SHA256 的实现

> 学号：3190104783
> 姓名：欧翌昕

SHA256 原理

对于任意长度的输入，SHA256 都会产生一个 256 位的哈希值，称作消息摘要。这个摘要相当于是个长度为 32 个字节的数组，通常由一个长度为 64 的十六进制字符串来表示，一个十六进制字符的长度为 4 位。

▼ 总体上，SHA256与MD4、MD5 以及 SHA-1 等哈希函数的操作流程类似，待哈希的消息在继续哈希计算之前首先要进行以下两个步骤：

- 对消息进行补位处理，使得最终的长度是512位的倍数
- 以 512 位为单位对消息进行分块为 $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$

消息区块将进行逐个处理：从一个固定的初始哈希 $H^{(0)}$ 开始，进行以下序列的计算：

$$H^{(i)} = H^{(i-1)} + C_{M^{(i)}}\left(H^{(i-1)}\right)$$

其中 C 是 SHA256 的压缩函数，$+$ 是 $\mod 2^{32}$ 加法，$H^{(N)}$ 是消息区块的哈希值。

算法细节

▼ SHA256 的压缩函数主要对 512 位的消息区块和 256 位的中间哈希值进行操作，本质上，它是一个通过将消息区块为密钥对中间哈希值进行加密的 256 位加密算法。 因此，为了描述 SHA256 算法，有以下两方面的组件需要描述：

- SHA256 压缩函数
- SHA256 消息处理流程

▼ 描述中所用到的标记（针对32位字节）：

- 按位异或：$\oplus$
- 按位与：$\wedge$
- 按位或：$\vee$
- 补位：$\neg$
- 相加以后对 2^{32} 求余：$+$
- 右移 n 位：R^n
- 循环右移 n 位：S^n

常量初始化

初始哈希值 $H^{(0)}$ 取自自然数中前面 8 个素数 (2,3,5,7,11,13,17,19) 的平方根的小数部分, 并且取前面的 32 位。

> $\sqrt{2}$ 小数部分约为 $0.4142135623730950 48 \approx 6 * 16^{-1} + a * 16^{-2} + 0 * 16^{-3} + \ldots$，于是质数 2 的平方根的小数部分取前 32 位就对应 0x6a09e667

如此类推, 初始哈希值 $H^{(0)}$ 由以下 8 个 32 位的哈希初值构成：

$$
\begin{aligned}
H_1^{(0)} &= 6a09e667 \\
H_2^{(0)} &= bb67ae85 \\
H_3^{(0)} &= 3c6ef372 \\
H_4^{(0)} &= a54ff53a \\
H_5^{(0)} &= 510e527f \\
H_6^{(0)} &= 9b05688c \\
H_7^{(0)} &= 1f83d9ab \\
H_8^{(0)} &= 5be0cd19
\end{aligned}
$$

SHA256 算法当中还使用到 64 个常数，取自自然数中前面 64 个素数的立方根的小数部分的前 32 位, 如果用 16 进制表示, 则相应的常数序列如下：

```
428a2f98 71374491 b5c0fbcf e9b5dba5
3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3
72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc
2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13
650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3
d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5
391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208
90befffa a4506ceb bef9a3f7 c67178f2
```

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：

$$
l + 1 + k \equiv 448 \mod 512
$$

以消息"abc"为例显示补位的过程：a,b,c 对应的 **ASCII 码**和二进制编码分别如下：

原始字符	ASCII码	二进制编码
a	97	01100001
b	98	01100010
c	99	01100011

因此, 原始信息"abc"的二进制编码为:01100001 01100010 01100011, 第一步补位, 首先在消息末尾补上一位"1", 结果为：01100001 01100010 01100011 1；然后进行第二步的补位, 因为 $l = 24$, 可以得到 $k = 423$, 在第一步补位后的消息后面再补423个"0", 结果如下：

$$
01100001011000100110001110\underbrace{0\ldots0}_{423}
$$

最后还需要在上述字节串后面继续进行补码, 这个时候补的是原消息"abc"的二进制长度 $l = 24$ 的64位二进制表示形式, 补完以后的结果如下：

$$
01100001011000100110001110\underbrace{0\ldots0}_{423}\underbrace{0\ldots011000}_{64}
$$

