xuperunion-doc Documentation

xuper

2019 年 08 月 29 日

XuperUnion 介绍:

CHAPTER 1

简介

XuperUnion 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，XuperUnion 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。在账户安全方面，XuperUnion 内置了多私钥保护的账户体系，支持权重累计、集合运算等灵活的策略。

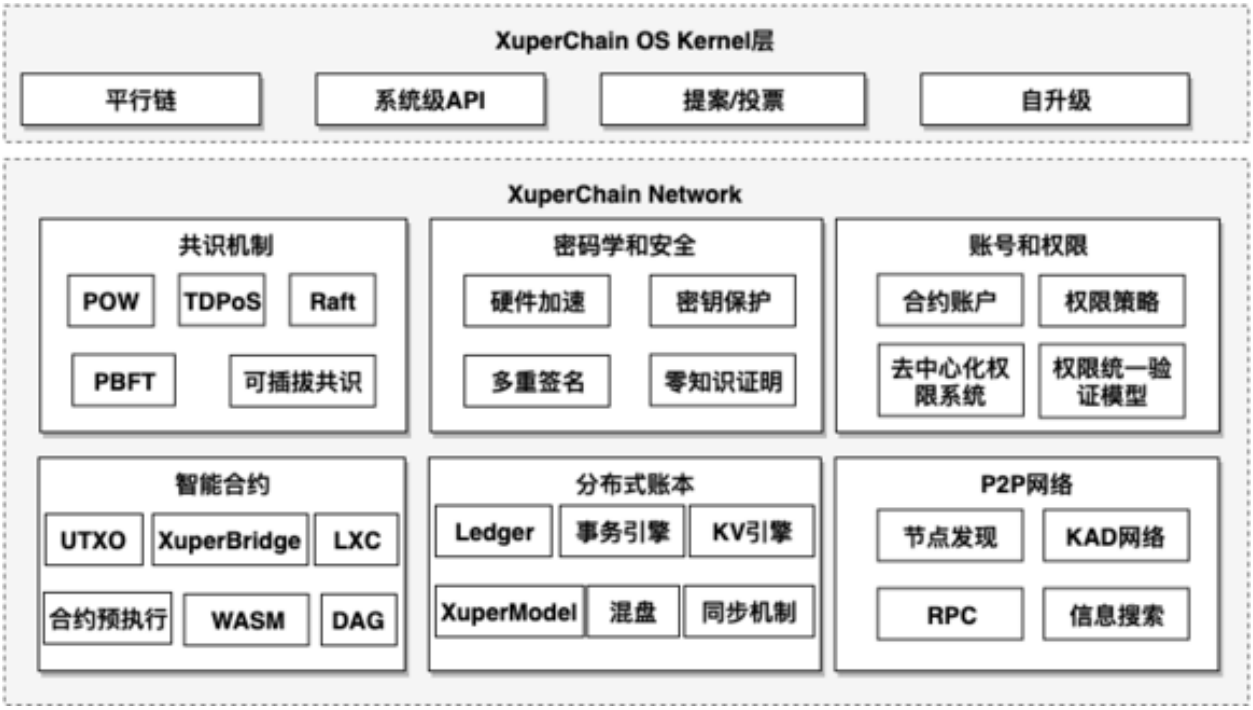

图 1: XuperChain 架构

CHAPTER 2

模块

模块	特性
存储	XuperUnion 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、账户余额、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。XuperUnion 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 Gas，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 XuperUnion 网络的健康运转，自研并实现了一套基于账户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

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 Gas，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 4

权限系统

实现一个去中心化，区块链内置的合约账户权限系统。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 5

隐私和保密

XuperUnion 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. 多私钥保护的账户体系；
4. 基于椭圆曲线算法的公私钥加密和签名体系；

CHAPTER 6

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 7

总结

XuperUnion 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，拥有完善的账户与权限体系，采用 DPOS 作为共识算法，交易处理速度可达到 9 万 TPS。

CHAPTER 8

介绍

XuperUion 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。XuperUnion 设计上采用了模块化插件化的设计，具有高性能、安全、高可扩展、多语言开发智能合约和灵活等特点。

CHAPTER 9

超级节点

实际的网络中，超级节点的实体可以是一个机构，一家公司或政府部门，对外呈现是一个节点，其内部是分布式网络能力，超级节点参与记账权竞争，保证了全网运行的效率，利用超级计算机和分布式架构，突破单机限制，解决区块链网络算力和存储问题，使节点具备无限的计算力和存储力。

CHAPTER 10

链内并行

当下区块链技术的实现是将所有事物打包后顺序执行。随着智能合约越来越复杂，如果顺序执行智能合约，高并发度将难以实现。顺序执行智能合约不能充分利用多核和分布式的计算能力，造成资源浪费。XuperUnion 将合约中互相依赖的事物挖掘成 DAG 图的形式，便于并发的执行。依据生成的 DAG 图来控制事务的并发，最大化资源的利用，提高合约效率。

