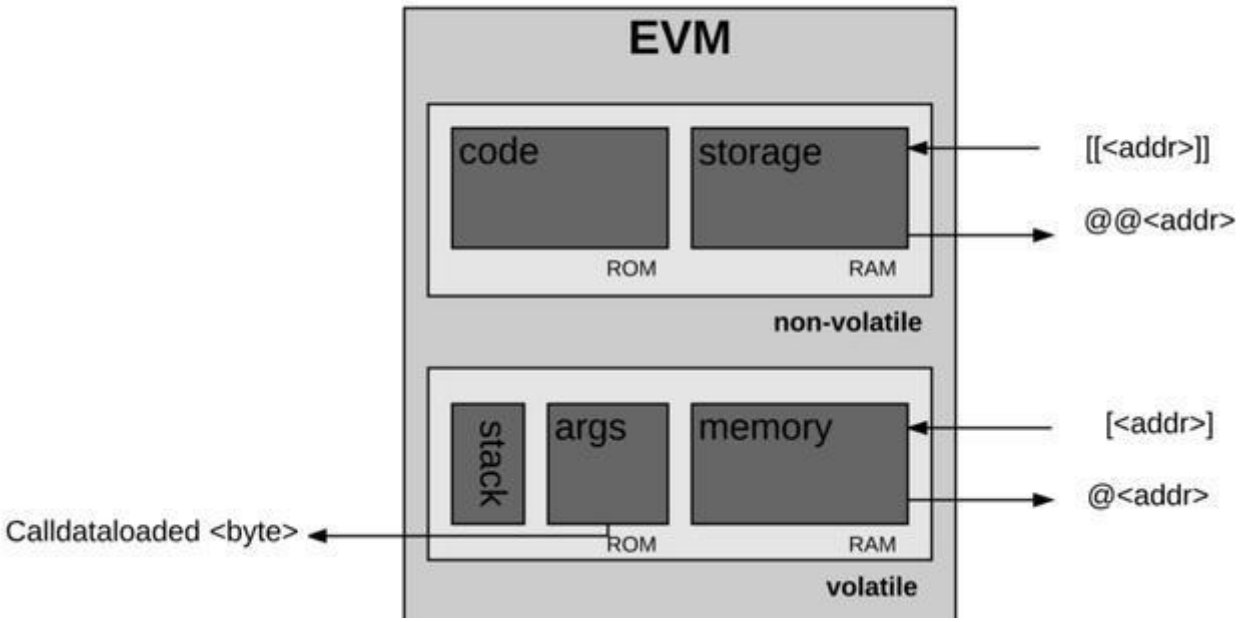
以太坊黄皮书详解（三）

2018–06–12 Ethereum

六、虚拟机的执行模型

上文三四五部分都是从流程上对交易执行（包括合约创建和合约调用）进行了介绍。本章节介绍以太坊中的虚拟机执行的流程。以太坊虚拟机EVM是图灵完备虚拟机器。EVM存在而典型图灵完备机器不存在的唯一限制就是EVM本质上是被gas束缚。因此，可以完成的计算总量本质上是被提供的gas总量限制的。

6.1 基本模型



- EVM是基于栈（先进后出）的架构。EVM中每个堆栈项的大小为256位，堆栈有一个最大的大小，为1024位。
- EVM有内存，项目按照可寻址字节数组来存储。内存是易失性的，也就是数据是不持久的。

- EVM也有一个存储器。不像内存，存储器是非易失性的，并作为系统状态的一部分进行维护。EVM分开保存程序代码，在虚拟ROM 中只能通过特殊指令来访问。
- EVM同样有属于它自己的语言：“EVM字节码”，在以太坊上运行的智能合约时，通常都是用高级语言例如Solidity来编写代码。然后将它编译成EVM可以理解的EVM字节码。

```
// EVM is the Ethereum Virtual Machine base object and provides
// the necessary tools to run a contract on the given state with
// the provided context. It should be noted that any error
// generated through any of the calls should be considered a
// revert-state-and-consume-all-gas operation, no checks on
// specific errors should ever be performed. The interpreter makes
// sure that any errors generated are to be considered faulty code.
//
// The EVM should never be reused and is not thread safe.
type EVM struct {
        // Context provides auxiliary blockchain related information
        Context
        // StateDB gives access to the underlying state
        StateDB StateDB
        // Depth is the current call stack
        depth int

        // chainConfig contains information about the current chain
        chainConfig *params.ChainConfig
        // chain rules contains the chain rules for the current epoch
        chainRules params.Rules
        // virtual machine configuration options used to initialise the
        // evm.
        vmConfig Config
        // global (to this context) ethereum virtual machine
        // used throughout the execution of the tx.
        interpreter *Interpreter
        // abort is used to abort the EVM calling operations
        // NOTE: must be set atomically
        abort int32
        // callGasTemp holds the gas available for the current call. This is neede
        // available gas is calculated in gasCall* according to the 63/64 rule and
        // applied in opCall*.
        callGasTemp uint64
}

// Interpreter is used to run Ethereum based contracts and will utilise the
// passed environment to query external sources for state information.
// The Interpreter will run the byte code VM based on the passed
```

```
// configuration.
type Interpreter struct {
        evm      *EVM
        cfg      Config
        gasTable params.GasTable
        intPool  *intPool  //栈

        readOnly   bool   // Whether to throw on stateful modifications
        returnData []byte // Last CALL's return data for subsequent reuse
}
```

