区块链底层设计

Java实战

牛冬 / 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

2018 年是中国区块链发展的元年，火热的市场环境下，各互联网公司纷纷试水区块链落地项目。

本书以区块链原理及其对应的 Java 实现为主线展开，详细剖析区块链底层技术，主要内容包括区块链的底层架构、密码学原理、P2P 网络原理、分布式一致性算法、知名公链区块设计、知名公链区块存储技术、知名公链币的设计、联盟链管理后台的原理等。读者在学完本书后，可自行设计联盟链。

本书内容基于 Java 语言，为读者打开了区块链底层研发大门。

图书在版编目（CIP）数据

区块链底层设计 Java 实战 / 牛冬编著. —北京：电子工业出版社，2019.1
ISBN 978-7-121-35525-7

Ⅰ. ①区… Ⅱ. ①牛… Ⅲ. ①电子商务－支付方式②JAVA 语言－程序设计 Ⅳ. ①F713.361.3
②TP312.8

中国版本图书馆 CIP 数据核字(2018)第 259630 号

责任编辑：安　娜
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：18.5　字数：352.8 千字
版　　次：2019 年 1 月第 1 版
印　　次：2019 年 1 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序

自 20 世纪 80 年代计算机技术兴起以来，几乎每隔 10 年就会有一次比较重大的技术变革。20 世纪 90 年代互联网和搜索技术从美国风靡全球；21 世纪初，云计算和移动互联网成了弄潮儿。而今，区块链浪潮袭来！特别是 2018 年初以来，区块链成了街谈巷议的话题。

在招聘市场上，区块链人才炙手可热，一些城市为了吸引区块链人才下足血本。杭州区块链产业园为了吸引高层次区块链人才入驻，实行购房补贴最高达 300 万元人民币的政策，并补贴公用住房、云服务补助和研发经费；上海杨浦区对引入的区块链人才给予 10 万元人民币住房补贴，补贴最长时限可达 3 年。

在研发领域，各个互联网公司纷纷试水区块链落地项目。与之对应，区块链职位的招聘市场也是热火朝天。随着区块链研发的热浪，区块链培训机构也纷纷进场。目前区块链研发培训周期基本都在两周左右，收费 20000 元人民币以上。但培训效果却不尽如人意，笔者面试区块链岗位的人才时，发现几乎所有的小伙伴只是对区块链概念有模糊印象，至于如何实战知之甚少，如何自研区块链底层技术知之更少。

所以说，区块链人才的火热最主要或者最根本的原因是真正的区块链高端人才极为稀少。因此，这也成了本书写作的初衷，即试图降低区块链底层学习和开发的门槛，缩小学习区块链原理和理论到进入实战的鸿沟。

本书以区块链原理及其对应的 Java 实现为主线展开，各章内容如下：

第 1 章是区块链简介，从研发维度戏说、正说区块链，评说区块链的应用前景。

第 2 章介绍区块链的底层架构。

第 3 章讲区块链中所用的密码学原理及 Java 实现。

第 4 章讲 P2P 网络原理及 Java 实现。

第 5 章讲分布式一致性算法及 Java 实现。

第 6 章讲知名公链的区块设计及 Java 实现。

第 7 章讲知名公链的区块存储技术及 Java 实现。

第 8 章讲知名公链币的设计及 Java 实现。

第 9 章讲联盟链管理后台的原理及实现。

第 10 章讲联盟链的运营。

本书适用于区块链爱好者、区块链初学者、想自行开发设计区块链底层的有 Java 基础的读者。

当然，笔者学习和实践区块链技术刚刚 2 年，因此书中难免有理解和实践不足之处，“卑辞俚语，不揣浅陋”，欢迎读者和笔者交流学习，共同进步，一起为区块链落地和人才培养体系建设略尽绵薄！

目录

第 1 章
区块链简介

与君初相识
犹如故人归

1.1 戏说区块链

当笔者奉调出任区块链研发负责人之初，加班相对之前又多了些。加班多了，自然陪家里小宝宝玩儿的时间就少了。为此，小宝宝有点儿不开心。

家里三岁的小宝宝和笔者有过这样一段对话。

小宝宝：“爸爸，你怎么不回来陪我玩儿啊，我睡觉的时候你还没回来！”

笔者：“宝宝，爸爸去做区块链了。事情很多，所以加班多啦。”

小宝宝：“什么是区块链啊？好玩不？”

笔者：“区块链是一个游戏，这个游戏可好玩了！”

小宝宝：“我也想玩，怎么玩啊？”

笔者：“比如，过年的时候，你会收到什么呀？”

小宝宝：“压岁钱！”

笔者：“对，那爸爸妈妈还会说什么呢？”

小宝宝：“爸爸妈妈先把毛爷爷帮我收起来，我长大了再花！”

笔者：“对。可是，如果等你长大了，爸爸妈妈没给你曾经收到的这么多压岁钱花，你怎么办？”

小宝宝：……

笔者：“有了区块链就不会出现这种假设的问题啦。比如过年的时候，爷爷给了你 1000 块压岁钱，爷爷就在自己的小本本上写：今天给了宝宝 1000 块压岁钱。然后爷爷大声告诉奶奶、爸爸、妈妈：‘我今天给了宝宝 1000 块压岁钱。’奶奶、爸爸、妈妈听到之后都在自己的小本本上写：爷爷今天给了宝宝 1000 块压岁钱。”

“然后奶奶给了你 2000 块压岁钱，奶奶就在自己的小本本上写：今天给了宝宝 2000 块压岁钱。然后奶奶大声告诉爷爷、爸爸、妈妈：‘我今天给了宝宝 2000 块压岁钱。’爷爷、爸爸、妈妈听到之后都在自己的小本本上写：奶奶今天给了宝宝 2000 块压岁钱。”

“之后爸爸给了你 3000 块压岁钱，爸爸就在自己的小本本上写：今天给了宝宝 3000 块压岁钱。然后爸爸大声告诉爷爷、奶奶、妈妈：‘我今天给了宝宝 3000 块压岁钱。’爷爷、奶奶、妈妈听到之后都在自己的小本本上写：爸爸今天给了宝宝 3000 块压岁钱。”

“最后妈妈给了你 4000 块压岁钱，妈妈就在自己的小本本上写：今天给了宝宝 4000 块压岁钱。然后妈妈大声告诉爷爷、奶奶、爸爸：‘我今天给了宝宝 4000 块压岁钱。’爷爷、奶奶、爸爸听到之后都在自己的小本本上写：妈妈今天给了宝宝 4000 块压岁钱。”

“每年过年大家给完宝宝压岁钱之前后都这样写在小本本上，然后告诉其他人也写在自己的小本本上，这就是区块链。”

“我们还可以约定宝宝 10 岁时就可以拿出 1000 块钱买好吃的，到你 10 岁的时候呢，1000 块钱就会送到你手上。这就是智能合约。”

小宝宝：“区块链这个游戏真好玩，快给我拿纸和笔，我也要写！”

1.2　正说区块链

2018 年年初以来，区块链一词火遍了大街小巷，出租车司机、程序员、培训机构等都在谈论区块链。区块链在网络上也成了热门搜索词汇，在百度指数上搜索区块链可以看到，区块链从 2018 年 1 月陡然成为热词，之后持续保持“网红”趋势，搜索指数居高不下，如图 1-1 所示。

什么是区块链呢？

区块链一词最早出现在中本聪（Satoshi Nakamoto）的“比特币：一种点对点的电子现金系统（*Bitcoin: A Peer-to-Peer Electronic Cash System*）”一文中。

在比特币系统中，区块链作为存储底层，承载了上层众多认同比特币相关协议并严格遵守及维护协议的节点（个人或组织）的记录各类信息的行为。

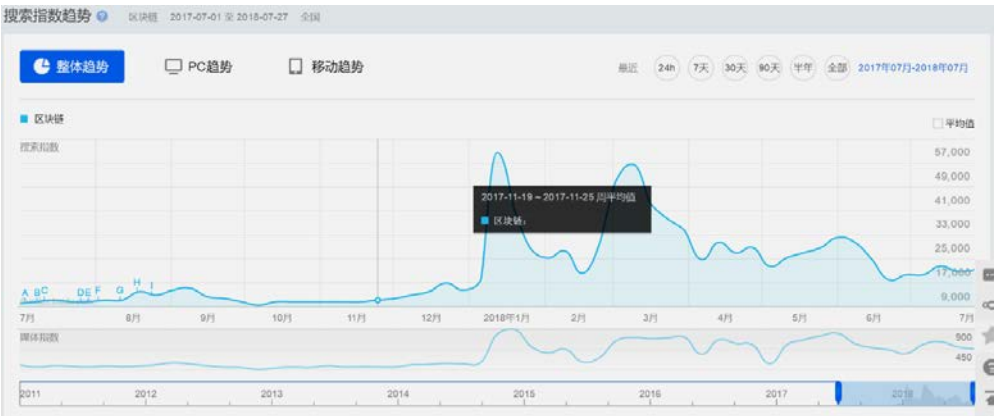


图 1-1　区块链百度搜索指数

区块链的存储是一种链式存储，区块按照生成的时间顺序前后链接，区块的链接基于区块存储内容的哈希值构建。区块生成后会在区块链系统的各个节点（个人或组织）中同步，因此各个节点最终均保存了一份完整且一致的数据。也就是说，区块链系统会保持最终一致性，但不保证实时一致性。

比特币系统中理论上不支持数据的删除和修改。数据的删除和修改意味着其修改所在区块的哈希值也需要随之修改，而该区块之后的所有区块的前向链接哈希值均需改变。随着比特币系统中节点（个人或组织）数量的增多，前向哈希值的修改和同步工作会愈发繁重。在比特币全网络中对数据删除和修改变得愈发困难，因此比特币系统中区块数据的不可篡改性就体现了出来。

由于加入比特币网络的节点（个人或组织）严格遵守比特币协议，因此区块链从理论上具备了承载信用的特性。

从技术上来说，区块链技术并非凭空产生，而是基于已有技术演变而来的。

区块链的链式构成和我们上学时所学的《数据结构和算法》这类书籍中的单链表有一定的相似性，链表中的各部分内容均通过指针（哈希值也是泛化指针的一种形式）相连。不同之处在于，单链表是从前向后构建关联关系，即新插入一个内容时，需从链表的表头开始移动指针至链表尾部，再将尾部的指针指向新建的内容，新建的内容自动成为链表的尾部。而区块链是从后向前构建关联关系，即新生成的区块通过前面区块的哈希值与前面区块建立连接。

区块中 header+body 的结构设计与 Web 开发中常见的 HTTP 协议有一定的相似性。HTTP 协议的 request 和 response 均由三部分组成：request/response line、request/response body、request/response header。

区块内容的加密就更常见了，无论是对称加密，抑或是非对称加密，在 MySQL 敏感数据存储、接口参数加密、OAuth 授权等无处不在。而区块内容中多条交易信息对应的 Merkle 树源于《数据结构和算法》中的二叉树。

区块链中 P2P 网络是区块链系统的基础。共识的达成、数据的同步均依赖于高效的 P2P 网络系统。P2P 网络的构建源于少见的 Java 网络编程部分。在生活中，QQ、微信、微博等场景均有类似应用。如果读者研究过 ElasticSearch、Redis 集群、Kafka 集群等数据同步的代码，对开发和构建 P2P 网络应该更有信心。不过随着技术的发展，Java 开发小伙伴可以不再基于原始的 Socket 进行开发，可以选用 WebSocket 或 t-io 等组件进行开发。

区块链上共识的达成基于共识算法。共识算法的起源算法很多读者都熟知，如 Paxos 一致性算法，分布式数据库的二、三阶段提交协议、ZooKeeper 的快速选举算法等。当然，不同的公链、联盟链的共识算法在不同场景有不同的选择，但算法基础都是一脉相承的。

区块的存储和常见的分布式应用开发略有不同。在分布式应用开发中，我们熟悉的分库分表、分布式缓存集群等并不适用。区块链的每个节点是一个单机，存储也随本机进行。因此，区块的存储往往选用高性能的文件系统或高性能的单机版 SQL/NoSQL 数据库，如 SQLite、LevelDB、RocksDB 等。当然，单机的存储空间始终是有限的，随着存储承载压力的增加，后续区块链存储的扩容也可能成为区块链领域的另一种创新。

看到这么多相似的技术，是不是已经觉得区块链底层的研发不再高不可攀了呢？来吧，笔者将带你走进区块链底层开发世界，曾经以为的高门槛将不再遥不可及！

1.3　区块链的未来：联盟链

2018 年伊始，区块链在国内开始火热起来。各个行业各个领域都开始寻找自身

业务和区块链的结合，不断有一些尝试性的落地应用上线，其中以金融、游戏领域为主，内容、房产信息、商品溯源等场景也开始引入区块链。

不同场景下，区块链类型的选型也不尽相同。

在区块链世界中，一般划分为公链、联盟链、私链。从技术视角而言，公链、联盟链、私链的底层技术大抵相同，但适用场景则大相径庭。我们可以从不同的视角来看待公链、联盟链、私链。

从去中心化这一区块链系统最大的特色来说，公链是完全去中心化的，公链网络的各个节点都可以读取和写入数据；联盟链则是部分去中心化，节点由联盟内成员部署，读写权限也可以根据联盟内的协议来定制；私链本质上还是中心化的，数据的写入由私链所属组织控制，数据的读取和使用则由私链所属组织的应用场景来定。

从代码开源程度而言，公链的开放程度最大，全世界的个人和组织均可获取其完整代码，个人和组织可以直接部署，亦可修改为己所用。联盟链则是对联盟内部用户开源其最核心代码，代码的写权限可以在联盟内部分级管理。私链的代码归属于个人或组织，一般不对外开放源码。

从激励体系角度而言，激励体系是公链的灵魂，不可或缺；联盟链可以根据场景选择是否使用激励体系；私链则更加灵活。

那么公链、联盟链、私链谁代表了区块链未来的发展方向呢？特别是谁代表了企业级区块链的未来呢？笔者判断是联盟链。

激励体系作为公链的灵魂，其对应的优秀经济模型的设计比较困难。同时，激励体系往往是为了吸引更多的节点进入公链挖矿，挖矿在现有的共识算法体系下是一种耗费计算资源的低效行为，并不经济。而且企业相关的业务信息往往私密性较强，不适合进入公链，即便有部分信息可以在公链落地，数据落地的经济成本也不便宜，毕竟矿工打包数据是需要付费的。

而私链既可以使用公链开源代码，也可以基于联盟链开源代码实现，只是节点数量要少得多。私链的应用往往是在组织内部，并不会对外产生多大影响。

联盟链则不然。各个联盟链组建之初往往都立足于行业，着眼于解决行业共性

问题，是能促进行业效率和发展的底层支撑技术。由于企业级应用往往涉及业务逻辑甚至商业机密，因此公链目前并不太适合企业级区块链的应用场景，而联盟链给业务逻辑和商业信息限定了范围，使得区块链技术应用的普适性大大增加。

作为研发，在学会驳接各类公链、联盟链、私链的同时，更应知晓区块链的底层实现技术。本书以联盟链为主线，以区块链底层技术为支撑展开内容，一步步引导读者构建区块链底层平台。

1.4　小结

本章主要从戏说和正说两个维度向读者展示了区块链的概况，以期降低读者心里对区块链底层技术的认知高度，客观而轻松地了解区块链的底层实现概览。

同时向读者分享了笔者对区块链未来的判断，供读者思考和研究。

区块链的研究和落地，我们仍然在路上！

第 2 章
区块链架构

会当凌绝顶
一览众山小

正如开篇所言：会当凌绝顶，一览众山小。进入区块链底层开发前，我们需要了解区块链底层的通用架构是如何设计的，从上而下地审视区块链底层的结构，做到了然于胸，才能胸有成竹。

他山之石，可以攻玉。在介绍区块链底层通用架构之前，我们不妨先从比特币、以太坊、Hyperledger 的架构解读开始。

2.1　比特币架构

根据中本聪的论文“*Bitcoin*: *A Peer-to-Peer Electronic Cash System*”中对比特币系统的描述，我们可以整理出如图 2-1 所示的比特币系统架构。

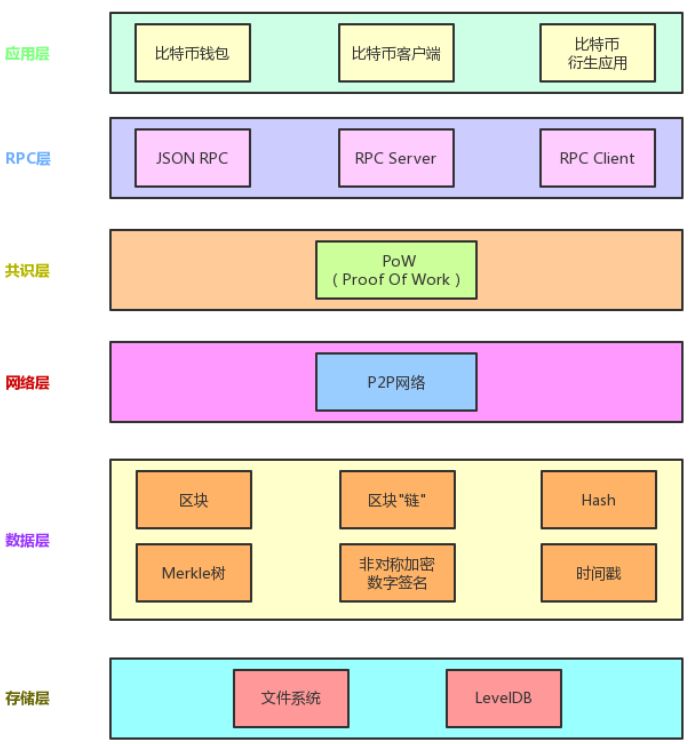


图 2-1　比特币系统架构

如图 2-1 所示，比特币系统分为 6 层，由下至上依次是存储层、数据层、网络层、共识层、RPC 层、应用层。

其中，存储层主要用于存储比特币系统运行中的日志数据及区块链元数据，存储技术主要使用文件系统和 LevelDB。

数据层主要用于处理比特币交易中的各类数据，如将数据打包成区块，将区块维护成链式结构，区块中内容的加密与哈希计算，区块内容的数字签名及增加时间戳印记，将交易数据构建成 Merkle 树，并计算 Merkle 树根节点的哈希值等。

区块构成的链有可能分叉，在比特币系统中，节点始终都将最长的链条视为正确的链条，并持续在其后增加新的区块。

网络层用于构建比特币底层的 P2P 网络，支持多节点动态加入和离开，对网络连接进行有效管理，为比特币数据传输和共识达成提供基础网络支持服务。

共识层主要采用了 PoW（Proof Of Work）共识算法。在比特币系统中，每个节点都不断地计算一个随机数（Nonce），直到找到符合要求的随机数为止。在一定的时间段内，第一个找到符合条件的随机数将得到打包区块的权利，这构建了一个工作量证明机制。从 PoW 的角度，是不是发现 PoW 和分布式锁有异曲同工之妙呢？

RPC 层实现了 RPC 服务，并提供 JSON API 供客户端访问区块链底层服务。

应用层主要承载各种比特币的应用，如比特币开源代码中提供了 bitcoin client。该层主要是作为 RPC 客户端，通过 JSON API 与 bitcoin 底层交互。除此之外，比特币钱包及衍生应用都架设在应用层上。

2.2 以太坊架构

根据以太坊白皮书 *A Next-Generation Smart Contract and Decentralized Application Platform* 的描述，以太坊架构如图 2-2 所示。

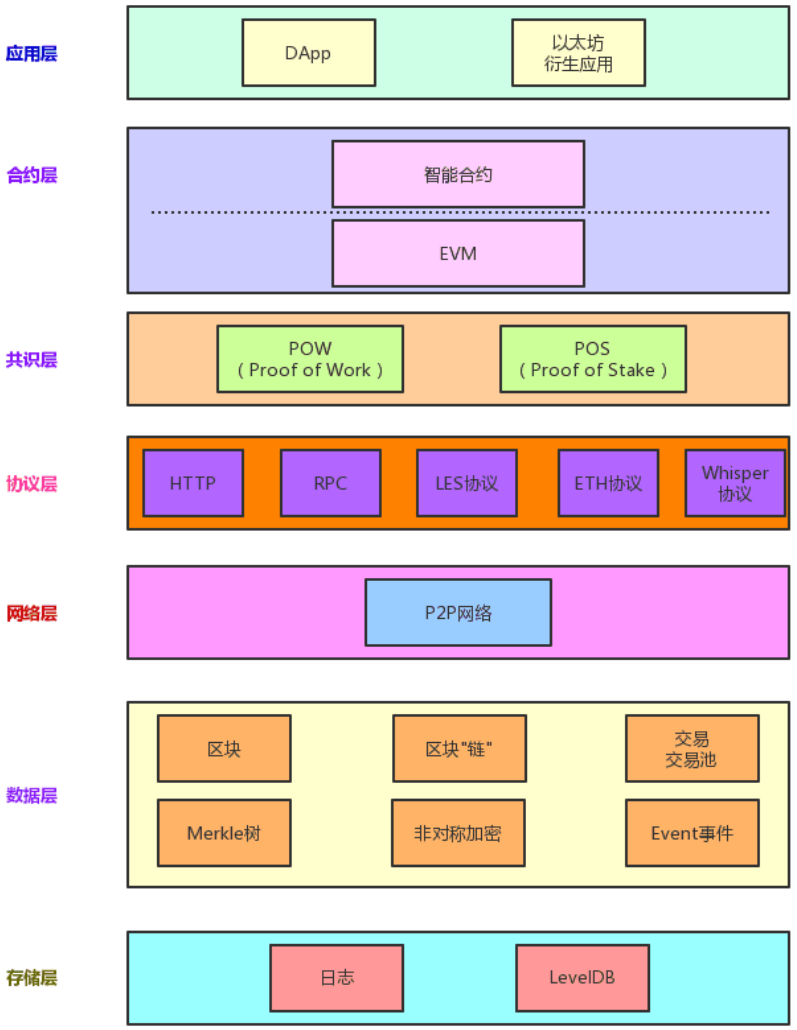


图 2-2　以太坊架构

如图 2-2 所示，以太坊架构分为 7 层，由下至上依次是存储层、数据层、网络层、协议层、共识层、合约层、应用层。

其中存储层主要用于存储以太坊系统运行中的日志数据及区块链元数据，存储技术主要使用文件系统和 LevelDB。

数据层主要用于处理以太坊交易中的各类数据，如将数据打包成区块，将区块维护成链式结构，区块中内容的加密与哈希计算，区块内容的数字签名及增加时间戳印记，将交易数据构建成 Merkle 树，并计算 Merkle 树根节点的 hash 值等。

与比特币的不同之处在于以太坊引入了交易和交易池的概念。交易指的是一个账户向另一个账户发送被签名的数据包的过程。而交易池则存放通过节点验证的交易，这些交易会放在矿工挖出的新区块里。

以太坊的 Event（事件）指的是和以太坊虚拟机提供的日志接口，当事件被调用时，对应的日志信息被保存在日志文件中。

与比特币一样，以太坊的系统也是基于 P2P 网络的，在网络中每个节点既有客户端角色，又有服务端角色。

协议层是以太坊提供的供系统各模块相互调用的协议支持，主要有 HTTP、RPC 协议、LES、ETH 协议、Whipser 协议等。

以太坊基于 HTTP Client 实现了对 HTTP 的支持，实现了 GET、POST 等 HTTP 方法。外部程序通过 JSON RPC 调用以太坊的 API 时需通过 RPC（远程过程调用）协议。

Whisper 协议用于 DApp 间通信。

LES 的全称是轻量级以太坊子协议（Light Ethereum Sub-protocol），允许以太坊节点同步获取区块时仅下载区块的头部，在需要时再获取区块的其他部分。

共识层在以太坊系统中有 PoW（Proof of Work）和 PoS（Proof of Stake）两种共识算法。

合约层分为两层，底层是 EVM（Ethereum Virtual Machine，即以太坊虚拟机），上层的智能合约运行在 EVM 中。智能合约是运行在以太坊上的代码的统称，一个智能合约往往包含数据和代码两部分。智能合约系统将约定或合同代码化，由特定事件驱动触发执行。因此，在原理上适用于对安全性、信任性、长期性的约定或合同场景。在以太坊系统中，智能合约的默认编程语言是 Solidity，一般学过 JavaScript 语言的读者很容易上手 Solidity。

应用层有 DApp（Decentralized Application，分布式应用）、以太坊钱包等多种衍生应用，是目前开发者最活跃的一层。

2.3　Hyperledger 架构

超级账本（Hyperledger）是 Linux 基金会于 2015 年发起的推进区块链数字技术和交易验证的开源项目，该项目的目标是推进区块链及分布式记账系统的跨行业发展与协作。

目前该项目最著名的子项目是 Fabric，由 IBM 主导开发。按官方网站描述，Hyperledger Fabric 是分布式记账解决方案的平台，以模块化体系结构为基础，提供高度的弹性、灵活性和可扩展性。它旨在支持不同组件的可插拔实现，并适应整个经济生态系统中存在的复杂性。

Hyperledger Fabric 提供了一种独特的弹性和可扩展的体系结构，使其不同于其他区块链解决方案。我们必须在经过充分审查的开源架构之上对区块链企业的未来进行规划。超级账本是企业级应用快速构建的起点。

目前，Hyperledger Fabric 经历了两大版本架构的迭代，分别是 0.6 版和 1.0 版。其中，0.6 版的架构相对简单，Peer 节点集众多功能于一身，模块化和可拓展性较差。1.0 版对 0.6 版的 Peer 节点功能进行了模块化分解。目前最新的 1.1 版本处于 Alpha 阶段。

在 1.0 版中，Peer 节点可分为 peers 节点和 orderers 节点。peers 节点用于维护状态（State）和账本（Ledger），orderers 节点负责对账本中的各条交易达成共识。

系统中还引入了认证节点（Endorsing Peers），认证节点是一类特殊的 peers 节点，负责同时执行链码（Chaincode）和交易的认证（Endorsing Transactions）。

Hyperledger Fabric 的分层架构设计如图 2-3 所示。

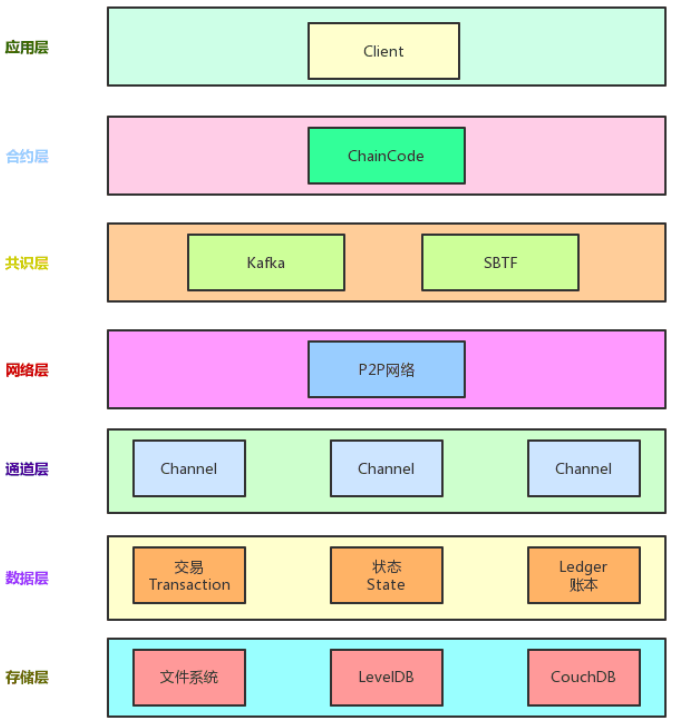


图 2-3　Hyperledger Fabric 的分层架构设计

Hyperledger Fabric 可以分为 7 层，分别是存储层、数据层、通道层、网络层、共识层、合约层、应用层。

其中存储层主要对账本和交易状态进行存储。账本状态存储在数据库中，存储的内容是所有交易过程中出现的键值对信息。比如，在交易处理过程中，调用链码执行交易可以改变状态数据。状态存储的数据库可以使用 LevelDB 或者 CouchDB。LevelDB 是系统默认的内置的数据库，CouchDB 是可选的第三方数据库。区块链的账本则在文件系统中保存。

数据层主要由交易（Transaction）、状态（State）和账本（Ledger）三部分组成。

其中，交易有两种类型：

- 部署交易：以程序作为参数来创建新的交易。部署交易成功执行后，链码就被安装到区块链上。

- 调用交易：在上一步部署好的链码上执行操作。链码执行特定的函数，这个函数可能会修改状态数据，并返回结果。

状态对应了交易数据的变化。在 Hyperledger Fabric 中，区块链的状态是版本化的，用 key/value store（KVS）表示。其中 key 是名字，value 是任意的文本内容，版本号标识这条记录的版本。这些数据内容由链码通过 PUT 和 GET 操作来管理。如存储层的描述，状态是持久化存储到数据库的，对状态的更新是被文件系统记录的。

账本提供了所有成功状态数据的改变及不成功的尝试改变的历史。

账本是由 Ordering Service 构建的一个完全有序的交易块组成的区块哈希链（Hash Chain）。

账本既可以存储在所有的 peers 节点上，又可以选择存储在几个 orderers 节点上。

此外，账本允许重做所有交易的历史记录，并且重建状态数据。

通道层指的是通道（Channel），通道是一种 Hyperledger Fabric 数据隔离机制，用于保证交易信息只有交易参与方可见。每个通道都是一个独立的区块链，因此多个用户可以共用同一个区块链系统，而不用担心信息泄漏问题。

网络层用于给区块链网络中各个通信节点提供 P2P 网络支持，是保障区块链账本一致性的基础服务之一。

在 Hyperledger Fabric 中，Node 是区块链的通信实体。Node 仅仅是一个逻辑上的功能，多个不同类型的 Node 可以运行在同一个物理服务器中。Node 有三种类型，分别是客户端、peers 节点和 Ordering Service。

其中，客户端用于把用户的交易请求发送到区块链网络中。

peers 节点负责维护区块链账本，peers 节点可以分为 endoring peers 和 committing peers 两种。endoring peers 为交易作认证，认证的逻辑包含验证交易的有效性，并对交易进行签名；committing peers 接收打包好的区块，并写入区块链中。与 Node 类似，peers 节点也是逻辑概念，endoring peers 和 committing peers 可以同时部署在一台物理机上。

Ordering Service 会接收交易信息，并将其排序后打包成区块，然后，写入区块链中，最后将结果返回给 committing peers。

共识层基于 Kafka、SBTF 等共识算法实现。Hyperledger Fabric 利用 Kafka 对交易信息进行排序处理，提供高吞吐、低延时的处理能力，并且在集群内部支持节点故障容错。相比于 Kafka，SBFT（简单拜占庭算法）能提供更加可靠的排序算法，包括容忍节点故障以及一定数量的恶意节点。

合约层是 Hyperledger Fabric 的智能合约层 Blockchain，Blockchain 默认由 Go 语言实现。Blockchain 运行的程序叫作链码，持有状态和账本数据，并负责执行交易。在 Hyperledger Fabric 中，只有被认可的交易才能被提交。而交易是对链码上的操作的调用，因此链码是核心内容。同时还有一类称之为系统链码的特殊链码，用于管理函数和参数。

应用层是 Hyperledger Fabric 的各个应用程序。

此外，既然是联盟链，在 Hyperledger Fabric 中还有一个模块专门用于对联盟内的成员进行管理，即 Membership Service Provider（MSP），MSP 用于管理成员认证信息，为客户端和 peers 节点提供成员授权服务。

2.4 区块链通用架构

至此，我们已经了解了比特币、以太坊和 Hyperledger 的架构设计，三者根据使用场景的不同而有不同的设计，但还是能抽象出一些共同点，我们可以基于这些共同点设计企业级联盟链的底层架构。

本书提供的联盟链底层架构如图 2-4 所示。

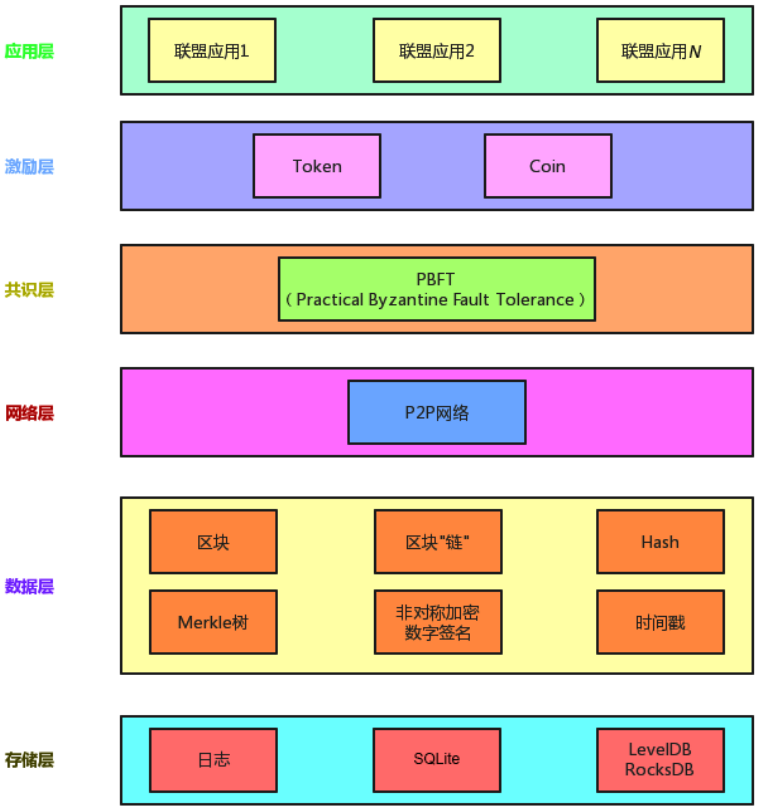


图 2-4　聪盟链底层架构

在图 2-4 中，我们将区块链底层分为 6 层，从下至上分别是存储层、数据层、网络层、共识层、激励层和应用层。

存储层主要存储交易日志和交易相关的内容。其中，交易日志基于 LogBack 实现。交易的内容由内置的 SQLite 数据库存储，读写 SQLite 数据库可以基于 JPA 实现；交易的上链元数据信息由 RocksDB 或 LevelDB 存储。

数据层由区块和区块“链”（区块的链式结构）组成。其中，区块中还会涉及交易列表在 Merkle 树中的存储及根节点哈希值的计算。交易的内容也需要加密处理。由于在联盟链中有多个节点，为有效管理节点数据及保障数据安全，建议为不同节点分配不同的公、私钥，以便加密使用。

网络层主要提供共识达成及数据通信的底层支持。在区块链中，每个节点既是数据的发送方，又是数据的接收方。可以说每个节点既是客户端，又是服务端，因此需要基于长连接来实现。在本书中，我们可以基于 WebSocket 用原生方式建立长连接，也可以基于长连接第三方工具包实现。

共识层采用 PBFT（Practical Byzantine Fault Tolerance）共识算法。不同于公链的挖矿机制，联盟链中更注重各节点信息的统一，因此可以省去挖矿，直奔共识达成的目标。

激励层主要是币（Coin）和 Token 的颁发和流通。在公链中，激励是公链的灵魂；但在联盟链中不是必需的。本书不对联盟链中币或 Token 如何建立经济模型和高效使用，甚至是增值进行说明，仅从技术维度实现币或 Token 的相关逻辑。

应用层主要是联盟链中各个产品的落地。一般联盟链的应用层都是面向行业的，解决行业内的问题。

Java 版联盟链的部署架构如图 2-5 所示。

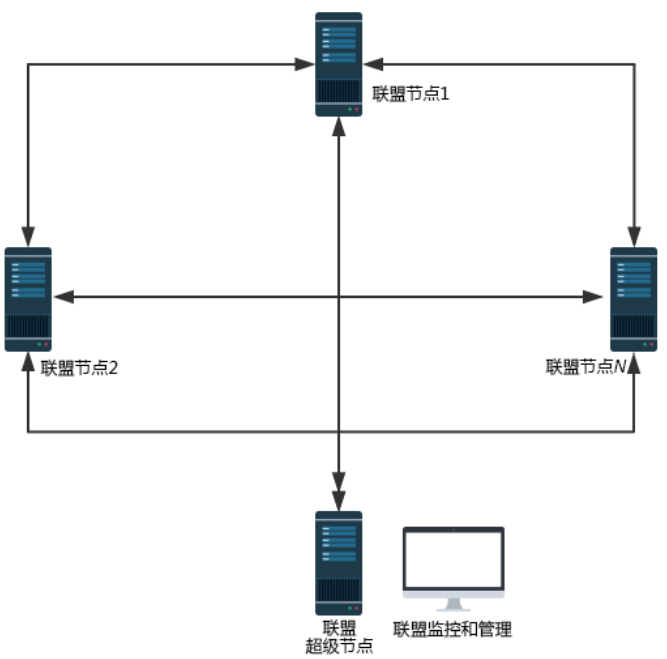


图 2-5　Java 版联盟链的部署架构

联盟链由 1 个超级节点和若干个普通节点组成，超级节点除具备普通节点的功能外，还具备在联盟中实施成员管理、权限管理、数据监控等工作。因此相较于完全去中心化的公链，联盟链是部分去中心化的，或者说联盟的“链”是去中心化的，但是联盟链的管理是中心化的。

整个开发环境建议基于 Spring Boot 2.0 实现。基于 Spring Boot 开发，可以省去大量的 xml 配置文件的编写，能极大简化工程中在 POM 文件配置的复杂依赖。Spring Boot 还提供了各种 starter，可以实现自动化配置，提高开发效率。

2.5　小结

本章介绍了比特币、以太坊、Hyperledger 的架构设计，并归纳了通用联盟链底层架构设计方案。后续章节将对每一部分的实现进行理论阐述和 Java 实操介绍。

第 3 章
密码学

九层之台，起于累土
千里之行，始于足下

密码在日常生活中屡见不鲜，购物支付用支付密码、在 ATM 机取款要用取款密码、手机屏幕解锁要用解锁密码，等等。

密码学一词源自希腊文 kryptos 及 logos，在希腊语中意为隐藏及消息。世界上最早的密码是在公元前 405 年，古希腊雅典和斯巴达之间的伯罗奔尼撒战争末期。在斯巴达军队准备对雅典发动最后一击之际，战前与斯巴达联盟的波斯帝国突然准备反戈一击。为此斯巴达急需摸清波斯帝国的行动部署。恰巧，斯巴达军队捕获了一名从波斯帝国回雅典送信的雅典信使。从信使身上搜出一条布满杂乱无章的希腊字母的腰带，斯巴达军队统帅莱桑德无意中把腰带缠绕在手中的剑鞘上时，竟然发现腰带上那些杂乱无章的字母组成了一段文字。这便是雅典间谍送回的情报，原来波斯军队准备在斯巴达军队发起最后攻击时，对斯巴达军队进行突袭。斯巴达军队根据这份情报马上改变了作战计划，以迅雷不及掩耳之势击溃了毫无戒备的波斯军队，从而解除了后顾之忧。随后，斯巴达军队回师征伐雅典，最后赢得了战争的胜利。

而在我国古代，藏头诗可谓是密码学的另一种浪漫应用了。

时代车轮滚滚向前，密码学的发展也蒸蒸日上。随着现代信息社会的到来，密码学的作用也愈发重要。特别是很多信息都必须经过加密之后才能在互联网上传送，这都离不开现代密码技术。现代密码技术在信息加密、信息认证、数字签名和密钥管理方面都有很多应用。

区块链技术也离不开密码学，可以说密码学是区块链系统的基石之一。

从本章起，我们将逐步介绍区块链系统的各个核心模块的实现逻辑。

3.1　加密与解密

3.1.1　加密与解密简介

加密与解密技术是对信息进行编码和解码的技术。编码的过程即加密的过程，加密模块把可读信息（即明文）处理成代码形式（即密文）。解码的过程即解密的过程，解密模块把代码形式（即密文）转换回可读信息（即明文）。在加密和解密的过程中，密钥是非常关键的角色。

目前，加密技术主要分为对称加密、不对称加密和不可逆加密三类算法。

对称加密算法是应用较早的加密算法，技术十分成熟。在对称加密算法中，加密和解密的密钥相同。对称密钥技术有两种基本类型：分组密码和序列密码。

对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高。不足之处是，交易双方使用相同的密钥，安全性得不到保证。目前，广泛使用的对称加密算法有 DES、3DES、IDEA 和 AES 等，其中，美国国家标准局倡导的 AES 即将作为新标准取代 DES。

不对称加密算法使用两把完全不同但又完全匹配的一对钥匙，称之为公钥和私钥，公钥和私钥成对配合使用。目前，广泛应用的不对称加密算法有 RSA 算法和美国国家标准局提出的 DSA。我们常见的数字签名（Digital Signature）技术就是以不对称加密算法为基础的。虽然不对称加密算法的安全性得到了提高，但相较于对称加密算法，加密速度慢、效率低。

不可逆加密算法的使用和上述两类算法略有不同，加密过程中不需要使用密钥，输入明文后由系统直接经过加密算法处理成密文，加密后的数据是无法被解密的，只有重新输入明文，并再次经过同样不可逆的加密算法处理，得到相同的加密密文并被系统重新识别后，才能真正解密。目前，应用较多的不可逆加密算法有 RSA 公司发明的 MD5 算法和由美国国家标准局建议的不可逆加密标准 SHS（Secure Hash Standard，安全 Hash 标准）等。

3.1.2 Java 实现

我们既可以基于 Java 原生实现加密和解密，又可以基于第三方的工具包实现。

下面我们首先介绍基于 Java 原生 API 的实现方法，接着介绍第三方工具包 hutool 和 Tink 的加密解密 API 使用方法。

1. Cipher 类

Java 中的 Cipher 类主要提供加密和解密的功能，该类位于 javax.crypto 包下，声明为 public class Cipher extends Object，它构成了 Java Cryptographic Extension (JCE) 框架的核心。

使用 Cipher 类时，需构建 Cipher 对象，再调用 Cipher 的 getInstance 方法来实现。需要注意的是，这里可以由用户自定义传参。在 Cipher 的概念中称之为“转换”，“转换”用于描述生成对象使用的算法，其格式如下：

“算法/模式/填充”或“算法”

例如：Cipher c = Cipher.getInstance("*DES/CBC/PKCS5Padding*");

Cipher 类中常用的常量字段主要有：

- public static final int ENCRYPT_MODE：用于将 Cipher 初始化为加密模式的常量。
- public static final int DECRYPT_MODE：用于将 Cipher 初始化为解密模式的常量。
- public static final int WRAP_MODE：用于将 Cipher 初始化为密钥包装模式的常量。
- public static final int UNWRAP_MODE：用于将 Cipher 初始化为密钥解包模式的常量。
- public static final int PUBLIC_KEY：用于表示要解包的密钥为“公钥”的常量。
- public static final int PRIVATE_KEY：用于表示要解包的密钥为“私钥”的常量。
- public static final int SECRET_KEY：用于表示要解包的密钥为“秘密密钥”的常量。

Cipher 类中提供了构造方法，供用户自定义使用，构造方法如下所示：

```
   protected Cipher(CipherSpi cipherSpi,Provider provider, String
transformation)
```

Cipher 类中的核心方法总结如下：

1）public static final Cipher getInstance(String transformation)

该方法返回实现指定转换的 Cipher 对象。transformation 为转换的名称，例如，*DES/CBC/PKCS5Padding*。

2）public static final Cipher getInstance(String transformation, String provider)

该方法用于返回实现指定转换的 Cipher 对象。

3）public static final Cipher getInstance(String transformation, Provider provider)

该方法同样用于返回实现指定转换的 Cipher 对象。

4）public final Provider getProvider()

该方法用于返回 Cipher 对象的提供者。

5）public final String getAlgorithm()

该方法用于返回此 Cipher 对象的算法名称。

6）public final int getBlockSize()

该方法用于返回块的大小（以字节为单位）。

7）public final int getOutputSize(int inputLen)

该方法根据给定的输入长度 inputLen（以字节为单位），返回保存下一个 update 或 doFinal 操作结果所需的输出缓冲区长度（以字节为单位）。

8）public final byte[] getIV()

该方法用于返回新缓冲区中的初始化向量 (IV)。

9）public final AlgorithmParameters getParameters()

该方法返回此 Cipher 使用的参数。返回的参数可能与初始化此 Cipher 所使用的参数相同。如果此 Cipher 需要算法参数，但却未使用任何参数进行初始化，则返回的参数将由默认值和底层 Cipher 实现所使用的随机参数值组成。

10）public final ExemptionMechanism getExemptionMechanism()

该方法返回此 Cipher 使用的豁免（exemption）机制对象。

11）public final void init(int opmode, Key key)

该方法用密钥初始化此 Cipher。

12）public final void init(int opmode, Key key, SecureRandom random)

该方法用密钥和随机源初始化此 Cipher。

13）public final void init(int opmode, Key key, AlgorithmParameterSpec params)

该方法用密钥和一组算法参数初始化此 Cipher。

14）public final void init(int opmode, Key key, AlgorithmParameterSpec params, SecureRandom random)

该方法用一个密钥、一组算法参数和一个随机源初始化此 Cipher。

15）public final void init(int opmode, Key key, AlgorithmParameters params)

该方法用密钥和一组算法参数初始化此 Cipher。

16）public final void init(int opmode, Key key, AlgorithmParameters params, SecureRandom random)

该方法用一个密钥、一组算法参数和一个随机源初始化此 Cipher。

17）public final void init(int opmode, Certificate certificate)

该方法用取自给定证书的公钥初始化此 Cipher。

18）public final void init(int opmode, Certificate certificate, SecureRandom random)

该方法用取自给定证书的公钥和随机源初始化此 Cipher。

19）public final byte[] update(byte[] input)

20）public final byte[] update(byte[] input, int inputOffset,int inputLen)

21）public final int update(byte[] input, int inputOffset, int inputLen, byte[] output)

22）public final int update(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)

23）public final int update(ByteBuffer input, ByteBuffer output)

上述几种方法用于大量数据需要进行加密或解密的场景，此时需将大量数据分

批次进行加密或解密，对每一个批次数据的加密或解密都需要调用 update 方法。

24 ） public final byte[] doFinal() throws IllegalBlockSizeException, BadPaddingException

该方法用于大量数据需要进行加密或解密的场景，此时需将大量数据分批次进行加密或解密，doFinal()方法用于完成多批次加密或解密的收尾工作。比如，待加密或解密的大量数据可以分为 9 份，每批次 4 份，则第三批次就剩 1 份，这 1 份数据由 doFinal()方法执行对应的加密或解密操作。

25）public final int doFinal(byte[] output, int outputOffset)

26）public final byte[] doFinal(byte[] input)

27）public final byte[] doFinal(byte[] input, int inputOffset, int inputLen)

28）public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output)

29）public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)

30）public final int doFinal(ByteBuffer input, ByteBuffer output)

上述几种方法用于大量数据和少量数据进行加密或解密的场景。少量数据时，按单一部分操作加密或解密数据；大量数据需要进行加密或解密的场景中，需将大量数据分批次进行加密或解密，doFinal()方法用于完成多批次加密或解密的收尾工作。比如，待加密或解密的大量数据可以分为 9 份，每批次 4 份，则第三批次就剩 1 份，这 1 份数据由 doFinal()方法执行对应的加密或解密操作。

31 ） public final byte[] wrap(Keykey) throws IllegalBlockSizeException, InvalidKeyException

该方法用于包装密钥。

32）public final Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm,int wrappedKeyType)throws InvalidKeyException,NoSuchAlgorithmException

该方法用于解包一个以前包装的密钥。

参数列表释义如下：

- wrappedKey：待解包的密钥。
- wrappedKeyAlgorithm：与此包装密钥关联的算法。
- wrappedKeyType：已包装密钥的类型。此类型必须为 SECRET_KEY、PRIVATE_KEY 或 PUBLIC_KEY 之一。

