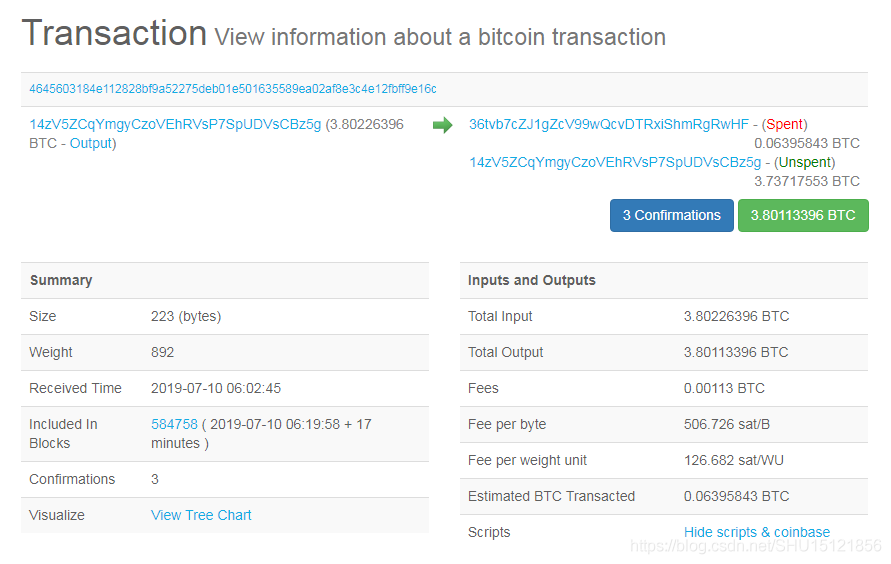
一、比特币脚本的整体流程

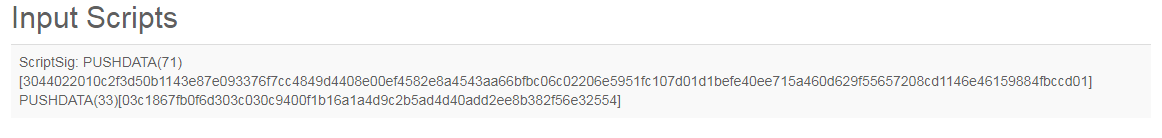
比特币系统中使用的脚本语言很简单，唯一能访问的内存空间就是一个栈，这点和通用脚本语言的区别很大。

