北京郵電大學
Beijing University of Posts and Telecommunications

网络信息安全密码算法编程

期中作业

所在学院：北京邮电大学网络空间安全学院

所在班级：2022211802

学生学号：2020211837

学生姓名：王鑫

任课老师：马兆丰 老师

2025 年 4 月 13 日

北京邮电大学

《网络信息安全密码算法编程》评分表

选题基本信息			
选题名称	网络信息安全密码算法编程		
组长姓名	王鑫	联系电话	15909814268

项目评分准则				
序号	评分指标	评分要求	基准分值	实际得分
1	功能实现	➢ 核心技术功能覆盖全面。 ➢ 关键技术实现严谨正确。 ➢ 对应功能性能经过测试。	70	
2	用户界面	➢ 系统用户界面专业友好。 ➢ 系统设计布局色彩合理。 ➢ 人机交互体验感觉良好。	10	
3	报告格式	➢ 项目技术报告须有大纲。 ➢ 项目技术报告条理清晰。 ➢ 作业内容格式规范合理。	10	
4	其他要求	➢ 涉及的图公式不可截图。 ➢ 作业报告须有参考文献。	10	
项目得分			100	

项目人员得分				
项目成员	姓　　名	班级+学号	任务分工描述	个人得分
组长	王鑫	2022211802 班 +2020211837	密码算法实现、前后端交互、接口设计与调用、前端界面优化	
组员				
组员				
组员				
组员				

目录

网络信息安全密码算法编程

王鑫

北京邮电大学网络空间安全学院，北京，100876

摘要：本项目围绕网络信息安全中的密码算法实现与演示，采用 Java 语言结合 Spring Boot 框架构建后端服务，使用 HTML + CSS + JavaScript 构建前端页面，实现了对称加密、哈希函数、编码算法以及公钥加密算法四大类共计十余种主流算法的可视化加解密平台。系统采用前后端分离架构，用户可通过网页交互式地选择算法、输入明文/密钥等参数，实时获取加密/解密/签名等结果。项目同时支持接口调用、参数传输和结果返回，便于其他系统对接测试。平台界面简洁直观、功能完整，适用于教学演示、安全实验与算法验证等场景。

关键词：网络信息安全、对称加密、哈希算法、编码转换、公钥密码、Spring Boot、可视化演示平台、Java

一、 网络信息安全密码算法的发展背景与研究意义

随着全球数字化进程的加速推进，信息作为核心生产要素在各类业务系统、工业控制、金融通信等关键领域中呈爆发式增长。与此同时，网络攻击、数据泄露、身份伪造等安全事件频发，信息安全已从“功能保障”上升为“战略优先”。在此背景下，密码算法作为网络信息安全的底层核心技术，正承担着前所未有的重任。（本项目已上传至 https://github.com/Wangbeifang404/CryptoAlgo.git）

1.1 发展背景

随着全球数字化进程的加速推进，信息作为核心生产要素在各类业务系统、工业控制、金融通信等关键领域中呈爆发式增长。与此同时，网络攻击、数据泄露、身份伪造等安全事件频发，信息安全已从“功能保障”上升为“战略优先”。在此背景下，密码算法作为网络信息安全的底层核心技术，正承担着前所未有的重任。

1.2 研究意义与行业驱动力

密码算法的研究与应用不仅是保障信息安全的核心，更是国家层面构建数字主权与信任体系的重要基石。当前行业中普遍面临以下驱动力：

（一） **法律与合规驱动**：如《数据安全法》《个人信息保护法》《关键信息基础设施保护条例》等，对加密算法提出明确技术要求；

（二） **产业自主可控要求**：推动国产密码算法（如 SM 系列）在金融、政务、国防等领域落地；

（三） **可信计算与区块链等新兴场景需求**：对椭圆曲线加密（ECC）、哈希签名、密钥派生算法提出了多元适配要求；

（四） **安全能力可视化与可测性需求**：对密码算法可交互、可配置、可验证的演示系统提出实际应用价值。

1.3 难点与挑战

尽管密码算法本身已有完备的理论体系，但在工程实践层面仍面临多方面挑战：

（一） **算法多样化与接口统一的矛盾**：对称、非对称、哈希等算法功能差异大，难以用统一的方式封装调用；

（二） **密钥管理与用户交互的复杂性**：加密安全依赖密钥，但过多暴露用户输入会降低使用体验；

（三） **性能与安全性的权衡**：部分算法（如 PBKDF2）为提升抗破解能力需高轮次计算，可能影响运行效率

（四） **前后端联调与平台兼容性问题**：需在 Web 端直观展示加密过程，且保证后端接口的正确性与安全性。

基于上述背景与挑战，本项目从教学与实验角度出发，构建一个涵盖多类密码算法的可交互演示平台，既能够对经典密码学理论进行工程化实现，也为理解算法特性与对比应用效果提供良好的验证环境。

二、 系统功能与架构设计

2.1 系统功能概述

本项目旨在构建一个基于 Web 的网络信息安全密码算法演示平台，用户可通过简洁直观的前端界面，选择不同类型的密码算法，输入明文/密钥等参数，在线执行加密、解密、哈希、签名等操作，实时获取执行结果。平台支持以下核心功能：

（一） **算法分类展示：**通过一级目录划分对称加密、哈希算法、编码算法、公钥密码四大类服务；

（二） **算法选择与参数输入：**用户可通过二级菜单选择具体算法，并输入原文、密钥、签名等相关参数；

（三） **双栏结果展示：**采用左右并列结构，一侧输入原文，另一侧实时展示结果（密文/哈希值/签名）；

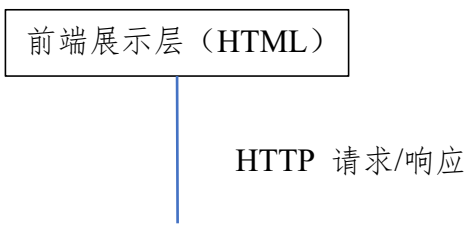
（四） **正向与反向操作支持：**对加密算法支持加解密切换，对哈希/签名支持结果验签；

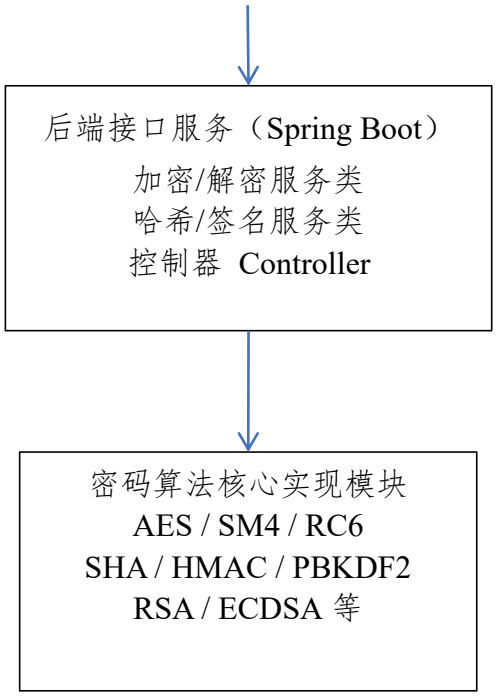
（五） **Web API 接口调用：**后端提供标准化接口，支持第三方系统调用测试，便于集成与扩展。

通过以上功能，平台不仅具备较强的演示性和可交互性，也为教学、测试与算法理解提供了良好的支持工具。

2.2 系统架构设计

系统整体采用“前后端分离架构”，后端基于 Spring Boot 构建服务接口，前端采用原生 HTML + CSS + JavaScript 实现用户交互界面，数据通过 HTTP 接口进行通信。整体架构如下图所示：





