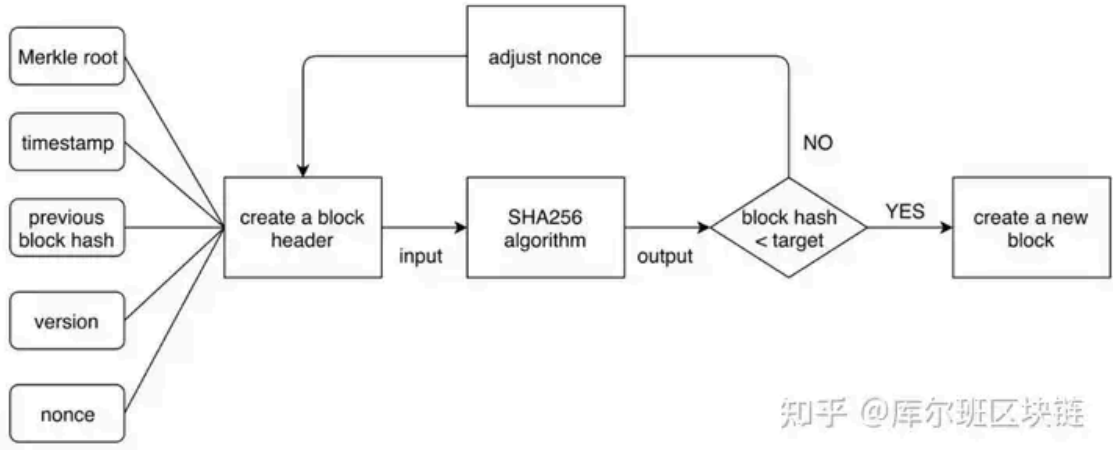
区块链共识机制 - POW 工作量证明 Proof Of Work 学习

> **工作量证明PoW（Proof of Work）**，通过算力的比拼来选取一个节点，由该节点决定下一轮共识的区块内容（记账权）。PoW要求节点消耗自身算力尝试不同的随机数（nonce），从而寻找符合算力难度要求的哈希值，不断重复尝试不同随机数直到找到符合要求为止，此过程称为“挖矿”。具体的流程如下图：
>
> 
>
>
> 第一个找到合适的nonce的节点获得记账权。节点生成新区块后广播给其他节点，其他节点对此区块进行验证，若通过验证则接受该区块，完成本轮共识，否则拒绝该区块，继续寻找合适的nonce。
>
> 来自--一文读懂主流共识机制：PoW、PoS和DPoS - 知乎 (zhihu.com)

这里通过`NodeJs`来实现--程序员来讲讲什么是区块链 | 小白也能听懂的通俗解释 | 区块链原理 | 比特币 | 数字货币*哔哩哔哩*bilibili

1、简易区块链搭建

区块链，顾名思义是由一个个区块相连接而成组成的**链式结构**

所以我们先定义Chain和Block类

```
const sha256 = require('crypto-js/sha256')

class Block{

    constructor(data, preHash){
        this.data = data // 区块储存的数据
        this.preHash = preHash // 储存上一个区块数据的哈希值
        this.hash = this.computeHash() // 当前区块数据的哈希值
    }

    // 生成该区块的Hash
```

```
    computeHash() {
        return sha256(
            this.data +
            this.preHash
        ).toString()
    }
}

class Chain{
    constructor(){
        this.chain = [this.makeGenesis()] // 一个链中应由多个区块组成所以赋值成数组
    }

    // 生成起始区间
    makeGenesis(){
        return new Block('Origin','') // 起始区块，索引为0，所以preHash为0
    }
}

const fanChain = new Chain()
const block1 = new Block('b1', '111')
console.log(fanChain)
console.log(block1)
```

运行得：

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      hash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c'
    }
  ]
}
Block {
  data: 'b1',
  preHash: '111',
  hash: '2e72de1974c5b2f96057efd46a0f777edb0305f58d3c9f977a26656c8843e01b'
}
```

此时链上只有**起始区块**

那要把我们定义的`block1`加到`fanChain`上，就要在`chain`上构造相应方法

