

目錄

前言	1.1
整体结构	1.2
bccsp	1.3
factory	1.3.1
factory.go	1.3.1.1
nopkcs11.go	1.3.1.2
opts.go	1.3.1.3
pkcs11.go	1.3.1.4
pkcs11factory.go	1.3.1.5
swfactory.go	1.3.1.6
mocks	1.3.2
mocks.go	1.3.2.1
pkcs11	1.3.3
conf.go	1.3.3.1
ecdsa.go	1.3.3.2
ecdsakey.go	1.3.3.3
impl.go	1.3.3.4
pkcs11.go	1.3.3.5
signer	1.3.4
signer.go	1.3.4.1
sw	1.3.5
mocks	1.3.5.1
mocks.go	1.3.5.1.1
aes.go	1.3.5.2
aeskey.go	1.3.5.3
conf.go	1.3.5.4
dummyks.go	1.3.5.5
ecdsa.go	1.3.5.6
ecdsakey.go	1.3.5.7
fileks.go	1.3.5.8

hash.go	1.3.5.9
impl.go	1.3.5.10
internals.go	1.3.5.11
keyderiv.go	1.3.5.12
keygen.go	1.3.5.13
keyimport.go	1.3.5.14
rsa.go	1.3.5.15
rsakey.go	1.3.5.16
utils	1.3.6
errs.go	1.3.6.1
io.go	1.3.6.2
keys.go	1.3.6.3
slice.go	1.3.6.4
x509.go	1.3.6.5
aesopts.go	1.3.7
bccsp.go	1.3.8
ecdsaopts.go	1.3.9
hashopts.go	1.3.10
keystore.go	1.3.11
opts.go	1.3.12
rsaopts.go	1.3.13
bddtests	1.4
common	1.4.1
common_pb2.py	1.4.1.1
common_pb2_grpc.py	1.4.1.2
configtx_pb2.py	1.4.1.3
configtx_pb2_grpc.py	1.4.1.4
configuration_pb2.py	1.4.1.5
configuration_pb2_grpc.py	1.4.1.6
ledger_pb2.py	1.4.1.7
ledger_pb2_grpc.py	1.4.1.8
policies_pb2.py	1.4.1.9
policies_pb2_grpc.py	1.4.1.10
features	1.4.2

bootstrap.feature	1.4.2.1
endorser.feature	1.4.2.2
orderer.feature	1.4.2.3
msp	1.4.3
identities_pb2.py	1.4.3.1
identities_pb2_grpc.py	1.4.3.2
msp_config_pb2.py	1.4.3.3
msp_config_pb2_grpc.py	1.4.3.4
msp_principal_pb2.py	1.4.3.5
msp_principal_pb2_grpc.py	1.4.3.6
orderer	1.4.4
ab_pb2.py	1.4.4.1
ab_pb2_grpc.py	1.4.4.2
configuration_pb2.py	1.4.4.3
configuration_pb2_grpc.py	1.4.4.4
kafka_pb2.py	1.4.4.5
kafka_pb2_grpc.py	1.4.4.6
peer	1.4.5
admin_pb2.py	1.4.5.1
admin_pb2_grpc.py	1.4.5.2
chaincode_event_pb2.py	1.4.5.3
chaincode_event_pb2_grpc.py	1.4.5.4
chaincode_pb2.py	1.4.5.5
chaincode_pb2_grpc.py	1.4.5.6
chaincode_shim_pb2.py	1.4.5.7
chaincode_shim_pb2_grpc.py	1.4.5.8
configuration_pb2.py	1.4.5.9
configuration_pb2_grpc.py	1.4.5.10
events_pb2.py	1.4.5.11
events_pb2_grpc.py	1.4.5.12
peer_pb2.py	1.4.5.13
peer_pb2_grpc.py	1.4.5.14
proposal_pb2.py	1.4.5.15

proposal_pb2_grpc.py	1.4.5.16
proposal_response_pb2.py	1.4.5.17
proposal_response_pb2_grpc.py	1.4.5.18
query_pb2.py	1.4.5.19
query_pb2_grpc.py	1.4.5.20
transaction_pb2.py	1.4.5.21
transaction_pb2_grpc.py	1.4.5.22
regression	1.4.6
go	1.4.6.1
ote	1.4.6.1.1
tdk	1.4.6.1.2
node	1.4.6.2
performance	1.4.6.2.1
results	1.4.6.3
daily_test_suite.sh	1.4.6.4
longrun_test_suite.sh	1.4.6.5
scripts	1.4.7
wait-for-it.sh	1.4.7.1
steps	1.4.8
bdd_grpc_util.py	1.4.8.1
bdd_test_util.py	1.4.8.2
bootstrap_impl.py	1.4.8.3
bootstrap_util.py	1.4.8.4
compose.py	1.4.8.5
contexthelper.py	1.4.8.6
coverage.py	1.4.8.7
docgen.py	1.4.8.8
endorser_impl.py	1.4.8.9
endorser_util.py	1.4.8.10
orderer_impl.py	1.4.8.11
orderer_util.py	1.4.8.12
templates	1.4.9
html	1.4.9.1
appendix-py.html	1.4.9.1.1

cli.html	1.4.9.1.2
composition-py.html	1.4.9.1.3
directory-py.html	1.4.9.1.4
directory.html	1.4.9.1.5
error.html	1.4.9.1.6
graph.html	1.4.9.1.7
header.html	1.4.9.1.8
main.html	1.4.9.1.9
org-py.html	1.4.9.1.10
org.html	1.4.9.1.11
protobuf-py.html	1.4.9.1.12
protobuf.html	1.4.9.1.13
report.css	1.4.9.1.14
scenario.html	1.4.9.1.15
step.html	1.4.9.1.16
tag.html	1.4.9.1.17
user.html	1.4.9.1.18
chaincode.go	1.4.10
compose.go	1.4.11
conn.go	1.4.12
context.go	1.4.13
context_bootstrap.go	1.4.14
context_endorser.go	1.4.15
dc-base.yml	1.4.16
dc-orderer-base.yml	1.4.17
dc-orderer-kafka-base.yml	1.4.18
dc-orderer-kafka.yml	1.4.19
dc-peer-base.yml	1.4.20
dc-peer-couchdb.yml	1.4.21
docker.go	1.4.22
environment.py	1.4.23
tlsca.cert	1.4.24
tlsca.priv	1.4.25

users.go	1.4.26
util.go	1.4.27
common	1.5
cauthdsl	1.5.1
cauthdsl.go	1.5.1.1
cauthdsl_builder.go	1.5.1.2
policy.go	1.5.1.3
policy_util.go	1.5.1.4
policyparser.go	1.5.1.5
config	1.5.2
msp	1.5.2.1
config.go	1.5.2.1.1
config_util.go	1.5.2.1.2
api.go	1.5.2.2
application.go	1.5.2.3
application_util.go	1.5.2.4
applicationorg.go	1.5.2.5
channel.go	1.5.2.6
channel_util.go	1.5.2.7
consortium.go	1.5.2.8
consortiums.go	1.5.2.9
consortiums_util.go	1.5.2.10
orderer.go	1.5.2.11
orderer_util.go	1.5.2.12
organization.go	1.5.2.13
proposer.go	1.5.2.14
root.go	1.5.2.15
standardvalues.go	1.5.2.16
configtx	1.5.3
api	1.5.3.1
api.go	1.5.3.1.1
test	1.5.3.2
helper.go	1.5.3.2.1
compare.go	1.5.3.3

config.go	1.5.3.4
configmap.go	1.5.3.5
initializer.go	1.5.3.6
manager.go	1.5.3.7
template.go	1.5.3.8
update.go	1.5.3.9
util.go	1.5.3.10
crypto	1.5.4
random.go	1.5.4.1
signer.go	1.5.4.2
errors	1.5.5
codes.go	1.5.5.1
errors.go	1.5.5.2
flogging	1.5.6
grpclogger.go	1.5.6.1
logging.go	1.5.6.2
genesis	1.5.7
genesis.go	1.5.7.1
ledger	1.5.8
blkstorage	1.5.8.1
fsblkstorage	1.5.8.1.1
blockstorage.go	1.5.8.1.2
testutil	1.5.8.2
test_helper.go	1.5.8.2.1
test_util.go	1.5.8.2.2
util	1.5.8.3
leveldbhelper	1.5.8.3.1
ioutil.go	1.5.8.3.2
protobuf_util.go	1.5.8.3.3
util.go	1.5.8.3.4
ledger_interface.go	1.5.8.4
localmsp	1.5.9
signer.go	1.5.9.1

metadata	1.5.10
metadata.go	1.5.10.1
mocks	1.5.11
config	1.5.11.1
channel.go	1.5.11.1.1
orderer.go	1.5.11.1.2
configtx	1.5.11.2
configtx.go	1.5.11.2.1
crypto	1.5.11.3
localsigner.go	1.5.11.3.1
ledger	1.5.11.4
queryexecutor.go	1.5.11.4.1
msp	1.5.11.5
noopmsp.go	1.5.11.5.1
peer	1.5.11.6
mockccostream.go	1.5.11.6.1
mockpeerccsupport.go	1.5.11.6.2
policies	1.5.11.7
policies.go	1.5.11.7.1
scc	1.5.11.8
scprovider.go	1.5.11.8.1
policies	1.5.12
implicitmeta.go	1.5.12.1
implicitmeta_util.go	1.5.12.2
policy.go	1.5.12.3
tools	1.5.13
configtxgen	1.5.13.1
localconfig	1.5.13.1.1
metadata	1.5.13.1.2
provisional	1.5.13.1.3
main.go	1.5.13.1.4
configtxlator	1.5.13.2
metadata	1.5.13.2.1
rest	1.5.13.2.2

sanitycheck	1.5.13.2.3
update	1.5.13.2.4
main.go	1.5.13.2.5
cryptogen	1.5.13.3
ca	1.5.13.3.1
csp	1.5.13.3.2
metadata	1.5.13.3.3
msp	1.5.13.3.4
main.go	1.5.13.3.5
protolator	1.5.13.4
testprotos	1.5.13.4.1
api.go	1.5.13.4.2
dynamic.go	1.5.13.4.3
json.go	1.5.13.4.4
nested.go	1.5.13.4.5
statically_opaque.go	1.5.13.4.6
variably_opaque.go	1.5.13.4.7
util	1.5.14
utils.go	1.5.14.1
viperutil	1.5.15
config_util.go	1.5.15.1
core	1.6
chaincode	1.6.1
accesscontrol	1.6.1.1
ca.go	1.6.1.1.1
key.go	1.6.1.1.2
mapper.go	1.6.1.1.3
platforms	1.6.1.2
car	1.6.1.2.1
golang	1.6.1.2.2
java	1.6.1.2.3
util	1.6.1.2.4
platforms.go	1.6.1.2.5

shim	1.6.1.3
java	1.6.1.3.1
chaincode.go	1.6.1.3.2
handler.go	1.6.1.3.3
inprocstream.go	1.6.1.3.4
interfaces.go	1.6.1.3.5
mockstub.go	1.6.1.3.6
response.go	1.6.1.3.7
testdata	1.6.1.4
server1.key	1.6.1.4.1
server1.pem	1.6.1.4.2
ccproviderimpl.go	1.6.1.5
chaincode_support.go	1.6.1.6
chaincodeexec.go	1.6.1.7
chaincodetest.yaml	1.6.1.8
exectransaction.go	1.6.1.9
handler.go	1.6.1.10
comm	1.6.2
testdata	1.6.2.1
certs	1.6.2.1.1
grpc	1.6.2.1.2
impersonation	1.6.2.1.3
prime256v1-openssl-cert.pem	1.6.2.1.4
prime256v1-openssl-key.pem	1.6.2.1.5
config.go	1.6.2.2
connection.go	1.6.2.3
creds.go	1.6.2.4
producer.go	1.6.2.5
server.go	1.6.2.6
committer	1.6.3
txvalidator	1.6.3.1
validator.go	1.6.3.1.1
committer.go	1.6.3.2
committer_impl.go	1.6.3.3

common	1.6.4
ccpackage	1.6.4.1
ccpackage.go	1.6.4.1.1
ccprovider	1.6.4.2
ccinfocache.go	1.6.4.2.1
ccprovider.go	1.6.4.2.2
cdspackage.go	1.6.4.2.3
sigcdspackage.go	1.6.4.2.4
sysccprovider	1.6.4.3
sysccprovider.go	1.6.4.3.1
validation	1.6.4.4
msgvalidation.go	1.6.4.4.1
config	1.6.5
config.go	1.6.5.1
container	1.6.6
api	1.6.6.1
core.go	1.6.6.1.1
ccintf	1.6.6.2
ccintf.go	1.6.6.2.1
dockercontroller	1.6.6.3
dockercontroller.go	1.6.6.3.1
inproccontroller	1.6.6.4
inproccontroller.go	1.6.6.4.1
inprocstream.go	1.6.6.4.2
msp	1.6.6.5
sampleconfig	1.6.6.5.1
util	1.6.6.6
dockerutil.go	1.6.6.6.1
writer.go	1.6.6.6.2
controller.go	1.6.6.7
vm.go	1.6.6.8
deliverservice	1.6.7
blocksprovider	1.6.7.1

	blocksprovider.go	1.6.7.1.1
mocks		1.6.7.2
	blocksprovider.go	1.6.7.2.1
	orderer.go	1.6.7.2.2
client.go		1.6.7.3
deliveryclient.go		1.6.7.4
requester.go		1.6.7.5
endorser		1.6.8
	endorser.go	1.6.8.1
ledger		1.6.9
kvledger		1.6.9.1
	example	1.6.9.1.1
	history	1.6.9.1.2
	marble_example	1.6.9.1.3
	txmgmt	1.6.9.1.4
	kv_ledger.go	1.6.9.1.5
	kv_ledger_provider.go	1.6.9.1.6
	recovery.go	1.6.9.1.7
ledgerconfig		1.6.9.2
	ledger_config.go	1.6.9.2.1
ledgerngmt		1.6.9.3
	ledger_mgmt.go	1.6.9.3.1
	ledger_mgmt_test_exports.go	1.6.9.3.2
testutil		1.6.9.4
	test_util.go	1.6.9.4.1
util		1.6.9.5
	couchdb	1.6.9.5.1
	txvalidationflags.go	1.6.9.5.2
	util.go	1.6.9.5.3
	ledger_interface.go	1.6.9.6
mocks		1.6.10
ccprovider		1.6.10.1
	ccprovider.go	1.6.10.1.1
txvalidator		1.6.10.2

	support.go	1.6.10.2.1
	validator	1.6.10.3
	validator.go	1.6.10.3.1
peer		1.6.11
	testdata	1.6.11.1
	generate.go	1.6.11.1.1
	Org1-cert.pem	1.6.11.1.2
	Org1-server1-cert.pem	1.6.11.1.3
	Org1-server1-key.pem	1.6.11.1.4
	Org2-cert.pem	1.6.11.1.5
	Org2-child1-cert.pem	1.6.11.1.6
	Org2-child1-key.pem	1.6.11.1.7
	Org2-child1-server1-cert.pem	1.6.11.1.8
	Org2-child1-server1-key.pem	1.6.11.1.9
	Org2-server1-cert.pem	1.6.11.1.10
	Org2-server1-key.pem	1.6.11.1.11
	Org3-cert.pem	1.6.11.1.12
	Org3-server1-cert.pem	1.6.11.1.13
	Org3-server1-key.pem	1.6.11.1.14
	config.go	1.6.11.2
	peer.go	1.6.11.3
policy		1.6.12
	mocks	1.6.12.1
	mocks.go	1.6.12.1.1
	policy.go	1.6.12.2
policyprovider		1.6.13
	provider.go	1.6.13.1
scc		1.6.14
	cscc	1.6.14.1
	configure.go	1.6.14.1.1
	escc	1.6.14.2
	endorser_onevalidsignature.go	1.6.14.2.1
lscc		1.6.14.3

lscc.go	1.6.14.3.1
qscC	1.6.14.4
query.go	1.6.14.4.1
samplesyscc	1.6.14.5
samplesyscc.go	1.6.14.5.1
vsCC	1.6.14.6
validator_onevalidsignature.go	1.6.14.6.1
importsyCCs.go	1.6.14.7
sccproviderimpl.go	1.6.14.8
sysccapi.go	1.6.14.9
testutil	1.6.15
config.go	1.6.15.1
admin.go	1.6.16
fsm.go	1.6.17
devenv	1.7
images	1.7.1
tools	1.7.2
couchdb	1.7.2.1
failure-motd.in	1.7.3
golang_buildcmd.sh	1.7.4
golang_buildpkg.sh	1.7.5
limits.conf	1.7.6
setup.sh	1.7.7
setupRHELonZ.sh	1.7.8
setupUbuntuOnPPC64le.sh	1.7.9
Vagrantfile	1.7.10
docs	1.8
custom_theme	1.8.1
searchbox.html	1.8.1.1
source	1.8.2
_static	1.8.2.1
css	1.8.2.1.1
_templates	1.8.2.2
footer.html	1.8.2.2.1

layout.html	1.8.2.2.2
dev-setup	1.8.2.3
build.rst	1.8.2.3.1
devenv.rst	1.8.2.3.2
headers.txt	1.8.2.3.3
Gerrit	1.8.2.4
best-practices.rst	1.8.2.4.1
changes.rst	1.8.2.4.2
gerrit.rst	1.8.2.4.3
lf-account.rst	1.8.2.4.4
reviewing.rst	1.8.2.4.5
images	1.8.2.5
Setup	1.8.2.6
TLSSetup.rst	1.8.2.6.1
Style-guides	1.8.2.7
go-style.rst	1.8.2.7.1
arch-deep-dive.rst	1.8.2.8
architecture.rst	1.8.2.9
blockchain.rst	1.8.2.10
build_network.rst	1.8.2.11
capabilities.rst	1.8.2.12
chaincode.rst	1.8.2.13
chaincode4ade.rst	1.8.2.14
chaincode4noah.rst	1.8.2.15
channels.rst	1.8.2.16
conf.py	1.8.2.17
configtx.rst	1.8.2.18
configtxgen.rst	1.8.2.19
configtxlator.rst	1.8.2.20
CONTRIBUTING.rst	1.8.2.21
DCO1.1.txt	1.8.2.22
endorsement-policies.rst	1.8.2.23
error-handling.rst	1.8.2.24

