

# 智能合约

路远

中国科学院软件研究所

# 前章要点回顾

- **共识机制的定义**

- 拜占庭将军问题
- 状态机复制/原子广播
- 同步、半同步、异步网络模型

- **典型的共识机制设计**

- 基于**工作量证明**的共识机制
- 基于**权益证明**的共识机制
- 传统的**拜占庭容错共识**机制 --- PBFT
- 新型的**异步拜占庭容错共识**机制 --- 小飞象机制

# 本讲内容

- 智能合约的“前世”与“今生”
- 比特币脚本
- 以太坊智能合约简介
  - 以太坊虚拟机 EVM
  - 高层次合约语言 Solidity
- 验证者困境 和 “燃料” 机制

# 智能合约的 “前世” 与 “今生”

# 从公平交易说起~

- **两个参与方：**

- Alice 持有 **物品X** (也可以是数据或数字签名等)
- Bob 持有 **物品Y** (也可以是数据或数字签名等)



- **公平交易：**

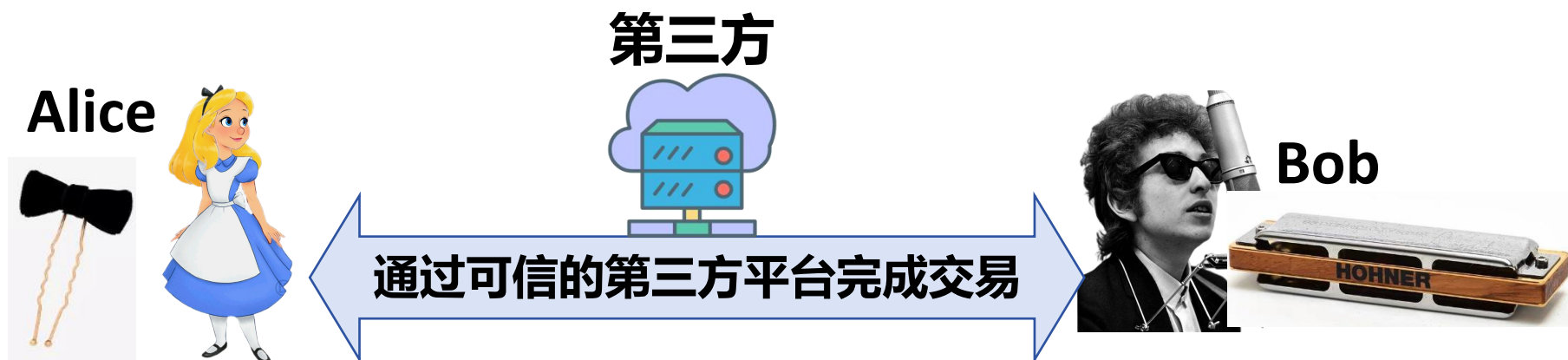
- 要么Alice得到 物品Y 并且 Bob得到 物品X；
- 要么Alice仍持有 物品X 并且 Bob仍持有 物品Y。

- **公平性被破坏的场景：**

- Alice 持有物品X和物品Y
- 或 Bob 持有物品X和物品Y。

# 从公平交易说起~

- 社会经验：只有两个参与方很难做到公平交易！
- 社会实践：往往需要交易双方(在某种程度)都信任的第三方平台



- 可信第三方的作用：根据双方的约定 (合同)，提供仲裁机制！



# 从公平交易说起~

- 1994年，Nick Szabo第一次提出了 smart contract 的概念
- 智能合约：不依靠对可信第三方的依赖，实现可以自动化强制执行的合同条款

f i ® s t m x ñ d @ ¥

PEER-REVIEWED JOURNAL ON THE INTERNET

Read related articles on [Internet economics](#) and [Security](#).

## Formalizing and Securing Relationships on Public Networks by Nick Szabo

### Abstract

*Smart contracts combine protocols with user interfaces to formalize and secure relationships over computer networks. Objectives and principles for the design of these systems are derived from legal principles, economic theory, and theories of reliable and secure protocols. Similarities and differences between smart contracts and traditional business procedures based on written contracts, controls, and static forms are discussed. By using cryptographic and other security mechanisms, we can secure many algorithmically specifiable relationships from breach by principals, and from eavesdropping or malicious interference by third parties, up to considerations of time, user interface, and completeness of the algorithmic specification. This article discusses protocols with application in important contracting areas, including credit, content rights management, payment systems, and contracts with bearer.*

# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型

- 所有节点可以在“公告板”上写入交易
- 交易一旦被“公告板”记录，无法被恶意篡改，并且世界上的所有节点都可以读取该交易





# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上“张贴”可执行的代码？



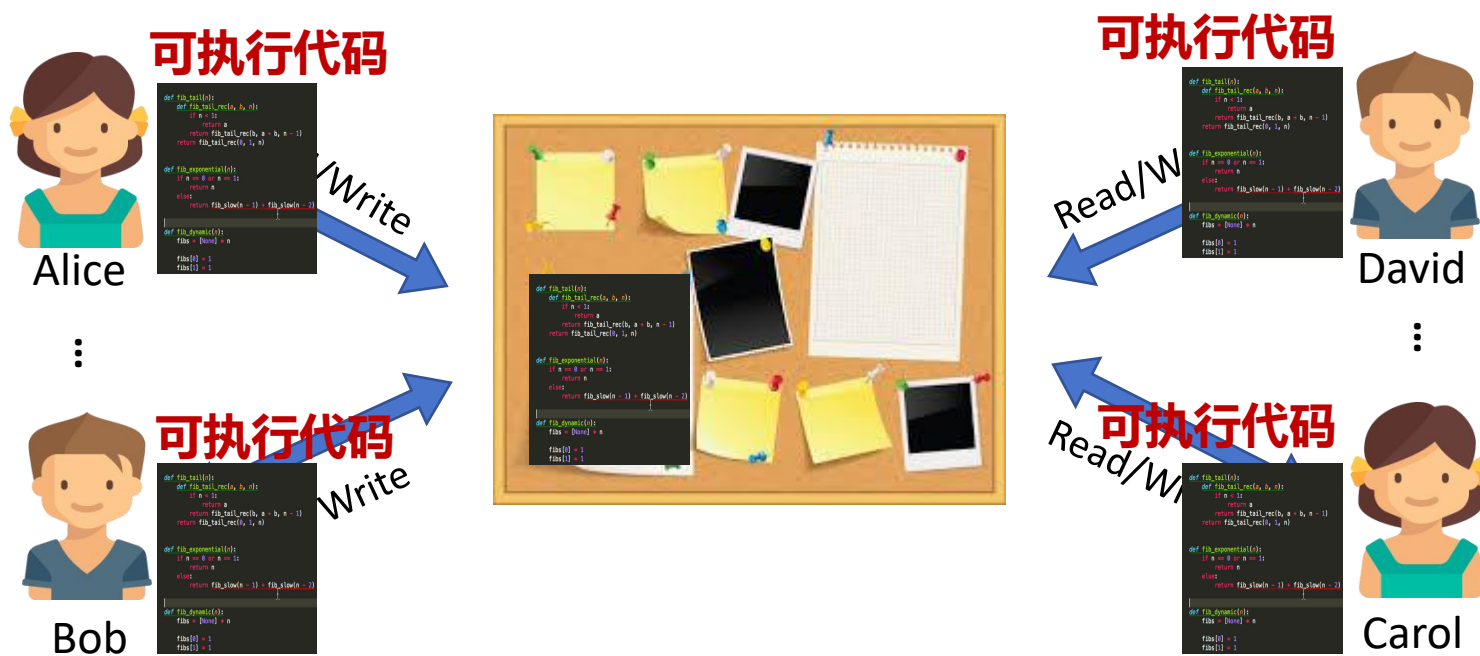
# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上“张贴”可执行的代码？



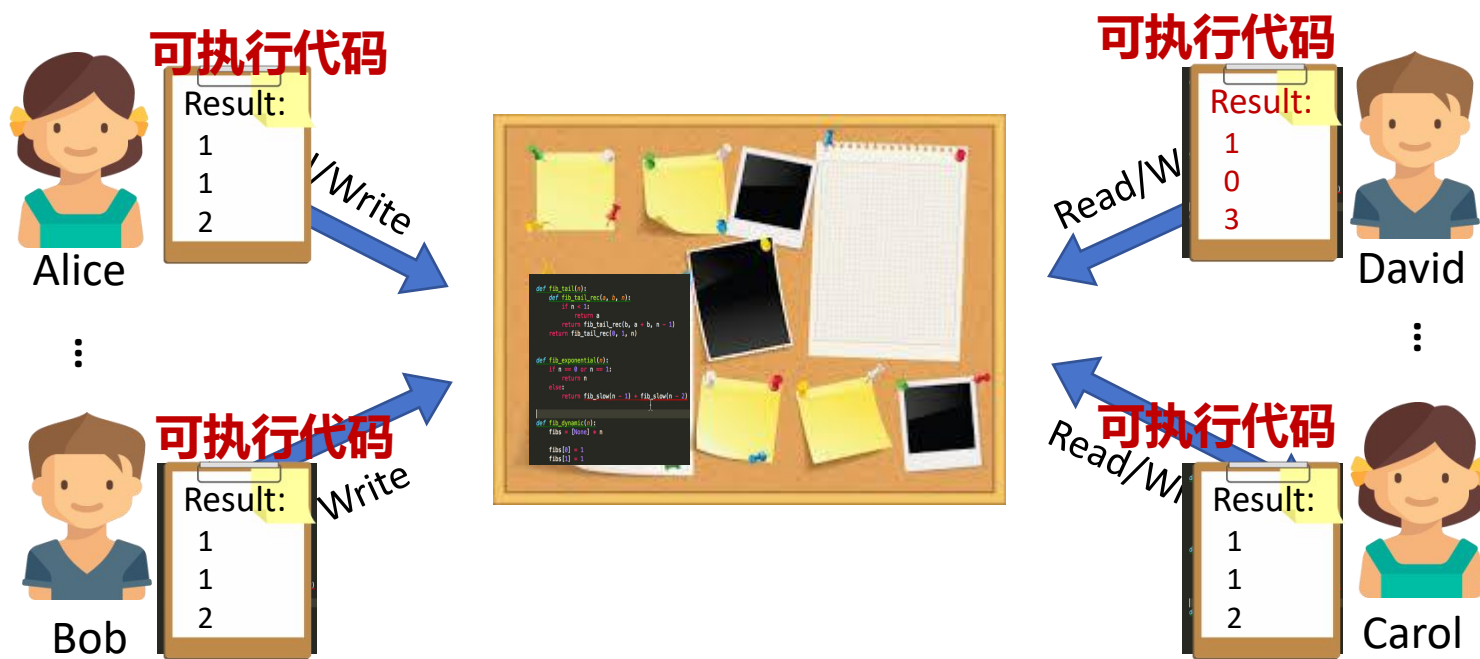
# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上“张贴”可执行的代码？
- 所有节点都可以下载得到相同的可执行代码



