创新创业实验 project4

一、核心类 SM3 的设计与实现

SM3 类封装了算法的全部逻辑，遵循“流式处理”模式（支持分块输入消息），核心成员包括：

- 状态变量（8 个 32 位寄存器，存储当前哈希值）；
- 缓存区（存储未处理的消息字节，凑齐 512 位块后触发压缩）；
- 消息总长度（64 位，用于最终填充）。

1. 初始化

将哈希状态初始化为国家标准规定的初始值 IV（8 个 32 位常量），重置缓存区和长度计数器，为处理新消息做准备。

```
const uint32_t SM3::IV[8] = {
    0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
    0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e
};
```

2. SM3 类成员与构造函数

- state[8]：存储当前哈希值，对应算法中的 8 个 32 位寄存器（A、B、C、D、E、F、G、H），所有压缩操作均围绕这些寄存器展开。
- buffer[64]：临时存储未处理的消息字节，因为 SM3 按 512bit（64 字节）块处理消息，不足一块的部分需暂存。
- 构造函数的作用是“重置”算法状态，确保每次计算哈希都从标准 IV 开始，不受之前计算的影响。

```
SM3::SM3() {
    memcpy(state, IV, sizeof(IV));
    totalBits = 0;
    bufferSize = 0;
    memset(buffer, 0, sizeof(buffer));
}
```

二、核心置换函数（P0、P1）

1. 置换函数 P0

- P0 是 SM3 定义的**非线性置换函数**，通过异或（^）和循环左移组合，对输入 x 进行位重排，增强算法的“混淆性”（使输入的微小变化导致输出的显著变化）。
- 应用场景：在压缩函数的 64 轮迭代中，用于更新寄存器 E（E = P0(TT2)）

```
uint32_t SM3::P0(uint32_t x) {
    return x ^ rotateLeft(x, 9) ^ rotateLeft(x, 17);
}
```

2. 置换函数 P1

- P1 与 P0 类似，但左移位数不同（15 和 23），进一步增强非线性。
- 应用场景：在消息扩展阶段生成 W[16..67]（扩展消息字），确保扩展后的消息具有足够的随机性。

```
uint32_t SM3::P1(uint32_t x) {
    return x ^ rotateLeft(x, 15) ^ rotateLeft(x, 23);
}
```

三、布尔函数（FF、GG）

1. 布尔函数 FF

- FF 用于更新寄存器 A（TT1 = FF(...)），输入为当前轮次 j 和寄存器 A、B、C 的值。
- 前 16 轮使用简单异或（x^y^z）：计算高效，适合初始扩散。
- 后 48 轮使用复杂与或运算（(x&y)|(x&z)|(y&z)）：增强非线性，提升抗攻击能力（避免被线性分析破解）。

```
uint32_t SM3::FF(uint32_t x, uint32_t y, uint32_t z, int j) {
    if (j <= 15) {
        return x ^ y ^ z;
    }
    else {
        return (x & y) | (x & z) | (y & z);
    }
}
```

2. 布尔函数 GG

- GG 用于更新寄存器 E（TT2 = GG(...)），输入为当前轮次 j 和寄存器 E、F、G 的值。
- 前 16 轮同 FF，使用异或；后 48 轮使用选择运算（(x&y)|(~x&z)）：等价于“若 x 为 1 则选 y，否则选 z”，进一步增强非线性。

```
uint32_t SM3::GG(uint32_t x, uint32_t y, uint32_t z, int j) {
    if (j <= 15) {
        return x ^ y ^ z;
    }
    else {
        return (x & y) | (~x & z);
    }
}
```

四、消息处理（update 方法）

- update 负责接收输入数据，按 512bit 块拆分并触发压缩，支持“流式处理”。SM3 处理消息的基本单位是 **512bit 块**，update 的作用是将输入数据“攒”成块：
    - 累计消息总长度（total_len），用于最终填充阶段（需在消息末尾附加原始长度）。
    - 用 memcpy 将输入数据填充到 buffer 中，若 buffer 满 64 字节（512bit），则调用 compress 处理该块，随后重置 buffer_pos。
- 流式处理优势：无需一次性加载大消息到内存（如 1GB 文件可分 1MB 块多次调用 update），降低内存占用。

```
void SM3::update(const uint8_t* data, size_t len) {
    totalBits += len * 8;  // 更新总长度(bit)

    // 处理缓存区已有数据
    if (bufferSize > 0) {
        size_t fill = 64 - bufferSize;
        if (len <= fill) {
            memcpy(buffer + bufferSize, data, len);
            bufferSize += len;
            return;  // 缓存区未满，无需压缩
        }
        else {
            memcpy(buffer + bufferSize, data, fill);
            compress(buffer);  // 处理满的块
            data += fill;
            len -= fill;
            bufferSize = 0;
        }
    }
```

五、压缩函数

compress 是 SM3 最核心的函数，对单个 512bit 消息块进行处理，通过**消息扩展**和**64 轮迭代**更新哈希状态 state。

1. 消息扩展（生成 W 和 W'数组）

- 消息扩展将 512bit 输入块转换为 132 个 32 位字（68 个 W + 64 个 W'），为后续轮迭代提供输入，增强信息扩散。
- 输入 block 是 64 字节（512bit）的消息块，按 4 字节一组拆分为 16 个 32 位字（W[0]到 W[15]）。
- 转换为**大端序**（高位字节存于高地址）：如 block[4i]是第 i 组的最高位字节，通过移位（<<24）放入 32 位字的最高位。

```
void SM3::compress(const uint8_t* block) {
    // 1. 消息扩展：将512bit块扩展为68个32bit字W和64个32bit字W'
    uint32_t W[68], W1[64];

    // 优化：一次性完成字节序转换(小端转大端)
    for (int i = 0; i < 16; ++i) {
        W[i] = (uint32_t)block[4 * i] << 24 |
            (uint32_t)block[4 * i + 1] << 16 |
            (uint32_t)block[4 * i + 2] << 8 |
            (uint32_t)block[4 * i + 3];
    }
```

