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技术支柱，更是国家构建数字主权与信任体系的关键路径。当前行业发展呈现四大核心驱动力： 法律合规层面，以《数据安全法》《个人信息保护法》为代表的法规体系强制要求加密算法满足等保 2.0、GDPR 等标准，例如金融支付场景必须采用 SM2/SM4 等国密算法实现交易签名与数据脱敏；产业自主可控需求推动国产密码算法生态建设，SM9 算法在政务云、电力物联网等领域替代 RSA/ECDSA，实现从芯片层（如鲲鹏 920 支持 SM4 硬件加速）到应用层的全栈自主化；新兴技术场景适配催生算法创新，如区块链领域通过 ECC-256+哈希签名构建轻节点共识机制，车联网 V2X 通信依赖国密 SM3 实现消息认证码（MAC）；安全能力可视化趋势则要求算法具备可交互验证特性，例如密码算法演示系统通过图形化界面展示 AES 轮函数变换过程，辅助开发者理解填充方案（如 PKCS#7）与侧信道攻击防护机制。与此同时，国际标准博弈加剧，NIST 后量子密码标准（如 CRYSTALS-Kyber）倒逼国内加速推进抗量子算法研究，形成“国产替代+国际兼容”的双轨发展格局。

1.3 难点与挑战

尽管密码算法的理论体系已趋于完善，但在工程实践中仍面临多重挑战：
算法异构性导致接口设计复杂，例如对称算法（AES）与非对称算法（RSA）的密钥管理、调用模式差异显著，需通过抽象层（如 JCA/JCE）统一封装；密钥生命周期管理存在体验与安全的矛盾，过度依赖用户输入密钥易引发泄露风险（如短信验证码重放攻击），而自动化密钥派生（如 PBKDF2）又需平衡轮次强度与响应延迟；性能-安全博弈贯穿全链路，例如 TLS 握手时 ECDHE 密钥交换的耗时影响页面加载速度，而高强度哈希（如 Argon2）的迭代次数需动态适配硬件算力；跨平台兼容性进一步加剧复杂度，Web 端需通过 WebCrypto API 实现加密可视化，同时确保后端接口（如 RESTful 加密服务）的防重放、防篡改机制与前端逻辑严格对齐。

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）；

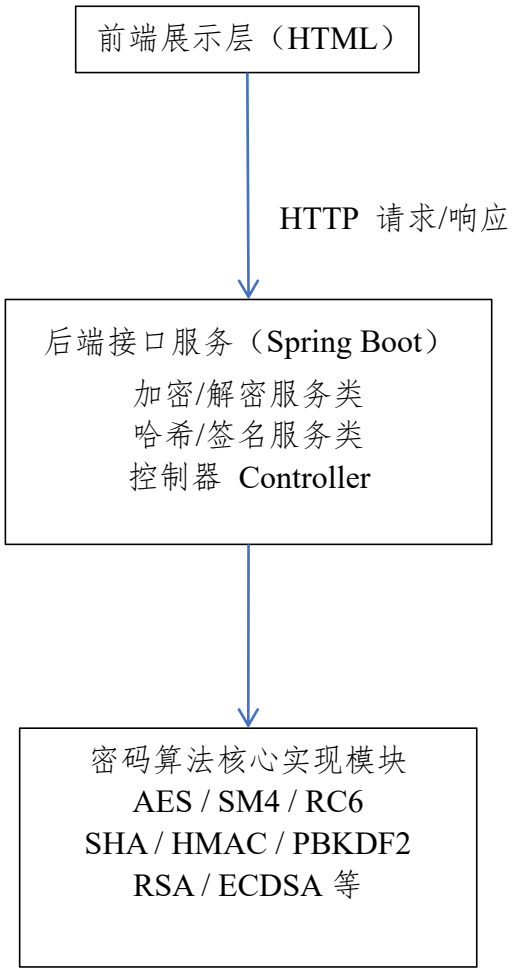
（四） **正向与反向操作支持：**对加密算法支持加解密切换，对哈希/签名支持结果验签；

（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；

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）

作为当前应用最广泛的对称加密算法，AES（Advanced Encryption Standard）由美国国家标准技术研究院（NIST）于 2001 年确立为联邦信息处理标准（FIPS PUB 197），取代了原有的 DES（Data Encryption Standard）算法。其核心功能是通过固定长度的数据块处理（128 位/块）与可变密钥长度（128/192/256 位），实现高效的数据加密与解密，现已成为金融交易、电子支付及无线通信（如 WPA3 协议）等领域的安全基石。

