// 超时时间包括建立连接和等待返回结果得时间

func getClient(url string) \*rpcclient.JSONRPCClient {

return rpcclient.NewJSONRPCClientExWithTimeout(url, "", true, 5)

}

func abciQueryAndParse(url, path string, data interface{}) (err error) {

var result \*ResultABCIQuery

if result, err = abciQuery(url, path); err != nil {

return

}

return jsoniter.Unmarshal(result.Response.Value, data)

}

func blockQuery(url string, height int64) (resultBlock \*ResultBlock, err error) {

resultBlock = new(ResultBlock)

client := getClient(url)

\_, err = client.Call("block", map[string]interface{}{"height": height}, resultBlock)

return

}

func blockResultQuery(url string, height int64) (resultBlockResults \*ResultBlockResults, err error) {

resultBlockResults = new(ResultBlockResults)

client := getClient(url)

\_, err = client.Call("block\_results", map[string]interface{}{"height": height}, resultBlockResults)

return

}

func abciInfoQuery(url string) (resultABCIInfo \*ResultABCIInfo, err error) {

resultABCIInfo = new(ResultABCIInfo)

client := getClient(url)

\_, err = client.Call("abci\_info", nil, resultABCIInfo)

return

}

func generateTx(contract types.Address, method uint32, params []interface{}, nonce uint64, gaslimit int64, note, privKey, toChainID string) string {

items := tx3.WrapInvokeParams(params...)

message := types.Message{

Contract: contract,

MethodID: method,

Items: items,

}

payload := tx3.WrapPayload(nonce, gaslimit, note, message)

return tx3.WrapTxEx(toChainID, payload, privKey)

}

func queryIBCContract(url, orgID string) (\*Contract, error) {

versionList, err := queryVersionList(url, "ibc", orgID)

if err != nil {

return nil, errors.New("query ibc version list failed：" + err.Error())

}

if len(versionList.ContractAddrList) == 0 {

return nil, errors.New("can not get ibc contract version list")

}

remoteBlockHeight, err := queryCurrentHeight(url)

if err != nil {

return nil, err

}

for i := len(versionList.ContractAddrList) - 1; i >= 0; i-- {

contract, err := queryContract(url, versionList.ContractAddrList[i])

if err != nil {

return nil, errors.New("query ibc contract list failed：" + err.Error())

}

if contract.EffectHeight <= remoteBlockHeight {

return contract, nil

}

}

return nil, errors.New("can not get valid ibc contract address")

}

func getCurrentNodePrivKey(config \*cfg.Config) crypto.PrivKey {

privValidatorFile := config.PrivValidatorFile()

return pvm.LoadFilePV(privValidatorFile).PrivKey

}

func getOtherOrgID(pktsProofs []\*PktsProof, genesisOrgID string) (otherOrgID string) {

for \_, pktProof := range pktsProofs {

for \_, p := range pktProof.Packets {

if p.OrgID != genesisOrgID {

otherOrgID = p.OrgID

return

}

}

}

return

}

func splitQueueID(queueID string) (fromChainID, toChainID string) {

idList := strings.Split(queueID, "->")

return idList[0], idList[1]

}

func makeQueueID(fromChainID, toChainID string) string {

return fromChainID + "->" + toChainID

}

func getCurrentNodeAddress(config \*cfg.Config) string {

privValidatorFile := config.PrivValidatorFile()

return pvm.LoadFilePV(privValidatorFile).GetAddress()

}

func getMainChaidID(chainID string) string {

if strings.Contains(chainID, "[") {

return chainID[:strings.Index(chainID, "[")]

}

return chainID

}

func replaceChainID(address, queueID string) string {

localChainID, remoteChainID := splitQueueID(queueID)

return strings.Replace(address, localChainID, remoteChainID, 1)

}

func queryContract(url, address string) (\*Contract, error) {

contract := new(Contract)

err := abciQueryAndParse(url, keyOfContract(address), contract)

return contract, err

}

func queryVersionList(url, contractName, orgID string) (\*ContractVersionList, error) {

versionList := new(ContractVersionList)

err := abciQueryAndParse(url, keyOfVersionList(contractName, orgID), versionList)

return versionList, err

}

func queryCurrentHeight(url string) (int64, error) {

result, err := abciInfoQuery(url)

if err != nil {

return 0, err

}

return result.Response.LastBlockHeight, nil

}

func queryIBCMsgIndex(url, queueID string, seq uint64) (\*MessageIndex, error) {

msgIndex := new(MessageIndex)

err := abciQueryAndParse(url, keyOfMessageIndex(queueID, seq), msgIndex)

return msgIndex, err

}

func querySequence(url, queueID string) (sequence uint64, err error) {

resultQuery := new(ResultABCIQuery)

resultQuery, err = abciQuery(url, keyOfSequence(queueID))

if err != nil {

return

}

if len(resultQuery.Response.GetValue()) == 0 {

return 0, nil

}

err = jsoniter.Unmarshal(resultQuery.Response.GetValue(), &sequence)

return

}

func queryAccountNonce(url, address string) (uint64, error) {

type account struct {

Nonce uint64 `json:"nonce"`

}

result, err := abciQuery(url, keyOfAccountNonce(address))

if err != nil {

return 0, err

}

if len(result.Response.GetValue()) == 0 {

return 0, nil

}

acc := new(account)

if err := jsoniter.Unmarshal(result.Response.GetValue(), acc); err != nil {

return 0, err

}

return acc.Nonce, nil

}

type QueueRelay struct {

LocalURL string

RemoteURLs []string

QueueID string

genesisOrgID string

signalChan chan bool

remoteSeq uint64

currentRoundURL string

remoteIBC \*IBCContractInfo

currentNode \*CurrentNodeInfo

logger log.Logger

}

type CurrentNodeInfo struct {

Address string

HexPrivKey string

Nonce uint64

}

type IBCContractInfo struct {

Address string

MethodID uint32

}

// Start start relay goroutine for queueID

func (qr \*QueueRelay) Start() {

running := false

lastResult := false

for {

if running {

lastResult = qr.carry(lastResult)

if lastResult == false {

running = false

}

} else {

running = <-qr.signalChan

lastResult = false

}

// 如果 signalChan 中存在多个数据，全部读取出来，取最后的状态

for {

select {

case running = <-qr.signalChan:

continue

default:

}

break

}

}

}

func (qr \*QueueRelay) calcRoundURL() {

r := make([]byte, 4)

\_, e := rand.Read(r)

if e != nil {

panic(e)

}

buf := bytes.NewBuffer(r)

var random uint8

\_ = binary.Read(buf, binary.BigEndian, &random)

qr.currentRoundURL = qr.RemoteURLs[int(random)%len(qr.RemoteURLs)]

}

func (qr \*QueueRelay) carry(lastResult bool) bool {

qr.calcRoundURL()

if qr.remoteIBC == nil {

if err := qr.getTargetIBCContract(); err != nil {

qr.logger.Debug("RELAY", "query remote ibc failed", err)

return false

}

}

if qr.currentNode.Nonce == 0 {

if err := qr.getNonce(); err != nil {

qr.logger.Debug("RELAY", "get nonce failed", err)

return false

}

}

pktsProof := qr.collectIBCPktsProof(lastResult)

if len(pktsProof) == 0 {

// 此时说明还没有发生跨链交易

qr.logger.Debug("RELAY", "no packets")

return false

}

if err := qr.sendIBCPackets(pktsProof); err != nil {

// 不能发送跨链交易到目标链

qr.logger.Debug("RELAY", "send tx failed", err)

return false

}

return true

}

func (qr \*QueueRelay) sendIBCPackets(pktsProofs []\*PktsProof) error {

tx, err := qr.packTx(pktsProofs)

if err != nil {

return err

}

result := new(ResultBroadcastTxCommit)

client := getClient(qr.currentRoundURL)

if \_, err = client.Call(

"broadcast\_tx\_commit",

map[string]interface{}{"tx": []byte(tx)},

result); err != nil {

return err

}

return qr.processTxResult(result)

}

func (qr \*QueueRelay) processTxResult(result \*ResultBroadcastTxCommit) error {

if result.CheckTx.Code == 200 && result.DeliverTx.Code == 200 {

qr.currentNode.Nonce++

return nil

}

qr.remoteIBC = nil

qr.currentNode.Nonce = 0

if result.CheckTx.Code != 200 {

return errors.New(result.CheckTx.Log)

} else {

return errors.New(result.DeliverTx.Log)

}

}

func (qr \*QueueRelay) packTx(pktsProofs []\*PktsProof) (string, error) {

params := make([]interface{}, 1)

params[0] = pktsProofs

otherOrgID := getOtherOrgID(pktsProofs, qr.genesisOrgID)

ibcContract := qr.remoteIBC.Address

to := ""

if len(otherOrgID) != 0 {

to = otherOrgID + "." + ibcContract

} else {

to = qr.genesisOrgID + "." + ibcContract

}

\_, toChainID := splitQueueID(qr.QueueID)

return generateTx(to, qr.remoteIBC.MethodID, params, qr.currentNode.Nonce+1, 0, "", qr.currentNode.HexPrivKey, toChainID), nil

