目錄

configtxgen.rst	1.9.2.35
consensus.rst	1.9.2.36
CONTRIBUTING.rst	1.9.2.37
debug.rst	1.9.2.38
demos.rst	1.9.2.39
docker-2peer.yml	1.9.2.40
docker-compose-channel.yml	1.9.2.41
dockercompose.rst	1.9.2.42
endorsement-policies.rst	1.9.2.43
endorser.rst	1.9.2.44
fabric_model.rst	1.9.2.45
glossary.rst	1.9.2.46
gossip.rst	1.9.2.47
included.rst	1.9.2.48
index.rst	1.9.2.49
install_instantiate.rst	1.9.2.50
javasdk.rst	1.9.2.51
jira_navigation.rst	1.9.2.52
learn_chaincode.rst	1.9.2.53
ledger.rst	1.9.2.54
license.rst	1.9.2.55
MAINTAINERS.rst	1.9.2.56
make.bat	1.9.2.57
marbles.rst	1.9.2.58
mdtorst.sh	1.9.2.59
multichannel.rst	1.9.2.60
nodesdk.rst	1.9.2.61
orderingservice.rst	1.9.2.62
overview.rst	1.9.2.63
paper.rst	1.9.2.64
peer-chaincode-devmode.rst	1.9.2.65
pluggableos.rst	1.9.2.66
policies.rst	1.9.2.67

Hyperledger 源码分析之 Fabric

0.4.6

区块链技术是计算机技术与金融技术交融的成功创新，被认为是极具潜力的分布式账本平台的核心技术。如果你还不了解区块链，可以阅读 区块链技术指南。

作为 Linux 基金会支持的开源分布式账本平台，Hyperledger 受到了众多企业的支持和开源界的关注。本书将试图剖析 Hyperledger 核心的平台实现 Fabric 子项目相关源码，帮助大家深入理解其实现原理。

在线阅读：GitBook 或 GitHub。

欢迎大家加入 区块链技术讨论群。

版本历史

- 0.4.0: 2016-12-12
  - 完成 0.6 分支；
  - 开始 1.0 分支新架构代码。
- 0.3.0: 2016-08-04
  - 完成主要模块内容。
- 0.2.0: 2016-07-01
  - 基本功能分析。
- 0.1.0: 2016-06-08
  - 完成基础框架。

参与贡献

贡献者 名单。

本书源码开源托管在 Github 上，欢迎参与维护：github.com/yeasy/hyperledger_code_fabric。

首先，在 GitHub 上 `fork` 到自己的仓库，如 `docker_user/hyperledger_code_fabric`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/hyperledger_code_fabric.git
$ cd hyperledger_code_fabric
$ git config user.name "yourname"
$ git config user.email "your email"
```

更新内容后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

最后，在 GitHub 网站上提交 pull request 即可。

另外，建议定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/hyperledger_code_fabric
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

本书结构由 *gitbook_gen* 维护。

整体结构

Hyperledger Fabric 在 1.0 中，架构已经解耦为三部分：

- fabric-peer：主要起到 peer 作用，包括 endorser、committer 两种角色；
- fabric-ca：即原先的 membersrvc，独立成一个新的项目。
- fabric-order：起到 order 作用。

其中，fabric-peer 和 fabric-order 代码暂时都在 fabric 项目中，未来可能进一步拆分。

核心代码

fabric 核心源代码目前约为 117K 行，主要包括代码、工具、脚本等部分。

```
$ find fabric -name "*.go" -not -path "fabric/vendor/*" | xargs cat | wc -l
116826
```

源代码

实现 fabric 功能的核心代码，包括：

- accesscontrol 包：实现对 chaincode 的权限管理和属性校验等；
- common 包：一些通用的模块；
- core 包：大部分核心实现代码都在本包下。其它包的代码封装上层接口，最终调用本包内代码；
- events 包：支持 event 框架；
- examples 包：包括一些示例的 chaincode 代码；
- flogging 包：封装 go-logging，提供日志支持；
- gossip 包：实现 gossip 协议；
- metadata 包：版本信息等；
- msp 包：Member Service Provider 包；
- order 包：order 服务相关的入口和框架代码；
- peer 包：peer 的入口和框架代码；
- protos 包：包括各种协议和消息的 protobuf 定义文件；

源码相关工具

一些辅助代码包，包括：

- bddtests：测试包，含有大量 bdd 测试用例；
- gotools：golang 开发相关工具安装；
- vendor 包：管理依赖；

安装部署

包括：

- busybox：busybox 环境，精简的 linux；
- devenv：配置开发环境；
- images：镜像生成模板等。
- scripts：各种安装配置脚本；

其它工具

其他工具，包括：

- docs：文档，大部分文档都可以 在线查阅；

配置、脚本和文档

除了些目录外，还包括一些说明文档、安装需求说明、License 信息文件等。

Docker 相关文件

- .baseimage-release：生成 baseimage 时候的版本号。
- .dockerignore：生成 Docker 镜像时忽略一些目录，包括 .git 目录。

git 相关文件

- .gitattributes：git 代码管理时候的属性文件，带有不同类型文件中换行符的规则，默认都为 linux 格式，即 `\n`。
- .gitignore：git 代码管理时候忽略的文件和目录，包括 build 和 bin 等中间生成路径。
- .gitreview：使用 git review 时候的配置，带有项目的仓库地址信息。
- README.md：项目的说明文件，包括一些有用的链接等。

travis 相关文件

- .travis.yml：travis 配置文件，目前是使用 golang 1.6 编辑，运行了三种测试：unit-test、behave、node-sdk-unit-tests。

其它

- LICENSE：Apache 2 许可文件。
- docker-env.mk：被 Makefile 引用，生成 Docker 镜像时的环节变量。
- Makefile：执行测试、格式检查、安装依赖、生成镜像等操作。
- mkdocs.yml：生成 http://hyperledger-fabric.readthedocs.io 在线文档的配置文件。
- TravisCI_Readme.md

accesscontrol

负责跟访问权限相关的接口和方法，主要包括 crypto 包。

api

authshim.go

attributes

负责 transaction 里的属性。

proto

attributes.pb.go

attributes.proto

test_resources

prek0.dump

tcert.dump

attributes.go

crypto

包括 attr、ecdsa、utils 子包。

- attr：chaincode 属性检查校验；
- ecdsa：ecdsa 加密算法接口，一些 hash 和签名校验方法。
- utils：提供 aes、ecdsa、x509 等方法。

attr

test_resources

attr_support.go

ecdsa

ecdsa.go

hash.go

x509.go

utils

aes.go

ecdsa.go

io.go

keys.go

x509.go

crypto.go

impl

chaincode.go

api.go

bccsp

区块链加密服务提供者（Blockchain Crypto Service Provider），提供一些密码学相关操作的实现，包括 Hash、签名、校验、加解密等。

主要支持 MSP 的相关调用。

factory

提供工厂模式支持，将来包括若干类型的 BCCSP 实现。

- null：空实现，测试用；
- software：软件实现；
- HSM：PKCS11，基于高安全模块的实现。

factory.go

opts.go

pkcs11factory.go

swfactory.go

pkcs11

aes.go

aeskey.go

conf.go

dummyks.go

ecdsa.go

ecdsakey.go

fileks.go

impl.go

pkcs11.go

rsakey.go

signer

signer.go

SW

aes.go

aeskey.go

conf.go

dummyks.go

ecdsa.go

ecdsakey.go

fileks.go

impl.go

rsakey.go

utils

errs.go

io.go

keys.go

slice.go

x509.go

aesopts.go

AES 算法相关选项结构。

bccsp.go

- BCCSP 接口：定义密码学相关操作，包括加解密、签名和验证、签名、Hash、Key 的生命周期管理等方法。