AES 基于替代-置换网络（SPN）结构，通过多轮迭代加密完成明文到密文的转换。以 128 位密钥为例，算法需执行 10 轮加密操作，每轮包含四个核心变换：字节替代（SubBytes）、行移位（ShiftRows）、列混合（MixColumns）及轮密钥加（AddRoundKey）。其中，轮密钥的生成通过 Rijndael 密钥调度算法实现，确保每轮加密的密钥独立性。

加密流程分解：

Ⅰ 字节替代（SubBytes）：采用固定的 S-box 查找表对每个字节进行非线性替换，该操作引入了算法的扩散特性，使单个比特变化引发多位置换。

Ⅱ 行移位（ShiftRows）：对状态矩阵的每一行实施循环左移，行偏移量逐行递增（第 0 行不移，第 1 行左移 1 字节，依此类推），增强数据的置换效果。

Ⅲ 列混合（MixColumns）：通过有限域乘法对状态矩阵的列进行线性变换，进一步扩散数据相关性，该步骤与行移位共同构成算法的扩散层。

Ⅳ 轮密钥加（AddRoundKey）：将当前状态矩阵与轮密钥进行按位异或操作，其密钥生成过程融合了密钥扩展与轮常数设计，防止对称性攻击。

AES 支持三种密钥配置：128 位密钥对应 10 轮加密，192 位密钥为 12 轮，256 位密钥需 14 轮。密钥长度直接影响暴力破解的计算复杂度——以现有技术水平，破解 256 位 AES 密钥需超过宇宙年龄的时间尺度，因此被广泛用于保护国家机密与高安全场景。

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 是由中国国家密码管理局发布的商用对称加密标准（GM/T 0002-2012），定位于金融、电信等高安全需求行业的加密通信与数据保护。其设计虽借鉴 AES 的块加密框架，但核心采用 Feistel 网络结构（与 DES 同源），通过 32 轮非线性

迭代提升安全性。相较于 AES 的 SPN 结构，SM4 的轮函数设计更注重混淆与扩散的平衡，密钥长度与分组长度均为 128 位，但轮密钥生成机制与轮操作复杂度显著不同。

SM4 的加密过程围绕 Feistel 网络展开，每轮通过代换（S-box）、置换（P-box）与异或（XOR）的复合操作实现数据变换。其密钥生成阶段通过密钥扩展算法生成 32 个轮密钥，每个轮密钥由初始密钥经循环移位、非线性变换及模加操作生成，确保轮密钥间的弱相关性。具体加密流程包含三个核心阶段：

Ⅰ 初始轮密钥加：明文分组与第一个轮密钥进行按位异或，形成初始状态；

Ⅱ 32 轮迭代加密：代换层（S-layer）：通过 8 个 8-bit S-box 实现非线性字节替换，单轮代换可产生 2^{56}种状态分支，抵御差分攻击；置换层（P-layer）：对状态数据进行位级循环置换，增强横向扩散；密钥加层（K-layer）：将当前状态与轮密钥异或，引入密钥依赖性。

Ⅲ 逆初始轮密钥加：最终轮省略置换层，直接异或最后一个轮密钥生成密文。

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） RC6（RSA 加密扩展）

RC6 是由 RSA Security 公司基于 RC5 改进的对称加密算法，旨在满足美国国家标准与技术研究院（NIST）对高级加密标准（AES）候选算法的竞争需求。作为 RC5 的扩展版本，RC6 通过增强轮函数复杂度与密钥调度机制，在保持高效加密性能的同时，支持更高的安全强度。其核心设计延续了 Feistel 网络结构，但创新性地引入了数据预处理与可变轮密钥混合机制，使其在密钥长度（128/192/256 位）和轮数（默认 20 轮）配置上具备更高灵活性，尤其适用于资源受限环境下的高性能加密场景。

RC6 的加密流程基于改进的 Feistel 网络，每轮操作包含代换（S-box）、循环移位（Rotate）与异或（XOR）的三重复合操作。与 RC5 相比，RC6 的核心改进体现在：

