# 区块链实验

## 首先创建实验环境

> 这里默认大家的Python环境已经安装好了

实验环境：Python 3.7 实验目的：复习上节课的区块链原理，动手实现Python语言下的区块链编写

- 打开 CMD 或者 PS (Bash|Terminal) 运行 `pip install virtualenv` 安装 Python 的环境管理器
- 创建一个文件夹目录 => 建议取名 BlockChain 并进入该目录
- 通过 CMD 或者 PS (Bash|Terminal) 进入之前创建的目录之后，运行 `virtualenv --no-site-packages --no-download -p python3 venv` 创建一个 Python3 的 venv 环境，作为本次动手实验的环境
- 创建下述文件
    - src 文件价 ---> 用来存放代码文件
    - README.md 文件 ---> 用来编写说明文档

## 区块链实现步骤

- 激活刚才已经创建的 Python3 的虚拟环境 `source venv/bin/activate`
- 安装所需的依赖包 `pip install Flask requests`
- 在 src 文件夹中创建 block_chain.py 文件，录入下述代码：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# ================================================
# 本项目由@Ryuchen开发维护，使用Python3.7
# ================================================

import json
```

```
import hashlib
import requests

from time import time
from uuid import uuid4
from flask import Flask
from flask import jsonify
from flask import request
from textwrap import dedent
from urllib.parse import urlparse


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
```

- 然后我们需要定义我们区块链中的每一个区块的内容格式，这里我们设计如下：

```
# This is a Block
{
    'index': 1,
    'timestamp': 1506057125.900785,
    'transactions': [
        {
```

```
            'sender': "8527147fe1f5426f9dd545de4b27ee00",
            'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",
            'amount': 5,
            }
        ],
        'proof': 324984774000,
        'previous_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
    }
```

- 然后我们实现上面定义的区块链框架中的添加交易区块的方法，补充 `new_transaction` 方法如下：

```
def new_transaction(self, sender, recipient, amount):
    """
    创建一笔新的交易到下一个被挖掘的区块中
    :param sender: <str> 发送人的地址
    :param recipient: <str> 接收人的地址
    :param amount: <int> 金额
    :return: <int> 持有本次交易的区块索引
    """
    self.current_transactions.append({
        'sender': sender,
        'recipient': recipient,
        'amount': amount,
    })
    return self.last_block['index'] + 1
```

- 然后我们实现创建区块的方法，这里需要注意当我们的 `BlockChain` 被实例化后，我们需要将 创世区块（一个没有前导区块的区块）添加进去进去，需要补充的代码如下：

```
class BlockChain(object):

    def __init__(self):
        self.current_transactions = []
        self.chain = []
        # 创建创世区块
        self.new_block(previous_hash=1, proof=100)

    def new_block(self, proof, previous_hash=None):
        """
        创建一个新的区块到区块链中
```

```
        :param proof: <int> 由工作证明算法生成的证明
        :param previous_hash: (Optional) <str> 前一个区块的 hash 值
        :return: <dict> 新区块
        """
        block = {
            'index': len(self.chain) + 1,
            'timestamp': time(),
            'transactions': self.current_transactions,
            'proof': proof,
            'previous_hash': previous_hash or self.hash(self.chain[-
1]),
        }
        # 重置当前交易记录
        self.current_transactions = []
        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
        创建一笔新的交易到下一个被挖掘的区块中
        :param sender: <str> 发送人的地址
        :param recipient: <str> 接收人的地址
        :param amount: <int> 金额
        :return: <int> 持有本次交易的区块索引
        """
        self.current_transactions.append({
            'sender': sender,
            'recipient': recipient,
            'amount': amount,
        })
        return self.last_block['index'] + 1

    @staticmethod
    def hash(block):
        """
        给一个区块生成 SHA-256 值
        :param block: <dict> Block
        :return: <str>
        """
        # 我们必须确保这个字典（区块）是经过排序的，否则我们将会得到不一致的散列
        block_string = json.dumps(block, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

    @property
    def last_block(self):
```

```
        return self.chain[-1]
```

- 然后我们要让我们的区块的Hash是有意义的，所以我们需要定义我们的工作量证明，这里我们就使用最基础的工作量证明算法来实现这个机制（POW），所以我们补充我们的代码如下：

