New era 技术黄皮书

黄皮书目录

目录

简介

Abstract. 由密码学安全交易（cryptographically-secured transactions）所构成的区块链范式，已经通过一系列项目展示了它的实用性，不止是比特币。

每一个这类项目都可以看作是一个基于去中心化的单实例计算资源所构建的简单应用程序。我们可以把这种范式称为具有共享状态（shared-state）的交易化单例状态机（transactional singleton machine）。

New era 以更通用的方式实现了这种范式。它提供了大量的资源，每一个资源都拥有独立的状态和操作码，并且可以通过消息传递方式和其它资源交互。我们将讨论它的设计、实现上的问题、它所提供的机遇以及我们所设想的未来障碍。

项目背景

随着互联网连接了世界上绝大多数地方，全球信息共享 的成本越来越低。比特币网络通过共识机制、自愿遵守的社 会合约，实现了一个去中心化的价值转移系统且可以在全 球范围内自由使用，这样的技术改革展示了它的巨大力量。 这样的系统可以说是加密安全、基于交易的状态机的一种 具体应用。尽管还很简陋，但类似域名币（Namecoin）这 样的后续系统，把这种技术从最初的“货币应用”发展到了 其它领域。

New era 是以太坊成功应用范例的升级版，在保留以太坊特性的同时，解决了以太坊在区块链实践过程时遇到的主要问题，New era 是一种面向未来的区块链技术，这个系统提供了一种可信对象消息传递计算框架（a trustful object messaging compute framework)，使开发者可以用一种前所未有的计算范式来 构建软件。

黄皮书的第 9 章和第 10 章是 New era 在以太坊基础上的新技术特点，介绍 New era 的超级节点和链桥技术，使用各大洲的超级节点高速同步数据，能够有效的解决当前以太坊上的拥堵问题，使用链桥技术，可以实现 New era 上的数据资产与以太坊公链上的资产进行互通，用户在进行链交易的时候，可以进行交易链路的通道完成选择。

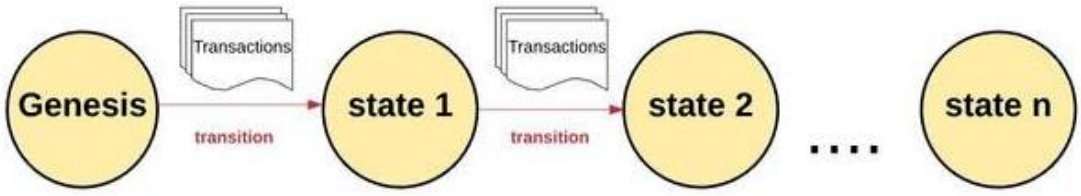
1、New era 的区块链范式

New era 本质是一个基于交易的状态机（transaction-based state machine)。其以初始状态（genesis state）为起点，通过执行交易来到达新的状态。

1.1 数学公式

$$\sigma_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \qquad (1)$$

公式 1 表示 t+1 时的状态，是由 t 时的状态经过交易 T 转变而来。转变函数为 Y。 如下图所示



$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \quad (2)$$

$$B \equiv (\ldots, (T_0, T_1, \ldots)) \quad (3)$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)\ldots) \quad (4)$$

公式 2-4 是从区块的角度来描述状态的转化过程。 公式 2: t+1 时的状态，是由 t 时的状态经过区块 B 转变而来。转变函数为 Π。

公式 3: 区块 B 是包含了一系列交易 T 的集合。

公式 4: 区块的状态转变函数 Π，相当于逐条的执行交易的状态转变 Y，然后完成所有交易转变后再经过 Ω 进行一次状态转换。（这个地方 Ω 实际上是给矿工挖坑奖励。）

1.2 数据公式的执行

在 New era 中的实际情况就是区块验证和执行的过程。

- 逐一的执行交易（也就是使用交易转变函数操作 Y 状态集）。实际就是交易比方是 A 向 B 转 10ether，则 A 账户值-10，B 账户值+10。（当然执行过程中还有 gas 消耗，这个后面详述）

- 等整个 block 交易执行完毕后，需要对矿工进行奖励。也就是需要使用 Ω 进行一次状态转换。

2、货币

New era 中有以下四种单位的货币。New era 中的各种计算都是以 Wei 为单位的。

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

3、分叉

New era 的正确运行建立在其链上只有一个链是有效的，所有人都必须要接受它。拥有多个状态（或多个链）会摧毁这个系统，因为它在哪个是正确状态的问题上不可能得到统一结果。

如果链分叉了，你有可能在一条链上拥有 10 个币，一条链上拥有 20 个币，另一条链上拥有 40 个币。在这种场景下，是没有办法确定哪个链才是最”有效的“。

不论什么时候只要多个路径产生了，一个”分叉“就会出现。为了确定哪个路径才是最有效的以及防止多条链的产生，New era 使用了一个叫做“GHOST 协议(GHOST protocol.)”的数学机制。

简单来说，GHOST 协议就是让我们必须选择一个在其上完成计算最多的路径。一个方法确定路径就是使用最近一个区块（叶子区块）的区块号，区块号代表着当前路径上总的区块数（不包含创世纪区块）。区块号越大，路径就会越长，就说明越多的挖矿算力被消耗在此路径上以达到叶子区块。

4、世界状态

New era 中的世界状态指地址(Address)与账户状态(Account State)的集合。世界状态并不是存储在链上，而是通过 Merkle Patricia tree 来维护。

