CITA-Cloud

Rivtower

2022 年 03 月 24 日

Contents

CITA-Cloud 是云原生的区块链定制框架，使企业用户能够快速构建面向自身业务的区块链系统。

- Github 主页
- 官方网站
- 技术论坛
- rfcs

CHAPTER 1

介绍

CITA-Cloud 由国内区块链行业的先驱和资深人士创建，他们致力于克服区块链技术在企业应用领域的难点。

通过早期在国产高性能联盟链 CITA 上开发以及落地应用方面的经验积累，发现联盟链虽然在架构上和公链大体一致，但在技术选型，规模、运维、互操作性等方面有很大的不同。

CITA-Cloud 是一个面向企业场景，**灵活**，**开放**，**可互操作**和**云原生**的区块链框架。

1.1 灵活

企业场景非常多变，单独一种实现无法满足所有场景。

CITA-Cloud 采用微服务架构，组件可以灵活替换，针对场景定制最适合的链。

1.1.1 快速定制

每个微服务可以有多种不同的实现，相互之间可以替换。

用户根据需要选择适合的组件，无需底层开发，就可以快速定制一条适合具体场景链。

比如，客户原来使用 *Fabric*，因此已经积累了一些使用 *chaincode* 编写的智能合约。

但是因为在某些项目中必须使用国密算法，而无法使用 *Fabric*。

使用 *CITA-Cloud* 框架，只要 *executor_chaincode* 和 *kms_sm* 两个微服务实现的组合，就能同时实现复用 *chaincode* 编写的智能合约和支持国密的目标。

在企业场景中，用户需求场景非常多变，共识算法也可以有不同的选择。

使用 *CITA-Cloud* 框架，在 *consensus_bft* 和 *consensus_raft* 两个共识微服务实现之间选择即可，不用任何的额外工作。

1.1.2 场景定制

如果已有的组件都不能满足用户需求，可以针对场景定制某个组件。

同时可以复用其他已有的组件，达到快速定制的目标。

例如，某著名科研机构使用 *CITA-Cloud* 框架，只用了两周的时间就完成其自主研发的国密算法实现到微服务的封装，快速完成特定场景的定制。

1.2 开放

CITA-Cloud 使用 Apache 2.0 开源协议，并提供定制所需的架构、开发工具，以及一个开放的社区。

1.2.1 架构

使用标准的微服务架构，本身只规定了微服务间的接口，给具体微服务实现留有非常大的发挥空间。

微服务划分采用正交分解方式，实现核心流程可定制，功能组件可替换的能力。

借鉴控制面和数据面分离的架构思想。将区块链相关的核心流程和核心数据全部集中在名为 *controller* 的微服务中，其他五个微服务则类似于数据平面，被动调用。

每一个微服务都能独立完成某项功能，接口能够自洽，保证每个微服务都是标准组件。

1.2.2 语言无关

微服务间通信使用 gRPC 和 ProtoBuf 的组合，各种语言的开发者都可以方便的参与。

在开发过程中可以选择最适合的语言，方便复用已有的软件栈或者库。

1.3 可互操作

随着联盟链在金融，政企领域的应用，越来越多的同构和异构的区块链应运而生。

在促进区块链生态环境日渐丰富的同时，也呈现出割裂和碎片化的趋势。

如何实现区块链之间的互操作，使不同区块链能够协同工作，这是一个非常重要的挑战。

1.3.1 智能合约生态

CITA-Cloud 可以通过替换 executor 微服务的实现来兼容多种智能合约引擎。

目前兼容了以太坊和 *Fabric* 两个最大的智能合约生态，未来还可以针对具体场景兼容更多的链的生态。

比如针对隐私，支持基于零知识证明的合约引擎。

1.3.2 跨链协议

CITA-Cloud 兼容陆羽跨链协议，实现对异构链的互操作。

陆羽跨链协议是一个面向可信源的互操作协议，旨在成为一套灵活、统一、可靠的互操作协议，实现对不同可信源的便捷接入与可靠操作。

1.4 云原生

区块链与云原生都是非常基础的分布式技术，采用云原生已有的发展思路对区块链来说是一条捷径，可以复用云原生社区成熟的资源。

联盟链的生态相比公链差很多，只有借助云原生社区的生态，才能有足够的发展空间。

区块链相比云原生更侧重去中心化的特性，两者可以互补。

1.4.1 复用成熟组件

区块链涉及的技术非常多，包括网络，存储，共识，智能合约引擎等。

完全自己开发，投入非常大，且达到企业级可靠性需要的时间比较漫长。

但是其中很多技术在云原生社区已经有成熟的组件。

CITA-Cloud 可以方便的将已有的成熟技术封装成微服务实现。

前面提到的 *Raft* 共识算法，复用 *PingCAP* 的*Raft*实现。

此实现在 *PingCA* 的产品中使用多年，经过生产环境的检验。

复用此实现，节省了大量的开发测试人力，也使得该微服务实现可靠性直接达到生产可用级别。

1.4.2 BaaS 服务

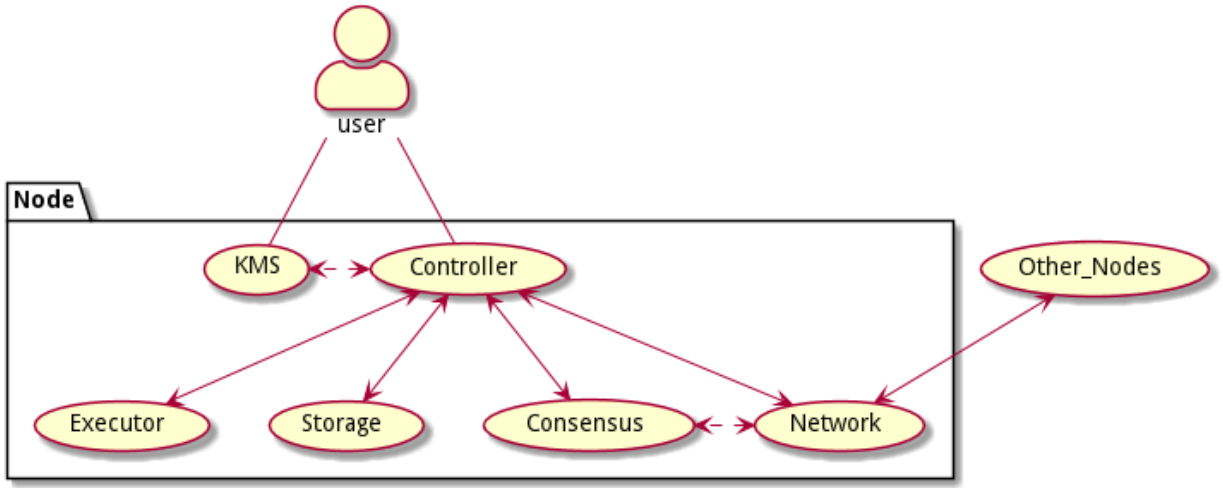
借助云原生的能力，CITA-Cloud 可以提升相关资源管理能力，实现自动化运维。

支持同一条链的多个节点部署在异地集群，甚至是不同实体的多个集群中。

CHAPTER 2

架构设计

CITA-Cloud 总体设计上采用微服务架构，划分为 Controller，Network，Consensus，Storage，Executor，KMS 六个微服务。



微服务架构图

微服务接口定义参见cita_cloud_proto。

微服务之间相互解耦，达到不同实现可以灵活替换，自由组合的目的。解耦设计的细节参见底层链技术白皮书。

为了能够快速构建起完整的，成熟的生态。在之前解耦的基础上，让解耦出的每一个微服务都能独立完成某项功能，每个微服务的接口能够自洽。使其不只是作为 CITA-Cloud 的组件存在，还可以拥有自己独立的生态。

