开发手册&用户指南

—— written by 荔枝

1. 简介

1.1 系统概述

该系统是我们“荔枝”团队为用户设计的一款简单、开源的 DES 算法加解密系统，能够满足用户简单的加解密需要（二进制加解密和 ASCII 码字符串加解密），同时可以帮助用户在已知明密文的条件下暴力破解出密钥，以及向用户展示 DES 算法的雪崩效应。

同时，也欢迎广大程序设计与编码工作者继续完善该加密系统。如有任何疑问和建议，欢迎联系我们团队（Email：1635487611@qq.com）。

1.2 主要功能和特点

该系统主要功能包括对 8-bits 二进制数进行 DES 算法的加解密，对 1-byte ASCII 码进行 DES 算法的加解密、根据明密文对进行暴力破解计算、验证 DES 算法的雪崩效应等。

该系统的特点在于简单易懂、容易上手，可以帮助不太了解 DES 算法的用户快速获取加解密结果以及相应的密钥信息，也可以帮助正在学习 DES 算法学生更直观的接触到简单的 DES 应用，验证 DES 的雪崩效应特性，感受 DES 加密算法的伟大和奇妙之处。

1.3 目标用户群体

初步学习和希望了解 DES 算法的用户。

1.4 使用说明

该系统可以在浏览器运行，无需特殊的环境配置。具体页面及功能详见下文。

2. 开发者团队

2.1 开发者队名：荔枝

2.2 开发者姓名：张芷芮、刘俐莹

2.3 联系信息：1635487611@qq.com

3. 开发环境

3.1 开发工具和技术栈

3.1.1 开发工具

- 集成开发化境（IDE）：PyCharm 2023.3
- 版本控制：Github

3.1.2 技术栈

- 前端技术：语言：HTML5，CSS3，JavaScript

  框架和库：jQuery
- 后端技术：语言：Python

  框架：Flask 框架
- Web 浏览器：Microsoft Edge

3.2 运行

python 文件在根目录下运行：main.py

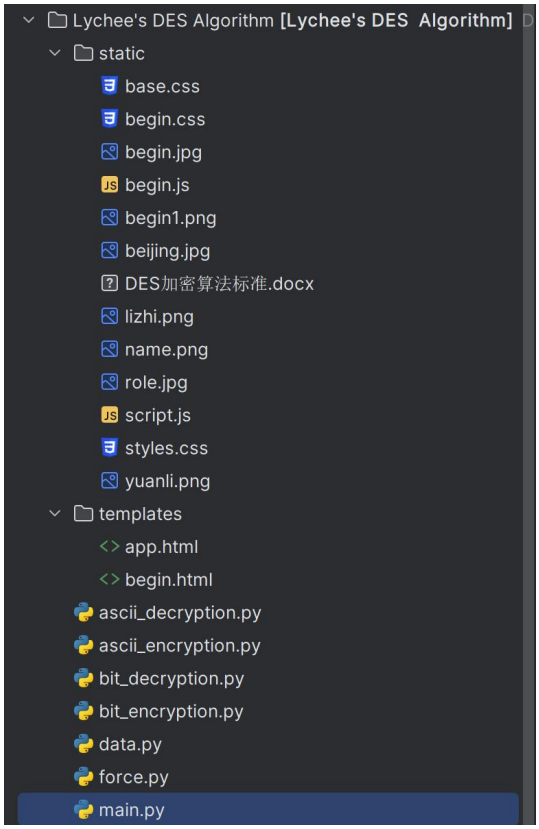
3.3 网址与端口号

该系统在 http://127.0.0.1:6606 上运行

4. 代码结构

4.1 项目的目录结构

项目的目录结构图如下：



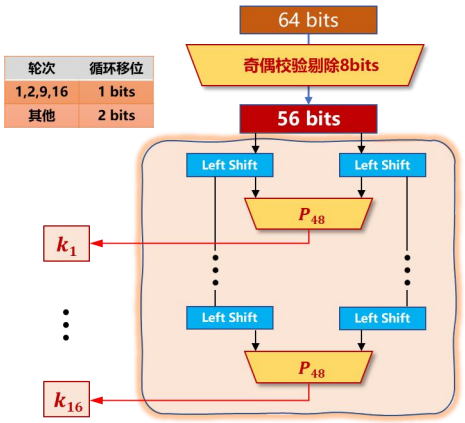
- static 文件夹中存放图片、js 文件和 css 样式表等

  （begin.html：启动页面；app.html：主页面）
- templates 文件夹中存放 html 文件
- data.py 文件中存放标准加解密数据和两个基础函数（密钥生成函数、轮函数）
- main.py 文件中主要存放连接前后端的路由
- bit_encryption.py 文件中主要存放二进制加密算法
- bit_decryption.py 文件中主要存放二进制解密算法
- ascii_encryption.py 文件中主要存放 ASCII 码的加密算法
- ascii_decryption.py 文件中主要存放 ASCII 码的解密算法
