《密码学》课程设计实验报告

实验序号：01　　实验项目名称：分组密码 DES

学　　号	2021302141097	姓　　名	李宇	专业、班	信安 9 班
实验地点	C102 机房	指导教师	王后珍	时间	23-12-01

一、实验目的及要求

教学目的：

(1) 掌握分组密码的基本概念；

(2) 掌握 DES（3DES）密码算法；

(3) 了解 DES（3DES）密码的安全性；

(4) 掌握分组密码常用工作模式及其特点；

(5) 熟悉分组密码的应用。

实验要求：

(1) 复习掌握（古典密码）使用的置换、代替、XOR、迭代等技术；

(2) 比较 DES 中代替技术与古典密码中的联系与区别；

(3) 理解 S 盒、P 置换等部件的安全性准则；

(4) 实现 DES 算法的编程与优化。

二、实验设备（环境）及要求

Windows 操作系统，高级语言开发环境

三、实验内容与步骤

1. DES 子密钥扩展算法的实现

输入：64 位密钥

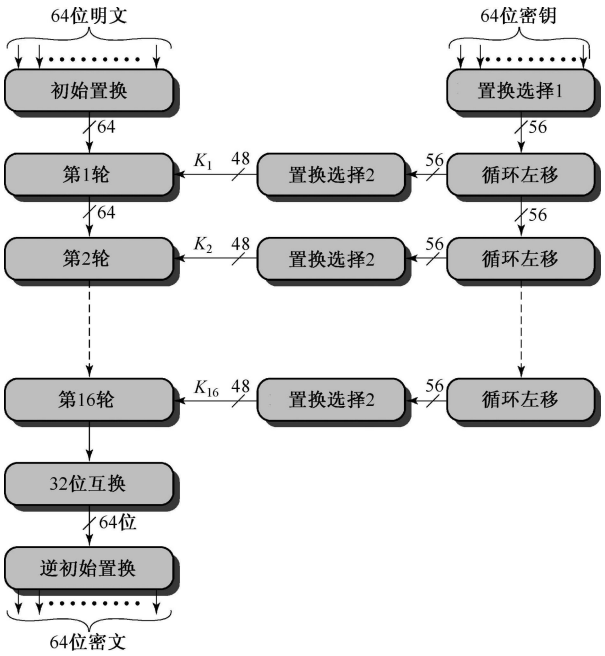
子过程：

（1）置换选择 1（教材 图 3-3）

（2）循环左移（教材 表 3-1）

（3）置换选择 2（教材 图 3-4）

输出：16 个 48 位长的子密钥。



2．DES 局部加密函数 *f* 的实现

加密函数是 DES 的核心部分。它的作用是在第 *i* 次加密迭代中用子密钥 K_i 对 R_{i-1} 进行加密。

输入：32 位 R_{i-1} 和 48 位子密钥 K_i

子过程：

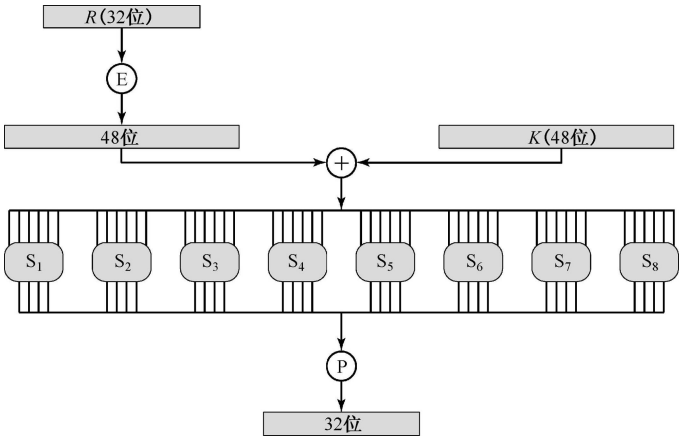
（1） 扩展置换 E（教材 图 3-7）：将 32 位 R_{i-1} 扩展为 48 位；