2.1 Network

Network 微服务，维护与其他节点之间的网络连接，为本节点的其他微服务提供网络服务。

主要功能有收/发网络消息和节点管理，状态查询。

2.1.1 接收网络消息

收的部分采用了控制反转，收到的网络消息根据消息头中的 module 字段分发到其他微服务，并通过回调其他微服务的 gRPC 接口的方式在微服务间传递网络消息。

因此只有一个注册接口，用于其他需要网络服务的微服务注册信息。

```
// 注册网络服务接口
rpc RegisterNetworkMsgHandler(RegisterInfo) returns (common.StatusCode);

// 注册网络服务所需的信息
message RegisterInfo {
    string module_name = 1;  // 微服务名称
    string hostname = 2;     // 网络消息分发时，回调地址的域名
    string port = 3;         // 网络消息分发时，回调地址的端口
}

// 网络消息分发时，回调的 gRPC 接口
// 注册网络服务的其他微服务必须实现该接口
service NetworkMsgHandlerService {
    rpc ProcessNetworkMsg(NetworkMsg) returns (common.StatusCode);
}

// 网络消息结构
message NetworkMsg {
    string module = 1;  // 接收的微服务名称
    string type = 2;    // 消息类型，用于在同一个微服务内区分不同的消息
    uint64 origin = 3;  // 消息的节点标识
    bytes msg = 4;      // 消息数据
}
```

2.1.2 发送网络消息

发的部分提供了单播 (SendMsg) 和广播 (Broadcast) 两个接口。

```
// 发送消息给一个特定的节点
// 通过消息中的 origin 字段指定接收节点的标识
rpc SendMsg(NetworkMsg) returns (common.StatusCode);

// 广播消息
// 消息中的 origin 字段被忽略
rpc Broadcast(NetworkMsg) returns (common.StatusCode);
```

关于消息中的 origin 字段，在收到网络消息之后，需要对其进行一个处理。

发送时填的是接收节点的标识，接收到之后会将该字段修改为发送节点的标识。

2.1.3 状态查询

```
message NetworkStatusResponse {
    uint64 peer_count = 1;
}

rpc GetNetworkStatus(common.Empty) returns (NetworkStatusResponse);
```

查询网络连接状态的接口，返回当前连接的节点数量。

注意，这个数量里不包括节点自身。因此，4 个子节点的链，正常查询结果是 3。

```
message NodeNetInfo {
    string multi_address = 1;
    uint64 origin = 2;
}

message TotalNodeNetInfo {
    repeated NodeNetInfo nodes = 1;
}

rpc GetPeersNetInfo(common.Empty) returns (common.TotalNodeNetInfo);
```

查询节点网络信息的接口，返回连接的邻居节点的网络地址和标识信息。

注意：网络微服务的实现可以是任意的网络协议，为了兼容不同的协议，这里展示用的是multi_address。

2.1.4 节点管理

```
message NodeNetInfo {
    string multi_address = 1;
    uint64 origin = 2;
}

rpc AddNode(common.NodeNetInfo) returns (common.StatusCode);
```

增加节点信息的接口 (AddNode)，用于临时增加一个节点到网络中。

2.1.5 发展方向

独立出该微服务的初衷是网络部分比较复杂，希望该服务能隔离这部分复杂性，其他微服务就可以不用关心网络的具体情况。因此，其实现会朝着如下方向发展：

1. 处理复杂的网络场景。比如，p2p，防火墙穿透，虚拟私有网络等场景。
2. 对接多种协议。比如，TCP，UDP 等。
3. 提供更高的可靠性。比如，提供重发，限流，QoS，保证消息到达且仅到达一次等。

2.2 Storage

Storage 微服务，主要提供 KV 存储相关的功能，涵盖了常用的增删改查功能。

用于保存交易，区块和一些链相关的全局信息。

2.2.1 存储分区

针对区块链业务，预先定义了不同的 region，将不同类别的数据分别存放：

```
enum Regions {
    GLOBAL = 0;
    TRANSACTIONS = 1;
    HEADERS = 2;
    BODIES = 3;
    BLOCK_HASH = 4;
    PROOF = 5;
    RESULT = 6;
    TRANSACTION_HASH2BLOCK_HEIGHT = 7;
    BLOCK_HASH2BLOCK_HEIGHT = 8;  // In SQL db, reuse 4
    TRANSACTION_INDEX = 9;
```

(下页继续)

(续上页)

```
    COMPAT_BLOCK = 10;
    FULL_BLOCK = 11;
    BUTTON = 12;
}
```

GLOBAL 保存链相关的全局信息，比如当前链的最新高度，当前链的最新区块 Hash 等，对应的 key 为自定义的固定值。

TRANSACTIONS 保存交易哈希 -> 交易原始数据的对应关系。

HEADERS 保存区块高度 -> 区块头数据的对应关系。

BODIES 保存区块高度 -> 区块体数据的对应关系。

BLOCK_HASH 保存区块高度 -> 区块哈希的对应关系。

PROOF 保存区块高度 -> 区块证明的对应关系。

RESULT 保存区块高度 -> 区块执行结果的对应关系。

TRANSACTION_HASH2BLOCK_HEIGHT 保存交易哈希 -> 交易所在区块高度的对应关系。

BLOCK_HASH2BLOCK_HEIGHT 保存区块哈希 -> 区块高度的对应关系。即 BLOCK_HASH 的反查。

TRANSACTION_INDEX 保存交易哈希 -> 交易在所在区块中的序号的对应关系。

COMPAT_BLOCK 保存区块高度 -> 紧凑区块的对应关系。

COMPAT_BLOCK 保存区块高度 -> 完整区块的对应关系。

定制开发者可以根据自己的需要调整 region 列表。

2.2.2 store

```
message Content {
    uint32 region = 1;
    bytes key = 2;
    bytes value = 3;
}

// store key/value
rpc Store(Content) returns (common.StatusCode);
```

其中 region 即前述存储分区的枚举值。

注意:

1. key 和 value 类型为 bytes，需要调用方提前进行类型转换。
2. 其语义是 updata，同时包含增和改的功能。

2.2.3 load

```
message ExtKey {
    uint32 region = 1;
    bytes key = 2;
}

message Value {
    common.StatusCode status = 1;
    bytes value = 2;
}

// given a ext key return value
rpc Load(ExtKey) returns (Value);
```

2.2.4 delete

```
message ExtKey {
    uint32 region = 1;
    bytes key = 2;
}

// given a ext key delete it
rpc Delete(ExtKey) returns (common.StatusCode);
}
```

2.2.5 发展方向

联盟链的存储压力相较公链会大很多，可靠性要求也更高。因此，其实现会朝着如下方向发展：

1. 大数据量。比如，分布式数据库。
2. 更多功能。比如，冷热数据分离，备份等。

2.3 KMS

KMS 微服务，主要提供私钥加密存储，以及其他微服务需要的密码学服务。

目前提供区块链最基础的签名和哈希服务。

2.3.1 GetCryptoInfo

```
message GetCryptoInfoResponse {
    common.StatusCode status = 1;
    string name = 2;
    uint32 hash_len = 3;
    uint32 signature_len = 4;
    uint32 address_len = 5;
}

// Get crypto info
rpc GetCryptoInfo(common.Empty) returns (GetCryptoInfoResponse);
```

查询结果：

1. name 算法组合的名称。
2. hash_len 哈希算法得出的哈希值的字节长度。
3. signature_len 签名算法得出的签名的字节长度。
4. address_len 账户地址的字节长度。