- force.py 文件中主要存放暴力破解的算法

4.2 关键组件和模块的描述

4.2.1 密钥生成函数

1、原理：



2、代码演示：

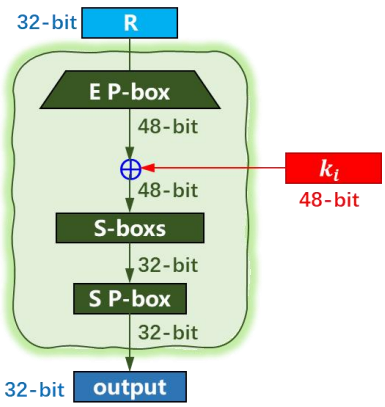
该代码首先执行置换操作。再将密钥分成两部分 left_half 和 right_half，分别执行左移一位的操作，然后执行第一次压缩置换的操作，生成第一个密钥 K1。接着重复拆分左移和压缩置换的操作，得到第二个密钥 K2。

具体代码逻辑如下所示：

```python
# 密钥生成
2 个用法
def generate_keys(key, p10, p8):
    # 置换输入密钥，生成 P10 密钥
    p10_key = ''
    for i in p10:
        p10_key += key[i - 1]

    # 生成第一个子密钥
    # 1、对 P10 密钥（p10_key）左移 1 位
    left_half = p10_key[:5]
    right_half = p10_key[5:]
    shifted_left = left_half[1:] + left_half[:1]
    shifted_right = right_half[1:] + right_half[:1]
    shifted_key1 = shifted_left + shifted_right
    # 2、进行第一次压缩置换，生成k1
    key1 = ''
    for i in p8:
        key1 += shifted_key1[i - 1]

    # 生成第二个子密钥
    # 1、对第一个子密钥（shifted_key1）左移 1 位
    left_half1 = shifted_key1[:5]
    right_half1 = shifted_key1[5:]
    shifted_left1 = left_half1[1:] + left_half1[:1]
    shifted_right1 = right_half1[1:] + right_half1[:1]
    shifted_key2 = shifted_left1 + shifted_right1
    # 2、进行第二次压缩置换，生成k1
    key2 = ''
    for i in p8:
        key2 += shifted_key2[i - 1]
    # 得到两个生成的子密钥k1,k2
    return key1, key2
```

4.2.2 轮函数

1、原理：



2、代码演示：

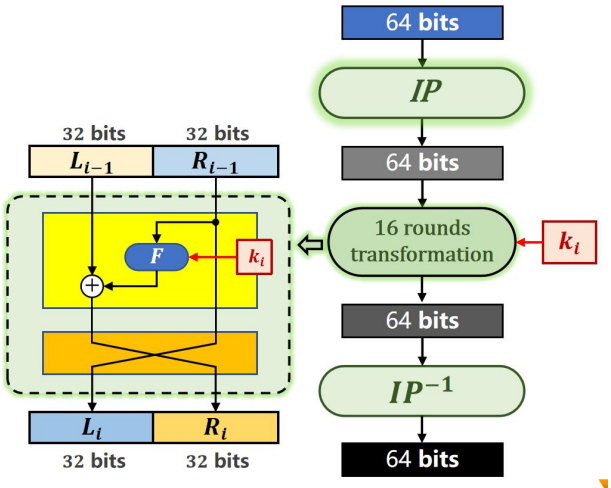
该函数接受加解密过程中拆分出的右半部分以及轮密钥输入，首先将右半部分（right-half）进行 E P-Box 的扩展置换，扩展为 10-bits，再与轮密钥（10-bits）进行亦或操作。进而对所得结果通过 S-boxs 进行压缩替换，得到 8-bits 结果。最后，进行 S P-box 的直接置换。

