// 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 abciQuery(url, path string) (resultQuery \*ResultABCIQuery, err error) {

resultQuery = new(ResultABCIQuery)

client := getClient(url)

\_, err = client.Call("abci\_query", map[string]interface{}{"path": path}, resultQuery)

return

}

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 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 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 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

}

// 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) 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) 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

}

// 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

}

//Ibc This is struct of contract

//@:contract:ibc

//@:version:2.1

//@:organization:orgJgaGConUyK81zibntUBjQ33PKctpk1K1G

//@:author:5e8339cb1a5cce65602fd4f57e115905348f7e83gie38dd77694dbe1f8903c9

type Ibc struct {

sdk sdk.ISmartContract

}

//InitChain Constructor of this Ibc

//@:constructor

func (i \*Ibc) InitChain() {

}

//@:public:ibc:gas[1500]

func (i \*Ibc) Register(toChainID string) {

sdk.Require(len(toChainID) > 0,

types.ErrInvalidParameter, "Invalid toChainID")

localChainID := i.sdk.Helper().GenesisHelper().ChainID()

sdk.Require(toChainID != localChainID,

types.ErrInvalidParameter, "")

queueID := ""

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

// side chain

mainChainID := i.sdk.Helper().BlockChainHelper().GetMainChainID()

if toChainID != mainChainID {

sdk.Require(i.isValidSideChainID(toChainID),

types.ErrInvalidParameter, "Invalid toChainID")

}

queueID = i.makeQueueID(localChainID, mainChainID)

} else {

// main chain

chainInfo := i.\_chainInfo(toChainID)

sdk.Require(chainInfo.Status == "ready", types.ErrInvalidParameter, "chain status must be ready")

queueID = i.makeQueueID(localChainID, toChainID)

}

origins := i.sdk.Message().Origins()

sdk.Require(len(origins) == 2, types.ErrInvalidParameter, "invalid roigins")

invokeContract := i.sdk.Helper().ContractHelper().ContractOfAddress(origins[1])

packet := ibc.Packet{

FromChainID: localChainID,

ToChainID: toChainID,

QueueID: queueID,

Seq: i.\_sequence(queueID) + 1,

OrgID: invokeContract.OrgID(),

ContractName: invokeContract.Name(),

IbcHash: i.sdk.Helper().IBCHelper().IbcHash(toChainID),

Type: ibc.TccTxType,

State: ibc.State{

Status: ibc.NoAckWanted,

Tag: ibc.RecastPending,

},

Receipts: i.sdk.Message().InputReceipts(),

}

i.savePacket(toChainID, nil, &packet)

i.sdk.Helper().ReceiptHelper().Emit(packet)

}

//@:public:ibc:gas[1000]

func (i \*Ibc) Notify(toChainIDs []string) {

origins := i.sdk.Message().Origins()

sdk.Require(len(origins) == 2,

types.ErrInvalidParameter, "invalid origins")

newChainIDs := i.checkToChainIDs(toChainIDs)

sdk.Require(len(newChainIDs) > 0,

types.ErrInvalidParameter, "toChainIDs can not be empty")

i.notify(newChainIDs)

}

//@:public:ibc:gas[50000]

func (i \*Ibc) Broadcast() {

origins := i.sdk.Message().Origins()

sdk.Require(len(origins) == 2,

types.ErrInvalidParameter, "invalid origins")

toChainIDs := i.filterInvalidSideChain()

if len(toChainIDs) == 0 {

return

}

i.notify(toChainIDs)

}

//@:public:method:gas[0]

func (i \*Ibc) Input(pktsProofs []ibc.PktsProof) {

validPackets := i.checkPktsProof(pktsProofs)

i.input(validPackets)

}

func (i \*Ibc) notify(toChainIDs []string) {

origins := i.sdk.Message().Origins()

originContract := i.sdk.Helper().ContractHelper().ContractOfAddress(origins[1])

localChainID := i.sdk.Helper().GenesisHelper().ChainID()

packet := ibc.Packet{

FromChainID: localChainID,

OrgID: originContract.OrgID(),

ContractName: originContract.Name(),

Type: ibc.NotifyType,

Receipts: i.sdk.Message().InputReceipts(),

}

forx.Range(toChainIDs, func(index int, toChainID string) bool {

packet.ToChainID = toChainID

packet.QueueID = i.makeQueueID(localChainID, toChainID)

packet.Seq = i.\_sequence(packet.QueueID) + 1

i.\_setSequence(packet.QueueID, packet.Seq)

packet.IbcHash = i.sdk.Helper().IBCHelper().IbcHash(toChainID)

packet.State = ibc.State{

Status: ibc.NoAck,

Tag: ibc.NotifyPending,

}

i.\_setState(packet.IbcHash, &packet.State)

packetsOfIbcHash := i.\_packets(packet.IbcHash)

packetsOfIbcHash = append(packetsOfIbcHash, packet)

i.\_setPackets(packet.IbcHash, packetsOfIbcHash)

i.\_setQueueIndex(packet.QueueID, packet.Seq, MessageIndex{

Height: i.sdk.Block().Height(),

IbcHash: packet.IbcHash,

})

i.sdk.Helper().ReceiptHelper().Emit(packet)

return true

})

}

func (i \*Ibc) checkToChainIDs(toChainIDs []string) []string {

localChainID := i.sdk.Block().ChainID()

newToChainIDs := make([]string, 0)

mainChainID := i.sdk.Helper().BlockChainHelper().GetMainChainID()

forx.Range(toChainIDs, func(index int, toChainID string) bool {

if len(toChainID) == 0 || toChainID == localChainID {

return forx.Continue

}

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

if toChainID == mainChainID &&

!inSlice(newToChainIDs, toChainID) {

newToChainIDs = append(newToChainIDs, toChainID)

return forx.Continue

}

} else {

chainInfo := i.\_chainInfo(toChainID)

if chainInfo.Status != "" && chainInfo.Status != "disabled" &&

!inSlice(newToChainIDs, toChainID) {

newToChainIDs = append(newToChainIDs, toChainID)

return forx.Continue

}

}

return forx.Continue

})

return newToChainIDs

}

func inSlice(slice []string, item string) bool {

exist := false

forx.Range(slice, func(index int, s string) bool {

if item == s {

exist = true

return forx.Break

}

return forx.Continue

})

return exist

}

func (i \*Ibc) filterInvalidSideChain() (validChainIDs []string) {

allSideChainIDs := i.\_sideChainIDs()

validChainIDs = make([]string, 0, len(allSideChainIDs))

forx.Range(allSideChainIDs, func(index int, chainID string) bool {

chainInfo := i.\_chainInfo(chainID)

if chainInfo.Status != "disabled" {

validChainIDs = append(validChainIDs, chainID)

