2020 秋 - 密码学课程设计

目录

1 设计摘要

> 本课程设计实现了`AES-128、AES-256、DES、RC4`算法，并结合使用了`MySQL`数据库和`SHA-256`算法实现了**文件加密，文件传输，哈希校验、密钥管理**的功能。其中所有使用的加密算法均完全由本人编写，未调用第三方库。相关数据的存储则采用了`MySQL`数据库进行，在加解密过程中尝试使用了**多线程并行**技术，一定程度上抵消了`java`语言速度上的不足。

1.1 实验设备(环境)要求

本课程设计全程使用`java`语言编写，使用`jdk-14`进行编译

前端`UI`设计使用`java`自带的`swing`框架，支持`Windows、macOS、Linux`等跨平台运行。

1.2 功能模块摘要

本课程设计各个模块支持的算法和功能略有不同，摘要如下

模块\ 支持	AES-128	AES-256	DES	RC4	多线程加速	SHA-256校验
文件加密	√	√	√	√	√	√
文件传输				√		√
密钥管理	√	√	√	√		

2 基本设计要求

> 设计采用了前后端分离的设计思想，所有加密算法使用统一格式的命名规范，通过同一个文件处理接口提供给前端`GUI`调用。加密算法只负责处理数据，由`swing`界面进行事件监听和调用。

2.1 前端功能

2.1.1 文件加密

启动文件加密主界面，在`windows10`平台下效果如图2-1-1：

图2-1-1

文件加密部分主要实现了`AES-128、AES-256、DES、RC4`4个加密算法对文件的加密，用户选择对应的选择按钮即可确定当前使用的加密算法。

选择好加密算法后，用户可以选择文件，这里设计了两种设置文件路径的方式：

1、用户直接在**确认文件路径**的输入框中输入文件路径

2、为了方便用户，设计时提供了文件选择界面，点击选择文件按钮即可在选择框中选择文件，如图2-1-2

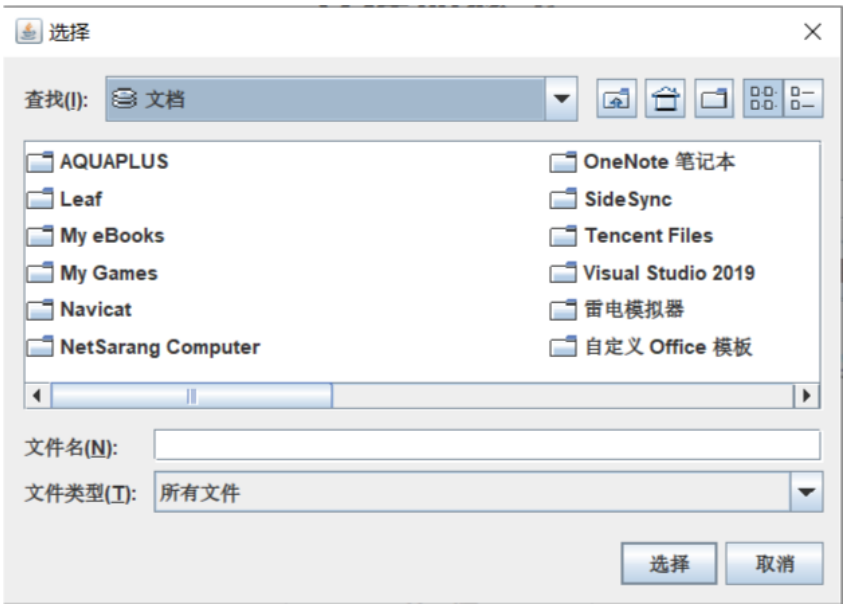


图2-1-2

同时，为了优化使用体验，设计了加解密进度条显示窗口，在加解密过程中会实时显示当前的进度，默认使用文件分块、4线程并行加速文件处理

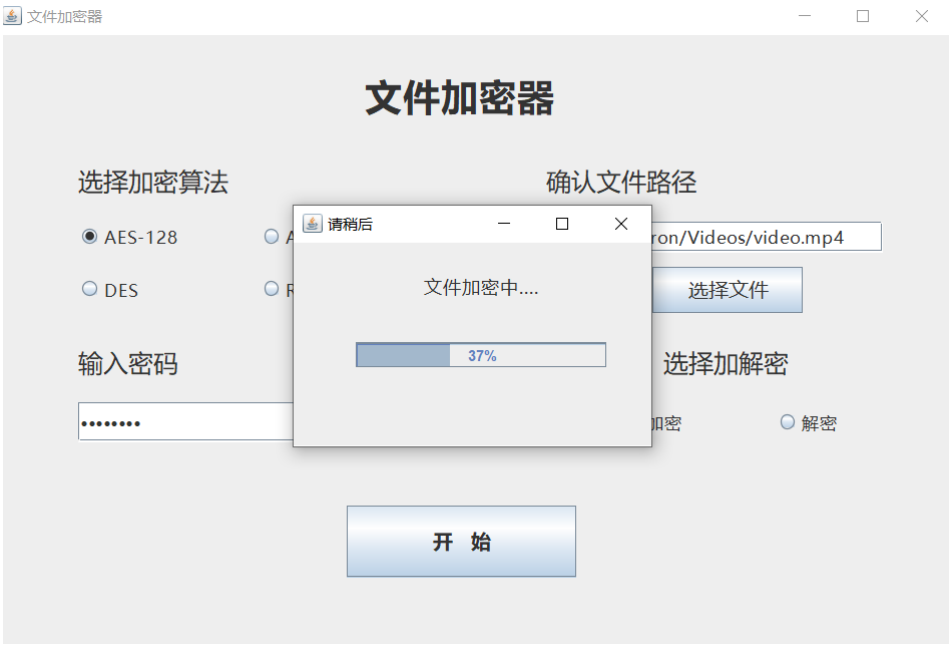


图2-1-3

为保证用户加解密文件的正确性，文件加密器添加了`MySQL`记录的数据表，当用户进行加密操作时，会计算源文件和加密后文件的`SHA-256`值，并添加到`MySQL`数据库中。

