**作业1 – SHA256算法实现及其在区块链中的作用**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名： | 程梦烨 | | 学号： | 3180103733 | | | 专业： | 软件工程 | | |
| 课程名称： | | 区块链与数字货币 | | | 同组学生姓名： |  | | | |
| 实验时间： | | 2020-12-4 | 实验地点： | |  | | 指导老师： | | 杨小虎 | |

# 实验环境

* 操作系统：windows10
* 编程语言：python
* 集成开发环境：python 3.6 IDLE+vscode

# 实验内容和要求

## 2.1 实验要求

提交一个SHA256的代码实现，并用实验数据说明SHA256在比特币区块链中发挥的作用。

## 2.2 实验内容

* 使用python语言实验SHA256加密算法，并对输出结果 python自带的的SHA256函数验证正确性。经检验可以对任意长度的任意可显示符号（中文、英文、数字等）进行hash散列加密，并设计实验证明其具有的特性。
* 设计多个实验说明SHA256在比特币区块链中发挥的作用

# SHA256代码实现设计思路

## 3.1 消息预处理

在计算消息的哈希摘要之前需要对消息进行预处理：

对消息进行补码处理：假设消息M的二进制编码长度为l位。首先在消息末尾补上一位”1”, 然后再补上k个"0", 其中k为下列方程的最小非负整数:



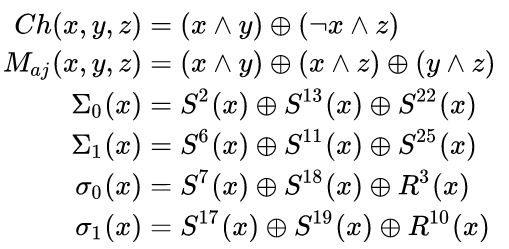
最终补完以后的消息二进制位数长度是512的倍数。

这里需要注意的两点是不管原来的消息长度是多少，即使长度已经满足对512取模后余数是448，补位也必须要进行，这时要填充512位。另外，考虑到最后要将消息长度l转换为64位二进制编码，因此，长度的必须小于264，绝大多数情况，这个足够大了。

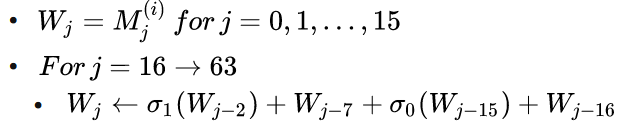
将补码处理后的消息以512位为单位分块为：，其中第i个消息块的前32位表示为：，后面32位为： ， 以此类推，最后32位的消息块可表示为：。我们采用Big endian约定对数据进行编码，即认为第一个字节是最高位字节，因此，对于每一个32位字节，最最左边的比特是最大的比特位。

## 3.2 逻辑函数定义

SHA256算法当中所使用到的6个逻辑函数如下：每个函数都对32位字节进行操纵，并输出32位字节。



扩展消息块通过以下方式进行计算：

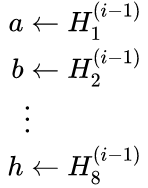


## 3.3 哈希计算主循环

哈希计算算法如下:

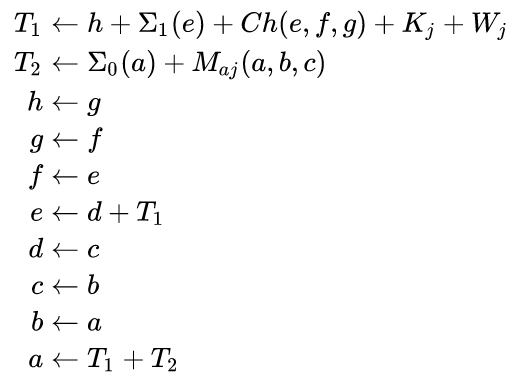
* For i=1->N (N=补码后消息块个数)