}

return true

})

return validChainIDs

}

func (i \*Ibc) checkSideChain(packet \*ibc.Packet) types.Error {

err := types.Error{

ErrorCode: types.CodeOK,

}

if i.\_chainInfo(packet.FromChainID).Status == "disabled" {

err.ErrorCode = types.ErrInvalidParameter

err.ErrorDesc = "From chain status can not be disabled"

return err

}

if packet.ToChainID != i.sdk.Block().ChainID() && i.\_chainInfo(packet.ToChainID).Status != "ready" {

err.ErrorCode = types.ErrInvalidParameter

err.ErrorDesc = "To chain status must be ready"

return err

}

return err

}

func (i \*Ibc) processNotifyResult(packet \*ibc.Packet, outReceipts []types.KVPair, notifyErr types.Error) {

newPacket := i.newPacket(packet)

if notifyErr.ErrorCode != types.CodeOK {

newPacket.State.Tag = ibc.NotifyFailure

newPacket.State.Log = notifyErr.ErrorDesc

} else {

newPacket.State.Tag = ibc.NotifySuccess

}

newPacket.State.Status = ibc.NoAck

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

i.\_setState(newPacket.IbcHash, &newPacket.State)

packetsOfIbcHash := i.\_packets(newPacket.IbcHash)

packetsOfIbcHash = append(packetsOfIbcHash, \*newPacket)

i.\_setPackets(newPacket.IbcHash, packetsOfIbcHash)

}

func (i \*Ibc) processCancelRecastResult(packet \*ibc.Packet, outReceipts []types.KVPair, cancelHubErr types.Error) {

newPacket := i.newPacket(packet)

if cancelHubErr.ErrorCode != types.CodeOK {

newPacket.State.Log = cancelHubErr.ErrorDesc

}

newPacket.State.Status = ibc.NoAck

newPacket.State.Tag = packet.State.Tag

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

newPacket.QueueID = i.makeQueueID(i.sdk.Helper().GenesisHelper().ChainID(), newPacket.FromChainID)

newPacket.Seq = i.\_sequence(newPacket.QueueID) + 1

i.savePacket(newPacket.FromChainID, packet, newPacket)

i.sdk.Helper().ReceiptHelper().Emit(newPacket)

}

func (i \*Ibc) processCancelResult(packet \*ibc.Packet, outReceipts []types.KVPair, cancelErr types.Error) {

newPacket := i.newPacket(packet)

if cancelErr.ErrorCode != types.CodeOK {

newPacket.State.Log = cancelErr.ErrorDesc

}

newPacket.State.Status = ibc.NoAck

newPacket.State.Tag = ibc.Canceled

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

packetsOfIbcHash := i.\_packets(newPacket.IbcHash)

packetsOfIbcHash = append(packetsOfIbcHash, \*packet)

packetsOfIbcHash = append(packetsOfIbcHash, \*newPacket)

i.\_setPackets(newPacket.IbcHash, packetsOfIbcHash)

i.\_setState(newPacket.IbcHash, &newPacket.State)

final := ibc.Final{

IBCHash: newPacket.IbcHash,

State: ibc.State{

Status: ibc.NoAck,

Tag: ibc.Canceled,

Log: cancelErr.ErrorDesc,

},

}

i.sdk.Helper().ReceiptHelper().Emit(final)

}

func (i \*Ibc) processConfirmRecastResult(packet \*ibc.Packet, outReceipts []types.KVPair, confirmHubErr types.Error) {

newPacket := i.newPacket(packet)

if confirmHubErr.ErrorCode != types.CodeOK {

newPacket.State.Log = confirmHubErr.ErrorDesc

}

newPacket.State.Status = ibc.NoAck

newPacket.State.Tag = packet.State.Tag

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

newPacket.QueueID = i.makeQueueID(i.sdk.Helper().GenesisHelper().ChainID(), packet.FromChainID)

newPacket.Seq = i.\_sequence(newPacket.QueueID) + 1

i.savePacket(newPacket.FromChainID, packet, newPacket)

i.sdk.Helper().ReceiptHelper().Emit(newPacket)

}

func (i \*Ibc) processConfirmResult(packet \*ibc.Packet, outReceipts []types.KVPair, confirmErr types.Error) {

newPacket := i.newPacket(packet)

if confirmErr.ErrorCode != types.CodeOK {

newPacket.State.Log = confirmErr.ErrorDesc

}

newPacket.State.Tag = ibc.Confirmed

newPacket.State.Status = ibc.NoAck

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

packetsOfIbcHash := i.\_packets(newPacket.IbcHash)

packetsOfIbcHash = append(packetsOfIbcHash, \*packet)

packetsOfIbcHash = append(packetsOfIbcHash, \*newPacket)

i.\_setPackets(newPacket.IbcHash, packetsOfIbcHash)

i.\_setState(newPacket.IbcHash, &newPacket.State)

final := ibc.Final{

IBCHash: packet.IbcHash,

State: ibc.State{

Status: ibc.NoAck,

Tag: ibc.Confirmed,

Log: confirmErr.ErrorDesc,

},

}

i.sdk.Helper().ReceiptHelper().Emit(final)

}

func (i \*Ibc) isRelay(packet \*ibc.Packet) bool {

return !i.sdk.Helper().BlockChainHelper().IsSideChain() &&

i.sdk.Helper().GenesisHelper().ChainID() != packet.FromChainID &&

i.sdk.Helper().GenesisHelper().ChainID() != packet.ToChainID

}

func (i \*Ibc) processTryRecastResult(packet \*ibc.Packet, recastOk bool, outReceipts []types.KVPair, tryHubErr types.Error) {

newPacket := i.newPacket(packet)

toChainID := ""

if tryHubErr.ErrorCode != types.CodeOK {

toChainID = packet.FromChainID

newPacket.State.Status = ibc.NoAck

newPacket.State.Tag = ibc.CancelPending

newPacket.State.Log = tryHubErr.ErrorDesc

} else if !recastOk {

toChainID = packet.FromChainID

newPacket.State.Status = ibc.NoAck

newPacket.State.Tag = ibc.CancelPending

} else {

toChainID = packet.ToChainID

newPacket.State.Status = ibc.NoAckWanted

newPacket.State.Tag = ibc.RecastPending

}

newPacket.QueueID = i.makeQueueID(i.sdk.Helper().GenesisHelper().ChainID(), toChainID)

newPacket.Receipts = packet.Receipts

seq := i.\_sequence(newPacket.QueueID) + 1

newPacket.Seq = seq

i.savePacket(toChainID, packet, newPacket)

i.sdk.Helper().ReceiptHelper().Emit(newPacket)

}

func (i \*Ibc) processRecastResult(packet \*ibc.Packet, recastOk bool, outReceipts []types.KVPair, recastErr types.Error) {

newPacket := i.newPacket(packet)

if recastErr.ErrorCode != types.CodeOK {

newPacket.State.Tag = ibc.CancelPending

newPacket.State.Log = recastErr.ErrorDesc

} else if !recastOk {

newPacket.State.Tag = ibc.CancelPending

} else {

newPacket.State.Tag = ibc.ConfirmPending

}

if len(outReceipts) > 0 {

newPacket.Receipts = append(packet.Receipts, outReceipts[1:]...)