在blockchain.info上观察一个交易，这个交易有一个输入和两个输出，其中一个输出已经被花出去了，另一个没有被花出去。



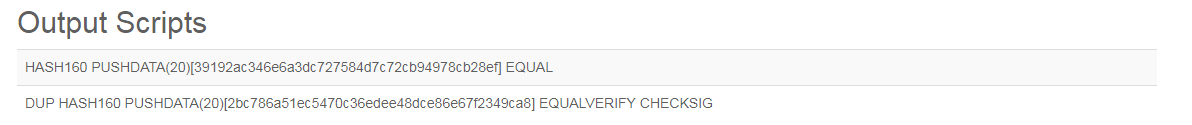
**输入脚本**

输入脚本包含两个操作，分别将两个很长的数压入栈中。

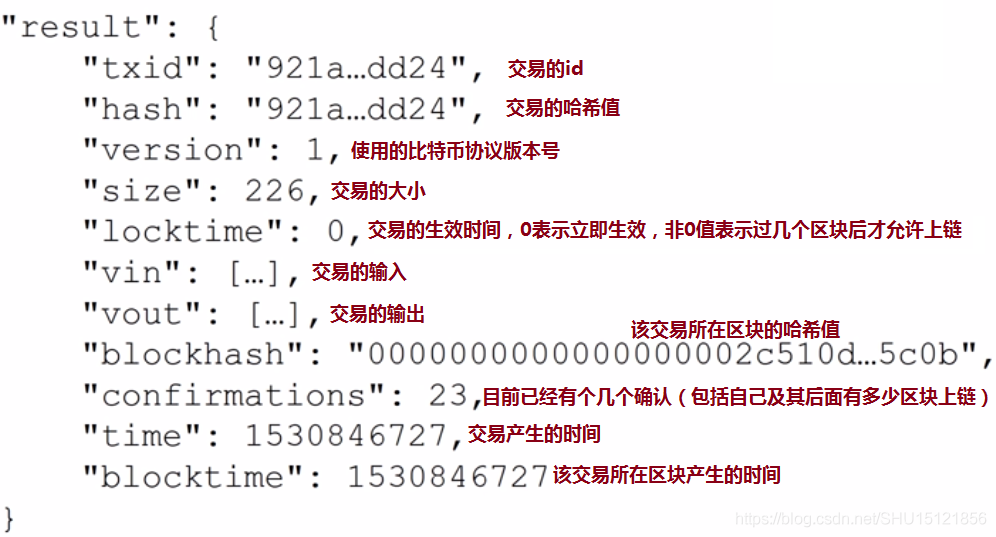


**输出脚本**

输出脚本有两行，分别对应上面的两个输出，即每个输出有自己单独的一段脚本。

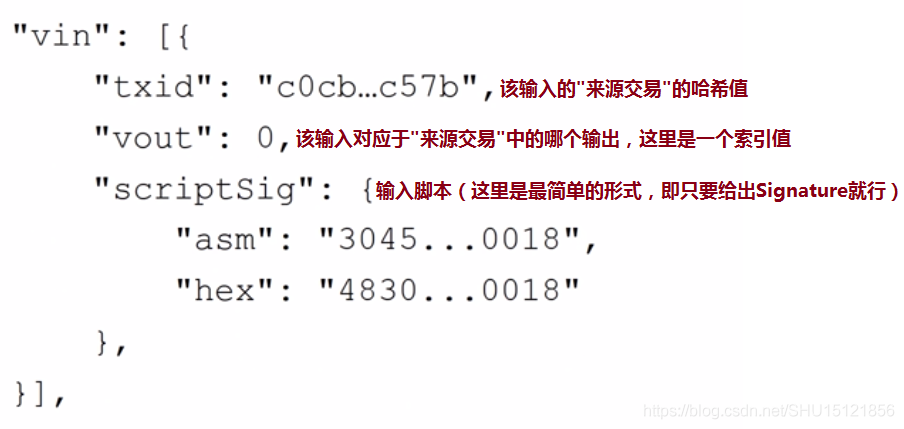


**交易的整体结构**



**交易的输入**

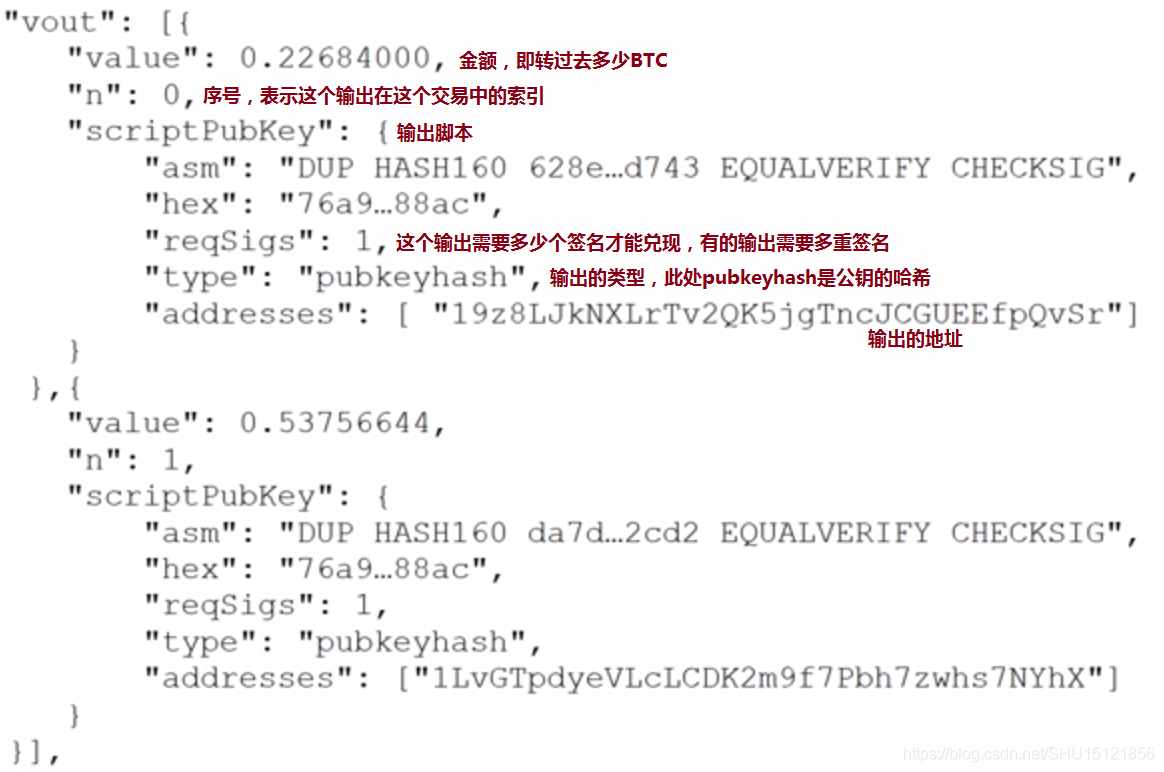
交易的输入是一个列表，这个例子中输入只有一个，所以这个列表也就只有一个列表项。



如果一个交易有多个输入，那么每个输入都要指明来源，并给出签名。

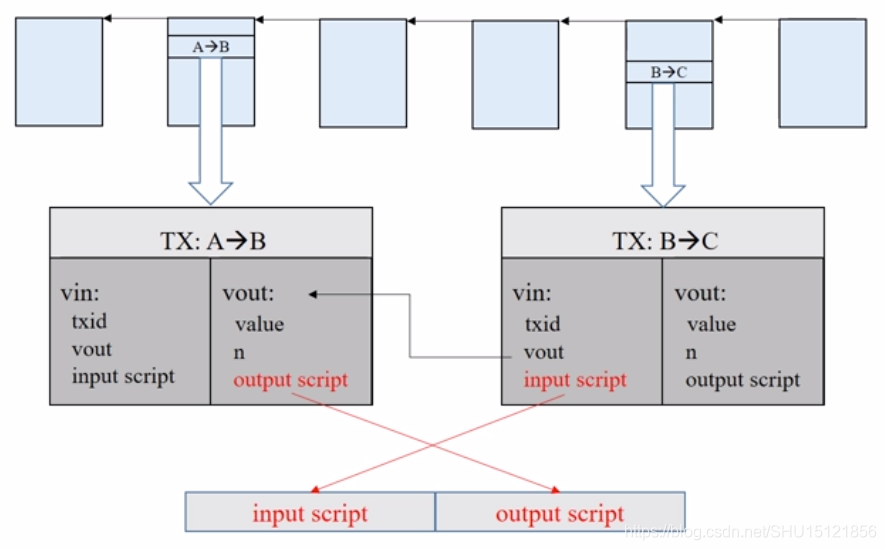
**交易的输出**

交易的输出也可以有多个，形成列表。



**一个直观的例子**

下图中两个交易分属两个区块，中间隔了两个区块，B转给C的这个交易的BTC的来源是前面A转给B的这个交易。所以**右边这个交易中的相应输入的txid是左边这个交易的id，右边这个交易中的vout指向的是左边这个交易的对应输出**。



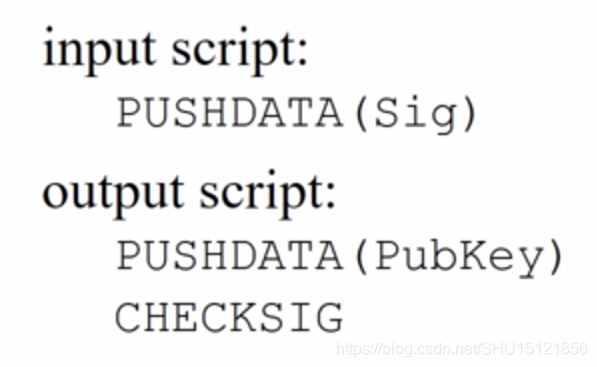
在早期的比特币系统中，要验证这个交易的合法性，就要把B->C这个交易的输入脚本，和A->B这个交易的输出脚本拼在一起执行，看看能不能执行通过。后来，出于安全因素的考虑，这两个脚本改为分别执行，首先执行输入脚本，如果没有出错，那么再执行输出脚本，如果能顺利执行，并且最后得到非零值（true），那么这个交易就是合法的。

如果一个交易有多个输入，**每个输入脚本都要去找到前面特定区块中所对应的输出脚本，匹配之后来进行验证**。全部验证通过后，这个交易才是合法的。

二、输入/输出脚本的几种形式

1. P2PK(**Pay to Public Key**)

输入脚本中直接给出收款人的签名（收款人用自己的私钥对输入脚本所在的整个交易的签名），输出脚本中直接给出收款人（注意在这里肯定要是同一个人）的公钥，最后的CHECKSIG是检查签名时用的指令。

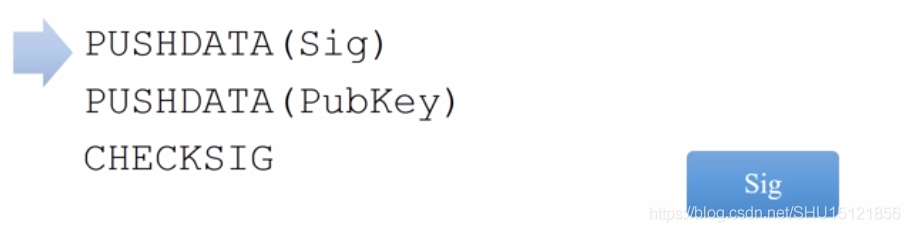


这种形式的脚本是最简单的，因为Public Key是直接在上一区块的输出脚本中给出的。

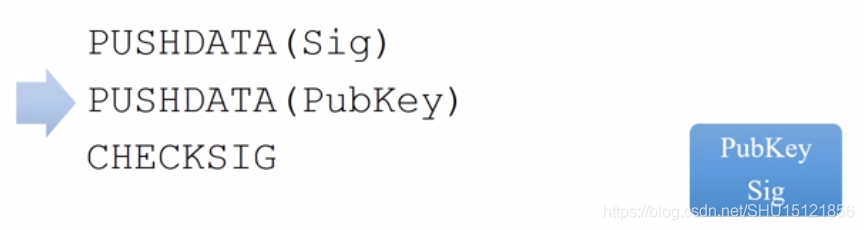
**执行情况**

一共三条语句，从上往下执行：

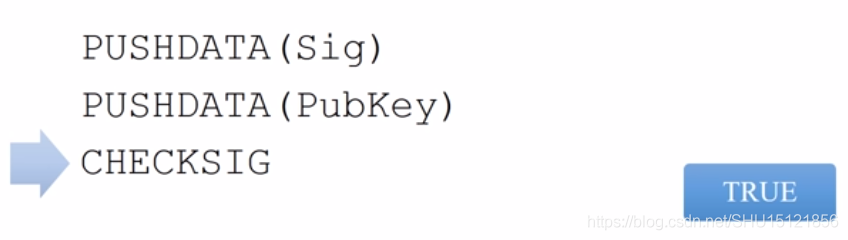
第一条语句，将输入脚本中的收款方签名压入栈：



第二条语句，将输出脚本中的收款方公钥压入栈：

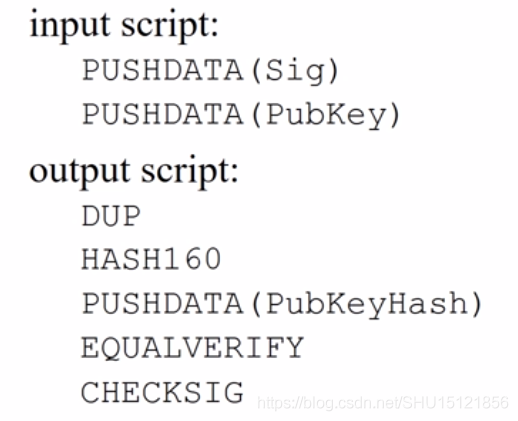


第三条语句，弹出栈顶的两个元素，用公钥PubKey检查一下签名Sig是否正确。如果正确，返回True，说明验证通过：



1. P2PKH（Pay to Public Key Hash）

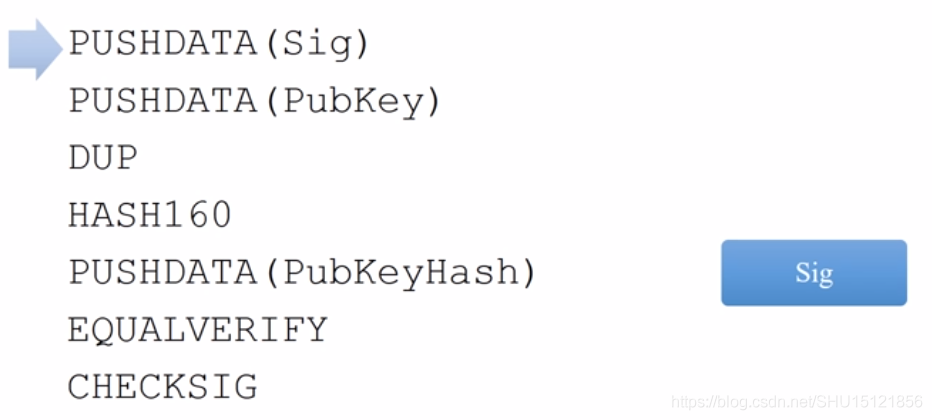
P2PKH的输出脚本中没有给出收款人的公钥，而是公钥的哈希值。在输入脚本中给出了这个人的公钥（也就是既要给出公钥又要给出签名）。其它的都是一些操作指令，后面详细学习它们是怎么运行的。P2PKH是最常用的一种形式。