```
type BCCSP interface {

    // KeyGen generates a key using opts.
    KeyGen(opts KeyGenOpts) (k Key, err error)

    // KeyDeriv derives a key from k using opts.
    // The opts argument should be appropriate for the primitive used.
    KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)

    // KeyImport imports a key from its raw representation using opts.
    // The opts argument should be appropriate for the primitive used.
    KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)

    // GetKey returns the key this CSP associates to
    // the Subject Key Identifier ski.
    GetKey(ski []byte) (k Key, err error)

    // Hash hashes messages msg using options opts.
    // If opts is nil, the default hash function will be used.
    Hash(msg []byte, opts HashOpts) (hash []byte, err error)

    // GetHash returns and instance of hash.Hash using options opts.
    // If opts is nil, the default hash function will be returned.
    GetHash(opts HashOpts) (h hash.Hash, err error)

    // Sign signs digest using key k.
    // The opts argument should be appropriate for the algorithm used.
    //
    // Note that when a signature of a hash of a larger message is needed,
    // the caller is responsible for hashing the larger message and passing
    // the hash (as digest).
    Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)

    // Verify verifies signature against key k and digest
    // The opts argument should be appropriate for the algorithm used.
    Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)

    // Encrypt encrypts plaintext using key k.
    // The opts argument should be appropriate for the algorithm used.
    Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error
)

    // Decrypt decrypts ciphertext using key k.
    // The opts argument should be appropriate for the algorithm used.
    Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error
)
}
```

ecdsaopts.go

ECDSA 算法相关选项结构。

hashopts.go

Hash 算法（SHA）相关选项结构。

keystore.go

定义 KeyStore 接口，存储加密秘钥。

```
type KeyStore interface {

    // ReadOnly returns true if this KeyStore is read only, false otherwise.
    // If ReadOnly is true then StoreKey will fail.
    ReadOnly() bool

    // GetKey returns a key object whose SKI is the one passed.
    GetKey(ski []byte) (k Key, err error)

    // StoreKey stores the key k in this KeyStore.
    // If this KeyStore is read only then the method will fail.
    StoreKey(k Key) (err error)
}
```

opts.go

提供一些基础的密码学选项，包括常见算法名称、秘钥生成参数等。

rsaopts.go

RSA 算法相关选项结构，包括RSA2048、RSA3072、RSA4096 算法秘钥生成选项等。

bddtests

行为驱动测试（Behaviour Driven Development）相关代码。

bdd-docker

docker-compose-4-consensus-batch-nosnapshotbuffer.yml

common

common_pb2.py

common_pb2_grpc.py

configtx_pb2.py

configtx_pb2_grpc.py

configuration_pb2.py

configuration_pb2_grpc.py

ledger_pb2.py

ledger_pb2_grpc.py

msp_principal_pb2.py

msp_principal_pb2_grpc.py

policies_pb2.py

policies_pb2_grpc.py

environments

kafka

启动一个 kafka 环境，包括 zookeeper 和 kakka。

启动

```
$ docker-compose up -d
Creating kafka_zookeeper_1
Creating kafka_kafka_1
$
```

扩展到多个

```
$ docker-compose scale kafka=3
Creating and starting kafka_kafka_2 ... done
Creating and starting kafka_kafka_3 ... done
$
```

停止

```
$ docker-compose stop
Stopping kafka_kafka_3 ... done
Stopping kafka_kafka_2 ... done
Stopping kafka_kafka_1 ... done
Stopping kafka_zookeeper_1 ... done
$
```

docker-compose.yml

orderer

docker-compose.yml

docker-entrypoint.sh

Dockerfile

orderer-1-kafka-1

启动一个 3 个节点的服务，包括 zookeeper、fabric-order 和 kafka。

docker-compose.yml

启动一个 5 个节点的服务，包括 zookeeper、fabric-order 和 3 个 kafka 节点。

orderer-1-kafka-3

docker-compose.yml

orderer-n-kafka-n

启动一个可扩展的服务，包括 zookeeper、若干个 fabric-order 和若干个 kafka 节点。

如，启动 3 个 order 节点和 5 个 kafka 节点。

```
$ docker-compose up -d zookeeper
$ docker-compose up -d kafka
$ docker-compose scale kafka=5
$ docker-compose up -d orderer
$ docker-compose scale orderer=3
```

docker-compose.yml

features

对 endorser 和 orderer 服务的功能进行简单测试。

包括 BDD 测试的脚本。

bootstrap.feature

endorser.feature

orderer.feature

gossip

message_pb2.py

message_pb2_grpc.py

msp

mspconfig_pb2.py

mspconfig_pb2_grpc.py

orderer

ab_pb2.py

ab_pb2_grpc.py

configuration_pb2.py

configuration_pb2_grpc.py

kafka_pb2.py

kafka_pb2_grpc.py

peer

admin_pb2.py

admin_pb2_grpc.py

chaincode_pb2.py

chaincode_pb2_grpc.py

chaincodeevent_pb2.py

chaincodeevent_pb2_grpc.py

chaincodeshim_pb2.py

chaincodeshim_pb2_grpc.py

configuration_pb2.py

configuration_pb2_grpc.py

events_pb2.py

events_pb2_grpc.py

peer_pb2.py

peer_pb2_grpc.py

proposal_pb2.py

proposal_pb2_grpc.py

proposal_response_pb2.py

proposal_response_pb2_grpc.py

query_pb2.py

query_pb2_grpc.py

transaction_pb2.py

transaction_pb2_grpc.py

regression

go

ote

tdk

node

performance

results

daily_test_suite.sh

longrun_test_suite.sh

scripts

启动 peer 节点的脚本。

wait-for-it.sh

等待某给定服务的端口可用（服务启动起来），否则不断探测。

steps

bdd_grpc_util.py

bdd_test_util.py

bootstrap_impl.py

bootstrap_util.py

compose.py

contexthelper.py

coverage.py

docgen.py

endorser_impl.py

endorser_util.py

orderer_impl.py

orderer_util.py

templates

html

cli.html

error.html

graph.html

header.html

main.html

org-py.html

org.html

protobuf-py.html

protobuf.html

report.css

scenario.html

step.html

tag.html

user.html

chaincode.go

compose-defaults.yml

compose.go

conn.go

context.go

context_bootstrap.go

context_endorser.go

docker-compose-1-profiling.yml

docker-compose-next-4.yml

docker-compose-next.yml

docker-compose-orderer-base.yml

docker-compose-orderer-kafka.yml

启动 1 个 orderer 节点和 1 个 kafka 节点。

docker-compose-orderer-solo.yml

启动 1 个 orderer 节点，类型为 solo。

docker-compose-with-orderer.yml

启动 1 个 orderer 节点和 4 个 peer 节点。

docker.go

environment.py

identities_pb2.py

identities_pb2_grpc.py

server_admin_pb2.py

tlsca.cert

tlsca.priv

users.go

util.go

common

一些通用的功能模块。

cauthdsl

cauthdsl.go

cauthdsl_builder.go

policy.go

policy_util.go

policyparser.go

config

msp

config.go

config_util.go

api.go

application.go

application_util.go

applicationorg.go

channel.go

channel_util.go

orderer.go

orderer_util.go

organization.go

proposer.go

root.go

standardvalues.go

configtx

负责 config 类型的 transaction。

api

api.go

test

helper.go

tool

configtxgen

localconfig

provisional

configtx.yaml

compare.go

config.go

configmap.go

initializer.go

manager.go

template.go

update.go

util.go

crypto

signer.go

flogging

logging.go

genesis

genesis.go

ledger

blkstorage

fsblkstorage

blockstorage.go

testutil

test_helper.go

test_util.go

util

leveldbhelper

ioutil.go

protobuf_util.go

util.go

ledger_interface.go

localmsp

signer.go

metadata

metadata.go

mocks

configtx

configtx.go

configvalues

channel

crypto

localsigner.go

policies

policies.go

policies

负责对签名进行检查的规则。

implicitmeta.go

implicitmeta_util.go

policy.go

tools

cryptogen

ca

csp

msp

main.go

util

utils.go

viperutil

config_util.go

core

大部分核心实现代码都在本包下。其它包的代码封装上层接口，最终调用本包内代码。

chaincode

chaincode 相关，包括生成 chaincode 镜像，支持对 chaincode 的调用、查询等。

比较核心的结构包括：

- ChaincodeSupport：通过调用 vmc 驱动来支持对 chaincode 容器的管理，包括部署、执行合约等；
- Handler：通过一个状态机来维护 peer 侧对于 chaincode 各种状态下的响应，before、after 等触发条件；

此外，shim 包负责提供 Chaincode 跟账本结构打交道的中间层。

- ChaincodeStub：chaincode 中代码通过该结构提供的方法来修改账本状态；
- Handler：chaincode 一侧用状态机来跟踪 shim 相关事件。

platforms 包提供具体的对 Chaincode 运行类型的支持，包括 golang、java、car 等。例如在部署的时候完成打包任务。

platforms

这里面主要是生成 chaincode 容器镜像时候，进行打包的一些方法类。

目前支持三种类型：car、golang 和 java。

定义了接口 Platform。

