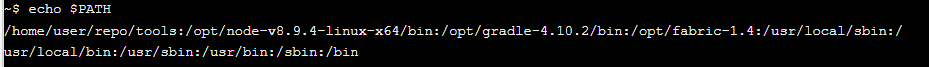
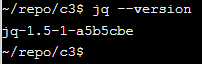
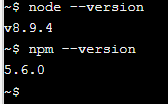
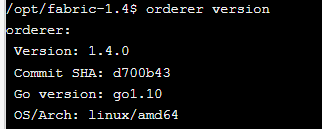
## 快速上手Fabric区块链开发

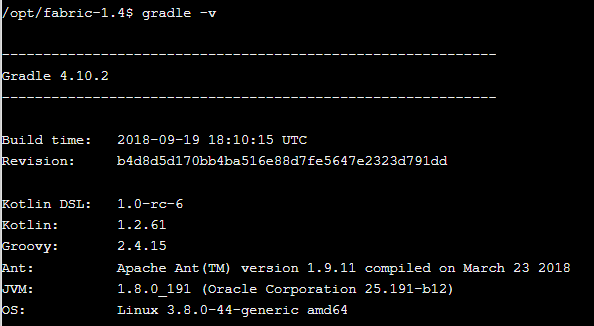
* 了解PKI和X509证书
* 使用OpenSSL建立最小CA
* 构造最简Fabric网络
* 开发计数器链码
* 开发去中心化计数器应用。

## 各种环境版本



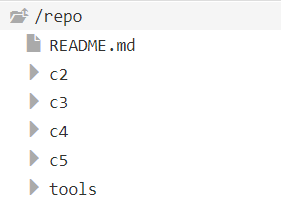
 





### 代码目录结构：

/home/user/repo



#### README.md内容

# Fabric区块链Java开发详解

本课程适合Java开发人员快速掌握超级账本Fabric区块链的设计思路、

核心概念、网络搭建、链码开发、应用开发等方面的知识和技能。

## 一、预置代码

- ~/repo/c2：第二章预置代码

- ~/repo/c3：第三章预置代码

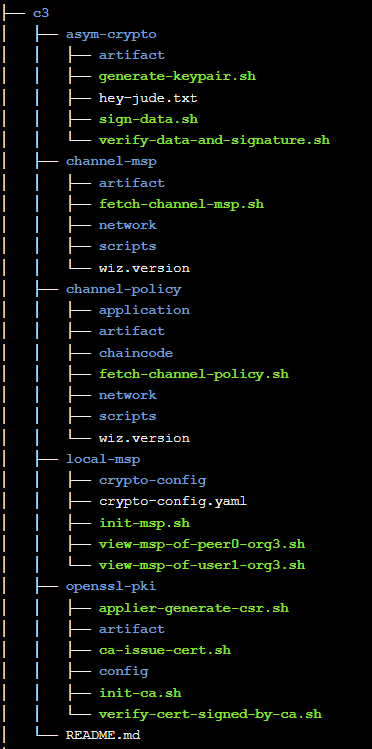
- ~/repo/c4：第四章预置代码

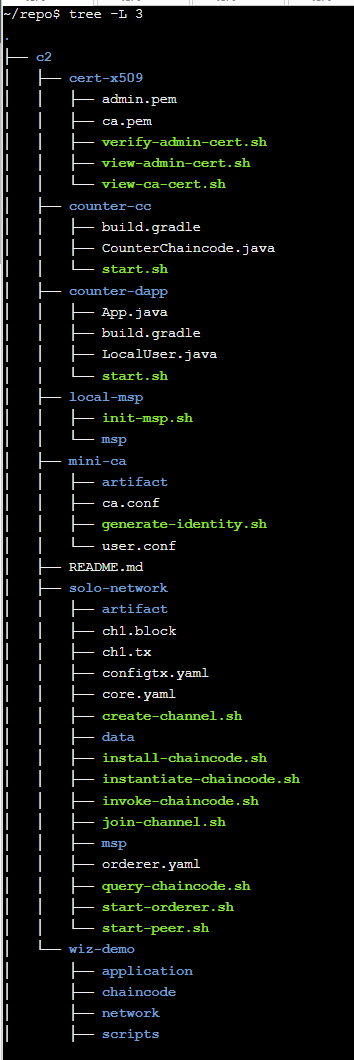
- ~/repo/c5：第五章预置代码

## 二、开发工具箱

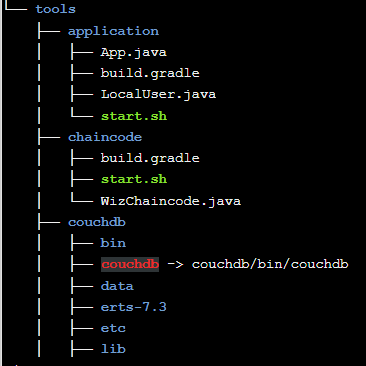
代码目录：~/repo/tools

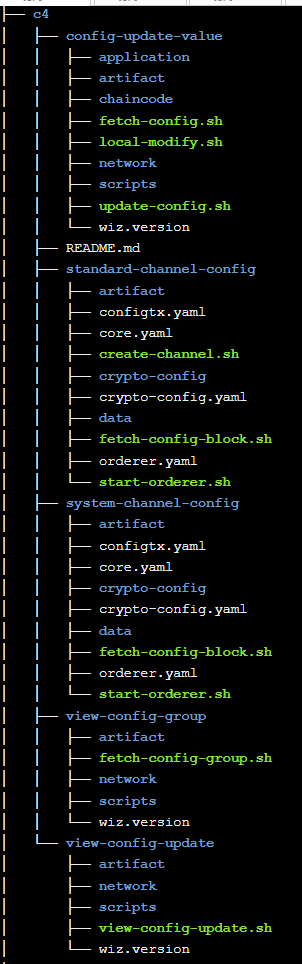
#### 详细目录结构（只截取了3级目录）

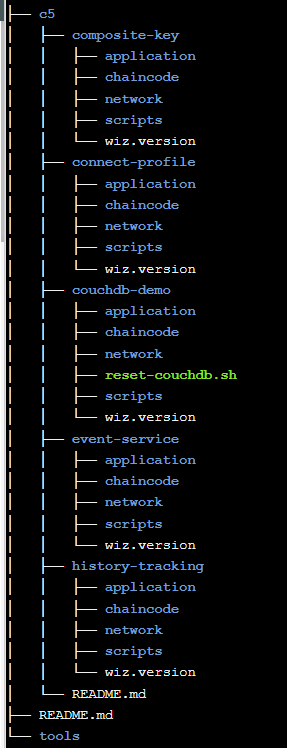




|--- wiz.version







### 使用预置代码

#### 1、查验X509证书

在1#终端进入项目目录：

~$ cd ~/repo/c2/cert-x509

查看CA根证书和Admin用户证书：

~/repo/c2/cert-x509$ ./view-ca-cert.sh

~/repo/c2/cert-x509$ ./view-admin-cert.sh

验证Admin证书是否由CA签发：

~/repo/c2/cert-x509$ ./verify-admin-cert.sh

#### 2、发行X509证书

使用openssl建立一个CA，并发行用于Fabric网络的X509V3证书。

在终端1#执行如下命令：

~$ cd ~/repo/c2/mini-ca

~/repo/c2/mini-ca$ ./generate-identity

查看artifact目录下生成的证书：

~/repo/c2/mini-ca$ ls artifact/\*-cert.pem

#### 3、组织本地MSP目录

在1#终端进入项目目录，执行脚本初始化本地MSP目录：

~$ cd ~/repo/c2/local-msp

~/repo/c2/local-msp$ ./init-msp.sh

查看本地msp目录的结构和内容：

~/repo/c2/local-msp$ tree msp

#### 4、启动最小Fabric网络

##### 4.1 启动排序节点

在1#终端进入项目目录,启动Fabric排序节点：

~$ cd ~/repo/c2/solo-network

~/repo/c2/solo-network$ ./start-orderer.sh

##### 4.2 启动对等节点

在2#终端进入项目目录，启动Fabric对等节点：

~$ cd ~/repo/c2/solo-network

~/repo/c2/solo-network$ ./start-peer.sh

##### 4.3 创建通道

在3#终端进入项目目录，创建通道ch1：

~$ cd ~/repo/c2/solo-network

~/repo/c2/solo-network$ ./create-channel.sh

然后将当前对等节点加入通道：

~/repo/c2/solo-network$ ./join-channel.sh

##### 4.4 启动Fabric链码

在4#终端进入项目目录，启动链码：

~$ cd ~/repo/c2/counter-cc

~/repo/c2/counter-cc$ ./start.sh

##### 4.5 在Fabric通道安装并激活链码

在3#终端执行如下命令安装并激活counter-cc链码：

~/repo/c2/solo-network$ ./install-chaincode.sh

~/repo/c2/solo-network$ ./instantiate-chaincode.sh

##### 4.6 测试链码调用

在3#终端的网管环境内执行如下命令查询链码：

~/repo/c2/solo-network$ ./query-chaincode.sh

使用如下命令提交计数器增长交易：

~/repo/c2/solo-network$ ./invoke-chaincode.sh

#### 5、运行Fabric应用

在5#终端进入项目目录，启动应用并观察其输出：

~$ cd ~/repo/ch2/counter-dapp

~/repo/ch2/counter-dapp$ ./start.sh

#### 6、使用WIZ工具箱

在1#终端进入项目目录，启动网络：

~/repo/c2/wiz-demo$ wiz net reset && wiz net start

在2#终端进入项目目录，启动链码：

~/repo/c2/wiz-demo$ wiz cc start

在3#终端进入项目目录，启动通道环境，然后进入管理终端查看已激活链码

~/repo/c2/wiz-demo$ wiz ch start

~/repo/c2/wiz-demo$ wiz admin

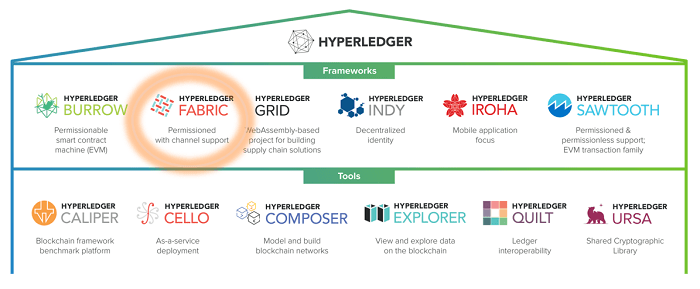
admin@org1> peer chaincode list --instantiated -C ch1

在4#终端进入项目目录，启动应用：

~/repo/c2/wiz-demo$ wiz app start

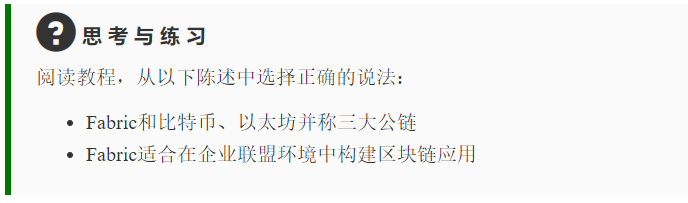
## Hyperledger及Fabric项目概述

Hyperledger 是一个旨在推动区块链跨行业应用的开源项目， 由 Linux基金会在2015年12月主导发起该项目， 成员包括金融、银行、物联网、供应链、制造和科技等多个行业的领头羊，托管了众多面向企业的区块链 开源框架和工具：



Hyperledger Fabric（后文简称Fabric）是其中发展最好的一个**企业级区块链平台**，最初由Digital Asset和IBM 贡献，目前已经应用于沃尔玛的食物溯源链（Foodtrust）和马士基的物流跟踪链（TradeLens）中， 代表了当下企业级区块链应用的最高水平。可以认为Fabric是一种联盟链（Consortium Blockchain）平台， 它适合构建跨越多个企业边界的去中心化应用。

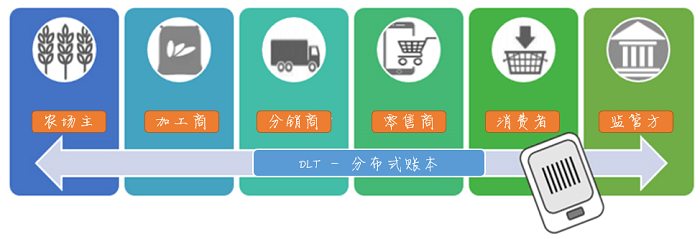
由于Fabric项目的目标是应用于相对可信的企业联盟环境，因此其设计思路与比特币、以太坊 等公链平台有明显的差异。Fabric借鉴了区块链的数据结构，但引入了相当多的身份验证与 权限控制机制，以及数据隐私保护机制，以适应企业级应用的要求。同时由于企业联盟环境 要比完全开放的公链环境可控，**因此Fabric没有强调其共识体系对拜占庭容错的实现，允许使用非拜占庭容错算法建立共识，从而可以达到相当实用的交易吞吐量。**



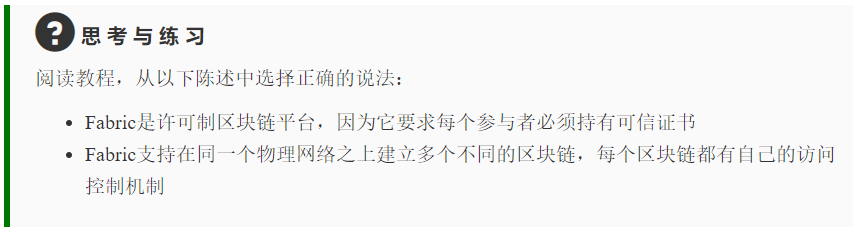
## Fabric的定位与特点

毫无疑问，Fabric是受到比特币的启发而诞生的，因此它借鉴了比特币、以太坊这些公链中的一些 核心特性，例如采用不可篡改的区块链结构来保存数据、采用非对称加密技术来进行身份识别 与认证、支持智能合约等等。

但是Fabric定位于企业级的分布式账本技术（**DLT** - **D**istributed **L**edger **T**echnology）平台， 它的主要目的是为跨越多个企业边界的活动提供不可篡改的分布式记账平台。例如在食物溯源 应用中，为了让消费者可以了解到所购买食物是否安全，就必须将从农场到加工商、分销商、 零售商乃至监管机构等各个环节的检验与放行信息记录到区块链上，以保证溯源信息的透明与可信：



因此Fabric是一种联盟链（Consortium Blockchain），它适合在多个企业间实现分布式记账，这一 定位使Fabric的实现与以太坊这样的公链有了明显的差异：



### 分布式账本 vs. 区块链

分布式账本是比区块链更加宽泛的概念，可以认为区块链只是分布式账本的一种实现技术， 其他的分布式账本实现还包括哈希图等。

### 去中心化 vs. 分布式

Fabric淡化了去中心化（Decentralized），而以分布式（Distributed）代替，这一思路带来了系统设计 与实现上的巨大影响。 例如，在Fabric中，采用中心化的CA机制来发放证书，只有持有有效证书的节点和用户才可以访问区块链 上的账本数据。因此Fabric是许可制 / Permissioned的区块链，这与不需要许可 / Permissionless 的以太坊这样的公链形成了鲜明的对比。

### 拜占庭容错 vs. 崩溃容错

由于采用许可机制，Fabric也淡化了对不可信（Trustless）环境下共识达成的依赖性，而假设 联盟链中的企业有可能是值得信赖的，因此并不依赖于工作量证明这样的拜占庭容错算法 —— 虽然 Fabric模块化的设计可以支持引入不同的共识算法实现，但目前的产品化方案是Kafka共识，它显然 是不能对抗拜占庭错误的 —— 不过对不可信环境支持的淡化处理有利于提高交易的吞吐量，这对于企 业级应用也是有益的。

### 数据隐私保护

在另一方面，Fabric强化了隐私保护能力。例如，Fabric支持在同一套企业网络上建立多个不同 的**通道/Channel**，每一个通道都有自己的区块链和访问控制，彼此互不影响，这有利于 复用基础设施，例如不同企业间的销售部门可以建立一个通道来分享市场数据，而这些企业间的 研发部门可以建立另一个通道来分享技术数据。

Fabric并不是唯一的联盟链解决方案，但目前可以说是最复杂的企业联盟链实现，这种复杂性 源于设计者对应用场景的假设和推演，以及对Fabric广泛适用性的考量，这是我们在学习过程 中需要换位思考的一点。

### 课程内容概述

本课程适合Java开发人员快速掌握超级账本Fabric区块链的设计思路、 核心概念、网络搭建、链码及应用开发等知识点，课程主要内容简述如下：

**第一章、课程概述**

简介Fabric项目来源、定位特点、适用场景以及其与公链平台的重要区别。

**第二章、初识Fabric**

创建一个最小CA，搭建一个最小Farbic网络，开发一个最小Fabric链码，实现 一个最小Fabric应用，从零开始学习并掌握Farbic中的核心概念、设计思路、 实用工具与开发模型。

