第11章 Proof of Work

工作量证明（Proof of Work）是一种在网络上所有节点共享正确信息的方式。本章将详细介绍工作量证明的内容，这是最重要的区块链技术之一。

11.1 Proof of Work

工作量证明（PoW）旨在确定诸如 P2P 网络这样的分布式系统中的正确信息。

11.1.1 Proof of Work 的过程

工作量证明的过程如图 11.1 所示。

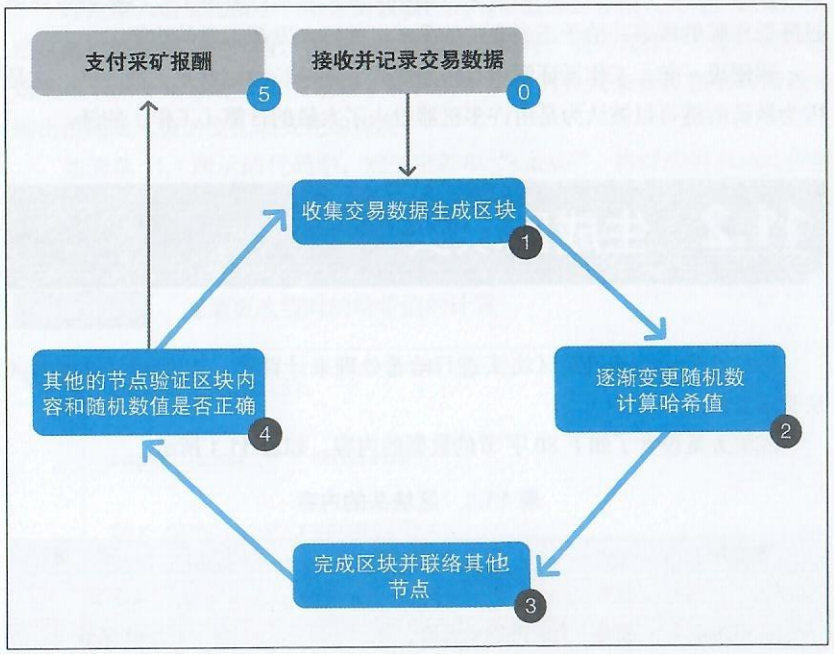

图 11.1 Proof of Work 的过程

可以看出，通过充分利用加密技术中哈希函数的特性，可以保持区块和其存储数据的完整性。

11.1.2 通过 Proof of Work 实现不易篡改的原因

区块链具有较强的防篡改的特性，其原因之一就是工作量证明。创建新的区块时，它将被链接在区块链的前端。例如，如果更改区块链中某部分的数据，则与其连接的区块中的所有数据都会改变，此时会执行大量的哈希计算。如果要使更改后的数据正确无误，则需要重新计算相关数据的哈希值并将其全部重写，为此，基于工作量证明需要投入大量的计算资源，即高性能的机器和大量的投资。此时，如果能够确保这些用来重新计算的资源，则进行常规采矿并获得采矿奖励在经济上被认为是合理的。

此外，如果发现了类似偷偷修改数据这样的欺诈行为，则比特币本身的价值可能会暴跌，所持有资产的价值将急剧下降。在这种背景下，区块链可以通过降低作弊的概率并给予正当合法的采矿以奖励来发挥正常的功能。

顺便说一句，工作量证明中有一条规则，即最长的链就是正确的链。这是因为最长的链可以被认为是由许多机器投入了大量的计算（工作）的链。

11.2 生成区块头

工作量证明是通过对区块头进行哈希处理来计算的。让我们尝试查看区块头的生成。

区块头是包含了如下 80 字节的数据的内容，如表 11.1 所示。

表 11.1 区块头的内容

数据项	说 明	长 度
Version	版本	4 字节
prev block hash	前面一个区块的哈希值	32 字节
Merkleroot	使用哈希函数摘要的交易的哈希值	32 字节
Time	表示生成区块的时间的时间戳	4 字节
Difficulty bits	采矿的难易度	4 字节
Nonce	符合采矿条件的 Nonce（随机数）的值	4 字节

通常情况下，在进行采矿时，在确定了除区块头的随机数之外的数据之后，通过逐渐改变随机数的值来计算哈希值。

11.3 通过更改 Nonce 计算哈希值

当生成了不包括 Nonce（随机数）的区块头时，可以一边更改 Nonce 一边执行哈希计算。此时，重复这样的操作直到找到满足特定条件的哈希值为止。

11.3.1 哈希计算的概念

即使输入值变化微小，哈希函数输出的哈希值也会有很大变化。因此，无法从输出值推断出输入值。矿工在执行哈希计算时，别无选择，只能一点一点地更改 Nonce 的值进行蛮力破解。让我们以编程的方式来看看在哈希函数中输出值随输入值的变化而变化的程度。

