**NETCA软件JCE**

**使用手册**

**广东省电子商务认证有限公司**

**Guangdong Electronic certification Authority**

**（版权所有，翻版必究）**

目 录

[1 简介 1](#_Toc59011543)

[2 注册 1](#_Toc59011544)

[3 支持的算法 2](#_Toc59011545)

[4 用法说明 4](#_Toc59011546)

[4.1 MessageDigest 4](#_Toc59011547)

[4.2 Mac 4](#_Toc59011548)

[4.3 对称加解密 5](#_Toc59011549)

[4.4 X509Certificate 6](#_Toc59011550)

[4.4.1 证书解析 6](#_Toc59011551)

[4.4.2 证书路径构造 6](#_Toc59011552)

[4.4.3 证书路径验证 7](#_Toc59011553)

[4.5 私钥 8](#_Toc59011554)

[4.6 Signature 8](#_Toc59011555)

[4.7 非对称加解密 9](#_Toc59011556)

[4.8 配置tomcat的服务器证书 10](#_Toc59011557)

# 简介

NETCA软件JCE是广东省电子商务认证有限公司的开发的一个纯软件实现JCE的扩展包，它既支持国产的SM2、SM3和SM4算法，也支持国际通用的SHA-1、SHA-2、RSA、ECDSA和AES等算法，并提供证书解析及证书路径验证功能。NETCA软件 JCE是个jar包，名称为netcapki-softwarejce.jar，提供者名为NetcaProvider，对应的类名为net.netca.pki.jce.software.NetcaProvider。使用NETCA软件 JCE要求JAVA 8及其以上版本，并且需要依赖于netcapki-common.jar和netcapki-core.jar。

# 注册

NETCA软件JCE的注册分为静态注册和动态注册。

静态注册是在编辑JRE目录/lib/security/java.security文件，加上一行：

security.provider.n=net.netca.pki.jce.software.NetcaProvider

n为序号，后面行的序号依次递增。

比如，初始安装的时候，java.security的相关内容如下：

security.provider.1=sun.security.provider.Sun

security.provider.2=sun.security.rsa.SunRsaSign

security.provider.3=sun.security.ec.SunEC

security.provider.4=com.sun.net.ssl.internal.ssl.Provider

security.provider.5=com.sun.crypto.provider.SunJCE

security.provider.6=sun.security.jgss.SunProvider

security.provider.7=com.sun.security.sasl.Provider

security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI

security.provider.9=sun.security.smartcardio.SunPCSC

security.provider.10=sun.security.mscapi.SunMSCAPI

如果要使用NETCA软件JCE最优先执行，可以改为：

security.provider.1= net.netca.pki.jce.software.NetcaProvider

security.provider.2=sun.security.provider.Sun

security.provider.3=sun.security.rsa.SunRsaSign

security.provider.4=sun.security.ec.SunEC

security.provider.5=com.sun.net.ssl.internal.ssl.Provider

security.provider.6=com.sun.crypto.provider.SunJCE

security.provider.7=sun.security.jgss.SunProvider

security.provider.8=com.sun.security.sasl.Provider

security.provider.9=org.jcp.xml.dsig.internal.dom.XMLDSigRI

security.provider.10=sun.security.smartcardio.SunPCSC

security.provider.11=sun.security.mscapi.SunMSCAPI

动态注册是先产生一个net.netca.pki.jce.software.NetcaProvider对象，然后使用Security 类的addProvider 或者insertProviderAt进行注册。

