var Version = make(map[string][]string)

func blockHeight(chainID string) error {

blkHeight, err := core.BlockHeight(chainID)

if err != nil {

Error(fmt.Sprintf("Query Block Height failed, %v", err.Error()))

return err

}

if blkHeight.LastBlock == 0 {

fmt.Println(" BlockHeight query failed. Please check the input parameters")

return err

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&blkHeight, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return nil

}

func block(height, bTime, num, chainID string) (err error) {

if height != "" && bTime != "" {

fmt.Println("height and time cannot be assigned together")

return nil

}

// if height is empty, then set it current height

if height == "" && bTime == "" {

blkResult, err := core.BlockHeight(chainID)

if err != nil {

return err

}

height = fmt.Sprintf("%d", blkResult.LastBlock)

}

var iHeight, iNum \*int64

if height != "" {

iHeight = new(int64)

\*iHeight, err = strconv.ParseInt(height, 10, 64)

if err != nil {

return

}

}

if num != "" {

iNum = new(int64)

\*iNum, err = strconv.ParseInt(num, 10, 64)

if err != nil {

return err

}

}

blk, err := core.Block(iHeight, bTime, iNum, chainID)

if err != nil {

Error(fmt.Sprintf("Query Block \"%v\" information failed, %v", height, err.Error()))

return err

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&blk, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return nil

}

func balance(accAddress types.Address, name, password, tokenName, allStr string, chainID, keyStorePath string) error {

all, err := strconv.ParseBool(allStr)

if err != nil {

return err

}

result, err := core.Balance(accAddress, name, password, tokenName, all, chainID, keyStorePath)

if err != nil {

Error(fmt.Sprintf("Query balance \"%v\" information failed, %v", accAddress, err.Error()))

return err

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return nil

}

func nonce(accAddress types.Address, name, password, chainID, keyStorePath string) error {

if accAddress == "" && name == "" {

fmt.Println("Need name or accAddress, cannot all be empty")

return nil

}

result, err := core.Nonce(accAddress, name, password, chainID, keyStorePath)

if err != nil {

Error(fmt.Sprintf("Query nonce \"%v\" information failed, %v", accAddress, err.Error()))

return err

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return nil

}

func commitTx(tx, file, chainID string) error {

if tx == "" && file == "" {

fmt.Println("tx or file cannot be empty")

return nil

}

txs := make([]string, 0)

var err error

if file != "" {

txs, err = checkFileForCommitTx(file, chainID)

if err != nil {

return err

}

}

if tx != "" {

err = checkTxData(tx, chainID)

if err != nil {

return err

}

txs = append(txs, tx)

}

fmt.Println("OK")

fmt.Printf("Response: \n")

for \_, tx := range txs {

result, err := core.CommitTx(chainID, tx)

if err != nil {

Error(fmt.Sprintf("Commit transaction \"%v\" information failed, %v", tx, err.Error()))

return err

}

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Println(string(jsIndent))

}

return nil

}

func checkFileForCommitTx(file, chainID string) (txs []string, err error) {

con, err := ioutil.ReadFile(file)

if err != nil {

return

}

if len(con) == 0 {

return nil, errors.New("file cannot be empty")

}

conStr := strings.Trim(string(con), "\r\n")

conStr = strings.Trim(conStr, "\n")

lines := strings.Split(string(conStr), "\r\n")

if len(lines) <= 1 {

lines = strings.Split(string(conStr), "\n")

}

for index, v := range lines {

err := checkTxData(v, chainID)

if err != nil {

return nil, errors.New(fmt.Sprintf("%d %v", index+1, err))

}

txs = append(txs, strings.TrimSpace(v))

}

return

}

func checkTxData(tx, chainID string) error {

if chainID == "" {

chainID = common.GetBCCConfig().DefaultChainID

}

MAC := chainID + "<tx>"

strSplit := strings.Split(tx, ".")

if len(strSplit) != 5 {

return errors.New("tx string seg number must be 5 that split by dot")

}

if strSplit[0] != MAC {

return errors.New("tx string must prefix with " + MAC)

}

if strSplit[1] != "v1" && strSplit[1] != "v2" {

return errors.New(`tx string version wrong, must be "v1" or "v2"`)

}

return nil

}

func deployContract(name, password, contractName, version, orgName, codeFile,

effectHeight, owner, keyStorePath, gasLimit, note, chainID string) error {

param := core.DeployContractParam{

ContractName: contractName,

Version: version,

OrgName: orgName,

CodeFile: codeFile,

EffectHeight: effectHeight,

Owner: owner,

ChainID: chainID,

KeyStorePath: keyStorePath,

GasLimit: gasLimit,

Note: note,

}

result, err := core.DeployContract(name, password, param)

if err != nil {

Error(err.Error())

}

// 记录合约迭代版本

//myVersion := new(core.VersionOfContract)

//myVersion.Version[name] = append(myVersion.Version[name], version)

Version[name] = append(Version[name], version)

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func registerToken(name, password, tokenName, tokenSymbol, totalSupply, gasPrice, gasLimit, note, keyStorePath, chainID,

addSupplyEnabled, burnEnabled string) error {

param := core.RegisterTokenParam{

TokenName: tokenName,

TokenSymbol: tokenSymbol,

TotalSupply: totalSupply,

AddSupplyEnabled: addSupplyEnabled,

BurnEnabled: burnEnabled,

GasPrice: gasPrice,

ChainID: chainID,

KeyStorePath: keyStorePath,

GasLimit: gasLimit,

Note: note,

}

result, err := core.RegisterToken(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func registerOrg(name, password, orgName, gasLimit, note, keyStorePath, chainID string) error {

param := core.RegisterOrgParam{

OrgName: orgName,

ChainID: chainID,

KeyStorePath: keyStorePath,

GasLimit: gasLimit,

Note: note,

}

result, err := core.RegisterOrg(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func setOrgSigners(name, password, orgName, pubKeys, gasLimit, note, keyStorePath, chainID string) error {

param := core.SetOrgSignersParam{

OrgName: orgName,

PubKeys: pubKeys,

ChainID: chainID,

KeyStorePath: keyStorePath,

GasLimit: gasLimit,

Note: note,

}

result, err := core.SetOrgSigners(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func setOrgDeployer(name, password, orgName, deployer, gasLimit, note, keyStorePath, chainID string) error {

param := core.SetOrgDeployerParam{

OrgName: orgName,

Deployer: deployer,

ChainID: chainID,

KeyStorePath: keyStorePath,

GasLimit: gasLimit,

Note: note,

}

result, err := core.SetOrgDeployer(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func transfer(name, password, token, gasLimit, note, to, value, keyStorePath, chainID string) error {

param := core.TransferParam{

Token: token,

GasLimit: gasLimit,

Note: note,

To: to,

Value: value,

ChainID: chainID,

KeyStorePath: keyStorePath,

}

result, err := core.Transfer(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return err

}

func runAsRPCService() (err error) {

cmd := exec.Command("/bin/bash", "-c", "./gicrpcservice")

var out bytes.Buffer

cmd.Stdout = &out

err = cmd.Start()

if err != nil {

return

}

fmt.Println("Execute finished")

return nil

}

// Query the contract information based on the parameters

func ContractInfo(orgName, contractName, orgID, contractAddr string) (err error) {

if orgID != "" && contractName != "" && contractAddr == "" {

contractList, err := core.ContractInfo(chainID, orgID, contractName)

if err != nil {

Error(err.Error())

}

for \_, v := range contractList {

// 校验其他输入参数

if orgName != "" {

OrgInfo, err := core.QueryOrgInfo(orgID, chainID)

if err != nil {

Error(err.Error())

}

if orgName != OrgInfo.Name {

fmt.Println("Error: Input orgName is wrong.")

return err

}

}

err = ParamsExample(&v)

if err != nil {

Error(err.Error())

}

}

} else if orgName != "" && contractName != "" && contractAddr == "" {

contract, err := core.QueryContractInfo(orgName, contractName, chainID, keyStorePath)

if err != nil {

Error(err.Error())

}

// 校验其他输入参数

if orgID != "" && orgID != contract.OrgID {

fmt.Println("Error: Input orgID is wrong.")

return err

}

err = ParamsExample(contract)

if err != nil {

Error(err.Error())

}

} else if contractAddr != "" {

contract, err := core.ContractInfoWithAddr(chainID, contractAddr)

if err != nil {

Error(err.Error())

}

// 校验其他输入参数

if orgName != "" && orgID != "" {

OrgInfo, err := core.QueryOrgInfo(orgID, chainID)

if err != nil {

Error(err.Error())

}

if orgName != OrgInfo.Name {

fmt.Println("Error: Input orgName is wrong.")

return err

}

}

if orgID != "" && orgID != contract.OrgID {

fmt.Println("Error: orgID orgName is wrong.")

return err

}

err = ParamsExample(contract)

if err != nil {

Error(err.Error())

}

} else if contractName != "" {

\_, contractList, err := core.AllContractInfo(chainID)

if err != nil {

fmt.Println("Query ContractInfo failed")

return err

}

var validContractIndex int

for i, contract := range contractList {

if contract.(std.Contract).Name == contractName {

validContractIndex = i

if validContractIndex != i {

if contractList[validContractIndex].(std.Contract).Version < contractList[i].(std.Contract).Version {

validContractIndex = i

}

}

}

}

fmt.Println("OK")

fmt.Println("Response: ")

contract := contractList[validContractIndex].(std.Contract)

err = ParamsExample(&contract)

if err != nil {

Error(err.Error())

}

} else if orgName == "" && contractName == "" && orgID == "" && contractAddr == "" {

\_, contractList, err := core.AllContractInfo(chainID)

if err != nil {

fmt.Println("Query ContractInfo failed")

return err

}

fmt.Println("OK")

fmt.Println("Response: ")

for \_, contract := range contractList {

fmt.Printf(" contract name: %s\n contract addr: %s\n\n", contract.(std.Contract).Name, contract.(std.Contract).Address)

}

} else if contractName != "" && orgName == "" && orgID == "" && contractAddr == "" {

fmt.Println("More parameters are required: orgName or orgID")

return err

} else {

if (orgName != "" || orgID != "") && contractName == "" {

fmt.Println("contract name cannot be emtpy")

}

return err

}

return

}

func TokenInfo(tokenName, chainId string) (err error) {

if tokenName != "" {

err = core.TokenInfo(tokenName, chainID)

} else {

err = core.AllTokenInfo(chainId)

}

if err != nil {

return err

}

return

}

func ParamsExample(contract \*std.Contract) (err error) {

address, \_ := json.MarshalIndent(&contract.Address, "", " ")

account, \_ := json.MarshalIndent(&contract.Account, "", " ")

orgid, \_ := json.MarshalIndent(&contract.OrgID, "", " ")

name, \_ := json.MarshalIndent(&contract.Name, "", " ")

owner, \_ := json.MarshalIndent(&contract.Owner, "", " ")

codeHash, \_ := json.MarshalIndent(&contract.CodeHash, "", " ")

version, \_ := json.MarshalIndent(&contract.Version, "", " ")

EffectHeight, \_ := json.MarshalIndent(&contract.EffectHeight, "", " ")

loseEffect, \_ := json.MarshalIndent(&contract.LoseHeight, "", " ")

keyPrefix, \_ := json.MarshalIndent(&contract.KeyPrefix, "", " ")

interfaces, \_ := json.MarshalIndent(&contract.Interfaces, "", " ")

token, \_ := json.MarshalIndent(&contract.Token, "", " ")

fmt.Println("OK")

fmt.Printf("Response: \n")

fmt.Printf(" Version: %s\n", string(version))

fmt.Printf(" Name: %s\n", string(name))

fmt.Printf(" OrgID: %s\n", string(orgid))

fmt.Printf(" Address: %s\n", string(address))

fmt.Printf(" Account: %s\n", string(account))

fmt.Printf(" Owner: %s\n", string(owner))

fmt.Printf(" CodeHash: %s\n", string(codeHash))

fmt.Printf(" EffectHeight: %s\n", string(EffectHeight))

fmt.Printf(" LoseHeight: %s\n", string(loseEffect))

fmt.Printf(" KeyPrefix: %s\n", string(keyPrefix))

fmt.Printf(" Token: %s\n", string(token))

fmt.Printf(" Interfaces: %s\n", string(interfaces))

fmt.Printf(" Method: \n")

var example2 = ""

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

leftBracketIndex := strings.Index(v.ProtoType, "(")

rightBracketIndex := strings.Index(v.ProtoType, ")")

splitTypes := strings.Split(v.ProtoType[leftBracketIndex+1:rightBracketIndex], ",")

example := make([]string, 0)

for \_, v := range splitTypes {

v = checkType(v)

example = append(example, v)

example2 = strings.Join(example, "@")

}

fmt.Printf(" %s\n methodId: %s\n Params: %s\n\n", v.ProtoType, v.MethodID, example2)

