**SHA256算法及找相关nonce**

1. **SHA256算法原理简介**

SHA-256（Secure Hash Algorithm 256-bit）是一种广泛使用的加密哈希算法，它将任意长度的输入数据转化为长度固定的256位哈希值（散列值）。SHA-256算法的原理包括以下几个关键步骤：

1. 消息填充（Padding）

长度扩展：为了确保消息长度为512位的倍数，先在消息后添加一个1位，后面填充若干个0，再附加一个64位的消息长度（以比特为单位）。

结果：填充后的消息长度是512的倍数。

2. 初始化哈希值

使用八个32位的常数作为初始哈希值（H0, H1, ..., H7）。这些常数是由平方根的前八个素数的前32位组成。（0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19）

3. 消息分块和扩展

将消息分成多个512位的分组块。对每个512位的分组，进一步划分为16个32位的字，然后通过特定的位运算扩展到64个字。

4. 压缩函数

对每个512位的分组，SHA-256依次对64个字进行处理。算法使用64个混淆常数（由前64个素数的立方根计算得出），结合之前的哈希值、当前消息块，通过一系列的逻辑运算（包括循环右移、选择函数、合并函数等）来更新哈希值。

每个迭代过程中，消息通过一些复杂的位运算与初始哈希值（H0~H7）混合，生成新的临时哈希值。

5. 更新哈希值

处理完每个分组后，当前的哈希值会与上一步的结果累加，更新初始哈希值。

6. 输出

所有分组处理完成后，最终得到的256位的输出，即哈希值。

1. **代码实现及实验结果**

实验代码如下（源文件放在同一目录下的SHA256Encoder.java文件里）：

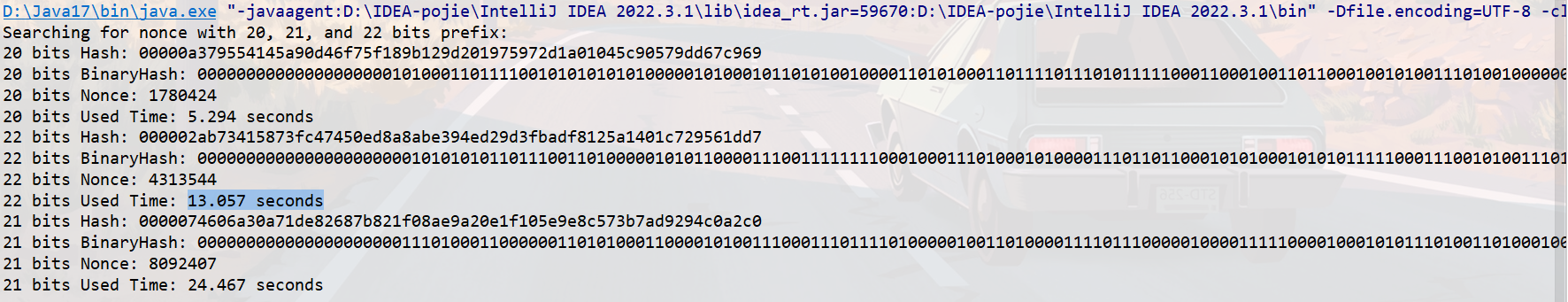
import java.util.Arrays;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.atomic.AtomicBoolean;  
import java.util.concurrent.atomic.AtomicLong;  
  
*/\*\*  
@author hwj  
@create: 2024-09-22 20:41  
@Description:   
\*/*public class SHA256Encoder {  
  
 *// 初始哈希值（H0, H1, ..., H7）* private static final int[] *H* = {  
 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,  
 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19  
 };  
  
 *// SHA-256的64个混淆常量K* private static final int[] *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  
 };  
  
 *// 循环右移* private static int rightRotate(int value, int bits) {  
 return (value >>> bits) | (value << (32 - bits));  
 }  
  
 *// SHA-256算法实现* public static String sha256(String message) {  
 byte[] bytes = message.getBytes();  
 int[] paddedMessage = *padMessage*(bytes);  
  
 int[] hash = Arrays.*copyOf*(*H*, *H*.length); *// 初始化hash值* int[] w = new int[64]; *// 消息扩展块  
  
 // 处理每一个512位的块* for (int i = 0; i < paddedMessage.length / 16; i++) {  
 *// 拆分（每一个 512bits 块拆成 16 个 32bits 块放入w[0~15]中）* System.*arraycopy*(paddedMessage, i \* 16, w, 0, 16);  
  
 *// 扩散* for (int t = 16; t < 64; t++) {  
 int s0 = *rightRotate*(w[t - 15], 7) ^ *rightRotate*(w[t - 15], 18) ^ (w[t - 15] >>> 3);  
 int s1 = *rightRotate*(w[t - 2], 17) ^ *rightRotate*(w[t - 2], 19) ^ (w[t - 2] >>> 10);  
 w[t] = w[t - 16] + s0 + w[t - 7] + s1;  
 }  
  