# 支持的算法

NetcaProvider支持的算法如下表所示：

|  |  |
| --- | --- |
| Engine | 算法名 |
| MessageDigest | SM3、SHA-1、SHA-224、SHA-256、SHA-384、SHA-512 |
| Mac | HMACSHA1、HMACSHA224、HMACSHA256、HMACSHA384、HMACSHA512、HMACSM3 |
| Signature | SM3withSM2、SHA1withRSA、SHA224withRSA、SHA256withRSA、SHA384withRSA、SHA512withRSA、SM3withRSA、NONEwithRSA、RSASSA-PSS、SHA1withRSAandMGF1、SHA224withRSAandMGF1、SHA256withRSAandMGF1、SHA384withRSAandMGF1、SHA512withRSAandMGF1、SHA1withECDSA、SHA224withECDSA、SHA256withECDSA、SHA384withECDSA、SHA512withECDSA、SM3withECDSA、NONEwithECDSA |
| KeyAgreement | ECDH、ECCDH |
| Cipher | AES/ECB/NOPADDING \*、AES/ECB/PKCS7PADDING \*、AES/CBC/NOPADDING \*、AES/CBC/PKCS7PADDING \*、AES/OFB/NOPADDING \*、AES/GCM/NOPADDING \*、AESWrap\*、SM4/ECB/NOPADDING、SM4/ECB/ PKCS7PADDING、SM4/CBC/NOPADDING、SM4/CBC/ PKCS7PADDING、SM4/OFB/NOPADDING、SM4/GCM/NOPADDING、 SM4Wrap、SM2 |
| KeyPairGenerator | EC、RSA |
| KeyFactory | EC、RSA |
| SecretKeyFactory | PBKDF2WithHmacSHA1、PBKDF2WithHmacSHA224、PBKDF2WithHmacSHA256、PBKDF2WithHmacSHA384、PBKDF2WithHmacSHA512、PBKDF2WithHmacSM3 |
| CertificateFactory | X.509、X509 |
| CertPathBuilder | PKIX |
| CertPathValidator | PKIX |
| KeyStore | PFX、PKCS12 |

表3 NetcaProvider支持的算法表

注：AES算法需要设置net.netca.pki.jce.NetcaProvider.aes.enabled系统属性值为on才能使用。

表中，Hash算法和签名算法都可以使用相应的OID。

MessageDigest类用于进行Hash运算。Mac 类用于进行Mac运算。Signature类用于进行签名和验证签名。KeyAgreement 类用于密钥协商。Cipher类用于加解密，包括对称和非对称算法。KeyPairGenerator类用于产生密钥对。KeyFactory 类用于构造公钥和私钥。SecretKeyFactory 类用于导出对称密钥。CertificateFactory类用于解析公钥证书和CRL，支持SM2证书。CertPathBuilder类用于构造和验证公钥证书。CertPathValidator类用于验证证书。KeyStore类用于获取私钥。

通常使用有两种方式：Engine名.getInstance(算法名)和Engine名.getInstance(算法名, "NetcaProvider")。比如：要进行SM3 Hash，分别是MessageDigest.getInstance("SM3")和MessageDigest.getInstance("SM3", "NetcaProvider")。前者是根据注册的顺序，找到一个该算法的JCE提供者（不一定是NetcaProvider），来进行运算。后者是直接使用NetcaProvider这个提供者。

# 用法说明

## MessageDigest

NETCA软件JCE支持SHA-1、SHA-256、SHA-512、SM3算法。

进行Hash算法，通常使用以下步骤：

1. MessageDigest.getInstance方法获取MessageDigest对象
2. 使用MessageDigest对象的update对输入的数据进行Hash，可以多次调用直到输入所有的数据
3. 使用MessageDigest对象的digest方法得到Hash值

例：

对"abc"的UTF-8编码进行SM3

MessageDigest mdObj= MessageDigest.getInstance("SM3","NetcaProvider");

mdObj.update("abc".getBytes("UTF-8"));

mdObj.digest();//返回值即为结果

## Mac

NETCA软件JCE支持HMACSHA1、HMACSHA256、HMACSHA512、HMACSM3算法。

进行Mac算法，通常使用以下步骤：

1. Mac.getInstance方法获取Mac对象
2. new SecretKeySpec创建Mac的密钥
3. 使用Mac对象的init方法输入上一步得到的密钥进行初始化
4. 使用Mac对象的update对输入的数据进行Mac运算，可以多次调用直到输入所有的数据
5. 使用Mac对象的doFinal方法得到Mac值

