//NewBCChainApplication create an application object

func NewBCChainApplication(config common.Config, logger log.Loggerf) \*BCChainApplication {

logger.Info("Init gichain begin", "version", version.Version)

db, ret := statedb.Init(config.DBName, config.DBIP, config.DBPort)

if !ret {

err := errors.New("init statedb error")

logger.Fatal("Failed to startup the stateDB", "error", err)

panic(err)

}

app := BCChainApplication{

connQuery: &query.QueryConnection{},

connCheck: &check.AppCheck{},

connDeliver: &deliver.AppDeliver{},

logger: logger,

db: db,

}

softforks.Init()

app.connQuery.SetLogger(logger)

app.connCheck.SetLogger(logger)

app.connDeliver.SetLogger(logger)

chainID := statedbhelper.GetChainID()

app.connCheck.SetChainID(chainID)

app.connDeliver.SetChainID(chainID)

crypto.SetChainId(chainID)

adapterIns := adapter.GetInstance()

adapterIns.Init(logger, 32333)

adapter.SetSdbCallback(statedbhelper.AdapterGetCallBack, statedbhelper.AdapterSetCallBack, builderhelper.AdapterBuildCallBack)

if checkGenesisChainVersion() == 0 {

app.appv1 = appv1.NewBCChainApplication(db, logger)

}

logger.Info("Init gichain end")

return &app

}

//Echo echo interface

func (app \*BCChainApplication) Echo(req types.RequestEcho) types.ResponseEcho {

res := app.connQuery.Echo(req)

return res

}

//Info info interface

func (app \*BCChainApplication) Info(req types.RequestInfo) types.ResponseInfo {

res := app.connQuery.Info(req)

return res

}

//SetOption set option interface

func (app \*BCChainApplication) SetOption(req types.RequestSetOption) types.ResponseSetOption {

res := app.connQuery.SetOption(req)

return res

}

//Query query interface

func (app \*BCChainApplication) Query(reqQuery types.RequestQuery) types.ResponseQuery {

res := app.connQuery.Query(reqQuery)

return res

}

//Query queryEx interface

func (app \*BCChainApplication) QueryEx(reqQuery types.RequestQueryEx) types.ResponseQueryEx {

res := app.connQuery.QueryEx(reqQuery)

return res

}

//CheckTx checkTx interface

func (app \*BCChainApplication) CheckTx(tx []byte) types.ResponseCheckTx {

var res types.ResponseCheckTx

state := statedbhelper.GetWorldAppState(0, 0)

splitTx := strings.Split(string(tx), ".")

if len(splitTx) == 5 {

if splitTx[1] == "v1" && app.appv1 != nil {

var connV2 \*check.AppCheck

if state.ChainVersion == 2 {

connV2 = app.connCheck

}

res = app.appv1.CheckTx(tx, connV2)

} else if (splitTx[1] == "v2" || splitTx[1] == "v3") && state.ChainVersion == 2 {

res = app.connCheck.CheckTx(tx)

} else {

res.Code = types2.ErrLogicError

res.Log = "invalid transaction 1"

}

} else {

res.Code = types2.ErrLogicError

res.Log = "invalid transaction 2"

fmt.Println("tx:", string(tx))

fmt.Println("tx len:", len(splitTx))

}

res.TxHash = cmn.HexBytes(algorithm.CalcCodeHash(string(tx)))

return res

}

//DeliverTx deliverTx interface

func (app \*BCChainApplication) DeliverTx(tx []byte) types.ResponseDeliverTx {

var res types.ResponseDeliverTx

state := statedbhelper.GetWorldAppState(0, 0)

splitTx := strings.Split(string(tx), ".")

if len(splitTx) == 5 {

if splitTx[1] == "v1" && app.appv1 != nil {

// if chain version never upgrade, give appv2 nil.

var connV2 \*deliver.AppDeliver

if state.ChainVersion == 2 {

connV2 = app.connDeliver

}

res = app.appv1.DeliverTx(tx, connV2)

} else if (splitTx[1] == "v2" || splitTx[1] == "v3") && state.ChainVersion == 2 {

var txBuffer map[string][]byte

res, txBuffer = app.connDeliver.DeliverTx(tx)

if app.appv1 != nil {

app.appv1.CommitTx2V1(txBuffer)

}

} else {

res.Code = types2.ErrLogicError

res.Log = "invalid transaction"

}

} else {

res.Code = types2.ErrLogicError

res.Log = "invalid transaction"

}

res.TxHash = cmn.HexBytes(algorithm.CalcCodeHash(string(tx)))

return res

}

//Flush flush interface

func (app \*BCChainApplication) Flush(req types.RequestFlush) types.ResponseFlush {

res := app.connDeliver.Flush(req)

return res

}

//Commit commit interface

func (app \*BCChainApplication) Commit() types.ResponseCommit {

var res types.ResponseCommit

state := statedbhelper.GetWorldAppState(0, 0)

if state.ChainVersion == 0 {

res = app.appv1.Commit()

} else if state.ChainVersion == 2 {

if checkGenesisChainVersion() == 0 {

app.appv1.CommitToV2()

}

res = app.connDeliver.Commit()

} else {

panic("invalid chain version in state")

}

return res

}

//InitChain 初次初始化链后立马注册合约

func (app \*BCChainApplication) InitChain(req types.RequestInitChain) types.ResponseInitChain {

var res types.ResponseInitChain

if req.ChainVersion == 0 {

if app.appv1 == nil {

app.appv1 = appv1.NewBCChainApplication(app.db, app.logger)

}

res = app.appv1.InitChain(req)

} else if req.ChainVersion == 2 {

res = app.connDeliver.InitChain(req)

app.appv1 = nil

} else {

res.Code = types2.ErrLogicError

res.Log = "invalid genesis doc"

}

return res

}

//BeginBlock beginblock interface

func (app \*BCChainApplication) BeginBlock(req types.RequestBeginBlock) types.ResponseBeginBlock {

var res types.ResponseBeginBlock

state := statedbhelper.GetWorldAppState(0, 0)

if state.ChainVersion == 0 {

res = app.appv1.BeginBlock(req)

} else if state.ChainVersion == 2 {

// if chain was upgrade from v1, then invoke appv1 BeginBlockToV2 before v2 BeginBlock

if checkGenesisChainVersion() == 0 {

app.appv1.BeginBlockToV2(req)

}

var txBuffer map[string][]byte

res, txBuffer = app.connDeliver.BeginBlock(req)

if app.appv1 != nil {

app.appv1.CommitTx2V1(txBuffer)

}

} else {

panic("invalid chain version in state")

}

return res

}

// CleanData clean all gichain data when side chain genesis

func (app \*BCChainApplication) CleanData() types.ResponseCleanData {

response := types.ResponseCleanData{

Code: 200,

Log: "",

}

if err := app.connDeliver.CleanData(); err != nil {

response.Code = types2.ErrLogicError

response.Log = err.Error()

}

return response

}

func checkGenesisChainVersion() int {

value, err := statedbhelper.Get(std.KeyOfGenesisChainVersion())

if err != nil {

panic(err)

}

if len(value) == 0 {

return 0

}

var genesisChainVersion int64

err = jsoniter.Unmarshal(value, &genesisChainVersion)

if err != nil {

panic(err)

}

if genesisChainVersion == 0 {

return 0

} else if genesisChainVersion == 2 {

return 2

}

panic("invalid genesisChainVersion")

}

//SetLogger set logger

func (app \*AppCheck) SetLogger(logger log.Logger) {

app.logger = logger

}

//SetChainID set chainID

func (app \*AppCheck) SetChainID(chainID string) {

app.chainID = chainID

}

//CheckTx check tx

func (app \*AppCheck) CheckTx(tx []byte) types.ResponseCheckTx {

app.logger.Info("Recv ABCI interface: CheckTx", "tx", string(tx))

return app.CheckBCTx(tx)

}

// ------------- add for support v1 transaction begin ----------------

//RunCheckTx - invoked by v1 checkTx, if it's standard transfer method.

func (app \*AppCheck) RunCheckTx(tx []byte, transaction types2.Transaction, pubKey crypto.PubKeyEd25519) types.ResponseCheckTx {

app.logger.Debug("Recv ABCI interface: CheckTx", "transaction", transaction)

return app.runCheckBCTx(tx, transaction, pubKey)

}

// CheckBCTx check tx data

func (app \*AppCheck) CheckBCTx(tx []byte) types.ResponseCheckTx {

if app.chainID == "" {

app.SetChainID(statedbhelper.GetChainID())

}

// for base58

tx2.Init(app.chainID)

transaction, pubKey, err := tx2.TxParse(string(tx))

if err != nil {

// for base64

tx3.Init(app.chainID)

transaction, pubKey, err = tx3.TxParse(string(tx))

if err != nil {

app.logger.Error("tx parse failed:", "error", err)

return types.ResponseCheckTx{

Code: types2.ErrCheckTx,

Log: err.Error()}

}

}

return app.runCheckBCTx(tx, transaction, pubKey)

}

func (app \*AppCheck) runCheckBCTx(tx []byte, transaction types2.Transaction, pubKey crypto.PubKeyEd25519) types.ResponseCheckTx {

// Check note

if len(transaction.Note) > types2.MaxSizeNote {

return types.ResponseCheckTx{

Code: types2.ErrCheckTx,

Log: "Invalid transaction note"}

}

transID := statedbhelper.NewTransactionID()

txID := int64(1)

