

浙江财经大学

实验（实训）报告

项目名称 区块链的简单实现
所属课程名称 区块链
项目类型 短学期实验报告
实验(实训)日期 2021/7/20

学 院 数据科学学院
班 级 18 大数据
学 号 180110950601
姓 名 陈鸿达

指导教师 郑胃强

区块链的简单实现

摘要：2018 年 4 月以来，随着加密货币价格的回升，各种关于区块链的话题甚嚣尘上，各大媒体对其商业性质、技术原理、发展前景各个方面，都有各种非常深入的探讨。同时，新的项目、概念也层出不穷，让人眼花缭乱。本文认为“区块链”本质上还是一种软件架构上的创新。所以本文从 0 开始利用 python 语言对区块链进行实现，同时租用了阿里云服务器，将这条链公布到互联网上。

关键词：区块链；比特币；数字货币；python；flask；postman

第 1 章 区块链简介

经过这十年的发展，区块链已经发展成为全球最具影响力的创新技术之一。从金融行业、制造产品到教育机构的各行各业，都可能会被这项技术全面改造。它的三大最主要的特点是：

1. 去中心化交易

区块链的本质是一种分散在所有用户电脑上（即所谓的“分布式”）的计算机账本，每个分散的账本会记录区块链上进行的所有交易活动的信息。所以，它不需要一个集中的机构、网站、公司来管理这些信息。

2. 信息不可篡改，一旦写入无法改变

作为一个记录交易的账本，人们最不希望的是它被坏人恶意篡改。任何一个用户，都可以通过交易编号，访问区块链上发生的所有交易记录和注释。由于中本聪巧妙的算法设计，配合密码限制和共识机制，如果要修改区块链中的某一个数据，就必须更改其后发生的所有数据记录，计算量无可想象，非常庞大，几乎不可能实现。实际上，比特币诞生到现在已经 10 多年，每天都有无数的黑客绞尽脑汁攻击这个系统，但是从来没有发生过一起交易记录被篡改的事件，这足以证明了它的安全性。

3. 完全匿名

在互联网诞生初期，有句话说：“在互联网上，没人知道你是一只狗。”这强调的是在互联网上的匿名访问性。在区块链世界里，所有的账户（或者说“钱包”）都是通过一个密码来访问。如果你失去了密码，也就失去了账户里面的所有货币。

在现实生活中，如果你忘记了密码或者丢失了银行卡，你可以去银行柜台申请补办，手续很简单。但是在“去中心化”的区块链世界里，没有这样的“银行柜台”，谁也不知道你是谁，你也无法向别人证明你是某个钱包的主人。

当然，与大部分技术一样，区块链也是一个“双刃剑”，有它的不足之处：

1. 过度资源消耗

想要生成一个新的区块，必须要大量服务器资源进行大量无谓的尝试性计算进行“挖矿”，严重耗费电能（后文将详细介绍“挖矿”的过程）。

2. 信息的网络延迟

以比特币为例，任何一笔交易数据都需要同步到其他所有节点，同步过程中难免会受到网络传输延迟的影响，带来较长的耗时。

本文区块链应用的开发，可以分为以下几步：



第 2 章 环境配置：建立交易所

Mac 系统安装 anaconda 环境

```
aiked aer@aiked aerMac ~ % conda activate base
(base) aiked aer@aiked aerMac ~ % _
```

安装 flask 和 request 包

```
(base) aiked aer@aiked aerMac ~ % conda list | grep flask
flask                1.1.2                py_0
(base) aiked aer@aiked aerMac ~ % conda list | grep request
requests             2.24.0               py_0
```

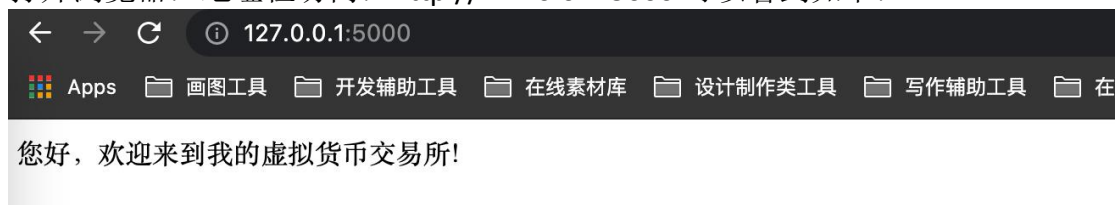
建立一个“交易所”帮助我们理解 flask 的作用

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return '您好，欢迎来到我的虚拟货币交易所！'
~
~
```

运行程序：

