const (

usage = "giXwallet\_rpc's url"

)

var RootCmd = &cobra.Command{

Use: "giXwallet",

Short: "gi exchange wallet console",

Long: "giXwallet client that it can perform the wallet operation, query chain information and so on.",

}

func Execute() error {

addFlags()

addCommands()

return RootCmd.Execute()

}

func addFlags() {

addWalletCreateFlag()

addWalletExportFlag()

addWalletImportFlag()

addWalletListFlag()

addTransferFlag()

addTransferOfflineFlag()

addBlockHeightFlag()

addBlockFlag()

addTransactionFlag()

addBalanceFlag()

addBalanceOfTokenFlag()

addAllBalanceFlag()

addNonceFlag()

addCommitTxFlag()

}

func addCommands() {

RootCmd.AddCommand(walletCreateCmd)

RootCmd.AddCommand(walletExportCmd)

RootCmd.AddCommand(walletImportCmd)

RootCmd.AddCommand(walletListCmd)

RootCmd.AddCommand(transferCmd)

RootCmd.AddCommand(transferOfflineCmd)

RootCmd.AddCommand(blockHeightCmd)

RootCmd.AddCommand(blockCmd)

RootCmd.AddCommand(transactionCmd)

RootCmd.AddCommand(balanceCmd)

RootCmd.AddCommand(balanceOfTokenCmd)

RootCmd.AddCommand(allBalanceCmd)

RootCmd.AddCommand(nonceCmd)

RootCmd.AddCommand(commitTxCmd)

}

var walletCreateCmd = &cobra.Command{

Use: "walletCreate",

Short: "Create wallet",

Long: "Create a new wallet",

Args: cobra.ExactArgs(0),

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

return client.WalletCreate(flagName, flagPassword, flagRpcUrl)

},

}

func addWalletCreateFlag() {

walletCreateCmd.PersistentFlags().StringVarP(&flagName, "name", "n", "", "wallet name")

walletCreateCmd.PersistentFlags().StringVarP(&flagPassword, "password", "p", "", "wallet password")

walletCreateCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var walletExportCmd = &cobra.Command{

Use: "walletExport",

Short: "Export wallet",

Long: "Export the private key and walletAddr of wallet",

Args: cobra.ExactArgs(0),

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

return client.WalletExport(flagName, flagPassword, flagAccessKey, flagRpcUrl, flagPlainText)

},

}

func addWalletExportFlag() {

walletExportCmd.PersistentFlags().StringVarP(&flagName, "name", "n", "", "wallet name")

walletExportCmd.PersistentFlags().StringVarP(&flagPassword, "password", "p", "", "wallet password")

walletExportCmd.PersistentFlags().StringVarP(&flagAccessKey, "accessKey", "a", "", "wallet accessKey")

walletExportCmd.PersistentFlags().StringVarP(&flagPlainText, "plainText", "t", "", "export plain text(default false)")

walletExportCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var walletImportCmd = &cobra.Command{

Use: "walletImport",

Short: "Import wallet",

Long: "Import the private key to a new wallet",

Args: cobra.ExactArgs(0),

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

return client.WalletImport(flagName, flagEncPrivateKey, flagPassword, flagAccessKey, flagRpcUrl, flagPlainText)

},

}

func addWalletImportFlag() {

walletImportCmd.PersistentFlags().StringVarP(&flagName, "name", "n", "", "wallet name")

walletImportCmd.PersistentFlags().StringVarP(&flagEncPrivateKey, "privateKey", "k", "", "wallet privateKey")

walletImportCmd.PersistentFlags().StringVarP(&flagPassword, "password", "p", "", "wallet password")

walletImportCmd.PersistentFlags().StringVarP(&flagAccessKey, "accessKey", "a", "", "wallet accessKey")

walletImportCmd.PersistentFlags().StringVarP(&flagPlainText, "plainText", "t", "", "import plain text(default false)")

walletImportCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var walletListCmd = &cobra.Command{

Use: "walletList",

Short: "Wallet list",

Long: "Query all wallet names and walletAddrs",

Args: cobra.ExactArgs(0),

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

return client.WalletList(flagPageNum, flagRpcUrl)

},

}

func addWalletListFlag() {

walletListCmd.PersistentFlags().Uint64VarP(&flagPageNum, "pageNum", "p", 1, "page index, default first page")

walletListCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var transferCmd = &cobra.Command{

Use: "transfer",

Short: "Transfer token",

Long: "Transfer token to someone with value",

Args: cobra.ExactArgs(0),

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

return client.Transfer(flagName, flagAccessKey, flagSmcAddress, flagGasLimit, flagNote, flagTo, flagValue, flagRpcUrl)

},

}

func addTransferFlag() {

transferCmd.PersistentFlags().StringVarP(&flagName, "name", "n", "", "wallet name")

transferCmd.PersistentFlags().StringVarP(&flagAccessKey, "accessKey", "a", "", "wallet accessKey")

transferCmd.PersistentFlags().StringVarP(&flagSmcAddress, "smcAddress", "s", "", "smart contract address")

transferCmd.PersistentFlags().StringVarP(&flagGasLimit, "gasLimit", "g", "5000", "gas limit ")

transferCmd.PersistentFlags().StringVarP(&flagNote, "note", "o", "", "note")

transferCmd.PersistentFlags().StringVarP(&flagTo, "to", "t", "", "to address")

transferCmd.PersistentFlags().StringVarP(&flagValue, "value", "v", "", "transfer value")

transferCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var transferOfflineCmd = &cobra.Command{

Use: "transferOffline",

Short: "Offline transaction",

Long: "Offline pack and sign transfer transaction",

Args: cobra.ExactArgs(0),

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

return client.TransferOffline(flagName, flagAccessKey, flagSmcAddress, flagGasLimit, flagNote, flagTo, flagValue, flagNonce, flagRpcUrl)

},

}

func addTransferOfflineFlag() {

transferOfflineCmd.PersistentFlags().StringVarP(&flagName, "name", "n", "", "wallet name")

transferOfflineCmd.PersistentFlags().StringVarP(&flagAccessKey, "accessKey", "a", "", "wallet accessKey")

transferOfflineCmd.PersistentFlags().StringVarP(&flagSmcAddress, "smcAddress", "s", "", "smart contract address")

transferOfflineCmd.PersistentFlags().StringVarP(&flagGasLimit, "gasLimit", "g", "5000", "gas limit ")

transferOfflineCmd.PersistentFlags().StringVarP(&flagNonce, "nonce", "c", "", "nonce")

transferOfflineCmd.PersistentFlags().StringVarP(&flagNote, "note", "o", "", "note")

transferOfflineCmd.PersistentFlags().StringVarP(&flagTo, "to", "t", "", "to address")

transferOfflineCmd.PersistentFlags().StringVarP(&flagValue, "value", "v", "", "transfer value")

transferOfflineCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var blockHeightCmd = &cobra.Command{

