



@hidehiro98 (/hidehiro98) 2018年02月09日に更新
(/hidehiro98)

...

ブロックチェーンを作ること学ば 〜ブロックチェーンがどのように動 いているのか学ば最速の方法は作って みることだ〜

Python(/tags/Python) python3(/tags/python3) Bitcoin(/tags/Bitcoin) Blockchain(/tags/Blockchain)

👍 1770



この記事について

この記事はDaniel van Flymen (<https://medium.com/@vanflymen>)さんのLearn Blockchains by Building One - The fastest way to learn how Blockchains work is to build one (<https://hackernoon.com/learn-blockchains-by-building-one-117428612f46>)を本人の許可を得て翻訳したものです。

このブロックチェーンのリポジトリ (<https://github.com/dvf/blockchain>)ではPython以外での言語の実装者の募集も行われているので、興味がある方は是非。

また、この翻訳で出てくる日本語版のリポジトリはこちら

(https://github.com/hidehiro98/blockchain_by_dvf)にあるので参考にしてみてください。

はじめに

あなたがここにいるのは、私と同じように、暗号通貨の盛り上がりに対して心構えが出来ているからだ。そしてあなたはブロックチェーンがどのように動いているのか - その裏にある基本的なテクノロジー-を理解したいと思っている。

しかしブロックチェーンを理解するのは簡単ではない、少なくとも私にとっては簡単ではなかった。私は多くのビデオを見て、抜け漏れの多いチュートリアルをこなし、少なすぎる事例から来るフラストレーションに苦しんでいた。

私は手を動かして学ぶことが好きだ。手を動かすことは私を、抽象的なことをコードのレベルで扱うことを強制し、コードのレベルで扱うことにより抽象的なことが頭に定着する。もしあなたが同じことをすれば、このガイドが終わる頃には動いているブロックチェーンが、確かな理解とともに手に入るだろう。

始める前に

ブロックチェーンとは、不変の、連続したブロックと呼ばれる記録のチェーンであることを覚えておいてほしい。ブロックは取引、ファイルやあらゆるデータを格納することが出来る。しかし重要なことは、ハッシュを使って繋がっているということだ。

もしハッシュが何かについて自信がない場合は、ここ (<https://learncryptography.com/hash-functions/what-are-hash-functions>)に例がある。

この記事の対象者は？ 以下が必要な要素だ。基本的なPythonの読み書きが出来ること、HTTPリクエストがどのように働くかについての理解していること（私達のブロックチェーンはHTTPを介して動くためだ）。

何が必要ですか？ Python 3.6以上と pip がインストールされていることが必要だ。また、Flaskと素晴らしいRequest libraryもインストールする必要がある。

```
pip install Flask==0.12.2 requests==2.18.4
```

そうだった、PostmanやcURLのようなHTTPクライアントも必要だ。まあ動けばなんでも大丈夫だ。

最終的なコードはどこ？ ソースコードはここ (<https://github.com/dvf/blockchain>)にある。

ステップ1: ブロックチェーンを作る

あなたの好きなエディタかIDEを開いてほしい、個人的にはPyCharm (<https://www.jetbrains.com/pycharm/>)が好きだ。そして `blockchain.py` という新しいファイルを作る。私たちは一つのファイルしか使わないが、もしどこにいるのかわからなくなったら、いつでもソースコード (<https://github.com/dvf/blockchain>)を参照することが出来る。

ブロックチェーンを書く

私たちがつくる `Blockchain` クラスのコンストラクタが、ブロックチェーンを納めるための最初の空のリストと、トランザクションを納めるための空のリストを作る。これが私たちのクラス的设计図だ。

```
blockchain.py
```

```
# coding: UTF-8

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

    def new_block(self):
        # 新しいブロックを作り、チェーンに加える
        pass

    def new_transaction(self):
        # 新しいトランザクションをリストに加える
        pass

    @staticmethod
    def hash(block):
        # ブロックをハッシュ化する
        pass

    @property
    def last_block(self):
        # チェーンの最後のブロックをリターンする
        pass
```

私たちの Blockchain クラスはチェーンの取り扱いを司っている。チェーンはトランザクションを収納し、新しいブロックをチェーンに加えるためのヘルパーメソッドを持っている。早速いくつかのメソッドを肉付けしていこう。

ブロックとはどのようなものなのか

それぞれのブロックは、インデックス、タイムスタンプ（UNIXタイム）、トランザクションのリスト、ブルーフ（詳細は後ほど）そしてそれまでの全てのブロックから生成されるハッシュを持っている。

これが一つのブロックの例だ。

```
block = {
    'index': 1,
    'timestamp': 1506057125.900785,
    'transactions': [
        {
            'sender': "8527147fe1f5426f9dd545de4b27ee00",
            'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",
            'amount': 5,
        }
    ],
    'proof': 324984774000,
    'previous_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
```

この時点で、チェーンのアイデアは明確だ - 全ての新しいブロックはそれまでの全てのブロックから生成されるハッシュを自分自身の中に含んでいる。**これこそがまさにブロックチェーンに不変性を与えているものであり、そのために重要なポイントだ**。もしアタッカーがチェーン初期のブロックを破壊した場合、それに続く**全ての**ブロックが不正なハッシュを含むことになる。