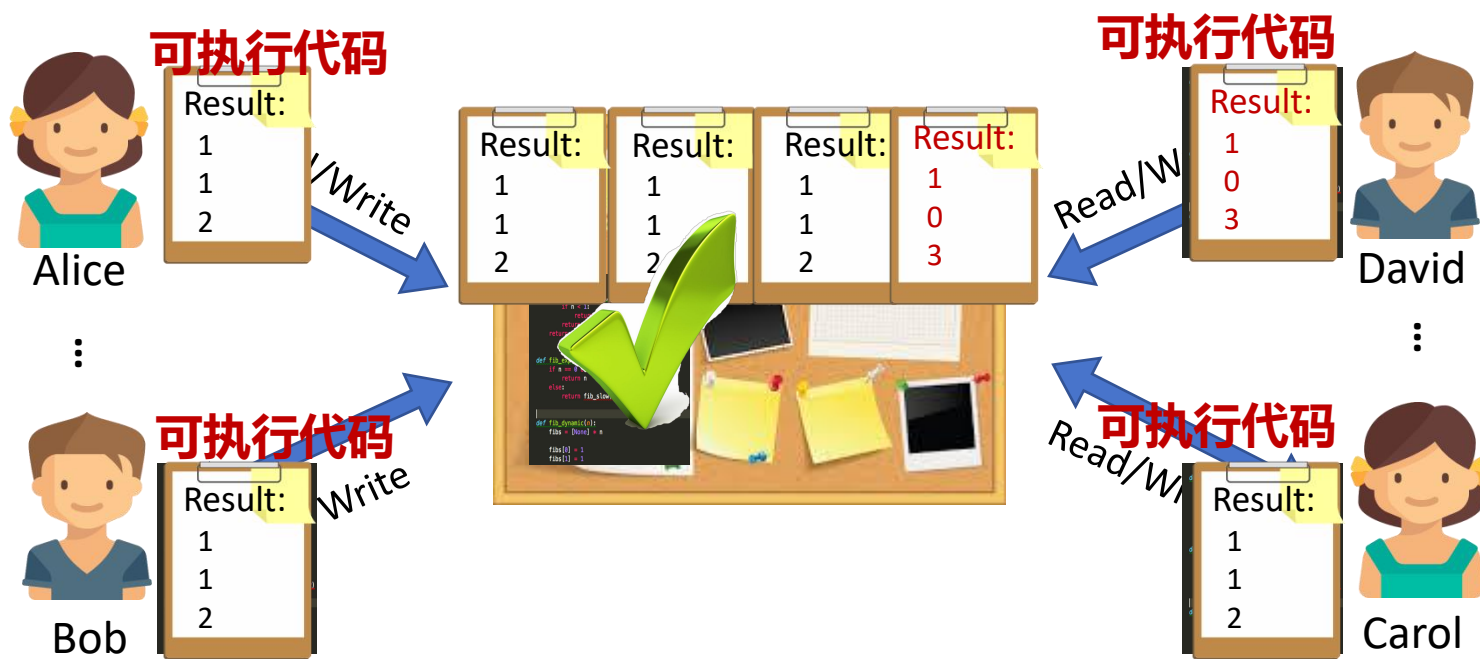
# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上“张贴”可执行的代码？
- 所有节点都能够下载相同的可执行代码
- 如果代码是确定性的，所有节点都可以执行得到相同的结果



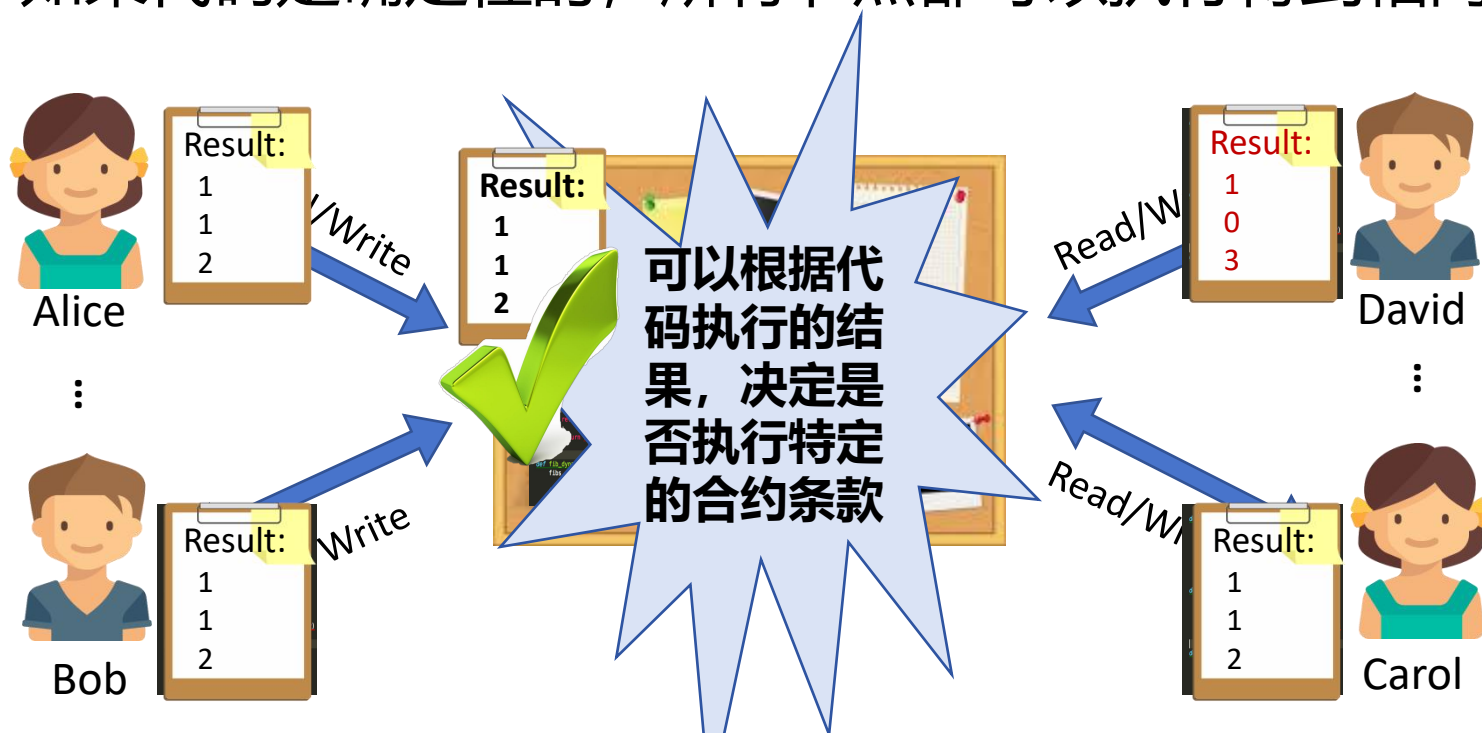
# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上 “张贴” 可执行的代码？
- 所有节点都能够下载相同的可执行代码
- 如果代码是确定性的，所有节点都可以执行得到相同的结果



# 区块链为智能合约提供了一种可能性

- 区块链的 **全局“公告板”** 模型
- 如果 Alice 在公告板上 “张贴” 可执行的代码？
- 所有节点都能够下载相同的可执行代码
- 如果代码是确定性的，所有节点都可以执行得到相同的结果



# 比特币脚本

# Pay-to-Pubkey (P2PK)

在区块30948中，通过哈希值为  
3fa41c9fa50219b23f1c5e395faacfdb17519d264744dd4f19f61664  
69c3f5a3 的 **coinbase**交易 奖励总计50个BTC 给了 **公钥**：

0x046d7853f761df080eec41069291fd8734fd54445f75178477e7c  
5124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802a  
ff523b2ba7bb9686cc

## 输入与输出

Coinbase (新产生的货币)

ScriptSig (ASM) `OP_PUSHBYTES_4 ffff001d`  
`OP_PUSHBYTES_1 0f`

ScriptSig (HEX) `04ffff001d010f`

nSequence `0xffffffff`

P2PK 046d7853f761df080e... bb9686cc 50.00000000 BTC

ScriptPubKey (ASM) `OP_PUSHBYTES_65 046d7853f761df080eec41069291fd8734fd54445f75178477e7c5124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802aff523b2ba7bb9686cc`  
`OP_CHECKSIG`

ScriptPubKey (HEX) `41046d7853f761df080e... 291fd87... 4445f75178477e7c5124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802aff523b2ba7bb9686cc`

类型 P2PK

交易3fa41c...  
有一笔 50 BTC  
的未花费输出  
(UTXO)



# Pay-to-Pubkey (P2PK)

- 交易3fa41c... 输出脚本 (16进制):

**41**046d7853f761df080eec41069291fd8734fd54445f75178477e7c5124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802aff523b2ba7bb9686cc**ac**

- 交易3fa41c... 输出脚本 (汇编码):

**OP\_PUSHBYTES\_65**

046d7853f761df080eec41069291fd8734fd54445f75178477e7c5124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802aff523b2ba7bb9686cc

**OP\_CHECKSIG**

# Pay-to-Pubkey (P2PK)

- 使用 交易3fa41c... 输出脚本 时, 需要提供 **正确的** 输入脚本:

**OP\_PUSHBYTES\_73**

**<signature>**

输入脚本

**OP\_PUSHBYTES\_65**

046d7853f761df080eec41069291fd8734fd54445f75178477e7c5  
124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802  
aff523b2ba7bb9686cc

**OP\_CHECKSIG**

输出脚本

# Pay-to-Pubkey (P2PK)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	<sig>	<pubkey>
		OP_CHECKSIG

# Pay-to-Pubkey (P2PK)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	<sig>	<pubkey>
		OP_CHECKSIG
<sig>		

# Pay-to-Pubkey (P2PK)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		<pubkey>
		OP_CHECKSIG
<pubkey>		
<sig>		

# Pay-to-Pubkey (P2PK)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		OP_CHECKSIG
<pubkey>		
<sig>		

# Pay-to-Pubkey (P2PK)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
true		

# Pay-to-PubkeyHash (P2PKH)

区块222222的  
coinbase交易奖励  
25个BTC给地址：

ecash:qqn6ruf8w8  
09esah89qkvje9xl  
q4x947gvd8x5vhca

(或者Base58Check  
编码  
14cZMQk89mRYQ  
kDEj8Rn25AnGoBi  
5H6uer)

**Block #222,222**

000000000000000b8b49d0b61b14994b5c0a511c4b48a1e251ff2b479b2e6f678

« Prev Block: #222,221

Next Block: #222,223 »

Details

JSON

Summary

Date

2/20, 2013 19:20 utc (11y, 0mo, 29d ago)

Total Output

74,299,121.88 xec

In # / Out #

1,088 / 1,538

UTXO Δ

+450 (+33.8KB)

Min, Max Tx Size

223 - 12,681 B

Size

248.997 KB

Confirmations

614,653

Technical Details

Difficulty

3.651 x 10<sup>6</sup>

Version

0x00000002 (decimal: 2)

Nonce

1684170936

Bits

1a04985c

Merkle Root

14b91663b303a8c805bb169f8c998755a58ce2cc2db7c6361896f85560c8edeaa

Chainwork

798.21 x 10<sup>18</sup> hashes (2b4562c8f6c1b4cc88)

749 Transactions 

Show 20 50 100 all

#0 - 04a77994d23c32a8ad4780ee592b2459985395a4e22e9c2684c94d1059dc7b19

> #0

coinbase

25,000,000 xec

< #0

p2pkh

25,583,400 xec

data(utf-8) - 00000000Mined by BTC Guild000000

ecash:qqn6ruf8w809esah89qkvje9xlq4x947gvd8x5vhca

show raw

25,000,000 xec

25,583,400 xec



# Pay-to-PubkeyHash (P2PKH)

- 交易04a779... 输出脚本 (16进制):

**76a914**27a1f12771de5c  
c3b73941664b2537c153  
16be43**88ac**

- 交易04a779... 输出脚本 (汇编码):

**OP\_DUP**  
**OP\_HASH160**  
**OP\_PUSHBYTES\_20**  
27a1f12771de5cc3b739416  
64b2537c15316be43  
**OP\_EQUALVERIFY**  
**OP\_CHECKSIG**

# Pay-to-PubkeyHash (P2PKH)

- 使用 交易04a779... 输出脚本 时, 需要提供 **正确的** 输入脚本:

```
OP_PUSHBYTES_73 <signature>  
OP_PUSHBYTES_33 <public key>
```