```
class BlockChain(object):

    def __init__(self):
        self.current_transactions = []
        self.chain = []
        # 创建创世区块
        self.new_block(previous_hash=1, proof=100)

    def new_block(self, proof, previous_hash=None):
        """
        创建一个新的区块到区块链中
        :param proof: <int> 由工作证明算法生成的证明
        :param previous_hash: (Optional) <str> 前一个区块的 hash 值
        :return: <dict> 新区块
        """
        block = {
            'index': len(self.chain) + 1,
            'timestamp': time(),
            'transactions': self.current_transactions,
            'proof': proof,
            'previous_hash': previous_hash or self.hash(self.chain[-
1]),
        }
        # 重置当前交易记录
        self.current_transactions = []
        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
        创建一笔新的交易到下一个被挖掘的区块中
        :param sender: <str> 发送人的地址
        :param recipient: <str> 接收人的地址
        :param amount: <int> 金额
        :return: <int> 持有本次交易的区块索引
        """
        self.current_transactions.append({
            'sender': sender,
```

```
            'recipient': recipient,
            'amount': amount,
        })
        return self.last_block['index'] + 1

    @staticmethod
    def hash(block):
        """
        给一个区块生成 SHA-256 值
        :param block: <dict> Block
        :return: <str>
        """
        # 我们必须确保这个字典（区块）是经过排序的，否则我们将会得到不一致的散列
        block_string = json.dumps(block, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

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

    # 找到一个数字 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
    @staticmethod
    def valid_proof(last_proof, proof):
        """
        Validates the Proof: Does hash(last_proof, proof) contain 4 leadi
        :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"

    def proof_of_work(self, last_proof):
        """
        Simple Proof of Work Algorithm:
        - Find a number p' such that hash(pp') contains leading 4 zeroes,
        - p is the previous proof, and p' is the new proof
        :param last_proof: <int>
        :return: <int>
        """
        proof = 0
        while self.valid_proof(last_proof, proof) is False:
            proof += 1
```

```
        return proof
```

# 构建联网的区块链节点

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： + /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 返回整个区块链

- 我们先按照上面的设计实现我们的基本框架，在刚才的脚本补充代码如下：

```
class BlockChain(object):
    ...

# Instantiate our Node (实例化我们的节点)
app = Flask(__name__)
# Generate a globally unique address for this node (为这个节点生成一个全球唯一的地址)
node_identifier = str(uuid4()).replace('-', '')
# Instantiate the Blockchain (实例化 Blockchain类)
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)
```

- 然后设计我们的交易方法，因为我们已经有了添加交易的方法，所以基于接口来添加交易就很简单了。让我们为添加事务写函数：

```
##########################################
# 我们之前定义的交易的数据结构如下:
#
# {
#     "sender": "my address",
#     "recipient": "someone else's address",
#     "amount": 5
# }
##########################################

class BlockChain(object):
    ...

# Instantiate our Node (实例化我们的节点)
app = Flask(__name__)
# Generate a globally unique address for this node (为这个节点生成一个全球唯一的地址)
node_identifier = str(uuid4()).replace('-', '')
# Instantiate the Blockchain (实例化 Blockchain类)
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():
    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': '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

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

- 然后我们实现计算Hash的操作，这个动作实际上就是我们称为挖矿，它的机制其实很简单，只是做了以下三件事：
  - 计算工作量证明 PoW
  - 通过新增一个交易授予矿工（自己）一个币
  - 构造新区块并将其添加到链中

```
############################################
# 我们之前定义的交易的数据结构如下：
#
# {
#     "sender": "my address",
#     "recipient": "someone else's address",
#     "amount": 5
# }
############################################

class BlockChain(object):
    ...

# Instantiate our Node (实例化我们的节点)
app = Flask(__name__)
# Generate a globally unique address for this node (为这个节点生成一个全球唯一的地址)
node_identifier = str(uuid4()).replace('-', '')
# Instantiate the Blockchain (实例化 Blockchain类)
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
    last_proof = last_block['proof']
    proof = blockchain.proof_of_work(last_proof)
    # We must receive a reward for finding the proof.
```

```
    # The sender is "0" to signify that this node has mined a new co
in.
    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['rec
ipient'], values['amount'])
    response = {'message': 'Transaction will be added to Block {inde
x}'}
    return jsonify(response), 201

@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)
```

# 连接区块链进行实际操作

你可以使用 cURL 或 Postman 去和 API 进行交互

- 启动 Server：

```
>$ python blockchain.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

- 让我们通过请求 `http://localhost:5000/mine` （GET）来进行挖矿：

```
>$ curl -X GET -H "Content-Type: application/json" http://localhost:
5000/mine
```

- 创建一个交易请求，请求 `http://localhost:5000/transactions/new` （POST）：

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

- 通过请求 `http://localhost:5000/chain` （GET）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
}
```


# 区块链系统应该是分布式的，解决共识问题

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 init 函数并提供一个注册节点方法：

```
class BlockChain(object):

    def __init__(self):
        self.current_transactions = []
        self.chain = []
        # 创建创世区块
        self.new_block(previous_hash=1, proof=100)
        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://192.168.0.5:50
        :return: None
        """
        parsed_url = urlparse(address)
        self.nodes.add(parsed_url.netloc)
```