```
////////// class Chain //////////

// 获得最新的区块
  getLatestBlock() {
    return this.chain[this.chain.length - 1]
  }

// 手动增加区块
  addBlock(newBlock) {
```

```
    newBlock.preHash = this.getLatestBlock().hash
    // 因为在外部无法得知此时的preHash是什么，所以手动赋值
    newBlock.hash = newBlock.computeHash()
    // computeHash()在Block类是构造时候调用的，但是newBlock传入的时候没有preHash，所以构造的Hash是错的，需要重新生成
    this.chain.push(newBlock) // 别忘了往链上添加
  }

const fanChain = new Chain()
const block1 = new Block('b1', '111')
const block2 = new Block('b2', '2')
fanChain.addBlock(block1)
fanChain.addBlock(block2)
console.log(fanChain)

```

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      hash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c'
    },
    Block {
      data: 'b1',
      preHash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c',
      hash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425'
    },
    Block {
      data: 'b2',
      preHash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425',
      hash: 'fa1a7c974fce650cd370836fbbccb8453a0b2dc3375e41443255838b69978878'
    }
  ]
}
```

可以看到：

- `b1`的`preHash`与上一个区块`Origin`的`hash`一致
- `b2`的`preHash`与上一个区块`b1`的`hash`一致
- 成功链接

这时候需要考虑一个问题，因为区块链具有不可篡改性

但是如果我们直接通过`fanChain.chain[1].data = 'fake'`进行更改data数据

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      hash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c'
    },
    Block {
      data: 'fake',
      preHash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c',
      hash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425'
    },
    Block {
      data: 'b2',
      preHash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425',
      hash: 'fa1a7c974fce650cd370836fbbccb8453a0b2dc3375e41443255838b69978878'
    }
  ]
}
```

会发现`b1`的数据被篡改了，但是`hash`依旧不变，所以我们要对链中的区块增加校验过程

校验要进行的过程有：

- 对比`Block`上储存的`hash`（也就是刚创建区块时候的`hash`）与通过当前`Block`数据再次生成的`hash`，以判断区块的`data`有没有被篡改
- 上一个区块的`hash`与当前区块的`preHash`是否一致，也就是是否构成链式结构

```
1  ////////// class Chain //////////
2
3  // 对区块进行校验
4    validateChain() {
5        // 因为起始区块没有preHash所以单独校验
6      if (this.chain.length === 1) {
7        if (this.chain[0].hash !== this.chain[0].computeHash()) return
   false
8        // 对初始进行校验，原始的Hash与再次生成的Hash进行对比
9        return true
10     }
11     for (let i = 1; i < this.chain.length; i++) {
12
13       // 判断当前Block的data是否被篡改
14       const blockToValidate = this.chain[i]
15       if (blockToValidate.hash !== blockToValidate.computeHash())
   return false
16
17       // 比对前一个Block与当前Block的hash值是否相同
18       const preBlock = this.chain[i - 1]
19       if (preBlock.hash !== blockToValidate.preHash) return false
20     }
21     return true
22   }
23
24 const fanChain = new Chain()
25 const block1 = new Block('b1', '111')
26 const block2 = new Block('b2', '2')
27 fanChain.addBlock(block1)
28 fanChain.addBlock(block2)
29 console.log(fanChain.validateChain())
30
31 // 尝试篡改区块
32 fanChain.chain[1].data = 'fake'
33 console.log(fanChain.validateChain())
34 console.log(fanChain)
35
```

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
true
false
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      hash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c'
    },
    Block {
      data: 'fake',
      preHash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c',
      hash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425'
    },
    Block {
      data: 'b2',
      preHash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425',
      hash: 'fa1a7c974fce650cd370836fbbccb8453a0b2dc3375e41443255838b69978878'
    }
  ]
}
```

可以发现`fanChain`在被人工篡改之后，被校验出来了

其次，我们也可以同时篡改区块的`hash`

`fanChain.chain[1].hash = fanChain.chain[1].computeHash()`

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
true
false
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      hash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c'
    },
    Block {
      data: 'fake',
      preHash: 'd3e1c5f8a6e240fc97f057908e6f99feebc73acb865b641a38e28d6ae9b1264c',
      hash: '776624efc798e98a147d5385e571a7e95a31179d5ac95509207a3ac5f0e2f395'
    },
    Block {
      data: 'b2',
      preHash: '79a9b412a92c884e8c340e2fc7c2ded7d24fb5c411f72575f5ae3619c0c79425',
      hash: 'fa1a7c974fce650cd370836fbbccb8453a0b2dc3375e41443255838b69978878'
    }
  ]
}
PS D:\UESTC\BlockChain\BlockChain-Learning>
```

但是发现依然校验失败，这是因为`b1`的`hash`改变了，所以导致`b2`的`preHash`与`b1`的`hash`不一致了，上下无法链接

2、实现工作量证明机制

上述实现了一个简单的区块链结构，我们只要通过`computeHash`算出哈希值就可以把区块加到链上

但是比特币中，需要生成的`hash`满足特定条件，比如哈希前缀三位全为0，等...

所以如果需要满足的条件越多，需要的算力就越大

- 这里我们定义`nonce`作为改变量
- 用`mine()`进行穷举比对

