

作者认为最快的学习区块链的方式是自己创建一个，本文就跟随作者用Python来创建一个区块链。

对数字货币的崛起感到新奇的我们，并且想知道其背后的技术——区块链是怎样实现的。

但是完全搞懂区块链并非易事，我喜欢在实践中学习，通过写代码来学习技术会掌握得更牢固。通过构建一个区块链可以加深对区块链的理解。

准备工作

本文要求读者对Python有基本的理解，能读写基本的Python，并且需要对HTTP请求有基本的了解。

我们知道区块链是由区块的记录构成的不可变、有序的链结构，记录可以是交易、文件或任何你想要的数据，重要的是它们是通过哈希值（hashes）链接起来的。

如果你还不是很了解哈希，可以查看这篇文章

环境准备

环境准备，确保已经安装Python3.6+, pip , Flask, requests

安装方法：

|  |
| --- |
| pip install Flask==0.12.2 requests==2.18.4 |

同时还需要一个HTTP客户端，比如Postman，cURL或其它客户端。

参考源代码（原代码在我翻译的时候，无法运行，我fork了一份，修复了其中的错误，并添加了翻译，感谢star）

开始创建Blockchain

新建一个文件 blockchain.py，本文所有的代码都写在这一个文件中，可以随时参考源代码

Blockchain类

首先创建一个Blockchain类，在构造函数中创建了两个列表，一个用于储存区块链，一个用于储存交易。

以下是Blockchain类的框架：

|  |
| --- |
| class Blockchain(object):  def \_\_init\_\_(self):  self.chain = []  self.current\_transactions = []  def new\_block(self):  # Creates a new Block and adds it to the chain  pass  def new\_transaction(self):  # Adds a new transaction to the list of transactions  pass  @staticmethod  def hash(block):  # Hashes a Block  pass  @property  def last\_block(self):  # Returns the last Block in the chain  pass |

Blockchain类用来管理链条，它能存储交易，加入新块等，下面我们来进一步完善这些方法。

块结构

每个区块包含属性：索引（index），Unix时间戳（timestamp），交易列表（transactions），工作量证明（稍后解释）以及前一个区块的Hash值。

以下是一个区块的结构：

|  |
| --- |
| block = {  'index': 1,  'timestamp': 1506057125.900785,  'transactions': [  {  'sender': "8527147fe1f5426f9dd545de4b27ee00",  'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",  'amount': 5,  }  ],  'proof': 324984774000,  'previous\_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e  1b161e5c1fa7425e73043362938b9824"  } |

到这里，区块链的概念就清楚了，每个新的区块都包含上一个区块的Hash，这是关键的一点，它保障了区块链不可变性。如果攻击者破坏了前面的某个区块，那么后面所有区块的Hash都会变得不正确。不理解的话，慢慢消化，可参考区块链记账原理

加入交易

接下来我们需要添加一个交易，来完善下new\_transaction方法

|  |
| --- |
| class Blockchain(object):   ...   def new\_transaction(self, sender, recipient, amount):   """   生成新交易信息，信息将加入到下一个待挖的区块中   :param sender: <str> Address of the Sender   :param recipient: <str> Address of the Recipient   :param amount: <int> Amount   :return: <int> 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 |

生成新交易信息，信息将加入到下一个待挖的区块中

|  |
| --- |
| import hashlib   import json   from time import time   class Blockchain(object):   def \_\_init\_\_(self):   self.current\_transactions = []   self.chain = []   # Create the genesis block   self.new\_block(previous\_hash=1, proof=100)   def new\_block(self, proof, previous\_hash=None):   """   生成新块   :param proof: <int> The proof given by the Proof of Work algorithm   :param previous\_hash: (Optional) <str> Hash of previous Block   :return: <dict> 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):   """   生成新交易信息，信息将加入到下一个待挖的区块中   :param sender: <str> Address of the Sender   :param recipient: <str> Address of the Recipient   :param amount: <int> Amount   :return: <int> 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):   """   生成块的 SHA-256 hash值   :param block: <dict> Block   :return: <str>   """   # 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() |

