**NETCA PKI Python库使用手册**

# 1 简介

NETCA PKI Python库是广东省电子商务认证有限公司对自己中间件的C语言库的常用功能做的一个python3语言的高层封装。这个库主要包括基本的加密运算、证书及其管理、数字信封、SignedData和时间戳等功能。基本的加密运算主要处理最基本的加密运算，包括产生随机数、Hash、非对称加解密、基本的签名和验证等。证书及其管理主要实现证书的解析、证书链构造和证书验证等功能。数字信封和SignedData是相对高层的操作，主要分别实现EnvelopedData和SignedData。时间戳主要是从时间戳服务器获取时间戳和验证时间戳。

使用NETCA PKI Python库除了安装NETCA的中间件之外，只需要netcacrypto.py这个python文件。

# 2 基本的加密运算

基本的加密运算是处理最基本的密码运算的，当前主要包括：产生随机数、Hash、非对称加解密、基本的签名和验证等。

## 2.1 产生随机数

产生随机数使用generate\_random函数。

例：

产生16个字节的随机数。

import netcacrypto

netcacrypto.generate\_random(16)

## 2.2 Hash运算

Hash运算是由Hash类来实现。当前支持的Hash运算主要有：MD5、SHA1、SHA256、SHA512、SHA3-256、SHA3-512和SM3等。

如果要进行Hash的数据可以一次性获取到，可以直接使用Hash.hash来计算Hash值。如果不能一次性获取到，可以使用下面的方法：

1、 创建Hash对象

2、 多次调用update方法，输入要进行Hash的数据，直到所有的数据输入完毕

3、 调用final方法，得到Hash值。

4、 调用free方法释放Hash对象

例：

一步计算abc的SHA-256的Hash值

Hash.hash(Hash.SHA256, b"abc")

多步计算abc的SHA-256的Hash值

with Hash(Hash.SHA256) as hash\_obj:

hash\_obj.update(b"abc")

hash\_obj.final()

## 2.3 非对称加解密

使用非对称加密运算，首先需要获取到证书对象，获取证书对象的方法参见[3.1](#_3.1_获取证书句柄)。然后使用证书对象的encrypt方法可以进行非对称加密；decrypt方法可以非对称解密，但是要求证书必须有对应私钥。

注意：非对称加解密通常仅仅应用于小数据（比如1024位的RSA只能加密少于117字节数据），如果数据太大，请使用数字信封。

例：

使用证书加密abc

with Certificate(certEncode) as cert\_obj:

cert\_obj.encrypt(b"abc")

使用证书对应的私钥解密

cert\_obj= Certificate.get\_user\_cert(Certificate.USER\_CERT\_TYPE\_ENCRYPT)

if cert\_obj is None:

#出错处理

with cert\_obj:

cert\_obj.decrypt(cipher)

## 2.4 签名和验证

使用签名和验证，首先首先需要获取到证书，获取证书的方法参见[3.1](#_3.1_获取证书句柄)。然后使用证书对象的sign方法可以产生数字签名，要求证书必须有对应私钥。verify\_signature方法可以验证数字签名。

例：

对abc进行数字签名

cert\_obj= Certificate.get\_user\_cert(Certificate.USER\_CERT\_TYPE\_SIGN)

if cert\_obj is None:

#出错处理

with cert\_obj:

cert\_obj.sign(b"abc")

对abc的签名进行验证

with Certificate(certEncode) as cert\_obj:

cert\_obj.verify\_signature (b"abc" ,signature)

# 3 证书

## 3.1 获取证书对象

要对证书进行处理，首先要获取证书对象，证书对象使用完毕后是需要使用free方法来释放。获得获取证书对象的主要方法如下：

1、 根据证书的编码，使用Certificate类的构造函数解析得到，这时得到的证书是不带私钥的。如果需要获取私钥，可以使用get\_privatekey方法。

例：

根据证书编码获取证书对象及私钥

cert\_obj= Certificate(certEncode)

cert\_obj.get\_privatekey(pwd);