}

fmt.Println("PS: If the string is just a string, Example: \"example\"\n " +

"If the string is a special string, Example: \"recvFeeRatio\":[500,500], \"recvFeeAddr\":[\"localKrHJUVGAt4R9gcfsBthu3dWJR7bAYq1c8\",\"localNwdwjpDotDDLGiB9pARk1CcSM71bdgTef\"]")

return

}

func checkType(Type interface{}) string {

switch Type {

case "int", "int8", "int16", "int32", "int64":

return "200000"

case "uint", "uint8", "uint16", "uint32", "uint64":

return "200000"

case "float32", "float64":

return "20.11"

case "types.Address":

return "localL9BzYNYns5VCRaJgfHEBJLzS1bhpHjx7j"

case "bn.Number":

return "1000000000000"

case "bool":

return "true"

case "byte":

return "0x01bd6c29d63f5f32aa33955f26a28459988edea4de517f77372e77db33958e6e"

case "types.Hash", "types.HexBytes", "types.PubKey", "[]byte":

return "0x01bd6c29d63f5f32aa33955f26a28459988edea4de517f77372e77db33958e6e"

case "string":

return "example"

default:

return ""

}

}

// query blockChain information

func query(key, chainID string) error {

if key == "" {

fmt.Println("key cannot be empty")

return nil

}

result, err := core.QueryOfRpc(key, chainID)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

fmt.Printf("Response: \n")

fmt.Printf(" Code: %v\n", result.Response.Code)

fmt.Printf(" Key: %s\n", string(result.Response.Key))

fmt.Printf(" Value: %s\n", string(result.Response.Value))

return nil

}

// SolDeploy for solidity contract create and call

func SolDeploy(name, password, tokenAddr, tokenName, sourceFile, binFile, abiFile, gasLimit, note, chainID, keyStorePath string, params []string) (err error) {

param := core.BVMDeployParam{

TokenAddr: tokenAddr,

TokenName: tokenName,

SourceFile: sourceFile,

BinFile: binFile,

AbiFile: abiFile,

ParamsArray: params,

GasLimit: gasLimit,

Note: note,

ChainID: chainID,

KeyStorePath: keyStorePath,

}

result, err := core.SolDeploy(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return

}

// SolCall for solidity contract create and call

func SolCall(abiFile, name, password, contractAddr, value, gasLimit, note, chainID, keyStorePath, method string, paramsArr []string) (err error) {

param := core.BVMCallParam{

AbiFile: abiFile,

ContractAddr: contractAddr,

Value: value,

Method: method,

ParamsArray: paramsArr,

Note: note,

GasLimit: gasLimit,

ChainID: chainID,

KeyStorePath: keyStorePath,

}

result, err := core.SolCall(name, password, param)

if err != nil {

Error(err.Error())

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&result, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return

}

const (

layout = "2006-01-02 15:04:05 +0000 UTC"

)

var (

db \*gidb.GILevelDB

addrS []string

dbName = ".gicCache"

dbIP = "127.0.0.1"

dbPort = "55678"

)

// KeyOfHeight return key of height for access time

func KeyOfHeight(height int64) []byte {

return []byte(fmt.Sprintf("/%d", height))

}

// SetAddrList set nodes addresses

func SetAddrList(addrList []string) {

addrS = addrList

}

// BinarySearchEx binary search algorithm with interval

func BinarySearchEx(start, end, interval int64, t time.Time) (height int64) {

if start != 0 && start >= end {

if CompareWithTime(start, t, false) != 0 {

return

}

return start

}

// set start with current block height

if start == 0 {

result := new(core\_types.ResultABCIInfo)

err := common.DoHttpRequestAndParse(addrS, "abci\_info", map[string]interface{}{}, result)

if err != nil {

panic(err)

}

start = result.Response.LastBlockHeight

// check latest block time

if r := CompareWithTime(start, t, false); r == -1 {

return

} else if r == 0 {

return end

}

// check oldest block time

if r := CompareWithTime(1, t, true); r == 1 {

return

} else if r == 0 {

return 1

}

}

// reset start and end

if end == 0 {

end = start

start = (start / interval) \* interval

}

// compare time between start's time and t

r := CompareWithTime(start, t, false)

if r == 1 {

if start != 1 {

end = start

start = 1

}

m := ((end - 1) / interval / 2) \* interval

r = CompareWithTime(m, t, true)

if r == 1 {

return BinarySearchEx(start, m, interval, t)

} else if r == -1 {

return BinarySearchEx(m, end, interval, t)

} else {

return m

}

} else if r == -1 {

if end-start <= interval {

return binarySearch(start, end, t)

} else {

m := start + ((end-start)/interval/2)\*interval

if m == start {

m = start + interval

}

r = CompareWithTime(m, t, true)

if r == 1 {

return BinarySearchEx(start, m, interval, t)

} else if r == -1 {

return BinarySearchEx(m, end, interval, t)

} else {

return m

}

}

} else {

return start

}

}

func binarySearch(start, end int64, t time.Time) (height int64) {

if start > end {

return

}

if start == end {

return start

} else if start+1 == end {

return nearlyBetweenTwoHeight(start, end, t)

} else {

m := (start + end) / 2

r := CompareWithTime(m, t, false)

if r == 1 {

return binarySearch(start, m, t)

} else if r == -1 {

return binarySearch(m, end, t)

} else {

return m

}

}

}

// CompareWithTime compare time between block time and t,

// then return 1 if block time is bigger, else return -1,

// block time equal t return 0

func CompareWithTime(h int64, t time.Time, bSave bool) int {

ht := timeOfHeightFromBlock(h)

if bSave {

setTimeOfHeight(h, ht.String())

}

if ht.Sub(t) > 0 {

return 1

} else if ht.Sub(t) < 0 {

return -1

}

return 0

}

func timeOfHeight(height int64) string {

v, err := db.Get(KeyOfHeight(height))

if err != nil {

return ""

}

return string(v)

}

func nearlyBetweenTwoHeight(h1, h2 int64, t time.Time) (h int64) {

t1 := timeOfHeightFromBlock(h1)

t2 := timeOfHeightFromBlock(h2)

d1 := math.Abs(float64(t1.Sub(t).Nanoseconds()))

d2 := math.Abs(float64(t2.Sub(t).Nanoseconds()))

if d1 > d2 {

return h2

} else {

return h1

}

}

type accountCache struct {

Nonce uint64 `json:"nonce"`

}

var (

cacheFilePath = ""

)

func Init(path string) {

cacheFilePath = path

var err error

db, err = gidb.OpenDB(dbName, dbIP, dbPort)

if err != nil {

panic(err)

}

}

func pathOfContract(orgID, contractName, keyStorePath string) string {

return filepath.Join(keyStorePath, orgID+"\_"+contractName+".toolCache")

}

func pathOfAccount(accountName, keyStorePath string) string {

return filepath.Join(keyStorePath, accountName+".toolCache")

}

// Contract contract toolCache

func Contract(orgID, contractName, keyStorePath string) (contract \*std.Contract, err error) {

if keyStorePath == "" {

keyStorePath = cacheFilePath

}

contractCachePath := pathOfContract(orgID, contractName, keyStorePath)

f, err := os.Open(contractCachePath)

if err != nil {

return

}

defer f.Close()

var b []byte

\_, err = f.Read(b)

if err != nil {

return

}

contract = new(std.Contract)

err = jsoniter.Unmarshal(b, contract)

return

}

// SetContract

func SetContract(contract \*std.Contract, keyStorePath string) (err error) {

if keyStorePath == "" {

keyStorePath = cacheFilePath

}

contractCachePath := pathOfContract(contract.OrgID, contract.Name, keyStorePath)

f, err := os.Create(contractCachePath)

if err != nil {

return

}

defer f.Close()

resBytes, err := jsoniter.Marshal(contract)

if err != nil {

return

}

\_, err = f.Write(resBytes)

return

}

// Nonce account toolCache

func Nonce(name, keyStorePath string) (nonce uint64, err error) {

if keyStorePath == "" {

keyStorePath = cacheFilePath

}

accountCachePath := pathOfAccount(name, keyStorePath)

f, err := os.Open(accountCachePath)

if err != nil {

return

}

defer f.Close()

var b []byte

\_, err = f.Read(b)

if err != nil {

return

}

var ac accountCache

err = jsoniter.Unmarshal(b, &ac)

if err != nil {

return

}

return ac.Nonce, nil

}

// SetNonce

func SetNonce(name string, nonce uint64, keyStorePath string) (err error) {

if keyStorePath == "" {

keyStorePath = cacheFilePath

}

accountCachePath := pathOfAccount(name, keyStorePath)

f, err := os.OpenFile(accountCachePath, os.O\_CREATE|os.O\_RDWR, os.ModePerm)

if err != nil {

return

}

var b []byte

\_, err = f.Read(b)

if err != nil {

return

}

var ac accountCache

if len(b) > 0 {

err = jsoniter.Unmarshal(b, &ac)

if err != nil {

return

}

}

ac.Nonce = nonce

resBytes, err := jsoniter.Marshal(ac)

if err != nil {

return

}

\_, err = f.Write(resBytes)

return

}

//网络请求和结果解析-故障队列版

func DoHttpRequestAndParseExBlock(nodeAddrSlice []string, methodName string, params map[string]interface{}, result interface{}) (err error) {

for {

RWLock.Lock()

if len(nodeAddrSlice) == 0 {

RWLock.Unlock()

return errors.New("no available nodes can to connect")

}

length := len(nodeAddrSlice)

var rnd int

if length > 1 {

rnd = rand.Intn(length - 1)

} else {

rnd = 0

}

url := nodeAddrSlice[rnd]

RWLock.Unlock()

err = CallChainApi(url, methodName, params, result)

if err == nil {

break

} else {

RWLock.Lock()

if \_, ok := FaultCounterMap[url]; !ok {

FaultCounterMap[url] = 0

}

FaultCounterMap[url] += 1

if FaultCounterMap[url] > 10 {

if rnd == length-1 {

nodeAddrSlice = append(nodeAddrSlice[:rnd])

} else {

nodeAddrSlice = append(nodeAddrSlice[:rnd], nodeAddrSlice[rnd+1:]...)

}

length -= 1

}

RWLock.Unlock()

if length <= len(nodeAddrSlice)/3 {

go DealFaultUrls(nodeAddrSlice, methodName, params, result)

}

if length == 0 {

splitErr := strings.Split(err.Error(), ":")

return errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

return

}

func CallChainApi(url string, methodName string, params map[string]interface{}, result interface{}) (err error) {

rpc := NewJSONRPCClientEx(url, "", true)

\_, err = rpc.Call(methodName, params, result)

return

}

func NewJSONRPCClientEx(remote, certFile string, disableKeepAlive bool) \*JSONRPCClient {

var pool \*x509.CertPool

if certFile != "" {

pool = x509.NewCertPool()

caCert, err := ioutil.ReadFile(certFile)

if err != nil {

fmt.Println(err.Error())

return nil

}

pool.AppendCertsFromPEM(caCert)

}

address, client := makeHTTPSClient(remote, pool, disableKeepAlive)

return &JSONRPCClient{

address: address,

client: client,

cdc: rpcclient.CDC,

}

}

func makeHTTPSClient(remoteAddr string, pool \*x509.CertPool, disableKeepAlive bool) (string, \*http.Client) {

//\_, dialer := makeHTTPDialer(remoteAddr)

tr := new(http.Transport)

tr.DisableKeepAlives = disableKeepAlive

tr.IdleConnTimeout = time.Second \* 120

if pool != nil {

tr.TLSClientConfig = &tls.Config{RootCAs: pool}

} else {

tr.TLSClientConfig = &tls.Config{InsecureSkipVerify: true}

}

return remoteAddr, &http.Client{Transport: tr, Timeout: time.Duration(time.Second \* 3)}

}

func (c \*JSONRPCClient) Call(method string, params map[string]interface{}, result interface{}) (interface{}, error) {

//request, err := types.MapToRequest("jsonrpc-client", method, params)

request, err := rpctypes.MapToRequest("jsonrpc-client", method, params)

if err != nil {

return nil, err

}

requestBytes, err := json.Marshal(request)

if err != nil {

fmt.Println("lib client http\_client error to json.Marshal(request)")

return nil, err

}

// log.Info(string(requestBytes))

requestBuf := bytes.NewBuffer(requestBytes)

// log.Info(Fmt("RPC request to %v (%v): %v", c.remote, method, string(requestBytes)))

httpResponse, err := c.client.Post(c.address, "text/json", requestBuf)

if err != nil {

return nil, err

}

defer httpResponse.Body.Close() // nolint: errcheck

responseBytes, err := ioutil.ReadAll(httpResponse.Body)

if err != nil {

return nil, err

}

// log.Info(Fmt("RPC response: %v", string(responseBytes)))

return unmarshalResponseBytes(c.cdc, responseBytes, result)

}

func unmarshalResponseBytes(cdc \*amino.Codec, responseBytes []byte, result interface{}) (interface{}, error) {

// Read response. If rpc/core/types is imported, the result will unmarshal

// into the correct type.

