用Go构建区块链——5.地址

本篇是"用Go构建区块链"系列的第五篇，主要对原文进行翻译。对应原文如下：

Building Blockchain in Go. Part 5: Addresses

1、介绍

在上篇文章中，我们已经着手实现了交易。您也了解了交易的天然属性：在比特币中，没有用户帐户，不需要也不会在任何地方存储您的个人数据（例如姓名，护照号码或身份证号）。但是仍然必须有一些东西将您标识为交易输出的所有者（即锁定在这些输出上的货币的所有者）。这就是比特币地址所需要的。到目前为止，我们已经使用任意用户定义的字符串作为地址，而且是时候去实现一个真实的地址了，就像他们在比特币中已经实现的那样。

> 这部分介绍了重要的代码更改，所以在这里解释它们是没有意义的。请参阅此页面以查看自上一篇文章以来的所有更改。

2、比特币地址

这里有个比特币地址的例子：1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa 。这是一个很早的比特币地址，据说它属于中本聪。比特币地址是公开的。如果你想将货币发送给某个人，你需要知道他们的地址。但地址（尽管是唯一的）并不能将您标识为"钱包"的所有者。事实上，这样的地址是公钥表示为人类可读的形式而已。在比特币中，您的身份是存储在您的计算机上（或存储在您有权访问的其他位置）的私钥和公钥的一对（或多对）密钥。比特币依靠密码算法的组合来创建这些密钥，并保证世界上没有其他人可以在没有物理访问密钥的情况下访问您的货币。我们来讨论一下这些算法是什么。

3、公钥加密

公钥密码算法使用密钥对：公钥和私钥。公钥不敏感，可以向任何人透露。相反，不应该公开私钥：除了所有者之外，没有人能够访问它们，因为私钥是作为所有者的标识符的私钥。你是你的私钥（当然是加密货币的世界）。

实质上，比特币钱包只是一对这样的密钥。当您安装钱包应用程序或使用比特币客户端生成新的地址时，会为您生成一对密钥。在比特币中，谁控制了私钥，谁就控制了所有进入这个钱包的货币。

私钥和公钥只是随机的字节序列，因此它们不能被打印在屏幕上，也不能被人读取。这就是为什么比特币使用算法将公钥转换为可读的字符串的原因。

> 如果您曾经使用比特币钱包应用程序，则可能会为您生成助记符密码短语。这些短语被用来

> 代替私钥，并且可以用来生成它们。该机制在BIP-039中实现。

好的，我们现在知道识别比特币用户的标识。但是比特币如何确认交易输出（以及存储在其上的货币）的所有权呢?