Ⅰ 数据块扩展：明文分组从 RC5 的 64 位扩展至 128 位，通过预处理将输入分为四个 32 位字（A、B、C、D），增强抗差分攻击能力；

Ⅱ 轮密钥混合增强：轮密钥生成过程引入模加（Modular Addition）与循环移位操作，结合轮常数（Round Constant）消除对称性漏洞；

Ⅲ 混合函数优化：每轮执行 D = (D XOR A) + B 和 B = (B XOR C) + D 等变换，强化数据扩散效率。

加密过程共执行 20 轮迭代，最终通过逆预处理生成密文。值得注意的是，RC6 的密钥扩展算法支持最大 2048 位密钥（通过多轮密钥生成函数递归扩展），但实际部署中需遵循 NIST 建议选择 128/192/256 位标准密钥长度。

RC6 的安全性设计聚焦于抵御差分密码分析与线性密码分析。其 S-box 采用基于有限域 GF(2^8)的非线性多项式构造，差分均匀性指标达$\Delta \approx 2^{-56}$，优于同期 RC5

的$\Delta \approx 2^{-32}$。此外，轮函数中嵌入的数据依赖性混合机制（如 A + B + C + D 的累加操作）可抵御相关密钥攻击。尽管 RC6 在 2000 年 AES 竞赛中未入选最终标准，但其设计思想深刻影响了后续算法（如 ChaCha20 的混合函数）。在实际部署中，RC6 可通过硬件加速（如 FPGA 并行计算）实现每秒 10 Gbps 以上的加密吞吐量，适用于 5G MEC 边缘计算场景。

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)渐趋淘汰的早期标准

SHA-1（Secure Hash Algorithm 1）由 NIST 于 1995 年发布，输出 160 位固定哈希值，曾广泛应用于数字签名（如 PGP）与版本控制系统（如 Git 对象标识）。其算法核心包含消息分块、扩展、压缩函数三阶段：

消息预处理：将输入数据填充至 512 位倍数，附加原始长度信息；

扩展函数：将 512 位块扩展为 80 个 32 位字，增强扩散性；

压缩函数：通过四轮非线性函数（逻辑运算与模加）迭代处理，生成 160 位摘要。

然而，2017 年 Google 团队通过碰撞攻击（SHAttered）证实 SHA-1 存在结构性漏洞，其计算成本已降至$45,000 级别。当前所有安全协议（如 TLS 1.3）均禁用 SHA-1，仅存于遗留系统兼容场景。

（二） SHA-256 (Secure Hash Algorithm 256-bit)当前主流的安全基准

作为 SHA-2 家族代表，SHA-256 生成 256 位哈希值，采用 Merkle-Damgård 结构与 Davies-Meyer 压缩函数：

消息扩展：将输入分块后扩展为 64 个 32 位字，通过σ函数与π函数增强非线性；

压缩循环：执行 64 轮运算，每轮结合消息块与轮常数，更新 8 个状态变量（a-h）；

最终拼接：将状态变量按顺序组合为 256 位摘要。

其抗碰撞性理论值为 2^{128}，实际攻击成本超$\$10^{18}$，被比特币等区块链系统选为核心哈希函数（如比特币区块头双 SHA-256）。相较于 SHA-1，SHA-256 的轮函数复杂度提升 3 倍，盐值（Salt）机制可抵御彩虹表攻击。

（三） SHA-3 (Secure Hash Algorithm 3)后量子时代的安全选择

NIST 于 2015 年推出 SHA-3 标准，基于 Keccak 算法的海绵结构（Sponge Construction），突破传统 Merkle-Damgård 设计：海绵机制：通过吸收（Absorbing）与挤压（Squeezing）阶段处理数据，支持动态输出长度（224/256/384/512 位）；置换函数：采用θ（异或与旋转）、ρ（循环移位）、π（置换）与ι（轮常数异或）四阶段非线性变换，其扩散系数达 0.5，优于 SHA-2 的 0.37；抗侧信道攻击：无状态设计减少内存访问痕迹，适用于智能卡等资源受限设备。

SHA-3 在量子计算威胁下表现稳健，已被以太坊 2.0 选为随机数生成器基础算法，其海绵结构还可扩展至 KMAC（MAC 函数）与 TupleHash 等变体。