- 就像先前讲的那样，当一个节点与另一个节点有不同的链时，就会产生冲突。为了解决这个问题，我们将制定最长的有效链条是最权威的规则。换句话说就是：在这个网络里最长的链就是最权威的。我们将使用这个算法，在网络中的节点之间达成共识。我们添加代码如下：

```
class BlockChain(object):
    ...
    # 第一个方法 valid_chain() 负责检查一个链是否有效，方法是遍历每个块并验证散列和证明。
    # 第二个方法 resolve_conflicts() 是一个遍历我们所有邻居节点的方法，下载它们的链并使用上面的方法验证它们。如果找到一个长度大于我们的有效链条，我们就取代我们的链条。

    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):
        """
        This is our Consensus Algorithm, it resolves conflicts
        by replacing our chain with the longest one in the network.
        :return: <bool> 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 netwo
rk
        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 val
id
                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
```

- 然后我们需要增加注册节点的方法和检验权威的方法接口，让更多的人来加入到我们的区块中：

```
class BlockChain(object):
    ...

@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
```

# 完整代码附录

到此我们的全部区块链的动手实验就结束了，完整代码如下：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# ==============================================
# 本项目由@Ryuchen开发维护，使用Python3.7
# ==============================================

import json
import hashlib
import requests

from time import time
from uuid import uuid4
from flask import Flask
from flask import jsonify
from flask import request
```

```
from textwrap import dedent
from urllib.parse import urlparse


class BlockChain(object):

    def __init__(self):
        self.current_transactions = []
        self.chain = []
        # 创建创世区块
        self.new_block(previous_hash=1, proof=100)
        # 加入区块链的节点
        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://192.168.0.5:50
        :return: None
        """
        parsed_url = urlparse(address)
        self.nodes.add(parsed_url.netloc)

    def new_block(self, proof, previous_hash=None):
        """
        创建一个新的区块到区块链中
        :param proof: <int> 由工作证明算法生成的证明
        :param previous_hash: (Optional) <str> 前一个区块的 hash 值
        :return: <dict> 新区块
        """
        block = {
            'index': len(self.chain) + 1,
            'timestamp': time(),
            'transactions': self.current_transactions,
            'proof': proof,
            'previous_hash': previous_hash or self.hash(self.chain[-
1]),
        }
        # 重置当前交易记录
        self.current_transactions = []
        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
```

```
        创建一笔新的交易到下一个被挖掘的区块中
        :param sender: <str> 发送人的地址
        :param recipient: <str> 接收人的地址
        :param amount: <int> 金额
        :return: <int> 持有本次交易的区块索引
        """
        self.current_transactions.append({
            'sender': sender,
            'recipient': recipient,
            'amount': amount,
        })
        return self.last_block['index'] + 1

    @staticmethod
    def hash(block):
        """
        给一个区块生成 SHA-256 值
        :param block: <dict> Block
        :return: <str>
        """
        # 我们必须确保这个字典（区块）是经过排序的，否则我们将会得到不一致的散列
        block_string = json.dumps(block, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

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

    # 找到一个数字 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
    @staticmethod
    def valid_proof(last_proof, proof):
        """
        Validates the Proof: Does hash(last_proof, proof) contain 4 leadi
        :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"

    def proof_of_work(self, last_proof):
        """
        Simple Proof of Work Algorithm:
```

```
        - Find a number p' such that hash(pp') contains leading 4 zeroes,
        - p is the previous proof, and p' is the new proof
        :param last_proof: <int>
        :return: <int>
        """
        proof = 0
        while self.valid_proof(last_proof, proof) is False:
            proof += 1
        return proof

    # 第一个方法 valid_chain() 负责检查一个链是否有效，方法是遍历每个块并验证散列和证明。
    # 第二个方法 resolve_conflicts() 是一个遍历我们所有邻居节点的方法，下载它们的链并使用上面的方法验证它们。如果找到一个长度大于我们的有效链条，我们就取代我们的链条。

    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['proo
f']):
                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: <bool> 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 netwo
rk
        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 val
id
                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

# Instantiate our Node (实例化我们的节点)
app = Flask(__name__)
# Generate a globally unique address for this node (为这个节点生成一个全
球唯一的地址)
node_identifier = str(uuid4()).replace('-', '')
# Instantiate the Blockchain (实例化 Blockchain类)
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
    last_proof = last_block['proof']
    proof = blockchain.proof_of_work(last_proof)
    # We must receive a reward for finding the proof.
    # The sender is "0" to signify that this node has mined a new co
in.
```

```
    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['rec
ipient'], values['amount'])
    response = {'message': 'Transaction will be added to Block {inde
x}'}
    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__':
    app.run(host='0.0.0.0', port=5000)
```