输入脚本

```
OP_DUP OP_HASH160  
OP_PUSHBYTES_20  
27a1f12771de5cc3b73941664b2537c15316be43  
OP_EQUALVERIFY  
OP_CHECKSIG
```

输出脚本

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	<sig>	OP_DUP
	<pubkey>	OP_HASH160
		<pubkeyHash>
		OP_EQUALVERIFY
		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	<sig>	OP_DUP
	<pubkey>	OP_HASH160
		<pubkeyHash>
<pubkey>		OP_EQUALVERIFY
<sig>		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		<del>OP_DUP</del>
		OP_HASH160
<pubkey>		<pubkeyHash>
<pubkey>		OP_EQUALVERIFY
<sig>		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		OP_HASH160
<pubkeyHashNew>		<pubkeyHash>
<pubkey>		OP_EQUALVERIFY
<sig>		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
<pubkeyHash>		
<pubkeyHashNew>		<pubkeyHash>
<pubkey>		OP_EQUALVERIFY
<sig>		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
<pubkeyHash>		
<pubkeyHashNew>		
<pubkey>		OP_EQUALVERIFY
<sig>		OP_CHECKSIG



# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
<pubkey>		
<sig>		OP_CHECKSIG

# Pay-to-PubkeyHash (P2PKH)

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
true		

# 不可花费输出

- **如何使用 比特币 存储 任意数据?**
- 一种思路:
  - 使用极小的转账额
  - 把数据编码成P2PKH输出中的公钥 (无法再花费)
- 另一种思路:
  - **使用 OP\_RETURN 输出脚本**

# 不可花费输出

栈

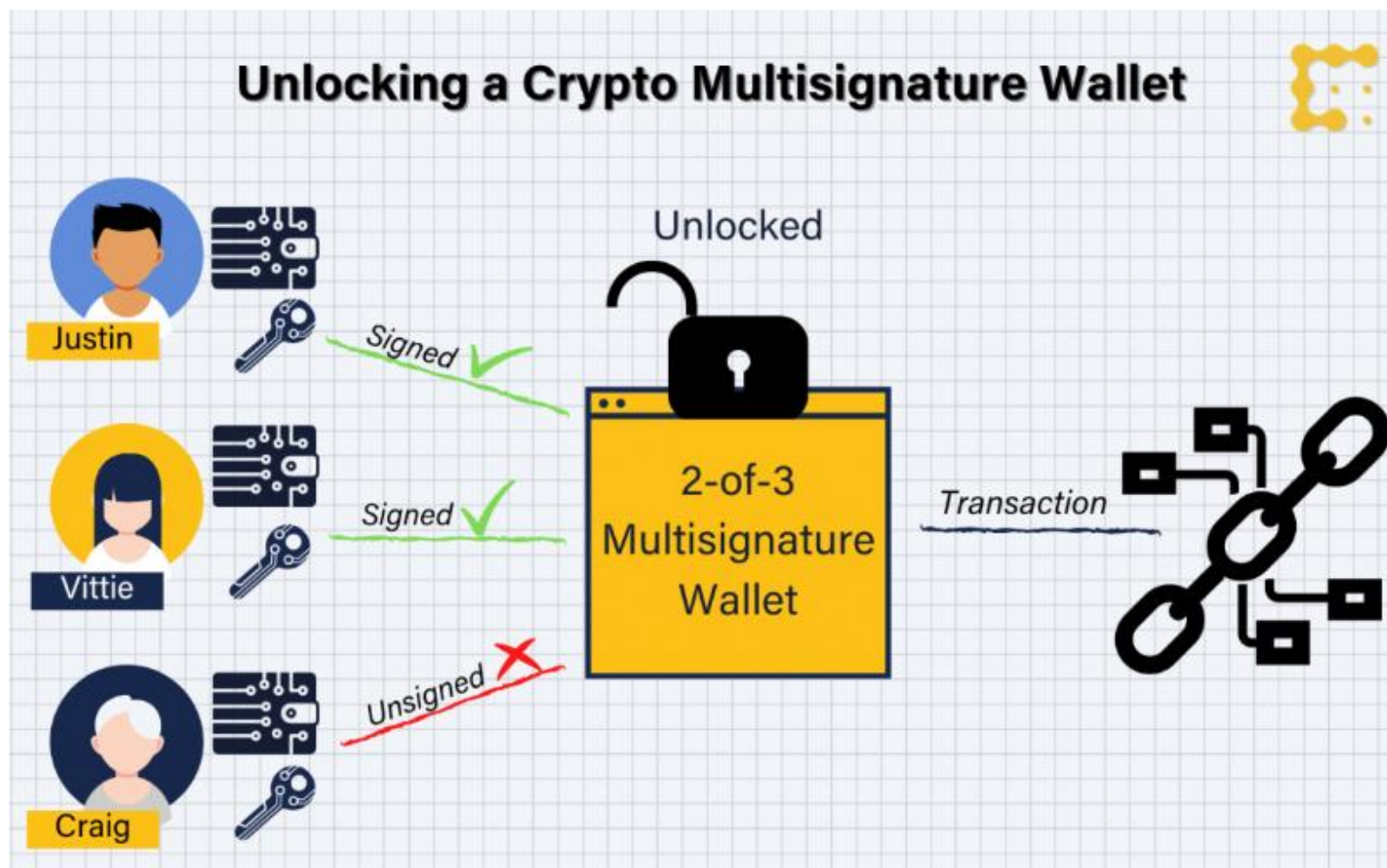
输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	任何输入都无法有效	OP_RETURN
		data 待记录的数据
unspendable		

# Multisig

- **输出脚本**：指定 $n$ 个公钥，要求有效输入至少提供对应其中 $m$ 个不同公钥的有效数字签名



# Multisig

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	OP_0	m
	<sig 1> ... <sig m>	<pubkey 1> ... <pubkey n>
		n
		OP_CHECKMULTISIG

# Multisig

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		m
		<pubkey 1> ... <pubkey n>
		n
<sig 1> ... <sig m>		OP_CHECKMULTISIG
OP_0		

# Multisig

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
n		
<pubkey 1> ... <pubkey n>		
m		
<sig 1> ... <sig m>		OP_CHECKMULTISIG
OP_0		



# Multisig

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
n		
<pubkey 1> ... <pubkey n>		
m		
<sig 1> ... <sig m>		OP_CHECKMULTISIG
OP_0		

由于实现bug, OP\_CHECKMULTISIG  
会从stack中多弹出一个字节, 因此  
需要添加OP\_0作为

# Multisig

栈

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
true		

# Pay-to-ScriptHash (P2SH)

- 普通的 P2PK 输出：67个字节

## **OP\_PUSHBYTES\_65**

046d7853f761df080eec41069291fd8734fd54445f75178477e7c5  
124b7a8775c3c7511f1c9ae975a47b4404e7528bff3e2692614c802  
aff523b2ba7bb9686cc

## **OP\_CHECKSIG**

输出脚本

- P2SH => 向上述脚本的摘要付款

## **OP\_HASH160**

08c30a74cc459e12e1eb3be56605c4aa2bd3a233

## **OP\_EQUAL**

P2SH  
输出脚本

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

<pubKey>  
OP\_CHECKSIG

Stack	scriptSig	scriptPubkey
	<sig>	OP_HASH160
	<serializedscript>	<scriptHash>
		OP_EQUAL

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
	<sig>	OP_HASH160
	<serializedscript>	<scriptHash>
		OP_EQUAL
<serializedscript>		
<sig>		

复本

<serializedscript>  
<sig>

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		OP_HASH160
		<scriptHash>
		OP_EQUAL
scriptHashNew		
<sig>		

复本

<serializedscript>  
<sig>

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
		<scriptHash>
<scriptHash>		OP_EQUAL
scriptHashNew		
<sig>		

复本

<serializedscript>  
<sig>

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
<scriptHash>		OP_EQUAL
scriptHashNew		
<sig>		

复本

<serializedscript>  
<sig>



# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
true		
<sig>		

复本

<serializedscript>  
<sig>

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
OP_CHECKSIG		
<pubkey>		
<sig>		

复本  
(反序列化)

<serializedscript>  
<sig>

# Pay-to-ScriptHash (P2SH)

输入脚本

输出脚本

Stack	scriptSig	scriptPubkey
true		

# 一些比特币脚本举例

- 交易 a4bfa8....31b 的输出脚本 (奖励哈希函数原像):

OP\_HASH256

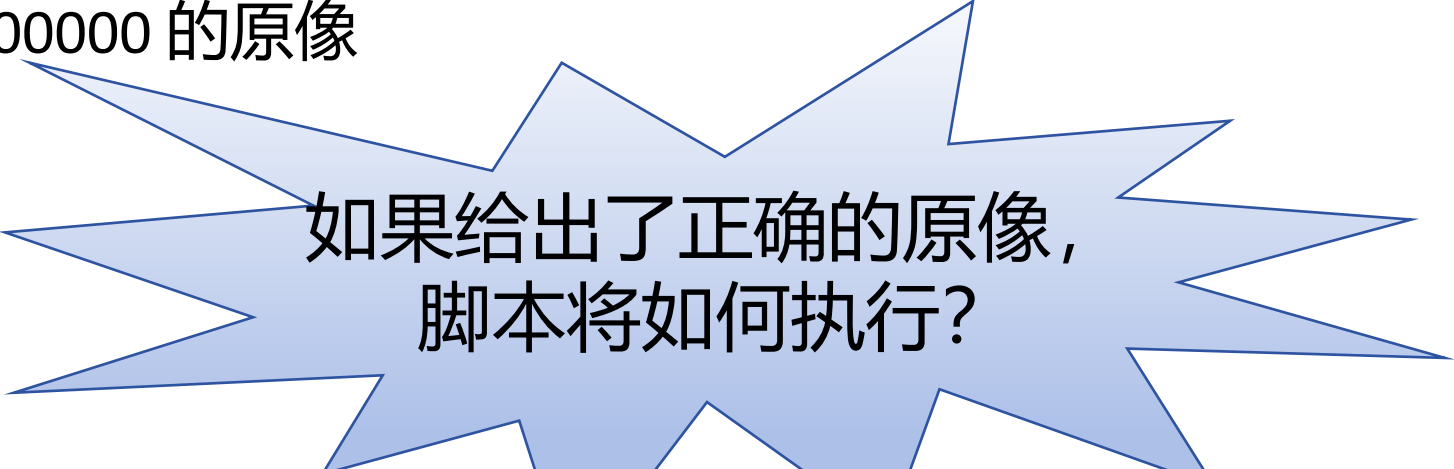
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d61900