4.1 账户状态

账户状态（Account State）包含四个属性。

- nonce: 如果账户是一个外部拥有账户，nonce 代表从此账户地址发送的交易序号。如果账户是一个合约账户，nonce 代表此账户创建的合约序号。用 $\sigma[a]\mathbf{n}$ 来表示。
- balance：此地址拥有 Wei 的数量。1Ether=10^18Wei。用 $\sigma[a]\mathbf{b}$ 来表示。
- storageRoot： 理论上是指 Merkle Patricia 树的根节点 256 位的 Hash 值。用 $\sigma[a]\mathbf{s}$ 来表示。公式 6 中有介绍。
- codeHash：此账户 EVM 代码的 hash 值。对于外部拥有账户，codeHash 域是一个空字符串 的 Hash 值。对于合约账户，就是代码的 Hash 作为 codeHash 保存。用 $\sigma[a]\mathbf{c}$ 来表示。

4.2 账户结构代码

```
// github.com/ethereum/go-ethereum/core/state/state_object.go
type Account struct {
        Nonce    uint64
        Balance  *big.Int
        Root     common.Hash // merkle root of the storage trie
        CodeHash []byte
}
```

4.3 关于 storageRoot 的补充

$$TRIE(LI(\boldsymbol{\sigma}[a]\mathbf{s})) \equiv \boldsymbol{\sigma}[a]\mathrm{s} \quad (6)$$

$$LI((k,v)) \equiv (KEC(k), RLP(v)) \quad (7)$$

$$k \in \mathrm{B32} \wedge v \in \mathrm{N} \quad (8)$$

公式 6，由于有些时候我们不仅需要 state 的 hash 值的 trie，而是需要其对应的 kv 数据也包含其中。所以 New era 中的存储 State 的树，不仅包含 State 的 hash，同时也包含了存储这个账户的 address 的 hash 和它对应的 data 也就是其 Account 的值的数据对的集合。这里 storageRoot 实际上是这样的树的根节点 hash 值。

公式 7，指 state 对应的 kv 数据的 RLP 的形式化表示，是 k 的 hash 值作为 key，value 是 v 的 RLP 表示。也就是 New era 中实际存储的 state 是账户 address 的 hash ($KEC(k)$) ,与其数据 Account 内容的 RLP ($RLP(v)$)。

公式 8，指公式 7 中的 k 是 32 的字符数组。这个是由 KECCAK256 算法保证的。

4.4 一些符号化定义

New era 中的账户有两类，一类是外部账户，一类是合约账户。其中外部账户被私钥控制且没有任何代码与之关联。合约账户，被它们的合约代码控制且有代码与之关联。以下几个公式定义了账户的各种状态。

$$LS(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \quad (9)$$

其中

$$p(a) \equiv (KEC(a), RLP((\boldsymbol{\sigma}[a]\mathrm{n}, \boldsymbol{\sigma}[a]\mathrm{b}, \boldsymbol{\sigma}[a]\mathrm{s}, \boldsymbol{\sigma}[a]\mathrm{c}))) \quad (10)$$

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \vee (a \in \mathrm{B}20 \wedge \nu(\boldsymbol{\sigma}[a])) \quad (11)$$

$$\nu(x) \equiv x\mathrm{n} \in \mathrm{N}256 \wedge x\mathrm{b} \in \mathrm{N}256 \wedge x\mathrm{s} \in \mathrm{B}32 \wedge x\mathrm{c} \in \mathrm{B}32 \quad (12)$$

$$\mathrm{EMPTY}(\boldsymbol{\sigma}, a) \equiv \boldsymbol{\sigma}[a]\mathrm{c} = KEC(()) \wedge \boldsymbol{\sigma}[a]\mathrm{n} = 0 \wedge \boldsymbol{\sigma}[a]\mathrm{b} = 0 \quad (13)$$

$$\mathrm{DEAD}(\boldsymbol{\sigma}, a) \equiv \boldsymbol{\sigma}[a] = \varnothing \vee \mathrm{EMPTY}(\boldsymbol{\sigma}, a) \quad (14)$$

公式 9，定义了函数 *LS*，意思是若账户 *a* 不为空，则返回账户 $p(a)$。

公式 10，定义 $p(a)$,$p(a)$其实就是我们上面公式 7，解释 *k*，*v* 对的时候的 kv 对。包括 address 的 hash 值，以及 Account 内容的 RLP 结果。

公式 11 与公式 12，对账户 *a* 做了定义，表示账户要么为空，要么就是一个 a 为 20 个长度的字符，其 nonce 值为小于 2^{256} 的正整数，balance 值为小于 2^{256} 的正整数，storageRoot 为 32 位的字符，codeHash 为 32 的字符。

公式 13，定义了空账户。若一个账户，其地址为空字符，并且该账户 nonce 值为 0，balance 值也为 0.

公式 14，定义了无效账户，无效账户要么为空∅，要么是一个 EMPTY 账户。

5、交易

5.1 交易的步骤

- nonce: 与发送该交易的账户的 nonce 值一致。用 T_{n} 表示。
- gasPrice: 表示每 gas 的单价为多少 wei。用 T_{p} 表示。
- gasLimit:执行该条交易最大被允许使用的 gas 数目。用 T_{g} 表示。
- to:160 位的接受者地址。当交易位创建合约时，该值位空。用 T_{t} 表示。
- value:表示发送者发送的 wei 的数目。该值为向接受者转移的 wei 的数目，或者是创建合约时作为合约账户的初始 wei 数目。用 T_{v} 表示。
- v,r,s: 交易的签名信息，用以决定交易的发送者。分别用 $T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}$ 表示。

- init:如果是创建合约的交易,则 init 表示一段不限长度的 EVM-Code 用以合约账户初始化的过程。用 T_i 表示。
- data: 调用合约的交易，会包含一段不限长度的输入信息，用 T_d 表示。