```
1 ////////// class Block //////////
```

```
2
3   constructor(data, preHash) {
4     this.data = data // 区块储存的数据
5     this.preHash = preHash // 储存上一个区块数据的哈希值
6     this.nonce = 0 // 挖（比对）的次数
7     this.hash = this.computeHash() // 当前区块数据的哈希值
8   }
9
10 // 生成该区块的Hash
11   computeHash() {
12     return sha256(this.data + this.nonce + this.preHash).toString() //
   增加nonce作为改变量
13   }
14
15 // 生成挖矿所比对的前缀 difficulty越大需要时间越长
16   getAnswer(difficulty) {
17     let Ans = ''
18     for (let u = 0; u < difficulty; u++) {
19       Ans += '0'
20     }
21     return Ans
22   }
23
24 // 挖！
25   mine(difficulty) {
26     while (true) {
27       this.hash = this.computeHash()
28       if (this.hash.substring(0, difficulty) ===
   this.getAnswer(difficulty))
29         // 进行前缀比对
30         break
31       this.nonce++
32     }
33   }
34
35
36 /////////// class Chain ///////////
37
38   constructor() {
39     this.chain = [this.makeGenesis()] // 一个链中应由多个区块组成所以赋值
   成数组
40     this.difficulty = 3 // 设置难度为3，也就是需要hash满足前缀三位全为0
41   }
42
43 // 手动增加区块
44   addBlock(newBlock) {
45     newBlock.preHash = this.getLatestBlock().hash
46     // 因为在外部无法得知此时的preHash是什么，所以手动赋值
47     newBlock.hash = newBlock.computeHash()
48     // computeHash()在Block类是构造时候调用的，但是newBlock传入的时候没有
   preHash，所以构造的Hash是错的，需要重新生成
49     newBlock.mine(this.difficulty) // 进行比对前缀
```

```
    this.chain.push(newBlock) // 别忘了往链上添加
  }


const fanChain = new Chain()
const block1 = new Block('b1', '111')
const block2 = new Block('b2', '2')
fanChain.addBlock(block1)
fanChain.addBlock(block2)
console.log(fanChain)
```

```
PS D:\UESTC\BlockChain\BlockChain-Learning> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      data: 'Origin',
      preHash: '',
      nonce: 0,
      hash: '3d7c3bbab3d81ae42007f0fb02d345a0821bff3f8af93ea0f248d3b95ae87d87'
    },
    Block {
      data: 'b1',
      preHash: '3d7c3bbab3d81ae42007f0fb02d345a0821bff3f8af93ea0f248d3b95ae87d87',
      nonce: 4649,
      hash: '00054fa3c49d794dba656f5f21fc368f00a3ff69d74f6409a59bc25efe522fbc'
    },
    Block {
      data: 'b2',
      preHash: '00054fa3c49d794dba656f5f21fc368f00a3ff69d74f6409a59bc25efe522fbc',
      nonce: 3687,
      hash: '000ab7120f71529c0b6d12a72a184e43ad9870ed407d2e71e0c3e091170b06e4'
    }
  ],
  difficulty: 3
}
```

观察发现在`nonce`为4649和3687时得到可行的`hash`

3、创造自己的数字货币

既然说货币，那便是用来交易的

所以我们可以把上述的`Block`类中的`data`改变为交易内容`Transaction`，以实现转账

所以我们追加一个`Transaction`类，既然是交易，那必然包含

- 转账人
- 收款人
- 数额
- 时间

```
class Transaction {
  constructor(from, to, amount, timeStamp) {
    this.from = from // 发送者公钥
    this.to = to // 接收者公钥
    this.amount = amount // 转账数量
    this.timeStamp = timeStamp
  }