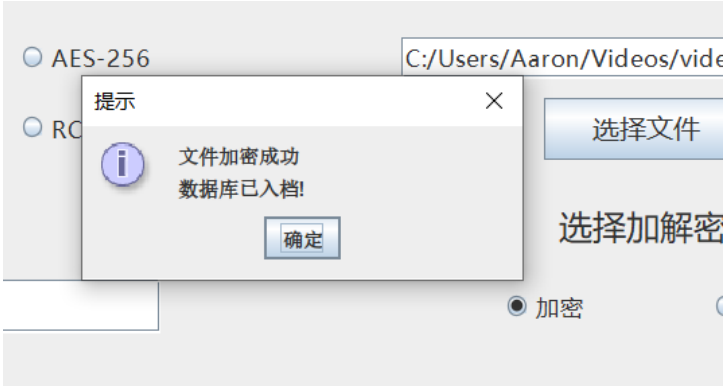


图2-1-4

当用户进行解密操作时，会计算当前加密文件的`SHA-256`值，并在数据库中寻找其对应源文件的哈希值，并将当前解密出的文件哈希值和找到的数据库记录的源文件哈希值进行比对，一并反馈给用户，用于检验解密文件是否是正确的。

图2-1-5

2.1.2 文件传输

启动文件传输`Server`接收方主界面，在`windows10`平台下效果如图2-1-6：

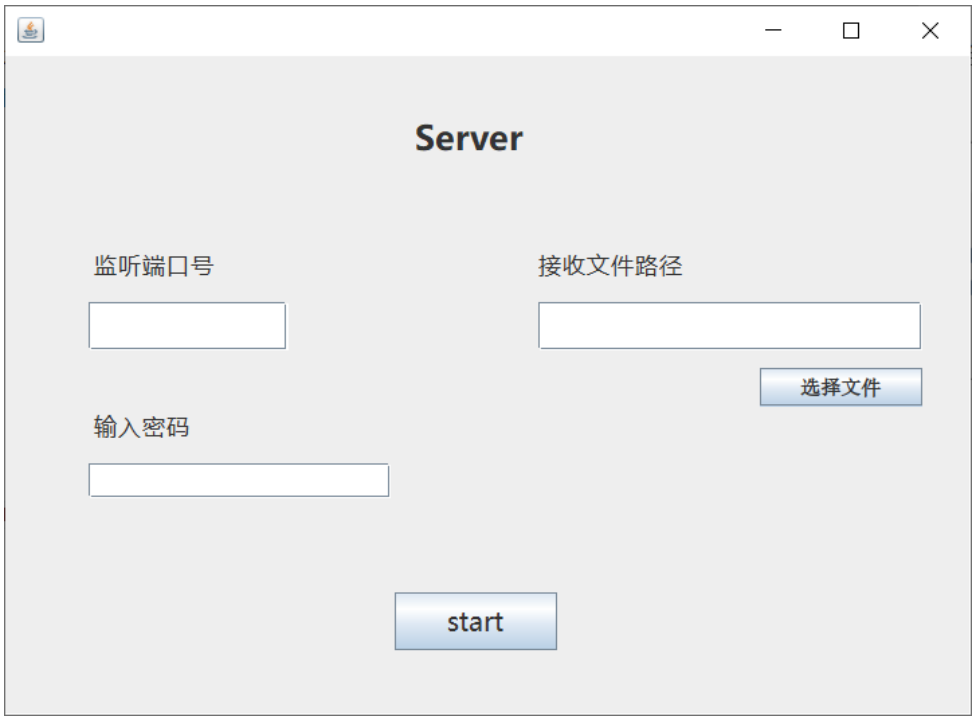


图2-1-6

启动文件传输`Client`发送方主界面，在`Windows10`平台下效果如图2-1-7：

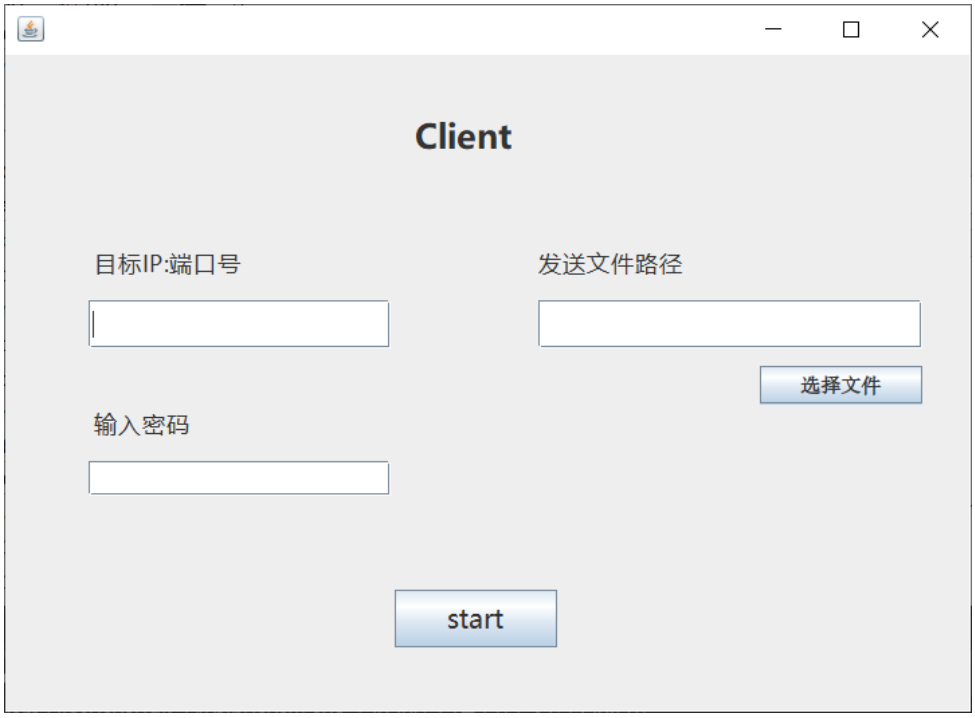


图2-1-7

传输文件模块使用`RC4`对数据流进行加密传输，`Client`输入密码后选择文件向`Server`指定端口发送连接请求，`Server`获取文件后同样需要输入密码进行文件解密，当且仅当两端密码相同时才可以正确的接收文件。

除此之外，`Client`还会在文件流传输完成后发送一份源文件的`SHA-256`哈希值，`Server`在解密后同样会对解密文件进行`SHA-256`计算，并比对两个哈希值，当且仅当完全一致时才会弹出正确的信息告诉用户传输成功，否则报错。