5.2 代码示例

```
type Transaction struct {
        data txdata
        // caches
        hash atomic.Value
        size atomic.Value
        from atomic.Value
}
type txdata struct {
        AccountNonce uint64          `json:"nonce"
gencodec:"required"`
        Price        *big.Int        `json:"gasPrice"
gencodec:"required"`
        GasLimit     uint64          `json:"gas"
gencodec:"required"`
```

```
        Recipient     *common.Address `json:"to"        rlp:"nil"` // nil
means contract creation
        Amount        *big.Int        `json:"value"
gencodec:"required"`
        Payload       []byte          `json:"input"
gencodec:"required"`

        // Signature values
        V *big.Int `json:"v" gencodec:"required"`
        R *big.Int `json:"r" gencodec:"required"`
        S *big.Int `json:"s" gencodec:"required"`

        // This is only used when marshaling to JSON.
        Hash *common.Hash `json:"hash" rlp:"-"`
}
```

5.3 一些符号化定义

New era 中根据交易中的 to 值是否为空，可以判断交易是创建合约还是执行合约。

$$LT(T)\equiv\{(Tn,Tp,Tg,Tt,Tv,Ti,Tw,Tr,Ts)(Tn,Tp,Tg,Tt,Tv,Td,Tw,Tr,Ts)ifTt=\varnothing otherwise(15)$$
$$Tn\in N256Tg\in N256Ts\in N256\wedge\wedge\wedge Tv\in N256Tw\in N5Td\in B\wedge\wedge\wedge Tp\in N256Tr\in N256Ti\in B\wedge\wedge(16)$$

$$Nn=\{P:P\in N\wedge P<2n\}(17)$$

$$Tt\in\{B20B0ifTt\neq\varnothing otherwise(18)$$

公式 15 表示，如果 to 的值 Tt 为空，则交易是创建合约的交易，需要有 init 数据。交易的 RLP 形式化可以表示为 nonce Tn，gasPrice Tp, gasLimit Tg, to Tt，value Tv，**init** Ti，“v，r，s” Tw, Tr, Ts。如果 Tt 不为空，则交易是执行合约的交易，需要有 data 数据。交易的 RLP 形式化可以表示为 nonce Tn，gasPrice Tp, gasLimit Tg, to Tt，value Tv，**data** Td，“v，r，s” Tw, Tr, Ts。

公式 16，是对交易的各个字段限制的符号化定义。其意思是 nonce、value、gasPrice、gasLimit 以及特殊的用来验证签名的 r 和 s 都是小于 2^{256} 的正整数，用来验证签名的 v (Tw)是小于 2^5 的正整数。而 init 和 data 都是未知长度的字符数组。

公式 17，是对 Nn 的定义，即小于 2^n 的正整数。

公式 18，是对交易中的 to 字段的符号化定义，当其不为空的时候，是 20 位的字符，为空的时候是 0 位字符。

6、区块

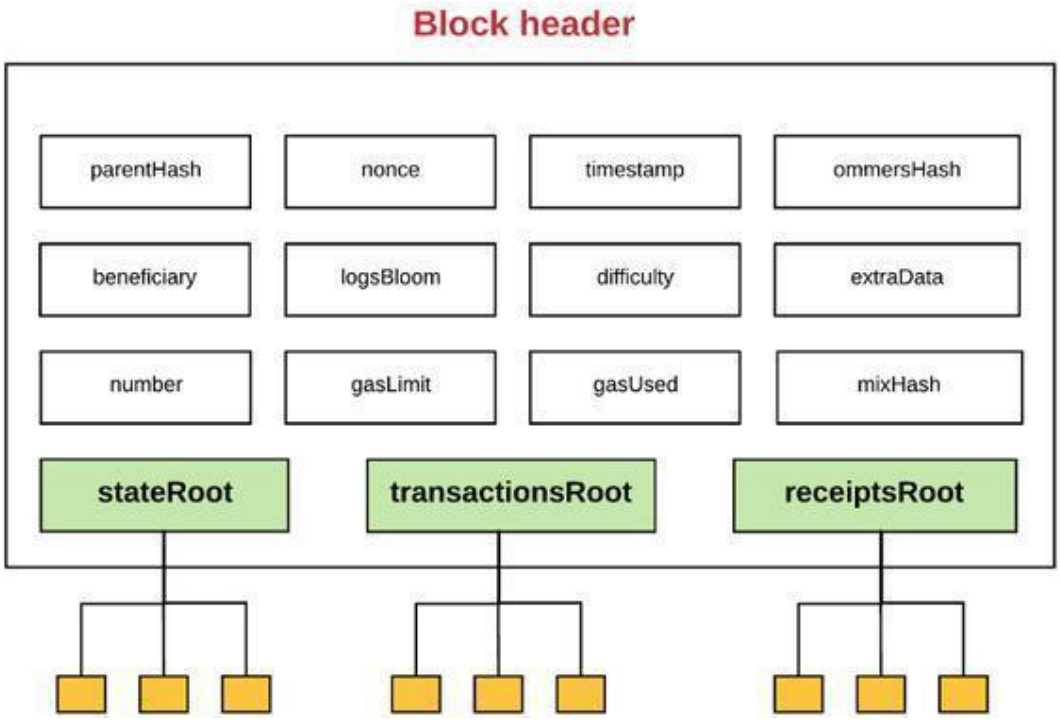
New era 中的一个区块由区块头 Header，以及交易列表 BT,以及 ommerblock 的 header 集合 BU 三部分组成。