} else {

newPacket.Receipts = packet.Receipts

}

toChainID := ""

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

toChainID = i.sdk.Helper().BlockChainHelper().GetMainChainID()

} else {

toChainID = packet.FromChainID

}

newPacket.State.Status = ibc.NoAck

newPacket.QueueID = i.makeQueueID(i.sdk.Helper().GenesisHelper().ChainID(), toChainID)

seq := i.\_sequence(i.makeQueueID(i.sdk.Helper().GenesisHelper().ChainID(), toChainID)) + 1

newPacket.Seq = seq

i.savePacket(toChainID, packet, newPacket)

i.sdk.Helper().ReceiptHelper().Emit(\*newPacket)

}

func (i \*Ibc) savePacket(toChainID string, oldPacket \*ibc.Packet, newPacket \*ibc.Packet) {

packetsOfIbcHash := i.\_packets(newPacket.IbcHash)

if oldPacket != nil {

packetsOfIbcHash = append(packetsOfIbcHash, \*oldPacket)

}

if newPacket != nil {

packetsOfIbcHash = append(packetsOfIbcHash, \*newPacket)

}

i.\_setPackets(newPacket.IbcHash, packetsOfIbcHash)

i.\_setState(newPacket.IbcHash, &newPacket.State)

i.\_setSequence(newPacket.QueueID, newPacket.Seq)

i.\_setQueueIndex(newPacket.QueueID, newPacket.Seq, MessageIndex{

Height: i.sdk.Block().Height(),

IbcHash: newPacket.IbcHash,

})

}

func (i \*Ibc) isProcessedPacket(packet \*ibc.Packet) bool {

currentSeq := i.\_sequence(packet.QueueID)

if packet.Seq > currentSeq {

return false

} else {

packets := i.\_packets(packet.IbcHash)

sdk.Require(len(packets) > 0,

types.ErrInvalidParameter, "invalid packets seq")

return true

}

}

func (i \*Ibc) makeQueueID(fromChainID, toChainID string) string {

var sb strings.Builder

sb.WriteString(fromChainID)

sb.WriteString("->")

sb.WriteString(toChainID)

return sb.String()

}

func (i \*Ibc) splitQueue(queueID string) (fromChainID, toChainID string, err error) {

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

if len(result) != 2 {

return "", "", errors.New("Invalid queue: " + queueID)

}

return result[0], result[1], nil

}

func (i \*Ibc) newPacket(oldPacket \*ibc.Packet) \*ibc.Packet {

newPacket := ibc.Packet{

FromChainID: oldPacket.FromChainID,

ToChainID: oldPacket.ToChainID,

OrgID: oldPacket.OrgID,

ContractName: oldPacket.ContractName,

IbcHash: oldPacket.IbcHash,

Type: oldPacket.Type,

}

return &newPacket

}

func (i \*Ibc) hasGovernanceNotify(packets []ibc.Packet) bool {

result := false

forx.Range(packets, func(index int, packet ibc.Packet) bool {

if packet.ContractName == "governance" {

result = true

return forx.Break

}

return forx.Continue

})

return result

}

// checkPktsProof check packets and proof data

func (i \*Ibc) checkPktsProof(pktsProofs []ibc.PktsProof) []ibc.Packet {

sdk.Require(len(pktsProofs) > 0 && len(pktsProofs) <= 10,

types.ErrInvalidParameter, "invlaid pktsProofs length")

validPackets := make([]ibc.Packet, 0)

var lastSeq uint64

forx.Range(pktsProofs, func(index int, pktsProof ibc.PktsProof) bool {

if len(validPackets) != 0 {

lastSeq = validPackets[len(validPackets)-1].Seq

}

newPackets := i.checkPackets(pktsProof.Packets, lastSeq)

if len(newPackets) == 0 {

return forx.Continue

}

sdk.Require(i.checkPrecommits(&pktsProof.Header, pktsProof.Precommits, i.hasGovernanceNotify(newPackets)),

types.ErrInvalidParameter, "checkPrecommits failed")

i.checkQueueHash(pktsProof.Header.LastQueueChains, newPackets)

validPackets = append(validPackets, newPackets...)

return forx.Continue

})

sdk.Require(len(validPackets) > 0,

types.ErrInvalidParameter, "no valid packet")

return validPackets

}

// checkPacket check packets data

func (i \*Ibc) checkPackets(packets []ibc.Packet, lastSeq uint64) []ibc.Packet {

newPackets := make([]ibc.Packet, 0)

var localSeq uint64

forx.Range(packets, func(index int, packet ibc.Packet) bool {

i.checkQueueID(packet.QueueID)

if localSeq == 0 {

localSeq = i.\_sequence(packet.QueueID)

}

if packet.Seq <= localSeq {

return forx.Continue // 已处理过的packet，跳过

}

if lastSeq == 0 {

lastSeq = localSeq

}

sdk.Require(packet.Seq == lastSeq+1,

types.ErrInvalidParameter, fmt.Sprintf("invalid packet: expected seq %d, obtain %d", localSeq+1, packet.Seq))

i.checkTypeState(packet.Type, packet.State)

i.checkFromToChainID(packet.State.Tag, packet.FromChainID, packet.ToChainID)

sdk.Require(len(packet.IbcHash) == 32,

types.ErrInvalidParameter, "invalid ibcHash")

newPackets = append(newPackets, packet)

if len(newPackets) == 1 {

lastSeq = newPackets[0].Seq

} else {

lastSeq += 1

}

return forx.Continue

})

return newPackets

}

// checkTypeState check packet's type and state, it's type must be TccTxType or NotifyType;

// if type is TccTxType then state's tag must be RecastPending/ConfirmPending/CancelPending;

// if type is NotifyType then state's tag must be NotifyPending.

func (i \*Ibc) checkTypeState(packetType string, state ibc.State) {

sdk.Require(state.Status == ibc.NoAck || state.Status == ibc.NoAckWanted,

types.ErrInvalidParameter, "invalid packet's state status")

switch packetType {

case ibc.TccTxType:

sdk.Require(

state.Tag == ibc.RecastPending ||

state.Tag == ibc.ConfirmPending ||

state.Tag == ibc.CancelPending,

types.ErrInvalidParameter, "invalid packet's state tag")

case ibc.NotifyType:

sdk.Require(state.Tag == ibc.NotifyPending,

types.ErrInvalidParameter, "invalid packet's state tag")

default:

sdk.Require(false,

types.ErrInvalidParameter, "invalid packet's type")

}

}

// checkQueueID check queueID, it must split by `->`, and must be match next rule