}

func (qr \*QueueRelay) collectIBCPktsProof(lastResult bool) (pktsProofs []\*PktsProof) {

if lastResult == false {

if seq, err := qr.getRemoteSequence(); err != nil {

return

} else {

qr.remoteSeq = seq

}

}

for {

height := qr.getHeight(qr.remoteSeq + 1)

if height == 0 {

return

}

pktsProof, ibcMsgCount := qr.getPacketsProof(height)

if pktsProof == nil {

return

}

qr.remoteSeq += ibcMsgCount

pktsProofs = append(pktsProofs, pktsProof)

if len(pktsProofs) == 10 {

return

}

}

}

func (qr \*QueueRelay) getPacketsProof(height int64) (\*PktsProof, uint64) {

blkResults, err := qr.getBlockResult(height)

if err != nil {

return nil, 0

}

pktsProof, err := qr.getProof(height + 1)

if err != nil {

qr.logger.Warn("RELAY", "get proof err", err)

return nil, 0

}

pktsProof.Packets = qr.getIBCPackets(blkResults)

return pktsProof, uint64(len(pktsProof.Packets))

}

func (qr \*QueueRelay) getProof(headerHeight int64) (pktsProof \*PktsProof, err error) {

var (

block1, block2 \*ResultBlock

)

if block1, err = qr.getBlock(headerHeight); err != nil {

return

}

if block2, err = qr.getBlock(headerHeight + 1); err != nil {

return

}

headerBytes, err := jsoniter.Marshal(block1.BlockMeta.Header)

if err != nil {

return

}

var header Header

err = jsoniter.Unmarshal(headerBytes, &header)

if err != nil {

return

}

pktsProof = new(PktsProof)

pktsProof.Header = header

preCommitBytes, err := jsoniter.Marshal(block2.Block.LastCommit.Precommits)

var preCommits []Precommit

err = jsoniter.Unmarshal(preCommitBytes, &preCommits)

if err != nil {

return

}

pktsProof.Precommits = preCommits

return

}

func (qr \*QueueRelay) getRemoteSequence() (sequence uint64, err error) {

sequence, err = querySequence(qr.currentRoundURL, qr.QueueID)

return

}

func (qr \*QueueRelay) getHeight(sequence uint64) int64 {

msgIndex, err := queryIBCMsgIndex(qr.LocalURL, qr.QueueID, sequence)

if err != nil {

qr.logger.Debug("RELAY", "query msgIndex err", err)

return 0

}

return msgIndex.Height

}

func (qr \*QueueRelay) getBlockResult(height int64) (abciRes \*ABCIResponses, err error) {

response, err := blockResultQuery(qr.LocalURL, height)

if err != nil {

return

}

abciRes = response.Results

return

}

func (qr \*QueueRelay) getBlock(height int64) (resultBlock \*ResultBlock, err error) {

resultBlock, err = blockQuery(qr.LocalURL, height)

return

}

func (qr \*QueueRelay) getIBCPackets(abciResponse \*ABCIResponses) (packets []Packet) {

for \_, deliverTx := range abciResponse.DeliverTx {

packets = append(packets, qr.getPacketsFromTx(deliverTx)...)

}

return

}

func (qr \*QueueRelay) getPacketsFromTx(deliverTx \*types.ResponseDeliverTx) []Packet {

if deliverTx.Code != types.CodeTypeOK {

return nil

}

packets := make([]Packet, 0)

for \_, tag := range deliverTx.Tags {

if strings.HasSuffix(string(tag.Key), "/ibc::packet/"+qr.QueueID) {

var receipt Receipt

err := jsoniter.Unmarshal(tag.Value, &receipt)

if err != nil {

panic(err)

}

var packet Packet

err = jsoniter.Unmarshal(receipt.Bytes, &packet)

if err != nil {

panic(err)

}

packets = append(packets, packet)

}

}

return packets

}

func (qr \*QueueRelay) getTargetIBCContract() error {

contract, err := queryIBCContract(qr.currentRoundURL, qr.genesisOrgID)

if err != nil {

qr.logger.Debug("RELAY", "query ibc err", err)

return err

}

var item Method

for \_, methodItem := range contract.Methods {

if strings.HasPrefix(methodItem.ProtoType, "Input") {

item = methodItem

break

}

}

methodID, \_ := strconv.ParseUint(item.MethodID, 16, 64)

qr.remoteIBC = &IBCContractInfo{

Address: contract.Address,

MethodID: uint32(methodID),

}

return nil

}

func (qr \*QueueRelay) getNonce() error {

nonce, err := queryAccountNonce(qr.currentRoundURL, qr.currentNode.Address)

if err != nil {

return err

}

qr.currentNode.Nonce = nonce

return nil

}

type RelayController struct {

LocalURL string // local url

ChainIDToURLs sync.Map // chainID => openURLs

QueueIDToQueueRelay map[string]\*QueueRelay // queueID => QueueRelay

currentNodeAddress string

config \*cfg.Config

abciClient proxy.AppConns

logger log.Logger

}

var (

gRelay \*RelayController

initOnce sync.Once

)

// Init init relay controller

func Init(config \*cfg.Config, logger log.Logger, conns proxy.AppConns) \*RelayController {

initOnce.Do(func() {

temp := strings.Split(config.RPC.ListenAddress, ":")

localURL := "http://127.0.0.1:" + temp[len(temp)-1]

gRelay = &RelayController{

LocalURL: localURL,

currentNodeAddress: getCurrentNodeAddress(config),

config: config,

abciClient: conns,

logger: logger,

}

gRelay.init()

logger.Info("RELAY init", "gRelay", gRelay)

})

return gRelay

}

// GetRelayController get instance

func GetRelayController() \*RelayController {

return gRelay

}

// SetNewHeader determines whether to start or stop a relay by header

func (rc \*RelayController) SetNewHeader(header \*types.Header) {

if header.Relayer == nil {

return

}

if header.Relayer.Address == rc.currentNodeAddress {

rc.logger.Debug("RELAY SetNewHeader", "rc.startRelay()")

rc.startRelay()

} else {

rc.logger.Debug("RELAY SetNewHeader", "rc.stopRelay()",

fmt.Sprintf("expcted: %s obtain: %s", header.Relayer.Address, rc.currentNodeAddress))

rc.stopRelay()

}

}

// UpdateOpenURL update relay controller.ChainIDToURLS, overwrite existing data.

func (rc \*RelayController) UpdateOpenURL(chainID string, urls []string) {

rc.logger.Info("RELAY UpdateOpenURL", "chainID", chainID, "urls", urls)

localChainID := rc.getLocalChainID()

if localChainID == chainID {

return

}

if \_, ok := rc.ChainIDToURLs.Load(chainID); ok {

queueID := makeQueueID(localChainID, chainID)

qr, ok := rc.QueueIDToQueueRelay[queueID]

if !ok {

return

}

qr.RemoteURLs = urls

rc.QueueIDToQueueRelay[queueID] = qr

} else {

rc.addQueueRelay(localChainID, chainID, urls)

}

rc.ChainIDToURLs.Store(chainID, urls)

}

// relayControler initialize

func (rc \*RelayController) init() {

localChainID := rc.getLocalChainID()

if len(localChainID) == 0 {

panic("can not get local chainID.")

}

if strings.Contains(localChainID, "[") {

// side chain

rc.getMainChainURLs(getMainChaidID(localChainID))

} else {

// main chain

rc.getSideChainOpenURL()

}

}

func (rc \*RelayController) addQueueRelay(localChainID, toChainID string, urls []string) {

queueID := makeQueueID(localChainID, toChainID)

qr := QueueRelay{

LocalURL: rc.LocalURL,

RemoteURLs: urls,

QueueID: queueID,

genesisOrgID: gRelay.queryGenesisOrgID(),

signalChan: make(chan bool, 100),

currentNode: rc.getCurrentNode(queueID),

logger: rc.logger,

}

go qr.Start()

qr.signalChan <- true

if len(rc.QueueIDToQueueRelay) == 0 {

rc.QueueIDToQueueRelay = make(map[string]\*QueueRelay)

}

rc.QueueIDToQueueRelay[qr.QueueID] = &qr

rc.logger.Debug("RELAY addQueueRelay", "queueRelay", qr)

}

func (rc \*RelayController) getMainChainURLs(mainChainID string) {

var urls []string

rc.abciQueryAndParse(keyOfOpenURLs(mainChainID), &urls)

if len(urls) == 0 {

panic("can not get main chain URL")

}

rc.ChainIDToURLs.Store(mainChainID, urls)

}

func (rc \*RelayController) getSideChainOpenURL() {

sideChainIDs := rc.getSideChainIDs()

for \_, chainID := range sideChainIDs {

status := rc.getSideChainStatus(chainID)

if status != "ready" && status != "clear" {

continue

}

urls := rc.getOepnURLs(chainID)

rc.ChainIDToURLs.Store(chainID, urls)

}

}

func (rc \*RelayController) getOepnURLs(chainID string) []string {

urls := new([]string)

rc.abciQueryAndParse(keyOfOpenURLs(chainID), urls)

return \*urls

}

func (rc \*RelayController) getSideChainIDs() []string {

sideChainIDs := new([]string)

rc.abciQueryAndParse(keyOfSideChainIDs(), &sideChainIDs)

return \*sideChainIDs

}

func (rc \*RelayController) getSideChainStatus(chainID string) string {

ci := new(ChainInfo)

rc.abciQueryAndParse(keyOfChainInfo(chainID), ci)

return ci.Status

}

func (rc \*RelayController) abciQueryAndParse(key string, data interface{}) {

r, err := rc.abciClient.Query().QuerySync(types2.RequestQuery{

Path: key,

})

if err != nil {

panic(err)

