算法原理概述

DES 算法是一个加密算法，其采用的是块加密的方法，并采用 Electronic Code Book (ECB) 模式，将文件分为固定 64 位长度的分组，对每个分组分别进行加密，然后将各个分组的密文组合成为整个文件的密文。如果最后一个分组不够 64 位，那么使用不足的字节数补足剩余字节。

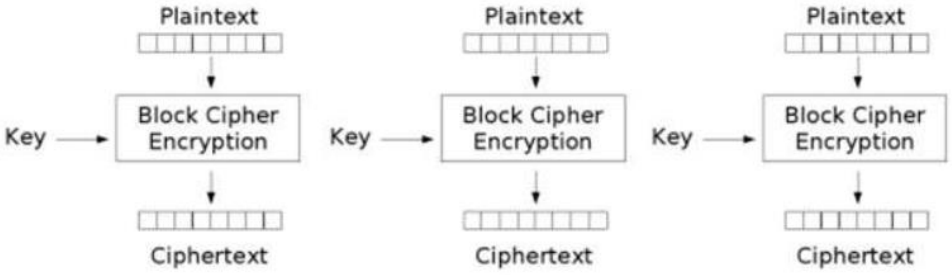


图 1 Electronic Code Book (ECB) Mode encryption

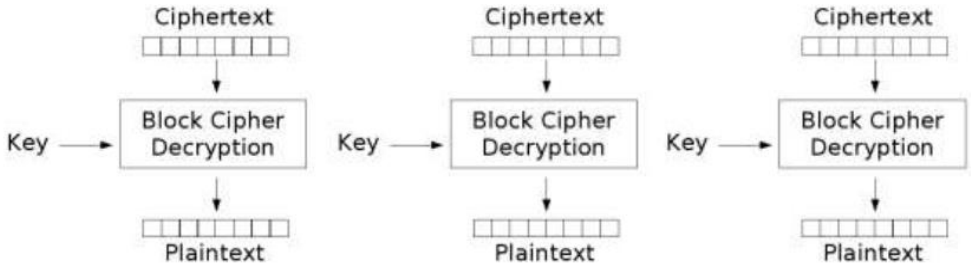


图 2 Electronic Code Book (ECB) Mode decryption

DES 算法采用 Feistel 结构，对于每一组 64 的输入，首先对其进行 IP 置换，然后使用 16 轮迭代进一步混淆，最后再使用 IP 逆置换，即可生成结果。

在迭代中，加密过程按照 1-16 的顺序使用秘钥 K 生成的 16 个 48 位的子秘钥进行该 16 次迭代，解密过程则反过来，使用 16-1 的顺序进行迭代。在每次迭代中，将输入的一个 64 位的串的右半部分作为下一次迭代的 64 位串的左半部

分，而下次迭代的右半部分由本次的左半部分和 Feistel 轮函数的结果做异或得出。该轮函数以本次输入的右半部分和本次迭代对应的子秘钥作为输入，输出一个 32 位的串。

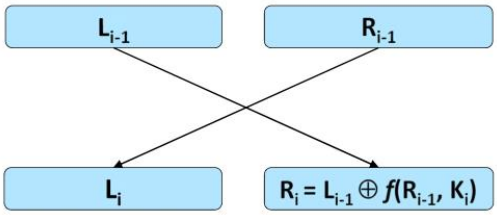


图 3 16 轮迭代

因此我们可以将此过程逆转以得到解密步骤，由于 IP 置换和 IP 逆置换为互逆的过程。因此我们将加密的最后一步逆转，即进行一次 IP 置换，得到加密过程中 16 轮迭代后的结果。又根据迭代过程的结构，反过来进行 16 轮迭代。再进行 IP 逆置换，即可得到原本的明文。

总体结构及模块分解

总体来说实现位一个 DES 类

类内使用 **bitset 类**来进行 bit 级别的操作和访问。

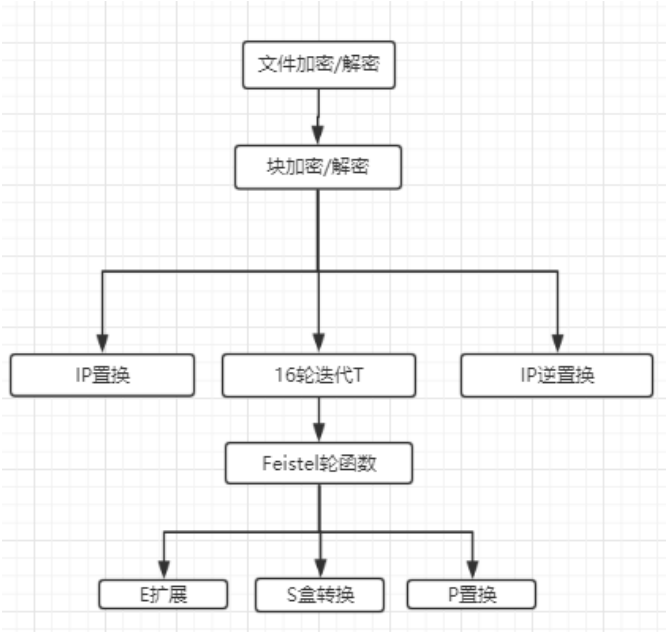
暴露接口如图所示，分别为构造函数（以秘钥 K 作为参数），文件加密、文件解密、加密 1 个 64 位的块，解密 1 个 64 位的块。

在构造函数中，要使用生成秘钥 K 生成 16 个 48 位子秘钥并保存。

```
public:
    DES(bitset<64> K_) {

    /**
    void encryptFile(const char* plaintextFileName = "plaintext.txt",
                     const char* cipherFileName = "cipher.txt") {

    /**
    void decryptFile(const char* cipherFileName = "cipher.txt",
                     const char* decryptionFileName = "decryptionResult.txt") {

    /**
    bitset<64> encrypt64(bitset<64> plaintext) {

    /**
    bitset<64> decrypt64(bitset<64> cipher) {
```

按照 top-down 的思想，将完成文件加密解密过程按照如下图的依赖关系拆

分为单独的子函数。其中，每个子函数都是很小的一个独立部分。



在块加密解密部分之后的部分，都完成在 **private** 声明中。

除此以外，还有一些辅助性的函数

```
// 一些我们需要的辅助函数

    template <size_t T>
    static void showBitset (bitset<T> s, bool nl = true) {

    template <size_t T>
    static bitset<T> reverse (bitset<T> s) {

    bitset<64> ToBitset64(char s[8], int len = 8) {

    // 循环移位
    template <size_t T>
    bitset<T> shiftLeft(bitset<T> K, int shiftLen) {
```

而我们所需的各种置换中的矩阵都以数组的形式声明在类中。

```
const int IP[64] = { 58, 50, 42, 34, 26, 18, 10, 2,

const int IP_1[64] = { 40, 8, 48, 16, 56, 24, 64, 32,

const int E[48] = {   32,  1,  2,  3,  4,  5,

const int S_BOX[8][4][16] = {

const int P[32] = {   16,  7, 20, 21,

const int PC_1[56] = {  57, 49, 41, 33, 25, 17, 9,

const int PC_2[48] = {  14, 17, 11, 24,  1,  5,

const int shiftBits[16] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1};
```

数据结构

即上文所提到的类，类内部关于比特操作均使用 STL 中的 bitset 类。由于实际上只使用了 bitset 类的下标访问，而没有使用到 bitset 内部提供的其他函数，因此在没有 STL 的环境中使用 int 数组或 char 数组等一些可进行比特级操作的结构也可以代替。