**第三章、身份与权限管理机制**

深入学习Fabric中的身份验证机制与权限管理机制，理解证书、成员服务提供器、 策略、访问控制清单等核心概念并掌握其创建、部署和使用方法。

**第四章、通道配置与更新**

深入学习Fabric的通道配置初始化与更新流程、相关数据结构以及配置工具的 操作方法。

**第五章、链码与应用开发进阶**

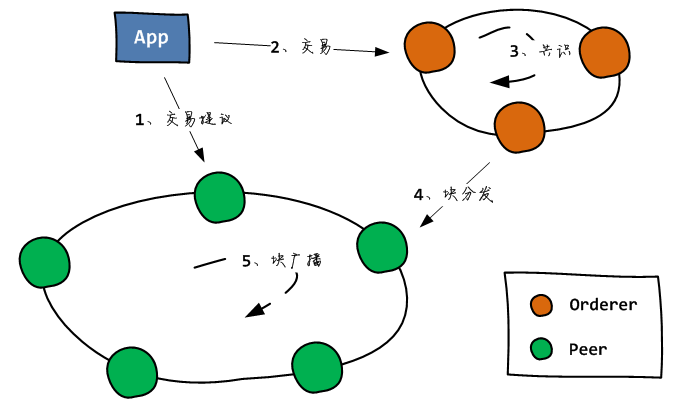
学习理解链码运行机制、掌握状态历史跟踪、富查询、通道事件监听等链码与应用 开发进阶知识。



## 初识Fabric

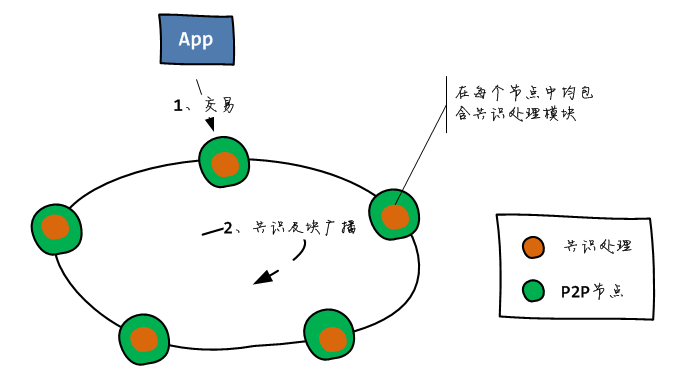
### Fabric架构概述

Fabric的架构中包含相当多的细节，为了便于理解，我们先观察最基本的两个 组成部分 - 对等节点（Peer）和排序器（Orderer）：



Fabric的设计采用分布式系统中经典的[状态机复制模型](https://en.wikipedia.org/wiki/State_machine_replication" \t "_blank)， 对等节点负责交易区块链的保存以及状态的更新，而排序器则负责**确定应用所提交的交易的先后 执行顺序**，并**将交易打包成块**后分发给**主导**对等节点，再由主导对等节点通过gossip协议扩散至其他对等节点，对等节点进行验证后执行交易从而更新其状态。

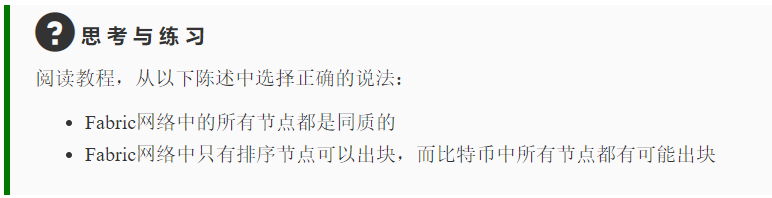
这一架构与大部分的区块链实现有显著的不同 —— 共识处理模块通常总是包含在每一个区块链的节点实现当中，例如比特币或者以太坊 —— 这使得所有的节点完全是同质的：



为什么Fabric要搞的这么复杂？

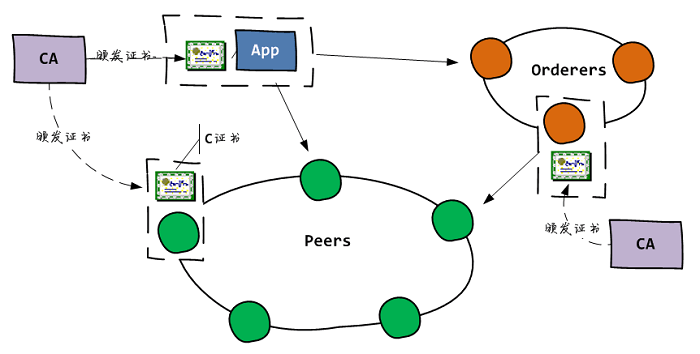
根据官方的说法，将共识处理从节点的功能中单独抽取出来作为一个可插拔的模块， 是为了能更好地利用已有的各种共识算法实现，例如崩溃容错（CFT）和拜占庭容错（BFT） 都已经有现成的实现工具包，这样就容易为Fabric添加各种共识支持。

例如，在当前的1.4版本中，Fabric提供了一个基于Kafka和Zookeeper实现的崩溃容错排序器，在接下来的发布中，还将提供一个基于etcd的**raft共识排序器**以及一个真正去中心化的**BFT排序器**。



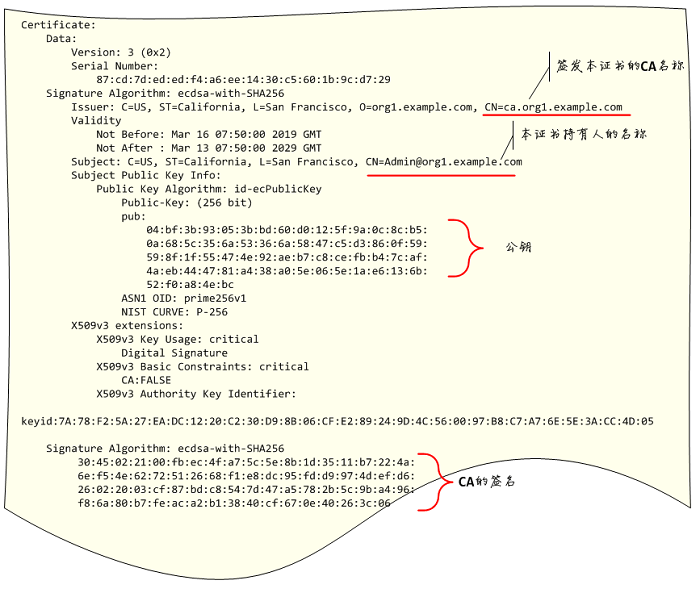
### 身份标识与验证机制

和大多数区块链一样，Fabric也采用**非对称加密技术**来标识用户的身份，**区别在于 Fabric取消了用户自己生成身份标识的权利，而采用中心化的CA来统一颁发证书作为**区块链参与各方（对等节点、排序器、应用用户等）的身份标识：



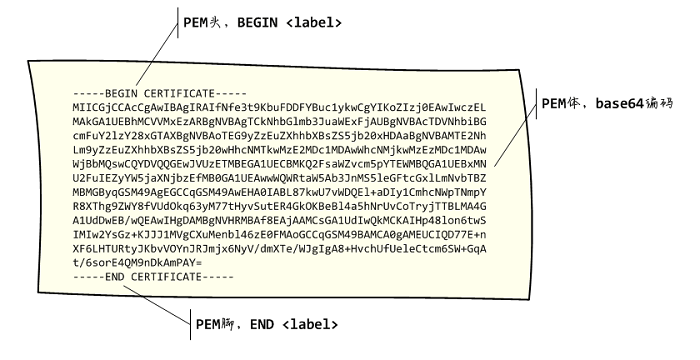
**在一个Fabric网络中可以使用单一的CA，也可以存在多个不同的CA，例如每个参与机构都由自己的CA为自己组织内的节点和用户颁发证书，并单独为排序节点建立一个CA。**

CA颁发的证书中包含了用户公钥和唯一标识名称等信息，同时也包含了CA的签名， 因此第三方可以验证用户持有的证书是否由CA签发，从而验证用户的身份。下图 展示了Fabric网络中使用的X509证书的主要内容：



在X509证书中，Issuer字段标识签发此证书的CA机构的唯一标识名，Subject字段 标识此证书持有者的唯一标识名。在唯一标识名中最常用的字段是常用名（CN：Common Name） 字段，例如上图中的ca.org1.example.com或者Admin@org1.example.com。

由于证书中包含用户公钥以及CA签名这些二进制数据，为了便于在网络环境中传输交换，通常会编码为仅包含ASCII码的 [PEM](https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail) 格式，即**Privacy-Enhanced Mail** - 隐私增强邮件：



需要指出的是，证书是公开的，任何人都可以查看验证，因此在证书中并不包含用户的私钥信息。

可以使用openssl命令来查看一个证书的内容，例如：

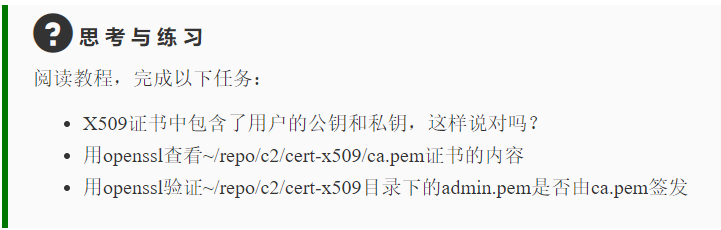
openssl x509 -in admin.pem \ *# 查看证书admin.pem的内容*

-text \ *# 输出格式为文本*

-noout *# 不显示原始PEM内容*

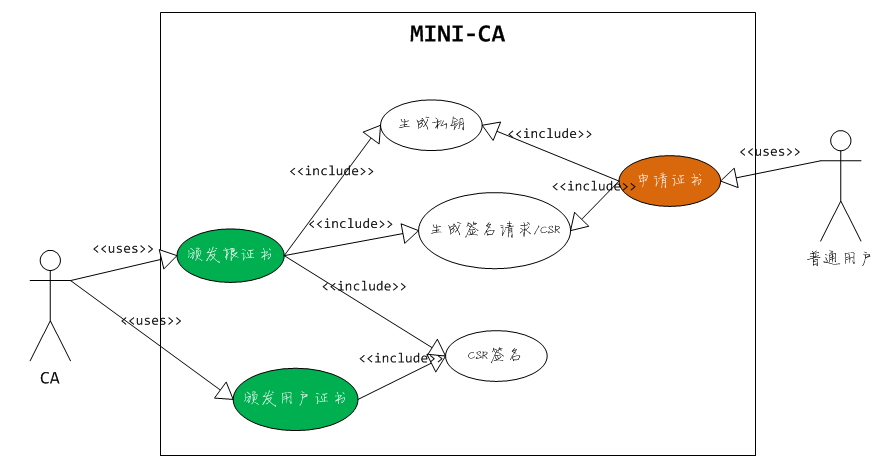
利用CA的证书可以验证一个指定的证书是否由该CA签发，例如：

**openssl** **verify** **-CAfile** **ca**.pem **admin**.pem



### 使用OpenSSL构建小型CA

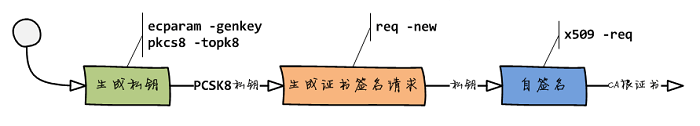
我们可以使用OpenSSL来建立一个CA，并为Peer、Orderer和用户颁发证书：



由于CA需要为自己签名，因此我们首先需要生成CA的自签名证书。

**CA自签名证书**

生成CA自签名证书主要包括三个环节：



首先使用ecparam子命令选择prime256v1椭圆曲线参数创建CA根密钥，这是Fabric中使用 的非对称加密算法：

openssl ecparam -genkey \ *# 生成椭圆曲线私钥*

-name prime256v1 \ *# 使用prime256v1椭圆曲线参数*

-out ca-key\_.pem *# 生成的EC私钥存入ca-key\_.pem文件*

然后将上述输出文件转换为不加密的PKCS8格式私钥并保存在ca-key.pem文件中， 以便后续在其他命令中利用私钥签名：

openssl pkcs8 -topk8 \ **# 将输入私钥转换为PKCS8格式**

-nocrypt \ **# 不加密输出的PKCS8私钥**

-**in** ca-key\_.pem \ **# 输入私钥为ca-key\_.pem**

-**out** ca-key.pem **# 输出存入ca-key.pem**

PKCS8的全文是Private Key Cryptography Standard #8，即私钥密码学8#标准， 是以ASCII存储密钥的互联网标准，可以点击[这里](https://tools.ietf.org/html/rfc5208) 查看规范原文。

现在利用私钥生成证书签名请求（CSR：Certificate Signing Request）文件，其中的配置文件 ca.conf中声明了CSR相关信息，例如唯一标识名称、启用的扩展等信息：

**openssl** **req** **-new** \ # 生成证书签名请求

**-key** **ca-key**.pem \ # 使用**ca-key**.pem中的私钥

**-out** **ca-csr**.pem \ # 输出的**CSR**存入**ca-csr**.pem文件

**-config** **ca**.conf # 使用**ca**.conf中的信息

由于我们是要生成**自签名的根证书**，因此直接使用根密钥签发请求文件ca-csr.pem， 就得到了自签名的根证书ca-cert.pem：

openssl x509 -req \ **# 生成x509证书**

-extensions ca\_cert \ **# 在证书中添加ca\_cert段声明的扩展**

-extfile ca.conf \ **# 包含扩展声明的文件**

-**in** ca-csr.pem \ **# 输入文件为ca-csr.pem**

-**out** ca-cert.pem \ **# 输出文件为ca-cert.pem**

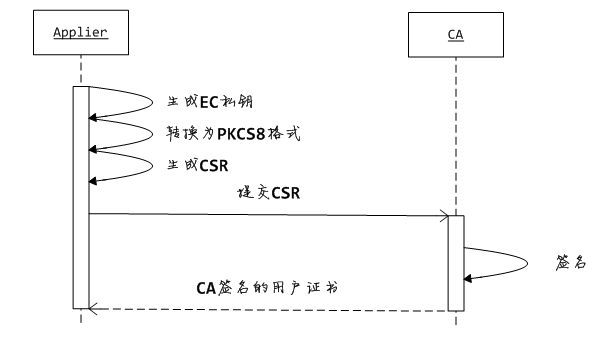
-signkey ca-key.pem \ **# 用来自签名的私钥**

-CAcreateserial \ **# 创建证书序列号**

-days 3650 **# 证书有效期为3650天，大约10年**

**签发用户证书**

一旦有了根证书，我们就可以签发用于节点、排序器或应用用户的证书了。 前面的步骤基本一样，都是生成EC私钥、转换为PCKS8格式、生成证书签名请求，只是分别由申请用户和CA完成：



申请用户首先生成私钥及证书签名请求：

**openssl** **ecparam** **-name** **prime256v1** **-genkey** **-out** **user-key\_**.pem

**openssl** **pkcs8** **-topk8** **-nocrypt** **-in** **user-key\_**.pem **-out** **user-key**.pem

**openssl** **req** **-new** **-key** **user-key**.pem **-out** **user-csr**.pem **-config** **user**.conf

最后的签名是由CA完成的，因此改为如下命令：

openssl x509 -req \

-extensions user\_cert \

-extfile ca.conf \

-**in** user-csr.pem \

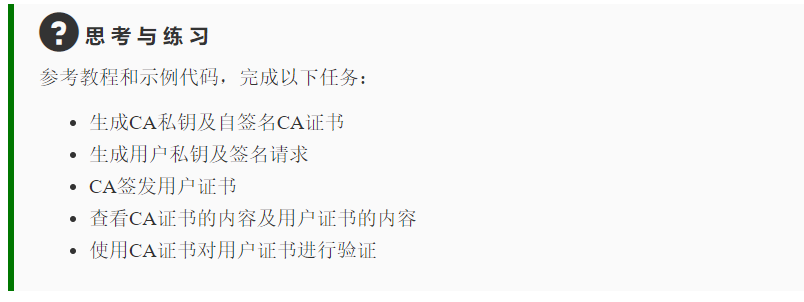
-**out** user-cert.pem \

-CA ca-cert.pem \ **# CA证书**

-CAkey ca-key.pem \ **# CA私钥**

-CAcreateserial \

-days 3650



### MSP：成员服务提供器

在许可制的系统中，仅有身份标识还不够，还需要一种机制来判定一个身份 标识是否是系统可接受的。例如，你可能有好几张不同银行发行的信用卡， 但在某个特定的超市却只能使用其中的一部分卡进行支付：



在Fabric中，这一判定机制由成员服务提供器（MSP：**M**embership **S**ervice **P**rovider） 来实现。每一个需要保护的资源都有自己的MSP，并利用MSP实现从身份向 角色的转换。

例如，对于Orderer和Peer而言，它们都需要一个本地的MSP配置（默认为：msp）目录来建立 其MSP实例，并且按照约定的目录结构来组织各种文件：

* admincerts：该目录中的所有PEM证书对应有管理员角色
* cacerts：该目录的每个PEM证书都是CA根证书
* keystore：该目录中包含节点私钥的单一PEM证书
* signcerts：该目录中包含节点的X509证书
* crls：可选，该目录中包含所有已回收的证书
* intermediatecerts：可选，该目录中的PEM证书对应中间CA证书
* config.yaml：可选
* tlscacerts：可选，该目录中包含用于TLS的CA根证书
* tlsintermediatecerts：可选，该目录中包含用于TLS的CA中间证书