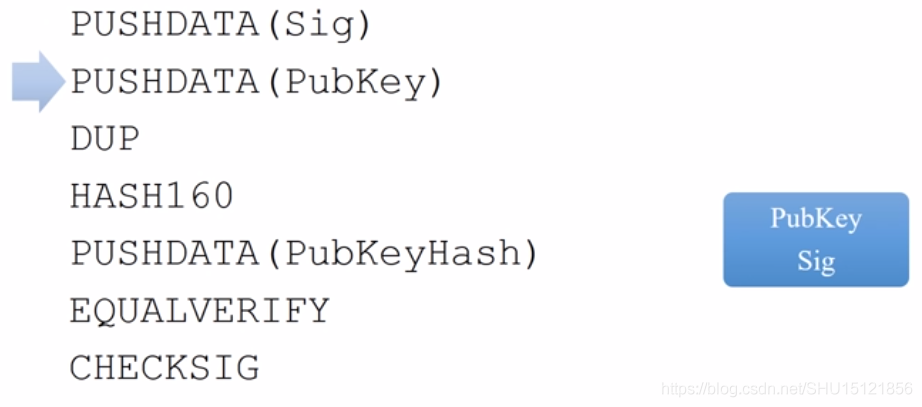
**执行情况**

一共七条语句，从上往下执行：

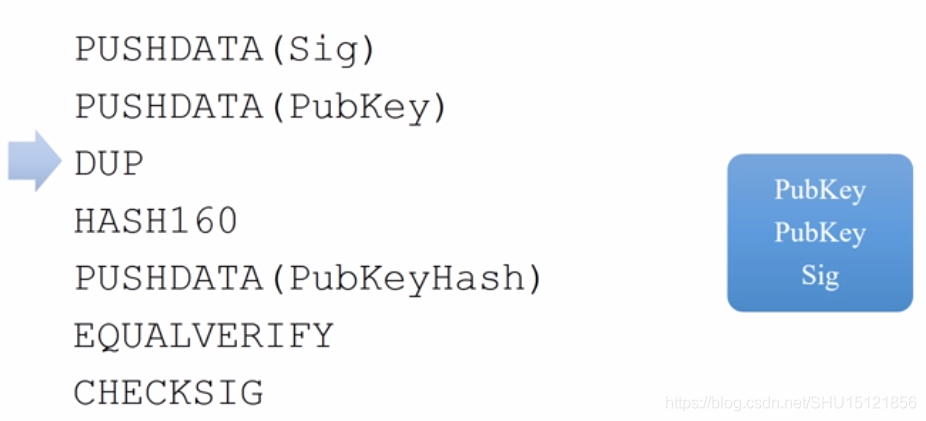
第一条语句，将输入脚本中的收款方签名压入栈：



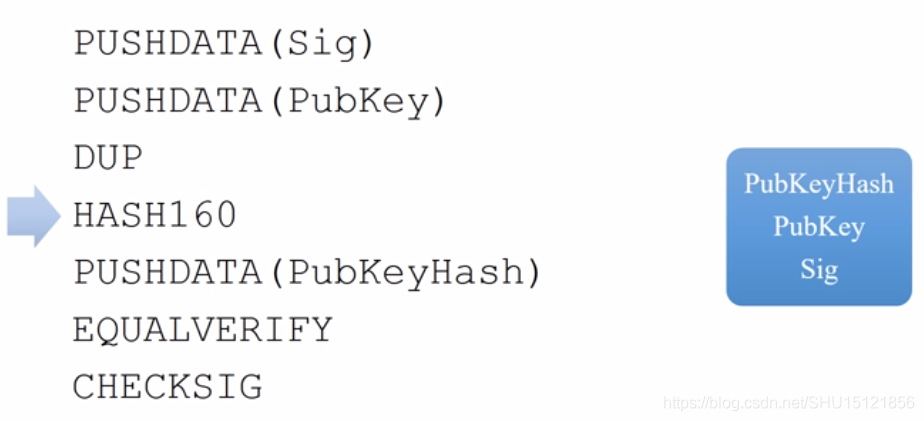
第二条语句，将输入脚本中的收款方公钥压入栈：



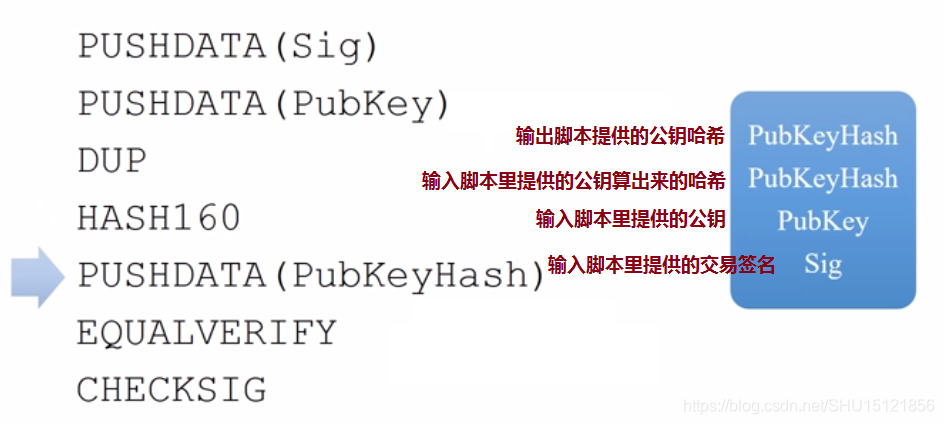
第三条语句，将栈顶元素复制一遍（所以又压入了一次公钥）：



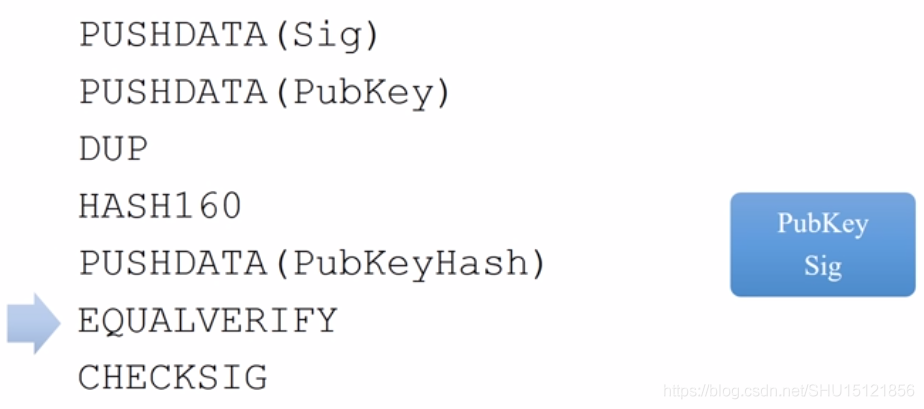
第四条语句，将栈顶元素取出来取哈希，再将得到的哈希值压入栈（也就是将栈顶的公钥变成了其哈希值）：



