SM3 算法原理：

算法整体结构：

SM3 的计算过程分为消息预处理、消息扩展和压缩函数三个阶段，核心操作包括布尔函数、置换函数和状态更新。

1.布尔函数：

$$FF_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z & j = 0,1,...,15 \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & j = 16,17,...,63 \end{cases}$$

$$GG_j(X,Y,Z) = \begin{cases} X \oplus Y \oplus Z & j = 0,1,...,15 \\ (X \wedge Y) \vee ((\neg X) \wedge Z) & j = 16,17,...,63 \end{cases}$$

2.置换函数：

用于扩散状态信息，增强算法的雪崩效应：

P0 函数（用于压缩函数输出）：

$$P_0(X) = X \oplus (X \lll 9) \oplus (X \lll 17)$$

P1 函数（用于消息扩展）：

$$P_1(X) = X \oplus (X \lll 15) \oplus (X \lll 23)$$

算法流程：

1.消息预处理（填充）：

为使消息长度为 512 位的整数倍，对消息进行填充：附加一个比特“1”；附加 k 个比特“0”，使得填充后消息长度模 512 等于 448；

附加 64 位消息原始长度（以比特为单位）。填充后消息长度为 L + 1 + k + 64 = 512m（m 为正整数）。

2.消息扩展：

将每个 512 位消息块扩展为 132 个字（32 位），分为两组：

第一组：

$$W_j = P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \lll 15)) \oplus (W_{j-13} \lll 7) \oplus W_{j-6}$$

第二组：

$$W'_j = W_j \oplus W_{j+4} \ \ (j = 0 \sim 63)$$

3.压缩函数

4.最终输出：

所有消息块处理完成后，将最终状态的 8 个 32 位字按顺序拼接，得到 256 位哈希值。

重点部分代码实现：

1.SM3 实现：

消息填充（_padding）：

```
def _padding(self, message: bytes) -> bytes:
length = len(message) * 8  # 消息长度（比特）
message += b'\x80'  # 附加 0x80（二进制 10000000）
# 填充 0 至长度模 512=448
while (len(message) * 8) % 512 != 448:
```

```
        message += b'\x00'
    # 附加原始长度（64位，大端序）
    message += length.to_bytes(8, byteorder='big')
    return message
```

消息扩展（_message_extension）：

```
def _message_extension(self, b: bytes) -> tuple[list[int], list[int]]:
    # 拆分消息块为16个32位字
    w = [int.from_bytes(b[i:i+4], byteorder='big') for i in range(0, 64, 4)]
    # 扩展为68个字
    for i in range(16, 68):
        w.append(p1(w[i-16] ^ w[i-9] ^ rotl(w[i-3], 15)) ^
                 rotl(w[i-13], 7) ^ w[i-6])
    # 生成W'
    w1 = [w[i] ^ w[i+4] for i in range(64)]
    return w, w1
```

压缩函数（_compress）：

```
def _compress(self, v: List[int], b: bytes) -> List[int]:
    a, b_val, c, d, e, f, g, h = v  # b_val避免与参数b冲突
    w, w1 = self._message_extension(b)
```

```
for j in range(64):
    # 计算临时变量
    tt1 = (rotl(a, 12) + e + rotl(SM3_T[j], j)) % 0x100000000
    tt1 = rotl(tt1, 7)
    tt2 = tt1 ^ rotl(a, 12)
    # 选择布尔函数
    if j < 16:
        f_func = ff0(b_val, c, d)
        g_func = gg0(e, f, g)
    else:
        f_func = ff1(b_val, c, d)
        g_func = gg1(e, f, g)
    # 更新状态变量
    t = (h + g_func + rotl(e, 12) + w1[j] + tt2) % 0x100000000
    h_new = (f_func + tt1 + t) % 0x100000000
    # 状态轮转
    a, b_val, c, d, e, f, g, h = h_new, a, rotl(b_val, 9), c, p0(t), e, rotl(f, 19), g
# 与初始向量异或输出
return [(a ^ v[0]) % 0x100000000, (b_val ^ v[1]) %
```