```
type Platform interface {
        ValidateSpec(spec *pb.ChaincodeSpec) error
        WritePackage(spec *pb.ChaincodeSpec, tw *tar.Writer) error
      }
```

ValidateSpec 用来对一个 chaincodeSpec 进行检查；WritePackage 根据一个 spec 来生成一个 tar 包，是对应的容器镜像内容。

三种类型分别都实现了这两个方法。

car

car 是打包好的 chaincode 的一种格式，可以参考 chaincodetool。

golang

golang 格式的 chaincode。

java

java 格式的 chaincode。

util

platforms.go

定义抽象的 platform 接口。

shim

shim 包可以让 chaincode 代码跟 ledger 进行交互。

比较重要的 ChaincodeStub 结构，提供了用户 chaincode 代码跟 ledger 进行交互的接口，例如 PutState 与 GetState 来写入和查询链上键值对的状态。

java

chaincode.go

定义接口 Chaincode 和 ChaincodeStub。

```
type Chaincode interface {
// Init is called during Deploy transaction after the container has been
// established, allowing the chaincode to initialize its internal data
Init(stub *ChaincodeStub, function string, args []string) ([]byte, error)

// Invoke is called for every Invoke transactions. The chaincode may change
// its state variables
Invoke(stub *ChaincodeStub, function string, args []string) ([]byte, error)

// Query is called for Query transactions. The chaincode may only read
// (but not modify) its state variables and return the result
Query(stub *ChaincodeStub, function string, args []string) ([]byte, error)}
```

用户自己编写的 chaincode 代码需要实现这三个接口，在代码中通过 stub 来跟 ledger 交互。

```
type ChaincodeStub struct {
UUID string
securityContext *pb.ChaincodeSecurityContext
chaincodeEvent *pb.ChaincodeEvent
args [][]byte
}
```

handler.go

inprocstream.go

interfaces.go

mockstub.go

response.go

testdata

server1.key

server1.pem

ccproviderimpl.go

chaincode_support.go

chaincodeexec.go

chaincodetest.yaml

config.go

exectransaction.go

handler.go

comm

一些简单的常见函数。

testdata

certs

grpc

prime256v1-openssl-cert.pem

prime256v1-openssl-key.pem

config.go

cache 配置中的一些变量的值。

connection.go

跟 GRPC 连接相关的一些方法。

```
func NewClientConnectionWithAddress(peerAddress string, block bool, tslEnabled bool, c
reds credentials.TransportAuthenticator) (*grpc.ClientConn, error)
```

向指定地址建立一条 grpc 通道连接。

```
func InitTLSForPeer() credentials.TransportAuthenticator
```

返回 peer 的 TLS 客户端认证信息，从 peer.tls.cert.file 文件中读取生成。

server.go

committer

Committer 角色接口，实现上部分地方调用 peer 下面的方法。

txvalidator

validator.go

committer.go

committer_impl.go

common

ccprovider

ccprovider.go

sysccprovider

sysccprovider.go

validation

msgvalidation.go

config

主要包括 config.go。

config 包和 comm 包其实可以合并。

config.go

配置一些测试变量，主要供测试时使用。

container

容器操作相关的方法实现。

chaincode 目前是运行在容器内的，这个包主要提供与之相关的操作。

api

core.go

ccintf

定义 chaincode 和 peer 之间进行通信的接口，目前只有 chaincode 裸跑的时候才被使用。

ccintf.go

比较重要的是

```
//CCID encapsulates chaincode ID
type CCID struct {
    ChaincodeSpec *pb.ChaincodeSpec
    NetworkID     string
    PeerID        string
}
```

dockercontroller

Docker 相关的操作。

dockercontroller.go

抽象一个 Docker 主机。

主要结构为

```
type DockerVM struct {
    id string
}
```

支持的方法包括：

- Deploy：利用给定的 tar.gz 文件生成一个镜像；
- Destroy：删除一个镜像。
- Start：启动一个 Docker 容器，命名为 网络 id + peer id + chaincode 名（hash 串），会从配置中读取 hostconfig 信息。
- Stop：停止一个 Docker 容器。

inproccontroller

chaincode 直接裸跑在主机上时候的一些操作。

inproccontroller.go

inprocstream.go

util

一些辅助方法。

dockerutil.go

从配置中读取信息，创建一个 docker client 等。

writer.go

将目录、包、流等的内容写到 tar 包中。

config.go

用于测试时候的一些配置。

controller.go

抽象出来的 VM 控制器，目前支持 Docker 和 系统运行，将来可以支持更多类型的容器。

vm.go

主要数据结构为

```
type VM struct {
    Client *docker.Client
}
```

一个 VM，实际代表的是一个容器主机，目前支持列出镜像（用来测试）、生成 chaincode 容器等方法。

主要的方法是 buildChaincodeContainerUsingDockerfilePackageBytes()，根据传入的字节来生成一个 chaincode 镜像。

crypto

跟加密相关的一些功能，包括安全验证等。

primitives

elliptic.go

hash.go

init.go

random.go

deliverservice

blocksprovider

blocksprovider.go

mocks

blocksprovider.go

deliveryclient.go

endorser

Endorser 角色接口，实现上部分地方调用 peer 下面的方法。

config.go

endorser.go

errors

errors.go

ledger

账本的实现，包括区块链（blockchain）和世界状态（world state）。

kvledger

example

history

marble_example

txmgmt

kv_ledger.go

kv_ledger_provider.go

recovery.go

ledgerconfig

ledger_config.go

ledgermgmt

ledger_mgmt.go

ledger_mgmt_test_exports.go

testutil

test_util.go

util

couchdb

filterbitarray.go

txvalidationflags.go

util.go

辅助函数。

ledger_interface.go

mocks

ccprovider

ccprovider.go

txvalidator

support.go

validator

validator.go

peer

构成一个 fabric peer 的核心实现。

主要包括：

- peer 包：负责定义一个 peer 节点具有的各种行为。
- handler 包：处理 peer 收到的各种消息，十分关键。

testdata

generate.go

Org1-cert.pem

Org1-key.pem

Org1-server1-cert.pem

Org1-server1-key.pem

Org1-server2-cert.pem

Org1-server2-key.pem

Org2-cert.pem

Org2-key.pem

Org2-server1-cert.pem

Org2-server1-key.pem

Org2-server2-cert.pem

Org2-server2-key.pem

Org3-cert.pem

Org3-key.pem

Org3-server1-cert.pem

Org3-server1-key.pem

Org3-server2-cert.pem

Org3-server2-key.pem

config.go

跟配置相关的一些方法。

主要是配置的一些选项可能需要一些计算和检测，通过这些方法可以做一些 cache 等。

包括下面一些配置项：