 *// 初始哈希值* int a = hash[0];  
 int b = hash[1];  
 int c = hash[2];  
 int d = hash[3];  
 int e = hash[4];  
 int f = hash[5];  
 int g = hash[6];  
 int h = hash[7];  
  
 *// 64次迭代混淆* for (int t = 0; t < 64; t++) {  
 int s1 = *rightRotate*(e, 6) ^ *rightRotate*(e, 11) ^ *rightRotate*(e, 25);  
 int ch = (e & f) ^ ((~e) & g);  
 int temp1 = h + s1 + ch + *K*[t] + w[t];  
 int s0 = *rightRotate*(a, 2) ^ *rightRotate*(a, 13) ^ *rightRotate*(a, 22);  
 int maj = (a & b) ^ (a & c) ^ (b & c);  
 int temp2 = s0 + maj;  
  
 h = g;  
 g = f;  
 f = e;  
 e = d + temp1;  
 d = c;  
 c = b;  
 b = a;  
 a = temp1 + temp2;  
 }  
  
 *// 计算下一块的初始哈希值* hash[0] += a;  
 hash[1] += b;  
 hash[2] += c;  
 hash[3] += d;  
 hash[4] += e;  
 hash[5] += f;  
 hash[6] += g;  
 hash[7] += h;  
 }  
  
 *// 将哈希值转换为十六进制字符串* StringBuilder hexString = new StringBuilder();  
 for (int hval : hash) {  
 hexString.append(String.*format*("%08x", hval));  
 }  
 return hexString.toString();  
 }  
  
 *// 填充消息* private static int[] padMessage(byte[] message) {  
 int originalLength = message.length;  
 int blockCount = ((originalLength + 8) / 64) + 1; *// 保证填充位数在1~512之间* int totalLength = blockCount \* 16; *// 以32位整数为单位* int[] paddedMessage = new int[totalLength];  
  
 *// 将消息复制到填充后的数组中* for (int i = 0; i < originalLength; i++) {  
 paddedMessage[i / 4] |= (message[i] & 0xFF) << (24 - (i % 4) \* 8);  
 }  
  
 *// 添加1位'1'，其余位补零* paddedMessage[originalLength / 4] |= 0x80 << (24 - (originalLength % 4) \* 8);  
  
 *// 添加原消息的长度（以比特为单位）* long messageBitsLength = (long) originalLength \* 8;  
 paddedMessage[totalLength - 2] = (int) (messageBitsLength >>> 32);  
 paddedMessage[totalLength - 1] = (int) messageBitsLength;  
  
 return paddedMessage;  
 }  
  
 *// 将哈希值转换为二进制字符串* public static String hexToBinary(String hex) {  
 StringBuilder binary = new StringBuilder();  
 for (int i = 0; i < hex.length(); i++) {  
 String bin = Integer.*toBinaryString*(Integer.*parseInt*(hex.substring(i, i + 1), 16));  
 bin = "0".repeat(4-bin.length()) + bin;  
 binary.append(bin);  
 }  
 return binary.toString();  
 }  
  
 *// 查找前n位为0的nonce* public static void findNonce(String baseText, int zeroBits) {  
 long startTime = System.*currentTimeMillis*();  
 long nonce = 0;  
 while (true) {  
 String testText = baseText + nonce;  
 String hash = *sha256*(testText);  
 String binaryHash = *hexToBinary*(hash);  
 *// 检查前n位是否为0* if (binaryHash.startsWith("0".repeat(zeroBits))) {  
 long endTime = System.*currentTimeMillis*();  
 long duration = endTime - startTime;  
 System.*out*.println("Hash: " + hash);  
 System.*out*.println("Nonce: " + nonce);  
 System.*out*.println("Used Time: " + (duration / 1000.0) + " seconds");  
 break;  
 }  
 nonce++;  
 }  
 }  
  
 *// 查找前n位为0的nonce（多线程版本）* public static void findNonceParallel(String baseText, int zeroBits, int threadCount) {  
 long startTime = System.*currentTimeMillis*();  
 AtomicLong nonce = new AtomicLong(0);  
 AtomicBoolean found = new AtomicBoolean(false); *// 是否找到nonce* ExecutorService executor = Executors.*newFixedThreadPool*(threadCount);  
  
 for (int i = 0; i < threadCount; i++) {  
 executor.submit(() -> {  
 while (!found.get()) {  
 long currentNonce = nonce.getAndIncrement();  
 String testText = baseText + currentNonce;  
 String hash = *sha256*(testText);  
 String binaryHash = *hexToBinary*(hash);  
 *// 检查前n位是否为0* if (binaryHash.startsWith("0".repeat(zeroBits))) {  
 *// 确保是前zeroBits位为0，且后面位不为0* if (binaryHash.charAt(zeroBits) != '0') {  
 found.set(true); *// 标记为找到* long endTime = System.*currentTimeMillis*();  
 System.*out*.println("Hash: " + hash);  
 System.*out*.println("BinaryHash: " + binaryHash);  
 System.*out*.println("Nonce: " + currentNonce);  
 System.*out*.println("Used Time: " + (endTime - startTime) / 1000.0 + " seconds");  
 }  
 }  
 }  
 });  
 }  
  
 executor.shutdown();  
 while (!executor.isTerminated()) {  
 *// 等待所有线程完成* }  
 }  
  