Use: "blockHeight",

Short: "Get current block height",

Long: "Get BlockChain current block height",

Args: cobra.ExactArgs(0),

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

return client.BlockHeight(flagRpcUrl)

},

}

func addBlockHeightFlag() {

blockHeightCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var blockCmd = &cobra.Command{

Use: "block",

Short: "Get block information",

Long: "Get block information with height, must great than zero",

Args: cobra.ExactArgs(0),

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

return client.Block(flagHeight, flagRpcUrl)

},

}

func addBlockFlag() {

blockCmd.PersistentFlags().Int64VarP(&flagHeight, "height", "t", 0, "block height")

blockCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var transactionCmd = &cobra.Command{

Use: "transaction",

Short: "Get transaction information",

Long: "Get transaction information with txHash and cannot be empty",

Args: cobra.ExactArgs(0),

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

return client.Transaction(flagTxHash, flagRpcUrl)

},

}

func addTransactionFlag() {

transactionCmd.PersistentFlags().StringVarP(&flagTxHash, "txHash", "t", "", "transaction's hash")

transactionCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var balanceCmd = &cobra.Command{

Use: "balance",

Short: "Get balance information",

Long: "Get balance of GI token for specific address",

Args: cobra.ExactArgs(0),

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

return client.Balance(flagAddress, flagRpcUrl)

},

}

func addBalanceFlag() {

balanceCmd.PersistentFlags().StringVarP(&flagAddress, "address", "a", "", "account's address")

balanceCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var balanceOfTokenCmd = &cobra.Command{

Use: "balanceOfToken",

Short: "Get balance information of address",

Long: "Get balance of specific token for specific address",

Args: cobra.ExactArgs(0),

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

return client.BalanceOfToken(flagAddress, flagTokenAddress, flagTokenName, flagRpcUrl)

},

}

func addBalanceOfTokenFlag() {

balanceOfTokenCmd.PersistentFlags().StringVarP(&flagAddress, "address", "a", "", "account's address")

balanceOfTokenCmd.PersistentFlags().StringVarP(&flagTokenAddress, "tokenAddress", "t", "", "token's address")

balanceOfTokenCmd.PersistentFlags().StringVarP(&flagTokenName, "tokenName", "n", "", "token's address")

balanceOfTokenCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var allBalanceCmd = &cobra.Command{

Use: "allBalance",

Short: "Get all balance information",

Long: "Get balance of all tokens for specific address",

Args: cobra.ExactArgs(0),

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

return client.AllBalance(flagAddress, flagRpcUrl)

},

}

func addAllBalanceFlag() {

allBalanceCmd.PersistentFlags().StringVarP(&flagAddress, "address", "a", "", "account's address")

allBalanceCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var nonceCmd = &cobra.Command{

Use: "nonce",

Short: "Get account nonce",

Long: "Get the next usable nonce for specific address",

Args: cobra.ExactArgs(0),

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

return client.Nonce(flagAddress, flagRpcUrl)

},

}

func addNonceFlag() {

nonceCmd.PersistentFlags().StringVarP(&flagAddress, "address", "a", "", "account's address")

nonceCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

var commitTxCmd = &cobra.Command{

Use: "commitTx",

Short: "Commit transaction",

Long: "Commit transaction with tx's data",

Args: cobra.ExactArgs(0),

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

return client.CommitTx(flagTx, flagRpcUrl)

},

}

func addCommitTxFlag() {

commitTxCmd.PersistentFlags().StringVarP(&flagTx, "tx", "t", "", "packed and signed transaction's data")

commitTxCmd.PersistentFlags().StringVarP(&flagRpcUrl, "url", "u", serverAddr(common.GetConfig().ServerAddr, true), usage)

}

func BlockHeight(url string) (err error) {

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

result := new(rpc3.BlockHeightResult)

\_, err = rpc.Call("gi\_blockHeight", map[string]interface{}{}, result)

if err != nil {

fmt.Printf("Cannot get block height, error=%s \n", err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func Block(height int64, url string) (err error) {

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

result := new(rpc3.BlockResult)

\_, err = rpc.Call("gi\_block", map[string]interface{}{"height": height}, result)

if err != nil {

fmt.Printf("Cannot get block data, height=%d, error=%s \n", height, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func Transaction(txHash, url string) (err error) {

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

txHash = txHash[2:]

}

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

result := new(rpc3.TxResult)

\_, err = rpc.Call("gi\_transaction", map[string]interface{}{"txHash": txHash}, result)

if err != nil {

fmt.Printf("Cannot get transaction, txHash=%s, error=%s \n", txHash, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func Balance(address keys.Address, url string) (err error) {

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

result := new(rpc3.BalanceResult)

\_, err = rpc.Call("gi\_balance", map[string]interface{}{"address": address}, result)

if err != nil {

fmt.Printf("Cannot get balance, address=%s, error=%s \n", address, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func BalanceOfToken(address, tokenAddress keys.Address, tokenName string, url string) (err error) {

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

result := new(rpc3.BalanceResult)

\_, err = rpc.Call("gi\_balanceOfToken", map[string]interface{}{"address": address, "tokenAddress": tokenAddress, "tokenName": tokenName}, result)

if err != nil {

fmt.Printf("Cannot get balance of token, address=%s, tokenAddress=%s, error=%s \n", address, tokenAddress, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func AllBalance(address keys.Address, url string) (err error) {

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

result := new([]rpc3.AllBalanceItemResult)

\_, err = rpc.Call("gi\_allBalance", map[string]interface{}{"address": address}, result)

if err != nil {

fmt.Printf("Cannot all balance, address=%s, error=%s \n", address, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func Nonce(address keys.Address, url string) (err error) {

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

result := new(rpc3.NonceResult)

\_, err = rpc.Call("gi\_nonce", map[string]interface{}{"address": address}, result)

if err != nil {

fmt.Printf("Cannot get nonce, address=%s, error=%s \n", address, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func CommitTx(tx, url string) (err error) {

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

result := new(rpc3.CommitTxResult)

\_, err = rpc.Call("gi\_commitTx", map[string]interface{}{"tx": tx}, result)

if err != nil {

fmt.Printf("Cannot commit transation, tx=%s, error=%s \n", tx, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func WalletCreate(name, password, url string) (err error) {

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

result := new(rpc3.WalletCreateResult)

\_, err = rpc.Call("gi\_walletCreate", map[string]interface{}{"name": name, "password": password}, result)

if err != nil {

fmt.Printf("Cannot create wallet, name=%s, password=%s,\n error=%s \n", name, password, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func WalletExport(name, password, accessKey, url, plainText string) (err error) {

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

bPlainText, err := strconv.ParseBool(plainText)

if err != nil {

return

}

result := new(rpc3.WalletExportResult)