Fabric-FAQ.rst	1.8.2.25
fabric-sdks.rst	1.8.2.26
fabric_model.rst	1.8.2.27
getting_started.rst	1.8.2.28
glossary.rst	1.8.2.29
gossip.rst	1.8.2.30
index.rst	1.8.2.31
install_instantiate.rst	1.8.2.32
jira_navigation.rst	1.8.2.33
kafka.rst	1.8.2.34
ledger.rst	1.8.2.35
logging-control.rst	1.8.2.36
MAINTAINERS.rst	1.8.2.37
mdtorst.sh	1.8.2.38
msp-identity-validity-rules.rst	1.8.2.39
msp.rst	1.8.2.40
peer-chaincode-devmode.rst	1.8.2.41
policies.rst	1.8.2.42
prereqs.rst	1.8.2.43
questions.rst	1.8.2.44
readwrite.rst	1.8.2.45
releases.rst	1.8.2.46
requirements.txt	1.8.2.47
samples.rst	1.8.2.48
security_model.rst	1.8.2.49
smartcontract.rst	1.8.2.50
status.rst	1.8.2.51
testing.rst	1.8.2.52
txflow.rst	1.8.2.53
usecases.rst	1.8.2.54
videos.rst	1.8.2.55
whyfabric.rst	1.8.2.56
write_first_app.rst	1.8.2.57
Makefile	1.8.3

requirements.txt	1.8.4
events	1.9
consumer	1.9.1
adapter.go	1.9.1.1
consumer.go	1.9.1.2
producer	1.9.2
eventhelper.go	1.9.2.1
events.go	1.9.2.2
handler.go	1.9.2.3
producer.go	1.9.2.4
register_internal_events.go	1.9.2.5
examples	1.10
ccchecker	1.10.1
chaincodes	1.10.1.1
newkeyperinvoke	1.10.1.1.1
chaincodes.go	1.10.1.1.2
registershadow.go	1.10.1.1.3
ccchecker.go	1.10.1.2
ccchecker.json	1.10.1.3
init.go	1.10.1.4
main.go	1.10.1.5
chaincode	1.10.2
chaintool	1.10.2.1
example02	1.10.2.1.1
go	1.10.2.2
chaincode_example01	1.10.2.2.1
chaincode_example02	1.10.2.2.2
chaincode_example03	1.10.2.2.3
chaincode_example04	1.10.2.2.4
chaincode_example05	1.10.2.2.5
eventsender	1.10.2.2.6
invokereturnsvalue	1.10.2.2.7
map	1.10.2.2.8

marbles02	1.10.2.2.9
passthru	1.10.2.2.10
sleeper	1.10.2.2.11
utxo	1.10.2.2.12
java	1.10.2.3
chaincode_example02	1.10.2.3.1
chaincode_example04	1.10.2.3.2
chaincode_example05	1.10.2.3.3
chaincode_example06	1.10.2.3.4
eventsender	1.10.2.3.5
Example	1.10.2.3.6
LinkExample	1.10.2.3.7
MapExample	1.10.2.3.8
RangeExample	1.10.2.3.9
SimpleSample	1.10.2.3.10
cluster	1.10.3
compose	1.10.3.1
peer-base	1.10.3.1.1
docker-compose.yaml	1.10.3.1.2
report-env.sh	1.10.3.1.3
config	1.10.3.2
configtx.yaml	1.10.3.2.1
core.yaml	1.10.3.2.2
cryptogen.yaml	1.10.3.2.3
fabric-ca-server-config.yaml	1.10.3.2.4
fabric-tlsca-server-config.yaml	1.10.3.2.5
orderer.yaml	1.10.3.2.6
configure.sh	1.10.3.3
Makefile	1.10.3.4
usage.txt	1.10.3.5
configtxupdate	1.10.4
bootstrap_batchsize	1.10.4.1
script.sh	1.10.4.1.1
common_scripts	1.10.4.2

common.sh	1.10.4.2.1
reconfig_batchsize	1.10.4.3
script.sh	1.10.4.3.1
reconfig_membership	1.10.4.4
script.sh	1.10.4.4.1
e2e_cli	1.10.5
base	1.10.5.1
docker-compose-base.yaml	1.10.5.1.1
peer-base.yaml	1.10.5.1.2
channel-artifacts	1.10.5.2
crypto-config	1.10.5.3
ordererOrganizations	1.10.5.3.1
peerOrganizations	1.10.5.3.2
examples	1.10.5.4
chaincode	1.10.5.4.1
scripts	1.10.5.5
script.sh	1.10.5.5.1
configtx.yaml	1.10.5.6
crypto-config.yaml	1.10.5.7
docker-compose-cli.yaml	1.10.5.8
docker-compose-couch.yaml	1.10.5.9
docker-compose-e2e-template.yaml	1.10.5.10
docker-compose-e2e.yaml	1.10.5.11
download-dockerimages.sh	1.10.5.12
end-to-end.rst	1.10.5.13
generateArtifacts.sh	1.10.5.14
network_setup.sh	1.10.5.15
events	1.10.6
block-listener	1.10.6.1
block-listener.go	1.10.6.1.1
gossip	1.11
api	1.11.1
channel.go	1.11.1.1

crypto.go	1.11.1.2
comm	1.11.2
mock	1.11.2.1
mock_comm.go	1.11.2.1.1
comm.go	1.11.2.2
comm_impl.go	1.11.2.3
conn.go	1.11.2.4
crypto.go	1.11.2.5
demux.go	1.11.2.6
msg.go	1.11.2.7
common	1.11.3
common.go	1.11.3.1
discovery	1.11.4
discovery.go	1.11.4.1
discovery_impl.go	1.11.4.2
election	1.11.5
adapter.go	1.11.5.1
election.go	1.11.5.2
filter	1.11.6
filter.go	1.11.6.1
gossip	1.11.7
algo	1.11.7.1
pull.go	1.11.7.1.1
channel	1.11.7.2
channel.go	1.11.7.2.1
msgstore	1.11.7.3
msgs.go	1.11.7.3.1
pull	1.11.7.4
pullstore.go	1.11.7.4.1
batcher.go	1.11.7.5
certstore.go	1.11.7.6
chanstate.go	1.11.7.7
gossip.go	1.11.7.8
gossip_impl.go	1.11.7.9

identity	1.11.8
identity.go	1.11.8.1
integration	1.11.9
integration.go	1.11.9.1
service	1.11.10
eventer.go	1.11.10.1
gossip_service.go	1.11.10.2
state	1.11.11
mocks	1.11.11.1
gossip.go	1.11.11.1.1
metastate.go	1.11.11.2
payloads_buffer.go	1.11.11.3
state.go	1.11.11.4
util	1.11.12
logging.go	1.11.12.1
misc.go	1.11.12.2
msgs.go	1.11.12.3
pubsub.go	1.11.12.4
gootools	1.12
Makefile	1.12.1
images	1.13
ccenv	1.13.1
Dockerfile.in	1.13.1.1
couchdb	1.13.2
docker-entrypoint.sh	1.13.2.1
Dockerfile.in	1.13.2.2
local.ini	1.13.2.3
vm.args	1.13.2.4
javaenv	1.13.3
Dockerfile.in	1.13.3.1
kafka	1.13.4
docker-entrypoint.sh	1.13.4.1
Dockerfile.in	1.13.4.2

	kafka-run-class.sh	1.13.4.3
orderer		1.13.5
	Dockerfile.in	1.13.5.1
peer		1.13.6
	Dockerfile.in	1.13.6.1
testenv		1.13.7
	Dockerfile.in	1.13.7.1
	install-softhsm2.sh	1.13.7.2
tools		1.13.8
	Dockerfile.in	1.13.8.1
zookeeper		1.13.9
	docker-entrypoint.sh	1.13.9.1
	Dockerfile.in	1.13.9.2
msp		1.14
mgmt		1.14.1
	testtools	1.14.1.1
	config.go	1.14.1.1.1
	deserializer.go	1.14.1.2
	mgmt.go	1.14.1.3
	principal.go	1.14.1.4
testdata		1.14.2
	badadmin	1.14.2.1
	admincerts	1.14.2.1.1
	cacerts	1.14.2.1.2
	keystore	1.14.2.1.3
	signcerts	1.14.2.1.4
	config.yaml	1.14.2.1.5
	badconfigou	1.14.2.2
	admincerts	1.14.2.2.1
	cacerts	1.14.2.2.2
	keystore	1.14.2.2.3
	signcerts	1.14.2.2.4
	config.yaml	1.14.2.2.5
	badconfigoucert	1.14.2.3

admincerts	1.14.2.3.1
cacerts	1.14.2.3.2
keystore	1.14.2.3.3
signcerts	1.14.2.3.4
config.yaml	1.14.2.3.5
external	1.14.2.4
admincerts	1.14.2.4.1
cacerts	1.14.2.4.2
intermediatecerts	1.14.2.4.3
keystore	1.14.2.4.4
signcerts	1.14.2.4.5
config.yaml	1.14.2.4.6
intermediate	1.14.2.5
admincerts	1.14.2.5.1
cacerts	1.14.2.5.2
intermediatecerts	1.14.2.5.3
keystore	1.14.2.5.4
signcerts	1.14.2.5.5
intermediate2	1.14.2.6
admincerts	1.14.2.6.1
cacerts	1.14.2.6.2
intermediatecerts	1.14.2.6.3
keystore	1.14.2.6.4
signcerts	1.14.2.6.5
users	1.14.2.6.6
mspid	1.14.2.7
admincerts	1.14.2.7.1
cacerts	1.14.2.7.2
keystore	1.14.2.7.3
signcerts	1.14.2.7.4
tlscacerts	1.14.2.7.5
revocation	1.14.2.8
admincerts	1.14.2.8.1

cacerts	1.14.2.8.2
crls	1.14.2.8.3
keystore	1.14.2.8.4
signcerts	1.14.2.8.5
revocation2	1.14.2.9
admincerts	1.14.2.9.1
cacerts	1.14.2.9.2
crls	1.14.2.9.3
keystore	1.14.2.9.4
signcerts	1.14.2.9.5
revokedica	1.14.2.10
admincerts	1.14.2.10.1
cacerts	1.14.2.10.2
crls	1.14.2.10.3
intermediatecerts	1.14.2.10.4
keystore	1.14.2.10.5
signcerts	1.14.2.10.6
tls	1.14.2.11
admincerts	1.14.2.11.1
cacerts	1.14.2.11.2
intermediatecerts	1.14.2.11.3
keystore	1.14.2.11.4
signcerts	1.14.2.11.5
tlscacerts	1.14.2.11.6
tlsintermediatecerts	1.14.2.11.7
config.yaml	1.14.2.11.8
cert.go	1.14.3
configbuilder.go	1.14.4
identities.go	1.14.5
msp.go	1.14.6
mspimpl.go	1.14.7
mspmgrimpl.go	1.14.8
orderer	1.15
common	1.15.1

blockcutter	1.15.1.1
blockcutter.go	1.15.1.1.1
bootstrap	1.15.1.2
file	1.15.1.2.1
bootstrap.go	1.15.1.2.2
broadcast	1.15.1.3
broadcast.go	1.15.1.3.1
deliver	1.15.1.4
deliver.go	1.15.1.4.1
ledger	1.15.1.5
file	1.15.1.5.1
json	1.15.1.5.2
ram	1.15.1.5.3
ledger.go	1.15.1.5.4
util.go	1.15.1.5.5
localconfig	1.15.1.6
config.go	1.15.1.6.1
metadata	1.15.1.7
metadata.go	1.15.1.7.1
msgprocessor	1.15.1.8
filter.go	1.15.1.8.1
msgprocessor.go	1.15.1.8.2
sigfilter.go	1.15.1.8.3
sizefilter.go	1.15.1.8.4
standardchannel.go	1.15.1.8.5
systemchannel.go	1.15.1.8.6
systemchanelfilter.go	1.15.1.8.7
multichannel	1.15.1.9
chainsupport.go	1.15.1.9.1
registrar.go	1.15.1.9.2
performance	1.15.1.10
server.go	1.15.1.10.1
utils.go	1.15.1.10.2

server		1.15.1.11
main.go		1.15.1.11.1
server.go		1.15.1.11.2
util.go		1.15.1.11.3
consensus		1.15.2
kafka		1.15.2.1
chain.go		1.15.2.1.1
channel.go		1.15.2.1.2
config.go		1.15.2.1.3
conserter.go		1.15.2.1.4
partitioner.go		1.15.2.1.5
retry.go		1.15.2.1.6
solo		1.15.2.2
consensus.go		1.15.2.2.1
consensus.go		1.15.2.3
mocks		1.15.3
common		1.15.3.1
blockcutter		1.15.3.1.1
multichannel		1.15.3.1.2
util		1.15.3.2
util.go		1.15.3.2.1
sample_clients		1.15.4
broadcast_config		1.15.4.1
client.go		1.15.4.1.1
newchain.go		1.15.4.1.2
broadcast_timestamp		1.15.4.2
client.go		1.15.4.2.1
deliver_stdout		1.15.4.3
client.go		1.15.4.3.1
single_tx_client		1.15.4.4
single_tx_client.go		1.15.4.4.1
main.go		1.15.5
peer		1.16
chaincode		1.16.1

chaincode.go	1.16.1.1
common.go	1.16.1.2
install.go	1.16.1.3
instantiate.go	1.16.1.4
invoke.go	1.16.1.5
package.go	1.16.1.6
query.go	1.16.1.7
signpackage.go	1.16.1.8
upgrade.go	1.16.1.9
channel	1.16.2
channel.go	1.16.2.1
create.go	1.16.2.2
deliverclient.go	1.16.2.3
fetchconfig.go	1.16.2.4
join.go	1.16.2.5
list.go	1.16.2.6
signconfigtx.go	1.16.2.7
update.go	1.16.2.8
clilogging	1.16.3
common.go	1.16.3.1
getlevel.go	1.16.3.2
logging.go	1.16.3.3
revertlevels.go	1.16.3.4
setlevel.go	1.16.3.5
common	1.16.4
common.go	1.16.4.1
mockclient.go	1.16.4.2
ordererclient.go	1.16.4.3
gossip	1.16.5
mocks	1.16.5.1
mocks.go	1.16.5.1.1
mcs.go	1.16.5.2
sa.go	1.16.5.3

node	1.16.6
node.go	1.16.6.1
start.go	1.16.6.2
status.go	1.16.6.3
version	1.16.7
version.go	1.16.7.1
main.go	1.16.8
proposals	1.17
r1	1.17.1
protos	1.18
common	1.18.1
block.go	1.18.1.1
common.go	1.18.1.2
common.pb.go	1.18.1.3
common.proto	1.18.1.4
configtx.go	1.18.1.5
configtx.pb.go	1.18.1.6
configtx.proto	1.18.1.7
configuration.go	1.18.1.8
configuration.pb.go	1.18.1.9
configuration.proto	1.18.1.10
ledger.pb.go	1.18.1.11
ledger.proto	1.18.1.12
policies.go	1.18.1.13
policies.pb.go	1.18.1.14
policies.proto	1.18.1.15
signed_data.go	1.18.1.16
gossip	1.18.2
extensions.go	1.18.2.1
message.pb.go	1.18.2.2
message.proto	1.18.2.3
ledger	1.18.3
queryresult	1.18.3.1
kv_query_result.pb.go	1.18.3.1.1

kv_query_result.proto	1.18.3.1.2
rwset	1.18.3.2
kvrwset	1.18.3.2.1
tests	1.18.3.2.2
rwset.pb.go	1.18.3.2.3
rwset.proto	1.18.3.2.4
msp	1.18.4
identities.pb.go	1.18.4.1
identities.proto	1.18.4.2
msp_config.go	1.18.4.3
msp_config.pb.go	1.18.4.4
msp_config.proto	1.18.4.5
msp_principal.go	1.18.4.6
msp_principal.pb.go	1.18.4.7
msp_principal.proto	1.18.4.8
orderer	1.18.5
ab.pb.go	1.18.5.1
ab.proto	1.18.5.2
configuration.go	1.18.5.3
configuration.pb.go	1.18.5.4
configuration.proto	1.18.5.5
kafka.pb.go	1.18.5.6
kafka.proto	1.18.5.7
peer	1.18.6
admin.pb.go	1.18.6.1
admin.proto	1.18.6.2
chaincode.pb.go	1.18.6.3
chaincode.proto	1.18.6.4
chaincode_event.pb.go	1.18.6.5
chaincode_event.proto	1.18.6.6
chaincode_shim.pb.go	1.18.6.7
chaincode_shim.proto	1.18.6.8
chaincodeunmarshall.go	1.18.6.9

configuration.go	1.18.6.10
configuration.pb.go	1.18.6.11
configuration.proto	1.18.6.12
events.pb.go	1.18.6.13
events.proto	1.18.6.14
init.go	1.18.6.15
peer.pb.go	1.18.6.16
peer.proto	1.18.6.17
proposal.go	1.18.6.18
proposal.pb.go	1.18.6.19
proposal.proto	1.18.6.20
proposal_response.go	1.18.6.21
proposal_response.pb.go	1.18.6.22
proposal_response.proto	1.18.6.23
query.pb.go	1.18.6.24
query.proto	1.18.6.25
signed_cc_dep_spec.pb.go	1.18.6.26
signed_cc_dep_spec.proto	1.18.6.27
transaction.go	1.18.6.28
transaction.pb.go	1.18.6.29
transaction.proto	1.18.6.30
testutils	1.18.7
txtestutils.go	1.18.7.1
utils	1.18.8
blockutils.go	1.18.8.1
commonutils.go	1.18.8.2
proputils.go	1.18.8.3
txutils.go	1.18.8.4
release	1.19
templates	1.19.1
get-byfn.in	1.19.1.1
get-docker-images.in	1.19.1.2
release_notes	1.20
v1.0.0-rc1.txt	1.20.1

v1.0.0.txt	1.20.2
sampleconfig	1.21
msp	1.21.1
admincerts	1.21.1.1
admincert.pem	1.21.1.1.1
cacerts	1.21.1.2
cacert.pem	1.21.1.2.1
keystore	1.21.1.3
key.pem	1.21.1.3.1
signcerts	1.21.1.4
peer.pem	1.21.1.4.1
tlscacerts	1.21.1.5
cert.pem	1.21.1.5.1
config.yaml	1.21.1.6
configtx.yaml	1.21.2
core.yaml	1.21.3
orderer.yaml	1.21.4
scripts	1.22
bootstrap-1.0.0-alpha2.sh	1.22.1
bootstrap-1.0.0-beta.sh	1.22.2
bootstrap-1.0.0-rc1.sh	1.22.3
bootstrap-1.0.0.sh	1.22.4
bootstrap-1.0.1.sh	1.22.5
changelog.sh	1.22.6
check_license.sh	1.22.7
check_spelling.sh	1.22.8
compile_protos.sh	1.22.9
containerlogs.sh	1.22.10
foldercopy.sh	1.22.11
golinter.sh	1.22.12
goListFiles.sh	1.22.13
infiniteloop.sh	1.22.14
install_behave.sh	1.22.15

test		1.23
chaincodes		1.23.1
AuctionApp		1.23.1.1
art.go		1.23.1.1.1
image_proc_api.go		1.23.1.1.2
table_api.go		1.23.1.1.3
BadImport		1.23.1.2
main.go		1.23.1.2.1
envsetup		1.23.2
channel-artifacts		1.23.2.1
docker-compose.yaml		1.23.2.2
generateCfgTrx.sh		1.23.2.3
feature		1.23.3
configs		1.23.3.1
configtx.yaml		1.23.3.1.1
crypto.yaml		1.23.3.1.2
docker-compose		1.23.3.2
docker-compose-kafka.yml		1.23.3.2.1
docker-compose-solo.yml		1.23.3.2.2
steps		1.23.3.3
basic_impl.py		1.23.3.3.1
compose_util.py		1.23.3.3.2
config_util.py		1.23.3.3.3
endorser_impl.py		1.23.3.3.4
endorser_util.py		1.23.3.3.5
orderer_impl.py		1.23.3.3.6
orderer_util.py		1.23.3.3.7
bootstrap.feature		1.23.3.4
environment.py		1.23.3.5
orderer.feature		1.23.3.6
peer.feature		1.23.3.7
README.rst		1.23.3.8
regression		1.23.4
daily		1.23.4.1

chaincodeTests	1.23.4.1.1
Example.py	1.23.4.1.2
ledger_lte.py	1.23.4.1.3
README.rst.orig	1.23.4.1.4
runDailyTestSuite.sh	1.23.4.1.5
SampleScriptFailTest.sh	1.23.4.1.6
SampleScriptPassTest.sh	1.23.4.1.7
systest_pte.py	1.23.4.1.8
testAuctionChaincode.py	1.23.4.1.9
TestPlaceholder.sh	1.23.4.1.10
release	1.23.4.2
byfn_release_tests.py	1.23.4.2.1
e2e_sdk_release_tests.py	1.23.4.2.2
make_targets_release_tests.py	1.23.4.2.3
run_byfn_cli_release_tests.sh	1.23.4.2.4
run_e2e_java_sdk.sh	1.23.4.2.5
run_e2e_node_sdk.sh	1.23.4.2.6
run_make_targets.sh	1.23.4.2.7
run_node_sdk_byfn.sh	1.23.4.2.8
runReleaseTestSuite.sh	1.23.4.2.9
weekly	1.23.4.3
runGroup1.sh	1.23.4.3.1
runGroup2.sh	1.23.4.3.2
runGroup3.sh	1.23.4.3.3
runGroup4.sh	1.23.4.3.4
systest_pte.py	1.23.4.3.5
testAuctionChaincode.py	1.23.4.3.6
tools	1.23.5
AuctionApp	1.23.5.1
api_driver.sh	1.23.5.1.1
LTE	1.23.5.2
chainmgmt	1.23.5.2.1
common	1.23.5.2.2

experiments	1.23.5.2.3
scripts	1.23.5.2.4
OTE	1.23.5.3
README.rst	1.23.5.3.1
PTE	1.23.5.4
SCFiles	1.23.5.4.1
userInputs	1.23.5.4.2
chaincode_sample.go	1.23.5.4.3
pte-execRequest.js	1.23.5.4.4
pte-main.js	1.23.5.4.5
pte-util.js	1.23.5.4.6
pte_driver.sh	1.23.5.4.7
docker-compose.yml	1.23.6
unit-test	1.24
docker-compose.yml	1.24.1
run.sh	1.24.2

Hyperledger 源码分析之 Fabric

0.5.7

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 Fabric 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