この意味がわかるだろうか？もし分からなければ、少し考える時間を取ってほしい - これはブロックチェーンのコアとなるアイデアだ。

トランザクションをブロックに加える

私たちにはトランザクションをブロックに加える方法が必要だ。 `new_transaction()` メソッドがそれを司っており、非常に簡単だ。

blockchain.py

```
# coding: UTF-8
```

```
class Blockchain(object):
    ...

    def new_transaction(self, sender, recipient, amount):
        """
        次に採掘されるブロックに加える新しいトランザクションを作る
        :param sender: <str> 送信者のアドレス
        :param recipient: <str> 受信者のアドレス
        :param amount: <int> 量
        :return: <int> このトランザクションを含むブロックのアドレス
        """

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

        return self.last_block['index'] + 1
```

`new_transaction()` メソッドは、新しいトランザクションをリストに加えた後、そのトランザクションが加えられるブロック-次に採掘されるブロックだ-のインデックスをリターンする。

新しいブロックを作る

我々の `Blockchain` がインスタンス化されるとき、私たちはジェネシスブロック -先祖を持たないブロック- とともにシードする必要がある。それと同時に、ジェネシスブロックにブルーフ-マイニング（またはブルーフ・オブ・ワーク）の結果- も加える必要がある。マイニングについては後で取り上げる。

ジェネシスブロックを加えるのと同時に、 `new_block()` メソッド、 `last_block()` メソッドと `hash()` メソッドも作成しよう。

blockchain.py

```

# coding: UTF-8

import hashlib
import json
from time import time

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

        # ジェネシスブロックを作る
        self.new_block(previous_hash=1, proof=100)

    def new_block(self, proof, previous_hash=None):
        """
        ブロックチェーンに新しいブロックを作る
        :param proof: <int> ブルーフ・オブ・ワークアルゴリズムから得られるブルーフ
        :param previous_hash: (オプション) <str> 前のブロックのハッシュ
        :return: <dict> 新しいブロック
        """

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

        # 現在のトランザクションリストをリセット
        self.current_transactions = []

        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
        次に採掘されるブロックに加える新しいトランザクションを作る
        :param sender: <str> 送信者のアドレス
        :param recipient: <str> 受信者のアドレス
        :param amount: <int> 量
        :return: <int> このトランザクションを含むブロックのアドレス
        """

        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):
        """
        ブロックの SHA-256 ハッシュを作る
        :param block: <dict> ブロック
        :return: <str>
        """

        # 必ずディクショナリ (辞書型のオブジェクト) がソートされている必要がある。そうでないと、一貫性のな

```

```
block_string = json.dumps(block, sort_keys=True).encode()
return hashlib.sha256(block_string).hexdigest()
```

ここで追加したものは非常に簡単はずだ。何をしたかをクリアにしておくために、いくつかのコメントと`docstrings`を加えておいた。我々のブロックチェーンはもうすぐ完成だ。だがしかし、ここで新しいブロックがどのように出来るのかを考える必要がある - 鑄造 (forged) か 採掘 (mined) か -

プルーフ・オブ・ワークを理解する

プルーフ・オブ・ワークアルゴリズム (PoW) とは、ブロックチェーン上でどのように新しいブロックが作られるか、または採掘されるかということを表している。PoWのゴールは、問題を解く番号を発見することだ。その番号はネットワーク上の誰からも**見つけるのは難しく、確認するのは簡単** - コンピュータ的に言えば - なものでなければならない。これがプルーフ・オブ・ワークのコアとなるアイデアだ。

理解するために簡単な例を見てみよう。

ある整数 x かけるある整数 y の`hash`が0で終わらないといけないとしよう。というわけで、
`hash(x * y) = ac23d...0` というようになる。そしてこの簡単な例では、 $x = 5$ と固定しよう。Pythonで実装するとこうなる。

```
from hashlib import sha256

x = 5
y = 0 # まだこのyがどの数字であるべきかはわからない

while sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
    y += 1

print(f'The solution is y = {y}')
```

解は $y = 21$ 。よってそれにより作られたハッシュは 0 で終わる。

```
>>> sha256(f'{5*21}'.encode()).hexdigest()
'1253e9373e781b7500266caa55150e08e210bc8cd8cc70d89985e3600155e860'
```

ビットコインでは、プルーフ・オブ・ワークのアルゴリズムはハッシュキャッシュ (Hashcash) (<https://en.wikipedia.org/wiki/Hashcash>) と呼ばれている。そしてそれはこの基本的な例とそこまで違うものではない。ハッシュキャッシュは、採掘者が競い合って新しいブロックを作るために問題を解く、というものだ。一般的に、難易度は探す文字の数によって決まる。採掘者はその解に対して、報酬としてトランザクションの中でコインを受け取る。

ネットワークは簡単に採掘者の解が正しいかを確認することが出来る。

基本的なプルーフ・オブ・ワークを実装する

私たちのブロックチェーンのために似たアルゴリズムを実装しよう。ルールは上の例と似ている。

前のブロックの解とともにハッシュを作ったときに、最初に4つの 0 が出てくるような番号 p を探そう。

```
blockchain.py

# coding: UTF-8

import hashlib
import json

from time import time
from uuid import uuid4

class Blockchain(object):
    ...

    def proof_of_work(self, last_proof):
        """
        シンプルなプルーフ・オブ・ワークのアルゴリズム:
        - hash(pp') の最初の4つが0となるような p' を探す
        - p は1つ前のブロックのプルーフ、 p' は新しいブロックのプルーフ
        :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):
        """
        プルーフが正しいかを確認する: hash(last_proof, proof)の最初の4つが0となっているか?
        :param last_proof: <int> 前のプルーフ
        :param proof: <int> 現在のプルーフ
        :return: <bool> 正しければ true 、そうでなければ false
        """

        guess = f'{last_proof}{proof}'.encode()
        guess_hash = hashlib.sha256(guess).hexdigest()

        return guess_hash[:4] == "0000"
```

