# 以太坊黄皮书详解（二）

2018-06-06 Ethereum

## 三、交易执行

交易执行是以太坊中最为重要的部分。

在执行交易之前首先需要对交易进行初步校验:

- 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v_0 当交易满足上述条件后，交易才会被执行。

```
// 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commitTrans
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
```

```
        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
}
```

**交易的形式化表示** 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\sigma和交易T,通过交易转变函数\Upsilon,到达新的状态{\sigma}'。

## 3.1 子状态

在交易执行的整个过程中，以太坊保持跟踪“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。

**子状态的形式化表示**

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.

## 3.2 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）

- 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。

```
/*
The State Transitioning Model

A state transition is a change made when a transaction is applied to the current w
The state transitioning model does all all the necessary work to work out a valid

1) Nonce handling
2) Pre pay gas
3) Create a new state object if the recipient is \0*32
4) Value transfer
== If contract creation ==
  4a) Attempt to run transaction data
  4b) If valid, use result as code for the new state object
== end ==
5) Run Script section
6) Derive new state root
*/
//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
        //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 {
```



```
        }
```

```
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.val
} else {
        // Increment the nonce for the next transaction
        //执行合约
        st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+
        ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st
}
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).SetUint
```

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

### 3.2.1 每一步的形式化表示

#### 计算固有gas消耗的形式化表示

公式54-56为固有gas消耗g_0的计算方式。

- 统计T_i,T_d，即交易的init和data字段，0和非0分开计算。为0的字节数 *TxDataZeroGas，非0字节数*TxDataNonZeroGas，并求和。
- 如果是创建合约，则加上创建合约的固定消耗。
- 如果是执行合约，加上合约执行的固定消耗。

```
// IntrinsicGas computes the 'intrinsic gas' for a message with the given data.
// 计算g0
func IntrinsicGas(data []byte, contractCreation, homestead bool) (uint64, error) {
        // Set the starting gas for the raw transaction
        var gas uint64
        //公式55和56
        if contractCreation && homestead {
                gas = params.TxGasContractCreation
        } else {
                gas = params.TxGas
        }
        //公式54，根据non-zero和zero data进行计算
        // Bump the required gas by the amount of transactional data
        if len(data) > 0 {
                // Zero and non-zero bytes are priced differently
                var nz uint64
                for _, byt := range data {
                        if byt != 0 {
                                nz++
                        }
                }
                // Make sure we don't exceed uint64 for all data combinations
                if (math.MaxUint64-gas)/params.TxDataNonZeroGas < nz {
                        return 0, vm.ErrOutOfGas
                }
                gas += nz * params.TxDataNonZeroGas

                z := uint64(len(data)) - nz
                if (math.MaxUint64-gas)/params.TxDataZeroGas < z {
```

```
                }
                gas += z * params.TxDataZeroGas
        }
        return gas, nil
}
```

**计算预付交易费的形式化表示** 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.

**交易初步校验的形式化表示** 公式58为交易的初步验证的形式化表示。第一和第二行表示发送者账户不为空，且存在。第三行表示交易的nonce值为发送者的当前nonce值。第四行表示固定消耗g_0小于等于gasLimit(否则连固定消耗都不够)。第五行表示当前账户余额必须足够支付预付费用。第六行表示当前区块的gasLimit-已经消耗掉的gas值大于等于交易的gasLimit。

**交易初始状态（虚拟机执行之前）的形式化表示**

公式59-61表示交易执行时的初始状态，该处也是一个检查点。用于后续操作中出错时的回滚。 交易开始执行时，会首先将发送者账户的balance减去预付gas（gasLimit*gasPrice），并将该账户的nonce加一。其他与原状态相同。

**虚拟机执行的形式化表示**

公式62表示

- 如果交易的to为空，则是合约创建交易，状态转变函数为\Lambda_{4}。
- 否则为合约执行，状态转变函数为\Theta_{4}。
- 两个转变的共同参数有：初始状态${\sigma}*{0}，发送者账户地址S(T),可用gas值g，gasPriceT_p,交易valueT_v，原始调用者T*{\mathrm{o}},0,和\top.其中可用gas值g为gasLimit-g_0，如公式63所示。原始调用者T_{\mathrm{o}}$,根据交易是创建合约还是执行合约会有所不同，不由交易控制，由虚拟机来控制。
- 接收账户T_t 经过虚拟机执行后状态从${\sigma}*{0}转变为{\sigma}*{P}，剩余的gas为g’$,交易子状态为A，交易的状态码为z。