00000000 OP\_EQUAL

有效的输入脚本 => 给出 哈希值

6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d61900

00000000 的原像



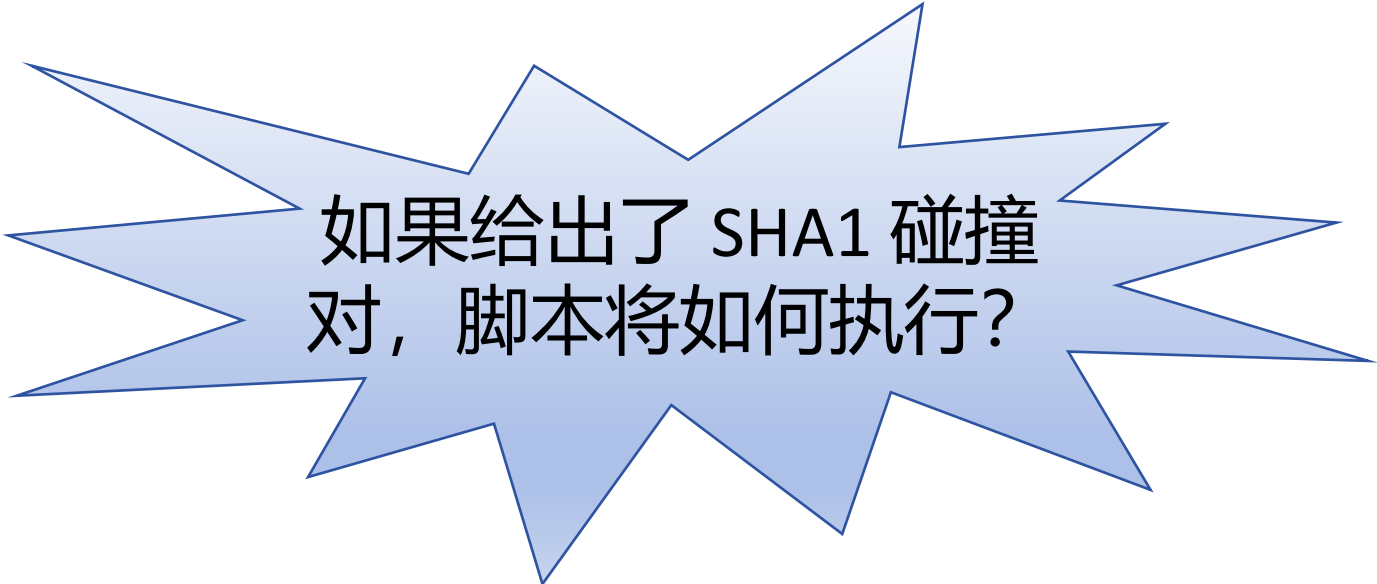
如果给出了正确的原像，  
脚本将如何执行？

# 一些比特币脚本举例

- 一个奖励 SHA1 碰撞对的输出脚本:

```
OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP  
OP_SHA1 OP_EQUAL
```

有效的输入脚本 => 给出 SHA1 碰撞对



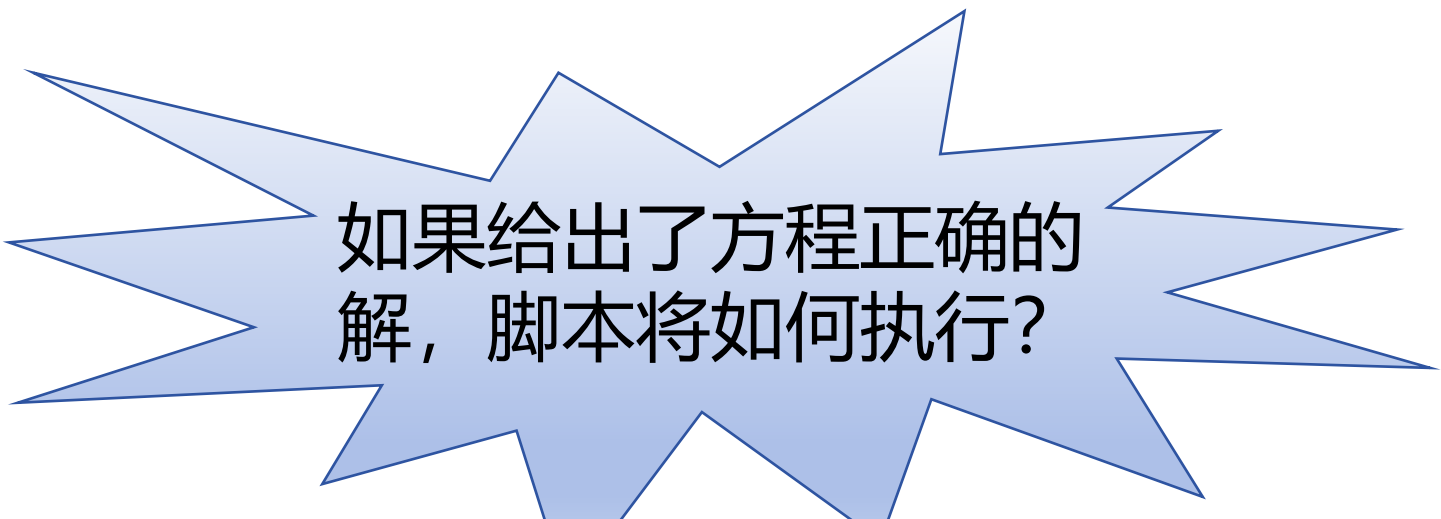
如果给出了 SHA1 碰撞对，脚本将如何执行？

# 一些比特币脚本举例

- 一个奖励 线性方程  $3x + 7 = 13$  根的输出脚本:

OP\_DUP OP\_DUP 7 OP\_ADD OP\_ADD OP\_ADD 13 OP\_EQUAL

有效的输入脚本 => 给出 方程  $3x + 7 = 13$  的根  $x=2$



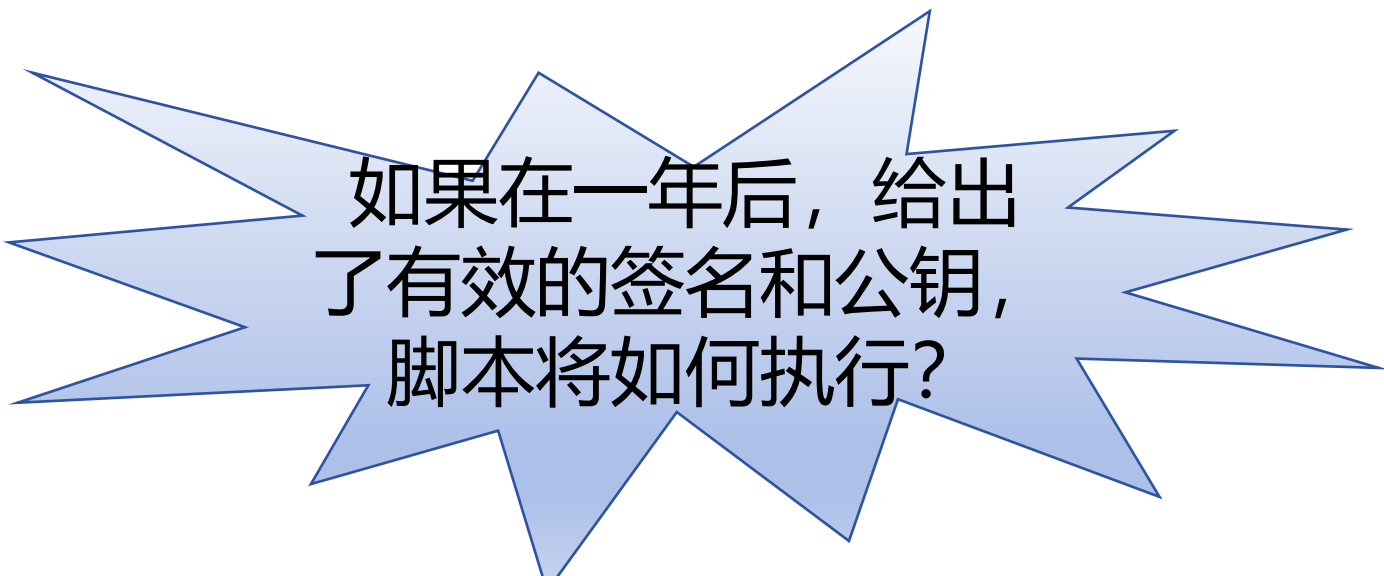
如果给出了方程正确的解，脚本将如何执行？

# 一些比特币脚本举例

- 第二年才可以花费的P2PKH输出脚本:

```
<NextYear> OP_CHECKLOCKTIMEVERIFY OP_DROP  
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

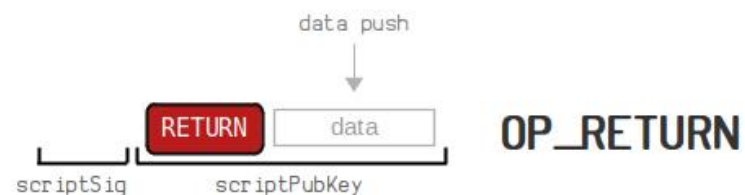
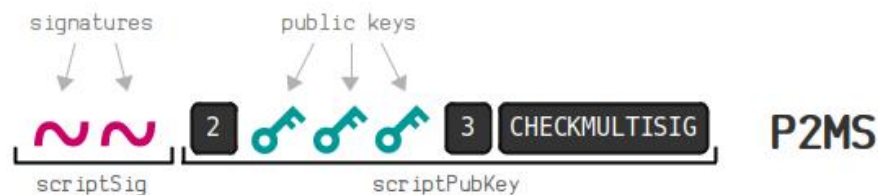
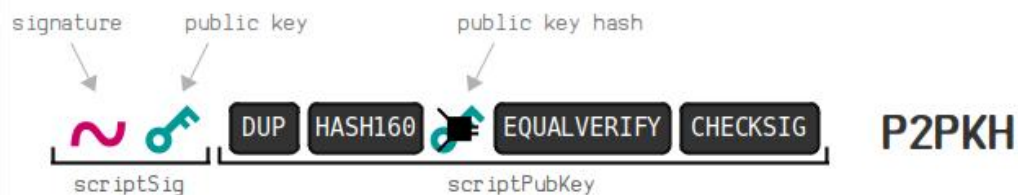
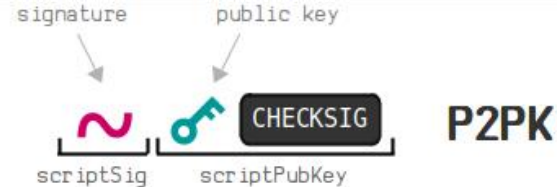
有效的输入脚本: <signature> <pubKey>



如果在一年后，给出了有效的签名和公钥，脚本将如何执行？

# BTC脚本小结

- P2PK (Pay To Public Key)
- P2PKH (Pay To Pubkey Hash)
- P2MS (Pay To Multisig)
- P2SH (Pay To Script Hash)
- OP\_RETURN





# 以太坊智能合约简介

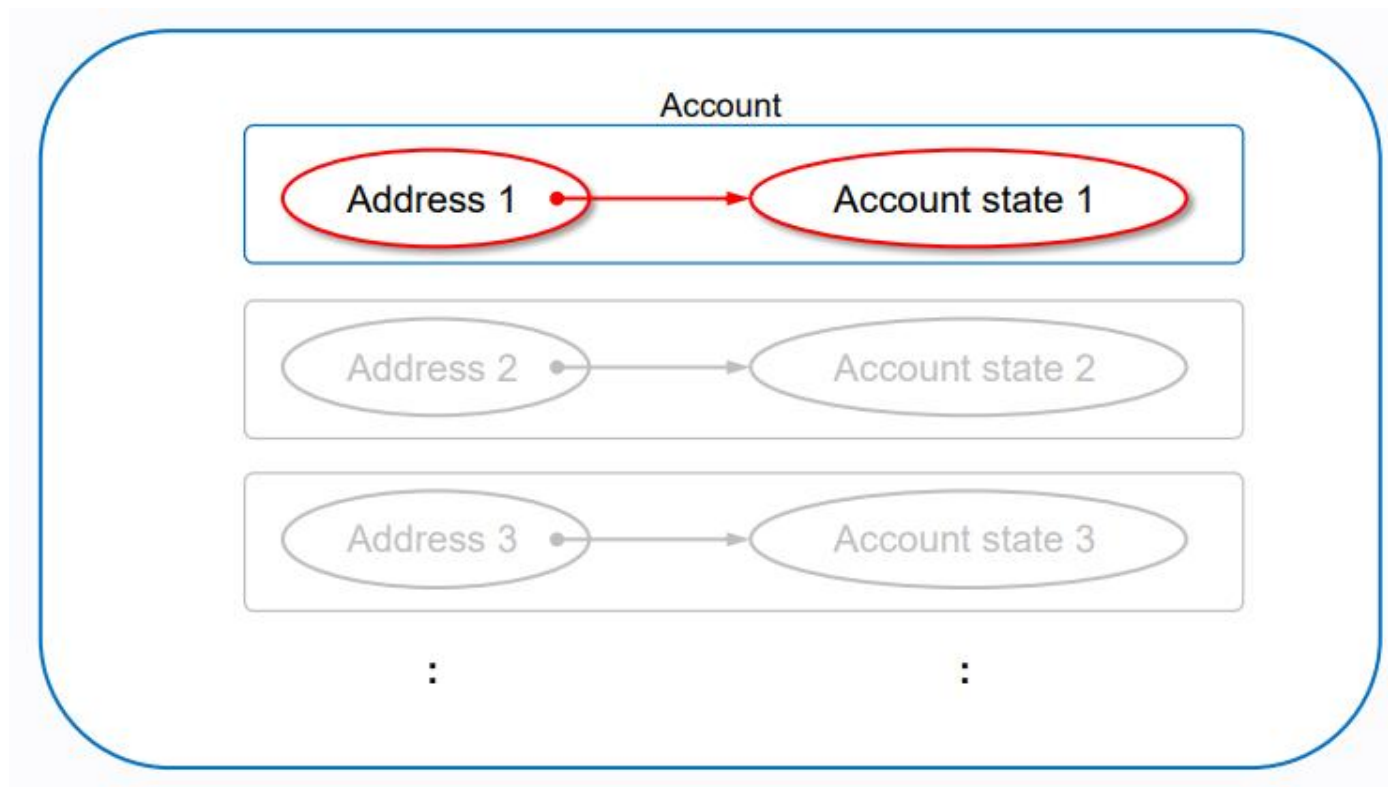
# 比特币脚本的局限

- 没有循环
  - 非图灵完备
  - 没有复杂的协议流控制
  - 不支持除法运算
  - 等等.....
- 
- 能否支持更复杂的合约功能?



# 账户模型

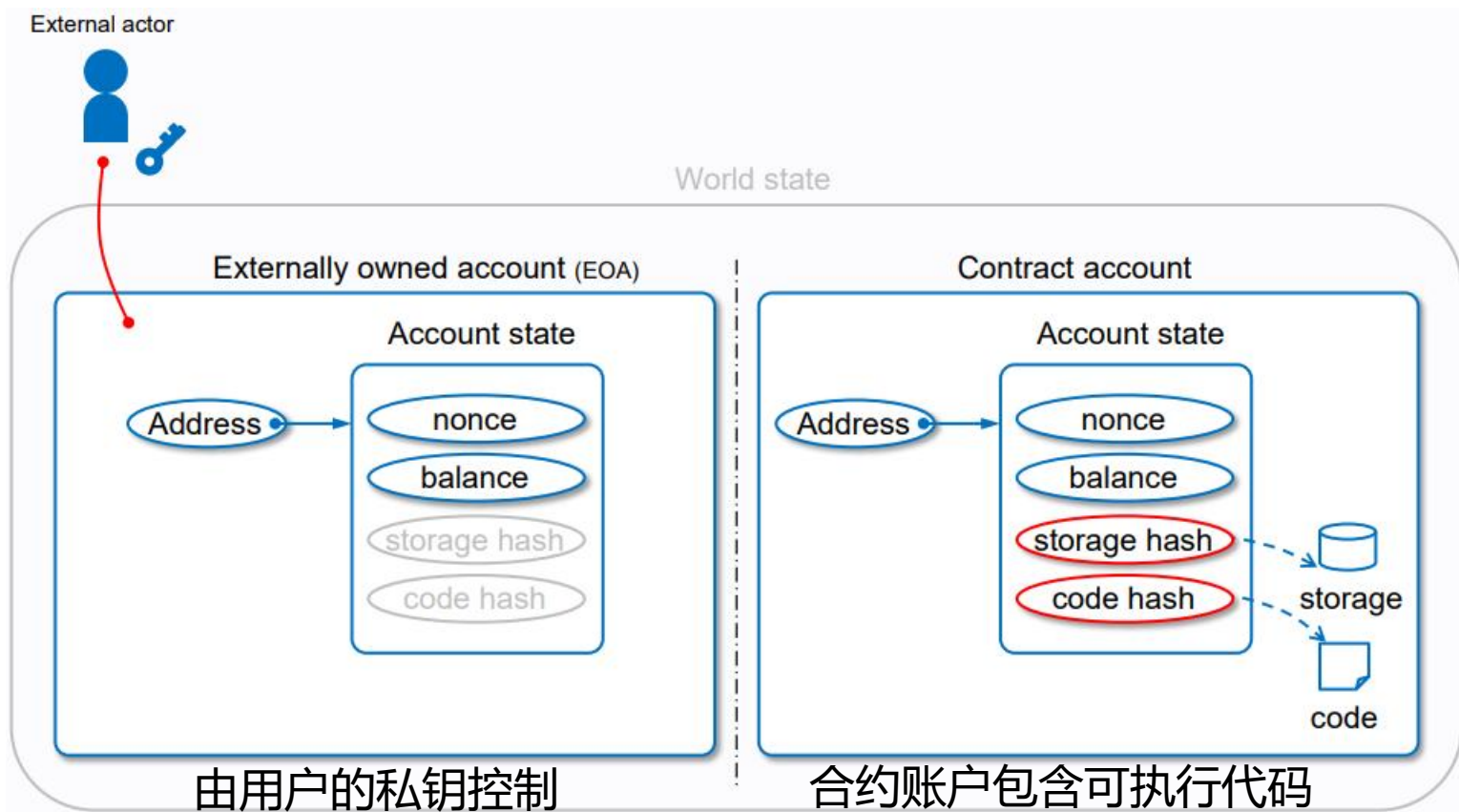