（2） 异或操作：步骤（1）的 48 位结果与子密钥 K_i 按位模 2 相加；

（3） 代替 S 盒（教材 表 3-2）：步骤（2）的 48 位结果分成 6 位×8 组压缩为 4 位×8 组，即 32 位输出；

（4） 置换运算 P（教材 图 3-8）：32 位输入/输出。

输出：32 位 $f(R_{i-1}, K_i)$



3. DES 加密过程完整实现

① 64 位密钥经子密钥产生算法产生出 16 个子密钥：K_1 ，K_2 ，...，K_{16}，分别供第一次，第二次，...，第十六次加密迭代使用。

② 64 位明文首先经过初始置换 ***IP***（Initial permutation），将数据打乱重新排列并分成左右两半。左边 32 位构成 L_0，左边 32 位构成 R_0。

③ 由加密函数 f 实现子密钥 K_1 对 R_0 的加密，结果为 32 位的数据组 f（R_0 ，K_1)。f（R_0 ，K_1)再与 L_0 模 2 相加，又得到一个 32 位的数据组 $L_0 \oplus f$（R_0 ，K_1)。以 $L_0 \oplus f$（R_0 ，K_1)作为第二次加密迭代的 R_1，以 R_0 作为第二次加密迭代的 L_1。至此，第一次加密迭代结束。

④ 第二次加密迭代至第十六次加密迭代分别用子密钥 K_2 ，...，K_{16} 进行，其过程与第一次加密迭代相同。

⑤ 第十六次加密迭代结束后，产生一个 64 位的数据组。以 R_{16} 作为其左边 32 位，以 L_{16} 作为其右边 32 位，两者合并再经过逆初始置换 IP^{-1}，将数据重新排列，便得到 64 位密文。至此加密过程全部结束。

综上可将 DES 的加密过程用如下的数学公式描述：

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \\ i = 1,2,3,\ldots 16 \end{cases} \qquad (3\text{-}1)$$

4. DES 解密过程实现

由于 DES 的运算是对和运算，所以解密和加密可共用同一个运算，只是子密钥使用的顺序不同。

把 64 位密文当作明文输入，而且第一次解密迭代使用子密钥 K_{16}，第二次解密迭代使用子密钥 K_{15}，...，第十六次解密迭代使用子密钥 K_1，最后的输出便是 64 位明文。

解密过程可用如下的数学公式描述：

$$\begin{cases} R_{i-1} = L_i \\ L_{i-1} = R_i \oplus f(L_i, K_i) \\ i = 16,15,14,\ldots,1 \end{cases} \qquad (3\text{-}2)$$

5. DES 的 S 盒密码学特性（重点）

通过编程实现或者手工计算，试验证 ***S*** 盒的以下准则：

① 输出不是输入的线性和仿射函数；

② 任意改变输入中的一位，输出至少有两位发生变化；

③ 对于任何 ***S*** 盒和任何输入 x，$\boldsymbol{S}(x)$和 $\boldsymbol{S}(x \oplus 001100)$至少有两位不同，这里 x 是一个 6 位的二进制串；

④ 对于任何 ***S*** 盒和任何输入 x，以及 $y,z \in GF(2)$，$\boldsymbol{S}(x) \neq \boldsymbol{S}(x \oplus 11yz00)$，这里 x 是一个 6 位的二进制串；