2.3.2 签名

```
message GenerateKeyPairRequest {
    string Description = 1;
}

message GenerateKeyPairResponse {
    uint64 key_id = 1;
    bytes address = 2;
}

// Generate a KeyPair
rpc GenerateKeyPair(GenerateKeyPairRequest) returns (GenerateKeyPairResponse);
```

本接口生成一个账户的公私钥对，私钥加密后持久化保存在微服务内部，返回账户的序号和对应的账户地址。

```
message SignMessageRequest {
    uint64 key_id = 1;
    bytes msg = 2;
}

message SignMessageResponse {
```

(下页继续)

(续上页)

```
    common.StatusCode status = 1;
    bytes signature = 2;
}

// Sign a message
rpc SignMessage(SignMessageRequest) returns (SignMessageResponse);
```

本接口入参为签名所使用的账户的序号和要签名的消息，返回数字签名。

```
message RecoverSignatureRequest {
    bytes msg = 1;
    bytes signature = 2;
}

message RecoverSignatureResponse {
    common.StatusCode status = 1;
    bytes address = 2;
}

// Recover signature
rpc RecoverSignature(RecoverSignatureRequest) returns (RecoverSignatureResponse);
```

本接口入参为消息和其对应的数字签名，返回执行签名的账户地址。

注意：从接口定义看，似乎只能支持能恢复出公钥的签名算法。但是实际上可以把公钥附在签名后面，模拟出能恢复出公钥的签名算法。

2.3.3 哈希

```
message HashDataRequest {
    bytes data = 1;
}

message Hash {
    bytes hash = 1;
}

message HashResponse {
    StatusCode status = 1;
    Hash hash = 2;
}

// Hash data
```

(下页继续)

(续上页)

```
rpc HashData(HashDataRequest) returns (common.HashResponse);
```

本接口入参为要哈希的数据，返回哈希值。

```
message VerifyDataHashRequest {
    bytes data = 1;
    bytes hash = 2;
}

// Verify hash of data
rpc VerifyDataHash(VerifyDataHashRequest) returns (common.StatusCode);
```

本接口入参为要哈希的数据和相应的哈希值，返回校验结果。

2.3.4 CheckTransactions

```
// check transactions
rpc CheckTransactions(blockchain.RawTransactions) returns (common.StatusCode);
```

本接口用于批量校验交易。

因为交易验证过程中设计大量的密码学校验，如果每次都发起 `rpc` 调用性能会比较差。

可以认为是一个特殊的批量调用接口。

2.3.5 发展方向

密码学在区块链中至关重要，独立出该微服务的初衷是将系统与所使用的密码学算法解耦，方便将来替换密码学算法。

因此，其实现会朝着如下方向发展：

1. 新密码学算法的支持。比如，更加安全，更加高效的密码学算法。
2. 更多密码学相关功能集成。比如，对称加密，零知识证明等。

2.4 Executor

`Executor` 微服务，提供智能合约能力。

根据交易内容执行对应的智能合约，改变链上状态或者查询链上状态的。

2.4.1 Exec

```
// exec a block return executed_block_hash
rpc Exec(blockchain.Block) returns (common.HashResponse);
```

本接口入参为一个完整的区块数据，返回执行完区块中所有交易后的状态结果。

此结果数据类型为哈希值，类似以太坊的 `state_root`。

2.4.2 Call

```
message CallRequest {
    bytes to = 1;
    bytes from = 2;
    bytes method = 3;
    repeated bytes args = 4;
}

message CallResponse {
    bytes value = 1;
}

rpc Call(CallRequest) returns (CallResponse);
```

合约查询功能，调用合约中的指定方法，返回调用该方法的返回值。

2.4.3 发展方向

智能合约是区块链在可编程性方面很重要的功能。

该微服务只做了非常粗粒度的抽象，至于实现的细节，比如采用何种 `VM`；状态有哪些内容；状态如何组织和保存，都由具体实现来决定。

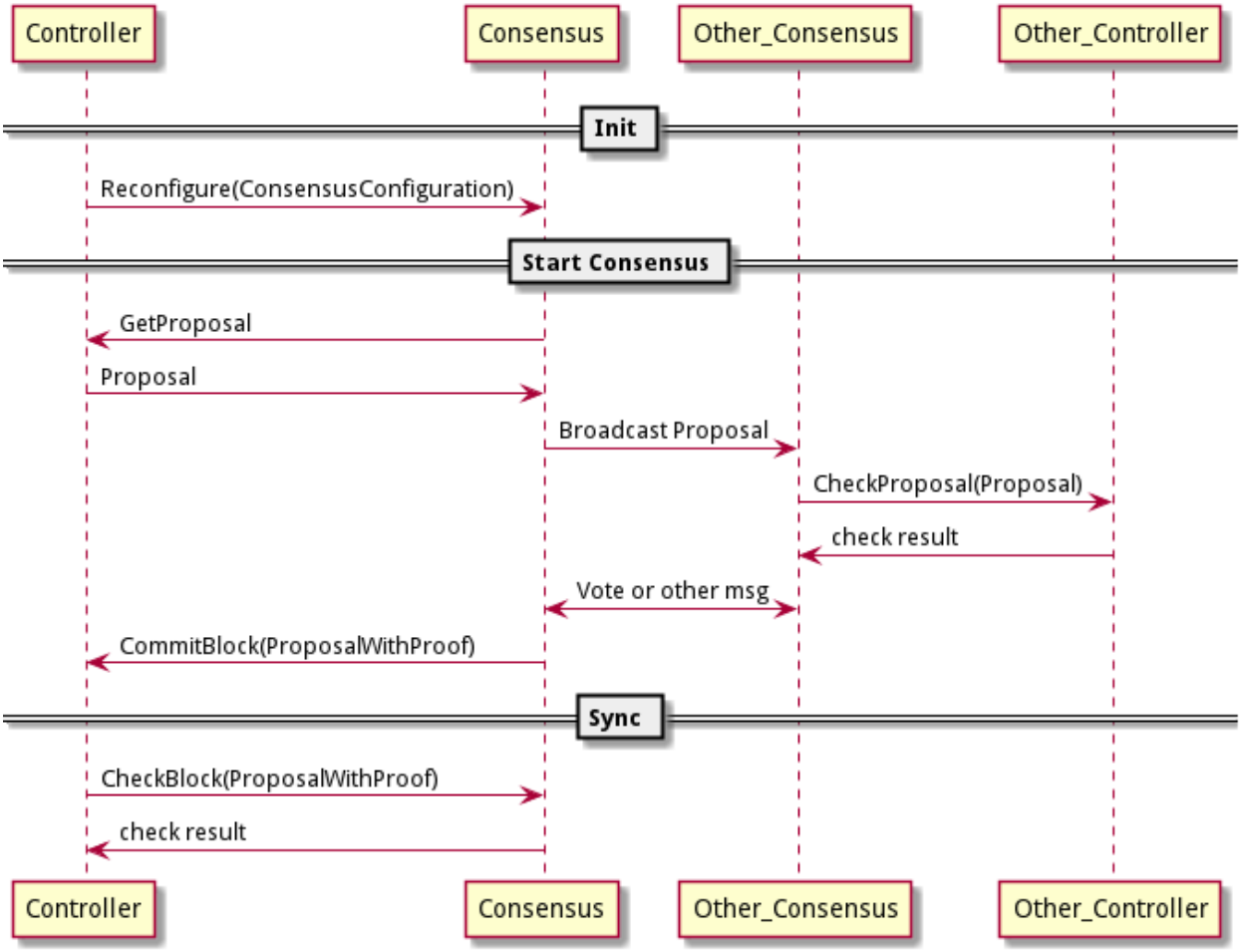
其实现会朝着如下方向发展：

1. 移植有广泛智能合约生态的引擎。比如，以太坊的 `EVM`。
2. 提供通用语言的 `Runtime`，使得用户可以用通用编程语言编写智能合约，降低合约开发门槛。
3. 针对一些特定应用场景，提供特定的 `VM` 和智能合约编程语言。比如可信计算，隐私计算，数据格式转换等。