例：

对"abc"的UTF-8编码进行HMACSM3,密钥为{0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8}

Mac macObj= Mac.getInstance("HMACSM3","NetcaProvider");

SecretKeySpec key=new SecretKeySpec(new byte[]{0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8},"HMACSM3");

macObj.init(key);

macObj.update("abc".getBytes("UTF-8"));

macObj.doFinal ();//返回值即为结果

## 对称加解密

NETCA软件JCE支持AES和SM4算法，支持ECB、CBC、OFB和GCM模式。AES算法默认是关闭的，需要设置net.netca.pki.jce.NetcaProvider.aes.enabled系统属性值为on才能使用。

进行对称加解密，通常使用以下步骤：

1. 使用Cipher.getInstance得到Cipher对象
2. 使用Cipher对象的init方法输入对称密钥进行初始化
3. 使用Cipher对象的update方法对输入的数据进行加密或者解密运算得到部分结果，可以多次调用直到输入所有的数据
4. 使用Cipher对象的doFinal方法得到最后部分的结果。

3、4两步拼接起来就是最终的结果。

如果数据可以一次获得也可以不用第3步，直接在doFinal方法中传入所有的数据。

如果是GCM模式且有附加的认证数据，则第2步和第3步之间需要使用updateAAD来设置附加的认证数据，可以多次调用。

例：

对"abc"的UTF-8编码进行SM4 CBC模式加密,密钥为{0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf,0x10}，IV为{0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf}。使用PKCS#7填充

Cipher cipherObj=Cipher.getInstance("SM4/CBC/PKCS7PADDING","NetcaProvider");

SecretKeySpec key=new SecretKeySpec(new byte[]{0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf,0x10},"SM4");

IvParameterSpec iv=new IvParameterSpec(new byte[]{0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf});

cipherObj.init(Cipher.ENCRYPT\_MODE, key, iv);

cipherObj.doFinal ("abc".getBytes("UTF-8")); //返回值即为密文

## X509Certificate

### 证书解析

JAVA自带的SUN提供者不能解析SM2证书，会抛出以下的异常：

java.security.cert.CertificateParsingException: java.io.IOException: Unknown named curve: 1.2.156.10197.1.301

NETCA软件JCE使用自己的证书类解析证书从而解决了这个问题。也可以把NetcaProvider提供者放到SunEC提供者之前来解决。

例：

从SM2证书文件中得到证书对象

CertificateFactory cf=CertificateFactory.getInstance("X.509","NetcaProvider");

FileInputStream fIn=new FileInputStream(certFileName);

X509Certificate certObj=(X509Certificate)cf.generateCertificate(fIn);

fIn.close();

### 证书路径构造

NETCA软件JCE实现了证书路径构造。

证书路径不包括根证书，是一组有序的证书列表，第0个元素是要验证的证书，最后一个是根证书签发的证书，第i+1个是第i个的颁发者证书

例：

构造证书路径，构造过程中不验证作废状态

X509CertSelector selector = new X509CertSelector();

selector.setCertificate(eeCert); // eeCert为要构造证书路径的证书

Set<TrustAnchor> trustAnchors = new HashSet<>();

trustAnchors.add(new TrustAnchor(rootCert, null)); // rootCert为根证书

PKIXBuilderParameters pkixParams = new PKIXBuilderParameters(trustAnchors, selector);

pkixParams.setRevocationEnabled(false);

CertStore intermediateCertStore = CertStore.getInstance("Collection",

new CollectionCertStoreParameters(certList));// certList包含CA证书

pkixParams.addCertStore(intermediateCertStore);

CertPathBuilder builder = CertPathBuilder.getInstance("PKIX","NetcaProvider");

CertPathBuilderResult certPathResult=builder.build(pkixParams); //成功返回CertPathBuilderResult对象，构造失败抛出异常

CertPath certPath=certPathResult.getCertPath();

