碳链记·GreenTrace Chain

基于区块链的碳核算和碳交易系统智能合约代码介绍

开发团队：你的外包我来包

目录

合约一：CarbonCoin

1. 合约概述

功能介绍： 碳币合约旨在实现碳信用额度的数字化管理。该合约允许企业之间的碳币和碳额度的发行、查询和转移，支持环境责任的履行和交易。

重要性和应用场景： 在应对气候变化和推广可持续发展政策的背景下，此合约为企业提供了一种有效的工具来管理其碳排放权和环保责任。通过区块链技术，合约确保了交易的透明性和数据的不可篡改性。

2. 合约代码解析

```
package main

import (
	"chainmaker.org/chainmaker/contract-sdk-go/v2/pb/protogo"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sandbox"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sdk"
	"encoding/json"
	"strconv"
)

// CarbonCoinContract 碳币合约实现
type CarbonCoinContract struct{}

// CreditInfo 用于存储和转移的碳币和碳额度信息
type CreditInfo struct {
	CarbonCoin    int       `json:"carbonCoin"`
	CarbonCredit int       `json:"carbonCredit"`
	Token             string `json:"token"`
	Hash              string `json:"hash"`
	PublicKey       string `json:"publicKey"`
}

// InitContract 初始化合约
func (c *CarbonCoinContract) InitContract() protogo.Response {
	return sdk.Success(nil)
}

// UpgradeContract 升级合约
func (c *CarbonCoinContract) UpgradeContract() protogo.Response {
```

```
	return sdk.Success(nil)
}

// InvokeContract 调用合约方法
func (c *CarbonCoinContract) InvokeContract(method string) (result protogo.Response) {

	switch method {
	case "IssueCredit":
		return c.IssueCredit()
	case "QueryCredit":
		return c.QueryCredit()
	case "TransferCredit":
		return c.TransferCredit()
	default:
		return sdk.Error("invalid method")
	}
}

// IssueCredit 分配碳币和碳额度
func (c *CarbonCoinContract) IssueCredit() protogo.Response {
	params := sdk.Instance.GetArgs()
	companyID := string(params["companyID"])
	carbonCoin, err := strconv.Atoi(string(params["carbonCoin"]))
	if err != nil {
		return sdk.Error(err.Error())
	}
	carbonCredit, err := strconv.Atoi(string(params["carbonCredit"]))
	if err != nil {
		return sdk.Error(err.Error())
	}

	// 新增：处理 token, hash, publicKey 参数
	token := string(params["token"])
	hash := string(params["hash"])
	publicKey := string(params["publicKey"])

	creditInfo := CreditInfo{CarbonCoin: carbonCoin, CarbonCredit: carbonCredit,
Token: token, Hash: hash, PublicKey: publicKey}
	creditInfoBytes, err := json.Marshal(creditInfo)
	if err != nil {
		return sdk.Error(err.Error())
	}

	err = sdk.Instance.PutStateByte(companyID, "creditInfo", creditInfoBytes)
```

```
    if err != nil {
        return sdk.Error(err.Error())
    }

    return sdk.Success(nil)
}

// QueryCredit 查询公司的碳币和碳额度
func (c *CarbonCoinContract) QueryCredit() protogo.Response {
    params := sdk.Instance.GetArgs()
    companyID := string(params["companyID"])

    creditInfoBytes, err := sdk.Instance.GetStateByte(companyID, "creditInfo")
    if err != nil {
        return sdk.Error(err.Error())
    }
    if creditInfoBytes == nil {
        return sdk.Error("credit info not found")
    }

    return sdk.Success(creditInfoBytes)
}

// TransferCredit 两公司之间转移碳币和碳额度，包含交易 ID，以及 token, hash,
publicKey
func (c *CarbonCoinContract) TransferCredit() protogo.Response {
    params := sdk.Instance.GetArgs()

    companyAID := string(params["companyAID"])
    companyBID := string(params["companyBID"])
    carbonCoinToTransfer, err := strconv.Atoi(string(params["carbonCoin"]))
    if err != nil {
        return sdk.Error("Invalid carbonCoin: " + err.Error())
    }
    carbonCreditToTransfer, err := strconv.Atoi(string(params["carbonCredit"]))
    if err != nil {
        return sdk.Error("Invalid carbonCredit: " + err.Error())
    }

    // 新增：处理 token, hash, publicKey 参数
    token := string(params["token"])
    hash := string(params["hash"])
    publicKey := string(params["publicKey"])
```

```
    // 获取公司 A 的碳币和碳额度信息
    creditInfoABytes, err := sdk.Instance.GetStateByte(companyAID, "creditInfo")
    if err != nil {
        return sdk.Error(err.Error())
    }
    if creditInfoABytes == nil {
        return sdk.Error("Credit info for company A not found")
    }
    var creditInfoA CreditInfo
    err = json.Unmarshal(creditInfoABytes, &creditInfoA)
    if err != nil {
        return sdk.Error("Failed to unmarshal credit info for company A: " + err.Error())
    }

    // 获取公司 B 的碳币和碳额度信息
    creditInfoBBytes, err := sdk.Instance.GetStateByte(companyBID, "creditInfo")
    if err != nil {
        return sdk.Error(err.Error())
    }
    if creditInfoBBytes == nil {
        return sdk.Error("Credit info for company B not found")
    }
    var creditInfoB CreditInfo
    err = json.Unmarshal(creditInfoBBytes, &creditInfoB)
    if err != nil {
        return sdk.Error("Failed to unmarshal credit info for company B: " + err.Error())
    }

    // 确保公司 A 有足够的碳币和碳额度进行转移
    if creditInfoA.CarbonCoin < carbonCoinToTransfer || creditInfoA.CarbonCredit <
carbonCreditToTransfer {
        return sdk.Error("Company A does not have enough carbon coin or carbon credit
to transfer")
    }

    // 更新公司 A 和公司 B 的碳币和碳额度
    creditInfoA.CarbonCoin -= carbonCoinToTransfer
    creditInfoA.CarbonCredit -= carbonCreditToTransfer
    creditInfoB.CarbonCoin += carbonCoinToTransfer
    creditInfoB.CarbonCredit += carbonCreditToTransfer

    // 更新公司 A 和公司 B 的信息，包括 token, hash, publicKey
    creditInfoA.Token = token
    creditInfoA.Hash = hash
```

```
    creditInfoA.PublicKey = publicKey
    creditInfoB.Token = token
    creditInfoB.Hash = hash
    creditInfoB.PublicKey = publicKey

    // 序列化并保存更新后的公司 A 和公司 B 的信息
    creditInfoABytes, err = json.Marshal(creditInfoA)
    if err != nil {
        return sdk.Error("Failed to marshal updated credit info for company A: " +
err.Error())
    }
    err = sdk.Instance.PutStateByte(companyAID, "creditInfo", creditInfoABytes)
    if err != nil {
        return sdk.Error("Failed to update credit info for company A: " + err.Error())
    }

    creditInfoBBytes, err = json.Marshal(creditInfoB)
    if err != nil {
        return sdk.Error("Failed to marshal updated credit info for company B: " +
err.Error())
    }
    err = sdk.Instance.PutStateByte(companyBID, "creditInfo", creditInfoBBytes)
    if err != nil {
        return sdk.Error("Failed to update credit info for company B: " + err.Error())
    }

    return sdk.Success(nil)
}

func main() {
    err := sandbox.Start(new(CarbonCoinContract))
    if err != nil {
        sdk.Instance.Errorf(err.Error())
    }
}
```

