var RootCmd = &cobra.Command{

Use: "robot",

Short: "robot",

Long: "robot",

}

var (

password string

)

func Execute() error {

addFlags()

addCommand()

return RootCmd.Execute()

}

// start method

var startCmd = &cobra.Command{

Use: "start",

Short: "Robot start",

Long: "Robot start",

Args: cobra.ExactArgs(0),

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

return start(password)

},

}

// genKey method

var genKeyCmd = &cobra.Command{

Use: "genkey",

Short: "create keyPair",

Long: "create keyPair",

Args: cobra.ExactArgs(0),

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

return genKey(password)

},

}

func startCmdFlags() {

startCmd.PersistentFlags().StringVarP(&password, "password", "p", "", "robot start")

}

func genKeyCmdFlags() {

genKeyCmd.PersistentFlags().StringVarP(&password, "password", "p", "", "create keyPair")

}

func addFlags() {

startCmdFlags()

genKeyCmdFlags()

}

func addCommand() {

RootCmd.AddCommand(startCmd)

RootCmd.AddCommand(genKeyCmd)

}

func Error(s string) {

fmt.Printf("ERROR! -- %v\n", s)

os.Exit(1)

}

func start(password string) error {

var err error

if password == "" {

password, err = speakeasy.Ask("Enter password :")

if err != nil {

return err

}

}

if !common.CheckPassword(password) {

Error("Password contains by [small letters, capital letters, numbers, Special characters of ASCII 33 through 126] and length must be [8-40] ")

}

err = common.LoadRobotConfig()

if err != nil {

return err

}

common.InitLogger()

err = db.InitDB()

if err != nil {

return err

}

http.Init(TaskManager.ApplyNotify)

go http.HTTPServer()

secretByte := crypto.Sha256([]byte(password))

secretDBByte := db.GetPassword()

if !bytes.Equal(secretByte, secretDBByte) {

Error("password wrong")

}

priKey := getPrvKey(password)

types.PriKey = crypto.GenPrivKeyEd25519FromSecret(priKey)

//查找是否有存商户信息,如果没有则获取商户信息然后保存

initialAccountNum := 10

accountMaxNum := db.GetAccountMaxNum()

if accountMaxNum < uint64(initialAccountNum-1) {

err := initMerchantAddr(uint64(initialAccountNum-1) - accountMaxNum)

if err != nil {

Error(fmt.Sprintf("InitMerchantAddr fail: %s", err.Error()))

}

}

tag := db.GetTag()

if !tag {

fmt.Printf("Did you transfer to these 10 accounts? y/n ")

var hasTransfer string

\_, err := fmt.Scanf("%s", &hasTransfer)

if err != nil {

return err

}

switch hasTransfer {

case "y":

db.SetTag(true)

case "n":

Error("You must transfer to these 10 accounts first ")

default:

Error("Invalid choose ")

}

}

err = excel.InitExcelConfig()

if err != nil {

return err

}

common.GetLogger().Info("Start", "ExcelConfigs", excel.ExConfigs)

taskManager := TaskManager.NewTaskMgr()

taskManager.Start()

return nil

}

func genKey(password string) error {

var err error

if password == "" {

password, err = checkPassword(

"Enter password : ",

"Repeat password : ")

if err != nil {

return err

}

}

if !common.CheckPassword(password) {

Error("Password contains by [small letters, capital letters, numbers, Special characters of ASCII 33 through 126] and length must be [8-40] ")

}

err = common.LoadRobotConfig()

if err != nil {

return err

}

common.InitLogger()

log := common.GetLogger()

err = db.InitDB()

if err != nil {

return err

}

pwdDBByte := db.GetPassword()

defer db.Close()

if len(pwdDBByte) == 0 {

log.Debug("GenKey", "stage", "save password")

updatePassword(password)

} else {

fmt.Printf("Are you sure to update password? y/n ")

var ifUpdate string

\_, err := fmt.Scanf("%s", &ifUpdate)

if err != nil {

return err

}

switch ifUpdate {

case "y":

fmt.Println("update success")

updatePassword(password)

case "n":

os.Exit(1)

default:

Error("Invalid choose ")

}

}

log.Info("GenKey Finish")

return nil

}

func createKeyPair(password string) (crypto.PubKey, []byte, string) {

rand.Seed(time.Now().UnixNano())

randomNum := rand.Int

random := strconv.Itoa(randomNum())

