Openssl 编程

江南计算技术研究所

赵春平 著

Email: forxy@126.com

版本： 1.0

前言

最早接触 openssl 是做硕士论文的时候，研究网格安全。实际工作是替换 globus 中 openssl 实现的 gssapi(对称算法和非对称算法)的底层算法。初次接触 openssl，都不知道怎么下手，我用了最笨的方法来替换对称算法：将 RC4 关键字替换掉，换成自己的对称算法。虽然最后也做完了，客户端支持读卡器之类的硬件，服务端支持 pkcs11 接口的 PCI 密码卡。但是做完之后也不知道有没用过(只有我自己测试过，从来没有跟其他系统联调过)，因为 globus 很多还是用 java 来实现的。没见过巨型机长啥样，也没有网格应用环境，我的研究也就是理论研究而已。由于有些原因，第一年的硕士答辩没参加，延迟与一年。主要是论文没准备好，并且春节刚好家里有事情。第二年答辩勉强过。结果，省里后来抽检硕士论文，我的论文不合格。郁闷不已。所以我要写这篇文章，是动机之一了。

后来做 PMI，要做属性证书编解码。我又把 openssl 拿起来了，单独剥离了 asn1 模块。对 openssl 有更深的了解。我又没有记笔记的习惯，知道一点就在电脑上记录一点，甚至还把 window 下 openssl 提供的所有接口都写在一个文件里面。会一个函数就写一个，注释一个。时间长了，也有一定积累了。干脆，写本书得了。不过，openssl 很多方面我到现在还是不清楚。有时候学习一个函数还得专门写程序来学习和调试它。有时候对我而言，学习是快乐的。我大概花了好几个月的时间在家里写代码调试程序，最终完成这本书。

在 openssl.cn 论坛混了个斑竹当当，回答些问题，也是我学习

的动力之一。作为斑竹，我为不能回答的问题而汗颜。

评定职称需要论文的。本来想得很理想：出书，评职称。结果写出来也没太大用处。我甚至想过每一章节去投稿，但是那样太慢了，我也懒得去弄，就放弃了。2001 年本科毕业，到现在还是助理工程师，我是不是太笨了。研究生也读的不咋的，我都不知道我现在算本科生还是研究生。

有时候我在家写书的时候，老婆打电话来，我就说忙着呢，写书呢。我老婆听了挺高兴。等到写完了，开始还问我，写得书怎么样了，我就跟她说，纯属个人爱好。现在，她再也不问我了。所以，还是得感谢老婆大人滴。

第一章　基础知识

1.1　对称算法

对称算法使用一个密钥。给定一个明文和一个密钥，加密产生密文，其长度和明文大致相同。解密时，使用读密钥与加密密钥相同。

对称算法主要有四种加密模式：

(1) 电子密码本模式　Electronic Code Book(ECB)

这种模式是最早采用和最简单的模式，它将加密的数据分成若干组，每组的大小跟加密密钥长度相同，然后每组都用相同的密钥进行加密。

其缺点是：电子编码薄模式用一个密钥加密消息的所有块，如果原消息中重复明文块，则加密消息中的相应密文块也会重复，因此，电子编码薄模式适于加密小消息。

（2）加密块链模式　Cipher Block Chaining(CBC)

CBC 模式的加密首先也是将明文分成固定长度的块，然后将前面一个加密块输出的密文与下一个要加密的明文块进行异或操作，将计算结果再用密钥进行加密得到密文。第一明文块加密的时候，因为前面没有加密的密文，所以需要一个初始化向量。跟 ECB 方式不一样，通过连接关系，使得密文跟明文不再是一一对应的关系，破解起来更困难，而且克服了只要简单调换密文块可能达到目的的攻击。

（3）加密反馈模式　Cipher Feedback Mode(CFB)

面向字符的应用程序的加密要使用流加密法，可以使用加密反馈模式。在此模式下，数据用更小的单元加密，如可以是 8 位，这个长度小于定义的块长（通常是 64 位）。其加密步骤是：

a） 使用 64 位的初始化向量。初始化向量放在移位寄存器中，在第一步加密，产生相应的 64 位初始化密文；

b) 始化向量最左边的 8 位与明文前 8 位进行异或运算，产生密文第一部分（假设为 c），然后将 c 传输到接收方；

c) 向量的位（即初始化向量所在的移位寄存器内容）左移 8 位，使移位寄存器最右边的 8 位为不可预测的数据，在其中填入 c 的内容；

d) 第 1-3 步，直到加密所有的明文单元。

解密过程相反

4）输出反馈模式　Output Feedback Mode(OFB)

输出反馈模式与 CFB 相似，惟一差别是，CFB 中密文填入加密过程下一阶段，而在 OFB 中，初始化向量加密过程的输入填入加密过程下一阶段。

1.2　摘要算法

摘要算法是一种能产生特殊输出格式的算法，这种算法的特点是：无论用户输入什么长度的原始数据，经过计算后输出的密文都是固定长度的，这种算法的原理是根据一定的运算规则对原数据进行某种形式的提取，这种提取就是摘要，被摘要的数据内容与原数据有密切联系，只要原数据稍有改变，输出的“摘要”便完全不同，因此，基于这种原理的算法便能对数据完整性提供较为健全的保障。但是，由于输出的密文是提取原数据经过处理的定长值，

所以它已经不能还原为原数据，即消息摘要算法是不可逆的，理论上无法通过反向运算取得原数据内容，因此它通常只能被用来做数据完整性验证。

如今常用的“消息摘要”算法经历了多年验证发展而保留下来的算法已经不多，这其中包括 MD2、MD4、MD5、SHA、SHA-1/256/383/512 等。

常用的摘要算法主要有 MD5 和 SHA1。D5 的输出结果为 16 字节，sha1 的输出结果为 20 字节。

1.3 公钥算法

在公钥密码系统中，加密和解密使用的是不同的密钥，这两个密钥之间存在着相互依存关系：即用其中任一个密钥加密的信息只能用另一个密钥进行解密。这使得通信双方无需事先交换密钥就可进行保密通信。其中加密密钥和算法是对外公开的，人人都可以通过这个密钥加密文件然后发给收信者，这个加密密钥又称为公钥；而收信者收到加密文件后,它可以使用他的解密密钥解密，这个密钥是由他自己私人掌管的，并不需要分发，因此又成称为私钥，这就解决了密钥分发的问题。

主要的公钥算法有：RSA、DSA、DH 和 ECC。

（1）RSA 算法

当前最著名、应用最广泛的公钥系统 RSA 是在 1978 年，由美国麻省理工学院(MIT)的 Rivest、Shamir 和 Adleman 在题为《获得数字签名和公开钥密码系统的方法》的论文中提出的。它是一个基于数论的非对称(公开钥)密码体制，是一种分组密码体制。其名称来自于三个发明者的姓名首字母。 它的安全性是基于大整数素因子分解的困难性，而大整数因子分解问题是数学上的著名难题，至今没有有效的方法予以解决，因此可以确保 RSA 算法的安全性。RSA 系统是公钥系统的最具有典型意义的方法，大多数使用公钥密码进行加密和数字签名的产品和标准使用的都是 RSA 算法。

RSA 算法是第一个既能用于数据加密也能用于数字签名的算法，因此它为公用网络上信息的加密和鉴别提供了一种基本的方法。它通常是先生成一对 RSA 密钥，其中之一是保密密钥，由用户保存；另一个为公开密钥，可对外公开，甚至可在网络服务器中注册，人们用公钥加密文件发送给个人，个人就可以用私钥解密接受。为提高保密强度，RSA 密钥至少为 500 位长，一般推荐使用 1024 位。

RSA 算法是 R.Rivest、A.Shamir 和 L.Adleman 于 1977 年在美国麻省理工学院开发，于 1978 年首次公布。

RSA 公钥密码算法是目前网络上进行保密通信和数字签名的最有效的安全算法之一。RSA 算法的安全性基于数论中大素数分解的困难性，所以，RSA 需采用足够大的整数。因子分解越困难，密码就越难以破译，加密强度就越高。

其算法如下：

A. 选择两质数 p、q

B. 计算 $n = p * q$

C. 计算 n 的欧拉函数 $\Phi(n) = (p - 1)(q - 1)$

D. 选择整数 e，使 e 与 $\Phi(n)$ 互质，且 $1 < e < \Phi(n)$

E. 计算 d，使 $d * e = 1 \bmod \Phi(n)$

其中，公钥 KU＝{e, n}，私钥 KR={d, n}。

加密/解密过程：

利用 RSA 加密，首先需将明文数字化，取长度小于 log2n 位的数字作为明文块。

对于明文块 M 和密文块 C，加/解密的形式如下：

加密： C = Me mod n

解密： M = Cd mod n = (Me)d mod n = Med mod n

RSA 的安全性基于大数分解质因子的困难性。因为若 n 被分解为 $n = p * q$，则$\Phi(n)$、e、d可依次求得。目前，因式分解速度最快的方法的时间复杂性为 exp(sqrt(ln(n)lnln(n)))。统计数据表明，在重要应用中，使用 512 位的密钥已不安全，需要采用 1024 位的密钥。

（2）DSA 算法

DSA（Digital Signature Algorithm，数字签名算法，用作数字签名标准的一部分），它是另一种公开密钥算法，它不能用作加密，只用作数字签名。DSA 使用公开密钥，为接受者验证数据的完整性和数据发送者的身份。它也可用于由第三方去确定签名和所签数据的真实性。DSA 算法的安全性基于解离散对数的困难性，这类签字标准具有较大的兼容性和适用性，成为网络安全体系的基本构件之一。

DSA 签名算法中用到了以下参数：

p 是 L 位长的素数，其中 L 从 512 到 1024 且是 64 的倍数。

q 是 160 位长且与 p-1 互素的因子，其中 h 是小于 p-1 并且满足 大于 1 的任意数。

x 是小于 q 的数。

另外，算法使用一个单向散列函数 H（m）。标准指定了安全散列算法（SHA）。三个参数 p，q 和 g 是公开的，且可以被网络中所有的用户公有。私人密钥是 x，公开密钥是 y。

对消息 m 签名时：

（1） 发送者产生一个小于 q 的随机数 k。

（2） 发送者产生：

r 和 s 就是发送者的签名，发送者将它们发送给接受者。

（3） 接受者通过计算来验证签名：

如果 v=r，则签名有效。

（3）Diffie-Hellman 密钥交换

DH 算法是 W.Diffie 和 M.Hellman 提出的。此算法是最早的公钥算法。它实质是一个通信双方进行密钥协定的协议：两个实体中的任何一个使用自己的私钥和另一实体的公钥，得到一个对称密钥，这一对称密钥其它实体都计算不出来。DH 算法的安全性基于有限域上计算离散对数的困难性。离散对数的研究现状表明：所使用的 DH 密钥至少需要 1024 位，才能保证有足够的中、长期安全。

(4) 椭圆曲线密码体制(ECC)

1985 年，N. Koblitz 和 V. Miller 分别独立提出了椭圆曲线密码体制(ECC)，其依据就是定义在椭圆曲线点群上的离散对数问题的难解性。

为了用椭圆曲线构造密码系统，首先需要找到一个单向陷门函数，椭圆曲线上的数量乘就是这样的单向陷门函数。

椭圆曲线的数量乘是这样定义的：设 E 为域 K 上的椭圆曲线，G 为 E 上的一点，这个点被一个正整数 k 相乘的乘法定义为 k 个 G 相加，因而有

kG = G + G + ⋯ + G (共有 k 个 G)

若存在椭圆曲线上的另一点 N ≠ G，满足方程 kG = N。容易看出，给定 k 和 G，计算 N 相对容易。而给定 N 和 G，计算 $k = \log_G N$ 相对困难。这就是椭圆曲线离散对数问题。

离散对数求解是非常困难的。椭圆曲线离散对数问题比有限域上的离散对数问题更难求

解。对于有理点数有大素数因子的椭圆离散对数问题，目前还没有有效的攻击方法。

1.4 回调函数

Openssl 中大量用到了回调函数。回调函数一般定义在数据结构中，是一个函数指针。通过回调函数，客户可以自行编写函数，让 openssl 函数来调用它，即用户调用 openssl 提供的函数，openssl 函数再回调用户提供的函数。这样方便了用户对 openssl 函数操作的控制。在 openssl 实现函数中，它一般会实现一个默认的函数来进行处理，如果用户不设置回调函数，则采用它默认的函数。

回调函数举例：

头文件：

```
#ifndef RANDOM_H
#define RANDOM_H 1
typedef int *callback_random(char *random,int len);
void    set_callback(callback_random *cb);
int     genrate_random(char *random,int len);
#endif
```

源代码：

```
#include "random.h"
#include <stdio.h>
callback_random *cb_rand=NULL;
static int default_random(char *random,int len
{
    memset(random,0x01,len);
    return 0;
}
void    set_callback(callback_random *cb)
{
    cb_rand=cb;
}
int     genrate_random(char *random,int len)
{
    if(cb_rand==NULL)
        return default_random(random,len);
    else
        return cb_rand(random,len);
    return 0;
}
```

测试代码：

```
#include "random.h"
static int my_rand(char *rand,int len)
{
```

```
        memset(rand,0x02,len);
        return 0;
    }
    int     main()
    {
        char    random[10];
        int     ret;
        set_callback(my_rand);
        ret=genrate_random(random,10);
        return 0;
    }
```

本例子用来生产简单的随机数，如果用户提供了生成随机数回调函数，则生成随机数采用用户的方法，否则采用默认的方法。

第二章 openssl 简介

2.1 openssl 简介

openssl是一个功能丰富且自包含的开源安全工具箱。它提供的主要功能有：SSL协议实现(包括SSLv2、SSLv3和TLSv1)、大量软算法(对称/非对称/摘要)、大数运算、非对称算法密钥生成、ASN.1编解码库、证书请求(PKCS10)编解码、数字证书编解码、CRL编解码、OCSP协议、数字证书验证、PKCS7标准实现和PKCS12个人数字证书格式实现等功能。

openssl采用C语言作为开发语言，这使得它具有优秀的跨平台性能。openssl支持Linux、UNIX、windows、Mac等平台。openssl目前最新的版本是0.9.8e.