}

if len(r.GetValue()) == 0 {

return

}

\_ = jsoniter.Unmarshal(r.GetValue(), data)

}

func (rc \*RelayController) startRelay() {

localChainID := rc.getLocalChainID()

if len(rc.QueueIDToQueueRelay) == 0 {

rc.QueueIDToQueueRelay = make(map[string]\*QueueRelay)

rc.ChainIDToURLs.Range(func(chanID, urls interface{}) bool {

queueID := makeQueueID(localChainID, chanID.(string))

qr := QueueRelay{

LocalURL: rc.LocalURL,

RemoteURLs: urls.([]string),

QueueID: queueID,

genesisOrgID: gRelay.queryGenesisOrgID(),

signalChan: make(chan bool, 100),

currentNode: rc.getCurrentNode(queueID),

logger: rc.logger,

}

rc.QueueIDToQueueRelay[qr.QueueID] = &qr

go qr.Start()

qr.signalChan <- true

return true

})

} else {

for \_, v := range rc.QueueIDToQueueRelay {

v.signalChan <- true

}

}

}

func (rc \*RelayController) stopRelay() {

for \_, v := range rc.QueueIDToQueueRelay {

v.signalChan <- false

}

}

func (rc \*RelayController) getLocalChainID() string {

chainID := new(string)

r, e := rc.abciClient.Query().QuerySync(types2.RequestQuery{

Path: keyOfChainID(),

})

if e != nil {

rc.logger.Error("RELAY", "can not get local chainID", e)

return ""

}

e = jsoniter.Unmarshal(r.GetValue(), chainID)

if e != nil {

// 正式链 1.0 和 2.0 的格式不一样

return string(r.GetValue())

}

return \*chainID

}

func (rc \*RelayController) getCurrentNode(queueID string) \*CurrentNodeInfo {

privKey := getCurrentNodePrivKey(rc.config)

priKey := privKey.(crypto.PrivKeyEd25519)

p := "0x" + hex.EncodeToString(priKey[:])

currentNodeInfo := &CurrentNodeInfo{

Address: replaceChainID(getCurrentNodeAddress(rc.config), queueID),

HexPrivKey: p,

Nonce: 0,

}

return currentNodeInfo

}

func (rc \*RelayController) queryGenesisOrgID() string {

r, e := rc.abciClient.Query().QuerySync(types2.RequestQuery{

Path: keyOfGenesisOrgID(),

})

if e != nil {

rc.logger.Error("RELAY", "can not get local genesis org ID", e)

return ""

}

genesisOrgID := new(string)

e = jsoniter.Unmarshal(r.GetValue(), genesisOrgID)

if e != nil {

return ""

}

return \*genesisOrgID

}

// Precommit

type Precommit struct {

Round int `json:"round"` // 投票轮次

Timestamp string `json:"timestamp"`

VoteType byte `json:"type"` // 投票类型

BlockID BlockID `json:"block\_id"`

ValidatorAddress string `json:"validator\_address"` // 验证者节点地址

ValidatorIndex int `json:"validator\_index"` // 验证者节点索引号

Signature SignatureEd25519 `json:"signature"` // 签名数据

}

type BlockID struct {

Hash cmn.HexBytes `json:"hash"`

PartsHeader PartSetHeader `json:"parts"`

}

type PartSetHeader struct {

Total int `json:"total"`

Hash cmn.HexBytes `json:"hash"`

}

// Receipt receipt information

type Receipt struct {

Name string `json:"name"` // 收据名称：标准名称（trnsfer，...) 非标准名称（...）

ContractAddr string `json:"contractAddress"` // 合约地址

Bytes []byte `json:"receiptBytes"`

Hash cmn.HexBytes `json:"receiptHash"`

}

// Info abci msg

type ResultABCIInfo struct {

Response abci.ResponseInfo `json:"response"`

}

type ABCIResponses struct {

DeliverTx []\*abci.ResponseDeliverTx

EndBlock \*abci.ResponseEndBlock

}

type ResultBlock struct {

BlockMeta \*types.BlockMeta `json:"block\_meta"`

Block \*types.Block `json:"block"`

BlockSize int `json:"block\_size"`

}

type ResultABCIQuery struct {

Response abci.ResponseQuery `json:"response"`

}

type ResultBlockResults struct {

Height int64 `json:"height"`

Results \*ABCIResponses `json:"results"`

}

type ResultBroadcastTxCommit struct {

CheckTx abci.ResponseCheckTx `json:"check\_tx,omitempt"`

DeliverTx abci.ResponseDeliverTx `json:"deliver\_tx,omitempt"`

Hash cmn.HexBytes `json:"hash,omitempt"`

Height int64 `json:"height,omitempt"`

}

type MessageIndex struct {

Height int64 `json:"height"`

IbcHash cmn.HexBytes `json:"ibcHash"`

}

type ContractVersionList struct {

Name string `json:"name"` // 合约名称

ContractAddrList []string `json:"contractAddrList"` // 合约地址列表

EffectHeights []int64 `json:"effectHeights"` // 合约生效高度列表

}

// Method method information

type Method struct {

MethodID string `json:"methodId"` //方法ID

Gas int64 `json:"gas"` //方法需要消耗的燃料

ProtoType string `json:"prototype"` //方法原型

}

// Contract contract detail information

type Contract struct {

Address string `json:"address"` //合约地址

Account string `json:"account"` //合约的账户地址

Owner string `json:"owner"` //合约拥有者的账户地址

Name string `json:"name"` //合约名称

Version string `json:"version"` //合约版本

CodeHash cmn.HexBytes `json:"codeHash"` //合约代码的哈希

EffectHeight int64 `json:"effectHeight"` //合约生效的区块高度

LoseHeight int64 `json:"loseHeight"` //合约失效的区块高度

KeyPrefix string `json:"keyPrefix"` //合约在状态数据库中KEY值的前缀

Methods []Method `json:"methods"` //合约对外提供接口的方法列表

Interfaces []Method `json:"interfaces"` //合约提供的跨合约调用的方法列表

Mine []Method `json:"mine"` //合约提供的挖矿方法

IBCs []Method `json:"ibcs,omitempty"` //合约提供的执行跨链业务的方法列表

Token string `json:"token"` //合约代币地址

OrgID string `json:"orgID"` //组织ID

ChainVersion int64 `json:"chainVersion"` //链版本

}

type Header struct {

ChainID string `json:"chain\_id"`

Height int64 `json:"height"`

Time string `json:"time"`

NumTxs int64 `json:"num\_txs"`

LastBlockID BlockID `json:"last\_block\_id"`

TotalTxs int64 `json:"total\_txs"`

LastCommitHash cmn.HexBytes `json:"last\_commit\_hash"`

DataHash cmn.HexBytes `json:"data\_hash"`

ValidatorsHash cmn.HexBytes `json:"validators\_hash"`

ConsensusHash cmn.HexBytes `json:"consensus\_hash"`

LastAppHash cmn.HexBytes `json:"last\_app\_hash"`

LastResultsHash cmn.HexBytes `json:"last\_results\_hash"`

EvidenceHash cmn.HexBytes `json:"evidence\_hash"`

LastFee uint64 `json:"last\_fee,omitempty"`

LastAllocation Allocation `json:"last\_allocation,omitempty"`

ProposerAddress string `json:"proposer\_address,omitempty"`

RewardAddress string `json:"reward\_address,omitempty"`

RandomOfBlock cmn.HexBytes `json:"random\_of\_block,omitempty"`

LastMining \*int64 `json:"last\_mining,omitempty"`

Version \*string `json:"version,omitempty"`

ChainVersion \*int64 `json:"chain\_version,omitempty"`

LastQueueChains \*QueueChain `json:"last\_queue\_chains,omitempty"`

Relayer \*Relayer `json:"relayer,omitempty"`

}

type Relayer struct {

Address string `json:"address"`

StartTime string `json:"start\_time"`

}

type AllocItem struct {

Addr string `json:"addr"`

Fee uint64 `json:"fee"`

}

type Allocation []AllocItem

type QueueChain struct {

QueueBlocks []QueueBlock `json:"ibc\_queue\_blocks"`

}

type QueueBlock struct {

QueueID string `json:"queue\_id"`

QueueHash cmn.HexBytes `json:"queue\_hash"`

LastQueueHash cmn.HexBytes `json:"last\_queue\_hash"`

LastQueueHeight int64 `json:"last\_queue\_height"`

}

type ChainInfo struct {

SideChainName string `json:"sideChainName"` //侧链名称

ChainID string `json:"chainID"` //侧链ID

NodeNames []string `json:"NodeNames"` //节点名称列表

OrgName string `json:"orgName"` //侧链所属组织名称

Owner string `json:"owner"` //侧链的所有者地址

Height int64 `json:"height"` //侧链创世时在主链上的高度

Status string `json:"status"` //侧链状态

GasPriceRatio string `json:"gasPriceRatio"` //燃料价格调整比例

}

func initFilesWithConfig(config \*cfg.Config) error {

// private validator

privValFile := config.PrivValidatorFile()

var pv \*pvm.FilePV

if cmn.FileExists(privValFile) {

pv = pvm.LoadFilePV(privValFile)

logger.Info("Found private validator", "path", privValFile)