函数详解：

- **InitContract**：
  - **功能**：初始化合约实例。
  - **参数**：无。
  - **返回值**：操作成功的响应。
  - **作用描述**：部署或重置合约时调用，初始化合约状态。
- **UpgradeContract**：
  - **功能**：升级合约。
  - **参数**：无。

    - **返回值**：操作成功的响应。
    - **作用描述**：在合约逻辑或存储需要更新时调用。
- **InvokeContract**：
    - **功能**：根据提供的方法名调用相应功能。
    - **参数**：方法名（字符串）。
    - **返回值**：根据调用的方法返回相应的响应。
    - **作用描述**：是合约的主要入口点，根据传入的方法名调用特定的合约功能。
- **IssueCredit**：
    - **功能**：为指定公司发行碳币和碳额度。
    - **参数**：companyID, carbonCoin, carbonCredit, token, hash, publicKey。
    - **返回值**：操作成功或错误的响应。
    - **作用描述**：允许合约管理员为注册的公司账户增加碳币和碳额度。
- **QueryCredit**：
    - **功能**：查询指定公司的碳币和碳额度。
    - **参数**：companyID。
    - **返回值**：包含碳币和碳额度信息的响应或错误信息。
    - **作用描述**：提供一个接口查询公司账户的当前碳币和碳额度状态。
- **TransferCredit**：
    - **功能**：在两个公司之间转移碳币和碳额度。
    - **参数**：companyAID, companyBID, carbonCoin, carbonCredit, token, hash, publicKey。
    - **返回值**：操作成功或错误的响应。
    - **作用描述**：允许公司之间转移碳币和碳额度，实现碳信用的交易。