```
(base) aiked aer@aiked aerMac ~ % python -m flask run
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Jul/2021 11:00:34] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Jul/2021 11:00:34] "GET /favicon.ico HTTP/1.1" 404 -
```

打开浏览器：地址栏访问：<http://127.0.0.1:5000> 可以看到如下：



第 3 章 把“区块”，“链”到一起

“区块链”（Blockchain）可以理解为把一个个符合特定格式的区块（Block），按照一定的方法“链”（chain）到一起。

首先我们明确两个概念：

“类”（class）：类是面向对象程序设计中的概念，是面向对象编程的基础。类是对现实生活中一类具有共同特征的事物的抽象，譬如区块链这个概念，就是一类具有共同特征的事物，我们可以用一个类来代表它。类可以描述一个对象（在本文中即某个区块链）能够做什么，以及做的方法（method）。

哈希值（Hash）：所谓“哈希值”就是计算机可以对任意内容，计算出一个长度相同的特征值。区块链的哈希值长度是 256 位，这也就是说，不管原始内容是什么，最后都会计算出一个 256 位的二进制数字。而且可以保证，只要原始内容不同，对应的哈希一定是不同的。举例来说，字符串 123 的哈希值是 a8fdc205a9f19cc1c7507a60c4f01b13d11d7fd0（十六进制），转成二进制就是 256 位，而且只有 123 能得到这个哈希。（理论上，其他字符串也有可能得到这个哈希，但是概率极低，可以近似认为不可能发生。）由此可以得到两个重要的结论：

- 结论 1：每个区块的哈希都是不一样的，可以通过哈希标识一个区块。
- 结论 2：如果区块的内容变了，它的哈希一定会改变。

代码实现：

首先，我们要用一个构造函数来创建一个区块链类，其中包括两个表：一个用于存储区块，一个用于记录交易。另外，我们还要定义一个方法，用于生成区块的哈希值。此外，我们还为这个类定义了一个属性 `last_block`，这样可以通过调用该类获得区块链中最后一个区块的信息。下面是这个类的初步结构

```

class Blockchain(object):
    def __init__(self):
        self.chain = []
        self.current_transactions = []
    def new_block(self):
        # 创建一个新的区块
        pass
    def new_transaction(self):
        # 把新的交易添加到交易列表中
        pass
    @staticmethod
    def hash(block):
        # 生成一个区块的哈希值
        pass
    @property
    def last_block(self):
        # 返回链中的最后一个区块
        pass

```

这个类负责管理整个区块链，包括存储交易信息，把新的区块添加到整个区块链之中。下面，我们再来看看一个典型的区块是什么样的，以及它们是怎么构成一条区块链的。按照中本聪的原始定义，下面是一个典型的区块

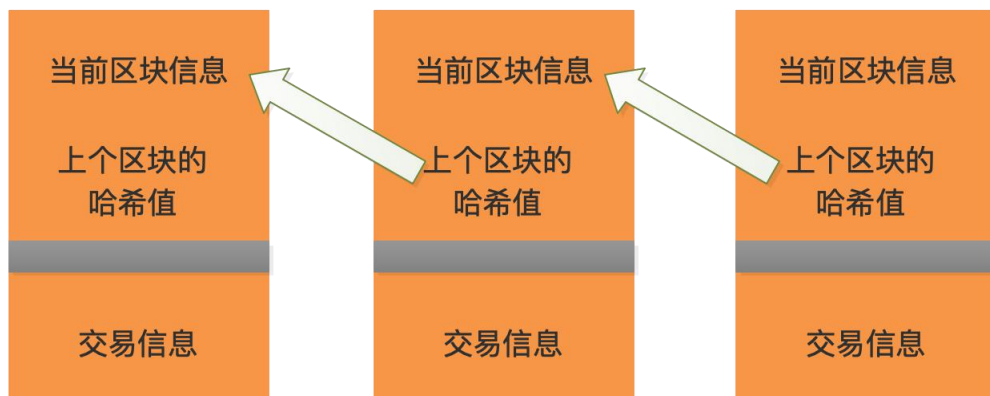
```

