xuperchain-doc Documentation

xuper

Mar 10, 2020

XuperChain 介绍:

CHAPTER 1

简介

XuperChain 是超级链体系下的第一个开源项目，是构建超级联盟网络的底层方案。

其主要特点是高性能，通过原创的 XuperModel 模型，真正实现了智能合约的并行执行和验证，通过自研的 WASM 虚拟机，做到了指令集级别的极致优化。

在架构方面，其可插拔、插件化的设计使得用户可以方便选择适合自己业务场景的解决方案，通过独有的 XuperBridge 技术，可插拔多语言虚拟机，从而支持丰富的合约开发语言。

在网络能力方面，XuperChain 具备全球化部署能力，节点通信基于加密的 P2P 网络，支持广域网超大规模节点，且底层账本支持分叉管理，自动收敛一致性，TDPOS 算法确保了大规模节点下的快速共识。在账号安全方面，XuperChain 内置了多私钥保护的账号体系，支持权重累计、集合运算等灵活的策略。

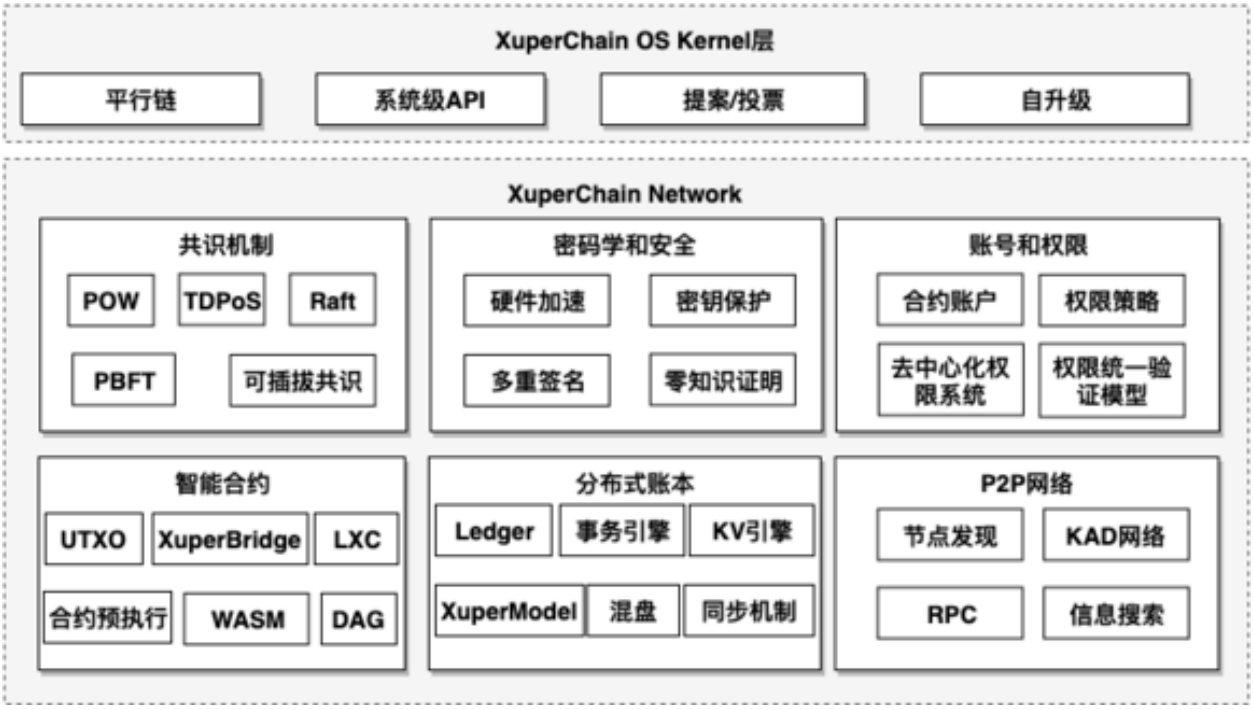


Fig. 1: XuperChain 架构

CHAPTER 2

模块

模块	特性
存储	XuperChain 的底层存储基于 KV 数据库，存储的数据包括区块数据、交易数据、账号余额、DPOS 投票数据、合约账号数据、智能合约数据等，上链的数据全部持久化到底层存储。不同的链，存储独立。底层存储支持可插拔，从而可以满足不同的业务场景
网络	负责交易数据的网络传播和广播、节点发现和维护。以 P2P 通信为基础，实现全分布式结构化拓扑网络结构，数据传输全程加密。局域网穿透技术采用 NAT 方案，同一条流保持长连接且复用。多条链复用同一个 p2p 网络
共识	共识模块用于解决交易上链顺序问题，过滤无效交易并达成全网一致。XuperChain 实现了更加高效的 DPOS 共识算法。支持可插拔，从而可以支持不同的业务场景
密码学	用于构造和验证区块、交易的完整性，采用非对称加密算法生成公私钥、地址。匿名性较好。支持可插拔，从而可以支持不同的业务场景
智能合约	自研并实现了一套智能合约虚拟机 XVM，支持丰富的开发语言，智能合约之间并发执行，支持执行消耗资源，避免恶意攻击
提案	一种解决系统升级问题的机制。比如修改区块大小，升级共识算法。提案整个过程涉及到发起提案、参与投票、投票生效三个阶段
账号与权限	为了满足合约调用的权限控制，保证 XuperChain 网络的健康运转，自研并实现了一套基于账号的去中心化的合约权限系统。支持权重累计、集合运算等灵活的策略，可以满足不同的业务场景

CHAPTER 3

核心数据结构

3.1 背景

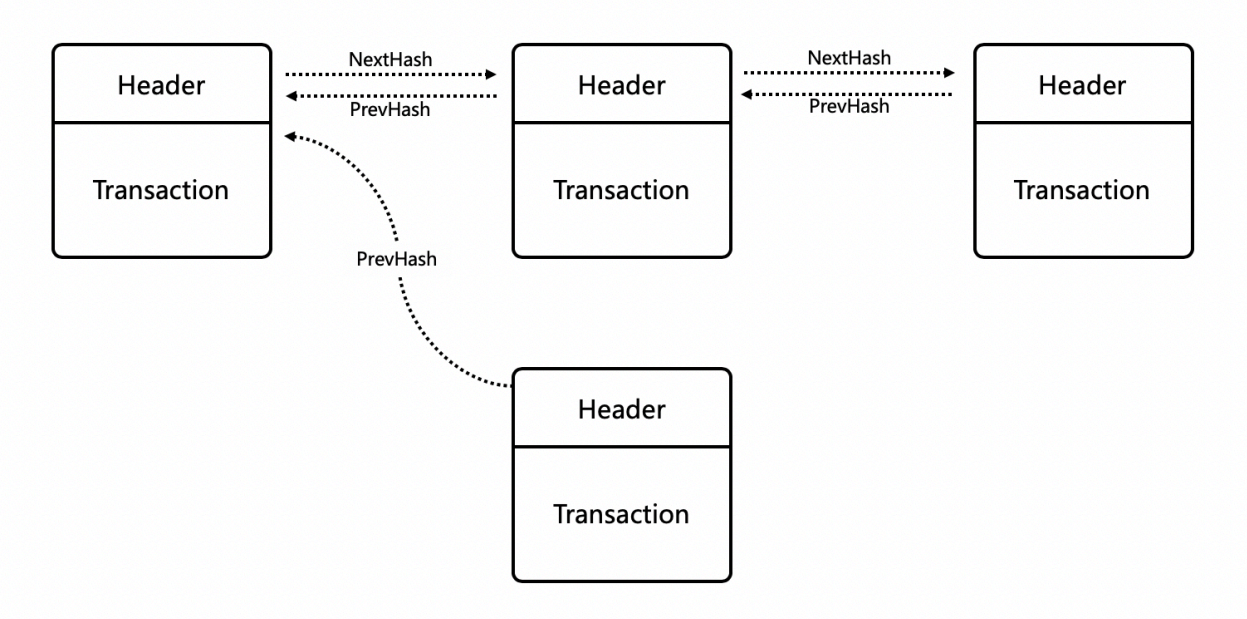
众所周知，程序 = 数据结构 + 算法，了解一个程序的数据结构有助于掌握一个程序的关键设计。本文从背景、功能以及各个字段的用意来剖析 XuperChain 底层核心数据结构，从而方便 XuperChain 开发者以及用户更深入地了解 XuperChain 底层框架的核心数据结构的设计缘由，有助于提高 XuperChain 开发者更高效的开发，有助于 XuperChain 用户更好的使用 XuperChain 来服务自己的业务。

3.2 核心数据结构

涉及到的核心数据结构包括：区块、交易、UTXO、读写集。

3.2.1 区块

- 背景：所谓区块链，简单来说就是不同的区块以 DAG 方式链接起来形成的链。因此，区块是区块链的基本单元。
- 功能：区块是区块链的基本单元，通常为了提高区块链网络的吞吐，矿工会在一个区块中打包若干个交易。一个区块通常由区块头以及区块体组成。



- 代码：区块的 Proto 如下

```
message InternalBlock {
    // block version
    // 区块版本
    int32 version = 1;
    // Random number used to avoid replay attacks
    // 随机数，用来避免重放攻击
    int32 nonce = 2;
    // blockid generate the hash sign of the block used by sha256
    // 区块的唯一标识
    bytes blockid = 3;
    // pre_hash is the parent blockid of the block
    // 区块的前置依赖区块 ID
    bytes pre_hash = 4;
    // The miner id
    // 矿工 ID
    bytes proposer = 5;
    // 矿工对区块的签名
    // The sign which miner signed: blockid + nonce + timestamp
    bytes sign = 6;
    // The pk of the miner
    // 矿工公钥
    bytes pubkey = 7;
    // The Merkle Tree root
```

(continues on next page)

(continued from previous page)

```
    // 默克尔树树根
    bytes merkle_root = 8;
    // The height of the blockchain
    // 区块所在高度
    int64 height = 9;
    // Timestamp of the block
    // 打包区块的时间戳
    int64 timestamp = 10;
    // Transactions of the block, only txid stored on kv, the detail information
    // stored in another table
    // 交易内容
    repeated Transaction transactions = 11;
    // The transaction count of the block
    // 区块中包含的交易数量
    int32 tx_count = 12;
    // 所有交易 hash 的 merkle tree
    repeated bytes merkle_tree = 13;
    // 采用 DPOS 共识算法时，当前是第几轮
    int64 curTerm = 16;
    int64 curBlockNum = 17;
    // 区块中执行失败的交易以及对应的失败原因
    map<string, string> failed_txs = 18; // txid -> failed reason
    // 采用 POW 共识算法时，对应的挖矿难度值
    int32 targetBits = 19;
    // 下面的属性会动态变化
    // If the block is on the trunk
    // 该区块是否在主干上
    bool in_trunk = 14;
    // Next next block which on trunk
    // 当前区块的后继区块 ID
    bytes next_hash = 15;
}
```

3.2.2 交易

- 背景：区块链网络中的每个节点都是一个状态机，为了给每个节点传递状态，系统引入了交易，作为区块链网络状态更改的最小操作单元。
- 功能：通常表现为普通转账以及智能合约调用。
- 代码：交易的 Proto 如下

```
message Transaction {
    // txid is the id of this transaction
    // 交易的唯一标识
    bytes txid = 1;
    // the blockid the transaction belong to
    // 交易被打包在哪个区块中
    bytes blockid = 2;
    // Transaction input list
    // UTXO 来源
    repeated TxInput tx_inputs = 3;
    // Transaction output list
    // UTXO 去处
    repeated TxOutput tx_outputs = 4;
    // Transaction description or system contract
    // 交易内容描述或系统合约
    bytes desc = 6;
    // Mining rewards
    // 矿工奖励
    bool coinbase = 7;
    // Random number used to avoid replay attacks
    // 随机数
    string nonce = 8;
    // Timestamp to launch the transaction
    // 发起交易的时间戳
    int64 timestamp = 9;
    // tx format version; tx 格式版本号
    int32 version = 10;
    // auto generated tx
    // 该交易是否属于系统自动生成的交易
    bool autogen = 11;
    // 读写集中的读集
    repeated TxInputExt tx_inputs_ext = 23;
    // 读写集中的写集
    repeated TxOutputExt tx_outputs_ext = 24;
    // 该交易包含的合约调用请求
    repeated InvokeRequest contract_requests = 25;
    // 权限系统新增字段
    // 交易发起者，可以是一个 Address 或者一个 Account
    string initiator = 26;
    // 交易发起需要被收集签名的 AddressURL 集合信息，包括用于 utxo 转账和用于合约调用
```

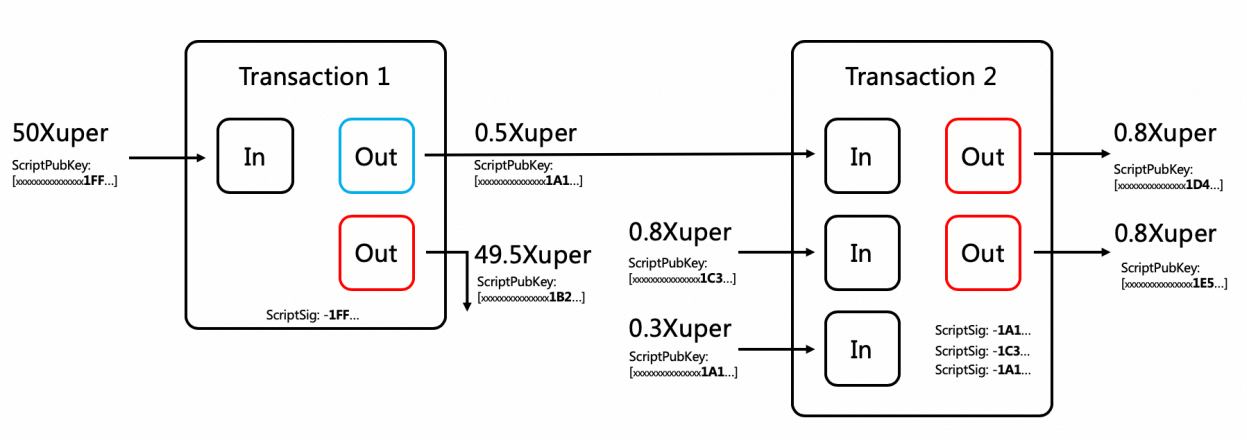
(continues on next page)

(continued from previous page)

```
    repeated string auth_require = 27;
    // 交易发起者对交易元数据签名，签名的内容包括 auth_require 字段
    repeated SignatureInfo initiator_signs = 28;
    // 收集到的签名
    repeated SignatureInfo auth_require_signs = 29;
    // 节点收到 tx 的时间戳，不参与签名
    int64 received_timestamp = 30;
    // 统一签名（支持多重签名/环签名等，与 initiator_signs/auth_require_signs 不同时使用）
    XuperSignature xuper_sign = 31;
}
```

3.2.3 UTXO

- 背景：区块链中比较常见的两种操作，包括普通转账以及合约调用，这两种操作都涉及到了数据状态的引用以及更新。为了描述普通转账涉及到的数据状态的引用以及更新，引入了 UTXO(Unspent Transaction Output)。
- 功能：一种记账方式，用来描述普通转账时涉及到的数据状态的引用以及更新。通常由转账来源数据 (UtxoInput) 以及转账去处数据 (UtxoOutput) 组成。



- 代码：UTXO 的 Proto 如下

```
message Utxo {
    // 转账数量
    bytes amount = 1;
    // 转给谁
    bytes toAddr = 2;
    // 转给谁的公钥
    bytes toPubkey = 3;
```

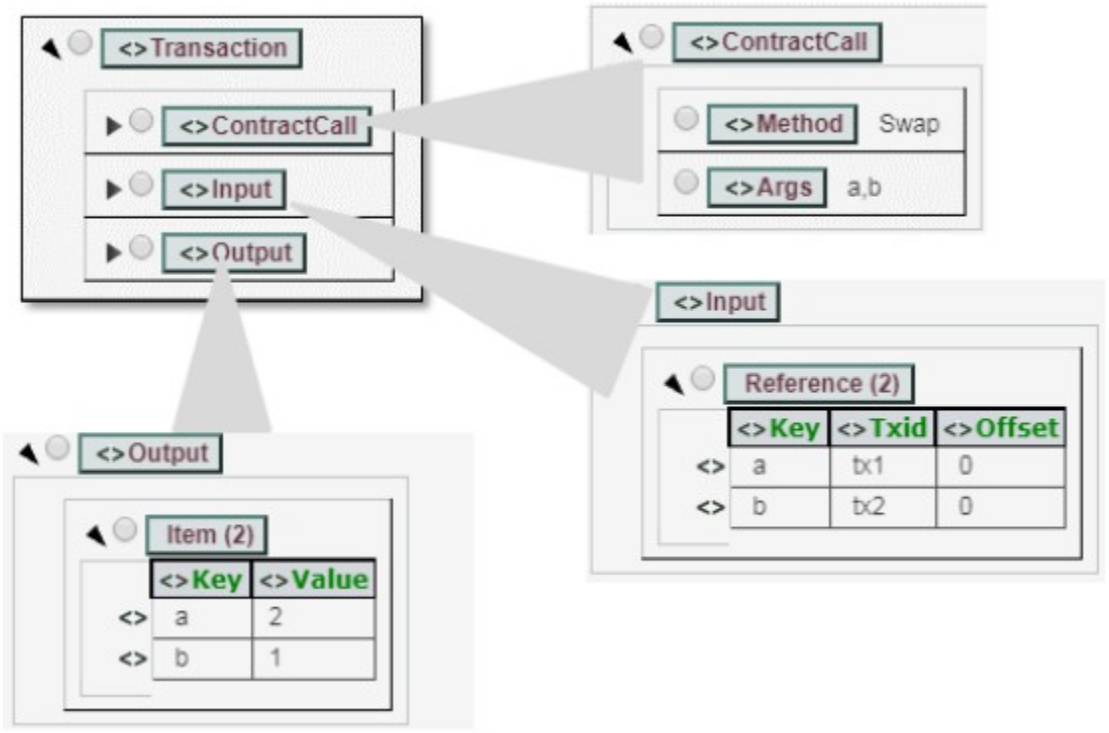
(continues on next page)

(continued from previous page)

```
    // 该 Utxo 属于哪一个交易
    bytes refTxid = 4;
    // 该 Utxo 数据哪一个交易的哪一个 offset
    int32 refOffset = 5;
}
// UtxoInput query info to query utxos
// UTXO 的转账来源
message UtxoInput {
    Header header = 1;
    // which bcname to select
    // UTXO 来源属于哪一条链
    string bcname = 2;
    // address to select
    // UTXO 来源属于哪个 address
    string address = 3;
    // publickey of the address
    // UTXO 来源对应的公钥
    string publickey = 4;
    // totalNeed refer the total need utxos to select
    // 需要的 UTXO 总额
    string totalNeed = 5;
    // userSign of input
    // UTXO 来源的签名
    bytes userSign = 7;
    // need lock
    // 该 UTXO 是否需要锁定 (内存级别锁定)
    bool needLock = 8;
}
// UtxoOutput query results
// UTXO 的转账去处
message UtxoOutput {
    Header header = 1;
    // utxo list
    // UTXO 去处
    repeated Utxo utxoList = 2;
    // total selected amount
    // UTXO 去处总额
    string totalSelected = 3;
}
```

3.2.4 读写集

- 背景：区块链中比较常见的两种操作，包括普通转账以及合约调用，这两种操作都涉及到了数据状态的引用以及更新。为了描述合约调用涉及到的数据状态的引用以及更新，引入了读写集。
- 功能：一种用来描述合约调用时涉及到的数据状态的引用以及更新的技术。通常由读集 (TxInputExt) 以及写集 (TxOutputExt) 组成。



- 代码：读写集的 Proto 如下

```
// 扩展输入
message TxInputExt {
    // 读集属于哪一个 bucket
    string bucket = 1;
    // 读集对应的 key
    bytes key = 2;
    // 读集属于哪一个 txid
    bytes ref_txid = 3;
    // 读集属于哪一个 txid 的哪一个 offset
    int32 ref_offset = 4;
}
// 扩展输出
message TxOutputExt {
    // 写集属于哪一个 bucket
```

(continues on next page)

(continued from previous page)

```
    string bucket = 1;
    // 写集对应的 key
    bytes key = 2;
    // 写集对应的 value
    bytes value = 3;
}
```

CHAPTER 4

智能合约

自研并实现了一套智能合约虚拟机 XVM。特点如下：

1. 合约状态数据与合约代码运行环境分离，从而能够支持多语言虚拟机且各种合约虚拟机只需要做纯粹的无状态合约代码执行；
2. 支持执行消耗资源，避免恶意攻击；
3. 支持丰富的智能合约开发语言，比如 go，Solitidy，C/C++，Java 等；
4. 利用读写集确保普通合约调用支持并发执行，充分利用计算机多核特性；

CHAPTER 5

权限系统

实现一个去中心化，区块链内置的合约账号权限系统。特点如下：

1. 支持多种权限模型，比如背书数、背书率、AK 集合、CA 鉴权、社区治理等；
2. 支持完善的账号权限管理，比如账号的创建、添加和删除 AK、设置 AK 权重、权限模型；
3. 支持设置合约调用权限，添加和删除 AK、设置 AK 权重、权限模型；

CHAPTER 6

隐私和保密

XuperChain 支持多种隐私保护和保密机制，包括但不限于：

1. 数据在 p2p 网络中采用 ECDH 加密传输，保障区块链数据的安全性；
2. 通过助记词技术，在用户私钥丢失的情况下可以恢复；
3. 多私钥保护的账号体系；
4. 基于椭圆曲线算法的公私钥加密和签名体系；

CHAPTER 7

性能

交易处理速度：达到 9 万 TPS

1. 默认采用 DPOS 作为共识算法；
2. 交易处理充分利用计算机多核，支持并发执行；
3. 智能合约通过读写集技术能够支持并发执行；

CHAPTER 8

总结

XuperChain 是百度自研的一套区块链解决方案，采用经典的 UTXO 记账模式，并且支持丰富的智能合约开发语言，交易处理支持并发执行，拥有完善的账号与权限体系，采用 DPOS 作为共识算法，交易处理速度可达到 9 万 TPS。

本章节将指导您获取 XuperChain 的代码并部署一个基础的可用环境，还会展示一些基本操作

CHAPTER 9

XuperChain 环境部署

9.1 准备环境

XuperChain 主要由 Golang 开发，需要首先准备编译运行的环境

- **安装 go 语言编译环境，版本为 1.11 或更高**
  - 下载地址：golang
- **安装 git**
  - 下载地址：git

9.2 编译 XuperChain

- 使用 git 下载源码到本地
  - git clone https://github.com/xuperchain/xuperchain.git
- 执行命令

```
cd src/github.com/xuperchain/xuperchain
make
```

- 在 output 目录得到产出 xchain 和 xchain-cli

Note: 可能需要配置 go 语言环境变量（$GOROOT, $PATH）

GOPATH 问题报错（go1.11 版本之后无需关注）

- 在 1.11 版本之前需要配置。配置成以下形式：
- export GOPATH=xxx/github.com/xuperchain/xuperchain

GCC 版本需要升级到 4 或 5 以上

CHAPTER 10

XuperChain 基本操作

在 output 下，主要目录有 data, logs, conf, plugins 等, 二进制文件有 xchain，xchain-cli

各目录的功能如下表：

目录名	功能
output/	节点根目录
conf	xchain.yaml: xchain 服务的配置信息（注意端口冲突）plugins.conf: 插件的配置信息
data	数据的存放目录，创世块信息，以及共识和合约的样例
··· blockchain	账本目录
··· keys	此节点的地址，具有全局唯一性
··· netkeys	此节点的网络标识 ID，具有全局唯一性
··· config	包括创始的共识，初始的资源数，矿工奖励机制等
logs	程序日志目录
plugins	so 扩展的存放目录
xchain	xchain 服务的二进制文件
xchain-cli	xchain 客户端工具
wasm2c	wasm 工具（智能合约会用到）

10.1 部署 xchain 服务

10.1.1 创建链

在启动 xchain 服务之前，我们首先需要创建一条链（创世区块），xchain 客户端工具提供了此功能

```
# 创建 xuper 链
./xchain-cli createChain
```

这样我们就使用 config/xuper.json 中的配置创建了一条链（此时 data/blockchain 中会生成 xuper 目录，里面即是我们创建的链的账本等文件）

10.1.2 启动服务

启动服务命令十分简单，还可以配合多种参数使用，详见命令行的 -h 输出

```
# 启动服务节点
nohup ./xchain &
```

10.1.3 确认服务状态

按照默认配置，xchain 服务会监听 37101 端口，可以使用如下命令查看 xchain 服务的运行状态

```
# check 服务运行状况
./xchain-cli status -H 127.0.0.1:37101
```

10.2 基本功能的使用

10.2.1 创建新账号

xchain 中，账号分为普通账号和 “合约账号”，这里先介绍普通账号的创建，命令如下

```
# 创建普通用户，包含地址，公钥，私钥
./xchain-cli account newkeys --output data/bob
# 在 bob 目录下会看到文件 address，publickey，privatekey 生成
```

10.2.2 查询资源余额

对于普通账号，可使用如下命令查询账号资源余额，其中 -H 参数为 xchain 服务的地址

```
./xchain-cli account balance --keys data/keys -H 127.0.0.1:37101
```

10.2.3 转账

转账操作需要提供源账号的私钥目录，也就类似“1.2.4.1”中生成的目录，这里注意到并不需要提供目标账号的任何密钥，只需要提供地址即可

```
# --keys 从此地址 转给 --to 地址 --amount 钱
./xchain-cli transfer --to czojZcZ6cHSiDVJ4jFoZMB1PjKnfUiuFQ --amount 10 --keys data/
↪keys/ -H 127.0.0.1:37101
```

命令执行的返回是转账操作的交易 id（txid）

10.2.4 查询交易信息

通过以下命令可以查询交易的信息，包括交易状态、交易的源和目标账号、交易的金额、所在的区块（如果已上链）等内容

```
# 可查询上一步生成的 txid 的交易信息
./xchain-cli tx query cbbda2606837c950160e99480049e2aec3e60689a280b68a2d253fdd8a6ce931 -
↪H 127.0.0.1:37101
```

10.2.5 查询 block 信息

通过 blockid 可以查询区块的相关信息，包括区块内打包的交易、所在链的高度、前驱/后继区块的 id 等内容

```
# 可查询上一步交易所在的 block id 信息
./xchain-cli block 0354240c8335e10d8b48d76c0584e29ab604cfdb7b421d973f01a2a49bb67fee -H␣
↪127.0.0.1:37101
```

10.2.6 发起多重签名交易

对于需要多个账号签名才可以生效的交易，需要先发起多重签名交易，收集需要的签名，然后在发出

对需要收集签名的账号地址，需要事先维护在一个文件中（假定名为 addr_list），每个地址一行

```
YDYBchKWXpG7HSkHy4YoyzTJnd3hTFBgG
ZAmWoJViiNn5pKz32m2MVgmPnSpgLia7z
```

假设要发起一笔转账操作

```
# 从账号发起
./xchain-cli multisig gen --to czojZcZ6cHSiDVJ4jFoZMB1PjKnfUiuFQ --amount 100 -A addr_
↪list
# 从合约账号发起
./xchain-cli multisig gen --to czojZcZ6cHSiDVJ4jFoZMB1PjKnfUiuFQ --amount 100 -A addr_
↪list --from XC11111111111111@xuper
```

这样会生成一个 tx.out 文件，包含了需发起的交易内容

```
# 各方在签名之前可以 check 原始交易是否 ok，查看 visual.out
./xchain-cli multisig check --input tx.out --output visual.out
```

然后收集需要的签名

```
# 首先需要发起者自己的签名
./xchain-cli multisig sign --tx tx.out --output my.sign
# 假设 addr_list 中的地址对应的私钥存放在 alice、bob 中
./xchain-cli multisig sign --keys data/account/alice --tx tx.out --output alice.sign
./xchain-cli multisig sign --keys data/account/bob --tx tx.out --output bob.sign
```

最后将交易和收集好的签名发出

```
# send 后第一个参数是发起者的签名文件，第二个参数是需要收集的签名文件，均为逗号分割
./xchain-cli multisig send --tx tx.out my.sign alice.sign,bob.sign
```

CHAPTER 11

XuperModel

XuperChain 能够支持合约链内并行的很大的原因是由于其底层自研的 XuperModel 数据模型。

XuperModel 是一个带版本的存储模型，支持读写集生成。该模型是比特币 utxo 模型的一个演变。在比特币的 utxo 模型中，每个交易都需要在输入字段中引用早期交易的输出，以证明资金来源。同样，在 XuperModel 中，每个事务读取的数据需要引用上一个事务写入的数据。在 XuperModel 中，事务的输入表示在执行智能合约期间读取的数据源，即事务的输出来源。事务的输出表示事务写入状态数据库的数据，这些数据在未来事务执行智能合约时将被引用，如下图所示：

为了在运行时获取合约的读写集，在预执行每个合约时 XuperModel 为其提供智能缓存。该缓存对状态数据库是只读的，它可以为合约的预执行生成读写集和结果。验证合约时，验证节点根据事务内容初始化缓存实例。节点将再次执行一次合约，但此时合约只能从读集读取数据。同样，写入数据也会在写入集中生效。当验证完生成的写集和事务携带的写集一致时合约验证通过，将事务写入账本，cache 的原理如下所示，图中左边部分是合约预执行时的示意图，右边部分是合约验证时的示意图：

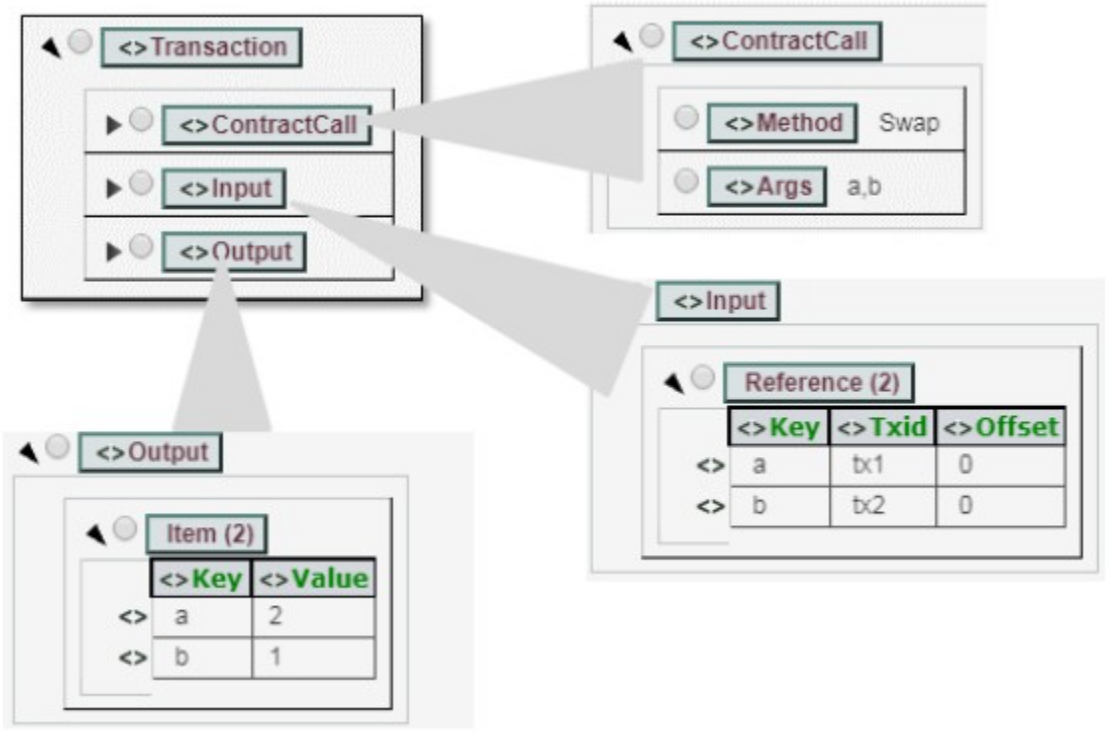


Fig. 1: XuperModel 事务

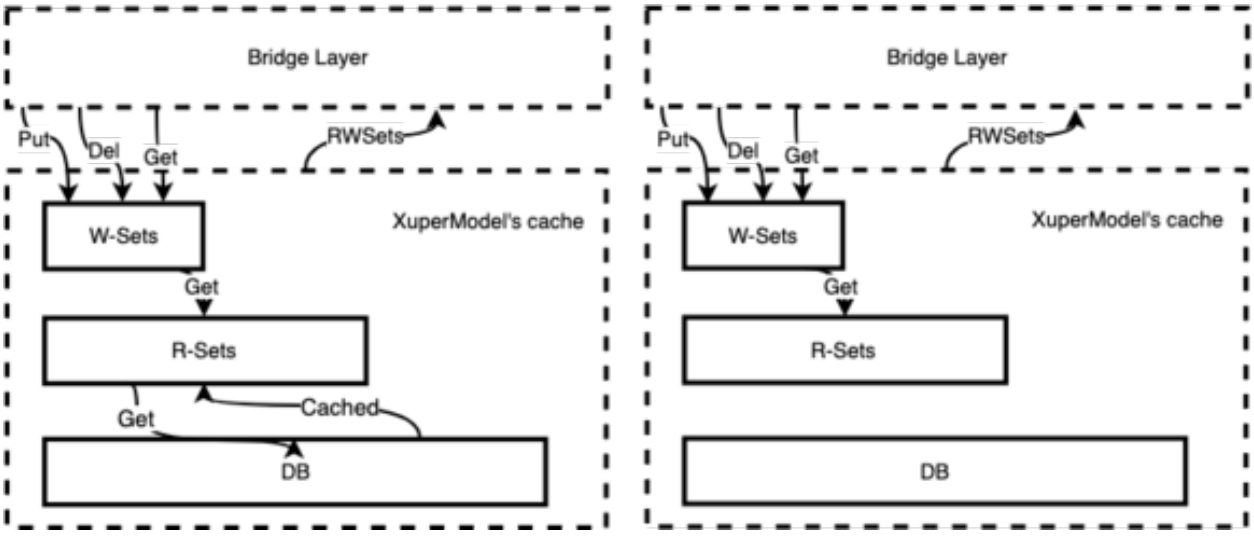


Fig. 2: XuperModel 合约验证

CHAPTER 12

XuperBridge

12.1 内核调用设计

XuperBridge 为所有合约提供统一的合约接口，从抽象方式上类似于 linux 内核对应于应用程序，内核代码是一份，应用程序可以用各种语言实现，比如 go,c。类比到合约上就是各种合约的功能，如 KV 访问，QueryBlock, QueryTx 等，这些请求都会通过跟 xchain 通信的方式来执行，这样在其上实现的各种合约虚拟机只需要做纯粹的无状态合约代码执行。

12.1.1 合约与 xchain 进程的双向通信

xchain 进程需要调用合约虚拟机来执行具体的合约代码，合约虚拟机也需要跟 xchain 进程通信来进行具体的系统调用，如 KV 获取等，这是一个双向通信的过程。

这种双向通信在不同虚拟机里面有不同的实现，

- 在 native 合约里面由于合约是跑在 docker 容器里面的独立进程，因此牵扯到跨进程通信，这里选用了 unix socket 作为跨进程通信的传输层，xchain 在启动合约进程的时候把 syscall 的 socket 地址以及合约进程的 socket 地址传递给合约进程，合约进程一方面监听在 unix socket 上等待 xchain 调用自己运行合约代码，另一方面通过 xchain 的 unix socket 创建一个指向 xchain syscall 服务的 grpc 客户端来进行系统调用。
- 在 WASM 虚拟机里面情况有所不同，WASM 虚拟机是以 library 的方式链接到 xchain 二进制里面，所以虚拟机和 xchain 在一个进程空间，通信是在 xchain 和 WASM 虚拟机之间进行的，这里牵扯到 xchain 的数据跟虚拟机里面数据的交换，在实现上是通过 WASM 自己的模块机制实现的，xchain 实

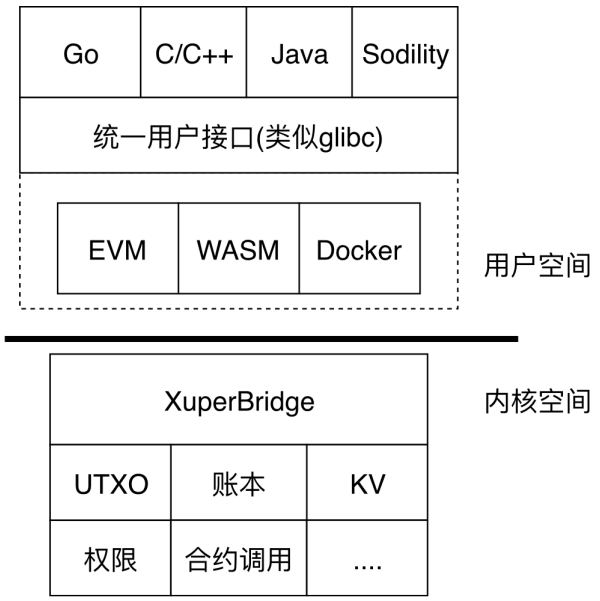


Fig. 1: XuperBridge

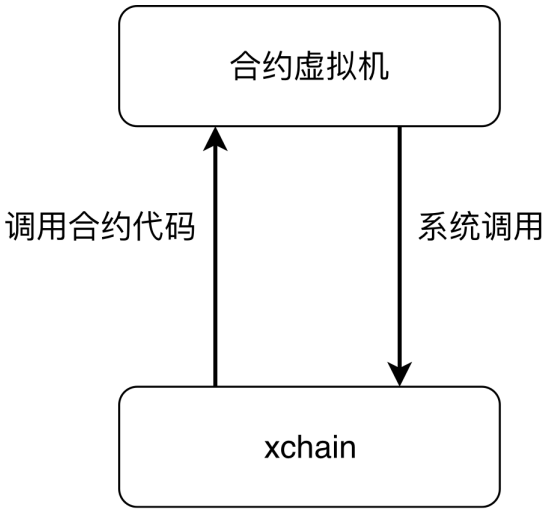


Fig. 2: 合约双向通信

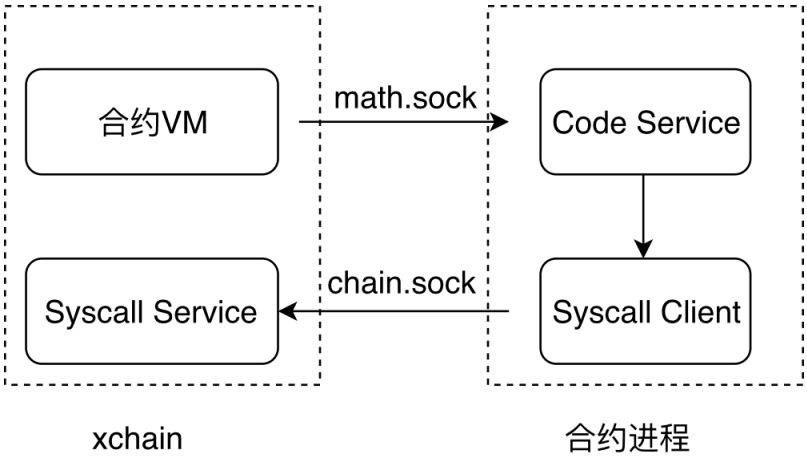


Fig. 3: 合约 socket

现了一个虚拟的 WASM 模块，合约代码执行到外部模块调用的时候就转到对应的 xchain 函数调用，由于 xchain 和合约代码的地址空间不一样，还是牵扯到序列化和反序列化的动作。