模块说明：

（一） **前端界面模块：**实现算法选择、参数输入、结果展示、按钮触发等交互逻辑；

（二） **接口控制模块：**统一负责接收前端请求、路由到对应服务类；

（三） **服务实现模块：**按照算法类型划分为 Symmetric、Hash、Encoding、Asymmetric 四个子模块，分别实现加解密逻辑；

（四） **核心算法模块：**封装了所有具体的密码算法调用与实现逻辑，部分依赖 Bouncy Castle 加密库。

此种模块化架构既保证了系统的功能完整性，又具备良好的可维护性与扩展性。后续若需添加新算法，仅需新增对应服务类与接口路由即可快速集成。

通过上述功能设计与系统架构，本平台实现了多个密码算法的统一展示、灵活调用和前后端协同运行，为用户提供了一个稳定、直观、安全的实验与演示环境。

三、 加解密算法及后端代码实现

在本章中，我们将介绍实现的四大类密码算法服务（对称加密、哈希算法、编码算法、公钥密码算法），并附上相关的后端代码实现。每个 service 内部都

通过特定的算法实现了加解密、哈希、编码等功能，并暴露标准化的接口供前端或其他系统调用。

3.1 对称加密算法服务

对称加密算法使用相同的密钥进行加密和解密。常见的对称加密算法包括 AES、SM4 和 RC6。

（1） AES（Advanced Encryption Standard）

AES 是当前最广泛应用的对称加密算法之一，也是美国国家标准技术研究院（NIST）推荐的标准加密算法。它取代了老旧的 DES 算法，主要用于数据的加密和解密，广泛应用于银行、电子支付、无线通信等领域。

AES 算法原理：AES 算法基于替代-置换网络（Substitution-Permutation Network, SPN） 的结构，由多轮加密组成。其基本原理是通过一系列的代换（Substitution）和 置换（Permutation）操作来将明文转化为密文。AES 使用的是块加密，每次处理固定大小的 128 位数据块。

密钥长度：AES 支持三种密钥长度：128 位、192 位、256 位，分别对应 10、12、14 轮加密操作。

轮函数（Rounds）：每一轮加密包括：字节替代（SubBytes）、行移位（ShiftRows）、列混合（MixColumns）、加密密钥加（AddRoundKey）等步骤。

主要步骤：

AddRoundKey：每一轮的密文与轮密钥进行按位异或。

SubBytes：通过查找表（S-box）替换字节，增加密码的复杂度。

ShiftRows：每一行字节循环左移，增加置换效果。

MixColumns：列级的混合，进一步加密。

AddRoundKey：与相应的轮密钥进行异或运算。

应用场景：

AES 被广泛应用于文件加密、虚拟专用网络（VPN）等需要快速加密的场景。

代码实现：

```java
import org.springframework.stereotype.Service;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

@Service 2 usages
public class AES {
    public String encrypt(String input, String key) throws Exception { 1 usage
        SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), algorithm: "AES");
        Cipher cipher = Cipher.getInstance( transformation: "AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encrypted = cipher.doFinal(input.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }

    public String decrypt(String encrypted, String key) throws Exception { 1 usage
        SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), algorithm: "AES");
        Cipher cipher = Cipher.getInstance( transformation: "AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] decrypted = cipher.doFinal(Base64.getDecoder().decode(encrypted));
        return new String(decrypted);
    }
}
```

解析：SecretKeySpec: 用于构造 AES 的密钥，key.getBytes() 将用户提供的密钥字符串转为字节数组。

Cipher.getInstance("AES"): 通过 JCE（Java Cryptography Extension）库获取 AES 加解密算法实例。

cipher.init(Cipher.ENCRYPT_MODE, secretKey): 初始化加密模式，使用给定的密钥进行加密。

doFinal(): 执行加密（或解密）操作。doFinal 方法是实际执行加密或解密操作的地方。

Base64 编码/解码: 为了便于传输和展示，输出的加密密文和解密结果都使用 Base64 编码。

（2） SM4（国家商用密码算法）

SM4 是中国国家密码管理局发布的商用对称加密算法，属于国家商用密码标准，广泛应用于金融、电信等领域的加密通信和数据保护。它的设计理念与 AES 类似，但采用不同的加密轮数和结构。

SM4 算法原理：SM4 基于 Feistel 网络结构进行加密和解密，类似于 DES 算法，但不同之处在于 SM4 的加密轮数为 32 轮，每轮采用的是字节替换、移位和异或等操作。

密钥长度：SM4 使用 128 位密钥。

分组长度：SM4 对 128 位数据进行加密，和 AES 类似。

Feistel 网络：每轮加密由非线性替换、线性移位和轮密钥异或组成，确保加密的安全性。

加密步骤：

通过轮密钥生成 过程得到 32 个轮密钥。

每轮使用 Substitution（代换）、Permutation（置换）和 XOR（异或）等操作进行加密。

最终的输出密文由经过加密的多轮数据块组成。

应用场景：

SM4 主要用于国内的金融、政府通信等行业，也用于身份认证、数据保护 和数字签名等领域。

代码实现：

```java
import org.bouncycastle.crypto.engines.SM4Engine;
import org.bouncycastle.crypto.paddings.PaddedBufferedBlockCipher;
import org.bouncycastle.crypto.params.KeyParameter;
import org.springframework.stereotype.Service;
import java.util.Base64;

@Service  2 usages
public class SM4 {
    public String encrypt(String input, String key) throws Exception {  1 usage
        byte[] keyBytes = key.getBytes();
        PaddedBufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new SM4Engine());
        cipher.init( b: true, new KeyParameter(keyBytes));
        byte[] inputBytes = input.getBytes();
        byte[] output = new byte[cipher.getOutputSize(inputBytes.length)];
        int len = cipher.processBytes(inputBytes,  i: 0, inputBytes.length, output,  i2: 0);
        len += cipher.doFinal(output, len);
        byte[] encrypted = new byte[len];
        System.arraycopy(output,  srcPos: 0, encrypted,  destPos: 0, len);
        return Base64.getEncoder().encodeToString(encrypted);
    }

    public String decrypt(String encrypted, String key) throws Exception {  1 usage
        byte[] keyBytes = key.getBytes();
        PaddedBufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new SM4Engine());
        cipher.init( b: false, new KeyParameter(keyBytes));
        byte[] encryptedBytes = Base64.getDecoder().decode(encrypted);
        byte[] output = new byte[cipher.getOutputSize(encryptedBytes.length)];
        int len = cipher.processBytes(encryptedBytes,  i: 0, encryptedBytes.length, output,  i2: 0);
        len += cipher.doFinal(output, len);
        byte[] decrypted = new byte[len];
        System.arraycopy(output,  srcPos: 0, decrypted,  destPos: 0, len);
        return new String(decrypted);
    }
}
```

解析：

SM4Engine: SM4 算法引擎，是来自 BouncyCastle 库，实现了 SM4 算法的加解密操作。

PaddedBufferedBlockCipher: 与 RC6 类似，使用填充缓冲块密码加密器来处理数据，确保数据符合 SM4 的块大小要求。

KeyParameter: 将密钥（通过 key.getBytes() 转换）传入加密引擎。

加密过程：

cipher.init(true, new KeyParameter(keyBytes)) 初始化加密模式。

使用 cipher.processBytes() 和 cipher.doFinal() 对输入数据进行加密。

解密过程：

cipher.init(false, new KeyParameter(keyBytes)) 初始化解密模式，解密的步骤与加密类似。

（3） 1.3 RC6（RSA 加密扩展）

RC6 是由 RSA Security 公司设计的一种对称加密算法，是 RC5 的扩展版本。RC6 具有较高的加密效率，并且支持不同的密钥长度和加密轮数，是一个非常灵活的算法。