2. 基于 Cipher 实现加密和解密

在熟悉了 Cipher 的主要 API 后，我们基于 Cipher 编写加密和解密代码如下：

```
package com.niudong.demo.util;

import java.io.IOException;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;

import org.testng.util.Strings;

import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;

/**
 * 基于 Cipher 实现的加密和解密工具类
 *
 * @author 牛冬
 *
 */
public class DeEnCoderCipherUtil {
  // 加密、解密模式
  private final static String CIPHER_MODE = "DES";

  // DES 密钥
  public static String DEFAULT_DES_KEY =
        "区块链是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型
        应用模式。";
```

```
    /**
     * function 加密通用方法
     *
     * @param originalContent:明文
     * @param key 加密密钥
     * @return 密文
     */
    public static String encrypt(String originalContent, String key) {
    // 明文或加密密钥为空时
    if(Strings.isNullOrEmpty(originalContent) || Strings.isNullOrEmpty
       (key)){
      return null;
    }

    // 明文或加密密钥不为空时
    try {
      byte[] byteContent = encrypt(originalContent.getBytes(),
          key.getBytes());
      return new BASE64Encoder().encode(byteContent);
    } catch (Exception e) {
       e.printStackTrace();
    }
    return null;
  }

    /**
     * function 解密通用方法
     *
     * @param ciphertext 密文
     * @param key DES 解密密钥（同加密密钥）
     * @return 明文
     */
     public static String decrypt(String ciphertext, String key) {
      // 密文或加密密钥为空时
      if (Strings.isNullOrEmpty(ciphertext) || Strings.isNullOrEmpty(key)) {
          return null;
      }

  // 密文或加密密钥不为空时
  try {
      BASE64Decoder decoder = new BASE64Decoder();
      byte[] bufCiphertext = decoder.decodeBuffer(ciphertext);
```

```
      byte[] contentByte = decrypt(bufCiphertext, key.getBytes());
      return new String(contentByte);
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
    * function 字节加密方法
    *
    * @param originalContent：明文
    * @param key 加密密钥的 byte 数组
    * @return 密文的 byte 数组
    */
    private static byte[] encrypt(byte[] originalContent, byte[] key)
        throws Exception {
      // 步骤 1：生成可信任的随机数源
      SecureRandom secureRandom = new SecureRandom();

      // 步骤 2：基于密钥数据创建 DESKeySpec 对象
      DESKeySpec desKeySpec = new DESKeySpec(key);

      // 步骤 3：创建密钥工厂，将 DESKeySpec 转换成 SecretKey 对象来保存对称密钥
      SecretKeyFactory keyFactory=SecretKeyFactory.getInstance
          (CIPHER_MODE);
      SecretKey securekey = keyFactory.generateSecret(desKeySpec);

      // 步骤 4：Cipher 对象实际完成加密操作，指定其支持指定的加密和解密算法
      Cipher cipher = Cipher.getInstance(CIPHER_MODE);

      // 步骤 5：用密钥初始化 Cipher 对象，ENCRYPT_MODE 表示加密模式
      cipher.init(Cipher.ENCRYPT_MODE, securekey, secureRandom);

      // 返回密文
      return cipher.doFinal(originalContent);
    }

    /**
      * function 字节解密方法
      *
      * @param ciphertextByte:字节密文
```

```
     * @param key 解密密钥（同加密秘钥）byte 数组
     * @return 明文 byte 数组
     */
    private static byte[] decrypt(byte[] ciphertextByte, byte[] key)
        throws Exception {
      // 步骤 1：生成可信任的随机数源
      SecureRandom secureRandom = new SecureRandom();

      // 步骤 2：从原始密钥数据创建 DESKeySpec 对象
      DESKeySpec desKeySpec = new DESKeySpec(key);

      // 步骤 3：创建密钥工厂，将 DESKeySpec 转换成 SecretKey 对象来保存对称密钥
      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance
         (CIPHER_MODE);
      SecretKey securekey = keyFactory.generateSecret(desKeySpec);

      // 步骤 4：Cipher 对象实际完成解密操作，指定其支持响应的加密和解密算法
      Cipher cipher = Cipher.getInstance(CIPHER_MODE);

      // 步骤 5：用密钥初始化 Cipher 对象，DECRYPT_MODE 表示解密模式
      cipher.init(Cipher.DECRYPT_MODE, securekey, secureRandom);

      // 返回明文
      return cipher.doFinal(ciphertextByte);
    }


}
```

我们基于 TestNG 和 Mockito（使用说明详见附录 A 和附录 B）编写 DeEnCoderCipherUtil 对应的单元测试代码 DeEnCoderCipherUtilTest 如下：

```
package com.niudong.demo.util;

import org.testng.annotations.Test;
import org.testng.Assert;

/**
 * DeEnCoderCipherUtil 的单元测试类
 *
 * @author 牛冬
 *
```

```
 */
public class DeEnCoderCipherUtilTest {
  private static String ciphertextGlobal;

  @Test
  public void testEncrypt() {
    // case1:originalContent = null;key = null;
    String originalContent = null;
    String key = null;
    Assert.assertEquals(DeEnCoderCipherUtil.encrypt(originalContent,
        key), null);

    // case2:originalContent != null;key = null;
    originalContent = "2019 届校园招聘开启啦! ";
    key = null;
    Assert.assertEquals(DeEnCoderCipherUtil.encrypt(originalContent,
        key), null);

    // case3:originalContent = null;key != null;
    originalContent = null;
    key = " 2019 届校园招聘开启啦!内推简历扔过来呀! ";
    Assert.assertEquals(DeEnCoderCipherUtil.encrypt(originalContent,
        key), null);

    // case3:originalContent = null;key != null;
    originalContent = " 2019 届校园招聘开启啦! ";
    key = " 2019 届校园招聘开启啦!内推简历扔过来呀! ";
    ciphertextGlobal = DeEnCoderCipherUtil.encrypt(originalContent,
        key);
    Assert.assertEquals(ciphertextGlobal,
       " Jd/2DCl5MX6g8EKfqR/kGzy9OUSBxsfoQKMlpR3FCaE= ");
}
@Test(dependsOnMethods = {"testEncrypt"})
public void testDecrypt() {
  // case1:String ciphertext = null, String key =null
  String ciphertext = null, key = null;
  Assert.assertEquals(DeEnCoderCipherUtil.decrypt(ciphertext, key),
       null);

  // case2:String ciphertext != null, String key =null
  ciphertext = ciphertextGlobal;
  Assert.assertEquals(DeEnCoderCipherUtil.decrypt(ciphertext, key),
```

```
        null);


    // case3:String ciphertext = null, String key !=null
    ciphertext = null;
    key = " 2019 届校园招聘开启啦!内推简历扔过来呀! ";
    Assert.assertEquals(DeEnCoderCipherUtil.decrypt(ciphertext,
        key), null);

    // case4:String ciphertext != null, String key !=null
    ciphertext = ciphertextGlobal;
    key = " 2019 届校园招聘开启啦!内推简历扔过来呀! ";
    Assert.assertEquals(DeEnCoderCipherUtil.decrypt(ciphertext,
        key), "2019 届校园招聘开启啦! ");
  }
}
```

3. Hutool 简介

在 Java 世界中，AES、DES 的加密解密需要使用 Cipher 对象构建加密解密系统，有没有更好的封装工具类来简化开发呢？当然有！Hutool 就是一个好助手。

Hutool 是一个实用的 Java 工具包，有 pom.jar、javadoc.jar 和 sources.jar 等文件，可对文件、流、加密解密、转码、正则、线程、XML 等 JDK 方法进行封装，组成各种 Util 工具类。适用于 Web 开发，与其他框架无耦合，高度可替换。

在加密解密这部分，Hutool 针对 JDK 支持的所有对称加密算法都做了封装，封装为 SymmetricCrypto 类，AES 和 DES 是此类的简化表示。通过实例化这个类，传入相应的算法枚举即可使用相同方法加密解密字符串或对象。

当前 Hutool 支持的对称加密算法枚举有：

- AES
- ARCFOUR
- Blowfish
- DES
- DESede
- RC2

- PBEWithMD5AndDES
- PBEWithSHA1AndDESede
- PBEWithSHA1AndRC2_40

这些枚举全部在 SymmetricAlgorithm 中被列举。

对于不对称加密，最常用的就是 RSA 和 DSA，在 Hutool 中使用 AsymmetricCrypto 对象来负责加密解密。

4．基于 Hutool 实现加密解密

使用 Hutool 之前，在工程中需引入 Hutool 的依赖配置，配置代码如下所示：

```
<dependency>
   <groupId>com.xiaoleilu</groupId>
   <artifactId>hutool-all</artifactId>
   <version>3.0.9</version>
</dependency>
```

基于 Hutool 工具类的加密解密类 DeEnCoderHutoolUtil 的代码如下：

```
package com.niudong.demo.util;

import java.security.PrivateKey;
import java.security.PublicKey;
import org.testng.util.Strings;

import cn.hutool.core.util.CharsetUtil;
import cn.hutool.core.util.StrUtil;
import cn.hutool.crypto.SecureUtil;
import cn.hutool.crypto.asymmetric.KeyType;
import cn.hutool.crypto.asymmetric.RSA;
import cn.hutool.crypto.symmetric.AES;
import cn.hutool.crypto.symmetric.DES;

/**
 * 基于 Hutool 工具类的加密解密类
 *
 * @author 牛冬
 *
 */
```

```
public class DeEnCoderHutoolUtil {

  // 构建 RSA 对象
  private static RSA rsa = new RSA();
  // 获得私钥
  private static PrivateKey privateKey = rsa.getPrivateKey();
  // 获得公钥
  private static PublicKey publicKey = rsa.getPublicKey();


  /**
    * function RSA 加密通用方法:对称加密解密
    *
    * @param originalContent:明文
    * @return 密文
    */
  public static String rsaEncrypt(String originalContent) {
    // 明文或加密密钥为空时
    if (Strings.isNullOrEmpty(originalContent)) {
       return null;
    }

    // 公钥加密，之后私钥解密
    return rsa.encryptBase64(originalContent, KeyType.PublicKey);
  }

  /**
    * function RSA 解密通用方法:对称加密解密
    *
    * @param ciphertext 密文
    * @param key RSA 解密密钥（同加密密钥）
    * @return 明文
    */
  public static String rsaDecrypt(String ciphertext) {
    // 密文或加密密钥为空时
    if (Strings.isNullOrEmpty(ciphertext)) {
      return null;
    }

    return  rsa.decryptStr(ciphertext, KeyType.PrivateKey);
  }
  /**
```

```
    * function DES 加密通用方法:对称加密解密
    *
    * @param originalContent:明文
    * @param key 加密密钥
    * @return 密文
    */
  public static String desEncrypt(String originalContent, String key) {
    // 明文或加密密钥为空时
    if (Strings.isNullOrEmpty(originalContent) ||
        Strings.isNullOrEmpty(key)) {
      return null;
    }

    // 还可以随机生成密钥
    // byte[] key = SecureUtil.generateKey(SymmetricAlgorithm.DES.
    // getValue()).getEncoded();

    // 构建
    DES des = SecureUtil.des(key.getBytes());

    // 加密
    return des.encryptHex(originalContent);

  }
  /**
    * function DES 解密通用方法:对称加密解密
    *
    * @param ciphertext 密文
    * @param key DES 解密密钥（同加密秘钥）
    * @return 明文
    */
  public static String desDecrypt(String ciphertext, String key) {
    // 密文或加密密钥为空时
    if(Strings.isNullOrEmpty(ciphertext)||Strings.isNullOrEmpty(key)){
        return null;
    }

    // 还可以随机生成密钥
    // byte[] key = SecureUtil.generateKey(SymmetricAlgorithm.DES.
    // getValue()).getEncoded();

    // 构建
```

```
    DES des = SecureUtil.des(key.getBytes());
    // 解密

    return des.decryptStr(ciphertext);
  }


}
```

同样为了方便测试，我们基于 TestNG 和 Mockito 框架编写单元测试代码如下：

```
package com.niudong.demo.util;

import org.testng.Assert;
import org.testng.annotations.Test;

import cn.hutool.crypto.SecureUtil;
import cn.hutool.crypto.symmetric.SymmetricAlgorithm;

/**
 * 基于 Hutool 工具的加密解密的单元测试类
 *
 * @author 牛冬
 *
 */
public class DeEnCoderHutoolUtilTest {

  @Test
  public void testDesEncrypt() {
    // case1:String originalContent=null, String key=null
    String originalContent = null, key = null;
    Assert.assertEquals(DeEnCoderHutoolUtil.desEncrypt
        (originalContent, key), null);

    // case2:String originalContent!=null, String key=null
    originalContent = "2019 届校园招聘开启啦！ ";
    Assert.assertEquals(DeEnCoderHutoolUtil.desEncrypt
        (originalContent, key), null);

    // case2:String originalContent=null, String key!=null
    originalContent = null;
    key = " 2019 届校园招聘开启啦！内推简历扔过来呀！ ";
    Assert.assertEquals(DeEnCoderHutoolUtil.desEncrypt
```

```
        (originalContent, key), null);

    // case4:String originalContent!=null, String key!=null
    originalContent = "2019 届校园招聘开启啦!";
    key = new String(SecureUtil.generateKey(SymmetricAlgorithm.DES.
        getValue()).getEncoded());
    Assert.assertNotNull(DeEnCoderHutoolUtil.desEncrypt
        (originalContent, key));
  }

  @Test
  public void testDesDecrypt() {
    // case1:String ciphertext =null, String key = null
    String ciphertext = null, key = null;
    Assert.assertEquals(DeEnCoderHutoolUtil.desDecrypt(ciphertext,
        key), null);

    // case2:String ciphertext ! =null, String key = null
    String originalContent = "2019 届校园招聘开启啦!";
    String keyTmp =
         new String(SecureUtil.generateKey(SymmetricAlgorithm.DES.
              getValue()).getEncoded());
    ciphertext = DeEnCoderHutoolUtil.desEncrypt(originalContent,
        keyTmp);
    Assert.assertEquals(DeEnCoderHutoolUtil.desDecrypt(ciphertext,
        key), null);

    // case3:String ciphertext =null, String key != null
    ciphertext = null;
    key = new String(SecureUtil.generateKey(SymmetricAlgorithm.DES.
        getValue()).getEncoded());
Assert.assertEquals(DeEnCoderHutoolUtil.desDecrypt(ciphertext, key),
    null);

    // case4:String ciphertext !=null, String key != null
    ciphertext = DeEnCoderHutoolUtil.desEncrypt(originalContent,
        key);
    Assert.assertNotNull(DeEnCoderHutoolUtil.desDecrypt(ciphertext,
        key));
  }
}
```

5. Tink 简介

Tink 是由谷歌的一群密码学家和安全工程师编写的密码库。GitHub 开源地址：https://github.com/google/tink/。

Tink 的诞生融合了 Google 产品团队的丰富工程经验，现已修复了原有实现中的缺陷，并提供了简单的 API，用户可以安全使用，无须具备密码学知识背景。

Tink 提供了易于正确使用且不易被误用的安全 API。Tink 通过以用户为中心的设计、严谨的代码实现和代码审查以及广泛的测试，显著减少了工程开发中常见的密码陷阱。在谷歌，Tink 已经被用来保护许多产品的数据，如广告、谷歌薪酬、谷歌助理、Firebase、Android 搜索应用等。

使用 Tink 最简单的方法是安装 Bazel，然后构建、运行和播放 GitHub 中预设的 hello world 示例。

Tink 通过原语执行加密任务，每个原语都是通过指定原语功能的相应接口定义的。例如，对称密钥加密是通过 AEAD 原语（带有关联数据的认证加密）提供的，它支持两种操作：

1）加密（明文，associated _ data），加密给定明文（使用 associated _ data 作为额外的 AEAD 输入），并返回结果密文。

2）解密（密文，associated _ data），它解密给定的密文（使用 associated _ data 作为额外的 AEAD 输入），并返回结果明文。

目前，Tink 的 Java 语言版、Android 语言版、C++语言版和 Obj−C 语言版都已通过了严格的测试，并已投入生产部署。当前 Tink 的最新版本是 1.2.0，发布于 2018 年 8 月 9 日。此外，Tink 的 Go 语言版本和 JavaScript 语言版本正在积极开发中。

6. Tink 的使用

Tink 可以通过 Maven 或 Gradle 来实现依赖管理。在 Maven 中，Tink 的 group ID 是 com.google.crypto.tink，artifact ID 是 tink。

Java 开发人员基于 Maven 在工程的 POM 文件中添加 Tink 的依赖配置如下：

```
    <dependency>
      <groupId>com.google.crypto.tink</groupId>
      <artifactId>tink</artifactId>
      <version>1.2.0</version>
  </dependency>
```

1）Tink 的初始化

Tink 提供了可定制的初始化，允许用户选择所需原语的特定实现（由键类型标识）。Tink 的初始化是通过注册来实现的，以便 Tink“知道”用户期望的实现方式。

例如，要想使用 Tink 中的所有原语来实现初始化，则初始化的代码逻辑如下所示：

```
import com.google.crypto.tink.config.TinkConfig;

public void register() {
          try {
                TinkConfig.register();
                } catch (Exception e) {
                        e.printStackTrace();
                }
}
```

如果仅使用 AEAD 原语实现，则可以执行以下操作：

```
import com.google.crypto.tink.aead.AeadConfig;

public void register() {
          try {
                   AeadConfig.register();
                } catch (Exception e) {
                        e.printStackTrace();
                }
}
```

当然，Tink 还支持用户自定义初始化，即直接通过注册表类进行注册，代码如下所示：

```
import com.google.crypto.tink.Registry;
import com.niudong.demo.util.MyAeadKeyManager;

public void register(){
```

```
        // Register a custom implementation of AEAD.
        Registry.registerKeyManager(new MyAeadKeyManager());
    }
```

其中 MyAeadKeyManager 为自定义的 KeyManager。

注册原语实现后，Tink 的基本使用分三步进行：

（1）加载或生成加密密钥（Tink 术语中的密钥集）。

（2）使用 key 获取所选原语的实例。

（3）使用原语完成加密任务。

例如，使用 Java 中的 AEAD 原语加密解密时的代码示例如下：

```
        import com.google.crypto.tink.Aead;
        import com.google.crypto.tink.KeysetHandle;
        import com.google.crypto.tink.aead.AeadFactory;
        import com.google.crypto.tink.aead.AeadKeyTemplates;

    public void aead(byte[] plaintext, byte[] aad){
    try {
        // 1. 配置生成密钥集
        KeysetHandle keysetHandle = KeysetHandle.generateNew(
            AeadKeyTemplates.AES128_GCM);

        // 2. 使用 key 获取所选原语的实例
        Aead aead = AeadFactory.getPrimitive(keysetHandle);

        // 3. 使用原语完成加密任务
        byte[] ciphertext = aead.encrypt(plaintext, aad);
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
```

2）生成新密钥（组）

每个密钥管理器 KeyManager 的实现都提供了新密钥的生成接口 newKey(...)，该接口根据用户设置的密钥类型生成新密钥。然而，为了避免敏感密钥信息的意外泄漏，开发人员在代码中应该小心将密钥（集合）生成与密钥（集合）使用混合。为

了支持这些工作之间的分离，Tink 包提供了一个名为 Tinkey 的命令行工具，可用于公共密钥的管理。

如果用户需要在 Java 代码中直接用新的密钥生成 KeysetHandle，则用户可以使用 keysteadle 工具类。例如，用户可以生成包含随机生成的 AES 128 – GCM 密钥的密钥集，代码如下所示：

```
import com.google.crypto.tink.KeysetHandle;
import com.google.crypto.tink.aead.AeadKeyTemplates;

public void createKeySet(){
  try {
    KeyTemplate keyTemplate = AeadKeyTemplates.AES128_GCM;
    KeysetHandle keysetHandle = KeysetHandle.generateNew(keyTemplate);
  } catch (Exception e) {
        e.printStackTrace();
  }
}
```

3）存储密钥

生成密钥后，用户可以将其保存到存储系统中，例如写入文件，代码如下：

```
   import com.google.crypto.tink.CleartextKeysetHandle;
   import com.google.crypto.tink.KeysetHandle;
   import com.google.crypto.tink.aead.AeadKeyTemplates;
   import com.google.crypto.tink.JsonKeysetWriter;
   import java.io.File;

public void save2File(){
   try {
   // 创建 AES 对应的 keysetHandle
   KeysetHandle keysetHandle = KeysetHandle.generateNew(
      AeadKeyTemplates. AES128_GCM);

   // 写入 json 文件
   String keysetFilename = "my_keyset.json";
   CleartextKeysetHandle.write(keysetHandle,
   JsonKeysetWriter.withFile(
   new File(keysetFilename)));
   } catch (Exception e) {
            e.printStackTrace();
```

```
    }
}
```

用户还可以使用 Google Cloud KMS key 来对 key 加密，其中 Google Cloud KMS key 位于 gcp-kms:/projects/tink-examples/locations/global/keyRings/foo/cryptoKeys/bar as follows，保存在文件中的代码示例如下：

```
import com.google.crypto.tink.JsonKeysetWriter;
import com.google.crypto.tink.KeysetHandle;
import com.google.crypto.tink.aead.AeadKeyTemplates;
import com.google.crypto.tink.integration.gcpkms.GcpKmsClient;
import java.io.File;

public void save2FileBaseKMS(){
  try {
    // 创建 AES 对应的 keysetHandle
    KeysetHandle keysetHandle = KeysetHandle.generateNew(
        AeadKeyTemplates.AES128_GCM);

    // 写入 json 文件
    String keysetFilename = "my_keyset.json";
    // 使用 gcp-kms 方式对密钥加密
    String masterKeyUri = "gcp-kms:
    //projects/tink-examples/locations/global/keyRings/foo/
    cryptoKeys/bar";
    keysetHandle.write(JsonKeysetWriter.withFile(new
    File(keysetFilename)),
        new GcpKmsClient().getAead(masterKeyUri));
    } catch (Exception e) {
            e.printStackTrace();
    }
}
```

4）加载密钥

可以使用 KeysetHandle 加载加密的密钥集，代码示例如下：

```
    import com.google.crypto.tink.JsonKeysetReader;
    import com.google.crypto.tink.KeysetHandle;
    import com.google.crypto.tink.integration.awskms.AwsKmsClient;
    import java.io.File;
```

```
public void loadKeySet(){
   try {
      String keysetFilename = "my_keyset.json";
      // 使用 aws-kms 方式对密钥加密
      String masterKeyUri = "aws-kms:
      //arn:aws:kms:us-east-1:007084425826:key/84a65985-f868-4bfc-
      83c2-366618acf147";
      KeysetHandle keysetHandle = KeysetHandle.read(
        JsonKeysetReader.withFile(new File(keysetFilename)),
        new AwsKmsClient().getAead(masterKeyUri));
      } catch (Exception e) {
           e.printStackTrace();
      }
}
```

如果加载明文的密钥集，则需要使用 CleartextKeysetHandle 类：

```
    import com.google.crypto.tink.CleartextKeysetHandle;
    import com.google.crypto.tink.KeysetHandle;
    import java.io.File;

public void loadCleartextKeySet(){
  try {
    String keysetFilename = "my_keyset.json";
    KeysetHandle keysetHandle = CleartextKeysetHandle.read(
        JsonKeysetReader.withFile(new File(keysetFilename)));
  } catch (Exception e) {
            e.printStackTrace();
  }
}
```

5）原语的使用和获取

原语在 Tink 中指的是加密操作，因此它们构成了 Tink API 的核心。

原语是一个接口，它指定了原语能提供的基本操作。一个原语可以有多个实现，用户可以通过使用某种类型的键来设定想要的实现。

表 3-1 总结了当前可用或计划中的原语的 Java 实现。

表 3-1 原语的 Java 实现映射关系表

编号	原　　语	Java 实现
1	AEAD	AES-EAX, AES-GCM, AES-CTR-HMAC, KMS Envelope
2	Streaming AEAD	AES-GCM-HKDF-STREAMING, AES-CTR-HMAC-STREAMING
3	Deterministic AEAD	AES-SIV
4	MAC	HMAC-SHA2
5	Digital Signatures	ECDSA over NIST curves, ED25519
6	Hybrid Encryption	ECIES with AEAD and HKDF

6）对称密钥加密

获得和使用 AEAD（通过认证的加密，以及加密或解密数据）代码示例如下：

```
import com.google.crypto.tink.Aead;
import com.google.crypto.tink.KeysetHandle;
import com.google.crypto.tink.aead.AeadFactory;
import com.google.crypto.tink.aead.AeadKeyTemplates;

public void aeadAES(byte[] plaintext, byte[] aad){
  try {
    // 1. 创建 AES 对应的 keysetHandle
    KeysetHandle keysetHandle = KeysetHandle.generateNew(
AeadKeyTemplates.AES128_GCM);

    // 2. 获取私钥
    Aead aead = AeadFactory.getPrimitive(keysetHandle);

    // 3. 用私钥加密明文
    byte[] ciphertext = aead.encrypt(plaintext, aad);

    // 解密密文
    byte[] decrypted = aead.decrypt(ciphertext, aad);
  } catch (Exception e) {
        e.printStackTrace();
  }
}
```

7）数字签名

数字签名的签名或验证示例代码如下所示：

```
import com.google.crypto.tink.KeysetHandle;
import com.google.crypto.tink.PublicKeySign;
import com.google.crypto.tink.PublicKeyVerify;
import com.google.crypto.tink.signature.PublicKeySignFactory;
import com.google.crypto.tink.signature.PublicKeyVerifyFactory;
import com.google.crypto.tink.signature.SignatureKeyTemplates;

    // 签名
    public  void signatures(byte[] data){
    try{
    // 1. 创建 ESCSA 对应的 KeysetHandle 对象
    KeysetHandle privateKeysetHandle = KeysetHandle.generateNew(
        SignatureKeyTemplates.ECDSA_P256);

    // 2. 获取私钥
    PublicKeySign signer = PublicKeySignFactory.getPrimitive(
        privateKeysetHandle);

    // 3. 用私钥签名
    byte[] signature = signer.sign(data);

    // 签名验证

    // 1. 获取公钥对应的 KeysetHandle 对象
    KeysetHandle publicKeysetHandle =
        privateKeysetHandle.getPublicKeysetHandle();

   // 2. 获取私钥
   PublicKeyVerify verifier = PublicKeyVerifyFactory.getPrimitive(
        publicKeysetHandle);
    // 3. 使用私钥校验签名
   verifier.verify(signature, data);
  } catch (Exception e) {
        e.printStackTrace();
  }
}
```

使用 Tink 的注意事项如下：

（1）不要使用标有@ Alpha 注释的字段和方法的 API。这些 API 可以以任何方式修改，甚至可以随时删除。它们仅用于测试，不是官方生产发布的。

（2）不要在 com.google.crypto.tink.subtle 上使用 API。虽然这些 API 通常使用起来是安全的，但并不适合公众消费，因为可以随时以任何方式修改甚至删除它们。

3.2 哈希

3.2.1 散列函数简介

哈希函数也称为散列函数，在维基百科中的定义如下：

散列函数是可以将任意大小的数据映射到固定大小数据的任何函数。散列函数返回的值称为散列值、哈希代码、摘要或简单散列。散列函数通常与哈希表配合使用，哈希表是计算机软件中用于快速查找数据的常用数据结构。

作为加密算法的一种，散列函数是一种单向密码体制，即一个从明文到密文的不可逆映射，只有加密过程，没有解密过程。

散列函数的主要特点有：压缩性强、计算简单、计算结果单向性。压缩性强是指对于任意大小的输入内容，哈希值的长度很小，而且长度是固定位数的。计算简单是指对于任意给定的消息，计算其哈希值都比较简单。计算结果单向性是指对于给定的哈希值，无法倒推输入的原始数值。这也是散列函数安全性的重要基础。

当然，将无限内容压缩到有限位数的取值范围内，不可避免会出现不同的内容却有相同哈希值的碰撞结果。因此抗碰撞性是一个良好的散列函数的必备条件。

目前，哈希算法主要有两类：MD 系列和 SHA 系列。MD（Message Digest，消息摘要）系列有 MD4、MD5、HAVAL 等，SHA（Secure Hash Algorithm，安全散列算法）系列有 SHA1、SHA256 等。其中，MD5 是密码学专家 R.L.Rivest 设计的，SHA 是美国算法制定机构设计的。

常见的哈希算法有 SHA-1 算法、SHA-2 算法、SHA-3 算法，分别介绍如下。

1）SHA-1 算法

SHA-1 算法的输入是最大长度小于 264 位的消息，输入消息以 512 位的分组为单位进行处理，输出是 160 位的消息摘要。SHA-1 算法具有实现速度高、容易实现、

应用范围广等优点。

2）SHA-2 算法

SHA-2 算法的输出长度可取 224 位、256 位、384 位、512 位，分别对应 SHA-224、SHA-256、SHA-384、SHA-512。它还包含另外两个算法：SHA-512/224、SHA-512/256。比之前的哈希算法具有更强的安全强度和更灵活的输出长度，其中，SHA-256 是常用算法。SHA-256 算法的输入是最大长度小于 264 位的消息，输出是 256 位的消息摘要，输入消息以 512 位的分组为单位进行处理。

3）SHA-3 算法

美国政府选择 Keccak 算法作为 SHA-3 算法的加密标准，因为 Keccak 算法拥有良好的加密性能以及抗解密能力。

4）RIPEMD 算法

RIPEMD（RACE Integrity Primitives Evaluation Message Digest），即 RACE 原始完整性校验消息摘要。RIPEMD 使用 MD4 的设计原理，并针对 MD4 的算法缺陷进行了改进，1996 年首次发布 RIPEMD-128 版本，它在性能上与 SHA-1 算法类似。

上述算法中，SHA-1 算法已经破解，SHA-2 算法是使用范围较为广泛的算法，经常用于数字签名领域。对研发小伙伴而言，我们去开源网站下载源代码的过程中完整性校验就是基于哈希算法的。

在区块链系统中，目前使用的主流哈希算法是 SHA-256，下面将从实战的角度，为读者展示如何进行哈希计算。

3.2.2　SHA-256 Java 实战

我们可以用多种方法来实现用 Java 编写 SHA-256 的哈希计算接口。可以基于 Apache commons-codec 的工具类来实现 SHA-256 加密，也可以利用 Hutool 的工具类来实现 SHA-256 加密。

下面展示利用 Apache commons-codec 的工具类实现 SHA-256 加密的代码和利用 Hutool 的工具类实现 SHA-256 加密的代码，代码如下：

```
package com.niudong.demo.util;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import org.apache.commons.codec.binary.Hex;

import cn.hutool.crypto.digest.DigestUtil;

public class SHAUtil {

  /***
   * 利用 Apache commons-codec 的工具类实现 SHA-256 加密
   *
   * @param originalStr 加密前的报文
   * @return String 加密后的报文
   */
  public static String getSHA256BasedMD(String originalStr) {
    MessageDigest messageDigest;
    String encdeStr = "";
    try {
      messageDigest = MessageDigest.getInstance("SHA-256");
      byte[] hash = messageDigest.digest(originalStr.getBytes("UTF-8"));
      encdeStr = Hex.encodeHexString(hash);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    }
    return encdeStr;
  }

  /***
   * 利用 Hutool 的工具类实现 SHA-256 加密
   *
   * @param originalStr 加密前的报文
   * @return String 加密后的报文
   */
  public static String sha256BasedHutool(String originalStr) {
    return DigestUtil.sha256Hex(originalStr);
  }
```

```
    }
```

如上述代码所示，我们实现了 getSHA256BasedMD()和 sha256BasedHutool()方法，下面编写单元测试代码对两个方法进行测试，单元测试代码如下：

```
    package com.niudong.demo.util;

    import org.testng.Assert;
    import org.testng.annotations.Test;


    public class SHAUtilTest {
      /**
       * SHAUtil 测试类
       */

      // 测试 getSHA256 方法
      @Test
      public void testGetSHA256() {
        String originalStr = "区块链是分布式数据存储、点对点传输、共识机制、加密
          算法等计算机技术的新型应用模式。";

Assert.assertEquals("3c8fede03b42a9a3186fb96d7f22a1862bcb00e445e0b2956
6c613590306e3da",
            SHAUtil.getSHA256BasedMD(originalStr));
      }

      // 测试 getSHA256 方法
      @Test
      public void testSha256BasedHutool() {
        String originalStr = "区块链是分布式数据存储、点对点传输、共识机制、加密
          算法等计算机技术的新型应用模式。";

Assert.assertEquals("3c8fede03b42a9a3186fb96d7f22a1862bcb00e445e0b2956
6c613590306e3da",
            SHAUtil.sha256BasedHutool(originalStr));
      }
    }
```

在 IDE 窗口右击鼠标，在右键快捷菜单中单击“Run As”，在弹出的下拉列表中选择“TestNG test”，执行单元测试代码。测试结果输出如下：

```
PASSED: testGetSHA256
PASSED: testSha256BasedHutool

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.
    JUnitReportReporter@2d3fcdbd:  9 ms
[TestNG] Time taken by org.testng.reporters.
    EmailableReporter2@1936f0f5: 12 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@62043840: 66 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 0 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@3567135c: 10 ms
[TestNG] Time taken by
    org.testng.reporters.SuiteHTMLReporter@180bc464: 54 ms
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
```

3.3 Merkle 树

3.3.1 Merkle 树简介

Merkle 树是 Ralph Merkle 于 1979 提出并用自己名字命名的一种数据结构。

什么是 Merkle 树呢？维基百科中对 Merkle 树的定义如下：

> 在密码学和计算机科学中，哈希树或 Merkle 树是一种树，其中每个叶子节点都标记有数据块的哈希，而每个非叶子节点都标记有其子节点标签的加密哈希。Merkle 树允许对大型数据结构的内容进行有效、安全的验证，是散列列表和散列链的泛化。

Merkle 树的结构如图 3-1 所示。

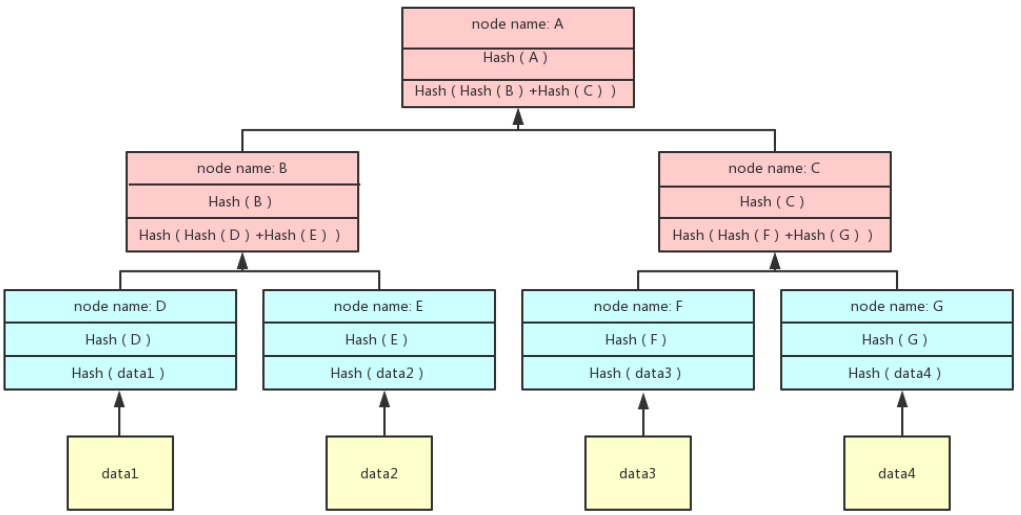


图 3-1　Merkle 树的结构

通过图 3-1 可以看到，数据 data1、data2、data3、data4 和其对应的 Merkle 树的映射关系。首先，对数据 data1、data2、data3、data4 计算哈希值，哈希计算方法可以基于 SHA-2 系列算法或 MD5，并将哈希值放进新生成的叶子节点。

从叶子节点开始逐级构建上一级节点的哈希值，直至构建到只剩一个根节点为止。上一级节点的哈希值依赖其左右孩子节点的哈希值，一般将这两个哈希组合后计算哈希值作为上一级节点的哈希值。

从上述 Merkle 树构建的过程不难发现：Merkle 树本质上就是一种树型结构，无论是二叉树还是多叉树。Merkle 树的叶子节点的 value 源于数据的哈希。非叶子节点的 value 是根据它左右孩子节点所有的叶子节点值，然后按照哈希算法计算而得出的。

目前，Merkle 树在数字货币、零知识证明、文件完整性校验等领域有广泛的应用，如在比特币和以太坊系统中利用 Merkle proofs 来存储每个区块的交易，而我们开发中常用的文件版本管理工具 Git 也是通过 Merkle 树来进行完整性校验的。

在区块链系统中，Merkle 树最常见的使用方式就是构建交易数据对应的 Merkle 树，并计算得到 Merkle 树根节点的哈希值。下面我们将展示如何计算 Merkle 树根节

点的哈希值及如何构建交易数据对应的 Merkle 树。

3.3.2 Merkle 树 Java 实战

在展示如何构建交易数据对应的 Merkle 树代码前，我们先对 Merkle 树根节点哈希值的计算进行一个简单的展示，代码如下：

```
package com.niudong.demo.util;

import java.util.ArrayList;
import java.util.List;

/**
 * 简化的 Merkle 树根节点哈希值计算
 *
 * @author 牛冬
 *
 */
public class SimpleMerkleTree {

  // 按 Merkle 树思想计算根节点哈希值
  public static String getTreeNodeHash(List<String> hashsList) {
    if (hashsList == null || hashsList.size() == 0) {
      return null;
    }

    while (hashsList.size() != 1) {
      hashsList = getMerKleNodeList(hashsList);
    }

    // 最终只剩下根节点
    return hashsList.get(0);
  }

  // 按 Merkle 树思想计算根节点哈希值
  public static List<String> getMerKleNodeList(List<String>
      contentList) {
    List<String> merKleNodeList = new ArrayList<String>();

    if (contentList == null || contentList.size() == 0) {
      return merKleNodeList;
```

```
        }

        int index = 0, length = contentList.size();
        while (index < length) {
          // 获取左孩子节点数据
          String left = contentList.get(index++);

          // 获取右孩子节点数据
          String right = "";
          if (index < length) {
            right = contentList.get(index++);
          }

          // 计算左右孩子节点的父节点的哈希值
          String sha2HexValue = SHAUtil.sha256BasedHutool(left + right);
          merKleNodeList.add(sha2HexValue);
        }
        return merKleNodeList;
      }
    }
```

如上述代码所示，我们通过 SimpleMerkleTree 类的静态方法 getTreeNodeHash() 来计算 Merkle 树根节点哈希值。其中输入内容为字符串类型的 List hashsList，在检验 hashsList 的有效性后，开始依次取 List 相邻元素的值，拼接后通过 SHAUtil.sha256BasedHutool()计算哈希值。若 List 中的元素个数为奇数，则用最后一个元素和空字符串组合计算哈希值。

每一轮 List 元素两两计算完哈希值后，当前数据的上一级哈希节点就产生完毕了。然后重复刚才对 List 的遍历，再对新 List 进行类似处理，直至 List 中仅剩一个元素，此时该元素的哈希值就是 Merkle 树根节点的哈希值。

上述代码对应的单元测试类代码如下：

```
package com.niudong.demo.util;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.testng.Assert;
```

```
import org.testng.annotations.Test;

/**
 * SimpleMerkleTree 的单元测试类
 *
 * @author 牛冬
 *
 */
public class SimpleMerkleTreeTest {
  @Test
  public void testGetMerKleNodeList() {
    // case1:List<String> contentList = null;
    List<String> contentList = null;
    Assert.assertEquals(SimpleMerkleTree.getMerKleNodeList
        (contentList).size(), 0);

    // case2:List<String> contentList = new ArrayList<>();但无内容
    contentList = new ArrayList<>();
    Assert.assertEquals(SimpleMerkleTree.getMerKleNodeList
        (contentList).size(), 0);

    // case3:contentList 有内容填充
    contentList = Arrays.asList("区块链", "人工智能", "脑科学", "K12 教育
        全球优质公司");
    Assert.assertEquals(SimpleMerkleTree.getMerKleNodeList
         (contentList).size(), 2);
  }

  @Test
  public void testGetTreeNodeHash() {
    // case1:List<String> contentList = null;
    List<String> contentList = null;
    Assert.assertEquals(SimpleMerkleTree.getTreeNodeHash
         (contentList), null);

    // case2:List<String> contentList = new ArrayList<>();但无内容
    contentList = new ArrayList<>();
    Assert.assertEquals(SimpleMerkleTree.getTreeNodeHash
         (contentList), null);

    // case3:contentList 有内容填充
    contentList = Arrays.asList("区块链", "人工智能", "脑科学", "K12 教育
```

```
        全球优质公司");
    Assert.assertEquals(SimpleMerkleTree.getTreeNodeHash
        (contentList),
        "f0e7e2c8716a92d1fa9909149279ea2201d488cea0278ba23033b8aed13a667d");
  }
}
```

在 IDE 窗口右击鼠标，在右键快捷菜单中单击“Run As”，在弹出的下拉列表中选择“TestNG test”，执行单元测试代码。测试结果输出如下：

```
PASSED: testGetMerKleNodeList
PASSED: testGetTreeNodeHash

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by
    org.testng.reporters.JUnitReportReporter@2d3fcdbd: 17 ms
[TestNG] Time taken by
    org.testng.reporters.EmailableReporter2@1936f0f5: 7 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@62043840: 106 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]:
    1 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@3567135c: 24
    ms
[TestNG] Time taken by
    org.testng.reporters.SuiteHTMLReporter@180bc464: 47 ms
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
```

上面的代码展示了如何利用 Merkle 树的原理来计算输入内容对应的 Merkle 树根节点哈希值，但实际在区块链系统中，不仅会用到根节点的哈希值，还需要基于 Merkle 树验证交易的有效性，因此，下面将展示如何构建真正意义上的 Merkle 树。

在构建 Merkle 树之前，我们先定义 Merkle 树中每个节点的数据结构如下：

```
package com.niudong.demo.util;

/**
 * 树节点定义
 *
 * @author 牛冬
 *
 */
public class TreeNode {
  // 二叉树的左孩子
  private TreeNode left;
  // 二叉树的右孩子
  private TreeNode right;
  // 二叉树中孩子节点的数据
  private String data;
  // 二叉树中孩子节点的数据对应的哈希值，此处采用 SHA-256 算法处理
  private String hash;
  // 节点名称
  private String name;


  // 构造函数 1
  public TreeNode() {

  }

  // 构造函数 2
  public TreeNode(String data) {
    this.data = data;
    this.hash = SHAUtil.sha256BasedHutool(data);
    this.name = "[节点：" + data+"]";
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getData() {
```

```
    return data;
  }

  public void setData(String data) {
    this.data = data;
  }

  public TreeNode getRight() {
    return right;
  }

  public void setRight(TreeNode right) {
    this.right = right;
  }

  public TreeNode getLeft() {
    return left;
  }

  public void setLeft(TreeNode left) {
    this.left = left;
  }

  public String getHash() {
    return hash;
  }

  public void setHash(String hash) {
    this.hash = hash;
  }
}
```

如上述代码所示，Merkle 树的中节点中包含左右孩子节点信息、自身的数据、基于左右孩子数据计算得到的哈希值和节点名称。

下面基于 TreeNode 构建 Merkle 树的代码如下：

```
package com.niudong.demo.util;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```
/**
 * Merkle 树构建、生成根节点哈希值的工具类
 *
 * @author 牛冬
 *
 */
public class MerkleTree {
  // TreeNode List
  private List<TreeNode> list;
  // 根节点
  private TreeNode root;

  // 构造函数
  public MerkleTree(List<String> contents) {
    createMerkleTree(contents);
  }

  // 构建 Merkle 树
  private void createMerkleTree(List<String> contents) {
    // 输入为空则不进行任何处理
    if (contents == null || contents.size() == 0) {
      return;
    }

    // 初始化
    list = new ArrayList<>();

    // 根据数据创建叶子节点
    List<TreeNode> leafList = createLeafList(contents);
    list.addAll(leafList);

    // 创建父节点
    List<TreeNode> parents = createParentList(leafList);
    list.addAll(parents);

    // 循环创建各级父节点直至根节点
    while (parents.size() > 1) {
      List<TreeNode> temp = createParentList(parents);
      list.addAll(temp);
      parents = temp;
    }
```

```
    root = parents.get(0);
  }

  // 创建父节点列表
  private List<TreeNode> createParentList(List<TreeNode> leafList) {
    List<TreeNode> parents = new ArrayList<TreeNode>();

    // 空检验
    if (leafList == null || leafList.size() == 0) {
      return parents;
    }

    int length = leafList.size();
    for (int i = 0; i < length - 1; i += 2) {
      TreeNode parent = createParentNode(leafList.get(i),
          leafList.get(i + 1));
      parents.add(parent);
    }

    // 奇数个节点时，单独处理最后一个节点
    if (length % 2 != 0) {
      TreeNode parent = createParentNode(leafList.get(length - 1),
          null);
      parents.add(parent);
    }

    return parents;
  }

  // 创建父节点
  private TreeNode createParentNode(TreeNode left, TreeNode right) {
    TreeNode parent = new TreeNode();

    parent.setLeft(left);
    parent.setRight(right);

    // 如果 right 为空，则父节点的哈希值为 left 的哈希值
    String hash = left.getHash();
    if (right != null) {
      hash = SHAUtil.sha256BasedHutool(left.getHash() +
          right.getHash());
```

```
        }
        // hash 字段和 data 字段同值
        parent.setData(hash);
        parent.setHash(hash);

        if (right != null) {
          parent.setName("(" + left.getName() + "和" + right.getName() +
              "的父节点)");
        } else {
          parent.setName("(继承节点{" + left.getName() + "}成为父节点)");
        }

        return parent;
      }

      // 构建叶子节点列表
      private List<TreeNode> createLeafList(List<String> contents) {
        List<TreeNode> leafList = new ArrayList<TreeNode>();

        // 空检验
        if (contents == null || contents.size() == 0) {
          return leafList;
        }

        for (String content : contents) {
          TreeNode node = new TreeNode(content);
          leafList.add(node);
        }

        return leafList;
      }

      // 遍历树
      public void traverseTreeNodes() {
        Collections.reverse(list);
        TreeNode root = list.get(0);
        traverseTreeNodes(root);
      }

      private void traverseTreeNodes(TreeNode node) {
        System.out.println(node.getName());
        if (node.getLeft() != null) {
```

```
      traverseTreeNodes(node.getLeft());
    }

    if (node.getRight() != null) {
      traverseTreeNodes(node.getRight());
    }
  }

  public List<TreeNode> getList() {
    if (list == null) {
      return list;
    }
    Collections.reverse(list);
    return list;
  }

  public void setList(List<TreeNode> list) {
    this.list = list;
  }

  public TreeNode getRoot() {
    return root;
  }

  public void setRoot(TreeNode root) {
    this.root = root;
  }

}
```

上述代码对应的单元测试代码如下：

```
package com.niudong.demo.util;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.testng.Assert;
import org.testng.annotations.Test;

/**
```

```
 * MerkleTree 单元测试类
 *
 * @author 牛冬
 *
 */
public class MerkleTreeTest {

  @Test
  public void testMerkleTree() {
    // case1:List<String> contents = null;
    List<String> contents = null;
    Assert.assertEquals(new MerkleTree(contents).getList(), null);
    Assert.assertEquals(new MerkleTree(contents).getRoot(), null);

    // case2:List<String> contents = new ArrayList<>();
    contents = new ArrayList<>();
    Assert.assertEquals(new MerkleTree(contents).getList(), null);
    Assert.assertEquals(new MerkleTree(contents).getRoot(), null);

    // case3:List<String> contents 有内容
    contents = Arrays.asList("区块链", "人工智能", "脑科学", "K12 教育全球
        优质公司");
    Assert.assertEquals(new MerkleTree(contents).getRoot().getHash(),
        " a1e908f0d5bd38569525c1a002af3fde5b2773d94514159f1bd2e37db59
          69cc4");
    Assert.assertEquals(new MerkleTree(contents).getRoot().getName(),
        " (([节点：区块链]和[节点：人工智能]的父节点)和([节点：脑科学]和[节点：
            K12 教育全球优质公司]的父节点)的父节点)");

    new MerkleTree(contents).traverseTreeNodes();
  }
}
```

在 IDE 窗口右击鼠标，在右键快捷菜单中单击“Run As”，在弹出的下拉列表中选择“TestNG test”，执行单元测试代码。测试结果输出如下：

(([节点：区块链]和[节点：人工智能]的父节点)和([节点：脑科学]和[节点：K12 教育全球优质公司]的父节点)的父节点)

([节点：区块链]和[节点：人工智能]的父节点)

[节点：区块链]

[节点：人工智能]

([节点：脑科学]和[节点：K12 教育全球优质公司]的父节点)

[节点：脑科学]

[节点：K12 教育全球优质公司]

PASSED: testMerkleTree

```
===============================================
   Default test
   Tests run: 1, Failures: 0, Skips: 0
===============================================



===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.
     JUnitReportReporter@2d3fcdbd: 9 ms
[TestNG] Time taken by org.testng.reporters.
     EmailableReporter2@1936f0f5: 6 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@62043840: 52 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 0 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@3567135c: 8 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@180bc464:
     29 ms
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
```

3.4　小结

本章主要从理论角度解释了区块链中使用的密码学的几个分支，如加密、哈希、Merkle 树，并从 Java 实战的角度为读者展示了如何用代码实现加密、哈希和 Merkle 树。

第 4 章
P2P 网络构建

叫嚣乎东西
隳突乎南北

第 3 章讲述了区块链系统中的基石之一密码学在区块链系统中的应用，本章将介绍区块链系统的另一个基石——P2P 网络的构建。

在常见的分布式服务中，实现不同节点间的高效通信是一项基础技术。同理，区块链系统作为分布式服务的一种，不同区块链节点间的高效通信同样非常重要。P2P 通信网络是区块链系统中数据传输、共识达成一致的基础。本章将从 P2P 的概念和历史发展讲起，逐步介绍比特币和以太坊中 P2P 网络的构建，最后重点介绍类似比特币的 P2P 网络构建方法。

4.1　P2P 简介

P2P 即 Peer-to-Peer，最早起源于 1997 年。这一年，Hotline Communications 公司研制了能让用户从别人电脑中下载内容的软件，这便是最早的 P2P。当时 P2P 网络一词的定义也与此相关，即 P2P 网络是一类允许一组用户互相连接并直接从用户硬盘上获取文件的网络。Hotline 也曾一度成为 P2P 网络的代名词，曾有文章以“*Hotline – The Glory Days Of P2P*”为题介绍当时的盛况。

随着互联网技术的发展，P2P 演化成了一种分布式网络。在分布式网络中，网络的各个节点，无论是机构还是个人，可以共享他们所拥有的一部分软、硬件资源，如数据处理能力、信息存储能力、互联网连接能力、打印机等。在 P2P 分布式网络中，这些共享资源能被其他对等节点（peer）直接访问，无须经过中间的服务器。

此网络中的每个节点都是双重身份，既是服务和内容等共享资源的提供者（server），又是服务和内容等资源的获取者（client）。

从上面的描述可以总结出 P2P 分布式网络的特点如下：

（1）无中央服务器，打破了 C/S 模式。

（2）用户之间互联并可以分享文件。

随着技术的发展，P2P 技术和 P2P 网络的好处（即资源能得到充分利用和最大化的共享）在不断涌现的大量应用中得以体现。对此，Microsoft 在“*Introduction to Windows Peer-to-Peer Networking*”一文中总结了部分应用情景，如即时通信应用，