方法向列表中添加一个交易记录，并返回该记录将被添加到的区块(下一个待挖掘的区块)的索引，等下在用户提交交易时会有用。

创建新块

当Blockchain实例化后，我们需要构造一个创世块（没有前区块的第一个区块），并且给它加上一个工作量证明。

每个区块都需要经过工作量证明，俗称挖矿，稍后会继续讲解。

为了构造创世块，我们还需要完善newblock(), newtransaction() 和hash() 方法：

|  |
| --- |
| from hashlib import sha256   x = 5   y = 0 # y未知   while sha256(f'{x\*y}'.encode()).hexdigest()[-1] != "0":   y += 1   print(f'The solution is y = {y}') |

通过上面的代码和注释可以对区块链有直观的了解，接下来我们看看区块是怎么挖出来的。

理解工作量证明

新的区块依赖工作量证明算法（PoW）来构造。PoW的目标是找出一个符合特定条件的数字，这个数字很难计算出来，但容易验证。这就是工作量证明的核心思想。

为了方便理解，举个例子：

假设一个整数 x 乘以另一个整数 y 的积的 Hash 值必须以 0 结尾，即 hash(x \* y) = ac23dc…0。设变量 x = 5，求 y 的值？

用Python实现如下：

|  |
| --- |
| from hashlib import sha256   x = 5   y = 0 # y未知   while sha256(f'{x\*y}'.encode()).hexdigest()[-1] != "0":   y += 1   print(f'The solution is y = {y}') |

结果是y=21. 因为：

|  |
| --- |
| hash(5 \* 21) = 1253e9373e...5e3600155e860 |

在比特币中，使用称为Hashcash的工作量证明算法，它和上面的问题很类似。矿工们为了争夺创建区块的权利而争相计算结果。通常，计算难度与目标字符串需要满足的特定字符的数量成正比，矿工算出结果后，会获得比特币奖励。

当然，在网络上非常容易验证这个结果。

实现工作量证明

让我们来实现一个相似PoW算法，规则是：寻找一个数 p，使得它与前一个区块的 proof 拼接成的字符串的 Hash 值以 4 个零开头。

|  |
| --- |
| import hashlib   import json   from time import time   from uuid import uuid4   class Blockchain(object):   ...   def proof\_of\_work(self, last\_proof):   """   简单的工作量证明:   - 查找一个 p' 使得 hash(pp') 以4个0开头   - p 是上一个块的证明, p' 是当前的证明   :param last\_proof: <int>   :return: <int>   """   proof = 0   while self.valid\_proof(last\_proof, proof) is False:   proof += 1   return proof   @staticmethod   def valid\_proof(last\_proof, proof):   """   验证证明: 是否hash(last\_proof, proof)以4个0开头?   :param last\_proof: <int> Previous Proof   :param proof: <int> Current Proof   :return: <bool> True if correct, False if not.   """   guess = f'{last\_proof}{proof}'.encode()   guess\_hash = hashlib.sha256(guess).hexdigest()   return guess\_hash[:4] == "0000" |

衡量算法复杂度的办法是修改零开头的个数。使用4个来用于演示，你会发现多一个零都会大大增加计算出结果所需的时间。

现在Blockchain类基本已经完成了，接下来使用HTTP requests来进行交互。

Blockchain作为API接口

我们将使用Python Flask框架，这是一个轻量Web应用框架，它方便将网络请求映射到 Python函数，现在我们来让Blockchain运行在基于Flask web上。

我们将创建三个接口：

1./transactions/new 创建一个交易并添加到区块

2./mine 告诉服务器去挖掘新的区块

3./chain 返回整个区块链

创建节点

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：

|  |
| --- |
| import hashlib   import json   from textwrap import dedent   from time import time   from uuid import uuid4   from flask import Flask   class Blockchain(object):   ...   # Instantiate our 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():   return "We'll mine a new Block"   @app.route('/transactions/new', methods=['POST'])   def new\_transaction():   return "We'll add a new transaction"   @app.route('/chain', methods=['GET'])   def full\_chain():   response = {   'chain': blockchain.chain,   'length': len(blockchain.chain),   }   return jsonify(response), 200   if \_\_name\_\_ == '\_\_main\_\_':   app.run(host='0.0.0.0', port=5000) |

简单的说明一下以上代码：

第15行: 创建一个节点.

第18行: 为节点创建一个随机的名字.

第21行: 实例Blockchain类.

第24–26行: 创建/mine GET接口。

第28–30行: 创建/transactions/new POST接口,可以给接口发送交易数据.