6.2 费用

以太坊虚拟机执行过程中，主要有3类费用。

- 执行过程中的运算费用。
- 创建或者调用其他合约消耗的费用。
- 新增的存储的费用。

6.3 运行环境

合约执行过程中的运行环境包括：系统状态σ,可用gas值g，以及其他包含在I中的一些值，在第四和第五部分也都有涉及到。

- I_a为合约执行的当前账户。对于合约创建则为新创建的合约账户，如果是合约调用，则为接收者账户。
- I_o为原始调用者，即该条交易的发送者。
- I_p为gas价格。
- I_d为合约执行的输入数据。
- I_s为合约当前调用者，如果是条简单的交易，则为交易的发送者。
- I_v为value，单位为Wei，即要转移给当前账户的以太币。
- I_b需要被执行的机器码。合约创建的时候该处即为初始化合约的字节码。
- I_H当前区块的header。
- I_e当前的栈深度。
- I_w相关权限。

执行模型可以用公式121表示。其中子状态如公式122所示。（该处子状态定义和之前是相同的）

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{121}$$

$$A \equiv (\mathbf{s}, \mathbf{l}, \mathbf{t}, r) \tag{122}$$

6.4 执行过程

- 执行刚开始时，内存和堆栈都是空的，程序计数器为0。
- 然后EVM开始递归的执行交易，为每个循环计算系统状态和机器状态。系统状态也就是以太坊的全局状态(global state)。机器状态包含：可获取的gas，程序计数器，内存的内容，内存中字的活跃数，堆栈的内容。
- 堆栈中的项从栈顶被删除（POP）或者添加（PUSH）。
- 每个循环，剩余的gas都会被减少相应的量，程序计数器也会增加。

在每个循环的结束，都有四种可能性：

- 机器到达异常状态（err != nil 的情况。例如 gas不足，无效指令，堆栈项不足，堆栈项会溢出1024，无效的JUMP/JUMPI目的地等等）因此停止，并丢弃任何的更改进入后续处理下一个循环。
- 机器到达了受控停止（到达执行过程的终点，halts或者revert），整个结束了，机器就会产生一个合成状态，执行之后的剩余gas、产生的子状态、以及组合输出。
- 假设执行没有遇到异常状态，继续循环执行下一步。

```
// Run loops and evaluates the contract's code with the given input data and retur
// the return byte-slice and an error if one occurred.
//
// It's important to note that any errors returned by the interpreter should be
// considered a revert-and-consume-all-gas operation except for
// errExecutionReverted which means revert-and-keep-gas-left.
func (in *Interpreter) Run(contract *Contract, input []byte) (ret []byte, err erro
        // Increment the call depth which is restricted to 1024
        in.evm.depth++
        defer func() { in.evm.depth-- }()

        // Reset the previous call's return data. It's unimportant to preserve the
        // as every returning call will return new data anyway.
        in.returnData = nil

        // Don't bother with the execution if there's no code.
        if len(contract.Code) == 0 {
                return nil, nil
        }

        //机器状态包含：可获取的gas，程序计数器pc，内存的内容mem，内存中字的活跃数，堆栈的内容s
```

```
var (
        op    OpCode        // current opcode
        mem   = NewMemory() // bound memory
        stack = newstack()  // local stack
        // For optimisation reason we're using uint64 as the program count
        // It's theoretically possible to go above 2^64. The YP defines th
        // to be uint256. Practically much less so feasible.
        pc   = uint64(0) // program counter
        cost uint64
        // copies used by tracer
        pcCopy  uint64 // needed for the deferred Tracer
        gasCopy uint64 // for Tracer to log gas remaining before execution
        logged  bool   // deferred Tracer should ignore already logged ste
)
contract.Input = input

if in.cfg.Debug {
        defer func() {
                if err != nil {
                        if !logged {
                                in.cfg.Tracer.CaptureState(in.evm, pcCopy,
                        } else {
                                in.cfg.Tracer.CaptureFault(in.evm, pcCopy,
                        }
                }
        }()
}
// 循环直到三种结束状态中的一种。
// 1.异常情况，各种不合法的情况出现时，或者执行出错，会直接return
// 2.执行中返回REVERT错误。
// 3.执行中中断。
// The Interpreter main run loop (contextual). This loop runs until either
// explicit STOP, RETURN or SELFDESTRUCT is executed, an error occurred du
// the execution of one of the operations or until the done flag is set by
// parent context.
for atomic.LoadInt32(&in.evm.abort) == 0 {
        if in.cfg.Debug {
                // Capture pre-execution values for tracing.
                logged, pcCopy, gasCopy = false, pc, contract.Gas
        }

        // Get the operation from the jump table and validate the stack to
        // enough stack items available to perform the operation.
        // 获取当前要执行的操作
        op = contract.GetOp(pc)
        // 确认操作是否在操作集合表中
```

```
operation := in.cfg.JumpTable[op]
if !operation.valid {
        return nil, fmt.Errorf("invalid opcode 0x%x", int(op))
}
// 确保目前栈满足操作要求，比方说有些操作是二元的，那么栈中至少要有两个数据。
if err := operation.validateStack(stack); err != nil {
        return nil, err
}
// If the operation is valid, enforce and write restrictions
if err := in.enforceRestrictions(op, operation, stack); err != nil
        return nil, err
}

var memorySize uint64
// calculate the new memory size and expand the memory to fit
// the operation
// 计算新操作需要的内存空间
if operation.memorySize != nil {
        memSize, overflow := bigUint64(operation.memorySize(stack)
        if overflow {
                return nil, errGasUintOverflow
        }
        // memory is expanded in words of 32 bytes. Gas
        // is also calculated in words.
        if memorySize, overflow = math.SafeMul(toWordSize(memSize)
                return nil, errGasUintOverflow
        }
}
// consume the gas and return an error if not enough gas is availa
// cost is explicitly set so that the capture state defer method c
// 消耗gas
cost, err = operation.gasCost(in.gasTable, in.evm, contract, stack
if err != nil || !contract.UseGas(cost) {
        return nil, ErrOutOfGas
}
if memorySize > 0 {
        mem.Resize(memorySize)
}

if in.cfg.Debug {
        in.cfg.Tracer.CaptureState(in.evm, pc, op, gasCopy, cost,
        logged = true
}

// execute the operation
// 执行具体的操作，比方说加法，就是从栈中取出两个值后相加
```

```
                res, err := operation.execute(&pc, in.evm, contract, mem, stack)
                // verifyPool is a build flag. Pool verification makes sure the in
                // of the integer pool by comparing values to a default value.
                if verifyPool {
                        verifyIntegerPool(in.intPool)
                }
                // if the operation clears the return data (e.g. it has returning
                // set the last return to the result of the operation.
                if operation.returns {
                        in.returnData = res
                }

                switch {
                case err != nil:
                        return nil, err
                case operation.reverts:
                        return res, errExecutionReverted
                case operation.halts:
                        return res, nil
                case !operation.jumps:
                        pc++ //计数器加一，接着执行
                }
        }
        return nil, nil
}
```