```
var localAddress string
var localAddressError error
var peerEndpoint *pb.PeerEndpoint
var peerEndpointError error

// Cached values of commonly used configuration constants.
var syncStateSnapshotChannelSize int
var syncStateDeltasChannelSize int
var syncBlocksChannelSize int
var validatorEnabled bool
```

errors.go

一些错误信息。

peer.go

peer 相关

比较核心的数据结构。

Peer 接口，两个方法：包括获取一个 peer 端点和发送探测 hello 消息。

```
type Peer interface {
    GetPeerEndpoint() (*pb.PeerEndpoint, error)
    NewOpenchainDiscoveryHello() (*pb.Message, error)
}
```

具体实现的数据结构为 PeerImpl。

```
type PeerImpl struct {
    handlerFactory HandlerFactory  // 生成一个 MessageHandler
    handlerMap     *handlerMap     // 所有注册上来的消息处理器
    ledgerWrapper  *ledgerWrapper  // ledger 操作句柄
    secHelper      crypto.Peer     // 处理身份验证和安全相关
    engine         Engine          // handler 工厂 + 本地交易处理的引擎
    isValidator    bool
    reconnectOnce  sync.Once
    discHelper     discovery.Discovery //探测任务句柄
    discPersist    bool
}
```

核心方法，包括：

- ExecuteTransaction：准备执行一个交易，发送给本地的引擎（VP 节点），或给远端的 peer；
- Broadcast：向所有注册的消息句柄发送消息；
- Unicast：向指定的 peer 发送消息；
- 一系列 Get 方法：包括获取 Block 内容、获取当前链的大小、当前状态的 hash、获取已注册的 peer 端点、远端 ledger 等等，很多功能实际上都是通过其它包来完成。
- sendTransactionsToLocalEngine：交易发给本地的引擎处理；
- SendTransactionsToPeer：交易发给其它 peer 处理；

消息相关

两个基础接口 MessageHandler 和 MessageHandlerCoordinator。

```
type MessageHandler interface {
    RemoteLedger    // 获取远端的 ledger
    HandleMessage(msg *pb.Message) error // 接收到某个消息进行处理
    SendMessage(msg *pb.Message) error //发送消息到对端
    To() (pb.PeerEndpoint, error) // 对端是哪个节点
    Stop() error
}
```

```
type MessageHandlerCoordinator interface {
    Peer
    SecurityAccessor
    BlockChainAccessor
    BlockChainModifier
    BlockChainUtil
    StateAccessor
    RegisterHandler(messageHandler MessageHandler) error
    DeregisterHandler(messageHandler MessageHandler) error
    Broadcast(*pb.Message, pb.PeerEndpoint_Type) []error
    Unicast(*pb.Message, *pb.PeerID) error
    GetPeers() (*pb.PeersMessage, error)
    GetRemoteLedger(receiver *pb.PeerID) (RemoteLedger, error)
    PeersDiscovered(*pb.PeersMessage) error
    ExecuteTransaction(transaction *pb.Transaction) *pb.Response
    Discoverer
}
```

policy

mocks.go

policy.go

scc

cscc

configure.go

escc

endorser_onevalidsignature.go

lccc

lccc.go

qscc

query.go

samplesyscc

samplesyscc.go

vscc

validator_onevalidsignature.go

importsysccs.go

sccproviderimpl.go

sysccapi.go

admin.go

对 peer 服务的管理：启动和停止、查看等。

config.go

配置相关。

fsm.go

peer connection 的状态机。

```
func NewPeerConnectionFSM(to string) *PeerConnectionFSM {
    d := &PeerConnectionFSM{
        To: to,
    }

    d.FSM = fsm.NewFSM(
        "created",
        fsm.Events{
            {Name: "HELLO", Src: []string{"created"}, Dst: "established"},
            {Name: "GET_PEERS", Src: []string{"established"}, Dst: "established"},
            {Name: "PEERS", Src: []string{"established"}, Dst: "established"},
            {Name: "PING", Src: []string{"established"}, Dst: "established"},
            {Name: "DISCONNECT", Src: []string{"created", "established"}, Dst: "closed"
},
        },
        fsm.Callbacks{
            "enter_state":  func(e *fsm.Event) { d.enterState(e) },
            "before_HELLO": func(e *fsm.Event) { d.beforeHello(e) },
            "after_HELLO":  func(e *fsm.Event) { d.afterHello(e) },
            "before_PING":  func(e *fsm.Event) { d.beforePing(e) },
            "after_PING":   func(e *fsm.Event) { d.afterPing(e) },
        },
    )

    return d
}
```

devenv

主要是方便本地搭建开发平台的一些脚本。

images

tools

couchdb

failure-motd.in

golang_buildcmd.sh

golang_buildpkg.sh

limits.conf

setup.sh

setupRHELonZ.sh

setupUbuntuOnPPC64le.sh

Vagrantfile

docs

项目相关的所有文档。

custom_theme

searchbox.html

source

_static

css

_templates

layout.html

API

Samples

AttributesUsage.rst

ChaincodeAPI.rst

CLI.rst

biz

images

DCO1.1.txt

usecases.rst

dev-setup

build.rst

devenv.rst

headers.txt

FAQ

architecture_FAQ.rst

chaincode_FAQ.rst

confidentiality_FAQ.rst

consensus_FAQ.rst

identity_management_FAQ.rst

usage_FAQ.rst

Gerrit

images

sec-firstrel-1.jpg

sec-sources-1.pptx

sec-sources-2.pptx

sec-txconf-v1.2.pptx

Setup

ca-setup.rst

Chaincode-setup.rst

JAVAChaincode.rst

logging-control.rst

Network-setup.rst

NodeSDK-setup.rst

TLSSetup.rst

starter

fabric-starter-kit.rst

Style-guides

go-style.rst

tech

application-ACL.rst

attributes.rst

best-practices.rst

wiki-images

abstract_v1.rst

adminops.rst

arch-deep-dive.rst

architecture.rst

asset_cli.rst

asset_sdk.rst

asset_setup.rst

asset_trouble.rst

auction.rst

best_practices.rst

blockchain.rst

bootstrap.rst

capabilities.rst

carlease.rst

case_for_fabric.rst

chaincode.rst

channel-setup.rst

channels.rst

committer.rst

components.rst

conf.py

configtxgen.rst

consensus.rst

CONTRIBUTING.rst

debug.rst

demos.rst

docker-2peer.yml

docker-compose-channel.yml

dockercompose.rst

endorsement-policies.rst

endorser.rst

fabric_model.rst

glossary.rst

gossip.rst

included.rst

index.rst

install_instantiate.rst

javasdk.rst

jira_navigation.rst

learn_chaincode.rst

ledger.rst

license.rst

MAINTAINERS.rst

make.bat

marbles.rst

mdtorst.sh

multichannel.rst

nodesdk.rst

orderingservice.rst

overview.rst

paper.rst

peer-chaincode-devmode.rst

pluggableos.rst

policies.rst

protocol-spec.rst

protocol-spec_zh.rst

pythonsdk.rst

quality.rst

questions.rst

readwrite.rst

releases.rst

requirements.txt

sampleapp.rst

security_model.rst

smartcontract.rst

status.rst

SystemChaincode-noop.rst

testing.rst

troubleshooting.rst

txflow.rst

videos.rst

whyfabric.rst

Makefile

requirements.txt

events

EventHub 服务处理相关的模块。

Event 包括四种类型：