- 0.6.0: 2017-XX-YY
 - configtxgen 模块；
 - cryptogen 模块；
 - 根据 fabric 1.0.* 调整结构。
- 0.5.0: 2017-05-08
 - 完成 peer 模块；
 - 按照最新代码结构更新；
 - 完成部分 orderer 模块分析。
- 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 membersrv，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 项目中主要包括代码、工具、脚本等部分，核心源代码目前约为 430 个文件，80K 行。

```
$ cd fabric
$ find bccsp common core events gossip msp orderer peer protos \
    -not -path "*/vendor/**" \
    -name "*.go" \
    -not -name "*_test.go" \
    | wc -l
431
$ find bccsp common core events gossip msp orderer peer protos \
    -not -path "*/vendor/**" \
    -name "*.go" \
    -not -name "*_test.go" \
    | xargs cat | wc -l
80560
```

源代码

实现 fabric 功能的核心代码，包括：

- **bccsp** 包：实现对加解密算法和机制的支持。
- **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；
- **gossip** 包：实现 gossip 协议；
- **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` 定义文件和生成的 `go` 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；
- `gootools`：`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`

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。

实现了软件机制（sw）和硬件机制（pkcs11）。

bccsp 主要支持 MSP 的相关调用。

factory

提供工厂模式支持，包括若干类型的 BCCSP 工厂实现。

通用工厂接口为：

```
type BCCSPFactory interface {  
  
    // Name returns the name of this factory  
    Name() string  
  
    // Get returns an instance of BCCSP using opts.  
    Get(opts *FactoryOpts) (bccsp.BCCSP, error)  
}
```

实现了两个工厂结构：

- **SWFactory**：软件 bccsp 的工厂；
- **PKCS11Factory**：基于高安全模块的实现。

factory.go

nopkcs11.go

opts.go

pkcs11.go

pkcs11factory.go

swfactory.go

mocks

mocks.go

pkcs11

conf.go

ecdsa.go

ecdsakey.go

impl.go

pkcs11.go

signer

signer

signer.go

SW

mocks

mocks.go

aes.go

aeskey.go

conf.go

dummyks.go

ecdsa.go

ecdsakey.go

fileks.go

hash.go

impl.go

internals.go

keyderiv.go

keygen.go

keyimport.go

rsa.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。

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

policies_pb2.py

policies_pb2_grpc.py

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

msp

identities_pb2.py

identities_pb2_grpc.py

msp_config_pb2.py

msp_config_pb2_grpc.py

msp_principal_pb2.py

msp_principal_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_event_pb2.py

chaincode_event_pb2_grpc.py

chaincode_pb2.py

chaincode_pb2_grpc.py

chaincode_shim_pb2.py

chaincode_shim_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

go

go

ote

go

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

appendix-py.html

cli.html

composition-py.html

directory-py.html

directory.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.go

conn.go

context.go

context_bootstrap.go

context_endorser.go

dc-base.yml

dc-orderer-base.yml

dc-orderer-kafka-base.yml

dc-orderer-kafka.yml

dc-peer-base.yml

dc-peer-couchdb.yml

docker.go

environment.py

tlsca.cert

tlsca.priv

users.go

util.go

common

一些通用的功能模块。

定义相应的接口和结构。

cauthdsl

实现 Policy 相关的数据结构和方法。

fabric 中，策略主要分为签名策略（SignaturePolicy）和隐式策略（ImplicitMetaPolicy）两种。

最常见的签名策略采用 SignaturePolicyEnvelope 结构来表达。定义在 [protos/common/policies.go](#) 中。

```
type SignaturePolicyEnvelope struct {
    Version      int32           `protobuf:"varint,1,opt,name=version" json:"version,omitempty"`
    Rule         *SignaturePolicy `protobuf:"bytes,2,opt,name=rule" json:"rule,omitempty"`
    Identities  []*common1.MSPPrincipal `protobuf:"bytes,3,rep,name=identities" json:"identities,omitempty"`
}
```

签名策略的规则为检查签名是否满足指定的 principal（特定个体、身份等）。

更通用的 Implicit 策略是一个递归结构，可以指定依赖其它策略，最终底层为签名策略。

cauthdsl.go

主要实现了 `func compile(policy *cb.SignaturePolicy, identities []*mb.MSPPrincipal, deserializer msp.IdentityDeserializer) (func([]*cb.SignedData, []bool) bool, error)` 方法。

该方法根据传入的策略结构，返回一个函数 `func([]*cb.SignedData, []bool) bool, error`，作为策略 `evaluate` 方法。

cauthdsl_builder.go

定义一些常见的策略。

包括：

- AcceptAllPolicy
- RejectAllPolicy

另外，提供了一些常用的构造方法，包括

- Envelope(policy *cb.SignaturePolicy*, identities *[]byte*) *cb.SignaturePolicyEnvelope*：快速构造一个策略信封结构。

一些策略构造方法：

- SignedBy(index int32) **cb.SignaturePolicy*：构造一条指定签名者的签名策略。
- SignedByMspMember(mspId string) **cb.SignaturePolicyEnvelope*：指定一条指定签名身份为某 MSP 成员（至少一个）的签名策略。
- SignedByMspAdmin(mspId string) **cb.SignaturePolicyEnvelope*：指定一条指定签名身份为某 MSP 管理员（至少一个）的签名策略。
- SignedByAnyMember(ids []string) **cb.SignaturePolicyEnvelope*：被给定组织列表中任意成员签名。
- SignedByAnyAdmin(ids []string) **cb.SignaturePolicyEnvelope*：被给定组织列表中任意管理员签名。

一些策略的组合逻辑（与、或、集合中的若干个）方法：

- And(lhs, rhs *cb.SignaturePolicy*) *cb.SignaturePolicy*：两个策略的与组合。
- Or(lhs, rhs *cb.SignaturePolicy*) *cb.SignaturePolicy*：两个策略的或组合。
- NOutOf(n int32, policies []*cb.SignaturePolicy*) *cb.SignaturePolicy*：集合中的至少若干个。

policy.go

实现上，一条 policy 结构实现一个 evaluator 方法。

```
type policy struct {
    evaluator func([]*cb.SignedData, []bool) bool
}
```

对外提供的方法是 Evaluate() 方法，调用了 evaluator() 私有方法。

```
func (p *policy) Evaluate(signatureSet []*cb.SignedData) error {
    if p == nil {
        return fmt.Errorf("No such policy")
    }

    ok := p.evaluator(signatureSet, make([]bool, len(signatureSet)))
    if !ok {
        return errors.New("Failed to authenticate policy")
    }
    return nil
}
```

同时，提供了 NewPolicyProvider(deserializer msp.IdentityDeserializer) policies.Provider 方
法，作为工厂方法。

policy_util.go

policyparser.go

主要提供了 `FromString(policy string) (*common.SignaturePolicyEnvelope, error)` 方法，将字符串指定的策略生成 Policy 信封结构。

config 包

通道相关的配置，主要是声明一些接口和结构。

msp 包

msp 配置相关。

config.go

config_util.go

提供一些辅助方法，如：

- `TemplateGroupMSPWithAdminRolePrincipal(configPath []string, mspConfig mspprotos.MSPConfig, admin bool) cb.ConfigGroup`：创建在指定 config 路径上的 config 项，以及该项相关的 Admins、Readers、Writers 的策略。
- `TemplateGroupMSP(configPath []string, mspConfig mspprotos.MSPConfig)` `cb.ConfigGroup`：在指定配置路径上创建 MSP 配置值，调用的 `TemplateGroupMSPWithAdminRolePrincipal()` 方法。

api.go

定义了一系列的配置中相关的项的接口，包括：

- Application
- ApplicationOrg
- Channel
- Consortium
- Consortiums
- Orderer
- Org
- ValueProposer

application.go

应用配置相关的数据结构。

application_util.go

applicationorg.go

主要提供三个重要的数据结构，代表应用组织。

```
type ApplicationOrgProtos struct {
    AnchorPeers *pb.AnchorPeers
}

type ApplicationOrgConfig struct {
    *OrganizationConfig
    protos *ApplicationOrgProtos

    applicationOrgGroup *ApplicationOrgGroup
}

// ApplicationOrgGroup defines the configuration for an application org
type ApplicationOrgGroup struct {
    *Proposer
    *OrganizationGroup
    *ApplicationOrgConfig
}
```

channel.go

通道相关的配置。

channel_util.go

consortium.go

consortiums.go

consortiums_util.go

orderer.go

orderer_util.go

organization.go

包括几个重要的数据结构和它们的操作方法。

- **OrganizationProtos**：定义一个 MSP 配置，包括这个 MSP 的证书、签名、验证等需要的信息。实际上就代表了一个 MSP 的完整信息。
- **OrganizationConfig**：代表组织的配置，不仅包括 MSP 信息，还包括配置组的引用。
- **OrganizationGroup**：组合自

```
type OrganizationProtos struct {
    MSP *mspprotos.MSPConfig
}

type OrganizationConfig struct {
    *standardValues
    protos *OrganizationProtos

    organizationGroup *OrganizationGroup

    msp    msp.MSP
    mspID string
}

// Config stores common configuration information for organizations
type OrganizationGroup struct {
    *Proposer
    *OrganizationConfig
    name      string
    mspConfigHandler *mspconfig.MSPConfigHandler
}
```

proposer.go

root.go

standardvalues.go

configtx

负责 config transaction（新建或更新）相关的数据结构和处理。

api

定义一些重要的接口。

- **Resources**：代表通道配置资源的接口，支持获取通道配置的一些方法。
- **Proposer**：代表对配置进行修改的接口，包括 `ValueProposer` 和 `PolicyProposer` 两个成员。

api.go

test

test

helper.go

compare.go

config.go

configmap.go

initializer.go

manager.go

template.go

update.go

util.go

crypto

主要定义了 LocalSigner 接口。

random.go

signer.go

errors

错误相关代码。

codes.go

errors.go

flogging

grpclogger.go

logging.go

genesis

提供初始区块的工厂接口和实现，提供生成的方法。

genesis.go

ledger

账本结构的接口和实现。

blkstorage 包

链结构存放在本地文件系统上。由 **blkstorage** 包负责提供底层的支持。

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

对主要模块们提供 mock 包。

config

channel.go

orderer.go

configtx

configtx.go

crypto

localsigner.go

ledger

queryexecutor.go

msp

noopmsp.go

peer

mockccstream.go

mockpeerccsupport.go

policies

policies.go

SCC

sccprovider.go

policies

策略的处理。

策略目前包括 ImplicitMetaPolicy 和 SignaturePolicy 两种。

- ImplicitMetaPolicy：基于其它策略（往往是子空间中策略）的结果来判断。
- SignaturePolicy：基于签名的策略，签名集合中某种特定组合符合特定条件。如至少 3 个签名，其中 1 个管理员，2 个成员。

```
type implicitMetaPolicy struct {
    conf      *cb.ImplicitMetaPolicy
    threshold int
    subPolicies []Policy
}
```

implicitmeta.go

implicitmeta_util.go

policy.go

主要定义了三个策略相关接口和实现了这三个接口的结构。

- ChannelPolicyManagerGetter：获取指定通道的策略管理器。
- Manager：一个通道相关策略的管理器。
- Proposer：负责处理策略更新的相关操作。

ManagerImpl 结构实现了这三个接口，成为策略相关操作的主要入口结构。

```
type ManagerImpl struct {
    parent      *ManagerImpl
    basePath    string
    fqPrefix    string
    providers   map[int32]Provider
    config      *policyConfig
    pendingConfig map[interface{}]*policyConfig
    pendingLock  sync.RWMutex

    // SuppressSanityLogMessages when set to true will prevent the sanity checking log
    // messages. Useful for novel cases like channel templates
    SuppressSanityLogMessages bool
}
```

tools

几个有用的工具，包括 configtxlator 和 cryptogen 等。

tool

localconfig

metadata 包

provisional

configtxgen

configtxgen 工具是一个很重要的离线辅助工具，它的主要功能有如下几个：

- 生成启动 orderer 需要的初始区块，并检查区块内容；
- 生成创建 channel 需要的配置交易，并检查交易内容；
- 生成锚点 Peer 的更新配置信息。

默认情况下，configtxgen 工具会依次尝试从 `$FABRIC_CFG_PATH` 环境变量指定的路径，当前路径和 `/etc/hyperledger/fabric` 路径下查找 configtx.yaml 配置文件并读入。

main.go 文件为入口。

核心代码十分简单：

- 首先通过 `factory.InitFactories(nil)` 读入 BCCSP 配置；
- 通过 `config := genesisconfig.Load(profile)` 读入 configtx.yaml 配置文件；
- 然后根据指定的参数分别进行相应的操作。

```
func main() {
    var outputBlock, outputChannelCreateTx, profile, channelID, inspectBlock, inspectC
    hannelCreateTx, outputAnchorPeersUpdate, asOrg string

    flag.StringVar(&outputBlock, "outputBlock", "", "The path to write the genesis blo
    ck to (if set)")
    flag.StringVar(&channelID, "channelID", provisional.TestChainID, "The channel ID t
    o use in the configtx")
    flag.StringVar(&outputChannelCreateTx, "outputCreateChannelTx", "", "The path to w
    rite a channel creation configtx to (if set)")
    flag.StringVar(&profile, "profile", genesisconfig.SampleInsecureProfile, "The prof
    ile from configtx.yaml to use for generation.")
    flag.StringVar(&inspectBlock, "inspectBlock", "", "Prints the configuration contai
    ned in the block at the specified path")
    flag.StringVar(&inspectChannelCreateTx, "inspectChannelCreateTx", "", "Prints the
    configuration contained in the transaction at the specified path")
    flag.StringVar(&outputAnchorPeersUpdate, "outputAnchorPeersUpdate", "", "Creates a
    n config update to update an anchor peer (works only with the default channel creation
    , and only for the first update)")
    flag.StringVar(&asOrg, "asOrg", "", "Performs the config generation as a particula
    r organization, only including values in the write set that org (likely) has privilege
    to set")

    flag.Parse()

    logging.SetLevel(logging.INFO, "")

    logger.Info("Loading configuration")
    factory.InitFactories(nil)
    config := genesisconfig.Load(profile)
```

```

    if outputBlock != "" {
        if err := doOutputBlock(config, channelID, outputBlock); err != nil {
            logger.Fatalf("Error on outputBlock: %s", err)
        }
    }

    if outputChannelCreateTx != "" {
        if err := doOutputChannelCreateTx(config, channelID, outputChannelCreateTx); err != nil {
            logger.Fatalf("Error on outputChannelCreateTx: %s", err)
        }
    }

    if inspectBlock != "" {
        if err := doInspectBlock(inspectBlock); err != nil {
            logger.Fatalf("Error on inspectBlock: %s", err)
        }
    }

    if inspectChannelCreateTx != "" {
        if err := doInspectChannelCreateTx(inspectChannelCreateTx); err != nil {
            logger.Fatalf("Error on inspectChannelCreateTx: %s", err)
        }
    }

    if outputAnchorPeersUpdate != "" {
        if err := doOutputAnchorPeersUpdate(config, channelID, outputAnchorPeersUpdate, asOrg); err != nil {
            logger.Fatalf("Error on inspectChannelCreateTx: %s", err)
        }
    }
}

```

doOutputBlock

生成初始区块，存放到指定路径。

- 首先调用 `provisional.New(config)` 根据读入的配置生成启动参数结构。
- 调用 `genesisBlock := pgen.GenesisBlockForChannel(channelID)` 生成指定通道的区块，核心数据是一个 `ConfigEnvelope` 结构。
- 将区块文件写到本地。

doOutputChannelCreateTx

生成创建新通道的交易，存放到指定路径。

- 获取签名实体信息。
- 生成创建新通道的交易结构。
- 写入到本地文件。

doInspectBlock

读入区块内容，并解析为 ConfigEnvelope 结构，打印出来。

doInspectChannelCreateTx

读入交易文件内容，并解析为 ConfigUpdateEnvelope 结构，打印出来。

doOutputAnchorPeersUpdate

生成一个锚点 Peer 更新的 ConfigUpdateEnvelope，并存储到指定路径。

configtxlator

metadata

rest

sanitycheck

update

main.go

cryptogen

在 Fabric 中，通过证书和密钥来管理身份，经常需要进行证书生成和配置操作。

使用 cryptogen 工具，可以快速根据配置自动生成所需要的密钥和证书文件，或者查看配置模板信息。

入口代码在 main.go。

ca

csp

metadata

msp

main.go

核心代码十分简单。

```
//command line flags
var (
    app = kingpin.New("cryptogen", "Utility for generating Hyperledger Fabric key mate
rial")

    gen      = app.Command("generate", "Generate key material")
    outputDir = gen.Flag("output", "The output directory in which to place artifacts"
).Default("crypto-config").String()
    configFile = gen.Flag("config", "The configuration template to use").File()

    showtemplate = app.Command("showtemplate", "Show the default configuration templat
e")
)

func main() {
    kingpin.Version("0.0.1")
    switch kingpin.MustParse(app.Parse(os.Args[1:])) {

        // "generate" command
        case gen.FullCommand():
            generate()

        // "showtemplate" command
        case showtemplate.FullCommand():
            fmt.Print(defaultConfig)
            os.Exit(0)
    }
}
```

generate

主要调用 `generate()` 方法。

- 首先调用 `getConfig()` 读取本地配置或使用默认配置。
- 使用 `renderOrgSpec()` 和 `generatePeerOrg()` 方法依次生成每个 `peer` 类型的组织。
- 使用 `renderOrgSpec()` 和 `generateOrdererOrg()` 方法依次生成每个 `orderer` 类型的组织。

`renderOrgSpec()` 负责将模板中具体组织数据解析出来。

peer 组织

`generatePeerOrg()` 会具体生成每个 `peer` 类型的组织，结构如下。

peerOrganizations :

- org1 : 存放第一个组织的相关材料，每个组织会生成单独的根证书。
 - ca : 存放组织的根证书和对应的私钥文件，默认采用 EC 算法，证书为自签名。组织内的实体将基于该根证书作为相同的证书根。
 - msp : 存放代表该组织的身份信息
 - admincerts : 组织管理员的身份验证证书。
 - cacerts : 组织的根证书。
 - keystore : 组织的私钥文件，用来签名。
 - signcerts : 组织的签名验证证书。
 - peers : 存放该组织下的所有 peer 节点
 - peer1 : 第一个 peer 的信息，包括 msp 证书和 tls 证书两类。
 - msp :
 - admincerts : 组织管理员的身份验证证书。
 - cacerts : 存放组织的根证书。
 - keystore : 本节点的身份私钥，用来签名。
 - signcerts : 验证本节点签名的证书，被组织根证书签名。
 - tls : 存放 tls 相关的证书和私钥
 - ca.crt : 组织的根证书。
 - server.crt : 验证本节点签名的证书，被组织根证书签名。
 - server.key : 本节点的身份私钥，用来签名。
 - users : 存放属于该组织的用户的实体。
 - Admin : 管理员用户的信息，包括 msp 证书和 tls 证书两类。
 - msp :
 - admincerts : 组织根证书作为管理者身份验证证书。
 - cacerts : 存放组织的根证书。
 - keystore : 本用户的身份私钥，用来签名。
 - signcerts : 管理员用户的身份验证证书，被组织根证书签名。
 - tls : 存放 tls 相关的证书和私钥
 - ca.crt : 组织的根证书。
 - server.crt : 管理员的用户身份验证证书，被组织根证书签名。
 - server.key : 管理员用户的身份私钥，用来签名。
 - User1 : 第一个用户的信息，包括 msp 证书和 tls 证书两类。
 - msp :
 - admincerts : 组织根证书作为管理者身份验证证书。
 - cacerts : 存放组织的根证书。
 - keystore : 本用户的身份私钥，用来签名。
 - signcerts : 验证本用户签名的身份证件证书，被组织根证书签名。
 - tls : 存放 tls 相关的证书和私钥
 - ca.crt : 组织的根证书。
 - server.crt : 验证用户签名的身份证件证书，被组织根证书签名。

- `server.key`：用户的身份私钥，用来签名。

- `org2`

- ...