\_, err = rpc.Call("gi\_walletExport", map[string]interface{}{"name": name, "password": password, "accessKey": accessKey, "plainText": bPlainText}, result)

if err != nil {

fmt.Printf("Cannot export wallet, name=%s, password=%s, accessKey=%s, plainText=%v,\n error=%s \n", name, password, accessKey, plainText, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func WalletImport(name, privateKey, password, accessKey, url, plainText string) (err error) {

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

bPlainText, err := strconv.ParseBool(plainText)

if err != nil {

return

}

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

privateKey = privateKey[2:]

}

result := new(rpc3.WalletImportResult)

\_, err = rpc.Call("gi\_walletImport", map[string]interface{}{"name": name, "privateKey": privateKey, "password": password, "accessKey": accessKey, "plainText": bPlainText}, result)

if err != nil {

fmt.Printf("Cannot import wallet, name=%s, privateKey=%s, password=%s, accessKey=%s, plainText=%v,\n error=%s \n", name, privateKey, password, accessKey, plainText, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func WalletList(pageNum uint64, url string) (err error) {

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

result := new(rpc3.WalletListResult)

\_, err = rpc.Call("gi\_walletList", map[string]interface{}{"pageNum": pageNum}, result)

if err != nil {

fmt.Printf("Cannot list wallet, error=%s \n", err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func Transfer(name, accessKey, smcAddress, gasLimit, note, to, value, url string) (err error) {

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

transferParam := rpc3.TransferParam{SmcAddress: smcAddress, GasLimit: gasLimit, Note: note, To: to, Value: value}

result := new(rpc3.TransferResult)

\_, err = rpc.Call("gi\_transfer", map[string]interface{}{"name": name, "accessKey": accessKey, "walletParams": transferParam}, result)

if err != nil {

fmt.Printf("Cannot transfer, name=%s, accessKey=%s, walletParam=%v,\n error=%s \n", name, accessKey, transferParam, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

return

}

func TransferOffline(name, accessKey, smcAddress, gasLimit, note, to, value, nonce, url string) (err error) {

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

uNonce, err := strconv.ParseUint(nonce, 10, 64)

if err != nil {

return

}

transferParam := rpc3.TransferOfflineParam{SmcAddress: smcAddress, GasLimit: gasLimit, Note: note, Nonce: uNonce, To: to, Value: value}

result := new(rpc3.TransferOfflineResult)

\_, err = rpc.Call("gi\_transferOffline", map[string]interface{}{"name": name, "accessKey": accessKey, "walletParams": transferParam}, result)

if err != nil {

fmt.Printf("Cannot transferOffline, name=%s, accessKey=%s, walletParam=%v,\n error=%s \n", name, accessKey, transferParam, err.Error())

return nil

}

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

fmt.Println(string(jsIndent))

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(CorrectUrls) == 0 {

RWLock.Unlock()

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

}

length := len(CorrectUrls)

var rnd int

if length > 1 {

rnd = rand.Intn(length - 1)

} else {

rnd = 0

}

url := CorrectUrls[rnd]

RWLock.Unlock()

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

if err == nil {

break

} else {

RWLock.Lock()

FaultCounterMap[url] += 1

if FaultCounterMap[url] > 10 {

if rnd == length-1 {

CorrectUrls = append(CorrectUrls[:rnd])

} else {

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

}

length -= 1

}

RWLock.Unlock()

if length <= len(nodeAddrSlice)/3 {

go DealFaultUrls(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 DealFaultUrls(methodName string, params map[string]interface{}, result interface{}) {

RWLock.Lock()

FaultUrls2 := FaultCounterMap

RWLock.Unlock()

for k, \_ := range FaultUrls2 {

err := CallChainApi(k, methodName, params, result)

if err == nil {

RWLock.Lock()

CorrectUrls = append(CorrectUrls, k)

FaultCounterMap[k] = 0

RWLock.Unlock()

}

}

}

type JSONRPCClient struct {

address string

client \*http.Client

cdc \*amino.Codec

}

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

}

func InitAll() error {

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

moduleName := "giXwallet"

err := giXWalletConfig.InitConfig(configFile)

if err != nil {

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

}

initLog(moduleName)

if giXWalletConfig.ChainID == "" {

return errors.New(" chainId cannot be empty")

}

crypto.SetChainId(giXWalletConfig.ChainID)

tx2.Init(giXWalletConfig.ChainID)

tx3.Init(giXWalletConfig.ChainID)

CheckChainVersion()

for \_, v := range GetConfig().NodeAddrSlice {

FaultCounterMap[v] = 0

CorrectUrls = append(CorrectUrls, v)

}

rand.Seed(time.Now().Unix())

return nil

}

func initLog(moduleName string) {

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

l.SetOutputToFile(giXWalletConfig.LoggerFile)

l.SetOutputToScreen(giXWalletConfig.LoggerScreen)

l.AllowLevel(giXWalletConfig.LoggerLevel)

logger = l

}

func GetConfig() config.Config {

return giXWalletConfig

}

func GetLogger() log.Logger {

return logger

}

func FuncRecover(l log.Logger, 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

}

l.Error("FuncRecover", "error", msg)

\*errPtr = errors.New(msg)

}

}

func OutCertFileIsExist() (string, string) {

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

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

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

if err != nil {

return crtPath, keyPath

}

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

if err != nil {

return crtPath, keyPath

}

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

}

func CheckChainVersion() {

cfg := giXWalletConfig

if cfg.ChainVersion != "1" && cfg.ChainVersion != "2" && cfg.ChainVersion != "" {

fmt.Println("Config file error, please check chainVersion!")

return

}

if cfg.ChainVersion == "2" {

return

}

ChainVersion, err := queryChainVersion()

if err != nil {

fmt.Println("Query ChainVersion failed, please check!")

return

}

if ChainVersion == "0" {

ChainVersion = "1"

}

if cfg.ChainVersion != ChainVersion {

changeChainVersion(ChainVersion)

giXWalletConfig.ChainVersion = ChainVersion

}

}

func queryChainVersion() (chainVersion string, err error) {

result := new(core\_types.ResultHealth)

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

err = DoHttpRequestAndParseExBlock(GetConfig().NodeAddrSlice, "health", params, result)

if err != nil {

return "", err

}

chainVersion = strconv.FormatInt(result.ChainVersion, 10)

return

}

func changeChainVersion(chainversion string) {

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

f, err := os.Open(configFile)

if err != nil {

fmt.Println("OpenFile failed, please check!")

return

}

defer f.Close()

buf := bufio.NewReader(f)

var Str string

for {

line, err := buf.ReadString('\n')

if strings.HasPrefix(line, "chainVersion:") {

line = "chainVersion: " + chainversion + "\n"

}

Str = Str + line

if err != nil {

if err == io.EOF {

break

}

return

}

}

file2, err := os.Create(configFile)

if err != nil {

fmt.Println("CreateFile failed, please check!")