⑤ 保持输入中的 1 位不变，其余 5 位变化，输出中的 0 和 1 的个数接近相等。

例如，可通过如下步骤验证②、③两条：

设 S 盒的输入为 X，输出为 Y。（X 和 Y 都以二进制表示）

（1）对于已知输入值 X_1=110010 和 X_2=100010，分别求出对应的输出值 Y_1 和 Y_2。

（2）比较输出值 Y_1 和 Y_2 各位的异同，即按位计算 $Y_1 \oplus Y_2$。

根据上面得出的结果试说明 S 盒对于 DES 的安全性影响。

6. 验证教材 P64 页实例（重点）

7.扩展思考

（1）Feistel 结构为什么可以保证算法的对合性？

（2）第 16 轮为什么不做左右互换？

（3）如果去掉初始置换和逆初始置换，对算法安全性有影响吗？（提示：算法所有的细节都是公开的）

（4）证明 DES 解密算法是加密算法的逆，即 DES 的对合性。

四、实验结果与数据处理

1. 实验环境

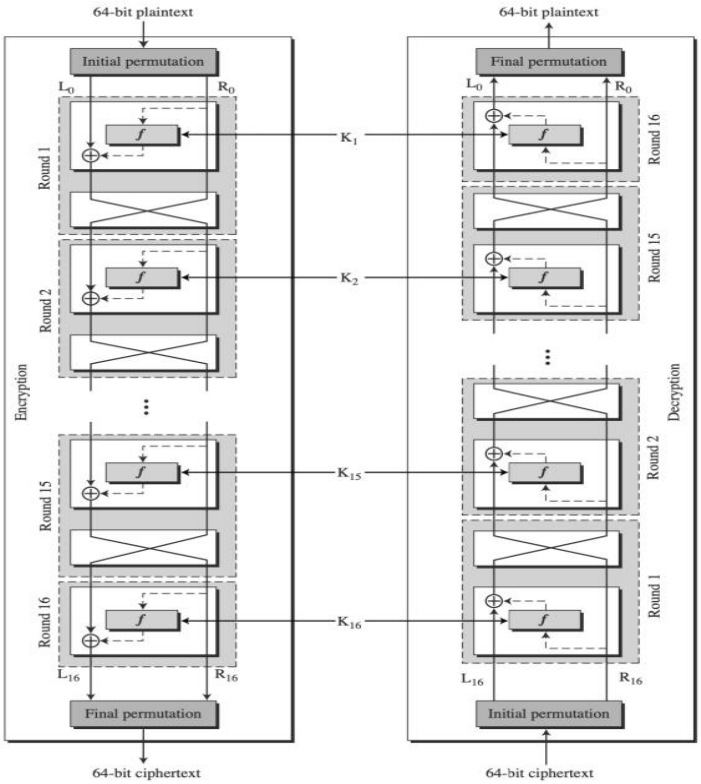
操作系统：Windows11 家庭版

基本硬件：CPU：AMD Ryzen 5 5600H, 16G 内存

所用软件：Pycharm 2023.2.1 专业版

2. 实验具体过程

在这一部分我会给出 DES 加解密算法的每一步的具体代码实现，以下图的流程为准。



将 DES 的加密过程分为了以下具体步骤，分别是

(1) 数据预处理

(2) 初始变换 IP

(3) 生成子密钥

(4) 使用子密钥与核函数迭代加密

(5) 逆初始变换 IP^{-1}

(6) 输出格式转换

由于 DES 为对合运算，所以在解密过程中只需要在第三步生成子密钥后将子密钥颠倒即可，操作较为简单。下面对加密过程的上述 6 步进行代码实现。

(1)数据预处理

在代码的编写中，我使用的是 64 位 16 进制明文与 64 位 16 进制密钥，因此在开始正式加密前需要将它们转化为64位2进制数，具体操作如下代码所示：

```
01. def hex_to_binary(hex_str):
02.     return bin(int(hex_str, 16))[2:].zfill(4)
03. 
04. plaintext = "3031323334353637"  # 64位十六进制明文
05. key = "3132333435363738"  # 64位十六进制密钥
06. 
07. binary_data = ''.join(hex_to_binary(hex_digit) for hex_digit in data)
08. binary_key = ''.join(hex_to_binary(hex_digit) for hex_digit in key)
```

通过数据预处理，原本的 16 进制明文和密钥，转化为了 64 位 2 进制字符串。

(2)初始变换 IP

由初始变换的原理可知，IP 即为一个 **reshuffle 过程**，将原始的 64 位明文数据进行**非线性映射**。为实现此功能，只需编写置换矩阵，将原本的 64 位明文重新排放位置即可，代码如下所示：

```
01. def permute(data, table):
02.     return ''.join(data[i - 1] for i in table)
03. 
04. def initial_permutation(data):
05.     permutation_table = [
06.         58, 50, 42, 34, 26, 18, 10, 2,
07.         60, 52, 44, 36, 28, 20, 12, 4,
08.         62, 54, 46, 38, 30, 22, 14, 6,
09.         64, 56, 48, 40, 32, 24, 16, 8,
10.         57, 49, 41, 33, 25, 17, 9, 1,
11.         59, 51, 43, 35, 27, 19, 11, 3,
12.         61, 53, 45, 37, 29, 21, 13, 5,
13.         63, 55, 47, 39, 31, 23, 15, 7
14.     ]
15.     return permute(data, permutation_table)
```