アルゴリズムの難易度を調整するためには、最初の0の数を変えることで出来る。しかし4は充分な数だ。0を一つ加えることで、解を見つけるための時間にマンモス級の違いが出ることに気がつくだろう。

私たちのクラスはほとんど完成しているので、HTTPリクエストとともにこのクラスを使ってみよう。

ステップ2: APIとしての私たちのブロックチェーン

ここではPython Flaskフレームワークを使う。Flaskはマイクロフレームワークであり、簡単にエンドポイントをPythonのファンクションに対応させることが出来る。そして、私たちのブロックチェーンがHTTPリクエストを使ってWebで通信することが出来るようになる。

ここでは3つのメソッドを作る:

- * ブロックへの新しいトランザクションを作るための /transactions/new
- * サーバーに対して新しいブロックを採掘するように伝える /mine
- * フルブロックチェーンを返す /chain

Flaskをセットアップする

サーバーは、このブロックチェーンのネットワークに一つのノードを作り出す。早速いくつかの例となるコードを作ろう。

```
blockchain.py

# coding: UTF-8

import hashlib
import json
from textwrap import dedent
from time import time
from uuid import uuid4

from flask import Flask

class Blockchain(object):
    ...

# ノードを作る
# Flaskについて詳しくはこちらを読んでほしい http://flask.pocoo.org/docs/0.12/quickstart/#a-minimal-a
app = Flask(__name__)

# このノードのグローバルにユニークなアドレスを作る
node_idenfifire = str(uuid4()).replace('-', '')

# ブロックチェーンクラスをインスタンス化する
blockchain = Blockchain()

# メソッドはPOSTで/transactions/newエンドポイントを作る。メソッドはPOSTなのでデータを送信する
@app.route('/transactions/new', methods=['POST'])
def new_transactions():
    return '新しいトランザクションを追加します'

# メソッドはGETで/mineエンドポイントを作る
@app.route('/mine', methods=['GET'])
def mine():
    return '新しいブロックを採掘します'

# メソッドはGETで、フルのブロックチェーンをリターンする/chainエンドポイントを作る
@app.route('/chain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200

# port5000でサーバーを起動する
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

トランザクションエンドポイント

これらトランザクションのリクエストの例だ。このようなものをユーザーはサーバーに送る:

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

すでにブロックにトランザクションを加えるメソッドは作ってあるため、残りは簡単だ。トランザクションを加えるためのメソッドを書いていこう。

```
blockchain.py

# coding: UTF-8

import hashlib
import json
from textwrap import dedent
from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

...

# メソッドはPOSTで/transactions/newエンドポイントを作る。メソッドはPOSTなのでデータを送信する
@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    values = request.get_json()

    # POSTされたデータに必要なデータがあるかを確認
    required = ['sender', 'recipient', 'amount']
    if not all(k in values for k in required):
        return 'Missing values', 400

    # 新しいトランザクションを作る
    index = blockchain.new_transaction(values['sender'], values['recipient'], values['amount'])

    response = {'message': f'トランザクションはブロック {index} に追加されました'}
    return jsonify(response), 201
```

採掘のエンドポイント

採掘のエンドポイントは魔法が起きるところだが、簡単だ。3つのことを行う必要がある。

1. ブルーフ・オブ・ワークを計算する
2. 1コインを採掘者に与えるトランザクションを加えることで、採掘者（この場合は我々）に利益を与える
3. チェーンに新しいブロックを加えることで、新しいブロックを採掘する

```
blockchain.py
```

```
# coding: UTF-8

import hashlib
import json
from textwrap import dedent
from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

...

# メソッドはGETで/mineエンドポイントを作る
@app.route('/mine', methods=['GET'])
def mine():
    # 次のブルーフを見つけるためブルーフ・オブ・ワークアルゴリズムを使用する
    last_block = blockchain.last_block
    last_proof = last_block['proof']
    proof = blockchain.proof_of_work(last_proof)

    # ブルーフを見つけたことに対する報酬を得る
    # 送信者は、採掘者が新しいコインを採掘したことを表すために"0"とする
    blockchain.new_transaction(
        sender="0",
        recipient=node_identifire,
        amount=1,
    )