2.5 Consensus

Consensus 微服务，主要提供让提案在多个共识参与方之间达成一致的功能。

单独这个微服务的功能，可以认为是一个分歧解决机。



共识微服务时序图

2.5.1 获取本节点提案

```
message Proposal {
    uint64 height = 1;
    bytes data = 2;
}

message ProposalResponse {
    StatusCode status = 1;
    Proposal proposal = 2;
```

(下页继续)

(续上页)

```
}

rpc GetProposal(common.Empty) returns (common.ProposalResponse);
```

该接口实现在 Controller 微服务中,Consensus 微服务去调用。

返回的提案数据类型为 bytes，因为 Consensus 微服务不需要了解提案的具体内容。

2.5.2 检查其他节点的提案

```
rpc CheckProposal(common.Proposal) returns (common.StatusCode);
```

该接口实现在 Controller 微服务中,Consensus 微服务去调用。

当本节点的 Consensus 微服务收到其他节点发送的提案，调用该接口检查提案是否合法。

2.5.3 提交共识结果

```
message ProposalWithProof {
    Proposal proposal = 1;
    bytes proof = 2;
}

message ConsensusConfiguration {
    uint64 height = 1;
    uint32 block_interval = 2;
    repeated bytes validators = 3;
}

message ConsensusConfigurationResponse {
    StatusCode status = 1;
    ConsensusConfiguration config = 2;
}

rpc CommitBlock(common.ProposalWithProof) returns (common.
↪ConsensusConfigurationResponse);
```

该接口实现在 Controller 微服务中,Consensus 微服务去调用。

共识达成之后，提交经过共识的提案以及相关证明，例如投票信息等。

Controller 微服务会给 Consensus 微服务返回新的配置信息。

目前配置信息包括:

1. 出块间隔。
2. 共识参与方账户地址列表。

2.5.4 检查同步的提案

```
rpc CheckBlock(common.ProposalWithProof) returns (common.StatusCode);
```

该接口实现在 Consensus 微服务中,Controller 微服务调用。

本节点进度落后的时候，Controller 微服务会从其他节点同步已经共识过的提案及相关的证明。

Controller 微服务本身无法验证证明是否合法，只能交由 Consensus 微服务来验证。

2.5.5 配置变更

```
rpc Reconfigure(common.ConsensusConfiguration) returns (common.StatusCode);
```

该接口实现在 Consensus 微服务中,Controller 微服务调用。

2.5.6 发展方向

共识是区块链非常核心的功能，但是共识算法实现非常多样化。

该微服务尽量做到抽象，以适应不同的共识算法。

其实现会朝着如下方向发展:

1. 更新更高效的共识算法。
2. 针对一些特定应用场景。比如非拜占庭容错的共识算法等。

2.6 Controller

Controller 微服务在整个区块链中处于核心的位置，主导所有主要的流程，并给上层用户提供 RPC 接口。

接口除了前述的针对 Consensus 微服务的接口，就是针对上层用户的 RPC 接口。其中最重要的是 SendRawTransaction 发送交易接口，剩下的都是一些信息查询接口。

单独就这个微服务来说，可以认为是一个提案管理系统。用户通过发送交易接口，提交原始交易数据，Controller 管理这些原始交易数据。通过计算原始交易数据的哈希，组装成区块，并形成 Consensus 需要的提案，管理这些提案。这里所说的管理，包括持久化，同步，以及验证其合法性。

Blockchain.proto 文件中定义了一套交易和区块的数据结构，但是前面所述的从原始交易数据如何产生最终 Consensus 需要的提案，并且这个过程还是要可验证的，这些都由具体实现决定。

2.6.1 发展方向

`Controller` 微服务是整个区块链系统的控制中枢，其内部逻辑和流程非常复杂，可定制部分也比较多。

未来我们会进一步梳理该微服务，并尝试提供一个框架，方便用户自定义流程，甚至是自定义交易和块等核心数据结构。

CHAPTER 3

组件

组件是各个微服务的实现。

每个组件单独一个代码仓库，仓库名称以微服务名称开头，下划线后接用于标识不同实现的名称。

构建物使用微服务名称，以便于相互替换。

比如 `storage_rocksdb`，是基于 `rocksdb` 实现的 `Storage` 微服务，其构建物为可执行文件 `storage`。

组件分为两类：

- 原厂组件，即 `CITA-Cloud` 自带的组件。
- 第三方组件。

组件还有以下一些指标：

1. 组件的成熟度：1-5，1 表示仅实现必要的功能的最小实现，5 表示非常成熟的实现。
2. 组件的状态：开发中，维护中，废弃。
3. 组件的授权状态：商业，或者开源。

3.1 原厂组件

3.1.1 network_p2p

介绍：基于网络库tentacle实现。

特点：

- 支持secio，通信加密保证安全。
- 支持多路复用 (yamux)，可以自定义协议。
- 支持节点发现，节点之间会自动交换连接的节点信息。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.2 network_tls

介绍：基于tokio-rustls实现。

特点：

- 支持 TLS1.3，通信加密保证安全。
- 使用标准的 x509 证书，方便复用已有的基础设施。
- 支持白名单，便于权限管理。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.3 network_quic

介绍：基于QUIC网络协议的实现。

特点：

- 高效，基于 UDP 协议，开销更小。
- 安全，默认支持 TLS，通信加密。
- 可靠，弱网络下效果更高。

代码仓库

镜像仓库

成熟度：2

状态：开发中

授权：开源，Apache-2.0 License

3.1.4 storage_rocksdb

介绍：基于rocksdb的实现。

特点：

- 高效，KV 数据库，读写效率高。
- 可靠，多数区块链项目都使用 rocksdb 作为存储引擎，稳定性好。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.5 storage_sled

介绍：基于sled的实现。

特点：

- 高效，读写效率高。
- 目前尚未稳定。

代码仓库

镜像仓库

成熟度：2

状态：开发中

授权：开源，Apache-2.0 License

3.1.6 kms_sm

介绍：国密算法的实现，使用 sm2 签名算法和 sm3 哈希算法。

特点：

- 符合中国国家密码标准。
- 高效，纯 Rust 实现，采用多种优化技术。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.7 kms_eth

介绍：兼容以太坊算法的实现，使用 secp256k1 签名算法和 keccak 哈希算法。

特点：

- 兼容以太坊。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.8 executor_evm

介绍：基于以太坊的 EVM 实现。

特点：

- 兼容以太坊的智能合约生态。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.9 consensus_bft

介绍：基于CITA-BFT实现。

特点：

- 拜占庭容错。
- 线性消息复杂度。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.10 consensus_raft

介绍：基于Raft实现。

特点：

- 非拜占庭容错。
- 成熟实现，稳定可靠。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.11 controller

