第 3 章　区块链的技术范式与生态

提及对技术变化(变革)的描述,通常会想起两种理论,一种是需求拉动论,另一种是技术推动论。在需求拉动论看来,技术的主要推动力在于市场,在于市场中生产单元的“需求认知”,技术的变化用于实现或满足这些需求。技术推动论则主张技术本身是自治的,或半自治的。无论是需求拉动论或者是技术推动论,单一的强调市场或技术的作用都不能很好地阐释技术创新中既会出现渐进式的技术进步,又会出现所谓“变革式”的技术进步。

Giovanni Dosi 借鉴 Thomas Samuel Kuhn 在《科学革命的结构》中所提出的“科学范式”的概念,提出了“技术范式”(Technological paradigms)和“技术轨迹”(Technological trajectories)的概念,将它们作为知识与技术嵌入产业增长过程中连续与断裂之间的交互作用的一种隐喻[①]。在 Dosi 看来,从认识论的视角,技术范式是一个“愿景”,一个过程的集合,与相关问题及其解决方案相关。因此,Dosi 将技术范式定义为:基于所选择的自然科学原理和所选择的材料技术,针对所选择的技术问题所给出的解决方案“模型”与“模式”,而技术轨迹是在技术范式内技术发展的方向。通过定义技术范式,可以将渐进式的技术进步理解为技术轨迹下的累加式的技术发展,而将变革式的技术进步理解为技术范式的转换。

Dosi 认为技术范式对所遵循的技术方向给出了非常强的规范与指导,也使得范式内的工程师和组织对另外一些技术可能性“视而不见”。可以从四个维度来识别技术范式:第一维度涉及它所应用的通用任务;第二维度涉及它选择的材料技术;第三维度涉及它所利用的物理与化学特性;第四维度是与之相关的技术和经济维度以及折中。

依据技术范式理论,要界定区块链技术范式,可从以下几个方面入手:

---

① Dosi G, Nelson R R. *Technological Paradigms and Technological Trajectories*[M]. Research Policy, Volume 11, Issue 3, June 1982, Pages 147-162.

1. 所解决的关键技术问题

区块链作为一个技术范式，必然是围绕着所应用的通用任务。2008年中本聪的《比特币：一种点对点的电子现金系统》一文中就旗帜鲜明地提出了这一任务，我们可以称之为“中本聪计划”，即：“我们所需要的就是基于加密证明而非信任的电子支付系统，该系统允许交易双方在不需要可信第三方的前提下直接交易。”

对“中本聪计划”稍加修改，把其货币属性进行抽象，便可以得出区块链范式所应对的通用问题：“**在不需要可信第三方的前提下，交易双方就可以进行可信交易**。”其中，交易是广义上的事务处理，而可信交易指的是交易双方与交易有关的所有事物（包括交易时间、方式以及具体内容）都不能抵赖，以及交易双方的任意方都不能对其进行篡改等。

2. 所采用的解决方案模式

区块链将交易信息以加密区块的方式进行存储，同时以哈希的方式存储上一个区块的哈希值，从而形成一条全局有序的区块链。区块链技术也因此得名。

从技术轨迹而言，在区块链技术范式中，也有不同的技术路线。按照区块链上用户的权限控制，可以划分为公链、联盟链和私链。此外从技术体系而言，目前在全球范围内，也形成了几个主要的区块链生态，包括比特币生态系、以太坊生态系、石墨烯生态系、Hyperledger生态系和IPFS分布式储存生态系等。下面我们就这几个比较重要的区块链技术生态做一个相对详细的阐述。

3.1 比特币生态系

比特币生态系包括比特币以及其他一些分叉币（BCH、BTG、SBTC等）。下面主要介绍比特币。

比特币是一种数字加密货币，可以算是区块链技术的首个应用。第一笔 50 个比特币于 2009 年 1 月 3 日挖掘出来。比特币不需要印刷纸币，更不需要铸造硬币。它是完全无中心化的，没有政府、机构（如银行）或其他权力机构来控制。比特币的持有者也是匿名的，交易时不用使用真实姓名、真实银行账号等，而是通过加密密钥连接买卖双方。比特币不像传统货币一样从上而下发行。