2、 客户端使用Certificate.get\_user\_cert方法来选择证书，这些证书通常是有私钥的。第一个参数为Certificate.USER\_CERT\_TYPE\_ENCRYPT表示选择解密证书，为Certificate.USER\_CERT\_TYPE\_SIGN表示选择签名证书。第二个参数是可选的，表示过滤证书的条件。如果不存在符合条件直接返回None，如果仅仅存在一张符合条件的证书，则直接返回，存在多张证书的话，弹出对话框，让用户选择。

例：

获取解密用的用户证书

cert\_obj= Certificate.get\_user\_cert(Certificate.USER\_CERT\_TYPE\_ENCRYPT)

获取签名用的用户证书

cert\_obj= Certificate.get\_user\_cert(Certificate.USER\_CERT\_TYPE\_SIGN)

获取序列号为abcd的签名证书

cert\_obj=Certificate.get\_user\_cert(Certificate.USER\_CERT\_TYPE\_SIGN,"SN='0xABCD'")

3、 使用Certificate.select方法来选择证书。第一个参数表示怎么获取证书，第二个参数是可选的，表示过滤证书的条件。如果不存在符合条件直接返回None，如果仅仅存在一张符合条件的证书，则直接返回，存在多张证书的话，弹出对话框，让用户选择。

例：

选择设备中的证书

cert\_obj=Certificate.select("device")

## 3.2 证书信息的获取

证书对象有很多方法来获取证书的相关信息。可以使用get\_der\_encode方法来获得证书的DER编码。使用get\_pem\_encode方法来获得证书的带头和尾的Base64编码。使用get\_c方法获取证书主体的C。使用get\_cn方法获取证书主体的CN。使用get\_st方法获取证书主体的ST。使用get\_l方法获取证书主体的L。使用get\_o方法获取证书主体的O。使用get\_ou方法获取证书主体的OU。使用get\_subject方法获取证书主体。使用get\_issuer\_c方法获取证书颁发者的C。使用get\_issuer\_cn方法获取证书颁发者的CN。使用get\_issuer\_st方法获取证书颁发者的ST。使用get\_issuer\_l方法获取证书颁发者的L。使用get\_issuer\_o方法获取证书颁发者的O。使用get\_issuer\_ou方法获取证书颁发者的OU。使用get\_issuer方法获取证书颁发者。使用get\_serialnumber方法获取证书的序列好。使用get\_dec\_serialnumber方法获取证书的10进制表示的序列号。使用get\_notbefore方法获取证书有效期的开始时间。使用get\_notafter方法获取证书有效期的结束时间。使用get\_string\_extension\_value方法获取证书的字符串类型的扩展的值。使用computeThumbprint方法来获取证书的微缩图。

例：

获取证书的主体名

cert\_obj.get\_subject()

获取证书的序列号

cert\_obj.get\_serialnumber()

获取证书的微缩图，使用SHA-256算法

cert\_obj.computeThumbprint(Hash.SHA256)

获取UTF8Strting编码的1.2.3.4.5.6.7.8扩展的值

cert\_obj.get\_string\_extension\_value("1.2.3.4.5.6.7.8")

## 3.3 证书匹配表达式

证书对象的match方法使用了一种简单的表达式来判断证书是否符合某种条件。Certificate.get\_user\_cert和Certificate.select的第二个参数也使用了同样的表达式。这种匹配表达式由基本项、与项、非项和或项组成。基本项有两种形式：一种是完全匹配，格式为：名称='值'；一种是包含，格式为名称~'值'，只要值被包含都算匹配。名称和值都是区分大小写的，如果值包含'或者\，需要在前面添加\来转义。与项是项和项之间用&&来连接，非项是在项的前面添加！，或项是项和项之间用||来连接。项还可以用小括号括起来。