图2-1-8

2.1.3 密钥管理

密钥管理器默认使用`AES-128`加密密钥，也可以使用`DES、RC4、AES-256`进行密钥加密。密钥由`MySQL`数据库存储，获取`MySQL`数据库中加密密钥后需要根据用户输入的主密钥进行解密操作才可以显示，在解密成功后支持用户修改密码并重新存回数据库表中。默认一个密码描述只能够对应一个密码。

图2-1-9

2.2 后端

2.2.1 DES

> DES算法加密过程
>
> 1.64位密钥经子密钥产生算法产生出16个子密钥: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,将数据打乱重新排列并分成左右两半。
>
> 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。此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$K_2 \ldots K_{16}$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，两者合并再经过逆初始置换 ，将数据重新排列组合，便得到64位密文。至此加密过程全部结束。

DES算法模块代码文件组织如下:

```
|  test.txt
|
└─encrypt
    |  DECRYPT_des.java
    |  E.java
    |  ENCRYPT_des.java
    |  F.java
    |  P.java
    |  S.java
    |
    ├─box
    |      SBox.java
    |
    ├─keygenerate
    |      GenerateKey.java
    |
    └─permutation
            IP.java
            IP_inverse.java
```

位于包 `encrypt.keygenerate` 下的 `GenerateKey.java` 是子密钥产生，将输入的密钥转换为2进制数组，总共为64位。随后我们通过置换选择1将其中的8位奇偶校验位删除掉，随后不断地通过循环左移和置换选择2生成16个子密钥。

位于 `encrypt` 包下的 `E,F,P,S.java` 文件分别对应着DES算法的加密函数$F()$各个中间计算过程。如 `E.java` 就是扩展置换，代码如下：

```java
public class E {
     /** 选择运算矩阵 */
    static int[] EMatrix = {
        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
    };
    public static byte[] EWork(byte[] input,byte[] subkey){
        byte[] middleData=new byte[48];
        for(int i=0;i<48;i++)
            middleData[i]=(byte)(input[EMatrix[i]-1]^subkey[i]);
        return middleData;
    }
}
```

而 `F.java` 则是合并了$F()$中各个运算的加密函数整体过程，代码如下：

```java
public class F {
    public static byte[] function(byte[]input,byte[]subkey){
           byte[] middleData=E.EWork(input, subkey);
           middleData=S.SWork(middleData);
           return P.PWork(middleData);
    }
}
```

位于包`encrypt.box`下的文件`sbox.java`则是存储了`DES`的S盒数据，用于进行S盒置换。部分代码如下：

```java
public class SBox {
    public static int[][][] box={
            //1
            {
                {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}
            },
            //2
            {
                {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}
            },
        //....
    }
}
```

位于包`encrypt.permutation`下的`IP.java , IP_inverse.java`分别是IP和IP^{-1}置换，代码如下：

```java
public class IP {
     /** 初始置换矩阵 */
    static int[] IPMatrix = {
        58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17,  9, 1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7
    };
    /**
     * 初始置换IP
     * @return byte arrays
    */
    public static byte[] InitalPermutation(byte[] text){
        byte[] newText=new byte[64];
        for(int i=0;i<64;i++){
            newText[i]=text[IPMatrix[i]-1];
        }
```

```
        return newText;
    }
}
```

由于DES的运算是对合运算，因此DES算法的加密和解密过程除了子密钥的使用顺序相反之外，其他的部分都是相同的，因此在这里不做赘述。

2.2.2 AES

> AES算法的加密过程
>
> 1.首先进行密钥扩展与轮密钥选择。
>
> 2.随后进行10轮的循环迭代，循环迭代的过程包括：S盒变换、行移位变换、列混合变换和轮密钥加变换。
>
> 3.10轮循环迭代结束之后，进行最后一次迭代，最后一次迭代与前面的迭代相比去掉了列混合变换。

AES算法模块代码文件组织如下:

```
aes
|  FileAPI.java
|  SBoxes.java
|  Xtime.java
|
├─decrypt
|      DECRYPT.java
|      Inv_RoundFunction.java
|
├─encrypt
|      ENCRYPT.java
|      RoundFunction.java
|
└─keygenerate
        KeyExpansion.java
```

位于包`aes.keygenerate`下的文件`KeyExpansion.java`用于密钥扩展，将主密钥存放到密钥扩展的数组中，随后进行扩展。扩展的时候分两种情况，如果密钥的位置是4的倍数，则需要先进行$Rotl$变换和S盒变换，随后再异或一个轮常数$Rcon$才能与4个位置之前的字；而除此之外只需要将当前密钥与4个位置之前的密钥做与运算，就可以得到新的密钥

```
public class KeyExpansion {
    static int Rcon[]={
        0xFFFFFFFF,0x01000000,0x02000000,0x04000000,0x08000000,0x10000000,
        0x20000000,0x40000000,0x80000000,0x1B000000,0x36000000
    };
    public static int[] keyExpansion(byte[]key){
        //...
    }
    public static int SubByte(int input){
       //...
    }
    public static int Rotl(int input){//ABCD
        //...
```

```
    }

    public static int ByteArrayToInt(byte[] input){
       //...
    }
}
```

位于包`aes`下的`SBoxes,Xtime.java`分别是记录AES的S盒和乘法记录的静态表，用于节约计算时间。

```
public class Xtime {
    /** xtimeTable[i][j]表示byte为Ox(ij)的多项式xtimes后的结果 */
    public static byte[][]xtimeTable={
        {(byte)0x00, (byte)0x02, (byte)0x04, (byte)0x06, (byte)0x08,
(byte)0x0a, (byte)0x0c, (byte)0x0e, (byte)0x10, (byte)0x12, (byte)0x14,
(byte)0x16, (byte)0x18, (byte)0x1a, (byte)0x1c, (byte)0x1e},
        {(byte)0x20, (byte)0x22, (byte)0x24, (byte)0x26, (byte)0x28,
(byte)0x2a, (byte)0x2c, (byte)0x2e, (byte)0x30, (byte)0x32, (byte)0x34,
(byte)0x36, (byte)0x38, (byte)0x3a, (byte)0x3c, (byte)0x3e},
      ...
    };
public class SBoxes {
    public static byte[][] SBox={
        {(byte)0x63, (byte)0x7c, (byte)0x77, (byte)0x7b, (byte)0xf2,
(byte)0x6b, (byte)0x6f, (byte)0xc5, (byte)0x30, (byte)0x01, (byte)0x67,
(byte)0x2b, (byte)0xfe, (byte)0xd7, (byte)0xab, (byte)0x76},
        {(byte)0xca, (byte)0x82, (byte)0xc9, (byte)0x7d, (byte)0xfa,
(byte)0x59, (byte)0x47, (byte)0xf0, (byte)0xad, (byte)0xd4, (byte)0xa2,
(byte)0xaf, (byte)0x9c, (byte)0xa4, (byte)0x72, (byte)0xc0},
        {(byte)0xb7,
      ...
    };
```