RC6 算法原理：RC6 算法也是基于 Feistel 网络结构进行加密和解密操作。它的基本思想是通过多个加密轮次（通常为 20 轮）和灵活的密钥长度（128 位、192 位、256 位）来对明文进行加密。

密钥长度：RC6 支持 128 位、192 位、256 位密钥长度，最大支持 2048 位密钥。

分组长度：RC6 也采用 128 位的分组进行加密。

Feistel 网络结构：每轮加密操作由 Substitution（代换）、Permutation（置换）和 XOR（异或）操作组成，与 AES 类似，但增加了更多的可配置性。

加密步骤：

轮密钥生成：通过密钥扩展过程生成一组轮密钥。

Feistel 运算：每一轮中，数据块经过加密变换，轮密钥与数据块进行异或。

最终结果是加密后的密文数据块。

应用场景：

RC6 适用于需要较高加密速度和灵活性要求的应用场景，比如虚拟私人网络（VPN），高安全性数据存储等。

代码实现：

```
import org.bouncycastle.crypto.engines.RC6Engine;
import org.bouncycastle.crypto.paddings.PaddedBufferedBlockCipher;
import org.bouncycastle.crypto.params.KeyParameter;
import org.springframework.stereotype.Service;
import java.util.Base64;

@Service 2 usages
public class RC6 {
    public String encrypt(String input, String key) throws Exception { 1 usage
        byte[] keyBytes = key.getBytes();
        PaddedBufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new RC6Engine());
        cipher.init( b: true, new KeyParameter(keyBytes));
        byte[] inputBytes = input.getBytes();
        byte[] output = new byte[cipher.getOutputSize(inputBytes.length)];
        int len = cipher.processBytes(inputBytes,  i: 0, inputBytes.length, output,  i2: 0);
        len += cipher.doFinal(output, len);
        byte[] encrypted = new byte[len];
        System.arraycopy(output,  srcPos: 0, encrypted,  destPos: 0, len);
        return Base64.getEncoder().encodeToString(encrypted);
    }

    public String decrypt(String encrypted, String key) throws Exception { 1 usage
        byte[] keyBytes = key.getBytes();
        PaddedBufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new RC6Engine());
        cipher.init( b: false, new KeyParameter(keyBytes));
        byte[] encryptedBytes = Base64.getDecoder().decode(encrypted);
        byte[] output = new byte[cipher.getOutputSize(encryptedBytes.length)];
        int len = cipher.processBytes(encryptedBytes,  i: 0, encryptedBytes.length, output,  i2: 0);
        len += cipher.doFinal(output, len);
        byte[] decrypted = new byte[len];
        System.arraycopy(output,  srcPos: 0, decrypted,  destPos: 0, len);
        return new String(decrypted);
    }
}
```

解析：

RC6Engine: 这是来自 BouncyCastle 加密库的 RC6 算法引擎，它实现了 RC6 加密标准的具体运算。

PaddedBufferedBlockCipher: 这是 BouncyCastle 提供的一个封装，允许我们在加密过程中对输入数据进行填充（padding），确保数据符合 RC6 算法的块大小要求。

KeyParameter: 用于初始化加密器时传递密钥，密钥通过 key.getBytes() 转换为字节数组。

加密流程：

初始化加密器：cipher.init(true, new KeyParameter(keyBytes))，true 表示加密模式。

加密数据：cipher.processBytes() 和 cipher.doFinal() 用来处理输入数据，最终返回密文。

解密流程：

与加密类似，只不过 init(false, ...) 表示解密模式。

3.2 哈希算法服务

（一） SHA-1 (Secure Hash Algorithm 1)

SHA-1 是由 NIST（美国国家标准技术研究院）设计的哈希算法，属于 SHA（Secure Hash Algorithm）系列中的一种。它将输入数据映射为一个 160 位（20 字节） 的哈希值。尽管 SHA-1 在早期被广泛使用，但随着计算能力的提高，SHA-1 被证明易受到碰撞攻击，因此不再推荐用于安全敏感的应用。

SHA-1 算法原理：

输入：任意长度的数据（比如消息、文件等）。

输出：固定长度的 160 位（20 字节） 哈希值。

步骤：SHA-1 通过多个 循环运算 和 非线性变换，以及分段处理输入数据，最后生成一个 160 位的输出。

应用场景：数字签名、文件完整性校验（如 Git 中的对象哈希）

注意：由于碰撞攻击，SHA-1 已不再被推荐用于高安全性应用，如 SSL/TLS、数字证书等。

（二） SHA-256 (Secure Hash Algorithm 256-bit)

SHA-256 是 SHA-2（Secure Hash Algorithm 2）系列的一部分，是目前应用最广泛的哈希算法之一。它生成 256 位（32 字节） 长度的哈希值，比 SHA-1 更为安全。

SHA-256 算法原理：

输入：任意长度的数据。

输出：固定长度的 256 位（32 字节） 哈希值。

步骤：SHA-256 使用一种更为复杂的 消息扩展、数据压缩 和 位操作，使得输出更为安全，较少受到碰撞攻击的威胁。

应用场景：数字货币（如比特币的交易哈希）、文件完整性验证、数据签名验证

推荐使用： SHA-256 是目前使用较为广泛且安全性较高的哈希算法之一。

（三） SHA-3 (Secure Hash Algorithm 3)

SHA-3 是 NIST 在 2015 年推出的全新哈希算法标准，使用了与 SHA-2 完全不同的内部结构，基于 Keccak 算法。SHA-3 提供了多种输出长度的选择：224 位、256 位、384 位、512 位等。

SHA-3 算法原理：

输入：任意长度的数据。

输出：可以是 224 位、256 位、384 位或 512 位的哈希值。

步骤：SHA-3 使用 海绵结构（Sponge Construction），该结构具有很高的抗碰撞能力和灵活性，因此可以生成不同长度的哈希值。

应用场景：加密货币（如以太坊）、数据完整性验证、数字签名

推荐使用： SHA-3 具有更高的安全性，特别适用于需要高安全保障的场景。

（四） RIPEMD-160 (RACE Integrity Primitives Evaluation Message Digest)

RIPEMD-160 是一个 160 位输出的哈希算法，是 RIPEMD（RACE Integrity Primitives Evaluation Message Digest）系列中的一个成员。它是由比利时的研究团队提出的，广泛应用于一些欧洲地区的加密协议中。