// log.Notice("response", "response", string(responseBytes))

var err error

//response := &types.RPCResponse{}

response := &rpctypes.RPCResponse{}

err = json.Unmarshal(responseBytes, response)

if err != nil {

return nil, errors.New("Error unmarshalling rpc response: " + err.Error())

}

if response.Error != nil {

return nil, errors.New("Response error: " + response.Error.Error())

}

// Unmarshal the RawMessage into the result.

err = cdc.UnmarshalJSON(response.Result, result)

if err != nil {

return nil, errors.New("Error unmarshalling rpc response result: " + err.Error())

}

return result, nil

}

//网络请求和结果解析

func DoHttpRequestAndParse(nodeAddrSlice []string, methodName string, params map[string]interface{}, result interface{}) (err error) {

for i, nodeAddr := range nodeAddrSlice {

rpc := rpcclient.NewJSONRPCClientEx(nodeAddr, "", true)

\_, err = rpc.Call(methodName, params, result)

if err == nil {

break

} else {

if i == len(nodeAddrSlice)-1 {

splitErr := strings.Split(err.Error(), ":")

return errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

return

}

// 网络请求和结果解析

func DoHttpCommitTxAndParse(nodeAddrSlice []string, txStr string) (result \*core\_types.ResultBroadcastTxCommit, err error) {

result = new(core\_types.ResultBroadcastTxCommit)

for i, nodeAddr := range nodeAddrSlice {

rpc := rpcclient.NewJSONRPCClientEx(nodeAddr, "", true)

\_, err = rpc.Call("broadcast\_tx\_commit", map[string]interface{}{"tx": []byte(txStr)}, result)

if err == nil {

break

} else {

if i == len(nodeAddrSlice)-1 {

splitErr := strings.Split(err.Error(), ":")

return nil, errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

return result, nil

}

// 网络请求和结果解析

func DoHttpCommitTxAndParseAsync(nodeAddrSlice []string, txStr string) (result \*core\_types.ResultTx, err error) {

git := new(core\_types.ResultBroadcastTx)

for i, nodeAddr := range nodeAddrSlice {

rpc := rpcclient.NewJSONRPCClientEx(nodeAddr, "", true)

\_, err = rpc.Call("broadcast\_tx\_async", map[string]interface{}{"tx": []byte(txStr)}, git)

if err == nil {

result = new(core\_types.ResultTx)

for {

err = DoHttpRequestAndParse(nodeAddrSlice, "tx", map[string]interface{}{"hash": strings.ToUpper(hex.EncodeToString(git.Hash))}, result)

if err != nil {

return

}

if result.CheckResult.Code != 0 && result.CheckResult.Code != types.CodeOK {

return

}

if result.DeliverResult.Code != 0 {

return

}

time.Sleep(1 \* time.Second)

}

} else {

if i == len(nodeAddrSlice)-1 {

splitErr := strings.Split(err.Error(), ":")

return nil, errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

return result, nil

}

func DoHttpQueryAndParse(nodeAddrSlice []string, key string, data interface{}) (err error) {

value, err := DoHttpQuery(nodeAddrSlice, key)

if err != nil {

return

}

if len(value) == 0 {

return errors.New("return value is empty, please check key=" + key)

}

err = json.Unmarshal(value, data)

return

}

func DoBulkHttpQueryAndParse(nodeAddrSlice []string, key string, data interface{}) (result []interface{}, err error) {

values, err := DoBulkHttpQuery(nodeAddrSlice, key)

if err != nil {

return

}

if len(values) == 0 {

return nil, errors.New("return value is empty, please check key=" + key)

}

for \_, value := range values {

err = json.Unmarshal(value, data)

temp := reflect.ValueOf(data).Elem().Interface()

result = append(result, temp)

}

return

}

func DoBulkHttpQuery(nodeAddrSlice []string, key string) (value [][]byte, err error) {

result := new(core\_types.ResultABCIQueryEx)

for i, nodeAddr := range nodeAddrSlice {

rpc := rpcclient.NewJSONRPCClientEx(nodeAddr, "", true)

\_, err = rpc.Call("abci\_query\_ex", map[string]interface{}{"path": key}, result)

if err == nil {

break

} else {

if i == len(nodeAddrSlice)-1 {

splitErr := strings.Split(err.Error(), ":")

return nil, errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

for \_, KeyValues := range result.Response.KeyValues {

value = append(value, KeyValues.Value)

}

return

}

func DoHttpQueryForRpc(nodeAddrSlice []string, key string) (result \*core\_types.ResultABCIQuery, err error) {

result = new(core\_types.ResultABCIQuery)

for i, nodeAddr := range nodeAddrSlice {

rpc := rpcclient.NewJSONRPCClientEx(nodeAddr, "", true)

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

if err == nil {

break

} else {

if i == len(nodeAddrSlice)-1 {

splitErr := strings.Split(err.Error(), ":")

return nil, errors.New(strings.Trim(splitErr[len(splitErr)-1], " "))

}

}

}

return

}

var (

gicConfig BccConfig

gicRPCServiceConfig BccRPCServiceConfig

logger log.Logger

FaultCounterMap = make(map[string]int)

CorrectUrls = make([]string, 0)

ContractMap = make(map[string]\*types.Contract)

RWLock = new(sync.RWMutex)

)

func LoadBCCConfig() error {

configFile := "./.config/gic.yaml"

err := InitConfig(&gicConfig, configFile)

if err != nil {

return errors.New("Init config fail err info : " + err.Error())

}

return nil

}

func InitRPC() error {

configFile := "./.config/gicRpcService.yaml"

moduleName := "gicRPCService"

err := InitConfig(&gicRPCServiceConfig, configFile)

if err != nil {

return errors.New("Init config fail err info : " + err.Error())

}

initLog(moduleName)

return nil

}

func GetBCCConfig() BccConfig {

return gicConfig

}

func GetBCCServiceConfig() BccRPCServiceConfig {

return gicRPCServiceConfig

}

func initLog(moduleName string) {

l := log.NewTMLogger("./log", moduleName)

l.SetOutputToFile(true)

l.SetOutputToScreen(false)

l.AllowLevel(gicRPCServiceConfig.LoggerLevel)

logger = l

}

func GetLogger() log.Logger {

return logger

}

func OutCertFileIsExist() (string, string) {

crtPath := "./.config/server.crt"

keyPath := "./.config/server.key"

\_, err := os.Stat(gicRPCServiceConfig.OutCertPath + ".crt")

if err != nil {

return crtPath, keyPath

}

\_, err = os.Stat(gicRPCServiceConfig.OutCertPath + ".key")

if err != nil {

return crtPath, keyPath

}

return gicRPCServiceConfig.OutCertPath + ".crt", gicRPCServiceConfig.OutCertPath + ".key"

}

type BccConfig struct {

DefaultChainID string `yaml:"defaultChainID"`

Urls map[string][]string `yaml:"urls"`

}

type BccRPCServiceConfig struct {

ServerAddr string `yaml:"serverAddr"`

UseHttps bool `yaml:"useHttps"`

OutCertPath string `yaml:"outCertPath"`

LoggerLevel string `yaml:"loggerLevel"`

}

func InitConfig(c interface{}, configFile string) error {

yamlFile, err := ioutil.ReadFile(configFile)

if err != nil {

fmt.Printf("yamlFile.Get err #%v\n ", err)

return err

}

err = yaml.Unmarshal(yamlFile, c)

if err != nil {

fmt.Printf("Unmarshal: %v\n", err)

return err

}

return nil

}

const (

genesisOrgName = "genesis"

)

func RegisterOrg(name, password string, gicParams RegisterOrgParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

contractName := "organization"

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

values := make([]interface{}, 0)

values = append(values, gicParams.OrgName)

//查询方法ID

methodID, err := QueryMethodID("genesis", contractName, "RegisterOrganization", chainID, keyStorePath, false)

if err != nil {

return

}

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, false, methodID, values)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, true, false, methodID, values)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, true, methodID, values)

}

count++

}

if result.Code == types.CodeOK {

orgIDs := new([]string)

err = json.Unmarshal([]byte(result.Data), orgIDs)

if err != nil || len(\*orgIDs) != 1 {

return

}

result.OrgID = (\*orgIDs)[0]

result.Data = ""

}

return

}

func SetOrgSigners(name, password string, gicParams SetOrgSignersParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

contractName := "organization"

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("orgName", gicParams.OrgName)

requireNotEmpty("pubKeys", gicParams.PubKeys)

pubKeys := make([]types2.HexBytes, 0)

var pubKeyStrs []string

err = jsoniter.Unmarshal([]byte(gicParams.PubKeys), &pubKeyStrs)

if err != nil {

return

}

for \_, item := range pubKeyStrs {

temp, err := hex.DecodeString(item[2:])

if err != nil {

return nil, err

}

pubKeys = append(pubKeys, temp)

}

bh := helper.BlockChainHelper{}

orgID := bh.CalcOrgID(gicParams.OrgName)

values := make([]interface{}, 0)

values = append(values, orgID)

values = append(values, pubKeys)

//查询方法ID

methodID, err := QueryMethodID("genesis", contractName, "SetSigners", chainID, keyStorePath, false)

if err != nil {

return

}

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, false, methodID, values)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, true, false, methodID, values)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, true, methodID, values)

}

count++

}

return

}

func SetOrgDeployer(name, password string, gicParams SetOrgDeployerParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

contractName := "smartcontract"

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("orgName", gicParams.OrgName)

err = algorithm.CheckAddress(chainID, gicParams.Deployer)

if err != nil {

return

}

bh := helper.BlockChainHelper{}

orgID := bh.CalcOrgID(gicParams.OrgName)

values := make([]interface{}, 0)

values = append(values, gicParams.Deployer)

values = append(values, orgID)

//查询方法ID

methodID, err := QueryMethodID("genesis", contractName, "Authorize", chainID, keyStorePath, false)

if err != nil {

return

}

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, false, methodID, values)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, false, methodID, values)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID, false, false, methodID, values)

}

count++

}

return

}

func DeployContract(name, password string, gicParams DeployContractParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

contractName := "smartcontract"

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("name", name)

requireNotEmpty("password", password)

requireNotEmpty("contractName", gicParams.ContractName)

requireNotEmpty("orgName", gicParams.OrgName)

// check arguments

err = algorithm.CheckAddress(chainID, gicParams.Owner)

if err != nil {

return

}

err = checkVersion(gicParams.Version)

if err != nil {

return

}

effectHeight, orgID, codeHash, codeData, devSig, orgSig, err := getDeployContractData(gicParams, chainID)

if err != nil {

return

}

values := make([]interface{}, 0)

values = append(values, gicParams.ContractName)

values = append(values, gicParams.Version)

values = append(values, orgID)

values = append(values, codeHash)

values = append(values, codeData)

values = append(values, devSig)

values = append(values, orgSig)

values = append(values, effectHeight)

values = append(values, gicParams.Owner)

//查询方法ID

methodID, err := QueryMethodID("genesis", contractName, "DeployContract", chainID, keyStorePath, false)

if err != nil {

return

}

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

false, false, methodID, values)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

true, false, methodID, values)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

false, true, methodID, values)

}

count++

}

if result.Code == types.CodeOK {

addrList := new([]string)

err = json.Unmarshal([]byte(result.Data), addrList)

if err != nil || len(\*addrList) != 1 {

return

}

result.SmcAddress = (\*addrList)[0]

result.Data = ""

}

return

}

func RegisterToken(name, password string, gicParams RegisterTokenParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

contractName := "token-issue"

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("tokenName", gicParams.TokenName)

requireNotEmpty("tokenSymbol", gicParams.TokenSymbol)

requireNotEmpty("totalSupply", gicParams.TotalSupply)

totalSupply, addSupplyEnabled, burnEnabled, gasPrice, err := getRegisterTokenData(gicParams)

if err != nil {

return

}

values := make([]interface{}, 0)

values = append(values, gicParams.TokenName)

values = append(values, gicParams.TokenSymbol)

values = append(values, totalSupply)

values = append(values, addSupplyEnabled)

values = append(values, burnEnabled)

values = append(values, gasPrice)

//查询方法ID

methodID, err := QueryMethodID("genesis", contractName, "NewToken", chainID, keyStorePath, false)

if err != nil {

return

}

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

false, false, methodID, values)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

true, false, methodID, values)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTx(name, password, contractName, gicParams.GasLimit, gicParams.Note, keyStorePath, chainID,

false, true, methodID, values)