注意每个实体（组织、节点、用户）最终都会拥有 MSP 来代表身份信息，其中包括三种证书：管理员身份的验证证书、实体信任的 CA 的根证书、自身身份验证（检查签名）的证书。此外，还包括对应身份验证的签名用的私钥。

`GenerateVerifyingMSP()`会生成组织的 MSP 信息。

`generateNodes` 会针对每个实体生成所需要的 `msp` 和 `tls` 信息，证书都会被组织根证书签名，信任的根都是组织 CA 证书。

orderer 组织

跟 `peer` 组织类似过程。

protolator

testprotos

api.go

dynamic.go

json.go

nested.go

statically_opaque.go

variably_opaque.go

util 包

一些辅助方法，包括计算各种 Hash，获取默认链和组织 ID 等。

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。

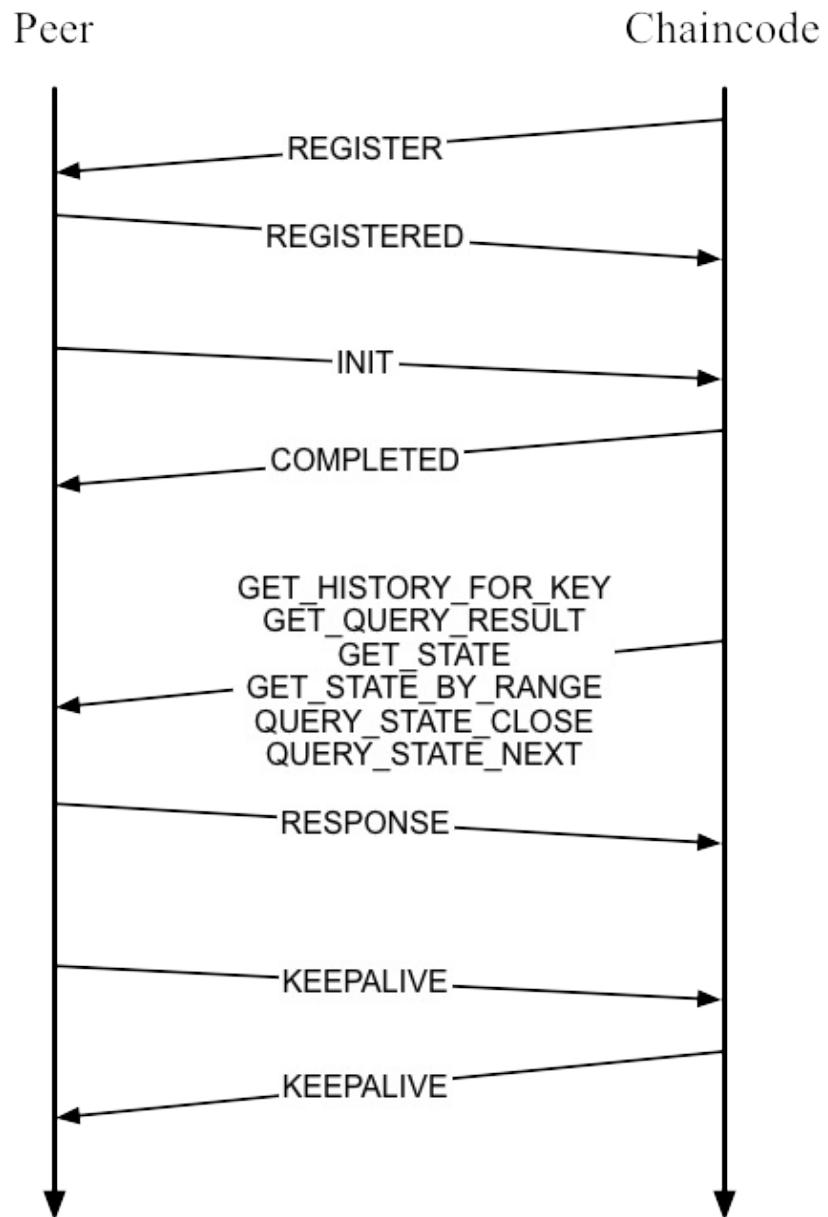
Peer 侧比较核心的结构包括：

- 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：cc 容器启动后，会发送注册消息让 peer 创建一个 handler 结构，并进入主循环响应消息。peer 侧通过一个状态机来维护对于 chaincode 各种消息的响应，利用 before、after 等触发条件。

shim 包则提供 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。



accesscontrol 包

ca.go

key.go

mapper.go

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 进行交互的一系列方法，例如 PutState 与 GetState 来写入和查询链上键值对的状态。

用户链码 --> ChaincodeStub --> Handler -----ChaincodeMessage -----> Peer

java

chaincode.go

提供 ChaincodeStub 结构，支持一系列对账本进行操作的方法（如 GetState、PutState、DelState 等），这些方法用户可以直接在链码中进行调用。

```
type ChaincodeStub struct {
    TXID           string
    chaincodeEvent *pb.ChaincodeEvent
    args           [][]byte
    handler        *Handler
    signedProposal *pb.SignedProposal
    proposal       *pb.Proposal

    // Additional fields extracted from the signedProposal
    creator      []byte
    transient    map[string][]byte
    binding      []byte
}
```

stub 会进一步调用本地的 Handler 结构提供的方法，主要过程都是封装为 ChaincodeMessage，发给 peer 节点进行指定的操作。

handler.go

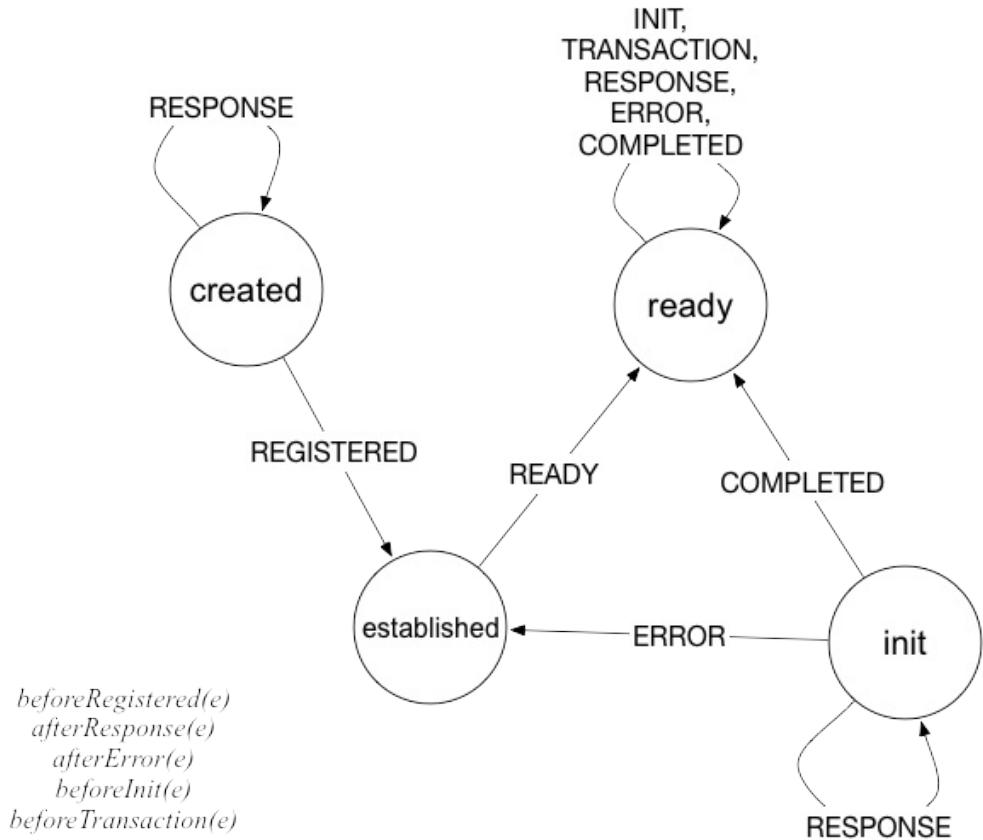
最主要的是实现了 Handler 结构体，通过各种 handleXXX 方法具体实现来自 Chaincode 接口中定义的各种对账本的操作。

```
type Handler struct {
    sync.RWMutex
    //shim to peer grpc serializer. User only in serialSend
    serialLock sync.Mutex
    To          string
    ChatStream PeerChaincodeStream
    FSM         *fsm.FSM
    cc          Chaincode
    // Multiple queries (and one transaction) with different txids can be executing in
    parallel for this chaincode
    // responseChannel is the channel on which responses are communicated by the shim
    to the chaincodeStub.
    responseChannel map[string]chan pb.ChaincodeMessage
    nextState       chan *nextStateInfo
}
```

成员主要包括：

- ChatStream：跟 Peer 进行通信的 grpc 流。
- FSM：最重要的事件处理状态机，根据收到不同事件调用不同方法。
- cc：所面向的链码。
- responseChannel：本地 chan。字典结构，key 是 TxID，value 里面可以放上一些消息，供调用者后面使用。
- nextState：本地 chan，可以存放下一步要进行的操作和数据。

FSM



FSM 的初始化在 `newChaincodeHandler()` 方法中。

下面是 FSM 可能会触发的方法，每个方法首先都会从 `e.Args[0]` 尝试解析 `ChaincodeMessage` 结构，如果失败则退出，成功则继续。

- `beforeRegistered(e)`：收到了注册到 peer 成功的消息，不进行任何操作。
- `afterResponse(e)`：将消息放到 `responseChannel` 中。
- `afterError(e)`：将消息放到 `responseChannel` 中。
- `beforeInit(e)`：收到初始化请求 `INIT` 消息。解析消息后调用 `Handler.handleInit()` 方法进行处理。该方法从消息 `Payload` 中解析出 `ChaincodeInput` 结构，利用这些信息，新建 `stub` 结构，并调用 `stub.init` 方法对 `stub` 进行初始化（配置 `TxID`、`args`、`handler`、`signedProposal`、`creator`、`transient`、`binding` 等成员）。之后，调用 `Handler` 结构成员 `chaincode` 结构的 `Init` 方法（由用户编写）。将收到的结果构造一个 `COMPLETED` `ChaincodeMessage`，放到 `nextState` 里面待发送。
- `beforeTransaction(e)`：收到发起交易的请求 `TRANSACTION` 消息。解析消息后调用 `Handler.handleTransaction()` 方法进行处理。该方法从消息 `Payload` 中解析出 `ChaincodeInput` 结构，利用这些信息，新建 `stub` 结构，并调用 `stub.init` 方法对 `stub` 进行初始化（配置 `TxID`、`args`、`handler`、`signedProposal`、`creator`、`transient`、`binding` 等成员）。之后，调用 `Handler` 结构成员 `chaincode` 结构的 `Invoke` 方法（由用户编

写)。将收到的结果构造一个 COMPLETED ChaincodeMessage, 放到 nextState 里面待发送。

inprocstream.go

interfaces.go

定义接口 Chaincode 和 ChaincodeStubInterface。

```
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 ChaincodeStubInterface) pb.Response
    // Invoke is called for every Invoke transactions. The chaincode may change
    // its state variables
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

用户自己编写的 chaincode 代码需要实现这两个方法，在代码中通过 stub 来跟 ledger 进行交互。

mockstub.go

response.go

testdata

server1.key

server1.pem

ccproviderimpl.go

ccProviderImpl 结构封装了对链码操作的上层方法。

```
type ccProviderImpl struct {
    txsim ledger.TxSimulator
}
```

chaincode_support.go

ChaincodeSupport 结构是 peer 侧对链码支持的主要数据结构。

```
type ChaincodeSupport struct {
    runningChaincodes *runningChaincodes
    peerAddress       string
    ccStartupTimeout time.Duration
    peerNetworkID    string
    peerID            string
    peerTLSCertFile  string
    peerTLSKeyFile   string
    peerTLSSvrHostOrd string
    keepalive         time.Duration
    chaincodeLogLevel string
    shimLogLevel      string
    logFormat          string
    executetimeout    time.Duration
    userRunsCC        bool
    peerTLS           bool
}
```

包括几个主要方法。

- Execute() 方法：在链码侧执行一个交易。
- HandleChaincodeStream() 方法：响应链码容器消息流。
- Launch() 方法：启动一个链码容器，并完成注册。
- Register()方法：创建并初始化 peer 端的 chaincode 处理的 Handler。
- stop()方法：停止一个链码容器。

chaincodeexec.go

支持从 LSCC 中获取 CDS 和 链码数据。通过调用 ExecuteChaincode() 来实现。

- GetCDSFromLSCC()
- GetChaincodeDataFromLSCC()

ExecuteChaincode

- 创建简单的 ChaincodeInvocationSpec，只包含链码名字和Args。
- 核心：调用 Execute()方法（见core/chaincode/exectransaction.go），返回响应 response和事件event。
- 返回response、event。

chaincodetest.yaml

exectransaction.go

提供 `func Execute(ctx context.Context, cccid *ccprovider.CCContext, spec interface{}) (*pb.Response, *pb.ChaincodeEvent, error)` 方法。

主要过程：

- 提起 ChaincodeSupport，Launch()（检查链码容器是否已经提起来了，如果没有则提起 chaincode，等待 register 状态，转换为 ready 状态）。
- 通过 ChaincodeSupport 来执行，Execute()（检查链码容器是否运行，`handler.sendExecuteMessage()`，监听并返回消息 response，类型为 ChaincodeMessage）。
- 检查 `response.Type` 是否为 COMPLETED
- 返回解码后的 `response.Payload`、`response.ChaincodeEvent`。

handler.go

Handler 结构

Handler 结构，负责 Peer 响应 chaincode 容器过来的各种消息。

```
type Handler struct {
    sync.RWMutex
    //peer to shim grpc serializer. User only in serialSend
    serialLock sync.Mutex
    ChatStream ccintf.ChaincodeStream
    FSM        *fsm.FSM
    ChaincodeID *pb.ChaincodeID
    ccInstance *sysccprovider.ChaincodeInstance

    chaincodeSupport *ChaincodeSupport
    registered      bool
    readyNotify      chan bool
    // Map of tx txid to either invoke tx. Each tx will be
    // added prior to execute and remove when done execute
    txCtxs map[string]*transactionContext

    txidMap map[string]bool

    // used to do Send after making sure the state transition is complete
    nextState chan *nextStateInfo

    policyChecker policy.PolicyChecker
}
```

chaincode 容器启动后，会调用到服务端的 Register() 方法，该方法进一步调用到 HandleChaincodeStream()

```
// HandleChaincodeStream Main loop for handling the associated Chaincode stream
func HandleChaincodeStream(chaincodeSupport *ChaincodeSupport, ctxt context.Context, s
tream ccintf.ChaincodeStream) error {
    deadline, ok := ctxt.Deadline()
    chaincodeLogger.Debugf("Current context deadline = %s, ok = %v", deadline, ok)
    handler := newChaincodeSupportHandler(chaincodeSupport, stream)
    return handler.processStream()
}
```

newChaincodeSupportHandler 方法中会初始化 FSM。

之后，调用 handler.processStream() 进入对来自 chaincode 容器消息处理的主循环。

handler.processStream() 主消息循环

Peer 侧维护一个到 cc 的双向流，循环处理消息。主要在 `func (handler *Handler) processStream() error` 方法中（cc 到 peer 注册后会自动调用到该方法）。

主循环过程代码如下：

```

for {
    in = nil
    err = nil
    nsInfo = nil
    if recv {
        recv = false
        go func() {
            var in2 *pb.ChaincodeMessage
            in2, err = handler.ChatStream.Recv()
            msgAvail <- in2
        }()
    }
    select {
    case sendErr := <-errc:
        if sendErr != nil {
            return sendErr
        }
        //send was successful, just continue
        continue
    case in = <-msgAvail:
        // Defer the deregistering of the this handler.
        if err == io.EOF {
            chaincodeLogger.Debugf("Received EOF, ending chaincode support stream,
%s", err)
            return err
        } else if err != nil {
            chaincodeLogger.Errorf("Error handling chaincode support stream: %s",
err)
            return err
        } else if in == nil {
            err = fmt.Errorf("Received nil message, ending chaincode support strea
m")
            chaincodeLogger.Debug("Received nil message, ending chaincode support
stream")
            return err
        }
        chaincodeLogger.Debugf("[%s]Received message %s from shim", shorttxid(in.T
xid), in.Type.String())
        if in.Type.String() == pb.ChaincodeMessage_ERROR.String() {
            chaincodeLogger.Errorf("Got error: %s", string(in.Payload))
        }

        // we can spin off another Recv again
        recv = true
    }
}

```

```

        if in.Type == pb.ChaincodeMessage_KEEPALIVE {
            chaincodeLogger.Debug("Received KEEPALIVE Response")
            // Received a keep alive message, we don't do anything with it for now
            // and it does not touch the state machine
            continue
        }
    case nsInfo = <-handler.nextState:
        in = nsInfo.msg
        if in == nil {
            err = fmt.Errorf("Next state nil message, ending chaincode support str
eam")
            chaincodeLogger.Debug("Next state nil message, ending chaincode support
t stream")
            return err
        }
        chaincodeLogger.Debugf("[%s]Move state message %s", shorttxid(in.Txid), in
.Type.String())
    case <-handler.waitForKeepaliveTimer():
        if handler.chaincodeSupport.keepalive <= 0 {
            chaincodeLogger.Errorf("Invalid select: keepalive not on (keepalive=%d
)", handler.chaincodeSupport.keepalive)
            continue
        }

        //if no error message from serialSend, KEEPALIVE happy, and don't care abo
ut error
        //((maybe it'll work later)
        handler.serialSendAsync(&pb.ChaincodeMessage{Type: pb.ChaincodeMessage_KEE
PALIVE}, nil)
        continue
    }

    err = handler.HandleMessage(in)
    if err != nil {
        chaincodeLogger.Errorf("[%s]Error handling message, ending stream: %s", sh
orttxid(in.Txid), err)
        return fmt.Errorf("Error handling message, ending stream: %s", err)
    }

    if nsInfo != nil && nsInfo.sendToCC {
        chaincodeLogger.Debugf("[%s]sending state message %s", shorttxid(in.Txid),
in.Type.String())
        //ready messages are sent sync
        if nsInfo.sendSync {
            if in.Type.String() != pb.ChaincodeMessage_READY.String() {
                panic(fmt.Sprintf("[%s]Sync send can only be for READY state %s\n",
shorttxid(in.Txid), in.Type.String()))
            }
            if err = handler.serialSend(in); err != nil {
                return fmt.Errorf("[%s]Error sending ready message, ending stream
: %s", shorttxid(in.Txid), err)
            }
        } else {
    }
}

```

```
//if error bail in select  
    handler.serialSendAsync(in, errc)  
}  
}  
}
```

首先是利用 `select` 结构尝试读取各种消息。包括：

- case in = <-msgAvail : 从 CC 侧读取到请求消息；
 - case nsInfo = <-handler.nextState : 读取切换到下个状态的附加消息。
 - case <-handler.waitForKeepaliveTimer() : 定期发出心跳刷新消息。

读取到合法消息后，会分别调用 `handler.HandleMessage(in)` 处理 cc 消息；以及检查状态切换消息（仅允许消息类型为 READY，意味着此时 cc 在正常运行状态），是否要发送给 cc 侧（`sendToCC` 为 True）。

FSM

定义的状态、事件主要在 `func newChaincodeSupportHandler(chaincodeSupport *ChaincodeSupport, peerChatStream ccintf.ChaincodeStream) *Handler` 方法中。

一般对应 GET STATE、GET STATE BY RANGE 等简单事件，调用 handleXXX 方法。

PUT_STATE、DEL_STATE、INVOKE_CHAINCODE 三个事件，则会触发 enterBusyState() 方法。

comm

一些简单的常见函数。

testdata

certs

grpc

impersonation

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。

creds.go

producer.go

server.go

committer

Committer 角色接口，实现上部分地方调用 peer 下面的方法。

txvalidator

负责在 `commit` 阶段对区块和交易进行合法性检查。

validator.go

主要实现 txValidator 结构体。

```
type txValidator struct {
    support Support
    vscc     vsccValidator
}
```

该结构体提供了 `func (v *txValidator) Validate(block *common.Block) error` 方法，对给定区块进行合法性检查。

主要过程如下：

- 创建一个交易过滤器，初始时给区块中所有交易都打上`valid`标签，然后依次检查每笔交易，再修改标签，该过滤器为了方便之后`commit`过程筛选有效交易。
- 对区块中的每笔交易(`env`)都执行 `ValidateTransaction()`，其主要内容是检查交易的组成的完整性，签名的正确性，交易ID的正确性（检查重复交易的前提）以及交易请求 `Proposal`的一致性（比较hash）。
- 检查交易所在通道是否存在。
- 若通道头的类型是背书交易，通过`TxID`检验交易是否已经存在，若否然后执行 `vsccValidateTx()`，最后通过获取交易`instance`来标记下是`invoke`交易还是`upgrade`交易。`vsccValidateTx()`主要内容是获取交易读写集，检查写集的合法性（非SCC的话不能对LSCC和不可调用的SCC写入，SCC的话如果自身不能从外部调用也不可以写入），获取对应的VSCC和`policy`，发起对VSCC的调用来验证。
- 若通道头的类型是配置，执行 `Apply()` 去应用配置。
- 根据标记的`invoke`交易和`upgrade`交易，若有同一`cc`的多个`upgrade`交易，保留最后一个，前面的都标记为非法；若`invoke`交易对应的`cc`（在该区块中）存在`upgrade`交易，则把所有该`cc`的`invoke`交易标记为非法。
- 把交易过滤器添加进`block`的`metadata`中，用于之后的`commit`过程。

committer.go

定义 Committer 接口。Committer 角色必须实现这一接口。