12.1.2 PB 接口

合约暴露的代码接口

```
service NativeCode {
    rpc Call(CallRequest) returns (CallResponse);
}
```

xchain 暴露的 syscall 接口

```
service Syscall {
    // KV service
    rpc PutObject(PutRequest) returns (PutResponse);
    rpc GetObject(GetRequest) returns (GetResponse);
    rpc DeleteObject(DeleteRequest) returns (DeleteResponse);
    rpc NewIterator(IteratorRequest) returns (IteratorResponse);

    // Chain service
    rpc QueryTx(QueryTxRequest) returns (QueryTxResponse);
    rpc QueryBlock(QueryBlockRequest) returns (QueryBlockResponse);
    rpc Transfer(TransferRequest) returns (TransferResponse);
}
```

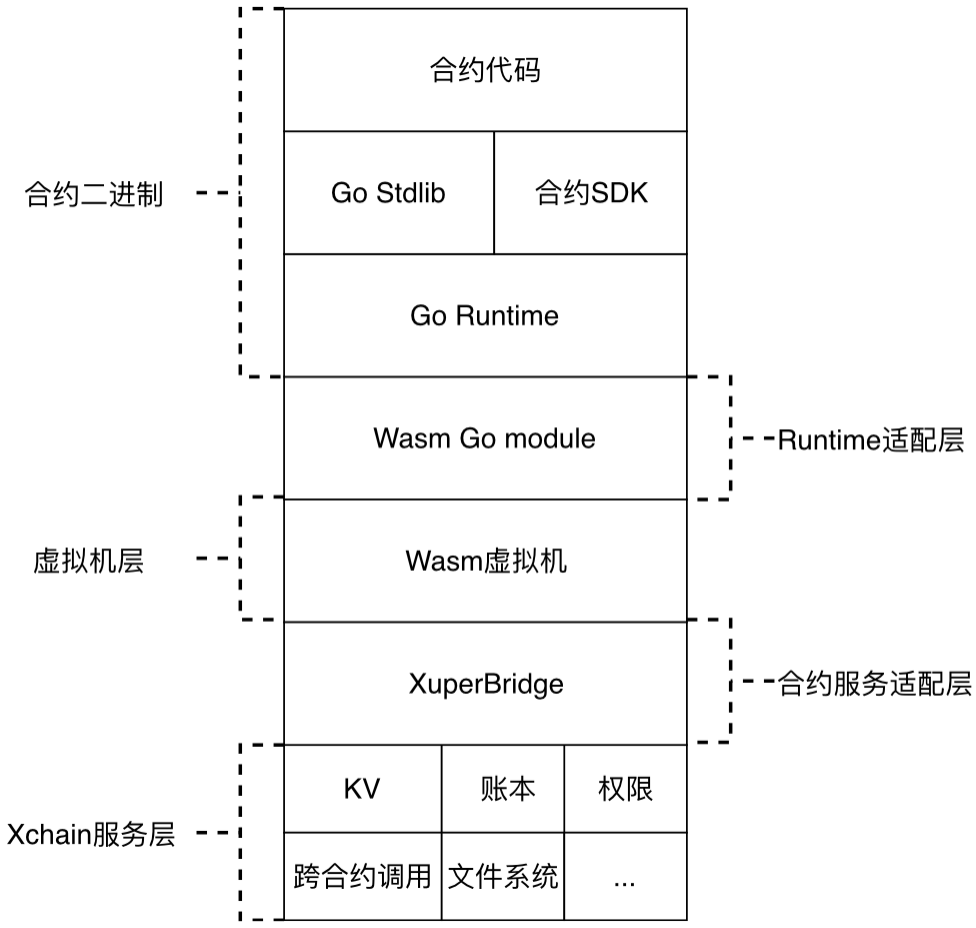

合约代码
Go Stdlib
合约SDK
Go Runtime
Wasm Go module
Wasm虚拟机
XuperBridge
KV
账本
权限
跨合约调用
文件系统
...
合约二进制
虚拟机层
Xchain服务层
Runtime适配层
合约服务适配层

Fig. 4: WASM 合约

12.2 KV 接口与读写集

合约每次执行的产出为一系列 KV 操作的读写集，读写集的概念详细见*XuperModel*。

KV 的接口：

- GetObject(key)
- PutObject(key, value)
- DeleteObject(key)
- NewIterator(start, limit)

各个接口对读写集的影响：

- Get 会生成一个读请求
- Put 会产生一个读加一个写
- Delete 会产生一个读加一个特殊的写（TODO）
- Iterator 会对迭代的 key 产生读

效果：

- 读请求不会读到最新的其他 tx 带来的变更
- 读请求会读到最新的自己的写请求（包括删除）的变更
- 写请求在提交前不会被其他合约读到
- 新写入的会被迭代器读到

12.3 合约上下文

每次合约运行都会有一个伴随合约执行的上下文 (context) 对象，context 里面保存了合约的 kv cache 对象，运行参数，输出结果等，context 用于隔离多个合约的执行，也便于合约的并发执行。

12.3.1 Context 的创建和销毁

context 在合约虚拟机每次执行合约的时候创建。每个 context 都有一个 context id，这个 id 由合约虚拟机维护，在 xchain 启动的时候置 0，每次创建一个 context 对象加 1，合约虚拟机保存了 context id 到 context 对象的映射。context id 会传递给合约虚拟机，在 Docker 里面即是合约进程，在之后的合约发起 KV 调用过程中需要带上这个 context id 来标识本次合约调用以找到对应的 context 对象。

context 的销毁时机比较重要，因为我们还需要从 context 对象里面获取合约执行过程中的 Response 以及读写集，因此有两种解决方案，一种是由调用合约的地方管理，这个是 xuper3 里面做的，一种是统一销毁，这个是目前的做法，在打包成块结束调用 Finalize 的时候统一销毁所有在这个块里面的合约 context 对象。

12.3.2 合约上下文的操作

- NewContext，创建一个 context，需要合约的参数等信息。
- Invoke，运行一个 context，这一步是执行合约的过程，合约执行的结果会存储在 context 里面。
- Release，销毁 context，context 持有的所有资源得到释放。

CHAPTER 13

XVM 虚拟机

13.1 背景

XVM 为合约提供一个稳定的沙盒运行环境，有如下目标:

- 隔离性，合约运行环境和 xchain 运行环境互不影响，合约的崩溃不影响 xchain。
- 确定性，合约可以访问链上资源，但不能访问宿主机资源，保证在确定的输入下有确定的输出
- 可停止性，设置资源 quota，合约对资源的使用超 quta 自动停止
- 可以统计合约的资源使用情况，如 CPU，内存等
- 运行速度尽量快。

13.2 WASM 简介

WASM 是 WebAssembly 的缩写，是一种运行在浏览器上的字节码，用于解决 js 在浏览器上的性能不足的问题。WASM 的指令跟机器码很相似，因此很多高级语言如 C，C++，Go，rust 等都可以编译成 WASM 字节码从而可以运行在浏览器上。很多性能相关的模块可以通过用 C/C++ 来编写，再编译成 WASM 来提高性能，如视频解码器，运行在网页的游戏引擎，React 的虚拟 Dom 渲染算法等。

WASM 本身只是一个指令集，并没有限定运行环境，因此只要实现相应的解释器，WASM 也可以运行在非浏览器环境。xchain 的 WASM 合约正是这样的应用场景，通过用 C++，go 等高级语言来编写智能合约，再编译成 WASM 字节码，最后由 XVM 虚拟机来运行。XVM 虚拟机在这里就提供了一个 WASM 的运行环境。

13.3 WASM 字节码编译加载流程

WASM 字节码的运行有两种方式，一种是解释执行，一种是编译成本地指令后再运行。前者针对每条指令挨个解释执行，后者通过把 WASM 指令映射到本地指令如 (x86) 来执行，解释执行有点是启动快，缺点是运行慢，编译执行由于有一个预先编译的过程因此启动速度比较慢，但运行速度很快。

XVM 选用的是编译执行模式。

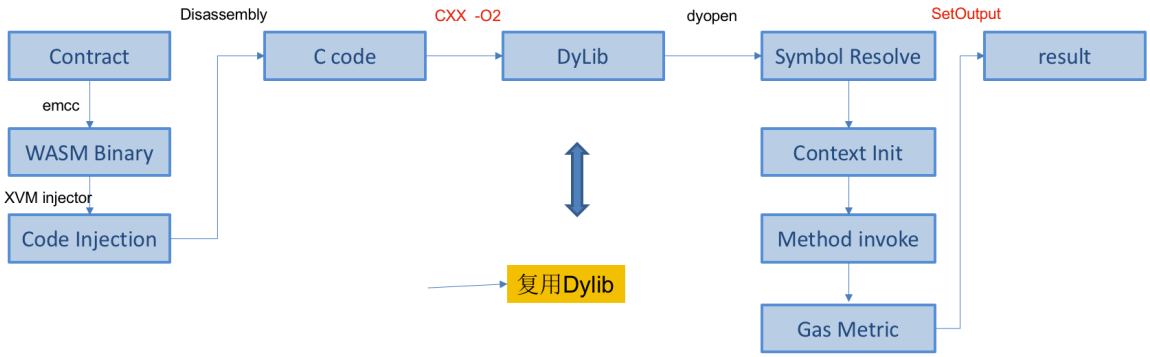


Fig. 1: XVM 编译加载流程

13.3.1 字节码编译

用户通过 c++ 编写智能合约，通过 emcc 编译器生成 wasm 字节码，xvm 加载字节码，生成加入了指令资源统计的代码以及一些运行时库符号查找的机制，最后编译成本地指令来运行。

c++ 合约代码

```
int add(int a, int b) {
        return a + b;
}
```

编译后的 WASM 文本表示

```
(module
(func $add (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add)
(export "_add" (func $add)))
```

XVM 编译 WASM 到 c，最后再生成动态链接库。

```
static u32 _add(wasm_rt_handle_t* h, u32 p0, u32 p1) {
    FUNC_PROLOGUE;
    u32 i0, i1;
    ADD_AND_CHECK_GAS(3);
    i0 = p0;
    i1 = p1;
    i0 += i1;
    FUNC_EPILOGUE;
    return i0;
}
/* export: '_add' */
u32 (*export__add)(wasm_rt_handle_t*, u32, u32);

static void init_exports(wasm_rt_handle_t* h) {
    /* export: '_add' */
    export__add = (&_add);
}
```

13.3.2 加载运行

在了解如何加载运行之前先看下如何使用 xvm 来发起对合约的调用，首先生成 Code 对象，Code 对象管理静态的指令代码以及合约所需要的符号解析器 Resolver。之后就可以通过实例化 Context 对象来发起一次合约调用，GasLimit 等参数就是在这里传入的。Code 和 Context 的关系类似 Docker 里面的镜像和容器的关系，一个是静态的，一个是动态的。

```
func run(modulePath string, method string, args []string) error {
    code, err := exec.NewCode(modulePath, emscripten.NewResolver())
    if err != nil {
            return err
    }
    defer code.Release()

    ctx, err := exec.NewContext(code, exec.DefaultContextConfig())
    if err != nil {
            return err
    }
    ret, err := ctx.Exec(method, []int64{int64(argc), int64(argv)})
    fmt.Println(ret)
    return err
}
```

转换后的 c 代码最终会编译成一个动态链接库来给 XVM 运行时来使用，在每个生成的动态链接库里面都有如下初始化函数。这个初始化函数会自动对 wasm 里面的各个模块进行初始化，包括全局变量、内存、table、外部符号解析等。

```
typedef struct {
    void* user_ctx;
    wasm_rt_gas_t gas;
    u32 g0;
    uint32_t call_stack_depth;
}wasm_rt_handle_t;


void* new_handle(void* user_ctx) {
    wasm_rt_handle_t* h = (*g_rt_ops.wasm_rt_malloc)(user_ctx, sizeof(wasm_rt_handle_t));
    (h->user_ctx) = user_ctx;
    init_globals(h);
    init_memory(h);
    init_table(h);
    return h;
}
```

13.4 语言运行环境

13.4.1 c++ 运行环境

c++ 因为没有 runtime，因此运行环境相对比较简单，只需要设置基础的堆栈分布以及一些系统函数还有 emscripten 的运行时函数即可。

c++ 合约的内存分布

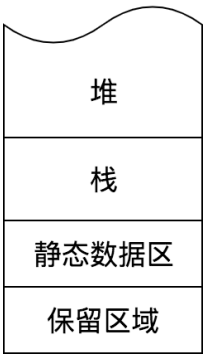


Fig. 2: c++ 合约的内存分布

普通调用如何在 xvm 解释

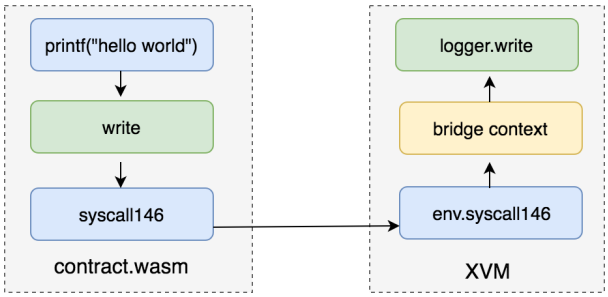


Fig. 3: xvm 符号解析

13.4.2 go 运行环境

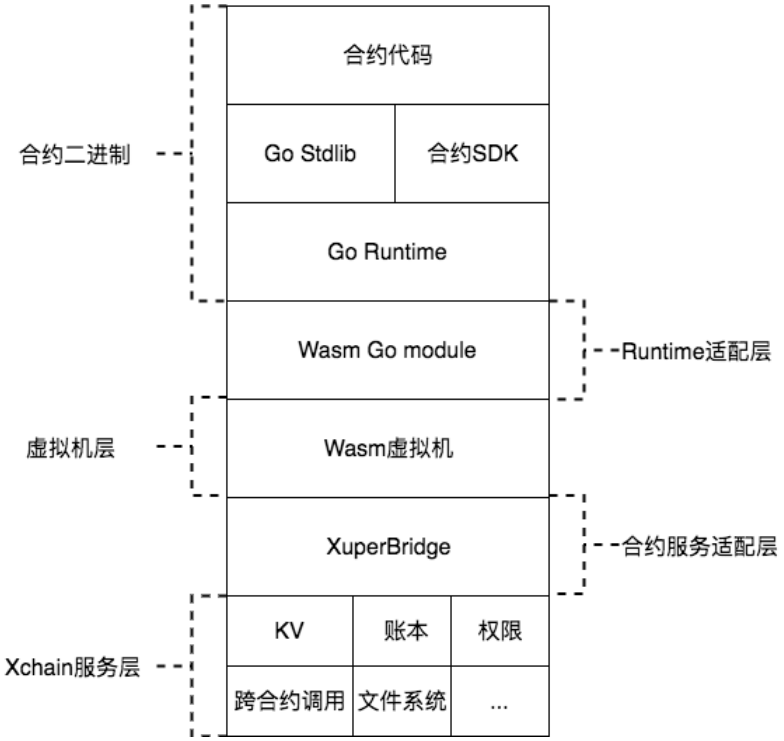


Fig. 4: go 合约运行时结构

13.5 XuperBridge 对接

XVM 跟 XuperBridge 对接主要靠两个函数

- call_method，这个函数向 Bridge 传递需要调用的方法和参数
- fetch_response，这个函数向 Bridge 获取上次调用的结果

```
extern "C" uint32_t call_method(const char* method, uint32_t method_len,
                            const char* request, uint32_t request_len);
extern "C" uint32_t fetch_response(char* response, uint32_t response_len);

static bool syscall_raw(const std::string& method, const std::string& request,
                        std::string* response) {
    uint32_t response_len;
    response_len = call_method(method.data(), uint32_t(method.size()),
                            request.data(), uint32_t(request.size()));
    if (response_len <= 0) {
        return true;
    }
    response->resize(response_len + 1, 0);
    uint32_t success;
    success = fetch_response(&(*response)[0u], response_len);
    return success == 1;
}
```

13.6 资源消耗统计

考虑到大部分指令都是顺序执行的，因此不需要在每个指令后面加上 gas 统计指令，只需要在 control block 最开头加上 gas 统计指令，所谓 control block 指的是 loop, if 等会引起跳转的指令。

c++ 代码

```
extern int get(void);
extern void print(int);

int main() {
        int i = get();
        int n = get();
        if (i < n) {
                i += 1;
                print(i);
        }
        print(n);
}
```

编译后生成的 wast 代码

```
(func (;2;) (type 1) (result i32)
    (local i32 i32)
    call 1
    local.tee 0
    call 1
    local.tee 1
    i32.lt_s
    if  ;; label = @1
        local.get 0
        i32.const 1
        i32.add
        call 0
    end
    local.get 1
    call 0
    i32.const 0)
```

生成的带统计指令的 c 代码

```
static u32 wasm__main(wasm_rt_handle_t* h) {
    u32 l0 = 0, l1 = 0;
    FUNC_PROLOGUE;
    u32 i0, i1;
    ADD_AND_CHECK_GAS(11);
    i0 = wasm_env__get(h);
    l0 = i0;
    i1 = wasm_env__get(h);
    l1 = i1;
    i0 = (u32)((s32)i0 < (s32)i1);
    if (i0) {
        ADD_AND_CHECK_GAS(6);
        i0 = l0;
        i1 = 1u;
        i0 += i1;
        wasm_env__print(h, i0);
    }
    ADD_AND_CHECK_GAS(5);
    i0 = l1;
    wasm_env__print(h, i0);
    i0 = 0u;
    FUNC_EPILOGUE;
```

(continues on next page)

(continued from previous page)

```
23      return i0;
24  }
```

CHAPTER 14

账号权限控制模型

14.1 背景

超级链需要一套去中心化的，内置的权限系统为了实现这个目标，我们借鉴了业界很多现有系统如 Ethereum、EOS、Fabric 的优点，设计一个基于账号的合约权限系统

14.2 名词解释

- `AK(Access Key)`：具体的一个 address，由密码学算法生成一组公私钥对，然后将公钥用指定编码方式压缩为一个地址。
- 账号（`Account`）：在超级链上部署合约需要有账号，账号可以绑定一组 AK（公钥），并且 AK 可以有不同的权重。账号的名字具有唯一性。
- 合约（`Contract`）：一段部署在区块链上的可执行字节码，合约的运行会更新区块链的状态。我们允许一个账号部署多个合约。合约的名字具有唯一性。

14.3 模型简介

系统会首先识别用户，然后根据被操作对象的 ACL 的信息来决定用户能否对其进行哪些操作

- **个人账号 AK：是指一个具体的地址 Addr**
  - 用户的创建是离线的行为，可以通过命令行工具或者 API 进行创建

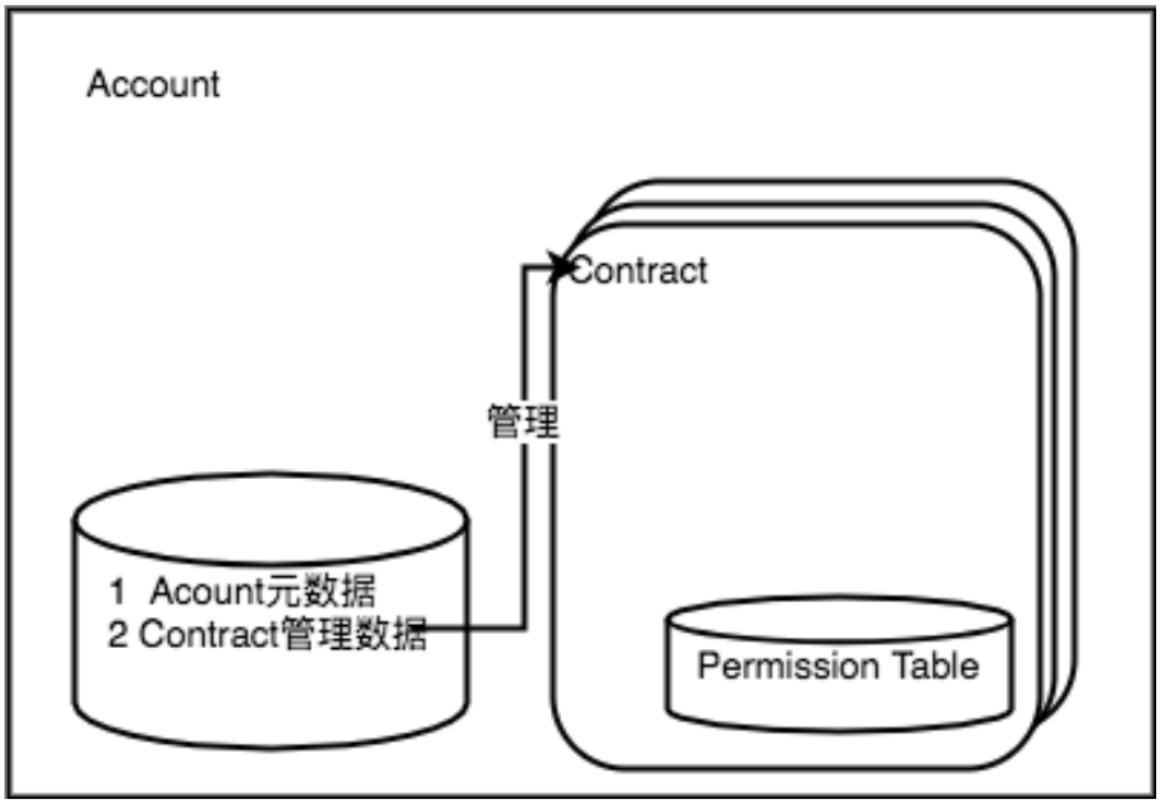

Account
Contract
管理
1 Acount元数据
2 Contract管理数据
Permission Table

Fig. 1: ACL 简介

- **合约账号：超级链智能合约的管理单元。**
  - **账号的创建：**
    * 任何账号或者 AK 都可以调用系统级智能合约创建账号
    * 创建账号需要指定账号对应的拥有者的地址集，如果一个账号中只有一个地址, 那么这个 Addr 对账号完全控制；
    * 创建账号需要指定 ACL 控制策略，用于账号其他管理动作的权限控制；
    * 创建账号需要消耗账号资源；
  - **账号名命名规则；**
    * 合约账号由三部分组成，分为前缀，中间部分，后缀。
    * 前缀为 XC，后缀为 @ 链名
    * 中间部分为 16 个数字组成。
    * 在创建合约账号的时候，只需要传入 16 位数字，在使用合约账号的时候，使用完整的账号。
  - **账号管理：依地址集合据创建时指定的地址集和权限策略，管理账号其他操作**
    * 账号股东剔除和加入
    * 账号资产转账
    * 创建合约，创建智能合约需要消耗账号资源，先将 utxo 资源打到账号下，通过消耗账号的 utxo 资源创建合约，验证的逻辑需要走账号的 ACL 控制
    * 合约 Method 权限模型管理
  - **智能合约：超级链中的一个具体的合约，属于某个账号**
    * 账号所属人员允许在账号内部署合约
    * 账号所属人员可以定义合约管理的权限模型
    * 设置合约方法的权限模型，合约内有一个权限表，记录：{ contract.method，permission_model}
  - 合约命名规则：长度为 4~16 个字符 (包括 4 和 16)，首字母可选项为 [a-ZA-Z_]，末尾字符可选项为 [a-zA-Z0-9_]，中间部分的字符可选项为 [a-zA-Z_.]

14.4 实现功能

主要有两个功能：账号权限管理、合约权限管理

1. 账号权限管理账号的创建、添加和删除 AK、设置 AK 权重、权限模型
2. **合约权限管理设置合约调用权限，支持 2 种权限模型：**

a. 背书阈值：在名单中的 AK 或 Account 签名且他们的权重值加起来超过一定阈值，就可以调用合约

b. AK 集合：定义多组 AK 集合，集合内的 AK 需要全部签名，集合间只要有一个集合有全部签名即可

14.5 系统设计

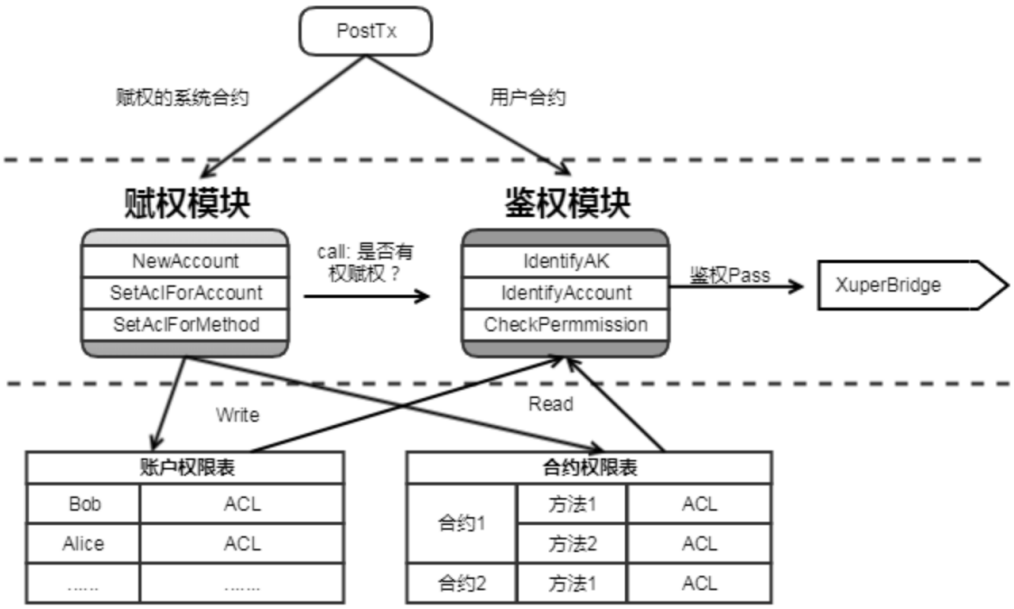


Fig. 2: ACL 架构

14.5.1 ACL 数据结构说明

```
// --------   Account and Permission Section -------
enum PermissionRule {
    NULL = 0;              // 无权限控制
    SIGN_THRESHOLD = 1;    // 签名阈值策略
    SIGN_AKSET = 2;        // AKSet 签名策略
    SIGN_RATE = 3;         // 签名率策略
    SIGN_SUM = 4;          // 签名个数策略
    CA_SERVER = 5;         // CA 服务器鉴权
    COMMUNITY_VOTE = 6;    // 社区治理
}

```

(continues on next page)

(continued from previous page)

```
message PermissionModel {
    PermissionRule rule = 1;
    double acceptValue = 2;     // 取决于用哪种 rule, 可以表示签名率, 签名数或权重阈值
}

// AK 集的表示方法
message AkSet {
    repeated string aks = 1; //一堆公钥
}

message AkSets {
    map<string, AkSet> sets = 1;   // 公钥 or 账号名集
    string expression = 2;       // 表达式, 一期不支持表达式, 默认集合内是 and, 集合间是 or
}

// Acl 实际使用的结构
message Acl {
    PermissionModel  pm = 1;              // 采用的权限模型
    map<string, double>  aksWeight = 2;  // 公钥 or 账号名  -> 权重
    AkSets akSets = 3;
}
```

签名阈值策略：`Sum{Weight(AK_i) , if sign_ok(AK_i)} >= acceptValue`

14.5.2 系统合约接口

合约接口	用途
NewAccountMethod	创建新的账号
SetAccountACLMethod	更新账号的 ACL
SetMethodACLMethod	更新合约 Method 的 ACL

14.5.3 样例

acl 模型如下：

```
{
    "pm": {
        "rule": 1,

```

(continues on next page)

(continued from previous page)

```
        "acceptValue": 1.0
    },
    "aksWeight": {
        "AK1": 1.0,
        "AK2": 1.0
    }
}
```

- 其中 rule=1 表示签名阈值策略，rule=2 表示 AKSet 签名策略
- 签名的 ak 对应的 weight 值加起来 >acceptValue，则符合要求

CHAPTER 15

超级链 p2p 网络

15.1 p2p 网络概述

依据 p2p 网络中节点相互之间如何联系，可以将 p2p 网络简单区分为无结构和结构化两大类:

1. 非结构化 p2p 网络：这种 p2p 网络是最普通的，没有对结构做特别的设计。优点在于结构简单易于组件，网络局部区域内个体可以任意分布。对于节点的加入和离开网络也表现地非常稳定，比特币网络使用的就是无结构化的网络。但是这种网络主要有 3 个缺点，一是公网网络拥塞时传输效率低，二是存在泛洪循环，三是消息风暴问题。
2. 结构化 p2p 网络：这种 p2p 网络的结构经过精心设计，目的是为了增加路由效率，提高查询数据的效率，结构化 p2p 最普遍的实现方案是使用分布式哈希表（DHT），以太坊网络中使用的就是结构化的网络。

互联网的发展速度远远超过人们的预期，人们在制定网络协议之初没有考虑过网络规模会获得如此迅速的增长，导致 ip 地址的短缺。NAT 技术通过将局域网内的主机地址映射为互联网上的有效 ip 地址，实现了网络地址的复用，从而部分解决了 ip 地址短缺的问题。网络中大部分用户处于各类 NAT 设备之后，导致在 p2p 网络中两个节点之间直接建立 udp 或者 tcp 链接难度比较大，应运而生的是 NAT 穿透技术。目前主要有两种途径，一种称为打洞，即 UDP Punch 技术；另一种是利用 NAT 设备的管理接口，称为 UPnP 技术。

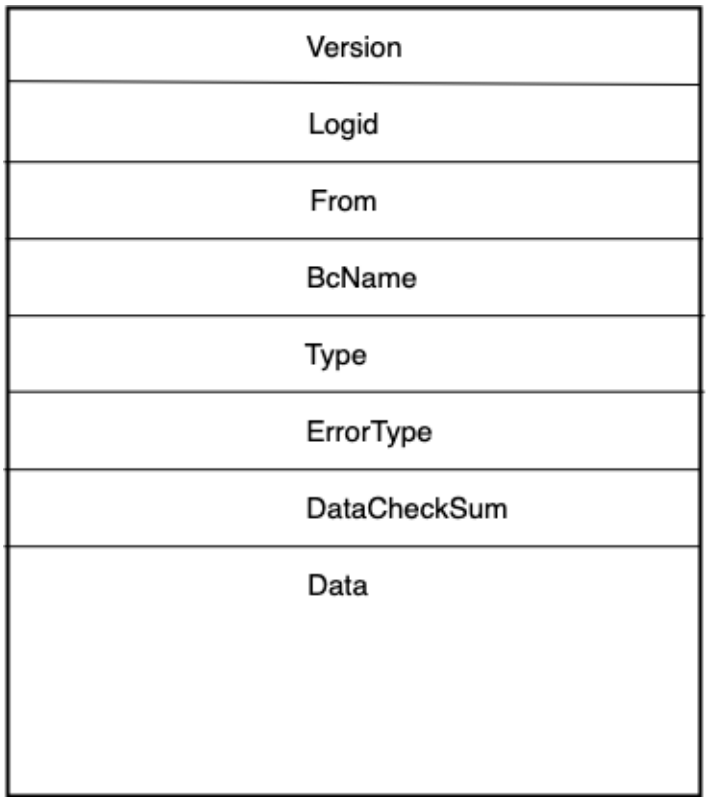
15.2 超级链 p2p 网络

超级链采用 libp2p 作为网络的基础设施，使用 KAD 进行节点的路由管理，支持 NAT 穿透。超级链的定义了自己的协议类型 **XuperProtocolID = "/xuper/2.0.0"**，所有的超级链网络节点除了基础的消息类型

外还会监听并处理这个协议的网络消息。

15.2.1 超级链 p2p 消息

超级链消息采用 Protobuf 定义，整个消息包括 2 部分，分别是消息头 `MessageHeader` 和消息体 `MessageData`，具体如下所示：



其 proto 消息定义如下：

```
// XuperMessage is the message of Xuper p2p server
message XuperMessage {
    // MessageHeader is the message header of Xuper p2p server
    message MessageHeader {
        string version = 1;
        // dataCheckSum is the message data checksum, it can be used check where the␣
↪message have been received
        string logid = 2;
        string from = 3;
        string bcname = 4;
        MessageType type = 5;
        uint32 dataCheckSum = 6;
        ErrorType errorType = 7;
```

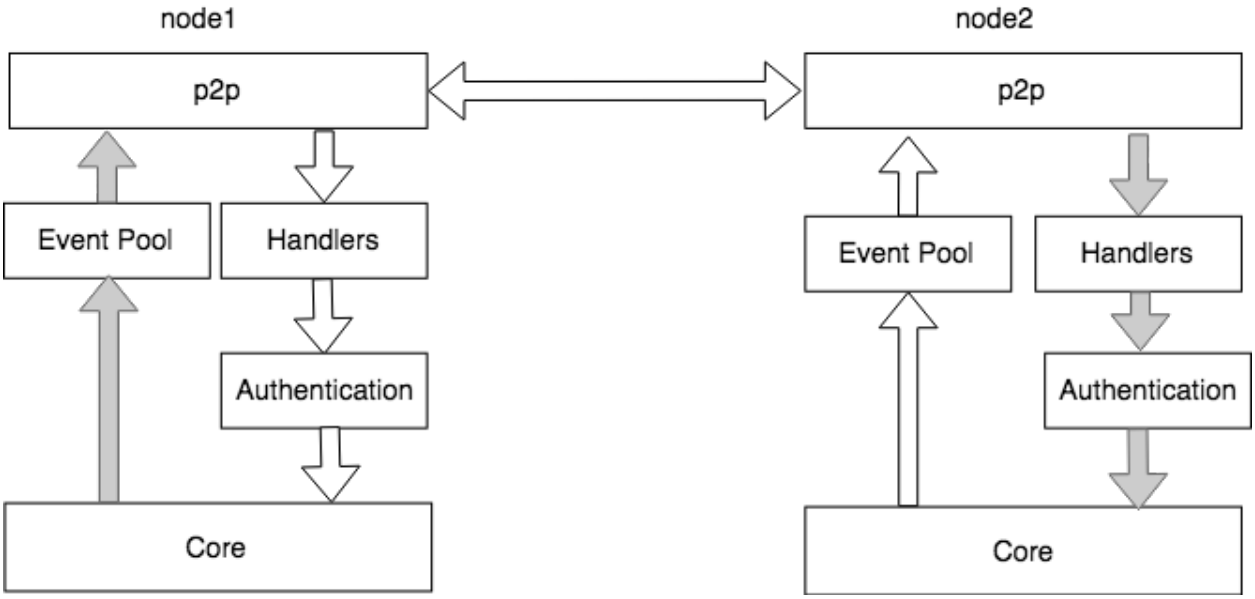
(continues on next page)

(continued from previous page)

```
    }
    // MessageData is the message data of Xuper p2p server
    message MessageData {
        // msgInfo is the message infomation, use protobuf coding style
        bytes msgInfo = 3;
    }
    MessageHeader Header = 1;
    MessageData Data = 2;
}
```

15.2.2 模块交互图

超级链 p2p 网络模块与其他模块的交互如图所示：



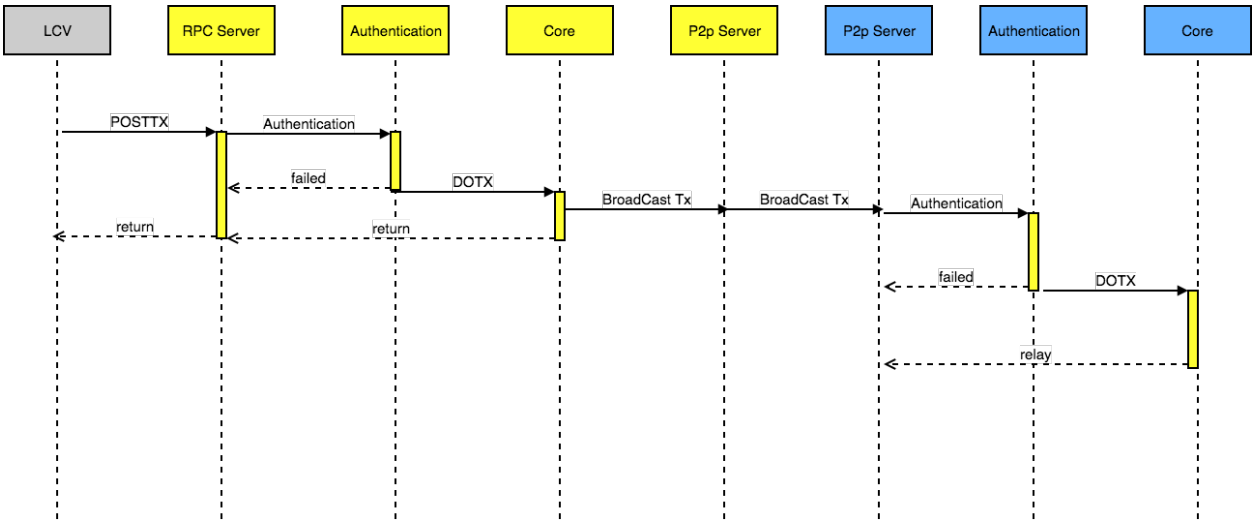
上图左边是 Xuper 的启动流程，其中 InitP2PServer 的流程为启动 P2P 的核心流程，如右半部分所示，右半部分主要包括 4 个阶段，分别为：

1. InitP2pInstance：创建 libp2p host 实例
2. SetXuperStreamHandler：初始化 p2p 通信消息 protocols，XuperProtocol 为 Xuper 节点之间进行消息通信和消息处理的核心逻辑。
3. InitKadDht: 初始化 libp2p KadDht，通过设置的 bootstrap 节点，建立自己的 kad dht。
4. InitStreams: 前一步已经建立了自己的 kad dht，下一步就是与这些邻近的节点之间建立通信流，通过 libp2p 的 NewStream 接口实现通信流建立。

至此，Xuper 的 p2p 连接建立完毕。

15.2.3 交易消息处理流程

用户提交的交易消息在超级链网络中传输的处理流程如下所示：



用户通过 RPC 将交易提交到网络中，交易执行成功后会通过 p2p 模块广播给网络中的其他节点。

CHAPTER 16

身份认证

16.1 背景

Xuperchain 节点之间存在双重身份：P2P 节点 ID 和 Xuperchain address，为了解决节点间的身份互信，防止中间人攻击和消息篡改，节点间需要一种身份认证机制，可以证明对称节点声明的 XChain address 是真实有效的

16.2 名词解释

- Xuperchain address：当前节点的 address，一般为 data/keys/address
- P2P 节点 ID：当前节点 P2P 的 peer.ID

16.3 P2P 建立连接过程

16.4 实现过程

- 新建的 net.Stream 连接，已经完成了 ECDH 密钥协商流程，因此此时节点间已经是加密连接。
- 连接建立后，增加一步身份认证流程，如果通过，则 stream 建立成功，加入到 streamPool 中