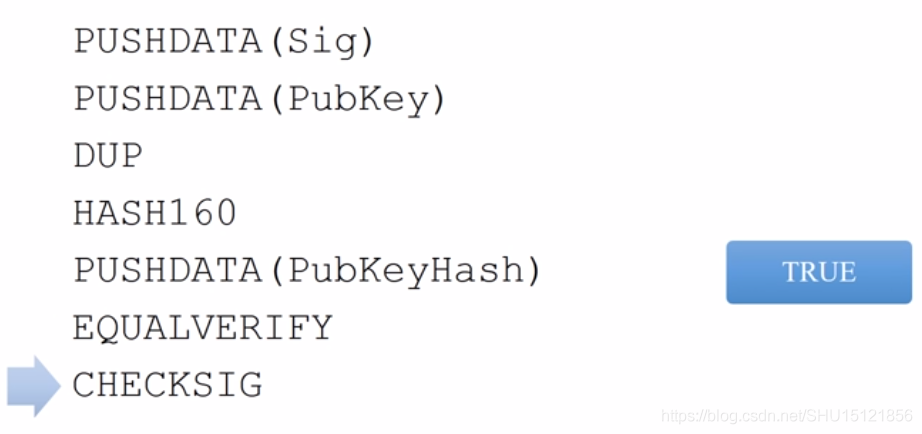
第五条语句，将输出脚本中提供的公钥的哈希值压入栈：



第六条语句，弹出栈顶的两个元素，比较它们是否相等（防止有人用自己的公钥冒充币的来源的交易的收款人的公钥）：

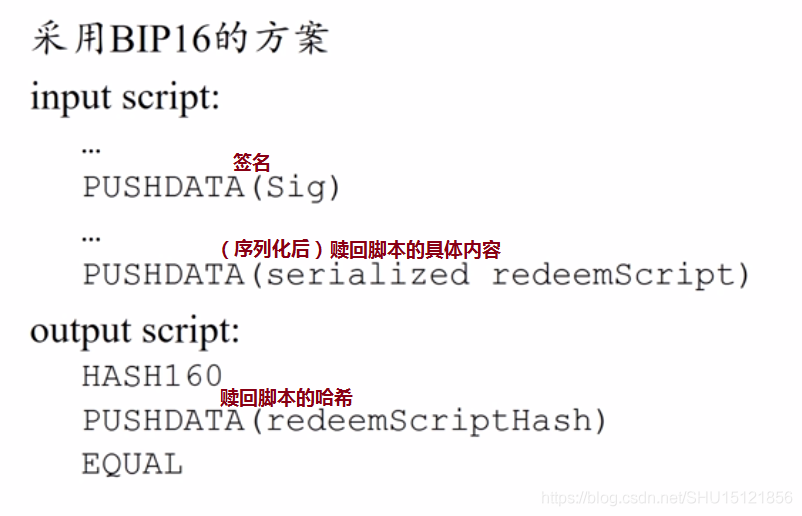


第七条语句，弹出栈顶的两个元素，用公钥PubKey检查一下签名Sig是否正确。如果正确，返回True，说明验证通过：



1. P2SH（Pay to Script Hash）

这是最复杂的一种形式，这种形式下输出脚本给出的不是收款人的公钥的哈希，而是收款人提供的赎回脚本（Redeem Script）的哈希。将来要花这个输出脚本的BTC的时候，相应交易的输入脚本要给出赎回脚本的具体内容，同时还要给出让赎回脚本能正确运行所需要的签名。



**两步验证**

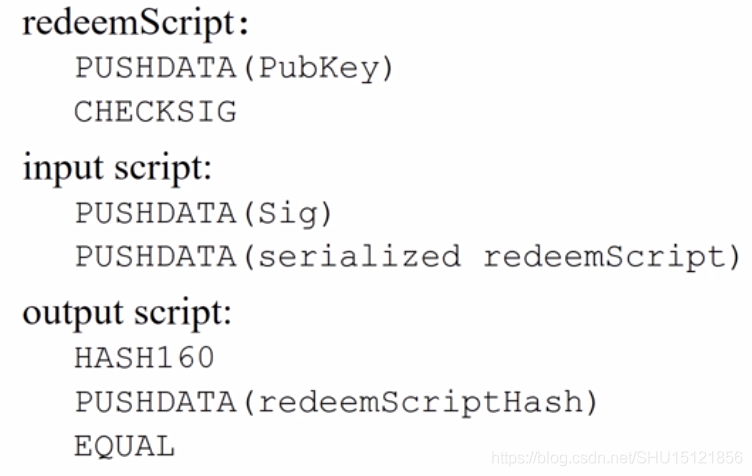
在验证时，一方面，验证输入脚本给出的赎回脚本内容，是否和对应输出脚本给出的赎回脚本哈希值相匹配；另一方面，反序列化并执行赎回脚本，以验证输入脚本给出的签名是否正确。

**赎回脚本的形式**

* P2PK形式
* P2PKH形式
* 多重签名形式

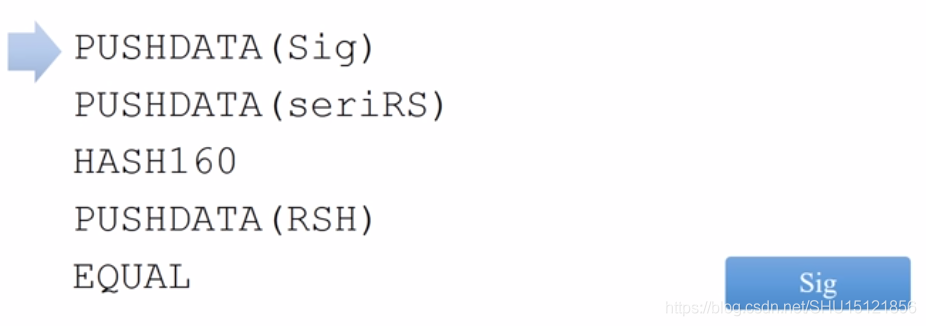
1. 用P2SH实现P2PK的功能

赎回脚本中给出公钥，输入脚本中给出交易签名和赎回脚本具体内容，输出脚本中给出了赎回脚本的哈希值。

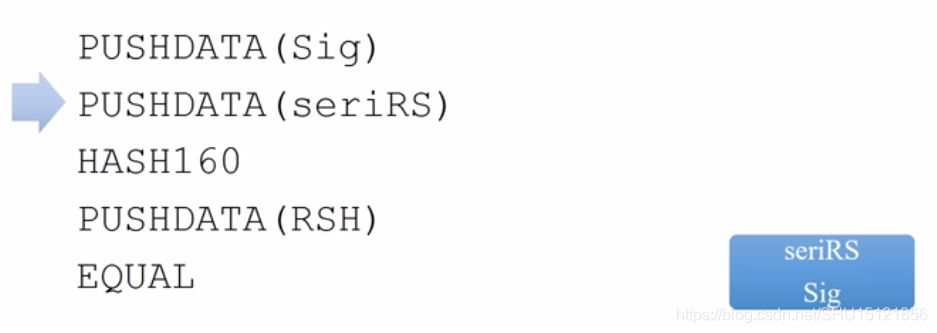


第一阶段的验证，先验证输入脚本和输出脚本在一起执行的结果:

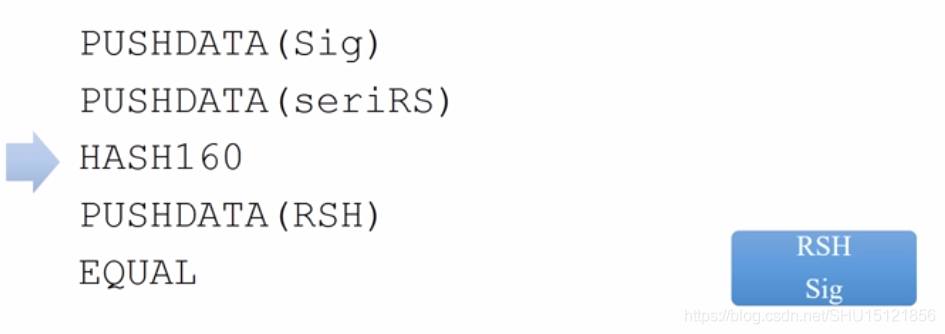
第一步，将输入脚本中的交易签名压入栈：



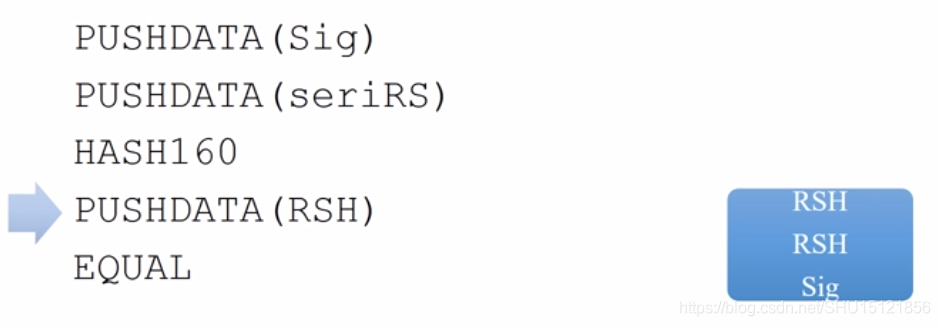
第二步，将输入脚本中给出的赎回脚本压入栈：



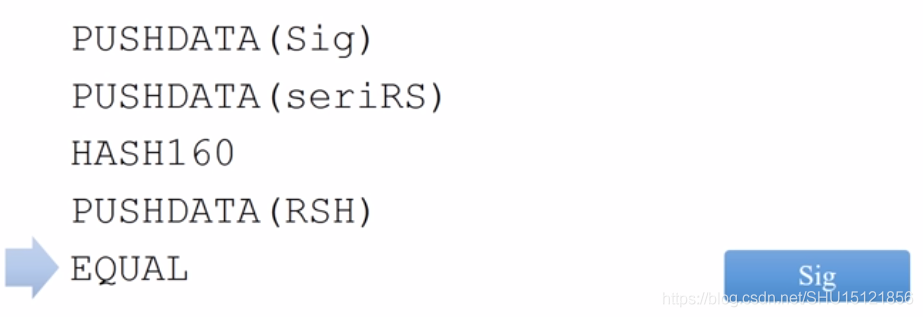
第三步，弹出栈顶元素取哈希再压栈，也就得到了赎回脚本的哈希（Redeem Script Hash）：