6.4.1 整体流程的形式化表示

虚拟机行执行合约的整体流程可以看作是输入系统状态σ,可用gas值g，输入项I，可以获取到的账户集合T。经过虚拟机函数Ξ后，得到新的系统状态σ',执行剩余可用gasμ'_g,交易子状态A和组合输出$\mathbf{o}$。如公式123所示。

执行的过程是一个迭代执行的过程，定义X为该迭代函数。如公式124所示。

其中机器状态μ,包括：

- 可用gas值g，初始值即为输入的可用gas值，公式125。
- 程序计数器pc，初始值为0，公式126。
- 内存内容m，初始为全0，公式127。
- 内存内容中活跃的字数i，初始为0，公式128。
- 栈s，初始为空的栈，公式129。

- 组合输出，初始为空，公式130.

$$\Xi(\boldsymbol{\sigma}, g, I, T) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_{\mathrm{g}}, A, \mathbf{o}) \tag{123}$$
$$(\boldsymbol{\sigma}', \boldsymbol{\mu}', A, \ldots, \mathbf{o}) \equiv X\Big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\Big) \tag{124}$$
$$\boldsymbol{\mu}_{\mathrm{g}} \equiv g \tag{125}$$
$$\boldsymbol{\mu}_{\mathrm{pc}} \equiv 0 \tag{126}$$
$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, \ldots) \tag{127}$$
$$\boldsymbol{\mu}_{\mathrm{i}} \equiv 0 \tag{128}$$
$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{129}$$
$$\boldsymbol{\mu}_{\mathbf{o}} \equiv () \tag{130}$$

$$X\Big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\Big) \equiv \begin{cases} \Big(\varnothing, \boldsymbol{\mu}, A^0, I, \varnothing\Big) & \text{if} \quad Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ \Big(\varnothing, \boldsymbol{\mu}', A^0, I, \mathbf{o}\Big) & \text{if} \quad w = REVERT \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if} \quad \mathbf{o} \neq \varnothing \\ X\Big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\Big) & \text{otherwise} \end{cases} \tag{131}$$

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{132}$$
$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{133}$$
$$\boldsymbol{\mu}' \equiv \boldsymbol{\mu} \text{ except:} \tag{134}$$
$$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{135}$$

迭代函数X定义如公式131–135所示。

- 如果迭代过程中遇到异常，即$Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I)$,异常的具体情况见下一小节。则迭代结束，返回空的状态集$\varnothing$，原始机器状态$\boldsymbol{\mu}$，空的子状态A^0,空的组合输出$\varnothing$。
- 如果执行中隐式中断，当前要执行的操作为revert，则迭代结束，返回空的状态集$\varnothing$，临时机器状态$\boldsymbol{\mu}'$（扣除了gas消耗公式134–135），空的子状态A^0,组合输出$\mathbf{o}$。
- 如果执行过程正常，且达到了受控停止的状态，即组合输出$\mathbf{o}$不为空，则返回执行结果以及组合输出。组合输出由$H(\boldsymbol{\mu}, I)$确定公式132，详解见后文6.4.4。
- 如果执行过程正常，且组合输出为空，继续进行下一轮迭代。

当前要被执行的操作定义如公式136.