//todo 加上本机的Mac地址

str := "security" + password + random + "robot"

privKey := crypto.Sha256([]byte(str))

return crypto.GenPrivKeyEd25519FromSecret(privKey).PubKey(), privKey, random

}

func updatePassword(password string) {

pubKey, \_, random := createKeyPair(password)

pwdByte := crypto.Sha256([]byte(password))

fmt.Printf("pubkey: %s \n", pubKey)

db.SetPassword(pwdByte)

db.SetRandom(random)

db.SetPubKey(pubKey.Bytes())

file, \_ := os.OpenFile("history.txt", os.O\_CREATE|os.O\_APPEND|os.O\_RDWR, 0660)

defer file.Close()

str := "time:" + time.Now().String() + " " + "PubKey:" + fmt.Sprint(pubKey) + "\n"

\_, err := file.WriteString(str)

if err != nil {

panic(err)

}

}

func initMerchantAddr(createAccountNum uint64) error {

// 初始账户个数

for num := uint64(0); num <= createAccountNum; num++ {

userAccount, addr, index, err := http.CreateAccount()

if err != nil {

common.GetLogger().Error("Create account fail", "err", err)

return err

}

// 根据商户地址信息保存到数据库

db.SetAccountName(addr, userAccount)

// 保存最大账户数

db.SetAccountMaxNum(index)

fmt.Printf("userAccount:%s ", userAccount)

fmt.Printf("accountAddr:%s \n", addr)

}

return nil

}

func checkPassword(prompt1, prompt2 string) (string, error) {

buf := bufio.NewReader(os.Stdin)

pass1, err := getPassword(prompt1, buf)

if err != nil {

return "", err

}

pass2, err := getPassword(prompt2, buf)

if err != nil {

return "", err

}

if pass1 != pass2 {

return "", errors.New("Password don't match \n")

}

return pass2, nil

}

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

pass, err = speakeasy.Ask(prompt)

if err != nil {

return "", err

}

return pass, nil

}

type RobotConfig struct {

DBName string `yaml:"dbName"`

DBPort string `yaml:"dbPort"`

ExcelFile string `yaml:"excelFile"`

MerchantPubKey string `yaml:"merchantPubKey"`

MerchantID string `yaml:"merchantID"`

WalletURL string `yaml:"walletURL"`

LoggerFile bool `yaml:"loggerFile"`

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 (

yamlPath = ".config/robot.yaml"

)

var (

RobotConfig \*config.RobotConfig

pubKey crypto.PubKey

logger log.Loggerf

)

func GetConfig() \*config.RobotConfig {

return RobotConfig

}

func InitLogger() {

if RobotConfig == nil {

panic("must init robot config")

}

logger = log.NewTMLogger("./log", "robot")

logger.SetOutputToScreen(false)

logger.SetOutputToFile(RobotConfig.LoggerFile)

logger.AllowLevel(RobotConfig.LoggerLevel)

}

func GetLogger() log.Loggerf {

return logger

}

func LoadRobotConfig() error {

err := config.InitConfig(&RobotConfig, yamlPath)

if err != nil {

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

}

pubKeyBytes, err := hex.DecodeString(RobotConfig.MerchantPubKey)

if err != nil {

return err

}

pubKey = crypto.PubKeyEd25519FromBytes(pubKeyBytes)

return nil

}

func CreateName(index uint64) string {

return fmt.Sprintf("%s%d", types.AccountPrefix, index)

}

func GetMerchantPubKey() crypto.PubKey {

return pubKey

}

func CheckPassword(password string) (flag bool) {

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

count := 0

number := false

upper := false

lower := false

special := false

other := true

for \_, c := range password {

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 <= 40

return flag

}

var (

sdb \*gidb.GILevelDB

once sync.Once

)

func InitDB() (err error) {

once.Do(func() {

conf := robot.GetConfig()

sdb, err = gidb.OpenDB(conf.DBName, "", conf.DBPort)

})

return err

}

func Close() {

sdb.Close()

}

func SetRandom(random string) {

key := KeyOfRandom()

value, err := jsoniter.Marshal(random)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func GetRandom() string {

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

if err != nil {

panic(err)

}

random := ""

err = jsoniter.Unmarshal(value, &random)

if err != nil {

panic(err)