(3)生成子密钥

在这一步需要根据原本 64 位的密钥生成 **16 个 48 位的子密钥**。子密钥的生成需要经过**置换选择 1、循环左移、置换选择 2** 三部分完成，由于两个置换选择的原理与初始变换 IP 一样，只是将原本的密钥位置重排，所以我们只需要编写好相应的置换矩阵，**调用 permute 函数**置换即可；在循环左移上，编写每一次循环时循环左移的位数，同时在每次循环时高位溢出的部分填补在低位即可。具体的实现代码如下所示：

```
def generate_subkeys(key):
    # 生成16个子密钥
    key_permutation_table = [
        57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4
    ]

    key_shift_table = [
        1, 1, 2, 2, 2, 2, 2, 2,
        1, 2, 2, 2, 2, 2, 2, 1
    ]

    key = permute(key, key_permutation_table)
    left, right = key[:28], key[28:]  #初始的C0，D0

    subkeys = []
    for shift in key_shift_table:
        left = left[shift:] + left[:shift]
        right = right[shift:] + right[:shift]
        subkey = permute(left + right, [
            14, 17, 11, 24, 1, 5, 3, 28,
            15, 6, 21, 10, 23, 19, 12, 4,
            26, 8, 16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55, 30, 40,
            51, 45, 33, 48, 44, 49, 39, 56,
            34, 53, 46, 42, 50, 36, 29, 32
        ])
        subkeys.append(subkey)

    return subkeys
```

只需要将原始的密钥 key 作为输入，调用函数 **generate_subkeys** 即可成功产生 16 个子密钥，子密钥作为**列表 subkeys 的成员**返回。

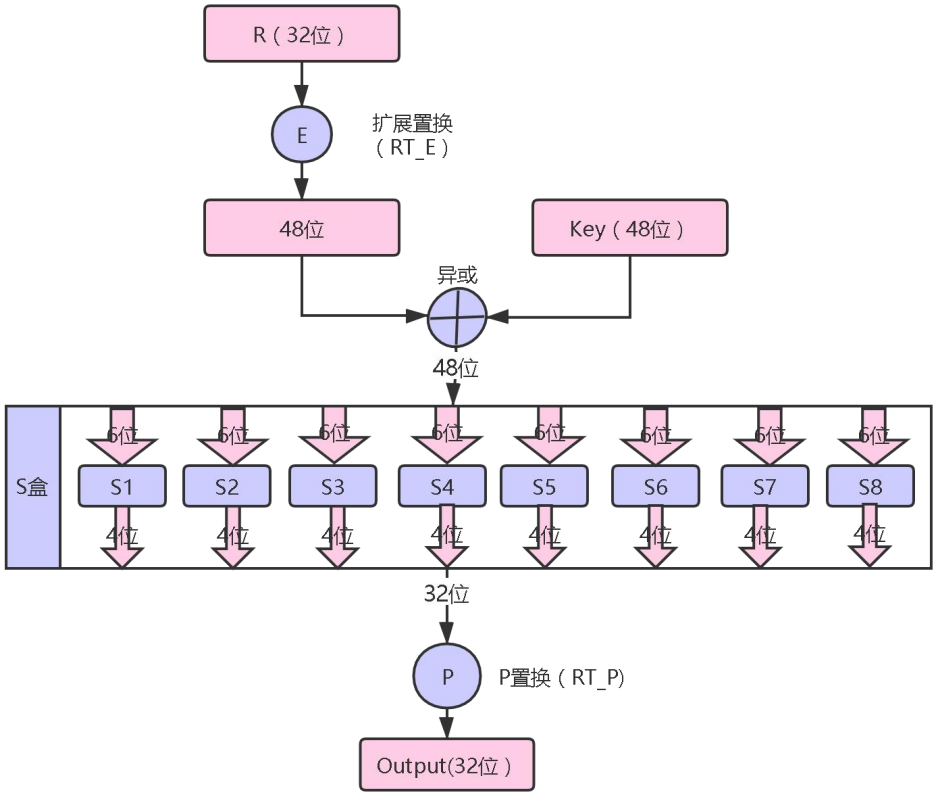
(4)使用子密钥与核函数进行迭代加密

在使用核函数进行加密的过程中，首先需要对将原始明文划分为L,R 两部分，后续主要对 R 进行加密。

首先对 R 进行选择运算 E，这一步实际上是根据矩阵进行扩位，将原本的 32 位 R 扩展为 48 位，将得到的 48 位中间结果与 48 位子密钥进行**模 2 相加**（实际

上就是**异或**运算），将异或运算得到的 48 位结果送入**核函数**中进行处理。每个置换函数**输入为 6 位，输出为 4 位**，最终 48 位的输入被转化为了 32 位输出。32 位的输出再经过一个**置换运算 P** 即得到最终的加密函数的输出。