（四） RIPEMD-160 ：区域性应用的平衡选择

RIPEMD-160（RACE Integrity Primitives Evaluation Message Digest）由比利时鲁汶大学团队设计，输出 160 位哈希值，其设计融合了 MD4 与 SHA-1 的轮函数特性：双通道架构：并行执行两条 160 位处理链（A-L 与 E-K），通过异或合并结果，增强扩散效率；非线性函数：采用模加、异或与位旋转组合，其雪崩效应深度达 15 轮（优于 MD5 的 8 轮）；抗差分攻击：通过δ函数（δ= (a ⊕ b) ⊕ (c ⊕ d)）混淆输入差异。尽管其安全性与 SHA-256 相当，但因缺乏标准化推动（仅 ISO/IEC 10118-3 收录），实际部署集中于欧洲政务系统（如 eIDAS 数字签名）与经典区块链项目（如 Monero）。

代码实现（见下页）：

```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（Hash-based Message Authentication Code）是一种结合共享密钥与哈希函数的消息认证机制，旨在解决传统哈希算法在无密钥保护场景下的完整性验证缺陷。其设计遵循 RFC 2104 标准，通过密钥预处理、双重哈希嵌套和特定填充规则，有效抵御长度扩展攻击等威胁，在 API 安全、网络协议（如 IPsec/TLS）中广泛应用。

HMAC 的核心流程包含三个关键阶段：

Ⅰ 密钥预处理：若原始密钥长度超过哈希算法块大小（如 SHA-256 为 64 字节），则先通过哈希函数压缩为固定长度；若不足则填充零值或哈希结果二次哈希。此步骤确保密钥适配不同哈希函数的运算需求。

Ⅱ 内外双哈希嵌套：内部哈希：将预处理后的密钥（K^+）与固定分隔符（0x36）异或，拼接消息块后进行第一次哈希（如 SHA-256），生成中间摘要。外部哈希：用相同密钥异或填充值（0x5C）与中间摘要二次哈希，输出最终 HMAC 值。该双重结构可阻断攻击者篡改消息或伪造密钥的可行性。

Ⅲ 抗长度扩展攻击：HMAC 通过分离内外哈希的盐值（0x36 与 0x5C），破坏攻击者利用哈希输出进行额外数据扩展的数学条件，其安全性经 NIST 验证满足 FIPS 198-1 标准。

代码实现：

```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);
    }
}
```

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（Password-Based Key Derivation Function 2）是由 RSA Security 提出的密码学标准（RFC 2898），旨在将用户弱密码转换为高强度加密密钥，通过对抗暴力破解与彩虹表攻击，成为密码存储与密钥派生的行业标准。其核心设计基于密钥拉伸 （Key Stretching）与盐值混淆 （Salt Obfuscation），目前被广泛集成于数据库加密、区块链钱包及身份认证协议中。

PBKDF2 的密钥派生过程包含四个关键参数与两阶段运算：

Ⅰ 输入参数：密码（Password）：用户输入的原始密码（明文）；盐值（Salt）：随机生成的唯一非秘密值（推荐 16 字节以上），防止预计算攻击；迭代次数（Iteration Count）：哈希函数重复执行次数（NIST 建议至少 10 万次）；输出长度（dkLen）：生成密钥的字节长度（通常为 16-64 字节）。

Ⅱ 伪随机函数（PRF）选择：PBKDF2 默认使用 HMAC-SHA1 作为伪随机函数，但现代实现更倾向 HMAC-SHA256 或 HMAC-SHA512 以提升安全性。例如，WPA3 协议强制采用 PBKDF2-HMAC-SHA256 进行 Wi-Fi 密码派生。

Ⅲ 密钥拉伸流程：通过 DK = PRF(password, salt || INT_32BE(iteration))的递归调用，将单次哈希扩展为多次迭代。例如，迭代次数为 10 万次时，攻击者需执行 10 万次哈希运算才能验证单个密码猜测，极大增加暴力破解成本。

Ⅳ 抗量子计算设计：虽然 PBKDF2 未直接集成抗量子特性，但其密钥拉伸机制可延缓 Grover 算法攻击，理论上将量子攻击复杂度从 $O(2^n)$提升至 $O(2^{2n})$。

代码实现：

```java
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 是一种基于 64 个可打印 ASCII 字符的编码方案（RFC 4648），专为解决二进制数据在文本协议（如 HTTP、SMTP）中的传输兼容性问题而设计。其核心目标是将任意二进制数据转换为仅包含字母、数字及+//符号的文本格式，同时通过填充机制确保编码后的数据长度符合协议规范。