**计算退款的形式化表示**

公式64表示交易执行之后的子状态的退款额A’为原先子状态退款额与此次需要销毁的自毁集合返回的退款的和。

Loading [MathJax]/jax/output/HTML-CSS/fonts/STIX/General/Regular/SuppMathOperators.js

公式65时交易执行后的退款额的计算，退款额为交易执行后剩余gas值g’加上，子状态退款额A’和用掉的gas（T_g-g’）的一半中较小的那个。

```
func (st *StateTransition) refundGas() {
        // Apply refund counter, capped to half of the used gas.
        // 用掉gas的一半，与子状态退款额比较
        refund := st.gasUsed() / 2
        if refund > st.state.GetRefund() {
                refund = st.state.GetRefund()
        }
        //剩余的总gas值
        st.gas += refund


        // Return ETH for remaining gas, exchanged at the original rate.
        remaining := new(big.Int).Mul(new(big.Int).SetUint64(st.gas), st.gasPrice)
        st.state.AddBalance(st.msg.From(), remaining)

        // Also return remaining gas to the block gas counter so it is
        // available for the next transaction.
        st.gp.AddGas(st.gas)
}
```

**交易之后状态的形式化表示** 公式66-69定义了交易之后的预结束状态（之所以说是预结束状态是这个时候一些需要销毁的状态尚未销毁），其和虚拟机执行后的状态{\sigma}_{P}的区别为：

- 将退款额返回给交易发送者账户。公式67。
- 支付交易费给矿工。公式68。其中m为矿工的地址。公式69。

公式70-72定义了，交易的结束状态。结束状态{\sigma}’与预结束状态的区别为，将自毁集合中的账户置为空，公式71，将可控的账户集合中的死掉的账户，置为空。公式72.

公式73-75，定义了交易的其他几个相关字段。其中公式73，定义了交易的花费为，gasLimit-退款。公式74定义了交易的日志序列即为交易子状态中的日志序列。公式75定义了交易的最终状态即为虚拟机执行后的状态码。

## 四、合约创建

第三部分于合约创建和合约执行的具体过程未详细介绍。该部分对合约创建的过程进行详细的解释。

以太坊中有两类账户，一类为外部拥有账户，即通常意义上的用户账户。一类为合约账户。当一个交易是合约创建，是指该交易的目的是创建一个新的合约账户。合约账户创建的过程如下：