、

`FileAPI`是用于单独测试`aes`算法正确性并且利用`aes`算法加密文件的测试类，此处略。

位于`aes.encrypt`下的文件`ENCRYPT.java`是AES加密的主体部分，`RoundFunction.java`则是加密对应的轮函数部分。同理，位于`ase.decrypt`下的文件对应的是AES解密主体部分，`Inv_RoundFunction.java`则是解密对应的逆轮函数部分。

2.2.3 RC4

> 相比DES、AES128/256算法，RC4算法的实现简单许多，其加解密过程如下：
>
> 1、初始化S和T
>
> 2、初始排列S
>
> 3、产生密钥流
>
> 4、异或

RC4算法模块代码组织如下：

```
rc4
|    Init.java
|    KeyStreamGeneration.java
|    Stable.java
```

其中`Init.java`负责初始化Stable部分，代码如下：

```java
public class Init {
    public static void InitStable(byte[]password){
        Stable.LinearFill();
        byte[]Rtable=new byte[256];
        int passwordLength=password.length;
        int j=0;
        for(int i=0;i<256;i++){
            Rtable[i]=password[i%passwordLength];
        }
        for(int i=0;i<256;i++){
            j=(j+Stable.S[i]+Rtable[i]+256)%256;
            swap(Stable.S,i,j);
        }
    }
    public static void swap(byte[]data,int i,int j){
        // for debug
        if(i>=data.length||j>=data.length){
            System.out.println("Error: out of index");
            System.exit(-1);
        }
        byte tmp=data[i];
        data[i]=data[j];
        data[j]=tmp;
    }
}
```

`Stable.java`描述了Stable，可进行线性填充

```java
public class Stable {
    static byte[]S=new byte[256];
    public static void LinearFill(){
        for(int i=0;i<256;i++)
            S[i]=(byte)i;
    }
}
```

`KeyStreamGeneration.java`则是产生密钥流的核心部分

```java
public class KeyStreamGeneration {
    public static byte[] GenerationKey(int needlength){
        byte []key=new byte[needlength];
        int i=0;
        int j=0;
        for(int t=0;t<needlength;t++){
            i=(i+1)%256;
            j=(j+Stable.S[i]+256)%256;
            Init.swap(Stable.S, i, j);
            key[t]=(byte)((256+Stable.S[i]+Stable.S[j])%256);
```

```
        }
        return key;
    }
}
```

2.2.4SHA-256

哈希函数位于包`hash`中，主体是`MessageDigestUtil.java`，其中主体方法如下：

```
public class MessageDigestUtil {
    public static String getDigest(String filePath,String algorithm){
        // 获取对应文件哈希字符串
    }
    /**
     * 将字节数组转换成16进制字符串
     * @param bytes 即将转换的数据
     * @return 16进制字符串
     */
    private static String bytesToHexString(byte[] bytes){
        //...
    }

    public static void main(String[]args) {
        // 测试
    }
}
```

2.2.5 文件处理接口

> 前文的设计思路提到，所有的加密算法统一使用一个文件处理接口，位于包`encrypttools`下，接口文件是`FileProcessingAPI.java`，在这个文件定义了对文件的加解密统一操作，并提供了**获取当前进度，完成状态，线程加速**的方法。前端在调用时，只需要传入待处理文件的路径及将要使用的算法和相关设置参数，即可对文件进行处理。

`FileProcessingAPI`类成员表2-2-1：

成员	属性	作用	返回值
`progressNow`	`protected static volatile int`	记录当前处理进度	null
`fileLength`	`protected static int`	记录处理的文件的长度	null
`alreadyread`	`public static volatile boolean`	是否已经将文件读入内存	null
`EncryptFile(String keyStr,String filePath,String outputPath,String type)`	`public static boolean`	根据参数加密文件	`boolean`
`DecryptFile(String keyStr,String filePath,String outputPath,String type)`	`public static boolean`	根据参数解密文件	`boolean`
`readFile(String filePath,boolean isNormal)`	`public static byte[]`	指定路径类型读取文件	`byte[]`
`writeFile(String filePath,byte[]outputData,boolean isNormal)`	`public static boolean`	指定路径类型导出文件	`boolean`
`ConvertBitToByte(byte[] input)`	`private static byte[]`	将bit转为byte	`byte[]`
`GetprogressNow()`	`public static int`	获取当前处理进度	`int`
`GetfileLength()`	`public static int`	获取当前处理的文件的长度	`int`

表2-2-1

2.2.6 JDBC+MySQL

> 设计中使用了MySQL数据库存储密钥信息和加解密文件的SHA-256哈希数据，因此需要使用JDBC进行数据库操作，同时也需要设计相应的数据库和数据表

负责`jdbc`操作的代码文件在`jdbc.DatabaseManager.java`中，提供了项目中需要的基本的数据库操作，`DatabaseManager`中的各个主要成员表2-2-2：

成员	属性	作用	返回值
`connection`	`java.sql.Connection`	进行`jdbc`连接	null
`statement`	`java.sql.Statement`	`jdbc`声明，用于执行SQL语句的	null
`connect()`	`public static void`	进行数据库连接	void
`close()`	`public static void`	关闭数据库连接	void
`query(String sql)`	`public static ResultSet`	数据库查询操作	`java.sql.ResultSet`
`execute(String sql)`	`public static void`	数据库操作	void

表2-2-2

数据库表的设计比较简单，哈希值存储表基本结构表2-2-3，2-2-4：

表**hash_check**

表项\属性	类型	长度	主键
`originalHash`	`varchar`	255	×
`encryptHash`	`varchar`	255	√

表2-2-3

表**password_table**

表项\属性	类型	长度	主键
`remark`	`varchar`	255	√
`password`	`varchar`		×

表2-2-4

2.3 前端界面

> 课程设计中，界面部分负责获取用户输入，监听按钮事件，根据用户的输入传递合适的参数调用后端的各个算法，并使用多线程加速和通知用户进度。界面和交互逻辑力争清晰友好，并且避免在前端界面代码中写入复杂的算法。

