
xuperunion-doc Documentation

xuper

2019 年 09 月 04 日

1	简介	1
2	模块	3
3	智能合约	5
4	权限系统	7
5	隐私和保密	9
6	性能	11
7	总结	13
8	XuperUnion 环境部署	15
8.1	准备环境	15
8.2	编译 XuperUnion	15
9	XuperUnion 基本操作	17
9.1	部署 xchain 服务	18
9.2	基本功能的使用	18
10	XuperModel	21
11	XuperBridge	23
11.1	内核调用设计	23
11.2	KV 接口与读写集	27
11.3	合约上下文	27
11.4	跨合约调用	28
12	账户权限控制模型	31

12.1	背景	31
12.2	名词解释	31
12.3	模型简介	31
12.4	实现功能	33
12.5	系统设计	34
13	身份认证	37
13.1	背景	37
13.2	名词解释	37
13.3	P2P 建立连接过程	37
13.4	实现过程	37
13.5	主要结构修改点	38
14	合约账号	39
14.1	访问控制列表 (ACL)	39
14.2	合约账号创建	39
14.3	合约账号基本操作	40
15	多节点部署	43
15.1	p2p 网络配置	43
15.2	搭建 TDPoS 共识网络	44
16	创建合约	47
16.1	编写合约	47
16.2	部署 wasm 合约	48
16.3	部署 native 合约	49
17	发起提案	51
18	配置变更	53
18.1	配置多盘存储	53
18.2	替换扩展插件	53
19	电子存证合约	55
19.1	电子存证合约简介	55
19.2	电子存证合约具备的读写操作	55
19.3	调用 json 文件示例	55
20	数字资产交易	57
20.1	ERC721 简介	57
20.2	ERC721 具备哪些功能	57
20.3	调用 json 文件示例	58
21	智能合约 SDK 使用说明	61
21.1	C++ 接口 API	61

21.2 Go 接口 API	68
22 操作指导	71
22.1 如何获取 XuperUnion	71
22.2 如何升级软件	71
22.3 配置文件说明	71
22.4 各文件说明	74
23 指令介绍 (API)	75
23.1 节点 rpc 接口	75
23.2 开发者接口	76
24 常见问题解答	79
24.1 XuperUnion 是完全匿名的嘛	79
24.2 XuperUnion 出块时间是多少	79
24.3 XuperUnion 安全嘛	79
25 词汇表	81
26 Indices and tables	83

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。

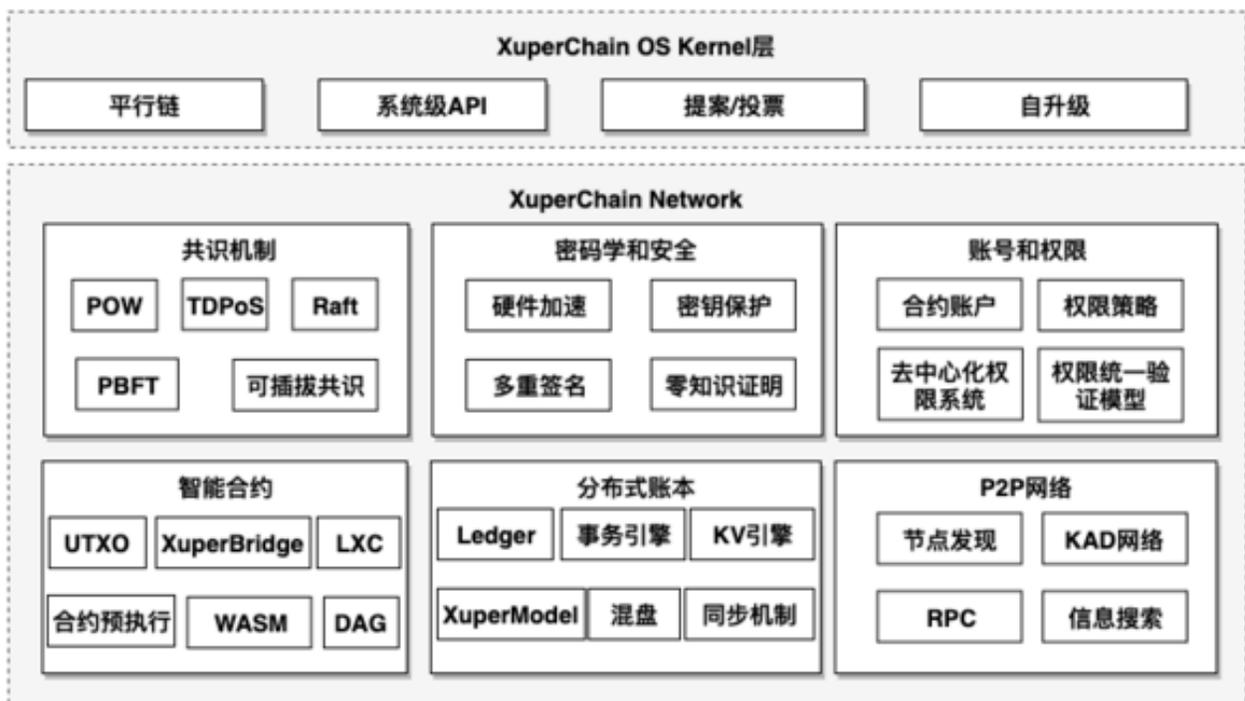


图 1: XuperChain 架构

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 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

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；

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；

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；

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；

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。

本章节将指导您获取 XuperUnion 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

8.1 准备环境

XuperUnion 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`

8.2 编译 XuperUnion

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

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

- 在 output 目录得到产出 `xchain` 和 `xchain-cli`

注解: 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:
- 比如代码路径 xxx/github.com/xuperchain/xuperunion/src/baidu.com/xchain/xxx
- export GOPATH=xxx/github.com/xuperchain/xuperunion