4、数字签名

在数学和密码学中，有一个数字签名的概念 - 保证：

1. 该数据在从发送者转移到接收者时，数据不会被修改;
2. 该数据是由某个确定的发送者创建的;
3. 发送者不能否认发送数据。

通过对数据应用签名算法（即对数据进行签名），可以得到一个签名，随后可以对其进行验证。数字签名发生在使用私钥的情况下，验证需要公钥。

为了对数据前面，我们需要以下的东西：

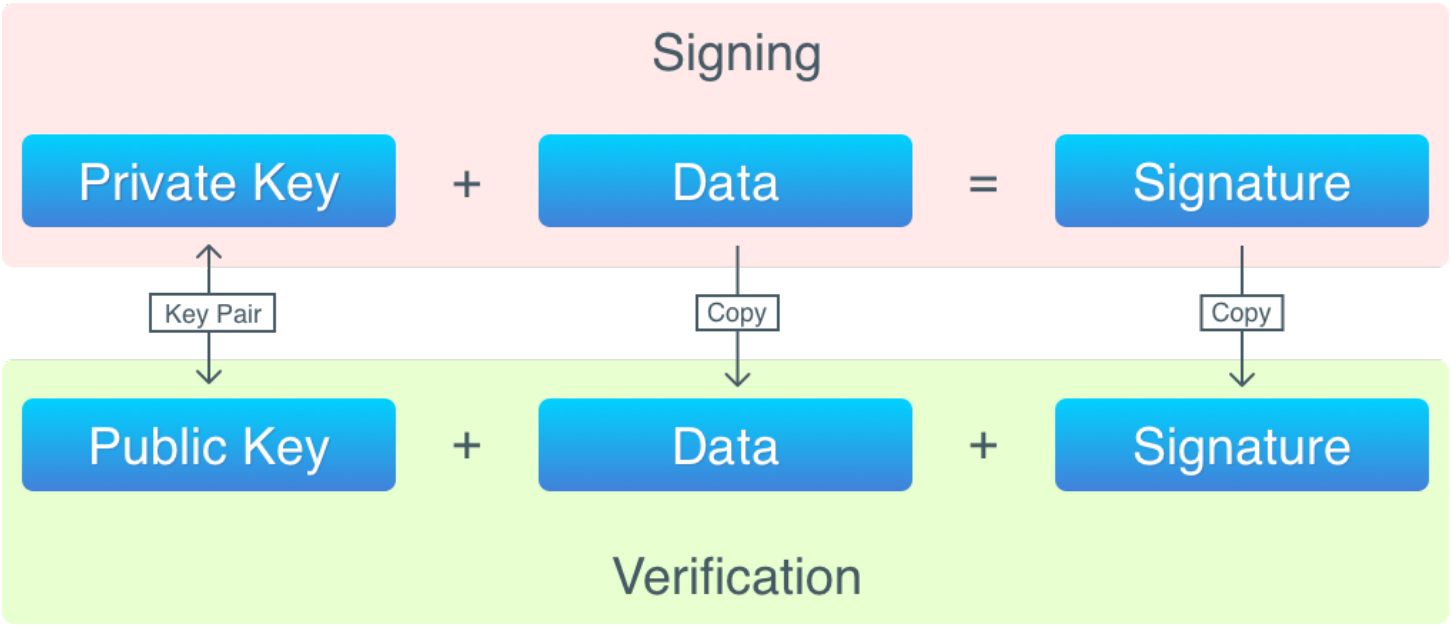
1. 被签名的数据;
2. 签名;
3. 公钥。

简而言之，验证过程可以描述为：检查是否使用用于生成公钥的私钥从此数据中获取此签名。

> 数字签名不是加密，您不能从签名中重建数据。这与哈希相似：通过对数据运行哈希算法并获取数据的唯一表示。签名和哈希之间的区别是密钥对：它们使签名验证成为可能。但密钥对也可用于加密数据：私钥用于加密，公钥用于解密数据。比特币虽然不使用加密算法。

比特币中的每笔交易都由创建交易的人签名。比特币中的每笔交易都必须经过验证，然后才能放入区块中。手段意味着（除其他程序外）：
1. 检查交易输入是否有权使用上笔交易的输出。
2. 确认交易签名是对的。

示意图上，数据签名和验证签名的过程看起来像这样：



现在来回顾一个交易完整的生命周期：

1. 起初，有一个包含coinbase交易的创世区块。在coinbase交易中没有真正的输入，所以签名是没有必要的。coinbase交易的输出包含一个哈希过的公钥（使用`RIPEMD16(SHA256(PubKey))`算法）。
2. 当一个人发送货币时，就会创建一个交易。交易的输入将参考先前交易的输出。每个输入都会存储一个公钥（不是哈希）和整个交易的签名。
3. 接收交易的比特币网络中的其他节点将对其进行验证。除此之外，他们将检查：输入中公钥的哈希与引用输出的哈希匹配（这可确保发送者仅花费属于它们的货币）；签名是正确的（这确保交易是由货币的真正所有者创建的）。
4. 当一个矿工节点准备挖一个新的区块时，它将把这个交易放在一个区块中并开始挖掘它。
5. 当新区块被挖到了，网络中的每一个其他节点都会收到一个消息说区块已经被挖到了，并把这个区块添加到区块链中。
6. 将区块添加到区块链后，交易完成，其输出可在新交易中引用。

5、椭圆曲线加密

如上所述，公钥和私钥是随机字节的序列。由于它是用来识别货币所有者的私钥，因此有一个必要条件：随机性算法必须产生真正的随机字节。我们不希望意外生成其他人拥有的私钥。

比特币使用椭圆曲线来生成私钥。椭圆曲线是一个复杂的数学概念，我们不会在这里详细解释（如果您有兴趣，请查看椭圆曲线的简单介绍 警告：数学公式！）。我们需要知道的是，这些曲线可以用来生成真正大而随机的数字。比特币使用的曲线可以随机选取一个介于0和 2^{256} 之间的数字（当可见宇宙中的原子数介于10和10之间时，大约为10^{77}）。如此巨大的上限意味着几乎不可能两次生成相同的私钥。