RIPEMD-160 算法原理：

输入：任意长度的数据。

输出：固定长度的 160 位（20 字节） 哈希值。

步骤：RIPEMD-160 通过将数据分为多个块进行迭代计算，并使用多个加密学上的运算和函数，确保生成的哈希值较难发生碰撞。

应用场景：公钥基础设施（PKI）和数字签名、文件完整性验证

使用推荐：尽管 RIPEMD-160 比较安全，但由于相对较少的应用，现阶段并不如 SHA-256 常见。

代码实现：

```java
import org.bouncycastle.jcajce.provider.digest.RIPEMD160;
import org.springframework.stereotype.Service;
import java.security.MessageDigest;

@Service  2 usages
public class SHA {

    public String hashSHA1(String input) throws Exception {  1 usage
        return hash(input,  algorithm: "SHA-1");
    }

    public String hashSHA256(String input) throws Exception {  1 usage
        return hash(input,  algorithm: "SHA-256");
    }

    public String hashSHA3(String input) throws Exception {  1 usage
        return hash(input,  algorithm: "SHA3-256");
    }

    public String hashRIPEMD160(String input) {  1 usage
        RIPEMD160.Digest digest = new RIPEMD160.Digest();
        byte[] result = digest.digest(input.getBytes());
        return bytesToHex(result);
    }

    private String hash(String input, String algorithm) throws Exception {  3 usages
        MessageDigest digest = MessageDigest.getInstance(algorithm);
        byte[] result = digest.digest(input.getBytes());
        return bytesToHex(result);
    }

    private String bytesToHex(byte[] bytes) {  2 usages
        StringBuilder sb = new StringBuilder();
        for (byte b : bytes) sb.append(String.format("%02x", b));
        return sb.toString();
    }
}
```

解析：

hashSHA1, hashSHA256, hashSHA3:

这些方法分别实现了 SHA-1、SHA-256 和 SHA3-256 的哈希计算。它们都调用了 hash() 方法，并传递相应的算法类型（如 SHA-1、SHA-256 等）。

hash() 方法：

该方法通过 MessageDigest.getInstance(algorithm) 获取对应的哈希算法实例。MessageDigest 是 Java 提供的类，支持常见的哈希算法（如 SHA 系列）。

digest(input.getBytes()) 用于计算输入数据的哈希值，返回一个字节数组。

然后调用 bytesToHex() 方法，将字节数组转换为十六进制的字符串，便于展示和使用。

hashRIPEMD160() 方法：

这是使用 RIPEMD-160 算法进行哈希计算。使用 BouncyCastle 提供的 RIPEMD160.Digest 实现哈希计算。与 SHA 系列类似，返回的也是一个字节数组，转换为十六进制字符串返回。

bytesToHex() 方法：

将字节数组转化为十六进制字符串。每个字节通过 String.format("%02x", b) 转换为两位的十六进制字符。

（五） HMAC (Hash-based Message Authentication Code)

HMAC 是一种基于哈希算法的 消息认证码，结合了哈希函数和一个 共享密钥，用于验证消息的完整性和来源。常用于与 SHA-1 或 SHA-256 等哈希算法结合。

HMAC 算法原理：

输入：数据和密钥。

输出：固定长度的消息认证码。

步骤：HMAC 使用两次哈希操作：一次是在密钥与消息的结合下，第二次则是对其结果的哈希计算。通过这种方式，HMAC 增加了安全性，防止了基于哈希的碰撞攻击。

应用场景：身份验证（如 API 签名）、数据完整性校验、加密协议中的认证机制（如 IPsec、TLS）

推荐使用： HMAC 提供了更高的安全性，适用于需要密钥验证的加密通信场景。

```java
import org.springframework.stereotype.Service;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

@Service 2 usages
public class HMAC {

    public String hmacSHA1(String input, String key) throws Exception { 1 usage
        return hmac(input, key, algorithm: "HmacSHA1");
    }

    public String hmacSHA256(String input, String key) throws Exception { 1 usage
        return hmac(input, key, algorithm: "HmacSHA256");
    }

    private String hmac(String input, String key, String algorithm) throws Exception { 2 usages
        SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), algorithm);
        Mac mac = Mac.getInstance(algorithm);
        mac.init(secretKey);
        byte[] result = mac.doFinal(input.getBytes());
        return Base64.getEncoder().encodeToString(result);
    }
}
```

代码实现：

解析：

hmacSHA1, hmacSHA256:

这两个方法分别实现了 HMAC-SHA1 和 HMAC-SHA256，它们都调用了 hmac() 方法。

hmac() 方法：

创建 SecretKeySpec 对象，将密钥转换为字节数组，并指定 HMAC 的算法（HmacSHA1 或 HmacSHA256）。

使用 Mac.getInstance(algorithm) 获取 HMAC 算法实例。Mac 类是 Java 提供的一个用于消息认证的类。

调用 mac.doFinal(input.getBytes()) 执行 HMAC 操作，生成哈希值。

最后通过 Base64 编码 将哈希值转换为字符串返回。

（六） PBKDF2 (Password-Based Key Derivation Function 2)

PBKDF2 是一种基于密码的密钥派生函数，用于将密码转化为加密密钥。它通过多次哈希计算来增强密码的安全性，并防止密码被暴力破解。

PBKDF2 算法原理：

输入：密码、盐值、迭代次数。

输出：加密密钥。

步骤：PBKDF2 通过多次迭代使用哈希函数对密码和盐值进行计算，生成一个较长的加密密钥。迭代次数越高，破解难度越大。

应用场景：密码存储（如加密数据库中的用户密码）、密钥派生（如加密协议中的密钥生成）

推荐使用： PBKDF2 在密码学中非常重要，特别是在存储用户密码时非常有用，具有防止彩虹表攻击的能力。

代码实现：

```
import org.springframework.stereotype.Service;

import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.util.Base64;

@Service  2 usages
public class PBKDF2 {

    public String pbkdf2(String input, String salt) throws Exception {  1 usage
        int iterations = 10000;
        int keyLength = 256;

        PBEKeySpec spec = new PBEKeySpec(
                input.toCharArray(),
                salt.getBytes(),
                iterations,
                keyLength
        );
        SecretKeyFactory skf = SecretKeyFactory.getInstance( algorithm: "PBKDF2WithHmacSHA256");
        byte[] hash = skf.generateSecret(spec).getEncoded();
        return Base64.getEncoder().encodeToString(hash);
    }
}
```

解析：

pbkdf2() 方法：

使用 PBKDF2 算法将输入密码（input）与 盐值（salt） 一起转化为加密密钥。

PBEKeySpec 类用于定义密码派生的参数：密码（input）、盐值（salt）、迭代次数和密钥长度（256 位）。

SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256") 获取 PBKDF2 算法实例，使用 HMAC-SHA256 作为哈希函数。

调用 skf.generateSecret(spec).getEncoded() 来生成密钥的哈希值。

最后通过 Base64 编码 返回结果密钥。

3.3 编码算法服务

（一） Base64 编码

Base64 是一种将二进制数据编码为文本数据的编码方法，它常用于在 HTTP 协议中传输二进制数据（如图片、文件等），因为 HTTP 仅支持传输文本格式。Base64 将数据转化为可打印的字符，确保在传输过程中不会出现非打印字符。

Base64 编码原理：

输入：任意二进制数据（如文件、图片、文本等）。

输出：一个由 ASCII 字符（如字母、数字、加号“+”、斜杠“/”等）组成的字符串，长度通常比原始数据要长大约 33%。

步骤：

将原始数据按照 6 位一组 分割。

每个 6 位组对应一个 Base64 字符集中的字符。

如果数据长度不是 6 位的整数倍，使用 填充字符（=）填充，使其成为 6 位的倍数。

Base64 编码的特点：

输出的字符集：Base64 编码使用了 64 个字符（字母大小写、数字、加号 + 和斜杠 /），这些字符在大多数编码系统中都是可打印的字符。

填充：当原始数据不能整除 6 位时，Base64 会使用等号 (=) 作为填充字符，确保输出数据的长度是 4 的倍数。

应用场景：

邮件传输：Base64 广泛应用于电子邮件附件的编码。

文件传输：用于将二进制数据（如图片、文档）编码为文本，以便通过文本协议（如 HTTP、JSON）传输。

数据存储：在数据库中存储文件时，常使用 Base64 编码。

代码实现：

```java
import org.springframework.stereotype.Service;
import java.util.Base64;
import java.nio.charset.StandardCharsets;


@Service  2 usages
public class Base64code {

    // Base64 编码
    public String encode(String input) {  1 usage
        return Base64.getEncoder().encodeToString(input.getBytes(StandardCharsets.UTF_8));
    }

    // Base64 解码
    public String decode(String base64) {  1 usage
        byte[] decodedBytes = Base64.getDecoder().decode(base64);
        return new String(decodedBytes, StandardCharsets.UTF_8);
    }
}
```