常见的表达式项入下表所示：

名称 值的说明

C 主体的C项

ST 主体的ST项

L 主体的L项

O 主体的O项

OU 主体的OU项

CN 主体的CN项

Email 主体或主体备用名的Email项

IssuerC 颁发者的C项

IssuerST 颁发者的ST项

IssuerL 颁发者的L项

IssuerO 颁发者的O项

IssuerOU 颁发者的OU项

IssuerCN 颁发者的CN项

IssuerEmail 颁发者或颁发者备用名的Email项

CertType Encrypt表示加密证书，Signature表示签名证书, KeyAgreement表示密钥协商证书

HasPrivKey True表示有私钥，False表示没私钥

InValidity True表示在有效期内，False表示不在有效期内

SN 证书的序列号，如果是以0x开头则是Hex编码的，否则是10进制表示

Algorithm 公钥算法。RSA为RSA算法，SM2为SM2算法

KeyBits 公钥的位数。

CheckPrivKey 不输入密码连接设备判断私钥是否存在

CSPName CSP的名称

Thumbprint 证书微缩图，格式为：Hash算法名:Hex格式的微缩图值

SubjectKeyId 主体密钥标识符

HasExt 扩展的OID

ExtValue#扩展OID

比如：ExtValue#1.2.3.4 扩展的字符串值UTF-8编码。

HexExtValue#扩展OID

比如：HexExtValue#1.2.3.4 扩展值的Hex编码

CriticalExt#扩展OID

比如：CriticalExt #1.2.3.4 True表示该扩展为关键扩展。False为改扩展为非关键扩展

CertClass 证书的类别，10进制的整数值，参见7.2.4

例：

判断证书的颁发者CN是否包含NETCA

cert\_obj.match("IssuerCN~'NETCA'")

判断证书是否签名证书

cert\_obj.match("CertType='Signature'")

判断证书是否是在有效期内的签名证书

cert\_obj.match("InValidity ='True' && CertType='Signature'")

判断证书是否真的存在私钥

cert\_obj.match("CheckPrivKey ='True'")

判断证书使用SHA-256算法的微缩图是否为42F1C81E38A1819209BCA645CBFF1915E7634C19A3563C447607EB45D87849E3

cert\_obj.match ("Thumbprint ='SHA-256: 42F1C81E38A1819209BCA645CBFF1915E7634C19A3563C447607EB45D87849E3'")

判断证书是否SM2证书

cert\_obj.match("Algorithm ='SM2'")

判断证书是否有扩展1.2.3.4.5.6

cert\_obj.match ("HasExt ='1.2.3.4.5.6'")

判断证书的字符串编码的1.2.3.4.5.6扩展的值是否为abcd

cert\_obj.match ("ExtValue#1.2.3.4.5.6 ='abcd'")

## 3.4 构造证书链

Certificate对象的build\_certpath方法可以构造证书链。得到的证书链是证书对象元组，其中，第0个证书句柄是根证书对象，最后一个证书句柄是要构造证书链的那张证书对象，中间的第i个证书对象是第i+1个证书的颁发者的证书对象。

例：

构造证书链

cert\_obj.build\_certpath()

## 3.5 证书验证

Certificate对象的verify方法可以验证证书。参数flag用来控制验证的级别。如果Certificate.VERIFY\_FLAG\_VERIFY\_REVOKE 没有置位，则只验证证书链，不验证证书的作废信息。如果Certificate.VERIFY\_FLAG\_VERIFY\_REVOKE和Certificate.VERIFY\_FLAG\_VERIFY\_CRL都置位，则使用CRL来验证要验证证书的作废状态。如果Certificate.VERIFY\_FLAG\_ONLINE同时也置位的话，则可能会联网下载CRL。如果Certificate.VERIFY\_FLAG\_VERIFY\_CACERT\_REVOKE同时也置位则会验证CA证书的作废状态。如果需要使用OCSP验证，则需要Certificate.VERIFY\_FLAG\_VERIFY\_REVOKE、Certificate.VERIFY\_FLAG\_VERIFY\_OCSP和Certificate. VERIFY\_FLAG\_ONLINE同时置位。verify方法的返回值是一个字典对象。其中，status的值表示证书的状态，有4种情况：Certificate.STATUS\_UNREVOKED表示未作废，Certificate.STATUS\_UNDETERMINED表示证书状态不确定，Certificate.STATUS\_REVOKED表示证书已经作废，Certificate.STATUS\_CA\_REVOKED表示证书链上的某个CA证书已经作废。如果是Certificate.STATUS\_REVOKED或者Certificate.STATUS\_CA\_REVOKED，revoked\_time的值表示作废时间，如果存在作废原因，则保存在revoked\_reason的值中。

例：

验证证书，但是不验证证书的作废状态

cert\_obj.verify(0)

