## 一．算法设计

1. **字符串的处理**

填充字符串末尾，使其长度在对512取模以后的余数是448：先补第一个比特为1，然后都补0，直到长度满足对512取模后余数是448。此时再在填充字符串末尾加上表示原始字符串长度的64位数据，则有：448+64=512，刚好拼成一个完整的结构，用于下一步算法的迭代实现。

1. **将输入的字符串str转换成N个大小为512bit的块**

整个算法不断进行迭代，每次迭代产生结果Hi，Map(Hi-1) = Hi，通过N次迭代产生HN，即256bit的结果哈希值。将该256bit的哈希值分成8块，每块32bit，恰好是一个字word。所以可以将每次迭代看成是8个子部分的迭代。

**(3)每一次迭代Map(Hi-1) = Hi的具体实现方式**

原始字符串(N\*512bits)，可以分成N块，每块512bits，将每一块分解为16个32bits (16\*32=512)的字，记为w[0], …, w[15]， 前16个字直接由字符串的第i个块分解得到。

第一次迭代将映射的初始值设置为自然数前8个质数的平方根的小数部分的前32位。

其余的字由如下迭代公式得到：



**(4)64个加密循环完成一次迭代**

将初始的8个字，即自然数的前8个质数的平方根的小数部分的前32位，按照一定的规则进行更新。每次迭代后的初始值分别为H[i-1][0],H[i-1][1],…,H[i-1][7].

设置64个密钥K[64]：自然数前64个质数的平方根的小数部分的前32位。

Wt是本区块产生第t个word。原消息被切成固定长度512-bit的区块，对每一个区块，产生64个word，通过重复运行循环n次对每次迭代传下来的那八个字循环加密。最后一次循环所产生的八个字合起来就是第i个块对应到的散列字符串Hi。即，完成了哈希算法。

**经过多次实验和正确的SHA256算法下多个字符串的哈希值进行对比，该实验结果正确。**

## 二．实验数据说明

通过不断输入字符串，用SHA256算法进行实验，总结出SHA256的以下几点特点：

1. **SHA256在验证改动方面有着巨大的优势。**

只要输入数据有微小的区别，通过SHA256产生的哈希值都会有巨大的差距。如下面3个字符串，原文仅仅存在微小的不同，但是通过SHA256的哈希处理所产生的哈希值的差异巨大，且这种差异毫无规律可循。  
str1：1   
SHA156-Value: 6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b

Str2：2   
SHA156-Value: d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35

Str2：3   
SHA156-Value: 4e07408562bedb8b60ce05c1decfe3ad16b72230967de01f640b7e4729b49fce

基于这个特性，SHA-256成为了比特币网络的重要组成部分:

**参与比特币地址的创建，区块链可以基于哈希值进行连接。**每一个区块中保留了上一个区块的哈希值，merkle根也是由SHA-256创建的，并放置在块头中。层层相扣的结构以及分布式账本的记账机制维护了区块链的加密性。在自动生成输入的同时，任何用户都可以随机生成输出。任何人都无法更改或中断现有输入。使得区块链中数据无法篡改或删除。同时由于SHA-256的唯一性，区块头对应产生的哈希值可以作为区块的标识符，维护了比特币的唯一性。

1. **SHA256可作为公平比特币获取的依据。**

通过算法可知，SHA256加密的过程、计算、产生的过程中存在众多的位运算等逻辑运算，对算力的要求很高。是对计算机算力比拼的一个很好的机制。

机器通过运行SHA256程序不断的生成随机数，在众多机器中，谁先生成符合的随机数谁就可以得到该比特币，如，设置谁先生成前四位均为0的哈希值谁就可以获得该次的比特币，这种机制实现了对计算机算力的考验，也实现了比特币在算力面向相对公平的竞争，不会存在普通货币所谓的“阶级差距”、“贫富差距”带来的购买力的不同。

## 三．算法实现代码

**Main.cpp**

1. #include <iostream>
2. #include "sha256.h"
3. **using** **namespace** std;
5. **int** main()
6. {
7. string str;                //输入字符串
8. unsigned **char** res[64] = {};//产生的64位哈希值
9. **char** finalRes[65] = {};    //格式化输出
11. cout << "Input your string: " << endl;//输入随机字符串
12. cin >> str;
14. SHA256 sha256 = SHA256();
15. //迭代过程和生成过程
16. sha256.Iteration((unsigned **char**\*)str.c\_str(), str.length());
17. sha256.Generate(res);
18. **for** (**int** i = 0; i < 32; i++) sprintf(finalRes + i \* 2, "%02x", res[i]);
20. cout << endl;//输出结果
21. cout << "SHA256-Value: "<< string(finalRes) << endl;
22. **return** 0;
23. }

**SHA256.h**

1. #pragma once
2. #include <string>
3. **using** **namespace** std;

