第3章

以 太 坊

以太坊是目前市场份额仅次于比特币的第二大区块链系统，它在比特币原有的性能和应用场景基础上进行了拓展，是第一个支持智能合约的区块链系统。区块链的应用场景，因以太坊的诞生，从原本单一的加密数字货币交易，延伸到灵活多样的自定义应用设计。本章将介绍以太坊的设计原理，主要包括以太坊的简介和架构、账户及交易、数据结构与存储、共识机制等部分。

3.1 以太坊简介

3.1.1 以太坊的诞生

比特币的诞生开创了区块链技术的先河，并受到了很多关注。在比特币诞生后的几年里，一些性能提升或带有新功能的区块链系统也开始出现，如具有更快交易确认速度的莱特币(Litecoin)、引入权益证明(Proof of Stake，PoS)的点点币(Peercoin)、更注重安全和隐私的门罗币(Monero)等。但是，这些项目的可扩展性有限，不是能面向多种使用场景自定义数字资产(如自定义债券、凭证等需要流通的资产)的通用系统。同时，虽然原始的比特币系统可以运行简单的脚本语言，满足一定的灵活性需求，但是这种脚本语言是图灵不完备的，不足以构建更高级的去中心化应用。

2013 年，年仅 19 岁的 Vitalik Buterin 初步提出了以太坊的设计思想。这种新的区块链系统内置了成熟的图灵完备的编程语言，允许将区块链技术拓展到更多可能的应用场景。2014 年，许多开发者加入以太坊项目，成立以太坊基金会并开始以太坊项目的研发。2015 年 7 月，以太坊主网上线，正式开启以智能合约为代表的区块链 2.0 时代。利用智能合约，用户可以基于公共的区块链平台开发属于自己的分布式应用(Decentralized Application，DApp)以及发布代币，使区块链商业应用成为可能。在之后的几年里，以太坊的技术得到了认可和发展，目前已有数千个基于以太坊的 DApp，其主要的加密数字货币以太币(Ether)，也发展成为继比特币之后的第二大加密数字货币。

从以太坊白皮书发表到现在，以太坊经历了多次硬分叉和版本升级。由于其版本和路线图变更频繁，请读者自行参阅社区中的介绍文档。

在未来的阶段，以太坊计划增加更多先进技术（如分片技术、新一代虚拟机 Ewasm），朝着更安全和更持久的去中心化区块链平台发展。

3.1.2　以太坊与比特币对比

同样作为公有链平台，以太坊在比特币的思想上进行了拓展，与比特币之间有着相似的地方，同时也有着与比特币迥异的特性。比特币和以太坊都是基于区块链技术的平台，在它们各自的区块链网络中，节点之间彼此对等而无特权，每一个节点既可以发起交易，又可以参与交易的验证。这些交易会由矿工进行验证并打包成区块，再通过全网广播和共识机制达成分布式账本的一致性。交易数据公开透明、不可篡改，记录在由许多区块有序连接而来的区块链中。

与比特币相比，以太坊初期主要在技术特点、共识机制和社区 3 方面有着不同之处。

（1）在技术方面，以太坊最大的创新是提供了对智能合约的支持。不同于比特币系统仅对比特币流通的支持，以太坊上的用户可以通过智能合约自定义数字资产和流通规则，而以太坊则作为一个更通用的底层区块链平台为各类 DApp 提供运行环境支撑。另外，不同于比特币的 UTXO 模型，以太坊使用了账户模型，使得账户的状态可以实时保存到账户里，不需要回溯交易历史。通过燃料费（Gas）的设置，以太坊对合约指令执行进行限制，降低被攻击的风险。

（2）在共识机制方面，以太坊采用了基于 PoW 的 Ethash 变种算法作为共识机制。首先，以太坊增加了叔块奖励，使得未被纳入主链而挖矿成功的区块也能有挖矿奖励，对矿工更加友好和公平。由于比特币价格的上涨，导致出现了强算力矿机和大型矿池的出现，这些都违背了比特币原本去中心化的理念。Ethash 在执行过程中需要消耗大量的内存，从而降低了强算力矿机在采矿中的优势，并减少受矿池攻击的风险。近年来，以太坊通过硬分叉升级将共识机制逐步变更为 PoS 机制，采用更高效的 Casper 共识算法，并计划通过 Danksharding 分片技术等技术提高网络的可扩展性。

（3）在社区方面，以太坊社区相对于比特币社区更为活跃。以太坊社区成员积极探索新技术，多次召开讨论会议，对以太坊未来的发展路线进行规划，并按照既定路线对系统进行版本更新。到本书写作完为止，以太坊官方已有 227 个 GitHub 开源项目以及各种技术文档。活跃的社区生态促使以太坊在技术发展的过程中，以及可扩展性、安全性等需求增加时，更有迎接变化、拥抱未来的能力。

3.1.3　以太坊的特色与应用

以太坊的出现开启了区块链 2.0 时代——以智能合约为核心的可编程金融时代，所有的金融交易都可以在建立于区块链上的分布式应用中进行。在以太坊上，交易不只是加密数字货币的转账，还可以是智能合约的创建和调用。支持用户自定义和调用一些复

杂的逻辑,面对各种不同的应用场景创建分布式应用,是以太坊区别于以往区块链平台的最大的特色。

作为以太坊的重要组成部分,智能合约本质上是一段可以根据预先指定的条件被触发执行的代码。其概念早在1995年被提出,但由于缺乏可靠执行环境而没有被推广使用。以太坊提供以太坊虚拟机(Ethereum Virtual Machine,EVM)作为智能合约的执行环境,并支持图灵完备的高级语言用于智能合约的开发,其中Solidity应用最为广泛。智能合约的编写可以定义各种变量和函数。在合约部署之后,系统会为部署的合约对象生成一个合约账户。用户通过向合约账户发送交易并指定调用的函数和参数,进行智能合约的调用。智能合约接收到交易请求,触发执行指定的代码逻辑,进一步产生新的交易。智能合约的执行过程和执行结果通过各节点达成共识,随同交易和账户状态存储在区块链中,一经确认,无法篡改。

基于智能合约,区块链技术从最初的加密数字货币流通扩展到了新的应用场景,如溯源存证、数字资产发行和流通、数据共享等。

(1) 溯源存证。传统信息的存储通常采用纸质存储和单一数据库存储,信息丢失和数据伪造的风险较高。如纸质发票在流通过程中容易发生丢失的情况,也容易出现重复报销、金额篡改的可能,使报销单位遭受损失,而进行信息的核对和验证需要很大的成本。区块链的交易数据不可篡改、交易数据公开、分布式存储为传统的溯源取证带来了很大的方便。溯源取证数据可以以交易的形式存储在区块链中,用户简单地通过DApp接口对数据进行查询溯源,既方便快捷,又保证了数据的防伪可信。

(2) 数字资产发行和流通。智能合约的出现使得用户可以按照自己的需要定义数字资产,实现任何形式的数字资产(如积分、股权、游戏币等)在区块链用户之间的自由流通。以太坊作为一个通用的区块链平台,为这些数字资产提供了公开透明的记录账本。