包括生活中我们熟悉的 QQ、微信等场景。

此外，P2P 的应用还包括文件共享类的 P2P 网络服务，如 Napster、Gnutella、eDonkey、emule、BitTorrent 等；提供挖掘 P2P 对等计算能力和存储共享能力的应用，如 SETI@home、Avaki、Popular Power 等；基于 P2P 方式的协同处理与服务共享平台，如 JXTA、Magi、Groove 等。

在 P2P 分布式网络中，其核心在于数据存储在客户端本地，同时提供本地存储信息的查询服务，让网络中各个节点之间能直接进行数据传递，数据传递可以通过文件的名称、文件的地址等实现。

此外，P2P 分布式网络中，数据流量得到了分流，管理节点也不再有服务容量的压力，只需存储数据相关的索引与链接等内容即可，如 IPFS 文件系统就是基于 P2P 分布式网络的一种实现，在这个系统中可以百度搜索相关网址。

4.2 区块链 P2P 网络实现技术总结

在区块链领域，不同的公链系统所使用的 P2P 网络技术大不相同。国内迅雷发布的区块链文件系统 TCFS 是基于 IPFS 实现的，比特币系统的 P2P 网络是无结构的，而以太坊的 P2P 网络是有结构的。

在以太坊中，P2P 网络主要采用 Kademlia 算法实现，Kademlia 是分布式散列表（Distributed Hash Table，DHT）技术的一种。借助该技术，以太坊系统实现了在分布式环境下快速准确地路由和定位数据。

分布式散列表是一个由广域网内大量节点共同维护的巨大散列表。这张散列表被分割成不连续的块，每个节点被分配给一个属于自己的散列块，并成为这个散列块的管理者。DHT 的节点是动态的，在节点中，通过加密散列函数，对象的名字或关键词就可以被映射为 128 位或 160 位的散列值。目前，Tapestry、Chord、CAN 和 Pastry 都使用了分布式散列表技术。

当然，不同结构的 P2P 网络，有不同的优点和缺点。比特币系统的网络结构是最容易理解、最容易实现的一种形式，而以太坊网络引入了分布式散列表、异或距离、二叉前缀树、K-桶等，结构上更复杂，实现起来也更困难，但在节点路由上要

远快于比特币，本质上属于空间换时间的做法。

在本书中，我们会实现类似比特币系统中所用的网络结构。

网络编程可能是不少研发人员的弱项，甚至不少研发人员会感觉到陌生。不过，无须过多担心！在 Java 领域中，可用于实现网络编程或者远程通信的技术种类繁多，如传统技术中的 Socket 编程、RMI（Remote Method Invocation，远程方法调用）、RPC（Remote Procedure Call，远程过程调用）、Apache Mina（一个基于 TCP/IP、UDP/IP 协议栈的通信框架）、Hessian（一个轻量级的 remoting onhttp 工具）、JMS（Java Message Service，Java 消息服务）、Netty（JBOSS 提供的一个 Java 版网络服务器和客户端程序开源框架）、WebService（如 HTTP）等。

当然，除了这些基础技术，比较新的技术也是可以用于远程通信的。如 WebSocket、Vert.x、t-io、J-IM、http-kit、netty-socketio、async-http-client 等。其中，WebSocket 是 HTML5 中的新协议；Vert.x（网址：http://vertx.io/）是一个基于 JVM、轻量级、高性能的应用平台，基于全异步 Java 服务器 Netty；t-io 是基于 Java Aio 的网络编程框架，号称可以让天下没有难开发的网络编程；J-IM 是用 Java 编写的，基于 t-io 开发的轻量级、高性能网络框架。

以上这些技术可以在公链、联盟链、私链中构建节点间的通信。但在联盟链或私链的多个节点中，由于是部分去中心化的部署方式，因此技术选型的空间更多、更大。我们可以基于 MQ 或 Kafka 等中间件实现消息的发布订阅，即联盟链中各个节点可以共同订阅相同的 topic，节点需要达成共识或同步数据时，向该 topic 推送不同类型的消息即可，包括该节点在内的所有节点均可以通过订阅 topic 来实现消息的接收并进行相应处理。

在上述这些技术中，Socket 编程是最基础的 Java 技术，而 WebService（如 HTTP）、RPC（Remote Procedure Call，远程过程调用）的方式可能对分布式服务开发的小伙伴而言相对较为熟悉。一般，Web 服务器在普通阿里云或腾讯云服务器中可以支持上万个 HTTP 连接或 WebSocket 连接。因此在联盟链中使用 WebService 或 RPC 完全没有问题。

此时，在联盟链中，通过区块链底层管理后台，我们可以获取各个节点部署服务器的公网 IP。因此我们可以基于 HTTP 或 RPC（如 ProtoBuffer、Thrift）构建节点

的通信网络。即客户端发消息时，遍历所有节点公网 IP 列表，分别向这些服务器发送 HTTP 请求或 RPC 请求；同时，由于各个节点都部署了对应的服务端程序，因此可以对收到的 HTTP 请求或 RPC 请求进行处理。

当使用 HTTP 协议构建节点的通信网络时，还可以使用 HTTP 2.0 协议。2017 年，Java 9 和 Spring Boot 2.0 开始支持 HTTP 2.0 协议，Tomcat 9 于 2018 年 8 月 17 日也支持了 HTTP 2.0。HTTP 2.0 中的反向推送（Server Push）功能极大地简化了基于 HTTP 1.1 版本的开发工作量。

虽然 P2P 网络通信实现技术有所差异，但原理基本一致。本质上，P2P 网络中各节点的通信就是字节将流从一台服务器传送到另外一台服务器的过程，基于传输协议（如 HTTP、TCP、UDP 等协议）和网络 I/O 来实现。而 HTTP、TCP、UDP 其实都是基于 Socket 实现的。

由于 WebService、RPC、MQ 和 Kafka 等技术在分布式服务应用开发中很常见，因此本书不再过多阐述其用法。而在大部分分布式服务应用场景中鲜见使用 Socket、Mina、Netty 等技术，因此本书不会从最原始的 Socket 实现，也不会基于 Netty 实现，而是基于研发人员较为熟悉的、新的技术，如 WebSocket 技术，和相对不熟悉但很容易上手且应用场景较为广泛的 t-io 组件来实现，借此机会帮助读者拓展实用技术栈。

4.3 基于 WebSocket 构建 P2P 网络

4.3.1 WebSocket 介绍

WebSocket 是 HTML5 提出的一个协议规范。

WebSocket 约定了通信的规范，通过握手机制，客户端（如浏览器和服务器）之间建立一个类似 TCP 的长连接，从而方便客户端和服务端之间的通信，特别是服务端能主动向客户端推送消息。在 WebSocket 协议出现之前，Web 交互一般是基于 HTTP 的短连接或者长连接实现的，甚至是最原始的轮询或 Comet 方式。正是因为 WebSocket 实现了浏览器与服务器的全双工通信，才真正实现了 Web 的实时通信。

WebSocket 的工作流程大致如下：浏览器通过 JS 代码（JavaScript 代码）向服务端发出建立 WebSocket 连接的请求。在 WebSocket 连接建立成功后，客户端和服务端就可以通过 TCP 连接传输数据了。

下面将介绍基于 WebSocket 如何构建 P2P 网络。

4.3.2　基于 WebSocket 构建 P2P 网络

在常见的 Java Web 开发中，我们常使用 Spring Boot 框架，引入 spring-boot-starter-WebSocket 依赖，配置 ServerEndpointExporter 后，使用 @ServerEndpoint 创建 WebSocket 的 endpoint。

编写 endpoint 时，需要重写以下方法：连接建立成功调用的方法 onOpen()，连接关闭调用的方法 onClose()，收到客户端消息后调用的方法 onMessage()，发生错误时调用的方法 onError()。

在区块链系统中，每个节点既是服务端，又是客户端；既是消息的发出者，又是消息的接收者。与常见的 Java Web 开发中使用 WebSocket 不同，需要在服务端之间进行消息通信。因此，不能使用 spring-boot-starter-WebSocket 依赖。

在区块链系统中，我们需要借助 Java-WebSocket 来构建区块链系统中不同节点间的长连接，以便进行消息通信。在工程的 POM 文件中，我们配置 Java-WebSocket 依赖信息如下：

```
<dependency>
   <groupId>org.java-websocket</groupId>
   <artifactId>Java-WebSocket</artifactId>
   <version>1.3.8</version>
</dependency>
```

在 mvnresponsity 网站中可查得 Java-WebSocket 的最新版本为 1.3.8，如图 4-1 所示。

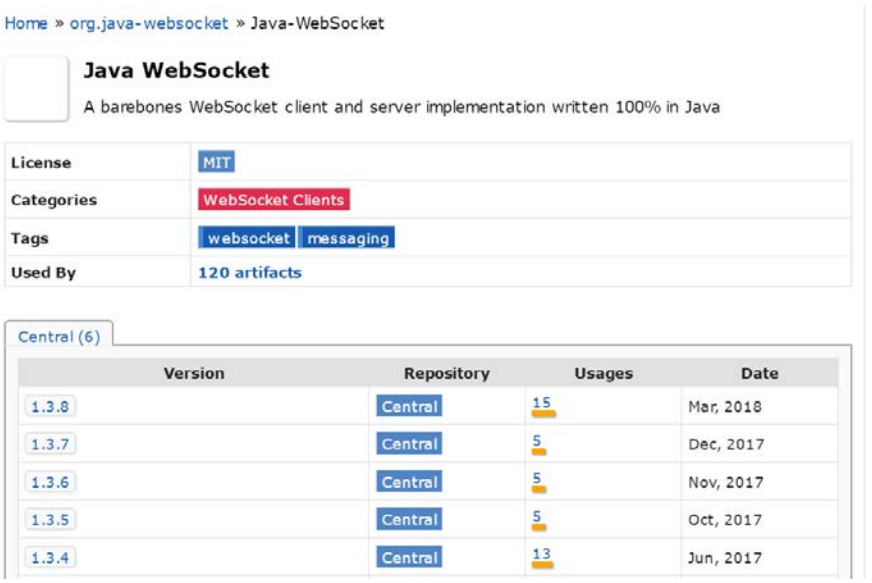


图 4-1　Java-WebSocket 版本列表

Java-WebSocket 是一个轻量级、100%用 Java 实现的 WebSocket 客户端和服务端工具包。因此我们将展示如何基于 Java-WebSocket 构建 P2P 网络。

下面将分别展示服务端和客户端的代码实现，服务端代码如下所示：

```
package com.niudong.demo.p2p;

import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;

import org.java_websocket.WebSocket;
import org.java_websocket.handshake.ClientHandshake;
import org.java_websocket.server.WebSocketServer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.testng.util.Strings;

/**
```

```
 * 基于 Spring Boot 2.0 的 WebSocket 服务端
 *
 * @author 牛冬
 *
 */
@Component
public class P2pPointServer {
  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(P2pPointServer.class);

  // 本机 Server 的 WebSocket 端口
  // 多机测试时可改变该值
  private int port = 7001;

  // 所有连接到服务端的 WebSocket 缓存器
  private List<WebSocket> localSockets = new ArrayList<WebSocket>();

  public List<WebSocket> getLocalSockets() {
    return localSockets;
  }

  public void setLocalSockets(List<WebSocket> localSockets) {
    this.localSockets = localSockets;
  }

  /**
   * 初始化 P2P Server 端
   *
   * @param Server 端的端口号 port
   */
  @PostConstruct
  @Order(1)
  public void initServer() {
    /**
     * 初始化 WebSocket 的服务端定义内部类对象 socketServer，源于
       WebSocketServer； new
     * InetSocketAddress(port)是 WebSocketServer 构造器的参数
       InetSocketAddress 是（IP 地址+端口号）类型，即端口地址类型
     */
    final WebSocketServer socketServer = new WebSocketServer(new
        InetSocketAddress(port)) {
```

```
        /**
         * 重写 5 个事件方法，事件发生时触发对应的方法
         */

        @Override
        // 创建连接成功时触发
        public void onOpen(WebSocket webSocket, ClientHandshake
            clientHandshake) {
          sendMessage(webSocket, "北京服务端成功创建连接");

          // 当成功创建一个 WebSocket 连接时，将该连接加入连接池
          localSockets.add(webSocket);
        }

        @Override
        // 断开连接时候触发
        public void onClose(WebSocket webSocket, int i, String s, boolean
           b) {
          logger.info(webSocket.getRemoteSocketAddress() + "客户端与服务器
              断开连接！");

          // 当客户端断开连接时，WebSocket 连接池删除该连接
          localSockets.remove(webSocket);
        }

        @Override
        // 收到客户端发来的消息时触发
        public void onMessage(WebSocket webSocket, String msg) {
          logger.info("北京服务端接收到客户端消息：" + msg);
          sendMessage(webSocket, "收到消息");
        }

        @Override
        // 连接发生错误时调用，紧接着触发 onClose 方法
        public void onError(WebSocket webSocket, Exception e) {
          logger.info(webSocket.getRemoteSocketAddress() + "客户端链接错
              误！");
          localSockets.remove(webSocket);
        }

        @Override
        public void onStart() {
```

```
            logger.info("北京的 WebSocket Server 端启动...");
          }
        };

        socketServer.start();
        logger.info("北京服务端监听 socketServer 端口：" + port);
      }

      /**
       * 向连接到本机的某客户端发送消息
       *
       * @param ws
       * @param message
       */
      public void sendMessage(WebSocket ws, String message) {
        logger.info("发送给" + ws.getRemoteSocketAddress().getPort() + "的
            p2p 消息是:" + message);
        ws.send(message);
      }

      /**
       * 向所有连接到本机的客户端广播消息
       *
       * @param message：待广播内容
       */
      public void broatcast(String message) {
        if (localSockets.size() == 0 || Strings.isNullOrEmpty(message)) {
          return;
        }

        logger.info("Glad to say broatcast to clients being startted!");
        for (WebSocket socket : localSockets) {
          this.sendMessage(socket, message);
        }
        logger.info("Glad to say broatcast to clients has overred!");
      }
    }
```

如上述代码所示，服务端用 initServer()方法初始化，默认打开本机 7001 端口作为服务端口。在 initServer()中重写 5 个事件方法，对应的事件发生时触发对应的方法。这 5 个方法主要有服务端启动时调用的方法 onStart()，创建连接成功时触发的方法

onOpen()，断开连接时触发的方法 onClose()，收到客户端发来的消息时触发的方法 onMessage()连接发生错误时调用的方法 onError()。onError()方法会在调用完毕后触发 onClose 方法，onMessage 收到消息后向客户端返回“收到消息”的消息。

为保证 initServer()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证服务端先于客户端加载，用@Order(1)标识了 initServer()。

此外，服务端代码中还实现了向连接到本机的某客户端发送消息的方法 sendMessage()和向所有连接到本机的客户端广播消息的方法 broatcast()。

客户端代码实现逻辑如下所示：

```
package com.niudong.demo.p2p;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;

import org.java_websocket.WebSocket;
import org.java_websocket.client.WebSocketClient;
import org.java_websocket.handshake.ServerHandshake;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.testng.util.Strings;



/**
 * 基于 Spring Boot 2.0 的 WebSocket 客户端
 *
 * @author 牛冬
 *
 */
@Component
```

```
public class P2pPointClient {
  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(P2pPointClient.class);

  // P2P 网络中的节既是服务端，又是客户端。作为服务端运行在 7001 端口
  //（P2pPointServer 的 port 字段），同时作为客户通过 ws://localhost:7001
  // 连接到服务端
  private String wsUrl = "ws://localhost:7001/";

  // 所有客户端 WebSocket 的连接池缓存
  private List<WebSocket> localSockets = new ArrayList<WebSocket>();


  public List<WebSocket> getLocalSockets() {
    return localSockets;
  }

  public void setLocalSockets(List<WebSocket> localSockets) {
    this.localSockets = localSockets;
  }

  /**
   * 连接到服务端
   */
  @PostConstruct
  @Order(2)
  public void connectPeer() {
    try {
      // 创建 WebSocket 的客户端
      final WebSocketClient socketClient = new WebSocketClient(new
          URI(wsUrl)) {
        @Override
        public void onOpen(ServerHandshake serverHandshake) {
          sendMessage(this, "北京客户端成功创建客户端");

          localSockets.add(this);
        }

        @Override
        public void onMessage(String msg) {
          logger.info("北京客户端收到北京服务端发送的消息:" + msg);
```

```
        }

        @Override
        public void onClose(int i, String msg, boolean b) {
          logger.info("北京客户端关闭");
          localSockets.remove(this);
        }

        @Override
        public void onError(Exception e) {
          logger.info("北京客户端报错");
          localSockets.remove(this);
        }
      };
      // 客户端开始连接服务端
      socketClient.connect();
    } catch (URISyntaxException e) {
      logger.info("北京连接错误:" + e.getMessage());
    }
  }

  /**
   * 向服务端发送消息，当前 WebSocket 的远程 Socket 地址就是服务端
   *
   * @param ws:
   * @param message
   */
  public void sendMessage(WebSocket ws, String message) {
    logger.info("发送给" + ws.getRemoteSocketAddress().getPort() + "的
        p2p 消息:" + message);
    ws.send(message);
  }

  /**
   * 向所有连接过的服务端广播消息
   *
   * @param message：待广播的消息
   */
  public void broatcast(String message) {
    if (localSockets.size() == 0 || Strings.isNullOrEmpty(message)) {
      return;
    }
```

```
        logger.info("Glad to say broatcast to servers being startted!");
        for (WebSocket socket : localSockets) {
          this.sendMessage(socket, message);
        }
        logger.info("Glad to say broatcast to servers has overred!");
      }
    }
```

如上述代码所示，客户端用 connectPeer()方法连接到服务端，在方法中，客户端通过 ws://localhost:7001 连接到服务端。与服务端类似，客户端在 connectPeer()中重写 5 个事件方法，对应的事件发生时触发对应的方法。这 5 个方法主要有客户端启动时调用的方法 onStart()、创建连接成功时触发的方法 onOpen()、断开连接时触发的方法 onClose()、收到服务端发来的消息时触发的方法 onMessage()、连接发生错误时调用的方法 onError()，onError()方法会在调用完毕后触发 onClose 方法，并且 onMessage 里不进行逻辑处理，仅仅打印日志信息。

为保证 connectPeer()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证客户端后于服务端加载，用@Order(2)标识了 connectPeer()。

此外，客户端端代码还实现了向服务端发送消息的方法 sendMessage()、向所有连接过的服务端广播消息的方法 broatcast()。

代码编写完成后，在名为 demo 的工程目录下运行 mvn clean package 命令，将 demo 工程打包成 demo.jar，随后切换到 target 目录，执行 java -jar demo.jar，得到结果如图 4-2 所示。

图 4-2　代码运行日志

日志前两行是节点作为 Sever 端输出，下边是节点作为客户端和服务端互相发送的消息。具体如下：

1）当 Client 端执行 connectPeer ()方法时，成功连接到 Server 端 onOpen 方法。

服务端向客户端发送消息如下：

发送给 59376 的 p2p 消息是：北京服务端成功创建连接

2）客户端收到服务端消息后，触发 onMessage 方法，打印日志如下：

北京客户端收到北京服务端发送的消息：北京服务端成功创建连接

随后客户端 onOpen 方法执行 sendMessage (this,“北京客户端成功创建客户端”)语句，发送消息到 Server 端，因此输出：

发送给 7001 的 p2p 消息：北京客户端成功创建客户端

3）Server 端收到 Client 端发送的客户端创建消息，触发 onMessage 方法，并回复 Client 端收到消息如下：

北京服务端接收到客户端消息：北京客户端成功创建客户端

随后调用 sendMessage()方法，发送给客户端消息如下：

发送给 59376 的 p2p 消息是：收到消息

4）Client 端收到服务端发送的消息，触发 onMessage()方法，打印日志如下：

北京客户端收到北京服务端发送的消息：收到消息

4.4 基于 t-io 构建 P2P 网络

4.4.1 t-io 介绍

t-io/tio 是一个网络编程框架，或称为 TCP 长连接框架，其官方网站中宣称，t-io/tio 不仅仅是百万级 TCP 长连接框架。

基于 t-io 可以开发 IM、TCP 私有协议、RPC、游戏服务器端、推送服务、实时监控、物联网、UDP、Socket 等将会变得空前的简单。

目前，t-io/tio 的代码已在码云的开源平台中开源。在 2017 年码云首批最有价值

开源项目评选中，t-io 上榜，码云平台为其颁发了证书，如图 4-3 所示。

图 4-3　码云最有价值开源项目证书证书

2017 年最受欢迎开源软件榜单中，同期上榜的还有 Hutool，POI 的封装工具类 EasyPoi，开源的互联网支付系统 roncoo-pay，强力 Java 爬虫工具 Spiderman，基于 Spring Cloud 微服务化开发平台 AG-Admin，爬虫框架 WebMagic，IP 到地名映射库 ip2region，轻量级分布式任务调度框架 xxl-job，中文的开源车牌识别系统 EasyPR，微信 Java 开发工具包 weixin-java-tools，秒杀、抢购解决方案 miaosha，号称分布式高效 ID 生产黑科技的 sequence 轻量级的关系型数据库中间件 Sharding-JDBC，500 行代码实现的极简、易用、高性能 AIO 框架 smart-socket 等。

那么 t-io 是什么样的工具类呢？按官网的介绍，我们总结如下：

t-io 一般是指 tio-core，它是基于 Java AIO 的网络编程框架，和 Netty 属于同类。

t-io 家族除 tio-core 外，还有 tio-websocket-server、tio-http-server、tio-webpack-core、tio-flash-policy-server 等，后面所列都是基于 tio-core 开发的应用层组件。下面分别介绍 t-io 家族的各个成员，其中：

- tio-core：基于 Java AIO 的网络编程框架。使用示例是 tio-showcase。
- tio-websocket-server：基于 tio-core 开发的 WebSocket 服务器。使用示例是 tio-websocket-showcase。
- tio-http-server：基于 tio-core 开发的 HTTP 服务器。使用示例是 tio-http-server-showcase
- tio-webpack-core：基于 tio-core 开发的 JS/CSS/HTML 编译压缩工具。
- tio-flash-policy-server：基于 tio-core 开发的 flash-policy-server。

目前应用 tio-core 技术的案例主要有：

IM：J-IM。

游戏服务器端：贝密游戏。

推送服务：牛吧云播。

物联网：氦氪云等。

4.4.2 t-io 的主要用法

下面介绍 t-io 常用类。t-io 常用类主要有 ChannelContext、GroupContext、AioHandler、AioListener、Packet、AioServer 和 AioClient。其中，ChannelContext 是通道上下文相关的类，GroupContext 用于服务配置与维护，AioHandler 是消息处理接口，AioListener 是通道监听者，Packet 是应用层的数据包，AioServer 是 t-io 服务端的入口类，AioClient 是 t-io 客户端的入口类，具体介绍如下。

1. ChannelContext（通道上下文）

每一个 TCP 连接的建立都会产生一个 ChannelContext 对象，这是一个抽象类，

如果用 t-io 作为 TCP 客户端，那么就是 ClientChannelContext。如果用为 t-io 作为 TCP 服务端，那么就是 ServerChannelContext。

客户端和服务端常用的 ChannelContext 如下：

1）ServerChannelContext

ChannelContext 的子类，当 t-io 作为 TCP 服务端时，业务层接触的是这个类的实例。

2）ClientChannelContext

ChannelContext 的子类，当 t-io 作为 TCP 客户端时，业务层接触的是这个类的实例。

2. GroupContext（服务配置与维护）

GroupContext 用来配置线程池，确定监听端口，维护客户端的各种数据等。GroupContext 是个抽象类，如果 t-io 作为 TCP 客户端，那么需要创建 ClientGroupContext。如果 t-io 作为 TCP 服务端，那么需要创建 ServerGroupContext。

其中：

1）ServerGroupContext

GroupContext 的子类，当 t-io 作为 TCP 服务端时，业务层接触的是这个类的实例。

2）ClientGroupContext

GroupContext 的子类，当 t-io 作为 TCP 客户端时，业务层接触的是这个类的实例。

3. AioHandler（消息处理接口）

AioHandler 是处理消息的核心接口，它有两个子接口：ClientAioHandler 和 ServerAioHandler。当 t-io 作为 TCP 客户端时，需要实现 ClientAioHandler。当 t-io 作为 TCP 服务端时，需要实现 ServerAioHandler。

AioHandler 主要定义了 3 个方法，decode()、encode()和 handler()，分别处理编码、解码和消息包。其中：

1）ServerAioHandler

AioHandler 的子类，当用 t-io 作为 TCP 服务端时，业务层实现该接口。

2）ClientAioHandler

AioHandler 的子类，当用 t-io 作为 TCP 客户端时，业务层实现该接口。

4. AioListener（通道监听者）

AioListener 是处理消息的核心接口，它有两个子接口，ClientAioListener 和 ServerAioListener。当 t-io 作为 TCP 客户端时，需要实现 ClientAioListener。当 t-io 作为 TCP 服务端时，需要实现 ServerAioListener。

AioListener 主要定义了如下方法：onAfterClose()、onAfterConnected()、OnAfterSend()、OnAfterReceived()和 OnBeforeClosed()。其中，onAfterClose()在连接关闭后触发，onAfterConnected()在连接建立后触发，OnAfterSend()在消息包发送之后触发，OnAfterReceived()在收到消息并解码成功后触发，OnBeforeClosed()在连接关闭前触发。

其中：

1）ServerAioListener

AioListener 的子类，当用 t-io 作为 TCP 服务端时，业务层实现该接口。

2）ClientAioListener

AioListener 的子类，当用 t-io 作为 TCP 客户端时，业务层实现该接口。

5. Packet（应用层数据包）

TCP 层过来的数据，都会按 t-io 要求解码成 Packet 对象，应用都需要继承这个类，从而实现自己的业务数据包。

6. AioServer（tio 服务端入口类）

AioServer 是 t-io 服务端入口类，在 start 方法中启动指定 IP 和端口的服务。

7. AioClient（tio 客户端入口类）

AioClient 是 t-io 客户端入口类，提供 connect()、asynconnect()方法族用于和服务端的同步和异步信息传输。

4.4.3　基于 t-io 构建 P2P 网络

在工程中使用 t-io 构建 P2P 网络时，需要引入 t-io 相关的依赖配置，配置代码如下：

```
<dependency>
   <groupId>org.t-io</groupId>
   <artifactId>tio-core</artifactId>
   <version>3.1.4.v20180726-RELEASE</version>
</dependency>
```

在工程中我们需要一些常量和定制化 Packet，因此先建立一个常量公共类如下所示：

```
package com.niudong.demo.tiop2p;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台所需的常量
 *
 * @author 牛冬
 *
 */
public class Const {
  /**
   * 服务器地址
   */
  public static final String SERVER = "127.0.0.1";

  /**
   * 监听端口
   */
  public static final int PORT = 6789;
```

```
    /**
     * 心跳超时时间
     */
    public static final int TIMEOUT = 5000;
}
```

在区块链系统中，我们把定制化的 Packet 命名为 BlockHelloPacket，代码如下所示：

```
package com.niudong.demo.tiop2p;

import org.tio.core.intf.Packet;

/**
 * 区块链底层定制的 Packet
 *
 * @author 牛冬
 *
 */
public class BlockHelloPacket extends Packet {
    // 网络传输需序列化，这里采用 Java 自带序列化方式
    private static final long serialVersionUID = -172060606924066412L;
    // 消息头的长度
    public static final int HEADER_LENGHT = 4;
    // 字符编码类型
    public static final String CHARSET = "utf-8";
    // 传输内容的字节
    private byte[] body;

    /**
     * @return the body
     */
    public byte[] getBody() {
        return body;
    }

    /**
     * @param body the body to set
     */
    public void setBody(byte[] body) {
        this.body = body;
```

```
  }
}
```

服务端相关的代码如下所示：

```
package com.niudong.demo.tiop2p;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.tio.core.ChannelContext;
import org.tio.core.GroupContext;
import org.tio.core.Tio;
import org.tio.core.exception.AioDecodeException;
import org.tio.core.intf.Packet;
import org.tio.server.intf.ServerAioHandler;

import java.nio.ByteBuffer;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的服务端 Handler
 * @author 牛冬
 *
 */
public class BlockChainServerAioHandler implements ServerAioHandler{

  //日志记录
  private Logger logger =
      LoggerFactory.getLogger(BlockChainServerAioHandler.class);

  /**
   * 解码：把接收到的 ByteBuffer 解码成应用可以识别的业务消息包
   * 总的消息结构：消息头 + 消息体
   * 消息头结构：4 个字节，存储消息体的长度
   * 消息体结构：对象的 JSON 串的 byte[]
   */
  @Override
  public BlockPacket decode(ByteBuffer buffer, int limit, int position,
      int readableLength, ChannelContext channelContext) throws
      AioDecodeException {
     //提醒：buffer 的开始位置并不一定是 0，应用需要从 buffer.position()开始
     //读取数据
     //若收到的数据无法组成业务包 BlockPacket，则返回 null 以表示数据长度不够
```

```
        if (readableLength < BlockPacket.HEADER_LENGHT) {
            return null;
        }

        //读取消息体的长度
        int bodyLength = buffer.getInt();

        //数据不正确，则抛出 AioDecodeException 异常
        if (bodyLength < 0) {
            throw new AioDecodeException("bodyLength [" + bodyLength + "]
                is not right, remote:" + channelContext.getClientNode());
        }

        //计算本次需要的数据长度
        int neededLength = BlockPacket.HEADER_LENGHT + bodyLength;
        //收到的数据是否足够组包
        int isDataEnough = readableLength - neededLength;
        // 不够消息体长度（剩下的 buffer 组不了消息体）
        if (isDataEnough < 0) {
            return null;
        } else //组包成功
        {
          BlockPacket imPacket = new BlockPacket();
            if (bodyLength > 0) {
                byte[] dst = new byte[bodyLength];
                buffer.get(dst);
                imPacket.setBody(dst);
            }
            return imPacket;
        }
    }

    /**
     * 编码：把业务消息包编码为可以发送的 ByteBuffer
     * 总的消息结构：消息头 + 消息体
     * 消息头结构：4 个字节，存储消息体的长度
     * 消息体结构：对象的 JSON 串的 byte[]
     */
    @Override
    public ByteBuffer encode(Packet packet, GroupContext groupContext,
        ChannelContext channelContext) {
      BlockPacket helloPacket = (BlockPacket) packet;
```

```
        byte[] body = helloPacket.getBody();
        int bodyLen = 0;
        if (body != null) {
            bodyLen = body.length;
        }

        //bytebuffer 的总长度是 = 消息头的长度 + 消息体的长度
        int allLen = BlockPacket.HEADER_LENGHT + bodyLen;
        //创建一个新的 bytebuffer
        ByteBuffer buffer = ByteBuffer.allocate(allLen);
        //设置字节序
        buffer.order(groupContext.getByteOrder());

        //写入消息头，消息头的内容就是消息体的长度
        buffer.putInt(bodyLen);

        //写入消息体
        if (body != null) {
            buffer.put(body);
        }
        return buffer;
    }


    /**
     * 处理消息
     */
    @Override
    public void handler(Packet packet, ChannelContext channelContext)
        throws Exception {
      BlockPacket helloPacket = (BlockPacket) packet;
        byte[] body = helloPacket.getBody();
        if (body != null) {
            String str = new String(body, BlockPacket.CHARSET);
            logger.info("北京服务端收到消息：" + str);

            BlockPacket resppacket = new BlockPacket();
            resppacket.setBody(("北京服务端收到了你的消息，你的消息是:" +
                str).getBytes(BlockPacket.CHARSET));
            Tio.send(channelContext, resppacket);
        }
        return;
```

```
  }
}

package com.niudong.demo.tiop2p;

import javax.annotation.PostConstruct;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.tio.server.ServerGroupContext;
import org.tio.server.TioServer;
import org.tio.server.intf.ServerAioHandler;
import org.tio.server.intf.ServerAioListener;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的服务端
 *
 * @author 牛冬
 *
 */
@Component
public class BlockChainServerStarter {
  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(BlockChainServerStarter.class);

  // handler，包括编码、解码、消息处理
  public static ServerAioHandler aioHandler = new
      BlockChainServerAioHandler();

  // 事件监听器，可以为 null，但建议自己实现该接口，可以参考 showcase 了解接口
  public static ServerAioListener aioListener = null;

  // 一组连接共用的上下文对象
  public static ServerGroupContext serverGroupContext =
     new ServerGroupContext("hello-tio-server", aioHandler,
        aioListener);

  // tioServer 对象
  public static TioServer tioServer = new
```

```
        TioServer(serverGroupContext);

    // 有时候需要绑定 IP，不需要则为 null
    public static String serverIp = null;//Const.SERVER;

    // 监听的端口
    public static int serverPort = Const.PORT;

    @PostConstruct
    @Order(1)
    public void start() {
      try {
        logger.info("北京服务端即将启动");

        serverGroupContext.setHeartbeatTimeout(Const.TIMEOUT);
        tioServer.start(serverIp, serverPort);

        logger.info("北京服务端启动完毕");
      } catch (Exception e) {
        logger.error(e.getMessage());
      }
    }
  }
```

如上述代码所示，基于 t-io 的区块链底层 P2P 网络平台的服务端类 BlockChainServerStarte 的 start()方法启动服务端时，主要逻辑是初始化 serverGroupContext，并通过 tioServer.start()方法启动指定的 IP 和端口。为保证 start()在服务启动时就能加载，用@PostConstruct 标记使其在服务端加载 bean 的时候运行，并且只会被服务端执行一次。同时，为了保证服务端先于客户端加载，用@Order(1)标识了 start()。

基于 t-io 的区块链底层 P2P 网络平台的服务端 Handle 的类 BlockChainServerAioHandler 提供了消息的解码方法 decode()、消息的编码方法 encode()及处理信息逻辑的 handler()方法。其中，handler()方法的逻辑是一边打印消息日志，一边返回客户端消息。

此外，由于是单机测试，特对判断已经在连接的节点个数的方法 getConnecttedNodeCount()和计算 pbft 消息节点最少确认个数的方法

getLeastNodeCount()进行硬编码，返回值均为 1。

客户端相关的代码如下所示：

```
package com.niudong.demo.tiop2p;

import java.nio.ByteBuffer;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.tio.client.intf.ClientAioHandler;
import org.tio.core.ChannelContext;
import org.tio.core.GroupContext;
import org.tio.core.exception.AioDecodeException;
import org.tio.core.intf.Packet;

/**
 * 基于 t-io 的区块链底 P2P 网络平台的客户端 Handler
 *
 * @author 牛冬
 *
 */
public class BlockChainClientAioHandler implements ClientAioHandler {

  // 日志记录
  private Logger logger =
LoggerFactory.getLogger(BlockChainClientAioHandler.class);

  private static BlockPacket heartbeatPacket = new BlockPacket();


  /**
   * 解码：把接收到的 ByteBuffer 解码成应用可以识别的业务消息包。总的消息结构为
     消息头 + 消息体。消息头结构：4 个字节，存储消息体的长度。消息体结构：对象的
     JSON 串的 byte[]
   */
  @Override
  public BlockPacket decode(ByteBuffer buffer, int limit, int position,
      int readableLength,
     ChannelContext channelContext) throws AioDecodeException {
    // 若收到的数据若无组成业务包，则返回 null，以表示数据长度不够
    if (readableLength < BlockPacket.HEADER_LENGHT) {
```

```
        return null;
      }

      // 读取消息体的长度
      int bodyLength = buffer.getInt();

      // 数据不正确，则抛出 AioDecodeException 异常
      if (bodyLength < 0) {
        throw new AioDecodeException(
            "bodyLength [" + bodyLength + "] is not right, remote:" +
                channelContext.getClientNode());
      }

      // 计算本次需要的数据长度
      int neededLength = BlockPacket.HEADER_LENGHT + bodyLength;
      // 收到的数据是否足够组包
      int isDataEnough = readableLength - neededLength;
      // 不够消息体长度（剩下的 buffer 组不了消息体）
      if (isDataEnough < 0) {
        return null;
      } else // 组包成功
      {
        BlockPacket imPacket = new BlockPacket();
        if (bodyLength > 0) {
          byte[] dst = new byte[bodyLength];
          buffer.get(dst);
          imPacket.setBody(dst);
        }
        return imPacket;
      }
    }

    /**
     * 编码：把业务消息包编码为可以发送的 ByteBuffer。总的消息结构：消息头 + 消息
       体。消息头结构：4 个字节，存储消息体的长度。消息体结构：对象的 JSON 串的 byte[]
     */
    @Override
    public ByteBuffer encode(Packet packet, GroupContext groupContext,
        ChannelContext channelContext) {
      BlockPacket helloPacket = (BlockPacket) packet;
      byte[] body = helloPacket.getBody();
      int bodyLen = 0;
```

```
    if (body != null) {
      bodyLen = body.length;
    }

    // ByteBuffer 的总长度 = 消息头的长度 + 消息体的长度
    int allLen = BlockPacket.HEADER_LENGHT + bodyLen;
    // 创建一个新的 ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate(allLen);
    // 设置字节序
    buffer.order(groupContext.getByteOrder());

    // 写入消息头，消息头的内容就是消息体的长度
    buffer.putInt(bodyLen);

    // 写入消息体
    if (body != null) {
      buffer.put(body);
    }
    return buffer;
  }

  /**
   * 处理消息
   */
  @Override
  public void handler(Packet packet, ChannelContext channelContext)
      throws Exception {
    BlockPacket helloPacket = (BlockPacket) packet;
    byte[] body = helloPacket.getBody();
    if (body != null) {
      String str = new String(body, BlockPacket.CHARSET);
      logger.info("北京客户端收到消息：" + str);
    }

    return;
  }

  /**
   * 此方法如果返回 null，则框架层面不会发心跳；如果返回非 null，则框架层面会定时
     发本方法返回的消息包
   */
  @Override
```

```
  public BlockPacket heartbeatPacket() {
    return heartbeatPacket;
  }
}

package com.niudong.demo.tiop2p;

import org.tio.client.TioClient;

import javax.annotation.PostConstruct;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.tio.client.ClientChannelContext;
import org.tio.client.ClientGroupContext;
import org.tio.client.ReconnConf;
import org.tio.client.intf.ClientAioHandler;
import org.tio.client.intf.ClientAioListener;
import org.tio.core.Tio;
import org.tio.core.Node;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的客户端
 *
 * @author 牛冬
 *
 */
@Component
public class BlockChainClientStarter {
  // 日志记录
  private Logger logger =
  LoggerFactory.getLogger(BlockChainClientStarter.class);

  // 服务端节点
  private Node serverNode;

  // handler，包括编码、解码、消息处理
  private ClientAioHandler tioClientHandler;

  // 事件监听器，可以为 null，但建议自己实现该接口，可以参考 showcase 了解接口
```

```
    private ClientAioListener aioListener = null;

    // 断链后自动连接，若不想自动连接请设为 null
    private ReconnConf reconnConf = new ReconnConf(5000L);

    // 一组连接共用的上下文对象
    private ClientGroupContext clientGroupContext;

    private TioClient tioClient = null;
    private ClientChannelContext clientChannelContext = null;

    /**
     * 启动程序入口
     */
    @PostConstruct
    @Order(2)
    public void start() {
      try {
        logger.info("北京客户端即将启动");

        //初始化
        serverNode = new Node(Const.SERVER, Const.PORT);
        tioClientHandler = new BlockChainClientAioHandler();
        clientGroupContext = new ClientGroupContext(tioClientHandler,
            aioListener, reconnConf);

        clientGroupContext.setHeartbeatTimeout(Const.TIMEOUT);
        tioClient = new TioClient(clientGroupContext);
        clientChannelContext = tioClient.connect(serverNode);

        // 连上后，发一条消息测试
        sendMessage();

        logger.info("北京客户端启动完毕");
      } catch (Exception e) {
        logger.error(e.getMessage());
      }
    }

    private void sendMessage() throws Exception {
      BlockPacket packet = new BlockPacket();
```

```
        packet.setBody("tal say hello world to
blockchain!".getBytes(BlockPacket.CHARSET));
        Tio.send(clientChannelContext, packet);
      }
    }
```

如上述代码所示，基于 t-BlockChainClientStarter 的 start()方法启动客户端，主要逻辑是初始化 clientGroupContext 和 tioClient，与服务端建立连接后，通过 sendMessage()方法向服务端发送一条消息“tal say hello world to blockchain!”。

为保证 start()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证客户端后于服务端加载，用@Order(2)标识了 start()。

基于 t-io 的区块链底层 P2P 网络平台的客户端 Handler 的类 BlockChainClientAioHandler 提供了消息的解码方法 decode()、消息的编码方法 encode()及处理信息逻辑的 handler()方法。其中，handler()方法的逻辑是仅仅打印消息，不进行任何处理。

在名为 demo 的工程根目录运行 maven 命令：mvn clean package 将 demo 工程打包成 demo.jar，随后切换到 target 目录下，执行命令 java -jar demo.jar，程序执行的效果如图 4-4 所示。

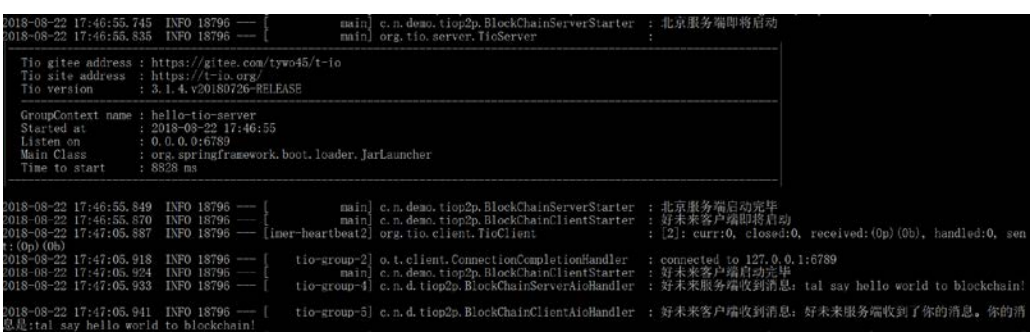


图 4-4　代码运行日志

如上述日志所示：

1）服务端先行启动，随后客户端启动。

2）客户端启动完毕后，通过调用 sendMessage 方法向服务端发送消息：tal say hello world to blockchain!

3）服务器 BlockChainServerAioHandler 收到消息后，经解码交由 handler 方法处理，并打印日志数据如下：

好未来客户端收到消息：好未来服务端收到了你的消息。你的消息是：tal say hello world to blockchain!