用第(i-1)个中间哈希值来对a, b, c, d, e, f, g, h 进行初始化，当i=1时, 就使用初始化哈希，即：

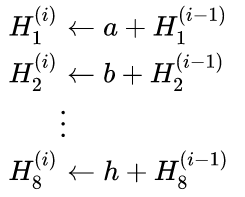


* 应用SHA256压缩函数来更新a, b, …, h

For j=0->63，计算（具体定义如下）



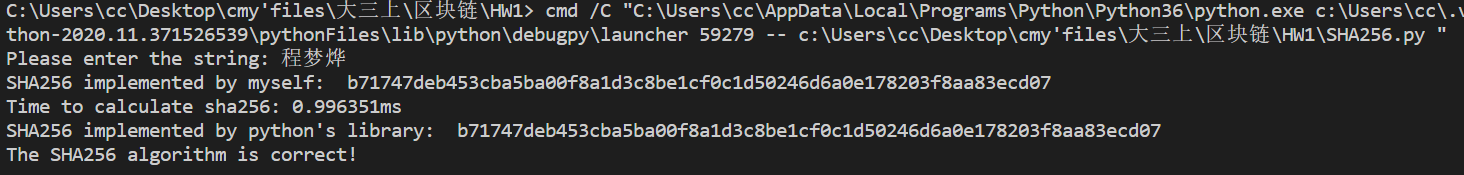
* 计算第i个中间哈希值



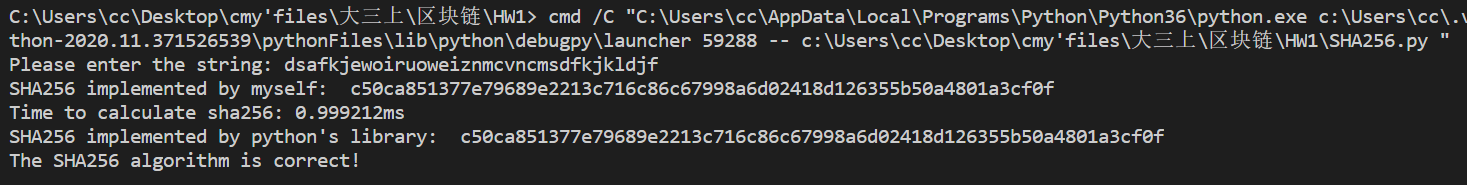
* 为最终需要的哈希M

# 测试结果

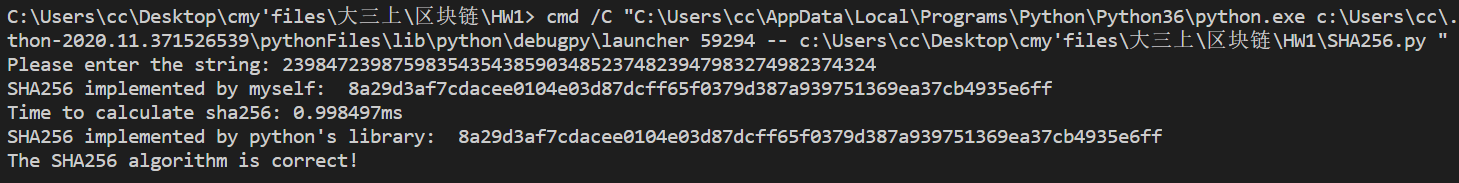
* **中文测试**

****

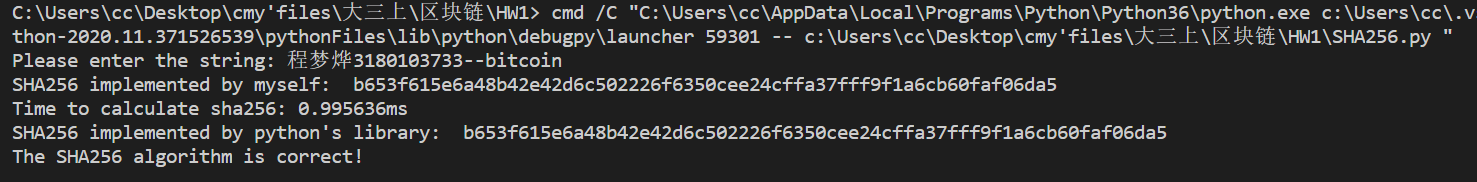
* **英文测试**

****

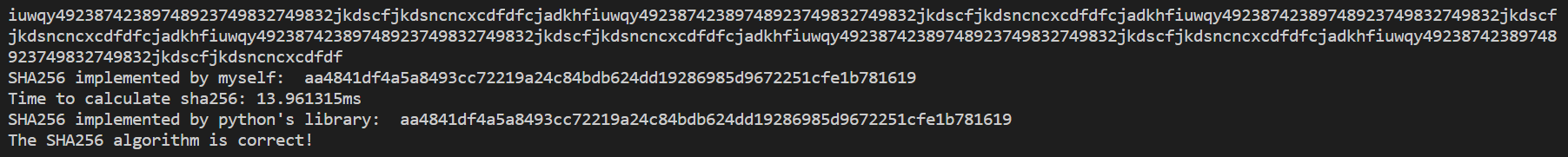
* **数字测试**

****

* **混合测试**

****

* **长字符串测试（测试案例过长，这里不予展示）**

****

**结果分析：**

从以上测试结果可见，已正确实现了SHA256加密算法，并通过python的SHA256函数验证了结果的正确性，对任意长度的任意字符实现了Hash散列算法。

# 源代码

import hashlib

import time

h0 = 0x6a09e667

h1 = 0xbb67ae85

h2 = 0x3c6ef372

h3 = 0xa54ff53a

h4 = 0x510e527f

h5 = 0x9b05688c

h6 = 0x1f83d9ab

h7 = 0x5be0cd19

k = [0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2]