    # チェーンに新しいブロックを加えることで、新しいブロックを採掘する
    block = blockchain.new_block(proof)

    response = {
        'message': '新しいブロックを採掘しました',
        'index': block['index'],
        'transactions': block['transactions'],
        'proof': block['proof'],
        'previous_hash': block['previous_hash'],
    }
    return jsonify(response), 200
```

採掘されたブロックに含まれるトランザクションの受信者のアドレスは、自分のノードのアドレスであることに注意してほしい。そして、ここで行っていることの殆どは、ブロックチェーンクラスとのインタラクションにすぎない。一旦ここまでにして、このブロックチェーンとのインタラクションを始めよう。

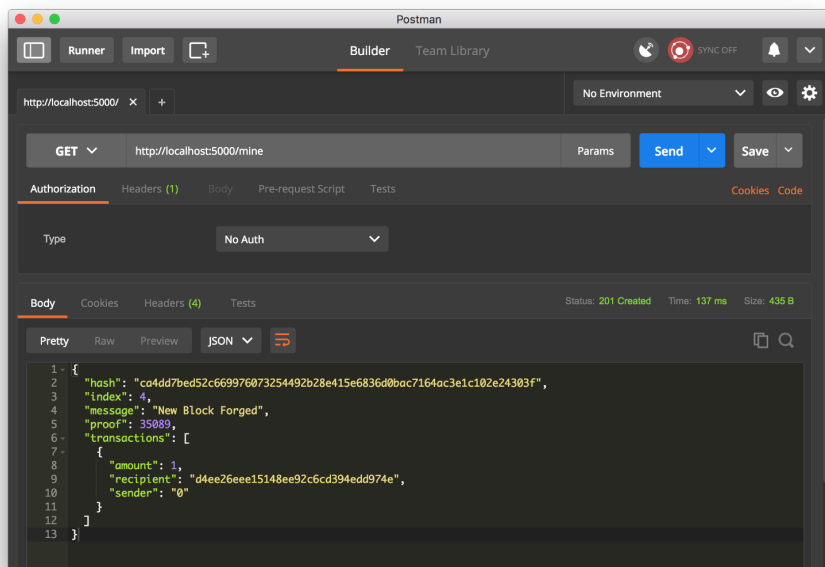
ステップ3: オリジナルブロックチェーンとのインタラクション

cURLかPostmanを使ってAPIを叩いてみよう

サーバーを起動する:

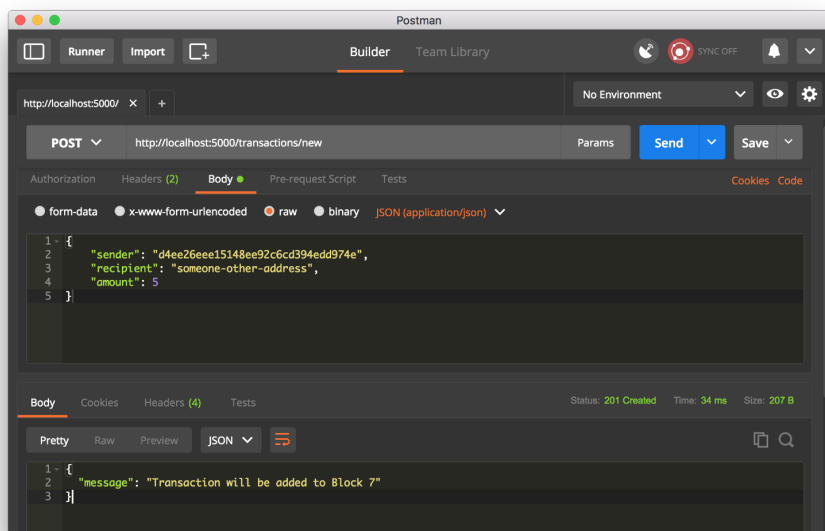
```
$ python blockchain.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

http://localhost:5000/mine への GET リクエストを作って採掘しよう



(<https://camo.qiitusercontent.com/6362bb20a4a2a1e6d1fc55c66e319288dbea75a9/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36383231332f38373738666265392d343034652d383139622d316465352d3263376531373436653730352e706e67>)

`http://localhost:5000/transactions/new` への POST リクエストを作って新しいトランザクションを作ろう。ボディに取引の内容を入れておく。



(<https://camo.qiitusercontent.com/6aa5bc7b15fbb7fd63d79528bfdf93e2401d27/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36383231332f63336265633166312d356136652d316632332d376134382d3535633539666163626163662e706e67>)

もしPostmanを使っていない場合、cURLを使っても同じことが出来る。

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

サーバーを再起動して、2ブロックを採掘した、つまりトータル3ブロックだ。ここでチェーン全体を `http://localhost:5000/chain` をリクエストすることで見てみよう。

```
{
  "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
}
```

ステップ4: コンセンサス

これはクールだ。トランザクションを受け付けて、新しいブロックを採掘できるブロックチェーンを作ることが出来た。しかしブロックチェーンの重要なポイントは、非中央集権的であることだ。そしてもし非中央集権的であれば、我々はどのように地球上の全員が同じチェーンを反映していると確認することが出来るだろうか。これはコンセンサスの問題と呼ばれており、もし1つより多くのノードをネットワーク上に持ちたければ、コンセンサスのアルゴリズムを実装しなければならない。

新しいノードを登録する

コンセンサスアルゴリズムを実装する前に、ネットワーク上にある他のノードを知る方法を作ろう。それぞれのノードがネットワーク上の他のノードのリストを持っていなければならない。なのでいくつかのエンドポイントが追加が必要となる。

1. URLの形で新しいノードのリストを受け取るための `/nodes/register`
2. あらゆるコンフリクトを解消することで、ノードが正しいチェーンを持っていることを確認するための `/nodes/resolve`

これから、我々のブロックチェーンの構造を編集し、ノード登録のためのメソッドを追加する:

```
blockchain.py

...
from urllib.parse import urlparse
...

class Blockchain(object):
    def __init__(self):
        ...
        self.nodes = set()
        ...

    def register_node(self, address):
        """
        ノードリストに新しいノードを加える
        :param address: <str> ノードのアドレス 例: 'http://192.168.0.5:5000'
        :return: None
        """

        parsed_url = urlparse(address)
        self.nodes.add(parsed_url.netloc)
```

ノードのリストを保持するのに `set()` を使ったことに注意してほしい。これは、新しいノードの追加がべき等 -同じノードを何回加えても、一度しか現れない- ということを実現するための簡単な方法だ。