return

}

defer file2.Close()

\_, err = file2.WriteString(Str)

if err != nil {

fmt.Println("WriteFile failed, please check!")

return

}

}

func ToHex(val uint64) string {

valBytes := make([]byte, 8)

binary.BigEndian.PutUint64(valBytes, val)

return string("0x") + hex.EncodeToString(valBytes)

}

func BytesToHex(valBytes []byte) string {

return string("0x") + hex.EncodeToString(valBytes)

}

func FloatToHex(val string) string {

resultInt := FloatStrToBigInt(val)

return BytesToHex(resultInt.Bytes())

}

func ParseHexString(hexStr string, fieldName string, lenConstraint int) error {

if len(hexStr)%2 != 0 {

return errors.New(fieldName + " must be hex string with even length")

}

hexBytes, \_ := hex.DecodeString(hexStr)

if lenConstraint > 0 && len(hexBytes) != lenConstraint {

return errors.New(fieldName + " must be " + strconv.Itoa(lenConstraint\*2) + " hex-chars")

}

return nil

}

func UintToHex(val uint64) string {

var buf = make([]byte, 8)

binary.BigEndian.PutUint64(buf, val)

return string("0x") + hex.EncodeToString(buf)

}

func UintToBigInt(val uint64) big.Int {

var buf = make([]byte, 8)

binary.BigEndian.PutUint64(buf, val)

return \*new(big.Int).SetBytes(buf)

}

func IntToHex(val int64) string {

var buf = make([]byte, 8)

binary.BigEndian.PutUint64(buf, uint64(val))

return string("0x") + hex.EncodeToString(buf)

}

func IntToByte(val int64) []byte {

var buf = make([]byte, 8)

binary.BigEndian.PutUint64(buf, uint64(val))

return buf

}

func JudgeFloatStr(val string) {

pattern := `^\d+(\.\d{0,9})?$`

valid, err := regexp.Match(pattern, []byte(val))

if err != nil {

logger.Info("Regular expression error")

panic(" Regular expression error")

}

if !valid {

logger.Info(`The money is illegal. It can only be float and >= 0.000000001`)

panic(`The money is illegal. It can only be float and >= 0.000000001`)

}

//return true

}

func FloatStrToBigInt(val string) big.Int {

JudgeFloatStr(val)

var valStr []string

valInt := big.NewInt(0)

valFloat := big.NewInt(0)

if strings.Contains(val, ".") {

valStr = strings.Split(val, ".")

var valFloatStr string

length := len(valStr[1])

if length > 9 {

valFloatStr = (valStr[1])[0:9]

length = 9

} else {

valFloatStr = valStr[1]

}

intNum, \_ := strconv.ParseInt(valFloatStr, 10, 64)

valFloat = big.NewInt(intNum)

var mulTemp int64 = 1

for i := 0; i < (9 - length); i++ {

mulTemp \*= 10

}

valFloat.Mul(valFloat, big.NewInt(mulTemp))

} else {

valStr = append(valStr, val)

}

intStr := valStr[0]

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

intNum, \_ := strconv.ParseInt(string(intStr[i]), 10, 64)

valInt.Mul(valInt, big.NewInt(10))

valInt.Add(valInt, big.NewInt(intNum))

}

valInt.Mul(valInt, big.NewInt(1E9))

return \*valInt.Add(valInt, valFloat)

}

func CurrentDirectory() (string, error) {

file, err := exec.LookPath(os.Args[0])

if err != nil {

return "", err

}

path, err := filepath.Abs(file)

if err != nil {

return "", err

}

i := strings.LastIndex(path, "/")

if i < 0 {

i = strings.LastIndex(path, "\\")

}

if i < 0 {

return "", errors.New(`error: Can't find "/" or "\".`)

}

return string(path[0 : i+1]), nil

}

func newAccount(name, password string) (\*Account, []byte, error) {

cfg := common.GetConfig()

isExist, \_ := db.IsExist(name)

if isExist {

return nil, nil, errors.New("The account of " + name + " is already exist!")

}

priKey := crypto.GenPrivKeyEd25519()

priKeyByte := priKey[:]

accessKey := crypto.CRandBytes(32)

priKeyWithPWBytes := algorithm.EncryptWithPassword(priKeyByte, []byte(password), accessKey)

acct := Account{

Name: name,

Address: priKey.PubKey().Address(cfg.ChainID),

EncPrivateKey: priKeyWithPWBytes,

PrivateKey: priKeyByte,

}

return &acct, accessKey, nil

}

func (acct \*Account) Save(accessKey []byte) error {

return db.SetAccount(acct, accessKey)

}

// GetPassword will prompt for a password one-time (to sign a tx)

// It enforces the password length

func getPassword(prompt string, buf \*bufio.Reader) (pass string, err error) {

pass, err = speakeasy.Ask(prompt)

if err != nil {

return "", err

}

if len(pass) < MinPassLength {

return "", errors.Errorf("Password must be at least %d characters", MinPassLength)

}

if len(pass) > MaxPassLength {

return "", errors.Errorf("Password must be at most %d characters", MaxPassLength)

}

return pass, nil

}

func keyOfAccountNumber() []byte {

return []byte("/giXWallet/accountNumber")

}

func keyOfWalletList(pageNumber uint64) []byte {

return []byte(fmt.Sprintf("/giXWallet/walletList/page%d", pageNumber))

}

// Init DB

func InitDB() error {

var err error

dbPath := absolutePath(common.GetConfig().KeyStorePath)

db.GILevelDB, err = gidb.OpenDB(dbPath, "", "")

return err

}

func absolutePath(path string) string {

if filepath.IsAbs(path) {

path = filepath.Join(path, dbName)

} else {

dir, err := common.CurrentDirectory()

if err != nil {

panic(err)

}

path = filepath.Join(dir, path, dbName)

}

return path

}

// IsExist - get true of account is exist or not

func (db \*DB) IsExist(name string) (bool, error) {

//获取账户信息

acctBytes, err := db.Get([]byte(name))

if err != nil {

return false, err

}

if len(acctBytes) == 0 {

return false, errors.New("account does not exist")

}

return true, nil

}

func (db \*DB) SetAccount(acct \*Account, accessKey []byte) error {

jsonBytes, err := cdc.MarshalJSON(acct)

if err != nil {

return err

}

walBytes := algorithm.EncryptWithPassword(jsonBytes, nil, accessKey)

acctNumber, err := db.AccountNumber()

if err != nil {

panic(err)