2.2 openssl 安装

对应不同的操作系统，用户可以参考 INSTALL、INSTALL.MacOS、INSTALL.NW、INSTALL.OS2、INSTALL.VMS、INSTALL.W32、INSTALL.W64 和 INSTALL.WCE 等文件来安装 openssl。安装时，需要如下条件：

Make 工具、Perl 5、编译器以及 C 语言库和头文件。

2.2.1 linux 下的安装

1）解压 openssl 开发包文件；

2）运行./config --prefix=/usr/local/openssl (更多选项用./config --help 来查看),可用的选项有：no-mdc2、no-cast no-rc2、no-rc5、no-ripemd、no-rc4 no-des 、no-md2、no-md4、no-idea 、no-aes、no-bf、no-err、no-dsa、no-dh、no-ec、no-hw、no-asm、no-krb5、no-dso 、no-threads 、no-zlib、-DOPENSSL_NO_HASH_COMP、-DOPENSSL_NO_ERR、-DOPENSSL_NO_HW 、-DOPENSSL_NO_OCSP、-DOPENSSL_NO_SHA256 和-DOPENSSL_NO_SHA512 等。去掉不必要的内容可以减少生成库的大小。 若要生成 debug 版本的库和可执行程序加-g 或者-g3(openssl 中有很多宏，需要调试学习最好加上-g3)。

3）make test (可选)

4）make install

完成后,openssl 会被安装到/usr/local/openssl 目录，包括头文件目录 include、可执行文件目录 bin、man 在线帮助、库目录 lib 以及配置文件目录(ssl)。

2.2.2 windows 编译与安装

安装步骤如下：

1） 安装 VC6.0；0.9.7i 及以上版本支持 VC++ 2005

2) 安装 perl5；

3） 解压 openssl；

4） 在控制台下进入 openssl 目录；

5） 运行 perl Configure VC-WIN32，其他可选项参见 2.2.1 节；

6） ms\do_ms.bak

7） nmake -f ms\ntdll.mak（动态库）或者 nmake –f ms\nt.mak(静态库)；

编译 debug 版本在 ms\do_ms.bat 中加上 debug,，见 INSTALL.W32，具体做法如下：

编辑 do_ms.bak，修改前内容如下：

```
perl util\mkfiles.pl >MINFO
perl util\mk1mf.pl no-asm VC-WIN32 >ms\nt.mak
perl util\mk1mf.pl dll no-asm VC-WIN32 >ms\ntdll.mak
perl util\mk1mf.pl no-asm VC-CE >ms\ce.mak
perl util\mk1mf.pl dll no-asm VC-CE >ms\cedll.mak
perl util\mkdef.pl 32 libeay > ms\libeay32.def
perl util\mkdef.pl 32 ssleay > ms\ssleay32.def
```

添加 debug 后如下：

```
perl util\mkfiles.pl >MINFO
perl util\mk1mf.pl debug no-asm VC-WIN32 >ms\nt.mak                #添加 debug
perl util\mk1mf.pl debug dll no-asm VC-WIN32 >ms\ntdll.mak         #添加 debug
perl util\mk1mf.pl debug no-asm VC-CE >ms\ce.mak                   #添加 debug
perl util\mk1mf.pl debug dll no-asm VC-CE >ms\cedll.mak            #添加 debug
perl util\mkdef.pl 32 libeay > ms\libeay32.def
perl util\mkdef.pl 32 ssleay > ms\ssleay32.def
```

安装完毕后，生成的头文件放在 inc32 目录，动/静态库和可执行文件放在 outdll 目录。

另外用户可以在 windows 集成环境下建自己的工作环境，来编译 openssl，操作比较烦琐，也可以从网上址下载已有的 vc6.0 工程。

2.3 openssl 源代码

openssl 源代码主要由 eay 库、ssl 库、工具源码、范例源码以及测试源码组成。

eay 库是基础的库函数，提供了很多功能。源代码放在 crypto 目录下。包括如下内容：

1） asn.1 DER 编码解码(crypto/asn1 目录)，它包含了基本 asn1 对象的编解码以及数字证书请求、数字证书、CRL 撤销列表以及 PKCS8 等最基本的编解码函数。这些函数主要通过宏来实现。

2） 抽象 IO(BIO,crypto/bio 目录)，本目录下的函数对各种输入输出进行抽象，包括文件、内存、标准输入输出、socket 和 SSL 协议等。

3） 大数运算(crypto/bn 目录)，本目录下的文件实现了各种大数运算。这些大数运算主要用于非对称算法中密钥生成以及各种加解密操作。另外还为用户提供了大量辅助函数，比如内存与大数之间的相互转换。

4） 字符缓存操作(crypto/buffer 目录)。

5） 配置文件读取(crypto/conf 目录)，openssl 主要的配置文件为 openssl.cnf。本目录下的函数实现了对这种格式配置文件的读取操作。

6） DSO(动态共享对象,crypto/dso 目录)，本目录下的文件主要抽象了各种平台的动态库加载函数，为用户提供统一接口。

7） 硬件引擎(crypto/engine 目录)，硬件引擎接口。用户如果要写自己的硬件引擎，必须实现它所规定的接口。

8） 错误处理(crypto/err 目录)，当程序出现错误时，openssl 能以堆栈的形式显示各个错误。本目录下只有基本的错误处理接口，具体的的错误信息由各个模块提供。各个模块专门用于错误处理的文件一般为*_err..c 文件。

9） 对称算法、非对称算法及摘要算法封装(crypto/evp 目录)。

10） HMAC(crypto/hmac 目录)，实现了基于对称算法的 MAC。

11） hash 表(crypto/lhash 目录)，实现了散列表数据结构。openssl 中很多数据结构都是以散列表来存放的。比如配置信息、ssl session 和 asn.1 对象信息等。

12） 数字证书在线认证(crypto/ocsp 目录)，实现了 ocsp 协议的编解码以及证书有效性计算等功能。

13） PEM 文件格式处理(crypto/pem)，用于生成和读取各种 PEM 格式文件，包括各种密钥、数字证书请求、数字证书、PKCS7 消息和 PKCS8 消息等。

14） pkcs7 消息语法(crypto/pkcs7 目录)，主要实现了构造和解析 PKCS7 消息；

15） pkcs12 个人证书格式(crypto/pckcs12 目录)，主要实现了 pkcs12 证书的构造和解析。

16） 队列(crypto/pqueue 目录)，实现了队列数据结构，主要用于 DTLS。

17） 随机数(crypto/rand 目录)，实现了伪随机数生成，支持用户自定义随机数生成。

18） 堆栈(crypto/stack 目录)，实现了堆栈数据结构。

19） 线程支持(crypto/threads)，openssl 支持多线程，但是用户必须实现相关接口。

20） 文本数据库(crypto/txt_db 目录)。

21） x509 数字证书(crypto/x509 目录和 crypto/x509v3)，包括数字证书申请、数字证书和 CRL 的构造、解析和签名验证等功能了；

22） 对称算法(crypto/aes、crypto/bf、crypto/cast、ccrypto/omp 和 crypto/des 等目录)。

23） 非对称算法(crypto/dh、crypto/dsa、crypto/ec 和 crypto/ecdh)。

24） 摘要算法(crypto/md2、crypto/md4、crypto/md5 和 crypto/sha)以及密钥交换/认证算法(crypto/dh 和 crypto/krb5)。

ssl 库所有源代码在 ssl 目录下，包括了 sslv2、sslv3、tlsv1 和 DTLS 的源代码。各个版本基本上都有客户端源码(*_clnt.c)、服务源码(*_srvr.c)、通用源码(*_both.c)、底层包源码（*_pkt.c)、方法源码(*_meth.c)以及协议相关的各种密钥计算源码(*_enc.c)等，都很有规律。

工具源码主要在 crypto/apps 目录下，默认编译时只编译成 openssl(windows 下为 openssl.exe)可执行文件。该命令包含了各种命令工具。此目录下的各个源码可以单独进行编译。

范例源码在 demo 目录下，另外 engines 目录给出了 openssl 支持的几种硬件的 engines 源码，也可以作为 engine 编写参考。

测试源码主要在 test 目录下。

2.4 openssl 学习方法

通过学习 openssl，用户能够学到 PKI 方面的各种知识，其重要性不言而喻。以下为学习 openssl 的方法，供参考。

1） 建立学习环境

建立一个供调试的 openssl 环境，可以是 windows 平台，也可以是 linux 或者其他平台。用户需有在这些平台下调试源代码的能力。

2）学习 openssl 的命令

通过 openssl 命令的学习，对 openssl 有基本的了解。

3） 学习 openssl 源代码并调试

主要的源代码有：

apps 目录下的各个程序，对应于 openssl 的各项命令；

demos 下的各种源代码；

engines 下的各种 engine 实现；

test 目录下的各种源代码。

对于 openssl 函数的学习，主要查看 openssl 自身是如何调用的，或者查看函数的实现。对于 openssl 中只有实现而没有调用的函数，读者需要自己写源码或研究源代码去学习。

4） 学会使用 openssl 的 asn.1 编解码

openssl 中很多函数和源码都涉及到 asn1 编解码，比如数字证书申请、数字证书、crl、ocsp、pkcs7、pkcs8、pkcs12 等。

5） 查找资料

Linux 下主要用 man 就能查看 openssl 命令和函数的帮助。Windows 用户可用到 www.openss.org 去查看在线帮助文档，或者用 linux 下的命令 man2html 将帮助文档装换为 html 格式。用户也可以访问 openssl.cn 论坛来学习 openssl。

6） 学习 openssl 相关书籍

读者可以参考《OpenSSL与网络信息安全--基础、结构和指令》、《Network Security with OpenSSL》(OReilly出版)和《OpenSSL for windows Developer’s Guide》。

第三章　堆栈

3.1　openssl 堆栈

堆栈是一种先进后出的数据结构。openssl 大量采用堆栈来存放数据。它实现了一个通用的堆栈，可以方便的存储任意数据。它实现了许多基本的堆栈操作，主要有：堆栈拷贝(sk_dup)、构建新堆栈（sk_new_null，sk_new）、插入数据（sk_insert）、删除数据（sk_delete）、查找数据（sk_find，sk_find_ex）、入栈（sk_push）、出栈（sk_pop）、获取堆栈元素个数（sk_num）、获取堆栈值（sk_value）、设置堆栈值（sk_set）和堆栈排序（sk_sort）。

3.2　数据结构

openssl 堆栈数据结构在 stack.h 中定义如下：

```
typedef struct stack_st
{
    int num;
    char **data;
    int sorted;
    int num_alloc;
    int (*comp)(const char * const *, const char * const *);
} STACK;
```

各项意义如下：

num:　堆栈中存放数据的个数。

data:　用于存放数据地址，每个数据地址存放在 data[0]到 data[num-1]中。

sorted:　堆栈是否已排序，如果排序则值为 1，否则为 0，堆栈数据一般是无序的，只有当用户调用了 sk_sort 操作，其值才为 1。

comp:　堆栈内存放数据的比较函数地址，此函数用于排序和查找操作；当用户生成一个新堆栈时，可以指定 comp 为用户实现的一个比较函数；或当堆栈已经存在时通过调用 sk_set_cmp_func 函数来重新指定比较函数。

注意，用户不需要调用底层的堆栈函数(sk_sort、sk_set_cmp_func 等)，而是调用他通过宏实现的各个函数。

3.3　源码

openssl 堆栈实现源码位于 crypto/stack 目录下。下面分析了部分函数。

1) sk_set_cmp_func

此函数用于设置堆栈存放数据的比较函数。由于堆栈不知道用户存放的是什么数据，所以，比较函数必须由用户自己实现。

2) sk_find

根据数据地址来查找它在堆栈中的位置。当堆栈设置了比较函数时，它首先对

堆栈进行排序，然后通过二分法进行查找。如果堆栈没有设置比较函数，它只是简单的比较数据地址来查找.

3） sk_sort

本函数对堆栈数据排序。它首先根据 sorted 来判断是否已经排序，如果未排序则调用了标准 C 函数 qsort 进行快速排序。

4） sk_pop_free

本函数用于释放堆栈内存放的数据以及堆栈本身，它需要一个由用户指定的针对具体数据的释放函数。如果用户仅调用 sk_free 函数，则只会释放堆栈本身所用的内存，而不会释放数据内存。

3.4 定义用户自己的堆栈函数

用户直接调用最底层的堆栈操作函数是一个麻烦的事情，对此 openssl 提供了用宏来帮助用户实现接口。用户可以参考 safestack.h 来定义自己的上层堆栈操作函数，举例如下，safestack.h 定义了如下关于 GENERAL_NAME 数据结构的堆栈操作：

```
#define sk_GENERAL_NAME_new(st) SKM_sk_new(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_new_null() SKM_sk_new_null(GENERAL_NAME)
#define sk_GENERAL_NAME_free(st) SKM_sk_free(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_num(st) SKM_sk_num(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_value(st, i) SKM_sk_value(GENERAL_NAME, (st), (i))
#define sk_GENERAL_NAME_set(st, i, val) SKM_sk_set(GENERAL_NAME, (st), (i), (val))
#define sk_GENERAL_NAME_zero(st) SKM_sk_zero(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_push(st, val) SKM_sk_push(GENERAL_NAME, (st), (val))
#define sk_GENERAL_NAME_unshift(st, val) SKM_sk_unshift(GENERAL_NAME, (st), (val))
#define sk_GENERAL_NAME_find(st, val) SKM_sk_find(GENERAL_NAME, (st), (val))
#define sk_GENERAL_NAME_find_ex(st, val) SKM_sk_find_ex(GENERAL_NAME, (st), (val))
#define sk_GENERAL_NAME_delete(st, i) SKM_sk_delete(GENERAL_NAME, (st), (i))
#define sk_GENERAL_NAME_delete_ptr(st, ptr) SKM_sk_delete_ptr(GENERAL_NAME, (st), (ptr))
#define sk_GENERAL_NAME_insert(st, val, i) SKM_sk_insert(GENERAL_NAME, (st), (val), (i))
#define sk_GENERAL_NAME_set_cmp_func(st, cmp) SKM_sk_set_cmp_func(GENERAL_NAME, (st), (cmp))
#define sk_GENERAL_NAME_dup(st) SKM_sk_dup(GENERAL_NAME, st)
#define sk_GENERAL_NAME_pop_free(st, free_func) SKM_sk_pop_free(GENERAL_NAME, (st), (free_func))
#define sk_GENERAL_NAME_shift(st) SKM_sk_shift(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_pop(st) SKM_sk_pop(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_sort(st) SKM_sk_sort(GENERAL_NAME, (st))
#define sk_GENERAL_NAME_is_sorted(st) SKM_sk_is_sorted(GENERAL_NAME, (st))
```

当用户想对 GENERAL_NAME 数据进行堆栈操作时，调用上面由宏定义的函数即可，即直观又方便。比如用户想设置堆栈数据的比较函数和对堆栈排序时，他分别调用：sk_GENERAL_NAME_set_cmp_func 和 sk_GENERAL_NAME_sort。

3.5 编程示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/safestack.h>

#define sk_Student_new(st) SKM_sk_new(Student, (st))
#define sk_Student_new_null() SKM_sk_new_null(Student)
#define sk_Student_free(st) SKM_sk_free(Student, (st))
#define sk_Student_num(st) SKM_sk_num(Student, (st))
#define sk_Student_value(st, i) SKM_sk_value(Student, (st), (i))
#define sk_Student_set(st, i, val) SKM_sk_set(Student, (st), (i), (val))
#define sk_Student_zero(st) SKM_sk_zero(Student, (st))
#define sk_Student_push(st, val) SKM_sk_push(Student, (st), (val))
#define sk_Student_unshift(st, val) SKM_sk_unshift(Student, (st), (val))
#define sk_Student_find(st, val) SKM_sk_find(Student, (st), (val))
#define sk_Student_delete(st, i) SKM_sk_delete(Student, (st), (i))
#define sk_Student_delete_ptr(st, ptr) SKM_sk_delete_ptr(Student, (st), (ptr))
#define sk_Student_insert(st, val, i) SKM_sk_insert(Student, (st), (val), (i))
#define sk_Student_set_cmp_func(st, cmp) SKM_sk_set_cmp_func(Student, (st), (cmp))
#define sk_Student_dup(st) SKM_sk_dup(Student, st)
#define sk_Student_pop_free(st, free_func) SKM_sk_pop_free(Student, (st), (free_func))
#define sk_Student_shift(st) SKM_sk_shift(Student, (st))
#define sk_Student_pop(st) SKM_sk_pop(Student, (st))
#define sk_Student_sort(st) SKM_sk_sort(Student, (st))

typedef    struct      Student_st
{
      char        *name;
      int         age;
      char        *otherInfo;
}Student;
typedef    STACK_OF(Student)     Students;