}

count++

}

if result.Code == types.CodeOK {

addrList := new([]string)

err = json.Unmarshal([]byte(result.Data), addrList)

if err != nil || len(\*addrList) != 1 {

return

}

result.TokenAddress = (\*addrList)[0]

result.Data = ""

}

return

}

func Transfer(name, password string, gicParams TransferParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

\_, keyStorePath, chainID := prepare("", gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("token", gicParams.Token)

requireNotEmpty("value", gicParams.Value)

requireNotEmpty("password", password)

requireNotEmpty("gasLimit", gicParams.GasLimit)

//err = algorithm.CheckAddress(chainID, gicParams.To)

//if err != nil {

// return

//}

value, err := checkTransfer(gicParams.Value)

if err != nil {

return

}

var method uint32 = 1155058272

result, err = transfer(name, password, gicParams.Token, gicParams.GasLimit, gicParams.Note, gicParams.To, value, keyStorePath, chainID, false, method)

if err != nil {

return

}

if result.Code != types.CodeOK {

if result.Log == nonceErrDesc {

result, err = transfer(name, password, gicParams.Token, gicParams.GasLimit, gicParams.Note, gicParams.To, value, keyStorePath, chainID, true, method)

}

}

return

}

func packAndCommitTx(name, password, contractName, gasLimit, note, keyStorePath, chainID string,

bNonceErr, bSmcErr bool, methodID uint32, values []interface{}) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

nonce, err := getNonce(keyStorePath, chainID, name, password, bNonceErr)

if err != nil {

return

}

contract, err := getContract(genesisOrgName, contractName, chainID, bSmcErr, keyStorePath)

if err != nil {

return

}

uGasLimit, err := strconv.ParseUint(gasLimit, 10, 64)

if err != nil {

return

}

privStr, err := getAccountPriKey(keyStorePath, name, password)

if err != nil {

return

}

txStr := GenerateTx(contract.Address, methodID, values, nonce, int64(uGasLimit), note, privStr)

result, err = CommitTx(chainID, txStr)

return

}

//GenerateTx generate tx with one contract method request

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

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

message := types.Message{

Contract: contract,

MethodID: method,

Items: items,

}

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

return tx3.WrapTx(payload, privKey)

}

func getRegisterTokenData(gicParams RegisterTokenParam) (totalSupply bn.Number, addSupplyEnabled, burnEnabled bool, gasPrice int, err error) {

addSupplyEnabled, err = strconv.ParseBool(gicParams.AddSupplyEnabled)

if err != nil {

return

}

burnEnabled, err = strconv.ParseBool(gicParams.BurnEnabled)

if err != nil {

return

}

totalSupply = bn.NewNumberStringBase(gicParams.TotalSupply, 10)

if totalSupply.IsLEI(0) {

err = errors.New("invalid totalSupply")

return

}

gasPrice, err = strconv.Atoi(gicParams.GasPrice)

if err != nil {

return

}

return

}

func getDeployContractData(gicParams DeployContractParam, chainID string) (

effectHeightInt int,

orgID string,

codeHash []byte,

codeData []byte,

devSigStr string,

orgSigStr string,

err error) {

// setup data

codeData, err = ioutil.ReadFile(gicParams.CodeFile)

if err != nil {

return

}

devSigStr, err = getSigStr(gicParams.CodeFile + ".sig")

if err != nil {

return

}

orgSigStr, err = getSigStr(gicParams.CodeFile + ".sig.sig")

if err != nil {

return

}

codeHash = sha3.Sum256(codeData)

blh, err := BlockHeight(chainID)

if err != nil {

return

}

HeightInt, err := strconv.Atoi(gicParams.EffectHeight)

if err != nil {

return

}

effectHeightInt = int(blh.LastBlock) + HeightInt

bh := helper.BlockChainHelper{}

orgID = bh.CalcOrgID(gicParams.OrgName)

return

}

func makeParams(values ...interface{}) []interface{} {

gicParamss := make([]interface{}, 0)

for \_, v := range values {

gicParamss = append(gicParamss, v)

}

return gicParamss

}

func getSigStr(path string) (s string, err error) {

type signature struct {

PubKey string `json:"pubkey"`

Signature string `json:"signature"`

}

sigData, err := ioutil.ReadFile(path)

if err != nil {

return

}

devSig := new(signature)

err = json.Unmarshal(sigData, devSig)

if err != nil {

return

}

signaBtyes, err := json.Marshal(devSig)

if err != nil {

return

}

return string(signaBtyes), nil

}

func SolDeploy(name, password string, bvmParam BVMDeployParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

\_, keyStorePath, chainID := prepare("", bvmParam.KeyStorePath, bvmParam.ChainID)

// require not empty

requireNotEmpty("name", name)

requireNotEmpty("password", password)

if bvmParam.TokenAddr == "" && bvmParam.TokenName == "" {

err = errors.New("tokenAddr and tokenName can not all be empty")

return

}

addrS := nodeAddrSlice(chainID)

if bvmParam.TokenName != "" {

var tokenAddr types.Address

key := std.KeyOfTokenWithName(bvmParam.TokenName)

err = common.DoHttpQueryAndParse(addrS, key, &tokenAddr)

if err != nil {

return nil, errors.New("tokenName is right? error: " + err.Error())

}

bvmParam.TokenAddr = tokenAddr

} else {

// 添加代币地址验证

key := std.KeyOfToken(bvmParam.TokenAddr)

result := new(std.Token)

err = common.DoHttpQueryAndParse(addrS, key, &result)

if err != nil {

return nil, errors.New("tokenAddr is right? error: " + err.Error())

}

}

codeData := make([]byte, 0)

// read the sol contract binFile

if bvmParam.BinFile != "" {

codeData, err = ioutil.ReadFile(bvmParam.BinFile)

if err != nil {

return

}

}

// Compile the sol contract

if bvmParam.SourceFile != "" && bvmParam.BinFile == "" {

codeDataStr, err := CompileSol(bvmParam.SourceFile)

if err != nil {

return nil, errors.New("sol contract compile failed, error: " + err.Error())

}

codeData = []byte(codeDataStr)

}

codeDataStr := string(codeData)

newCodeData, err := hex.DecodeString(codeDataStr)

if err != nil {

fmt.Println(err.Error())

return

}

Array := make([]interface{}, 0)

for i := 0; i < len(bvmParam.ParamsArray); i++ {

Array = append(Array, bvmParam.ParamsArray[i])

}

newAbi, abiStr, err := GetAbiObject(bvmParam.AbiFile, "")

if err != nil {

return nil, err

}

newAbiData := []byte(abiStr)

if len(bvmParam.ParamsArray) > 0 {

param, err := PackParams(newAbi, "", Array...)

if err != nil {

return nil, err

}

newCodeData = append(newCodeData, param...)

}

result, err = packAndCommitTxForBVM(name, password, "", bvmParam.TokenAddr, bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

false, true, 0, newCodeData, nil, newAbiData)

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = packAndCommitTxForBVM(name, password, "", bvmParam.TokenAddr, bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

true, true, 0, newCodeData, nil, newAbiData)

} else if result.Log == smcErrDesc {

result, err = packAndCommitTxForBVM(name, password, "", bvmParam.TokenAddr, bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

false, true, 0, newCodeData, nil, newAbiData)

}

count++

}

return

}

func SolCall(name, password string, bvmParam BVMCallParam) (result \*BVMCallResult, err error) {

defer FuncRecover(&err)

\_, keyStorePath, chainID := prepare("", bvmParam.KeyStorePath, bvmParam.ChainID)

// require not empty

requireNotEmpty("name", name)

requireNotEmpty("password", password)

requireNotEmpty("method", bvmParam.Method)

requireNotEmpty("contractAddr", bvmParam.ContractAddr)

bvmParams := make([]byte, 0)

Array := make([]interface{}, 0)

for i := 0; i < len(bvmParam.ParamsArray); i++ {

Array = append(Array, bvmParam.ParamsArray[i])

}

newAbi, \_, err := GetAbiObject(bvmParam.AbiFile, bvmParam.ContractAddr)

if err != nil {

return nil, err

}

param, err := PackParams(newAbi, bvmParam.Method, Array...)

if err != nil {

return nil, err

}

bvmParams = append(bvmParams, param...)

transParams := make([]byte, 0)

transMethodID := uint32(0)

value, \_ := strconv.Atoi(bvmParam.Value)

if value > 0 {

transMethodID = 1155058272

transParams = []byte(bvmParam.Value)

}

res, err := packAndCommitTxForBVM(name, password, bvmParam.ContractAddr, "", bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

false, false, transMethodID, bvmParams, transParams, nil)

var count = 0

for (res.Code != types.CodeOK || res.Code != types.CodeBVMQueryOK) && count < 2 {

if res.Log == nonceErrDesc {

res, err = packAndCommitTxForBVM(name, password, bvmParam.ContractAddr, "", bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

true, false, transMethodID, bvmParams, transParams, nil)

} else if res.Log == smcErrDesc {

res, err = packAndCommitTxForBVM(name, password, bvmParam.ContractAddr, "", bvmParam.GasLimit, bvmParam.Note, keyStorePath, chainID,

false, false, transMethodID, bvmParams, transParams, nil)

}

count++

}

if res == nil {

return nil, err

}

result, err = ParseResult(newAbi, bvmParam, \*res)

return

}

func ParseResult(Abi abi.ABI, bvmParam BVMCallParam, res CommitTxResult) (result \*BVMCallResult, err error) {

result = new(BVMCallResult)

if res.Data != "" {

v := make(map[string]interface{})

data, err := hex.DecodeString(res.Data)

if err != nil {

fmt.Println("Data decode failed!")

return nil, err

}

err = Abi.UnpackIntoMap(v, bvmParam.Method, data)

if err != nil {

fmt.Println("Data unpack failed!")

return nil, err

}

result.Data = v

}

if res.Height != 0 || res.Fee != 0 {

result.Height = res.Height

result.Fee = res.Fee

}

result.Log = res.Log

result.Code = res.Code

result.TxHash = res.TxHash

return

}

func packAndCommitTxForBVM(name, password, contractAddr, tokenAddr, gasLimit, note, keyStorePath, chainID string,

bNonceErr, IsCreateCall bool, TransMethodID uint32, BVMParams, transParams, BVMAbi []byte) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

nonce, err := getNonce(keyStorePath, chainID, name, password, bNonceErr)

if err != nil {

return

}

Messages := tx3.PrepareMessages(contractAddr, tokenAddr, TransMethodID, transParams, BVMParams, BVMAbi, IsCreateCall)

uGasLimit, err := strconv.ParseUint(gasLimit, 10, 64)

if err != nil {

return

}

privStr, err := getAccountPriKey(keyStorePath, name, password)

if err != nil {

return

}

payLoad := tx3.WrapPayload(nonce, int64(uGasLimit), note, Messages...)

txStr := tx3.WrapTx(payLoad, privStr)

result, err = CommitTx(chainID, txStr)

return

}

// CompileSol - compile sol contract source file

func CompileSol(path string) (string, error) {

systemStr := runtime.GOOS

if systemStr == "windows" {

cmd := exec.Command("cmd", "/C", "--bin "+path, "solc.exe")

cmd.Stdin = strings.NewReader("some input")

var out bytes.Buffer

cmd.Stdout = &out

err := cmd.Run()

if err != nil {

fmt.Println("CompileSol failed, please check ! Error: ", err)

return "", err

}

str := out.String()

strs := strings.Split(str, "Binary:")

str2 := strings.Replace(strings.Replace(strs[1], " ", "", -1), "\n", "", -1)

return str2, err

} else {

cmd := exec.Command("/bin/sh", "-c", "./solc --bin "+path)

cmd.Stdin = strings.NewReader("some input")

var out bytes.Buffer

cmd.Stdout = &out

err := cmd.Run()

if err != nil {

fmt.Println("CompileSol failed, please check ! Error: ", err)

return "", err

}

str := out.String()

strs := strings.Split(str, "Binary:")

str2 := strings.Replace(strings.Replace(strs[1], " ", "", -1), "\n", "", -1)

return str2, err

}

}

func MustDecodeString(str string) []byte {

dst, err := hex.DecodeString(str)

if err != nil {

panic(err)

}

return dst

}

// Call call contract's method with params, save params to paramsFile if it's length too long

func Call(name, password string, gicParams CallParam) (result \*CommitTxResult, err error) {

defer FuncRecover(&err)

// reset value or not

splitBy, keyStorePath, chainID := prepare(gicParams.SplitBy, gicParams.KeyStorePath, gicParams.ChainID)

// require not empty

requireNotEmpty("orgName", gicParams.OrgName)

requireNotEmpty("contractName", gicParams.Contract)

requireNotEmpty("name", name)

requireNotEmpty("password", password)

requireNotEmpty("methodName", gicParams.Method)