}

pageNumber := acctNumber/countOfOnePage + 1

walletList, err := db.WalletList(pageNumber)

if err != nil {

return err

}

walletList = append(walletList, acct.Name+"#"+acct.Address)

jsonList, err := cdc.MarshalJSON(&walletList)

if err != nil {

return err

}

//存储总的钱包数

acctNumber++

jsonCount, err := cdc.MarshalJSON(&acctNumber)

if err != nil {

return err

}

// batch set

dbBatch := db.NewBatch()

dbBatch.Set([]byte(acct.Name), walBytes)

dbBatch.Set([]byte(keyOfWalletList(pageNumber)), jsonList)

dbBatch.Set([]byte(keyOfAccountNumber()), jsonCount)

return dbBatch.Commit()

}

func (db \*DB) WalletList(pageNumber uint64) ([]string, error) {

bytes, err := db.Get(keyOfWalletList(pageNumber))

if err != nil {

return nil, err

}

if len(bytes) == 0 {

return nil, nil

}

list := make([]string, 0)

err = cdc.UnmarshalJSON(bytes, &list)

return list, err

}

func (db \*DB) AccountNumber() (uint64, error) {

bytes, err := db.Get(keyOfAccountNumber())

if err != nil {

return 0, err

}

if len(bytes) == 0 {

return 0, nil

}

number := uint64(0)

err = cdc.UnmarshalJSON(bytes, &number)

return number, err

}

func blockHeight() (blkHeight \*BlockHeightResult, err error) {

result := new(core\_types.ResultABCIInfo)

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

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

if err != nil {

return

}

blkHeight = new(BlockHeightResult)

blkHeight.LastBlock = result.Response.LastBlockHeight

return

}

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

result := new(core\_types.ResultBlock)

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

err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "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

var blkResults \*core\_types.ResultBlockResults

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

return

}

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

}

func transactionBlock(k int, ResDeliver \*types2.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 types3.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 = make([]Message, 0)

tx.Messages = messages

tx.TransferReceipts, err = transferReceipts(\*ResDeliver)

return

}

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

result := new(core\_types.ResultTx)

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

err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "tx", params, result)

if err != nil {

return

}

if resultBlock == nil {

resultBlock = new(core\_types.ResultBlock)

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

err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "block", params, resultBlock)

if err != nil {

return

}

}

//ParseTX

var (

transaction tx1.Transaction

fromAddr string

msg Message

GasLimit uint64

Nonce uint64

Note string

txStr string

)

var blkResults \*core\_types.ResultBlockResults

if blkResults, err = blockResults(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])

}

}

messages := make([]Message, 0)

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 types3.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(crypto.GetChainId())

var msg Message

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

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

if err != nil {

return

}

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" + 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 = make([]Message, 0)

tx.Messages = messages

tx.TransferReceipts, err = transferReceipts(result.DeliverResult)

return

}

func messageV2Parse(message types3.Message) (msg Message, err error) {

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

msg.SmcAddress = message.Contract

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

return

}

if len(message.Items) != 2 {

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

}

if methodID == transferMethodIDV2 {

var to types3.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 balanceOfToken(address, tokenAddress keys.Address, tokenName string) (result \*BalanceResult, err error) {

var value []byte

if tokenName != "" {

var tmpAddress keys.Address

param := map[string]interface{}{"path": keyOfTokenName(tokenName)}

result := new(types.ResultABCIQuery)

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, result); err != nil {

return nil, err

}

value = result.Response.Value

if len(value) == 0 {

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

}

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

return nil, err

}

if tokenAddress != "" && tokenAddress != tmpAddress {

return nil, errors.New("tokenAddress and tokenName not be same token")

}

tokenAddress = tmpAddress

} else if tokenAddress == "" {

return nil, errors.New("tokenAddress and tokenName cannot be empty with both")

}

NewResult := new(types.ResultABCIQuery)

param := map[string]interface{}{"path": keyOfAccountToken(address, tokenAddress)}

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, &NewResult); err != nil {

return

}

value = NewResult.Response.Value

result = new(BalanceResult)

if len(value) == 0 {

result.Balance = "0"

} else {

var tokenBalance types.TokenBalance

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

return

}

result.Balance = tokenBalance.Balance.String()

}

return

}

func allBalance(address keys.Address) (items \*[]AllBalanceItemResult, err error) {

tokens := make([]string, 0)

result := new(types.ResultABCIQuery)

param := map[string]interface{}{"path": keyOfAccount(address)}

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, &result); err != nil {

return

}

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

if err != nil {

return

}

balanceItems := make([]AllBalanceItemResult, 0)

for \_, token := range tokens {

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

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

continue

}

tokenBalance := new(types.TokenBalance)

result := new(types.ResultABCIQuery)

param := map[string]interface{}{"path": token}

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, &result); err != nil {

return

}

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

if err != nil {

return

}

var name string

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

return

}

balanceItems = append(balanceItems,

AllBalanceItemResult{

TokenAddress: tokenBalance.Address,

TokenName: name,

Balance: tokenBalance.Balance.String()})

}

return &balanceItems, err

}

func nonce(acctAddress keys.Address) (result \*NonceResult, err error) {

type account struct {

Nonce uint64 `json:"nonce"`

}

a := new(account)

NewResult := new(types.ResultABCIQuery)

param := map[string]interface{}{"path": keyOfAccountNonce(acctAddress)}

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, &NewResult); err != nil {

return

}

value := NewResult.Response.Value

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

}

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

result := new(types.ResultBroadcastTxCommit)

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

err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "broadcast\_tx\_commit", param, result)

if err != nil {

return

}

commit = new(CommitTxResult)

if result.CheckTx.Code != types2.CodeTypeOK {

commit.Code = result.CheckTx.Code

commit.Log = result.CheckTx.Log

} else {

commit.Code = result.DeliverTx.Code

commit.Log = result.DeliverTx.Log

}

commit.Fee = result.DeliverTx.Fee

commit.TxHash = "0x" + hex.EncodeToString(result.Hash)

commit.Height = result.Height

return

}

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

blkResults = new(core\_types.ResultBlockResults)

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

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

if err != nil {

return

}

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 = contractNameAndMethod(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 contractNameAndMethod(contractAddress keys.Address, methodID string) (contractName string, method string, err error) {

contract := new(types.Contract)

common.RWLock.RLock()

v, ok := common.ContractMap[contractAddress]

common.RWLock.RUnlock()

if ok == true {

contract = v

} else {

param := map[string]interface{}{"path": keyOfContract(contractAddress)}

result := new(types.ResultABCIQuery)

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "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 tokenName(tokenAddress keys.Address) (name string, err error) {

token := new(types.IssueToken)

param := map[string]interface{}{"path": keyOfToken(tokenAddress)}

result := new(types.ResultABCIQuery)

if err = common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, result); err != nil {

return

}

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

if err != nil {

return

}

return token.Name, err

}

func genesisToken() string {

if genesisTokenAddr == "" {

token := new(types.IssueToken)

param := map[string]interface{}{"path": keyOfGenesisToken()}

result := new(types.ResultABCIQuery)

if err := common.DoHttpRequestAndParseExBlock(common.GetConfig().NodeAddrSlice, "abci\_query", param, result); err != nil {

return ""