} else {

pv = pvm.GenFilePV(privValFile)

pv.Save()

logger.Info("Generated private validator", "path", privValFile)

}

nodeKeyFile := config.NodeKeyFile()

if cmn.FileExists(nodeKeyFile) {

logger.Info("Found node key", "path", nodeKeyFile)

} else {

if \_, err := p2p.LoadOrGenNodeKey(nodeKeyFile); err != nil {

return err

}

logger.Info("Generated node key", "path", nodeKeyFile)

}

// genesis file

genFile := config.GenesisFile()

if cmn.FileExists(genFile) {

logger.Info("Found genesis file", "path", genFile)

} else {

genDoc := types.GenesisDoc{

ChainID: cmn.Fmt("test-chain-%v", cmn.RandStr(6)),

}

genDoc.Validators = []types.GenesisValidator{{

PubKey: pv.GetPubKey(),

Power: 10,

}}

if err := genDoc.SaveAs(genFile); err != nil {

return err

}

logger.Info("Generated genesis file", "path", genFile)

}

return nil

}

var (

genesisPath string

chainID string

byzantium string

proxyApp string

aAddr string

listenPort string

listenPortN = 0

)

func AddInitFlags(cmd \*cobra.Command) {

cmd.Flags().String("chain\_id", chainID, "Specify the chain ID")

cmd.Flags().String("genesis\_path", genesisPath, "Specify the path of genesis files")

cmd.Flags().String("follow", byzantium, "Main nodes to follow, split by comma(only for follower)")

cmd.Flags().String("proxy\_app", proxyApp, "Gichain's ip address(only for follower)")

cmd.Flags().String("listen\_port", listenPort, "p2p listen port(only for follower)")

cmd.Flags().String("a\_address", aAddr, "ReAnnounce listen address(only for follower)")

}

func initFiles(cmd \*cobra.Command, args []string) {

var err error

c := make(chan os.Signal, 1)

signal.Notify(c, os.Interrupt, syscall.SIGHUP, syscall.SIGTERM)

go func() {

for sig := range c {

fmt.Println("TERM Signal received, exiting.... sig:", sig)

os.Exit(0)

}

}()

chainID, err = cmd.Flags().GetString("chain\_id")

if err != nil {

fmt.Printf("init tendermint parse chain\_id err: %s\n", err)

return

}

genesisPath, err = cmd.Flags().GetString("genesis\_path")

if err != nil {

fmt.Printf("init tendermint parse genesis\_path err: %s\n", err)

return

}

byzantium, err = cmd.Flags().GetString("follow")

if err != nil {

fmt.Printf("init tendermint parse follow err: %s\n", err)

return

}

proxyApp, err = cmd.Flags().GetString("proxy\_app")

if err != nil {

fmt.Printf("init tendermint parse proxyApp err: %s\n", err)

}

aAddr, err = cmd.Flags().GetString("a\_address")

if err != nil {

fmt.Printf("init tendermint parse announced address err: %s\n", err)

}

listenPort, err = cmd.Flags().GetString("listen\_port")

if err != nil {

fmt.Printf("init tendermint parse listen port err: %s\n", err)

}

if listenPort != "" {

listenPortN, err = strconv.Atoi(listenPort)

if err != nil {

fmt.Printf("invalid listenPort: %s, err:%v\n", listenPort, err)

os.Exit(1)

}

}

if genesisPath != "" {

if chainID == "" {

fmt.Printf("init tendermint must use flag \"--genesis\_path\" with \"--chain\_id\"\n")

return

}

if byzantium != "" {

fmt.Printf("init tendermint flag \"--genesis\_path\" conflict with \"--follow\"\n")

return

}

genesisPath = filepath.Join(genesisPath, chainID)

} else if byzantium == "" {

fmt.Printf("init tendermint must use flag \"--genesis\_path\" or \"--follow\"\n")

return

} else {

genDoc := getGenDoc(byzantium)

chainID = genDoc.ChainID

genesisDir := filepath.Join(config.RootDir, "genesis")

if \_, err := os.Stat(genesisDir); os.IsNotExist(err) {

if err = os.Mkdir(genesisDir, 0755); err != nil {

fmt.Println("init tendermint make genesis dir error", err)

return

}

}

gzFile := filepath.Join(genesisDir, chainID+".tar.gz")

tarFile := filepath.Join(genesisDir, chainID+".tar")

pkg := getPkg(byzantium)

err = ioutil.WriteFile(gzFile, pkg, 0755)

if err != nil {

fmt.Println("init tendermint write pkg file error", err)

return

}

err = cmn.UnGzip(gzFile, genesisDir)

if err != nil {

fmt.Println("init tendermint unGzip pkg file error", err)

return

}

err = cmn.UnTar(tarFile, genesisDir)

if err != nil {

fmt.Println("init tendermint unTar pkg file error", err)

return

}

genesisPath = filepath.Join(genesisDir, chainID)

}

// copy files to config dir

genPath := config.GenesisFile()

err = types.CopyFile(genesisPath+"/"+chainID+"-genesis.json", genPath)

if err != nil {

fmt.Printf("copy file err: %s\n", err)

return

}

err = types.CopyFile(genesisPath+"/"+chainID+"-genesis.json.sig", genPath[:len(genPath)-5]+".json.sig")

if err != nil {

fmt.Printf("copy file err: %s\n", err)

return

}

err = types.CopyFile(genesisPath+"/"+chainID+"-validators.json", config.ValidatorsFile())

if err != nil {

fmt.Printf("copy file err: %s\n", err)

return

}

// move any .tar.gz file to config directory, they are system contracts. (and maybe garbage :-)

allFiles, err := ioutil.ReadDir(genesisPath)

if err != nil {

fmt.Printf("List Directory err: %s\n", err)

return

}

for \_, file := range allFiles {

if !file.IsDir() &&

(strings.HasSuffix(file.Name(), ".tar.gz") || file.Name() == "genesis-smart-contract-release-version.txt") {

err = types.CopyFile(

filepath.Join(genesisPath, file.Name()),

filepath.Join(config.RootDir, "config", file.Name()),

)

if err != nil {

fmt.Printf("copy file err: %s\n", err)

return

}

}

}

genDoc, err := types.GenesisDocFromFile(config)

if err != nil {

fmt.Printf("tendermint can't parse genesis file, %v\n", err)

return

}

if chainID != genDoc.ChainID {

fmt.Printf("tendermint parsed chainid(%v) is not match path name(%v)\n", genDoc.ChainID, chainID)

return

}

crypto.SetChainId(genDoc.ChainID)

privValFile := config.PrivValidatorFile()

var pv \*pvm.FilePV

if cmn.FileExists(privValFile) {

\_ = pvm.LoadFilePV(privValFile)

logger.Info("Found private validator", "path", privValFile)

} else {

pv = pvm.GenFilePV(privValFile)

pv.Save()

logger.Info("Generated private validator", "path", privValFile)

}

nodeKeyFile := config.NodeKeyFile()

if cmn.FileExists(nodeKeyFile) {

logger.Info("Found node key", "path", nodeKeyFile)

} else {

if \_, err = p2p.LoadOrGenNodeKey(nodeKeyFile); err != nil {

fmt.Printf("init tendermint parse node\_list err: %s\n", err)

return

}

logger.Info("Generated node key", "path", nodeKeyFile)

}

if byzantium == "" {

nodeListFilename := filepath.Join(genesisPath, chainID+"-nodes.json")

ProcessP2P(\*genDoc, nodeListFilename, proxyApp)

} else {

ProcessFollower(byzantium, proxyApp, aAddr, listenPortN)

}

}

// ProcessFollower - processing the follower :-) nonsense

func ProcessFollower(byzantium, proxyApp, aAddr string, listenPortN int) {

persistentPeers := getPersistentPeers(byzantium)

genesisDoc := getGenDoc(byzantium)

conf := cfg.DefaultConfig()

\_ = viper.Unmarshal(conf) // nolint unhandled

configFilePath := filepath.Join(conf.RootDir, "config", "config.toml")

home := os.Getenv("TMHOME")

if strings.HasPrefix(home, "/etc") {

paths := strings.Split(home, "/")

var myHome string

if len(paths) > 2 {

myHome = "/home/" + paths[2]

} else {

myHome = "/home/tmcore"

}

conf.DBPath = myHome + "/data"

conf.LogPath = myHome + "/log"

conf.Mempool.WalPath = myHome + "/data/mempool.wal"

conf.Consensus.WalPath = myHome + "/data/cs.wal/wal"

}

conf.P2P.PersistentPeers = persistentPeers

if proxyApp != "" {

conf.ProxyApp = []string{proxyApp}

}

if aAddr != "" {

conf.P2P.AAddress = aAddr

}

if listenPortN != 0 {

conf.P2P.ListenAddress = fmt.Sprintf("tcp://0.0.0.0:%d", listenPortN)

conf.RPC.ListenAddress = fmt.Sprintf("tcp://0.0.0.0:%d", listenPortN+1)