此外，比特币使用（我们将）ECDSA（椭圆曲线数字签名算法）算法来对交易进行签名。

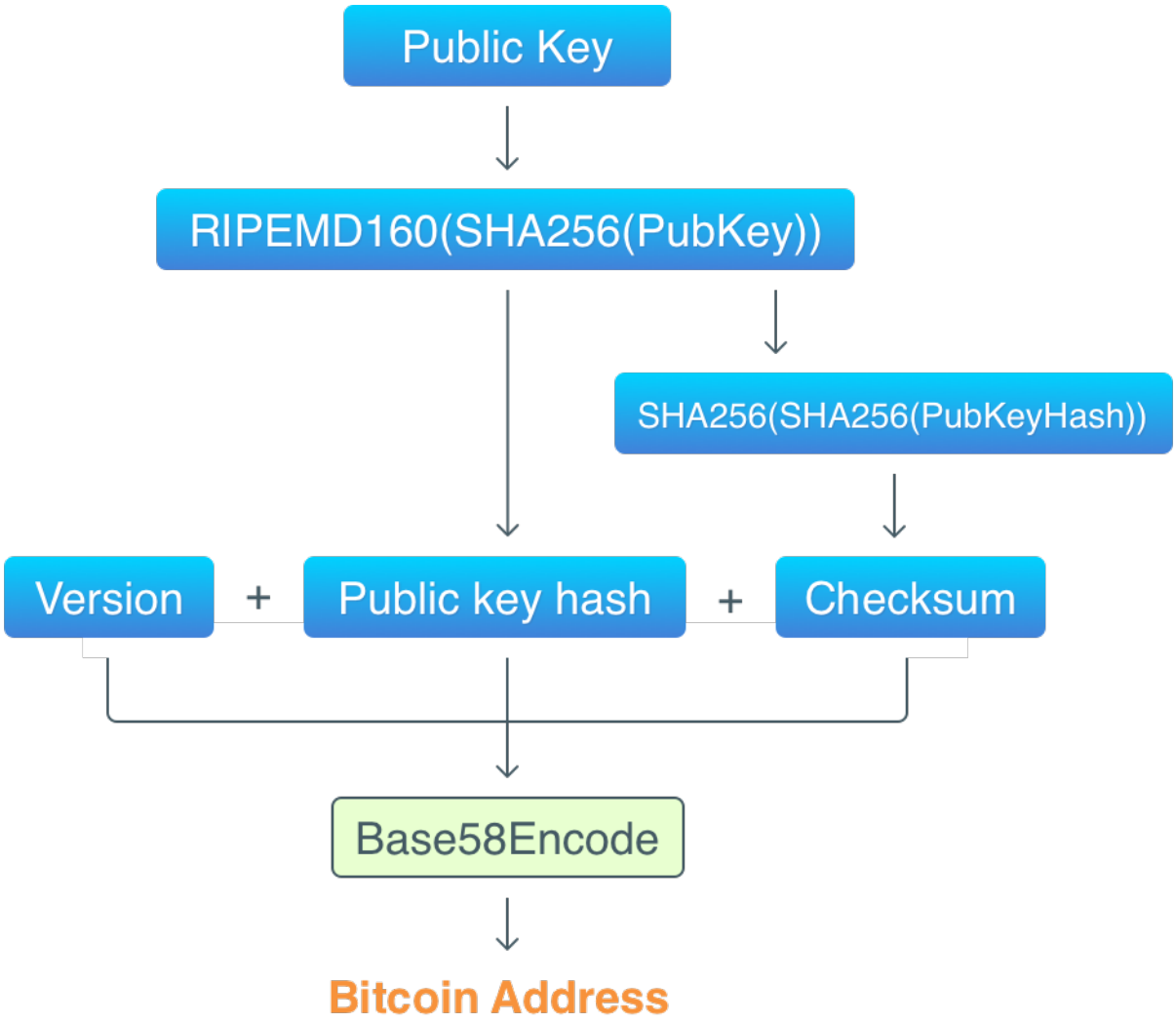
6、Base58

现在让我们回到上面提到的比特币地址：1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa。现在我们知道这是一个公开密钥的人类可读表示。如果我们对它进行解码，那么公钥看起来像什么（作为用十六进制写的字节序列）：

```
0062E907B15CBF27D5425399EBF6F0FB50EBB88F18C29B7D93
```