3. 主键和存储

状态变量：
- **creditInfo**：存储每个公司的碳币和碳额度信息，包括碳币数量、碳额度、关联的 token、hash 值和公钥。

4. 调用方法

外部调用接口： 通过合约的 **InvokeContract** 方法，外部可以调用 **IssueCredit**、**QueryCredit** 和 **TransferCredit** 方法，实现合约功能的操作。每个方法需要的参数和返回的结果详细说明如上。

5. 安全性和限制

已知风险：
- 操作错误可能导致不正确的碳币和碳额度记录。
- 系统安全性依赖于区块链平台的整体安全性。

使用限制:

- 只有管理员和用户可以发行和转移碳币和碳额度。
- 查询功能公开可供任何参与者使用。

合约二：CarbonModel

1. 合约概述

功能介绍： 碳模型合约旨在存储和查询与碳排放相关的交易数据。此合约允许上传包含碳排放量、交易信息以及其他相关数据的记录，并可以基于交易 ID 查询这些记录。
重要性和应用场景： 此合约对于那些需要精确记录和追踪碳排放量和交易的企业至关重要。它支持环境监管和碳排放交易系统，提供一个可靠的数据基础以确保碳交易的透明性和准确性。

2. 合约代码解析

```
package main

import (
	"encoding/json"

	"chainmaker.org/chainmaker/contract-sdk-go/v2/pb/protogo"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sandbox"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sdk"
)

// CarbonModelContract 合约实现
type CarbonModelContract struct{}

// 数据结构定义
type CarbonData struct {
	CompanyID          string `json:"CompanyID"`
	CompanyType        string `json:"CompanyType"`
	CarbonModelTxid    string `json:"CarbonModelTxid"`
	TransactionData    string `json:"TransactionData"`
	Rate               string `json:"Rate"`
	Emissions          string `json:"Emissions"`
	Hash               string `json:"Hash"`
	PublicKey          string `json:"PublicKey"`
	Signature          string `json:"Signature"`
	DeductCarbonCredit string `json:"DeductCarbonCredit"`
```

```
    GetCarbonCoin          string `json:"GetCarbonCoin"`
    CarbonCredit           string `json:"CarbonCredit"`
    CarbonCoin             string `json:"CarbonCoin"`
}

// InitContract 初始化合约，必须实现的接口
func (c *CarbonModelContract) InitContract() protogo.Response {
    return sdk.Success(nil)
}

// UpgradeContract 升级合约，必须实现的接口
func (c *CarbonModelContract) UpgradeContract() protogo.Response {
    return sdk.Success(nil)
}

// InvokeContract 调用合约，必须实现的接口
func (c *CarbonModelContract) InvokeContract(method string) (result protogo.Response)
{
    // 记录异常结果日志
    defer func() {
        if result.Status != 0 {
            sdk.Instance.Warnf(result.Message)
        }
    }()

    switch method {
    case "UploadData":
        return c.UploadData()
    case "QueryData":
        return c.QueryData()
    default:
        return sdk.Error("invalid method")
    }
}

// UploadData 上传数据
func (c *CarbonModelContract) UploadData() protogo.Response {
    params := sdk.Instance.GetArgs()

    // 组织数据对象
    data := CarbonData{
        CompanyID:          string(params["CompanyID"]),
        CompanyType:        string(params["CompanyType"]),
        CarbonModelTxid:    string(params["CarbonModelTxid"]),
```

```
        TransactionData:       string(params["TransactionData"]),
        Rate:                   string(params["Rate"]),
        Emissions:              string(params["Emissions"]),
        Hash:                   string(params["Hash"]),
        PublicKey:              string(params["PublicKey"]),
        Signature:              string(params["Signature"]),
        DeductCarbonCredit: string(params["DeductCarbonCredit"]),
        GetCarbonCoin:          string(params["GetCarbonCoin"]),
        CarbonCredit:           string(params["CarbonCredit"]),
        CarbonCoin:             string(params["CarbonCoin"]),
    }

    dataBytes, err := json.Marshal(data)
    if err != nil {
        return sdk.Error("Failed to marshal data")
    }

    // 使用 CarbonModelTxid 作为 Key
    txid := params["CarbonModelTxid"]
    if txid == nil || len(txid) == 0 {
        return sdk.Error("CarbonModelTxid is required")
    }

    err = sdk.Instance.PutStateByte(string(txid), "", dataBytes)
    if err != nil {
        return sdk.Error(err.Error())
    }

    return sdk.Success(nil)
}

// QueryData 查询数据
func (c *CarbonModelContract) QueryData() protogo.Response {
    params := sdk.Instance.GetArgs()

    txid := params["CarbonModelTxid"]
    if txid == nil || len(txid) == 0 {
        return sdk.Error("CarbonModelTxid is required")
    }

    dataBytes, err := sdk.Instance.GetStateByte(string(txid), "")
    if err != nil {
        return sdk.Error(err.Error())
    }
```

```
    if dataBytes == nil {
        return sdk.Error("Data not found")
    }

    // 直接返回 JSON 字符串
    return sdk.Success(dataBytes)
}

func main() {
    err := sandbox.Start(new(CarbonModelContract))
    if err != nil {
        sdk.Instance.Errorf(err.Error())
    }
}
```