// Check Nonce

err := statedbhelper.CheckAccountNonce(transID, txID, pubKey.Address(statedbhelper.GetChainID()), transaction.Nonce)

if err != nil {

app.logger.Debug("check nonce error:", "err", err)

return types.ResponseCheckTx{

Code: types2.ErrCheckTx,

Log: "Invalid nonce"}

}

statedbhelper.BeginBlock(transID)

defer statedbhelper.RollbackBlock(transID)

adp := adapter.GetInstance()

defer adp.Rollback(transID)

appStat := statedbhelper.GetWorldAppState(0, 0)

blockHeader := types.Header{}

if appStat.BlockHeight == 0 {

blockHeader.ChainID = app.chainID

blockHeader.Height = 0

} else {

blockHeader = appStat.BeginBlock.Header

blockHeader.Height = blockHeader.Height + 1

}

app.logger.Debug("CheckTx", "block height", blockHeader.Height)

txHash := common.HexBytes(algorithm.CalcCodeHash(string(tx)))

result := adp.InvokeTx(blockHeader, transID, txID, pubKey.Address(statedbhelper.GetChainID()), transaction, pubKey.Bytes(), txHash, appStat.BeginBlock.Hash)

if result.Code == types2.CodeBVMQueryOK {

return types.ResponseCheckTx{

Code: types2.CodeBVMQueryOK,

Log: result.Log,

Data: result.Data}

} else if result.Code != types2.CodeOK {

app.logger.Error("CheckTx failed", "code", result.Code, "error", result.Log)

return types.ResponseCheckTx{

Code: result.Code,

Log: result.Log}

}

return types.ResponseCheckTx{

Code: types2.CodeOK,

Log: "CheckTx success"}

}

//GIBeginBlock beginblock implementation of app

func (app \*AppDeliver) GIBeginBlock(req abci.RequestBeginBlock) (abci.ResponseBeginBlock, map[string][]byte) {

transID := statedbhelper.NewTransactionID()

app.transID = transID

app.txID = 0

app.logger.Info("Recv ABCI interface: BeginBlock", "height", req.Header.Height, "transID", app.transID)

//Checking app state

app.getAndVerifyAppState(req.Header)

//Init app members

app.appState.BlockHeight = req.Header.Height

app.appState.BeginBlock = req

app.hashList = list.New().Init()

app.sponser = req.Header.ProposerAddress

app.rewarder = req.Header.RewardAddress

app.blockHash = req.Hash

app.blockHeader = req.Header

// Reset fee & rewards for the block

app.fee = 0

app.rewards = map[string]int64{}

app.rewardStrategy = statedbhelper.GetRewardStrategy(app.transID, app.txID, app.blockHeader.Height)

//statedbhelper.BeginBlock(transID)

//app.logger.Debug("SetAppState", "new appState", app.appState)

// Set the last app state due to SDK depends on it to check/get block data,

// update it when commit

//statedbhelper.SetWorldAppState(transID, app.txID, app.appState)

// call smcrunsvc to initChain or updateChain smart contract

r, txBuffer := app.initOrUpdateSMC()

if r.Code != types.CodeOK {

return abci.ResponseBeginBlock{Code: r.Code, Log: r.Log}, nil

}

return abci.ResponseBeginBlock{Code: types.CodeOK}, txBuffer

}

func (app \*AppDeliver) getAndVerifyAppState(blockHeader abci.Header) {

app.appState = statedbhelper.GetWorldAppState(app.transID, app.txID)

app.logger.Debug("WorldAppState",

"height", app.appState.BlockHeight,

"LastBlockHash", app.appState.AppHash)

// Checking on new block height

if blockHeader.Height != app.appState.BlockHeight+1 {

app.logger.Fatal("Block height does not match",

"abci\_height", app.appState.BlockHeight,

"block\_height", blockHeader.Height)

panic("Block height does not match")

}

// Checking on app hash

if !bytes.EqualFold(blockHeader.LastAppHash, app.appState.AppHash) {

app.logger.Fatal("App hash does not match",

"abci\_app\_hash", app.appState.AppHash,

"block\_last\_app\_hash", blockHeader.LastAppHash)

panic(fmt.Sprintf("App hash does not match, req.Header.LastAppHash %x:%d, app.appState.AppHash:%x:%d",

blockHeader.LastAppHash, blockHeader.Height, app.appState.AppHash, app.appState.BlockHeight))

}

}

// forbidContract - forbid contract if initChain/updateChain failed

func (app \*AppDeliver) forbidContract(contractAddr types.Address) {

contract := statedbhelper.GetContract(contractAddr)

contract.LoseHeight = app.blockHeader.Height

statedbhelper.SetContract(app.transID, app.txID, contract)

v, err := statedbhelper.Get(std.KeyOfMineContracts())

if err != nil {

panic(err)

}

if len(v) == 0 {

return

}

var mines []std.MineContract

err = jsoniter.Unmarshal(v, &mines)

if err != nil {

panic(err)

}

for index, mine := range mines {

if mine.Address == contractAddr {

mines = append(mines[:index], mines[index+1:]...)

break

}

}

statedbhelper.SetMineContract(app.transID, app.txID, mines)

}

//nolint unhandled

func (app \*AppDeliver) commit() abci.ResponseCommit {

app.logger.Info("Recv ABCI interface: Commit",

"height", app.appState.BlockHeight)

// For empty block, its apphash is exactly same to the last one.

if app.hashList.Len() == 0 {

return app.commitEmptyBlock()

}

//txHashList

hasherSHA3256 := sha3.New256()

hashListBytes := make([]crypto.Hash, 0)

for txHash := app.hashList.Front(); txHash != nil; txHash = txHash.Next() {

deliverHash := txHash.Value.([]byte)

hasherSHA3256.Write(deliverHash)

hashListBytes = append(hashListBytes, crypto.Hash(deliverHash))

app.logger.Info("Commit: txHash", "txHash", hex.EncodeToString(deliverHash))

}

hashListSha := hasherSHA3256.Sum(nil)

appHash := sha3.New256()

appHash.Write(app.appState.AppHash)

appHash.Write(hashListSha)

app.logger.Debug("666666 calc appHash", "lastAppHash", app.appState.AppHash, "hashListSha", hashListSha)

app.appState.AppHash = appHash.Sum(nil)

app.appState.TxsHashList = hashListBytes

app.logger.Info("commitTx", "appHash", app.appState.AppHash)

//rewards

keys := make([]string, 0)

for k := range app.rewards {

keys = append(keys, k)

}

sort.Strings(keys)

kvps := make([]common.KVPair, len(keys))

for i, k := range keys {

uintByte := make([]byte, 8)

binary.BigEndian.PutUint64(uintByte, uint64(app.rewards[k]))

kvps[i] = common.KVPair{Key: []byte(k), Value: uintByte}

}

app.appState.Rewards = kvps

app.appState.Fee = uint64(app.fee)

// SetWorldAppState and commit block

app.commitBlock()

//SDK commit

adapter.GetInstance().Commit(app.transID)

return abci.ResponseCommit{AppState: abci.AppStateToByte(app.appState)}

}

func (app \*AppDeliver) commitEmptyBlock() abci.ResponseCommit {

app.commitBlock()

// app hash of empty block is exactly same to the last one.

appState := &abci.AppState{

BlockHeight: app.appState.BlockHeight,

AppHash: app.appState.AppHash,

}

return abci.ResponseCommit{AppState: abci.AppStateToByte(appState)}

}

func (app \*AppDeliver) commitBlock() {

statedbhelper.SetWorldAppState(app.transID, app.txID, app.appState)

statedbhelper.CommitTx(app.transID, app.txID)

statedbhelper.CommitBlock(app.transID)

app.sponser = "" //清空提案者

app.rewarder = ""

app.logger.Info(" commit",

"height", app.appState.BlockHeight,

"app\_hash", hex.EncodeToString(app.appState.AppHash))

}

func (app \*AppDeliver) deliverBCTx(tx []byte) (resDeliverTx types.ResponseDeliverTx, txBuffer map[string][]byte) {

app.logger.Info("Recv ABCI interface: DeliverTx", "tx", string(tx))

if app.chainID == "" {

app.SetChainID(statedbhelper.GetChainID())

}

app.txID++

// for base58

tx2.Init(app.chainID)

transaction, pubKey, err := tx2.TxParse(string(tx))

if err != nil {

// for base64

tx3.Init(app.chainID)

transaction, pubKey, err = tx3.TxParse(string(tx))

if err != nil {

app.logger.Error("tx parse failed:", err)

return app.reportFailure(tx, types2.ErrDeliverTx, "tx parse failed"), nil

}

}

app.logger.Debug("DELIVER.TX", "height", app.blockHeader.Height, "tx", transaction.String(), "pubKey", pubKey, "addr", pubKey.Address(statedbhelper.GetChainID()))

return app.runDeliverTx(tx, transaction, pubKey)

}

func (app \*AppDeliver) runDeliverTx(tx []byte, transaction types2.Transaction, pubKey crypto.PubKeyEd25519) (resDeliverTx types.ResponseDeliverTx, txBuffer map[string][]byte) {

resDeliverTx.Code = types2.CodeOK

if len(transaction.Note) > types2.MaxSizeNote {

return app.reportFailure(tx, types2.ErrDeliverTx, "tx note is out of range"), nil