CHAPTER 11

XuperModel

XuperUnion 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：

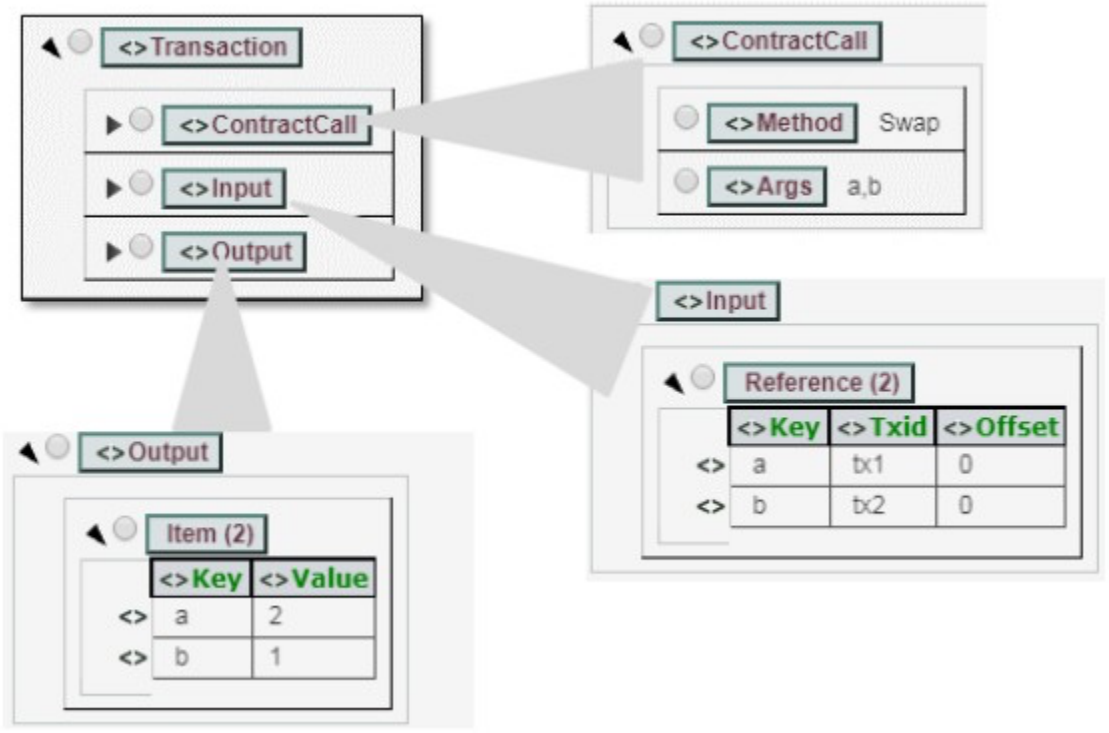

图 1: XuperModel 事务

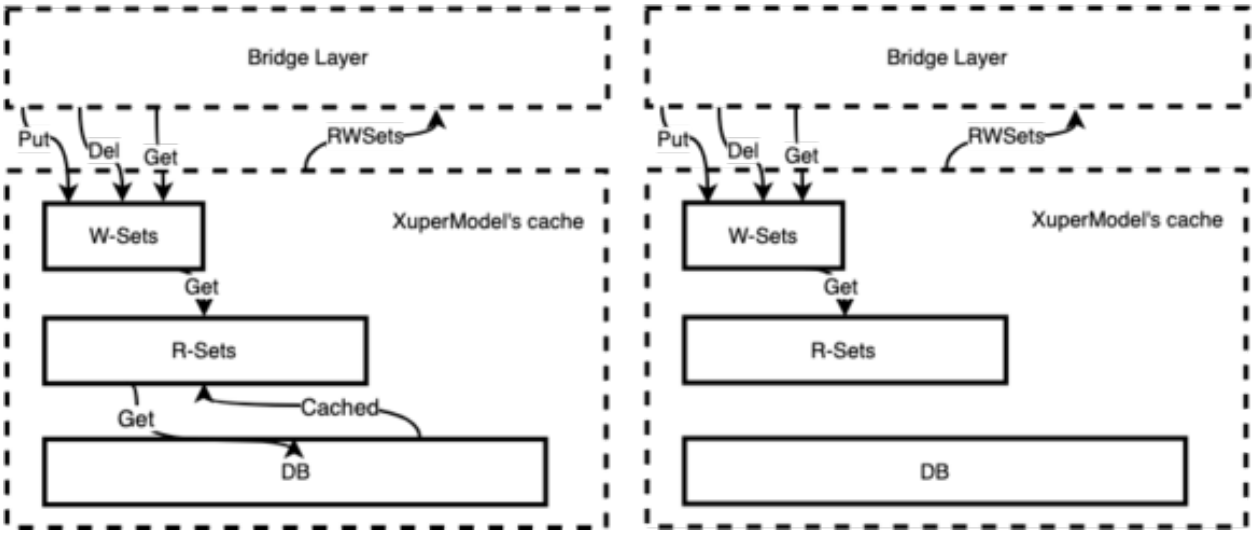

图 2: XuperModel 合约验证

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

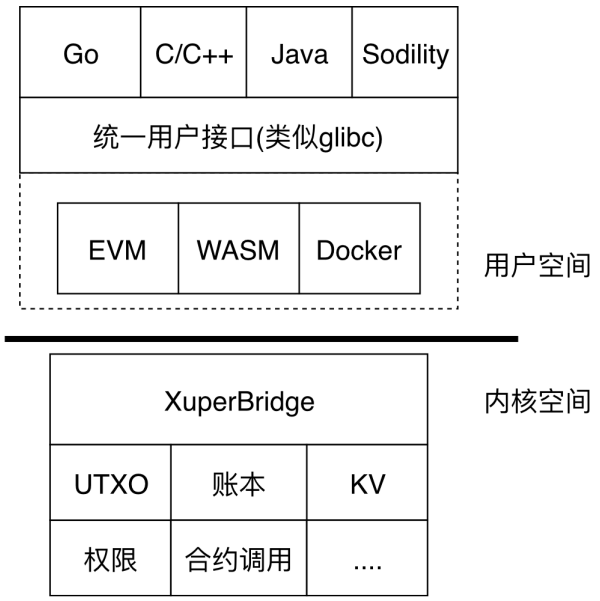

图 1: XuperBridge

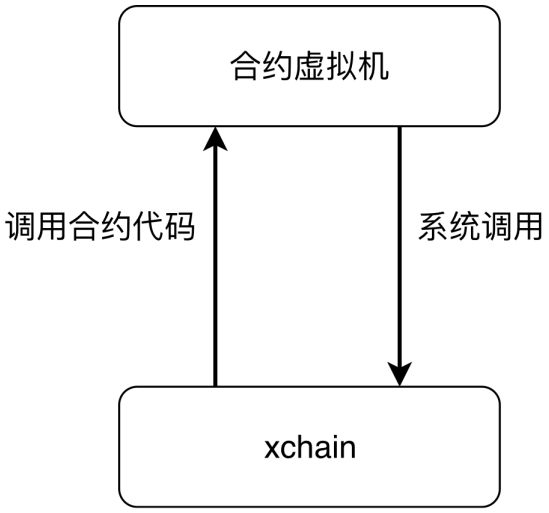

图 2: 合约双向通信