#将x循环右移b个bit

def RightR(x,b):

    return (x>>b)|(x<<32-b)

#逻辑函数定义

def Ch(x,y,z):

    return (x & y) ^ (~x & z)

def Ma(x,y,z):

    return (x & y) ^ (x & z) ^ (y & z)

def Sum0(x):

    return RightR(x,2) ^ RightR(x,13) ^ RightR(x,22)

def Sum1(x):

    return RightR(x,6) ^ RightR(x,11) ^ RightR(x,25)

def Sigma0(x):

    return RightR(x,7) ^ RightR(x,18) ^ (x>>3)

def Sigma1(x):

    return RightR(x,17) ^ RightR(x,19) ^ (x>>10)

#哈希计算算法

def Hash(message,h0,h1,h2,h3,h4,h5,h6,h7,k):

    #处理多种输入，中文/英文

    if isinstance(message, str):

        message = bytearray(message, 'ascii')

    elif isinstance(message, bytes):

        message = bytearray(message)

    elif not isinstance(message, bytearray):

        raise TypeError

    #消息预处理

    l=len(message)\*8

    message.append(0x80)

    while (len(message)\*8+64)%512!=0:

        message.append(0x00)

    message+=l.to\_bytes(8,'big')

    #将消息分解成512-bit大小的块

    blocks=[]

    for i in range(0,len(message),64):

        blocks.append(message[i:i+64])

    #哈希计算主循环（64次）

    for m\_block in blocks:

        words=[]

        #构造64个word

        for j in range(0,16):

            words.append(bytes(m\_block[j\*4:j\*4+4]))

        for j in range(16,64):

            temp1=Sigma1(int.from\_bytes(words[j-2],'big'))

            temp2=int.from\_bytes(words[j-7],'big')

            temp3=Sigma0(int.from\_bytes(words[j-15],'big'))

            temp4=int.from\_bytes(words[j-16],'big')

            words.append(((temp1+temp2+temp3+temp4)%2\*\*32).to\_bytes(4,'big'))

        a=h0

        b=h1

        c=h2

        d=h3

        e=h4

        f=h5

        g=h6

        h=h7

        #进行64次加密循环

        for j in range(0,64):

            t1=(h+Sum1(e)+Ch(e,f,g)+k[j]+int.from\_bytes(words[j],'big'))%2\*\*32

            t2=(Sum0(a)+Ma(a,b,c))%2\*\*32

            h=g

            g=f

            f=e

            e=(d+t1)%2\*\*32

            d=c

            c=b

            b=a

            a=(t1+t2)%2\*\*32

        h0=(a+h0)%2\*\*32

        h1=(b+h1)%2\*\*32

        h2=(c+h2)%2\*\*32

        h3=(d+h3)%2\*\*32

        h4=(e+h4)%2\*\*32

        h5=(f+h5)%2\*\*32

        h6=(g+h6)%2\*\*32

        h7=(h+h7)%2\*\*32

    return ((h0).to\_bytes(4,'big') + (h1).to\_bytes(4,'big') +(h2).to\_bytes(4,'big') + (h3).to\_bytes(4,'big')

            +(h4).to\_bytes(4,'big') + (h5).to\_bytes(4,'big') +(h6).to\_bytes(4,'big') + (h7).to\_bytes(4,'big'))

#main函数

#------------------------------------------------------------------------------

message=input("Please enter the string: ")

start=time.time()#记录SHA256算法所用时间

result1=Hash(message.encode('utf-8'),h0,h1,h2,h3,h4,h5,h6,h7,k).hex()

end=time.time()

print('SHA256 implemented by myself: ',result1)

print('Time to calculate sha256: %fms'%((end-start)\*1000))

#调用python库的SHA256检验加密结果

result2=hashlib.sha256(message.encode('utf-8')).hexdigest()

print('SHA256 implemented by python\'s library: ',result2)

if result1==result2:

    print('The SHA256 algorithm is correct!')