}

cfg.WriteConfigFile(configFilePath, conf)

validatorsFile := conf.ValidatorsFile()

outByte, err := cdc.MarshalJSONIndent(genesisDoc.Validators, "", " ")

if err != nil {

fmt.Printf("last step,marshal genesisDoc err: %v\n", err)

return

}

\_ = ioutil.WriteFile(validatorsFile, outByte, 0600) // nolint unhandled

}

type nodeInfo struct {

ID string `json:"id"`

AnnouncedAddr string `json:"listen\_addr"`

}

type statusResult struct {

NodeInfo nodeInfo `json:"node\_info"`

}

type statusResponse struct {

Result statusResult `json:"result"`

}

func getPersistentPeers(byzantium string) string {

persistentPeers := ""

voters := strings.Split(byzantium, ",")

for i, v := range voters {

nodeID := getNodeID(v)

if nodeID != "" {

if i != 0 {

persistentPeers += ","

}

persistentPeers += nodeID

}

}

return persistentPeers

}

func getGenDoc(byzantium string) \*types.GenesisDoc {

voters := strings.Split(byzantium, ",")

for \_, v := range voters {

genesisDoc := getGenesis(v)

if genesisDoc != nil {

return genesisDoc

}

}

return nil

}

func getPkg(byzantium string) []byte {

voters := strings.Split(byzantium, ",")

for \_, v := range voters {

pkg := getPkgFromNode(v)

if pkg != nil {

return pkg

}

}

return nil

}

func getNodeID(node string) string {

url := "https://" + node + "/status"

url2 := "http://" + node + "/status"

nodeID := getNodeIDFromURL(url)

if nodeID == "" {

return getNodeIDFromURL(url2)

}

return nodeID

}

func getNodeIDFromURL(url string) string {

req, err := http.NewRequest("GET", url, nil)

if err != nil {

fmt.Printf("get status from %s cause err1: %v\n", url, err)

return ""

}

client := &http.Client{

Timeout: time.Duration(10 \* time.Second),

}

resp, err := client.Do(req)

if err != nil {

fmt.Printf("get status from %s cause err2: %v\n", url, err)

return ""

}

defer func() { \_ = resp.Body.Close() }() // nolint unhandled

// fmt.Println("response Status:", resp.Status)

// fmt.Println("response Headers:", resp.Header)

body, err := ioutil.ReadAll(resp.Body)

if err != nil {

fmt.Printf("get status from %s cause err3: %v\n", url, err)

return ""

}

// fmt.Println("response Body:", string(body))

var nodeStatus statusResponse

err = json.Unmarshal(body, &nodeStatus)

if err != nil {

fmt.Printf("get Status from %s parse err: %v\n", url, err)

return ""

}

if nodeStatus.Result.NodeInfo.ID == "" {

fmt.Printf("got node Id='' from %s\n", url)

return ""

}

if nodeStatus.Result.NodeInfo.AnnouncedAddr == "" {

fmt.Printf("bad listen address:(%v) from %s", nodeStatus.Result.NodeInfo.AnnouncedAddr, url)

return ""

}

return nodeStatus.Result.NodeInfo.ID + "@" + nodeStatus.Result.NodeInfo.AnnouncedAddr

}

func getGenesisFromURL(url string) \*types.GenesisDoc {

req, err := http.NewRequest("GET", url, nil)

if err != nil {

fmt.Printf("get genesis from %s cause err1: %v\n", url, err)

return nil

}

client := &http.Client{

Timeout: 10 \* time.Second,

}

resp, err := client.Do(req)

if err != nil {

fmt.Printf("get genesis from %s cause err2: %v\n", url, err)

return nil

}

defer func() { \_ = resp.Body.Close() }() // nolint unhandled

// fmt.Println("response Status:", resp.Status)

// fmt.Println("response Headers:", resp.Header)

body, err := ioutil.ReadAll(resp.Body)

if err != nil {

fmt.Printf("get genesis from %s cause err3: %v\n", url, err)

return nil

}

type genResult struct {

Genesis types.GenesisDoc `json:"genesis"`

}

type genResponse struct {

Result genResult `json:"result"`

}

// fmt.Println("response Body:", string(body))

var genesisResponse genResponse

err = cdc.UnmarshalJSON(body, &genesisResponse)

if err != nil {

fmt.Printf("get genesis from %s parse err: %v\n", url, err)

return nil

}

genesis := genesisResponse.Result.Genesis

return &genesis

}

func getGenesis(node string) \*types.GenesisDoc {

url := "https://" + node + "/genesis"

url2 := "http://" + node + "/genesis"

genesisDoc := getGenesisFromURL(url)

if genesisDoc != nil {

return genesisDoc

}

return getGenesisFromURL(url2)

}

func getPkgFromNode(node string) []byte {

url := "https://" + node + "/genesis\_pkg"

url2 := "http://" + node + "/genesis\_pkg"

pkg := getPkgFromURL(url)

if pkg != nil {

return pkg

}

return getPkgFromURL(url2)

}

func getPkgFromURL(url string) []byte {

req, err := http.NewRequest("GET", url, nil)

if err != nil {

fmt.Printf("get pkg from %s cause err1: %v\n", url, err)

return nil

}

client := &http.Client{

Timeout: 10 \* time.Second,

}

resp, err := client.Do(req)

if err != nil {

fmt.Printf("get pkg from %s cause err2: %v\n", url, err)

return nil

}

defer func() { \_ = resp.Body.Close() }() // nolint unhandled

// fmt.Println("response Status:", resp.Status)

// fmt.Println("response Headers:", resp.Header)

body, err := ioutil.ReadAll(resp.Body)

if err != nil {

fmt.Printf("get pkg from %s cause err3: %v\n", url, err)

return nil

}

// fmt.Println("response Body:", string(body))

type pkgResult struct {

F json.RawMessage `json:"f"`

}

type pkgResponse struct {

Result pkgResult `json:"result"`

}

var peerResponse pkgResponse

err = cdc.UnmarshalJSON(body, &peerResponse)

if err != nil {

fmt.Printf("get pkg from %s parse err4: %v\n", url, err)

return nil

}

var pkg []byte

err = jsoniter.Unmarshal(peerResponse.Result.F, &pkg)

if err != nil {

fmt.Printf("get pkg from %s unmarshal err5: %v\n", url, err)

return nil

}

return pkg

}

func init() {

registerFlagsRootCmd(RootCmd)

sidechain.ConfigPathFunc = GetConfigFiles

}

func registerFlagsRootCmd(cmd \*cobra.Command) {

cmd.PersistentFlags().String("log\_level", config.LogLevel, "Log level")

// For log customization, to support Log file

cmd.PersistentFlags().String("log\_file", config.LogFile, "Log file")

}

// ParseConfig retrieves the default environment configuration,

// sets up the Tendermint root and ensures that the root exists

func ParseConfig(isInit bool) (\*cfg.Config, error) {

conf := cfg.DefaultConfig()

confStat, err0 := os.Stat(conf.ConfigFilePath())

genStat, err1 := os.Stat(conf.GenesisFile())

if err0 == nil && confStat.Mode().IsRegular() && err1 == nil && genStat.Mode().IsRegular() {

err := viper.Unmarshal(conf)

if err != nil {

return nil, err

}

return conf, nil

}

tmHome := os.Getenv("TMHOME")

conf.SetRoot(tmHome)

confStat, err0 = os.Stat(conf.ConfigFilePath())

genStat, err1 = os.Stat(conf.GenesisFile())

if err0 == nil && confStat.Mode().IsRegular() && err1 == nil && genStat.Mode().IsRegular() {

err := viper.Unmarshal(conf)

if err != nil {

return nil, err

}

return conf, nil

}

pwd, err := os.Getwd()

if err == nil {

conf.SetRoot(pwd)

confStat, err0 = os.Stat(conf.ConfigFilePath())

genStat, err1 = os.Stat(conf.GenesisFile())

if err0 == nil && confStat.Mode().IsRegular() && err1 == nil && genStat.Mode().IsRegular() {

err := viper.Unmarshal(conf)

if err != nil {

return nil, err

}

return conf, nil

}

}

usr, err := user.Current()

if err == nil {

conf.SetRoot(filepath.Join(usr.HomeDir, ".tendermint"))

confStat, err0 = os.Stat(conf.ConfigFilePath())

genStat, err1 = os.Stat(conf.GenesisFile())

if err0 == nil && confStat.Mode().IsRegular() && err1 == nil && genStat.Mode().IsRegular() {

err := viper.Unmarshal(conf)

if err != nil {

return nil, err

}

return conf, nil

}

}

if !isInit {

return nil, errors.New("you must init genesis")

} else {

if tmHome != "" {

conf.SetRoot(tmHome)

} else {

conf.SetRoot(filepath.Join(usr.HomeDir, ".tendermint"))

}

cfg.EnsureRoot(conf.RootDir)

return conf, nil

}

}

// RootCmd is the root command for Tendermint core.