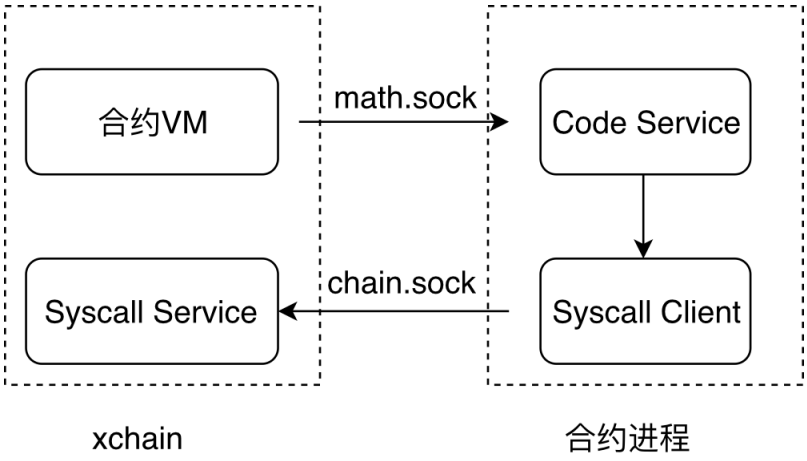

图 3: 合约 socket

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);
}
```

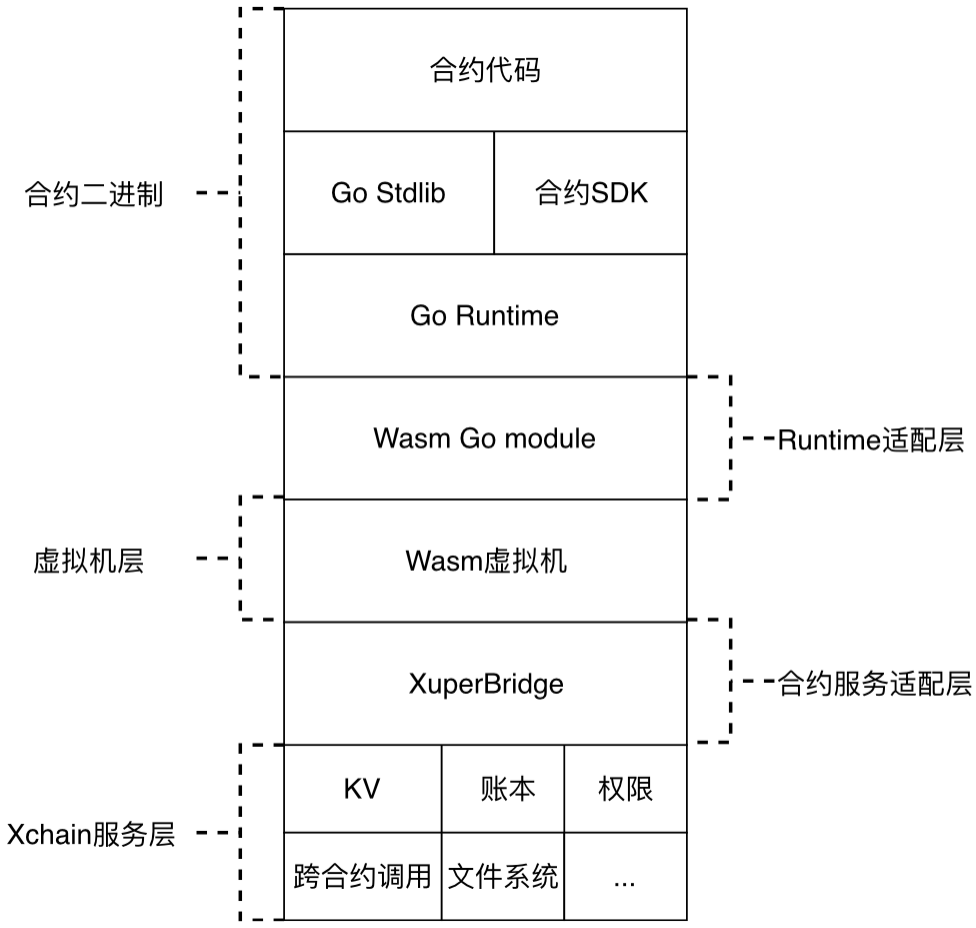
合约代码
Go Stdlib
合约SDK
Go Runtime
Wasm Go module
Wasm虚拟机
XuperBridge
KV
账本
权限
跨合约调用
文件系统
...
合约二进制
虚拟机层
Xchain服务层
Runtime适配层
合约服务适配层

图 4: WASM 合约

12.2 KV 接口与读写集

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

实现：

cache 部分采用 ordered map 实现, value 部分结构如下