### 证书路径验证

NETCA软件JCE实现了证书路径验证。

例：

验证证书路径，不验证作废状态

CertPathValidator certPathValidator=CertPathValidator.getInstance("PKIX","NetcaProvider");

Set<TrustAnchor> trustAnchors = new HashSet<>();

trustAnchors.add(new TrustAnchor(rootCert, null)); // rootCert为根证书对象

PKIXParameters params = new PKIXParameters(trustAnchors);

params.setRevocationEnabled(false);

CertificateFactory cf=CertificateFactory.getInstance("X.509","NetcaProvider");

CertPath certPath=cf.generateCertPath(certList);// certList为列表对象，里面每个元素是一个证书，第0个元素是要验证的证书，最后一个是根证书签发的证书，第i+1个是第i个的颁发者证书

PKIXCertPathValidatorResult certPathResult=(PKIXCertPathValidatorResult)certPathValidator.validate(certPath, params); //验证成功返回PKIXCertPathValidatorResult类型的对象，失败返回

## 私钥

NETCA软件JCE的私钥通过KeyStore来获取，当前只支持PFX类型的KeyStore

使用KeyStore获取私钥的步骤如下：

1. KeyStore.getInstance方法获取KeyStore对象
2. 使用KeyStore对象的load方法读入PFX文件
3. 使用KeyStore对象的getKey方法获取对应的私钥

当前实现的别名为证书的DER编码的SHA256后再进行Hex编码（使用大写字母）。

例：

获取文件abc.pfx中别名为"62595320F08F991EC13F16510F0928C2CD12AA7B775F709CE19ABD8ED86252FA"的私钥,口令为12345678

KeyStore keyStoreObj= KeyStore.getInstance ("PFX","NetcaProvider");

using(FileInputStream fIn=new FileInputStream("abc.pfx"))

{

char[] pwd="12345678".toCharArray();

keyStoreObj.load(fIn,pwd);

PrivateKey privateKey= (PrivateKey)keyStoreObj.get("62595320F08F991EC13F16510F0928C2CD12AA7B775F709CE19ABD8ED86252FA",pwd);

}

## Signature

NETCA软件JCE支持SHA256withRSA、SHA512withRSA、SHA256withECDSA 、SHA512withECDSA SM3withSM2签名算法。

SM3WithSM2签名的结果遵循《GB/T 35276-2017 信息安全技术 SM2密码算法使用规范》。

进行签名需要使用私钥，私钥的获取参见4.5。验证签名需要可以使用证书或者公钥。证书的获取参见4.4.1。

进行签名，通常使用以下步骤：

1. Signature.getInstance方法获取Signature对象
2. 使用Signature对象的initSign方法传入私钥进行签名初始化
3. 使用Signature对象的update对输入的数据进行Hash，可以多次调用直到输入所有的数据
4. 使用Signature对象的sign方法进行签名得到签名值

例：

进行SM3WithSM2签名

Signature signObj =Signature.getInstance("SM3WithSM2", "NetcaProvider");

signObj.initSign(privKeyObj); //获取私钥参见4.5

signObj.update(tbs); //byte[] tbs，要签名的原文

signObj.sign(); //返回值就是签名值

进行验证签名，通常使用以下步骤：

1. Signature.getInstance方法获取Signature对象
2. 使用Signature对象的initVerify方法传入证书进行验证签名初始化
3. 使用Signature对象的update对输入的数据进行Hash，可以多次调用直到输入所有的数据
4. 使用Signature对象的verify方法传入签名值进行验证。返回true表示验证成功，false表示失败

例：

验证SM3WithSM2签名

Signature signObj =Signature.getInstance("SM3WithSM2", "NetcaProvider");

signObj.initVerify (certObj); //获取证书参见4.4.1

signObj.update(tbs); //byte[] tbs，要签名的原文

signObj.verify (signValue); //返回值表示是否验证成功

## 非对称加解密

NETCA软件JCE支持PKCS#1 V1.5 RSA、SM2等加密算法。

SM2加密的结果遵循《GB/T 35276-2017 信息安全技术 SM2密码算法使用规范》。

进行加密需要可以使用证书或者公钥。证书的获取参见4.4.1。解密需要使用私钥，私钥的获取参见4.5。

SM2的加密需要使用不限制强度的策略文件（Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files）。

