8.比特币使用的脚本与原理

如图(第15秒)是比特币的一个交易实例。该交易有一个输入两个输出。左上角写着output，其实是这个交易的输入。右边两个输出，上面unspent即没有花出，下面spent表示已花出。该交易已经收到了23个确认，所以回滚的可能性很小了。

下面是这个交易的输入输出脚本，输入脚本包含两个操作，分别把两个很长的数压入栈里。比特币使用的脚本语言是非常简单的，唯一能访问的内存空间就是一个堆栈。不像通用的编程语言，像C语言C++那样有全局变量、局部变量、动态分配的内存空间，它这里就是一个栈，所以叫做基于栈的语言。这里输出脚本有两行，分别对应上面的两个输出。每个输出有自己单独的一段脚本。

如图(第1分第40秒)是交易的具体内容。首先看交易的一些宏观信息。第一行:transaction ID，第二行hash，该交易的哈希值。第三行:使用的比特币协议的版本。第四行:该交易的大小。第五行:用来设定交易的生效时间。此处的0表示立即生效。绝大多数情况下，locktime都是0。如果是非零值，那么该交易要过一段时间才能生效。比如要等10个区块以后才能被写入区块链里。第六行第七行的vin、vout是输入输出部分，后面会详细讲解。第八行是这个交易所在区块的哈希值。第九行:该交易已经有多少个确认信息。第十行是交易产生的时间，第十一行是这个区块产生的时间。(time 和block time都是指很早的一个时间到现在过了多少秒)

如图(第3分第32秒)是交易的输入结构。一个交易可以有多个输入，在这个例子中只有一个输入。每个输入都要说明该输入花的币是来自之前哪个交易的输出，所以前两行给出输出币的来源。第一行:之前交易的哈希值。vout表示这个交易里的第几个输出。所以这里表示花的币来自于哈希值为c0cb...c57b的交易中第0个输出。接下来是输入脚本，输入脚本最简单的形式就是给出signification就行了，证明你有权利花这个钱。(后面的PPT中scriptsig就写成input script输入脚本)。如果一个交易有多个输入，每个输入都要说明币的来源，并且要给出签名，也就是说比特币中的一个交易可能需要多个签名。

如图(第5分)是交易的输出，也是一个数组结构。该例子中有两个输出，value是输出的金额，就是给对方转多少钱，单位是比特币，即0.22684个比特币。还有的单位是satoshi(一聪)，是比特币中最小的单位。1比特币=10的8次方聪。n是序号，表示这是这个交易里的第几个输出。

scriptpubkey是输出脚本，后面都写成output script。输出脚本最简单的形式就是给出一个pubkey。下面asm是输出脚本的内容，里面包含一系列的操作，在后面会详细解释。require sigs表示这个输出需要多少个签名才能兑现，这两个例子中都是只需要一个签名。type是输出的类型，这两个例子类型都是pubkeyhash，是公钥的哈希。addresses是输出的地址。

如图(第6分 第36秒)是展示输入和输出脚本是怎样执行的。在区块链第二个区块里有A→B的转账交易，B收到转来的钱后，又隔了两个区块，把币又转给了C。所以B→C交易的txid、vout是指向A→B交易的输出。而要验证交易的合法性，是要把B→C的输入脚本，跟A→B交易的输出脚本拼接在一起执行。

如图(第7分 第40秒)这里有个交叉，前面交易的输出脚本放在后面，后面交易的输入脚本放在前面。在早期的比特币实践中，这两个脚本是拼接在一起，从头到尾执行一遍。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，如果没有出错就再执行输出脚本。如果能顺利执行，最后栈顶的结果为非零值，也就是true，那么验证通过，这个交易就是合法的。如果执行过程中出现任何错误，这个交易就是非法的。如果一个交易有多个输入的话，那么每个输入脚本都要和所对应的交易的输出脚本匹配之后来进行验证。全都验证通过了，这个交易才是合法的。

如图(第8分第45秒)是输入、输出脚本的几种形式。一种最简单的形式就是P2PK(pay to public key)。输出脚本里直接给出收款人的公钥，下面一行checksig，是检查签名的操作。在输入脚本里，直接给出签名就行了。这