}

return random

}

func GetAccountMaxNum() uint64 {

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

if err != nil {

panic(err)

}

if len(value) == 0 {

return 0

}

var num uint64

err = jsoniter.Unmarshal(value, &num)

if err != nil {

panic(err)

}

return num

}

func SetAccountMaxNum(num uint64) {

key := KeyOfAccountMaxNum()

value, err := jsoniter.Marshal(num)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func SetPassword(password []byte) {

key := KeyOfPassword()

value, err := jsoniter.Marshal(password)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func GetPassword() []byte {

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

if err != nil {

panic(err)

}

if len(value) == 0 {

return value

}

password := make([]byte, 0)

err = jsoniter.Unmarshal(value, &password)

if err != nil {

panic(err)

}

return password

}

func SetPubKey(pubkey []byte) {

key := KeyOfPubKey()

value, err := jsoniter.Marshal(pubkey)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func GetPubKey() []byte {

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

if err != nil {

panic(err)

}

pubkey := make([]byte, 0)

err = jsoniter.Unmarshal(value, &pubkey)

if err != nil {

panic(err)

}

return pubkey

}

func SetCompletedDailyTask(date string, newAccountCount, transferCount uint64) {

key := KeyOfCompletedDailyTask(date)

t := types.DailyTask{

TransferCount: transferCount,

NewAccountCount: newAccountCount,

}

value, err := jsoniter.Marshal(t)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func GetCompletedDailyTask(date string) (newAccountCount, transferCount uint64) {

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

if err != nil {

panic(err)

}

if len(value) == 0 {

return 0, 0

}

t := types.DailyTask{}

err = jsoniter.Unmarshal(value, &t)

if err != nil {

panic(err)

}

return t.NewAccountCount, t.TransferCount

}

func SetAccountName(addr, name string) {

key := KeyOfAccountName(name)

value, err := jsoniter.Marshal(addr)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func GetAccountAddr(accName string) string {

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

if err != nil {

panic(err)

}

if len(value) == 0 {

return ""

}

t := ""

err = jsoniter.Unmarshal(value, &t)

if err != nil {

panic(err)

}

return t

}

func GetTag() bool {

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

if err != nil {

panic(err)

}

if len(value) == 0 {

return false

}

var tag bool

err = jsoniter.Unmarshal(value, &tag)

if err != nil {

panic(err)

}

return tag

}

func SetTag(tag bool) {

key := KeyOfTag()

value, err := jsoniter.Marshal(tag)

if err != nil {

panic(err)

}

if err = sdb.SetSync([]byte(key), value); err != nil {

panic(err)

}

}

func InitExcelConfig() error {

ExConfigs = map[string]ExConfig{}

err := excelConfig.InitConfig(ExConfigs, common.RobotConfig.ExcelFile)

if err != nil {

common.GetLogger().Error("InitExcelConfig fail", "error", err)

}

ifExcelExit, \_ := fs.PathExists(common.RobotConfig.ExcelFile)

if ifExcelExit {

fileInfo, \_ := os.Stat(common.RobotConfig.ExcelFile)

common.GetLogger().Debug("Get fileInfo success", "fileInfo", fileInfo)

lastUpdateTime = fileInfo.ModTime().UTC().String()

common.GetLogger().Debug("Get fileInfo success", "lastUpdateTime", lastUpdateTime)

}

go checkIsUpdate()

return nil

}

func (e \*ExConfig) InitConfig(c map[string]ExConfig, configFile string) error {

xlsx, err := excelize.OpenFile(configFile)

if err != nil {

return err

}

rows, err := xlsx.GetRows("Sheet1")

if err != nil {

return err

}

for \_, row := range rows {

if strings.Contains(row[0], ".") &&

strings.Count(row[0], ".") == 2 {

newAccountCount, err := strconv.ParseUint(row[1], 10, 64)

if err != nil {

panic(err)

}

transferCount, err := strconv.ParseUint(row[2], 10, 64)

if err != nil {

panic(err)

}

var excel ExConfig

excel.Date = row[0]

excel.NewAccountCount = newAccountCount

excel.TransfCount = transferCount

if newAccountCount >= transferCount {

fmt.Println(newAccountCount)

fmt.Println(transferCount)

fmt.Println("交易笔数必须大于新增账户数")

os.Exit(1)