其中，身份认证流程如下：

- 身份认证流程通过开关控制，可开启和关闭 DefaultIsAuthentication: true or false

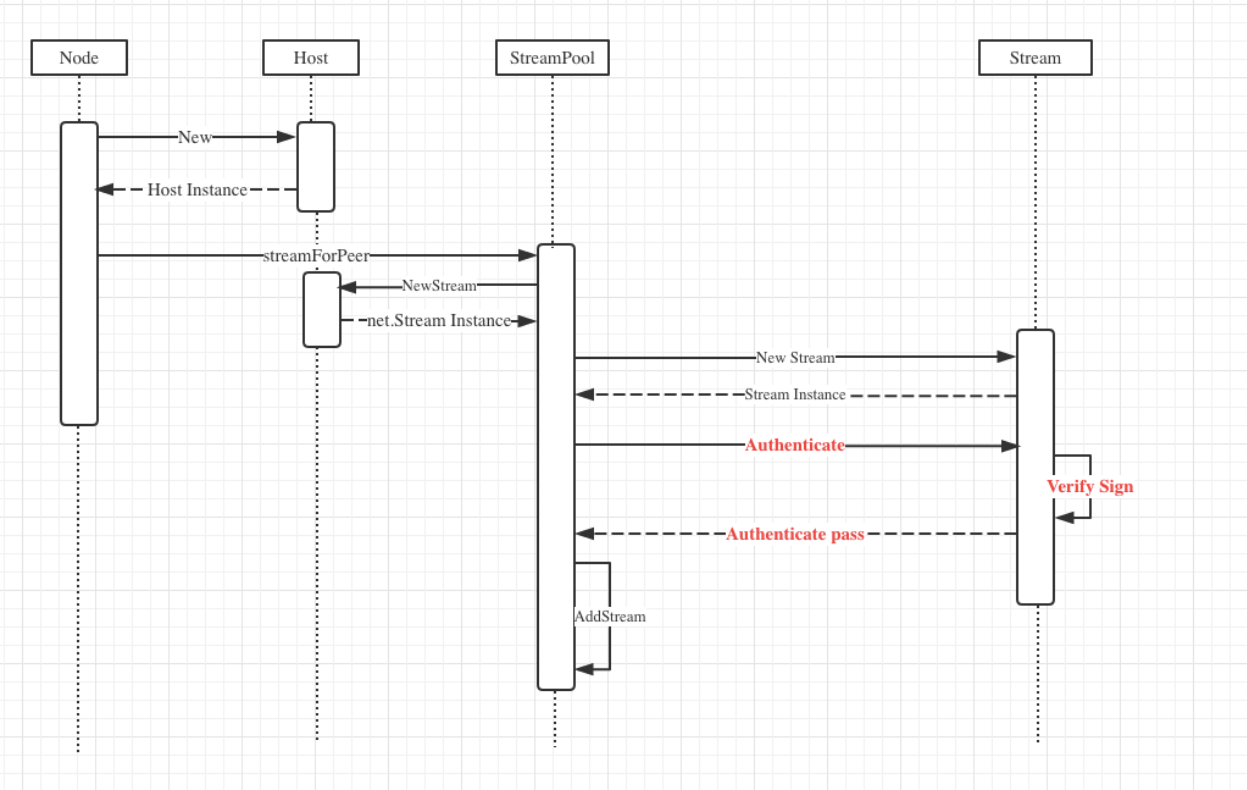


Fig. 1: 连接建立时序

- 身份验证支持 XChain address 的验证方式
- 如果开启身份验证，则身份验证不通过的 Stream 直接关闭
- 身份验证是使用 XChain 的私钥对 PeerID+XChain 地址的 SHA256 哈希值进行签名，并将 PeerID、Xuperchain 公钥、Xuperchain 地址、签名数据一起传递给对方进行验证

16.5 主要结构修改点

```
// stream 增加 authenticate 接口
func (s *Stream) Authenticate() error {}

// 收到身份验证消息后的回调处理函数接口

func (p *P2PServerV2) handleGetAuthentication(ctx context.Context, msg *xuper_p2p.
↪XuperMessage) (*xuper_p2p.XuperMessage, error) {}
```

CHAPTER 17

提案和投票机制

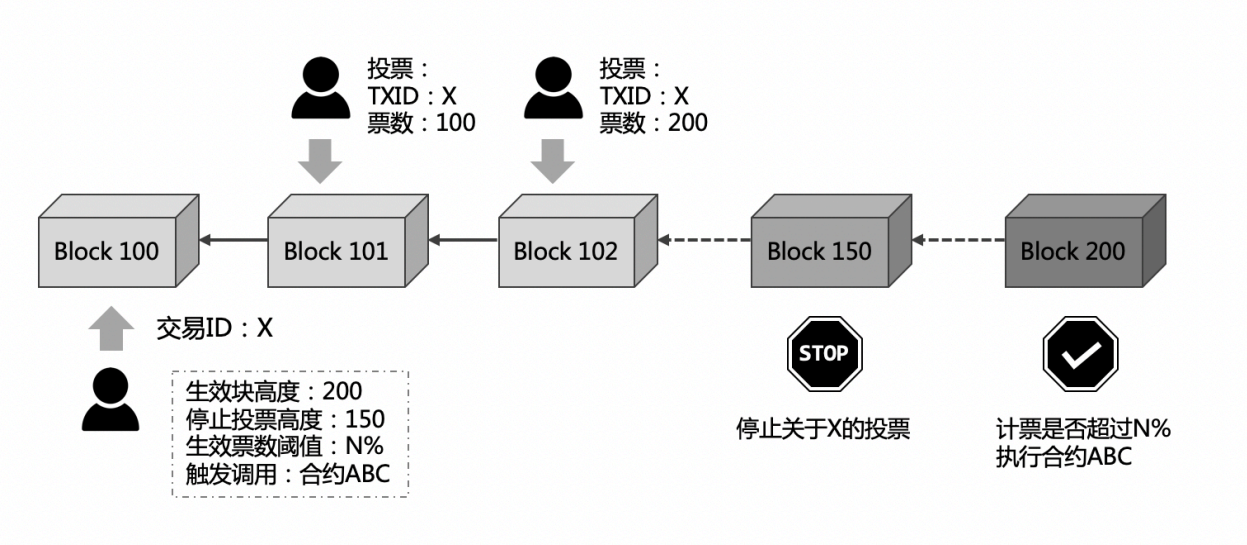


Fig. 1: 图 1：提案和投票机制示意图

提案和投票机制是区块链系统实现自我进化的关键。系统首次上线后难免遇到很多问题，我们提供提案/投票机制为区块链的社区治理提供便利的工具，以保证未来系统的可持续发展。具体实现方法如下：

Step1：提案者（proposer) 通过发起一个事务声明一个可调用的合约，并约定提案的投票截止高度，生效高度；Step2：投票者 (voter) 通过发起一个事务来对提案投票，当达到系统约定的投票率并且账本达到合约的生效高度后，合约就会自动被调用；Step3：为了防止机制被滥用，被投票的事务的需要冻结参与者的一笔燃料，直到合约生效后解冻。

17.1 共识可升级

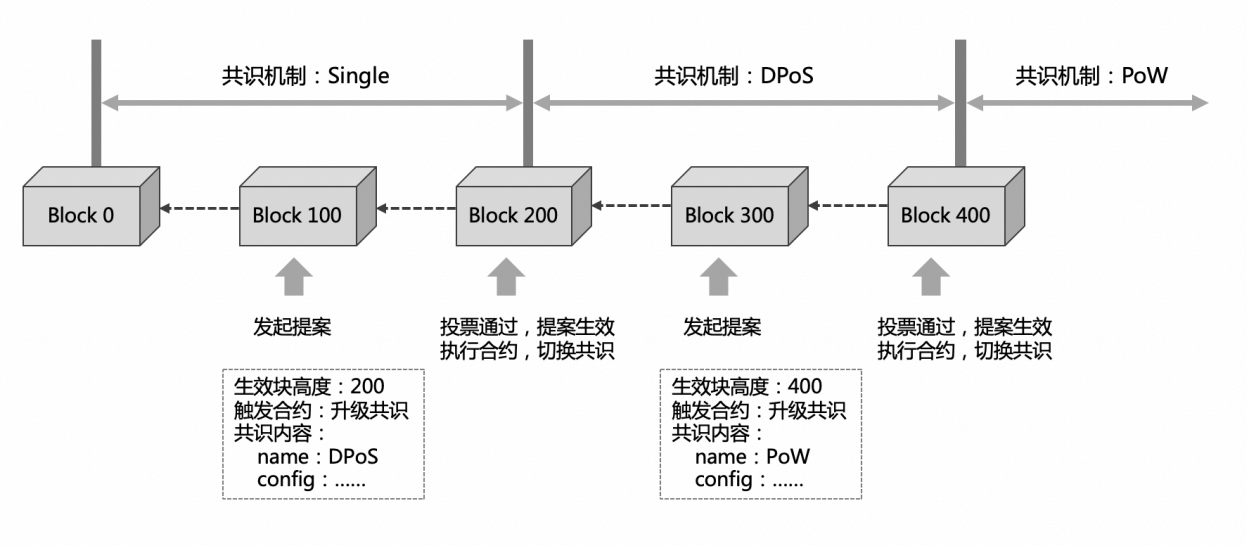


Fig. 2: 图 2：XuperChain 提案机制进行共识升级

XuperChain 提供可插拔共识机制，通过提案和投票机制，升级共识算法或者参数。图 2 简要说明了如何使用 XuperChain 的提案机制进行共识升级。

17.2 系统参数可升级

通过提案和投票机制，区块链自身的运行参数也是可升级的。包括：block 大小、交易大小、挖矿奖励金额和衰减速度等。

下面通过一个例子来说明，假设一条链，最开始用的是 POW 共识，创始块如下：

```
{
    "version" : "1",
    "predistribution": [
        {}
    ],
    "maxblocksize" : "128",
    "award" : "1000000",
    "decimals" : "8",
    "award_decay": {
        "height_gap": 31536000,
        "ratio": 0.5
    },
```

(continues on next page)

(continued from previous page)

```
    "genesis_consensus": {
        "name": "pow",
        "config": {
                "defaultTarget": "19",      # 默认难度 19 个 0 bits 前缀
                "adjustHeightGap": "10",    # 每 10 个区块调整一次难度
                "expectedPeriod": "15",     # 期望 15 秒一个区块
                "maxTarget": "22"
        }
    }
}
```

然后，我们想将其共识切换到 TDPOS 共识。

步骤 1: 由提案者发起提案，提案没有额外的代价，通过命令行的 desc 选项指向提案用的 json 即可。提案 json 的内容如下：

```
{
    "module": "proposal",
    "method": "Propose",
    "args" : {
        "min_vote_percent": 51,     # 当投票者冻结的资产占全链的 51% 以上时生效提案
        "stop_vote_height": 120     # 停止计票的高度是:120
    },
    "trigger": {
        "height": 130,              # 提案生效高度是: 130
        "module": "consensus",
        "method": "update_consensus",
        "args" : {
            "name": "tdpos",
            "config": {
                "proposer_num":"3",
                "period":"3000",
                "term_gap":"60000",
                "alternate_interval": "3000",
                "term_interval": "6000",
                "block_num":"10",
                "vote_unit_price":"1",
                "init_proposer": {
                    "1": ["dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
↪"f3prTg9itaZY6m48wXXikXdcxiByW7zgk", "U9sKwFmgJVfzgWcfAG47dKn1kLQTqeZN3"]
                }
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

把上面的 json 保存在 myprop.json, 然后运行：

```
./xchain-cli transfer --to `cat data/keys/address` --desc ./myprop.json --amount 1
```

得到一个 txid，此处为 67cc7cd23b7fcbe0a4919d5c581b3fda759da13cdd97414afa7539e221727594

然后，通过

```
./xchain-cli tx query 67cc7cd23b7fcbe0a4919d5c581b3fda759da13cdd97414afa7539e221727594
```

确认该交易已经上链（标志是 blockid 不为空了）

步骤 2：可以对这个提案投票。投票需要冻结自己资产，并且冻结高度必须大于停止计票的高度。

```
./xchain-cli vote -amount 100000000 -frozen 121␣
↪67cc7cd23b7fcbe0a4919d5c581b3fda759da13cdd97414afa7539e221727594
```

Note: 注意：冻结高度 121 需要大于提案停止计票高度 120, 否则是无效投票。

另外，累计投票金额数量必须大于全链总量的 51% (51% 是提案 json 中指定的，但是最小不能少于 50%)

```
./xchain-cli account balance -Z # 可以查看自己被冻结的资产总量。
./xchain-cli status --host localhost:37301  | grep -i total # 查询全链的资产总量。
```

步骤 3：最后，等到当前生效高度到达，会发现共识已经切换到 TDPOS 了。

```
./xchain-cli tdpos status
```

此命令可以查看 tdpos 状态。

CHAPTER 18

密码学和隐私保护

18.1 背景

密码学技术是区块链的核心基础技术之一，承担着区块链不可篡改和去中心化验证等特性的底层支撑。在超级链中，密码学技术广泛应用在账户体系、交易签名、数据隐私保护等方面，主要以 ECC(椭圆曲线密码体系) 以及多种 Hash 散列算法为基础，发展出的一个单独的模块。

18.2 密码学基础

18.2.1 哈希函数

加密哈希函数（Hash Function）是适用于密码学的哈希散列函数，是现代密码学的基本工具。它是一种数学算法，将任意大小的数据（通常称为“消息”）映射到固定大小的二进制串（称之为“散列值”，“散列”或“消息摘要”），并且是单向的功能，即一种实际上不可逆转的功能。理想情况下，查找生成给定哈希的消息的唯一方法是尝试对可能的输入进行暴力搜索，以查看它们是否产生匹配，或使用匹配哈希的彩虹表。

- MD5：摘要长度为 128bit，由于容易受到碰撞攻击，目前使用越来越少。
- SHA256：SHA 系列哈希算法由美国国家安全局制定，具有多个 hash 算法标准，可以产生 160~512bit 不等的哈希摘要。目前在区块链中使用较多的是 SHA256，摘要长度为 256bit，具有较高的抗碰撞攻击安全性。
- RIPEMD-160：产生长度为 160bit 的摘要串。相比于美国国家安全局设计的 SHA-1 和 SHA-2 算法，RIPEMD-160 的设计原理是开放的。

关于一些典型的 Hash 算法的对比，可以参考 这里 。

18.2.2 ECC

构建区块链的去中心化交易，需要一种加密算法，使交易发起人使用持有的密钥对交易数据进行数字签名，而交易验证者只需要知道交易发起人的公开信息，即可对交易有效性进行验证，确定该交易确实来自交易发起者。这种场景在密码学中称之为 '公开密钥加密 '，也称之为非对称密钥加密。

常见的公开密钥算法如 RSA、ECC(Elliptic Curve Cryptography，缩写为 ECC) 等，RSA 起步较早，此前在非对称加密领域使用范围最广，例如目前的 SSL 证书大多采用 RSA 算法。而在 ECC 算法问世后，由于在抗攻击性、资源消耗等方面相比 RSA 具有更好的表现，其使用也越来越广泛。

公钥密码算法一般都基于一个数学难题，比如 RSA 的依据是给定两个数 p, q 很容易相乘得到 N，当 N 足够大时，对 N 进行因式分解则相对困难的多。**ECC 是建立在基于椭圆曲线的离散对数问题上的密码体制，给定椭圆曲线上的一个点 P，一个整数 k，求解 Q=kP 很容易；给定一个点 P、Q，知道 Q=kP，求整数 k 却是一个难题**。具体的理论知识可以参考 椭圆曲线密码学。

椭圆曲线密码学包含了多种密码学算法，下面列出在超级链中涉及到的一些算法：

- `ECIES` (Elliptic Curve Integrated Encryption Scheme): 椭圆曲线集成加密算法，主要用于基于椭圆曲线的数据加解密。
- `ECDH` (Elliptic Curve Diffie–Hellman): 基于 Diffie–Hellman 算法的一种密钥协商算法，定义了双方如何安全的生成和交换密钥。
- `ECDSA` (Elliptic Curve Digital Signature Algorithm): 是使用椭圆曲线密码学实现的 DSA(数字签名算法)，一般发起人对消息摘要使用私钥签名，验证者可以通过公钥对签名有效性进行验证。

椭圆曲线算法由于采用的椭圆曲线的不同，具有多种不同的算法标准，典型的如：

- NIST 标准，典型的曲线如 P-256/P-384/P-521 等；
- SECG 标准，典型的如 Secp256k1/Secp256r1/ secp192k1/ secp192r1 等；
- ECC25519，主要指 Ed25519 数字签名和 Curve25519 密钥协商标准等；
- 国产密码算法，中国国家密码局制定的密码学算法标准，典型的如 SM2/3/4 等。

18.2.3 多重签名和环签名

多重签名是指在数字签名中，有时需要多个用户对同一个交易进行签名和认证，例如某些合约账户下的数据需要多个人授权才能修改或转账。

在密码学中，通过多重签名可以将多个用户的授权签名信息压缩在同一个签名中，这样相比于每个用户产生一个签名的数据体量会小很多，因此其验签计算、网络传输的资源开销也会少很多。

环签名是一种数字签名技术，环签名的一个安全属性是无法通过计算还原出一组用户中具体使用私钥签名的用户。也就是说，使用环签名技术可以使一组用户中的某一个人对消息进行签名，而并不会泄露签名者是这

组用户中的哪个人。环签名与组签名类似，但在两个关键方面有所不同：第一，单个签名具有匿名性; 第二，任何一批用户都可以作为一个组使用，无需额外设置。

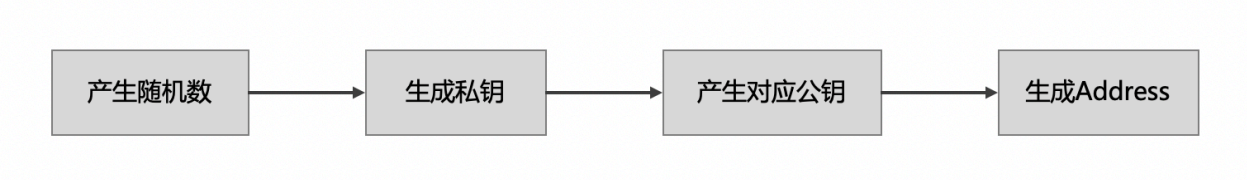
在实际使用中，多重签名主要用作多人实名授权的交易，通过产生更小的签名数据提升网络传输和计算效率，而环签名则主要用于对交易隐私保护和匿名性有要求的交易场景中。

18.3 超级链中密码学的使用

密码学作为区块链系统的底层基础技术，在很多方面都会使用到。这里介绍几个超级链中几个密码学典型的使用场景。

18.3.1 用户公私钥账户

超级链的用户账户体系基于非对称公私钥对，每个用户账户对应这一组公私钥对，并采用一定的哈希算法将公钥摘要成一个字符串作为用户账户地址 (address)。



超级链中公私钥对使用椭圆曲线算法生成，用户账户地址主要使用 SHA256 和 RIPEMD-160 哈希算法生成。

考虑到密钥不具备可读性，为了帮助用户保存密钥，超级链实现了 BIP39 提案的助记词技术。

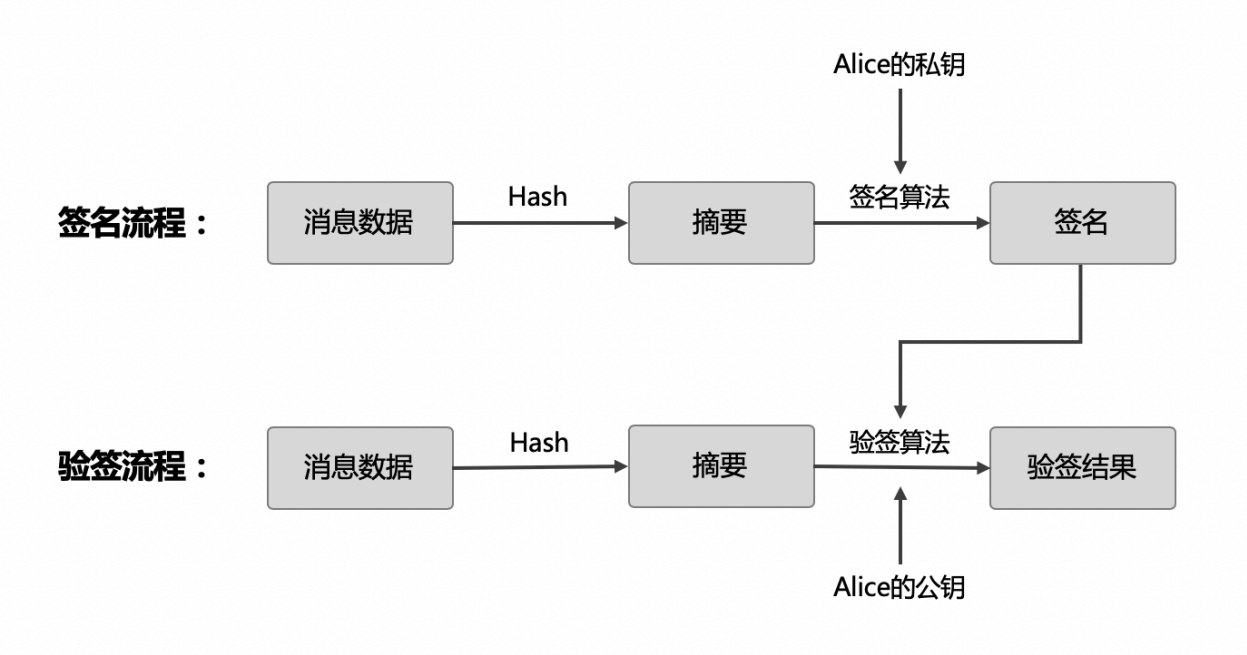
- 助记词的生成过程：首先生成一个长度在 128~256bit 之间的随机熵，由此在助记词表中选出对应的单词列表，形成助记词。
- 助记词产生私钥：使用基于口令的密钥派生算法 PBKDF2，将上述生成的助记词和用户指定的密钥作为密钥派生算法参数，生成长度为 512bit 的种子，以此种子作为生成密钥的随机参数，便产生了了从助记词生成的私钥。
- 通过助记词恢复密钥：由于用户持有生成密钥的助记词和口令，因此在用户私钥遗忘或丢失时，可以通过同样的助记词和口令，执行助记词产生私钥的过程，从而恢复出账户密钥。

18.3.2 数据签名

超级链中，每个交易都需要交易发起人以及交易背书人的签名；在每个块生成时，也需要加上打包块的节点的签名。

- 交易签名：基于交易数据摘要，会包含交易输入输出、合约调用、合约读写集、发起人和背书人信息等，并将交易数据序列化后的字节数组使用双重 SHA256 得到摘要数据，最后对摘要数据用 ECDSA 或其他数字签名算法产生交易签名。

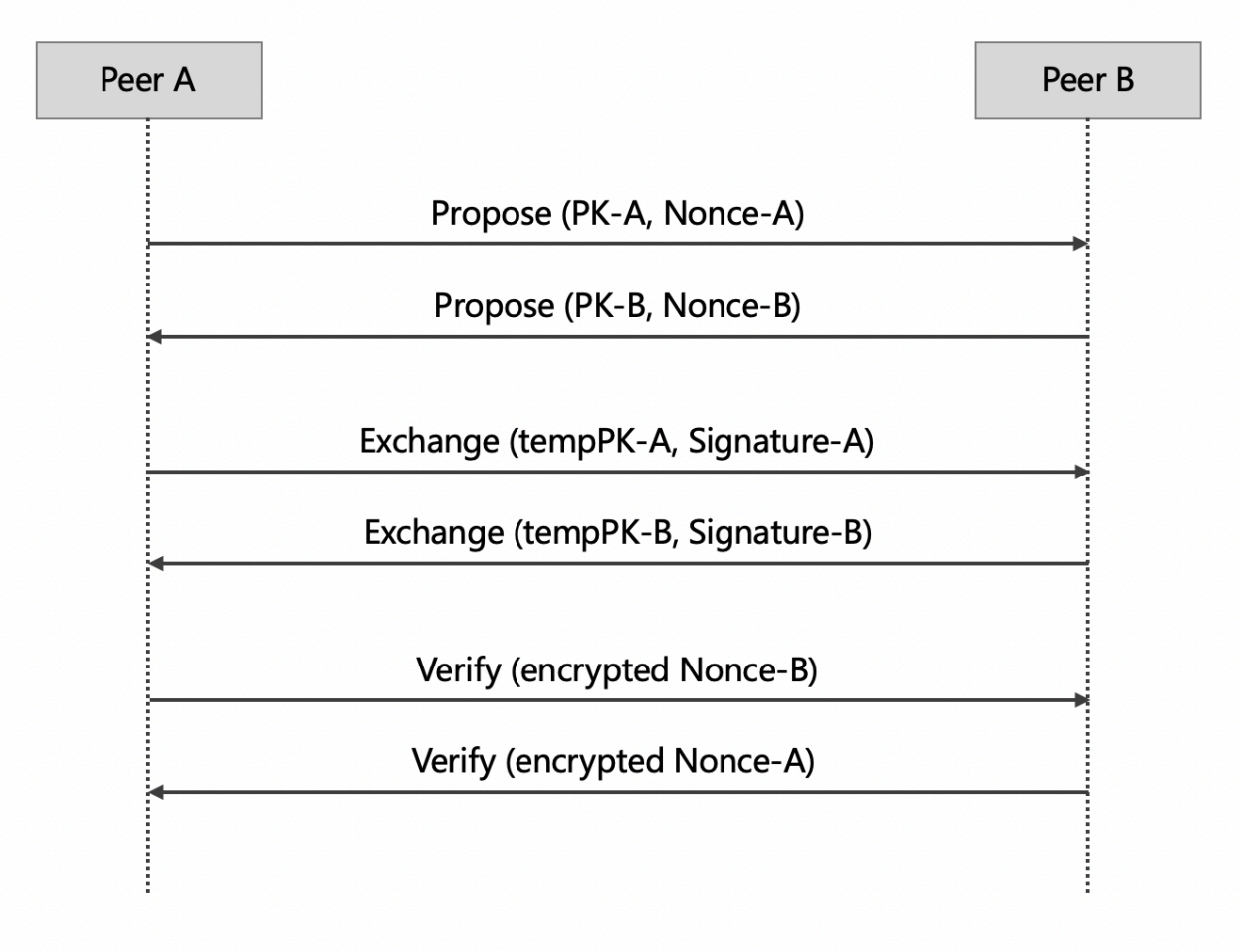
- 块签名：基于区块数据摘要，会包含区块元信息如前序块 Hash 值、交易 Merkle 树根、打包时间、出块节点等数据，并在序列化后使用双重 SHA256 得到摘要数据，最后对摘要数据用 ECDSA 或其他数字签名算法产生区块签名。



18.3.3 通信加密

超级链底层采用 P2P 网络传播交易和区块数据，在一些许可区块链网络场景中，需要对节点间的数据传输进行加密提升安全和隐私性，因此超级链的 P2P 连接支持基于 ECDH 的密钥交换算法的 TLS 连接。

ECDH 的原理是交换双方可以在不共享任何秘密的情况下协商出一个密钥，双方只要知道对方的公钥，就能和自己的私钥通过计算得出同一份数据，而这份数据就可以作为双方接下来对称加密的密钥。



超级链 P2P 网络通过 ECDH 建立通信加密通道的过程如上图所示：

- 第一阶段是 Propose 阶段，这一阶段，对等节点间互相交换双方永久公钥 PK。
- 第二阶段是 Exchange 阶段，本质是基于 ECDH 的密钥交换。双方通过 ECC 算法随机生成一组临时密钥对 (tempPK, tempSK)，然后用自己的永久私钥对临时公钥 tempPK 进行签名并交换。这时，双方可以通过第一步的公钥进行验签，同时拿到供本次会话使用的临时公钥。使用临时公钥的好处是一话一密，即使本次会话密钥泄露也不会导致以前的加密数据被破解。ECDH 算法使得双方通过对方的公钥和自己的私钥，可以获得一致的共享密钥 SharedKey。
- 第三阶段是 Verify 阶段。双方使用 ShareKey 产生两组密钥 Key1，Key2 分别作为读写密钥，并使用支持的对称加密算法 (AES/blowfish) 加密传输第一步中发送给对方的 Nonce，而接收方则使用刚才协商的密钥对数据解密，并验证 Nonce 是不是等于第一步中自己发送给对方的值。

通过这三次握手，双方建立了加密通信通道，并且节点间通信加密信道满足一话一密的高安全等级。

18.4 密码学模块

18.4.1 Crypto Provider Interface

密码学作为区块链系统的底层技术，相对比较独立。考虑到超级链作为区块链底层系统方案的模块化目标，我们将密码学相关的功能设计成一个单独的模块，并通过插件化技术实现了模块可插拔、插件可替换。

因此，超级链首先抽象出了统一的密码学相关的功能，并在此之上定义了统一的密码学接口，我们称之为 Crypto Provider Interface，并通过 CryptoClient 接口向上层区块链系统系统密码学功能。CryptoClient 目前由一组接口构成：

```
// CryptoClient is the interface of all Crypto functions
type CryptoClient interface {
    CryptoCore
    KeyUtils
    AccountUtils
    MultiSig
}
```

整个 CryptoClient 由四部分功能接口组成：

- `CryptoCore`： 主要提供包括加解密、签名等密码学核心功能；
- `KeyUtils`：主要提供公私钥相关工具，例如密钥对象和 JSON、文件格式之间的转换等；
- `AccountUtils`：主要提供账户相关的功能接口，例如创建账户、助记词导出私钥等；
- `MultiSig`：主要提供多重签名、环签名相关功能接口。

18.4.2 密码学插件

由于抽象出了统一的密码学模块和接口，在此基础上实现插件化就比较容易。目前超级链已经实现了包括 *Nist P256 + ECDSA/Schnorr* 以及 国密等多种密码学插件，其中已经开源了 *Nist P256 + ECDSA/Schnorr*，ECDSA 和 Schnorr 签名作为两种可选的签名方案，分别提供了密码学插件。

为了方便框架使用密码学插件，超级链在 **crypto/client** 包中封装了一层密码学插件管理器，支持创建指定类型的密码学对象，或者通过公私钥自动识别需要加载的插件类型。通过密码学插件管理器，可以支持隔绝框架对密码学插件的感知，对上层框架提供一种无缝的使用体验。

超级链中默认密码学插件使用的是 Nist P256 + ECDSA，在不额外指定的情况下，超级链启动后会加载默认密码学插件。

之前说过，通过密码学插件管理器可以按照公私钥自动识别需要加载的插件类型，那么超级链如何根据密钥来判断应该使用哪种密码学插件呢？其实，不同的密码学插件是通过密钥中的曲线类型来确定的，目前系统中定义了三种不同的曲线类型：

- `P-256`：使用 Nist P256+ECDSA 的默认插件；

- `P-256-SN`：使用 Nist P256 + Schnorr 签名的插件，可以提供更高的签名验签性能;
- `SM2-P-256`：使用 SM2/3/4 的国密插件，符合中国国家密码局制定的密码学标准。

实际使用中，可以通过创建链时的配置中的密码学类型指定使用哪种密码学插件，以 schnorr 签名为例，在创世块配置中添加下述配置即可：

```
"crypto": "schnorr"
```

在 cli 命令行工具中已经支持了通过命令行参数 **–cryptotype** 指定密码学插件的类型，例如需要创建一个使用 Nist P256 + Schnorr 的密码学插件的用户账户，可以使用下述命令行：

```
./xchain-cli account newkeys --output data/tmpkey --cryptotype schnorr
```

CHAPTER 19

插件机制

19.1 可插拔架构

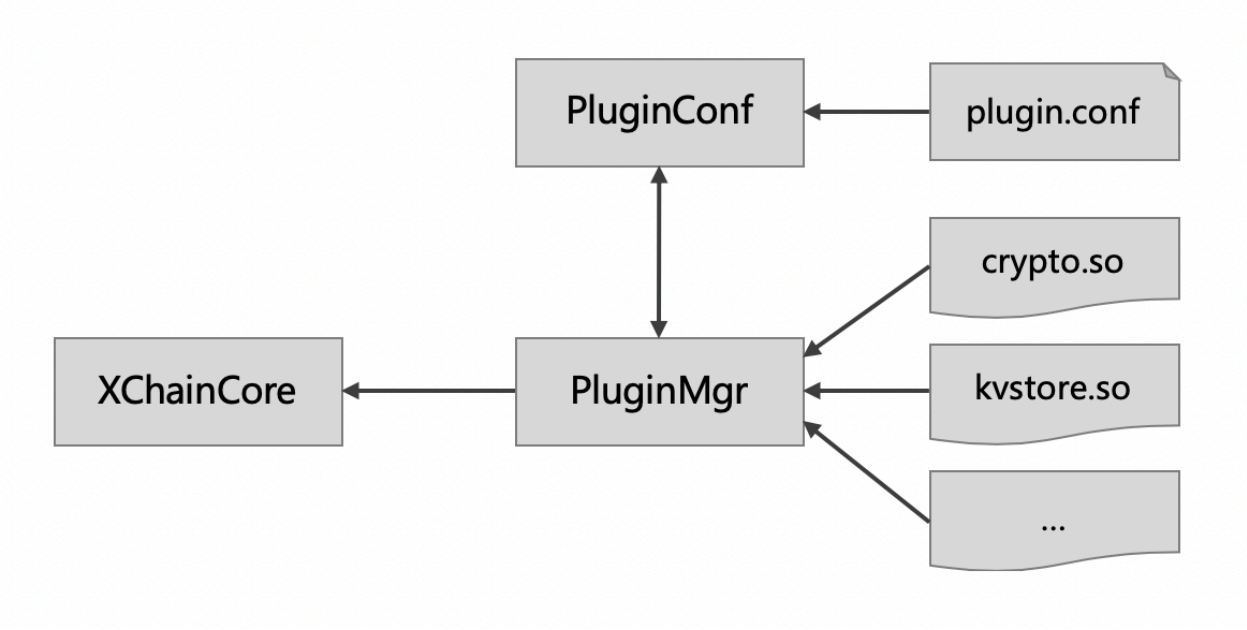
超级链从设计之初就以高性能、可插拔的区块链底层技术架构为目标，因此整个超级链在模块化、接口化设计上做了很多的抽象工作。而插件化机制就是服务于超级链可插拔的架构目标，使得所有模块具有同样的可插拔机制，并能满足对模块插件的加载、替换等生命周期的管理。

通过插件化机制可以实现如下架构优点：

- **代码解耦**：插件化机制使超级链的架构框架与各个模块的实现相解耦，模块统一抽象出基本数据结构与框架的交互接口，模块只要符合统一接口即可做到插拔替换。
- **高可扩展**：用户可以自己实现符合业务需求的模块插件，直接替换插件配置就可以实现业务扩展。
- **发布灵活**：插件可以单独发布，配合插件生命周期管理甚至可以实现插件的单独更新，而作为插件的开发者也可以自由选择开源发布或者只发布插件二进制文件。

19.2 插件框架设计

插件框架用以根据需求创建插件实例，考虑到超级链单进程多链、插件多实例多版本等需求，整体设计



19.2.1 模块和插件定义

超级链中，一种 **模块**是指：包含一组相同数据结构和接口的代码集合，能实现相对独立的功能。

一个模块可以有多种实现，每种实现形成一个 **插件**。

模块和插件具有如下约束：

- 同一种模块，需要抽象出公共数据接口和接口方法。
- 该模块的所有插件，需要实现定义的所有公共接口，并不包含定义接口以外的 public 接口。
- 每个插件需要实现一个 `GetInstance` 接口，该接口创建并返回一个插件对象引用，该插件对象包含插件定义的所有公共接口。

因此，我们可以在框架中定义一组公共的数据结构和接口，例如：

```
package kvstore

type KVStore interface {
    Init(string)
    Get(string) (string, error)
    Set(string, string) error
}
```

并在插件代码中，引用并实现该公共接口定义，例如：

```go
import "framework/kvstore"

type KVMem struct {
    meta    kvstore.Meta
    data    map[string]string
    rwmutex sync.RWMutex
}

// 每个插件必须包含此方法，返回一个插件对象
func GetInstance() interface{} {
    kvmem := KVMem{}
    return &kvmem
}

// 插件需要实现接口定义中的所有方法
func (ys *YourKVStore) Init(conf string) {
    // Your code here
}

func (ys *YourKVStore) Get(key string) (string, error) {
    // Your code here
}

func (ys *YourKVStore) Set(key string, value string) error {
    // Your code here
}
```

19.2.2 配置化管理

插件通过配置文件组织可选插件以及 **插件子类型**、**插件路径**、**版本**等信息。考虑到同一个链中可能需要创建某个插件的多种实例，因此所有的插件都以数组的方式声明该插件不同的子插件类型对应的链接库地址。

举例如下：

```
{
    "kvstore":[
        {
            "subtype": "Memory",
            "path": "plugins/kv-memory.so.1.0.1",
            "version": "1.0.1",
            "ondemand": false
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "subtype": "Json",
            "path": "plugins/kv-json.so.1.0.0",
            "version": "1.0.0",
            "ondemand": false
        }
    ],
    "crypto":[
        {
            "subtype": "GuoMi",
            "path": "plugins/crypto/crypto-gm.so.1.1.0",
            "version": "1.1.0",
            "ondemand": false
        },
    ]
}
```

19.2.3 PluginMgr

PluginMgr 定义了插件管理的对外接口。

```
// 根据插件配置文件初始化插件管理对象
func CreateMgr(confPath string) (pm *PluginMgr, err error);

// 指定插件名称和插件子类型，获取该插件的一个实例
func (pm *PluginMgr) CreatePluginInstance(name string, subtype string)
```

需要插件功能的主逻辑中，要通过 **CreateMgr** 创建一个 PluginMgr 的实例，该实例会根据传入的配置文件创建插件实例。

19.2.4 PluginMgr 使用

每个模块可以定义自己的实例创建方法，并可以自行确定是否使用默认模块，或使用插件化的模块。

```
func NewKVStore(pm *pluginmgr.PluginMgr, subType string) (store KVStore, err error) {
    var iface interface{}
    iface, err = pm.CreatePluginInstance(KV_PLUGIN_NAME, subType)
    if err != nil {
```

(continues on next page)

(continued from previous page)