解析：

encode(String input):

将输入的字符串转换为字节数组，然后用 Base64.getEncoder().encodeToString() 方法将字节数组编码为 Base64 字符串。

input.getBytes(StandardCharsets.UTF_8) 将字符串转换为 UTF-8 编码的字节数组。

Base64.getEncoder().encodeToString() 是 Java 8 提供的 Base64 编码工具，将字节数组编码为可打印的 Base64 字符串。

decode(String base64):

使用 Base64.getDecoder().decode(base64) 将 Base64 字符串解码为字节数组。

new String(decodedBytes, StandardCharsets.UTF_8) 将解码后的字节数组转换回原始字符串。

（二） UTF-8 编码

UTF-8 是一种变长的字符编码方式，是 Unicode 字符集的实现之一。UTF-8 编码可以对 Unicode 中的字符进行编码，使其在计算机和网络中传输时不会出现乱码。

UTF-8 编码原理：

输入：任意字符（包括 ASCII 字符和其他语言字符，如中文、日文等）。

输出：对每个字符使用 1 到 4 个字节 进行编码。

对于 ASCII 字符，UTF-8 使用 1 字节 编码（兼容 ASCII）。

对于其他字符，UTF-8 使用 2 到 4 字节 编码，根据字符集的不同范围使用不同的字节数。

UTF-8 编码的特点：

兼容 ASCII：UTF-8 对于 ASCII 字符（0-127）使用 1 个字节编码，完全兼容 ASCII 编码。

变长编码：UTF-8 根据字符的 Unicode 编码值来确定需要多少字节进行编码，不会浪费空间。

支持多语言字符：UTF-8 支持全球所有语言的字符，能够表示从 基本拉丁字母 到 汉字、表情符号等。

UTF-8 编码示例：

输入（原始字符）：A（字符 "A"）

输出（UTF-8 编码）：0x41（1 字节）

输入（原始字符）：汉（汉字）

输出（UTF-8 编码）：0xE6 0xB1 0x89（3 字节）

应用场景：

网页与文档：UTF-8 被广泛应用于网页编码，确保能够显示所有语言的字符（如 HTML 页面的编码声明）。

数据传输：通过网络传输文本数据时，通常使用 UTF-8 编码，以支持多语言字符集。

数据库存储：大多数现代数据库（如 MySQL）都使用 UTF-8 编码来存储多语言文本数据。

代码实现：

```java
import org.springframework.stereotype.Service;

import java.nio.charset.StandardCharsets;

@Service 2 usages
public class UTF8code {

    // 转换为 UTF-8 字节十六进制表示（可视化编码效果）
    public String encode(String input) { 1 usage
        byte[] utf8Bytes = input.getBytes(StandardCharsets.UTF_8);
        StringBuilder sb = new StringBuilder();
        for (byte b : utf8Bytes) {
            sb.append(String.format("%02x ", b));
        }
        return sb.toString().trim();
    }

    // 还原 UTF-8 编码字符串（例如："e4 bd a0 e5 a5 bd" → "你好"）
    public String decode(String hexEncoded) { 1 usage
        String[] hexParts = hexEncoded.split( regex: " ");
        byte[] bytes = new byte[hexParts.length];
        for (int i = 0; i < hexParts.length; i++) {
            bytes[i] = (byte) Integer.parseInt(hexParts[i], radix: 16);
        }
        return new String(bytes, StandardCharsets.UTF_8);
    }
}
```

解析：

encode(String input):

将输入的字符串转换为 UTF-8 字节数组，然后将每个字节转换为十六进制进行输出。

通过 input.getBytes(StandardCharsets.UTF_8) 方法获取 UTF-8 编码的字节数组。

使用 String.format("%02x", b) 将字节转为十六进制格式，%02x 保证输出的每个字节都是两位的十六进制数。

最后返回拼接好的字符串，表示为每个字节的十六进制形式。

decode(String hexEncoded):

将十六进制字符串 转换回原始的 UTF-8 字符串。

通过 hexEncoded.split(" ") 将十六进制字符串拆分为每个字节的十六进制表示。

使用 Integer.parseInt(hexParts[i], 16) 将每个十六进制部分转换为字节，并将所有字节重新组合成字节数组。

最后使用 new String(bytes, StandardCharsets.UTF_8) 将字节数组重新转换回 UTF-8 编码的字符串。

3.4 公钥密码算法服务

（一） RSA（Rivest－Shamir－Adleman）

RSA 是最常用的公钥加密算法之一，由 Ron Rivest、Adi Shamir 和 Leonard Adleman 于 1977 年提出。它广泛应用于加密、数字签名、密钥交换等场景。

RSA 算法原理：

密钥生成：

1. 选择两个大素数 p 和 q，计算它们的乘积 n=p×q。

2. 计算 n 的欧拉函数 $\phi(n)=(p-1)(q-1)$。

3. 选择一个整数 e（$1<e<\phi(n)$），使得 e 与$\phi(n)$互质。

4. 计算 d，使得 $d\times e\equiv 1 \pmod{\phi(n)}$。

这里，e 是公钥，d 是私钥，n 是公钥和私钥共同的部分。

加密过程：

使用公钥 (e,n) 对明文进行加密：密文=明文^e (mod n)。

解密过程：

使用私钥 (d,n) 对密文进行解密：明文=密文^d (mod n)。

RSA 算法的特点：

安全性依赖于大数分解问题，当前的计算能力下很难从 n 推算出 p 和 q。加密和解密的速度较慢，通常用于小数据量加密或密钥交换。

应用场景：

1. 数字签名：用私钥签名，公钥验证。

2. 密钥交换：用于安全的密钥交换协议（如 SSL/TLS）。

3. 数据加密：在互联网通信中加密数据。

代码实现：

```java
import org.springframework.stereotype.Service;

import javax.crypto.Cipher;
import java.security.*;
import java.util.Base64;

@Service 2 usages
public class RSA {
    private KeyPair keyPair; 3 usages

    public RSA() throws NoSuchAlgorithmException { no usages
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance( algorithm: "RSA");
        keyGen.initialize( keysize: 1024);
        this.keyPair = keyGen.generateKeyPair();
    }

    public String encrypt(String input) throws Exception { 1 usage
        Cipher cipher = Cipher.getInstance( transformation: "RSA");
        cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
        byte[] encrypted = cipher.doFinal(input.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }

    public String decrypt(String encrypted) throws Exception { 1 usage
        Cipher cipher = Cipher.getInstance( transformation: "RSA");
        cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
        byte[] decrypted = cipher.doFinal(Base64.getDecoder().decode(encrypted));
        return new String(decrypted);
    }
}
```

解析：

密钥对生成：

KeyPairGenerator.getInstance("RSA") 使用 RSA 算法 生成密钥对，密钥长度是 1024 位，并通过 keyGen.generateKeyPair() 获取公私钥对。