个签名是用私钥对输入脚本所在的整个交易的签名。这种形式是最简单的，因为公钥是直接在输出脚本里给出的。

如图(第9分第18秒)是脚本的实际执行情况。这三行是把输入脚本和输出脚本拼接起来之后的结果。第一行来自输入脚本，后两行来自输出脚本。注意，实际代码中出于安全考虑，这两个脚本实际上是分别执行的。第一行:把输入脚本提供的签名压入栈，第二条把输出里提供的公钥压入栈，第三条checksig是把栈顶的这两个元素弹出来。用公钥检查一下这个签名是否正确。如果正确，返回true，说明验证通过。否则，执行出错，这个交易就是非法的。

如图(第10分第24秒)是P2PK的一个实例。上面交易的输入脚本就是把签名压入栈，下面交易是上面交易输入的币的来源。它的输出有两行，第一行是把公钥压入栈，第二行就是checksig。这是第一种形式。

如图(第10分第52秒)是第二种形式P2PKH(pay to public key hash)，跟第一种区别是输出脚本里没有直接给出收款人的公钥，给出的是公钥的哈希。公钥是在输入脚本里给出的。输入脚本既要给出签名，也要给出公钥。输出脚本里还有一些其他操作，DUP、HASH160等等，这些操作都是为了验证签名的正确性。P2PKH是最常用的形式。

如图(第11分第37秒)是脚本的执行结果，这个是把上一页的输入脚本和输出脚本拼接之后得到的，前两条语句来自输入脚本，后面的语句来自输出脚本，还是从上往下执行。第一条语句先把签名压入栈，第二条语句把公钥压入栈。第三条语句是把栈顶的元素复制一遍，所以栈顶又多了一个公钥。HASH160是把栈顶元素弹出来，取哈希，然后把得到的哈希值再压入栈。所以栈顶变成了公钥的哈希值。

第五行是把输出脚本里提供的公钥的哈希值压入栈。这个时候栈顶有两个哈希值，上面的哈希值是输出脚本里面提供的，收款人公钥的哈希，即我发布交易时，转账的钱是转给谁的，在输出脚本里提供一个收款人的公钥的哈希。下面的哈希是指你要花这个钱时在输入脚本里给出的公钥，然后前面的操作HASH160是取哈希后得到的。倒数第二行操作的作用是弹出栈顶的两个元素，比较是否相等，即比较其哈希值是否相等。这样做的目的是防止有人莫名顶替，用自己的公钥冒充收款人的公钥。假设两个哈希是相等的，那么就从栈顶消失了。最后一条作用是用公钥检查弹出栈顶的元素是否正确。假设签名是正确的，整个脚本就顺利运行结束，栈顶留下的是true。如果执行过程任何一个环节发生错误，比如输入里给出的公钥跟输出里给出的哈希值对不上，或者是输入里给出的签名跟给出的公钥对不上，那么这个交易就是非法的。

P2PKH是最常用的脚本信息，该实例(第14分第20秒)用的就是这种脚本。输入脚本就是把签名压入栈，把公钥压入栈。下面的输出脚本复制栈顶元素，然后取哈希值，hash160。然后把公钥的哈希压入栈，最后比较栈顶的两个哈希值，检查签名。

最后一种如图(第15分第25秒)，也是最复杂的一种脚本形式，是Pay to Script Hash。这种形式的输出脚本给出的不是收款人的公钥的哈希，而是收款人提供的一个脚本的哈希，这个脚本叫redeemscript，赎回脚本。将来花这个钱时输入脚本里要给出redeemscript(这个赎回脚本的具体内容)，同时还要给出让赎回脚本能够正确运行所需要的签名。