```
        return
    }

    if iface != nil {
        // registered external plugin
        store = iface.(KVStore)
    } else {
        // no plugin registered, use default one
        store = new(KVText)
    }
    return
}
```

19.3 超级链的插件

目前，插件化机制已经在超级链中应用于包括密码学、共识、KV 引擎等多个核心模块中，初步实现了插件的解耦和可扩展性目标。

以密码学为例，通过插件机制，我们可以实现多套不同的密码学算法的封装，目前超级链已经实现了包括 Nist P256、Schnorr 签名、国密算法等多个不同的密码学插件，并支持代码和二进制产出的独立发布。

当然，目前插件机制是基于 go plugin 的实现，限于 go plugin 本身实现上的一些局限性，插件机制也具有如下需要改进的地方：

- **跨平台支持**：目前尚不支持 Windows 系统的插件化，只支持 Mac/Linux 系统。
- **依赖版本限制**：插件的依赖库版本和框架的依赖库版本不能有任何的差别，否则会加载失败。

相信在后续超级链迭代过程中，上述问题也会得到解决。

CHAPTER 20

超级链共识框架

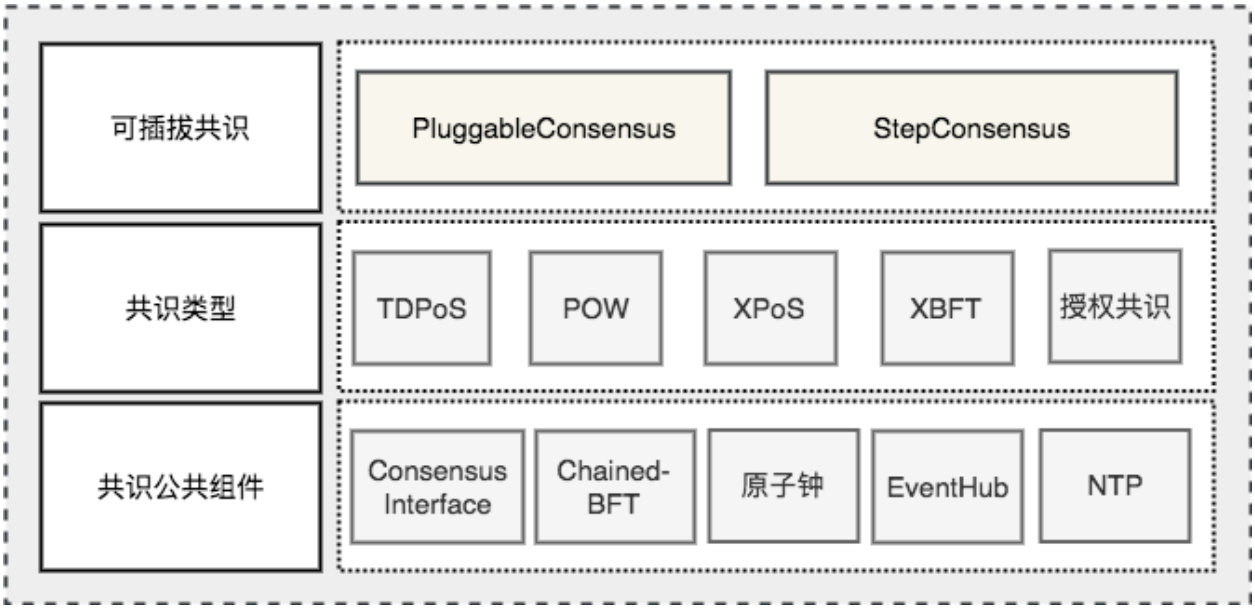
20.1 区块链共识机制概述

区块链系统多数采用去中心化的分布式设计，节点是分散在各处，系统需要设计一套完善的制度，以维护系统的执行顺序与公平性，统一区块链的版本，并奖励提供资源维护区块链的使用者，以及惩罚恶意的危害者。这样的制度，必须依赖某种方式来证明，是由谁取得了一个区块链的打包权 (或称记帐权)，并且可以获取打包这一个区块的奖励; 又或者是谁意图进行危害，就会获得一定的惩罚，这些都是区块链系统的共识机制需要解决的问题。

随着区块链应用落地场景越来越多，很多适应不同应用场景的共识算法先后被提出。但是在当前的技术背景下，功能过于全面的共识算法无法真正可用。在新一代区块链共识机制的设计过程中，根据实际应用场景，有的放矢的选择去中心化、节能、安全等设计原则，对一些原则支持强弱进行取舍，将一定程度上提升系统的整体运行效率。

我们超级链设计上是一个通用的区块链框架，用户可以方便地进行二次开发定制。超级链的共识模块设计上是一个能够复用底层共识安全的共识框架，用户基于这样的框架可以轻松地定义其自己的链，而不需要考虑底层的共识安全和网络安全。

20.2 超级链共识框架概览



上图是超级链共识模块的整体架构图，自底向上主要包括 3 层：

1. **共识公共组件层**：该层主要是不同共识可以共用的组件，包括共识公共节点 Consensus、Interface、Chained-BFT、GPS 原子钟等，它可以为链提供底层的共识安全性保障；

2. **共识类型层**：中间层为超级链以及支持或者即将支持的共识类型，主要包括 TDPoS、Pow、授权共识等，基于底层的共识安全能力。在这一层，用户可以定义有自己特色的共识类型，如类似 TDPoS 这种选举机制的共识，也可以定义 Stakeing 等的相关逻辑；

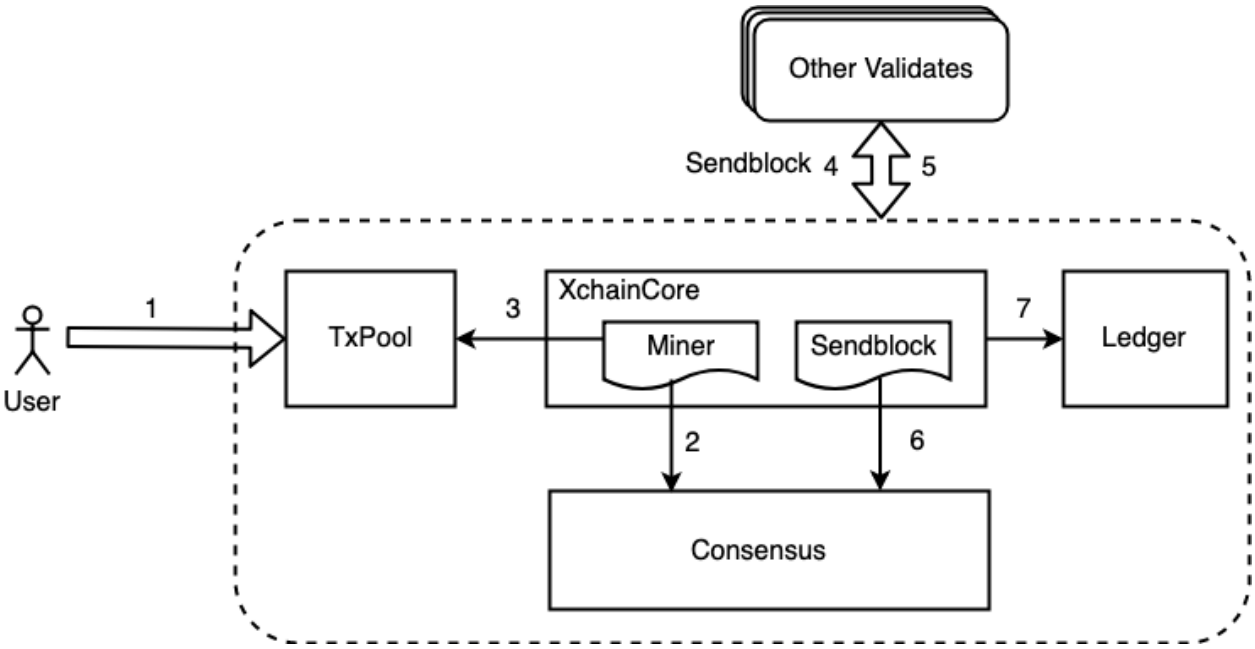
3. **可插拔共识层**：最上层是可插拔共识的运行态，包括 Step Consensus 和 Pluggable Consensus 两个实例，该层主要负责维护了链从创建到当前高度的共识的所有升级历史。超级链的共识升级主要依赖系统的提案和投票机制，详情请查看 提案和投票机制文档

20.3 超级链共识主流程

超级链的是一个多链架构，其中单个链的主要逻辑在 `core/xchaincore.go` 文件中，其中与共识模块交互的函数主要有 2 个，分别是 `Miner()` 和 `SendBlock()` ：

1. Miner(): 这个函数的主要功能有 2 点，首先判断自己是否为当前的矿工，当判断自己是矿工时需要进行区块打包。

2. SendBlock(): 这个函数是节点收到区块时的核心处理逻辑，当节点收到一个区块时会调用共识模块的相关接口进行区块有效性的验证，当验证通过后会将区块写入到账本中。



超级链的共识整体流程如上图所示，主要包括 7 个步骤：

1. 用户提交交易到网络，交易执行完后会进入未确认状态，并记录在交易的未确认列表中 TxPool 中；
2. 节点的 Miner 流程通过访问 Consensus 模块判断自己是否为当前的矿工；
3. 当节点判断自己是矿工时需要从 TxPool 中拉取交易并进行区块的打包；
4. 当矿工完成打包后会将区块广播给其他的验证节点，同时会通过步骤 7 将区块写入到账本；
5. 如果某个时刻其他节点判断自己是矿工，同样地会按照上述 1-5 流程进行区块打包，打包完后会将区块广播给该节点；
6. 节点收到区块后，会调用 consensus 模块进行区块的有效性验证；
7. 矿工打包完后或者验证节点收到一个有效的区块后，将区块写入账本；

20.4 接口介绍

整个共识框架主要有 2 套接口，分别是共识基础接口和共识安全接口，适用的场景不同。

场景一：用户希望定义自己的共识功能并独立负责共识安全；那么用户仅需要实现共识基础接口；场景二：用户希望定义自己的共识功能，但是希望框架底层能帮助保证共识安全；那么用户需要实现共识基础接口和共识安全接口；

20.4.1 共识基础接口

共识基础接口是共识模块的核心接口，是与 core 模块交互的主要部分。其中最核心的部分主要是 `CompeteMaster` 和 `CheckMinerMatch` 两个。`CompeteMaster` 是一个节点判断自己是否为主的主要逻辑，

CheckMinerMatch 是节点收到一个区块验证其区块有效性的主要逻辑。

```
// consensus/base/consensusinterface.go
type ConsensusInterface interface {
    Type() string
    Version() int64
    InitCurrent(block *pb.InternalBlock) error
    Configure(xlog log.Logger, cfg *config.NodeConfig, consCfg map[string]interface{},
        extParams map[string]interface{}) error
    CompeteMaster(height int64) (bool, bool)
    CheckMinerMatch(header *pb.Header, in *pb.InternalBlock) (bool, error)
    ProcessBeforeMiner(timestamp int64) (map[string]interface{}, bool)
    ProcessConfirmBlock(block *pb.InternalBlock) error
    GetCoreMiners() []*MinerInfo
    GetStatus() *ConsensusStatus
}
```

20.4.2 共识安全接口

共识安全接口是保证底层共识安全的核心接口，共识框架底层支持了 Hotstuff 算法的的高性能的共识安全模块 Chained-BFT 。暴露出了 PacemakerInterface 和 ExternalInterface 接口，其中 PacemakerInterface 是 Chained-BFT 的活性保证，此外为了扩展 Chained-BFT 安全模块能够应用于更多的仲裁类型，底层 Chained-BFT 设计上不需要理解仲裁的具体内容，通过 ExternalInterface 会与外层的共识进行通信，接口的具体定义如下，更详细的内容可以参见 Chained-BFT 的介绍。

```
// consensus/common/chainedbft/liveness/pacemaker_interface.go
// PacemakerInterface is the interface of Pacemaker. It responsible for generating a new␣
↪round.
// We assume Pacemaker in all correct replicas will have synchronized leadership after␣
↪GST.
// Safty is entirely decoupled from liveness by any potential instantiation of Packmaker.
// Different consensus have different pacemaker implement
type PacemakerInterface interface {
    // NextNewView sends new view msg to next leader
    // It used while leader changed.
    NextNewView(viewNum int64, proposer, preProposer string) error
    // NextNewProposal generate new proposal directly while the leader haven't changed.
    NextNewProposal(proposalID []byte, data interface{}) error
    // UpdateQCHigh update QuorumCert high of this node.
    //UpdateQCHigh() error
    // CurretQCHigh return current QuorumCert high of this node.
```

(continues on next page)

(continued from previous page)

```
    CurrentQCHigh(proposalID []byte) (*pb.QuorumCert, error)
    // CurrentView return current vie of this node.
    CurrentView() int64
    // UpdateValidatorSet update the validator set of BFT
    UpdateValidatorSet(validators []*cons_base.CandidateInfo) error
}
// consensus/common/chainedbft/external/external_interface.go
// ExternalInterface is the interface that chainedbft can communicate with external interface
// external consensus need to implements this.
type ExternalInterface interface {
    // CallPreQc call external consensus for the PreQc with the given Qc
    //  PreQc is the the given QC's ProposalMsg's JustifyQC
    CallPreQc(*pb.QuorumCert) (*pb.QuorumCert, error)
    // CallProposalMsg call external consensus for the marshal format of proposalMsg's parent block
    CallPreProposalMsg([]byte) ([]byte, error)
    // CallPrePreProposalMsg call external consensus for the marshal format of proposalMsg's grandpa's block
    CallPrePreProposalMsg([]byte) ([]byte, error)
    // CallVerifyQc call external consensus for proposalMsg verify with the given QC
    CallVerifyQc(*pb.QuorumCert) (bool, error)
    // CallProposalMsgWithProposalID call  external consensus for proposalMsg  with the given ProposalID
    CallProposalMsgWithProposalID([]byte) ([]byte, error)
    // IsFirstProposal return true if current proposal is the first proposal of bft
    // First proposal could have empty or nil PreQC
    IsFirstProposal(*pb.QuorumCert) (bool, error)
}
```

CHAPTER 21

Chained-BFT 共识公共组件

21.1 概述

在 超级链共识框架 一文中介绍了超级链底层有一个共识的公共组件叫 chained-bft，其是 Hotstuff 算法的实现。HotStuff 是一种简洁而优雅的 bft 改进算法。它具有以下优点：

- 它的设计中将 liveness 和 safty 解耦开来，使得非常方便与其他的共识进行扩展；
- 将 bft 过程拆解成 3 阶段，每个阶段都是 o(n) 的通信；
- 它允许一个节点处于不同的 view，并且将 view 的切换与区块结合起来，使得其能够实现异步共识，进一步提升共识的效率。

这样一个 chained-bft 可以在给定主集合的场景下确保网络的共识安全性，并且通过与外层共识配合工作实现共识的活性保证。

21.2 核心数据结构

```
enum QCState {
    NEW_VIEW = 0;
    PREPARE = 1;
    PRE_COMMIT = 2;
    COMMIT = 3;
    DECIDE = 4;
```

(continues on next page)

(continued from previous page)

```
}
// QuorumCert is a data type that combines a collection of signatures from replicas.
message QuorumCert {
    // The id of block this QC certified.
    bytes BlockId = 1;
    // The current type of this QC certified.
    // the type contains `NEW_VIEW`, `PREPARE`,`PRE_COMMIT`, `COMMIT`, `DECIDE`.
    State Type = 2;
    // The view number of this QC certified.
    int64 ViewNumber = 3;
    // SignInfos is the signs of the leader gathered from replicas
    // of a specifically certType.
    QCSignInfos SignInfos  = 4;
}
// QCSignInfos is the signs of the leader gathered from replicas of a specifically certType.
// A slice of signs is used at present.
// TODO zq: It will be change to Threshold-Signatures
// after Crypto lib support Threshold-Signatures.
message QCSignInfos {
    // QCSignInfos
    map<string, SignInfo> QCSignInfos = 1;
}
// SignInfo is the signature information of the
message SignInfo {
string Address = 1;
string PublicKey = 2;
bytes  Sign = 3;
}
// ChainedBftMessage is the message of the protocal
// In hotstuff, there are two kinds of messages, "NEW_VIEW_Message" and "QC_Message".
// In XuperChain, there is only one kind of message, "NEW_VIEW. The "QC_Message" is resuded with "BroadcastBlock" message.
message ChainedBftMessage {
    // Message Type
    QCState Type = 1;
    // Justify is the QC of the leader gathered, send to next leader.
    QuorumCert Justify = 2;
}
// ChainedBftMessage is the vote message of
```

(continues on next page)

(continued from previous page)

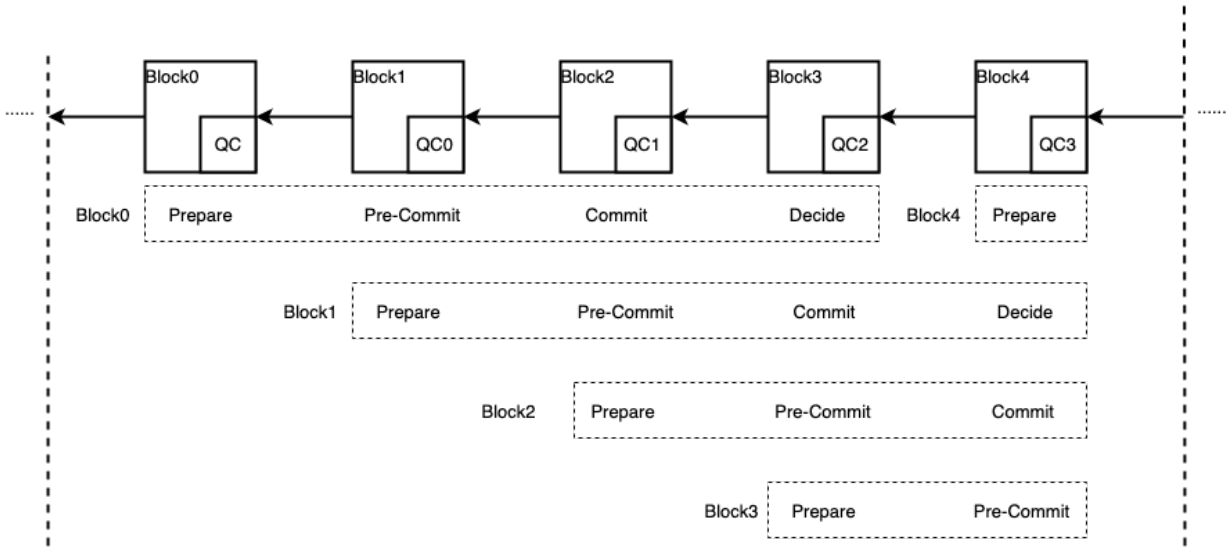
```
message ChainedBftVoteMessage {
    // The id of block this message vote for.
    bytes BlockId = 1;
    // Replica will sign the QCMessage if the QuorumCert if valid.
    SignInfo signature = 2;
}
```

整个 chained-bft 中主要包括三部分，分别是状态机 `Smr` 、`SafetyRules` 和 `PacemakerInterface` 。

21.3 Smr

Smr 是 `chained-bft` 的核心实例。他的主要的作用有以下几点：

1. 维护节点链的 chained-bft 共识状态机；
2. 在外层共识的驱动下发起 `NewView` 和 `NewProposal` 等消息并更新本地状态；
3. 处理其他验证节点的消息并更新本地状态；



21.4 Safety Rule

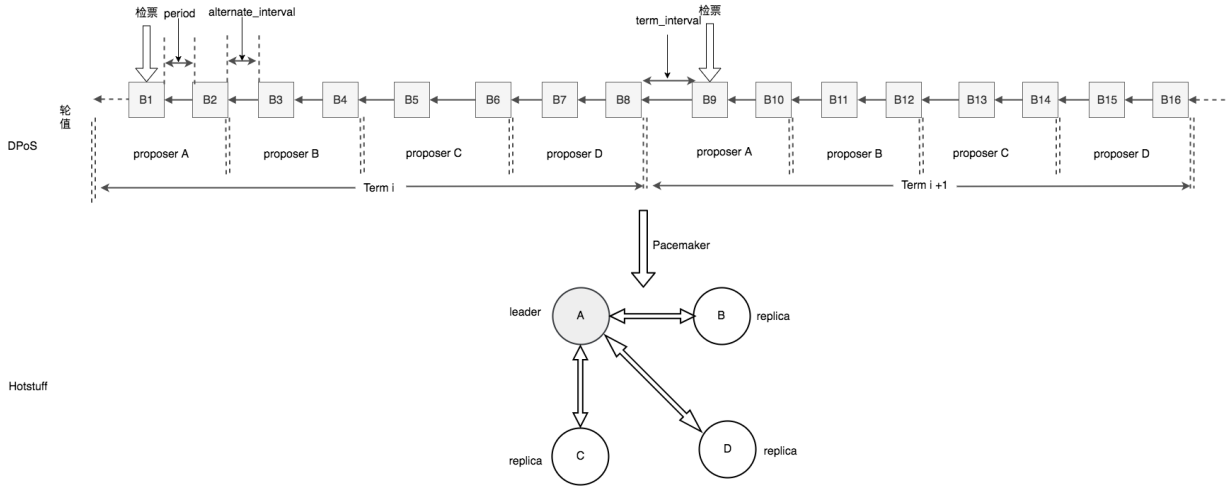
Safety Rule 是一个验证节点是否要接受一个新的 Proposal 的安全性规则，主要有三条：

1. 判断当前 Proposal 的 View 值是否大于本地 locked Proposal 的 View 值；
2. 验证当前 Proposal 中上一个 Proposal 的投票信息有效性和投票个数是否大于系统矿工数目的 2/3；
3. 验证当前 Proposal 的 ProposalMsg 是否有效；

当一个验证节点收到一个新的提案时，如果满足上述 `Safety Rule` 的认证，则会给这个提案进行投票，否则拒绝这次提案。

21.5 PacemakerInterface

Hotstuff 算法的一大特点就是将共识的 liveness 和 safety 分开。PacemakerInterface 是 Hotstuff 算法 Pacemaker 的接口定义，外层共识通过实现这些接口，可以推进内层共识的状态轮转。不同的外层共识可以有不同的实现。目前超级链已经实现了 DPoS+Hotstuff，具体的方案如下所示：



CHAPTER 22

TDPoS 共识

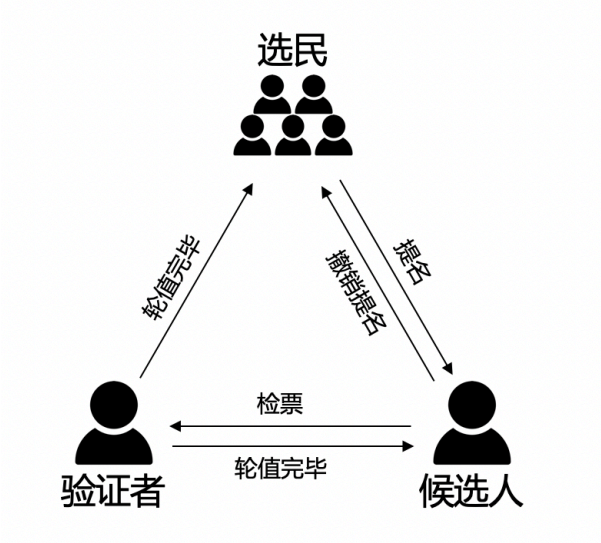
22.1 介绍

TDPoS 是超级链的一种改进型的 DPoS 算法，他是在一段预设的时间长度（一轮区块生产周期）内选择若干个验证节点，同时将这样一轮区块生产周期分为 N 个时间段，这若干个候选节点按照约定的时间段协议协同挖矿的一种算法。在选定验证节点集合后，TDPoS 通过 Chained-BFT 算法来保证轮值期间的安全性。总结一下，整个 TDPoS 主要包括 2 大阶段：

1. 验证人选举：通过 pos 相关选举规则选出一个验证者集合；
2. 验证人轮值：验证者集合按照约定的协议规则进行区块生产；

22.1.1 候选人选举

节点角色



在 TDPoS 中，网络中的节点有三种角色，分别是“普通选民”、“候选人”、“验证者”：

1. 选民：所有节点拥有选民的角色，可以对候选节点进行投票；
2. 候选人：需要参与验证人竞选的节点通过注册机制成为候选人，通过注销机制退出验证人竞选；
3. 验证人：每轮第一个节点进行检票，检票最高的 topK 候选人集合成为该轮的验证人，被选举出的每一轮区块生产周期的验证者集合，负责该轮区块的生产和验证，某个时间片内，会有一个矿工进行区块打包，其余的节点会对该区块进行验证。

网络中的三种角色之间是可以相互转换的，转换规则如下：

1. 所有地址都具有选民的特性，可以对候选人进行投票；
2. 选民经过“候选人提名”提名接口成为候选人，参与竞选；
3. 候选人经过“候选人退选”注销接口退出竞选；
4. 候选人经过检票产出验证者，得票 topK 的候选人当选验证者；
5. 验证者轮值完恢复候选人或者选民角色；

提名规则

节点想要参与竞选，需要先被提名为候选人，只有被提名的地址才能接受投票。为了收敛候选人集合，并一定程度上增加候选人参与的门槛，提名为候选人会有很多规则，主要有以下几点：

1. 提名候选人需要冻结燃料，并且金额不小于系统总金额的十万分之一；
2. 该燃料会被一直冻结，直到节点退出竞选；
3. 提名支持自提和他提，即允许第三方节点对候选人进行提名；
4. 被提名者需要知晓自己被提名，需要对提名交易进行背书；

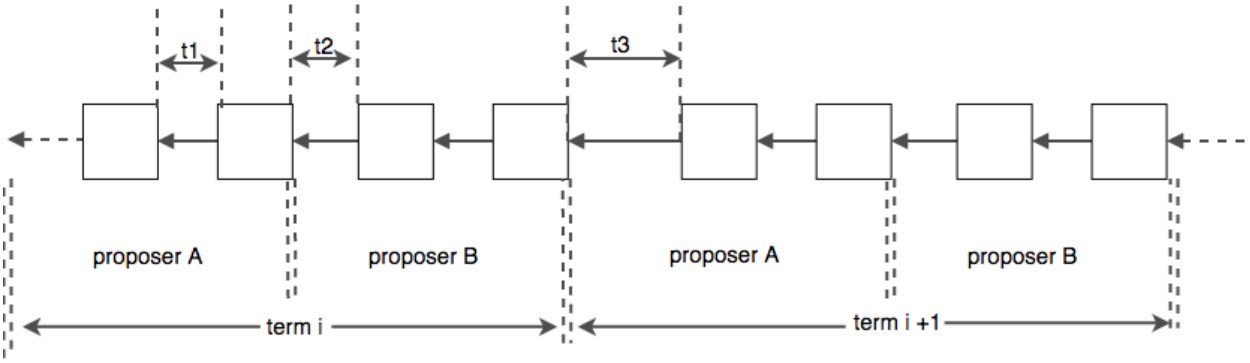
选举规则

候选人被提名后，会形成一个候选人池子，投票需要针对该池子内部的节点进行。TDPoS 的投票也有很多规则，主要有以下几点：

1. 任何地址都可以进行投票，投票需要冻结燃料，投票的票数取决于共识配置中每一票的金额，票数 = 冻结金额 / 投票单价；
2. 该燃料会被一直冻结，直到该投票被撤销；
3. 投票采用博尔达计分法，支持一票多投，每一票最多投给设置的验证者个数，每一票中投给不同候选人的票数相同；

22.1.2 候选人轮值

每一轮开始的第一个区块会自动触发检票的交易，该交易会进行下一轮候选人的检票，被选举出的节点会按照既定的时间片协同出块，每一个区块都会请求所有验证节点的验证。TDPoS 的时间片切分如下图所示：



为了降低切主时容易造成分叉，TDPoS 将出块间隔分成了 3 个，如上图所示：

- **t1**：同一轮内同一个矿工的出块间隔；
- **t2**：同一轮内切换矿工时的出块间隔，需要为 t1 的整数倍；
- **t3**：不同轮间切换时的出块间隔，需要为 t1 的整数倍；

拜占庭容错

TDPoS 验证节点轮值过程中，采取了 Chained-Bft 防止矿工节点的作恶。

22.1.3 技术细节

TDPoS 实现主要在 `consensus/tdpos` 路径下，其主要是通过智能合约的方式实现的，主要有以下几个合约方法：

```
voteMethod = "vote"
// 候选人投票撤销
revokeVoteMethod = "revoke_vote"
// 候选人提名
```

(continues on next page)

(continued from previous page)

```
nominateCandidateMethod = "nominate_candidate"
// 候选人罢黜
revokeCandidateMethod = "revoke_candidate"
// 验证人生成
checkvValidaterMethod = "check_validater"
```

核心接口如下：

```
func (tp *TDpos) runVote(desc *contract.TxDesc, block *pb.InternalBlock) error {
    // ......
    return nil
}
func (tp *TDpos) runRevokeVote(desc *contract.TxDesc, block *pb.InternalBlock) error {
    // ......
    return nil
}
func (tp *TDpos) runNominateCandidate(desc *contract.TxDesc, block *pb.InternalBlock)␣
↪error {
    // ......
    return nil
}
func (tp *TDpos) runRevokeCandidate(desc *contract.TxDesc, block *pb.InternalBlock)␣
↪error {
    // ......
    return nil
}
func (tp *TDpos) runCheckValidater(desc *contract.TxDesc, block *pb.InternalBlock) error
↪{
    // ......
    return nil
}
```

CHAPTER 23

Single 及 PoW 共识

23.1 介绍

Single 以及 PoW 属于不同类型的区块链共识算法。其中，PoW(Proof Of Work，工作量证明) 是通过解决一到特定的问题从而达成共识的区块链共识算法；而 Single 亦称为授权共识，在一个区块链网络中授权固定的 address 来记账本。Single 一般在测试环境中使用，不适合大规模的应用环境。PoW 适用于公有链应用场景。

23.2 算法流程

Single 共识

- 对于矿工：Single 是固定 address 周期性出块，因此在调用 CompeteMaster 的时候主要判断当前时间与上一次出块时间间隔是否达到一个周期；
- 对于验证节点：验证节点除了密码学方面必要的验证之外，还会验证矿工与本地记录的矿工是否一致；

Pow 共识

- 对于矿工：每次调用 CompeteMaster 都返回 true，表明每次调用 CompeteMaster 的结果都是矿工该出块了；
- 对于验证节点：验证节点除了密码学方面必要的验证之外，还会验证区块的难度值是否符合要求；

23.3 在超级链中使用 Single 或 PoW 共识

只需修改 data/config 中的创世块配置即可指定使用共识

23.3.1 使用 Single 共识的创世块配置

```
{
    "version" : "1",
    "consensus" : {
        # 共识算法类型
        "type"  : "single",
        # 指定出块的 address
        "miner" : "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN"
    },
    # 预分配
    "predistribution":[
        {
            "address" : "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
            "quota" : "100000000000000000000"
        }
    ],
    # 区块大小限制
    "maxblocksize" : "128",
    # 出块周期
    "period" : "3000",
    # 出块奖励
    "award" : "428100000000",
    # 精度
    "decimals" : "8",
    # 出块奖励衰减系数
    "award_decay": {
        "height_gap": 31536000,
        "ratio": 1
    },
    # 系统权限相关配置
    "permission": {
        "CreateAccount" : { "rule" : "NULL", "acl": {}},
        "SetAccountAcl": { "rule" : "NULL", "acl": {}},
        "SetContractMethodAcl": { "rule" : "NULL", "acl": {}}
    }
```

(continues on next page)

(continued from previous page)

```
}
```

23.3.2 使用 PoW 共识的创世块配置

```
{
    "version" : "1",
    # 预分配
    "predistribution":[
        {
            "address" : "Y4TmpfV4pvhYT5W17J7TqHSLo6cqq23x3",
            "quota" : "1000000000000000"
        }
    ],
    "maxblocksize" : "128",
    "award" : "1000000",
    "decimals" : "8",
    "award_decay": {
        "height_gap": 31536000,
        "ratio": 0.5
    },
    "genesis_consensus":{
        "name": "pow",
        "config": {
                # 默认难度值
                "defaultTarget": "19",
                # 每隔 10 个区块做一次难度调整
                "adjustHeightGap": "10",
                "expectedPeriod": "15",
                "maxTarget": "22"
        }
    }
}
```

23.4 关键技术

Single 共识的原理简单，不再赘述。

PoW 共识

解决一道难题过程，执行流程如下：

- **step1** 每隔一个周期判断是否接收到新的区块。若是，跳出解决难题流程，若不是，进行 **step2** ；
- **step2** 判断当前计算难度值是否符合要求。若是，跳出难题解决流程，若不是难度值加 1，继续 **step1** ；

伪代码如下：

```
// 在每次挖矿时，设置为 true
// StartPowMinning
for {
    // 每隔 round 次数，判断是否接收到新的区块，避免与网络其他节点不同步
    if gussCount % round == 0 && !l.IsEnablePowMinning() {
        break
    }
    // 判断当前计算难度值是否符合要求
    if valid = IsProofed(block.Blockid, targetBits); !valid {
        guessNonce += 1
        block.Nonce = guessNonce
        block.Blockid, err = MakeBlockID(block)
        if err != nil {
            return nil, err
        }
        guessCount++
        continue
    }
    break
}
// valid 为 false 说明还没挖到块
// l.IsEnablePowMinning() == true  --> 自己挖出块
// l.IsEnablePowMinning() == false --> 被中断
if !valid && !l.IsEnablePowMinning() {
    l.xlog.Debug("I have been interrupted from a remote node, because it has a higher␣
↪block")
    return nil, ErrMinerInterrupt
}
```

计算当前区块难度值过程，执行流程如下：

- **step1** 判断当前区块所在高度是否比较小。若是，直接复用默认的难度值，跳出计算区块难度值过程，若不是，继续 **step2** ；
- **step2** 获取当前区块的前一个区块的难度值；
- **step3** 判断当前区块是否在下一个难度调整周期范围内。若是，继续 **step4** ；若不是，继续 **step5** ；

- **step4** 获取当前区块的前一个区块的难度值，并计算经历 N 个区块，预期/实际消耗的时间，并根据公式调整难度值，跳出计算区块难度值过程；
- **step5** 如果当前区块所在高度在下一次区块难度调整的周期范围内，直接复用前一个区块的难度值，跳出计算区块难度值过程；

伪代码如下：

```
func (pc *PowConsensus) calDifficulty(curBlock *pb.InternalBlock) int32 {
    // 如果当前区块所在高度比较小，直接复用默认的难度值
    if curBlock.Height <= int64(pc.config.adjustHeightGap) {
        return pc.config.defaultTarget
    }
    height := curBlock.Height
    preBlock, err := pc.getPrevBlock(curBlock, 1)
    if err != nil {
        pc.log.Warn("query prev block failed", "err", err, "height", height-1)
        return pc.config.defaultTarget
    }
    // 获取当前区块前一个区块的难度值
    prevTargetBits := pc.getTargetBitsFromBlock(preBlock)
    // 如果当前区块所在高度恰好是难度值调整所在的高度周期
    if height%int64(pc.config.adjustHeightGap) == 0 {
        farBlock, err := pc.getPrevBlock(curBlock, pc.config.adjustHeightGap)
        if err != nil {
            pc.log.Warn("query far block failed", "err", err, "height", height-int64(pc.
↪config.adjustHeightGap))
            return pc.config.defaultTarget
        }
        // 经历 N 个区块，预期消耗的时间
        expectedTimeSpan := pc.config.expectedPeriod * (pc.config.adjustHeightGap - 1)
        // 经历 N 个区块，实际消耗的时间
        actualTimeSpan := int32((preBlock.Timestamp - farBlock.Timestamp) / 1e9)
        pc.log.Info("timespan diff", "expectedTimeSpan", expectedTimeSpan,
↪"actualTimeSpan", actualTimeSpan)
        //at most adjust two bits, left or right direction
        // 避免难度值调整太快，防止恶意攻击
        if actualTimeSpan < expectedTimeSpan/4 {
            actualTimeSpan = expectedTimeSpan / 4
        }
        if actualTimeSpan > expectedTimeSpan*4 {
            actualTimeSpan = expectedTimeSpan * 4
```

(continues on next page)

(continued from previous page)

```
        }
        difficulty := big.NewInt(1)
        difficulty.Lsh(difficulty, uint(prevTargetBits))
        difficulty.Mul(difficulty, big.NewInt(int64(expectedTimeSpan)))
        difficulty.Div(difficulty, big.NewInt(int64(actualTimeSpan)))
        newTargetBits := int32(difficulty.BitLen() - 1)
        if newTargetBits > pc.config.maxTarget {
            pc.log.Info("retarget", "newTargetBits", newTargetBits)
            newTargetBits = pc.config.maxTarget
        }
        pc.log.Info("adjust targetBits", "height", height, "targetBits", newTargetBits,
↪"prevTargetBits", prevTargetBits)
        return newTargetBits
    } else {
        // 如果当前区块所在高度在下一次区块难度调整的周期范围内，直接复用前一个区块的难度值
        pc.log.Info("prev targetBits", "prevTargetBits", prevTargetBits)
        return prevTargetBits
    }
}
```

CHAPTER 24

超级链监管机制

24.1 监管机制概述

超级链是一个具备政府监管能力的区块链系统。在设计上我们需要充分考虑监管和安全问题，做到安全可控。基于此我们超级链底层设计了一个监管合约的机制，通过该机制，超级链具备了对链上用户的实名、交易的安全检查等监管能力。

超级链在初始化时候，可以通过创世块配置的方式，配置这条链是否需要支持监管类型。对于配置了监管合约的链，这个链上所有的事务发起，无论是转账还是合约调用，系统会默认插入监管合约的执行，执行结果体现在读写集中，执行过程不消耗用户资源，执行结果所有节点可验证。

目前超级链支持的监管合约主要有以下几个：

- 实名制合约: identity
- DApp 封禁合约: banned
- 合规性检查合约: complianceCheck
- 交易封禁合约: forbidden

下面将会以实名合约为例对监管合约的使用步骤进行说明

24.2 监管机制使用说明

24.2.1 创世块配置

创世块配置新增 reserved_contracts 配置，内容如下：

```
"reserved_contracts": [
    {
        "module_name": "wasm",
        "contract_name": "identity",
        "method_name": "verify",
        "args":{}
    }
]
```

这个配置中配置了 identity 监管合约。

24.2.2 搭建网络

搭建网络的方式与以前的方式没有区别，用户可以依据需求选择搭建单节点网络还是多节点网络。

搭建网络参见如下链接：单节点网络搭建 多节点网络搭建

24.2.3 部署 Reserved 合约

1. 编译实名合约

实名合约代码路径如下：core/contractsdk/cpp/reserved/identity.cc

实名合约实名的对象是一个具体的 ak。

```
cd ./contractsdk/cpp
cp reserved/identity.cc example
./build.sh
```

编译好的产出为./build 文件夹下的 identity.wasm 文件。

2. 创建合约账户

在 XuperChain 中所有的合约都是部署在具体的某个账户下的，所以，为了部署实名合约，我们需要首先创建一个合约账户，注意，这里账户的拥有者可以修改其内合约 Method 的 ACL 权限管理策略，通过这种机制实现对谁可以添加实名状态和删除实名状态的控制。这里是由超级链的 多节点网络搭建 支持的。

```
# 快速创建合约方式:
./xchain-cli account new --account 1111111111111111
```

3. 部署实名合约

部署合约需要消耗资源，所以先给上述合约账户转移一笔资源，然后在合约内部署上面的合约:

```
# 1  转移资源
./xchain-cli transfer --to XC1111111111111111@xuper --amount 100000
# 2 部署实名合约
# 通过 -a 的 creator 参数，可以初始化被实名的 ak。
./xchain-cli wasm deploy --account XC1111111111111111@xuper --cname identity -H␣
↪localhost:37101 identity.wasm -a '{"creator":"addr1"}'
```

Note: 上述实名合约初始化的被实名的 address 需要和实名合约添加实名信息保持相同，否则会由于初始实名的 ak 和添加实名权控不一致而导致系统无法添加新的实名状态。

24.2.4 Reserved 合约调用

实名合约部署完成后，就可以进行实名合约信息的添加和删除了

1. 添加实名信息

合约调用 json 文件如下:

```
{
    "module_name": "wasm",
    "contract_name": "identity",
    "method_name": "register_aks",
    "args":{
        "aks":"ak1,ak2"
    }
}
```

具体步骤如下:

```
# 1: 生成原始交易
./xchain-cli multisig gen --desc identity_add.json --host localhost:37101 --fee 1000 --
↪output tx_add.out
# 2: 本地签名
./xchain-cli multisig sign --output tx_add_my.sign --tx tx_add.out
```

(continues on next page)

(continued from previous page)

```
# 3: 交易发送
./xchain-cli multisig send tx_add_my.sign --host localhost:37101 --tx tx_add.out
```

2. 删除实名信息

合约调用 json 文件如下:

```
{
    "module_name": "wasm",
    "contract_name": "identity",
    "method_name": "unregister_aks",
    "args":{
        "aks":"ak1,ak2"
    }
}
```

具体步骤如下:

```
# 1: 生成原始交易
./xchain-cli multisig gen --desc identity_del.json --host localhost:37101 --fee 1000 --
↪output tx_del.out
# 2: 本地签名
./xchain-cli multisig sign --output tx_del_my.sign --tx tx_del.out
# 3: 交易发送
./xchain-cli multisig send tx_del_my.sign tx_del_compliance_sign.out --host␣
↪localhost:37101 --tx tx_del.out
```

3. 实名信息验证

当用户向网络发起事务请求时，网络会验证交易中的 `initiator` 和 `auth_require` 字段是否都经过实名，如果都经过实名，则通过，否则，失败。

CHAPTER 25

多盘散列

25.1 背景

区块链中的账本数据通常是只增不减，而单盘存储容量有上限。目前单盘最高容量是 14TB 左右，需要花费 4000 块钱；以太坊账本数据已经超过 1TB，即使是在区块大小上精打细算的比特币账本也有 0.5TB 左右。区块链账本数据不断增加，单盘容量上限成为区块链持续发展的天花板。目前对 leveldb 的多盘扩展方案，大部分是采用了多个 leveldb 实例的方式，也就是每个盘一个单独的 leveldb 实例。这种做法的好处是简单，不需要修改 leveldb 底层代码，缺点是牺牲了多行原子写入的功能。在区块链的应用场景中，我们是需要这种多个写入操作原子性的。所以选择了改 leveldb 底层模型的技术路线。

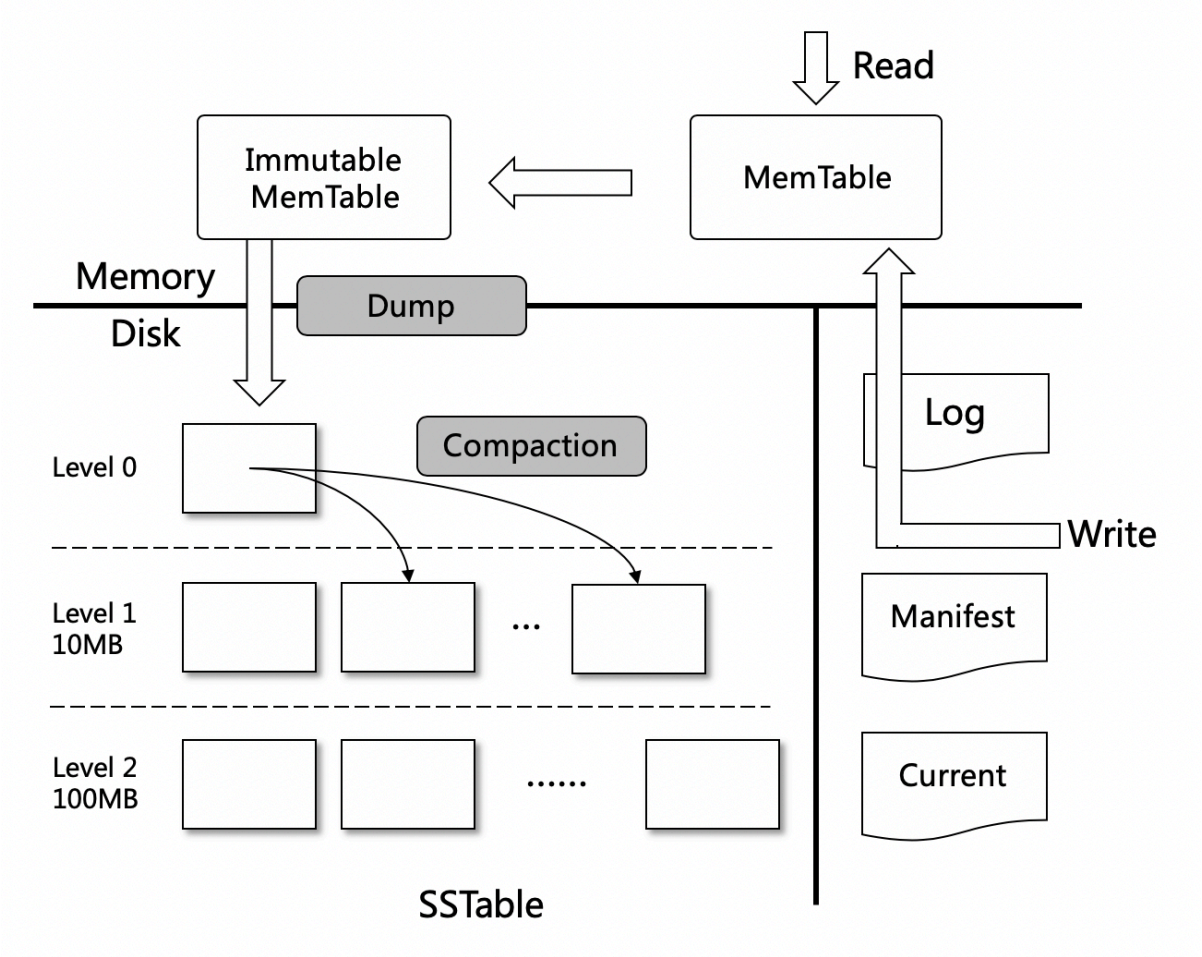
25.2 LevelDB 数据模型分析

- **Log 文件**：写 Memtable 前会先写 Log 文件，Log 通过 append 的方式顺序写入。Log 的存在使得机器宕机导致的内存数据丢失得以恢复；
- **Manifest 文件**：Manifest 文件中记录 SST 文件在不同 Level 的分布，单个 SST 文件的最大最小 key，以及其他一些 LevelDB 需要的元信息；
- **Current 文件**：LevelDB 启动时的首要任务就是找到当前的 Manifest，而 Manifest 可能有多个。Current 文件简单的记录了当前 Manifest 的文件名；

以上 3 种文件可以称之为元数据文件，它们占用的存储空间通常是几十 MB，最多不会超过 1GB

- **SST 文件**：磁盘数据存储文件。分为 Level 0 到 Level N 多层，每一层包含多个 SST 文件；单个 SST 文件容量随层次增加成倍增长；文件内数据有序；其中 Level0 的 SST 文件由 Immutable 直接 Dump

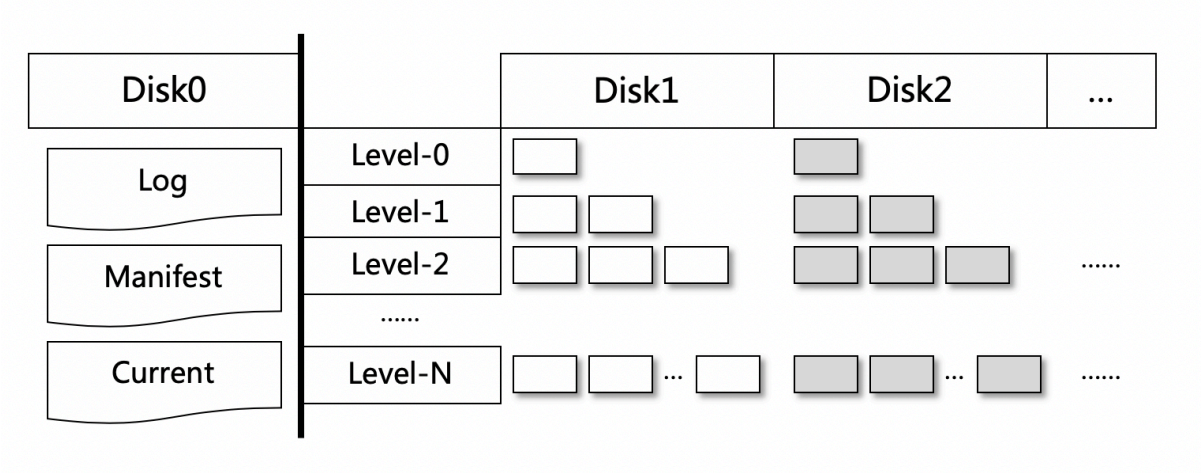
产生，其他 Level 的 SST 文件由其上一层的文件和本层文件归并产生；SST 文件在归并过程中顺序写生成，生成后仅可能在之后的归并中被删除，而不会有任何的修改操作。



25.3 核心改造点

Leveldb 的数据主要是存储在 SST(Sorted String Table) 文件中，写放大的产生就是由于 compact 的时候需要顺序读取 Level-N 中的 sst 文件，写出到 Level-N+1 的 sst 文件中。我们将 SST 文件分散在多块盘上存储，具体的方法是根据 sst 的编号做取模散列，取模的底数是盘的个数，理论上数据量和 IO 压力会均匀分散在多块盘上。

举个例子，假设某 sst 文件名是 12345.ldb，而节点机器有 3 块盘用于存储（/disk1, /disk2, /disk3），那么就将改 sst 文件放置在 (12345 % 3) + 1, 也就是 disk1 盘上



25.4 使用方式

leveldb.OpenFile 有两个参数，一个是 db 文件夹路径 path，一个是打开参数 Options；如果要使用多盘存储，调用者需要设置 Options.DataPaths 参数，它是一个 []string 数组，声明了各个盘的文件夹路径，可参考 配置多盘存储 。

25.5 扩容问题

假设本来是 N 块盘，扩容后是 (N+M) 块盘。对于已有的 sst 文件，因为取模的底数变了，可能会出现按照原有的取模散列不命中的情况。规则是：

- 对于读 Open，先按照 (N+M) 取模去 Open，如果不存在，则遍历各盘直到能 Open 到相应的文件，由于 Open 并不是频繁操作，代价可接受，且 SST 的编号是唯一且递增的，所以不存在读取脏数据的问题；
- 对于写 Open，就按照 (N+M) 取模，因为写 Open 一定是生成新的文件。

随着 Compact 的不断进行，整个数据文件的分布会越来越趋向于均匀分布在 (N+M) 个盘，扩容完成。

25.6 实验

写入测试可参考代码 kv/mstorage/test/test_write.go

读取测试可参考代码 kv/mstorage/test/test_read.go

CHAPTER 26

平行链与群组

26.1 背景

超级链具备平行链特性，能够实现业务的混部，确保整体架构性能上可以水平扩展；

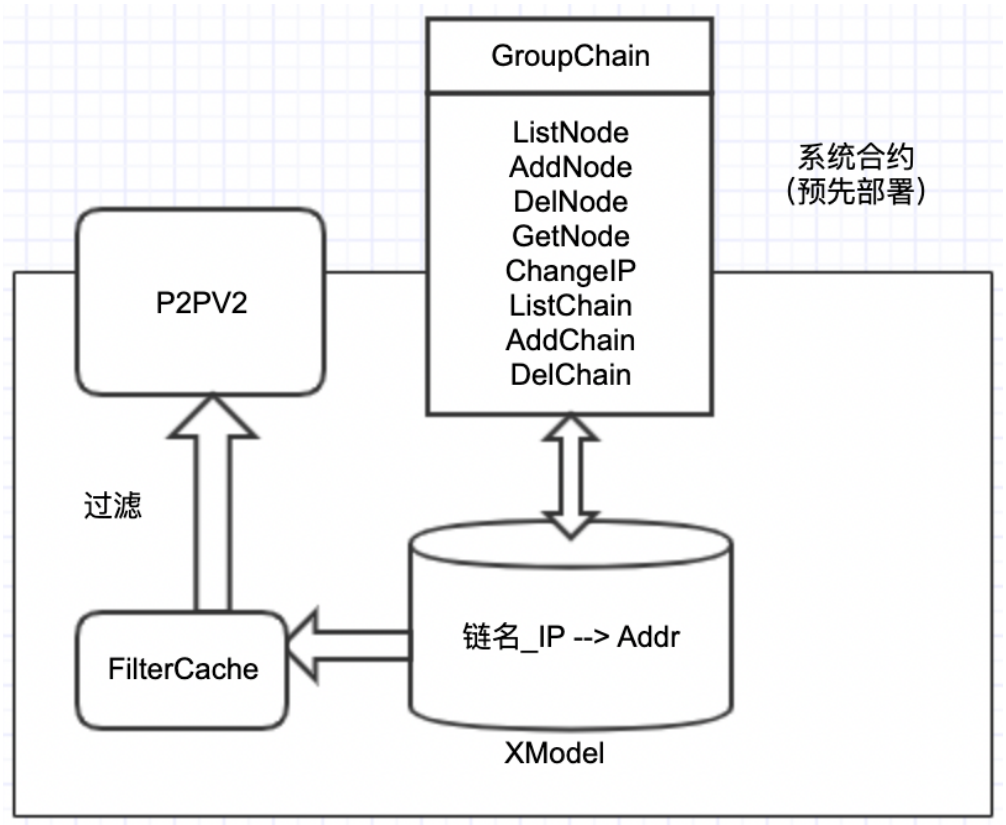
同时，平行链还具备群组特性，能够一定程度上实现平行链隐私数据的隔离，只有群组内的节点才能有这个平行链的数据。

26.2 术语

- **平行链**：相对于主链而言，运行在超级链中的用户级区块链实例，用户通过调用主链的智能合约创建。功能与主链无区别，全网节点均可以获取平行链账本数据，实现整体架构水平可扩展。
- **群组**：作用于平行链，具备群组特性的平行链，只有特定节点才拥有该平行链的账本数据。群组具备的特性包括私密性、动态性。

26.3 架构

平行链的群组特性通过白名单机制来实现，在网络层进行过滤。平行链的群组架构，如下图



26.4 设计思路

1. 如何要支持群组，需要在 xuper 链部署一个系统合约：GroupChain（一个网络有且仅有一个）

   这样是为了保证兼容性，如果没有部署这个 GroupChain 合约，那么行为和旧版本一致。

   < 平行链名字，IP> → Address

   为什么把 IP 放在 Key 中，是为了方便做过滤的时候查找更快，直接 Get。

   平行链名字作为前缀，方便列出这条链的所有合法成员节点。

   备注：此处 IP 是代指一个 TCP 协议定位符，可以是 libp2p 风格的 URL。

2. 查询特定的链是否具备群组关系

   Case1: 部分链希望有群组特性，即只有特定的节点才能同步账本数据；

   Case2: 剩下的链还是期望所有节点都参与同步、验证区块；

   基于以上两种场景，需要增加一层映射，即 < 平行链，是否支持群组 >

   如果每次转发都 Lookup 数据库过滤 IP，性能有影响，可以考虑在 p2p 中维护一个 Cache；

3. 通过这个智能合约接口，可以修改（address, IP）的映射关系

   合约的 Owner（GroupChain 这个合约的 Owner）可以添加或删除 address

节点也可以后期自己修改 IP（节点有权更换自己的 IP），合约里面会判断 Initiator() 字段和 address 是否一致，确保每个 address 只能修改自己的 IP

4. 平行链中转消息的时候，必须确保目的 IP 在智能合约的映射表中存在

   如果每次转发都 Lookup 数据库过滤 IP，性能有影响，可以考虑在 p2p 中维护一个 Cache；

CHAPTER 27

合约账号

27.1 访问控制列表（ACL）

如果把合约账号当作一家股份制公司，那么 ACL 便是公司股东投票的机制，ACL 可以规定合约账号背后各“股东”账号的权重，只有当“股东”签名的权重之和大于设定阈值时操作才会有效地进行。

超级链中 ACL 配置格式如下：

```
{
    "pm": {
        "rule": 1,              # rule=1 表示签名阈值策略，rule=2 表示 AKSet 签名策略
        "acceptValue": 0.6      # acceptValue 为签名需达到的阈值
    },
    "aksWeight": {              # aksWeight 里规定了每个地址对应账号签名的权重
        "AK1": 0.3,
        "AK2": 0.3
    }
}
```

了解了访问控制列表的概念，下面我们就来演示一下如何创建一个合约账号

27.2 合约账号创建

Xchain 的客户端工具提供了新建账号的功能，基本用法如下：

```
xchain-cli account new --desc account.des
```

这里的 account.des 就是创建账号所需要的配置了，内容如下：

```
{
    "module_name": "xkernel",
    "method_name": "NewAccount",
    "args" : {
        "account_name": "1111111111111111",  # 说明：账号名称是 16 位数字组成的字符串
        # acl 中的内容注意转义
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\": {\"AK1\": 0.3,
↪\"AK2\": 0.3}}"
    }
}
```

命令运行后就会调用 xchain 的系统合约功能 `NewAccount` 创建一个名为 `XC1111111111111111@xuper` （如果链名字为 xuper）的账号

除了上述方法，我们还提供了一个比较简易的方式来创建合约账号，命令如下：

```
xchain-cli account new --account 1111111111111111 # 16 位数字组成的字符串
```

上述命令也会创建一个名为 `XC1111111111111111@xuper` 的账号，由于我们没有制定 ACL 的具体内容，其 ACL 被赋值为默认状态，即背后有权限的账号只有当前节点上默认账号一个（地址默认位于 data/keys/address）

Note: 创建合约账号的操作需要提供手续费，需要按照命令行运行结果给出的数值，添加一个不小于它的费用（使用 –fee 参数）

27.3 合约账号基本操作

27.3.1 查询账号 ACL

XuperChain 的客户端工具提供了 ACL 查询功能，只需如下命令

```
xchain-cli acl query --account XC1111111111111111@xuper # account 参数为合约账号名称
```

27.3.2 查询账号余额

合约账号查询余额和普通账号类似，只是命令行的参数有些许变化

```
./xchain-cli account balance XC1111111111111111@xuper -H 127.0.0.1:37101
```

使用此命令即可查询 'XC1111111111111111@xuper'的余额

27.3.3 修改账号 ACL

修改 ACL 的配置和创建账号的配置类似

```
{
    "module_name": "xkernel",
    "method_name": "SetAccountAcl",  # 这里的方法有了变更
    "args" : {
        "account_name": "1111111111111111",
        # acl 字段为要修改成的新 ACL
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\": {\"AK3\": 0.3,\"AK4\": 0.3}}"
    }
}
```

修改 ACL 的操作，需要符合当前 ACL 中设置的规则，即需要具有足够权重的账号签名

我们首先生成一个多重签名的交易

```
./xchain-cli multisig gen --desc acl_new.json --from XC1111111111111111@xuper
```

这样就会生成一个默认为 'tx.out'的文件，之后使用原 ACL 中的账号对其进行签名

```
./xchain-cli multisig sign --keys data/account/AK1 --output AK1.sign
./xchain-cli multisig sign --keys data/account/AK2 --output AK2.sign
```

最后把生成的 'tx.out'发出去

```
./xchain-cli multisig send --tx tx.out AK1.sign,AK2.sign AK1.sign,AK2.sign
```

至此便完成了 ACL 的修改

CHAPTER 28

多节点部署

在阅读本节前，请先阅读“快速入门”，当中介绍了创建单节点网络的创建，在该基础上，搭建一个 SINGLE 共识的多节点网络，其他节点只要新增 p2p 网络 bootNodes 配置即可。如果你想搭建一个 TDPoS 共识的链，仅需要修改创世块参数中“genesis_consensus”配置参数即可。下面将详细介绍相关操作步骤。

28.1 p2p 网络配置

我们以搭建 3 个节点的网络为例来说明（其实搭建更多节点的原理是一致的），首先需要有一个节点作为“bootNode”，其他节点启动前都配置这个“bootNode”的地址即可实现

对于 bootNode 节点，我们需要先获取它的 netUrl，具体命令如下：

```
./xchain-cli netUrl get -H 127.0.0.1:37101
```

如果不是以默认配置启动的，我们需要先生成它的 netUrl，然后再获取

```
./xchain-cli netUrl gen -H 127.0.0.1:37101
```

如此我们会获得一个类似于 `/ip4/127.0.0.1/tcp/47101/p2p/QmVxeNubpg1ZQjQT8W5yZC9fD7ZB1ViArwvyGUB53sqf8e` 样式的返回

对其他的节点，我们需要修改其服务配置 *conf/xchain.yaml* 中 p2pv2 一节

```
p2pV2:
    // port 是节点 p2p 网络监听的默认端口，如果在一台机器上部署注意端口配置不要冲突，
```

(continues on next page)

(continued from previous page)

```
    // node1 配置的是 47101, node2 和 node3 可以分别设置为 47102 和 47103
    port: 47102
    // 节点加入网络所连接的种子节点的链接信息,
    bootNodes:
    - "/ip4/127.0.0.1/tcp/47101/p2p/QmVxeNubpg1ZQjQT8W5yZC9fD7ZB1ViArwvyGUB53sqf8e"
```

Note: 需要注意的是，如果节点分布在不同的机器之上，需要把 netUrl 中的本地 ip 改为机器的实际 ip

修改完配置后，即可在每一个节点使用相同配置创建链，然后分别启动 bootNode 和其他节点，即完成了多节点环境的部署

这里可以使用系统状态的命令检查环境是否正常

```
./xchain-cli status -H 127.0.0.1:37101
```

通过变更 -H 参数，查看每个节点的状态，若所有节点高度都是一致变化的，则证明环境部署成功

28.2 搭建 TDPoS 共识网络

XuperChain 系统支持可插拔共识，通过修改创世块的参数，可以创建一个以 TDPoS 为共识的链。

下面创世块配置（一般位于 core/data/config/xuper.json）和单节点创世块配置的区别在于创世共识参数 genesis_consensus 的 config 配置，各个配置参数详解配置说明如下所示：

```
{
    "version" : "1",
    "predistribution":[
        {
            "address" : "mahtKhdV5SZP4FveEBzX7j6FgUGfBS9om",
            "quota" : "100000000000000000000"
        }
    ],
    "maxblocksize" : "128",
    "award" : "1000000",
    "decimals" : "8",
    "award_decay": {
        "height_gap": 31536000,
        "ratio": 1
    },
    "genesis_consensus": {
```

(continues on next page)

(continued from previous page)

```
        "name": "tdpos",
        "config": {
            # tdpos 共识初始时间，声明 tdpos 共识的起始时间戳，建议设置为一个刚过去不旧的时间戳
            "timestamp": "1548123921000000000",
            # 每一轮选举出的矿工数，如果某一轮的投票不足以选出足够的矿工数则默认复用前一轮的矿工
            "proposer_num":"3",
            # 每个矿工连续出块的出块间隔
            "period":"3000",
            # 每一轮内切换矿工时的时间间隔，需要为 period 的整数倍
            "alternate_interval":"6000",
            # 切换轮时的出块间隔，即下一轮第一个矿工出第一个块距离上一轮矿工出最后一个块的时间间隔，需要为 period 的整数配
            "term_interval":"9000",
            # 每一轮内每个矿工轮值任期内连续出块的个数
            "block_num":"200",
            # 为被提名的候选人投票时，每一票单价，即一票等于多少 Xuper
            "vote_unit_price":"1",
            # 指定第一轮初始矿工，矿工个数需要符合 proposer_num 指定的个数，所指定的初始矿工需要在网络中存在，不然系统轮到该节点出块时会没有节点出块
            "init_proposer": {
                "1":["RU7Qv3CrecW5waKc1ZWYnEuTdJNjHc43u",
↪"XpQXiBNo1eHRQpD9UbzBisTPXojpyzkxn","SDCBba3GVYU7s2VYQVrhMGLet6bobNzbM"]
            }
        }
    }
}
```

修改完每个节点的创世块配置后，需要确认各节点的 data/blockchain 目录下内容为空。然后重新按照上一节的步骤，在各节点上创建链，启动所有节点，即完成 TDPoS 共识的环境部署。

28.3 选举 TDPOS 候选人

选举候选人包括提名和投票两个环节，具体操作和 发起提案 类似

28.3.1 提名候选人

首先需要准备一个提名的配置，json 格式

```
{
    "module": "tdpos",
    "method": "nominate_candidate",
    "args": {
        # 此字段为要提名的候选人的地址
        "candidate": "kJFcY3FjmNU8xk6cRzHvTPmChUQ3SBGVE",
        # 此字段为候选人节点的 netURL
        "neturl": "/ip4/10.0.4.6/tcp/47101/p2p/
↪QmRmdBSyHpKPvhsvmyys8f1jDM4x1S9cbCwZaBMqMKjwhV"
    }
}
```

然后将这个 json 文件（假定文件名为 nominate.json）通过多重签名命令发出。提名候选人的操作需要提名者和被提名候选人的两个签名（如果是自己提名自己，那么就只需要一个签名了）

首先要准备一个需收集签名的地址列表，可以参考 发起多重签名交易

```
YDYBchKWXpG7HSkHy4YoyzTJnd3hTFBgG
kJFcY3FjmNU8xk6cRzHvTPmChUQ3SBGVE
```

然后生成一个提名交易，超级链上进行候选人提名需要冻结大于链上资产总量的十万分之一的 utxo（当前的总资产可以通过 status 查询命令 查看结果的 utxoTotal 字段）

```
# 这里转账的目标地址可以任意，转给自己也可以，注意冻结参数为-1，表示永久冻结
./xchain-cli multisig gen --to=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --desc=nominate.json --
↪amount=10000000000000000 --frozen -1 -A addr_list --output nominate.tx
```

命令会生成交易内容，然后对其进行签名

```
# 提名者签名
./xchain-cli multisig sign --tx nominate.tx --output nominate.sign --keys path/to/
↪nominate
# 候选人签名
./xchain-cli multisig sign --tx nominate.tx --output candidate.sign --keys path/to/
↪candidate
```

然后将生成的交易发送

```
# send 后面的签名有两个参数，第一个为发起方的签名，第二个为需要收集的签名（列表逗号分隔）
./xchain-cli multisig send --tx nominate.tx nominate.sign nominate.sign,candidate.sign
```

发送交易会返回一个 txid，这里需要记录下来，后面可能会用到

28.3.2 投票

投票的配置也是一个 json 格式

```
{
    "module": "tdpos",
    "method": "vote",
    "args": {
        # 提名过的 address
        "candidates":["RU7Qv3CrecW5waKc1ZWYnEuTdJNjHc43u"]
    }
}
```

同样使用转账的命令发出，注意投票的 utxo 需要永久冻结。

```
# 同样，转账目标地址可任意填写，转给自己也可以
./xchain-cli transfer --to=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --desc=vote.json --amount=1␣
↪--frozen -1
```

根据共识算法配置的候选人集合大小（上面配置中的”proposer_num”字段，假设为 n），每一轮出块结束后系统都会查看被提名的候选人数目是否达到 n，如果没有达到则继续按上一轮的顺序出块；如果达到 n 则会统计得票靠前的 n 个节点为新一轮的矿工集合

Note: 细心的读者可能已经发现这些配置文件的 json key 都类似，可以参考 xuperchain/core/contract/contract.go 中 TxDesc 的定义

28.3.3 撤销提名 && 撤销投票

Json 格式的配置又来了

```
{
    "module":"proposal",
    "method": "Thaw",
    "args" : {
        # 此处为提名或者投票时的 txid，且 address 与提名或者投票时需要相同
        "txid":"02cd75a721f2589a3ff6768b49650b46fa0b042f970df935b4d28a15aa19e49a"
    }
}
```

然后使用转账操作发出（注意 address 一致），撤销提名/投票后，当时被冻结的资产会解冻，可以继续使用了

```
./xchain-cli transfer --to=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --desc=thaw.json --amount=1
```

28.3.4 TDPOS 结果查询

超级链的客户端提供了这一功能

1. 查询候选人信息

```
./xchain-cli tdpos query-candidates
```

2. 查看某一轮的出块顺序

```
./xchain-cli tdpos query-checkResult -t=30
```

3. 查询提名信息：某地址发起提名的记录

```
./xchain-cli tdpos query-nominate-records -a=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN
```

4. 被提名查询：某个候选人被提名的记录

```
./xchain-cli tdpos query-nominee-record -a=RU7Qv3CrecW5waKc1ZWYnEuTdJNjHc43u
```

5. 某选民的有效投票记录

```
./xchain-cli tdpos query-vote-records -a=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN
```

6. 某候选人被投票记录

```
./xchain-cli tdpos query-voted-records -a=RU7Qv3CrecW5waKc1ZWYnEuTdJNjHc43u
```

各种查询命令的详细参数列表可以通过./xchain-cli tdpos -h 查询

28.4 常见问题

- 端口冲突：注意如果在一台机器上部署多个节点，各个节点的 RPC 监听端口以及 p2p 监听端口都需要设置地不相同，避免冲突；
- 节点公私钥和节点 netUrl 冲突：注意网络中不同节点./data/keys 下的文件和./data/netkeys 下的内容都应该不一样，这两个文件夹是节点在网络中的唯一标识，每个节点需要独自生成，否则网络启动异常；
- 启动时链接 bootNodes 节点失败：注意要先将 bootNodes 节点启动，再起动其他节点，否则会因为加入网络失败而启动失败；
- 遇到 The gas you cousume is: XXXX, You need add fee 通过加–fee XXXX 参数附加资源；

CHAPTER 29

创建合约

29.1 编写合约

源码可以参考 xuperchain/core/contractsdk/go/example/math/math.go

主要实现 struct 中 initialize，invoke 和 query 三个方法来实现自己的逻辑

```
func (m *math) Initialize(nci code.Context) code.Response { ... }
func (m *math) Invoke(nci code.Context) code.Response { ... }
func (m *math) Query(nci code.Context) code.Response { ... }
```

每个函数的入口参数均为 code.Context，具体结构可参考 xuperchain/core/contractsdk/go/code/context.go 接口中定义了如何获取传入方法的参数，如何使用读写功能，以及如何在链上进行交易/区块的查询、转账或调用其他合约

```
type Context interface {
    Args() map[string][]byte
    Caller() string
    Initiator() string
    AuthRequire() []string

    PutObject(key []byte, value []byte) error
    GetObject(key []byte) ([]byte, error)
    DeleteObject(key []byte) error
```

(continues on next page)

(continued from previous page)

```
    NewIterator(start, limit []byte) Iterator

    QueryTx(txid []byte) (*TxStatus, error)
    QueryBlock(blockid []byte) (*Block, error)
    Transfer(to string, amount *big.Int) error
    Call(module, contract, method string, args map[string][]byte) (*Response, error)
}
```

对于 C++ 版本的合约，可以参考代码 contractsdk/cpp/example/counter.cc 原理和 Golang 合约是一致的

Note: 除了 Initialize 外的其他函数，是可以自行定义函数名的，可参考 contractsdk/go/example/counter/counter.go 中的具体实例，在之后调用合约时写明函数名即可

29.2 部署 wasm 合约

1. 编译合约 - Golang

   注意合约编译环境与源码编译环境一致，编译参数如下

   ```
   GOOS=js GOARCH=wasm go build XXX.go
   ```

2. 编译合约 - C++

   对于 C++ 合约，已提供编译脚本，位于 contractsdk/cpp/build.sh，需要注意的是，脚本依赖从 hub.baidubce.com 拉取的 docker 镜像，请在编译前确认 docker 相关环境是可用的

3. 部署 wasm 合约

   将编译好的合约二进制文件（以 counter 为例）放到目录 node/data/blockchain/${chain name}/wasm/下，这里我们默认的链名 ${chain name}=xuper

   部署合约的操作需要由合约账号完成，部署操作同样需要支付手续费，操作前需要确保合约账号下有足够的余额

   示例中我们的环境里创建了一条名为 xuper 的链，包含一个合约账号 **XC1111111111111111@xuper**

   为部署合约，我们需要事先准备一个符合权限的地址列表（示例中将其保存在 data/acl/addrs 文件），这里因为 acl 里只有一个 AK，我们只需在文件中添加一行（如果 acl 中需要多个 AK，那么编辑文件，每行填写一个即可）

   ```
   echo "XC1111111111111111@xuper/dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN" > data/
   ↪acl/addrs
   ```

然后我们按照以下命令来部署 wasm 合约 counter

```
./xchain-cli wasm deploy --account XC1111111111111111@xuper --cname counter␣
↪-m -a '{"creator": "someone"}' -A data/acl/addrs -o tx.output --keys data/
↪keys --name xuper -H localhost:37101 counter
```

此命令看起来很长，但是其中很多参数都有默认值，我们先来看一下参数的含义：

- `wasm deploy`：此为部署 wasm 合约的命令参数，不做过多解释
- `--account XC1111111111111111@xuper`：此为部署 wasm 合约的账号（只有合约账号才能进行合约的部署）
- `--cname counter`：这里的 counter 是指部署后在链上的合约名字，可以自行命名（但有规则，长度在 4 ~ 16 字符）
- `-m`：意为多重签名的方式，目前版本的 xchain 部署 wasm 合约都需要以这种方式
- `-a '{"creator": "someone"}'`：此为传入合约的参数，供合约 Initialize 方法使用（此参数并非必须，只不过此处的 counter 合约需要传一个”creator”参数，参见 contractsdk/cpp/example/counter.cc）
- `-A data/acl/addrs`：此即为需要收集签名的列表，默认路径为 data/acl/addrs，如不是则需要显式传入（注意权重要满足 acl 要求）
- `-o tx.output`：此为输出的 tx 文件，可不传，默认文件名为 tx.out
- `--keys data/keys`：此为部署发起者的密钥地址，可不传，默认值即为 data/keys（部署发起者也要进行签名）
- `--name xuper`：此为区块链名称，默认为 xuper，如果创建链名称不是 xuper 则需要显式传入
- `-H localhost:37101`：xchain 服务的地址，默认是本机的 37101 端口，如不是则需要显式传入
- 最后的 counter 是合约编译好的文件（编译完成默认是 counter.wasm）

在此处，我们大部分参数取的是默认值，所以命令参数不必这么多了

```
./xchain-cli wasm deploy --account XC1111111111111111@xuper --cname counter␣
↪-m -a '{"creator": "someone"}' counter
```

运行时会提示手续费的数目，使用 –fee 参数传入即可

然后收集所需 AK 的签名，因为示例中我们只有一个 AK（同时也是发起者），所以只需要签名一次

```
./xchain-cli multisig sign --tx tx.out --output sign.out --keys data/keys
```

这里的 `--output --keys` 参数也有默认值（输出到 sign.out 文件，密钥位于 data/keys），可以不加。运行后我们即可获得此 AK 的签名

收集完发起者和 acl 需要的签名后，我们即可发送交易，完成合约部署了

```
./xchain-cli multisig send --tx tx.out sign.out sign.out
```

这里 multisig send 为发送多重签名的命令参数，`--tx` 是交易文件，后边的两个参数分别为发起者的签名和 acl 的签名（acl 中有多个 AK 时，用逗号连接多个签名文件）。运行命令可得到交易上链后的 id，我们也可以使用以下命令来查询部署结果

```
./xchain-cli account contracts --account XC1111111111111111@xuper
```

会显示此合约账号部署过的所有合约

29.3 部署 native 合约

1. 编译合约

   编译 native 合约时，只要保持环境和编译 XuperChain 源码时一致即可，我们还是以 example 中的 counter 为例

   ```
   cd contractsdk/go/example/counter
   go build
   # 产出二进制 counter
   ```

2. 激活合约

   native 合约部署需要进行一次 提案-投票 操作，

CHAPTER 30

发起提案

XuperChain 中有多种提案-投票操作场景，但原理都是一致的，我们以通过提案更改共识算法（single 改为 tdpos）来介绍具体的操作流程

部署一个 Single 共识的超级链环境已经在“快速入门”一节有介绍

首先我们需要准备一个 tdpos 共识的配置，包括出块时间、代表名单等（假设文件名为 proposal.json）

```
{
    "module": "proposal",
    "method": "Propose",
    "args" : {
        "min_vote_percent": 51,                 # 生效的资源比例
        "stop_vote_height": 800                 # 计票截至的高度
    },
    "trigger": {
        "height": 1000,                         # 期望生效的高度
        "module": "consensus",
        "method": "update_consensus",
        "args" : {
            "name": "tdpos",
            "config": {
                "version":"2",
                "proposer_num":"2",             # 代表个数
                "period":"3000",
```

(continues on next page)

(continued from previous page)

```
                "alternate_interval":"6000",
                "term_interval":"9000",
                "block_num":"20",
                "vote_unit_price":"1",
                "init_proposer": {              # 出块的代表名单
                    "1":["dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
↪"U5SHuTiiSP1JAAHVMknqrm66QXk2VhXsK"]
                }
            }
        }
    }
}
```

需要注意的是当前的区块高度，来设置合理的截至计票高度和生效高度。然后在矿工节点下，执行给自己转账的操作，并在 –desc 参数里传入提案

```
./xchain-cli transfer --to dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --desc proposal.json
```

运行后会得到本次提案的交易 id，需要记录下来供投票使用

对提案进行投票操作由如下命令执行

```
./xchain-cli vote f26d670b695d9fd5da503a34d130ef19e738b35e031b18b70ad4cbbf6dfe2656 --
↪frozen 1100 --amount 100002825031900000000
```

这里需要注意进行投票的节点需要有矿工账号的密钥对，以及 –frozen 参数的冻结高度大于提案生效的高度。因为最终通过的规则是投票资源大于总资源的 51%，所以需要初始 token 量最多的矿工账号来进行投票，并保证 token 数符合要求。

如此进行后，等到区块出到设定的生效高度，便完成了提案-投票的整个流程。其他场景的提案机制都是类似的，仅是 json 配置文件不同而已。

CHAPTER 31

配置变更

31.1 配置多盘存储

由区块链本身特点决定的，区块链服务启动后需要的存储空间会逐渐变多，即使交易不频繁，每到固定出块时间也会占用少量的存储空间。XuperChain 提供了一种可以将存储路径配置在多个磁盘上的功能，来更好地支持单个磁盘存储空间不充裕的场景。

位于代码目录下的 core/conf/xchain.yaml ，包含了大部分 XuperChain 服务启动的配置项，其中有磁盘相关的章节

```
# 数据存储路径
datapath: ./data/blockchain

# 多盘存储的路径
datapathOthers:
    - /ssd1/blockchain
    - /ssd2/blockchain
    - /ssd3/blockchain
```

只需将“多盘存储路径”部分去掉注释，便可以灵活配置多个数据存储位置。

31.2 替换扩展插件

XuperChain 采用了动态链接库的方式实现了加密、共识算法等扩展插件，可以根据实际使用场景进行替换。

插件目录位于 plugins ， 对应的配置文件为 conf/plugins.conf （json 格式）