比特币使用Base58算法将公钥转换为可读格式。该算法与著名的Base64非常相似，但它使用较短的字母表：从字母表中删除一些字母以避免使用字母相似性的一些攻击。因此，没有这些符号：0（零），O（大写o），I（大写i），l（小写L），因为它们看起来相似。此外，没有+和/符号。

让我们以示意图形式显示从公钥获取地址的过程：



因此，上述解码的公钥由三部分组成：

```
Version  Public key hash                             Checksum
00       62E907B15CBF27D5425399EBF6F0FB50EBB88F18   C29B7D93
```

由于哈希函数是一种方式（即它们不能被反向解码），因此无法从哈希中提取公钥。不过通过运行哈希函数并与之进行哈希比较，我们可以检查一个公钥是否被用于哈希的生成。

好的，现在我们有了所有的东西，让我们来编写一些代码。用代码编写时，一些概念应该更清晰。

7、实现地址

我们将从 `Wallet` 结构开始：

```
type Wallet struct {
    PrivateKey ecdsa.PrivateKey
    PublicKey  []byte
}

type Wallets struct {
    Wallets map[string]*Wallet
}

func NewWallet() *Wallet {
    private, public := newKeyPair()
    wallet := Wallet{private, public}

    return &wallet
}

func newKeyPair() (ecdsa.PrivateKey, []byte) {
    curve := elliptic.P256()
    private, err := ecdsa.GenerateKey(curve, rand.Reader)
    pubKey := append(private.PublicKey.X.Bytes(), private.PublicKey.Y.Bytes(
)...)

    return *private, pubKey
}
```

钱包不过是一对钥匙。我们还需要这种 `Wallets` 类型来保存一些钱包，将它们保存到一个文件中，然后从中加载它们。在 `Wallet` 的构造函数会生成一个新的密钥对。该 `newKeyPair` 函数很简单：ECDSA基于椭圆曲线，所以我们需要一个。接下来，使用曲线生成私钥，并从私钥生成公钥。有一点需要注意：在基于椭圆曲线的算法中，公钥是曲线上的点。因此，公钥是X，Y坐标的组合。在比特币中，这些坐标被连接起来形成一个公钥。

现在，来生成一个地址：

```
func (w Wallet) GetAddress() []byte {
```

```
    pubKeyHash := HashPubKey(w.PublicKey)

    versionedPayload := append([]byte{version}, pubKeyHash...)
    checksum := checksum(versionedPayload)

    fullPayload := append(versionedPayload, checksum...)
    address := Base58Encode(fullPayload)

    return address
}

func HashPubKey(pubKey []byte) []byte {
    publicSHA256 := sha256.Sum256(pubKey)

    RIPEMD160Hasher := ripemd160.New()
    _, err := RIPEMD160Hasher.Write(publicSHA256[:])
    publicRIPEMD160 := RIPEMD160Hasher.Sum(nil)

    return publicRIPEMD160
}

func checksum(payload []byte) []byte {
    firstSHA := sha256.Sum256(payload)
    secondSHA := sha256.Sum256(firstSHA[:])

    return secondSHA[:addressChecksumLen]
}
```

以下是将公钥转换为Base58地址的步骤：

1. 取公钥并用 `RIPEMD160(SHA256(PubKey))` 哈希算法对它进行两次哈希。
2. 将地址生成算法的版本添加到哈希。
3. 用步骤2的结果散列来计算校验和 `SHA256(SHA256(payload))` 。校验和是结果散列的前四个字节。
4. 将校验和附加到 `version+PubKeyHash` 组合。
5. `version+PubKeyHash+checksum` 使用Base58 编码组合。

结果，你会得到一个 `真正的比特币地址` ，你甚至可以在blockchain.info上查看它的余额。但是我可以向你保证，无论你多少次生成一个新地址并检查其余额，余额都是0。这就是为什么选择合适的公钥密码算法如此重要：考虑到私钥是随机数字，产生相同数字的机会必须尽可能低。理想情况下，它必须低至"永不重复"。