```
enum EventType {
        REGISTER = 0;
        BLOCK = 1;
    CHAINCODE = 2;
    REJECTION = 3;
}
```

consumer

adapter.go

consumer.go

producer

eventhelper.go

events.go

handler.go

producer.go

register_internal_events.go

config.go

examples

示例文件，包括一些 chaincode 示例和监听事件的示例。

ccchecker

chaincodes

newkeyperinvoke

chaincodes.go

registershadow.go

ccchecker.go

ccchecker.json

init.go

main.go

chaincode

chaintool

example02

go

asset_management

asset_management02

asset_management_interactive

asset_management_with_roles

attributes_to_state

authorizable_counter

chaincode_example01

chaincode_example02

chaincode_example03

chaincode_example04

chaincode_example05

eventsender

invokereturnsvalue

map

marbles02

passthru

rbac_tcerts_no_attrs

sleeper

utxo

java

Example

LinkExample

MapExample

RangeExample

SimpleSample

e2e_cli

crypto

orderer

peer

examples

chaincode

peer-base

peer-base-no-tls.yaml

peer-base.yaml

scripts

script.sh

configtx.yaml

docker-compose-no-tls.yaml

docker-compose.yaml

end-to-end.rst

generateCfgTrx.sh

network_setup.sh

events

block-listener

block-listener.go

sfhackfest

ccenv

Dockerfile

tmp

ca

orderer

peer0

peer1

peer2

peer3

docker-compose-gettingstarted.yml

sfhackfest.tar.gz

gossip

api

channel.go

crypto.go

comm

mock

mock_comm.go

comm.go

comm_impl.go

conn.go

crypto.go

demux.go

msg.go

common

common.go

discovery

discovery.go

discovery_impl.go

election

adapter.go

election.go

filter

filter.go

gossip

algo

pull.go

channel

channel.go

msgstore

msgs.go

pull

pullstore.go

batcher.go

certstore.go

chanstate.go

gossip.go

gossip_impl.go

identity

identity.go

integration

integration.go

service

eventer.go

gossip_service.go

state

metastate.go

payloads_buffer.go

state.go

util

logging.go

misc.go

msgs.go

gotools

go 相关的开发工具的安装脚本：golint、govendor、goimports、protoc-gen-go、ginkgo、gocov、gocov-xml 等。

- golint：支持 golang 的静态语法和格式检查；
- govendor：管理第三方引入包；
- goimports：格式化 import 的包；
- protoc-gen-go：对 protobuf 的支持；
- ginkgo：支持 BDD 的框架；
- gocov：golang 的单元测试覆盖率检查工具；
- gocov-xml：支持 gocov 生成 xml 格式的报告。

在该包下执行 `make` 或在上层执行 make gotools 会自动安装上述工具。

Makefile

images

一些跟 Docker 镜像生成相关的配置和脚本。主要包括各个镜像的 Dockerfile.in 文件。这些文件是生成 Dockerfile 的模板。

主要包括如下镜像的 Dockerfile 的模板和生成相关脚本：

- ccenv：golang chaincode 运行的基础环境，包括 chaintool、protoc-gen-go 和 goshim 包。
- javaenv：运行 java chaincode 的基础环境，包括 javashim、protos、maven、java 环境等。
- kafka：kafka 镜像。
- order：fabric-orderer 镜像，包括 orderer 二进制文件、orderer.yaml 配置文件、msp-sampleconfig 包等。
- peer：fabric-peer 镜像，包括 peer 二进制文件、core.yaml 配置文件、msp-sampleconfig、genesis-sampleconfig 包等。
- tetenv：测试环境镜像，包括 gotools、peer、orderer 等二进制和配置文件、msp-sampleconfig、softhsm2。
- zookeeper：zookeeper 镜像。

ccenv

生成 chaincode 运行的环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
COPY payload/chaintool payload/protoc-gen-go /usr/local/bin/
ADD payload/goshim.tar.bz2 $GOPATH/src/
```

Dockerfile.in

couchdb

docker-entrypoint.sh

Dockerfile.in

local.ini

vm.args

javaenv

生成 java chaincode 运行的环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
RUN wget https://services.gradle.org/distributions/gradle-2.12-bin.zip -P /tmp --quiet
RUN unzip -q /tmp/gradle-2.12-bin.zip -d /opt && rm /tmp/gradle-2.12-bin.zip
RUN ln -s /opt/gradle-2.12/bin/gradle /usr/bin
ADD payload/javashim.tar.bz2 /root
ADD payload/protos.tar.bz2 /root
ADD payload/settings.gradle /root
WORKDIR /root
# Build java shim after copying proto files from fabric/proto
RUN core/chaincode/shim/java/javabuild.sh
```

Dockerfile.in

kafka

docker-entrypoint.sh

Dockerfile.in

kafka-run-class.sh

orderer

生成 fabric-order 镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-runtime:_TAG_
ENV ORDERER_CFG_PATH /etc/hyperledger/fabric
RUN mkdir -p /var/hyperledger/db /etc/hyperledger/fabric
COPY payload/orderer /usr/local/bin
COPY payload/orderer.yaml $ORDERER_CFG_PATH
EXPOSE 7050
CMD orderer
```

Dockerfile.in

peer

生成 fabric-peer 镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-runtime:_TAG_
ENV PEER_CFG_PATH /etc/hyperledger/fabric
RUN mkdir -p /var/hyperledger/db $PEER_CFG_PATH/msp/sampleconfig/signcerts $PEER_CFG_P
ATH/msp/sampleconfig/admincerts $PEER_CFG_PATH/msp/sampleconfig/keystore $PEER_CFG_PATH
/msp/sampleconfig/cacerts
COPY payload/peer /usr/local/bin
COPY payload/core.yaml $PEER_CFG_PATH
COPY payload/msp/sampleconfig/signcerts/peer.pem $PEER_CFG_PATH/msp/sampleconfig/signc
erts
COPY payload/msp/sampleconfig/admincerts/admincert.pem $PEER_CFG_PATH/msp/sampleconfig
/admincerts
COPY payload/msp/sampleconfig/keystore/key.pem $PEER_CFG_PATH/msp/sampleconfig/keystor
e
COPY payload/msp/sampleconfig/cacerts/cacert.pem $PEER_CFG_PATH/msp/sampleconfig/cacer
ts
CMD peer node start
```

Dockerfile.in

testenv

生成一个测试环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
ADD payload/gotools.tar.bz2 /usr/local/bin/
WORKDIR /opt/gopath/src/github.com/hyperledger/fabric
```