第四步，将输出脚本中给出的赎回脚本的哈希值压入栈：

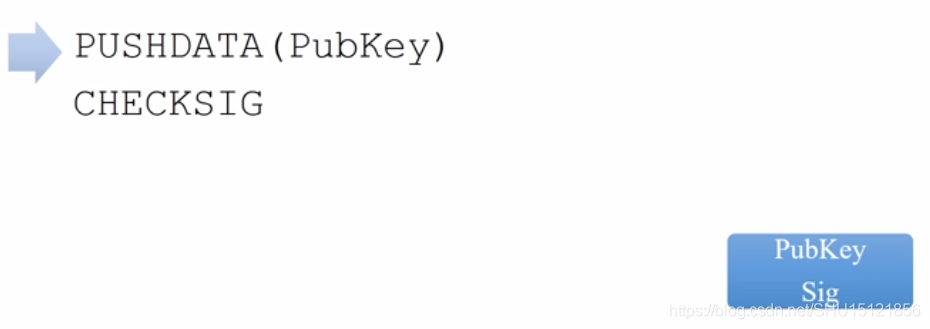


第五步，比较栈顶两个元素是否相等，相当于用之前的输出脚本给出的赎回脚本哈希，验证了输入脚本提供的赎回脚本是否是正确的：



第二阶段的验证，是对输入脚本提供的赎回脚本的验证，首先要将其反序列化，得到可以执行的赎回脚本。然后执行这个赎回脚本:

第一步，将赎回脚本中写死的公钥压入栈：



第二步，验证输入脚本中给出的交易签名的正确性。验证通过就会返回True

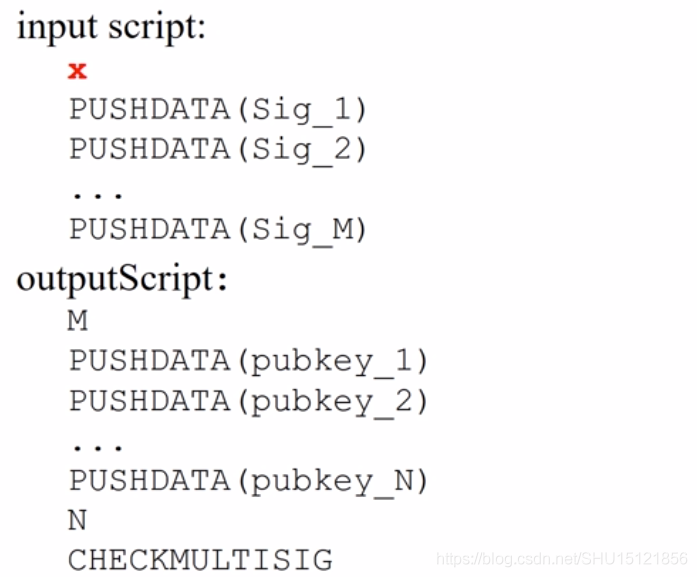
注：P2SH在最初版本的比特币系统中是没有的，后来通过软分叉的形式加进去了，它常用的一个场景就是多重签名。

三、多重签名

比特币系统中一个交易输出可能要求使用它的交易输入提供多个签名，才能把BTC取出来。比如某个公司可能要求5个合伙人中的任意三个提供签名，才能把公司的钱转走。这样设计不但为私钥的泄露提供了一定安全性保护，也为私钥的丢失提供了一定的容错性。

* 1. 最早的多重签名

最早的多重签名是通过比特币脚本中的CEHCKMULTISIG操作来实现的，输出脚本中指定N个公钥，同时指定一个不超过N的阈值M，输入脚本中只要提供任意M个签名，就能够通过验证。

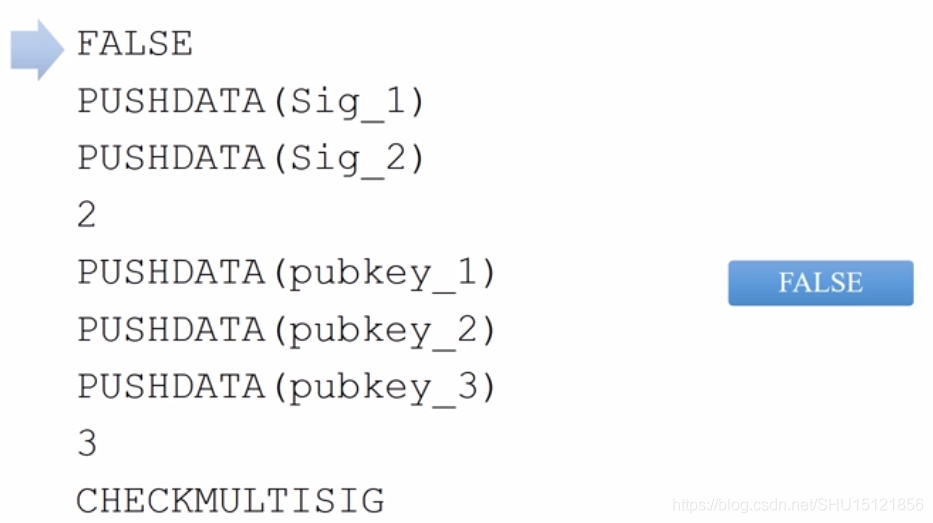


注意到图中有一个红叉，这是因为比特币系统中的CEHCKMULTISIG操作的实现有一个bug，这个bug会导致多从堆栈中弹出一个元素，因为这是一个去中心化的系统，这个bug到现在已经没法修复了，要改只能去硬分叉，代价很大。这个红叉的意思也就是在输入脚本里往栈中添加一个没用的元素，这样来抵消掉这个bug的影响。

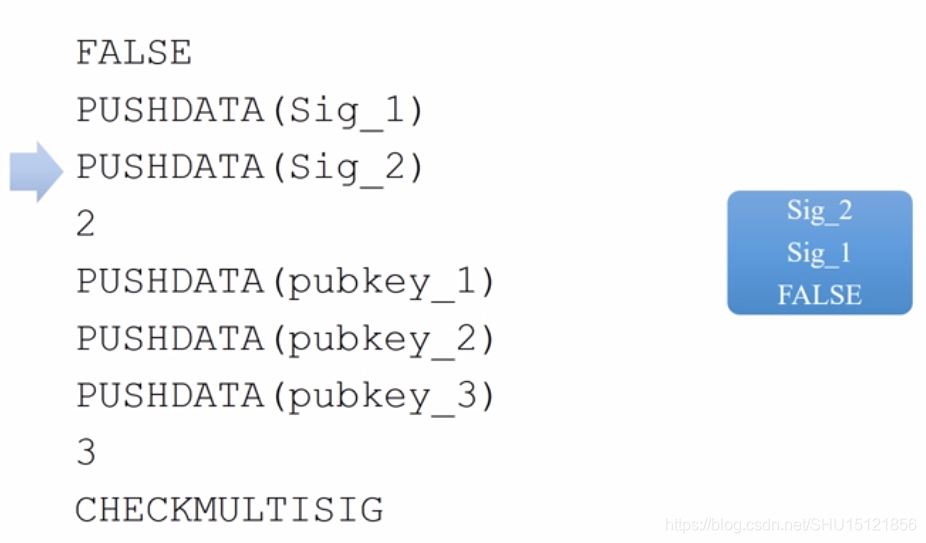
另外，给出的M个签名的相对顺序，要和对应的输出脚本中N个公钥中对应公钥的相对顺序一致才行。

