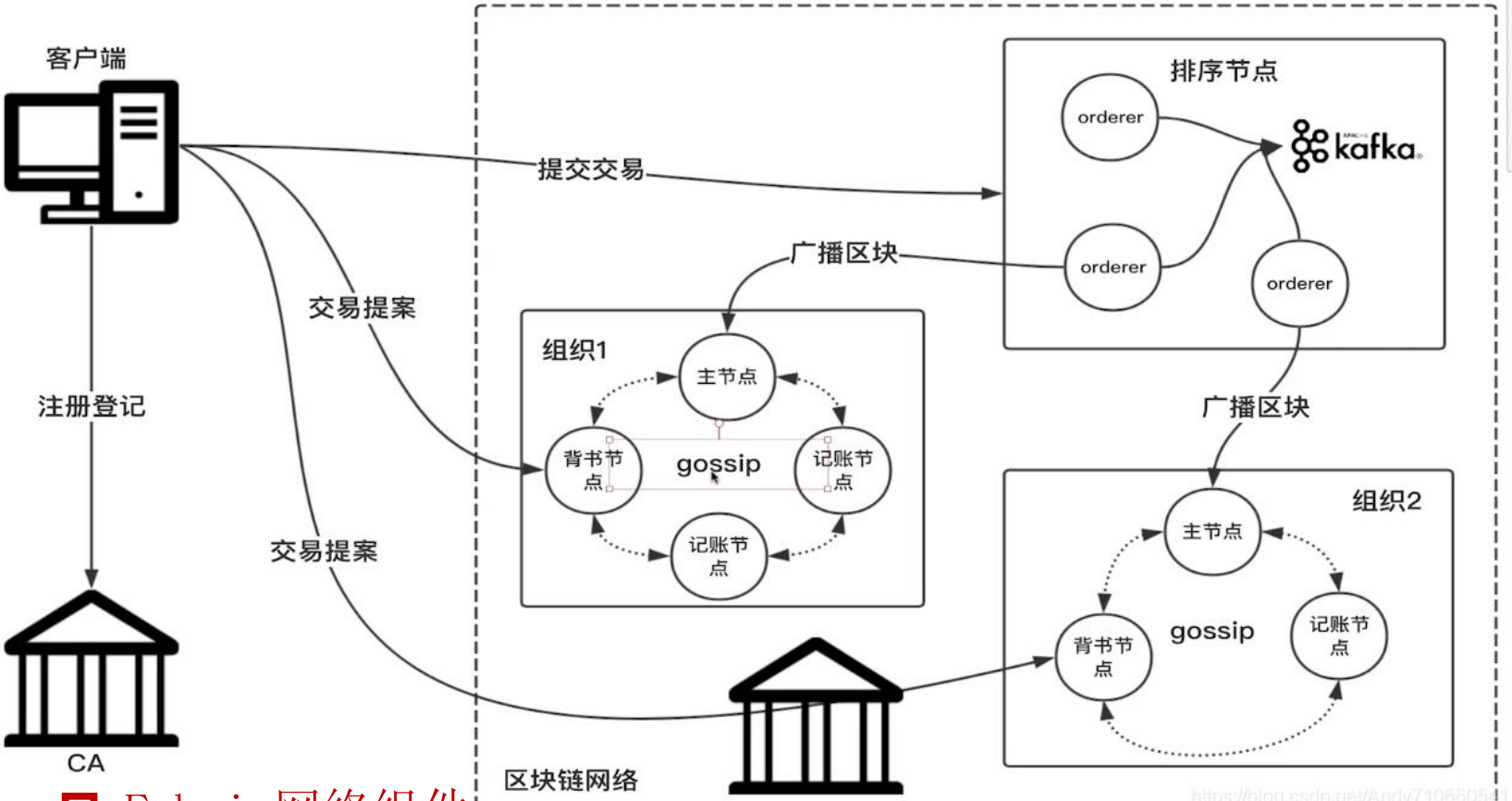
联盟链技术
Alliance-chain Technology

北京交通大学
计算机与信息技术学院
信息安全系

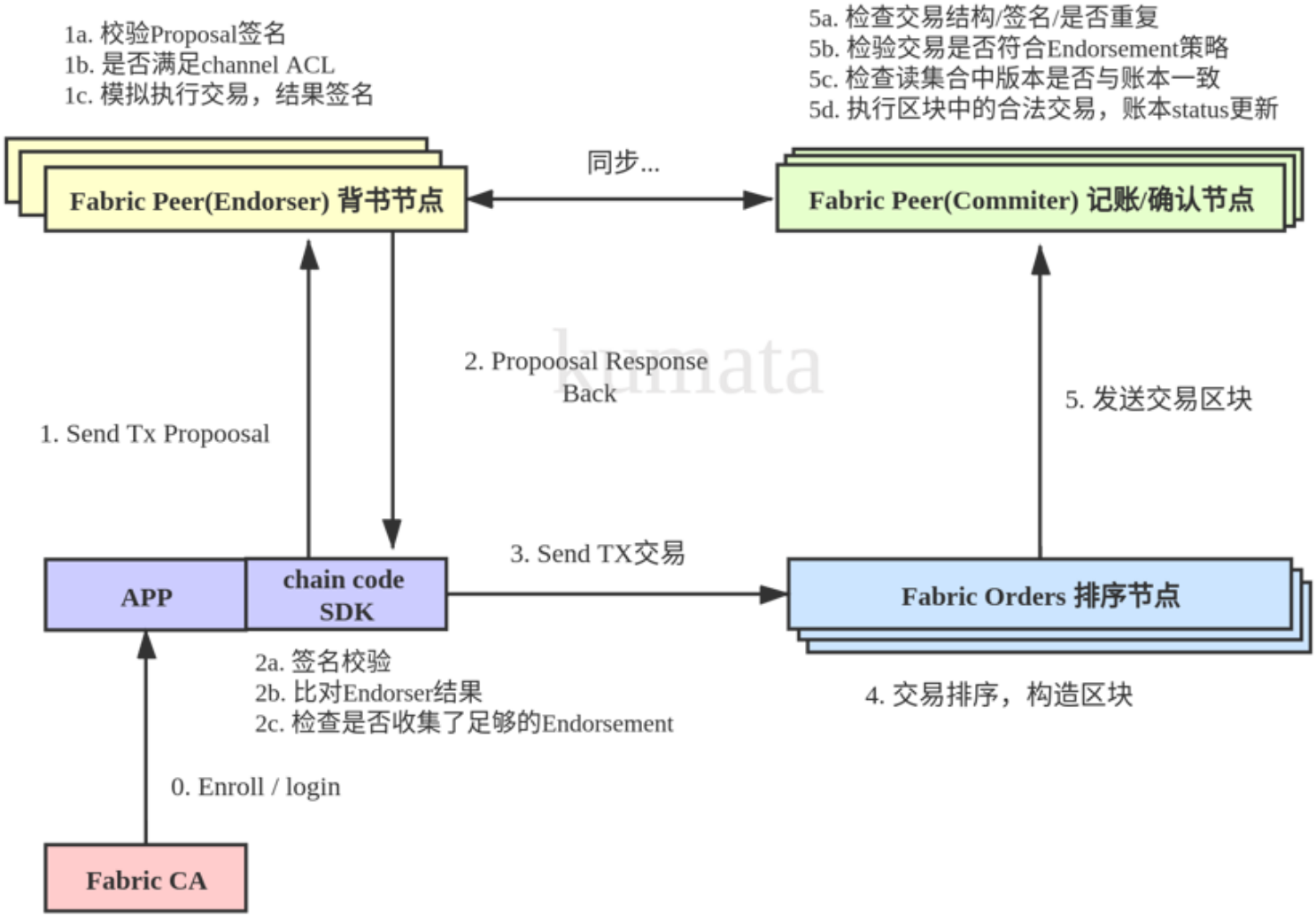
李超 (li.chao@bjtu.edu.cn)
段莉 (duanli@bjtu.edu.cn)