// check pay

value, token, err := checkPay(gicParams.Pay)

if err != nil {

return

}

result, err = call(name, password, gicParams.OrgName, gicParams.Contract, gicParams.Method, gicParams.ParamsFile,

gicParams.Params, splitBy, token, value, gicParams.GasLimit, gicParams.Note, keyStorePath,

chainID, false, false)

if err != nil {

return

}

var count = 0

for result.Code != types.CodeOK && count < 2 {

if result.Log == nonceErrDesc {

result, err = call(name, password, gicParams.OrgName, gicParams.Contract, gicParams.Method, gicParams.ParamsFile,

gicParams.Params, splitBy, token, value, gicParams.GasLimit, gicParams.Note, keyStorePath,

chainID, true, false)

} else if result.Log == smcErrDesc {

result, err = call(name, password, gicParams.OrgName, gicParams.Contract, gicParams.Method, gicParams.ParamsFile,

gicParams.Params, splitBy, token, value, gicParams.GasLimit, gicParams.Note, keyStorePath,

chainID, false, false)

}

count++

}

return

}

func call(name, password, orgName, contractName, methodName, file, params, splitBy, token string, value bn.Number,

gasLimit, note, keyStorePath, chainID string, bNonceErr, bSmcErr bool) (result \*CommitTxResult, err error) {

// get account transaction nonce

nonce, err := getNonce(keyStorePath, chainID, name, password, bNonceErr)

if err != nil {

return nil, errors.New("getNonce error: " + err.Error())

}

// get contract information with orgName and contractName

contract, err := getContract(orgName, contractName, chainID, bSmcErr, keyStorePath)

if err != nil {

return nil, errors.New("getContract error: " + err.Error())

}

var item std.Method

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

if strings.HasPrefix(methodItem.ProtoType, methodName+"(") {

item = methodItem

break

}

}

if len(item.MethodID) == 0 {

return nil, errors.New("invalid method")

}

arrayMethod := strings.Split(item.ProtoType, "(")

rlpBytes := make([]common.HexBytes, 0)

if arrayMethod[1][:1] != ")" {

// encode method parameters

rlpBytes, err = encode(item, splitBy, file, params)

if err != nil {

return

}

}

uGasLimit, err := requireUint64("gasLimit", gasLimit, 10)

if err != nil {

return

}

methodID, \_ := requireUint64("methodID", item.MethodID, 16)

var msgList []types.Message

// pack tx

// if pay option not empty, then create transfer message

if value.IsGreaterThanI(0) {

transferMsg, err := createTransferMsg(contract, value, token, chainID)

if err != nil {

return nil, err

}

msgList = append(msgList, transferMsg)

}

msg := types.Message{Contract: contract.Address, MethodID: uint32(methodID), Items: rlpBytes}

msgList = append(msgList, msg)

pl := tx3.WrapPayload(nonce, int64(uGasLimit), note, msgList...)

priKeyHex, err := getAccountPriKey(keyStorePath, name, password)

if err != nil {

return

}

txStr := tx3.WrapTxEx(chainID, pl, priKeyHex)

// commit transaction

result, err = CommitTx(chainID, txStr)

if err != nil {

return

}

return

}

func getNonce(keyStorePath, chainID, name, password string, bNonceErr bool) (nonce uint64, err error) {

nonce, err = cache.Nonce(name, keyStorePath)

if err != nil || bNonceErr {

var nonceResult \*NonceResult

nonceResult, err = Nonce("", name, password, chainID, keyStorePath)

if err != nil {

return

}

nonce = nonceResult.Nonce

err = cache.SetNonce(name, nonce, keyStorePath)

if err != nil {

return

}

}

return

}

func getContract(orgName, contractName, chainID string, bSmcErr bool, keyStorePath string) (contract \*std.Contract, err error) {

bh := helper.BlockChainHelper{}

orgID := bh.CalcOrgID(orgName)

contract, err = cache.Contract(orgID, contractName, keyStorePath)

if err != nil || bSmcErr {

contract, err = contractOfName(chainID, orgID, contractName)

if err != nil {

return

}

err = cache.SetContract(contract, keyStorePath)

if err != nil {

return

}

}

return

}

// 生成转账message

func createTransferMsg(contract \*std.Contract, value bn.Number, token, chainID string) (transferMsg types.Message, err error) {

rlpBytes := tx2.WrapInvokeParams(contract.Account, value)

tokenAddress, err := tokenAddressFromName(chainID, token)

if err != nil {

return

}

transferMsg = types.Message{Contract: tokenAddress, MethodID: 0x44d8ca60, Items: rlpBytes}

return

}

func readParamFile(file string) (params string, err error) {

b, err := ioutil.ReadFile(file)

if err != nil {

return

}

params = string(b)

return

}

// 查询合约信息

func QueryContractInfo(OrgName, ContractName, chainID, keyStorePath string) (contract \*std.Contract, err error) {

contract, err = getContract(OrgName, ContractName, chainID, false, keyStorePath)

if err != nil {

return

}

if contract.Methods == nil {

contract, err = getContract(OrgName, ContractName, chainID, true, keyStorePath)

if err != nil {

return

}

}

return

}

func nodeAddrSlice(chainID string) []string {

if len(chainID) == 0 {

chainID = common.GetBCCConfig().DefaultChainID

}

if \_, ok := common.GetBCCConfig().Urls[chainID]; ok {

return common.GetBCCConfig().Urls[chainID]

}

return []string{}

}

func getAccountPriKey(keyStorePath, name, password string) (priKeyHex string, err error) {

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

return

}

priKey := acct.PrivateKey.(crypto.PrivKeyEd25519)

return "0x" + hex.EncodeToString(priKey[:]), nil

}

func FuncRecover(errPtr \*error) {

if err := recover(); err != nil {

msg := ""

if errInfo, ok := err.(error); ok {

msg = errInfo.Error()

}

if errInfo, ok := err.(string); ok {

msg = errInfo

}

\*errPtr = errors.New(msg)

}

}

func prepare(splitBy, keyStorePath, chainID string) (string, string, string) {

if splitBy == "" {

splitBy = "@"

}

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if chainID == "" {

chainID = common.GetBCCConfig().DefaultChainID

}

crypto.SetChainId(chainID)

tx2.Init(chainID)

tx3.Init(chainID)

return splitBy, keyStorePath, chainID

}

func requireNotEmpty(key, data string) {

if len(data) == 0 {

panic(errors.New(fmt.Sprintf("%s cannot be emtpy", key)))

}

}

func requireUint64(key, valueStr string, base int) (uint64, error) {

value, err := strconv.ParseUint(valueStr, base, 64)

if err != nil {

return 0, errors.New(fmt.Sprintf("%s error=%s", key, err.Error()))

}

return value, nil

}

func checkPay(pay string) (value bn.Number, token string, err error) {

token = ""

value = bn.N(0)

if len(pay) > 0 {

// step 1. check format

firstIndex := strings.Index(pay, "(")

lastIndex := strings.Index(pay, ")")

if firstIndex == -1 || lastIndex < firstIndex {

err = errors.New("pay option's format error, right format example: 1.02(gi)")

return

}

// step 2. check token

token = pay[firstIndex+1 : lastIndex]

if len(token) <= 0 {

err = errors.New("pay option's format error, token cannot be empty")

return

}

// step 3. check value

valueStr := pay[:firstIndex]

potIndex := strings.Index(valueStr, ".")

if potIndex != -1 && len(strings.TrimRight(valueStr[potIndex+1:], "0")) > 9 {

err = errors.New("pay option's format error, value's decimals cannot great than 9 chars")

return

}

zeroCount := 0

valueStr = strings.Replace(valueStr, ".", "", -1)

if potIndex == -1 {

zeroCount = 9

} else {

zeroCount = 9 - (len(valueStr) - potIndex)

}

for zeroCount > 0 {

valueStr += "0"

zeroCount--

}

value = bn.NewNumberStringBase(valueStr, 10)

if value.IsLEI(0) {

err = errors.New("pay option's format error, value must be number and greater than zero")

}

}

return

}

func checkTransfer(value string) (validValue string, err error) {

potIndex := strings.Index(value, ".")

if potIndex != -1 && len(strings.TrimRight(value[potIndex+1:], "0")) > 9 {

err = errors.New("pay option's format error, value's decimals cannot great than 9 chars")

return "", err

}

zeroCount := 0

validValue = strings.Replace(value, ".", "", -1)

if potIndex == -1 {

zeroCount = 9

} else {

zeroCount = 9 - (len(validValue) - potIndex)

}

for zeroCount > 0 {

validValue += "0"

zeroCount--

}

return

}

func checkVersion(version string) (err error) {

if len(version) < 3 {

return errors.New("invalid version")

}

if len(strings.Trim(version, ".")) != len(version) {

return errors.New("invalid version")

}

verStr := strings.Replace(version, ".", "", -1)

verN := bn.NewNumberStringBase(verStr, 10)

if verN.IsLessThanI(0) {

return errors.New("invalid version")

}

return

}

// CheckUTF8 check format

func CheckUTF8(buf []byte) bool {

nBytes := 0

for i := 0; i < len(buf); i++ {

if nBytes == 0 {

if (buf[i] & 0x80) != 0 {

for (buf[i] & 0x80) != 0 {

buf[i] <<= 1

nBytes++

}

if nBytes < 2 || nBytes > 6 {

return false

}

nBytes--

}

} else {

if buf[i]&0xc0 != 0x80 {

return false

}

nBytes--

}

}

return nBytes == 0

}

// 查询方法ID

func QueryMethodID(orgName, contractName, method, chainID, keyStorePath string, bSmcErr bool) (uint32, error) {

contract, err := getContract(orgName, contractName, chainID, bSmcErr, keyStorePath)

if err != nil {

return 0, errors.New("getContract error: " + err.Error())

}

var item std.Method

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

if strings.HasPrefix(methodItem.ProtoType, method) {

item = methodItem

break

}

}

if len(item.MethodID) == 0 {

return 0, errors.New("invalid method")

}

methodid, \_ := requireUint64("methodID", item.MethodID, 16)

return uint32(methodid), nil

}

func GetAbiObject(path, addr string) (abi2 abi.ABI, abiStr string, err error) {

// read the sol contract abiFile

if path != "" {

codeData, err := ioutil.ReadFile(path)

if err != nil {

newAbi := new(abi.ABI)

return \*newAbi, "", err

}

abiStr = string(codeData)

abi2, err = abi.JSON(strings.NewReader(abiStr))

if err != nil {

newAbi := new(abi.ABI)

return \*newAbi, "", err

}

} else {

addrs := nodeAddrSlice(common.GetBCCConfig().DefaultChainID)

res := new(std.BvmContract)

err = common.DoHttpQueryAndParse(addrs, "/bvm/contract/"+addr, &res)

if err != nil {

return

}

abi2, err = abi.JSON(strings.NewReader(res.BvmAbi))

if err != nil {

return

}

}

return

}

func PackParams(abi2 abi.ABI, method string, param ...interface{}) ([]byte, error) {

var length int

newParam := make([]interface{}, 0)

// 合约部署参数构造

if method == "" {

length = len(abi2.Constructor.Inputs)

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

paramType := abi2.Constructor.Inputs[i].Type.String()

newParam = append(newParam, DetermineType(paramType, param[i]))

}

} else {

// 合约调用参数构造

length = len(abi2.Methods[method].Inputs)

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

paramType := abi2.Methods[method].Inputs[i].Type.String()

newParam = append(newParam, DetermineType(paramType, param[i]))

}

}

paramBin, err := abi2.Pack(method, newParam...)

if err != nil {

return nil, err

}

return paramBin, nil

}

func DetermineType(paramType string, param interface{}) interface{} {

if strings.HasPrefix(paramType, "bytes") {

paramType = "bytes"

}

if strings.HasPrefix(paramType, "int") {

if !strings.HasSuffix(paramType, "]") {

paramType = "int"

} else if strings.Contains(paramType, "][") {

paramType = "int[][]"

} else {

paramType = "int[]"

}

}

if strings.HasPrefix(paramType, "uint") {

if !strings.HasSuffix(paramType, "]") {

paramType = "uint"

} else if strings.Contains(paramType, "][") {

paramType = "uint[][]"

} else {

paramType = "uint[]"

}

}

switch paramType {

case "bool":

all, err := strconv.ParseBool(param.(string))

if err != nil {

return err

} else {

return all

}

case "int", "int8", "int32", "int64", "int256":

i, \_ := strconv.Atoi(param.(string))

return int64(i)

case "uint", "uint8", "uint32", "uint64", "uint256":

i, \_ := strconv.Atoi(param.(string))

return uint64(i)

case "int[]", "int8[]", "int32[]", "int64[]", "int256[]":

Islice := make([]int64, 0)

slices := param.(string)

str := strings.Split(slices[1:len(slices)-1], ",")

for \_, v := range str {

i, \_ := strconv.Atoi(v)

Islice = append(Islice, int64(i))

}

return Islice

case "uint[]", "uint8[]", "uint32[]", "uint64[]", "uint256[]":