} else {

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

genesisTokenAddr = token.Address

}

}

return genesisTokenAddr

}

func checkName(name string) error {

valid, err := regexp.Match(pattern, []byte(name))

if err != nil {

return errors.New("Regular expression error=" + err.Error())

}

if !valid {

return errors.New(`Name contains by [letters, numbers, "\_", "@", "." and "-"] and length must be [1-40] `)

}

return nil

}

func checkPrivateKey(privateKey string, plainText bool) error {

switch plainText {

case true:

if len(privateKey) != 128 && len(privateKey) != 64 {

return errors.New("The length of privateKey is wrong ")

}

case false:

if len(privateKey) != 160 {

return errors.New("The length of privateKey is wrong ")

}

}

return nil

}

func checkPassword(s string) (flag bool) {

ascOther := ` !"#$%&'()\*+,-/:;<=>?[]\^{|}~@\_.` + "`"

count := 0

number := false

upper := false

lower := false

special := false

other := true

for \_, c := range s {

switch {

case unicode.IsNumber(c):

number = true

count++

case unicode.IsUpper(c):

upper = true

count++

case unicode.IsLower(c):

lower = true

count++

case strings.Contains(ascOther, string(c)):

special = true

count++

default:

other = false

}

}

flag = number && upper && lower && special && other && 8 <= count && count <= 20

return

}

// nolint

func checkAddress(chainID string, addr smc.Address) error {

if !strings.HasPrefix(addr, chainID) {

return errors.New("Address chainID is error! ")

}

base58Addr := strings.Replace(addr, chainID, "", 1)

addrData := base58.Decode(base58Addr)

dataLen := len(addrData)

if dataLen < 4 {

return errors.New("Base58Addr parse error! ")

}

hasher := ripemd160.New()

hasher.Write(addrData[:dataLen-4])

md := hasher.Sum(nil)

if bytes.Compare(md[:4], addrData[dataLen-4:]) != 0 {

return errors.New("Address checksum is error! ")

}

return nil

}

func requireUint64(valueStr string) (uint64, error) {

value, err := strconv.ParseUint(valueStr, 10, 64)

if err != nil {

return 0, err

}

return value, nil

}

// WalletCreate - create wallet

func WalletCreate(name, password string) (result \*WalletCreateResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_walletCreate", "name", name)

if err = checkName(name); err != nil {

return

}

if password != "" && !checkPassword(password) {

return nil, pwErr

}

if len(password) == 0 {

buf := bufio.NewReader(os.Stdin)

password, err = getPassword("Enter Password("+name+"):", buf)

if err != nil {

return

}

}

result = new(WalletCreateResult)

result, err = walletCreate(name, password)

if err != nil {

logger.Error("Cannot create wallet", "error", err)

}

return

}

// WalletExport - export wallet

func WalletExport(name, password, accessKey string, plainText bool) (result \*WalletExportResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_walletExport", "name", name, "plainText", plainText)

if err = checkName(name); err != nil {

return

}

if password != "" && !checkPassword(password) {

return nil, pwErr

}

if len(password) == 0 {

buf := bufio.NewReader(os.Stdin)

password, err = getPassword("Enter Password("+name+"):", buf)

if err != nil {

return

}

}

if accessKey == "" {

return nil, errors.New("The accessKey can not be empty ")

}

result, err = walletExport(name, password, accessKey, plainText)

if err != nil {

logger.Error("Cannot export wallet", "error", err)

}

if plainText && len(result.PrivateKey) > 66 {

result.PrivateKey = result.PrivateKey[:len(result.PrivateKey)/2]

}

result.PrivateKey = "0x" + result.PrivateKey

return

}

// WalletImport - import wallet

func WalletImport(name, privateKey, password, accessKey string, plainText bool) (result \*WalletImportResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_walletImport", "name", name)

if err = checkName(name); err != nil {

return

}

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

privateKey = privateKey[2:]

}

// check private length

if err = checkPrivateKey(privateKey, plainText); err != nil {

return

}

if plainText == true {

newPrivateKey, err := hex.DecodeString(privateKey)

if err != nil {

fmt.Println("Private Key conversion failed")

return nil, err

}

CompletePrivateKey := crypto.PrivKeyEd25519FromBytes(newPrivateKey[:32])

pub := CompletePrivateKey.PubKey().Bytes()

newPrivateKey2 := append(newPrivateKey[:32], pub[5:]...)

if len(newPrivateKey2) != 64 {

fmt.Println("Private key length incorrect")

return nil, err

}

if password != "" && !checkPassword(password) {

return nil, pwErr

}

if len(password) == 0 {

buf := bufio.NewReader(os.Stdin)

password, err = getPassword("Enter Password("+name+"):", buf)

if err != nil {

return nil, err

}

}

result, err = walletImport(name, hex.EncodeToString(newPrivateKey2), password, accessKey, plainText)

if err != nil {

logger.Error("Cannot import wallet", "error", err)

}

} else {

result, err = walletImport(name, privateKey, password, accessKey, plainText)

if err != nil {

logger.Error("Cannot import wallet", "error", err)

}

}

return

}

// WalletList - list wallet of local

func WalletList(pageNum uint64) (result \*WalletListResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_walletList")

result, err = walletList(pageNum)

if err != nil {

logger.Error("Cannot list wallet", "error", err)

}

return

}

// WalletTransfer - transfer token

func WalletTransfer(name, accessKey string, walletParams TransferParam) (result \*TransferResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_transfer", "name", name, "gasLimit", walletParams.GasLimit, "note", walletParams.Note, "to", walletParams.To, "Value", walletParams.Value)

if err = checkName(name); err != nil {

return

}

//parse gasLimit

gasLimit, err := requireUint64(walletParams.GasLimit)

if err != nil {

return

}

// check value

if \_, err = requireUint64(walletParams.Value); err != nil {

return

}

// check smcAddress

if err = checkAddress(crypto.GetChainId(), walletParams.SmcAddress); err != nil {

return

}

// check to address

if err = checkAddress(crypto.GetChainId(), walletParams.To); err != nil {

return

}

result, err = transfer(name, accessKey, gasLimit, walletParams)

if err != nil {

logger.Error("Cannot transfer", "error", err)

