用正确的姿势开发 Hyperledger Fabric 系列

【智能合约&账本存储】

功夫小猫

tanzhiguo@cn.ibm.com

计划要写一系列关于 Hyperledger Fabric 开发相关的技术文章，基于版本 Fabric v1.2.0，面向有一定区块链开发基础的编程人员，侧重标准和规范方面，如果您有任何疑问、建议，欢迎通过公众号下方回复！这是系列的第二篇，关于智能合约以及账本存储。

0x00 准备工作

在 Fabric 中，通常把 chaincode 分为系统 chaincode 和用户 chaincode，我们可以把用户的 chaincode 称为智能合约，Fabric 官方 example02 的例子就是一个很有代表性的智能合约，A 和 B 两个实体转移资产的场景，合约中有三个操作，分别是：资产分配、资产转移、当前资产查询，当然这三个操作是比较基本的，更进一步能想到的潜在需求，

- **历史记录问题：**查看 A 实体或者 B 实体的资产历史转移记录
- **按条件查询记录问题：**查看某个时间段，比如 2018 年 7 月内，A 实体和 B 实体资产转移记录

带着以上问题，我们看看智能合约以及智能合约在 Fabric 中的存储。

0x01 智能合约单元测试

在 chaincode/shim 下的账本操作中，大部分 API 都可以通过 shim.MockStub 进行模拟测试而无需启动网络环境，比如 State 的操作，包括 init、invoke、GetStateByPartialCompositeKey 等，某些涉及 history 的 API 目前还没有现实可以模拟测试，比如 GetHistoryForKey，具体单元测试方法可以参考 e2e_cli 下 example02 的例子，比如对于 State，

```
34  func checkState(t *testing.T, stub *shim.MockStub, name string, value string) {
35      bytes := stub.State[name]
36      if bytes == nil {
37          fmt.Println("State", name, "failed to get value")
38          t.FailNow()
39      }
40      if string(bytes) != value {
41          fmt.Println("State value", name, "was not", value, "as expected")
42          t.FailNow()
43      }
44  }
```

0x02 智能合约部署

合约的部署，可以通过 cli 方式部署，也可以通过 SDK 方式部署，按照上一篇文章中，关于 Node.js SDK 里面讲到的，代码类似如下方式，

```
const installChaincode = async function(peer, chaincodeId, chaincodePath, org) {
  try {
    const client = await initClient(org)
    const request = {
      targets: [peer],
      chaincodePath: chaincodePath,
      chaincodeId: chaincodeId,
      chaincodeVersion: 'v1',
      chaincodeType: 'golang'
    // channelNames:['mychannel']
    }
    const installResult = await client.installChaincode(request, 10000)
    return installResult
  } catch (error) {
    return error.toString()
  }
}
```

假如智能合约中使用了外部依赖，那么可以将依赖包通过 vender 的方式放置到平级目录下，系统在打包时会收集当前智能合约下的第三方依赖库。

智能合约一般是部署到某个 peer 上，也可以指定具体的 channel「也就是账本」，而接下来实例化则需要由 channel 来完成，只要部署了这个智能合约的 peer 都可以进行实例化操作，而且只需一次，关于这方面的 workflow，后面会有专门的文章说明。

0x03 更多逻辑的实现

文章开始提到的第一个问题，从 API 中可以直接找到答案，GetHistoryForKey 的作用就是提供对于某个 Key 的历史记录查询；对于第二个问题，可以通过 CompositeKey 的方式来处理，这是一种前端匹配的方式，代码实现也比较简单，首先设计类似这样的一个结构体，

```
type Tx struct {
    Id        string `json:"id"`
    Lender    string `json:"lender"`
    Message   string `json:"message"`
    Amount    int    `json:"amount"`
    Timestamp int64  `json:"timestamp"`
}
```

再设计一个组合 Key 方法，

```go
func (t *SimpleChaincode) composite(stub shim.ChaincodeStubInterface, attributes []string, lender string, message string, amount int) pb.Response {

    txKey, err := stub.CreateCompositeKey(searchIndex, attributes)
    if err != nil {
        return shim.Error(err.Error())
    }

    var id string
    id = genUlid()
    var tx = Tx{Id: id, Lender: lender, Message: message, Amount: amount, Timestamp: nanos}

    txAsBytes, _ := json.Marshal(tx)
    stub.PutState(txKey, txAsBytes)

    return shim.Success(nil)
}
```

这个方法在 invoke 时调用「attributes 用字符串数组方式记录 invoke 的年、月、日、具体交易数额以及交易方向」，

```go
    attributes := []string{strconv.Itoa(now.Year()), strconv.Itoa(int(now.Month())), strconv.Itoa(now.Day()), strconv.Itoa(X)}
    message := "Transfer " + A + " to " + B + " " + strconv.Itoa(X)

    return t.composite(stub, attributes, A, message, X)
```

然后，通过

GetStateByPartialCompositeKey(searchIndex, []string{"2018","7"})