```
type Committer interface {
    // Commit block to the ledger
    Commit(block *common.Block) error

    // Get recent block sequence number
    LedgerHeight() (uint64, error)

    // Gets blocks with sequence numbers provided in the slice
    GetBlocks(blockSeqs []uint64) []*common.Block

    // Closes committing service
    Close()
}
```

其中，最重要的是 `Commit(block *common.Block) error` 方法，负责完成对某个区块的验证的最终确认提交。

committer_impl.go

LedgerCommitter 结构体实现了 Committer 接口。

```
type LedgerCommitter struct {
    ledger    ledger.PeerLedger
    validator txvalidator.Validator
    eventer   ConfigBlockEventer
}
```

其中，最重要的是 `Commit(block *common.Block) error` 方法，负责完成对某个区块的验证的最终确认提交。主要流程如下图所示。

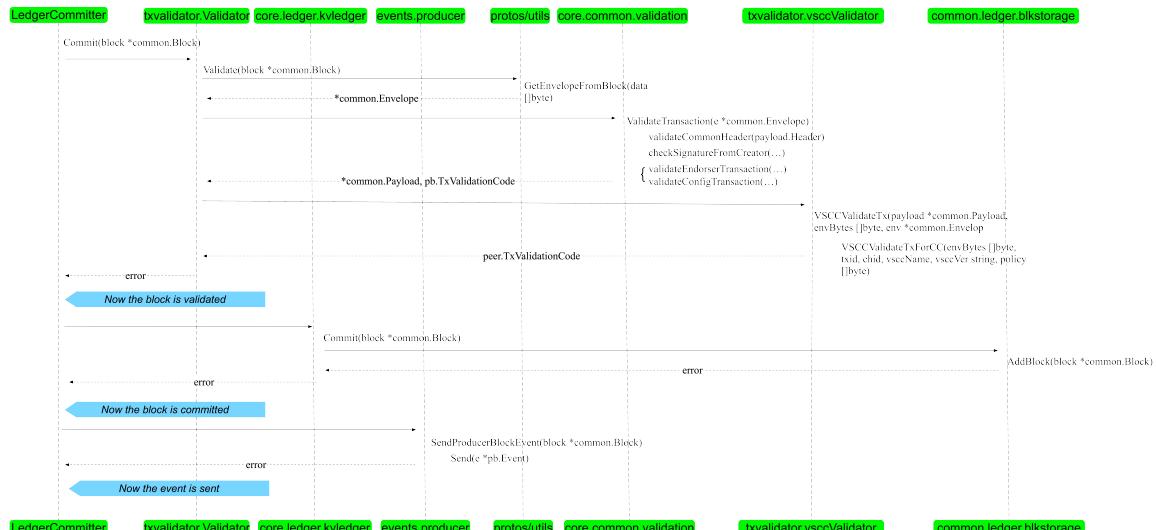


图 1.6.3.3.1 - Commit 流程

common

ccpackage

ccpackage.go

ccprovider

ccinfocache.go

ccprovider.go

cdspackage.go

sigcdspackage.go

sysccprovider

sysccprovider.go

validation

msgvalidation.go

config

config.go

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

msp

sampleconfig

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。

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。

deliverservice

blocksprovider

blocksprovider.go

mocks

blocksprovider.go

orderer.go

client.go

deliveryclient.go

requester.go

endorser

Endorser 角色接口，实现上部分地方调用 peer 下面的方法。

endorser.go

提供 Endorser 结构。

```
// Endorser provides the Endorser service ProcessProposal
type Endorser struct {
    policyChecker policy.PolicyChecker
}
```

最关键的是提供了 ProcessProposal 方法，供 grpc 客户端远程调用，用来接受交易提案，进行背书处理。

ProcessProposal 方法

主要过程如下：

- 调用 ValidateProposalMessage() 方法对签名的提案进行格式检查，主要检查 Channel 头（是否合法头部类型）、签名头（是否包括了 nonce 和 creators 数据），检查签名域（creator 是合法证书，签名是否正确）。
- 如果是系统 CC，检查是否是可以从外部调用的三种之一：csc、lsc 或 qsc。
- 如果 chainID 不为空，获取对应 chain 的账本结构，检查 TxID 在账本上没出现过；对于非系统 CC，检查 ACL（根据 chaincode 指定的 endorsement Policy，签名提案在指定 channel 上有写权限，最终是调用 common/cauthdsl 下面代码，支持指定必须包括某个成员来签名，或者是凑够若干个合法签名）。
- 如果 chainID 不为空，获取交易模拟器和历史查询器（通过 ledger 去 new txsimulator 和 historyqueryexecutor，而且交易模拟器是不包含历史信息的，所以为了查询历史需要拿到一个 historyqueryexecutor），把 historyqueryexecutor 加入到 Context 的 K-V 储存中。
- 如果 chainID 不为空，调用 simulateProposal() 方法获取模拟执行的结果，检查返回的响应 response 的状态，若不小于错误 500 则创建并返回一个失败的 ProposalResponse。
- chainID 不为空下，调用 endorseProposal() 方法对之前得到的模拟执行的结果进行背书，返回 ProposalResponse，检查 simulateProposal 返回的 response 的状态，若不小于错误阈值 400（被背书节点反对），返回 ProposalResponse 及链码错误 chaincodeError（endorseProposal 里有检查链码执行结果的状态，而 simulateProposal 没有检查）。
- 将 response.Payload 赋给 ProposalResponse.Response.Payload（因为 simulateProposal 返回的 response 里面包含链码调用的结果）。
- 返回响应消息 ProposalResponse。

simulateProposal 方法

主要过程如下：

- 获取ChaincodeInvocationSpec，里面包含了链码调用时传入的各种参数。
- 检查ESCC和VSCC（TODO）
- 不是系统链码的话，检查是否实例化，并检验本地文件系统得到的实例化Policy是否跟LSCC得到的实例化Policy匹配。
- 执行Proposal，调用callChaincode()方法，返回response和ccevent。
- 对transactionSimulator执行GetTxSimulationResults()拿到交易读写集simResult。
- 返回链码数据ChaincodeData（LSCC中的）、响应response、交易读写集simResult、链码事件信息ccevent。

endorseProposal 方法

主要过程如下：

- 获取被调用的链码指定的背书链码的名字。
- 通过callChaincode()实现对背书链码的调用，返回响应response（对ESCC的调用同样也会产生simulation results，但ESCC不能背书自己产生的simulation results，需要背书最初被调用的链码产生的simulation results）。
- 检查response.Status，是否大于等于400（错误阈值），若是则把response赋给proposalResponse.Response并返回proposalResponse。
- 将response.Payload解码后（ProposalResponse类型）返回。

callChaincode 方法

主要过程如下：

- 判断交易模拟器，不为空则把它加入到Context的K-V存储中。
- 判断被call的cc是不是系统链码，创建CCContext（包含通道名、链码名、版本号、交易ID、是否SCC、签名Prop、Prop）
- 调用core/chaincode/chaincodeexec.go下的ExecuteChaincode()，返回响应response和事件ccevent。
- 返回response和ccevent。

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

validateTx(txRWSet, updates)

参数注释：txRWSet交易模拟结果/读写集，updates账本更新（顺序校验一个区块中的交易，每成功验证一个交易，就把交易的写集放进去）

- 该校验只校验读写集中的读集以及范围查询，不做写集的校验（不需要）。
- 针对读集，就是对于每一个Key检查是否在updates中存在，如果有，验证就不成功，另外还要检查当前账本（截至上一个区块）中它的version是否跟读集（背书节点模拟时）中的version一致，因为交易受orderer排序影响以及网络延迟影响，检查version是保证该交易之前没有交易变更过要读的Key，保证信息读取的准确性。
- 针对范围查询，就是在现有账本状态（截至上一个区块）+update中重新执行范围查询，比对结果是否还是一样。

kv_ledger.go

实现一个支持键值对的 peer 账本结构体。

```
type kvLedger struct {
    ledgerID    string
    blockStore  blkstorage.BlockStore
    txtmgmt    txmgr.TxMgr
    historyDB  historydb.HistoryDB
}
```

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

txvalidationflags.go

util.go

辅助函数。

ledger_interface.go

定义账本结构相关的接口。

- HistoryQueryExecutor
- PeerLedger
- PeerLedgerProvider
- QueryExecutor
- TxSimulator
- ValidatedLedger

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-server1-cert.pem

Org1-server1-key.pem

Org2-cert.pem

Org2-child1-cert.pem

Org2-child1-key.pem

Org2-child1-server1-cert.pem

Org2-child1-server1-key.pem

Org2-server1-cert.pem

Org2-server1-key.pem

Org3-cert.pem

Org3-server1-cert.pem

Org3-server1-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
```

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          // 标记是否是验证者
    reconnectOnce sync.Once     // 重新连接的定时器
    discHelper    discovery.Discovery // 探测任务句柄
    discPersist   bool          // 是否持久化探测
}
```

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

mocks.go

policy.go

policyprovider

provider.go

SCC

System Chaincode

已更新escc、lscc和vscc。

CSCC

configure.go

escc

Endorsement System Chaincode

负责对模拟执行结果进行签名。

目前只有签名功能，以后会扩展。

进一步的分析请查阅 `endorser_onevalidsignature_go.md`。

endorser_onelvalidsignature.go

`Invoke()`用来背书特定的Proposal。

目前，只能对输入进行签名并返回背书的结果。以后会对chaincode进行扩展，来提供更复杂的背书策略过程。例如把策略详细信息编码成链码的调用交易，以及允许客户通过参数去选择使用哪一个背书策略。

调用ESCC时有4个必须的参数，还有两个可选。

序号	内容	备注
0	函数名	目前未使用
1	序列化的Header	必需
2	序列化的ChaincodeProposalPayload	必需
3	被调用的链码的ChaincodeID	必需
4	链码调用的结果 (Response)	必需
5	模拟结果 (读写集)	必需
6	序列化的事件	非必需
7	可见度	非必需，目前是完全可见

主要过程如下：

- 检查各个参数。
- 获取该节点的签名身份。
- 根据传进来的参数创建ProposalResponse。
- 将ProposalResponse编码为prBytes，返回shim.Success(prBytes)。

lccc

Lifecycle System Chaincode。

负责Chaincode的安装、部署、更新、查询等操作。

进一步的分析请查阅 `lscc.go.md`。

Iccc.go

LSCC---Lifecycle System Chaincode, 负责chaincode的全生命周期管理。

Invoke操作有INSTALL、DEPLOY(语义上的实例化)、UPGRADE、GETCCINFO、GETDEPSPEC、GETCCDATA、GETCHAINCODES、GETINSTALLEDCHAINCODES。

操作	参数
INSTALL	1.ChaincodeDeploymentSpec
DEPLOY UPGRADE	1.chainName 2.ChaincodeDeploymentSpec 3.SignaturePolicyEnvelope(endorsement policy) 4.escc 5.vscc
GETCCINFO GETDEPSPEC GETCCDATA	1.chainName 2.ccName
GETCHAINCODES GETINSTALLEDCHAINCODES	None

INSTALL

- 验证SignedProposal是不是admin节点签署的（check时传入的ADMIN）。
- executeInstall()
 - 检查cc名字和版本是否合法，不包含非法字符。
 - 把ccPackage写入文件系统。

DEPLOY

- 检查参数，如果空则赋值默认值，policy默认是被任意一个成员签署，escc和vscc默认就是系统的escc和vscc，暂不支持自己指定，因为后面会检测指定的cc是不是scc，而scc目前就是固定的这几个不能动态增加。
- executeDeploy()
 - 检查cc名字和版本是否合法，不包含非法字符（以及ACL，access control，TODO）。
 - 检查链码是否存在，就是已经被实例化。
 - 获取ccPackage并转换成ChaincodeData。
 - 验证实例化策略，ccPackage有两种，源码打包生成的，还有install时直接传入的，直接传入的包有可能是被签名的，被签名的ccPackage是被指定了实例化策略的，只有指定的身份才可以实例化这个cc，其他所有的没被签名的ccPackage默认任意一个ADMIN节点都可以实例化。平常用的install命令，本地传入地址和参数打包生成的

ccPackage都是未被签名的。

- createChaincode()，检查指定的esc和vsc是不是scc（所以暂不支持自己的背书和验证链码），最后putState()，其中key是cc名字，value是序列化后的ChaincodeData。

UPGRADE

- 跟DEPLOY类似，参考DEPLOY，多了一步检查版本号是不是跟旧的版本号不同。

GETCCINFO GETDEPSPEC GETCCDATA

- 检查通道Readers策略，是否有可读权限。
- 检查cc是否被实例化，成功则按照函数名返回相应信息。
- GETCCINFO返回cc名（成功获取ChaincodeData）。
- GETDEPSPEC返回序列化后的ChaincodeDeploymentSpec。
- GETCCDATA返回序列化后的ChaincodeData。

GETCHAINCODES GETINSTALLEDCHAINCODES

- 检查是不是ADMIN节点。
- GETCHAINCODES返回所有已经实例化的cc（对LSCC的state进行范围查询）。
- GETINSTALLEDCHAINCODES返回节点上所有安装的cc（不一定实例化）。

qscC

query.go

samplesyscc

samplesyscc.go

VSCC

Validate System Chaincode

负责背书的校验。

相对于目前escc只做签名，vscC的工作也就是验证（每个）签名的有效性，以及是否符合背书策略（有效签名个数是否满足）。

特殊地，vscC中有一部分专门针对lSCC的校验。

进一步的分析请查阅 `validator_onevalidsignature_go.md`。

validator_onevalidsignature.go

参数

序号	内容	备注
1	函数名	未使用
2	序列化的Env	无
3	序列化的policy	无

Invoke()

- 检查参数。
- 获取policy，`NewPolicy()`会把传入的签名策略编译成验证函数。签名策略有两种类型，一种是NOOutOf（就是我们看到的AND和OR组合的背书策略，本质上是多个NOOutOf的组合，表达一定个数中至少要有几个的意思），一种是SignedBy（由特定的身份签名），对于编译出来的验证函数，SignedBy就是校验给定的SignedData数组中有没有该特定身份签名的，有就返回TRUE，而NOOutOf相当于SignedBy的数组，根据里面所有SignedBy返回的TRUE的个数是否满足设定的数值来返回TRUE或FAUSE。
- 从交易中抽取出所有的背书endorsements，去掉重复的身份identity（背书节点），构建一个签名集signatureSet，实质上是一个SignedData的数组，SignedData由原始数据、身份和签名组成。
- 执行`policy.Evaluate(signatureSet)`，就是去执行编译出来的验证函数，其验证思路参考上面获取policy步骤。
- 另外如果被调用的cc是LSCC的话，则执行特殊的validate过程，`ValidateLSCCInvocation()`，主要检查调用LSCC的参数，如果是实例化或更新操作则另外检查cc是否install，写集是否有且只有两个（一个LSCC，一个cc），且符合LSCC写入的规范（Key必须是要实例化或更新的cc的名字，Value必须是ChaincodeData且名字版本都匹配，对LSCC只能putstate()一次），最后检查instantiatePolicy。
- 返回`shim.Success(nil)`。

importsccs.go

sccproviderimpl.go

sysccapi.go

testutil

config.go

admin.go

对 peer 服务的管理：启动和停止、查看等。

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

footer.html

layout.html

dev-setup

build.rst

devenv.rst

headers.txt

Gerrit

best-practices.rst

changes.rst

gerrit.rst

If-account.rst

reviewing.rst

images

Setup

TLSSetup.rst

Style-guides

go-style.rst

arch-deep-dive.rst

architecture.rst

blockchain.rst

build_network.rst

capabilities.rst

chaincode.rst

chaincode4ade.rst

chaincode4noah.rst

channels.rst

conf.py

configtx.rst

configtxgen.rst

configtxlator.rst

CONTRIBUTING.rst

DCO1.1.txt

endorsement-policies.rst

error-handling.rst

Fabric-FAQ.rst

fabric-sdks.rst

fabric_model.rst

这个章节描述了*Hyperledger Fabric*的关键设计功能，实现了其全面但可定制的企业级区块链解决方案的承诺：

- **资产**---资产定义是能通过网络交换获得的任何具有货币价值的东西，从食品到古董车到货币期货。
- **链码**---链码的执行是从交易排序中分离的，限制跨节点类型所需的信任和验证级别，并优化网络扩展性和性能。
- **账本**---不变的共享账本对每个通道的整个交易历史记录进行编码，并且包括类似SQL的查询功能，用于高效的审计和争议解决。
- **专用通道**---通道可以提供竞争企业和在普通网络上交换资产的受监管行业所需的高度隐私和保密性进行多边交易。
- **共识**---*Fabric*独特的共识方法可以提供实现企业所需的灵活性和可扩展性。

资产

资产范围从有形（房地产和硬件）到无形（合同和知识产权）。您可以轻松地在客户端 *JavaScript* 中定义资产，并使用包含的 *Fabric Composer* 工具在 *Fabric* 应用程序中使用它们。

Fabric 支持使用未使用的交易输出作为后续交易的输入来交换资产的能力。资产（和资产注册表）存放在 *Fabric* 中，作为键值对的集合，状态更改记录为通道账本上的交易。*Fabric* 允许以二进制或 JSON 格式表示任何资产。

链码

链码是定义一个资产或资产们的软件，以及用于修改资产的交易指令。换句话说，它是业务逻辑。链码是强制执行的读取或更改键值对或者其他状态数据库信息的一套规则。链码功能是对总账的当前状态数据库执行，并通过交易提案被启动。链码执行结果是一组可以提交到网络并且被所有对等节点写入账本的键值写入（写入集）。

账本功能

账本是 *Fabric* 中所有状态转换的顺序的、防篡改的记录。状态转换是参与方提交的链码调用（“交易”）的结果。每个交易都会产生一组资产键值对，这些对象将作为创建、更新或删除操作提交给账本。

账本是由一条区块链组成，就像状态数据库一样维护这当前*Fabric*的状态。区块链是由存储着不可更改的、序列化的记录的区块组成。每个通道都有一个账本。每个对等节点都维护着它们所属的每个通道的账本拷贝。

- 查询和更新账本可以使用基于键的查询、范围查询和符合关键字查询。
- 只读查询可以使用富查询语言（如果使用CouchDB作为状态数据库）。
- 只读历史查询---通过关键字查询账本历史记录，获取数据来源的情况。
- 交易包括从链码中读取的键值(读操作)和写进链码的键值(写操作)。
- 交易包含每个背书节点的签名并提交给排序服务。
- 交易在区块中被排序，并且被排序服务“传送”到通道上的对等节点。
- 对等节点根据背书策略验证交易并且执行策略。
- 在加入区块之前，将先执行版本检查，以确保从链码执行时间到现在读取的资产状态没有改变。
- 一旦交易被验证并提交，就具有不变性。
- 通道的账本包含一个配置区块定义策略、访问控制列表和其他相关信息。
- 通道包含会员服务提供者实例，允许从不同的证书颁发机构派生加密资料。

请参阅账本章节，深入了解数据库，存储结构和“查询能力”。

私有通道

*Fabric*采用基于每个通道的不可变的账本，以及可以操作和修改当前资产状态（即更新键值对）的链码。账本存在于通道的范围内 - 可以在整个网络中共享（假设每个参与者都在一个公共通道上运行） - 或者可以将其私有化，只包括一组特定的参与者。

在后一种情况下，这些参与者将创建一个单独的通道，从而隔离/分离其交易和账本。*Fabric*甚至解决了希望弥合透明度和隐私之间差距的场景。链码仅在需要访问资产状态以及执行读和写操作的对等体上安装（换句话说，如果链码未安装在对等体上，则无法与账本正确连接）。为了进一步模糊数据，链码中的值可以使用常用的加密算法（如SHA-256）进行加密（部分或全部），然后记录到账本中。

安全性和成员服务

*Hyperledger Fabric*支撑起一个所有参与者都具有已知身份的交易网络。公共密钥用于生成与组织，网络组件以及最终用户或客户端应用程序相关联的加密证书。因此，可以在更广泛的网络和通道级别上操作和管理对数据的访问控制。*Fabric*的“许可”概念，加上通道的存在和功能，有助于解决隐私和机密性至关重要的情况。

请参阅*Fabric CA*部分，以更好地了解加密实现，以及*Fabric*中使用的签名，验证，验证方法。

共识

在分布式分类帐技术中，共识最近已成为单一功能中特定算法的代名词。然而，共识不仅仅是简单地同意交易顺序，而是通过其在整个交易流程中的基本角色，从提案和认可到排序，验证和提交，在`Hyperledger Fabric`中强调了这种差异化。简而言之，共识被定义为对包含块的一组交易的正确性的全方位验证。

当区块的交易排序和结果符合明确的政策标准检查时，最终达成共识。这些检查和平衡是在交易的生命周期内进行的，并且包括命令某些交易类型必须得到特定成员认可的认可策略的用法，以及确保这些策略得到执行和维护的系统链码。在交易提交之前，对等节点将部署这些系统链码来确保它有足够的认可，并且它们是从适当的实体派生出来的。此外，在包含交易的任何区块写入到账本之前，在账本的当前状态同意或者同意之前要进行版本检查。这个最终检查提供了针对双重支付操作和可能危及数据完整性的其他威胁的保护，并允许对非静态变量执行函数。

除了发生的众多认可，有效性和版本检查之外，还有在交易流程的所有方向上发生的持续身份验证。访问控制列表在网络的层次层次上实现（从订单服务到通道），并且随着交易提议通过不同架构组件，有效负载被重复签名，确认和验证。总而言之，共识不仅仅是对一批交易进行排序，而是作为在交易从提议到确认提交的过程中进行的正在进行的验证的副产品而实现的总体表征。

查阅交易流程图，可以直观的感受共识

getting_started.rst

glossary.rst

gossip.rst

index.rst

install_instantiate.rst

jira_navigation.rst

kafka.rst

ledger.rst

logging-control.rst

MAINTAINERS.rst

mdtorst.sh

msp-identity-validity-rules.rst

msp.rst

peer-chaincode-devmode.rst

policies.rst

prereqs.rst

questions.rst

readwrite.rst

releases.rst

requirements.txt

samples.rst

security_model.rst

smartcontract.rst

status.rst

testing.rst

txflow.rst

usecases.rst

videos.rst

whyfabric.rst

write_first_app.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

事件消费者，负责从系统中获取事件。

客户端如果想监听系统中事件，可以通过使用消费者模块提供的方法进行。

主要代码在 [consumer.go](#)。

adapter.go

定义一个事件的 Adapter 接口，获取感兴趣的事件类型，接收事件。