- 根据规则生成合约账户地址。公式77.
- 设置合约账户nonce值为1，其balance设为捐献值，storageRoot设为空，codeHash为空的Hash。公式78–79.
- 当前账户余额中减去捐献值。公式80–81.
- 运行合约，进行合约的初始化工作。如果运行过程中gas不足，则所有状态回滚，消耗掉所有gas。也就是被创建的合约账户也会被回滚掉，捐献值回滚到原账户。
- 如果合约初始化运行成功，则计算存储code的花费，若成功，则设置账户的code。
- 如果不足以支付存储费用，回滚状态。（这个地方根据配置不同，homestead 和 byzantium会有所不同）

```
// Create creates a new contract using code as deployment code.
func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int

        // Depth check execution. Fail if we're trying to execute above the
        // limit.
        if evm.depth > int(params.CallCreateDepth) {
                return nil, common.Address{}, gas, ErrDepth
        }
        if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
                return nil, common.Address{}, gas, ErrInsufficientBalance
        }
        // Ensure there's no existing contract already at the designated address
        nonce := evm.StateDB.GetNonce(caller.Address())
        evm.StateDB.SetNonce(caller.Address(), nonce+1)
        //生成合约账户地址
        contractAddr = crypto.CreateAddress(caller.Address(), nonce)
        //若之前该合约账户对应的地址不空，则返回地址冲突的错误
        contractHash := evm.StateDB.GetCodeHash(contractAddr)
        if evm.StateDB.GetNonce(contractAddr) != 0 || (contractHash != (common.Has
                return nil, common.Address{}, 0, ErrContractAddressCollision
        }
        // Create a new account on the state
        snapshot := evm.StateDB.Snapshot()
        //创建合约账户
        evm.StateDB.CreateAccount(contractAddr)
        if evm.ChainConfig().IsEIP158(evm.BlockNumber) {
```

```
//将捐献值转移给合约账户
evm.Transfer(evm.StateDB, caller.Address(), contractAddr, value)

// initialise a new contract and set the code that is to be used by the
// EVM. The contract is a scoped environment for this execution context
// only.
contract := NewContract(caller, AccountRef(contractAddr), value, gas)
contract.SetCallCode(&contractAddr, crypto.Keccak256Hash(code), code)

if evm.vmConfig.NoRecursion && evm.depth > 0 {
        return nil, contractAddr, gas, nil
}

if evm.vmConfig.Debug && evm.depth == 0 {
        evm.vmConfig.Tracer.CaptureStart(caller.Address(), contractAddr, t
}
start := time.Now()

//运行合约，进行合约的初始化，错误交由最后处理
ret, err = run(evm, contract, nil)

// check whether the max code size has been exceeded
maxCodeSizeExceeded := evm.ChainConfig().IsEIP158(evm.BlockNumber) && len(
// if the contract creation ran successfully and no errors were returned
// calculate the gas required to store the code. If the code could not
// be stored due to not enough gas set an error and let it be handled
// by the error checking condition below.
//如果合约创建成功，无错误返回，则计算合约存储代码的花费。成功的话，设置合约账户的code。
if err == nil && !maxCodeSizeExceeded {
        createDataGas := uint64(len(ret)) * params.CreateDataGas
        if contract.UseGas(createDataGas) {
                evm.StateDB.SetCode(contractAddr, ret)
        } else {
                err = ErrCodeStoreOutOfGas
        }
}

// When an error was returned by the EVM or when setting the creation code
// above we revert to the snapshot and consume any gas remaining. Addition
// when we're in homestead this also counts for code storage gas errors.
// 如果不是ErrCodeStoreOutOfGas的话，revert当前状态，消耗gas。说明ErrCodeStoreOu
if maxCodeSizeExceeded || (err != nil && (evm.ChainConfig().IsHomestead(ev
        evm.StateDB.RevertToSnapshot(snapshot)
        if err != errExecutionReverted {
                contract.UseGas(contract.Gas)
```

```
        }
        // Assign err if contract code size exceeds the max while the err is still
        if maxCodeSizeExceeded && err == nil {
                err = errMaxCodeSizeExceeded
        }
        if evm.vmConfig.Debug && evm.depth == 0 {
                evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time.Since(s
        }
        return ret, contractAddr, contract.Gas, err
}
```

## 4.1 形式化表示

**合约创建流程的形式化表示** 公式76表示合约创建的形式化表示。

合约创建需要的参数有：系统状态\sigma,发送者（s），原始调用者（o），可用gas值（g），gas价格（p），捐献值（v），虚拟机的初始化代码其实际为一段任意长度的字节数组（i）,当前虚拟机调用的栈深度（e），以及权限控制列表（w）。

虚拟机执行合约创建的结果为新的中间过程状态集合{\sigma}’，剩余的gas值g’,交易子状态A，交易状态码z，合约的body code \mathbf{o}.

