地址P2SH的创建流程示例如下：

```
bytom/account/accounts.go#L459
func (m *Manager) createP2SH(account *Account, change bool)
(*CtrlProgram, error) {
    idx := m.getNextContractIndex(account.ID)
    path := signers.Path(account.Signer,
signers.AccountKeySpace, idx)
    derivedXPubs := chainkd.DeriveXPubs(account.XPubs, path)
    derivedPKs := chainkd.XPubKeys(derivedXPubs)
    signScript, err := vmutil.P2SPMultiSigProgram(derivedPKs,
account.Quorum)
    // ...
    scriptHash := crypto.Sha256(signScript)

    address, err :=
common.NewAddressWitnessScriptHash(scriptHash,
&consensus.ActiveNetParams)
    // ...

    control, err := vmutil.P2WSHProgram(scriptHash)
    // ...

    return &CtrlProgram{
        AccountID:      account.ID,
        Address:        address.EncodeAddress(),
        KeyIndex:       idx,
        ControlProgram: control,
        Change:         change,
    }, nil
}
```

以上代码的主要逻辑为：

1）根据account.ID更新ContractIndex索引自增1，即为新增加的账户地址序号。每创建一个地址都会自增1。

2）根据账户签名、账户密钥空间及新地址索引，获取新公钥生成路径（分层钱包层次）以供新的公钥派生使用。

3）根据账户的扩展公钥信息以及新公钥派生路径，派生扩展公钥。

4）由于是多签地址，派生多个经ED25519（椭圆加密算法）加密算法改进格式的扩展子公钥。

5）根据新派生的子公钥和多签地址最少签名人数等信息，通过vmutil.P2SPMultiSig-Program()生成多签控制程序脚本。

6）针对多签控制程序脚本，经SHA256运算获取脚本哈希值。

7）根据脚本哈希和当前网络参数，方法common.NewAddressWitnessScriptHash()生成支持隔离见证的多签地址。

8）根据脚本哈希获取BVM虚拟机执行的控制程序。

9）构建CtrlProgram，其中Address是通过bech32编码的，在比原链中引入bech32编码主要为了支持隔离见证，参见[1]BTC的SegWit扩展解决方案。

从两种地址的生成过程可以看出，最终生成的地址参与了账户控制程CtrlProgram结构的构造。把新生成的CtrlProgram对象存入本地LevelDB数据库中，其中插入记录的key为CtrlProgram经过SHA256算法得到的哈希值与前缀"Contract:"拼接后的字符串，记录的Value为CtrlProgram的JSON格式数据，这种存储形式为以后从区块网络中进行本账户交易归集提供了极大的便利，当要确认新的交易是否与自己相关时，就可以取出交易的Outputs中CtrlProgram进行上述Key的拼接，如果能够以此Key从本地帐户表中查询到任何记录，即代表此交易是与我相关的。

[1] 参见网址为https://en.wikipedia.org/wiki/SegWit。

9.4.3 账户余额

账户余额本身也是一个为了便于用户理解而抽象的概念，它表示使用了该账户的地址或控制程序的UTXO中所有资产的总和。当构建一笔资产转账交易时，账户会使用足够数量的UTXO来作为新交易的输入。在wallet数据库中，会将账户的UTXO单独持久化存储。统计余额只需要将账户的UTXO的Amount资产数量相加即可。代码示例如下：

```
wallet/indexer.go
func (w *Wallet) GetAccountBalances(id string)
([]AccountBalance, error) {
    return w.indexBalances(w.GetAccountUtxos("", false, false))
}
```

首先，w.GetAccountUtxos根据UTXOPreFix前缀，查找LevelDB数据库下的所有UTXO。之后根据账户、资产分类，w.indexBalances将所有的UTXO的Amount资产数量相加，得到该账户当前钱包下所有资产的余额。

9.5 资产管理

比原链是多资产管理模型，资产模块主要进行资产的创建、展示和管理。比原链支持多资产的交互，任何人都可以发行自己的资产，然后通过资产管理个人或者组织机构下的资产。Asset结构体示例如下：

```
asset/asset.go
type Asset struct {
    *signers.Signer
    AssetID           bc.AssetID
    Alias             *string
    VMVersion         uint64
    IssuanceProgram   chainjson.HexBytes
    RawDefinitionByte chainjson.HexBytes
    DefinitionMap     map[string]interface{}
}
```

Asset结构体说明如下。

- AssetID：资产ID。
- Alias：资产别名。
- VMVersion：版本号。
- IssuanceProgram：发布程序。
- RawDefinitionByte：字节形式的资产定义信息。

· DefinitionMap：结构化的资产定义信息。

9.5.1 初始默认资产

默认资产是节点第一次启动时创建的管理BTM代币的资产。首先介绍比原链默认资产是如何定义和初始化的，代码示例如下：

```
asset/asset.go#L39
var DefaultNativeAsset *Asset
func initNativeAsset() {
    signer := &signers.Signer{Type: "internal"}
    alias := consensus.BTMAlias

    definitionBytes, _ :=
serializeAssetDef(consensus.BTMDefinitionMap)
    DefaultNativeAsset = &Asset{
        Signer:            signer,
        AssetID:           *consensus.BTMAssetID,
        Alias:             &alias,
        VMVersion:         1,
        DefinitionMap:     consensus.BTMDefinitionMap,
        RawDefinitionByte: definitionBytes,
    }
}
```

以上代码的主要逻辑为：

1）生成一个类型为“internal”类型的签名。

2）资产别名取默认值BTM。

3）资产定义信息，默认资产定义的内容——name：BTM，symbol：BTM，decimals：8，description：`Bytom Official Issue`。

4）在internal类型签名、资产别名、资产定义字节形式已经获取的基础上，版本号固定值为1，加上资产ID，至此形成一个Asset的资产结构体。

默认资产ID生成，代码示例如下：

```
/bytom/consensus/general.go#L42
var BTMAssetID = &bc.AssetID{
    V0: binary.BigEndian.Uint64([]byte{0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff}),
    V1: binary.BigEndian.Uint64([]byte{0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff}),
    V2: binary.BigEndian.Uint64([]byte{0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff}),
    V3: binary.BigEndian.Uint64([]byte{0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff}),
}
```

默认资产ID以16进制显示为（共64字节）：ffffffff...ffffffff。

9.5.2 发行资产

默认资产是用于管理BTM代币的资产。我们可以通过比原链发行自己的资产，并成为资产上链。比原链支持BUTXO模型，对多种资产交易提供支持，用户可以方便地定义新的资产类型，并可以在网络上自由流通。代码示例如下：

```
bytom/api/assets.go#L15
func (a *API) createAsset(ctx context.Context, ins struct {
    Alias           string
    RootXPubs       []chainkd.XPub
    Quorum          int
    Definition      map[string]interface{}
    IssuanceProgram chainjson.HexBytes
}) Response {}
```

通过调用/create-asset接口来创建资产，传入字段主要包括资产别名、公钥集合、支配资产最少签名数、资产定义信息、资产发行程序等。在接到新资产定义信息后，钱包层根据用户信息定义新资产，流程示例如下：

```
bytom/asset/asset.go#L129
func (reg *Registry) Define(xpubs []chainkd.XPub, quorum int,
definition map[string]interface{}, alias string,
issuanceProgram chainjson.HexBytes) (*Asset, error) {
    // ...
    vmver := uint64(1)
    if len(issuanceProgram) == 0 {
        // ...

        nextAssetIndex := reg.getNextAssetIndex()
        assetSigner, err = signers.Create("asset", xpubs,
quorum, nextAssetIndex)
        // ...
        path := signers.Path(assetSigner,
signers.AssetKeySpace)
        derivedXPubs := chainkd.DeriveXPubs(assetSigner.XPubs,
path)
        derivedPKs := chainkd.XPubKeys(derivedXPubs)
        issuanceProgram, vmver, err =
```

```
multisigIssuanceProgram(derivedPKs, assetSigner.Quorum)
        // ...
    }

    defHash := bc.NewHash(sha3.Sum256(rawDefinition))
    a := &Asset{
        DefinitionMap:     definition,
        RawDefinitionByte: rawDefinition,
        VMVersion:         vmver,
        IssuanceProgram:   issuanceProgram,
        AssetID:           bc.ComputeAssetID(issuanceProgram,
vmver, &defHash),
        Signer:            assetSigner,
        Alias:             &alias,
    }
    return a, reg.SaveAsset(a, alias)
}
```

以上代码的主要逻辑为：

1）对传入的资产别名进行去空格转大写的格式处理，不允许别名为空，也不允许别名使用BTM这个保留字。

2）对资产定义信息进行序列化，主要是转成JSON格式并以字节形式返回。

如果未提供发行资产程序（issuanceProgram），则按以下步骤构造发行资产程序。

a）为新的资产构建索引。

b）根据公钥数组、签名数和新的资产索引信息构建一个资产类型的签名。

c）获取派生密钥的完整路径。

d）根据扩展公钥集合和密钥完整路径派生出子公钥集合。

e）通过Ed25519签名算法改进子公钥集合。

f）根据新派生的子公钥和签名数量，通过P2SPMultiSigProgram构建新发行资产程序。

3）对生成的资产定义信息（rawDefinition）进行SHA3-256运算，获取资产定义信息的哈希值。

4）实例化Asset对象，根据发行程序、VM版本号以及资产定义哈希值运算获得AssetID。

5）存储资产信息并返回。通过SaveAsset方法，以“Asset：”+资产ID拼接成的字符串作为Key，以JSON结构的资产信息作为Value存入LevelDB，同时以"AssetAlias："+资产别名拼接成的字符串作为Key，以资产ID字节形式作为Value存入LevelDB，方便以后通过资产别名来查询资产。

9.6 交易管理

交易模块管理与自己相关的交易数据。在区块链上每天可能会有无数笔的交易，但其中可能只有几笔是与自己相关的，交易模块会在本地将与自己相关的交易筛选出来，维护钱包层的UTXO的数据库，记录本人所拥有的UTXO。钱包层的UTXO和内核中记录的UTXO不同，内核记录了全网的UTXO，但并没有记录UTXO的详细数据，只是记录了哈希；而钱包层记录了与这个UTXO相关的详细数据，包括与哪个账户相关、在哪个区块产生出来、输入输出等。

无论从块中筛选与自己相关的交易还是记录本人所拥有的UTXO，这两个动作都与钱包的更新过程密切相关，钱包对象Wallet的AttachBlock方法包含了这两个过程，代码示例如下：

```
bytom/wallet/wallet.go#L92
func (w *Wallet) AttachBlock(block *types.Block) error {
    // ...
    w.indexTransactions(storeBatch, block, txStatus)
    w.attachUtxos(storeBatch, block, txStatus)
    // ...
}
```

这两行代码即从传入的block块中筛选属于自己的交易，统计自己的UTXO。

9.6.1 筛选交易

筛选时将交易分为两类：一类是交易的输出指向“我”，即收款人是“我”的交易。另一类是交易的输入指向“我”，即付款人是“我”的交易。筛选过程代码示例如下：

```
/bytom/wallet/indexer.go#L120
func (w *Wallet) filterAccountTxs(b *types.Block, txStatus
*bc.TransactionStatus) []*query.AnnotatedTx {
    annotatedTxs := make([]*query.AnnotatedTx, 0,
len(b.Transactions))

transactionLoop:
    for pos, tx := range b.Transactions {
        statusFail, _ := txStatus.GetStatus(pos)
        for _, v := range tx.Outputs {
            var hash [32]byte
            sha3pool.Sum256(hash[:], v.ControlProgram)
            if bytes := w.DB.Get(account.ContractKey(hash));
bytes != nil {
                annotatedTxs = append(annotatedTxs,
w.buildAnnotatedTransaction (tx, b, statusFail, pos))
                continue transactionLoop
            }
        }

        for _, v := range tx.Inputs {
            outid, err := v.SpentOutputID()
            if err != nil {
                continue
            }
            if bytes :=
w.DB.Get(account.StandardUTXOKey(outid)); bytes != nil {
                annotatedTxs = append(annotatedTxs,
w.buildAnnotatedTransaction (tx, b, statusFail, pos))
                continue transactionLoop
            }
        }
    }

    return annotatedTxs
}
```

以上代码的主要逻辑为：

1）遍历块中的所有交易。

2）对每一笔交易的输出Outputs进行遍历，将Outputs中的ControlProgram经SHA3-256算法得出的哈希值拼上前缀字符串"Contract："形成查询字段，在Wallet数据库中通过此字段进行查询，如果能够查询到记录，说明这笔交易中有输出地址指向我的收款地址，此交易是与我相关的交易，放入结果集。

3）对每一笔交易的输入Inputs进行遍历，将Inputs中的SpendOutputID拼上标准UTXO前缀"ACU："形成查询字段，在Wallet数据库中通过此字段进行查询，如果能查询到记录，说明这个要消费的UTXO是我的，此交易是与我相关的交易，放入结果集。

4）返回annotatedTxs结果集合，即此块中所有与我相关的交易。

5）在filterAccountTxs方法中筛选所有与我相关的交易后，indexTransactions方法将所有查找到的结果序列化为JSON格式，存入本地钱包数据库。

9.6.2 筛选UTXO

对UTXO的筛选可以通过两类操作执行：一是从本地数据库中删除已经消费的UTXO；二是把支付给我的UTXO存储起来。经此两类操作，本地数据库中存储的UTXO都是我所拥有的。筛选过程示例代码如下：

```
/bytom/wallet/utxo.go#L43
func (w *Wallet) attachUtxos(batch db.Batch, b *types.Block,
txStatus *bc.TransactionStatus) {
    for txIndex, tx := range b.Transactions {
        statusFail, err := txStatus.GetStatus(txIndex)
        // ...

        inputUtxos := txInToUtxos(tx, statusFail)
        for _, inputUtxo := range inputUtxos {
            if segwit.IsP2WScript(inputUtxo.ControlProgram) {

batch.Delete(account.StandardUTXOKey(inputUtxo.OutputID))
            } else {

batch.Delete(account.ContractUTXOKey(inputUtxo.OutputID))
            }
        }

        validHeight := uint64(0)
        if txIndex == 0 {
            validHeight = b.Height +
consensus.CoinbasePendingBlockNumber
        }
        outputUtxos := txOutToUtxos(tx, statusFail,
validHeight)
        utxos := w.filterAccountUtxo(outputUtxos)
        if err := batchSaveUtxos(utxos, batch); err != nil {
            log.WithField("err", err).Error("attachUtxos fail
on batchSaveUtxos")
        }
    }
}
```

以上代码的主要逻辑分为：

1）遍历所有交易，剔除交易状态有问题的交易。

2）优先根据交易输入中account.UTXO的ControlProgram，判定是不是自己花费的UTXO，如果是则直接从本地数据库中删除相应的UTXO记录。

3）判断交易输出，通过filterAccountUtxo方法过滤所有输出的UTXO的CtrlProgram，如果根据CtrlProgram经SHA3-256哈希后拼接对应前缀而形成的查询字段，能够从本地数据库中查到记录，并能成功反序列化为JSON，则认为这笔交易中的UTXO是我的，加入结果集。

4）把所有找到的UTXO序列化为JSON结构，存入数据库。

经上述步骤，当一个新区块被广播到钱包节点，新块中所有交易会被梳理一遍，从本地数据库中剔除花费的UTXO，增加收到的UTXO，保证钱包账户余额的准确性。

9.6.3 UTXO花费选择算法

现实生活中当我们与商家达成一笔交易时，从个人钱包中掏出一把钱，有100元、50元、20元等，金额不定。假设我们在商家购买了70元的商品，在现实生活中我们会选择使用50元和20元面额的钞票进行支付。在UTXO中同样，我们优先使用较小面额支付。optUTXOs函数实现了对可用的UTXO进行选择，尽量使用多笔小额来代替单笔大额，以减少残存UTXO的数量，避免UTXO越来越“散”。

假设我们从钱包中支出9块钱，那么UTXO选择流程如图9-3所示。

图9-3 UTXO花费选择算法

optUTXOs函数实现在account/utxo_keeper.go，代码示例如下。

1）将所有的UTXO按金额由大到小进行排序：

```
var optAmount, reservedAmount uint64
sort.Slice(utxos, func(i, j int) bool {
    return utxos[i].Amount > utxos[j].Amount
})
```

2）将utxos（可用的UTXO）push到list中，进行slice和list之间的数据结构类型转换，最后得到可用的UTXO列表utxoList，其中的utxo按面值从大到小的顺序排列：

```
utxoList := list.New()
for _, u := range utxos {
    if _, ok := uk.reserved[u.OutputID]; ok {
        reservedAmount += u.Amount
        continue
    }
    utxoList.PushBack(u)
}
```

3）遍历utxoList中的所有UTXO。第一步，从面值大的开始选择UTXO，直到所有选到的UTXO面值累加金额大于等于要花费的金额amount；第二步，从已经选定的UTXO列表中尝试取出最大面值，用更小面值进行替换，如果替换后的合计金额仍然大于目标金额，继续拿更小面值替换最大面值，直到选到的UTXO集合大于等于5（desireUtxoCount=5，比原链权衡性能，当选定UTXO数超过5就不继续优化），且合计金额大于等于目标金额，或无小面值可以替换最大面值：

```
for node := utxoList.Front(); node != nil; node = node.Next() {
        //append utxo if we haven't reached the required amount
        if optAmount < amount {
            optList.PushBack(node.Value)
            optAmount += node.Value.(*UTXO).Amount
continue
        }
```

```
        largestNode := optList.Front()
        replaceList := list.New()
        replaceAmount := optAmount - largestNode.Value.
(*UTXO).Amount //replace = 15-8=7

        for ; node != nil && replaceList.Len() <=
desireUtxoCount-optList.Len(); node = node.Next() {
            replaceList.PushBack(node.Value)
            if replaceAmount += node.Value.(*UTXO).Amount;
replaceAmount >= amount {
                optList.Remove(largestNode)
                optList.PushBackList(replaceList)
                optAmount = replaceAmount
                break
            }
        }

        //largestNode remaining the same means that there is
nothing to be replaced
        if largestNode == optList.Front() {
            break
        }
    }
```

4）返回最终选定的结果集，结果集合计金额大于等于目标金额。示例代码如下：

```
optUtxos := []*UTXO{}
for e := optList.Front(); e != nil; e = e.Next() {
    optUtxos = append(optUtxos, e.Value.(*UTXO))
}
```

9.7 钱包管理

9.7.1 数据更新

钱包的更新是交易与UTXO信息归集的过程。Wallet对象实例化时，启动一个goroutine等待信号，触发信号遍历所有本地区块，检索其中符合条件的交易和UTXO进行归集，其过程是遍历所有区块找到与我相关的交易信息，并对本地的wallet数据库进行相应的更新。

钱包Wallet对象中有一个字段rescanCh，该字段是将通知钱包数据更新的信号作为开关，当rescanCh收到信号时，钱包会从高度为0的块开始重新扫描遍历所有的块，重新进行钱包的交易归集、资产归集等一系列数据更新。代码示例如下：

```
bytom/wallet/wallet.go#L185
func (w *Wallet) getRescanNotification() {
    select {
    case <-w.rescanCh:
        w.setRescanStatus()
    default:
        return
    }
}

func (w *Wallet) setRescanStatus() {
    block, _ := w.chain.GetBlockByHeight(0)
    w.status.WorkHash = bc.Hash{}
    w.AttachBlock(block)
}
```

与钱包更新相关的方法walletUpdater()和walletBlockWaiter()会触发rescanCh信号的操作，setRescanStatus开始遍历本地区块，并更新钱包数据。

9.7.2 备份

账户资产安全是非常重要的，关系到每个账户所有人的利益，其中一种保护账户资产的方式是定期进行钱包备份。钱包备份的信息主要分为加密机、资产、账户三大类。代码示例如下：

```
/bytom/api/wallet.go#L38
func (a *API) backupWalletImage() Response {
    keyImages, err := a.wallet.Hsm.Backup()
    // ...
    assetImage, err := a.wallet.AssetReg.Backup()
    // ...

    accountImage, err := a.wallet.AccountMgr.Backup()
    // ...

    image := &WalletImage{
        KeyImages:    keyImages,
        AssetImage:   assetImage,
        AccountImage: accountImage,
    }
    return NewSuccessResponse(image)
}
```

以上代码的最终目的是形成钱包的备份镜像，主要包含密钥库的镜像、资产的镜像、账户的镜像。主要逻辑为：

1）加密机进行密钥库的备份，最终形成JSON格式的数据，其结构参见bytom/blockchain/pseudohsm/key.go下的encryptedKeyJSON结构。

2）资产数据备份，获取数据库中所有以"Asset："开头的记录，序列化为JSON格式的数据，其结构参见/bytom/asset/asset.go下的Asset struct结构。

3）账户数据备份，获取数据库中所有以”Account：”开头的记录，序列化为JSON格式的数据，其结构参见/bytom/account/image.go下的ImageSlice结构。

完成以上操作步骤，即可得到钱包的备份镜像信息，完成备份。

9.7.3 恢复

当使用中的钱包发生问题，或者在新的环境中希望直接恢复钱包信息，可将备份好的钱包镜像重新导入，恢复钱包数据，代码示例如下：

```
/bytom/api/wallet.go#L24
func (a *API) restoreWalletImage(ctx context.Context, image
WalletImage) Response {
    if err := a.wallet.Hsm.Restore(image.KeyImages); err != nil
{
        return NewErrorResponse(errors.Wrap(err, "restore key
images"))
    }
    if err := a.wallet.AssetReg.Restore(image.AssetImage); err
!= nil {
        return NewErrorResponse(errors.Wrap(err, "restore asset
image"))
    }
    if err := a.wallet.AccountMgr.Restore(image.AccountImage);
err != nil {
        return NewErrorResponse(errors.Wrap(err, "restore
account image"))
    }
    a.wallet.RescanBlocks()
    return NewSuccessResponse(nil)
}
```

以上代码可以对照9.7.2节中的内容（正好是其逆操作），主要逻辑为：

1）从镜像文件恢复密钥库。

2）从镜像文件恢复资产信息。

3）从镜像文件恢复账户信息。

4）检测是否需要重构钱包。

5）完成整个钱包的恢复。

9.8 本章小结

本章基于钱包层涉及的密钥管理、账户管理、资产管理、交易管理等模块，介绍了HD-Wallet分层确定性钱包的特性及相关算法，Account账户模型以及地址的产生，新资产的定义以及钱包账户相关的交易归集等诸多方面的知识。对Wallet钱包对象的更新以及钱包的备份与恢复操作进行了分析。

钱包由于涉及资产的安全性以及使用的方便性，在各个环节穿插了许多密码学相关知识，比如非对称加密，HSM、HMAC，以及种类多样的算法，如ED25519椭圆加密算法、SHA512、SHA256、Ripemd160等，还有与BTC相关的各种提案，如BIP32、BECH32等。读者如果希望对这些知识进行更深入的了解，可以参考现代密码学相关的书籍和资料。

第10章

P2P分布式网络

10.1 概述

本章内容涉及区块链底层技术实现，即P2P分布式网络原理。在设计公链时，节点与节点之间建立连接需要P2P协议，从而实现数据的同步。上层需要封装一些通信逻辑，比如节点之间的区块同步、交易同步等，在比原链中，SyncManager对象则承担该角色。

本章主要内容如下：

- P2P网络通信的模型初始化过程。
- 节点发现机制及实现。
- 节点状态机。
- 区块同步和交易同步。
- 快速广播及节点管理。

10.2　P2P的四种网络模型

P2P网络不同于传统的客户端/服务端（C/S）结构，P2P网络中的每个节点都可以既是客户端也是服务端，因此不适合使用HTTP协议进行节点之间的通信，且一般都是直接使用Socket进行网络编程。[1]

P2P主要存在四种不同的网络模型，代表着P2P技术的四个发展阶段：集中式，分布式，混合式，结构化网络。区块链技术本身是一个可靠的分布式网络，结构化网络模型是本章重点讲解内容。

1. 集中式

集中式是P2P网络模式中最简单的路由方式，即存在一个中心节点，它保存了其他所有节点的索引信息，索引信息一般包括节点IP、端口、节点资源等。集中式路由的优点是结构简单、实现容易。但缺点也很明显，由于中心节点需要存储所有节点的路由信息，当节点规模扩展时，就很容易出现性能瓶颈；而且也存在单点故障问题。

2. 分布式

分布式结构是指，移除了中心节点，在P2P节点之间建立随机网络。在新加入节点与P2P网络中的某个节点之间随机建立连接通道，从而形成一个随机拓扑结构。新节点加入该网络时随机选择一个已经存在的节点并建立邻居关系。

新节点与邻居节点建立连接后，还需要进行全网广播，让整个网络知道该节点的存在。全网广播的步骤是，该节点首先向邻居节点广播，邻居节点收到广播消息后，再继续向自己的邻居节点广播，以此类推，从而将消息广播到整个网络。这种广播方法也称为泛洪机制。

分布式结构不存在集中式结构的单点性能瓶颈问题和单点故障问题，具有较好的可扩展性。

泛洪机制的问题主要是可控性差，具体讲包括两个较大的问题：一个问题是容易形成泛洪循环，比如节点A发出的消息经过节点B到节点C，节点C再广播到节点A，这就形成了一个循环；另一个问题是响应消息风暴，比如节点A想请求的资源被很多节点所拥有，那么在很短时间内，会出现大量节点同时向节点A发送响应消息，这就可能会让节点A瞬间瘫痪。

消除泛洪循环问题的方法可以借鉴IP网络路由协议中有关泛洪广播的控制，一种方法是对每个查询消息设置TTL值，泛洪消息每被转发一次，TTL值减1，当节点接受的TTL为0时，不再转发消息，这样可以避免查询消息在网络中产生死循环。还可以为泛洪消息设置唯一的标志，对接收到的重复消息不进行转发以防止消息产生死循环。解决响应消息风暴问题，一般会在数据链路层进行网络分段，减少消息跨段广播。

3.混合式

混合式其实就是混合集中式和分布式结构。网络中存在多个超级节点组成的分布式网络，而每个超级节点有多个普通节点与它组成的局部集中式网络。一个新的普通节点加入，先选择一个超级节点进行通信，该超级节点再推送其他超级节点列表给新加入节点，加入节点根据列表中的超级节点状态决定选择哪个具体的超级节点作为父节点。这种结构的泛洪广播只是发生在超级节点之间，因此可以避免大规模泛洪问题。在实际应用中，混合式结构是相对灵活且比较有效的组网架构，实现难度也相对较小，因此目前较多系统基于混合式结构进行开发实现。

4.结构化网络

结构化P2P网络是一种分布式网络结构，但与上面所讲的分布式结构不同。分布式网络就是一个随机网络，而结构化网络则将所有节点按照某种结构进行有序组织，比如形成一个环状网络或树状网络。结构化网络在具体实现上，普遍基于分布式哈希表（Distributed Hash Table，DHT）算法。具体的实现方案有Chord、Pastry、CAN、Kademlia（简写Kad）等算法。

四种网络结构的对比见表10-1。

表10-1　几种网络模型对比

分类	集中式	分布式	混合式	结构化
可扩展性	差	差	中	好
可靠性	差	好	中	好
可维护性	最好	最好	中	好
节点发现效率	最高	中	中	高

从表10-1中的对比可以看出，结构化网络最优。目前比原链、以太坊都是基于Kademlia算法实现结构化网络。具体实现方式，我们将在后面10.4.2节中从代码角度逐步分析。

[1] 参见《P2P对等网络原理与应用》，作者：蔡康等。

10.3 网络节点初始化

在比原链中，P2P通信模块主要由SyncManager管理，SyncManager负责节点业务层信息的同步工作，即区块和交易信息的同步。P2P Switch实现底层的DHT分布式网络架构、节点发现及节点间的加密通信等。本小节介绍P2P Switch和SyncManager的初始化流程。

10.3.1 SyncManager初始化

1）SyncManager结构参见如下代码：

```
netsync/handle.go
type SyncManager struct {
    sw           *p2p.Switch
    genesisHash bc.Hash

    privKey      crypto.PrivKeyEd25519 // local node's p2p key
    chain        Chain
    txPool       *core.TxPool
    blockFetcher *blockFetcher
    blockKeeper  *blockKeeper
    peers        *peerSet

    newTxCh    chan *types.Tx
    newBlockCh chan *bc.Hash
    txSyncCh   chan *txSyncMsg
    quitSync   chan struct{}
    config     *cfg.Config
}
```

结构体字段说明如下。

· sw：switch对象，管理整个P2P模块。负责与对等节点的加密连接，将对等节点收到的信息流分给不同的Reactor进行处理。这里的Reactor可以理解为动作。

· genesisHash：创世区块的Hash值。

· privKey：节点生成的随机私钥，用于节点与节点之间的加密连接（RLPx加密握手协议）。

· chain：本地主链。

· txPool：交易池。

· blockFetcher：接收、验证并处理对等节点同步过来的区块。

· blockKeeper：向对等节点请求同步区块（包括快速同步和普通同步）。

· peers：存储所有对等节点的信息。

· newTxCh：快速广播交易。当交易池添加一笔交易时需快速广播给当前节点已知的其他节点。通道大小为10000。

· newBlockCh：快速广播区块。当前节点挖掘出了新的区块时需快速广播给当前已知的其他节点。通道大小为1024。

· txSyncCh：接收、验证并处理对等节点同步过来的交易。

· quitSync：节点退出时通知SyncManager提前退出。

· config：节点的全局配置。

2）初始化SyncManager的代码如下：

```
netsync/handle.go
func NewSyncManager(config *cfg.Config, chain Chain, txPool
*core.TxPool, newBlockCh chan *bc.Hash) (*SyncManager, error) {
    genesisHeader, err := chain.GetHeaderByHeight(0)
    // ...

    sw := p2p.NewSwitch(config)
    peers := newPeerSet(sw)
    manager := &SyncManager{
        sw:          sw,
        genesisHash: genesisHeader.Hash(),
        txPool:      txPool,
        chain:       chain,
        privKey:     crypto.GenPrivKeyEd25519(),
        blockFetcher: newBlockFetcher(chain, peers),
        blockKeeper: newBlockKeeper(chain, peers),
        peers:       peers,
        newTxCh:     make(chan *types.Tx, maxTxChanSize),
        newBlockCh:  newBlockCh,
```

```
        txSyncCh:     make(chan *txSyncMsg),
        quitSync:     make(chan struct{}),
        config:       config,
    }

    protocolReactor := NewProtocolReactor(manager,
manager.peers)
    manager.sw.AddReactor("PROTOCOL", protocolReactor)

    // Create & add listener
    var listenerStatus bool
    var l p2p.Listener
    if !config.VaultMode {
        p, address :=
protocolAndAddress(manager.config.P2P.ListenAddress)
        l, listenerStatus = p2p.NewDefaultListener(p, address,
manager.config.P2P.SkipUPNP)
        manager.sw.AddListener(l)

        discv, err := initDiscover(config, &manager.privKey,
l.ExternalAddress().Port)
        // ...
        manager.sw.SetDiscv(discv)
    }

manager.sw.SetNodeInfo(manager.makeNodeInfo(listenerStatus))
    manager.sw.SetNodePrivKey(manager.privKey)
    return manager, nil
}
```

SyncManager在初始化时需要完成诸多的工作：

1）初始化P2P switch。

2）初始化peerSet，存储对等节点的信息，使用键值结构存储，保证不会有重复的对等节点信息。

3）实例化SyncManager，privKey字段使用crypto.GenPrivKeyEd25519()方法生成节点的私钥，用于节点与对等节点加密连接。

4）初始化blockFetcher。

5）初始化blockKeeper。内部会启动一个syncWorker的goroutine，用于定时同步区块。

6）添加switch Reactor动作，当与对等节点握手成功后，发送当前节点的状态信息。而且当握手成功后，将当前节点中交易池中未打包的交易同步给对等节点。

7）初始化listenser，绑定TCP端口监听，用于监听对等节点的连接和数据信息交换。

8）初始化Discover节点，通过种子节点和kad算法完成节点发现，建立节点P2P网络拓扑。

9）sw.SetNodePrivKey根据privKey生成节点的PubKey公钥。

SyncManger启动过程代码示例如下：

```
netsync/handle.go
func (sm *SyncManager) Start() {
    if _, err := sm.sw.Start(); err != nil {
        cmn.Exit(cmn.Fmt("fail on start SyncManager: %v", err))
    }
    // broadcast transactions
    go sm.txBroadcastLoop()
    go sm.minedBroadcastLoop()
    go sm.txSyncLoop()
}
```

SyncManager在启动时需启动多个goroutine，以便快速同步节点新产生的交易和区块。启动操作如下：

1）启动switch。

2）txBroadcastLoop：当接收到新的交易时，快速广播给已知的对等节点。

3）minedBroadcastLoop：当节点挖掘出新区块时，快速广播给已知的对等节点。

4）txSyncLoop：接收、验证并处理对等节点同步过来的交易。

10.3.2 P2P Switch初始化

1）P2P Switch结构参见如下代码：

```
p2p/switch.go
type Switch struct {
    cmn.BaseService

    Config       *cfg.Config
    peerConfig   *PeerConfig
    listeners    []Listener
    reactors     map[string]Reactor
    chDescs      []*connection.ChannelDescriptor
    reactorsByCh map[byte]Reactor
    peers        *PeerSet
    dialing      *cmn.CMap
    nodeInfo     *NodeInfo              // our node info
    nodePrivKey  crypto.PrivKeyEd25519 // our node privkey
    discv        *discover.Network
    bannedPeer   map[string]time.Time
    db           dbm.DB
    mtx          sync.Mutex
}
```

结构体字段说明如下。

· Config：节点的全局配置。

· peerConfig：与对等节点连接的配置，包括连接超时、握手超时时间、连接管理等配置。

· listeners：端口监听的listener。

· reactors：将对等节点收到的信息流分给不同的Reactor进行处理。

· chDescs：channel配置信息。

· reactorsByCh：存储channel和reactor的对应关系。

· peers：存储完成握手过程的对等节点信息。

· dialing：线程安全的map结构，存储正在拨号的对等节点信息。

· nodeInfo：当前节点的信息。

· nodePrivKey：当前节点的私钥。

· discv：管理Kademlia算法与各种协议之间的交互。

· bannedPeer：异常节点黑名单，存储异常对等节点。

· db：存放网络中恶意节点的信息，网络中存在一些伪造区块信息的节点，这类节点会被当前节点拉黑，并写入trusthistory.db数据库中。

2）初始化Switch的代码如下：

```
p2p/switch.go
func NewSwitch(config *cfg.Config) *Switch {
    sw := &Switch{
        Config:       config,
        peerConfig:   DefaultPeerConfig(config.P2P),
        reactors:     make(map[string]Reactor),
        chDescs:      make([]*connection.ChannelDescriptor, 0),
        reactorsByCh: make(map[byte]Reactor),
        peers:        NewPeerSet(),
        dialing:      cmn.NewCMap(),
        nodeInfo:     nil,
        db:           dbm.NewDB("trusthistory",
config.DBBackend, config.DBDir()),
    }
    sw.BaseService = *cmn.NewBaseService(nil, "P2P Switch", sw)
    sw.bannedPeer = make(map[string]time.Time)
    if datajson := sw.db.Get([]byte(bannedPeerKey)); datajson
!= nil {
        if err := json.Unmarshal(datajson, &sw.bannedPeer); err
!= nil {
            return nil
        }
    }
```

```
    trust.Init()
    return sw
}
```

P2P Switch在初始化时需要完成诸多的工作：

· 实例化Switch对象。

· 初始化trusthistory.db数据库。从数据库中读取前缀为bannedPeerKey的所有数据，并加载到sw.hannedPeer对象中。我们会在后面10.10节详细讲解如何在区块链网络中对节点进行评分，如何识别恶意节点。