# 设计实验说明SHA256在区块链中的作用

## 6.1 实验一：利用SHA256对比两个大文件内容是否相同

* **实验说明：**

SHA256具有快速验证和防止篡改的作用。哈希函数在区块链中，生成各种数据的摘要，当比较两个数据是否相等时，只需要比较他们的摘要就可以了。例如，比较两个交易是否相等，只需要比较两者的hash值，快捷又方便。

传递一个数据，要保证它在传递过程中不被篡改，只需要同时传递它的摘要即可。收到数据的人将这个数据重新生成摘要，然后比较传递的摘要和生成的摘要是否相等，如果相等，则说明数据在传递过程中没有被篡改。在设计比特币的时候之所以选择SHA256，主要是看中了SHA256在验证改动方面有着巨大的优势。因为只要输入数据有微小的区别，通过SHA256计算出来的数值都会有巨大的差距。

因此本实验通过设计对大文件或者长字符串比较hash值来比较其具体内容，来证明SHA256在区块链应用中具有验证速度快和防止篡改的作用。

* **源代码**

#实验1：利用SHA256对比两个大文件内容是否相同

#1.证明SHA256验证速度快

print('Please input two long strings to show the verification speed of SHA256 is fast.')

print('Ouput True if the two strings are the same.')

a=input('Please input the string1: ')

b=input('Please input the string2: ')

#通过SHA256比较两个大文件

start\_t=time.time()

m1=hashlib.sha256(a.encode('utf-8'))

m2=hashlib.sha256(b.encode('utf-8'))

print((m1.hexdigest()) == (m2.hexdigest()))

end\_t=time.time()

print('Time to compare two strings by SHA256: %fms'%((end\_t-start\_t)\*1000))

#普通字符串比较

flag=1

for i in range(0,len(a)):

    if a[i]!=b[i] or i>len(b):

        flag=0

print(bool(flag))

end\_t2=time.time()

print('Time to compare two strings normally: %fms'%((end\_t2-end\_t)\*1000))

#2.证明SHA256有防止篡改的作用

print('Please input more than one string with slight difference to prove that SHA256 can prevent tampering.')

a=input('Please input the string1: ')

print('S1: ',hashlib.sha256(a.encode('utf-8')))

b=input('Please input the string2: ')

print('S2: ',hashlib.sha256(b.encode('utf-8')))

c=input('Please input the string1: ')

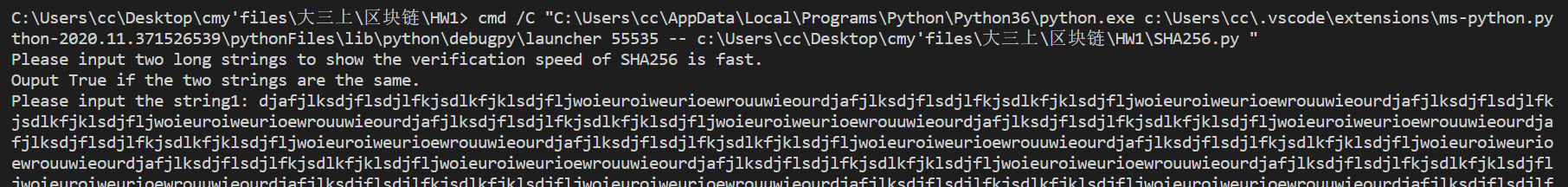
print('S3: ',hashlib.sha256(c.encode('utf-8')))

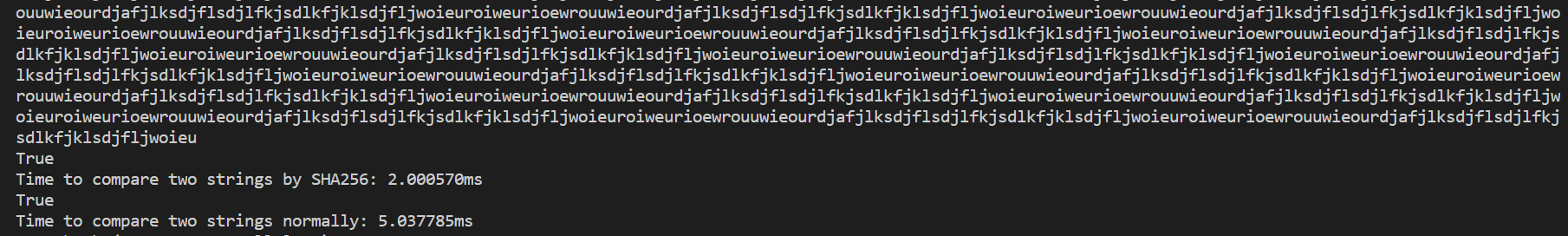
d=input('Please input the string2: ')

print('S4: ',hashlib.sha256(d.encode('utf-8')))