验证证书，并且在线下载CRL来验证证书链上所有证书的作废状态

cert\_obj.verify(Certificate.VERIFY\_FLAG\_VERIFY\_REVOKE|Certificate.VERIFY\_FLAG\_VERIFY\_CACERT\_REVOKE|Certificate.VERIFY\_FLAG\_ONLINE|Certificate.VERIFY\_FLAG\_VERIFY\_CRL)

# 4 数字信封

这里的数字信封指的是EnvelopedData，是加密消息语法（CMS）的一种，其主要是先使用对称密钥加密数据，然后使用接收者的公钥来加密对称密钥，并把这些信息组合在一起。EnvelopedData主要有版本号、接收者信息和加密信息组成。接收者信息可以是一个或者多个。每个接收者信息主要由版本号、接收者标识、非对称加密算法和加密过的对称加密密钥组成。接收者标识可以使颁发者名和证书序列号的组合或者主体密钥标识符。加密信息主要是对称加密密钥加密数据得到的密文。通常情况下，EnvelopedData并不单独编码，而是封装在ContentInfo中。

NETCA PKI Python库使用EnvelopedData类来处理数字信封。有两种方式处理数字信封，一种是使用配置的方式，这种方式会从配置中获取一些常用的参数进行处理。这种方式的使用相对比较简单。另外一种是不使用配置的方式，这种方式对每个参数都需要自己设置。如果是使用EnvelopedData.new(enc)来创建的EnvelopedData对象，则是使用配置的方式，如果是解密，解密成功后，还需要调用decrypt\_postcheck方法来验证是否符合配置。如果是EnvelopedData(enc)来创建的EnvelopedData对象，则是不使用配置的方式。

## 4.1 构造数字信封

如果要进行加密的数据可以一次性获取到，可以直接使用EnvelopedData.encrypt\_data方法来构造数字信封，这是使用配置的方式。

如果不能一次性获取到或者需要加密给多个人，对于使用配置的方式来说，使用以下的步骤进行构造数字信封：

1、 调用EnvelopedData.new(True)创建进行加密的EnvelopedData对象。

2、 调用EnvelopedData.is\_default\_use\_subject\_key\_id方法获取是否使用主体密钥标准符

3、调用add\_encrypt\_cert方法，逐个添加各个接收者加密证书。如果第2步返回True，则flag设为EnvelopedData.USEKEYID。否则flag设为0。

4、 调用encrypt\_init方法，开始进行加密。同时得到部分结果。

5、 多次调用encrypt\_update方法进行加密，直到明文输入完毕为止。得到加密的结果。

6、 调用encrypt\_final方法结束加密，得到最后部分的结果。4-6步得到的结果串起来就是整个的BER编码的数字信封。

7、 调用free方法释放数字信封对象。

上面4-6步，也可以用encrypt函数来代替。这时得到的是DER编码的数字信封

对于不使用配置的方式来说，使用以下步骤进行构造数字信封：

1. 调用EnvelopedData.new(True)创建进行加密的EnvelopedData对象。
2. 调用set\_encrypt\_algo方法设置对称加密算法
3. 调用add\_encrypt\_cert方法，逐个添加各个接收者加密证书。
4. 可选地，调用set\_int\_attribute方法或者set\_string\_attribute方法设置其他的属性
5. 调用encrypt\_init方法，开始进行加密。同时得到部分结果。
6. 多次调用encrypt\_update方法进行加密，直到明文输入完毕为止。得到加密的结果。
7. 调用encrypt\_final方法结束加密，得到最后部分的结果。5-7步得到的结果串起来就是整个的BER编码的数字信封。
8. 调用free方法释放数字信封对象。

上面5-7步，也可以用encrypt函数来代替。这时得到的是DER编码的数字信封

例：

一步计算使用数字信封加密abc

EnvelopedData.encrypt\_data(cert\_obj,b"abc")

配置方式，多步计算使用数字信封加密abc

with EnvelopedData.new(True) as env\_obj:

flag=0

if EnvelopedData.is\_default\_use\_subject\_key\_id():

flag=EnvelopedData.USEKEYID

env\_obj.add\_encrypt\_cert(cert\_obj,flag)

env\_data=env\_obj.encrypt\_init()

env\_data=env\_obj.encrypt\_update(b"abc")

env\_data=env\_obj.encrypt\_final()