Dockerfile.in

install-softhsm2.sh

zookeeper

docker-entrypoint.sh

Dockerfile.in

msp

成员服务提供者（Member Service Provider），提供一组认证相关的密码学机制和协议，用来负责对网络提供证书分发、校验，身份认证管理等。

通常情况下，一个组织可以作为一个 MSP，负责对旗下所有成员的管理。

该包下面的 `sampleconfig` 目录中提供了样例配置文件，主要包括各个证书。

mgmt

testtools

config.go

deserializer.go

mgmt.go

principal.go

sampleconfig

提供了一些样例证书文件。

pem （Privacy Enhanced Mail，属于 X.509 证书标准格式）格式，内容是 BASE64 编码，可以通过 openssl 来查看内容。

如

```
$ openssl x509 -in msp/sampleconfig/admincerts/admincert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:70:93:0c:e5:38:ee:c5:02:e4:ae:24:9f:f0:9a:aa:78:75:d7:86
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=California, L=San Francisco, O=Internet Widgets, Inc., OU=WWW
, CN=example.com
        Validity
            Not Before: Oct 12 19:31:00 2016 GMT
            Not After : Oct 11 19:31:00 2021 GMT
        Subject: C=US, ST=California, L=San Francisco, O=Internet Widgets, Inc., OU=WW
W, CN=example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
            EC Public Key:
                pub:
                    04:a2:07:e5:bd:89:69:29:aa:89:05:c7:ca:a0:be:
                    72:69:27:20:27:05:8b:90:1d:75:58:ec:42:2c:c3:
                    57:ab:99:e2:fe:ac:f8:f5:a2:27:2c:df:03:fa:d3:
                    b4:cb:d8:00:fa:bf:c9:e1:07:a4:15:01:d6:3d:39:
                    de:6c:67:a4:cb
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                17:67:42:3D:AA:9E:82:3F:C4:C5:1D:9F:5B:C3:99:D1:B5:9C:48:10
            X509v3 Authority Key Identifier:
                keyid:17:67:42:3D:AA:9E:82:3F:C4:C5:1D:9F:5B:C3:99:D1:B5:9C:48:10

    Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:07:a7:94:5b:a7:d1:26:d4:6f:d4:98:a9:fc:5a:
        c0:9b:a8:37:d6:79:c5:5b:64:f4:23:dd:12:b8:a0:6e:64:41:
        02:20:40:07:b8:38:e2:98:84:97:61:dd:fe:d4:45:a2:9f:19:
        37:f8:f7:6f:e7:99:19:ad:2b:ec:92:2a:3a:47:4a:b5
```

admincerts

admincert.pem

cacerts

cacert.pem

keystore

key.pem

signcerts

peer.pem

configbuilder.go

identities.go

定义了 identity 和 signingidentity 私有结构。

```
type identity struct {
    // id contains the identifier (MSPID and identity identifier) for this instance
    id *IdentityIdentifier

    // cert contains the x.509 certificate that signs the public key of this instance
    cert *x509.Certificate

    // this is the public key of this instance
    pk bccsp.Key

    // reference to the MSP that "owns" this identity
    msp *bccspmsp
}

type signingidentity struct {
    // we embed everything from a base identity
    identity

    // signer corresponds to the object that can produce signatures from this identity
    signer *signer.CryptoSigner
}
```

identities.pb.go

identities.proto

msp.go

定义了一些基础功能接口，包括

- MSPManager：管理 MSP
- MSP：服务提供者的抽象，提供签名、校验等功能。
- Identity：跟证书相关的操作，包括获取内容，校验内容等。
- SigningIdentity：继承自 Identity，进一步支持签名功能。

mspimpl.go

对 MSP 接口的实现，实现了 bccspmsp 结构，可以通过 NewBccspMsp() 方法生成。

bccspmsp 提供一个默认的 MSP 实现，成员包括：

- Type 为 `FABRIC = 0`；
- bccsp 为 SHA-2 256；

方法主要包括：

- Setup()：利用给定的配置信息，进行初始化操作。
- GetType()：返回类型，目前为 `FABRIC`（值为 0）。
- GetIdentifier()：返回 msp 的名称。
- GetDefaultSigningIdentity()：获取本 MSP 中的默认签名个体。
- GetSigningIdentity()：获取本 MSP 中的签名个体。
- Validate（）：对给定的 Identity 对象进行校验。
- DeserializeIdentity()：从序列化对象中解析 Identity 对象。
- SatisfiesPrincipal()：检查某个 Identity 是否符合给定的策略。
- getIdentityFromConf()：从本地配置文件中解析出 x.509 格式的证书信息，包括公钥等，利用这些信息生成 Identity 对象。
- getSigningIdentityFromConf()：从本地配置文件中解析出 x.509 格式的证书信息，包括公钥等，利用这些信息生成带签名功能的 Identity 对象。