Fabric网络拓扑图



- Fabric网络组件：
  - 账本
  - Peer节点
  - 通道
  - 智能合约
  - 排序节点
  - Fabric证书颁发机构

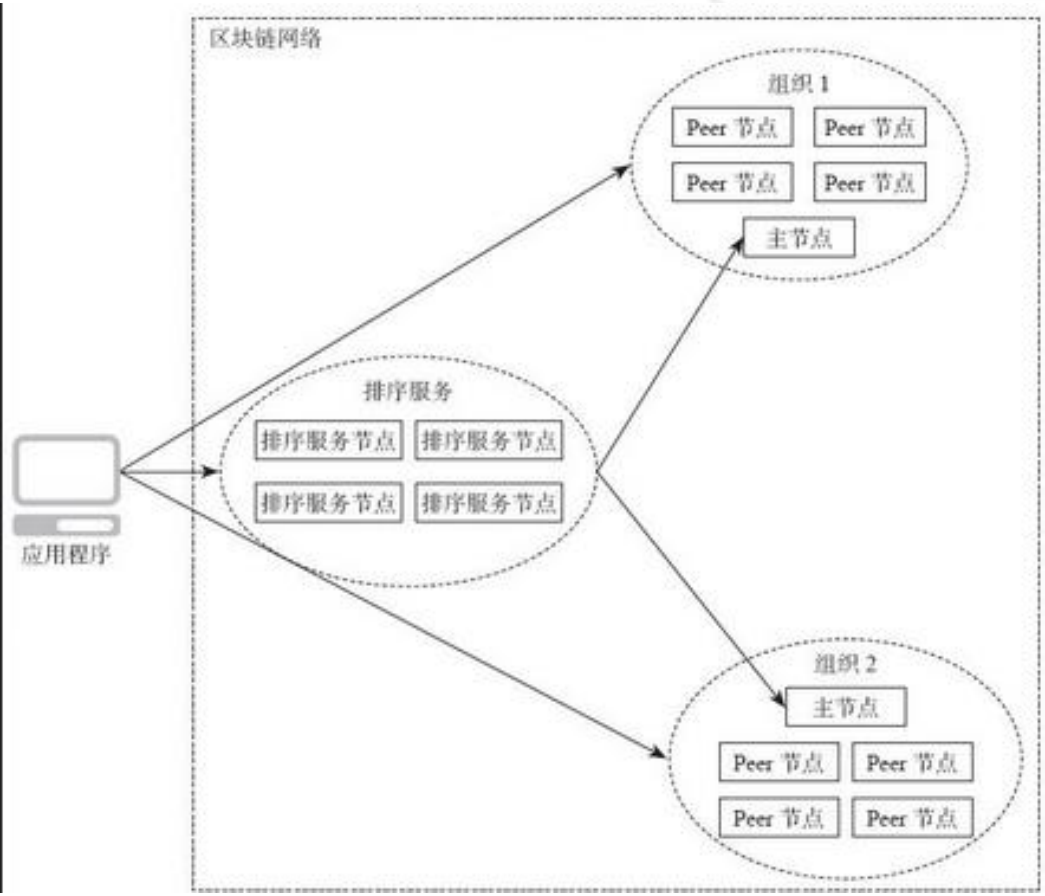
Fabric交易流程


1a. 校验Proposal签名
1b. 是否满足channel ACL
1c. 模拟执行交易，结果签名
5a. 检查交易结构/签名/是否重复
5b. 检验交易是否符合Endorsement策略
5c. 检查读集合中版本是否与账本一致
5d. 执行区块中的合法交易，账本status更新
Fabric Peer(Endorser) 背书节点
同步...
Fabric Peer(Commiter) 记账/确认节点
2. Propoosal Response Back
1. Send Tx Propoosal
5. 发送交易区块
APP
chain code SDK
3. Send TX交易
Fabric Orders 排序节点
2a. 签名校验
2b. 比对Endorser结果
2c. 检查是否收集了足够的Endorsement
4. 交易排序，构造区块
0. Enroll / login
Fabric CA

内容

- Gossip 数据传输协议
- Fabric共识机制
- Fabric智能合约

Gossip 数据传输协议



- 超级账本的Peer节点组成了一个P2P的网络;
- 客户端SDK会提交请求给Peer节点，Peer节点处理后会提交交易提案给背书节点（Endorser），然后进行背书签名（Endorsement），最后经过排序服务达成共识后广播给Peer节点;
- Gossip模块负责连接排序服务和Peer节点上，实现从单个源节点到所有节点高效的数据分发，在后台实现不同节点间的状态同步，并且可以 处理拜占庭问题、动态的节点增加和网络分区。账本信息、状态信息、成员信息等都会通过Gossip协议进行分发。

共识算法

- 根据错误类型的不同，共识算法可以满足两种范围的容错：
  - 崩溃故障容错（Crash Fault-Tolerance，CFT）
  - 拜占庭容错（Byzantine Fault-Tolerance，BFT）

共识算法

核心功能。

- 交易有效：能够根据背书及共识策略确保区块中所有交易有效。
- 交易有序：能够确保所有节点提交和执行交易顺序的一致性，这才能保证执行结果的一致性和最终全局状态的一致性。
- 交易验证：能够利用智能合约的接口，验证交易的有效性和提交顺序。

共识算法特征

- 节点一致性在账本上的体现
  - 区块的确定性：在不同节点上相同区块号的区块内容完全一致；
  - 区块的完整性：在不同节点上有相同的区块数，且按照顺序形成相同的区块链
- 共识算法特征
  - 安全性（Safety）
  - 存活性（Liveness）

共识算法类型

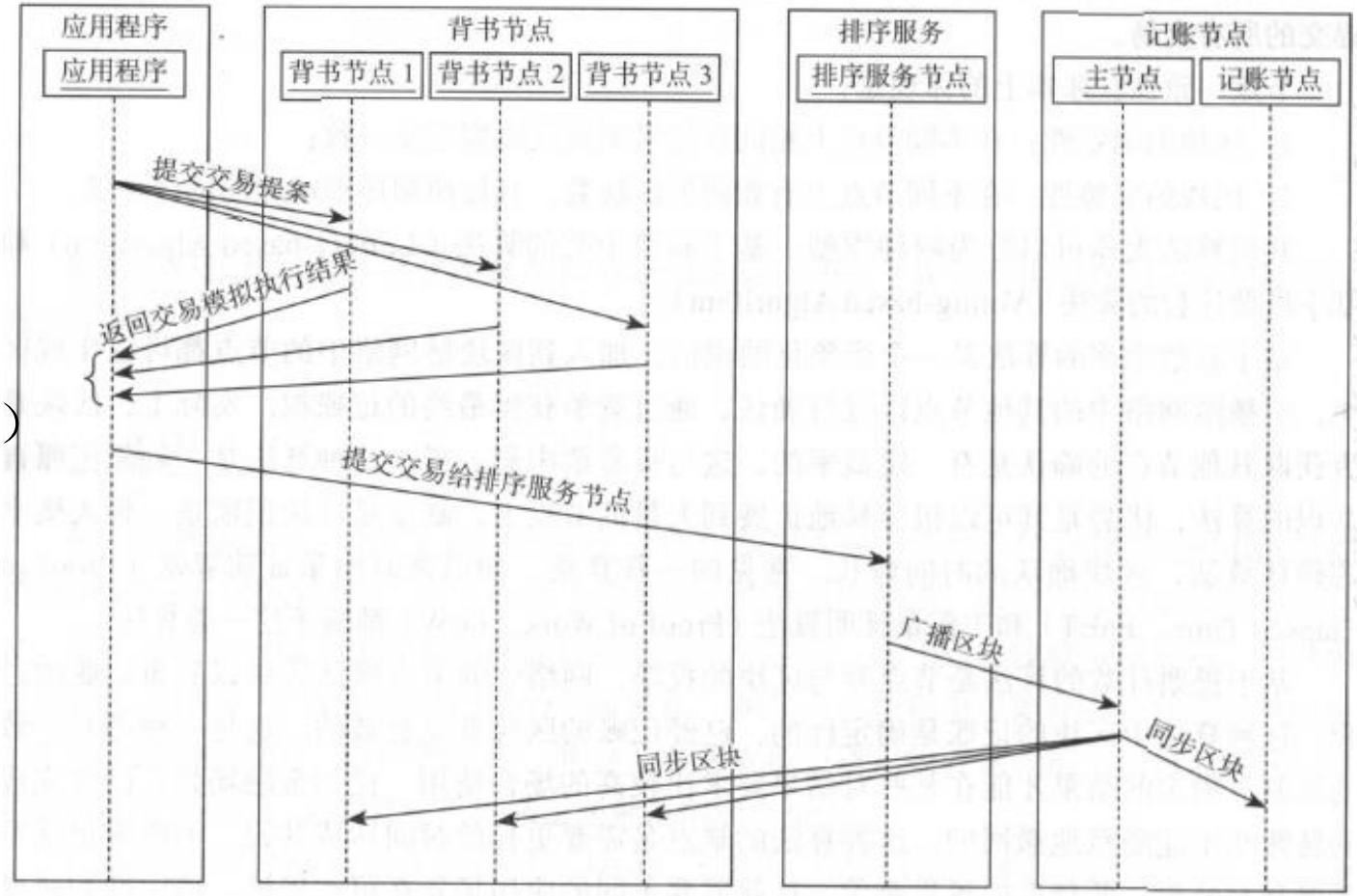
- 基于彩票中奖的算法（Lottery-based Algorithm）：PoW
- 基于投票计数的算法（Voting-based Algorithm）：RBFT 、 Paxos

表 6-1 共识算法类型的比较

	基于彩票中奖的算法	基于投票计数的算法
节点可扩展性	好	一般
交易确定性	大概率一致的共识机制	绝对一致的共识机制
交易完成速度	快	慢
共识网络流量	少	多
共识算法特点	先记账再共识	先共识再记账
典型算法示例	PoET,Pow	RBFT,Paxos