```
type Value struct {
    Value []byte
    Flag  uint32
    Ref   *Txinput
}
```

在生成读写集的时候根据 Flag 判断是只读引用还是写入或者是删除，最后遍历一遍整个 map 按照 key 的顺序生成读写集。

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 在合约虚拟机的 Run 函数里面创建，在 xuper3 里面已经是每次执行合约的时候创建 context。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

- MakeContext，创建一个 context，需要合约的参数等信息
- RunContext，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
- CommitContext，提交 context，这一步会把上下文里面对状态的修改提交到持久化层里面，对应 MPT 就是提交根，对应 XuperModule 就是生成读写集

12.4 跨合约调用

受限于目前的合约调用方式，跨合约调用目前只在 native 合约里面实现，等后面 xuper3 统一存储模型就可以真正做到真正的跨虚拟机实现的跨合约调用。

在跨合约调用模型中 2 个状态需要记录：

12.4.1 KV 状态的修改

KV 在目前的合约实现里面是通过 MPT 来实现的，合约执行完毕后生成 root hash，验证节点通过验证 root hash 是否一致来验证 KV 状态是否一致，在跨合约调用中由于牵扯了多个合约的 MPT 状态修改，同时每个合约都是一棵单独的树，因此没办法统一 root hash，因此在这里我们采用默克尔树的思想，把多个合约修改后的 root hash 通过字符串拼接的方式组成一个新的串，在 hash 之后生成多个 MPT root 的联合 hash，验证节点采用相同的算法生成 hash 串，比对是否相同即可。

还有一个问题需要确定：多个 root hash 通过什么顺序来组合?

有两个方案：

- root hash 按字母序排列之后再组合
- 按合约的调用顺序以后序遍历的顺序连接 root hash

12.4.2 转账带来的对 UTXO 的修改

跟之前的合约内 transfer 一样，通过生成新 tx 的方式来支持被调用合约内部调用转账，所有新生成的 tx 的 ContractRef 指向 root tx id，新生成的排列方式按照调用转账的顺序生成。

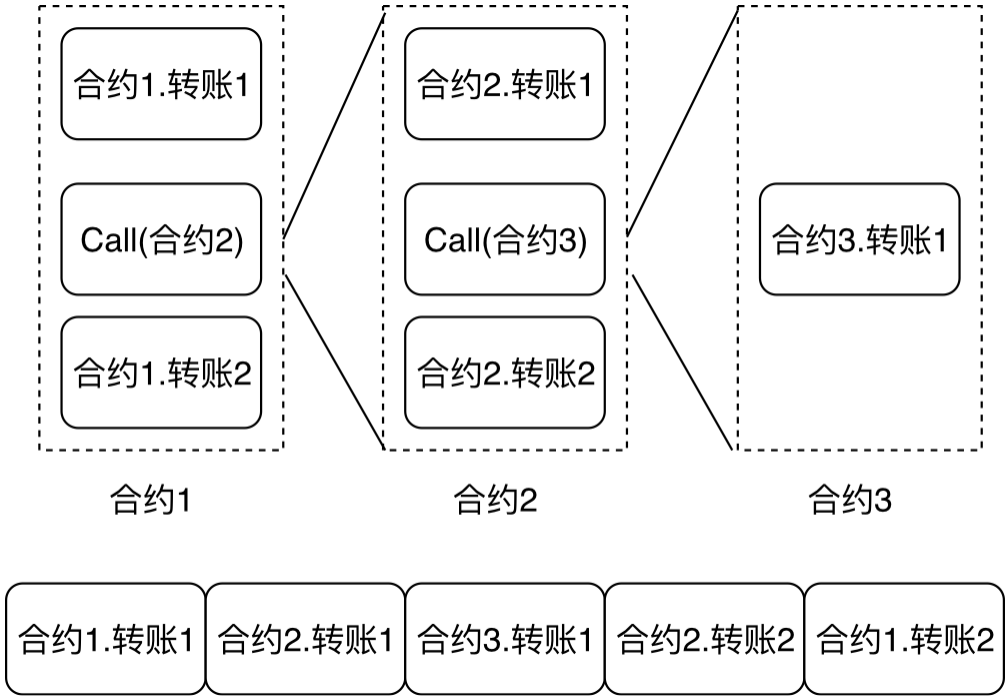

图 5: 合约内转账排列

12.4.3 调用递归层数的限制

合约调用在入口的地方可以传递一个计数器，每次进行合约调用就加一，如果计数器超过指定阈值则直接执行失败。

12.4.4 合约调用方式的更改

在目前版本 (<=2.3) 里面，native 合约的调用方式为在执行合约之前统一执行 SetContext，打包块结束后再执行 Finalize，从这种调用方式上可以看出我们假定合约上下文的生命周期是在一个块里面，我们需要改成状态内敛到合约上下文。