GCC 版本需要升级到 4 或 5 以上

XuperUnion 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）

9.1 部署 xchain 服务

9.1.1 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

```
1 # 创建 xuper 链
2 ./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）

9.1.2 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

```
1 # 启动服务节点
2 nohup ./xchain &
```

9.1.3 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

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

9.2 基本功能的使用

9.2.1 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

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

9.2.2 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

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

9.2.3 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

```
1 # --keys 从此地址 转给 --to 地址 --amount 钱
2 ./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)

9.2.4 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

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

9.2.5 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

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


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：

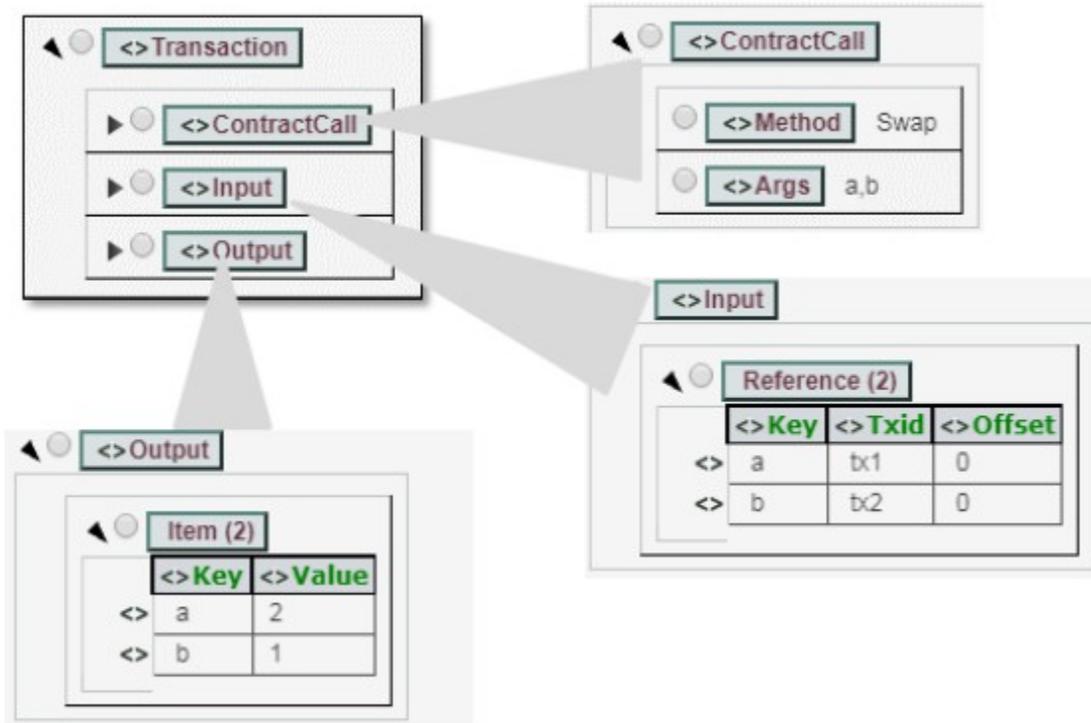


图 1: XuperModel 事务

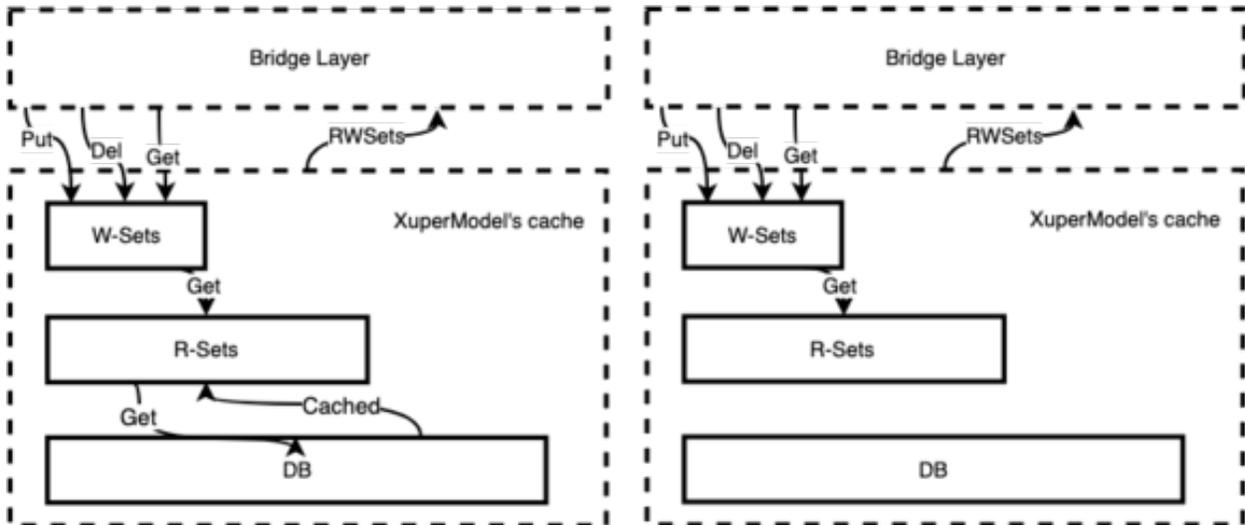


图 2: XuperModel 合约验证

11.1 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。

11.1.1 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

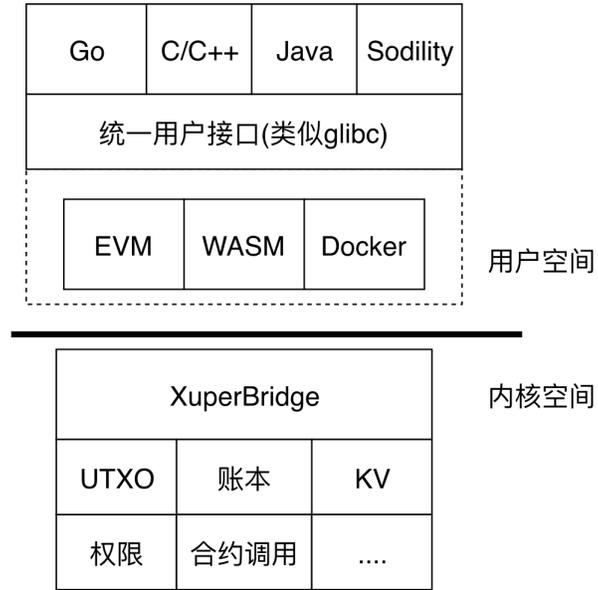


图 1: XuperBridge

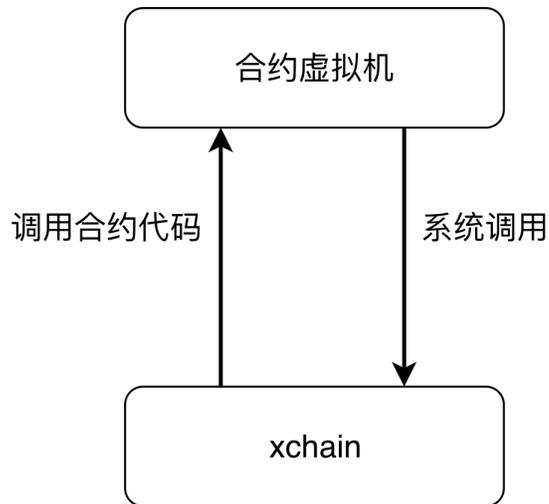


图 2: 合约双向通信

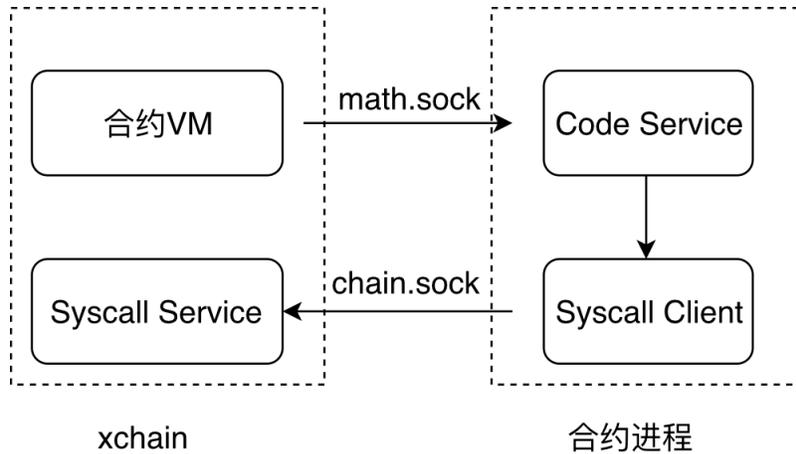


图 3: 合约 socket

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。

11.1.2 PB 接口

合约暴露的代码接口

```

1 service NativeCode {
2     rpc Call(CallRequest) returns (CallResponse);
3 }
  
```

xchain 暴露的 syscall 接口

```

1 service Syscall {
2     // KV service
3     rpc PutObject(PutRequest) returns (PutResponse);
4     rpc GetObject(GetRequest) returns (GetResponse);
5     rpc DeleteObject(DeleteRequest) returns (DeleteResponse);
6     rpc NewIterator(IteratorRequest) returns (IteratorResponse);
7
8     // Chain service
9     rpc QueryTx(QueryTxRequest) returns (QueryTxResponse);
10    rpc QueryBlock(QueryBlockRequest) returns (QueryBlockResponse);
11    rpc Transfer(TransferRequest) returns (TransferResponse);
12 }
  