3.1.1 比特币的体验

比特币核心源码地址如下：https：//github. com/bitcoin/bitcoin。

下面以 Ubuntu 16. 04 为例，来说明比特币核心的编译过程。当然也可以选择其他系统，其过程大同小异，具体可参考相应的官方文档。

由于编译过程占用内存较多，推荐选用 2G 以上 RAM 的机器来编译比特币核心软件。

以下是具体编译过程。

1. 准备操作系统环境

安装 Ubuntu 16. 04 操作系统，系统安装完毕，更新系统软件源（如果默认的国外软件源下载慢，可更新到国内的源地址），命令如下：

```
sudo apt update
```

2. 安装相关依赖包

命令如下：

```
sudo apt-get install build-essential libtool autotools-dev automake
pkg-config libssl-dev libevent-dev bsdmainutils python3
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:bitcoin/bitcoin
sudo apt-get update
sudo apt-get install libdb4.8-dev libdb4.8++-dev
```

3. 获取比特币核心源码

从 Github 上获取比特币核心的源码，选取稳定版本，目前稳定版本中的最高版本为 v0.16.0，命令如下：

```
cd ~
git clone https://github.com/bitcoin/bitcoin.git
cd bitcoin
git checkout v0.16.0
```

4. 编译

编译时可以根据自己的需求在./configure 后加上相应的参数。默认编译完后的可执行程序有 bench_bitcoin、bitcoin-cli、bitcoind、bitcoin-tx、test_bitcoin 等，命令如下：

```
./autogen.sh
./configure
make
make install
```

5. 运行测试

bitcoind 默认会安装在/usr/local/bin 中，命令如下：

```
bitcoind -printtoconsole
```

可看到比特币核心软件开始运行，会直接连到主网，不断下载同步区块信息。执行 Ctrl+C 命令可中止。

比特币是被连接到互联网的强大计算机“挖掘”出来的。

比特币经过近 10 年的发展，其源码也在不断地被优化与改进，现在被称做“比特币核心”，包括了数据存储、比特币钱包、交易、区块验证以及 P2P 网络等所有的方面。

3.1.2 比特币的认识

作为一种加密货币，比特币的总体思路在中本聪(Satoshi Nakamoto)的论文《比特币：一种点对点的电子现金系统》都已经做了比较清晰而准确的阐述。

从系统的边界来看，比特币主要面向的应用场景如图 3.1 所示，包括如下几种。

1. 钱包

比特币的用户主要是将比特币视为一种货币，因此可以在网络中用比特币进行货币的交易、转移、支付以及管理相关的资产等。

2. 交易所

作为一种数字货币，比特币可以在交易所与其他数字货币(甚至法定货币)进行交易和流通。

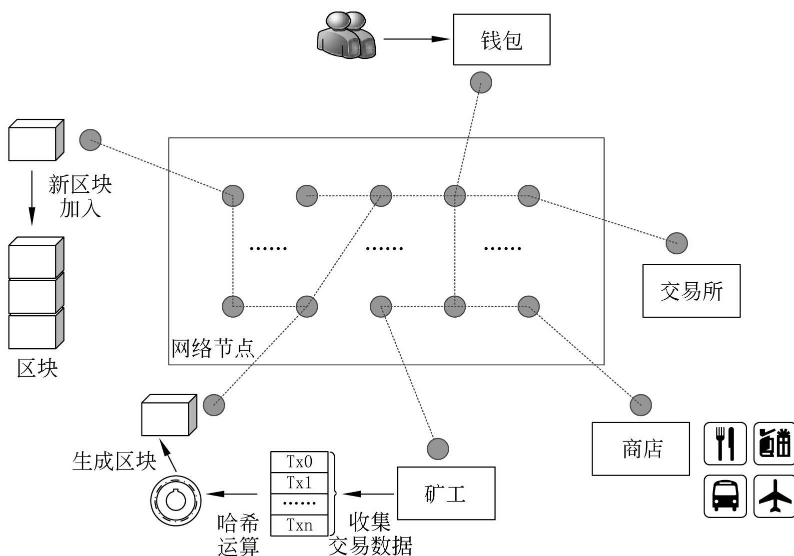

图 3.1 比特币的系统边界及应用场景示意

3. 商店

作为数字货币，比特币可以直接在线下的商店使用来购买商品。目前德国、日本、加拿大、阿根廷等国家与地区都承认比特币的合法货币身份。

4. 矿工

在比特币中，矿工是一种产生比特币的主要方式。犹如现实生活中挖掘黄金一般，有不少专门挖矿的计算设备或者组织机构。

从参考实现而言，比特币的实现主要涵盖如下几个部分。

1. 网络视角

从网络视角来看，比特币应该是一种点对点(P2P)网络系统。在点对点的网络系统中，基本单元是 Peer，涉及 Peer 的发现、Peer 之间的通信等。

2. 应用服务视角

从应用服务视角来看，外部应用可以通过远程过程调用(RPC)来调用钱包，或者与区块或交易相关的一些应用。

3. 内部构成视角

从内部构成视角来看，比特币主要包括四个核心实体部件和三个核心引擎，其中核心实体部件包括交易、区块、钱包、矿工，而三个核心引擎主要处理连接(连接管理器)、存储(存储引擎)和验证(验证引擎)。