超级账本的共识机制

- 交易背书
  （模拟 @Endorser）
- 交易排序
  （排序 @Orderer）
- 交易验证
  （验证 @Committer）



Orderer

- 交易排序
- 区块分发
- 多通道数据隔离

交易排序

- 目的：保证系统交易顺序的一致性（有限状态机）
- solo：单节点排序，所见即所得
- kafka：外置消息队列保证一致性

区块分发

- 中间状态区块（非落盘区块）
- 有效交易 & 无效交易

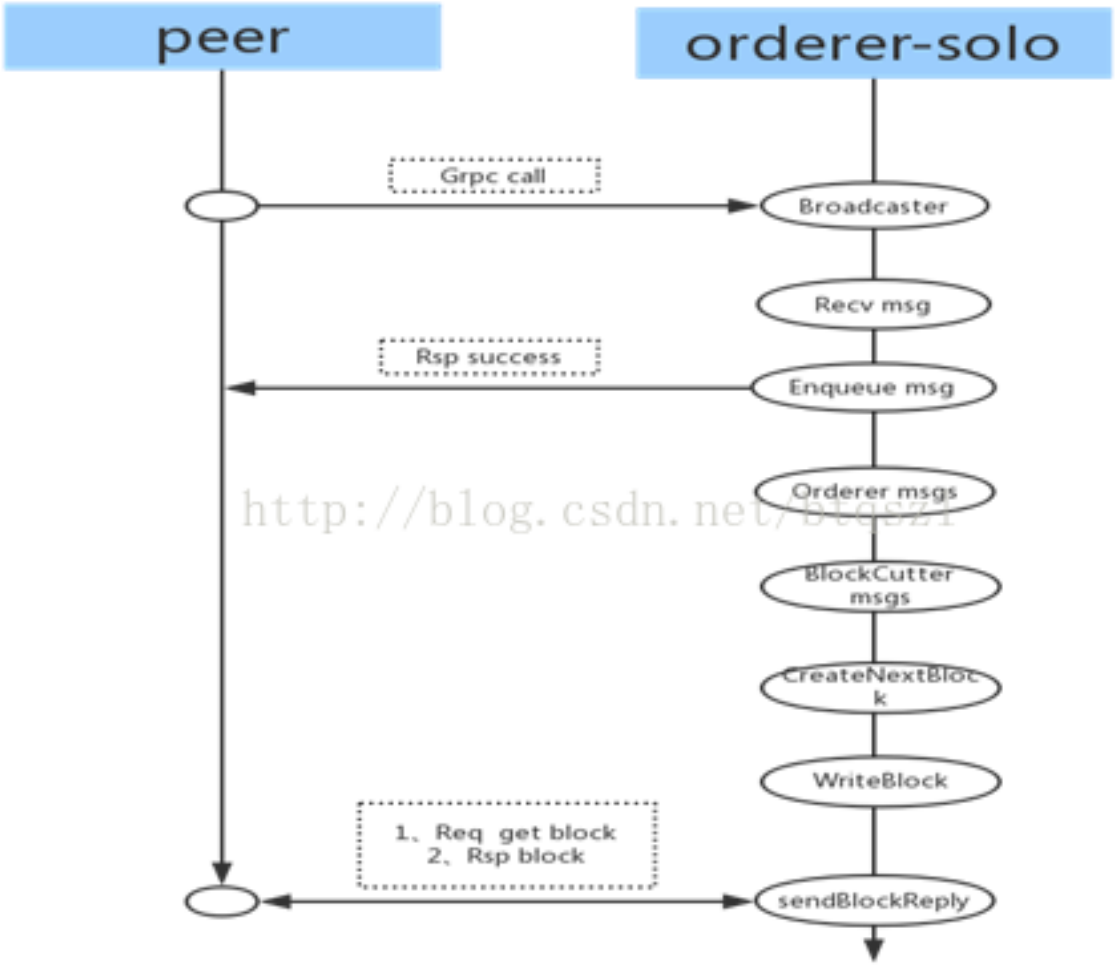
Fabric支持的排序服务

- 基于单进程solo的排序服务
- 基于Kafka的排序服务
- 基于*BFT的排序服务（×）

solo机制

Solo共识模式调用过程：

- Peer（客户端）通过GRPC发起通信，与Orderer连接成功之后，便可以向Orderer发送消息；
- Orderer通过Recv接口接收Peer发送过来的消息；
- Orderer将接收到的消息生成数据块，将数据块存入ledger
- peer通过deliver接口从orderer中的ledger获取数据块。