6. **class** SHA256
7. {
8. **public**:
9. **void** Generate(unsigned **char** \*res);
10. **void** Iteration(**const** unsigned **char** \*input, unsigned **int** len);
12. **protected**:
13. unsigned **char** input\_block[128];        //存放处理成的块
14. unsigned **int** total\_length, init\_length;//输入字符串补好的长度和初始长度
15. **const** **static** unsigned **int** zhishu64[64];//前8(64)个质数的平方根(立方根)的小数部分的前64位
16. unsigned **int** zhishu8[8] = { 0x6a09e667 ,0xbb67ae85,0x3c6ef372,0xa54ff53a,0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19 };
17. **void** Padding(**const** unsigned **char** \*input, unsigned **int** block\_nb);
18. };
20. #define SHA2\_SHFR(a, b)    (a >> b)
21. #define SHA2\_ROTR(a, b)   ((a >> b) | (a << ((sizeof(a) << 3) - b)))
22. #define SHA2\_ROTL(a, b)   ((a << b) | (a >> ((sizeof(a) << 3) - b)))
23. #define SHA2\_CH(a, b, c)  ((a & b) ^ (~a & c))
24. #define SHA2\_MAJ(a, b, c) ((a & b) ^ (a & c) ^ (b & c))
25. #define SHA256\_F1(a) (SHA2\_ROTR(a,  2) ^ SHA2\_ROTR(a, 13) ^ SHA2\_ROTR(a, 22))
26. #define SHA256\_F2(a) (SHA2\_ROTR(a,  6) ^ SHA2\_ROTR(a, 11) ^ SHA2\_ROTR(a, 25))
27. #define SHA256\_F3(a) (SHA2\_ROTR(a,  7) ^ SHA2\_ROTR(a, 18) ^ SHA2\_SHFR(a,  3))
28. #define SHA256\_F4(a) (SHA2\_ROTR(a, 17) ^ SHA2\_ROTR(a, 19) ^ SHA2\_SHFR(a, 10))
29. #define SHA2\_UNPACK32(a, s)                 \
30. {                                             \
31. \*((s) + 3) = (unsigned **char**) ((a)      );       \
32. \*((s) + 2) = (unsigned **char**) ((a) >>  8);       \
33. \*((s) + 1) = (unsigned **char**) ((a) >> 16);       \
34. \*((s) + 0) = (unsigned **char**) ((a) >> 24);       \
35. }
36. #define SHA2\_PACK32(s, a)                   \
37. {                                             \
38. \*(a) =   ((unsigned **int**) \*((s) + 3)      )    \
39. | ((unsigned **int**) \*((s) + 2) <<  8)    \
40. | ((unsigned **int**) \*((s) + 1) << 16)    \
41. | ((unsigned **int**) \*((s) + 0) << 24);   \
42. }

**SHA256.CPP**

1. #include <cstring>
2. #include <fstream>
3. #include "SHA256.h"
4. **using** **namespace** std;
6. //前64个质数的立方根的小数部分的前64位
7. **const** unsigned **int** SHA256::zhishu64[64] =
8. {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 };
10. **void** SHA256::Padding(**const** unsigned **char** \*input, unsigned **int** block\_number)
11. {//填充字符串
12. unsigned **int** word[64],v[8];
13. **const** unsigned **char** \*temp;
14. **for** (**int** i = 0; i < (**int**)block\_number; i++) {
15. temp = input + (i << 6);
16. **for** (**int** j = 0; j < 16; j++) SHA2\_PACK32(&temp[j << 2], &word[j]);
17. **for** (**int** j = 16; j < 64; j++) word[j] = SHA256\_F4(word[j - 2]) + word[j - 7] + SHA256\_F3(word[j - 15]) + word[j - 16];//构造64个word
18. **for** (**int** j = 0; j < 8; j++)  v[j] = zhishu8[j];
19. **for** (**int** j = 0; j < 64; j++) {
20. unsigned **int** m = v[7] + SHA256\_F2(v[4]) + SHA2\_CH(v[4], v[5], v[6])+ zhishu64[j] + word[j];
21. unsigned **int** n = SHA256\_F1(v[0]) + SHA2\_MAJ(v[0], v[1], v[2]);
22. v[7] = v[6];
23. v[6] = v[5];
24. v[5] = v[4];
25. v[4] = v[3] + m;
26. v[3] = v[2];
27. v[2] = v[1];
28. v[1] = v[0];
29. v[0] = m+n;
30. }
31. **for** (**int** j = 0; j < 8; j++)  zhishu8[j] += v[j];
32. }
33. }

36. **void** SHA256::Iteration(**const** unsigned **char** \*input, unsigned **int** len)
37. {//加密循环
38. unsigned **int** block\_number, tmp;
39. **const** unsigned **char** \*new\_input;
40. **if** (init\_length + len < 64) tmp = len;
41. **else** tmp = 64 - init\_length;
42. memcpy(&input\_block[init\_length], input, tmp);
43. **if** (init\_length + len < 64)  init\_length += len;
44. **else** {
45. block\_number = (len - tmp) / 64;
46. new\_input = input + tmp;
47. Padding(input\_block, 1);
48. Padding(new\_input, block\_number);
49. tmp = (len - tmp) % 64;
50. memcpy(input\_block, &new\_input[block\_number << 6], tmp);
51. init\_length = tmp;
52. total\_length += (block\_number + 1) << 6;
53. }
54. }
56. **void** SHA256::Generate(unsigned **char** \*res)
57. {//生成哈希值
58. unsigned **int** block\_number = (1 + (55 < (init\_length % 64)));
59. unsigned **int** len1 = block\_number << 6, len2 = (total\_length + init\_length) << 3;
60. memset(input\_block + init\_length, 0, len1 - init\_length);
61. input\_block[init\_length] = 0x80;
62. SHA2\_UNPACK32(len2, input\_block + len1 - 4);
63. Padding(input\_block, block\_number);
64. **for** (**int** i = 0; i < 8; i++) SHA2\_UNPACK32(zhishu8[i], &res[i << 2]);
65. }