- 回顾：以太坊采用**账户模型**，使用默克尔基数树存储
- 默克尔基数树是key-value类型的认证数据结构，账户地址是key，账户状态是value



# 普通账户 vs 合约账户

- 以太坊有两类账户

- **外部拥有账户** (普通账户): 由用户的私钥控制, 状态包括余额信息等
- **智能合约账户**: 地址由创建者的普通账户地址自动导出, 包含可执行代码、变量状态、余额等信息



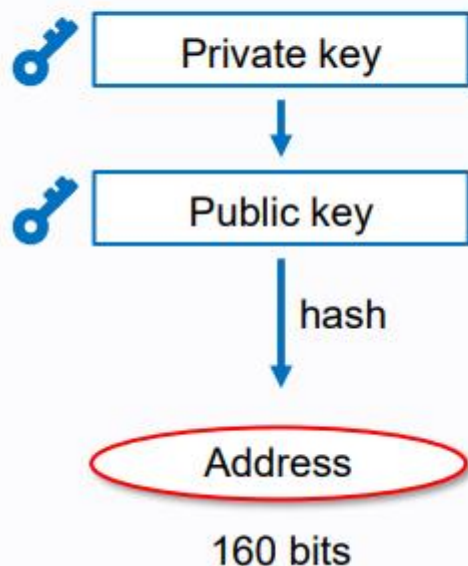
# 普通账户 vs 合约账户

- 以太坊有两类账户

- 外部拥有账户** (普通账户) **EOA**: 由用户的私钥控制, 状态包括余额信息等
- 智能合约账户 CA**: 地址由创建者的普通账户地址导出, 包含可执行代码、变量状态、余额等信息

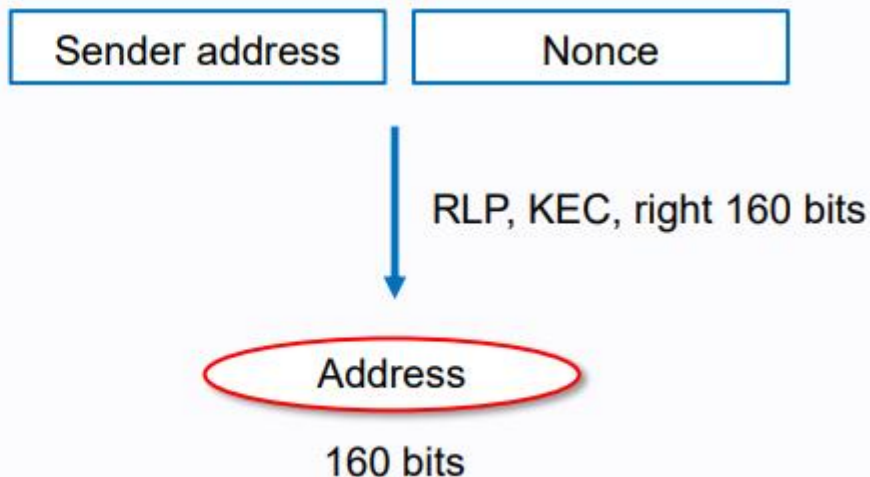
Externally owned account (EOA)

外部拥有账户地址由密钥导出



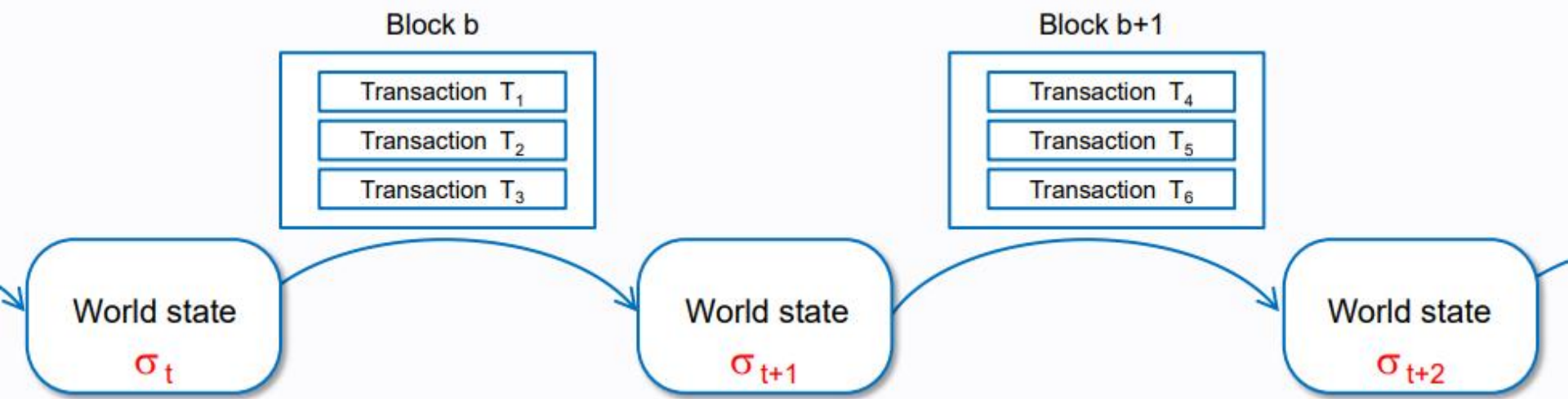
Contract account

合约地址由创建者的普通账户地址导出



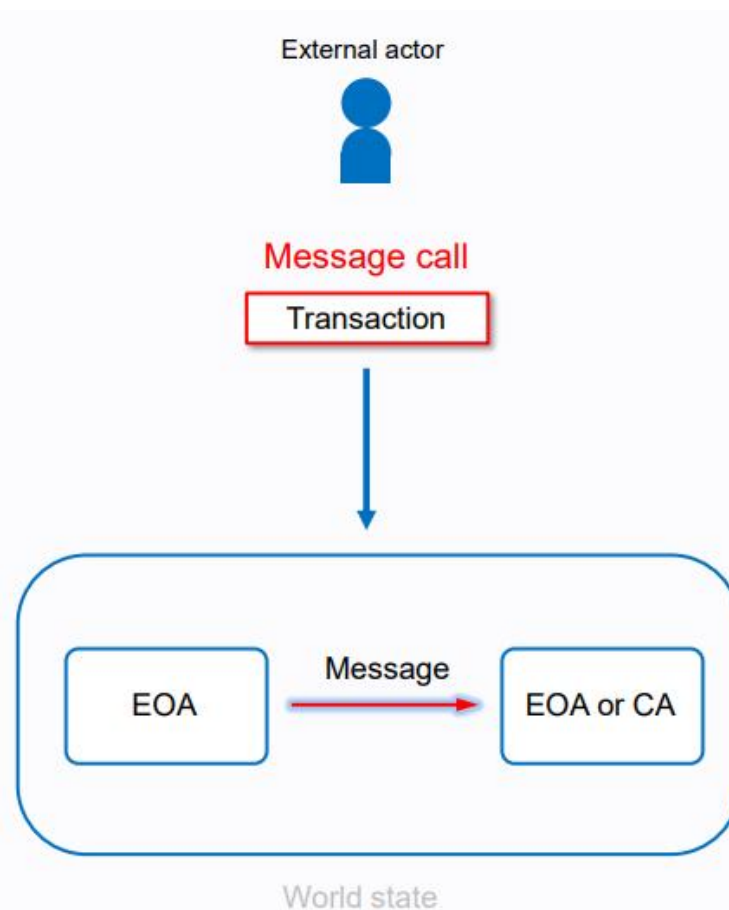
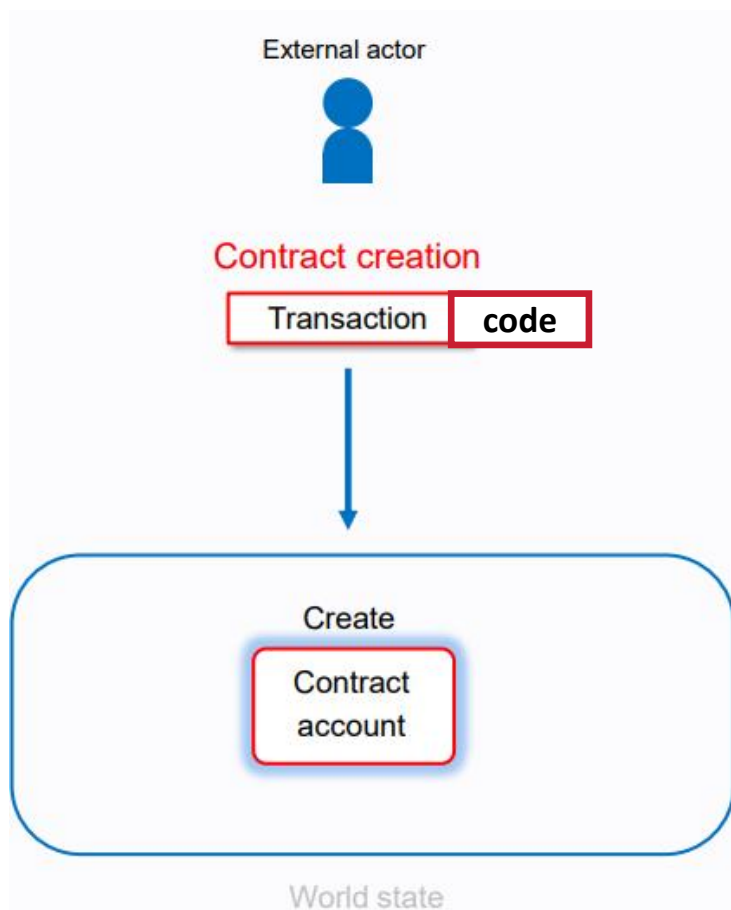
# 以太坊：交易驱动的状态机

- 所有的账户状态 => 世界状态
- 外部账户拥有者可以发送新的交易 (有效的数字签名)
- 新区块打包了新交易 => 世界状态在交易的驱动下发生变化
  - 如转账交易：收款方账户余额增长，付款方账户余额减少
  - 也可以创建新的智能合约，或执行已经部署的智能合约



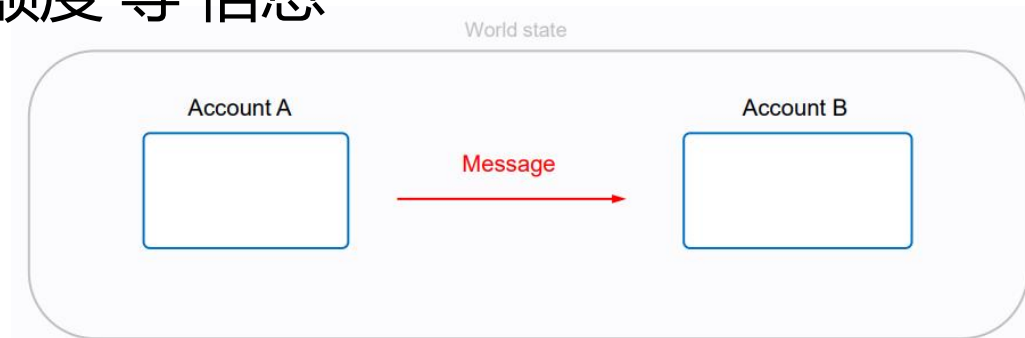