4.5 小结

本章从 P2P 的概念和历史发展讲起，逐步介绍了比特币和以太坊中的 P2P 网络构建，并总结了公链及联盟链中可以用于构建 P2P 网络通信的技术，最后重点介绍了比特币的 P2P 网络构建方法。本章主要向读者展示了基于 WebSocket 和 t-io 组件实现 P2P 网络的方法，读者可以根据自己的技术积累和认知去判断自己的区块链场景中需要使用何种技术实现区块链 P2P 网络的构建。笔者推荐用简单易上手的基于 WebSocket 方法来实现。

第 5 章
分布式一致性与共识算法

愿得一人心
白首不相离

前面讲述了区块链系统中的两大基石——密码学的应用和 P2P 网络的构建，本章将介绍区块链系统的另一个基石——共识算法。

什么是共识呢？通常的理解是共同的认识、一致的看法。在区块链系统中，共识指的是区块链系统中各个节点账本数据同步的实现。

共识算法并非凭空而生，而是基于分布式系统中为众人所知的分布式一致算法得来的。因此，本章将从简单的分布式一致算法开始，一步步引导读者学习共识算法。只要能掌握分布式一致算法的“美人心”，各类共识算法都会有似曾相识之感。

5.1 区块链的分布式

区块链系统本质上是分布式网络系统，而在分布式系统中，最核心的问题就是“一致性（Consistency）的问题”，即在分布式网络的各个节点中如何保持节点数据的一致性。如果在分布式系统中，一致性无法保证，那么分布式系统就变成了不可用的系统，因此保证一致性是分布式系统中最为重要的内容。

熟悉分布式网络的读者，看到一致性时，往往会自然联想到分布式领域的 CAP 理论。CAP 理论中论述了在任何一个分布式系统都无法同时满足 Consistency（一致性）、Availability（可用性）和 Partition tolerance（分区容错性）这三个基本需求。最多只能满足其中两项，而一致性是不能丢弃的。

当然，分布式系统中的分布式的一致性问题主要还是用来保证数据的一致性。

目前，各类分布式系统中使用的分布式一致性算法比较多，但基本都是源于 Paxos 分布式一致算法。正如 *Google Chubby* 的作者 Mike Burrows 所言：

“there is only one consensus protocol, and that’s Paxos – all other approaches are just broken versions of Paxos. ”

中文可翻译为：世上只有一种一致性算法，那就是 Paxos，所有其他一致性算法都是 Paxos 算法的不完整版。

因此分布式系统的一致性算法讲解将从 Paxos 算法开始。

5.2　Paxos 算法

Paxos 算法是莱斯利 • 兰伯特（Leslie Lamport，现就职于微软研究院）于 1990 年提出的，是一种基于消息传递的一致性算法。莱斯利 • 兰伯特于 2013 年获得了图灵奖。他的分布式计算理论奠定了这门学科的基础。

莱斯利 • 兰伯特在 1978 年发表了论文《分布式系统内的时间、时钟事件顺序》（*Time, Clocks, and the Ordering of Events in a Distributed System*），这篇论文成为目前计算机科学史上被引用最多的文献。他的论文为并发系统的规范与验证课题的研究贡献了核心原理。

Paxos 算法是在莱斯利 • 兰伯特的论文 *The Part-Time Parliament* 中提出的。在论文中，他以故事的方式讲述了 Paxos 算法。

这个故事的主要内容如下：

古希腊有一个叫 Paxos 的岛屿，是爱琴海上的一个小岛，Paxos 是一个兴盛的商业贸易中心。在这个岛屿上，法律的制定与修订通过议会表决的形式进行，而非传统的神权统治。

所有法律都必须经由议会成员投票表决后才能生效实施，而且已通过的律法必须被记录在案。

在岛上，商业繁荣，做生意赚钱才是头等大事，因此没有人愿意始终在议会大厅里从头到尾参与每一个法律表决的会议。为此，每一个议员都来维护一个法律律簿，用来记录一系列已通过的法令，每个法令带有一个唯一编号。为了保持各个议员法律律簿内容的一致性，法律律簿是用擦不掉的墨水书写而成的，所以内容一旦书写就不能改变。

在议会中有多个角色的成员：议员和服务员。服务员的工作是在比较嘈杂的议会厅里传递信息，议员的工作是发起法律提案或将通过的法律记录在自己的法律律簿上。

由于议员和服务员有可能并不可靠，他们可能随时会因为各种事情临时甚至是彻底离开议会大厅，服务员也有可能重复传递消息，当然也可能有新的议员在临时事务处理完毕后再回到议会大厅进行法律表决，因此议会的协议要求保证在上述情况下能够正确地修订法律并且不会产生冲突。

在法律表决时，议员的角色分为 proposers 和 acceptors。

通过一个法律决议时，分为两个阶段：

阶段 1：prepare 阶段

proposer 选择一个提案编号 *n*，并将 prepare 请求发送给 acceptors 群体。

acceptor 收到 prepare 消息后，如果提案的编号大于它已经回复的所有 prepare 消息，则 acceptor 将自己上次接受的提案回复给 proposer，并承诺不再回复小于 *n* 的提案；如果提案的编号小于等于它已经回复的所有 prepare 消息，则说明是重复消息，不再重复处理。

阶段 2：批准阶段

当 proposer 收到多数 acceptors 对 prepare 的回复后，就进入批准阶段。它要向回复 prepare 请求的 acceptors 发送 accept 请求。acceptor 收到 accept 请求后，则立即接受这个请求。

除经典的分布式一致性算法 Paxos 外，还有一些算法也隐藏于我们常用的中间件中，如 ZooKeeper，二、三阶段提交协议等。下面将分别介绍分布式一致性算法在 Zookeeper 和二、三阶段提交协议中的应用。

5.3 ZooKeeper 中的分布式一致算法实现

ZooKeeper 是一个开源的分布式协调服务，分布式应用程序可以基于它实现诸如数据发布/发布、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。ZooKeeper 的设计思想源于 Google 的 Chubby，目前

是 Hadoop 和 Hbase 中的重要组件。

ZooKeeper 使用了 ZAB（ZooKeeper Atomic Broadcast，ZooKeeper 原子消息广播）协议作为其数据一致性的核心算法。ZAB 协议是为 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。

基于 ZAB 协议，ZooKeeper 实现了在主/备模式下的系统架构中来保持集群中各副本之间的数据一致性。ZooKeeper 使用一个单一的主进程来接收并处理客户端的所有事务请求，并采用 ZAB 协议将服务器数据的状态变更以事务 Proposal 的形式广播到所有的副本进程中，ZAB 协议的主备模型架构保证了同一时刻集群中只能有一个主进程来广播服务器的状态变更，因此能够很好地处理客户端的大量并发请求。

ZAB 协议的核心是定义了能改变 ZooKeeper 服务器数据状态的事务请求的处理方式，即所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被称为 Leader 服务器，其他服务器则称为 Follower 服务器。Leader 服务器负责将一个客户端事务请求转化成一个事务 Proposal（提议），并将该 Proposal 分发给集群中所有的 Follower 服务器，之后 Leader 服务器需要等待所有 Follower 服务器的反馈。一旦超过半数的 Follower 服务器进行了正确的反馈后，那么 Leader 就会再次向所有的 Follower 服务器分发 Commit 消息，要求其提交前一个 Proposal。

在整个服务框架启动过程中，如果 Leader 服务器出现网络中断、崩溃退出与重启等异常情况，则 ZAB 协议就会进入恢复模式，并选举产生新的 Leader 服务器。当选举产生了新的 Leader 服务器，且集群中已经有过半的机器与该 Leader 服务器完成了状态同步之后，ZAB 协议就会退出恢复模式。状态同步是指数据同步，用来保证集群中过半的机器能够和 Leader 服务器的数据状态保持一致。

新 leader 的选举算法是分布式一致性算法的一种。在 ZooKeeper 中，提供了三种选举算法，分别是 LeaderElection、AuthFastLeaderElection 和 FastLeaderElection。默认的选举算法是 FastLeaderElection。FastLeaderElection 的 Leader 选举过程如下。

每个在 ZooKeeper 集群中的服务器都会读取各自服务器中当前保存在磁盘的数据，ZooKeeper 系统中的每份数据，都有一个唯一的 id 值。id 值是依次递增的，因此越新的数据，对应的 id 值就越大。

数据读取完毕之后，ZooKeeper 服务器就可以开启 Leader 选举了。通常每个 ZooKeeper 服务器发送自己选举的 Leader 信息中会包含以下几部分内容：

1）所选举 Leader 的 id。Leader 的 id 源于配置文件中配置的每个服务器的 id。在选举开始阶段，每台服务器都会选举自己为 Leader，这个值就是自己服务器的 id。

2）当前服务器中最大数据的 id。服务器中这个值越大，说明存放的数据越新。

3）逻辑时钟的值。逻辑时钟的值从 0 开始递增，每次选举对应一个。因此，在同一次选举中，逻辑时钟的值应该是一致的；若逻辑时钟的值不同，则逻辑时钟值越大，说明这次选举 Leader 的进程越新。

4）本机在当前选举过程中的状态。选举状态有 4 种类型，分别是：LOOKING、FOLLOWING、OBSERVING、LEADING。

每台服务器将自己服务器的状态数据发送到集群中的其他服务器之后，也会接收来自其他服务器的状态数据，并进行如下处理。

1）如果所接收数据服务器的状态是在选举阶段（即 LOOKING 状态），那么首先判断逻辑时钟的值，此时分为以下三种情况：

a）如果发送过来的逻辑时钟大于当前的逻辑时钟。

说明这是比当前保存的选举更新一次的选举，此时需要更新本机的逻辑时钟数据，同时将之前收集到的来自其他服务器的选举清空，因为这些数据已经不再有效了。

随后判断是否需要更新当前自己的选举情况。服务器根据选举 Leader id 保存的最大数据 id 来判断是否需要更新选举情况：

- 当数据 id 不同时，数据 id 大者胜出。
- 当数据 id 相同时，比较 Leader id 的值，Leader id 大者胜出。

随后再将自身最新的选举结果广播给集群中的其他服务器。

b）发送过来的逻辑时钟数据小于本机的逻辑时钟。

这说明对方在一个相对较早的选举进程中，属于重复数据，因此服务器无须处

理该部分数据，只需将本机的数据发送过去即可。

c）发送过来的逻辑时钟数据与本机的逻辑时钟数据相同。

此时调用 totalOrderPredicate 函数判断是否需要更新本机的数据，如果需要更新，则再将自己最新的选举结果广播给集群中的其他服务器。

以上三种情况处理完毕之后，需处理如下两种情况：

a）判断服务器是不是已经收集到了集群中所有服务器的选举状态。

如果是，则根据选举结果设置自己的角色（FOLLOWING 还是 LEADER），并退出选举过程。

b）若没有收集到所有服务器的选举状态。

则需要判断根据以上过程选举出的 Leader 是不是得到了超过半数以上服务器的支持。如果是，则尝试在 200ms 内接收数据。如果没有新的数据到来，则说明大家都已经默认了这个结果，设置自己的角色，退出选举过程。

2）如果服务器当前不处于选举状态，也就是在 FOLLOWING 或者 LEADING 状态，此时会作以下两个判断：

a）如果逻辑时钟相同，则将该数据保存到 recvset。如果所接收服务器宣称自己是 Leader，那么将判断是不是有半数以上的服务器选举它。如果是，则设置选举状态，退出选举过程。

b）如果逻辑时钟不同，则说明在另一个选举过程中已经有了选举结果，于是将该选举结果加入到 outofelection 集合中，再根据 outofelection 来判断是否可以结束选举。如果可以结束选举，则保存逻辑时钟，设置选举状态，退出选举过程。

5.4　二、三阶段提交协议

2PC（Two-Phase Commit，二阶段提交协议）和 3PC（Three-Phase Commit，三阶段提交协议）是经典的分布式一致性算法。下面分别介绍 2PC 和 3PC 的实现原理。

5.4.1 二阶段提交协议

二阶段提交协议，顾名思义就是将事务的提交过程分为两个阶段来进行。这两个阶段分别是准备阶段和提交阶段。在事务处理过程中，涉及两个角色，分别是协调者和参与者。其中，事务的发起者称为协调者，事务的执行者称为参与者。

第 1 阶段（即准备阶段）的逻辑如下：

1）协调者向所有参与者发送事务内容，询问各个参与者是否可以提交事务，并等待所有参与者答复。

2）各参与者执行事务操作，将 Undo 和 Redo 信息记入事务日志中（此时不提交事务）。

3）如过参与者执行成功，则给协调者反馈可以成功提交的结果；如果执行失败，则给协调者反馈提交失败的结果。

在第 2 阶段（即提交阶段）的逻辑如下。

协调者根据收到的各个参与者的反馈结果，进行不同的处理。此时若所有参与者均反馈可以成功提交结果，则开启事务的提交。此时的协调者和参与者的处理逻辑如下：

1）协调者向所有参与者发出开始提交事务的请求，即 Commit 请求。

2）参与者接收到 Commit 请求后，执行 Commit 请求。Commit 请求执行完毕后释放整个事务期间占用的资源。

3）参与者向协调者反馈 Commit 请求执行完成的消息。

4）协调者收到所有参与者反馈的消息后就完成了当前事务的提交。

当有任何一个参与者反馈提交失败的结果时，协调者就将事务中断。此时的协调者和参与者的处理逻辑如下：

1）协调者向所有参与者发出回滚请求，即 Rollback 请求。

2）参与者收到 Rollback 回滚请求后，使用第 1 阶段中的 Undo 信息执行回滚操作。Rollback 请求执行完毕后，同样需要释放整个事务期间占用的资源。

3）参与者向协调者反馈 Rollback 请求执行完成的消息。

4）协调者收到所有参与者反馈的消息后就完成当前事务的中断。

在整个提交过程中，协调者和参与者都是阻塞状态，同时协调者是一个单点的角色，一旦协调者宕机，二阶段提交协议则不能实现。因此改进版的三节点提交协议应运而生，该协议将在下一节介绍。

5.4.2　三阶段提交协议

三阶段提交协议，顾名思义就是分布式事务分三个阶段进行。这三个阶段分别是 CanCommit、PreCommit 和 DoCommit。下面分别介绍这三个阶段的处理逻辑。

第 1 阶段，即 CanCommit 阶段的处理逻辑如下。

1）协调者向各个参与者发送 Commit 请求。

2）参与者收到 Commit 请求后，判断是否可以提交。如果可以提交，则返回 Yes 响应，否则返回 No 响应。

回顾二阶段提交协议，不难发现三阶段提交协议中的 CanCommit 阶段与二阶段提交协议的准备阶段很相似。

第 2 阶段，即 PreCommit 阶段的处理逻辑如下。

协调者根据各个参与者的返回结果来决定是否要继续事务的 PreCommit 操作。综合各个参与者的返回结果，可以分为两类：

1）所有参与者返回的结果都是 Yes 响应。此时协调者会组织进行事务的预执行，即 PreCommit 操作。

a）协调者向所有参与者发送 PreCommit 请求后，进入 Prepared 阶段。

b）各个参与者接收到 PreCommit 请求后，会执行事务预提交的操作，并将 Undo 和 Redo 信息记录到事务日志中。当参与者成功执行了事务的预提交操作后，则向协调者返回 ACK 响应，同时开始等待最终指令。

2）若有任何一个参与者向协调者发送了 No 响应，或者协调者等待超时也没有

接到参与的响应，则中断当前的事务。中断事务的处理逻辑如下：

a）协调者向各个参与者发送中断请求，即 abort 请求。

b）参与者收到来自协调者的 abort 请求之后便开始执行事务的中断操作。

第三阶段，即 DoCommit 阶段。该阶段是真正的事务提交处理阶段，其处理逻辑如下。

1）当协调者收到所有参与者的 ACK 响应后，进入执行提交状态。此时的处理逻辑如下。

a）协调者向各个参与者发送提交请求，即 doCommit 请求，并进入提交状态。

b）参与者接收到协调者的 doCommit 请求之后，执行正式的事务提交。完成事务提交后，释放事务执行过程中占用的资源。

c）参与者提交完事务后，向协调者发送 ACK 响应。

d）协调者接收到所有参与者的 ACK 响应之后就完成了事务的提交工作。

2）当协调者并未收到参与者的 ACK 响应信息时，则启动事务的中断逻辑。

5.5 区块链中的分布式一致性

在区块链中，实现各个节点数据一致性的算法称之为共识算法。

区块链中的共识算法有工作量证明（PoW，Proof of Work）、权益证明（PoS，Proof of Stake）、延迟工作量证明（dPoW，Delayed Proof-of-Work）、授权 PoS（DPoS，Delegated Proof-of-Stake）、权威证明（PoA，Proof-of-Authority）、权重证明（PoWeight，Proof-of-Weight）、声誉证明（PoR，Proof of Reputation）、所用时间证明（PoET，Proof of Elapsed Time）、容量证明（PoC，Proof of Capacity）、历史证明（PoHistory，Proof of History）、权益流通证明（PoSV，Proof of Stake Velocity）、重要性证明（PoImportance，Proof of Importance）、拜占庭容错算法（Byzantine Fault Tolerance）、实用拜占庭容错算法（PBFT，Practical Byzantine Fault Tolerance）、授权拜占庭容错算法（dBFT，Delegated Byzantine Fault Tolerance）等。

其中比特币的 PoW 算法、PBFT（Practical Byzantine Fault Tolerance，实用拜占庭容错算法）和 PoS 算法，比较常见。下面分别介绍 PoW 算法在比特币和以太坊中的实现，并基于 PBFT 实现联盟链的共识算法。

5.5.1　PoW 算法

比特币系统和以太坊系统均基于 PoW 算法来实现其共识机制。下面分别介绍 PoW 共识算法在比特币系统和以太坊系统中的源码实现。

5.5.2　PoW 算法在比特币系统的源码实现

在比特币的开源实现中，PoW 算法在 mining.cpp 类中实现，其核心代码如下：

```
   UniValue generateBlocks(std::shared_ptr<CReserveScript>
coinbaseScript, int nGenerate, uint64_t nMaxTries, bool keepScript)
    {
        static const int nInnerLoopCount = 0x10000;
        int nHeightEnd = 0;
        int nHeight = 0;


        {   // Don't keep cs_main locked
            LOCK(cs_main);
            nHeight = chainActive.Height();
            nHeightEnd = nHeight+nGenerate;
        }
        unsigned int nExtraNonce = 0;
        UniValue blockHashes(UniValue::VARR);
        while (nHeight < nHeightEnd && !ShutdownRequested())
        {
            std::unique_ptr<CBlockTemplate>
               pblocktemplate(BlockAssembler(Params()).
               CreateNewBlock(coinbaseScript->reserveScript));
            if (!pblocktemplate.get())
                throw JSONRPCError(RPC_INTERNAL_ERROR, "Couldn't create new
                   block");
            CBlock *pblock = &pblocktemplate->block;
            {
                LOCK(cs_main);
```

```
            IncrementExtraNonce(pblock, chainActive.Tip(),
                nExtraNonce);
        }
        while (nMaxTries > 0 && pblock->nNonce < nInnerLoopCount
            && !CheckProofOfWork(pblock->GetHash(), pblock->nBits,
            Params().GetConsensus())) {
            ++pblock->nNonce;
            --nMaxTries;
        }
        if (nMaxTries == 0) {
            break;
        }
        if (pblock->nNonce == nInnerLoopCount) {
            continue;
        }
        std::shared_ptr<const CBlock> shared_pblock =
             std::make_shared<const CBlock>(*pblock);
        if (!ProcessNewBlock(Params(), shared_pblock, true, nullptr))
            throw JSONRPCError(RPC_INTERNAL_ERROR, "ProcessNewBlock,
                block not accepted");
        ++nHeight;
        blockHashes.push_back(pblock->GetHash().GetHex());


        //mark script as important because it was used at least for one
        //coinbase output if the script came from the wallet
        if (keepScript)
        {
            coinbaseScript->KeepScript();
        }
    }
    return blockHashes;
}
```

下面解析比特币 PoW 算法的实现原理。

在比特币系统中，PoW 算法的过程可理解为不断调整 Nonce 值。调整 Nonce 值就是对区块头做双重 SHA-256 哈希运算，使得结果满足给定数量前导 0 的哈希值的过程。其中前导 0 的个数，取决于挖矿难度，前导 0 的个数越多，挖矿难度越大。

具体的处理逻辑如下。

1）生成铸币交易，然后与其他所有准备打包进区块的交易一起组成交易列表。随后基于交易列表中的数据生成 Merkle 树，并计算 Merkle 树根节点的哈希值。

2）将 Merkle 树根节点的哈希值与其他字段（如区块高度、时间戳等）组成区块头，将 80 字节长度的区块头作为 PoW 算法的输入。

3）不断变更区块头中的随机数 Nonce，对变更后的区块头做双重 SHA−256 哈希运算。将得到的结果与当前难度的目标值做比对，如果小于目标难度，即 PoW 算法完成。

在一定时间段内率先完成 PoW 算法的区块向全网广播，其他节点验证其是否符合规则，如果验证有效，其他节点将接收此区块，并附加在已有区块链之后，随后开启下一轮挖矿。

目前比特币系统已经吸引了全球大部分的算力，其他在基于 PoW 共识机制的区块链公链应用很难获得相同的算力来保障期自身的安全。此外，共识达成的周期较长，挖矿造成大量的电力和计算资源的浪费使其不适合商业应用。

5.5.3　以太坊的 PoW 实现

在以太坊（开源代码地址：https://github.com/ethereum/go-ethereum/）的开源实现中，PoW 算法在 sealer.go 类中实现，其核心代码如下：

```
// mine is the actual proof-of-work miner that searches for a nonce
// starting from
// seed that results in correct final block difficulty.
func (ethash *Ethash) mine(block *types.Block, id int, seed uint64,
   abort chan struct{}, found chan *types.Block) {
    // Extract some data from the header
    var (
        header  = block.Header()
        hash    = header.HashNoNonce().Bytes()
        target  = new(big.Int).Div(maxUint256, header.Difficulty)
        number  = header.Number.Uint64()
        dataset = ethash.dataset(number)
    )
    // Start generating random nonces until we abort or find a good one
    var (
```

```
            attempts = int64(0)
            nonce    = seed
        )
        logger := log.New("miner", id)
        logger.Trace("Started ethash search for new nonces", "seed", seed)
           search:
        for {
            select {
            case <-abort:
                // Mining terminated, update stats and abort
                logger.Trace("Ethash nonce search aborted", "attempts",
                    nonce-seed)
                ethash.hashrate.Mark(attempts)
                break search


            default:
                // We don't have to update hash rate on every nonce, so update
                //after after 2^X nonces
                attempts++
                if (attempts % (1 << 15)) == 0 {
                    ethash.hashrate.Mark(attempts)
                    attempts = 0
                }
                // Compute the PoW value of this nonce
                digest, result := hashimotoFull(dataset.dataset, hash,
                    nonce)
                if new(big.Int).SetBytes(result).Cmp(target) <= 0 {
                    // Correct nonce found, create a new header with it
                    header = types.CopyHeader(header)
                    header.Nonce = types.EncodeNonce(nonce)
                    header.MixDigest = common.BytesToHash(digest)


                    // Seal and return a block (if still needed)
                    select {
                    case found <- block.WithSeal(header):
                        logger.Trace("Ethash nonce found and reported",
                            "attempts", nonce-seed, "nonce", nonce)
                    case <-abort:
                        logger.Trace("Ethash nonce found but discarded",
                            "attempts", nonce-seed, "nonce", nonce)
```

```
                }
                break search
            }
            nonce++
        }
    }
    // Datasets are unmapped in a finalizer. Ensure that the dataset
    // stays live
    // during sealing so it's not unmapped while being read.
    runtime.KeepAlive(dataset)
}
```

如代码所示，以太坊的 PoW 算法是 Ethash。Ethash 算法的核心思想是找到一个合适的 Nonce 值，作为算法接口的入参，经计算后得到的结果希望是低于特定困难度的阈值。

Ethash 算法的逻辑如下所示。

1）存在一个种子 seed，通过扫描块头为每个块计算出来那个点。

2）根据这个种子 seed，可以计算一个 16MB 的伪随机缓存 cache，轻客户端存储这个缓存。

从这个缓存 cache 中，我们能够生成一个 1GB 的数据集，该数据集中的每一项都取决于缓存中的一小部分。完整客户端和矿工存储了这个数据集，数据集随着时间线性增长。

在以太坊系统中，挖矿的工作包含了抓取数据集的随机片，以及运用哈希函数计算它们。校验工作能够在低内存的环境下完成，通过使用缓存再次生成所需的特性数据集的片段，所以矿工只需存储缓存 cache 即可。

5.6　联盟链中 PBFT 的实现

在联盟链中，联盟各个节点往往都来自同一行业，有着共同的行业困扰和痛点，因此联盟链往往注重对实际问题的高效解决。而 PoW 算法相对低效且费时费力，因此在联盟链中并不适用。

相反，在公链中很小适用的 PBFT 算法在联盟链中却有用武之地。PBFT 算法及

变种算法应用较为广泛，目前超级账本 Hyperledger 项目正在开发基于 PBFT 实现的共识算法。因此，本节将重点介绍 PBFT 算法，并基于 WebSocket 和 t-io 将其实现。

5.6.1 什么是 PBFT

PBFT，全称为 Practical Byzantine Fault Tolerance，意为实用拜占庭容错算法。该算法由卡斯特罗（Miguel Castro）和利斯科夫（Barbara Liskov）于 1999 年提出，旨在解决原始拜占庭容错算法效率不高的问题。熟悉设计模式的读者对利斯科夫（Barbara Liskov）应该不陌生，他就是提出著名的里氏替换原则（LSP）的人，曾于 2008 年获得图灵奖。

PBFT 算法的提出是为了解决拜占庭将军问题。什么是拜占庭将军问题呢？拜占庭将军问题最早是由 Leslie Lamport、Robert Shostak 和 Marshall Pease 在 1982 年发表的论文 *The Byzantine Generals Problem* 中提出的。

拜占庭将军问题源于古代拜占庭的军事战场，其背景大致如下：

拜占庭位于如今的土耳其的伊斯坦布尔，是古代东罗马帝国的首都。拜占庭罗马帝国幅员辽阔，为了实现积极防御国防政策，拜占庭罗马帝国的每块封地都驻扎一支由将军统领的军队，每支军队都分隔很远，两支军队的将军之间只能靠信差传递消息。

在战争发生时，拜占庭帝国的各支军队内所有将军必需达成一个共识，根据是否有赢的机会再去攻打敌人的阵营。不过，在军队内有可能存有叛徒或敌军的间谍，这些人混迹于军队中，左右着将军们的决定，影响将军们达成共识。那么，在已知将军们中有叛徒的情况下，其余忠诚的将军们如何才能达成一致协议呢？这就是拜占庭将军问题。

Leslie Lamport 在论文中证明了在将军总数大于 3*f*，背叛者为 *f* 或者更少的情况下，忠诚的将军可以达成命令上的一致，即 3*f*+1≤*n*。算法复杂度为 *o*（*n*^(*f*+1)），显然时间复杂度很高。为此，卡斯特罗和利斯科夫在 1999 年发表的论文 *Practical Byzantine Fault Tolerance* 中首次提出 PBFT，该算法容错数量也满足 3*f*+1≤*n*，但算法复杂度已优化为 *o*（*n*^2）。

PBFT 分五个阶段实施，执行前需要先在全网节点选举出一个主节点，新区块的生成由主节点负责。选举出主节点后，开始落实 PBFT。PBFT 的五个阶段分别是：Request 阶段、Pre-prepare 阶段、Prepare　阶段、Commit 阶段、执行并 Reply。下面分别阐述各个阶段的处理逻辑。

假设此时有主节点、从 1 节点、从 2 节点和从 3 节点共四个节点。

1）第一阶段，即 Request 阶段的处理逻辑如下：

客户端发起请求。

2）第二阶段，即 Pre-prepare 阶段的处理逻辑如下：

主节点收到客户端请求后给请求进行编号，并发送 pre-pre 类型信息给其他从节点。

3）第三阶段，即 Prepare 阶段的处理逻辑如下：

从 1 节点同意主节点请求的编号，并发送 prepare 类型消息给主节点和其他两个从节点。如果从 1 节点不同意请求的编号，则不进行处理。

从 1 节点如果收到另外两个从节点都发出的同意主节点分配的编号的 prepare 类型的消息，则表示从 1 节点的状态置为 Prepared，进入 Commit 阶段。

4）第四阶段，即 Commit 阶段的处理逻辑如下：

从 1 节点进入 Prepared 状态后，将发送一条 commit 类型信息给其他所有节点，告诉其他节点当前从 1 节点已经进入 Prepared 状态了。

如果从 1 节点收到 $2f+1$ 条 commit 信息，则证明从 1 节点已经进入 Commit 状态。

当一个请求在某个节点中到达 Commit 状态后，该请求就会被该节点执行。

5）第五阶段，即执行及 Reply 阶段的处理逻辑如下：

执行区块生成并 Reply 生成结果。

目前，Hyperledger Fabric 中已将 PBFT 纳入其候选共识算法集。不过，PBFT 的缺点也很明显，由于先选举主节点，因此当主节点宕机不得不重新选举主节点时，PBFT 将无法正常达成共识。

此外，虽然联盟链中各个节点都是由同行业内知根知底的机构组成的，但安全性同样不可小觑。一旦主节点被攻击甚至是作恶，其他节点并不能及时发现。这也是联盟链中不可避免要处理的问题。

上文已经介绍了 PBFT 的原理，下面我们将基于第 4 章 P2P 网络构建的实例，介绍 PBFT 的实现，即我们通过 WebSocket 和 t-io 分别实现 PBFT。

5.6.2 PBFT 基于 WebSocket 的实现

在 PBFT 中，节点是有状态的，如 Pre-prepare 状态、Prepare 状态、Commit 状态等。对应到 Java 代码中，我们设定投票阶段的 3 个状态的 Enum 如下所示：

```
package com.niudong.demo.p2ppbft;

/**
 * pbft 投票 Enum 类
 *
 * @author 牛冬
 *
 */
public enum VoteEnum {
  PREPREPARE("节点将自己生成 Block", 100), PREPARE("节点收到请求生成 Block
      的消息，进入准备状态，并对外广播该状态", 200), COMMIT("每个节点收到超过
      2f+1 个不同节点的 commit 消息后，则认为该区块已经达成一致，即进入 Commit 状
      态，并将其持久化到区块链数据库中", 400);

  // 投票情况描述
  private String msg;
  // 投票情况状态码
  private int code;

  // 根据状态码返回对应的 Enum
  public static VoteEnum find(int code) {
    for (VoteEnum ve : VoteEnum.values()) {
      if (ve.code == code) {
        return ve;
      }
    }
    return null;
  }
```

```
    public String getMsg() {
      return msg;
    }

    public void setMsg(String msg) {
      this.msg = msg;
    }

    public int getCode() {
      return code;
    }

    public void setCode(int code) {
      this.code = code;
    }

    VoteEnum(String msg, int code) {
      this.msg = msg;
      this.code = code;
    }
}
```

PBFT 在投票过程中的投票实体需要包含上述 VoteEnum 中的投票状态码，还可以包含待写入区块的内容及对应的 Merkle 树根节点的哈希值，因此我们构建投票实体类 VoteInfo。VoteInfo 类的代码如下所示：

```
package com.niudong.demo.p2ppbft;

import java.util.List;

/**
 * 投票信息类
 *
 * @author 牛冬
 *
 */
public class VoteInfo {

    // 投票状态码
    private int code;
    // 待写入区块的内容
```

```
    private List<String> list;
    // 待写入区块的内容的 Merkle 树根节点哈希值
    private String hash;

    public String getHash() {
      return hash;
    }

    public void setHash(String hash) {
      this.hash = hash;
    }

    public List<String> getList() {
      return list;
    }

    public void setList(List<String> list) {
      this.list = list;
    }

    public int getCode() {
      return code;
    }

    public void setCode(int code) {
      this.code = code;
    }
  }
```

服务端代码如下所示：

```
  package com.niudong.demo.p2ppbft;

  import java.net.InetSocketAddress;
  import java.util.ArrayList;
  import java.util.List;

  import javax.annotation.PostConstruct;

  import org.java_websocket.WebSocket;
  import org.java_websocket.handshake.ClientHandshake;
  import org.java_websocket.server.WebSocketServer;
  import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.testng.util.Strings;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.alibaba.fastjson.parser.JSONScanner;
import com.niudong.demo.util.SHAUtil;
import com.niudong.demo.util.SimpleMerkleTree;


/**
 * 基于 Spring Boot 2.0 的 WebSocket 服务端
 *
 * @author 牛冬
 *
 */
@Component
public class P2pPointPbftServer {
  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(P2pPointPbftServer.class);

  // 本机 Server 的 WebSocket 端口
  // 多机测试时可改变该值
  private int port = 7001;

  // 所有连接到服务端的 WebSocket 缓存器
  private List<WebSocket> localSockets = new ArrayList<WebSocket>();

  public List<WebSocket> getLocalSockets() {
    return localSockets;
  }

  public void setLocalSockets(List<WebSocket> localSockets) {
    this.localSockets = localSockets;
  }

  /**
   * 初始化 P2P Server 端
   *
```

```
     * @param Server 端的端口号 port
     */
    @PostConstruct
    @Order(1)
    public void initServer() {
      /**
       * 初始化 WebSocket 的服务端，定义内部类对象 socketServer，源于
         WebSocketServer； new
       * InetSocketAddress(port)是 WebSocketServer 构造器的参数，
         InetSocketAddress 是（IP 地址+端口号）类型，即端口地址类型
       */
      final WebSocketServer socketServer = new WebSocketServer(new
          InetSocketAddress(port)) {
        /**
         * 重写 5 个事件方法，事件发生时触发对应的方法
         */

        @Override
        // 创建连接成功时触发
        public void onOpen(WebSocket webSocket, ClientHandshake
            clientHandshake) {
          sendMessage(webSocket, "北京服务端成功创建连接");

          // 当成功创建一个 WebSocket 连接时，将该连接加入连接池
          localSockets.add(webSocket);
        }

        @Override
        // 断开连接时候触发
        public void onClose(WebSocket webSocket, int i, String s, Boolean
            b) {
          logger.info(webSocket.getRemoteSocketAddress() + "客户端与服务器
              断开连接！");

          // 当客户端断开连接时，WebSocket 连接池删除该连接
          localSockets.remove(webSocket);
        }

        @Override
        // 收到客户端发来的消息时触发
        public void onMessage(WebSocket webSocket, String msg) {
          logger.info("北京服务端接收到客户端消息：" + msg);
```

```
        //收到入库的消息则不再发送
        if("北京客户端开始区块入库啦".equals(msg)) {
          return;
        }

        // 如果收到的不是 JSON 化数据，则说明仍处在双方建立连接的过程中。目前连接
        //已经建立完毕，发起投票
        if (!msg.startsWith("{")) {
          VoteInfo vi = createVoteInfo(VoteEnum.PREPREPARE);

          sendMessage(webSocket, JSON.toJSONString(vi));
          logger.info("北京服务端发送到客户端的 pbft 消息：" +
          JSON.toJSONString(vi));
          return;
        }

        // 如果是 JSON 化数据，则表明已经进入到了 PBFT 投票阶段
        JSONObject json = JSON.parseObject(msg);
        if (!json.containsKey("code")) {
          logger.info("北京服务端收到非 JSON 化数据");
        }

        int code = json.getIntValue("code");
        if (code == VoteEnum.PREPARE.getCode()) {
          // 校验哈希
          VoteInfo voteInfo = JSON.parseObject(msg, VoteInfo.class);
          if (!voteInfo.getHash().equals(SimpleMerkleTree.
              getTreeNodeHash(voteInfo.getList()))) {
            logger.info("北京服务端收到错误的 JSON 化数据");
            return;
          }

          // 校验成功，发送下一个状态的数据
          VoteInfo vi = createVoteInfo(VoteEnum.COMMIT);
          sendMessage(webSocket, JSON.toJSONString(vi));
          logger.info("北京服务端发送到客户端 pbft 消息：" +
              JSON.toJSONString(vi));
        }

      }
```

```
        @Override
        // 连接发生错误时调用，紧接着触发 onClose 方法
        public void onError(WebSocket webSocket, Exception e) {
          logger.info(webSocket.getRemoteSocketAddress() + "客户端连接错
            误！");
          localSockets.remove(webSocket);
        }

        @Override
        public void onStart() {
          logger.info("北京的 WebSocket Server 端启动...");
        }
      };

      socketServer.start();
      logger.info("北京服务端监听 socketServer 端口" + port);
    }

    // 根据 VoteEnum 构建对应状态的 VoteInfo
    private VoteInfo createVoteInfo(VoteEnum ve) {
      VoteInfo vi = new VoteInfo();
      vi.setCode(ve.getCode());

      List<String> list = new ArrayList<>();
      list.add("AI");
      list.add("BlockChain");
      vi.setList(list);
      vi.setHash(SimpleMerkleTree.getTreeNodeHash(list));

      return vi;
    }

    /**
     * 向连接到本机的某客户端发送消息
     *
     * @param ws
     * @param message
     */
    public void sendMessage(WebSocket ws, String message) {
      logger.info("发送给" + ws.getRemoteSocketAddress().getPort() + "的
          p2p 消息是:" + message);
      ws.send(message);
```

```
    }

    /**
     * 向所有连接到本机的客户端广播消息
     *
     * @param message：待广播内容
     */
    public void broatcast(String message) {
      if (localSockets.size() == 0 || Strings.isNullOrEmpty(message)) {
        return;
      }

      logger.info("Glad to say broatcast to clients being startted!");
      for (WebSocket socket : localSockets) {
        this.sendMessage(socket, message);
      }
      logger.info("Glad to say broatcast to clients has overred!");
    }
  }
```

如上述代码所示，服务端用 initServer()方法初始化，默认打开本机 7001 端口作为服务端口。在 initServer()中重写 5 个事件方法，对应的事件发生时触发对应的方法。这 5 个方法分别是服务端启动时调用的方法 onStart()、创建连接成功时触发的方法 onOpen()、断开连接时候触发的方法 onClose()、收到客户端发来消息时触发的方法 onMessage()、连接发生错误时调用的方法 onError()。onError()方法调用完毕后会触发 onClose 方法。

其中，onMessage 方法里实现了服务端的 PBFT，主要逻辑如下：

1）当服务端收到“北京客户端开始区块入库啦”的消息时，服务端认为 PBFT 共识已经达成，因此不再对消息进行处理。

2）当服务端收到的消息内容不是 JSON 化数据时，说明仍在双方建立连接的过程中。目前连接刚刚建立完毕，于是发起投票，投票的数据状态置为 Pre-prepare 状态。

3）当服务端收到的消息内容是 JSON 化数据，则说明目前已经进入到了 PBFT 投票阶段。服务端对数据信息进行成功校验后，将投票状态置为 Commit 状态。

为保证 initServer()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证服务端先于客户端加载，用@Order(1)标识了 initServer()。

此外，服务端代码中还实现了向连接到本机的某客户端发送消息的方法 sendMessage()和向所有连接到本机的客户端广播消息的方法 broatcast()。

客户端代码实现逻辑如下所示：

```
package com.niudong.demo.p2ppbft;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;

import org.java_websocket.WebSocket;
import org.java_websocket.client.WebSocketClient;
import org.java_websocket.handshake.ServerHandshake;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.testng.util.Strings;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.niudong.demo.util.SimpleMerkleTree;



/**
 * 基于 Spring Boot 2.0 的 WebSocket 客户端
 *
 * @author 牛冬
 *
 */
@Component
public class P2pPointPbftClient {
```

```
    // 日志记录
    private Logger logger =
        LoggerFactory.getLogger(P2pPointPbftClient.class);

    // P2P 网络中的节点既是服务端，又是客户端。作为服务端运行在 7001 端口，同时作
    // 为客户通过 ws://localhost:7001 连接到服务端
    private String wsUrl = "ws://localhost:7001/";

    // 所有客户端 WebSocket 的连接池缓存
    private List<WebSocket> localSockets = new ArrayList<WebSocket>();


    public List<WebSocket> getLocalSockets() {
        return localSockets;
    }

    public void setLocalSockets(List<WebSocket> localSockets) {
        this.localSockets = localSockets;
    }

    /**
     * 连接到服务端
     */
    @PostConstruct
    @Order(2)
    public void connectPeer() {
      try {
        // 创建 WebSocket 的客户端
        final WebSocketClient socketClient = new WebSocketClient(new
            URI(wsUrl)) {
          @Override
          public void onOpen(ServerHandshake serverHandshake) {
              sendMessage(this, "北京客户端成功创建客户端");

              localSockets.add(this);
          }

          @Override
          public void onMessage(String msg) {
              logger.info("北京客户端收到北京服务端发送的消息:" + msg);

            // 如果收到的不是 JSON 化数据，则说明不是 PBFT 阶段
```

```
            if (!msg.startsWith("{")) {
              return;
            }

            // 如果收到的是 JSON 化数据，则说明是 PBFT 阶段
            // 如果是 JSON 化数据，则进入到了 PBFT 投票阶段
            JSONObject json = JSON.parseObject(msg);
            if (!json.containsKey("code")) {
                logger.info("北京客户端收到非 JSON 化数据");
            }

            int code = json.getIntValue("code");
            if (code == VoteEnum.PREPREPARE.getCode()) {
              // 校验哈希
              VoteInfo voteInfo = JSON.parseObject(msg, VoteInfo.class);
              if (!voteInfo.getHash().equals(SimpleMerkleTree.
                  getTreeNodeHash(voteInfo.getList()))) {
                logger.info("北京客户端收到北京服务端错误的 JSON 化数据");
                return;
              }

              // 校验成功，发送下一个状态的数据
              VoteInfo vi = createVoteInfo(VoteEnum.PREPARE);
              sendMessage(this, JSON.toJSONString(vi));
              logger.info("北京客户端发送到服务端 PBFT 消息: " +
                  JSON.toJSONString(vi));
              return;
            }

            if (code == VoteEnum.COMMIT.getCode()) {
              // 校验哈希
              VoteInfo voteInfo = JSON.parseObject(msg, VoteInfo.class);
              if (!voteInfo.getHash().equals(SimpleMerkleTree.
                  getTreeNodeHash(voteInfo.getList()))) {
                logger.info("北京客户端到北京服务端错误的 JSON 化数据");
                return;
              }

              // 校验成功，检验节点确认个数是否有效
              if (getConnecttedNodeCount() >= getLeastNodeCount()) {
                sendMessage(this, "北京客户端开始区块入库啦");
                logger.info("北京客户端开始区块入库啦");
```

```
            }
          }
        }

        @Override
        public void onClose(int i, String msg, boolean b) {
          logger.info("北京客户端关闭");
          localSockets.remove(this);
        }

        @Override
        public void onError(Exception e) {
          logger.info("北京客户端报错");
          localSockets.remove(this);
        }
      };
      // 客户端开始连接服务端
      socketClient.connect();
    } catch (URISyntaxException e) {
      logger.info("北京连接错误:" + e.getMessage());
    }
  }

  // 已经在连接的节点的个数
  private int getConnecttedNodeCount() {
    // 本机测试时，写死为 1。实际开发部署多个节点时，按实际情况返回
    return 1;
  }

  // PBFT 消息节点最少确认个数计算
  private int getLeastNodeCount() {
    // 本机测试时，写死为 1。实际开发部署多个节点时，PBFT 算法中拜占庭节点数量 f，
    // 总节点数 3f+1
    return 1;
  }

  // 根据 VoteEnum 构建对应状态的 VoteInfo
  private VoteInfo createVoteInfo(VoteEnum ve) {
    VoteInfo vi = new VoteInfo();
    vi.setCode(ve.getCode());

    List<String> list = new ArrayList<>();
```

```
        list.add("AI");
        list.add("BlockChain");
        vi.setList(list);
        vi.setHash(SimpleMerkleTree.getTreeNodeHash(list));

        return vi;
    }

    /**
     * 向服务端发送消息。当前 WebSocket 的远程 Socket 地址就是服务端
     *
     * @param ws:
     * @param message
     */
    public void sendMessage(WebSocket ws, String message) {
        logger.info("发送给" + ws.getRemoteSocketAddress().getPort() + "的
            p2p 消息:" + message);
        ws.send(message);
    }

    /**
     * 向所有连接过的服务端广播消息
     *
     * @param message：待广播的消息
     */
    public void broatcast(String message) {
        if (localSockets.size() == 0 || Strings.isNullOrEmpty(message)) {
            return;
        }

        logger.info("Glad to say broatcast to servers being startted!");
        for (WebSocket socket : localSockets) {
            this.sendMessage(socket, message);
        }
        logger.info("Glad to say broatcast to servers has overred!");
    }
}
```

如上述代码所示，客户端用 connectPeer()方法连接到服务端，在方法中客户端通过 ws://localhost:7001 连接到服务端。与服务端类似，客户端在 connectPeer()中重写 5 个事件方法，对应的事件发生时触发对应的方法。这 5 个方法分别是客户端启动时

调用的方法 onStart()、创建连接成功时触发的方法 onOpen()、断开连接时候触发的方法 onClose()、收到服务端发来消息时触发的方法 onMessage()、连接发生错误时调用的方法 onError()，onError()方法调用完毕后会触发 onClose 方法。

其中，onMessage()方法里实现了客户端的 PBFT 算法，主要逻辑如下：

1）当客户端收到的是非 JSON 化数据时，说明当前不是 PBFT 阶段。

2）当客户端收到的消息内容是 JSON 化数据时，说明目前已经进入到了 PBFT 投票阶段。客户端对数据信息进行成功校验后，将投票状态置为 Prepare 状态

3）当客户端校验消息成功后，且当前检验节点确认个数有效，则开始向服务端发送“北京客户端开始区块入库啦”消息。

为保证 connectPeer()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证客户端后于服务端加载，用@Order(2)标识了 connectPeer()。

此外，客户端代码中还实现了计算已经在连接的节点的个数的方法 getConnecttedNodeCount()、PBFT 消息节点最少确认个数计算的方法 getLeastNodeCount()、向服务端发送消息的方法 sendMessage()，以及向所有连接过的服务端广播消息的方法 broatcast()。

代码编写完成后，我们在名为 demo 的工程目录下运行 mvn clean package 命令，将 demo 工程打包成 demo.jar，随后切换到 target 目录，执行 java -jar demo.jar，得到的结果如图 5-1 所示。

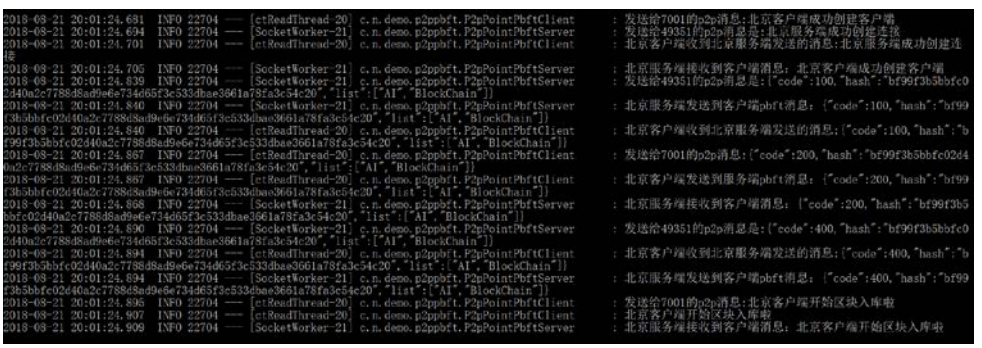


图 5-1　代码运行日志图

5.6.3 PBFT 基于 t-io 的实现

和前面基于 WebSocket 实现 PBFT 类似，在基于 t-io 实现 PBFT 时，也需要设定投票阶段的状态，标识投票 3 个状态的 Enum 的代码如下所示。

```
package com.niudong.demo.tiop2ppbft;


/**
 * PBFT 投票 Enum 类
 *
 * @author 牛冬
 *
 */
public enum VoteEnum {
  PREPREPARE("节点将自己生成 Block", 100), PREPARE("节点收到请求生成 Block
  的消息，进入准备状态，并对外广播该状态", 200), COMMIT("每个节点收到超过 2f+1
  个不同节点的 commit 消息后，则认为该区块已经达成一致，即进入 Committ 状态，并将
  其持久化到区块链数据库中", 400);

  // 投票情况描述
  private String msg;
  // 投票情况状态码
  private int code;

  // 根据状态码返回对应的 Enum
  public static VoteEnum find(int code) {
    for (VoteEnum ve : VoteEnum.values()) {
      if (ve.code == code) {
        return ve;
      }
    }
    return null;
  }

  public String getMsg() {
    return msg;
  }

  public void setMsg(String msg) {
    this.msg = msg;
  }
```

```
    public int getCode() {
      return code;
    }

    public void setCode(int code) {
      this.code = code;
    }

    VoteEnum(String msg, int code) {
      this.msg = msg;
      this.code = code;
    }
}
```

表示 PBFT 在投票过程中的投票实体需要包含上述 VoteEnum 中的投票状态码，还可以包含待写入区块的内容及对应的 Merkle 树根节点的哈希值，因此我们构建投票实体类 VoteInfo，VoteInfo 类的代码如下所示：

```
package com.niudong.demo.tiop2ppbft;

import java.util.List;

/**
 * 投票信息类
 *
 * @author 牛冬
 *
 */
public class VoteInfo {
  // 投票状态码
  private int code;
  // 待写入区块的内容
  private List<String> list;
  // 待写入区块的内容的 Merkle 树根节点的哈希值
  private String hash;

  public String getHash() {
    return hash;
  }

  public void setHash(String hash) {
```

```
    this.hash = hash;
  }

  public List<String> getList() {
    return list;
  }

  public void setList(List<String> list) {
    this.list = list;
  }

  public int getCode() {
    return code;
  }

  public void setCode(int code) {
    this.code = code;
  }
}
```

在服务端，基于 t-io 的区块链底层 P2P 网络平台所需的常量类定义如下：

```
package com.niudong.demo.tiop2ppbft;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台所需的常量
 *
 * @author 牛冬
 *
 */
public class Const {
  /**
   * 服务器地址
   */
  public static final String SERVER = "127.0.0.1";

  /**
   * 监听端口
   */
  public static final int PORT = 6789;

  /**
   * 心跳超时时间
```

```
    */
    public static final int TIMEOUT = 5000;
}
```

服务端定制化的区块链底层 Packet 代码如下所示：

```
package com.niudong.demo.tiop2ppbft;

import org.tio.core.intf.Packet;

/**
 * 区块链底层定制的 Packet
 *
 * @author 牛冬
 *
 */
public class BlockPacket extends Packet {
    // 网络传输需序列化，这里采用 Java 自带序列化方式
    private static final long serialVersionUID = -172060606924066412L;
    // 消息头的长度
    public static final int HEADER_LENGHT = 4;
    // 字符编码类型
    public static final String CHARSET = "utf-8";
    // 传输内容的字节
    private byte[] body;

    /**
     * @return the body
     */
    public byte[] getBody() {
        return body;
    }

    /**
     * @param body the body to set
     */
    public void setBody(byte[] body) {
        this.body = body;
    }
}
```

服务端相关的代码如下所示：

```
package com.niudong.demo.tiop2ppbft;

import javax.annotation.PostConstruct;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.tio.server.ServerGroupContext;
import org.tio.server.TioServer;
import org.tio.server.intf.ServerAioHandler;
import org.tio.server.intf.ServerAioListener;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的服务端
 *
 * @author 牛冬
 *
 */
@Component
public class BlockChainPbftServerStarter {
  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(BlockChainPbftServerStarter.class);

  // handler，包括编码、解码、消息处理
  public static ServerAioHandler aioHandler = new
      BlockChainPbftServerAioHandler();

  // 事件监听器，可以为 null，但建议自己实现该接口，可以参考 showcase 了解接口
  public static ServerAioListener aioListener = null;

  // 一组连接共用的上下文对象
  public static ServerGroupContext serverGroupContext =
     new ServerGroupContext("hello-tio-server", aioHandler,
         aioListener);

  // tioServer 对象
  public static TioServer tioServer = new
      TioServer(serverGroupContext);

  // 有时候需要绑定 IP，不需要则为 null
```

```
    public static String serverIp = null;//Const.SERVER;

    // 监听的端口
    public static int serverPort = Const.PORT;

    @PostConstruct
    @Order(1)
    public void start() {
      try {
        logger.info("北京服务端即将启动");

        serverGroupContext.setHeartbeatTimeout(Const.TIMEOUT);
        tioServer.start(serverIp, serverPort);

        logger.info("北京服务端启动完毕");
      } catch (Exception e) {
        logger.error(e.getMessage());
      }
    }
  }

  package com.niudong.demo.tiop2ppbft;

  import org.slf4j.Logger;
  import org.slf4j.LoggerFactory;
  import org.tio.core.ChannelContext;
  import org.tio.core.GroupContext;
  import org.tio.core.Tio;
  import org.tio.core.exception.AioDecodeException;
  import org.tio.core.intf.Packet;
  import org.tio.server.intf.ServerAioHandler;

  import com.alibaba.fastjson.JSON;
  import com.alibaba.fastjson.JSONObject;
  import com.niudong.demo.util.SimpleMerkleTree;

  import java.nio.ByteBuffer;
  import java.util.ArrayList;
  import java.util.List;

  /**
   * 基于 t-io 的区块链底层 P2P 网络平台的服务端 Handler
```

```
 *
 * @author 牛冬
 *
 */
public class BlockChainPbftServerAioHandler implements
    ServerAioHandler {

  // 日志记录
  private Logger logger =
      LoggerFactory.getLogger(BlockChainPbftServerAioHandler.class);

  /**
   * 解码：把接收到的 ByteBuffer 解码成应用可以识别的业务消息包。总的消息结构：
     消息头 + 消息体。消息头结构： 4 个字节，存储消息体的长度。消息体结构：对象的
     JSON 串的 byte[]
   */
  @Override
  public BlockPacket decode(ByteBuffer buffer, int limit, int position,
      int readableLength,
     ChannelContext channelContext) throws AioDecodeException {
    // 提醒：buffer 的开始位置并不一定是 0，应用需要从 buffer.position()开始读
    // 取数据
    // 若收到的数据无法组成业务包，则返回 null，告诉框架数据不够
    if (readableLength < BlockPacket.HEADER_LENGHT) {
      return null;
    }

    // 读取消息体的长度
    int bodyLength = buffer.getInt();

    // 数据不正确，则抛出 AioDecodeException 异常
    if (bodyLength < 0) {
      throw new AioDecodeException(
          "bodyLength [" + bodyLength + "] is not right, remote:" +
               channelContext.getClientNode());
    }

    // 计算本次需要的数据长度
    int neededLength = BlockPacket.HEADER_LENGHT + bodyLength;
    // 收到的数据是否足够组包
    int isDataEnough = readableLength - neededLength;
    // 不够消息体长度（剩下的 buffer 组不了消息体）
```

```
      if (isDataEnough < 0) {
        return null;
      } else // 组包成功
      {
        BlockPacket imPacket = new BlockPacket();
        if (bodyLength > 0) {
          byte[] dst = new byte[bodyLength];
          buffer.get(dst);
          imPacket.setBody(dst);
        }
        return imPacket;
      }
    }

    /**
     * 编码：把业务消息包编码为可以发送的 ByteBuffer。总的消息结构：消息头 + 消息
       体。消息头结构：4 个字节，存储消息体的长度。消息体结构：对象的 JSON 串的 byte[]
     */
    @Override
    public ByteBuffer encode(Packet packet, GroupContext groupContext,
        ChannelContext channelContext) {
      BlockPacket helloPacket = (BlockPacket) packet;
      byte[] body = helloPacket.getBody();
      int bodyLen = 0;
      if (body != null) {
        bodyLen = body.length;
      }

      // ByteBuffer 的总长度 = 消息头的长度 + 消息体的长度
      int allLen = BlockPacket.HEADER_LENGHT + bodyLen;
      // 创建一个新的 ByteBuffer
      ByteBuffer buffer = ByteBuffer.allocate(allLen);
      // 设置字节序
      buffer.order(groupContext.getByteOrder());

      // 写入消息头，消息头的内容就是消息体的长度
      buffer.putInt(bodyLen);

      // 写入消息体
      if (body != null) {
        buffer.put(body);
      }
```

```
    return buffer;
  }


  /**
   * 处理消息
   */
  @Override
  public void handler(Packet packet, ChannelContext channelContext)
      throws Exception {
    BlockPacket helloPacket = (BlockPacket) packet;
    byte[] body = helloPacket.getBody();
    if (body != null) {
      String str = new String(body, BlockPacket.CHARSET);
      logger.info("北京服务端收到消息：" + str);

      // 如果收到的不是 JSON 化数据，则说明不是 PBFT 阶段
      if (!str.startsWith("{")) {
        BlockPacket resppacket = new BlockPacket();
        resppacket.setBody(("北京服务端收到了客户端的消息，客户端的消息是:" +
            str).getBytes(BlockPacket.CHARSET));
        Tio.send(channelContext, resppacket);
        return;
      }

      // 如果收到的是 JSON 数据，则说明是 PBFT 阶段
      // 如果是 JSON 化数据，则进入到了 PBFT 投票阶段
      JSONObject json = JSON.parseObject(str);
      if (!json.containsKey("code")) {
        logger.info("北京服务端收到 JSON 化数据");
      }

      int code = json.getIntValue("code");
      if (code == VoteEnum.PREPREPARE.getCode()) {
        // 校验哈希
        VoteInfo voteInfo = JSON.parseObject(str, VoteInfo.class);
        if (!voteInfo.getHash().equals(SimpleMerkleTree.
            getTreeNodeHash(voteInfo.getList()))) {
          logger.info("北京服务端收到北京客户端错误的 JSON 化数据");
          return;
        }
```

```
            // 校验成功，发送下一个状态的数据
            VoteInfo vi = createVoteInfo(VoteEnum.PREPARE);
            BlockPacket resppacket = new BlockPacket();
            resppacket.setBody(JSON.toJSONString(vi).
                getBytes(BlockPacket.CHARSET));
            Tio.send(channelContext, resppacket);
            logger.info("北京服务端发送到北京客户端 PBFT 消息：" +
               JSON.toJSONString(vi));
            return;
          }

          if (code == VoteEnum.COMMIT.getCode()) {
            // 校验哈希
            VoteInfo voteInfo = JSON.parseObject(str, VoteInfo.class);
            if (!voteInfo.getHash().equals(SimpleMerkleTree.
                getTreeNodeHash(voteInfo.getList()))) {
              logger.info("北京服务端收到北京客户端错误的 JSON 化数据");
              return;
            }

            // 校验成功，检验节点确认个数是否有效
            if (getConnecttedNodeCount() >= getLeastNodeCount()) {
              BlockPacket resppacket = new BlockPacket();
              resppacket.setBody("北京服务端开始区块入库啦
                 ".getBytes(BlockPacket.CHARSET));
              Tio.send(channelContext, resppacket);
              logger.info("北京服务端开始区块入库啦");
            }
          }

        }
        return;
      }

      // 已经连接的节点个数
      private int getConnecttedNodeCount() {
        // 本机测试时，写死为 1。实际开发部署多个节点时，按实际情况返回
        return 1;
      }

      // PBFT 消息节点最少确认个数计算
      private int getLeastNodeCount() {
```

```
        // 本机测试时，写死为 1。实际开发部署多个节点时，PBFT 算法中拜占庭节点数量 f，
        // 总节点数 3f+1
        return 1;
      }

      // 根据 VoteEnum 构建对应状态的 VoteInfo
      private VoteInfo createVoteInfo(VoteEnum ve) {
        VoteInfo vi = new VoteInfo();
        vi.setCode(ve.getCode());

        List<String> list = new ArrayList<>();
        list.add("AI");
        list.add("BlockChain");
        vi.setList(list);
        vi.setHash(SimpleMerkleTree.getTreeNodeHash(list));

        return vi;
      }
    }
```