block = {
    'index': len(self.chain) + 1,
    # 区块编号，即区块链之前长度+1
    'timestamp': time(),
    # 区块生成时间的 UNIX 时间戳
    'previous_hash': previous_hash or self.hash(self.chain[-1]),
    # 上一个区块的哈希值，另外需要考虑第一个区块的情况
    'proof': proof,
    # 工作量证明（PoW），稍后会详细介绍
    'transactions': [
        {
            'sender': sender_hash,
            # 付款人钱包地址
            'recipient': recipient_hash,
            # 收款人钱包地址
            'amount': transactions_amount,
            # 交易金额
        }
    ],
    # 区块中的交易信息
}

```

由上面的定义可以看出，一个区块包括下面几项内容：

区块索引 / 编号、区块产生时间、上一个区块的哈希值、工作量证明（PoW）、交易信息



这揭示了区块链的核心理念：每个区块中包含了上一个区块的哈希值或者说信息。这点对区块链有重大意义：如果有人修改了一个区块，该区块的哈希值就变了。为了让后面的区块还能连到它，就必须依次修改后面所有的区块，否则被改掉的区块就脱离区块链了。由于哈希值的计算非常耗时（后面将解释原因），短时间内修改多个区块几乎不可能发生，除非有人真的掌握了全球网络中 51% 以上的计算能力。正是通过这种联动机制，区块链保证了自身的可靠性，数据一旦写入，就无法被篡改。这就像历史一样，发生了就是发生了，从此再无法改变。每个区块都连着上一个区块，这也是“区块链”这个名字的由来。

第 4 章 怎么给区块链添加交易记录

在建立起区块链之后，接下来我们看看怎么为其添加交易信息。所谓“交易”（Transaction）的过程，实际上就是给区块链加一笔数据更新的记录，其中包含了付款人的钱包地址、收款人的钱包地址、交易金额。如果把区块链作为一个状态机，则每次交易就是试图改变一次状态，而每次生成的区块，就是参与者对于区块中交易导致状态改变的结果进行确认。

具体的添加过程，是通过 `new_transaction` 这个方法来实现的。下面是对这个方法的定义实例。

```
def new_transaction(self, sender, recipient, amount):  
    self.current_transactions.append({
```

```

'sender': sender,
# 付款人的钱包地址
'recipient': recipient,
# 收款人的钱包地址
'amount': amount,
# 交易金额
})
return self.last_block['index'] + 1
# 返回当前区块链最后一个区块的索引值加1，即下一个区块的索引值

```

简单来说，每笔交易的核心，就是一句话，比如“张三向李四转移了 N 个比特币”。区块链作为一个数据库，记录了所有这些交易的信息。当然，为了证明这句话可行，需要给每笔交易加上数字签名，而后任何人都可以用张三的公钥，证明这确实是张三本人的行为。另一方面，其他人无法伪造张三的数字签名，所以不可能伪造这笔交易。

第5章 “挖矿”

上文我们介绍了交易的概念。但是，交易只是一笔笔财富转移的记录，并不会创造财富。那么，“真金白银”的比特币到底是从哪里来的呢？这就涉及到了“挖矿”。也就是说，每个比特币，都是通过挖矿这种活动所产生的。

在区块链网络中，存在着成千上万的网络节点。为了保证节点之间的同步，新区块的添加速度不能太快。试想一下，你刚刚同步了一个区块，准备基于它生成下一个区块，但这时别的节点又有新区块生成，你不得不放弃做了一半的计算，再次去同步。因为每个区块的后面，只能跟着一个区块，你永远只能在最新区块的后面，生成下一个区块。所以，你别无选择，一听到信号，就必须立刻同步。所以，区块链的发明者中本聪故意让添加新区块，变得很困难。他的设计是，平均每 10 分钟，全网才能生成一个新区块，一小时也就六个。

这种产出速度不是通过命令达成的，而是故意设置了海量的计算。也就是说，只有通过极其大量的计算，才能得到当前区块的有效哈希值，从而把新区块添加到区块链。由于计算量太大，所以快不起来。这个过程就叫做挖矿（Mining），因为计算有效哈希值的难度，就好像在全世界的沙子里面，找到一粒符合条件的沙子。计算哈希的机器就叫做“矿机”，操作矿机的人就叫做“矿工”。

之所以计算这个有效哈希值很难，是因为不是任意一个哈希值都可以，只有满足条件的哈希值才会被区块链接受。这个条件特别苛刻，使得绝大部分哈希值都不满足要求，必须重算。在区块中包含了一个难度系数（Difficulty），这个值决定了计算哈希值的难度。举例来说，第 100000 个区块的难度系数是 14484.16236122。区块链协议规定，使用一个常量除以难度系数，可以得到目标值（Target）。显然，难度系数越大，目标值就越小。然而，只有小于目标值的哈希才是有效的，否则哈希无效，必须重算。

由于目标值非常小，哈希值小于该值的机会极其渺茫，可能计算 10 亿次，才算中一次。这就是采矿如此之慢的根本原因。前面说过，一个区块的哈希值是

唯一的。如果要对同一个区块反复计算哈希值，就意味着区块必须不停地变化，否则不可能算出不一样的哈希值。区块里面所有的特征值都是固定的，为了让区块产生变化，中本聪故意增加了一个随机项，叫做 **Nonce**。

Nonce 是一个随机值，“矿工”的工作其实就是猜出 **Nonce** 的值，使得区块的哈希值可以小于目标值，从而能够写入区块链。**Nonce** 是非常难猜的，目前只能通过穷举法一个个试错。根据协议，**Nonce** 是一个 32 位的二进制值，即最大可以到 21.47 亿。第 100000 个区块的 **Nonce** 值是 274148111，可以理解成，矿工从 0 开始，一直计算了 2.74 亿次，才得到了一个有效的 **Nonce** 值，使得算出的哈希值能够满足条件。

运气好的话，也许一会就找到了 **Nonce**。运气不好的话，可能算完了 21.47 亿次，都没有发现 **Nonce**，即当前区块体不可能算出满足条件的哈希值。这时，协议允许矿工改变区块体，开始新的计算。

简单来说，工作量证明或者说挖矿算法，目的就是寻找一个特殊的数字，使得哈希值（即 SHA256 函数）的输出字符串的前 n 位是零。

现在，我们可以把这个算法简化成一个数学题：一个整数 x 乘以另一个整数 y 的积的哈希值必须以 0 结尾，即 $\text{hash}(x * y) = \text{ac23dc}...0$ 。设 $x=5$ ，求 y 的值？

下面是用 Python 求解这个数学题的代码：

