Geth-query获取数据

> 本文承接Geth-query设计文档，主要阐述通过Geth-query功能获得哪些数据，以及怎么获得，需要修改哪些代码。当然Geth-query是一个自定义的合约重放工具，通过修改`go-ethereum`的源码以及该工具的代码能够自定义获取任何在EVM中产生过的中间数据。

重发智能合约主要要使用`achieve`模式全节点的2种数据：1. 区块数据。2.区块的世界状态数据。前者通过`achieve`模式全节点通过p2p网络同步获取，后者通过全节点同步过程中通过EVM执行区块数据获取每一个区块的世界状态，然后将两者都通过RLP编码格式进行压缩，存储到底层的levelDB数据库中。

在重放过程中，获取上述两种数据可以通过`go-ethereum`的接口获取。

获取区块数据`block := chain.GetBlockByNumber(currentBlock)`。

获取区块世界状态数据`statedb, err := blockchain.StateAt(blockchain.GetBlockByHash(block.ParentHash()).Root())`

执行每笔交易的EVM api接口`_, gas, failed, err := core.ApplyMessage(vmenv, msg, gp)`

block数据

对于区块数据而言，可以直接通过`achieve`模式的全节点同步区块数据，然后按照区块数据存入blocks的表中。

Key	evmType	pgType
blockNumber	uint64	bigint
blockHash	a hex string	text (32 byte Keccak256)
parentHash	a hex string	text (32 byte Keccak256)
nonce	uint64	bigint
miner	a hex string	text (Address length 20)
difficulty	bigint_to_string	bigint
totalDifficulty	bigint_to string	bigint
extraData	byte[]	text
size	float64->string	text
gasLimit	uint64	bigint
gasUsed	uint64	bigint
_timestamp	uint64	bigint
transactionCount	uint64	int

transfers数据

对于transfers数据可以理解为由交易和内部交易导致账户余额世界状态的变化，通过深入修改EVM的源码代码，通过分析EVM在执行合约过程中哪里修改了账户地址的世界状态，然后在该处添加`capturetransfers`动作，抓取出每次EVM执行合约过程中账户世界状态变动的transfers数据。下面EVM中的动作导致了账户世界状态的变化：

1. Call 指令中如果`msg.value`的值不为零，并且校验`!evm.Context.CanTransfer`校验通过，会进行交易转账或者内部交易转账`evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)`，因此需要在该处添加`capturetransfer`动作。

   `evm.go 240`

   ```golang

   if evm.Interpreter().(*EVMInterpreter).GetConfig().Debug {
       evm.Interpreter().(*EVMInterpreter).GetConfig().Tracer.
   (*StructLogger).CaptureTransferState(evm.StateDB.(*state.StateDB), evm.depth, common.Hash{},
   caller.Address(),
           to.Address(), value, "CALL", evm.StateDB.(*state.StateDB).GetNextRevisionId())
   }
   ```

2. Create 指令中`msg.value`值不为零，并且校验`!evm.Context.CanTransfer`校验通过，会导致创建合约过程中给合约账户转账`evm.Transfer(evm.StateDB, caller.Address(), address, value)`。

> 这里可能存在一种情况，就是合约hash已经创建成功，但是由于gas费用不足以将合约的code存储进入世界状态会产生`ErrCodeStoreOutOfGas`错误，但是此时合约的地址已经存在世界状态中，并且转给合约账户的转账操作也已经生效。`evm.go 470`

`evm.go  420`插入`capturetransfer`动作

```
    if evm.Interpreter().(*EVMInterpreter).GetConfig().Debug {
        evm.Interpreter().(*EVMInterpreter).GetConfig().Tracer.
(*StructLogger).CaptureTransferState(evm.StateDB.(*state.StateDB), evm.depth,
common.Hash{}, caller.Address(),
            address, value, "CREATE", evm.StateDB.
(*state.StateDB).GetNextRevisionId())
    }
```