2.3.1 encryptGUI包

> `src.encryptGUI`包中存放着文件加密模块的前端界面类，负责进行文件加解密模块的展示和用户交互

GUIFrame

在这个类中定义了文件加解密界面的主体，并且添加了事件监听，其主要成员表格如表2-3-1：

成员	属性	作用	事件
`filePath`	`public String`	存储目标文件路径	null
`fileName`	`public String`	存储目标文件名	null
`ENorDEorNochoose`	`public int`	加密1解密2, 0为未选择	null
`iscompleted`	`public volatile boolean`	标记是否完成	null
`chooseFileButton`	`JButton`	选择文件按钮	弹出文件选择界面，引导用户选择文件，并返回文件路径给`filePath`
`startButton`	`JButton`	开始按钮	开始执行，在检查所用参数均设置成功后开启多线程处理文件和通知用户
`Encrypt extends Thread`	`class`	加密线程类	文件加密
`Decrypt extends Thread`	`class`	解密线程类	文件解密

表2-3-1

WaitingTips

> `WaitingTips`是加解密文件前端的进度条通知解密类，在用户开始处理文件时弹出，处理结束时关闭，其会连续刷新自己界面的进度条通知用户当前处理进度

其主要成员表格如表2-3-2：

成员	属性	返回值	作用
`InitEncryptShow()`	`public void`	void	设置加密界面大小、布局、开启定时器刷新进度条
`show()`	`public void`	void	显示界面，开启定时器检测是否完成，完成则关闭
`InitDecryptShow`	`public void`	void	设置解密界面大小、布局、开启定时器刷新进度条
`turnOff`	`private void`	void	强制关闭界面

表2-3-2

2.4 正确性

> 本课程设计使用的算法在实现时就已经进行了完备的正确性测试，下面利用后端算法保留的测试接口导出计算数据并与《密码学引论》上给出的例子进行比对，证明算法的正确性。

2.4.1 DES的正确性

修改DES源码，导出DES各个步骤的详细计算结果，运行截图如图2-4-1：

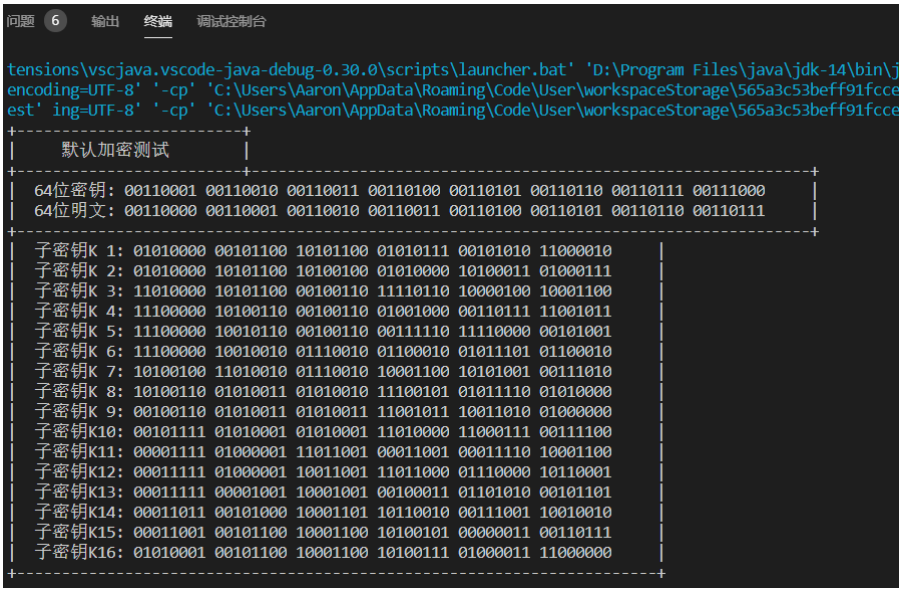


图2-4-1

导出数据为：

```
+--------------------------+
|     默认加密测试           |
+--------------------------+-------------------------------------------------
-------------+
|  64位密钥: 00110001 00110010 00110011 00110100 00110101 00110110 00110111
00111000      |
|  64位明文: 00110000 00110001 00110010 00110011 00110100 00110101 00110110
00110111      |
+--------------------------------------------------------------------------
-------------+
|  子密钥K 1: 01010000 00101100 10101100 01010111 00101010 11000010     |
|  子密钥K 2: 01010000 10101100 10100100 01010000 10100011 01000111     |
|  子密钥K 3: 11010000 10101100 00100110 11110110 10000100 10001100     |
|  子密钥K 4: 11100000 10100110 00100110 01001000 00110111 11001011     |
|  子密钥K 5: 11100000 10010110 00100110 00111110 11110000 00101001     |
|  子密钥K 6: 11100000 10010010 01110010 01100010 01011101 01100010     |
|  子密钥K 7: 10100100 11010010 01110010 10001100 10101001 00111010     |
|  子密钥K 8: 10100110 01010011 01010010 11100101 01011110 01010000     |
|  子密钥K 9: 00100110 01010011 01010011 11001011 10011010 01000000     |
|  子密钥K10: 00101111 01010001 01010001 11010000 11000111 00111100     |
|  子密钥K11: 00001111 01000001 11011001 00011001 00011110 10001100     |
|  子密钥K12: 00011111 01000001 10011001 11011000 01110000 10110001     |
|  子密钥K13: 00011111 00001001 10001001 00100011 01101010 00101101     |
|  子密钥K14: 00011011 00101000 10001101 10110010 00111001 10010010     |
|  子密钥K15: 00011001 00101100 10001100 10100101 00000011 00110111     |
|  子密钥K16: 01010001 00101100 10001100 10100111 01000011 11000000     |
```

```
+-------------------------------------------------------------------------+
+-----------------------------------------+
|  L 0: 00000000111111111111000010101010    |
|  R 0: 00000000111111110000000011001100    |
|                                           |
|  L 1: 00000000111111110000000011001100    |
|  R 1: 00010010100001110011011110110011    |
|                                           |
|  L 2: 00010010100001110011011110110011    |
|  R 2: 11100001100111001000011010001010    |
|                                           |
|  L 3: 11100001100111001000011010001010    |
|  R 3: 11010110001011101111011101100101    |
|                                           |
|  L 4: 11010110001011101111011101100101    |
|  R 4: 00011110111001010111111100100110    |
|                                           |
|  L 5: 00011110111001010111111100100110    |
|  R 5: 01011000010000001110001001011100    |
|                                           |
|  L 6: 01011000010000001110001001011100    |
|  R 6: 00011010011000000110100000101100    |
|                                           |
|  L 7: 00011010011000000110100000101100    |
|  R 7: 11010001011100100100110001010100    |
|                                           |
|  L 8: 11010001011100100100110001010100    |
|  R 8: 01101001101101100001001111111010    |
|                                           |
|  L 9: 01101001101101100001001111111010    |
|  R 9: 10101110100001011111100010000110    |
|                                           |
|  L10: 10101110100001011111100010000110    |
|  R10: 00010101101110011000100100011001    |
|                                           |
|  L11: 00010101101110011000100100011001    |
|  R11: 00010011011001010001111111011000    |
|                                           |
|  L12: 00010011011001010001111111011000    |
|  R12: 11110000111011000111011010001110    |
|                                           |
|  L13: 11110000111011000111011010001110    |
|  R13: 00100111110111000010101111001011    |
|                                           |
|  L14: 00100111110111000010101111001011    |
|  R14: 00011000111101010110001110010100    |
|                                           |
|  L15: 00011000111101010110001110010100    |
|  R15: 00110011111101101010110101000101    |
|                                           |
|  L16: 00110011111101101010110101000101    |
|  R16: 11010100000101101000101010100001    |
|                                           |
+-----------------------------------------+
+-------------------------------------------------------------------------
-------------+
|  64位密文: 10001011 10110100 01111010 00001100 11110000 10101001 01100010
01101101      |
```

```
+-------------------------------------------------------------------------------------------+
+--------------------------+
|      默认解密测试           |
+--------------------------+-------------------------------------------------------------------+
|  64位密钥: 00110001 00110010 00110011 00110100 00110101 00110110 00110111 00111000       |
|  64位密文: 10001011 10110100 01111010 00001100 11110000 10101001 01100010 01101101       |
+-------------------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------------------+
|  64位明文: 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111       |
+-------------------------------------------------------------------------------------------+
```