具体代码逻辑如下所示：

```python
def f(right_half, ki):
    # 1、进行EPBox扩展置换
    expanded = ''
    for i in EPBox:
        expanded += right_half[i - 1]
    # 2、与轮密钥ki进行亦或操作
    xored = int(expanded, 2) ^ int(ki, 2)
    xored_str = format(xored, '08b')  # 转换为二进制字符串

    # 3、进行S-Box压缩替换（根据所在行列进行定位）
    s0_input = xored_str[:4]
    s1_input = xored_str[4:]
    s0_row = int(s0_input[0] + s0_input[3], 2)  # 计算 S-box 0 的行
    s0_col = int(s0_input[1:3], 2)  # 计算 S-box 0 的列
    s1_row = int(s1_input[0] + s1_input[3], 2)  # 计算 S-box 1 的行
    s1_col = int(s1_input[1:3], 2)  # 计算 S-box 1 的列
    s0_output = format(SBox1[s0_row][s0_col], '02b')  # 获取 S-box 0 的输出
    s1_output = format(SBox2[s1_row][s1_col], '02b')  # 获取 S-box 1 的输出
    s_output = s0_output + s1_output  # 合并 S-box 的输出

    # 4、进行E P-box直接置换（二维置换）
    permuted_output = ''
    for i in SPBox:
        permuted_output += s_output[i - 1]

    return permuted_output
```

4.2.3 二进制加密算法

1、原理：



2、代码演示：

该加密函数接受输入框中明文和密钥的输入，首先调用密钥生成函数生成密钥 K1，K2。然后，利用 IP 进行初始置换操作。再进入多轮变换部分，此时需要把现有 8-bits 经过转换的明文分为左右两部分（left-half 和 right-half）。执行第一轮加密，先用右半部分和轮密钥 key1 执行轮函数操作。再将结果与左半部分进行亦或操作。然后按照同样的过程执行第二轮加密。最终将经过一次交换（第二轮不会发生左右部分的交换）的内容合并，利用 IP^{-1} 进行最终置换操作。

具体代码逻辑如下所示：

```python
# 编码过程
4 个用法
def encrypt(plaintext, key):
    # 1、生成密钥k1 k2
    key1, key2 = generate_keys(key, P10, P8)

    # 初始置换（利用IP）
    ip_output = ''
    for i in IP:
        ip_output += plaintext[i - 1]

    # 多轮变换部分
    # 将明文分为左右两半部分
    left_half = ip_output[:4]
    right_half = ip_output[4:]
    left_previous = right_half

    # 第一轮加密
    # （1）将右半部分和子密钥key1进行f操作。
    right_result = f(right_half, key1)
    # （2）将f的结果（right_result）与左半部分进行XOR操作
    right_half1_int = int(left_half, 2) ^ int(right_result, 2)
    right_half1 = format(right_half1_int, '04b')
```

```python
    # 第二轮加密
    # （1）将右半部分和子密钥key2进行f操作。
    right_result = f(right_half1, key2)
    # （2）将f的结果（right_result）与左半部分进行XOR操作
    right_half2_int = int(left_previous, 2) ^ int(right_result, 2)
    right_half2 = format(right_half2_int, '04b')

    # 将经过一次交换的内容合并（最后一轮不发生交换）
    final_output = right_half2 + right_half1
    # 最终置换（利用IP1）
    ip1_output = ''
    for i in IP1:
        ip1_output += final_output[i - 1]

    return ip1_output
```