6.1 区块头讲解

Header 包括以下字段。

- parentHash: 父节点的 hash 值。用 H_p 表示。
- ommersHash: uncle 节点的 hash 值，这块是跟 GHOST 相关的，用 H_o 表示。
- beneficiary: 矿工 address，用 H_c 表示。
- stateRoot: 当所有交易都执行完毕后的世界状态树的根节点，用 H_r 表示。
- transactionsRoot:交易列表的根节点，用 H_t 表示。
- receiptsRoot:收据的根节点，用 H_e 表示。
- logsBloom:日志过滤器，用 H_b 表示。*这个暂时没细看，不太确定。*
- difficulty:区块难度，根据上一个区块的难度以及时间戳算出来的值，用 H_d 表示。
- number:区块号，用 H_i 表示。
- gasLimit: 区块的 gas 数量限制，即区块中交易使用掉的 gas 值不应该超过该值。用 H_l 表示。
- gasUsed: 区块使用掉的 gas 数量，用 H_g 表示。
- timestamp:时间戳，用 H_s 表示。

- extraData:额外的数据，合法的交易对长度有限制，用 H_x 表示。
- mixHash: 与 nonce 一起用作工作量证明，用 H_m 表示。
- nonce:与 mixHash 一起用作工作量证明，用 H_n 表示。



6.2 区块头结构代码

```
type extblock struct {
        Header *Header
        Txs    []*Transaction
```

```
        Uncles []*Header
    }

// Header represents a block header in the Ethereum blockchain.
type Header struct {
        ParentHash  common.Hash    `json:"parentHash"
gencodec:"required"`
        UncleHash   common.Hash    `json:"sha3Uncles"
gencodec:"required"`
        Coinbase    common.Address `json:"miner"
gencodec:"required"`
        Root        common.Hash    `json:"stateRoot"
gencodec:"required"`
        TxHash      common.Hash    `json:"transactionsRoot"
gencodec:"required"`
        ReceiptHash common.Hash    `json:"receiptsRoot"
gencodec:"required"`
        Bloom       Bloom          `json:"logsBloom"
gencodec:"required"`
        Difficulty  *big.Int       `json:"difficulty"
gencodec:"required"`
```

```
        Number      *big.Int        `json:"number"
gencodec:"required"`
        GasLimit    uint64          `json:"gasLimit"
gencodec:"required"`
        GasUsed     uint64          `json:"gasUsed"
gencodec:"required"`
        Time        *big.Int        `json:"timestamp"
gencodec:"required"`
        Extra       []byte          `json:"extraData"
gencodec:"required"`
        MixDigest   common.Hash     `json:"mixHash"
gencodec:"required"`
        Nonce       BlockNonce      `json:"nonce"
gencodec:"required"`
    }
// Receipt represents the results of a transaction.
type Receipt struct {
        // Consensus fields
        PostState         []byte `json:"root"`
        Status            uint   `json:"status"`
        CumulativeGasUsed uint64 `json:"cumulativeGasUsed"
gencodec:"required"`
```

```
        Bloom           Bloom   `json:"logsBloom"
gencodec:"required"`
        Logs            []*Log `json:"logs"
gencodec:"required"`

        // Implementation fields (don't reorder!)
        TxHash          common.Hash      `json:"transactionHash"
gencodec:"required"`
        ContractAddress common.Address `json:"contractAddress"`
        GasUsed         uint64          `json:"gasUsed"
gencodec:"required"`
    }
```

区块的符号化定义 $B\equiv(B_H,B_T,B_U)(19)$ 公式 19，表示区块由三部分组成，区块头 B_H，交易列表 B_T,以及 ommerblock 的 header 集合 B_U。

6.3 交易收据

New era 中为了将交易的信息进行编码，以方便索引以及查找或者零知识证明等相关的东西，为每条交易定义了一定收据。对于第 i 个交易，其收据用 B_R[i]表示。 每条收据都是一个四元组，包括区块

当前累计使用的 gas 值 R_{u}，交易执行产生的 log R_{l}，日志过滤器 R_{b}，以及状态码 R_{z}。

$$R \equiv (R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathrm{l}}, R_{\mathrm{z}}) \quad (20)$$

$$L_R(R) \equiv (0 \in \mathbb{B}_{256}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathrm{l}}) \quad (21)$$

$$R_{\mathrm{z}} \in \mathbb{N} \quad (22)$$

$$R_{\mathrm{u}} \in \mathbb{N} \wedge R_{\mathrm{b}} \in \mathbb{B}_{256} \quad (23)$$

$$O \equiv (O_{\mathrm{a}}, (O_{\mathrm{t}0}, O_{\mathrm{t}1}, \ldots), O_{\mathrm{d}}) \quad (24)$$

$$O_{\mathrm{a}} \in \mathbb{B}_{20} \wedge \forall t \in O_{\mathrm{t}} : t \in \mathbb{B}_{32} \wedge O_{\mathrm{d}} \in \mathbb{B} \quad (25)$$

公式 20 对区块头中的收据作了定义，收据是个四元组，四元组定义如前文。

公式 21，收据的 RLP 形式化表示为 $L_R(R)$。其中 $0 \in \mathbb{B}_{256}$ 在之前版本的协议中是交易执行之前的 stateRoot。现在被替换为 0.

