区块链与数字货币 Lab 3

姓名：王昊元

学号：3220105114

1. 实验目的：

分析Fabric v2.x 源代码中的相关数据结构，说明Fabric区块链中的如下数据的组成要素一句这些数据的相互间关系：

- 1. 区块
- 2. 交易
- 3. 世界状态
- 4. 日志

2. 实验步骤：

2.1 Fabric v2.x 代码源

本实验中使用的代码仓库为：`https://github.com/hyperledger/fabric/tree/release-2.0`，其版本为`Fabric v2.0`。

（注意到在数据结构中都相似地存在如下字段，这里单独列出以说明其作用）：

- 1. `struct{} XXX_NoUnkeyedLiteral`：空结构体，避免某些编译器警告或错误
- 2. `[]byte XXX_unrecognized`：存储未知的字段数据
- 3. `int32 XXX_sizecache`：用于缓存结构体的大小信息

2.2 数据结构分析：区块

在fabric-release-2.0\vendor\github.com\hyperledger\fabric-protos-go\common\common.pb.go中，可找到对区块Block以及相关其他数据结构的定义：

```
// 1. Block：区块的数据结构
type Block struct {
        Header               *BlockHeader
`protobuf:"bytes,1,opt,name=header,proto3" json:"header,omitempty"`
        Data                 *BlockData
`protobuf:"bytes,2,opt,name=data,proto3" json:"data,omitempty"`
        Metadata             *BlockMetadata
`protobuf:"bytes,3,opt,name=metadata,proto3" json:"metadata,omitempty"`
        XXX_NoUnkeyedLiteral struct{}       `json:"-"`
        XXX_unrecognized     []byte         `json:"-"`
        XXX_sizecache        int32          `json:"-"`
}

// 2. BlockHeader：区块头的数据结构
type BlockHeader struct {
        Number               uint64   `protobuf:"varint,1,opt,name=number,proto3"
json:"number,omitempty"`
        PreviousHash         []byte
`protobuf:"bytes,2,opt,name=previous_hash,json=previousHash,proto3"
json:"previous_hash,omitempty"`
        DataHash             []byte
`protobuf:"bytes,3,opt,name=data_hash,json=dataHash,proto3"
json:"data_hash,omitempty"`
        XXX_NoUnkeyedLiteral struct{} `json:"-"`
        XXX_unrecognized     []byte   `json:"-"`
        XXX_sizecache        int32    `json:"-"`
}

// 3. BlockData：区块数据的数据结构
type BlockData struct {
        Data                 [][]byte `protobuf:"bytes,1,rep,name=data,proto3"
json:"data,omitempty"`
        XXX_NoUnkeyedLiteral struct{} `json:"-"`
        XXX_unrecognized     []byte   `json:"-"`
        XXX_sizecache        int32    `json:"-"`
}

// 4. BlockMetadata：区块元数据的数据结构
type BlockMetadata struct {
        Metadata             [][]byte `protobuf:"bytes,1,rep,name=metadata,proto3"
json:"metadata,omitempty"`
        XXX_NoUnkeyedLiteral struct{} `json:"-"`
        XXX_unrecognized     []byte   `json:"-"`
        XXX_sizecache        int32    `json:"-"`
}
```

可见，Block的组成要素如下：

Block

- 1. `BlockHeader* header`：区块头
  - 1. `uint64 number`：区块编号
  - 2. `[]byte PreviousHash`：上一个区块的哈希值
  - 3. `[]byte DataHash`：区块数据哈希值
- 2. `BlockData* data`：区块数据
  - 1. `[][]byte data`：区块数据
- 3. `BlockMetadata* metadata`：区块元数据
  - 1. `[][]byte metadata`：区块元数据