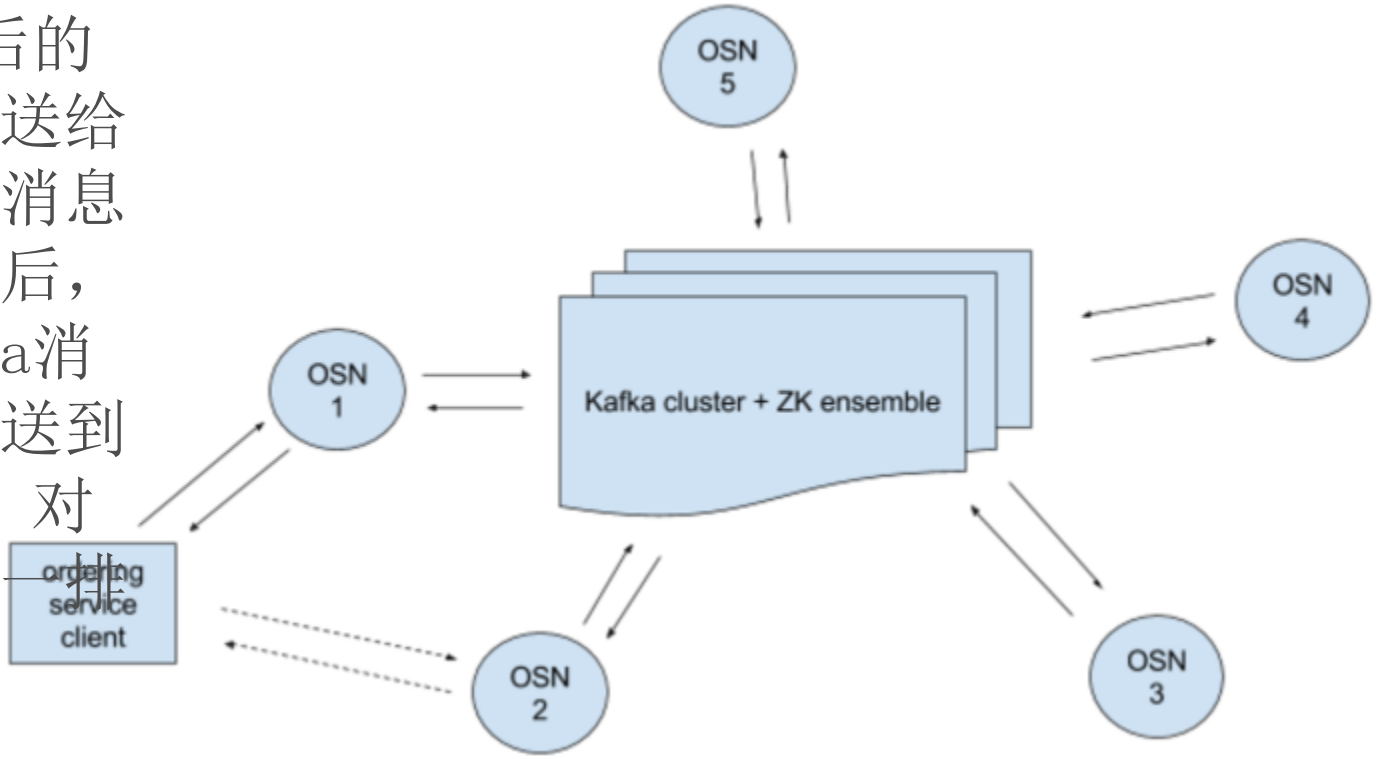
Kafka排序服务原理

工作原理

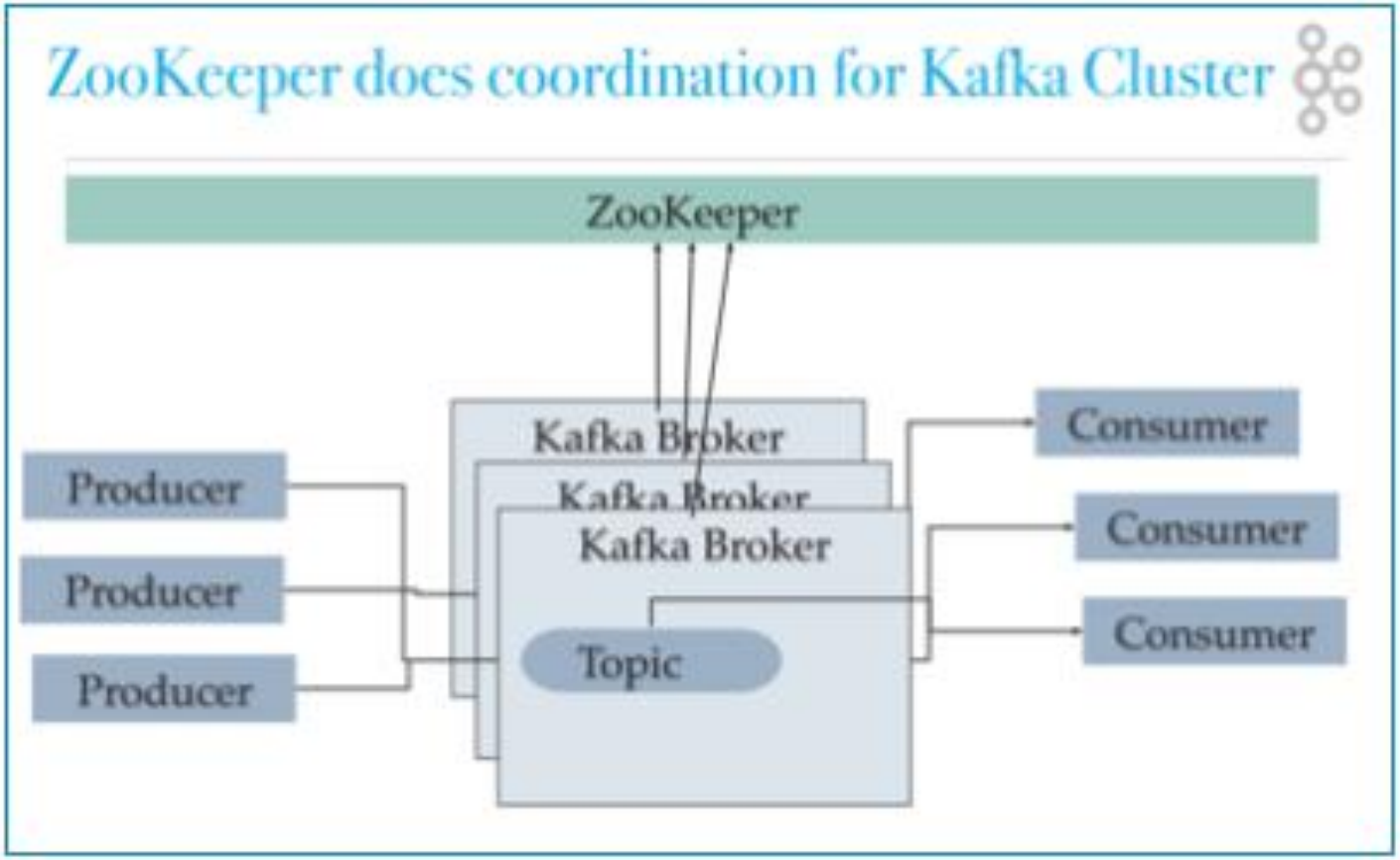
- Orderer 节点（Ordering Service Node，OSN）在网络中起到代理作用，多个 Orderer 节点会连接到 Kafka 集群，利用 Kafka 的共识功能，完成对网络中交易的排序和打包成区块的工作。
- Fabric 网络提供了多通道特性，为了支持这一特性，同时保障每个 Orderer 节点上数据的一致性，排序服务进行了一些特殊设计。
- 对于每个通道，Orderer 将其映射到 Kafka 集群中的一个 topic （topic 名称与 channelID 相同）上。由于 Orderer 目前并没有使用 Kafka Topic 的多分区负载均衡特性，默认每个 topic 只创建了一个分区（0 号分区）。

基于Kafka的共识机制

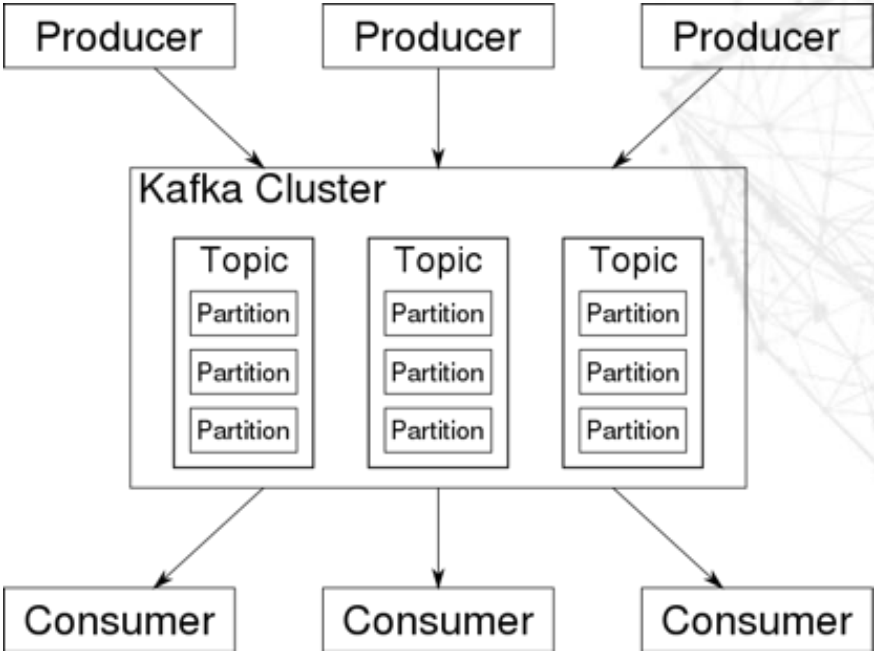
- 客户端APP通过SDK将验证后的交易信息发送给OSN，OSN对消息做初步校验后，封装成Kafka消息格式，发送到Kafka集群，对交易信息统一排序。