```

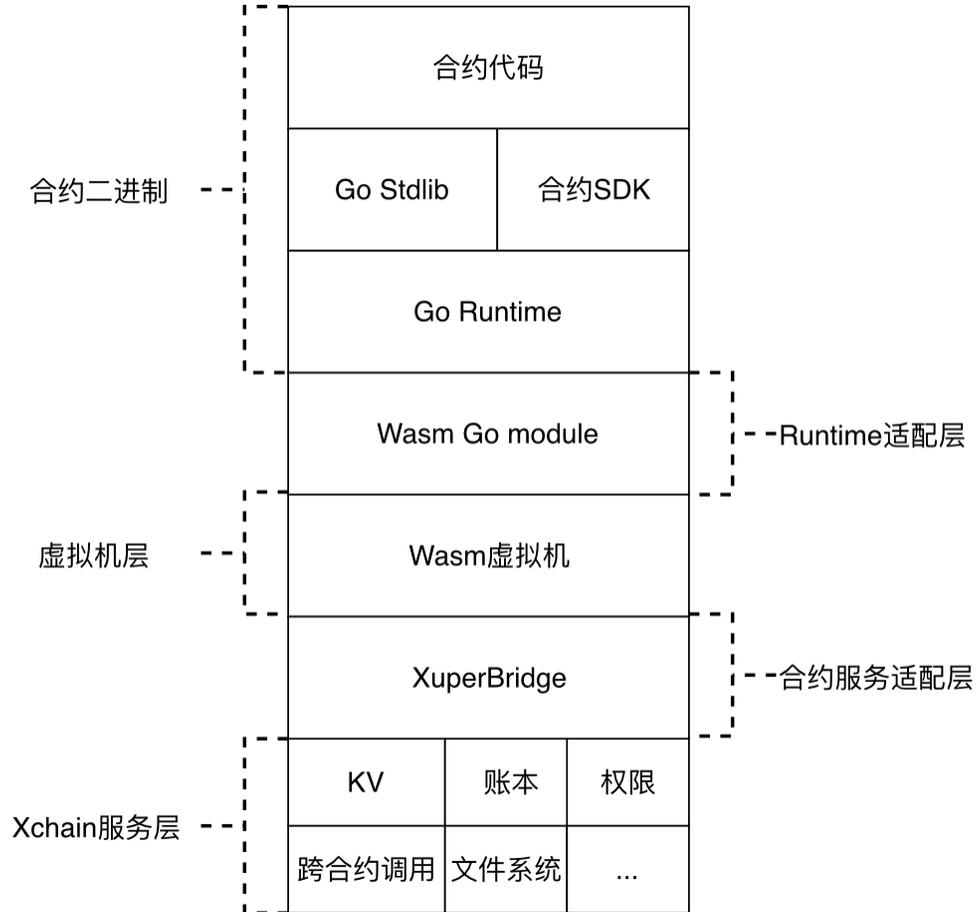


图 4: WASM 合约

11.2 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

```

1 type Value struct {
2     Value []byte
3     Flag  uint32
4     Ref   *Txinput
5 }