// 1, first chainID must not equal second chainID,

// 2, in side chain, first chainID must be mainChainID, in main chain, first chainID must be valid sideChainID

// 3, second chainID must be localChainID,

func (i \*Ibc) checkQueueID(queueID string) {

errMsg := "invalid queueID"

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

sdk.Require(len(queueIDSplit) == 2,

types.ErrInvalidParameter, errMsg)

localChainID := i.sdk.Block().ChainID()

queueFromChainID := queueIDSplit[0]

queueToChainID := queueIDSplit[1]

sdk.Require(queueToChainID == localChainID,

types.ErrInvalidParameter, errMsg)

sdk.Require(queueFromChainID != queueToChainID,

types.ErrInvalidParameter, errMsg)

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

sdk.Require(i.isMainChainID(queueFromChainID),

types.ErrInvalidParameter, errMsg)

} else {

sdk.Require(i.isValidSideChainID(queueFromChainID),

types.ErrInvalidParameter, errMsg)

}

}

// checkFromToChainID check fromChainID and toChainID, fromChainID cannot equal toChainID,

// and they must match different rule with different tag value

func (i \*Ibc) checkFromToChainID(tag string, fromChainID, toChainID string) {

localChainID := i.sdk.Block().ChainID()

// fromChainID 必须不能等于 toChainID

sdk.Require(fromChainID != toChainID,

types.ErrInvalidParameter, "fromChainID cannot equal to toChainID")

switch tag {

case ibc.RecastPending:

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

// 当前链是侧链，toChainID必须为当前链ID，同时fromChainID必须是有效的链ID

sdk.Require(toChainID == localChainID,

types.ErrInvalidParameter, "invalid to chainID")

sdk.Require(i.isValidChainID(fromChainID),

types.ErrInvalidParameter, "invalid from chainID")

} else {

// 当前链是主链，toChainID必须为有效链ID，同时fromChainID必须是有效的侧链ID

sdk.Require(i.isValidChainID(toChainID),

types.ErrInvalidParameter, "invalid to chainID")

sdk.Require(i.isValidSideChainID(fromChainID),

types.ErrInvalidParameter, "invalid from chainID")

}

case ibc.ConfirmPending, ibc.CancelPending:

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

// 当前链是侧链，fromChainID必须为当前链ID，同时toChainID必须是有效的链ID

sdk.Require(fromChainID == localChainID,

types.ErrInvalidParameter, "invalid from chainID")

sdk.Require(i.isValidChainID(toChainID),

types.ErrInvalidParameter, "invalid to chainID")

} else {

// 当前链是主链，fromChainID必须为有效链ID，同时toChainID必须是有效的侧链ID

sdk.Require(i.isValidChainID(fromChainID),

types.ErrInvalidParameter, "invalid from chainID")

sdk.Require(i.isValidSideChainID(toChainID),

types.ErrInvalidParameter, "invalid to chainID")

}

case ibc.NotifyPending:

sdk.Require(toChainID == localChainID,

types.ErrInvalidParameter, "invalid to chainID")

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

// 当前链是侧链，fromChainID必须是主链ID

sdk.Require(i.isMainChainID(fromChainID),

types.ErrInvalidParameter, "invalid from chainID")

} else {

// 当前链是主链，fromChainID必须是有效侧链ID

sdk.Require(i.isValidSideChainID(fromChainID),

types.ErrInvalidParameter, "invalid from chainID")

}

default:

sdk.Require(false,

types.ErrInvalidParameter, "invalid packet's state tag")

}

}

// isValidChainID if chainID is valid chainID return true, else return false

func (i \*Ibc) isValidChainID(chainID string) bool {

return i.isMainChainID(chainID) || i.isValidSideChainID(chainID)

}

// isMainChainID if chainID is main chain id return true, else return false

func (i \*Ibc) isMainChainID(chainID string) bool {

mainChainID := i.sdk.Helper().BlockChainHelper().GetMainChainID()

return mainChainID == chainID

}

// isValidSideChainID if chainID is valid side chain return true, else return false

func (i \*Ibc) isValidSideChainID(chainID string) bool {

mainChainID := i.sdk.Helper().BlockChainHelper().GetMainChainID()

pattern := "^" + mainChainID + `\[[A-Za-z][a-zA-Z0-9\_]{1,39}\]$`

r, \_ := regexp.Compile(pattern)

return r.MatchString(chainID)

}

// 先判断 QueueID 中 toChainID 一定是我，然后再验签，验签之后的数据只允许在 ，log 中保存，交易是成功的。

func (i \*Ibc) input(packets []ibc.Packet) {

var (

queueID = packets[0].QueueID

sequence uint64

)

forx.Range(packets, func(index int, packet ibc.Packet) bool {

i.processPacket(packet)

i.\_setQueueIndex(queueID, packet.Seq, MessageIndex{

Height: i.sdk.Block().Height(),

IbcHash: packet.IbcHash,

})

sequence = packet.Seq

return true

})

// Saves the sequence number of the source chain to the current chain.

i.\_setSequence(queueID, sequence)

}

func (i \*Ibc) processPacket(packet ibc.Packet) {

switch packet.Type {

case ibc.TccTxType:

i.processTccTxPacket(packet)

case ibc.NotifyType:

i.processNotifyPacket(packet)

}

}

func (i \*Ibc) processTccTxPacket(packet ibc.Packet) {

if i.isRelay(&packet) {

switch packet.State.Tag {

case ibc.RecastPending:

if err := i.checkSideChain(&packet); err.ErrorCode != types.CodeOK {

i.processTryRecastResult(&packet, false, nil, err)

} else {

ok, outReceipts, err := i.sdk.Helper().IBCStubHelper().TryRecast(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processTryRecastResult(&packet, ok, outReceipts, err)

}

case ibc.ConfirmPending:

outReceipts, err := i.sdk.Helper().IBCStubHelper().ConfirmRecast(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processConfirmRecastResult(&packet, outReceipts, err)

case ibc.CancelPending:

outReceipts, err := i.sdk.Helper().IBCStubHelper().CancelRecast(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processCancelRecastResult(&packet, outReceipts, err)

}

} else {

switch packet.State.Tag {

case ibc.RecastPending:

if i.sdk.Helper().BlockChainHelper().IsSideChain() {

ok, outReceipts, err := i.sdk.Helper().IBCStubHelper().Recast(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processRecastResult(&packet, ok, outReceipts, err)

} else {

if err := i.checkSideChain(&packet); err.ErrorCode != types.CodeOK {

i.processRecastResult(&packet, false, nil, err)

} else {

ok, outReceipts, err := i.sdk.Helper().IBCStubHelper().Recast(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processRecastResult(&packet, ok, outReceipts, err)

}

}

case ibc.ConfirmPending:

outReceipts, err := i.sdk.Helper().IBCStubHelper().Confirm(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processConfirmResult(&packet, outReceipts, err)

case ibc.CancelPending:

outReceipts, err := i.sdk.Helper().IBCStubHelper().Cancel(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processCancelResult(&packet, outReceipts, err)