Student *Student_Malloc()
{
      Student *a=malloc(sizeof(Student));
      a->name=malloc(20);
      strcpy(a->name,"zcp");
      a->otherInfo=malloc(20);
      strcpy(a->otherInfo,"no info");
      return a;
}
```

```
void Student_Free(Student *a)
{
	free(a->name);
	free(a->otherInfo);
	free(a);
}

static	int Student_cmp(Student *a,Student *b)
{
	int		ret;

	ret=strcmp(a->name,b->name);
	return ret;
}

int	main()
{
	Students	*s,*snew;
	Student		*s1,*one,*s2;
	int		i,num;

	s=sk_Student_new_null();
	snew=sk_Student_new(Student_cmp);
	s2=Student_Malloc();
	sk_Student_push(snew,s2);
	i=sk_Student_find(snew,s2);
	s1=Student_Malloc();
	sk_Student_push(s,s1);
	num=sk_Student_num(s);
	for(i=0;i<num;i++)
	{
		one=sk_Student_value(s,i);
		printf("student name :	%s\n",one->name);
		printf("sutdent age	:	%d\n",one->age);
		printf("student otherinfo :	%s\n\n\n",one->otherInfo);
	}
	sk_Student_pop_free(s,Student_Free);
	sk_Student_pop_free(snew,Student_Free);
	return 0;
	}
```

第四章　哈希表

4.1　哈希表

在一般的数据结构如线性表和树中，记录在结构中的相对位置是与记录的关键字之间不存在确定的关系，在结构中查找记录时需进行一系列的关键字比较。这一类查找方法建立在“比较”的基础上，查找的效率与比较次数密切相关。理想的情况是能直接找到需要的记录，因此必须在记录的存储位置和它的关键字之间建立确定的对应关系，使每个关键字和结构中一个唯一的存储位置相对应。在查找时，只需根据这个对应关系找到给定值。这种对应关系既是哈希函数，按这个思想建立的表为哈希表。

哈希表存在冲突现象：不同的关键字可能得到同一哈希地址。在建造哈希表时不仅要设定一个好的哈希函数，而且要设定一种处理冲突的方法。

4.2　哈希表数据结构

openssl 函数使用哈希表来加快查询操作，并能存放任意形式的数据，比如配置文件的读取、内存分配中被分配内存的信息等。其源码在 crypto/lhash 目录下。

openssl 中的哈希表数据结构在 lhash.h 中定义如下：

```
typedef struct lhash_node_st
        {
        void *data;
        struct lhash_node_st *next;
    #ifndef OPENSSL_NO_HASH_COMP
        unsigned long hash;
    #endif
        } LHASH_NODE;
```

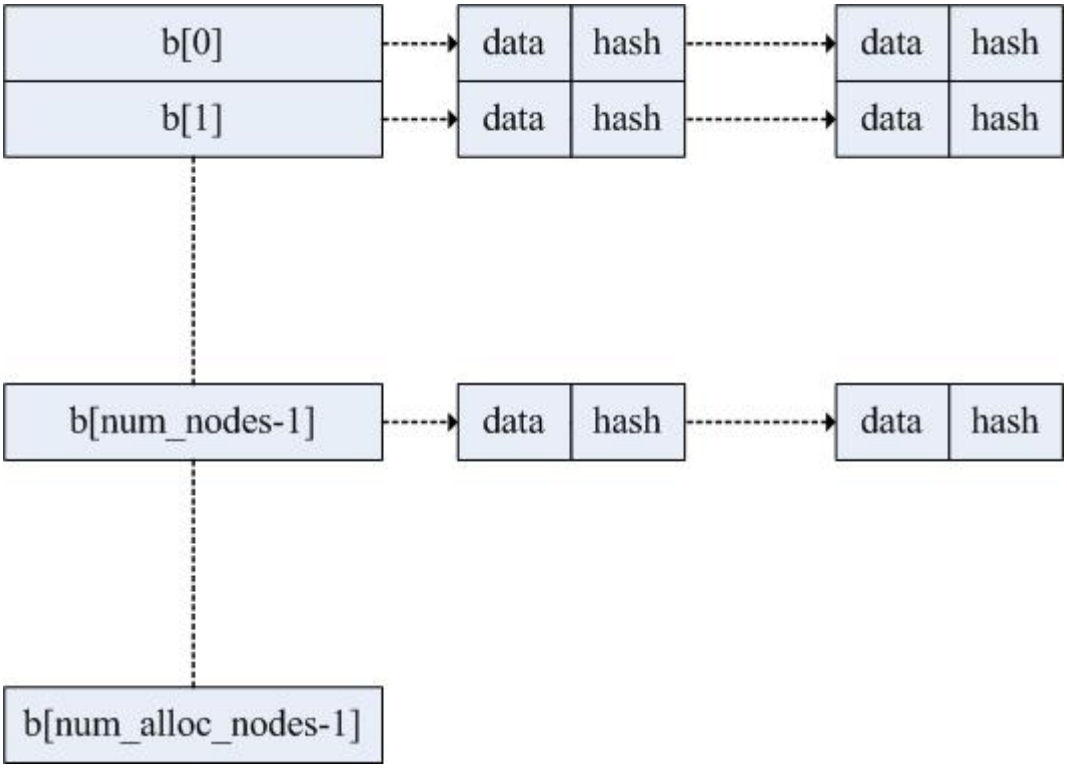
本结构是一个单链表。其中，data 用于存放数据地址，next 为下一个数据地址，hash 为数据哈希计算值。

```
    typedef struct lhash_st
        {
        LHASH_NODE **b;
        LHASH_COMP_FN_TYPE comp;
        LHASH_HASH_FN_TYPE hash;
        unsigned int num_nodes;
        unsigned int num_alloc_nodes;
        unsigned int p;
        unsigned int pmax;
        unsigned long up_load; /* load times 256 */
        unsigned long down_load; /* load times 256 */
        unsigned long num_items;
```

```
unsigned long num_expands;
unsigned long num_expand_reallocs;
unsigned long num_contracts;
unsigned long num_contract_reallocs;
unsigned long num_hash_calls;
unsigned long num_comp_calls;
unsigned long num_insert;
unsigned long num_replace;
unsigned long num_delete;
unsigned long num_no_delete;
unsigned long num_retrieve;
unsigned long num_retrieve_miss;
unsigned long num_hash_comps;
int error;
} LHASH;
```

其中，b 指针数组用于存放所有的数据，数组中的每一个值为数据链表的头指针；comp 用于存放数据比较函数地址；hash 用于存放计算哈希值函数的地址；num_nodes 为链表个数；num_alloc_nodes 为 b 分配空间的大小。

基本的结构如下示图：



4.3 函数说明

1) LHASH *lh_new(LHASH_HASH_FN_TYPE h, LHASH_COMP_FN_TYPE c)

功能：生成哈希表

源文件：lhash.c

说明：输入参数 h 为哈希函数，c 为比较函数。这两个函数都是回调函数。 因为哈希表用于存放任意的数据结构，哈希表存放、查询、删除等操作都需要比较数据和进行哈希运算，而哈希表不知道用户数据如何进行比较，也不知道用户数据结构中需要对哪些关键项进行散列运算。所以，用户必须提供这两个回调函数。

2) void *lh_delete(LHASH *lh, const void *data)

源文件：lhash.c

功能：删除散列表中的一个数据

说明：data 为数据结构指针。

3) void lh_doall(LHASH *lh, LHASH_DOALL_FN_TYPE func)

源文件：lhash.c

功能：处理哈希表中的所有数据

说明：func 为外部提供的回调函数，本函数遍历所有存储在哈希表中的数据，每个数据被 func 处理。

4) void lh_doall_arg(LHASH *lh, LHASH_DOALL_ARG_FN_TYPE func, void *arg)

源文件：lhash.c

功能：处理哈希表中所有数据

说明：此参数类似于 lh_doall 函数，func 为外部提供的回调函数，arg 为传递给 func 函数的参数。本函数遍历所有存储在哈希表中的数据，每个数据被 func 处理。

5) void lh_free(LHASH *lh)

源文件：lhash.c

功能：释放哈希表。

6） void *lh_insert(LHASH *lh, void *data)

源文件：lhash.c

功能：往哈希表中添加数据。

说明：data 为需要添加数据结构的指针地址。

7） void *lh_retrieve(LHASH *lh, const void *data)

源文件：lhash.c

功能：查询数据。

说明：从哈希表中查询数据，data 为数据结构地址，此数据结构中必须提供关键项(这些关键项对应于用户提供的哈希函数和比较函数)以供查询，如果查询成功，返回数据结构的地址，否则返回 NULL。比如 SSL 握手中服务端查询以前存储的 SESSION 时，它需要提供其中关键的几项：

```
SSL_SESSION *ret=NULL,data;
data.ssl_version=s->version;
data.session_id_length=len;
memcpy(data.session_id,session_id,len);
ret=(SSL_SESSION *)lh_retrieve(s->ctx->sessions,&data);
```

8） void lh_node_stats_bio(const LHASH *lh, BIO *out)

源文件：lh_stats.c

功能：将哈希表中每个链表下的数据状态输出到 BIO 中。

9） void lh_node_stats(const LHASH *lh, FILE *fp)

源文件：lh_stats.c

功能：将哈希表中每个链表下数据到个数输出到 FILE 中。

说明：此函数调用了 lh_node_stats_bio 函数。

10）void lh_node_usage_stats_bio(const LHASH *lh, BIO *out)

源文件：lh_stats.c

功能：将哈希表的使用状态输出到 BIO 中。

11）void lh_node_usage_stats(const LHASH *lh, FILE *fp)

源文件：lh_stats.c

功能：将哈希表的使用状态输出到 FILE 中

说明：此函数调用了 lh_node_usage_stats_bio 函数

12）unsigned long lh_num_items(const LHASH *lh)

源文件：lhash.c

功能：获取哈希表中元素的个数。

13）void lh_stats_bio(const LHASH *lh, BIO *out)

源文件：lh_stats.c

功能：输出哈希表统计信息到 BIO 中

14）void lh_stats(const LHASH *lh, FILE *fp)

源文件：lh_stats.c

功能：打印哈希表的统计信息，此函数调用了 lh_stats_bio。

15）unsigned long lh_strhash(const char *c)

源文件：lhash.c

功能：计算文本字符串到哈希值。

4.4 编程示例

```
#include <string.h>
#include <openssl/lhash.h>

typedef  struct   Student_st
{
    char name[20];
    int  age;
    char otherInfo[200];
}Student;
static int Student_cmp(const void *a, const void *b)
{
    char *namea=((Student *)a)->name;
    char *nameb=((Student *)b)->name;
    return strcmp(namea,nameb);
}
/* 打印每个值*/
static void PrintValue(Student *a)
{
    printf("name :%s\n",a->name);
    printf("age    :%d\n",a->age);
    printf("otherInfo : %s\n",a->otherInfo);
```

```
}
static void PrintValue_arg(Student *a,void *b)
{
    int  flag=0;

    flag=*(int *)b;
    printf("用户输入参数为:%d\n",flag);
    printf("name :%s\n",a->name);
    printf("age    :%d\n",a->age);
    printf("otherInfo : %s\n",a->otherInfo);
}
int  main()
{
    int      flag=11;
    LHASH    *h;
    Student  s1={"zcp",28,"hu bei"},
             s2={"forxy",28,"no info"},
             s3={"skp",24,"student"},
             s4={"zhao_zcp",28,"zcp's name"},
             *s5;
    void *data;

    h=lh_new(NULL,Student_cmp);
    if(h==NULL)
    {
        printf("err.\n");
        return -1;
    }
    data=&s1;
    lh_insert(h,data);
    data=&s2;
    lh_insert(h,data);
    data=&s3;
    lh_insert(h,data);
    data=&s4;
    lh_insert(h,data);
    /* 打印*/
    lh_doall(h,PrintValue);
    lh_doall_arg(h,PrintValue_arg,(void *)(&flag));
    data=lh_retrieve(h,(const void*)"skp");
    if(data==NULL)
    {
        printf("can not look up skp!\n");
        lh_free(h);
```

```
        return -1;
    }
    s5=data;
    printf("student name : %s\n",s5->name);
    printf("sutdent    age  :   %d\n",s5->age);
    printf("student otherinfo : %s\n",s5->otherInfo);
    lh_free(h);
    getchar();
    return 0;
}
```

第五章　内存分配

5.1　openssl 内存分配

用户在使用内存时，容易犯的错误就是内存泄露。当用户调用内存分配和释放函数时，查找内存泄露比较麻烦。openssl 提供了内置的内存分配/释放函数。如果用户完全调用 openssl 的内存分配和释放函数，可以方便的找到内存泄露点。openssl 分配内存时，在其内部维护一个内存分配哈希表，用于存放已经分配但未释放的内存信息。当用户申请内存分配时，在哈希表中添加此项信息，内存释放时删除该信息。当用户通过 openssl 函数查找内存泄露点时，只需查询该哈希表即可。用户通过 openssl 回调函数还能处理那些泄露的内存。

openssl 供用户调用的内存分配等函数主要在 crypto/mem.c 中实现，其内置的分配函数在 crypto/mem_dbg.c 中实现。默认情况下 mem.c 中的函数调用 mem_dbg.c 中的实现。如果用户实现了自己的内存分配函数以及查找内存泄露的函数，可以通过调用 CRYPTO_set_mem_functions 函数和 CRYPTO_set_mem_debug_functions 函数来设置。下面主要介绍了 openssl 内置的内存分配和释放函数。

5.2　内存数据结构

openssl 内存分配数据结构是一个内部数据结构，定义在 crypto/mem_dbg.c 中。如下所示：