```
//EventAdapter is the interface by which a fabric event client registers interested events and
//receives messages from the fabric event Server
type EventAdapter interface {
    GetInterestedEvents() ([]*peer.Interest, error)
    Recv(msg *peer.Event) (bool, error)
    Disconnected(err error)
}
```

consumer.go

消费者模块的主要结构和实现。

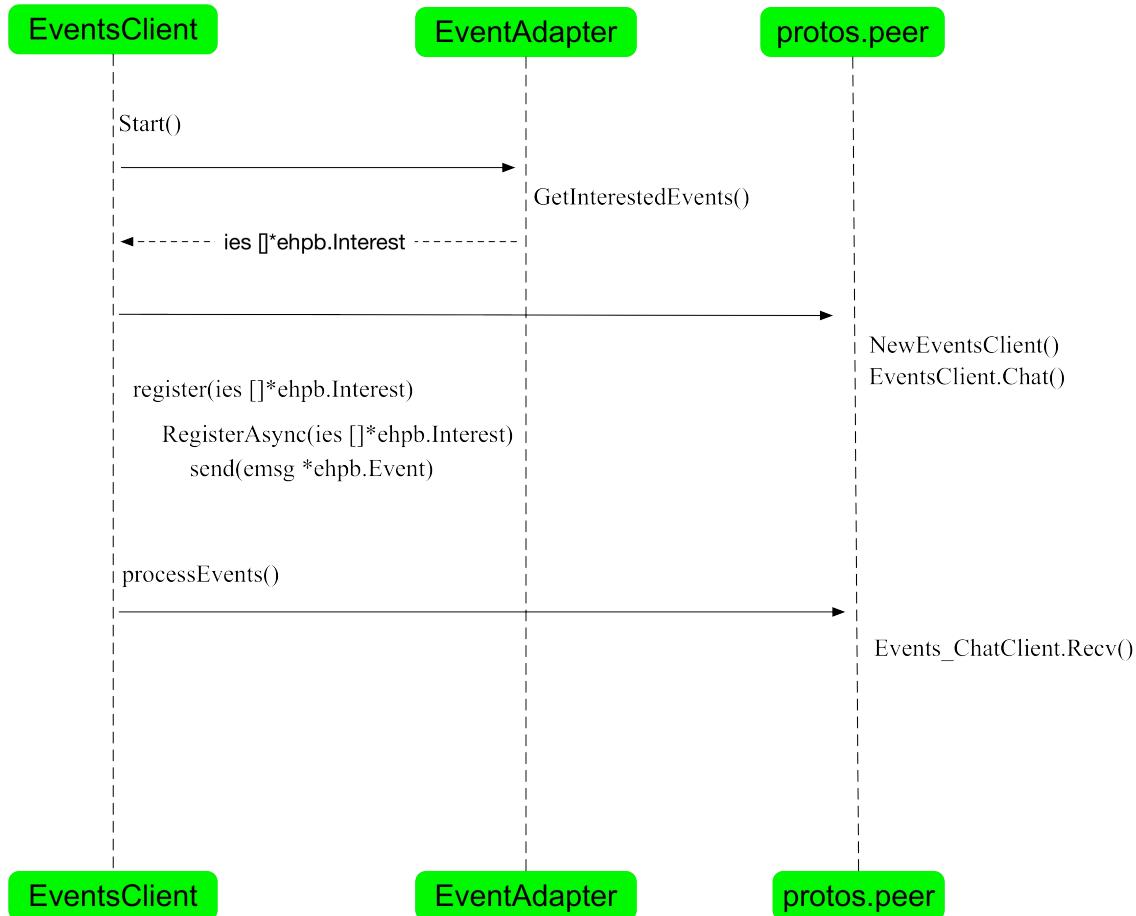
主要实现了 EventsClient 结构体，供用户使用获取系统中事件。

```
type EventsClient struct {
    sync.RWMutex
    peerAddress string
    regTimeout  time.Duration
    stream       ehpb.Events_ChatClient
    adapter      EventAdapter
}
```

提供几个主要方法：

- `func (ec *EventsClient) Start() error` : 入口方法，该方法会自动创建到 peer 事件流的连接，并开始处理收到的消息。
- `func (ec *EventsClient) Recv() (*ehpb.Event, error)` : 从流连接中接收一个事件。
- `func (ec *EventsClient) Stop() error` : 终止事件流连接。

启动后主要调用过程如下图所示。

图 1.9.1.2.1 - *consumer* 客户端启动后流程

producer

事件产生者，负责提供一个事件服务器。

任何希望生成事件的模块，可以使用产生模块提供的方法进行。

主要代码在 [producer.go](#)。

eventhelper.go

events.go

事件产生的入口。

主要实现了 `eventProcessor` 结构体，负责实现对事件的处理。

```
type eventProcessor struct {
    sync.RWMutex
    eventConsumers map[pb.EventType]handlerList

    //we could generalize this with mutiple channels each with its own size
    eventChannel chan *pb.Event

    //timeout duration for producer to send an event.
    //if < 0, if buffer full, unblocks immediately and not send
    //if 0, if buffer full, will block and guarantee the event will be sent out
    //if > 0, if buffer full, blocks till timeout
    timeout time.Duration
}
```

`start()` 方法启动后会进入主循环，不断检测 `eventChannel` 中是否有事件（由组件产生），如果有，则调用对应的 `handler` 处理。

handler.go

handler 是针对某种消息的处理句柄。

```
type handler struct {
    ChatStream      pb.Events_ChatServer
    interestedEvents map[string]*pb.Interest
}
```

producer.go

NewEventsServer() 方法

该方法会创建一个全局的事件服务器（EventsServer），并完成初始化工作。

```
// NewEventsServer returns a EventsServer
func NewEventsServer(bufferSize uint, timeout time.Duration) *EventsServer {
    if globalEventsServer != nil {
        panic("Cannot create multiple event hub servers")
    }
    globalEventsServer = new(EventsServer)
    initializeEvents(bufferSize, timeout)
    //initializeCCEventProcessor(bufferSize, timeout)
    return globalEventsServer
}
```

代码中会初始化两个全局变量：

- globalEventsServer：全局唯一的事件服务器。
- gEventProcessor：全局唯一的 事件处理器。

initializeEvents(bufferSize, timeout)方法会注册内部消息类型（包括 EventType_BLOCK、EventType_CHAINCODE、EventType_REJECTION 和 EventType_REGISTER），并启动 gEventProcessor。

```
func initializeEvents(bufferSize uint, tout time.Duration) {
    if gEventProcessor != nil {
        panic("should not be called twice")
    }

    gEventProcessor = &eventProcessor{eventConsumers: make(map[pb.EventType]handlerList),
        eventChannel: make(chan *pb.Event, bufferSize), timeout: tout}

    addInternalEventTypes()

    //start the event processor
    go gEventProcessor.start()
}
```

gEventProcessor 启动后会不断从 eventChannel 通道中读取消息，并调用绑定的 handler 的 SendMessage() 方法发送出去。

EventsServer 结构体

主要实现了事件服务器结构体。

```
// EventsServer implementation of the Peer service
type EventsServer struct {
}
```

结构体实现了如下方法：

- `func (p *EventsServer) Chat(stream pb.Events_ChatServer) error`：调用该方法会发送消息到该事件服务器。事件服务器会生成一个新的事件处理句柄，并调用句柄的 `HandleMessage()` 方法对消息进行处理。

register_internal_events.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

go

chaincode_example01

chaincode_example02

chaincode_example03

chaincode_example04

chaincode_example05

go

eventsender

invokereturnsvalue

go

map

go

marbles02

go

passthru

go

sleeper

go

utxo

java

chaincode_example02

chaincode_example04

chaincode_example05

chaincode_example06

eventsender

Example

LinkExample

MapExample

RangeExample

SimpleSample

cluster

compose

compose

peer-base

docker-compose.yaml

report-env.sh

config

configtx.yaml

core.yaml

cryptogen.yaml

fabric-ca-server-config.yaml

fabric-tlsca-server-config.yaml

orderer.yaml

configure.sh

Makefile

usage.txt

configtxupdate

bootstrap_batchsize

script.sh

common_scripts

common.sh

reconfig_batchsize

script.sh

reconfig_membership

script.sh

e2e_cli

base

docker-compose-base.yaml

peer-base.yaml

channel-artifacts

crypto-config

ordererOrganizations

peerOrganizations

examples

chaincode

scripts

script.sh

configtx.yaml

crypto-config.yaml

docker-compose-cli.yaml

docker-compose-couch.yaml

docker-compose-e2e-template.yaml

docker-compose-e2e.yaml

download-dockerimages.sh

end-to-end.rst

generateArtifacts.sh

network_setup.sh

events

block-listener

block-listener.go

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

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

pull

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

mocks

gossip.go

metastate.go

payloads_buffer.go

state.go

util

logging.go

misc.go

msgs.go

pubsub.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

tools

Dockerfile.in

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

testdata

badadmin

admincerts

cacerts

keystore

signcerts

config.yaml

badconfigou

admincerts

cacerts

keystore

signcerts

config.yaml

badconfigoucert

admincerts

cacerts

keystore

signcerts

config.yaml

external

admincerts

cacerts

intermediatecerts

keystore

signcerts

config.yaml

intermediate

admincerts

cacerts

intermediatecerts

keystore

signcerts

intermediate2

admincerts

cacerts

intermediatecerts

keystore

signcerts

users

mspid 包

admincerts 包

cacerts 包

keystore 包

signcerts 包

tlscacerts 包

revocation

admincerts

cacerts

crls

keystore

signcerts

revocation2

admincerts

cacerts

crls

keystore

signcerts

revokedica

admincerts

cacerts

crls

intermediatecerts

keystore

signcerts

tls

admincerts

cacerts

intermediatecerts

keystore

signcerts

tlscacerts

tlsintermediatecerts

config.yaml

cert.go

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
}
```

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。
- `DeserializelIdentity()`：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

orderer

在 fabric 1.0 架构中，共识功能被抽取出来，作为单独的 fabric-orderer 模块来实现，完成核心的排序功能。最核心的功能是实现从客户端过来的 broadcast 请求，和从 orderer 发送到 peer 节点的 deliver 接口。同时，orderer 需要支持多 channel 的维护。

目前，orderer 模块支持三种排序类型：

- Solo：单节点的排序功能，试验性质，不具备可扩展性和容错；
- Kafka：基于 Kafka 集群的排序实现，支持可持久化和可扩展性；
- BFT：支持 BFT 容错的排序实现，尚未完成。

账本记录上支持两种类型：

- Ram：存放近期若干（默认为 1000 个）区块到内存中。
- File：存放区块记录到文件系统，默认是临时目录下的 `hyperledger-fabric-ordererledger` 文件。

[sampleconfig](#) 目录下有示例的配置文件 `orderer.yaml`。

common

通用函数。

blockcutter

blockcutter.go

bootstrap

file

bootstrap.go

broadcast

broadcast

broadcast.go

deliver

deliver.go

ledger 包

各种类型的账本结构，包括ram、json 和文件等。

file 包

json 包

ram 包

ledger.go

util.go

localconfig 包

config.go

metadata 包

metadata.go

msgprocessor 包

filter.go

msgprocessor.go

sigfilter.go

sizefilter.go

standardchannel.go

systemchannel.go

systemchannelfilter.go

multichannel 包

chainsupport.go

registrar.go

performance 包

server.go

utils.go

server 包

main.go

orderer 启动后的 main 方法会调用到这里的 Main() 方法。

核心代码非常简单。

```
// Main is the entry point of orderer process
func Main() {
    fullCmd := kingpin.MustParse(app.Parse(os.Args[1:]))

    // "version" command
    if fullCmd == version.FullCommand() {
        fmt.Println(metadata.GetVersionInfo())
        return
    }

    conf := config.Load()
    initializeLoggingLevel(conf)
    initializeLocalMsp(conf)

    Start(fullCmd, conf)
}
```

整体的调用流程如下图所示。

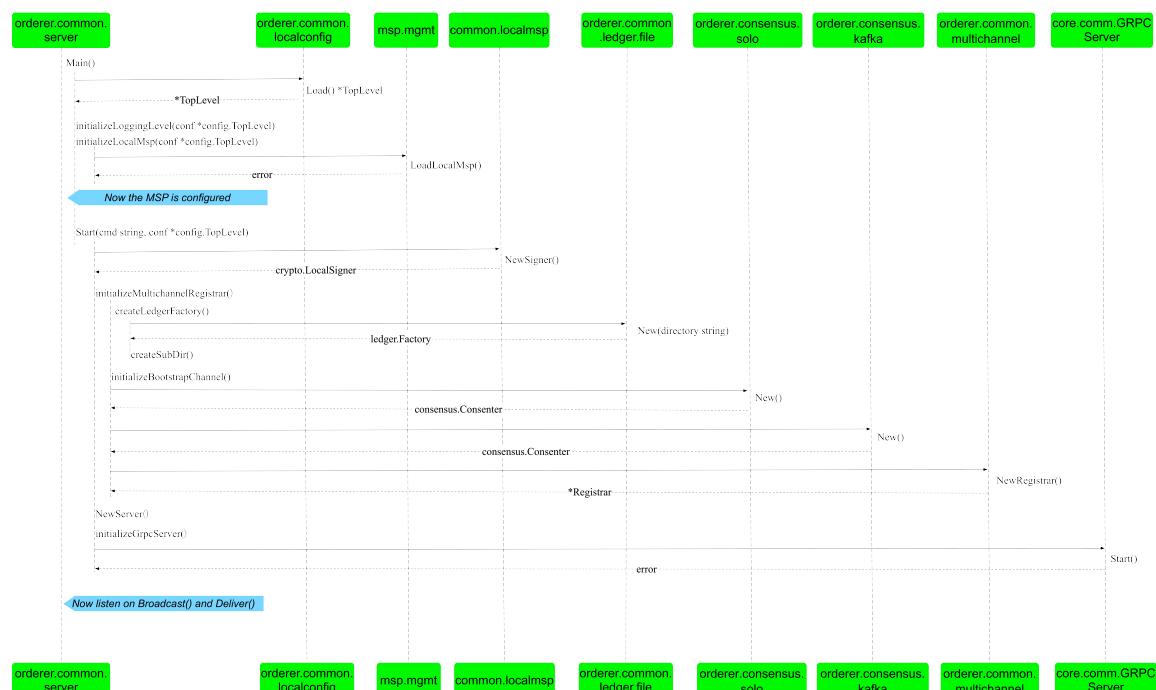


图 1.15.1.11.1.1 - orderer.common.server 包中的 Main() 方法

其中，`initializeLocalMsp()` 和 `Start()`做了大部分的工作。

initializeLocalMsp

根据 `orderer.yaml` 配置中路径，读取本地的 `msp` 数据。

```
func initializeLocalMsp(conf *config.TopLevel) {
    // Load local MSP
    err := mspmgmt.LoadLocalMsp(conf.General.LocalMSPDir, conf.General.BCCSP, conf.General.LocalMSPID)
    if err != nil { // Handle errors reading the config file
        logger.Fatal("Failed to initialize local MSP:", err)
    }
}
```

Start()

`Start()` 完成了主要的操作。

首先，利用 `localmsp`，创建签名者结构。

接下来是初始化各个账本结构。如果本地不存在旧的账本文件时，需要初始化系统通道，判断是否需要从外部 `genesis` 区块文件读入数据，还是根据给定配置初始化 `genesis` 区块结构。

接下来，新建 `grpc server`，其中包括 `Broadcast()` 和 `Deliver()` 两个服务接口。

最后，启动 `grpc` 服务。

```
func Start(cmd string, conf *config.TopLevel) {
    signer := localmsp.NewSigner()
    manager := initializeMultichannelRegistrar(conf, signer)
    server := NewServer(manager, signer, &conf.Debug)

    switch cmd {
    case start.FullCommand(): // "start" command
        logger.Infof("Starting %s", metadata.GetVersionInfo())
        initializeProfilingService(conf)
        grpcServer := initializeGrpcServer(conf)
        ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)
        logger.Info("Beginning to serve requests")
        grpcServer.Start()
    case benchmark.FullCommand(): // "benchmark" command
        logger.Info("Starting orderer in benchmark mode")
        benchmarkServer := performance.GetBenchmarkServer()
        benchmarkServer.RegisterService(server)
        benchmarkServer.Start()
    }
}
```

initializeGrpcServer

创建 grpc 的服务器。

```
grpcServer, err := comm.NewGRPCServerFromListener(lis, secureConfig)
if err != nil {
    logger.Fatal("Failed to return new GRPC server:", err)
}
```

initializeMultiChainManager

初始化一个 multichain.Manager 结构。十分核心的代码。

通过 `createLedgerFactory()` 初始化账本结构，包括 `file`、`json`、`ram` 等类型。`file` 的话会在本地指定目录 (`/var/production/chains`) 下创建账本结构。账本所关联的链名称会自动命名为 `chain_FILENAME`。之后通过指定初始区块，或自动生成来初始化区块链结构。至此，账本结构初始化完成。

接下来，初始化 `consenter` 部分，初始化 `solo`、`kafka` 两种类型。

账本、`consenter`，再加上传入的签名者结构，通过 `NewManagerImpl()` 方法构造一个 `multichain.Manager` 结构，负责处理消息。并且会挨个检查区块链结构，调用 `start` 方法启动。

NewServer

新建一个 `Server` 结构，包括一个 `broadcast` 的处理句柄，以及一个 `deliver` 的处理句柄。

```
type server struct {
    bh broadcast.Handler
    dh deliver.Handler
}
```

`NewServer` 分别初始化这两个句柄，挂载上前面初始化的 `multichain.Manager` 结构。

`broadcast` 句柄还需要初始化一个配置更新的处理器，负责处理 `CONFIG_UPDATE` 交易。

```
func NewServer(ml multichain.Manager, signer crypto.LocalSigner) ab.AtomicBroadcastServer {
    s := &server{
        dh: deliver.NewHandlerImpl(deliverSupport{Manager: ml}),
        bh: broadcast.NewHandlerImpl(broadcastSupport{
            Manager:           ml,
            ConfigUpdateProcessor: configupdate.New(ml.SystemChannelID(), configUpdateSupport{Manager: ml}, signer),
        }),
    }
    return s
}
```

server.go

util.go

consensus 包

kafka 包

跟 Kafka 集群打交道的接口。

chain.go

channel.go

config.go

consenter.go

partitioner.go

retry.go

solo 包

consensus.go

consensus.go

mocks

common 包

blockcutter 包

multichannel 包

util

util.go

sample_clients

broadcast_config

client.go

newchain.go

broadcast_timestamp

broadcast_timestamp

client.go

deliver_stdout

client.go

single_tx_client

single_tx_client.go

main.go

主入口文件。

调用 `orderer.common.server` 包中的 `Main()` 方法。

```
func main() {
    server.Main()
}
```

peer

主命令模块。

作为服务端时候，支持 `node` 子命令；作为命令行时候，支持 `chaincode`、`channel` 等子命令。

作为命令行时候，会维持一个 `ChaincodeCmdFactory` 结构。

```
type ChaincodeCmdFactory struct {
    EndorserClient  pb.EndorserClient
    Signer          msp.SigningIdentity
    BroadcastClient common.BroadcastClient
}
```