进行非对称加密，通常使用以下步骤：

1. 使用Cipher.getInstance得到Cipher对象
2. 使用Cipher对象的init方法输入公钥或者证书进行初始化
3. 使用Cipher对象的doFinal方法输入明文,返回值就是密文

例：

对"abc"的UTF-8编码进行SM2加密

Cipher cipher= Cipher.getInstance("SM2/ECB/NOPADDING","NetcaProvider");

cipher.init(Cipher.ENCRYPT\_MODE, certObj);//获取证书参见4.4.1

cipher.doFinal("abc".getBytes("UTF-8"));//返回值为加密的结果。

进行非对称解密，通常使用以下步骤：

1. 使用Cipher.getInstance得到Cipher对象
2. 使用Cipher对象的init方法输入私钥进行初始化
3. 使用Cipher对象的doFinal方法输入密文,返回值就是明文

例：

SM2解密

Cipher cipher= Cipher.getInstance("SM2/ECB/NOPADDING","NetcaProvider");

cipher.init(Cipher.DECRYPT\_MODE, privKeyObj); //获取私钥参见4.5

cipher.doFinal("abc".getBytes("UTF-8"));//返回值为加密的结果。

## 配置tomcat的服务器证书

下面以windows下64位的tomcat 9配置PFX文件为例进行说明。下面例子的所有值都仅仅是示例使用，实际配置应该替换为对应的值。

具体步骤如下：

1. 产生密钥对，证书请求，并发送给CA，CA签发出SSL/TLS服务器证书，把服务器证书和私钥合并成PFX文件。假设为RSA类型的证书
2. 去https://tomcat.apache.org/download-90.cgi下载[64-bit Windows zip](http://mirrors.hust.edu.cn/apache/tomcat/tomcat-9/v9.0.7/bin/apache-tomcat-9.0.7-windows-x64.zip)。并解压到C:\ apache-tomcat-9.x.y
3. 修改JAVA的java.security。假设使用JAVA 8，其配置文件为C:\Program Files\Java\jre1.8.0\_xxx\lib\security\java.security。插入security.provider.3= net.netca.pki.jce.software.NetcaProvider，把其他的提供者往后移，并增加其序号。参见第2节注册
4. 把netcapki-common.jar、netcapki-core.jar和netcapki-softwarejce.jar拷贝到C:\Program Files\Java\jre1.8.0\_xxx\lib\ext目录下
5. 编辑C:\apache-tomcat-9.x.y\conf\server.xml开启SSL，在

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

的下面，增加：

<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"

               maxThreads="150" SSLEnabled="true">

        <SSLHostConfig>

            <Certificate certificateKeystoreFile="pfx具体文件全路径名" certificateKeystorePassword="具体的PFX保护口令" certificateKeystoreType="PFX"

 certificateKeystoreProvider="NetcaProvider" type="RSA" />

        </SSLHostConfig>

    </Connector>

其中，certificateKeystoreFile的值应为真正的PFX文件名，certificateKeystorePassword应该为真正的PFX文件对应的保护口令。

如果需要限定算法套件，还应该在Connector元素中使用ciphers属性指定只支持的算法套件，每个算法套件名称用空格分开。

1. 把Web网站放进tomcat中，并进行相关配置
2. 打开命令行窗口cmd.exe，下面的8、9、10步都在该命令行窗口执行
3. 设置环境变量JRE\_HOME指向java的jre目录。

SET JRE\_HOME=C:\Program Files\Java\jre1.8.0\_xxx

1. 进入tomcat启动目录

cd C:\apache-tomcat-9.x.y\bin

1. 启动tomcat

.\catalina.bat start

如果没有抛出异常，最终显示

Server startup in xxxx ms

则表示tomcat启动成功