12.4.5 共享 tx 结构体

整个跨合约调用的过程中 tx 结构体是共享的，所有合约执行的时候被认为是在同一个交易里面进行的，但参数会专门设置。

12.4.6 状态的回滚

native 合约在执行失败的时候，不会提交 MPT，在多合约执行的情况下

CHAPTER 13

账户权限控制模型

13.1 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，设计一个基于账户的合约权限系统

13.2 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。账户的名字具有唯一性。
- 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。我们允许一个账户部署多个合约。合约的名字具有唯一性。

13.3 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

- **个人账户 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

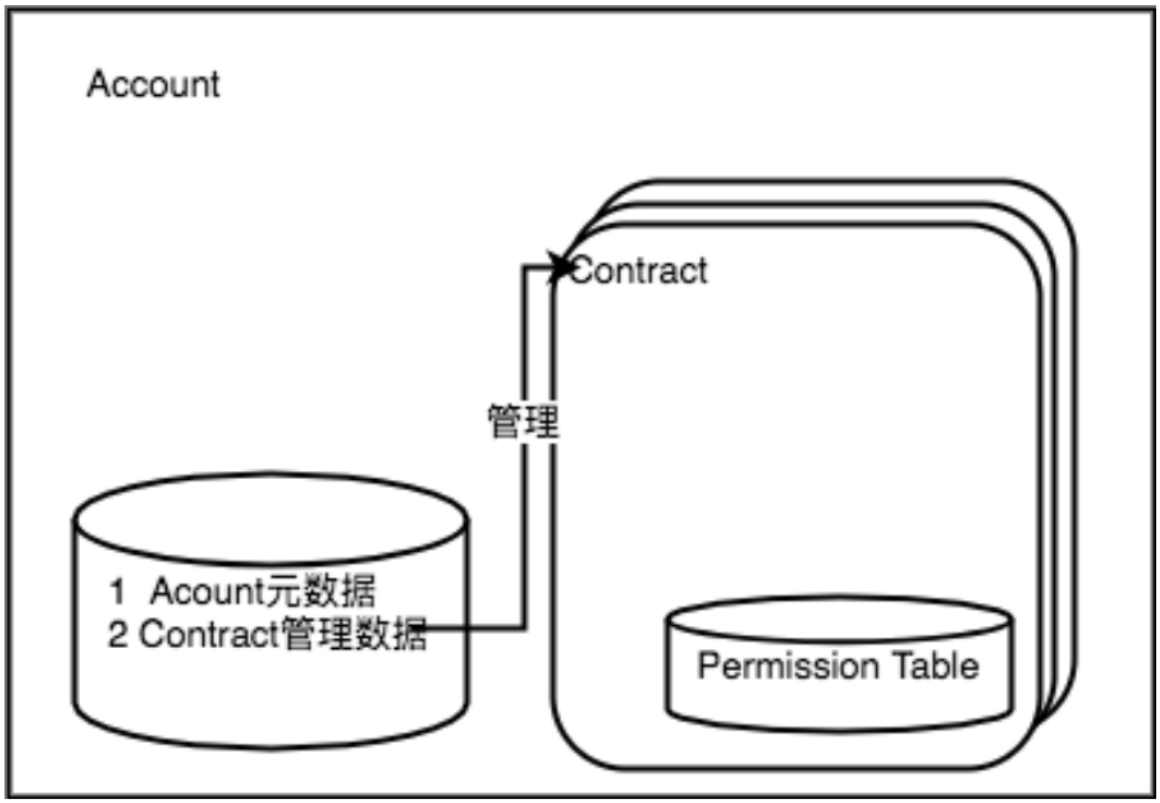
Account
Contract
管理
1 Acount元数据
2 Contract管理数据
Permission Table

图 1: ACL 简介

- **合约账户：超级链智能合约的管理单元。**
  - **账户的创建：**
    * 任何账户或者 AK 都可以调用系统级智能合约创建账户
    * 创建账户需要指定账户对应的拥有者的地址集，如果一个账户中只有一个地址, 那么这个 Addr 对账户完全控制；
    * 创建账户需要指定 ACL 控制策略，用于账户其他管理动作的权限控制；
    * 创建账户需要消耗账户资源；
  - **账户名命名规则；**
    * 合约账户由三部分组成，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。
  - **账户管理：依地址集合据创建时指定的地址集和权限策略，管理账户其他操作**
    * 账户股东剔除和加入
    * 账户资产转账
    * 创建合约，创建智能合约需要消耗账户资源，先将 utxo 资源打到账户下，通过消耗账户的 utxo 资源创建合约，验证的逻辑需要走账户的 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，属于某个账户**
    * 账户所属人员允许在账户内部署合约
    * 账户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_.]

13.4 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

13.5 系统设计

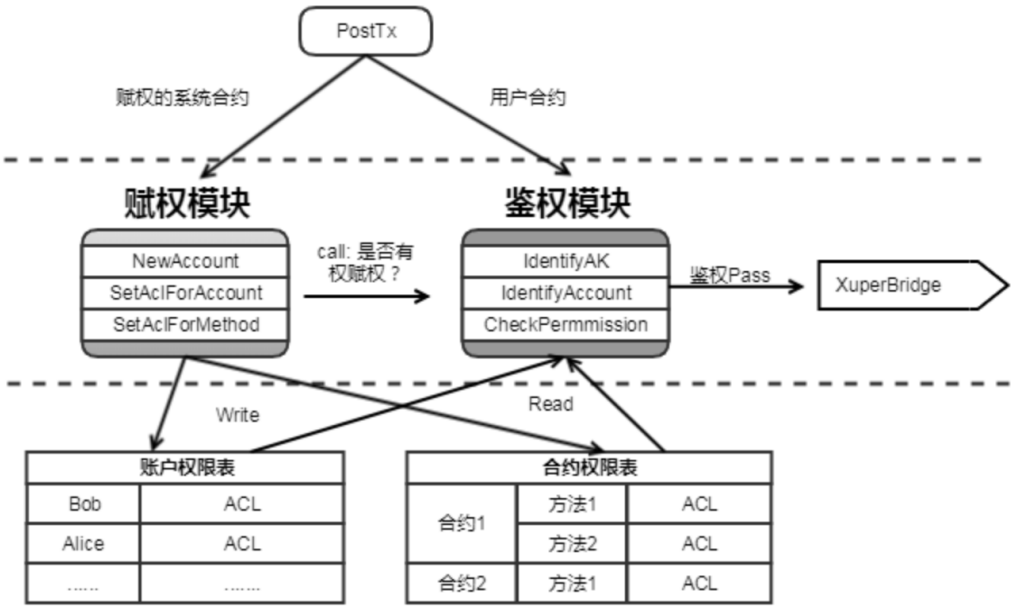

图 2: ACL 架构

13.5.1 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
}