```
from hashlib import sha256
x = 5
y = 0 # 从 0 开始穷举，直到生成的哈希值符合条件为止
while sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
    y += 1
print(f'y= {y}')
```

结果是：y = 21 因为，生成的 Hash 值结尾必须为 0。 $\text{hash}(5 * 21) = 1253e9373e...5e3600155e860$

这就意味着经过 22 次尝试，我们成功地找到了一个 **Nonce**，也得到了一个符合要求的哈希值，这样就完成了一次“挖矿”，可以生成一个区块了。

在此基础上，下面是一个简化版的挖矿算法（也就是工作量证明）的问题和求解代码：

找到一个使得 pp'的哈希值包含 4 个 0 的数字 p'
p 是上一个工作量证明值，p'是新的证明值

```
def proof_of_work(self, last_proof):
    proof = 0
    while self.valid_proof(last_proof, proof) is False:
        proof += 1
    return proof
```


要调节挖矿的难度，实际上就是控制需要哈希值包含的 0 的个数。实际上，四个 0 已经相当具有挑战性了，随着 0 的数目一个一个增加，需要的计算时间将会呈指数级增长。

第 6 章 在网络中传播区块链：共识算法

上文我们学会了怎么在同一个节点（或者说计算机）上构建区块链的主要功能，例如挖矿和进行交易。但是，区块链的关键就在于去中心化，也就是说我们需要在不同的节点上部署类似的功能。

这样的话，怎么样确保所有节点都在使用同一个区块链呢？中本聪为此提出了著名的“共识算法”：如果一个节点的区块链与另外一个节点的不同，这就意味着冲突。解决这个问题的办法就是，“最长的有效的链”应当获得认可。换句话说，网络上最长的链就是事实上的标准链。利用这种算法，我们可以在我们网络中的所有节点中达成共识。

要判断“最长的有效的链”，我们可以分为两步。首先，判断某个给定的区块链是否有效：

```
def valid_chain(self, chain):
    last_block = chain[0]
    current_index = 1
    while current_index < len(chain):
        block = chain[current_index]
        print(f'{last_block}')
        print(f'{block}')
        print("\n-----\n")
        # 检查区块的哈希值是否正确
        if block['previous_hash'] != self.hash(last_block):
            return False
        # 检查工作量证明是否正确
        if not self.valid_proof(last_block['proof'],
                                block['proof']):
            return False
        last_block = block
        current_index += 1
    return True
```

下一步是使用消除冲突的“共识算法”。原理是将本节点上区块链和网络上所有节点的区块链做比较。一旦在网络上找到了一个有效的区块链，而且长度比本节点的区块链长，我们就换用这个区块链。具体的实现如下：