函数详解:

- **UploadData**:
  - **功能**: 上传关于碳排放和交易的详细数据。
  - **参数**: CompanyID, CompanyType, CarbonModelTxid, TransactionData, Rate, Emissions, Hash, PublicKey, Signature, DeductCarbonCredit, GetCarbonCoin, CarbonCredit, CarbonCoin。
  - **返回值**: 操作成功或错误的响应。
  - **作用描述**: 接收和存储关于公司碳交易的详细信息，为合约提供持久化数据存储。
- **QueryData**:
  - **功能**: 基于交易 ID 查询碳排放和交易数据。
  - **参数**: CarbonModelTxid。
  - **返回值**: 包含碳交易数据的响应或错误信息。
  - **作用描述**: 提供一个接口来查询和检索之前上传的碳交易数据。

3. 主键和存储

状态变量:

- **CarbonModelTxid**: 用作唯一标识符，存储对应的碳交易数据。

4. 调用方法

外部调用接口: 通过合约的 **InvokeContract** 方法，外部可以调用 **UploadData** 和 **QueryData** 方法，实现合约功能的操作。每个方法需要的参数和返回的结果详细说明如上。

5. 安全性和限制

已知风险:

- 数据错误或遗漏可能导致碳排放记录不准确。
- 系统安全性依赖于区块链平台的整体安全性。

使用限制:

- 只有授权的用户或系统可以上传和查询碳交易数据。

合约三：CarbonTrade

1. 合约概述

功能介绍： 碳币合约用于管理碳币和碳额度，允许企业之间进行碳信用的交易。合约提供了发行、查询和转移碳币和碳额度的功能。

重要性和应用场景： 该合约对于碳交易市场和环境责任管理至关重要，帮助企业在满足环保规定的同时，通过碳信用交易优化其经济效益。

2. 合约代码解析

```
package main

import (
	"chainmaker.org/chainmaker/contract-sdk-go/v2/pb/protogo"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sandbox"
	"chainmaker.org/chainmaker/contract-sdk-go/v2/sdk"
	"encoding/json"
	"strconv"
)

// CarbonCoinContract 碳币合约实现
type CarbonCoinContract struct{}

// CreditInfo 用于存储和转移的碳币和碳额度信息
type CreditInfo struct {
	CarbonCoin   int `json:"carbonCoin"`
	CarbonCredit int `json:"carbonCredit"`
}

// InitContract 初始化合约
func (c *CarbonCoinContract) InitContract() protogo.Response {
```

```
    return sdk.Success(nil)
}

// UpgradeContract 升级合约
func (c *CarbonCoinContract) UpgradeContract() protogo.Response {
    return sdk.Success(nil)
}

// InvokeContract 调用合约方法
func (c *CarbonCoinContract) InvokeContract(method string) (result protogo.Response) {

    switch method {
    case "IssueCredit":
        return c.IssueCredit()
    case "QueryCredit":
        return c.QueryCredit()
    case "TransferCredit":
        return c.TransferCredit()
    default:
        return sdk.Error("invalid method")
    }
}

// IssueCredit 分配碳币和碳额度
func (c *CarbonCoinContract) IssueCredit() protogo.Response {
    params := sdk.Instance.GetArgs()
    companyID := string(params["companyID"])
    carbonCoin, err := strconv.Atoi(string(params["carbonCoin"]))
    if err != nil {
        return sdk.Error(err.Error())
    }
    carbonCredit, err := strconv.Atoi(string(params["carbonCredit"]))
    if err != nil {
        return sdk.Error(err.Error())
    }

    creditInfo := CreditInfo{CarbonCoin: carbonCoin, CarbonCredit: carbonCredit}
    creditInfoBytes, err := json.Marshal(creditInfo)
    if err != nil {
        return sdk.Error(err.Error())
    }

    err = sdk.Instance.PutStateByte(companyID, "creditInfo", creditInfoBytes)
    if err != nil {
```

```
            return sdk.Error(err.Error())
        }

        return sdk.Success(nil)
}

// QueryCredit 查询公司的碳币和碳额度
func (c *CarbonCoinContract) QueryCredit() protogo.Response {
        params := sdk.Instance.GetArgs()
        companyID := string(params["companyID"])

        creditInfoBytes, err := sdk.Instance.GetStateByte(companyID, "creditInfo")
        if err != nil {
            return sdk.Error(err.Error())
        }
        if creditInfoBytes == nil {
            return sdk.Error("credit info not found")
        }

        return sdk.Success(creditInfoBytes)
}

func (c *CarbonCoinContract) TransferCredit() protogo.Response {
        params := sdk.Instance.GetArgs()
        sellerLastTxid := string(params["SellerLastTxid"])
        buyerLastTxid := string(params["BuyerLastTxid"])

        companyAID := string(params["companyAID"])
        companyBID := string(params["companyBID"])

        carbonCoinToTransfer, err := strconv.Atoi(string(params["carbonCoin"]))
        if err != nil {
            return sdk.Error("invalid carbonCoin: " + err.Error())
        }
        carbonCreditToTransfer, err := strconv.Atoi(string(params["carbonCredit"]))
        if err != nil {
            return sdk.Error("invalid carbonCredit: " + err.Error())
        }

        creditInfoABytes, err := sdk.Instance.GetStateByte(companyAID, "creditInfo")
        if err != nil {
            return sdk.Error("failed to get credit info for company A: " + err.Error())
        }
        if creditInfoABytes == nil {
```

```
            return sdk.Error("credit info for company A not found")
        }
        var creditInfoA CreditInfo
        err = json.Unmarshal(creditInfoABytes, &creditInfoA)
        if err != nil {
            return sdk.Error("failed to unmarshal credit info for company A: " + err.Error())
        }

        creditInfoBBytes, err := sdk.Instance.GetStateByte(companyBID, "creditInfo")
        if err != nil {
            return sdk.Error("failed to get credit info for company B: " + err.Error())
        }
        if creditInfoBBytes == nil {
            return sdk.Error("credit info for company B not found")
        }
        var creditInfoB CreditInfo
        err = json.Unmarshal(creditInfoBBytes, &creditInfoB)
        if err != nil {
            return sdk.Error("failed to unmarshal credit info for company B: " + err.Error())
        }

        if creditInfoA.CarbonCoin < carbonCoinToTransfer || creditInfoA.CarbonCredit <
carbonCreditToTransfer {
            return sdk.Error("company A does not have enough carbonCoin or carbonCredit
to transfer")
        }
        creditInfoA.CarbonCoin -= carbonCoinToTransfer
        creditInfoA.CarbonCredit -= carbonCreditToTransfer
        creditInfoB.CarbonCoin += carbonCoinToTransfer
        creditInfoB.CarbonCredit += carbonCreditToTransfer

        creditInfoABytes, err = json.Marshal(creditInfoA)
        if err != nil {
            return sdk.Error("failed to marshal updated credit info for company A: " +
err.Error())
        }
        err = sdk.Instance.PutStateByte(companyAID, "creditInfo", creditInfoABytes)
        if err != nil {
            return sdk.Error("failed to update credit info for company A: " + err.Error())
        }

        creditInfoBBytes, err = json.Marshal(creditInfoB)
        if err != nil {
            return sdk.Error("failed to marshal updated credit info for company B: " +
```

```
err.Error())
    }
    err = sdk.Instance.PutStateByte(companyBID, "creditInfo", creditInfoBBytes)
    if err != nil {
        return sdk.Error("failed to update credit info for company B: " + err.Error())
    }

    // Record last transaction IDs and credit information after transfer
    err = sdk.Instance.PutStateByte(companyAID, "SellerLastTxid",
[]byte(sellerLastTxid))
    if err != nil {
        return sdk.Error("failed to record SellerLastTxid for company A: " + err.Error())
    }
    err = sdk.Instance.PutStateByte(companyBID, "BuyerLastTxid",
[]byte(buyerLastTxid))
    if err != nil {
        return sdk.Error("failed to record BuyerLastTxid for company B: " + err.Error())
    }

    return sdk.Success(nil)
}
func main() {
    err := sandbox.Start(new(CarbonCoinContract))
    if err != nil {
        sdk.Instance.Errorf(err.Error())
    }
}
```