```

(下页继续)

(续上页)

```
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 账户名集
    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 账户名  -> 权重
    AkSets akSets = 3;
}
```

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

13.5.2 系统合约接口

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

13.5.3 样例

acl 模型如下：

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

```

(下页继续)

(续上页)

```
        "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 14

身份认证

14.1 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

14.2 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

14.3 P2P 建立连接过程

14.4 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
- 身份验证支持 XChain address 的验证方式

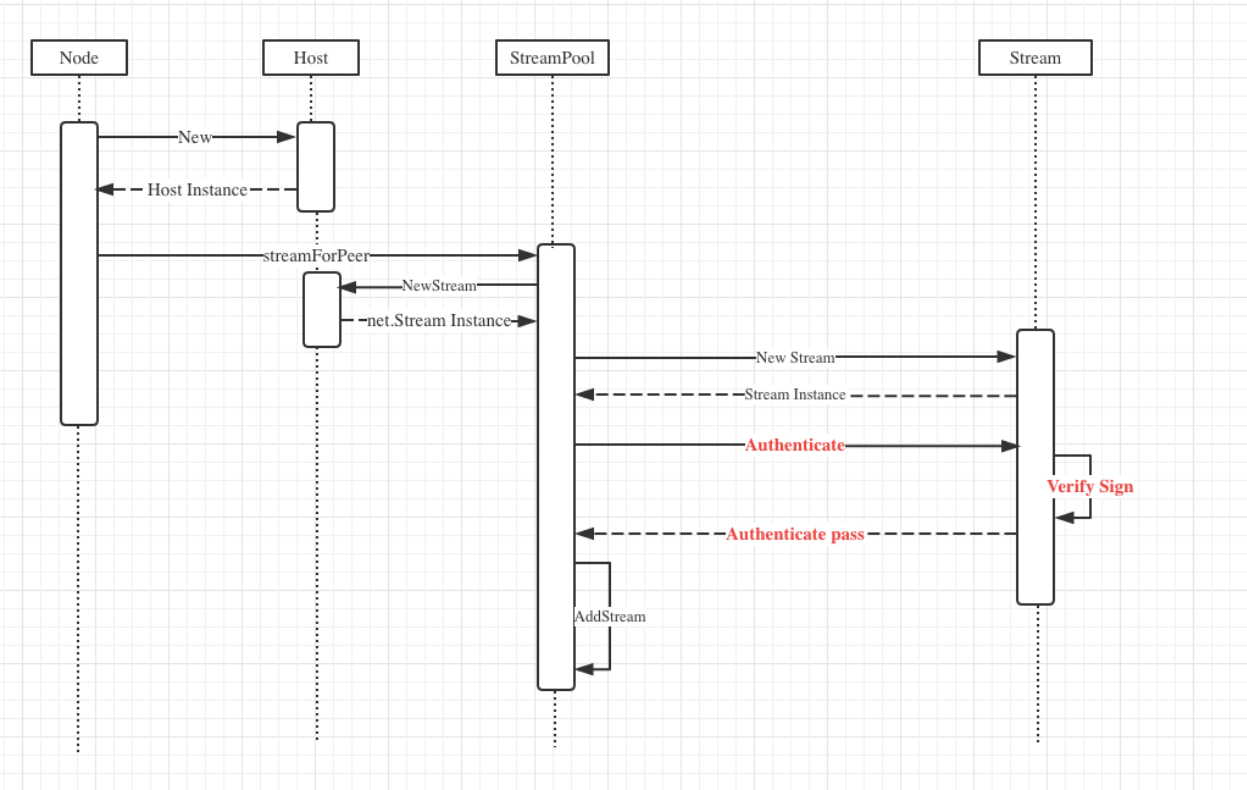

图 1: 连接建立时序

- 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

14.5 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 15

XuperUnion 的编译

15.1 准备环境

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

15.2 编译步骤

- 使用 git 下载源码到本地
  - git clone xuperunion 地址
- 执行命令

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

- 得到产出 xchain 和 xchain-cli

15.3 常见问题

- 配置 go 语言环境变量

```
export GOROOT=.../gotool/go
export PATH=$GOROOT/bin:$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：
  - 比如代码路径 xxx/github.com/xuperchain/xuperunion/src/baidu.com/xchain/xxx
  - export GOPATH=xxx/github.com/xuperchain/xuperunion
- gcc 版本
  - 升级到 4 或 5 以上

CHAPTER 16

单节点网络

16.1 启动单节点模拟的测试网络

16.1.1 获取编译产出

在当前目录下：主要获取目录为 data, logs, conf, plugins, 二进制文件为 xchain，xchain-cli

16.1.2 建立目录

目录名	功能
node/	根节点目录
conf	xchain.yaml:xchain 服务的配置信息,plugins.conf: 插件的配置信息（防止冲突，部署时根据需要修改端口号）
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，唯一性
netkeys	此节点的网络标识 ID，唯一性
config	创始的共识采用 single 模式，指定单一地址有权利出块
logs	程序日志目录
plugins	so 的存放目录
xchain	服务的二进制文件
xchain-cli	客户端工具
wasm2c	wasm 工具

16.1.3 创建链

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

16.1.4 启动服务

```
# 启动服务节点
nohup ./xchain &
```

16.1.5 观察区块

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

16.1.6 测试基本功能

创建新账户

```
# 创建普通用户，包含地址，公钥，私钥
./xchain-cli account newkeys --output data/accounts/bob
# 在 bob 目录下会看到文件 address，publickey，privatekey 生成
# 创建合约账户
# 通用方式
./xchain-cli account new --desc account.des
其中，account.des 内容如下
{
    "module_name": "xkernel",
    "method_name": "NewAccount",
    "args" : {
        "account_name": "1111111111111111",  //说明：16 位数字组成的字符串
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\": {\"AK1\": 0.3,
↪\"AK2\": 0.3}}"
    }
}
# 简易方式
./xchain-cli account new --account 1111111111111111  // 说明：16 位数字组成的字符串
```

查余额

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

转账

```
# --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
```

查询交易信息

```
# 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
```

查询 block 信息

```
# 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
```

发起多重签名交易

```
# generate raw tx
# data/acl/addrs 维护好涉及到操作权限的所有的 address 信息，默认从此地址文件读取，可用参数指定自己文件
## 某个 address 发起
./xchain-cli multisig gen --to YDYBchKWXpG7HSkHy4YoyzTJnd3hTFBgG --amount 100 --desc␣
↪contract.desc -H 127.0.0.1:37101
# 从账户发起
./xchain-cli multisig gen --to YDYBchKWXpG7HSkHy4YoyzTJnd3hTFBgG --amount 100 --desc␣
↪contract.desc -H 127.0.0.1:37101 --from XC11111111111111
# 各方在签名之前可以 check 原始交易是否 ok
./xchain-cli multisig check --input tx.data --ouput visual.out
# 各方签名生成签名文件
./xchain-cli multisig sign --keys data/account/bob --output bob.sign
# 组装成带有签名的完整 tx，并更新账本，同时发送到周边节点
./xchain-cli multisig send --tx tx.out a.sign,b.sign c.sign,d.sign
```

16.1.7 常见问题

tdpos 默认共识，json 文件如下：

```
{
    "version" : "1"
    , "predistribution":[
        {
            "address" : "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN"
            , "quota" : "100000000000000000000"
        }
    ]
    , "maxblocksize" : "128"
    , "award" : "1000000"
    , "decimals" : "8"
```

(下页继续)

(续上页)

```
    , "award_decay": {
        "height_gap": 31536000,
        "ratio": 1
    }, "genesis_consensus":{
    "name": "tdpos",
    "config": {
        "timestamp": "1556444792000000000",
        "proposer_num":"1",
            "period":"3000",
                "alternate_interval":"3000",
                "term_interval":"6000",
            "block_num":"20",
            "vote_unit_price":"1",
            "init_proposer": {
            "1":["dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN"]
            }
        }
    }
}
```

single 共识，json 文件如下

```
{
    "version" : "1"
    , "predistribution":[
        {
            "address" : "dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN"
            , "quota" : "100000000000000000000"
        }
    ]
    , "maxblocksize" : "128"
    , "award" : "428100000000"
    , "decimals" : "8"
    , "award_decay": {
        "height_gap": 31536000,
        "ratio": 1
    },"genesis_consensus":{
        "name": "single",
        "config": {
                        "miner":"dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
```

(下页继续)

(续上页)

```
                        "period": "3000"
                }
        }
}
```

CHAPTER 17

账号与权限管理

17.1 创建合约账号

1. 快速创建一个合约账号

```
./xchain-cli account new —account 1111111111111111 --fee 1000 -H 127.0.0.1:37101
```

2. 按照 json 文件创建一个合约账号

```
./xchain-cli account new —desc account.json --fee 1000 -H 127.0.0.1:37101
```

account.json 模版如下:

```
{
    "module_name": "xkernel",
    "method_name": "NewAccount",
    "args" : {
        "account_name": "1111111111111111",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\": {\"AK1\": 0.5,\"AK2\": 0.5}}"
    }
}
```

17.2 管理合约账号/合约方法 ACL

1. 重新设置一个账号/合约方法的 ACL

   走 multisig 命令

   step1: 准备好 ACL 的 json 文件，该 json 文件中有新的 ACL 配置，文件名记为 acl_new.json

   step2: 针对 acl_new.json 生成原始交易

   先查看合约账号余额是否充足：

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

   如果不足，可以把当前个人账户（data/keys/address）的资产转移一些给合约账户

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

   step3: 使用旧有的 acl 权限模型对新生成的原始交易做签名（假设旧有的权限模型的成员包括 bob 和 alice）

```
./xchain-cli multisig sign --keys data/account/bob --output bob.sign
./xchain-cli multisig sign --keys data/account/alice --output alice.
↪sign
```

   step4: 将签名后的交易 post 出去