```

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。

11.3 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。

11.3.1 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。

11.3.2 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

11.4 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：

11.4.1 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

11.4.2 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。

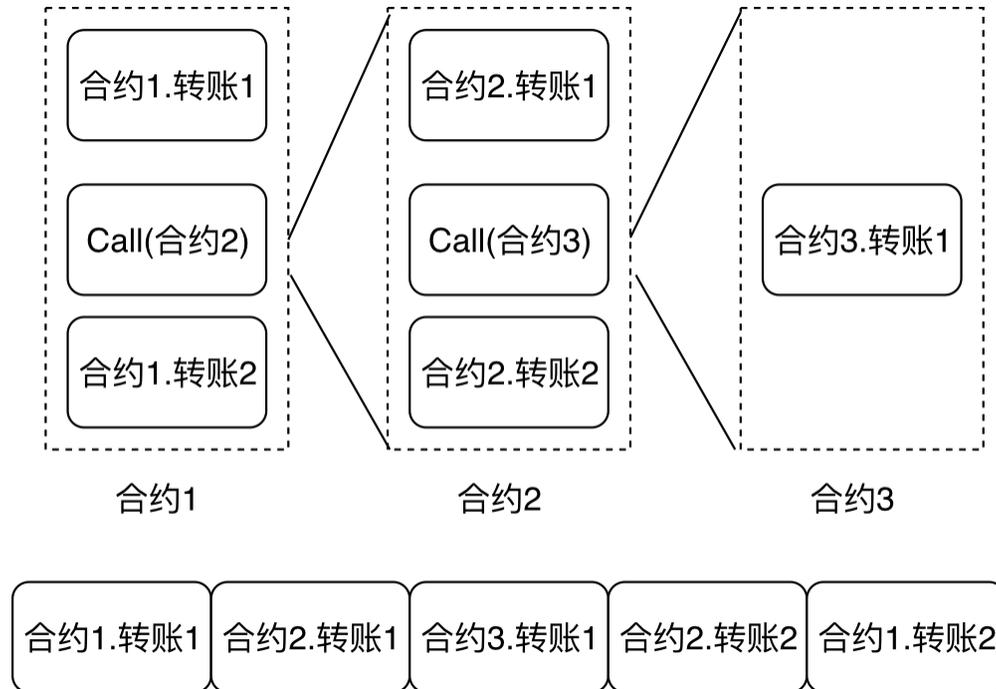


图 5: 合约内转账排列

11.4.3 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。

11.4.4 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。

11.4.5 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。

11.4.6 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

账户权限控制模型

12.1 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

12.2 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。

12.3 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

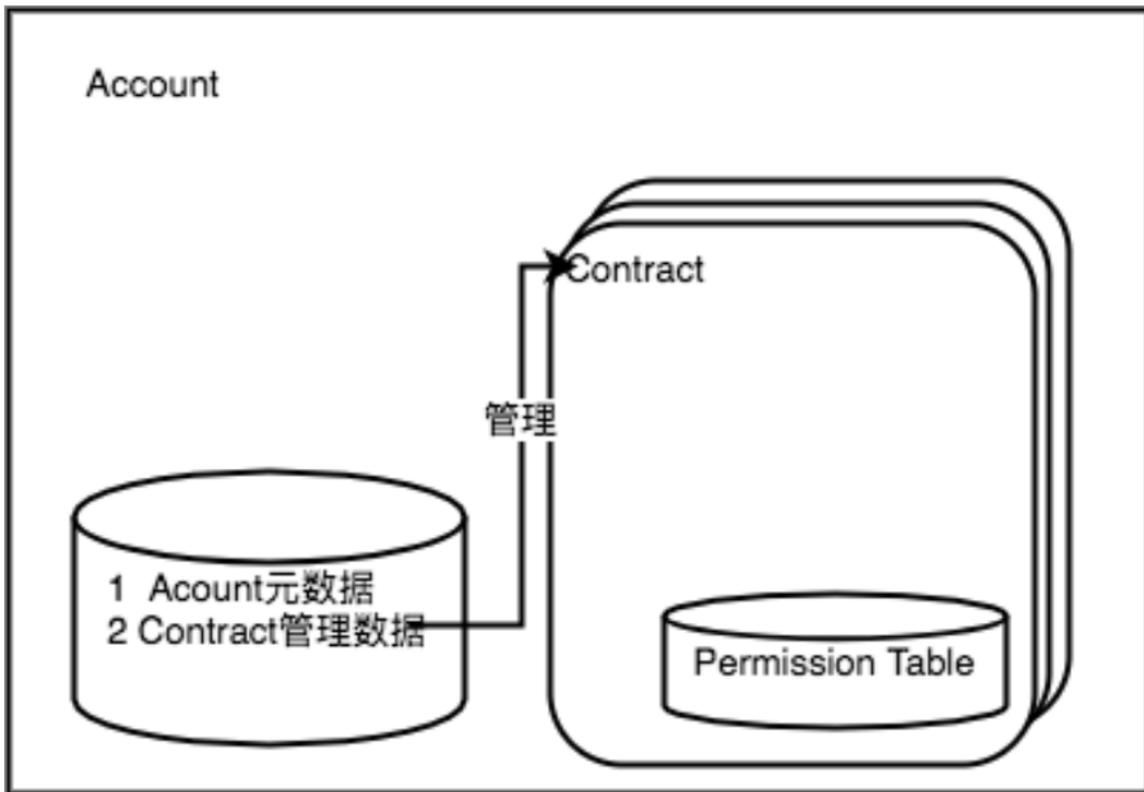


图 1: ACL 简介

- **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_]。**

12.4 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

12.5 系统设计

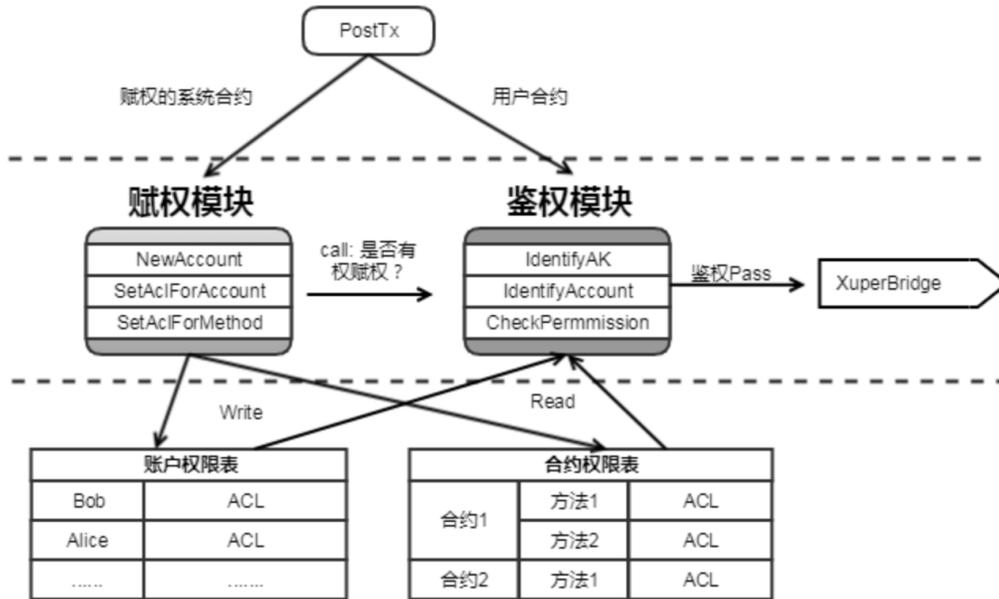


图 2: ACL 架构

12.5.1 ACL 数据结构说明

```

1 // ----- Account and Permission Section -----
2 enum PermissionRule {
3     NULL = 0;           // 无权限控制
4     SIGN_THRESHOLD = 1; // 签名阈值策略
5     SIGN_AKSET = 2;     // AKSet 签名策略
6     SIGN_RATE = 3;     // 签名率策略
7     SIGN_SUM = 4;      // 签名个数策略
8     CA_SERVER = 5;     // CA 服务器鉴权
9     COMMUNITY_VOTE = 6; // 社区治理
10 }
11

```

(下页继续)

(续上页)

```

12 message PermissionModel {
13     PermissionRule rule = 1;
14     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
15 }
16
17 // AK 集 的表示方法
18 message AkSet {
19     repeated string aks = 1; //一堆公钥
20 }
21
22 message AkSets {
23     map<string, AkSet> sets = 1;    // 公钥 or 账户名集
24     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
25 }
26
27 // Acl 实际使用的结构
28 message Acl {
29     PermissionModel pm = 1;        // 采用的权限模型
30     map<string, double> aksWeight = 2; // 公钥 or 账户名 -> 权重
31     AkSets akSets = 3;
32 }

```

签名阈值策略: $\text{Sum}\{\text{Weight}(\text{AK}_i), \text{if sign_ok}(\text{AK}_i)\} \geq \text{acceptValue}$

12.5.2 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

12.5.3 样例

acl 模型如下:

```

1 {
2     "pm": {
3         "rule": 1,
4     }

```

(下页继续)

```
5     "acceptValue": 1.0
6   },
7   "aksWeight": {
8     "AK1": 1.0,
9     "AK2": 1.0
10  }
11 }
```