不使用配置的的方式，数字信封加密abc

with EnvelopedData(True) as env\_obj:

env\_obj.set\_encrypt\_algo(EnvelopedData.AES128\_CBC)

env\_obj.add\_encrypt\_cert(cert\_obj)

env\_obj.encrypt(b"abc")

## 4.2 解开数字信封

如果数字信封的数据可以一次性获得，客户端解开数字信封可以直接使用EnvelopedData.decrypt\_data方法，这是使用配置的方式。对于服务端，如果知道解密的证书的话，需要先使用EnvelopedData.set\_decrypt\_cert设置解密的证书（需要有私钥），再使用EnvelopedData.decrypt\_data方法。

如果数字信封的数据不能一次性获取到或者需要进一步的处理，对于使用配置的方式来说，可以使用下面的方法解开数字信封：

1、 EnvelopedData.new(False)创建进行加密的EnvelopedData对象。

2、 如果知道用来加密的证书，调用add\_decrypt\_cert方法，设置自己的证书（需要有私钥）。

3、 调用decrypt\_init方法进行解密初始化

4、 多次调用decrypt\_update方法进行解密，直到数字信封输入完毕为止。得到解密后的明文。

5、 调用decrypt\_final方法结束解密

6、 调用decrypt\_postcheck方法来检查数字信封是否符合要求

7、 调用get\_decrypt\_cert方法来获取解密的证书

9、 调用free方法释放数字信封对象

上面的3-5步，也可以使用decrypt方法来完成。

对于不使用配置的方式，使用以下步骤解开数字信封：

1. EnvelopedData(False)创建进行解密的EnvelopedData对象。
2. 如果知道用来加密的证书，调用add\_decrypt\_cert方法，设置自己的证书（需要有私钥）。
3. 调用decrypt\_init方法进行解密初始化
4. 多次调用decrypt\_update方法进行解密，直到数字信封输入完毕为止。得到解密后的明文。
5. 调用decrypt\_final方法结束解密
6. 调用get\_decrypt\_cert方法来获取解密的证书
7. 可选地，调用get\_encrypt\_algo方法来获取对称加密算法
8. 调用free方法释放数字信封对象

上面的3-5步，也可以使用decrypt方法来完成

例：

一步解开数字信封

EnvelopedData.decrypt\_data(env\_data)

使用配置的方式，多步解开数字信封

with EnvelopedData.new(False) as env\_obj:

env\_obj.decrypt\_init()

env\_obj.decrypt\_update(env\_data)

env\_obj.decrypt\_final()

env\_obj.decrypt\_postcheck()

不使用配置的方式，多步解开数字信封

with EnvelopedData(False) as env\_obj:

env\_obj.decrypt\_init()

env\_obj.decrypt\_update(env\_data)

env\_obj.decrypt\_final()

# 5 SignedData

SignedData也是加密消息语法的一种，其主要是把签名值和签名使用的相关信息组合在一起方便验证。SignedData主要由版本号、摘要算法、要签名的明文（可选）、签名相关的证书（可选）和签名者信息组成。签名者信息可以是一个或者多个，每个签名者信息包含版本号、签名者标识、摘要算法、签名属性（可选）、签名算法、签名值和未签名属性组成。签名者标识可以使颁发者名和证书序列号的组合或者主体密钥标识符。同样地，SignedData并不单独编码，而是封装在ContentInfo中。

SignedData签名通常分为两种：一种是带明文的签名，一种是不带明文的前面。前者会因为明文的增大而增大，但是方便进行验证，验证通过后可以得到明文。后者比较小，但验证时需要能够构造出明文才行。

NETCA PKI Python库使用SignedData类来处理SignedData。和数字信封类型，同样有两种方式来处理，一种是使用配置的方式，这种方式会从配置中获取一些常用的参数进行处理。这种方式的使用相对比较简单。另外一种是不使用配置的方式，这种方式对每个参数都需要自己设置。对于签名，如果使用了apply\_sign\_config则是配置的方式，否则是不使用配置的方式。对于验证签名，如果是使用SignedData.new\_verify\_object()来创建的SignedData对象，则是使用配置的方式，验证签名成功后，还需要调用verify\_postcheck方法来验证是否符合配置。如果是SignedData (False)来创建的SignedData对象，则是不使用配置的方式。