```
typedef struct app_mem_info_st
{
    unsigned long thread;
    const char *file;
    int line;
    const char *info;
    struct app_mem_info_st *next; /* tail of thread's stack */
    int references;
} APP_INFO;
typedef struct mem_st
{
    void *addr;
    int num;
    const char *file;
    int line;
    unsigned long thread;
    unsigned long order;
    time_t time;
    APP_INFO *app_info;
} MEM;
```

各项意义：

addr：分配内存的地址。

num：分配内存的大小。

file：分配内存的文件。

line：分配内存的行号。

thread：分配内存的线程 ID。

order：第几次内存分配。

time：内存分配时间。

app_info:用于存放用户应用信息，为一个链表，里面存放了文件、行号以及线程 ID 等信息。

references：被引用次数。

5.3 主要函数

1) CRYPTO_mem_ctrl

本函数主要用于控制内存分配时，是否记录内存信息。如果不记录内存信息，将不能查找内存泄露。开启内存记录调用 CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_ON)，关闭内存记录调用 CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_OFF)。一旦 CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_ON) 被调用，直到用户调用 CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_OFF)前，用户所有的 opessl 内存分配都会被记录。

2) CRYPTO_is_mem_check_on

查询内存记录标记是否开启。

3） CRYPTO_dbg_malloc

本函数用于分配内存空间，如果内存记录标记开启，则记录用户申请的内存。当需要记录内存信息时，该函数本身也需要申请内存插入哈希表，为了防止递归申请错误，它申请内存记录信息前必须暂时关闭内存记录标记，申请完毕再放开。

4） CRYPTO_dbg_free

释放内存，如果内存记录标记开启，还需要删除哈希表中对应的记录。

5） CRYPTO_mem_leaks

将内存泄露输出到 BIO 中。

6） CRYPTO_mem_leaks_fp

将内存泄露输出到 FILE 中(文件或者标准输出)，该函数调用了 CRYPTO_mem_leaks。

7） CRYPTO_mem_leaks_cb

处理内存泄露，输入参数为用户自己实现的处理内存泄露的函数地址。该函数只需要处理一个内存泄露，openssl 通过 lh_doall_arg 调用用户函数来处理所有记录(泄露的内存)。

5.4 编程示例

1）示例 1

```
#include <string.h>
#include <openssl/crypto.h>
int   main()
{
      char *p;
      int         i;
      p=OPENSSL_malloc(4);
      p=OPENSSL_remalloc(p,40);
      p=OPENSSL_realloc(p,32);
      for(i=0;i<32;i++)
            memset(&p[i],i,1);
      /* realloc 时将以前的内存区清除(置乱) */
      p=OPENSSL_realloc_clean(p,32,77);
      p=OPENSSL_remalloc(p,40);
      OPENSSL_malloc_locked(3);
      OPENSSL_free(p);
      return 0;
}
```

上述示例使用了基本的 openssl 内存分配和释放函数。

OPENSSL_malloc:　　分配内存空间。

OPENSSL_remalloc：　重新分配内存空间。

OPENSSL_realloc_clean：　重新分配内存空间，将老的数据进行拷贝，置乱老的数据空间并释放。

OPENSSL_malloc_locked　与锁有关。

OPENSSL_free：　　释放空间。

2）示例 2

```
include <openssl/crypto.h>
#include <openssl/bio.h>
int   main()
{
      char *p;
      BIO *b;
      CRYPTO_malloc_debug_init();
      CRYPTO_set_mem_debug_options(V_CRYPTO_MDEBUG_ALL);
      CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_ON);
      p=OPENSSL_malloc(4);
      CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_OFF);
      b=BIO_new_file("leak.log","w");
      CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_ON);
      CRYPTO_mem_leaks(b);
      OPENSSL_free(p);
      BIO_free(b);
      return 0;
}
```

第六章　动态模块加载

6.1　动态库加载

动态库加载函数能让用户在程序中加载所需要的模块，各个平台下的加载函数是不一样的。动态加载函数一般有如下功能：

1） 加载动态库

比如 windows 下的函数 LoadLibraryA；linux 下的函数 dlopen。这些函数一般需要动态库的名字作为参数。

2） 获取函数地址

比如 windows 下的函数 GetProcAddress 已及 linux 下的函数 dlsym。这些函数一般需要函数名作为参数，返回函数地址。

3） 卸载动态库

比如 windows 下的函数 FreeLibrary 和 linux 下的函数 dlclose。

6.2　DSO 概述

DSO 可以让用户动态加载动态库来进行函数调用。各个平台下加载动态库的函数是不一样的，openssl 的 DSO 对各个平台台下的动态库加载函数进行了封装，增加了源码的可移植性。Openssl 的 DSO 功能主要用于动态加载压缩函数（ssl 协议）和 engine(硬件加速引擎)。Openssl 的 DSO 功能除了封装基本的功能外还有其他辅助函数，主要用于解决不同系统下路径不同的表示方式以及动态库全名不一样的问题。比如 windows 系统下路径可以用“\\”和“/”表示，而 linux 下只能使用“/”；windows 下动态库的后缀为.dll 而 linux 下动态库名字一般为 libxxx.so。

6.3　数据结构

dso 数据结定义在 crypto/dso/dso.h 中，如下所示：

```
struct dso_st
{
    DSO_METHOD *meth;
    STACK *meth_data;
    int references;
    int flags;
    CRYPTO_EX_DATA ex_data;
    DSO_NAME_CONVERTER_FUNC name_converter;
    DSO_MERGER_FUNC merger;
    char *filename;
    char *loaded_filename;
};
```

meth：指出了操作系统相关的动态库操作函数。

meth_data：堆栈中存放了加载动态库后的句柄。

reference：引用计数，DSO_new 的时候置 1，DSO_up_ref 时加 1,DSO_free 时减 1。

当调用 DSO_free 时,只有当前的 references 为 1 时才真正释放 meth_data 中存放的句柄。

flag：与加载动态库时加载的文件名以及加载方式有关，用于 DSO_ctrl 函数。

DSO_convert_filename：当加载动态库时会调用 DSO_convert_filename 函数来确定所加载的文件。而 DSO_convert_filename 函数会调用各个系统自己的 convert 函数来获取这个文件名。

对于 flag 有三种种操作命令：设置、读取和或的关系，对应定义如下：

```
#define DSO_CTRL_GET_FLAGS      1
#define DSO_CTRL_SET_FLAGS      2
#define DSO_CTRL_OR_FLAGS       3
```

而 flag 可以设置的值有如下定义：

```
#define DSO_FLAG_NO_NAME_TRANSLATION          0x01
#define DSO_FLAG_NAME_TRANSLATION_EXT_ONLY    0x02
#define DSO_FLAG_UPCASE_SYMBOL                0x10
#define DSO_FLAG_GLOBAL_SYMBOLS                   0x20
```

意义说明如下：

DSO_FLAG_NO_NAME_TRANSLATION

加载的文件名与指定的文件名一致，不加后缀.dll(windows)或.so(linux 或 unix)。

DSO_FLAG_NAME_TRANSLATION_EXT_ONLY

加载的文件名会加上 lib 串，比如用户加载 eay32,真正加载时会加载 libeay32(适用于 linux 或 unix)。

DSO_FLAG_UPCASE_SYMBOL

适用于 OpenVMS。

DSO_FLAG_GLOBAL_SYMBOLS

适用于 unix,当在 unix 下调用加载函数 dlopen 时,参数会被或上 RTLD_GLOBAL。

ex_data：扩展数据，没有使用。

name_converter:：指明了具体系统需要调用的名字计算函数。

loaded_filename：指明了加载动态库的全名。

6.4 编程示例

示例 1：

```
#include <openssl/dso.h>
#include <openssl/bio.h>
int   main()
{
      DSO            *d;
      void           (*f1)();
      void           (*f2)();
      BIO       *(*BIO_newx)(BIO_METHOD *a);
      BIO       *(*BIO_freex)(BIO_METHOD *a);
      BIO       *test;
```

```
        d=DSO_new();
        d=DSO_load(d,"libeay32",NULL,0);
        f1=DSO_bind_func(d,"BIO_new");
        f2=DSO_bind_var(d,"BIO_free");
        BIO_newx=(BIO *(*)(BIO_METHOD *))f1;
        BIO_freex=(BIO *(*)(BIO_METHOD *))f2;
        test=BIO_newx(BIO_s_file());
        BIO_set_fp(test,stdout,BIO_NOCLOSE);
        BIO_puts(test,"abd\n\n");
        BIO_freex(test);
        DSO_free(d);
        return 0;
    }
```

本例动态加载 libeay32 动态库，获取 BIO_new 和 BIO_free 的地址并调用。

示例 2：

```
#include <openssl/dso.h>
#include <openssl/bio.h>
int   main()
{
        DSO     *d;
        void    (*f)();
        BIO     *(*BIO_newx)(BIO_METHOD *a);
        BIO     *test;
        char    *load_name;
        const char      *loaded_name;
        int     flags;

        d=DSO_new();
#if 0
DSO_set_name_converter
DSO_ctrl(d,DSO_CTRL_SET_FLAGS,DSO_FLAG_NO_NAME_TRANSLATION,NULL);
DSO_ctrl(d,DSO_CTRL_SET_FLAGS,DSO_FLAG_NAME_TRANSLATION_EXT_ONLY,NU
LL);
DSO_ctrl(d,DSO_CTRL_SET_FLAGS,DSO_FLAG_GLOBAL_SYMBOLS,NULL);
/* 最好写成 libeay32 而不是 libeay32.dll，除非前面调用了
DSO_ctrl(d,DSO_CTRL_SET_FLAGS,DSO_FLAG_NO_NAME_TRANSLATION,NULL)否则
它会加载 libeay32.dll.dll
*/
load_name=DSO_merge(d,"libeay32","D:\\zcp\\OpenSSL\\openssl-0.9.8b\\out32dll\\Debug");
#endif
        d=DSO_load(d,"libeay32",NULL,0);
        if(d==NULL)
        {
            printf("err");
```

```
        return -1;
    }
    loaded_name=DSO_get_loaded_filename(d);
    if(loaded_name!=NULL)
    {
        printf("loaded file is %s\n",loaded_name);

    }
    flags=DSO_flags(d);
    printf("current falgs is %d\n",flags);
    DSO_up_ref(d);
    f=(void (*)())DSO_bind_var(d,"BIO_new");
    BIO_newx=(BIO *(*)(BIO_METHOD *))f;
    test=BIO_newx(BIO_s_file());
    BIO_set_fp(test,stdout,BIO_NOCLOSE);
    BIO_puts(test,"abd\n\n");
    BIO_free(test);
    DSO_free(d);
    printf("handle in dso number is : %d\n",d->meth_data->num);
    DSO_free(d);
    printf("handle in dso number is : %d\n",d->meth_data->num);
    return 0;
}
```

本例主要演示了 DSO 的控制函数。

第七章　抽象 IO

7.1　openssl 抽象 IO

openssl 抽象 IO(I/O abstraction，即 BIO)是 openssl 对于 io 类型的抽象封装，包括：内存、文件、日志、标准输入输出、socket（TCP/UDP）、加/解密、摘要和 ssl 通道等。Openssl BIO 通过回调函数为用户隐藏了底层实现细节，所有类型的 bio 的调用大体上是类似的。Bio 中的数据能从一个 BIO 传送到另外一个 BIO 或者是应用程序。

7.2　数据结构

BIO 数据结构主要有 2 个，在 crypto/bio.h 中定义如下：

1）BIO_METHOD

```
typedef struct bio_method_st
{
    int type;
    const char *name;
    int (*bwrite)(BIO *, const char *, int);
    int (*bread)(BIO *, char *, int);
    int (*bputs)(BIO *, const char *);
    int (*bgets)(BIO *, char *, int);
    long (*ctrl)(BIO *, int, long, void *);
    int (*create)(BIO *);
    int (*destroy)(BIO *);
    long (*callback_ctrl)(BIO *, int, bio_info_cb *);
} BIO_METHOD;
```

该结构定义了 IO 操作的各种回调函数，根据需要，具体的 bio 类型必须实现其中的一种或多种回调函数，各项意义如下：

type：具体 BIO 类型；

name：具体 BIO 的名字；

bwrite：具体 BIO 写操作回调函数；

bread：具体 BIO 读操作回调函数；

bputs：具体 BIO 中写入字符串回调函数；

bgets：具体 BIO 中读取字符串函数；

ctrl：具体 BIO 的控制回调函数；

create：生成具体 BIO 回调函数；

destroy：销毁具体 BIO 回调函数；

callback_ctrl：具体 BIO 控制回调函数，与 ctrl 回调函数不一样，该函数可由调用者（而不是实现者）来实现，然后通过 BIO_set_callback 等函数来设置。

2）BIO

```
truct bio_st
```

```
{
    BIO_METHOD *method;
    /* bio, mode, argp, argi, argl, ret */
    long (*callback)(struct bio_st *,int,const char *,int, long,long);
    char *cb_arg; /* first argument for the callback */
    int init;
    int shutdown;
    int flags;  /* extra storage */
    int retry_reason;
    int num;
    void *ptr;
    struct bio_st *next_bio;  /* used by filter BIOs */
    struct bio_st *prev_bio;  /* used by filter BIOs */
    int references;
    nsigned long num_read;
    unsigned long num_write;
    CRYPTO_EX_DATA ex_data;
};
```

主要项含义：

init：具体句柄初始化标记，初始化后为 1。比如文件 BIO 中，通过 BIO_set_fp 关联一个文件指针时，该标记则置 1；socket BIO 中通过 BIO_set_fd 关联一个链接时设置该标记为 1。

shutdown：BIO 关闭标记，当该值不为 0 时，释放资源；改值可以通过控制函数来设置。

flags：有些 BIO 实现需要它来控制各个函数的行为。比如文件 BIO 默认该值为 BIO_FLAGS_UPLINK，这时文件读操作调用 UP_fread 函数而不是调用 fread 函数。

retry_reason：重试原因，主要用在 socket 和 ssl BIO 的异步阻塞。比如 socket bio 中，遇到 WSAEWOULDBLOCK 错误时，openssl 告诉用户的操作需要重试。

num：该值因具体 BIO 而异，比如 socket BIO 中 num 用来存放链接字。

ptr：指针，具体 bio 有不同含义。比如文件 BIO 中它用来存放文件句柄；mem bio 中它用来存放内存地址；connect bio 中它用来存放 BIO_CONNECT 数据，accept bio 中它用来存放 BIO_ACCEPT 数据。

next_bio：下一个 BIO 地址，BIO 数据可以从一个 BIO 传送到另一个 BIO，该值指明了下一个 BIO 的地址。

references：被引用数量。

num_read：BIO 中已读取的字节数。

num_write：BIO 中已写入的字节数。

ex_data：用于存放额外数据。

7.3 BIO 函数

BIO 各个函数定义在 crypto/bio.h 中。所有的函数都由 BIO_METHOD 中的回调函数来实现。函数主要分为几类：

1） 具体 BIO 相关函数

比如：BIO_new_file（生成新文件）和 BIO_get_fd（设置网络链接）等。

2） 通用抽象函数

比如 BIO_read 和 BIO_write 等。

另外，有很多函数是由宏定义通过控制函数 BIO_ctrl 实现，比如 BIO_set_nbio、BIO_get_fd 和 BIO_eof 等等。

7.4 编程示例

7.4.1 mem bio