与《应用密码学》给出的测试例子完全一致

2.4.2 AES的正确性

修改AES-128部分的代码，导出AES各个步骤的详细计算结果，导出数据如下：

加密部分

```
63  63  b7  63  51  e0  63  7c  63  7c  63  b7  8e  63  63  89
 63  e0  63  89  51  7c  63  63  63  63  b7  7c  8e  63  63  b7
 17  94  c5  2f  26  6f  4e  2a  a8  1b  f1  89  76  5a  e9  fc
 4f  3  c7  fe  f  f9  d4  55  5b  f4  7c  e2  e5  a0  51  3
 84  7b  c6  bb  76  99  48  fc  39  bf  10  98  d9  e0  d1  7b
 84  99  10  7b  76  bf  d1  bb  39  e0  c6  fc  d9  7b  48  98
 c8  e6  b0  e8  5c  c0  a6  99  73  4f  51  8e  f4  6f  81  68
 bf  1d  a4  e5  2  ad  28  eb  de  cd  52  97  ca  17  3a  8e
 8  a4  49  d9  77  95  34  e9  1d  bd  0  88  74  f0  80  19
 8  95  0  19  77  bd  80  d9  1d  f0  49  e9  74  a4  34  88
 ad  20  b6  bf  6b  54  a1  d  91  d4  5f  57  a3  f3  3b  7
 62  31  2c  0  fa  28  b5  c0  ad  2a  48  83  a1  75  97  35
 aa  c7  71  63  2d  34  d5  ba  95  e5  52  ec  32  9d  88  96
 aa  34  52  96  2d  e5  88  63  95  9d  71  ba  32  c7  d5  ec
 d7  a2  9b  b4  85  1c  66  dc  46  9d  3f  27  f  2f  6b  87
 54  22  22  7c  97  e0  cb  d9  68  9f  85  f6  23  ab  7d  64
 20  93  93  10  88  e1  1f  35  45  db  97  42  26  62  ff  43
 20  e1  97  43  88  db  ff  10  45  62  93  35  26  93  1f  42
 ac  18  31  90  92  2f  86  87  8a  1a  45  54  bf  78  4d  62
 60  df  99  29  4c  14  83  3b  7a  23  fa  39  63  c5  e4  ec
 d0  9e  ee  a5  29  fa  ec  e2  da  26  2d  12  fb  a6  69  ce
 d0  fa  2d  ce  29  26  69  a5  da  a6  ee  e2  fb  9e  ec  12
 4d  86  39  3b  f4  7b  29  65  52  46  86  e2  aa  e1  90  40
 db  92  88  4  bc  54  9d  e6  ea  50  8d  c  ce  4a  32  20
 b9  4f  c4  f2  65  20  5e  8e  87  53  5d  fe  8b  d6  23  b7
 b9  20  5d  b7  65  53  23  f2  87  d6  c4  8e  8b  4f  5e  fe
 e3  a9  e1  d8  ee  54  7d  20  3e  e9  4b  87  7c  9  61  70
 57  87  80  a4  12  55  a8  df  7a  fe  95  96  5c  b5  1d  1
 5b  17  cd  49  c9  fc  c2  9e  da  bb  2a  90  4a  d5  a4  7c
 5b  fc  2a  7c  c9  bb  a4  49  da  d5  cd  9e  4a  17  c2  90
```

```
 ff  ba  77  c3  b2  1a  fa  cd  98  b9  37  4a  ff  a9  69  30
 ae  84  b5  8  1f  25  ed  f9  71  91  fe  6f  36  3d  dc  64
 e4  5f  d5  30  c0  3f  55  99  a3  81  bb  a8  5  27  86  43
 e4  3f  bb  43  c0  81  86  30  a3  27  d5  99  5  5f  55  a8
 6a  f  73  35  b5  78  6  3c  78  10  85  25  16  ec  13  4e
 2  e4  91  23  70  ac  f3  1e  54  ec  b9  22  f3  84  9a  1d
6c dd 59 6b 8f 56 42 cb d2 3b 47 98 1a 65 42 2a
```