如上述代码所示，基于 t-io 的区块链底层 P2P 网络平台服务端类 BlockChainPbftServerStarter 的 start()方法启动服务端，主要逻辑是初始化 serverGroupContext，并通过 tioServer.start()方法启动指定的 IP 和端口。为保证 start()在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时为了保证服务端先于客户端加载，用@Order(1)标识了 start()。

基于 t-io 的区块链底层 P2P 网络平台服务端 Handle 的类 BlockChainPbftServerAioHandler 提供了消息的解码方法 decode()、消息的编码方法 encode()，以及处理信息逻辑的 handler()方法。其中，handler()方法的逻辑是：

1）当服务端收到来自客户端的非 JSON 化数据时，则说明当前不是 PBFT 阶段，这里仅将收到的消息回传给客户端。

2）当服务端收到的是 JSON 化数据时，则说明目前是 PBFT 阶段。服务端先校验消息的有效性，校再通过后再根据消息的状态码进行相应的 PBFT 逻辑处理。

a）若服务端收到的是 pre-pre 消息，则将消息的状态置为 Prepare，并将消息发送给客户端。

b）若服务端收到的是 commit 消息，则检验节点确认个数是否有效，若有效则说明 PBFT 一致性达成，返回客户端“北京服务端开始区块入库啦”的消息。

此外，由于是单机测试，特对判断已经在连接的节点个数的方法 getConnecttedNodeCount() 和计算 PBFT 消息节点最少确认个数的方法 getLeastNodeCount()进行硬编码，返回值均为 1。

客户端相关代码如下所示：

```
package com.niudong.demo.tiop2ppbft;

import org.tio.client.TioClient;

import javax.annotation.PostConstruct;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.tio.client.ClientChannelContext;
import org.tio.client.ClientGroupContext;
import org.tio.client.ReconnConf;
import org.tio.client.intf.ClientAioHandler;
import org.tio.client.intf.ClientAioListener;
import org.tio.core.Tio;
import org.tio.core.Node;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的客户端
 *
 * @author 牛冬
 *
 */
@Component
public class BlockChainPbftClientStarter {
  // 日志记录
  private Logger logger =
LoggerFactory.getLogger(BlockChainPbftClientStarter.class);

  // 服务器节点
  private Node serverNode;
```

```
// handler，包括编码、解码、消息处理
private ClientAioHandler tioClientHandler;

// 事件监听器，可以为 null，但建议自己实现该接口，可以参考 showcase 了解接口
private ClientAioListener aioListener = null;

// 断连后可自动连接，若不想自动连接请设为 null
private ReconnConf reconnConf = new ReconnConf(5000L);

// 一组连接共用的上下文对象
private ClientGroupContext clientGroupContext;

private TioClient tioClient = null;
private ClientChannelContext clientChannelContext = null;

/**
 * 启动程序入口
 */
@PostConstruct
@Order(20)
public void start() {
  try {
    logger.info("北京客户端即将启动");

    //Thread.sleep(10*1000);

    //初始化
    serverNode = new Node(Const.SERVER, Const.PORT);
    tioClientHandler = new BlockChainPbftClientAioHandler();
    clientGroupContext = new ClientGroupContext(tioClientHandler,
       aioListener, reconnConf);

    clientGroupContext.setHeartbeatTimeout(Const.TIMEOUT);
    tioClient = new TioClient(clientGroupContext);
    clientChannelContext = tioClient.connect(serverNode);

    // 连上后，发一条消息测试
    sendMessage();

    logger.info("北京客户端启动完毕");
  } catch (Exception e) {
```

```
      logger.error(e.getMessage());
    }
  }

  private void sendMessage() throws Exception {
    BlockPacket packet = new BlockPacket();
    packet.setBody("tal say hello world to
        blockchain!".getBytes(BlockPacket.CHARSET));
    Tio.send(clientChannelContext, packet);
  }

}

package com.niudong.demo.tiop2ppbft;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.tio.client.intf.ClientAioHandler;
import org.tio.core.ChannelContext;
import org.tio.core.GroupContext;
import org.tio.core.Tio;
import org.tio.core.exception.AioDecodeException;
import org.tio.core.intf.Packet;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.niudong.demo.util.SimpleMerkleTree;

/**
 * 基于 t-io 的区块链底层 P2P 网络平台的客户端 Handler
 *
 * @author 牛冬
 *
 */
public class BlockChainPbftClientAioHandler implements
    ClientAioHandler {

  // 日志记录
```

```
    private Logger logger =
        LoggerFactory.getLogger(BlockChainPbftClientAioHandler.class);

    private static BlockPacket heartbeatPacket = new BlockPacket();


    /**
     * 解码：把接收到的 ByteBuffer 解码成应用可以识别的业务消息包。总的消息结构：
       消息头 + 消息体。消息头结构：4 个字节，存储消息体的长度。消息体结构：对象的
       JSON 串的 byte[]
     */
    @Override
    public BlockPacket decode(ByteBuffer buffer, int limit, int position,
        int readableLength,
        ChannelContext channelContext) throws AioDecodeException {
      // 若收到的数据无法组成业务包，则返回 null，告诉框架数据不够
      if (readableLength < BlockPacket.HEADER_LENGHT) {
        return null;
      }

      // 读取消息体的长度
      int bodyLength = buffer.getInt();

      // 数据不正确，则抛出 AioDecodeException 异常
      if (bodyLength < 0) {
        throw new AioDecodeException(
            "bodyLength [" + bodyLength + "] is not right, remote:" +
            channelContext.getClientNode());
      }

      // 计算本次需要的数据长度
      int neededLength = BlockPacket.HEADER_LENGHT + bodyLength;
      // 收到的数据是否足够组包
      int isDataEnough = readableLength - neededLength;
      // 不够消息体长度（剩下的 buffer 组不了消息体）
      if (isDataEnough < 0) {
        return null;
      } else // 组包成功
      {
        BlockPacket imPacket = new BlockPacket();
        if (bodyLength > 0) {
          byte[] dst = new byte[bodyLength];
```

```
            buffer.get(dst);
            imPacket.setBody(dst);
        }
        return imPacket;
    }
}

/**
 * 编码：把业务消息包编码为可以发送的 ByteBuffer。总的消息结构：消息头 + 消息体。消息头结构：4 个字节，存储消息体的长度。消息体结构：对象的 JSON 串的 byte[]
 */
@Override
public ByteBuffer encode(Packet packet, GroupContext groupContext, ChannelContext channelContext) {
    BlockPacket helloPacket = (BlockPacket) packet;
    byte[] body = helloPacket.getBody();
    int bodyLen = 0;
    if (body != null) {
        bodyLen = body.length;
    }

    // ByteBuffer 的总长度是 = 消息头的长度 + 消息体的长度
    int allLen = BlockPacket.HEADER_LENGHT + bodyLen;
    // 创建一个新的 ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate(allLen);
    // 设置字节序
    buffer.order(groupContext.getByteOrder());

    // 写入消息头，消息头的内容就是消息体的长度
    buffer.putInt(bodyLen);

    // 写入消息体
    if (body != null) {
        buffer.put(body);
    }
    return buffer;
}

/**
 * 处理消息
 */
@Override
```

```
    public void handler(Packet packet, ChannelContext channelContext)
        throws Exception {
      BlockPacket helloPacket = (BlockPacket) packet;
      byte[] body = helloPacket.getBody();
      if (body != null) {
        String str = new String(body, BlockPacket.CHARSET);
        logger.info("北京客户端收到消息：" + str);

        // 收到入库的消息则不再发送
        if ("北京服务端开始区块入库啦".equals(str)) {
          return;
        }

        // 发送 PBFT 投票信息
        // 如果收到的不是 JSON 化数据，说明仍在双方建立连接的过程中。目前连接已经建
        // 立完毕，发起投票
        if (!str.startsWith("{")) {
          VoteInfo vi = createVoteInfo(VoteEnum.PREPREPARE);
          BlockPacket bp = new BlockPacket();
          bp.setBody(JSON.toJSONString(vi).
              getBytes(BlockPacket.CHARSET));
          Tio.send(channelContext, bp);
          logger.info("北京客户端发送到服务端的 pbft 消息：" +
              JSON.toJSONString(vi));
          return;
        }

        // 如果是 JSON 化数据，则表明进入了 PBFT 投票阶段
        JSONObject json = JSON.parseObject(str);
        if (!json.containsKey("code")) {
          logger.info("北京客户端收到非 JSON 化数据");
        }

        int code = json.getIntValue("code");
        if (code == VoteEnum.PREPARE.getCode()) {
          // 校验哈希
          VoteInfo voteInfo = JSON.parseObject(str, VoteInfo.class);
          if (!voteInfo.getHash().equals(SimpleMerkleTree.
              getTreeNodeHash(voteInfo.getList()))) {
            logger.info("北京客户端收到错误的 JSON 化数据");
            return;
          }
```

```
            // 校验成功，发送下一个状态的数据
            VoteInfo vi = createVoteInfo(VoteEnum.COMMIT);
            BlockPacket bp = new BlockPacket();
            bp.setBody(JSON.toJSONString(vi).
                getBytes(BlockPacket.CHARSET));
            Tio.send(channelContext, bp);

            logger.info("北京客户端发送到服务端的 pbft 消息：" +
                JSON.toJSONString(vi));
          }
        }

        return;
    }

    /**
     * 此方法如果返回 null，则框架层面不会发心跳；如果返回非 null，则框架层面会定时
       发本方法返回的消息包
     */
    @Override
    public BlockPacket heartbeatPacket() {
        return heartbeatPacket;
    }

    // 根据 VoteEnum 构建对应状态的 VoteInfo
    private VoteInfo createVoteInfo(VoteEnum ve) {
        VoteInfo vi = new VoteInfo();
        vi.setCode(ve.getCode());

        List<String> list = new ArrayList<>();
        list.add("AI");
        list.add("BlockChain");
        vi.setList(list);
        vi.setHash(SimpleMerkleTree.getTreeNodeHash(list));

        return vi;
    }
}
```

如上述代码所示，基于 t-io 的区块链底层 P2P 网络平台的客户端类 BlockChainPbftClientStarter 的 start()方法启动客户端，主要逻辑是初始化 clientGroupContext 和 tioClient，与服务端建立连接后，通过 sendMessage()方法向服务端发送一条消息“tal say hello world to blockchain!”。

为保证 start()方法在服务启动时就能加载，用@PostConstruct 标记使其在服务器加载 bean 的时候运行，并且只会被服务器执行一次。同时，为了保证客户端后于服务端加载，用@Order(2)标识了 start()。

基于 t-io 的区块链底层 P2P 网络平台的客户端 Handler 的类 BlockChainPbftClientAioHandler 提供了消息的解码方法 decode()、消息的编码方法 encode()及处理信息逻辑的 handler()方法。其中 handler()方法的逻辑是：

1）当服务端收到了来自客户端的非 JSON 化数据，且数据为“北京服务端开始区块入库啦”，则说明 PBFT 阶段一致性已经达成，此时不处理该消息。否则，进行下一步处理。

2）当服务端收到来自客户端的非 JSON 化数据，且数据不为“北京服务端开始区块入库啦”，则说明当前不是 PBFT 阶段，这里仅将收到的消息回传给服务端。

3）当服务端收到的是 JSON 化数据，则说明目前是 PBFT 阶段。服务端先校验消息的有效性，校验通过后再根据消息的状态码进行相应的 PBFT 逻辑处理。

若服务端收到的是 prepare 消息，则将消息的状态置为 Commit，并将消息发送给服务端。

在名为 demo 的工程根目录运行 maven 命令：mvn clean package，将 demo 工程打包成 demo.jar，随后切换到 target 目录下，执行命令 java -jar demo.jar，程序执行效果如图 5-2 所示。

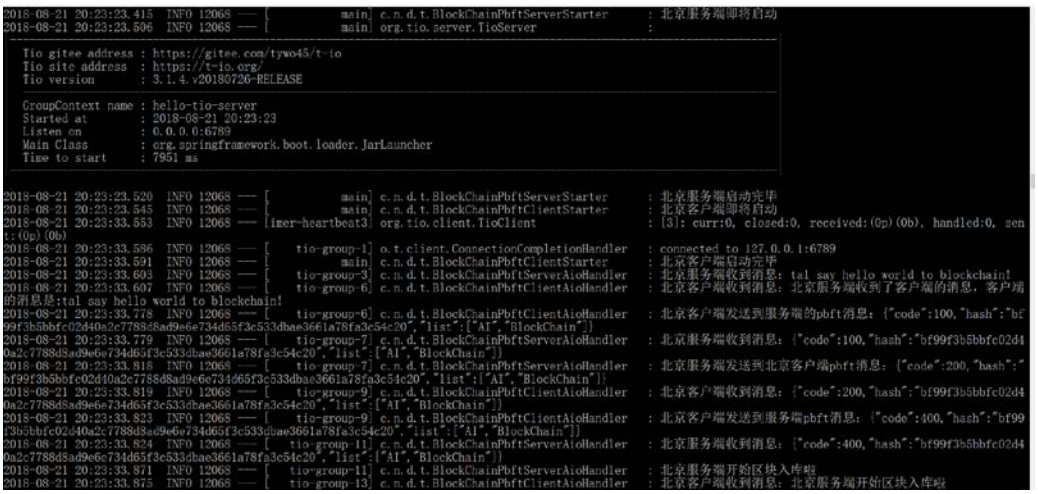


图 5-2　代码运行日志图

5.7　小结

本章介绍了另一个区块链系统的基石——共识算法。

本章从分布式一致性算法开始，逐步讲解了分布式应用开发中常用的中间件内部的分布式一致性算法的应用，最后介绍了区块链中共识算法的实现，最后分别基于 WebSocket 和 t-io 实现了联盟链中常用的 PBFT 共识算法。

在本书中，笔者推荐用简单易上手的基于 WebSocket 方法来实现。

第 6 章
区块设计

莫愁前路无知己
天下谁人不识君

区块是区块链中核心的数据结构。好比行驶在马路上的汽车，虽然大小、颜色、形态各异，但核心的组成是相近的。与之类似，虽然各个公链、联盟链的区块设计不同，但区块的核心字段都是相似的。本章从比特币、以太坊、超级账本（Hyperledger）的区块设计开始介绍，随后介绍 Java 版区块链中的区块应如何设计。

6.1　比特币的区块设计

众所周知，比特币系统的源代码是开源的。比特币的区块设计在 block.h 文件中，由文件中的代码逻辑可知，比特币的区块由 CBlock 类实现，CBlock 的主体字段是 CBlockHeader。

CBlockHeader 的代码结构如下所示：

```
class CBlockHeader
    {
    public:
    // header
    int32_t nVersion;
    uint256 hashPrevBlock;
    uint256 hashMerkleRoot;
    uint32_t nTime;
    uint32_t nBits;
    uint32_t nNonce;

    CBlockHeader()
    {
    SetNull();
    }
  //其他代码略
```

CBlockHeader 的字段释义如表 6-1 所示。

表 6-1　CBlockHeader 的字段释义

字段名称	类型	说明
nVersion	int32	版本号
hashPrevBlock	uint256	上一个区块的哈希值
hashMerkleRoot	uint256	交易列表的 Merkle 树根节点哈希值

续表

字段名称	类型	说明
nTime	uint32	当前时间戳
nBits	uint32	当前挖矿难度，nBits 越小，难度越大
nNonce	uint32	随机数 Nonce 值

CBlock 的代码结构如下所示：

```
class CBlock :
public CBlockHeader
 {
 public:
    // network and disk
    std::vector<CTransactionRef> vtx;


    // memory only
    mutable bool fChecked;


    CBlock()
    {
        SetNull();
    }


    CBlock(const CBlockHeader &header)
    {
        SetNull();
        *(static_cast<CBlockHeader*>(this)) = header;
    }
  //其他代码略
```

CBlock 的字段释义如下：

std::vector<CTransactionRef>代指交易列表，CTransactionRef 为交易内容实体，CBlockHeader 为区块的 Header 设计。

6.2　以太坊的区块设计

以太坊系统的源代码也是开源的，以太坊的区块设计在 block.go 文件中。由文件中的代码逻辑可知，以太坊的区块由 block.go 类实现。从代码中可以看出，以太坊区块由 Header 和 Body 两部分组成。

其中，Header 的代码结构如下所示：

```
type Header struct {
    ParentHash  common.Hash   'json:"parentHash"    gencodec:"required"'
    UncleHash   common.Hash   'json:"sha3Uncles"    gencodec:"required"'
    Coinbase    common.Address 'json:"miner"        gencodec:"required"'
    Root        common.Hash   'json:"stateRoot"     gencodec:"required"'
    TxHash     common.Hash  'json:"transactionsRoot" gencodec:"required"'
    ReceiptHash  common.Hash  'json:"receiptsRoot"  gencodec:"required"'
    Bloom      Bloom        'json:"logsBloom"       gencodec:"required"'
    Difficulty  *big.Int      'json:"difficulty"    gencodec:"required"'
    Number     *big.Int     'json:"number"          gencodec:"required"'
    GasLimit    uint64       'json:"gasLimit"      gencodec:"required"'
    GasUsed     uint64       'json:"gasUsed"       gencodec:"required"'
    Time       *big.Int     'json:"timestamp"       gencodec:"required"'
    Extra      []byte       'json:"extraData"      gencodec:"required"'
    MixDigest   common.Hash   'json:"mixHash"       gencodec:"required"'
    Nonce      BlockNonce    'json:"nonce"          gencodec:"required"'
}
```

Header 的主要字段释义如表 6-2 所示。

表 6-2　Header 的主要字段释义

字段名称	类　　型	说　　明
ParentHash	common.Hash	父区块哈希
UncleHash	common.Hash	叔区块哈希，具体为 Body 中 Uncles 数组的 RLP 哈希值
Coinbase	common.Address	矿工地址
Root	common.Hash	StateDB 中 state Trie 根节点的 RLP 哈希值
TxHash	common.Hash	Block 中 tx Trie 根节点的 RLP 哈希值
ReceiptHash	common.Hash	Block 中 Receipt Trie 根节点的 RLP 哈希值
Difficulty	int	区块难度，即当前挖矿难度

续表

字段名称	类　型	说　明
Number	int	区块序号，即父区块 Number+1
GasLimit	uint6	区块内所有 Gas 消耗的理论上限，创建时指定，由父区块 GasUsed 和 GasLimit 计算得出
GasUsed	uint64	区块内所有 Transaction 执行时消耗的 Gas 总和
Time	int	当前时间戳
Nonce	BlockNonce	随机数 Nonce 值

在上述字段中，叔区块比较特殊。叔区块是一个孤立的块。由于以太坊成块速度较快，因此会产生孤块。以太坊会对发现孤块的矿工给予回报，激励矿工在新块中引用孤块，引用孤块会使主链变重。一般在以太坊中，主链是指最重的链。

Body 的代码结构如下所示：

```
type Body struct {
    Transactions []*Transaction
    Uncles  []*Header
}
```

Body 的成员变量释义如下：

- Transactions 表示交易列表。
- Uncles 表示引用的叔区块列表。

6.3　Hyperledger 的区块设计

Hyperledger_系统的源代码同样是开源的。Hyperledger_系统包含的 fabric、composer、sawtooth-core、indy-node、burrow 和 iroha 6 个子项目。

Hyperledger_系统的区块设计在 fabric 子项目的 block.go 和 common.proto 文件中，由文件中的代码逻辑可知，Hyperledger 区块由 Header、Body 和 Metadata 三部分组成。

其中，Block 的代码结构如下所示：

```
Block
message Block {
```

```
        BlockHeader header = 1;
        BlockData data = 2;
        BlockMetadata metadata = 3;
    }
message BlockHeader {
        uint64 number = 1; // 区块高度
        bytes previous_hash = 2; // 前一区块的哈希值
        bytes data_hash = 3; //交易 Merkle 树根节点生成的哈希值
    }

message BlockData {
        repeated bytes data = 1;
    }

message BlockMetadata {
        repeated bytes metadata = 1;
    }
```

Block 的字段释义如下：

Number 为区块编号，previous_hash 为指向前一个区块的指针，data_hash 为当前区块的哈希值。

6.4　Java 版区块设计

根据前面的介绍，读者应该对目前主流区块链系统的区块设计有了一些了解。

如果对 HTTP 比较清楚，就会发现区块的结构设计和 Request/HTTP Response 结构设计类似，如 HTTP Request 也是由三部分组成的，分别是 Request Line、Request Header 和 Request Body；HTTP Response 也是由三部分组成的，分别是 Response Line、Response Header 和 Response Body。类似的区块一般也由 Header 和 Body 组成。

下面我们借鉴这些区块链系统的区块设计方案来设计 Java 版区块。Java 版区块 Block 由 BlockHeader、BlockBody 和 blockHash 组成，其中，BlockHeader 指的是区块头，BlockBody 指的是区块 body，blockHash 指的是区块的哈希，由区块所有内容，也就是 BlockHeader 和 BlockBody 根据 SHA256 计算得到。

其中，Block 的代码设计如下：

```
package com.niudong.demo.blockchain.block;

import cn.hutool.crypto.digest.DigestUtil;

/**
 * 区块
 *
 * @author niudong.
 */
public class Block {
  /**
   * 区块头
   */
  private BlockHeader blockHeader;
  /**
   * 区块 body
   */
  private BlockBody blockBody;
  /**
   * 该区块的哈希
   */
  private String blockHash;

  /**
   * 根据该区块所有属性计算 SHA256
   *
   * @return sha256hex
   */
  private String getBlockHash() {
    return DigestUtil.sha256Hex(blockHeader.toString() +
        blockBody.toString());
  }

  public BlockHeader getBlockHeader() {
    return blockHeader;
  }

  public void setBlockHeader(BlockHeader blockHeader) {
    this.blockHeader = blockHeader;
  }

  public BlockBody getBlockBody() {
```

```
    return blockBody;
  }

  public void setBlockBody(BlockBody blockBody) {
    this.blockBody = blockBody;
  }

/**
   * 根据该区块所有属性计算 SHA256
   *
   * @return sha256hex
   */

  public String getBlockHash() {
      return DigestUtil.sha256Hex(blockHeader.toString() +
             blockBody.toString());

  }

  public void setBlockHash(String blockHash) {
    this. blockHash = blockHash;
  }

}
```

BlockBody 主要用于存取交易信息列表，代码设计如下：

```
package com.niudong.demo.blockchain.block;

import java.util.List;

/**
 * 区块 body，里面存放交易的数组
 *
 * @author niudong.
 */
public class BlockBody {
  private List<ContentInfo> contentInfos;

  public List<ContentInfo> getContentInfos () {
    return contentInfos;
  }
```

```
    public void setContentInfos (List< ContentInfo > contentInfos) {
      this. contentInfos = contentInfos;
    }
}
```

交易信息 ContentInfo 包含时间戳字段、交易签名信息、交易信息的哈希值、交易发起方的公钥（私钥加密，公钥解密）和交易信息内容的 JSON 化字符串。这里交易信息内容采用 String 字符串，可以承载各个交易实体的 JSON 化数据。JSON 化数据使用起来不仅方便，而且通用性更强。

区块中承载的交易信息 ContentInfo 代码设计如下：

```
package com.niudong.demo.blockchain.block;

/**
 * 区块 body 内的一条内容实体
 *
 * @author niudong.
 */
public class ContentInfo {
  /**
   * 新的 JSON 内容
   */
  private String jsonContent;
  /**
   * 时间戳
   */
  private Long timeStamp;
  /**
   * 公钥
   */
  private String publicKey;
  /**
   * 签名
   */
  private String sign;
  /**
   * 该操作的哈希
   */
  private String hash;
```

```
  public String getJson() {
    return jsonContent;
  }

  public void setJsonContent(String jsonContent) {
    this. jsonContent = jsonContent;
  }

  public String getPublicKey() {
    return publicKey;
  }

  public void setPublicKey(String publicKey) {
    this.publicKey = publicKey;
  }

  public Long getTimeStamp() {
    return timeStamp;
  }

  public void setTimeStamp(Long timeStamp) {
    this.timeStamp = timeStamp;
  }

  public String getSign() {
    return sign;
  }

  public void setSign(String sign) {
    this.sign = sign;
  }

  public String getHash() {
    return hash;
  }

  public void setHash(String hash) {
    this.hash = hash;
  }
}
```

BlockHeader 包含当前区块运行的版本、上一区块的哈希值、Merkle 树根节点哈希值、区块的序号、时间戳、32 位随机数 Nonce、交易信息的哈希集合等。

BlockHeader 的设计如下：

```
package com.niudong.demo.blockchain.block;

import java.util.List;

/**
 * 区块头
 *
 * @author niudong.
 */
public class BlockHeader {
  /**
   * 版本号
   */
  private int version;
  /**
   * 上一区块的哈希
   */
  private String hashPreviousBlock;
  /**
   * Merkle 树根节点哈希值
   */
  private String hashMerkleRoot;
  /**
   * 生成该区块的公钥
   */
  private String publicKey;
  /**
   * 区块的序号
   */
  private int number;
  /**
   * 时间戳
   */
  private long timeStamp;
  /**
   * 32 位随机数
   */
```

```
    private long nonce;
    /**
     * 该区块里每条交易信息的哈希集合，按顺序来的，通过该哈希集合能算出根节点哈希值
     */
    private List<String> hashList;

    public int getVersion() {
      return version;
    }

    public void setVersion(int version) {
      this.version = version;
    }

    public String getHashPreviousBlock() {
      return hashPreviousBlock;
    }

    public void setHashPreviousBlock(String hashPreviousBlock) {
      this.hashPreviousBlock = hashPreviousBlock;
    }

    public String getHashMerkleRoot() {
      return hashMerkleRoot;
    }

    public void setHashMerkleRoot(String hashMerkleRoot) {
      this.hashMerkleRoot = hashMerkleRoot;
    }

    public String getPublicKey() {
      return publicKey;
    }

    public void setPublicKey(String publicKey) {
      this.publicKey = publicKey;
    }

    public int getNumber() {
      return number;
    }
```

```
    public void setNumber(int number) {
      this.number = number;
    }

    public long getTimeStamp() {
      return timeStamp;
    }

    public void setTimeStamp(long timeStamp) {
      this.timeStamp = timeStamp;
    }

    public long getNonce() {
      return nonce;
    }

    public void setNonce(long nonce) {
      this.nonce = nonce;
    }

    public List<String> getHashList() {
      return hashList;
    }

    public void setHashList(List<String> hashList) {
      this.hashList = hashList;
    }
  }
```

6.5 小结

本章主要介绍了区块链中的核心数据结构——区块。分别从比特币、以太坊、超级账本的维度，先后介绍了不同区块链系统中的区块设计，随后总结了三类区块链系统区块设计的共同点，并引出了 Java 版区块链中区块设计的方法。

第 7 章
区块存储

海纳百川
有容乃大

上一章介绍了区块设计，区块借助区块链中的 P2P 网络，在区块链系统的各个节点中穿梭，最终在各个节点落地持久化。

与分布式服务研发过程中常见的 MySQL 集群、数据库分库分表、NoSQL 型数据库集群的使用习惯不同，区块链的存储系统设计强调本地高效存储。目前市场上主流的区块链系统有比特币系统、Ripple 币系统、以太坊系统和超级账本，这些区块链中所使用的存储技术各不相同。

7.1 区块存储技术

比特币、Ripple 币、以太坊和超级账本的存储设计各不相同，各区块链系统所使用的存储方案分别介绍如下。

比特币存储是系统由普通文件和 LeveldB 数据库组成的。普通文件用于存储区块数据，LevelDB 数据库用于存储区块元数据。区块数据每个文件的大小是 128MB。每个区块的数据（区块头和区块里的所有交易）都会序列成字节码的形式写入 dat 文件中。

比特币存储系统的缺点主要在于区块数据文件大小受限制，当超过 128MB 时要分割文件，在文件中检索数据相对慢一些。

Ripple 币存储系统是由 SQLite 关系型数据库和 RocksDB 数据库组成的，其中，关系型数据库用来存储区块头信息和每笔交易的具体信息，RocksDB 数据库主要存储区块头、交易和状态表序列化后的数据。

Ripple 币存储系统的缺点在于，在 RocksDB 存储的 value 数据字段多而复杂。

以太坊存储系统（区块）主要由区块头和交易组成。区块在存储过程中分别将区块头和交易体经过 RLP 编码后存入 LevelDB 数据当中。以太坊在数据存储的过程中，每个 value 对应的 key 都有相对应的前缀，不同类型的 value 对应不同的前缀。同时，以太坊系统允许日志跟踪各种交易和信息。

以太坊存储系统的缺点是使用了文件系统，日志文件大小的阈值默认为 1MB，文件个数较多，文件管理较复杂。

超级账本的存储系统和比特币一样，也是由普通文件和 KV 数据库

（LevelDB/CouchDB）组成。在超级账本中，每个 channel 对应一个账本目录，账本目录是由 blockfile_000000、blockfile_000001 命名格式的文件名组成的。区块数据每个文件的大小是 64MB。每个区块的数据（区块头和区块里的所有交易）都会序列成字节码的形式写入 blockfile 文件中。

超级账本存储系统的缺点在于区块数据文件大小受限制，当超过 64MB 时要分割文件，在文件中检索数据相对慢一些。

至此，比特币系统、Ripple 币系统、以太坊系统和 Hyperledger Fabric 的存储设计简要介绍完毕。下面将分别介绍文件存储、SQLite 存储、LevelDB 存储、RocksDB 存储和 CouchDB 存储的 Java 代码实现。

7.2　用 Java 实现文件存储

文件存储数据是 Java 开发中常用的方式之一，我们可以借助 Guava 来简化开发。

Guava 是 Google 开源的 Java 库，其中包含了 Google 内部很多项目使用的核心库。Guava 是为了方便编码，并减少编码错误而设立的，它提供了用于集合、缓存、并发、字符串处理、I/O 流处理的多种 API 接口，同时支持原语，并提供了注解的使用方式。

由于本章介绍的是存储相关的内容，因此仅仅介绍涉及文件操作相关的 API。

7.2.1　Guava 文件操作

Guava 中文件操作相关的内容主要有：

- Files 类：执行诸如移动、复制文件或读取文件内容到一个字符串集合的操作。
- Closer 类：提供了 Closeable 实例被正确、简捷关闭的方法。
- ByteSource 和 CharSource 类：提供不可变的输入流（Input）和读（Reader）。
- ByteSink 和 CharSink 类：提供不可变的输出流（Output）和写（Writer）。
- CharStreams 和 ByteStreams 类：为读、写、输入流、输出流提供了静态的实用方法，如限制输入流大小等 API。

- BaseEncoding 类：用于字节编码，提供编码和解码字节序列，以及 ASCII 字符的方法。

在区块链的文件存储中主要使用 Files 类，下面详细介绍 Files 类的 API 使用。

（1）touch 方法：创建或者更新文件的时间戳。

（2）createTempDir()方法：创建临时目录。

（3）Files.createParentDirs(File)：创建父级目录。

（4）getChecksum(File)：获得文件的 checksum。

（5）hash(File)：获得文件的哈希值。

（6）map 系列方法：获得文件的内存映射。

（7）getFileExtension(String)钉钉：获得文件的扩展名。

（8）getNameWithoutExtension(String file)方法：获得不带扩展名的文件名。

（9）Files.equal(File,File)方法：比较两个文件的内容是否完全一致。

（10）Files.copy()方法：用于复制文件。使用方法如下：

```
   File sourceFile = new File(sourceFileName);
   File targetFile = new File(targetFileName);
   try {
      Files.copy(sourceFile, targetFile);
   } catch (IOException fileIoEx) {
   System.out.println( "ERROR trying to copy file '" + sourceFileName +
"' to file '" + targetFileName + "' - " + fileIoEx.toString());
    }
```

（11）Files.move()方法：用于移动文件。使用方法如下：

```
   try {
        File from = new File(fromFileName);
        File to = new File(toFileName);
        Files.move(from, to);
     } catch (Exception e) {
        e.printStackTrace();
     }
```

（12）Files.write()方法：用于向文件中写入内容。使用方法如下：

```
   File newFile = new File(fileName);
   try {
      Files.write(contents.getBytes(), newFile);
      } catch (IOException fileIoEx) {
      System.out.println( "ERROR trying to write to file '" + fileName +
"' - " + fileIoEx.toString());
   }
```

（13）Files.readLines()方法：用于读取文件内容。使用方法如下：

```
   String testFilePath = "d:\\niudong\\readtest.txt";
   File testFile = new File(testFilePath);
   List<String> lines = Files.readLines(testFile, Charsets.UTF_16);
   for (String line : lines) {
      System.out.println(line);
   }
```

7.2.2　Guava 实现文件存储

使用 Guava 时，在项目的 POM 文件中加入 guava 依赖，配置代码如下。

```
   <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>21.0</version>
   </dependency>
```

本节我们基于 Guava 实现类似比特币的文件存储，主要逻辑如下：

- 写入的内容是基于一个字符串列表生成的 Merkle 树根节点的哈希值。
- 字符串列表由四部分组成，其中三部分为固定内容，分别是“AI”、“BlockChain”和“BrainScience”，第四部分由 generateVCode()方法生成，generateVCode()方法基于可配置的 length 长度生成随机数字串。
- 文件内容写入时采用向文件尾部添加内容的方式，这里我们提供了两种方式：基于 Java 原生的 FileWriter 方式和基于 Guava 的方式（append 方法）。

这里我们规定区块链文件大小为 1 MB，即 1024KB。当写入文件大小超过 1MB

时，则创建新的文件。正在写入的文件名后缀为“loging”，文件大小超过 1MB 的部分文件名后缀为“log”。

模拟区块链系统中内容写入文件时的步骤如下。

步骤 1 查看当前目录下是否有正在写入的 loging 文件，有则继续使用，无则创建。

步骤 2 在后缀为“loging”的文件中模拟写入区块链内容。

具体代码如下：

```
package com.niudong.demo.util;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import com.google.common.base.Charsets;
import com.google.common.io.Files;

/**
 * 模拟区块链文件存储方式：Guava 版
 *
 * @author 牛冬
 *
 */
public class FileStoreUtil {
  // 定义区块链文件大小
  private static final int FILE_SIZE = 1024;// 单位 KB

  // 将文件内容写入目标文件：Guava 方式
  public static void writeIntoTargetFile(String targetFileName, String
      content) {
    File newFile = new File(targetFileName);
    try {
      Files.write(content.getBytes(), newFile);
    } catch (Exception e) {
      e.printStackTrace();
    }
```

```
    }

    // 将文件内容向后追加写入目标文件：FileWriter 方式
    public static void appendToTargetFile(String targetFileName, String
        content) {
      try {
        // 打开一个写文件器，构造函数中的第二个参数 true，表示以追加形式写文件
        FileWriter writer = new FileWriter(targetFileName, true);
        writer.write(content);
        writer.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    // 将文件内容向后追加写入目标文件：Guava 方式
    public static void appendToTargetFileByGuava(String targetFileName,
        String content) {
      File file = new File(targetFileName);
      try {
        Files.append(content, file, Charsets.UTF_8);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    // 模拟区块链内容写入文件
    public static void writeIntoFile(String content) {
      try {
        /**
         * 步骤 1 查看当前目录下是否有正在写入的 loging 文件，有则继续使用，无则创建
         */
        File root = new File(".//");
        // 获取当前文件下的所有文件
        File[] files = root.listFiles();
        if (files == null) {
          // 如果根目录下没有任何文件则创建新的文件
          String targetFileName = ".//blockchain-" +
              System.currentTimeMillis() + ".loging";
          appendToTargetFileByGuava(targetFileName, content);
          return;
        }
```

```
        // 如果根目录下有文件则寻找是否存在后缀为 loging 的文件
        boolean has = false;
        for (File file : files) {
          String name = file.getName();
          if (name.endsWith(".loging") && name.startsWith("blockchain-")) {
            // 有则继续写入
            System.out.println(file.getPath());
            appendToTargetFileByGuava(file.getPath(), content);
            has = true;

            // 写入后如果文件大小超过固定大小，则将 loging 后缀转为 log 后缀
            if (file.length() >= FILE_SIZE) {
              String logPath = file.getPath().replace("loging", "log");
              File log = new File(logPath);
              Files.copy(file, log);
              // 删除已经写满的 loging 文件
              file.delete();
            }
          }
        }

        // 无则创建新的文件
        if (!has) {
          String targetFileName = ".//blockchain-" +
              System.currentTimeMillis() + ".loging";
          appendToTargetFileByGuava(targetFileName, content);
          return;
        }

      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    //模拟区块链内容的写入
    public static void writeIntoBlockFile() {
      List<String> list = new ArrayList<>();
      list.add("AI");
      list.add("BlockChain");
      list.add("BrainScience");
```

```
        for (int i = 0; i < 20; i++) {
          list.add(generateVCode(6));
          writeIntoFile(SimpleMerkleTree.getTreeNodeHash(list) + "\n");
        }
      }

      // 根据 length 长度生成数字符串
      public static String generateVCode(int length) {
        Long vCode = new Double((Math.random() + 1) * Math.pow(10, length
            - 1)).longValue();
        return vCode.toString();
      }

      public static void main(String[] args) {
        writeIntoBlockFile();
      }
    }
```

在 IDE 中运行上述代码，可以在工程根目录中看到生成的文件名后缀为“loging”和“log”的文件，如图 7-1 所示。

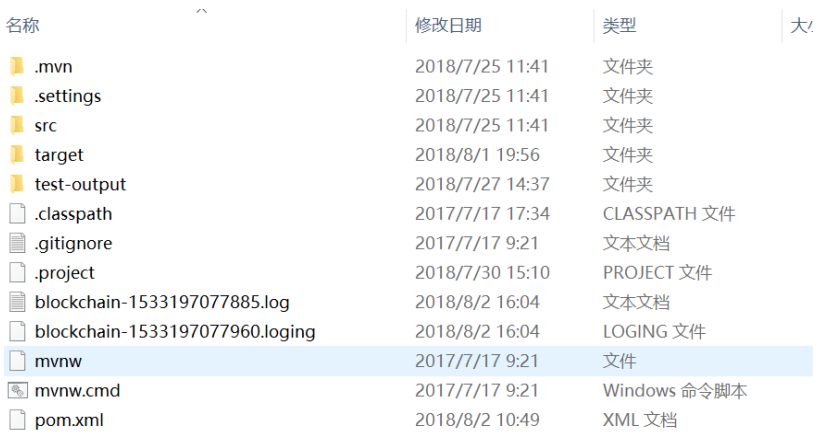


图 7-1　工程根目录文件列表

其中“log”后缀的文件大小如图 7-2 所示。

图 7-2　log 后缀的文件大小

7.3　用 Java 实现 SQLite 存储

7.3.1　SQLite 介绍

SQLite 是目前世界上使用最多的数据库引擎。

不同于常用的 MySQL 或 Oracle 数据库，SQLite 是一个进程内的库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。数据库零配置是 SQLite 区别于其他数据库的主要特点。

与 MySQL 或 Oracle 数据库类似之处在于，SQLite 引擎不是一个独立的进程，可以按应用程序需求进行静态或动态连接，SQLite 可直接访问其存储文件。

什么场景适用于 SQLite 呢？比如不需要一个单独的服务器进程或操作的系统（无服务器）的场景。因为 SQLite 不需要配置，所以使用 SQLite 并不需要像 MySQL 那样进行安装或管理。

此外，SQLite 非常小，是轻量级的，完全配置时小于 400KB，省略可选功能配置时小于 250KB。SQLite 无须任何外部的依赖，能实现自给自足。

与 MySQL 或 Oracle 数据库一样，SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问。SQLite 支持 SQL92（SQL2）标准的大多数查询语言的功

能，为了方便用户使用，SQLite 提供了简单和易于使用的 API。

SQLite 的系统适配性良好，不仅可以在 UNIX、Linux、Windows、Mac OS-X 系统中运行，还能在 Android、iOS 系统中运行。

7.3.2　SQLite 的使用

我们基于 Spring Boot 2.0 使用 SQLite，在项目的 POM 文件中添加 SQLite 配置如下：

```
<!-- 用于配置 SQLite 依赖 -->
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
</dependency>
<!-- 用于配置数据源 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
</dependency>
```

同时，在 application.properties 中配置时 SQLite 参数如下：

```
sqlite.dbName: sqlite.db
```

我们可以对 sqlite.dbName 的值进行个性化配置，sqlite.dbName 的值用于指定初始化 SQLite 时生成的 SQLite 数据库名称。

配置完 POM 文件和 application.properties 文件后，需要配置数据源 DataSource 信息。我们用 DataSourceConfiguration 类实现对数据源 DataSource 信息的配置，代码设计如下：

```
package com.niudong.blockchain.core.sqlite.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.jdbc.
   DataSourceBuilder;
//import org.springframework.boot.autoconfigure.jdbc.
   DataSourceAutoConfiguration;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import org.sqlite.SQLiteDataSource;

import javax.sql.DataSource;


/**
 * 配置 SQLite 数据库的 DataSource
 *
 * @author niudong
 */
@Configuration
public class DataSourceConfiguration {
  @Value("${sqlite.dbName}")
  private String dbName;

  @Bean(destroyMethod = "", name = "EmbeddeddataSource")
  public DataSource dataSource() {
    //spring boot 1.5.4

    DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
    dataSourceBuilder.driverClassName("org.sqlite.JDBC");
    dataSourceBuilder.url("jdbc:sqlite:" + dbName);
    dataSourceBuilder.type(SQLiteDataSource.class);
    return dataSourceBuilder.build();


  }

}
```

如上述代码所示，其中，dbName 的值来自 application.properties 文件中的 sqlite.dbName 配置。

当启动程序时，在 Tomcat 的 bin 目录下会出现 sqlite.dbName 名称的文件，里面存储的是表结构及数据。

本书基于 JPA 来操作 SQLite，JPA 在代码中的配置如下：

```
package com.niudong.blockchain.core.sqlite.config;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.autoconfigure.orm.jpa.JpaProperties;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.
    EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.
    LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.annotation.
    EnableTransactionManagement;

import javax.annotation.Resource;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Map;

/**
 * @author niudong
 */
@Configuration
@EnableJpaRepositories(basePackages = "com. niudong.blockchain.base.dao",
    transactionManagerRef = "jpaTransactionManager",
    entityManagerFactoryRef = "localContainerEntityManagerFactoryBean")
@EnableTransactionManagement
public class JpaConfiguration {
  @Resource
  private JpaProperties jpaProperties;

  @Autowired
  @Bean
  public JpaTransactionManager jpaTransactionManager(
      @Qualifier(value = "EmbeddeddataSource") DataSource dataSource,
      EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager jpaTransactionManager = new
        JpaTransactionManager();
    jpaTransactionManager.setEntityManagerFactory
        (entityManagerFactory);
    jpaTransactionManager.setDataSource(dataSource);

    return jpaTransactionManager;
```

```
    }

    @Autowired
    @Bean
    LocalContainerEntityManagerFactoryBean
localContainerEntityManagerFactoryBean(
        @Qualifier(value = "EmbeddeddataSource") DataSource dataSource,
        EntityManagerFactoryBuilder builder) {
      return builder.dataSource(dataSource).packages("com.
niudong.blockchain.base.dao.entity")
          .properties(getVendorProperties(dataSource)).build();
    }

    private Map<String, String> getVendorProperties(DataSource
dataSource) {
      return jpaProperties.getHibernateProperties(dataSource);
    }

  }
```

同时，在 application.properties 中配置 JPA 参数如下：

```
  #jpa
  spring.jpa.show-sql = true
  spring.jpa.ddl-auto= update
  spring.jpa.generate-ddl=true
  spring.jpa.database-platform=com.niudong.blockchain.core.sqlite.con
fig.SQLiteDialect
```