```
./xchain-cli multisig send --tx tx.out bob.sign,alice.sign bob.sign,
↪alice.sign
```

2. 查询一个账号的 Acl

```
./xchain-cli acl query —account XC1111111111111111@xuper
```

3. 查询一个合约方法的 Acl

```
./xchain-cli acl query —contract contractName —method methodName
```

17.3 常见问题

1. 创建合约账户，重置合约账户 / 合约方法的 ACL 时，配置文件的案例：

   a. 创建合约账号，配置文件案例如下：

```
{
    "module_name": "xkernel",
    "method_name": "NewAccount",
    "args" : {
        "account_name": "1111111111111111",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\":
↪{\"AK1\": 0.5,\"AK2\": 0.5}}"
    }
}
```

b. 重置合约账户的 ACL, 配置文件案例如下：

```
{
    "module_name": "xkernel",
    "method_name": "SetMethodAcl",
    "args" : {
        "account_name": "1111111111111111",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\":
↪{\"AK1\": 0.5,\"AK2\": 0.5}}"
    }
}
```

c. 重置合约方法的 ACL, 配置文件案例如下：

```
{
    "module_name": "xkernel",
    "method_name": "SetAccountAcl",
    "args" : {
        "contract_name": "math",
        "method_name": "transfer",
        "acl": "{\"pm\": {\"rule\": 1,\"acceptValue\": 0.6},\"aksWeight\":
↪{\"AK1\": 0.5,\"AK2\": 0.5}}"
    }
}
```

CHAPTER 18

智能合约开发

18.1 编写合约

参考源码样例.../github.com/xuperchain/xuperunion/contractsdk/go/example/counter/counter.go 主要实现一个 struct 的三个方法，initialize，invoke 和 query，来实现自己的逻辑

18.2 wasm 合约

18.2.1 部署合约

编译

注意合约编译环境与源码编译环境一致

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

将编译好的合约二进制 counter 放到目录 node/data/blockchain/${chain name}/native/下其中 ${chain name}=xuper，因为样例是 xuper 链

部署

```
# 便捷方式
## 账户下权限 AK 是自己，提前创建好账号，并保证账号下有钱
./xchain-cli wasm deploy --account XC1111111111110600@xuper --cname counter -H␣
↪localhost:37101 data/blockchain/xuper/native/counter
# 多重签名场景
## 提前维护好 data/acl/addrs 需要的合作的地址
./xchain-cli wasm deploy --account XC1111111111110600@xuper --cname counter -H␣
↪localhost:37801 -m data/blockchain/xuper/native/counter
## 后续参看多重签名交易的后续 check, sign, send 场景
```

调用合约

```
./xchain-cli wasm query -a '{"key":"counter"}' -H localhost:37101 counter
# 便捷方式
./xchain-cli wasm invoke -a '{"key":"counter"}' -H localhost:37101 counter
# 多重签名场景
# 参考部署多重签名场景
./xchain-cli wasm invoke -a '{"key":"counter"}' -H localhost:37101 counter -m
```

18.2.2 native 合约

部署合约

编译文件

编译合约，注意合约编译环境与源码编译环境一致

```
cd counter
go build
# 产出二进制 counter
```

将编译好的合约二进制 counter 放到目录 node/data/blockchain/${chain name}/native/下其中 ${chain name}=xuper，因为样例是 xuper 链

激活合约

- 发起提案

  合约可以被使用需要发起提案，并投票，通过投票后方可激活合约

```
./xchain-cli native activate --vote-height-offset 50 counter --version 1.0.
↪0 -H 127.0.0.1:37101
# --vote-height-offset 表明距离当前高度多高后开始计票判断合约是否可以生效
# 执行完后得到 proposal id
```

- 投票

```
# data/keys 下的账户对此提案投票 token 数量为 amount
./xchain-cli vote --amount 100000499592699999999 --frozen 5550␣
↪abd9bf4472a833b096a5dc58847cc249b9765a49511d4a69e364e6651607bf94
↪#proposal id
# 提案可以生效需要提案的票数占据总币量的 51%，当然这是默认配置比例
```

- 确认是否激活

```
# 查看区块高度，是否达到提案生效高度
./xchain systemstatus -H 127.0.0.1:37101
# 查看合约 math 的状态，status 为 1，代表激活成功
./xchain-cli native status -H 127.0.0.1:37101
# status 为 1 表示激活成功
```

调用合约

- json 文件示例

```
{
    "module_name":"native",      # 还可以是 wasm
    "contract_name":"counter",   # 自己编写的合约名字
    "method_name": "initialize", # 还可写 invoke 和 query
    "args" : {
        "key":"mycounter"        # 调用的参数是 kv 形式
    }
}
```

- 调用合约

```
# 参看发起多重签名交易
# data/acl/addrs 维护好调用合约所需的权限集合 addrs
./xchain-cli multisig gen --desc desc.json --amount=1 --to $address -H 127.0.0.1:37101
# 查看合约预执行结果，通过文件 visualtx.out
./xchain-cli multisig check
```

(下页继续)

(续上页)

```
# 继续进行后续操作...
# 查询合约还可以通过此命令
./xchain-cli native query counter --args '{"key":"mycounter"}'
```

访问权限管理

合约方法的 ACL 控制参看 设置合约方法权限样例

CHAPTER 19

多节点网络搭建

在阅读本节前，请先阅读“单节点网络”。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。

19.1 p2p 网络配置

1. 假设搭建 3 个节点的网络，在单节点的基础上再新增两个节点 node2 和 node3；
2. 创建 node2 和 node3 的部署路径，并按照 5.2.1.2 建立每个节点自己的部署目录；
3. 查询 node 节点的 netUrl，会得到节点 netUrl

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

4. 在 node2 和 node3 的 p2pv2 配置中的 bootNodes 增加 node 的 netUrl，如下所示：

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

5. 设置 node2 和 node3 节点 RPC 服务暴露的端口

```
// port 是节点启动时 RPC 服务监听的端口，如果在一台机器上部署注意端口配置不要冲突，node 配置的是 :37101，node2 和 node3 可以分别设置为 :37102 和 :37103
tcpServer:
port: :37102
```

6. 启动 node2 和 node3，check 服务是否正常：

```
./xchain systemstatus -H 127.0.0.1:37102
./xchain systemstatus -H 127.0.0.1:37103
```

若节点高度一致，则网络启动成功。

19.2 搭建 TDPoS 共识网络

TODO 是搭建一个 SINGLE 共识的多节点网络。网络中的矿工节点是 node。

XuperUnion 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 TDPoS 链。

下面创世块配置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":{
        "name": "tdpos",
        "config": {
                "timestamp": "1548123921000000000", # 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
```