公式 22，表示 R_{z} 状态码是正整数。

公式 23 对收据中当前累计使用的 gas 值 R_{u}，和日志过滤器 R_{b} 进行了描述。显然累计 gas 值 R_{u} 是一个正整数。而日志过滤器 R_{b} 是 256 位字符。

公式 24 对交易执行的日志 R_{l} 进行了解释。

公式 25 对其限制进行了描述。R_{l} 是日志条目的序列。日志条目需要包括纪录日志者的地址，以及日志话题分类，以及实际数据。日志条

目用 O 来表示，用 O_a 表示日志纪录者的 address，用 O_t 来表示一些列 32 位字符的日志主题(log topics),用 O_d 来表示字符数据。其中日志纪录者的 address O_a 是 20 位字符，每一个日志分类话题 O_t 是一个 32 位字符，而日志数据 O_d 是未知长度的字符。

公式 26-30.对日志过滤函数做了定义，这块涉及到东西是数据操作层面的，不影响对流程的理解。暂不作解释。

7、交易的执行

交易执行是 New era 中最为重要的部分。

在执行交易之前首先需要对交易进行初步校验:

7.1 交易执行的校验

- 交易是 RLP 格式的，无多余字符
- 交易的签名是有效的
- 交易的 nonce 是有效的（与发送者账户的 nonce 值一致）
- gasLimit 的值不小于固有 gas g_0

- 账户余额至少够支付预付费用 $v0$ 当交易满足上述条件后，交易才会被执行。

7.2 交易结构校验代码

```
    // preCheck 校验的后三条。
    //交易校验的前两条是在其他地方执行的。对于矿工来说交易签名在加
txpool 的时候会检查，在 commitTransactions 的时候也会检查。
    func (st *StateTransition) preCheck() error {
        // Make sure this transaction's nonce is correct.
        // 检查 nonce 值
        if st.msg.CheckNonce() {
                nonce := st.state.GetNonce(st.msg.From())
                if nonce < st.msg.Nonce() {
                        return ErrNonceTooHigh
                } else if nonce > st.msg.Nonce() {
                        return ErrNonceTooLow
                }
        }
        return st.buyGas()
    }
```

```
func (st *StateTransition) buyGas() error {
        //gasLimit*gasPrice 即为预付的费用 v0
        mgval := new(big.Int).Mul(new(big.Int).SetUint64(st.msg.Gas()),
st.gasPrice)
        if st.state.GetBalance(st.msg.From()).Cmp(mgval) < 0 {
                return errInsufficientBalanceForGas
        }
        if err := st.gp.SubGas(st.msg.Gas()); err != nil {
                return err
        }
        st.gas += st.msg.Gas()
        //initialGas
        st.initialGas = st.msg.Gas()
        st.state.SubBalance(st.msg.From(), mgval)
        return nil
}
```

7.3 交易的形式化表示

$$\boldsymbol{\sigma}'=\Upsilon(\boldsymbol{\sigma},T) \quad (51)$$

公式 51，是对交易的形式化定义。交易的执行，相当于当前状态 σ 和交易 T,通过交易转变函数 Υ,到达新的状态 σ'。

7.4 子状态

在交易执行的整个过程中，New era 保持跟踪“子状态”。子状态是纪录交易中生成信息的一种方式，当交易完成时会立即需要这些信息。 交易的子状态包含：

- 自毁集合（self-destruct set），用 A_s 表示，指在交易完成之后需要被销毁的账户集合。
- 日志序列 (log series)，用 A_l 表示，指虚拟机代码执行的归档的和可检索的检查点。
- 账户集合（touched accounts），用 A_t 表示，其中空的账户在交易结束时将被删除。
- 退款余额（refund balance），用 A_r，指在交易完成之后需要退还给发送账户的总额。

7.5 子状态的形式化表示

$$A \equiv (A_s, A_l, A_t, A_r) \quad (52)$$

$$A_0 \equiv (\emptyset, (), \emptyset, 0) \quad (53)$$

公式 52，是交易子状态的形式化表示.

公式 53，定义了空的子状态 A_0.

计算预付交易费的形式化表示

$$v_0 \equiv T_g T_p + T_v \quad (57)$$ 公式 57

表示预付费用 v_0 的计算。表示预付费用为 gasLimit*gasPrice+value。实际代码中如上文 buyGas.

7.6 执行

- 对交易进行初步检查，从发送者账户中扣除预付的交易费。预付交易费值如公式 57 所示，为 gasLimit*gasPrice + value。（代码中是 gasLimit*gasPrice,Value 是在 Call 的过程中判断和扣除的）
- 计算固有 gas 消耗 g_0，如公式 54-56 所示。并消耗掉该花费。
- 如果是创建合约，则走合约创建流程。消耗相应花费。
- 如果是合约执行，则走合约执行流程。消耗相应花费。
- 计算退款余额，将余额退还到发送者账户。
- 将交易的交易费加到矿工账户。
- 返回当前状态，以及交易的花费。

7.7 执行过程中的数据结构

```
    //gp 中一开始有 gasLimit 数量的 gas
    type StateTransition struct {
        gp          *GasPool
        msg         Message
        gas         uint64
        gasPrice    *big.Int
        initialGas uint64
        value       *big.Int
        data        []byte
        state       vm.StateDB
        evm         *vm.EVM
}
```

```
    // TransitionDb will transition the state by applying the current
message and
    // returning the result including the the used gas. It returns an error
if it
    // failed. An error indicates a consensus issue.
```

```
    func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64,
failed bool, err error) {
        //交易检查，检查正确的话，st.gas 为 gasPrice*gasLimit，即预付的
交易费。
        if err = st.preCheck(); err != nil {
            return
        }
        msg := st.msg
        sender := vm.AccountRef(msg.From())
        homestead :=
st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
        contractCreation := msg.To() == nil

        // Pay intrinsic gas
        // 固有 gas，也就是 g0
        gas, err := IntrinsicGas(st.data, contractCreation, homestead)
        if err != nil {
            return nil, 0, false, err
        }
        if err = st.useGas(gas); err != nil {
            return nil, 0, false, err
        }
```

```
        var (
                evm = st.evm
                // vm errors do not effect consensus and are therefor
                // not assigned to err, except for insufficient balance
                // error.
                vmerr error
        )
        //创建合约
        if contractCreation {
                ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas,
st.value)
        } else {
                // Increment the nonce for the next transaction
                //执行合约
                st.state.SetNonce(msg.From(),
st.state.GetNonce(sender.Address())+1)
                ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data,
st.gas, st.value)
        }
        if vmerr != nil {
                log.Debug("VM returned with error", "err", vmerr)
```