# 两种实际的交易类型

- 合约创建交易：在世界状态中创建新的合约账户 (包括初始状态)
- 消息呼叫交易：触发“消息”，实现EOA/CA与EOA/CA间的状态读写

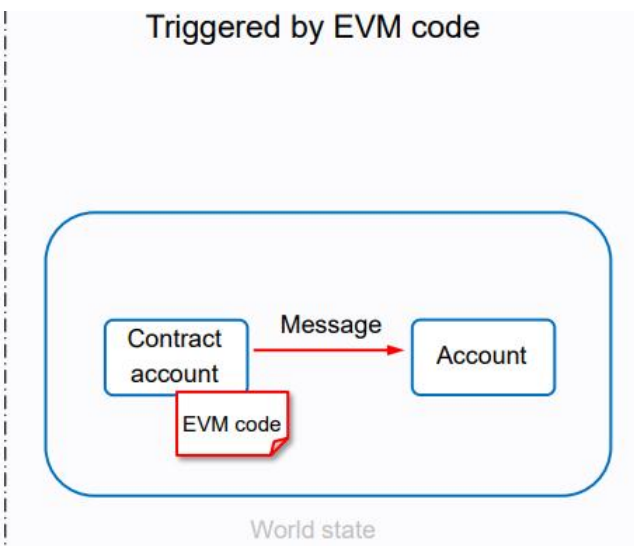
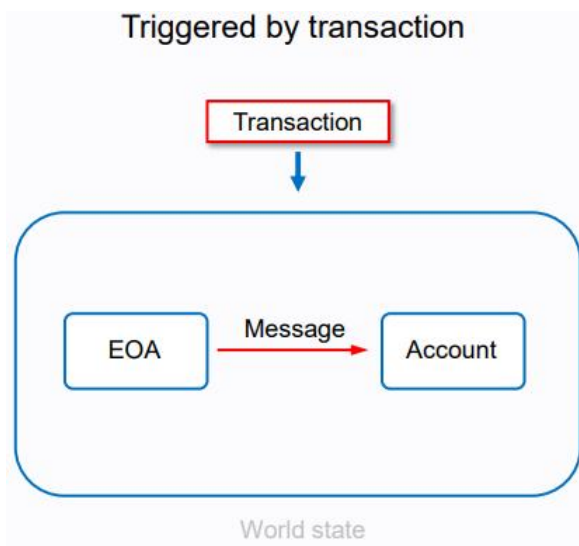


# 消息

- “消息”在两个账户之间传输：携带数据(如调用特定函数的输入)或者转账额度等信息



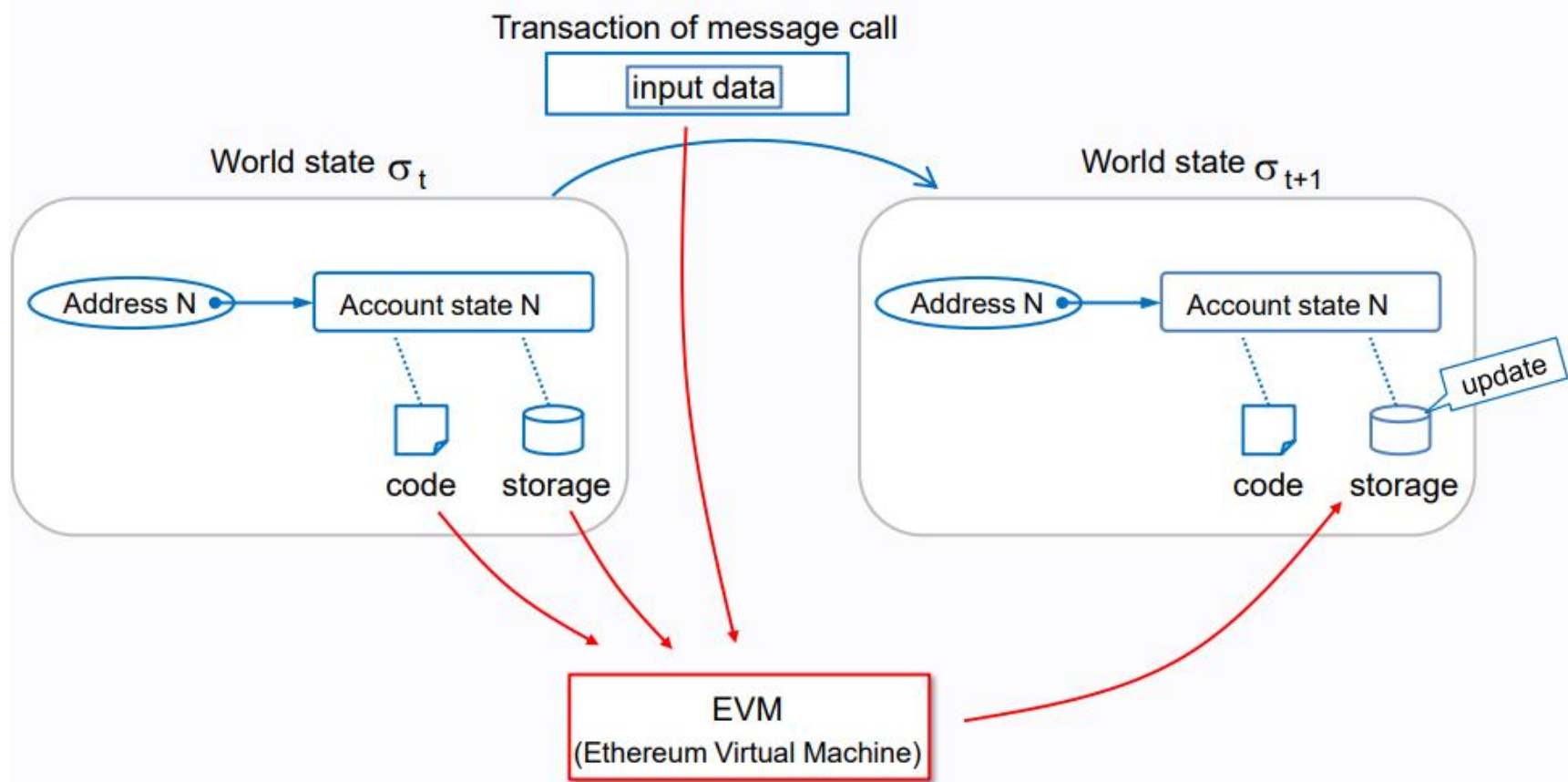
- 消息可以由交易触发，也可以由智能合约代码的执行过程触发





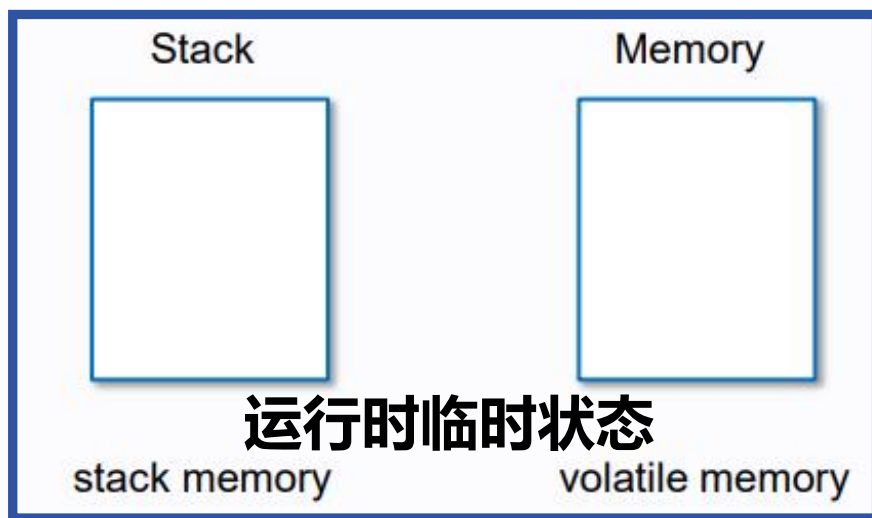
# 以太坊虚拟机 EVM

- 以太坊中的智能合约代码需要按照提前规定的以太坊虚拟机执行
  - 栈机、256-bit字长、**确定性执行**、汇编型代码



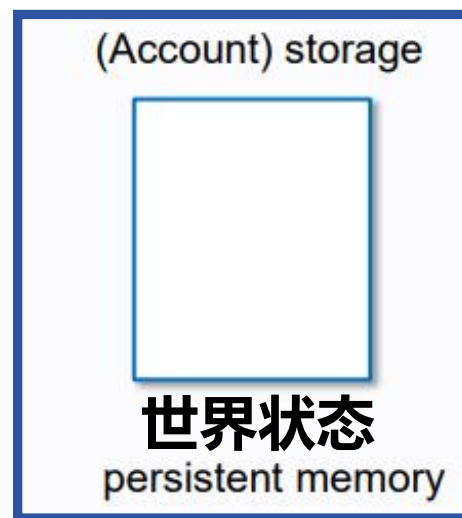
# 以太坊虚拟机 EVM

- EVM中的三类空间
  - 栈：栈中的数据可以被执行各类计算操作
  - 内存：记录智能合约执行过程中的临时变量
  - 账户(存储)：永久存储的世界状态变量



256 bits x 1024 elements

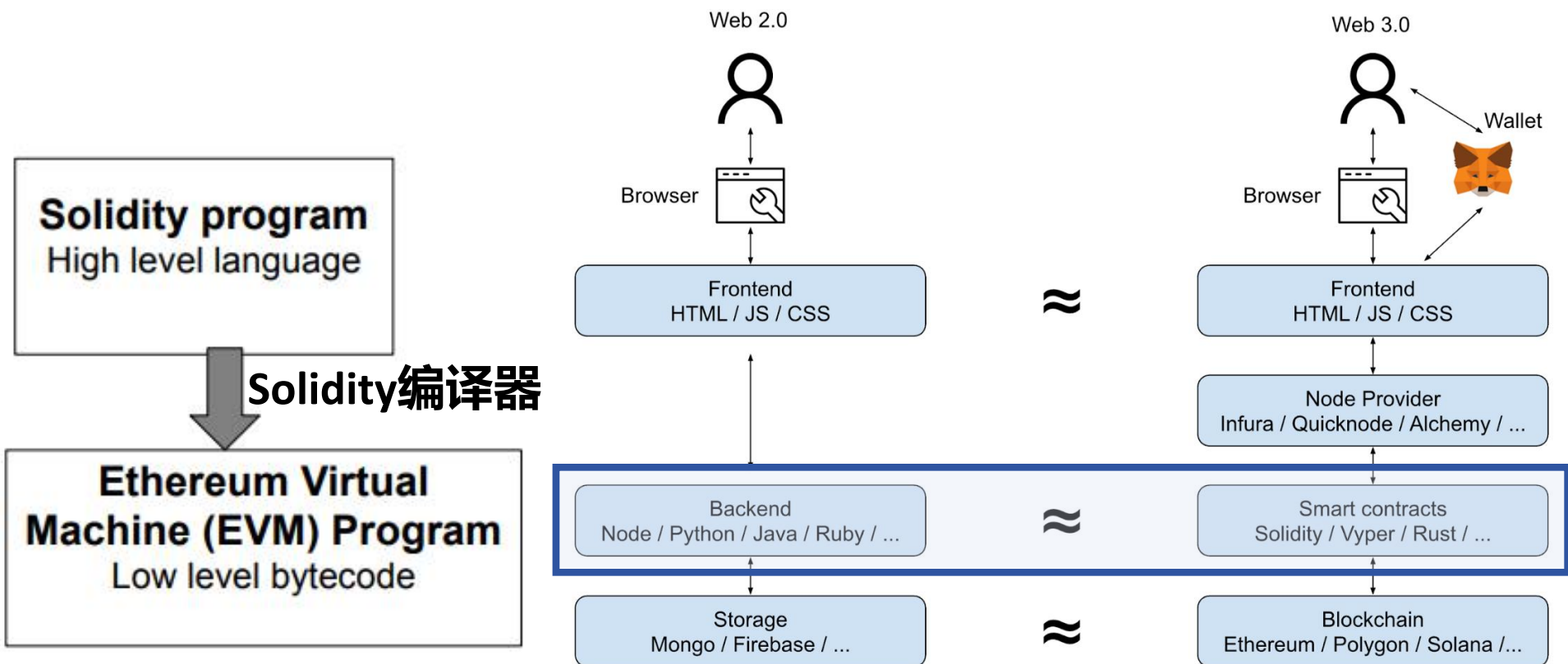
byte addressing  
linear memory



256 bits to 256 bits  
key-value store

# 高层次编程语言

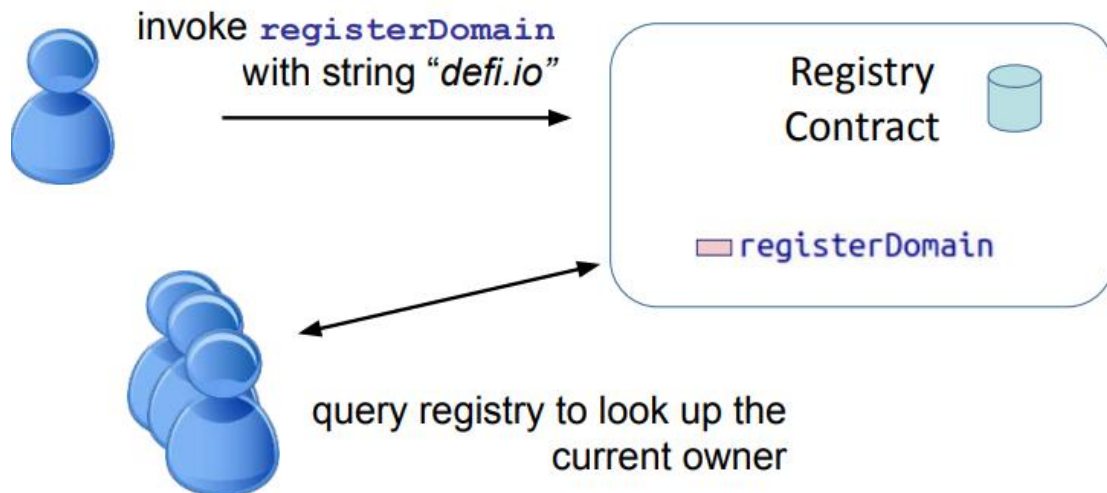
- EVM汇编代码 => **开发效率低、学习门槛高**
- 智能合约的高层次开发语言? => **Solidity 等**



# 高层次编程语言

- 一个（简化的）Solidity 例子 —— 链上域名注册