}

nonceBuffer, err := statedbhelper.SetAccountNonce(app.transID, app.txID, pubKey.Address(statedbhelper.GetChainID()), transaction.Nonce)

if err != nil {

app.logger.Error("SetAccountNonce failed:", err)

return app.reportFailure(tx, types2.ErrDeliverTx, "SetAccountNonce failed"), nil

}

txHash := common.HexBytes(algorithm.CalcCodeHash(string(tx)))

adp := adapter.GetInstance()

response := adp.InvokeTx(app.blockHeader, app.transID, app.txID, pubKey.Address(statedbhelper.GetChainID()), transaction, pubKey.Bytes(), txHash, app.blockHash)

response.Fee = gatherFees(response.Tags)

if response.Code != types2.CodeOK {

app.logger.Error("docker invoke error.....", "error", response.Log)

app.logger.Debug("docker invoke error.....", "response", response.String())

statedbhelper.RollbackTx(app.transID, app.txID)

adp.RollbackTx(app.transID, app.txID)

resDeliverTx, txBuffer = app.reportInvokeFailure(tx, transaction, response)

return resDeliverTx, combineBuffer(nonceBuffer, txBuffer)

}

app.logger.Debug("docker invoke response.....", "code", response.Code)

// pack validators if update validator info

if hasUpdateValidatorReceipt(response.Tags) {

app.packValidators()

}

// pack side chain genesis info

if t, ok := hasSideChainGenesisReceipt(response.Tags); ok {

app.packSideChainGenesis(t)

}

//emit new summary fee and transferFee receipts

tags := app.emitFeeReceipts(transaction, response, true)

resDeliverTx.Code = response.Code

resDeliverTx.Log = response.Log

resDeliverTx.Tags = tags

resDeliverTx.GasLimit = uint64(transaction.GasLimit)

resDeliverTx.GasUsed = uint64(response.GasUsed)

resDeliverTx.Fee = uint64(response.Fee)

resDeliverTx.Data = response.Data

resDeliverTxStr := resDeliverTx.String()

app.logger.Debug("deliverBCTx()", "resDeliverTx length", len(resDeliverTxStr), "resDeliverTx", resDeliverTxStr) // log value of async instance must be immutable to avoid data race

stateTx, txBuffer := statedbhelper.CommitTx(app.transID, app.txID)

app.calcDeliverHash(tx, &resDeliverTx, stateTx)

app.logger.Debug("deliverBCTx() ", "stateTx length", len(stateTx), "stateTx ", string(stateTx))

//calculate Fee

app.fee = app.fee + response.Fee

app.logger.Debug("deliverBCTx()", "app.fee", app.fee, "app.rewards", map2String(app.rewards))

app.logger.Debug("end deliver invoke.....")

return resDeliverTx, combineBuffer(nonceBuffer, txBuffer)

}

//nolint unhandled

func (app \*AppDeliver) calcDeliverHash(tx []byte, response \*types.ResponseDeliverTx, stateTx []byte) {

md5TX := md5.New()

if tx != nil {

md5TX.Write(tx)

}

app.logger.Debug("deliverHash", "resp", response.String(), "stateTx", string(stateTx))

if response != nil {

md5TX.Write([]byte(response.String()))

}

if stateTx != nil {

md5TX.Write(stateTx)

}

if tx != nil || response != nil || stateTx != nil {

deliverHash := md5TX.Sum(nil)

app.hashList.PushBack(deliverHash)

}

}

func (app \*AppDeliver) reportFailure(tx []byte, errorCode uint32, msg string) (response types.ResponseDeliverTx) {

response.Code = errorCode

response.Log = msg

app.calcDeliverHash(tx, &response, nil)

return

}

func (app \*AppDeliver) reportInvokeFailure(tx []byte, transaction types2.Transaction, response \*types2.Response) (resDeliverTx types.ResponseDeliverTx, txBuffer map[string][]byte) {

rcpts := app.emitFeeReceipts(transaction, response, false)

resDeliverTx = types.ResponseDeliverTx{

Code: response.Code,

Log: response.Log,

GasLimit: uint64(transaction.GasLimit),

GasUsed: uint64(response.GasUsed),

Fee: uint64(response.Fee),

Tags: rcpts,

}

//commit transactions of fee

var stateTx []byte

if len(rcpts) > 0 {

stateTx, txBuffer = statedbhelper.CommitTx(app.transID, app.txID)

}

app.calcDeliverHash(tx, &resDeliverTx, stateTx)

return

}

func mapFee2String(m map[types2.Address]std.Fee) string {

b := new(bytes.Buffer)

b.WriteString("{")

for key, value := range m {

\_, \_ = fmt.Fprintf(b, "%s:'%s',", key, value.String())

}

b.WriteString("}")

return b.String()

}

func map2String(m map[types2.Address]int64) string {

b := new(bytes.Buffer)

b.WriteString("{")

for key, value := range m {

\_, \_ = fmt.Fprintf(b, "%s:%d,", key, value)

}

b.WriteString("}")

return b.String()

}

func (app \*AppDeliver) emitFeeReceipts(transaction types2.Transaction, response \*types2.Response, isDlvOK bool) (tags []common.KVPair) {

fees, feetags := gatherFeesByFromAddr(response.Tags, isDlvOK)

app.logger.Debug("get fee receipts", "receipts", mapFee2String(fees))

totalFeeReceipts, err := emitTotalFeeReceipt(fees)

if err != nil {

app.logger.Error("emit fee receipt failed", "error", err.Error())

return nil

}

// if transaction was succeed, save response tags

if isDlvOK {

tags = response.Tags

} else {

//fill original fee receipts in deliver response even it's failed.

tags = feetags

}

//nolint

keys := make([]types2.Address, 0)

for k := range totalFeeReceipts {

keys = append(keys, k)

}

sort.Strings(keys)

methodID := transaction.Messages[len(transaction.Messages)-1].MethodID // todo 级联调用需要检查手续费，或者测试两种合约能否级联

isBVM := methodID == 0 || methodID == 0xFFFFFFFF

for \_, k := range keys {

transRcpt, \_ := app.distributeFee(fees[k], app.rewarder, isDlvOK, isBVM)

kv := common.KVPair{

Key: []byte(fmt.Sprintf("/%d/0/totalFee", len(transaction.Messages))),

Value: totalFeeReceipts[k],

}

tags = append(tags, kv)

for index, r := range transRcpt {

kv := common.KVPair{

Key: []byte(fmt.Sprintf("/%d/%d/transferFee", len(transaction.Messages), index+1)),

Value: r,

}

tags = append(tags, kv)

}

}

return tags

}

//distributeFee transfer fee from sender's balance to rewards address, and emit transferFee receipt accordingly

func (app \*AppDeliver) distributeFee(fee std.Fee, proposerReward types2.Address, isDlvOK, isBVM bool) (receipts [][]byte, bcerr types2.BcError) {

if !isDlvOK || isBVM {

app.logger.Debug("DeliverTx was failed, pay fee from sender")

//deliverTx fails, transaction be rollback, set sender's balance for Fee

v := statedbhelper.BalanceOf(app.transID, app.txID, fee.From, fee.Token).SubI(fee.Value)

statedbhelper.SetBalance(app.transID, app.txID, fee.From, fee.Token, v)

}

// Set rewards balance

leftFee := fee.Value

for i, reward := range app.rewardStrategy {

app.logger.Debug("reward strategy", "reward", reward.String())

addr := reward.Address

//revard name "validators" writes into genesis file, cannot be modified.

if reward.Address == "" && reward.Name == "validators" {

addr = proposerReward

}

percent, err := strconv.ParseFloat(reward.RewardPercent, 64)

if err != nil {

bcerr.ErrorCode = types2.ErrDeliverTx

bcerr.ErrorDesc = err.Error()

app.logger.Error("Get reward percent failed", "error", err)

return

}

// Using Number in case it's overflow by multiple 10000

award := bn.N(fee.Value).MulI(int64(percent \* 100)).DivI(10000)

//the last reward, set all left Fee to him.

if i == len(app.rewardStrategy) {

award = bn.N(leftFee)

}

app.logger.Debug("The rewards of Fee", "reward", addr, "percent", percent, "value", award)

v := statedbhelper.BalanceOf(app.transID, app.txID, addr, fee.Token)

v = v.Add(award)

statedbhelper.SetBalance(app.transID, app.txID, addr, fee.Token, v)

if softforks.V2\_0\_1\_13780(app.blockHeader.Height) {

// 没有将资产信息添加到账户资产列表

} else {

statedbhelper.AddAccountToken(app.transID, app.txID, addr, fee.Token)

}

app.logger.Debug("The reward's balance", "reward", addr, "balance", v)

r := emitTransferReceipt(fee.From, addr, fee.Token, award)

receipts = append(receipts, r)

ia, \_ := strconv.ParseInt(award.String(), 10, 64)

app.rewards[addr] = app.rewards[addr] + ia

leftFee = leftFee - ia

}

bcerr.ErrorCode = types2.CodeOK

return

}

// hasUpdateValidatorReceipt check if there is a receipt for updating validator

func hasUpdateValidatorReceipt(tags []common.KVPair) bool {

isUpdtValidators := false

for \_, r := range tags {

if strings.HasSuffix(string(r.Key), "governance.newValidator") ||

strings.HasSuffix(string(r.Key), "governance.setPower") {

//if strings.Contains(string(r.Key), "governance") {

isUpdtValidators = true

}

}

return isUpdtValidators

}

// packValidators pack validators info when governance contract update validator info

func (app \*AppDeliver) packValidators() {

app.udValidator = true

var tempVal []types.Validator

validators := statedbhelper.GetAllValidators(app.transID, app.txID)

for \_, validator := range validators {

pkBytes := crypto.PubKeyEd25519FromBytes(validator.PubKey).Bytes()

val := types.Validator{

PubKey: pkBytes,

Power: uint64(validator.Power),

RewardAddr: validator.RewardAddr,

Name: validator.Name,

}

tempVal = append(tempVal, val)