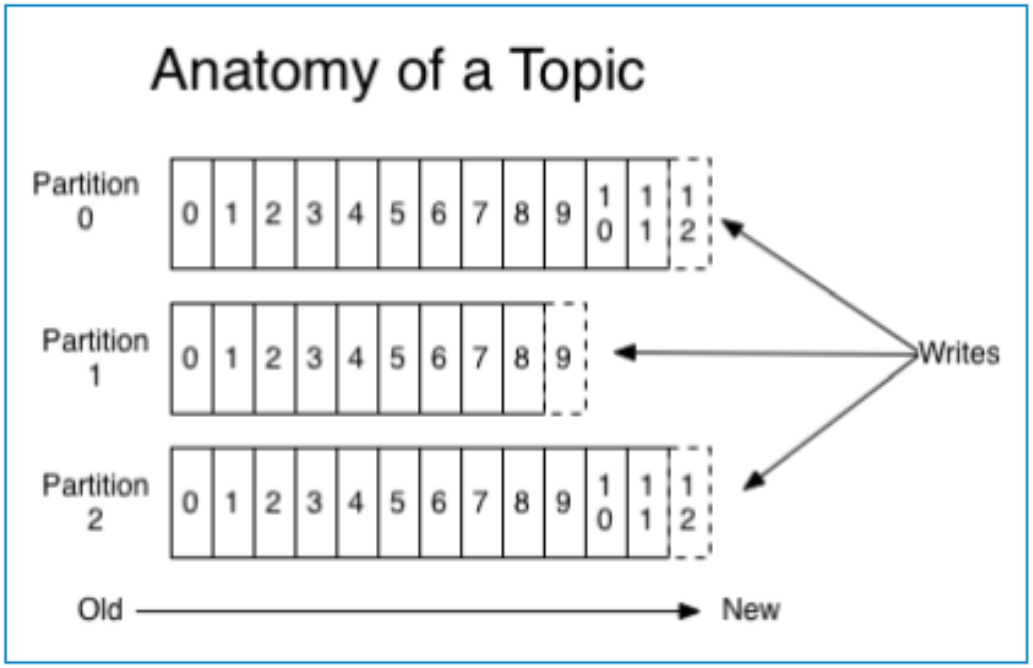


Kafka

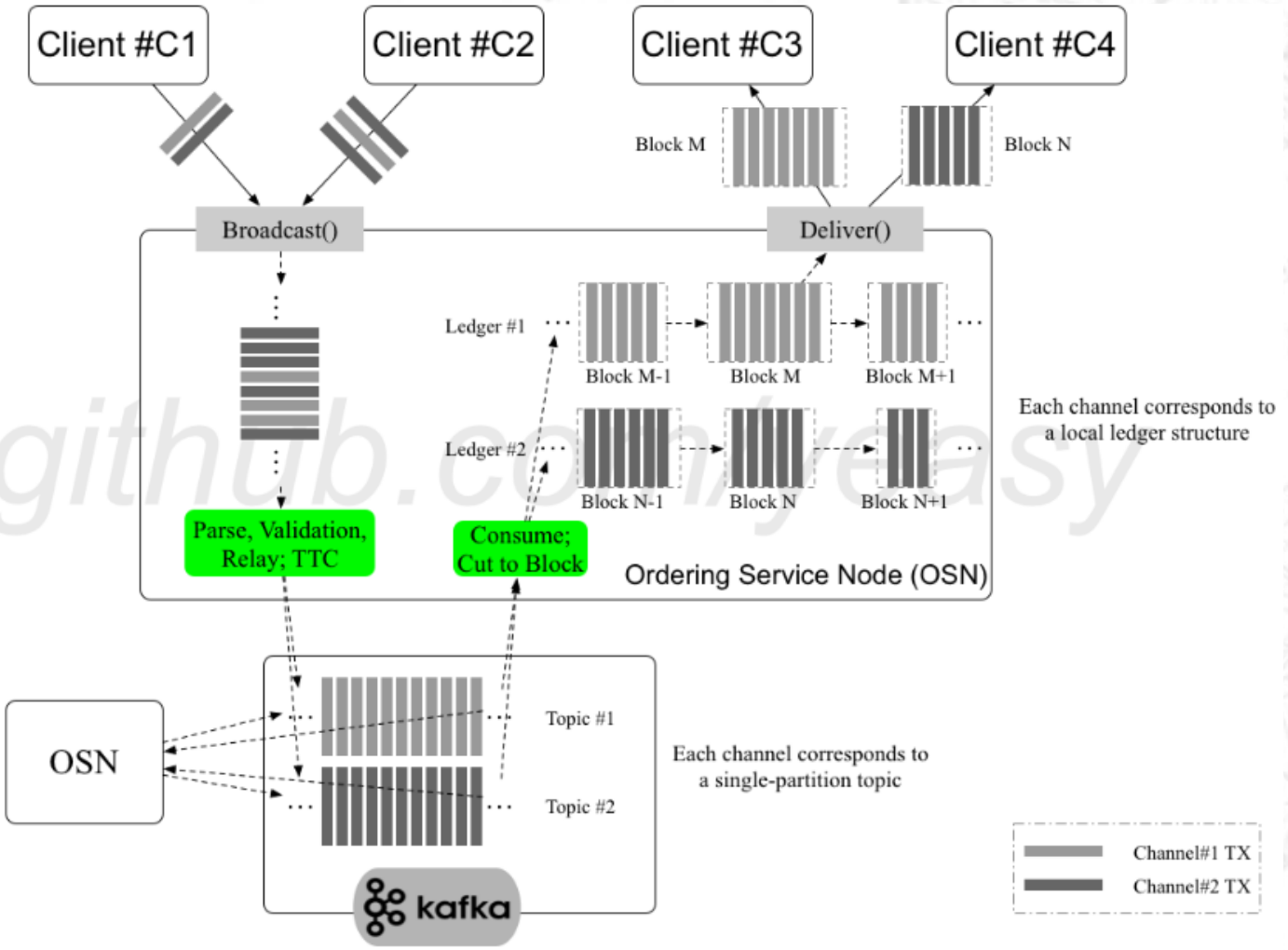

ZooKeeper does coordination for Kafka Cluster
ZooKeeper
Kafka Broker
Kafka Broker
Kafka Broker
Topic
Producer
Producer
Producer
Consumer
Consumer
Consumer

基于Kafka的共识机制





核心过程



多通道

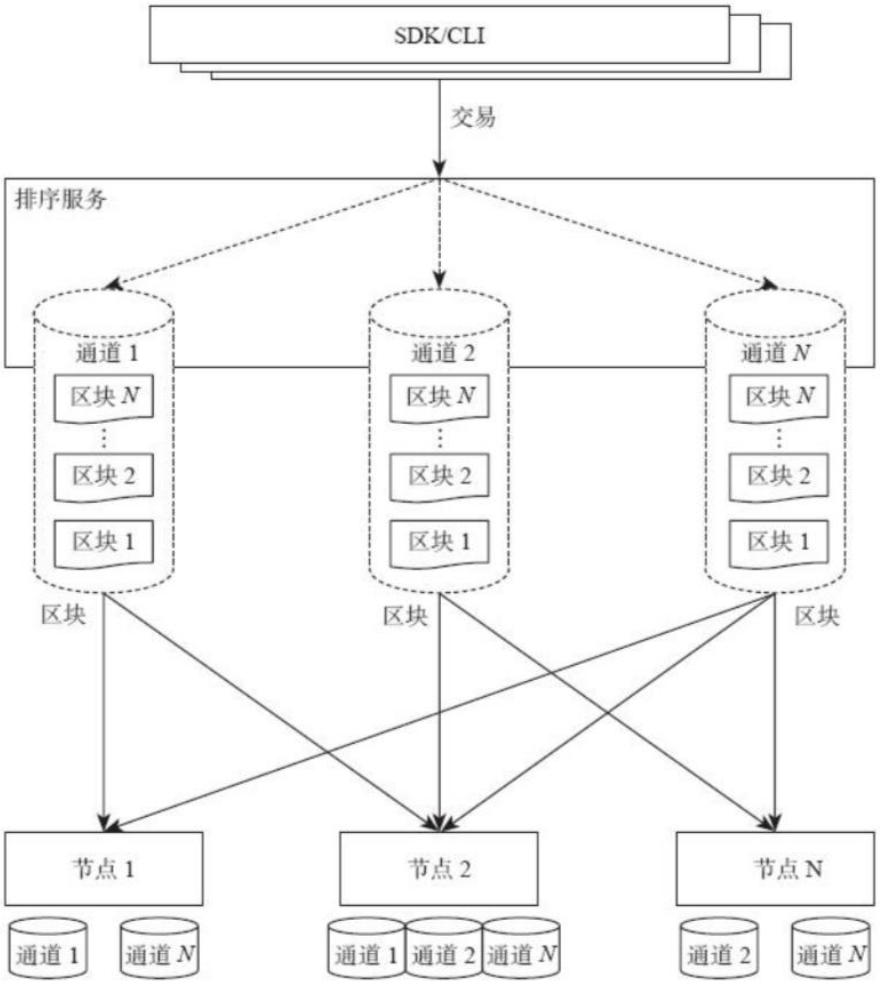
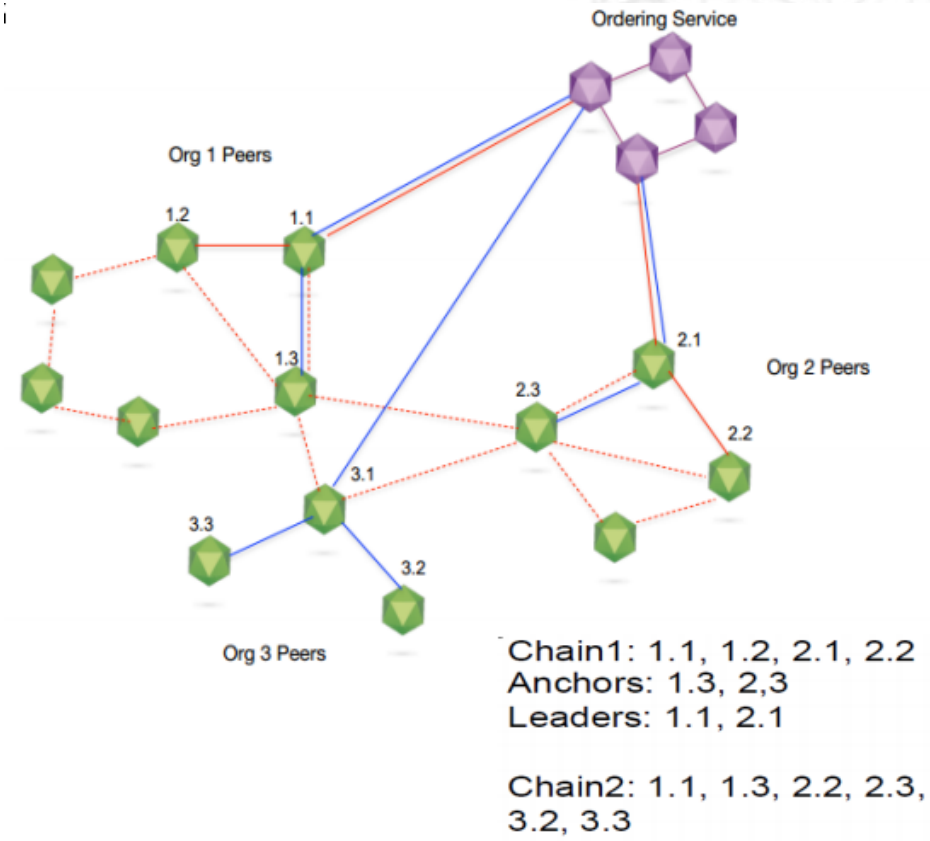


图6-2 多通道示例

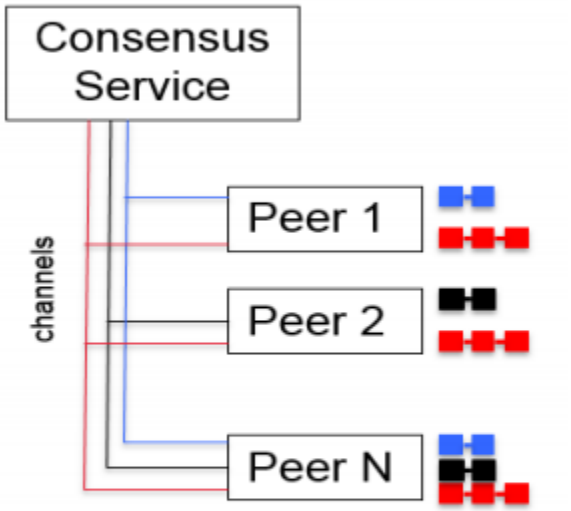
Fabric多链及多通道

- 多链chain



- 链=Peers + Ledger + Ordering Channel
- 链将参与者和数据（包含chaincode）进行隔离
- 一个Peer节点可以参与多个链