我们可以将之前使用openssl创建的证书和user私钥整理为MSP配置目录，例如， 下面的命令创建必要的MSP目录并将相应的证书或密钥拷贝如对应的目录：

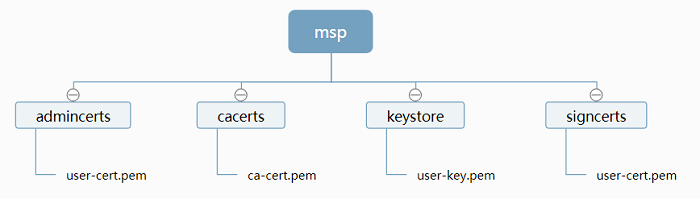
**mkdir** -p msp/admincerts && cp ../mini-ca/user-cert.pem msp/admincerts *# 赋予user管理员角色*

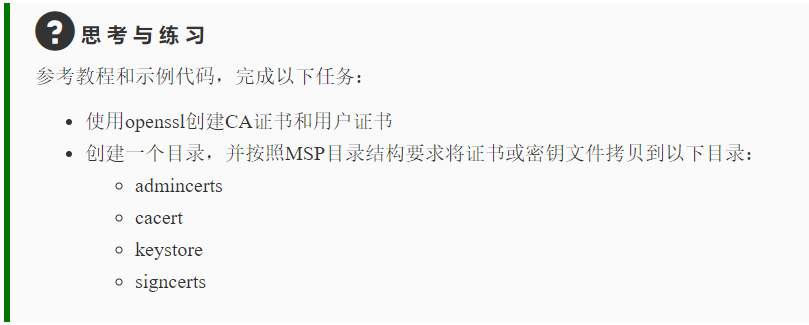
**mkdir** -p msp/cacerts && cp ../mini-ca/ca-cert.pem msp/cacerts *# CA证书*

**mkdir** -p msp/signcerts && cp ../mini-ca/user-cert.pem msp/signcerts *# 当前身份为user*

**mkdir** -p msp/keystore && cp ../mini-ca/user-key.pem msp/keystore *# 当前身份的私钥*

最终的目录结构如下图所示：



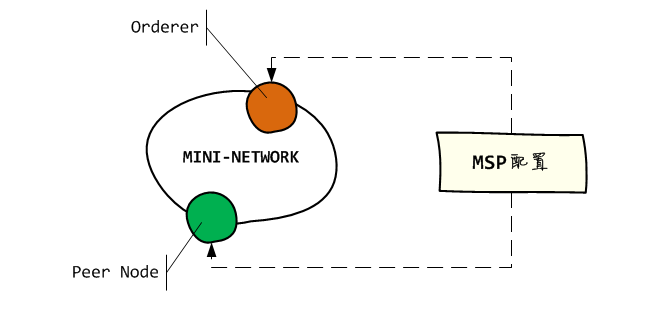


### 启动最小Fabric网络（终端1、终端2）

$ cd ~/repo/c2/solo-network

注意！！！需要清理之前的数据，比如data和artifact文件夹。不然会报错。

有了MSP配置，我们就可以启动一个最简单的Fabric网络了：



在这个网络中，只包含一个Orderer节点和一个Peer节点，而且两个节点使用同一个MPS配置 —— 这样做的好处是我们可以暂时忽略身份的差异对系统功能的影响，以便将注意力集中在对Fabric的整体理解上。

**启动排序节点（终端1）**

**启动排序节点之前，需要首先生成一个创世块，排序节点将利用这个创世块启动整个Farbic网络的系统通道。**

**可以使用官方提供的configtxgen工具来生成Farbic网络系统通道的创世块（**注意它不同于peer节点加入到普通通道!!!**）。configtxgen依赖于配置文件configtx.yaml中定义的网络结构 、访问策略等信息，并提供了不同的Profile以便应用于不同的场景， 可以认为Profile是网络结构、访问策略等网络配置信息的组合套餐。**

**例如，下面的命令使用SampleInsecureSolo套餐配置生成适用于单节点网络 的创世块并存入genesis.pb文件：**

$ configtxgen -outputBlock genesis.block \ *# 生成创世块并存入genesis.block*

-profile SampleInsecureSolo *# 使用SampleInsecureSolo配置套餐*

（也可以加入生成genesis.block的步骤，否则会在后续通道初始化出错）

有了创世块，我们就可以启动排序节点了。**排序节点程序依赖于配置文件 orderer.yaml**。同样，Fabric提供了默认的配置文件，我们只需要做简单的个性化调整即可。

例如，下面的命令使用刚才生成的创世块genesis.block启动排序节点：ORDERER\_GENERAL\_GENESISMETHOD=file \ # 使用创世块文件

ORDERER\_GENERAL\_GENESISFILE=./genesis.block \ # 创建块文件的路径

ORDERER\_FILELEDGER\_LOCATION=./data/orderer \ # 系统通道账本（区块链）数据的路径

（注意！！！将提供的start-orderer.sh中的前两行换成这里的两行）（貌似不需要？）

orderer *# 排序节点程序*

需要指出的是，上面的命令使用环境变量来调整排序节点的配置项，你也可以直接修改orderer.yaml中的对应配置项。例如，环境变量ORDERER\_GENERAL\_GENESISMETHOD 对应于orderer.yaml中的General.GenesisMethod配置，而ORDER\_FILEGEDGER\_LOCATION 则对应于orderer.yaml中的FileLedger.Location配置。

**启动对等节点（终端2）**

类似的，对等节点依赖于配置文件core.yaml，我们也可以在Farbic提供的 默认配置文件上进行自定义调整。例如：

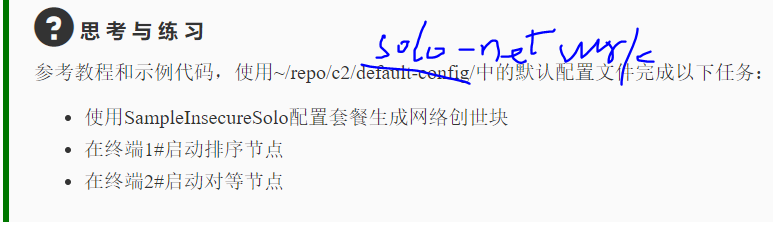
CORE\_PEER\_FILESYSTEMPATH=./data/peer \ # 通道账本（区块链）数据的路径

CORE\_PEER\_LISTENADDRESS=127.0.0.1:7051 \ # 节点监听地址

CORE\_PEER\_CHAINCODELISTENADDRESS=127.0.0.1:7052 \ # 节点链码服务监听地址

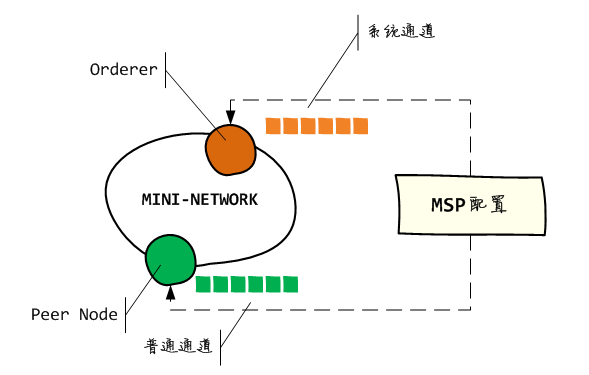
CORE\_CHAINCODE\_MODE=dev \ # 链码服务设置为开发模式

peer node start



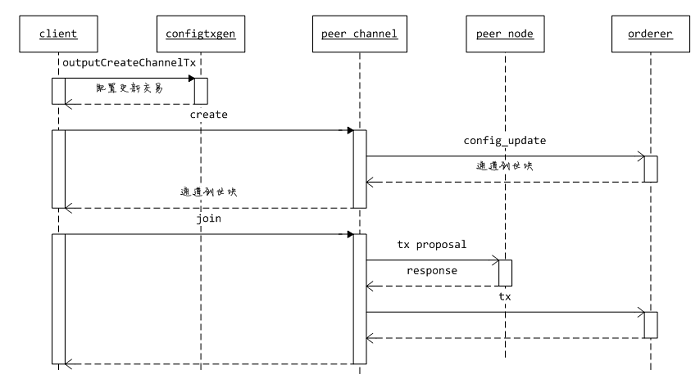
### 通道初始化与节点加入（终端3）

当我们启动排序节点时，利用创世块也同时启动了保存整个网络配置的系统通道：



不过我们通常提到的通道，都是指的由对等节点维护的**普通通道**，一个对等节点可以加入多个普通通道，通道彼此是隔离的，都有自己的区块链和访问控制机制。

节点在初始启动后是没有加入任何通道的，因此我们需要先创建一个通道并让节点加入这个通道。整个流程分为三个步骤：**生成通道创建交易**、**创建通道**和 **节点加入通道**：



* **首先使用configtxgen工具生成通道创建交易：**

configtxgen -outputCreateChannelTx ch1.tx \ # 输出通道创建交易并存入指定文件

-profile SampleSingleMSPChannel # 使用的配置套餐

-channelID ch1 #指定通道ID

* **然后生成通道创世块ch1.block：**

peer channel create -c ch1 \ *# 创建通道ch1*

-f ch1.tx \ *# 使用交易文件ch1.tx*

-o localhost:7050 \ *# 排序器节点地址*

--outputBlock ch1.block *# 输出创世块文件*

（注意！！！如果完全按照教程来的话，这里会出错，是因为之前已经创建过通道ch1的原因。删除data目录：rm -rf data，删除data目录就可以清理之前的数据），然后从启动orderer节点开始。

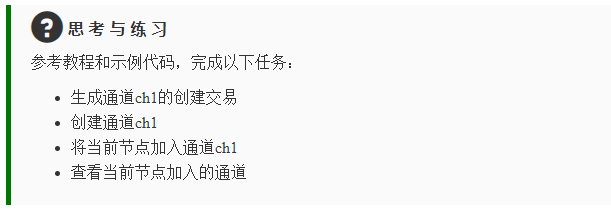
* 最后将当前对等节点加入通道：

peer channel join -b ch1.block -\ *# 通道创世块文件*

-o localhost:7050 *# 排序节点地址*

* 可以使用channel list命令查看当前节点加入的通道：

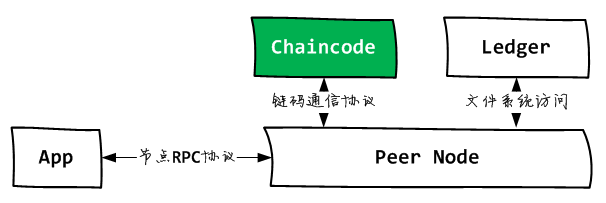
peer channel list



### 开发计数器链码（终端4）

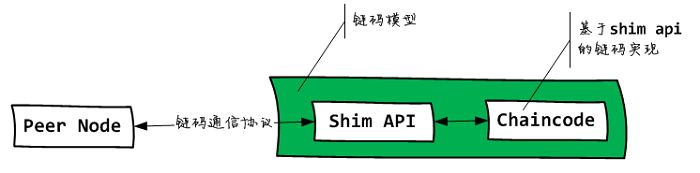
（注意！！！这一节要新的终端进入/repo/c2/counter-cc这个文件夹操作。）

在Fabric中，**智能合约**被称为链码（Chaincode），**链码负责定义区块链交易以及状态，应用程序则通过链码来访问通道区块链**：



与以太坊不同，Fabric并没有实现专门的智能合约虚拟机，链码实际上就是一个 操作系统进程，它通过链码通信协议与Peer节点通信，负责执行节点转发的调用 请求，同时也利用节点提供的API访问通道账本（区块链）。因此理论上可以使用 任何图灵完备的语言开发Fabric链码，不过目前官方仅提供了NodeJS、Java和GO 的链码开发包。

使用Java链码开发包中的[shim API](https://github.com/hyperledger/fabric-chaincode-java" \t "_blank) 开发链码非常简单：只需要实现init()和ivoke() 这两个接口即可。当Fabric通道激活链码时将执行init()方法，而当应用调用链码 方法时，将执行invoke()方法。shim API封装了链码通信协议：



下面的Java代码实现了一个最简的链码，当链码激活时，初始化计数器值为0， 当调用链码方法时，计数器值加1：

**package** com.hubwiz.demo;

**import** org.hyperledger.fabric.shim.ChaincodeBase;

**import** org.hyperledger.fabric.shim.ChaincodeStub;

**public** **class** **CounterChaincode** **extends** **ChaincodeBase**{ *//继承ChaincodeBase*

**@Override**

**public** Response **init**(ChaincodeStub stub){ *//init接口实现*

stub.putStringState("value","0");

**return** newSuccessResponse("init"); *//返回调用端的信息*

}

（注意！！！：示例代码提供的repo/c2/counter-cc/ CounterChaincode.java错误，原因是init方法缺少对value变量的初始化）

**@Override**

**public** Response **invoke**(ChaincodeStub stub){ *//invoke接口实现*

String valueStr = stub.getStringState("value");

**int** value = Integer.parseInt(valueStr);

stub.putStringState("value",(value+1)); *//更新链码状态*

**return** newSuccessResponse(String.format("value updated: %d",value));

}

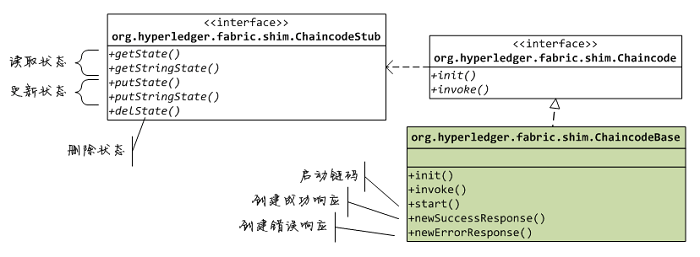
**public** **static** **void** **main**(String[] args){

**new** CounterChaincode().start(args); *//****启动链码***

}

}

链码的基类ChaincodeBase封装了链码通信协议，因此我们开发的链码需要继承这个类， 并重写init()方法和invoke()方法：



init()和invoke()方法的stub参数是Shim API传入的对**通道账本**的**抽象封装**，可以 使用它的putState()方法设置**通道Key-Value状态库**的指定键值，例如上面的代码中在初始化激活时将counter状态的值设置为0；类似的，使用stub对象的getState() 方法获取状态库中指定键的值。需要指出的是，**getState()和putState()的值都采用 byte[]类型**，因此我们使用了语法糖getStringState()和putStringState()，可以方便地处理字符串类型的状态值。

链码实例的start()方法将启动与节点链码服务的连接，因此在上面代码的最后，使用该方法启动Counter链码实例。

**启动链码（终端4）**

**在开发模式下（**在“启动最小Fabric网络”这一节中，就是以开发模式启动peer节点的**），需要自己启动链码，启动链码和运行普通的Java程序一样**， 只是需要指定节点的链码服务访问地址和链码的名称，例如直接用gralde运行：

gradle run --args='--peer.address 127.0.0.1:7052 --id counter-cc:0' # 最后一个参数是：链码ID:版本号

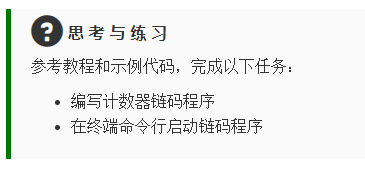
peer.address参数声明链码要连接的节点链码服务地址，id参数声明链码包含版本号的标识名称。

也可以生成jar包后再运行，例如，用下面的终端命令生成jar包：

gradle clean shadowJar # 生成的jar包在build/lib/chaincode.jar

然后直接启动生成的jar包：

java -jar build/lib/chaincode.jar --peer.address 127.0.0.1:7052 --id counter-cc:0



### 链码方法的路由

Fabric的Shim API仅仅定义了init()和invoke()这两个链码接口，如果我们需要细化链码 方法，就需要在这两个接口中进行方法的路由了。不过由于init()方法仅在链码激活时调用，因此通常我们只需要处理invoke()接口的实现。

可以将链码invoke()接口的实现方法进行改进，以便路由到链码实例的其他方法。例如下面的代码为链码定义了两个新的方法，并根据用户请求的第一个参数（视为方法名）分别路由到inc()方法和value()方法，或者返回一个错误响应：

@Override

**public** Response **invoke**(ChaincodeStub stub){

String fcn = stub.getFunction(); **# 链码请求中的第一个参数**

List<String> args = stub.getParameters(); **# 链码请求中的其他参数**

**switch**(fcn){

**case** "inc": **return** inc(stub,args); **# 路由到inc()方法**

**case** "value": **return** **value**(stub,args); **# 路由到value()方法**

}

**return** newErrorResponse("unimplemented method");

}

**private** Response **inc**(ChaincodeStub stub,List<String> args){...}

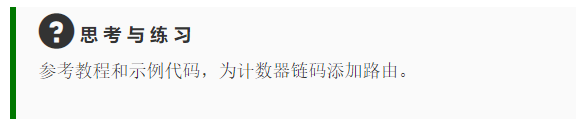
**private** Response **value**(ChaincodeStub stub,List<String> args){...}