```

def resolve_conflicts(self):
    neighbours = self.nodes
    new_chain = None
    # 寻找比当前链长的区块链
    max_length = len(self.chain)
    # 从网络中所有节点获取区块链并加以验证
    for node in neighbours:
        response = requests.get(f'http://{node}/chain')
        # 从节点的网络端口获取区块链信息
        if response.status_code == 200:
            # HTTP 请求返回状态字 200 表示成功
            length = response.json()['length']
            chain = response.json()['chain']
            # 检查区块链的长度和是否有效
            if length > max_length and self.valid_chain(chain):
                max_length = length
                new_chain = chain
    # 一旦找到了一个有效的链，而且长度比本节点的链长，就换用这个区块链
    if new_chain:
        self.chain = new_chain
        return True
    return False

```

“共识算法”的本质，是选择计算量最大的链条最为主链条。这样，即使有人恶意破坏，也要付出大量的经济成本，达到不可承受的程度。

举个例子来说，超市付款需要排成一队，可能有人不守规矩要插队，这样分出好几条队伍。超市管理员会检查队伍，找出最长的一条队伍是合法的，奖励其中拍得久的人，并让不合法的分叉队伍重新排队。只要大部分人不傻，就会自觉在最长的队伍上排队。

至此，我们已经明确了主要的规则和算法，下面我们就可以开始实际交易了。作为一个区块链或者加密货币的普通使用者，可能并不需要关心起其核心算法是什么，但是需要了解交易的过程意味着什么。这就如同一个购买股票的交易者，可能并不需要知道交易所采用了什么样的通信技术，但是要知道自己买卖股票的步骤。在这里笔者将引导您以用户的身份，在自己创建的“虚拟交易所”中进行交易。

交易的过程，实际上就是通过 HTTP 请求，告诉服务器（或者说交易所）我们想要做什么。为此，我们将使用 Python Flask 框架。如前所述，这是一个轻量 Web 应用框架，它所提供的 API 可以把网络请求映射到 Python 函数。这里，我们将创建三个接口（endpoint）：

/transactions/new: 创建一个交易并添加到区块

/mine: 告诉服务器去挖掘新的区块

/chain: 展示当前整个区块链的信息

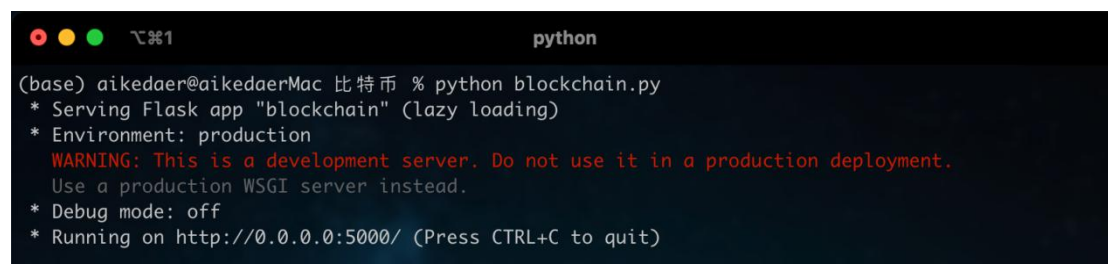
我们的“Flask 服务器”将扮演区块链网络中的一个节点。我们先添加一些框架代码：

```
# 初始化节点
app = Flask(__name__)
# 为该节点生成一个唯一的地址
node_identifier = str(uuid4()).replace('-', '')
# 初始化区块链
blockchain = Blockchain()
@app.route('/mine', methods=['GET'])
# 建立一个用于访问挖矿功能的网址终端，即/mine；请求类型为 GET
def mine():
    return "挖一个新的区块"
@app.route('/transactions/new', methods=['POST'])
# 建立一个用于访问添加交易功能的网址终端，即/transactions/new；请求类型为 POST
def new_transaction():
    return "添加交易到 Y 区块"
@app.route('/chain', methods=['GET'])
# 建立获取当前区块链信息的网址终端，即/chain；请求类型为 GET
def full_chain():
    return "区块链信息"
```

现在，我们就可以直接对我们的交易所发出请求了。

第 7 章 执行交易

本地运行 blockchain.py,

A terminal window titled 'python' showing the execution of 'python blockchain.py'. The output includes: '(base) aiked aer@aiked aerMac 比特币 % python blockchain.py', '* Serving Flask app "blockchain" (lazy loading)', '* Environment: production', 'WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.', '* Debug mode: off', and '* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)'.

