交易Trace步骤设计说明

一、步骤格式

由于我们trace的目的是跟踪交易的处理过程，所以将步骤(Step)设计为如下形式：

TxHash.Stage.Level

TxHash:交易的hash或打包在一个区块的所有交易的hash的集合(以逗号分隔)

Stage：阶段，分为三个阶段，从rpc收到交易开始到加入mempool中为第一阶段，Stage为1；从Consensus从mempool取到交易开始到WriteBlock之前为第二阶段，Stage为2；从Consensus的WriteBlock开始到交易从mempool中删除为第三阶段，Stage为3。

Level：函数调用深度，从1开始，子调用在后面再加一层，同级后面的调用数字依次累加。

二、设计实现

数据结构设计：

1.步骤信息：

**type** StackItem **struct** {  
 maxChildNum int //同级步骤数  
 step string //步骤信息  
 module string //模块信息  
}

2.跟踪信息：

**type** TraceInfo **struct** {  
 timestamp int64 //时间戳  
 id string //消息Id或TxHash  
 ty int //1:message,2:transaction  
 module string //所属模块  
 state int //状态  
 level int //打印级别  
 function string //函数名  
 maxChildNum int //同级步骤数  
 step string //步骤信息  
 data **interface**{} //数据  
}

3.全局存储结构：用于模块间消息传递时保存消息id与交易哈希以及Step信息，在接收模块收到消息后清除。

**var** mutexId2Hashs sync.Mutex  
**var** mutexId2Step sync.Mutex  
//消息Id与交易hash或交易hash集合的映射  
**var** globalMsgId2Hashs = make(**map**[int64]string)  
//消息Id与当前步骤信息的映射  
**var** globalMsgId2Step = make(**map**[int64]\*StackItem)

1.Msgid—>hashs：消息Id与交易hash或交易hash集合的映射

2.Msgid--->StackItem：消息Id与当前步骤信息的映射

4.模块存储结构：

**type** ModuleStack **struct** {  
 mutexId sync.Mutex  
 //收到消息时临时保存下来  
 msgId2Hash **map**[int64]string  
  
 mutexStack sync.RWMutex  
 //保存交易hash，交易hashs以及区块hash的step stack  
 stack **map**[string]\*TxStack  
  
 mutexState sync.Mutex  
 //保存第一阶段结束时(mempool模块)交易的step；保存第二阶段开始时(consensus模块)pull到交易的初始step  
 //如果在本模块没有查询到hash对应的step，就依次在consensus模块和mempool模块的该字段找  
 state **map**[string]\*StackItem  
}

1.Msgid--->hashs：用于通过id查找到交易hash进而查到当前处理的step。模块收到消息时CreateTraceMsg接口state为MsgReceived时在对应的模块中存储，End()和Error()接口处理中清除。

2.hash---->Stack：保存交易哈希处理过程的step或区块hash处理过程的step。Begin()接口加入，End()和Error()接口弹出。当hash对应的stack为空时，清除该hash记录。

3.hash---->StackItem:会保存交易哈希停留的step，包括

1.mempool模块：在mempool的txCache的Push()函数中PushBack成功后保存txhash对应的step，在mempool的txCache的Remove()函数中Remove成功后清除txhash对应的记录，没有进入consensus的pull阶段的以后的并行处理(非主线的，没有father的处理)都以该step为初始。

2.consensus模块：在Consensus模块RequestTx()函数中把从mempool请求到的交易初始化step为hash.2(阶段2)；以下情况清除此记录：

a.本阶段交易查重后查出是重复的交易

b.进入WriteBlock阶段后

函数说明：

1. **func** CreateTraceMsg(id int64, msgTy int64, txHash string, state State, module, function string) \*TraceInfo

涉及消息的处理时，创建TraceInfo对象。

2. **func** CreateTraceTx(txHashs string, state State, module, function string, blockHash []byte) \*TraceInfo

非消息处理时，创建TraceInfo对象。

3. **func** (info \*TraceInfo) Begin()

打印开始信息。

4. **func** (info \*TraceInfo) UpdateState(state State)

打印状态更新信息。

5. **func** (info \*TraceInfo) End()

打印正常结束信息。

6. **func** (info \*TraceInfo) Error(err error)

打印非正常结束错误信息。

7. **func** FindFront(key string, module string) \*StackItem

查找当前hash所处的步骤，创建返回子步骤。

FindFront首先在本模块中查找hash对应的步骤信息，如果没有找到，再在阶段二中查找，如果仍然没有找到，再在阶段一中查找，如果都没找到则有错误。

注：

所有交易由RPC模块的SendTransactiong开始，CreateTraceTx函数分别针对TxAccepted和TxWriteBlock状态初始化Step的Stage为1和3，针对ProcAddBlockMsg函数除了本地处理AddBlockDetail消息调用外，还会由广播消息，同步消息调用，所以增加TxSyncBlock和TxBroadcastAddBlock两个状态，并针对这两个状态初始化step为4和5。

由于第三阶段几乎都是针对区块的处理，只有Execs模块和Mempool模块在收到block模块发来的EventAddBlock，EventDelBlock消息以及Execs模块收到EventExecTxList消息时才对block中的具体交易进行处理，所以在Begin()函数中对上述三个消息做了特殊处理，把交易hash集合中包含的每个交易hash对应的步骤存入对应模块的hash对应的处理过程”堆栈”中，在对应的End()函数中再做反向操作进行清除。

关于回滚时的步骤在此做一下说明，回滚操作是由blockchain模块处理的，具体是在connectBestChain函数中处理的，正常流程进入connectBlock，异常流程进入重组链过程(reorganizeChain)就是所谓的回滚，重组流程由disconnectBlock和connectBlock两步组成，所以回滚流程其实是与正常流程同级的，函数调用上是连续的，所以没有定义新的stage。