如清单 11.1 所示的代码中，对于字符串“satoshi”，将对应的 Nonce 的数字一个一个地附加进去，计算哈希值。这里的 Nonce 使用了 0 ~ 19 这 20 个数字。

清单 11.1 逐渐更改值时的哈希值的计算

输入

```
import hashlib

input_text = "satoshi"

for nonce in range(20):
    input_data = input_text + str(nonce)
    hash = hashlib.sha256(input_data.encode("UTF-8")).➡
hexdigest()
    print(input_data + " → " + hash)
```

在该程序中，在 input_data = input_text + str（nonce）语句中，将字符串 satoshi 和 Nonce 结合在一起，并通过 hashlib.sha256（input_data.encode（"UTF-8"））.hexdigest() 语句获得哈希值。Nonce 通过使用 for 语句重复执行从 0 到 19 这样的 20 次循环。运行该代码将产生如清单 11.2 所示的结果。

清单 11.2　逐渐更改值时的哈希值的计算

输出

```
satoshi0 → 4f9f790663d4a6cdb46c5636c7553f6239a2eca0319➡
4bb8e704c47718670faa0
satoshi1 → 0952ab89f7cb3010a3048adcbd58b43c9e82c61622a➡
c2d8893ff7c568f55c0bf
satoshi2 → 43c1badd8834f74a03471ef4f51e2d9f69c4fb9bbb4➡
76aeb2bd6ac4e41432aa1
satoshi3 → b8a25eafd4a22ba5d4510cfd5afe628a3497e42cd18➡
bf8d35e736c260d967a27
satoshi4 → be36eed769f4fa336843a5808bda6ef029d5cc9a917➡
e14bb38f6d740cefc29ac
satoshi5 → 1db1ebb2c528d0527b609b27c1d363f057e64d0c6e0➡
45dda339c860d726b23d8
satoshi6 → 445cb342afee90ebbd82490006a54f70646e1be2019➡
61d849c033bbaf36dcba1
satoshi7 → 8dabdf2893aa4693f36da21a5a32ddbf944f1a7e405➡
c5c5ce30cdfa2458b99e4
satoshi8 → ca35ad0c5f723110b34510b7396614f4d0b4a94081d➡
b918252009a53cf830da6
satoshi9 → 659edacbf0973449d9e1b3fa5c47f15411acd7ffa52➡
a8d334dd1c7fd546a4cd5
satoshi10 → 7388c143e1d4a951fb684798f39c30136019624eb8➡
a339e176c4fd6bae4f01ef
satoshi11 → 7ad1b4eb050e4538e9fde9ee378ce7f979bd922d1e➡
6ba9576c6ee5ac63648739
satoshi12 → 18d541974dd684ae87f69f18debafed9bd44ab50b7➡
6d27b5175ec580524681f0
satoshi13 → 6633993ec7430ef4ce49d2a541d40fd5a3fb406912➡
8d3626661bd75114909ad5
satoshi14 → cc53b18c134fc60dd99e8d7b6025996b3e782045f8➡
effb3d0a68992fbd65c2b3
satoshi15 → e22d46cf55a0b046cf0743da9a014b9c3bd9b50b30➡
085cbd0edb1648d483f4fe
satoshi16 → b66b71f3127b4bde44ca8007ad3ae52a56060e8877➡
de2b7b6fa6d0d53d412f3c
satoshi17 → 1289e0fa01d44d5d11a4c79a1508308c46d8e27089➡
8524312eb8325bdc73bb80
satoshi18 → 288e50d9d3ecdb9cce7ba388080d7adf01e91dfade➡
6ba78ee5f31c7642135c5d
satoshi19 → a4d8586fdc9529f56a53236f54f78b57b7538ccd08➡
dca50f327ca15d82ef05f6
```

从输出结果可以看出，即使 Nonce 是逐步一点一点变化的，输出的哈希值也是随机的，无法推测。

11.3.2 Difficulty bits 和 Difficulty Target

在 Proof of Work 中计算出的哈希值应满足的条件是怎样设定的呢？答案是在区块头的 Difficulty bits 里。在这个 4 字节的数据项中，记录了哈希值必须满足的条件。