· trust.Init初始化动态节点评分。

P2P Switch启动过程代码示例如下：

```
p2p/switch.go
func (sw *Switch) OnStart() error {
    for _, reactor := range sw.reactors {
        if _, err := reactor.Start(); err != nil {
            return err
        }
    }
    for _, listener := range sw.listeners {
        go sw.listenerRoutine(listener)
    }
    go sw.ensureOutboundPeersRoutine()
    return nil
}
```

Switch启动时分三步：

1）启动switch中注册的Reactor，用于处理节点接收到的数据。

2）启动sw.listenerRoutine的goroutine，监听端口上的连接，与已经连接上的节点完成密钥交换和握手协议，并添加到PeerSet中。

3）启动sw.ensureOutboundPeersRoutine，确保当前节点此时向对端节点发起连接的节点数不小于5个。每隔10s动态调整一次。

10.4 节点发现机制

节点发现是任何节点接入P2P网络的第一步。这就好比你刚到一个陌生的地方，在这里你没有任何朋友，没有任何可以给你指路和导航的东西。如果你想对这个地方有所了解，你应该怎么办呢？估计只能搜索周边的路人，向他们打听。“搜索周边的路人”这个动作就可以理解为节点发现。

节点发现可以分为两种：初始节点发现和已知节点发现。初始节点发现是指，节点是一个全新的、从未运行的节点，该节点没有网络中其他节点的任何数据，此时节点发现只能依靠节点中硬编码的种子节点获得P2P网络的信息。已知节点发现是指，节点之前运行过，节点数据库中保存着网络中其他节点的信息，此时节点发现可以依靠节点数据库中的节点获得P2P网络的信息，从而构建自己的网络拓扑。

10.4.1 种子节点

在P2P网络中，初始节点在启动时，会通过一些长期稳定运行的节点快速发现网络中其他节点，这样的节点被称为“种子节点”（代码中硬编码种子节点信息）。一般情况下种子节点可分为两种：

第一种是DNS-Seed，又称为“DNS种子节点”。DNS是互联网提供的一种域名查询服务，它将域名和IP地址相互映射保存在一个分布式的数据库中。当我们访问DNS服务器时，给它提供一个域名；DNS服务器会将该域名对应的IP地址返回。

第二种是指，将这些IP地址硬编码到代码中，硬编码的这些节点的地址被称为种子节点。

在比原链中目前使用的是第二种种子节点。在默认的config/toml.go配置文件中将不同网络的种子节点编码到配置中。当节点执行初始节点发现的时候，通过读取seeds中的种子节点，与其交换信息，获取P2P网络信息，代码如下：

```
config/toml.go
var mainNetConfigTmpl = `chain_id = "mainnet"
[p2p]
laddr = "tcp://0.0.0.0:46657"
seeds =
"45.79.213.28:46657,198.74.61.131:46657,212.111.41.245:46657,47
.100.214.154:46657,47.100.109.199:46657,47.100.105.165:46657"
`
```

10.4.2 Kademlia算法

2002年美国纽约大学的PetarP.Maymounkov和David Mazieres在论文《Kademlia：A peer to peer information systembased on the XOR metric》中首次提出了Kademlia算法。Kademlia是一种分布式哈希表（DHT）技术，与其他DHT技术相比，Kademlia使用异或算法计算节点之间的距离，进而建立了全新的DHT拓扑结构。这种算法可以极大地提高路由的查询速度。

Kademlia算法的应用是比较成熟的。早在2005年在BiTtorrent 4.1.0版本中就基于Kademlia算法实现了其结构化的P2P网络拓扑结构。随后国内的BitComt和BitSpirit也开始使用Kademlia算法实现了结构化的P2P网络拓扑结构。

1.关键术语

（1）哈希表

哈希表是用来存储键值对的一种容器。键值对又称为key/value对。

哈希表数据结构中包含N个bucket（桶）。对于某个具体的哈希表，N（桶的数量）通常是固定不变的。于是可以对每个桶进行编号，0～N－1。桶是用来存储键值对的，可以简单地将其理解成一个动态数组，里面存放多个键值对。

图10-1展示了哈希表的查找原理。我们可以方便快速地通过key来获得value。当使用某个key进行查找时，先用某个哈希函数计算这个key的哈希值，得到的哈希值通常是一个整数。然后用哈希值对N（桶数）进行取模运算（除法求余数），就可以算出对应的桶编号。

（2）哈希表碰撞

两个不同的key进行哈希计算，得到相同的哈希值，就是所谓的哈希函数碰撞。一旦出现这种情况，这两个key对应的两个键值对就会被存储在同一个桶（bucket）中。另一种情况散列碰撞是，虽然计算出来的哈希值不同，但经过取模运算之后得到相同的桶编号，这时候也会将两个键值对存储在一个桶中。

哈希碰撞原理如图10-2所示。

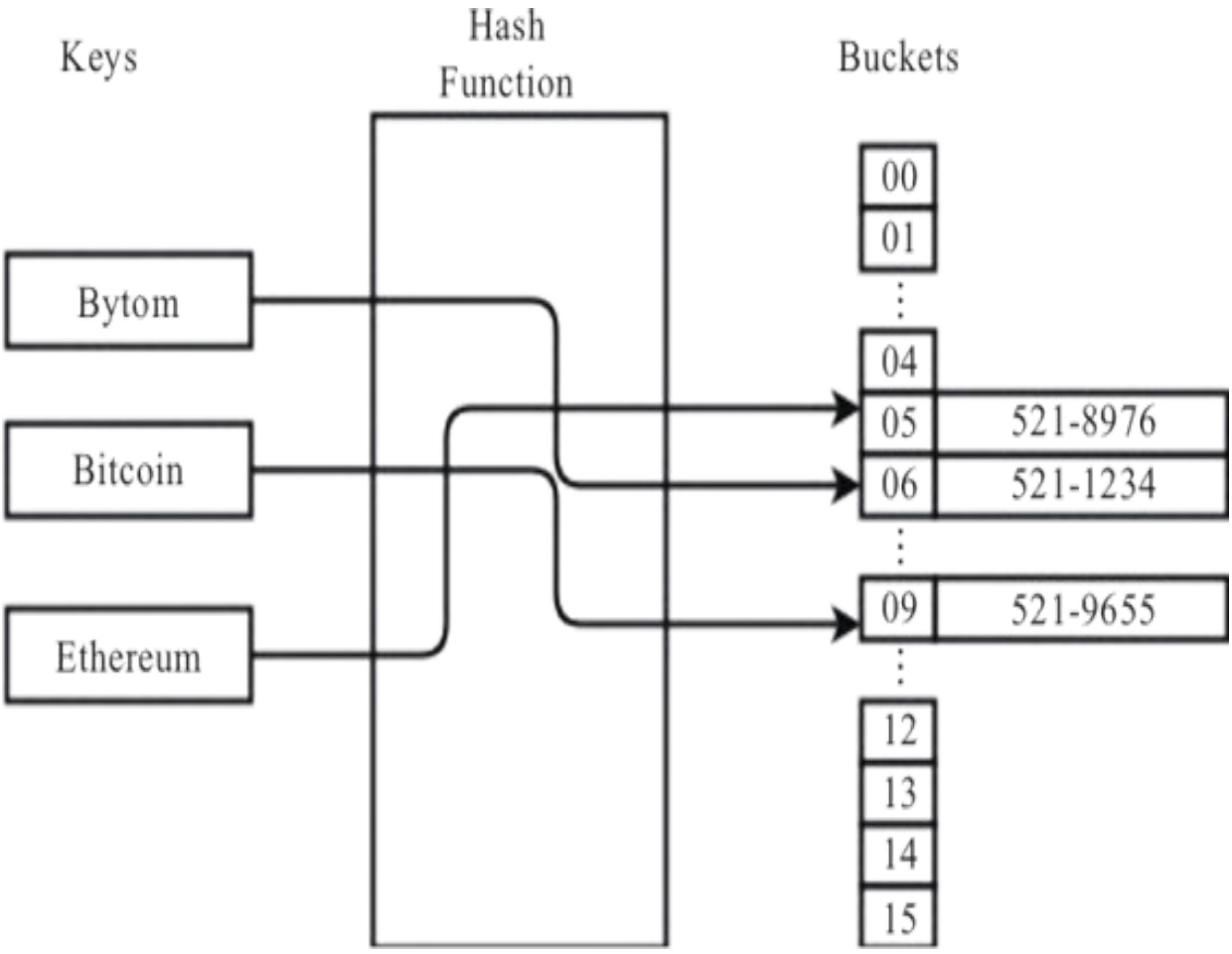


图10-1　哈希表原理

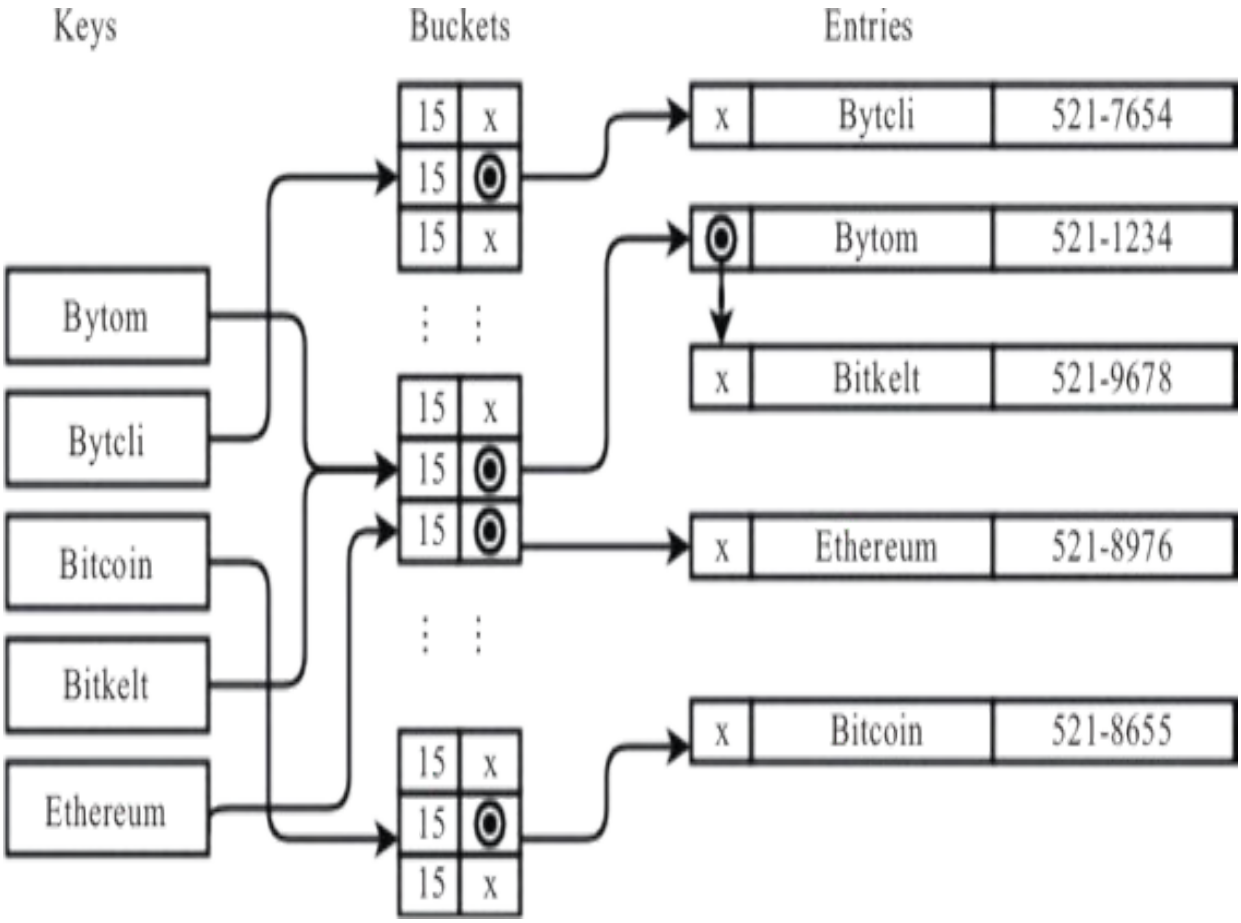


图10-2　哈希表碰撞原理

如果某个哈希表在存储数据时完全没有碰撞，那么每个桶里都只有0个或1个键值对，这样查找起来就非常快。反之，如果某个哈希表在存储数据时出现严重碰撞，就会导致某些桶里存储了很多键值对，那么在查找key的时候，如果定位到的是这种大桶，就需要在这个桶里面逐一比对key是否相同，查找效率会变得很低。

（3）分布式哈希表（DHT）

分布式哈希表在概念上类似于传统的哈希表，差异在于传统的哈希表主要用于单机上的某个软件中；分布式哈希表主要用于分布式系统（此时，分布式系统的节点可以通俗地理解为哈希表中的bucket），分布式哈希表主要用来存储大量的（甚至是海量）数据。

分布式哈希表原理如图10-3所示。

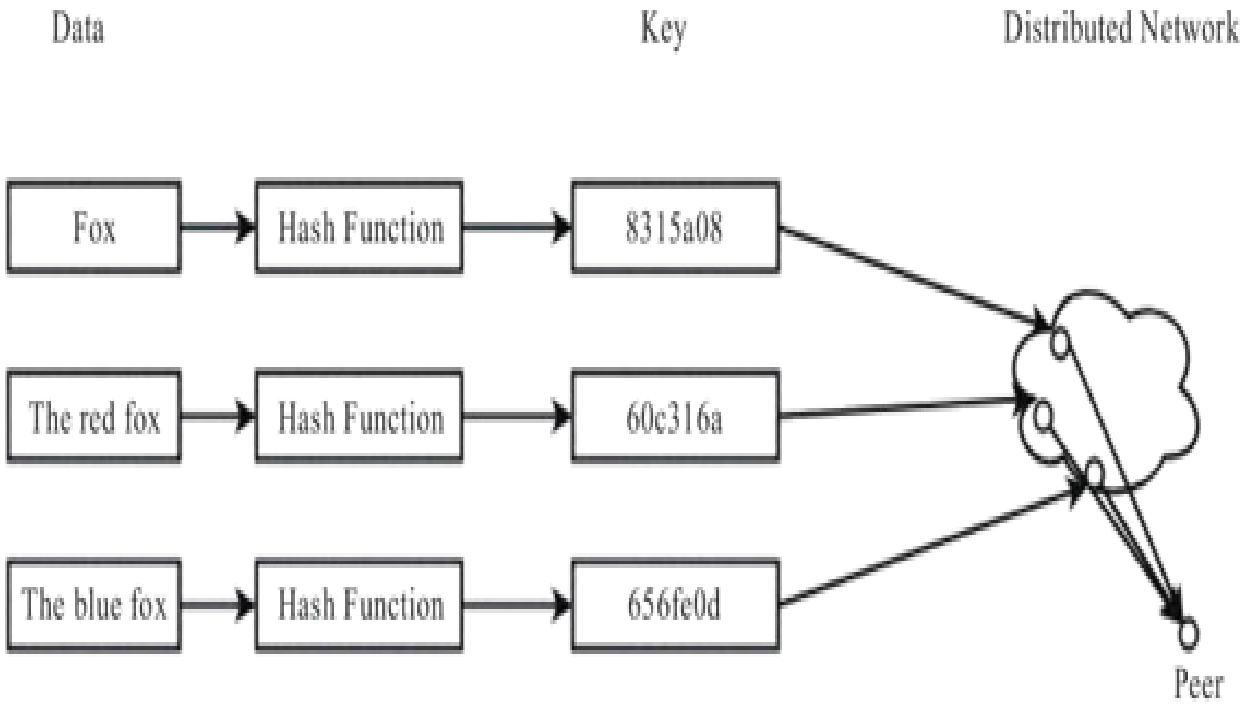


图10-3　分布式哈希表原理

2. Kademlia分布式哈希表

（1）标识

在Kademlia算法网络中，每一个节点都用一个Node ID来唯一标识。Kademlia使用160位的哈希算法（比如SHA1），完整的key用二进制表示有160个数位。实际运行的Kademlia网络，即使有几百万个节点，相比keyspace（key的最大空间是2160）也只是很小很小很小的一个子集。而且，由于哈希函数的特点，key的分布是高度随机的，因此也是高度离散的，任何两个key都不会非常临近。

在比原链中使用[32]byte来存储节点的Node ID。Node ID是由64位的节点私钥经过sha512和edwards25519运算后产生的一个32位的节点公钥。

（2）节点距离

在Kademlia网络中，任意两个节点间的距离是通过对两个节点的Node ID进行异或（XOR）运算得出来的。这里为了对节点距离做简单说明，我们假设NodeID由6位构成，节点X的NodeID为011101，节点Y的

NodeID为100100。那么节点X与节点Y的距离是d（X，Y）=X⊕Y=（011101）⊕（100100）=111001。异或得出这两个节点距离为57，显然高位上数值的差异对结果的影响更大。对于异或操作，拥有类似于几何距离的某些特性（⊕表示XOR）：

· A⊕A=0，反身性，自身距离为零。

· A⊕B=B⊕A，XOR符合交换律，具备对称性。

· A⊕B⊕C=A⊕（B⊕C）=（A⊕B）⊕C，结合律。

·（A⊕B）>0，不同的两个key之间的距离必大于零。

·（A⊕B）+（B⊕C）>=（A⊕C），三角不等式。

异或操作具有单向性，对于任意给定的节点和距离，总能找到另一个存在的节点，使两个节点的距离正好等于给定的距离。

（3）路由表

在Kademlia网络中，每个节点都维护了自己已知的路由信息，这些信息构成了节点的路由表，如图10-4所示。

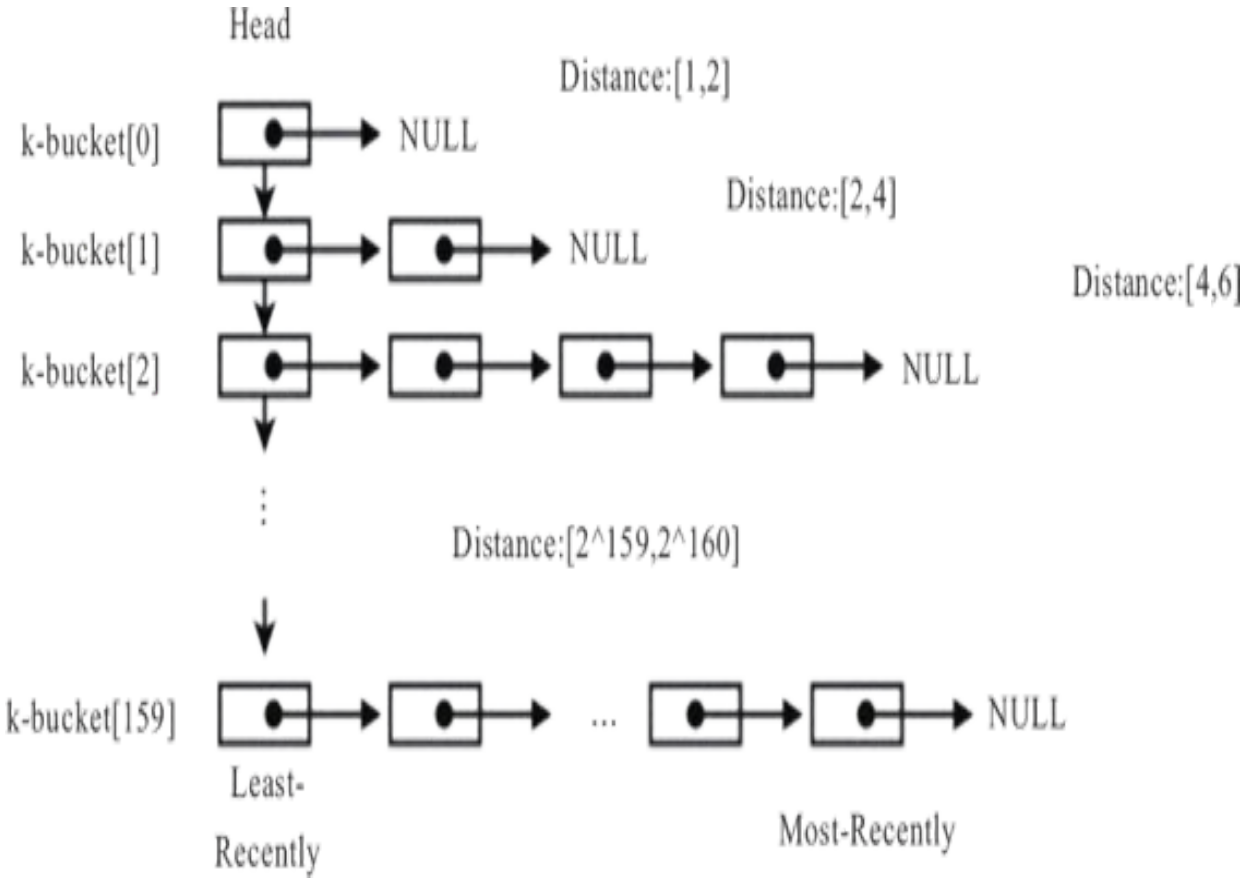


图10-4　路由表示例

路由表一般由160个链表构成，每个链表上记录了距离自己（2^i - 2^{i+1}）的节点信息（整个网络最多可以容纳2^{160}个节点）。这些链表称为k-bucket，简称“k桶”。每条链上最多只能保存K个节点的信息，这是为了在节点查询的时候能够不断地向目标节点靠近，从达到快速收敛的效果。链上节点信息是按照节点发现的时间顺序排列的，最先发现的节点保存到链表的头部，最后发现的节点保存在链表的尾部。之所以将节点信息的路由表保存为k-bucket，就是因为每个链表上最多只能保存K个节点的信息。K可以看做一个系统参数，能够对节点的运行性能进行调节，K值能够保证系统的稳定性。

同样，我们可以将路由表中保存的Kademlia网络节点，根据节点ID按照如下规则构造成一颗二叉树（如图10-5所示）。

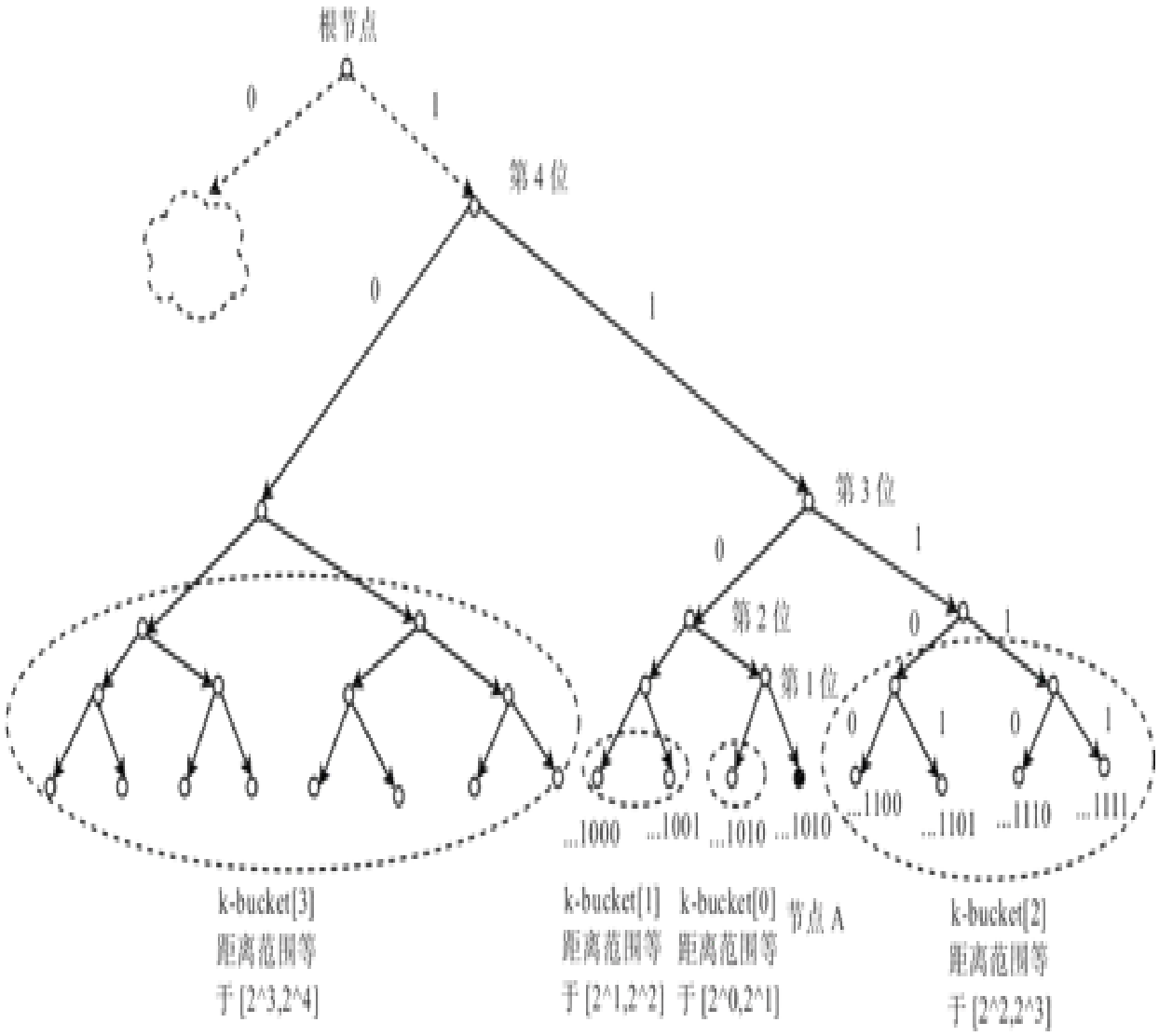


图10-5　路由表的二叉树结构

· 二叉树共有160层，二叉树的每一个叶子节点代表一个Kademlia网络上的节点。

· 从根节点到叶子节点的路径等于节点ID。

· 节点ID共160位，每位只能有两种情况：0或1。从高位开始依次遍历节点ID。若某一位的节点ID为0，则节点向左偏转，即当前位的节点作为前一位二叉树节点的左子节点。若某一位的节点ID为1，则节点向右偏转，即当前位的节点作为前一位二叉树节点的右子节点。

（4）通信信议

通信信议如图10-6所示。

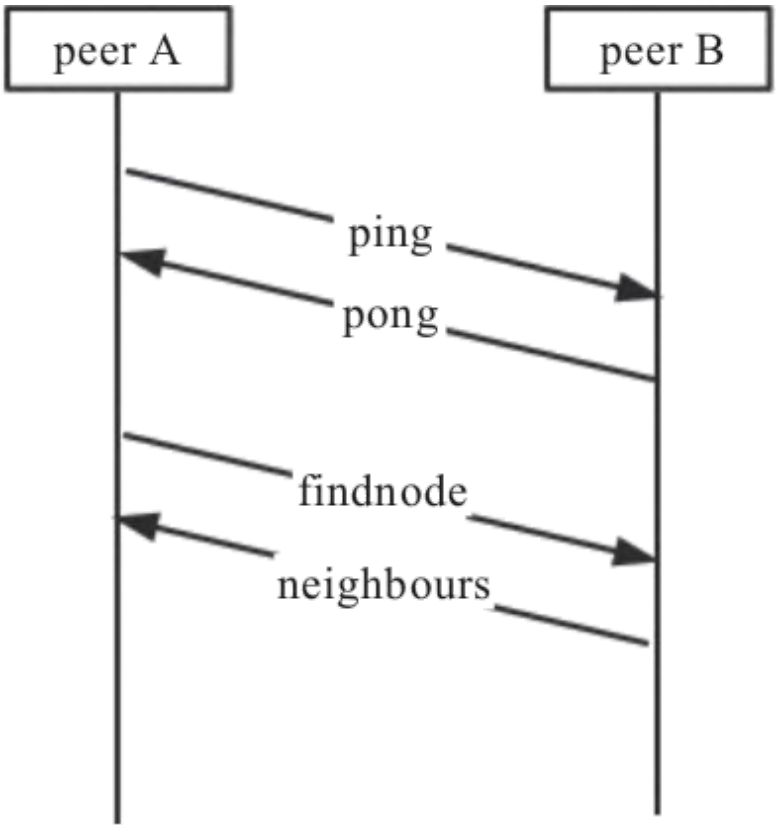


图10-6　通信信议

图10-6中字段说明如下。

· Ping：用于节点探测，判断节点是否存活。

· Pong：Ping响应。

· Findnode：向节点查询与目标节点距离接近的节点信息。

· Neighbours：Findnode响应。

3. 节点路由算法

（1）节点加入

当节点x要加入Kademlia网络，它必须与一个已经在Kademlia网络中的节点y取得联系。x首先将y加入到自己的路由表中，然后x向y发送FIND_NODE请求，y收到后返回x临近的节点。x根据接收到的消息更新自己的路由表。x通过对自己邻近节点由近至远的查询，完成自身路由表信息的建设，同时也把自己的信息发布到其他节点的路由表中。

我们以节点x为例，简述节点路由表的生成算法：

1）节点x的路由表是一个覆盖整个地址空间的单个k-bucket。

2）当节点x从其他节点获得了新的节点消息后，x会根据新节点ID的前缀，将其加入到k-bucket中。

a）若k-bucket没有满，则直接将新节点插入到k-bucket。

b）若k桶已经满了，则判断k-bucket的地址空间是否覆盖了节点的ID。如果已经覆盖了节点ID，则把k-bucket分裂为两个大小相同的新k-bucket，并对原k-bucket中的节点信息按照新的k-bucket前缀值进行重新分配。如果k-bucket的地址空间不包含新节点的ID，则直接抛弃该新节点。

3）不断重复上述过程，构造节点x的路由信息表。由此，构造的路由表中包含距离近的节点信息多，距离远的节点信息少，保证了路由查询过程能够快速收敛。

4）节点在离开Kademlia网络的时候并不需要发布任何信息。因为Kademlia协议要求每个节点必须周期性地发布自己保存的键值数据，并把这些数据缓存在离自己最近的K个节点上，因此失效节点的数据也会很快地在其他节点上得到更新。

（2）路由表刷新

路由表在建立后还需要定期刷新，以保证节点信息的实时性，刷新机制大致有如下几种：

· 节点主动发起“查询节点”的请求（协议类型为FIND_NODE），从而刷新路由表中的节点信息。

· 节点收到其他节点发来的请求（协议类型为FIND_NODE或FIND_VALUE），会把对方的ID加入自己路由表中的某个k-bucket。

· 探测失效节点时（协议类型为PING探测机制），可以判断某个ID的节点是否在线。因此可以定期探测路由表中的每一个节点，把下线的节点从路由表中删除。

（3）节点查找

Kademlia算法的最显著优势是提供了快速的节点查找机制，并且还可以调节参数K（对等节点返回的距离目标节点最近节点的个数）控制朝目标节点收敛的速度。

假设节点x要查找NodeID为t的节点，Kademlia算法将会按照如下步骤递归查找：

1）计算节点x到节点t的距离d。

2）从节点x的第[log d]个k-bucket中取出k个节点的信息，向这k个节点发送FIND_NODE消息。如果当前k-bucket中节点个数不足k个，则从附近桶（多个）中选择距离最接近d的节点，总共k个。

3）接收到FIND_NODE消息的每个节点，如果发现自己是节点t，则回答自己是最接近t的节点，否则测量自己和节点t的距离，并从自己对应的K桶中选择k个节点，将k个节点的信息发送给x。

4）x对新接收到的每个节点再次执行FIND_NODE操作，直到每个分支都有节点响应自己是最接近t的节点。

5）通过上述查找操作，x得到最近t的k个节点信息。

整个路由查找过程其实就是一个递归操作，递归方程式为：

n_0=x（查询操作的发起者）

$N_1=\int \mathrm{indnode}_{n0}(t)$

$N_2=\int \mathrm{indnode}_{n1}(t)$

……

$N_l = \int indnode_{nl-1}(t)$

该递归过程一直持续到N_1=t，或者N_1的路由表中没有任何关于t的信息，即查询失败。由于每次查询都是从最接近t的k-bucket中获取信息，这样的机制保证了每一次递归操作都能够获得距离减半的效果，从而保证了整个查询过程的收敛速度的算法复杂度为o（log N）。

仔细分析整个查询过程会发现，它和我们实际生活中去找某个人打听消息的过程非常相似。例如，你想询问“某个地方”如何走，首先你肯定去向你身边最有可能知道的几个人请教，但是如果他们不知道，他们一般会跟你说：“你可以去问问某某。这时候他们就是把身边最有可能了解“某个地方”的人推荐给你。如此下去，你就能找到你想去的地方。

4. Kademlia算法的优势

Kademlia算法具有以下优势：

· 简单性。与其他DHT协议相比，Kademlia是实现和原理都特别简单的协议。例如，与CAN相比，CAN的拓扑结构是基于多维笛卡尔环面的，而Kademlia是基于二叉树的拓扑结构的。Kademlia除了拓扑结构很简单，它的距离算法也很简单，即使用节点ID的异或运算（XOR）。

· 灵活性。Kademlia的“K-bucket”可以根据使用场景来动态调整K值，而且对K值的调整完全不影响代码实现。

· 性能。Kademlia的路由算法天生就支持并发，而很多DHT协议（包括Chord）没有这种优势。公网上哪怕是同样两个节点之间的传输速率都可能时快时慢。由于Kademlia路由请求支持并发，发出请求的节点总是可以获得最快的那个peer的响应。

· 安全性。在Kademlia网络中，攻击者想在Kademlia网络中添加一个恶意节点来攻击Kademlia网络是十分困难的。

10.4.3 UPnP协议

通用即插即用（Universal Plug and Play，UPnP），网络协议的目标是，使多个网络中的各种设备能够相互无缝连接，以简化相关网络的实现。

我们通过下面的例子来介绍UPnP协议的应用场景。NAT技术设置了一个出口路由，只有这个出口路由拥有公网IP，其他设备则在这个路由的后面组成一个局域网。当节点需要和外部通信时，就在路由处将内部地址替换成公用地址。这样就可以使多台网络设备共享Internet连接。在P2P协议中会存在问题。比如，有一台局域网内的设备想要连接另一个局域网的一个对等节点peer，因为它们都藏在出口路由的后面，它们是没有办法发现彼此的。它们如果想发现彼此就需要使用UPnP协议，在公网找寻找一个支持UPnP协议的设备，分别将它们自己和找到的设备做一个EndPoint映射。