(3) 数据共享。区块链数据的公开透明、多方验证、不可伪造的特性为用户提供可靠的信息共享环境,如征信黑名单信息共享、车辆设备之间的路况信息共享、联邦学习场景下的梯度共享等。每个区块链用户都有一份数据的备份,由一个区块链平台进行信息记录,特别是应用在跨组织的场景下,对共享的信息可信任性有所保障。

(4) 目前,以太坊上的DApp多集中于金融交易、游戏等领域。表3.1展示以太坊上较为流行的10个区块链应用。

表3.1 以太坊当前最流行的10个区块链应用

名　称	类别	功能描述
MakerDAO	金融	去中心化借贷应用
Chainlink	安全	去中心化预言机
KyberNetwork	交换	代币交换协议
Status	钱包	DApp浏览器发行的代币

续表

名　　称	类别	功 能 描 述
My Crypto Heroes	游戏	角色扮演游戏
Uniswap	交换	代币交换协议
Axie Infinity	游戏	宠物养成游戏
Synthetix	金融	去中心化合成资产平台
Basic Attention Token	钱包	Brave 浏览器的激励代币
Knight Story	游戏	角色扮演游戏

其中，MakerDAO 是成立于 2014 年的一款代币自动抵押平台，采用了稳定币 Dai 以及权益和管理代币 MKR 双币模式。Dai 通过超额抵押加密数字货币获得价值，而 MKR 在用户赎回抵押的 Ether 时会被销毁，从而用户持有的 MKR 会越来越值钱。通过 MakerDAO，用户可以实现价值存储、杠杆交易等多种功能。目前，MakerDAO 平台的累计交易额已高达 20 亿美元。

另外，一些基于区块链技术的应用也通过区块链技术对传统行业进行革新。比如，Brave 浏览器基于以太坊平台利用 Basic Attention Token 打造了一个去中心化、透明的数字广告平台。传统的网页广告行业由用户、内容商和广告商组成，但是这三者之间充斥着行为追踪和欺诈，即用户的隐私可能得不到保护，内容商可能通过虚假用户刷量来提升广告收入。面对这个问题，Brave 浏览器实现了基于区块链的广告投放，通过加密安全算法保护用户的隐私，让用户有选择地观看广告，并且使用代币奖励浏览广告的用户和优质的内容商。

3.2　以太坊基本架构及原理

一般，所有的账户、余额、智能合约代码、智能合约状态等统称为状态，在区块 N 执行前状态为 S，经过区块 N 的交易进行状态转换后，转换为状态 S′，再经过区块 $N+1$ 的转换后，转换为状态 S″，如图 3.1 所示。采用这一模型后，智能合约即是作用于该状态机转换的代码，在以太坊中，执行状态转换代码的虚拟机是以太坊虚拟机（Ethereum Virtual Machine，EVM）。

区块链上的智能合约需要在多个节点间保持执行过程和结果的一致。相比比特币的 UTXO 模型，状态转换模型虽然使得智能合约的各种变量存储、传参等变得更加灵活，但也带来了多方共识上的困难，如发生分叉时的处理。

在公有链上，由于网络的延迟，区块链的分叉是极为常见的。在比特币系统中，当发生分叉时，需要进行从一个分叉到另一个分叉的验证，重新计算回滚 UTXO 集合即可继续验证新区块。但是，在以太坊的状态转换模型中，如果发生分叉，需要回到分叉前的状态，重新验证另一条分支上的区块。如何快速回滚到分叉前的状态便成了难点。一种可

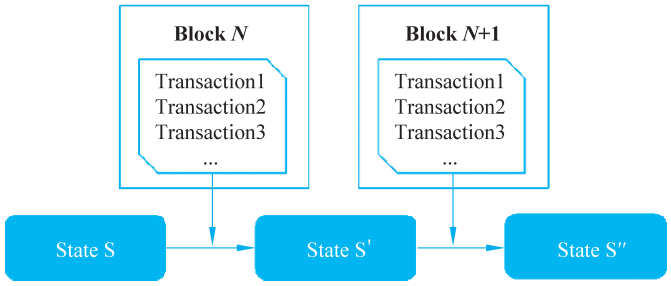

图 3.1 以太坊中的状态转换模型示意图

行的选择是将每个区块对应的状态独立复制存储,但是这将需要极大的存储空间和复制时间开销,是低效的做法。

在以太坊中,状态的存储采用了一种独立于区块链存在的树状结构,该状态树结构的树根登记在区块中,记为 stateRoot,从而使得状态能够在全网得到共识确认,并在分叉时能够快速回滚。

图 3.2 展示了以太坊的简易架构图,该架构包括以下几个基本概念。

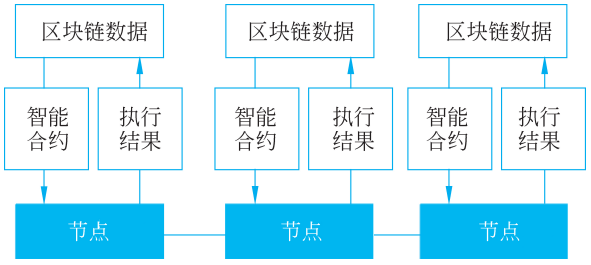

图 3.2 以太坊简易架构示意图

(1) 区块链数据。在本章的余下叙述中,为了表示方便,将以太坊区块链及其状态数据、收据数据(将在下文中介绍)等,统称为区块链数据。该数据同样由前后相连的区块组成,每个区块中包含了区块头和区块体(即交易)。

(2) 智能合约。智能合约是按一定预先设定的逻辑改变以太坊状态的代码,存储在以太坊状态数据中,由区块链交易进行触发,使得以太坊状态发生改变。

(3) 节点。即保存有区块链数据的节点。每个节点独立地维护区块链数据、执行智能合约,并通过 P2P 网络进行通信,采用一定的共识机制(如工作量证明)达成共识,维护全网的状态一致。