介绍：目前唯一的 Controller 实现。

特点：

- 先共识后执行。
- 高性能，流水线式并行。
- utxo 模型的系统配置管理。
- 丰富的治理功能。

代码仓库

镜像仓库

成熟度：4

状态：维护中

授权：开源，Apache-2.0 License

3.1.12 废弃组件

network_direct

介绍：基于 tokio 网络库的实现。

特点：

- 网络直连，简单可靠
- 无通信加密

代码仓库

镜像仓库

成熟度：3

状态：废弃

授权：开源，Apache-2.0 License

废弃原因：被 network_tls 替代。因为区块链的去中心化属性，网络通信加密是比较基础的需求。

storage_sqlite

介绍：基于sqlite的实现。

特点：

- 轻量，嵌入式数据库，开销小。
- 功能丰富，完整支持 SQL。

代码仓库

镜像仓库

成熟度：3

状态：废弃

授权：开源，Apache-2.0 License

废弃原因：被 storage_rocksdb 替代。目前区块链的实现中主要还是以 KV 存储为主，SQL 数据库的优势发挥不出来，反而性能上不如 KV 数据库。也许将来对链上数据分析有更多需求的时候可以切换至 SQL 数据库。

storage_tikv

介绍：基于tikv的实现。

特点：

- 扩展能力强，分布式 KV 数据库。
- 稳定可靠，支持分布式事务操作，得到广泛应用。

代码仓库

镜像仓库

成熟度：2

状态：废弃

授权：开源，Apache-2.0 License

废弃原因：这个组件主要就是验证使用分布式 KV 数据库的可行性。但是目前数据量还没有到这个程度，所以暂时搁置。

executor_chaincode

介绍：实验性兼容 Fabric Chaincode 实现。

特点：

- 兼容 Fabric 的智能合约生态。

代码仓库

镜像仓库

成熟度：1

状态：废弃

授权：开源，Apache-2.0 License

废弃原因：本身就是实验性质的组件，为了验证框架有足够的灵活性。后期有相关需求之后完善之后成为 executor_chaincode_ext。

3.2 第三方组件

3.2.1 废弃组件

executor_chaincode_ext

介绍：增强型兼容 Fabric Chaincode 实现。

特点：

- 兼容 chaincode 合约。
- 支持了 CouchDB。
- 增加了 chaincode 事件相关功能。

代码仓库：无镜像仓库：无

成熟度：3

状态：废弃

授权：商业

废弃原因：专门为某个商业项目定制，后续没有类似的需求。

kms_sdibc

介绍：基于高性能国密算法实现。

特点：

- 性能好。

代码仓库：无镜像仓库：无

成熟度：4

状态：废弃

授权：商业

废弃原因：专门为某个商业项目定制，后续没有类似的需求。

CHAPTER 4

部署指南

因为采用微服务架构，相比单体软件来说，运维部署比较复杂。

例如：微服务之间如何相互调用；启动顺序如何保证；配置项如何管理等等。

所幸微服务架构已经非常流行，相应的基础设施也已经非常成熟。其中最重要的一点就是云原生技术的发展，它大大简化了微服务架构的应用在运维部署，配置管理方面的工作。

4.1 运行环境

CITA-Cloud 推荐的运行环境为 k8s 集群。

- 对于开发，可以是 minikube 等单机版本的 k8s 集群。
- 对于测试，可以是几台机器搭建的简单的 k8s 集群。
- 对于生产环境，推荐有专人维护的高可用 k8s 集群，或者云厂商提供的容器云方案。

其优点是：

- 不同环境的操作方式是统一的。
- 功能灵活，强大。可以实现各种复杂的配置，自动化运维等。
- 生态繁荣。基于云原生社区，有非常多成熟的配套工具和解决方案。

部署一条 CITA-Cloud 产生的链，除了准备好运行环境，还需要事先进行持久化存储和网络的设置。

4.2 持久化存储

链的节点是有状态的服务，需要挂载持久化存储保存数据。

为了方便对接不同类型的存储服务，我们使用了 k8s 中的 PV/PVC 概念对存储进行了抽象。

建议由运维人员配置 StorageClass，对 PV/PVC 实行动态绑定。

- 对于开发环境，可以使用简单的 hostPath，在宿主机本地存储。
- 对于测试环境，可以使用 NFS，由单独一台磁盘比较大的机器提供存储。
- 对于生产环境，推荐使用各种成熟的云存储，分布式存储，NAS 等专业存储系统。

4.3 网络

网络方面，需要微服务之间，以及节点之间可以通过网络相互访问。

目前推荐的部署方式是，一个节点一个 Pod，里面包含 6 个微服务的容器。

微服务之间可以直接通过本地环回网络通信。

节点间的网络通信，如果所有节点都在一个 k8s 集群内部，可以通过 k8s 的 Service 来暴露节点的网络端口。如果是跨集群的情况，则需要使用 NodePort 或者 LoaderBalancer 等服务对外暴露节点的网络端口。

4.4 工具

4.4.1 配置工具

当前的配置工具为cloud-config，用于生成一条链多个节点，以及每个节点内多个微服务的配置文件。

该配置工具支持常用的大部分组件；适用于各种场景，开发或是生产，单集群或者多集群；支持多种配置模式，集中式，去中心化方式。

具体使用方法，请参考代码仓库中的 README。

4.4.2 Chart 工程

为了简化部署工作，按照云原生的指导原则，提供了Charts 工程。

使用 Helm 工具，可以实现一条命令完成一条链的部署。

注意：

- 该 Charts 工程已经包含了配置功能，无需再单独使用配置工具。

- 该 Charts 工程支持多种场景和环境，单集群/多集群，开发环境/公有云环境。

具体使用方法，请参考代码仓库中的 README。

4.4.3 Operator

区块链类似于数据库，但与数据库又有一些不同的地方。相同的是都是有状态的服务，不同的是区块链在备份，迁移，升级，扩容等运维操作时都有特殊的要求。

- 对于备份来说，区块链每个节点都是一个副本，相当于自带热备方案。冷备也跟数据库不一样，不是按时间（比如每天备份），而是要考虑区块高度。
- 对于迁移和升级来说，区块链的节点是有身份的，因此不能先部署新的节点，再停老的节点，而是要反过来。
- 对于扩容来说，增加节点并不能提升区块链的性能，反而会造成反效果，只能扩容 cpu，内存。存储也必须所有节点一起扩充。

因此，比较简单的配置部署工作，Charts 工程就足以支撑。一旦涉及到比较复杂的运维操作，则需要一些定制开发。而 Operator 在 k8s 领域就是用来扩展自定义功能的。

我们的 Operator 以 CRD 的形式自定义了 Account/Chain/Node 等高层的资源类型，并提供了对这些自定义资源的操作方法。进一步简化了配置部署工作，同时也为将来提供复杂的运维功能打下了坚实的基础。

基于 Operator 的工具有以下几个部分：

1. Operator 本身cita-cloud-operator。
2. 代理服务端operator-proxy。
3. 代理客户端 cco-cli。

注意：

- Operator 已经包含了配置功能，无需再单独使用配置工具。
- cita-cloud-operator作为一个标准的 Operator 可以单独使用。
- 代理服务端和客户端 cco-cli 进一步降低了操作难度，无需用户编写 yaml 文件。

