# CryptoPP 库使用

> 参考书籍：《深入浅出CryptoPP密码学库》
>
> crypto++ 密码学库： https://github.com/weidai11/cryptopp
>
> 电子书下载： https://gitee.com/locomotive_crypto
>
> 官方文档

## string 和 SecByteBlock类型互换

C++

```cpp
// SecByteBlock 转 string
SecByteBlock iv; ... // C++-style cast
std::string token = std::string(reinterpret_cast<const char*>(iv.data()), iv.size());

// string 转 SecByteBlock
std::string str; ... // C++-style cast
SecByteBlock sbb(reinterpret_cast<const byte*>(str.data()), str.size());
```

## 第四章 初始CryptoPP库

### Hex编解码字符串

C++

```cpp
#include <iostream>
#include <filters.h>
#include <hex.h>
using namespace std;
using namespace CryptoPP;

int main() {
    // 第一种编码方式
    HexEncoder hex;
    string str = "I like";
    string hexstr;
    hex.Detach(new StringSink(hexstr));
    hex.Put(reinterpret_cast<byte*>(&str[0]),str.size());  // 注意是会追加写入的
    cout << "str:" << str << endl;
    cout << "hexstr:" << hexstr << endl;

    // 第二种编解码写法  Source -> Filter -> Sink  这是一种Pipeline的方式
    string encode,decode;
    StringSource enc(str, true, new HexEncoder(new StringSink(encode)));
    StringSource dec(hexstr, true, new HexDecoder(new StringSink(decode)));
    cout << "encode:"<<encode << endl;
    cout << "decode:"<<decode << endl;
}
```

# 第五章 随机数生成器

主要可以关注GenerateBlock方法，生成指定字节长度的随机数

C++

```cpp
#include<osrng.h> // 可以使用 AutoSeededRandomPool ，该随机器不用设置种子
#include<rng.h> //包含LC_RNG算法的头文件
#include<iostream> //使用cout、cin
using namespace std; //std是C++的命名空间
using namespace CryptoPP; //CryptoPP是CryptoPP库的命名空间
#define Array_Size 64
int main()
{
    //定义一个LC_RNG随机数发生器对象，并设置其种子
    LC_RNG  rng(time(0));
    cout << "产生一个比特的随机数：" << rng.GenerateBit() << endl;
    cout << "产生一个字节的随机数：" << rng.GenerateByte() << endl;
    byte output[Array_Size + 1] = {0};//定义一个缓冲区

    //产生Array_Size字节长度的随机数
    rng.GenerateBlock(output, Array_Size);  // 这里也可直接传入SecByteBlock
    cout << "产生Array_Size长度的随机数（十六进制）：" << endl;
    for (int i = 0; i < Array_Size; ++i)
    {//将获得的随机数转换成十六进制并输出
        printf("%02X", output[i]);
    }
    cout << endl;
    cout << "产生一个100到1000之间的随机数：" << rng.GenerateWord32(100, 1000) << endl;

    //丢弃掉随机数发生器接下来产生的100个字节数据
    rng.DiscardBytes(100);
    int arry[] = { 1,2,3,4,5,6,7,8,9,10 };
    rng.Shuffle(arry, arry + 10); //打乱数组arry中元素的顺序

    return 0;
}
```

# 第六章 Hash函数

C++

```cpp
#include<sha.h> //使用SHA384
#include<filters.h>
#include<hex.h>
#include<iostream> //使用cout、cin
using namespace std; //std是C++的命名空间
using namespace CryptoPP; //CryptoPP是CryptoPP库的命名空间
int main()
```

```
int main()
{
    try
    {
        SHA256 sha; //定义一个SHA256的类对象
        byte msg[] = "I like cryptography very much";

        // 使用pipeline范式
        SecByteBlock tmp(msg, sizeof(msg)-1);
        string r;
        StringSource s1 (tmp, tmp.size(),true,
                         new HashFilter(sha,    // 使用其他hash函数，更换一下类型即可
                         new HexEncoder(
                         new StringSink(r))));
        cout << r << endl;
        r.clear();  // 清空一下，不然后面会追加

        // 连续两次Hash256，再用Hex编码输出
        SHA256 sha2;
        StringSource s2 ("I like cryptography very much",true,
                         new HashFilter(sha,
                         new HashFilter(sha2,
                         new HexEncoder(
                         new StringSink(r)))) );
        cout << r << endl;

        SecByteBlock digest(sha.DigestSize()); //申请内存空间以存放消息摘要
        //CalculateDigest()相当于Update()+Final()
        // Update用来向sha输入，Final计算hash值，同时重置hash函数内部状态
        sha.CalculateDigest(digest, msg, sizeof(msg) - 1);
        cout << "digest2=";
        for (size_t i = 0; i < sha.DigestSize(); ++i)
        {//以十六进制输出Hash值
            printf("%02X", digest[i]);
        }
        cout << endl;

        //计算msg消息的Hash值
        bool res;
        res = sha.VerifyDigest(digest, //可能抛出异常
                               msg, sizeof(msg) - 1); //去掉字符串最后的'\0'
        cout << "res = " << boolalpha <<res << endl;  // 这里的boolalpha 是为了输出bool值true或者false

    }
    catch (const Exception& e)
    {//出现异常
```

```
53          cout << e.what() << endl; //异常原因
54      }
55
56      return 0;
57  }
```