}

}

}

func (i \*Ibc) processNotifyPacket(packet ibc.Packet) {

if packet.ContractName == "netgovernance" {

forx.Range(packet.Receipts, func(index int, kvPair types.KVPair) {

var receipt std.Receipt

\_ = jsoniter.Unmarshal(kvPair.Value, &receipt)

if strings.HasSuffix(string(kvPair.Key), "/netgovernance.setOpenURL") {

newOpenURLs := new(setOpenURL)

\_ = jsoniter.Unmarshal(receipt.Bytes, newOpenURLs)

i.\_setOpenURLs(newOpenURLs.ChainID, newOpenURLs.OpenURLs)

i.sdk.Helper().ReceiptHelper().Emit(newOpenURLs)

} else if strings.HasSuffix(string(kvPair.Key), "/netgovernance.setGasPriceRatio") {

newGasPriceRatio := new(setGasPriceRatio)

\_ = jsoniter.Unmarshal(receipt.Bytes, newGasPriceRatio)

i.\_setGasPriceRatio(newGasPriceRatio.GasPriceRatio)

i.sdk.Helper().ReceiptHelper().Emit(newGasPriceRatio)

}

})

} else {

outReceipts, err := i.sdk.Helper().IBCStubHelper().Notify(packet.IbcHash, packet.OrgID, packet.ContractName, packet.Receipts)

i.processNotifyResult(&packet, outReceipts, err)

}

}

func (i \*Ibc) checkPrecommits(h \*ibc.Header, precommits []ibc.Precommit, hasGovernance bool) bool {

chainValidators := i.\_chainValidators(h.ChainID)

if len(chainValidators) == 0 {

return false

}

var totalPower, verifiedPower int64

forx.Range(chainValidators, func(key string, validator InfoOfValidator) bool {

totalPower += validator.Power

return forx.Continue

})

calcHash := i.sdk.Helper().IBCHelper().CalcBlockHash(h)

forx.Range(precommits, func(index int, precommit ibc.Precommit) bool {

validator, ok := chainValidators[precommit.ValidatorAddress]

if !ok {

return forx.Continue

}

if bytes.Compare(calcHash, precommit.BlockID.Hash) != 0 {

return forx.Continue

}

if i.sdk.Helper().IBCHelper().VerifyPrecommit(validator.PubKey, precommit, h.ChainID, h.Height) {

verifiedPower += validator.Power

}

return forx.Continue

})

if hasGovernance {

return verifiedPower >= totalPower\*1/2

}

return verifiedPower > totalPower\*2/3

}

func (i \*Ibc) checkQueueHash(queueChain \*ibc.QueueChain, packets []ibc.Packet) {

queueID := packets[0].QueueID

lastQueueHash := i.\_lastQueueHash(queueID)

var queueBlock ibc.QueueBlock

forx.Range(queueChain.QueueBlocks, func(index int, queueItem ibc.QueueBlock) bool {

if queueItem.QueueID == queueID {

queueBlock = queueItem

return forx.Break

}

return forx.Continue

})

// check last queue hash

sdk.Require(lastQueueHash == nil || bytes.Compare(lastQueueHash, queueBlock.LastQueueHash) == 0,

types.ErrInvalidParameter, "invalid lastQueueHash")

// check queue hash

queueHash := i.calcQueueHash(queueID, queueChain, packets)

sdk.Require(bytes.Compare(queueHash, queueBlock.QueueHash) == 0,

types.ErrInvalidParameter, "invalid queueHash")

i.\_setLastQueueHash(queueID, queueHash)

}

func (i \*Ibc) calcQueueHash(queueID string, queueChain \*ibc.QueueChain, packets []ibc.Packet) types.Hash {

lastQueueHash := i.\_lastQueueHash(queueID)

// calc queue hash

var queueHash types.Hash

lastIndex := len(packets) - 1

forx.Range(packets, func(index int, packet ibc.Packet) bool {

itemHash := i.sdk.Helper().IBCHelper().CalcQueueHash(packet, lastQueueHash)

if index != lastIndex {

lastQueueHash = itemHash

} else {

queueHash = itemHash

}

return forx.Continue

})

return queueHash

}

func (i \*Ibc) \_sequence(queueID string) uint64 {

return i.sdk.Helper().StateHelper().GetUint64(keyOfSequence(queueID))

}

func (i \*Ibc) \_setSequence(queueID string, value uint64) {

i.sdk.Helper().StateHelper().SetUint64(keyOfSequence(queueID), value)

}

func (i \*Ibc) \_packets(ibcHash types.Hash) []ibc.Packet {

return \*i.sdk.Helper().StateHelper().GetEx(keyOfPackets(ibcHash), new([]ibc.Packet)).(\*[]ibc.Packet)

}

func (i \*Ibc) \_setPackets(ibcHash types.Hash, packets []ibc.Packet) {

i.sdk.Helper().StateHelper().Set(keyOfPackets(ibcHash), &packets)

}

func (i \*Ibc) \_chainInfo(chainID string) ChainInfo {

return \*i.sdk.Helper().StateHelper().GetEx(keyOfChainInfo(chainID), new(ChainInfo)).(\*ChainInfo)

}

func (i \*Ibc) \_sideChainIDs() []string {

return \*i.sdk.Helper().StateHelper().GetEx(keyOfSideChainIDs(), new([]string)).(\*[]string)

}

func (i \*Ibc) \_setState(ibcHash types.Hash, state \*ibc.State) {

i.sdk.Helper().StateHelper().Set(keyOfState(ibcHash), state)

}

func (i \*Ibc) \_setQueueIndex(queueID string, seq uint64, msgIndex MessageIndex) {

i.sdk.Helper().StateHelper().Set(keyOfSequenceHeight(queueID, seq), &msgIndex)

}

// --------------- proof ------------------------

func (i \*Ibc) \_lastQueueHash(queueID string) types.Hash {

return \*i.sdk.Helper().StateHelper().GetEx(keyOfLastQueueHash(queueID), new(types.Hash)).(\*types.Hash)

}

func (i \*Ibc) \_setLastQueueHash(queueID string, lastQueueHash types.Hash) {

i.sdk.Helper().StateHelper().Set(keyOfLastQueueHash(queueID), lastQueueHash)

}

func (i \*Ibc) \_chainValidators(chainID string) (chainValidators map[string]InfoOfValidator) {

return \*i.sdk.Helper().StateHelper().GetEx("/ibc/"+chainID, &chainValidators).(\*map[string]InfoOfValidator)

}

// --------------- openUrls ------------------------

func (i \*Ibc) \_setOpenURLs(chainID string, urls []string) {

i.sdk.Helper().StateHelper().Set(keyOfSetOpenURLs(chainID), &urls)