连接管理器负责网络和外部应用与核心实体之间的交互；存储引擎负责区块、币等实体的序列化存储；验证引擎负责区块以及挖矿等的验证。

参考实现中的各种视角以及实体和引擎如图 3.2 所示。

3.1.3 比特币的技术范式解析

比特币是区块链技术范式的缔造者。正是比特币明确阐释了区块链技术所要解决的核心问题，也正是比特币系统地给出了对这一核心问题的解决方案：加密签名的区块，区块之间用哈希方法“铁索连舟”。通过这种“铁

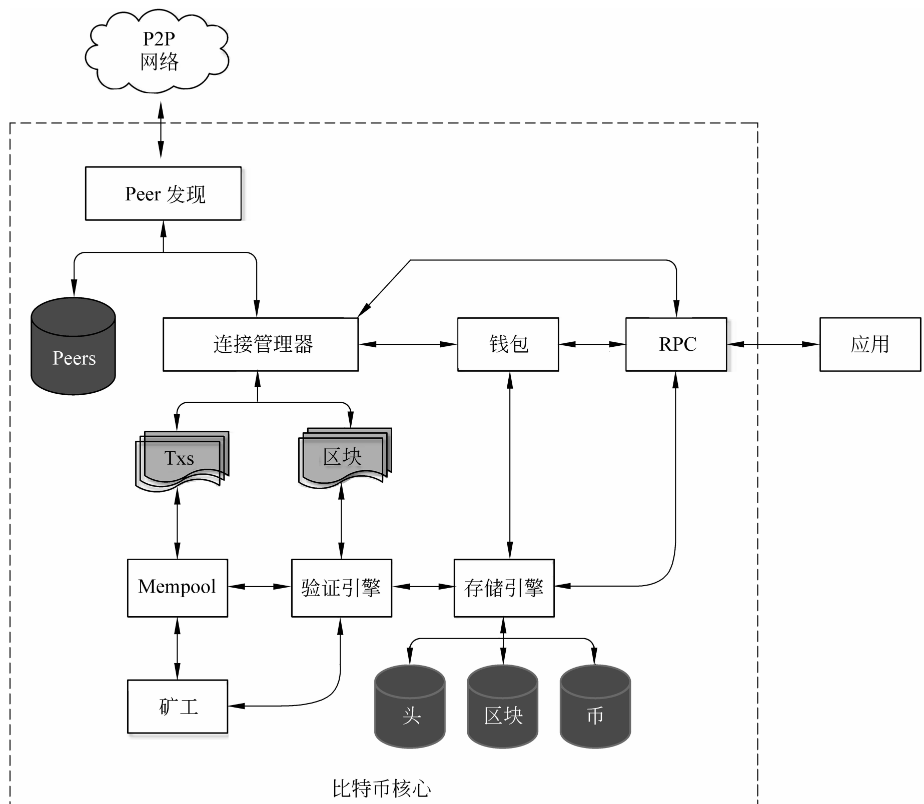

图 3.2 比特币核心架构[①]

索连舟”,每个区块之间相互依赖,相互验证,形成一条几乎不可攻破的信任链。

在这种范式中,核心是区块和区块链。比特币的区块和区块链的结构如图 3.3 所示。

每个区块由一个区块头及负载构成。区块头存储当前的区块头版本(nVersion),一个指向上一个区块的引用(HashPrevBlock),Merkle 树的根

① Antonopoulos A M. *Mastering Bitcoin*: *Programming the Open Blockchain*[M]. "O'Reilly Media, Inc.", 2017.