0x100000000,

(c ^ v[2]) % 0x100000000, (d ^ v[3]) % 0x100000000,

(e ^ v[4]) % 0x100000000, (f ^ v[5]) % 0x100000000,

(g ^ v[6]) % 0x100000000, (h ^ v[7]) % 0x100000000]

哈希主函数（hash)：

```
def hash(self, message: bytes) -> bytes:
padded = self._padding(message)  # 填充消息
state = self.iv.copy()  # 初始化状态
# 分块处理
for i in range(0, len(padded), 64):
    block = padded[i:i+64]
    state = self._compress(state, block)
# 拼接状态为256位哈希值
return b''.join([x.to_bytes(4, byteorder='big') for x in state])
```

2.优化 1：(SM3Optimized1)：预计算常量

预计算轮常量的旋转结果，避免 64 轮迭代中重复计算：

```
class SM3Optimized1(SM3Base):
def __init__(self):
```

```
        super().__init__()
        # 预计算 rotl(SM3_T[j], j)，减少轮迭代中的重复计算
        self.rotated_T = [rotl(SM3_T[j], j) for j in range(64)]
    def _compress(self, v: List[int], b: bytes) -> List[int]:
        # 复用基础版逻辑，仅替换轮常量为预计算值
        tt1 = (rotl(a, 12) + e + self.rotated_T[j]) % 0x100000000  # 优化点
```

3.优化 2（SM3Optimized2）：向量化消息扩展

使用 numpy 的向量化操作加速消息扩展，减少 Python 循环开销：

```
class SM3Optimized2(SM3Optimized1):
    def _message_extension(self, b: bytes) -> tuple[Any, list[int]]:
        w = np.zeros(68, dtype=np.uint32)  # 使用 numpy 数组存储
        for i in range(16):
            w[i] = int.from_bytes(b[i*4:(i+1)*4], byteorder='big')
        # 向量化扩展计算（numpy 操作比 Python 循环更快）
        for i in range(16, 68):
            w[i] = p1((w[i-16] ^ w[i-9] ^ rotl(w[i-3], 15)) & 0xFFFFFFFF) ^ \
```

```
            (rotl(w[i-13], 7) & 0xFFFFFFFF) ^ w[i-6]
        w[i] &= 0xFFFFFFFF
    # 生成W'并转换为Python列表
    w1 = np.zeros(64, dtype=np.uint32)
    for i in range(64):
        w1[i] = w[i] ^ w[i+4]
    return w.tolist(), [int(x) for x in w1.tolist()]
```

4.优化3（SM3Optimized3）：块处理优化

针对大消息优化块处理流程，减少内存分配和拷贝：

```
class SM3Optimized3(SM3Optimized2):
  def hash(self, message: bytes) -> bytes:
    length = len(message)
    block_count = (length + 8 + 63) // 64  # 预计算总块数
    state = self.iv.copy()

    # 处理完整块（避免重复切片和内存分配）
    ptr = 0
    while ptr + 64 <= length:
        block = message[ptr:ptr+64]
        state = self._compress(state, block)
        ptr += 64
```

        # 处理剩余部分和填充（仅对剩余数据填充，减少大消息的内存占用）

```
        remaining = message[ptr:]
        padded = self._padding(remaining)
        for i in range(0, len(padded), 64):
            block = padded[i:i+64]
            state = self._compress(state, block)
        return b''.join([x.to_bytes(4, byteorder='big') for x in state])
```

5.长度扩展攻击实现：

```
def sm3_length_extension_attack(original_hash: bytes, original_length: int, append_data: bytes, sm3_impl=SM3Base) -> Tuple[bytes, bytes]:
    # 1.将原始哈希转换为状态向量（压缩函数的输出即下一轮输入）
    state = [int.from_bytes(original_hash[i:i+4], 'big') for i in range(0, 32, 4)]
    # 2.计算原始消息的填充（不包含原始消息本身）
    original_bits = original_length * 8
    pad_length = 64 - (original_length % 64)
    if pad_length < 9:  # 确保至少1字节0x80 + 8字节长度
        pad_length += 64
    padding = b'\x80' + b'\x00'*(pad_length-9) +
```

original_bits.to_bytes(8, 'big')

```
    # 3.构造新消息后缀：填充 + 附加数据
    new_message = padding + append_data
    # 4.以原始哈希为初始状态，继续处理新消息
    sm3 = sm3_impl()
    current_state = state
    for i in range(0, len(new_message), 64):
        block = new_message[i:i+64]
        if len(block) < 64:
            block += b'\x00'*(64 - len(block))
        current_state = sm3._compress(current_state, block)
    # 5.生成伪造哈希
    forged_hash = b''.join([x.to_bytes(4, 'big') for x in
current_state])
    return new_message, forged_hash
```