}

// --------------- gasPriceRatio ------------------------

func (i \*Ibc) \_setGasPriceRatio(gasPriceRatio string) {

i.sdk.Helper().StateHelper().Set(std.KeyOfGasPriceRatio(), gasPriceRatio)

}

//NetGovernance This is struct of contract

//@:contract:netgovernance

//@:version:2.1

//@:organization:orgJgaGConUyK81zibntUBjQ33PKctpk1K1G

//@:author:5e8339cb1a5cce65602fd4f57e115905348f7e83gie38dd77694dbe1f8903c9

type NetGovernance struct {

sdk sdk.ISmartContract

chainInfo \*ChainInfo

}

//InitChain Constructor of this NetGovernance

//@:constructor

func (ng \*NetGovernance) InitChain() {

sdk.RequireMainChain()

}

//UpdateChain Constructor of this NetGovernance

//@:constructor

func (ng \*NetGovernance) UpdateChain() {

sdk.RequireMainChain()

}

//@:public:method:gas[50000]

func (ng \*NetGovernance) RegisterSideChain(chainName, orgName string, ownerAddr types.Address) {

// check sender

sdk.RequireOwner()

// check chainName

sideChainID := ng.checkChainName(chainName, false)

// check organization

ng.checkOrganization(orgName)

// check ownerAddr

ng.checkOwnerAddr(ownerAddr)

// save chainInfo

chainInfo := ChainInfo{

SideChainName: chainName,

ChainID: sideChainID,

OrgName: orgName,

Owner: ownerAddr,

Status: Init,

}

ng.\_setChainInfo(chainInfo)

//save sideChainID

sideChainIDList := ng.\_sideChainIDs()

sideChainIDList = append(sideChainIDList, sideChainID)

ng.\_setSideChainIDs(sideChainIDList)

// send receipt

ng.emitRegisterSideChainReceipt(sideChainID, orgName, ownerAddr)

}

//@:public:method:gas[5000000]

func (ng \*NetGovernance) GenesisSideChain(

chainName string,

nodeName string,

nodePubKey types.PubKey,

rewardAddr types.Address,

openURL string,

gasPriceRatio string) {

mainOpenURLs := ng.\_openURLs(ng.sdk.Block().ChainID())

sdk.Require(len(mainOpenURLs) != 0,

types.ErrInvalidParameter, "main chain must set openURLs")

// check chainName

sideChainID, chainInfo := ng.checkSideChainInfo(chainName, Init)

// check nodeName

ng.checkNodeName(nodeName)

// check nodePubKey

nodeAddr := ng.checkNodePubKey(nodePubKey, sideChainID)

// check rewardAddr

sdk.RequireAddressEx(sideChainID, rewardAddr)

// check openurl

openURLs := []string{openURL}

ng.checkOpenUrls(openURLs)

// check gasPriceRatio

gasPriceRatio = ng.checkGasPriceRatio(gasPriceRatio)

// save urls

ng.\_setOpenURLs(sideChainID, openURLs)

chainInfo.NodeNames = []string{nodeName}

chainInfo.Status = Ready

chainInfo.Height = ng.sdk.Block().Height()

chainInfo.GasPriceRatio = gasPriceRatio

ng.\_setChainInfo(chainInfo)

// generate validator

validator := Validator{

PubKey: nodePubKey,

Power: 10,

RewardAddr: rewardAddr,

Name: nodeName,

NodeAddr: nodeAddr,

}

//save Validator

nodes := make(map[string]Validator)

nodes[nodeAddr] = validator

ng.\_setChainValidator(sideChainID, nodes)

// send receipt

genesisInfo, contractsData := ng.makeGenesisInfo(chainInfo, validator)

ng.emitGenesisSideChainReceipt(

sideChainID,

openURLs,

genesisInfo,

contractsData)

}

//@:public:method:gas[50000]

func (ng \*NetGovernance) SetOpenURLs(chainName string, openURLs []string) {

//check chainInfo

var chainID string

if chainName != ng.sdk.Block().ChainID() {

chainID, \_ = ng.checkSideChainInfo(chainName, Ready)

} else {

chainID = ng.checkMainChainInfo()

}

//check openURLs

ng.checkOpenUrls(openURLs)

//save openURLs

ng.\_setOpenURLs(chainID, openURLs)

// broadcast if chainID is mainChain

if chainID == ng.sdk.Helper().BlockChainHelper().GetMainChainID() {

ng.sdk.Helper().IBCHelper().Run(func() {

ng.emitSetOpenURLReceipt(chainID, openURLs)

}).Broadcast()

} else {

//send receipt

ng.emitSetOpenURLReceipt(chainID, openURLs)

}

}

//@:public:method:gas[50000]

func (ng \*NetGovernance) SetStatus(chainName string, status string) {

sdk.RequireOwner()

// check chain name

chainID := ng.checkChainName(chainName, true)

// check status

chainInfo := ng.checkStatus(chainID, status)

//save chain status

chainInfo.Status = status

ng.\_setChainInfo(chainInfo)

//send receipt

ng.emitSetStatusReceipt(chainID, status)

}

//@:public:method:gas[50000]

func (ng \*NetGovernance) SetGasPriceRatio(chainName string, gasPriceRatio string) {

//check sideChainInfo

chainID, chainInfo := ng.checkSideChainInfo(chainName, Ready)

//check gasPriceRatio

ng.checkGasPriceRatio(gasPriceRatio)

// update chainInfo

chainInfo.GasPriceRatio = gasPriceRatio

ng.\_setChainInfo(chainInfo)

toChainIDs := make([]string, 0)

toChainIDs = append(toChainIDs, chainID)

ng.sdk.Helper().IBCHelper().Run(func() {

ng.emitSetGasPriceRatioReceipt(chainInfo.SideChainName, chainID, gasPriceRatio)

}).Notify(toChainIDs)

}

func (ng \*NetGovernance) makeGenesisInfo(

chainInfo ChainInfo,

validator Validator) (string, []ContractData) {

genesisTokenAddr := ng.sdk.Helper().GenesisHelper().Token().Address()

baseToken := ng.sdk.Helper().TokenHelper().TokenOfAddress(genesisTokenAddr)

contracts, contractsData := ng.getContractsForGenesis(chainInfo)

mainChainValidators := ng.getMainChainValidators()

chainVersion := ng.\_chainVersion()

var mainChainInfo MainChainInfo

mainChainUrls := ng.\_openURLs(ng.sdk.Block().ChainID())

mainChainInfo.OpenUrls = mainChainUrls

mainChainInfo.Validators = mainChainValidators

bcHelper := ng.sdk.Helper().BlockChainHelper()

genesis := GenesisInfo{

ChainID: chainInfo.ChainID,

ChainVersion: fmt.Sprintf("%d", chainVersion),

GenesisTime: ng.sdk.Helper().BlockChainHelper().FormatTime(ng.sdk.Block().Time(), "2006-01-02T15:04:05.999999999Z07:00"),