最终补完以后的消息二进制位数长度是 512 的倍数。

将补码处理后的消息以 512 位为单位分块为 $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$，其中第 i 个消息块的前 32 位表示为 $M_0^{(i)}$，以此类推，最后 32 位的消息块可表示为 $M_{15}^{(i)}$。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。

摘要计算主循环

哈希计算算法如下：

$for\, i = 1 \rightarrow N$（$N =$ 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：

$$
\begin{aligned}
a &\leftarrow H_1^{(i-1)} \\
b &\leftarrow H_2^{(i-1)} \\
&\vdots \\
h &\leftarrow H_8^{(i-1)}
\end{aligned}
$$

应用 SHA256 压缩函数来更新 $a, b, \cdots, h$：$for\, j = 0 \rightarrow 63$，计算：

$$
\begin{aligned}
T_1 &\leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j \\
T_2 &\leftarrow \Sigma_0(a) + M_{aj}(a, b, c) \\
h &\leftarrow g \\
g &\leftarrow f \\
f &\leftarrow e \\
e &\leftarrow d + T_1 \\
d &\leftarrow c \\
c &\leftarrow b \\
b &\leftarrow a \\
a &\leftarrow T_1 + T_2
\end{aligned}
$$

计算第 i 个中间哈希值 $H^{(i)}$：

$$
\begin{aligned}
H_1^{(i)} &\leftarrow a + H_1^{(i-1)} \\
H_2^{(i)} &\leftarrow b + H_2^{(i-1)} \\
&\vdots \\
H_8^{(i)} &\leftarrow h + H_8^{(i-1)}
\end{aligned}
$$

逻辑函数定义

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

$$
\begin{aligned}
Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
M_{aj}(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= S^2(x) \oplus S^{13}(x) \oplus S^{22}(x) \\
\Sigma_1(x) &= S^6(x) \oplus S^{11}(x) \oplus S^{25}(x) \\
\sigma_0(x) &= S^7(x) \oplus S^{18}(x) \oplus R^3(x) \\
\sigma_1(x) &= S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)
\end{aligned}
$$

扩展消息块 $W_0, W_1, \cdots, W_{63}$ 通过以下方式计算：

$$W_j = M_j^{(i)} \text{ for } j = 0, 1, \ldots, 15$$
$$\text{For } j = 16 \to 63 \quad W_j \leftarrow \sigma_1\left(W_{j-2}\right) + W_{j-7} + \sigma_0\left(W_{j-15}\right) + W_{j-16}$$

SHA 伪代码

```
Note 1: All variables are 32 bit unsigned integers and addition is calculated modulo 232
Note 2: For each round, there is one round constant k[i] and one entry in the message schedule array w[i], 0 ≤ i ≤ 63
Note 3: The compression function uses 8 working variables, a through h
Note 4: Big-endian convention is used when expressing the constants in this pseudocode,
    and when parsing message block data from bytes to words, for example,
    the first word of the input message "abc" after padding is 0x61626380

Initialize hash values:
(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):
k[0..63] :=
   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

Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a multiple of 512
append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    create a 64-entry message schedule array w[0..63] of 32-bit words
    (The initial values in w[0..63] don't matter, so many implementations zero them here)
    copy chunk into first 16 words w[0..15] of the message schedule array

    Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift  3)
        s1 := (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1

    Initialize working variables to current hash value:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    Compression function main loop:
    for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
```

```
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        temp2 := S0 + maj

        h := g
        g := f
        f := e
        e := d + temp1
        d := c
        c := b
        b := a
        a := temp1 + temp2

    Add the compressed chunk to the current hash value:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

SHA256 代码实现

下面是基于上述伪代码用`Python`语言对 SHA256 进行的实现。

```
import numpy as np
import platform
import time

def preprocess(message):
    msg_list = list(message)
    msgLen = len(msg_list)*8

    result = bytes(message, "ascii")
    result += b"\x80"

    if len(msg_list)%64 < 56:
        suffix = b"\x00" * (55-len(msg_list)%64)
    else:
        suffix = b"\x00" * (64+55-len(msg_list)%64)
    result += suffix

    result += msgLen.to_bytes(8, "big")
    return result

# 32个字节右移n位
def R(x, n):
    return (x >> n)

# 32个字节循环右移n位
def S(x, n):
    return ((x >> n) | (x << (32-n))) & (2**32-1)

# 逻辑函数的定义
def Ch(x, y ,z):
    return ((x & y) ^ (~x & z))

def Maj(x, y ,z):
    return ((x & y) ^ (x & z) ^ (y & z))