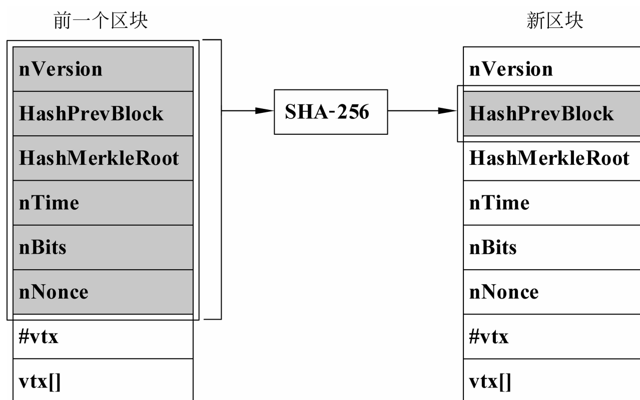

图 3.3　比特币中的区块及区块链

节点(HashMerkleRoot),一个时间戳(nTime),一个目标值(nBits)和一个随机数(nNonce)。最后,负载存储包含在区块中的交易数量(＃vtx)和交易向量(vtx)。区块头中所包含的各字段及描述如表 3.1 所示。

表 3.1　比特币中的区块头字段

字段名	类型(大小)	描　　述
nVersion	int(4 字节)	区块格式版本(当前为 2)
HashPrevBlock	unit256(32 字节)	前一个区块头的哈希 SHA-256(SHA-256(SHA-256(nVersion ||..||nNonce)))
HashMerkleRoot	unit256(32 字节)	从所有交易中构造的 Merkle 树的顶层哈希
nTime	unsigned int(4 字节)	UNIX 格式的接近区块创建时间的时间戳
nBits	unsigned int(4 字节)	以精简格式的证明工作量问题的目标 T。完整的目标值推导如下: $T = 0xh_2h_3h_4h_5h_6h_7 * 2^{8*(0xh0h1-3)}$
nNonce	unsigned int(4 字节)	Nonce 允许求解工作量证明问题产生变化
＃vtx	VarInt(1～9 字节)	在 vtx 中的交易条目的数量
vtx[]	交易(可变)	交易向量

3.2 以太坊生态系

以太坊由俄罗斯的 Vitalik Buterin 于 2013 年首次提出。经历了近 5 年的发展，已成为区块链中最著名的几个生态之一。目前市场上的绝大多数的 Token，都是基于以太坊 ERC20 发行的。与比特币比起来，以太坊提供了一个带有内置的图灵完备语言的区块链，以太坊中的智能合约可以非常方便地发行项目 Token 和开发相应的 DApp。

3.2.1 以太坊的体验

以太坊中使用最广泛的客户端是 geth，geth 是 go-ethereum 开源项目的简称，使用 Go 语言编写，完全兼容以太坊协议。通过 geth 客户端，可以实现与以太坊网络进行连接、交互、账户管理、合约部署、挖矿等所有操作。

下面以 Ubuntu 16.04 为例，介绍 geth 的安装。

方法 1：使用 Personal Package Archives 直接安装。

命令如下：

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

方法 2：源码自行编译。

（1）下载源码，命令如下：

```
git clone https://github.com/ethereum/go-ethereum
```

（2）golang 安装。下载 golang 安装包，命令如下：

```
wget https://dl.google.com/go/go1.10.1.linux-amd64.tar.gz
```

解压至/usr/local 目录，命令如下：

```
tar -C /usr/local -xzf go1.10.1.linux-amd64.tar.gz
```

在 $HOME/.profile 文件中配置环境变量，命令如下：

```
export PATH=$PATH:/usr/local/go/bin
```

（3）编译 geth 源码，命令如下：

```
cd go-ethereum
make geth
```

（4）运行，命令如下：

```
build/bin/geth
```

此外，以太坊还支持用户通过自定义创世区块来搭建私有链，下面是搭建私有链的过程。

（1）新建一个目录，命令如下：

```
mkdir private-chain
cd private-chain
```

（2）建立创世区块文件 genesis.json。

编辑 genesis.json 文件，写入以下内容：

```
{
    "config": {
        "chainId": 15,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
```

```
    "difficulty": "200000000",
    "gasLimit": "2100000",
    "alloc": {
        "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance":
        "300000" },
        "f41c74c9ae680c1aa78f42e5647a62f353b7bdde": { "balance":
        "400000" }
    }
}
```

（3）初始化，命令如下：

```
geth --datadir path/to/private-chain init genesis.json
```

看到日志信息中显示“Successfully wrote genesis state”信息，说明初始化成功。

（4）启动节点，命令如下：

```
geth --datadir path/to/private-chain --networkid 15 console 2>>
output.log
```