加密过程：

encrypt() 使用 公钥 对输入数据进行加密。Cipher.getInstance("RSA") 获取 RSA 加解密的实例，使用公钥进行加密。

加密结果是字节数组，通过 Base64 编码 转换为字符串。

解密过程：

decrypt() 使用 私钥 对加密数据进行解密。

Base64.getDecoder().decode(encrypted) 将 Base64 编码的密文解码为字节数组，最后通过私钥解密。

（二） ECC（Elliptic Curve Cryptography）

ECC 是基于椭圆曲线数学的公钥密码算法，具有与 RSA 相同的安全性，但其密钥长度相对较短，因此更加高效。

ECC 算法原理：

ECC 基于椭圆曲线的离散对数问题：给定一个点 P 和倍数 k，计算 kP（曲线上的另一个点），但反过来从 kP 求出 k 是计算上非常困难的。

密钥生成：选择一个椭圆曲线 E，生成公私钥对。公钥是曲线上的点，私钥是与公钥相关的倍数 kP。

加密：使用公钥加密消息。

解密：使用私钥解密。

ECC 相比于 RSA 的优点：

在相同的安全性级别下，ECC 使用的密钥长度比 RSA 短得多。例如，ECC 的 256 位密钥相当于 RSA 的 3072 位密钥。

更高效：ECC 可以提供相同的安全性但需要较少的计算和存储资源，因此在移动设备和嵌入式系统中非常适用。

应用场景：

1. 数字签名：如 ECDSA。
2. 密钥交换：如 ECDH（Elliptic Curve Diffie-Hellman）。
3. 数字证书：使用 ECC 密钥交换和签名算法进行身份验证。

代码生成：

```java
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.springframework.stereotype.Service;

import java.security.*;
import java.security.spec.ECGenParameterSpec;

@Service 2 usages
public class ECC {
    private KeyPair keyPair; 3 usages

    public ECC() throws Exception { no usages
        Security.addProvider(new BouncyCastleProvider());
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance( algorithm: "EC", provider: "BC");

        // 使用标准命名曲线，如 secp256r1
        ECGenParameterSpec ecSpec = new ECGenParameterSpec( stdName: "secp256r1");
        keyGen.initialize(ecSpec);

        this.keyPair = keyGen.generateKeyPair();
    }

    // 获取公钥
    public PublicKey getPublicKey() { 1 usage
        return keyPair.getPublic();
    }

    // 获取私钥
    public PrivateKey getPrivateKey() { 1 usage
        return keyPair.getPrivate();
    }
}
```

代码解析：

依赖 BouncyCastle：

Security.addProvider(new BouncyCastleProvider()); 添加 BouncyCastle 提供的加密算法库。BouncyCastle 提供了对椭圆曲线（ECC）加密的支持。

密钥对生成：

使用 KeyPairGenerator 生成公钥和私钥对。KeyPairGenerator.getInstance("EC", "BC") 表示使用 椭圆曲线算法（EC） 生成密钥对，使用 BouncyCastle 提供的实现。

标准命名曲线 secp256r1：ECGenParameterSpec("secp256r1") 指定使用标准的椭圆曲线 secp256r1，这是目前常用的安全性较高的椭圆曲线之一。

公私钥的获取：

getPublicKey() 返回公钥，getPrivateKey() 返回私钥。

（三） ECDSA（Elliptic Curve Digital Signature Algorithm）

ECDSA 是一种基于椭圆曲线的数字签名算法，是 ECC 的一种应用，用于验证数据的完整性和身份。

ECDSA 算法原理：

密钥生成：

1. 选择一个椭圆曲线 E 和基点 G。

2. 随机选择私钥 d，然后计算公钥 Q=dG。

签名生成：

1. 选择一个随机数 k，计算 r=(kG)x (mod n)。

2. 计算 s=k^(−1)(H(m)+rd) (mod n)，其中 H(m)是消息 m 的哈希值。

签名验证：

使用公钥 Q 和签名 r,s 验证签名的有效性。

ECDSA 与 RSA 的区别：

1. ECDSA 使用的是椭圆曲线而不是大数分解的数学基础，使其在较小密钥长度下仍然具有较高的安全性。

2. ECDSA 生成和验证签名的速度更快，效率更高，适用于资源受限的设备。

应用场景：

1. 数字签名：用于数字证书、SSL/TLS 协议。

2. 加密货币：如比特币和以太坊中使用 ECDSA 签名。

代码生成：

```java
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.springframework.stereotype.Service;

import java.security.*;
import java.util.Base64;
import java.security.spec.ECGenParameterSpec;

@Service  2 usages
public class ECDSA {
    private KeyPair keyPair;  3 usages

    public ECDSA() throws Exception {  no usages
        Security.addProvider(new BouncyCastleProvider());
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance( algorithm: "EC",  provider: "BC");
        // ✅ 使用标准命名曲线，不能直接写 keyGen.initialize(160);
        ECGenParameterSpec ecSpec = new ECGenParameterSpec( stdName: "secp256r1");
        keyGen.initialize(ecSpec);
        this.keyPair = keyGen.generateKeyPair();
    }

    // 生成签名
    public String sign(String input) throws Exception {  1 usage
        Signature signature = Signature.getInstance( algorithm: "SHA256withECDSA",  provider: "BC");
        signature.initSign(keyPair.getPrivate());
        signature.update(input.getBytes());
        return Base64.getEncoder().encodeToString(signature.sign());
    }

    // 验证签名
    public boolean verify(String input, String sig) throws Exception {  1 usage
        Signature signature = Signature.getInstance( algorithm: "SHA256withECDSA",  provider: "BC");
        signature.initVerify(keyPair.getPublic());
        signature.update(input.getBytes());
        return signature.verify(Base64.getDecoder().decode(sig));
    }
}
```

代码解析：

密钥对生成：

使用 BouncyCastle 提供的 EC 算法生成密钥对，和上面的 ECC 类一样，使用 secp256r1 曲线。

签名生成：

sign() 方法使用 私钥 对输入数据进行签名。签名算法是 SHA256withECDSA，即先对数据进行 SHA-256 哈希，然后使用 ECDSA 算法生成签名。

签名结果使用 Base64 编码 转换成字符串。

签名验证：

verify() 方法使用 公钥 对签名进行验证，确保输入数据与签名匹配。

（四） RSA-SHA1（RSA 与 SHA-1 结合）

RSA-SHA1 是 RSA 和 SHA-1 的结合，它用于 数字签名。SHA-1 被用来对消息进行哈希处理，然后使用 RSA 私钥对哈希值进行签名。

RSA-SHA1 算法原理：

对消息进行哈希（SHA-1）。

使用 RSA 私钥 对哈希值进行签名，生成数字签名。

应用场景：

数字签名：常用于消息验证和身份认证。

代码生成：

```java
import org.springframework.stereotype.Service;

import javax.crypto.Cipher;
import java.security.*;
import java.util.Base64;

@Service 2 usages
public class RSASHA1 {
    private KeyPair keyPair; 3 usages

    public RSASHA1() throws NoSuchAlgorithmException { no usages
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance( algorithm: "RSA");
        keyGen.initialize( keysize: 1024);
        this.keyPair = keyGen.generateKeyPair();
    }

    // RSA-SHA1 加密
    public String encrypt(String input) throws Exception { 1 usage
        Cipher cipher = Cipher.getInstance( transformation: "RSA");
        cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
        byte[] encrypted = cipher.doFinal(input.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }

    // RSA-SHA1 解密
    public String decrypt(String encrypted) throws Exception { 1 usage
        Cipher cipher = Cipher.getInstance( transformation: "RSA");
        cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
        byte[] decrypted = cipher.doFinal(Base64.getDecoder().decode(encrypted));
        return new String(decrypted);
    }
}
```