## 5.1 产生签名

如果能一次获取完所有的明文，构造SignedData签名直接使用SignedData.sign\_data方法即可，这是使用配置方式的。

如果不能，对于配置方式，可以使用以下步骤来完成：

1. SignedData(True)创建进行签名的SignedData对象
2. 使用set\_signcert方法设置签名证书
3. 使用apply\_sign\_config方法设置配置属性
4. 使用set\_detached方法设置是否带原文
5. 使用sign\_init方法开始签名，得到开始部分SignedData的编码
6. 使用sign\_update方法输入原文，直到所有的原文输完为止。得到中间部分SignedData的编码
7. 使用sign\_final方法结束签名。得到最后部分的SignedData的编码。5-7步的结果拼起来就是整个SignedData的编码。
8. 使用free方法释放SignedData对象

5-7步也可以使用sign\_onestep方法替代。

不使用配置的方式，构造SignedData签名，使用以下步骤：

1. SignedData(True)创建进行签名的SignedData对象
2. 使用set\_detached方法设置是否带原文
3. 使用set\_signcert方法设置签名证书
4. 使用set\_sign\_algo方法设置签名算法
5. 可选地，设置其他相关的信息
6. 使用sign\_init方法开始签名，得到开始部分SignedData的编码
7. 使用sign\_update方法输入原文，直到所有的原文输完为止。得到中间部分SignedData的编码
8. 使用sign\_final方法结束签名。得到最后部分的SignedData的编码。5-7步的结果拼起来就是整个SignedData的编码。
9. 使用free方法释放SignedData对象

6-8步也可以使用sign\_onestep方法替代。

例：

一步完成abc进行带原文的签名

SignedData.sign\_data(cert\_obj,False, b"abc")

一步完成abc进行不带原文的签名

SignedData.sign\_data(cert\_obj,True, b"abc")

配置方式的多步完成签名

with SignedData(True) as sign\_obj:

sign\_obj.set\_signcert(cert\_obj)

sign\_obj.apply\_sign\_config()

sign\_obj.set\_detached(detached)

sign\_data=sign\_obj.sign\_init()

sign\_data=sign\_obj.sign\_update(tbs\_data)

sign\_data=sign\_obj.sign\_final()

不使用配置方式的多步完成签名

with SignedData(True) as sign\_obj:

sign\_obj.set\_signcert(cert\_obj)

sign\_obj.set\_sign\_algo(algo)

sign\_obj.set\_detached(detached)

sign\_data=sign\_obj.sign\_init()

sign\_data=sign\_obj.sign\_update(tbs\_data)

sign\_data=sign\_obj.sign\_final()

## 5.2 验证签名

所有的验证签名，默认仅仅验证签名，不验证证书本身，对证书仅仅验证有效期和密钥用法。

如果能一次获取完所有的明文，验证SignedData签名使用SignedData.verify\_data方法即可。

如果不能，对于配置方式，可以使用以下步骤来完成带原文的SignedData的验证：

1. SignedData.new\_verify\_object()创建进行验证签名的SignedData对象
2. 可选地，使用add\_cert方法添加可能的签名证书。如果SignedData编码中带有签名证书不需要这一步
3. 调用verify\_init方法开始验证带原文的SignedData
4. 多次调用verify\_update方法，直到SignedData的编码完全输入为止。返回原文
5. 调用verify\_final方法，结束验证签名
6. 调用verify\_postcheck方法，验证签名是否符合配置的要求
7. 比对原文，看看是否符合要求
8. 调用get\_signcert方法获取签名证书，并通过其他方式验证签名证书
9. 调用free方法释放SignedData对象

其中3-5步，可以使用verify\_onestep方法来替代

配置方式的不带原文的SignedData的验证的步骤如下：

1. SignedData.new\_verify\_object()创建进行验证签名的SignedData对象
2. 可选地，使用add\_cert方法添加可能的签名证书。如果SignedData编码中带有签名证书不需要这一步
3. 调用detached\_verify\_init方法输入SignedData的编码，开始不带原文的签名的验证
4. 多次调用detached\_verify\_update方法，直到原文完全输入为止。
5. 调用detached\_verify\_final方法结束签名的验证
6. 调用verify\_postcheck方法，验证签名是否符合配置的要求
7. 调用get\_signcert方法获取签名证书，并通过其他方式验证签名证书
8. 调用free方法释放SignedData对象