}

c[row[0]] = excel

}

}

return nil

}

func GetTodayTask(date string) (dailyTask types.DailyTask) {

dailyTask.NewAccountCount = ExConfigs[date].NewAccountCount

dailyTask.TransferCount = ExConfigs[date].TransfCount

return

}

func HasExcelUpdate() bool {

if isExcelUpdate {

isExcelUpdate = false

return true

}

return false

}

func checkIsUpdate() {

for {

common.GetLogger().Debug("checkIsUpdate", "stage", "update")

fileInfo, \_ := os.Stat(common.RobotConfig.ExcelFile)

common.GetLogger().Debug("checkIsUpdate", "fileInfo", fileInfo)

ifFileExit, \_ := fs.PathExists(common.RobotConfig.ExcelFile)

if ifFileExit {

newUpdateTime := fileInfo.ModTime().UTC().String()

if lastUpdateTime != newUpdateTime {

ExConfigs = map[string]ExConfig{}

err := excelConfig.InitConfig(ExConfigs, common.RobotConfig.ExcelFile)

if err != nil {

common.GetLogger().Debug("checkIsUpdate", "error", err)

os.Exit(1)

}

common.GetLogger().Debug("checkIsUpdate", "lastUpdateTime", lastUpdateTime)

common.GetLogger().Debug("checkIsUpdate", "newUpdateTime", newUpdateTime)

common.GetLogger().Debug("checkIsUpdate", "ExcelConfigs", ExConfigs)

lastUpdateTime = newUpdateTime

isExcelUpdate = true

}

}

common.GetLogger().Debug("checkIsUpdate", "ifFileExit", ifFileExit)

ticker := time.NewTicker(time.Minute \* 5)

<-ticker.C

}

}

type ApplyCallBack func(serial string, status int64, message string, from string)

var (

applyFunc ApplyCallBack

)

func Init(applyF ApplyCallBack) {

applyFunc = applyF

}

func CreateAccount() (string, types2.Address, uint64, error) {

// prepare data

index := db.GetAccountMaxNum()

if index == 0 {

addr := db.GetAccountAddr(common.CreateName(index))

if len(addr) != 0 {

index++

}

} else {

index++

}

userAccount := common.CreateName(index)

urlPrefix := types.WalletAPIs["create"]

params := types.CreateParams{

ChainType: types.ChainType,

UserAccount: userAccount,

}

data, err := post(urlPrefix, params, types.PriKey, common.GetMerchantPubKey(), 5)

if err != nil {

return "", "", 0, err

}

var resp types.CreateData

err = jsoniter.Unmarshal(data, &resp)

if err != nil {

return "", "", 0, err

}

return userAccount, resp.Address, index, nil

}

func QueryAccountBalance(address types2.Address) (float64, error) {

urlPrefix := types.WalletAPIs["query"]

params := types.QueryParams{

Address: address,

TokenType: types.TokenType,

}

type AccountBal struct {

Balance float64 `json:"confirmBalance"`

Height int64 `json:"height"`

}

var bal AccountBal

data, err := post(urlPrefix, params, types.PriKey, common.GetMerchantPubKey(), 5)

if err != nil {

return 0, err

}

err = jsoniter.Unmarshal(data, &bal)

if err != nil {

return 0, err

}

return bal.Balance, nil

}

func Apply(userAccount, to types2.Address, value string) (string, error) {

serial := serial()

from := db.GetAccountAddr(userAccount)

common.GetLogger().Debug("Apply", "fromAddr", from, "userAccount", userAccount, "to", to)

urlPrefix := types.WalletAPIs["apply"]

params := types.ApplyParams{

{

Serial: serial,

TokenType: types.TokenType,

UserAccount: userAccount,

From: from,

To: to,

Value: value,

Memo: "",

},

}

\_, err := post(urlPrefix, params, types.PriKey, common.GetMerchantPubKey(), 5)

if err != nil {

return "", err

}

return serial, nil

}

func ApplyNotify(w http.ResponseWriter, r \*http.Request) {

p, err := ioutil.ReadAll(r.Body)

type Response struct {

Code int `json:"code"`

Message string `json:"message"`

}

sigStr := r.Header.Get("signature")

if err := verifySig(sigStr, common.GetMerchantPubKey(), p); err != nil {

common.GetLogger().Error("ApplyNotify verify sig failed", "err", err, "sigStr", sigStr)

resp := Response{

Code: -1,

Message: "check signature failed",

}

rb, \_ := jsoniter.Marshal(resp)