加密函数的输出再与原本的 L 进行异或运算得到的值作为新 R，旧 R 值作为新 L 值。实现流程如下图所示：



具体的实现代码如下所示：

```
def feistel_network(data, subkey,k):
    # Feistel网络
    left, right = data[:32], data[32:]
    expanded_right = permute(right, [   #选择运算E
        32, 1, 2, 3, 4, 5, 4, 5,
        6, 7, 8, 9, 8, 9, 10, 11,
        12, 13, 12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21, 20, 21,
        22, 23, 24, 25, 24, 25, 26, 27,
        28, 29, 28, 29, 30, 31, 32, 1
    ])
    xor_result = bin(int(expanded_right, 2) ^ int(subkey, 2))[2:].zfill(48)  #模2相加（等同于异或）
    substitution_result = substitution(xor_result)   #S盒置换，输入一个48位，返回32位
    permuted_result = permute(substitution_result, [   #置换运算P
        16, 7, 20, 21, 29, 12, 28, 17,
        1, 15, 23, 26, 5, 18, 31, 10,
        2, 8, 24, 14, 32, 27, 3, 9,
        19, 13, 30, 6, 22, 11, 4, 25
    ])
    new_right = bin(int(left, 2) ^ int(permuted_result, 2))[2:].zfill(32)
    new_lift=right
    if k==16:
        return new_right + new_lift
    return new_lift + new_right
```

而核函数实际上就是一个**替代函数组**，代替函数组由 8 个代替函数(也称 S 盒子)组成。代替函数组的输人是一个 48 位的数据，从第 1 位到第 48 位依次加到 8 个 S 盒的输入端。每个 S 盒有一个**代替矩阵**，规定了其输出与输入的代替规则。代替矩阵有 4 行 16 列、每行都是 0 到 15 这 16 个数字、但每行的数字排列都不同，而且 8 个代替矩阵彼此也不同。每个 S 盒有 6 位输入，产生 4 位的输出。S 盒运算的结果是用输出数据代替了输入数据，所以称其为代替函数。

S 盒的代替规则是:S 盒的 6 位输入 b1,b2,b3,b4,b5,b6 中的第 1 位 b1 和第 6 位 b6 组成的**二进制数 b1b6 代表选中的行号**，其余 4 位数字 b2,b3,b4,b5 所组成的**二进制数代表选中的列号**,而处在被选中的行号和列号交点处的数字便是 S 盒的输出(以二进制形式输出)。以 S 为例，设输入 b1,b2,b3,b4,b5,b6=101011 第 1 位和第 6 位数字组成的二进制数 b1b6=l l=(3)，表示选中 S 的行号为 3 的那一行，其余 4 位数字所组成的二进制数为 b2,b3,b4,b5=0101=(5)，表示选中 S 的列号为 5 的那一列。交点处的数字是 9，则 S 的输出为 1001。关于加密函数的整体流程如下所示：

```
def substitution(data):
    # S盒替换
    s_boxes = [
        [
            [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
            [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
            [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
            [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
        ],
        [
            [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
            [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
            [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
            [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]
        ],
        [
            [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
            [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
            [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
            [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]
        ],
        [
            [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
            [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
            [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
            [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]
        ],
        [
            [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
            [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
            [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
            [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]
        ],
        [
            [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
            [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
            [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
            [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]
```

```
        ],
        [
            [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
            [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
            [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
            [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]
        ],
        [
            [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
            [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
            [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
            [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
        ]
    ]

    result = ''
    for i in range(0, len(data), 6):
        block = data[i:i + 6]
        row = int(block[0] + block[5], 2)
        col = int(block[1:5], 2)
        result += bin(s_boxes[i // 6][row][col])[2:].zfill(4)

    return result
```

(5) 逆初始变换 IP^{-1}

经过 16 次循环加密后，得到最终的 **L_{16} 与 R_{16} 即 64 位密文**，由于在加密开始时经过了一步初始变换 IP，所以还需对密文进行一次**逆初始变换**，原理与初始变换 IP 相同，只是将原本的 64 位数据进行位置重排。具体的实现代码如下所示：

```
def final_permutation(data):
    # 最终置换
    permutation_table = [
        40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
        36, 4, 44, 12, 52, 20, 60, 28,
        35, 3, 43, 11, 51, 19, 59, 27,
        34, 2, 42, 10, 50, 18, 58, 26,
        33, 1, 41, 9, 49, 17, 57, 25
    ]
    return permute(data, permutation_table)
```