}

app.validators = tempVal

app.logger.Debug("deliverBCTx() update validators", "validators", app.validators)

}

func (app \*AppDeliver) packSideChainGenesis(tag common.KVPair) {

type Validator struct {

PubKey types3.PubKey `json:"nodepubkey,omitempty"` //节点公钥

Power int64 `json:"power,omitempty"` //节点记账权重

RewardAddr string `json:"reward\_addr,omitempty"` //节点接收奖励的地址

Name string `json:"name,omitempty"` //节点名称

NodeAddr string `json:"nodeaddr,omitempty"` //节点地址

}

type ContractData struct {

Name string `json:"name"`

Version string `json:"version"`

CodeByte types3.HexBytes `json:"codeByte"`

}

type genesisSideChain struct {

SideChainID string `json:"sideChainID"`

OpenURLs []string `json:"openURLs"`

GenesisInfo string `json:"genesisInfo"`

ContractData []ContractData `json:"contractData"`

}

type genesisInfo struct {

Validators []Validator `json:"validators"`

}

var r std.Receipt

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

if err != nil {

panic(err)

}

app.logger.Info("侧链创世收据：", r.Name)

gsc := new(genesisSideChain)

if err = jsoniter.Unmarshal(r.Bytes, gsc); err != nil {

panic(err)

}

app.logger.Info("侧链registerSideChain sideChainID：", gsc.SideChainID)

app.logger.Info("侧链registerSideChain OpenURLs：", gsc.OpenURLs)

conDatas := make([]types.ContractData, len(gsc.ContractData))

for i, v := range gsc.ContractData {

conDatas[i] = types.ContractData{

Name: v.Name,

Version: v.Version,

CodeData: v.CodeByte,

}

}

gi := new(genesisInfo)

if err = jsoniter.Unmarshal(bytes.NewBufferString(gsc.GenesisInfo).Bytes(), gi); err != nil {

panic(err)

}

vals := make([]types.Validator, len(gi.Validators))

for i, v := range gi.Validators {

vals[i] = types.Validator{

PubKey: v.PubKey,

Power: uint64(v.Power),

RewardAddr: v.RewardAddr,

Name: v.Name,

}

}

scg := &types.SideChainGenesis{

SideChainID: gsc.SideChainID,

GenesisInfo: gsc.GenesisInfo,

ContractData: conDatas,

Validators: vals,

}

app.logger.Info("侧链创世返回给 tendermint 的 validator ：", scg.Validators)

app.scGenesis = []\*types.SideChainGenesis{scg}

}

//gatherFeesByFromAddr loop up tags of receipts and find out Fee receipts for each "From" address,

// and return fees' receipts also

func gatherFeesByFromAddr(tags []common.KVPair, isDlvOK bool) (fees map[types2.Address]std.Fee, feeTags []common.KVPair) {

fees = make(map[types2.Address]std.Fee)

feeTags = make([]common.KVPair, 0)

for \_, t := range tags {

if strings.Contains(string(t.Key), "std::fee") {

receipt := std.Receipt{}

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

if err != nil {

panic(err)

}

rf := std.Fee{}

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

if err != nil {

panic(err)

}

if v, ok := fees[rf.From]; ok {

rf.Value = v.Value + rf.Value

}

fees[rf.From] = rf

if !isDlvOK {

//fill original fee receipt in

feeTags = append(feeTags, t)

}

}

}

return

}

//emitTotalFeeReceipt generate new total Fee receipt for each of sender

func emitTotalFeeReceipt(fees map[types2.Address]std.Fee) (map[types2.Address][]byte, error) {

bbrs := make(map[types2.Address][]byte)

for \_, fee := range fees {

rbyte, err := jsoniter.Marshal(fee)

if err != nil {

return nil, err

}

receipt := types2.Receipt{

Name: "totalFee",

ReceiptBytes: rbyte,

ReceiptHash: nil,

}

receipt.ReceiptHash = sha3.Sum256([]byte(receipt.Name), rbyte)

br, err := jsoniter.Marshal(receipt)

if err != nil {

return nil, err

}

bbrs[fee.From] = br

}

return bbrs, nil

}

func emitTransferReceipt(sender, to, tokenAddr types2.Address, value bn.Number) []byte {

trans := std.Transfer{

Token: tokenAddr,

From: sender,

To: to,

Value: value,

}

bz, err := jsoniter.Marshal(trans)

if err != nil {

return nil

}

receipt := types2.Receipt{

Name: "transferFee",

ReceiptBytes: bz,

ReceiptHash: nil,

}

receipt.ReceiptHash = sha3.Sum256([]byte(receipt.Name), bz)

bz, err = jsoniter.Marshal(receipt)

if err != nil {

return nil

}

return bz

}

func combineBuffer(nonceBuffer, txBuffer map[string][]byte) map[string][]byte {

if txBuffer == nil {

txBuffer = make(map[string][]byte)

}

for k, v := range nonceBuffer {

txBuffer[k] = v

}

return txBuffer

}

//call mine method

func (app \*AppDeliver) mine() (result \*types.Response, txBuffer map[string][]byte) {

app.logger.Debug("mine")

result = new(types.Response)

result.Code = types.CodeOK

mineContracts := statedbhelper.GetMineContract(app.transID, app.txID)

if len(mineContracts) == 0 {

app.logger.Debug("mine contracts is not exist in stateDB")

return

}

for \_, v := range mineContracts {

contract := statedbhelper.GetContract(v.Address)

if contract == nil {

result.Code = types.ErrLogicError

result.Log = "can not get smart contract to call mine method when begin block"

return

}

if contract.ChainVersion == 2 {

mgr := adapter.GetInstance()

if v.MineHeight <= app.appState.BlockHeight {

app.txID++

result = mgr.Mine(app.transID, app.txID, app.blockHeader, v.Address, contract.Owner)

if result.Code != types.CodeOK {

app.logger.Info(fmt.Sprintf("[transID=%d][txID=%d]call mine method failed", app.transID, app.txID), "error", result.Log)

result.Code = types.CodeOK

}

app.logger.Info("MiningTrans", "RewardAddress", app.blockHeader.RewardAddress, "bal", result.Data)

var stateTx []byte

stateTx, txBuffer = statedbhelper.CommitTx(app.transID, app.txID)

if stateTx != nil {

app.calcDeliverHash(nil, nil, stateTx)

}

}

}

}

return

}

func (conn \*QueryConnection) BCInfo(req abci.RequestInfo) (resInfo abci.ResponseInfo) {

appState := statedbhelper.GetWorldAppState(0, 0)

if appState == nil {

conn.logger.Info("first time to init chain and get stateDB BCINFO")

respAppState := abci.AppState{BlockHeight: 0}

return abci.ResponseInfo{

Version: version.Version,

LastBlockHeight: 0,

LastAppState: abci.AppStateToByte(&respAppState),

}

}

if req.Port != "" {

TmCoreURL = "http://" + req.Host + ":" + req.Port

}

//BCInfo是gichain每次启动后，第一个被调用的函数，在此对chainID进行设置

chainID := statedbhelper.GetChainID()

crypto.SetChainId(chainID)

return abci.ResponseInfo{

Version: version.Version,

LastBlockHeight: appState.BlockHeight,

LastAppState: abci.AppStateToByte(appState),

}

}

func (conn \*QueryConnection) query(req types.RequestQuery) (resQuery types.ResponseQuery) {

var query bctypes.Query

if len(req.Data) != 0 {

chainID := statedbhelper.GetChainID()

addrStr, query2, err := tx2.QueryDataParse(chainID, string(req.Data))

if err != nil {

conn.logger.Error("QueryDataParse parse failed:", err)

return types.ResponseQuery{

Code: bctypes.ErrLogicError,

Log: err.Error(),

}

}

query = query2

if strings.HasPrefix(query.QueryKey, "/account") { //如果key包含了签名的地址才让查询

if !strings.Contains(query.QueryKey, addrStr) {

conn.logger.Error("Query only can query itself,but get other address")

bcerr := bctypes.BcError{

ErrorCode: bctypes.ErrNoAuthorization,

ErrorDesc: "",

}

return types.ResponseQuery{

Code: bcerr.ErrorCode,

Log: bcerr.Error(),

}

}

}

} else if req.Path != "" {

query.QueryKey = req.Path

}

conn.logger.Debug("key info:", "key:", req.Path)

var kBytes []byte

kBytes, err := statedbhelper.Get(query.QueryKey)

if err != nil {

conn.logger.Fatal("query DB failed ", "error", err)

panic(err)

}

conn.logger.Debug("value info:", "value byte length:", len(kBytes))

return types.ResponseQuery{

Code: types.CodeTypeOK,

Key: []byte(req.Path),

Value: kBytes,

}

}

func (conn \*QueryConnection) queryEx(req types.RequestQueryEx) (resQuery types.ResponseQueryEx) {

var query bctypes.Query

if req.Path != "" {

query.QueryKey = req.Path

}

conn.logger.Debug("key info:", "key:", req.Path)