```
#include <stdio.h>
#include <openssl/bio.h>

int        main()
{
      BIO         *b=NULL;
      int         len=0;
      char        *out=NULL;

      b=BIO_new(BIO_s_mem());
      len=BIO_write(b,"openssl",4);
      len=BIO_printf(b,"%s","zcp");
      len=BIO_ctrl_pending(b);
      out=(char *)OPENSSL_malloc(len);
      len=BIO_read(b,out,len);
      OPENSSL_free(out);
      BIO_free(b);
      return 0;
}
```

说明：

b=BIO_new(BIO_s_mem());生成一个 mem 类型的 BIO。

len=BIO_write(b,"openssl",7);将字符串"openssl"写入 bio。

len=BIO_printf(b,"bio test",8);将字符串"bio test"写入 bio。

len=BIO_ctrl_pending(b);得到缓冲区中待读取大小。

len=BIO_read(b,out,50);将 bio 中的内容写入 out 缓冲区。

7.4.2 file bio

```
#include <stdio.h>
#include <openssl/bio.h>
```

```
int     main()
{
        BIO     *b=NULL;
        int     len=0,outlen=0;
        char    *out=NULL;

        b=BIO_new_file("bf.txt","w");
        len=BIO_write(b,"openssl",4);
        len=BIO_printf(b,"%s","zcp");
        BIO_free(b);
        b=BIO_new_file("bf.txt","r");
        len=BIO_pending(b);
        len=50;
        out=(char *)OPENSSL_malloc(len);
        len=1;
        while(len>0)
        {
            len=BIO_read(b,out+outlen,1);
            outlen+=len;
        }
        BIO_free(b);
        free(out);
        return 0;
    }
```

7.4.3 socket bio

服务端：

```
#include <stdio.h>
#include <openssl/bio.h>
#include <string.h>

int     main()
{
    BIO             *b=NULL,*c=NULL;
    int             sock,ret,len;
    char            *addr=NULL;
    char            out[80];

    sock=BIO_get_accept_socket("2323",0);
    b=BIO_new_socket(sock, BIO_NOCLOSE);
    ret=BIO_accept(sock,&addr);
    BIO_set_fd(b,ret,BIO_NOCLOSE);
    while(1)
```

```
        {
            memset(out,0,80);
            len=BIO_read(b,out,80);
            if(out[0]=='q')
                break;
            printf("%s",out);
        }
        BIO_free(b);
        return 0;
}
```

客户端 telnet 此端口成功后，输入字符，服务端会显示出来(linux 下需要输入回车)。

客户端：

```
#include <openssl/bio.h>
int   main()
{
    BIO *cbio, *out;
    int len;
    char tmpbuf[1024];

    cbio = BIO_new_connect("localhost:http");
    out = BIO_new_fp(stdout, BIO_NOCLOSE);
    if(BIO_do_connect(cbio) <= 0)
    {
        fprintf(stderr, "Error connecting to server\n");
    }
    BIO_puts(cbio, "GET / HTTP/1.0\n\n");
    for(;;)
    {
            len = BIO_read(cbio, tmpbuf, 1024);
            if(len <= 0) break;
            BIO_write(out, tmpbuf, len);
    }
    BIO_free(cbio);
    BIO_free(out);
    return 0;
}
```

说明：本示例用来获取本机的 web 服务信息。

cbio = BIO_new_connect("localhost:http");用来生成建立连接到本地 web 服务的 BIO。

out = BIO_new_fp(stdout, BIO_NOCLOSE);生成一个输出到屏幕的 BIO。

BIO_puts(cbio, "GET / HTTP/1.0\n\n");通过 BIO 发送数据。

len = BIO_read(cbio, tmpbuf, 1024);将 web 服务响应的数据写入缓存,此函数循环调用直到无数据。

BIO_write(out, tmpbuf, len);通过 BIO 打印收到的数据。

7.4.4 md BIO

```
#include <openssl/bio.h>
#include <openssl/evp.h>

int        main()
{
    BIO                 *bmd=NULL,*b=NULL;
    const      EVP_MD       *md=EVP_md5();
    int                 len;
    char            tmp[1024];

    bmd=BIO_new(BIO_f_md());
    BIO_set_md(bmd,md);
    b= BIO_new(BIO_s_null());
    b=BIO_push(bmd,b);
    len=BIO_write(b,"openssl",7);
    len=BIO_gets(b,tmp,1024);
    BIO_free(b);
    return 0;
}
```

说明：本示例用 md BIO 对字符串"opessl"进行 md5 摘要。

bmd=BIO_new(BIO_f_md());生成一个 md BIO。

BIO_set_md(bmd,md);设置 md BIO 为 md5 BIO。

b= BIO_new(BIO_s_null());生成一个 null BIO。

b=BIO_push(bmd,b);构造 BIO 链,md5 BIO 在顶部。

len=BIO_write(b,"openssl",7);将字符串送入 BIO 做摘要。

len=BIO_gets(b,tmp,1024);将摘要结果写入 tmp 缓冲区。

7.4.5 cipher BIO

加/解密示例：

```
#include <string.h>
#include <openssl/bio.h>
#include <openssl/evp.h>

int        main()
{
    /*  加密  */
    BIO                 *bc=NULL,*b=NULL;
    const               EVP_CIPHER*c=EVP_des_ecb();
    int                 len,i;
    char            tmp[1024];
```

```
    unsigned char  key[8],iv[8];

    for(i=0;i<8;i++)
    {
        memset(&key[i],i+1,1);
        memset(&iv[i],i+1,1);
    }

    bc=BIO_new(BIO_f_cipher());
    BIO_set_cipher(bc,c,key,iv,1);
    b= BIO_new(BIO_s_null());
    b=BIO_push(bc,b);
    len=BIO_write(b,"openssl",7);
    len=BIO_read(b,tmp,1024);
    BIO_free(b);

    /*  解密  */
    BIO                 *bdec=NULL,*bd=NULL;
    const               EVP_CIPHER*cd=EVP_des_ecb();

    bdec=BIO_new(BIO_f_cipher());
    BIO_set_cipher(bdec,cd,key,iv,0);
    bd= BIO_new(BIO_s_null());
    bd=BIO_push(bdec,bd);
    len=BIO_write(bdec,tmp,len);
    len=BIO_read(bdec,tmp,1024);
    BIO_free(bdec);
    return 0;
}
```

说明：本示例采用 cipher BIO 对字符串"openssl"进行加密和解密,本示例编译需要用 c++编译器;

关键说明:

BIO_set_cipher(bc,c,key,iv,1);设置加密 BI。

BIO_set_cipher(bdec,cd,key,iv,0);设置解密 BIO。

其中 key 为对称密钥,iv 为初始化向量。

加/解密结果通过 BIO_read 获取。

7.4.6 ssl BIO

编程示例：

```
#include <openssl/bio.h>
#include <openssl/ssl.h>

int        main()
```

```
{
    BIO *sbio, *out;
    int len;
    char tmpbuf[1024];
    SSL_CTX *ctx;
    SSL *ssl;

    SSLeay_add_ssl_algorithms();
    OpenSSL_add_all_algorithms();
    ctx = SSL_CTX_new(SSLv3_client_method());
    sbio = BIO_new_ssl_connect(ctx);
    BIO_get_ssl(sbio, &ssl);
    if(!ssl)
    {
        fprintf(stderr, "Can not locate SSL pointer\n");
        return 0;
    }
    SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
    BIO_set_conn_hostname(sbio, "mybank.icbc.com.cn:https");
    out = BIO_new_fp(stdout, BIO_NOCLOSE);
    BIO_printf(out,"链接中….\n");
    if(BIO_do_connect(sbio) <= 0)
    {
        fprintf(stderr, "Error connecting to server\n");
        return 0;
    }
    if(BIO_do_handshake(sbio) <= 0)
    {
        fprintf(stderr, "Error establishing SSL connection\n");
        return 0;
    }
    BIO_puts(sbio, "GET / HTTP/1.0\n\n");
    for(;;)
    {
        len = BIO_read(sbio, tmpbuf, 1024);
        if(len <= 0) break;
        BIO_write(out, tmpbuf, len);
    }
    BIO_free_all(sbio);
    BIO_free(out);
    return 0;
}
```

本函数用 ssl bio 来链接 mybank.icbc.com.cn 的 https 服务，并请求首页文件。其中 SSLeay_add_ssl_algorithms 和 OpenSSL_add_all_algorithms 函数必不可少，否则不能找

到 ssl 加密套件并且不能找到各种算法。

7.4.7 其他示例

```
#include <openssl/bio.h>
#include <openssl/asn1.h>
int    main()
{
       int          ret,len,indent;
       BIO          *bp;
       char *pp,buf[5000];
       FILE         *fp;

       bp=BIO_new(BIO_s_file());
       BIO_set_fp(bp,stdout,BIO_NOCLOSE);
       fp=fopen("der.cer","rb");
       len=fread(buf,1,5000,fp);
       fclose(fp);
       pp=buf;
       indent=5;
       ret=BIO_dump_indent(bp,pp,len,indent);
       BIO_free(bp);
       return 0;
}
```

第八章　配置文件

8.1　概述

Openssl 采用自定义的配置文件来获取配置信息。Openssl 的配置文件主要由如下内容组成：

注释信息，注释信息由#开头；

段信息，段信息由[xxx]来表示，其中 xxx 为段标识；

属性-值信息，表示方法为 a = b，这种信息可以在一个段内也可以不属于任何段。

典型配置文件为 apps/openssl.cnf(同时该文件也是 openssl 最主要的配置文件)。摘取部分内容如下：

```
# OpenSSL example configuration file.
oid_section           = new_oids
[ CA_default ]
dir         = ./demoCA          # Where everything is kept
certs       = $dir/certs        # Where the issued certs are kept
default_days   = 365            #注意，这里是一个数字
```

8.2　openssl 配置文件读取

Openssl 读取配置文件的实现源码在 crypto/conf 中，主要函数定义在 conf.h 中。函数一般以 CONF 或 NCONF(new conf，新函数)开头。本文主要介绍了新的 conf 函数的使用方。主要的数据结构在 crypto/conf.h 中定义如下：

```
typedef struct
{
char *section;
char *name;
char *value;
} CONF_VALUE;
```

section 表明配置文件的段，name 表示这个段中的一个属性，value 则是这个属性的值。Openssl 采用哈希表来存放这些信息，便于快速查找。

8.3　主要函数

1） NCONF_new

生成一个 CONF 结构。

2） CONF_free

释放空间，以及释放存储在散列表中的数据。

3) CONF_load

函数定义：LHASH *CONF_load(LHASH *conf, const char *file, long *eline)，该函数根据输入配置文件名，读取信息存入散列表，如果有错，eline 为错误行。

4) CONF_load_bio/ CONF_load_fp

根据 bio 或者文件句柄读取配置信息并存入散列表。

5） CONF_get_section

给定段信息，得到散列表中的所有对应值。用于获取配置文件中指定某个段下的所有信息,这些信息存放在 CONF_VALUE 的堆栈中。

6） CONF_get_string

给定段以及属性值，得到对应的字符串信息。

7） CONF_get_number

给定段和属性值，获取对应的数值信息。

8） CONF_get1_default_config_file

获取默认的配置文件名，比如 openssl.cnf。

8.4 编程示例

示例 1：

```
#include <openssl/conf.h>
int   main()
{
      CONF     *conf;
      long eline,result;
      int          ret;
      char *p;
      BIO *bp;

      conf=NCONF_new(NULL);
#if 0
      bp=BIO_new_file("openssl.cnf","r");
      NCONF_load_bio(conf,bp,&eline);
#else
      ret=NCONF_load(conf,"openssl.cnf",&eline);
      if(ret!=1)
      {
            printf("err!\n");
            return -1;
      }
#endif
      p=NCONF_get_string(conf,NULL,"certs");
      if(p==NULL)
            printf("no global certs info\n");
      p=NCONF_get_string(conf,"CA_default","certs");
      printf("%s\n",p);
      p=NCONF_get_string(conf,"CA_default","default_days");
      printf("%s\n",p);
      ret=NCONF_get_number_e(conf,"CA_default","default_days",&result);
```

```
    printf("%d\n",result);
    ret=NCONF_get_number(conf,"CA_default","default_days",&result);
    printf("%d\n",result);
    NCONF_free(conf);
    return 0;
}
```

本示例用来读取配置文件信息，这些信息可以是字符串也可以是数字。

示例 2：

NCONF_get_section 的用法：

```
#include <openssl/conf.h>
int   main()
{
    CONF                        *conf;
    BIO                         *bp;
    STACK_OF(CONF_VALUE)        *v;
    CONF_VALUE                  *one;
    int                         i,num;
    long                    eline;

    conf=NCONF_new(NULL);
    bp=BIO_new_file("openssl.cnf","r");
    if(bp==NULL)
    {
        printf("err!\n");
        return -1;
    }
    NCONF_load_bio(conf,bp,&eline);
    v=NCONF_get_section(conf,"CA_default");
    num=sk_CONF_VALUE_num(v);
    printf("section CA_default :\n");
    for(i=0;i<num;i++)
    {
        one=sk_CONF_VALUE_value(v,i);
        printf("%s = %s\n",one->name,one->value);
    }
    BIO_free(bp);
    printf("\n");
    return 0;
}
```

第九章　随机数

9.1　随机数

随机数是一种无规律的数，但是真正做到完全无规律也较困难，所以一般将它称之为伪随机数。随机数在密码学用的很多，比如 ssl 握手中的客户端 hello 和服务端 hello 消息中都有随机数；ssl 握手中的预主密钥是随机数；RSA 密钥生成也用到随机数。如果随机数有问题，会带来很大的安全隐患。

软件生成随机数一般预先设置随机数种子，再生成随机数。设置随机数种子可以说是对生成随机数过程的一种扰乱，让产生的随机数更加无规律可循。

生成随机数有多种方法，可以是某种算法也可以根据某种或多种随机事件来生成。比如，鼠标的位置、系统的当前时间、本进程/线程相关信息以及机器噪声等。

安全性高的应用一般都采用硬件方式(随机数发生器)来生成随机数。

9.2　openssl 随机数数据结构与源码

openssl 生成随机数的源码位于 crypto/rand 目录下。rand.h 定义了许多与随机数生成相关的函数。openssl 通过使用摘要算法来生成随机数，可用的摘要算法有：sha1、md5、mdc2 和 md2。具体采用那种摘要算法在 crypto/rand_lcl.h 中由宏来控制。Openssl 维护一个内部随机状态数据(md_rand.c 中定义的全局变量 state 和 md)，通过对这些内部数据计算摘要来生成随机数。

Openssl 随机数相关的数据结构如下，定义在 rand.h 中：