(6) 输出格式转换

在开始时，我们输入的明文以及密钥都为 16 进制数，但在加密过程中为了方便我们参考课本内容进行处理，将 16 进制数转化为了对应的 2 进制数，即原本的 16 位数据转换为了 64 位数据，因此在加密完成后还需要将 64 位 2 进制数据转换为对应的 16 进制。如下代码所示：

```
encrypted_data= hex(int(encrypted_data, 2))[2:].zfill(16)
```

最终 DES 的整体加密流程函数如下所示：

```python
def des_encrypt(data, key):
    # DES加密
    binary_data = ''.join(hex_to_binary(hex_digit) for hex_digit in data)
    binary_key = ''.join(hex_to_binary(hex_digit) for hex_digit in key)
    data = initial_permutation(binary_data)  #初始变换IP
    subkeys = generate_subkeys(binary_key)  #生成子密钥
    k=0
    for subkey in subkeys:
        k+=1
        data = feistel_network(data, subkey,k)
    encrypted_data = final_permutation(data)  #逆初始变换IP-
    encrypted_data= hex(int(encrypted_data, 2))[2:].zfill(16)
    return encrypted_data
```

由于 DES 具有**对合性**，在解密过程中只需将密文作为加密输入，将子密钥的使用顺序进行**逆序处理**，最终就可得到原始明文，实现代码如下所示：

```python
def des_decrypt(data, key):
    # DES加密
    binary_data = ''.join(hex_to_binary(hex_digit) for hex_digit in data)
    binary_key = ''.join(hex_to_binary(hex_digit) for hex_digit in key)
    data = initial_permutation(binary_data)  #初始变换IP
    subkeys =  generate_subkeys(binary_key)  #生成子密钥
    subkeys.reverse()  #颠倒子密钥的顺序
    k = 0
    for subkey in subkeys:
        k += 1
        data = feistel_network(data, subkey,k)
    decrypted_data = final_permutation(data)  #逆初始变换IP-
    decrypted_data= hex(int(decrypted_data, 2))[2:].zfill(16)
    return decrypted_data
```

3. 实验结果展示

在测试阶段，我选取以下明文以及密钥进行加解密处理(**其中第一组测试数据为要求验证的教材 P64 页示例**)：

64 位明文(16 进制)	64 位密钥(16 进制)
3031323334353637	3132333435363738
1234567890abcdef	31abc134aaa63738
a1a2a3a4b1b2b3b4	ffffffffffffffff
fdf1f2fafef5f900	fdf1f2fafef5f900

运行代码可得到下图结果：

```
D:\Anaconda\envs\pytorch\python.exe D:\python\pythonProject\main.py
原始数据 3031323334353637  密钥 3132333435363738  加密后的数据: 8bb47a0cf0a9626d  解密后的数据: 3031323334353637
原始数据 1234567890abcdef  密钥 31abc134aaa63738  加密后的数据: ee1475c776fe2828  解密后的数据: 1234567890abcdef
原始数据 a1a2a3a4b1b2b3b4  密钥 ffffffffffffffff  加密后的数据: a139bd0c93917178  解密后的数据: a1a2a3a4b1b2b3b4
原始数据 fdf1f2fafef5f900  密钥 fdf1f2fafef5f900  加密后的数据: da950df938f37d7f  解密后的数据: fdf1f2fafef5f900

进程已结束，退出代码为 0
```

可以看到，程序成功使用密钥加密了原始数据，解密所得的数据与原始数据相同。并且**教材 P64 页也得到了成功验证，**说明我们编写的 python 代码成功实现了 DES 加解密。

五、S 盒密码学特性讨论

为了单独探究 S 盒的密码学特性，新编**函数 *substitution2*** 如下所示：

```
def substitution2(data):
    # S盒替换
    s_boxes = [
        [
            [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
            [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
            [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
            [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
        ],
    ]
    result = ''
    block = data
    row = int(block[0] + block[5], 2)
    col = int(block[1:5], 2)
    result += bin(s_boxes[0][row][col])[2:].zfill(4)

    return result
```