（1）UPnP协议栈

完整的UPnP由设备寻址、设备发现、设备描述、设备控制、事件通知和基于Html的描述等几部分构成。UPnP设备协议栈如图10-7所示。

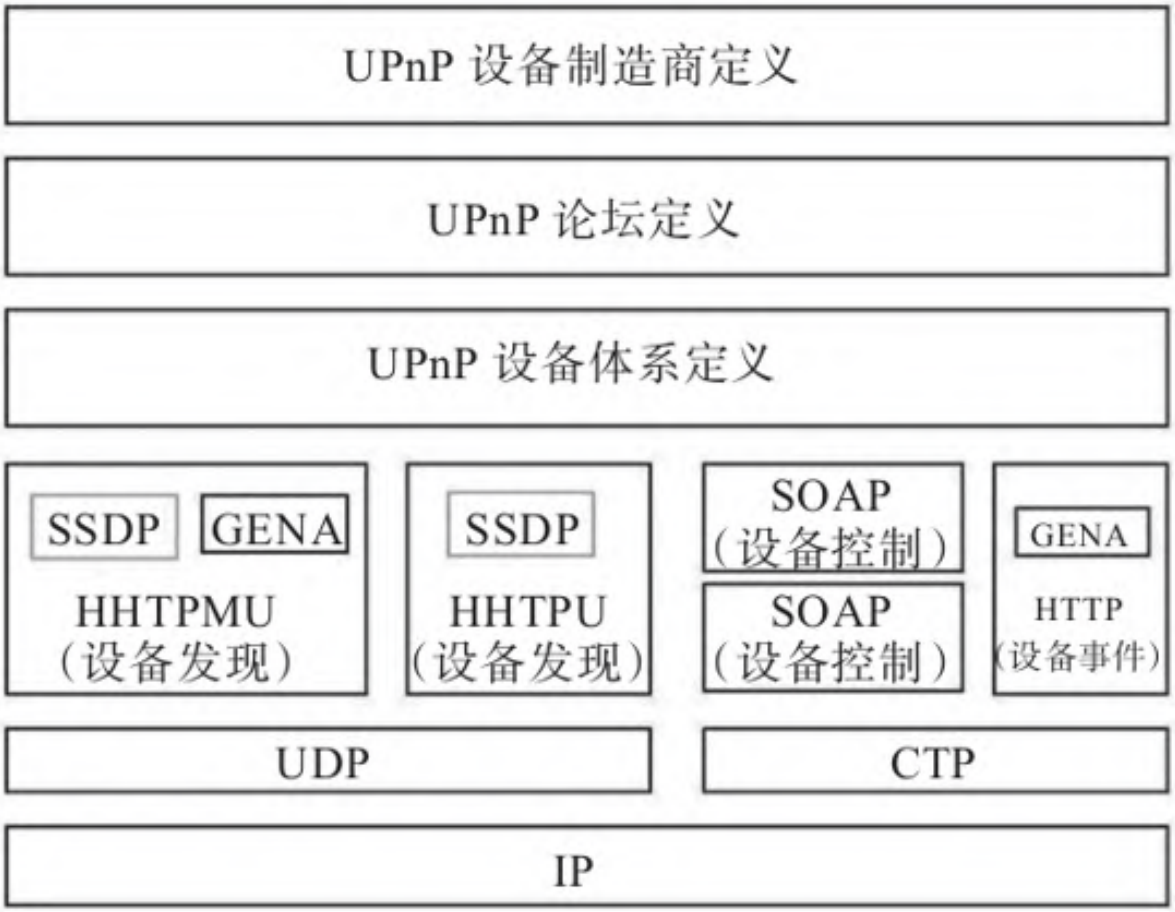


图10-7　UPnP协议栈

图10-7中，SSDP（Simple Service Discovery Protocol）为简单服务发现协议；GENA（Generic Event Notification Architecture）为通用事件通知结构；SOAP（Simple Object Access Protocol）为简单对象访问协议；XML（Extensible Markup Language）为可扩张标记语言。

（2）UPnP工作流

UPnP的工作过程分为5部分：寻址（Addressing），发现（Discovery），描述（Description），控制（Control），事件（Eventing）。

（3）UPnP穿透过程

1）发送多播请求。一个节点加入到网络后，在网络中会多播，大量的“发现消息”来通知其嵌入式设备和服务，所有的控制点都可以监听多播地址以接收通知，标准的多播地址是239.255.255.250：1900。我们可以通过发送HTTP请求查询局域网中UPnP设备，消息形式如下：

```
M-SEARCH * HTTP/1.1\r\n
HOST: 239.255.255.250:1900\r\n
ST: ssdp:all\r\n
MAN: \"ssdp:discover\"\r\n
MX: 2\r\n\r\n
```

2）获取设备描述URL地址。如果网络中存在UPnP设备，此设备会发送响应消息，其形式如下：

```
HTTP/1.1 200 OK
CACHE_CONTROL: max-age=100
DATE: XXXX
LOCATION:http://192.168.1.1:1900/igd.xml
SERVER: TP-LINK Wireness Router UPnP1.0
ST: upnp:rootdevice
```

响应消息中，200 OK确定成功地找到了设备。然后要从响应内容中找到根设备的描述URL（例如上面响应报文中的http://192.168.1.1:1900/igd.xml），通过此URL就可以找到根设备的描述信息，从根设备的描述信息中又可以得到设备的控制URL，通过控制URL就可以控制UPnP的行为。上面的响应消息表示，在局域网中成功地找到了一台支持UPnP的无线路由器设备。

3）获取设备描述文档XML。通过2）找到的设备描述URL的地址得到XML文档。发送HTTP请求消息：

```
GET /igd.xml HTTP/1.1
HOST:192.168.1.1:1900
Connection: Close
```

最终能得到一个设备描述文档，从中可以找到服务和UPnP控制URL。每一种设备都有对应的serviceURL和controlURL。

4）端口映射。拿到设备的控制URL后就可以发送控制信息了，每一种控制都是根据HTTP请求发送的，请求形式如下：

```
POST path HTTP/1.1
HOST: host:port
SOAPACTION:serviceType#actionName
CONTENT-TYPE: text/xml
CONTENT-LENGTH: XXX
....
```

其中，path为控制URL；host：port为目的主机地址；actionName为控制UPnP设备执行响应的指令。

本节简单地介绍了UPnP协议，读者如果想深入了解关于UPnP协议详细内容可自行参考（https://zh.wikipedia.org/wiki/UPnP）。在10.5节中我们使用GO语言通过UPnP协议进行端口映射，能够发现区块链网络中的其他节点。

10.4.4 RLPX网络协议

比原链的P2P网络部分借鉴了一些RLPX网络协议的思想。RLPX是以太坊的底层网络协议套件，包括P2P加密通信、节点发现等功能。当节点启动时，RLPX网络协议会同时监听TCP和UDP数据包，UDP用来处理节点发现的数据包；TCP用来处理P2P通信，主要是区块和交易同步数据。

RLPX协议规定节点间在必要时能互相识别并通信，节点需要提供node_id，ip，tcp端口号。node_id同时也是节点的公钥地址。下面是一个简单的enode地址示例：

```
enode://a6ca8f9641ea3b963a8d8b972338cd173d07312e08f04d0e3f695b2
e334d819d@10
.3.58.6:46657
```

参数说明如下：

- node_id：a6ca8f964...e334d819d
- ip：10.3.58.6
- TCP端口：46657
- UDP端口：46657

在上面的例子中节点会在同一个端口上同时监听TCP和UDP数据包，RLPX协议也支持将TCP和UDP端口分开。若要将UDP和TCP端口分开，需要通过discport参数指定UDP端口。下面是将TCP和UDP数据包分开的一个enode地址：

```
enode://a6ca8f9641ea3b963a8d8b972338cd173d07312e08f04d0e3f695b2
e334d819d@10.3.58.6:46657?discport=30301
```

参数说明如下：

· node_id：a6ca8f964...e334d819d

· IP：10.3.58.6

· TCP端口号：46657

· UDP端口号：46657

虽然RLPX协议支持使用两个端口，但在实现时，所有的链一般都同时监听同一个端口的TCP和UDP数据包。关于RLPX协议的使用请参考10.5节和10.7.4节。

10.5 节点发现代码实现

10.5.1 节点发现初始化

节点在进行P2P通信模块SyncManager初始化的时候，会同时初始化节点发现模块。初始化代码如下所示：

```
bytom/netsync/handle.go
func initDiscover(config *cfg.Config, priv
*crypto.PrivKeyEd25519, port uint16) (*discover.Network, error)
{
    addr, err := net.ResolveUDPAddr("udp",
net.JoinHostPort("0.0.0.0", strconv.FormatUint(uint64(port),
10)))
    // ...

    conn, err := net.ListenUDP("udp", addr)
    // ...

    realaddr := conn.LocalAddr().(*net.UDPAddr)
    ntab, err := discover.ListenUDP(priv, conn, realaddr,
path.Join(config.DBDir(), "discover.db"), nil)
    // ...

    nodes := []*discover.Node{}
    for _, seed := range strings.Split(config.P2P.Seeds, ",") {
        version.Status.AddSeed(seed)
        url := "enode://" +
hex.EncodeToString(crypto.Sha256([]byte(seed))) + "@" + seed
        nodes = append(nodes, discover.MustParseNode(url))
    }
    if err = ntab.SetFallbackNodes(nodes); err != nil {
        return nil, err
    }
```

```
    return ntab, nil
}
```

节点发现模块初始化流程如下：

1）通过golang net包提供的ListenUDP方法监听46656端口的UPD数据包。

2）通过discover.ListenUDP方法初始化UDP Network，ListenUDP方法和newNetwork方法分别会启动两个goroutine处理remote节点的UDP数据包。这两个goroutine分别是transport.readLoop()和net.loop()。

3）生成种子节点的RLPX协议地址。

4）通过SetFallbackNodes方法，将种子节点添加到节点刷新列表中，节点开始进行路由表新操作。

10.5.2　路由表实现

Kademlia的路由表是由一组称为k-bucket的数据结构组成，k-bucket中的每个节点记录了Kademlia网络节点的NodeID、State、IP等信息。这组k-bucket按照与target节点的距离进行排序，共257个k-bucket，每个k-bucket包含16个节点，每个k-bucket内的节点则按照节点last activity的时间排序，结构如图10-8所示。

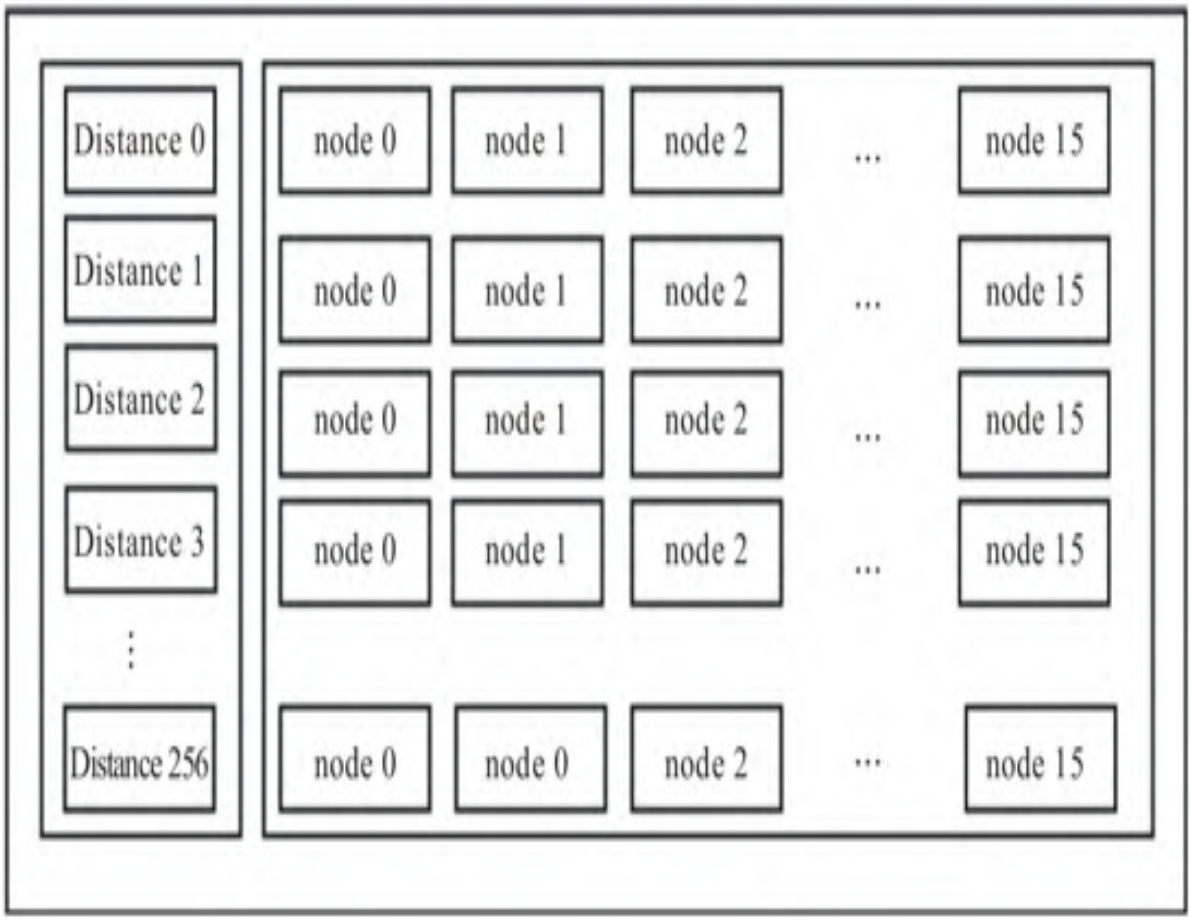


图10-8　路由表结构

1. 路由表的增、删、定位

路由表数据结构如下：

```
p2p/discover/table.go
const (
    alpha      = 3  // Kademlia concurrency factor
    bucketSize = 16 // Kademlia bucket size
    hashBits   = len(common.Hash{}) * 8
    nBuckets   = hashBits + 1 // Number of buckets

    maxFindnodeFailures = 5
)


type Table struct {
    count         int                // number of nodes
    buckets       [nBuckets]*bucket // index of known nodes by
distance
    nodeAddedHook func(*Node)        // for testing
    self          *Node              // metadata of the local
node
}

type bucket struct {
    entries      []*Node
    replacements []*Node
}
```

常量字段说明如下。

· alpha：Kademlia算法中的FindNode操作并发数。

· bucketSize：Kademlia算法中的每个k-bucket的长度。

· hashBits：hash值的二进制长度。

· nBuckets：Kademlia路由表中k-bucket的个数。

· maxFindnodeFailures：Kademlia算法执行FindNode操作失败上限。

k-bucket字段说明如下。

· entries：节点存储桶，最大存放16个节点。按照recently active进行排序，最活跃的节点为第一个元素。

· replacements，临时存储桶。如果entries存储桶有可用空间则成功添加节点；否则，将该节点添加到临时存储桶中。同样，临时存储桶最多存放16个节点。

table路由表字段说明如下。

· count：路由表中的节点数量。

· buckets：由一组k-bucket组成，按节点距离存储节点信息。

· nodeAddedHook：回调函数，测试时使用，目前不需要关注。

· self：当前节点。

table路由表具体实现函数为：p2p/discover/table.go。

其中字段说明如下。

· func newTable()：初始化table路由表。

· func add()：尝试添加节点到table路由表。

· func chooseBucketRefershTarget()：刷新路由表，保持路由表中的节点都是可用的。

· func closest()：从路由表中查找最靠近target的N个节点。

· func delete()：从路由表中删除节点。

· func deleteFromReplacement()：从路由表的临时表中删除节点。

· func deleteReplace()：从路由表中删除节点，并从临时表中补一个节点到正式表中。

· func readRandomNodes()：从路由表中批量随机读取节点。

· func stuff()：批量添加多个节点到路由表。

向路由表中添加节点的代码如下：

```
func (tab *Table) add(n *Node) (contested *Node) {
    if n.ID == tab.self.ID {
        return
    }
    b := tab.buckets[logdist(tab.self.sha, n.sha)]
    switch {
    case b.bump(n):
        return nil
    case len(b.entries) < bucketSize:
        b.addFront(n)
        tab.count++
        if tab.nodeAddedHook != nil {
            tab.nodeAddedHook(n)
        }
        return nil
    default:
        tab.deleteFromReplacement(b, n)
        b.replacements = append(b.replacements, n)
        if len(b.replacements) > bucketSize {
            copy(b.replacements, b.replacements[1:])
            b.replacements =
b.replacements[:len(b.replacements)-1]
        }
        return b.entries[len(b.entries)-1]
    }
}
```

向路由表中添加节点共有以下三个步骤：

1）判断添加的节点是否是本节点，如果是本节点则不添加。

2）根据logdist算法计算节点应该位于table路由表中第几个k-bucket中。

3）如果k-bucket的正式表没有满，则将新节点加入正式表的表头。否则将新节点加入临时表。将节点加入临时表时，为了保证临时表中节点不重复，先执行deleteFromReplacement方法将节点从临时表中删除。这样既可以保证临时表中节点不重复，还可以保证节点信息的及时性。如果新加入节点导致临时表满了，临时表会将表头节点删除。

从table路由表中删除节点的代码如下：

```
func (tab *Table) delete(node *Node) {
    bucket := tab.buckets[logdist(tab.self.sha, node.sha)]
    for i := range bucket.entries {
        if bucket.entries[i].ID == node.ID {
            bucket.entries = append(bucket.entries[:i],
bucket.entries[i+1:]...)
            tab.count--
            return
        }
    }

    tab.deleteFromReplacement(bucket, node)
}
```

从路由表中删除节点是一个比较简单的操作，首先根据logdist算法计算节点应该位于路由表中第几个k-bucket中。然后遍历k-bucket中的节点，找到目标节点后直接删除，同时将路由表中节点总数减1。

logdist（logarithmic distance）算法用于计算a，b节点间距离的对数值。根据对数值可快速定位节点位于table路由表中第几个K桶中。代码如下：

```
func logdist(a, b common.Hash) int {
    lz := 0
    for i := range a {
        x := a[i] ^ b[i]
        if x == 0 {
            lz += 8
        } else {
            lz += lzcount[x]
            break
        }
    }
    return len(a)*8 - lz
}
var lzcount = [256]int{
  8, 7, 6, 6, 5, 5, 5, 5,
  4, 4, 4, 4, 4, 4, 4, 4,
  3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 3, 3, 3,
```

```
  2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0,
}
```

---

2. 节点距离计算

在Kademlia算法的实际使用过程中，并不需要完整地计算出实际距离。因为在节点查找的过程中，我们选的是距离目标节点最近的n个节点，因此，只要能比较出哪个节点比目标节点更近即可。

代码示例如下：

```
p2p/discover/node.go
func distcmp(target, a, b common.Hash) int {
    for i := range target {
        da := a[i] ^ target[i]
        db := b[i] ^ target[i]
        if da > db {
```

```
            return 1
        } else if da < db {
            return -1
        }
    }
    return 0
}
```

对节点ID从高位到低位做异或运算，哪个异或出来的值小，则哪个距离节点近。例如，节点target的ID为011011，节点a的ID为010010，节点b的ID为001001。从高位开始逐位取节点a和b的值与target值做异或运算。第一位，3个节点都是0，暂时无法比较。继续看第二位的取值：target^a=0，target^b=1，这时target^a<target^b的值，选择a，因为a距离target更近。

3. 节点ID生成

在比原链中，NodeID类型共32个字节，共32×8=256位。NodeID使用节点的公钥作为标识。例如：

```
p2p/discover/node.go

type NodeID [32]byte

p2p/discover/net.go

ourID := NodeID(ourPubkey)
```

10.5.3 Kademlia通信协议

在比原链中，Kademlia网络节点间通信基于UDP，双方都有一个UDP监听特定的端口，由于UDP协议属于不可靠协议，丢包或重复、数据校验等需要由应用层来做。Kad通信协议的命令构成见表10-2。

表10-2 Kademlia通信协议命令

消息类型	说明	结构体
Ping	用于节点探测，判断节点是否存活	[version，from，to，expiration]
Pong	Ping 响应	[to，ping-hash，expiration]
Findnode	向节点查询与目标节点距离接近的节点信息	[target，expiration]
Neighbours	Findnode 响应	[node，expiration]

10.5.4　邻居节点发现实现

1.节点发现流程

比原链中，节点使用Table结构保存邻居节点的路由信息。使用nodesByDistance结构查找最靠近目标节点的n个节点。nodesByDistance是根据目标的距离进行排序的节点列表。使用nodesByDistance的push方法向entries中加入节点时，会比较加入节点和entries数组中保存的节点与目标节点的距离，只保存距离目标节点最近的n个节点。节点发现流程如图10-9所示。

发现流程如下：

1）节点启动时，节点将公钥作为节点的NodeID。

2）节点读取配置文件中种子节点信息，与种子节点完成ping-pong握手后，将种子节点加入路由表中。

3）节点首次构造一个以自己为目标节点的nodesByDistance对象。

4）节点将自己加入到nodesByDistance.entries数组中。

5）节点开始定时刷新路由表（定时器每分钟执行一次）。

6）节点依次向nodesByDistance数据库中的节点发送findnode命令。

7）对等节点收到findnode命令，从k-bucket中找出最靠近目标节点的16个节点并返回。

8）节点收到对等节点的neighborsPacket命令，将临近节点加入k-bucket中。

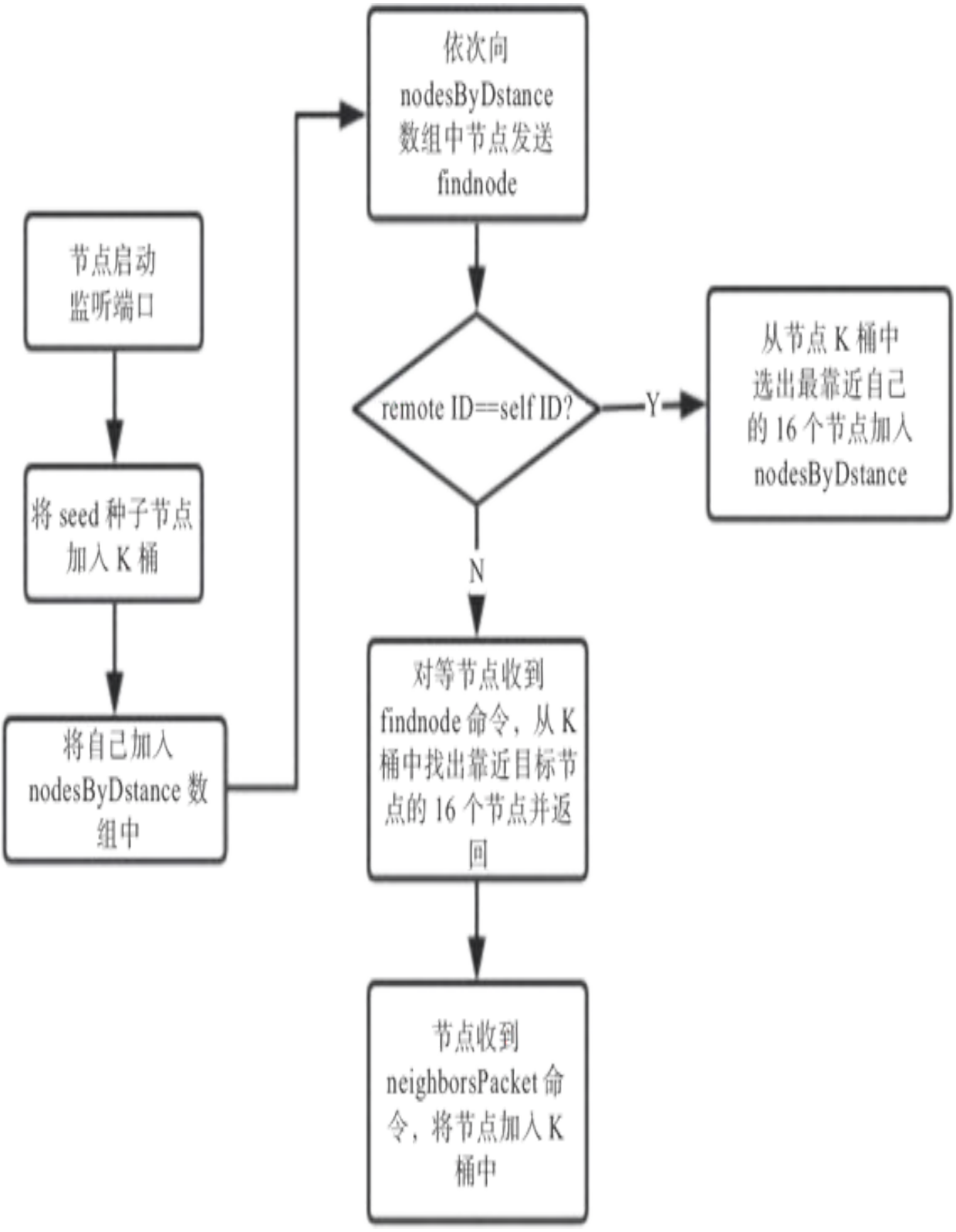


图10-9　节点发现流程

2. UPnP端口映射

在API Server服务中，我们使用HTTP标准库来监听端口以获得其他节点的连接信息。同样，P2P服务需要设置一个监听端口，用来监听来自对等节点的连接，用于块和交易的数据同步。监听本地端口后需要通过UPnP协议发现网络中支持该协议的设备，并将端口与公网地址端口相互映射。

如图10-10所示，节点启动时，会同时监听TCP和UDP的端口（通常是用同一个端口），UDP用来处理节点发现协议，TCP用来接收P2P通信。在映射过程中需要将TCP、UDP端口同时映射，代码实现流程如下：

```
p2p/listener.go
func NewDefaultListener(protocol string, lAddr string, skipUPNP
bool) (Listener, bool) {
    listener, err := net.Listen(protocol, lAddr)
    // ...

    intAddr, err := NewNetAddressString(lAddr)
    if err != nil {
        cmn.PanicCrisis(err)
    }

    // ...
    if !skipUPNP && (lAddrIP == "" || lAddrIP == "0.0.0.0") {
        extAddr, err = getUPNPExternalAddress(lAddrPort,
listenerPort)
        upnpMap = err == nil
        log.WithField("err", err).Info("get UPNP external
address")
    }

    if extAddr == nil {
        if address := GetIP(); address.Success == true {
            extAddr =
NewNetAddressIPPort(net.ParseIP(address.IP), uint16(lAddrPort))
        }
    }
    if extAddr == nil {
        extAddr = getNaiveExternalAddress(listenerPort, false)
    }
    if extAddr == nil {
        cmn.PanicCrisis("could not determine external address!")
```

```
    }

    dl := &DefaultListener{
        listener:    listener,
        intAddr:     intAddr,
        extAddr:     extAddr,
        connections: make(chan net.Conn,
numBufferedConnections),
    }

    // ...
    dl.Start() // Started upon construction
    // ...
}
```

代码中，如果节点需要开起UPnP功能，则会通过UPnP获取节点对外提供的外网IP和端口信息并进行绑定。如果通过UPnP没有获取到外部IP和端口信息，则节点会向一系列查询外网IP的网站发送请求，以获取节点的外网IP。如果仍然获取不到节点的外部IP信息，则认为当前节点属于无网络节点，通过net.InterfaceAddrs()方法获取节点的本地IP信息，通过以上信息构造一个DefaultListener对象，调用其OnStar方法启动端口监听服务。代码如下：

```
func (l *DefaultListener) OnStart() error {
    l.BaseService.OnStart()
    go l.listenRoutine()
    return nil
}
```

在dl.Start()函数中，调用listenRoutine方法进入for循环持续监听其他节点的连接信息，如果发现有其他节点发出的Dial信息，就将其加入到connections中。connections是一个大小为10的缓冲channel，用来存放连接上该端口的链接。这里获取的连接对象将在后面与节点握手交换密钥的时候使用。

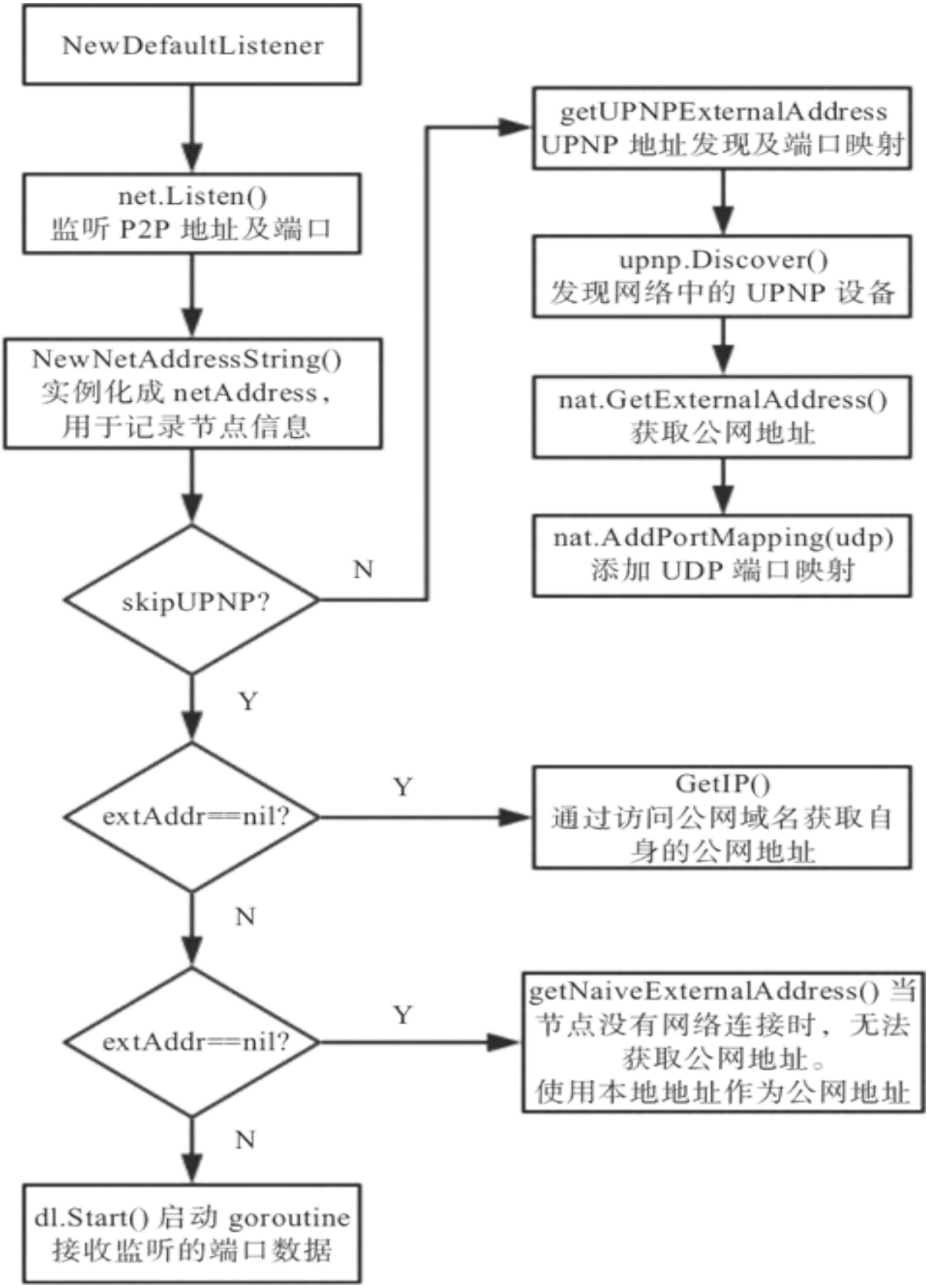


图10-10　UPnP端口映射

3. 加密握手连接

在公链设计中，为了防止中途被窃取或修改，节点与节点之间传输信息要加密。节点第一次握手时需要互换一次性公钥，在这个过程中是明文传输。当公钥交互完成后，数据的传输才能是加密传输。

在p2p.Switch的OnStart方法中，节点为每一个Listener启动了一个监听的goroutine。这个goroutine持续地监听DefaultListener的Connctions通道。当通道中有TCP连接加入的时候，会从通道中取出连接，但并不是每一个TCP连接都会被节点使用，当前节点连接数超过MaxNumPeers（默认50）个连接时，会拒绝其他的连接。在公链设计中如果不限制节点的最大链接个数，会影响节点的网络带宽，消耗节点的网络资源。最后节点与对等节点交换密钥完成握手过程，将节点加入p2p.Switch的PeerSet中。代码示例如下：

```
p2p/switch.go
func (sw *Switch) listenerRoutine(l Listener) {
    for {
        inConn, ok := <-l.Connections()
        if !ok {
            break
        }

        // disconnect if we alrady have MaxNumPeers
        if sw.peers.Size() >= sw.Config.P2P.MaxNumPeers {
            inConn.Close()
            log.Info("Ignoring inbound connection: already have
enough peers.")
            continue
        }

        // New inbound connection!
        if err := sw.addPeerWithConnection(inConn); err != nil {
            log.Info("Ignoring inbound connection: error while
adding peer.", " address:", inConn.RemoteAddr().String(), "
error:", err)
            continue
        }
    }
}
```

与一个节点连接并加入到连接池中，在addPeerWithConnection函数中主要有两步：第一步，通过newInboundPeerConn交换密钥完成握手过程。第二步，通过sw.AddPeer将对等节点加入到连接池中。代码示例如下：

```
func (sw *Switch) addPeerWithConnection(conn net.Conn) error {
    peerConn, err := newInboundPeerConn(conn, sw.nodePrivKey,
sw.Config.P2P)
    if err != nil {
        conn.Close()
        return err
    }

    if err = sw.AddPeer(peerConn); err != nil {
        conn.Close()
        return err
    }
    return nil
}
```

（1）密钥交换过程

密钥交换过程如图10-11所示，具体过程如下：

图10-11　密钥交互过程

1）根据节点的私钥生成节点的公钥。这里生成的密钥对适用于Ed25519加密算法。这个算法有一个最奇特的特点——公钥和私钥的长度是不一样的。私钥的长度是64字节，公钥长度只有32字节。这是因为公钥通常用来做加密运算，较短的公钥会使加密运算的速度快一些。

2）节点根据curve25519算法生成一对临时的公私钥locEpnPub和locEphPriv，用于密钥交换过程的信息加密。这里生成的公私钥的长度都是32字节。

3）通过shareEphPubKey方法把2）中生成的临时公钥locEpnPub发送给对方，同时获取对方生成的临时公钥。

4）双方拿到对方的临时公钥后，都会与自己生成的临时私钥做一个computeSharedSecret运算，生成shrSecret密钥用于后面消息传递的加密和解密操作。这里双方利用computeSharedSecret算法使用对方的公钥和自己的私钥计算得到的shrSecret是相同的，这样双方才可以拿各自算出来的shrSecret去解密对方的加密数据。

5）拿对方和自己的临时公钥进行排序，使两边获得相同的loEphPub和hiEphPub。

6）根据loEphPub和hiEphPub计算nance值，计算出来的recvNonce与sendNonce，一个是用于接收数据后解密，一个是用于发送数据时加密。连接双方的这两个数据正好是相反的，也就是说，一方的recvNonce与另一方的sendNonce相等，这样当一方使用sendNonce加密后，另一方才可以使用相同数值的recvNonce进行解密。虽然nonces和shrSecret都是供公钥和签名数据加密和解密使用的，但是shrSecret是固定不变的，而nonces在不同的信息之间是不同的。当一方发送完数据后，其持有的sendNonce会增2，另一方接收并解密后，其recvNonce也会增2，双方始终保持一致。

7）把两个临时公钥放组合到一起，通过SHA256算法得到一个32字节的challenge。

8）根据之前获得的nonces和shrSecret，实例化一个SecretConnection对象，用于建立一个安全的连接通道。

9）使用自己的私钥对challenge数据进行签名，得到一个32字节的签名信息。然后跟自己的公钥一起发送给对方，同时从对方那里读取它的公钥和签名数据，解密验证签名。如果验证通过，则把对方的公钥也加到SecretConnection对象中，供以后使用。

（2）添加对等节点到PeerSet

在节点完成密钥交换后，节点通过AddPeer方法将对等节点加入到p2p.Switch的PeerSet中保存。代码示例如下：

```
p2p/switch.go
func (sw *Switch) AddPeer(pc *peerConn) error {
    peerNodeInfo, err := pc.HandshakeTimeout(sw.nodeInfo,
time.Duration(sw.peerConfig.HandshakeTimeout))
    // ...

    if err := version.Status.CheckUpdate(sw.nodeInfo.Version,
peerNodeInfo.Version, peerNodeInfo.RemoteAddr); err != nil {
        return err
    }
    if err := sw.nodeInfo.CompatibleWith(peerNodeInfo); err !=
nil {
        return err
    }

    peer := newPeer(pc, peerNodeInfo, sw.reactorsByCh,
sw.chDescs, sw.StopPeerForError)
    if err := sw.filterConnByPeer(peer); err != nil {
        return err
    }

    if pc.outbound &&
!peer.ServiceFlag().IsEnable(consensus.SFFullNode) {
        return ErrConnectSpvPeer
    }

    // Start peer
    if sw.IsRunning() {
        if err := sw.startInitPeer(peer); err != nil {
            return err
        }
    }
    return sw.peers.Add(peer)
}
```