4.2.4 二进制解密算法

1、原理：解密过程与加密过程基本一致，只有轮密钥的顺序有变化

2、代码演示：

```python
# 解码过程
5 个用法
def decrypt(ciphertext, key):
    # 1、生成密钥k1 k2
    key1, key2 = generate_keys(key, P10, P8)

    # 初始置换（利用IP）
    ip_output = ''
    for i in IP:
        ip_output += ciphertext[i - 1]

    # 多轮变换部分
    # 将密文分为左右两半部分
    right_half = ip_output[:4]
    left_half = ip_output[4:]

    # 第一轮加密
    # （1）将左半部分和子密钥key2进行f操作。
    left_result = f(left_half, key2)
    # （2）将f的结果（left_result）与右半部分进行XOR操作
    left_half1_int = int(right_half, 2) ^ int(left_result, 2)
```

```
    left_half1 = format(left_half1_int, '04b')

    # 第二轮加密
    # （1）将左半部分和子密钥key1进行f操作。
    left_result = f(left_half1, key1)
    # （2）将f的结果（left_result）与左半部分进行XOR操作
    right_half1_int = int(left_half, 2) ^ int(left_result, 2)
    right_half1 = format(right_half1_int, '04b')

    # 将经过一次交换的内容合并（最后一轮不发生交换）
    final_output = right_half1 + left_half1
    # 最终置换（利用IP1）
    ip1_output = ''
    for i in IP1:
        ip1_output += final_output[i - 1]

    return ip1_output
```

4.2.5 ASCII 码的加解密算法

ASCII 码的加解密算法与二进制加解密算法基本一致，只需要加入一个将 ASCII 码转化为二进制数/将二进制数转化为 ASCII 码的过程。故此处不再赘述。

4.2.6 暴力破解函数

1、原理：已知明密文对，寻找可能的密钥空间

2、代码演示：

（1）创建暴力破解类（方便调用），创建暴力破解函数 force，接受一个明密文对列表的输入、一个起始密码值和一个终止密码值的输入。

```
class force:                                                    ⚠ 2
    def __init__(self):
        self.correct_keys = []
        self.lock = threading.Lock()

    # pairs：一个包含明密文对的列表
    # start：暴力破解的起始密码
    # end：暴力破解的结束密码
    2 个用法
    def force(self, pairs, start, end):
        local_correct_keys = []
        # 从start到end循环遍历可能的密钥
        for key in range(start, end):
            # 密钥格式化
            key_str = '{0:010b}'.format(key)
            for plaintext, ciphertext in pairs:
                decrypted = decrypt(ciphertext, key_str)
                if decrypted != plaintext:
                    break
            else:
                if len(key_str) == 10:
                    local_correct_keys.append(key_str)
        with self.lock:
            # 保存正确的密钥
            self.correct_keys.extend(local_correct_keys)
```

（2）单线程暴力破解函数，遍历密钥空间为 2 的 10 次方。

```
    # 单线程暴力破解，start是0，end是2的10次方
    1 个用法
    def single_thread_force(self, pairs):
        self.force(pairs,  start: 0, 2 ** 10)
        return self.correct_keys
```

（3）多线程暴力破解：根据线程的数量分配各个线程的任务（在不同的子密钥空间内遍历）。这里设置为 4 个线程，所以每个线程需要遍历的密钥空间为 2 的 8 次方。

```
    # 多线程暴力破解，均分任务
    1 个用法
    def multi_thread_force(self, pairs, num_threads):
        key_space = 2 ** 10
        step = key_space // num_threads

        futures = []
        # 通过线程池管理线程，减少创建和销毁线程的开销。最大线程为num_threads
        with ThreadPoolExecutor(max_workers=num_threads) as executor:
            for i in range(num_threads):
                start = i * step
                end = start + step if i < num_threads - 1 else key_space
                futures.append(executor.submit(self.force, pairs, start, end))

        for future in as_completed(futures):
            future.result()

        return self.correct_keys
```