该函数可完成对给定的 6 位输入，得到 S 盒 4 位输出结果，且仅保留了 1 个 S 盒用于探究 S 盒的密码学特性。

(1) 任意改变输入中的一位，输出至少有两位发生变化；

令输入分别为$(110010)_2$和$(100010)_2$将它们送入上述函数之中得到结果如下图所示：

```
D:\Anaconda\envs\pytorch\python.exe D:\python\pythonProject\main.py
S盒输入 110010  S盒输出 1100
S盒输入 100010  S盒输出 0001

进程已结束，退出代码为 0
```

可以看到当输入改变一位时，输出有 3 位发送了改变。

(2) 对于任何 *S* 盒和任何输入 *x*，*S*(*x*)和 *S*(x⊕001100)至少有两位不同，这里 *x* 是一个 6 位的二进制串；

令输入 x 为$(110010)_2$那么 x 异或 001100 的结果即为 111110，将它们送入 S 盒中进行处理，得到结果如下所示：

```
D:\Anaconda\envs\pytorch\python.exe D:\python\pythonProject\main.py
S盒输入 110010  S盒输出 1100
S盒输入 111110  S盒输出 0000

进程已结束，退出代码为 0
```

可以发现输出中有两位发生了改变，即 $S(x)$和 $S(x\oplus 001100)$至少有两位不同。

(3) 对于任何 S 盒和任何输入 x，以及 $y,z\in GF(2)$，$S(x)\neq S(x\oplus 11yz00)$，这里 x 是一个 6 位的二进制串；

选定输入 x 为$(110010)_2$那 y,z 均取为 0，可得 $x\oplus 11yz00$ 的结果为 000010，将它们送入 S 盒中进行处理，得到结果如下所示：

```
D:\Anaconda\envs\pytorch\python.exe D:\python\pythonProject\main.py
S盒输入 110010  S盒输出 1100
S盒输入 000010  S盒输出 0100

进程已结束，退出代码为 0
```

从代码的运行结果可以看到输出发生了改变，即 $S(x)\neq S(x\oplus 11yz00)$。

(4) 保持输入中的 1 位不变，其余 5 位变化，输出中的 0 和 1 的个数接近相等；

此处令输入 x 为$(111111)_2$，保持第 2 位不发生改变，其余 5 位均置为 0，得到结果如下所示：

```
D:\Anaconda\envs\pytorch\python.exe D:\python\pythonProject\main.py
S盒输入 111111  S盒输出 1101
S盒输入 010000  S盒输出 0011

进程已结束，退出代码为 0
```

六、拓展思考

（1）Feistel 结构为什么可以保证算法的对合性？

（2）第 16 轮为什么不做左右互换？

（3）如果去掉初始置换和逆初始置换，对算法安全性有影响吗？（提示：算法所有的细节都是公开的）

（4）证明 DES 解密算法是加密算法的逆，即 DES 的对合性。

实际上问题 1，2，4 的本质都是一样的，即 DES 的对合性的体现，下面给出 DES 对合性的证明过程：

设 L、R 为 64 位数据的左右两半，定义变换 T 如下：

$$T(L,R)=(R,L) \tag{1}$$

记 DES 第 i 轮中一步主要运算为：

$$F_i(L_{i-1},R_{i-1})=(L_{i-1}\oplus f(R_{i-1},k_i),R_{i-1}) \tag{2}$$

将(1)(2)式结合，即可构成 DES 的轮运算：

$$H_i=F_iT$$

由 $$T^2(L,R)=(L,R)=I \tag{3}$$

其中 I 为恒等变换可得 $T=T^{-1}$

所以 T 变换为对合运算

同理：

$$\begin{aligned}F_i^2(L_{i-1},R_{i-1})&=F_i(L_{i-1}\oplus f(R_{i-1},k_i),R_{i-1})\\&=(L_{i-1}\oplus f(R_{i-1},k_i)\oplus f(R_{i-1},k_i),R_{i-1})\\&=(L_{i-1},R_{i-1})\\&=I\end{aligned} \tag{4}$$