第32–38行: 创建 /chain 接口, 返回整个区块链。

第40–41行: 服务运行在端口5000上.

发送交易

发送到节点的交易数据结构如下：

|  |
| --- |
| {   "sender": "my address",   "recipient": "someone else's address",   "amount": 5   } |

之前已经有添加交易的方法，基于接口来添加交易就很简单了

|  |
| --- |
| import hashlib   import json   from textwrap import dedent   from time import time   from uuid import uuid4   from flask import Flask, jsonify, request   ...   @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 |

挖矿

挖矿正是神奇所在，它很简单，做了一下三件事：

1.计算工作量证明PoW

2.通过新增一个交易授予矿工（自己）一个币

3.构造新区块并将其添加到链中

|  |
| --- |
| import hashlib   import json   from time import time   from uuid import uuid4   from flask import Flask, jsonify, request   ...   @app.route('/mine', methods=['GET'])   def mine():   # We run the proof of work algorithm to get the next proof...   last\_block = blockchain.last\_block   last\_proof = last\_block['proof']   proof = blockchain.proof\_of\_work(last\_proof)   # 给工作量证明的节点提供奖励.   # 发送者为 "0" 表明是新挖出的币   blockchain.new\_transaction(   sender="0",   recipient=node\_identifier,   amount=1,   )   # Forge the new Block by adding it to the chain   block = blockchain.new\_block(proof)   response = {   'message': "New Block Forged",   'index': block['index'],   'transactions': block['transactions'],   'proof': block['proof'],   'previous\_hash': block['previous\_hash'],   }   return jsonify(response), 200 |

注意交易的接收者是我们自己的服务器节点，我们做的大部分工作都只是围绕Blockchain类方法进行交互。到此，我们的区块链就算完成了，我们来实际运行下

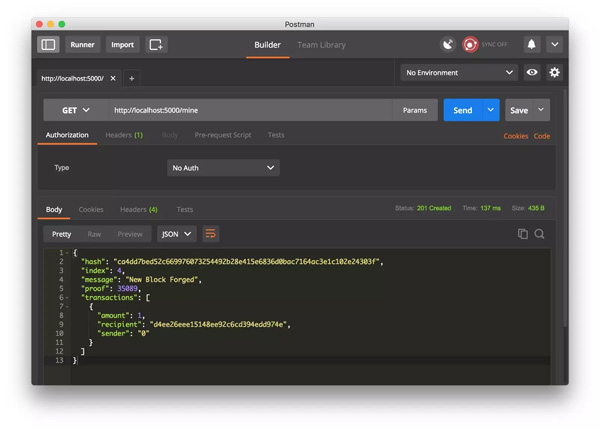
运行区块链

你可以使用cURL 或Postman 去和API进行交互

启动server:

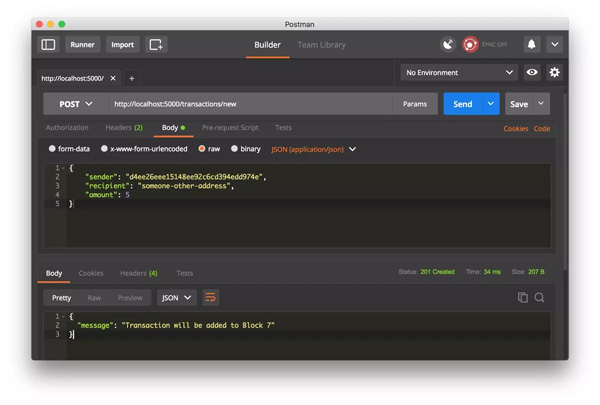
|  |
| --- |
| $ python blockchain.py   \* Runing on http://127.0.0.1:5000/ (Press CTRL+C to quit) |

让我们通过请求 http://localhost:5000/mine 来进行挖矿



用Postman请求挖矿

通过post请求，添加一个新交易



用Postman请求挖矿

如果不是使用Postman，则用一下的cURL语句也是一样的：

|  |
| --- |
| $ curl -X POST -H "Content-Type: application/json" -d '{   "sender": "d4ee26eee15148ee92c6cd394edd974e",   "recipient": "someone-other-address",   "amount": 5   }' "http://localhost:5000/transactions/new" |