其中3-5步，可以使用detached\_verify方法来替代

不使用配置方式的，带原文的SignedData的验证的步骤如下：

1. SignedData(False)创建进行验证签名的SignedData对象
2. 可选地，使用add\_cert方法添加可能的签名证书。如果SignedData编码中带有签名证书不需要这一步
3. 调用verify\_init方法开始验证带原文的SignedData
4. 多次调用verify\_update方法，直到SignedData的编码完全输入为止。返回原文
5. 调用verify\_final方法，结束验证签名
6. 比对原文，看看是否符合要求
7. 调用get\_signcert方法获取签名证书，并通过其他方式验证签名证书
8. 调用free方法释放SignedData对象

其中3-5步，可以使用verify\_onestep方法来替代

不使用配置方式的，不带原文的SignedData的验证的步骤如下：

1. SignedData(False)创建进行验证签名的SignedData对象
2. 可选地，使用add\_cert方法添加可能的签名证书。如果SignedData编码中带有签名证书不需要这一步
3. 调用detached\_verify\_init方法输入SignedData的编码，开始不带原文的签名的验证
4. 多次调用detached\_verify\_update方法，直到原文完全输入为止。
5. 调用detached\_verify\_final方法结束签名的验证
6. 调用get\_signcert方法获取签名证书，并通过其他方式验证签名证书
7. 调用free方法释放SignedData对象

其中3-5步，可以使用detached\_verify方法来替代

例：

一步验证SignedData的签名

SignedData.verify\_data(tbs\_data,sign\_data)

配置的方式，验证带原文的SignedData签名

with SignedData.new\_verify\_object() as sign\_obj:

sign\_obj.verify\_init()

tbs\_data=sign\_obj.verify\_update(sign\_data)

sign\_obj.verify\_final()

sign\_obj.verify\_postcheck()

配置的方式，验证不带原文的SignedData签名

with SignedData.new\_verify\_object() as sign\_obj:

sign\_obj.detached\_verify\_init(sign\_data)

sign\_obj.detached\_verify\_update(tbs\_data)

sign\_obj.detached\_verify\_final()

sign\_obj.verify\_postcheck()

不使用配置的方式，验证带原文的SignedData签名

with SignedData(False) as sign\_obj:

sign\_obj.verify\_init()

tbs\_data=sign\_obj.verify\_update(sign\_data)

sign\_obj.verify\_final()

不使用配置的方式，验证不带原文的SignedData签名

with SignedData(False) as sign\_obj:

sign\_obj.detached\_verify\_init(sign\_data)

sign\_obj.detached\_verify\_update(tbs\_data)

sign\_obj.detached\_verify\_final()

## 5.3 带时间戳的签名

带时间戳的签名是在签名完成后，把签名值发送给时间戳服务器，得到时间戳Token，把它作为一个不签名的属性，编码进原来的SignedData。时间戳可以由签名者在签名完成后马上进行，也可以以后由其他人添加。

马上添加时间戳的方式和普通签名的类似，只是把最后的sign\_final方法替换为sign\_with\_timestamp\_final方法。把sign\_onestep方法替换为sign\_with\_timestamp\_onestep方法。sign\_data方法替换为sign\_with\_timestamp方法（需要配置）。

以后添加时间戳，首先需要验证SignedData的签名，带原文的签名，验证的之前（也就是在detached\_verify\_init、verify\_init、verify\_onestep、detached\_verify方法之前）需要调用keep\_tbs方法。然后使用attach\_timestamp方法（需要配置）或attach\_signature\_timestamp方法追加时间戳。

验证带时间戳的签名，首先要验证SignedData的签名，然后使用has\_tsa\_time方法判断是否带有时间戳，带有则用verify\_timestamp方法进行验证。

# 6 时间戳

时间戳是把数据的Hash值发送到时间戳服务器，时间戳服务器把这个Hash值和当前的时间值以及其他相关信息签名返回给用户。时间戳主要用于证明数据在某个时间之前已经存在。