- 当程序计数器小于虚拟机字节码时，当前要执行的操作为字节码中对应操作。
- 否则为STOP

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ STOP & \text{otherwise} \end{cases} \tag{136}$$

6.4.2异常的形式化定义

虚拟机执行过程中的异常有以下几种情况,如公式137：

$$
\begin{aligned}
Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv\ & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\
& \delta_w = \varnothing \quad \vee \\
& \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\
& (w = JUMP \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\
& (w = JUMPI \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\
& (w = RETURNDATACOPY \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|) \quad \vee \\
& \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \quad \vee \\
& (\neg I_{\mathrm{w}} \wedge W(w, \boldsymbol{\mu}))
\end{aligned} \tag{137}
$$

$$
\begin{aligned}
W(w, \boldsymbol{\mu}) \equiv\ & w \in \{CREATE, SSTORE, SELFDESTRUCT\} \quad \vee \\
& LOG0 \leq w \wedge w \leq LOG4 \quad \vee \\
& w \in \{CALL, CALLCODE\} \wedge \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0
\end{aligned} \tag{138}
$$

- gas值不足以支付花费。即$\boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I)$。

```
func (in *Interpreter) Run(contract *Contract, input []byte) (ret []byte, err e
  //...
  for atomic.LoadInt32(&in.evm.abort) == 0 {
      //...
      // gas值不足以支付花费
      cost, err = operation.gasCost(in.gasTable, in.evm, contract, stack, mem,
      if err != nil || !contract.UseGas(cost) {
          return nil, ErrOutOfGas
      }
      //...
  }
}
```

- 当前的操作是无效的。$\delta_w = \varnothing$。当前操作w对应的字节码为空。

```
func (in *Interpreter) Run(contract *Contract, input []byte) (ret []byte, err e
  //...
  for atomic.LoadInt32(&in.evm.abort) == 0 {
      //...
      // 获取当前要执行的操作
      op = contract.GetOp(pc)
      // 确认操作是否在操作集合表中
      operation := in.cfg.JumpTable[op]
      if !operation.valid {
          return nil, fmt.Errorf("invalid opcode 0x%x", int(op))
      }
```

```
        //...
    }
}
```

- 当前栈不合法（比方说二元操作，栈里只有一个数据）$\|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w$

```
func (in *Interpreter) Run(contract *Contract, input []byte) (ret []byte, err e
    //...
    for atomic.LoadInt32(&in.evm.abort) == 0 {
        //...
        // 确保目前栈满足操作要求，比方说有些操作是二元的，那么栈中至少要有两个数据。
        if err := operation.validateStack(stack); err != nil {
            return nil, err
        }
        // If the operation is valid, enforce and write restrictions
        if err := in.enforceRestrictions(op, operation, stack); err != nil {
            return nil, err
        }
        //...
    }
}
```

- 当前操作为跳转，但是跳转到的地方不合法 $(w = JUMP \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}}))(w = JUMPI \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}}))$

```
//跳转操作
func opJump(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack
        pos := stack.pop()
        if !contract.jumpdests.has(contract.CodeHash, contract.Code, pos) {
                nop := contract.GetOp(pos.Uint64())
                return nil, fmt.Errorf("invalid jump destination (%v) %v", nop, po
        }
        *pc = pos.Uint64()

        evm.interpreter.intPool.put(pos)
        return nil, nil
}

func opJumpi(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stac
        pos, cond := stack.pop(), stack.pop()
        if cond.Sign() != 0 {
                if !contract.jumpdests.has(contract.CodeHash, contract.Code, pos)
                        nop := contract.GetOp(pos.Uint64())
                        return nil, fmt.Errorf("invalid jump destination (%v) %v",
                }
                *pc = pos.Uint64()
```

```
        } else {
                *pc++
        }

        evm.interpreter.intPool.put(pos, cond)
        return nil, nil
}
```

- 当前操作为返回数据操作，但是返回的数据长度超过了允许长度。$(w = RETURNDATACOPY \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|)$

```
func opReturnDataCopy(pc *uint64, evm *EVM, contract *Contract, memory *Memory, st
        var (
                memOffset  = stack.pop()
                dataOffset = stack.pop()
                length     = stack.pop()

                end = evm.interpreter.intPool.get().Add(dataOffset, length)
        )
        defer evm.interpreter.intPool.put(memOffset, dataOffset, length, end)
        //判断返回数据长度是否超过允许长度
        if end.BitLen() > 64 || uint64(len(evm.interpreter.returnData)) < end.Uint
                return nil, errReturnDataOutOfBounds
        }
        memory.Set(memOffset.Uint64(), length.Uint64(), evm.interpreter.returnData

        return nil, nil
}
```