当调用链码时，传入的第一个参数对应于上面代码中的fcn变量，而后续的参数则对应于 params数组。例如，对inc()方法的调用方法如下（注意！！！只是个例子，目前链码还没安装和激活，不可请求）：

peer chaincode invoke -n counter-cc \ *# 调用counter-cc链码*

-c '{"Args":["inc","10"]}' *# 调用链码的inc方法，step为10*

-C ch1 *# 通道限定为ch1*



### 链码的安装与激活（终端3）

链码开发后需要部署到通道上才能生效。部署流程分为两步：**1、安装到节点；2、在通道激活。**

使用chaincode install命令安装链码到当前节点上。例如：

peer chaincode install -n counter-cc

-v 0 \ *# 链码名称及版本号*

-l java *# 链码开发语言*

-p /home/user/c2/counter-cc *# 链码全路径*

安装到节点上的链码还需要在特定通道上激活才能被访问。使用chaincode instantiate命令 激活指定的已安装链码：

peer chaincode instantiate -n counter-cc -v 0 \ *# 要激活的链码名称及版本号*

-o localhost:7050 *# 排序节点地址*

-C ch1 *# 在ch1通道上激活链码*

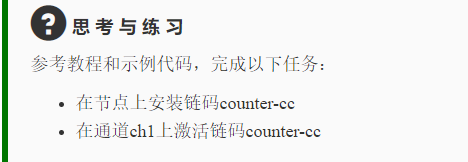
链码激活后就可以使用chaincode query或chaincode invoke命令调用链码了， 两者的区别在于，前者用于对链码状态的查询而后者用于提交交易。

例如，下面的命令提交链码交易，在执行后链码状态counter将加1：

peer chaincode invoke -n counter-cc -C ch1

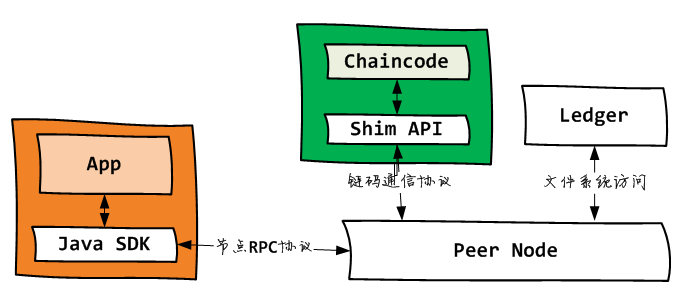
而下面的命令虽然也可以调用链码，但是它并不能更新链码状态，因此无论 调用多少次都返回相同的结果：

peer chaincode query -n counter-cc -C ch1

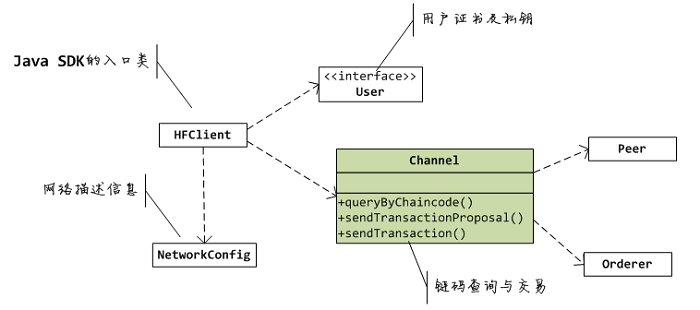


### 在应用中访问链码

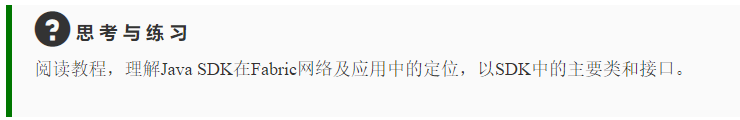
在Fabric中，应用通过节点的RPC协议接口访问链码：



类似于Shim API对链码通信协议的封装，Fabric Java SDK提供了对节点RPC协议接口的封装，其入口**类为HFClient**，对**链码的查询**和**交易操作**则封装在**Channel类**中：

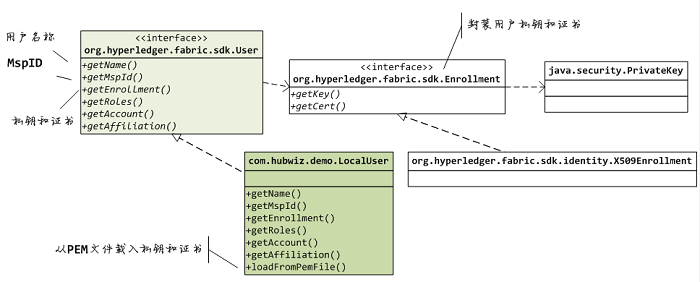


由于Fabric是许可制区块链，因此应用也需要持有证书和私钥，HFClient实例 依赖于User接口的实现对象来访问一个特定身份的证书和私钥，**因此在访问 链码之前，我们需要首先定义一个简单的User接口实现类。**



### User接口实现类

HFClient实例访问Fabric网络的身份使用User接口实现对象来表征。因此我们需要实现一个简单的User接口实现类：



首先完成实现类LocalUser的骨架：

**import** org.hyperledger.fabric.sdk.User;

**import** org.hyperledger.fabric.sdk.security.CryptoPrimitives;

**public** **class** **LocalUser** **implements** **User**{ *//实现User接口*

**private** String name;

**private** String mspId;

**private** Enrollment enrollment;

LocalUser(String name,String mspId){

**this**.name = name;

**this**.mspId = mspId;

}

**private** Enrollment **loadFromPemFile**(String keyFile,String certFile){ */\*见下文说明\*/* }

**@Override** **public** String **getName**(){ **return** name; }

**@Override** **public** String **getMspId**() { **return** mspId}

**@Override** **public** Enrollment **getEnrollment**() { **return** enrollment; }

**@Override** **public** String **getAccount**() { **return** **null**; }

**@Override** **public** String **getAffiliation**() { **return** **null**; }

**@Override** **public** Set<String> **getAccounts**() { **return** **null**; }

}

在Fabric Java SDK中，**Enrollment接口用来提供对用户的私钥和证书的访问**， 并且预置了一个适合X509证书的实现类X509Enrollment，因此我们可以从本地 MSP目录中的PEM文件中载入**用户私钥**和**签名证书**：

**private** Enrollment **loadFromPemFile**(String keyFile,String certFile) throws Exception{

**byte**[] keyPem = Files.readAllBytes(Paths.**get**(keyFile)); *//载入私钥PEM文本*

**byte**[] certPem = Files.readAllBytes(Paths.**get**(certFile)); *//载入证书PEM文本*

CryptoPrimitives suite = **new** CryptoPrimitives(); *//载入密码学套件*

PrivateKey privateKey = suite.bytesToPrivateKey(keyPem); *//将PEM文本转换为私钥对象*

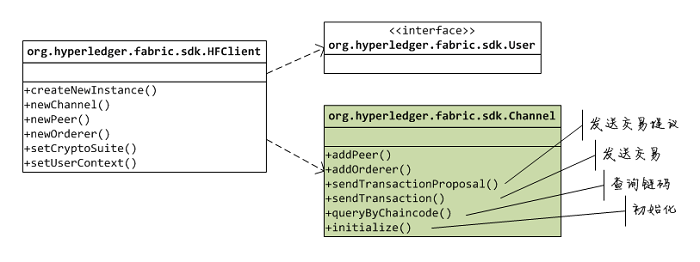
**return** **new** X509Enrollment(privateKey,**new** String(certPem)); *//创建并返回X509Enrollment对象*

}



### 通道访问参数设置

如前所述，Java SDK的入口类是HFClient，不过我们要访问链码则主要是 通过Channel对象来完成的，例如查询链码或者向链码提交交易：



因此首先创建一个HFClient实例，并设置它的密码学套件和用户身份：

HFClient client = client.createNewInstance();

client.setCryptoSuite(CryptoSuite.Factory().getCryptoSuite());

LocalUser user = **new** LocalUser("admin","SampleOrg");

user.loadFromPemFile("msp/keystore/user-key.pem","msp/signcerts/user-cert.pe");

client.setUserContext(user);

接下来就可以创建Channel对象了。由于Channel对象需要向背书对等节点 发送交易提议，也需要向排序节点发送交易，因此还需要为Channel对象 添加对等节点和排序节点，并在配置之后进行必要的初始化：

Channel channel = client.newChannel("ch1");

Peer peer = client.newPeer("peer1","grpc://127.0.0.1:7051");

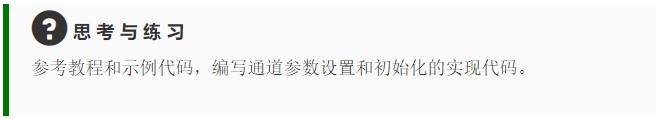
channel.addPeer(peer);

Orderer orderer = client.newOrderer("orderer1","grpc://127.0.0.1:7051");

channel.addOrderer(orderer);

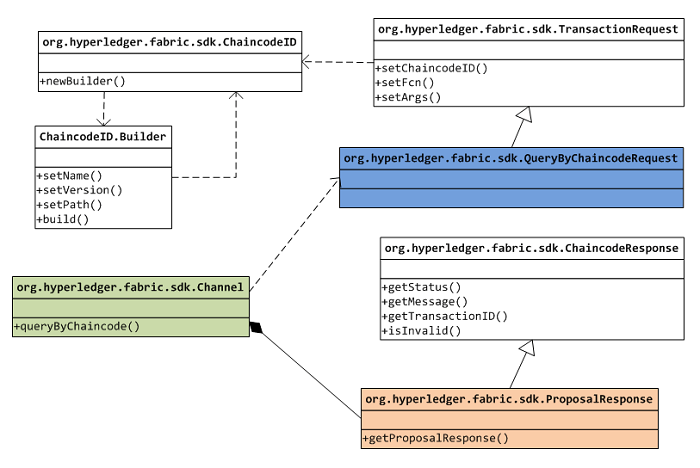
channel.initialize();

（注意！！！这一段代码的意思应该是该客户端节点加入通道ch1，并以orderer1和peer1节点为锚点，与orderers节点和peers节点交互）



### 链码查询

通道对象Channel提供了链码的查询方法queryByChaincode()，其输入参数为一个 QueryByChaincodeRequest对象，而返回值则是ProposalResponse的集合：



下面我们通过对counter-cc链码的value方法的调用，来详细学习如何在Java代码 中实现链码查询功能。

首先按照要求准备请求对象，由于链码的value方法不需要参数，因此我们只需要 设置链码的ID和方法名即可：

QueryByChaincodeRequest req = client.newQueryProposalRequest();

ChaincodeID cid = ChaincodeID.newBuilder().setName("counter-cc").build();

req.setChaincodeID(cid);

req.setFcn("value");

现在把准备好的请求对象传入通道的queryByChaincode()方法，以便它提交给节点处理：

Collection<ProposalResponse> rspc = channel.queryByChaincode(req);

**queryByChainCode()返回的是一组ProposalResponse，为什么？**

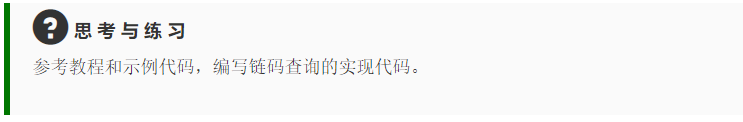
这是因为如果在通道对象上添加了多个对等节点（一个客户端节点可以添加多个peer节点为它服务？），queryByChaincode()方法会向所有的节点发送请求并收集所有的响应到一个集合对象中再返回：

**for**(ProposalResponse rsp: rspc){

System.**out**.format("status: %d\s",rsp.getStatus().getStatus());

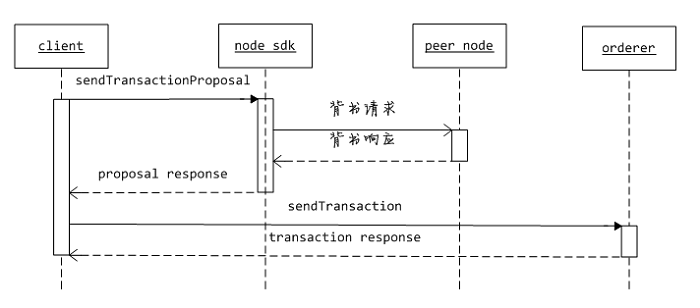
System.**out**.format("payload: %s\n", rsp.getProposalResponse().getResponse().getPayload().toStringUtf8());

}



### 链码交易

向链码提交交易则复杂一些，需要依次使用通道的sendTransactionProposal()和sendTransaction() 这两个方法，分别向对等节点和排序器提交请求。这个过程的时序表示如下：



首先构造请求对象TransactionProposalRequest：

TransactionProposalRequest req = client.newTransactionProposalRequest();

ChaincodeID cid = ChaincodeID.newBuilder().setName("counter-cc").build();

req.setChaincodeID(cid);

req.setFcn("inc");

req.setArgs("10");

然后向对等节点提交提议，等待节点背书结果集：

Collection<ProposalResponse> rsp = channel.sendTransactionProposal(req);

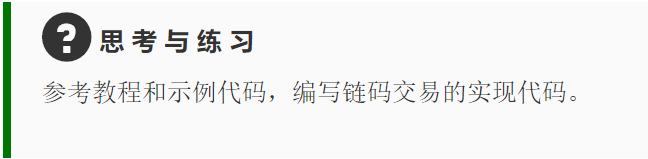
可以对背书结果集进行检查，以避免将无效的响应提交给排序节点，也可以简单 地直接提交给排序节点：

CompletableFuture<TransactionEvent> rsp = channel.sendTransaction(rsp);

TransactionEvent **event** = rsp.**get**();

System.**out**.format("txid: %s\n", **event**.getTransactionID());

System.**out**.format("valid: %b\n", **event**.isValid());

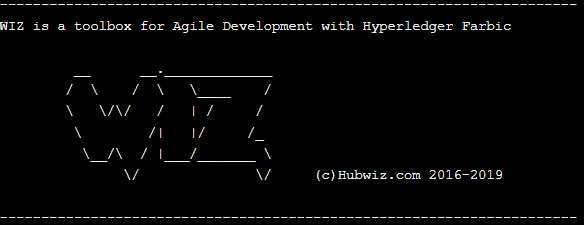


### 运行/repo/c2/counter-dapp（终端5）

（注意！！！：示例代码提供的repo/c2/counter-dapp/ App.java中的” Peer peer = client.newPeer("peer1`","grpc://127.0.0.1:7051");”错误，删除`即可）

### 使用WIZ开发工具箱（终端1-5）

**Fabric的环境配置相当繁琐**，为了提高开发效率，课程提供了一个便利的开发工具箱**WIZ**：



wiz提供了一个简单的Fabric应用开发模板，以及一组用于提高效率的脚本。

**初始化项目目录**

创建一个新目录，然后执行wiz init命令初始化目录结构：

~$ mkdir -p ~repo/test-wiz && cd ~/repo/test-wiz

~/repo/test-wiz$ wiz init

wiz项目的目录结构组织如下：

* network：fabric网络定义与配置数据目录
* chaincode：链码源代码目录
* application：应用源代码目录
* script：脚本目录

**启动网络（终端1）**

在wiz项目目录内，使用wiz net start命令启动网络，在初次启动网络前 需要使用wiz net reset命令初始化网络配置。例如，在1#终端执行如下命令：

~/repo/test-wiz$ wiz net rest

~/repo/test-wiz$ wiz net start

**启动链码（终端2）**

在wiz项目目录内，使用wiz cc start命令启动链码。例如，在2#终端 执行如下命令：

~/repo/test-wiz$ wiz cc start

**启动通道环境（终端3）**

在wiz项目目录内，使用wiz ch start命令启动通道并进行必要的环境初始化， 包括将当前节点加入通道、安装并激活链码。例如，在3#终端执行如下命令：

~/repo/test-wiz$ wiz ch start

**启动管理终端（终端4）**

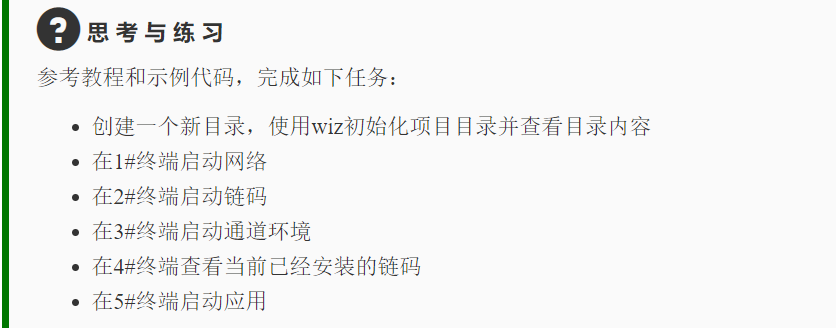
在wiz项目目录内，使用wiz admin命令进入管理终端，管理终端配置了必要的环境变量，因此可以直接以管理员身份执行peer命令。例如，在4#终端执行如下命令进入管理终端，并查看当前节点加入的通道：