AddPeer方法的具体流程如下：

1）pc.HandshakeTimeout方法向对等节点发送节点的nodeInfo信息，包括节点的公钥、名称、版本号、监听的地址等。对等节点同时也需要把它的nodeInfo信息返回。

2）节点收到对等节点的nodeInfo信息后，检查对等节点的版本号和网络信息是否与自己的是匹配的。

3）检查通过后，节点会查看对等节点是否在节点黑名单中，如果对等节点之前被记录在节点黑名单中，则返回ErrConnectBannedPeer错误，同时会检查对等节点之前是否已将加入了PeerSet。

4）以上检查项通过后，sw.startInitPeer（peer）会启动goroutine收发来自对等节点的消息，同时也会将节点加入PeerSet中保存。

10.6 节点状态机

我们用状态机来控制对象状态转换的逻辑，通过不同的状态迁移完成一些特定的顺序逻辑。当需要转换到某个状态时，生成一个新的状态对象并触发新状态的一些特定操作。比如，P2P网络中的对等节点从unknow状态变成know状态时，当前状态机会生成一个新的know状态对象。当状态机转为know状态的时候，触发添加到路由表的操作，这就是状态转换后的特定操作。

节点状态机是“节点发现”中非常重要的一个应用。通过上面的分析我们发现无论是种子节点状态刷新还是节点对于对等节点的UDP数据包处理，最终都是交由节点状态机处理。节点状态机主要就是根据节点的响应数据包更新节点状态的。比原的节点状态共分七种，分别是：

```
p2p/discover/net.go
var (
    unknown          *nodeState
    verifyinit       *nodeState
    verifywait       *nodeState
    remoteverifywait *nodeState
    known            *nodeState
    contested        *nodeState
    unresponsive     *nodeState
)
```

节点状态机结构体如下：

```
type nodeState struct {
    name     string
```

```
    handle   func()
    enter    func()
    canQuery bool
}
```

节点状态机结构体说明如下。

· name：状态名，共有七种状态：unknown、verifyinit、verifywait、remoteverifywait、known、contested、unresponsive。

· handle：接收并处理其他节点的消息。

· enter：节点由状态A变为状态B时，需要执行状态B的entry函数做状态初始化。

· canQuery：是否可以向节点发送查询消息。

节点状态转换代码如下：

```
func (net *Network) transition(n *Node, next *nodeState) {
    if n.state != next {
        n.state = next
        if next.enter != nil {
            next.enter(net, n)
        }
    }
}
```

transition函数是做节点状态转移的。transition函数需要两个参数：节点node和节点的下一个状态next。transition函数执行的时候，会先将节点的状态变更为next，然后执行next状态对应的enter函数。这里存在一个问题——并不是所有的状态都有enter函数。例如，unresponsive和verifywait并没有实现enter函数。所以在执行enter函数前需要判断节点是否实现了enter函数。

通常在以下几种情况下，会发生节点状态的改变：

· refresh方法刷新节点状态时，请求的节点如果是unknown状态，则会将状态变为verifyinit。

· findnodeQuery方法向remote节点查询其他节点的信息时，remote节点状态必须是known，如果节点状态为unknown，会通过transition方法将节点状态变更为verifyinit。之后通过在节点间发送探测包，将节点状态慢慢转移为known。

· handleNeighboursPacket方法接收到remote节点发送的邻居节点，如果邻居节点的状态为unknown，则会通过transition方法将邻居节点状态转移为verifyinit。

· 当节点状态为known时，需要将节点加入到K桶中，如果节点对应的K桶满了，需要将节点变为contested节点。

节点状态转换如图10-12所示。

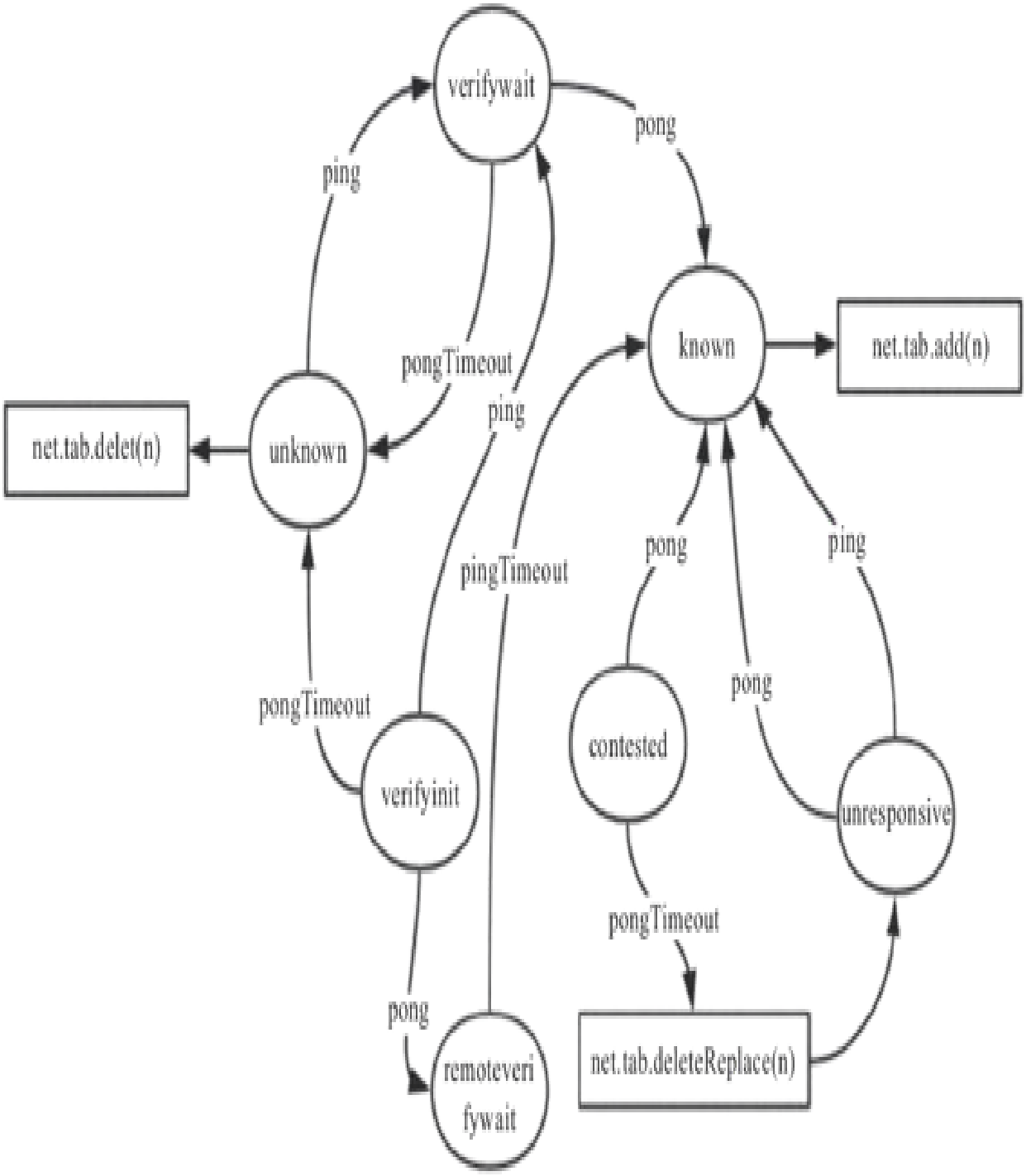


图10-12　节点状态机

通过以上状态机信息分析得出，当对等节点状态为known时，节点会将对等节点加入到路由表中。当对等节点状态为unknown时，节点会从路由表中删除对等节点信息。状态转换说明如下。

（1）当对等节点状态为unknown时

节点收到对等节点的Ping包，节点会给对等节点响应一个Pong包。同时，节点再向对等节点发送Ping包，这时节点将对等节点的状态由unknown设置为verifywait。

enter操作：如果对等节点状态为unknown，则从路由表中清理对等节点信息。

（2）当对等节点状态为verifyinit时

节点收到对等节点的ping包，节点会给对等节点响应一个pong包，并将对等节点状态设置为verifywait。

节点收到对等节点的pong包，节点将对等节点状态设置为remoteverifywait。

节点向对等节点发送ping包，如果节点在pongTimeout（1s）内没有收到对等节点发回的包，节点会将对等节点状态设置为unknown。

enter操作：节点向对等节点发送ping包。

（3）当对等节点状态为verifywait时

节点收到对等节点的ping包，节点会给对等节点响应一个pong包，并将对等节点状态设置为verifywait。

节点收到对等节点的pong包，节点将对等节点状态设置为known。

节点向对等节点发送ping包，如果节点在pongTimeout（1s）内没有收到对等节点发回的包，节点会将对等节点状态设置为unknown。

enter操作：无。

（4）当对等节点状态为remoteverifywait时

节点收到对等节点的ping包，节点会给对等节点响应一个pong包，并将对等节点状态设置为remoteverifywait。

节点之前发送的ping包，在pingtTimeout（1s）时间内没有收到对等节点的响应，节点会将对等节点状态设置为known。

enter操作：等待pongTimeout事件。

（5）当对等节点状态为known时

节点收到对等节点的ping包，节点会给对等节点响应一个pong包，并将对等节点状态设置为known。

节点收到对等节点的pong包，节点将对等节点状态设置为known。

enter操作：节点将对等节点加入路由表。

（6）当对等节点状态为contested时

节点收到对等节点的pong包，节点将对等节点状态设置为known。

节点向对等节点发送ping包，如果节点在pongTimeout（1s）内没有收到对等节点发回的包，节点会将对等节点状态设置为unresponsive。

节点收到对等节点的ping包，节点将对等节点状态设置为contested。

enter操作：节点向对等节点发送ping包。

（7）当对等节点状态为unresponsive时

节点收到对等节点的ping包，节点会给对等节点响应一个pong包，并将对等节点状态设置为known。

节点收到对等节点的pong包，节点将对等节点状态设置为known。

enter操作：无。

10.7 区块同步

10.7.1 区块同步流程

在公链设计中，区块同步可分为快速同步和普通同步。随着时间的推移，区块的数量会越来越多，新的节点加入区块链网络时从创世区块开始同步，速度较慢。为了解决新节点同步慢的问题，需要支持快速同步算法。区块同步流程如图10-13所示。

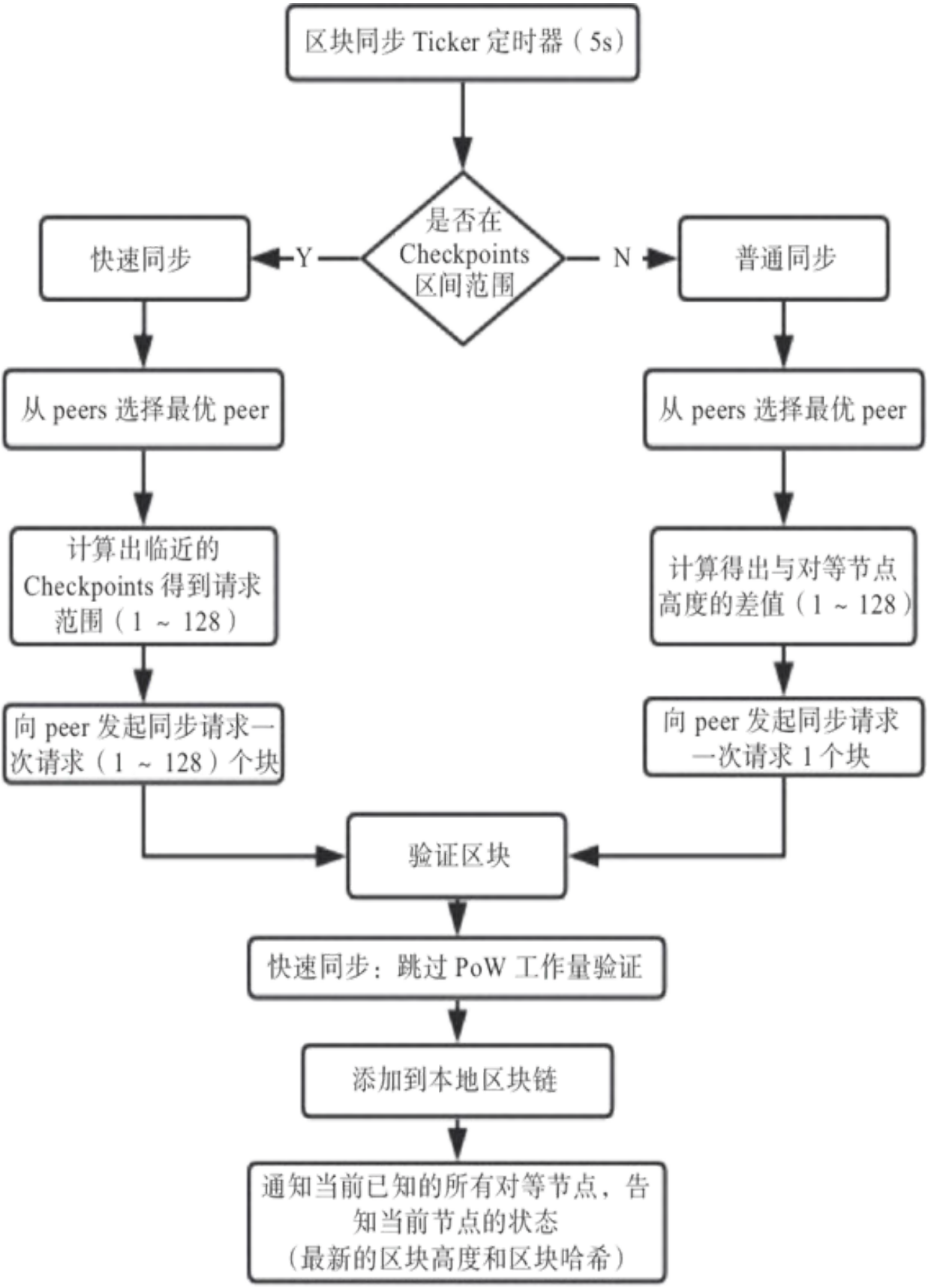


图10-13　区块同步流程

快速同步和普通同步的概念：

· 区块快速同步。同步定时器触发时，检测当前节点的高度是否在Checkpoints区间中，如果在Checkpoints区间范围内则使用快速同步，节点向对等节点一次请求1～128个区块（根据当前高度计算出所在Checkpoints的区间范围，决定每次向对等节点请求的区块数量）。最后验证所有的区块并添加到本地区块链上（快速同步过程中会跳过PoW工作量验证的步骤）。在快速同步过程中，对等节点需验证节点是否存在分叉。

· 区块普通同步。同步定时器触发时，如果当前节点的高度不在Checkpoints区间中，则使用普通同步。根据当前节点高度与对等节点高度计算出的差值，节点每次向对等节点请求1个区块。最后验证区块并添加到主链上。

1. blockerKeeper区块同步结构

在比原链中区块同步的操作主要由SyncManager的blockKeeper方法实现。blockerKeeper结构如下：

```
netsync/block_keeper.go
type headersMsg struct {
    headers []*types.BlockHeader
    peerID  string
}

type blockKeeper struct {
    chain Chain
    peers *peerSet

    syncPeer          *peer
    blockProcessCh    chan *blockMsg
    blocksProcessCh   chan *blocksMsg
    headersProcessCh chan *headersMsg

    headerList *list.List
}
```

headersMsg结构字段说明如下。

· headers：对等节点同步的区块头信息。

· peerID：对等节点ID，用于验证数据来源。

blockKeeper结构字段说明如下。

· syncPeer：当前同步的对等节点。每次执行同步操作时会从peerSet中选择最优节点进行同步，所以每次同步时对等节点可能会变更。

· blockProcessCh：单区块同步通道。

· blocksProcessCh：多区块同步通道。

· headersProcessCh：区块头同步通道。

· headerList*list.List：双向链表数据结构，用于保存快速同步过程中获得的多个区块头的缓存区。

2. blockerKeeper区块同步初始化

blockerKeeper的初始化是在NewSyncManager时完成，blockKeeper在完成初始化的同时，会启动一个goroutine执行syncWorker方法，开始区块同步工作。syncWorker其实是一个定时器，方法中首先声明了一个time tricker定时器，tricker计时周期为5秒。然后通过for循环持续地检测tricker是否到期。计时器到期后，开始执行startSync同步区块。代码示例如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) syncWorker() {
    // ...
    syncTicker := time.NewTicker(syncCycle)
    for {
        <-syncTicker.C
        if update := bk.startSync(); !update {
            continue
        }
```

```
        block, err :=
bk.chain.GetBlockByHeight(bk.chain.BestBlockHeight())
        if err != nil {
            log.WithField("err", err).Error("fail on syncWorker
get best block")
        }

        if err := bk.peers.broadcastMinedBlock(block); err !=
nil {
            log.WithField("err", err).Error("fail on syncWorker
broadcast new block")
        }

        if err = bk.peers.broadcastNewStatus(block,
genesisBlock); err != nil {
            log.WithField("err", err).Error("fail on syncWorker
broadcast new status")
        }
    }
}
```

在syncWorker中，当bk.startSync同步到了新的区块时，会将本地区块链中最高的区块通过bk.peers.broadcastMinedBlock函数广播给当前已知的对等节点（更严格来说应该是“未知该区块的所有对等节点”）。最后通过bk.peers.broadcastNewStatus方法向当前已知的所有对等节点告知当前节点的状态（最新的区块高度和区块哈希）。

10.7.2 快速同步算法

10.7.1节展示了快速同步和普通同步的区别及过程。本节详细展开快速同步算法的实现。代码示例如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) startSync() bool {
    checkPoint := bk.nextCheckpoint()
    peer := bk.peers.bestPeer(consensus.SFFastSync |
consensus.SFFullNode)
    if peer != nil && checkPoint != nil && peer.Height() >=
checkPoint.Height {
        bk.syncPeer = peer
        if err := bk.fastBlockSync(checkPoint); err != nil {
            log.WithField("err", err).Warning("fail on
fastBlockSync")
            bk.peers.errorHandler(peer.ID(), err)
            return false
        }
        return true
    }
    // ...
}
```

首先计算得到临近的Checkpoint，然后从当前节点的peers中选出最优的peer节点。选择最优节点的方式为，遍历当前节点已知的所有peers，找到区块最高的peer，则该peer为最优节点。最后执行fastBlockSync。

1.计算得到临近的Checkpoint

代码如下：

```
func (bk *blockKeeper) nextCheckpoint() *consensus.Checkpoint {
    height := bk.chain.BestBlockHeader().Height
    checkpoints := consensus.ActiveNetParams.Checkpoints
    if len(checkpoints) == 0 || height >=
checkpoints[len(checkpoints)-1].Height {
        return nil
    }
```

```
    nextCheckpoint := &checkpoints[len(checkpoints)-1]
    for i := len(checkpoints) - 2; i >= 0; i-- {
        if height >= checkpoints[i].Height {
            break
        }
        nextCheckpoint = &checkpoints[i]
    }
    return nextCheckpoint
}
```

---

nextCheckpoint函数的作用是，根据当前高度计算当前节点高度所在Checkpoints的临近位置。假设当前节点的高度为48000，那我们最终得到的是高度为50000的Checkpoint并返回。

2. 找出最优节点

代码如下：

```
netsync/peer.go
func (ps *peerSet) bestPeer(flag consensus.ServiceFlag) *peer {
    ps.mtx.RLock()
    defer ps.mtx.RUnlock()

    var bestPeer *peer
    for _, p := range ps.peers {
        if !p.services.IsEnable(flag) {
            continue
        }
        if bestPeer == nil || p.height > bestPeer.height {
            bestPeer = p
        }
    }
    return bestPeer
}
```

遍历当前节点已知的所有peers，找到区块高度最高的peer，则该peer为最优节点。

3. 快速同步

代码如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) fastBlockSync(checkPoint
*consensus.Checkpoint) error {
    bk.resetHeaderState()
    lastHeader := bk.headerList.Back().Value.
(*types.BlockHeader)
    for ; lastHeader.Hash() != checkPoint.Hash; lastHeader =
bk.headerList.Back().Value.(*types.BlockHeader) {
        if lastHeader.Height >= checkPoint.Height {
            return errors.Wrap(errPeerMisbehave, "peer is not in
the checkpoint branch")
        }

        lastHash := lastHeader.Hash()
        headers, err := bk.requireHeaders([]*bc.Hash{&lastHash},
&checkPoint.Hash)
        // ...

        if err := bk.appendHeaderList(headers); err != nil {
            return err
        }
    }

    fastHeader := bk.headerList.Front()
    for bk.chain.BestBlockHeight() < checkPoint.Height {
        locator := bk.blockLocator()
        blocks, err := bk.requireBlocks(locator,
&checkPoint.Hash)
        // ...

        for _, block := range blocks {
            if fastHeader = fastHeader.Next(); fastHeader == nil
{
                return errors.New("get block than is higher than
checkpoint")
            }
            // ...
            seed, err :=
bk.chain.CalcNextSeed(&block.PreviousBlockHash)
            if err != nil {
                return errors.Wrap(err, "fail on fastBlockSync
calculate seed")
            }

            tensority.AIHash.AddCache(&blockHash, seed,
&bc.Hash{})
            _, err = bk.chain.ProcessBlock(block)
```

```
            tensority.AIHash.RemoveCache(&blockHash, seed)
            if err != nil {
                return errors.Wrap(err, "fail on fastBlockSync
process block")
            }
        }
    }
    return nil
}
```

快速同步算法如图10-14所示。

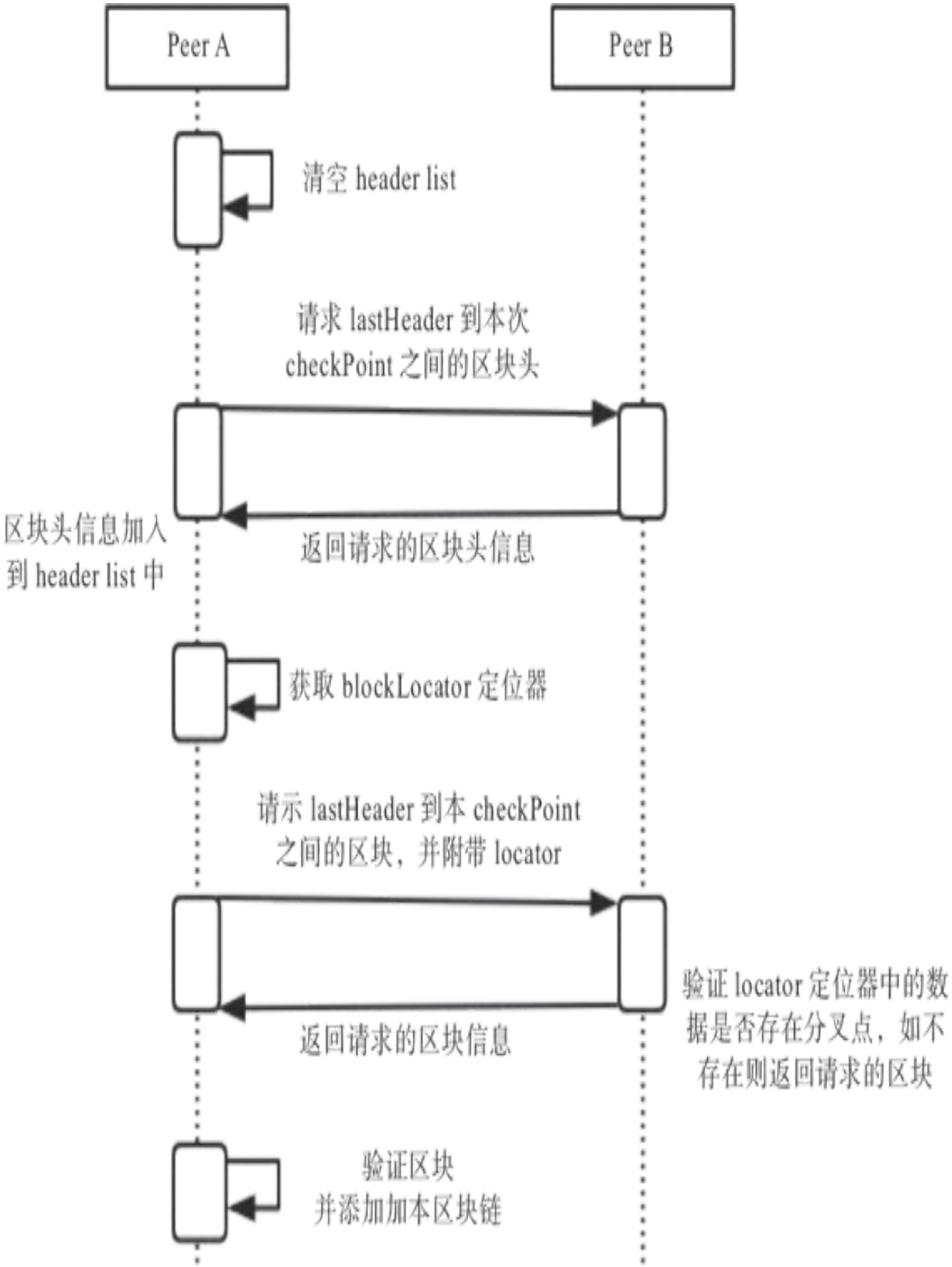


图10-14　快速同步算法

快速同步算法流程如下：

1）清空header list，获取节点当前区块链最高的区块头lastHeader，加入到header list中。

2）向best peer请求lastHeader到本次checkPoint之间的区块头。

3）best peer返回lastHeader到本次checkPoint之间的区块头信息。

4）节点将best peer返回的区块头信息加入到header list中。

5）节点向best peer请求节点当前区块到checkPoint之间区块信息，并附带block locator定位器信息，定位器用于检测是否存在分叉点。

6）best peer返回当前区块到checkPoint之间区块信息。

7）当前节点对返回的区块信息进程验证，验证通过后加入到节点本地区块链中。

4.blockLocator区块定位器

在一个互相不信任的区块链网络中，如何检测对等节点是否是分叉节点？这就需要使用locator定位器，找到区块中是否有分叉点。定位器在下载链头附近较为密集，随着它朝向起源呈指数级稀疏。代码示例如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) blockLocator() []*bc.Hash {
    header := bk.chain.BestBlockHeader()
    locator := []*bc.Hash{}

    step := uint64(1)
    for header != nil {
        headerHash := header.Hash()
        locator = append(locator, &headerHash)
```

```
        if header.Height == 0 {
            break
        }

        var err error
        if header.Height < step {
            header, err = bk.chain.GetHeaderByHeight(0)
        } else {
            header, err =
bk.chain.GetHeaderByHeight(header.Height - step)
        }
        if err != nil {
            log.WithField("err", err).Error("blockKeeper fail on
get blockLocator")
            break
        }

        if len(locator) >= 9 {
            step *= 2
        }
    }
    return locator
}
```

执行原理：将本地区块链按照高度从后往前遍历，先获取前9个header，后续按照二分法取header。比如，locator中定位在“当前高度，当前高度－1，当前高度－2，当前高度－n，当前高度/2，当前高度/4，当前高度/8，当前高度/16”等。

得到locator区块定位器以后，需将locator发送给对等节点，代码示例如下：

```
locator := bk.blockLocator()
blocks, err := bk.requireBlocks(locator, &checkPoint.Hash)
if err != nil {
    return err
}
```

当前节点每次请求对等节点时，将locator传给对等节点，对等节点验证locator定位器中的数据是否存在分叉点，如不存在分叉点则返回请求的区块。

10.7.3 普通同步算法

与快速同步相比，普通同步比较简单，peer与peer之间不需要检查是否存在分叉点。代码示例如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) startSync() bool {
    // ...
    blockHeight := bk.chain.BestBlockHeight()
    peer = bk.peers.bestPeer(consensus.SFFullNode)
    if peer != nil && peer.Height() > blockHeight {
        bk.syncPeer = peer
        targetHeight := blockHeight + maxBlockPerMsg
        if targetHeight > peer.Height() {
            targetHeight = peer.Height()
        }

        if err := bk.regularBlockSync(targetHeight); err != nil
{
        // ...
    }
}
```

首先获取当前节点最高区块，然后从当前节点的peers中选出最优的peer节点。根据对等节点的高度与当前节点的高度计算出差值（该差值范围在1-128），得到的差值即是当前节点所需要同步的区块高度。最后执行regularBlockSync。

普通同步代码示例如下：

```
func (bk *blockKeeper) regularBlockSync(wantHeight uint64) error
{
    i := bk.chain.BestBlockHeight() + 1
    for i <= wantHeight {
        block, err := bk.requireBlock(i)
        if err != nil {
            return err
        }

        isOrphan, err := bk.chain.ProcessBlock(block)
        if err != nil {
```

```
            return err
        }

        if isOrphan {
            i--
            continue
        }
        i = bk.chain.BestBlockHeight() + 1
    }
    return nil
}
```

普通同步算法如图10-15所示。

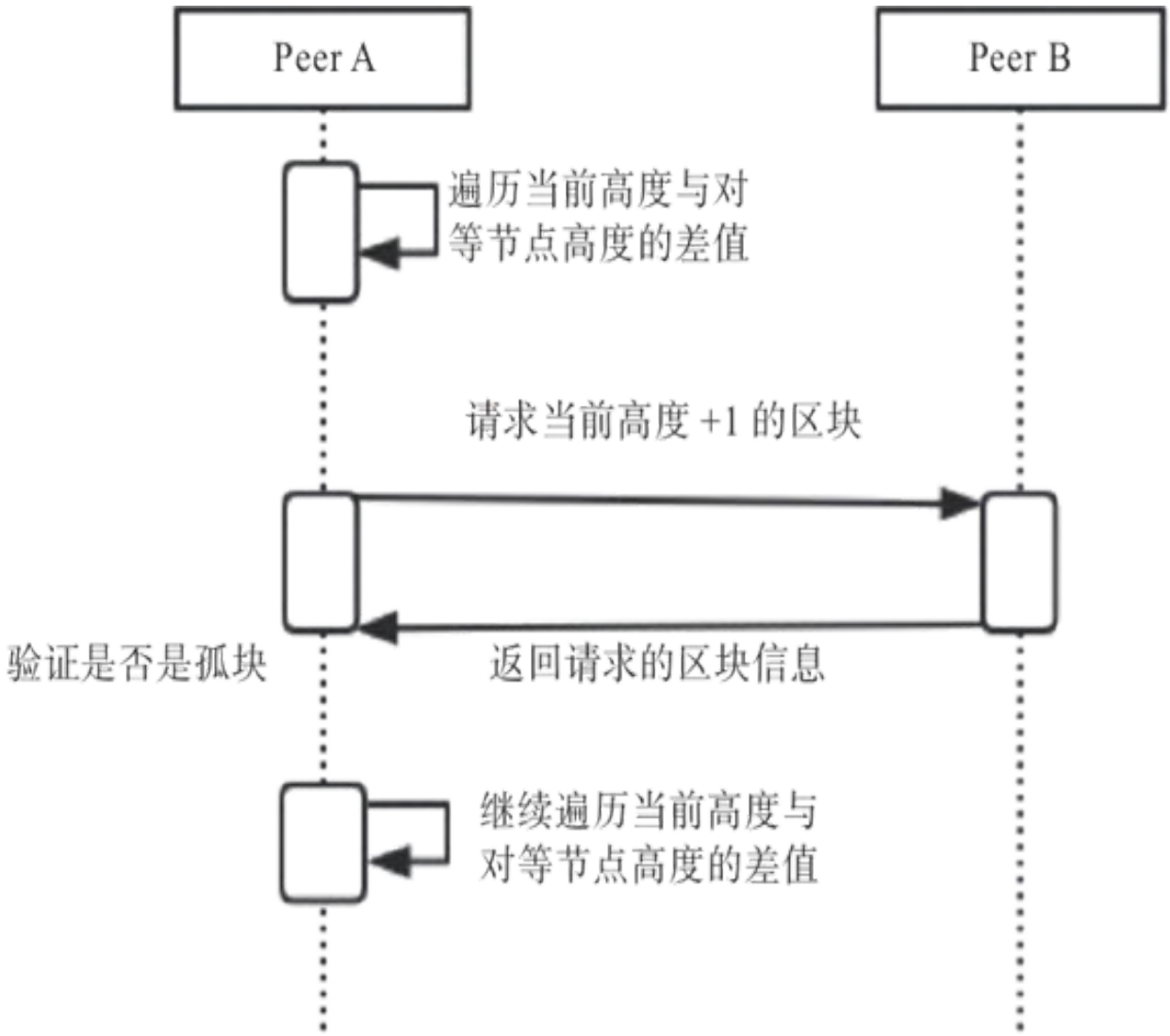


图10-15　普通同步算法

普通同步算法流程如下：

1）遍历当前节点最高高度与对等节点最高高度差值。

2）每次向对等节点请求当前高度+1的区块。

3）对等节点返回请求区块。

4）当前节点使用bk.chain.ProcessBlock验证同步的区块，验证该区块是否是孤块。如果是孤块则存入孤块池，并继续请求当前高度父区块；如果不是孤块则添加到本地区块链中，且高度+1。

10.7.4 区块数据请求与发送

请求区块数据主要通过netsync包下peer struct中的方法来实现。peer对象中提供多种get请求和send发送方法。

1. 区块请求

peer的区块请求有getBlockByHeight、getBlocks、getHeaders等方法，请求方法说明如下。

· getBlockByHeight：向对等节点请求指定的高度区块。

· getBlocks：向对等节点请求指定区块范围（1～128），locator参数为定位器中的数据，用于对等节点检验是否存在分叉点。

· getHeaders：向对等节点请求指定的区块头范围（1～128）。

peer中的get和send方法都调用同一个TrySend函数来实现请求和数据发送。我们以getBlocks方法为例说明该过程，代码示例如下：

```
netsync/peer.go
func (p *peer) getBlocks(locator []*bc.Hash, stopHash *bc.Hash)
bool {
    msg := struct{ BlockchainMessage }
{NewGetBlocksMessage(locator, stopHash)}
    return p.TrySend(BlockchainChannel, msg)
}
```

peer中的get和send方法都是先构造请求消息msg。msg的构造结构定义在“netsync/message.go”下的BlockchainMessage。BlockchainMessage相当于节点与对等节点之间消息通信的协议。

NewGetBlocksMessage中主要包含的是需要请求的区块信息。可使用p.TrySend方法将消息发送出去，代码示例如下：

```
p2p/peer.go
func (p *Peer) TrySend(chID byte, msg interface{}) bool {
```

```
    if !p.IsRunning() {
        return false
    }
    return p.mconn.TrySend(chID, msg)
}
```

TrySend方法中，mconn是MConnection的实例，TrySend接收两个函数，chID为消息通信管道，msg为消息内容。实际调用的是MConnection的TrySend方法来发送数据包。MConnection的TrySend方法根据chID找到相应的channel。在发送数据的时候，使用了github.com/tendermint/go-wire库，将msg序列化为二进制数组。

channel.trySendBytes方法并没有直接把数据发送出去，而是把要发送的数据放到了该channel对应的sendQueue中，交由其他函数发送，并通知c.send。在sendRoutine()中，接收到c.send通知。最终使用c.sendSomeMsgPackets()将数据发送给对方。代码示例如下：

```
//p2p/connection/connection.go
func (c *MConnection) sendRoutine()
case <-c.send:
    if eof := c.sendSomeMsgPackets(); !eof {
        select {
        case c.send <- struct{}{}:
        default:
        }
    }
```

2. 发送和接收速率限制

在c.sendSomeMsgPackets()方法中，发送数据包会限制速率，代码示例如下：