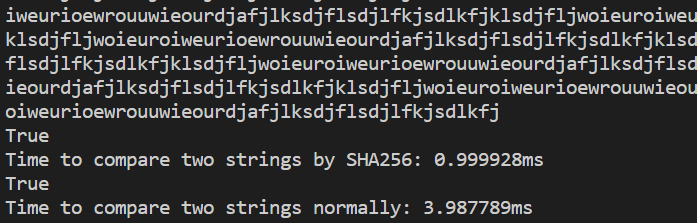
* **运行结果**

使用了较长的字符串对比运行时间：（超过30KB）

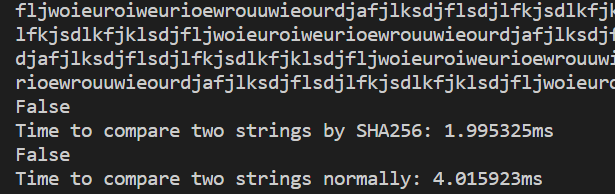
****

****

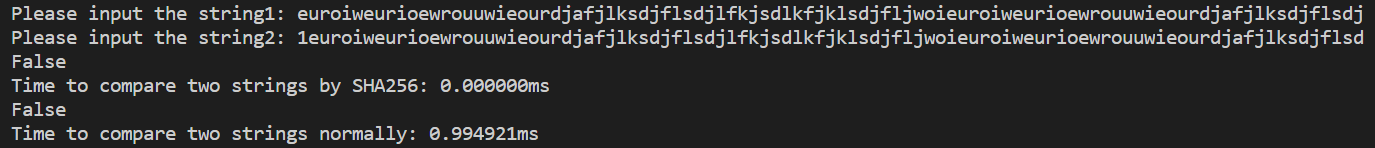
更长的字符串：（超过60KB）

****

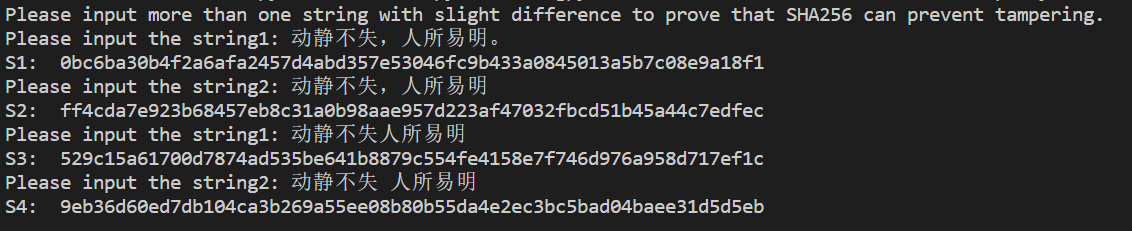
修改字段2使其与字段1不同（多次验证，这里仅展示一次）：

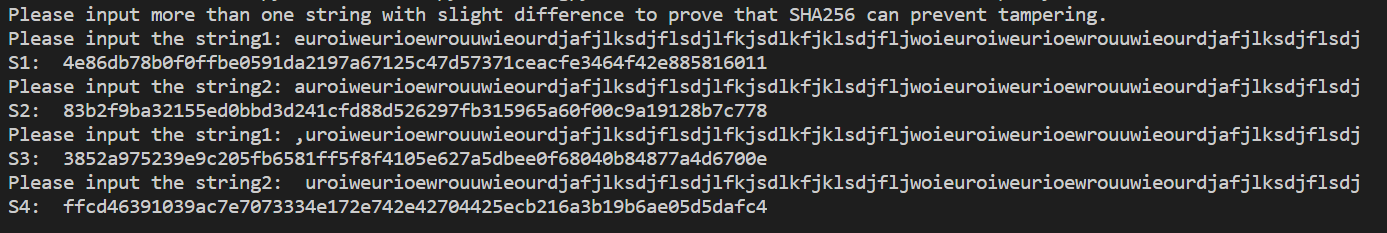


如果数据较小的话，体现不出来SHA256加密比较hash值的优势：



输入差异很小的字符串，得到的SHA256加密结果差异很大：





* **结果分析**

为何要使用SHA，这与区块链的技术要求以及SHA算法的特性有直接的关系。

首先，区块链本身要保证每一条都是独一无二。因为绝不能出现两条一样的数据，而SHA256极低的碰撞概率正好能够保证这一点。每个区块的Hash之都是唯一的正好可以用来标识该区块。

其次，区块链要保证每条记录的安全性，而SHA算法可以保证哪怕只有微不足道的差别，也能产生大相径庭的结果。结合区块链的结构，当任意区块发生变化时，都会影响后续的所有区块，所以如果想要修改某个区块的内容，就必须修改后许所有区块的内容，而这几乎是不可能的，从而保证了区块数据的安全。