- 新的栈的大小大于1024,$\|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024$
- 没用权限却尝试非静态调用$\neg I_{\mathrm{w}} \wedge W(w, \boldsymbol{\mu})$。其中非静态调用有，CREATE，SSTORE，SELFDESTRUCT，LOG0，LOG1，LOG2，LOG3，LOG4，以及CALL和CALLCHAINCODE，如公式138。

```
func (in *Interpreter) enforceRestrictions(op OpCode, operation operation, stack *
        if in.evm.chainRules.IsByzantium {
                if in.readOnly {
                        // If the interpreter is operating in readonly mode, make
                        // state-modifying operation is performed. The 3rd stack i
                        // for a call operation is the value. Transferring value f
                        // account to the others means the state is modified and s
                        // return with an error.
                        if operation.writes || (op == CALL && stack.Back(2).BitLen
                                return errWriteProtection
```

```
                        }
                }
        }
        return nil
}
```

6.4.3 跳转表的形式化定义

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{139}$$

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{if} \quad \mathbf{c}[i] = JUMPDEST \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{140}$$

$$N(i, w) \equiv \begin{cases} i + w - PUSH1 + 2 & \text{if} \quad w \in [PUSH1, PUSH32] \\ i + 1 & \text{otherwise} \end{cases} \tag{141}$$

公式139–141为对跳转表的定义。

- 跳转到c，指跳转到c的第0个指令。公式140.
- 跳转到c的第i个指令，如果i大于等于c的长度，则跳转到空，否则跳转到i的下一个有效指令的位置。公式140.

其中$N(i, w)$指下一个有效的操作指令的位置，因为是找有效的指令，和不是数据，所以遇到push指令的时候需要跳过push进去的数据。

- 如果当前命令为PUSHX，则跳转到i+X+1（因为栈里接下来全是push进去的数据，寻找指令，要跳过这些数据）。
- 否则为i + 1.

6.4.4 正常的可控中断的形式化定义

正常的可控中断有以下三种：

- 当前的操作为RETURN或者REVERT，相应的进行返回。
- 当前操作为STOP或者SELFDESTRUCT。
- 其他情况返回空。

注：这里STOP或者SELFDESTRUCT返回的并不是空，所以在6.1中也是o不为空的情况。

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{RETURN}(\boldsymbol{\mu}) \text{ if } & w \in REVERT\} \\ () & \text{if } \quad w \in \{STOP, SELFDESTRUCT\} \\ \varnothing & \text{otherwise} \end{cases} \tag{142}$$

6.5 执行的生命周期的形式化表示

执行过程中栈元素从栈顶入栈和出栈（黄皮书中栈顶元素为0），每次从栈顶取出操作执行，并根据操作进行相应的出栈和入栈操作。栈中其余的元素不变。如公式143–146所示。

$$O\Big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\Big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{143}$$

$$\Delta \equiv \alpha_w - \delta_w \tag{144}$$

$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{145}$$

$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x - \Delta] \tag{146}$$

公式143表示，每次操作原先的状态$\boldsymbol{\sigma}$,原先的机器状态$\boldsymbol{\mu}$,原先的子状态A,原先的参数I，执行后为新临时状态$\boldsymbol{\sigma}'$,新的机器状态$\boldsymbol{\mu}'$,新的子状态A’,参数I不变。

公式144表示，栈的变化，其中α_w为当前操作的入栈集合,δ_w为当前操作的出栈集合。

公式145表示，栈的长度，即为在原先栈的长度+变化的长度。

公式146表示，栈中的元素，除了新增的元素α_w，其余的不变。（黄皮书中栈顶是0，所以如果入栈元素多于出栈元素，则原先元素的索引都会加相应偏移）。

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{147}$$

$$\boldsymbol{\mu}'_{\mathrm{pc}} \equiv \begin{cases} J_{\mathrm{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = JUMP \\ J_{\mathrm{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = JUMPI \\ N(\boldsymbol{\mu}_{\mathrm{pc}}, w) & \text{otherwise} \end{cases} \tag{148}$$

公式147–148表示每一次迭代gas和程序计数器pc的变化。其中gas值为原有gas减去当前操作的花费，pc在JUMP和JUMPI的时候是跳转到相应位置，其余情况是加一，到下一条指令。