```
func (c *MConnection) sendSomeMsgPackets() bool {
    c.sendMonitor.Limit(maxMsgPacketTotalSize,
atomic.LoadInt64(&c.config.SendRate), true)
    for i := 0; i < numBatchMsgPackets; i++ {
        if c.sendMsgPacket() {
            return true
        }
```

```
    }
    return false
}
```

c.sendMonitor.Limit的作用是限制发送速率，第一个参数maxMsgPacketTotalSize即每个packet的最大长度为常量1034，第二个参数是预先指定的发送速率，默认值为500KB/s，第三个参数是当实际速度过大时，是否暂停发送，直到变得正常。接收速率的限制与发送时类似。

3.区块发送

MConnection启动时，不仅启动了c.sendRoutine()用于发送数据的goroutine，还启动了c.recvRoutine()用于接收数据的goroutine。代码示例如下：

```
func (c *MConnection) recvRoutine() {
    // ...
    case packetTypeMsg:
        // ...
        msgBytes, err := channel.recvMsgPacket(pkt)
        if err != nil {
            if c.IsRunning() {
                // ...
                c.stopForError(err)
            }
            return
        }
}
```

在recvRoutine()中，从c.bufReader中读取下一个数据包类型。packetTypePing和packetTypePong为心跳类型，packetTypeMsg是数据传输类型。channel.recvMsgPacket（pkt）的作用是从packet包中读取相应的二进制数据。

recvMsgPacket使用了recving的通道，把packet中的字节数组加到其中，判断该packet是否EOF，如果是，则代表整个数据包完整接收到了，把ch.recving的内容完整返回，供上层逻辑处理；否则返回一个nil，表示还有未读取到完整的数据包。

10.8　交易同步

10.7节介绍了区块同步相关流程，本节介绍交易同步相关流程。

在P2P网络节点启动时，会启动一个goroutine专门负责处理新添加节点的交易信息同步。当一个新的对等节点加密握手后，当前节点将会当前处于pending状态的交易打包发送到对等节点。比原链中为了减少交易同步占用的节点网络带宽，要求节点一次最多只能同步102400条交易信息。

交易同步处理流程如图10-16所示。

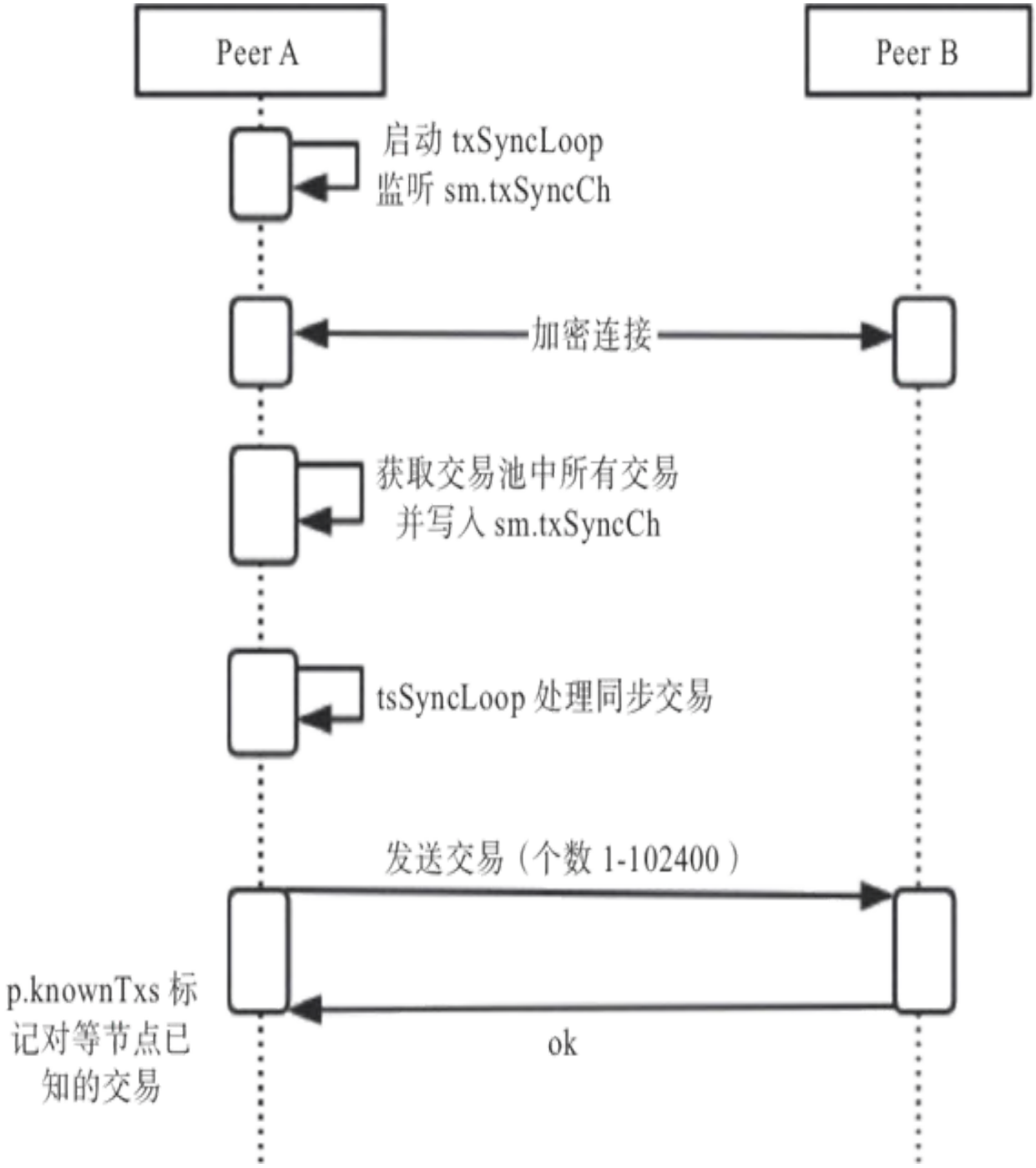


图10-16　交易同步流程

交易同步流程如下：

1）启动txSyncLoop gorutine监听sm.txSyncCh通道。

2）当前节点与对等节点建立加密连接通道。

3）当前节点获取本地交易池中所有交易并写入sm.txSyncCh通道。

4）当前节点过滤从交易池中获取的交易，限制交易个数的范围（1～102400），防止节点占用过多的网络带宽。

5）将过滤后的交易发送给对等节点。

6）当前节点发送成功后，修改本地的p.knownTxs（存储对等节点已知的交易哈希）集合中。

7）如果在交易同步过程中出现同步失败，节点会通过pick方法将交易从新加入pending中，等待重试。

代码示例如下：

```
netsync/tx_keeper.go
func (sm *SyncManager) txSyncLoop() {
    pending := make(map[string]*txSyncMsg)
    sending := false            // whether a send is active
    done := make(chan error, 1) // result of the send
    // ...
    for {
        select {
        case msg := <-sm.txSyncCh:
            pending[msg.peerID] = msg
            if !sending {
                send(msg)
            }

        case err := <-done:
            sending = false
            if err != nil {
                log.WithField("err", err).Warning("fail on
txSyncLoop sending")
            }

            if s := pick(); s != nil {
                send(s)
            }
        }
```

```
    }
}
```

在txSyncLoop中接收sm.txSyncCH通道中的交易，通过send函数发送给对等节点。send函数会过滤交易池中的交易数据，比如交易数量范围（1～102400）。send发送代码示例如下：

```
send := func(msg *txSyncMsg) {
    peer := sm.peers.getPeer(msg.peerID)
    if peer == nil {
        delete(pending, msg.peerID)
        return
    }
    totalSize := uint64(0)
    sendTxs := []*types.Tx{}
    for i := 0; i < len(msg.txs) && totalSize < txSyncPackSize;
i++ {
        sendTxs = append(sendTxs, msg.txs[i])
        totalSize += msg.txs[i].SerializedSize
    }
    if len(msg.txs) == len(sendTxs) {
        delete(pending, msg.peerID)
    } else {
        msg.txs = msg.txs[len(sendTxs):]
    }
    sending = true
    go func() {
        ok, err := peer.sendTransactions(sendTxs)
        if !ok {
            sm.peers.removePeer(msg.peerID)
        }
        done <- err
    }()
}
```

10.9 快速广播

快速广播使网络中的交易能够快速被确认。比原链中设有支持快速广播的机制，节点新产生的交易和区块快速地在全网传播。比原链提供快速广播算法主要是为了解决以下两个问题：

· 保证交易被快速确认。交易只有被打包到区块，并被广播到全网所有的节点，这笔交易才算真正被确认。在网络中交易被提交得越早，被篡改的可能性就越低。因此，比原链为了新产生的交易能够被较早地打包到区块中，提供了快速传播算法，将交易迅速广播到其他节点，提高交易被打包的可能性。

· 保证新产生的区块快速地广播到全网节点。尤其对于矿工来说，他们需要在第一时间获得新块被挖掘的消息，以开始下一次的挖矿工作。在比原链中，新收到的区块高度如果和节点目前区块高度差大于64，节点将抛弃新收到的区块。因此，如果不能及时同步区块，将造成网络中出现大量的失败区块同步操作。

10.9.1 新交易快速广播

当前节点发生的任何交易，如果被验证通过，都将被加入到节点交易池中。比原链中实现了一个sm.newTxCh通道，sm.new.TxCh会同步交易池中的交易。发生在节点上的交易如果被验证成功，交易在加入交易池的同时，也会同步到sm.new.TxCh中。

当sm.new.TxCh收到了tx交易，把该笔发送给已知所有peers（不存在该笔交易的节点）。最后标记对等节点已知的交易哈希。

简单来说，交易快速广播的原理是，当节点收到一笔合法交易后，立即广播给当前节点的所有已知对等节点。

交易快速广播流程如图10-17所示。

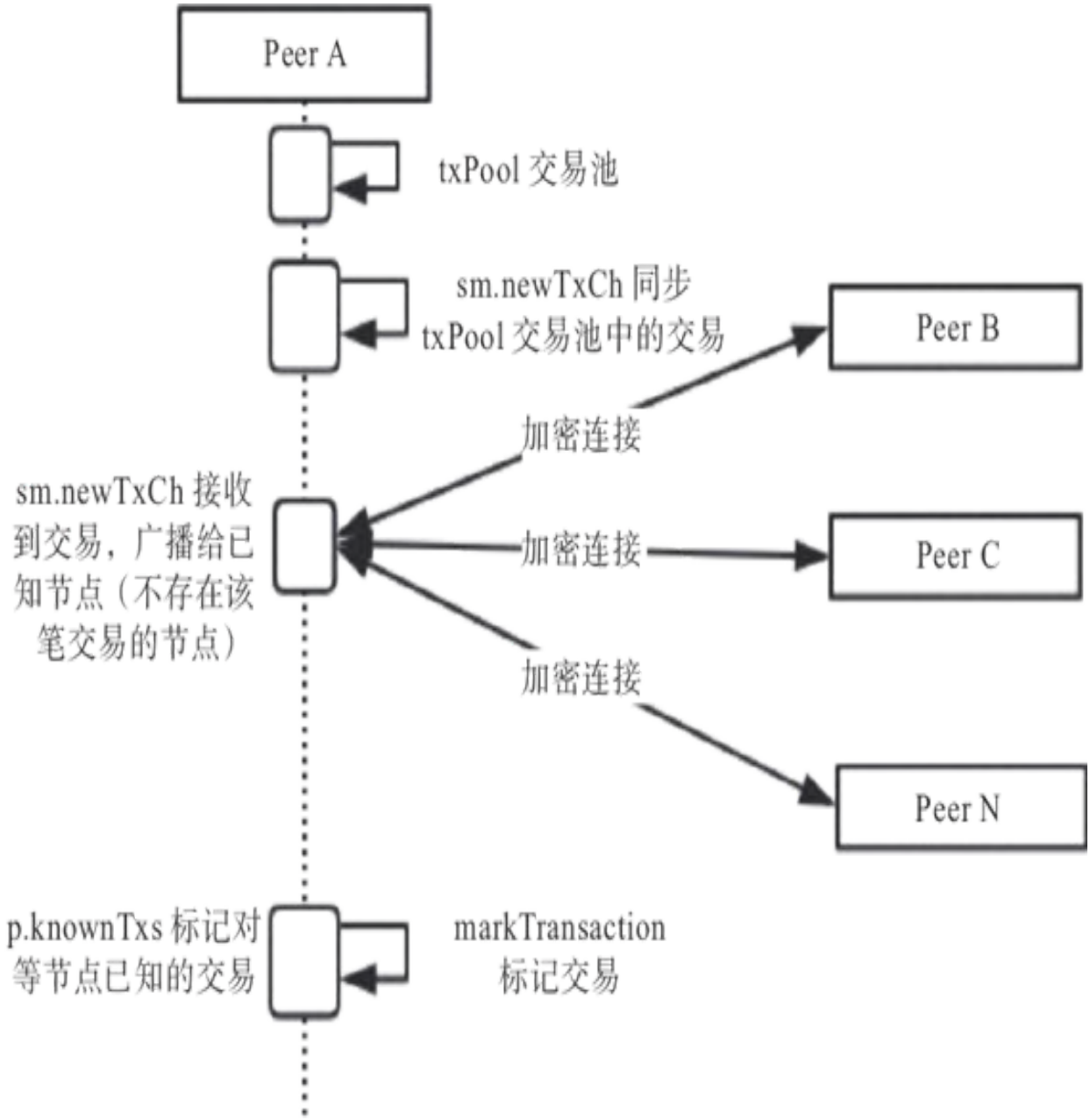


图10-17　交易快速广播

代码示例如下：

```
netsync/tx_keeper.go
func (sm *SyncManager) txBroadcastLoop() {
    for {
        select {
        case newTx := <-sm.newTxCh:
```

```
            if err := sm.peers.broadcastTx(newTx); err != nil {
                log.Errorf("Broadcast new tx error. %v", err)
                return
            }
        case <-sm.quitSync:
            return
        }
    }
}
```

---

txBroadcastLoop中，当sm.newTxCh收到了交易池同步的交易，则执行sm.peers.broadcastTx广播该交易。广播代码示例如下：

```
netsync/peer.go
func (ps *peerSet) broadcastTx(tx *types.Tx) error {
    msg, err := NewTransactionMessage(tx)
    if err != nil {
        return errors.Wrap(err, "fail on broadcast tx")
    }

    peers := ps.peersWithoutTx(&tx.ID)
    for _, peer := range peers {
        if peer.isSPVNode() && !peer.isRelatedTx(tx) {
            continue
        }
        if ok := peer.TrySend(BlockchainChannel, struct{
BlockchainMessage }{msg}); !ok {
            ps.removePeer(peer.ID())
            continue
        }
        peer.markTransaction(&tx.ID)
    }
    return nil
}
```

广播流程说明如下：

1）构建msg交易消息。

2）遍历所有对等节点，查询没有该笔交易的所有对等节点。

3）通过peer.TrySend发送给所有没有该笔交易的对等节点。

4）标记对等节点已有当前交易。TrySend方法在10.7节已经仔细分析过了，这里不再赘述。

10.9.2 新区块快速广播

区块快速广播主要是针对节点新挖掘出的区块信息。当挖矿节点通过PoW共识最终将交易打包成区块后，快速通知给已知节点广播金网，整体流程与交易快速广播类似。

区块快速广播流程如图10-18所示。

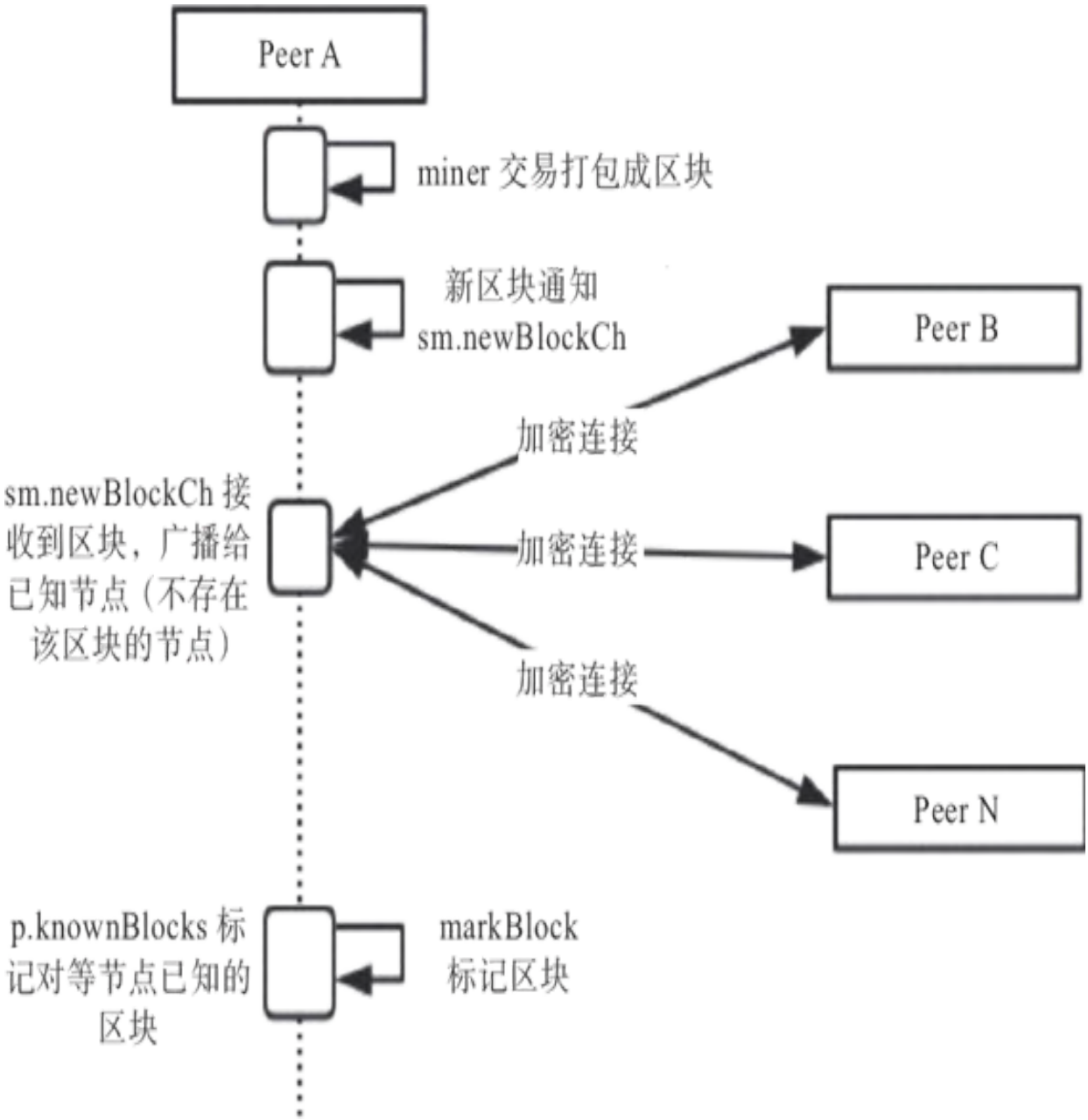


图10-18　区块快速广播

代码示例如下：

```
netsync/handle.go
func (sm *SyncManager) minedBroadcastLoop() {
    for {
        select {
        case blockHash := <-sm.newBlockCh:
            block, err := sm.chain.GetBlockByHash(blockHash)
            if err != nil {
                log.Errorf("Failed on mined broadcast loop get
block %v", err)
                return
            }
            if err := sm.peers.broadcastMinedBlock(block); err
!= nil {
                log.Errorf("Broadcast mine block error. %v",
err)
                return
            }
        case <-sm.quitSync:
            return
        }
    }
}
```

minedBroadcastLoop中，接收到sm.newBlockCh同步的区块信息（该区块信息是当前节点新挖掘出来的区块），广播给当前已知的所有节点。广播代码示例如下：

```
netsync/peer.go
func (ps *peerSet) broadcastMinedBlock(block *types.Block) error
{
    msg, err := NewMinedBlockMessage(block)
    // ...

    hash := block.Hash()
    peers := ps.peersWithoutBlock(&hash)
    for _, peer := range peers {
        if peer.isSPVNode() {
            continue
        }
```

```
        if ok := peer.TrySend(BlockchainChannel, struct{
BlockchainMessage }{msg}); !ok {
            ps.removePeer(peer.ID())
            continue
        }
        peer.markBlock(&hash)
    }
    return nil
}
```

广播区块流程说明如下：

1）构建msg区块消息。

2）遍历所有对等节点，找到没有该区块的所有对等节点。

3）通过peer.TrySend将消息发送给所有没有该区块的对等节点。

4）标记对等节点已有当前区块。

10.10 节点管理

节点连接管理主要为了解决以下问题：

· 如果连接节点数量过多，将会占用节点资源。比原链使用的是由Kademlia算法实现的结构化的P2P网络结构。每个节点都保存了与自己相邻节点的信息。依次类推，构建了比原链整个P2P网络结构。因此，节点在与其他节点建立P2P连接时，并不需要与全部的相邻节点都建立TCP连接。只要保证节点存在足够数量的连接节点，使节点到网络中任意一个节点能够找到一个链路即可。由此也可以保证节点产生的交易或者区块可以被广播到网络中任意一个节点。

· 如果节点连接数量过少，则节点获取网络中交易和区块信息将会有延迟。尤其对于挖矿节点，如果获取区块信息有延迟，则会严重影响矿机的挖矿效率和收益。

· 减少恶意节点对网络的影响。如果网络中存在较多的恶意节点，这些恶意节点会发送大量的无效交易和区块信息，这些无效信息不仅占用了整个网络的资源，也占用了节点的资源，降低节点同步数据的效率。

比原链中提供了以下几种节点连接管理策略。

10.10.1 TCP连接数管理

在公链设计中，我们不建议节点与过多的对等节点建立TCP连接。在比原链中节点默认最多可以同时与50个节点建立TCP连接。当TCP连接数超过50时，节点会主动关闭连接。这部分的实现主要在节点与对等节点交换密钥和完成握手之前，代码示例如下：

```
p2p/switch.go
func (sw *Switch) listenerRoutine(l Listener) {
    for {
        inConn, ok := <-l.Connections()
        if !ok {
            break
        }

        if sw.peers.Size() >= sw.Config.P2P.MaxNumPeers {
            inConn.Close()
            log.Info("Ignoring inbound connection: already have
enough peers.")
            continue
        }

        if err := sw.addPeerWithConnection(inConn); err != nil
{
            log.Info("Ignoring inbound connection: error while
adding peer.", " address:", inConn.RemoteAddr().String(), "
error:", err)
            continue
        }
    }
}
```

当有新的连接完成握手过程后，判断当前节点已连接的对等节点数是否超过了MaxNumPeers（默认50个），如果超过了，则当前节点主动关闭与该对等节点的连接。如果未超过MaxNumPeers，则与对等节点进行加密握手连接过程。

10.10.2 Outbound连接数管理

节点的连接状态主要分为三种：outbound、inbound和outbound-dialing。

哪类节点的状态是outbound？比如现在存在两个节点A和B。当节点A监听到节点B正在连接自己的TCP端口，节点A会与节点B完成一系列密钥交换等握手环节，之后节点A会将节点B信息保存到自己的peerSet列表中。此时节点B的连接状态就被设置为outbound。可以认为outbound状态是指节点之间已经建立了TCP连接。

inbound状态是指节点之间没有建立TCP连接。

outbound-dialing状态是指两个节点正处于建立连接的过程中。

比原链要求peerSet列表保存的节点中必须至少有5个节点处于inbound和outbound-dialing状态。其实，也可以认为比原链中要求节点至少要与5个节点建立TCP连接。如果建立连接的节点数不足5个，节点将从K桶中随机选择节点建立连接。代码示例如下：

```
func (sw *Switch) ensureOutboundPeers() {
    numOutPeers, _, numDialing := sw.NumPeers()
    numToDial := (minNumOutboundPeers - (numOutPeers +
numDialing))
    if numToDial <= 0 {
        return
    }

    connectedPeers := make(map[string]struct{})
    for _, peer := range sw.Peers().List() {
        connectedPeers[peer.RemoteAddrHost()] = struct{}{}
    }

    var wg sync.WaitGroup
    nodes := make([]*discover.Node, numToDial)
    n := sw.discv.ReadRandomNodes(nodes)
    for i := 0; i < n; i++ {
        try := NewNetAddressIPPort(nodes[i].IP, nodes[i].TCP)
        if sw.NodeInfo().ListenAddr == try.String() {
            continue
        }
```

```
        if dialling := sw.IsDialing(try); dialling {
            continue
        }
        if _, ok := connectedPeers[try.IP.String()]; ok {
            continue
        }

        wg.Add(1)
        go sw.dialPeerWorker(try, &wg)
    }
    wg.Wait()
}
```

当Outbound连接数管理由switch启动时，启动的一个goroutine每10秒钟执行一次ensureOutboundPeers方法处理。

10.10.3 动态节点评分机制DynamicBanScore

为防止网络中的恶意节点发送大量无效的数据包占用网络的带宽，需为公链设计一套评分机制。在比原链中引入了节点评分机制，节点根据收到的对等节点的数据包的合法性，给对等节点打分。当对等节点的分数累加到100，会认为对等节点是一个恶意节点。节点会将恶意节点写入本地黑名单，从节点peerSet中删除恶意节点信息，主动停止与恶意节点的TCP连接。同时，至少1小时内节点不会再次与该恶意节点建立TCP连接。

恶意节点评分机制采用的是一种动态评分机制。动态评分机制规定节点的最终评分由一个持久性分数和一个衰退性分数组成。持久性分数可以用来创建简单的禁止策略。衰退性分数用来创建规避逻辑，该逻辑通过断开并禁止尝试等手段来规避恶意节点在应用层上DDoS攻击。

任意一个节点的初始评分都被设置为0分。随着节点发送的各种数据包不被接收节点验证通过，评分会慢慢增加。

1.动态节点评分算法

DynamicBanScore结构如下：

```
p2p/trust/banscore.go
type DynamicBanScore struct {
    lastUnix   int64
    transient  float64
    persistent uint64
}
```

DynamicBanScore字段说明如下。

· lastUnix：最后调整分数的时间。

· transient：衰退性分数。该分数根据时间间隔动态变化。时间间隔一般是指当前调整分数时间距上次调整分数时间的间隔。

· persistent：持久性分数。该分数会被直接加入到节点评分中。

一个节点的分数是衰退性分数和持久性分数之和。每个节点的初始分数为0，评分代码示例如下：

```
p2p/trust/banscore.go
func (s *DynamicBanScore) increase(persistent, transient uint64,
t time.Time) uint64 {
    s.persistent += persistent
    tu := t.Unix()
    dt := tu - s.lastUnix

    if transient > 0 {
        if Lifetime < dt {
            s.transient = 0
        } else if s.transient > 1 && dt > 0 {
            s.transient *= decayFactor(dt)
        }
        s.transient += float64(transient)
        s.lastUnix = tu
    }
    return s.persistent + uint64(s.transient)
}
```

主动调节score值时，先将persistent值直接相加，然后算出传入时刻t的transient值，再与传入的transient值相加得到新的transient值，新的persistent与新的transient值相加得到新的score。实际上，就是t时刻的score加上传入的persistent和transient即得到新的score。

具体流程如下：

1）本次评分时间戳与上次评分时间戳之差大于Lifetime（1800s）时，节点的衰退性分数s.transient置为0。

2）如果衰退性分数大于1且本次评分时间戳与上次评分时间戳之差大于0时，则节点s.transient分数等于s.transient乘以一个衰减系数。其中，衰减系数随时间变化，时间越长，衰减系数越小。它由decayFactor()决定：

```
func decayFactor(t int64) float64 {
  if t < precomputedLen {
     return precomputedFactor[t]
  }
  return math.Exp(-1.0 * float64(t) * lambda)
}
```

衰减系数是随时间间隔呈指数分布的，其中Lambda=ln2/60。动态分值随时间时隔变化的曲线如图10-19所示。

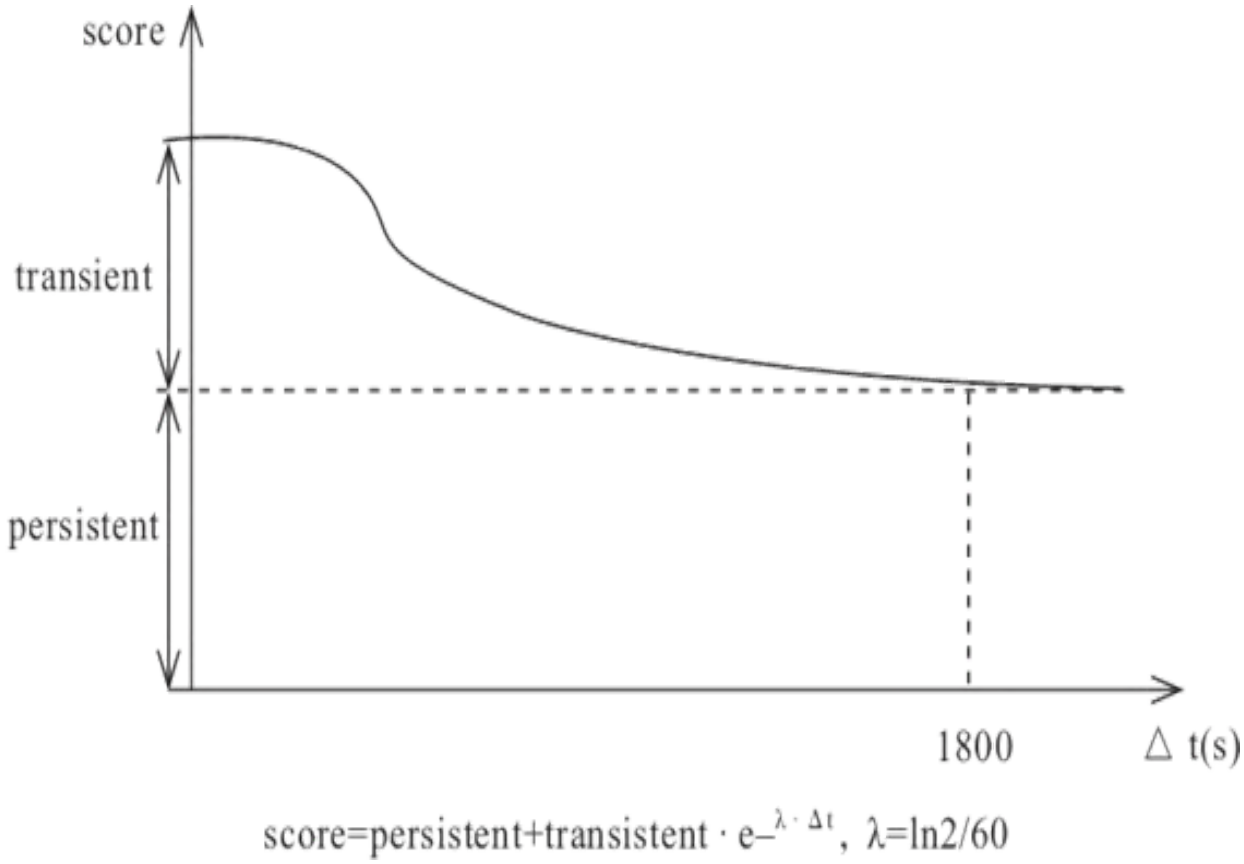


图10-19　动态分值变化曲线

3）increase函数返回该节点的分数（衰退性分数与持久性分数之和）。

总之，两次评分时间间隔越长，衰退性评分越小。节点的本次评分等于节点的持久性评分加上节点的衰退性评分。peer之间交换消息时，每一个peer连接会利用一个动态计分器监控节点之间收发消息的

频率，若某个peer发送过来的消息过于频繁，将怀疑遭到DDoS攻击，从而主动断开与它的连接。

2. 增加分数的设置

在如下情况下节点评分会增加：

· 节点发送的交易信息不能被接收节点获取到，接收节点会给节点增加10个衰退性分数。

· 节点发送的交易信息不能被接收节点验证通过，接收节点会给节点增加10点持久性分数。

代码示例如下：

```
netsync/handle.go
func (sm *SyncManager) handleTransactionMsg(peer *peer, msg
*TransactionMessage) {
    tx, err := msg.GetTransaction()
    if err != nil {
        sm.peers.addBanScore(peer.ID(), 0, 10, "fail on get tx
from message")
        return
    }

    if isOrphan, err := sm.chain.ValidateTx(tx); err != nil &&
isOrphan == false {
        sm.peers.addBanScore(peer.ID(), 10, 0, "fail on validate
tx transaction")
    }
}
```

· 当节点与对等节点同步区块信息时，如果发生了“peer is misbehave”异常，节点会给对等节点增加20分的持久性分数。

代码示例如下：

```
netsync/block_keeper.go
func (bk *blockKeeper) startSync() bool {
```

```
    // ...
    if err := bk.fastBlockSync(checkPoint); err != nil {
        log.WithField("err", err).Warning("fail on
fastBlockSync")
        bk.peers.errorHandler(peer.ID(), err)
        return false
    }

    // ...
    if err := bk.regularBlockSync(targetHeight); err != nil {
        log.WithField("err", err).Warning("fail on
regularBlockSync")
        bk.peers.errorHandler(peer.ID(), err)
        return false
    }
    // ...
}

func (ps *peerSet) errorHandler(peerID string, err error) {
    if errors.Root(err) == errPeerMisbehave {
        ps.addBanScore(peerID, 20, 0, err.Error())
    } else {
        ps.removePeer(peerID)
    }
}
```

· 当接收到其他节点新挖掘的区块信息，但从区块信息中取出的peerID不在本地peerSet中时，节点会将peerID对应节点的持久分数增加20分。

代码示例如下：