//提取字符串

keys, err := ResolvePath(query.QueryKey)

if err != nil {

return types.ResponseQueryEx{

Code: bctypes.ErrPath,

Log: err.Error(),

}

}

//var kBytes []byte

kv := make([]types.KeyValue, len(keys))

for i, v := range keys {

kBytes, err := statedbhelper.Get(v)

if err != nil {

conn.logger.Fatal("query DB failed ", "error", err)

panic(err)

}

conn.logger.Debug("value info:", "value byte length:", len(kBytes))

kv[i].Key = []byte(v)

kv[i].Value = kBytes

}

return types.ResponseQueryEx{

Code: types.CodeTypeOK,

KeyValues: kv,

}

}

func ResolvePath(path string) ([]string, error) {

keys := make([]string, 0, 0)

//判断url的有效性

if !strings.HasPrefix(path, "/") {

return nil, fmt.Errorf("path does not start with '/'")

}

if strings.HasSuffix(path, "/") {

return nil, fmt.Errorf("path cannot end with '/'")

}

if !strings.Contains(path, "/[") {

return nil, fmt.Errorf("path does not contain '/['")

}

if strings.Index(path, "/[") > strings.LastIndex(path, "]") {

return nil, fmt.Errorf("']' is not on the right of '/['")

}

//解析url

subPath := path[strings.Index(path, "/[")+2 : strings.LastIndex(path, "]")]

elements := strings.Split(subPath, ",")

for \_, v := range elements {

keys = append(keys, fmt.Sprint(path[:strings.Index(path, "/[")+1], v, path[strings.LastIndex(path, "]")+1:]))

}

return keys, nil

}

//GetContractInvokeURL get contract invoke URL

func (sd \*SMCDocker) GetContractInvokeURL(transID, txID int64, contractAddr types.Address) (string, string, error) {

sd.logger.Trace("smcdocker GetContractInvokeURL", "transID", transID, "contract", contractAddr)

//根据合约地址，查询组织ID

var orgID, dockerName string

if contractAddr == smcbuilder.ThirdPartyContract {

orgID = smcbuilder.ThirdPartyContract

dockerName = orgID

} else if contractAddr == std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

dockerName = "genesis"

} else {

split := strings.Split(contractAddr, ".")

if len(split) == 2 {

orgID = split[0]

contractAddr = split[1]

} else {

orgID = statedbhelper.GetOrgID(transID, txID, contractAddr)

}

sd.logger.Debug("smcdocker GetContractInvokeURL", "transID", transID, "orgID", orgID, "contract", contractAddr)

if orgID == "" {

return "", "", errors.New("invalid contractAddr: " + contractAddr)

}

orgCodeHash := statedbhelper.GetOrgCodeHash(transID, txID, orgID)

genesisOrgID := statedbhelper.GetGenesisOrgID(transID, txID)

dockerName = orgID

var genesisOrgHashStr string

if orgID != genesisOrgID {

genesisOrgHashStr = string(statedbhelper.GetOrgCodeHash(transID, txID, genesisOrgID))

}

smcsvcFilePath := algorithm.CalcCodeHash(genesisOrgHashStr + string(orgCodeHash))

smcsvcFilePathStr := hex.EncodeToString(smcsvcFilePath)

targetBin := filepath.Join(smcbuilder.GetInstance().WorkDir, "bin", orgID, smcsvcFilePathStr, "smcrunsvc")

\_, err := os.Stat(targetBin)

if err != nil {

sd.logger.Debug("genesis org contract has updated, rebuild current org's contracts")

if v, ok := sd.orgNameToURL.Load(orgID); ok {

fc(v.(string))

}

sd.orgNameToURL.Delete(orgID)

}

}

v, ok := sd.orgNameToURL.Load(orgID)

if ok {

url := v.(string)

sd.orgIdToLastTime.Store(orgID, time.Now())

sd.logger.Debug("smcdocker GetContractInvokeURL map exist ", "transID", transID, "url", url)

return contractAddr, url, nil

} else {

sd.logger.Debug("smcdocker GetContractInvokeURL map not exist",

"transID", transID, "orgID", orgID, "contract", contractAddr)

c := make(chan RunDockerRes)

sd.RunDocker <- RunDocker{

TransID: transID,

TxID: txID,

ContractAddr: contractAddr,

OrgID: orgID,

DockerName: dockerName,

c: c,

}

res := <-c

if res.Error != "" {

return "", "", errors.New(res.Error)

}

return contractAddr, res.Url, nil

}

}

//DirtyContractInvokeURL dirty ContractInvokeURL, next invoke the contract will build a new image

func (sd \*SMCDocker) DirtyContractInvokeURL(transID, txID int64, contractAddr types.Address) {

orgID := ""

if contractAddr == std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

blh := helper.BlockChainHelper{}

orgID = blh.CalcOrgID("genesis")

} else {

orgID = statedbhelper.GetOrgID(transID, txID, contractAddr)

}

if v, ok := sd.orgNameToURL.Load(orgID); ok {

fc(v.(string))

}

sd.orgNameToURL.Delete(orgID)

d := dockerlib.GetDockerLib()

sd.logger.Debug("DirtyContractInvokeURL", "orgID", orgID)

isKilled := d.Kill(orgID)

sd.logger.Debug("DirtyContractInvokeURL", "orgID", orgID, "killResult", isKilled)

if !isKilled {

panic(fmt.Sprintf("kill docker for %v fail!", orgID))

}

}

// DirtyAllURL dirty all containers URL and kill all containers

func (sd \*SMCDocker) DirtyAllURL() {

sd.orgNameToURL.Range(func(key, value interface{}) bool {

k := key.(string)

sd.orgNameToURL.Delete(k)

d := dockerlib.GetDockerLib()

sd.logger.Debug("DirtyAllURL", "orgID", k)

isKilled := d.Kill(k)

sd.logger.Debug("DirtyAllURL", "orgID", k, "killResult", isKilled)

if !isKilled {

panic(fmt.Sprintf("kill docker for %v fail!", k))

}

return true

})

}

// CheckDockerLiveTime 检查 docker 上一次发生交易的时间，超过一定时间就杀掉。

func (sd \*SMCDocker) CheckDockerLiveTime() {

sd.orgIdToLastTime.Range(func(key, value interface{}) bool {

k := key.(string)

v := value.(time.Time)

timeout := common.GlobalConfig.ContainerTimeout

if timeout == 0 {

timeout = 30

}

if time.Since(v) > time.Duration(timeout\*int64(time.Minute)) {

v, ok := sd.orgNameToURL.Load(k)

if ok {

d := dockerlib.GetDockerLib()

sd.logger.Debug("CheckDockerLiveTime kill", "orgID", k)

isKilled := d.Kill(k)

sd.logger.Debug("CheckDockerLiveTime kill", "orgID", k, "killResult", isKilled)

if !isKilled {

panic("")

}

fc(v.(string))

sd.orgNameToURL.Delete(k)

sd.orgIdToLastTime.Delete(k)

}

}

return true

})

}

func (sd \*SMCDocker) runDockerSever() {

var startingDocker sync.Map // orgID => url

imageName := "alpine:latest"

for {

rd := <-sd.RunDocker

var v []chan RunDockerRes

if value, ok := startingDocker.Load(rd.OrgID); ok {

v = value.([]chan RunDockerRes)

v = append(v, rd.c)

startingDocker.Store(rd.OrgID, v)

continue

}

v = append(v, rd.c)

startingDocker.Store(rd.OrgID, v)

go func(rd RunDocker) {

sd.logger.Debug("smcdocker GetContractInvokeURL map not exist,begin builder.GetContractDllPath ", "transID", rd.TransID)

portStr := strconv.Itoa(int(nu.GetIdlePort()))

runParam := dockerlib.DockerRunParams{}

builder := smcbuilder.GetInstance()

callBackUrl := sd.callbackURL

if rd.OrgID == smcbuilder.ThirdPartyContract {

callBackUrl = strings.Replace(callBackUrl, "32333", "32332", 1)

}

dllPath, err := builder.GetContractDllPath(rd.TransID, rd.TxID, rd.OrgID)

if err != nil {

if value, ok := startingDocker.Load(rd.OrgID); ok {

res := value.([]chan RunDockerRes)

for \_, v := range res {

v <- RunDockerRes{

Url: "",

Error: err.Error(),

}

}

startingDocker.Delete(rd.OrgID)

}

return

}

sd.logger.Debug("Contract dll path:" + dllPath)

if runtime.GOOS == "windows" {

runParam.Cmd = []string{

".\\smcrunsvc.exe",

"start",

"-p",

portStr,

"-c",

callBackUrl,

}

dllPath = strings.Replace(dllPath, "\\smcrunsvc.exe", "", 1)

runParam.WorkDir = dllPath

} else {

logPath := filepath.Join(builder.WorkDir, "log", rd.DockerName)

err := os.MkdirAll(logPath, 0750)

if err != nil {

startingDocker.Delete(rd.OrgID)

startingDocker.Range(func(key, value interface{}) bool {

v := value.(chan RunDockerRes)

v <- RunDockerRes{

Url: "",

Error: err.Error(),

}

return true

})

return

}

runParam.Cmd = []string{

"/smcrunsvc",

"start",

"-p",

portStr,

"-c",

callBackUrl,

}

workDirDocker := "/log/" + rd.DockerName

runParam.WorkDir = workDirDocker

runParam.AutoRemove = false

runParam.Mounts = []dockerlib.Mounts{

{

Source: dllPath,

Destination: "/smcrunsvc",

},

{

Source: builder.WorkDir + "/log",

Destination: "/log",

},

}

hostPort := dockerlib.HostPort{Port: portStr, Host: "0.0.0.0"}

runParam.PortMap = make(map[string]dockerlib.HostPort)

runParam.PortMap[portStr+"/tcp"] = hostPort

}

d := dockerlib.GetDockerLib()

sd.logger.Debug("runDockerSever kill", "killOrgID", rd.DockerName)

if v, ok := sd.orgNameToURL.Load(rd.OrgID); ok {

fc(v.(string))