验证时分为两部(如图第15分第40秒)，第一步验证输入脚本里给出的赎回脚本是不是跟输出脚本里给出的哈希值匹配，如果不匹配说明给出的赎回脚本是不对的，就类似于刚才讲的pay to public key hash里面给出的公钥不对一样。匹配不上说明给出的赎回脚本是不对的，那么验证就失败了。如果输入里给出的赎回脚本是正确的，那么第二步还要把赎回脚本的内容当做操作指令来执行一遍，看看最后能不能顺利执行。如果两步验证都通过了，那么这个交易才是合法的。听上去有点抽象，那么下面看一个具体的例子。

(如图第16分第47秒)用pay to script hash实现pay to public key 的功能。这里的输入脚本就是给出签名，再给出序列化的赎回脚本，赎回脚本的内容就是给出公钥，然后用checksig检查签名。下面这个输出脚本是用来验证输入脚本里给出的赎回脚本是否正确。

如图(第17分第13秒)看一下pay to script hash的执行过程。开始也是把输入脚本和输出脚本拼接在一起，前两行来自输入脚本，后面三行来自输出脚本。首先把输入脚本的签名压入栈，然后把赎回脚本压入栈，然后是取哈希的操作，得到赎回脚本的哈希。这里RSH是指redeem script hash，赎回脚本的哈希值。接下来还要把输出脚本里给出的哈希值压入栈，这时栈里就有两个哈希值了。最后用equal比较这两个哈希值是否相等，如果不等就失败了。假设相等，那这两个哈希值就从栈顶消失了，到这里第一阶段的验证就算结束了，接下来还要进行第二个阶段的验证。

如图(第18分第28秒)第二个阶段首先要把输入脚本提供的序列化的赎回脚本进行反序列化，这个反序列化的操作在PPT上并没有展现出来，这是每个节点自己要完成的。然后执行赎回脚本，首先把public key压入栈，然后用checksig验证输入脚本里给出的签名的正确性。验证痛过之后，整个pay to script hash才算执行完成。

有人可能会问:干脆用pay to public key就行了，搞这么复杂干嘛?为什么非要把这些功能嵌入到赎回脚本里面?对于这个简单的例子来说确实是复杂了，但pay to script hash它的常见的应用场景是对多重签名的支持。

比特币系统中一个输出可能要求多个签名才能把钱取出来，比如某个公司的账户，可能要求五个合伙人中任意三个人签名才能把公司账户上的钱取走，这样为私钥的泄露提供了一些安全的保护。

比如说有某个合伙人私钥泄露出去了，那么问题也不大，因为还需要两个人的签名才能把钱取走。这同时也为私钥的丢失提供了一些冗余，即使有两个人把私钥忘掉了，省下的三个人依然可以把钱取出来，然后转到某一个安全的账户。

以上的功能是通过check multisig来实现的。 如图(第21分)，输出脚本里给出N个公钥，同时指定一个预值M。输入脚本只要提供接N个公钥对应的签名中任意M个合法的签名就能通过验证。

比如刚才举的例子中，N=5，M=3，五个合伙人中任意三个的签名都可以，输入脚本的第一行有一个红色的“✗”，这是什么意思呢?

比特币中check multisig的实现，有一个bug，执行的时候会从堆栈上多弹出一个元素，这个就是它的代码实现的一个bug。这个bug现在已经没有办法改了，因为这是个去中心化的系统，要想通过软件升级的方法去修复这个bug代价是很大的，要改的话需要硬分叉。所以实际采用的解决方案，是在输入脚本里，往栈上多压进去一个没用的元素，第一行的“✗”就是没用的多余的元素。另外需要注意给出的M个签名的相对顺序，要跟它们在N个公钥中的相对顺序是一致的才行。

如图(第22分第48秒)是check multisig的执行过程。这个例子假设三个签名中给出两个就行。图中可以看到这两个签名给出的相对顺序也是跟它们在公钥中的顺序是一样的。在公钥当中，第一个公钥排在第二个公钥前面。那么给出这两个签名的时候也是第一个签名排在第二个的前面。

第一行的false就是前面说的多余的元素。首先把多余的元素压入栈里，然后把两个签名依次压入栈，这个时候输入脚本就执行完了。接下来的输出脚本里把M的值，即预值M压入栈。然后把三个公钥压入栈，接着把N的值压入栈，最后执行check multisig，看看堆栈里是不是包含了这三个签名中的两个，如果是那么验证通过。