```
struct rand_meth_st
{
    void (*seed)(const void *buf, int num);
    int (*bytes)(unsigned char *buf, int num);
    void (*cleanup)(void);
    void (*add)(const void *buf, int num, double entropy);
    int (*pseudorand)(unsigned char *buf, int num);
    int (*status)(void);
};
```

本结构主要定义了各种回调函数，如果用户需要实现自己的随机数生成函数，他需要实现本结构中的各个函数。Openssl 给出了一个默认的基于摘要的 rand_meth 实现(crypto/md_rand.c)。各项意义如下：

seed：种子函数，为了让 openssl 内部维护的随机数据更加无序，可使用本函数。buf 为用户输入的随机数地址，num 为其字节数。Openssl 将用户提供的 buf 中的随机内容与其内部随机数据进行摘要计算，更新其内部随机数据。本函数无输出；

bytes：生成随机数，openssl 根据内部维护的随机数状态来生成结果。buf 用于存放生成的随机数。num 为输入参数，用来指明生成随机数的字节长度；

cleanup：清除函数，本函数将内部维护的随机数据清除；

add：与 seed 类似，也是为了让 openssl 内部随机数据更加无序，其中 entropy(信息熵)

可以看作用户本次加入的随机数的个数。Openssl 默认的随机数熵为 32 字节，在 rand_lcl.h 中由 ENTROPY_NEEDED 定义。Openssl 给出随机数之前，用户提供的所有的随机种子数之和必须达到 32 字节。在 openssl 实现的 md_rand 中，即使用户不调用种子函数来直接生成随机数，openssl 也会调用 RAND_poll 函数来完成该操作。

pseudorand：本函数与 bytes 类似也是来生成随机数。

status：查看熵值是否达到预定值，openssl 中为 32 字节，如果达到则返回 1，否则返回 0。在 openssl 实现的 md_rand 中该函数会调用 RAND_poll 函数来使熵值合格。如果本函数返回 0，则说明此时用户不应生成随机数，需要调用 seed 和 add 函数来添加熵值。

cypto/rand 目录下的主要源码有：

1) md_rand.c

   它实现了基于摘要的随机数生成。

2) rand_lib.c

   该文件中的源码简单调用了 rand_meth 中的回调函数。

3） rand_win.c/rand_unix.c/rand_os2.c 等

   这些源码主要提供了平台相关的 RAND_poll 函数实现和其他系统特有函数的实现。比如 rand_win.c 实现了 RAND_screen 函数，用户根据屏幕来设置随机数种子。

4） randfile.c

   用于从随机文件中加载种子、生成随机数文件以及获取随机文件名。比如默认的随机数文件为.rnd 文件，如果找不到该文件，openbsd 可能会返回/dev/arandom。

9.3 主要函数

1) int RAND_load_file(const char *file, long bytes)

   本函数将 file 指定的随机数文件中的数据读取 bytes 字节(如果 bytes 大于 1024，则读取 1024 字节)，调用 RAND_add 进行计算，生成内部随机数。

2) RAND_write_file

   生成一个随机数文件。

3) const char *RAND_file_name(char *file,size_t num)

   获取随机数文件名，如果随机数文件长度小于 num 则返回空，否则返回文件名。

4) RAND_poll

   用于计算内部随机数，各个平台有各自的实现。

5) RAND_screen/RAND_event

   Windows 特有函数，用来计算内部随机数，他们调用了 RAND_seed。

6) RAND_seed/RAND_add

   用来计算内部随机数。

7) RAND_bytes/RAND_pseudo_bytes

   用来计算随机数。

8) RAND_cleanup

   清除内部随机数。

10) RAND_set_rand_method

   用来设置 rand_meth，当用户实现了自己的随机数生成函数时(实现 rand_meth 中的回调函数)，调用该方法来替换 openssl 所提供的随机数功能。

11) RAND_status

用来查看内部随机数熵值是否已达到预定值，如果未达到，则不应该生成随机数。

9.4 编程示例

```
#include <stdio.h>
#include <openssl/bio.h>
#include <openssl/rand.h>
int   main()
{
      char                buf[20],*p;
      unsigned            char out[20],filename[50];
      int                 ret,len;
      BIO                 *print;

      RAND_screen();
      strcpy(buf,"我的随机数");
      RAND_add(buf,20,strlen(buf));
      strcpy(buf,"23424d");
      RAND_seed(buf,20);
      while(1)
      {
            ret=RAND_status();
            if(ret==1)
            {
                  printf("seeded enough!\n");
                  break;
            }
            else
            {
                  printf("not enough sedded!\n");
                  RAND_poll();
            }
      }
      p=RAND_file_name(filename,50);
      if(p==NULL)
      {
            printf("can not get rand file\n");
            return -1;
      }
      ret=RAND_write_file(p);
      len=RAND_load_file(p,1024);
      ret=RAND_bytes(out, 20);
```

```
    if(ret!=1)
    {
        printf("err.\n");
        return -1;
    }
    print=BIO_new(BIO_s_file());
    BIO_set_fp(print,stdout,BIO_NOCLOSE);
    BIO_write(print,out,20);
    BIO_write(print,"\n",2);
    BIO_free(print);
    RAND_cleanup();
    return 0;
}
```

第十章　文本数据库

10.1 概述

Openss 实现了一个简单的文本数据库，它可以从文件读取数据和将数据写到文件中，并且可以根据关键字段来查询数据。Openssl 的文本数据库供 apps/目录下的文件调用，比如 apps.c、ca.c 和 ocsp.c。openssl 文本数据库典型的例子为 apps/demoCA/index.txt。文本数据库一行代表数据库的一行，各个列之间必须用一个\t 隔开，用#进行注释(#必须在开始位置)，以空行结束。比如下面的例子：

赵春平　28　湖北

zcp　28　荆门

文本数据库的查找用到了哈希表。openssl 读取的所有行数据存放在堆栈中，并为每一列数据建立一个单独的哈希表。每个哈希表中存放了所有行数据的地址。查询时，用户指定某一列，openssl 根据对应的哈希表进行查找。

10.2 数据结构

数据结构在 crypto/txt_db/txt_db.h 中定义，如下：

```
typedef struct txt_db_st
{
    int num_fields;
    STACK *data;
    LHASH **index;
    int (**qual)(char **);
    long error;
    long arg1;
    long arg2;
    char **arg_row;
} TXT_DB;
```

意义如下：

num_fields：表明文本数据库的列数。

data：用来存放数据，每一行数据组织成为一个字符串数组(每个数组值对应该行的一列)，　　并将此数组地址 push 到堆栈中。

index：哈希表数组，每一列对应一个哈希表。每一列都可以建哈希表，如果不建哈希表将不能查找该列数据。

qual：一个函数地址数组，数组的每个元素对应一列，进行插入该列哈希表前的过滤。这些函数用于判断一行数据的一列或者多列是否满足某种条件，如果满足将不能插入到哈希表中去(但是能存入堆栈)。每一列都可以设置一个这样的函数。这些函数由用户实现。比如，一个文本数据库中，有名字列和年龄列，并且要求名字长度不能小于 2，年龄不能小于 0 和大于 200。用户为名字列实现了一个 qual 函数，只用来检查名字长度，对于年龄列实现一个 qual 函数，只用来检查年龄。当用户要插入一条记录，名字长度

为 1，但是年龄合法，那么该记录能插入到年龄列对应的哈希表中，而不能插入名字列对应的哈希表。

error、arg1、arg2 和 arg_row 用于存放错误信息。

10.3 函数说明

1） TXT_DB *TXT_DB_read(BIO *in, int num)

用于从 BIO 中读入数据，转换为 TXT_DB，num 用于明确指明列数，本函数不建立哈希表。

2） long TXT_DB_write(BIO *out, TXT_DB *db)

将 TXT_DB 内容写入 BIO；

3） int TXT_DB_create_index(TXT_DB *db,int field,int (*qual)(char **),
LHASH_HASH_FN_TYPE hash, LHASH_COMP_FN_TYPE cmp)

给 field 指定的列建立哈希表。db 为需要建索引的 TXT_DB，hash 为一行数据的 hash 运算回调函数，cmp 为一行数据的比较函数。

4） char **TXT_DB_get_by_index(TXT_DB *db, int idx, char **value)

根据关键字段来查询数据，查询结果返回一行数据 db 为文本数据库，idx 表明采用哪一列的哈希表来查找；value 为查询条件。

5） int TXT_DB_insert(TXT_DB *db,char **value)

往 TXT_DB 中插入一行数据。value 数组以 NULL 表示结束。

6) void TXT_DB_free(TXT_DB *db)

清除 TXT_DB。

10.4 编程示例

```
/* txtdb.dat 的内容
赵春平   28   湖北      无
zcp  28   荆门      无
*/
#include <openssl/bio.h>
#include <openssl/txt_db.h>
#include <openssl/lhash.h>

/*  名字过滤  */
static      int    name_filter(char **in)
{
      if(strlen(in[0])<2)
            return 0;
      return 1;
}

static unsigned long index_name_hash(const char **a)
{
```

```
        const char *n;

         n=a[0];
         while (*n == '0') n++;
         return(lh_strhash(n));
}

static int index_name_cmp(const char **a, const char **b)
{
            const char *aa,*bb;

            for (aa=a[0]; *aa == '0'; aa++);
            for (bb=b[0]; *bb == '0'; bb++);
            return(strcmp(aa,bb));
}

int    main()
{
       TXT_DB        *db=NULL,*out=NULL;
       BIO           *in;
       int           num,ret;
       char        **added=NULL,**rrow=0,**row=NULL;

       in=BIO_new_file("txtdb.dat","r");
       num=1024;
       db=TXT_DB_read(in,4);

       added=(char **)OPENSSL_malloc(sizeof(char *)*(3+1));
             added[0]=(char *)OPENSSL_malloc(10);
       #if 1
       strcpy(added[0],"skp");
       #else
       strcpy(added[0],"a");     /*  不能插入名字对应的哈希表  */
       #endif

       added[1]=(char *)OPENSSL_malloc(10);
       strcpy(added[1],"22");

       added[2]=(char *)OPENSSL_malloc(10);
       strcpy(added[2],"chairman");

       added[3]=NULL;

       ret=TXT_DB_insert(db,added);
```

```
        if(ret!=1)
        {
            printf("err!\n");
            return -1;
        }
        ret=TXT_DB_create_index(db,0, name_filter,index_name_hash,index_name_cmp);
        if(ret!=1)
        {
            printf("err\n");
            return 0;
        }
        row=(char **)malloc(2*sizeof(char *));
            row[0]=(char *)malloc(10);
        strcpy(row[0],"skp");
        row[1]=NULL;
        rrow=TXT_DB_get_by_index(db,0,row);
        if(rrow!=NULL)
            printf("%s      %s   %s\n",rrow[0],rrow[1],rrow[2]);
        out=BIO_new_file("txtdb2.dat","w");
        ret=TXT_DB_write(out,db);
        TXT_DB_free(db);
        BIO_free(in);
        BIO_free(out);
        return 0;
    }
```

本示例只对第一列做了哈希。需要注意的是，added 数组及其元素申请空间时尽量采用 OPENSSL_malloc 而不是 malloc，且其申请的空间由 TXT_DB_free(调用 OPENSSL_free)释放。

第十一章　大数

11.1　介绍

大数一般指的是位数很多的数。计算机表示的数的大小是有限的，精度也是有限的，它不能支持大数运算。密码学中采用了很多大数计算，为了让计算机实现大数运算，用户需要定义自己的大数表示方式并及实现各种大数运算。Openssl 为我们提供了这些功能，主要用于非对称算法。

11.2　openssl 大数表示

crypto/bn.h 中定义了大数的表示方式，如下：

```
struct bignum_st
{
    BN_ULONG *d;
    int top;
    int dmax;
    int neg;
    int flags;
};
```

各项意义如下：

d：BN_ULONG(应系统而异，win32 下为 4 个字节)数组指针首地址，大数就存放在这里面，不过是倒放的。比如，用户要存放的大数为 12345678000（通过 BN_bin2bn 放入），则 d 的内容如下：0x30 0x30 0x30 0x38 0x37 0x36 0x35 0x34 0x33 0x32 0x31；

top：用来指明大数占多少个 BN_ULONG 空间，上例中 top 为 3。

dmax：d 数组的大小。

neg：是否为负数，如果为 1，则是负数，为 0，则为正数。

flags：用于存放一些标记，比如 flags 含有 BN_FLG_STATIC_DATA 时，表明 d 的内存是静态分配的；含有 BN_FLG_MALLOCED 时，d 的内存是动态分配的。

11.3　大数函数

大数函数一般都能根据函数名字知道其实现的功能。下面简单介绍了几个函数。

1） BN_rand/BN_pseudo_rand

生成一个随机的大数。

2） BN_rand_range/BN_pseudo_rand_range

生成随机数，但是给出了随机数的范围。

3） BN_dup

大数复制。

4) BN_generate_prime

生成素数。

5） int BN_add_word(BIGNUM *a, BN_ULONG w)

给大数a加上w，如果成功，返回1。

示例：

```
#include <openssl/bn.h>

int    main()
{
 int              ret;
 BIGNUM           *a;
 BN_ULONG  w;

 a=BN_new();
 BN_one(a);
 w=2685550010;
 ret=BN_add_word(a,w);
 if(ret!=1)
 {
      printf("a+=w err!\n");
      BN_free(a);
      return -1;
 }
 BN_free(a);
 return 0;
}
```

6) BIGNUM *BN_bin2bn(const unsigned char *s, int len, BIGNUM *ret)

将内存中的数据转换为大数，为内存地址，len 为数据长度，ret 为返回值。

示例：

```
#include <openssl/bn.h>
int main()
{
    BIGNUM  *ret1,*ret2;

    ret1=BN_new();
    ret1=BN_bin2bn("242424ab",8, ret1);
    ret2=BN_bin2bn("242424ab",8,NULL);
    BN_free(ret1);
    BN_free(ret2);
    return 0;
}
```

注意：输入参数“242424ab”是asc码，对应的大数值为16进制的0x3234323432346162

7) int BN_bn2bin(const BIGNUM *a, unsigned char *to)

将大数转换为内存形式。输入参数为大数 a，to 为输出缓冲区地址，缓冲区需要预先分配，返回值为缓冲区的长度。

示例：

```
#include <openssl/bn.h>
int   main()
{
    BIGNUM      *ret1=NULL;
    char bin[50],*buf=NULL;
    int         len;

    ret1=BN_bin2bn("242424ab",8, NULL);
    len=BN_bn2bin(ret1,bin);
    len=BN_num_bytes(ret1);
    buf=malloc(len);
    len=BN_bn2bin(ret1,buf);
    free(buf);
    BN_free(ret1);
    return 0;
}
```

本例的缓冲区分配有两种方法：静态分配和动态分配。动态分配时，先调用 BN_num_bytes 函数获取大数对应的缓冲区的大小。

8） char *BN_bn2dec(const BIGNUM *a)

将大数转换成整数字符串。返回值中存放整数字符串，它由内部分配空间，用户必须在外部用OPENSSL_free函数释放该空间。

示例：

```
#include <openssl/bn.h>
#include <openssl/crypto.h>
int main()
{
    BIGNUM  *ret1=NULL;
    char    *p=NULL;
    int     len=0;

    ret1=BN_bin2bn("242424ab",8, NULL);
    p=BN_bn2dec(ret1);
    printf("%s\n",p);  /* 3617571600447332706 */
    BN_free(ret1);
    OPENSSL_free(p);
    getchar();
    return 0;
}
```

9） char *BN_bn2hex(const BIGNUM *a)

将大数转换为十六进制字符串。返回值为生成的十六进制字符串，外部需要用 OPENSSL_free 函数释放

示例：

