区块链应用开发

简介

数字货币曾是区块链技术的唯一应用场景

对智能合约的支持突破了场景限制, 丰富了区块链应用的适用范围, 可以支持多行业、大规模的商业应用

区块链应用

区块链应用: 一般由若干部署在区块链网络中的智能合约, 以及调用这些智能合约的应用程序组成

用户专注于与业务本身相关的应用程序

智能合约则封装了与区块链账本直接交互的相关过程, 被应用程序调用

智能合约开发

智能合约本质上是为了对上层业务逻辑进行支持且直接与账本结构打交道, 处于核心位置.

所以设计得当可以简化上层应用开发的过程

应用程序开发

应用程序通过调用智能合约提供的方法接口实现业务逻辑, 可以使用JavaScript、Python、Go、Java等主流语言进行开发

链码的原理

链码延伸自智能合约的概念, 支持使用主流高级编程语言实现

区块链网络中的成员商定业务逻辑后, 可将业务逻辑编程到链码中, 所有人遵守合约执行

链码会创建一些状态（state）并写入账本中。状态带有绑定到链码的命名空间，仅限于创建他的链码使用，不能被其他链码直接访问。不过，在合适的范围内，一个链码也可以调用另一个链码，间接访问其状态

链码在Fabric节点上的隔离沙盒(目前为Docker容器)中运行, 并通过gRPC协议与节点进行交互

- 调用链码
- 读写账本
- 返回响应
- ……

Fabric中支持多种语言实现链码，包括Golang、JavaScript、Java等

基本工作原理

1. 首先用户通过客户端向Fabric的背书节点发出调用链码的交易提案
2. 节点对交易提案进行包括ACL权限检查在内的各种检验, 通过后则创建模拟执行这一交易的环境
3. 之后, 节点和链码容器之间通过gRPC消息来交互, 模拟执行交易并给出背书结论
4. 当链码的代码逻辑需要读写账本时,通过shim层发送相应操作类型给节点, 节点本地操作账本后返回响应消息
5. 客户端收到足够的背书节点的支持后, 便可以将这笔交易发送给排序节点进行排序, 并最终写入区块链

链码接口与结构

依赖包

链码实现需要引入如下依赖包

- "github.com/hyperledger/fabric/core/chaincode/shim"
  1. shim包提供了链码与账本交互的中间层
  2. 链码通过shim.ChaincodeStub提供的方法来读取和修改账本状态
- "github.com/hyperledger/fabric/protos/peer"
  - peer.Response: 响应信息

链码接口

每个链码都需要实现chaincode接口：

```
type Chaincode interface{
    Init(stub ChaincodeStubInterface) peer.Response
    Invoke(stub ChaincodeStubInterface) peer.Response
}
```

Init与Invoke方法

编写链码, 关键是实现Init与Invoke两个方法

Init方法在链码部署或升级时被调用, 完成初始化工作

对数据进行操作时, Invoke方法被调用, 因此响应调用或查询的业务逻辑都需要在此方法中实现

必要结构

一个链码的必要结构如下

```
package main

//引入必要的包
import(
    "fmt"
```

```
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

//声明一个结构体
type SimpleChaincode struct {

}

//为结构体添加Init方法
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response{
  //在该方法中实现链码初始化或升级时的处理逻辑
  //编写时可灵活使用stub中的API
}

//为结构体添加Invoke方法
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
peer.Response{
  //在该方法中实现链码运行中被调用或查询时的处理逻辑
  //编写时可灵活使用stub中的API
}

//主函数，需要调用shim.Start（ ）方法
func main() {
  err := shim.Start(new(SimpleChaincode))
  if err != nil {
     fmt.Printf("Error starting Simple chaincode: %s", err)
  }
}
```

链码API

账本状态交互API

链码需要将数据记录在分布式账本中.需要记录的数据称为状态, 以K-V对的形式存储

账本状态交互API可以对账本状态进行操作

GetState(key string) ([]byte, error) 通过Key来返回数组的特定值

PutState(key string, value []byte) error 账本中写入特定的键和值

DelState(key string) error 从账本中移除指定的键和值

GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error) 根据指定的范围内的健值

GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error) 返回指定健的所有历史值

GetQueryResult(query string) (StateQueryIteratorInterface, error) 对(支持富查询功能的)状态数据库进行富查询

交易信息相关API

GetTxID() string 返回交易提案中指定的交易ID

GetTxTimestamp() (*timestamp.Timestamp, error) 返回交易创建的时间戳，这个时间戳是peer收到交易的当前时间

GetBinding() ([]byte, error) 返回交易的binding信息

GetSignedProposal() (*pb.SignedProposal, error) 返回与交易提案相关的所有数据

GetCreator() ([]byte, error) 返回该交易的提交者的身份信息

GetTransient() (map[string][]byte, error) 返回交易中不会被写至账本中的一些临时信息

参数API

GetArgs() [][]byte 返回调用链码时交易提案中指定的参数

GetArgsSlice() ([]byte, error) 返回调用链码时交易提案中指定的参数

GetFunctionAndParameters() (function string, params []string) 返回调用链码时交易提案中指定的被调用的函数名称及其参数

GetStringArgs() []string 返回调用链码时指定的参数

-c '{"Args":["fn", "param1", "param2", "paramN"]}'

示例(HelloWorld)

Init方法

- 获取参数并判断参数长度是否为2
  - 参数: Key, Value
- 调用PutState方法将状态写入账本中
- 如果有错误, 则返回
- 打印输出提示信息
- 返回成功

Invoke方法

- 获取参数并判断长度是否为1
- 利用第1个参数获取对应状态GetState(key)
- 如果有错误则返回
- 如果返回值为空则返回错误
- 返回成功状态


区块链部落
专注于区块链技术
识别图中二维码关注我们