表格中的运行时间都是对同一明密文的加密时间

1.无符号整数类型加上位运算

我修改了数据类型能够得到较为显著的加速效果，那么再进一步地选择更为合适的数据类型，理论上是能够获得优化的，这里我选择的是无符号整型类型，它读取的速度要快于 string 类型和 char 类型的读取，且在运算上，它能直接在内存处理数据，写起来也方便快捷些（位运算可以用 | & ^来实现）。

之后涉及到单线程简单 SM4 算法都是基本算法

具体实现如下：

```
//解密与加密类似
//运行结果
===加密===
明文为:1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
密钥为:1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
所得密文为：68 1e df 34 d2 6 96 5e 86 b3 e9 4f 53 6e 42 46
总时间为(毫秒/ms)：7
===解密===
密文为:68 1e df 34 d2 6 96 5e 86 b3 e9 4f 53 6e 42 46
密钥为:1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
所得明文为：1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
总时间为(毫秒/ms)：4

uint32_t RKey[32]; //子密钥
uint32_t num = 0xffffffff;
// T 置换
uint32_t T(uint32_t m)
{
uint8_t s[4];
uint32_t res = 0;
for (int i = 0; i < 4; i++)
{s[i] = m >> (24 - i * 8);
s[i] = Sbox[s[i] >> 4][s[i] & 0x0f];
res |= s[i] << (24 - i * 8);
```

```
}
return res ^ (((res << 2) | (res >> 30)) & num) ^ (((res
<< 10) | (res >> 22)) & num) ^ (((res << 18) | (res >> 14))
& num) ^ (((res << 24) | (res >> 8)) & num);
}
// T' 置换
uint32_t T1(uint32_t m)
{
uint8_t s[4];
uint32_t res = 0;
for (int i = 0; i < 4; i++)
{
s[i] = m >> (24 - i * 8);
s[i] = Sbox[s[i] >> 4][s[i] & 0x0f];
res |= s[i] << (24 - i * 8);
}
return res ^ (((res << 13) | (res >> 19)) & num) ^
(((res << 23) | (res >> 9)) & num);
}
// 子密钥产生
void RK(uint32_t Key[])
{
uint32_t k[36];
memset(k, 0, sizeof(k)); //填充 0
for (int i = 0; i < 4; i++)
{
k[i] = Key[i] ^ FK[i];
}
for (int i = 0; i < 32; i++)
{
k[i + 4] = k[i] ^ T1(k[i + 1] ^ k[i + 2] ^ k[i + 3]
^ CK[i]);
RKey[i] = k[i + 4];
}
}
//加密 void en_SM4(uint32_t Plain[4], uint32_t Secret[4])
{
uint32_t X[36];
for (int i = 0; i < 4; i++)
{
X[i] = Plain[i];
}
for (int i = 0; i < 32; i++)
{
```

```
X[i + 4] = X[i] ^ T(X[i + 1] ^ X[i + 2] ^ X[i + 3] ^
RKey[i]);
}
for (int i = 0; i < 4; i++)
{
Secret[i] = X[35 - i];
}
}
//解密
void de_SM4(uint32_t Secret[4], uint32_t de_Plain[4])
{
uint32_t X[36];
for (int i = 0; i < 4; i++)
{
X[i] = Secret[i];
}
for (int i = 0; i < 32; i++)
{
X[i + 4] = X[i] ^ T(X[i + 1] ^ X[i + 2] ^ X[i + 3] ^
RKey[31 - i]);//子密钥倒序
}
for (int i = 0; i < 4; i++)
{
de_Plain[i] = X[35 - i];
}
```

2.查表优化

查表实现是密码算法软件实现的最基本方法，其核心思想是将密码算法轮函数中尽可能多的变换操作制成表。SM4 中 S 盒操作为 x_0,x_1,x_2,x_3——> S(x_0),S(x_1),S(x_2),S(x_3)，其中 x_i 为 8bit 字。为了提升效率，可将 S 盒与后续的循环移位变换 L 并，即可定义 4 个 8bit——>32bit 查找表 T_i 这样可以节省后续的循环移位操作，大致操作如下：

1、通过移位取出 x_0,x_1,x_2,x_3

2、返回 T_0(x_0)⊕T_1(x_1)⊕T_2(x_2)⊕T_3(x_3)