需要注意的是,以上是便于读者理解的简易架构图,在具体的工程实现中,以太坊的组成更为复杂,本书将在后续章节中详细介绍。

如图 3.3 所示,下面以“张三买电影票”的例子介绍以太坊执行区块链交易的基本原理。

(1) 网络中所有节点独立地维护区块链数据,这些节点可以是张三、李四、赵四、影院主管等人。

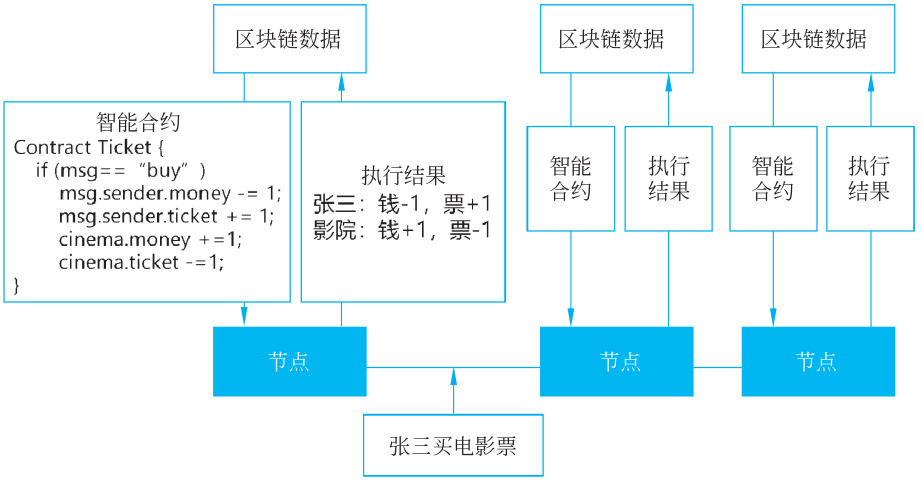

图 3.3　以太坊基本原理及例子

(2) 每当有新的区块链交易(智能合约部署、调用等)产生时,各个节点会从各自冗余的区块链数据中读取智能合约代码、状态等信息,并独立地在以太坊虚拟机中进行执行。在这一例子中,"张三买电影票"的交易将会在所有节点中被验证。节点读出的智能合约代码如图 3.3 中所示,该 Ticket 合约首先将判断这一交易的动作,如果是买票,则把人的钱扣掉,增加一张票,同时把钱转给影院,并减去剩余的票数。再次强调,每个节点都会独立地完成这一校验和计算,也就是说,每个节点都会对张三买电影票这件事进行独立检验。

(3) 以太坊虚拟机的执行结果将以某种方式写回到区块链数据中,以保证各个节点合约执行的强一致性。比如张三的余额、影院的余票等。每个区块中会保留一段摘要,这段摘要为执行完区块中交易后以太坊状态的 stateRoot,任一子状态的不同都将导致 stateRoot 的不同。如张三余额为 100 的 stateRoot 和张三余额为 99 的 stateRoot 是完全不同的,其详细原理将在余下章节中阐述。

(4) 在智能合约执行的过程中,如果节点受到非法攻击或篡改,则执行结果及区块链数据将与网络中其他节点不符,无法参与到网络的下一步共识中。如张三控制的节点被张三篡改,使得他的钱不被扣除,那么他的执行结果输出的区块链数据将会和李四、赵四、影院的节点不符。由区块链的基本特性可知,张三篡改后的区块执行结果将不被承认、无法参与下一步共识。由此,以太坊智能合约可被认为是难以篡改的。

3.3　账户模型与转账

3.3.1　账户模型

比特币系统中采用了计算用户持有的 UTXO 方式来实现记录用户持有的比特币数额,而以太坊则采用了更为简单的账户模型来记录用户持有的以太币的数额。账户模型

其实好比人们日常生活中常见的银行系统。如图3.4所示,一个银行里的每个储户所存的金额,本质上都是存储在银行账目中的一个数字,例如Alice在银行中存款100万元,Bob在银行中存款10元。以太坊的账户模型基本上是一样的设计,只不过在以太坊系统中,对于用户账户的记录不再是由一个中心化的节点,比如银行来提供,而是将这个账户记录保存到所有以太坊的节点中,由整个以太坊的系统来确定记录用户的账户行为,整个过程是去中心化的。

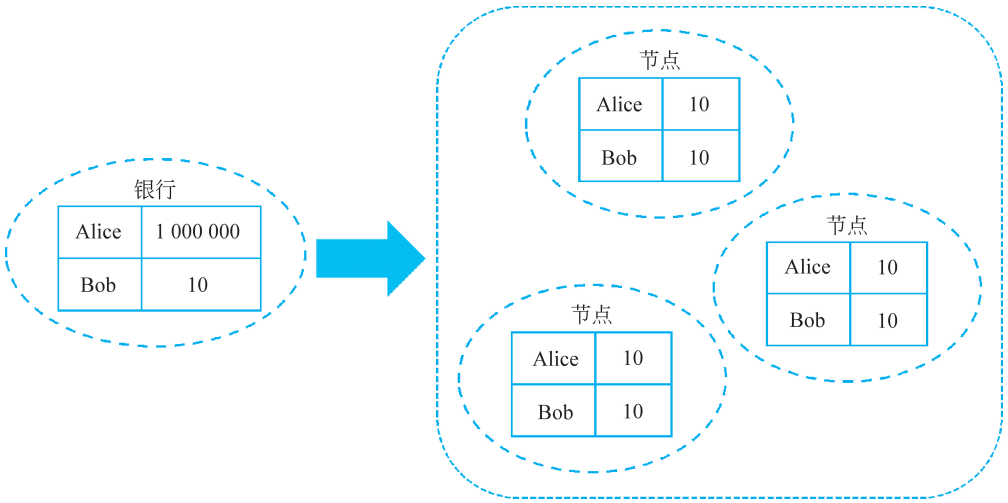

图3.4 以太坊的账户模型

1. 地址与账户

与比特币系统类似的是,以太坊系统作为一个去中心化的系统,没有办法实现用户到密码的映射关系。因此,以太坊的用户同样是通过非对称密码学来进行认证和控制的。对于每一个私钥,可以计算得到这个私钥对应的公钥,进而可以使用公钥来计算得到私钥对应的以太坊地址。在以太坊中,地址是用户的一个身份标识,而账户代表了这个地址对应的信息,比如这个地址持有的具体的以太币数额。

在下文的例子中,有时候会使用Alice、Bob等称呼来指代用户,在实际的以太坊中并不存在这种特定的标识,而是以地址作为用户的标识。这里使用Alice等用户名是为了方便理解,实际中替换为对应的以太坊地址即可。

2. 地址的生成

从以太坊用户的公私钥到用户地址的计算过程相对比较简单,主要是经历了一次哈希算法。具体流程如下。

(1) 计算椭圆曲线下的私钥与公钥。这里采用与比特币章节中一样的私钥与公钥作为例子。

```
7C7FC7CCECC1FEE3B3E4AC63ADD864BD786696C20CA15683A269AA9AD8639443(私钥)
FBD0A98CEF180CDAA3FC2B64A4CA56FBC8E8AF5CC74E820DA34EB9DAB0D46FF2E38587079
29D0A9BEACF5533C22BF02D34E0A6EE4F1A19C1CEAC4FC4251810A6(公钥)
```

(2) 对公钥使用 KECCAK256 哈希算法,计算得到一个 64 位的十六进制哈希值(256 位)。

```
DDC0D707E349E9B726925710238661F085A338F04B0C7C956A796B57018151F0
```

(3) 截取这个哈希值的最后 40 位作为一个以太坊地址。

```
238661F085A338F04B0C7C956A796B57018151F0
```

3. 账户结构

在以太坊中,用户账户结构保存了用户地址对应账户的数据信息,主要有这个地址持有的以太币余额(Balance),以及用于记录这个地址发起交易次数的计数器 Nonce,结构大致如图 3.5 所示。其中,余额记录了当前地址持有的以太币的数额,单位是 Wei,这是以太坊的最小货币单位,1 个以太币等于 10^{18} Wei。而 Nonce 值则是记录了这个地址创建以来累计发起的交易次数,每当从这个地址发起的交易得到确认之后,账户的 Nonce 值便会相应地增加。

图 3.5　账户结构

3.3.2　转账

以太坊的转账同样使用交易来进行,但是比起比特币更类似于日常生活中的转账行为。例如,图 3.6,Alice 向 Bob 转账 1 ETH,那么将会生成一个<From: Alice, To: Bob, value: 1 ETH>的转账记录。当交易得到确认时,以太坊会在自己的存储数据中将 Alice 的账户金额减少 1 ETH,在 Bob 的账户中增加 1 ETH。