 *// 同时查找30/31/32前缀为0的nonce（多线程版本）* public static void findMultiNonceParallel(String baseText, int threadCount) {  
 long startTime = System.*currentTimeMillis*();  
 AtomicLong nonce = new AtomicLong(0);  
 AtomicBoolean found30 = new AtomicBoolean(false);  
 AtomicBoolean found31 = new AtomicBoolean(false);  
 AtomicBoolean found32 = new AtomicBoolean(false);  
 ExecutorService executor = Executors.*newFixedThreadPool*(threadCount);  
  
 for (int i = 0; i < threadCount; i++) {  
 executor.submit(() -> {  
 while (!(found30.get() && found31.get() && found32.get())) {  
 long currentNonce = nonce.getAndIncrement();  
 String testText = baseText + currentNonce;  
 String hash = *sha256*(testText);  
 String binaryHash = *hexToBinary*(hash);  
  
 *// 检查前n位是否为0* if (!found30.get() && binaryHash.startsWith("0".repeat(30)) && binaryHash.charAt(30) != '0') {  
 found30.set(true);  
 long endTime = System.*currentTimeMillis*();  
 System.*out*.println("30 bits Hash: " + hash);  
 System.*out*.println("30 bits BinaryHash: " + binaryHash);  
 System.*out*.println("30 bits Nonce: " + currentNonce);  
 System.*out*.println("30 bits Used Time: " + (endTime - startTime) / 1000.0 + " seconds");  
 }  
  
 if (!found31.get() && binaryHash.startsWith("0".repeat(31)) && binaryHash.charAt(31) != '0') {  
 found31.set(true);  
 long endTime = System.*currentTimeMillis*();  
 System.*out*.println("31 bits Hash: " + hash);  
 System.*out*.println("31 bits BinaryHash: " + binaryHash);  
 System.*out*.println("31 bits Nonce: " + currentNonce);  
 System.*out*.println("31 bits Used Time: " + (endTime - startTime) / 1000.0 + " seconds");  
 }  
  
 if (!found32.get() && binaryHash.startsWith("0".repeat(32)) && binaryHash.charAt(32) != '0') {  
 found32.set(true);  
 long endTime = System.*currentTimeMillis*();  
 System.*out*.println("32 bits Hash: " + hash);  
 System.*out*.println("32 bits BinaryHash: " + binaryHash);  
 System.*out*.println("32 bits Nonce: " + currentNonce);  
 System.*out*.println("32 bits Used Time: " + (endTime - startTime) / 1000.0 + " seconds");  
 }  
 }  
 });  
 }  
  
 executor.shutdown();  
 while (!executor.isTerminated()) {  
 *// 等待所有线程完成* }  
 }  
  
 public static void main(String[] args) {  
 String baseText = "Blockchain@ZhejiangUniversity";  
 *// 优化版，在一个方法里同时找30、31、32前缀0的nonce* System.*out*.println("Searching for nonce with 30, 31, and 32 bits prefix:");  
 *findMultiNonceParallel*(baseText, 4); *// 4 个线程  
// System.out.println("30bits zero prefix nonce:");  
// //findNonce(baseText,30);  
// findNonceParallel(baseText, 30, 4);  
// System.out.println("-------------------------");  
// System.out.println("31bits zero prefix nonce:");  
// //findNonce(baseText,31);  
// findNonceParallel(baseText, 31, 4);  
// System.out.println("-------------------------");  
// System.out.println("32bits zero prefix nonce:");  
// //findNonce(baseText,32);  
// findNonceParallel(baseText, 32, 4);* }  
}

实验结果：

由于一开始没有用多线程优化计算，导致时间开销巨大，后期迭代了两版优化方案（第一版采用多线程来计算，但采用串行化来计算30/31/32对应情形下的nonce，第二版采取在一个方法里同时计算30/31/32对应情形下的nonce，避免了重复计算）。

优化之后计算30/31/32预估仍旧需要2h以上的计算时间，故我将bits改为了20/21/22，结果如下：

20位前缀0的nonce是1780424，用时5.294 seconds；21位前缀0的nonce是8092407，用时24.467 seconds；22位前缀0的nonce是4313544，用时13.057 seconds



1. **参考文章**

[SHA256算法原理详解-CSDN博客](https://blog.csdn.net/u011583927/article/details/80905740)

[一文读懂哈希算法SHA256-CSDN博客](https://blog.csdn.net/qq_51473302/article/details/124851177)