解密部分：

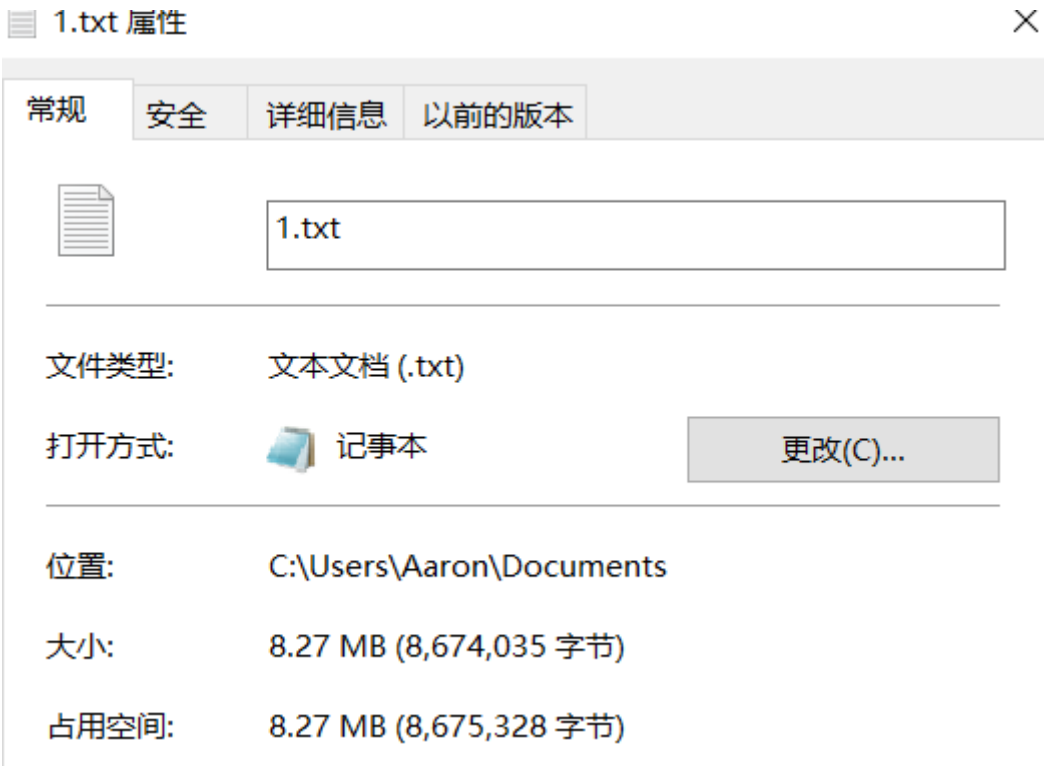
```
0x0 0x1 0x0 0x1 0x1 0xa1 0x98 0xaf 0xda 0x78 0x17 0x34 0x86 0x15 0x35 0x66
```

同样与《应用密码学》给出的测试例子完全一致，由此可以证明算法是正确的。

3 扩展设计

> 算法在模块的使用中使用了多线程技术、少量空间换取大量时间的策略，一定程度上弥补了java语言在运行速度上的不足。同时通过优化编译和简化组件，减少了文件大小。前后端分离的设计使得课程设计的二次开发相对容易

3.1 速度

> 项目统一使用一个大小为2.55GB的`.rar`压缩包和一个随机生成的txt作为测试文件对速度进行测试
>
> 
>

3.1.1 DES

在本项目的DES算法测试中，DES的速度较慢，对于压缩包花费了`3min30s`的时间，速度约为$12.43MB/s$，但是处理小型文本文件反应较为灵敏，基本上普通的文本文件可以快速加解密。

考虑到DES本身算法安全性已经降低，在本设计中DES主要用于密码管理模块，并且在使用前给予了用户风险提示。

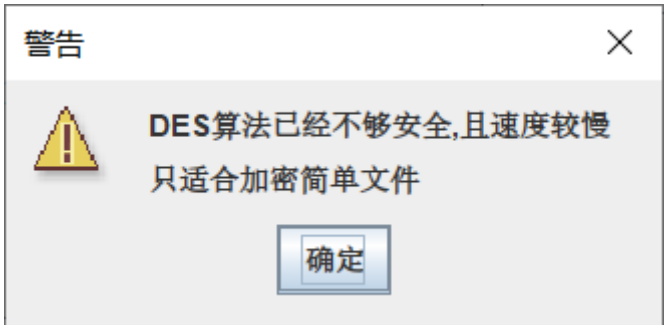


图3-1-1

这样的速度可以接受

3.1.2 AES

AES算法支持多线程加速，在测试中无论是AES-128还是AES-256均显著优于DES，128和256两者速度差别不大。对于小型文件$(100MB)$,AES算法可以在小于1s的时间内完成并通知用户。多线程整体速度大概在$120MB/s$左右，单个线程大约$30MB/s$

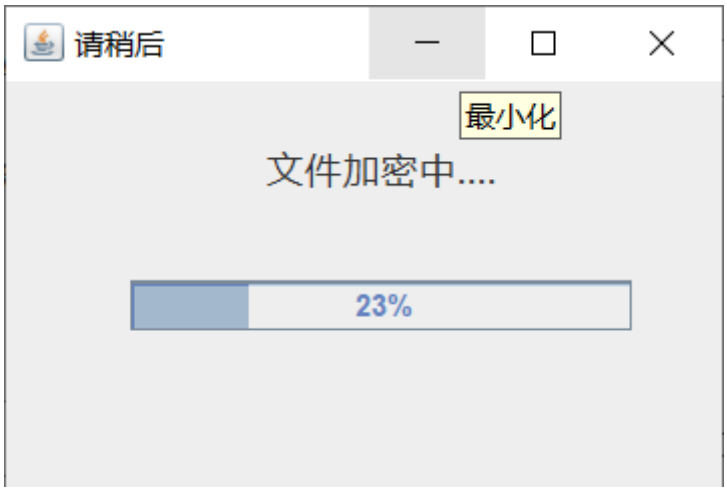


图3-1-2

处理完成

图3-1-3

3.1.3 RC4

RC4算法在本课程设计中速度最快，虽然不支持多线程加速，但是速度仍然显著快于AES，对于小于$200MB$的文件，RC4算法几乎在未1的文件，RC4算法几乎在未弹出进度条提示框时就已经完成加解密运算。单线程速度大概在$210MB/s$左右