再者，SHA的结果长度是固定的，而与具体的原文内容无关，这样可以保证区块链中每个区块的格式以及用来标识区块的区块头的大小是完全一致的，但却绝对不会重复。

综上所述，区块链使用SHA算法是一种技术上的必然选择，即使不采用SHA也必是与SHA具有相似特性的其它算法。

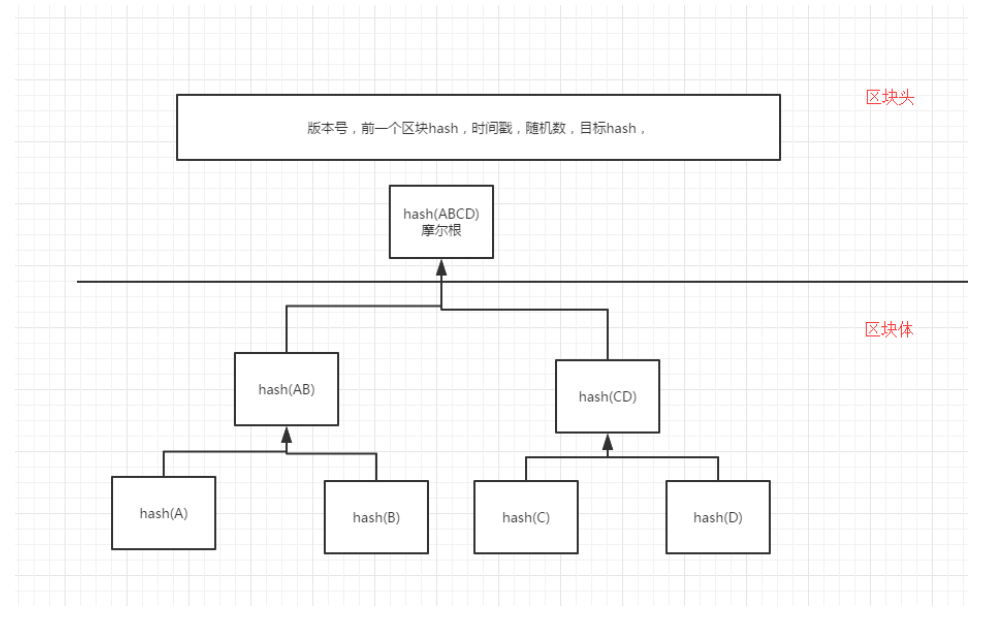
而通过对大文件计算Hash值摘要的思想也应用在数字签名当中。公钥加密然后私钥解密，可以用于通信中拥有公钥的一方向拥有私钥的另一方传递机密信息，不被第三方窃听。实际应用中，由于直接对原消息进行签名有安全性问题，而且原消息往往比较大，直接使用RSA算法进行签名速度会比较慢，所以我们一般对消息计算其摘要（使用SHA-256等安全的摘要算法），然后对摘要进行签名。只要使用的摘要算法是安全的（MD5、SHA-1已经不安全了），那么这种方式的数字签名就是安全的。

## 6.2 实验二：Merkle Tree在区块链中的应用

在比特币中，Merkle Tree的每个节点的值都是通过两次SHA256算法得到的：

HA = SHA256(SHA256(Transaction A))，HB = SHA256(SHA256(Transaction B))，

那么HA&B 就是对HA和HB串联起来在进行两次SHA256算法。



每个摩尔根是多个交易的集合，区块链中每个摩尔根的值都是唯一的。除此之外，merkle树还提供了一种在比特币网络中快速校验某个交易是否存在的有效途径。

这里我们使用一个二叉树来模拟Merkle Tree的操作，过程分为：准备交易的数据；计算出每个数据的哈希值，从左到右逐步组成树的左右节点；执行循环直到最后一个节点。这里核心的算法就是构造树，并且要计算每个节点的哈希值。

private List<String> getNewTxList(List<String> tempTxList) {

    List<String> newTxList = new ArrayList<String>();

    int index = 0;

    while (index < tempTxList.size()) {

      // left

      String left = tempTxList.get(index);

      index++;

      // right

      String right = "";

      if (index != tempTxList.size()) {

        right = tempTxList.get(index);

      }

      // sha2 hex value

      String sha2HexValue = getSHA2HexValue(left + right);

      newTxList.add(sha2HexValue);

      index++;

}

}

测试： 我们构造一个5个叶子节点的二叉树，然后测试返回根节点的哈希值

public static void main(String[] args) {

        List<String> tempTxList = new ArrayList<String>();

        tempTxList.add("a1");

        tempTxList.add("b2");

        tempTxList.add("c3");

        tempTxList.add("d4");

        tempTxList.add("e5");