Difficulty bits 是以 0x1e777777 的形式记录的。将它分解为前 1 字节和后 3 字节来进行说明，如图 11.2 所示。

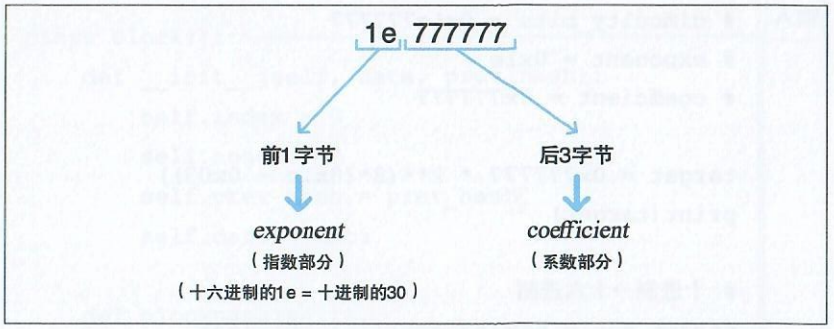

图 11.2　Difficulty bits 的思考方法（Difficulty bits 为 1e777777 的情况）

前 1 字节表示指数（exponent），后 3 字节表示系数（coefficient）或数值（value），可以将其理解为“coefficient 从右第 exponent 字节开始 32 字节的值”。在工作量证明中，区块头是通过 Double-SHA256 进行哈希处理的，因此输出的哈希值是 32 字节。成功的条件是 32 字节的哈希值小于“coefficient 从右第 exponent 字节开始 32 字节的值”。这时，“coefficient 从右第 exponent 字节开始 32 字节的值”被称为难度目标（Difficulty Target），如图 11.3 所示。

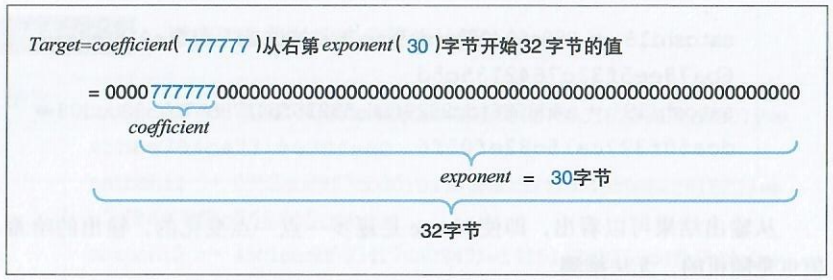

图 11.3　Target 的计算方法（Difficulty bits 是 1e777777 的情况）

实际上，从 bits 转换为 Target 时，使用以下的计算公式。

$$Target = coefficient \times 2^{8(exponent-3)}$$

基于此计算公式，转换“0x1e 777777”的程序如清单 11.3 所示。

清单 11.3　由 Difficulty bits 计算 Target

输入

```
# difficulty_bits = 0x1e777777
# exponent = 0x1e
# coefficient = 0x777777

target = 0x777777 * 2**(8*(0x1e - 0x03))
print(target)

# 十进制→十六进制
target_hex = hex(target)[2:].zfill(64)
print(target_hex)
```

清单 11.3 所示代码的输出结果如清单 11.4 所示。输出中上面的数值是 Difficulty Target 的十进制表示，下面的数值是十六进制表示。另外，十六进制总共 32 字节，除数值位外，其余位用 0 填充。.zfill() 是通过参数指定的位数，在右端填充 0 的方法。另外，使用 hex(target)[2:] 排除前两个字符，这两个字符表示十六进制显示的 0x。

清单 11.4　清单 11.3 的输出结果

输出

```
8245285810841765751455183746516157216732907363909109 93➡
533830975258099712
0000777777000000000000000000000000000000000000000000➡
0000000000
```

11.3.3 区块头的哈希处理

如清单 11.5 所示显示了一个程序示例，该程序生成一个区块头并通过逐步更改 Nonce 的值来计算哈希值。让我们看一下工作量证明过程。

清单 11.5　简单工作量证明的实现

```
import hashlib

class Block():
    def __init__(self, data, prev_hash):
        self.index = 0
        self.nonce = 0
        self.prev_hash = prev_hash
        self.data = data

    def blockhash(self):
        blockheader = str(self.index) + str(➡
self.prev_hash) + str(self.data) + str(self.nonce)    ①
        block_hash = hashlib.sha256(➡
blockheader.encode()).hexdigest()
        return block_hash

    def __str__(self):
        return "Block Hash: " + self.blockhash() + ➡
"\nPrevious Hash: " + self.prev_hash + "\nindex: " + ➡
str(self.index) + "\nData: " + str(self.data) + ➡
"\nNonce: " + str(self.nonce) + "\n--------------"
```

```
class Hashchain():
    def __init__(self):
        self.chain = ["00000000000000000000000000000000➡
00000000000000000000000000000000"]

    def add(self, hash):
        self.chain.append(hash)

hashchain = Hashchain()
target = 0x777777 * 2**(8*(0x1e - 0x03))

for i in range(30):
    block = Block("Block " + str(i+1), hashchain.chain[-1])
    block.index = block.index + i + 1
    for n in range(4294967296):
        block.nonce = block.nonce + n
        if int(block.blockhash(), 16) < target:
            print(block)
            hashchain.add(block.blockhash())
            break
```

②

清单 11.5 所示的程序由 Block 和 Hashchain 两个类组成。Block 类定义了区块的索引和前一个区块的哈希值等变量，如清单 11.5 ①所示。