```
#include <openssl/bn.h>
```

```
#include <openssl/crypto.h>
int main()
{
    BIGNUM  *ret1=NULL;
    char    *p=NULL;
    int     len=0;

    ret1=BN_bin2bn("242424ab",8, NULL);
    p=BN_bn2hex(ret1);
    printf("%s\n",p);
    BN_free(ret1);
    OPENSSL_free(p);
    getchar();
    return 0;
}
```

输出的结果为：323432346162

10) BN_cmp

比较两个大数。

11）BIGNUM *BN_mod_inverse(BIGNUM *in,const BIGNUM *a,
const BIGNUM *n, BN_CTX *ctx)

计算 ax=1(mod n)。

用户使用 openssl 函数编程时，一般用不着进行大数运算。BN_bin2bn、BN_hex2bn、BN_dec2bn、BN_bin2bn、BN_bn2bin、BN_bn2hex 和 BN_bn2dec 比较常用。比如给定 RSA 密钥的内存形式，用户可以调用 BN_bin2bn 来构造 RSA 密钥的大数元素来进行 RSA 运算，或者已经生成了 RSA 密钥，用户调用 BN_bn2bin 将 RSA 各个元素导出到内存中再写入密钥文件。

11.4 使用示例

1）示例 1

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
        BIGNUM      *bn;
        BIO       *b;
        char a[20];
        int         ret;

        bn=BN_new();
        strcpy(a,"32");
        ret=BN_hex2bn(&bn,a);
```

```
    b=BIO_new(BIO_s_file());
    ret=BIO_set_fp(b,stdout,BIO_NOCLOSE);
    BIO_write(b,"aaa",3);
    BN_print(b,bn);
    BN_free(bn);
    return 0;
}
```

2）示例 2

加法运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
    BIGNUM      *a,*b,*add;
    BIO         *out;
    char c[20],d[20];
    int         ret;

    a=BN_new();
    strcpy(c,"32");
    ret=BN_hex2bn(&a,c);
    b=BN_new();
    strcpy(d,"100");
    ret=BN_hex2bn(&b,d);
    out=BIO_new(BIO_s_file());
    ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
    add=BN_new();
    ret=BN_add(add,a,b);
    if(ret!=1)
    {
        printf("err.\n");
        return -1;
    }
    BIO_puts(out,"bn 0x32 + 0x100 = 0x");
    BN_print(out,add);
    BIO_puts(out,"\n");
    BN_free(a);
    BN_free(b);
    BN_free(add);
    BIO_free(out);
    return 0;
```

```
}
```

3） 示例 3

减法运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
      BIGNUM      *a,*b,*sub;
      BIO       *out;
      char c[20],d[20];
      int         ret;

      a=BN_new();
      strcpy(c,"100");
      ret=BN_hex2bn(&a,c);
      b=BN_new();
      strcpy(d,"32");
      ret=BN_hex2bn(&b,d);
      out=BIO_new(BIO_s_file());
      ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
      sub=BN_new();
      ret=BN_sub(sub,a,b);
      if(ret!=1)
      {
            printf("err.\n");
            return -1;
      }
      BIO_puts(out,"bn 0x100 - 0x32 = 0x");
      BN_print(out,sub);
      BIO_puts(out,"\n");
      BN_free(a);
      BN_free(b);
      BN_free(sub);
      BIO_free(out);
      return 0;
}
```

4）示例 4

乘法运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>
```

```
int   main()
{
      BIGNUM      *a,*b,*mul;
      BN_CTX*ctx;
      BIO         *out;
      char c[20],d[20];
      int         ret;

      ctx=BN_CTX_new();
      a=BN_new();
      strcpy(c,"32");
      ret=BN_hex2bn(&a,c);
      b=BN_new();
      strcpy(d,"100");
      ret=BN_hex2bn(&b,d);
      out=BIO_new(BIO_s_file());
      ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
      mul=BN_new();
      ret=BN_mul(mul,a,b,ctx);
      if(ret!=1)
      {
            printf("err.\n");
            return -1;
      }
      BIO_puts(out,"bn 0x32 * 0x100 = 0x");
      BN_print(out,mul);
      BIO_puts(out,"\n");
      BN_free(a);
      BN_free(b);
      BN_free(mul);
      BIO_free(out);
      BN_CTX_free(ctx);
      return 0;
}
```

5）示例 5

除法运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
      BIGNUM      *a,*b,*div,*rem;
      BN_CTX*ctx;
```

```
    BIO     *out;
    char c[20],d[20];
    int     ret;

    ctx=BN_CTX_new();
    a=BN_new();
    strcpy(c,"100");
    ret=BN_hex2bn(&a,c);
    b=BN_new();
    strcpy(d,"17");
    ret=BN_hex2bn(&b,d);
    out=BIO_new(BIO_s_file());
    ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
    div=BN_new();
    rem=BN_new();
    ret=BN_div(div,rem,a,b,ctx);
    if(ret!=1)
    {
        printf("err.\n");
        return -1;
    }
    BIO_puts(out,"bn 0x100 / 0x17 =0x");
    BN_print(out,div);
    BIO_puts(out,"\n");
    BIO_puts(out,"bn 0x100 % 0x17 =0x");
    BN_print(out,rem);
    BIO_puts(out,"\n");
    BN_free(a);
    BN_free(b);
    BN_free(div);
    BN_free(rem);
    BIO_free(out);
    BN_CTX_free(ctx);
    return 0;
}
```

6）示例 6

平方运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
    BIGNUM      *a,*sqr;
```

```
    BN_CTX*ctx;
    BIO     *out;
    char c[20];
    int     ret;

    ctx=BN_CTX_new();
    a=BN_new();
    strcpy(c,"100");
    ret=BN_hex2bn(&a,c);
    sqr=BN_new();
    out=BIO_new(BIO_s_file());
    ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
    ret=BN_sqr(sqr,a,ctx);
    if(ret!=1)
    {
        printf("err.\n");
        return -1;
    }
    BIO_puts(out,"bn 0x100 sqr    =0x");
    BN_print(out,sqr);
    BIO_puts(out,"\n");
    BN_free(a);
    BN_free(sqr);
    BIO_free(out);
    BN_CTX_free(ctx);
    return 0;
}
```

7）示例 7

次方运算

```
#include <openssl/bn.h>
#include <string.h>
#include <openssl/bio.h>

int   main()
{
    BIGNUM      *a,*exp,*b;
    BN_CTX*ctx;
    BIO     *out;
    char c[20],d[20];
    int     ret;

    ctx=BN_CTX_new();
    a=BN_new();
    strcpy(c,"100");
```

```
	ret=BN_hex2bn(&a,c);
	b=BN_new();
	strcpy(d,"3");
	ret=BN_hex2bn(&b,d);
	exp=BN_new();
	out=BIO_new(BIO_s_file());
	ret=BIO_set_fp(out,stdout,BIO_NOCLOSE);
	ret=BN_exp(exp,a,b,ctx);
	if(ret!=1)
	{
		printf("err.\n");
		return -1;
	}
	BIO_puts(out,"bn 0x100 exp 0x3    =0x");
	BN_print(out,exp);
	BIO_puts(out,"\n");
	BN_free(a);
	BN_free(b);
	BN_free(exp);
	BIO_free(out);
	BN_CTX_free(ctx);
	return 0;
}
```

第十二章 BASE64 编解码

12.1 BASE64 编码介绍

BASE64 编码是一种常用的将十六进制数据转换为可见字符的编码。与 ASCII 码相比，它占用的空间较小。BASE64 编码在 rfc3548 中定义。

12.2 BASE64 编解码原理

将数据编码成 BASE64 编码时，以 3 字节数据为一组，转换为 24bit 的二进制数，将 24bit 的二进制数分成四组，每组 6bit。对于每一组，得到一个数字：0-63。然后根据这个数字查表即得到结果。表如下：

Value Encoding	Value Encoding	Value Encoding	Value Encoding
0 A	17 R	34 i	51 z
1 B	18 S	35 j	52 0
2 C	19 T	36 k	53 1
3 D	20 U	37 l	54 2
4 E	21 V	38 m	55 3
5 F	22 W	39 n	56 4
6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7
9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	(pad) =
15 P	32 g	49 x	
16 Q	33 h	50 y	

比如有数据：0x30 0x82 0x02

编码过程如下：

1）得到 16 进制数据： 30 82 02

2）得到二进制数据： 00110000 10000010 00000010

3）每 6bit 分组： 001100 001000 001000 000010

4）得到数字： 12 8 8 2

5）根据查表得到结果 ： M I I C

BASE64 填充：在不够的情况下在右边加 0。

有三种情况：

1) 输入数据比特数是 24 的整数倍（输入字节为 3 字节整数倍），则无填充；

2) 输入数据最后编码的是 1 个字节(输入数据字节数除 3 余 1)，即 8 比特，则需要填

充 2 个"==",因为要补齐 6 比特，需要加 2 个 00；

3）输入数据最后编码是 2 个字节(输入数据字节数除 3 余 2)，则需要填充 1 个"="，因为补齐 6 比特，需要加一个 00。

举例如下：

对 0x30 编码：

1) 0x30 的二进制为：00110000

2) 分组为：001100 00

3) 填充 2 个 00：001100 000000

4) 得到数字：12 0

5) 查表得到的编码为 MA，另外加上两个==

所以最终编码为：MA==

base64 解码是其编码过程的逆过程。解码时，将 base64 编码根据表展开，根据有几个等号去掉结尾的几个 00，然后每 8 比特恢复即可。

12.3 主要函数

Openssl 中用于 base64 编解码的函数主要有：

1） 编码函数

- EVP_EncodeInit
  编码前初始化上下文。
- EVP_EncodeUpdate
  进行 BASE64 编码，本函数可多次调用。
- EVP_EncodeFinal
  进行 BASE64 编码，并输出结果。
- EVP_EncodeBlock
  进行 BASE64 编码。

2） 解码函数

- EVP_DecodeInit
  解码前初始化上下文。
- EVP_DecodeUpdate
  BASE64 解码，本函数可多次调用。
- EVP_DecodeFinal
  BASE64 解码，并输出结果。
- EVP_DecodeBlock
  BASE64 解码，可单独调用。

12.4 编程示例

1) 示例 1

```
#include <string.h>
#include <openssl/evp.h>
int  main()
{
```

```
    EVP_ENCODE_CTX    ectx,dctx;
    unsigned char  in[500],out[800],d[500];
    int           inl,outl,i,total,ret,total2;

    EVP_EncodeInit(&ectx);
    for(i=0;i<500;i++)
        memset(&in[i],i,1);
    inl=500;
    total=0;
    EVP_EncodeUpdate(&ectx,out,&outl,in,inl);
    total+=outl;
    EVP_EncodeFinal(&ectx,out+total,&outl);
    total+=outl;
    printf("%s\n",out);

    EVP_DecodeInit(&dctx);
    outl=500;
    total2=0;
    ret=EVP_DecodeUpdate(&dctx,d,&outl,out,total);
    if(ret<0)
    {
        printf("EVP_DecodeUpdate err!\n");
        return -1;
    }
    total2+=outl;
    ret=EVP_DecodeFinal(&dctx,d,&outl);
    total2+=outl;
    return 0;
}
```

本例中先编码再解码。

编码调用次序为 EVP_EncodeInit、EVP_EncodeUpdate(可以多次)和 EVP_EncodeFinal。

解码调用次序为 EVP_DecodeInit、EVP_DecodeUpdate(可以多次)和 EVP_DecodeFinal。

注意：采用上述函数 BASE64 编码的结果不在一行，解码所处理的数据也不在一行。用上述函数进行 BASE64 编码时，输出都是格式化输出。特别需要注意的是，BASE64 解码时如果某一行字符格式超过 80 个，会出错。如果要 BASE64 编码的结果不是格式化的，可以直接调用函数：EVP_EncodeBlock。同样对于非格式化数据的 BASE64 解码可以调用 EVP_DecodeBlock 函数，不过用户需要自己去除后面填充的 0。

2) 示例 2

```
#include <string.h>
#include <openssl/evp.h>
int  main()
{
    unsigned char  in[500],out[800],d[500],*p;
    int                    inl,i,len,pad;
```