var RootCmd = &cobra.Command{

Use: "tendermint",

Short: "Tendermint Core (BFT Consensus) in Go",

PersistentPreRunE: func(cmd \*cobra.Command, args []string) (err error) {

if cmd.Name() == VersionCmd.Name() {

return nil

}

config, err = ParseConfig(cmd == InitFilesCmd)

if err != nil {

return err

}

if len(config.LogFile) > 0 {

output, err = os.OpenFile(config.LogFile, os.O\_WRONLY|os.O\_CREATE|os.O\_APPEND, 0600)

if err != nil {

return err

}

} else {

output = os.Stdout

}

logger1 := log.NewTMLogger(config.LogDir(), "tmcore")

logger1.SetOutputAsync(true)

logger1.SetWithThreadID(true)

logger1.AllowLevel("debug")

logger = logger1

logger, err = tmflags.ParseLogLevel(config.LogFile, config.LogLevel, logger, cfg.DefaultLogLevel())

if err != nil {

return err

}

if viper.GetBool(cli.TraceFlag) {

logger = log.NewTracingLogger(logger)

}

logger = logger.With("module", "main")

return nil

},

}

func GetConfig() \*cfg.Config {

return config

}

func GetConfigFiles() (string, string, string, string, string) {

return config.GenesisFile(), config.ConfigFilePath(), config.DBDir(), config.ValidatorsFile(), config.PrivValidatorFile()

}

func showNodeID(cmd \*cobra.Command, args []string) error {

genDoc, err := types.GenesisDocFromFile(config)

if err != nil {

logger.Error("tendermint can't parse genesis file", "parse", err)

return err

}

crypto.SetChainId(genDoc.ChainID)

nodeKey, err := p2p.LoadNodeKey(config.NodeKeyFile())

if err != nil {

return err

}

fmt.Println(nodeKey.ID())

return nil

}

func showValidator(cmd \*cobra.Command, args []string) {

genDoc, err := types.GenesisDocFromFile(config)

if err != nil {

logger.Error("tendermint can't parse genesis file", "parse", err)

return

}

crypto.SetChainId(genDoc.ChainID)

privValidator := privval.LoadOrGenFilePV(config.PrivValidatorFile())

pubKey := privValidator.GetPubKey()

pubKeyEd := pubKey.(crypto.PubKeyEd25519)

pubKeyJSONBytes, \_ := cdc.MarshalJSON(pubKey)

fmt.Println(string(pubKeyJSONBytes))

fmt.Println(hex.EncodeToString(pubKeyEd[:]))

}

func init() {

TestnetFilesCmd.Flags().IntVar(&nValidators, "v", 4,

"Number of validators to initialize the testnet with")

TestnetFilesCmd.Flags().IntVar(&nNonValidators, "n", 0,

"Number of non-validators to initialize the testnet with")

TestnetFilesCmd.Flags().StringVar(&outputDir, "o", "./mytestnet",

"Directory to store initialization data for the testnet")

TestnetFilesCmd.Flags().StringVar(&nodeDirPrefix, "node-dir-prefix", "node",

"Prefix the directory name for each node with (node results in node0, node1, ...)")

TestnetFilesCmd.Flags().BoolVar(&populatePersistentPeers, "populate-persistent-peers", true,

"Update config of each node with the list of persistent peers build using either hostname-prefix or starting-ip-address")

TestnetFilesCmd.Flags().StringVar(&hostnamePrefix, "hostname-prefix", "node",

"Hostname prefix (node results in persistent peers list ID0@node0:46656, ID1@node1:46656, ...)")

TestnetFilesCmd.Flags().StringVar(&startingIPAddress, "starting-ip-address", "",

"Starting IP address (192.168.0.1 results in persistent peers list ID0@192.168.0.1:46656, ID1@192.168.0.2:46656, ...)")

TestnetFilesCmd.Flags().IntVar(&p2pPort, "p2p-port", 46656,

"P2P Port")

}

// TestnetFilesCmd allows initialisation of files for a Tendermint testnet.

var TestnetFilesCmd = &cobra.Command{

Use: "testnet",

Short: "Initialize files for a Tendermint testnet",

Long: `testnet will create "v" + "n" number of directories and populate each with

necessary files (private validator, genesis, config, etc.).

Note, strict routability for addresses is turned off in the config file.

Optionally, it will fill in persistent\_peers list in config file using either hostnames or IPs.

Example:

tendermint testnet --v 4 --o ./output --populate-persistent-peers --starting-ip-address 192.168.10.2

`,

RunE: testnetFiles,

}

func testnetFiles(cmd \*cobra.Command, args []string) error {

config := cfg.DefaultConfig()

genVals := make([]types.GenesisValidator, nValidators)

for i := 0; i < nValidators; i++ {

nodeDirName := cmn.Fmt("%s%d", nodeDirPrefix, i)

nodeDir := filepath.Join(outputDir, nodeDirName)

config.SetRoot(nodeDir)

err := os.MkdirAll(filepath.Join(nodeDir, "config"), nodeDirPerm)

if err != nil {

\_ = os.RemoveAll(outputDir)

return err

}

\_ = initFilesWithConfig(config)

pvFile := filepath.Join(nodeDir, config.BaseConfig.PrivValidator)

pv := pvm.LoadFilePV(pvFile)

genVals[i] = types.GenesisValidator{

PubKey: pv.GetPubKey(),

Power: 1,

Name: nodeDirName,

}

}

for i := 0; i < nNonValidators; i++ {

nodeDir := filepath.Join(outputDir, cmn.Fmt("%s%d", nodeDirPrefix, i+nValidators))

config.SetRoot(nodeDir)

err := os.MkdirAll(filepath.Join(nodeDir, "config"), nodeDirPerm)

if err != nil {

\_ = os.RemoveAll(outputDir)

return err

}

\_ = initFilesWithConfig(config)

}

// Generate genesis doc from generated validators

genDoc := &types.GenesisDoc{

GenesisTime: time.Now(),

ChainID: "chain-" + cmn.RandStr(6),

Validators: genVals,

}

// Write genesis file.

for i := 0; i < nValidators+nNonValidators; i++ {

nodeDir := filepath.Join(outputDir, cmn.Fmt("%s%d", nodeDirPrefix, i))

if err := genDoc.SaveAs(filepath.Join(nodeDir, config.BaseConfig.Genesis)); err != nil {

\_ = os.RemoveAll(outputDir)

return err

}

}

if populatePersistentPeers {

err := populatePersistentPeersInConfigAndWriteIt(config)

if err != nil {

\_ = os.RemoveAll(outputDir)

return err

}

}

fmt.Printf("Successfully initialized %v node directories\n", nValidators+nNonValidators)

return nil

}

func hostnameOrIP(i int) string {

if startingIPAddress != "" {

ip := net.ParseIP(startingIPAddress)

ip = ip.To4()

if ip == nil {

fmt.Printf("%v: non ipv4 address\n", startingIPAddress)

os.Exit(1)

}

for j := 0; j < i; j++ {

ip[3]++

}

return ip.String()

}

return fmt.Sprintf("%s%d", hostnamePrefix, i)

}

func populatePersistentPeersInConfigAndWriteIt(config \*cfg.Config) error {

persistentPeers := make([]string, nValidators+nNonValidators)

for i := 0; i < nValidators+nNonValidators; i++ {

nodeDir := filepath.Join(outputDir, cmn.Fmt("%s%d", nodeDirPrefix, i))

config.SetRoot(nodeDir)

nodeKey, err := p2p.LoadNodeKey(config.NodeKeyFile())

if err != nil {

return err

}

persistentPeers[i] = p2p.IDAddressString(nodeKey.ID(), fmt.Sprintf("%s:%d", hostnameOrIP(i), p2pPort))

}

persistentPeersList := strings.Join(persistentPeers, ",")

for i := 0; i < nValidators+nNonValidators; i++ {

nodeDir := filepath.Join(outputDir, cmn.Fmt("%s%d", nodeDirPrefix, i))

config.SetRoot(nodeDir)

config.P2P.PersistentPeers = persistentPeersList

config.P2P.AddrBookStrict = false

// overwrite default config

cfg.WriteConfigFile(filepath.Join(nodeDir, "config", "config.toml"), config)

}

return nil

}

type AppConnMempool interface {

SetResponseCallback(abcicli.Callback)

Error() error

CheckTxAsync(tx []byte) \*abcicli.ReqRes

FlushAsync() \*abcicli.ReqRes

FlushSync() error

}

type AppConnQuery interface {

Error() error

EchoSync(string) (\*types.ResponseEcho, error)

InfoSync(types.RequestInfo) (\*types.ResponseInfo, error)

QuerySync(types.RequestQuery) (\*types.ResponseQuery, error)

QueryExSync(types.RequestQueryEx) (\*types.ResponseQueryEx, error)

// SetOptionSync(key string, value string) (res types.Result)

}

//-----------------------------------------------------------------------------------------

// Implements AppConnConsensus (subset of abcicli.Client)

type appConnConsensus struct {

appConn abcicli.Client

}

func NewAppConnConsensus(appConn abcicli.Client) \*appConnConsensus {

return &appConnConsensus{

appConn: appConn,

}

}

func (app \*appConnConsensus) SetResponseCallback(cb abcicli.Callback) {

app.appConn.SetResponseCallback(cb)

}

func (app \*appConnConsensus) Error() error {

return app.appConn.Error()

}

func (app \*appConnConsensus) InitChainSync(req types.RequestInitChain) (\*types.ResponseInitChain, error) {

return app.appConn.InitChainSync(req)

}

func (app \*appConnConsensus) BeginBlockSync(req types.RequestBeginBlock) (\*types.ResponseBeginBlock, error) {

return app.appConn.BeginBlockSync(req)