mspmgrimpl.go

noopmsp.go

orderer

common

blockcutter

blockcutter.go

bootstrap

file

bootstrap.go

broadcast

broadcast.go

configtxfilter

filter.go

deliver

deliver.go

filter

filter.go

sigfilter

sigfilter.go

sizefilter

sizefilter.go

configupdate

configupdate.go

kafka

broker.go

chain_partition.go

consumer.go

log.go

orderer.go

partitioner.go

producer.go

util.go

ledger

file

factory.go

impl.go

json

factory.go

impl.go

ram

factory.go

impl.go

ledger.go

util.go

localconfig

config.go

mocks

blockcutter

blockcutter.go

multichain

multichain.go

util

util.go

multichain

chainsupport.go

manager.go

systemchain.go

sample_clients

broadcast_config

client.go

newchain.go

broadcast_timestamp

client.go

deliver_stdout

client.go

single_tx_client

single_tx_client.go

sbft

backend

backend.go

consensus.pb.go

consensus.proto

events.go

connection

connection.go

crypto

crypto.go

persist

persist.go

simplebft

backlog.go

batch.go

checkpoint.go

commit.go

connection.go

crypto.go

newview.go

prepare.go

preprepare.go

request.go

simplebft.go

simplebft.pb.go

simplebft.proto

viewchange.go

xset.go

testdata

cert1.pem

config.json

key.pem

config.go

config.pb.go

config.proto

consenter.go

local-deploy.sh

solo

consensus.go

main.go

orderer.yaml

server.go

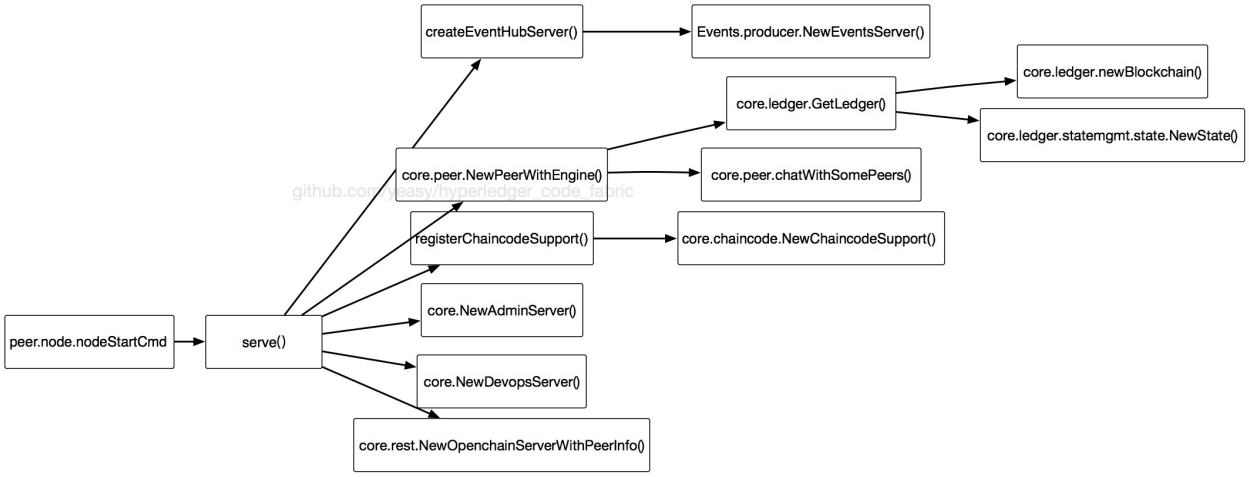
util.go

peer

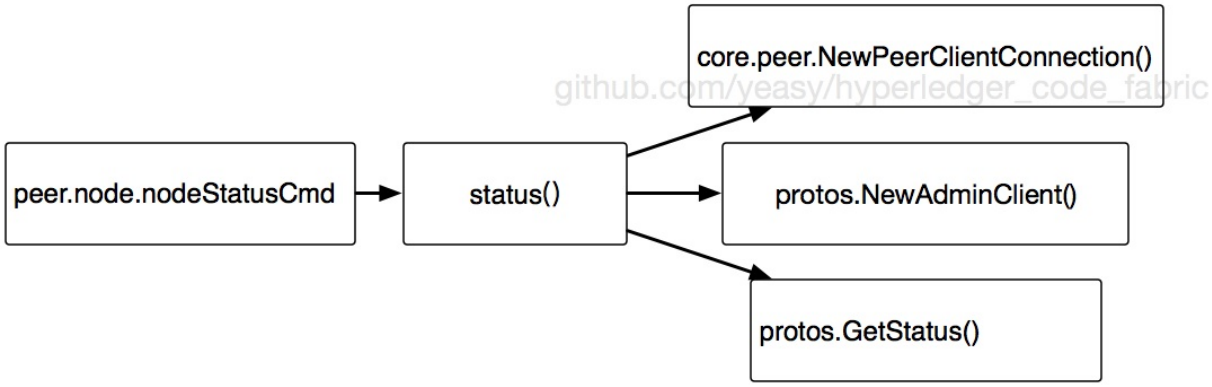
主模块，代表一个节点。

调用其他功能模块来具体实现。

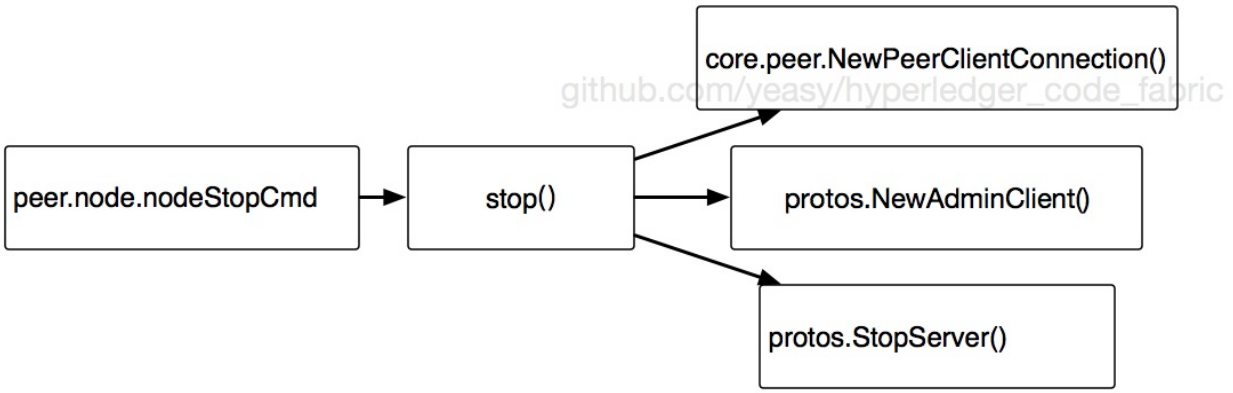
启动服务



查看状态



停止服务



chaincode

features

steps

environment.py

install.feature

instantiate.feature

chaincode.go

common.go

install.go

instantiate.go

invoke.go

package.go

query.go

upgrade.go

channel

channel.go

create.go

deliverclient.go

fetchconfig.go

join.go

clilogging

common.go

getlevel.go

logging.go

setlevel.go

common

common.go

mockclient.go

ordererclient.go

gossip

mcs

mcs.go

mocks.go

sa

sa.go

node

负责 `peer node` 子命令。

node.go

node 子命令进一步支持 start、stop、join、status 等子命令。

```
// Cmd returns the cobra command for Node
func Cmd() *cobra.Command {
    nodeCmd.AddCommand(startCmd())
    nodeCmd.AddCommand(statusCmd())
    nodeCmd.AddCommand(stopCmd())
    nodeCmd.AddCommand(joinCmd())

    return nodeCmd
}
```

start.go

负责 `peer node start` 命令。

最重要的是 `func serve(args []string) error` 函数，启动一个节点服务，主要是启动各个 GRPC 的服务端。

配置读取和缓存

首先是进行配置管理，根据配置信息和一些计算来构建 cache 结构，探测节点信息等。主要调用 core.peer 包来实现。

```
    if err := peer.CacheConfiguration(); err != nil {
        return err
    }

    peerEndpoint, err := peer.GetPeerEndpoint()
    if err != nil {
        err = fmt.Errorf("Failed to get Peer Endpoint: %s", err)
        return err
    }
```

创建 eventHub 服务

eventHub 服务监听到 7053 端口，仅在 VP 节点上打开。

调用 `createEventHubServer` 方法实现，主要过程为：

创建 EventHub 服务，通过调用 createEventHubServer() 方法来实现，该服务也是 grpc，只有 vp 节点才开启。

```
    lis, err = net.Listen("tcp", viper.GetString("peer.validator.events.address"))
    if err != nil {
        return nil, nil, fmt.Errorf("failed to listen: %v", err)
    }

    //TODO - do we need different SSL material for events ?
    var opts []grpc.ServerOption
    if comm.TLSEnabled() {
        creds, err := credentials.NewServerTLSFromFile(viper.GetString("peer.tls.cert.
file"), viper.GetString("peer.tls.key.file"))
        if err != nil {
            return nil, nil, fmt.Errorf("Failed to generate credentials %v", err)
        }
        opts = []grpc.ServerOption{grpc.Creds(creds)}
    }

    grpcServer = grpc.NewServer(opts...)
    ehServer := producer.NewEventsServer(uint(viper.GetInt("peer.validator.events.buff
ersize")), viper.GetInt("peer.validator.events.timeout"))
    pb.RegisterEventsServer(grpcServer, ehServer)
```

eventHub 服务支持的方法为 Chat。

```
type EventsServer interface {
    // event chatting using Event
    Chat(Events_ChatServer) error
}
```

创建和注册 grpc 服务

创建 gprc 服务，并注册上 chaincode、admin、endorser、gossip 等服务，并初始化注册 chainless 系统 chaincode 和创建初始区块。

```
grpcServer := grpc.NewServer(opts...)

registerChaincodeSupport(grpcServer)

logger.Debugf("Running peer")

// Register the Admin server
pb.RegisterAdminServer(grpcServer, core.NewAdminServer())

// Register the Endorser server
serverEndorser := endorser.NewEndorserServer()
pb.RegisterEndorserServer(grpcServer, serverEndorser)

// Initialize gossip component
bootstrap := viper.GetStringSlice("peer.gossip.bootstrap")
service.InitGossipService(peerEndpoint.Address, grpcServer, bootstrap...)
defer service.GetGossipService().Stop()
```

其中，chaincode 服务支持方法为

```
type ChaincodeSupportServer interface {
    Register(ChaincodeSupport_RegisterServer) error
}
```

admin 服务支持方法为

```
type AdminServer interface {
    // Return the serve status.
    GetStatus(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    StartServer(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    StopServer(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    GetModuleLogLevel(context.Context, *LogLevelRequest) (*LogLevelResponse, error)
    SetModuleLogLevel(context.Context, *LogLevelRequest) (*LogLevelResponse, error)
}
```

endorser 服务支持方法为

```
type EndorserServer interface {
    ProcessProposal(context.Context, *SignedProposal) (*ProposalResponse, error)
}
```

gossip 服务支持方法为

```
type GossipServer interface {
    // GossipStream is the gRPC stream used for sending and receiving messages
    GossipStream(Gossip_GossipStreamServer) error
    // Ping is used to probe a remote peer's aliveness
    Ping(context.Context, *Empty) (*Empty, error)
}
```