# 第十一章 公钥密码学数学基础

## 大整数 与 大素数生成

C++

```
1   #include<integer.h>//使用Integer
2   #include<iostream>//使用cout、cin
3   #include<osrng.h>//使用AutoSeededRandomPool
4   #include<nbtheory.h>//使用PrimeAndGenerator、VerifyPrime
5   using namespace std;//std是C++的命名空间
6   using namespace CryptoPP;//CryptoPP是CryptoPP库的命名空间
7   int main()
8   {
9       AutoSeededRandomPool rng; // 定义随机数发生器对象
10      // 定义PrimeAndGenerator对象，利用随机数发生器rng产生素数p和q
11      // 要求产生的p是1024比特的素数，q是512比特的素数
12      PrimeAndGenerator pag(1, rng, 1024, 512);
13      Integer p = pag.Prime(); // 获取素数p的值
14      Integer q = pag.SubPrime(); // 获取素数q的值
15      Integer r = (p - 1) / q / 2; // 计算r的值，因为p=2*r*q+1，delta=1
16      cout << "p(" << p.BitCount() << "): " << p << endl; // 打印p的值及比特数
17      cout << "q(" << q.BitCount() << "): " << q << endl; // 打印q的值及比特数
18      cout << "r(" << r.BitCount() << "): " << r << endl; // 打印r的值及比特数
19      if (VerifyPrime(rng, r, 10)) // 验证r是否为素数
20      {
21          cout << "r是素数" << endl; // 如果r为素数，则输出该信息
22      }
23      else
24      {
25          cout << "r不是素数" << endl; // 如果r不为素数，则输出该信息
26      }
27      return 0;
28  }
```

# 大整数和椭圆曲线的一些API

C++

```cpp
#include "util.h"
#include <string>
#include <iostream>
#include <sm3.h>
#include <oids.h>
#include<osrng.h>//使用AutoSeededRandomPool
#include<eccrypto.h>

using namespace std;

int main() {
    Integer i1 = 13;
    Integer i2 = 11;
    Integer i3 = i1.InverseMod(i2);
    //    i3.Negate();        // 取反
    cout << "121 是平方数吗 IsSquare: " << Integer(121).IsSquare() << endl;
    cout << "121 SquareRoot 开方: " <<Integer(121).SquareRoot() << endl;
    cout << "11 Squared 平方: " <<Integer(11).Squared() << endl;
    cout << "13 % 11 = " << i1.Modulo(i2) << endl;
    cout << "2 关于 11 的乘法逆元: " << Integer(2).InverseMod(i2)<< endl;

    // hash值转Integer
    string hash =
"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F";
    Integer hashValue = Util::StringToInteger(hash);

    // 椭圆曲线的点
    ECP::Point zero;
    // 获取点的x和y坐标
    cout << zero.x << " " << zero.y << endl;

    // 椭圆曲线构造，传入大素数p，a，b
    ECP ecp(hashValue,i1,i2);
    // 椭圆曲线点的计算，乘法传入一个 Integer，  一个 Point
    ECP::Point point = ecp.Multiply(i2,zero);
    point = ecp.Add(point, point);

    Integer n,d;
    // Singer构造，传入椭圆曲线，生成点，生成点对应的阶数n，私钥d
    ECDSA<ECP,SM3>::Signer signer(ecp, point,n,d);
    ECDSA<ECP,SM3>::PrivateKey privateKey;
    // 私钥初始化也是类似
    privateKey.Initialize(ecp,point,n,d);
    // 可以利用已有的曲线
```

```
    // 可以利用已有的曲线
    privateKey.Initialize(ASN1::secp256r1(), d);

    // 自己进行签名，传入k 和 e 值， 返回 r 和 s
    Integer k,e,r,s;
    signer.RawSign(k, e,r,s);
}
```