（4）破解函数：可以处理输入多组明密文对的情况，可以根据逗号分割输入字符串并创建明密文对 pairs。进而分别计算单线程和多线程暴力破解的时间，与密钥一起返回。

```
    def decrypt(self, plaintext_str, ciphertext_str):
        # 分割输入字符串，用于有多组明密文的情况
        plaintexts = plaintext_str.split(",")
        ciphertexts = ciphertext_str.split(",")

        # 创建明密文对。zip:将明文列表和密文列表配对
        pairs = list(zip(plaintexts, ciphertexts))

        # 单线程暴力破解计算时间
        start_time = time.time()
        single_result = self.single_thread_force(pairs)
        single_time = time.time() - start_time

        # 清空密钥列表（否则会显示两遍）
        self.correct_keys.clear()

        # 多线程暴力破解计算时间
        start_time = time.time()
        multi_result = self.multi_thread_force(pairs, num_threads=4)
        multi_time = time.time() - start_time

        # 分别得到单线程和多线程暴力破解的各项值
        return {
            "single_thread": {
                "keys": single_result,
                "time": single_time
            },
            "multi_thread": {
```

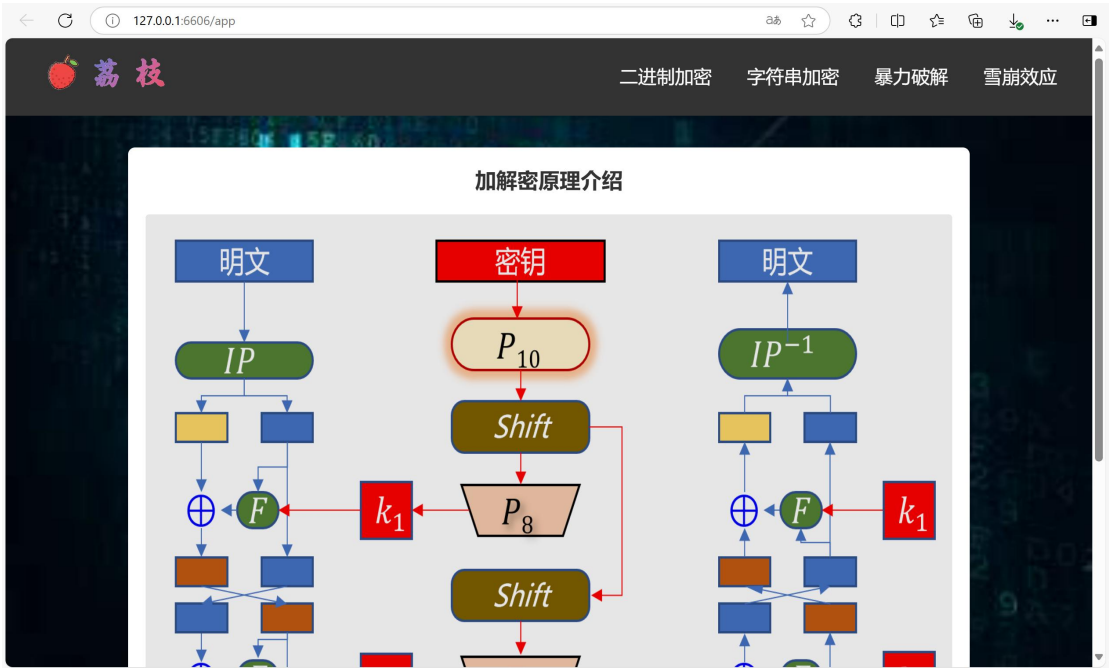
5. 基础功能及界面

5.1 启动动画

该动画页面主要显示欢迎信息和关于开发者团队的信息，可以帮助大家了解我们开发者团队。

5.2 原理讲解页面

在该页面内可以看到 DES 加解密算法的基础原理。



5.3 二进制加密页面

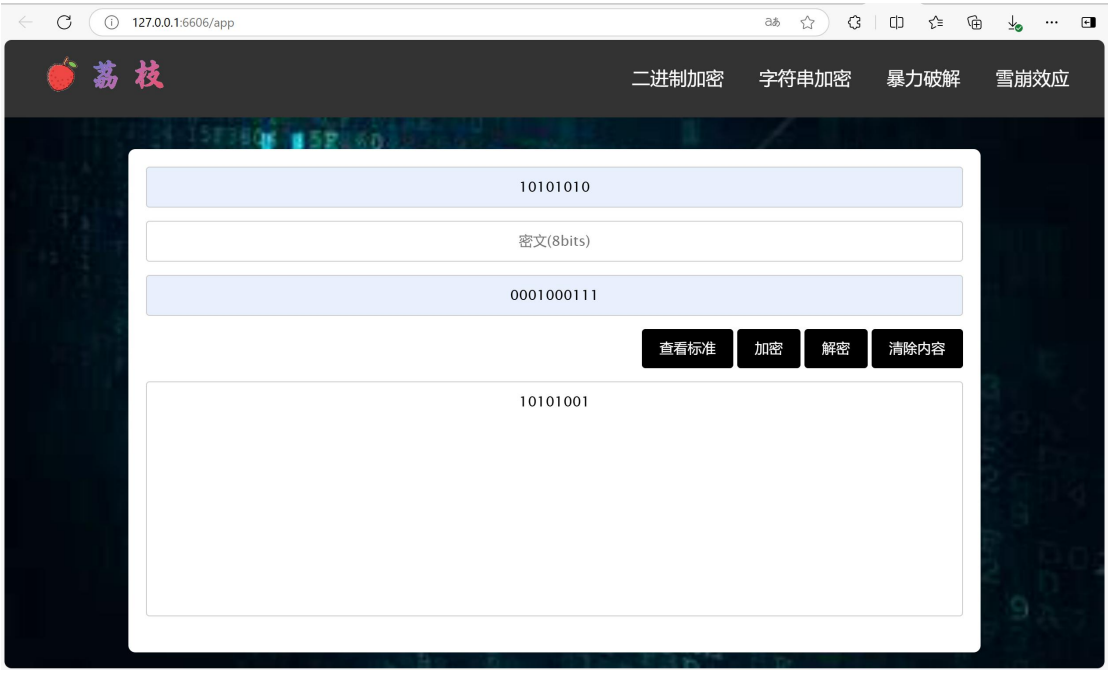
该页面提供了二进制加解密功能。

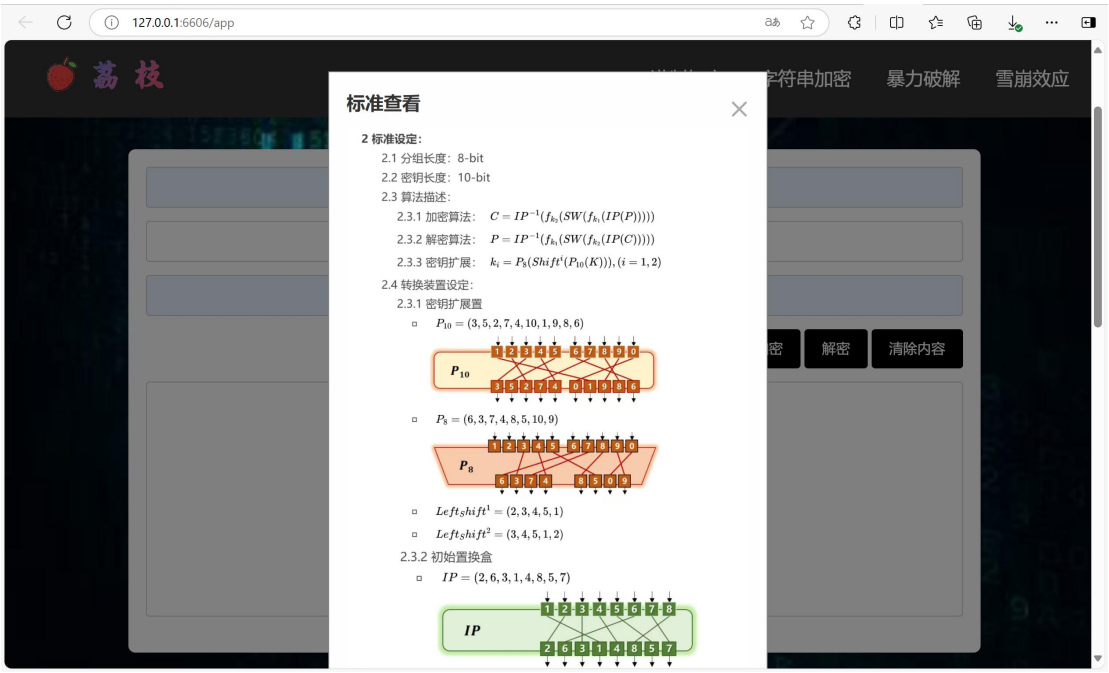
1、用户可以在明文文本框和密钥文本框分别输入 8-bits 和 10-bits 的明文和密钥，点击“加密”按钮，执行加密操作。