发送给时间戳服务器的是数据的Hash值，并不是数据本身。时间戳服务器的返回包括一个状态值和一个时间戳Token。时间戳Token是个SignedData结构，其签名的内容主要包括Hash算法和Hash值、时间戳序列号、时间戳的时间值、时间戳服务器的策略OID等。

NETCA PKI Python库使用TimeStamp类来处理时间戳。和数字信封类型，同样有两种方式来处理，一种是使用配置的方式，这种方式会从配置中获取一些常用的参数进行处理。这种方式的使用相对比较简单，但需要预先配置好时间戳服务器地址。另外一种是不使用配置的方式，这种方式对每个参数都需要自己设置。

## 6.1获取时间戳

配置的方式获取时间戳的步骤如下：

1、使用TimeStamp.get\_timestamp方法传入要进行时间戳的数据（Hash之前的数据）来获取时间戳，并得到时间戳对象

2、使用get\_time方法来获取时间戳的时间

3、使用get\_token方法来是否时间戳Token

4、使用free方法来是否时间戳对象

不使用配置的方式获取时间戳的步骤如下：

1. 使用TimeStamp()创建时间戳对象
2. 使用set\_tsa\_url方法设置时间戳服务器的URL
3. 使用set\_hash\_algo方法设置Hash算法
4. 根据Hash算法计算Hash值，并使用set\_hash\_value方法进行设置
5. 使用get\_resp方法向时间戳服务器发送请求，并获取响应。
6. 如果响应状态为TimeStamp.RESP\_STATUS\_GRANTED或者TimeStamp. RESP\_STATUS\_GRANTEDWITHMODS则表示成功
7. 使用get\_time方法来获取时间戳的时间
8. 使用get\_token方法来是否时间戳Token
9. 使用free方法来是否时间戳对象

例：

配置方式，为abc获取时间戳

with TimeStamp.get\_timestamp(b"abc") as timestamp\_obj:

timestamp\_obj.get\_time()

timestamp\_obj.get\_token()

不使用配置的方式，为abc获取时间戳

with TimeStamp() as timestamp\_obj:

timestamp\_obj.set\_tsa\_url(tsa\_url)

timestamp\_obj.set\_hash\_algo(hash\_algo)

timestamp\_obj.set\_hash\_value(hash\_value)

hash\_value=Hash.hash(hash\_algo, b"abc")

status=timestamp\_obj.get\_resp()

if status!=TimeStamp.RESP\_STATUS\_GRANTED and status!=TimeStamp. RESP\_STATUS\_GRANTEDWITHMODS:

#出错了，进行错误处理

……

timestamp\_obj.get\_time()

timestamp\_obj.get\_token()

## 6.2验证时间戳

使用配置的方式，验证时间戳Token的步骤如下：

1、使用TimeStamp.verify\_timestamp方法传入要进行时间戳的数据（Hash之前的数据）和时间戳Token来验证时间戳，并得到时间戳对象

2、使用get\_time方法来获取时间戳的时间

3、使用get\_tsa\_cert方法获取时间戳证书

4、使用free方法来是否时间戳对象

不使用配置的方式，验证时间戳Token的步骤如下：

1、使用TimeStamp.verify\_timestamp\_token传入时间戳Token来验证时间戳，并得到时间戳对象

2、使用get\_hash\_algo方法和get\_hash\_value方法获取Hash算法和Hash值

3、根据Hash算法，输入数据重新计算Hash值

4、验证计算的Hash值和时间戳Token里的Hash值是否一致

5、使用get\_time方法来获取时间戳的时间

7、使用get\_tsa\_cert方法获取时间戳证书

8、使用free方法来是否时间戳对象

例：

使用配置的方式，验证时间戳

with TimeStamp.verify\_timestamp(data,token) as timestamp\_obj:

timestamp\_obj.get\_time()

不使用配置的方式，验证时间戳

with TimeStamp.verify\_timestamp\_token(token) as timestamp\_obj:

hash\_algo=timestamp\_obj.get\_hash\_algo()

hash\_value=timestamp\_obj.get\_hash\_value()

hash\_value2=Hash.hash(hash\_algo,data)

if hash\_value!=hash\_value2:

#出错处理

……

timestamp\_obj.get\_time()