注意:这个过程中并没有用到pay to script hash。就是用比特币脚本中原生的check multisig来实现的。这么实现有什么问题吗？ 早期的多重签名就是这样实现的，在实际的应用当中，有一些不是很方便的地方。

比如:网上购物。某个电商用multi签名，要求有五个合伙人中任意三个人的签名才能把钱取出来，要求网上购物的用户在支付的时候生成的转账交易里给出这五个合伙人的公钥，同时要给出N和M值。在这个例子中，N=5，M=3，这些都是用户在网上购物的时候生成转账交易时输出脚本里要给出的信息，给出这五个公钥，给出N和M值。

那么用户怎么知道这些信息呢?需要购物网站在网上公布出来，比如网上可以公布我们用了多重签名，我们用的五个签名中要给出三个，这是五个公钥，然后用户生成这个转账交易的时候，就把这些信息填进去。那么不

同的电商采用的多重签名的规则是不一样的。有的电商可能是五个签名中要任意三个，有的可能要四个。这就给用户生成转账交易带来了一些不方便的地方，因为这些复杂性都暴露给用户了。

那么该如何解决?这里就要用到pay to script hash。 如图(第26分第39秒)是用pay to script hash实现的多重签名，它的本质是把复杂度从输出脚本转移到了输入脚本。现在这个输出脚本变得非常简单，只有这三行。原来的复杂度被转移到redeemscript赎回脚本里。输出脚本只要给出这个赎回脚本的哈希值就可以了。赎回脚本里要给出这N个公钥，还有N和M的值，这个赎回脚本是在输入脚本里提供的，也就是说是由收款人提供的。

像前面网上购物的例子，收款人是电商，他只要在网站上公布赎回脚本的哈希值，然后用户生成转账交易的时候把这个哈希值包含在输出脚本里就行了。至于这个电商用什么样的多重签名规则，对用户来说是不可见的，用户没必要知道。从用户的角度来看采用这种支付方式跟采用pay to public key hash没有多大区别，只不过把公钥的哈希值换成了赎回脚本的哈希值。当然，输出脚本的写法上也有一些区别，但不是本质性的。这个输入脚本是电商在花掉这笔输出的时候提供的，其中包含赎回脚本的序列化版本，同时还包含让这个赎回脚本验证通过所需的M个签名。将来如果这个电商改变了所采用的多重签名规则，比如由五个里选三个变成三个里选两个，那么只要改变输入脚本和赎回脚本的内容，然后把新的哈希值公布出去就行了。对用户来说，只不过是付款的时候，要包含的哈希值发生了变化，其他的变化没有必要知道。

如图(第29分第14秒)是具体的执行过程。这是把输入脚本和输出脚本拼接在一起后的情况，第一行的FALSE就是为了应付check multisig的bug而准备的一个没用的元素，执行的时候先把它压入栈，然后依次把两个签名压入栈，接下来是序列化的赎回脚本，目前只是把它作为数据压入栈，到这里输入脚本就执行完了。下面是输出脚本，取哈希，然后把输出脚本里提供的哈希值压入栈顶。最后判断两个哈希值是否相等，到这里第一阶段的验证就完成了。

如图(第30分第18秒)开始第二阶段的验证，把赎回脚本展开后执行。先把M压入栈，然后把三个公钥压入栈，把N压入栈，最后检查多重签名的正确性，三个里面有两个是正确的。第二阶段的验证过程跟前面直接使用check multisig的情况是类似的。

如图(第30分第52秒)是网上使用pay to script hash来做多重签名的一个实例。上面输入脚本的最后一个就是序列化的赎回脚本，反序列化之后得到的就是三个里面取两个的多重签名脚本。下面这个输出脚本的内容，跟前面讲的是一样的。现在的多重签名，一般都是采用这种pay to script hash的形式。

如图(第31分第25秒)这种脚本格式是比较特殊的，这种格式的输出脚本开头是return的操作，后面可以跟任意的内容。return操作的作用，是无条件的返回错误，所以包含这个操作的脚本永远不可能通过验证，执行到return语句，就会出错，然后执行就终止了，后面跟的内容根本没有机会执行。