函数详解：

- **IssueCredit**：
  - **功能**：为指定公司发行碳币和碳额度。
  - **参数**：companyID, carbonCoin, carbonCredit。
  - **返回值**：操作成功或错误的响应。
  - **作用描述**：为注册的公司账户增加碳币和碳额度。
- **QueryCredit**：
  - **功能**：查询指定公司的碳币和碳额度。
  - **参数**：companyID。
  - **返回值**：包含碳币和碳额度信息的响应或错误信息。
  - **作用描述**：提供一个接口查询公司账户的当前碳币和碳额度状态。
- **TransferCredit**：
  - **功能**：在两个公司之间转移碳币和碳额度。
  - **参数**：companyAID, companyBID, carbonCoin, carbonCredit, SellerLastTxid, BuyerLastTxid。
  - **返回值**：操作成功或错误的响应。
  - **作用描述**：允许公司之间转移碳币和碳额度，实现碳信用的交易，并记

录交易后的最终状态。

3. 主键和存储

状态变量:

- **creditInfo**：存储每个公司的碳币和碳额度信息，数据结构包括碳币数量和碳额度。

4. 调用方法

外部调用接口： 通过合约的 **InvokeContract** 方法，外部可以调用 **IssueCredit**、**QueryCredit** 和 **TransferCredit** 方法，实现合约功能的操作。每个方法需要的参数和返回的结果详细说明如上。

5. 安全性和限制

已知风险:

- 操作错误可能导致不正确的碳币和碳额度记录。
- 系统安全性依赖于区块链平台的整体安全性。

使用限制:

- 只有合约管理员或授权用户可以发行和转移碳币和碳额度。
- 查询功能公开可供任何参与者使用。

合约四：CarbonReport

1. 合约概述

功能介绍： 存证合约用于记录和验证文件的存证信息。它通过记录文件的哈希值、文件名和时间戳，帮助用户验证文件的真实性和不变性。

重要性和应用场景： 在法律、医疗、科研等需要确保文件真实性和不变性的领域，存证合约提供了一个关键的技术支持。合约确保了文件信息的不可篡改和可追踪，对于提高文件管理的透明度和安全性具有重要价值。