}

func (app \*appConnConsensus) DeliverTxAsync(tx []byte) \*abcicli.ReqRes {

return app.appConn.DeliverTxAsync(tx)

}

func (app \*appConnConsensus) EndBlockSync(req types.RequestEndBlock) (\*types.ResponseEndBlock, error) {

return app.appConn.EndBlockSync(req)

}

func (app \*appConnConsensus) CommitSync() (\*types.ResponseCommit, error) {

return app.appConn.CommitSync()

}

func (app \*appConnConsensus) CleanDataSync() (\*types.ResponseCleanData, error) {

return app.appConn.CleanDataSync()

}

//------------------------------------------------

// Implements AppConnMempool (subset of abcicli.Client)

type appConnMempool struct {

appConn abcicli.Client

}

func NewAppConnMempool(appConn abcicli.Client) \*appConnMempool {

return &appConnMempool{

appConn: appConn,

}

}

func (app \*appConnMempool) SetResponseCallback(cb abcicli.Callback) {

app.appConn.SetResponseCallback(cb)

}

func (app \*appConnMempool) Error() error {

return app.appConn.Error()

}

func (app \*appConnMempool) FlushAsync() \*abcicli.ReqRes {

return app.appConn.FlushAsync()

}

func (app \*appConnMempool) FlushSync() error {

return app.appConn.FlushSync()

}

func (app \*appConnMempool) CheckTxAsync(tx []byte) \*abcicli.ReqRes {

return app.appConn.CheckTxAsync(tx)

}

//------------------------------------------------

// Implements AppConnQuery (subset of abcicli.Client)

type appConnQuery struct {

appConn abcicli.Client

}

func NewAppConnQuery(appConn abcicli.Client) \*appConnQuery {

return &appConnQuery{

appConn: appConn,

}

}

func (app \*appConnQuery) Error() error {

return app.appConn.Error()

}

func (app \*appConnQuery) EchoSync(msg string) (\*types.ResponseEcho, error) {

return app.appConn.EchoSync(msg)

}

func (app \*appConnQuery) InfoSync(req types.RequestInfo) (\*types.ResponseInfo, error) {

return app.appConn.InfoSync(req)

}

func (app \*appConnQuery) QuerySync(reqQuery types.RequestQuery) (\*types.ResponseQuery, error) {

return app.appConn.QuerySync(reqQuery)

}

func (app \*appConnQuery) QueryExSync(reqQuery types.RequestQueryEx) (\*types.ResponseQueryEx, error) {

return app.appConn.QueryExSync(reqQuery)

}

type localClientCreator struct {

mtx \*sync.Mutex

app types.Application

}

func NewLocalClientCreator(app types.Application) ClientCreator {

return &localClientCreator{

mtx: new(sync.Mutex),

app: app,

}

}

func (l \*localClientCreator) NewABCIClient() (abcicli.Client, error) {

return abcicli.NewLocalClient(l.mtx, l.app), nil

}

//---------------------------------------------------------------

// remote proxy opens new connections to an external app process

type remoteClientCreator struct {

addr string

transport string

mustConnect bool

}

func NewRemoteClientCreator(addr, transport string, mustConnect bool) ClientCreator {

return &remoteClientCreator{

addr: addr,

transport: transport,

mustConnect: mustConnect,

}

}

func (r \*remoteClientCreator) NewABCIClient() (abcicli.Client, error) {

remoteApp, err := abcicli.NewClient(r.addr, r.transport, r.mustConnect)

if err != nil {

return nil, errors.Wrap(err, "Failed to connect to proxy")

}

return remoteApp, nil

}

//-----------------------------------------------------------------

// default

func DefaultClientCreator(addr, transport, dbDir string) ClientCreator {

switch addr {

case "kvstore":

fallthrough

case "dummy":

return NewLocalClientCreator(kvstore.NewKVStoreApplication())

case "persistent\_kvstore":

fallthrough

case "persistent\_dummy":

return NewLocalClientCreator(kvstore.NewPersistentKVStoreApplication(dbDir))

case "nilapp":

return NewLocalClientCreator(types.NewBaseApplication())

default:

mustConnect := false // loop retrying

return NewRemoteClientCreator(addr, transport, mustConnect)

}

}

// Make all necessary abci connections to the application

func NewMultiAppConn(clientCreator ClientCreator, handshaker Handshaker) \*multiAppConn {

multiAppConn := &multiAppConn{

handshaker: handshaker,

clientCreator: clientCreator,

}

multiAppConn.BaseService = \*cmn.NewBaseService(nil, "multiAppConn", multiAppConn)

return multiAppConn

}

// Returns the mempool connection

func (app \*multiAppConn) Mempool() AppConnMempool {

return app.mempoolConn

}

// Returns the consensus Connection

func (app \*multiAppConn) Consensus() AppConnConsensus {

return app.consensusConn

}

// Returns the query Connection

func (app \*multiAppConn) Query() AppConnQuery {

return app.queryConn

}

func (app \*multiAppConn) OnStart() error {

// query connection

querycli, err := app.clientCreator.NewABCIClient()

if err != nil {

return errors.Wrap(err, "Error creating ABCI client (query connection)")

}

querycli.SetLogger(app.Logger.With("module", "abci-client", "connection", "query"))

if err := querycli.Start(); err != nil {

return errors.Wrap(err, "Error starting ABCI client (query connection)")

}

app.queryConn = NewAppConnQuery(querycli)

// mempool connection

memcli, err := app.clientCreator.NewABCIClient()

if err != nil {

return errors.Wrap(err, "Error creating ABCI client (mempool connection)")

}

memcli.SetLogger(app.Logger.With("module", "abci-client", "connection", "mempool"))

if err := memcli.Start(); err != nil {

return errors.Wrap(err, "Error starting ABCI client (mempool connection)")

}

app.mempoolConn = NewAppConnMempool(memcli)

// consensus connection

concli, err := app.clientCreator.NewABCIClient()

if err != nil {

return errors.Wrap(err, "Error creating ABCI client (consensus connection)")

}

concli.SetLogger(app.Logger.With("module", "abci-client", "connection", "consensus"))

if err := concli.Start(); err != nil {

return errors.Wrap(err, "Error starting ABCI client (consensus connection)")

}

app.consensusConn = NewAppConnConsensus(concli)

// ensure app is synced to the latest state

if app.handshaker != nil {

return app.handshaker.Handshake(app)

}

return nil

}

func ABCIQuery(path string, data cmn.HexBytes, height int64, trusted bool) (\*ctypes.ResultABCIQuery, error) {

if completeStarted == false {

return nil, errors.New("service not ready")

}

resQuery, err := proxyAppQuery.QuerySync(abci.RequestQuery{

Path: path,

Data: data,

Height: height,

Prove: !trusted,

})

if err != nil {

return nil, err

}

logger.Trace("ABCIQuery", "path", path, "data", data, "result", resQuery)

return &ctypes.ResultABCIQuery{\*resQuery}, nil

}

func ABCIQueryEx(path string) (\*ctypes.ResultABCIQueryEx, error) {

if completeStarted == false {

return nil, errors.New("service not ready")

}

resQuery, err := proxyAppQuery.QueryExSync(abci.RequestQueryEx{

Path: path,

})

if err != nil {

return nil, err

}

logger.Trace("ABCIQueryEx", "path", path, "result", resQuery)

return &ctypes.ResultABCIQueryEx{\*resQuery}, nil

}

func parseConfig() { // 相当不 DRY， 避免循环引用

if cfg == nil {

cfg = config.DefaultConfig()

tmPath := os.Getenv("TMHOME")

if tmPath == "" {

home := os.Getenv("HOME")

if home != "" {

tmPath = filepath.Join(home, config.DefaultTendermintDir)

}

}

if tmPath == "" {

tmPath = "/" + config.DefaultTendermintDir

}

cfg.SetRoot(tmPath)

}

}

func GetGenesisPkg() (\*core\_types.ResultConfFile, error) {

if completeStarted == false {

return nil, errors.New("service not ready")

}

lock.Lock()

defer lock.Unlock()

parseConfig()

genesisDir := path.Join(cfg.RootDir, "genesis")

chainDir := path.Join(genesisDir, genDoc.ChainID)

targetFile := chainDir + ".tar.gz"

if !fileutil.Exist(targetFile) {

err := common.TarIt(chainDir, genesisDir)

if err != nil {

return nil, err

}

err = common.GzipIt(chainDir+".tar", genesisDir)

if err != nil {

return nil, err

}

}

byt, err := ioutil.ReadFile(targetFile)

if err != nil {

return nil, err

}

jsonBlob, err := jsoniter.Marshal(byt)

if err != nil {

return nil, err

}

return &core\_types.ResultConfFile{F: jsonBlob}, nil

}

func UnsafeFlushMempool() (\*ctypes.ResultUnsafeFlushMempool, error) {

mempool.Flush()

return &ctypes.ResultUnsafeFlushMempool{}, nil

}

var profFile \*os.File

func UnsafeStartCPUProfiler(filename string) (\*ctypes.ResultUnsafeProfile, error) {

var err error

profFile, err = os.Create(filename)

if err != nil {

return nil, err

}

err = pprof.StartCPUProfile(profFile)

if err != nil {

return nil, err

}

return &ctypes.ResultUnsafeProfile{}, nil

}

func UnsafeStopCPUProfiler() (\*ctypes.ResultUnsafeProfile, error) {

pprof.StopCPUProfile()

if err := profFile.Close(); err != nil {

return nil, err

}

return &ctypes.ResultUnsafeProfile{}, nil

}

func UnsafeWriteHeapProfile(filename string) (\*ctypes.ResultUnsafeProfile, error) {

memProfFile, err := os.Create(filename)

if err != nil {

return nil, err

}

if err := pprof.WriteHeapProfile(memProfFile); err != nil {

return nil, err

}

if err := memProfFile.Close(); err != nil {

return nil, err

}

return &ctypes.ResultUnsafeProfile{}, nil

}

// NewSideChain new SideChain instance

func NewSideChain(genesisInfo \*abci.SideChainGenesis) \*SideChain {

sc := &SideChain{

GenesisInfo: genesisInfo,

TempPath: filepath.Join(configPath(), ".sidechaintemp"),

}

return sc

}

// Genesis side chain genesis, copy prepare file to desitination.