- 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

13.1 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

13.2 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

13.3 P2P 建立连接过程

13.4 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

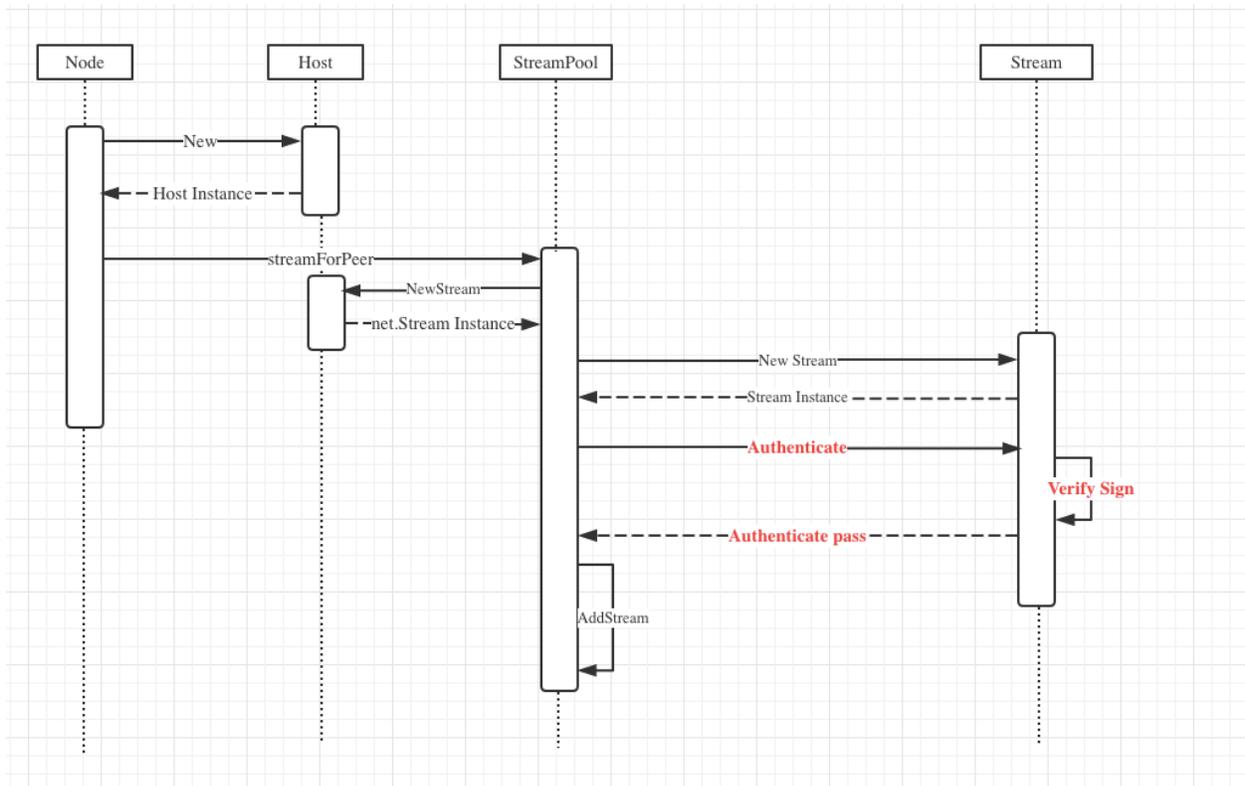


图 1: 连接建立时序

- 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

13.5 主要结构修改点

```

1 // stream 增加 authenticate 接口
2 func (s *Stream) Authenticate() error {}
3
4 // 收到身份验证消息后的回调处理函数接口
5
6 func (p *P2PServerV2) handleGetAuthentication(ctx context.Context, msg *xuper_p2p.
  ↪XuperMessage) (*xuper_p2p.XuperMessage, error) {}

```

14.1 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：

```
1 {
2   "pm": {
3     "rule": 1,           # rule=1 表示签名阈值策略, rule=2 表示 AKSet 签名策略
4     "acceptValue": 0.6 # acceptValue 为签名需达到的阈值
5   },
6   "aksWeight": {       # aksWeight 里规定了每个地址对应账号签名的权重
7     "AK1": 0.3,
8     "AK2": 0.3
9   }
10 }
```

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

14.2 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：

```
1 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：

```
1 {
2   "module_name": "xkernel",
3   "method_name": "NewAccount",
4   "args" : {
5     "account_name": "1111111111111111", # 说明：账号名称是 16 位数字组成的字符串
6     # acl 中的内容注意转义
7     "acl": "{\"pm\": {\"rule\": 1, \"acceptValue\": 0.6}, \"aksWeight\": {\"AK1\": 0.3,
8     ↪ \"AK2\": 0.3}}}"
9   }
}
```

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：

```
1 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`）

注解： 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）

14.3 合约账号基本操作

14.3.1 查询账号 ACL

Xchain 的客户端工具提供了 ACL 查询功能，只需如下命令

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

14.3.2 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

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。

15.1 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：

```
1 ./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

```
1 ./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

```
1 p2pv2:  
2 // 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，
```

(下页继续)

(续上页)

```

17     "name": "tdpos",
18     "config": {
19         # 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
20         "timestamp": "1548123921000000000",
21         # 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
22         "proposer_num": "3",
23         # 每个矿工连续出块的出块间隔
24         "period": "3000",
25         # 每一轮内切换矿工时的时间间隔, 需要为 period 的整数倍
26         "alternate_interval": "6000",
27         # 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
28         "term_interval": "9000",
29         # 每一轮内每个矿工轮值任期内连续出块的个数
30         "block_num": "200",
31         # 为被提名的候选人投票时, 每一票单价, 即一票等于多少 Xuper
32         "vote_unit_price": "1",
33         # 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
34         "init_proposer": {
35             "1": ["RU7Qv3CrecW5waKc1ZWYnEuTdJNjHc43u",
↪ "XpQXiBNoleHRQpD9UzbBisTPXojpyzkxn", "SDCBba3GVYU7s2VYQVrhMGLet6bobNzbM"]
36         }
37     }
38 }
39 }

```

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。

16.1 编写合约

源码可以参考 `xuperchain/xuperunion/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

```
1 func (m *math) Initialize(nci code.Context) code.Response { ... }
2 func (m *math) Invoke(nci code.Context) code.Response { ... }
3 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/xuperunion/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