}

isKill := d.Kill(rd.DockerName)

sd.logger.Debug("runDockerSever kill", "killOrgID", rd.DockerName, "killResult", isKill)

if !isKill {

panic(fmt.Sprintf("kill docker for %v fail!", rd.DockerName))

}

sd.logger.Debug("Run docker", "imageName", imageName, "orgID", rd.OrgID, "param", runParam)

ok, err := d.Run(imageName, rd.DockerName, &runParam)

sd.logger.Debug("Run docker result", "orgID", rd.DockerName, "result", ok)

if !ok {

if value, ok := startingDocker.Load(rd.OrgID); ok {

res := value.([]chan RunDockerRes)

for \_, v := range res {

v <- RunDockerRes{

Url: "",

Error: err.Error(),

}

}

startingDocker.Delete(rd.OrgID)

}

return

}

dockerURL := "tcp://" + d.GetDockerContainerIP(rd.DockerName) + ":" + portStr

sd.logger.Debug("waitSmcRunSvcReady begin", "dockerURL", dockerURL)

if waitSmcRunSvcReady(dockerURL, sd.logger) {

sd.logger.Info("smcdocker GetContractInvokeURL run docker ok ", "transID", rd.TransID, "URL", dockerURL)

} else {

panic(fmt.Sprintf("smcdocker GetContractInvokeURL run docker for %v failed", rd.DockerName))

}

sd.logger.Debug("put url to orgNameToURL map", "dockerURL", dockerURL)

sd.orgNameToURL.Store(rd.DockerName, dockerURL)

sd.orgIdToLastTime.Store(rd.DockerName, time.Now())

sd.logger.Debug("write response to run docker channel", "orgID", rd.OrgID, "dockerURL", dockerURL)

if value, ok := startingDocker.Load(rd.OrgID); ok {

res := value.([]chan RunDockerRes)

for \_, v := range res {

v <- RunDockerRes{

Url: dockerURL,

Error: "",

}

}

startingDocker.Delete(rd.OrgID)

}

}(rd)

}

}

func waitSmcRunSvcReady(url string, logger log.Logger) bool {

logger.Trace("WaitSmcSvcReady()", "url", url)

beginTime := time.Now()

timeOut := 10.00

for {

time.Sleep(time.Duration(time.Millisecond \* 500))

n := time.Now()

sub := n.Sub(beginTime).Seconds()

if sub > timeOut {

logger.Error("WaitSmcSvcReady() timed out")

return false

}

cli, err := socket.NewClient(url, true, logger)

if err != nil {

continue

}

value, err := cli.Call("Health", map[string]interface{}{"transID": 0}, 10)

if err != nil || value == nil {

continue

}

if value.(string) == "health" {

logger.Debug("WaitSmcSvcReady()", "url", url, "checkHealth", value.(string))

SetDockerLogLevel(url, common.GlobalConfig.LogLevel, logger)

InitDockerSoftForks(url, logger)

return true

} else {

panic("connect to smcrunsvc failed")

}

}

}

func SetDockerLogLevel(url, level string, logger log.Logger) {

cli, err := socket.NewClient(url, true, logger)

if err != nil {

panic(err)

}

value, err := cli.Call("SetLogLevel", map[string]interface{}{"level": level}, 10)

if err != nil {

panic(err)

}

if !value.(bool) {

panic("can not set docker log level")

}

}

func InitDockerSoftForks(url string, logger log.Logger) {

forksBytes, err := jsoniter.Marshal(softforks.TagToForkInfo)

if err != nil {

panic(err)

}

cli, err := socket.NewClient(url, true, logger)

if err != nil {

panic(err)

}

value, err := cli.Call("InitSoftForks", map[string]interface{}{"softforks": string(forksBytes)}, 10)

if err != nil {

panic(err)

}

if !value.(bool) {

panic("can not init docker soft forks")

}

}

// DirtyURL - when the docker closed, then dirty url from map

func (im \*InvokerMgr) DirtyURL(url string) {

im.dockerMapConnPool.Delete(url)

im.dockerUrlMap.Range(func(key, value interface{}) bool {

urlMap := value.(\*UrlMap)

delete(urlMap.Map, url)

return true

})

im.transMap.Range(func(key, value interface{}) bool {

transMap := value.(\*TxID2UrlMap)

for txID, urls := range transMap.Map {

for i, v := range urls {

if v == url {

urls = append(urls[:i], urls[i+1:]...)

}

}

transMap.Map[txID] = urls

}

return true

})

}

// CallMcDirtyTx - dirty tx if it failed

func (im \*InvokerMgr) CallMcDirtyTx(urls []string, transId, txId int64) {

for \_, url := range urls {

im.logger.Debug(url)

pool := im.dockerConnPool(transId, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

result, err := cli.Call("McDirtyTransTx", map[string]interface{}{"transID": transId, "txID": txId}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

if !result.(bool) {

panic("CallMcDirtyTx result is false")

}

}

}

// InvokeTx - invoke tx's message one by one

func (im \*InvokerMgr) InvokeTx(

blockHeader types2.Header,

transId, txId int64,

sender types.Address,

tx types.Transaction,

pubKey types.PubKey,

txHash types.Hash,

blockHash types.Hash) (result \*types.Response) {

//从tx中解析出多个Message，InvokeMessage

receipts := make([]common.KVPair, 0)

result = new(types.Response)

var gasUsed, fee int64

var url string

var err types.BcError

urls := make([]string, 0, len(tx.Messages))

addrOfNewContract := make([]string, 0, len(tx.Messages))

for index, message := range tx.Messages {

payer, e := im.getPayer(blockHeader.Height, sender, tx, index)

if e != nil {

result.Code = types.ErrLogicError

result.Log = e.Error()

return

}

url, result, err = im.invoke(blockHeader, transId, txId, tx.GasLimit-gasUsed, sender, payer, tx, message, result.Tags, pubKey, txHash, blockHash)

// 无论失败与成功，均将收据和Fee等数据返回给调用者

// 调用者是 checker将会把无用数据丢弃， deliver根据手续费收据从发送者账户扣除手续费

if url != "" {

urls = append(urls, url)

}

// 在收据前部加上message序号

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

tag.Key = []byte(fmt.Sprintf("/%d%s", index, tag.Key))

receipts = append(receipts, tag)

}

gasUsed = result.GasUsed

fee = fee + result.Fee

if result.Code != types.CodeOK || err.ErrorCode != types.CodeOK {

//调用rpc清除该tx影响的Message的缓存

if len(urls) != 0 {

im.CallMcDirtyTx(urls, transId, txId)

}

break //跳出，不执行级联交易的下一条

}

newContractAddr := im.cleanDockerCache(result)

if newContractAddr != "" {

addrOfNewContract = append(addrOfNewContract, newContractAddr)

}

}

result.GasLimit = tx.GasLimit

result.Fee = fee

result.GasUsed = gasUsed

result.Tags = receipts

// 记录调用过的url

im.setValToTransMap(transId, txId, urls)

if len(addrOfNewContract) > 0 {

im.setValToTransCon(transId, txId, addrOfNewContract)

}

// if height in [23706999, forkHeight] then reset gas\_used

if softforks.V2\_0\_2\_14654(blockHeader.Height) {

im.resetGasUsed(blockHeader.Height, result, tx)

}

return

}

// TransferID - methodID of standard transfer method

var TransferID = algorithm.BytesToUint32(algorithm.CalcMethodId("Transfer(types.Address,bn.Number)"))

// invoke - invoke message in tx

func (im \*InvokerMgr) invoke(

blockHeader types2.Header,

transId, txId, gasLeft int64,

sender, payer types.Address,

tx types.Transaction,

message types.Message,

receipts []common.KVPair,

pubKey types.PubKey,

txHash types.Hash,

blockHash types.Hash) (url string, result \*types.Response, error types.BcError) {

error.ErrorCode = types.CodeOK

tx.Messages = nil

result = &types.Response{}

var to types.Address

//进行rpc调用

contractAddr, url, err := smcdocker.GetInstance().GetContractInvokeURL(transId, txId, message.Contract)

if err != nil {

error.ErrorCode = types.ErrInternalFailed

error.ErrorDesc = err.Error()

result.Code = types.ErrInternalFailed

result.Log = err.Error()

im.logger.Error("GetContractInvokeURL()", "error", err.Error())

return

}

if Debug {

url = "tcp://localhost:8080"