--networkid 选项后面跟的数字表示这个私有链的网络 id。以太坊主网的网络 id 是 1，为了防止与主网冲突，运行私有链节点时需指定自己的网络 id。

这样，一个单节点的以太坊私链就搭建好了，可以进行创建账户、转账、挖矿、查看区块等操作。

下面是 geth 中常见的一些操作，详细内容可参见官方文档：

https://github.com/ethereum/wiki/wiki

创建账户，命令如下：

```
>personal.newAccount('123456')
```

查看系统账户，命令如下：

```
>eth.accounts
```

查看账户余额，命令如下：

```
>eth.getBalance(eth.accounts[0])
0
>eth.getBalance(eth.accounts[1])
0
```

开始挖矿，命令如下：

```
>miner.start()
```

停止挖矿，命令如下：

```
>miner.stop()
```

转账测试，命令如下：

```
>eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1],
value: web3.toWei(8, 'ether') })
```

以太坊也支持多节点集群搭建，为了保证多个节点能正常启动，需要确保以下几点：

（1）每个节点有不同的数据目录（--datadir）。

（2）每个节点有不同的端口，（--port 和--rpcport）。

（3）每个节点的 ipc 唯一或禁用 ipc（--ipcpath 或--ipcdisable）。

（4）节点需要建立连接。

启动第一个节点，命令如下：

```
geth - datadir path/to/private - chain00 - - networkid 15 - -
ipcdisable --port 30000 --rpcport 8200 console 2>>output0.log
```

新开一个终端以开启另一个节点，命令如下：

```
geth -datadir path/to/private-chain01 --networkid 15 --
ipcdisable --port 30001 --rpcport 8201 console 2>>output1.log
```

查看节点 2 的 enode 信息，命令如下：

```
>admin.nodeInfo.enode
"enode://ad307e052d......@192.168.33.1:30001"
```

查看节点 1 信息，命令如下：

```
>net.peerCount
1
```

使用 admin.addPeer 建立连接，命令如下：

```
>admin.addPeer("enode://ad307e052d......@192.168.33.1:30001")
true
```

参数是节点 2 的 enode 值，返回 true，说明成功建立连接。

再次查看节点数量，命令如下：

```
>net.peerCount
2
```

可以看到连接的节点数增加了 1 个，变为 2。

至此，多节点基本搭建完成，可以继续添加第 3 个、第 4 个节点。

3.2.2 以太坊的认识

从技术逻辑体系角度来看，以太坊的架构如图 3.4 所示，总共涵盖五个层次。

第一层（最底层）是基础设施，包括数据、加密以及数学，还有 Solidity 编程语言（在以太坊虚拟机上运行的程序语言）。

第二层是核心层，包括区块链、共识、矿工以及网络等核心组成部分。

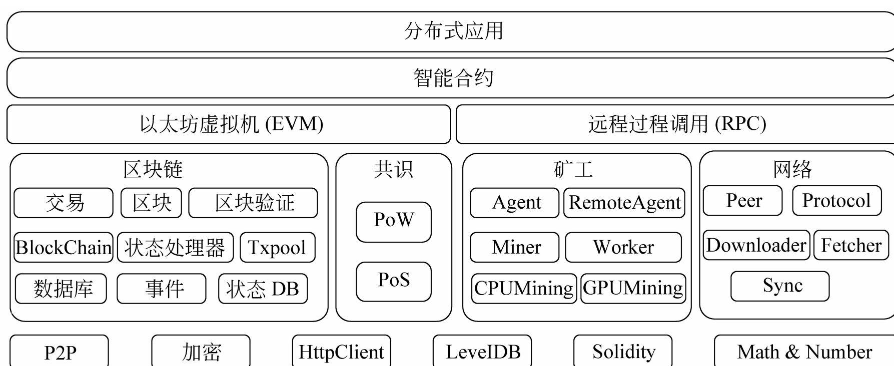

图 3.4　以太坊的逻辑体系架构

第三层是接口层，提供外部应用或者用户访问的接口或者系统调用，主要包括远程过程调用和以太坊虚拟机。

第四层是智能合约层，智能合约是以太坊的一个标志性部件，用户可以通过撰写智能合约来开发各类 DApp。

第五层（最上层）是应用层，指的是各类在以太坊上运行的分布式应用。

此外，从概念以及关系来看，以太坊中各种概念以及相互之间的关系如图 3.5 所示。

以太坊构建的本意在于融合并改进脚本、可替代 Token 和链上元协议等概念，同时允许开发者创建任意的基于共识的应用，这些应用同时还具有可扩展性、标准化、特征完备、易于开发且能互操作等所有特性。要实现上述目标，以太坊构造一个内置图灵完备编程语言的智能合约虚拟机，将其作为以太坊的抽象基础层。