其中：

- `EndorserClient` 是跟 `peer.address` 指定地址通信的 `grpc` 通道；
- `Signer` 为 `LocalMSP` 中的默认签名实体；
- `BroadcastClient` 是连接到通过 `-o` 指定的 `orderer` 服务的 `grpc` 通道。

chaincode

包括 `install`、`instantiate`、`invoke`、`query` 等命令，大家的过程都是类似的，创建 `Proposal`，发给 `peer`，获取 `Response`。

`instantiate`、`upgrade`、`invoke` 等子命令还需要根据 `Response` 创建 `SignedTX`，发送给 `Orderer`。

`package`、`signpackage` 子命令则是本地操作。

各子命令的全局参数支持情况如下。

命令	<code>-C</code> 通道	<code>-c cc</code> 参数	<code>-E</code> <code>escc</code>	<code>-n</code> 名称	<code>-o</code> <code>Orderer</code>	<code>-p</code> 路径	<code>-P</code> <code>policy</code>	<code>-v</code> 版本	<code>-V</code> <code>vscc</code>
install	不支持	支持	不支持	必需	不支持	必需	不支持	必需	不支持
instantiate	必需	必需	支持	必需	支持	不支持	支持	必需	支持
upgrade	必需	必需	支持	必需	支持	不支持	不支持	必需	支持
package	不支持	支持	不支持	必需	不支持	必需	不支持	必需	不支持
invoke	支持	必需	不支持	必需	支持	不支持	不支持	不支持	不支持
query	支持	必需	不支持	必需	不支持	不支持	不支持	不支持	不支持

- 不支持：代表该参数即便被指定，也不会被使用。
- 支持：代表该参数可以被使用，如果不指定，则可能采取默认值或自动获取。
- 必需：该参数必需被指定。

chaincode.go

common.go

提供一些通用方法。

chaincodeInvokeOrQuery

- 获取链码详细信息（含有发出的指令里的信息，比如-C 通道、-n 名字、-c 参数）。
- ChaincodeInvokeOrQuery()（创建proposal，对proposal进行签名，给endorser去执行 proposal，拿到proposalResponse后组合成完整的签名交易，类型是一个Envelope，将 envelope发给orderer去排序，返回proposalResponse）。
- 检查proposalResponse。
- 返回nil。

install.go

响应 `peer chaincode install` 命令，将智能合约源码和环境传输到指定的 `peer` 节点上，并生成智能合约的部署打包文件（`name.version`）到默认的 `/var/hyperledger/production/chaincodes/` 目录下。

例如

```
peer chaincode install -n test_cc -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v 1.0
```

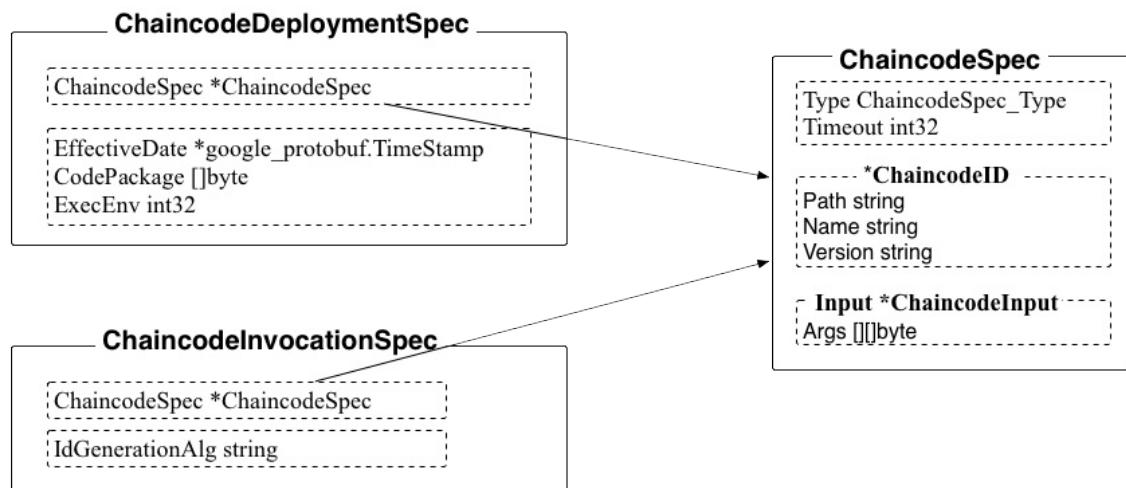
命令调用 `chaincodeInstall` 方法。

一种是通用的方式，通过传入的参数进行打包；一种是直接读入传入的打包文件 `ccpackfile` 进行处理。

整体流程如下：

- 首先会调用 `InitCmdFactory`，初始化 `Endosermmentclient`、`Signer` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。
- 调用 `chaincodeInstall` 方法，解析命令行参数，生成 `ChaincodeSpec`；
- 根据 `CS`，结合 `chaincode` 相关数据生成一个 `ChaincodeDeploymentSpec`（`CDS`）结构（`chainID` 为空），并传入 `install` 方法；
- `install` 方法基于传入的 `CDS`，生成一个 `install` 类型的 `Proposal`，进行签名和转化为一个 `protobuf` 提案消息；
- 通过 `EndorserClient` 经由 `grpc` 通道发送给 `peer` 进行背书。

生成 `ChaincodeDeploymentSpec` 结构

图 1.16.1.3.1 - *ChaincodeDeploymentSpec* 结构

其中，*ChaincodeDeploymentSpec* 结构的 *CodePackage* 变量包括所调用的 *chaincode* 的代码和所需要的环境代码（例如整个 `$GOPATH/src` 目录下数据），为 *tar* 格式的二进制数据。

进行 endorsement

调用 *install* 方法，根据 CDS 来生成 *Install Proposal*。

首先，从本地 MSP 中拿到签名体身份（签名体在初始化阶段完成导入，包括证书和私钥）。

首先，创建 *Install Proposal*。

```
prop, _, err := utils.CreateInstallProposalFromCDS(msg, creator)
```

实际调用的是 `protos/utils/proptools.go` 中的 `createProposalFromCDS` 方法，创建一个对 LSCC 的 *ChaincodeInvocationSpec*，然后基于它创建一个 *Proposal* 结构。在此期间，需要生成 transaction id（随机生成的 nonce 值和 creator 信息，一起进行摘要）。

Proposal 结构体内容被 `cf.Signer` 进行签名，生成 *SignedProposal* 消息。*SignedProposal* 通过 *endorsement* 客户端通过 *grpc* 发送到 *peer* 节点进行背书，正常会收到 *ProposalResponse* 消息。

instantiate.go

响应 `peer chaincode instantiate` 命令，生成智能合约容器，在 peer 节点上启动。

例如

```
peer chaincode instantiate -n test_cc -c '{"Args":["init","a","100","b","200"]}' -o orderer0:7050 -v 1.0
```

instantiate 支持包括 policy、channel、escc、vscc 在内更多的命令行参数。

命令调用 `chaincodeDeploy` 方法。

首先会调用 `InitCmdFactory` 方法，初始化 `EndosermmentClient`、`Signer`、`BroadcastClient` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。

之后，会调用 `instantiate` 方法，`instantiate` 方法的流程如下：

- 根据传入的各种参数，生成 `ChaincodeSpec`，注意 `instantiate` 和 `upgrade` 是支持 `policy`、`escc`、`vscc` 等参数。
- 生成 `ChaincodeDeploymentSpec` 结构。
- 根据 `CDS`、签名实体、策略、通道、`escc`、`vscc` 等信息，创建一个 `LSCC` 的 `ChaincodeInvocationSpec`，根据这个 `CIS`，添加上 `TxID`（随机数+签名实体，进行 hash），创建一个 `Proposal` 出来。
- 根据签名实体，对 `Proposal` 进行签名。
- 调用 `EndorserClient`，发送 `gRPC` 消息，将签名后 `Proposal` 发给指定的 `peer`。
- 根据 `peer` 的返回，创建一个 `Envelop` 结构并进行签名。
- 将 `Envelop` 通过 `gRPC` 通道发给 `orderer`。

invoke.go

响应 `peer chaincode invoke` 命令，执行某个 `chaincode` 中操作。

例如

```
peer chaincode invoke -n test_cc -c '{"Args":["invoke","a","b","10"]}' -o orderer0:7050
```

命令会调用 `chaincodeQuery`。

首先会调用 `InitCmdFactory`，初始化 `Endosermmentclient`、`Signer` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 `chaincodeInvokeOrQuery` 方法。

`chaincodeInvokeOrQuery` 方法主要过程如下：

- 生成 `ChaincodeSpec`。
- 根据 `CS`、`chainID`、签名实体等，生成 `ChaincodeInvocationSpec`。
- 根据 `CIS`，生成 `Proposal`，并进行签名。
- 签名后，通过 `endorserClient` 发送给指定的 `peer`。
- 利用获取到的 `Response`，创建 `SignedTX`，发送给 `orderer`。
- 成功的话，输出成功消息，注意 `invoke` 是异步操作，无法获取到执行结果。

注意 `invoke` 和 `query` 的区别，`query` 不需要创建 `SignedTx` 发送到 `orderer`，而且会返回结果。

package.go

响应 `peer chaincode package` 命令，将某个 `chaincode` 打包为 `deployment` 的 `spec`。

例如

```
$ peer chaincode package -n test_cc -c '{"Args":["init","a","100","b","200"]}' -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v 1.0 test_cc_1.0.pkg
```

命令会调用 `chaincodePackage` 方法。

首先会调用 `InitCmdFactory`，初始化 `Signer` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。

之后主要过程如下：

- 根据传入的各种参数，生成 `ChaincodeSpec`。
- 生成 `ChaincodeDeploymentSpec` 结构。
- 根据 `CDS` 等创建 签名后的 `ChaincodeDeploymentSpec`，并封装为 `Envelop` 结构（其中数据是一个 `SignedChaincodeDeploymentSpec`）。
- 将 `Envelop` 结构写到本地指定的文件中。

query.go

响应 `peer chaincode query` 命令，查询某个 `chaincode` 中变量。

例如

```
peer chaincode query -n test_cc -c '{"Args":["query","a"]}'
```

命令会调用 `chaincodeQuery`。

首先会调用 `InitCmdFactory`，初始化 `Endosermmentclient`、`Signer` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 `chaincodeInvokeOrQuery` 方法。

`chaincodeInvokeOrQuery` 方法主要过程如下：

- 生成 `ChaincodeSpec`。
- 根据 `CS`、`chainID`、签名实体等，生成 `ChaincodeInvocationSpec`。
- 根据 `CIS`，生成 `Proposal`，并进行签名。
- 签名后，通过 `endorserClient` 发送给指定的 `peer`。
- 成功的话，获取到 `ProposalResponse`，打印出 `proposalResp.Response.Payload`。

注意 `invoke` 和 `query` 的区别，`query` 不需要创建 `SignedTx` 发送到 `orderer`，而且会返回结果。

signpackage.go

upgrade.go

响应 `peer chaincode upgrade` 命令，升级某个 `chaincode` 到新的版本。

例如

```
$ peer chaincode upgrade -n test_cc -o orderer0:7050 -c '{"Args":["init","a","100","b","200"]}' -v 1.1
```

命令会调用 `chaincodeUpgrade`。

首先会调用 `InitCmdFactory`，初始化 `Endosermmentclient`、`Signer`、`OrdererClient` 等结构。这一步对于所有 `chaincode` 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 `upgrade` 方法。

`upgrade` 方法主要过程如下：

- 根据传入的各种参数，生成 `ChaincodeSpec`，注意 `instantiate` 和 `upgrade` 是支持 `policy`、`escc`、`vscc` 等参数。
- 生成 `ChaincodeDeploymentSpec` 结构。
- 根据 `CDS`、签名实体、策略、通道、`escc`、`vscc` 等信息，创建一个 `LSCC` 的 `ChaincodeInvocationSpec`，根据这个 `CIS`，添加上 `TxID`（随机数+签名实体，进行 hash），创建一个 `Proposal` 出来；
- 根据签名实体，对 `Proposal` 进行签名。
- 调用 `EndorserClient`，发送 `gRPC` 消息，将签名后 `Proposal` 发给指定的 `peer`。
- 根据 `peer` 的返回，创建一个 `Envelop` 结构（`SignedTx`）并进行签名。
- 将 `Envelop` 通过 `gRPC` 通道发给 `orderer`。

channel

包括 create、fetch、join、list 等命令。

所有命令都会先执行初始化方法 `InitCmdFactory`，初始化需要的 `EndorserClient` 或 `OrdererClient`。

- `create`：创建一个新的 channel。根据指定的配置交易文件路径或默认配置，创建 `Envelope` 结构，发送给 Orderer，并将所指定创建通道中的初始区块写到本地文件 `chainID.block`。
- `fetch`：获取指定通道的初始区块。从 Orderer 获取指定通道的初始区块，写到本地文件 `chainID.block`。
- `join`：让 peer 加入某个通道。读取本地的 `block` 文件，生成一个 cscc 的 `JoinChain` 交易 spec，进一步封装为一个 `ChaincodeInvocationSpec`，创建 `CONFIG` 类型的 proposal，并签名，通过 `EndorserClient` 发给 peer。
- `list`：列出 peer 所加入的所有通道。生成一个 cscc 的 `GetChannels` 交易 spec，进一步封装为一个 `ChaincodeInvocationSpec`，创建 `ENDORSER_TRANSACTION` 类型的 proposal，并签名，通过 `EndorserClient` 发给 peer。

各子命令的参数支持情况如下。

命令	-b 区块文件路径	-c chainID	-f 配置交易文件路径	-o Orderer	--tls	--cafile tls 证书路径
create	不支持	必需	可选	必需	可选	可选
fetch	不支持	必需	不支持	必需	可选	可选
join	必需	不支持	不支持	不支持	不支持	不支持
list	不支持	不支持	不支持	不支持	不支持	不支持

channel.go

peer channel 相关命令的入口。

进一步支持 join、create、fetch、list 等子命令。

```
// Cmd returns the cobra command for Node
func Cmd(cf *ChannelCmdFactory) *cobra.Command {
    AddFlags(channelCmd)
    channelCmd.AddCommand(joinCmd(cf))
    channelCmd.AddCommand(createCmd(cf))
    channelCmd.AddCommand(fetchCmd(cf))
    channelCmd.AddCommand(listCmd(cf))

    return channelCmd
}
```

所有 channel 子命令都会先调用 InitCmdFactory 来进行必要的初始化，根据命令需求来生成 endorserClient、BroadcastClient 和 DeliverClient。

create.go

`peer channel create` 命令的入口。

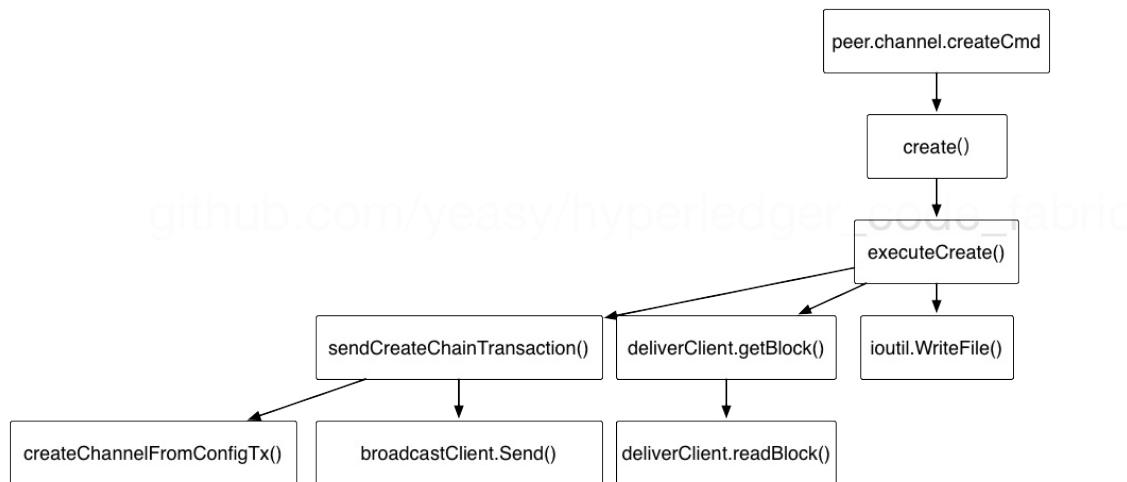


图 1.16.2.2.1 - *peer channel create*

调用 `create` 方法，首先通过 `InitCmdFactory` 进行初始化，然后调用 `executeCreate`。

`executeCreate` 流程包括：

- 客户端调用 `sendCreateChainTransaction`，检查指定的配置交易文件，或者利用默认配置，构造一个创建应用通道的配置交易结构，封装为 `Envelope`，指定类型为 `CONFIG_UPDATE`。
- 客户端发送配置交易到 `Orderer` 服务。
- `Orderer` 收到 `CONFIG_UPDATE` 消息后，检查指定的通道还不存在，则开始新建过程，构造该应用通道的初始区块。
 - `Orderer` 首先检查通道应用（`Application`）配置中的组织都在创建的联盟（`Consortium`）配置组织中。
 - 之后从系统通道中获取 `Orderer` 相关的配置，并创建应用通道配置，对应 `mod_policy` 为系统通道配置中的联盟指定信息。
 - 接下来根据 `CONFIG_UPDATE` 消息的内容更新获取到的配置信息。所有配置发生变更后版本号都要更新。
 - 最后，发送到系统通道中，完成应用通道的创建过程。
- 客户端从 `Orderer` 获取到该应用通道的初始区块。
- 客户端将收到的区块写入到本地的 `chainID + ".block"` 文件。这个文件后续会被需要加入到通道的节点使用。

`sendCreateChainTransaction` 方法会检查，如果提供了 `tx` 文件了，则直接读取为 `Envelope` 结构；如果不存在，则通过默认值来创建一个 `Envelope` 结构。之后将 `Envelope` 结构发给 `orderer`。