}

smcdocker.InitDockerSoftForks(url, im.logger)

message.Contract = contractAddr

// 创世时不需要执行如下代码

if message.Contract != std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

contract, e := im.getEffectContract(transId, txId, blockHeader.Height, message.Contract, message.MethodID)

if e.ErrorCode != types.CodeOK {

error.ErrorCode = e.ErrorCode

error.ErrorDesc = e.ErrorDesc

result.Code = e.ErrorCode

result.Log = e.ErrorDesc

return

}

message.Contract = contract.Address

}

//构造rpc参数

invokeParam := types.RPCInvokeCallParam{

Sender: sender,

Payer: payer,

To: to,

Tx: tx,

GasLeft: gasLeft,

TxHash: txHash,

Message: message,

Receipts: receipts,

SenderPublicKey: pubKey}

if softforks.V2\_0\_1\_13780(blockHeader.Height) {

invokeParam.BlockHash = nil

} else {

invokeParam.BlockHash = blockHash

}

im.logger.Debug("GetContractInvokeURL", "url", url)

im.logger.Trace("rpcCallInvoke", "invokeParamData", invokeParam)

pool := im.dockerConnPool(transId, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

defer pool.ReleaseClient(cli)

timeout := time.Duration(160)

if message.Contract == std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

timeout = 300

}

resp, err := cli.Call("Invoke", map[string]interface{}{"blockHeader": blockHeader, "transID": transId, "txID": txId, "callParam": invokeParam}, timeout)

if err != nil {

im.logger.Info("Client call error: " + err.Error())

// 之前有在失败时再重试一次，现在改为失败了直接 panic

panic(err)

}

result = new(types.Response)

err = jsoniter.Unmarshal([]byte(resp.(string)), result)

if err != nil {

panic(err)

}

if message.Contract == std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

smcdocker.GetInstance().DirtyContractInvokeURL(0, 0, message.Contract)

}

im.logger.Debug("rpcCallInvoke", "returned code", result.Code)

return

}

// Rollback - rollback transaction's data when it failed

func (im \*InvokerMgr) Rollback(transID int64) {

//依次获取url，进行rollback

if v, ok := im.dockerUrlMap.Load(transID); ok {

urlMap := v.(\*UrlMap).Map

for url := range urlMap {

pool := im.dockerConnPool(transID, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

\_, err = cli.Call("McDirtyTrans", map[string]interface{}{"transID": transID}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

}

}

im.dockerUrlMap.Delete(transID)

im.transMap.Delete(transID)

im.transIDToContractAddr.Delete(transID)

}

// RollbackTx - rollback tx's data when it failed

func (im \*InvokerMgr) RollbackTx(transID, txID int64) {

v1, ok := im.transMap.Load(transID)

if !ok {

return

}

vEx := v1.(\*TxID2UrlMap).Map

urls, ok := vEx[txID]

if !ok {

return

}

for \_, url := range urls {

pool := im.dockerConnPool(transID, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

\_, err = cli.Call("McDirtyTransTx", map[string]interface{}{"transID": transID, "txID": txID}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

}

delete(vEx, txID)

v, ok := im.transIDToContractAddr.Load(transID)

if !ok {

return

}

vEx = v.(\*TxID2ContractAddrMap).Map

delete(vEx, txID)

}

// Commit - commit data when block finished

func (im \*InvokerMgr) Commit(transId int64) {

//依次获取url，进行commit

if v, ok := im.dockerUrlMap.Load(transId); ok {

urlMap := v.(\*UrlMap).Map

for url := range urlMap {

pool := im.dockerConnPool(transId, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

result, err := cli.Call("McCommitTrans", map[string]interface{}{"transID": transId}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

if !result.(bool) {

panic("Commit result is false")

}

}

}

//判断transId对应的缓存中是否有需要通知dockermgr需要更新的合约地址

if v, ok := im.transIDToContractAddr.Load(transId); ok {

m := v.(\*TxID2ContractAddrMap).Map

for \_, addrs := range m {

for \_, addr := range addrs {

smcdocker.GetInstance().DirtyContractInvokeURL(0, 0, addr)

}

}

}

im.dockerUrlMap.Delete(transId)

im.transMap.Delete(transId)

im.transIDToContractAddr.Delete(transId)

// 检查长时间没有发生交易的docker并杀掉

smcdocker.GetInstance().CheckDockerLiveTime()

}

// McDirtyToken - dirty cache data of token, if any contract change it

func (im \*InvokerMgr) McDirtyToken(tokenAddr types.Address) {

im.dockerMapConnPool.Range(func(key, value interface{}) bool {

pool := value.(\*socket.ConnectionPool)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

result, err := cli.Call("McDirtyToken", map[string]interface{}{"tokenAddr": tokenAddr}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

if !result.(bool) {

panic("McDirtyToken result is false")

}

return true

})

}

// McDirtyContract - dirty cache data of contract, if any contract change it.

func (im \*InvokerMgr) McDirtyContract(contractAddr types.Address) {

im.dockerMapConnPool.Range(func(key, value interface{}) bool {

pool := value.(\*socket.ConnectionPool)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

result, err := cli.Call("McDirtyContract", map[string]interface{}{"contractAddr": contractAddr}, 60)

if err != nil {

panic(err)

}

pool.ReleaseClient(cli)

if !result.(bool) {

panic("McDirtyContract result is false")

}

return true

})

}

// Health -

func (im \*InvokerMgr) Health() \*types.Health {

return nil

}

// InitOrUpdateSMC - invoke InitChain/UpdateChain when any contract begin effect at this height.

func (im \*InvokerMgr) InitOrUpdateSMC(transId, txId int64, header types2.Header, contractAddr, owner types.Address, inUpgrade bool) (result \*types.Response) {

result = new(types.Response)

contractAddr, url, err := smcdocker.GetInstance().GetContractInvokeURL(transId, txId, contractAddr)

if err != nil {

panic(err)

}

m := types.Message{

Contract: contractAddr,

}

invokeParam := types.RPCInvokeCallParam{Sender: owner, Message: m}

method := ""

if inUpgrade {

method = "UpdateChain"

} else {

method = "InitChain"

}

pool := im.dockerConnPool(transId, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

defer pool.ReleaseClient(cli)

resp, err := cli.Call(method, map[string]interface{}{"blockHeader": header, "transID": transId, "txID": txId, "callParam": invokeParam}, 60)

if err != nil {

panic(err)

}

err = jsoniter.Unmarshal([]byte(resp.(string)), result)

if err != nil {

panic(err)

}

if im.isGenesisOrgContract(contractAddr) {

im.McDirtyContract("\*")

}

return

}

func (im \*InvokerMgr) isGenesisOrgContract(contractAddr types.Address) bool {

contract := statedbhelper.GetContract(contractAddr)

if contract == nil {

return false

}

return contract.OrgID == statedbhelper.GetGenesisOrgID(0, 0)

}

// cleanDockerCache - clean docker cache

func (im \*InvokerMgr) cleanDockerCache(result \*types.Response) (newContractAddr string) {

for \_, v := range result.Tags {

var receipt std.Receipt

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

if err != nil {

panic(err.Error())

}

/\*\*

根据收据判断，如果 token 或者 contract 的信息有修改，清空所有 docker 中对应的缓存

目前针对 token 修改的信息包括：setGasPrice，burn，addSupply，setOwner

如果合约被禁用或部署了新的合约，通知所有的 docker 清空缓存

\*\*/

switch receipt.Name {

case "std::setGasPrice":

var obj std.SetGasPrice

if err := jsoniter.Unmarshal(receipt.Bytes, &obj); err != nil {

panic(err.Error())

}

im.McDirtyToken(obj.Token)

case "std::burn":

var obj std.Burn

if err := jsoniter.Unmarshal(receipt.Bytes, &obj); err != nil {

panic(err.Error())

}

im.McDirtyToken(obj.Token)

case "std::addSupply":

var obj std.AddSupply

if err := jsoniter.Unmarshal(receipt.Bytes, &obj); err != nil {

panic(err.Error())

}

im.McDirtyToken(obj.Token)

case "std::setOwner", "smartcontract.forbidContract", "IBC.setGasPriceRatio": // 更新了出合约信息外的其它信息，如果合约有token，对应的token的owner也会转移

im.McDirtyContract("\*")

case "smartcontract.deployContract": // 更新了出合约信息外的其它信息

type deployContract struct {

ContractAddr types.Address `json:"contractAddr"`

}

var obj deployContract

if err := jsoniter.Unmarshal(receipt.Bytes, &obj); err != nil {

panic(err.Error())

}

newContractAddr = obj.ContractAddr

im.McDirtyContract("\*")

}

}

return

}

// dockerConnPool get connectionPool object from dockerMapConnPool if it's exist,

// or NewConnectionPool for create connection pool and object,

// Note: bNew means the docker pointed by url is new docker,

// if url exists in dockerMapConnPool, it will be NewConnectionPool also.

func (im \*InvokerMgr) dockerConnPool(transID int64, url string) \*socket.ConnectionPool {

var pool \*socket.ConnectionPool

var err error

value, ok := im.dockerMapConnPool.Load(url)

if !ok {

pool, err = socket.NewConnectionPool(url, 2, im.logger)

if err != nil {

panic(url + ":" + err.Error())

}

im.dockerMapConnPool.Store(url, pool)

} else {

pool = value.(\*socket.ConnectionPool)

}

im.setValDockerMap(transID, url)

return pool

}

// getEffectContract - return effect contract

func (im \*InvokerMgr) getEffectContract(

transId, txId, currentBlock int64,

contractAddr types.Address,

methodID uint32) (contract \*std.Contract, error types.BcError) {

error.ErrorCode = types.CodeOK

// 先判断是不是 token，如果是，找到对应生效的合约地址，

// 如果不是token，找到合约， 如果接口是五个标准接口并且合约发布了 token，则调用失败，否则继续。

// 只有五个标准接口可以使用token地址调用

token := statedbhelper.GetTokenByAddress(transId, txId, contractAddr)