其中，spring.jpa.show-sql 字段用于配置是否需要在控制台输出 SQL 语句，在本书中我们配置其值为 true，即需要在控制台输出 SQL 语句。

spring.jpa.ddl-auto 字段用于配置 ddl-auto 信息。当配置为 update 时，每次运行程序，若没有相关表格，则会新建表格；若存在表格，则表内数据不会被清空，只进行更新操作。同时，ddl-auto 还可以配置为 create、create-drop 等。配置为 create 时，每次运行程序，若没有表格，则会新建表格；若表格存在，则表中数据会被清空。配置为 create-drop 时，程序结束时会清空表格内容。

spring.jpa.generate-ddl 字段用于配置是否生成 ddl 语句，可配置为 true 或 false。这里配置为 true，即生成 ddl 语句。

spring.jpa.database-platform 字段用于配置数据库的方言，这里配置为自定义的 SQLiteDialect。SQLiteDialect 类的代码设计如下：

```
package com.niudong.blockchain.core.sqlite.config;

import com.niudong.blockchain.core.sqlite.config.identity.
   SQLiteDialectIdentityColumnSupport;
import org.hibernate.JDBCException;
import org.hibernate.ScrollMode;
import org.hibernate.dialect.Dialect;
import org.hibernate.dialect.function.*;
import org.hibernate.dialect.identity.IdentityColumnSupport;
import org.hibernate.dialect.pagination.AbstractLimitHandler;
import org.hibernate.dialect.pagination.LimitHandler;
import org.hibernate.dialect.pagination.LimitHelper;
import org.hibernate.dialect.unique.DefaultUniqueDelegate;
import org.hibernate.dialect.unique.UniqueDelegate;
import org.hibernate.engine.spi.RowSelection;
import org.hibernate.exception.DataException;
import org.hibernate.exception.JDBCConnectionException;
import org.hibernate.exception.LockAcquisitionException;
import org.hibernate.exception.spi.SQLExceptionConversionDelegate;
import org.hibernate.exception.spi.
   TemplatedViolatedConstraintNameExtracter;
import org.hibernate.exception.spi.ViolatedConstraintNameExtracter;
import org.hibernate.internal.util.JdbcExceptionHelper;
import org.hibernate.mapping.Column;
import org.hibernate.type.StandardBasicTypes;

import java.sql.SQLException;
import java.sql.Types;

/**
 * SQLite 的方言，目的是使用 Hibernate jpa 操作 SQLite
 *
 * @author niudong
 */
public class SQLiteDialect extends Dialect {
  private final UniqueDelegate uniqueDelegate;

  public SQLiteDialect() {
    registerColumnType(Types.BIT, "boolean");
```

```
        registerColumnType(Types.FLOAT, "float");
        registerColumnType(Types.DOUBLE, "double");
        registerColumnType(Types.DECIMAL, "decimal");
        registerColumnType(Types.CHAR, "char");
        registerColumnType(Types.LONGVARCHAR, "longvarchar");
        registerColumnType(Types.TIMESTAMP, "datetime");
        registerColumnType(Types.BINARY, "blob");
        registerColumnType(Types.VARBINARY, "blob");
        registerColumnType(Types.LONGVARBINARY, "blob");

        registerFunction("concat", new VarArgsSQLFunction
           (StandardBasicTypes.STRING, "", "||", ""));
        registerFunction("mod", new SQLFunctionTemplate
           (StandardBasicTypes.INTEGER, "?1 % ?2"));
        registerFunction("quote", new StandardSQLFunction("quote",
           StandardBasicTypes.STRING));
        registerFunction("random", new NoArgSQLFunction("random",
           StandardBasicTypes.INTEGER));
        registerFunction("round", new StandardSQLFunction("round"));
        registerFunction("substr", new StandardSQLFunction("substr",
           StandardBasicTypes.STRING));
        registerFunction("trim", new AbstractAnsiTrimEmulationFunction()
          {
          @Override
          protected SQLFunction resolveBothSpaceTrimFunction() {
            return new SQLFunctionTemplate(StandardBasicTypes.STRING,
               "trim(?1)");
          }

          @Override
          protected SQLFunction resolveBothSpaceTrimFromFunction() {
            return new SQLFunctionTemplate(StandardBasicTypes.STRING,
               "trim(?2)");
          }

          @Override
          protected SQLFunction resolveLeadingSpaceTrimFunction() {
            return new SQLFunctionTemplate(StandardBasicTypes.STRING,
                "ltrim(?1)");
          }

          @Override
          protected SQLFunction resolveTrailingSpaceTrimFunction() {
```

```
        return new SQLFunctionTemplate(StandardBasicTypes.STRING,
            "rtrim(?1)");
      }

      @Override
      protected SQLFunction resolveBothTrimFunction() {
        return new SQLFunctionTemplate(StandardBasicTypes.STRING,
            "trim(?1, ?2)");
      }

      @Override
      protected SQLFunction resolveLeadingTrimFunction() {
        return new SQLFunctionTemplate(StandardBasicTypes.STRING,
            "ltrim(?1, ?2)");
      }

      @Override
      protected SQLFunction resolveTrailingTrimFunction() {
        return new SQLFunctionTemplate(StandardBasicTypes.STRING,
            "rtrim(?1, ?2)");
      }
    });
    uniqueDelegate = new SQLiteUniqueDelegate(this);
  }

  private static final SQLiteDialectIdentityColumnSupport
      IDENTITY_COLUMN_SUPPORT =
      new SQLiteDialectIdentityColumnSupport(new SQLiteDialect());


  @Override
  public IdentityColumnSupport getIdentityColumnSupport() {
    return IDENTITY_COLUMN_SUPPORT;
  }

  private static final AbstractLimitHandler LIMIT_HANDLER = new
      AbstractLimitHandler() {
    @Override
    public String processSql(String sql, RowSelection selection) {
      final boolean hasOffset = LimitHelper.hasFirstRow(selection);
      return sql + (hasOffset ? " limit ? offset ?" : " limit ?");
    }
```

```
    @Override
    public boolean supportsLimit() {
      return true;
    }

    @Override
    public boolean bindLimitParametersInReverseOrder() {
      return true;
    }
  };

  @Override
  public LimitHandler getLimitHandler() {
    return LIMIT_HANDLER;
  }

  @Override
  public boolean supportsLockTimeouts() {
    // may be http://sqlite.org/c3ref/db_mutex.html ?
    return false;
  }

  @Override
  public String getForUpdateString() {
    return "";
  }

  @Override
  public boolean supportsOuterJoinForUpdate() {
    return false;
  }

  @Override
  public boolean supportsCurrentTimestampSelection() {
    return true;
  }

  @Override
  public boolean isCurrentTimestampSelectStringCallable() {
    return false;
  }

  @Override
```

```
    public String getCurrentTimestampSelectString() {
      return "select current_timestamp";
    }

    private static final int SQLITE_BUSY = 5;
    private static final int SQLITE_LOCKED = 6;
    private static final int SQLITE_IOERR = 10;
    private static final int SQLITE_CORRUPT = 11;
    private static final int SQLITE_NOTFOUND = 12;
    private static final int SQLITE_FULL = 13;
    private static final int SQLITE_CANTOPEN = 14;
    private static final int SQLITE_PROTOCOL = 15;
    private static final int SQLITE_TOOBIG = 18;
    private static final int SQLITE_CONSTRAINT = 19;
    private static final int SQLITE_MISMATCH = 20;
    private static final int SQLITE_NOTADB = 26;

    @Override
    public SQLExceptionConversionDelegate
        buildSQLExceptionConversionDelegate() {
      return new SQLExceptionConversionDelegate() {
        @Override
        public JDBCException convert(SQLException sqlException, String
            message, String sql) {
          final int errorCode = JdbcExceptionHelper.
              extractErrorCode(sqlException) & 0xFF;
          if (errorCode == SQLITE_TOOBIG || errorCode == SQLITE_MISMATCH) {
            return new DataException(message, sqlException, sql);
          } else if (errorCode == SQLITE_BUSY || errorCode == SQLITE_LOCKED)
          {
            return new LockAcquisitionException(message, sqlException,
                sql);
          } else if ((errorCode >= SQLITE_IOERR && errorCode <=
                      SQLITE_PROTOCOL)
              || errorCode == SQLITE_NOTADB) {
            return new JDBCConnectionException(message, sqlException,
                sql);
          }

          // returning null allows other delegates to operate
          return null;
        }
      };
```

```
    }

    @Override
    public ViolatedConstraintNameExtracter
       getViolatedConstraintNameExtracter() {
      return EXTRACTER;
    }

    private static final ViolatedConstraintNameExtracter EXTRACTER =
       new TemplatedViolatedConstraintNameExtracter() {
         @Override
         protected String doExtractConstraintName(SQLException sqle)
            throws NumberFormatException {
           final int errorCode =
              JdbcExceptionHelper.extractErrorCode(sqle) & 0xFF;
           if (errorCode == SQLITE_CONSTRAINT) {
             return extractUsingTemplate("constraint ", " failed",
                sqle.getMessage());
           }
           return null;
         }
       };

    @Override
    public boolean supportsUnionAll() {
      return true;
    }

    @Override
    public boolean canCreateSchema() {
      return false;
    }

    @Override
    public boolean hasAlterTable() {
      // Hibernate 框架中定义的方言
      return false;
    }

    @Override
    public boolean dropConstraints() {
      return false;
    }
```

```
@Override
public boolean qualifyIndexName() {
  return false;
}

@Override
public String getAddColumnString() {
  return "add column";
}

@Override
public String getDropForeignKeyString() {
  throw new UnsupportedOperationException(
     "No drop foreign key syntax supported by SQLiteDialect");
}

@Override
public String getAddForeignKeyConstraintString(String
   constraintName, String[] foreignKey,
   String referencedTable, String[] primaryKey, boolean
      referencesPrimaryKey) {
  throw new UnsupportedOperationException("No add foreign key syntax
     supported by SQLiteDialect");
}

@Override
public String getAddPrimaryKeyConstraintString(String
   constraintName) {
  throw new UnsupportedOperationException("No add primary key syntax
     supported by SQLiteDialect");
}

@Override
public boolean supportsCommentOn() {
  return true;
}

@Override
public boolean supportsIfExistsBeforeTableName() {
  return true;
}
```

```
    @Override
    public boolean doesReadCommittedCauseWritersToBlockReaders() {
      return true;
    }

    @Override
    public boolean doesRepeatableReadCauseReadersToBlockWriters() {
      return true;
    }

    @Override
    public boolean supportsTupleDistinctCounts() {
      return false;
    }

    @Override
    public int getInExpressionCountLimit() {
   http://sqlite.org/limits.html#max_variable_number
      return 1000;
    }

    @Override
    public UniqueDelegate getUniqueDelegate() {
      return uniqueDelegate;
    }

    private static class SQLiteUniqueDelegate extends
        DefaultUniqueDelegate {
      public SQLiteUniqueDelegate(Dialect dialect) {
        super(dialect);
      }

      @Override
      public String getColumnDefinitionUniquenessFragment(Column
column) {
        return " unique";
      }
    }

    @Override
    public String getSelectGUIDString() {
      return "select hex(randomblob(16))";
```

```
  }

  @Override
  public ScrollMode defaultScrollMode() {
    return ScrollMode.FORWARD_ONLY;
  }
}
```

SQLiteMetadataBuilderInitializer 代码实现如下：

```
package com.niudong.blockchain.core.sqlite.config;

import org.hibernate.boot.MetadataBuilder;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.spi.MetadataBuilderInitializer;
import org.hibernate.engine.jdbc.dialect.internal.DialectResolverSet;
import org.hibernate.engine.jdbc.dialect.spi.DialectResolver;
import org.jboss.logging.Logger;

/**
 * SQLite 工具
 */
public class SQLiteMetadataBuilderInitializer implements
    MetadataBuilderInitializer {

  private final static Logger logger =
     Logger.getLogger(SQLiteMetadataBuilderInitializer.class);

  @Override
  public void contribute(MetadataBuilder metadataBuilder,
     StandardServiceRegistry serviceRegistry) {
    DialectResolver dialectResolver =
       serviceRegistry.getService(DialectResolver.class);

    if (!(dialectResolver instanceof DialectResolverSet)) {
      logger.warn(
         "DialectResolver '%s' is not an instance of DialectResolverSet,
            not registering SQLiteDialect",
         dialectResolver);
      return;
    }

    ((DialectResolverSet) dialectResolver).addResolver(resolver);
```

```
  }

  static private final SQLiteDialect dialect = new SQLiteDialect();

  static private final DialectResolver resolver = (DialectResolver)
      info -> {
    if (info.getDatabaseName().equals("SQLite")) {
      return dialect;
    }

    return null;
  };
}
```

SQLiteDialectIdentityColumnSupport 的代码如下所示：

```
package com.niudong.blockchain.core.sqlite.config.identity;

import org.hibernate.dialect.Dialect;
import org.hibernate.dialect.identity.IdentityColumnSupportImpl;

/**
 * @author 牛冬
 */
public class SQLiteDialectIdentityColumnSupport extends
    IdentityColumnSupportImpl {
  public SQLiteDialectIdentityColumnSupport(Dialect dialect) {
    super(dialect);
  }

  @Override
  public boolean supportsIdentityColumns() {
    return true;
  }


@Override
public boolean supportsInsertSelectIdentity() {
 return true;
   }


  @Override
```

```
    public boolean hasDataTypeInIdentityColumn() {
      return false;
    }

    @Override
    public String getIdentitySelectString(String table, String column,
        int type) {
      return "select last_insert_rowid()";
    }

    @Override
    public String getIdentityColumnString(int type) {
      return "integer";
    }
  }
```

配置完上述代码后，就可以像普通 JPA 一样使用了。先构建数据库表对应的实体类，随后构建对应的 JPA Repository，即可操作对应表的 CUID（即增删改查）了。

7.4　用 Java 实现 LevelDB 存储

7.4.1　LevelDB 介绍

LealDB 是谷歌公司编写的一个键值（K/V，即 Key/Value）存储库，它提供了从字符串键到字符串值的有序映射。

目前，LevelDB 的 1.2 版本已经能够支持十亿级别的数据量存储了。更难能可贵的是，LevelDB 是单机版的持久化 KV 数据库，具有很高的随机写、顺序读/写性能，但是随机读的性能很一般，换句话说，LevelDB 很适合应用在查询较少但写很多的场景，而区块链恰好就是这样的场景之一。

LevelDB 的特点如下：

（1）键和值可以是任意字节数组。

（2）数据按密钥排序后存储。

（3）调用方可以提供自定义比较函数来重写排序顺序。

（4）基本操作有存放 put(key,value)、获取 get/batch(key)、删除 delete(key)。

（5）可以在一个原子批次中多次改变数据。

（6）用户可以创建一个瞬态快照以便获得一致的数据视图。

（7）在数据上支持前向和后向迭代。

（8）使用快速压缩库自动压缩数据。

（9）外部活动（文件系统操作等）通过虚拟接口进行中继，以便用户可以定制操作系统交互。

在官网，LevelDB 的写性能测试结果统计如下：

```
fillseq      :      1.765 micros/op;62.7 MB/s
fillsync     :      268.409 micros/op;0.4 MB/s (10000 ops)
fillrandom   :      2.460 micros/op;45.0 MB/s
overwrite    :      2.380 micros/op;46.5 MB/s
```

可以看到，LevelDB 随机写入的速度可以达到每秒 40 万条。

在官网，LevelDB 的读性能测试结果统计如下：

```
readrandom   : 16.677 micros/op;  (approximately 60,000 reads per
                                    second)
readseq      :  0.476 micros/op;  232.3 MB/s
readreverse  :  0.724 micros/op;  152.9 MB/s
```

可以看到，LevelDB 顺序读的性能尤为突出，在每秒 230 万条以上。

7.4.2 LevelDB 的使用

在项目中使用 LevelDB 时，需在 POM 文件中增加 LevelDB 相关的配置信息：

```
<dependency>
    <groupId>org.iq80.leveldb</groupId>
    <artifactId>leveldb-api</artifactId>
    <version>0.10</version>
</dependency>
<dependency>
    <groupId>org.iq80.leveldb</groupId>
    <artifactId>leveldb</artifactId>
```

```
        <version>0.10</version>
    </dependency>
```

在 mvnresponsity 网站中可查得 LevelDB-API 和 LevelDB 的最新版本均为 0.10，如图 7-3 和图 7-4 所示。

图 7-3　mvnresponsity 网站 LevelDB 版本列表

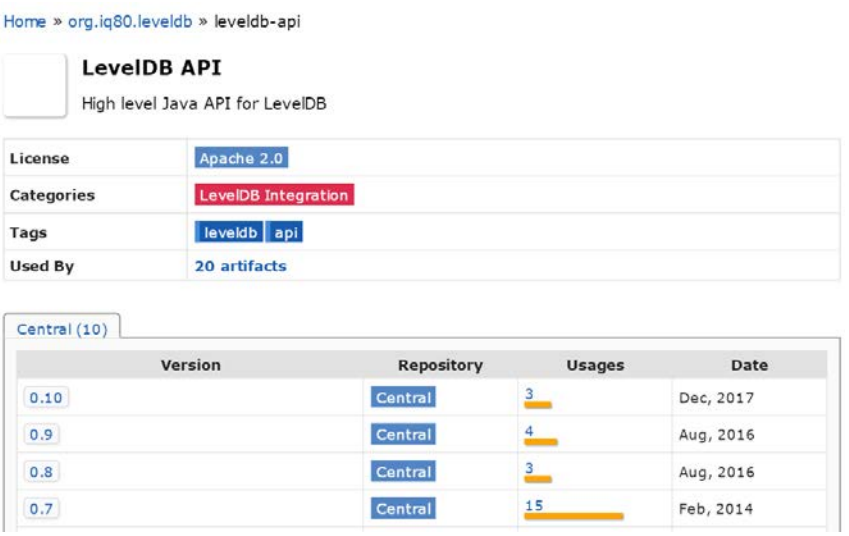


图 7-4　mvnresponsity 网站 LevelDB-API 版本列表

使用 LevelDB 时，可以构建 LevelDB 的操作类：LeveldbDAO。LeveldbDAO 的代码设计如下所示：

```
package com.niudong.demo.dao;

import java.io.File;
import java.util.Map;

import javax.annotation.PostConstruct;

import org.iq80.leveldb.DB;
import org.iq80.leveldb.DBFactory;
import org.iq80.leveldb.DBIterator;
import org.iq80.leveldb.Options;
import org.iq80.leveldb.impl.Iq80DBFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.google.common.base.Strings;

/**
 * LeveldbDAO 工具类
 *
 * @author 牛冬
 *
 */
public class LeveldbDAO {
  // 日志记录
  private Logger logger = LoggerFactory.getLogger(LeveldbDAO.class);

  // LevelDB 的 DB 对象
  private DB db;

  @PostConstruct
  public void init() {
    try {
      DBFactory factory = new Iq80DBFactory();
      Options options = new Options();
      options.createIfMissing(true);
      // 配置 LevelDB 存储目录
      String path = ".//leveldb";
      db = factory.open(new File(path), options);
```

```
        db.put("hello world".getBytes(), "tal tech".getBytes());
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    // 存入数据或更改
    public void put(String key, String value) {
      if (Strings.isNullOrEmpty(key) || Strings.isNullOrEmpty(value)) {
        return;
      }
      db.put(Iq80DBFactory.bytes(key), Iq80DBFactory.bytes(value));
    }

    // 获取数据
    public String get(String key) {
      if (Strings.isNullOrEmpty(key)) {
        return null;
      }

      byte[] valueBytes = db.get(Iq80DBFactory.bytes(key));

      return Iq80DBFactory.asString(valueBytes);
    }

    // 删除数据
    public void delete(String key) {
      if (Strings.isNullOrEmpty(key)) {
        return;
      }

      db.delete(Iq80DBFactory.bytes(key));
    }

    // 遍历所有入库数据
    public void traverseAllDatas() {
      DBIterator iterator = db.iterator();

      while (iterator.hasNext()) {
        Map.Entry<byte[], byte[]> next = iterator.next();
        String key = Iq80DBFactory.asString(next.getKey());
```

```
            String value = Iq80DBFactory.asString(next.getValue());
            logger.info("traverse all data,levelDb key=" + key + ";value=" +
                value);
        }
      }
    }
```

如代码所示，该类中提供了通用的 LevelDB 数据库初始化方法 init()。在该方法中，我们配置 LevelDB 存储目录为 leveldb。通用的读、写、删除方法分别是 get()、put()和 delete()。此外，还提供了 LevelDB 数据库内容的遍历方法 traverseAllDatas()。

当执行代码时，LevelDB 会在配置的目录中生成一个按配置名称命名（本书样例代码中配置为 leveldb）的目录，如图 7-5 所示。

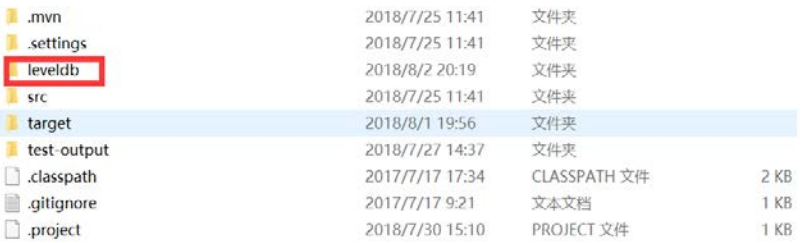


图 7-5　工程根目录下的 leveldb 文件夹

当运行一次 LevelDB 来写入数据时，LevelDB 可能会生成很多个 log 文件和 SSTable 文件（即.sst 文件）。这些文件的命名都是类似的，均为固定前缀+文件编号+固定后缀，这些文件的名称是通过调用函数 MakeFileName()来生成的。

打开 leveldb 文件夹，可以看到文件夹中的内容如图 7-6 所示。

图 7-6　leveldb 文件夹中的内容

如图 7-6 所示，除 log 文件（文件编号都是按 6 位数字进行输出的）和.sst 文件外，还有 MANIFEST 文件、Current 文件和 LOCK 文件。其中，MANIFEST 文件用于记录 LevelDB 的信息，包括 SSTable 文件的管理信息。

Current 文件用来保存当前使用的 MANIFEST 文件名。

<dbname>/log 是系统的运行日志，记录系统的运行信息或者错误日志。该 log 文件与 LevelDB 生成的以编号+.log 后缀的 log 文件不同，千万不要混为一谈。

LOCK 文件是一个空文件，它存在的目的在于实现单例模式，即帮助实现一个只存在一个实例的应用程序。

7.5　用 Java 实现 RocksDB 存储

7.5.1　RocksDB 介绍

RocksDB 项目始于 Facebook 一个实验项目中开发的高效的数据库软件，可以实现在服务器较高负载下快速存储。RocksDB 基于 C++编写，可用于存储 KV（Key Value）数据，包括任意大小的字节流。此外 RocksDB 还支持原子读写。

RocksDB 按顺序组织所有数据，常用操作是 get(key)、put(key)、delete(key)和 scan(key)。key 和 value 是纯字节流，对 key 或 value 的大小没有限制。

在配置上，RocksDB 的配置高度灵活，可以在各种生产环境（包括纯内存、闪存、硬盘或 HDFS）上运行。RocksDB 支持各种压缩算法，并且有生产和调试环境的各种便利工具。

当然，RocksDB 并非从零起步编写，而是站在巨人的肩膀上。RocksDB 借用了 LevelDB 开源项目的核心代码，以及来自 Apache HBase 的重要思想。初始代码采用了 LevelDB 开源代码的 1.5 版本。同时，RocksDB 还融入了 Facebook 团队在开发 RocksDB 之前的若干代码及想法。

RocksDB 官方测试得到的 RocksDB 性能数据统计如下。

按随机顺序批量加载数据性能：

```
rocksdb:157.6 minutes, 353.6 MB/sec (3.2TB ingested, file size 1.5TB)
```

按顺序批量加载数据：

```
rocksdb:159 minutes, 335.7 MB/sec (3.2TB ingested, file size 1.45 TB)
```

随机写：

```
rocksdb: 15 hours 38 min;  56.295 micros/op, 17K ops/sec,  13.8 MB/sec
```

随机读：

```
rocksdb:  18 hours,  64.7 micros/op, 15.4K ops/sec
```

多线程读和单线程写：

```
rocksdb: 75 minutes, 1.42 micros/read, 713376 reads/sec
```

以上性能数据是 2018 年 7 月基于 RocksDB 5.131 生成的。

7.5.2 RocksDB 的使用

在项目中使用 RocksDB 时，在工程的 POM 文件中配置 RocksDB 的依赖如下：

```
<dependency>
    <groupId>org.rocksdb</groupId>
    <artifactId>rocksdbjni</artifactId>
    <version>5.14.2</version>
</dependency>
```

在 mvnresponsity 网站中可查得 RocksDB JNI 的最新版本为 5.14.2，如图 7-7 所示。

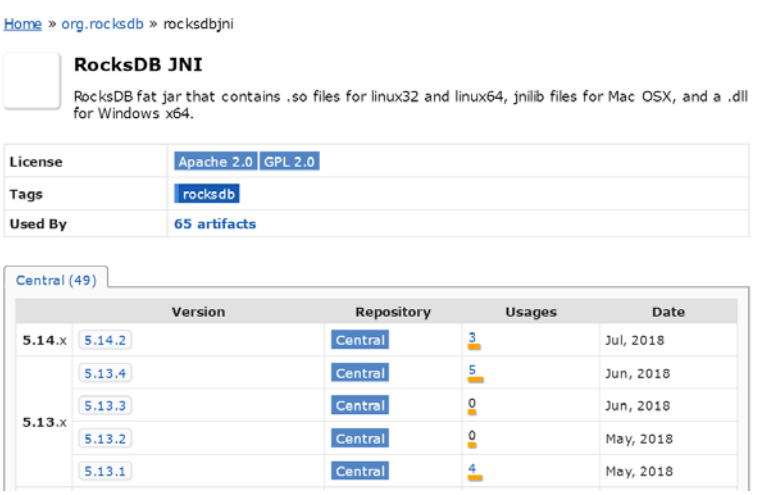


图 7-7　mvnresponsity 网站 RocksDB JNI 版本

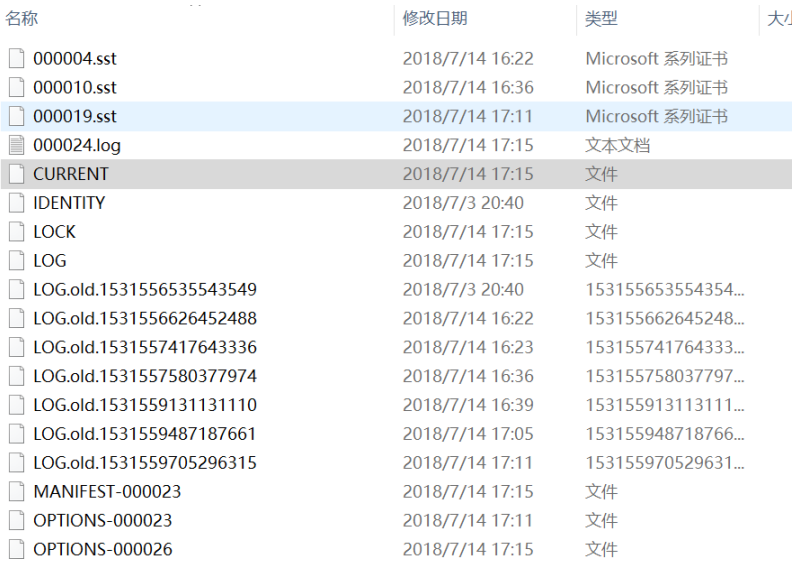
使用 RocksDB 时，可以构建 RocksDB 的操作类：RocksdbDAO。RocksdbDAO 的代码设计如下所示：

```
package com.niudong.demo.dao;

import javax.annotation.Resource;

import org.rocksdb.RocksDB;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import com.google.common.base.Strings;

/**
 * RocksdbDAO 工具类
 *
 * @author 牛冬
 *
 */
@Component
public class RocksdbDAO {
  // 日志记录
  private Logger logger = LoggerFactory.getLogger(RocksdbDAO.class);

  @Resource
  private RocksDB rocksDB;

  private static final String CHARSET = "utf-8";

  // 存入数据或更改
  public void put(String key, String value) {
    if (Strings.isNullOrEmpty(key) || Strings.isNullOrEmpty(value)) {
      return;
    }
    try {
      rocksDB.put(key.getBytes(CHARSET), value.getBytes(CHARSET));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

```
    // 获取数据
    public String get(String key) {
      if (Strings.isNullOrEmpty(key)) {
        return null;
      }

      try {
        byte[] bytes = rocksDB.get(key.getBytes(CHARSET));
        if (bytes != null) {
          return new String(bytes);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
      return null;
    }

    // 删除数据
    public void delete(String key) {
      if (Strings.isNullOrEmpty(key)) {
        return;
      }

      try {
        rocksDB.delete(rocksDB.get(key.getBytes(CHARSET)));
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

  }
```

如上述 RocksdbDAO 类代码所示，该类中提供了通用的 RocksdbDAO 读、写、删除方法分别是 get()、put()和 delete()。

RocksDB 通过 RocksdbDAO 初始化时，会在根目录中生成 rocksdb 文件夹，文件内容如图 7-8 所示。



名称	修改日期	类型	大小
000004.sst	2018/7/14 16:22	Microsoft 系列证书	
000010.sst	2018/7/14 16:36	Microsoft 系列证书	
000019.sst	2018/7/14 17:11	Microsoft 系列证书	
000024.log	2018/7/14 17:15	文本文档	
CURRENT	2018/7/14 17:15	文件	
IDENTITY	2018/7/3 20:40	文件	
LOCK	2018/7/14 17:15	文件	
LOG	2018/7/14 17:15	文件	
LOG.old.1531556535543549	2018/7/3 20:40	153155653554354...	
LOG.old.1531556626452488	2018/7/14 16:22	153155662645248...	
LOG.old.1531557417643336	2018/7/14 16:23	153155741764333...	
LOG.old.1531557580377974	2018/7/14 16:36	153155758037797...	
LOG.old.1531559131131110	2018/7/14 16:39	153155913113111...	
LOG.old.1531559487187661	2018/7/14 17:05	153155948718766...	
LOG.old.1531559705296315	2018/7/14 17:11	153155970529631...	
MANIFEST-000023	2018/7/14 17:15	文件	
OPTIONS-000023	2018/7/14 17:11	文件	
OPTIONS-000026	2018/7/14 17:15	文件	

图 7-8　rocksdb 文件夹中的内容

如图 7-8 所示，rocksdb 文件夹中有多种类型的文件。sst 文件存储的是持久化在数据库的数据。CURRENT 文件存储的是当前最新的 MANIFEST 文件；MANIFEST 文件存储的是 Version 值，而 Version 是不断变化的。LOG 文件是 RocksDB 的 write ahead log，在数据写入 RocksDB 之前写入，写的数据内容是日志文件。LOCK 可以打开 rocksdb 锁，同一时间只允许一个进程打开 RocksDB，这与 LevelDB 中的 LOCK 文件作用相同。

7.6　用 Java 实现 CouchDB 存储

7.6.1　CouchDB 介绍

CouchDB 是一个开源的 NoSQL 数据库，专注于提高易用性，最早发布于 2005 年，2008 年成为 Apache 基金会的开源项目，它由 Apache 开发，完全兼容 Web。

CouchDB 具有面向文档的 NoSQL 数据库体系结构，在这一点与 MongoDB 有些类似。在 CouchDB 中，所有文档域（Field）都是以键值对的形式存储的。域（Field）可以是一个简单的键值对，也可以是复杂的列表或者 MAP。

CouchDB 基于面向并发的语言 Erlang 实现，利用 JSON 结构来存储数据，以 JavaScript 作为查询语言，还可以使用 MapReduce 和 HTTP 作为 API。

根据 CouchDB 官方网站的描述，CouchDB 具有以下特性：

- 易于在多个服务器的实例之间进行数据库复制。
- 能快速地索引和检索数据。
- 提供了 REST 风格的文档插入、更新、检索和删除的接口。
- 基于 JSON 的文档格式（使其易于在不同语言之间转换）。
- 为用户选择的语言提供了多个库（特别是流行的编程语言）。
- 可以通过_changes 来订阅数据更新。

为了有效地管理 CouchDB，可以通过命令行或者一个称为 Futon 的 Web 界面来管理 CouchDB。Futon 可用于执行管理任务，例如，创建和操作 CouchDB 的数据库、文档和用户等。

与 LevelDB、RocksDB 这类内置、免安装的数据库不同，CouchDB 需要先安装才能使用，本节仅介绍 CouchDB 的使用，CouchDB 的安装内容详见附录 C。

目前，操作 CouchDB 时，有多个工具类库可供 Java 研发人员选择，如 Ektorp、JRelax、CouchDB4J、DroidCouch、JCouchDB 等。

本节我们使用 JCouchDB 与 CouchDB 数据库交互。JCouchDB 是一个经过良好测试且易于使用的 Java 库，它能自动地将 Java 对象序列化、反序列化进 CouchDB 数据库。特别是 JCouchDB 提供的 API 和 CouchDB 自身的 API 非常相似，特别容易上手。

7.6.2 CouchDB 的使用

在项目中使用 CouchDB 时，在工程的 POM 文件中配置 CouchDB 的依赖如下：

```
<dependency>
    <groupId>com.google.code.jcouchdb</groupId>
    <artifactId>jcouchdb</artifactId>
    <version>1.0.1-1</version>
</dependency>
```

在 mvnresponsity 网站中可查得 JCouchDB 的最新版本为 1.0.1-1，如图 7-9 所示。

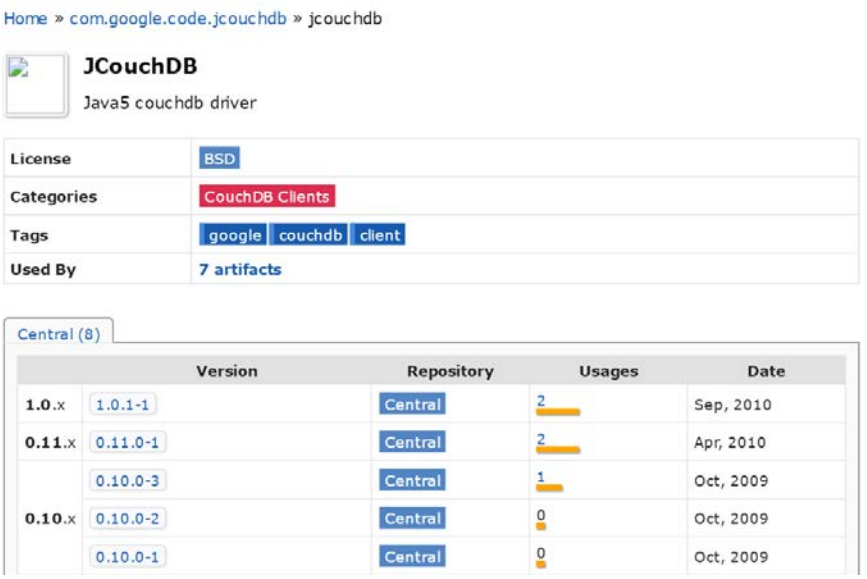


图 7-9　mvnresponsity 网站中的 JCouchDB 版本

可以看到 JCouchDB 类库已有 8 年不再更新。

上述引用 JCouchDB 类库的背景是基于 Spring 框架方式的，在 Spring Boot 火热的今天，显然这种使用 CouchDB 的方式效率偏低。下面来看基于 Spring Boot 框架下 CouchDB 简单高效的使用方式。

引入的依赖配置如下：

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-couchbase</artifactId>
    <version>3.0.9.RELEASE</version>
</dependency>
```

在使用 CouchDB 前需完成 CouchDB 配置。CouchDB 最低配置时要求用户设置主数据桶的名称和密码；还需配置一个 IPS 或主机名的列表，用于引导 GETBootStRAPHOST()。这些配置数据仅用于启动与集群的连接，随后客户端将发现所有节点，并保持群集映射最新。

CouchDB 的配置代码 CouchdbConfig 如下所示：

```
package com.niudong.demo.dao;

import java.util.Arrays;
import java.util.List;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.couchbase.config.
    AbstractCouchbaseConfiguration;
import org.springframework.data.couchbase.repository.
    config.EnableCouchbaseRepositories;

/**
 * CouchDB 的 Config 类
 *
 * @author 牛冬
 *
 */
@Configuration
@EnableCouchbaseRepositories
public class CouchdbConfig extends AbstractCouchbaseConfiguration {

  //启动时建立配置的集群的连接，客户端将发现所有节点并保持群集映射最新
  @Override
  protected List<String> getBootstrapHosts() {
    return Arrays.asList("host1", "host2");
  }

  // 配置主数据桶的名称
  @Override
  protected String getBucketName() {
    return "default";
  }

  // 配置主数据桶的密码
  @Override
  protected String getBucketPassword() {
    return "";
  }
}
```

下面介绍 CouchDB 的具体使用。与 MongoDB 类似，首先要构建存储的实体类，代码设计如下所示：

```
package com.niudong.demo.dao.entity;

import org.springframework.data.couchbase.core.mapping.Document;

import com.couchbase.client.java.repository.annotation.Field;
import com.couchbase.client.java.repository.annotation.Id;

@Document
public class UserCouchdb {
  @Id
  private String id;

  // 姓
  @Field
  private String lastname;


  // 性别:1 为女，0 为男
  @Field
  private int sex;

  //年龄
  @Field
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public int getSex() {
    return sex;
  }

  public void setSex(int sex) {
    this.sex = sex;
```

```
  }

  public String getLastname() {
    return lastname;
  }

  public void setLastname(String lastname) {
    this.lastname = lastname;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
    this.id = id;
  }
}
```

随后构建对应的 JPA 类 UserCouchdbRepository，代码如下：

```
package com.niudong.demo.dao;

import java.util.List;

import org.springframework.data.couchbase.core.query.Dimensional;
import org.springframework.data.couchbase.core.query.View;
import org.springframework.data.geo.Box;
import org.springframework.data.repository.CrudRepository;

import com.niudong.demo.dao.entity.UserCouchdb;

public interface UserCouchdbRepository extends
    CrudRepository<UserCouchdb, String> {
  List<UserCouchdb> findByLastnameAndAgeBetween(String lastName, int
     minAge, int maxAge);

  @View(designDocument = "user", viewName = "byName")
  List<UserCouchdb> findByLastname(String lastName);

  @Dimensional(designDocument = "userGeo", spatialViewName =
     "byLocation")
  List<UserCouchdb> findByLocationWithin(Box cityBoundingBox);
}
```

7.7　小结

本章介绍了比特币、Ripple 币、以太坊、超级账本的存储设计及对应的缺点，随后用 Java 实现了文件存储、SQLite 存储、LevelDB 存储、RocksDB 存储和 CouchDB 存储。

本书推荐读者使用 SQLite + LevelDB/RocksDB 的存储方案。

第 8 章
联盟链中的币设计

著叶满枝翠羽盖
开花无数黄金钱

联盟链系统中需要币的激励么？这还真不好一概而论，要视不同的场景而定。

诚然联盟链中的币或 Token 的影响力远没有公有链那么大，但联盟内部还是有真实的激励需求的。

当然，相较于公链中无门槛的公开进入，联盟链和私有链相对是“内部团体”，往往都是来自同一行业，相对要知根知底，激励体系自然也可以用其他的链外激励方式。链外激励方式有多种，这点可以向传统方式学习，比如收会费模式（超级账本就是按这种形式展开会员服务的）。

不论是传统激励也好，还是区块链领域特色的币或 Token 激励也罢，激励体系设计的关键不在于其和业务绑定的耦合度有多高，而在于激励体系的公开透明。

本书不对如何设计高效的激励体系进行阐述，也不对联盟链中是否必须有币的激励进行讨论，仅从技术层面论述联盟链中的币可以怎样来实现。

本章将先后介绍比特币和以太坊的币的设计，最后介绍 Java 版联盟链中的币如何来实现。

8.1　比特币的币设计

根据比特币作者中本聪的论文 *Bitcoin*: *A Peer-to-Peer Electronic Cash System* 可以知道，比特币的激励机制设计思路如下。

在比特币的区块链中，会对每个区块的第一笔交易进行特殊化处理，该交易会产生一枚由该区块的创造者（矿工）拥有的新的比特币。其实这就是比特币系统对个人或组织的挖矿节点支持比特币网络的一种激励。这也是比特币在没有中央集权机构发行货币的情况下，提供的一种将电子货币发行到流通领域的一种方法。当然，这个挖矿的过程需要消耗 CPU 的时间和电力，这和传统的金矿开发有一定的相似性，即挖得金矿前先付出一定的成本。

在比特币系统中，对挖矿人的激励并不仅仅是挖出区块的奖励，另外一个激励的来源是比特币系统中的交易费（Transaction Fees）。如果某笔交易的输出值小于输入值，那么两个数值的差额就是交易费，该交易费将被增加到新区块被挖出的激励中。

在比特币系统中，只要有一定数量的电子货币（即比特币）已经进入流通，那么激励就可以逐渐转换为完全依靠交易费来支付，因而理论上比特币系统就能够免于通货膨胀。

此外，比特币的激励系统也有助于鼓励节点保持诚实地付出。如果有一个贪婪的攻击者能够调集比所有诚实节点加起来还要多的 CPU 计算力，那么他就面临一个选择：要么将其用于诚实工作产生新的电子货币，即比特币，要么将其用于进行二次支付攻击。其实他会发现，按照规则行事、诚实工作是更有利可图的。因为遵守比特币系统的规则使得他能够拥有更多的电子货币（即比特币），而不是破坏这个系统使其自身财富蒙受损失。

在比特币系统的设计中，比特币总量为 2100 万个。首个比特币的区块——创世区块，诞生于格林威治时间 2009 年 1 月 3 日 18:15:05，创世区块的编号是 0。从创世区块开始的第一阶段，每个区块被挖出时，矿工将被奖励系统中新产生的 50 个比特币。

比特币系统的第二阶段从格林威治时间 2012 年 11 月 28 日 15:24:38 产生的编号第 210000 个区块开启，每个区块被挖出时，矿工将被奖励系统中新产生的 25 个比特币。这是比特币系统中预设的第一次激励减半。从这一阶段开始，比特币系统中每新产生 210000 个区块，比特币的数量都会依次减半。直到第 33 次减半时，直接减为 0 个。

由于比特币系统中设定大约每 10 分钟就能产生一个区块，因此，挖出 210000 个区块需要 210000 个 10 分钟，折合接近 4 年。

为了保持这样的挖矿速度，比特币系统会调节挖矿难度。比特币系统调节挖矿难度的原理是：根据前 2016 个块产生的总时间，调整后 2016 个块的挖矿难度，让挖出这 2016 个块的时间为 14 天。因为每小时的 6 个 10 分钟乘以 24 小时再乘以 14 天=2016。所以就能做到平均每 10 分钟生成一个区块。当然，由于目前计算能力提升很快，因此实际上挖出 2016 个块的速度往往少于 14 天。

在日常生活付现金购物的场景中，少不了用多张钞票累计以便达到商品价格，也少不了大额钞票的找零情况。在比特币系统中，类似的场景同样存在，在比特币系统中称之为价值的组合与分割（Combining and Splitting Value）。

虽然我们可以用 1 个比特币接 1 个比特币的方式进行比特币的交易处理，但是对于每一枚比特币就单独发起一次交易很明显是一个低效率甚至有点略显笨拙的方法。为了使得价值易于组合与分割，交易被设计为可以纳入多个输入和输出。一般而言，是某次价值较大的前次交易构成的单一输入，或者由某几个价值较小的前次交易共同构成的并行输入，但是输出最多只有两个：一个用于支付，另一个用于找零（如果需要的话）。

比特币找零机制的原理是，当比特币交易中需要输出的金额超过了用户想要支付的金额时，比特币客户端将创建一个新的比特币地址，并把差额发送给这个地址，这个地址对应的比特币则属于交易发起的用户。

在比特币系统中，为了防止双重支付和伪造，必须要确保在任何时候，新创建的货币金额与被销毁的货币金额是完全一样的。

下面我们通过两个实例来解释比特币的价值的组合与分割原理（不考虑交易对矿工的付费激励）。

假设比特币系统中的某用户 A 需要向比特币系统中的用户 B 转账 10 个比特币，此时，用户 A 的比特币钱包中含有 5 个比特币地址，5 个比特币地址分别对应的比特币数量如表 8-1 所示。

表 8-1　用户 A 交易前的比特币钱包中的比特币地址及数量

比特币地址编号	比特币地址	比特币数量
1	1Qwe123rty456UIOP7890AS12DF34gH56	1
2	2Qwe123rty456UIOP7890AS12DF34gH56	2
3	3Qwe123rty456UIOP7890AS12DF34gH56	3
4	4Qwe123rty456UIOP7890AS12DF34gH56	4
5	5Qwe123rty456UIOP7890AS12DF34gH56	5

此时，用户 A 可以选择将编号为第 1、2、3、4 的比特币地址转给用户 B，这 4 个比特币地址对应的比特币数量之和刚好是 10，因此用户 B 愉悦地笑纳。交易成功后，用户 A 和用户 B 的比特币钱包中的比特币地址及数量分别如表 8-2 和表 8-3 所示。

表 8-2　用户 B 交易后的比特币钱包中新增的比特币地址及数量

比特币地址编号	比特币地址	比特币数量
1	1Qwe123rty456UIOP7890AS12DF34gH56	1
2	2Qwe123rty456UIOP7890AS12DF34gH56	2
3	3Qwe123rty456UIOP7890AS12DF34gH56	3
4	4Qwe123rty456UIOP7890AS12DF34gH56	4

表 8-3　用户 A 交易后的比特币钱包中的比特币地址及数量

比特币地址编号	比特币地址	比特币数量
5	5Qwe123rty456UIOP7890AS12DF34gH56	5

若此时用户 A 不是向用户 B 转账 10 个比特币，而是 5.5 个，则用户 A 可以选择将编号为第 1、2、3 的比特币地址转给用户 B。由于第 1、2、3 的比特币地址对应的比特币数量为 6，超过了交易金额 5.5 个比特币，因此比特币系统中会新生成一个比特币地址用于给用户 A 找零 0.5 个比特币。交易成功后，用户 A 和用户 B 的比特币钱包中的比特币地址及数量分别如表 8-4 和表 8-5 所示。

表 8-4　用户 B 交易后的比特币钱包中新增的比特币地址及数量

比特币地址编号	比特币地址	比特币数量
1	1Qwe123rty456UIOP7890AS12DF34gH56	1
2	2Qwe123rty456UIOP7890AS12DF34gH56	2
3	3Qwe123rty456UIOP7890AS12DF34gH56	2.5

表 8-5　用户 A 交易后的比特币钱包中的比特币地址及数量

比特币地址编号	比特币地址	比特币数量
4	4Qwe123rty456UIOP7890AS12DF34gH56	4
5	5Qwe123rty456UIOP7890AS12DF34gH56	5
6	6Qwe123rty456UIOP7890AS12DF34gH56	0.5

8.2　以太币的激励机制

根据以太坊白皮书 *A Next-Generation Smart Contract and Decentralized Application Platform*，我们可以获取以太坊的激励机制设计如下。

以太坊网络包含内置货币即以太币，以太币扮演了双重角色，为各种数字资产交易提供主要的流动性，更重要的是，提供了一种支付交易费用的度量机制。在以太坊系统中，为便利及避免将来可能产生的争议，不同面值的名称已被提前设置好，具体如下：

1：伟

10^{12}：萨博

10^{15}：芬尼

10^{18}：以太

这和人民币面额设计系统中的圆、角、分有异曲同工之妙。以太坊的开发者希冀在不远的将来，“以太”被用作普通交易，“芬尼”被用来进行微交易，“萨博”和“伟”用来进行关于费用和协议实施的讨论。

以太币的发行模式为：通过发售活动，以太币将以每比特币 1337~2000 以太的价格发售，一个旨在为以太坊组织筹资并且为开发者支付报酬的机制已经在其他一些密码学货币平台上成功使用。早期购买者会享受较大的折扣，发售所得的比特币将完全用来支付开发者和研究者的工资和悬赏，以及投入密码学货币生态系统的项目。

此外，以太坊系统中设计了 2 个 0.099x（x 为发售总量）机制。其中一个 0.099x 用于分配给比特币融资或其他确定性融资成功之前参与开发的早期贡献者，另外一个 0.099x 将分配给长期研究项目。

自以太坊系统上线起每年都将有 0.26x（x 为发售总量）被矿工挖出。

与比特币定量发行不同的是，以太币理论上是没有总量限制的。假设以太币预售总量为 x，其发行总量即为

$$X = x + 0.099x + 0.099x$$

以太坊系统每年挖出固定数量的币的设计类似于线性模型，相较于比特币系统，降低了在以太坊系统中出现财富过于集中的概率，并且给予了当下和将来的矿工公平获取电子货币的机会。

由于每年都可挖出固定数量的币，因此以太坊系统中电子货币的货币供应增长率为 0，从理论上这种机制能长期保持对获取和持有以太币人的激励。

8.3 Java 版联盟链的币设计与实现

8.3.1 管理后台币的配置

本书中，笔者推荐在联盟链中设计可配置、可插拔的币激励系统。这部分配置可以放到联盟链的管理后台实现。关于联盟链管理后台的设计详见第 9 章，本节只介绍和币配置相关的部分。

我们可以构建一张币信息的配置表，在表中，可以配置币的发行总量、币在联盟节点的预留数量、区块挖出的奖励、最小的交易费用等内容，如下所示：

```
SET FOREIGN_KEY_CHECKS=0;
-- ----------------------------
-- Table structure for 'tb_coin_config'
-- ----------------------------
DROP TABLE IF EXISTS 'tb_coin_config';
CREATE TABLE 'tb_coin_config' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'biz_type' varchar(255) NOT NULL COMMENT '业务类型',
  'coin_total' int(11) NOT NULL COMMENT '币的总量',
  'coin_reserved' int(11) NOT NULL COMMENT '机构预留币的数量',
  'coin_per_deal' double NOT NULL COMMENT '每笔交易最小的币支付金额',
  'coin_block_create' double NOT NULL COMMENT
  '生成一个区块的奖励币数量',
  'create_time' datetime NOT NULL COMMENT '规则配置时间',
  'update_time' timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
     CURRENT_TIMESTAMP COMMENT '规则修改时间',
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

如上所示，币信息的配置表名为 tb_coin_config，表中字段释义如下：

id 字段表示表自增 id，biz_type 字段表示业务类型，coin_total 字段表示币的发行总量，coin_reserved 字段表示联盟内机构预留币的数量，coin_per_deal 字段表示每笔交易最小的币支付金额，coin_block_create 字段表示生成一个区块的奖励币数量，

create_time 字段表示币体系规则的配置时间，update_time 字段表示币体系规则的修改时间。

我们基于 Spring Boot 构建的表 tb_coin_config 对应的 Java 实体类如下：

```
package com.niudong.demo.dao.entity;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * 联盟内激励配置表
 *
 * @author niudong
 */
@Entity
@Table(name = "tb_coin_config")
public class CoinConfigEntity {
  // 数据库 ID
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // 业务类型
  private String bizType;
  // 币的总量
  private int coinTotal;
  // 机构预留币的数量
  private int coinReserved;
  // 每笔交易最小的币支付金额
  private double coinPerDeal;
  // 生成一个区块的奖励币数量
  private double coinBlockCreate;
  // 激励规则配置信息创建时间
  private Date createTime;
  // 激励规则配置信息更新时间
  private Date updateTime;
```

```
    public Date getCreateTime() {
      return createTime;
    }

    public void setCreateTime(Date createTime) {
      this.createTime = createTime;
    }

    public Date getUpdateTime() {
      return updateTime;
    }

    public void setUpdateTime(Date updateTime) {
      this.updateTime = updateTime;
    }

    public double getCoinBlockCreate() {
      return coinBlockCreate;
    }

    public void setCoinBlockCreate(double coinBlockCreate) {
      this.coinBlockCreate = coinBlockCreate;
    }

    public double getCoinPerDeal() {
      return coinPerDeal;
    }

    public void setCoinPerDeal(double coinPerDeal) {
      this.coinPerDeal = coinPerDeal;
    }

    public int getCoinReserved() {
      return coinReserved;
    }

    public void setCoinReserved(int coinReserved) {
      this.coinReserved = coinReserved;
    }

    public int getCoinTotal() {
```

```
    return coinTotal;
  }

  public void setCoinTotal(int coinTotal) {
    this.coinTotal = coinTotal;
  }

  public String getBizType() {
    return bizType;
  }

  public void setBizType(String bizType) {
    this.bizType = bizType;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }
}
```

基于 Spring Boot JPA 构建的 DAO 层代码如下：

```
package com.niudong.demo.dao;

import org.springframework.stereotype.Repository;
import org.springframework.data.jpa.repository.JpaRepository;

import com.niudong.demo.dao.entity.CoinConfigEntity;