**执行情况**

第一步，将输入脚本中的多余元素（前述的红叉）压栈：

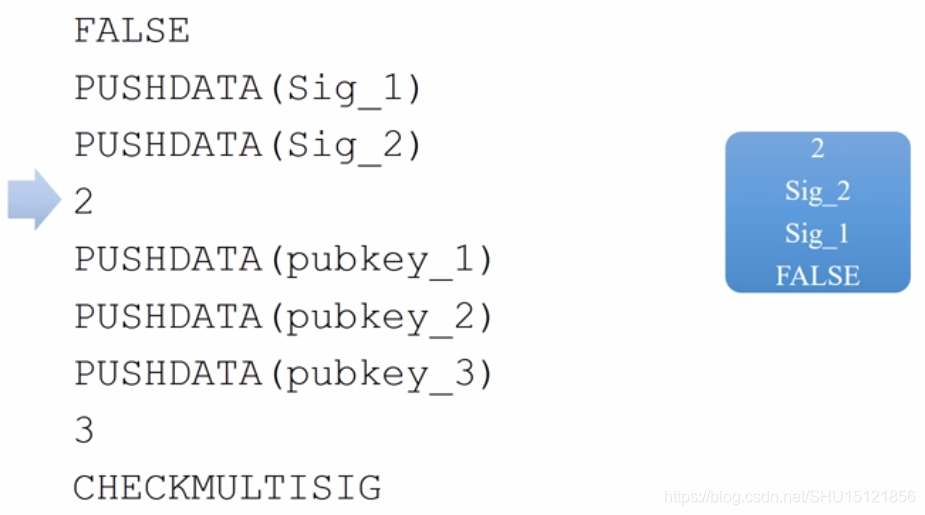


第二步，将输入脚本里的M个签名依次压入栈中（这里M=2）：

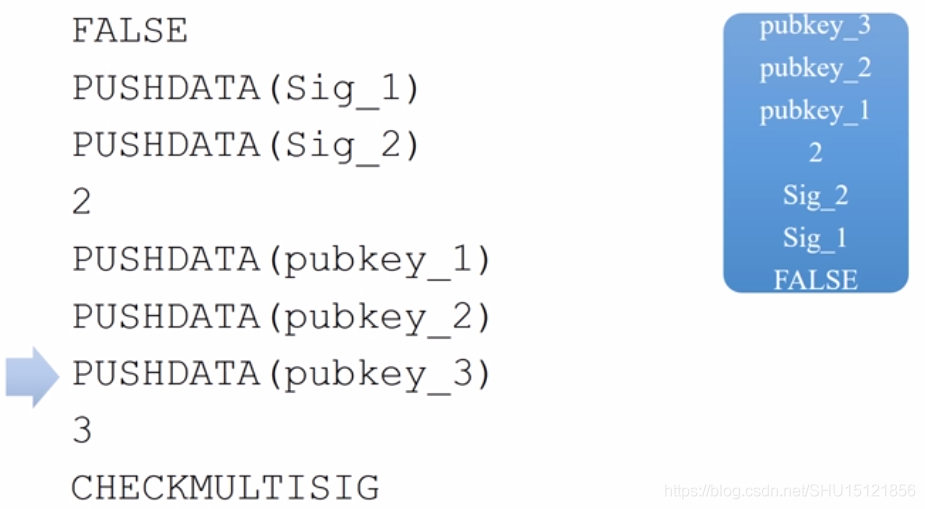


|  |
| --- |
| 输入脚本到此就执行完了。 |

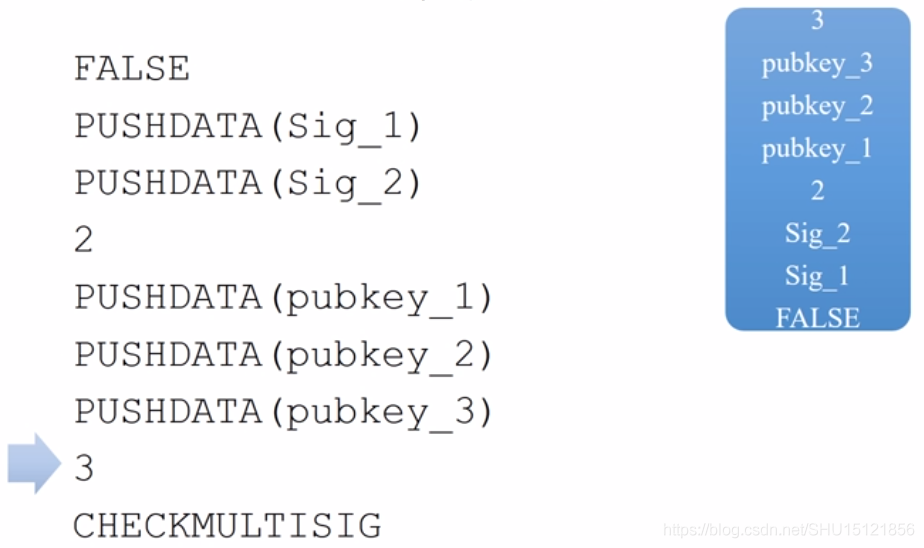
第三步，将输出脚本中给定的阈值M压栈：



第四步，将输出脚本中给定的N个公钥压栈：



第五步，将输出脚本中给定的公钥数N压栈：



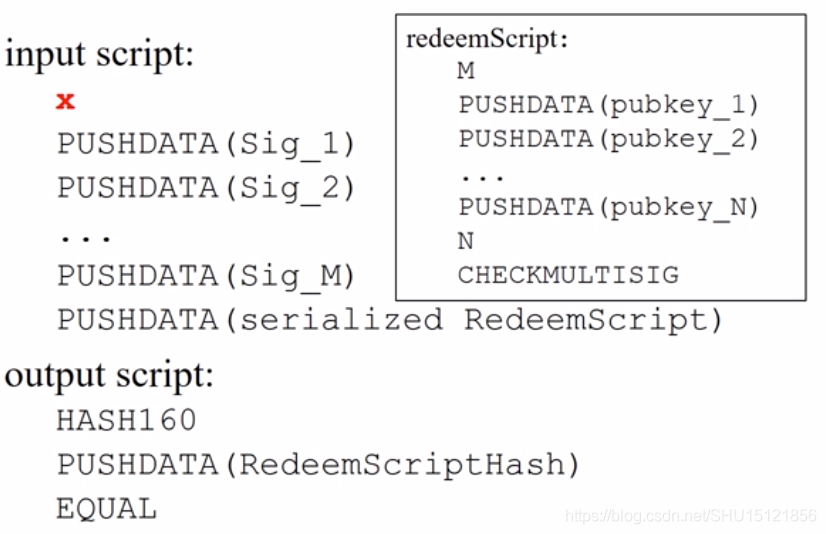
第六步，执行CEHCKMULTISIG，以检查堆栈中是否按顺序包含了N个签名中的M个。

注意，这是**最早的多重签名，并没有用到P2SH**，就是用比特币脚本中原生的CEHCKMULTISIG实现的。

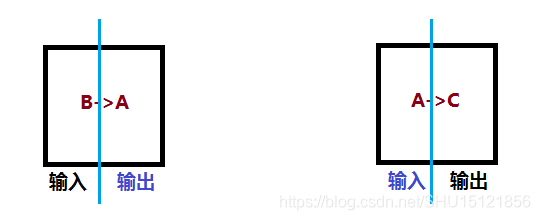
这样在实际使用时有些不方便的地方，例如电商网站开通了比特币支付渠道，但要求要有5个合伙人中3个人的签名才能把BTC转走。但这样做之后，用户在BTC支付的时候，生成的转账交易里也要给出5个合伙人的公钥，同时还要给出N和M的值。而这些公钥，以及N和M的值就要电商网站公布给用户，而且不同的电商网站规则也不一样，这就让用户生成转账交易变得不方便。因为这些**复杂性都暴露给用户**了。

* 1. 用P2SH实现多重签名

相比前面的实现，这样的本质是将复杂性从输出脚本转移到了赎回脚本中，输出脚本只需要给出赎回脚本的哈希值就行了。N个公钥以及N、M的值都在赎回脚本中给出来，而赎回脚本由输入脚本提供，这样也就和支付给它的用户们隔离开了。