根据《以太坊白皮书：下一代智能合约与分布式应用平台》的描述[①]，以太坊主要涉及的实体如下。

---

① Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. https://github.com/ethereum/wiki/wiki/White-Paper. 访问于 2018 年 4 月 21 日。

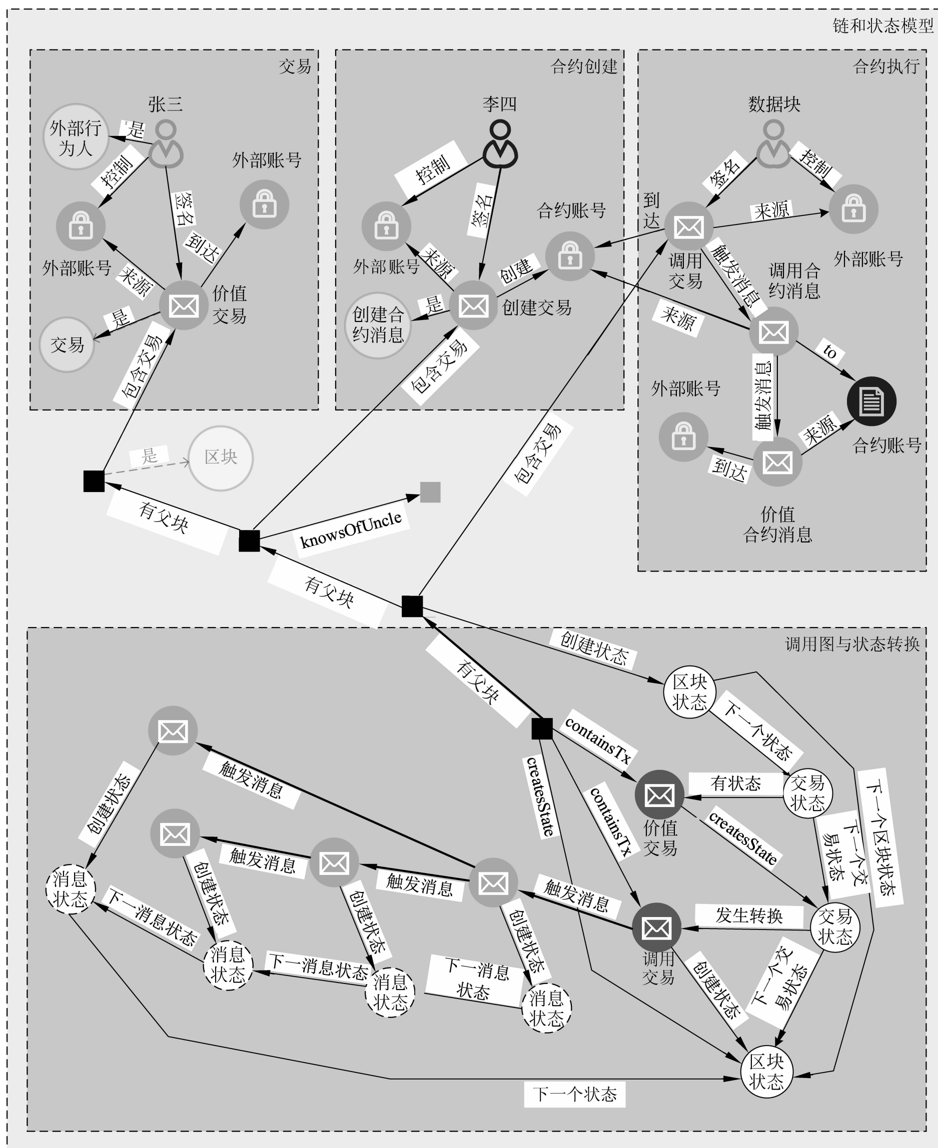

图 3.5　以太坊中的概念关系图

1. 以太坊账户

在以太坊系统中,“账户”是由20字节的地址和账户状态构成的,账户状态包含合同余额和完整的消息数量。

(1) 随机数,用于确定每笔交易只能被处理一次的计数器。

(2) 账户目前的Ether余额。

(3) 账户的合约代码(如果有的话)。

(4) 账户的存储(默认为空)。

Ether是以太坊内部的主要加密燃料,用于支付交易费用。账户包括外部账户和合约账户两种类型。合约账户有相关的存储状态和EVM代码。外部账户的地址从公钥和私钥对派生,而合约账户的地址是由创建者账户地址和它的Nonce串接而成。外部所有的账户没有代码,人们可以通过创建和签名一笔交易从一个外部账户发送消息。每当合约账户收到一条消息,合约内部的代码就会被激活,允许它对内部存储进行读取和写入,以及发送其他消息或创建合约。