2.3 数据结构分析：交易

在`fabric-release-2.0\vendor\github.com\hyperledger\fabric-protos-go\peer\transaction.pb.go`中，可找到对交易`Transaction`以及相关其他数据结构的定义：

```
//1. Transaction：交易的数据结构
type Transaction struct {
        // The payload is an array of TransactionAction. An array is necessary to
        // accommodate multiple actions per transaction
        Actions              []*TransactionAction
`protobuf:"bytes,1,rep,name=actions,proto3" json:"actions,omitempty"`
        XXX_NoUnkeyedLiteral struct{}             `json:"-"`
        XXX_unrecognized     []byte               `json:"-"`
        XXX_sizecache        int32                `json:"-"`
}

//2. TransactionAction：交易动作的数据结构
type TransactionAction struct {
        // The header of the proposal action, which is the proposal header
        Header []byte `protobuf:"bytes,1,opt,name=header,proto3"
json:"header,omitempty"`
        // The payload of the action as defined by the type in the header For
        // chaincode, it's the bytes of ChaincodeActionPayload
        Payload              []byte   `protobuf:"bytes,2,opt,name=payload,proto3"
json:"payload,omitempty"`
        XXX_NoUnkeyedLiteral struct{} `json:"-"`
        XXX_unrecognized     []byte   `json:"-"`
        XXX_sizecache        int32    `json:"-"`
}
```

可见，Transaction的组成要素如下：

Transaction

- 1. `[]*TransactionAction Actions`：交易动作列表，每个指针对应交易中的一个动作
  - 1. `[]byte header`：动作的头部信息，通常包含动作的元数据，例如动作类型、签名等
  - 2. `[]byte Payload`：动作的负载信息。根据头部的类型，负载可能包含不同的数据。

2.4 数据结构分析：世界状态

在Fabric中，世界状态由数据库存储。Fabric支持couchdb和leveldb两种数据库。这里以leveldb为例。

2.4.1 基本定义

Fabric将数据库作为“实体”总体进行管理，其中有关`lebeldb`的信息位于`fabric-release-2.0\common\ledger\util\leveldbhelper`中。

整体数据库的定义可分为**三层**架构，其中：

- `leveldb_helper.go`：定义了数据库本身以及其基本操作，包括`Get`、`Put`、`Delete`等。
- `leveldb_provider.go`：定义了数据库的对传入数据操作，如批处理、数据序列化等操作。
- 上层方法：直接引用`Provider`实例，避免直接对数据库进行操作。

其数据库结构如下所示：

```
// 1. DB: 整个数据库对象
type DB struct {
        conf    *Conf
        db      *leveldb.DB
        dbState dbState
        mutex   sync.RWMutex

        readOpts        *opt.ReadOptions
        writeOptsNoSync *opt.WriteOptions
        writeOptsSync   *opt.WriteOptions
}

// 2. Conf: 数据库配置信息
type Conf struct {
        DBPath                  string
```

```
        ExpectedFormatVersion string
}

// 3. Provider: 数据库管理器
type Provider struct {
        db *DB

        mux       sync.Mutex
        dbHandles map[string]*DBHandle
}

// 4. DBHandle: 数据库句柄
type DBHandle struct {
        dbName string
        db     *DB
}
```

可见：Provider内置结构如下所示：

Provider

- 1. *DB db：数据库对象
  - 1. Conf conf：数据库配置信息
    - 1. string DBPath：数据库路径
    - 2. string ExpectedFormatVersion：数据库格式版本
  - 2. *leveldb.DB db：数据库实例
  - 3. dbState dbState：数据库状态
  - 4. sync.RWMutex mutex：读写锁
  - 5. *opt.ReadOptions readOpts：读选项
  - 6. *opt.WriteOptions writeOptsNoSync：写选项（不同步）
  - 7. *opt.WriteOptions writeOptsSync：写选项（同步）
- 2. sync.Mutex mux：互斥锁
- 3. map[string]* DBHandle dbHandles：数据库句柄，一边对多个数据库实例进行管理
  - 1. string dbName：数据库名称
  - 2. *DB db：数据库实例

2.4.2 接口方法

接口方法的概述与定义位于fabric-release-2.0\core\ledger\kvledger\txmgmt\statedb中。

在接口方法定义中，Fabric采用类似“父类/继承类”的方法，将共同的接口定义在statedb.go中，具体的实现分别在stateleveldb.go与statecouchdb.go中。