~/repo/test-wiz$ wiz admin admin@org1> peer channel list

**启动应用（终端5）**

在wiz项目目录内，使用wiz app start启动项目应用。例如，在5#终端执行 如下命令：

~/repo/test-wiz$ wiz app start

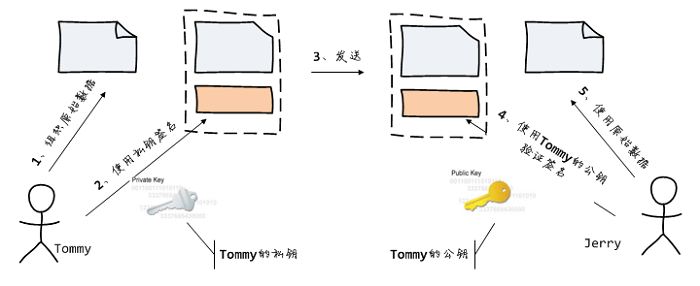


## 身份与权限管理机制

### 非对称密码学

非对称密码学是密码学中的一种算法，也被称为公钥密码学，因为它需要两个密钥：一个 是可以公开的密钥即公钥，另一个则是需要保密的私有密钥即私钥。

对称加密采用一对密钥（私钥、公钥）进行数据的加密或解密： 用私钥加密，则需要用 公钥解密；用公钥加密，则需要用私钥解密。 这一非对称特性使得其非常适合用于身份 表示与验证 —— 公钥用于 身份的表示，而私钥则用于身份的验证：



在上图中，当tommy需要向jerry发送原始数据时，他首先使用自己的私钥对 原始数据进行签名， 得到的签名数据附加在原始数据后面，一同发送给jerry。 jerry收到数据后，使用tommy的公钥 就可以验证签名是否是由tommy的私钥签发的， 从而确认该数据确实来自于tommy。

例如，我们可以用openssl来实践这一过程。

首先为tommy创建私钥，然后导出公钥，分别保存在tommy-priv.pem和tommy-pub.pem中：

openssl ecparam -genkey \ **# 生成私钥**

-name prime256v1 \ **# 椭圆曲线参数为prime256v1**

-**out** tommy-priv.pem **# 输出保存文件**

openssl ec -pubout \ **# 生成公钥**

-**in** tommy-priv.pem \ **# 输入私钥文件**

-**out** tommy-pub.pem **# 输出公钥文件**

然后使用tommy的私钥对hey-jude.txt文件签名，并将得到的二进制签名文件signature.sha256 进行base64编码得到签名的ASCII表示并存入文件signature.base64，以便通过邮件等形式发送：

~$ openssl dgst -sha256 \ *# 生成sha256哈希*

-sign tommy-priv.pem \ *# 签名私钥*

-out signature.sha256 \ *# 输出文件* hey-jude.txt *# 待签名文件*

~$ openssl base64 -**in** signature.sha256 \ *# 输入哈希文件*

-out signature.base64 *# 输出base64编码文件*

当jerry收到hey-jude.txt和signature.base64后，首先将base64的编码还原回二进制格式， 然后就可以利用tommy的公钥进行签名验证：

~$ openssl base64 -d -**in** signature.base64 \ *# 输入base64编码文件*

-out signature.bin *# 输出解码后的哈希文件*

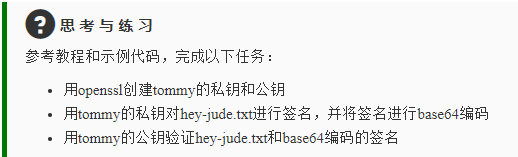
~$ openssl dgst -verify tommy-pub.pem \ *# 使用指定的公钥验证签名*

-sha256 \ *# 签名哈希算法为sha256*

-signature signature.bin \ *# 签名文件*

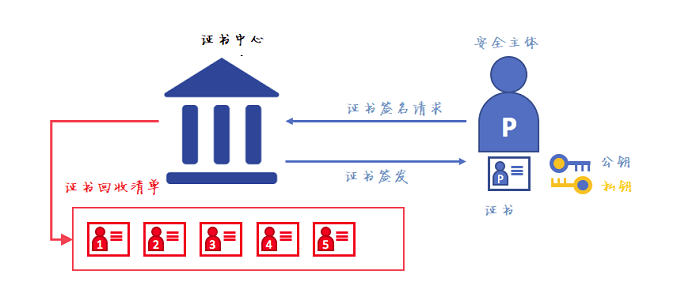
hey-jude.txt *# 原始数据文件*

一旦认证OK，就可以表明：1、签名确实是jerry的 2、hey-jude.txt原文未被篡改。



### PKI体系与MSP机制

在去中心化的区块链中，例如比特币和以太坊，采用公钥或者将公钥单向变换为地址， 就可以作为用户身份标识了，因此在这些区块链平台，并不需要中心化的身份生成与 验证。但是在Fabric中，应该是考虑到要实现相对复杂的**访问权限控制**，因此引入了中心化的PKI体系：



在PKI中，首先需要一个证书颁发机构（CA：Certificate Authority）来统一 的为用户签发证书（Certificate），在证书中包含有用户的公钥，因此第三方可以 方便地验证某个用户的证书是否是指定的CA所颁发。

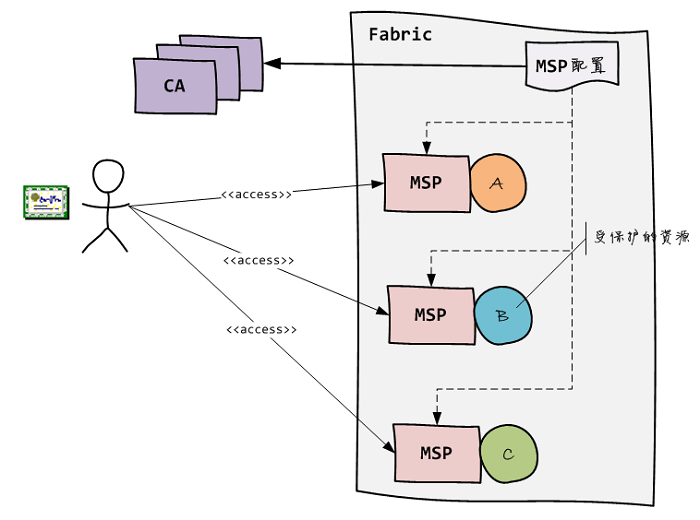
需要指出的是，密钥对还是由用户自己生成，CA只负责为用户提交的信息进行签名即可得到证书。 在上述过程中，用户提交的信息被称为证书签名请求（CSR：Certificate Signing Request）， 它除了需要包含用户的公钥之外，还需要包含关于持有者（Subject）的一些额外信息， 例如所属机构名称等等。

**MSP：会员许可机制**

我们知道，Fabric是许可制区块链平台，因此它需要PKI这种中心化的机制 来管理证书的发放与回收，同时也是利用CA来验证用户持有的证书是否有效。

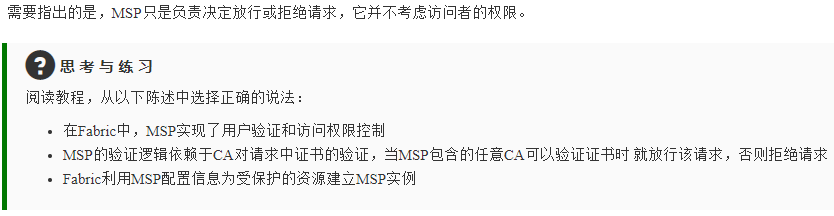
可以认为CA本身就提供了一种会员（Membership）机制：只有持有CA签发证书的用户，才可以访问区块链，否则将被拒之门外——会员制是一种有效的身份验证机制。

Fabric将会员验证机制定义为MSP（Membership Service Provider）组件， 它使用MSP来保护区块链上的每一个资源：当用户访问区块链时，**相应资源对应 的MSP**将验证用户是否属于该资源的会员，并根据检查结果拒绝或放行用户的访问：



Fabric会利用MSP配置信息为每个受保护资源（例如**链数据、排序节点**等）建立一个MSP实例，当用户试图访问受限资源时，首先需要通过MSP的验证。验证的逻辑 很简单：Fabric提取用户请求中的证书信息，然后验证是否是**MSP中的CA**签发的证书，是的话就放行，不是的话就拒绝。

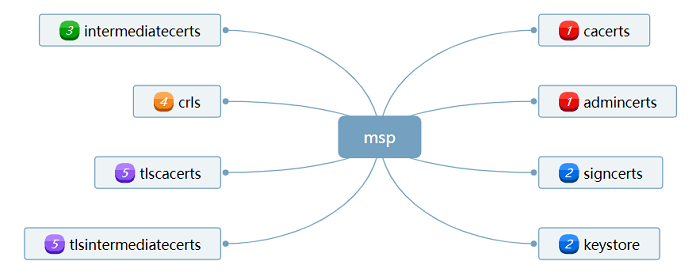
需要指出的是，MSP只是负责决定放行或拒绝请求，它并不考虑访问者的权限。



### 使用cryptogen配置本地MSP

MPS配置信息有两种，对于**排序器和对等节点**而言，其MSP配置保存在本地文件夹中，这种配置信息被成为**本地MSP配置**；而对于通道而言，其MSP配置保存在**通道区块**中，这种配置信息被称为**通道MSP配置**。让我们以Peer程序为例进一步了解理解本地MSP的配置与作用。

Fabric约定了本地MSP配置信息的目录结构，以便节点启动时可以顺利的利用 配置信息建立MSP实例：



对于MSP的本职工作而言，只有两个目录是必须的，用来验证用户证书的有效性：

* cacerts：存储本MSP认可的合法CA根证书
* admincerts：存储本MSP认可的所有管理员证书

如果存在中间CA，可以将这些中间CA的证书放入可选的intermediatecerts目录， 以便建立完整的证书验证链。类似的，如果存在回收的证书，那么应当将 这些已经被回收的证书放入可选的crls目录。

本地MSP除了用于验证来访身份是否有效之外，还有另一个作用，就是提供当前节点的标识信息和签名私钥，因此下面两个目录也是必须的：

* signcerts：保存节点提交请求时要使用的证书，作为节点的身份标识
* keystore：保存身份证书对应的私钥

如果节点要启用tls安全通信，那么应当使用以下两个目录存放tls通信的 CA根证书和中间证书：

* tlscacerts：保存用于TLS通信的CA根证书
* tlsintermediatecerts：保存用于TLS通信的CA中间证书

之前我们使用openssl生成密钥和证书，然后手工构建了一个本地MSP目录。 现在我们使用Fabric官方提供的cryptogen工具来一步生成MSP配置。

cryptogen命令依赖于配置一个描述组织机构的YAML配置文件。例如下面的 配置文件cryptogen-config.yaml描述了一个包含两个机构、具有单一排序节点的Fabric网络构成：

OrdererOrgs: *# 定义排序节点机构*

- Name: Orderer *# 机构名称*

Domain: example.com *# 机构域名*

Specs: *# 机构MSP配置生成约定*

- Hostname: orderer *# 主机名*

PeerOrgs: *# 定义对等节点机构*

- Name: Org1 *# 机构名称：Org1*

Domain: org1.example.com *# 机构域名*

EnableNodeOUs: **false**

Template: *# 对等节点配置生成参数*

Count: 1 *# 生成节点数量：1*

Users: *# 用户配置生成参数*

Count: 1 *# 除Admin之外另生成的用户数量：1*

- Name: Org2

Domain: org2.example.com

EnableNodeOUs: **false**

Template:

Count: 1

Users:

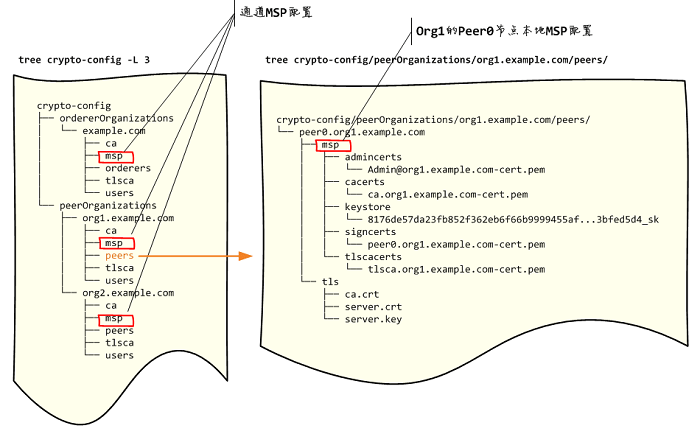
Count: 1

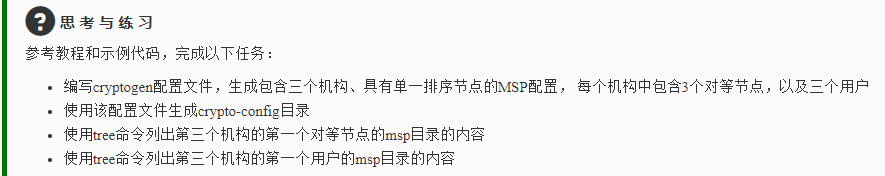
一旦完成配置文件的定义，就可以使用cryptogen的generate子命令生成 MSP配置了：

cryptogen generate \ *# 生成MSP配置*

--config cryptogen-config.yaml *# 使用指定的配置文件*

生成的结果保存在crypto-config目录中，你可以使用tree命令查看该目录内容，例如：

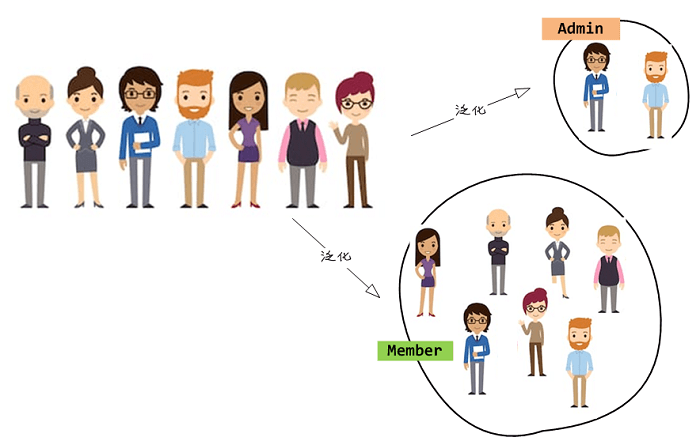




### 从个体身份到安全主体

我们用证书标识单一用户的身份，但是对于权限控制来讲，使用针对 特定个体的访问控制策略这种粒度太小了，不是灵活的解决方案。 例如，当我们对用户A制订了访问控制策略后，如果后续又添加了新 的用户B，那么我们还需要再补充针对用户B的访问控制策略。

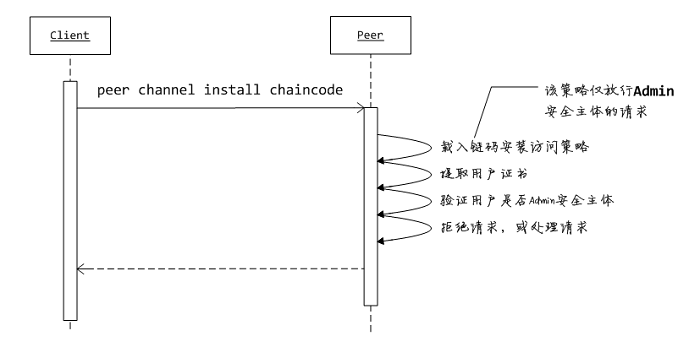
因此常用的解决思路是针对具有共性的**个体/ Identity**集合来制定访问控制策略， 这种具有共性的个体集合，被称为**安全主体/ Principal**。可以认为 Principal是泛化的Identity，例如，可以将用户泛化为管理员（Admin） 和普通成员（Member）这两种角色而忽略其个体特征，这样只需要针对 Admin和Memeber这两种安全主体制定访问控制策略就行了：

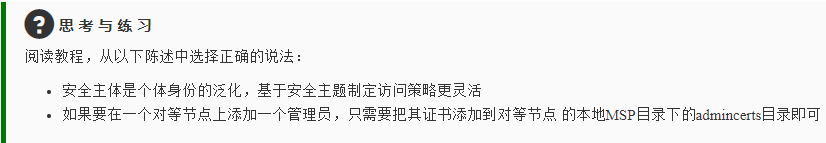


按角色进行泛化只是Fabric支持的四种安全主体分类之一。这四种分类列举如下：

* role：按角色泛化，例如：管理员、普通成员、节点、应用程序
* organization\_unit：按组织部门泛化
* identity：对应单独个体，在这种情况下没有泛化
* anonymous：匿名

例如，在peer节点上安装链码的策略是**要求安全主体是该节点的管理员（Admin）**，那么当用户尝试在peer节点上安装链码时，节点会提取请求消息中的用户证书，然后判断该证书是否满足Admin安全主体的要求，只有当用户身份成功映射为Admin主体 后peer节点才会执行链码的安装流程 —— 对于本地MSP而言，这一要求就是用户证书必须出现在本地MSP配置文件夹的admincerts目录下：