可知 F_i 变换也是对合运算。

结合(3)(4)可知

$$(F_iT)(TF_i)=F_i(TT)F_i=F_iF_i=I$$

由 DES 加解密过程可知，写成表达式如下所示：

$$DES=IP(F_1T)(F_2T)\cdots(F_{15}T)(F_{16})IP^{-1}$$

$$(DES)^{-1}=IP(F_{16}T)(F_{15}T)\cdots(F_2T)(F_1)IP^{-1}$$

DES 先加密后解密的过程为：

$(DES)(DES^{-1}) = IP(F_1T)\cdots(F_{16})IP^{-1}\ IP(F_{16}T)\cdots(F_1)IP^{-1}$

$= IP(F_1T)\cdots(F_{16})(F_{16}T)\cdots(F_1)IP^{-1}$

$= IP\ IP^{-1}$

$= I$

此时可知 DES 算法具有可逆性

DES 先解密后加密的过程为：

$(DES^{-1})(DES) = IP(F_{16}T)\cdots(F_1)IP^{-1}\ IP(F_1T)\cdots(F_{16})IP^{-1}$

$= IP(F_{16}T)\cdots(F_1)(F_1T)\cdots(F_{16})IP^{-1}$

$= IP\ IP^{-1}$

$= I$

由 F_i 的过程可知，DES 与 DES^{-1} 除子密钥使用顺序互相反之外，其它都是相同的，即 DES 算法具有对合性。

证明过程直接解释了问题 1，4；至于问题 2，16 轮不做左右互换的原因是**为了保证 DES 算法的对合性**，从而降低算法的工作量。

对于问题 3，实际上是**可以去除**初始变换 IP 与逆初始变换 IP^{-1} 这两步操作的，不会影响最终结果。早期 DES 算法**只允许硬件实现**，为了加强硬件的破解难度而添加的，在软件实现中可以去掉而不影响安全性，同时也加快了加解密的速度。

七、分析与讨论

在此部分主要分析的是迭代加密的最后一步，即下图所示过程。

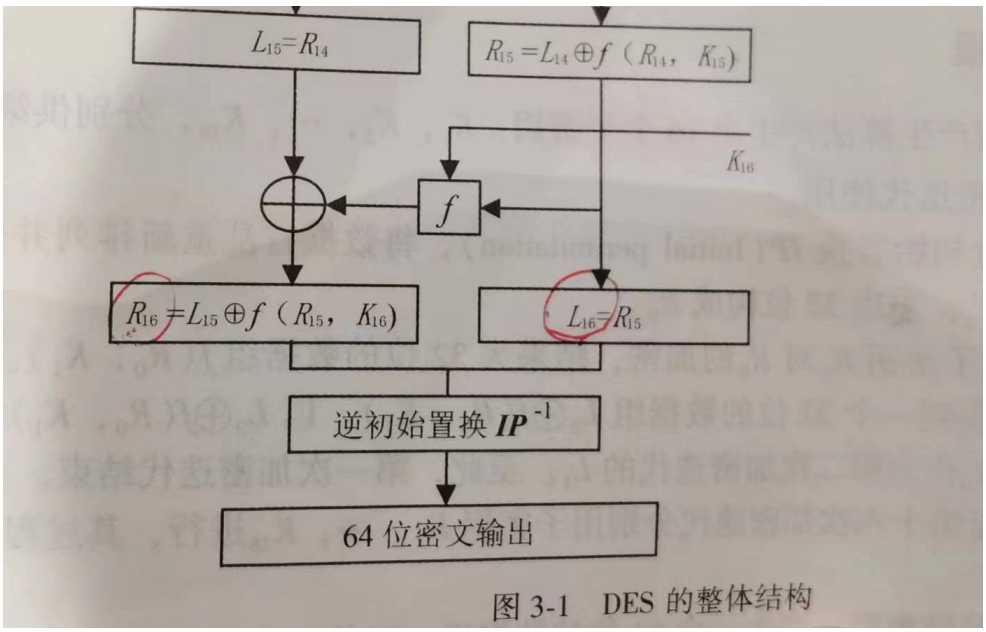


图 3-1　DES 的整体结构

在使用代码实现 DES 算法时，开始没有注意到最后一次加密时**不需要交换 L,R 两者的位置**，导致解密所得明文与初始时不同。通过调试**课本 64 页**密钥扩展过程，发现密钥扩展并没有问题；紧接着又调试了课本 66 页的加密过程，发现直到第 16 次加密前所有的输出均正确，但经过第 16 次加密之后 L,R 值出错，又联想到**图 3-1** 中的最后一次加密过程，发现没有交换置位的过程，修改代码逻辑后，再次加解密发现成功解密成功得到了原始明文。

八、教师评语

成绩：

签名：

日期：