// If token != nil, it means calling a token address

if token != nil {

contract = statedbhelper.GetContract(contractAddr)

if contract == nil {

error.ErrorDesc = ""

error.ErrorCode = types.ErrLogicError

return

}

addrList := statedbhelper.GetContractsByName(transId, txId, contract.Name, contract.OrgID)

if len(addrList) != 0 {

for \_, v := range addrList {

con := statedbhelper.GetContract(v)

if con != nil && con.EffectHeight <= currentBlock &&

(con.LoseHeight == 0 || con.LoseHeight > currentBlock) {

contract = con

return

}

}

}

error.ErrorDesc = "The contract has expired"

error.ErrorCode = types.ErrLogicError

return

} else {

contract = statedbhelper.GetContract(contractAddr)

if contract == nil {

error.ErrorDesc = "Invalid contract address"

error.ErrorCode = types.ErrLogicError

return

}

\_, ok := im.standardMethods[methodID]

if ok && contract.Token != "" {

error.ErrorDesc = "Can not call standard token method by contract address"

error.ErrorCode = types.ErrLogicError

return

}

if contract.EffectHeight > currentBlock {

error.ErrorDesc = "The smart contract is not yet in effect"

error.ErrorCode = types.ErrLogicError

return

}

if contract.LoseHeight != 0 && contract.LoseHeight <= currentBlock {

error.ErrorDesc = "The contract has expired"

error.ErrorCode = types.ErrLogicError

return

}

return

}

}

// Mine - invoke method of mine

func (im \*InvokerMgr) Mine(transId, txId int64, header types2.Header, contractAddr, owner types.Address) (result \*types.Response) {

result = new(types.Response)

contractAddr, url, err := smcdocker.GetInstance().GetContractInvokeURL(transId, txId, contractAddr)

if err != nil {

panic(err)

}

m := types.Message{

Contract: contractAddr,

}

invokeParam := types.RPCInvokeCallParam{Sender: owner, Message: m}

pool := im.dockerConnPool(transId, url)

cli, err := pool.GetClient()

if err != nil {

panic(err)

}

defer pool.ReleaseClient(cli)

resp, err := cli.Call("Mine", map[string]interface{}{"blockHeader": header, "transID": transId, "txID": txId, "callParam": invokeParam}, 60)

if err != nil {

panic(err)

}

err = jsoniter.Unmarshal([]byte(resp.(string)), result)

if err != nil {

panic(err)

}

return

}

// setValToTransMap - set value to transMap

func (im \*InvokerMgr) setValToTransMap(transID, txID int64, url []string) {

var m \*TxID2UrlMap

if v, ok := im.transMap.Load(transID); !ok {

m = &TxID2UrlMap{Map: make(map[int64][]string)}

} else {

m = v.(\*TxID2UrlMap)

}

m.Map[txID] = append(m.Map[txID], url...)

im.transMap.Store(transID, m)

}

// setValToTransCon - set value to transContractAddr

func (im \*InvokerMgr) setValToTransCon(transID, txID int64, addrs []string) {

var m \*TxID2ContractAddrMap

if v, ok := im.transIDToContractAddr.Load(transID); !ok {

m = &TxID2ContractAddrMap{Map: make(map[int64][]string)}

} else {

m = v.(\*TxID2ContractAddrMap)

}

m.Map[txID] = append(m.Map[txID], addrs...)

im.transIDToContractAddr.Store(transID, m)

}

// setValDockerMap - set value to dockerUrlMap

func (im \*InvokerMgr) setValDockerMap(transID int64, url string) {

var m \*UrlMap

if v, ok := im.dockerUrlMap.Load(transID); !ok {

m = &UrlMap{Map: make(map[string]struct{})}

} else {

m = v.(\*UrlMap)

}

m.Map[url] = struct{}{}

im.dockerUrlMap.Store(transID, m)

}

// getPayer - return payer address, the payer is sender if method gas is not negative,

// else return contract account address

func (im \*InvokerMgr) getPayer(height int64, sender types.Address, tx types.Transaction, index int) (types.Address, error) {

// 创世时走如下逻辑

if len(tx.Messages) == 1 && tx.Messages[0].Contract == std.GetGenesisContractAddr(statedbhelper.GetChainID()) {

return sender, nil

}

// load contract buffer

contracts := make([]\*std.Contract, 0)

for \_, msg := range tx.Messages {

contractSplit := strings.Split(msg.Contract, ".")

contractAddr := msg.Contract

if len(contractSplit) == 2 {

contractAddr = contractSplit[1]

}

contract, err := im.getEffectContract(0, 0, height, contractAddr, msg.MethodID)

if err.ErrorCode != types.CodeOK {

return "", errors.New(err.ErrorDesc)

}

contracts = append(contracts, contract)

}

if len(tx.Messages) == 2 && index == 0 {

for \_, method := range contracts[1].Methods {

if method.MethodID == fmt.Sprintf("%x", tx.Messages[1].MethodID) {

if method.Gas < 0 && tx.Messages[0].MethodID == 0x44d8ca60 && contracts[0].Token != "" {

return contracts[1].Account, nil

}

}

}

}

// construct key

for \_, method := range contracts[index].Methods {

if method.MethodID == fmt.Sprintf("%x", tx.Messages[index].MethodID) && method.Gas < 0 {

return contracts[index].Account, nil

}

}

return sender, nil

}

func (im \*InvokerMgr) loadContractInfo(height int64, contractAddr types.Address, methodID uint32) (\*std.Contract, error) {

contract, err := im.getEffectContract(0, 0, height, contractAddr, methodID)

if err.ErrorCode != types.CodeOK {

return nil, errors.New(err.ErrorDesc)

}

im.contractBuffer.Store(contract.Address, contract.Account)

im.contractBuffer.Store(contract.Address+"Token", contract.Token)

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

key := contract.Address + "\_" + method.MethodID

im.contractBuffer.Store(key, method.Gas)

}

return contract, nil

}

func Init(name string, ip string, port string) (\*bcdb.GILevelDB, bool) {

var err error

once.Do(func() {

sdb, err = bcdb.OpenDB(name, ip, port)

})

return sdb, err == nil && sdb != nil

}

func NewTransaction() int64 {

// Transaction could be concurrently.

var transID int64

{

mu.Lock()

transactionID += 1

transID = transactionID

defer mu.Unlock()

}

idToTrans.Store(transID, &Transaction{

TransBuffer: make(map[string][]byte),

TxIDToTx: make(map[int64]\*Tx)})

return transID

}

func Get(transID int64, txID int64, key string) []byte {

if transID == 0 {

//Get data from sdb directly

value, err := sdb.Get([]byte(key))

if err != nil {

panic(err)

}

return value

}

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic(fmt.Sprintf("Invalid transID: %d", transID))

}

// Get data from tx buffer

trans := transTemp.(\*Transaction)

if tx, ok := trans.TxIDToTx[txID]; ok {

if value, ok := tx.TxBuffer[key]; ok {

return value

}

}

// Get data from trans buffer

if value, ok := trans.TransBuffer[key]; ok {

return value

}

//Get data from sdb directly

value, err := sdb.Get([]byte(key))

if err != nil {

panic(err)

}

return value

}

func Set(transID int64, txID int64, key string, value []byte) {

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic(fmt.Sprintf("Invalid transID: %d", transID))

}

trans := transTemp.(\*Transaction)

tx, ok := trans.TxIDToTx[txID]

if ok {

tx.TxBuffer[key] = value

} else {

var tx Tx

tx.TxBuffer = make(map[string][]byte)

tx.TxID = txID

tx.TxBuffer[key] = value

trans.TxIDToTx[txID] = &tx

}

}

// SetToTrans set value to trans cache, RollbackTx func won't rollback this value.

// Set account nonce using this func.

func SetToTrans(transID int64, key string, value []byte) {

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic("Invalid transID.")

}

trans := transTemp.(\*Transaction)

trans.TransBuffer[key] = value

}

func BatchSet(transID int64, txID int64, data map[string][]byte) {

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic(fmt.Sprintf("Invalid transID: %d", transID))

}

trans := transTemp.(\*Transaction)

tx, ok := trans.TxIDToTx[txID]

if ok {

for k, v := range data {

tx.TxBuffer[k] = v

}

} else {

var tx Tx

tx.TxBuffer = make(map[string][]byte)

tx.TxID = txID

for k, v := range data {

tx.TxBuffer[k] = v

}

trans.TxIDToTx[txID] = &tx

}

}

func RollbackTx(transID int64, txID int64) {

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic(fmt.Sprintf("Invalid transID: %d", transID))

}

trans := transTemp.(\*Transaction)

\_, ok = trans.TxIDToTx[txID]

if !ok {

//panic(fmt.Sprintf("Invalid txID: %d", txID))

} else {

delete(trans.TxIDToTx, txID)

}

}

func CommitTx(transID int64, txID int64) ([]byte, map[string][]byte) {

transTemp, ok := idToTrans.Load(transID)

if !ok {

panic(fmt.Sprintf("Invalid transID: %d", transID))

}

trans := transTemp.(\*Transaction)

tx, ok := trans.TxIDToTx[txID]

if !ok {

//panic(fmt.Sprintf("Invalid txID: %d", txID))

return nil, nil

}

var keys []string

for k, v := range tx.TxBuffer {

trans.TransBuffer[k] = v

keys = append(keys, k)

}

sort.Strings(keys)

var buf bytes.Buffer

for \_, k := range keys {

v := tx.TxBuffer[k]

buf.Write([]byte(k))

buf.Write(v)

}

delete(trans.TxIDToTx, txID)

return buf.Bytes(), tx.TxBuffer

}