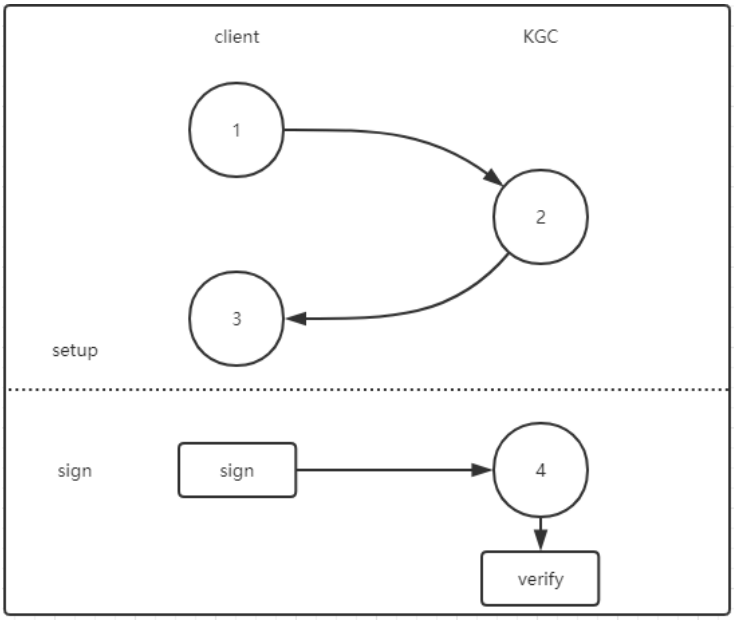
Fabric-ibpcla 版本详细说明文档

Fabric-v1.4 支持的 msp type 有默认的 bccsp(使用 x509 证书体系)和 idemix(使用匿名证书)，ibpcla 版本在此基础上添加了新的 msp type：ibpcla(基于身份的无证书签名体系)

Ibpcla 简介

Ibpcla 是无证书无双线性对密码体制，说是无证书，其实是用较轻量级的部分公钥来代替证书。
在注册阶段，用户将身份信息和初始化参数发给 KGC，KGC 生成部分公钥和部分私钥返还给用户，用户生成最终私钥。并保存部分公钥
在使用阶段(签名验签)，使用私钥正常签名，将签名和身份信息和部分公钥发送给验签方，验签方通过副标识和身份信息恢复出真实公钥，使用公钥进行验签
ibpcla 简易逻辑图：



对 fabric 的改动

当前生成 msp 的方式有两种：使用 fabric-ca-server/client 和使用本地 cryptogen
Ibpcla 版本对两种方法都提供支持，对 fabric-ca 的改动可参考 ca 部分的说明文档。
以下是本地工具的方式：
使用 clgen 生成 ibpcla 相关的 msp 结构，源码位置 Common/tools/clgen，修改 Makefile 添加 clgen 相关。
目标是把①②③④做到 fabric 里面，CA→KGC，目前，cryptogen 方式下每个 org 有两个根 CA：signCA 和 TLSCA，在 main.go 中调用 ca.NewCA 接口创建 CA 并生成自签证书。

思路是只改动 signCA→KGC，用主公钥 P 代替自签证书，用部分公钥 PA 代替签发的证书，TLSCA 保持不变。具体的：

1. 将原有 cryptogen 复制一份成 clgen，尽量保持原接口，增加 kgc 文件夹提供创建 KGC 接口 NewKGC 及生成部分公钥的接口 KGCGenPartialKey（图中②）。
2. 保留原 ca 文件夹来创建 tlsca 及证书。
3. 在原 csp/csp.go 里添加新接口 KGCGeneratePrivateKey 和 KGCGeneratePublicKey 供 kgc 的 NewKGC 调用，并添加 GenFinalKeyPair(图中③)用来生成最终用户公私钥。
4. 在原 msp/generator.go 里修改 GenerateVerifyingMSP 和 GenerateLocalMSP 接口：