# 第十三章 数字签名

电子书给出的参考是ECNR数字签名算法

ECDSA椭圆曲线 - Crypto++

https://www.cryptopp.com/wiki/Elliptic_Curve_Digital_Signature_Algorithm -》关于压缩公钥的实现可以参考 Compressed Point 这一部分

## 私钥和公钥生成、保存

C++

```
#include<integer.h>//使用Integer
#include<iostream>//使用cout、cin
#include<osrng.h>//使用AutoSeededRandomPool
#include<eccrypto.h>
#include <oids.h>
#include <files.h>
#include <filters.h>
#include <filesystem>
#include <hex.h>
#include <base32.h>
using namespace std;//std是C++的命名空间
using namespace CryptoPP;//CryptoPP是CryptoPP库的命名空间
void ECDSA_Generate()
{
    /*
    // 指定 private exponent  32字节随机数，生成对应的私钥
    string exp =
"E4A6CFB431471CFCAE491FD566D19C87082CF9FA7722D7FA24B2B3F5669DBEFB";

    HexDecoder decoder;
    decoder.Put((byte*)&exp[0], exp.size());
    decoder.MessageEnd();

    Integer x;
    x.Decode(decoder, decoder.MaxRetrievable());
```

```
    privateKey.Initialize(ASN1::secp256r1(), x);
     */

    AutoSeededRandomPool prng;
    ECDSA<ECP, SHA256>::PrivateKey privateKey;

    privateKey.Initialize( prng, ASN1::secp256k1() );

    /* 使用ByteQueue 方便将公私钥存储在内存中，如果要持久化到磁盘，可以使用FileSink
    ByteQueue queue;
    privateKey.Save(queue);
    privateKey.Load(queue);
    */

    // 验证密钥强度
    bool result = privateKey.Validate( prng, 3 );
    cout << boolalpha << result << endl;

    Integer p = privateKey.GetPrivateExponent();
    cout<< "len:" << p.BitCount() << " Private Key:" << p << endl;
    string priHexKey;
    HexEncoder encoder(new StringSink(priHexKey));
    p.Encode(encoder, 32);
    cout << "priHexKey:"<<priHexKey.size() << "  " << priHexKey << endl;

    // Save private key in PKCS #8 format
    FileSink fs1( "../../../keys/private.ec.der", true /*binary*/ );
    privateKey.Save( fs1 );

    // Generate publicKey
    ECDSA<ECP, CryptoPP::SHA256>::PublicKey publicKey;
    privateKey.MakePublicKey(publicKey);
    const ECP::Point& q = publicKey.GetPublicElement();
    const Integer& qx = q.x;
    const Integer& qy = q.y;
    string qxHex, qyHex;
    HexEncoder encoderx(new StringSink(qxHex));
    HexEncoder encodery(new StringSink(qyHex));
    qx.Encode(encoderx, 32);
    qy.Encode(encodery, 32);
    cout << "len:" << qx.BitCount() << " Public Point x:" << qx << endl;
    cout << "qxHex:" << qxHex << endl;
    cout << "len:" << qy.BitCount() << " Public Point y:" << qy << endl;
    cout << "qyHex:" << qyHex << endl;

    // Save public key in X.509 format
```

```
    FileSink fs2( "../../../keys/public.ec.der", true /*binary*/ );
    publicKey.Save( fs2 );

}
```

## 私钥和公钥的加载、签名和认证

C++

```
// 头文件和上述一样

void ECDSA_LOAD(){
    AutoSeededRandomPool prng;
    FileSource fs1( "../../../keys/private.ec.der", true /*pump all*/ );
    FileSource fs2( "../../../keys/public.ec.der", true /*pump all*/ );
    ECDSA<ECP, SHA256>::PrivateKey privateKey;
    ECDSA<ECP, SHA256>::PublicKey publicKey;
    // 加载私钥， 私钥格式：PKCS #8
    privateKey.Load( fs1 );
    publicKey.Load(fs2);

    // 用私钥进行签名
    ECDSA<ECP, SHA256>::Signer signer(privateKey);
    string message = "Yoda said, Do or do not. There is no try.";
    string signature;


    StringSource s( message, true /*pump all*/,
                    new SignerFilter( prng,
                                      signer,
                                      new StringSink( signature )
                    ) // SignerFilter
    ); // StringSource
    cout << "signature len:" << signature.size() << " output:" << signature <<
endl;

    // 将签名（包含R，S）转换成 DER 格式
    std::string derSign;
    // Make room for the ASN.1/DER encoding
    derSign.resize(3+3+3+2+signature.size());
    size_t converted_size = DSAConvertSignatureFormat(
            (byte*) (&derSign[0]), derSign.size(), DSA_DER,
            (const byte*) (signature.data()), signature.size(), DSA_P1363);
    derSign.resize(converted_size);
    cout << "DER len:" << derSign.size() << " DER:" << derSign << endl;
    string hexDER;
    StringSource toDER(derSign, true, new HexEncoder(new StringSink(hexDER)));
```

```
    cout << "DER hex:" << hexDER << endl;


    // 进行验证
    bool result;
    ECDSA<ECP, SHA256>::Verifier verifier(publicKey);
    StringSource ss( signature+message, true /*pump all*/,
                    new SignatureVerificationFilter(
                            verifier,
                            new ArraySink( (byte*)&result, sizeof(result) )
                    ) // SignatureVerificationFilter
    );

    // 传统的C 方式 - 函数调用形式
//    result = verifier.VerifyMessage( (const byte*)message.data(), message.size(), (const byte*)signature.data(), signature.size() );
    if(result)
        std::cout << "Verified signature on message" << std::endl;
    else
        std::cerr << "Failed to verify signature on message" << std::endl;

}
```