def Sigma0(x):
    return (S(x, 2) ^ S(x, 13) ^ S(x, 22))

def Sigma1(x):
    return (S(x, 6) ^ S(x, 11) ^ S(x, 25))

def sigma0(x):
```

```
    return (S(x, 7) ^ S(x, 18) ^ R(x, 3))

def sigma1(x):
    return (S(x, 17) ^ S(x, 19) ^ R(x, 10))

def compress(Kj, Wj, a, b, c, d, e, f, g, h):

    T1 = h + Sigma1(e) + Ch(e, f ,g) + Kj + Wj
    T2 = Sigma0(a) + Maj(a, b , c)
    h = g
    g = f
    f = e
    e = (d + T1) & (2**32-1)
    d = c
    c = b
    b = a
    a = (T1 + T2 ) & (2**32-1)
    return a, b, c, d, e, f, g, h

def SHA256(message):
    # 初始哈希值
    H = (0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
         0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19)
    # 计算过程当中用到的常数
    K = np.array([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])

    message = preprocess(message)

    for i in range(0, len(message), 64):
        S = message[i: i + 64]
        W = [int.from_bytes(S[e: e + 4], "big") for e in range(0, 64, 4)] + ([0] * 48)

        for j in range(16, 64):
            W[j] = (sigma1(W[j-2]) + W[j-7] + sigma0(W[j-15]) + W[j-16]) & ((2**32)-1)

        a, b, c, d, e, f, g, h = H

        for j in range(64):
            a, b, c, d, e, f, g, h = compress(K[j], W[j], a, b, c, d, e, f, g, h)

    H = [d.to_bytes(4, "big") for d in [int((x + y) & (2**32-1)) for x, y in zip(H, (a, b, c, d, e, f, g, h))]]

    return "".join(format(h, "02x") for h in b"".join(H))

if __name__ == '__main__':
    s = ""
    zero_cnt = 30
    zero_vec = [0]*zero_cnt
    zero_str = "".join([str(zero_vec[i]) for i in range(zero_cnt)])

    i = 8500000000
    time_start = time.perf_counter()
    while (True):
        result = SHA256(s+str(i))
        bin_str = "".join(['{:04b}'.format(int(result[i], 16)).replace("0b", "") for i in range(len(result))])
        if (i % 10000 == 0):
            time_now = time.perf_counter()

            print("i = "+str(i)+ " :", time_now-time_start, 's')
        if (bin_str[0:zero_cnt] == zero_str):
            time_now = time.perf_counter()
```

```
            print(s+str(i))
            print(result)
            print("Time used: ", time_now-time_start, 's')
            break
        else:
            i += 1
```

SHA256 实验结果

说明 SHA256 找到一个前30、31、32位（二进制）为0的数花了多少时间。

理论上0的位数越多，这样的字符串组合越少，也越难找到，花费的时间越多，但也和哈希字符串的生成方式有关。

在本次实验中选择从字符串 `8500000000` 开始遍历，每次通过+1操作来生成新的字符串，第一次找到的便是前32位为0的数，同时满足了前30、31、32位（二进制）为0的要求，实验结果如下图所示：

```
i = 8517720000 : 9859.34952674713 s
i = 8517730000 : 9864.89572622208 s
i = 8517740000 : 9870.476049019024 s
i = 8517750000 : 9876.021223523188 s
i = 8517760000 : 9881.570209198166 s
i = 8517770000 : 9887.116986797191 s
i = 8517780000 : 9892.670355597045 s
i = 8517790000 : 9898.226262270939 s
i = 8517800000 : 9903.775348697789 s
i = 8517810000 : 9909.323293984868 s
8517810597
00000000b576e5557f55542399e8b800b498bae023b116ab9ad43b96d610a86f
Time used:  9909.654537504073 s
```

找到一个前32位（二进制）为0的数花了使用了9909秒即2小时45分钟09秒。

后来也尝试去寻找恰好前30、31位为0的数，从 `8517810598` 开始继续向后遍历，但受限于 Python 程序的运行速度较慢，最终未能在有限的时间内找到恰好前30、31位为0的数。