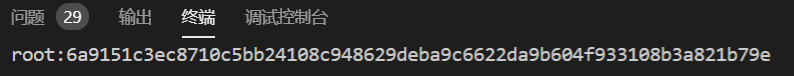
        MerkleTrees merkleTrees = new MerkleTrees(tempTxList);

        merkleTrees.merkle\_tree();

        System.out.println("root : " + merkleTrees.getRoot());

    }

执行结果：



**结果分析：**

在比特币的Merkle树中两次使用到了SHA256 算法，因此其加密哈希算法也被称为double-SHA256。

在比特币网络中，Merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，且提供了一种校验区块是否存在某交易的高效途径。生成一棵完整的Merkle树需要递归地对一对节点进行哈希，并将新生成的哈希节点插入到Merkle树中，直到只剩一个哈希节点，该节点就是Merkle树的根。

当N个数据元素经过加密后插入Merkle树时，你至多计算2\*log~2~(N) 次就能检查出任意某数据元素是否在该树中，这使得该数据结构非常高效。

## 6.3 实验三：区块链挖矿（寻找前置零）

加密电子货币中，比特币就是通过让参与者利用这样的加密算法求解出符合特定条件的哈希值来实现“挖矿”过程。具体来说，比特币要求参与者通过 double SHA-256 算法计算出“前导0”超过若干位的哈希值，第一个求解出来的参与者就是“获胜的矿工”。

简单的挖矿程序，实现逻辑为：进行无限次hash碰撞，取字符串前几位，如果前几位都为0，则碰撞成功，相当于暴力破解。

import hashlib

x=1

while x:

    mystr = hashlib.sha256(f"{x}".encode("utf-8")).hexdigest()

    print(x , mystr)

    if mystr[:6] == "000000":

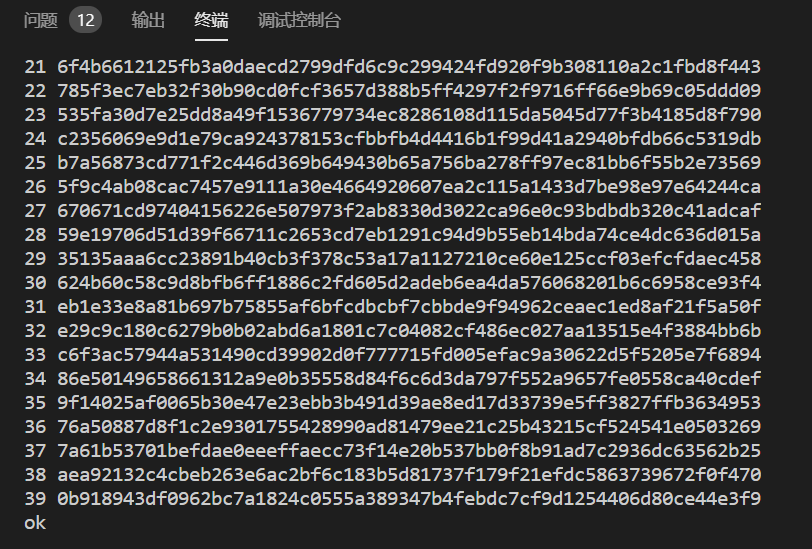
        break

    x += 1

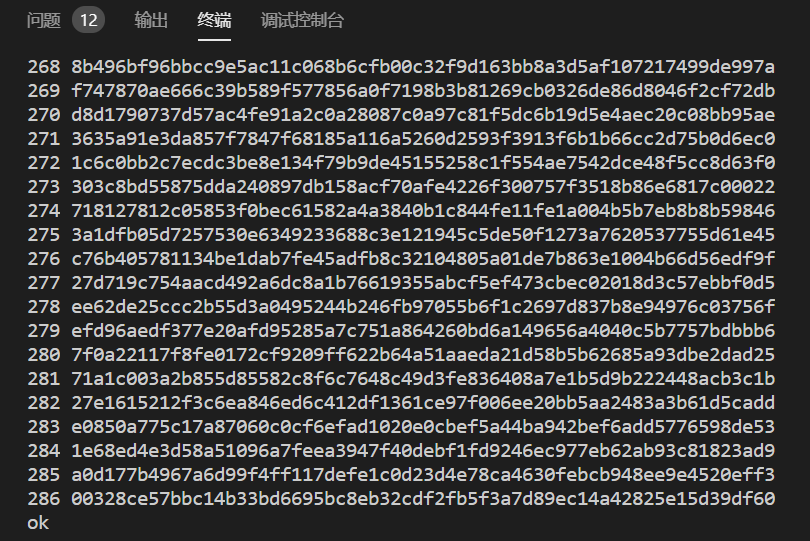
print("ok")