`Envelope` 结构中 `Payload.Data` 是一个 `ConfigUpdateEnvelope` 结构。

deliverclient.go

fetchconfig.go

join.go

join 过程也十分简单，主要是 peer 完成。

同样的，首先通过 `InitCmdFactory` 进行初始化。

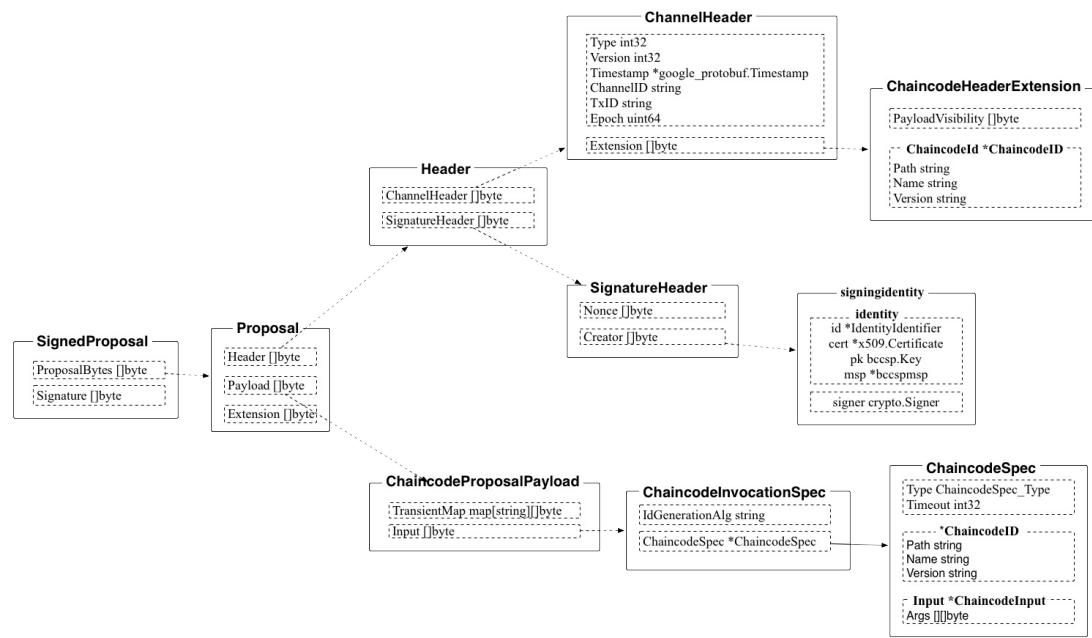


图 1.16.2.5.1 - *Signed Proposal* 结构

- 客户端首先创建一个 `ChaincodeSpec` 结构，其 `input` 中的 `Args` 第一个参数是 `CSCC.JoinChain`（指定调用配置链码的操作），第二个参数为所加入通道的初始区块。
- 利用 `CS` 构造一个 `ChaincodeInvocationSpec` 结构。
- 利用 `CIS`，创建 `Proposal` 结构并进行签名，`channel` 头部类型为 `CONFIG`。
- 客户端通过 gRPC 将 `Proposal` 发给 `Endorser`，调用 `ProcessProposal()` 方法进行处理，主要通过配置系统链码进行本地链的初始化工作，并通过获取通道的配置来得到 `Ordering` 服务地址。
- 初始化完成后，即可收到来自通道内的 `Gossip` 消息等。

list.go

同样的，首先通过 `InitCmdFactory` 进行初始化。

主要实现过程如下：

- 客户端首先创建一个 `ChaincodeSpec` 结构，其 `input` 中的 `Args` 第一个参数是 `CSCC.GetChannels`（指定调用配置链码的操作）。
- 利用 `CS` 构造一个 `ChaincodeInvocationSpec` 结构。
- 利用 `CIS`，创建 `Proposal` 结构并进行签名，`channel` 头部类型为 `ENDORSER_TRANSACTION`。
- 客户端通过 gRPC 将 `Proposal` 发给 `Endorser`（所操作的 `Peer`），调用 `ProcessProposal()` 方法进行处理，主要是通过配置系统链码查询本地链信息并返回。
- 命令执行成功后，客户端会受到来自 `Peer` 端的回复消息，从其中提取出应用通道列表信息并输出。

signconfigtx.go

update.go

clilogging

common.go

getlevel.go

logging.go

revertlevels.go

setlevel.go

common

common.go

mockclient.go

ordererclient.go

gossip

mocks

mocks.go

mcs.go

sa.go

node

负责 `peer node` 子命令。

调用其他功能模块来具体实现。

- `start`：启动节点。
- `stop`：停止节点。
- `status`：查看节点状态。

启动服务

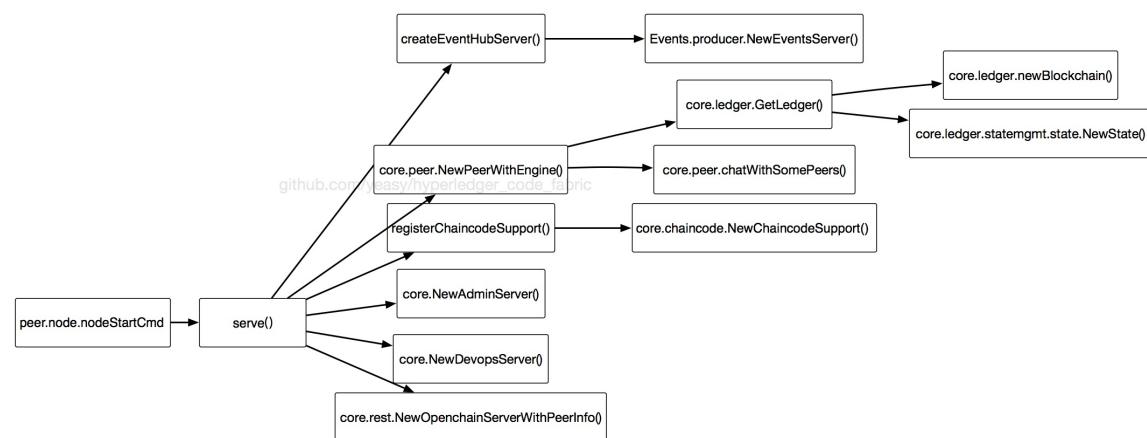
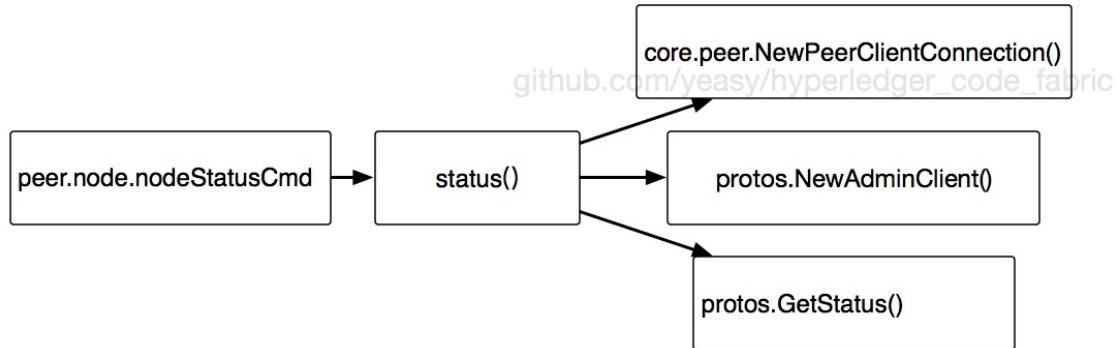
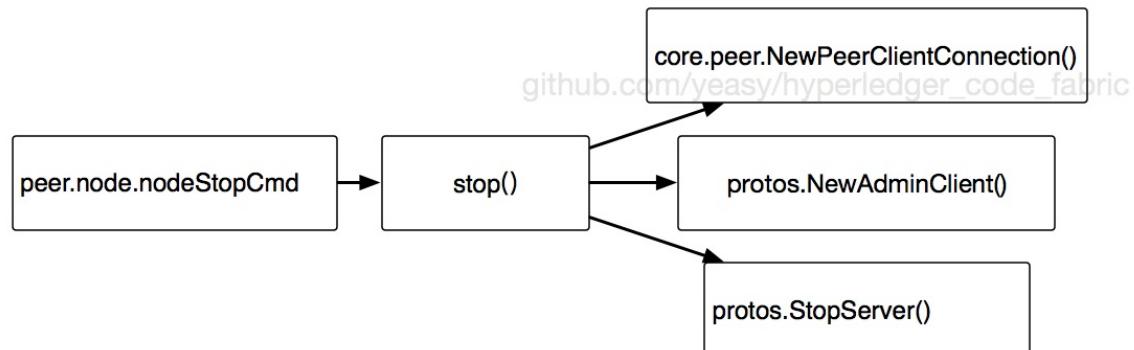


图 1.16.6.1 - `peer node start`

查看状态

图 1.16.6.2 - *peer node status*

停止服务

图 1.16.6.3 - *peer node stop*

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。包括 `EventsServer` 服务、`chaincodesupport` 服务、`admin` 服务、`endorser` 服务、`gossip` 服务。

- `EventsServer` 服务 (`events.producer.EventsServer`)：提供 Chat GRPC 调用。
- `ChaincodeSupport` (`core.chaincode.ChaincodeSupport`) 服务：提供 `Execute`、`Launch`、`Register`、`Stop` 等方法。
- `ServerAdmin` 服务 (`core.ServerAdmin`)：提供 `GetStatus`、`StartServer`、`StopServer`、`GetModuleLogLevel`、`SetModuleLogLevel` 等方法。
- `Endorser` 服务 (`core.endorser.Endorser`)：提供 `ProcessProposal` 方法。
- `GossipService` 服务 (`gossip.service.GossipService`)：提供 `NewConfigEventer`、`InitializeChannel`、`GetBlock`、`AddPayload` 方法。

`startCmd()` 方法调用 `serve()` 方法。

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 gRPC 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
}
```

version

version.go

main.go

主服务，所有 `peer` 命令都从这里入口。

各个子命令的处理在对应子包中，如

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

`main` 函数主要完成子命令的注册和一些初始化配置工作，为执行子命令准备好环境，包括：

- 调用 `InitConfig()` 从本地的 yaml、环境变量以及命令行选项中读取 `Peer` 命令相关的配置信息；
- 之后注册各个子命令；
- 最后调用 `InitCrypto()` 从本地读入 MSP 配置文件和 BCCSP 配置，初始化 MSP 部分。初始化会创建本地的一个 `bccspmsp` 结构 (`msp/mspimpl.go`)，然后利用本地读取的 MSP (包括各种证书和私钥等) 和 BCCSP 配置进行初始化。

`peer` 的 MSP 文件路径从 `peer.mspConfigPath` 变量读取 (相对路径，前面会拼接上配置文件路径，默认为 `$FABRIC_CFG_PATH/msp`)；默认的 `mspID` 是 `peer.localMspId` (`DEFAULT`)。BCCSP 配置从 `peer.BCCSP` 中读取。

MSP 初始化

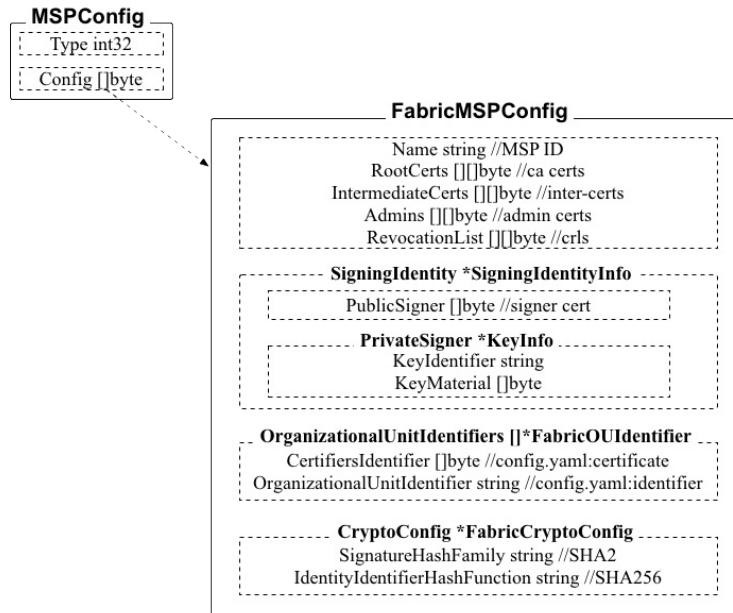


图 1.16.8.1 - MSP 配置结构

通过 peer/common/common.go 文件中的 InitCrypto 方法完成。

```

func InitCrypto(mspMgrConfigDir string, localMSPID string) error {
    // Init the BCCSP
    var bccspConfig *factory.FactoryOpts
    err := viperutil.EnhancedExactUnmarshalKey("peer.BCCSP", &bccspConfig)
    if err != nil {
        return fmt.Errorf("Could not parse YAML config [%s]", err)
    }

    err = mspmgmt.LoadLocalMsp(mspMgrConfigDir, bccspConfig, localMSPID)
    if err != nil {
        return fmt.Errorf("Fatal error when setting up MSP from directory %s: err %s\n",
            mspMgrConfigDir, err)
    }

    return nil
}

```

其中，默认的 BCCSP 配置从 `peer.BCCSP` 中读取，示例配置为

```

BCCSP:
  Default: SW
  SW:
    # TODO: The default Hash and Security level needs refactoring to be
    # fully configurable. Changing these defaults requires coordination
    # SHA2 is hardcoded in several places, not only BCCSP
  Hash: SHA2
  Security: 256
  # Location of Key Store, can be subdirectory of SbftLocal.DataDir
  FileKeyStore:
    # If "", defaults to 'mspConfigPath'/keystore
    # TODO: Ensure this is read with fabric/core/config.GetPath() once rea
dy
  KeyStore:

```

通过 `msp/mgmt/mgmt.go` 中的 `LoadLocalMsp` 方法来导入 MSP 相关的文件和配置。
`LoadLocalMsp` 会先读入各种文件和初始化配置，然后按照配置进行初始化。

首先，调用 `msp/cofnigbuilder.go` 中的 `GetLocalMspConfig` 方法，进一步调用 `getMspConfig` 方法。`getMspConfig` 方法导入所有提供的 PEM 格式的证书和密钥文件，并从 MSP 配置目录下查找 `config.yaml` 并读取 `OUIdentifier` 信息。

之后通过 `GetLocalMSP().Setup(conf)` 进行配置。`GetLocalMSP()` 会通过 `msp.NewBccspMsp()` 生成一个 `msp.bccspmsp` 对象。之后，调用 `msp/mspimpl.go` 中的 `Setup` 方法，将读入的证书文件等配置写到 `msp.bccspmsp` 对象中。

至此，完成 MSP 的初始化工作。

proposals

r1

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 `protoc` 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 \
--javabean_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.go

common.pb.go

common.proto

configtx.go

configtx.pb.go

configtx.proto

configuration.go

configuration.pb.go

configuration.proto

ledger.pb.go

ledger.proto

policies.go

policies.pb.go

policies.proto

signed_data.go

gossip

extensions.go

message.pb.go

message.proto

ledger

queryresult

kv_query_result.pb.go

kv_query_result.proto

rwset

kvrwset

tests 包

rwset.pb.go

rwset.proto

msp

msp

identities.pb.go

identities.proto

msp_config.go

msp_config.pb.go

msp_config.proto

msp_principal.go

msp_principal.pb.go

msp_principal.proto

orderer

ab.pb.go

ab.proto

configuration.go

configuration.pb.go

configuration.proto

kafka.pb.go

kafka.proto

peer

peer

admin.pb.go

admin.proto

chaincode.pb.go

chaincode.proto

chaincode_event.pb.go

chaincode_event.proto

chaincode_shim.pb.go

chaincode_shim.proto

chaincodeunmarshall.go

configuration.go

configuration.pb.go

configuration.proto

events.pb.go

events.proto

init.go

peer.pb.go

peer.proto

proposal.go

proposal.pb.go

proposal.proto

proposal_response.go

proposal_response.pb.go

proposal_response.proto

query.pb.go

query.proto

signed_cc_dep_spec.pb.go

signed_cc_dep_spec.proto

transaction.go

transaction.pb.go

transaction.proto

testutils

txtestutils.go

utils

包含了基本的数据结构的使用（可以理解为数据的编码和解码）使用的
是 `github.com/golang/protobuf/proto` 下的 `Marshal()` 和 `Unmarshal()` 方法

blockutils.go

commonutils.go

主要的函数：

字节数据解码为 Payload : UnmarshalPayload([]byte) return : *cb.Payload, err

字节数据解码为 Envelope : UnmarshalEnvelope([]byte) return : *cb.Envelope, err

从区块中提取指定序号的 Envelope : ExtractEnvelope(cb.Block, int) return : cb.Envelope, err

从 Envelope 中提取 Payload : ExtractPayload(cb.Envelope) return : cb.Payload, err

字节数据解码为 ChannelHeader : UnmarshalChannelHeader([]byte) return :

*cb.ChannelHeader, err

字节数据解码为 ChaincodeID : UnmarshalChaincodeID([]byte) return : *cb.ChaincodeID, err

对消息签名 : SignOrPanic(*LocalSigner, []byte) return : []byte Ps : 实际上 LocalSigner 的
Sign() 方法直接返回原数据

判断区块是否包含配置更新交易 : IsConfigBlock(*cb.Block) return : bool 解释 : 提取块的第
一个 Envelope , 提取该 Envelope 的 Payload , 解码 ChannelHeader , 判断类型

其他的函数 : 创建新 ChannelHeader 、 SignatureHeader 等

proputils.go

CreateChaincodeProposal

- 节点身份证书序列化后与随机数计算出交易ID。
- 构建ChaincodeHeaderExtension（含有ChaincodeID）。
- 构建ChaincodeProposalPayload（含有ChaincodeSpec）。
- 获取事件戳。
- 构建Header。
- 返回Proposal。

txutils.go

CreateSignedTx(proposal, signer, resps)

- 按照下图的结构封装交易（其中有很重要的一步是按字节比对所有的 `ProposalResponse.Payload` 是否相等，处理多个Endorser情况下出现模拟结果不一样的问题，直接返回nil）。

Ps. 传入的`resps`是一个 `ProposalResponse` 的数组。

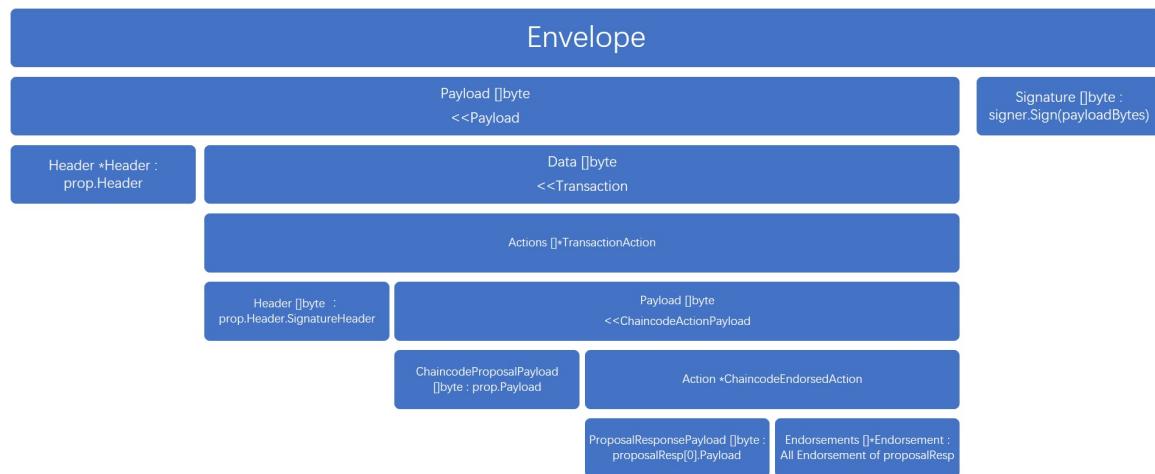


图 1.18.8.4.1 - 交易结构

release

release

templates

get-byfn.in

get-docker-images.in

release_notes

v1.0.0-rc1.txt

v1.0.0.txt

sampleconfig

提供了一些样例证书文件和配置文件。

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
```


msp

msp

admincerts

admincert.pem

cacerts

cacert.pem

keystore

key.pem

signcerts

peer.pem

tlscacerts

cert.pem

config.yaml

configtx.yaml

core.yaml

peer 节点相关的样例配置。

orderer.yaml

scripts

一些辅助脚本，多数为外部 Makefile 调用。

bootstrap-1.0.0-alpha2.sh

bootstrap-1.0.0-beta.sh

bootstrap-1.0.0-rc1.sh

bootstrap-1.0.0.sh

bootstrap-1.0.1.sh

changelog.sh

check_license.sh

check_spelling.sh

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。

install_behave.sh

test

用于测试的一些脚本。

chaincodes

AuctionApp

art.go

image_proc_api.go

table_api.go

BadImport

main.go

envsetup

channel-artifacts

docker-compose.yaml

generateCfgTrx.sh

feature

configs

configtx.yaml

crypto.yaml

docker-compose

docker-compose-kafka.yml

docker-compose-solo.yml

steps

basic_impl.py

compose_util.py

config_util.py

endorser_impl.py

endorser_util.py

orderer_impl.py

orderer_util.py

bootstrap.feature

environment.py

orderer.feature

peer.feature

README.rst

regression

daily

chaincodeTests

Example.py

ledger_lte.py

README.rst.orig

runDailyTestSuite.sh

SampleScriptFailTest.sh

SampleScriptPassTest.sh

systest_pte.py

testAuctionChaincode.py

TestPlaceholder.sh

release 包

byfn_release_tests.py

e2e_sdk_release_tests.py

make_targets_release_tests.py

run_byfn_cli_release_tests.sh

run_e2e_java_sdk.sh

run_e2e_node_sdk.sh

run_make_targets.sh

run_node_sdk_byfn.sh

runReleaseTestSuite.sh

weekly

weekly

runGroup1.sh

runGroup2.sh

runGroup3.sh

runGroup4.sh

systest_pte.py

testAuctionChaincode.py

tools

AuctionApp

api_driver.sh

LTE

chainmgmt

common

experiments

scripts

OTE

README.rst

PTE

SCFiles

userInputs

chaincode_sample.go

pte-execRequest.js

pte-main.js

pte-util.js

pte_driver.sh

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