AppHash: "",

AppState: AppState{

Organization: "genesis",

GasPriceRatio: chainInfo.GasPriceRatio,

Token: std.Token{

Address: bcHelper.RecalcAddress(genesisTokenAddr, chainInfo.ChainID),

Owner: bcHelper.RecalcAddress(baseToken.Owner().Address(), chainInfo.ChainID),

Name: baseToken.Name(),

Symbol: baseToken.Symbol(),

TotalSupply: bn.N(0),

AddSupplyEnabled: baseToken.AddSupplyEnabled(),

BurnEnabled: baseToken.BurnEnabled(),

GasPrice: baseToken.GasPrice(),

},

RewardStrategy: []Reward{

{

Name: "validators",

RewardPercent: "100.00",

Address: "",

},

},

Contracts: contracts,

OrgBind: OrgBind{

OrgName: chainInfo.OrgName,

Owner: bcHelper.RecalcAddress(chainInfo.Owner, chainInfo.ChainID),

},

MainChain: mainChainInfo,

},

Validators: []Validator{validator},

}

info, err := jsoniter.Marshal(genesis)

sdk.RequireNotError(err, types.ErrInvalidParameter)

return string(info), contractsData

}

func (ng \*NetGovernance) getContractsForGenesis(chainInfo ChainInfo) (contracts []Contract, contractData []ContractData) {

genesisContracts := []string{"token-basic", "token-issue", "governance",

"organization", "smartcontract", "ibc"}

forx.Range(genesisContracts, func(i int, genesisContract string) bool {

c := ng.sdk.Helper().ContractHelper().ContractOfName(genesisContract)

contractOwner := ""

if genesisContract == "governance" {

contractOwner = ng.sdk.Helper().BlockChainHelper().RecalcAddress(chainInfo.Owner, chainInfo.ChainID)

}

codeDevSig, codeOrgSig := ng.getConCodeSig(c.Address())

code := c.Name() + "-" + c.Version() + ".tar.gz"

con := Contract{

Name: c.Name(),

Version: c.Version(),

Code: code,

CodeHash: hex.EncodeToString(c.CodeHash()),

Owner: contractOwner,

CodeDevSig: codeDevSig,

CodeOrgSig: codeOrgSig,

}

contracts = append(contracts, con)

conData := ContractData{

Name: c.Name(),

Version: c.Version(),

CodeByte: ng.\_contractCode(c.Address()),

}

contractData = append(contractData, conData)

return forx.Continue

})

return contracts, contractData

}

// 公钥对应的节点不能是主链的节点

func (ng \*NetGovernance) checkNodePubKey(nodePubKey types.PubKey, sideChainID string) types.Address {

mainChainAddr := ng.sdk.Helper().BlockChainHelper().CalcAccountFromPubKey(nodePubKey)

sdk.Require(!ng.\_chkGIValidator(mainChainAddr),

types.ErrInvalidParameter, "Can not use mainChain validator")

sideChainNodeAddr := ng.sdk.Helper().BlockChainHelper().RecalcAddress(mainChainAddr, sideChainID)

return sideChainNodeAddr

}

func (ng \*NetGovernance) checkOrganization(orgName string) {

sdk.Require(len(orgName) > 0,

types.ErrInvalidParameter, "Invalid orgName")

genesisOrgID := ng.sdk.Helper().GenesisHelper().OrgID()

orgID := ng.sdk.Helper().BlockChainHelper().CalcOrgID(orgName)

sdk.Require(orgID != genesisOrgID,

types.ErrInvalidParameter, "SideChain organization could not be genesis organization")

sdk.Require(ng.\_chkOrganization(orgID),

types.ErrInvalidParameter, "There is no organization with name "+orgName)

}

func (ng \*NetGovernance) getConCodeSig(conAddr types.Address) (codeDevSig, codeOrgSig Signature) {

type contractMeta struct {

Name string `json:"name"`

ContractAddr types.Address `json:"contractAddr"`

OrgID string `json:"orgID"`

Version string `json:"version"`

EffectHeight int64 `json:"effectHeight"`

LoseHeight int64 `json:"loseHeight"`

CodeData []byte `json:"codeData"`

CodeHash []byte `json:"codeHash"`

CodeDevSig []byte `json:"codeDevSig"`

CodeOrgSig []byte `json:"codeOrgSig"`

}

key := std.KeyOfContractCode(conAddr)

conMeta := ng.sdk.Helper().StateHelper().Get(key, new(contractMeta)).(\*contractMeta)

var codeDevSigStr, codeOrgSigStr string

err := jsoniter.Unmarshal(conMeta.CodeDevSig, &codeDevSigStr)

sdk.RequireNotError(err, types.ErrInvalidParameter)

err = jsoniter.Unmarshal(conMeta.CodeOrgSig, &codeOrgSigStr)

sdk.RequireNotError(err, types.ErrInvalidParameter)

codeDevSigByte := []byte(codeDevSigStr)

codeOrgSigByte := []byte(codeOrgSigStr)

err = jsoniter.Unmarshal(codeDevSigByte, &codeDevSig)

sdk.RequireNotError(err, types.ErrInvalidParameter)

err = jsoniter.Unmarshal(codeOrgSigByte, &codeOrgSig)

sdk.RequireNotError(err, types.ErrInvalidParameter)

return codeDevSig, codeOrgSig

}

func (ng \*NetGovernance) getMainChainValidators() map[string]Validator {

mainChainValidatorAddrList := ng.\_giValidator()

mainChainValidators := make(map[string]Validator)

forx.Range(mainChainValidatorAddrList, func(i int, addr string) {

valInfo := \*ng.sdk.Helper().StateHelper().GetEx("/validator/"+addr,

&Validator{}).(\*Validator)

mainChainValidators[addr] = valInfo

})

return mainChainValidators

}

func (ng \*NetGovernance) checkChainName(chainName string, occupiedWanted bool) (chainID string) {

sdk.Require(chainName != "",

types.ErrInvalidParameter, "ChainName should not be nil")

sdk.Require(chainName != ng.sdk.Block().ChainID(),

types.ErrInvalidParameter, "ChainName should not be mainChain")

chainID = ng.sdk.Helper().BlockChainHelper().CalcSideChainID(chainName)

var errInfo string

if occupiedWanted == false {

errInfo = "ChainName has been occupied "

} else {

errInfo = "ChainName does not exits "

}

sdk.Require(ng.\_chkChainInfo(chainID) == occupiedWanted,

types.ErrInvalidParameter, errInfo)

return

}

func (ng \*NetGovernance) checkOwnerAddr(ownerAddr types.Address) {

//检查地址格式

sdk.RequireAddress(ownerAddr)