## 模拟数字签名过程和验证签名、公钥恢复

C++

```

// 先模拟签名过程
Integer k,e,r,s,a=0,b=7;
k = Util::GetRandomInteger(4);    // 随机生成的4字节整数
e = Util::StringToInteger("9CE3A97A43618E606AD1FCD7926DE493E69E58CC0DD3139183A4E337F8E81A41");        // 消息散列
Integer prime = Util::StringToInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F");

ECP secp256k1(prime, a, b);
Integer xg = Util::StringToInteger("79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798");
Integer yg = Util::StringToInteger("483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8");
ECP::Point G(xg,yg);        // 生成元
Integer n = Util::StringToInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141");
Integer d = Util::StringToInteger("E4A6CFB431471CFCAE491FD566D19C87082CF9FA7722D7FA24B2B3F5669DBEFB");      //私钥
```

```
ECP::Point Q = secp256k1.Multiply(d,G); //公钥

cout << "——————————椭圆曲线参数——————————"<< endl;
cout << "p:" << prime << endl;
cout << "a:" << a << endl;
cout << "b:" << b << endl;
cout << "生成点G:" << endl;
cout << "G.x:" << xg << endl;
cout << "G.y:" << yg <<endl;
cout << "n:" << n << endl;
cout << "私钥d:" << d << endl;
cout << "公钥Q:" << endl;
cout << "Q.x:" << IntToString(Q.x) << endl;
cout << "Q.y:" << IntToString(Q.y) << endl;


// 签名算法
ECP::Point R = secp256k1.Multiply(k, G);
r = R.x.Modulo(n);
Integer k_1 = k.InverseMod(n);
s = (k_1 * (d * r + e)).Modulo(n);
cout << "——————————签名过程——————————" << endl;
cout << "r:" << r << endl;
cout << "s:" << s << endl;

// 验证签名
Integer s_1 = s.InverseMod(n);
Integer u_1 = (e * s_1).Modulo(n);
Integer u_2 = (r * s_1).Modulo(n);
ECP::Point tmp1 = secp256k1.Multiply(u_1,G);
ECP::Point tmp2 = secp256k1.Multiply(u_2,Q);
ECP::Point X = secp256k1.Add(tmp1, tmp2);
cout << "——————————验证过程——————————" << endl;
cout << "签名中的r:" <<r % n << endl;
cout << "计算得到的x:"<<X.x % n<< endl;


// 公钥恢复
bool isOdd = R.y.IsOdd();
Integer r_1 = r.InverseMod(n);
Integer y_2 = (r * r * r + 7) ;

// 模素数平方根的求解: 当  p = 4u+3, g = y^2, y= g^(u+1) % p
Integer u = (prime-3).DividedBy(4);
Integer y = a_exp_b_mod_c(y_2,u+1,prime);      // 返回  a^b % c

if ((isOdd && y.IsEven()) || (!isOdd && y.IsOdd())) {       // 如果与原来的y坐标奇偶性相反
```

```
61      y = prime - y;
62  }
63
64  ECP::Point tmpR(r,y);
65  ECP::Point tmp3 = secp256k1.Multiply(s,tmpR);
66  ECP::Point tmp4 = secp256k1.Multiply(e,G);
67  tmp3 = secp256k1.Subtract(tmp3,tmp4);
68  tmp4 = secp256k1.Multiply(r_1,tmp3);
69
70  cout << "----------公钥恢复----------" << endl;
71  cout << "计算的x:" << tmp4.x << endl;
72  cout << "公钥Q.x:" <<Q.x << endl;
73  cout << "计算的y:"<< tmp4.y << endl;
74  cout << "公钥Q.y:"<<Q.y << endl;
```