```
{
    "crypto":[{
        "subtype": "default",
        "path": "plugins/crypto/crypto-default.so.1.0.0",
        "version": "1.0.0",
        "ondemand": false
    },{
        "subtype": "schnorr",
        "path": "plugins/crypto/crypto-schnorr.so.1.0.0",
        "version": "1.0.0",
        "ondemand": false
    }]
    # ......
}
```

需要替换插件则修改对应的.so 文件路径即可

CHAPTER 32

使用平行链与群组

32.1 创建平行链

现在超级链中创建平行链的方式是：发起一个系统智能合约，发到 xuper 链。

当前 xchain.yaml 有两个配置项：

```
Kernel:
    # minNewChainAmount 设置创建平行链时最少要转多少 utxo（门槛）到同链名的 address
    minNewChainAmount: "100"
    # newChainWhiteList 有权创建平行链的 address 白名单
    newChainWhiteList:
        - dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN: true
```

创建平行链的 json 文件（模版），如下：

```
{
    "Module": "kernel",
    "Method": "CreateBlockChain",
    "Args": {
        "name": "HelloChain",
        "data": "{\"version\": \"1\", \"consensus\": {\"miner\":\
↪"dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN\", \"type\":\"single\"},\"predistribution\":[{\
↪"address\": \"dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN\",\"quota\": \"1000000000000000\"}],\
↪"maxblocksize\": \"128\",\"period\": \"3000\",\"award\": \"1000000\"}"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

使用如下指令即可创建平行链：

```
./xchain-cli transfer --to HelloChain --amount 100 --desc createChain.json
```

32.2 获取 group_chain 合约

超级链提供了默认的群组合约(group_chain)的实现，路径为 core/contractsdk/cpp/example/group_chain.cc 。在 core/contractsdk/cpp 目录下执行 sh build.sh 即可编译生成 group_chain.wasm ，即可使用 group_chain.wasm 实现群组合约的部署。

32.3 创建群组

如果希望创建的平行链只在自己希望的小范围使用，那么可以参考此节配置群组功能

当前超级链中创建群组的方式是：在 xuper 链上部署 GroupChain 智能合约，将节点白名单加到 GroupChain 合约中。

在创世块中配置群组合约配置：

```
{
    "group_chain_contract": {
        "module_name": "wasm",
        "contract_name": "group_chain",
        "method_name": "list",
        "args":{}
    }
}
```

如果需要确保 HelloChain 具备群组属性，且白名单为 <ip1,addr1>,<ip2,addr2>，其他节点不能获取这条平行链的信息，可以按如下操作

step1: 在 xuper 链部署 GroupChain 合约

```
# 需要使用合约账号，部署编译好的合约文件
./xchain-cli wasm deploy --account XC1111111111111111@xuper --cname group_chain ./group_
↪chain.wasm --fee xxx
```

step2: 调用 GroupChain 合约的 AddNode 方法将 <ip1,add1>,<ip2,add2> 加入白名单

```
./xchain-cli wasm invoke group_chain --method addNode -a '{"bcname":"HelloChain", "ip":
↪"ip1", "address":"addr1"}'
./xchain-cli wasm invoke group_chain --method addNode -a '{"bcname":"HelloChain", "ip":
↪"ip2", "address":"addr2"}'
```

step3: 调用 GroupChain 合约的 AddChain 确保 HelloChain 具备群组特性

```
./xchain-cli wasm invoke group_chain --method addChain -a '{"bcname":"HelloChain"}'
```

至此即完成了群组的设置，只有 <ip1,add1>,<ip2,add2> 两个节点可以获取平行链 HelloChain 的内容了。

CHAPTER 33

电子存证合约

33.1 问题引入

假设我们面临着这样的一个问题：“几个摄影师朋友找到你，他们的摄影作品上传到自己的 blog 后总是被其他人盗用，使用水印之类的方法也无法避免像截取部分这种情况，他们需要一个能证明摄影作品最早是由自己上传、而且具有法律效力可供自己进行维权的工具”

显然区块链对于解决此问题有很大的帮助，它的不可篡改等特性很适合存证维权的场景，我们可以通过 XuperChain 来构建一个存取证据的智能合约（担心不被法院认可？这里 或许能够解答你的疑问）

下面我们就来教你帮助摄影师朋友开发一个能够存储照片版权、还能在发现被盗用后进行维权的智能合约

33.2 数据结构的设计

对于摄影作品，通常是一个图片文件，其大小根据清晰度等原因可以多达几十 MB（甚至更多），为避免存储空间浪费、以及保证区块链交易的效率，我们可以使用哈希算法（例如 SHA256）只将图片的哈希值上链，而原图可以保存在其他地方

我们可以这样定义“证据文件”的数据结构，包含哈希值和上传的时间戳

```
type UserFile struct {
    Timestamp int64
    Hashval   []byte
}
```

为了能够存储多个“证据文件”，并且能够服务于更多的摄影师朋友，我们可以定义一个上传者到文件的 map

```
type User struct {
    Owner     string
    UserFiles map[string]*UserFile
}
```

代码样例可以参看：contractsdk/go/example/eleccert.go

33.3 电子存证合约的功能实现

从场景我们可以大致推断，以下两个功能是必要的

- 存储一个到“证据文件”区块链（save 方法）
- 获取已经存储过的某一个“证据文件”（query 方法）

更底层考虑，我们可以使用 XuperChain 提供的合约 SDK 功能 PutObject 和 GetObject 来提供实际的存取功能

对于 XuperChain 中的智能合约，Initialize 是一个必须实现的方法，当且仅当合约被部署的时候会运行一次，我们这里采用“每个摄影师部署自己的合约来存储自己需要的作品”这种方式，将一些和上传者相关的初始化操作放在函数中

Save、Query 和 Initialize 方法的具体实现可以参考代码样例

33.4 合约使用方法

33.4.1 合约部署（Deploy）

编译并部署合约的过程可以参考 部署 wasm 合约 章节，注意资源消耗可以一开始不加 –fee 参数，执行后会给出需要消耗的资源数

33.4.2 合约执行（Save）

执行合约进行“存证操作”的命令如下（运行需要使用 –fee 参数提供资源消耗）：

```
./xchain-cli wasm invoke -a ' 下面 json 中 args 字段的内容' --method save -H␣
↪localhost:37101 eleccert
```

```
{
    "module_name": "wasm",          # native or wasm
```

(continues on next page)

(continued from previous page)

```
    "contract_name": "eleccert",    # contract name
    "method_name": "save",          # invoke or query
    "args": {
        "owner": "aaa",             # user name
        "filehash": "存证文件的 hash 值",
        "timestamp": "存证的 timestamp"
    }
}
```

33.4.3 合约查询（Query）

执行合约进行“取证操作”的命令如下（查询操作不需要提供资源）：

```
./xchain-cli wasm query -a 'args 内容' --method query -H localhost:37101 eleccert
```

```
{
    "module_name": "native",        # native or wasm
    "contract_name": "eleccert",    # contract name
    "method_name": "query",         # invoke or query
    "args": {
        "owner": "aaa",             # user name
        "filehash": "文件 hash 值"
    }
}
# output 如下
{
    "filehash": "文件 hash 值",
    "timestamp": "文件存入 timestamp"
}
```

CHAPTER 34

数字资产交易

代码样例参看：contractsdk/go/example/erc721.go

34.1 ERC721 简介

ERC721 是数字资产合约, 交易的商品是非同质性商品。其中，每一份资产，也就是 token_id 都是独一无二的类似收藏品交易。

34.2 ERC721 具备哪些功能

- **通过 initialize 方法，向交易池注入自己的 token_id**
  - 注意 token_id 必须是全局唯一
- **通过 invoke 方法，执行不同的交易功能**
  - transfer: userA 将自己的某个收藏品 token_id 转给 userB
  - approve: userA 将自己的某个收藏品 token_id 的售卖权限授予 userB
  - transferFrom: userB 替 userA 将赋予权限的收藏品 token_id 卖给 userC
  - pproveAll: userA 将自己的所有收藏品 token_id 的售卖权限授予 userB
- **通过 query 方法，执行不同的查询功能**
  - balanceOf: userA 的所有收藏品的数量

- totalSupply: 交易池中所有的收藏品的数量
- approvalOf: userA 授权给 userB 的收藏品的数量

34.3 调用 json 文件示例

Initialize

./xchain-cli wasm invoke -a '下面 json 中 args 字段的内容' –method initialize -H localhost:37101 erc721

```
{
    "module_name": "native",      # native 或 wasm
    "contract_name": "erc721",    # contract name
    "method_name": "initialize",  # initialize or query or invoke
    "args": {
        "from": "dudu",           # userName
        "supply": "1,2"           # token_ids
    }
}
```

Invoke

./xchain-cli native invoke -a 'args 内容' –method invoke -H localhost:37101 erc721

```
{
    "module_name": "native",      # native 或 wasm
    "contract_name": "erc721",    # contract name
    "method_name": "invoke",      # initialize or query or invoke
    "args": {
        "action": "transfer",     # action name
        "from": "dudu",           # usera
        "to": "chengcheng",       # userb
        "token_id": "1"           # token_ids
    }
}
{
    "module_name": "native",      # native 或 wasm
    "contract_name": "erc721",    # contract name
    "method_name": "invoke",      # initialize or query or invoke
    "args": {
        "action": "transferFrom", # action name
        "from": "dudu",           # userA
        "caller": "chengcheng",   # userB
```

(continues on next page)

(continued from previous page)

```
        "to": "miaomiao",          # userC
        "token_id": "1"            # token_ids
    }
}
{
    "module_name": "native",       # native 或 wasm
    "contract_name": "erc721",     # contract name
    "method_name": "invoke",       # initialize or query or invoke
    "args": {
        "action": "approve",       # action name
        "from": "dudu",            # userA
        "to": "chengcheng",        # userB
        "token_id": "1"            # token_ids
    }
}
```

Query

./xchain-cli native query -a ‘args 内容’ –method query -H localhost:37101 erc721

```
{
    "module_name": "native",      # native 或 wasm
    "contract_name": "erc721",    # contract name
    "method_name": "query",       # initialize or query or invoke
    "args": {
        "action": "balanceOf",    # action name
        "from": "dudu"            # userA
    }
}
{
    "module_name": "native",      # native 或 wasm
    "contract_name": "erc721",    # contract name
    "method_name": "query",       # initialize or query or invoke
    "args": {
        "action": "totalSupply"   # action name
    }
}
{
    "module_name": "native",       # native 或 wasm
    "contract_name": "erc721",     # contract name
    "method_name": "query",        # initialize or query or invoke
```

(continues on next page)

(continued from previous page)

```
    "args": {
        "action": "approvalOf",    # action name
        "from": "dudu",            # userA
        "to": "chengcheng"         # userB
    }
}
```

CHAPTER 35

智能合约 SDK 使用说明

XuperChain 为方便用户开发属于自己的智能合约，提供了一整套 SDK 套件，即 XuperCDT（XuperChain Crontract Development Toolkit），包含 C++ 语言和 Go 语言

35.1 C++ 接口 API

35.1.1 get_object

bool ContextImpl::get_object(const std::string& key, std::string* value)

输入

参数	说明
key	查询的 key 值
value	根据 key 查到的 value 值

输出

参数	说明
true	key 值查询成功，返回 value 值
false	key 值不存在

35.1.2 put_object

bool ContextImpl::put_object(const std::string& key, const std::string& value)

输入

参数	说明
key	存入的 key 值
value	存入 key 值对应的 value 值

输出

参数	说明
true	存入 db 成功
false	存入 db 失败

35.1.3 delete_object

bool ContextImpl::delete_object(const std::string& key)

输入

参数	说明
key	将要删除的 key 值

输出

参数	说明
true	删除成功
false	删除失败

35.1.4 query_tx

bool ContextImpl::query_tx(const std::string &txid, Transaction* tx)

输入

参数	说明
txid	待查询的 txid
tx	得到此 txid 的 transaction

输出

参数	说明
true	查询交易成功
false	查询交易失败

35.1.5 query_block

bool ContextImpl::query_block(const std::string &blockid, Block* block)

输入

参数	说明
blockid	待查询的 blockid
block	得到此 blockid 的 block

输出

参数	说明
true	查询 block 成功
false	查询 block 失败

35.1.6 table

定义表格

```
// 表格定义以 proto 形式建立，存放目录为 contractsdk/cpp/pb
syntax = "proto3";
option optimize_for = LITE_RUNTIME;
package anchor;
message Entity {
    int64 id = 1;
    string name = 2;
    bytes desc = 3;
}
// table 名称为 Entity，属性分别为 id，name，desc
```

初始化表格

```
// 定义表格的主键，表格的索引
struct entity: public anchor::Entity {
    DEFINE_ROWKEY(name);
    DEFINE_INDEX_BEGIN(2)
    DEFINE_INDEX_ADD(0, id, name)
    DEFINE_INDEX_ADD(1, name, desc)
    DEFINE_INDEX_END();
};
// 声明表格
xchain::cdt::Table<entity> _entity;
```

put

```
template <typename T>
bool Table<T>::put(T t)
```

输入

参数	说明
t	待插入的数据项

输出

参数	说明
true	插入成功
false	插入失败

样例

```
// 参考样例 contractsdk/cpp/example/anchor.cc
DEFINE_METHOD(Anchor, set) {
    xchain::Context* ctx = self.context();
    const std::string& id= ctx->arg("id");
    const std::string& name = ctx->arg("name");
    const std::string& desc = ctx->arg("desc");
    Anchor::entity ent;
    ent.set_id(std::stoll(id));
    ent.set_name(name.c_str());
    ent.set_desc(desc);
```

(continues on next page)

(continued from previous page)

```
    self.get_entity().put(ent);
    ctx->ok("done");
}
```

find

```
template <typename T>
bool Table<T>::find(std::initializer_list<PairType> input, T* t)
```

输入

参数	说明
input	查询关键字
t	返回的数据项

输出

参数	说明
true	查询成功
false	查询失败

样例

```
DEFINE_METHOD(Anchor, get) {
    xchain::Context* ctx = self.context();
    const std::string& name = ctx->arg("key");
    Anchor::entity ent;
    if (self.get_entity().find({{"name", name}}, &ent)) {
        ctx->ok(ent.to_str());
        return;
    }
    ctx->error("can not find " + name);
}
```

scan

```
template <typename T>
std::unique_ptr<TableIterator<T>> Table<T>::scan(std::initializer_list<PairType> input)
```

输入

参数	说明
input	查询关键字

输出

参数	说明
TableIterator	符合条件的迭代器

样例

```
DEFINE_METHOD(Anchor, scan) {
    xchain::Context* ctx = self.context();
    const std::string& name = ctx->arg("name");
    const std::string& id = ctx->arg("id");
    // const std::string& desc = ctx->arg("desc");
    auto it = self.get_entity().scan({{"id", id},{"name", name}});
    Anchor::entity ent;
    int i = 0;
    std::map<std::string, bool> kv;
    while(it->next()) {
        if (it->get(&ent)) {
            /*
            std::cout << "id: " << ent.id()<< std::endl;
            std::cout << "name: " << ent.name()<< std::endl;
            std::cout << "desc: " << ent.desc()<< std::endl;
            */
            if (kv.find(ent.name()) != kv.end()) {
                ctx->error("find duplicated key");
                return;
            }
            kv[ent.name()] = true;
            i += 1;
        } else {
            std::cout << "get error" << std::endl;
        }
    }
    std::cout << i << std::endl;
    if (it->error()) {
        std::cout << it->error(true) << std::endl;
```

(continues on next page)

(continued from previous page)

```
    }
    ctx->ok(std::to_string(i));
}
```

del

```
template <typename T>
bool Table<T>::del(T t)
```

输入

参数	说明
t	一个数据项

输出

参数	说明
true	删除成功
false	删除失败

样例

```
DEFINE_METHOD(Anchor, del) {
    xchain::Context* ctx = self.context();
    const std::string& id= ctx->arg("id");
    const std::string& name = ctx->arg("name");
    const std::string& desc = ctx->arg("desc");
    Anchor::entity ent;
    ent.set_id(std::stoll(id));
    ent.set_name(name.c_str());
    ent.set_desc(desc);
    self.get_entity().del(ent);
    ctx->ok("done");
}
```

35.2 Go 接口 API

35.2.1 GetObject

func GetObject(key []byte) ([]byte, error)

输入

参数	说明
key	查询的 key 值

输出

参数	说明
value, nil	key 值查询成功，返回 value 值
_, 非 nil	key 值不存在

35.2.2 PutObject

func PutObject(key []byte, value []byte) error

输入

参数	说明
key	存入的 key 值
value	存入 key 值对应的 value 值

输出

参数	说明
nil	存入 db 成功
非 nil	存入 db 失败

35.2.3 DeleteObject

func DeleteObject(key []byte) error

输入

参数	说明
key	将要删除的 key 值

输出

参数	说明
nil	删除成功
非 nil	删除失败

35.2.4 QueryTx

func QueryTx(txid string) (*pb.Transaction, error)

输入

参数	说明
txid	待查询的 txid

输出

参数	说明
tx, nil	查询交易成功, 得到此 txid 的 transaction
_, 非 nil	查询交易失败

35.2.5 QueryBlock

func QueryBlock(blockid string) (*pb.Block, error)

输入

参数	说明
blockid	待查询的 blockid

输出

参数	说明
block, nil	查询 block 成功, 得到此 blockid 的 block
_, 非 nil	查询 block 失败

35.2.6 NewIterator

func NewIterator(start, limit []byte) Iterator

输入

参数	说明
start	关键字
limit	数据项的最大数量

输出

参数	说明
Iterator	Interator 的接口

样例

```
Key() []byte
Value() []byte
Next() bool
Error() error
// Iterator 必须在使用完毕后关闭
Close()
```

CHAPTER 36

XuperChain RPC 接口使用说明

XuperChain 为方便用户深度使用超级链的各项功能，提供了多语言版本的 SDK（JS，Golang，C#，Java，Python），这里我们以 Golang 为例来介绍一下 XuperChain 的 RPC 接口使用方式。

36.1 RPC 接口介绍

查看 XuperChain 的 proto 文件 ，可以在 service 定义中获取所有支持的 RPC 接口

36.1.1 GetBalance

此接口用于查询指定地址中的余额

参数结构	AddressStatus
返回结构	AddressStatus

这里 AddressStatus 的定义如下

```
message AddressStatus {
    Header header = 1;
    string address = 2;
    repeated TokenDetail bcs = 3;
}
```

其中的 address 字段为需要查询的地址，传入 string 即可

其中的 bcs 字段为需要查询的链名，因为 XuperChain 支持平行链的功能，此字段为列表，亦可传入多个链名，

TokenDetail 定义如下：

```
message TokenDetail {
    string bcname = 1;
    string balance = 2;
    XChainErrorEnum error = 3;
}
```

请求时只需传入 bcname 字段，例如 “xuper”，其余字段为返回时携带的，balance 即为对应平行链上的余额

其中的 Header 如下

```
message Header {
    string logid = 1;
    string from_node = 2;
    XChainErrorEnum error = 3;
}
```

Header 中的 logid 是回复中也会携带的 id，用来对应请求或追溯日志使用的，一般用 core/global/common.go 中的 Glogid() 生成一个全局唯一 id

Header 中的 from_node 一般不需要填写，error 字段也是返回中携带的错误内容，发请求时不需填写

以下为 Golang 示例

```
opts := make([]grpc.DialOption, 0)
opts = append(opts, grpc.WithInsecure())
opts = append(opts, grpc.WithMaxMsgSize(64<<20-1))
conn, _ := grpc.Dial("127.0.0.1:37101", opts...)
cli := pb.NewXchainClient(conn)

bc := &pb.TokenDetail{
    Bcname: "xuper",
}
in := &pb.AddressStatus{
    Header: global.Glogid(),
    Address: "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
    Bcs: []*pb.TokenDetail{bc},
}
out, _ := cli.GetBalance(context.Background(), in)
```

36.1.2 GetBalanceDetail

此接口用于查询指定地址中的余额详细情况

参数结构	AddressBalanceStatus
返回结构	AddressBalanceStatus

AddressBalanceStatus 定义如下

```
message AddressBalanceStatus {
    Header header = 1;
    string address = 2;
    repeated TokenFrozenDetails tfds = 3;
}
```

address 字段与 GetBalance 一样，tfds 字段则多了是否冻结的内容，tfds 在请求中只需要填充 bcname，返回时会有 TokenFrozenDetail 数组给出正常余额和冻结余额的信息

以下为 Golang 示例

```
opts := make([]grpc.DialOption, 0)
opts = append(opts, grpc.WithInsecure())
opts = append(opts, grpc.WithMaxMsgSize(64<<20-1))
conn, _ := grpc.Dial("127.0.0.1:37101", opts...)
cli := pb.NewXchainClient(conn)

tfd := &pb.TokenFrozenDetails{
    Bcname: "xuper",
}
in := &pb.AddressBalanceStatus{
    Header: global.Glogid(),
    Address: "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
    Tfds: []*pb.TokenFrozenDetails{bc},
}
out, _ := cli.GetBalanceDetail(context.Background(), in)
```

36.1.3 GetFrozenBalance

此接口用于查询指定地址中的冻结余额，请求方式与 GetBalance 完全一致，这里不再赘述

36.1.4 GetBlock

此接口用于查询指定 id 的区块内容

参数结构	BlockID
返回结构	Block

BlockID 定义如下

```
message BlockID {
    Header header = 4;
    string bcname = 1;
    bytes blockid = 2;
    bool need_content = 3;  //是否需要内容
}
```

header 和 bcname 字段如上所述，blocked 为要查询的区块 id，注意是 bytes 类型，可能需要 hex decode

need_content 字段为布尔值，表明是否需要详细的区块内容（还是只查询区块是否在链和前驱后继）

以下为 Golang 示例

```
opts := make([]grpc.DialOption, 0)
opts = append(opts, grpc.WithInsecure())
opts = append(opts, grpc.WithMaxMsgSize(64<<20-1))
conn, _ := grpc.Dial("127.0.0.1:37101", opts...)
cli := pb.NewXchainClient(conn)

id, _ := hex.DecodeString(
↪"ee0d6fd34df4a7e1540df309d47441af4fda6fdd9d841046f18e7680fe0cea8c")
in := &pb.BlockID{
    Header: global.Glogid(),
    Bcname: "xuper",
    Blockid: id,
    NeedContent: true,
}
out, _ := cli.GetBlock(context.Background(), in)
```

36.1.5 GetBlockByHeight

此接口用于查询指定高度的区块内容

参数结构	BlockHeight
返回结构	Block

BlockHeight 定义如下

```
message BlockHeight {
    Header header = 3;
    string bcname = 1;
    int64 height = 2;
}
```

同 GetBlock 类似，id 换成整型的高度即可，返回内容也是类似的

36.1.6 GetBlockChainStatus

此接口用于查询指定链的当前状态

参数结构	BCStatus
返回结构	BCStatus

BCStatus 定义如下

```
message BCStatus {
    Header header = 1;
    string bcname = 2;
    LedgerMeta meta = 3;
    InternalBlock block = 4;
    UtxoMeta utxoMeta = 5;
    repeated string branchBlockid = 6;
}
```

传入参数只需填充 header，bcname 即可

以下为 Golang 示例

```
opts := make([]grpc.DialOption, 0)
opts = append(opts, grpc.WithInsecure())
opts = append(opts, grpc.WithMaxMsgSize(64<<20-1))
conn, _ := grpc.Dial("127.0.0.1:37101", opts...)
cli := pb.NewXchainClient(conn)

```

(continues on next page)

(continued from previous page)

```
in := &pb.BCStatus{
    Header: global.Glogid(),
    Bcname: "xuper",
}
out, _ := cli.GetBlockChainStatus(context.Background(), in)
```

36.1.7 GetBlockChains

此接口用于查询当前节点上有哪些链

参数结构	CommonIn
返回结构	BlockChains

CommonIn 结构很简单，只有 header 字段，返回的 BlockChains 也仅有一个链名的 string 数组

以下为 Golang 示例

```
opts := make([]grpc.DialOption, 0)
opts = append(opts, grpc.WithInsecure())
opts = append(opts, grpc.WithMaxMsgSize(64<<20-1))
conn, _ := grpc.Dial("127.0.0.1:37101", opts...)
cli := pb.NewXchainClient(conn)

in := &pb.CommonIn{
    Header: global.Glogid(),
}
out, _ := cli.GetBlockChains(context.Background(), in)
```

36.1.8 GetSystemStatus

此接口用于查询当前节点的运行状态

参数结构	CommonIn
返回结构	SystemsStatusReply

此接口相当于先查询了 GetBlockChains，在用 GetBlockChainStatus 查询每个链的状态，不在赘述

36.1.9 GetNetURL

此接口用于查询当前节点的 netUrl

参数结构	CommonIn
返回结构	RawUrl

RawUrl 除了 header 字段外仅有一个 string 字段，表示返回的 netURL

36.1.10 QueryACL

此接口用于查询指定合约账号的 ACL 内容

参数结构	AclStatus
返回结构	AclStatus

AclStatus 定义如下

```
message AclStatus {
    Header header = 1;
    string bcname = 2;
    string accountName = 3;
    string contractName = 4;
    string methodName = 5;
    bool confirmed = 6;
    Acl acl = 7;
}
```

请求中仅需填充 header，bcname，accountName 即可，其余为返回内容

以下为 Golang 示例

```
in := &pb.AclStatus{
    Header: global.Glogid(),
    Bcname: "xuper",
    AccountName: "XC1111111111111111@xuper",
}
out, _ := cli.QueryACL(context.Background(), in)
```

36.1.11 QueryTx

此接口用于查询指定 id 的交易内容

参数结构	TxStatus
返回结构	TxStatus

TxStatus 定义如下

```
message TxStatus {
    Header header = 1;
    string bcname = 2;
    bytes txid = 3;
    TransactionStatus status = 4;  //当前状态
    int64 distance = 5;  //离主干末端的距离（如果在主干上）
    Transaction tx = 7;
}
```

请求中仅需填充 header，bcname，txid 字段

以下为 Golang 示例

```
id, _ := hex.DecodeString(
↪"763ac8212c80b8789cefd049f1529eafe292f4d64eaffbc2d5fe19c79062a484")
in := &pb.AclStatus{
    Header: global.Glogid(),
    Bcname: "xuper",
    Txid: id,
}
out, _ := cli.QueryTx(context.Background(), in)
```

36.1.12 SelectUTXO

此接口用于获取账号可用的 utxo 列表

参数结构	UtxoInput
返回结构	UtxoOutput

UtxoInput 定义如下

```
message UtxoInput {
    Header header = 1;
    // which bcname to select
    string bcname = 2;
    // address to select
    string address = 3;
    // publickey of the address
    string publickey = 4;
    // totalNeed refer the total need utxos to select
```

(continues on next page)

(continued from previous page)

```
    string totalNeed = 5;
    // userSign of input
    bytes userSign = 7;
    // need lock
    bool needLock = 8;
}
```

请求中只需填充 header，bcname，address，totalNeed，needLock，其中 needLock 表示是否需要锁定 utxo（适用于并发执行场景）

UtxoOutput 中的返回即可在组装交易时使用，具体组装交易的过程可参考文档下方

```
in := &pb.UtxoInput{
    Header: global.Glogid(),
    Bcname: "xuper",
    Address: "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
    TotalNeed: "50",
    NeedLock: true,
}
out, _ := cli.SelectUTXO(context.Background(), in)
```

36.1.13 SelectUTXOBySize

此接口用于获取账号中部分 utxo，填满交易后便不在继续获取

参数结构	UtxoInput
返回结构	UtxoOutput

使用过程和 SelectUTXO 基本相同，仅少了 totalNeed 字段。适用拥有太多 utxo，一次 SelectUtxo 内容超过交易容纳上限时使用

36.1.14 PreExec

此接口用于在节点上进行合约的预执行操作，返回预执行后的请求和回复

参数结构	InvokeRPCRequest
返回结构	InvokeRPCResponse

InvokeRPCRequest 定义如下

```
message InvokeRPCRequest {
    Header header = 1;
    string bcname = 2;InvokeRequest
    repeated  requests = 3;
    string initiator = 4;
    repeated string auth_require = 5;
}
```

其中的 InvokeRequest 定义如下

```
message InvokeRequest {
    string module_name = 1;
    string contract_name = 2;
    string method_name = 3;
    map<string, bytes> args = 4;
    repeated ResourceLimit resource_limits = 5;
    string amount = 6;
}
```

其中必填字段有 module_name，contract_name，method_name，args，具体示例可参见下一章节

36.1.15 PreExecWithSelectUTXO

此接口用于在节点上进行消耗资源的合约预执行操作，内部是由一个 PreExec 加上一个 SelectUTXO 实现的，预执行并选择出需要消耗数额的 utxo

参数结构	PreExecWithSelectUTXORequest
返回结构	PreExecWithSelectUTXOResponse

PreExecWithSelectUTXORequest 定义如下，实际上就是把预执行的请求结构放在了 SelectUTXO 结构中

```
message PreExecWithSelectUTXORequest {
    Header header = 1;
    string bcname = 2;
    string address = 3;
    int64 totalAmount = 4;
    SignatureInfo signInfo = 6;
    bool needLock = 7;
    InvokeRPCRequest request = 5;
}
```

具体填充方式可参考下一章节

36.1.16 PostTx

此接口用于提交交易，是大部分操作都需要的最终环节

参数结构	TxStatus
返回结构	CommonReply

请求结构 TxStatus 定义在 QueryTx 中已经给出，但提交交易时需要填充 Transaction 字段，定义如下

```
message Transaction {
    // txid is the id of this transaction
    bytes txid = 1;
    // the blockid the transaction belong to
    bytes blockid = 2;
    // Transaction input list
    repeated TxInput tx_inputs = 3;
    // Transaction output list
    repeated TxOutput tx_outputs = 4;
    // Transaction description or system contract
    bytes desc = 6;
    // Mining rewards
    bool coinbase = 7;
    // Random number used to avoid replay attacks
    string nonce = 8;
    // Timestamp to launch the transaction
    int64 timestamp = 9;
    // tx format version; tx 格式版本号
    int32 version = 10;
    // auto generated tx
    bool autogen = 11;
    repeated TxInputExt tx_inputs_ext = 23;
    repeated TxOutputExt tx_outputs_ext = 24;
    repeated InvokeRequest contract_requests = 25;
    // 权限系统新增字段
    // 交易发起者，可以是一个 Address 或者一个 Account
    string initiator = 26;
    // 交易发起需要被收集签名的 AddressURL 集合信息，包括用于 utxo 转账和用于合约调用
    repeated string auth_require = 27;
    // 交易发起者对交易元数据签名，签名的内容包括 auth_require 字段
    repeated SignatureInfo initiator_signs = 28;
    // 收集到的签名
```

(continues on next page)

(continued from previous page)

```
    repeated SignatureInfo auth_require_signs = 29;
    // 节点收到 tx 的时间戳，不参与签名
    int64 received_timestamp = 30;
    // 统一签名 (支持多重签名/环签名等，与 initiator_signs/auth_require_signs 不同时使用)
    XuperSignature xuper_sign = 31;
    // 可修改区块链标记
    ModifyBlock modify_block = 32;
}
```

Transaction 属于 XuperChain 中比较核心的结构了，下一章我们将介绍各种场景的交易如何构造并提交

36.2 RPC 接口应用

本章节将以几个简单的场景为例描述 RPC 接口的使用方法，主要体现逻辑和步骤。代码中仅使用了原始的 RPC 接口，如果使用 SDK 则会简便很多。

36.2.1 发起一次转账

这里我们演示如何使用 RPC 接口实现从账号 Aclie 向账号 Bob 的一次数额为 10 的转账，为了进行此操作，我们事先需要有以下信息（均为 string）

Alice 的地址	addr_alice
Alice 的公钥	pub_alice
Alice 的私钥	pri_alice
Bob 的地址	addr_bob

发起转账交易的总体逻辑为，首先通过 SelectUTXO 获取 Alice 数额为 10 的资产，然后构造交易，最后通过 PostTx 提交

```
// 获取 Alice 的 utxo
utxoreq := &pb.UtxoInput{
    Header: global.Glogid(),
    Bcname: "xuper",
    Address: addr_alice,
    TotalNeed: "10",
    NeedLock: true,
}
utxorsp, _ := cli.SelectUTXO(context.Background(), utxoreq)
// 声明一个交易，发起者为 Alice 地址，因为是转账，所以 Desc 字段什么都不填
```

(continues on next page)

(continued from previous page)

```
// 如果是提案等操作，将客户端的 --desc 参数写进去即可
tx := &pb.Transaction{
    Version: 1,
    Coinbase: false,
    Desc: []byte(""),
    Nonce: global.GenNonce(),
    Timestamp: time.Now().UnixNano(),
    Initiator: addr_alice,
}
// 填充交易的输入，即 Select 出来的 Alice 的 utxo
for _, utxo := range utxorsp.UtxoList {
    txin := &pb.TxInput{
        RefTxid: utxo.RefTxid,
        RefOffset: utxo.RefOffset,
        FromAddr: utxo.ToAddr,
        Amount: utxo.Amount,
    }
    tx.TxInputs = append(tx.TxInputs, txin)
}
// 填充交易的输出，即给 Bob 的 utxo，注意 Amount 字段的类型
amount, _ := big.NewInt(0).SetString("10", 10)
txout := &pb.TxOutput{
    ToAddr: []byte(addr_bob),
    Amount: amount.Bytes(),
}
tx.TxOutputs = append(tx.TxOutputs, txout)
// 如果 Select 出来的 Alice 的 utxo 多于 10，需要构造一个给 Alice 的找零
total, _ := big.NewInt(0).SetString(utxorsp.TotalSelected, 10)
if total.Cmp(amount) > 0 {
    delta := total.Sub(total, amount)
    charge := &pb.TxOutput{
        ToAddr: []byte(addr_alice),
        Amount: delta.Bytes(),
    }
    tx.TxOutputs = append(tx.TxOutputs, charge)
}
// 接下来用 Alice 的私钥对交易进行签名，在此交易中，我们只需 Alice 签名确认即可
tx.AuthRequire = append(tx.AuthRequire, addr_alice)
// 签名需要的库在 github.com/xuperchain/xuperchain/core/crypto/client
// 和 github.com/xuperchain/xuperchain/core/crypto/hash
```

(continues on next page)

(continued from previous page)

```
cryptoCli, _ := client.CreateCryptoClient("default")
sign, _ := txhash.ProcessSignTx(cryptoCli, tx, []byte(pri_alice))
signInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: sign,
}
// 将签名填充进交易
tx.InitiatorSigns = append(tx.InitiatorSigns, signInfo)
tx.AuthRequireSigns = append(tx.AuthRequireSigns, signInfo)
// 生成交易 ID
tx.Txid, _ = txhash.MakeTransactionID(tx)
// 构造最终要 Post 的 TxStatus
txs := &pb.TxStatus{
    Bcname: "xuper",
    Status: pb.TransactionStatus_UNCONFIRM,
    Tx: tx,
    Txid: tx.Txid,
}
// 最后一步，执行 PostTx
rsp, err := cli.PostTx(context.Background(), txs)
// 这里的 rsp 即 CommonReply，包含 logid 等内容
// 交易 id 我们已经生成在 tx.Txid 中，不过是 bytes，输出可能需要 hex.EncodeToString 一下
```

36.2.2 新建合约账号

这里我们演示创建一个合约账号 XC1234567812345678@xuper ，ACL 如下

```
{
    "pm": {
        "rule": 1,
        "acceptValue": 1.0
    },
    "aksWeight": {
        "XXXaddress-aliceXXX" : 0.6,
        "XXXXaddress-bobXXXX" : 0.4
    }
}
```

为了进行此操作，我们事先需要有以下信息

Alice 的地址	addr_alice
Alice 的公钥	pub_alice
Alice 的私钥	pri_alice
ACL 的内容	acct_acl

创建合约账号的总体逻辑为，首先进行创建合约账号的预执行，然后构造相应的交易内容（如果需要支付资源由 Alice 出），最后提交交易

```
// 构造创建合约账号的请求
args := make(map[string][]byte)
args["account_name"] = []byte(1234567812345678)
args["acl"] = []byte(acct_acl)
invokereq := &pb.InvokeRequest{
    ModuleName: "xkernel",
    MethodName: "NewAccount",
    Args: args,
}
invokereqs := []*pb.InvokeRequest{invokereq}
// 构造合约预执行的请求
authrequire := []string{addr_alice}
rpcreq := &pb.InvokeRPCRequest{
    Header: global.Glogid(),
    Bcname: "xuper",
    Requests: invokereqs,
    Initiator: addr_alice,
    AuthRequire: authrequire,
}
// 花手续费需要出资的账号确认，填充一个验证的签名，才能正确的拿出 utxo 来
// 签名需要的库在 github.com/xuperchain/xuperchain/core/crypto/client
// 和 github.com/xuperchain/xuperchain/core/crypto/hash
content := hash.DoubleSha256([]byte("xuper" + addr_alice + "0" + "true"))
cryptoCli, _ := client.CreateCryptoClient("default")
prikey, _ := cryptoCli.GetEcdsaPrivateKeyFromJSON([]byte(pri_alice))
sign, _ := cryptoCli.SignECDSA(prikey, content)
signInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: sign,
}
// 组合一个 PreExecWithSelectUTXORequest 用来预执行同时拿出需要支付的 Alice 的 utxo
prereq := &pb.PreExecWithSelectUTXORequest{
```

(continues on next page)

(continued from previous page)

```
    Header: global.Glogid(),
    Bcname: "xuper",
    Address: addr_alice,
    TotalAmount: 0,
    SignInfo: signInfo,
    NeedLock: true,
    Request: rpcreq,
}
prersp := cli.PreExecWithSelectUTXO(context.Background(), prereq)
// 构造一个 Alice 发起的交易
tx := &pb.Transaction{
    Version: 1,
    Coinbase: false,
    Desc: []byte(""),
    Nonce: global.GenNonce(),
    Timestamp: time.Now().UnixNano(),
    Initiator: addr_alice,
}
// 填充支付的手续费，手续费需要“转账”给地址“$”
amount := big.NewInt(prersp.Response.GasUsed)
fee := &pb.TxOutput{
    ToAddr: []byte("$"),
    Amount: amount.Bytes(),
}
tx.TxOutputs = append(tx.TxOutputs, fee)
// 填充 select 出来的 Alice 的 utxo
for _, utxo := range prersp.UtxoOutput.UtxoList {
    txin := &pb.TxInput{
        RefTxid: utxo.RefTxid,
        RefOffset: utxo.RefOffset,
        FromAddr: utxo.ToAddr,
        Amount: utxo.Amount,
    }
    tx.TxInputs = append(tx.TxInputs, txin)
}
// 处理找零的逻辑
total, _ := big.NewInt(0).SetString(prersp.UtxoOutput.TotalSelected, 10)
if total.Cmp(amount) > 0 {
    delta := total.Sub(total, amount)
    charge := &pb.TxOutput{
```

(continues on next page)