具体接口方法定义如下：

```
type VersionedDB interface {
        // 1. 获取状态
        GetState(namespace string, key string) (*VersionedValue, error)
        // 2. 获取多个状态
        GetVersion(namespace string, key string) (*version.Height, error)
        // 3. 获取多个键的状态
        GetStateMultipleKeys(namespace string, keys []string) ([]*VersionedValue,
error)
        // 4. 获取键范围内的迭代器
        GetStateRangeScanIterator(namespace string, startKey string, endKey string)
(ResultsIterator, error)
        // 5. 获取键范围内的迭代器（带元数据）
        GetStateRangeScanIteratorWithMetadata(namespace string, startKey string,
endKey string, metadata map[string]interface{}) (QueryResultsIterator, error)
        // 6. 执行查询
        ExecuteQuery(namespace, query string) (ResultsIterator, error)
        // 7. 执行查询（带元数据）
        ExecuteQueryWithMetadata(namespace, query string, metadata
map[string]interface{}) (QueryResultsIterator, error)
        // 8. 执行更新
        ApplyUpdates(batch *UpdateBatch, height *version.Height) error
        // 9. 获取最新保存点
        GetLatestSavePoint() (*version.Height, error)
        // 10. 验证键值对
        ValidateKeyValue(key string, value []byte) error
        // 11. 验证键范围
        BytesKeySupported() bool
        // 12. 打开数据库
        Open() error
        // 13. 关闭数据库
        Close()
}
```

（具体每个接口的方法在注释中有简单描述）

2.5 数据结构分析：日志

日志的相关定义大部分位于：`fabric-release-2.0\common\flogging`中。

Fabric使用的是`zap`作为日志库，其日志分为8个等级，具体结构如下：

```
// 1. Logging: 日志管理器
type Logging struct {
        *LoggerLevels
```

```
        mutex           sync.RWMutex
        encoding        Encoding
        encoderConfig   zapcore.EncoderConfig
        multiFormatter *fabenc.MultiFormatter
        writer          zapcore.WriteSyncer
        observer        Observer
}

// 2. LoggerLevels：日志级别
type LoggerLevels struct {
        mutex        sync.RWMutex
        levelCache   map[string]zapcore.Level
        specs        map[string]zapcore.Level
        defaultLevel zapcore.Level
        minLevel     zapcore.Level
}

// 3. Level：日志级别
type Level int8

const (
        DebugLevel Level = iota - 1
        InfoLevel
        WarnLevel
        ErrorLevel
        DPanicLevel
        PanicLevel
        FatalLevel

        _minLevel = DebugLevel
        _maxLevel = FatalLevel
)
```

可见，日志的组成要素如下：

Logging

- 1. *LoggerLevels：日志级别
  - 1. sync.RWMutex mutex：读写锁
  - 2. map[string]zapcore.Level levelCache：日志级别缓存
  - 3. map[string]zapcore.Level specs：日志级别配置
  - 4. zapcore.Level defaultLevel：默认日志级别
  - 5. zapcore.Level minLevel：最小日志级别
- 2. sync.RWMutex mutex：读写锁
- 3. Encoding encoding：日志编码格式
- 4. zapcore.EncoderConfig encoderConfig：日志编码配置
- 5. fabenc.MultiFormatter multiFormatter：日志格式化器
- 6. zapcore.WriteSyncer writer：日志输出器

- 7. `Observer observer`：日志观察者

2.6 相互关系分析

1. 区块与交易

- 区块包含交易：每个区块中包含一个或多个交易。区块通过Merkle树的形式管理交易条目。
- 交易验证：在区块被添加到区块链之前，区块中的所有交易都需要经过验证。验证过程包括签名验证、交易有效性验证、交易顺序验证等。
- 交易顺序：区块中的交易按照一定的顺序排列，这个顺序决定了交易的执行顺序。

2. 区块与世界状态

- 区块提交后的状态记录：区块提交后，其状态信息会被记录在世界状态中。
- 状态回滚：如果区块中的某个交易执行失败，整个区块可能会被回滚，世界状态也会恢复到之前的状态。

3. 交易与世界状态

- 交易执行与世界状态变更：交易执行后，会更新世界状态。世界状态是一个键值存储，记录了所有链码的状态信息。
- 交易对世界状态进行查询/更新：交易可以通过查询世界状态获取当前状态信息，并在执行过程中更新状态。

4. 日志与区块、交易

- 日志记录：日志系统记录了区块链的所有操作，包括区块的创建、交易的执行、世界状态的更新等。
- 通过日志的追踪：通过日志的追踪，可以追踪到区块链的运行轨迹，便于问题的排查和分析。
- 日志的多层级别：日志系统支持不同的日志级别，可以根据需要调整日志的详细程度。

5. 世界状态与日志

- 状态变化的日志记录：世界状态的每次更新都会被记录在日志中，便于后续的查询和分析。
- 通过日志回滚状态：在系统故障或崩溃后，可以通过日志恢复世界状态到故障前的状态。