运行结果：

寻找一个前置0：



寻找两个前置0：



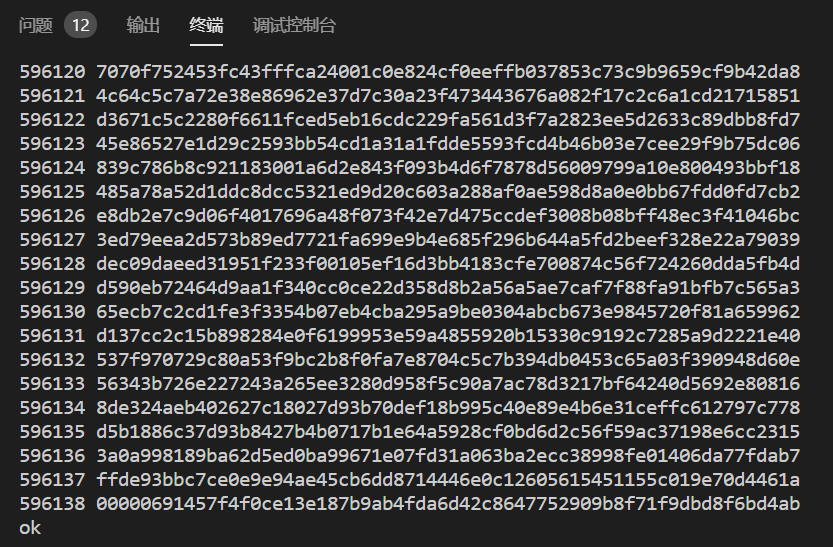
寻找3个前置0：



寻找4个前置0：



寻找5个前置0：



寻找6个前置0：



结果分析：

SHA 256在比特币协议中发挥着关键的作用，任何人都可以在自己的计算机上运行这个数学函数，但没有人能逆推它。如果你可以逆推这个算法，你就可以比其他人挖矿挖得更快，然后赚很多钱。而从以上实验设计中可以看出，挖矿难度和所耗时间随着寻找的前置零的个数增加大大增加。

比特币挖矿的过程，就是找到输入值的过程，这个输入值通过SHA 256算法产生了一串开头带有约70个零的输出值。但是，因为没有已知的公式，每个人能做的也只是通过蛮力一个数一个数去试，直到找到正确的输入值。不过比特币挖矿只需要找到一个接近的哈希值就好，不需要完全匹配。实际上，挖矿难度可以调整，保证大概每十分钟能有矿工找到匹配的输入值，然后赚取12.5比特币的出块奖励，这就是为什么我们认为它是地球上最流行的算法，或者应该说是地球上最常用的算法。

# 讨论与心得

SHA256是一种密码学单向哈希函数，比特币中有几处用到，一是公钥转换成公钥地址，也就是比特币地址；二是在计算Merkle树的时候用到了，对每笔交易进行两次sha256；三是在挖矿过程中的工作量证明（PoW）中用到，计算区块的sha256哈希值小于难度值。

SHA-256参与了比特币地址的创建，这些地址被称为私钥，是用户之间进行交易所必需的。地址是由随机数组成的唯一代码。创建的密钥应通过两种算法-SHA-256和RIPEMD160。这就是公式的样子（K代表密钥，B代表比特币地址）

B = RIPEMD160（SHA-256（K））

使用这两种算法的主要优点是有机会创建短私钥。公钥包含256位，而私钥几乎短两倍，仅包含160位。这是为用户和他们的方便而设计的，因为手动操作较短的地址要容易得多。

merkle根也是由SHA-256创建的，并放置在块头中。该元素存储有关网络中执行的所有交易的数据。

Merkel Tree合并结点和计算结点值，均是利用的SHA256算法计算hash值，同时，也可以在log(树高)时间内完成对Merkel Tree内叶结点元素的检索。

用于POW共识算法工作量证明。这个主要是在pow的共识算法中使用。详细说来，就是给定一定的数据，然后让你寻找其他的数据，合并起来计算出来的hash值小于某个值。比特币、目前的以太坊，都是使用的POW共识。

比特币本身是个公开的账本，每一个区块就是一页账，从第一页开始，每一页上都有一个这样的哈希值。每一页的哈希值，都是由上一页的哈希值和剩下的信息通过SHA256得到的结果。如果有人修改之前的账目，那么从他修改的下一页账开始，每一页的哈希值都会完全不同，会被所有人一眼看穿。这样被修改过的账本，无法得到51%以上的人承认，就无法写入区块中，因此修改是无效的。这样的设计，就保证了比特币这个分布式账本的不可篡改性。