另外，请注意，您无需连接到比特币节点即可获取地址。地址生成算法利用在许多编程语言和库中实现的开放算法的组合。

现在我们需要修改它们的输入和输出以使用地址：

```
type TXInput struct {
    Txid      []byte
    Vout      int
    Signature []byte
    PubKey    []byte
}

func (in *TXInput) UsesKey(pubKeyHash []byte) bool {
    lockingHash := HashPubKey(in.PubKey)

    return bytes.Compare(lockingHash, pubKeyHash) == 0
}

type TXOutput struct {
    Value      int
    PubKeyHash []byte
}

func (out *TXOutput) Lock(address []byte) {
    pubKeyHash := Base58Decode(address)
    pubKeyHash = pubKeyHash[1 : len(pubKeyHash)-4]
    out.PubKeyHash = pubKeyHash
}

func (out *TXOutput) IsLockedWithKey(pubKeyHash []byte) bool {
    return bytes.Compare(out.PubKeyHash, pubKeyHash) == 0
}
```

请注意，我们不再使用 `ScriptPubKey` 和 `ScriptSig` 字段，因为我们不打算实现一个脚本语言。相反， `ScriptSig` 被分成 `Signature` 和 `PubKey` 字段，并被 `ScriptPubKey` 重命名为 `PubKeyHash` 。我们将像比特币一样实现相同的输出锁定/解锁和输入签名逻辑，但是我们会在方法中执行此操作。

该 `UsesKey` 方法检查输入是否使用特定的键来解锁输出。请注意，输入存储原始公钥（即，未哈希），但需要一个哈希函数。 `IsLockedWithKey` 检查提供的公钥哈希是否用于锁定输出。这是一个补充功能 `UsesKey` ，并且它们都用于在 `FindUnspentTransactions` 交易之间建立连接。

`Lock` 只需锁定输出。当我们向别人发送货币时，我们只知道他们的地址，因此函数将地址作为唯一的参数。然后解码该地址，并从中提取公钥哈希并保存在该 `PubKeyHash` 字段中。

现在，让我们检查一切正常：

```
$ blockchain_go createwallet
Your new address: 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt

$ blockchain_go createwallet
Your new address: 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h

$ blockchain_go createwallet
Your new address: 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy

$ blockchain_go createblockchain -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
0000005420fbfdafa00c093f56e033903ba43599fa7cd9df40458e373eee724d

Done!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 10

$ blockchain_go send -from 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h -to 13Uu7B1vDP
4ViXqHFsWtbraM3EfQ3UkWXt -amount 5
2017/09/12 13:08:56 ERROR: Not enough funds

$ blockchain_go send -from 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt -to 15pUhCbtrG
h3JUx5iHnXjfpyHyTgawvG5h -amount 6
00000019afa909094193f64ca06e9039849709f5948fbac56cae7b1b8f0ff162

Success!

$ blockchain_go getbalance -address 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 4

$ blockchain_go getbalance -address 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h
Balance of '15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h': 6

$ blockchain_go getbalance -address 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy
Balance of '1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy': 0
```

太好了！现在我们来实现交易签名。

8、实现签名

交易必须被签名，因为这是比特币唯一能够保证不能花钱购买属于他人的货币的方法。如果一个签名是无效的，那么这笔交易就会被认为是无效的，因此，这笔交易也就无法被加到区块链中。

我们已经完成了所有的交易签名，除了一件事：数据的签名。交易的哪些部分要被签名？或者一项交易是整体签名的？选择要签名的数据非常重要。问题是要签名的数据必须包含以独特方式标识数据的信息。例如，仅对输出值进行签名是没有意义的，因为此签名不会考虑到发送者和接收

者。

考虑到事务解锁先前的输出，重新分配它们的值并锁定新的输出，必须对以下数据进行签名：

1. 公钥哈希存储在解锁输出中。这标识了交易的"发送者"。
2. 公钥哈希存储在新的锁定输出中。这标识了交易的"接收者"。
3. 新输出的值。

> 在比特币，锁定/解锁逻辑被存储在脚本，分别被存储输出和输入的 `ScriptSig` 和 `ScriptPubKey` 字段中。由于比特币允许不同类型的脚本，所以要对 `ScriptPubKey` 的整个内容签名。

正如你所看到的，我们不需要签名存储在输入中的公钥。正因为如此，在比特币中，这不是一个已签名的交易，而是一个去除部分内容的输入副本，输入里面存储了被引用输出的 `ScriptPubKey` 。

> 这里描述获取修剪后的交易副本的详细过程。它很可能已经过时，但我没有设法找到更可靠的信息来源。

好吧，它看起来很复杂，所以让我们开始编码吧。我们将从 `Sign` 方法开始：

```
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs map[string]Tra
nsaction) {
    if tx.IsCoinbase() {
        return
    }

    txCopy := tx.TrimmedCopy()

    for inID, vin := range txCopy.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil

        r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
        signature := append(r.Bytes(), s.Bytes()...)

        tx.Vin[inID].Signature = signature
    }
}
```