\_, \_ = w.Write(rb)

return

}

var notify types.ApplyNotifyData

err = jsoniter.Unmarshal(p, &notify)

if err != nil {

common.GetLogger().Error("ApplyNotify json unmarshal failed", "err", err)

resp := Response{

Code: -2,

Message: "unmarshal notify data failed",

}

rb, \_ := jsoniter.Marshal(resp)

\_, \_ = w.Write(rb)

return

}

applyFunc(notify.Params.ThirdSerial, notify.Params.Status, notify.Params.Message, notify.Params.From)

resp := Response{

Code: 0,

Message: "ok",

}

rb, \_ := jsoniter.Marshal(resp)

\_, \_ = w.Write(rb)

}

func serial() string {

// 随机6字节

suffixBytes := crypto.CRandBytes(6)

return "robot\_" + hex.EncodeToString(suffixBytes)

}

// Post sign request body and post

func post(urlPrefix string, params interface{}, priKey crypto.PrivKey, pubKey crypto.PubKey, timeout time.Duration) ([]byte, error) {

reqBody := types.HTTPRequest{

Timestamp: time.Now().Unix(),

Params: params,

}

bodyBytes, err := json.Marshal(reqBody)

if err != nil {

return nil, err

}

// 为数据签名并构造新url及调用对象

sig := priKey.Sign(bodyBytes).(crypto.SignatureEd25519)

sigStr := hex.EncodeToString(sig[:])

fullUrl := common.GetConfig().WalletURL + urlPrefix

tr := new(http.Transport)

tr.DisableKeepAlives = true

tr.IdleConnTimeout = time.Second \* 5

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

client := http.Client{Transport: tr, Timeout: timeout \* time.Second}

request, err := http.NewRequest("POST", fullUrl, bytes.NewBuffer(bodyBytes))

if err != nil {

return nil, err

}

request.Header.Add("signature", sigStr)

request.Header.Add("merchantId", common.GetConfig().MerchantID)

request.Header.Add("Content-Type", "application/json")

resp, err := client.Do(request)

if err != nil {

return nil, err

}

defer func() {

\_ = resp.Body.Close()

}()

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

if err != nil {

return nil, err

}

// 取出返回的数据内容

var r types.HTTPResponse

err = jsoniter.Unmarshal(responseBytes, &r)

if err != nil {

return nil, err

}

if r.Code != 0 {

return nil, errors.New(r.Message)

}

// 验证返回商户签名

responseSigStr := resp.Header.Get("signature")

if err = verifySig(responseSigStr, pubKey, responseBytes); err != nil {

return nil, err

}

dataBytes, err := jsoniter.Marshal(r.Data)

if err != nil {

return nil, err

}

return dataBytes, nil

}

func verifySig(sigStr string, pubKey crypto.PubKey, data []byte) error {

sigBytes, err := hex.DecodeString(sigStr)

if err != nil {

return err

}

signature := crypto.SignatureEd25519FromBytes(sigBytes)

if !pubKey.VerifyBytes(data, signature) {

return errors.New("check signature failed, signature: " + sigStr)

}

return nil

}

type DailyTask struct {

TransferCount uint64

NewAccountCount uint64

Date string

}

var SenderAddrMap sync.Map

func init() {

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

}

func (dt \*DailyTask) Start() {

for {

randomList := genRandomList(\*dt)

common.GetLogger().Debug("RandomList", "list", randomList)

if len(randomList) == 0 {

return

}

var runTimer \*time.Timer

for i, task := range randomList {

subInterval := task.WaitTime

if i != 0 {

subInterval = task.WaitTime - randomList[i-1].WaitTime

}

runTimer = time.NewTimer(time.Second \* time.Duration(subInterval))