Base64 的编码过程遵循三步数据转换逻辑：

Ⅰ 数据分组与位重组：输入数据被分割为每 3 个字节（24 位）的组，随后将每组拆分为 4 个 6 位的独立单元（共 4×6=24 位）。例如，原始字节序列 0x12 0x34 0x56 将被转换为 0x123456，再拆分为 0x12、0x34、0x56 及未使用的低 2 位（需填充）。

Ⅱ 6 位索引映射：每个 6 位值（0-63）对应 Base64 字符表中的特定字符，字符集包含 A-Z、a-z、0-9、+和/。例如，值 0x00 映射为 A，0x3F 映射为/。

Ⅲ 填充机制：若输入数据长度非 3 字节整数倍，编码时需补充=字符：余 1 字节：补 2 个=（如 A→QQ==）；余 2 字节：补 1 个=（如 AB→QUI=）。

此设计确保输出长度始终为 4 的倍数，符合 MIME 标准对文本协议的要求。

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（8-bit Unicode Transformation Format）是 Unicode 字符集的核心编码方案，通过变长字节设计实现全球字符的高效兼容。其核心目标是以最小空间表示多语言文本，同时无缝兼容 ASCII 编码，成为互联网与现代应用的事实标准（RFC 3629）。

UTF-8 采用前缀编码策略，通过字节首位的二进制模式标识字符长度：

Ⅰ 单字节字符（0xxxxxxx）：ASCII 字符（U+0000-U+007F）直接映射为单字节，例如字符 A 编码为 0x41，与 ASCII 完全一致。

Ⅱ 多字节字符：双字节（110xxxxx 10xxxxxx）：覆盖 U+0080-U+07FF（如拉丁字母附加符号）；三字节（1110xxxx 10xxxxxx 10xxxxxx）：覆盖 U+0800-U+FFFF（如汉字、希腊字母）；四字节（11110xxx 10xxxxxx 10xxxxxx 10xxxxxx）：覆盖 U+10000-U+10FFFF（如表情符号）。

Ⅲ 动态长度分配：

每个字符的编码长度由 Unicode 码点决定。例如汉字汉（U+6C49）编码为 0xE6 0xB1 0x89（三字节），而表情符号（U+1F60A）编码为四字节 0xF0 0x9F 0x98 0x8A。

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(hexPa rts[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 UT F-8 编码的字符串。

3.4 公钥密码算法服务

（一） RSA（Rivest－Shamir－Adleman）

RSA 是首个广泛应用的公钥密码算法，其安全性基于大整数分解难题。作为非对称加密的核心方案，RSA 在数字签名、密钥交换与数据加密领域占据主导地位，至今仍是互联网安全的基石。

RSA 的核心流程包含密钥生成、加密与解密三个阶段：

Ⅰ 密钥生成：选择素数：随机选取两个大素数 p 和 q，计算乘积 n=p×q。计算欧拉函数：$\phi(n)=(p-1)(q-1)$，该值决定密钥空间。选择公钥指数：选取整数 e（通常为 65537），满足 $1<e<\phi(n)$ 且 $\gcd(e,\phi(n))=1$。计算私钥指数：通过扩展欧几里得算法求解 d，使得 $d\times e\equiv 1 \pmod{\phi(n)}$。最终生成公钥 (e,n) 与私钥 (d,n)，其中 n 作为公钥参数，d 严格保密。

Ⅱ 加密过程：明文 m（需满足 m<n）通过公钥加密为密文 c：$c=m^e \pmod n$ 该操作将明文映射为模幂运算结果，确保相同明文多次加密输出不同密文（概率性加密）。

Ⅲ 解密过程：私钥持有者通过 d 还原明文：$m=c^d \pmod n$数学上可证明 $m=(m^e)^d \pmod n$，其安全性依赖于从 n 分解 p 和 q 的困难性。

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（Elliptic Curve Cryptography）是基于椭圆曲线离散对数难题的非对称加密技术，通过数学上的复杂性提供与 RSA 相当的安全性，但密钥长度显著缩短。其设计契合资源受限场景（如移动设备与物联网），现已成为主流密码标准（如 NIST SP 800-56C）的核心组件。