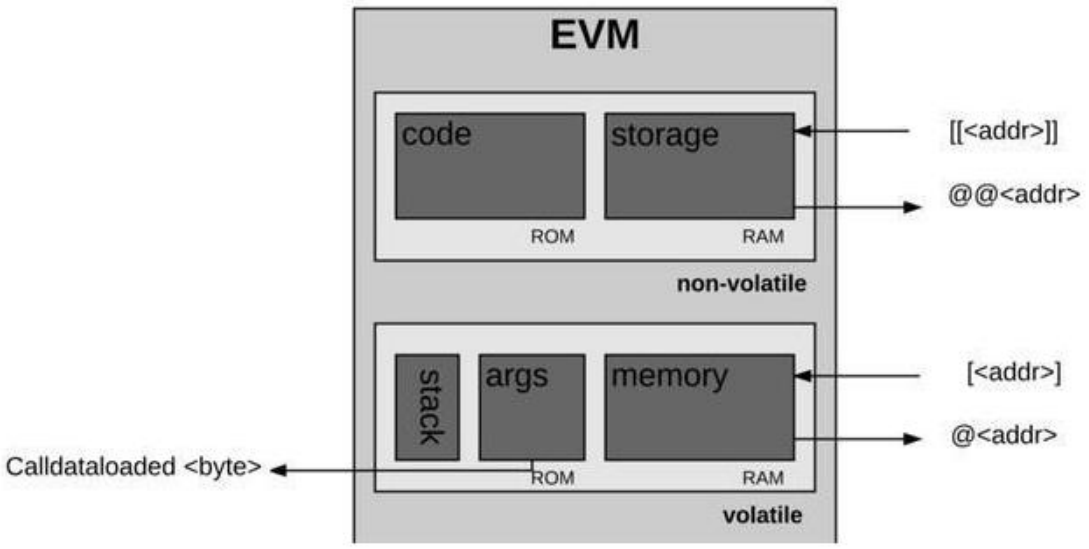
```
                // The only possible consensus-error would be if there
wasn't
                // sufficient balance to make the transfer happen. The
first
                // balance transfer may never fail.
                if vmerr == vm.ErrInsufficientBalance {
                        return nil, 0, false, vmerr
                }
        }
        //计算退款，并返回到发送者账户
        st.refundGas()
        //付交易费给矿工
        st.state.AddBalance(st.evm.Coinbase,
new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.gasPrice))

        return ret, st.gasUsed(), vmerr != nil, err
    }
```

8、虚拟机

New era 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。

8.1 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 New era 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。

8.2 虚拟机结构代码

```
// 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 needed because the
        // available gas is calculated in gasCall* according to the 63/64
rule and later
        // 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
```

8.3 虚拟机执行费用

New era 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。

8.4 虚拟机的运行环境

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 New era。
- 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）