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{149}$$

$$\boldsymbol{\mu}'_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathrm{i}} \tag{150}$$

$$A' \equiv A \tag{151}$$

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{152}$$

公式149–152，假设操作对其余的元素都不产生影响。实际上不同操作会对不同部分有影响。

七、区块的最终确定

在区块的生成过程中，更多的时候是一棵树状的结构。为了在树形结构中确认一条链作为区块链，在以太坊中定义“最重”的链为主链。 这里的重指的是累计难度（td，total difficulty定义如公式153–154），也就是td最大的链为主链。

$$B_{\mathrm{t}} \equiv B'_{\mathrm{t}} + B_{\mathrm{d}} \tag{153}$$

$$B' \equiv P(B_H) \tag{154}$$

公式153–154表示当前区块的累积难度td值为其父区块的累积难度td值加上当前区块的难度值。

区块的最终确定涉及到四个阶段：

- 校验ommers。
- 校验交易。
- 计算和提交奖励。
- 验证世界状态和区块的nonce值。

```
// ValidateBody validates the given block's uncles and verifies the the block
// header's transaction and uncle roots. The headers are assumed to be already
// validated at this point.
func (v *BlockValidator) ValidateBody(block *types.Block) error {
        // Check whether the block's known, and if not, that it's linkable
        if v.bc.HasBlockAndState(block.Hash(), block.NumberU64()) {
                return ErrKnownBlock
        }
        // 验证父区块是否存在
        if !v.bc.HasBlockAndState(block.ParentHash(), block.NumberU64()-1) {
                if !v.bc.HasBlock(block.ParentHash(), block.NumberU64()-1) {
                        return consensus.ErrUnknownAncestor
                }
                return consensus.ErrPrunedAncestor
        }
        // 验证ommers
        // Header validity is known at this point, check the uncles and transactio
        header := block.Header()
        if err := v.engine.VerifyUncles(v.bc, block); err != nil {
                return err
        }
        if hash := types.CalcUncleHash(block.Uncles()); hash != header.UncleHash {
                return fmt.Errorf("uncle root hash mismatch: have %x, want %x", ha
        }
        // 验证交易的hash值
        if hash := types.DeriveSha(block.Transactions()); hash != header.TxHash {
                return fmt.Errorf("transaction root hash mismatch: have %x, want %
        }
```

```
        return nil
}
```

7.1 Ommer的校验

Ommer或者说uncle是指父区块的兄弟区块，以太坊中block的uncles字段用以存储uncle区块，在区块的最终确定的时候需要对ommer进行校验。

```
// VerifyUncles verifies that the given block's uncles conform to the consensus
// rules of the stock Ethereum ethash engine.
func (ethash *Ethash) VerifyUncles(chain consensus.ChainReader, block *types.Block
        // If we're running a full engine faking, accept any input as valid
        if ethash.config.PowMode == ModeFullFake {
                return nil
        }
        // 每个区块最多有两个uncle区块
        // Verify that there are at most 2 uncles included in this block
        if len(block.Uncles()) > maxUncles {
                return errTooManyUncles
        }
        // Gather the set of past uncles and ancestors
        uncles, ancestors := set.New(), make(map[common.Hash]*types.Header)

        // 统计七代以内的区块
        number, parent := block.NumberU64()-1, block.ParentHash()
        for i := 0; i < 7; i++ {
                ancestor := chain.GetBlock(parent, number)
                if ancestor == nil {
                        break
                }
                ancestors[ancestor.Hash()] = ancestor.Header()
                for _, uncle := range ancestor.Uncles() {
                        uncles.Add(uncle.Hash())
                }
                parent, number = ancestor.ParentHash(), number-1
        }
        ancestors[block.Hash()] = block.Header()
        uncles.Add(block.Hash())

        // 确认该节点
        // Verify each of the uncles that it's recent, but not an ancestor
        for _, uncle := range block.Uncles() {
                // Make sure every uncle is rewarded only once
                hash := uncle.Hash()
                if uncles.Has(hash) {
```

```
                        return errDuplicateUncle
                }
                uncles.Add(hash)

                // Make sure the uncle has a valid ancestry
                // 确认uncle区块是确实是uncle区块
                if ancestors[hash] != nil {
                        return errUncleIsAncestor
                }
                if ancestors[uncle.ParentHash] == nil || uncle.ParentHash == block
                        return errDanglingUncle
                }
                // 验证uncle区块的header
                if err := ethash.verifyHeader(chain, uncle, ancestors[uncle.Parent
                        return err
                }
        }
        return nil
}
```

$$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{\mathbf{U} \in B_{\mathbf{U}}} V(\mathbf{U}) \ \wedge \ k(\mathbf{U}, P(\mathbf{B_H})_{\mathbf{H}}, 6) \tag{155}$$

$$k(U, H, n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U, H) \ \vee \ k(U, P(H)_H, n-1) & \text{otherwise} \end{cases} \tag{156}$$

$$s(U, H) \equiv (P(H) = P(U) \ \wedge \ H \neq U \ \wedge \ U \notin B(H)_{\mathbf{U}}) \tag{157}$$

在以太坊中一个区块的

- 兄弟区块$s(U, H)$定义如公式157，兄弟区块是指与区块自身有不同，且父区块相同$(P(H) = P(U) \wedge H \neq U$；并且该兄弟区块不在自己的uncles区块中$U \notin B(H)_{\mathbf{U}}$
- U是否是H的n代以内uncle定义如公式156，若n为0，则返回false（没有0代的概念），否则要么U是H的兄弟区块（一代），要么U是H的父区块的n-1代叔区块。
- uncle区块数目不超过2个，$\|B_{\mathbf{U}}\| \leqslant 2$,公式155
- uncle区块为7代以内的区块,$V(\mathbf{U}) \ \wedge \ k(\mathbf{U}, P(\mathbf{B_H})_{\mathbf{H}}, 6)$（公式里为父区块的6代以内，即自己的兄弟区块是不会包含在内的）