(continued from previous page)

```
        ToAddr: []byte(addr_alice),
        Amount: delta,
    }
}
// 填充预执行的结果
tx.ContractRequests = prersp.GetResponse().GetRequests()
tx.TxInputsExt = prersp.GetResponse().GetInputs()
tx.TxOutputsExt = prersp.GetResponse().GetOutputs()
// 给交易签名
tx.AuthRequire = append(tx.AuthRequire, addr_alice)
txsign, _ := txhash.ProcessSignTx(cryptoCli, tx, []byte(pri_alice))
txsignInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: txsign,
}
tx.InitiatorSigns = append(tx.InitiatorSigns, txsignInfo)
tx.AuthRequireSigns = append(tx.AuthRequireSigns, txsignInfo)
// 生成交易 ID
tx.Txid, _ = txhash.MakeTransactionID(tx)
// 构造最终要 Post 的 TxStatus
txs := &pb.TxStatus{
    Bcname: "xuper",
    Status: pb.TransactionStatus_UNCONFIRM,
    Tx: tx,
    Txid: tx.Txid,
}
// 最后一步，执行 PostTx
rsp, err := cli.PostTx(context.Background(), txs)
```

36.2.3 修改合约账号 ACL

延续上一小节的例子，假设我们要把 ACL 修改成以下状态

```
{
    "pm": {
        "rule": 1,
        "acceptValue": 1.0
    },
    "aksWeight": {
```

(continues on next page)

(continued from previous page)

```
        "XXXaddress-aliceXXX" : 1.0,
        "XXXXaddress-bobXXXX" : 1.0
    }
}
```

为了进行此操作，我们事先需要有以下信息

Alice 的地址	addr_alice
Alice 的公钥	pub_alice
Alice 的私钥	pri_alice
Bob 的地址	addr_bob
Bob 的公钥	pub_bob
Bob 的私钥	pri_bob
新 ACL 的内容	new_acl

修改 ACL 的总体逻辑为，首先进行修改的预执行，然后构造交易发送，这里需要注意的是，修改 ACL 操作需要满足现有的 ACL 要求才有权限，即 Alice Bob 都需要签名确认。简单起见，当中的手续费依然由 Alice 支付。

```
// 构造修改 ACL 的请求
args := make(map[string][]byte)
args["account_name"] = []byte(1234567812345678)
args["acl"] = []byte(new_acl)
invokereq := &pb.InvokeRequest{
    ModuleName: "xkernel",
    MethodName: "SetAccountAcl",
    Args: args,
}
invokereqs := []*pb.InvokeRequest{invokereq}

// 构造合约预执行的请求，和上一节一样，此处省略
////////////////////////////////////////////
// 花手续费需要出资的账号确认，填充验证的签名，和上一节一样，此处省略
////////////////////////////////////////////////////////////////////////
// 按上一节逻辑一样，填充花费、找零，然后填充预执行的结果
tx.ContractRequests = prersp.GetResponse().GetRequests()
tx.TxInputsExt = prersp.GetResponse().GetInputs()
tx.TxOutputsExt = prersp.GetResponse().GetOutputs()
// 给交易签名需要原 ACL 里的多个账号了
tx.AuthRequire = append(tx.AuthRequire, addr_alice)
```

(continues on next page)

(continued from previous page)

```
tx.AuthRequire = append(tx.AuthRequire, addr_bob)
alicesign, _ := txhash.ProcessSignTx(cryptoCli, tx, []byte(pri_alice))
alicesignInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: alicesign,
}
bobsign, _ := txhash.ProcessSignTx(cryptoCli, tx, []byte(pri_bob))
bobsignInfo := &pb.SignatureInfo{
    PublicKey: pub_bob,
    Sign: bobsign,
}
tx.InitiatorSigns = append(tx.InitiatorSigns, alicesignInfo)
tx.AuthRequireSigns = append(tx.AuthRequireSigns, alicesignInfo)
tx.AuthRequireSigns = append(tx.AuthRequireSigns, bobsignInfo)
// 然后和上一节一致了，生成交易 ID
tx.Txid, _ = txhash.MakeTransactionID(tx)
// 构造最终要 Post 的 TxStatus
txs := &pb.TxStatus{
    Bcname: "xuper",
    Status: pb.TransactionStatus_UNCONFIRM,
    Tx: tx,
    Txid: tx.Txid,
}
// 最后一步，执行 PostTx
rsp, err := cli.PostTx(context.Background(), txs)
```

36.2.4 部署一个合约

这里我们演示使用合约账号 XC1234567812345678@xuper 部署一个 C++ 的 counter 合约，init 参数为 { “creator” :” xchain” }，假设合约账号的 ACL 是修改过的版本

为了进行此操作，我们事先需要有以下信息

合约文件字节内容	contract_code
Alice 的地址	addr_alice
Alice 的公钥	pub_alice
Alice 的私钥	pri_alice

部署合约的总体逻辑为，首先构造 deploy 操作预执行，部署需要的手续费由合约账号出，需要的签名由 Alice 提供（因为一个签名就满足 ACL 了）

```
// 构造部署合约的请求，关注 args 的内容，基本上和使用 xchain-cli 一致
args := make(map[string][]byte)
args["account_name"] = []byte("XC1234567812345678@xuper")
args["contract_name"] = []byte("counter")
// github.com/golang/protobuf/proto
codedesc := desc := &pb.WasmCodeDesc{
    Runtime: "c",
}
desc, _ := proto.Marshal(codedesc)
args["contract_desc"] = desc
args["contract_code"] = contract_code
initarg := `{"creator":"` + base64.StdEncoding.EncodeToString([]byte("xchain")) + `"}`
args["init_args"] = []byte(initarg)
invokereq := &pb.InvokeRequest{
    ModuleName: "xkernel",
    MethodName: "Deploy",
    Args: args,
}
invokereqs := []*pb.InvokeRequest{invokereq}
// 这里预执行的 authrequire 格式为 XC1234567812345678@xuper/
↪dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN,
// 表示是“某个合约账号的股东”，与直接写账号地址含义是不同的，ACL 需求多个签名的时候即多个“股
东”
authrequires := []string{"XC1234567812345678@xuper/XXXaddress-aliceXXX"}
rpcreq := &pb.InvokeRPCRequest{
    Header: global.Glogid(),
    Bcname: "xuper",
    Requests: invokereqs,
    Initiator: addr_alice,
    AuthRequire: authrequires,
}
// SelectUTXO 的目标是合约账号中的余额，出资账号签名中的地址变成了合约账号，与“创建账号”小
节有区别
content := hash.DoubleSha256([]byte("xuper" + "XC1234567812345678@xuper" + "0" + "true"))
prikey, _ := cryptoCli.GetEcdsaPrivateKeyFromJSON([]byte(pri_alice))
sign, _ := cryptoCli.SignECDSA(prikey, content)
signInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: sign,
}
```

(continues on next page)

(continued from previous page)

```
// 组合一个 PreExecWithSelectUTXORequest 用来预执行同时拿出需要支付的合约账号的 utxo
prereq := &pb.PreExecWithSelectUTXORequest{
    Header: global.Glogid(),
    Bcname: "xuper",
    Address: "XC1234567812345678@xuper",
    TotalAmount: 0,
    SignInfo: signInfo,
    NeedLock: true,
    Request: rpcreq,
}
prersp, _ := cli.PreExecWithSelectUTXO(context.Background(), prereq)
// 构造一个 Alice 发起的交易
tx := &pb.Transaction{
    Version: 1,
    Coinbase: false,
    Desc: []byte(""),
    Nonce: global.GenNonce(),
    Timestamp: time.Now().UnixNano(),
    Initiator: addr_alice,
}
// 填充支付的手续费，手续费需要“转账”给地址“$”
amount := big.NewInt(prersp.Response.GasUsed)
fee := &pb.TxOutput{
    ToAddr: []byte("$"),
    Amount: amount.Bytes(),
}
tx.TxOutputs = append(tx.TxOutputs, fee)
// 填充 select 出来的 Alice 的 utxo
for _, utxo := range prersp.UtxoOutput.UtxoList {
    txin := &pb.TxInput{
        RefTxid: utxo.RefTxid,
        RefOffset: utxo.RefOffset,
        FromAddr: utxo.ToAddr,
        Amount: utxo.Amount,
    }
    tx.TxInputs = append(tx.TxInputs, txin)
}
// 处理找零的逻辑
total, _ := big.NewInt(0).SetString(prersp.UtxoOutput.TotalSelected, 10)
if total.Cmp(amount) > 0 {
```

(continues on next page)

(continued from previous page)

```
    delta := total.Sub(total, amount)
    charge := &pb.TxOutput{
        ToAddr: []byte("XC1234567812345678@xuper"),
        Amount: delta,
    }
}
// 填充预执行的结果
tx.ContractRequests = prersp.GetResponse().GetRequests()
tx.TxInputsExt = prersp.GetResponse().GetInputs()
tx.TxOutputsExt = prersp.GetResponse().GetOutputs()
// 给交易签名，此处也是以“股东”身份签名
tx.AuthRequire = append(tx.AuthRequire, "XC1234567812345678@xuper/XXXaddress-aliceXXX")
txsign, _ := txhash.ProcessSignTx(cryptoCli, tx, []byte(pri_alice))
txsignInfo := &pb.SignatureInfo{
    PublicKey: pub_alice,
    Sign: txsign,
}
// 虽然 Alice 和“股东 Alice”含义不同，但签名的私钥是一样的
tx.InitiatorSigns = append(tx.InitiatorSigns, signInfo)
tx.AuthRequireSigns = append(tx.AuthRequireSigns, signInfo)
tx.Txid, _ = txhash.MakeTransactionID(tx)
// 构造最终要 Post 的 TxStatus
txs := &pb.TxStatus{
    Bcname: "xuper",
    Status: pb.TransactionStatus_UNCONFIRM,
    Tx: tx,
    Txid: tx.Txid,
}
// 最后一步，执行 PostTx
rsp, err := cli.PostTx(context.Background(), txs)
```

36.2.5 执行一个 wasm 合约

这里我们演示使用 Alice 账号调用上一节部署的 counter 合约，执行 increase 方法，参数为 {“key”: “example” }

为了进行此操作，我们事先需要有以下信息

Alice 的地址	addr_alice
Alice 的公钥	pub_alice
Alice 的私钥	pri_alice

执行合约的总体逻辑为，首先构造相应预执行请求并预执行，如果是查询，那么直接读预执行结果即可，如果是要调用上链的操作，使用预执行结果组建交易并发送

```
// 构造执行合约的请求
args := make(map[string][]byte)
args["key"] = []byte("example")
invokereq := &pb.InvokeRequest{
    ModuleName: "wasm",
    MethodName: "increase",
    ContractName: "counter",
    Args: args,
}
invokereqs := []*pb.InvokeRequest{invokereq}
// 其他内容和“创建合约账号”一节完全一致
```

CHAPTER 37

超级链测试环境说明

区块链是信任的连接器，通过区块链可以做到很多之前中心化信息系统做不到的事情，使得参与者可以凭借这个“连接器”完成成可信环境的构建和价值的安全流转。然而，目前多数公链的性能和安全不足以支撑各行各业的诸多场景，百度超级链（XuperChain）是百度自主研发的区块链技术，目前已启动测试环境对外公开测试，欢迎各界开发者使用我们的产品并且提出宝贵意见。

37.1 超级链公开测试环境（XuperChain-testnet）

超级链公开测试环境是超级链许可开放网络的测试版本，目前超级链测试环境已经实现了超级链的主要功能，为超级链早期用户和开发者提供一个可供使用的测试环境。

用户可以在超级链测试环境上测试部署和使用智能合约等功能，用户可以通过开源代码直接开发及平台化操作两种方式获取测试环境资源以及开发智能合约。

37.2 测试环境使用场景

超级链测试环境 **适用于**：

- 创建测试账号，更为方便的按照教程尝试使用。
- 开发测试智能合约，而无需担心影响真实数据资产。
- 超级链新版本上线前的兼容性升级、功能测试等。

超级链测试环境 **不适用于**：

- 压力测试：如果有压力测试需求，请在自行搭建的测试环境上实验，数据会更准确。
- 可用性测试：测试环境并不保证高可用性，可能在某些情况下出现短暂的服务不可用。
- 长期数据存储：测试环境不保证数据长期有效，可能在系统 bug、不兼容升级、遭受攻击等情况下，会重置甚至关闭测试环境。我们会尽量保障用户数据不丢失，但在测试环境重置或关闭时，用户可能并不会得到通知，链上数据也可能无法找回。

37.3 测试环境资源

用户在测试环境部署和调用合约需要使用并且消耗测试资源。

目前测试环境获取测试资源会有以下两种方式：

- 通过超级链公开网络渠道微信群、邮箱等等获取测试资质并领取定量测试资源；
- 批量资源获取可邮件联系：xuper.baidu.com 。

测试资源仅用于测试环境消耗计算，只在测试环境上有效，没有任何经济价值和法律效力，也不支持用于任何形式的交易。如果测试环境因为遭受攻击、不兼容升级等情况下重建时，我们会尽量恢复用户账号中持有的测试资源。

37.4 测试环境开发方式

37.4.1 如何接入

测试环境通过 RPC 方式提供服务，开发者可以在 github 获取源代码，按照 README 说明编译得到 cli 客户端。

测试环境接入地址：**14.215.179.74:37101**

开发者只需要在使用 xchain-cli 时，通过-H 参数指定测试环境地址，即可将客户端命令发送到测试环境。例如查询账号测试资源：

```
./xchain-cli account balance dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN -H 14.215.179.74:37101
```

37.4.2 如何开发智能合约

开发者需要首先创建合约账号，合约账号必须消耗一定的测试资源，用于部署合约。

更多关于测试环境使用的方法，请参见 测试环境使用指南 。

37.5 用户使用条款

超级链用户在使用公开测试环境期间，不得访问或使用本网站采取任何可能损害我们或者任何第三方的行为，干扰本网络的运营或者以违反任何法律的方式使用本网络的行为，超级链有权删除相关数据或者追究法律责任，包括但不限于：

- 分享任何违反这些条款或其他适用条款的内容;
- 上传病毒或恶意代码或做任何可能导致我们网络无法正常工作，负担过重或损害的事情;
- 使用自动方式访问或收集我们产品的数据（未经我们事先许可）或尝试访问您无权访问的数据
- 从事任何限制或禁止任何人使用或享用本网站的行为，或根据我们的判断会使我们或我们的任何用户、关联公司或任何其他第三方承担任何责任，损害或损害任何类型的行为。
- 违反系统或网络安全可能导致责任。我们可以随时以任何理由暂停或终止您访问本网站，恕不另行通知
- 使用本网络对外提供服务时，业务须自行前往网信办备案，如由于备案原因造成的法律风险本网站不承担任何责任。

您在使用超级链测试环境前，请确认已经明确了解上述用户须知，当您使用测试环境时表示您已知悉并接收测试环境用户须知。如果测试环境不能满足您的需求，您也可以按照 超级链官方文档 搭建自己的测试环境。如果因为测试环境使用上带来的问题，我们不承担任何法律责任。

CHAPTER 38

超级链测试环境使用指南

38.1 测试环境说明

在使用测试环境之前，请先点开 超级链测试环境说明 ，认真阅读超级链测试环境的目的、使用场景、用户使用条款。

如果您对测试环境说明中的内容有任何疑问，可以通过 xchain.baidu.com 联系我们。如果您使用超级链测试环境，我们认为您已经明确并接受超级链测试环境说明中的相关内容和用户使用条款。

38.2 如何接入

测试环境通过 RPC 方式提供服务，开发者可以在 github 获取源代码，按照 README 说明编译得到 cli 客户端，当前测试环境使用 v3.2 分支。

- 测试环境接入地址：**14.215.179.74:37101**
- 黄反服务的 address：**XDxkpQkfLwG6h56e896f3vBHhuN5g6M9u**

开发者只需要在使用 xchain-cli 时，通过-H 参数指定测试环境地址，即可将客户端命令发送到测试环境。例如查询账号测试资源：

```
./xchain-cli account balance dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN -H 14.215.179.74:37101
```

38.3 关于测试资源

测试环境上部署和执行智能合约需要消耗测试资源，测试环境目前尚未开放外部全节点 P2P 连接，且测试环境上节点出块并不会获得奖励，但出块节点会获得该块中所有 Transaction 中支付的测试资源。

目前获取测试资源主要有两种方式：

1. 通过超级链公开网络渠道微信群、邮箱等等获取测试资质，可以免费获得测试资源。可以通过 这里 的微信二维码加入超级链用户微信群。
2. 批量资源获取可邮件联系：xuper.baidu.com

38.4 创建账号

38.4.1 创建个人账号 (AK)

个人账号 (AK) 其实是一组公私钥对，个人帐号地址 (address) 是根据公钥经过一定规则导出的一个散列值。个人账号可以离线生成，不需要上链，只有在个人账号产生测试资源变动时 (例如转入了一部分测试资源) 才会在 UTXO 中产生记录。

在 data/keys 下会有一个默认的个人账号 (AK)，包括 address(你的地址)、private.key(你的私钥)、public.key(你的公钥)，建议按照如下命令重新生成一个独有的个人账号。

- 指定私钥目录：在 data/test_demo 下生成 address、private.key、public.key: ./xchain-cli account newkeys –output data/test_demo
- 覆盖默认目录：覆盖 data/keys 下的文件，需要先删除 data/keys 目录，然后重新生成新的 address、private.key、public.key

```
rm -r data/keys
./xchain-cli account newkeys
```

个人账号地址默认在 data/keys/address 文件中，可通过 cat data/keys/address 查看自己的个人账号地址。

38.4.2 创建合约账号 (Account)

合约账号可以用来部署智能合约，创建合约账号是一个上链操作，因此也需要消耗一定量的测试资源。合约账号可以设置为多个个人账号共同持有，只有一个交易中的背书签名满足一定合约账号的 ACL 要求，才能代表这个合约账号进行操作。关于合约账号和 ACL 权限相关的内容，可以参考

Note: 创建合约账号需要向黄反服务拿一个签名，对应地，需要将黄反服务的 address 写到 data/acl/addrs 中，需要注意的是，multisig 最终合入签名时需要将签名顺序与 data/acl/addrs 里面的地址顺序保持一致，否则会签名校验失败。

- Step0: 创建合约账号是一个系统合约，可以通过多重签名的方式发起系统合约调用。系统合约调用需要先创建一个合约调用描述文件，例如下面 newAccount.json 是一个创建合约账号的描述文件。newAccount.json 文件内容：

```
{
    "module_name": "xkernel",
    "method_name": "NewAccount",
    "args" : {
        "account_name": "1234098776890654",  # 说明：16 位数字组成的字符串
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 1},\"aksWeight\": {\"dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN\": 1}}"  # 这里的 address 改成自己的 address
    }
}
```

- Step1: 多重签名需要收集多个账号的签名，在测试环境中主要是需要交易发起者以及黄反服务的签名，因此修改 data/acl/addrs 文件，将需要收集签名的 address 写入该文件中。以创建合约账号为例，需要黄反服务背书，因此在该文件中写入黄反服务 address。

```
XDxkpQkfLwG6h56e896f3vBHhuN5g6M9u
```

- Step2: 生成创建合约账号的原始交易，命令如下：

```
./xchain-cli multisig gen --desc newAccount.json -H 14.215.179.74:37101 --fee 1000 --output rawTx.out
```

- Step3: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./rawTx.out --host 14.215.179.74:37101 --output complianceCheck.out
```

- Step4: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./rawTx.out --output my.sign
```

- Step5: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out --tx ./rawTx.out -H 14.215.179.74:37101
```

Note: Step5 中放签名的地方：第一个 my.sign 签名对应的是交易发起者 (Initiator)，第二个 complianceCheck.out 签名对应的是需要背书 (AuthRequire) 的地址，发起者签名和背书签名用空格分开，如果需要多个账号背书，那么多个背书签名用, 隔开，且签名顺序需要与 data/acl/addrs 中的地址顺序一致。

创建成功后，你可以通过这个命令去查看你刚才创建的合约账号：

```
./xchain-cli account query --host 14.215.179.74:37101
```

38.4.3 设置合约账号 ACL

Note: 前置条件：将合约账号以及合约账号下的有权限的 AK 以合约账号/address 形式以追加方式存放到 data/acl/addrs

- Step1: 生成设置合约账号的原始交易，命令如下：

```
./xchain-cli multisig gen --desc accountAclSet.json -H 14.215.179.74:37101 --fee 10 --
↪output rawTx.out
```

- Step2: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./rawTx.out --host 14.215.179.74:37101 --output␣
↪complianceCheck.out
```

- Step3: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./rawTx.out --output my.sign
```

- Step4: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out,my.sign --tx ./rawTx.out -H 14.
↪215.179.74:37101
```

accountAclSet.json 模版如下：

```
{
    "module_name": "xkernel",
    "method_name": "SetAccountAcl",
    "args" : {
        "account_name": "XC1234098776890654@xuper",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 1},\"aksWeight\": {\"ak1\": 1}}"
    }
}
```

38.5 合约操作

Note: 合约操作包括编译、部署、调用、设置合约接口权限，目前 XuperChain 支持的合约语言包括 C++，Go，我们以 C++ 中的 counter.cc 为例，以此说明合约相关操作。

38.5.1 合约编译

Note: 合约编译是指将合约编译成二进制形式

例子：C++ 版本的 counter.cc，counter.cc 存放路径为 contractsdk/cpp/example 预置条件：安装 docker

```
cd contractsdk/cpp
sh build.sh
```

到当前目录 build 里，将编译好的合约二进制 counter.wasm，重新命名为 counter，放到某个目录下，比如笔者的目录是./output/

38.5.2 合约账号充入测试资源

合约部署需要合约账号才能操作，因此会消耗合约账号的测试资源，需要开发者先将个人账号的测试资源转一部分给合约账号。(注意，目前不支持合约账号的测试资源再转出给个人账号，因此请按需充入测试资源。)

- Step1: 生成测试资源转给合约账号的原始交易数据，命令如下：

```
./xchain-cli multisig gen --to XC1234098776890651@xuper --amount 150000 --output rawTx.
↪out --host 14.215.179.74:37101
```

其中: –amount 是转出的测试资源数量，–to 是接收测试资源的账号名。如果转出方不是./data/keys 下的默认地址，则可以使用–from 指定转账来源账号，并将该来源地址的签名在 multisig send 时写在 Initiator 的位置。

- Step2: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./rawTx.out --output complianceCheck.out --host 14.215.
↪179.74:37101
```

- Step3: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./rawTx.out --output my.sign
```

- Step4: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out --tx ./rawTx.out -H 14.215.179.
↪74:37101
```

- Step5: 查询合约账号的测试资源数额，确定转账成功：

```
./xchain-cli account balance XC1234098776890651@xuper -H 14.215.179.74:37101
```

38.5.3 合约部署

Note: 部署合约的前提条件是先创建一个合约账号，假设按照上述步骤已经创建了一个合约账号 `XC1234098776890651@xuper` ，并且对应的合约账号有充裕的测试资源前置条件：将合约账号以及合约账号下的有权限的 AK 以 **合约账号/address** 形式以追加方式存放到 data/acl/addrs

---

- Step0: 合约部署需要在交易中写入满足合约账号 ACL 的背书 AK 签名，为了表示某个 AK 在代表某个账号背书，超级链中定义了一种 AK URI，例如 *dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN* 代表 *XC1234098776890651@xuper* 这个合约账号，那么这个背书 AK 的 AK URI 可以写成：**XC1234098776890651@xuper/dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN** 。

以此为例，背书 AK URI 需要同时包含黄反服务和合约账号，因此需要将 data/acl/addrs 文件改成：

```
XDxkpQkfLwG6h56e896f3vBHhuN5g6M9u
XC1234098776890651@xuper/dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN
```

Step1: 生成部署合约的原始交易，命令如下：

```
./xchain-cli wasm deploy --account XC1234098776890651@xuper --cname counter -H 14.215.
↪179.74:37101 -m ./counter --arg '{"creator":"xchain"}' --output contractRawTx.out --
↪fee 137493
```

Step2: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./contractRawTx.out --host 14.215.179.74:37101 --output␣
↪complianceCheck.out
```

Step3: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./contractRawTx.out --output my.sign
```

Step4: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out,my.sign --tx ./contractRawTx.out -
↪H 14.215.179.74:37101
```

38.5.4 合约调用

编译合约，部署合约的目的都是为了能够在区块链系统上运行智能合约，本小节说明如下调用合约。

- Step1: 生成合约调用的原始交易，命令有下面两种实现方式：

```
./xchain-cli multisig gen --desc counterIncrease.json -H 14.215.179.74:37101 --fee 85 --
↪output rawTx.out
# 或者这样
./xchain-cli wasm invoke -a '{"key":"counter"}' --method increase counter -H 14.215.179.
↪74:37101 --fee 85 -m --output rawTx.out
```

- Step2: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./rawTx.out --host 14.215.179.74:37101 --output␣
↪complianceCheck.out
```

- Step3: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./rawTx.out --output my.sign
```

- Step4: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out --tx ./rawTx.out -H 14.215.179.
↪74:37101
```

counterIncrese.json 模板，如下：

```
{
    "module_name": "wasm",
    "contract_name": "counter",
    "method_name": "increase",
    "args":{
        "key":"counter"
    }
}
```

38.5.5 设置合约接口 ACL

Note:

有这么一种场景：合约账号 A 部署了 counter 合约，希望只有拿到特定签名的用户才能调用 counter 的 increase 方法，因此 XuperChain 提供对智能合约某个方法进行权限设置
前置条件：将合约账号以及合约账号下的有权限的 AK 以合约账号/address 形式以追加方式存放到 data/acl/addrs

- Step1: 生成设置合约方法权限 (ACL) 的原始交易，命令如下：

```
./xchain-cli multisig gen --desc methodAclSet.json -H 14.215.179.74:37101 --fee 10 --
↪output rawTx.out
```

- Step2: 向黄反服务获取签名，命令如下：

```
./xchain-cli multisig get --tx ./rawTx.out --host 14.215.179.74:37101 --output␣
↪complianceCheck.out
```

- Step3: 自己对原始交易签名，命令如下：

```
./xchain-cli multisig sign --tx ./rawTx.out --output my.sign
```

- Step4: 将原始交易以及签名发送出去，命令如下：

```
./xchain-cli multisig send my.sign complianceCheck.out,my.sign --tx ./rawTx.out -H 14.
↪215.179.74:37101
```

methodAclSet.json 的模版，如下：

```
{
    "module_name": "xkernel",
    "method_name": "SetMethodAcl",
    "args" : {
        "contract_name": "counter",
        "method_name": "increase",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 1},\"aksWeight\": {\
↪"TqnHT6QQnD9rjvqRJehEaAUB3ZwzSFZhR\": 1}}"
    }
}
```

38.6 FAQ

Q 为什么测试环境现在不开放全节点 P2P 账本同步?

A 目前超级链仍然处于高速迭代期，为了保证 bug 能够得到即时修复更新，我们暂时未开放外部 P2P 节点加入测试环境的功能，但用户通过 GRPC 接口已经能体验到测试环境的大部分功能。当然，我们会在测试环境运行一段时间后，开放 P2P 节点加入乃至开放外部节点成为超级节点，具体时间目前还没有确定，请大家继续关注。

Q 测试环境中的测试资源可以转给别的个人账号吗?

A 不能，测试资源仅供在测试环境上进行超级链体验、智能合约开发测试使用，用户可以通过加入测试计划免费获得，用户获得的测试资源无法转给其他任何个人账号。

Q 测试环境为什么所有交易都需要黄反服务签名?

A 超级链测试环境上的数据是所有用户透明可见的，为了保证所有用户的体验，我们会对每个 transaction 中的数据进行合规性检测，尽量避免涉嫌黄反内容上链。用户违规发起涉嫌黄反内容的 transaction 引起的任何后果，都需要自行承担。请各位测试用户也从自身做起，保障一个干净和谐的网络环境。

Q 编译 cpp 合约出现 "*Post http:///var/run/docker.sock/v1.19/containers/create: dial unix /var/run/docker.sock: permission denied. Are you trying to connect to a TLS-enabled daemon without TLS?*" 是什么原因?

A 这可能是因为用户安装 docker 后，没有创建 docker 用户组，或者当前运行的系统账号不在 docker 用户组中，可以尝试下面的命令:

```
sudo groupadd docker
sudo usermod -aG docker ${USER}  // 此处${USER}为你编译合约时使用的 linux 账号
service docker resteart
```

CHAPTER 39

操作指导

39.1 如何获取 XuperChain

目前 XuperChain 已经发布了 2 个版本，最新版本为 v3.2，可以在 github 获取发布包

- XuperChain v3.2
- XuperChain v3.1

39.2 如何升级软件

当版本升级时，需要更新为新版本的代码，并重新编译，然后将 plugins 文件夹，二进制文件 xchain，xchain-cli 全部替换后全部重新启动即可，注意多节点模式下需要先启动 bootNodes 节点。

39.3 配置文件说明

XuperChain 的配置文件默认读取有 3 个优先级：

- 默认配置：系统中所有配置项都有默认的配置信息，这个是优先级最低的配置；
- 配置文件：通过读取配置文件的方式，可以覆盖系统中默认的参数配置，默认的配置文件为./conf/xchain.yaml；
- 启动参数：有一些参数支持启动参数的方式设置，该设置方式的优先级最高，会覆盖配置文件中的配置项；