多通道



通道1（P1、 P2、P3）
通道2（P2、PN）
通道3（P1、P3）

- **通道：** 通道提供一种通讯机制，将peer和orderer连接在一起，形成一个个具有保密性的通讯链路（虚拟）
- Fabric的区块链网络缺省包含一个账本（称为：系统账本）和一个通道；
- 子账本可以被创建，并绑定到一个通道

应用多通道

如果当前网络开发测试模式，请先关闭：

```
$ cd ~/hyfa/fabric-samples/chaincode-docker-devmode

$ sudo docker-compose -f docker-compose-simple.yaml down
```

然后进入 `fabric-samples/first-network` 目录中：

```
$ cd ../first-network
```

创建一个应用通道的配置交易

由在要对一个网络进行分割，所以为了区分不同的“子网”，我们需要给不同的“子网”指定一个标识名称，所以请务必设置$CHANNEL_NAME环境变量为一个与之前通道名称完全不相同的值（代表新创建的另外一个应用通道名称）。

```
$ export CHANNEL_NAME=mychannel2
```

指定使用 `configtx.yaml` 配置文件中的 `TwoOrgsChannel` 模板，来生成新建通道的配置交易文件，

```
$ sudo ../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel2.tx -channelID
```

输出如下

```
[common/tools/configtxgen] main -> INFO 001 Loading configuration
[common/tools/configtxgen] doOutputChannelCreateTx -> INFO 002 Generating new channel configtx
[common/tools/configtxgen/encoder] NewApplicationGroup -> WARN 003 Default policy emission is deprecated, please inc
[msp] getMspConfig -> INFO 004 Loading NodeOUs
[common/tools/configtxgen/encoder] NewApplicationOrgGroup -> WARN 005 Default policy emission is deprecated, please
[msp] getMspConfig -> INFO 006 Loading NodeOUs
[common/tools/configtxgen/encoder] NewApplicationOrgGroup -> WARN 007 Default policy emission is deprecated, please
[common/tools/configtxgen] doOutputChannelCreateTx -> INFO 008 Writing new channel tx
```

生成锚节点配置更新文件

- 锚节点配置更新文件用来对组织的锚节点进行配置

同样基于 `configtx.yaml` 配置文件生成新建通道文件，每个组织都需要分别生成且注意指定对应的组织名称

```
$ sudo ../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors2.tx -c
```

执行完毕后查看 `channel-artifacts` 目录内容：

```
total 48
drwxr-xr-x 2 root root  4096 8月  28 16:29 ./
drwxr-xr-x 7 root root  4096 8月  28 10:41 ../
-rw-r--r-- 1 root root   348 8月  28 16:27 channel2.tx
-rw-r--r-- 1 root root   346 8月  28 10:41 channel.tx
-rw-r--r-- 1 root root 12639 8月  28 10:41 genesis.block
-rw-r--r-- 1 root root     0 8月   7 10:12 .gitkeep
-rw-r--r-- 1 root root   286 8月  28 16:28 Org1MSPanchors2.tx
-rw-r--r-- 1 root root   284 8月  28 10:41 Org1MSPanchors.tx
-rw-r--r-- 1 root root   286 8月  28 16:29 Org2MSPanchors2.tx
-rw-r--r-- 1 root root   284 8月  28 10:41 Org2MSPanchors.tx
```

如上输出内容所示，在 `channel-artifacts` 目录中新增了 `channel2.tx、Org1MSPanchors2.tx、Org2MSPanchors2.tx` 三个配置文件。

启动网络

```
$ sudo docker-compose -f docker-compose-cli.yaml up -d
```

命令执行后输出如下：

```
Creating network "net_byfn" with the default driver
Creating orderer.example.com
Creating peer1.org2.example.com
Creating peer0.org2.example.com
Creating peer1.org1.example.com
Creating peer0.org1.example.com
Creating cli
```

创建通道

进入Docker容器

```
$ sudo docker exec -it cli bash
```

检查环境变量是否正确设置

```
# echo $CHANNEL_NAME
```

设置环境变量

```
# export CHANNEL_NAME=mychannel2
```

创建通道

```
# peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/channel2.tx --tls --cafile
```

命令执行后输出如下：

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
[cli/common] readBlock -> INFO 002 Received block: 0
```

加入通道

应用通道所包含组织的成员节点可以加入通道中

```
# peer channel join -b mychannel2.block -o orderer:7050
```

命令执行后输出如下：

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
[channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
```

更新锚点

使用Org1的管理员身份更新锚节点配置

```
# peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/Org1MSPanchors2.tx --tls -
```

使用Org2的管理员身份更新锚节点配置

```
# CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.c
# CORE_PEER_ADDRESS=peer0.org2.example.com:7051
# CORE_PEER_LOCALMSPID="Org2MSP"
# CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.examp
# peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/Org2MSPanchors2.tx --tls -
```

列出所加入的通道

7.3.2.7 列出所加入的通道

```
# peer channel list
```

list命令会列出指定的Peer节点已经加入的所有应用通道的列表.

输出当前节点已加入的应用通道信息如下：

```
[channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
Channels peers has joined:
mychannel
mychannel2
```

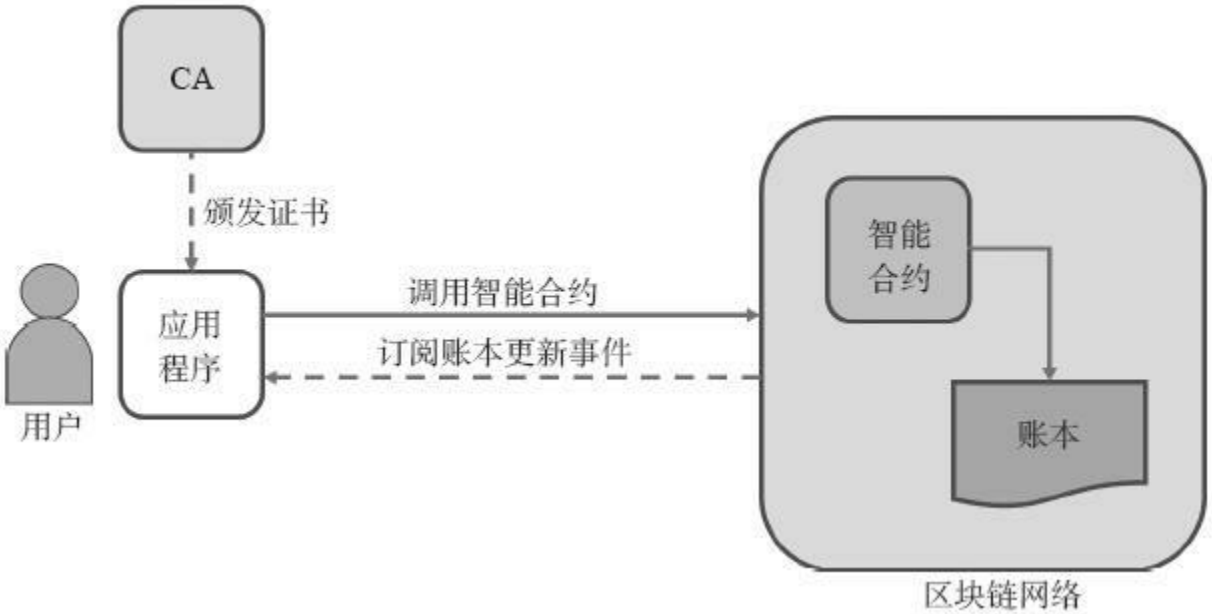
从如上终端输出中可以看到，当前 peer 节点加入了两个不同的应用通道，分别为 `mychannel、mychannel2` 。从而实现当前 peer 节点会维护两个账本。

链码ChainCode

- ChainCode与智能合约
- ChainCode的运行方式
- ChainCode的生命周期
- ChainCode的分类
- ChainCode的相互调用