代码解析：

RSA-SHA1 类与 RSA 类基本相同，但主要用于实现 数字签名 和 加密/解密 操作。在实际应用中，RSA-SHA1 是 RSA 与 SHA-1 的结合，通常用于生成签名或加密哈希值。

和 RSA 类的加密解密过程相同，不同之处在于 RSA-SHA1 通常配合 SHA-1 用于签名生成，或者对数据进行哈希加密。

四、 执行结果

在本节中将在前端页面中进行各类加密算法的使用并截图展示。

4.1 对称加密算法

（1） AES 算法

加密

解密


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法： AES
原文 / 输入：
SrdwRXdS9nrMg00YrbsFYQ==
密文 / 输出：
bupt
1234567890abcdef
执行
反向

（2） SM4 算法

加密


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法： SM4
原文 / 输入：
bupt
密文 / 输出：
YX/tSoQutzug+RqV8E8vCw==
1234567890abcdef
执行
反向

解密


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：SM4
原文 / 输入：
YX/tSoQutzug+RqV8E8vCw==
密文 / 输出：
bupt
1234567890abcdef
执行
反向

（3） RC6 算法

加密


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：RC6
原文 / 输入：
bupt
密文 / 输出：
s81QRvBR788/4gSWmGu00A==
1234567890abcdef
执行
反向

解密

4.2 哈希算法

（1） SHA1

（2） SHA256

（3） SHA3

（4） RIPEMD160

（5） HMAC-SHA1

（6） HMAC-SHA256

（7） PBKDF2

4.3 编码算法

（1） BASE64

编码

解码

（2） UTF8

编码


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：
UTF8
原文 / 输入：
bupt
密文 / 输出：
62 75 70 74
执行
反向

解码


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：
UTF8
原文 / 输入：
62 75 70 74
密文 / 输出：
bupt
执行
反向

4.4 公钥密码算法

（1） RSA

（2） RSA-SHA1

（3） ECDSA-SIGN

签名

验证


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：
ECDSA-VERIFY
原文 / 输入：
bupt
密文 / 输出：
true
执行
反向

（4） ECC

获取公钥


密码算法演示平台
王鑫 - 2020211837
对称加密
哈希算法
编码
公钥密码
选择算法：
ECC
原文 / 输入：
ECC 仅用于密钥演示
密文 / 输出：
X:
7f33560ac23a5eab94c780bbec55e289eb13c9290ba2005045f5c261051235fc
Y:
3948d4dd0c58bf271a90d78d4c7e4d59041400d34352d84cd830b3615d799fbe
获取公钥
获取私钥

获取私钥

五、 接口调用

在前端开发过程中，已经设计一个简洁明了的页面，并通过 JavaScript 实现与后端接口的交互。这里的操作包括：

1. 从后端获取返回的加密结果或解密结果。
2. 在页面中展示加密前后的数据。
3. 支持不同类型的算法（对称加密、哈希、编码等）的调用。

5.1 前端页面的功能设计

前端页面已经包括了以下功能：

1. 选择服务类型（对称加密、哈希、编码、公钥加密等）。
2. 选择算法（如 AES、SM4、RC6、SHA256、Base64 等）。
3. 输入框：用户输入需要加密、解密或哈希的原文。
4. 密钥输入框：对需要密钥的算法（如 AES、HMAC 等）提供输入框。
5. 展示密文/哈希值：在加密或哈希之后显示输出结果。

5.2 前端与后端接口的交互

前端需要通过 JavaScript 发送 HTTP 请求（通常是 POST 请求）来调用后端 API。

步骤 1：选择算法

前端界面提供了选择加解密算法的选项，用户可以选择对称加密、哈希、编码等。每个算法在选择时会触发 JavaScript 代码来确定后端需要调用的 API 路径。

步骤 2：填写输入和密钥

根据用户选择的算法，前端动态显示或隐藏密钥输入框。例如，对称加密（AES、SM4、RC6）算法需要密钥，而哈希和编码算法则不需要。

步骤 3：发送请求

前端通过 fetch 发送一个 POST 请求到后端，传递输入数据和密钥（如果需要）。

5.3 JavaScript 代码

1. 页面初始化

```
// 页面初始化
window.onload = function () {
  selectService(currentService);
};
```

页面加载时，默认调用 selectService(currentService)，这将初始化当前选择的服务（默认是 "symmetric"）并加载对应的算法。

2. selectService 方法

```
function selectService(service) {
  currentService = service;
  const algoSelect = document.getElementById("algorithmSelect");

  // 高亮当前按钮
  document.querySelectorAll(".service-tabs button").forEach(btn => {
    btn.classList.remove("active");
    if (btn.textContent.includes(getServiceLabel(service))) {
      btn.classList.add("active");
    }
  });

  // 清空并加载对应算法选项
  algoSelect.innerHTML = "";
  algorithms[service].forEach(algo => {
    const option = document.createElement("option");
    option.value = algo;
    option.text = algo.toUpperCase();
    algoSelect.appendChild(option);
  });

  currentAlgorithm = algorithms[service][0];
  resetFields();
}
```

这个方法负责根据当前选择的服务（如 "symmetric" 或 "hash"）动态加载对应的加密算法。

getServiceLabel(service) 用来获取对应服务的中文名（如 "对称"、"哈希"）。

resetFields() 用于在切换服务时清空输入框和输出框，并根据是否需要密钥（如对称加密算法）显示或隐藏密钥输入框。

3. ResetFields 方法

```
function resetFields() {
  currentAlgorithm = document.getElementById("algorithmSelect").value;
  document.getElementById("inputText").value = "";
  document.getElementById("outputText").value = "";
  document.getElementById("keyInput").style.display =
    currentService === "symmetric" || algoNeedsKey(currentAlgorithm) ? "inline-block" : "none";
  // 显示/隐藏签名输入框（仅对于 ECDSA 验证需要）
  document.getElementById("signatureInput").style.display =
    currentAlgorithm === "ecdsa-verify" ? "inline-block" : "none";

  inputText.value = currentAlgorithm === "ecc" ? "ECC 仅用于密钥演示" : "";  // ECC 显示提示文本
  // 动态创建 ECC 按钮
  const buttons = document.querySelectorAll(".buttons button");
  if (currentAlgorithm === "ecc") {
    buttons[0].textContent = "获取公钥";  // 修改为获取公钥
    buttons[1].textContent = "获取私钥";  // 修改为获取私钥
  } else {
    buttons[0].textContent = "执行";
    buttons[1].textContent = "反向";
  }
}
```