    // 生成对应数据的hash
```

```
  computeHash() {
    return sha256(this.from + this.to + this.amount  + this.timeStamp
).toString()
  }
}

////////// class Block //////////

  constructor(transactions/* data */, preHash) {
    this.transactions = transactions // 本来是this.data = data
    this.preHash = preHash
    this.nonce = 0 // 挖的次数
    this.hash = this.computeHash()
  }

// 生成该区块的Hash
  computeHash() {
    return sha256(
      JSON.stringify(this.transactions) + // Transaction是对象，所以用
JSON.stringify将其字符化
        this.preHash +
        this.nonce
    ).toString()
  }
```

但是上述的时间戳`timeStamp`在比特币作者的论文里，是要在`block`中进行生成的，而不是在`Transaction`，所以进行修改

```
class Transaction {
  constructor(from, to, amount, timeStamp) {
    this.from = from // 发送者公钥
    this.to = to // 接收者公钥
    this.amount = amount // 转账数量
    // this.timeStamp = timeStamp
  }

    // 生成对应数据的hash
  computeHash() {
    return sha256(this.from + this.to + this.amount /* + this.timeStamp
*/).toString()
  }
}

////////// class Block //////////

  constructor(transactions/* data */, preHash) {
    this.transactions = transactions // 本来是this.data = data
    this.preHash = preHash
    this.nonce = 0 // 挖的次数
    this.hash = this.computeHash()
    this.timeStamp = Date.now() // 在Block类中进行生成
```

```
  }

// 生成该区块的Hash
  computeHash() {
    return sha256(
      JSON.stringify(this.transactions) + // Transaction是对象，所以用JSON.stringify将其字符化
        this.preHash +
        this.nonce +
        this.timeStamp // 在这里添加
    ).toString()
  }
```

上面写道把data改为Transaction交易数据

所以现在的这个区块链相当于一个池子（TransactionPool）装满了交易（Transaction）

```
////////// class Chain //////////

  constructor() {
    this.chain = [this.makeGenesis()] // 一个链中应由多个区块组成所以赋值成数组
    this.difficulty = 3 // 设置难度为3，也就是需要hash满足前缀三位全为0
    this.transactionPool = [] // 装载交易数据的pool
    this.minerReward = 50 // 奖励数额，随着挖的区块数量越多随之减半
  }
```

这时候我们回到先前的`addBlock`

```
// 手动增加区块
  addBlock(newBlock) {
    newBlock.preHash = this.getLatestBlock().hash
    // 因为在外部无法得知此时的preHash是什么，所以手动赋值
    newBlock.hash = newBlock.computeHash()
    // computeHash()在Block类是构造时候调用的，但是newBlock传入的时候没有preHash，所以构造的Hash是错的，需要重新生成
    newBlock.mine(this.difficulty) // 进行比对前缀
    this.chain.push(newBlock) // 别忘了往链上添加
  }
```

这个方法实际上是为了我们调试用的，实际运用中并不能直接增加

实际中，我们是在慢慢的挖区块，这里假定有个矿工在帮我们挖矿，当他挖到区块链便会获得相应的矿工奖励，奖励的发放依然通过`Transaction`来发放

```
// 交易池
  mineTransactionPool(minerAddr) {
    // 先给矿工发钱
    const minerTransaction = new Transaction('', minerAddr, this.minerReward) // 因为矿工奖励由区块链本体发放，所以地址为空
    this.transactionPool.push(minerTransaction) // 往pool储存
```

```
    // 开挖
    const newBlock = new Block(this.transactionPool,
this.getLatestBlock().hash)
    newBlock.mine(this.difficulty)
    // 把区块加到区块链当中
    // 同时清空此时的Pool
    this.chain.push(newBlock) // 把挖到的区块放入链
    this.transactionPool = [] // 当挖到合法区块，且放入链后，清空pool
  }

 // 方便添加Transaction到Pool
  addTransaction(transaction) {
    this.transactionPool.push(transaction)
  }


const fanCoin = new Chain()
const t1 = new Transaction('addr1', 'addr2', 10)
const t2 = new Transaction('addr2', 'addr1', 20)
fanCoin.addTransaction(t1)
fanCoin.addTransaction(t2)
fanCoin.mineTransactionPool('fan')
console.log(fanCoin)
console.log(fanCoin.chain[1].transactions)

```