2. 合约代码解析

```
package main

import (
    "encoding/json"
```

```
	"fmt"
	"log"
	"strconv"

	"chainmaker/pb/protogo"
	"chainmaker/sandbox"
	"chainmaker/sdk"
)

type FactContract struct {
}

// 存证对象
type Fact struct {
	FileHash string
	FileName string
	Time     int
}

// 新建存证对象
func NewFact(fileHash string, fileName string, time int) *Fact {
	fact := &Fact{
		FileHash: fileHash,
		FileName: fileName,
		Time:     time,
	}
	return fact
}

func (f *FactContract) InitContract() protogo.Response {
	return sdk.Success([]byte("Init contract success"))
}

func (f *FactContract) UpgradeContract() protogo.Response {
	return sdk.Success([]byte("Upgrade contract success"))
}

func (f *FactContract) InvokeContract(method string) protogo.Response {
	switch method {
	case "save":
		return f.Save()
	case "findByFileHash":
		return f.FindByFileHash()
	default:
```

```
            return sdk.Error("invalid method")
        }
}

func (f *FactContract) Save() protogo.Response {
        params := sdk.Instance.GetArgs()

        // 获取参数
        fileHash := string(params["file_hash"])
        fileName := string(params["file_name"])
        timeStr := string(params["time"])
        time, err := strconv.Atoi(timeStr)
        if err != nil {
            msg := "time is [" + timeStr + "] not int"
            sdk.Instance.Errorf(msg)
            return sdk.Error(msg)
        }

        // 构建结构体
        fact := NewFact(fileHash, fileName, time)

        // 序列化
        factBytes, err := json.Marshal(fact)
        if err != nil {
            return sdk.Error(fmt.Sprintf("marshal fact failed, err: %s", err))
        }
        // 发送事件
        sdk.Instance.EmitEvent("topic_vx", []string{fact.FileHash, fact.FileName})

        // 存储数据
        err = sdk.Instance.PutStateByte("fact_bytes", fact.FileHash, factBytes)
        if err != nil {
            return sdk.Error("fail to save fact bytes")
        }

        // 记录日志
        sdk.Instance.Infof("[save] fileHash=" + fact.FileHash)
        sdk.Instance.Infof("[save] fileName=" + fact.FileName)

        // 返回结果
        return sdk.Success([]byte(fact.FileName + fact.FileHash))

}
```

```
func (f *FactContract) FindByFileHash() protogo.Response {
    // 获取参数
    fileHash := string(sdk.Instance.GetArgs()["file_hash"])

    // 查询结果
    result, err := sdk.Instance.GetStateByte("fact_bytes", fileHash)
    if err != nil {
        return sdk.Error("failed to call get_state")
    }

    // 反序列化
    var fact Fact
    if err = json.Unmarshal(result, &fact); err != nil {
        return sdk.Error(fmt.Sprintf("unmarshal fact failed, err: %s", err))
    }

    // 记录日志
    sdk.Instance.Infof("[find_by_file_hash] fileHash=" + fact.FileHash)
    sdk.Instance.Infof("[find_by_file_hash] fileName=" + fact.FileName)

    // 返回结果
    return sdk.Success(result)
}

func main() {
    err := sandbox.Start(new(FactContract))
    if err != nil {
        log.Fatal(err)
    }
}
```

函数详解：

- **Save**：
  - **功能**：保存文件的存证信息。
  - **参数**：file_hash, file_name, time。
  - **返回值**：操作成功的消息或错误信息。
  - **作用描述**：接收文件哈希值、文件名和时间戳，记录存证信息到区块链上，确保其不可篡改性。
- **FindByFileHash**：
  - **功能**：通过文件哈希值查询存证信息。
  - **参数**：file_hash。
  - **返回值**：包含存证信息的响应或错误信息。
  - **作用描述**：提供一个接口来查询和检索之前保存的文件存证信息。

3. 主键和存储

状态变量:

- **fact_bytes**：使用文件哈希值作为键，存储对应的存证信息。

4. 调用方法

外部调用接口： 通过合约的 **InvokeContract** 方法，外部可以调用 **Save** 和 **FindByFileHash** 方法，实现合约功能的操作。每个方法需要的参数和返回的结果详细说明如上。

5. 安全性和限制

已知风险:

- 数据错误或遗漏可能导致存证信息不准确。
- 系统安全性依赖于区块链平台的整体安全性。

使用限制:

- 只有授权的用户或系统可以上传和查询存证数据。

合约五：Secret

1. 合约概述

功能介绍： Secret 合约设计用于管理加密数据的部分解密信息共享。它支持共享部分解密信息以及查询这些信息。

重要性和应用场景： 在需要保护信息安全且要求数据共享的环境中，如加密通信、数据存储安全等，此合约通过分布式的方式共享部分解密信息，增强了数据的安全性和可访问性。

2. 合约代码解析

```
package main

import (
    "encoding/json"
    "fmt"

    "chainmaker.org/chainmaker/contract-sdk-go/v2/pb/protogo"
    "chainmaker.org/chainmaker/contract-sdk-go/v2/sandbox"
```

```
    "chainmaker.org/chainmaker/contract-sdk-go/v2/sdk"
)

// NTRUContract 合约结构体
type NTRUContract struct {
}

// InitContract 安装合约时会执行此方法，必须
func (f *NTRUContract) InitContract() protogo.Response {
    return sdk.Success([]byte("Init contract success"))
}

// UpgradeContract 升级合约时会执行此方法，必须
func (f *NTRUContract) UpgradeContract() protogo.Response {
    return sdk.Success([]byte("Upgrade contract success"))
}

// InvokeContract the entry func of invoke contract func
func (f *NTRUContract) InvokeContract(method string) protogo.Response {
    switch method {
    case "share":
        return f.share()
    case "getshare":
        return f.getShare()
    default:
        return sdk.Error("invalid method")
    }
}

// share 共享部分解密
func (f *NTRUContract) share() protogo.Response {
    params := sdk.Instance.GetArgs()

    // 获取参数
    cipherHash := string(params["cipher_hash"])
    decryptShare := string(params["decrypt_share"])

    // 查询已有
    result, err := sdk.Instance.GetStateByte("cipher_hash", cipherHash)
    if err != nil {
        return sdk.Error(fmt.Sprintf("failed to call get state, %s", err))
    }

    var decryptShares = make(map[string]struct{})
```

```
    if len(result) != 0 {
        err = json.Unmarshal(result, &decryptShares)
        if err != nil {
            return sdk.Error(fmt.Sprintf("unmarshal decrypt shares failed, err: %s", err))
        }
    }

    decryptShares[decryptShare] = struct{}{}

    // 序列化
    decryptSharesBytes, err := json.Marshal(decryptShares)
    if err != nil {
        return sdk.Error(fmt.Sprintf("marshal decrypt shares failed, err: %s", err))
    }

    // 发送事件
    sdk.Instance.EmitEvent("cipher_hash", []string{cipherHash, decryptShare})

    // 存储数据
    err = sdk.Instance.PutStateByte("cipher_hash", cipherHash, decryptSharesBytes)
    if err != nil {
        return sdk.Error("fail to save decrypt shares bytes")
    }

    // 记录日志
    sdk.Instance.Infof("[save] cipherHash=" + cipherHash)
    sdk.Instance.Infof("[save] decryptShare=" + decryptShare)

    return sdk.Success([]byte("share success"))
}

// getShare 获取部分解密
func (f *NTRUContract) getShare() protogo.Response {
    params := sdk.Instance.GetArgs()

    // 获取参数
    cipherHash := string(params["cipher_hash"])

    // 查询已有
    result, err := sdk.Instance.GetStateByte("cipher_hash", cipherHash)
    if err != nil {
        return sdk.Error(fmt.Sprintf("failed to call get state, %v", err))
    }
```

```
    // 记录日志
    sdk.Instance.Infof("[save] cipherHash=" + cipherHash)
    sdk.Instance.Infof("[save] decryptShare=" + string(result))

    return sdk.Success(result)
}

func main() {
    err := sandbox.Start(new(NTRUContract))
    if err != nil {
        sdk.Instance.Errorf(err.Error())
    }
}
```

函数详解:

- **share**:
  - **功能**: 共享部分解密信息。
  - **参数**: cipher_hash, decrypt_share。
  - **返回值**: 操作成功的消息或错误信息。
  - **作用描述**: 接收加密哈希和对应的部分解密信息，存储到区块链上，保证信息的安全性和可追踪性。
- **getShare**:
  - **功能**: 获取部分解密信息。
  - **参数**: cipher_hash。
  - **返回值**: 包含解密信息的响应或错误信息。
  - **作用描述**: 提供一个接口来查询先前共享的部分解密信息。

3. 主键和存储

状态变量:

- **cipher_hash**: 使用加密哈希作为键，存储对应的部分解密信息集合。

4. 调用方法

外部调用接口: 通过合约的 **InvokeContract** 方法，外部可以调用 **share** 和 **getShare** 方法，实现合约功能的操作。每个方法需要的参数和返回的结果详细说明如上。

5. 安全性和限制

已知风险:

- 数据错误或遗漏可能导致部分解密信息不准确。
- 系统安全性依赖于区块链平台的整体安全性。

使用限制:

- 只有授权的用户或系统可以上传和查询部分解密数据。