7.2 校验交易

$$B_{H\mathrm{g}} = \ell(\mathbf{R})_{\mathrm{u}} \tag{158}$$

- 校验block中的txHash是否为block.Transaction的hash。见本章开始部分代码中ValidateBody的最后部分。

- 虚拟机逐条执行交易，执行后校验所用的gas值是否和区块的已用gas值相同，见公式158.
- 校验虚拟机执行后的bloom以及receipt是否与区块头中的bloom以及receiptHash相同。
- 校验虚拟机执行之后的世界状态的树的根是否与区块头中的Root相同。

```
// ValidateState validates the various changes that happen after a state
// transition, such as amount of used gas, the receipt roots and the state root
// itself. ValidateState returns a database batch if the validation was a success
// otherwise nil and an error is returned.
func (v *BlockValidator) ValidateState(block, parent *types.Block, statedb *state.
        header := block.Header()
        // 校验交易的usedGas是否为区块的GasUsed
        if block.GasUsed() != usedGas {
                return fmt.Errorf("invalid gas used (remote: %d local: %d)", block
        }
        // Validate the received block's bloom with the one derived from the gener
        // For valid blocks this should always validate to true.
        rbloom := types.CreateBloom(receipts)
        if rbloom != header.Bloom {
                return fmt.Errorf("invalid bloom (remote: %x  local: %x)", header.
        }

        // 校验收款的hash值
        // Tre receipt Trie's root (R = (Tr [[H1, R1], ... [Hn, R1]]))
        receiptSha := types.DeriveSha(receipts)
        if receiptSha != header.ReceiptHash {
                return fmt.Errorf("invalid receipt root hash (remote: %x local: %x
        }
        // 校验state
        // Validate the state root against the received state root and throw
        // an error if they don't match.
        if root := statedb.IntermediateRoot(v.config.IsEIP158(header.Number)); hea
                return fmt.Errorf("invalid merkle root (remote: %x local: %x)", he
        }
        return nil
}
```

7.3 计算奖励

$$\Omega(B,\boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:} \tag{159}$$

$$\boldsymbol{\sigma}'[\mathbf{B}_{H\mathrm{c}}]_{\mathrm{b}} = \boldsymbol{\sigma}[\mathbf{B}_{H\mathrm{c}}]_{\mathrm{b}} + \left(1 + \frac{\|\mathbf{B}_{\mathbf{U}}\|}{32}\right) R_{\mathrm{block}} \tag{160}$$

$$\forall_{\mathbf{U}\in\mathbf{B}_{\mathbf{U}}} : \tag{161}$$

$$\boldsymbol{\sigma}'[\mathbf{U}_{\mathrm{c}}] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[\mathbf{U}_{\mathrm{c}}] = \varnothing \ \wedge \ R = 0 \\ \mathbf{a}' & \text{otherwise} \end{cases}$$

$$\mathbf{a}' \equiv (\boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{n}}, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{b}} + R, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathbf{s}}, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{c}}) \tag{162}$$

$$R \equiv \left(1 + \frac{1}{8}(U_{\mathrm{i}} - B_{H\mathrm{i}})\right) R_{\mathrm{block}} \tag{163}$$

$$\text{Let} \quad R_{\mathrm{block}} = 3 \times 10^{18} \tag{164}$$

- 矿工的奖励为固定奖励R_{block}如公式164,如果区块uncles不为空，则每个uncle会带来额外32分之一的R_{block}奖励。如公式160，矿工的余额为原余额加上奖励。
- 对于该区块中的uncles，每个uncle块，计算其矿工的奖励，如公式163，是根据uncle块与当前块的代差来计算的。例如差1代，uncle块为当前块父区块的兄弟，那么该uncle块的矿工可以拿到八分之七的R_{block}奖励。差的代数越多，拿到的奖励越少。

```
// AccumulateRewards credits the coinbase of the given block with the mining
// reward. The total reward consists of the static block reward and rewards for
// included uncles. The coinbase of each uncle block is also rewarded.
func accumulateRewards(config *params.ChainConfig, state *state.StateDB, header *t
        // Select the correct block reward based on chain progression
        blockReward := FrontierBlockReward
        if config.IsByzantium(header.Number) {
                blockReward = ByzantiumBlockReward
        }
        // Accumulate the rewards for the miner and any included uncles
        reward := new(big.Int).Set(blockReward)
        // 计算uncle块中矿工的奖励
        r := new(big.Int)
        for _, uncle := range uncles {
                r.Add(uncle.Number, big8)
                r.Sub(r, header.Number)
                r.Mul(r, blockReward)
                r.Div(r, big8)
                state.AddBalance(uncle.Coinbase, r)

                r.Div(blockReward, big32)
                reward.Add(reward, r)
        }
```

```
        // 当前矿工的奖励为固定奖励加上跟uncle区块数量相关的一部分奖励
        state.AddBalance(header.Coinbase, reward)
}
```