<-runTimer.C

go runTask(task)

if i == len(randomList)-1 {

return

}

if hasExcelUpdate() {

if !dt.updateTodayTasks() {

// 如果今天任务已经完成，即使修改了 excel 也不继续了

return

}

break

}

}

}

}

func (dt \*DailyTask) updateTodayTasks() bool {

// 从 excel 和 db 获取今天剩余得任务量

excelTask := dt.getTodayTaskFromExcel()

today := time.Now().Format(types.TimeLayout)

completedNewAcc, completedTransfer := db.GetCompletedDailyTask(today)

var needNewAccCount, needTransferCount uint64

if excelTask.TransferCount <= completedTransfer {

needTransferCount = 0

} else {

needTransferCount = excelTask.TransferCount - completedTransfer

}

if excelTask.NewAccountCount <= completedNewAcc {

needNewAccCount = 0

} else {

needNewAccCount = excelTask.NewAccountCount - completedNewAcc

}

if needNewAccCount <= 0 && needTransferCount <= 0 {

// 今日任务已经完成

return false

}

if needNewAccCount > 0 && needNewAccCount > needTransferCount {

needTransferCount = needNewAccCount + 1

}

if needNewAccCount > 0 {

dt.NewAccountCount = needNewAccCount

} else {

dt.NewAccountCount = 0

}

if needTransferCount > 0 {

dt.TransferCount = needTransferCount

} else {

dt.TransferCount = 0

}

return true

}

func (dt \*DailyTask) getTodayTaskFromExcel() types.DailyTask {

return excel.GetTodayTask(dt.Date)

}

func hasExcelUpdate() bool {

return excel.HasExcelUpdate()

}

func runTask(task types.PointTask) {

common.GetLogger().Info("runTask", "task", task)

var fromAccName, to, value, senderAddr string

value = getRandomTransferValue()

common.GetLogger().Debug("transfer value", "value", value)

fromAccName, senderAddr = randomSender(value)

common.GetLogger().Debug("transfer sender", "fromAccName", fromAccName, "senderAddr", senderAddr)

if task.NeedNewAccount {

accountName, addr, index, err := http.CreateAccount()

common.GetLogger().Debug("create account", "accountName", accountName, "addr", addr, "index", index, "err", err)

if err != nil {

common.GetLogger().Error("Create account failed", "err", err)

}

to = addr

db.SetAccountName(addr, accountName)

max := db.GetAccountMaxNum()

if index > max {

db.SetAccountMaxNum(index)

}

today := time.Now().Format(types.TimeLayout)

an, tn := db.GetCompletedDailyTask(today)

db.SetCompletedDailyTask(today, an+1, tn)

} else {

to = randomToAddr(fromAccName)

common.GetLogger().Debug("transfer to", "to", to)

}

SenderAddrMap.Store(senderAddr, nil)

common.GetLogger().Debug("transfer call", "fromAccName", fromAccName, "to", to, "value", value)

\_, err := http.Apply(fromAccName, to, value)

if err != nil {

SenderAddrMap.Delete(senderAddr)

common.GetLogger().Error("Transfer failed", "fromAccountName", fromAccName, "toAddress", to, "value", value, "err", err, "senderAddr", senderAddr)

}

}

func randomSender(value string) (string, string) {

fee := 0.00125

maxUserNum := db.GetAccountMaxNum()

valueFloat, err := strconv.ParseFloat(value, 64)

if err != nil {

common.GetLogger().Info("parseFloat failed", "value", value, "err", err, "valueFloat", valueFloat, "fee", fee)

}

for {

random := getRandom(int64(maxUserNum + 1))

userName := common.CreateName(uint64(random))

sender := db.GetAccountAddr(userName)

if \_, ok := SenderAddrMap.Load(sender); ok {

continue

}

bal, err := http.QueryAccountBalance(sender)

if err != nil {

common.GetLogger().Info("query account balance failed", "err", err, "sender", sender, "userName", userName, "bal", bal)

continue

}

if bal-valueFloat-fee > 0 {

return userName, sender

}

}

}

func randomToAddr(fromUserName string) string {

fromAddr := db.GetAccountAddr(fromUserName)

maxUserNum := db.GetAccountMaxNum()

for {

random := getRandom(int64(maxUserNum + 1))

userName := common.CreateName(uint64(random))

to := db.GetAccountAddr(userName)

if to != fromAddr {

return to

}

}

}

// 0.01~1 gi

func getRandomTransferValue() string {

for {

random := getRandom(101)

if random == 0 {

continue

}

if random == 100 {

return "1"

} else {

return "0." + strconv.Itoa(int(random))

}

}

}

func getRandom(max int64) int64 {

return rand.Int63n(max)