在挖了两次矿之后，就有3个块了，通过请求 http://localhost:5000/chain 可以得到所有的块信息。

|  |
| --- |
| {   "chain": [   {   "index": 1,   "previous\_hash": 1,   "proof": 100,   "timestamp": 1506280650.770839,   "transactions": []   },   {   "index": 2,   "previous\_hash": "c099bc...bfb7",   "proof": 35293,   "timestamp": 1506280664.717925,   "transactions": [   {   "amount": 1,   "recipient": "8bbcb347e0634905b0cac7955bae152b",   "sender": "0"   }   ]   },   {   "index": 3,   "previous\_hash": "eff91a...10f2",   "proof": 35089,   "timestamp": 1506280666.1086972,   "transactions": [   {   "amount": 1,   "recipient": "8bbcb347e0634905b0cac7955bae152b",   "sender": "0"   }   ]   }   ],   "length": 3   } |

一致性（共识）

我们已经有了一个基本的区块链可以接受交易和挖矿。但是区块链系统应该是分布式的。既然是分布式的，那么我们究竟拿什么保证所有节点有同样的链呢？这就是一致性问题，我们要想在网络上有多个节点，就必须实现一个一致性的算法。

注册节点

在实现一致性算法之前，我们需要找到一种方式让一个节点知道它相邻的节点。每个节点都需要保存一份包含网络中其它节点的记录。因此让我们新增几个接口：

/nodes/register 接收URL形式的新节点列表

/nodes/resolve执行一致性算法，解决任何冲突，确保节点拥有正确的链

我们修改下Blockchain的ini

|  |
| --- |
| ...   from urllib.parse import urlparse   ...   class Blockchain(object):   def \_\_init\_\_(self):   ...   self.nodes = set()   ...   def register\_node(self, address):   """   Add a new node to the list of nodes   :param address: <str> Address of node. Eg. 'http://192.168.0.5:5000'   :return: None   """   parsed\_url = urlparse(address)   self.nodes.add(parsed\_url.netloc) |

t函数并提供一个注册节点方法：

我们用 set 来储存节点，这是一种避免重复添加节点的简单方法。

实现共识算法

前面提到，冲突是指不同的节点拥有不同的链，为了解决这个问题，规定最长的、有效的链才是最终的链，换句话说，网络中有效最长链才是实际的链。

我们使用一下的算法，来达到网络中的共识

|  |
| --- |
| ...   import requests   class Blockchain(object)   ...   def valid\_chain(self, chain):   """   Determine if a given blockchain is valid   :param chain: <list> A blockchain   :return: <bool> 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']):   return False   last\_block = block   current\_index += 1   return True   def resolve\_conflicts(self):   """   共识算法解决冲突   使用网络中最长的链.   :return: <bool> True 如果链被取代, 否则为False   """   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 |

第一个方法 valid\_chain() 用来检查是否是有效链，遍历每个块验证hash和proof.

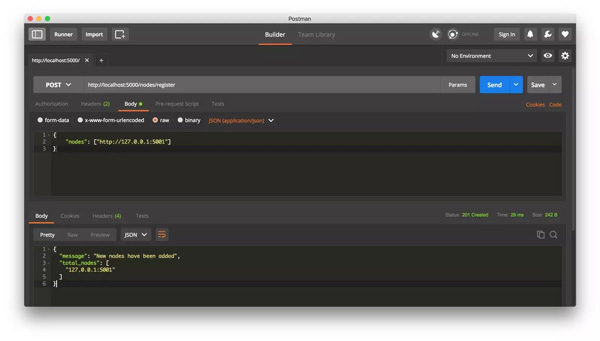
第2个方法 resolve\_conflicts() 用来解决冲突，遍历所有的邻居节点，并用上一个方法检查链的有效性， 如果发现有效更长链，就替换掉自己的链

让我们添加两个路由，一个用来注册节点，一个用来解决冲突。

|  |
| --- |
| @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 |

你可以在不同的机器运行节点，或在一台机机开启不同的网络端口来模拟多节点的网络，这里在同一台机器开启不同的端口演示，在不同的终端运行一下命令，就启动了两个节点：http://localhost:5000 和 http://localhost:5001

|  |
| --- |
| pipenv run python blockchain.py   pipenv run python blockchain.py -p 5001 |



注册新节点

然后在节点2上挖两个块，确保是更长的链，然后在节点1上访问接口/nodes/resolve ,这时节点1的链会通过共识算法被节点2的链取代。