```
1 type Context interface {
2     Args() map[string][]byte
3     Caller() string
4     Initiator() string
5     AuthRequire() []string
6
7     PutObject(key []byte, value []byte) error
8     GetObject(key []byte) ([]byte, error)
9     DeleteObject(key []byte) error
```

(下页继续)

发起提案

XuperUnion 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）

```
1 {
2   "module": "proposal",
3   "method": "Propose",
4   "args" : {
5     "min_vote_percent": 51,           # 生效的资源比例
6     "stop_vote_height": 800         # 计票截至的高度
7   },
8   "trigger": {
9     "height": 1000,                 # 期望生效的高度
10    "module": "consensus",
11    "method": "update_consensus",
12    "args" : {
13      "name": "tdpos",
14      "config": {
15        "version": "2",
16        "proposer_num": "2",         # 代表个数
17        "period": "3000",
```

(下页继续)

(续上页)

```
18         "alternate_interval": "6000",
19         "term_interval": "9000",
20         "block_num": "20",
21         "vote_unit_price": "1",
22         "init_proposer": {           # 出块的代表名单
23             "1": ["dpzuVdosQrF2kmzumhVeFQZa1aYcdgFpN",
24                 ↪ "U5SHuTiiSP1JAAHVMknqrm66QXk2VhXsK"]
25         }
26     }
27 }
28 }
```

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

```
1 ./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

```
1 ./xchain-cli vote --txid ↪
↪ f26d670b695d9fd5da503a34d130ef19e738b35e031b18b70ad4cbbf6dfe2656 --freeze 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 `-freeze` 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。

18.1 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。XuperUnion 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 `conf/xchain.yaml`，包含了大部分 Xuperunion 服务启动的配置项，其中有磁盘相关的章节

```
1 # 数据存储路径
2 datapath: ./data/blockchain
3
4 # 多盘存储的路径
5 datapathOthers:
6     - /ssd1/blockchain
7     - /ssd2/blockchain
8     - /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。

18.2 替换扩展插件

XuperUnion 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)

```
1 {
2   "crypto": [{
3     "subtype": "default",
4     "path": "plugins/crypto/crypto-default.so.1.0.0",
5     "version": "1.0.0",
6     "ondemand": false
7   }, {
8     "subtype": "schnorr",
9     "path": "plugins/crypto/crypto-schnorr.so.1.0.0",
10    "version": "1.0.0",
11    "ondemand": false
12  }]
13  # .....
14 }
```

需要替换插件则修改对应的.so 文件路径即可

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

19.1 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

19.2 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

19.3 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`

```
1 {  
2   "module_name": "native",      // native or wasm  
3   "contract_name": "eleccert", // contract name
```

(下页继续)

```
4  "method_name": "invoke",      // invoke or query
5  "args": {
6    "owner": "aaa",           // user name
7    "filehash": " 存证文件的 hash 值",
8    "timestamp": " 存证的 timestamp"
9  }
10 }
```

Query

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

```
1  {
2    "module_name": "native",    // native or wasm
3    "contract_name": "eleccert", // contract name
4    "method_name": "query",    // invoke or query
5    "args": {
6      "owner": "aaa",         // user name
7      "filehash": " 文件 hash 值"
8    }
9  }
10 // output
11 {
12   "filehash": " 文件 hash 值",
13   "timestamp": " 文件存入 timestamp"
14 }
```

代码样例参看: [contractsdk/go/example/erc721.go](#)

20.1 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。

20.2 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
 - `approveAll`: 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

20.3 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
```

```
1 {
2   "module_name": "native",      # native 或 wasm
3   "contract_name": "erc721",   # contract name
4   "method_name": "initialize", # initialize or query or invoke
5   "args": {
6     "from": "dudu",            # userName
7     "supply": "1,2"           # token_ids
8   }
9 }
```

Invoke

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

```
1 {
2   "module_name": "native",      # native 或 wasm
3   "contract_name": "erc721",   # contract name
4   "method_name": "invoke",     # initialize or query or invoke
5   "args": {
6     "action": "transfer",       # action name
7     "from": "dudu",            # usera
8     "to": "chengcheng",       # userb
9     "token_id": "1"           # token_ids
10  }
11 }
12 {
13   "module_name": "native",      # native 或 wasm
14   "contract_name": "erc721",   # contract name
15   "method_name": "invoke",     # initialize or query or invoke
16   "args": {
17     "action": "transferFrom",  # action name
18     "from": "dudu",           # userA
19     "caller": "chengcheng",   # userB
```

(下页继续)

(续上页)

```

20     "to": "miaomiao",      # userC
21     "token_id": "1"      # token_ids
22 }
23 }
24 {
25     "module_name": "native", # native 或 wasm
26     "contract_name": "erc721", # contract name
27     "method_name": "invoke", # initialize or query or invoke
28     "args": {
29         "action": "approve", # action name
30         "from": "dudu", # userA
31         "to": "chengcheng", # userB
32         "token_id": "1" # token_ids
33     }
34 }

```

Query

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

```

1 {
2     "module_name": "native", # native 或 wasm
3     "contract_name": "erc721", # contract name
4     "method_name": "query", # initialize or query or invoke
5     "args": {
6         "action": "balanceOf", # action name
7         "from": "dudu" # userA
8     }
9 }
10 {
11     "module_name": "native", # native 或 wasm
12     "contract_name": "erc721", # contract name
13     "method_name": "query", # initialize or query or invoke
14     "args": {
15         "action": "totalSupply" # action name
16     }
17 }
18 {
19     "module_name": "native", # native 或 wasm
20     "contract_name": "erc721", # contract name
21     "method_name": "query", # initialize or query or invoke

```

(下页继续)

(续上页)