方法就可以匹配出条件为 2018 年 7 月份的所有交易记录，再根据 Lender 判断交易的方向「是从 A 到 B，还是从 B 到 A」，当然，也可以把交易方向也做为查询的属性之一，这种设计相对来说缺乏了一些灵活性「如果同时查询 A 发出的交易和 B 发出的交易需要两次调用 API」。

0x04 账本的存储结构

根据上面的 API 方法，我们可以大致看到，Fabric 数据的存储，分为三种形式，账本数据库、历史数据库、状态数据库，其中账本数据库采用文件存储方式，区块链的不可篡改特性往往就是指这个账本数据库，LevelDB 充当了账本数据库索引的角色，可以想象当复杂的查询时，存在性能的问题；状态数据库，顾名思义，就是只记录当前状态「比如上面的智能合约中，A 和 B 交易之后的值」，也可以说是 invoke 操作的快照；而历史数据库，可以认为是为了查询需要而设计的一种类似缓存的做法。

目前，Fabric 通过内置的 LevelDB 或者 couchDB 来存储状态，我们可以通过 couchDB 更直观的看到交易中存储的变化，

上面的三个结构分别是在创建 channel、部署智能合约、实例化智能合约三个操作之后产生的。

	all~tx20187291	all~tx20187291	{ "rev": "3-bdde31ae5b60ef966...
	all~tx20187299	all~tx20187299	{ "rev": "1-11b1a04ed5b750d5...
	a	a	{ "rev": "5-b8d36c47452d8410...
	b	b	{ "rev": "5-9dabf89bad4155af1...

上面是 ID 为 mycc 的智能合约中，执行了两次 invoke 后产生的变化，前两条记录就是上面复合 Key 产生的记录，表示 2018 年 7 月 29 日，A 到 B 交易 1，以及 2018 年 7 月 29 日，B 到 A 交易 9，比如这条记录，

id "all~tx20187299"


```
{
 "id": "\u0000all~tx\u00002018\u00007\u000029\u00009\u0000",
 "key": "\u0000all~tx\u00002018\u00007\u000029\u00009\u0000",
 "value": {
  "rev": "1-11b1a04ed5b750d57063b8cba8ba4f18"
 },
 "doc": {
  "_id": "\u0000all~tx\u00002018\u00007\u000029\u00009\u0000",
  "_rev": "1-11b1a04ed5b750d57063b8cba8ba4f18",
  "amount": 9,
  "id": "01CKHWP81KFPE6XC2S43FPKQFR",
  "lender": "b",
  "message": "Transfer b to a 9",
  "timestamp": 1532830097436592400,
  "~version": "5:0"
 }
}
```


0x05 账本设计的思考

关于上面描述的状态数据库和历史数据库的设计，状态数据库是为了高效的执行智能合约的需要，历史数据库，很大程度上是为了来自查询的需要。

我们知道，对于账本的查询，并不会进行节点之间的共识，那么很容易产生一个疑问，如果是因为数据的查询需要，而把数据结构设计得更复杂，是否违背设计原则？

另外一个值得思考的问题：因为 Fabric 在整个区块链的商业环节中，还存在着应用层，那么应用层的数据存储和 Blockchain 的存储要怎么权衡轻重？既然 Blockchain 底层的历史数据库是一种类似缓存的计方案，那么为什么把这种存储设计到底层中，而不是留给应用层面去处理？

采用 Fabric 开发的实际的项目，开发者往往会掉到坑里，比如 couchDB 支持富查询，那么有的项目干脆把翻页的功能也都设计在智能合约层面，应用层面就不需要数据存储了，把 Blockchain 当成了传统数据库使用，这是一种错误的做法。
假想，来自 Blockchain 底层数据存储只有账本数据库，负责保存原始的「区块+链」数据结构和数据，SDK 层设计一种同步策略，根据应用层的业务需要，按照策略「比如网络 Idle 的时候」同步某些账本信息「比如某些符合需要的 History」到应用层数据库，这样会大大减轻账本查询的负担。

这种设计还有另外一个优势，假设联盟链有 10 个参与方，每个参与方都可以根据自己的需要来设计应用层存储，以上面的智能合约为例，10 个参与方中，如果只有一方有需求，要按照时间段查询，那么很显然，这种情况下如果把这种条件查询写到智能合约里，对另外 9 个参与方来说是一种浪费，如果在 SDK 层面设置账本同步策略到应用层存储，即满足的不同方的不同需求，又肯定了应用层在整个 Blockchain 商业环节中的意义。

0x06 小结

智能合约要做的事情，要偏重交易的产生，对于交易的查询应该设计出更灵活的接口给应用层面去处理，智能合约以及账本，都是一种崭新的应用，类似 LevelDB 这种数据库在未来的区块链存储中可能会逐渐暴露出不足，随着技术的发展，也许未来会诞生一种针对「区块+链」的全新存储形式。

这是一系列关于 Hyperledger Fabric 开发相关的技术文章，欢迎您在公众号下方反馈。