3. selfdestruct指令返回销毁账户的余额。

`instructions.go  881`

```
    if interpreter.GetConfig().Debug {
        interpreter.GetConfig().Tracer.
(*StructLogger).CaptureTransferState(interpreter.evm.StateDB.(*state.StateDB),
interpreter.evm.depth, common.Hash{}, contract.Address(),
            common.BigToAddress(stack.pop()), balance, "selfDestruct",
interpreter.evm.StateDB.(*state.StateDB).GetNextRevisionId())
    }
```

4. 区块中每笔交易产生的gas费用会直接在EVM中进行加减，因此在该处应该加入`capturetransfers`动作抓取`transfer`。注意这里的gas费用值为用户消耗的gas费用值。

> 这里计算gas费用也有一些规则：用户首先预支initialGas的上限值，然后在执行合约过程中会逐步消耗gas，但是对于用户的一些操作eth也会奖励用户gas，比如释放storage空间，selfdestruct合约(该操作会退回两笔费用，一笔是gas费用，一笔是合约余额)。最后用户支付的gas费用就是`st.initialGas - st.gas`。为了防止用户最后支付的gas费用为负值的情况，用户获得奖励的gas小于或等于用户消耗gas数目的一半。因此在销毁合约获得的奖励可能小于理论值，和用户本次交易消耗的gas有关。

`capturetansfer`动作可以添加在`state_transition.go 232行`，具体代码如　下：

```
    if st.evm.Interpreter().(*vm.EVMInterpreter).GetConfig().Debug {
        st.evm.Interpreter().(*vm.EVMInterpreter).GetConfig().Tracer.
(*vm.StructLogger).CaptureTransferState(st.evm.StateDB.(*state.StateDB), 0,
common.Hash{}, msg.From(),
            st.evm.Coinbase,
new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.gasPrice), "GASFEE",
0)
    }
```

5. 叔叔区块奖励和父区块的奖励也是直接在EVM中进行运算，因此也需要加入`capturetransfer`动作，该部分在geth-query重放工具中完成。

```
    for _, uncle := range block.Uncles() {
        r.Add(uncle.Number, big8)
        r.Sub(r, block.Header().Number)
        r.Mul(r, blockReward)
        r.Div(r, big8)
        data.transfers = append(data.transfers,
            newRewardTransfer(statedb, 0, common.Hash{}, common.Address{},
uncle.Coinbase, r, "UNCLE"))
        rd := new(big.Int)
        rd.Div(blockReward, big32)
        reward.Add(reward, rd)
    }

    data.transfers = append(data.transfers,
        newRewardTransfer(statedb, 0, common.Hash{}, common.Address{},
block.Coinbase(), reward, "MINED"))
```

`transfers`表的各项数据如下：

key	evmType	pgType
id	uint64	bigint
blockNumber	uint64	bigint
blockHash	a hex string	text
_timestamp	uint64	bigint
transactionHash	a hex string	text
transferIndex	int	bigint
depth	int	bigint
_from	a hex string	text
_to	a hex string	text
fromBalance	string	text
toBalance	string	text
transferValue	string	bigint
transferType	string	text
decollator	"" -> string	text

transactions数据

transactions数据主要是交易执行完后产生的receipt数据。通过组合`get-query`重放合约后的返回值和同步的`transactions`数据一起组合形成`receipt`。

主要的格式如下：

key	evmType	pgType
id	uint64	text (32 byte Keccak256)
blockNumber	uint64	bigint
blockHash	a hex string	text (32 byte Keccak256)
_timestamp	uint64	bigint
transactionHash	a hex string	text (32 byte Keccak256)
_from	a hex string	text
_to	a hex string	text
gas	uint64	bigint
gasUsed	uint64	bigint
gasPrice	big.Int	bigint
input	[]byte	text

key	evmType	pgType
logs	log	text
nonce	uint64	bigint
txStr	string	text
contractAddress	string	text
error	string	text
decollator	""	text