(下页继续)

(续上页)

```
                "proposer_num":"3",# 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
                "period":"3000",# 每个矿工连续出块的出块间隔
                "alternate_interval":"6000",# 每一轮内切换矿工时的时间间隔，需要为 period 的整数倍
                "term_interval":"9000", # 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
                "block_num":"200",# 每一轮内每个矿工轮值任期内连续出块的个数
                "vote_unit_price":"1",# 为被提名的候选人投票时，每一票单价，即一票等于多少 Xuper
        # 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"]
                        }
                }
        }
}
```

修改完创世块参数后，删除./data/blockchain 下的内容，3 个节点全部重新创建链：

```
./xchain createChain
```

先启动 node，再启动 node2 和 node3，至此 TDPoS 共识的集群启动成功。

19.3 提名候选人

```
./xchain-cli transfer --to=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --name=$chain --desc=./nominate.json --amount=11000002266 --frozen=-1 -H=$host
```

nominate.json

```
{
    "module": "tdpos",
    "method": "nominate_candidate",
    "args": {
        "candidate": " 提名 address"
    }
}
```

19.4 投票

```
./xchain-cli transfer --to=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN --desc=./vote.json --amount=
↪$ballots --frozen=-1 --name=$chain -H=$host
```

vote.json

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

19.5 撤销提名 && 撤销投票

```
./bin/xchain transfer --to Y4TmpfV4pvhYT5W17J7TqHSLo6cqq23x3 --desc=./revoke_demo.json --
↪amount=1
```

revoke_demo.json (txid 为提名或者投票时的 txid, 发起的交易的 input 需只有一个，且 address 与提名或者投票时需要相同)

```
{
    "module":"proposal",
    "method": "Thaw",
    "args" : {
        "txid":"02cd75a721f2589a3ff6768b49650b46fa0b042f970df935b4d28a15aa19e49a"
    }
}
```

19.6 TDPOS 结果查询

```
./xchain-cli tdpos -h
```

提示如下所示：

1. 查询候选人信息

图 1: 查询命令

```
./xchain-cli tdpos query-candidates --name=$chain -H=$host
```

2. 查看某一轮的出块顺序

```
./xchain-cli tdpos query-checkResult -t=30 --name=$chain -H=$host
```

3. 查询提名信息：某地址发起提名的记录

```
./xchain-cli tdpos query-nominate-records --name=$chain -H=$host -
↪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 --name=$chain -H=$host -
↪a=TyYCWDJ1pyV8fA3VyPenCdFdcPmHnwMhx
```

5. 某选民的有效投票记录

```
./xchain-cli tdpos query-vote-records --name=$chain -H=$host -
↪a=dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN
```

6. 某候选人被投票记录

```
./xchain-cli tdpos query-voted-records --name=$chain -H=$host -
↪a=TyYCWDJ1pyV8fA3VyPenCdFdcPmHnwMhx
```

19.7 常见问题

1. 端口冲突：注意如果在一台机器上部署 3 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；
2. 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。
3. 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，否则会加入网络失败而启动失败。
4. The gas you cousume is: XXXX, You need add fee. 通过加–fee XXXX 参数附加资源

CHAPTER 20

电子存证合约

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

20.1 电子存证合约简介

电子存证应用主要是通过区块链解决的存证中的信任问题，而存证合约只需做简单的读写操作即可

20.2 电子存证合约具备的读写操作

- 通过 invoke 方法，put 存证到区块链
- 通过 query 方法，get 存证

20.3 调用 json 文件示例

Invoke

./xchain-cli native invoke -a ' 下面 json 中 args 字段的内容' –method save -H localhost:37101 eleccert

```
{
    "module_name": "native",       // native or wasm
    "contract_name": "eleccert",   // contract name
```

(下页继续)

(续上页)

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

Query

./xchain-cli native 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
{
    "filehash": " 文件 hash 值",
    "timestamp": " 文件存入 timestamp"
}
```

CHAPTER 21

数字资产交易

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

21.1 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。

21.2 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

21.3 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
```

(下页继续)

(续上页)

```
        "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
```

(下页继续)

(续上页)

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

CHAPTER 22

操作指导

22.1 如何升级软件

当版本升级时，需要 pull 最新的代码，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，注意需要先启动 bootNodes 节点。

22.2 配置文件说明

XuperUnion 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:
```

(下页继续)

(续上页)

```
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
```

22.3 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	XuperUnion 命令行功能集合 xchain XuperUnion 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 生成用户账户 client 密码学模块的客户端接口 config 定义创建账户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 23

指令介绍 (API)

23.1 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/合约账户对应的余额是否足够
rpc QueryContract(QueryContractRequest) returns (QueryContractResponse)	查询合约数据

23.2 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 24

常见问题解答

24.1 XuperUnion 是完全匿名的嘛

XuperUnion 包括普通的 AK 以及合约账户。创建普通 AK 是免费的，而创建合约账号需要消耗账户资源，对于 AK 操作做是完全匿名的，而合约账号通常和若干个合约绑定，匿名性较 AK 差。

24.2 XuperUnion 出块时间是多少

XuperUnion 支持可插拔共识算法，默认共识算法为 TDPOS。对于 TDPOS 共识算法，出块时间由 period,alternate_interval,term_interval 决定。term_interval 表示更换轮的时间间隔，alternate_interval 表示每一轮内不同的候选人更换时间间隔，period 表示同一个候选人内部出块间隔。对于 Pow 算法，通过配置出块难度间接确定出块时间。对于 Single 算法，出块时间取决于配置文件中的配置参数。

24.3 XuperUnion 安全嘛

XuperUnion 通过椭圆曲线算法生成随机数并生成私钥和相应的公钥，对公钥进行 SHA256 散列后，使用 Ripemd160 生成散列摘要，然后使用 base58 生成地址，通过公钥无法推导出私钥，通过地址无法推导出公钥。匿名性比较好。同时，XuperUnion 自研并实现了一套基于合约账号的权限系统，通过多私钥保护，丰富的权限模型保障合约调用的安全性。

CHAPTER 25

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，
账户 (用户账户、合约账户)	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
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。地址可以是合约账户，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，通常为用户账户或合约账号。
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 26

Indices and tables

- genindex
- modindex
- search