   a) GenerateVerifyingMSP 修改前是拉取当前 org 的 signCA 和 TLSCA 的自签名证书分别保存到 msp/cacerts/ 和 msp/tlscacerts/ 然后用 signCA 的自签证书签发临时的 admin-cert(单元测试用，后面会被覆盖掉)。修改后，拉取 TLSCA 的自签证书不变，拉取 signCA 改为拉取 KGC 的主公钥 P 并保存到 msp/kgcpubs。签发 admin-cert 改为调用 kgc 的 KGCGenPartialKey 接口签发部分公钥 PA。

   b) GenerateLocalMSP 修改前是创建当前 org 下各 peer 或 orderer 以及 users 的 localMSP 结构，如图：

```
orderers
└── orderer.example.com
    ├── msp
    │   ├── admincerts
    │   │   └── Admin@example.com-cert.pem
    │   ├── cacerts
    │   │   └── ca.example.com-cert.pem
    │   ├── keystore
    │   │   └── db26e00bb268723532e7f2a65f42c66fd2777aabebe7498b47975d7525a2b566_sk
    │   ├── signcerts
    │   │   └── orderer.example.com-cert.pem
    │   └── tlscacerts
    │       └── tlsca.example.com-cert.pem
    └── tls
        ├── ca.crt
        ├── server.crt
        └── server.key
```

修改后，CLID 放身份信息，其中 adminConfig 是管理员信息，IDconfig 是本地信息，kgcpubs 放 KGC 的主公钥 P，tls 相关保持不变。如图：

```
orderers
└── orderer.example.com
    ├── msp
    │   ├── CLID
    │   │   ├── adminconfig
    │   │   └── IDconfig
    │   ├── config.yaml
    │   ├── kgcpubs
    │   │   └── kgc.example.com-pubkey
    │   └── tlscacerts
    │       └── tlsca.example.com-cert.pem
    └── tls
        ├── ca.crt
        ├── server.crt
        └── server.key
```

IDconfig 包含了 PA，私钥 Sk，组织 OU，角色 Role，身份标识 id：


```
{"Pk":"MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEMXP7ZCbDZlAO/FLnPteVP6iIdN2VIjU+bxYaqPnVN91F6SsseWW/f/7hIBboICh5ujJQL/PkhwT8fXkeRxxDfA==","sk":"MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQg5YVFcW9yE7qP+KBU6n5Ne9AlzCVQud4sKmc1ed8KndOhRANCAAQoXs2eI0B3PA1WBpD5KZAeIs1PAvK3NbkIWfS6qZjCZ4icqS9Gcdliv38LoPMvtZCyVBd8jrtl/Y+TeoSVFf9K","serial":"V3OulsVEhHp4xL7VUPtrwzHN+MpwEexB3WbsYrkhrIs=","organizational_unit_identifier":"example.com","role":"CLIENT","enrollment_id":"orderer.example.com"}
```


adminConfig 包含了管理员的 PA，OU, Role, ID:

{"Pk":"MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEO10DWP7+HAgWL/cJaTEvv76ux8/gj5add+yQ0WTt22mWzONEAt6Fa4PKRUpOSKJMvVWph/zwyVwAcBMZ57PYag==","organizational_unit_identifier":"example.com","role":"ADMIN","enrollment_id":"Admin@example.com"}

5. 创世块 configtxgen 里最终提取 crypto-config 的 msp 配置到了 Msp/configbuilder.go 里的 GetVerifyingMspConfig，目前 MSPType 有 FABRIC 和 IDEMIX，在这里添加 IBPCLA 分支，并添加对应的函数 GetCLMspConfig。使用时在 configtx.yaml 里注明 MSPType:ibpcla。使用 configtxgen 生成创世块时，创世块存放了各 org 的证书，最后追踪到在 encoder/encoder.go 里的 NewOrdererOrgGroup 和 NewApplicationOrgGroup

调用了 msp.GetVerifyingMspConfig 把 msp 验证配置(无私钥)写到创世区块里，

GetVerifyingMspConfig 在 msp/configbuilder.go 里:

```
// GetVerifyingMspConfig returns an MSP config given directory, ID and type
func GetVerifyingMspConfig(dir, ID, mspType string) (*msp.MSPConfig, error) {
        switch mspType {
        case ProviderTypeToString(FABRIC):
                return getMspConfig(dir, ID, nil)
        case ProviderTypeToString(IDEMIX):
                return GetIdemixMspConfig(dir, ID)
        default:
                return nil, errors.Errorf("unknown MSP type '%s'", mspType)
        }
}
```