traces数据

trace包含内容为：CREATE，CALL，CALLCODE，DELEGATECALL，SELFDESTRUCT操作及预编译合约的调用。这里抓取trace的种类可以自定义。该数据的获取通过打开`vm.Config`的Debug选项，然后对于每一个trace都会执行`captureState`函数捕获每一个trace的相关信息。

```
type StructLog struct {
    Pc            uint64                     `json:"pc"`
    Op            OpCode                     `json:"op"`
    Gas           uint64                     `json:"gas"`
    GasCost       uint64                     `json:"gasCost"`
    Memory        []byte                     `json:"memory"`
    MemorySize    int                        `json:"memSize"`
    Stack         []*big.Int                 `json:"stack"`
    Storage       map[common.Hash]common.Hash `json:"-"`
    Depth         int                        `json:"depth"`
    RefundCounter uint64                     `json:"refund"`
    Err           error                      `json:"-"`
}
```

在捕获trace的过程中由于stack是一个1M大小的数组，当碰到6810086中的`0x68b71d202dc52ad80812b563f3f6b0aaf1f19c04c1260d13055daad5b88a36a8`交易时，其trace数量为100W，因为StructLog中的指针有指向每一个trace中stack的副本，导致其内存分配巨大而一直得不到释放，并行重放区块会直接被卡死，因此在快速重放时一般会屏蔽statck和memory的抓取。发现问题后可以使用单线程抓取某一区块的全部trace信息。

> By using the three main trace types (call, create, suicide), we can generalise the type of ether transfers that can occur:
>
> - **call**: Used to **transfer** ether from one account to another and/or to call a smart contract function defined by parameters in the data field. This trace also encompasses **delegatecall** and **callcode**.
> - **create**: Used to create a smart contract, and ether is **transferred** to the newly created smart contract
> - **suicide**: Used by an owner of a smart contract to kill the smart contract. Triggers a **transfer** of ether for a refund for killing a contract. Additionally, killing a smart contract can free memory in the blockchain, which can also affects the value transferred.
> - [ethereum-traces-not-transactions-3f0533d26aa](

traces数据的格式为：

key	evmType	pgType
id	uint64	bigint
blockHash	[HashLength]byte	text (32 byte Keccak256)
blockHeight	uint64	bigint
txHash	[HashLength]byte	text (32 byte Keccak256)
txIndex	int	bigint
pc	int	bigint
op	string	string
depth	int	bignint
refundCounter	int	bigint
precompiled	string	text
err	string	text
decollator	""	text

events数据

events数据主要是用来分析`ERC20 token`的转账情况，因为根据`ERC 20`规范每一笔转账操作都会产生一个event事件。events数据主要通过交易执行之后产生的receipt.logs来获取。

```
receipt.Logs = statedb.GetLogs(tx.Hash())
```

```
id = blockNumber*10^5 + idx
```

key	evmType	pgType
id	uint64	bigint
eventAddress	[AddressLength]byte a hex string	text
topics	[]byte -> string	text
eventData	[]byte -> string	text
blockNumber	uint64	bigint
transactionHash	[HashLength]byte	text
transactionIndex	uint	bignint
blockHash	[HashLength]byte	text
logIndex	uint	bigint
removed	bool	boolean

key	evmType	pgType
decollator	""	text

其它数据

geth-query工具可以自定义导出数据，根据哪些数据具备分析价值可以通过修改工具代码或者`go-ethereum`代码进行导出数据分析。

1. 对于合约运行失败的交易统计，可以从以下几个方面统计：交易失败类型统计(柱状图)，交易失败数目统计(折线图，每1W个块或者10W个块为单位)。
2. trace数量由前面的块到后面的块，交易的trace数量逐步递增的曲线。
3. 交易消耗的gas分析。
4. 热点合约的分析。
5. 交易地址的网络图。
6. 交易运行特征trace分析。
7. 所有历史交易随时间变化trace数量的调用关系。
8. 交易执行高频trace，低频trace。