Uslice := make([]uint64, 0)

slices := param.(string)

str := strings.Split(slices[1:len(slices)-1], ",")

for \_, v := range str {

i, \_ := strconv.Atoi(v)

Uslice = append(Uslice, uint64(i))

}

return Uslice

case "int[][]":

return GetBetweenStrInt(param.(string))

case "uint[][]":

return GetBetweenStrUint(param.(string))

case "string":

return param.(string)

case "bytes":

return []byte(param.(string))

case "byte[]":

return []byte(param.(string))

case "byte[][]":

return GetBetweenStrByte(param.(string))

case "address":

addr := param.(string)

err := algorithm.CheckAddress(crypto.GetChainId(), addr)

if err != nil {

fmt.Println("Invalid address")

return nil

}

address := crypto2.ToBVM(addr).Bytes()

return abi.BytesToAddress(address)

case "address[]":

AddrSlice := make([]abi.Address, 0)

slices := param.(string)

str := strings.Split(slices[1:len(slices)-1], ",")

for \_, v := range str {

err := algorithm.CheckAddress(crypto.GetChainId(), v)

if err != nil {

fmt.Println("Invalid address")

return nil

}

address := crypto2.ToBVM(v).Bytes()

AddrSlice = append(AddrSlice, abi.BytesToAddress(address))

}

return AddrSlice

case "address[][]":

return GetBetweenStrAddr(param.(string))

default:

fmt.Println("The input parameter is invalid, please check！")

return nil

}

}

func GetBetweenStrUint(str string) (doubleSlice [][]uint64) {

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

reg := regexp.MustCompile(`\[(.\*?)\]`)

if reg != nil {

doubleSliceStr = reg.FindAllStringSubmatch(str, -1)

}

for i := 0; i < len(doubleSliceStr); i++ {

newSlice := make([]uint64, 0)

newStr := strings.Split(doubleSliceStr[i][1], ",")

for \_, v := range newStr {

newUint, \_ := strconv.Atoi(v)

newSlice = append(newSlice, uint64(newUint))

}

doubleSlice = append(doubleSlice, newSlice)

}

return

}

func GetBetweenStrInt(str string) (doubleSlice [][]int64) {

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

reg := regexp.MustCompile(`\[(.\*?)\]`)

if reg != nil {

doubleSliceStr = reg.FindAllStringSubmatch(str, -1)

}

for i := 0; i < len(doubleSliceStr); i++ {

newSlice := make([]int64, 0)

newStr := strings.Split(doubleSliceStr[i][1], ",")

for \_, v := range newStr {

newUint, \_ := strconv.Atoi(v)

newSlice = append(newSlice, int64(newUint))

}

doubleSlice = append(doubleSlice, newSlice)

}

return

}

func GetBetweenStrByte(str string) (doubleSlice [][]byte) {

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

reg := regexp.MustCompile(`\[(.\*?)\]`)

if reg != nil {

doubleSliceStr = reg.FindAllStringSubmatch(str, -1)

}

for i := 0; i < len(doubleSliceStr); i++ {

newStr := strings.Split(doubleSliceStr[i][1], ",")

for \_, v := range newStr {

doubleSlice = append(doubleSlice, []byte(v))

}

}

return

}

func GetBetweenStrAddr(str string) (doubleSlice [][]abi.Address) {

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

reg := regexp.MustCompile(`\[(.\*?)\]`)

if reg != nil {

doubleSliceStr = reg.FindAllStringSubmatch(str, -1)

}

for i := 0; i < len(doubleSliceStr); i++ {

newSlice := make([]abi.Address, 0)

newStr := strings.Split(doubleSliceStr[i][1], ",")

for \_, v := range newStr {

err := algorithm.CheckAddress(crypto.GetChainId(), v)

if err != nil {

fmt.Println("Invalid address")

return nil

}

address := crypto2.ToBVM(v).Bytes()

newSlice = append(newSlice, abi.BytesToAddress(address))

}

doubleSlice = append(doubleSlice, newSlice)

}

return

}

//Query query state db with path

func Query(path string, data []byte, height int64, trusted bool, chainID string) (query \*types2.ResultABCIQuery, err error) {

return

}

//BlockHeight blockHeight query

func BlockHeight(chainID string) (blkHeight \*BlockHeightResult, err error) {

defer FuncRecover(&err)

addrS := nodeAddrSlice(chainID)

result := new(core\_types.ResultABCIInfo)

params := map[string]interface{}{}

err = common.DoHttpRequestAndParseExBlock(addrS, "abci\_info", params, result)

if err != nil {

return

}

blkHeight = new(BlockHeightResult)

blkHeight.LastBlock = result.Response.LastBlockHeight

return

}

//Block block information query

func Block(height \*int64, bTime string, num \*int64, chainID string) (blk \*BlockResult, err error) {

\_, \_, chainID = prepare("", "", chainID)

if height == nil && bTime == "" {

blkResult, err := BlockHeight(chainID)

if err != nil {

return nil, err

}

height = &blkResult.LastBlock

}

if height != nil {

return block(\*height, chainID)

} else {

return blockEx(bTime, num, chainID)

}

}

func BlockForRpc(height int64, bTime string, num int64, chainID string) (blk \*BlockResult, err error) {

return Block(&height, bTime, &num, chainID)

}

//block block information query

func block(height int64, chainID string) (blk \*BlockResult, err error) {

defer FuncRecover(&err)

addrS := nodeAddrSlice(chainID)

result := new(core\_types.ResultBlock)

params := map[string]interface{}{"height": height}

err = common.DoHttpRequestAndParseExBlock(addrS, "block", params, result)

if err != nil {

return

}

blk = new(BlockResult)

blk.BlockHeight = result.BlockMeta.Header.Height

blk.BlockHash = "0x" + hex.EncodeToString(result.BlockMeta.BlockID.Hash)

blk.ParentHash = "0x" + hex.EncodeToString(result.BlockMeta.Header.LastBlockID.Hash)

blk.ChainID = result.BlockMeta.Header.ChainID

blk.ValidatorHash = "0x" + hex.EncodeToString(result.BlockMeta.Header.ValidatorsHash)

blk.ConsensusHash = "0x" + hex.EncodeToString(result.BlockMeta.Header.ConsensusHash)

blk.BlockTime = result.BlockMeta.Header.Time.String()

blk.BlockSize = result.BlockSize

blk.ProposerAddress = result.BlockMeta.Header.ProposerAddress

blk.Txs = make([]TxResult, 0)

var blkResults \*core\_types.ResultBlockResults

if blkResults, err = blockResults(chainID, height); err != nil {

return blk, nil

}

blk.Txs = make([]TxResult, 0)

for k, ResDeliver := range blkResults.Results.DeliverTx {

var tx \*TxResult

if tx, err = transactionBlock(k, ResDeliver, result); err != nil {

return

}

blk.Txs = append(blk.Txs, \*tx)

}

return

}

//blockEx block information query

func blockEx(bTime string, num \*int64, chainID string) (blk \*BlockResult, err error) {

addrS := nodeAddrSlice(chainID)

cache.SetAddrList(addrS)

t, err := time.ParseInLocation("2006-01-02 15:04:05", bTime, time.UTC)

if err != nil {

return

}

nearlyHeight := cache.BinarySearchEx(0, 0, 128, t)

if nearlyHeight <= 0 {

return nil, errors.New("cannot find nearly block about time=" + bTime)

}

// if it less than require time then nearlyHeight plus one

if cache.CompareWithTime(nearlyHeight, t, false) == -1 {

nearlyHeight += 1

}

if num != nil && \*num > 1 {

return blockSimpleResult(nearlyHeight, \*num, addrS)

} else {

return block(nearlyHeight, chainID)

}

}

// blockSimpleResult get simple block result from h to h+num

func blockSimpleResult(h, num int64, addrS []string) (blk \*BlockResult, err error) {

blk = new(BlockResult)

blk.Result = make([]simpleBlockResult, 0)

result := new(core\_types.ResultBlock)

index := int64(0)

for index < num {

err = common.DoHttpRequestAndParse(addrS, "block", map[string]interface{}{"height": h + index}, result)

if err != nil {

return

}

blk.Result = append(blk.Result, simpleBlockResult{

BlockHeight: h + index,

BlockHash: "0x" + result.BlockMeta.BlockID.Hash.String(),

BlockTime: result.BlockMeta.Header.Time.String()})

index++

}

return

}

//Transaction transaction information query

func transactionBlock(k int, ResDeliver \*types3.ResponseDeliverTx, resultBlock \*core\_types.ResultBlock) (tx \*TxResult, err error) {

//ParseTX

var (

transaction tx1.Transaction

fromAddr string

msg Message

GasLimit uint64

Nonce uint64

Note string

)

messages := make([]Message, 0)

txStr := string(resultBlock.Block.Txs[k])

splitTx := strings.Split(txStr, ".")

if splitTx[1] == "v1" {

// parse transaction V1

fromAddr, \_, err = transaction.TxParse(crypto.GetChainId(), txStr)

if err != nil {

return

}

msg, err = messageV1Parse(transaction)

if err != nil {

return

}

messages = append(messages, msg)

GasLimit = transaction.GasLimit

Nonce = transaction.Nonce

Note = transaction.Note

} else if splitTx[1] == "v2" {

// parse transaction V2

var txv2 types.Transaction

var pubKey crypto.PubKeyEd25519

txv2, pubKey, err := tx2.TxParse(txStr)

if err != nil {

txv2, pubKey, err = tx3.TxParse(txStr)

if err != nil {

return nil, err

}

}

fromAddr = pubKey.Address(crypto.GetChainId())

var msg Message

for i := 0; i < len(txv2.Messages); i++ {

msg, err = messageV2Parse(txv2.Messages[i])

if err != nil {

return nil, err

}

messages = append(messages, msg)

}

GasLimit = uint64(txv2.GasLimit)

Nonce = txv2.Nonce

Note = txv2.Note

} else {

err = errors.New("unsupported tx=" + txStr)

return

}

tx = new(TxResult)

tx.TxHash = "0x" + strings.ToLower(hex.EncodeToString(ResDeliver.TxHash))

tx.TxTime = resultBlock.BlockMeta.Header.Time.String()

tx.Code = ResDeliver.Code

tx.Log = ResDeliver.Log

tx.BlockHash = "0x" + hex.EncodeToString(resultBlock.BlockMeta.BlockID.Hash)

tx.BlockHeight = resultBlock.BlockMeta.Header.Height

tx.From = fromAddr

tx.Nonce = Nonce

tx.GasLimit = GasLimit

tx.Fee = ResDeliver.Fee

tx.Note = Note

tx.Messages = messages

tx.Tags = make(map[string]\*Tag)

for \_, item := range ResDeliver.Tags {

Tag := new(Tag)

Tag.Receipt = make(map[string]interface{})

Receipt := make(map[string]interface{})

err = json.Unmarshal(item.Value, &Tag)

if err != nil {

return nil, err

}

aDec, err := base64.StdEncoding.DecodeString(Tag.ReceiptBytes)

err = json.Unmarshal(aDec, &Receipt)

if err != nil {

return nil, err

}

Tag.Receipt = Receipt

Tag.ReceiptHash = "0x" + Tag.ReceiptHash

tx.Tags[string(item.Key)] = Tag

}

return

}

func messageV1Parse(transation tx1.Transaction) (msg Message, err error) {

var methodInfo tx1.MethodInfo

if err = rlp.DecodeBytes(transation.Data, &methodInfo); err != nil {

return

}

methodID := fmt.Sprintf("%x", methodInfo.MethodID)

msg.SmcAddress = transation.To

if msg.SmcName, msg.Method, err = contractNameAndMethod2(transation.To, methodID); err != nil {

return

}

if methodID == transferMethodIDV1 {

var itemsBytes = make([][]byte, 0)

if err = rlp.DecodeBytes(methodInfo.ParamData, &itemsBytes); err != nil {

return

}

msg.To = string(itemsBytes[0])

msg.Value = new(big.Int).SetBytes(itemsBytes[1][:]).String()

}

return

}

func messageV2Parse(message types.Message) (msg Message, err error) {

methodID := fmt.Sprintf("%x", message.MethodID)

msg.SmcAddress = message.Contract

if msg.SmcName, msg.Method, err = contractNameAndMethod2(message.Contract, methodID); err != nil {

return

}

if methodID == transferMethodIDV2 {

if len(message.Items) != 2 {

return msg, errors.New("items count error")

}

var to types.Address

if err = rlp.DecodeBytes(message.Items[0], &to); err != nil {

return

}

var value bn.Number

if err = rlp.DecodeBytes(message.Items[1], &value); err != nil {

return

}

msg.To = to

msg.Value = value.String()

msg.Method = "Transfer(types.Address,bn.Number)"