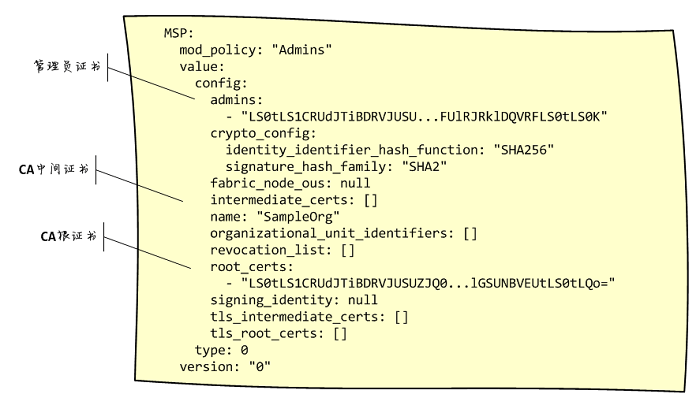




### 通道MSP配置

对通道资源，例如链码的访问，其身份验证由通道MSP完成。容易理解，通道的MSP配置信息保存在区块上，以便在所有节点上共享这一配置信息。

通道链上保存配置信息的区块是一种特殊类型的区块，被称为配置区块，它 的数据载荷不是交易，而是有关通道的完整的配置信息，其中包含了MSP配置：



**提取通道MSP配置**

可以使用peer channel来获取指定通道的当前配置区块，例如，获取系统通道 sysch的当前配置区块，并写入sysch.block文件：

peer channel fetch config \ *# 提取通道配置区块*

-c test-system-channel-name \ *# 通道ID*

-o localhost:7050 \ *# 排序节点地址*

sysch.block *# 配置区块保存文件*

注意，由于系统通道只保存在排序节点上，因此我们需要指定排序节点的连接地址，例如本机的7050端口。

**查看通道MSP配置**

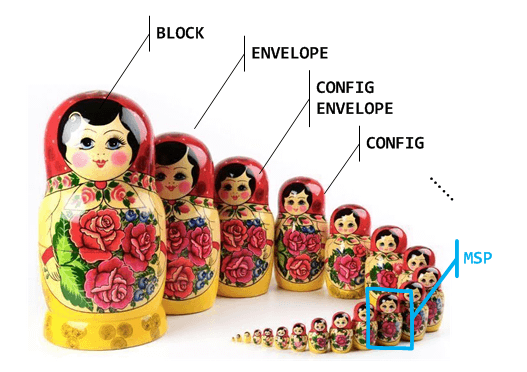
区块文件是二进制的，我们需要使用configtxgen命令将其转换为JSON格式， 并进一步转换为适合人类阅读的YAML格式：

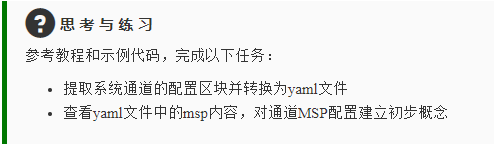
**configtxgen** **-inspectBlock** **sysch**.block > **sysch**.json

**json2yaml** **sysch**.json > **sysch**.yaml

**（注意：这里按照示例代码会出错，需要按顺序安装nodejs、jquery、json2yaml和yaml2json即可）**

Fabric的配置区块是层级嵌套的结构，需要一点时间从里面找到MSP配置组，这有点像俄罗斯套娃，MSP配置信息藏在其中的某一层：

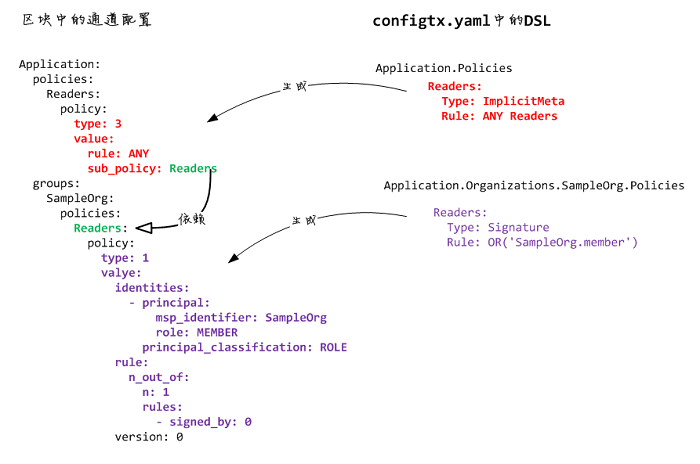




### 通道资源访问策略

访问通道的身份验证是**通道MSP组件**完成的，Fabric通道配置区块中的信息 实例化通道MSP组件，用来拦截非法用户对通道资源的访问请求。 而对于通过MSP组件验证的请求而言，是否能够成功执行还取决于该请求要访问 的资源的**访问策略/Policy**。

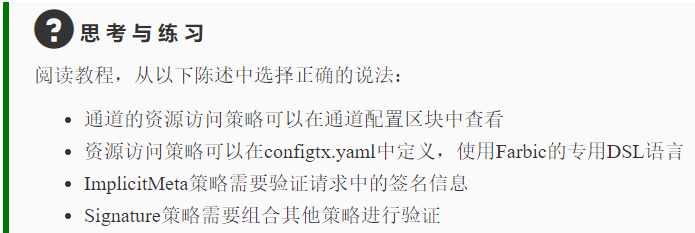
例如，当用户试图调用peer channel getinfo获取通道链的信息时，需要满足Channel/Application/Readers策略的要求，那么什么样的用户符合这一 策略的要求？让我们查看一下通道中配置区块的内容，以及对应的configtx.yaml配置文件中的内容：



在配置区块中的信息表明，Application/Readers策略是一个**ImplicitMeta** 类型的策略，其规则是ANY Readers，意思是如果来访用户满足任意后代配置组中的 名为Readers的策略就视为满足本策略 —— ImplicitMeta策略的验证完全依赖于其他策略， 而并不直接检查来访用户的身份信息。

在上面的配置中，我们看到后代的Application/SampleOrg/Readers策略， 它是一个**Signature**类型的策略，其规则为OR('SampleOrg.member')，即SampleOrg 中的任意成员（Member）均可满足此要求，显然，这是通过验证来访用户的证书是否由 SampleOrg的CA所颁发来实现的。

因此。刚才的问题答案就是，SampleOrg机构中的任何成员都可以满足Channel/Application/Readers 访问策略的要求，因为这些成员可以满足Channel/Application/SampleOrg/Readers 策略的要求，所以它也满足了Channel/Application/Readers策略的要求。



### 策略的DSL描述语言

在configtx.yaml配置文件中，策略是使用Farbic自己发明的**小轮子**语言来描述的，可以视为一种领域专用语言（DSL：Domain Specific Language）。

策略DSL规定了两种策略类型：ImplicitMeta和Signature。如前文所言，ImplicitMeta 策略的验证依赖于后代策略的验证，而Signature策略的验证则依赖于请求中包含的签名信息。

ImplicitMeta策略的语法是：

<KEYWORD> <SUB-POCICY>

ImplicitMeta策略有三个关键字：ANY、ALL、MAJORITY。ANY表示通过任何后代策略均视为通过本策略的验证，ALL表示必须通过所有后代策略的验证才能视为通过本策略的 验证，而MAJORITY表示必须通过超过半数的后台策略的验证才可以视为通过本策略的验证。

例如：

* ANY Readers：表示任何后代Readers策略满足即可
* ALL Writers：表示所有后代Writers策略都满足才可以
* MAJORITY Admins ：表示超过半数的后代Admins策略都满足才可以

Signature策略的语法是：

<GATE>(<PARAM1>[,<PARAM2>...])

Siganture策略的门限关键字有三种：OUTOF、OR、AND。例如下面的规则 表示请求中应当包含两个签名，并且分别由Org1成员和Org2成员签署：

OUTOF(2,'Org1.member','Org2.member')

AND和OR都可以视为具有特定门限值的OUTOF规则，例如下面的规则 等价于上面的OUTOF描述：

**AND**('Org1.member','Org2.member')

而OR则可以视为门限值为1的OUTOF规则，例如下面的两条规则是等价的：

**OR**('Org1.member','Org2.member')

OUTOF(1,'Org1.member','Org2.member')

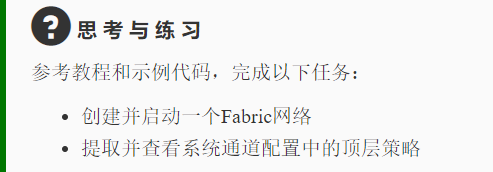
即请求中只需要包含一个签名即可，由Org1成员或者Org2成员签署都可以。

在Signature规则中出现的安全主体的表示方式为：

<ORG>.<ROLE>

其中的ORG表示机构的MSPID，而ROLE可以是如下四种之一：

* member：成员
* admin：管理员
* peer： 对等节点
* client：应用程序



### 通道访问控制清单

策略是一种被动型资产，你有了策略，还需要在合适的时候启动它，对于访问权限控制策略来讲，意思就是还需要指定一个策略要保护的资源，才能当用户 访问这个资源时激活策略。

在Fabric中有一些资源的保护策略是在代码中硬编码的，还有一些资源则提供了可以定制新策略的接口，这些可供自定义保护策略的资源及其对应的访问控制策略，就被称为**访问控制清单**（ACL - **A**ccess **C**ontrol **L**ist）。

例如，默认情况下，对于系统链码lscc的GetInstantiatedChaincodes方法的访问， 受到/Channel/Application/Readers策略的保护，如果我们定义了一个新的策略， 那么用新的策略在configtx.yaml中进行替换就可以了：

**Application:**

ACLs:

lscc/GetInstantiatedChaincodes: /Channel/Application/PowerReaders *#替换为新策略*

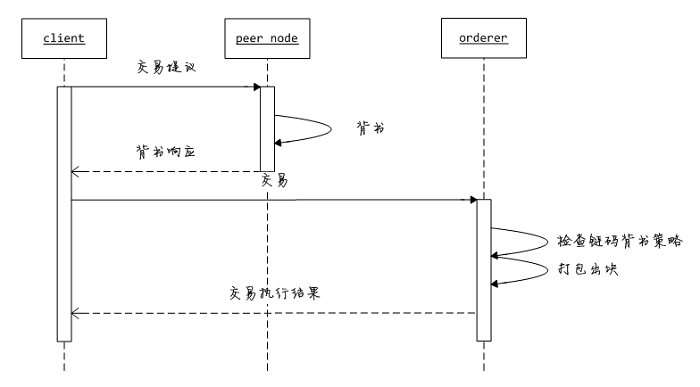
如果我们没有定义访问控制清单，那么Fabric提供了默认的保护策略：

| 资源标识 | 访问控制策略 |
| --- | --- |
| lscc/ChaincodeExists | /Channel/Application/Readers |
| lscc/GetDeploymentSpec | /Channel/Application/Readers |
| lscc/GetChaincodeData | /Channel/Application/Readers |
| lscc/GetInstantiatedChaincodes | /Channel/Application/Readers |
| lscc/GetCollectionsConfig | /Channel/Application/Readers |
| qscc/GetChainInfo | /Channel/Application/Readers |
| qscc/GetBlockByNumber | /Channel/Application/Readers |
| qscc/GetBlockByHash | /Channel/Application/Readers |
| qscc/GetTransactionByID | /Channel/Application/Readers |
| qscc/GetBlockByTxID | /Channel/Application/Readers |
| cscc/GetConfigBlock | /Channel/Application/Readers |
| cscc/GetConfigTree | /Channel/Application/Readers |
| cscc/SimulateConfigTreeUpdate | /Channel/Application/Readers |
| peer/Propose | /Channel/Application/Writers |
| peer/ChainToChain | /Channel/Application/Writers |
| peer/Event\_Block | /Channel/Application/Readers |
| peer/Event\_FilteredBlock | /Channel/Application/Readers |



### 通道链码背书策略

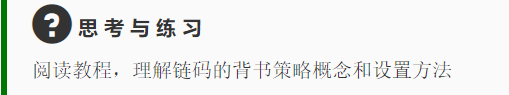
我们知道，提交链码交易是一个两步骤的操作：首先要将提议给背书对等节点，然后 再将响应结果提交给排序节点以便打包出块。在这个操作过程中，排序节点就是根据 链码的背书策略来确定交易的有效性的：



链码默认的背书策略是OR('<Org>.member')，即通道中任何机构的任何成员都可以 进行背书。通常可以在激活链码或更新链码时指定背书策略，例如下面的命令在部署wizcc链码时 指定一个自定义的背书策略，要求只有管理员才可以背书：

peer chaincode instantiate -n wizcc -v 0 -C ch1 \ *# 在通道1激活链码*

-P "OR('Org1.admin')" *# 设置链码背书策略*



## 通道配置与更新

### 系统通道的启动与配置块获取

在启动排序器时，需要指定一个创世区块（或者指定一个配置段由排序器自行 生成创世区块）。排序器就是用这个创世区块来启动引导整个网络的系统通道，系统 通道用来保存整个网络级的配置信息，因此这个创世区块也是一个配置区块。

例如，下面的命令使用configtx.yaml中的OrdererGenesis配置Profile来生成系统通道的创世区块，并将生成的区块数据存入文件genesis.pb：

configtxgen -outputBlock genesis.pb \ *# 创建系统通道创世块并存入指定文件*

-profile OrdererGenesis \ *# 使用configtx.yaml中的OrdereGenesis配置段*

-channelID system-channel *# 系统通道ID为system-channel*

**查看创世区块的内容**

可以将Protocol Buffer编码的区块转换为json格式以方便查看其内容或进行修改， 例如，将其转换为json格式的文件config-block.json：

configtxlator proto\_decode \ *# 解码protocol buffer数据*

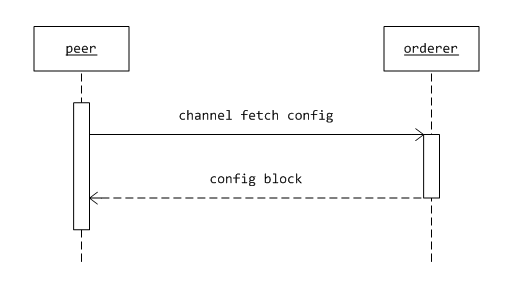
--input config-block.pb \ *# 输入文件*

--type common.Block \ *# 输入数据类型* --output config-block.json *# 输出文件*

如果json文件有很多层嵌套，那么应当将其转化为更适合查看的yaml文件：

**json2yaml** **config-block**.json > **config-block**.yaml

**获取系统通道的配置区块**



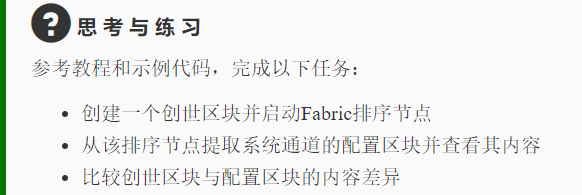
在使用peer channel fetch命令时，使用参数-o声明从排序器而不是从对等节点读取区块， 就可以获取系统通道的当前配置区块。例如：

peer channel fetch config config-block.pb \ *# 提取通道配置区块，写入指定文件*

-o localhost:7050 \ *# 使用排序器的区块分发服务*

-c system-channel *# 系统通道的ID为system-channel*

如果我们在系统通道启动引导后立刻获取配置区块，它的内容应当和我们提交的创世区块一样，你可以将config-block.pb转化为json或者yaml，与创世区块的内容比较一下。



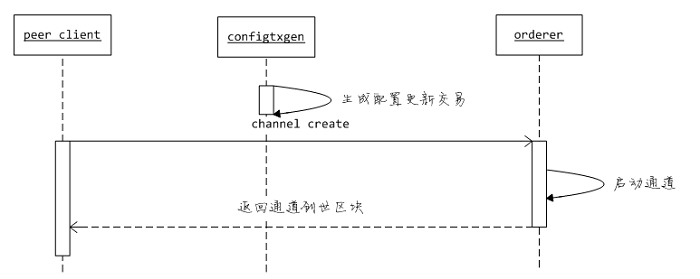
注意，示例代码有误，与下一节内容对比，并进行以下修改：

通道ID得改成: system-channel。

CORE\_PEER\_MSPCONFIGPATH=./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp \

### 标准通道的启动与配置块获取

标准通道的创世区块需要基于系统通道的当前配置来生成，因此启动引导过程 分为两个步骤：创建通道配置更新交易、生成通道创世区块：



首先利用configtx.yaml中的配置段生成一个通道创建交易：

configtxgen -outputCreateChannelTx ch1.tx \ *# 生成通道创建交易，保存在*

-profile DefaultChannel \ *# 使用configtx.yaml中的DefaultChannel配置段*

-channelID ch1 *# 通道ID为ch1*

通道创建交易是Protocol Buffer编码的Envelope封装数据，因此我们可以将其 转换为json：

configtxlator proto\_decode \ *# 解码protocol buffer数据*

--input ch1.tx \ *# 输入文件*

--type common.Envelope \ *# 输入数据类型*

--output ch1.json *# 输出文件*

一旦获得了通道配置更新交易，就可以将其提交给排序器来生成更新后的通道配置块 作为通道的创世区块：

peer channel create \ *# 创建通道创世区块*

-f ch1.tx \ *# 通道配置更新交易文件*

-c ch1 \ *# 通道ID为ch1*

-o localhost:7050 \ *# 使用本地排序器*

--outputBlock ch1.pb *# 输出创世区块文件*

当该命令结束后，在排序器上就已经启动了标准通道。利用这个返回的创世区块文件， 对等节点就可以加入通道了：

peer channel join \ *# 将当前对等节点加入通道*

-b ch1.pb \ *# 通道创世区块*

-o localhost:7050 *# 使用本地排序节点*

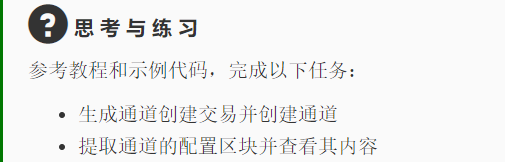
**提取通道配置区块**

和系统通道一样，可以从排序节点提取标准通道的配置区块，但由于在对等节点 上也保存有区块链数据，因此也可以直接从对等节点提取区块配置数据，例如：

peer channel fetch config config-block.pb \ *# 提取通道配置区块*

-c ch1 *# 通道ID为ch1*

（注意!示例代码没有启动peer节点，所以仍然是从orderer节点提取区块配置，需要加上-o localhost:7050 从orderer节点获取）



注意，示例代码有误，与上一节内容对比，并进行以下修改：

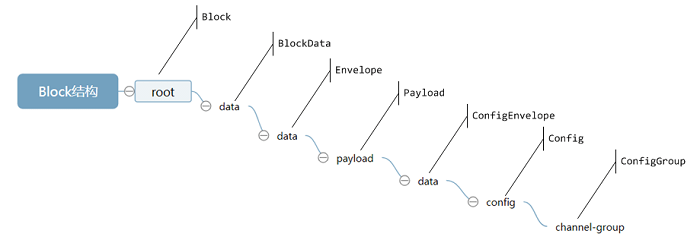
通道ID得改成: ch1。

CORE\_PEER\_MSPCONFIGPATH=./crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp \

peer channel fetch config config-block.pb -o localhost:7050 -c ch1

### 深入通道配置区块

通道配置区块中保存的不是交易，而是通道配置信息。不过Fabric的区块结构 嵌套之多极其令人恼火，我们需要跋涉四、五层才能访问到真正的配置数据 config，在Farbric实现中它对应的类型为common.Config结构：



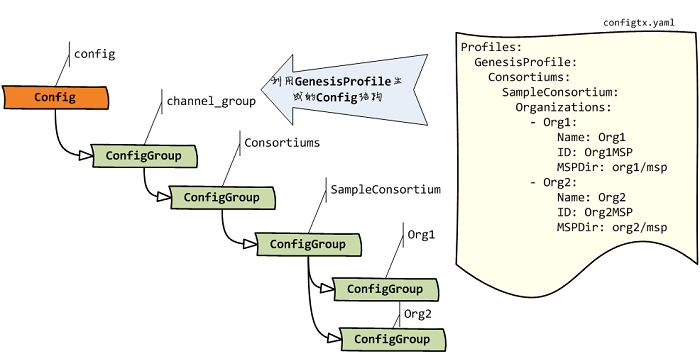
因此如果我们通过peer channel fetch config命令得到了一个配置区块并将它转化为json文件，例如config-block.json，那么可以使用jq来获得剥掉外层封装的配置数据，并存入config.json文件以便查看：

**jq** .data.data[0].payload.data.config \ # 配置数据路径

**config-block**.json \ # 输入**JSON**文件

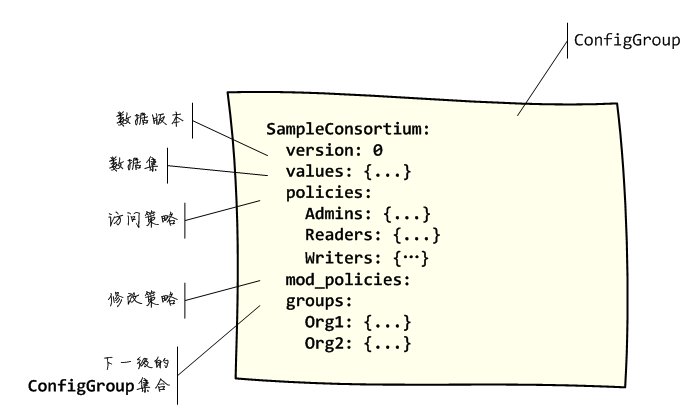
> **config**.json # 重定向输出到文件

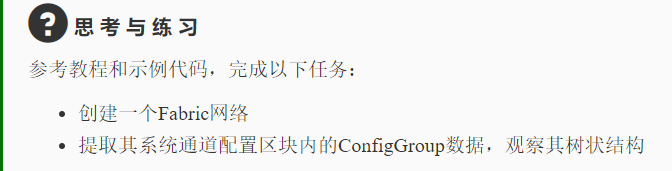
Config中的channel\_group利用configtx.yaml中的信息生成的，它将配置文件中的所有信息组织为ConfigGroup节点树，channel\_group为树根：



因此在配置区块中的配置信息虽然源于configtx.yaml，但其组织结构是有别 于原始的配置文件的。Fabric之所以将配置信息组织为树状同质节点，一个很重要的原因是进行权限控制策略的解析，例如，父级ConfigGroup节点可以基于后代ConfigGroup节点的策略组合出更灵活的权限控制方案。

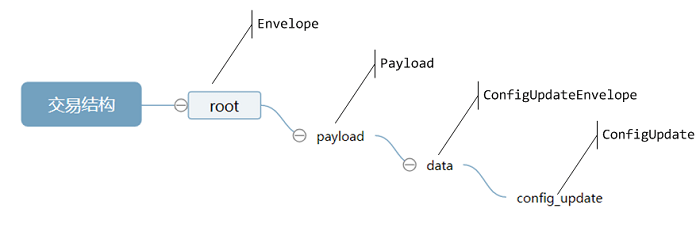
每个ConfigGroup节点都包含五个部分的内容：





### 通道配置更新交易

通道配置更新交易对应于Envelope类型，其层级结构如下：



创建标准通道时的交易就是一个配置更新交易，将其应用在原有的配置上， 就得到了更新后的完整配置，因此也可以认为ConfigUpdate数据是希望得到 的新的配置数据和现有配置数据之间的差异。当我们拿到了当前配置config.pb 和修改后的配置modified.pb后，可以使用configtxlator得到一个ConfigUpdate 数据。例如：

configtxlator compute\_update \ *# 计算配置更新数据*

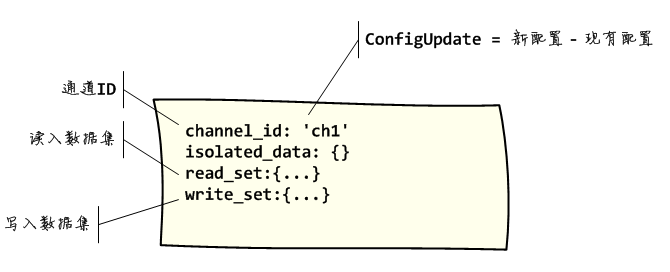
--original config.pb \ *# 初始配置数据*

--updated modified \ *# 修改后的配置数据*

--channel\_id ch1 \ *# 通道ID*

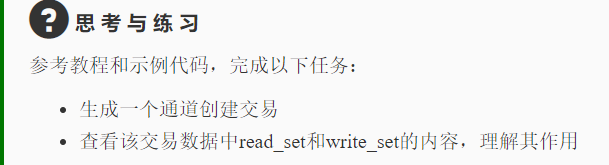
--output update.pb *# 输出文件*

ConfigUpdate结构的主要内容为read\_set和write\_set，分别表示读入数据集 和写入数据集：



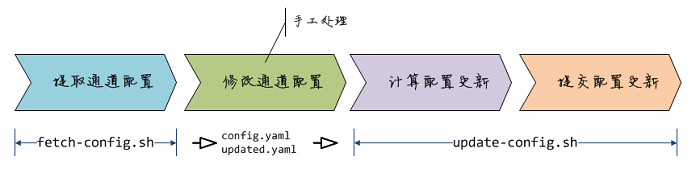
要更新的数据同时包含在read\_set和write\_set中，更新的数据包含在write\_set 中，带有新的版本号，而被更新数据的现有值及现有版本则包含在read\_set中。

使用read\_set的目的是避免版本冲突。当排序节点收到配置更新交易后，首先检查 read\_set中的值及版本号与链上配置是否一致，只有一致的情况下，才会应用write\_set 中的新值进行更新并提升版本号。



### 通道配置更新流水线

容易理解，当我们需要修改通道配置时，基本是按照如下的流水线进行：



提取通道配置环节从节点读取通道的配置区块，并从中提取Config数据，这部分可以 使用脚本自动完成；修改通道配置环节通常需要手工处理来产生新的Config数据，然后 计算新配置与现有配置之间的差异并提交给排序节点处理 —— 最后两个环节也是容易 脚本化的。

fetch-config.sh脚本的输出是通道当前配置config.yaml以及其副本updated.yaml， 期望的修改应当在updated.yaml上完成。update-config.sh脚本的输入则是原始的 config.yaml和修改后的updated.yaml，它封装了相当繁琐的更新计算、编码转换、 签名等操作。在理解了下面的内容之后，你可以直接使用这两个脚本来执行通道 配置更新工作。

**提取通道配置**

读取通道配置区块，转换为JSON数据并提取其中的Config数据：

# **fetch-config**.sh

**peer** **channel** **fetch** **config** **config-block** **-c** **ch1**

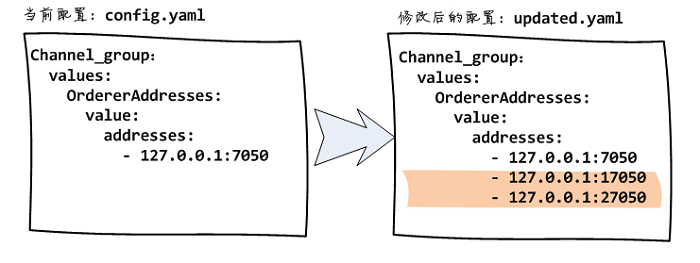
**configtxlator** **proto\_decode** **--input** **config-block**.pb **--type** **common**.Block > **config-block**.json

**jq** .data.data[0].payload.data.config **config-block**.json > **config**.json

**json2yaml** **config**.json > **config**.yaml

**修改通道配置**

修改通道配置的工作通常需要手工完成。例如，假设我们希望为通道增加几个新的排序节点：



**计算配置更新**

计算配置更新主要是使用configtxlator的compute\_update子命令。不过首先需要 将新旧两份配置文件转换为Protocol Buffer格式：

**yaml2json** **config**.yaml > **config**.json

**configtxlator** **proto\_encode** **--input** **config**.json **--type** **common**.Config > **config**.pb

**yaml2json** **udpated**.yaml > **updated**.json

**configtxlator** **proto\_encode** **--input** **updated**.json **--type** **common**.Config > **updated**.pb

**然后计算更新：**

configtxlator compute\_update \ *# 计算配置更新数据*

--original config.pb \ *# 原始配置数据*

--updated updated.pb \ *# 更新后的配置数据*

--channel\_id ch1 \ *# 要更新配置的通道ID*

--output update.pb *# 输出配置更新文件*

**提交配置更新**

向排序节点提交配置更新主要是利用peer channel update命令，但该命令要求将 配置更新封装为Envelope格式，因此我们还需要将获取的Protocol Buffer格式的 配置更新数据转化为JSON格式，添加到Envelope结构中之后，再转换回Protocol Buffer 格式。

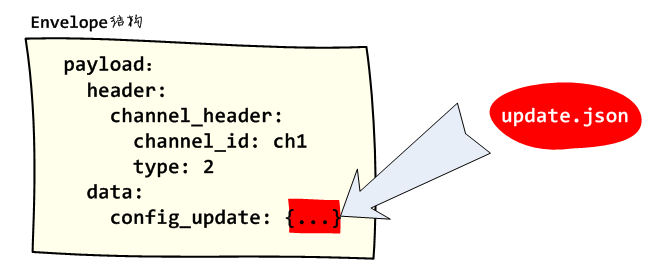
首先将update.pb转换为json格式：

**configtxlator** **proto\_decode** **--input** **update**.pb \ # 解码**update**.pb

**--type** **common**.ConfigUpdate \ # 输入数据类型为**ConfigUpdate**

**--output** **update**.json # 输出**JSON**保存到**update**.json

然后为配置更新数据添加Envelope封装：



可以使用如下的命令完成：

echo '{

"payload":{

"header":{

"channel\_header":{

"channel\_id":"ch1",

"type":2

}

},

"data":{

"config\_update":'$(cat update.json)'

}

}

}' > update-envelope.json

将得到的update-envelope.json转化回Protocol Buffer格式：

configtxlator proto\_encode --input update-envelope.json \ *# 编码json数据*

--type common.Envelope \ *# 数据格式为Envelope*

--output update-envelope.pb *# 输出保存文件*

现在我们就可以签名，然后提交给排序器了：

**peer** **channel** **signconfigtx** **-f** **update-envelope**.pb

**peer** **channel** **update** **-f** **update-envelope**.pb **-c** **ch1** **-o** **localhost**:7050

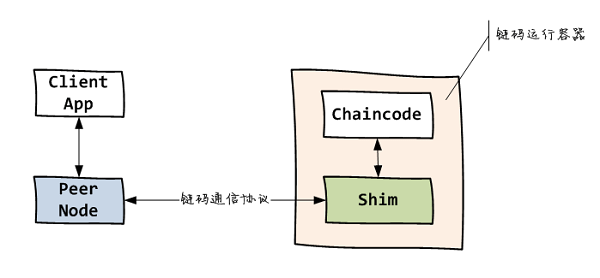


（注意：按照示例教程执行到./fetch-config.sh会出错，是因为从peer节点获取通道配置文件会失败，原因未知。所以加上-o localhost:7050，改成从orderer节点获取通道配置文件）

## 链码与应用开发阶段

### 链码运行机制概述

在Fabric中，链码（生产环境下）（HBJ:对应的是开发环境）是运行在容器中的，对等节点通过shim层与链码完成双向的通信：



shim层封装了与对等节点通信的细节，使得链码的开发者可以聚焦于业务 逻辑的实现。不过了解节点与链码的通信机制，对链码开发是有帮助的。

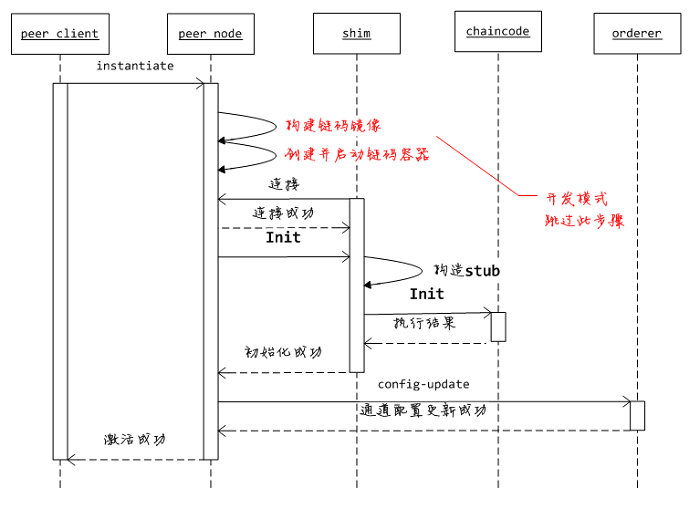
**链码的安装**

当执行peer chaincode install命令安装链码时，节点所做的工作就是将源代码文件打包保存到节点文件系统数据目录的chaincodes文件夹，并以链码名称 和版本号进行标识和区别。因此当你尝试再次安装同一个链码而没有提升版本号时，将提示该链码已经安装。

**链码的激活**

链码的安装是节点级别的，而链码的激活则是通道级别的 —— 需要在不同的节点安装链码，但是不需要在不同的节点激活链码。

当执行peer chaincode instantiate命令激活链码时，参与的各部分时序如下：



peer节点收到激活命令后，首先**构建链码的镜像**，包括源代码编译、依赖安装、 docker镜像构建等过程，然后**创建并启动链码容器**，等待链码的连接。

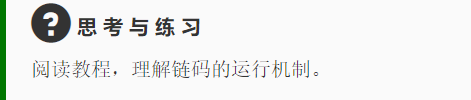
一旦链码容器启动，shim层会主动连接peer节点，在连接成功后即建立了链码与peer节点的双向连接。

peer节点在与链码的双向连接建立后，立刻调用链码的Init方法进行链码初始化， 并在初始化成功后向排序器提交**通道配置更新请求**，在得到排序节点的更新成功响应之后，链码激活过程结束。