/**
 * 联盟内激励配置表对应的 DAO 层
 *
 * @author 牛冬
 *
 */
@Repository
public interface ICoinConfigDAO extends
    JpaRepository<CoinConfigEntity, Long> {
```

```
    public CoinConfigEntity findByBizType(String bizType);

}
```

8.3.2 Java 实现币交易

8.3.1 节我们建立了币信息的配置表，编写了配置表对应的实体类和 DAO 层接口。

在区块生成时，联盟链系统会奖励相应的节点，系统生成一个新的币地址，并在地址中赋予配置好相应数量的币。

在联盟链中，除创建区块有币的流通外，币的交易也是重要的流通方式。而在币的交易过程中，特别是找零的处理逻辑，和区块生成时系统对联盟成员奖励若干数量的币的逻辑类似，因此本节以币的交易为例展开。

为了记录联盟节点钱包详情信息和联盟内各个节点的币交易信息，我们需要分别建立联盟节点钱包信息表和联盟内币交易信息表。

联盟节点钱包信息表中需要包含联盟节点 id 信息、每个币的地址及对应的余额信息、币的归属时间等。因此，我们建立的联盟节点钱包信息表如下所示：

```
SET FOREIGN_KEY_CHECKS=0;
-- ----------------------------
-- Table structure for "tb_user_wallet"
-- ----------------------------
DROP TABLE IF EXISTS "tb_user_wallet";
CREATE TABLE "tb_user_wallet" (
  "id" int(11) NOT NULL AUTO_INCREMENT,
  "user_id" int(11) NOT NULL COMMENT '联盟节点 ID',
  "coin_address" varchar(255) NOT NULL COMMENT '币地址',
  "coin_balance" double NOT NULL COMMENT '币地址对应的余额',
  "create_time" datetime NOT NULL COMMENT '交易创建时间',
  "update_time" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
     CURRENT_TIMESTAMP COMMENT '交易更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

其中，联盟节点钱包信息表名称为 tb_user_wallet，主要字段释义如下：

id 字段表示表的自增字段，user_id 字段表示联盟节点 ID，coin_address 字段表示币地址，coin_balance 字段表示币地址对应的余额，create_time 字段表示交易创建时间，update_time 字段表示交易更新时间。

基于 Spring Boot JPA 建立 tb_user_wallet 表对应的实体类代码如下所示：

```
package com.niudong.demo.dao.entity;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * 联盟内用户钱包表，用于记录钱包内的币地址和余额
 *
 * @author niudong
 */
@Entity
@Table(name = "tb_coin_deal")
public class UserWalletEntity {
  // 数据库 ID
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  // 用户 ID
  private int userId;
  // 币地址
  private String coinAddress;
  // 币地址对应的余额
  private double coinBalance;
  // 交易创建时间
  private Date createTime;
  // 交易更新时间
  private Date updateTime;

  public Date getUpdateTime() {
    return updateTime;
```

```
    }

    public void setUpdateTime(Date updateTime) {
      this.updateTime = updateTime;
    }

    public Date getCreateTime() {
      return createTime;
    }

    public void setCreateTime(Date createTime) {
      this.createTime = createTime;
    }

    public double getCoinBalance() {
      return coinBalance;
    }

    public void setCoinBalance(double coinBalance) {
      this.coinBalance = coinBalance;
    }

    public String getCoinAddress() {
      return coinAddress;
    }

    public void setCoinAddress(String coinAddress) {
      this.coinAddress = coinAddress;
    }

    public int getUserId() {
      return userId;
    }

    public void setUserId(int userId) {
      this.userId = userId;
    }

    public Long getId() {
      return id;
    }
```

```
    public void setId(Long id) {
      this.id = id;
    }
}
```

基于 Spring Boot JPA 建立 tb_user_wallet 表对应的 DAO 层代码如下：

```
package com.niudong.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.niudong.demo.dao.entity.UserWalletEntity;

/**
 * 联盟内用户钱包表对应的 DAO 层
 *
 * @author 牛冬
 *
 */
@Repository
public interface IUserWalletDAO extends
    JpaRepository<UserWalletEntity, Long> {

}
```

联盟内币交易信息表至少要包含交易双方的 ID 字段、交易发起方的币地址及余额、交易接收方的地址及余额、交易时间等字段。因此，我们建立的联盟内币的交易明细表如下：

```
SET FOREIGN_KEY_CHECKS=0;
-- ----------------------------
-- Table structure for "tb_coin_deal"
-- ----------------------------
DROP TABLE IF EXISTS "tb_coin_deal";
CREATE TABLE "tb_coin_deal" (
  "id" int(11) NOT NULL AUTO_INCREMENT,
  "from_user_id"  int(11) NOT NULL COMMENT '交易发起方用户 ID',
  "from_address" varchar(255) NOT NULL COMMENT '交易发起方地址',
  "coin_balance" double NOT NULL COMMENT '交易前账户余额',
  "to_user_id" int(11) DEFAULT NULL COMMENT '交易接收方 ID',
  "to_address" varchar(255) NOT NULL COMMENT '交易接收方地址',
```

```
    "coin_balance_dealed" double NOT NULL COMMENT '交易后账户余额',
    "create_time" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT '交易创建时间',
    PRIMARY KEY (`id`)
  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

如上所示，我们建立的联盟内币的交易明细表名称为 tb_coin_deal，该表的主要字段释义如下：

id 字段表示表内自增 ID，from_user_id 字段表示交易发起方用户 ID，from_address 字段表示交易发起方地址，coin_balance 字段表示交易前账户余额，to_user_id 字段表示交易接收方 ID，to_address 字段表示交易接收方地址，coin_balance_dealed 字段表示交易后账户余额，create_time 字段表示交易创建时间。

基于 Spring Boot JPA 建立 tb_coin_deal 表对应的实体类代码如下：

```
package com.niudong.demo.dao.entity;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * 联盟内币的交易明细表，用于记录交易历史
 *
 * @author niudong
 */
@Entity
@Table(name = "tb_coin_deal")
public class CoinDealEntity {

  // 数据库 ID
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  // 交易发起方用户 ID
  private int fromUserId;
```

```
    // 交易发起方地址
    private String fromAddress;
    // 交易前账户余额
    private double coinBalance;
    // 交易接收方 ID
    private int toUserId;
    // 交易接收方地址
    private String toAddress;
    // 交易后账户余额
    private double coinBalanceDealed;
    // 交易创建时间
    private Date createTime;

    public Date getCreateTime() {
      return createTime;
    }

    public void setCreateTime(Date createTime) {
      this.createTime = createTime;
    }

    public double getCoinBalanceDealed() {
      return coinBalanceDealed;
    }

    public void setCoinBalanceDealed(double coinBalanceDealed) {
      this.coinBalanceDealed = coinBalanceDealed;
    }

    public String getToAddress() {
      return toAddress;
    }

    public void setToAddress(String toAddress) {
      this.toAddress = toAddress;
    }

    public int getToUserId() {
      return toUserId;
    }

    public void setToUserId(int toUserId) {
```

```
    this.toUserId = toUserId;
  }

  public double getCoinBalance() {
    return coinBalance;
  }

  public void setCoinBalance(double coinBalance) {
    this.coinBalance = coinBalance;
  }

  public String getFromAddress() {
    return fromAddress;
  }

  public void setFromAddress(String fromAddress) {
    this.fromAddress = fromAddress;
  }

  public int getFromUserId() {
    return fromUserId;
  }

  public void setFromUserId(int fromUserId) {
    this.fromUserId = fromUserId;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }
}
```

基于 Spring Boot JPA 建立 tb_coin_deal 表对应的 DAO 层代码如下：

```
package com.niudong.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
```

```
import com.niudong.demo.dao.entity.CoinDealEntity;

/**
 * 联盟内币的交易明细表对应的 DAO 层
 *
 * @author 牛冬
 *
 */
@Repository
public interface ICoinDealDAO extends JpaRepository<CoinDealEntity,
    Long> {

}
```

在区块链中，币的信息随区块内容传播，因此我们需要在区块实体类中增加币交易相关的内容。我们可以在第 6 章提到的 Java 版区块设计的基础上增加币的交易实体类对象。该交易实体类对象需要包含交易双方 ID 信息、交易费用、给联盟链内节点（矿工）将信息打包到新生成区块的付费信息、交易发起方的支付币信息等。而币信息中需要包含币地址和对应的余额。

因此我们构建了 CoinInfo 和 DealInfo 分别表示币信息和交易实体类对象。

币信息类 CoinInfo 的代码如下：

```
package com.niudong.demo.blockchain;

/**
 * 每个币的实体信息
 *
 * @author 牛冬
 *
 */
public class CoinInfo {
  // 币地址
  private String address;
  // 币地址对应的余额
  private double balance;

  public String getAddress() {
    return address;
  }
```

```
  public void setAddress(String address) {
    this.address = address;
  }

  public double getBalance() {
    return balance;
  }

  public void setBalance(double balance) {
    this.balance = balance;
  }
}
```

交易实体类 DealInfo 的代码如下：

```
package com.niudong.demo.blockchain;

import java.util.List;

/**
 * 每个交易的实体信息
 *
 * @author 牛冬
 *
 */
public class DealInfo {
  // 交易发起人
  private int fromUserId;
  // 交易发起人用于转账的 CoinInfo 列表
  private List<CoinInfo> fromCoinList;
  // 交易收款人
  private int toUserId;
  // 交易费用
  private double dealCost;
  // 打包成区块的费用
  private double blockCost;

  public double getBlockCost() {
    return blockCost;
  }

  public void setBlockCost(double blockCost) {
```

```
    this.blockCost = blockCost;
  }

  public double getDealCost() {
    return dealCost;
  }

  public void setDealCost(double dealCost) {
    this.dealCost = dealCost;
  }

  public int getToUserId() {
    return toUserId;
  }

  public void setToUserId(int toUserId) {
    this.toUserId = toUserId;
  }

  public List<CoinInfo> getFromCoinList() {
    return fromCoinList;
  }

  public void setFromCoinList(List<CoinInfo> fromCoinList) {
    this.fromCoinList = fromCoinList;
  }

  public int getFromUserId() {
    return fromUserId;
  }

  public void setFromUserId(int fromUserId) {
    this.fromUserId = fromUserId;
  }
}
```

我们可以将 DealInfo 对象的 JSON 数据传入第 6 章提及的 ContentInfo 下的 jsonContent 中，用于在各联盟节点中传输交易数据。

当交易发生时，调用 CoinService 类接口。CoinService 代码如下所示：

```
package com.niudong.demo.service;
```

```
import com.niudong.demo.blockchain.DealInfo;

/**
 * 联盟链中币交易服务类
 *
 * @author 牛冬
 *
 */
public interface CoinService {
  public void deal(DealInfo dealInfo, int allianceNodeID);
}
```

CoinService 的实现类 CoinServiceImpl 的接口代码如下所示：

```
package com.niudong.demo.service.impl;

import java.util.Date;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.google.common.collect.Ordering;
import com.niudong.demo.blockchain.CoinInfo;
import com.niudong.demo.blockchain.DealInfo;
import com.niudong.demo.service.CoinService;

import cn.hutool.crypto.digest.DigestUtil;

import com.niudong.demo.dao.ICoinDealDAO;
import com.niudong.demo.dao.IUserWalletDAO;
import com.niudong.demo.dao.entity.CoinDealEntity;
import com.niudong.demo.dao.entity.UserWalletEntity;

/**
 * 联盟链中币交易服务的实现类
 *
 * @author 牛冬
 *
 */
```

```
@Service
public class CoinServiceImpl implements CoinService {
  protected static Logger logger =
      LoggerFactory.getLogger(CoinServiceImpl.class);

  @Autowired
  private IUserWalletDAO userWalletDAO;
  @Autowired
  private ICoinDealDAO coinDealDAO;

  // 节点新生成区块的奖励币数量
  private static final int CREATE_BLOCK_COST = 100;

  // 模拟区块处理交易过程，其中 dealInfo 是交易信息，allianceNodeID 是联盟节点
  // 信息
  public void deal(DealInfo dealInfo, int allianceNodeID) {

    //allianceNodeID 有效性校验
    if(allianceNodeID <= 0) {
      return;
}

    // 空校验
    if (dealInfo == null) {
      return;
    }

    // dealInfo 内容校验
    boolean isCheckedPass = checkDealInfo(dealInfo);
    // 若校验结果不合法，则抛弃，不进行处理
    if (!isCheckedPass) {
      return;
    }

    // 以下是校验结果合法时的处理逻辑
    processDeal(dealInfo, allianceNodeID);
  }

  // 校验结果合法时的处理逻辑，此处略去将数据 JSON 化写入区块的部分
  public void processDeal(DealInfo dealInfo, int allianceNodeID) {
    // 处理交易双方的逻辑
    processAndSaveDeal(dealInfo);
```

```
    // 将区块生成的奖励入库
    createBlockToAlliance(allianceNodeID);
  }

  // 处理交易双方的逻辑
  public void processAndSaveDeal(DealInfo dealInfo) {
    // 获取交易发起人 ID
    int fromUserId = dealInfo.getFromUserId();
    // 获取交易收款人 ID
    int toUserId = dealInfo.getToUserId();
    // 打包成区块的费用
    double blockCost = dealInfo.getBlockCost();
    // 获取交易发起人用于转账的 CoinInfo 列表
    List<CoinInfo> fromCoinList = dealInfo.getFromCoinList();
    // 获取交易费用
    double dealCost = dealInfo.getDealCost();

    // 对 List<CoinInfo>排序，按 balance 从小到大排序
    Ordering<CoinInfo> coinOrdering = new Ordering<CoinInfo>() {
      public int compare(CoinInfo left, CoinInfo right) {
        return Double.compare(left.getBalance(), right.getBalance());
      }
    };
    fromCoinList = coinOrdering.sortedCopy(fromCoinList);

    // 查找满足交易费用的 1 个或多个 address 及对应的余额 balance
    int index = 0, length = fromCoinList.size();
    double count = 0;
    for (int i = 0; i < length; i++) {
      count += fromCoinList.get(i).getBalance();

      // 保存该条币信息到收款人，同时保存到交易流水记录
      saveIntoTable(fromCoinList.get(i).getAddress(),
          fromCoinList.get(i).getAddress(),
          fromCoinList.get(i).getBalance(),
              fromCoinList.get(i).getBalance(), toUserId, fromUserId);

      if (count > dealCost) {
        index = i;

        // 分拆该条信息数据为两个账户
```

```
            double toUserDiffBalance = dealCost - (count –
                fromCoinList.get(i).getBalance());
            // 生成一个新币地址
            String coinAddress =
                DigestUtil.sha256Hex("create block,time is " +
                    System.currentTimeMillis());
            saveIntoTable(null, coinAddress, toUserDiffBalance,
                toUserDiffBalance, toUserId, fromUserId);

            // 交易发起方账户变化，同时保存到交易流水记录
            saveIntoTable(fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getBalance(),
                    fromCoinList.get(i).getBalance() - toUserDiffBalance,
                toUserId, fromUserId);
          }
        }

        // 处理交易打包成区块费用 blockCost
        for (int i = index; i < length; i++) {
          if (fromCoinList.get(i).getBalance() <= blockCost) {
            // 直接转移币地址即可

            // 保存该条币信息到收款人，同时保存到交易流水记录
            saveIntoTable(fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getBalance(),
                    fromCoinList.get(i).getBalance(), toUserId,
                fromUserId);

            blockCost -= fromCoinList.get(i).getBalance();
            if (blockCost == 0.0) {
              return;
            }
          }

          if (fromCoinList.get(i).getBalance() > blockCost) {
            // 拆分成两个新账户

            // 生成一个新币地址
            String coinAddress =
                DigestUtil.sha256Hex("create block,time is " +
```

```
                System.currentTimeMillis());
            saveIntoTable(null, coinAddress, blockCost, blockCost,
                toUserId, fromUserId);

            // 交易发起方账户变化，同时保存到交易流水记录
            saveIntoTable(fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getAddress(),
                fromCoinList.get(i).getBalance(),
                    fromCoinList.get(i).getBalance() - blockCost,
                toUserId, fromUserId);
            return;
          }
        }

    }

    // 保存信息到两个表
    public void saveIntoTable(String fromAddress, String toAddress,
        double coinBalance,
        double coinBalanceDealed, int toUserId, int fromUserId) {
      UserWalletEntity userWalletEntity = new UserWalletEntity();
      userWalletEntity.setCoinAddress(toAddress);
      userWalletEntity.setCoinBalance(coinBalance);
      userWalletEntity.setCreateTime(new Date());
      userWalletEntity.setUpdateTime(new Date());
      userWalletEntity.setUserId(toUserId);
      userWalletDAO.save(userWalletEntity);

      // 保存到交易流水记录
      CoinDealEntity coinDealEntity = new CoinDealEntity();
      coinDealEntity.setCoinBalance(coinBalance);
      coinDealEntity.setCoinBalanceDealed(coinBalanceDealed);
      coinDealEntity.setCreateTime(new Date());
      coinDealEntity.setFromAddress(fromAddress);
      coinDealEntity.setFromUserId(fromUserId);
      coinDealEntity.setToUserId(toUserId);
      coinDealEntity.setToAddress(toAddress);
      coinDealDAO.save(coinDealEntity);
    }

    // 给区块生成的节点增加费用收入记录
    public void insertBlockCostToAlliance() {
```

```
        UserWalletEntity userWalletEntity = new UserWalletEntity();

        // userWalletEntity.setCoinAddress(coinAddress);
        userWalletDAO.save(userWalletEntity);
    }

    // 将区块生成的奖励入库
    public void createBlockToAlliance(int allianceNodeID) {
        UserWalletEntity userWalletEntity = new UserWalletEntity();

        // 新生成一个币地址用于存储区块打包费用
        String coinAddress = DigestUtil.sha256Hex("create block,time is "
            + System.currentTimeMillis());
        userWalletEntity.setCoinAddress(coinAddress);

        // 设置新生成区块的奖励
        double coinBalance = CREATE_BLOCK_COST;
        userWalletEntity.setCoinBalance(coinBalance);

        // 设计奖励所属人
        int userId = allianceNodeID;
        userWalletEntity.setUserId(userId);

        userWalletEntity.setCreateTime(new Date());
        userWalletEntity.setUpdateTime(new Date());

        userWalletDAO.save(userWalletEntity);
    }

    // dealInfo 内容校验
    public boolean checkDealInfo(DealInfo dealInfo) {
        boolean result = false;

        // 若交易双方的 ID 不合法，则直接返回
        if (dealInfo.getFromUserId() <= 0 || dealInfo.getToUserId() <= 0) {
            return result;
        }

        // 如果生成区块的交易费用小于等于 0，则不合法
        if (dealInfo.getBlockCost() <= 0) {
            return result;
        }
```

```
        List<CoinInfo> fromCoinList = dealInfo.getFromCoinList();
        // 如果交易发起方的交易列表为空，则不合法
        if (fromCoinList == null || fromCoinList.size() == 0) {
          return result;
        }

        // 获取 fromCoinList 中涉及的交易地址对应的余额总量
        double allBalances = getAllCoinBalances(fromCoinList);
        // 获取交易费用
        double dealCost = dealInfo.getDealCost();
        // 余额不足交易费用或等于交易费用但没有用于支付区块创建的费用时，则不合法
        if (allBalances <= dealCost) {
          return result;
        }

        // 余额不足支付交易费用和创建区块的费用时不合法
        if (allBalances < dealCost + dealInfo.getBlockCost()) {
          return result;
        }

        return result;
      }

      // 计算 List<CoinInfo> fromCoinList 中涉及的交易地址对应的余额总量
      public double getAllCoinBalances(List<CoinInfo> fromCoinList) {
        double allBalances = 0d;
        for (CoinInfo coinInfo : fromCoinList) {
          allBalances += coinInfo.getBalance();
        }
        return allBalances;
      }

    }
```

CoinServiceImpl 类代码逻辑解析如下：调用入口为 deal(DealInfo dealInfo, int allianceNodeID)方法，该方法传入 DealInfo 类型的 dealInfo 参数和 int 类型的 allianceNodeID 参数，其中，allianceNodeID 表示交易打包的联盟节点 ID 信息。

在 deal(DealInfo dealInfo, int allianceNodeID)中需要先对输入数据进行参数校验，主要是校验 allianceNodeID 的合法性，如不能为非正数。对 dealInfo 的校验逻辑如下：

步骤 1　先校验交易双方的 ID。若不合法，则对本次交易不做处理，直接返回；若合法，则进入步骤 2 的校验。

步骤 2　校验区块生成的交易费用。如果生成区块的交易费用小于等于 0，则不合法，本次交易不做处理，直接返回；若合法，则进入步骤 3。

步骤 3　校验交易发起方的交易列表内容。如果交易发起方的交易列表为空，则不合法，本次交易不做处理，直接返回；若合法，则进入步骤 4。

步骤 4　校验余额和交易费用、区块创建交易费用的关系。若余额不足交易费用或等于交易费用，但没有用于支付区块创建的费用时，则不合法，本次交易不做处理，直接返回；若合法，则进入步骤 5。

步骤 5　继续校验余额和交易费用、区块创建交易费用的关系。若余额不足支付交易费用和创建区块的费用，则不合法，本次交易不做处理，直接返回；若合法，则证明 dealInfo 的信息完全通过校验。

当通过 checkDealInfo(dealInfo)的校验后，进入交易逻辑的实际处理方法 processDeal(dealInfo, allianceNodeID)中。该方法的逻辑分为两部分：处理交易双方的逻辑和将区块生成的奖励入库。

处理交易双方的逻辑相对复杂些，但也不难理解。主要逻辑如下：

步骤 1　将 dealInfo 中的交易币地址列表按币余额从小到大排序，排序完成后转步骤 2。

步骤 2　查找满足交易费用的 1 个或多个 address 及对应的余额 balance，由于已经在 checkDealInfo(dealInfo)中进行了必要的信息校验，所以进入当前逻辑时，余额和交易费用与区块创建交易费用的大小关系都是合法的。

依次遍历排序后的交易币地址列表信息，当找到小于交易费用的币地址及余额时，将该币地址和余额转到交易接收方的账户中，并将交易的流水信息入库，可以通过 saveIntoTable 方法实现。

同时，将该币地址对应的余额累计到已转账给交易接收方的交易费用。当转账累计币数量要超过本次交易费用时，转到步骤 3。

步骤 3　拆分该币实体信息数据为两个账户，其中生成一个新币地址，用于给交易发起方找零，该币地址对应的余额为币地址对应的余额–（交易费用–目前累计转账币数量的差值）；将现有币地址的余额更新为（交易费用–目前累计转账币数量的差值）。将账户数据保存到两个账户钱包的同时，将交易流水信息保存到流水记录表。然后转步骤 4。

步骤 4　此时进行处理交易打包成区块费用 blockCost 的处理逻辑。该部分的逻辑和两个用户转账类似，不同之处在于交易接收方变成了联盟链中的节点，因此不再重复介绍该部分的处理逻辑。

将区块生成的奖励入库的逻辑相对简单，主要是新生成一个币地址，并根据配置的区块生成的奖励币数量和该币地址建立映射，同时和交易的打包节点 ID 一起写入钱包表中。

CoinServiceImpl 代码编写完成后，我们基于 TestNG+Mockito 编写的单元测试类 CoinServiceImplTest 的代码如下所示：

```
package com.niudong.demo.service.impl;

import java.util.ArrayList;
import java.util.List;

import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import com.niudong.demo.blockchain.CoinInfo;
import com.niudong.demo.blockchain.DealInfo;
import com.niudong.demo.dao.ICoinDealDAO;
import com.niudong.demo.dao.IUserWalletDAO;

/**
 * CoinServiceImpl的测试类
 *
 * @author 牛冬
 *
```

```
 */
public class CoinServiceImplTest {

  @Mock
  private IUserWalletDAO userWalletDAO;
  @Mock
  private ICoinDealDAO coinDealDAO;
  @InjectMocks
  private CoinServiceImpl service;

  @BeforeTest
  public void before() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void testGetAllCoinBalances() {
    // 构造 Mock 数据
    List<CoinInfo> fromCoinList = new ArrayList<>();
    CoinInfo ci1 = new CoinInfo();
    ci1.setAddress("1");
    ci1.setBalance(1.1d);
    fromCoinList.add(ci1);
    CoinInfo ci2 = new CoinInfo();
    ci2.setAddress("2");
    ci2.setBalance(2.2d);
    fromCoinList.add(ci2);

    // 执行 Mock 测试并验证结果
    Assert.assertEquals(Math.abs(service.
        getAllCoinBalances(fromCoinList) - 3.3d) < 0.01, true);
  }

  @Test
  public void testCheckDealInfo() {
    // case 1： DealInfo dealInfo 字段不设置值
    DealInfo dealInfo = new DealInfo();
    Assert.assertEquals(service.checkDealInfo(dealInfo), false);

    // case 2:fromUserId = -1 & toUserId = 0（默认）
    dealInfo.setFromUserId(-1);
    Assert.assertEquals(service.checkDealInfo(dealInfo), false);
```

```
        // case 3:fromUserId = -1 & toUserId = -1
        dealInfo.setToUserId(-1);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 4:fromUserId = 1 & toUserId = -1
        dealInfo.setFromUserId(1);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 5:fromUserId = 1 & toUserId = 1 & blockCost 默认= 0
        dealInfo.setToUserId(1);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 6:fromUserId = 1 & toUserId = 1 & blockCost > 0 & FromCoinList
        // 默认为空
        dealInfo.setBlockCost(1);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 7:fromUserId = 1 & toUserId = 1 & blockCost > 0 & FromCoinList
        // 不为空，但 size=0
        List<CoinInfo> fromCoinList = new ArrayList<>();
        dealInfo.setFromCoinList(fromCoinList);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 8:fromUserId = 1 & toUserId = 1 & blockCost > 0 & FromCoinList
        // 不为空，size!=0
        CoinInfo ci = new CoinInfo();
        ci.setBalance(1);
        fromCoinList.add(ci);
        Assert.assertEquals(service.checkDealInfo(dealInfo), true);

        // case 9:fromUserId = 1 & toUserId = 1 & blockCost > 0 & FromCoinList
        // 不为空，size!=0，dealCost=2
        // 验证：余额不足交易费用或等于交易费用，但没有用于支付区块创建的费用时，则不
        // 合法
        dealInfo.setDealCost(2);
        Assert.assertEquals(service.checkDealInfo(dealInfo), false);

        // case 10:fromUserId = 1 & toUserId = 1 & blockCost > 0 & FromCoinList
        // 不为空，size!=0，dealCost=0.2
        // 余额不足支付交易费用和创建区块的费用时不合法
        dealInfo.setDealCost(0.2);
```

```
    Assert.assertEquals(service.checkDealInfo(dealInfo), false);
  }

  @Test
  public void testCreateBlockToAlliance() {
    // 无返回结果，直接运行
    int allianceNodeID = 124;
    service.createBlockToAlliance(allianceNodeID);
  }

  @Test
  public void testInsertBlockCostToAlliance() {
    // 无返回结果，直接运行
    service.insertBlockCostToAlliance();
  }

  @Test
  public void testSaveIntoTable() {
    // 无返回结果，直接运行
    String fromAddress = "123";
    String toAddress = "234";
    double coinBalance = 1.0;
    double coinBalanceDealed = 2.0;
    int toUserId = 123;
    int fromUserId = 234;
    service.saveIntoTable(fromAddress, toAddress, coinBalance,
        coinBalanceDealed, toUserId,
        fromUserId);
  }

  @Test
  public void testProcessDeal() {
    // 无返回结果，直接运行。由于本方法简单且会调用 processAndSaveDeal 方法，因
    // 此本测试方法主要用来测试 processAndSaveDeal
    DealInfo dealInfo = new DealInfo();
    int allianceNodeID = 123;

    List<CoinInfo> fromCoinList = new ArrayList<>();
    CoinInfo ci1 = new CoinInfo();
    ci1.setBalance(1);
    fromCoinList.add(ci1);
    CoinInfo ci2 = new CoinInfo();
```

```
        ci2.setBalance(2);
        fromCoinList.add(ci2);
        dealInfo.setFromCoinList(fromCoinList);

        dealInfo.setDealCost(1.2);

        // case1:count > dealCost
        service.processDeal(dealInfo, allianceNodeID);

        // case2:
        dealInfo.setBlockCost(3);
        service.processDeal(dealInfo, allianceNodeID);
      }

      @Test
      public void testDeal() {
        // case1 DealInfo dealInfo = null & allianceNodeID = -1
        DealInfo dealInfo = null;
        int allianceNodeID = -1;
        service.deal(dealInfo, allianceNodeID);

        allianceNodeID = 123;
        service.deal(dealInfo, allianceNodeID);

        // case2: dealInfo != null 无返回结果，直接运行。由于本方法简单且会调用
        // processDeal 方法，因此本测试方法主要用来测试 processDeal
        dealInfo = new DealInfo();

        List<CoinInfo> fromCoinList = new ArrayList<>();
        CoinInfo ci1 = new CoinInfo();
        ci1.setBalance(1);
        fromCoinList.add(ci1);
        CoinInfo ci2 = new CoinInfo();
        ci2.setBalance(2);
        fromCoinList.add(ci2);
        dealInfo.setFromCoinList(fromCoinList);

        dealInfo.setDealCost(1.2);


        service.deal(dealInfo, allianceNodeID);
      }
    }
```

该测试类已经测试完成，代码行的单元测试覆盖率为 98.3%，如图 8-1 所示。

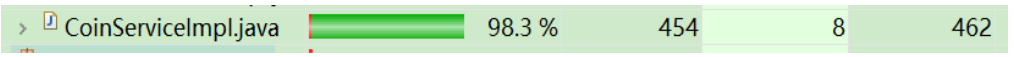


图 8-1　单元测试代码行覆盖率

8.4　小结

本章先后对比特币和以太坊系统中币的设计进行了说明，随后展示了 Java 版联盟链中币的设计和实现。

目前已有的联盟链中几乎没有涉及币或 token，但不排除日后会出现应用币或 token 的联盟链。

不论在联盟链中是否有币或 token，为了更好地解决联盟面向行业的既有问题，联盟内部还是需要有相应的激励体系来激活联盟链的长久健康发展。

第 9 章
联盟链管理后台

随风潜入夜
润物细无声

“欲知平直，则必准绳；欲知方圆，则必规矩”，又所谓“不以规矩，不能成方圆”，在联盟链中，联盟成员协作与共享需建立在公认的规矩之上。这里的规矩不仅仅是联盟协议的纸质版本，更有技术层面的规矩。而这些规矩，可以通过联盟链管理后台来实现。

作为联盟链的代表，超级账本给其他联盟链提供了不少借鉴。在联盟成员管理方面亦是如此。

本章首先介绍超级账本的成员管理模块的设计，随后我们用 Java 实现一种成员管理服务。

9.1　超级账本的成员管理

超级账本是 Linux 基金会于 2015 年 12 月发起的区块链开源项目。在超级账本的成员中，有国内外多家公司的身影，如 IBM、梅赛德斯 • 奔驰、印度喀拉拉邦区块链学院、埃森哲（Accenture）、空中客车、英特尔（Intel）、R3、摩根大通等，国内的百度、点融网、迅雷、中国印钞造币总公司等也是超级账本的会员。目前超级账本的会员已经超过 240 家。

超级账本的会员实行级别制管理，分为首席会员和普通会员。首席会员每年需要缴纳的会费为 25 万美元；普通会员分为五级，需缴纳的会费从 5000 到 5 万美元不等。

在超级账本的成员管理中，成员是拥有网络唯一根证书的合法独立实体。超级账本提供了 MSP（Membership Service Provider）和成员服务（Membership Services）。

MSP 是指为 Client 和 Peer 提供证书的组件。Client 用证书来认证他们的交易；Peer 用证书认证其交易背书。

成员服务用于在许可的区块链网络上认证、授权和管理会员身份。在 Peer 和 Order 中运行的成员服务的代码都需要认证和授权区块链操作。它是基于 PKI 的 MSP 实现。Fabric-CA 组件实现了用成员服务来管理身份。Fabric-CA 对 ECert 和 TCert 的颁发和撤销进行管理。ECert 是长期的身份凭证；TCert 是短期的身份凭证，是匿名且不可链接的。

如图 9-1 所示，超级账本的成员管理模块有会员注册、身份保护、内容保密和交易审核，以此来保证访问平台的安全性。

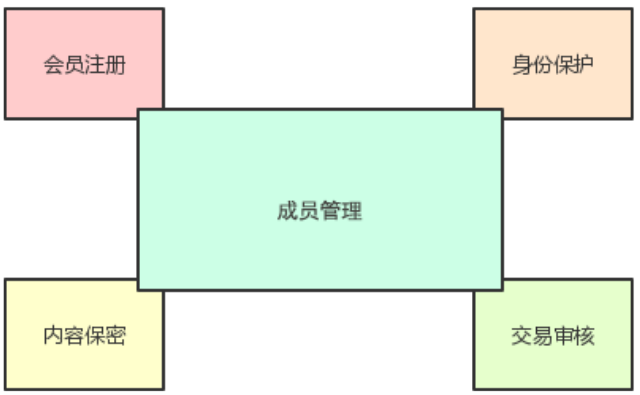


图 9-1　超级账本成员管理模块划分

9.2　Java 版联盟链成员管理设计与实现

结合超级账本的成员管理思路，我们设计的 Java 版联盟链成员管理服务模块如图 9-2 所示。

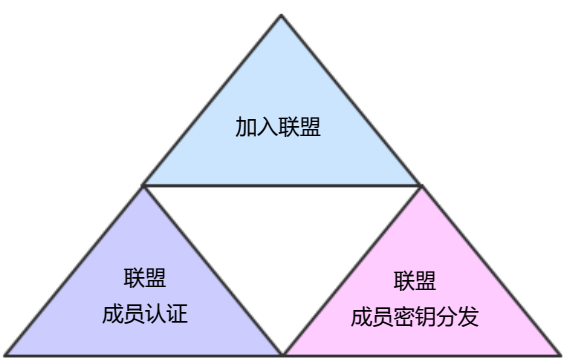


图 9-2　Java 版联盟链成员管理服务模块划分

Java 版联盟链成员管理服务模块分为三个子模块，分别是加入联盟模块、联盟成员认证模块、联盟成员密钥分发模块。

下面将依次介绍各个模块的设计和 Java 代码实现。

9.2.1　加入联盟模块的设计与实现

加入联盟模块主要为联盟意向成员的加入申请提供服务，在页面功能上与加入联盟按钮相对应。

为了记录联盟意向成员的申请信息，我们需建立一张表来记录意向成员的申请信息。在该表中，申请加入的组织名称、组织联系人、组织联系人的手机号码、申请加入时间、申请处理结果等字段都是必需的。

我们可以设计如下的表结构：

```
-- ----------------------------
-- Table structure for `tb_user_tojoin`
-- ----------------------------
DROP TABLE IF EXISTS `tb_user_tojoin`;
CREATE TABLE `tb_user_tojoin` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `create_time` datetime NOT NULL,
  `org_name` varchar(255) NOT NULL,
  `org_phone` varchar(255) NOT NULL,
  `org_represent` varchar(255) NOT NULL,
  `result ` tinyint NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4;
```

其中，表结构名称为 tb_user_tojoin。tb_user_tojoin 的字段如下：

id 是表的自增主键，create_time 是申请加入时间，org_name 所包含的是申请加入的组织名称，org_phone 是申请加入的组织联系人的手机号码，org_ represent 是申请加入的组织联系人，result 代表申请结果（已处理是 1，未处理是 0，因此可以用 tinyint 实现）。

基于 Spring Boot 实现的 tb_user_tojoin 实体类如下：

```
package com.niudong.demo.dao.entity;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "tb_user_tojoin")
public class JoinToUsEntity implements Serializable {

  private static final long serialVersionUID = -6550777752269466791L;
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // 机构名称
  private String orgName;

  // 机构联系方式
  private String orgPhone;

  // 机构联系人
  private String orgRepresent;

  // 申请时间
  private Date createTime = new Date();

  // 申请结果：已处理是 1，未处理是 0
  private int result = 0;

  public int getResult() {
    return result;
  }

  public void setResult(int result) {
    this. result = result;
  }
```

```
    public Date getCreateTime() {
      return createTime;
    }

    public void setCreateTime(Date createTime) {
      this.createTime = createTime;
    }

    public String getOrgRepresent() {
      return orgRepresent;
    }

    public void setOrgRepresent(String orgRepresent) {
      this.orgRepresent = orgRepresent;
    }

    public String getOrgPhone() {
      return orgPhone;
    }

    public void setOrgPhone(String orgPhone) {
      this.orgPhone = orgPhone;
    }

    public String getOrgName() {
      return orgName;
    }

    public void setOrgName(String orgName) {
      this.orgName = orgName;
    }

    public Long getId() {
      return id;
    }

    public void setId(Long id) {
      this.id = id;
    }
  }
```

表结构对应的实体编写完成后，我们基于 Spring Boot JPA 构建对应的 JpaRepository 类如下：

```
package com.niudong.demo.dao;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.niudong.demo.dao.entity.JoinToUsEntity;

@Repository
public interface IJoinToUsDAO extends JpaRepository<JoinToUsEntity,
   Long> {
public JoinToUsEntity getByOrgNameAndOrgPhoneAndOrgRepresent (String
   orgName, String orgPhone,String orgRepresent);

public List<JoinToUsEntity> getByResult(int result);

}
```

在 IJoinToUsDAO 中，我们提供了根据申请组织名称、组织联系人和组织联系人的电话号码为查询条件的 getByOrgNameAndOrgPhoneAndOrgRepresent 接口，该接口用来查看特定组织成员的申请处理情况。

还提供了根据处理结果查询对应申请结果列表的 getByResult 方法。

而成员申请信息的保存和更新都可以基于 JPA 原生方法 save()实现。

下面展示加入联盟模块的 Controller 层代码：

```
package com.niudong.demo.controller;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import org.testng.util.Strings;

import com.niudong.demo.service.JoinToUsService;


@RestController
@RequestMapping("/api/v1/member")
public class JoinToUsController {

  protected static Logger logger =
    LoggerFactory.getLogger(JoinToUsController.class);

  @Autowired
  private JoinToUsService joinToUsService;

  @RequestMapping("/join")
  // String orgName:机构，String orgPhone：联系人手机号码，String
  // orgRepresent:机构联系人
  public String join(String orgName, String orgPhone, String
      orgRepresent) {
    if (Strings.isNullOrEmpty(orgName) ||
        Strings.isNullOrEmpty(orgPhone)
        || Strings.isNullOrEmpty(orgRepresent)) {
      return "请将机构名称、机构联系人、联系人手机号码完整输入！";
    }

    if (!isMobileOrPhone(orgPhone)) {
      return "联系人手机号码格式不正确！";
    }

    joinToUsService.join(orgName, orgPhone, orgRepresent);
    return "success";
  }

  // 验证是否是有效的电话或手机
  private boolean isMobileOrPhone(String orgPhone) {
    boolean isMoblie = isMobile(orgPhone);
    if (isMoblie == true) {
      return true;
    }

    return isPhone(orgPhone);
```

```
    }

    // 验证是否为手机号码
    private boolean isMobile(String str) {
      Pattern pattern = Pattern.compile("^[1][3,4,5,8][0-9]{9}$");
      Matcher matcher = pattern.matcher(str);
      return matcher.matches();
    }

    // 验证是否是座机电话号码
    private boolean isPhone(String str) {
      // 验证带区号的
      Pattern p1 = Pattern.compile("^[0][1-9]{2,3}-[0-9]{5,10}$");
      // 验证没有区号的
      Pattern p2 = Pattern.compile("^[1-9]{1}[0-9]{5,8}$");

      Matcher m = null;
      boolean b = false;

      if (str.length() > 9) {
        m = p1.matcher(str);
        b = m.matches();
      } else {
        m = p2.matcher(str);
        b = m.matches();
      }

      return b;
    }
  }
```

在 Controller 层中，我们提供了通过 get 方法可以访问的 join 接口，该接口的入参为 String orgName、String orgPhone、String orgRepresent，分别表示申请机构、申请机构联系人的电话号码、申请机构的联系人。

在 join 的处理逻辑中，先进行输入参数的空检验。参数只要有 1 个为空，则输入数据不合法，后台向前端提示信息为“请将机构名称、机构联系人、联系人手机号码完整输入!”

输入参数校验均非空后，对手机号码进行有消息校验，通过 isMobileOrPhone 来

实现。isMobileOrPhone 先后调用 isMobile 和 isPhone 来分别校验输入的号码是座机号码还是手机号码。

手机号码校验失败时，后台向前端提示信息“联系人手机号码格式不正确!”。校验成功后，则调用 Service 层代码 joinToUsService.join(orgName, orgPhone, orgRepresent)来进行 Service 层的逻辑处理，即保存申请加入联盟的信息。下面展示 Service 层的代码逻辑。

Service 层采用面向接口编程方法，构建 JoinToUsService 的接口类，并构建 JoinToUsService 的实现类 JoinToUsServiceImpl，代码如下：

```
   package com.niudong.demo.service;

   /**
    * JoinToUsService 抽象类
    *
    * @author 牛冬
    **/
   public interface JoinToUsService {
     public void join(String orgName, String orgPhone, String
orgRepresent);
   }

   package com.niudong.demo.service.impl;

   import org.springframework.beans.factory.annotation.Autowired;
   import org.springframework.stereotype.Service;

   import com.niudong.demo.dao.IJoinToUsDAO;
   import com.niudong.demo.dao.entity.JoinToUsEntity;
   import com.niudong.demo.service.JoinToUsService;

   /**
    * JoinToUsService 实现类
    *
    * @author 牛冬
    *
    */
   @Service
   public class JoinToUsServiceImpl implements JoinToUsService {
```

```
    @Autowired
    private IJoinToUsDAO dao;

    // 增加机构
    public void join(String orgName, String orgPhone, String orgRepresent) {
      // 先检查这些信息是否已经录入
      JoinToUsEntity entity =
          dao.getByOrgNameAndOrgPhoneAndOrgRepresent(orgName, orgPhone,
              orgRepresent);

      // 若已经录入，则不再录入
      if (entity != null) {
        return;
      }

      // 尚未录入
      JoinToUsEntity joinToUsEntity = new JoinToUsEntity();
      joinToUsEntity.setOrgName(orgName);
      joinToUsEntity.setOrgPhone(orgPhone);
      joinToUsEntity.setOrgRepresent(orgRepresent);
      joinToUsEntity.setResult(0);

      dao.save(joinToUsEntity);
    }
  }
```

在 join 方法中，先校验机构申请信息是否已经录入，防止信息的重复录入。若信息已录入，则不必重复记录；若信息未录入，则构建 JoinToUsEntity 实体，通过 JPA 的 save()方法实现入库并保存数据，此时 JoinToUsEntity 对应的处理状态为 0，表示尚未处理。

9.2.2 联盟成员认证模块

联盟成员通过申请后，将进行联盟协议的签署。签署完毕，部署代码时，为了对联盟成员进行有效认证，需构建各个联盟部署服务器的白名单信息表。表结构设计如下：

```
  DROP TABLE IF EXISTS "tb_alliance";
  CREATE TABLE "tb_alliance"(
    "id" bigint(20) NOT NULL AUTO_INCREMENT,
```

```
  "alliance_id" varchar(255) DEFAULT NULL,
  "alliance_ip" varchar(255) DEFAULT NULL,
  "alliance_name" varchar(255) DEFAULT NULL,
  "create_time" datetime DEFAULT NULL,
  "update_time" datetime DEFAULT NULL,
  PRIMARY KEY ("id")
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
```

其中，表结构名称为 tb_alliance，表里面的字段主要有：id 表示表中的自增 id，alliance_id 表示联盟成员的 ID 号码，alliance_ip 表示联盟成员部署服务器的公网 IP 地址，alliance_name 表示联盟成员部署时配置的节点名称，create_time 表示联盟白名单信息表信息创建的时间，update_time 表示联盟白名单信息表信息更新的时间。

表设计完毕后，我们基于 JPA 构建 tb_alliance 的表结构对应的实体类如下：

```
package com.niudong.demo.dao.entity;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * 联盟的成员表
 *
 * @author niudong
 */
@Entity
@Table(name = "tb_alliance")
public class AllianceEntity {
  // 数据库 ID
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  // 联盟节点 id
  private String allianceId;
  // 联盟节点名称
  private String allianceName;
```

```
    // 联盟节点 IP 白名单
    private String allianceIp;
    // 联盟节点信息创建时间
    private Date createTime;
    // 联盟节点信息更新时间
    private Date updateTime;


    public Date getCreateTime() {
      return createTime;
    }

    public void setCreateTime(Date createTime) {
      this.createTime = createTime;
    }

    public Date getUpdateTime() {
      return updateTime;
    }

    public void setUpdateTime(Date updateTime) {
      this.updateTime = updateTime;
    }

    public Long getId() {
      return id;
    }

    public void setId(Long id) {
      this.id = id;
    }

    public String getAllianceId() {
      return allianceId;
    }

    public void setAllianceId(String allianceId) {
      this.allianceId = allianceId;
    }

    public String getAllianceName() {
      return allianceName;
```

```
    }

    public void setAllianceName(String allianceName) {
      this.allianceName = allianceName;
    }

    public String getAllianceIp() {
      return allianceIp;
    }

    public void setAllianceIp(String allianceIp) {
      this.allianceIp = allianceIp;
    }
  }
```

AllianceEntity 实体构建完毕后，我们构建 AllianceEntity 对应的 JPA 类，代码如下：

```
  package com.niudong.demo.dao;

  import org.springframework.data.jpa.repository.JpaRepository;
  import org.springframework.stereotype.Repository;

  import com.niudong.demo.dao.entity.AllianceEntity;

  /**
   * AllianceDAO
   *
   * @author 牛冬
   *
   */
  @Repository
  public interface IAllianceDAO extends JpaRepository<AllianceEntity,
      Long> {
    public AllianceEntity findByAllianceIp(String allianceIp);
  }
```

Service 层采用面向接口编程方法，构建 AllianceService 的接口类，并构建 AllianceService 的实现类 AllianceUsServiceImpl，代码如下：

```
  package com.niudong.demo.service;

  import java.util.List;
```

```
import com.niudong.demo.dao.entity.AllianceEntity;

/**
 * 联盟 Service
 *
 * @author 牛冬
 *
 */
public interface AllianceService {
  public AllianceEntity findByAllianceIp(String allianceIp);

  public void insertIntoAlliance(String allianceIp, String allianceId,
      String allianceName);

  public List<AllianceEntity> selectAllIp();
}


package com.niudong.demo.service.impl;

import java.util.Date;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.niudong.demo.dao.IAllianceDAO;
import com.niudong.demo.dao.entity.AllianceEntity;
import com.niudong.demo.service.AllianceService;

/**
 * 联盟 Service 实现类
 *
 * @author 牛冬
 *
 */
@Service
public class AllianceServiceImpl implements AllianceService {
  protected static Logger logger =
      LoggerFactory.getLogger(AllianceServiceImpl.class);
```

```
    @Autowired
    private IAllianceDAO allianceDAO;

    public AllianceEntity findByAllianceIp(String allianceIp) {
      return allianceDAO. findByAllianceIp(allianceIp);
    }

    public void insertIntoAlliance(String allianceIp, String allianceId,
        String allianceName) {
      AllianceEntity allianceEntity = new AllianceEntity();
      allianceEntity.setAllianceId(allianceId);
      allianceEntity.setAllianceIp(allianceIp);
      allianceEntity.setAllianceName(allianceName);
      allianceEntity.setCreateTime(new Date());
      allianceEntity.setUpdateTime(new Date());

      allianceDAO.save(allianceEntity);
    }

    public List<AllianceEntity> selectAllIp() {
      return allianceDAO.findAll();
    }
  }
```

从上述代码可见，Service 层 AllianceUsServiceImpl 类提供了基于 IP 查找对应的联盟成员白名单信息的接口，该接口被 Controller 层调用。Controller 层代码如下所示：

```
  package com.niudong.demo.controller;

  import org.testng.util.Strings;

  import java.util.List;

  import javax.servlet.http.HttpServletRequest;

  import org.slf4j.Logger;
  import org.slf4j.LoggerFactory;
  import org.springframework.beans.factory.annotation.Autowired;
  import org.springframework.web.bind.annotation.RequestMapping;
  import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.bind.annotation.RestController;

import com.alibaba.fastjson.JSON;
import com.google.common.base.Splitter;
import com.niudong.demo.dao.entity.AllianceEntity;
import com.niudong.demo.service.AllianceService;
import com.niudong.demo.service.KeySuitService;
import com.niudong.demo.util.IpUtil;

@RestController
@RequestMapping("/api/v1/alliance")
public class AllianceController {
  protected static Logger logger = LoggerFactory.getLogger
     (AllianceController.class);

  @Autowired
  private AllianceService allianceService;
  @Autowired
  private KeySuitService keySuitService;

  @RequestMapping(value = "/selectAllIp", method = RequestMethod.GET)
  public List<AllianceEntity> selectAllIp(String
     allianceIp,HttpServletRequest request) {
    // 如果 IP 数据为空
    if (Strings.isNullOrEmpty(allianceIp)) {
      return null;
    }

    // IP 格式不对
    if (!isIp(allianceIp)) {
      return null;
    }

    //如果输入的 IP 与请求的 IP 来源不一致，说明是非法请求
    if(!IpUtil.getIpAddr(request).equals(allianceIp)) {
      return null;
    }

    return allianceService.selectAllIp();
  }

  @RequestMapping(value = "/selectByIp", method = RequestMethod.GET)
```

```
    public AllianceEntity selectByAllianceIp(String allianceIp) {
      //如果 IP 数据为空
      if(Strings.isNullOrEmpty(allianceIp)) {
        return null;
      }