2、用户可以在密文文本框和密钥文本框输入 8-bits 和 10-bits 的密文和密钥，点击“解密”按钮，执行解密操作。

3、用户可以点击“查看标准”按钮进入模态窗口，查看该 DES 算法中的标准数据信息，同时下载相关文档以获取数据。

4、用户可以点击“清除内容”按钮，清除本页面文本框中的内容。





5.4 ASCII 码加密页面

基础功能与二进制加密页面类似。

1、用户在文本框输入明文为 1 byte 字符串，密钥为 10-bits 二进制数的内容，点击“加密”按钮，对字符串进行加密操作。

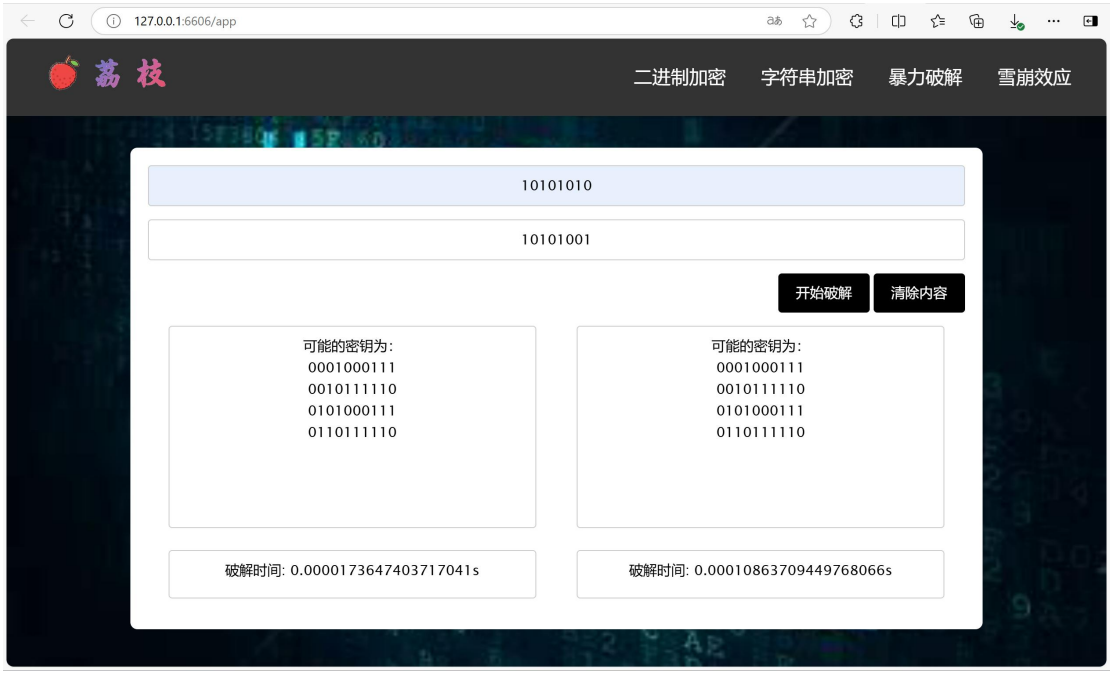
2、用户在文本框输入密文为 16-bits 二进制数，密钥为 10-bits 二进制数的内容，点击“解密”按钮，解密为 ASCII 码内容。