2. 消息和交易

在以太坊中的消息是两个账户之间传递的数据(作为字节集合)和价值(作为Ether),通过自主对象(合约账户)的确定性操作或外部账户的机密安全签名。

以太坊中的消息在某种程度上类似于比特币的交易,但是两者之间存在三点重要的不同:第一,以太坊的消息可以由外部实体或者合约创建,然而比特币的交易只能从外部创建;第二,以太坊的消息可以选择包含数据;第三,如果以太坊的消息的接受者是合约账户,可以选择进行回应,这意味着以太坊消息也包含函数概念。

以太坊中的交易是两个账户之间传输Ether并包含负载的消息。交易总是来自于外部账号,通过私钥由外部行为者控制的。交易的执行创建了一个“交易单据”。

以太坊中的交易是指存储从外部账户发出的消息的签名数据包。交易包含消息的接收者、用于确认发送者的签名、以太币账户余额、要发送的数据以及两个称为 STARTGAS 和 GASPRICE 的数值。为了防止代码的指数型爆炸和无限循环，每笔交易需要对执行代码所引发的计算步骤（包括初始消息和所有执行中引发的消息）做出限制。STARTGAS 就是限制，GASPRICE 是每一计算步骤需要支付矿工的费用。如果执行交易的过程中，“用完了燃料”，所有的状态改变恢复原状态，但是已经支付的交易费用不可收回了。如果执行交易中止时还剩余燃料，那么这些燃料将退还给发送者。创建合约有单独的交易类型和相应的消息类型；合约的地址是基于账号随机数和交易数据的哈希计算出来的。

消息机制的一个重要后果是以太坊的“一等公民”财产——合约与外部账户拥有同样权利，包括发送消息和创建其他合约的权利。这使得合约可以同时充当多个不同的角色，例如，用户可以使无中心化组织（一个合约）的一个成员成为一个中介账户（另一个合约），为一个偏执的、使用定制的、基于量子证明的兰伯特签名（第三个合约）的个人和一个自身使用由五个私钥保证安全的账户（第四个合约）的共同签名实体提供中间服务。以太坊平台的强大之处在于，无中心化的组织和代理合约不需要关心合约的每一参与方是什么类型的账户。