//侧链Owner地址不能为主链委员会地址

genesisOwnerAddr := ng.sdk.Message().Contract().Owner().Address()

sdk.Require(ownerAddr != genesisOwnerAddr,

types.ErrInvalidParameter, "Invalid owner Address")

}

func (ng \*NetGovernance) checkOpenUrls(openURLs []string) {

sdk.Require(len(openURLs) > 0,

types.ErrInvalidParameter, "OpenUrls should not be empty")

urlExpr := `^(https|http)://`

forx.Range(openURLs, func(i int, openURL string) bool {

match, \_ := regexp.MatchString(urlExpr, openURL)

sdk.Require(match,

types.ErrInvalidParameter, fmt.Sprintf("Invalid openURL: %v ", openURL))

return forx.Continue

})

}

func (ng \*NetGovernance) checkSideChainInfo(chainName, statusWanted string) (chainID string, chainInfo ChainInfo) {

// check chainName

chainID = ng.checkChainName(chainName, true)

// check sender

sdk.Require(ng.sdk.Message().Sender().Address() == ng.\_chainInfo(chainID).Owner,

types.ErrNoAuthorization, "Only SideChain Owner can do this ")

// check chain status

chainInfo = ng.\_chainInfo(chainID)

sdk.Require(chainInfo.Status == statusWanted,

types.ErrInvalidParameter, "The status of sideChain is error ")

return

}

func (ng \*NetGovernance) checkMainChainInfo() string {

//check Sender

sdk.RequireOwner()

return ng.sdk.Helper().BlockChainHelper().GetMainChainID()

}

func (ng \*NetGovernance) checkNodeName(nodeName string) {

sdk.Require(len(nodeName) > 0 && len(nodeName) <= MaxNameLen,

types.ErrInvalidParameter, fmt.Sprintf("The length of nodeName should be within (0,%d]", MaxNameLen))

}

func (ng \*NetGovernance) checkStatus(chainID, status string) ChainInfo {

chainInfo := ng.\_chainInfo(chainID)

sdk.Require((status == Ready && chainInfo.Status == Clear) ||

(status == Clear && chainInfo.Status == Ready) ||

(status == Disabled && chainInfo.Status == Clear),

types.ErrInvalidParameter, "Status value error!")

return chainInfo

}

func (ng \*NetGovernance) checkGasPriceRatio(gasPriceRatio string) string {

//检查为空

if gasPriceRatio == "" {

return "1.000"

}

//检查格式精确到小数点后三位

urlExpr := `^[0-9]+(.[0-9]{3})?$`

match, \_ := regexp.MatchString(urlExpr, gasPriceRatio)

sdk.Require(match,

types.ErrInvalidParameter, fmt.Sprintf("Invalid gasPriceRatio: %v ", gasPriceRatio))

return gasPriceRatio

}

func (ng \*NetGovernance) emitRegisterSideChainReceipt(sideChainID, orgName string, ownerAddr types.Address) {

type registerSideChain struct {

SideChainID string `json:"sideChainID"`

OrgName string `json:"orgName"`

OwnerAddr types.Address `json:"ownerAddress"`

}

receipt := registerSideChain{

SideChainID: sideChainID,

OrgName: orgName,

OwnerAddr: ownerAddr,

}

ng.sdk.Helper().ReceiptHelper().Emit(receipt)

}

func (ng \*NetGovernance) emitGenesisSideChainReceipt(sideChainID string,

openURLs []string, genesisInfo string,

contractsData []ContractData) {

type genesisSideChain struct {

SideChainID string `json:"sideChainID"`

OpenURLs []string `json:"openURLs"`

GenesisInfo string `json:"genesisInfo"`

ContractData []ContractData `json:"contractData"`

}

receipt := genesisSideChain{

SideChainID: sideChainID,

OpenURLs: openURLs,

GenesisInfo: genesisInfo,

ContractData: contractsData,

}

ng.sdk.Helper().ReceiptHelper().Emit(receipt)

}

func (ng \*NetGovernance) emitSetOpenURLReceipt(sideChainID string, openURLs []string) {

type setOpenURL struct {

SideChainID string `json:"sideChainID"`

OpenURLs []string `json:"openURLs"`

}

receipt := setOpenURL{

SideChainID: sideChainID,

OpenURLs: openURLs,

}

ng.sdk.Helper().ReceiptHelper().Emit(receipt)

}

func (ng \*NetGovernance) emitSetStatusReceipt(sideChainID string, status string) {

type setStatus struct {

SideChainID string `json:"sideChainID"`

Status string `json:"status"`

}

receipt := setStatus{

SideChainID: sideChainID,

Status: status,

}

ng.sdk.Helper().ReceiptHelper().Emit(receipt)

}

func (ng \*NetGovernance) emitSetGasPriceRatioReceipt(chainName, chainID, gasPriceRatio string) {

type setGasPriceRatio struct {

ChainName string `json:"chainName"`

ChainID string `json:"chainID"`

GasPriceRatio string `json:"gasPriceRatio"`

}

receipt := setGasPriceRatio{

ChainName: chainName,

ChainID: chainID,

GasPriceRatio: gasPriceRatio,

}

ng.sdk.Helper().ReceiptHelper().Emit(receipt)

}

// ChainInfo

func (ng \*NetGovernance) \_setChainInfo(ci ChainInfo) {

ng.sdk.Helper().StateHelper().Set(keyOfChainInfo(ci.ChainID), &ci)

}

func (ng \*NetGovernance) \_chainInfo(chainID string) ChainInfo {

return \*ng.sdk.Helper().StateHelper().Get(keyOfChainInfo(chainID), new(ChainInfo)).(\*ChainInfo)

}

// ChainValidatorPubKeys

func (ng \*NetGovernance) \_setChainValidator(chainID string, cvp map[string]Validator) {

ng.sdk.Helper().StateHelper().Set("/ibc/"+chainID, &cvp)

}

// Get open URL

func (ng \*NetGovernance) \_openURLs(chainID string) []string {

return \*ng.sdk.Helper().StateHelper().GetEx(keyOfOpenURLs(chainID), new([]string)).(\*[]string)

}

// contract code

func (ng \*NetGovernance) \_contractCode(contractAddr types.Address) []byte {

key := keyOfContractCode(contractAddr)

contractMeta := ng.sdk.Helper().StateHelper().GetEx(key, new(std.ContractMeta)).(\*std.ContractMeta)

return contractMeta.CodeData

}

// validator

func (ng \*NetGovernance) \_setSideChainIDs(sideChainIDs []string) {

ng.sdk.Helper().StateHelper().Set(keyOfSideChainIDs(), &sideChainIDs)

}

func (ng \*NetGovernance) \_chainVersion() int64 {

type appState struct {

ChainVersion int64 `json:"chain\_version,omitempty"` //当前链版本

}

return ng.sdk.Helper().StateHelper().Get(keyOfWorldAppState(), new(appState)).(\*appState).ChainVersion

}