根据配置文件 configtx.yaml 里的 MSPType 字段区分：默认 FABRIC，可选是 IDEMIX，修改 msp/configbuilder.go 添加 CL 的 case，添加函数 GetCLMspConfig()，该函数的功能是读取本地 crypto-config 文件结构，解析所需文件(IDconfig 等)并封装成 protos/msp/msp_config.proto 所定义的 CLMspConfig 结构。

6. MSP 的初始化配置，peer 端，在 peer/node/start.go 的 serve()的最开始就是 mspType := mgmt.GetLocalMSP().GetType(),其中在 msp/mgmt./mgmt.go 中的 GetLocalMSP()函数，进而调用 loadLocaMSP()，在其中首先读取 mspType，根据 mspType 生成配置 mspOpts，进而调用 msp.New 进行初始化的配置。

```
func loadLocaMSP() msp.MSP {
        // determine the type of MSP (by default, we'll use bccspMSP)
        mspType := viper.GetString("peer.localMspType")
        if mspType == "" {
                mspType = msp.ProviderTypeToString(msp.FABRIC)
        }

        var mspOpts = map[string]msp.NewOpts{
                msp.ProviderTypeToString(msp.FABRIC): &msp.BCCSPNewOpts{NewBaseOpts: msp.NewBaseOpts{Version: msp.MSPv1_0}},
                msp.ProviderTypeToString(msp.IDEMIX): &msp.IdemixNewOpts{NewBaseOpts: msp.NewBaseOpts{Version: msp.MSPv1_1}},
        }
        newOpts, found := mspOpts[mspType]
        if !found {
                mspLogger.Panicf("msp type " + mspType + " unknown")
        }

        mspInst, err := msp.New(newOpts)
        if err != nil {
                mspLogger.Fatalf("Failed to initialize local MSP, received err %+v", err)
        }
        switch mspType {
        case msp.ProviderTypeToString(msp.FABRIC):
                mspInst, err = cache.New(mspInst)
                if err != nil {
                        mspLogger.Fatalf("Failed to initialize local MSP, received err %+v", err)
                }
        case msp.ProviderTypeToString(msp.IDEMIX):
                // Do nothing
```

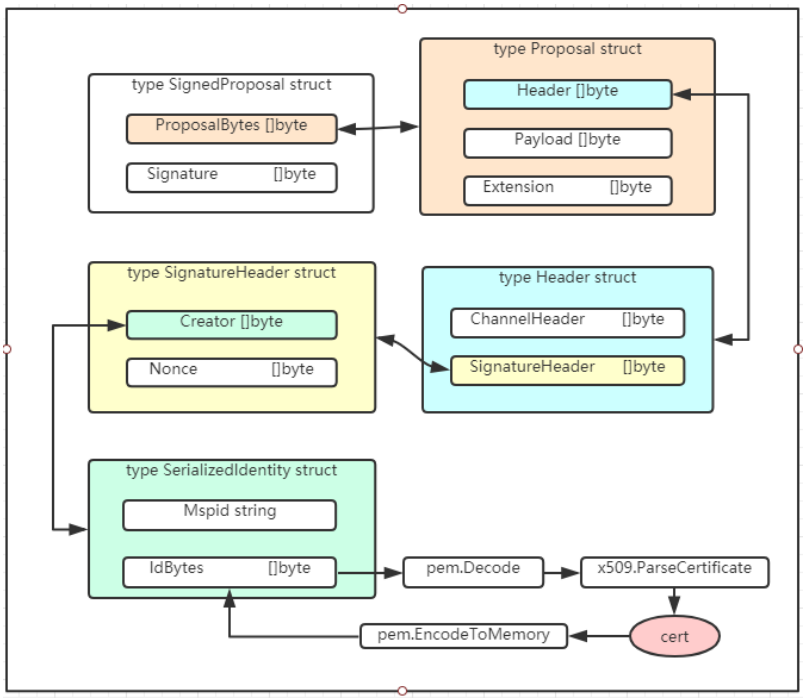
New 函数在 msp/factory.go 中定义，根据 opts 选择 bsccspMsp 分支或者 IDEMIX 分支，这里我们添加第三个分支 IBPCLAMsp，新建 clmsp.go，在里面实现 newIBPCLAMsp 函数，同时实现 CL 版的 MSP 接口的所有方法

```go
// New create a new MSP instance depending on the passed Opts
func New(opts NewOpts) (MSP, error) {
	switch opts.(type) {
	case *BCCSPNewOpts:
		switch opts.GetVersion() {
		case MSPv1_0:
			return newBccspMsp(MSPv1_0)
		case MSPv1_1:
			return newBccspMsp(MSPv1_1)
		case MSPv1_3:
			return newBccspMsp(MSPv1_3)
		default:
			return nil, errors.Errorf("Invalid *BCCSPNewOpts. Version not recognized [%v]", opts.GetVersion())
		}
	case *IdemixNewOpts:
		switch opts.GetVersion() {
		case MSPv1_3:
			return newIdemixMsp(MSPv1_3)
		case MSPv1_1:
			return newIdemixMsp(MSPv1_1)
		default:
			return nil, errors.Errorf("Invalid *IdemixNewOpts. Version not recognized [%v]", opts.GetVersion())
		}
	case *IBPCLANewOpts:
		switch opts.GetVersion() {
		case MSPv1_3:
			return newIBPCLAMsp(MSPv1_3)
		default:
			return nil, errors.Errorf("Invalid *IBPCLAOpts. Version not recognized [%v]", opts.GetVersion())
		}

	default:
		return nil, errors.Errorf("Invalid msp.NewOpts instance. It must be either *BCCSPNewOpts or *IdemixNewOpts. It was [%v]", opts)
	}
}
```