**新建合约账户地址的形式化表示** 公式77给出了合约账户的地址的计算方法，可以看出是跟发送者账户地址以及发送者的nonce值有关。

```
// CreateAddress creates an ethereum address given the bytes and the nonce
func CreateAddress(b common.Address, nonce uint64) common.Address {
        data, _ := rlp.EncodeToBytes([]interface{}{b, nonce})
        return common.BytesToAddress(Keccak256(data)[12:])
}
```

**合约账户初始化的形式化表示**

生成了合约账户的地址后，需要对账户进行相应的初始化。公式78–82给出了创建合约之后的相关状态。

公式79表示新建的合约账户，其nonce值为1，balance值为捐献值+原有值（如果原合约账户不为空，公式82），storage为空，codeHash为空的hash。

公式80–81表示如果调用账户若为空，则仍为空。若不为空，则相应的balance值减去捐献值v。

合约账户地址生成且初始化后，需要虚拟机执行合约的相应初始化代码。公式83给出了虚拟机执行的过程。

公式83表示，虚拟机在\sigma^*基础上执行，可用gas值为g，运行环境参数为I，s为合约创建的调用者地址，a为新生成的合约账户地址。 虚拟机执行之后生成新的临时状态$\sigma^**剩余的gas值g^**$,子状态A，以及状态码z。

其中I中的项如公式84-92所示。

- I_a为新生成的合约账户地址，即a；
- I_o为原始调用者，即o；
- I_p为gas价格，即p；
- I_d为虚拟机调用的input data，因为是合约创建，所以data段为空；
- I_s为发送者，或者说调用者，即s；
- I_v为捐献值，即v；
- I_b为初始化代码段，即i；
- I_e为当前调用栈深度，即e；
- I_w为权限管理列表，即w。

虚拟机执行过程中如果出现了gas值不足的情况，则会回滚所有的状态。状态回到调用合约创建开始时的状态，也就是调用过程中消耗掉了所有的gas值，但是合约账户被混滚掉，变成没有被创建的状态。如果有捐献值，捐献值回滚至原账户。

如果虚拟机执行初始化合约的操作成功，则需要存储相应的合约代码到新创建的合约账户。公式93时计算存储合约代码所需的花费，其值与合约的代码长度有关。

虚拟机执行之后根据执行的情况

- gas值，如果没有出错。当前的临时gas值为g^{**} – c。即在虚拟机执行完成后，再消耗掉存储code的gas。如果出错，则剩余gas值为0.公式94.
- 状态集合，如果出错，则回滚到虚拟机执行前的状态。若成功，则状态转变为临时状态集\sigma^{**}，然后进行一些后续处理，如果合约账户是个死账户，则将该账户置为空，否则对合约账户的code进行设置。
- 状态码，如果临时状态集为空（虚拟机运行初始化的时候就出错了），或者gas不足以支付存储code的费用，则状态码为0.没有出错则为1.
- 出错的情况有，1. 生成的临时状态集为空且code为空 2. gas不足以支付存储code的

### 4.2 特别提示

当初始化代码交由虚拟机执行的时候，新创建的合约账户地址是存在的，只不过是没有body code。因此任何调用该合约的代码都会因为没有可执行的代码而返回错误。如果初始化代码中是以自毁操作为结束的，那么合约账户在交易完成之前就会被删除掉，目前该问题还有争议。而对于正常的STOP代码，或者是初始化执行返回的代码为空，虚拟机执行完成后，判定到该合约账户的code也还是空，那么这个合约账户就会变成一个僵尸账户，而其余额也会被永久的冻结在里面。