启动 grpc 服务和 eventHub 服务

之后是启动 grpc 服务，监听到 7051 端口。

```
go func() {
        var grpcErr error
        if grpcErr = grpcServer.Serve(lis); grpcErr != nil {
            grpcErr = fmt.Errorf("grpc server exited with error: %s", grpcErr)
        } else {
            logger.Info("grpc server exited")
        }
        serve <- grpcErr
    }()
```

启动 eventHub 服务。

```
if ehubGrpcServer != nil && ehubLis != nil {
        go ehubGrpcServer.Serve(ehubLis)
    }
```

最后，如果需要 profiling，还会打开监听服务。

status.go

负责 `peer node status` 命令。

主要包括 status 方法，通过 admin 服务获取 peer 状态。

```
func status() (err error) {

    adminClient, err := common.GetAdminClient()
    if err != nil {
        logger.Warningf("%s", err)
        fmt.Println(&pb.ServerStatus{Status: pb.ServerStatus_UNKNOWN})
        return err
    }

    status, err := adminClient.GetStatus(context.Background(), &empty.Empty{})
    if err != nil {
        logger.Infof("Error trying to get status from local peer: %s", err)
        err = fmt.Errorf("Error trying to connect to local peer: %s", err)
        fmt.Println(&pb.ServerStatus{Status: pb.ServerStatus_UNKNOWN})
        return err
    }
    fmt.Println(status)
    return nil
}
```

stop.go

负责 `peer node stop` 命令。

主要包括 stop 方法，通过 admin 服务停止节点。

```
func stop() (err error) {
    clientConn, err := peer.NewPeerClientConnection()
    if err != nil {
        pidFile := stopPidFile + "/peer.pid"
        //fmt.Printf("Stopping local peer using process pid from %s \n", pidFile)
        logger.Infof("Error trying to connect to local peer: %s", err)
        logger.Infof("Stopping local peer using process pid from %s", pidFile)
        pid, ferr := readPid(pidFile)
        if ferr != nil {
            err = fmt.Errorf("Error trying to read pid from %s: %s", pidFile, ferr)
            return
        }
        killerr := syscall.Kill(pid, syscall.SIGTERM)
        if killerr != nil {
            err = fmt.Errorf("Error trying to kill -9 pid %d: %s", pid, killerr)
            return
        }
        return nil
    }
    logger.Info("Stopping peer using grpc")
    serverClient := pb.NewAdminClient(clientConn)

    status, err := serverClient.StopServer(context.Background(), &empty.Empty{})
    if err != nil {
        fmt.Println(&pb.ServerStatus{Status: pb.ServerStatus_STOPPED})
        return nil
    }

    err = fmt.Errorf("Connection remain opened, peer process doesn't exit")
    fmt.Println(status)
    return err
}
```

version

version.go

core.yaml

peer 节点相关的样例配置。

main.go

peer 服务是主服务。

该服务支持各种 peer 命令。

包括查询状态，和启动、停止节点服务等。

各个子命令在对应子包中，如

- node：node 对应子命令
- chaincode：chaincode 对应子命令

proposals

r1

protos

Protobuf 格式的数据结构和消息协议。都在同一个 protos 包内。

这里面是所有基本的数据结构（message）定义和 GRPC 的服务（service）接口声明。

所有的 `.proto` 文件是 protobuf 格式的声明文件，`.pb.go` 文件是基于 `.proto` 文件生成的 go 语言的类文件。

protobuf 工具可以从 这里 下载，推荐使用 3.0 版本系列。

下载安装后，需要安装对应语言的编译器，例如要生成 go 语言代码，则需要安装 `protoc-gen-go` 。

```
$ go get github.com/golang/protobuf/protoc-gen-go
```

可以使用 protobuf 编译器基于 protobuf 模板文件来生成各种语言的类文件。

```
$ protoc \
--proto_path=IMPORT_PATH \
--cpp_out=DST_DIR \
--java_out=DST_DIR \
--python_out=DST_DIR \
--go_out=DST_DIR \
--ruby_out=DST_DIR \
--javanano_out=DST_DIR \
--objc_out=DST_DIR \
--csharp_out=DST_DIR \
path/to/file.proto
```

其中，`--proto_path=IMPORT_PATH` 是当 proto 文件中存在导入时候，进行查找的路径，等价于 `-I=IMPORT_PATH` ，可以多次使用来指定多个导入路径。

为了生成支持 grpc 的代码，还可以提供生成参数 `plugins=grpc` ，例如

```
$ protoc \
--proto_path=IMPORT_PATH \
--go_out=plugins=grpc:DST_DIR \
path/to/file.proto
```

另外，生成的结构体，一般都至少默认支持 4 个默认生成的方法。

- Reset()：重置结构体。
- String() string：返回代表对象的字符串。

- ProtoMessage()：协议消息。
- Descriptor([]byte, []int)：描述信息。

common

block.go

common.pb.go

common.proto

configtx.go

configtx.pb.go

configtx.proto

configuration.pb.go

configuration.proto

ledger.pb.go

ledger.proto

msp_principal.pb.go

msp_principal.proto

policies.pb.go

policies.proto

signed_data.go

gossip

extensions.go

message.pb.go

message.proto

msp

mspconfig.pb.go

mspconfig.proto

orderer

ab.pb.go

ab.proto

configuration.pb.go

configuration.proto

kafka.pb.go

kafka.proto

peer

admin.pb.go

admin.proto

chaincode.pb.go

chaincode.proto

chaincodeevent.pb.go

chaincodeevent.proto

chaincodeshim.pb.go

chaincodeshim.proto

chaincodeunmarshall.go

configuration.pb.go

configuration.proto

events.pb.go

events.proto

init.go

peer.pb.go

peer.proto

proposal.pb.go

proposal.proto

proposal_response.pb.go

proposal_response.proto

query.pb.go

query.proto

transaction.pb.go

transaction.proto

testutils

txtestutils.go

utils

blockutils.go

commonutils.go

proputils.go

txutils.go

scripts

一些辅助脚本，多数为外部 Makefile 调用。

compile_protos.sh

找到所有的 `.proto` 文件，编译生成支持 grpc 的 `.go` 文件。

containerlogs.sh

将本地的日志文件上传到 chunk.io，方便进一步分析。

foldercopy.sh

clone 远端给定用户仓库下的 fabric 代码到本地的 Go 路径下。

golinter.sh

进行语法检查等，目前主要用 goimports 来检查引入包的语法格式。

goListFiles.sh

列出所有的包，检查导入路径存在情况。

infiniteloop.sh

循环来保持容器始终不退出。

test

用于测试的一些脚本。

docker-compose.yml

启动一个 order 和 一个 peer 节点。

```
orderer:
  image: hyperledger/fabric-orderer
  environment:
    - ORDERER_GENERAL_LEDGERTYPE=ram
    - ORDERER_GENERAL_BATCHTIMEOUT=10s
    - ORDERER_GENERAL_BATCHSIZE=10
    - ORDERER_GENERAL_MAXWINDOWSIZE=1000
    - ORDERER_GENERAL_ORDERERTYPE=solo
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_LISTENPORT=7050
    - ORDERER_RAMLEDGER_HISTORY_SIZE=100
  expose:
    - 7050

vp:
  image: hyperledger/fabric-peer
  links:
    - orderer
  ports:
    - 7051:7051
    - 7053:7053
    - 7054:7054
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
```

unit-test

docker-compose.yml

run.sh