在这个过程中,交易<From: Alice, To: Bob, value: 1 ETH>同样通过数字签名的方式来认证。交易的数据通过 Alice 的私钥进行签名之后会得到一个对应的数字签名。通过相应的签名验证流程,以太坊节点可以很容易地通过交易本身和对应的数字签名还原得到 Alice 的公钥值。通过以太坊的地址生成流程,可以通过 Alice 的公钥计算出

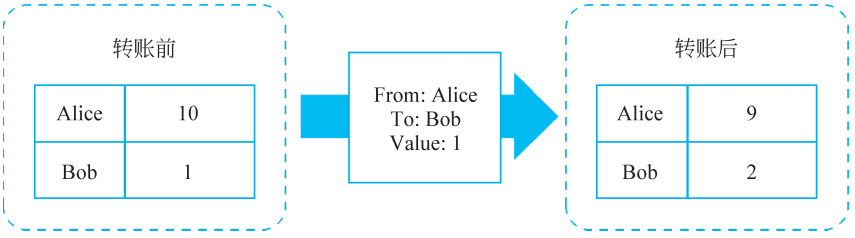

图 3.6 Alice 向 Bob 转账

对应的 Alice 的以太坊地址,如果交易受到篡改,计算通过交易哈希值和签名计算得到的地址便是错误的,也就是说,无法通过篡改 Alice 签名过的交易来攻击 Alice 的账户。

3.3.3 Nonce

在比特币中,交易一旦执行,交易的输入便会被花费而导致无法再次使用,但是在以太坊的模型中,交易的合法性检验在于转账发起者的账户余额,如果没有其他手段来使得发起过的交易失效,那么这个交易将可以被无限次地重新发起而不需要发起者的同意,因为发起者的签名对于交易始终是有效的。

如图 3.7 所示,Alice 发起交易 1 向 Bob 转账 1 ETH 后,这个交易<From: Alice, To: Bob, value: 1 ETH>依旧是有效的,因为 Alice 的余额仍然是足够的。Bob 再次向以太坊系统提交这笔交易,那么 Bob 不需要获得 Alice 的再次签名便可以转账,因为原本的交易已经带有 Alice 的合法签名了。这种情况下,对于验证交易的节点而言,无法判断这笔交易的提交是 Alice 继续向 Bob 进行转账,还是在没有 Alice 同意的情况下进行的一次恶意的攻击行为。

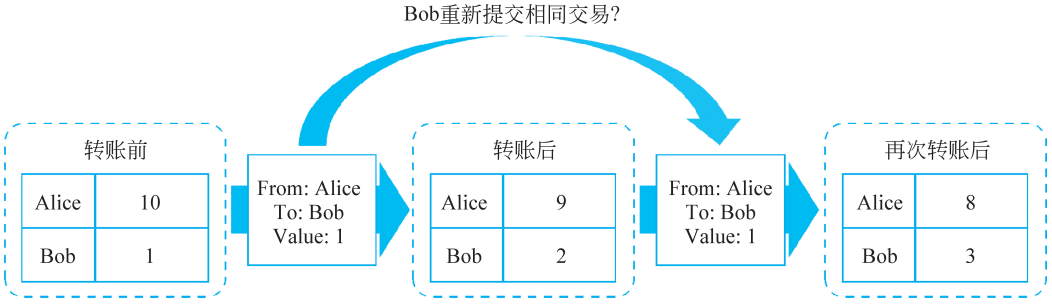

图 3.7 无法识别的交易重放

为此,以太坊系统增加了用于计算交易次数和序列的 Nonce 计数器,只有账户的 Nonce 和交易的 Nonce 能够对应的情况下,交易才是合法的。当一个交易执行完毕之后,账户的 Nonce 值便会增加。这时,原本执行完毕的交易中的 Nonce 值就无法与现在账户的 Nonce 值匹配,执行完毕的交易便失效了。如果修改对应的 Nonce 值,那么便意味着原有交易的签名失效,需要发起者的重新签名,这时候节点只要按照 Nonce 值和签

名值就可以实现对交易的有效验证。

例如图 3.8 中，最初 Alice 向 Bob 转账之前，Alice 的 Nonce 值为 0，当 Alice 向 Bob 转账 1 ETH 后，Alice 的 Nonce 值变成了 1。这时，Nonce 值为 1 并且具有合法签名的交易 2 将会是一个合法的交易。而对于重新提交的交易 1 而言，此时的交易记录的 Nonce 值依旧为 0，即使这个交易具有合法的签名，也不应该认为是一个合法的交易。通过 Nonce 值的设计，可以有效地保证交易最多只能够被执行一次，从而避免了交易重放的攻击。

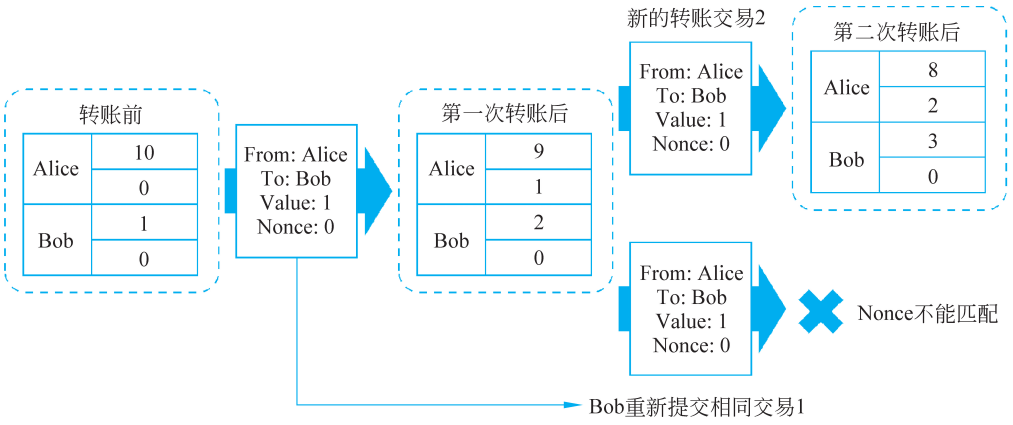

图 3.8　Nonce 值防止重复交易

除此之外，Nonce 值还能够用来控制账户发起交易的顺序，从而实现一些相对复杂的功能。例如，可以通过重复提交一个相同 Nonce 值的交易来使得一个已经提交但是尚未被确认的交易变得不合法，从而实现一定程度的撤销功能。

3.4　智能合约

以太坊的账户模型的最大优点是简单。相比于比特币的 UTXO 模型，账户模型可以不用维护大量的 UTXO 数据，同时还能够很方便地扩展实现智能合约的完整体系，从而实现以太坊“世界计算机”的构想。这个扩展便是状态模型。

3.4.1　状态模型

在账户模型中，用户的余额通过地址上的账户数据来表示，具体为账户数据结构中的一个余额的数值。在转账交易的过程中，通过转账预先定义好的语义，以太坊会在发起者的账户中减去交易中定义好的转账金额，然后在接收者的账户中增加相应的金额。在这个过程中，涉及了发起者与接收者两个账户的余额变化。