7.4 校验state和nonce

- 区块执行前的世界状态定义为$\Gamma(B)$,如公式165，即执行Process之前的世界状态。
- 执行即对区块的每条交易，运行交易转换函数（虚拟机执行，ApplyTrasaction）如公式170，每条交易执行完后，走区块的转换函数（Finalize，增加矿工奖励等），如公式169和174所示。
- 执行交易过程中会产生交易的收据证明，相关内容如公式171–173所示。
- 执行后的区块与最终区块的主要区别为区块头中的nonce值，mixHash值，如公式166–168所示。

```
// Process processes the state changes according to the Ethereum rules by runni
// the transaction messages using the statedb and applying any rewards to both
// the processor (coinbase) and any included uncles.
//
// Process returns the receipts and logs accumulated during the process and
// returns the amount of gas that was used in the process. If any of the
// transactions failed to execute due to insufficient gas it will return an err
func (p *StateProcessor) Process(block *types.Block, statedb *state.StateDB, cf
  var (
      receipts types.Receipts
      usedGas  = new(uint64)
      header   = block.Header()
      allLogs  []*types.Log
      gp       = new(GasPool).AddGas(block.GasLimit())
  )
  // Mutate the the block and state according to any hard-fork specs
  if p.config.DAOForkSupport && p.config.DAOForkBlock != nil && p.config.DAOFor
      misc.ApplyDAOHardFork(statedb)
  }
  // Iterate over and process the individual transactions
  for i, tx := range block.Transactions() {
      // 虚拟机执行每条交易
      statedb.Prepare(tx.Hash(), block.Hash(), i)
      receipt, _, err := ApplyTransaction(p.config, p.bc, nil, gp, statedb, hea
      if err != nil {
          return nil, nil, 0, err
      }
      // 得到交易的收据与日志
      receipts = append(receipts, receipt)
```

```
        allLogs = append(allLogs, receipt.Logs...)
    }
    // Finalize the block, applying any consensus engine specific extras (e.g. bl
    p.engine.Finalize(p.bc, header, statedb, block.Transactions(), block.Uncles()

    return receipts, allLogs, *usedGas, nil
}
```

$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_{\mathrm{i}} : \mathtt{TRIE}(L_S(\boldsymbol{\sigma}_{\mathrm{i}})) = P(B_H)_{H\mathrm{r}} & \text{otherwise} \end{cases} \tag{165}$$

公式165定义区块执行前的世界状态为$\Gamma(B)$，如果父区块为空，则世界状态为创世块状态，否则执行前的世界状态为其父区块头对应的世界状态树。

$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{except:} \tag{166}$$

$$B'_{\mathrm{n}} = n : \quad x \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \tag{167}$$

$$B'_{\mathrm{m}} = m \quad \text{with } (x, m) = \mathtt{PoW}(B^*_{\cancel{n}}, n, \mathbf{d}) \tag{168}$$

$$B^* \equiv B \quad \text{except:} \quad B^*_{\mathrm{r}} = r(\Pi(\Gamma(B), B)) \tag{169}$$

公式166–169定义了区块的世界状态为在执行了转换函数之后Π，其nonce值和mixHash值需要满足PoW即难度需求。

$$\boldsymbol{\sigma}[n] = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases} \tag{170}$$

公式170定义了第n条交易时的世界状态。当n小于0时，世界状态如前定义为$\Gamma(B)$,否则第n条交易时的世界状态为在第n–1条交易的基础上应用交易转变函数（执行交易）。

$$\mathbf{R}[n]_{\mathrm{u}} = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) + \mathbf{R}[n-1]_{\mathrm{u}} & \text{otherwise} \end{cases} \tag{171}$$

$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) \tag{172}$$

$$\mathbf{R}[n]_{\mathrm{z}} = \Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) \tag{173}$$

$\mathbf{R}[n]_{\mathrm{z}}, \mathbf{R}[n]_{\mathrm{l}}, \mathbf{R}[n]_{\mathrm{u}}$分别表示对应的第n条交易的状态码，日志，以及累积消耗的gas值。

- $\mathbf{R}[n]_{\mathrm{u}}$ 的计算如公式171所示，为第n–1个的累积消耗$\mathbf{R}[n-1]_{\mathrm{u}}$，加上当前交易的消耗$\Upsilon^g(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n])$ 。
- $\mathbf{R}[n]_{\mathrm{l}}$为在原先的日志基础上增加相应的交易日志。
- $\mathbf{R}[n]_{\mathrm{z}}$为在原先的状态码列表上增加相应的状态码。

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\boldsymbol{\sigma})) \tag{174}$$

公式174表示最终的区块为在最后交易后的世界状态上$\ell(\boldsymbol{\sigma})$,使用完善区块的转化函数$\Omega$（即本章的内容，计算奖励等等）

< 以太坊黄皮书详解（二） 搭建以太坊私链环境 >