```
netsync/block_fetcher.go
func (f *blockFetcher) insert(msg *blockMsg) {
    if _, err := f.chain.ProcessBlock(msg.block); err != nil {
        peer := f.peers.getPeer(msg.peerID)
        if peer == nil {
            return
        }

        f.peers.addBanScore(msg.peerID, 20, 0, err.Error())
        return
    }
```

```
    //  ...
}
```

10.11　本章小结

P2P网络是区块链分布式网络结构的基础。本章详细介绍了P2P网络的基本的原理，包括节点发现算法、Kademlia协议原理、UPnP协议原理、节点管理机制等。同时，对比原链中如何实现P2P协议，做了细致的代码分析。在本章中，我们还详细介绍了比原链的两种区块同步算法——快速同步算法和普通同步算法。

第11章

数据存储

11.1 概述

在公链设计中，为了提高计算速度，一份数据可能会存储多份，因为计算速度与存储空间可以互换。通俗地讲，为了追求更快的计算速度，在代码实现层面往往会做很多优化，其中比较常见的就是用存储换取时间。比如，原始数据按照需要存好几份，每一份的数据结构可能都不一样，然后中间数据也有可能被存储下来，这样的好处就是每次计算的时候并不需要从头开始计算，只需要借助存储的力量，使计算量大幅精简从而提升整体计算速度。但是这种方式的缺点是占用较多的存储空间。

比原链在数据存储层上存储所有链上地址和资产交易等信息。数据存储层分为两部分：第一部分为缓存，大部分查询首先从缓冲开始，以减少对磁盘的IO压力。第二部分为持久化存储，当缓存中查询不到数据时，直接从持久化存储中读取，并添加到缓存中。

本章内容如下：

- 为何比原链数据存储使用LevelDB。
- LevelDB的常用操作。
- 比原链缓存的实现。
- 存储层持久化的实现。

11.2 为什么使用键值数据库

公链需要一个数据库来存储块信息、交易信息、交易状态、未使用的交易输出（UTXO）等。在公链架构设计体系中，我们不需要C/S体系结构的数据库，因为我们只需要快速简单地读取和快速批量地随机更新，所以有些高效的非类型数据库就比较合适。

LevelDB是Google开源的高性能非关系型数据库。LevelDB是单机的键值数据库，适合写多读少，支持持久化和故障恢复等，是基于LSM（Log-Structured-Merge Tree）的典型实现。LSM的原理是：当读写数据库时，首先将读写操作记录到Op log文件中，然后再操作内存数据库，当达到checkpoint时，则写入磁盘，同时删除相应的Op log文件，后续重新生成新的内存文件和Op log文件。

官方曾给出一份关于LevelDB的性能压测报告，参考资料 http://www.lmdb.tech/bench/microbench/benchmark.html。

根据LevelDB官方网站的描述，我们可以了解LevelDB的特点和限制。

LevelDB特点如下：

- key和value都是任意长度的字节数组。
- 数据按照key排序进行存储。
- 开发者也可以提供自定义的排序来存储。
- 提供的基本操作接口包括Put()、Delete()、Get()、Batch()。

· 支持原子操作进行批量修改。

· 用户可以创建一致性的snapshot快照，并允许在快照中查找数据。

· 数据可以通过正向或反向迭代遍历数据。

· 默认使用Snappy自动压缩数据。

· 可移植性较好，用户可自定义与操作系统交互。

LevelDB限制：

· 非关系型数据模型（NoSQL），不支持SQL语句，也不支持索引。

· 只有一个进程（可能是多线程），一次只允许一个进程访问一个特定的数据库。

· 没有内置的C/S架构，但可以自己封装一个Server。

比原链存储层就采用了LevelDB数据库。

11.3 LevelDB常用操作

本节总结一下LevelDB的常用操作，LevelDB数据库的快照、数据压缩等相关知识，请读者自行了解。

安装LevelDB如下所示：

```
go get github.com/syndtr/goleveldb/leveldb
```

11.3.1 增删改查操作

数据库一般都会提供最基本的增、删、改、查四种操作。下面介绍LevelDB中增删改查基本操作，代码如下：

```
package main

import (
    "fmt"

    "github.com/syndtr/goleveldb/leveldb"
)

var (
    KEY = "TESTKEY"
)

func main() {
    db, err := leveldb.OpenFile("/tmp/data/db", nil)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    if err = db.Put([]byte(KEY), []byte("This is a test."),
nil); err != nil {
        panic(err)
    }

    data, err := db.Get([]byte(KEY), nil)
    if err != nil {
        panic(err)
    }
    fmt.Printf("key:%v, value:%v\n", KEY, string(data))

    if err = db.Delete([]byte(KEY), nil); err != nil {
        panic(err)
    }
}

// Output
// key:TESTKEY, value:This is a test.
```

Output是执行该程序得到的输出结果。该程序对LevelDB进行了增删改查操作。LevelDB数据库以一个目录作为数据存储，leveldb.OpenFile打开数据库，如不存在则新建数据库。当main函数退出时执行db.Close()来关闭与数据库的连接。leveldb.OpenFile返回DB对象，DB对象提供了以下接口：

· db.Put()：设置key的value值，若key不存在则新建；若key存在则修改。

· db.Get()：得到key中value数据。

· db.Delete()：删除key及value的数据。

11.3.2 迭代查询

LevelDB中数据按照key的排序进行存储，遍历键值存储中所有数据需使用迭代器。迭代器可以通过正向迭代或反向迭代来遍历数据。代码如下：

```
db.Put([]byte("key_one"), []byte("one"), nil)
db.Put([]byte("key_two"), []byte("two"), nil)
db.Put([]byte("key_three"), []byte("three"), nil)

iter := db.NewIterator(nil, nil)
defer iter.Release()
for iter.Next() {
    key := iter.Key()
    value := iter.Value()
    fmt.Printf("key:%v, value:%v\n", string(key),
string(value))
}

// Output
// key:key_one, value:one
// key:key_three, value:three
// key:key_two, value:two
```

首先在数据库中分别Put三个key—key_one、key_two、key_three。db.NewIterator返回一个迭代器，若传入的第一个参数为nil，则表示不设置查询区间，即查询所有数据。当iter.Release()函数退出时，释放当前的迭代器。使用iter.Next()可以正向遍历数据，而iter.Prev()可以反向遍历数据。

11.3.3 按前缀查询

前缀查询与使用正则方式匹配所有key并返回被匹配到的key类似，前缀查询需要借助LevelDB提供util工具包"github.com/syndtr/goleveldb/leveldb/util"，代码如下：

```
db.Put([]byte("key_one"), []byte("one"), nil)
db.Put([]byte("key_two"), []byte("two"), nil)
db.Put([]byte("key_three"), []byte("three"), nil)

iter := db.NewIterator(util.BytesPrefix([]byte("key_")), nil)
defer iter.Release()
for iter.Next() {
    key := iter.Key()
    value := iter.Value()
    fmt.Printf("key:%v, value:%v\n", string(key),
string(value))
}

// Output
// key:key_one, value:one
// key:key_three, value:three
// key:key_two, value:two
```

util提供Range对象，Range对象提供start和limit设置区间用于不同维度的查询匹配key。util.BytesPrefix函数匹配前缀为“key_”的所有链值对。

11.3.4　批量操作

batch批量操作的原理是，先将所有的操作记录下来，然后再一起操作。前面的代码示例中，我们都是一次执行一个操作。下面我们使用批量操作写入多个key值数据。代码如下：

```
batch := new(leveldb.Batch)
batch.Put([]byte("key_four"), []byte("four"))
batch.Put([]byte("key_five"), []byte("five"))
batch.Put([]byte("key_six"), []byte("six"))
if err = db.Write(batch, nil); err != nil {
    panic(err)
}
```

new（leveldb.Batch）实例化batch对象，我们执行多次Put操作，此时的插入数据只是写入一条记录，并未实际写入磁盘中。最后执行db.Write操作将所有Put操作的记录写入磁盘中。

11.4 存储层缓存

缓存（cache）是数据交换的缓冲区。当某一进程要读取数据时，会首先从缓存中查找需要的数据，如果找到了则直接获取，若找不到则从缓存中找，从而实现快速访问，减少磁盘IO读写操作。比原链中缓存通过缓存淘汰算法（LRU）实现。

11.4.1 缓存淘汰算法

LRU算法名称是Least recently used，即最近最少使用算法，根据数据的历史访问记录来淘汰数据，如果数据最近被访问过，那么数据在缓存里被命中的几率也更高。LRU一般使用哈希地图（hash map）和双向链表（doubly linked list）实现，list保存数据，map做快速访问，算法实现如图11-1所示。

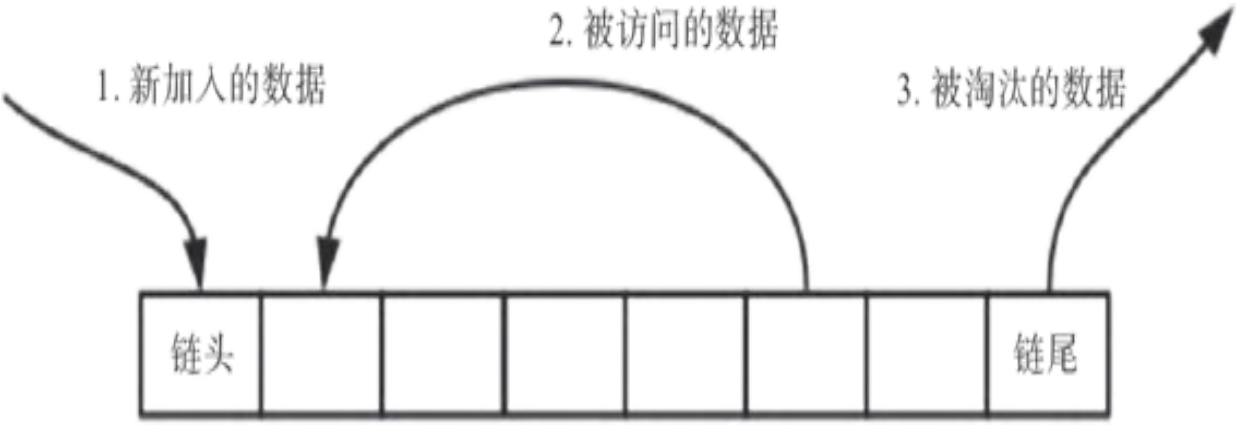


图11-1 LRU算法实现图

算法流程如下：

1）新的数据插入到链表头部。

2）每当数据缓存命中，则将数据移到链表头部。

3）当链表满的时候，将链表尾部的数据丢弃。

（1）LRU具体实现函数。

代码如下：

```
vendor/github.com/golang/groupcache/lru/lru.go
func New(maxEntries int) *Cache
func (c *Cache) Add(key Key, value interface{})
func (c *Cache) Get(key Key) (value interface{}, ok bool)
func (c *Cache) Remove(key Key)
func (c *Cache) RemoveOldest()
```

```
func (c *Cache) Len() int
func (c *Cache) Clear()
```

函数说明如下。

· Add()：添加数据到缓存中。

· Get()：从缓存中获取数据。

· Remove()：从缓存中移除指定数据。

· RemoveOldest()：从缓存中移除链尾数据。

· Len()：获取缓存长度。

· Clear()：清空缓存。

（2）实例化Cache结构体。

代码如下：

```
vendor/github.com/golang/groupcache/lru/lru.go
type Cache struct {
    MaxEntries int
    OnEvicted func(key Key, value interface{})
    ll    *list.List
    cache map[interface{}]*list.Element
}

func New(maxEntries int) *Cache {
    return &Cache{
        MaxEntries: maxEntries,
        ll:         list.New(),
        cache:      make(map[interface{}]*list.Element),
    }
}
```

函数说明如下。

· MaxEntries：最大缓存条目数，如果为0则表示无限制。

· ll：双向链表数据结构，用于保存数据。

· cache：基于hash map，快速访问到某个元素。

注意，当前Cache结构并非是线程安全，多并发访问时会导致数据不一致。

（3）添加数据到缓存中。

代码如下：

```
func (c *Cache) Add(key Key, value interface{}) {
    if c.cache == nil {
        c.cache = make(map[interface{}]*list.Element)
        c.ll = list.New()
    }
    if ee, ok := c.cache[key]; ok {
        c.ll.MoveToFront(ee)
        ee.Value.(*entry).value = value
        return
    }
    ele := c.ll.PushFront(&entry{key, value})
    c.cache[key] = ele
    if c.MaxEntries != 0 && c.ll.Len() > c.MaxEntries {
        c.RemoveOldest()
    }
}
```

Add添加元素有三步操作：

1）如果当前缓存中存在该元素，则将元素移至链表头并返回。

2）如果缓存中不存在该元素，则将元素插入链表头部。

3）如果缓存的元素超过MaxEntries限制，则移除链表最末尾的元素。

4）从缓存中获取数据。Get函数获取元素，如果数据命中则将元素移至链表头部，保持最新的访问。代码示例如下：

```
func (c *Cache) Get(key Key) (value interface{}, ok bool) {
    if c.cache == nil {
        return
    }
    if ele, hit := c.cache[key]; hit {
        c.ll.MoveToFront(ele)
        return ele.Value.(*entry).value, true
    }
    return
}
```

11.4.2 比原链缓存实现

比原链的缓存Block Cache实现有两个部分：通过LRU存储缓存数据，当读取数据时从cache中得到，实现快速访问。当缓存未命中时执行fillFn（回调函数）获取数据。

比原链的Block Cache实现过程如下：

1）执行GetBlock函数获取区块数据。

2）从Block Cache缓存池中查询，如果命中缓存则返回。

3）当Block Cache缓存池未命中则执行fillFn回调函数，从磁盘数据库（LevelDB）中获取并缓存至Block Cache中，然后返回。

1.blockCache实例化

1）blockCache结构体如下：

```
database/leveldb/cache.go
type blockCache struct {
    mu     sync.Mutex
    lru    *lru.Cache
    fillFn func(hash *bc.Hash) *types.Block
    single singleflight.Group
}
```

blockCache结构说明如下。

· mu：互斥锁，保证共享数据操作的一致性。

· lru：缓存淘汰。

· fillFn：回调函数。

· single：回调函数的调用机制，同一时间保证只有一个回调函数在执行。

2）blockCache实例化代码如下：

```
database/leveldb/cache.go
const maxCachedBlocks = 30
func newBlockCache(fillFn func(hash *bc.Hash) *types.Block)
blockCache {
    return blockCache{
        lru:    lru.New(maxCachedBlocks),
        fillFn: fillFn,
    }
}
database/leveldb/store.go
cache := newBlockCache(func(hash *bc.Hash) *types.Block {
    return GetBlock(db, hash)
})
```

在比原链中lru缓存对象并非是无限制的。默认情况下，lru缓存对象为30个，由maxCachedBlocks全局变量指定。若超过30个缓存对象，在新加入对象后，会逐出链表最末尾的对象。

在实例化blockCache对象时，需传入fillFn回调函数，以便从LevelDB数据库中查询回源数据。newBlockCache函数中指定GetBlock函数作为回调。

2.blockCache实现

当上层逻辑获取区块信息时，lookup函数从Block Cache缓存池中查询，如果命中缓存则返回区块信息。如果未命中则single.Do执行fillFn回调函数，从数据库中获取并缓存到Block Cache中，然后返回。代码如下：

```
database/leveldb/cache.go
func (c *blockCache) lookup(hash *bc.Hash) (*types.Block,
error) {
    if b, ok := c.get(hash); ok {
        return b, nil
    }

    block, err := c.single.Do(hash.String(), func()
(interface{}, error) {
```

```
        b := c.fillFn(hash)
        if b == nil {
            return nil, fmt.Errorf("There are no block with
given hash %s", hash.String())
        }

        c.add(b)
        return b, nil
    })
    if err != nil {
        return nil, err
    }
    return block.(*types.Block), nil
}

func (c *blockCache) get(hash *bc.Hash) (*types.Block, bool) {
    c.mu.Lock()
    block, ok := c.lru.Get(*hash)
    c.mu.Unlock()
    if block == nil {
        return nil, ok
    }
    return block.(*types.Block), ok
}
```

---

lookup函数执行流程如下：

1）c.get函数根据传入的block hash从缓存池中获取block信息，缓存命中则返回。

2）c.single.Do是回调函数的调用机制，作用如下：

a）执行并返回回调函数的执行结果。

b）确保只有一个执行在a）中的给定键。

c）如果有多个线程执行，则重复的调用者等待。

3）执行c.fillFn（hash）回调函数，从LevelDB中查询数据。

11.5 存储层持久化

一条公链正式运行以后，随着交易量的增加，块信息、交易信息等会逐渐增加，此时所有的数据并不能在内存中长久存储。在公链设计中，需要有持久化存储来保存块信息、交易信息等数据。

11.5.1 比原链数据库

比原链中db管理库使用tendermint框架。db管理库提供了数据库接口和许多方法实现，包括内存映射、文件系统目录结构。db管理库对多种键值数据库进行了封装，支持LevelDB、ClevelDB、GoLevelDB、MemDB等。

在比原链中一个节点会有多个LevelDB数据库，存储不同数据，数据存储在--home参数的data目录下。数据库的含义如下。

· accesstoken.db：token信息（钱包访问控制权限）。

· core.db：核心数据库，存储主链相关数据，包括块信息、交易信息、资产信息等。

· discover.db：分布式网络中端到端的节点信息。

· trusthistory.db：分布式网络中端到端的恶意节点信息。

· txfeeds.db：目前比原链代码版本未使用该功能，暂不介绍。

· wallet.db：本地钱包数据库，存储用户、资产、交易、UTOX等信息。

11.5.2 持久化存储接口

在比原链中数据持久化存储由database模块管理，但是持久化相关接口在protocol/store.go中，使用core.db核心数据库。上层逻辑通过持久化接口操作LevelDB数据库。持久化存储接口如下：

```
type Store interface {
    BlockExist(*bc.Hash) bool

    GetBlock(*bc.Hash) (*types.Block, error)
    GetStoreStatus() *BlockStoreState
    GetTransactionStatus(*bc.Hash) (*bc.TransactionStatus,
error)
    GetTransactionsUtxo(*state.UtxoViewpoint, []*bc.Tx) error
    GetUtxo(*bc.Hash) (*storage.UtxoEntry, error)

    LoadBlockIndex() (*state.BlockIndex, error)
    SaveBlock(*types.Block, *bc.TransactionStatus) error
    SaveChainStatus(*state.BlockNode, *state.UtxoViewpoint)
error
}
```

函数说明如下。

· BlockExist：根据哈希判断区块是否存在。

· GetBlock：根据哈希获取该区块。

· GetStoreStatus：获取主链最新的状态，当节点第一次启动时，节点会根据数据库中key为blockStore的内容判断是否初始化主链。

· GetTransactionStatus：返回block hash相关的交易状态。

· GetTransactionsUtxo：加载交易相关的所有UTXO。

· GetUtxo：根据交易的outputID hash获取所有UTXO。

· LoadBlockIndex：加载块索引，从db中读取所有块头信息并缓存在内存中。

· SaveBlock：保存块和交易状态。

· SaveChainStatus：保存主链最新的状态。

11.5.3 持久化key数据前缀

我们使用非关系型数据库存储持久化数据，为了标识不同的key，使用前缀来区分不同的key数据。代码示例如下：

```
database/leveldb/store.go
var (
    blockStoreKey     = []byte("blockStore")
    blockPrefix       = []byte("B:")
    blockHeaderPrefix = []byte("BH:")
    txStatusPrefix    = []byte("BTS:")
)
database/leveldb/utxo_view.go
const utxoPreFix = "UT:"
```

前缀说明如下：

- blockStoreKey：主链状态key名称。
- blockPrefix：块信息前缀。
- blockHeaderPrefix：块头信息前缀。
- txStatusPrefix：交易状态前缀。
- utxoPreFix，UTXO交易前缀。

除了blockStoreKey是完整的key命名之外，其他key以“前缀+哈希”的方式命名，比如，持久化存储一个哈希为“bm1q5”的块，那么在数据库中，它的key名为“B：bm1q5”，类型为[]byte。

11.5.4 持久化存储区块过程

当接收到对等网络节点同步的块并验证后，需要将块数据持久化到LevelDB中，过程如下：

```
func (s *Store) SaveBlock(block *types.Block, ts
*bc.TransactionStatus) error {
    binaryBlock, err := block.MarshalText()
    if err != nil {
        return errors.Wrap(err, "Marshal block meta")
    }

    binaryBlockHeader, err := block.BlockHeader.MarshalText()
    if err != nil {
        return errors.Wrap(err, "Marshal block header")
    }

    binaryTxStatus, err := proto.Marshal(ts)
    if err != nil {
        return errors.Wrap(err, "marshal block transaction
status")
    }

    blockHash := block.Hash()
    batch := s.db.NewBatch()
    batch.Set(calcBlockKey(&blockHash), binaryBlock)
    batch.Set(calcBlockHeaderKey(block.Height, &blockHash),
binaryBlockHeader)
    batch.Set(calcTxStatusKey(&blockHash), binaryTxStatus)
    batch.Write()

    log.WithFields(log.Fields{"height": block.Height, "hash":
blockHash.String()}).Info("block saved on disk")
    return nil
}
```

持久化存储区块共有四个步骤：

1）序列化块信息。

2）序列化块头信息。

3）序列化交易状态信息。

4）批量插入LevelDB数据库（原子操作）。calcBlockKey计算key的命名过程，最终key被命名为“块前缀+块hash”。

11.6 Varint变长编码

所有块数据和交易数据都通过Varint编码后存储到LevelDB中。block和tx对象都存在序列化和反序列化的方法，即MarshalText()和UnmarshalText()。这些方法均用于编解码数据。下面介绍下Varint变长编码的原理。

Varint是一种变长编码，相比于定长编码，编码小整数时可以只使用较少的字节数。它是一种紧凑的表示数字的编码方法。Varint用一个或多个字节来表示一个数字，值越小的数字使用越少的字节数，这能减少用来表示数字的字节数。

传统的integer类型是以32位来表示的，存储需要4个字节，如果整数大小在256之内，那么只需要用一个字节就可以存储这个整数，这样就可以节省3个字节的存储空间，Google Varint就是根据这种思想来序列化整数的。

在比原链中主要使用了Protocol Buffers协议中定义的varint编码方式来编码整形数值。Varint编码只使用每一个字节的低7位存储数值，而每字节的最高位用来标识整个Varint编码是否结束（最后一字节最高位为0，其他字节最高位为1），整个编码以低字节到高字节的顺序排列。

我们以块高度50000为例，将其转换为二进制数，并以7位分组：

```
(50000)10 = (11 0000110 1010000)2
```

从低字节到高字节转换：

原始数值 （由低字节到高字节）	varint 编码字节 （最高位表示是否结束）	16 进制表示
1010000	1 1010000	d0
0000110	1 0000110	86
11	0 0000011（补位，高位为 0 则结束）	03

根据上面转换得出，50000的Varint编码的16进制表示为d08603。将Varint编码转换为整数的过程与此相反。GO语言原生提供了Varint编解码的相关函数，位于encoding/binary包中。

11.7 本章小结

本章介绍了如何设计一条公链存储层。存储层共分为两层——缓存和持久化。缓存主要用于对象的快速访问，在公链设计中LRU使用比较广泛。持久化是将数据永久地写入磁盘中，我们对主流的LevelDB存储做了详细的操作介绍。

本章介绍了在比原链中使用tendermint框架的db管理库，对多种键值数据库进行封装，支持主流的键值数据库，如LevelDB、CleveldDB、GoLevelDB、MemDB等，供开发者选择。

第12章

共识算法

12.1 概述

共识算法是区块链稳定运行的核心，本章主要介绍挖矿及共识算法相关的内容。

本章主要内容包括：

- PoW（工作量证明）和PoS（权益证明）共识的原理。
- 使用GO语言实现一个简易PoW共识算法。
- 比原链Tensority共识算法。

12.2 PoW和PoS

在介绍挖矿共识算法之前，先介绍几个概念。

挖矿将交易打包为区块的过程称为挖矿，挖矿的人称为矿工。交易通过P2P协议广播到全网，节点需要验证收到的交易是否合法。并根据规则生成新的区块。

节点通过某种方式竞争“记账权”，得到记账权的节点可以将自己生成的区块追加到现有区块链末尾，获得块奖励和交易产生的手续费。这一过程类似于真实世界中付出劳动挖掘矿石的过程，所以被形象地称为“挖矿”，挖矿使用的设备或软件称为“矿机”。

算力指计算能力。矿机每秒能完成的哈希计算次数称为算力（单位是hash/s）。算力越高，得到区块奖励的概率越大。算力表明了矿机的计算能力。

截止本书写作当天，比原链全网算力为126.77 Mhash/s。在比原链中，大约每2.5分钟可以产出412.5个BTM，一天总产量大约为237600枚BTM。也就是说，当天有126.77 Mhash/s的算力在争抢237600枚BTM。如果未来算力上涨，则挖矿难度增加，单位算力的产量将同比降低。

挖矿的每秒算力单位：

- 1 Khash/s=1000 hash/s
- 1 Mhash/s=1000 Khash/s
- 1 Ghash/s=1000 Mhash/s

· 1 Thash/s=1000 Ghash/s

· 1 Phash/s=1000 Thash/s

全网算力数据可以从比原社区浏览器获取（http://www.btmscan.com/）。

目前主流的共识算法有：工作量证明（Proof-of-Work，PoW）和权益证明（Proof-of-Stake，PoS）。

1.Pow（工作量证明）

大部分加密货币采用PoW方式挖矿：节点需要找到特定的区块头，使得hash函数的计算结果（也称为哈希）小于特定的值（Target）。由于hash函数的特性，不可能通过函数值来反向计算自变量，所以要得到满足上述条件的区块头，必须用枚举的方式尝试不同的区块头，直到找出符合要求的为止。这一过程需要进行很多运算，我们把矿机每秒能完成的hash函数的计算次数称为算力（单位是hash/s）。而对于一个已经公开的区块头，可以很容易地计算出它的哈希，并且验证其是否满足Target的要求。

PoW的主要目的是防止对区块链网络的攻击，对于加密货币来说，账本必须得到全网的承认，且不能被随意更改。PoW的复杂性保证了任何人都需要付出大量的运算来产生新的块，如果要篡改已有的区块，需要付出的算力要比网络上的其他节点总和都大，一般来说，这是不可行的，即使可行，其付出和获得的收益相比也是不划算的。

2.PoS（权益证明）

PoS是另一种挖矿方式，这种方式要求节点将一部分加密货币锁定，并根据数量和锁定的时长等因素来分配记账权。PoS一般不需要大量的计算，所以比PoW更加迅速和高效。

PoS的问题在于，如有人持有大量的该币种，只要放着不动，就会自动产生源源不断的利息，随着时间的推移，马太效应将使得某些大户越来越有钱，小户越来越穷。在这个模式下，并没有体现“劳动产生效益”的思想。

3.PoW基本公式

Target表示目标挖矿难度。二进制表示为256位，十六进制表示为64位。下面举两个难度的例子，以十六进制表示如图12-1所示。

以上两个难度中，难度1要比难度2的难度高。我们可以举掷硬币的例子来说明。假设有6枚硬币，掷出6枚都是正面（难度1）比掷出4枚为正面（难度2）的难度要高。把硬币扩展到64枚，掷出6枚或更多枚正面即64位数字前面的一串0，这串0称为前导零。两个难度的比较即可以通过比较前导0的个数来进行难度判断，前导零个数越多难度越高。

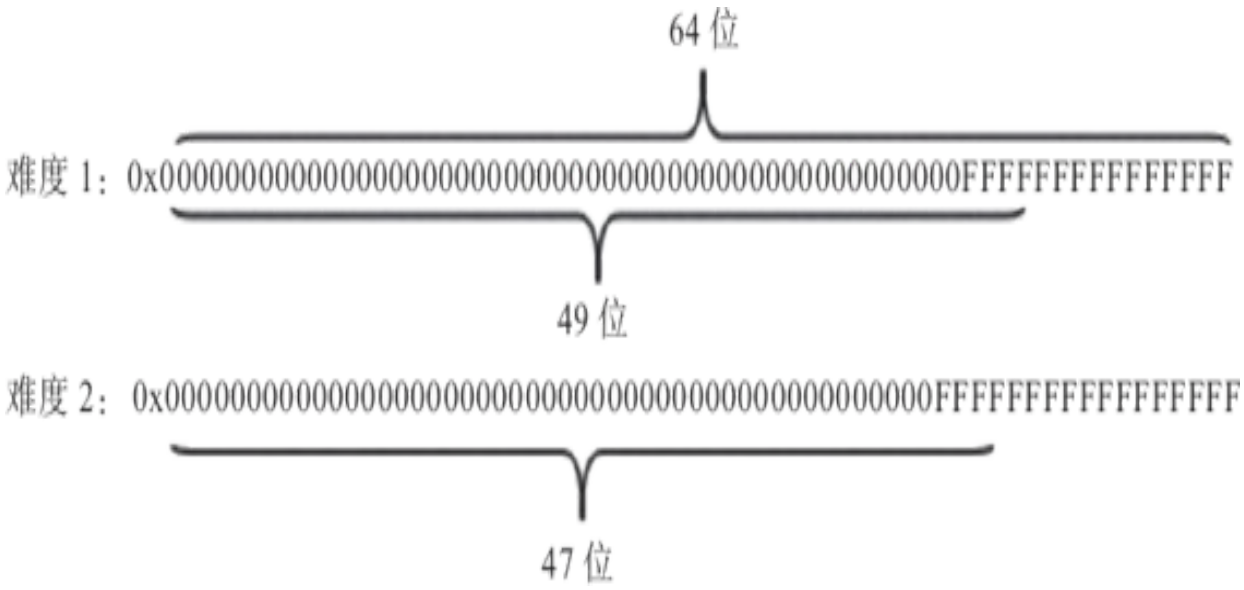


图12-1　挖矿难度

PoW挖矿的基本原理是找到特定的nonce，使得区块头（其中包含nonce）的哈希小于或等于Target。公式如下：

hash≤target

在比原链中PoW验证使用Tensority算法，基本公式如下：

powhash（header）≤target

比原链使用两种Hash算法：一种是用于PoW验证的Tensority算法，另一种是其他场景使用的SHA3算法。此处使用的powhash特指Tensority算法。

12.3 实现一个简易PoW共识算法

在本小节我们用GO语言实现一个简易的PoW挖矿算法，让读者详细了解PoW共识运行原理过程。

1.导入依赖包

将依赖包通过import语句导入进去，代码示例如下：

```
package main

import (
    "bytes"
    "crypto/sha256"
    "encoding/binary"
    "fmt"
    "math"
    "math/big"
    "time"
)
```

2.定义挖矿的目标难度值

定义难度值代码示例如下：

```
const targetBits = 24
```

targetBits为区块头中存储的挖矿难度。targetBits的赋值是24，表示计算出来的哈希值前24位必须是0，比如，二进制为“0000

0000 0000 0000 0000 0000 1001 1000”，若转换成十六进制（取四合一法）表示，则得到“0x00000098”，前导数6位是0。当挖矿节点计算出来的哈希满足“hash<=target”基本公式，则节点可以爆块。

目前不会实现一个动态调整目标的算法，所以将难度定义为一个全局的常量即可。难度动态调整的方法在12.6.2节中介绍。

3. 定义Block结构体

Block结构体代表组成区块链的每一个区块的数据模型，代码如下：

```
var (
    Blockchain []*Block
    maxNonce   = math.MaxInt64
)
```

Block结构体代码示例如下：

```
type Block struct {
    Timestamp     int64
    Data          []byte
    PrevBlockHash []byte
    Hash          []byte
    Nonce         int
    Height        int
}
```

Block结构体说明如下。

· Blockchain：模拟一个区块链表结构，每个元素为区块类型。

· maxNonce：nonce计数器的最大值。限制在math.MaxInt64内，目的是避免可能存在的nonce溢出。

· Timestamp：块生成时的时间戳。

· Data：区块中存储的数据。

· PrevBlockHash：通过SHA256算法生成的父哈希值。

· Hash通过SHA256算法生成的哈希值。

· Nonce：计数器。Nonce在挖矿软件中常被称为计数器，此处称为随机数比较合适。

· Height：当前区块的高度。

4. 定义ProofOfWork结构体

定义ProofOfWork结构体的示例代码如下：

```
type ProofOfWork struct {
    block  *Block
    target *big.Int
}

func NewProofOfWork(b *Block) *ProofOfWork {
    target := big.NewInt(1)
    target.Lsh(target, uint(256-targetBits))

    return &ProofOfWork{b, target}
}
```

ProofOfWork结构中存储了指向一个block块和一个target目标的指针。

target也就是规则，我们将节点计算的哈希与target目标进行比较：先把一个哈希转换成一个大整数，然后检测它是否小于或等于target目标。

实例化NewProofOfWork对象时，我们将big.Int初始化为1，然后将target左移256 - targetBits位。256是一个SHA-256哈希的位数，我们使用的是SHA-256哈希算法。target的十六进制形式为：

```
0x1000000000000000000000000000000000000000000000000000000000000000
```

它在内存上占据29个字节，即（256 - targetBits）/8。

5. 预备区块头数据进行哈希计算

将区块头的数据字段合并，然后计算区块哈希值。代码示例如下：

```
func (pow *ProofOfWork) prepareData(nonce int) []byte {
    data := bytes.Join(
        [][]byte{
            pow.block.PrevBlockHash,
            pow.block.Data,
            IntToHex(pow.block.Timestamp),
            IntToHex(int64(targetBits)),
            IntToHex(int64(nonce)),
        },
        []byte{},
    )

    return data
}

func IntToHex(num int64) []byte {
    buff := new(bytes.Buffer)
    err := binary.Write(buff, binary.BigEndian, num)
    if err != nil {
        panic(err)
    }

    return buff.Bytes()
}
```

prepareData中的nonce参数是计数器，用于每次hash计算时能得到不同的hash值。

IntToHex将int64类型转换为byte数组。

6. 实现PoW算法的核心

具体代码如下：

```
func (pow *ProofOfWork) Run() (int, []byte) {
    var hashInt big.Int
    var hash [32]byte
    nonce := 0

    fmt.Printf("Mining the block containing \"%s\"\n",
pow.block.Data)
    for nonce < maxNonce {
        data := pow.prepareData(nonce)

        hash = sha256.Sum256(data)
        fmt.Printf("\r%x", hash)
        hashInt.SetBytes(hash[:])

        if hashInt.Cmp(pow.target) == -1 {
            break
        } else {
            nonce++
        }
    }
    fmt.Print("\n\n")

    return nonce, hash[:]
}
```

Run函数是一个for循环，一直尝试计算区块头的哈希值（根据nonce计数器，每次得到的哈希值是不同的）。当计算出的哈希值小于target值，则表示节点挖矿成功。下面解读Run函数的详细过程。

首先对变量进行初始化：

· HashInt：哈希的整型表示。

· nonce：计数器，从0开始，小于maxNonce。

然后开始挖矿过程：

开启一个for无限循环。maxNonce对这个循环进行了限制，在这个循环中，需要做的事情包括——

1）准备区块头数据。

2）用SHA-256对数据进行哈希。

3）将哈希转换成一个big.Int大整数。

4）将big.Int大整数与target目标进行比较，如果target小于或等于big.Int则break，表示挖矿成功并返回该哈希值。

之后，将哈希值与目标值进行比较，比较公式为hash<=target。

假设我们的target为：

```
0000010000000000000000000000000000000000000000000000000000000000
```

假设第一个计算出来的哈希为：

```
f80867f6efd4484c23b0e7184e53fe4af6ab49b97f5293fcd50d5b2bfa73a4d0
```

当前哈希值大于目标值，其结果为false，则继续计算。

假设第二个计算出来的哈希为：

```
0000002f7c1fe31cb82acdc082cfec47620b7e4ab94f2bf9e096c436fc8cee06
```

当前哈希值大于目标值，其结果为true，则爆块。

7.对PoW进行验证

在P2P网络中，节点收到一个区块时需要对区块进行验证，这是重新计算的一个过程。代码示例如下：

```
func (pow *ProofOfWork) Validate() bool {
    var hashInt big.Int

    data := pow.prepareData(pow.block.Nonce)
    hash := sha256.Sum256(data)
    hashInt.SetBytes(hash[:])

    isValid := hashInt.Cmp(pow.target) == -1

    return isValid
}
```

---

8. 创建区块

具体代码示例如下：

```
func NewBlock(data []byte, prevBlockHash []byte, height int)
*Block {
    block := &Block{time.Now().Unix(), data, prevBlockHash,
[]byte{}, 0, height}
    pow := NewProofOfWork(block)
    nonce, hash := pow.Run()

    block.Hash = hash[:]
    block.Nonce = nonce

    return block
}

func main() {
    GenesisBlock := NewBlock([]byte("GenesisBlock"), []byte{},
0)
    SecondBlock := NewBlock([]byte("SecondBlock"),
GenesisBlock.Hash, GenesisBlock.Height+1)

    Blockchain = append(Blockchain, GenesisBlock, SecondBlock)
    for _, block := range Blockchain {
        fmt.Printf("PrevHash: %x\n", block.PrevBlockHash)
        fmt.Printf("Data: %s\n", block.Data)
        fmt.Printf("Hash: %x\n", block.Hash)
        fmt.Printf("\n")
    }
}
```

NewBlock函数执行PoW核心代码，直到挖出区块并返回，否则会一直计算哈希并与target进行比较。这个过程也就是挖矿过程。

我们在main函数中定义该程序去挖两个区块——GenesisBlock创世区块和SecondBlock第二个区块；然后进行遍历输出。

9. 执行结果

经过几十秒计算之后，可以看到输出如下区块信息。挖到的区块哈希值都小于target值。