这个方法在每次选择算法后清空输入框和输出框。

如果当前选择的算法需要密钥（如对称加密算法），则显示密钥输入框；否则隐藏密钥输入框。

4. run（mode）方法

```
function run(mode) {
  const input = document.getElementById("inputText").value;
  const key = document.getElementById("keyInput").value;
  const signature = document.getElementById("signatureInput").value;  // 获取签名
  const outputBox = document.getElementById("outputText");

  if (currentAlgorithm !== "ecc" && !input) {
    alert("请输入原文内容");
    return;
  }

  let url = "", body = {}, method = "POST";

  if (currentService === "symmetric") {
    url = `${baseURL}/symmetric/${currentAlgorithm}/${mode}`;
    body = { input, key };
  } else if (currentService === "hash") {
    url = `${baseURL}/hash/${currentAlgorithm}`;
    body = algoNeedsKey(currentAlgorithm) ? { input, key } : { input };
  } else if (currentService === "encoding") {
    url = `${baseURL}/encoding/${currentAlgorithm}/${mode}`;
    body = { input };
  } else if (currentService === "asymmetric") {
    if (currentAlgorithm === "ecdsa-sign") {
      url = `${baseURL}/asymmetric/ecdsa/sign`;
      body = { input };
    } else if (currentAlgorithm === "ecdsa-verify") {
      url = `${baseURL}/asymmetric/ecdsa/verify`;
      body = { input,signature };
    } else if (currentAlgorithm === "ecc") {
            // ECC 操作：获取公钥或私钥
            if (mode === 'encrypt') {
              url = `${baseURL}/asymmetric/ecc/public-key`;  // 获取公钥
              method = 'GET';
            } else if (mode === 'decrypt') {
              url = `${baseURL}/asymmetric/ecc/private-key`;  // 获取私钥
              method = 'GET';
            }
      body = undefined; // GET 请求不需要 body，确保为空
    } else {
      url = `${baseURL}/asymmetric/${currentAlgorithm}/encrypt`;
      body = { input };
    }
  }
```

解析：

这个方法是前端的核心，负责根据当前选择的服务和算法动态构建 API 请求并发送给后端。

url 是根据选择的服务类型（如对称加密、哈希、编码等）动态构建的 API 地址。例如：

对于对称加密：url = ${baseURL}/symmetric/${currentAlgorithm}/${mode}``。

对于哈希算法：url = ${baseURL}/hash/${currentAlgorithm}``。

对于公钥加密算法：url = ${baseURL}/asymmetric/${currentAlgorithm}/encrypt``。

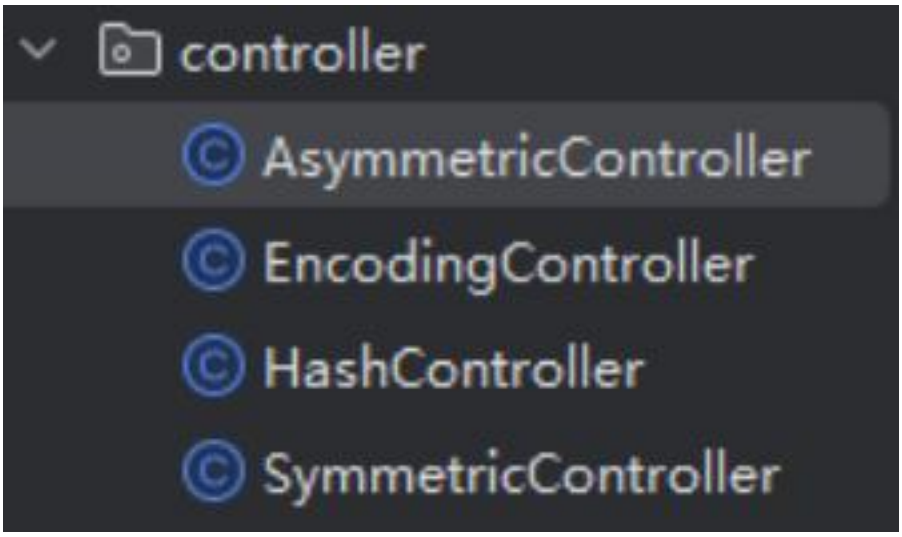
请求体（body）：根据当前选择的算法，发送输入数据和密钥（如果需要）。

fetch()：发送 POST 请求到后端，并获取返回的结果，最终将结果显示在页面的输出框。

5.4 后端接口提供

后端接口响应解析：

在后端代码中，controller 包中提供了与所有加密算法相关的接口，用于处理各个算法的加密、解密、签名、验证等操作。每个接口对应的 HTTP 请求会调用相应的服务方法并返回计算结果。



例如下图中展示的部分接口：

```
// ===== RSA =====
@PostMapping("/rsa/encrypt")  no usages
public String rsaEncrypt(@RequestBody CryptoRequest request) throws Exception {
    return rsaService.encrypt(request.getInput());
}

@PostMapping("/rsa/decrypt")  no usages
public String rsaDecrypt(@RequestBody CryptoRequest request) throws Exception {
    return rsaService.decrypt(request.getInput());
}

// ===== RSA-SHA1 =====
@PostMapping("/rsa-sha1/encrypt")  no usages
public String rsaSha1Encrypt(@RequestBody CryptoRequest request) throws Exception {
    return rsaSha1Service.encrypt(request.getInput());
}

@PostMapping("/rsa-sha1/decrypt")  no usages
public String rsaSha1Decrypt(@RequestBody CryptoRequest request) throws Exception {
    return rsaSha1Service.decrypt(request.getInput());
}

// ===== ECDSA =====
@PostMapping("/ecdsa/sign")  no usages
public String ecdsaSign(@RequestBody SignRequest request) throws Exception {
    return ecdsaService.sign(request.getInput());
}

@PostMapping("/ecdsa/verify")  no usages
public boolean ecdsaVerify(@RequestBody SignRequest request) throws Exception {
    return ecdsaService.verify(request.getInput(), request.getSignature());
}
```

六、 项目总结

本项目实现了一个密码算法演示平台，涵盖了多种常见的密码学算法，包括对称加密、哈希算法、公钥加密及编码算法。项目采用了前后端分离的架构，后端通过 Spring Boot 实现算法服务，前端则使用 HTML 和 JavaScript 提供了用户交互界面。通过实现具体的算法（如 AES、SM4、RC6、RSA、ECDSA、ECC 等），用户可以方便地在平台上进行加密、解密、签名和验证等操作。

项目在前端界面设计上注重简洁与易用性，算法选择和执行结果清晰明了，增强了用户体验。同时，后端接口的设计使得算法模块可以独立工作，增强了系统的扩展性与可维护性。

通过本项目，我深入理解了密码学算法的原理与应用，尤其是对称加密、哈希算法和公钥加密的实现过程。同时，我学会了前后端分离架构的实现与调试，通过优化前端界面和提升用户体验，增强了系统的易用性。整个项目的开发让我提高了系统设计和接口对接的能力，也加深了对 RESTful API、数据交互及算法实现的实践经验。

参考文献:

[1] Tony-老师. [EB/OL] https://blog.csdn.net/u014294681/article/details/86690241.

[2] TheFeasterfromAfar.

[EB/OL] https://blog.csdn.net/qq_40662424/article/details/121791745.

[3] Hollis Chuang.

[EB/OL] https://blog.csdn.net/hollis_chuang/article/details/110729762.

[4] 爱码叔. [EB/OL] https://zhuanlan.zhihu.com/p/436455172.

[5] MIKE笔记. [EB/OL] https://blog.csdn.net/m0_51607907/article/details/123884953.

[6] 佚名.

[EB/OL] https://blog.csdn.net/weixin_43720211/article/details/120200727.

[7] 李月月. [EB/OL] https://zhuanlan.zhihu.com/p/31671646.

[8] Beyond_2016 [EB/OL] https://blog.csdn.net/Beyond_2016/article/details/81286360.

[9] asdzheng. [EB/OL] https://blog.csdn.net/asdzheng/article/details/70226007.

[10] OpenAI. [EB/OL] https://chat.openai.com/.