为什么要设计这样的输出脚本呢？这样的输出岂不是永远花不出去吗？无论输入脚本写的是什么内容，执行到输出的return语句，它就会报错，那么这里的钱永远都花不出去。确实如此，这个脚本是销毁比特币的一种方法。

为什么要销毁比特币呢？这个一般有两种应用场景: ①有些小的币种要求销毁一定数量的比特币才能够得到这个币种，有时候把这种小币种称为AltCoin(Alternative coin)。除了比特币之外的其他小的加密货币都可以认为是Alternative Coin。比如有的小币种要求销毁一个比特币可以得到1000个小币，也就是说要用上述的方法证明已经付出了一定的代价才能够得到这个小币种

②往区块链里写入一些内容。区块链是个不可篡改的账本，有人就利用这个特性往里面添加一些需要永久保存的内容，比如第一节课讲的digital commitment。要证明在某个时间，知道某些事情。比如涉及知识产权保护的，把某项知识产权的内容取哈希之后，把哈希值放到return语句的后面，其后面的内容反正是永远不会执行的，往里面写什么都没关系。而且放在这里的是一个哈希值，不会占太大的地方，而且也没有泄露出来你知识产权的具体内容。将来如果出现了纠纷，像知识产权的一些专利诉讼，再把具体的哈希值的输入内容公布出去，证明你在某个时间点已经知道某个知识了。

这个应用场景和coinbase域相似。coinbase transaction里面有个coinbase域，在这个域里写什么内容同样是没人管的，那这里为什么不用coinbase的方法呢？coinbase还不用销毁比特币，就可以直接往里写。

coinbase的方法只有获得记账权的那个节点才能用。如果是一个全节点，挖矿挖到了，然后发布一个区块，可以往coinbase transaction 里的coinbase域写入一些内容，这是可以的。

而我们说的上述方法，是所有节点都可以用的，甚至不一定是个节点，可能就是一个普通的比特币上的一个用户，任何人都可以用这种方法去写入一些内容。发布交易不需要有记账权，发布区块才需要有记账权。任何用户都可以用这种方法销毁很少的比特币，比如0.0000001个比特币，换取往区块链里面写入一些内容的机会。其实有些交易根本没有销毁比特币，只不过支付了交易费。

下面看两个实例 如图(第37分第44秒)是一个coinbase transaction。这个交易有两个输出，第一个输出的脚本是正常的pay to public key hash，输出的金额就是得到的block reward加上transaction fee。第二个输出的金额是0，输出脚本就是刚才提到的格式:开头是return，后面跟了一些乱七八糟的内容，第二个输出的目的就是为了往区块链里写一些东西。

这种形式的脚本的一个好处是:矿工看到这种脚本的时候知道它里面的输出永远不可能兑现，所以就没必要把它保存在UTXO里面，这样对全节点是比较友好的。还有一点要说明:这个PPT当中涉及到比特币脚本的操作为了简单起见都没有加上OP前缀。比如CHECKSIG，实际上应该写成OP_CHECKSIG，CHECKMULTISIG、DUP也是如此。

比特币系统中用到的这种脚本语言是非常简单的，甚至连专门的名字都没有，它就叫比特币脚本语言(bitcoin scripting language)。后面可以看到，以太坊当中用的智能合约的语言比这个要复杂的多。比如说比特币的脚本语言不支持循环，所以有很多功能这个语言是实现不了的，这样的设计是有其用意的，不支持循环就不会有死循环，就不用担心停机问题。以太坊当中智能合约的语言表达能力很强，所以就要靠汽油费的机制来防止程序陷入死循环。

另外一方面，这个语言虽然在某些方面功能是很有限的，但是在另外一些方面它的功能却很强大，比如跟密码学相关的功能。如checkmultisig，检查多重签名用一条语句就能够完成，这个比很多通用的编程语言要方便的多。所以比特币的脚本语言虽然看上去很简单，但其实针对比特币的应用场景做了很好的优化。