具体使用方法，请参考代码仓库中的 README。

CHAPTER 5

快速入门

本章介绍的是使用 minikube 作为运行环境，使用默认推荐的组件，快速搭建一条链的操作方法。

关于更加深入的定制操作，请参阅定制章节。

5.1 环境准备

5.1.1 硬件配置建议

- CPU：4 核或以上
- 内存：8GB 或以上
- 硬盘：30G 或以上

5.1.2 软件依赖

操作系统

常见的 Linux 发行版本均可，例如：CentOS，Debian，Ubuntu 等等。

docker

安装方法参见官方文档。

Helm

Helm 是 Kubernetes 的包管理器，是寻找、共享和使用为 Kubernetes 构建的软件的最佳方式。

安装方法参见文档。

minikube

安装方法参见官方文档。

安装完成后用下面的命令启动 minikube，国内需要在启动 minikube 时设置一些镜像参数。

注意不能使用 root 权限启动 minikube。

```
minikube start --registry-mirror=https://hub-mirror.c.163.com --image-
↪repository=registry.cn-hangzhou.aliyuncs.com/google_containers --vm-driver=docker --
↪alsologtostderr -v=8 --base-image registry.cn-hangzhou.aliyuncs.com/google_
↪containers/kicbase:v0.0.17
```

耐心等待，看到以下信息代表启动成功。

```
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace␣
↪by default
```

kubectl

kubectl 是 Kubernetes 集群的命令行工具，通过 kubectl 能够对集群本身进行管理，并能够在集群上进行容器化应用的安装部署。

安装方法参见官方文档。

cloud-cli

该工具为 CITA-Cloud 链的命令行客户端，可以方便的对链进行常用的操作。

使用方法参见文档。

```
$ wget https://github.com/cita-cloud/cloud-cli/releases/download/v0.3.0/cldi-x86_64-
↪unknown-linux-musl.tar.gz
$ tar zxvf cldi-x86_64-unknown-linux-musl.tar.gz
$ sudo mv ./cldi /usr/local/bin/
```

(下页继续)

(续上页)

```
$ cldi -h
$ cldi -h
cldi 0.3.0
Rivtower Technologies <contact@rivtower.com>
The command line interface to interact with CITA-Cloud

USAGE:
    cldi [OPTIONS] [SUBCOMMAND]

OPTIONS:
    -c, --context <context>       context setting
    -r <controller-addr>          controller address
    -e <executor-addr>            executor address
    -u <account-name>             account name
        --crypto <crypto-type>    The crypto type of the target chain [possible␣
↪values: SM, ETH]
    -h, --help                    Print help information
    -V, --version                 Print version information

SUBCOMMANDS:
    get             Get data from chain
    send            Send transaction
    call            Call executor
    create          create an EVM contract
    context         Context commands
    account         Account commands
    admin           The admin commands for managing chain
    rpc             Other RPC commands
    ethabi          Ethereum ABI coder.
    bench           Simple benchmarks
    watch           Watch blocks
    completions     Generate completions for current shell. Add the output script to `.
↪profile`
                        or `.bashrc` etc. to make it effective.
    help            Print this message or the help of the given subcommand(s)
```

5.2 运行链

5.2.1 添加 Charts 仓库

```
$ helm repo add cita-cloud https://cita-cloud.github.io/charts
$ helm repo update
$ helm search repo cita-cloud/
NAME                                            CHART VERSION   APP VERSION     ␣
↪DESCRIPTION
cita-cloud/cita-cloud-aliyun-lb                 6.3.3           6.3.3            Setup␣
↪CITA-Cloud node SLB in aliyun
cita-cloud/cita-cloud-config                    6.3.3           6.3.3           ␣
↪Create a job to change config of CITA-Cloud blo...
cita-cloud/cita-cloud-huaweiyun-lb              6.3.3           6.3.3            A␣
↪Helm chart for Kubernetes
cita-cloud/cita-cloud-local-cluster             6.3.3           6.3.3            Setup␣
↪CITA-Cloud blockchain in one k8s cluster
cita-cloud/cita-cloud-multi-cluster-node        6.3.3           6.3.3            Setup␣
↪CITA-Cloud node in multi k8s cluster
cita-cloud/cita-cloud-nodeport                  6.3.3           6.3.3            A␣
↪Helm chart for Kubernetes
cita-cloud/cita-cloud-porter-lb                 6.3.3           6.3.3            Setup␣
↪porter Loadbalancer for CITA-Cloud node
cita-cloud/cita-cloud-pvc                       6.3.3           6.3.3           ␣
↪Create PVC for CITA-Cloud
```

5.2.2 创建 PVC

PVC(PersistentVolumeClaim) 是对 PV 的申请 (Claim)。PVC 通常由普通用户创建和维护。需要为 Pod 分配存储资源时，用户可以创建一个 PVC，指明使用的 StorageClass，Kubemetes 会动态创建并绑定相关的 PV。

这里使用的是 minikube 自带的名为 standard 的 StorageClass。

```
$ helm install local-pvc cita-cloud/cita-cloud-pvc --set scName=standard
$ kubectl get pvc
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS␣
↪MODES   STORAGECLASS    AGE
local-pvc   Bound    pvc-fd3eaebd-3413-4205-b88a-dbc6cee9a057   10Gi       RWO       ␣
↪    standard        18m
```

对应的路径在 minikube 虚拟机内的/tmp/hostpath-provisioner/default/local-pvc。

注意：如果 minikube 版本为 v1.20.0，这里会有一个 bug。详细情况和解决方法参见链接。

5.2.3 生成超级管理员账户

使用 `cldi` 创建账户

```
$ cldi account generate -h
cldi-account-generate
generate a new account

USAGE:
    cldi account generate [OPTIONS]

OPTIONS:
        --name <name>              The name for the new generated account, default to␣
↪account address
    -p, --password <password>      The password to encrypt the account
        --crypto <crypto-type>     The crypto type for the generated account.␣
↪[default: <current-
                                   context-crypto-type>] [possible values: SM, ETH]
    -h, --help                     Print help information
```

为了演示方便，这里不设置密码，加密算法也使用默认值。

```
$ cldi account generate --name admin
{
  "crypto_type": "SM",
  "address": "0xc8ca9cc77a7f822fdd0baef7a7740f9dba493455",
  "public_key":
↪"0x0d9edfd3889ec752e92fb1aa53fdfc26512c6a0ea39deb12510e7ac4d0915c4d4f0a18a3c1cf2a5950319d429af38b1
↪",
  "secret_key": "0x50dc0c1655419938d83d924a3c3b4cbbd57de5df901ce4772272445605a52d43"
}
```

5.2.4 启动链

```
$ helm install test-chain cita-cloud/cita-cloud-local-cluster --set config.
↪superAdmin=0xc8ca9cc77a7f822fdd0baef7a7740f9dba493455
NAME: test-chain
LAST DEPLOYED: Thu Mar 24 02:48:52 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

该命令会创建一条有 4 个节点，名为 `test-chain` 的链。

注意：`superAdmin` 参数必须设置为自己生成的账户地址，此处仅为演示，切勿在正式环境中使用演示值。

5.2.5 查看运行情况

```
$ kubectl get pod
NAME                                      READY   STATUS    RESTARTS   AGE
test-chain-0                              7/7     Running   0          8m3s
test-chain-1                              7/7     Running   0          8m3s
test-chain-2                              7/7     Running   0          8m3s
test-chain-3                              7/7     Running   0          8m3s
```

查看日志:

```
$ minikube ssh
docker@minikube:~$ tail -10f /tmp/hostpath-provisioner/default/local-pvc/test-chain-0/
↪logs/controller-service.log
2022-03-24T02:57:41.562867997+00:00 INFO controller::node_manager - update node:␣
↪0x6028b1113a9ac5f79d2fdfb37ca135812d675691
2022-03-24T02:57:43.863863944+00:00 INFO controller::controller - chain_check_
↪proposal: add remote␣
↪proposal(0x90543745dee079d9a37d2b5bd1e026ad092089c1f3fd88ebbc16b10b3d1926f3)
2022-03-24T02:57:43.864261069+00:00 INFO controller::controller - chain_check_
↪proposal: finished
2022-03-24T02:57:44.441848936+00:00 INFO controller::chain - height: 103 hash␣
↪0x90543745dee079d9a37d2b5bd1e026ad092089c1f3fd88ebbc16b10b3d1926f3
2022-03-24T02:57:44.672342679+00:00 INFO controller::node_manager - update node:␣
↪0x6028b1113a9ac5f79d2fdfb37ca135812d675691
2022-03-24T02:57:44.672366020+00:00 INFO controller::controller - update global␣
↪status node(0x6028b1113a9ac5f79d2fdfb37ca135812d675691) height(103)
2022-03-24T02:57:44.673242652+00:00 INFO controller::chain - exec_block(103): status:␣
↪Success, executed_block_hash:␣
↪0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
2022-03-24T02:57:44.783682991+00:00 INFO controller::chain - finalize_block: 103,␣
↪block_hash: 0x90543745dee079d9a37d2b5bd1e026ad092089c1f3fd88ebbc16b10b3d1926f3
```

5.3 基本操作

5.3.1 指定链的 RPC 端口

链有两个 `rpc` 地址，分别是 `controller` 和 `executor` 微服务。

我们可以通过`-r` 和`-e` 来告诉 `cldi` 如何访问链:

默认链会开启 `nodeport`，端口分别为 30004 和 30005。

参数为:

```
-r `minikube ip`:30004 -e `minikube ip`:30005
```

注意：这里 minikube 可能出现 `Service` 端口映射问题。可以换一种操作方式。

```
$ kubectl port-forward pod/test-chain-0 50002:50002 50004:50004
```

参数为：

```
-r localhost:50004 -e localhost:50002
```

5.3.2 查看块高

```
$ cldi -r localhost:50004 -e localhost:50002 get block-number
246
```

5.3.3 查看系统配置

```
$ cldi -r localhost:50004 -e localhost:50002 get system-config
{
  "admin": "0xc8ca9cc77a7f822fdd0baef7a7740f9dba493455",
  "admin_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
  "block_interval": 3,
  "block_interval_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
  "chain_id": "0x63586a3c0255f337c77a777ff54f0040b8c388da04f23ecee6bfd4953a6512b4",
  "chain_id_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
  "emergency_brake": false,
  "emergency_brake_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
  "validators": [
    "0x40283e85bfbe778a0ef0ee80688861ccc2c557a2",
    "0x7b9e332f91fe894da0260fe2207f021d2d24008c",
    "0x014146185e3b32e64f0d06021aa6fff8639d134a",
    "0x6028b1113a9ac5f79d2fdfb37ca135812d675691"
  ],
  "validators_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
  "version": 0,
```

(下页继续)

(续上页)

```
  "version_pre_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

5.3.4 停止链

```
$ helm uninstall test-chain
release "test-chain" uninstalled
```

5.3.5 删除链

```
$ helm install clean cita-cloud/cita-cloud-config --set config.action.type=clean
NAME: clean
LAST DEPLOYED: Thu Mar 24 02:47:45 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

注意：该命令将永久性的删除链的所有数据，请谨慎操作。

5.4 账户操作

5.4.1 创建账户

```
$ cldi account generate --name user
{
  "crypto_type": "SM",
  "address": "0xf4b80a27b7d526028183e705604b865c1458c838",
  "public_key":
↪"0x71ebc3701780b4ac6a6ae0817e2fa402fb26082e6eade00f117015a1063a5c3af6e1f74c26c01f87af5ddea20fb28ff
↪",
  "secret_key": "0x2757242a6138e9617b30ec63f6a240b880d25851d86d9d15849f2d45061d3288"
}
```

5.4.2 选择操作使用的账户

```
$ cldi -u user -r localhost:50004 -e localhost:50002
cldi>
```

-u 选择使用该账户，后续将以该账户的身份发送交易。

进入 cldi 的交互式模式。

5.5 发送交易

5.5.1 编译合约

这里以 Counter 合约为例:

```
$ cat Counter.sol
pragma solidity ^0.4.24;

contract Counter {
    uint public count;

    function add() public {
        count += 1;
    }

    function reset() public {
        count = 0;
    }
}

$ curl -o solc -L https://github.com/ethereum/solidity/releases/download/v0.4.24/solc-
↪static-linux
$ chmod +x solc
$ sudo mv ./solc /usr/local/bin/
$ solc --hashes --bin Counter.sol

======= Counter.sol:Counter =======
Binary:
608060405234801561001057600080fd5b5060f58061001f6000396000f3006080604052600436106053576000357c0100000
Function signatures:
4f2be91f: add()
06661abd: count()
d826f88f: reset()
```

5.5.2 创建合约

```
cldi> create␣
↪0x608060405234801561001057600080fd5b5060f58061001f6000396000f3006080604052600436106053576000357c010
0xf18153579e86b4d81617bf9d3b34e1ca2d0433296927f4b04d5c5219d2d82d46
```

创建合约的参数是编译合约输出的二进制字节码，注意前面要增加 0x 前缀。

返回值为这笔创建合约交易的交易哈希。

5.5.3 查看交易回执

```
cldi> get receipt 0xf18153579e86b4d81617bf9d3b34e1ca2d0433296927f4b04d5c5219d2d82d46
{
  "block_number": 454,
  "contract_addr": "0xa4582f4966bdef3a2839e2f256a714426508ddb7",
  "cumulative_quota_used":
↪"0x00000000000000000000000000000000000000000000000000000000000018ed3",
  "error_msg": "",
  "legacy_cita_block_hash":
↪"0x09b2e445d5b3a5e118bb0aad8d812d237ff24f2aa9e37da75e5c68d43ddcdbba",
  "logs": [],
  "logs_bloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
  "quota_used": "0x00000000000000000000000000000000000000000000000000000000000018ed3",
  "state_root": "0x28913fc520eaf50c72fad929d0f57c94ad487bfd63278a57cb40ef370a7697a0",
  "tx_hash": "0xf18153579e86b4d81617bf9d3b34e1ca2d0433296927f4b04d5c5219d2d82d46",
  "tx_index": 0
}
```

参数为前一步操作返回的交易哈希。

返回值为该笔交易的执行结果信息，其中 contract_addr 字段为刚才部署的合约的地址。

得到合约地址后，就可以调用其中的方法。

5.5.4 查询合约状态

查询合约中的 count 值：

```
cldi> call 0xa4582f4966bdef3a2839e2f256a714426508ddb7 0x06661abd
0x0000000000000000000000000000000000000000000000000000000000000000
```

第一个参数为合约地址。

第二个参数为要调用的 count() 方法的函数签名。

返回值为 count 的当前值。

5.5.5 发送交易

向合约发送交易，调用 add 方法改变 count 的值：

```
cldi> send 0xa4582f4966bdef3a2839e2f256a714426508ddb7 0x4f2be91f
0x24f0bc60340c49e875a8701e80849725a0a10e1bf47990e8c9cb399e31acea05
```

第一个参数为合约地址。

第二个参数为要调用的 add() 方法的函数签名。

等待交易上链之后，再次查询就可以发现合约状态有了变化：