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

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

9、超级节点

超级节点是 New era 在以太坊基础上的一个创新点，通过在各大洲选择链

路供应商，由链路供应商提供数据基站，数据供应商可以根据自己的出入口流量获得为用户提供高速网路的服务费。

9.1 超级节点状态数学公式

$$P=U(P1+P2+P3+P4+P5)$$

其中 P1,P2,P3,P4,P5 为五个超级节点的数据状态，U 为最新链长度仲裁变量，P 为链最新的数据状态。

10、链桥

链桥是 New era 的另一个创新技术，通过链桥，用户可以方便的将 New era 上的数据和以太坊公链上的数据进行双向迁移。

链桥主要由这三个模块组成，这三个模块对应的数据公式如下：

10.1 链桥的基础数学公式：

$$T=D\Leftrightarrow d$$

其中 T 为内部的数据链装，D New era 数据链的状态 d 为以太坊数据链的状态。

10.2 将 New era 数据写入公链数学公式：

$$D=M(d) \Rightarrow t1$$

其中 D 为 New era 上申请同步的数据集，M 为链桥链接函数，d 为外部链最新过程，t 为写出成功后的状态值

10.3 将公链数据同步到 New era 数学公式：

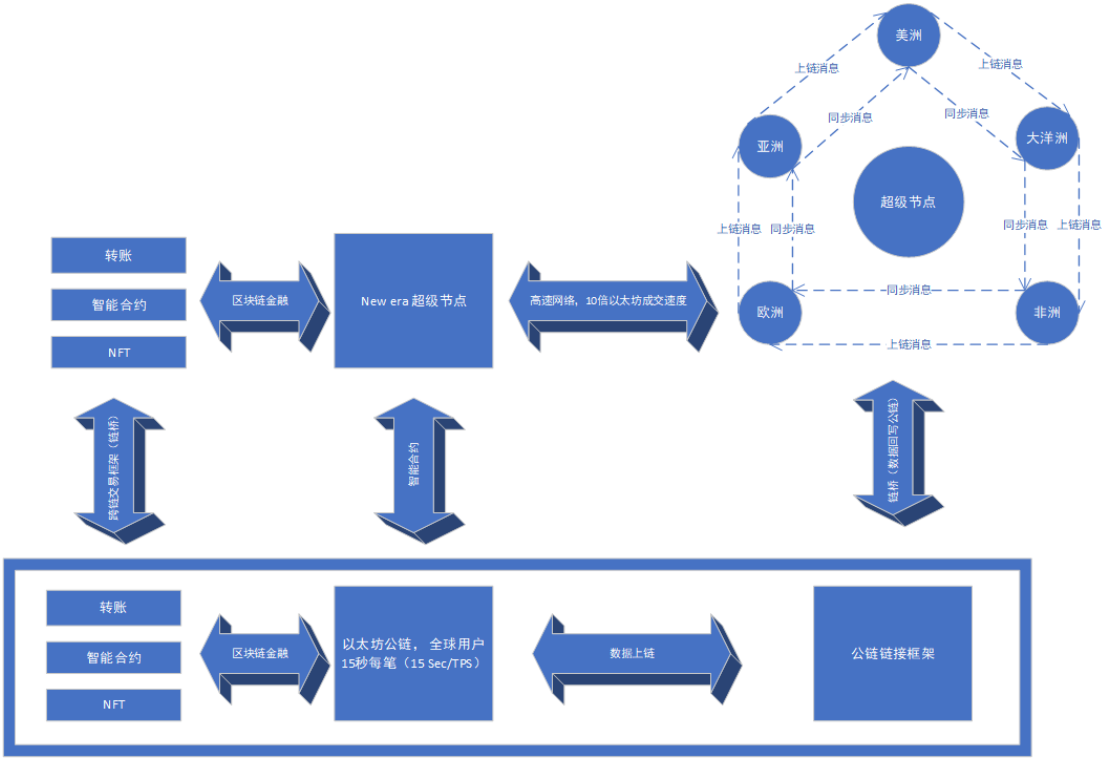
$$D=M(\mathbf{S}(d))=>t2$$

公式中 D 为最终写入状态，M 为链桥函数，S 为用户使用智能合约在 New era 上的开户账户，d 为需要同步的数据。

11、New era 架构示意图

New era 主要由数据链，超级节点和链桥这三部分组成

组成的示意图如下：



示意图说明：

基础数据层是以太坊的公链系统，每秒最大 15/Tps，上面部分是 New era 的高速链路系统，成交速度是以太坊的十倍。

New era 主要通过超级节点进行数据的提速，用户在以太坊上

的操作在 New era 上都能完成，并在完成后，可以通过链桥系统将数据结果回写到以太坊的公链上。

12、结论

我们已经介绍、讨论并正式定义了 New era 的协议。通过这个协议，读者可以在 New era 网络上申请一个节点并加入大家，成为一个去中心化的安全的社会化操作系统中的一员。合约可以被创作出来，在算法上指定并自主化地执行交互规则。

12.1、黄皮书参考资料

12.1.1 术语表：

Appendix A. Terminology

External Actor: A person or other entity able to interface to an Ethereum node, but external to the world of

Ethereum. It can interact with Ethereum through depositing signed Transactions and inspecting the blockchain

and associated state. Has one (or more) intrinsic Accounts.

以太坊：一种安全去中心化的通用交易账本 拜占庭版本 f72032b - 2018-05-04 14

Address: A 160-bit code used for identifying Accounts.

Account: Accounts have an intrinsic balance and transaction count maintained as part of the Ethereum state.

They also have some (possibly empty) EVM Code and a (possibly empty) Storage State associated with them.

Though homogenous, it makes sense to distinguish between two practical types of account: those with empty

associated EVM Code (thus the account balance is controlled, if at all, by some external entity) and those with

non-empty associated EVM Code (thus the account represents an Autonomous Object). Each Account has a

single Address that identifies it.

Transaction: A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous

Object. Transactions are recorded into each block of the blockchain.

Autonomous Object: A notional object existent only within the hypothetical state of Ethereum. Has an intrinsic

address and thus an associated account; the account will have non-empty associated EVM Code. Incorporated

only as the Storage State of that account.

Storage State: The information particular to a given Account that is maintained between the times that the

Account’ s associated EVM Code runs.

Message: Data (as a set of bytes) and Value (specified as Ether) that is passed between two Accounts, either

through the deterministic operation of an Autonomous Object or the cryptographically secure signature of the

Transaction.

Message Call: The act of passing a message from one Account to another. If the destination account is associated

with non-empty EVM Code, then the VM will be started with the state of said Object and the Message acted

upon. If the message sender is an Autonomous Object, then the Call passes any data returned from the VM

operation.

Gas: The fundamental network cost unit. Paid for exclusively by Ether (as of PoC-4), which is converted freely

to and from Gas as required. Gas does not exist outside of the internal Ethereum computation engine; its price

is set by the Transaction and miners are free to ignore Transactions whose Gas price is too low.

Contract: Informal term used to mean both a piece of EVM Code that may be

associated with an Account or an

Autonomous Object.

Object: Synonym for Autonomous Object.

App: An end-user-visible application hosted in the Ethereum Browser.

Ethereum Browser: (aka Ethereum Reference Client) A cross-platform GUI of an interface similar to a simplified

browser (a la Chrome) that is able to host sandboxed applications whose backend is purely on the Ethereum

protocol.

Ethereum Virtual Machine: (aka EVM) The virtual machine that forms the key part of the execution model

for an Account’ s associated EVM Code.

Ethereum Runtime Environment: (aka ERE) The environment which is provided to an Autonomous Object

executing in the EVM. Includes the EVM but also the structure of the world state on which the EVM relies for

certain I/O instructions including CALL & CREATE.

EVM Code: The bytecode that the EVM can natively execute. Used to formally specify the meaning and

ramifications of a message to an Account.

EVM Assembly: The human-readable form of EVM-code.