コンセンサスアルゴリズムを実装する

以前言及したとおり、コンフリクトはあるノードが他のノードと異なったチェーンを持っているときに発生する。これを解決するために、最も長いチェーンが信頼できるというルールを作る。別の言葉で言うと、ネットワーク上で最も長いチェーンは事実上正しいものといえる。このアルゴリズムを使って、ネットワーク上のノード間でコンセンサスに到達する。

```
blockchain.py
```

```
...
import requests

class Blockchain(object):
    ...

    def valid_chain(self, chain):
        """
        ブロックチェーンが正しいかを確認する

        :param chain: <list> ブロックチェーン
        :return: <bool> True であれば正しく、 False であればそうではない
        """

        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")

            # ブロックのハッシュが正しいかを確認
            if block['previous_hash'] != self.hash(last_block):
                return False

            # ブルーフ・オブ・ワークが正しいかを確認
            if not self.valid_proof(last_block['proof'], block['proof']):
                return False

            last_block = block
            current_index += 1

        return True

    def resolve_conflicts(self):
        """
        これがコンセンサスアルゴリズムだ。ネットワーク上の最も長いチェーンで自らのチェーンを
        置き換えることでコンフリクトを解消する。
        :return: <bool> 自らのチェーンが置き換えられると True 、そうでなければ False
        """

        neighbours = self.nodes
        new_chain = None

        # 自らのチェーンより長いチェーンを探す必要がある
        max_length = len(self.chain)

        # 他のすべてのノードのチェーンを確認
        for node in neighbours:
            response = requests.get(f'http://{node}/chain')

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

                # そのチェーンがより長い、有効を確認
                if length > max_length and self.valid_chain(chain):
                    max_length = length
                    new_chain = chain

        # もし自らのチェーンより長く、かつ有効なチェーンを見つけた場合それで置き換える
        if new_chain:
            self.chain = new_chain
            return True
```

```
return False
```

この最初のメソッド `valid_chain()` は、チェーンの中の全てのブロックに対してハッシュとブルーフが正しいかを確認することで、チェーンが有効かどうかの判定を行っている。

`resolve_conflicts()` メソッドは、全てのネットワーク上のノードに対して、それらのチェーンをダウンロードし、上記のメソッドを使うことで確認している。もし有効なチェーンで自らのチェーンよりも長いものがあった場合、それで自らのチェーンを入れ替える。

次に2つのエンドポイントをAPIに追加しよう。1つはネットワーク上に他のノードを追加するため、もう1つはコンフリクトを解消するためのものだ。

```
blockchain.py
```

```
...
```

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

    nodes = values.get('nodes')
    if nodes is None:
        return "Error: 有効ではないノードのリストです", 400

    for node in nodes:
        blockchain.register_node(node)

    response = {
        'message': '新しいノードが追加されました',
        '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': 'チェーンが置き換えられました',
            'new_chain': blockchain.chain
        }
    else:
        response = {
            'message': 'チェーンが確認されました',
            'chain': blockchain.chain
        }

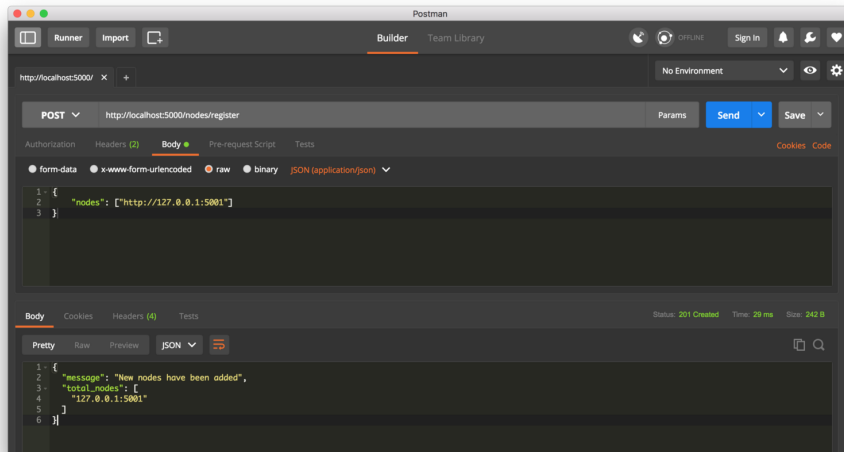
    return jsonify(response), 200
...
```

ここで、もう1つのマシンがあればそれを使って（訳注：複数マシン間でどのようにアクセスするのかは不明。ngrok使うとか？）別のノードを立ち上げる。または、同じマシンで違うポートから別のノードを立ち上げる。すなわち、 `http://localhost:5000` と `http://localhost:5001` という2つのノードが出来る。

まず、新しいノードを登録する。

訳注:ターミナルでcurlコマンドで日本語を表示するとユニコードエスケープで表示されてしまいます。その際は、jq (<https://stedolan.github.io/jq/>)をインストールして（macでHomebrewを使っていれば `brew install jq` で出来ます）、コマンドの後ろに `| jq` を加えるとデコードされて表示されます。

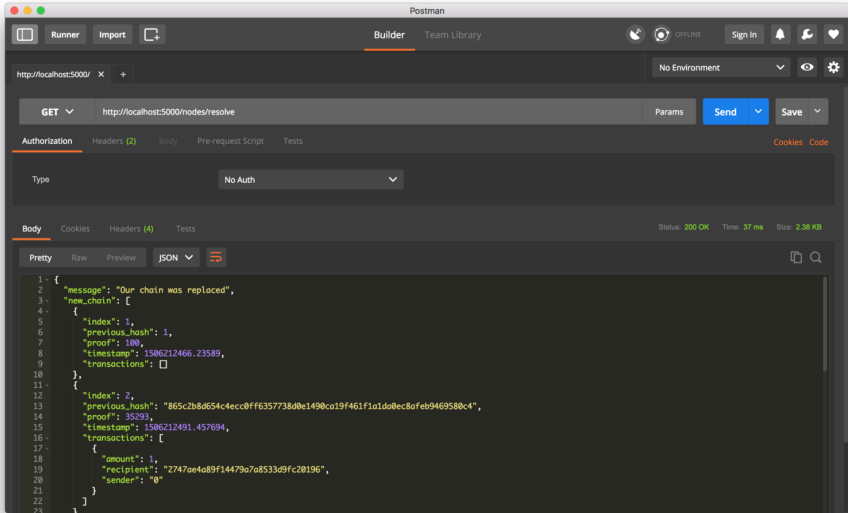
```
$ curl -X POST -H "Content-Type: application/json" -d '{
  "nodes": ["http://localhost:5001"]
}' "http://localhost:5000/nodes/register"
```



(<https://camo.qiitusercontent.com/8e10dcfbba489ecfa330adf0c6e48361bacb43a7/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36383231332f64313734616639312d666435382d376338612d346336382d3331346633336134633463312e706e67>)

そしてノード2のチェーンが長くなるように、いくつか新しいブロックをノード2で採掘する。その後、ノード1で `GET /nodes/resolve` を行い、コンセンサスアルゴリズムによりチェーンを置き換える。