3.2 程序占用空间

> 在Eclipse中监视JVM占用空间，计算加解密过程中程序的内存占用

在程序没有运行的时候，可以看到此时JVM占用的内存为138M

图3-2-1

打开程序，执行到加解密操作部分，可以看到此时JVM占用的内存是148M

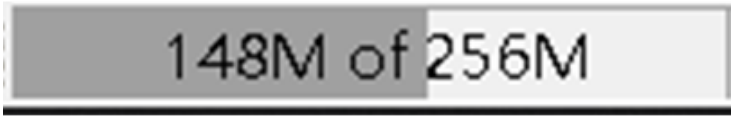


图3-2-2

可见程序至多占用了$10MB$左右的内存空间，这个数据是较为不错的。

3.3 程序文件大小

首先查看源代码`src`文件夹，可以看到所有代码文件总和实际大小为125KB，考虑到整个项目实现了文件加解密，通信加密，数据库交互等功能，代码体积总量是可以接受的，继续优化结构，删除`debug函数`可以进一步压缩。

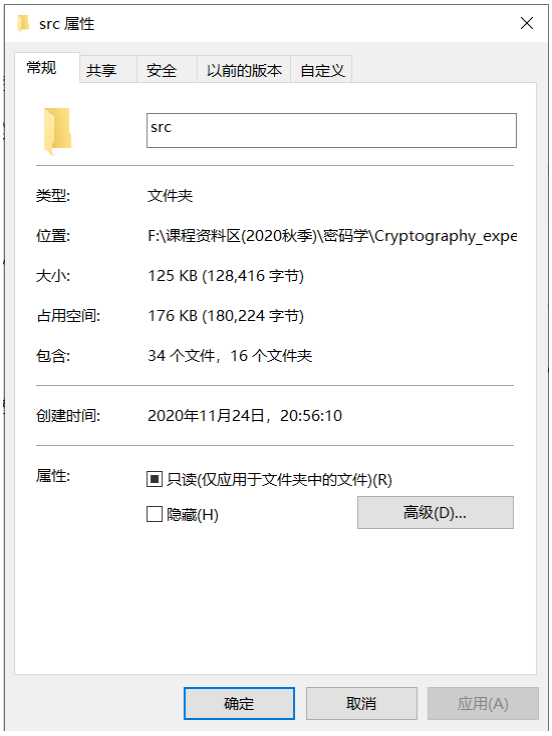


图3-3-1

而通过`eclipse`导出可执行文件，可以发现导出的可执行`.jar`文件体积也比较小，只有62KB，十分小巧，这是一个不错的数据。

图3-3-2

3.4 安全功能多样化

本课程设计实现了DES、AES族、RC4算法的文件与流的加解密操作，并使用了`SHA-256`进行自校验，引入了`MySQL`数据库管理相关信息，提升了程序日志和存储信息的安全性。多样的算法可以给用户更多的选择，而哈希校验则保证了文件解密的正确性，保证了传输文件的真实性。`MySQL`数据库自带的安全管理保护了程序存储的信息无法被他人恶意读取，提升了存储信息的安全性。

3.5 算法可扩展性

本课程设计使用了前后端分离的设计思想，算法可扩展性较好，所有的算法均有统一的接口规范和命名格式，同时各个算法也提供了自己的`FileAPI`可直接调用。`ENCRYPT() DECRYPT()`是每个算法的主体，可以直接迁移。此外，果我们需要对现有的本地加密的算法进行添加，或者对文件认证的算法进行添加，只需要将写好的算法插入到现有的框架即可，连用户界面的格式都不需要进行修改，只需要添加菜单栏的项目即可。

项目提供了统一的`FileProcessAPI`接口，只需要将写好的算法添加到这个接口中，就可以在界面里根据标准的参数格式进行调用。统一的接口也便于项目的二次开发，在此基础上开发文件签名认证等功能是较为简单的，并不需要对项目进行大幅改造，只需要将要添加的模块加入即可。删除模块也很简单，将对应的包和接口对应的处理信息删去即可。

3.6 执行速度统计

本项目界面提供了统一的`WaitingTips`类，在文件进行加解密操作时会实时刷新进度条，在结束后会弹出完成提示框并且通知用户时间，虽然完成提示框统计是每$200ms$一检测，可能会造成时间统计有最多$200ms$的误差，但是动态的进度条能够更加直观的让用户感受到文件处理的速度。

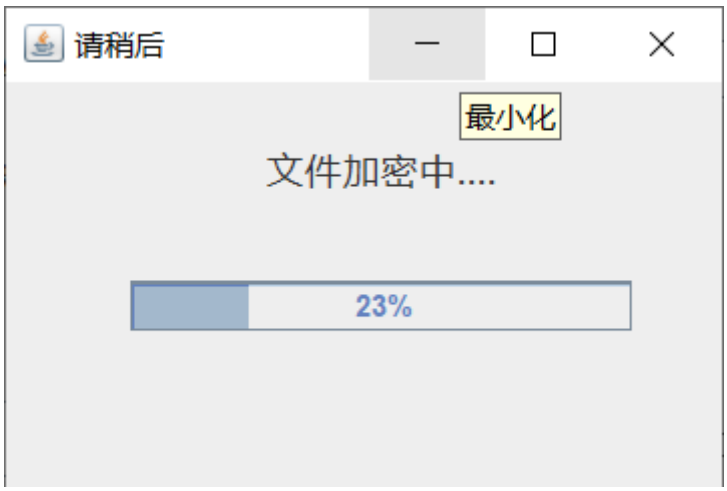


图3-6-1

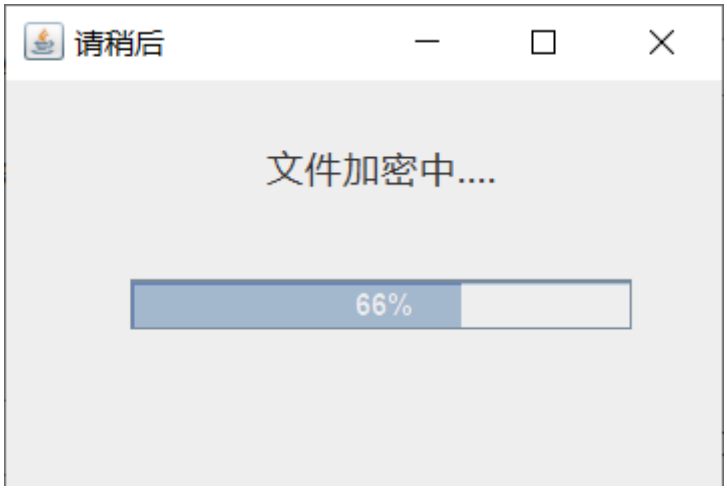


图3-6-2