func (sc \*SideChain) Genesis() error {

// copy all side chain genesis file

desContractPath := configPath()

err := filepath.Walk(sc.TempPath, func(path string, info os.FileInfo, err error) error {

if !info.IsDir() && info.Name() != "needgenesis" {

\_, err := fs.CopyFile(path, filepath.Join(desContractPath, info.Name()))

if err != nil {

return err

}

}

return nil

})

if err != nil {

return err

}

// remove tmcore db

dbDir := dbDir()

if err := os.RemoveAll(dbDir); err != nil {

return err

}

// remove addrbook.json

addrBook := filepath.Join(configPath(), "addrbook.json")

if err := os.RemoveAll(addrBook); err != nil {

return err

}

return os.RemoveAll(sc.TempPath)

}

// NeedSCGenesis return true if has side chain need genesis

func (sc \*SideChain) NeedSCGenesis() bool {

exist, err := fs.PathExists(filepath.Join(sc.TempPath, "needgenesis"))

if err != nil {

panic(err)

}

return exist

}

// PrepareSCGenesis generate temp files for side chain genesis.

func (sc \*SideChain) PrepareSCGenesis() error {

var err error

if err = os.MkdirAll(sc.TempPath, 0750); err != nil {

panic(err)

}

if err = sc.genContratTarGZ(); err != nil {

return err

}

if err = sc.genGenesisJson(); err != nil {

return err

}

if err = sc.genValidatorJson(); err != nil {

return err

}

if err = sc.genPrivValidatorJson(); err != nil {

return err

}

if err = sc.genConfigToml(); err != nil {

return err

}

if err = sc.delForksFiles(); err != nil {

return err

}

if \_, err = os.Create(filepath.Join(sc.TempPath, "needgenesis")); err != nil {

return err

}

return nil

}

// CopyGenesisFiles copy config files to genesis dir

func (sc \*SideChain) CopyGenesisFiles() error {

tmPath := filepath.Dir(configPath())

genesisPath := filepath.Join(tmPath, "genesis")

chainID, err := sc.getChainID()

if err != nil {

return err

}

genesisPath = filepath.Join(genesisPath, chainID)

if err := os.MkdirAll(genesisPath, 0750); err != nil {

return err

}

err = fs.CopyDir(configPath(), genesisPath, "",

"node\_key.json|priv\_validator.json|addrbook.json|config.toml")

if err != nil {

return err

}

err = filepath.Walk(genesisPath, func(path string, info os.FileInfo, err error) error {

if !info.IsDir() &&

info.Name() == "genesis.json" || info.Name() == "genesis.json.sig" || info.Name() == "validators.json" {

newPath := filepath.Join(genesisPath, chainID+"-"+filepath.Base(path))

return os.Rename(path, newPath)

}

return nil

})

return err

}

func (sc \*SideChain) getChainID() (string, error) {

b, err := ioutil.ReadFile(genesisFile())

if err != nil {

return "", err

}

type genesisInfo struct {

ChainID string `json:"chain\_id"`

}

g := new(genesisInfo)

err = jsoniter.Unmarshal(b, g)

if err != nil {

return "", err

}

return g.ChainID, nil

}

func (sc \*SideChain) delForksFiles() error {

currentPath, err := os.Executable()

if err != nil {

return err

}

currentDir := path.Dir(currentPath)

if err = os.RemoveAll(filepath.Join(currentDir, "tendermint-forks.json")); err != nil {

return err

}

if err = os.RemoveAll(filepath.Join(currentDir, "tendermint-forks.json.sig")); err != nil {

return err

}

return nil

}

func (sc \*SideChain) genContratTarGZ() error {

for \_, v := range sc.GenesisInfo.ContractData {

fileName := v.Name + "-" + v.Version + ".tar.gz"

fi, err := os.Create(filepath.Join(sc.TempPath, fileName))

if err != nil {

return err

}

if \_, err = fi.Write(v.CodeData); err != nil {

return err

}

\_ = fi.Close()

}

return nil

}

func (sc \*SideChain) genGenesisJson() error {

gensisFileName := filepath.Base(genesisFile())

genesisPath := filepath.Join(sc.TempPath, gensisFileName)

fi, err := os.Create(genesisPath)

if err != nil {

return err

}

defer func() {

\_ = fi.Close()

}()

\_, err = fi.WriteString(sc.GenesisInfo.GenesisInfo)

if err != nil {

return err

}

genesisBlob, e := ioutil.ReadFile(genesisPath)

if e != nil {

return err

}

// 对 genesis.json 签名并生成 genesis.json.sig

p := privValidatorFile()

pv := pvm.LoadFilePV(p)

sign := pv.PrivKey.Sign(genesisBlob)

type Signature struct {

PubKey string `json:"pubkey"`

Signature string `json:"signature"`

}

pk := pv.PubKey.(crypto.PubKeyEd25519)

sn := sign.(crypto.SignatureEd25519)

signature := Signature{

PubKey: hex.EncodeToString(pk[:]),

Signature: hex.EncodeToString(sn[:]),

}

signByte, err := jsoniter.Marshal(signature)

if err != nil {

return err

}

fi, err = os.Create(genesisPath + ".sig")

if err != nil {

return err

}

defer func() {

\_ = fi.Close()

}()

\_, err = fi.WriteString(string(signByte))

if err != nil {

return err

}

return nil

}

func (sc \*SideChain) genValidatorJson() error {

if len(sc.GenesisInfo.Validators) == 0 {

return errors.New("invalid side chain validator")

}

v := sc.GenesisInfo.Validators[0]

gv := types.GenesisValidator{

PubKey: crypto.PubKeyEd25519FromBytes(v.PubKey),

RewardAddr: v.RewardAddr,

Power: int64(v.Power),

Name: v.Name,

}

result := make([]types.GenesisValidator, 0, 1)

result = append(result, gv)

outByte, err := cdc.MarshalJSONIndent(result, "", " ")

if err != nil {

return err

}

p := filepath.Join(sc.TempPath, filepath.Base(validatorsFile()))

err = ioutil.WriteFile(p, outByte, 0600)

if err != nil {

return err

}

return nil

}

func (sc \*SideChain) genPrivValidatorJson() error {

privValidatorFile := privValidatorFile()

tempPrivValJson := filepath.Join(sc.TempPath, filepath.Base(privValidatorFile))

\_, err := fs.CopyFile(privValidatorFile, tempPrivValJson)

if err != nil {

return err

}

pv := pvm.LoadFilePV(tempPrivValJson)

pv.LastHeight = 0

pv.LastRound = 0

pv.LastStep = 0

pv.Address = pv.GetPubKey().Address(sc.GenesisInfo.SideChainID)

pv.LastSignature = nil

pv.LastSignBytes = nil

pv.Save()

return nil

}

func (sc \*SideChain) genConfigToml() error {

\_, configFile, \_, \_, \_ := ConfigPathFunc()

tempConfig := filepath.Join(sc.TempPath, filepath.Base(configFile))

configContent, err := ioutil.ReadFile(configFile)

if err != nil {

return err

}

tempFile, err := os.Create(tempConfig)

if err != nil {

return err

}

defer tempFile.Close()

configSplit := strings.Split(string(configContent), "\n")

for \_, line := range configSplit {

if strings.HasPrefix(line, "persistent\_peers") {

if strings.HasSuffix(line, "\r") {

line = `persistent\_peers = ""\r`

} else {

line = `persistent\_peers = ""`

}

}

line += "\n"

\_, err = tempFile.WriteString(line)

if err != nil {

return err

}

}

return nil

}

// ContainsCurrentNode if genesisInfoList contains current node,

// return genesis info and true, or else return nil and false

func ContainsCurrentNode(genesisInfoList []\*abci.SideChainGenesis) (genesisInfo \*abci.SideChainGenesis, ok bool) {

privValidatorFile := privValidatorFile()

currentNodePubKey := pvm.LoadFilePV(privValidatorFile).GetPubKey()

for \_, info := range genesisInfoList {

for \_, v := range info.Validators {

if currentNodePubKey.Equals(crypto.PubKeyEd25519FromBytes(v.PubKey)) {

genesisInfo = info

ok = true

return

}

}

}

return

}