具体操作如下：

```
//运行结果
明文为：123456789abcdeffedcba9876543210
加密结果为：681edf34d206965e86b3e94f536e4246
所用时间（毫秒）：0.0045
解密结果为：123456789abcdeffedcba9876543210
所用时间（毫秒）：0.0046
//4 个 T 表
//篇幅原因具体细节不予展示
static uint32_t Table0[256] = {...}
static uint32_t Table1[256] = {...}
static uint32_t Table2[256] = {...}
static uint32_t Table3[256] = {...}
/*
brief SM4 加解密运行时间大幅减少
```

3.多线程优化

因为 SM4 加解密过程采用了 32 轮迭代机制，这也就意味着没办法在 SM4 加密内部采用多线程优化（只有前一轮的轮迭代结束，后一轮的迭代才能开始），所以想用多线程优化 SM4 算法，只能在外部实现，对明密文的加解密采用多线程优化，具体操作如下：

```
26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
所用时间为(毫秒/ms)：0.0069
对上面的密文进行解密：
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
所用时间为(毫秒/ms)：0.0075
//外部多线程调用
void test(int times, uint32_t Plain[4], uint32_t Secret[4],
uint32_t de_Plain[4])
{
for (int i = 0; i < times; i++)
{
en_SM4(Plain, Secret);
de_SM4(Secret, de_Plain);
}
}
RK(Key); //产生子密钥
thread a(test, 125000, Plain, Secret, de_Plain);
thread b(test, 125000, Plain, Secret, de_Plain);
thread c(test, 125000, Plain, Secret, de_Plain);
```

4.循环展开优化

循环展开也就是将 SM4 中的 T 置换展开，具体实现如下:

```
thread d(test, 125000, Plain, Secret, de_Plain);
thread e(test, 125000, Plain, Secret, de_Plain);
thread f(test, 125000, Plain, Secret, de_Plain);
thread g(test, 125000, Plain, Secret, de_Plain);
thread h(test, 125000, Plain, Secret, de_Plain);
a.join(); b.join(); c.join(); d.join(); e.join();
f.join(); g.join(); h.join();


// T 置换
uint32_t T_(uint32_t m)
{
uint8_t s[4];
uint32_t res = 0;
s[0] = m >> 24; s[0] = Sbox[s[0] >> 4][s[0] & 0x0f]; res
|= s[0] << 24;
s[1] = m >> 16; s[1] = Sbox[s[1] >> 4][s[1] & 0x0f]; res
|= s[1] << 16;
s[2] = m >> 8; s[2] = Sbox[s[2] >> 4][s[2] & 0x0f]; res
|= s[2] << 8;
s[3] = m >> 0; s[3] = Sbox[s[3] >> 4][s[3] & 0x0f]; res
```

```
|= s[3] << 0;return res ^ (((res << 2) | (res >> 30)) & num)
^ (((res
<< 10) | (res >> 22)) & num) ^ (((res << 18) | (res >> 14))
& num) ^ (((res << 24) | (res >> 8)) & num);
}
// T' 置换
uint32_t T1_(uint32_t m)
{
uint8_t s[4];
uint32_t res = 0;
s[0] = m >> 24; s[0] = Sbox[s[0] >> 4][s[0] & 0x0f]; res
|= s[0] << 24;
s[1] = m >> 16; s[1] = Sbox[s[1] >> 4][s[1] & 0x0f]; res
|= s[1] << 16;
s[2] = m >> 8; s[2] = Sbox[s[2] >> 4][s[2] & 0x0f]; res
|= s[2] << 8;
s[3] = m >> 0; s[3] = Sbox[s[3] >> 4][s[3] & 0x0f]; res
|= s[3] << 0;
return res ^ (((res << 13) | (res >> 19)) & num) ^
(((res << 23) | (res >> 9)) & num);
}


//运行结果
加密结果为：681edf34d206965e86b3e94f536e4246 加密结果为：
123456789abcdeffedcba9876543210
通过 1_000_000 次加密解密 128 bits 的明文，模拟加密解密长度为
1_000_100 * 128 bits 长度的明文
加密解密 1_000_000 次 用时：6.762 seconds
加密解密 1_000_000 次(循环展开) 用时：4.548 seconds
加密解密 1_000_000 次(外部多线程) 用时：1.19 seconds
加密解密 1_000_000 次(循环展开 + 外部多线程) 用时：1.118
seconds
```