## 五、合约调用

合约调用的流程如下：

- 如果to账户地址不存在，则新建.
- 从sender中转账value值到to账户.
- 从合约账户中获取合约代码，进行设置，供虚拟机执行.
- 虚拟机执行合约代码。
- 如果合约执行出错，则回滚到合约执行之前的状态。

```
// Call executes the contract associated with the addr with the given input as
// parameters. It also handles any necessary value transfer required and takes
// the necessary steps to create accounts and reverses the state in case of an
// execution error or failed value transfer.
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas ui
        if evm.vmConfig.NoRecursion && evm.depth > 0 {
                return nil, gas, nil
        }

        // Fail if we're trying to execute above the call depth limit
        if evm.depth > int(params.CallCreateDepth) {
                return nil, gas, ErrDepth
        }
        // Fail if we're trying to transfer more than the available balance
        if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
                return nil, gas, ErrInsufficientBalance
        }

        var (
                to       = AccountRef(addr)
                snapshot = evm.StateDB.Snapshot()
```



```
        //to账户不存在，则新建
```

```
        if !evm.StateDB.Exist(addr) {
                precompiles := PrecompiledContractsHomestead
                if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
                        precompiles = PrecompiledContractsByzantium
                }
                if precompiles[addr] == nil && evm.ChainConfig().IsEIP158(evm.Bloc
                        // Calling a non existing account, don't do antything, but
                        if evm.vmConfig.Debug && evm.depth == 0 {
                                evm.vmConfig.Tracer.CaptureStart(caller.Address(),
                                evm.vmConfig.Tracer.CaptureEnd(ret, 0, 0, nil)
                        }
                        return nil, gas, nil
                }
                evm.StateDB.CreateAccount(addr)
        }
        //转账
        evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)

        // Initialise a new contract and set the code that is to be used by the EV
        // The contract is a scoped environment for this execution context only.
        //设置要执行的代码
        contract := NewContract(caller, to, value, gas)
        contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.Get

        start := time.Now()

        // Capture the tracer start/end events in debug mode
        if evm.vmConfig.Debug && evm.depth == 0 {
                evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, false, in

                defer func() { // Lazy evaluation of the parameters
                        evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time
                }()
        }
        //执行代码
        ret, err = run(evm, contract, input)

        // When an error was returned by the EVM or when setting the creation code
        // above we revert to the snapshot and consume any gas remaining. Addition
        // when we're in homestead this also counts for code storage gas errors.
        if err != nil {
                evm.StateDB.RevertToSnapshot(snapshot)
                if err != errExecutionReverted {
                        contract.UseGas(contract.Gas)
                }
```

```
        return ret, contract.Gas, err
}
```

## 5.1 合约调用的形式化表示

公式98为合约调用的形式化表示。

合约调用需要的参数有：系统状态\sigma,发送者（s），原始调用者（o），收款人（r），合约账户地址（c），可用gas值（g），gas价格（p），转账额（v），捐献值（\tilde{v}）,虚拟机的执行代码的input data其实际为一段任意长度的字节数组（d）,当前虚拟机调用的栈深度（e），以及权限控制列表（w）。

虚拟机执行合约调用的结果为新的中间过程状态集合{\sigma}’，剩余的gas值g’,交易子状态A，交易状态码z，合约的调用结果\mathbf{o}.

**合约调用前的状态**

公式99–105为虚拟机执行合约代码之前的一些临时状态。

在虚拟机执行合约代码之前，首先进行转账。（除非发送者和接收者相同）公式99.

由于调用者有可能是未定义的。所以更严谨的定义如公式100–105.

- 调用者如果之前为空且value为0，则调用者依然为空。否则，调用者的balance减去转账值。公式100–102.
- 如果接收者账户不存在，且转账值不为0，则新建接收者，将其nonce值0，余额为转账额v，storageRoot为空的TRIE，codeHash为空的hash。
- 如果接收者账户不存在，且转账值为0.则不处理。
- 如果接收者账户存在，则其余额为原先的余额加上转账值。

**虚拟机执行合约调用的形式化表示**

虚拟机执行的参数为，公式109–118：

- I_a为接收者账户，即r，（在合约创建的时候，该处为新创建的合约账户）；
- I_o为原始调用者，即o;
- I_p为gasPric，即p;
- I_d为输入数据（input data），即d;



- I_s为调用者账户，即s;

- l_v为捐献值（注意这里并不是转账值），即\tilde{v};
- l_e当前调用栈深度，即e;
- l_w权限管理，即w;
- t可控的账户（touched accounts）为调用者账户和接收者账户;

虚拟机执行的合约为公式119–120，其中公式119的前8种为预编译好的合约，主要完成一些基本的加密和运算等操作，最后一种即为调用用户的合约。

合约执行的结果如公式106–118所示。

- 如果合约执行失败，则状态回滚到之前的状态。如果执行成功，则状态转变为执行后状态。公式106.
- 如果合约执行失败，则消耗掉所有的gas，gas剩余值为0.成功，则消耗掉执行过程中的gas。公式107.
- 如果合约执行失败，则返回0，成功返回1.

< 以太坊黄皮书详解（一） 以太坊黄皮书详解（三） >