```
Mining the block containing "GenesisBlock"
0bd205e7f166f34ab7799e5027d1a1e1947d0c13bca5db65b79d5f78e50a85fe

Mining the block containing "SecondBlock"
0e8794a0e49a6cd40b406a11d082ade738e0ff831cb430aab10a7073c67de7de

PrevHash:
Data: GenesisBlock
Hash:
0bd205e7f166f34ab7799e5027d1a1e1947d0c13bca5db65b79d5f78e50a85fe

PrevHash:
0bd205e7f166f34ab7799e5027d1a1e1947d0c13bca5db65b79d5f78e50a85fe
Data: SecondBlock
Hash:
0e8794a0e49a6cd40b406a11d082ade738e0ff831cb430aab10a7073c67de7de
```

12.4　比原链PoW共识算法

12.4.1　PoW hash值

比原链将Tensority算法作为PoW共识算法。本节将详细描述Tensority的原理及实现。此处我们先了解Tensority是如何得到PoW hash的。代码示例如下：

```
protocol/state/blockindex.go
// CalcNextSeed calculate the seed for next block
func (node *BlockNode) CalcNextSeed() *bc.Hash {
    if node.Height == 0 {
        return consensus.InitialSeed
    }
    if node.Height%consensus.SeedPerRetarget == 0 {
        return &node.Hash
    }
    return node.Seed
}
```

CalcNextSeed函数生成seed种子，seed种子用于构建矩阵。在比原链中，seed每256个块更新一次，更新时把某一个父区块的256位哈希值作为新的seed（比如第0～256使用第0块的hash值，第257～512使用第256块的hash值，以此类推）。最终得到一个32字节的hash值。

```
mining/tensority/go_algorithm/algorithm.go
func LegacyAlgorithm(bh, seed *bc.Hash) *bc.Hash {
    cache := calcSeedCache(seed.Bytes())
    data := mulMatrix(bh.Bytes(), cache)
    return hashMatrix(data)
}
```

Tensority算法首先对32字节的seed进行扩展（calcSeedCache）形成256个256×256方阵，然后每8字节为一组的hash值继续进行SHA3运算，每次得到32字节的数据（每个字节的可取值范围是0～255）。将这32个数字作为数组下标，获取前述256个方阵中的32个进行矩阵乘法。上述操作重复4次，将乘积相加。最后进行FNV计算，得到一个256位整数，这就是最终的PoW hash值。

可以看到，整个hash计算中，最复杂的步骤就是进行256个矩阵的乘法，这也是整个Tensority算法中最耗时的步骤，无论在矿池还是在矿机的实现中，对这一步的优化都非常重要。

12.4.2 难度动态调整

在比原链中，从设计角度来看，难度动态调整（Retarget）每2016块进行一次。难度动态调整的目的是保证出块速率均匀，使得出块平均时间为2.5分钟，即150秒/块。主网的难度动态调整出现在主网高度为2016的倍数时，全网难度发生变化，这个变化往往是难度的提高，原因在于一般主网的算力在不断的增加。主网难度调整逻辑可以参见如下代码：

```
consensus/difficulty/difficulty.go
func CalcNextRequiredDifficulty(lastBH, compareBH
*types.BlockHeader) uint64 {
    if (lastBH.Height)%consensus.BlocksPerRetarget != 0 ||
lastBH.Height == 0 {
        return lastBH.Bits
    }

    targetTimeSpan := int64(consensus.BlocksPerRetarget *
consensus.TargetSecondsPerBlock)
    actualTimeSpan := int64(lastBH.Timestamp -
compareBH.Timestamp)

    oldTarget := CompactToBig(lastBH.Bits)
    newTarget := new(big.Int).Mul(oldTarget,
big.NewInt(actualTimeSpan))
    newTarget.Div(newTarget, big.NewInt(targetTimeSpan))
    newTargetBits := BigToCompact(newTarget)

    return newTargetBits
}
```

以上代码的主要逻辑为难度的动态调整，新难度的产生可以拆解为如下3步。

1）计算2016个块的理论出块时间，代码如下：

```
targetTimeSpan := int64(consensus.BlocksPerRetarget *
consensus.TargetSecondsPerBlock)
```

2）计算前2016个块的实际出块时间，代码如下：

```
actualTimeSpan := int64(lastBH.Timestamp - compareBH.Timestamp)
```

3）按照时间偏差比例计算新的难度，代码如下：

```
oldTarget := CompactToBig(lastBH.Bits)
newTarget := new(big.Int).Mul(oldTarget,
big.NewInt(actualTimeSpan))
newTarget.Div(newTarget, big.NewInt(targetTimeSpan))
```

以上3行代码用公式表示即为：

新的难度=旧难度×实际出块时间/理论出块时间

根据主网不断增加的算力，实际出块时间比理论出块时间要快，其比值小于1，所以新的难度比旧的难度数值小，如果用十六进制表示，前导数的“零”个数越多，难度越高。

对于矿池来说，在保证比原链高度每增加2016即调整一次难度的基础上，为了使矿池提高任务提交次数乃至提高爆块率，还要根据矿池的实际情况尽量保证每分钟提交三次，或者根据多次任务的频次随时调整矿池难度。矿池在下发给矿机任务时进行难度调整，主要是降低难度，意味着减少前导零的个数，若用十进制数值表示则放松难度即增加数值。

12.4.3 Tensority算法

Tensority算法[1]是比原链基金会提出的新型PoW共识算法。与以比特币、以太坊的共识算法为代表的传统算法不同，Tensority将矩阵和张量运算引入到哈希过程中，以替代传统的大量无意义的哈希运算。这样，运行Tensority算法的ASIC矿机就能够在闲置时或在挖矿收益的调节下，用于人工智能硬件的加速服务。Tensority的创新使其有望在PoW算法与人工智能之间搭起一座桥梁，产生额外的社会效益。

本节将着重介绍Tensority的算法细节，并从密码学、信息论的角度证明Tensority具备与传统PoW算法等效的安全性。

在正式介绍Tensority之前，首先对PoW算法在区块链中的应用做一个简单的回顾。典型的区块链PoW过程可抽象为以下3步：

1）根据交易池构造待进行工作量证明的区块模板，得到挖矿的工作区块。固定挖矿工作区块中除Nonce值以外的所有字段，为Nonce字段赋初值（一般为0）。

2）对挖矿工作区块做哈希运算，得到区块的哈希值H。

3）检查哈希值H是否满足某种给定的目标要求，如果满足，则完成工作量证明。否则，Nonce值递增，返回上一步重新执行。

之所以在步骤2）中采用哈希运算，是为了保证以任意方式构造的区块在工作量证明的过程中，其哈希值H的概率分布都遵循均匀分布，这样才能杜绝矿工在数学上作弊，从而维持整个共识过程中的公平性。

在比特币中，采用的哈希运算是双SHA256计算，这是一种除挖矿外很难用于别处的数学运算。而Tensority中采用的哈希运算是一些相对通用的矩阵运算，这种运算在AI领域的机器学习中十分常见。因此，Tensority在两个不同领域中找到相通的计算需求，使算力通过Tensority这座“桥梁”，自由穿行于两片“大陆”。

1. 算法框架

Tensority算法包含5个主要步骤：缓存计算、矩阵构造、矩阵运算、工作生成和工作验证。Tensority算法框架如图12-2所示。

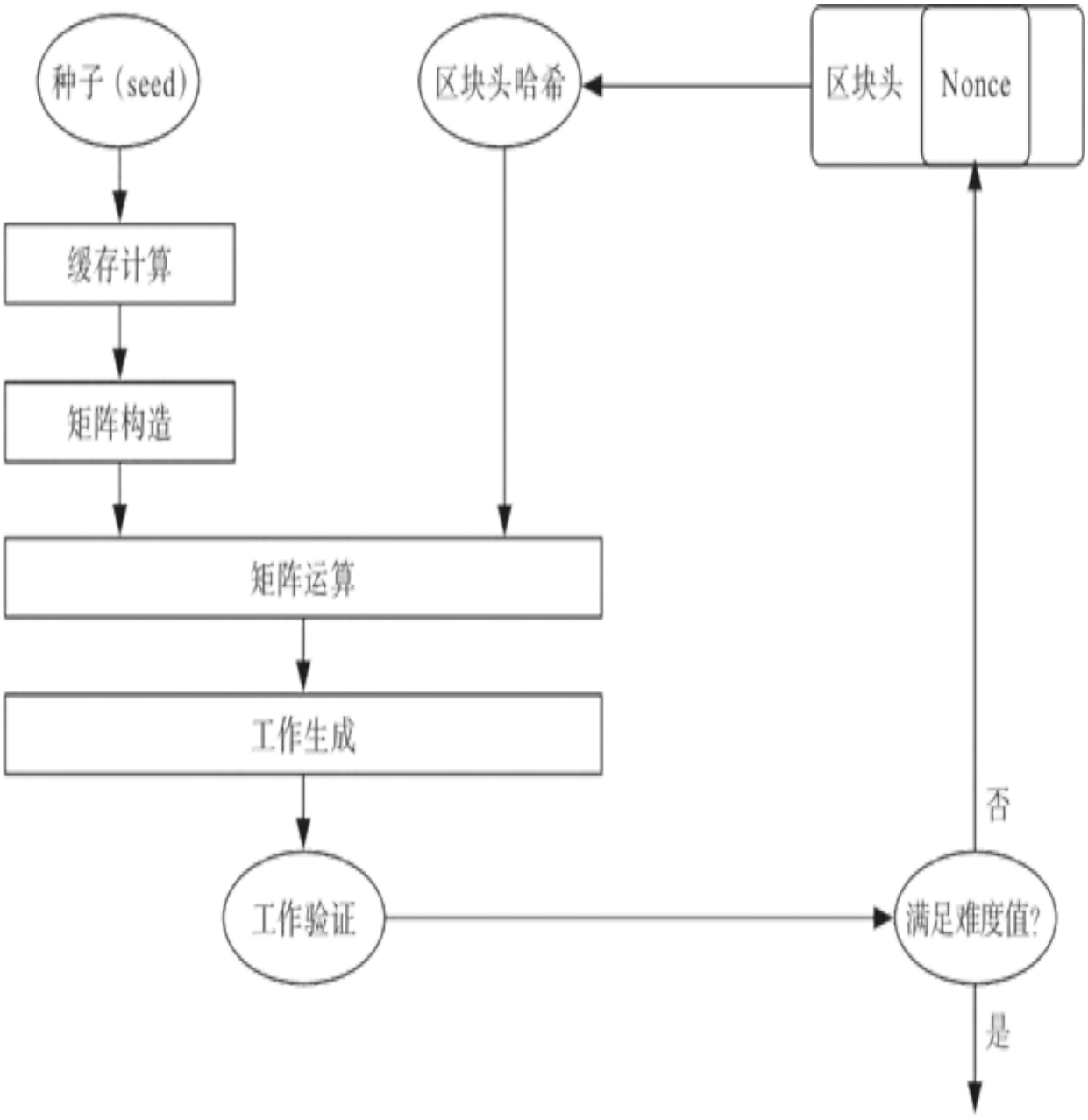


图12-2　Tensority算法框架

缓存计算、矩阵构造可以看作典型PoW过程的第一步，矩阵运算、工作生成可看作第二步，工作验证则可看作是第三步。通过与典型PoW过程的对照，就能够更容易地理解Tensority的算法框架。

2. 生成种子（seed）

种子用于构建矩阵，是生成cache的输入值。种子作为输入十分重要，它应当至少满足随机产生的需求。

在比原链中，种子每256个块更新一次，更新时把某一个父区块的256位哈希值作为新的种子（如本节开头所述）。代码示例如下：

```
protocol/state/blockindex.go
// CalcNextSeed calculate the seed for next block
func (node *BlockNode) CalcNextSeed() *bc.Hash {
    if node.Height == 0 {
        return consensus.InitialSeed
    }
    if node.Height%consensus.SeedPerRetarget == 0 {
        return &node.Hash
    }
    return node.Seed
}
consensus/general.go
SeedPerRetarget       = uint64(256)
```

3. 缓存计算

缓存（cache）是构建缓存矩阵的中介，由上一节中的种子（seed）生成。与出块速率相比，种子的更新速度较慢。因此，一次生成的缓存可以重复使用一段时间。构建缓存的主要步骤如下。

1）种子扩展（Seed Extention）。将seed0设置为种子，计算seed0的SHA3-256哈希值，从而得到seed1。类似地，我们可以通过计算seedi－1的SHA3-256哈希值来获得seedi。最后，我们将seed0，...，seedextround连起来，从而得到扩展种子。

2）扩展种子Scrypt加密（Scrypt Extseed）。递归调用Scrypt函数来获取缓存，即一个32×1024×128的uint32数组。Scrypt是一种KDF算法，主要用于密钥生成，旨在防止低成本的密码碰撞。值得一提的是，Scrypt自2011年起用于Litecoin。因此，它已被证明是一种可靠的种子扩展算法。

该算法的伪代码描述如下：

```
Algorithm 1: 根据种子生成缓存(calcSeedCache)
    Input: seed - a byte array of 32;
    Output: cache - an uint32 array of 32x1024x128;
1   Initialize extround = 3; scryptround = 128;
2   extseed = seed;
3   tmphash = seed;
4   for i = 1; i≤extround; i++ do
5       tmphash = SHA3-256(tmphash);
6       extseed = Append(extseed, tmphash);
7   end
8   cache = ø;
9   tmpv = ø;
10  for j = 1; j≤scryptround; j++ do
11      tmpv = Scrypt(extseed, tmpv);
12      cache = Append(cache, tmpv);
13  end
14  return cache;
```

于是，从一个256位的随机种子构造得到32×1024×128的缓存数组。代码示例如下：

```
mining/tensority/go_algorithm/seed.go
func calcSeedCache(seed []byte) (cache []uint32) {
    extSeed := extendBytes(seed, 3)
    v := make([]uint32, 32*1024)

    // Swap the byte order on big endian systems
    if !isLittleEndian() {
        swap(extSeed)
    }

    for i := 0; i < 128; i++ {
        scrypt.Smix(extSeed, v)
        cache = append(cache, v...)
    }

    return cache
}

func extendBytes(seed []byte, round int) []byte {
    extSeed := make([]byte, len(seed)*(round+1))
    copy(extSeed, seed)

    for i := 0; i < round; i++ {
```

```
        var h [32]byte
        sha3pool.Sum256(h[:], extSeed[i*32:(i+1)*32])
        copy(extSeed[(i+1)*32:(i+2)*32], h[:])
    }

    return extSeed
}
```

---

4. 矩阵构造

构造矩阵为下一个环节的矩阵运算做好准备。主要步骤如下。

1）缓存重构（Cache Recomposition）。设计缓存重构的目的在于提高ASCI矿机的效率，例如更快的memory accession（存储访问/缓存器加入）。考虑到矿工的数据对齐和内存访问，在这里设计了以下对缓存的重构。

首先，将缓存（cache）分为128组。每组包括32×1024个元素。在每组中，将32个元素作为一个单元进行聚合。于是获得了32×1024×128的uint32矩阵tmpmatrix。Recomposedmatrix的大小也是32×1024×128。矩阵中，第2维具有奇数索引的tmpmatrix元素，与recomposedmatrix中第2维索引从1～1024/2的元素一一对应相等。类似地，tmpmatrix中第2维具有偶数索引的元素与recomposedmatrix中第2维索引从1024/2+1到1024的元素一一对应相等。

2）缓存矩阵构造（Cache Matrix Construction）。扩展矩阵recomposedmatrix并将其设置为256×256×256的int8数组。然后我们通过类型转换获得256×256×256的float64数组。最后，我们获得了256×256×256的float64矩阵cachematrix。

该算法的伪代码描述如下：

```
Algorithm 2: 构造缓存矩阵(constructCacheMatrix)
    Input: cache - an uint32 array of 32x1024x128;
    Output: cachematrix - an float64 matrix of 256x256x256
1   Initialize dim1 = 32; dim2 = 1024; dim3 = 128; dim = 256;
2   tmpmatrix = Matrix(cache, dim1, dim2, dim3);
3   recomposedmatrix = NewMatrix(dim1, dim2, dim3);
4   cachematrix = NewMatrix(dim, dim, dim);
5   recomposedmatrix[:][1 : dim2/2][:] = tmpmatrix[:][all odd
```

```
index][:] ;
6   recomposedmatrix[:][dim2/2+1 : dim2][:] = tmpmatrix[:][all
even index][:];
7   cachematrix = Float64(Matrix(Int8Array(recomposedmatrix),
dim, dim, dim));
8   return cachematrix;
```

代码示例如下：

```
mining/tensority/go_algorithm/matrix.go
const (
    matSize = 1 << 8 // Size of matrix
    matNum  = 1 << 8 // Number of matrix
)

func mulMatrix(headerhash []byte, cache []uint32) []uint8 {
    ui32data := make([]uint32, matNum*matSize*matSize/4)
    for i := 0; i < 128; i++ {
        start := i * 1024 * 32
        for j := 0; j < 512; j++ {
            copy(ui32data[start+j*32:start+j*32+32],
cache[start+j*64:start+j*64+32])

copy(ui32data[start+512*32+j*32:start+512*32+j*32+32],
cache[start+j*64+32:start+j*64+64])
        }
    }

    // Convert our destination slice to a int8 buffer
    header := *(*reflect.SliceHeader)(unsafe.Pointer(&ui32data))
    header.Len *= 4
    header.Cap *= 4
    i8data := *(*[]int8)(unsafe.Pointer(&header))

    f64data := make([]float64, matNum*matSize*matSize)
    for i := 0; i < matNum*matSize*matSize; i++ {
        f64data[i] = float64(i8data[i])
    }
    // ...
}
```

---

5. 矩阵运算

矩阵运算的速度主要取决于矿工的计算能力，这一步也是Tensority算法中最为主要的工作量消耗部分。

在上一步“矩阵构造”中，有一项操作是不太常见的，即将int8数组转换为float64数组。之所以引入这样的类型转换操作，并且要求ASIC电路支持浮点数运算，目的是支撑AI算法，因为AI算法主要在浮点环境下运行。于是，矩阵运算时也相应地采用float64矩阵乘法代替整数乘法。

矩阵运算时将区块头哈希headerhash作为对cachematrix进行切片（slice）的索引，cachematrix是256×256×256的float64矩阵。在迭代计算多次切片的矩阵乘法后，最终得到工作矩阵workmatrix。注意，在256×256的矩阵之间总共有256轮矩阵乘法。主要步骤如下：

1）生成矩阵切片的索引（Generate Index of Matrix Slices）。

首先将区块头哈希分为4组。然后对每组数据做SHA3-256操作以获得32字节的相应序列。序列中的每个字节都被转换为整数，作为矩阵切片的索引。显然，在此过程中会生成4×32个矩阵切片。

2）矩阵计算（Matrix Caculation）。

根据索引获得相应的256×256的cachematrix的矩阵mb。矩阵mc是ma和mb^T相乘的结果。注意，ma是在第一轮中初始化的单位矩阵。然后我们将mc中的元素转换为int32。

这里定义一个名为Compress32to8的操作。它通过公式$(b_3+b_4) \bmod 2^8$将数据b=（b_1，b_2，b_3，b_4）（big endian）的数据类型从int32转换为uint8。引入Compress32to8以确保乘法结果具有更好的随机性。

之后，我们将mc的元素设置为相应的Compress32to8的结果。然后我们将mc的元素转换为float64并将结果分配给ma，直到序列最终耗尽。之前的步骤应当迭代2次。

最后，我们利用ms来更新hashmatrix。我们将获得ma和hashmatrix的Integer32的和。将更新后的hashmatrix中的每一个元素

用取低8位的方式，转换为int8，再将该元素转换为float64。

该算法的伪代码描述如下：

```
Algorithm 3: constructHashMatrix
    Input: cachematrix - a float64 array of 256x256x256;
headerhash - a byte array of 32
    Output: hashmatrix - an uint8 matrix of 256x256
1   Initialize drawround = 4; mulround = 2; dim = 256;
2   hashmatrix = Matrix(dim, dim);
3   drawmatrix = Matrix(headerhash, drawround,
sizeof(headerhash)/drawround);
4   for i = 1; i ≤ drawround; i++ do
5       ma = I;
6       mc = Matrix(dim, dim);
7       sequence = SHA3-256(drawmatrix[i]);
8       for j = 1; j ≤ mulround; j++ do
9           for k = 1; k ≤ sizeof(sequence); k++ do
10              index = Uint8(sequence[k]) + 1;
11              mb = srcmatrix[index][:][:];
12              mc = ma x mbT ;
13              for element ∈ mc do
14                  element =
Float64(Compress32to8(Int32(element)));
15              end
16              ma = mc
17          end
18      end
19      for row = 1; row ≤ dim; row++ do
20          for col = 1; col ≤ dim; col++ do
21              i32vhashmatrix = Int32(hashmatrix[row][col]);
22              i32vma := Int32(ma[row][col]);
23              i8v = Int8(i32vhashmatrix + i32vma);
24              hashmatrix[row][col] = Float64(i8v);
25          end
26      end
27  end
28  return hashmatrix;
```

代码示例如下：

```
mining/tensority/go_algorithm/matrix.go
func mulMatrix(headerhash []byte, cache []uint32) []uint8 {
```

```
    ......

    // 此处用 f64data = [matNum*matSize*matSize]float64 表示上一步
    // 得到的 cachematrix

    dataIdentity := make([]float64, matSize*matSize)
    for i := 0; i < 256; i++ {
        dataIdentity[i*257] = float64(1)
    }

    var tmp [matSize][matSize]float64
    var maArr [4][matSize][matSize]float64

    runtime.GOMAXPROCS(4)
    var wg sync.WaitGroup
    wg.Add(4)

    for k := 0; k < 4; k++ {
        go func(i int) {
            defer wg.Done()

            ma := mat.NewDense(matSize, matSize, dataIdentity)
            mc := mat.NewDense(matSize, matSize, make([]float64,
matSize*matSize))

            var sequence [32]byte
            sha3pool.Sum256(sequence[:], headerhash[i*8:
(i+1)*8])

            for j := 0; j < 2; j++ {
                for k := 0; k < 32; k++ {
                    index := int(sequence[k])
                    mb := mat.NewDense(matSize, matSize,
f64data[index*matSize*matSize:(index+1)*matSize*matSize])
                    mc.Mul(ma, mb.T())

                    for row := 0; row < matSize; row++ {
                        for col := 0; col < matSize; col++ {
                            i32v := int32(mc.At(row, col))
                            i8v := int8((i32v & 0xff) +
                                ((i32v >> 8) & 0xff))
                            mc.Set(row, col, float64(i8v))
                        }
                    }
                    ma = mc
                }
            }
```

```
                for row := 0; row < matSize; row++ {
                    for col := 0; col < matSize; col++ {
                        maArr[i][row][col] = ma.At(row, col)
                    }
                }
            }(k)
        }
        wg.Wait()

        for i := 0; i < 4; i++ {
            for row := 0; row < matSize; row++ {
                for col := 0; col < matSize; col++ {
                    i32vtmp := int32(tmp[row][col])
                    i32vma := int32(maArr[i][row][col])
                    i8v := int8(int32(i32vtmp+i32vma) & 0xff)
                    tmp[row][col] = float64(i8v)
                }
            }
        }

        result := make([]uint8, 0)
        for i := 0; i < matSize; i++ {
            for j := 0; j < matSize; j++ {
                result = append(result, uint8(tmp[i][j]))
            }
        }
        return result
}
```

---

6. 工作生成

在工作生成算法中，把hashmatrix作为输入，生成表示工作的32字节哈希值。由于这里采用的工作生成哈希算法本身具有最大随机性，所以无法从数学上找到优势更大的计算点，而只能尽量提高计算效率，更快地完成工作生成。

Tensority利用FNV实现更快的哈希到哈希矩阵运算，而不是用SH2或SH3来实现，因为FNV是非加密哈希算法。FNV已在以太坊的Ethash中运用了一段时间，其可靠性已得到证实。另外，我们选择0x01000193作为参数FNV_{prime}。最后，我们对FNV的结果做SHA3-256，以确保稳固的随机性。

1）调整矩阵hashmatrix的大小（Resize Matrix Hashmatrix）。

hashmatrix是256×256的uint8矩阵。对于每一行中的元素，对其下标按64取余，把余数相同的数作为一组。将每组的4个数整合为类型为uint32的数。这样，我们就得到了一个名为mat32的256×64的uint32矩阵。

2）Binary Forwarded FNV。

Binary Forwarded FNV实际上是一种哈希矩阵方法。首先，我们将n设置为mat32的第一维长度。从第1行到第n行，把行数按n/2取余，相同的两个同列元素送入FNV函数，将结果赋给行下标较小的元素。然后将n减半，重新执行上面的操作，直至n等于1。最后，取出mat32的第一行，将其转为byte数组，对数组做SHA3-256即得到工作成果（work）。

该算法的伪代码描述如下：

```
Algorithm 4: hashMatrix
    Input: mat32 - an uint32 matrix of 256x64;
    Output: hash - an byte array of 32;
1   Initialize dim1 = 256; dim2 = 64;
2   for k = dim1/2; k != 1; k = k/2 do
3       for i = 1; i ≤ k; i + + do
4           for j = 1; j ≤ dim2; j + + do
5               mat32[i][j] = FNV(mat32[i][j], mat32[i+k][j])
6           end
7       end
8   end
9   hash = SHA3-256(ToByteArray(mat32[0][:]));
10 return hash;
```

代码示例如下：

```
mining/tensority/go_algorithm/matrix.go
const (
    matSize = 1 << 8 // Size of matrix
    matNum  = 1 << 8 // Number of matrix
)
```

```
// hashMatrix hash result of mulMatrix
func hashMatrix(result []uint8) *bc.Hash {
    var mat8 [matSize][matSize]uint8
    for i := 0; i < matSize; i++ {
        for j := 0; j < matSize; j++ {
            mat8[i][j] = result[i*matSize+j]
        }
    }

    var mat32 [matSize][matSize / 4]uint32

    for i := 0; i < matSize; i++ {
        for j := 0; j < matSize/4; j++ {
            mat32[i][j] = ((uint32(mat8[i][j+192])) << 24) |
                ((uint32(mat8[i][j+128])) << 16) |
                ((uint32(mat8[i][j+64])) << 8) |
                ((uint32(mat8[i][j])) << 0)
        }
    }

    for k := matSize; k > 1; k = k / 2 {
        for j := 0; j < k/2; j++ {
            for i := 0; i < matSize/4; i++ {
                mat32[j][i] = fnv(mat32[j][i], mat32[j+k/2][i])
            }
        }
    }

    ui32data := make([]uint32, 0)
    for i := 0; i < matSize/4; i++ {
        ui32data = append(ui32data, mat32[0][i])
    }

    // Convert our destination slice to a byte buffer
    header := *(*reflect.SliceHeader)(unsafe.Pointer(&ui32data))
    header.Len *= 4
    header.Cap *= 4
    dataBytes := *(*[]byte)(unsafe.Pointer(&header))

    var h [32]byte
    sha3pool.Sum256(h[:], dataBytes)
    bcHash := bc.NewHash(h)
    return &bcHash
}

func fnv(a, b uint32) uint32 {
    return a*0x01000193 ^ b
}
```

7. 工作验证

最后一步，将工作成果（哈希）的值与块难度进行比较。如果工作成果的值较低，则将其视为通过验证的工作成果，并且矿工将会广播该块。对于一个诚实的矿工，如果在自己计算得到通过工作量证明的块之前，从其他节点接收到了已经过验证的块，则不会再继续挖当前块，也不会广播无效的块。

若本次的工作成果（哈希）未通过工作量验证，矿工将更换新的Nonce值，在接收到经过验证的块之前回到矩阵运算，继续运行Tensority。关于工作量证明的检测与Nonce值的重设，代码如下：

```
mining/cpuminer/cpuminer.go
func (m *CPUMiner) solveBlock(block *types.Block, ticker
*time.Ticker, quit chan struct{}) bool {
    ......

    for i := uint64(0); i <= maxNonce; i++ {
        ......

        header.Nonce = i
        headerHash := header.Hash()
        if difficulty.CheckProofOfWork(&headerHash, seed,
header.Bits) {
            return true
        }
    }
    return false
}

consensus/difficulty/difficulty.go
func CheckProofOfWork(hash, seed *bc.Hash, bits uint64) bool {
    compareHash := tensority.AIHash.Hash(hash, seed)
    return HashToBig(compareHash).Cmp(CompactToBig(bits)) <= 0
}
```

[1] Tensority算法的论文可参考https://bytom.io/wp-content/themes/freddo/book/tensority-v.1.2.pdf。

12.5 本章小结

本章我们向读者介绍了与挖矿相关的术语、PoW共识以及比原链的Tensority算法，并且通过算法步骤介绍以及严格的数学证明详细论述了Tensority的数学基础。

通过本章的学习，读者可以了解到Tensority算法的方方面面，并能学会如何通过GO语言实现一个简易的PoW共识算法。

第13章

矿池及挖矿流程

13.1 概述

在整个挖矿生态链条中，矿池是一个不可或缺的角色。矿池作为矿机与网络节点之间的桥梁，具备承上启下的作用。设计一款成熟的矿池需要考虑方方面面的需求，任何一点风险都有可能导致矿池与矿工的直接损失。

本章介绍矿池的相关知识，主要内容包括：

- 矿池概念、架构分析。
- 挖矿流程、矿池的收益分配方式。
- 交易打包至区块的流程。
- 矿池的优化建议。

13.2 与矿池相关的基本概念

1.矿池

如果全网的运算水准在不断地呈指数级上涨，单个矿机设备或少量的算力都无法在网络上获取区块奖励。在全网算力提升到了一定程度后，获取奖励的概率越来越低。此时便产生了矿池（Mining Pool），矿池可以将少量算力合并起来。

在矿池机制中，不论个人矿工能提供的运算力有多少，只要加入矿池参与挖矿过程，无论是否成功挖出有效块，最终都可按照对矿池的贡献来获得少量区块奖励，获得的区块奖励也由多人依照贡献度分享。

目前全球算力排名靠前的矿池有AntPool、BTC.com、BTC.TOP、ViaBTC、F2Pool等。

2.矿机

挖矿实际是矿工之间比拼算力，而挖矿的设备称为矿机。人们开发出专门用来挖矿的芯片，芯片是矿机最核心的零件。每种币的算法不同，所需要的矿机也各不相同。目前也有很多基于币的协议开发的矿池软件，可以运行在CPU、GPU设备上作为矿机设备使用。

3.矿场

矿场一般是具有固定场所的机房，能够为矿机提供电力资源以及托管维护服务，矿池主通常是有资源（主要是电力）优势的个人或团体，主要通过低成本电力赚取散户矿机的电费差额或托管服务费。矿场一般自己也有矿机参与挖矿。

4. 矿池难度

第12章中提到了target挖矿难度，target是被区块链网络节点认可的难度。而矿池下发给矿机的难度称为矿池难度。矿池难度往往比target难度要小，一般会动态调整，以保证矿机提交Share（工分）频率的稳定性。不同矿池调整矿池难度的机制和方法也不一样。

5. Share（工分）

矿机根据矿池下发的任务和难度，计算出符合要求的结果并提交，这一结果俗称Share，也可形象地称为“工分”。

6. 爆块

某一矿机提交的Share符合全网难度要求，并通过验证获得挖矿奖励，即为爆块，又称出块。

13.3 矿池总架构

下面从整个挖矿链条的角度来说明矿池、挖矿与矿机三者之间的联系。从上面的内容可以了解到，挖矿的过程是在做一道穷举数字的数学题，每猜一个数字都要验证一遍是否符合答案要求，直到自己找到正确答案或者得知别人已经解出问题，才会停止这个循环。获取新的问题和提交答案都需要自己是区块网络上的一个节点才行。

我们可以把计算与提交答案分为两部分：一部分只负责计算（即矿机）；一部分只负责提交答案和分配任务与收益（即矿池）。

以比原链为例，目前nonce计算结果的搜索空间为64位，以目前的芯片计算速度很难在短时间找到符合条件的计算结果，即便芯片速度不断提升，问题的难度也会进行相应的调整，使寻找计算结果的速度约为一个固定的值。这样一个矿机想在短时间内遍历结果空间基本是不可能的，所以需要抱团取暖。多个矿机分头计算，把搜索空间分段，每台矿机搜索其中的一段，这样就会大大提高寻找答案的效率，根据不同矿机在搜索结果的过程中付出的算力，按照一定的规则进行最终结果的分配，有效保证了所有参与计算的矿机都能得到收益。

矿池本质上就是一个中间代理，负责管理众多矿机，根据特定规则给矿机分配任务，协调不同矿机进行计算和提交结果验证，并根据不同矿机付出的算力和最终收益进行利益分配。图13-1展示了一个矿池的上下游关系。

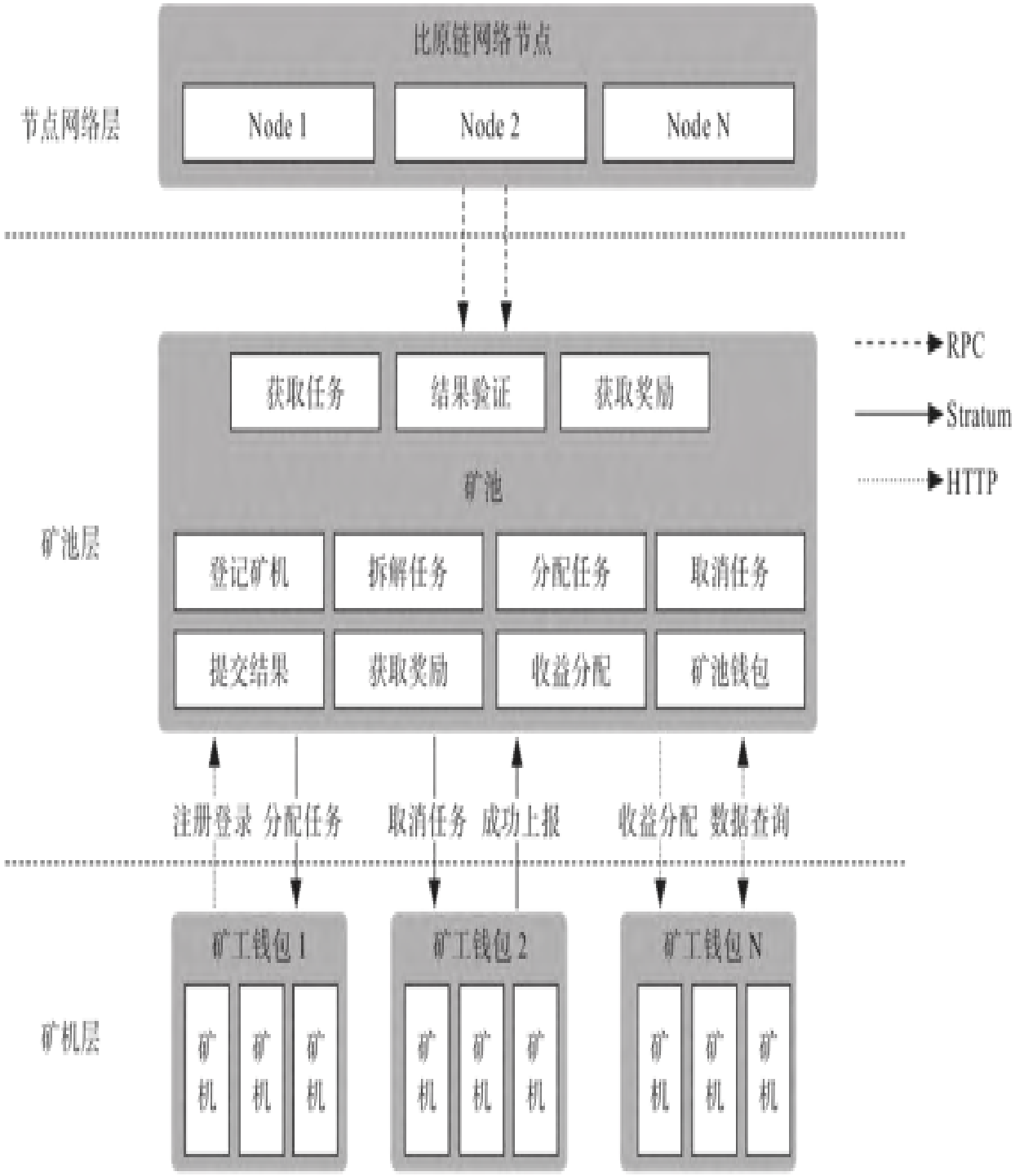


图13-1　矿池架构图

1. 节点网络层

节点网络层表示整个比原链的节点网络，从图13-1中可以看出，Node节点作为比原链节点网络的一部分，与网络中的其他节点进行交互通信（交易确认、块确认等）。

2. 矿池层

矿池层表示矿池软件在上游系统（节点网络层）和下游系统（矿机层）的代理商功能。对于上游系统，矿池软件从区块链网络中获取最新待计算的区块信息，根据自身的任务分配机制将待计算任务进行拆解并分配给下游系统，最终统计工作量并分配相应的收益。

3. 矿机层

矿机层包含形形色色的矿机，包括专业芯片矿机、CPU矿机、GPU矿机、个人电脑等。矿机通过Stratum协议与矿池软件进行交互，比如注册、获取新任务、提交Share等。

13.4　挖矿流程解析（矿池视角）

1. 获取最新待计算区块信息

以在本地调用为例，比原链节点开放API方法/get-work-json：