如果把账户的余额和转账交易进一步抽象，那么可以把账户的余额泛化成一种账户的状态，而把转账交易当作是改变这种账户状态的一种方式。例如图 3.9 中，对于一个售

票系统而言,当前的余票可以是一种状态,而售出可以是一种状态的变换规则;而对于一个签到系统而言,当前的已签到的集合可以是一种状态,而未签到者的签到可以是一种变换规则。

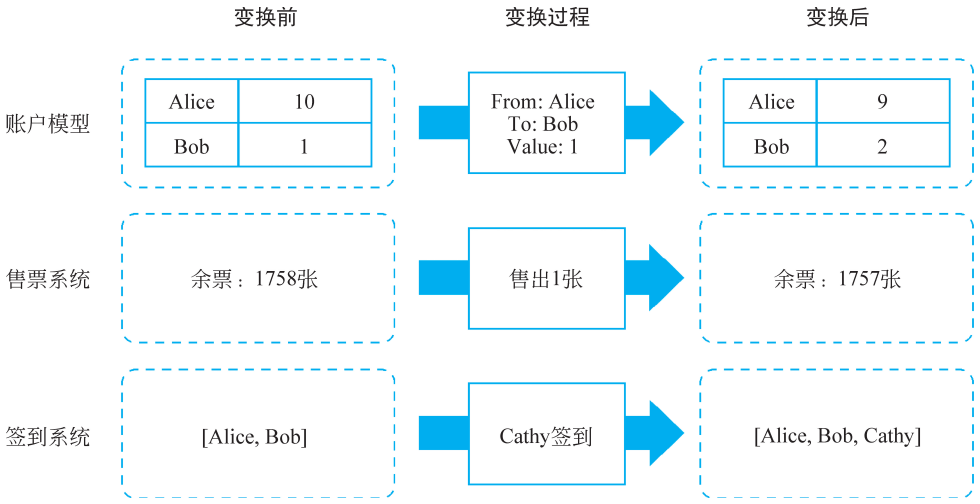

图 3.9 三种系统下的状态变换例子

可以看到,只要约定好的变换规则是固定的,不会产生二义性的结果,那么就可以不断地通过变换规则不停地作用到原有状态上产生新的状态。对于有着相同的初始状态的参与者来说,经过完全相同的变换过程,最后必然得到完全相同的当前状态。对于状态0按顺序执行交易1和交易2,必然得到状态2。这样一来,只要在所有参与者之间对变换过程,也就是执行的交易达成统一,那么对于当前的状态便可以达成一致的认识。

对于状态模型而言,原有的账户模型系统只是一个有限的子集,其中的状态和变换原语都是以太坊预先约定好的,也就是存储账户余额和实现账户转账。扩充完毕的状态模型不仅能够实现原有的账户和转账功能,还能够实现以太坊对于智能合约的支持。

3.4.2 智能合约简介

以太坊对于区块链技术体系的贡献有很多,但是最突出的贡献莫过于将智能合约引入区块链中,极大地丰富了整个区块链系统。简单地说,智能合约是一段在区块链上执行的代码,它依托于区块链系统在参与者之间实现对执行的一致认可。这里将会从原理与实现上简单介绍如何在以太坊的架构上运行智能合约的代码,具体如何编写、部署并运行一个可用的智能合约将在第6章里介绍。

1. 状态模型上的代码执行

可以把计算机程序执行过程中的变量等数据一同存储到区块链上,而交易的过程则是执行一段大家约定好的计算机程序。通过一次交易之后,这些存储的数据经过了这个

交易规定好的代码的执行，发生了一些变化。那么，在整个流程中，这些存储的数据便是一个状态，而执行的代码便是一种特定的变换——这同样符合状态模型的语义。唯一需要规定的是这个执行的过程只能是一个单射，也就是对于当前的状态，不能够得到多种结果。更通俗地讲便是这段程序代码里面不能够出现类似于随机数的实现，因为随机数对于同一个输入得到的是不同的输出。

智能合约便是通过这种方式，利用提前约定好的代码来管理和变化存储在以太坊上的状态变量，利用智能合约的代码来自定义交易过程中的状态变换过程，从而在可以受到以太坊系统的参与者一致认可的条件下不断地执行和变化，实现所谓的“世界状态机”。

如图 3.10 所示，使用了状态模型来保存用户的个人信息，并且制定了 Grow 和 Rename 两种计算机的代码来进行状态修改。当第一次状态变换时，执行的是 Grow(1,1)这一个状态转移的过程，那么个人信息中的年龄和身高就会根据 Grow 的代码分别从原来的 20 和 178 变为 21 和 179，这便完成了一个自定义的状态和状态转换。同样，经过 Rename('Cathy')的调用之后，整个状态中的名称便由原本的 Alice 变成了 Cathy。最后，得到了一个＜name：Cathy，age：21，height：179＞的状态，或者说这就是智能合约这段代码对应的存储数据。

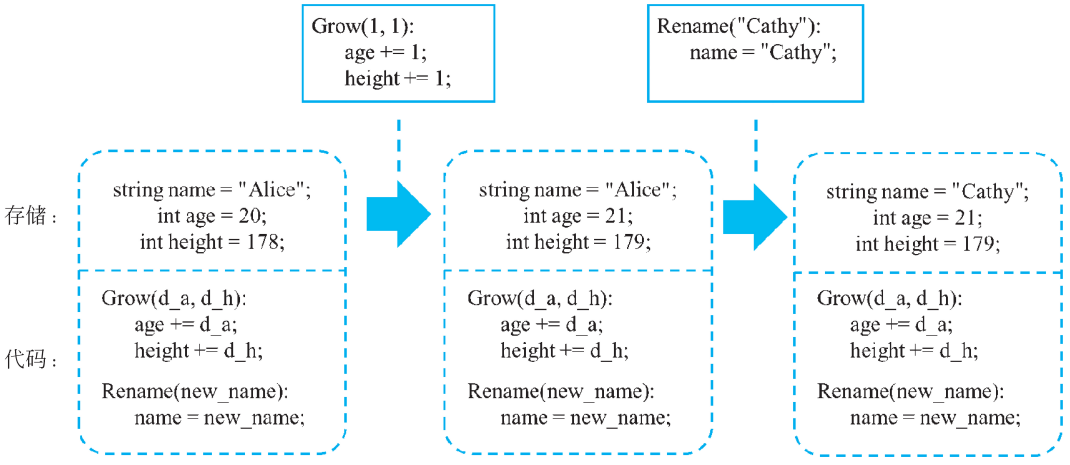

图 3.10　记录 Alice 个人信息的合约

2. 合约账户与数据存储

对于这样一个智能合约与状态的实体，以太坊同样使用了账户作为一个合约实例对象的抽象，它基本上是普通账户的扩展。这种表示合约的账户称为合约账户，它的地址并不是由公钥产生，而是通过特定的算法在创建合约时生成的。原本的表示用户余额的账户则被称为外部账户(Externally Owned Accounts)，因为这些账户是受外部私钥控制的，由外部的用户操控。这里的外部是相对于以太坊运行机制而言的。