Merkle 树实现

1. 树结构与哈希计算

```
   Def __init__(self, leaves: List[bytes],
sm3_impl=SM3Optimized3):
        self.sm3 = sm3_impl()
        self.leaves = leaves
```

    # 叶子节点为 32 字节哈希
        self.tree = self._build_tree() # 二维列表：tree[0]为叶子，tree[1]为父节点，
        self.root = self.tree[0][0] if self.tree else b''

    def _hash_leaf(self, data: bytes) -> bytes:
        # 叶子节点哈希：前缀 0x00 + 数据
        return self.sm3.hash(b'\x00' + data)
    def _hash_internal(self, left: bytes, right: bytes) -> bytes:
        # 内部节点哈希：前缀 0x01 + 左哈希 + 右哈希
        return self.sm3.hash(b'\x01' + left + right)

2.存在性证明

思路：通过提供从叶子到根的路径上的兄弟节点哈希

```
def get_proof(self, index: int) -> List[Tuple[bytes, bool]]:
    proof = []
    current_index = index
    current_level = 0
    while current_level < len(self.tree) - 1:
        # 记录兄弟节点哈希及位置（左/右）
        if current_index % 2 == 0:
```

```
            # 左节点，兄弟为右
                sibling_index = current_index + 1
                is_right = True
                if sibling_index >= len(self.tree[current_level]):
                    sibling_index = current_index
                      # 奇数节点处理
            else:  # 右节点，兄弟为左
                sibling_index = current_index - 1
                is_right = False

            proof.append((self.tree[current_level][sibling_index], is_right))
            # 上移至父节点
            current_index = current_index // 2
            current_level += 1
        return proof
```

验证时通过兄弟节点哈希逐步计算根哈希，与树的根比对：

```
    def verify_proof(self, leaf: bytes, proof: List[Tuple[bytes, bool]], root: bytes) -> bool:
        current_hash = self._hash_leaf(leaf)
        for (hash_val, is_right) in proof:
```

```
        if is_right:
            current_hash = self._hash_internal(current_hash, hash_val)
        else:
            current_hash = self._hash_internal(hash_val, current_hash)
    return current_hash == root
```

3.不存在性证明

思路：通过验证该位置左右相邻的存在节点及其路径

```
   def get_non_existence_proof(self, index: int) -> Tuple[List[Tuple[bytes, bool]], bytes, List[Tuple[bytes, bool]], bytes]:
    left_idx = index - 1  # 左侧最近存在节点
    while left_idx >= 0 and left_idx >= len(self.leaves):
        left_idx -= 1
    right_idx = index + 1  # 右侧最近存在节点
    while right_idx < len(self.leaves) and right_idx >= len(self.leaves):
        right_idx += 1
    # 返回左右节点的证明及哈希
    left_proof = self.get_proof(left_idx) if left_idx >=0 else []
```

```
    left_hash = self._hash_leaf(self.leaves[left_idx]) if left_idx >=0 else b''
    right_proof = self.get_proof(right_idx) if right_idx < len(self.leaves) else []
    right_hash = self._hash_leaf(self.leaves[right_idx]) if right_idx < len(self.leaves) else b''
    return left_proof, left_hash, right_proof, right_hash
```

参考文献：

1. GB/T 32905-2016，《信息安全技术 SM3 密码杂凑算法》

2. IETF RFC 6962，《Certificate Transparency》

3. 王小云等，《密码学中的杂凑函数》，科学出版社，2011

4. NIST FIPS 180-4，《Secure Hash Standard》