3. 状态转换函数

以太坊的状态转换函数 ST（如图 3.6 所示）：

$$S_{t+1} = ST(S_t, T)$$

其中，S_t 表示前一个状态，S_{t+1} 是之后的状态，T 是交易。

以太坊的交易过程（状态转换函数）如图 3.7 所示。

（1）检查交易的格式是否正确（即有正确数值）、签名是否有效和随机数是否与发送者账户的随机数匹配。如否，返回错误。

（2）计算交易费用：fee＝STARTGAS * GASPRICE，并从签名中确定

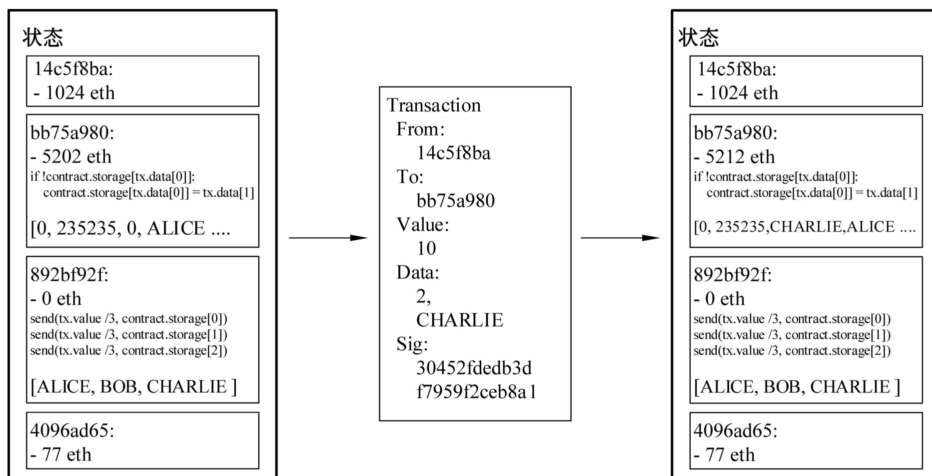

图 3.6 以太坊的状态转换图

发送者的地址。从发送者的账户中减去交易费用和增加发送者的随机数。如果账户余额不足，返回错误。

(3) 设定初值 GAS=STARTGAS，并根据交易中的字节数减去一定量的燃料值。

(4) 从发送者的账户转移价值到接收者账户。如果接收账户还不存在，创建此账户。如果接收账户是一个合约，运行合约的代码，直到代码运行结束或燃料用完。

(5) 如果因为发送者账户没有足够的钱或代码执行耗尽燃料导致价值转移失败，恢复原来的状态，但是还需要支付交易费用，交易费用加至矿工账户。

(6) 否则，将所有剩余的燃料归还给发送者，消耗掉的燃料作为交易费用发送给矿工。

例如，假设合约的代码如下：

```
if !contract.storage[msg.data[0]]:
    contract.storage[msg.data[0]] =msg.data[1]
```

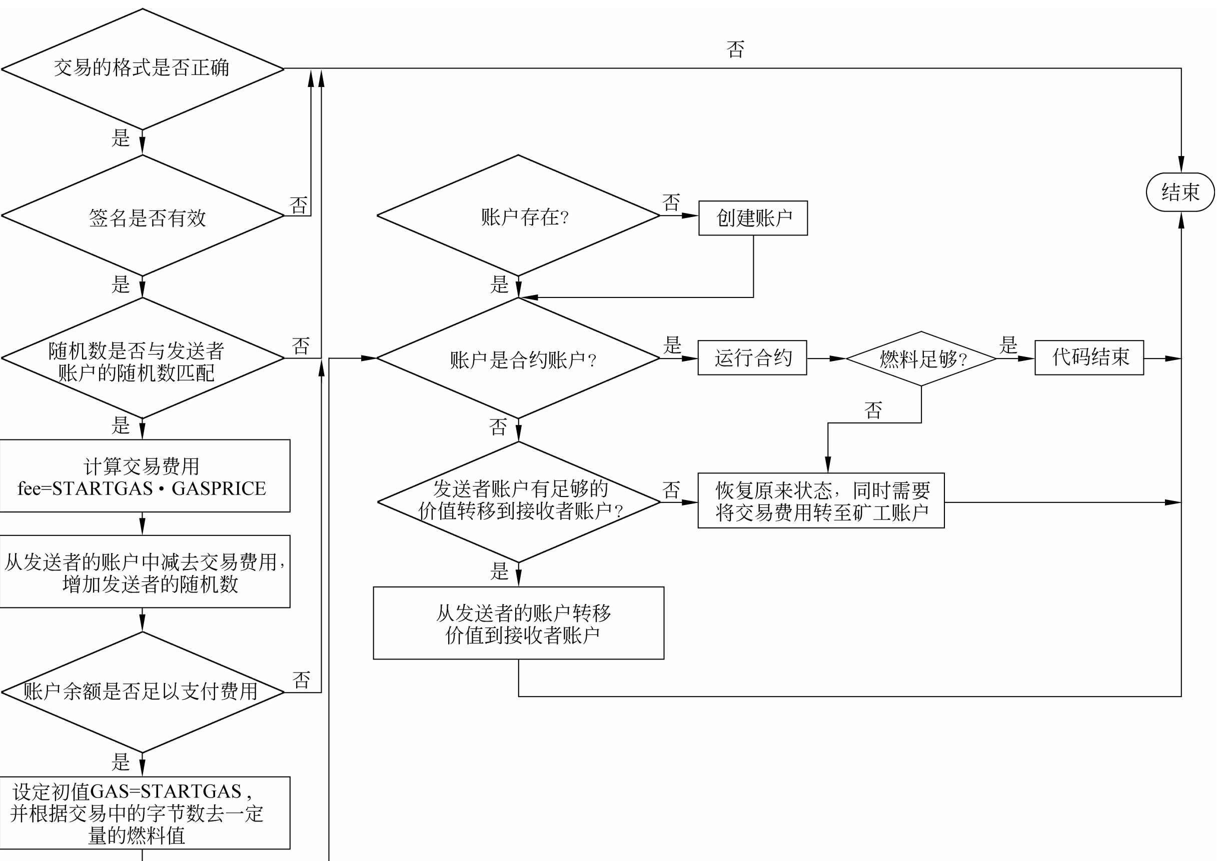

图 3.7　以太坊的交易过程