例如，下面B是用户，A是电商平台，C是A要把赚到的钱转出去时候转给的账户：



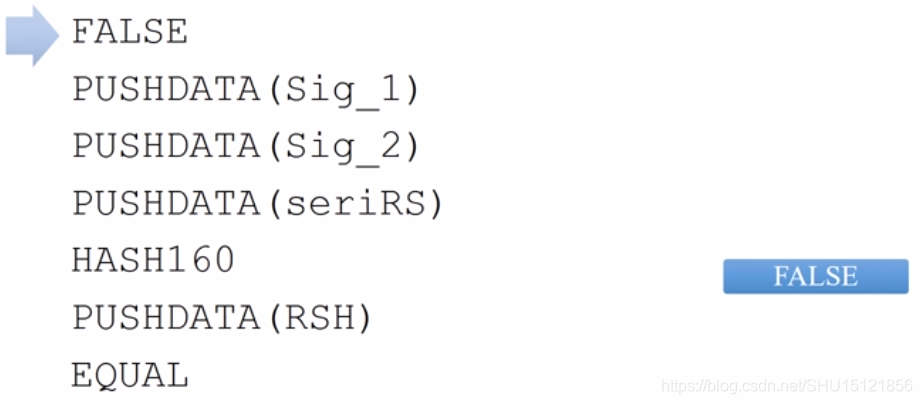
B要支付给电商平台A时，不需要A的赎回脚本，只要在输出脚本中写好A的赎回脚本的哈希值（RSH）就可以了(输入脚本由电商平台A提供)。在这种模式下，电商网站只需要在网站上公布赎回脚本的哈希值，用户生成转账交易时，把这个哈希值包含在输出脚本里就可以了，至于电商网站采用什么样的签名规则对用户而言是不可见的。

从用户的角度来看，采用这种P2SH的支付方式，和采用上节课学的P2PKH支付方式没有多大区别，只不过输出脚本中的是赎回脚本的哈希值而不是公钥的哈希值罢了（输出脚本写法上也有一些区别，见各自的指令）。

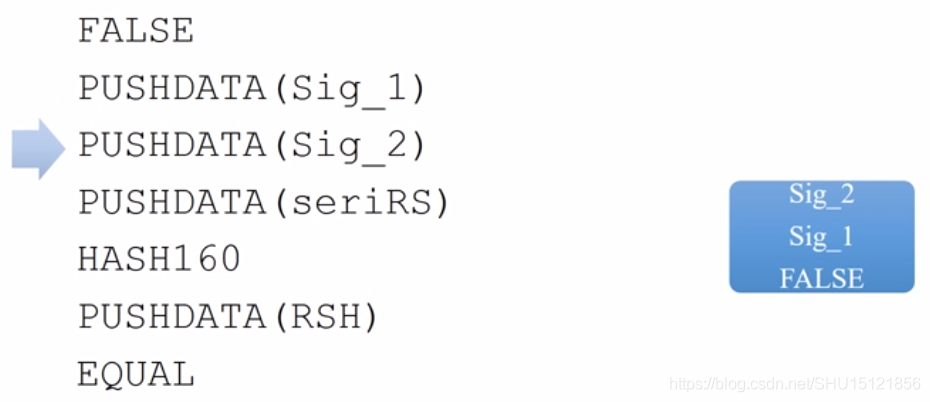
输入脚本就是电商网站要把这笔BTC转出去时候用的，这种方式下输入脚本要包含M个签名，以及赎回脚本的序列化版本。如果电商将来改变了采用的多重签名规则，就只需要改变一下赎回脚本的内容和输入脚本中的内容，然后把新的赎回脚本的哈希值公布出去就可以了。对用户而言也只是付款时候输出脚本中要包含的哈希值发生了变化。

**输入/输出脚本执行情况**

第一步，将红叉占位元素压栈：



第二步，将输入脚本中的M个签名压栈：

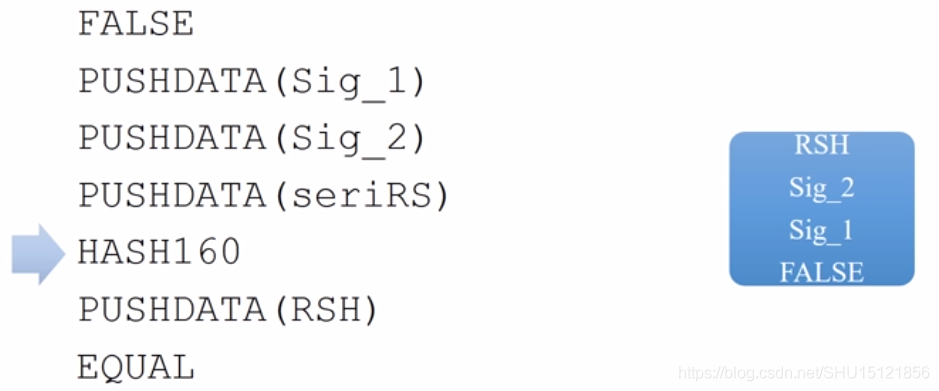


第三步，将输入脚本中保存的序列化的赎回脚本压栈：

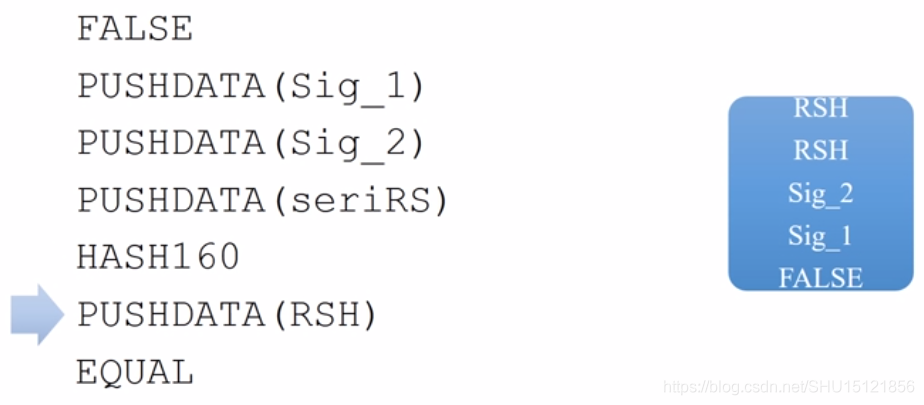


|  |
| --- |
| 输入脚本到此就执行完了。 |

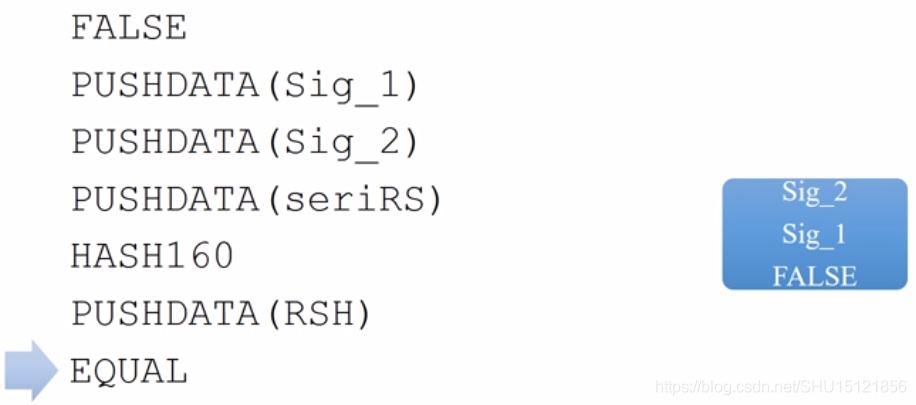
第四步，弹出栈顶元素取哈希再压栈，即将栈顶的赎回脚本取哈希：



第五步，将输出脚本中给出的赎回脚本哈希值（RSH）压栈：



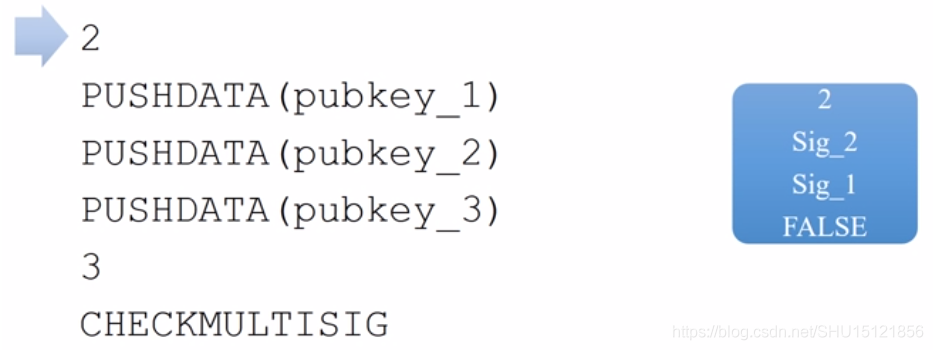
第六步，判断栈顶两个元素是否相等，即判断一下计算出的赎回脚本哈希和给定的赎回脚本哈希是否相等：