}

return

}

func Transaction(chainID, txHash string, resultBlock \*core\_types.ResultBlock) (tx \*TxResult, err error) {

defer FuncRecover(&err)

requireNotEmpty("txHash", txHash)

if txHash[:2] == "0x" {

txHash = txHash[2:]

}

if chainID == "" {

chainID = common.GetBCCConfig().DefaultChainID

}

tx2.Init(chainID)

tx3.Init(chainID)

crypto.SetChainId(chainID)

addrS := nodeAddrSlice(chainID)

result := new(core\_types.ResultTx)

params := map[string]interface{}{"hash": txHash}

err = common.DoHttpRequestAndParse(addrS, "tx", params, result)

if err != nil {

return

}

if resultBlock == nil {

resultBlock = new(core\_types.ResultBlock)

params = map[string]interface{}{"height": result.Height}

err = common.DoHttpRequestAndParse(addrS, "block", params, resultBlock)

if err != nil {

return

}

}

var txStr string

var blkResults \*core\_types.ResultBlockResults

if blkResults, err = blockResults(chainID, result.Height); err != nil {

return

}

for k, v := range blkResults.Results.DeliverTx {

hash := hex.EncodeToString(v.TxHash)

if hash[:2] == "0x" {

txHash = txHash[2:]

}

if strings.ToLower(txHash) == hash {

txStr = string(resultBlock.Block.Txs[k])

}

}

nonce, gasLimit, fromAddr, note, messages, err := parseTx(chainID, txStr, resultBlock.Block.Height, resultBlock.Block.ChainVersion)

if err != nil {

return

}

tx = new(TxResult)

tx.TxHash = "0x" + txHash

tx.TxTime = resultBlock.BlockMeta.Header.Time.String()

tx.Code = result.DeliverResult.Code

tx.Log = result.DeliverResult.Log

tx.BlockHash = "0x" + hex.EncodeToString(resultBlock.BlockMeta.BlockID.Hash)

tx.BlockHeight = resultBlock.BlockMeta.Header.Height

tx.From = fromAddr

tx.Nonce = nonce

tx.GasLimit = gasLimit

tx.Fee = result.DeliverResult.Fee

tx.Note = note

tx.Messages = messages

tx.Tags = make(map[string]\*Tag)

for \_, item := range result.DeliverResult.Tags {

Tag := new(Tag)

Tag.Receipt = make(map[string]interface{})

Receipt := make(map[string]interface{})

err = json.Unmarshal(item.Value, &Tag)

if err != nil {

return nil, err

}

aDec, err := base64.StdEncoding.DecodeString(Tag.ReceiptBytes)

err = json.Unmarshal(aDec, &Receipt)

if err != nil {

return nil, err

}

Tag.Receipt = Receipt

Tag.ReceiptHash = "0x" + Tag.ReceiptHash

tx.Tags[string(item.Key)] = Tag

}

return

}

func parseTx(chainID, txStr string, height int64, chainVersion \*int64) (nonce, gasLimit uint64, fromAddr, note string, messages []Message, err error) {

messages = make([]Message, 0)

splitTx := strings.Split(txStr, ".")

if splitTx[1] == "v1" {

var txv1 tx1.Transaction

fromAddr, \_, err = txv1.TxParse(chainID, txStr)

if err != nil {

return

}

nonce = txv1.Nonce

note = txv1.Note

gasLimit = txv1.GasLimit

var msg Message

msg, err = messageV1(chainID, txv1, height, chainVersion)

if err != nil {

return

}

messages = append(messages, msg)

} else if splitTx[1] == "v2" {

var txv2 types.Transaction

var pubKey crypto.PubKeyEd25519

txv2, pubKey, err = tx2.TxParse(txStr)

if err != nil {

txv2, pubKey, err = tx3.TxParse(txStr)

if err != nil {

return

}

}

fromAddr = pubKey.Address(chainID)

nonce = txv2.Nonce

note = txv2.Note

gasLimit = uint64(txv2.GasLimit)

var msg Message

for i := 0; i < len(txv2.Messages); i++ {

msg, err = message(chainID, txv2.Messages[i], height, chainVersion)

if err != nil {

return

}

messages = append(messages, msg)

}

} else {

err = errors.New("unsupported tx=" + txStr)

return

}

return

}

// Balance balance information query

func Balance(address types.Address, name, password, tokenName string, all bool, chainID, keyStorePath string) (result \*[]BalanceItemResult, err error) {

defer FuncRecover(&err)

\_, keyStorePath, chainID = prepare("", keyStorePath, chainID)

// if account address is empty, then load account with name and password

if address == "" {

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

return nil, err

}

address = acct.Address(chainID)

}

if all == false {

items := make([]BalanceItemResult, 0)

item, err := balanceOfToken(address, tokenName, chainID)

if err != nil {

return nil, err

}

items = append(items, \*item)

result = &items

} else {

temp, err := allBalance(chainID, address)

if err != nil {

return nil, err

}

result = temp

}

return

}

func balanceOfToken(accAddress types.Address, tokenName, chainID string) (result \*BalanceItemResult, err error) {

addrS := nodeAddrSlice(chainID)

var value []byte

var tokenAddress types.Address

if tokenName == "" {

tokenResult := new(std.Token)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfGenesisToken(), tokenResult)

if err != nil {

return nil, errors.New("get genesis token error: " + err.Error())

}

tokenAddress = tokenResult.Address

tokenName = tokenResult.Name

} else {

if value, err = common.DoHttpQuery(addrS, std.KeyOfTokenWithName(tokenName)); err != nil {

return

}

if len(value) == 0 {

return nil, errors.New("invalid tokenName")

}

if err = json.Unmarshal(value, &tokenAddress); err != nil {

return

}

}

if value, err = common.DoHttpQuery(addrS, std.KeyOfAccountToken(accAddress, tokenAddress)); err != nil {

return

}

result = new(BalanceItemResult)

if len(value) == 0 {

result.Balance = "0"

} else {

var tokenBalance TokenBalance

if err = json.Unmarshal(value, &tokenBalance); err != nil {

return

}

result.Balance = tokenBalance.Balance.String()

if tokenBalance.BVMBalanceMod != nil {

result.BVMBalance = tokenBalance.BVMBalanceMod.String()

}

}

tokenInfo := new(std.Token)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfGenesisToken(), tokenInfo)

if err != nil {

return

}

result.TokenAddress = tokenAddress

result.TokenName = tokenInfo.Name

if tokenName != "" {

result.TokenName = tokenName

}

return

}

func allBalance(chainID string, address types.Address) (items \*[]BalanceItemResult, err error) {

addrS := nodeAddrSlice(chainID)

tokens := make([]string, 0)

if err = common.DoHttpQueryAndParse(addrS, std.KeyOfAccount(address), &tokens); err != nil {

return

}

balanceItems := make([]BalanceItemResult, 0)

for \_, token := range tokens {

splitToken := strings.Split(token, "/")

if splitToken[4] != "token" || len(splitToken) != 6 {

continue

}

tokenBalance := new(TokenBalance)

if err = common.DoHttpQueryAndParse(addrS, token, &tokenBalance); err != nil {

return

}

var name string

if name, err = tokenName(chainID, tokenBalance.Address); err != nil {

return

}

BalanceItemRes := BalanceItemResult{

TokenAddress: tokenBalance.Address,

TokenName: name,

Balance: tokenBalance.Balance.String()}

if tokenBalance.BVMBalanceMod != nil {

BalanceItemRes.BVMBalance = tokenBalance.BVMBalanceMod.String()

}

balanceItems = append(balanceItems, BalanceItemRes)

}

items = &balanceItems

return

}

// Nonce nonce information query

func Nonce(address types.Address, name, password, chainID, keyStorePath string) (result \*NonceResult, err error) {

defer FuncRecover(&err)

addrS := nodeAddrSlice(chainID)

\_, keyStorePath, chainID = prepare("", keyStorePath, chainID)

if address == "" {

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

return nil, err

}

address = acct.Address(chainID)

}

type account struct {

Nonce uint64 `json:"nonce"`

}

a := new(account)

value, err := common.DoHttpQuery(addrS, std.KeyOfAccountNonce(address))

if err != nil {

return

}

result = new(NonceResult)

if len(value) == 0 {

result.Nonce = 1

} else {

err = json.Unmarshal(value, a)

if err != nil {

return

}

result.Nonce = a.Nonce + 1

}

return

}

// All info of contract

func ContractInfo(chainID, orgID, contractName string) (contracts map[string]std.Contract, err error) {

defer FuncRecover(&err)

ContractList := new(std.ContractVersionList)

addrS := nodeAddrSlice(chainID)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfContractsWithName(orgID, contractName), &ContractList)

contracts = make(map[string]std.Contract)

for k, v := range ContractList.ContractAddrList {

contract := new(std.Contract)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfContract(v), &contract)

contracts[contractName+string(k)] = \*contract

}

return

}

// Query contract information with address

func ContractInfoWithAddr(chainID, contractAddr string) (contract \*std.Contract, err error) {

addrS := nodeAddrSlice(chainID)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfContract(contractAddr), &contract)

return

}

// Query all contract information

func AllContractInfo(chainID string) (contractAddrList []string, contractList []interface{}, err error) {

contractAddrList = make([]string, 0)

addrS := nodeAddrSlice(chainID)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfAllContracts(), &contractAddrList)

key := "["

for \_, v := range contractAddrList {

key = key + v + ","

}

key = key[:len(key)-1] + "]"

contract := new(std.Contract)

contractList, err = common.DoBulkHttpQueryAndParse(addrS, std.KeyOfContract(key), contract)

return

}

func TokenInfo(tokenName, chainId string) (err error) {

tokenAddr := ""

addrS := nodeAddrSlice(chainId)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfTokenWithName(tokenName), &tokenAddr)

if err != nil {

return err

}

token := new(std.Token)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfToken(tokenAddr), &token)

if err != nil {

return err

}

fmt.Println("OK")

jsIndent, \_ := json.MarshalIndent(&token, "", " ")

fmt.Printf("Response: %s\n", string(jsIndent))

return

}

// Query all token information

func AllTokenInfo(chainID string) (err error) {

tokenAddrList := make([]string, 0)

addrS := nodeAddrSlice(chainID)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfAllToken(), &tokenAddrList)

key := "["

for \_, v := range tokenAddrList {

key = key + v + ","

}

key = key[:len(key)-1] + "]"

token := std.Token{}

tokenList, err := common.DoBulkHttpQueryAndParse(addrS, std.KeyOfToken(key), &token)

fmt.Println("OK")

fmt.Println("Response: ")

for \_, token := range tokenList {

fmt.Printf(" token name: %s token symbol: %s\n token addr: %s\n\n", token.(std.Token).Name, token.(std.Token).Symbol, token.(std.Token).Address)

}

return

}

// Query organization information

func QueryOrgInfo(orgID, chainID string) (OrgInfo \*std.Organization, err error) {

addrS := nodeAddrSlice(chainID)

OrgInfo = new(std.Organization)

err = common.DoHttpQueryAndParse(addrS, std.GetOrganizaitionInfo(orgID), &OrgInfo)

return

}

// CommitTx commit transaction information

func CommitTx(chainID, tx string) (commit \*CommitTxResult, err error) {

defer FuncRecover(&err)

requireNotEmpty("tx", tx)

addrS := nodeAddrSlice(chainID)

var result \*core\_types.ResultTx

result, err = common.DoHttpCommitTxAndParseAsync(addrS, tx)

if err != nil {

return

}

commit = new(CommitTxResult)

if result.CheckResult.Code != types.CodeOK {

commit.Code = result.CheckResult.Code

commit.Log = result.CheckResult.Log

} else {

commit.Code = result.DeliverResult.Code

commit.Log = result.DeliverResult.Log

}

commit.Fee = result.DeliverResult.Fee

commit.TxHash = "0x" + result.Hash

commit.Height = result.Height

commit.Data = result.DeliverResult.Data

if result.CheckResult.Code == types.CodeBVMQueryOK {

commit.Data = result.CheckResult.Data

}

return

}

//Version version information for the current block

func Version() (result \*VersionResult, err error) {

defer FuncRecover(&err)

result = new(VersionResult)

var version []byte

version, err = ioutil.ReadFile("./version")

if err != nil {

err = nil

result.Version = "0.0.0.1"

return

}

result.Version = string(version)

result.Version = strings.Replace(result.Version, "\r\n", "", -1)

result.Version = strings.Replace(result.Version, "\n", "", -1)

return

}

func blockResults(chainID string, height int64) (blkResults \*core\_types.ResultBlockResults, err error) {

addrS := nodeAddrSlice(chainID)

blkResults = new(core\_types.ResultBlockResults)

params := map[string]interface{}{"height": height}

err = common.DoHttpRequestAndParseExBlock(addrS, "block\_results", params, blkResults)

if err != nil {

return

}

return

}

func message(chainID string, message types.Message, height int64, chainVersion \*int64) (msg Message, err error) {

methodID := fmt.Sprintf("%x", message.MethodID)

msg.SmcAddress = message.Contract

if msg.SmcName, msg.Method, err = contractNameAndMethod(message.Contract, chainID, methodID, height, chainVersion); err != nil {

return

}

if methodID == transferMethodIDV2 {

if len(message.Items) != 2 {

return msg, errors.New("items count error")