```
$ curl "http://localhost:5001/mine"
$ curl "http://localhost:5000/nodes/resolve"
```

(<https://camo.qiitusercontent.com/a05f723ea2ca617f763c92fff5660db43a1f6829/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e616d617a6f6e6177732e636f6d2f302f36383231332f38313035313336382d616139392d346232642d363535622d3661623339366536363465322e706e67>)

これでおしまいだ。友達と我々のブロックチェーンを試してみしてほしい。

これがあなたを何か新しいものを作るよう奮い立たせるよう願っている。ブロックチェーンが、我々の経済・政府・記録の保管への考え方を急速に変えていくと信じているので、私は暗号通貨に対して非常に興奮している。

アップデート: この続きとなるパート2を計画中だ。そこでは、トランザクションを確認するメカニズムと、このブロックチェーンを実際に使えるようにするためのいくつかの方法についての議論を追加する予定だ。

もしこのガイドを楽しんでくれたのなら、または提案や質問があれば、コメントで知らせてほしい。また、バグを見つけたら気軽にここ (<https://github.com/dvf/blockchain>)にコントリビュートしてほしい！

📁

ストック

👍 いいね

1770

🔗

(<https://qiita.com/hidehiro98/items/841ece65d896aeaa8a2a/likers>)

🐦

Tweet

📢

251

👍

G+


(<https://qiitadon.com/share?text=%F3%R3%Q6%F3%R3%AF%F3%R3%A1%F3%R2%A7%F3%R3%AC%F3%R3%F3%R2%Q2%F4%RD%9C%F3%R2%RR%F3%R1%Q3%F3%R1%AR%F3%R1%A7%F5%AD>)

👍 Like

314

🔖 Pocket

553



Hidehiro Nagaoka (/hidehiro98) @hidehiro98 (/hidehiro98)

👤 フォロー


(/hidehiro98)


<https://qiita.com/hidehiro98/items/841ece65d896aeaa8a2a>


17/22


🔗 この記事は以下の記事からリンクされています


↑ 過去の3件を表示する

 **Swiftでブロックチェーンを実装してみる** (/shu223/items/ebe59325f36fbf25e3d6#_reference-659274524e259701a63f) からリンク 4ヶ月前

 **Rubyでブロックチェーンを実装してみる** (/shiki_tak/items/d890e909d14546147f8c#_reference-9400cd7d71b03dd671f6) からリンク 4ヶ月前

 **Rubyでブロックチェーンを実装してみる (進行中 1/3くらい)** (/fuji_syan/items/3a5fc1d910d968cbf041#_reference-ff7651696dc027299834) からリンク 4ヶ月前

 **シンプルなブロックチェーンをKotlin(サーバーサイド)で実装する** (/masayuki5160/items/a0ec176601d690ed0b8e#_reference-89f5f64de18494a7bdf1) からリンク 19日前

 **Clojure でブロックチェーンを実装してみる** (/snufkon/items/92d95bceaa6020d5d2ac#_reference-7e311d5ec7b161f12270) からリンク 12日前



ina111 (/ina111)

(/ina111) 128 contribution

2017-10-28 13:54

👍 いいね

2

タイポだと思いますが、誤記気づきました。

文中の

```
def new_transaction():
```

```
  の中身、value -> values
```



hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2017-11-01 11:29

👍 いいね

1

@ina111 (/ina111) さん、タイポでした！ご指摘ありがとうございます。直しておきました。



vneetcom (/vneetcom)

(/vneetcom) 1 contribution

2017-12-11 08:30

👍 いいね

1

Java版を作ってみました。

<https://github.com/vneetcom/javacoin/tree/master/javacoin>

(<https://github.com/vneetcom/javacoin/tree/master/javacoin>)

Daniel van Flymenさんのサイトに報告済みです。



hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2017-12-26 12:05

👍 いいね

2

@vneetcom (/vneetcom) さん、ご連絡ありがとうございます！



reon777 (/reon777)

(/reon777) 246 contribution

2018-01-12 17:00

👍 いいね

1

とても参考になりました！

小さい部分ですが、