```
(base) aiked aer@aiked aerMac 比特币 % python blockchain.py
* Serving Flask app "blockchain" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

并使用 Postman 发出一个 GET 请求，首先查看区块链的初始状态：
<http://localhost:5000/chain>

GET http://localhost:5000/chain Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results 200 OK 17 ms 261 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "chain": [
3     {
4       "index": 1,
5       "previous_hash": "1",
6       "proof": 100,
7       "timestamp": 1627015349.813355,
8       "transactions": []
9     }
10  ],
11  "length": 1
12 }
```

申请挖矿：http://localhost:5000/mine

GET http://localhost:5000/mine Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

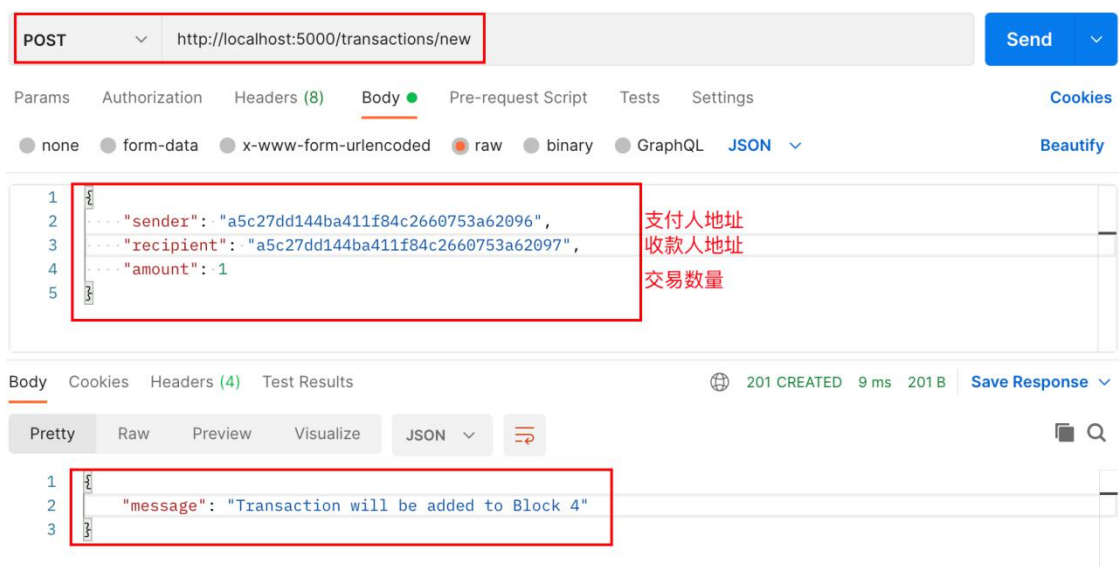
Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

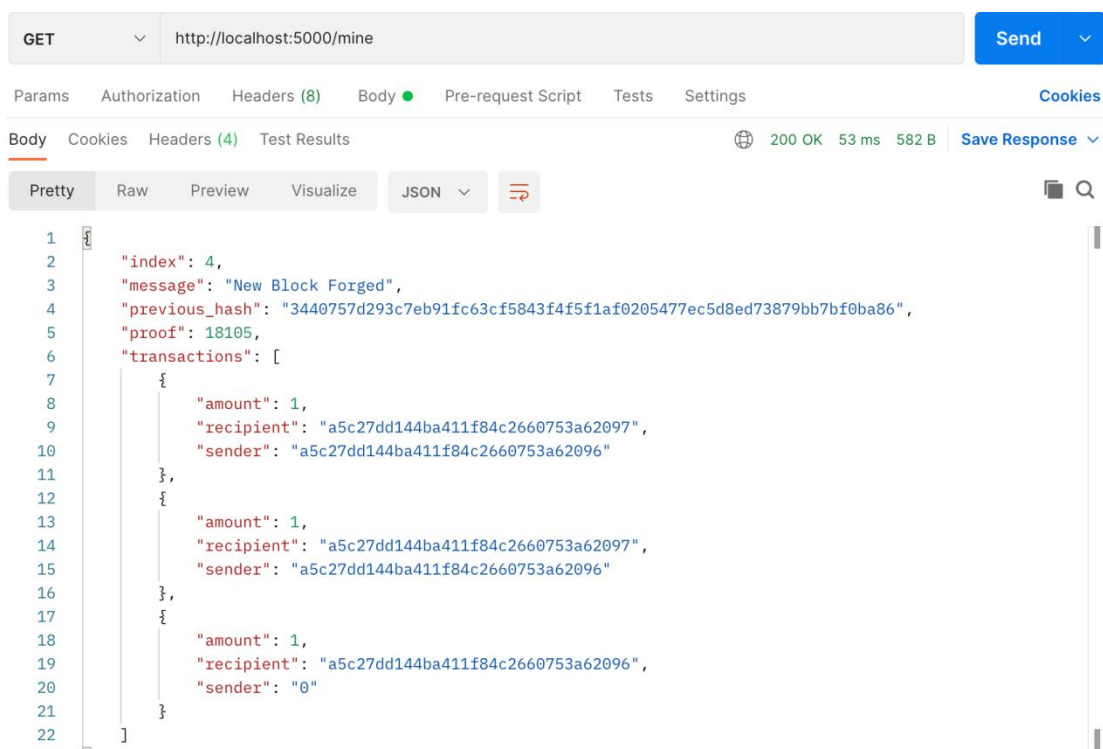
Body Cookies Headers (4) Test Results 200 OK 30 ms 373 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "index": 2,
3   "message": "New Block Forged",
4   "previous_hash": "80ef1bf4e01b633c7337c79f55422ef2c06f7d0f2e6d2e5afa197d1e0b0c08a4",
5   "proof": 6140,
6   "transactions": [
7     {
8       "amount": 1,
9       "recipient": "a5c27dd144ba411f84c2660753a62096",
10      "sender": "0"
11    }
12  ]
13 }
```