```
Chain {
  chain: [
    Block {
      transactions: 'Origin',
      preHash: '',
      nonce: 0,
      hash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
      timeStamp: 1700928623382
    },
    Block {
      transactions: [Array],
      preHash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
      nonce: 2967,
      hash: '00078733ac109540ffa398247497fb15284a5d73bf0b5c1d7f55b6d57f97ecc7',
      timeStamp: 1700928623383
    }
  ],
  difficulty: 3,
  transactionPool: [],
  minerReward: 50
}
[
  Transaction { from: 'addr1', to: 'addr2', amount: 10 },
  Transaction { from: 'addr2', to: 'addr1', amount: 20 },
  Transaction { from: '', to: 'fan', amount: 50 }
]
```

4、数字签名

上节我们实现了简易的数字货币，但是可以发现Transaction（交易）可以由任何人发起，而且也可以不经过你的批准而使用你钱包的钱，~~这显然不行，怎么能动我的小金库~~

所以在发起转账前，需要做出相应的验证措施，这就要用到了本节的数字签名

这里我们使用的是非对称加密

- 通过公开`Transaction`，`Public Key`与`Signature`，可以让其他所有人校验这个签名的合法性，证明这个交易属于你
- 同时，只有验证通过才能执行转账操作，~~保护了你的小金库~~

```
const ecLib = require('elliptic').ec
const ec = new ecLib('secp256k1') // 椭圆曲线

////////// class Transaction //////////

  constructor(from, to, amount) {
    // if (from === null || from == '') // 视频里没说，但是我觉得应该检验一下（这样检验不太行
        // throw new Error('Transaction cannot be empty!')
    this.from = from // 这里从id改成了发送者公钥
    this.to = to // 同理，接收者公钥
    this.amount = amount // 转账数量
    // this.timeStamp = timeStamp
  }

  // 对交易进行签名
  // 签名的意义是将公钥和Transaction公开使所有人都能对此交易进行校验，凭此证明其合法性
  sign(key) {
    this.signature = key.sign(this.computeHash(), 'base64').toDER('hex')
  }

  // 检查交易合法性
  isValid() {
    // 对矿工奖励进行特殊判断
    // 因为矿工奖励是区块链给矿工转账，所以公钥是空
    if (this.from === '') return true

    const keyObj = ec.keyFromPublic(this.from, 'hex')
    return keyObj.verify(this.computeHash(), this.signature) // 检验Signature是否合法
  }

////////// class Chain //////////

// 方便添加Transaction到Pool
  addTransaction(transaction) {
```

```
    // 检验transaction是否合法
    if (!transaction.isValid())
      throw new Error('Found InValid Transaction before adding to pool!')
    this.transactionPool.push(transaction)
  }
```

```
// 签名测试
const fanCoin = new Chain()

const keyPairSender = ec.genKeyPair()
const keyPairReceiver = ec.genKeyPair()

const t1 = new Transaction(
  keyPairSender.getPublic('hex'),
  keyPairReceiver.getPublic('hex'),
  10
)
t1.sign(keyPairSender)
console.log(fanCoin)

console.log(t1.isValid())
```

```
PS D:\UESTC\ichunqiu> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Block {
  transactions: [
    Transaction {
      from: '043b01668e50f813b3ac352a90fc6f2361876a3d33a7fd7f93508f4f8cdf8d70eebc5544fd1ebb0c02f45d0c48cc7f64ce3f8520dca7c9236e9e46d167382d8ab9',
      to: '04456f7a241606e7ecf4bffbf545c55b7890176c840ab21229a39f78f142f560ca9bf13540482859 8a7b68379f59d48400658de39f12884b6b6d8fe9be5d782ced',
      amount: 10,
      signature: '3045022100fb4e0f59120a9ac66cdb80541f26e48f8c748547f5da8c927274faf895ba887102204a92f70b685ef961292b4c5a2caad659b7fa9ae5a36267d92d02e242d3ff2a85'
    },
    Transaction {
      from: '',
      to: '043b01668e50f813b3ac352a90fc6f2361876a3d33a7fd7f93508f4f8cdf8d70eebc5544fd1ebb0c02f45d0c48cc7f64ce3f8520dca7c9236e9e46d167382d8ab9',
      amount: 50
    }
  ],
  preHash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
  nonce: 5781,
  hash: '0005559281b4e5f934148d66bdc65443555bf88636f1f01d32999fd5327e9e71',
  timeStamp: 1702714891945
}
true
```

可以看到验证成功，现在我们手动篡改一下，尝试一下