      // IP 格式不对
      if (!isIp(allianceIp)) {
        return null;
      }

      //如果输入的 IP 与请求的 IP 来源不一致，说明是非法请求
      if(!IpUtil.getIpAddr(request).equals(allianceIp)) {
        return null;
      }

      return allianceService. findByAllianceIp(allianceIp);
    }

  @RequestMapping(value = "/insert", method = RequestMethod.GET)
    public String insertIntoAlliance(String allianceIp, String
       allianceId, String allianceName) {
      // 如果 IP 数据为空
      if (Strings.isNullOrEmpty(allianceIp) ||
          Strings.isNullOrEmpty(allianceId)
         || Strings.isNullOrEmpty(allianceName)) {
        return "输入参数有空值，请将所有内容全部填写完整。";
      }

      // IP 格式不对
      if (!isIp(allianceIp)) {
        return "IP 地址格式不对。";
      }

      allianceService.insertIntoAlliance(allianceIp, allianceId,
         allianceName);;
      return JSON.toJSONString(keySuitService.getRandowKeySuit());
    }

    // 验证是否是 IP 地址
    private boolean isIp(String allianceIp) {
      // IP 地址不能为空
```

```
        if (Strings.isNullOrEmpty(allianceIp)) {
          return false;
        }

        // IP 地址是用.分隔的
        if (!allianceIp.contains(".")) {
          return false;
        }

        // IP 地址是 123.123.123.123 格式才行
        if (!allianceIp.matches("\\d{1,3}\\.\\d{1,3}\\.
            \\d{1,3}\\.\\d{1,3}")) {
          return false;
        }

        List<String> list = Splitter.on(".").splitToList(allianceIp);
        for (String str : list) {
          int value = Integer.parseInt(str);
          if (value < 255 && value > 0) {
            continue;
          }
          return false;
        }

        return true;
      }
    }
```

从 HttpServletRequest 获取 IP 地址的工具类 IpUtil 的代码如下所示：

```
    package com.niudong.demo.util;

    import java.net.InetAddress;

    import javax.servlet.http.HttpServletRequest;

    /**
     * IP 获取工具类
     *
     * @author 牛冬
     *
     */
    public class IpUtil {
```

```
public static String getIpAddr(HttpServletRequest request) {
  String ipAddress = null;

  try {
    ipAddress = request.getHeader("x-forwarded-for");
    if (ipAddress == null || ipAddress.length() == 0 ||
        "unknown".equalsIgnoreCase(ipAddress)) {
      ipAddress = request.getHeader("Proxy-Client-IP");
    }

    if (ipAddress == null || ipAddress.length() == 0 ||
        "unknown".equalsIgnoreCase(ipAddress)) {
      ipAddress = request.getHeader("WL-Proxy-Client-IP");
    }

    if (ipAddress == null || ipAddress.length() == 0 ||
        "unknown".equalsIgnoreCase(ipAddress)) {
      ipAddress = request.getRemoteAddr();
      if (ipAddress.equals("127.0.0.1")) {
        // 根据网卡取本机配置的 IP
        InetAddress inet = null;
        try {
          inet = InetAddress.getLocalHost();
        } catch (Exception e) {
          e.printStackTrace();
        }
        ipAddress = inet.getHostAddress();
      }
    }

    // 对于通过多个代理的情况，第一个 IP 为客户端真实 IP
    if (ipAddress != null && ipAddress.length() > 15) {

      if (ipAddress.indexOf(",") > 0) {
        ipAddress = ipAddress.substring(0, ipAddress.indexOf(","));
      }
    }

  } catch (Exception e) {
    ipAddress = "";
  }
```

```
        return ipAddress;
      }
    }
```

从上述代码可见，Controller 层提供了用于 IP 白名单验证的 selectByIp 接口，该接口的输入参数为联盟接口的 IP 字符串 allianceIp。此外，还有联盟成员数据插入表中的接口 insertIntoAlliance()和获取所有联盟成员部署服务器 IP 的接口 selectAllIp()。

selectByIp 接口在逻辑处理时，先对 allianceIp 的有效性进行校验，若校验失败，则返回给接口调用方空结果；若校验成功，则校验 IP 格式是否正确。若 IP 格式不正确，则返回给接口调用方空结果；若 IP 格式正确，则继续校验 IP 数据与 Request 请求中的 IP 是否一致。若不一致，则返回给接口调用方空结果；若一致，则调用 Service 层接口 selectByAllianceIp，获取 IP 白名单信息。

该接口在各联盟节点启动时调用，以此接口的返回结果来对联盟成员的合法性进行认证。

联盟成员数据插入表中的接口 insertIntoAlliance 的输入参数分别是联盟成员的 ID、联盟成员部署服务器的公网 IP、联盟成员名称。

进行逻辑处理时，先对 3 个输入参数的有效性进行校验，若校验失败，则返回接口调用方错误信息，错误信息为“输入参数有空值，请将所有内容全部填写完整。”若校验成功，则校验 IP 格式是否正确；若校验失败，则返回给接口调用方错误信息，错误信息为“IP 地址格式不对。”若校验成功，则执行 allianceService.insertIntoAlliance()方法完成联盟成员信息的写入。同时，将分配给该联盟成员的公私钥对返回给接口调用方，以便给联盟成员分配公私钥。

获取所有联盟成员部署服务器 IP 的接口 selectAllIp()的输入参数为联盟接口的 IP 字符串 allianceIp。此处接口逻辑处理与前面相同，不再赘述。

当联盟链 P2P 网络构建的底层采用 HTTP 协议、RPC 协议或 WebSocket 协议时，可以通过该接口获取所有联盟成员部署服务器的 IP，以便分别和这些服务器建立通信连接。

9.2.3　联盟成员密钥分发模块

区块链中各节点生成区块数据的传输需要解密。为了防止各个节点的加密数据混淆，需要为各个联盟节点分配不同的公私钥。因此联盟链系统需要有 CA 角色的模块，即联盟成员密钥分发模块。该模块为各个节点分配公私钥，公私钥用于联盟中各节点加密自己的数据，

联盟链的管理后台可以不保存各个节点的公私钥（保存也可以），仅需要进行验证。

密钥分发通过接口实现，下面依次从 controller 层、service 层展示相关代码。

controller 层用于提供密钥分发及密钥定期更新的入口，接口实现如下：

```
package com.niudong.demo.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.niudong.demo.vo.KeySuitVO;
import com.niudong.demo.service.KeySuitService;

/**
 * 公私密钥套装生成 Controller
 *
 * @author 牛冬
 *
 */
@RestController
@RequestMapping("/api/v1/key")
public class KeySuitController {
  protected static Logger logger =
      LoggerFactory.getLogger(KeySuitController.class);

  @Autowired
  private KeySuitService keySuitService;

  @GetMapping("/change")
```

```
    public KeySuitVO change() {
      return keySuitService.getRandowKeySuit();
    }
}
```

如上述代码所示，controller 层调用 service 层分发 getRandowKeySuit，实现公私钥对获取并返回给接口调用方使用。

Service 层采用面向接口编程的方法，构建 KeySuitService 的接口类，并构建 KeySuitService 的实现类 KeySuitServiceImpl，代码如下：

```
package com.niudong.demo.service;

import com.niudong.demo.vo.KeySuitVO;

/**
 * 公私密钥套装生成 Service
 *
 * @author 牛冬
 *
 */
public interface KeySuitService {
  public KeySuitVO getRandowKeySuit();
}


package com.niudong.demo.service.impl;

import org.springframework.stereotype.Service;

import com.niudong.demo.service.KeySuitService;
import com.niudong.demo.vo.KeySuitVO;

import cn.hutool.crypto.asymmetric.RSA;

/**
 * 公私密钥套装生成 Service 实现类
 *
 * @author 牛冬
 *
 */
@Service
```

```
public class KeySuitServiceImpl implements KeySuitService {

  // 返回公私密钥对
  public KeySuitVO getRandowKeySuit() {
    RSA rsa = new RSA();
    // 获得私钥
    String privateKey = rsa.getPrivateKeyBase64();
    // 获得公钥
    String publicKey = rsa.getPublicKeyBase64();

    KeySuitVO vo = new KeySuitVO(privateKey, publicKey);
    return vo;
  }
}
```

用于传输公私钥信息的 KeySuitVO 实体实现代码如下：

```
package com.niudong.demo.vo;

/**
 * 公私钥对
 *
 * @author 牛冬
 *
 */
public class KeySuitVO {
  // 私钥
  private String privateKey;
  // 公钥
  private String publicKey;

  public KeySuitVO() {

  }

  public KeySuitVO(String privateKey, String publicKey) {
    this.privateKey = privateKey;
    this.publicKey = publicKey;
  }

  public String getPublicKey() {
    return publicKey;
  }
```

```
    public void setPublicKey(String publicKey) {
      this.publicKey = publicKey;
    }

    public String getPrivateKey() {
      return privateKey;
    }

    public void setPrivateKey(String privateKey) {
      this.privateKey = privateKey;
    }
  }
```

9.3 小结

本章先介绍了超级账本的成员管理模块，随后通过加入联盟模块、联盟成员认证模块、联盟成员密钥分发模块展示了 Java 版联盟链成员管理的设计与实现。

在实际业务场景中，联盟链管理后台会有更复杂的逻辑和业务，如联盟成员部署的各个节点数据同步状态监控、密钥的定期更新等。读者需要根据自己的业务场景进行必要的补充设计。

第 10 章
联盟链的运营

以至诚为道
以至仁为德

在当前大部分互联网公司里，运营人员的工作主要围绕三部分展开：用户数、活跃度、赢收。

用户数指的是系统中用户累计的数量，活跃度指的是系统中新老用户的使用情况，赢收一般指的是广告和流量收入等。在用户数这一维度，互联网公司往往关心的是新用户的增加情况，俗称“拉新”；而活跃度往往和老用户的留存有关，俗称“留存”，留存率是活跃度中最受关注的指标，一般有“三日留存”“七日留存”“月度留存”等指标；如果老用户流失较多，运营人员还要想方设法让老用户重新来使用系统，俗称“促活”。

同样，在区块链的联盟链领域，也需要有运营人员甚至商务拓展人员。

运营人员和商务拓展人员的工作有联盟节点的拓展，联盟各个节点数据监控与系统稳定性、准确性支持，系统技术升级支持，新上线功能的宣传与答疑等。

本章中的联盟链运营主要从技术维度展开，对其他维度的运营工作不进行过多阐述。

10.1 联盟链会员章程

联盟链，顾名思义是需要多个联盟节点参与的区块链，各个联盟成员在统一的章程或者行为准则下开展工作。因此联盟链在拓展第二个联盟节点前就需要有明确的用于各个联盟成员统一践行的章程或者行为。超级账本章程可以供我们参考。

超级账本章程名为 *Hyperledger Project Charter*，以下简称 HPC。

在 HPC 中，规定了超级账本项目的任务（Mission of Hyperledger Project，简称 HLP）、会员身份（Membership.）、联盟理事会（Governing Board）、技术指导委员会（Technical Steering Committee）、营销委员会（Marketing Committee）、最终用户技术咨询委员会（End User Technical Advisory Board）、选举（Voting）、反托拉斯准则（Antitrust Guidelines）、行为守则（Code of Conduct）、预算（Budget）、一般费用及行政费用（General & Administrative Expenses）、一般规则和业务（General Rules and Operations）、知识产权政策（Intellectual Property Policy）、修正案（Amendments）14 部分，各部分内容要点总结如下。

1. 超级账本项目的任务

1）创建一个企业级、开源分布式分类账框架和代码库，用户可以在此基础上构建和运行强大的、行业特定的应用程序、平台和硬件系统来支持业务交易。

2）创建一个开源的技术社区，使 HLP 解决方案提供商和用户的生态系统受益，重点关注区块链和共享账本用例，这些用例将适用于各种行业解决方案。

3）促进生态系统主要成员的参与，包括开发人员、服务和解决方案提供商以及最终用户。

4）主持 HLP 的基础设施建设，为社区基础设施、会议、活动和协作讨论建立中立的环境，并围绕 HLP 的业务和技术治理提供解决方案。

2. 会员身份

一个 HLP 应该有主要会员（Premier Member）、普通会员（General Member），以及无须缴纳费用但无投票权的准会员（Associate Member）三种。只有 Linux 基金会的现任成员，才能作为成员参与 HLP。在 HLP 内部，任何人都可以提议，无论其成员身份如何。HLP 的所有参与者都享有本超级账本项目章程中所述的特权，并承担该章程中所述的义务。该章程可以由董事会在 Linux 基金会的批准下修订。成员在任期内需遵守 Linux 基金会、董事会或 HLP 在通知成员的情况下通过的所有政策。

准成员类别的成员仅限于非营利、开放源码项目和政府实体，并需要 HLP 理事会批准。如果理事会设定了加入准成员的标准，则准成员需要满足这些标准。

主要会员有权任命一名代表参加董事会、营销委员会和董事会设立的任何其他委员会。

普通会员有权每年在每 10 名普通会员中选举 1 名代表进入理事会，最多两名代表。此外，普通会员即使少于 10 名，总会员代表也应至少有 1 名。选举过程由理事会决定。

主要会员、普通会员和准会员有权参加项目大会、倡议、活动和任何其他活动，有权确认自己是 HLP 的成员或参与者。

3. 联盟理事会

理事会中投票成员最多有 21 名主要成员。

会议举行时应遵守以下原则：

1）董事会会议应仅限于董事会代表，并遵循本章程要求的法定人数和投票要求。理事会可决定是否允许 1 名指定代表作为候补出席。

2）除非理事会批准，否则理事会会议内容应保密。理事会可邀请客人参加对理事会特定议题的审议（但此类客人不得参加理事会对任何事项的投票）。理事会应鼓励会议的透明度，包括在理事会批准后的合理时间内公开发布会议记录。

董事会的职责如下：

1）批准一项预算，指导使用 HLP 筹集的资金。

2）选举高级别委员会主席主持理事会会议，批准预算，管理所有日常业务。

3）监督所有项目业务和营销事宜。

4）通过并维护 HLP 的政策或规则和程序（须经 LF 批准），包括但不限于行为守则、商标政策，以及合规或认证政策。

5）定义和管理认证程序，包括 HLP 的所有项目认证或流程。

6）支持最终用户采用、参与技术社区对话和全面参与 HLP。

7）批准提名和选举理事会的一般成员代表人员和理事会设立的所有官员或其他职位的程序。

8）对提交理事会的所有决定或事项进行表决。

4. 技术指导委员会

技术委员会的组织结构如下：

1）项目启动期：在项目启动后的前六个月内，投票成员应包括 1 名来自主要成员和每个顶级项目维护者的指定代表，条件是任何机构的投票不得超过 3 票。

2）项目稳定状态：启动期之后，将有一个提名和选举期来选举技术指导委员会

的贡献者或维护者。技术指导委员会投票成员应由 11 名由活跃贡献者选出的贡献者或维护者组成。“积极贡献者”是指在过去 12 个月内接受代码库贡献的任何贡献者。技术委员会应批准每年举行的提名和选举的过程和时间。

TSC 项目通常会涉及维护者和贡献者：

1）贡献者：技术社区中向 HLP 代码库贡献代码、文档或其他技术工件的任何人。

2）维护人员：能够向 HLP 项目的主要分支提交代码和贡献的贡献者。贡献者可以通过现有维护者的多数批准成为维护者。

任何人都可以通过成为贡献者和/或维护者参与 HLP。技术指导委员会可：

1）为项目的提交、批准和关闭或归档建立工作流程和程序。

2）建立促进维护者、贡献者的标准和流程。

3）修改、调整和完善 HLP 代码库的贡献者和维护者的角色规则，创建新的角色，并公开记录这些角色的职责和期望，视情况而定。

技术指导委员会应选举 1 名技术指导委员会主席，该主席也将担任理事会的投票成员，预计将担任理事会和高级别委员会技术领导之间的联络人。

技术指导委员会的主要职责如下：

1）协调 HLP 的技术方向。

2）根据技术指导委员会将要开发、批准和维护的项目生命周期文件，批准项目建议书。

3）指定顶级项目。

4）设立小组委员会或工作组，集中处理跨项目的技术问题或机会。

5）在需求、体系结构、实施、用例等方面，协调技术团体与最终用户团体和任何最终用户的合作。

6）就项目技术事项与外部和行业组织进行沟通。

7）任命代表与其他开源或标准社区合作。

8）发布和建立社区规范、工作流程或政策。

9）讨论、寻求共识，并在必要时就影响多个项目的代码库相关的技术事项进行表决。

10）为不在任何单一项目范围内的技术团体中的维护者或其他领导角色建立选举程序。

5. 营销委员会

营销委员会组织结构应包括：

1）每名主要成员任命 1 名投票代表。

2）无表决权的代表，由任何其他类别的成员任命。

3）技术指导委员会任命的任何无表决权的维护者。

营销委员会的职责如下：

1）营销委员会应代表董事会负责设计、开发和执行营销工作。

2）营销委员会将与管理委员会、最终用户和技术团体密切协调，最大限度地扩大 HLP 在整个行业的知名度。

6. 最终用户技术咨询委员会

最终用户被定义为运行或打算运行应用程序或服务的任何公司（或个人）。同时，最终用户提供的行业解决方案中，如果包含了 LP 技术所产出的行业解决方案，则不能提供或出售给他人。

最终用户技术咨询委员会由最终用户的 HLP 成员组成。最终用户技术咨询委员会应批准新的最终用户成员。理事会将批准足以设立最终用户技术咨询委员会的一组最终用户成员。

最终用户技术咨询委员会应通过会议、邮件列表或创建特殊兴趣小组来协调最终用户的努力。

最终用户技术咨询委员会可以决定任何会议或其他最终用户活动是仅限于成员

还是对非成员参与者开放。

7. 选举

虽然 HLP 的目标是作为一个基于共识的社区运作，但任何决定都需要投票才能推进，董事会、技术指导委员会、营销委员会和最终用户技术咨询委员会的代表应在每名投票代表的基础上再投一票。

理事会、技术指导委员会、营销委员会或最终用户技术咨询委员会会议的法定人数至少需要三分之二的投票代表。如果未达到法定人数，理事会、技术指导委员会、营销委员会或最终用户技术咨询委员会可继续开会，但不能在会议上做出任何决定。任何连续两次未能出席理事会会议的理事会投票代表，在第三次连续会议上，直到他们下次出席理事会会议，都不会被计算在内，以确定法定人数要求。

在会议上通过表决作出的决定需要多数票，前提是达到法定人数。未经会议通过电子表决作出的决定应要求所有表决代表的多数。

如果对理事会无法解决的行动进行了票数相等的表决，主席有权将此事提交给 Linux 基金会，以帮助其作出决定。对于技术指导委员会、营销委员会或理事会设立的其他委员会的所有决定，如果票数相等，那么该事项应提交理事会。

提议理事会在会议上通过的所有决议，除通过会议记录的决议外，应至少在会议日期前两个工作日以草案形式分发给理事会成员，此类表决草案的文本可在会议上修改。

8. 反垄断政策

所有成员都应遵守 Linux 基金会反垄断政策。

所有成员应鼓励任何能够满足成员资格要求的组织公开参与，无论其竞争利益如何。

换句话说，除在非歧视的基础上对所有成员适用的合理标准、要求或理由外，理事会不应试图排斥任何成员。

9. 行为守则

理事会通过的任何一项具体的项目行为守则，都需要得到 Linux 基金会的批准。

10. 预算

理事会应批准年度预算，承诺支出不能超过筹集的资金。预算及其应用的目的应该与 Linux 基金会的非营利使命相一致。

Linux 基金会应向理事会定期报告预算支出水平。任何情况下，Linux 基金会都没有义务承担如下工作：如由 HLP 提出的代表 HLP 与 HLP 相关的、但不能由 Linux 基金会全部承担的工作。

如果与 HLP 相关的任何未编入预算或没有资金的债务出现，Linux 基金会都将与理事会协调解决缺口资金需求。

11. 一般费用及行政费用

Linux 基金会对任何费用、资金和其他现金收入的使用拥有保管权和最终使用权。

Linux 基金会将对为支付财务、会计和运营而筹集的资金征收一般管理费。一般管理费用应相当于 HLP 最初 1000000 美元总收入的 9%，以及 HLP 超过 1000000 美元总收入的 6%。

任何情况下，都不应期望或要求 Linux 基金会代表 HLP 采取任何不符合 Linux 基金会免税目的的行动。

12. 一般规则和业务

HLP 项目的实施应确保：

1）以专业的方式参与项目的工作，保持社区的凝聚力，同时保持开源软件社区中 Linux 基金会的善意和尊重。

2）尊重所有商标的所有人权利，包括任何品牌和使用指南。

3）让 Linux 基金会参与所有 HLP 新闻和其他相关活动。

4）应要求向 Linux 基金会提供关于项目参与的信息，包括关于参加项目赞助活动的信息。

5）就任何直接为 HLP 创建的网站与 Linux 基金会协调。

6）根据董事会批准并经 Linux 基金会确认的规则和程序运作。

13. 知识产权政策

成员们同意，所有新的 HLP 入站代码贡献都应在 Apache 许可证 2.0 版下进行。所有捐款应附有通过理事会和 Linux 基金会批准的捐款程序提交的开发商原产地证书签字。这种贡献过程将包括非会员贡献者，如果不是自由职业者，那么他们的雇主需认同 Apache 许可证 2.0 版的内容。

所有开源代码将在 Apache 许可证 2.0 版下提供。

所有文件都将由 HLP 根据 *Creative Commons Attribution 4.0 International License* 提供。

如果开源项目的许可需要替代入站或出站许可，或者为了实现 HLP 的使命需要替代入站或出站许可，理事会可以例外地批准使用替代许可用于特定的入站或出站贡献。任何例外都必须得到整个理事会和 Linux 基金会三分之二投票的批准，并且必须限于为此目的所需的范围。请发电子邮件给 legal@hyperledger.org 以获得例外批准。

根据可用的项目资金，HLP 可能会聘请 Linux 基金会来确定 Linux 基金会拥有的商标、服务标志和认证标志的可用性和注册。

14. 修正案

经 Linux 基金会批准，这个章程可以由整个董事会三分之二的投票来修改。

10.2　联盟链代码使用方式

联盟链中的代码根据联盟各个节点的技术储备情况，可以有以下三种使用方式。

方式 1　全封闭式

在这种方式中，联盟中其他节点往往没有区块链底层相关技术研发能力或能力较弱，或没有相关人员实施系统运维，联盟链中的超级节点（一般都是联盟的发起方或代码的主要贡献方）提供技术研发、代码部署的环境、代码部署及服务器维护，即 BAAS（Blockchain as a Service），如图 10-1 所示。

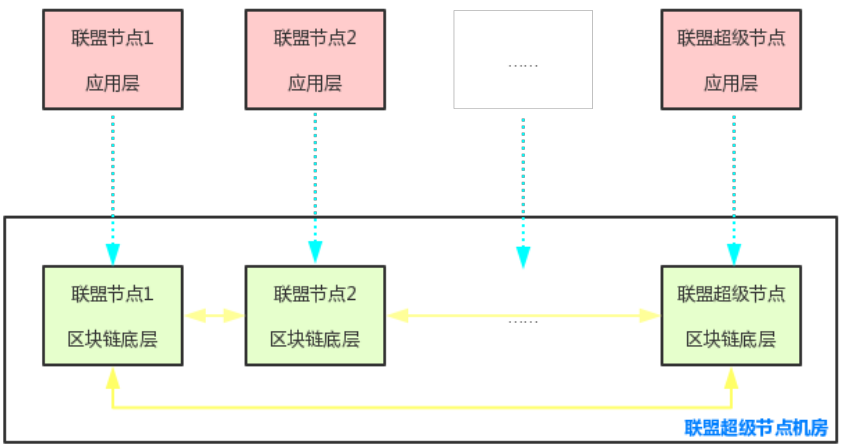


图 10-1　全封闭式联盟链部署结构

此时，区块链底层服务全部封闭于超级节点内部，因此称之为全封闭式。

超级节点可以向其他节点收取服务器使用费用、服务器管理费用、服务器使用的网络宽带费用、服务器升级费用等，并向联盟中其他节点提供节点各自服务器的 IP 地址，以便节点根据各自节点部署服务的 IP 地址和上链接口的要求拼接数据，发出上链请求后将数据上链。

在这种场景下，各个节点都在超级节点的机房部署各自的一套节点服务器，只不过服务器的管理权限交由超级节点实施。也就是除超级节点外，各个节点均有各自的区块链底层服务，但仅有所有权，没有使用权，使用权交接给超级节点实施。

联盟中的其他节点可透明使用区块链的底层服务，专注于区块链应用层的业务研发。

方式 2　半封闭半开放式

在这种方式中，联盟中其他节点往往有一定区块链相关技术研发能力，但没有相关人员实施区块链底层系统的运维，联盟链中的超级节点提供数据上链相关的 API 供其他节点接入，如图 10-2 所示。

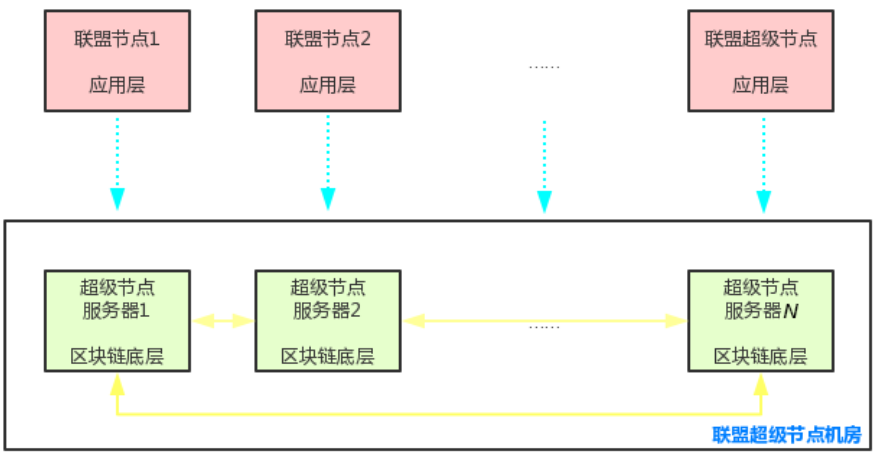


图 10-2　半封闭半开放式联盟链部署结构

此时，区块链底层服务半封闭于超级节点内部，因此称之为半封闭式半开放式。

超级节点可以向其他节点按调用频次收取服务使用费用等。

此时，各个节点除超级节点外，并没有专门的区块链底层服务器部署程序，而超级节点部署了多个节点，这些节点完全归超级节点所有，也就是除超级节点外，各个节点均没有各自的区块链底层服务，既没有区块链底层服务器的所有权，又没有使用权。

联盟中其他节点可通过 API 使用区块链的底层服务，以便专注于区块链应用层业务研发。

方式 3　全开放式

在这种方式中，联盟中其他节点和超级节点一样，有区块链相关技术研发能力，也有相关人员和服务器来实施区块链底层系统的部署和运维。联盟链中各个节点共同开发和维护区块链底层代码，如图 10-3 所示。

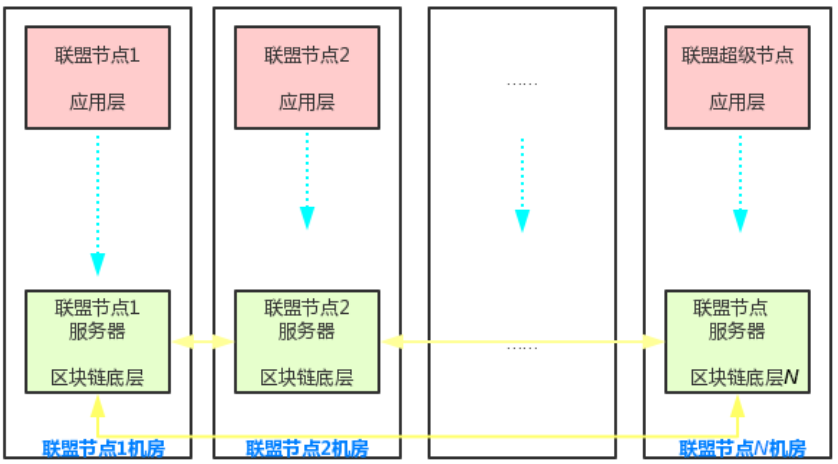


图 10-3　全开放式联盟链部署结构

区块链底层服务对所有节点完全透明，因此称之为全开放式。

联盟链中各个节点各自在自己的机房部署相同版本的区块链底层服务代码。也就是说，各个节点均有区块链底层服务器的所有权和使用权。但联盟链中各个节点需将各自部署的服务器相关信息（如公网、IP 等）上报超级节点，录入区块链底层服务监控平台。

同时，联盟链中各个节点均对区块链底层服务和应用层程序进行了开发。

10.3　联盟链代码升级

与区块链公链系统代码升级类似，我们在区块设计时，预留了区块版本字段。该字段主要在升级时使用，每次升级需将区块版本号加一。

联盟链系统处理交易信息时，若遇到不在当前版本处理逻辑范围内的交易信息，则不能对交易实施正确处理，此时可将这些交易信息写入本地固定文件，以便系统升级时优先处理这些未处理的交易信息。

此外，在联盟链中，无论是哪种代码使用方式（见 10.2 节），联盟中都可约定统一的上线时间。上线期间若业务不在线上运行，则不会有线上交易数据产生，因此可顺利实施上线。若上线期间业务属于正常的运行时间，则可将升级期间的交易信

息写入本地固定文件，以便系统升级时优先处理这些未处理的交易信息。

10.4　联盟链代码安全

与普通软件项目的代码安全一样，联盟链的代码同样面临代码安全和代码知识产权的问题。联盟链的底层代码和业务代码都是联盟成员内部的机密信息，因此，如何保证相关代码在联盟内外的安全性是运营层面不可回避的问题。

对外部而言，联盟内部可以设置 Git 仓库访问的 IP 白名单，这样联盟外部的人员和机构，包括联盟内部离职人员，就都不能访问 Git 仓库来下载和修改代码。对联盟内部离职人员，其 Git 仓库的访问账号也要在离职时及时清理。

需要为每个人设立一个登录账号，并设立不同的联盟链底层和应用层代码部署服务器的访问权限。当联盟内部人员离职时，同样要及时清理其对服务器的所有账号和访问权限信息。

此外，人员离职时，应避免其拷贝和带走相关代码，以免带来安全隐患和知识产权隐患。

对于每个联盟链中的节点，其使用的公私钥也要定期更新，如果部署服务器有变化，也要及时将新服务器的公网 IP 同步到联盟链的管理平台进行数据更新。公网 IP 的白名单机制能保证不在 IP 白名单内部的服务器不能正常启动。

当然，离职人员如果带走了代码，可以去掉 IP 白名单认证的逻辑，但可以要求每个访问联盟链底层和管理后台接口的参数中均必须携带所在服务器的公网 IP，若 IP 数据不存在或 IP 数据错误，则相关后台可拒绝提供接口服务。

10.5　联盟链激励体系运营

联盟链的激励体系即联盟链中发行的币或 Token。

在公链中，币的运营方式较为复杂。一般要根据自己发行的币和价格，围绕促销和营销的方式与渠道展开运作；同时，还要构建币值管理和运作团队，以便让币长久、健康、稳定运转，同时吸引更多的玩家进入系统；甚至还会组建相对专业的

金融操盘团队，同时结合币的社群运营，来激活发行币的生态。

但联盟链中币的运转方式与此大不相同。

在联盟链中，币更像是另一种形式的积分，只不过相对传统的积分体系，币是构建在区块链系统之上的，并且币是限量发行的，而积分是无限发行的。

因此，如何根据联盟的大小和运行周期来生产对应的币发行模式是非常关键的，同时由于币是限量发行的，因此如何让联盟中的币能被激活，并在联盟各个节点中流通起来，也是运营团队需要仔细设计的关键所在。

在这些维度，联盟链的运营人员可以参考各国央行货币投放量的动态模型设计。

10.6 小结

本章主要介绍了联盟链技术层面的运营，分别从联盟链会员章程制定、代码使用方式和升级方式、代码的安全问题、激励体系的运用展开阐述，其中联盟链会员章程制定主要向读者详细介绍了超级账本章程，为联盟链章程的制定提供一些参考和借鉴。

附录A

TextNG

TestNG 是什么

官方对 TestNG 的定义如下：

“TestNG 是一个测试框架，其灵感来自 JUnit 和 NUnit，但引入了一些新的功能，使其功能更强大，使用更方便。”

TestNG 是一个开源自动化测试框架。由 TestNG（Test Nest Generation）的名字就能知道，TestNG 想成为继 JUnit 后的新一代测试框架。但 TestNG 不是 JUnit 框架的简单功能扩展，虽然 TestNG 类似于 JUnit4，且设计灵感也来源于 JUnit，但它的设计目的是要优于 JUnit，特别是在测试集成多类时。

TestNG 消除了大部分旧框架的限制，引入了 Java 注解，使开发人员能够编写更加灵活且强大的测试。

TestNG 的主要特点如下：

- 广泛使用注解。
- TestNG 使用 Java 和面向对象的功能，让 Java 研发人员更易上手。
- 支持综合类测试。
- 配置灵活。

- 支持依赖测试方法、并行测试、负载测试和局部异常。
- 灵活的插件 API。
- 支持多线程测试。

TestNG 如何使用

在工程的 POM 文件中引入 TestNG 的依赖，依赖配置如下：

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.8.7</version>
    <scope>test</scope>
</dependency>
```

在介绍 TestNG 的使用方法前，先介绍 TestNG 支持的注解功能，注解列表如表 A-1 所示。

表 A-1　TestNG 支持的注解

注解名称	功能描述
@BeforeSuite	在该套件的所有测试运行前，执行该注解修饰的方法
@AfterSuite	在该套件的所有测试运行后，执行该注解修饰的方法
@BeforeClass	在调用当前测试类的第一个测试方法运行前，执行该注解修饰的方法
@AfterClass	在调用当前测试类的第一个测试方法运行后，执行该注解修饰的方法
@BeforeTest	被该注解修饰的方法将在属于@Test 范围内的类的所有测试方法运行之前执行
@AfterTest	被该注解修饰的方法将在属于@Test 范围内的类的所有测试方法运行之后执行
@BeforeGroups	被该注解修饰的方法在某个测试组第一个测试方法之前执行
@AfterGroups	被该注解修饰的方法在某个测试组第一个测试方法之后执行
@BeforeMethod	被该注解修饰的方法在某个测试方法之前执行
@AfterMethod	被该注解修饰的方法在某个测试方法之后执行
@DataProvider	被该注解修饰的方法用于提供测试用的公用数据。注释方法必须返回一个 Object [] []，其中，每个 Object []都可以分配给测试方法的参数列表。要想从该 DataProvider 接收数据的@Test 方法，需要使用与此注释名称相同的 DataProvider 名称
@Factory	将一个方法标记为工厂，返回的 TestNG 将被用作测试类的对象。该方法必须返回 Object[]

续表

注解名称	功能描述
@Listeners	定义测试类上的监听器
@Parameters	描述如何将参数传递给@Test 方法
@Test	将类或方法标记为测试的一部分

下面结合常用的场景来介绍 TestNG 的使用。

1）方法或程序运行出现异常，通过 throws 方式抛出

TestNG 提供的@Test 注解中可以配置 expectedExceptions 属性，来捕获方法或程序期望抛出的异常，示例如下。

整数相除，分母为 0，则抛出 ArithmeticException 异常。我们可以通过配置@Test 属性为@Test(expectedExceptions = ArithmeticException.class)来捕获该异常。

单元测试代码如下：

```
@Test(expectedExceptions = ArithmeticException.class)
  public void methodWithException() {
       int i = 1 / 0;
       System.out.println("i ="+ i);
}
```

2）忽略测试

TestNG 支持忽略测试，通过注释@Test 中的配置期属性 enabled = false 实现，即

```
@Test(enabled = false)
```

该配置特别适合由多人协作开发业务代码和单元测试代码的场景。在多人协作时，并不能实时保证大家提交的代码的完整性、可测试，因此可以在未完工的单元测试代码中增加@Test(enabled = false)来忽略该测试方法的执行，以保证当前研发人员自己负责代码的单元测试能正常进行。

3）依赖测试

TestNG 允许指定执行方法之间的依赖关系，可以通过如下两种方式落地：

- 在@Test 注释中使用属性 dependsOnMethods。
- 在@Test 注释中使用属性 dependsOnGroups。

@Test 的属性 dependsOnMethods 的实例代码如下：

```
@Test
public void method1() {
   System.out.println("Method 1");
}

@Test(dependsOnMethods = { "method1" })
public void method2() {
   System.out.println("Method 2");
}
```

单元测试代码执行时，method2 能否执行依赖于 method1 的执行结果，即如果 method1()能运行成功，那么将执行 method2()。

@Test 的属性 dependsOnGroups 与之同理。

4）参数化测试

在 TestNG 中，可以使用@Parameters 或@DataProvider 将参数传递给@Test 方法。我们以@DataProvider 为例介绍其使用方法，@Parameters 与之类似，读者可以举一反三。

@DataProvider 的实例代码如下：

```
@DataProvider(name = "providPaireNumbers")
public Object[][] provideDataByDataProvider () {

   return new Object[][] { { 11, 22 }, { 111,222 }, { 1111, 2222 } };
}

@Test(dataProvider = " providPaireNumbers ")
public void test(int number, int result) {
   Assert.assertEquals(number * 2, result);
}
```

上述代码中，provideDataByDataProvider 定义了名为 providPaireNumbers 的 DataProvider。在 test 方法中，使用 providPaireNumbers 进行了三次乘法运算，来验证结果的准确性。

附录B

Mockito

Mockito 是什么

Mockito 是一个 Mock 框架，它能让我们用简单明了的 API 做测试。Mockito 具有简单易学、可读性强、验证语法简捷的特点，非常容易上手。

为什么需要 Mock

在研发过程中，很多场景都会用到 Mock。如测试驱动的开发（TDD）一般会先写单元测试代码，再写具体的实现代码。

而在常规开发过程中，程序员对自己负责的业务代码编写单元测试代码也是提高代码质量、减少 bug 的方法之一。在写单元测试代码时，往往会遇到要测试的类有很多依赖的情形，这些依赖的类/对象/资源又有别的依赖，从而形成一个大的依赖树，如图 B-1 所示。

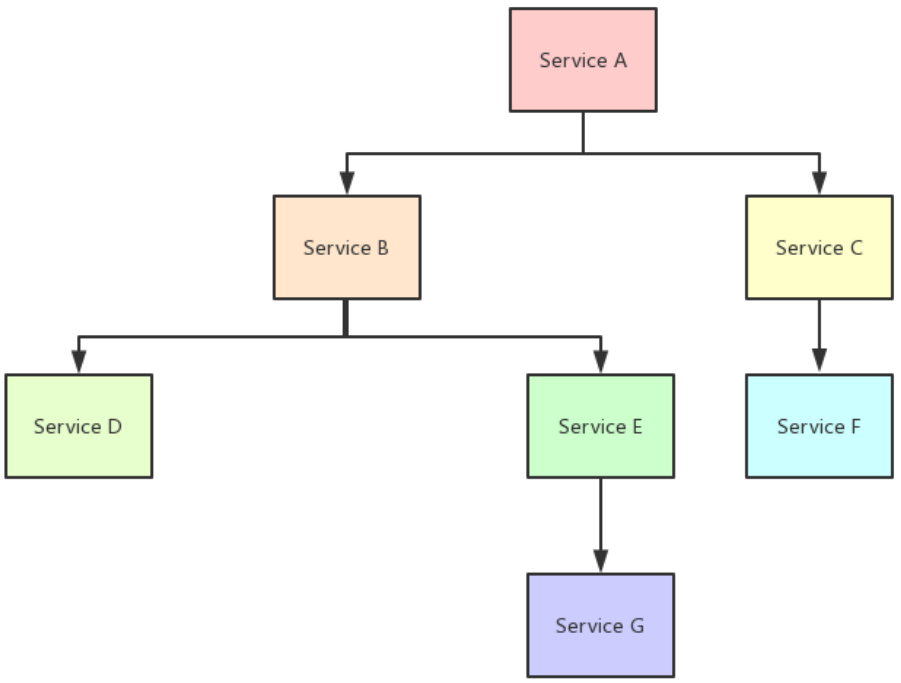


图 B-1　依赖树

Service A 依赖 Service B 和 Service C，而 Service B 依赖 Service D 和 Service E，Service C 依赖于 Service F，Service E 依赖于 Service G。

在这种情况下，要想在单元测试代码中完整地构建这样的依赖，是一件很困难的事情。而有了 Mock 方法后，测试这类依赖就简单多了。

应用 Mock 技术后，图 B-1 所示的依赖树会变成如图 B-2 所示的 Mock 依赖树。

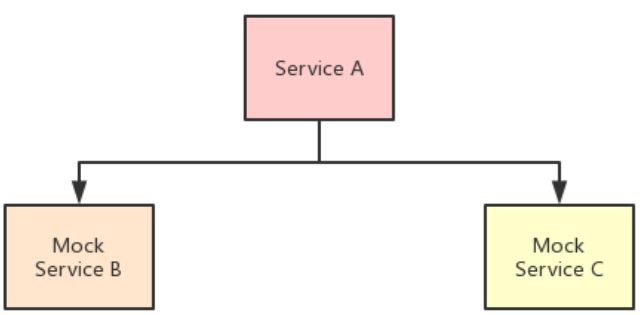


图 B-2　Mock 依赖树

Service A 仅仅依赖 Mock Service B 和 Mock Service C，也就是 Service B 和 C 对 Service A 而言变成了黑盒，至于 Mock Service B 和 Mock Service C 内部如何实现，Service A 并不关心。

Mockito 怎么玩

首先在 PoM 文件中引入 Mockito 的依赖：

```
<dependency>
     <groupId>org.mockito</groupId>
     <artifactId>mockito-all</artifactId>
     <version>1.9.5</version>
     <scope>test</scope>
</dependency>
```

下面结合一些常用场景来介绍 Mockito 的使用。

1）验证预期行为的发生

如验证 List 添加内容的行为。

我们先创建一个 Mock List 对象，可以用如下两种方式：

```
List mockList = mock(List.class);
```

或

```
@Mock
private List mockList;
```

下面模拟 List 添加内容的行为：

```
mockList.add(1);
```

用 verify 来验证行为的发生：

```
verify(mockList).add(1);
```

2）模拟设定结果的出现

比如模拟迭代器预设结果的出现。

我们先建立迭代器的 Mock 对象，如下所示：

```
    @Mock
    private Iterator iteratorMock;
```

然后对 iteratorMock 的调用结果作出设定，当 iteratorMock 调用 next()时，第一次返回 hello，第 *n* 次都返回 world，代码如下所示：

```
    when(iteratorMock.next()).thenReturn("hello").thenReturn("world");
```

使用 Mock 对象，获取设定结果，代码如下：

```
    String result = iteratorMock.next() + " " + iteratorMock.next() + " "
+ iteratorMock.next();
```

使用断言验证返回的结果 result 是否符合预期：

```
    Assert.assertEquals("hello world world", result);
```

3）参数匹配符合设定时返回设定结果

比如设定方法 method 的入参和出参分别如下：

```
    private int method(String str){
    //中间省略业务逻辑代码
    }
```

我们可以为对象 obj 的 method 的调用情况作出设定，如传参 str= "123" 时，返回 1，否则设定代码如下：

```
    when(obj.method("123")).thenReturn(1);
```

下面调用 method 方法并用断言验证返回结果，代码如下：

```
    Assert.assertEquals(1, obj.method("123"));
```

附录C

CouchDB 的安装

下载 CouchDB

我们可以从 CouchDB 官方网站获得安装包。

进入 CouchDB 官方网站，如图 C-1 所示。

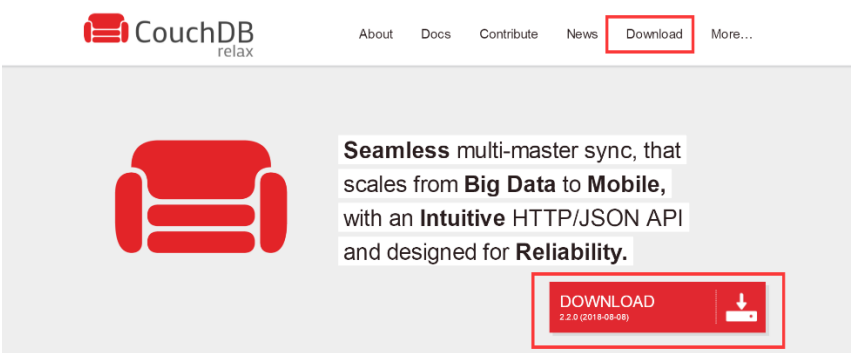


图 C-1　CouchDB 官方网站

单击导航栏的 Download 选项或单击 DOWNLOAD 按钮均可以切换到下载页面，下载页面如图 C-2 所示，读者可以根据自己安装 CouchDB 的系统环境来选择对应的安装包。

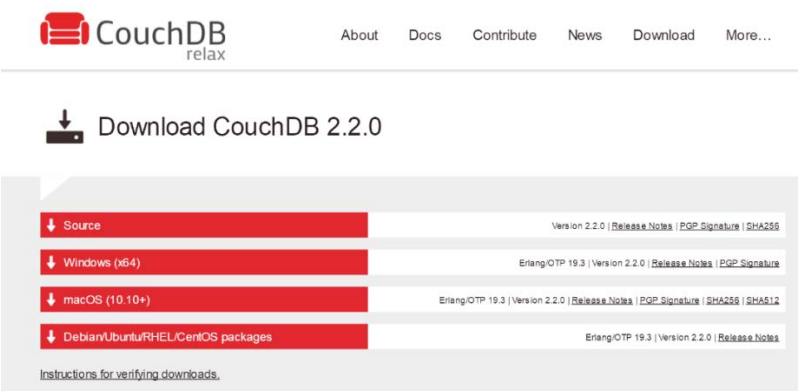


图 C-2　CouchDB 官方下载网站

安装 CouchDB

如果选择 Windows(x64)版本在 Windows 环境下安装，那么下载包之后，执行 exe 文件完成安装。安装完成后，打开本地浏览器，访问链接 http://127.0.0.1:5984/，会看到如下信息：

```
{
    "couchdb": "Welcome",
    "version": "2.2.0",
    "git_sha": "2a16ec4",
    "features": ["pluggable-storage-engines", "scheduler"],
    "vendor": {
        "name": "The Apache Software Foundation"
    }
}
```

这样 CouchDB 就安装好了。

还可以通过链接 http://127.0.0.1:5984/_utils/与 CouchDB Web 界面进行交互，打开链接后可看到如图 C-3 所示的页面。

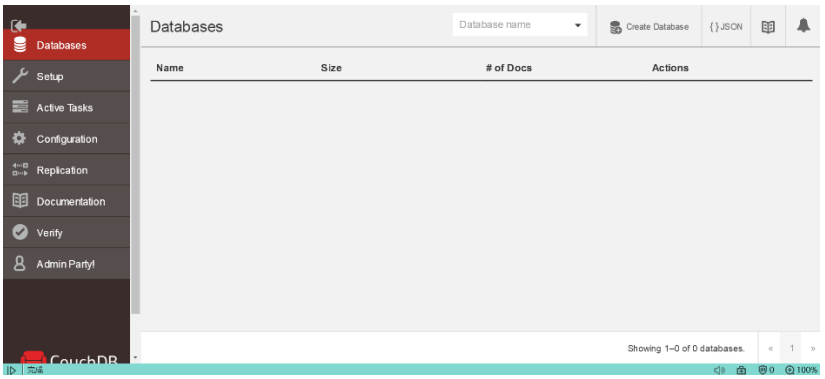


图 C-3　CouchDB Web 界面

在该页面可以进行各类配置，如管理员配置、集群或单节点参数配置等。

在 Linux 中可以通过命令 sudo apt install couchdb 完成 CouchDB 的安装，安装完成后，用 curl 命令访问 http://localhost:5984/，同样会得到和 Windows 下一样的返回内容。

后记

从约两年前笔者接触区块链伊始，笔者就一直在所做业务中尝试找到一个非区块链技术不可的场景来试手。可惜，一直未找到这样的场景。但从国内各个涉足区块链的公司所做的业务来看，也并非是非区块链技术不可的场景。同时，在新闻媒体中，我们不断看到各科技公司、互联网巨头、金融大鳄们将大把真金白银和人才投入到区块链的研发中。这是为什么呢？

笔者并不能完全揣测出他们的用意，但有一点可以看明白，目前科技公司、互联网巨头、金融大鳄们的精英们都认为区块链将会像互联网技术一样成为未来的普式技术。因此，这些机构都在追赶甚至引领着区块链技术风潮。

因此，笔者也转变了观念，开始追随巨人的脚步，着手展开区块链技术研发实践探索。在实践中，以培养具备区块链技术研发能力的种子团队和具备区块链产品设计思维的产品团队为主，在完成区块链项目研发的同时，注重自研区块链课程体系建设，不断打磨团队在区块链底层研发和开源区块链项目中的研究与开发能力。日后，一旦区块链应用前景天下大白，我们就可以迅速跟进趋势、扩建团队，快速形成有效战斗力。

当然，区块链技术日新月异，本书中所介绍的联盟链系统技术实践并不一定是最新的方式，建议读者要与时俱进，更新知识储备。同时，联盟链中的数据安全问题、系统安全、数据监控等内容十分重要，它们也是区块链底层研发的核心技术之一，本书虽未提及如何实操，但读者实践时务必将这些内容考虑在系统设计之中。

区块链是未来的技术，虽然我们现在还在步履蹒跚地前行，但跟上技术发展的列车，现在为时不晚。学习区块链原理之余，勿忘实践中总结真知，“纸上得来终觉浅，绝知此事要躬行”。加油！