继续挖矿，可以看到这个新的块包含了我们的交易信息



部署到阿里云服务器：地址为：47.100.55.201:5000/chain

第 8 章 附录：完整代码

```
import hashlib
import json
```

```

from time import time
from urllib.parse import urlparse
from uuid import uuid4

import requests
from flask import Flask, jsonify, request

class Blockchain:
    def __init__(self):
        self.current_transactions = []
        self.chain = []
        self.nodes = set()

        # Create the genesis block
        self.new_block(previous_hash='1', proof=100)

    def register_node(self, address):
        """
        Add a new node to the list of nodes
        :param address: Address of node. Eg. 'http://192.168.0.5:5000'
        """

        parsed_url = urlparse(address)
        if parsed_url.netloc:
            self.nodes.add(parsed_url.netloc)
        elif parsed_url.path:
            # Accepts an URL without scheme like '192.168.0.5:5000'.
            self.nodes.add(parsed_url.path)
        else:
            raise ValueError('Invalid URL')

    def valid_chain(self, chain):
        """
        Determine if a given blockchain is valid
        :param chain: A blockchain
        :return: True if valid, False if not
        """

        last_block = chain[0]
        current_index = 1

        while current_index < len(chain):

```

```

        block = chain[current_index]
        print(f'{last_block}')
        print(f'{block}')
        print("\n-----\n")
        # Check that the hash of the block is correct
        if block['previous_hash'] != self.hash(last_block):
            return False

        # Check that the Proof of Work is correct
        if not self.valid_proof(last_block['proof'], block['proof'],
last_block['previous_hash']):
            return False

        last_block = block
        current_index += 1

    return True

def resolve_conflicts(self):
    """
    This is our consensus algorithm, it resolves conflicts
    by replacing our chain with the longest one in the network.
    :return: True if our chain was replaced, False if not
    """

    neighbours = self.nodes
    new_chain = None

    # We're only looking for chains longer than ours
    max_length = len(self.chain)

    # Grab and verify the chains from all the nodes in our network
    for node in neighbours:
        response = requests.get(f'http://{node}/chain')

        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']

            # Check if the length is longer and the chain is valid
            if length > max_length and self.valid_chain(chain):
                max_length = length
                new_chain = chain