```
if not self.valid_proof(last_block['proof'], block['proof'])
```

で末尾の「:」が抜けているみたいです。



hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2018-01-12 17:06

👍 いいね

1

@reon777 (/reon777) さん、ご連絡ありがとうございます！修正いたしました。



mtdkki (/mtdkki)

(/mtdkki) 23 contribution

2018-01-24 10:44

👍 いいね

1

こんにちは。
 valid_proof内の、
 guess = f'{last_proof}{proof}'.encode()
 について質問させてください。
 例えば、
 last_proof, proof = (1,23)
 と、(12,3)で、同じhashが出来てしまうと思うんですけど、問題はないんでしょうか？


hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2018-01-24 14:08

👍 いいね

1

こんにちは！質問ありがとうございます。
 確かに同じhashが出来てしまいますが、「0000が出るまで試行を行う」ことが目的なので問題は
 ありません。


mtdkki (/mtdkki)

(/mtdkki) 23 contribution

2018-01-24 15:21

👍 いいね

1

確かにそうですね。
 ありがとうございます😊


takabosoft (/takabosoft)

(/takabosoft) 2196 contribution

2018-01-25 11:23

👍 いいね

1

素敵な記事をありがとうございます。苦戦しつつ読み進めております。

新しいノードを登録するところの

```
parsed_url = parse(address)
```

は

```
parsed_url = urlparse(address)
```

かなと思います。

ついでなのですが、「ブロックとはどのようなものなのか」の節の

「それまでのブロックのハッシュ」(previous_hash)という表現は

一つ前のブロックのハッシュという理解でよろしいでしょうか？

(「それまで」ですと、過去全てのブロック全体のハッシュのように読めてしまったのですが)


hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2018-01-26 20:29

👍 いいね

1

ご指摘ありがとうございます！修正いたしました。

「それまでのブロックのハッシュ」ですが、訳が分かり難かったですね。「一つ前のブロック
 のハッシュ」というよりも、「それまでの全てのブロックから生成されるハッシュ」ですね。
 Blockchain クラスの hash メソッドをご覧くださいとわかりやすいかと思います。訳も修正して
 おきます。


3timesdo (/3timesdo)

(/3timesdo) 1 contribution

2018-01-30 19:42

👍 いいね

1

こんにちは。素晴らしい記事をありがとうございます。

for node in neighbours の行末に : が無いようです。

また、質問なのですが、コンセンサスにより、短い方のチェーンにより書かれていたブロック
 が破棄された場合、
 記事の実装では、そのブロックに記述されていたトランザクションは失われると思います。

破棄したノードで新たなブロックにつなぎ直すのでしょうか、
 それともトランザクション自体が、ノード間で共有されているなどで、繋ぎ変えたチェーンに
 すでに記述されているのでしょうか。

こんなに単純では無いと思いますが、よくある実装ではどういった方法がとられているのか疑問に思いました。



hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2018-01-30 20:19

いいね

1

ご指摘ありがとうございます！修正いたしました。

はい、短いチェーンのみに存在するブロックは失われます。

その点はビットコインも同じです。

<https://bitflyer.jp/ja-jp/glossary/fork> (<https://bitflyer.jp/ja-jp/glossary/fork>)



3timesdo (/3timesdo)

(/3timesdo) 1 contribution

2018-01-31 10:09

いいね

1

回答ありがとうございます。

短いチェーンのブロックが失われた際に、そのブロックに"のみ"記述されていたトランザクションは、
どのように、生き残る方のチェーンに取り込まれるのかと思いましたが、

<https://coincheck.com/blog/292> (<https://coincheck.com/blog/292>)

新しい取引は全ノードに送信される。
各ノードが新しい取引をブロックに取り入れる。

ブロックチェーンでは、こういう前提になっているので、今回の実装のように、
いずれかのノードのチェーンにしか存在しないトランザクションは生まれないのですね。



raucha (/raucha)

(/raucha) 74 contribution

2018-02-06 21:58

いいね

2

こんにちは。

面白く拝見させてもらいました、ありがとうございます

プルーフオブワークの実装について疑問なのですが、こちらの実装では各ブロックのトランザクションが改ざんから保護されていないのでは無いでしょうか

具体的には、以下のPoWの実装部分について、新しいPoWの計算に前回のPoWしか使われていないため、過去のブロックを改ざんしようとした場合、PoWを書き換ええないままトランザクションの改ざんが可能だと思います