}

return

}

// WalletTransferOffline - pack transfer transaction offline

func WalletTransferOffline(name, accessKey string, walletParams TransferOfflineParam) (result \*TransferOfflineResult, err error) {

logger := common.GetLogger()

defer common.FuncRecover(logger, &err)

logger.Trace("gi\_transferOffline", "name", name, "gasLimit", walletParams.GasLimit, "note", walletParams.Note, "to", walletParams.To, "Value", walletParams.Value)

if err = checkName(name); err != nil {

return

}

//parse gasLimit

gasLimit, err := requireUint64(walletParams.GasLimit)

if err != nil {

return

}

// check value

if \_, err = requireUint64(walletParams.Value); err != nil {

return

}

// check smcAddress

if err = checkAddress(crypto.GetChainId(), walletParams.SmcAddress); err != nil {

return

}

// check to address

if err = checkAddress(crypto.GetChainId(), walletParams.To); err != nil {

return

}

result, err = walletTransferOffline(name, accessKey, gasLimit, walletParams)

if err != nil {

logger.Error("Cannot pack transfer transaction", "error", err)

}

return

}

// BlockHeight - get current block height

func BlockHeight() (result \*BlockHeightResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_blockHeight")

result, err = blockHeight()

if err != nil {

common.GetLogger().Error("Cannot get current block height", "error", err)

}

return

}

// Block - get block data with height

func Block(height int64) (result \*BlockResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_block", "height", height)

// if height is 0, set it current height

if height == 0 {

var blkHeight \*BlockHeightResult

if blkHeight, err = blockHeight(); err != nil {

common.GetLogger().Error("Cannot get current block height", "error", err)

return

}

height = blkHeight.LastBlock

} else if height < 0 {

return nil, errors.New("Height cannot be negative ")

}

result, err = block(height)

if err != nil {

common.GetLogger().Error("Cannot get block data", "height", height, "error", err)

}

return

}

// Transaction - get transaction data with txHash

func Transaction(txHash string) (result \*TxResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_transaction", "txHash", txHash)

if txHash == "" {

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

}

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

txHash = txHash[2:]

}

result, err = transaction(txHash, nil)

if err != nil {

common.GetLogger().Error("Cannot get transaction data", "error", err)

}

return

}

// Balance - get balance of account address

func Balance(address keys.Address) (result \*BalanceResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_balance", "address", address)

if address == "" {

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

}

if err = checkAddress(crypto.GetChainId(), address); err != nil {

return

}

result, err = balance(address)

if err != nil {

common.GetLogger().Error("Cannot get balance", "error", err)

}

return

}

// BalanceOfToken - get balance of account address and token address

func BalanceOfToken(address, tokenAddress keys.Address, tokenName string) (result \*BalanceResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_balanceOfToken", "address", address, "tokenAddress", tokenAddress, "tokenName", tokenName)

if address == "" {

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

}

if err = checkAddress(crypto.GetChainId(), address); err != nil {

return

}

if tokenAddress != "" {

if err = checkAddress(crypto.GetChainId(), tokenAddress); err != nil {

return

}

} else if tokenName == "" {

return nil, errors.New("TokenAddress and TokenName cannot empty with both ")

}

result, err = balanceOfToken(address, tokenAddress, tokenName)

if err != nil {

common.GetLogger().Error("Cannot get balance of token", "error", err)

}

return

}

// AllBalance - get all token balance of account address

func AllBalance(address keys.Address) (result \*[]AllBalanceItemResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_allBalance", "address", address)

if err = checkAddress(crypto.GetChainId(), address); err != nil {

return

}

result, err = allBalance(address)

if err != nil {

common.GetLogger().Error("Cannot get all balance", "error", err)

}

return

}

// Nonce - get nonce of account address

func Nonce(address keys.Address) (result \*NonceResult, err error) {

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_nonce", "address", address)

if err = checkAddress(crypto.GetChainId(), address); err != nil {

return

}

result, err = nonce(address)

if err != nil {

common.GetLogger().Error("Cannot get nonce", "error", err)

}

return

}

// CommitTx - commit transaction

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

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_commitTx", "tx", tx)

if tx == "" {

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

}

result, err = commitTx(tx)

if err != nil {

common.GetLogger().Error("Cannot commit tx", "error", err)

}

return

}

// Version - return current app version

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

defer common.FuncRecover(common.GetLogger(), &err)

common.GetLogger().Trace("gi\_version")

var version []byte

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

if err != nil {

common.GetLogger().Error("Read version file error", "error", err)

return

}

result = new(VersionResult)

NewVersion := string(version)

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

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

result.Version = NewVersion

return

}

func PackAndSignTx(nonce, gasLimit uint64, note, tokenAddress, toAddress string, value []byte, name, accessKey string) (string, error) {

var mi MethodInfo

var err error

methodId := atm.CalcMethodId(prototype.TtTransfer)

//dataBytes, \_ := hex.DecodeString(string(methodId))

mi.MethodID = binary.BigEndian.Uint32(methodId)

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

itemsBytes = append(itemsBytes, []byte(toAddress))

itemsBytes = append(itemsBytes, value)

mi.ParamData, err = rlp.EncodeToBytes(itemsBytes)

if err != nil {

return "", err

}

data, err := rlp.EncodeToBytes(mi)

if err != nil {

return "", err

}

tx1 := NewTransaction(nonce, gasLimit, note, tokenAddress, data)

return tx1.TxGen(name, accessKey)

}

func NewTransaction(nonce uint64, gasLimit uint64, note string, to keys.Address, data []byte) BcbXTransaction {

tx := BcbXTransaction{

Nonce: nonce,

GasLimit: gasLimit,

Note: note,

To: to,

Data: data,

}

return tx

}

// 定义生成交易的接口函数，其中tx.Data已经按RLP进行编码

//返回构造好的交易数据，MAC.Version.Payload.<1>.Signature，Payload和Signature格式是RLP编码后的HexString

func (tx \*BcbXTransaction) TxGen(name, accessKey string) (string, error) {

//RLP编码tx

size, r, err := rlp.EncodeToReader(tx)

if err != nil {

return "", err

}

txBytes := make([]byte, size)

\_, \_ = r.Read(txBytes)

sigInfo, err := SignData(name, accessKey, txBytes)

if err != nil {

return "", err

}

//RLP编码签名信息

size, r, err = rlp.EncodeToReader(sigInfo)

if err != nil {

return "", err

}

sigBytes := make([]byte, size)

\_, \_ = r.Read(sigBytes) //转换为字节流

txString := base58.Encode(txBytes)

sigString := base58.Encode(sigBytes)

MAC := string(crypto.GetChainId()) + "<tx>"

Version := "v1"

SignerNumber := "<1>"

return MAC + "." + Version + "." + txString + "." + SignerNumber + "." + sigString, nil

}

func SignData(name, accessKey string, data []byte) (\*types.Ed25519Sig, error) {

if name == "" || accessKey == "" {

return nil, errors.New("user name and accessKey cannot to te empty")

}

if name != "" && len(name) > 40 {

return nil, errors.New("user name length only can be 1-40")

}

if len(data) <= 0 {

return nil, errors.New("user data which wants be signed length needs more than 0")