3、加解密结果文本框会同时输出二进制表示和 ASCII 码表示。

5.5 暴力破解页面

1、该页面接收输入 8-bits 的明密文对，点击“开始破解”按钮，可以分别利用单线程和多线程的方式破解可能密钥并显示。

2、页面以计时器动画的形式展示整个破解的过程，并会在破解完成后显示最终破解时间。（由于目前电脑破解速度较快，动画可能不容易被很好的观察到）



6. 功能扩展

6.1 新功能——雪崩效应

6.1.1 雪崩效应

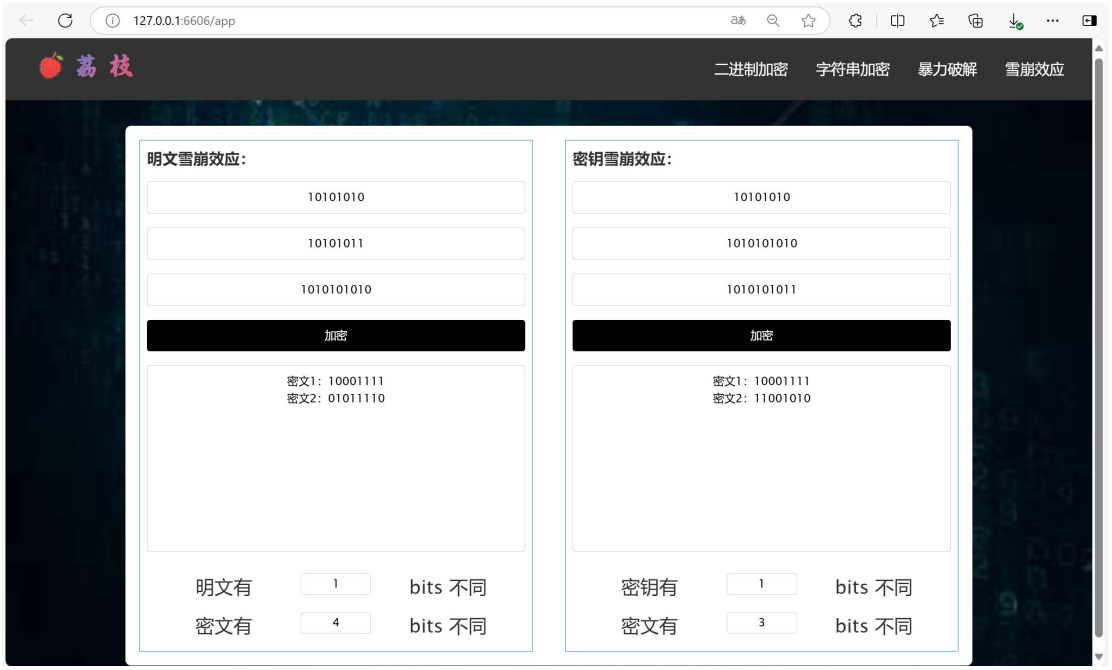
雪崩效应：明文或密钥输入发生微小变化，密文输出就会截然不同。

6.1.2 雪崩效应的实现

本系统的创新功能为雪崩效应的检验。可以帮助用户验证在 DES 算法加密系统中雪崩效应的出现。主要功能为在加密的时候可以显示两个明文和对应密文中不同的 bit 数目。经过检验我们发现，在 DES 算法中，雪崩效应确实存在，而且随着加密内容长度的增加，雪崩效应会更加明显。这个特性可以帮助加密系统消除密文的统计特征，使得破译难度更大。

6.2 功能展示与代码实例

可以看到该示例中，相同密钥条件下，明文 1-bit 的差异，密文会产生 4-bits 不同。相同明文条件下，密钥 1-bit 的差异，密文也会产生高达 3-bits 的不同。


127.0.0.1:6606/app
荔枝
二进制加密
字符串加密
暴力破解
雪崩效应
明文雪崩效应:
10101010
10101011
1010101010
加密
密文1: 10001111
密文2: 01011110
明文有 1 bits 不同
密文有 4 bits 不同
密钥雪崩效应:
10101010
1010101010
1010101011
加密
密文1: 10001111
密文2: 11001010
密钥有 1 bits 不同
密文有 3 bits 不同