ChainCode与智能合约

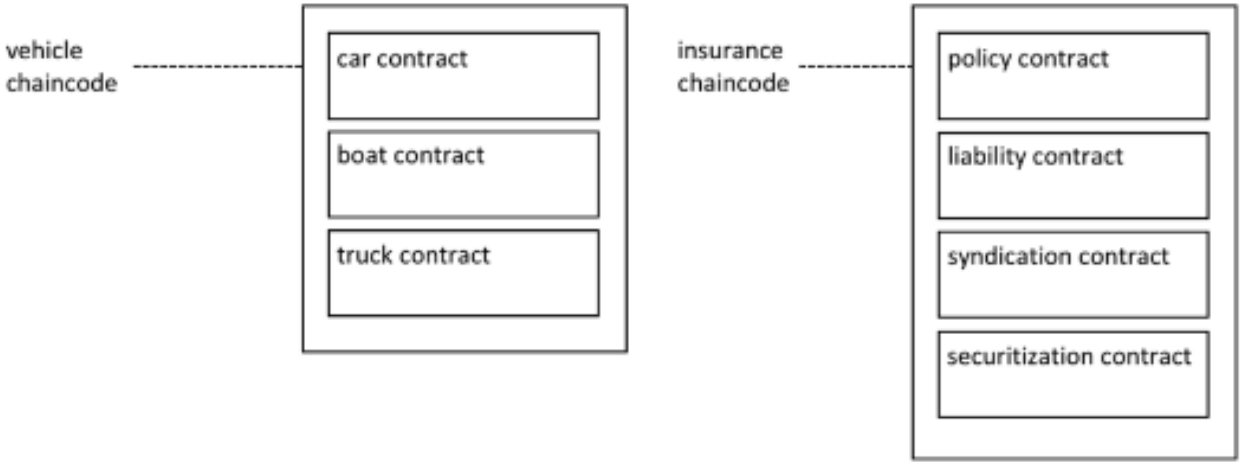
链码是**访问账本**的**基本方法**，一般是用Go等高级语言编写的、**实现规定接口的代码**。上层**应用**可以通过**调用链码**来**初始化和管理账本的状态**。只要有适当的**权限**，**链码**之间也可以**互相调用**。



链码被部署在Fabric网络节点上，运行在隔离沙盒（目前为Docker容器）中，并通过gRPC协议与相应的Peer节点进行交互，以操作分布式账本中的数据。

启动Fabric网络后，可以通过命令行或SDK进行链码操作，验证网络运行是否正常。

ChainCode与智能合约




```
async createCar(ctx, carNumber, make, model, color, owner) {

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
}
```

Seller Organization

ORG1

Buyer Organization

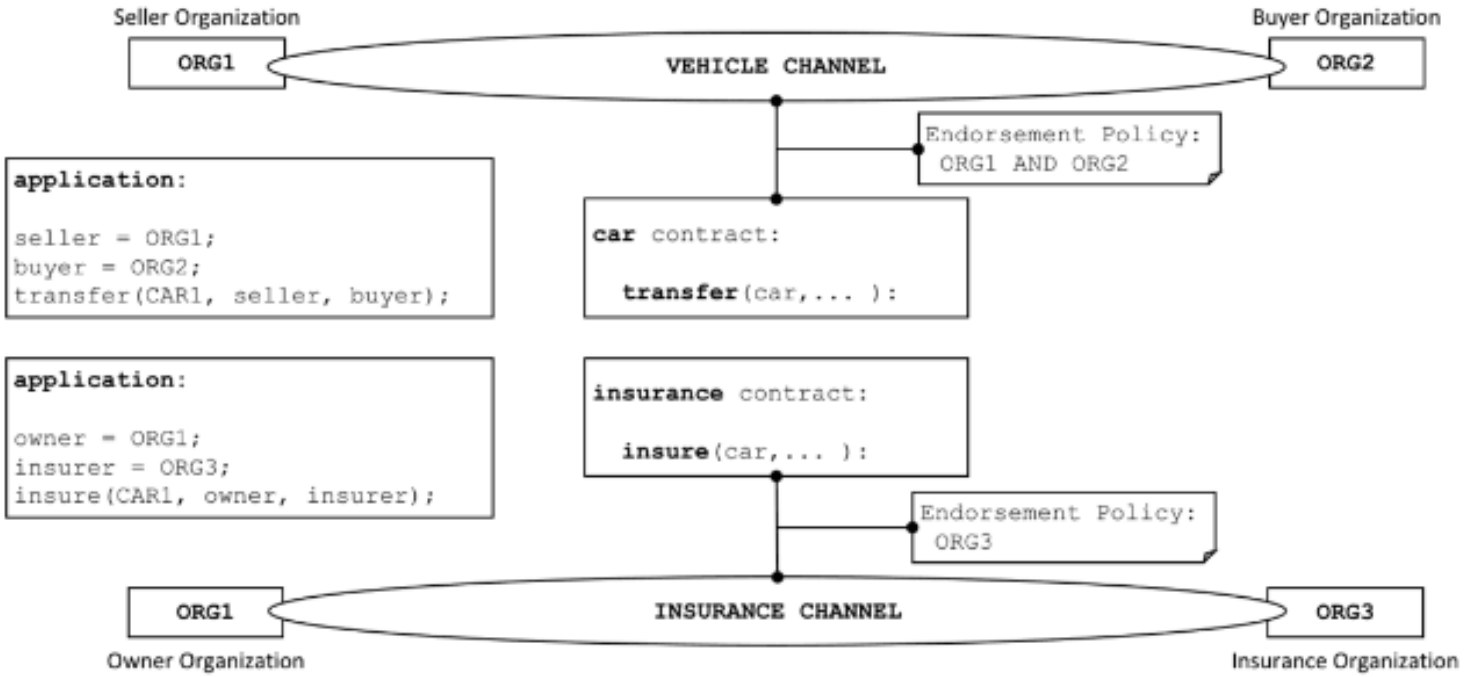
ORG2

```
application:

seller = ORG1;
buyer = ORG2;
transfer(CAR1, seller, buyer);
```

```
car contract:

  query(car):
    get(car);
    return car;

  transfer(car, buyer, seller):
    get(car);
    car.owner = buyer;
    put(car);
    return car;

  update(car, properties):
    get(car);
    car.colour = properties.colour;
    put(car);
    return car;
```

```
car interface:

  Transactions:
    query
    transfer
    update

  Endorsement Policy:
    ORG1 AND ORG2
```

Hyperledger Fabric 允许组织通过通道同时参与多个独立的区块链网络。通过加入多个通道，组织可以参与所谓网络的网络。通道可有效共享基础架构，同时保持数据和通信的隐私。它们足够独立，可以帮助组织与不同的交易对手分离其工作量，但又足够集成，可以在必要时协调独立的活动。



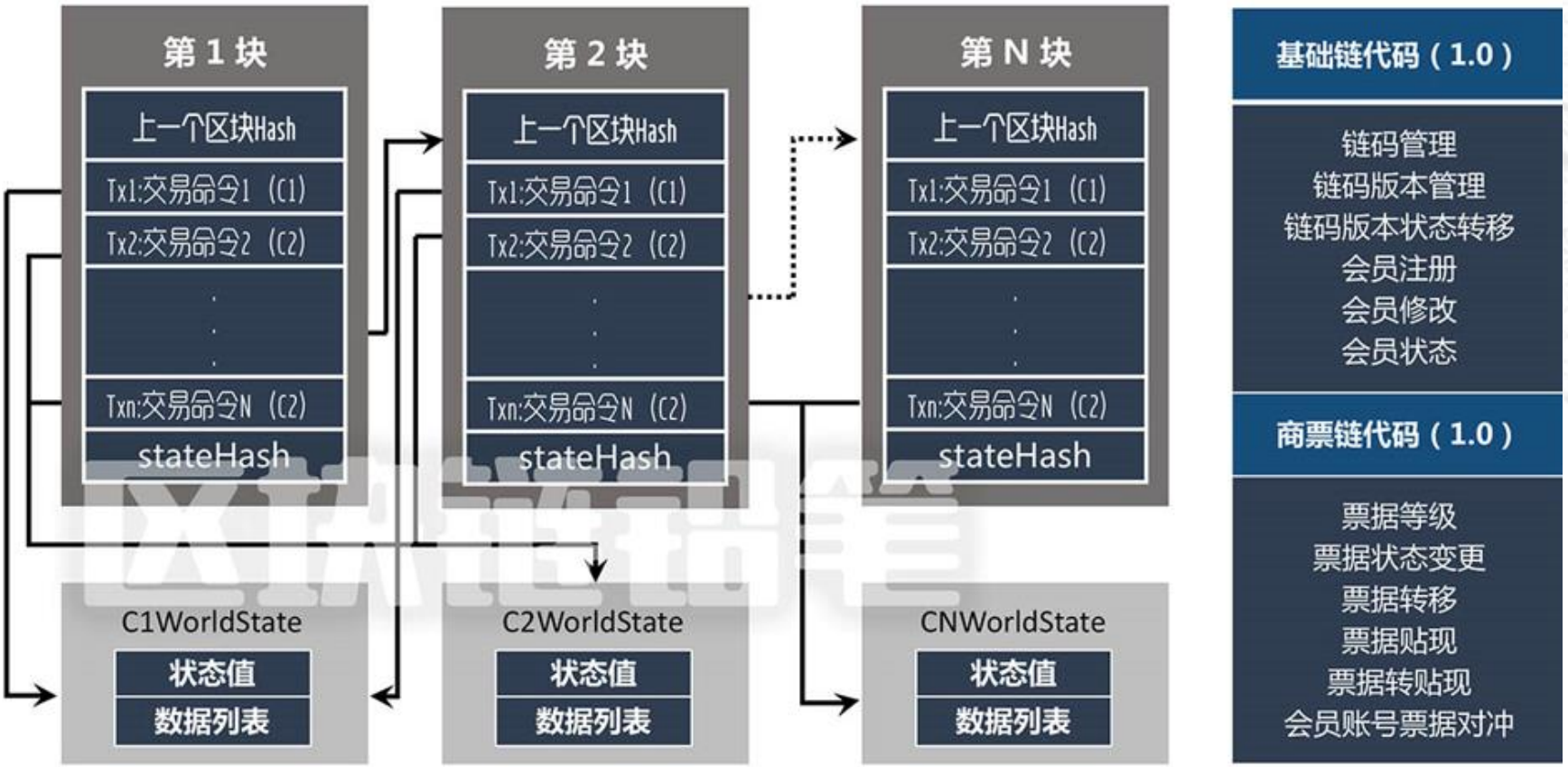
ChainCode的种类

- 在Fabric中，链码可以分为两种，系统chaincode与用户chaincode。

系统链码名称	功能介绍	是否支持链外调用
Configuration System Chaincode（CSCC）	负责账本和链的配置管理	是
Endorsement System Chaincode（ESCC）	负责背书签名过程	否
Lifecycle System Chaincode（LSCC）	负责管理用户链码的生命周期	是
Query System Chaincode（QSCC）	负责提供账本和链的信息查询功能，包括区块和交易等	是
Verification System Chaincode（VSCC）	交易提交前根据背书策略进行检查，并对读写集合的版本进行验证	否