```
def proof_of_work(self, last_proof):
    """
    シンプルなプルーフ・オブ・ワークのアルゴリズム:
    - hash(pp') の最初の4つが0となるような p' を探す
    - p は前のプルーフ、p' は新しいプルーフ
    :param last_proof: <int>
    :return: <int>
    """
    proof = 0
    while self.valid_proof(last_proof, proof) is False:
        proof += 1
    return proof
```

トランザクションの改ざんは previous_hash の値の矛盾で検出できますが、こちらは高速に再計算が可能のため、previous_hash 自体を改ざん後のブロックの内容に合わせて書き換え直せばトランザクションの改ざんの検出は不可能になります

PoWを生成する種の数値として、トランザクション内容のハッシュ等が含まれるような形になると思うのですが、どうなのでしょう



mtdkki (/mtdkki)

(/mtdkki) 23 contribution

2018-02-09 11:54

いいね

0

rauchaさんと同様の疑問を持っていて、それについて少し調べてみたので、コメントしてみます。

blockexplorer.comや、blockchain.infoでは、それぞれのブロックのハッシュ値と、previous

hashが、0000...となっているので、proof of workで出てきたハッシュを使いまわしているのかな、と思います。
もしそうだとすると当然、rauchaさんがおっしゃっているように、トランザクション内容のハッシュや、previous hash等もproof of workに含んでいるはずです。

**hidehiro98 (/hidehiro98)**

(/hidehiro98) 1797 contribution

2018-02-09 16:18

いいね

0

@raucha (/raucha) さん、コメントありがとうございます。
おっしゃる通り、このPoWではproofにはブロックの情報が含まれていないため、取引の改ざんが可能です。
これは、PoWをできるだけシンプルに作るためこうなっているものであり（参考：
<https://github.com/dvf/blockchain/issues/10#issuecomment-336665322>
(<https://github.com/dvf/blockchain/issues/10#issuecomment-336665322>)）、もちろん過去のhashを入れるなどして改ざんを難しくすることは可能です。

**apple_orange (/apple_orange)**

(/apple_orange) 0 contribution

2018-02-18 14:04

いいね

0

こんにちは。
大変勉強になる記事ありがとうございます。

一点質問ですが、ブロックチェーンの仕組みの中に

最初に計算に成功したノード（マイナー）が承認ブロックを全ノードに伝える

があると思うのですが、この部分は今回は実装されていないという認識で宜しいでしょうか？
宜しくお願い致します。

**masatoshi_Q (/masatoshi_Q)**

(/masatoshi_Q) 1 contribution

2018-03-05 14:21

いいね

0

貴重な記事のアップ、誠にありがとうございます。
子どもの頃、MSX パソコンの BASIC プログラムでゲームを作ったりした経験があるだけの、40代のおじさんです。
にわかなブロックチェーンブームにあやかり、何かいいことないかなと模索しておところの・・・ど素人です。

近々はこの半年の間に Python を勉強して、仕事で開発している装置に AI プログラムを実装したぐらいのプログラミングスキルしかないので、未知の分野に足を踏み入れると分からないことだらけで、毎日ググってばかりの日々です。

ようやくブロックチェーンに関する本記事を見つけ、しかも Python しか使えない私にとっては、まさに砂漠の中のオアシスにたどり着いたような心境です。

(プログラミング環境: Mac OS Sierra 10.12.4 Python 3.6.1)

早速、序盤とも言えるところで、つまづいてしまっておりまして・・・

ステップ 3: オリジナルブロックチェーンとのインタラクション

サーバーを起動させて、

<http://localhost:5000/mine> (<http://localhost:5000/mine>) への GET リクエストを作って採掘できたのですが、

次の

<http://localhost:5000/transactions/new> (<http://localhost:5000/transactions/new>) への POST リクエストを作って新しいトランザクションを作るところで、エラーになります。

POSTMAN では、500 Internal Server Error

ターミナルでは、TypeError: argument of type 'NoneType' is not iterable

いろいろと試行錯誤してみましたが・・・解決しません。

◆ 新しいトランザクションを作る（加える？）関数が2つ定義されているようですが、
class Blockchain 内で定義されているのが、関数 new_transaction
@app.route('/transactions/new', methods=['POST'])デコレータの次行にある、関数

```
new_transaction(s?)
```

これらは、別のものと考えてよいですか？

◆ POST リクエストの際に、ボディに入れる内容は下記のとおりで、よいでしょうか？

```
{
  "sender": "mine で採掘したときに、POSTMAN に表示された recipient をコピーすればよいですか？"
  "recipient": "someone-other-address"
  "amount": 5 "5"という具合に囲わなくてよいですか？
}
```

◆ また @app.route('/transactions/new', methods=['POST'])デコレータの次行にある、関数 new_transaction(s?)は、new_transaction ? new_transactions ? values = request.get_json() は、 values = request.json() ? ググっても request.get_json() というコードは、ちまたにあまり見当たらないです。

それとも、

ステップ 4: コンセンサス以降のコードも打ち込んでから、ステップ 3 における試運転を行った方がよいですか？

せっかくありがたい記事ですので、是非とも先に進めて、ブロックチェーンについての理解を深めたい次第です。

発展途上のブロックチェーン技術ですので、民主的に皆で盛り上げて発展させていければと思います。

お忙しいところ恐縮でございますが・・・お助けくださると幸いです。



pomtaro (/pomtaro)

(/pomtaro) 1 contribution

2018-04-01 18:59

👍 いいね

1

masatoshi_Qさん

<https://github.com/dvf/blockchain/issues/75> (<https://github.com/dvf/blockchain/issues/75>)
こちらに解決策ありました。



hidehiro98 (/hidehiro98)

(/hidehiro98) 1797 contribution

2018-04-03 12:45

👍 いいね

0

@pomtaro (/pomtaro) さん、ありがとうございます！



masatoshi_Q (/masatoshi_Q)

(/masatoshi_Q) 1 contribution

2018-04-09 10:00

👍 いいね

0

@pomtaro (/pomtaro) さん、@hidehiro98 (/hidehiro98) さん、たいへんありがとうございます。お手数お掛け致しました。
皆さんの足を引っ張らないよう、しっかり勉強してまいります。
仮想通貨の取引所も再編が進み、今後、ブロックチェーンもどうなることでしょう・・・数年後に残る技術なのか・・・
いろいろ調べたり、いろんな方のお話を聞きますが・・・結局は、もやもやした感じは解消されないです。