在以太坊中,合约账户的数据结构在外部账户的基础上,主要扩展了存储合约代码和存储合约状态的字段。合约的计算机代码通过机器码的形式保存在合约机器码的字段中,而合约的状态存储则是另外保存在一个存储的映射表中,账户的内部只保留了整个存储表的哈希值。如果存储表中的变量的数值发生了变化,那么对应整个存储表的哈希值也会发生变化,账户数据对应的合约存储字段也会发生变化。也就是说,通过数据表的哈希值可以记录账户状态的变化。

注意:由于合约账户并不由具体公钥和私钥进行控制,也就是说并不能从合约地址发起交易。这里的交易指的是记录到区块中的交易。在以太坊的设计中,合约账户可以在受到外部账户的触发后产生新的内部调用,一般被称为内部交易,即不体现在区块中,而是执行过程内部产生的交易。另外,随着以太坊账户抽象化的发展,这一特性可能会发生改变。

如图3.11所示,对于上述记录个人信息的合约,执行的Grow和Rename操作最终都会以机器码的形式来保存,而合约中记录的姓名、年龄和身高等变量,则分别保存在变量存储表0x0000000、0x00000001和0x00000002这3个位置上(实际上,存储位置和数据格式是由合约代码负责安排的,这里是简化情况)。最终,对整个变量存储表计算得到哈希值0xb39a372c93a8b8b970e359a978fba643f94ac966c0d862e27da7770d8f485396,这个哈希值将会保存到合约账户中。

图3.11 合约账户的数据存储与结构

3. 合约地址的生成

合约地址并不取决于外部的公钥,而是通过特定的算法计算得到。在以太坊中提供了两种生成合约地址的方法:一种是通过合约创建者的地址和Nonce计算得到;另一种是通过合约创建者地址、指定的初始化值和合约代码的哈希值计算得到。

对于创建者地址 0x238661F085A338F04B0C7C956A796B57018151F0 和对应的 Nonce 值 0，序列化后的数据为 0XD694238661F085A338F04B0C7C956A796B57018151F080，其中这个序列化的方法是以太坊自定义的 RLP(Recursive Length Prefix)编码格式。将序列化数据通过 SHA256 计算之后，可以得到一个长度为 256 位的哈希值，取最后的 160 位，便可以得到新建合约的地址 0x5499F82BE656085c9636d85b559df2B17d5db33A。

由于创建合约的时候需要使用到 Nonce 值，当出现合约创建合约的情况时，创建其他合约的合约账户的 Nonce 值便需要改变，不然这个合约账户每一次计算得到的新合约地址都是相同的。

对于第二种创建合约地址的生成算法，则是变更哈希的输入参数为确定的已知值，使得开发者能更好地确定部署合约的地址。

3.4.3　驱动智能合约

在状态模型的框架下，以太坊的状态通过交易来改变，智能合约的状态变化同样使用交易来实现。在整个合约的生命周期中，所有的状态变换都是通过执行特定交易来实现的，智能合约的每次运行都是通过交易来驱动的。

1. 调用合约

在一次交易中，以太坊按照事先的约定执行智能合约的代码，最后得到运行的结果，比如修改了某些合约的数据。其中，智能合约作为交易的接收方，按照交易发起者指定的函数和参数进行执行。这个过程与计算机程序中的调用过程并无太大差异，这些接收者是合约账户地址的交易，因而也被称为合约调用。

为了实现调用过程中指定智能合约的不同函数以及携带函数的参数，以太坊在交易中加入了 data 字段用于存放这些数据。在现有的合约二进制接口规范(Application Binary Interface，ABI)约定中，编译时使用函数名与参数类型构成的字符串的哈希值作为调用过程中函数的索引，调用函数需在这个哈希值的后面附上经过序列化编码的参数。

如图 3.12 所示，如果要调用上述信息合约中的 Grow 函数，首先要知道合约账户所

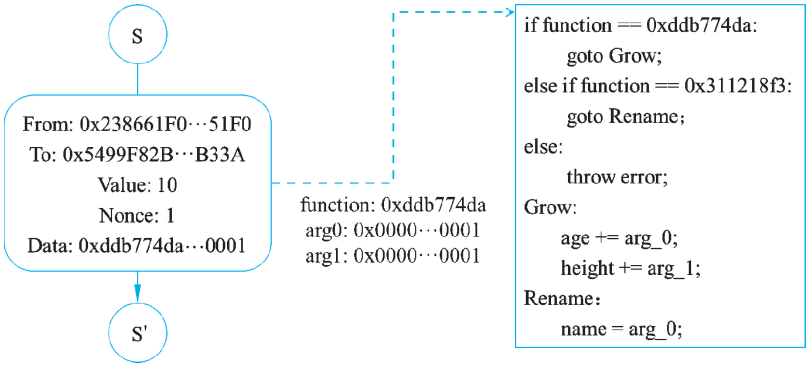

图 3.12　智能合约的调用

在的地址,其次还需要知道被调用的合约函数,Grow 包括参数类型的形式为 Grow(int256, int256)。然后计算这个字符串 SHA256 哈希值并取前 64 位,也就是 0xddb774da。于是可以构造一个交易的 data 为 0xddb774da0000…00010000…0001,其中 0xddb774da 便是要调用的函数,而 0x0000…0001 和 0x0000…0001 都是 256 位的参数,也就是进行了 Grow(1,1)这样一个函数的调用。

2. 创建合约

从前面的账户结构上知道,智能合约的代码也是存储在账户之中的。账户中的代码从无到有同样也是一种状态的变化,同样可以通过发送交易来实现将代码存储到以太坊的合约账户中,这个过程便是合约的创建,也称为合约的部署。

然而,发送交易时,当然不会有合约账户地址和合约账户,更不要提有可以被执行的智能合约代码。为了解决这个问题,以太坊专门特殊化了创建合约的交易,做了以下两点特殊化的处理。

(1) 创建合约的交易没有接收地址,交易中的 To 字段始终为 0。同样地,对于一个接收地址为 0 的交易,以太坊都会认为是一个创建合约的交易。然后,在检查交易无误之后,以太坊会根据发送者的地址和 Nonce 值来计算出这个交易创建的合约的账户地址。

(2) 交易的 data 字段不再是作为执行过程中的参数,而是直接运行交易 data 字段中的内容。也就是,创建合约时,需要将合约的代码和一些初始化的代码放到交易的 data 字段中,经过运行之后便可以得到合约的初始状态。

3. 停机问题与 Gas

以太坊的状态模型很好地契合了智能合约的运行,但还是存在一个很严重的问题。如果一个交易调用的智能合约存在死循环的话,那么这个交易的执行将不会停止,这对于整个以太坊系统来说是一个严重的打击。然而,对于一个计算机程序,或者说是一个图灵机而言,任何人都不能判断它是否能够在有限的时间内结束运行——这便是非常著名的停机问题。英国科学家图灵(Alan Mathison Turing)在 1936 年便证明了停机问题是不可解的。