- 用户chaincode
- 用户链码相关的代码都在core/chaincode路径下。其中core/chaincode/shim包中的代码主要是供链码容器侧调用使用，其他代码主要是Peer侧使用。

区块链票据账本数据结构



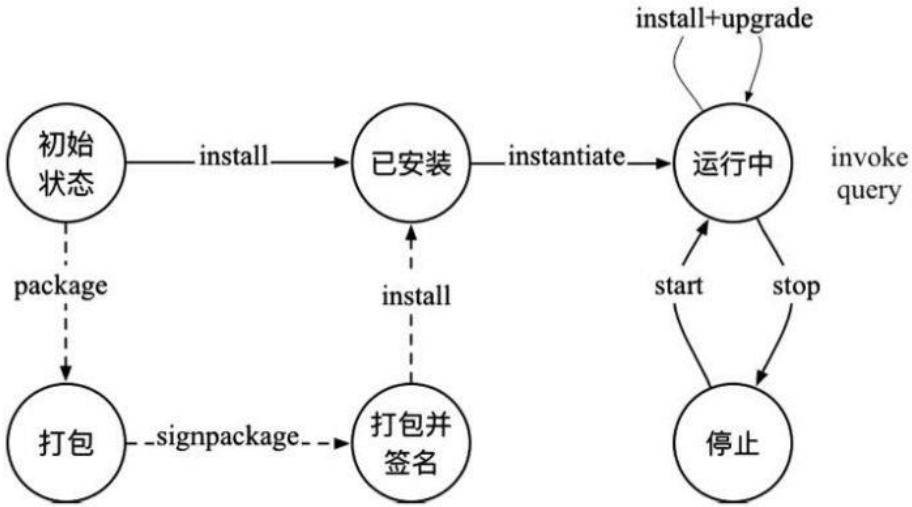
区块链数据结构示意

两者的比较

- 相同点
  - 系统chaincode与用户chaincode两种的编程模型相同，
- 不同点
  - 系统chaincode运行于peer节点内而用户chaincode运行在一个隔离的容器中。因此，系统chaincode在节点内构建且不遵循上文描述的chaincode生命周期。
  - 安装，实例化，升级这三项操作不适用于系统chaincode。

ChainCode的生命周期

链码的生命周期：
有6个状态，如下所示：Install → Instantiate → invocable → Upgrade → Deinstantiate → Uninstall.



- **install**：将已编写完成的链码安装在网络节点中；

```
peer chaincode install -n ChaincodeName -v version -p ChaincodePath
```

- **instantiate**：对已安装的链码进行实例化；

```
peer chaincode instantiate -n ChaincodeName -v version -c '{"Args":["john","0"]}'
-P "OR ('Org1.member','Org2.member')"
```

- **upgrade**：对已有链码进行升级，链代码可以在安装后根据具体需求的变化进行升级；
- **package**：对指定的链码进行打包的操作。
- **singnpackage**：对已打包的文件进行签名。

链码的相互调用

链码的相互调用分为以下两种情况：

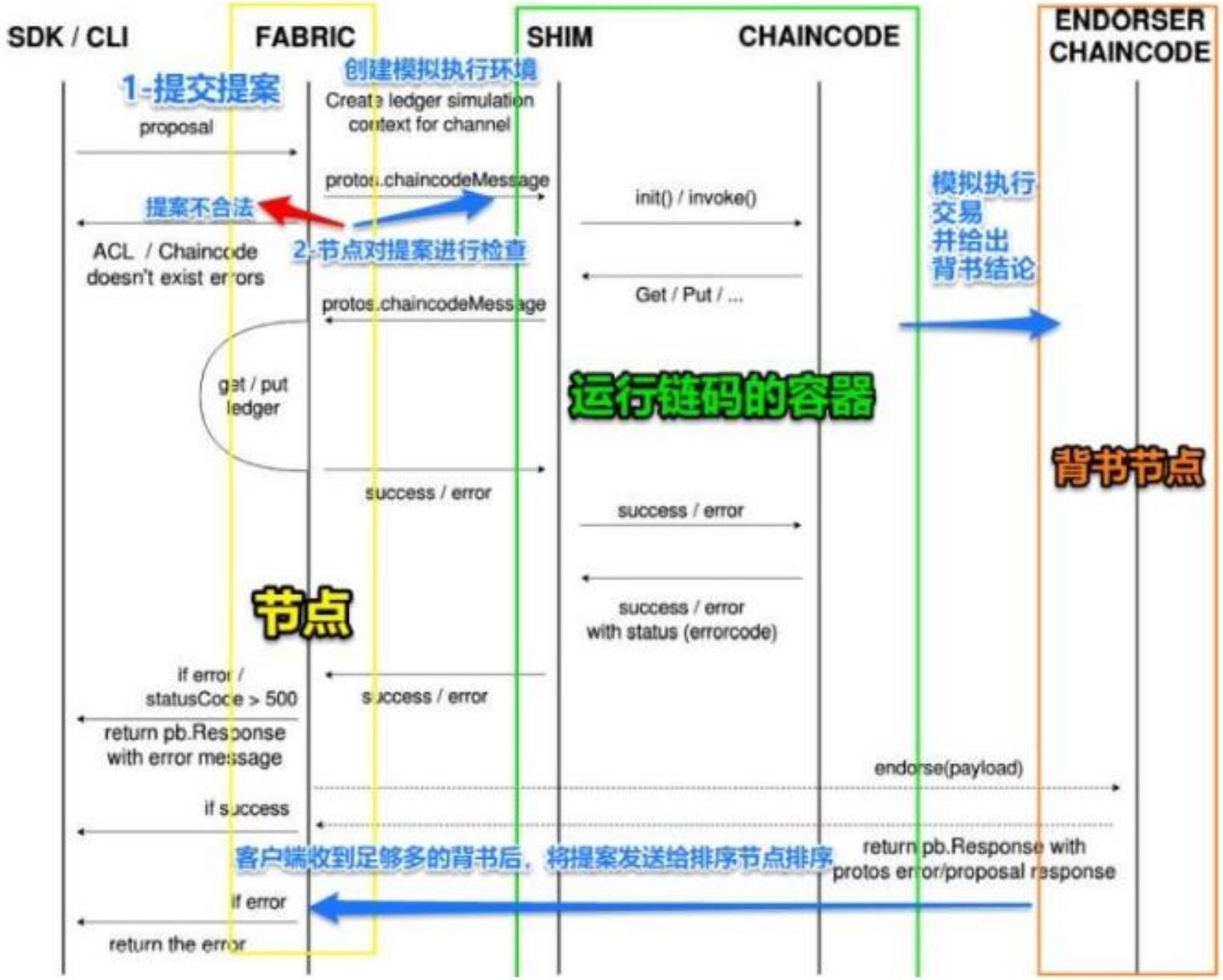
- 同一个链的链码相互调用；
- 不同链的链码相互调用。

- 调用方法通过shim.InvokeChaincode：

```
InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response
```

- 调用 shim.InvokeChaincode 后会构造一个类型为 ChaincodeMessage_INVOKE_ CHAIN CODE ChaincodeMessage 消息

链码的工作流程


SDK / CLI
FABRIC
SHIM
CHAINCODE
ENDORSER CHAINCODE
1-提交提案
proposal
创建模拟执行环境
Create ledger simulation context for channel
protos.chaincodeMessage
提案不合法
ACL / Chaincode doesn't exist errors
2-节点对提案进行检查
protos.chaincodeMessage
init() / invoke()
Get / Put / ...
模拟执行交易并给出背书结论
get / put ledger
运行链码的容器
背书节点
success / error
success / error
success / error with status (errorcode)
节点
if error / statusCode > 500
return pb.Response with error message
success / error
endorse(payload)
if success
客户端收到足够多的背书后，将提案发送给排序节点排序
return pb.Response with protos error/proposal response
if error
return the error

总结

- Gossip 数据传输协议
- Fabric共识机制
- Fabric智能合约