这个方法获取了一个私钥，还有上个交易信息的 map。如上所述，为了签名交易，，我们需要去

访问被入账交易引用的输出，所以我们需要交易保存着这些输出。

让我们一步一步回顾这个方法：

```
if tx.IsCoinbase() {
    return
}
```

Coinbase交易没有签名，因为它们没有真正的输入。

```
txCopy := tx.TrimmedCopy()
```

剪裁的交易副本将被签名，而不是完整的交易：

```
func (tx *Transaction) TrimmedCopy() Transaction {
    var inputs []TXInput
    var outputs []TXOutput

    for _, vin := range tx.Vin {
        inputs = append(inputs, TXInput{vin.Txid, vin.Vout, nil, nil})
    }

    for _, vout := range tx.Vout {
        outputs = append(outputs, TXOutput{vout.Value, vout.PubKeyHash})
    }

    txCopy := Transaction{tx.ID, inputs, outputs}

    return txCopy
}
```

该副本将包括所有的输入和输出，但 `TXInput.Signature` 和 `TXInput.PubKey` 被设置为 `nil` 。

接下来，我们遍历副本中的每个输入：

```
for inID, vin := range txCopy.Vin {
    prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
    txCopy.Vin[inID].Signature = nil
    txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
```

在每个输出中， `Signature` 被设为 `nil` (只是是一个双重检验)， `PubKey` 被设置为所引用输出

的 `PubKeyHash` 。此时，除了现有交易外，其他所有交易都是"空"的。它们的 `Signature` 和 `PubKey` 字段都被设成了 nil。因此， `输入是分开签署的` ，尽管对于我们的应用来说是没必要的，但比特币允许交易包含不同地址的入账。

```
    txCopy.ID = txCopy.Hash()
    txCopy.Vin[inID].PubKey = nil
```

`Hash` 方法会序列化交易，并用 `SHA-256` 算法其进行哈希处理。这个结果哈希就是我们要去签名的数据。得到散列后，我们应该重置该 `PubKey` 字段，所以它不会影响进一步的迭代。

现在，中心部分：

```
    r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
    signature := append(r.Bytes(), s.Bytes()...)

    tx.Vin[inID].Signature = signature
```

我们通过 `privKey` 对 `txCopy.ID` 进行签名。ECDSA签名是一对数字，我们连接并存储在输入 `Signature` 字段中。

现在，验证函数：

```
func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {
    txCopy := tx.TrimmedCopy()
    curve := elliptic.P256()

    for inID, vin := range tx.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil

        r := big.Int{}
        s := big.Int{}
        sigLen := len(vin.Signature)
        r.SetBytes(vin.Signature[:(sigLen / 2)])
        s.SetBytes(vin.Signature[(sigLen / 2):])

        x := big.Int{}
        y := big.Int{}
        keyLen := len(vin.PubKey)
        x.SetBytes(vin.PubKey[:(keyLen / 2)])
```

```
        y.SetBytes(vin.PubKey[(keyLen / 2):])

        rawPubKey := ecdsa.PublicKey{curve, &x, &y}
        if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
            return false
        }
    }

    return true
}
```

该方法非常简单。首先，我们需要相同的交易副本：

```
txCopy := tx.TrimmedCopy()
```

接下来，我们将需要用于生成密钥对的相同曲线：

```
curve := elliptic.P256()
```

接下来，我们检查每个输入中的签名：

```
for inID, vin := range tx.Vin {
    prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
    txCopy.Vin[inID].Signature = nil
    txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
    txCopy.ID = txCopy.Hash()
    txCopy.Vin[inID].PubKey = nil
```

这部分与该 `Sign` 方法中的相同，因为在验证过程中我们需要签名相同的数据。

```
    r := big.Int{}
    s := big.Int{}
    sigLen := len(vin.Signature)
    r.SetBytes(vin.Signature[:(sigLen / 2)])
    s.SetBytes(vin.Signature[(sigLen / 2):])

    x := big.Int{}
    y := big.Int{}
    keyLen := len(vin.PubKey)
    x.SetBytes(vin.PubKey[:(keyLen / 2)])
    y.SetBytes(vin.PubKey[(keyLen / 2):])
```