```
22     "args": {
23         "action": "approvalOf",    # action name
24         "from": "dudu",           # userA
25         "to": "chengcheng"       # userB
26     }
27 }
```

智能合约 SDK 使用说明

XuperUnion 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 Contract Development Toolkit)，包含 C++ 语言和 Go 语言

21.1 C++ 接口 API

21.1.1 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

21.1.2 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

21.1.3 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

21.1.4 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

21.1.5 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

21.1.6 table

定义表格

```

1 // 表格定义以 proto 形式建立, 存放目录为 contractsdk/cpp/pb
2 syntax = "proto3";
3 option optimize_for = LITE_RUNTIME;
4 package anchor;
5 message Entity {
6     int64 id = 1;
7     string name = 2;
8     bytes desc = 3;
9 }
10 // table 名称为 Entity, 属性分别为 id, name, desc

```

初始化表格

```

1 // 定义表格的主键，表格的索引
2 struct entity: public anchor::Entity {
3     DEFINE_ROWKEY(name);
4     DEFINE_INDEX_BEGIN(2)
5     DEFINE_INDEX_ADD(0, id, name)
6     DEFINE_INDEX_ADD(1, name, desc)
7     DEFINE_INDEX_END();
8 };
9 // 声明表格
10 xchain::cdt::Table<entity> _entity;

```

put

```

1 template <typename T>
2 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

```

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

```

(下页继续)

(续上页)

```

11 self.get_entity().put(ent);
12 ctx->ok("done");
13 }

```

find

```

1 template <typename T>
2 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

```

1 DEFINE_METHOD(Anchor, get) {
2     xchain::Context* ctx = self.context();
3     const std::string& name = ctx->arg("key");
4     Anchor::entity ent;
5     if (self.get_entity().find({"name", name}, &ent)) {
6         ctx->ok(ent.to_str());
7         return;
8     }
9     ctx->error("can not find " + name);
10 }

```

scan

```

1 template <typename T>
2 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

```

1 DEFINE_METHOD(Anchor, scan) {
2     xchain::Context* ctx = self.context();
3     const std::string& name = ctx->arg("name");
4     const std::string& id = ctx->arg("id");
5     // const std::string& desc = ctx->arg("desc");
6     auto it = self.get_entity().scan({"id", id}, {"name", name});
7     Anchor::entity ent;
8     int i = 0;
9     std::map<std::string, bool> kv;
10    while(it->next()) {
11        if (it->get(&ent)) {
12            /*
13             std::cout << "id: " << ent.id() << std::endl;
14             std::cout << "name: " << ent.name() << std::endl;
15             std::cout << "desc: " << ent.desc() << std::endl;
16            */
17            if (kv.find(ent.name()) != kv.end()) {
18                ctx->error("find duplicated key");
19                return;
20            }
21            kv[ent.name()] = true;
22            i += 1;
23        } else {
24            std::cout << "get error" << std::endl;
25        }
26    }
27    std::cout << i << std::endl;
28    if (it->error()) {
29        std::cout << it->error(true) << std::endl;

```

(下页继续)

(续上页)

```

30     }
31     ctx->ok(std::to_string(i));
32 }

```

del

```

1  template <typename T>
2  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

```

1  DEFINE_METHOD(Anchor, del) {
2      xchain::Context* ctx = self.context();
3      const std::string& id= ctx->arg("id");
4      const std::string& name = ctx->arg("name");
5      const std::string& desc = ctx->arg("desc");
6      Anchor::entity ent;
7      ent.set_id(std::stoll(id));
8      ent.set_name(name.c_str());
9      ent.set_desc(desc);
10     self.get_entity().del(ent);
11     ctx->ok("done");
12 }

```

21.2 Go 接口 API

21.2.1 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, true	key 值查询成功, 返回 value 值
_, false	key 值不存在

21.2.2 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
true	存入 db 成功
false	存入 db 失败

21.2.3 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
true	删除成功
false	删除失败

21.2.4 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, true	查询交易成功, 得到此 txid 的 transaction
_, false	查询交易失败

21.2.5 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, true	查询 block 成功, 得到此 blockid 的 block
_, false	查询 block 失败

21.2.6 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	Iterator 的接口

样例

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

22.1 如何获取 XuperUnion

目前 XuperUnion 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

- XuperUnion v3.2
- XuperUnion v3.1

22.2 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。

22.3 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，默认的配置文件夹为 ./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；

```
1 log:
2 filepath: logs // 日志输出目录
3 filename: xchain // 日志文件名
4 console: true //是否答应 console 日志
5 level : trace // 日志等级, debug < trace < info < warn < error < crit
6 tcpServer:
7 port: :57404 // 节点 RPC 服务监听端口
8 p2pv2:
9 port: 47404 // 节点 p2p 网络监听的端口
10 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
11 miner:
12 keypath: ./data/keys //节点 address 目录
13 datapath: ./data/blockchain //账本存储目录
14 utxo:
15 cachesize: 5000 //Utxo 内存 cache 大小设置
16 tmplockSeconds: 60 //GenerateTx 的临时锁定期限, 默认是 60 秒
```


22.4 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
p2p	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
pluginmgr	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
xmodel	xmodel 实现 xmodel/pb 版本数据 pb 定义 xmodel/dbutils.go xmodel 通

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(BatchTx) 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

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。

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能合约。
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，一个区块中的交易数据。
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 DAG。
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，解决跨链问题。
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，通常用于智能合约。
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，公钥通常用于验证。
地址	与用户数据挂钩的最小单元。地址可以是合约账户，也可以是由公钥生成的。
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操作。
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块。
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，其总体目标是满足合约条款。
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执行。
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`