为了规避不可能实现的停机问题,以太坊使用了 Gas 机制来保证智能合约能够在有限时间内能够终止。可以把通过 Gas 实现停机的想法类比成汽车的运行。对于一个不需要燃油的永动车和永不疲倦的超人驾驶员来说,不知道它何时会自己停下来。然而,对于现实中的汽车而言,只要汽车在不停地开动,就会每时每刻消耗燃油。由于携带燃油是有限的,那么汽车能够运行的时间必然有一个上限,最坏的情况也就是消耗完所有的燃油。

同样地,以太坊智能合约运行的每一个操作(例如计算、存储等)都规定了需要消耗的 Gas 的数值,并要求交易的发起者预先支付 Gas 额度。每次运行智能合约代码时,每一步操作都会消耗掉一些预先支付的 Gas 值,直到交易中预支付的 Gas 额度被消耗殆尽。这

样一来，交易中预支付的 Gas 值可以为将要执行的智能合约逻辑设置好一个确定的时间上限，而不会存在永不停机导致以太坊无法继续运行的安全问题。

4. 以太坊虚拟机

然而，除了停机问题外，智能合约的代码以何种格式存储和运行也是一个必须考虑的问题。对于去中心化的网络来说，参与的节点类型可能各有千秋，使用高级语言可能导致不同的执行结果，而直接使用机器码又会跟特定架构相关，折中的办法便是使用统一的虚拟架构和机器码。为此，以太坊引入了自己的虚拟机 EVM(Ethereum Virtual Machine)和相关的机器码。

EVM 是一个 256 位的栈虚拟机。其中，256 位，是指执行过程中的数据宽度是 256 位，相比之下普通的 x86 或者 ARM 架构通常是 32 位或者 64 位；栈虚拟机，是指 EVM 的执行流程基于一个栈结构，所有的指令都是操作栈顶的数据。

通常，开发者会使用类似于 Solidity 或者 Vyper 这些上层的高级语言来编写智能合约的代码，然后将其编译成 EVM 能够识别的机器码指令，最后才能够在以太坊的平台上使用 EVM 执行。

3.5　以太坊交易

3.5.1　交易内容

在以太坊中，交易承载了账户转账和创建、调用合约等功能，数据的内容更为复杂，大体上可以粗略地分为 3 个部分，即基本的交易、驱动的智能合约和交易的签名，更为详细的属性如下。

(1) From：交易发送者的地址。发送者地址可以通过合约的签名信息<r，s，v>计算得到，实现上交易的数据结构中并不会存储发送者地址。

(2) To：交易接收者的地址。在进行转账时是接受转账金额的地址，在创建合约时设置为空，也就是 0x0000…000，在调用合约时则是合约的地址。

(3) Value：交易的金额，单位是 Wei。Wei 是以太币最小单位，常说的 1 以太币是单位 Ether，1 Ether $=10^{18}$ Wei。

(4) Data：交易附带数据，传递创建合约的代码和构造函数，或调用合约的函数及参数。

(5) Nonce：交易发送者累计发出的交易数量，用于区分一个账户的不同交易及顺序。

(6) GasPrice：发送者支付给矿工的 Gas 的价格，用于实现从 Gas 到以太坊货币单位的转换，从而计算使用的 Gas 的总价格。

(7) GasLimit：该交易允许消耗的最大的 Gas，用于解决智能合约不能停机的问题。

(8) Hash：由以上字段生成的哈希值，也作为交易的ID。

(9) r、s、v：用于ECDSA验证的参数，由发送者的私钥对交易的哈希做数字签名生成，用于确认转账的合法性。

注意，交易本身不携带时间戳(Timestamp)属性，一般属智能合约开发者会以交易打包进区块的时间戳作为其执行时获取的时间戳。

3.5.2 交易费用

Gas机制不仅可以保证合约停机，同时可以对交易执行的成本进行归一化计算，以太坊中通过Gas进行计算交易的费用。以太坊的Gas机制有以下4个主要的概念。

(1) Gas：以太坊中资源消耗的基础单位。

(2) GasLimit：允许消耗的最大Gas值。

(3) GasUsed：执行后交易消耗的Gas值。

(4) GasPrice：用户为消耗的每个Gas单位支付的以太币。

在交易的执行过程中，每笔交易都带有基础Gas消耗值，用户在创建或调用智能合约的过程中，对以太坊虚拟机的不同操作都将消耗不同值的Gas，基础的交易Gas值加上以太坊虚拟机运行时的Gas消耗值，即构成了交易的GasUsed。

交易的GasUsed是实时计算的，即以太坊虚拟机的每步操作都将计算累积一次，如果交易的GasUsed超过了用户定义的GasLimit，则判定为Gas不足，交易执行失败。交易执行完成后，将得到交易的GasUsed乘上GasPrice，即为用户该笔交易应付的手续费，这一手续费从交易发起账户扣除，加到区块Coinbase账户中。换言之，挖到区块的节点除了得到区块奖励外，还将得到运行以太坊智能合约的手续费。同样地，区块中也带有GasLimit和GasUsed字段。

但是，Gas机制也存在一定的弊端。以太坊虚拟机的每个操作的定价是以太坊社区的开发者决定的，定价的合理性也时常受到质疑。在以太坊200多万的区块高度上，曾经出现过针对Gas定价不合理的攻击。然而，在去中心化的网络中要测量某段代码的软硬件资源消耗，还要保证这一测量方法为大众所容易知晓和认可，本身就是较为困难的，Gas机制经过以太坊社区的不断调整，也不断地在朝合理的方向优化。

3.5.3 交易的周期

如图3.13所示，为了便于读者理解，这里将以太坊交易在网络中的周期分为发起、广播、打包与执行、验证与执行4个阶段。

1. 发起

用户在本地的以太坊钱包软件中选择要发送交易的地址(From)，输入目标地址(To)、金额(Value)、是否部署或调用合约(Data)、手续费单价(GasPrice)等，确认发送至以太坊节点，节点和钱包可以是同一台物理服务器，也可以是分离的，即多个用户各自保

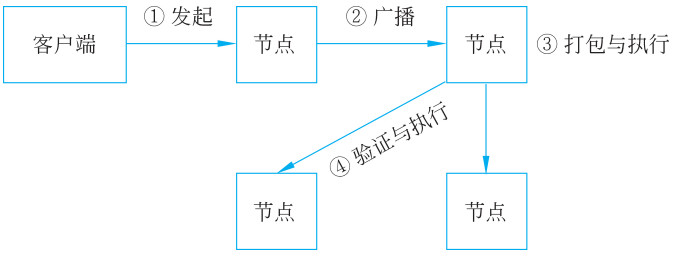

图 3.13　以太坊交易的周期

有钱包私钥，而通过同一个以太坊节点广播交易。

一般，以太坊钱包软件将自动为用户得出交易所需的燃料上限（Gaslimit），并给出用户地址对应的 Nonce 值，最后使用私钥得到 r、s、v，最终将交易序列化后发送到网络中。而在部分客户端中，Gaslimit 与 Nonce 也可以是用户自定义的。

2. 广播

节点收到（或自己发起）交易后，会对交易进行验证。验证的内容包括：交易的签名、交易的发起账号的余额是否能支付转账金额与手续费、交易的 Nonce 值是否为账号已发出的交易数等。节点验证交易为合法交易后，将交易加入节点的交易池中。