}

type TaskManager struct {

TodayTask DailyTask

Date time.Time

}

var taskMgr \*TaskManager

func NewTaskMgr() \*TaskManager {

if taskMgr != nil {

return taskMgr

}

taskMgr = &TaskManager{

Date: time.Now(),

}

return taskMgr

}

func GetTaskMgr() \*TaskManager {

return taskMgr

}

func (tm \*TaskManager) Start() {

common.GetLogger().Info("TaskMgr start")

for {

tm.getDailyTask()

common.GetLogger().Debug("TaskMgr", "task", tm.TodayTask)

tm.TodayTask.Start()

// 今天任务执行结束后，从 excel 获取第二天得任务

// 等到今天凌晨零点时再开始第二天的，如果 excel 有更新需要再次获取今天的任务

if tm.waitTomorrow() {

tm.Date = tm.Date.Add(time.Hour \* 24)

} else {

// 没有到第二天时，Excel 有更改，重新计算今天剩余的任务

}

}

}

func (tm \*TaskManager) waitExcelUpdate(stopChan, excelUpdateChan chan int) {

for {

select {

case <-stopChan:

return

default:

time.Sleep(time.Second \* 10)

if hasExcelUpdate() {

select {

case excelUpdateChan <- 1:

default:

return

}

}

}

}

}

func (tm \*TaskManager) waitTomorrow() bool {

today := tm.Date.Format(types.TimeLayout)

nowDay := time.Now().Format(types.TimeLayout)

if today != nowDay {

return true

}

endTime, \_ := time.ParseInLocation("2006.1.2 15:04:05", today+" 23:59:59", time.Local)

subTime := endTime.Unix() - time.Now().Unix()

var runTimer \*time.Timer

runTimer = time.NewTimer(time.Second \* time.Duration(subTime))

excelUpdateChan := make(chan int)

stopChan := make(chan int)

go tm.waitExcelUpdate(stopChan, excelUpdateChan)

select {

case <-excelUpdateChan:

// excel 有更新，重新执行今天剩余的任务

return false

case <-runTimer.C:

time.Sleep(time.Second \* 1)

select {

case stopChan <- 1:

default:

go func() { stopChan <- 1 }()

return true

}

return true

}

}

func (tm \*TaskManager) getDailyTask() {

today := tm.Date.Format(types.TimeLayout)

dTask := excel.GetTodayTask(today)

tm.TodayTask.NewAccountCount = dTask.NewAccountCount

tm.TodayTask.TransferCount = dTask.TransferCount

tm.TodayTask.Date = today

}

// 转账接口的回调

func ApplyNotify(serial string, status int64, message string, from string) {

SenderAddrMap.Delete(from)

common.GetLogger().Debug("withdraw end", "from", from, "status", status, "serial", serial, "msg", message)

today := time.Now().Format(types.TimeLayout)

if status != 0 {

common.GetLogger().Error("withdraw failed", "from", from, "status", status, "serial", serial, "msg", message)

} else {

an, tn := db.GetCompletedDailyTask(today)

db.SetCompletedDailyTask(today, an, tn+1)

}

}

// 根据今天的任务生成随机时间点列表

func genRandomList(tasks DailyTask) types.PointTaskList {

common.GetLogger().Info("genRandomList", "task.TransferCount", tasks.TransferCount,

"task.NewAccCount", tasks.NewAccountCount, "task.Date", tasks.Date)

if tasks.TransferCount <= 0 {

return types.PointTaskList{}

}

today := tasks.Date

eTime, \_ := time.ParseInLocation("2006.1.2 15:04:05", today+" 23:59:59", time.Local)

subTime := eTime.Unix() - time.Now().Unix()

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

randomList := make(types.PointTaskList, tasks.TransferCount)

for i := 0; i < int(tasks.TransferCount); i++ {

rTime := uint64(rand.Int63n(subTime))

randomList[i] = types.PointTask{

WaitTime: rTime,

}

}

sort.Sort(randomList)

var count uint64

randomLen := int64(len(randomList))

if tasks.NewAccountCount > 0 {

for {

random := uint64(rand.Int63n(randomLen))

if randomList[random].NeedNewAccount {

continue

} else {

randomList[random].NeedNewAccount = true

count++

}

if count == tasks.NewAccountCount {

break

}

}

}

return randomList

}