}

var to types.Address

if err = rlp.DecodeBytes(message.Items[0], &to); err != nil {

return

}

var value bn.Number

if err = rlp.DecodeBytes(message.Items[1], &value); err != nil {

return

}

msg.To = to

msg.Value = value.String()

}

return

}

func messageV1(chainID string, tx tx1.Transaction, height int64, chainVersion \*int64) (msg Message, err error) {

var methodInfo tx1.MethodInfo

if err = rlp.DecodeBytes(tx.Data, &methodInfo); err != nil {

return

}

methodID := fmt.Sprintf("%x", methodInfo.MethodID)

msg.SmcAddress = tx.To

if msg.SmcName, msg.Method, err = contractNameAndMethod(tx.To, chainID, methodID, height, chainVersion); err != nil {

return

}

if methodID == transferMethodIDV1 {

var itemsBytes = make([][]byte, 0)

if err = rlp.DecodeBytes(methodInfo.ParamData, &itemsBytes); err != nil {

return

}

msg.To = string(itemsBytes[0])

msg.Value = new(big.Int).SetBytes(itemsBytes[1][:]).String()

}

return

}

func contractNameAndMethod2(contractAddress keys.Address, methodID string) (contractName string, method string, err error) {

contract := new(types2.Contract)

common.RWLock.RLock()

v, ok := common.ContractMap[contractAddress]

common.RWLock.RUnlock()

if ok == true {

contract = v

} else {

param := map[string]interface{}{"path": std.KeyOfContract(contractAddress)}

result := new(types2.ResultABCIQuery)

if err = common.DoHttpRequestAndParseExBlock(common.GetBCCConfig().Urls[common.GetBCCConfig().DefaultChainID], "abci\_query", param, result); err != nil {

return

}

err = json.Unmarshal(result.Response.Value, contract)

if err != nil {

return

}

common.RWLock.Lock()

common.ContractMap[contractAddress] = contract

common.RWLock.Unlock()

}

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

if methodItem.MethodId == methodID {

method = methodItem.Prototype

break

}

}

return contract.Name, method, nil

}

func contractNameAndMethod(contractAddress types.Address, chainID, methodID string, height int64, chainVersion \*int64) (contractName string, method string, err error) {

addrS := nodeAddrSlice(chainID)

contract := new(std.Contract)

if err = common.DoHttpQueryAndParse(addrS, std.KeyOfContract(contractAddress), contract); err != nil {

return

}

if chainVersion != nil && contract.LoseHeight != 0 && contract.LoseHeight < height {

conVer := new(std.ContractVersionList)

if err = common.DoHttpQueryAndParse(addrS, std.KeyOfContractsWithName(contract.OrgID, contract.Name), conVer); err == nil {

for index, eh := range conVer.EffectHeights {

if eh <= height {

tmp := new(std.Contract)

if err = common.DoHttpQueryAndParse(addrS, std.KeyOfContract(conVer.ContractAddrList[index]), tmp); err == nil {

if tmp.LoseHeight == 0 || (tmp.LoseHeight != 0 && tmp.LoseHeight > height) {

contract = tmp

break

}

} else {

return

}

}

}

} else {

return

}

}

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

if methodItem.MethodID == methodID {

method = methodItem.ProtoType

break

}

}

return contract.Name, method, nil

}

func tokenName(chainID string, tokenAddress types.Address) (name string, err error) {

addrS := nodeAddrSlice(chainID)

token := new(std.Token)

if err = common.DoHttpQueryAndParse(addrS, std.KeyOfToken(tokenAddress), token); err != nil {

return

}

return token.Name, err

}

func contractOfName(chainID, orgID, contractName string) (contract \*std.Contract, err error) {

addrS := nodeAddrSlice(chainID)

key := std.KeyOfContractsWithName(orgID, contractName)

contractList := new(std.ContractVersionList)

err = common.DoHttpQueryAndParse(addrS, key, contractList)

if err != nil {

return nil, errors.New("Is orgName or contract's name right? error: " + err.Error())

}

var blkHeightResult \*BlockHeightResult

blkHeightResult, err = BlockHeight(chainID)

if err != nil {

return

}

effectiveAddr := func() types.Address {

heightLen := len(contractList.EffectHeights)

var effectIndex int

for index, item := range contractList.EffectHeights {

effectIndex = index

if item < blkHeightResult.LastBlock {

if index < heightLen-1 && contractList.EffectHeights[index+1] > blkHeightResult.LastBlock {

break

}

}

}

return contractList.ContractAddrList[effectIndex]

}

key = std.KeyOfContract(effectiveAddr())

contract = new(std.Contract)

err = common.DoHttpQueryAndParse(addrS, key, contract)

return

}

func tokenAddressFromName(chainID, tokenName string) (tokenAddr types.Address, err error) {

addrS := nodeAddrSlice(chainID)

key := std.KeyOfTokenWithName(tokenName)

err = common.DoHttpQueryAndParse(addrS, key, &tokenAddr)

if err != nil {

return

}

return

}

func QueryOfRpc(key, chainID string) (result \*core\_types.ResultABCIQuery, err error) {

addrS := nodeAddrSlice(chainID)

result = new(core\_types.ResultABCIQuery)

result, err = common.DoHttpQueryForRpc(addrS, key)

if err != nil {

return nil, errors.New("rpc query failed , error: " + err.Error())

}

return result, nil

}

// Query the contract information based on the parameters for Rpc

func ContractInfoForRPC(orgName, contractName, orgID, contractAddr, chainID string) (contract \*std.Contract, err error) {

if orgID != "" && contractName != "" && contractAddr == "" {

contractList, err := ContractInfo(chainID, orgID, contractName)

if err != nil {

return nil, err

}

for \_, v := range contractList {

// 校验其他输入参数

if orgName != "" {

OrgInfo, err := QueryOrgInfo(orgID, chainID)

if err != nil {

return nil, err

}

if orgName != OrgInfo.Name {

fmt.Println("Error: Input orgName is wrong.")

return nil, err

}

}

return &v, nil

}

} else if orgName != "" && contractName != "" && contractAddr == "" {

contract, err := QueryContractInfoForRpc(orgName, contractName, chainID)

if err != nil {

return nil, err

}

// 校验其他输入参数

if orgID != "" && orgID != contract.OrgID {

fmt.Println("Error: Input orgID is wrong.")

return nil, err

}

return contract, nil

} else if contractAddr != "" {

contract, err := ContractInfoWithAddr(chainID, contractAddr)

if err != nil {

return nil, err

}

// 校验其他输入参数

if orgName != "" && orgID != "" {

OrgInfo, err := QueryOrgInfo(orgID, chainID)

if err != nil {

return nil, err

}

if orgName != OrgInfo.Name {

fmt.Println("Error: Input orgName is wrong.")

return nil, nil

}

}

if orgID != "" && orgID != contract.OrgID {

fmt.Println("Error: orgID orgName is wrong.")

return nil, nil

}

return contract, nil

} else if orgName == "" && contractName == "" && orgID == "" && contractAddr == "" {

\_, contractList, err := AllContractInfo(chainID)

if err != nil {

fmt.Println("Query ContractInfo failed")

return nil, err

}

fmt.Println("OK")

fmt.Println("Response: ")

for \_, contract := range contractList {

fmt.Printf(" contract name: %s\n contract addr: %s\n\n", contract.(std.Contract).Name, contract.(std.Contract).Address)

}

} else {

fmt.Println("Insufficient input parameters")

return nil, nil

}

return

}

type BlockChainHelper struct {

smc sdk.ISmartContract

}

func QueryContractInfoForRpc(orgName, contractName, chainID string) (contract \*std.Contract, err error) {

bh := helper.BlockChainHelper{}

orgID := bh.CalcOrgID(orgName)

addrS := nodeAddrSlice(chainID)

List := new(std.ContractVersionList)

err = common.DoHttpQueryAndParse(addrS, std.KeyOfContractsWithName(orgID, contractName), &List)

for \_, v := range List.ContractAddrList {

contract, err = ContractInfoWithAddr(chainID, v)

if err != nil {

return nil, err

}

return contract, nil

}

return

}

func Create(keyStorePath, name, password string) error {

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if name == "" {

return errors.New("Need wallet name ")

}

acct, err := wal.NewAccount(keyStorePath, name, password)

if err != nil {

Error(fmt.Sprintf("New account \"%v\" failed, %v", name, err.Error()))

return err

}

PubK := acct.PubKey().(crypto.PubKeyEd25519)

fmt.Println("OK")

fmt.Println("PubKey: ", "0x"+hex.EncodeToString(PubK[:]))

return nil

}

func Export(keyStorePath, name, password string) error {

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if name == "" {

return errors.New("Need wallet name ")

}

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

Error(fmt.Sprintf("Load account \"%v\" failed, %v", name, err.Error()))

return err

}

PriK := acct.PrivateKey.(crypto.PrivKeyEd25519)

PubK := acct.PubKey().(crypto.PubKeyEd25519)

fmt.Println("OK")

fmt.Println("PrivateKey: ", "0x"+hex.EncodeToString(PriK[:len(PriK)-len(PubK)]))

fmt.Println("PubKey: ", "0x"+hex.EncodeToString(PubK[:]))

return nil

}

func Import(keyStorePath, name, password, privateKey string) error {

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if name == "" {

return errors.New("Need wallet name ")

}

if privateKey == "" {

return errors.New("Need private Key of account ")

}

if privateKey[:2] == "0x" {

privateKey = privateKey[2:]

}

newPrivateKey, err := hex.DecodeString(privateKey)

if err != nil {

Error(fmt.Sprintf("Private Key conversion \"%v\" failed, %v", name, err.Error()))

}

if len(newPrivateKey) != 64 && len(newPrivateKey) != 32 {

return errors.New(fmt.Sprintf("Private key \"%v\" length incorrect, %v", privateKey, err.Error()))

}

CompletePrivateKey := crypto.PrivKeyEd25519FromBytes(newPrivateKey[:32])

pub := CompletePrivateKey.PubKey().Bytes()

newPrivateKey2 := append(newPrivateKey[:32], pub[5:]...)

if len(newPrivateKey2) != 64 {

return errors.New(fmt.Sprintf("Private key \"%v\" length incorrect, %v", privateKey, err.Error()))

}

acct, err := wal.ImportAccount(keyStorePath, name, password, crypto.PrivKeyEd25519FromBytes(newPrivateKey2))

if err != nil {

Error(fmt.Sprintf("Import account \"%v\" failed, %v", name, err.Error()))

}

PubK := acct.PubKey().(crypto.PubKeyEd25519)

fmt.Println("OK")

fmt.Println("PubKey: ", "0x"+hex.EncodeToString(PubK[:]))

return nil

}

func SignFile(keyStorePath, name, password, file, mode string) error {

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if name == "" {

return errors.New("Need wallet name ")

}

if file == "" {

return errors.New("Need signed file ")

}

if mode == "" {

return errors.New("Need mode of file ")

}

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

Error(fmt.Sprintf("Load account \"%v\" failed, %v", name, err.Error()))

return err

}

SignatureFile := file + ".sig"

if mode == "b" {

acct.SignBinFile(file, SignatureFile)

} else if mode == "t" {

acct.SignTextFile(file, SignatureFile)

} else {

return errors.New(fmt.Sprintf("Invalid mode to \"%v\", %v", name, err.Error()))

}

fmt.Println("OK")

fmt.Println("SignatureFile: ", SignatureFile)

return err

}

func SignData(keyStorePath, name, password, data string, file string) error {

if keyStorePath == "" {

keyStorePath = ".keystore"

}

if name == "" {

return errors.New("Need wallet name ")

}

if data == "" {

return errors.New("Need signed data ")

}

if file == "" {

return errors.New("Need file to be output after signing ")

}

newData, err := hex.DecodeString(data)

if err != nil {

Error(fmt.Sprintf("Decode hex data failed, %v", err.Error()))

return err

}

acct, err := wal.LoadAccount(keyStorePath, name, password)

if err != nil {

Error(fmt.Sprintf("Load account \"%v\" in %v failed, %v", name, keyStorePath, err.Error()))

return err

}

err = acct.Sign2File(newData, file)

if err != nil {

Error(fmt.Sprintf("Sign to \"%v\" failed, %v", name, err.Error()))

return err

}

fmt.Println("OK")

fmt.Println("SignatureFile: ", file)

return nil

}

func Error(s string) {

fmt.Printf("ERROR! -- %v\n", s)

os.Exit(1)

}