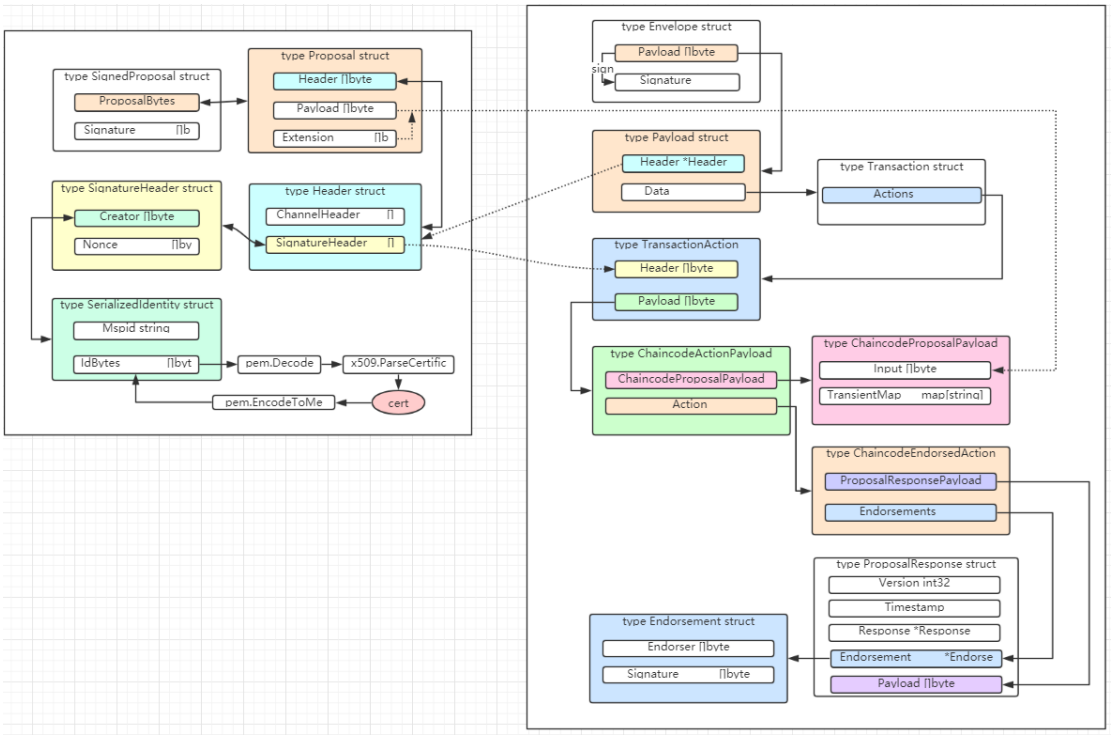
```
MSP : interface
  [embedded]
 +IdentityDeserializer
  [methods]
 +GetDefaultSigningIdentity() : Signin
 +GetIdentifier() : string, error
 +GetSigningIdentity(identifier *Ident
 +GetTLSIntermediateCerts() : [][]byte
 +GetTLSRootCerts() : [][]byte
 +GetType() : ProviderType
 +GetVersion() : MSPVersion
 +SatisfiesPrincipal(id Identity, prin
 +Setup(config *msp.MSPConfig) : error
 +Validate(id Identity) : error
```