当peer节点以开发模式启动时，将跳过链码镜像的构建和容器启动步骤，因此开发者需要自行启动链码以建立与peer节点的双向连接。

**链码的调用**

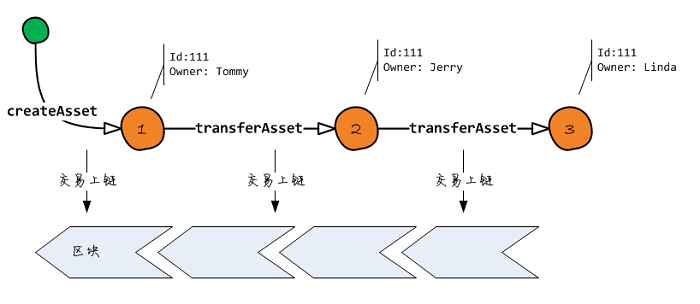
如前所述，对链码的调用是通过shim中转的，shim实现了Fabric的链码通信协议， 同时封装了对Peer节点的账本操作细节，这使得链码可以利用shim传入的stub完成状态更新等操作。



### 跟踪状态变迁历史

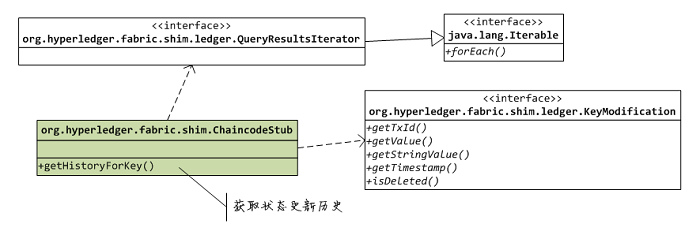
和其他区块链类似，Farbic的状态记录也是有版本的，这意味者所有的状态 更新历史都可以进行追溯，这一点在很多应用中非常有帮助，例如资产跟踪、 农产品溯源等等。

假设要实现一个链码来跟踪资产的所有权转移情况，那么我们即需要状态历史， 同时也需要发生状态迁移的时间，以及引发状态迁移的交易：



在上图中，当最初的createAsset动作创建资产111后，该资产的初始所有者 为Tommy，这一动作对应的交易也写入区块；当执行transferAsset动作将资产 111转移给Jerry后，资产状态发生的迁移，同时也向区块链写入一条新的交易。

Fabric-shim的ChaincodeStub提供了getHistoryForKey()方法来获取上述信息：



例如，下面的代码用来获取状态历史：

**private** Response **getAssetHistory**(ChaincodeStub stub,List<String> args){

String id = args.**get**(0);

QueryResultsIterator<KeyModification> modifications = stub.getHistoryForKey(id);

**for**(KeyModification m: modifications){

System.**out**.format("txid: %s\n", m.getTxId()); *//交易ID*

System.**out**.format("value: %s\n",m.getStringValue()); *//状态字符串*

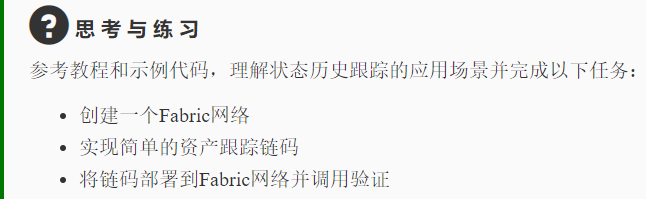
System.**out**.format("timestamp: %s\n",m.getTimestampe().toString()); *//时间戳*

System.**out**.format("deleted: %s\n", Boolean.valueOf(m.isDeleted())); *//是否已删除*

}

**return** newSuccessResponse();

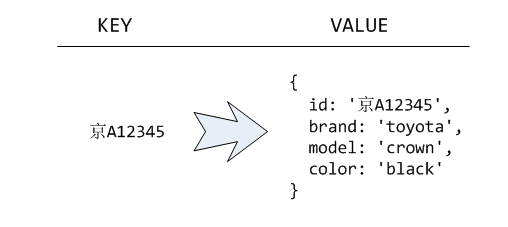
}



### 使用复合键

Fabric的对等节点默认使用leveldb保存状态数据。leveldb是一个简单的Key-Value 数据库，这意味着你只能使用Key来查询状态数据，在很多情况下这不能满足应用的 需求。

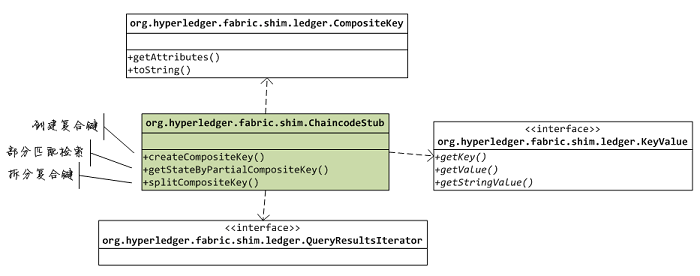
例如，假设你开发了一个基于区块链的二手车交易平台，每辆二手车都记录其车牌号、 品牌、车型、颜色等信息。显然用车牌号作为键来存储状态是合理的，例如：



但是考虑一下，人们并不总是用车牌号来查找二手车信息，更有可能的是，一个想 买二手车的潜在用户，会搜索指定品牌的二手车，也就是说，需要使用数据中的非 KEY字段来检索状态数据。

复合键是解决这一问题的一种方案。类似于关系数据库中的复合键，在Fabric中可以 使用多个字段的值建立一个复合键，例如使用非KEY字段和KEY字段建立一个复合键， 然后利用这个复合键完成从非KEY字段到KEY字段再到状态数据的检索过程。

Farbric-shim中的ChaincodeStub提供了与复合键相关的几个方法：



让我们看看如何使用复合键解决二手车按品牌查询的问题。

首先在登记二手车信息时，需要创建一个由品牌和车牌号组成的复合键：

**private** Response **registerCar**(ChaincodeStub stub,List<String> args){

*//省略代码，构造车辆信息状态字符串carStr*

stub.putStringState(args.**get**(0),carStr); **# 记录车辆状态**

CompositeKey key = stub.createCompositeKey("brand->id",args.**get**(1),args.**get**(0));

stub.putStringState(key.toString(),"\u0000"); **# 记录复合键，值不重要**

**return** newSuccessResponse(String.format("car registered: %s\n",args.**get**(0)));

}

利用getStateByPartialCompositeKey()方法就可以进行复合键的查询，它返回 一个状态数据迭代器：

**private** Response **getCarsByBrand**(ChaincodeStub stub,List<String> args){

String brand = args.**get**(0);

CompositeKey key = stub.createCompositeKey("brand->id",brand); **# 用部分字段构造复合键**

QueryResultsIterator<KeyValue> iterator = stub.getStateByPartialCompositeKey(key.toString());

**for**(KeyValue kv: iterator){

CompositeKey fkey = stub.splitCompositeKey(kv.getKey()); **# 拆分组成复合键的各属性**

String id = fkey.getAttributes().**get**(1); **# 提取复合键中的id值**

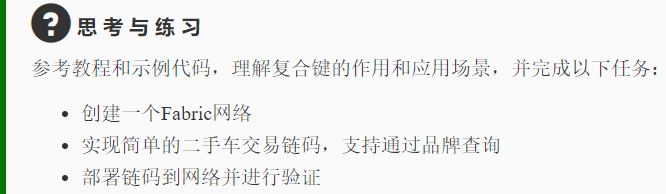
String carStr = stub.getStringState(id); **# 利用id读取对应状态**

System.**out**.format("car %s: %s\n", id,carStr);

} **return** newSuccessResponse();

}

splitComposite()将指定的复合键拆分并返回一个对象，其objectType字段为 复合键的类型，例如我们设定的brand->id，attributes字段为组成复合键的 各字段值，在上面的案例中，第1个字段为品牌名，第二个字段为车牌号，因此 我们可以使用attributes[1]来进一步提取车辆的状态数据了。



### 使用CouchDB状态库

（注意！！！本地安装CouchDB失败，回头再试，参考链接：

<http://docs.couchdb.org/en/stable/fauxton/install.html>）

虽然使用复合键可以解决一部分查询问题，但对于查询要求更高的应用而言，复合 键的能力还是有些捉襟见肘。在这种情况下，可以让peer节点启用CouchDB代替levelDB 作为状态数据库：



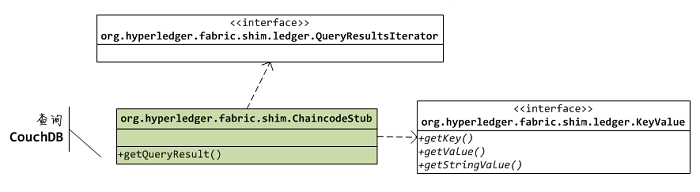
CouchDB是一个功能强大的类似于MongoDB的文档性数据库。 与leveldb不同，CouchDB是一个进程外数据库，因此在启动节点时，需要配置对应的CouchDB访问URL，并设置节点状态库为CouchDB。例如：

CORE\_LEDGER\_STATE\_STATEDATABASE=CouchDB \ # 启用CouchDB作为状态数据库

CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS=127.0.0.1:5984 \ # CouchDB的REST API地址

peer node start

当状态库引擎切换到CouchDB以后，Fabric-shim的ChaincodeStub就可以使用CouchDB 的强大查询语法来检索状态了：



例如，我们使用CouchDB来改写前一节的二手车查询。首先我们在登记车辆时取消掉复合键的使用。然后在按品牌查询车辆时，直接透过getQueryResult()向CouchDB传递原生查询语句：

private Response getCarsByBrand(ChaincodeStub stub,List<String> args){

String brand = args.get(0);

String qs = String.format("{ \"selector\": { \"brand\": \"%s\"} }",brand);

QueryResultsIterator<KeyValue> kvs = stub.getQueryResult(qs);

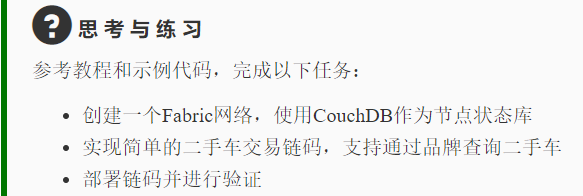
**for**(KeyValue kv: kvs){

System.out.format("%s: %s\n",kv.getKey(),kv.getStringValue());

}

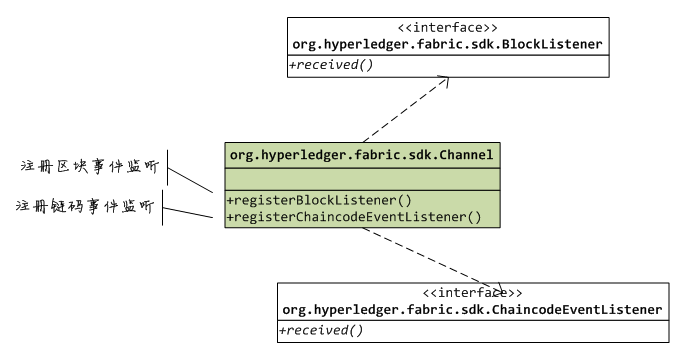
**return** newSuccessResponse();

}



### 监听通道事件

Fabric的通道在生成新的区块或交易时，其通道事件服务可以向应用提供实时的通知，如果应用需要实时跟踪新的区块或交易，例如实现一个区块浏览器， 就可以利用Channel类提供的方法来监听事件：



**通道事件监听**

当通道出新块时得到通知：

channel.registerBlockListener(**new** BlockListener(){

@Override

**public** **void** **received**(BlockEvent **event**){

System.**out**.format("block No.: %d\n",**event**.getBlockNumber());

}

});

**链码事件监听器**

在链码中可以使用stub对象的setEvent()方法触发自定义事件，例如：

stub.setEvent('DemoEvent',ByteString.copyFromUtf8("event payload here").toByteArray());

在应用中可以通过如下的代码监听链码事件：

Pattern pcc = Pattern.compile("wizcc");

Pattern pev = Pattern.compile("OnFire");

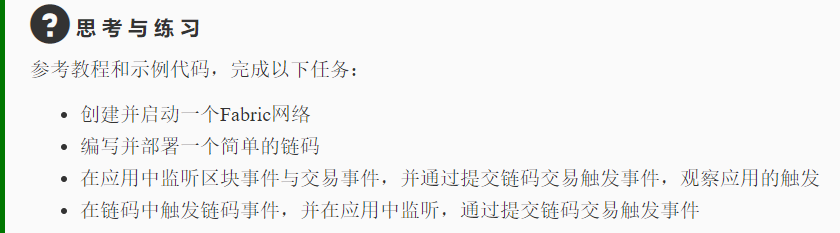
channel.registerChaincodeEventListener(pcc,pev,**new** ChaincodeEventListener(){

**@Override** **public** **void** **received**(String handle,BlockEvent bevt,ChaincodeEvent cevt){

System.out.format("chaincode event payload: %s\n",**new** String(cevt.getPayload()));

}

});

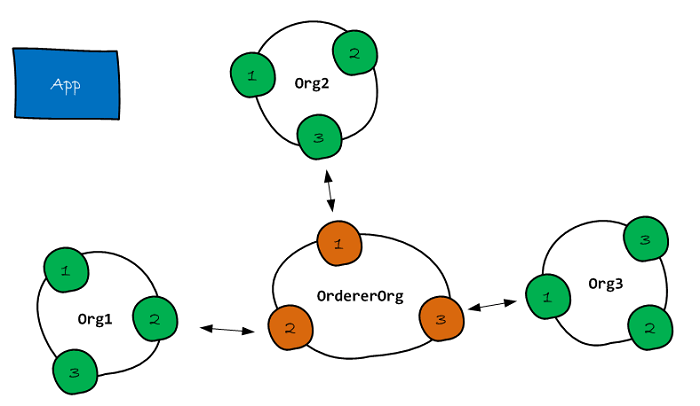


### 使用连接配置文件

在之前的应用开发中，我们都是使用一种命令式（Imperative）的方法来显式地连接排序节点和对等节点，在网络规模较小时这种方法没有问题，但是当网络规模逐渐扩大之后，这种命令式的连接方法就有些缺点了：当网络发生变化时，你需要修改源代码才能进行匹配。

这就是连接配置文件（Connection Profile）的作用，它以声明式（Declaretive） 的方法将网络结构从源代码逻辑中抽取出来，从而增加了源代码的灵活性。

例如，考虑下面的网络结构，其中包含了三个机构，每个机构分别包含三个 对等节点，以及三个排序节点：



如果网络要求交易的背书策略是三个机构成员的大多数（超过半数）签名， 这意味着App需要连接至少两个机构的对等节点进行交易提议，那么容易理解 App使用连接配置文件会比在代码中手写容易的多了。

可以使用JSON或YAML来编写连接配置文件，只需要最终转化为JSON对象即可。 例如，下面的YAML给出了一个单机构的简单网络的连接配置文件：

version: '1.0' *# 连接配置文件版本号*

name: test *# 配置文件名称*

client: *# 客户端信息*

organization: Org1 *# 所属机构*

channels: *# 通道连接配置信息*

ch1: *# 通道ch1*

orderers: *# 排序节点清单*

- orderer1 *# 排序节点引用名，参见orderers配置*

peers: *# 对等节点清单*

peer1: *# 对等节点引用名，参见peers配置*

endorsingPeer: true *# 该节点是否可背书*

chaincodeQuery: true *# 该节点是否可执行链码查询*

ledgerQuery: true *# 该节点是否可执行账本查询*

eventSource: true *# 该节点是否为事件源*

organizations: *# 机构定义*

Org1: *# 机构Org1*

mspid: Org1MSP *# 机构MSPID*

peers: *# 机构对等节点清单*

- peer1 *# 对等节点引用名，参见peers配置*

orderers: *# 排序节点配置*

orderer1: *# 排序节点orderer1配置*

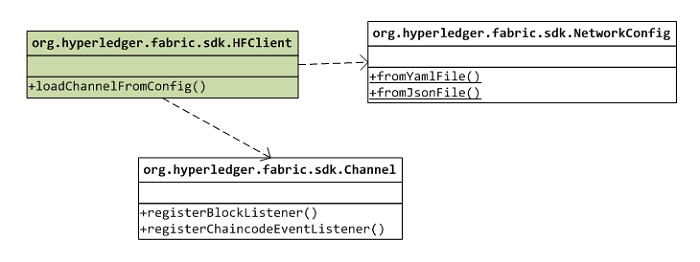
url: grpc://127.0.0.1:7050 *# 排序节点连接URL*

peers: *# 对等节点配置*

peer1: *# 对等节点peer1配置*

url: grpc://127.0.0.1:7051 *# 对等节点连接URL*

一旦定义了连接配置文件，就可以利用它创建一个NetworkConfig对象，然后利用 这个配置对象获取指定的通道对象：



例如，假设上面的内容保存为profile.yaml文件，那么下面的代码将使用该文件 创建一个Channel实例：

NetworkConfig nc = NetworkConfig.fromYamlFile(new File("./profile.yaml"));

Channel channel = client.loadChannelFromConfig("ch1",nc);

由于在连接配置文件中已经声明了网络的关键信息，因此不需要再显式的添加节点了。通道对象会根据连接配置文件中ch1的配置信息找到peer1节点 ，自动进行连接并执行链码查询。注意配置文件中peer1的能力chaincodeQuery 为true，这意味着当Client实例可以从一组节点清单中进行筛选，自动选中合适的节点。