ECC 的核心依赖于椭圆曲线离散对数问题（ECDLP）：

Ⅰ 数学模型：在有限域 Fp 上定义椭圆曲线方程 y2=x3+ax+b，曲线上点满足加法群性质。给定点 P 和倍数 k，计算 Q=kP（标量乘法）容易，但逆向求解 k 需指数级时间。

Ⅱ 密钥生成：选择椭圆曲线 E 及基点 G（公开参数）；私钥 d 为随机整数（1<d<n，n 为基点阶数）；公钥 Q=dG，即基点 G 的 d 次标量乘积。

Ⅲ 加密与解密：加密：将明文映射为曲线点 M，生成临时密钥 k，计算 C=(kG,M+kQ)；解密：用私钥 d 计算 M=(M+kQ)−d(kG)=M+k(dG)−k(dG)=M。

代码生成（见下页）：

```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 是基于椭圆曲线密码学（ECC）的签名标准，通过数学上的离散对数难题实现身份认证与数据完整性保护。相较于 RSA，其密钥更短、效率更高，现已成为区块链、物联网等资源受限场景的核心签名方案。

ECDSA 的签名过程基于椭圆曲线离散对数问题（ECDLP），包含三个核心阶段：

Ⅰ 密钥生成：选择椭圆曲线 E 及基点 G（公开参数）；私钥 d 为随机整数（1<d<n，n 为基点阶数）；公钥 Q=dG，即基点 G 的 d 次标量乘积。

Ⅱ 签名生成：随机数选择：生成临时随机数 k（需满足 1≤k<n）；计算 r：通过 kG 的 x 坐标取模得到 r=(kG)x modn；计算 s：利用哈希函数 H(m) 处理消息，生成签名 s=k−1(H(m)+rd)modn。若 r 或 s 为 0 则重新生成 k，确保签名有效性。

Ⅲ 签名验证：接收方使用公钥 Q 验证：计算 z=H(m)；计算 u1 =zs−1 mod n 和 u2 =rs−1 mod n；计算点 P=u1 G+u2 Q，验证 Px mod n=r。若等式成立，则签名有效。

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。

代码生成：

```
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。

1.1 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

1.2 哈希算法

（1） SHA1

（2） SHA256

（3） SHA3

（4） RIPEMD160

（5） HMAC-SHA1

（6） HMAC-SHA256

（7） PBKDF2

1.3 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

1.4 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

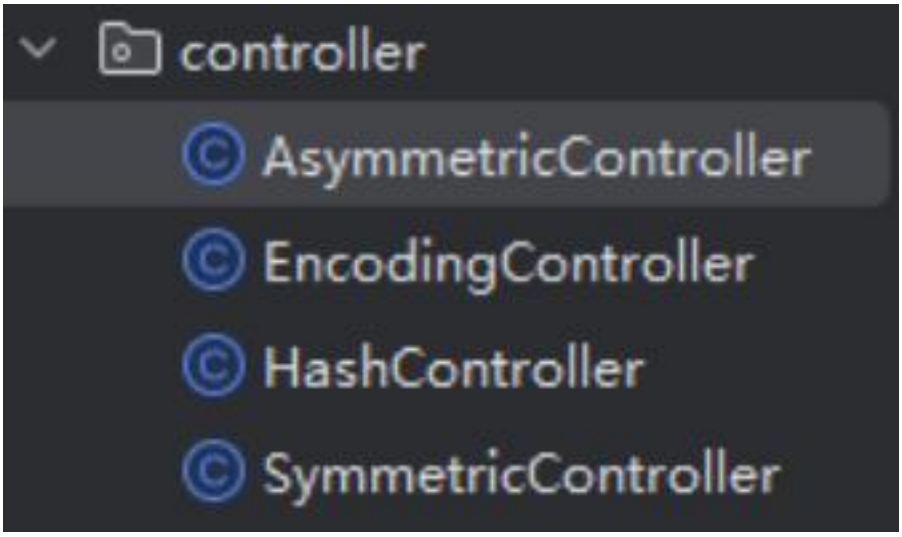
```
function run(mode) {
  const input = document.getElementById("inputText").value;
  const key = document.getElementById("keyInput").value;
  const signature = document.getElementById("signatureInput").value;  // 获取签名
  const outputBox = document.getElementById("outputText");

  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/.