交易池中存储着待打包的交易，交易经过验证并暂存到交易池的这一过程对区块链的数据结构本身没有影响。受限于节点资源，节点可暂存的待打包交易数量通常是有限的，一般可在启动以太坊程序时配置这一交易池数量的最大值。

节点验证交易通过后，除了加入节点的交易池中，还会根据 P2P 网络广播的策略向相邻节点继续广播该交易。

3. 打包与执行

交易进入内存池后，具有挖矿功能的全节点，开始打包下一个区块。一般情况下，节点从自身利益出发，会将交易池中的交易按 GasPrice 取出具有较高手续费的交易。在少数情况下，也存在部分节点只打包自己发起的交易的情况（如一些矿池提供商或交易所服务商的节点）。

节点将交易打包时，将对交易进行逐个执行，一般可以根据目标地址（To）值的不同，将交易分为以下执行类型。

（1）创建合约交易。To 为空的交易。对于创建合约交易，EVM 将会根据 From 值及 Nonce 值生成合约地址，执行 Data 中对应的智能合约代码（包含合约本身及其构造函数的代码），并最终将合约 EVM 代码存储到合约地址中。

（2）调用合约交易。To 为合约账户的交易。对于调用合约交易，EVM 将从世界状态中获取 To 地址中存储的 EVM 代码，并执行交易的 Data 字段中包含的代码。一般，To 地址中存储的是合约本身，而 Data 中则包含了调用合约的相应函数及其参数。本质

上来说,对合约的调用是对合约状态的修改。

(3) 普通转账交易。To为人控制的账户(也称为外部账户)的交易。这一交易的执行则是直接将以太币金额从From转到To。

每笔以太坊交易都是对以太坊状态的修改,而在每一笔交易执行后,会生成交易的收据,其中带有新建的合约地址、消耗的Gas总量、交易生成的事件日志(称为event或log,将在第6章中详细介绍)等。在所有需要打包的交易执行后,交易、状态和收据的信息也会打包到区块中。记账节点在打包交易并获得合法的区块后,将区块(包含交易数据)广播到网络中的相邻节点。

4. 验证与执行

没有获得记账权的节点,即未打包区块的节点,在收到广播的区块后,将对区块进行合法性的验证,并进行交易的执行。验证的内容与执行的过程与2、3中的相同,目的是保证智能合约执行的去中心化。

注意:在实际的通信中,即网络中节点互相传播的交易的原始报文中,并不需要包括Hash和From,因为Hash可以根据交易本身内容得到,而From实际是通过结合r、s、v等综合计算得到的。而交易Nonce值的存在及验证,则是为了保证交易发送者能够控制交易的确认顺序,因为在P2P网络中交易可能是乱序到达节点的。

在以太坊主网的发展过程中,社区又提出了EIP1559机制,对Gas计费机制进行了更新。但是,采用以太坊虚拟机作为底层组成的其他区块链(包括多种公有链、联盟链),对该机制的采纳态度不一,且该机制也引发了较大争议,在未来也可能面临更新。因此在本书中,不对该机制进行过多叙述,感兴趣的读者请自行查阅社区文档。

3.6 数据结构与存储

3.6.1 区块与叔块

以太坊的区块同样是打包成一批执行的交易,它的数据结构同样分成了区块头和区块体。如图3.14所示,区块体中除了交易组成的交易列表之外,还保存了由交易执行信息组成的收据列表,以及用于改进以太坊共识过程的叔块列表。对应地,区块头中则增加了收据列表和叔块列表对应的哈希值,以及用于记录以太坊状态的状态根(State Root),此外还有一个最长不超过100KB的额外数据,可以在挖矿时填入自定义的信息。

1. 世界状态

由于以太坊使用了状态模型,在区块中除了保存交易列表之外,还附带了当前区块中交易执行结束状态相关的信息。我们知道,在以太坊中账户内部存在着各自的状态,它们在特定交易下发生变化。如果将以太坊中所有存在的账户的状态全部汇总到一起,就可

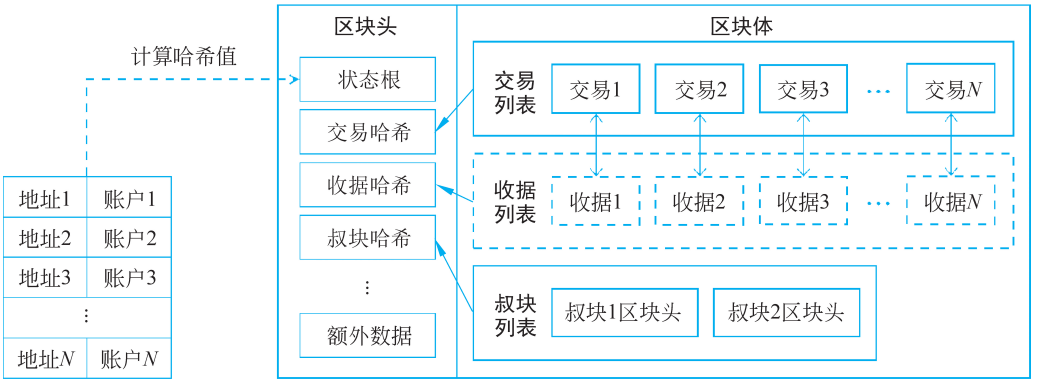

图 3.14　以太坊的区块

以得到一个全局的状态，它也称为世界状态。在区块头中，保存了对世界状态的一个信息摘要，也就是状态的哈希值。这个状态的哈希值在区块头中被称为状态根，这是因为状态根的计算是通过一个特殊的树状哈希数据结构来实现的，计算出来的哈希值处于这种树状哈希结构的根。

2. 叔块

以太坊定义了不在主链但被主链区块记录的满足难度的区块，这些区块称为叔块(Uncle Block)。一个叔块虽然不是主链的一部分，但也是合法的，只是其被发现得晚些或者网络传输慢些，导致其没能进入主链。叔块的设计是为了在尽可能减少两个相邻区块产生时间的条件下，尽量收缩和统一整个区块链的主链，同时通过叔块的激励来维护矿工的积极性。

如图 3.15 所示，创世区块后产生了三个分叉，其中中间的分叉被选为主链，其他两个区块可以被后继的区块当作叔块，被纳入主链中，获得一定的收益补偿(上面的区块获得 7/8 的区块奖励，下面的获得 6/8 的区块奖励)。其他矿工在打包区块时，可以自主地选择合法的 0～2 个叔块打包到新区块的区块头。同时，在以太坊的拜占庭硬分叉(Byzantium Fork，以太坊主链的一个重要升级)之后，拥有叔块的区块在进行难度计算时会有更多优势，进而在主链选举上有优势，进一步提高矿工纳入叔块的积极性。

3. 收据

收据是对应交易的数据结构，代表了交易执行的一些中间状态的写入和交易的执行结果等信息。以太坊的智能合约可以向虚拟机输出一些执行日志，这些日志就会被保存在收据之中。此外，收据中还会保存智能合约运行的 Gas 信息，以及单个交易执行完毕后以太坊的状态根。在交易的接收者是空，也就是交易创建智能合约时，如果执行成功还会把新建合约的地址写到收据中。

注意：对于一个以太坊节点来说，收据完全可以通过执行对应的交易来得到。因此，