```
// 签名测试
const fanCoin = new Chain()

const keyPairSender = ec.genKeyPair()
const keyPairReceiver = ec.genKeyPair()

const t1 = new Transaction(
  keyPairSender.getPublic('hex'),
  keyPairReceiver.getPublic('hex'),
  10
)
t1.sign(keyPairSender)
console.log(fanCoin)

console.log(t1.isValid())

t1.amount = 20
console.log(t1.isValid())

fanCoin.addTransaction(t1)
console.log(fanCoin)
```

把`amount`改成20

```
PS D:\UESTC\ichunqiu> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      transactions: 'Origin',
      preHash: '',
      nonce: 0,
      hash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
      timeStamp: 1702715908533
    }
  ],
  difficulty: 3,
  transactionPool: [],
  minerReward: 50
}
true
false
d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js:110
      throw new Error('Found InValid Transaction before adding to pool!')
      ^

Error: Found InValid Transaction before adding to pool!
    at Chain.addTransaction (d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js:110:13)
    at Object.<anonymous> (d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js:191:9)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47

Node.js v18.17.0
```

可以看到输出了`false`，并丢出来错误，且并没有往`Transactions`中放入错误的交易

这时候，在之前验证区块是否一一相连时，我们也应该加上验证

由于区块中可能有多个`Transaction`，所以采用穷举依次验证

```
1 ////////// class Block //////////
2
3
```

```
// 逐个验证transaction是否合法
  validateTransaction() {
    for (const transaction of this.transactions) {
      if (!transaction.isValid)
        // throw new Error("Found inValid Transaction!")
        return false
    }
    return true
  }

// 挖!
  mine(difficulty) {
    this.validateTransaction()
    while (true) {
      this.hash = this.computeHash()
      if (this.hash.substring(0, difficulty) === this.getAnswer(difficulty))
        // 进行前缀比对
        break
      this.nonce++
    }
  }
```

```
////////// class Chain //////////

// 对区块进行校验
  validateChain() {
    if (this.chain.length === 1) {
      if (this.chain[0].hash !== this.chain[0].computeHash()) return false
      // 对初始进行校验，原始的Hash与再次生成的Hash进行对比
      return true
    }
    for (let i = 1; i < this.chain.length; i++) {
      // 验证Block的每个Transaction是否合法
      if (!this.chain[i].validateTransaction())
        throw new Error('Found InValid Transaction while validating Chain')

      // 判断当前Block的data是否被篡改8
      const blockToValidate = this.chain[i]
      if (blockToValidate.hash !== blockToValidate.computeHash()) return false

      // 比对前一个Block与当前Block的hash值是否相同
      const preBlock = this.chain[i - 1]
      if (preBlock.hash !== blockToValidate.preHash) return false
    }
    return true
  }
```

```
// 签名测试
const fanCoin = new Chain()

const keyPairSender = ec.genKeyPair()
const keyPairReceiver = ec.genKeyPair()

const t1 = new Transaction(
  keyPairSender.getPublic('hex'),
  keyPairReceiver.getPublic('hex'),
  10
)
t1.sign(keyPairSender)
console.log(fanCoin)

console.log(t1.isValid())

fanCoin.mineTransactionPool(keyPairSender.getPublic('hex'))
console.log(fanCoin.chain)
```

```
  minerReward: 50
}
PS D:\UESTC\ichunqiu> node "d:\UESTC\BlockChain\BlockChain-Learning\ProofOfWork\test.js"
Chain {
  chain: [
    Block {
      transactions: 'Origin',
      preHash: '',
      nonce: 0,
      hash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
      timeStamp: 1702752608824
    }
  ],
  difficulty: 3,
  transactionPool: [],
  minerReward: 50
}
true
[
  Block {
    transactions: 'Origin',
    preHash: '',
    nonce: 0,
    hash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
    timeStamp: 1702752608824
  },
  Block {
    transactions: [ [Transaction] ],
    preHash: '2297836de0e614ae1a7508d45c2fc07ca93ffcfcb42edd59fade49af72a74bda',
    nonce: 2920,
    hash: '00034a5efeb00549ed2174eeb236e4361e6157d4755da47c2751238189484581',
    timeStamp: 1702752608859
  }
]
PS D:\UESTC\ichunqiu>
```

The End