```
1 pragma solidity ^0.5.0;
2
3 contract MyRegistry {
4
5     mapping ( string => address ) public registry;
6
7     function registerDomain(string memory domain) public {
8         // Can only reserve new unregistered domain names
9         require(registry[domain] == address(0));
10
11         // Update the owner of this domain
12         registry[domain] = msg.sender;
13     }
14 }
```



维护一个映射列表，

节点可以调用函数 `registerDomain()`，

在映射表中添加一对新的映射

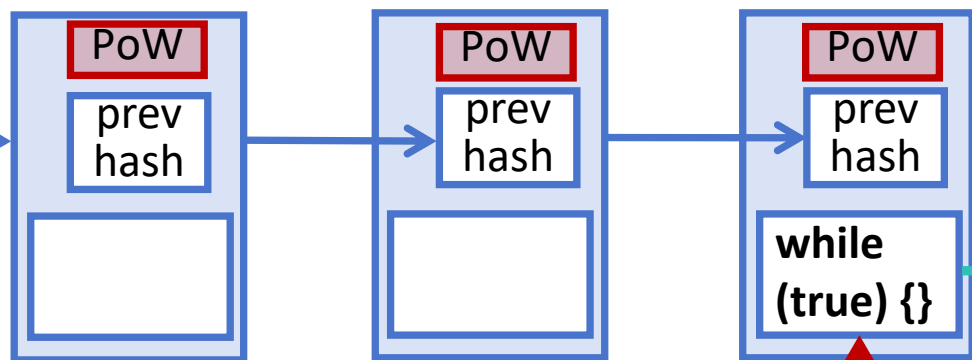
思考：存在什么安全隐患？有什么解决思路么？

验证者困境和 “燃料” 机制

# 停机问题

- **停机问题：** 判断一个程序是否能在有限的时间之内结束运行。图灵完备程序的停机问题是不可判定的。

- **验证者困境：**



不停机的恶意代码：  
`while (true) {}`

诚实矿工的两难抉择：

- 1) **验证合约结果** => 由于图灵机上的停机问题无法判定，可能无法执行结束，等待执行结束的过程种无法挖矿；
- 2) **跳过结果验证、打包空块** => 导致交易不再被处理，破坏共识机制的活性。

# 验证者困境

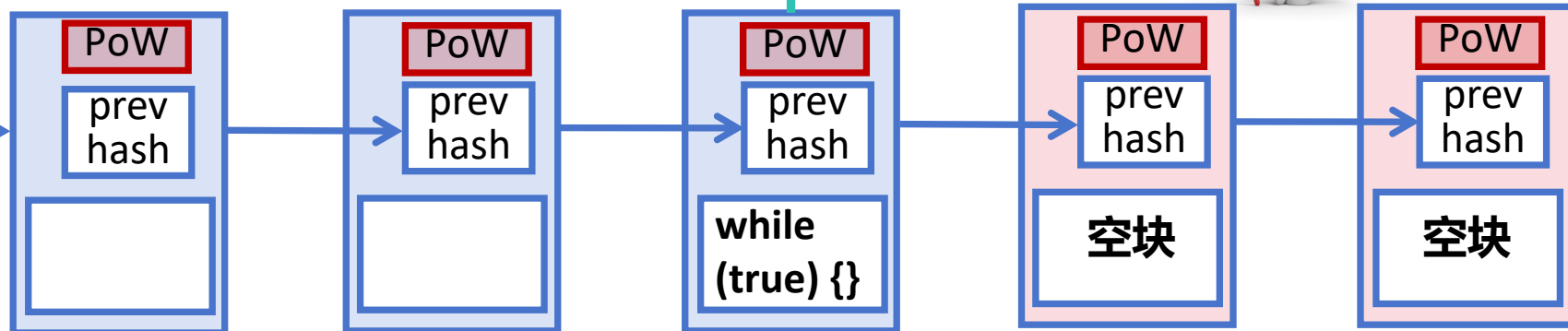
- 从验证者困境中学到了什么?
- 复杂的智能合约 带来安全性问题
  - => 诚实节点的哈希算力可能被浪费
  - => 恶意节点能够用少量算力控制整个区块链



诚实节点在死循环中“卡死”，无法继续开展挖矿

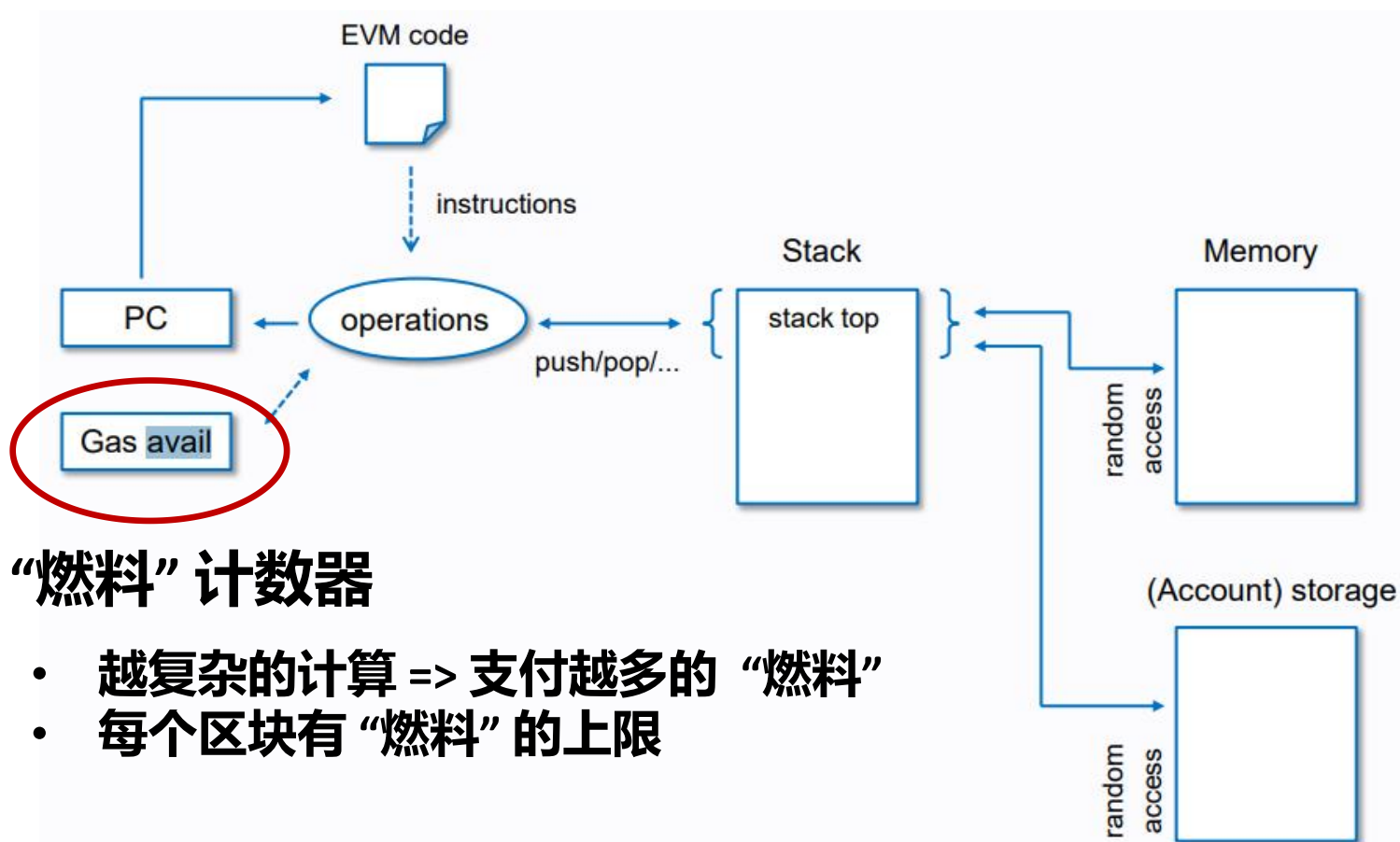


恶意节点继续挖取空块



# EVM中的燃料机制

- 在EVM中，每个操作码都需要一定的“燃料”
- 在EVM中，实现有“燃料”计数器！



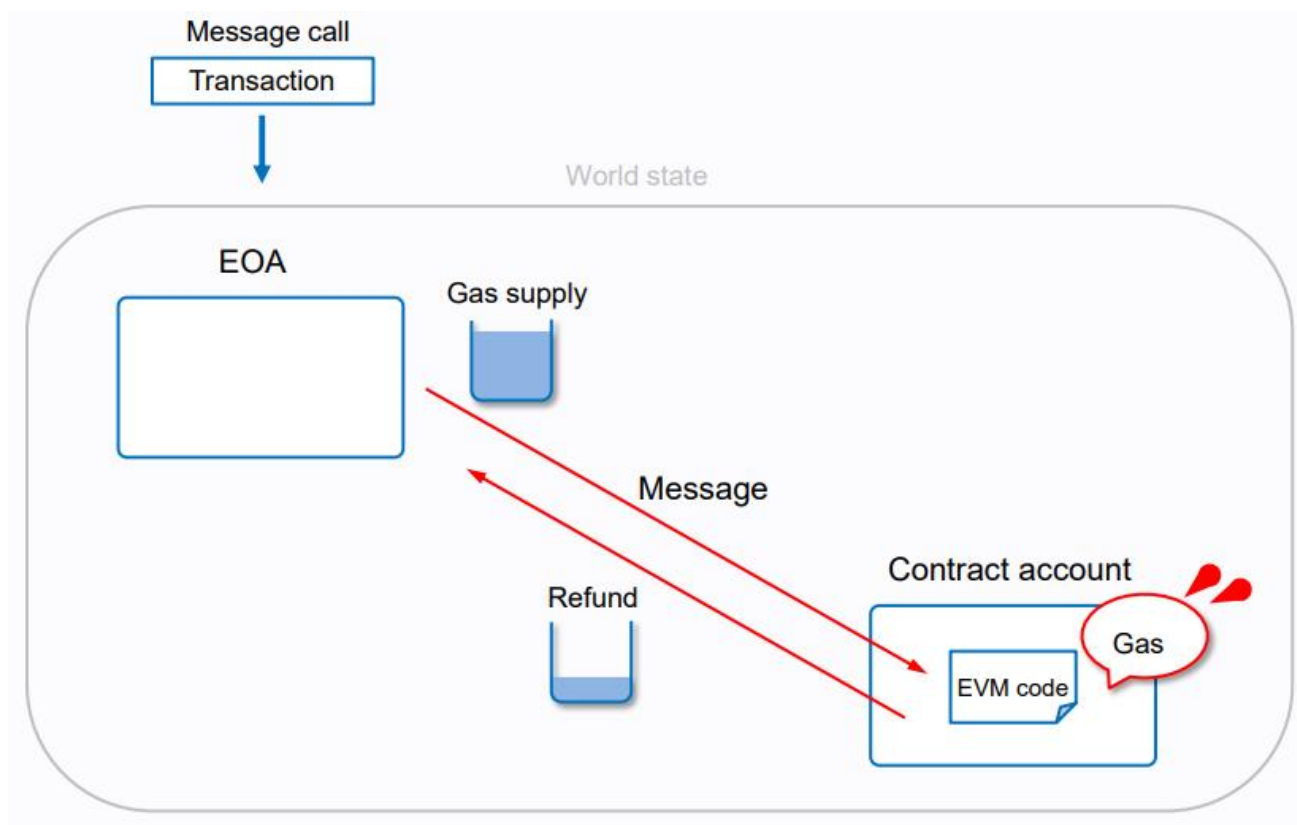
## “燃料” 计数器

- 越复杂的计算 => 支付越多的“燃料”
- 每个区块有“燃料”的上限



# EVM中的燃料机制

- 调用合约的外部拥有账户 (EOA) 负责支付 燃料费
- 如果 燃料费 有剩余, 会返回给支付 燃料费 的 EOA
- 如果 燃料费 不足, “燃料” 计数器会抛出 out-of-gas 异常



# EVM中的燃料机制

- **燃料费定价**

- EVM设计者根据操作的耗时程度，合理指定每个操作的燃料费

- **过低的燃料费定价**

- 可能导致 DDoS 攻击
- 攻击者发送包含过低估价 opcodes 的交易
- 仅需花费很少的燃料费即可以大量浪费诚实矿工的验证时间

	A	B	C
1	Value	Mnemonic	Gas Used
2	0x00	STOP	0
3	0x01	ADD	3
4	0x02	MUL	5
5	0x03	SUB	3
6	0x04	DIV	5
7	0x05	SDIV	5
8	0x06	MOD	5
9	0x07	SMOD	5
10	0x08	ADDMOD	8
11	0x09	MULMOD	8
12	0x0a	EXP	FORMULA
13	0x0b	SIGNEXTEND	5
14	0x10	LT	3

# EVM中的燃料机制

- 过低燃料费定价的问题和修复

- EIP-150

- Increase the gas cost of EXTCODESIZE to 700 (from 20).

- Increase the base gas cost of EXTCODECOPY to 700 (from 20).

- Increase the gas cost of BALANCE to 400 (from 20).

- Increase the gas cost of SLOAD to 200 (from 50).

- Increase the gas cost of CALL, DELEGATECALL, CALLCODE to 700 (from 40).

- Increase the gas cost of SELFDESTRUCT to 5000 (from 0).

- If SELFDESTRUCT hits a newly created account, it triggers an additional gas cost of 25000 (similar to CALLs).



- EIP-2929

- Increases gas cost for SLOAD , \*CALL , BALANCE , EXT\* and SELFDESTRUCT when used for the first time in a transaction.

练习

- 输出脚本：OP\_3DUP OP\_ADD 5 OP\_EQUALVERIFY OP\_ADD 4 OP\_EQUALVERIFY OP\_ADD 3 OP\_EQUAL 给出有效的输入脚本。
- 脚本：1 3 OP\_ADD 5 OP\_MAX 8 OP\_MUL OP\_DUP OP\_MUL 执行之后，栈顶是多少？
- 脚本：5 9 **OP\_2DUP** OP\_MUL OP\_MUL OP\_MUL 执行之后，栈顶是多少？
- 脚本：6 9 7 **OP\_DROP** OP\_ADD OP\_DUP OP\_MUL 执行之后，栈顶是多少？
- 脚本：0 OP\_IF 100 OP\_ELSE 200 OP\_ENDIF 执行之后，栈顶是多少？

- 输出脚本:

```
OP_IF
  SHA256 <digest> OP_EQUALVERIFY OP_DUP OP_HASH160 <Bob's pubkey
hash>
OP_ELSE
  <NextYear> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160
<Alice's pubkey hash>
OP_ENDIF
OP_EQUALVERIFY
OP_CHECKSIG
```

输入脚本:

```
<sig> <Bob's pubkey> <preimage> 1
或者
<sig> <Alice's pubkey> 0
```

解释上述脚本在Alice和Bob之间达成了怎样的交易?

谢谢