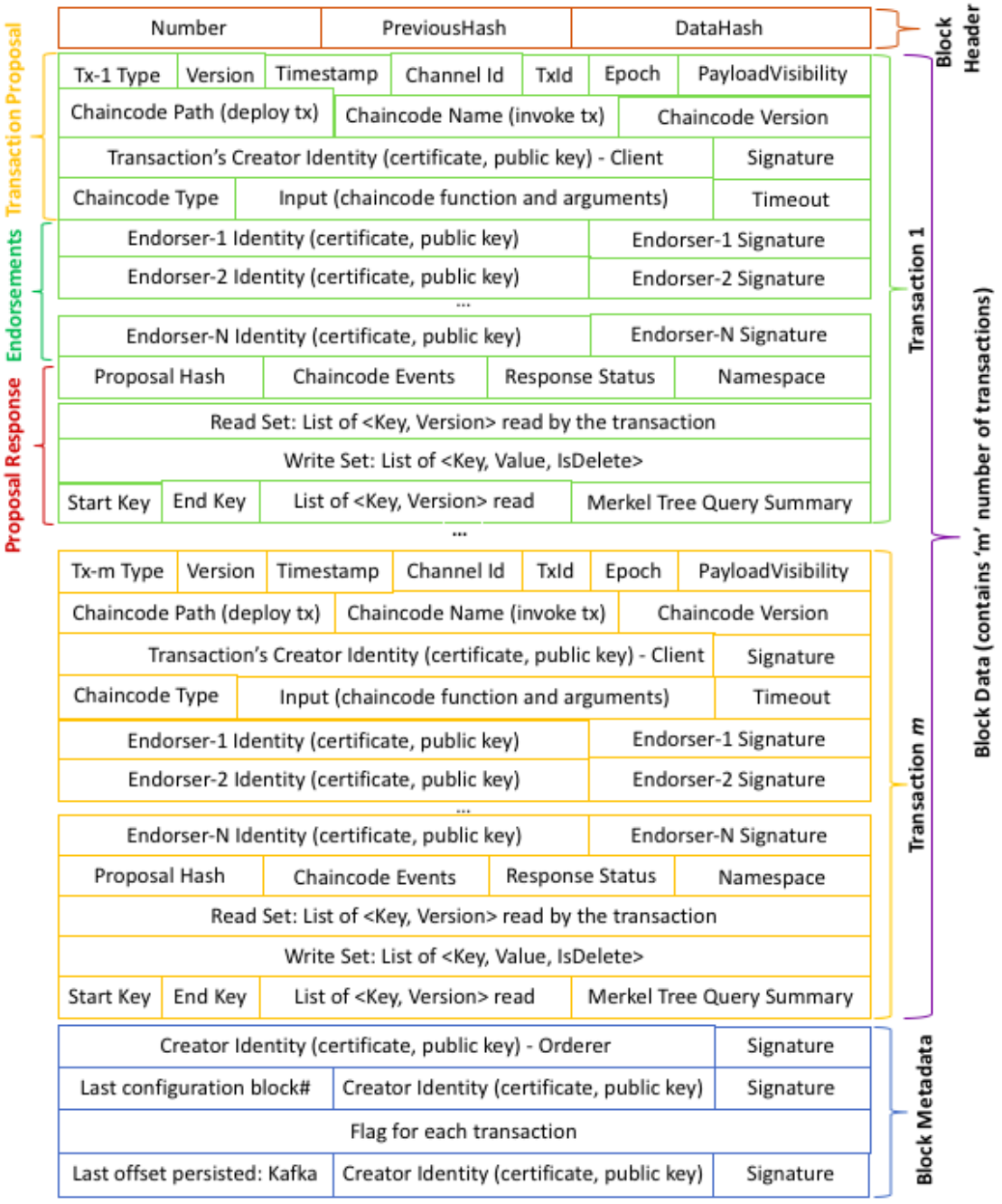
7. MSP 的初始化，MSP 真正的初始化是从 msp/mgmt/mgmt.go 中的 LoadLocalMsp 和 LoadLocalMspWithType 通过调用 Setup 方法而来的。
   Orderer 端初始化 msp 是在 orderer/common/server/main.go 中，在 main()里调用 initializeLocalMsp(conf)，在 initializeLocalMsp 里调用的是 mspmgmt.LoadLocalMsp，其中 mspType := viper.GetString("peer.localMspType")会返回空，因为 orderer 端没初始化 viper，会进入默认的 bccsp 模式，因此在此处添加 viper 获取 core.yaml 的 localMspType 来区分，进入 GetLocalMspConfigWithType，通过配置文件初始化，最终使用的是 setup 方法。
   Peer 端初始化 msp，在 peer/common/common.go 中的 InitCmd 中调用了 InitCrypto，在 InitCrypto 中调用了 LoadLocalMspWithType，最终也使用了 Setup 方法 CL 版的 Setup 方法在 clmsp.go 中实现。
8. 在实现 newIBPCLAMsp 函数时，首先要初始化 bccsp 模块，CLA 比标准验签流程多一步恢复公钥，因此需要修改 bccsp 模块的 verify，将 sw 复制一份成 cl 文件夹，在其中实现 bccsp 接口的 CL 版实现，修改 verifyECDSA 并添加一步恢复公钥接口 RecoverPub。调用 bccsp 的是在 msp 的 identity 的 Verify 方法里，传递的参数之前是 bccsp.Key，在 CLA 中对应的是部分公钥 PA，追踪到是由 msp 接口的 deserializeIdentityInternal 方法解析 x509 证书得到的 pk，在 msp/clmsp.go 中用 cl 的接口实现此方法，将解析证书改为解析 CLA 签名结构，得到 PA，将 PA 放在原 pk 的位置。在排序阶段，orderer 接收到 peer 发来的 envelope 也要验签，具体在 common/cauthdsl/ cauthdsl.go 里调用了 DeserializeIdentity 和 Verify 接口，同样使用 cl 的接口修改实现。
9. 签名消息封装，ProcessProposal 里的参数 signedPropsal 是提案阶段整体的消息封装，根据结构体定义可进一步细分：