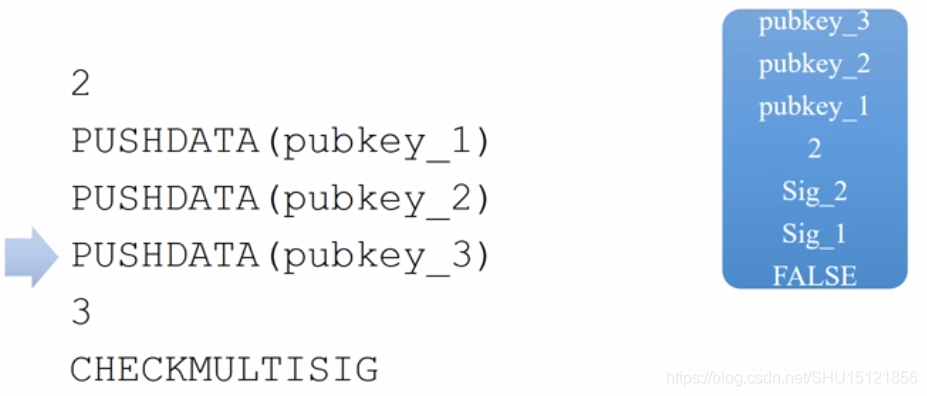
|  |
| --- |
| 输出脚本到此就执行完了，也即第一阶段的验证做完了。 |

**赎回脚本执行情况**

第一步，将阈值M压栈：



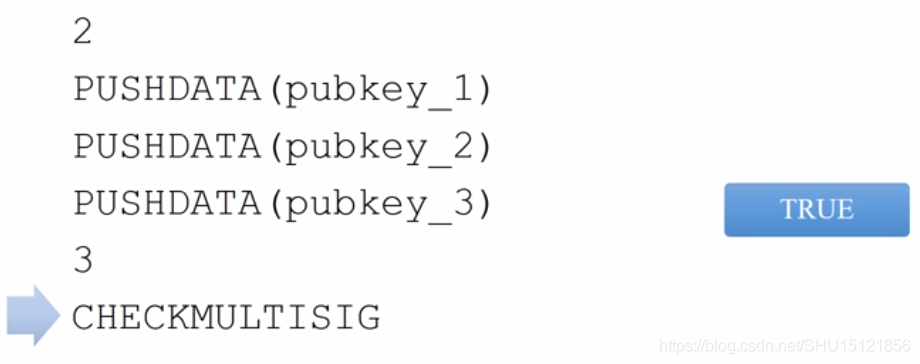
第二步，将N个公钥压栈：



第三步，将给定的公钥数N压栈：

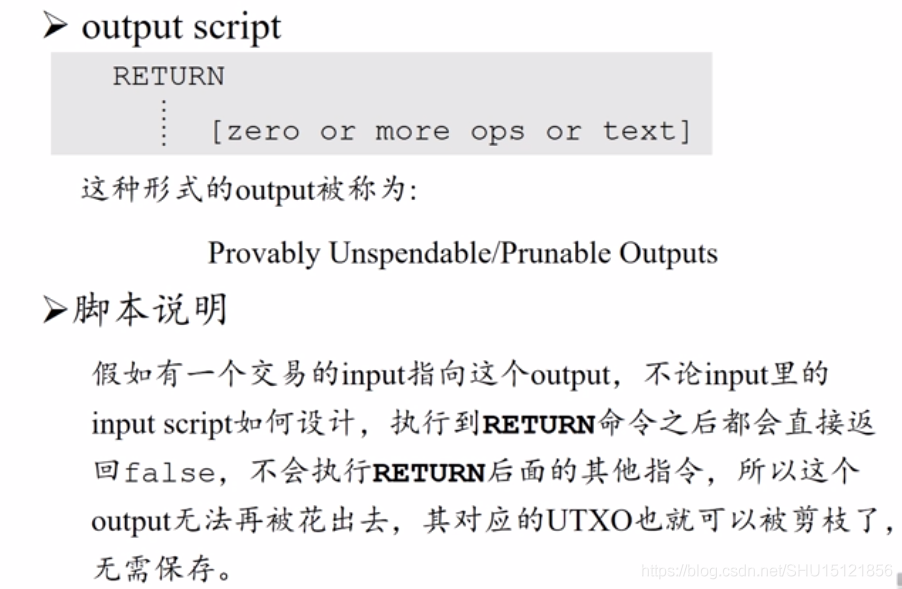


第四步，使用CEHCKMULTISIG操作检查多重签名的正确性：



四、**Proof of Burn**：销毁比特币

这是一种特殊的输出脚本，执行到RETURN语句就会出错，然后验证就会终止，后面的语句完全没有机会执行。



为什么要设计这样的输出脚本？这样的输出BTC永远都花不出去。这是用来证明销毁比特币的一种方法。

为什么要销毁比特币？一般有两种应用场景：

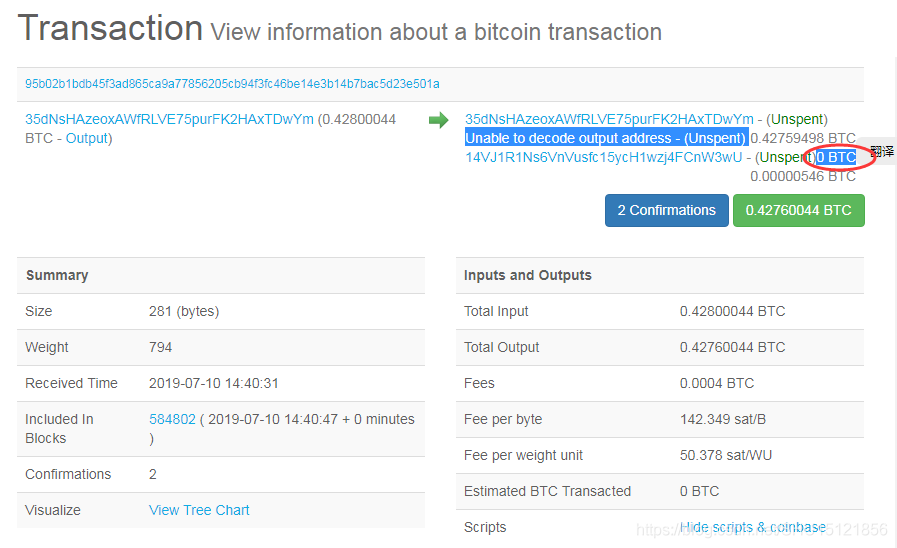
1. 一些小的加密货币（AltCoin:Alternative Coin），要求销毁一定数量的比特币可以得到一定数量的这种币。这时Proof of Burn就可以证明自己销毁了这些比特币。
2. 往区块链里写入一些内容。因为区块链是不可篡改的账本，有人就利用这个特性向其中写入一些需要永久保存的内容。比如第一节课学的digital commitment，即需要证明自己在某一时间知道某些内容。例如某些知识产权保护，可以将知识产权取哈希之后，将哈希值放在这种输出脚本的RETURN语句的后面。反正哈希值很小，而且哈希值没有泄露原来的内容，将来出现纠纷时，再将原来的内容公布出去，大家在区块链上找到这个交易的输出脚本里的哈希值，就可以证明自己在某个时间点已经掌握了这些知识了。

|  |
| --- |
| 对于上面说的第2种应用场景，回想在前面学习到铸币交易时，铸币交易的CoinBase域也可以随便写什么内容，为什么不在那里写呢？这种方法很难，必须要获得记账权，而且是在CoinBase域设定好内容的情况下，去获得记账权。根本来说，是因为发布交易不需要有记账权，但发布区块需要取得记账权。 |

任何用户都可以用Proof of Burn的方法，销毁极少量的比特币，换取向比特币系统的区块链中写入一些内容的机会。

**例子**

没有销毁比特币，仅仅支付了交易费，也可以向区块链中写入内容：



因为输出永远不会被花出去，所以不用保存在UTXO里面，这对全节点是很友好的。

总结

比特币系统中使用的脚本语言很简单，它也不是图灵完备的语言，甚至不支持循环，这样设计也有其用意，不支持循环也就不会有死循环。后面学的以太坊的脚本语言就是图灵完备的，这样就靠其它机制来防止进入死循环等。

比特币的脚本语言针对比特币应用场景做了很好的优化，如检查多重签名时的CHECKMULTISIG操作一条就能实现，这是其强大之处。