}

accessKeyBytes := base58.Decode(accessKey)

acct, err2 := db.Account(name, accessKeyBytes)

if acct == nil {

return nil, err2

}

priKey := crypto.PrivKeyEd25519FromBytes(acct.PrivateKey)

pubKey := priKey.PubKey()

sigInfo := types.Ed25519Sig{

SigType: "ed25519",

PubKey: pubKey.(crypto.PubKeyEd25519),

SigValue: priKey.Sign(data).(crypto.SignatureEd25519),

}

return &sigInfo, nil

}

var cdc = amino.NewCodec()

func walletCreate(name, password string) (result \*WalletCreateResult, err error) {

logger := common.GetLogger()

acct, accessKey, err := newAccount(name, password)

if err != nil {

logger.Info(err.Error())

return

}

err = acct.Save(accessKey)

if err != nil {

return

}

result = new(WalletCreateResult)

result.AccessKey = base58.Encode(accessKey)

result.WalletAddress = acct.Address

return

}

func walletExport(name, password, accessKey string, plainText bool) (result \*WalletExportResult, err error) {

accessKeyBytes := base58.Decode(accessKey)

acct, err := db.Account(name, accessKeyBytes)

if err != nil {

fmt.Println("Failed to get account from database, please name or accessKey.")

return

}

priKeyBytes, err := algorithm.DecryptWithPassword(acct.EncPrivateKey, []byte(password), accessKeyBytes)

if err != nil {

fmt.Println("Decrypt Password failed, please check password.")

return

}

result = new(WalletExportResult)

result.WalletAddress = acct.Address

if plainText {

result.PrivateKey = hex.EncodeToString(priKeyBytes)

} else {

result.PrivateKey = hex.EncodeToString(acct.EncPrivateKey)

}

return

}

func walletImport(name, privateKey, password, accessKey string, plainText bool) (result \*WalletImportResult, err error) {

result = new(WalletImportResult)

isExist, \_ := db.IsExist(name)

if isExist {

return nil, errors.New("The account of " + name + " is already exist!")

}

priKeyWithPWBytes, err := hex.DecodeString(privateKey)

if err != nil {

return result, errors.New(" The format of privateKey is wrong")

}

var priKeyBytes []byte

var accessKeyBytes []byte

if plainText {

accessKeyBytes = crypto.CRandBytes(32)

priKeyBytes = priKeyWithPWBytes

accessKey = base58.Encode(accessKeyBytes)

} else {

if accessKey == "" {

return nil, errors.New(" The accessKey can not be empty")

}

accessKeyBytes = base58.Decode(accessKey)

priKeyBytes, err = algorithm.DecryptWithPassword(priKeyWithPWBytes, []byte(password), accessKeyBytes)

if err != nil {

return

}

}

encPrivateKey := algorithm.EncryptWithPassword(priKeyBytes, []byte(password), accessKeyBytes)

cfg := common.GetConfig()

priKey := crypto.PrivKeyEd25519FromBytes(priKeyBytes)

address := priKey.PubKey().Address(cfg.ChainID)

acct := Account{

Name: name,

Address: address,

EncPrivateKey: encPrivateKey,

PrivateKey: priKeyBytes,

}

err = acct.Save(accessKeyBytes)

if err != nil {

return

}

result.WalletAddress = address

result.AccessKey = accessKey

return

}

func transfer(name, accessKey string, gasLimit uint64, walletParams TransferParam) (result \*TransferResult, err error) {

config := common.GetConfig()

result = new(TransferResult)

accessKeyBytes := base58.Decode(accessKey)

acct, err := db.Account(name, accessKeyBytes)

if err != nil {

return

}

// 获取nonce

nonceResult, err := nonce(acct.Address)

if err != nil {

return

}

value := bignumber.NewNumberString(walletParams.Value)

var txStr string

if config.ChainVersion == "1" {

txStr, err = PackAndSignTx(nonceResult.Nonce, gasLimit, walletParams.Note, walletParams.SmcAddress, walletParams.To, value.Bytes(), name, accessKey)

if err != nil {

return nil, err

}

} else if config.ChainVersion == "2" {

var method uint32 = 0x44D8CA60

v := bn.NewNumberStringBase(walletParams.Value, 10)

V2Paramss := []interface{}{walletParams.To, v}

prikey := "0x" + hex.EncodeToString(acct.PrivateKey)

txStr = GenerateTx(walletParams.SmcAddress, method, V2Paramss, nonceResult.Nonce, int64(gasLimit), walletParams.Note, prikey)

} else {

return nil, errors.New("ChainVersion wrong, please check!")

}

commitResult := new(types2.ResultBroadcastTxCommit)

param := map[string]interface{}{"tx": []byte(txStr)}

err = common.DoHttpRequestAndParseExBlock(config.NodeAddrSlice, "broadcast\_tx\_commit", param, commitResult)

if err != nil {

return

}

if commitResult.CheckTx.Code != 200 {

result.Log = commitResult.CheckTx.Log

result.Code = commitResult.CheckTx.Code

} else {

result.Log = commitResult.DeliverTx.Log

result.Code = commitResult.DeliverTx.Code

}

result.Fee = commitResult.DeliverTx.Fee

result.Height = commitResult.Height

result.TxHash = "0x" + hex.EncodeToString(commitResult.Hash)

return

}

func walletTransferOffline(name, accessKey string, gasLimit uint64, walletParams TransferOfflineParam) (result \*TransferOfflineResult, err error) {

config := common.GetConfig()

value := bignumber.NewNumberString(walletParams.Value)

accessKeyBytes := base58.Decode(accessKey)

acct, err := db.Account(name, accessKeyBytes)

if err != nil {

return

}

var txStr string

if config.ChainVersion == "1" {

txStr, err = PackAndSignTx(walletParams.Nonce, gasLimit, walletParams.Note, walletParams.SmcAddress, walletParams.To, value.Bytes(), name, accessKey)

if err != nil {

return nil, err

}

} else if config.ChainVersion == "2" {

var method uint32 = 0x44D8CA60

v := bn.NewNumberStringBase(walletParams.Value, 10)

V2Paramss := []interface{}{walletParams.To, v}

prikey := "0x" + hex.EncodeToString(acct.PrivateKey)

txStr = GenerateTx(walletParams.SmcAddress, method, V2Paramss, walletParams.Nonce, int64(gasLimit), walletParams.Note, prikey)

} else {

return nil, errors.New("ChainVersion wrong, please check!")

}

result = new(TransferOfflineResult)

result.Tx = txStr

return

}

//GenerateTx generate tx with one contract method request

func GenerateTx(contract types.Address, method uint32, V2Paramss []interface{}, nonce uint64, gaslimit int64, note string, privKey string) string {

items := tx3.WrapInvokeParams(V2Paramss...)

message := types.Message{

Contract: contract,

MethodID: method,

Items: items,

}

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

return tx3.WrapTx(payload, privKey)

}