```



```

# Replace our chain if we discovered a new, valid chain longer than ours
if new_chain:
    self.chain = new_chain
    return True

return False

def new_block(self, proof, previous_hash):
    """
    Create a new Block in the Blockchain
    :param proof: The proof given by the Proof of Work algorithm
    :param previous_hash: Hash of previous Block
    :return: New Block
    """

    block = {
        'index': len(self.chain) + 1,
        'timestamp': time(),
        'transactions': self.current_transactions,
        'proof': proof,
        'previous_hash': previous_hash or self.hash(self.chain[-1]),
    }

    # Reset the current list of transactions
    self.current_transactions = []

    self.chain.append(block)
    return block

def new_transaction(self, sender, recipient, amount):
    """
    Creates a new transaction to go into the next mined Block
    :param sender: Address of the Sender
    :param recipient: Address of the Recipient
    :param amount: Amount
    :return: The index of the Block that will hold this transaction
    """

    self.current_transactions.append({
        'sender': sender,
        'recipient': recipient,
        'amount': amount,
    })

    return self.last_block['index'] + 1

```

```

@property
def last_block(self):
    return self.chain[-1]

@staticmethod
def hash(block):
    """
    Creates a SHA-256 hash of a Block
    :param block: Block
    """

    # We must make sure that the Dictionary is Ordered, or we'll have
inconsistent hashes
    block_string = json.dumps(block, sort_keys=True).encode()
    return hashlib.sha256(block_string).hexdigest()

def proof_of_work(self, last_block):
    """
    Simple Proof of Work Algorithm:
    - Find a number p' such that hash(pp') contains leading 4 zeroes
    - Where p is the previous proof, and p' is the new proof

    :param last_block: <dict> last Block
    :return: <int>
    """

    last_proof = last_block['proof']
    last_hash = self.hash(last_block)

    proof = 0
    while self.valid_proof(last_proof, proof, last_hash) is False:
        proof += 1

    return proof

@staticmethod
def valid_proof(last_proof, proof, last_hash):
    """
    Validates the Proof

    :param last_proof: <int> Previous Proof
    :param proof: <int> Current Proof
    :param last_hash: <str> The hash of the Previous Block
    :return: <bool> True if correct, False if not.

```

```

"""

guess = f'{last_proof}{proof}{last_hash}'.encode()
guess_hash = hashlib.sha256(guess).hexdigest()
return guess_hash[:4] == "0000"

# Instantiate the Node
app = Flask(__name__)

# Generate a globally unique address for this node
node_identifier = str(uuid4()).replace('-', '')

# Instantiate the Blockchain
blockchain = Blockchain()

@app.route('/mine', methods=['GET'])
def mine():
    # We run the proof of work algorithm to get the next proof...
    last_block = blockchain.last_block
    proof = blockchain.proof_of_work(last_block)

    # We must receive a reward for finding the proof.
    # The sender is "0" to signify that this node has mined a new coin.
    blockchain.new_transaction(
        sender="0",
        recipient=node_identifier,
        amount=1,
    )

    # Forge the new Block by adding it to the chain
    previous_hash = blockchain.hash(last_block)
    block = blockchain.new_block(proof, previous_hash)

    response = {
        'message': "New Block Forged",
        'index': block['index'],
        'transactions': block['transactions'],
        'proof': block['proof'],
        'previous_hash': block['previous_hash'],
    }
    return jsonify(response), 200

```

```

@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    values = request.get_json()

    # Check that the required fields are in the POST'ed data
    required = ['sender', 'recipient', 'amount']
    if not all(k in values for k in required):
        return 'Missing values', 400

    # Create a new Transaction
    index = blockchain.new_transaction(values['sender'], values['recipient'],
    values['amount'])

    response = {'message': f'Transaction will be added to Block {index}'}
    return jsonify(response), 201


@app.route('/chain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200


@app.route('/nodes/register', methods=['POST'])
def register_nodes():
    values = request.get_json()

    nodes = values.get('nodes')
    if nodes is None:
        return "Error: Please supply a valid list of nodes", 400

    for node in nodes:
        blockchain.register_node(node)

    response = {
        'message': 'New nodes have been added',
        'total_nodes': list(blockchain.nodes),
    }
    return jsonify(response), 201

```

```

@app.route('/nodes/resolve', methods=['GET'])
def consensus():
    replaced = blockchain.resolve_conflicts()

    if replaced:
        response = {
            'message': 'Our chain was replaced',
            'new_chain': blockchain.chain
        }
    else:
        response = {
            'message': 'Our chain is authoritative',
            'chain': blockchain.chain
        }

    return jsonify(response), 200

if __name__ == '__main__':
    from argparse import ArgumentParser

    parser = ArgumentParser()
    parser.add_argument('-p', '--port', default=5000, type=int, help='port to
listen on')
    args = parser.parse_args()
    port = args.port

    app.run(host='0.0.0.0', port=port)

```