```
$ curl http://127.0.0.1:9888/get-work-json
```

返回结果如下：


```
{
  "status": "success",
  "data": {
    "block_header": {
      "version": 1,
      "height": 88855,
      "previous_block_hash":
"473bb6852560216d4308734bce56671d8976453c72897a289a1c2eb1ebbfec
12",
      "timestamp": 1536384485,
      "nonce": 0,
      "bits": 2017612633066440200,
      "block_commitment": {
        "transaction_merkle_root":
"6a580e16cff364a4c6f955f88db62efb6a268cdb52df92fcc2fa3955f53127
11",
        "transaction_status_hash":
"6978a65b4ee5b6f4914fe5c05000459a803ecf59132604e5d334d64249c5e5
0a"
      }
    },
    "seed":
```

```
"5318961688fbd8e56f100d827e80721b06697ce25b584fe32680c92804169e
de"
  }
}
```

可以看出，有效的返回信息分为两部分：block_header和seed，针对各字段说明如下。

· Version：版本。

· Height：最新区块高度。

· previous_block_hash：父区块哈希。

· Timestamp：时间戳。

· Nonce：计数器从0开始。

· Bits：当前主网挖矿难度。

· block_commitment：包含了默克尔根的值与状态码。

· Seed：作为tensority的输入参数，每256个区块改变一次。

矿池获取的最新待计算区块信息，建议存在本地内存中。

2. 任务拆解与分配

矿池软件根据最新待计算的块信息进行任务拆解。任务拆解有多种方式，此处以一种最简单的拆解方式为例，假设当前连接到矿池的矿机数量为1024台，随机字数nonce搜索空间为2^256，计算每台矿机平均搜索空间长度如下：

$L=2^{256}/1024=2^{246}$

任务明细如下：

矿机编号	计算搜索空间
M01	$0 \sim 2^{246}-1$
M02	$2^{246} \sim 2^{247}-1$
M03	$2^{247} \sim 2^{\wedge 248}-1$
……	……
M1024	$2^{248} \sim 2^{\wedge 256}-1$

3. 任务验证与提交

针对矿机提交的结果进行验证，首先验证结果是否符合矿池难度，然后计算是否符合标准难度。

验证后的结果通过比原链API接口/submit-work-json，提交到比原链节点，节点验证通过后，即爆块成功，获得爆块奖励。

矿池将爆块的任务ID、矿机ID、时间戳等信息记录在本地数据库中，并重新从网络节点获取最新的待计算区块信息，进行下一次任务拆解与分配。

4. 工作量统计与收益分配

矿机的每一次运算都是要消耗一定的成本，矿池软件作为矿机与全网节点的中间桥梁，必须记录矿机工作的明细以便作为将来收益分配的依据。

13.5 挖矿流程解析（矿机视角）

每一种矿机基本都内置了矿机商开发的挖矿软件，矿机连接好之后，一般只需要填入要挖矿的矿池地址以及自己的钱包地址，然后启动挖矿即可。填入矿池地址的目的是确定矿机在哪家矿池挖矿，填入钱包地址目的是明确矿池进行挖矿收益分配的收钱地址。

矿机与矿池的交互是通过Stratum协议进行的，标准的Stratum协议（参见https://en.bitcoin.it/wiki/Stratum_mining_protocol）定义了各种方法，不同的矿池与矿机软件商实现的方法不尽相同，但流程大同小异。具体到比原链，参与比原链挖矿并不需要太多交互方法。讨论矿机参与挖矿的流程，此处以B3矿机与矿池的交互过程为参考（https://github.com/HAOYUatHZ/B3-Mimic/blob/master/docs/STRATUM-BTM.md）。

1.登录并获取任务

矿机在通电并填入矿池地址后，首先，矿机软件调用矿池的Login方法，注册成为矿池中挖矿的一员。注册的过程是，矿机提交自己的钱包地址，矿池记录钱包地址并返回挖矿任务。

请求报文如下：

```
{
    "method": "login",
    "params": {
        "login": "antminer_1",
        "pass": "123",
        "agent": "bmminer/2.0.0"
    },
```

```
    "id": 1
}
```

本方法为Login方法，结构为JSON，其中主要参数的含义如下。

· login：登录用户名，多为矿工的钱包地址，必填。

· pass：登录密码，可以不设置，非必填。

· agent：矿机版本型号，非必填。

返回报文如下：

```
{
    "id":1,
    "jsonrpc":"2.0",
    "result":{
        "id":"antminer_1",
        "job":{
            "version":"0100000000000000",
            "height":"2315000000000000",

"previous_block_hash":"ecaeb84f7787aca9ed08199169fe8dc5a01f3ba5
0c90c6844452f916d645d911",
            "timestamp":"239dc75a00000000",

"transactions_merkle_root":"178f3dfaf916a5f8167100602254df50f68
21c243a1f6263efedde798e9271a2",

"transaction_status_hash":"c9c377e5192668bc0a367e4a4764f11e7c72
5ecced1d7b6a492974fab1b6d5bc",
            "nonce":"0c04000001000000",
            "bits":"4690890000000021",
            "job_id":"16368",

"seed":"a2a62d7715ee2234e1d73c22d26a1707fb7bc0f4ee0c01d43a4c97b
0328379c5",
            "target":"c5a70000"
        },
        "status":"OK"
    },
    "error":null
}
```

返回报文中，version、height、previous_block_hash、timestamp、transactions_merkle_root、transaction_status_hash、bits为区块头标准字段，含义不再重复解释。

关键参数说明如下。

· job_id：矿池分配给矿机的任务id，用来标识任务，当任务提交时需要原样返给矿池。

· nonce：矿池分配给矿机的随机数区间，此处的取值往往是区间的起始值，对区间结束值一般不设定。

· target：任务目标难度，即矿池给定的任务难度。

· seed：用于比原链进行Tensorty计算。

2. 计算并提交任务

矿机通过矿机软件与矿池进行交互，通过获取任务信息得到与计算相关的参数，根据参数和内置的算法不断计算，当找到满足条件的结果时进行任务提交。

请求报文如下：

```
{
    "method": "submit",
    "params": {
        "id": "antminer_1",
        "job_id": "4171",
        "nonce": "bc000d41"
    },
    "id":3
}
```

参数说明如下。

· job_id：矿池分配给矿机的任务id，用来标识任务，当任务提交时需要原样返给矿池，必填。

· nonce：矿机通过计算找到满足条件的结果，即搜索空间内满足条件的那个随机数，必填。

返回报文（成功时）如下：

```
{
    "id":3,
    "jsonrpc":"2.0",
    "result":{
        "status":"OK"
    },
    "error":null
}
```

返回报文（失败时）如下：

```
{
    "id":41,
    "jsonrpc":"2.0",
    "result":null,
    "error":{
        "code":-1,
        "message":"Low difficulty share"
    }
}
{
    "id":12,
    "jsonrpc":"2.0",
    "result":null,
    "error":{
        "code":-1,
        "message":"Block expired"
    }
}
```

参数很简单，不同的矿池可以定义不同的返回信息。

13.6 拒绝数与拒绝率

在挖矿过程中，矿机不断地进行任务计算，此时矿池定时或不定时下发新的任务，当矿池中的任务已经更新，矿机在新任务从矿池下发到矿机接受这个时间段内有旧任务提交，这些提交的旧任务会被矿池拒绝并丢弃，拒绝并丢弃的数量即拒绝数，拒绝数占所有提交任务的比率即拒绝率。公式为：

拒绝率=被拒绝任务数/提交任务总数×100%

如果要提高矿池的性能，必须尽可能地降低拒绝数和拒绝率。造成拒绝数的原因主要有两方面：网络延时和矿机软件。因此，降低拒绝数主要包括优化网络链路、降低网络延时等手段。

13.7　矿池的收益分配模式

目前主流的的矿池分配方式主要有PPS、PPLNS、PROP三种。

1.PPS（Pay-Per-Share）模式

此种模式下矿池会根据连接到矿池的矿机数量，统计整个矿池的总算力，估算矿池每天的总收益，按照矿机算力在总算力中的占比来分配收益，而不管实际是否爆块。计算公式为：

矿机每天收益=（矿机算力/矿池当天总算力）×矿池每天理论收益

这种分配模式存在三种情况：

· 全天无爆块，所有矿工收益按估算值支付，由矿池垫付，矿池当天亏损。

· 按理论值爆块，所有矿工收益按估算值支付，由矿池垫付，矿池当天收支平衡。

· 按超出理论值爆块，所有矿工收益按估算值支付，由矿池垫付，矿池当天盈利。

对比以上三种情况，只有极其幸运的情况下，实际爆块超出理论爆块数，矿池才有可能盈利，所以矿池为了自己能够持续运营下去必须收取一定比例的手续费。

2.PPLNS（Pay Per Last N Shares）模式

此种模式下矿池会统计每台矿机提交的有效Share（即达到爆块标准的Share）的数量，按照时间段内矿机有效Share占矿池总有效Share的比例进行分配。计算公式为：

矿机当天收益=（前段时间该矿机有效Share/前段时间矿池总有效Share）×矿池前段时间总收益

这种分配模式存在两种情况：

· 前段时间当前矿机提交有效Share数为0，则矿机前段时间收益为0，矿池收支平衡。

· 若前段时间当前矿机提交有效Share数为x（x！=0），则矿机前段时间的收益（x/X）×Y，其中X>=x，X为前段时间矿池总有效Share；Y为矿池前段时间总收益。矿池收支平衡。

对比以上两种情况，如果不考虑矿池的运营成本，矿池是不亏损的，所以这种收益分配模式下，矿池会收取低比例的手续费，甚至不收费。

3. PROP模式

此种模式与PPLNS模式的唯一区别是，结算时间点不同，矿工不能实时看到自己的当前实际收益值。PROP模式的收益分配时间点在区块确认后，而PPLNS模式的收益分配时间点在区块产生时，无需等待确认。

13.8　交易打包至区块

本节主要介绍矿工挖出块后将“交易打包到区块”的过程。矿工的根本收益源自块打包过程中的两部分——CoinBase交易奖励和常规交易手续费。交易打包过程如图13-2所示。

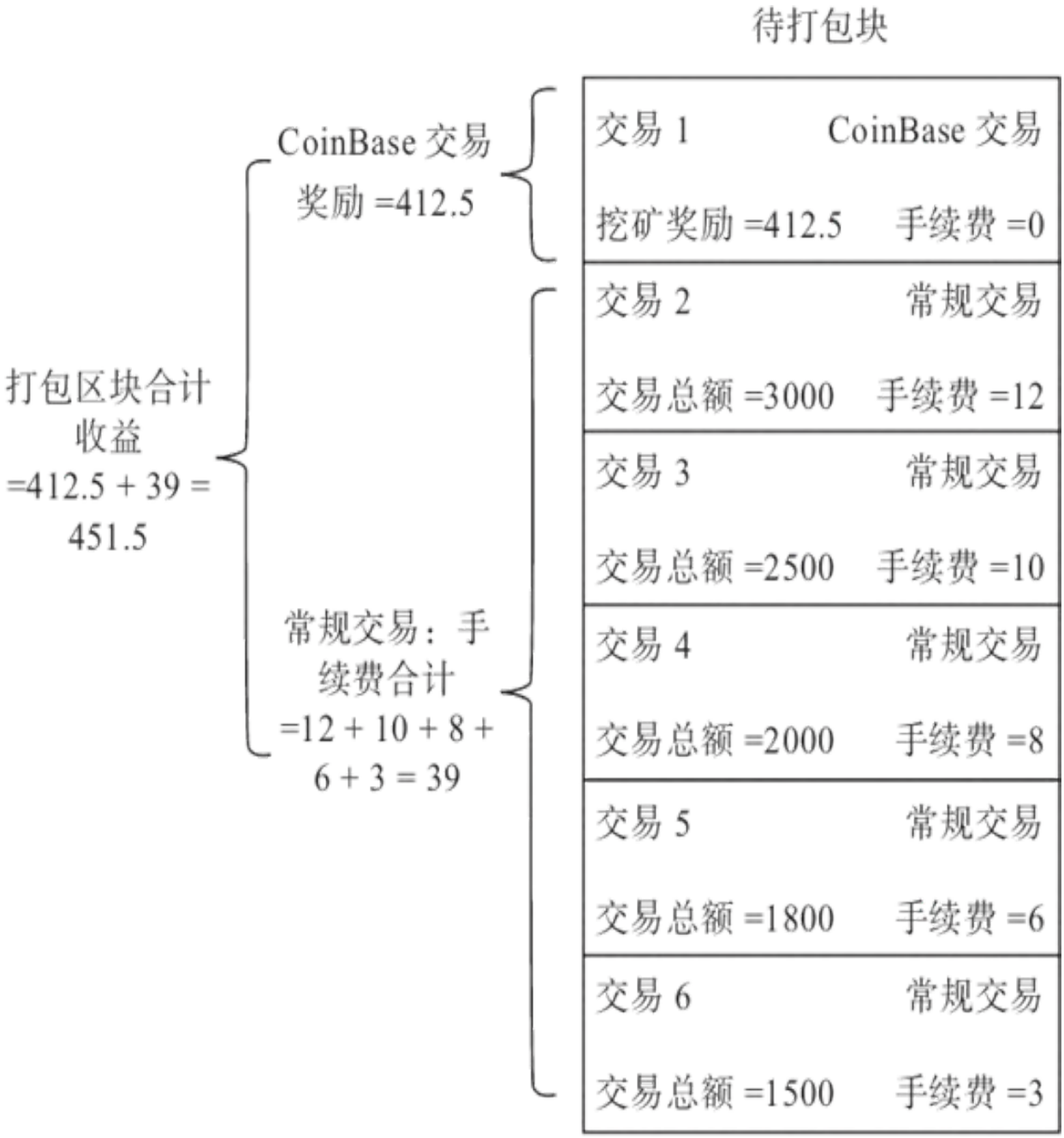


图13-2　交易打包

从图13-2中可以看到，矿工收益源自CoinBase交易奖励和交易的手续费。想要提升挖矿收益，只能从这两方面入手。由于CoinBase交易奖励在固定时间段内是定值，因此，只有尽可能提升常规交易的手续费才能最大化挖矿收益。

比原链的交易打包过程中，在选择交易列表时，按照时间进行排序（无论手续费高低都会按照时间顺序进行打包）。目前也有部分矿池按照Gas从大到小的顺序选择交易列表（优先将高Gas的交易打包），这样做的目的是将矿池的利益最大化。

13.8.1 Coinbase交易奖励

在区块链上每一个块中交易列表第一个元素往往是CoinBase交易（又叫创币交易）。这个交易由挖矿节点构造，目的是给打包交易节点以奖励。

具体一次打包的奖励数要参考比原链挖矿的减半周期，在比原链中可供挖矿的BTM数量是6.93亿个，每挖出840000个BTM，挖矿奖励减半。平均2.5分钟挖出一个块，如果要挖出840000个比原，大概需要4年（840000×2.5=2100000分钟）时间，即BTM每4年奖励减半，本书截稿之前，在比原链中每挖到一个块奖励412.5BTM。

CoinBase奖励公式：

412.5*2^（-Trunc（height/840000））

以当前高度除以840000取整（Trunc），再取负数作为2的指数。412.5表示初始奖励。代码示例如下：

```
consensus/general.go
func BlockSubsidy(height uint64) uint64 {
    if height == 0 {
        return InitialBlockSubsidy
    }
    return baseSubsidy >> uint(height/subsidyReductionInterval)
}
```

CoinBase交易打包构造过程的代码示例如下：

```
mining/mining.go
b.Transactions[0], err = createCoinbaseTx(accountManager,
txFee, nextBlockHeight)
if err != nil {
    return nil, errors.Wrap(err, "fail on createCoinbaseTx")
}
```

createCoinbaseTx函数创建CoinBase交易，并将该笔交易放入交易列表中的第一个元素中。代码如下：

```
mining/mining.go
func createCoinbaseTx(accountManager *account.Manager, amount
uint64, blockHeight uint64) (tx *types.Tx, err error) {
    amount += consensus.BlockSubsidy(blockHeight)

    // ...

    builder := txbuilder.NewBuilder(time.Now())
    if err =
builder.AddInput(types.NewCoinbaseInput([]byte(string(blockHeig
ht))), &txbuilder.SigningInstruction{}); err != nil {
        return
    }
    if err =
builder.AddOutput(types.NewTxOutput(*consensus.BTMAssetID,
amount, script)); err != nil {
        return
    }
    // ...
}
```

createCoinbaseTx函数主要包含以下过程。

· consensus.BlockSubsidy：根据全网最新高度获取当前CoinBase交易奖励。

· 根据矿工提供的账户信息，生成coinbase交易的output的锁定脚本。如果矿工提供的是一个空账户，则生成的锁定脚本是一个任何人都可以解锁的脚本（即任何知道该UTXO地址的人都可以花费它）。如果账户信息不为空，则会根据账户中的密钥个数生成相应的单签或多签地址。

· txbuilder.NewBuilder：构建一笔交易，填充arbitrary字段到inputs中，其中coinbase的input并无引用。

13.8.2 交易手续费Gas

每笔交易的发生都需要消耗手续费Gas，这部分手续费支付给网络上成功进行块打包（包含本笔交易的块）的挖矿节点。

挖矿节点从交易池中获取待打包的交易（待确认的交易列表）。目前比原链是按照时间顺序（无论手续费高低都会按照时间顺序进行打包）对交易列表进行排序。交易列表选择代码示例如下：

```
mining/mining.go
func NewBlockTemplate(c *protocol.Chain, txPool
*protocol.TxPool, accountManager *account.Manager) (b
*types.Block, err error) {
    // ...

    txs := txPool.GetTransactions()
    sort.Sort(ByTime(txs))
    for _, txDesc := range txs {
        tx := txDesc.Tx.Tx
        gasOnlyTx := false

        if err := c.GetTransactionsUtxo(view, []*bc.Tx{tx});
err != nil {
            // ...
            txPool.RemoveTransaction(&tx.ID)
            continue
        }

        gasStatus, err := validation.ValidateTx(tx, bcBlock)
        if err != nil {
            if !gasStatus.GasValid {
                // ...
                txPool.RemoveTransaction(&tx.ID)
                continue
            }
            gasOnlyTx = true
        }

        if gasUsed+uint64(gasStatus.GasUsed) >
consensus.MaxBlockGas {
            break
        }
```

```
        if err := view.ApplyTransaction(bcBlock, tx,
gasOnlyTx); err != nil {
            // ...
            txPool.RemoveTransaction(&tx.ID)
            continue
        }

        txStatus.SetStatus(len(b.Transactions), gasOnlyTx)
        b.Transactions = append(b.Transactions, txDesc.Tx)
        txEntries = append(txEntries, tx)
        gasUsed += uint64(gasStatus.GasUsed)
        txFee += txDesc.Fee

        if gasUsed == consensus.MaxBlockGas {
            break
        }
    }

    b.Transactions[0], err = createCoinbaseTx(accountManager,
txFee, nextBlockHeight)
    // ...
    txEntries[0] = b.Transactions[0].Tx

    b.BlockHeader.BlockCommitment.TransactionsMerkleRoot, err =
bc.TxMerkleRoot (txEntries)
    // ...

    b.BlockHeader.BlockCommitment.TransactionStatusHash, err =
bc.TxStatusMerkleRoot (txStatus.VerifyStatus)
    return b, err
}
```

NewBlockTemplate函数说明如下。

· txPool.GetTransactions：从交易池中获取待确认的交易列表。

· sort.Sort：根据时间顺序对待确认交易进行排序。

· 遍历待确认交易列表：验证交易，计算Gas费用。其中gasUsed值是根据BTM兑换过来的（1 Gas==200 Neu），txFee值是BTM单位。

· createCoinbaseTx：创建CoinBase交易。

· bc.TxMerkleRoot：构建交易的merkle树。

· bc.TxStatusMerkleRoot：构建交易状态的merkle树。

13.9 矿池优化建议

1. 性能

针对挖矿这样一个以效率取胜的数学游戏，性能是衡量一个矿池是否合格的重要因素，在此提出几个性能优化的方向：

· 任务的存储与下发。矿池从节点获取任务后进行任务的拆解与下发，拆解后的任务建议存放在内存中而非数据库中，方便矿机提交任务后尽快获取相关任务信息进行块验证，减少数据库硬盘IO，只有当新的任务更新时，所有任务信息才通过其他线程批量写入数据库。任务的下发通过MQ等消息中间件以及定时任务进行批量主动推送。

· 任务验证。由于使Tensority算法中穿插了矩阵运算，为了尽可能快地进行块验证，一定要使用GPU加速，在矿机任务验证过程中建议采用抽样算法进行部分块的难度和结果验证。

· 并发。在处理海量矿机并发以及短时间内任务多频次提交与下发时，建议采用Netty等技术对高并发提供支持，同时合理使用负载均衡保证矿池多节点压力均摊。

· 网络。影响矿池性能非常关键的因素是网络延时，一定要保证矿机拥有足够的网络带宽。有效降低延迟的最好办法是尽可能少地进行网络请求，最大限度地压缩网络请求次数，在必要的情况下可以针对Stratum协议报文进行自定义。

2. 安全

关于安全方面的建议如下：

· 防DDoS攻击。大多数矿池的终极灾难是DDoS攻击，由于矿池的地址是公开的，只要对外提供服务便很难避免恶意者进行攻击，目前比较有效的防范并降低DDoS攻击的建议是，直接采购成熟安全厂商的防DDoS服务，一般会以流量计费的方式收费。这类防范往往针对恶意攻击者，提供攻击前预判、攻击中导流、攻击后恢复等多阶段防范措施。

· 矿池钱包密钥防泄密。矿池的本质是在一个全节点钱包的基础上提供第三方服务。对于全节点钱包，在节点正常运行后，建议把钱包密钥文件备份到离线介质上，并立即删除服务器上的密钥文件。妥善保管存有钱包密钥的载体。

· 常规措施。在做好以上针对区块链的安全措施之后，还要做好传统服务器软件的安全措施，比如防SQL注入、防拖库、权限控制、防缓冲区溢出、防服务器漏洞等常规安全措施，不可忽略。

13.10 本章小结

本章主要从矿池、矿机等多角度分析与挖矿相关的内容，基于节点网络层、矿池层、矿机层三层来分析矿池的架构，从矿池和矿机两个视角分析挖矿的流程，详细说明了目前主流的矿池收益分配模式及矿池的优化建议等。

通过本章的学习，读者可以了解矿池与挖矿相关的主要流程，以及实现挖矿相关的技术点。

第14章

展望

14.1 概述

2008年11月1日，一篇讲述“我正在开发一种新的电子货币系统，采用完全点对点的形式，而且无须授信第三方的介入”的帖子悄然出现在网络上。

2018年正好是中本聪发布《比特币白皮书》十周年的日子。这十年时间，区块链从一个无人问当津的边缘世界走入了矛盾的传统世界，就像闯入了瓷器店的猴子，引起了所有人的注意，也培养出一批布道者和创业者。人们对区块链对未来的影响已达成共识。

从生命周期理论来说，区块链刚刚走出了第一阶段——发展阶段。前期由一些信仰者和早期尝试者推动了这个由技术极客发展起来的革命性产品。目前的区块链确实有很多问题，但这也正是现阶段区块链的魅力所在。接下来将会进入区块链的成长阶段，未来的跨链、侧链、闪电网络、子链等技术发展会推动区块链向成长阶段进阶。这个阶段的特征就是，技术慢慢变得成熟，技术不再是具体落地场景中的障碍，为区块链最终进入成熟阶段做好技术铺垫。

数据共享网络Spring Labs的首席执行官及联合创始人Adam Jiwan坚信：这场技术革命是无法阻挡的。在分布式总账技术的开发方面，现在才刚刚过去了十年而已。区块链技术在将来会变得无处不在，整个世界都必须适应这种形势。现在唯一的问题就是，这种情况将会多快发生。

本章从影响应用落地的技术困难出发，展望各类技术的发展趋势。

14.2 跨链

14.2.1 打通链与链的连接

从最初的比特币PoW，到后来的以太坊PoS，再到最近EOS使用的DPoS，随着技术的发展，越来越多的公链涌现出来，因为每条链之间彼此没有任务连接方式，好似每条公链都是一座孤岛，在很大程度上限制了区块链应用的发展。跨链就是架在各个孤岛之间的桥梁，在每条公链之间实现价值的自由转换，跨链技术是形成区块链生态的核心环节之一。

现有的跨链技术大致可以分为以下三类：

· 公证人机制（Notary schemes）。公证人机制是链与链之间操作最简单的方式，就是在不信任的A和B之间引入一个双方都信任的第三方充当公证人（中介），即中心化或多重签名的见证人模式。见证人是链A的合法用户，负责监听链B的事件和状态，进而操作链A。本质特点是完全不用关注所跨链的结构和共识特性等。

优点：公证人机制是双向跨链，能支持跨链资产交换及转移，是跨链合约和资产抵押的自由交易。链与链互操作简单，不需要工作证明和复杂证明。

缺点：这种机制和区块链的去中心化的理念存在一些冲突，所以很多人不认为它是区块链，而更多是一种中心化的产物。

· 侧链/中继器模式（Sidechains/relays）。一般来说，主链不知道侧链的存在，而侧链必须知道主链的存在。RootStock是一个建立在比

特币区块链上的智能合约分布式平台，目标是将复杂的智能合约实施为一个侧链，为核心比特币网络增加价值和功能。

比较有代表性的中继器模式是polkadot中继链。中继是链与链之间的通道，如果通道本身是区块链，那就是中继链。这个项目采用技巧性的办法，即多重签名的机制，对主链资产进行锁定，在侧链上锚定、执行，在侧链上的交易通过多重签名、共同投票决定交易是否有效。

·哈希锁定。哈希锁定起源于比特币闪电网络，闪电网络本身是一种小额的快速支付手段，后来其关键技术——哈希时间锁合约被应用到跨链技术上。虽然哈希锁定实现了跨链资产的交换，但是没有实现跨链资产的转移，更不能实现这种跨链合约，所以它的应用场景是相对受限的。

14.2.2 BTC、ETH与BTM的跨链资产交换

比原链是一种多元比特资产的交互协议，运行在比原链链上的不同形态、异构的比特资产（原生的数字货币、数字资产）和原子资产（有传统物理世界对应物的权证、权益、股息、债券、情报资讯、预测信息等）可以通过该协议进行登记、交换、对赌，以及基于合约的更具复杂性的交互操作。

双向锚定（two-peg）技术经常被认为是一种将BTC转换到SBTC（智能比特币，这里我们可以理解为就是现在的侧链）的方法，反之亦然。实际上，当BTC被转换成SBTC，区块链之间并没有货币被“转移”，没有任何一笔交易实施了这个动作。这是因为比特币不能验证另一条区块链上的余额属性。当用户打算把BTC转换成SBTC，将在比特币区块链上锁定部分比特币，同时在RSK上释放等量的SBTC。当SBTC需要换回比特币时，再次在RSK上锁定SBTC，同时在比特币区块链上释放等量的比特币。通过安全协议保证相同的比特币不会在两条区块链上同时释放。

比原链是基于双向锚定、实现联合锚定（fed-peg）的侧链模型。双向锚定与交易所的第三方担保不同，双向锚定引入多方中间人机制并使用中间人的多重签名来实现锚定资产的安全。

在这里比原主链不是作为比特币的侧链，而是为比原链生成一个比原侧链，这条侧链既可以锚定BTC，也可以锚定BTM，侧链在BTC锁定资产再释放对应资产到比原链上，实现了两条资产价值转移。

14.3 闪电网络

比特币拥堵问题由来已久，社区对链上链下的扩容方案也一直存有争议。BCH分叉后通过大区块进行扩容，而BTC则选择激活隔离见证，采用闪电网络off-chain来扩容。

闪电网络（Lighting Network）是一个链下服务方案，它不发送任何货币，而是在第二层级中进行账本的变更，随后在第一层级中完成结算，从而避免实际发生的数千次的链上资金交易。

通过将资金发送到由多方掌管密钥的多重签名地址，闪电网络构建起一个支付渠道。收付双方的交易在链下完成，无论这个交易渠道关闭时的余额是多少，这些余额都会被如数发回用户的钱包。这就是双向支付渠道的工作原理。闪电网络是一个典型的双向支付渠道网络，通过这个网络，用户可以与自己的承包商进行定期支付或每月结算。

闪电网络的目的是实现安全的链下通道交易，在链下通道内使用哈希时间锁定智能合约安全地进行点对点交易。实现交易0确认极低的交易费用。

闪电网络的核心的概念有两个：可撤销的顺序成熟度合同（Recoverable Sequence Maturity Contract，RSMC）和哈希的带时钟合约（Hashed Timelock Contract，HTLC）。RSMC保障了两个人之间的直接交易可以在链下完成，HTLC保障了任意两个人之间的转账都可以通过一条“支付”通道来完成。这两个类型的交易组合构成了闪电网络。从而实现任意两个人都可以在链下完成交易。

RSMC类似于准备金制度，即双方发生交易的条件是双方需要在一个微支付通道（资金池）中预存一部分资金，之后每次交易，要对交

易后的资金分配方案共同进行确认，同时签字作废旧的版本。当需要提现时，将最终交易结果写到区块链网络中，被最终确认。可以看到，只有在提现的时候才需要通过区块链。

HTLC可以理解为限时转账，即通过智能合约，双方约定转账方先冻结一笔钱，并提供一个哈希值，如果在一定时间内有人能提出一个字符串，字符串哈希后的值与已知值匹配（实际上意味着转账方授权了接收方来提现），则这笔钱转给接收方。

闪电网络采用了更合理的支付网络架构，提高了效率。不是向所有人广播交易，而是更直接地将交易发送给收款人。只有当交易双方不诚实时，才需要进入繁琐的流程——链上共识操作。通过这种方式，可以实现相当于互联网上各方直接沟通所能达到的性能和效率，同时保留比特币区块链的一些安全特性。然而，如果各方想在出现问题时随时回归到区块链上并收回资金，那么建立这样一种支付系统是非常复杂的，而且存在着一些重大风险和局限性。

14.4 子链

长铗在2014年提出了“不可能三角”理论，即去中心、安全、效率这三者不能同时共存。“三体世界”中解决一个平行世界不可调和的矛盾时，需要升维思考才能降维解决问题。但可惜的是以太坊与EOS还是在二维世界中进行升级改造，所以无法突破“不可能三角”，只能在三者之间做取舍。

如果把区块链理解为一维世界，解决了去中心化分布式存储问题以及在不可信环境下的信任问题（这与现在联盟链形态非常相似），那么比特币就是基于区块链的二维世界，解决了金融资产发行、结算、记账、节点激励问题，实现了价值传输，而且是一个金融自洽的体系。

如果要构建一个基于区块链之上的三维世界，我们不仅仅需要一个token，还要在token上实现商业价值的传递，在这个价值传递的过程中，我们并不需要像比特币那样特别强调去中心化与安全。

比如，我们在星巴克点一杯焦糖玛奇朵，此订单确认的过程并不需要全节点确认验证，只是我们双方签名确认就可以了，为什么要广播给所有人，让所有人知道这事呢？我点单完成后买单的时候，我支付给商家的钱就需要全节点验证了，否则商户会担心我是否有双花攻击嫌疑，导致他收不到款。如果我要开发票，这些还应该有税务局加入进来，这张发票也仅限于我、星巴克、税务局三者确认就可以了，别人也不需要知道。对于星巴克来说，他更加不想让别人知道自己的生意如何，只要参与方认可就行，也没有双花问题，因而不需要全局共识。这里的订单和开发票的过程，并不需要像比特币那样的全局帐本统一验证过程，只需要参与方验证即可。其实，很多场景下都没有双花问题，这类不需要考虑双花问题的共识称为局部共识，与此相对的就是全局共识。

这个订单的数据不需要全部节点同步储存，也不需要全部节点运行智能合约，这里的逻辑就是，不要使用比特币的数字货币场景逻辑处理整个事务，而要根据实际场景分层处理。这里的分层不是以太坊的分片，分片是对所有事务纵向切割，将整体事务并行处理后再组合，但是不同分片之间的数字资产共识同步是巨大的挑战。分层是对单个事务的横向切割，将单一事务解构为多个非关联事务。

针对此情境设计的子链是一种分层平行扩展方式，在共识上进行分层处理，是在主链的平台上面派生出来的，具有独立功能的区块链。但子链是不能单独存在的，它需要用到主链全局共识机制，保障子链的数据安全，共用主链的智能合约实现资产的转账等交易。

因为子链是针对局部达成共识的方式，所以子链可以无限增加节点的方式实现平行扩展。在使用全局共识的时候，因受到节点数量的限制导致性能上终归有天花板，像现在主流的Pos共识模式使用最多的拜占庭容错技术（BFT），仍存在节点之间异步网络连接、乱序、延时等问题，需要将节点控制在一定范围之内，最终还是受“不可能三角”理论的限制。

子链的挑战在于子链与公链之间的融合方式，在验证节点中需要支持子链的操作，这将比侧链对主链的要求更高，需要主链进行相应的修改，这也是目前Nervos、PlatOn同时搭建主链与子链的原因。当子链这类扩展方式成熟之后，将会出现各类应用场景的子链，包括快速支付、可信存储、可信计算等。

14.5 本章小结

中本聪发布《比特币白皮书》已经十年，分片、DAG、闪电网络、子链、随机数生成、WebAssembly虚拟机等技术的成熟，将会使各种应用场景开始真正落地，不再与传统世界隔离。电子邮件使得跨国通信瞬时实现，而数字货币将会开启同样的征程，将跨国支付的成本降到最低，效率提到最高。

今年各类稳定币的出现，能够让结算双方不受区块链主网币本身价格波动的影响，为自动清算扫除了最后的障碍，只待技术的成熟，电子货币将会在下一个经济周期中发挥重要的作用。

区块链作为一种影响到未来重要变革的技术，成为一个风口是必然的事，我们需要做的只是站在风口上。