这里，我们把存在 `TXInput.Signature` 和 `TXInput.PubKey` 里的值拿出来，由于签名是一对数字，而公钥是一对坐标。我们之前为了存储就把它们连在了一块，现在要用 crypto/ecdsa 方法去取出来。

```
    rawPubKey := ecdsa.PublicKey{curve, &x, &y}
    if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
        return false
    }
}

return true
```

这里：我们创建一个 `ecdsa.PublicKey` 使用从输入中提取的公钥并执行 `ecdsa.Verify` 传递从输入中提取的签名。如果所有输入都已验证，则返回true; 如果至少有一个输入未通过验证，则返回false。

现在，我们需要一个函数来获取以前的交易。由于这需要与区块链互动，我们将使其成为一种方法 `Blockchain` ：

```
func (bc *Blockchain) FindTransaction(ID []byte) (Transaction, error) {
    bci := bc.Iterator()

    for {
        block := bci.Next()

        for _, tx := range block.Transactions {
            if bytes.Compare(tx.ID, ID) == 0 {
                return *tx, nil
            }
        }

        if len(block.PrevBlockHash) == 0 {
            break
        }
    }

    return Transaction{}, errors.New("Transaction is not found")
}

func (bc *Blockchain) SignTransaction(tx *Transaction, privKey ecdsa.Private
Key) {
    prevTXs := make(map[string]Transaction)

    for _, vin := range tx.Vin {
        prevTX, err := bc.FindTransaction(vin.Txid)
```

```
        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    tx.Sign(privKey, prevTXs)
}

func (bc *Blockchain) VerifyTransaction(tx *Transaction) bool {
    prevTXs := make(map[string]Transaction)

    for _, vin := range tx.Vin {
        prevTX, err := bc.FindTransaction(vin.Txid)
        prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
    }

    return tx.Verify(prevTXs)
}
```

这些功能很简单：`FindTransaction` 通过ID查找交易（这需要迭代区块链中的所有区块）；`SignTransaction` 拿到交易，找到它引用的交易并签名；`VerifyTransaction` 做同样的事情，而是验证交易。

现在，我们需要实际签名和验证交易。签名发生在 `NewUTXOTransaction` 中：

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transa
ction {
    ...

    tx := Transaction{nil, inputs, outputs}
    tx.ID = tx.Hash()
    bc.SignTransaction(&tx, wallet.PrivateKey)

    return &tx
}
```

验证发生在交易被放入区块之前：

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
    var lastHash []byte

    for _, tx := range transactions {
        if bc.VerifyTransaction(tx) != true {
            log.Panic("ERROR: Invalid transaction")
        }
    }
    ...
```

```
}
```

就是这样！让我们再检查一次：

```
$ blockchain_go createwallet
Your new address: 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR

$ blockchain_go createwallet
Your new address: 1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab

$ blockchain_go createblockchain -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
000000122348da06c19e5c513710340f4c307d884385da948a205655c6a9d008

Done!

$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to 1NE86r4Esj
f53EL7fR86CsfTZpNN42Sfab -amount 6
0000000f3dbb0ab6d56c4e4b9f7479afe8d5a5dad4d2a8823345a1a16cf3347b

Success!

$ blockchain_go getbalance -address 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
Balance of '1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR': 4

$ blockchain_go getbalance -address 1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab
Balance of '1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab': 6
```

没有错误。真棒！

让我们来注释掉 `NewUTXOTransaction` 中的 `bc.SignTransaction(&tx, wallet.PrivateKey)` 的调用，以此来保证未被签名的交易无法被挖矿：

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain) *Transa
ction {
   ...
    tx := Transaction{nil, inputs, outputs}
    tx.ID = tx.Hash()
    // bc.SignTransaction(&tx, wallet.PrivateKey)

    return &tx
}
```

```
$ go install
$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to 1NE86r4Esj
```

```
f53EL7fR86CsfTZpNN42Sfab -amount 1
2017/09/12 16:28:15 ERROR: Invalid transaction
```

9、总结

这真的很棒，我们已经得到了这么多，并且实现了比特币的许多关键特性！我们已经实现了网络外的几乎所有内容，并且在下一部分中，我们将完成交易。

链接：

1. 完整的源代码
2. 公钥加密
3. 数字签名
4. 椭圆曲线
5. 椭圆曲线加密
6. ECDSA
7. 比特币地址的技术背景
8. 地址
9. Base58
10. 椭圆曲线密码学的简单介绍