- 从 W[16]到 W[67]共 52 个字，通过“前序字的异或 + 置换”生成，确保每个新字都依赖之前的多个字，增强扩散性。
- 公式解析：W[i] = P1(前序字混合) ^ 左移字 ^ 前序字，多层混合确保输入信息被充分打乱。

```
for (int j = 16; j < 68; ++j) {
    W[j] = P1(W[j - 16] ^ W[j - 9] ^ rotateLeft(W[j - 3], 15)) ^
        rotateLeft(W[j - 13], 7) ^ W[j - 6];
}
```

W'是 W 的衍生数组，通过相邻字异或生成，为轮迭代提供额外的输入组合，进一步增强混淆性。

```
for (int j = 0; j < 64; ++j) {
    W1[j] = W[j] ^ W[j + 4];
}
```

2. 64 轮迭代

64 轮迭代是压缩函数的核心，每轮通过复杂运算更新 8 个状态寄存器（A~H），最终将消息块的信息“融入”哈希状态。

将当前哈希状态（state）备份到局部变量 A~H，避免迭代过程中覆盖原始值（最终需与迭代结果异或）。

```
uint32_t A = state[0], B = state[1], C = state[2], D = state[3];
uint32_t E = state[4], F = state[5], G = state[6], H = state[7];
```

- **轮常量 T**:
    - 前 16 轮使用 0x79cc4519，后 48 轮使用 0x7a879d8a，是算法规定的固定常量，用于打破轮间对称性，增强安全性。
- **中间变量计算**:
    - SS1：融合 A 的左移、E 的值和轮常量 T 的左移，再左移 7 位，实现跨寄存器的信息混合。
    - SS2：SS1 与 A 的左移异或，进一步增强混淆。
    - TT1：结合 FF 函数结果、D 的值、SS2 和 W1[j]，作为更新 A 的中间值。
    - TT2：结合 GG 函数结果、H 的值、SS1 和 W[j]，作为更新 E 的中间值。
- **状态更新顺序**:
  按固定顺序更新 A~H（如 D = C、C = ROTL(B,9)等），确保每个寄存器的新值都依赖上一轮的多个寄存器，实现“链式扩散”。

```
for (int j = 0; j < 64; ++j) {
    // 优化：预计算常量移位值
    uint32_t Tj = T[j];
    uint32_t shiftT = rotateLeft(Tj, j);

    // 计算SS1和SS2
    uint32_t SS1 = rotateLeft(rotateLeft(A, 12) + E + shiftT, 7);
    uint32_t SS2 = SS1 ^ rotateLeft(A, 12);

    // 计算TT1和TT2
    uint32_t TT1 = FF(A, B, C, j) + D + SS2 + W1[j];
    uint32_t TT2 = GG(E, F, G, j) + H + SS1 + W[j];

    // 更新寄存器(优化：减少临时变量)
    D = C;
    C = rotateLeft(B, 9);
    B = A;
    A = TT1;
    H = G;
    G = rotateLeft(F, 19);
    F = E;
    E = P0(TT2);
}
```

64 轮迭代后，将局部变量 A~H（迭代结果）与原始状态 state 异或，形成新的哈希状态，确保当前消息块的信息被“累加”到哈希值中。

```
state[0] ^= A;
state[1] ^= B;
state[2] ^= C;
state[3] ^= D;
state[4] ^= E;
state[5] ^= F;
state[6] ^= G;
state[7] ^= H;
```

六、最终填充与结果生成

final 处理缓存区中未完成的消息，按标准填充规则补全为 512bit 块，最后生成 256bit 哈希值。

1.先补 1 个 0x80 字节（即二进制 10000000），标记消息正文结束，这是所有哈希算法的通用设计（区分不同长度但前缀相同的消息）。

```
void SM3::final(uint8_t* hash) {
    // 1. 填充消息
    buffer[bufferSize++] = 0x80;  // 添加1个"1"比特
```

2.SM3 要求填充后总长度为 512bit 的整数倍，且最后 8 字节必须存储原始消息长度（64bit）。若当前 buffer_pos 超过 56（剩余空间 < 8 字节），需先补 0 至 64 字节并压缩，再重新开始填充。

```
if (bufferSize > 56) {
    memset(buffer + bufferSize, 0, 64 - bufferSize);
    compress(buffer);
    bufferSize = 0;
}
```

3. （1）补 0 至 56 字节（使缓存区前 56 字节 + 最后 8 字节长度 = 64 字节）。

```
memset(buffer + bufferSize, 0, 56 - bufferSize);
bufferSize = 56;
```

（2）最后 8 字节存储 total_len（消息原始长度，单位 bit），按大端序排列（高位字节在前）。

```
uint64_t bits = totalBits;
for (int i = 0; i < 8; ++i) {
    buffer[bufferSize++] = (uint8_t)(bits >> (8 * (7 - i)));
}
```

（3）压缩最后一个块，完成所有消息处理。

```
compress(buffer);
```

```
    static const char* hex = "0123456789abcdef";
    std::string res;
    res.reserve(64);
    for (uint8_t b : hash) {
        res += hex[b >> 4];
        res += hex[b & 0x0f];
    }
    return res;
```

- **结果生成**:

1. 将 8 个 32 位状态寄存器（state[0]到 state[7]）按大端序转换为 32 字节数组（hash[32]）。

2. 调用 toHexString 将字节数组转换为 64 字符的 16 进制字符串（便于展示和传输）。