在 blockhash 方法中，创建区块头之后，通过 sha256 对其进行哈希处理，然后获取区块头的哈希值。block_hash 作为返回值，如清单 11.6 所示。

清单 11.6　blockhash 方法

```
def blockhash(self):
        blockheader = str(self.index) + str(self.prev_hash) + ➡
str(self.data) + str(self.nonce)
        block_hash = hashlib.sha256(blockheader.encode(➡
'utf-8')).hexdigest()
        return block_hash
```

同时，使用特殊方法 __str__ 输出区块的信息。

在 Hashchain 类中，定义了对所计算的哈希值的处理。首先，将 chain 定义为数组变量。数组中存储的 32 个字节全部为 0，是因为在第一次采矿中作为 prev_hash 没有哈希值。add 方法表示将计算得出的哈希值存储在准备好的 chain 数组中。通过调用 .append() 方法将计算出的哈希值追加到数组的末尾。

hashchain 是 Hashchain 类的一个实例。另外，计算 target 则作为采矿的难易度。在这里，通过 11.3.2 小节中提到的 Difficulty bits 的 1e777777 计算 Target。

清单 11.7 中有双重循环的 for 语句。在第一行的 for 语句中，从 Block 类中实例化一个区块，并对 Block 实例的索引值进行递增处理。第 4 行的 for 语句中，通过依次改变 Nonce 来计算符合条件的哈希值。在这个 for 语句中，将 range 的参数指定为 4294967296，因为 Nonce 是用 4 字节的十六进制表示的，所以 Nonce 的值为 16^8=4294967296。另外，4 字节的十六进制数能表现的最大值是 ffffffff。

清单 11.7　采矿处理的实现

```
for i in range(30):
    block = Block("Block " + str(i+1), hashchain.chain[-1])
    block.index = block.index + i + 1
    for n in range(4294967296):
        block.nonce = block.nonce + n
        if int(block.blockhash(), 16) < target:
            print(block)
            hashchain.add(block.blockhash())
            break
```

当哈希值小于设定的目标（target）的值时，将增加区块并输出区块的数据。如清单 11.8 所示，通过使用 int() 将哈希值设为十进制整数，可以将其与同样是整数的目标（target）进行比较。int() 方法的第一个参数是字符串，第二个参数是用来指定转换为几进制的参数。在这种情况下，哈希值被视为十六进制数，并转换为十进制整数。

清单 11.8　哈希值和目标值的比较

```
if int(block.blockhash(), 16) < target:
```

如果不满足该条件，则将 Nonce 值加 1，然后重新计算。通过执行这样的循环，来实现挖掘采矿的过程。

11.3.4 无法找到满足条件的哈希值时

顺便说一下，Nonce 在区块头中占了 4 字节。由于是用十六进制来表示的，所以 Nonce 值大约是 43 亿，即 16^8。无法知道通过这个 Nonce 值是否能找到符合条件的哈希值。但在这种情况下，可以用以下几种方法进行调整和计算。

首先举一个简单的例子，调整区块头的时间戳。因为采矿是使用全世界的机器进行的，所以每个机器内的时钟相互有误差的可能性很高。如果在实际的区块链中使用的时间戳不正确，那么区块的前后关系和时间戳的前后关系也会出现矛盾。因此，可以通过一点一点地调整时间戳来调整哈希计算的值。

另外，也有利用 Extra nonce 的情况。Extra nonce 是指包含在区块头以外的随机数，具体来说是利用基于代币的交易的 input 领域中的 coinbase script 的 8 字节。由于基于代币的交易是交易数据，所以和其他数据一起汇总后的摘要要存储在默克尔根上。当然，如果更改基于代币交易的数据时，默克尔根的值也会发生变动，由此区块头的哈希值也会发生变化。也就是说，基于代币的交易数据也可以和区块头中的 Nonce 一样作为调整对象。

本章习题

问题一

请选择一个关于 Proof of Work 的错误描述。

（1）在 Proof of Work 中，SHA-256 仅对上一个区块的区块头应用一次。
（2）整个网络的哈希能力越高，难度越高。
（3）Proof of Work 是在分布式网络上定义唯一信息的过程。

问题二

请选择一个关于 Proof of Work 过程的错误描述。

（1）通过更改 Nonce 并重复哈希计算来找到满足条件的哈希值。
（2）在比特币区块链上，采矿大约需要 10 秒钟。
（3）从哈希值推断 Nonce 非常困难，因此要进行蛮力破解式计算。

问题三

请选择一个关于 Proof of Work 难易度的正确描述。

（1）难易度是以 4 字节的形式存储在区块头中的数据。
（2）Difficulty bits 被分解成前 2 字节和后 2 字节进行解释。
（3）难度越高挖掘报酬越高。