```
log:
filepath: logs // 日志输出目录
filename: xchain // 日志文件名
console: true //是否答应 console 日志
level : trace // 日志等级, debug < trace < info < warn < error < crit
tcpServer:
port: :57404 // 节点 RPC 服务监听端口
p2pv2:
port: 47404 // 节点 p2p 网络监听的端口
bootNodes: /ip4/127.0.0.1/tcp/47401/p2p/QmXRyKS1BFmneUEuwxmEmHyeCSb7r7gSNZ28gmDXbTYEXK  /
↪/ 节点加入网络链接的种子节点的 netUrl
miner:
keypath: ./data/keys //节点 address 目录
datapath: ./data/blockchain //账本存储目录
utxo:
cachesize: 5000 //Utxo 内存 cache 大小设置
tmplockSeconds: 60 //GenerateTx 的临时锁定期限, 默认是 60 秒
```

39.4 core 目录各文件说明

模块	功能及子文件说明
acl	acl 查询 account_acl.go 查询合约账号 ACL 的接口定义 acl_manager.go 查询合约账号 ACL, 合约方法 ACL 的具体实现 contract_acl.go 查询合约方法 ACL 的接口定义
cmd	XuperChain 命令行功能集合，比如多重签名、交易查询、区块查询、合约部署、合约调用、余额查询等
common	公共组件 batch_chan.go 将交易批量写入到 channel 中 common.go 获取序列化后的交易/区块的大小 lru_cache.go lru cache 实现 util.go 去重 string 切片中的元素
config	系统配置文件 config.go 包括日志配置、Tcp 配置、P2p 配置、矿工配置、Utxo 配置、Fee 配置、合约配置、控制台配置、节点配置、raft 配置等
consensus	共识模块 base 共识算法接口定义 consensus.go 可插拔共识实现 tdpos dpos 共识算法的具体实现 single single 共识算法的具体实现
contract	智能合约 contract.go 智能合约接口定义 contract_mgr.go 创建智能合约实例 kernel 系统级串行智能合约 proposal 提案 wasm wasm 虚拟机
core	xchaincore.go 区块链的业务逻辑实现 xchainmg.go 负责管理多条区块链 xchainmg_validate.go 对区块、交易、智能合约的合法性验证业务逻辑 sync.go 节点主动向其它节点同步区块业务逻辑 xchaincore_net.go 通过广播形式向周围节点要区块 xchainmg_net.go 注册接收的消息类型 xchainmg_util.go 权限验证
crypto	密码学模块 account 生成用户账号 client 密码学模块的客户端接口 config 定义创建账号时产生的助记词中的标记符的值，及其所对应的椭圆曲线密码学算法的类 hash hash 算法 sign 签名相关 utils 常用功能
global	全局方法/变量 common.go 全局方法 global.go 全局变量
kv	存储接口与实现 kvdb 单盘存储 mstorage 多盘存储
ledger	账本模块 genesis.go 创世区块相关实现 ledger.go 账本核心业务逻辑实现 ledger_hash.go 账本涉及的 hash 实现，如生成 Merkle 树，生成区块 ID
log	日志模块 log.go 创建日志实例
p2pv2	p2p 网络模块 pb p2p 网络消息的 pb 定义 config.go p2p 网络配置 filter.go p2p 网络节点过滤实现 server.go p2p 网络对外接口实现 stream.go p2p 网络流的定义与实现 subscriber.go p2p 网络消息订阅定义与实现 util.go p2p 网络的全局方法 handlerMap.go p2p 网络消息处理入口 node.go p2p 网络节点定义与实现 stream_pool.go p2p 网络节点对应的流定义与实现 type.go p2p 网络对外接口定义
permission	权限验证模块 permission.go 权限验证的业务逻辑实现 ptree 权限树 rule 权限模型 utils 通用工具
plugin-mgr	插件管理模块 pluginmgr.go 插件管理的业务逻辑实现 xchainpm.go 插件初始化工作
replica	多副本模块 replica.go 多副本 raft 业务逻辑实现
server	util.go 通用工具实现，如获取远程节点 ip
xuper3	contract contract/bridge xuperbridge 定义与实现 contract/kernel 系统级合约 (走预执行) contract/vm.go 虚拟机接口定义
xu-	xmodel xmodel 实现 xmodel/pb 版本数据 pb 定义 xmodel/dbutils.go xmodel 通

CHAPTER 40

视频教程

40.1 如何搭建及使用超级链网络

40.1.1 如何快速建链

40.1.2 如何玩转 TDPOS

40.1.3 如何部署智能合约

40.2 带你轻松上手超级链测试环境

40.2.1 如何创建合约账号

40.2.2 智能合约的使用

CHAPTER 41

指令介绍 (API)

41.1 节点 rpc 接口

详细见：pb/xchain.proto

API	功能
rpc createAccount(AccountInput) returns (AccountOutput)	创建公私钥对
rpc GenerateTx(TxData) returns (TxStatus)	生成交易
rpc PostTx(TxStatus) returns (CommonReply)	对一个交易进行验证并转发给附近网络节点
rpc BatchPostTx(BatchTxs) returns (CommonReply)	对一批交易进行验证并转发给附近网络节点
rpc QueryAcl(AclStatus) returns (AclStatus)	查询合约账号/合约方法的Acl
rpc QueryTx(TxStatus) returns (TxStatus)	查询一个交易
rpc GetBalance(AddressStatus) returns (AddressStatus)	查询可用余额
rpc GetFrozenBalance(AddressStatus) returns (AddressStatus)	查询被冻结的余额
rpc SendBlock(Block) returns (CommonReply)	将当前区块为止的所有区块上账本
rpc GetBlock(BlockID) returns (Block)	从当前账本获取特定区块
rpc GetBlockChainStatus(BCStatus) returns (BCStatus)	获取账本的最新区块数据
rpc ConfirmBlockChainStatus(BCStatus) returns (BCTipStatus)	判断某个区块是否为账本主干最新区块
rpc GetBlockChains(CommonIn) returns (BlockChains)	获取所有的链名
rpc GetSystemStatus(CommonIn) returns (SystemsStatusReply)	获取系统状态
rpc GetNetUrl(CommonIn) returns (RawUrl)	获取区块链网络中某个节点的 url
rpc GenerateAccountByMnemonic(GenerateAccountByMnemonicInput) returns (AccountMnemonicInfo)	创建一个带助记词的账号
rpc CreateNewAccountWithMnemonic(CreateNewAccountWithMnemonicInput) returns (AccountMnemonicInfo)	通过助记词恢复账号
rpc MergeUTXO (TxData) returns (CommonReply)	将同一个地址的多个余额项合并
rpc SelectUTXOV2 (UtxoInput) returns(UtxoOutput)	查询一个地址/合约账号对应的余额是否足够
rpc QueryContract(QueryContractRequest) returns (QueryContractResponse)	查询合约数据

41.2 开发者接口

详细见：contractsdk/pb/contract.proto

API	功能
rpc PutObject(PutRequest) returns (PutResponse)	产生一个读加一个写
rpc GetObject(GetRequest) returns (GetResponse)	生成一个读请求
rpc DeleteObject(DeleteRequest) returns (DeleteResponse)	产生一个读加一个特殊的写
rpc NewIterator(IteratorRequest) returns (IteratorResponse)	对迭代的 key 产生读
rpc QueryTx(QueryTxRequest) returns (QueryTxResponse)	查询交易
rpc QueryBlock(QueryBlockRequest) returns (QueryBlockResponse)	查询区块
rpc ContractCall(ContractCallRequest) returns (ContractCallResponse)	合约调用
rpc Ping(PingRequest) returns (PingResponse)	探测是否存活

CHAPTER 42

常见问题解答

42.1 系统相关

Q 超级链按照一般的分法，是属于联盟链还是公链？仅从源码看，XuperChain 是否更偏向于采用公链的理念设计，先记账后共识，只有一条主链？

A XuperChain 在设计中以模块化和可插拔作为基本原则之一，对共识、合约等核心组件都可高度定制开发。因此从这点来讲，XuperChain 并不是只为公链或联盟链设计，而是一种通用可定制的区块链架构。在多链方面，未来我们也会开源我们的跨链技术，请持续关注。

Q 超级节点技术中说到，“超级链实现了计算和存储分离的这样一个架构，即一个节点，它表面上是节点网络中的一个节点，它的背后则是一个强大的、分布式的计算和存储集群。”，这段话怎么理解？

A 超级结点技术现在还不太完善，目前仅支持存储放到 nfs 上。我们的技术理想是，采用分布式存储 + 分布式计算调度技术，将超级结点的处理请求分发到这两个内部集群里面去。链内并行技术理论上是能分析出 DAG 并行处理依赖，然后将无依赖的请求调度计算集群里面分开计算。这样，以后超级结点其实是一个集群技术。

Q 超级链强调自己是 100% 自主研发，为什么要重新开发全新的区块链系统？新系统的可靠性和健壮性如何验证？

A 技术是无国界的，但是工程师是有国籍的。几乎每个流行的区块链项目都有一个稳定的核心开发团队圈子，为了不受制于人，尤其对于新兴技术，唯有自主研发才是正途。百度超级链的愿景是让信任的建立更加便捷，未来区块链会广泛应用在各行各业的大量业务，成为信任的连接器，当前我们已经在版权保护、司法存证、数据流通等领域落地了应用，可靠性健壮

性得到了验证。此次开源也是为了在更广泛的业务场景中应用，我们也会不断迭代代码、打磨系统，完善文档。

Q XuperChain 能跨链吗？XuperChain 上的各个平行链之间数据能相互访问交互吗？如果现有的 eth、eos 或者 fabric 要和 XuperChain 对接该如何对接呢？

A 超级链支持跨链，但此次开源的 XuperChain 里面没有包含跨链的组件，XuperChain 中的平行链不具备互操作性。跨链的功能后续将会开源在 XuperCross 这个项目里，目前已经实现对超级链平行链之间可以跨链，也可以实现对 fabric 的跨链，目前跨链技术仍在完善中。

Q 百度超级链在实现上和 Fabric 相比，具有哪些异同呢？

A 与 Fabric 相比，唯一的相似点是都有合约预执行阶段，其他很多地方都不一样。首先，Fabric 的 Ledger 是不支持分叉的，XuperChain 是支持的。Fabric 中数据的版本编码是和区块高度绑定的，因此它不支持在同一个区块内多笔事务修改同一个 Key。在智能合约开发方面，Fabric 没有合约虚拟机啊，不支持合约资源消耗控制，而且是通过 Docker 来实现粗粒度的资源限制。在网络方面，Fabric 是用的 RPC，并不能构建大规模的 P2P 网络。

Q 超级节点的服务器存储量是有限的，如果达到上限怎么办？是采取什么样的结构存储数据？

A 目前提供多盘存储能力，如果超过单机容量可以挂载 nfs，底层是扁平化的 key-value 存储，而且是可插拔的，目前支持 leveldb 和 badger 两种 key-value 存储引擎。如果有需要也可以开发分布式 key-value 的组件。

Q 超级链在部署方面的亮点和技术优势是什么？

A 部署亮点：不同于传统的联盟链系统，超级链具备全球化部署能力，节点通信基于加密的 P2P 网络，支持广域网超大规模节点，且底层账本支持分叉管理，自动收敛一致性，先进的 TDPOS 算法确保了大规模节点下的快速共识。主要技术优势是：高性能：通过原创的 XuperModel 模型，真正实现了智能合约的并行执行和验证，通过自研的 WASM 虚拟机，做到了指令集级别的极致优化。架构灵活：其可插拔、插件化的设计使得用户可以方便选择适合自己业务场景的解决方案，通过独有的 XuperBridge 技术，可插拔多语言虚拟机，从而支持丰富的合约开发语言。安全：内置了多私钥保护的账户体系，支持权重累计、集合运算等灵活的鉴权策略。保障了智能合约运行的安全和可控。

42.2 共识相关

Q 可插拔共识具体指什么？为什么设计成支持热更新的共识机制？

A 当前，每种共识都有各自的优缺点。POW 太消耗能源，而 DPOS 经常被质疑是不够去中心化的。所以，在 XuperChain 的代码中，我们实现了一种可插拔的共识机制，开发者自己实现相关接口，编译成插件就可以完成替换。并且我们认为在系统运行的不同阶段会对共识机制有不同的要求，例如某些场景下冷启动时可能使用 PoW 更合适，而当系统进入一个稳定状态时，可能使用 TDPoS 更合适，因此设计了一种共识热更新机制，通过提案投票，可以在区块链网络不停服的情况下完成共识更新。

Q 超级链目前支持哪些共识算法？推荐使用哪个共识算法？

A 超级链是可插拔共识，具有统一的共识接口，目前已经开源了包括 TDPoS、PoW、Chained-BFT、授权证明等多种共识机制。我们建议的是 TDPoS 共识，开发者也可以选择自己实现共识接口，扩展出更多的共识机制。

Q 此次开源的出来的共识中，最终确认时间是多少？

A 目前开源的共识算法有 TDPoS、PoW、Single、Chained-BFT。这几种共识算法的出块时间都是可配置的。一般来说 Single 是强一致模型，可以认为是即可生效；PoW 的交易确认时间根据网络情况而定，一般是 6 个块以上；如果采用 TDPoS+Chained-BFT，则交易确认时间是 3 个块。

Q 现在超级链支持一条链共识算法无缝由 pow 切换为 tdpos 吗？

A 支持，共识的热更新是基于提案投票机制，可以通过提交一个 proposal，并在提案中设定好触发高度，当提案收到足够比例的投票后，系统会在之前设定的高度时统一触发共识升级，调用 consensus 的 updateConsensusMethod 方法，来实现共识切换。

Q 在 TDPOS 机制是怎么优化 DPOS 机制的分叉问题的？如果此时有一两个块由于网络延时其他节点未收到该广播，那是再发广播，等到其他节点确认收到该广播以后再切换 BP 节点？

A

主要是有以下 2 点：

1. 首先在时间调度算法切片上，我们有 3 个时间间隔配置，分别为出块间隔、轮内 BP 节点切换的时间间隔和切换轮的时间间隔，这个其实很简单，这样在切换 BP 节点时会可以让区块有足够的时间广播给下一个 BP;

2. 在网络拓扑上进行的优化，我们超级节点的选举是在每轮的第一个区块，并且提前一轮确定，这时我们网络层有足够的时间在 BP 节点之间建立直接的链接，这样也可以降低我们切换 BP 节点的分叉率。

Q 如果想扩展自定义共识算法，有相关文档吗？

A 目前没有具体的文档，如果感兴趣的话可以参考一下开源共识代码的实现，主要接口参考文件 consensus/base/consensusinterface.go。

42.3 性能相关

Q 请问百度超级链 8.7 万的 TPS 是在什么样的硬件环境下测得的？

A 目前百度超级链通过了国家工业信息安全发展研究中心评测鉴定所的测评，在性能测试中，百度超级链并发可达每秒 87000 笔交易；测试环境采用 TDPOS 共识，5 节点异步模式，万兆网络，每个节点 64 核 2.4GHZ CPU，NVME SSD 存储。

Q 链内并行在开源版本实现了吗？

A 实现了，在 func (xc *XChainCore) PostTx 中。节点收到一个 tx 后，除了校验密码学签名，还会校验其依赖的 Input 哈希指针（指向依赖的其他 tx output）是否有效（up to date），如

果有效则加载数据构造合约执行环境，然后重新执行合约，看输出的数据是否和 tx 中声明的写集合 “TxOutputExts “一致。这整个过程都是无锁的，因此能并行，利用多核能力。验证通过后，最后调用 doTxInternal 写入账本，这个过程是有锁的，会再次 check 哈希指针的有效性，因为前面放锁了。简单而言，是一种乐观锁实现。

Q XuperChain 引入链内并发及多版本事务处理技术，那么它又是如何保证事务的原子性及时序性？是否与分布式数据库的事务相似呢？

A 通过将一个事务涉及的数据变更打包在一个底层 KV 数据库的 Batch 写，保证其原子性。事务的处理时序是通过事务的引用关系来定序：DAG 的拓扑序。在经典的 UTXO 模型中，事务声明了 “资金引用”，而 XuperChain 的事务模型中，有两种引用：资金引用和数据引用，通过数据引用来支持通用的智能合约。链内并行原理：节点收到一个事务后，除了校验密码学签名，还会校验其依赖的 Input 哈希指针（指向依赖的其他事务的 output）是否有效（up to date），如果有效则加载数据构造合约执行环境，然后重新执行合约，验证输出的数据是否和其声明的 Output 一致。这整个过程都是无锁的，因此能并行，利用多核能力。验证通过后，事务的 Output 写入账本，这个过程是有锁的，写入前会再次检查一次哈希指针的有效性。整体上的原理和分布式数据库的 MVCC 并发控制有相似之处。

Q 链内并行技术中，如果多个并发交易具有时序性，是否会产生死锁问题？为什么？

A 不会有死锁。从前面对链内并行的原理分析也可以看到，我们是采用的 “乐观锁” 的机制，有点类似 CPU 的硬件同步原语 Compare and swap。事务之间最差情况是冲突导致单次提交失败，不会死锁。超级链中事务的提交分为两阶段，预执行 (PreExec) 和提交 (PostTx)。预执行阶段合约对账本是只读 (Read-Only) 的，预执行结果会返回事务的 “读写集合”，其中读集合描述了事务依赖数据各个 Key 的 Hash 指针，这个 Hash 指针指向已经成功提交的事务的 Output 域。客户端将读写集和自己的签名组装起来，开始第二阶段：提交，节点验证成功后，事务输出才写入生效，进入待上链状态。如果提交失败，客户端可以返回第一阶段重新开始。

Q 请问 DAG 技术有较为详细的文档么？或者在代码中哪个模块能看到相关实现？

A DAG 并发执行目前已经开源了块验证时的 DAG 识别和并发执行，具体可以参见开源代码中的 UTXO 模块。对于通过 PostTx 接收到的交易，目前还没有开源 DAG 并行架构，会在未来的版本中

42.4 合约相关

Q 超级链合约虚拟机开源了吗，兼容性怎么样？

A 已经开源了，目前主要支持 XVM 和 WAVM 两种虚拟机。XVM 开源在 XuperChain 主框架项目中，可以看下 XVM 的实现，基本过程是 wasm -> code injection -> c -> dylib. 然后 go 里面调用特定的几个导出函数符号执行。wasm import 的符号会在 c 里面体现为外部导入符号，然后通过 cgo，在 go 里面暴露出来。这个方式算是一种比较巧妙而且简洁的方式吧。另外 wavm 开源在 XuperChain/Wavm 这个项目中。

Q XuperBridge 是合约虚拟机和区块链账本的桥梁，它的好处当然有很多，可以统一接口，更低的耦合度，但同时也会限制一些灵活性，关于这点是如何处理的呢？

A XuperBridge 通过统一接口降低了将不同类型的虚拟机接入到 XuperChain 的难度，给予开发者更多的选择来开发 Dapp，而不限于某一类特定的编程语言。目前我们已经接入了 WASM 和 Docker 来满足不同场景的业务需求，后续我们会开放更多的接口来满足开发者的多样的开发需求。事实上，我们通过 XuperBridge 也已经支持了以太坊的 solidity 虚拟机，只是由于 License 问题，此次不便开源。

Q 合约预执行，与 Fabric 的 endorser 阶段策略类似吗？先生成 read/write set？

A 流程上大体类似，实现上是有差别的，例如数据版本定义等。Fabric 中数据的版本编码是和区块高度绑定的，因此它不支持在同一个区块内多笔事务修改同一个 Key，超级链中的版本类似于 UTXO，同一个块中可以对一个 key 进行多次修改，因此可以大幅提升交易性能。

Q 合约之间是否可访问，例如 A 账户 a1 合约里面存储的数据在 B 账户 b1 合约里面可以访问么？ 那用户 B 能调用 A 账户部署的合约吗？

A 目前夸合约调用还不支持，出于用户权限和数据安全考虑，合约数据属于受保护的私有数据，不能被其他合约直接使用，即使两个合约都属于同一账户。未来可以通过系统级跨合约调用的方式实现数据共享，目前这部分技术还没有开源，请持续关注超级链后续版本。但用户可以调用其他账户部署的合约，每个合约接口可以设置单独的 ACL 权限控制，因此合约所有者可以在合约接口 ACL 中配置哪些账户有权限访问该接口，这里的账户并不限于合约所有者。1. ACL 权限模型可以自由扩充定制。2. 如果合约接口创建后没设置 ACL，默认是 public，所有用户都可以访问。

Q 是否支持原生合约？原生合约有没有资源消耗机制？

A 超级链支持原生 (Native) 合约，原生合约可以在 docker 环境中执行，但原生合约目前不支持资源消耗控制机制。

Q 智能合约是什么时候触发执行的？只能由客户端触发吗，有没有可能就是在某个条件满足的时候自动触发呢？

A 简单说，智能合约是在用户调用的时候触发执行的。更详细得说，用户在客户端发起一个智能合约调用，服务端会为该智能合约调用创建 Context，然后将 Context 相关信息通过 XBridge 传给虚拟机，虚拟机通过调用 SyscallService 服务，来修改/获取智能合约的数据状态。

Q 是不是每种合约都需要有一条单独的链呢？

A 合约由虚拟机来管理，一条链上可以部署很多智能合约，不过每一个智能合约都需要有不同的合约名字。

42.5 账户权限相关

Q 普通账户和合约账户的区别是什么？

A 账户是指一种本地或自定义权限的链上标识符。本地标识符称为用户账户，通常分配一个公钥和一个私钥，并对应一个 address；自定义权限的链上标识符称为合约账户，通常分配一个或多个密钥或多个账户。

Q 什么是 AK？超级链中所说的 AK 集合是什么？

A AK 是超级链中对一个公私钥用户账户的称呼，可以理解为 Address，即通过一对公私钥转换来的一个唯一用户账户地址。AK 集这是权限系统模块中的一种权限模型，是指多个 AK 组成的具有一定逻辑规则的权限模型，例如多个 AK 之间满足”或”的关系等。

Q 超级链的权限设计为什么采用的是 ACL 设计模式而不是 ABAC 的设计模式？关于权限验证目前是只有 SIGN_THRESHOLD 和 SIGN_AKSET 的权限规则在使用吗？

A

1) ACL 更加简单，我们基于 ACL 实现了一套可扩展的权限模型，能够满足去中心权限系统的要求，同时又可扩展。而 ABAC 相对比较复杂，不易用；

2) 目前对外开源的是 SIGN_THRESHOLD 和 SIGN_AKSET 这两种权限模型。

Q 在权限系统的设计中身份账户的验证中会 buildPermTree 和 validatePermTree，这个 PermTree 的设计思想是什么这个 Perm 这个词是跟权限框架 casbin 中的 PERM 模型有关系吗？

A 在身份验证中，PermTree 主要是验证客户提供的账户与签名是否正确以及它们的权重是否满足对应的 ACL 要求。超级链的 PERM 与 casbin 并不相同，超级链的 PERM 指的是可扩展规则权限模型 (Permission with Extensible Rule Model)，超级链而账户/合约权限中可能嵌套其他账户，在验证账户的权重是否满足要求时，由于这种嵌套关系，权限会自然形成树形结构，每个节点都是一个账户或合约方法的权限 ACL，而每个节点的 ACL 可以使用不同的权限规则模型，节点的子节点是代表子节点账户对父节点的授权关系。

Q 在执行背书签名时默认读取 data/acl/addrs 文件，例如 multisig gen 时要读取这个文件的数据，请问这个文件的内容应该是什么？

A 这是一个文本文件，文件内容中每一行表示一个需要签名的账户，是用于多重签名的地址，每个地址用换行分割。如果需要其他个人账户的授权，那么把个人账户的 address 作为一行写入这个文件中；如果需要合约账户授权，则需要使用” 合约账户/个人账户 address” 的写法，表示需要某个合约账户 ACL 中的某个个人账户签名。

42.6 使用问题

Q 平行链是什么角色有权限创建？创建平行链的权限白名单是写在创世块中的么？

A 目前创建平行连有两种方法：一种直接通过 xchain-cli 的 createChain 命令，这种没有权限限制，只在本机创建；另一种是调用创建链的合约，这种情况可以在全网创建平行链，节点可以通过配置白名单的方式指定哪些用户能调用创建链的合约。创建平行链的权限白名单目前不在创世块中，而是在节点配置文件中。这么做的初衷是使每个节点有能力通过配置决定

只托管符合自己要求的平行链。具体节点配置可以参考：https://github.com/xuperchain/xuperchain/blob/master/core/conf/xchain.yaml#L52

Q 自己搭链有 root 链和平行链之分么，还是自己搭的只是 xuperchain 的平行链？通过 xchain-cli 的 createChain 命令能在本地创建多条平行链吗？

A 自己搭建的网络首先需要创建主链 (代码里叫 xuper 链)，然后再创建其他平行链，主链具有部分管理其他链的能力，例如创建平行链的系统合约是在主链中执行。通过 xchain-cli 的 createChain 命令可以在本地创建任意多条链。

Q 搭建多节点网络时，其他节点需要在配置里面指定根节点的网络地址吗？

A 超级链的 P2P 网络具有自动路由功能，因此只要在配置中指定任意一个已经在网络中的节点地址即可。

Q 超级链中的 rootBlockid 和 tipBlockid 是什么意思，他们的关系是什么？

A rootBlockid 和 tipBlockid 都是指代区块的 id，区块的 id 就是区块的唯一标识，通过两次 SHA256 生成。而 rootBlockid 特指创世区块 id，即当前链的第一个区块的 id；tipBlockid 表示当前链主干分支最新的区块的 id。也可以从 rootBlockid 和 tipBlockid 两个变量名称得知它们的意思，root 有”根”的意思，而 tip 有”尖端”的意思。

Q 客户端发送交易后，怎么查询是否成功上链。通过 querytx 判断 tx 状态还是有其他事件机制？支持事件通知么？

A 首先查询交易所在区块，然后查询区块是否在区块链主干上，如果交易所在区块在区块链主干上，表明此交易已经生效。事件通知暂不支持，后续有计划支持部分类型事务执行结果的事件通知。

Q RPC 的文档中没有看到创建账户相关的说明？

A 创建普通账户属于本地操作，该数据不上链，考虑到创建账户涉及到用户私钥传输，处于安全原因不提供 RPC 接口；创建合约账户属于系统合约调用，没有单独的 RPC 接口。

Q Windows 系统可以编译运行么？可以用 windows 安装虚拟机来实现 Linux 环境么？

A 目前暂不支持 Windows 的，要求是 Linux 或者 Mac OS。可以使用虚拟机实现。

42.7 其他问题

Q 密码学中椭圆曲线选择使用 P-256 的考虑？

A 首先，P-256 曲线目前依然在密码学界被广泛使用。其次，这次开源的是我们的基础版本，而在 xchain 的代码架构下，密码学相关的模块是插件化使用的，密码学 crypto 模块是可以独立研发并集成进开源框架中的。同时，在 ECDSA 之外，也已开源了多重签名、EDDSA、环签名等多种签名算法。在尚未开源的版本中，通过可变签名算法，我们已经支持国密/NIST 的多条椭圆曲线。最后，这些算法和曲线被支持混合使用，开发者可以自由选择他们认为安全的曲线和算法来保护自己的数字资产。所以，敬请期待百度 xchain 后续的密码学相关开源进展。

Q 超级链有密钥保护机制吗？怎么实现的？

A 有的，主要实现了两种方式：1) 通过支付密码在私钥加密后保存在本地。2）云端密钥保险箱。密钥保护功能在 SDK 里有实现，目前尚未开源。

Q 环签名、零知识证明等技术开源了么？他们的使用场景是什么？

A 环签名、零知识证明等技术适用于对隐私保护有较高需求的网络中。目前环签名已经在 crypto 模块中开源，可以实现对交易发起者信息的混淆，例如在论文评审场景里，实现评审者对论文的匿名打分等；零知识证明目前尚未开源。

Q 超级链有区块链浏览器吗？

A 暂时没有，在计划中，敬请关注。

Q 什么是 VAT，它的作用是什么？

A VAT (Verifiable Auto-generated Transaction, 可验证的自动生成事务) 是智能合约在运行过程中，根据需要自动生成的系统 Tx。这些 Tx 无法手动发起，结果也会上链，系统可验证。目前主要用在共识，提案等模块中，例如切换到 tdpos 共识，会自动生成候选人选举检票的 VAT 等。

Q 商用环境中，需要考虑数据隐私保护的问题，在这一块，xuper 怎么考虑，有成熟的方案吗？

A 在 xuper+ 的应用中，有 xuperdata，它是基于百度超级链、多方安全计算、数据隐私保护等技术的多企业数据安全协同计算方案。相关文档介绍：https://xuperchain.baidu.com/case/xuperdata

Q 百度超级链的多盘存储是什么实现原理？

A 超级链按照 goleveldb 的 storage.go 中的接口实现了自己的 storage 逻辑作为自己的 file-system backed storage，代码可参考 multi_disk_storage.go。具体实现中，leveldb 的 sst 文件按照编号均匀散列放置在多块盘上，如果盘数扩容，第一次打开某个编号的 sst 文件的时候可能需要遍历尝试各个盘。另外，由于这个放置策略在 compact 的时候也生效，所以扩容的场景下，运行一段时间后，sst 就会按新的路径均匀分布了。

CHAPTER 43

词汇表

词汇	定义
交易	对区块链进行状态更改的最小操作单元。通常表现为普通转帐以及智
区块 (创世区块、普通区块、配置区块、当前区块)	区块链中的最小确认单元，由零个或多个交易组成，一个区块中的交
账本	存储区块数据、交易数据。
UTXO	一种余额记账方式。用于存储账号的余额数据。
区块链	由若干个区块组成的 DAG。从数据结构上来说，区块链就是一个 DA
系统链	区块链网络中的第一条区块链实例。
平行链	从系统链衍生出来的子链，解决扩展性问题。
跨链	不同的区块链之间的通信操作，目的是实现区块链世界的价值互连，
账号 (用户账号、合约账号)	一种本地或自定义权限的链上标识符。本地标识符称为用户账号，通
密钥对 (公钥、私钥)	私钥以及由私钥生成的对应的公钥。私钥通常用于签名，公钥通常用
地址	与用户数据挂钩的最小单元。地址可以是合约账号，也可以是由公钥
签名 (普通签名、多重签名)	利用密码学哈希函数单向不可逆、抗碰撞特性，进行身份确认的一种
共识	一种确认区块中交易集正确性以及交易上链顺序的机制。
委托权益证明	一种共识算法，通过选举出区块链网络中有限节点作为代表并轮流出
智能合约 (系统级、用户合约)	一个由计算机处理、可执行合约条款的交易协议，其总体目标是满足
权限 (许可)	一种安全机制，通过评估签名权限来验证一个或一组操作是否被正确
虚拟机	智能合约的运行环境。通常包括合约上下文管理。
最长链	区块链中高度最大的分支。
DAG	有向无环图。
双重消费	同一份数据同时消费多次。

词汇	定义
最终一致性	存在某个时刻，整个系统达成一致状态。区块链满足最终一致性。
发起人	发起交易的账号，通常为用户账号或合约账号。
见证人	当选为当前周期内出块节点。
节点	区块链网络中的一个节点。
周期 (Epoch)	在委托权益证明共识算法中，一轮出块时间为一个周期。
提案	一种区块链系统级配置进行升级的机制。
查询	对区块链中的数据按照 key 进行查询。
对等网络	网络中的节点直接互联并交互信息，不需要借助第三方。
拜占庭 (拜占庭问题、拜占庭容错)	拜占庭问题：在一个需要共识系统中，由于作恶节点导致的问题。拜
状态转移系统	一个维护状态变化的系统。区块链通常被认为是一种状态转移系统，
读写集	用于支持智能合约并发执行的一种技术。

CHAPTER 44

超级链小课堂

第一期

什么是区块链共识算法?

一种通过确认交易上链顺序从而保证网络节点状态一致并且结合经济学中的博弈论让攻击者的攻击成本远远大于收益的算法。常见的区块链共识算法包括：POW,POS,DPOS,VBFT,SPOS 等。

TDPOS 是什么共识机制?

TDPOS 是百度在 DPOS 共识算法 (Delegated Proof of Staking，委托权益证明) 基础上自研改进的共识算法。区块链网络中的持币用户通过不定期选举出一小群节点，让这些节点进行区块创建、签名、验证和相互监督。TDPOS 在 DPOS 基础上做了很多优化和细节改造并且 TDPOS 作为百度超级链的默认共识算法。

第二期

什么是权限控制系统?

权限控制系统是为了约束资源查询/更新等能力而引入的一种系统要素。常见的权限控制系统包括：基于 ACL 的权限控制系统，基于 RBAC 的权限控制系统，基于 ABAC 的权限控制系统。

超级链的权限控制系统是什么?

百度超级链基于 ACL 权限控制模型实现了一套去中心化的权限控制系统，用于限制合约资源数据的访问/更新等能力，从而保障合约资源数据的安全。百度超级链自研的权限控制系统目前支持签名阈值策略、AK 集

签名策略等。

第三期

什么是公有链、联盟链以及私有链?

公有链是指所有人都可以参与的区块链，所有人都可以查询、发送、接收、验证交易等特性；联盟链亦称为许可链，参与方限制为联盟成员，参与方通常是各个企业、机构，金融、物流以及司法等是联盟链的主要应用场景；私有链是指参与方受限为企业、机构或者个体，主要用于解决可审计问题。

超级链（XuperChain）是什么?

超级链是百度自研的区块链底层技术，具有链内并行技术、可插拔共识机制、一体化智能合约等领先技术，支持隐私保护和安全机制，具有可兼容性、扩展性强、高性能等特点。不仅适用于面向企业以及机构之间的联盟链应用场景，同时适用于公有链应用场景。目前 XuperChain 已经应用于司法、版权、边缘计算、数据协同、溯源、电子政务、智慧医疗等多个领域。

第四期

什么是区块链账户?

区块链账户是指用于标识区块链网络中不同身份的一种机制。

百度超级链中的账户是什么?

百度超级链使用了两种类型的账户，包括普通账户以及合约账户。(1) 普通账户通常由账号 (address)、公私钥对以及余额组成。账号可以操作转账。(2) 合约账户相比普通账户而言，是一个更高级的账户类型，可以为多个账户角色进行权重分配，分配之后多个角色 (包括普通账户或者合约账户) 可以共同管理合约账户下面的数据资产 (包括智能合约、访问权限列表、余额等)。相比传统中心化的权限管理模式而言，不是由某个账户对数据资产进行专权化管理；账户权限一但创建之后，即实现了不同权重账户的多中心化资产管理模式，不同账户角色也可以通过共同签名来更改权限。

第五期

普通账户与合约账户的应用场景各有哪些?

普通账户的资产只归属一个人，而合约账户会分配多个账户共同管理合约账户下面的资产 (包括智能合约、访问权限列表、余额等)。因此，在部署合约以及需要多方共同管理资产时，需要使用合约账户；其他应用场景下，普通账户与合约账户没有明显区别。

账户与账号的区别是什么?

账号是账户的组成部分，作为账户的唯一标识符，有固定的生成规则。比如，在百度超级链中，dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN 是一个有效的普通账户的账号；XC1111111111111111@xuper 是一个有效的合约账户的账号。

第六期

P2P 网络是什么?

P2P 网络也称为对等网络，它是一种网络技术和网络拓扑结构。目前 P2P 技术已经应用于流媒体、点对点通讯等领域，也就是我们常说的 BT 工具和电驴，比较常见的 P2P 网络协议有 BitTorrent、ED2K 等。

区块链 P2P 网络是什么?

区块链 P2P 网络主要用于区块链节点之间数据传输和广播、节点发现和维护。因此，区块链 P2P 网络主要解决数据获取以及节点定位两个问题。节点发现和局域网穿透技术主要解决的是节点定位问题，节点交互协议主要解决的是数据获取问题。节点发现主流协议有 Gossip 以及 KAD，局域网穿透协议主要有 NAT。

第七期

混盘技术是什么?

混盘技术也称为多盘技术，将多个磁盘从逻辑上当作一个磁盘来处理，主要用于解决只支持本地单盘场景下数据库空间不够的问题（即扩展性问题），比如被广泛使用的 LevelDB。目前对 LevelDB 的多盘扩展技术，大部分是采用了多个 LevelDB 实例的方式，也就是每个盘一个单独的 LevelDB 实例。

超级链自研的混盘技术

多个 LevelDB 实例的方式好处是简单，不需要修改 LevelDB 底层代码，缺点是牺牲了多行原子写入的功能。在区块链的应用场景中，需要保证多个写入操作是原子性的特性。超级链改造 LevelDB 存储引擎本身，在引擎内部完成了数据文件的多盘放置，能够确保所有写入更新操作的原子性，从而能够满足区块链场景的交易原子性要求。

第八期

平行链是什么?

平行链是相对于单链而言的，在只有一条链情况下，所有交易都由根链验证、执行、存储、打包到区块，在交易量高的情况下，存在吞吐低，延时高的问题。为了解决这类扩展性问题，从根链衍生出多条子链，各个子链打包自己链上的交易，账本、共识算法、网络等模块都可以相互独立。

百度超级链中的平行链是什么?

百度超级链通过系统合约方式创建平行链，平行链之间相互独立，拥有各自的账本和共识算法等模块，目前平行链之间共享 p2p 网络。不同的业务可以跑在不同的平行链上，起到了业务隔离效果，在使用平行链时，需要通过–name 指定需要操作的平行链的名字。

第九期

用户可以通过哪些客户端接口访问百度超级链?

1. xchain-cli，交互式命令行工具，直接使用 xchain-cli 即可发起操作，本质是通过 rpc 接口与服务端进行交互，可以从 xuperchain 库中获取；xchain-cli 具有丰富的命令，包括创建账户、普通转账、合约部署以及调用、提案、投票、链上交易以及区块查询等功能；比如./xchain-cli transfer –to bob –amount 1 就可以发起一笔向 bob 转账 1 个 utxo 的交易，更多命令可以通过./xchain-cli -h 获取；
2. SDK：提供一系列的 API 接口，用户可以基于提供的 API 接口做定制化的操作，相比 xchain-cli 更灵活；目前开源的 SDK 包括 Go SDK，Python SDK，C# SDK；
3. curl：直接通过 curl 命令来发起查询、构造交易等操作，客户端除了 curl，不依赖任何三方库，此时需要服务端启动 xchain-httpgw，然后通过 curl http://localhost:8098/v1/get_balance -d ‘{ “bcs” :[{ “bcname” :” xuper” }, “address” :” bob”], 即可查询 xuper 链上 bob 的余额信息

第十期

Gas 在区块链中的作用是什么?

Gas 是一种资源消耗计量单位，比如执行智能合约时消耗的资源数量。用于奖励矿工并防止恶意攻击，是区块链生态系统可持续发展的重要因素。通常，Gas 由各种可衡量资源按照特定比例累加而成。

百度超级链中，如何计算 Gas?

百度超级链中采用了如下可衡量资源：CPU，Mem，Disk，XFee。其中，CPU 是指一个合约执行时消耗的 CPU 指令，Mem 是指一个合约上下文消耗的内存大小，Disk 是指一个合约上下文的磁盘大小，而 XFee 是一种特殊资源，主要针对系统合约消耗的资源，比如创建一个合约账号、设置合约方法的 ACL 需要消耗的资源。Gas 计算公式为：Gas = CPU * cpu_rate + Mem * mem_rate + Disk * disk_rate + XFee * xfee_rate，其中 cpu_rate，mem_rate，disk_rate，xfee_rate 为资源与 Gas 的兑换比例。

第十一期

区块链的链上治理是指什么?

区块链的链上治理是指在一个涉及很多利益方的区块链网络中，为了升级系统共识参数并保证区块链网络持续演进的链上解决方案（比特币和以太坊就因为系统共识参数升级分歧发生过多次硬分叉）。

百度超级链的链上治理是如何做的?

百度超级链提出一种提案机制，首先，提案发起人会发起一笔修改系统共识参数的提案交易；然后，提案发起人将提案交易通过链外方式（比如邮件列表或者论坛、线下聚会等）告诉社区，对提案作进一步解释，并号召大家投票；之后，区块链网络中的用户可以对该提案交易进行投票；最后，如果投票数量超过提案交易中规定的最低票数，该提案交易就会生效。

第十二期

区块链中，虚拟机的作用是什么?

虚拟机为智能合约提供了一个对底层透明的执行环境，主要工作包括指令解析、链上交互、Gas 计算等。目前常见的虚拟机包括 EVM，基于 WASM 的虚拟机等。

超级链虚拟机是如何执行合约的?

XVM(Xuper Virtual Machine，超级链虚拟机) 目前支持在预编译模式以及解释模式下执行智能合约。1. 预编译模式下：在合约部署时，XVM 会将 wasm 指令编译成本地机器可以运行的指令 (由 wasm2c 来做，主要工作包括将 wasm 转换成 c、系统调用、Gas 统计等功能)；在合约调用时，直接执行相应的指令即可。因此，预编译模式下，合约部署需要消耗时间，通常为数秒；而合约调用因为不需要再次做指令映射，执行效率高；2. 解释模式下：在合约调用时，XVM 对 wasm 指令挨个解释执行 (主要工作包括对 wasm 指令进行解释执行、Gas 统计等功能)。因为在合约部署时不需要做指令映射，合约部署较快；在合约调用时，需要对 wasm 指令挨个做指令映射，执行效率低。

第十三期

区块链中常见的安全问题有哪些?

区块链中常见的攻击包括 DDoS 攻击、女巫攻击、整数溢出、可重入攻击、拜占庭攻击等，主要体现在网络层、智能合约层、共识层、数据层等方面。

超级链做了哪些安全工作?

超级链主要在密钥安全、网络安全、数据安全、共识安全以及智能合约安全等方面做了系列工作。1. 密钥安全方面，支持密钥加密存储、助记词恢复、密钥备份等能力；2. 网络安全方面，通过 TLS 进行数据加密传输，通过 CA 实现联盟准入机制，节点身份认证以及分层网络路由保护机制，来源 IP 数量限制等；3. 数据安全方面，除了基本密码学机制外，还实现账号与权限系统细粒度区分数据访问权限；4. 共识安全方面，实现了 bft 组件，能够抵抗拜占庭节点攻击；5. 智能合约方面，通过 wasm 实现指令级资源审计，屏蔽对底层存在较大风险的系统调用接口，保证应用层安全。

第十四期

关于 UTXO 的命令有哪些?

1. 查询用户 UTXO 面额：./xchain-cli account balance；
2. 查询用户 UTXO 详细信息：./xchain-cli account list-utxo；可以通过该命令查看哪些 utxo 当前可用，哪些 utxo 当前被锁定以及哪些 utxo 当前被冻结；
3. 合并 UTXO：./xchain-cli account merge；可以通过该命令将用户多个 utxo 合并，来解决因 UTXO 太零散导致交易过大问题；
4. 拆分 UTXO：./xchain-cli account split；可以通过该命令将用户的一个 UTXO 进行拆分，解决用户无法同时发起多笔交易的问题；

第十五期

超级链开放网络是什么?

超级链开放网络是基于百度完全自主研发的开源技术搭建的区块链基础服务网络，由分布在全国的超级联盟节点组成，符合中国标准，为用户提供区块链应用快速部署和运行的环境，以及计算和存储等资源的弹性付费能力，直接降低用户部署和运维成本，让信任链接更加便利。

超级链开放网络有哪些优势?

1. 自主安全高可靠：基于百度完全自主研发且开源的区块链技术搭建，满足中国区块链标准要求；
2. 灵活便捷低门槛：无需建链即可运用区块链技术，丰富的合约模板和强大的功能组件，降低使用门槛；
3. 弹性付费成本低：具备计算和存储等资源的弹性付费能力，可以实现按需按量灵活计费，一分钱即可用；
4. 节点开放公信强：由分布全国的超级联盟节点构成，面向社会开放节点接入，具备极强的公信力；

第十六期

超级链有哪些交易类型?

超级链主要包括三种交易类型：1. 普通转账交易：基于用户 utxo 进行转账，此类交易包含 utxo 的引用关系，即 TxInput 和 TxOutput，能够并行执行；2. 二代合约交易：主要用于修改系统共识参数，比如升级共识算法、提案等操作，此类交易执行顺序与区块高度绑定，只能串行执行；3. 三代合约交易：采用两阶段提交，首先通过预执行获取合约数据读写集，然后组装交易并转发给记账节点执行，此类交易执行顺序与区块高度无关，能够并行执行。

XuperChain 如何统一 UTXO 和智能合约模型?

UTXO 模型主要用于存储用户的 utxo 数据，一般适用于普通转账交易；而智能合约存储模型主要用于存储用户合约相关数据。本质上，这两种存储模型都是存储用户数据并且包含数据版本依赖关系。因此，超级链自研一套通用的存储模型 XuperModel，基于 key 记录用户数据的依赖关系，实现 UTXO 和智能合约底层数据存储模型的统一。而比特币和以太坊底层存储模型不同，导致它们无法做到兼容。

第十七期

超级链中，交易执行支持哪些模式?

超级链支持三种交易执行模式，分别为同步模式、纯异步模式以及异步阻塞模式。1. 同步模式：客户端发起一笔交易并等待交易执行结果；xchain 节点更新交易状态时，加锁，锁内只能同时更新一个交易状态；2. 纯异步模式：客户端发起一笔交易并直接返回；xchain 节点积攒批量交易，在更新交易状态时，加锁，锁内同时更新批量交易状态；3. 异步阻塞模式：客户端发起一笔交易并等待交易执行结果；xchain 节点积攒批量交易，在更新交易状态时，加锁，锁内同时更新批量交易状态；

如何使用同步、纯异步以及异步阻塞模式?

三种模式是互斥的，默认采用同步模式。在 xchain 节点启动时，通过 flag 来选择。通过 nohup ./xchain –asyncBlockMode=true & 启动异步阻塞模式；通过 nohup ./xchain –asyncMode=true & 启动纯异步模式。

第十八期

如何参与超级链的开发?

1. 可以通过阅读超级链任意开源项目，包括源代码、文档，以便了解当前的开发方向；
2. 找到自己感兴趣的功能或者模块；
3. 实际开发时需要自测功能是否正常、性能是否符合预期，并运行 make & make test 检查是否通过所有单测；
4. 发起一个 Pull Request，如果你的代码合入到主干后，就有机会运行在线上机器上。

如何提一个 PR?

1. 从 GitHub 上 fork 超级链的项目，并通过 git 拉取到本地；
2. 在本地用 git 新起一个分支，贡献的代码全部放在本地分支上；
3. 本地代码开发完毕，通过 git push 将本地分支代码提交至远程服务端；
4. 点击 GitHub 对应项目栏下面的 Pull Request 按钮，填写需要合并的分支以及被合并的分支，然后点击 create pull request 即发起一个 PR。

第十九期

超级链支持消息推送机制吗?

消息推送是指客户端主动向 xchain 节点订阅感兴趣的消息类型，当该类型的消息在链上被触发时，xchain 节点会主动将该行为推送给客户端；目前，超级链支持三种消息类型的推送，分别为区块消息、交易消息以及账户消息。1. 区块消息：用户可以订阅具有特定策略的区块，当链上触发这类区块时，会将消息主动推送给客户端；2. 交易消息：用户可以订阅具有特定策略的交易，当链上触发这类交易时，会将消息主动推送给客户端；3. 账户消息：当用户的余额发生变化时，会将消息推送给客户端。

如何使用超级链的消息推送机制?

目前，超级链的 master 分支支持消息推送机制。通过在 xchain.yaml 中增加 pubsubService 配置来启动事件推送服务。同时超级链提供了一个简单的客户端来订阅、接收自己感兴趣的消息。针对每种消息类型可用的策略可以参考 event.proto 文件。

第二十期

超级链支持群组功能吗?

群组是一种为了实现平行链之间隐私数据隔离，不同平行链只有指定节点才能参与区块打包、区块同步、区块/交易转发等能力的机制。

如何使用超级链的群组功能?

目前，超级链的 master 分支支持群组功能。在创世块配置文件中配置群组合约的相关参数，包括合约名、方法名等，并部署好群组合约 (超级链有群组合约的默认实现) 即可调用群组合约为平行链增加节点白名单，从而让平行链具备群组能力。

第二十一期

参与超级链贡献流程几何?

超级链的 XIPs（XuperChain Improvement Proposals）描述了一个需求的生命周期，包括需求收集、需求发布、功能设计、功能开发、功能测试以及最终发布；需求主要来源于工作组、社区以及合作伙伴；同时，工作组会对收集来的需求进行可行性、优先级评审；之后，开发者在 github 上选择感兴趣的需求进行设计并形成文档；再之后，即可以发起实际的代码开发流程；为了提高代码的质量，需要同时编写单元测试。

如何快速参与超级链的需求开发?

为了方便开发者更快地参与超级链的需求开发，超级链工作组已经将一些待开发的需求推到 github 。开发者可以选择感兴趣的需求直接进行开发。

CHAPTER 45

Indices and tables

- genindex
- modindex
- search