在 CLA 里 cert 变成了 PA，我们要改变编解码方式，编码是在 msp/identities.go 下的 Serialize()函数，解码在 msp/mspimpl.go 的 deserializeIdentityInternal 方法。

由 peer 发给 orderer 的 envelope：右侧第三层的 TransactionAction 结构的 Header 是由左侧 SignatureHeader 组成。修改 proposal 的证书结构即可。





10. Gossip 模块，有三部分使用到验签，一是消息在 peer 间散播，从 gossip/gossip/channel/channel.go 的 HandleMessage 里的 gc.verifyBloc 调用了 mcs 里的 VerifyBlock，进而调到 policy.Evaluate 最终在 common/cauthdsl/cauthdsl.go 里 compile 函数使用了 identity.verify，二个是 deliver 阶段从 orderer 拉取区块，core/deliverservice/blocksprovider/blocksprovider.go 的 DeliverBlocks 里也用到了 b.mcs.VerifyBlock，三是 commit 阶段需要 ValidateTx，调用了 validate 和 verify 方法。

文件结构

原先证书 800 字节左右，现在 IDconfig 490 字节

使用

- 编译 image:
  sampleconfig/core.yaml : localMspType: ibpcla
  sampleconfig/core.yaml: MSPType: ibpcla
  Make docker
- 测试:
  make checks

to do:

x509-ibpcla 共存
匿名化、(多 ID、多 PA、多属性)
Benchmark
与支持国密兼容，配置，是否使用 plugin