```
cldi> call 0xa4582f4966bdef3a2839e2f256a714426508ddb7 0x06661abd
0x0000000000000000000000000000000000000000000000000000000000000001
```

CHAPTER 6

定制

CITA-Cloud 本身不是一条链，而是一个区块链定制框架，定位类似于 Substrate 和 Cosmos SDK。

但是 Substrate 和 Cosmos SDK 更类似于传统的应用开发框架，比如 Java 的 Spring。框架提供一些现成的功能组件和代码自动生成的功能，目的是简化定制开发工作，但是仍然需要进行代码开发。

而 CITA-Cloud 沿袭了一直以来的微服务架构，并进一步结合了云原生的思想，更类似于 PaaS 化的开发框架。用户可以不需要代码开发，通过选择并配置组件来构建一条链；也可以根据自己的特殊需求来定制开发相应的组件，结合已有的其他组件构建一条链。

6.1 定制链

CITA-Cloud 划分为 Controller，Network，Consensus，Storage，Executor，KMS 六个微服务，详情参见架构设计章节。

用户可以从现有的组件（参见组件章节）中，根据自己的场景选择 6 个组件，每个组件必须对应一个前述的微服务，即可组合成一条链。类似于：

具体配置方法，参考部署指南章节配置工具部分。

6.2 定制组件

定制有两种方式：

1. `fork` 现有组件（参见组件章节），对其进行定制化开发。
2. 已有对应某个微服务的功能完善的库，新实现一个组件。

对于后一种情况，参考架构设计章节对应的微服务部分，封装已有的库，实现对应的 `gRPC` 接口，然后提供对应的 `Docker` 镜像。

CHAPTER 7

路线图

- 2020.4 项目启动
- 2020.8 白皮书发布
- 2020.10 首个版本发布
- 2021.7 v6.0.0 发布，协议稳定，将作为长期支持大版本
- 2022.3 引入 `Rollup` 方案，协议再次大升级，筹备 v7.0.0

CHAPTER 8

版本发布

8.1 最新版本

8.1.1 v6.3.3

本次版本在稳定性和运维方面做了优化，主要更新内容如下：

1. 修复上个版本 `WAL` 功能引入的问题。
2. 支持消块工具。可以修复因为意外导致区块链节点数据不一致的问题。
3. 梳理微服务的命令行参数。参数统一从配置文件传递，配置更清晰，更统一。也为将来支持 `gitops` 打下基础。
4. `network` 支持配置文件热更新。增加删除节点之后，已有节点可以自动感知到网络的变化。
5. `CLI` 重构。新增交互式模式，优化用户体验。参见文档。
6. 更新文档。新的文档更加面向链的定制开发者。参见文档。
7. `k8s Operator` 实验性支持。参见项目。

修改详情：

Controller

[Feature]

[feat] update readme @rink1969 @JLerxky

[feat] Support cloud-op @Pencil-Yao @JLerxky

[Fix]

[fix] Add the judgment of not redoing wal @JLerxky

[fix] save current_block_hash before saving current_block_height @Pencil-Yao

[Optimization]

[optim] handle add existed peer @Jayanring

[optim] use cloud_util::wal @JLerxky

[optim] run -c & -l @JLerxky

consensus_raft

[Feature]

[feat] update readme @rink1969 @NaughtyDogOfSchrodinger

[feat] support recover @Pencil-Yao

consensus_bft

[Feature]

[feat] update readme @NaughtyDogOfSchrodinger

[feat] Support cloud-op @Pencil-Yao @JLerxky

[Optimization]

[optim] use cloud_util::wal @JLerxky

[optim] run -c & -l @JLerxky

cloud-config

[Feature]

[feat] Update k8s subcmd @NaughtyDogOfSchrodinger

[feat] output stdout and log file at same time @rink1969

[Fix]

[fix] fix typo @rink1969

[fix] add license and copyright info @rink1969

[fix] update images @rink1969

[fix] fix update node, protect rewrite file in used @rink1969

[Optimization]

[optim] github ci & fix: fmt @Pencil-Yao @Jayanring

[chore] update clap @JLerxky

executor_evm

[Feature]

[feat] update readme @rink1969

[feat] Support cloud-op @Pencil-Yao @JLerxky

[Optimization]

[optim] run -c & -l @JLerxky

storage_rocksdb

[Feature]

[feat] update readme @rink1969

[feat] Support cloud-op @Pencil-Yao @JLerxky

[Optimization]

[optim] run -c & -l @JLerxky

kms_sm

[Feature]

[feat] remove create subcmd; update example and readme @rink1969

[Optimization]

[optim] run -c & -l @JLerxky

network_tls

[Feature]

[feat] update readme @rink1969

[feat] support hot update + peer-count + origin @Pencil-Yao @Jayanring

[Optimization]

[optim] run -c @JLerxky

network_p2p

[Feature]

[feat] update readme @rink1969

[Optimization]

[optim] run -c & -l @JLerxky

kms_eth

[Feature]

[feat] remove create subcmd; update example and readme @rink1969

[Fix]

[fix] check invalid msg @rink1969

[Optimization]

[optim] run -c & -l @JLerxky

兼容性问题

1. 数据兼容

   v6.3.3 与 v6.3.2 数据完全兼容。旧有的 v6.3.2 链跑出来的数据，只需要停链然后使用新的 v6.3.3 镜像启动，就可以正常出块。

2. 版本兼容

   v6.3.3 与 v6.3.2 版本的节点可以混合组成网络，并正常出块

3. 配置变更

   微服务的参数有变化，但是采用默认值的情况下，行为与 v6.3.2 一致。

   如果有自行设置参数的情况，可能需要进行修改。

8.2 历史版本

8.2.1 v6.3.2

本次版本更新对稳定性和运维作了优化，稳定性方面链能够在高负载的情况下更稳定的运行，主要更新内容如下：

1. 优化了 `consensus_bft` 由于网络时延问题导致的无法正常出块的问题；
2. `controller` 具备 WAL 的能力，可以处理进程突然被杀掉而引发的存储问题。
3. 升级的微服务添加了 apache license

修改详情：

Controller

[Feature]

[feat] add wal [@JLerxky @Pencil-Yao @Jayanring]

[feat] support retransmit chain_status_init regularly [@Pencil-Yao]

[feat] store genesis system config [@JLerxky @Jayanring]

[Fix]

[fix] executor init [@Pencil-Yao]

[fix] in csi mode, can’t send block request [@Pencil-Yao]

[fix] dup tx in busy environment [@Pencil-Yao]

[Optimization]

[optim] confirm executor when init chain [@Jayanring]

[optim] process earlystatus logic [@Pencil-Yao]

[optim] chain lock [@Pencil-Yao]

Consensus_bft

[Fix]

[fix] handle leader_commit error with dup-tx [@Pencil-Yao]

Executor_evm

[Feature]

[feat] hanle same or invalid block re-enter [@Pencil-Yao]

[Optimization]

[optim] use reenter-invalid block code [@Pencil-Yao]

Cloud-config

[Feature]

[feat] rewrite! from cita_cloud_config [@NaughtyDogOfSchrodinger, @Pencil-Yao]

[feat] support config [@NaughtyDogOfSchrodinger]

[BugFix]

[fix] fix series 6.3.0 adaption problem [@Pencil-Yao, @NaughtyDogOfSchrodinger]

兼容性问题

1. 数据兼容

   v6.3.2 与 v6.3.1 数据完全兼容。旧有的 v6.3.1 链跑出来的数据，只需要停链然后使用新的 v6.3.2 镜像启动，就可以正常出块。

2. 版本兼容

   v6.3.2 与 v6.3.1 版本的节点可以混合组成网络，并正常出块

8.2.2 v6.3.0

版本发布说明

8.2.3 v6.2.0

版本发布说明

8.2.4 v6.1.0

版本发布说明

8.2.5 v6.0.0

版本发布说明

8.2.6 v5.0.0

版本发布说明

8.2.7 v4.0.0

版本发布说明

8.2.8 v3.0.0

版本发布说明

CHAPTER 9

常见问题