LLL: The Lisp-like Low-level Language, a human-writable language used for

authoring simple contracts and general

low-level language toolkit for trans-compiling to.

12.1.2 参考文献：

References

Pierre Arnaud, Mathieu Schroeter, and Sam Le Barbare. Electrum, 2017. URL https://www.npmjs.com/

package/electrum.

Jacob Aron. BitCoin software finds new life.

New Scientist, 213(2847):20, 2012. URL

http://www.sciencedirect.com/science/article/

pii/S0262407912601055.

Adam Back. Hashcash - Amortizable Publicly Auditable

Cost-Functions, 2002. URL http://www.hashcash.

org/papers/amortizable.pdf.

Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van

Assche, and Ronny Van Keer. KECCAK, 2017. URL

https://keccak.team/keccak.html.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers, Innovators and Imitators.

In Growth Through

Innovation, pages 85–96. Springer, 2014. URL https:

//www.springer.com/gb/book/9783319040158.

Vitalik Buterin. Ethereum: A Next-Generation Smart

Contract and Decentralized Application Platform,

2013a. URL https://github.com/ethereum/wiki/

wiki/White-Paper.

Vitalik Buterin. Dagger: A Memory-Hard to Compute,

Memory-Easy to Verify Scrypt Alternative, 2013b. URL

http://www.hashcash.org/papers/dagger.html.

Vitalik Buterin. EIP-2: Homestead hard-fork changes,

2015. URL https://github.com/ethereum/EIPs/

blob/master/EIPS/eip-2.md.

Vitalik Buterin. EIP-100: Change difficulty adjustment to target mean block time including uncles,

April 2016. URL https://github.com/ethereum/EIPs/

blob/master/EIPS/eip-100.md.

Nicolas T. Courtois, Marek Grajek, and Rahul Naik.

Optimizing SHA256 in Bitcoin Mining, pages 131–

144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-44893-9.

doi: 10.

1007/978-3-662-44893-9_12. URL https://doi.org/

10.1007/978-3-662-44893-9_12.

B.A. Davey and H.A. Priestley. Introduction to lattices
and order. 2nd ed. Cambridge: Cambridge University
Press, 2nd ed. edition, 2002. ISBN 0-521-78451-4/pbk.

Thaddeus Dryja. Hashimoto: I/O bound proof of
work, 2014. URL http://diyhpl.us/~bryan/papers2/
bitcoin/meh/hashimoto.pdf.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In In
12th Annual International Cryptology Conference, pages 139–
147, 1992. URL http://www.wisdom.weizmann.ac.il/
~naor/PAPERS/pvp.pdf.

Phong Vo Glenn Fowler, Landon Curt Noll. Fowler–Noll–
Vo hash function, 1991. URL http://www.isthe.com/
chongo/tech/comp/fnv/index.html.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle,
and Sheueling Chang Shantz. Comparing elliptic curve
cryptography and RSA on 8-bit CPUs. In Cryptographic
Hardware and Embedded Systems-CHES 2004, pages
119–132. Springer, 2004. URL https://www.iacr.org/
archive/ches2004/31560117/31560117.pdf.

Christoph Jentzsch. Commit date for ethash, 2015. URL
https://github.com/ethereum/yellowpaper/commit/

77a8cf2428ce245bf6e2c39c5e652ba58a278666#

commitcomment-24644869.

Don Johnson, Alfred Menezes, and Scott Vanstone.

The Elliptic Curve Digital Signature Algorithm

(ECDSA), 2001. URL https://web.archive.org/web/

20170921160141/http://cs.ucsb.edu/~koc/ccs130h/

notes/ecdsa-cert.pdf. Accessed 21 September 2017,

but the original link was inaccessible on 19 October

2017. Refer to section 6.2 for ECDSAPUBKEY, and

section 7 for ECDSASIGN and ECDSARECOVER.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions, 2014. URL http://www.hashcash.org/papers/

memohash.pdf.

Mark Miller. The Future of Law. In paper delivered at the Extro 3 Conference (August 9), 1997. URL https://drive.google.com/file/d/

0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash

system, 2008. URL http://www.bitcoin.org/bitcoin.

pdf.

Meni Rosenfeld, Yoni Assia, Vitalik Buterin, m liorhakiLior, Oded Leiba, Assaf Shomer, and Eliran Zach. Colored Coins Protocol Specification,

2012. URL https://github.com/Colored-Coins/

Colored-Coins-Protocol-Specification.

Afri Schoedon and Vitalik Buterin. EIP-649: Metropolis difficulty bomb delay and block reward reduction,

June 2017. URL https://github.com/ethereum/EIPs/

blob/master/EIPS/eip-649.md.

Michael John Sebastian Smith. Application-Specific

Integrated Circuits. Addison-Wesley, 1997. ISBN

0201500221.

Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin' s transaction processing. fast money grows on trees,

not chains, 2013. URL https://eprint.iacr.org/

2013/881.

Simon Sprankel. Technical Basis of Digital Currencies,

2013. URL http://www.coderblog.de/wp-content/

uploads/technical-basis-of-digital-currencies.

pdf.

Nick Szabo. Formalizing and securing relationships on public networks. First Monday, 2(9),

1997. URL http://firstmonday.org/ojs/index.php/

fm/article/view/548.

Vivek Vishnumurthy, Sangeeth Chandrakumar, and

Emin Gün Sirer. KARMA: A secure economic framework for peer-to-peer resource

sharing, 2003. URL

https://www.cs.cornell.edu/people/egs/papers/

karma.pdf.

J. R. Willett. MasterCoin Complete Specification, 2013.

URL https://github.com/mastercoin-MSC/spec.