```
    for(i=0;i<500;i++)
        memset(&in[i],i,1);
    printf("please input how much(<500) to base64 : \n");
    scanf("%d",&inl);
    len=EVP_EncodeBlock(out,in,inl);
    printf("%s\n",out);
    p=out+len-1;
    pad=0;
    for(i=0;i<4;i++)
    {
        if(*p=='=')
            pad++;
        p--;
    }
    len=EVP_DecodeBlock(d,out,len);
    len-=pad;
    if((len!=inl) || (memcmp(in,d,len)))
        printf("err!\n");
    printf("test ok.\n");
    return 0;
}
```

第十三章 ASN1 库

13.1 ASN1 简介

ASN.1(Abstract Syntax Notation One，X.208)，是一套灵活的标记语言，它允许定义多种数据类型，从 integer、bit string 一类的简单类型到结构化类型，如 set 和 sequence，并且可以使用这些类型构建复杂类型。

DER 编码是 ANS.1 定义的将对象描述数据编码成八位串值的编码规则，它给出了对 ANS.1 值（对象的类型和值）的唯一编码规则。

在 ANS.1 中，一个类型是一组值，对于某些类型，值的个数是已知的，而有些类型中值的个数是不固定的。ANS.1 中有四种类型：

1) 简单类型

BIT STRING 任意 0、1 位串；
IA5String 任意 IA5(ASCII)字符串；
INTEGER 任意一个整数；
NULL 空值；
OBJECT IDENTIFIER 一个对象标识号（一串整数），标识算法或属性类型等对象；
OCTET STRING 8 位串；
PrintableString 任意可打印字符串；
T61String 任意 T.61（8 位）字符串；
UTCTime 一个“协同世界时”或“格林威治标准时（G.M.T）”。

2) 结构类型

结构类型由组件组成，ANS.1 定义了四种结构类型：
SEQUENCE 一个或多个类型的有序排列；
SEQUENCE OF 一个给定类型的 0 个或多个有序排列；
SET 一个或多个类型的无序集合；
SET OF 一个给定类型的 0 个或多个无序集合。

3) 带标记类型

在一个应用内部区分类型的有效方法是使用标记，标记也同样用于区分一个结构类型内部不同的组件。例如 SET 或 SEQUENCE 类型可选项通常使用上下文标记以避免混淆。有两种标记类型的方法：隐式和显式。隐式标记类型是将其它类型的标记改变，得到新的类型。隐式标记的关键字是 IMPLICIT。显式标记类型是将其它类型加上一个外部标记，得到新的类型。显式标记的关键字是 EXPLICIT。
为了进行编码，隐式标记类型除了标记不同以外，可以视为与其基础类型相同。显式标记类型可以视为只有一个组件的结构类型。

4) 其它类型

类型和值用符号::=表示，符号左边的是名字，右边是类型和值。名字又可以用于定义其它的类型和值。

除了 CHOICE 类型、ANY 类型以外，所有 ANS.1 类型都有一个标记，标记由一个类和一个非负的标记码组成，当且仅当标记码相同时，ANS.1 类型是相同的。也就是说，影响其抽象意义的不是 ANS.1 类型的名字，而是其标记。

通用标记在 X.208 中定义，并给出相应的通用标记码。其它的标记类型分别在很多地方定义，可以通过隐式和显式标记获得。

下表列出了一些通用类型及其标记：

类型	标记码（十六进制）
INTEGER	02
BIT STRING	03
OCTET STRING	04
NULL	05
OBJECT IDENTIFIER	06
SEQUENCE and SEQUENCEOF	10
SET and SET OF	11
PrintableString	13
T61String	14
IA5String	16
UTCTime	17

13.2 DER 编码

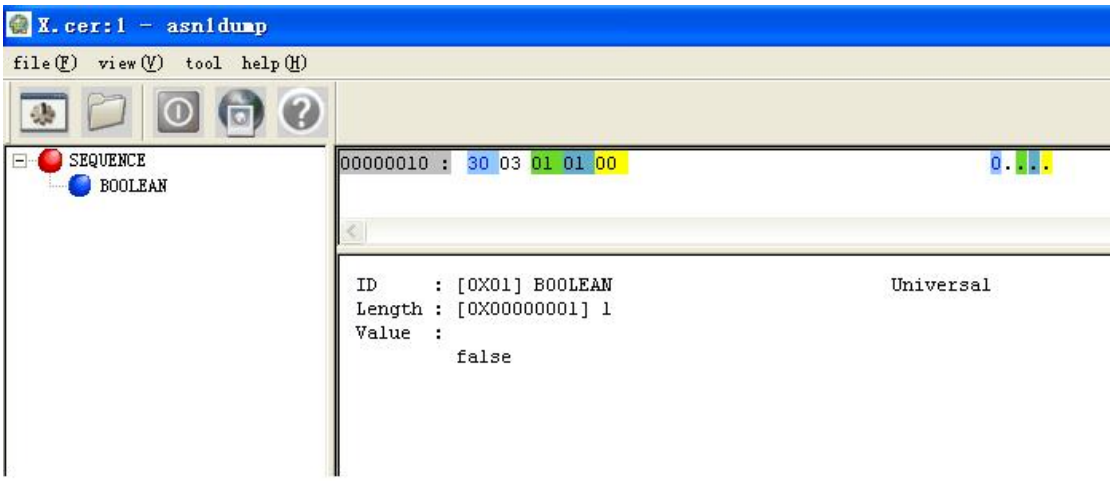
DER 给出了一种将 ASN.1 值表示为 8 位串的方法。DER 编码包含三个部分：

- 标识（一个或多个 8 位串）：定义值的类和标记码，指出是原始编码还是结构化编码。
- 长度（一个或多个 8 位串）：对于定长编码，指出内容中 8 位串的个数；对于不定长编码，指出长度是不定的。
- 内容（一个或多个 8 位串）：对于原始定长编码，给出真实值；对于结构化编码，给出各组件 BER 编码的按位串联结果。
- 内容结束（一个或多个 8 位串）：对于结构化不定长编码，标识内容结束；对于其它编码，无此项。

13.3 ASN1 基本类型示例

1） ASN1_BOOLEAN

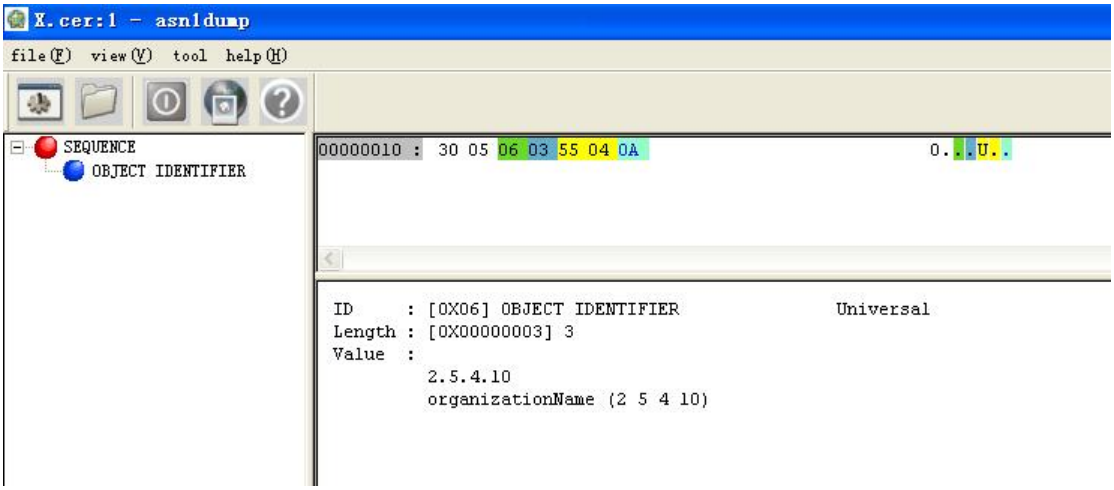
表明了ASN1语法中的true和flase。用户以用UltraEdit等工具编辑一个二进制文件来查看，此二进制文件的内容为：0x30 0x03 0x01 0x01 0x00，然后用asn1view工具查看此文件内容。显示如下：



其中0x01 (表示为BOOLEAN) 0x01(表示后面值的长度) 0x00（值）为本例BOOLEAN的DER编码。

2） ASN1_OBJECT

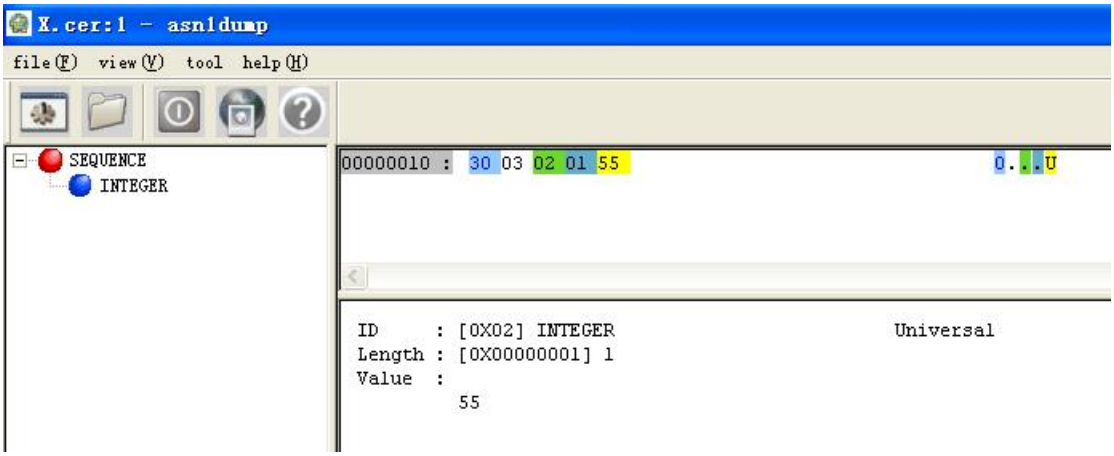
ASN1中的OBJECT表明来一个对象，每个对象有一个OID(object id)。例如：OU的OID为2.5.4.11。OBJECT对象在DER编码的时候通过计算将OID转换为另外一组数据(可用函数a2d_ASN1_OBJECTH函数)。用户编辑一个二进制文件，内容为：0x30 0x05 0x06 0x03 0x55 0x04 0x0A，用asn1view打开查看。如下：



其中0x06（表示为OBJECT类型） 0x03（值的长度） 0x55 0x04 0x0A（此三项由2.5.4.11计算而来）为此OBJECT的DER编码。

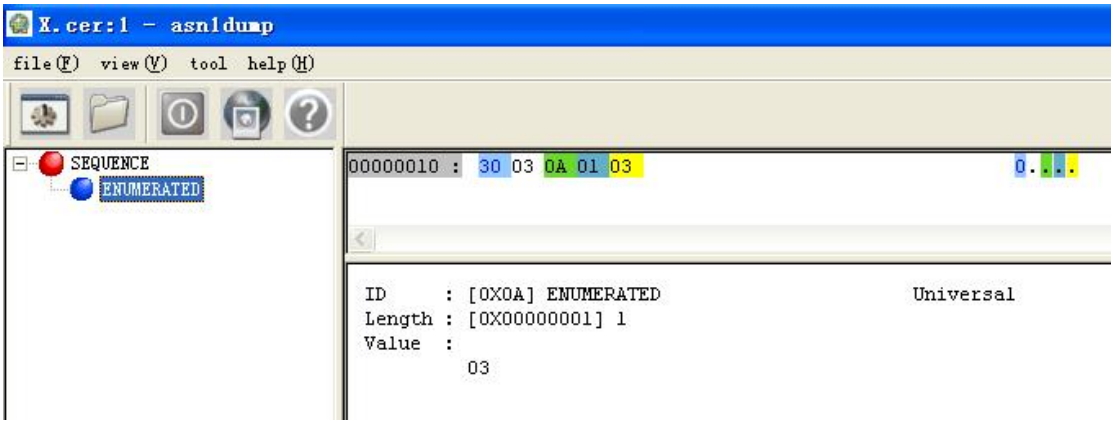
3) ASN1_INTEGER

ASN1中的INTEGER类型用于表示整数。编辑一个二进制文件，其内容为：0x30 0x03 0x02（整数） 0x01 （整数值长度）0x55 (整数值)。用an1view查看如下：



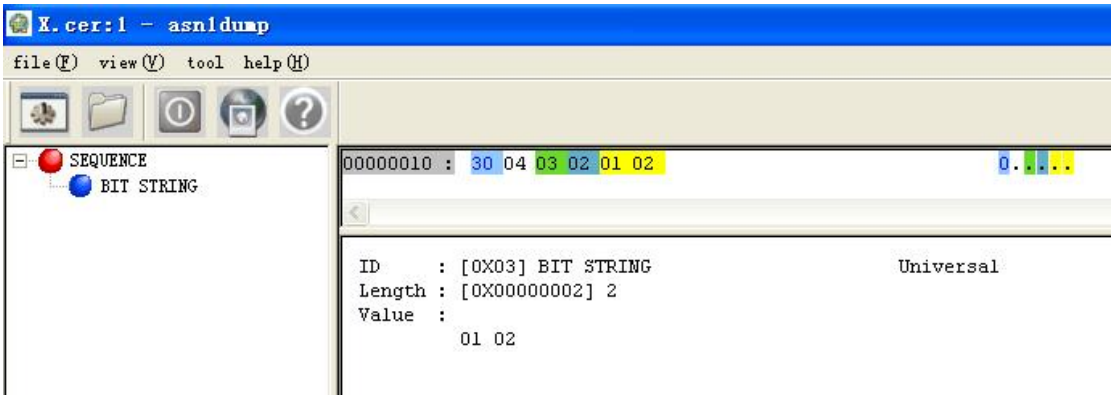
4） ASN1_ENUMERATED

ASN1枚举类型，示例如下：



5) ASN1_BIT_STRING

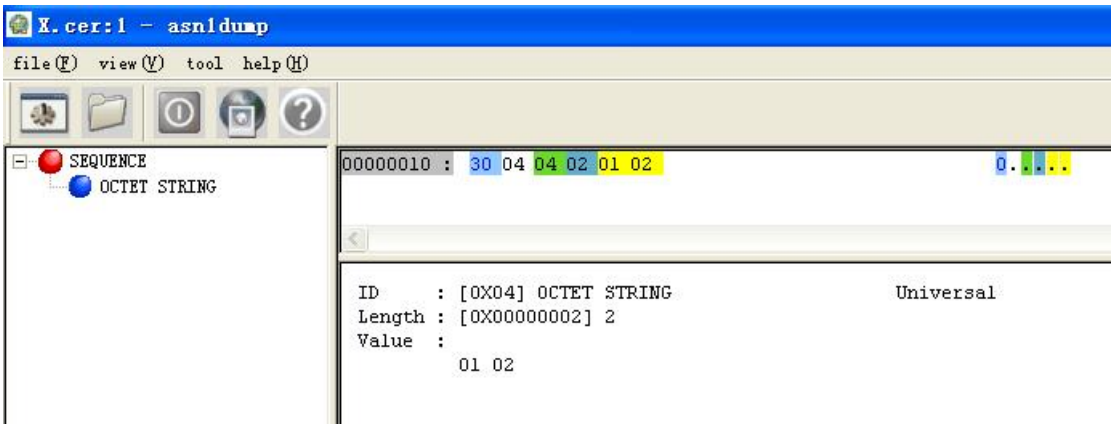
示例如下：



此图显示0x01 0x02的DER编码：0x03（BIT STRING 类型） 0x02（长度） 0x01 0x02（比特值）。

6） ASN1_OCTET_STRING

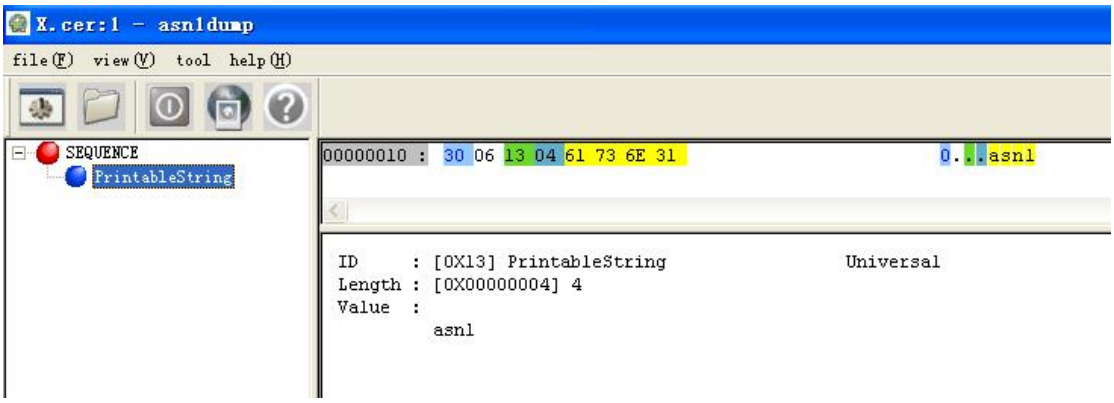
如下：



显示0x01 0x02的OCTET STRING编码：0x04(OCTET STRING) 0x02(长度) 0x01 0x02（值）。

7）ASN1_PRINTABLESTRING

可打印字符，如下：



显示来可打印字符“asn1“的DER编码，其编码值为0x13(PRINTABLESTRING) 0x04(值长度) 0x61 0x73 0x6E 0x31(值，即“asn1”)。

其他：

ASN1_UTCTIME：表示时间。

ASN1_GENERALIZEDTIME：表示时间。

ASN1_VISIBLESTRING：存放可见字符。

ASN1_UTF8STRING：用于存放utf8字符串，存放汉字需要将汉字转换为utf8字符串。

ASN1_TYPE：用于存放任意类型。

13.4 openssl 的 ASN.1 库

Openssl 的 ASN.1 库定义了 asn.1 对应的基本数据结构和大量用于 DER 编码的宏。比如整型定义如下：

typedef struct asn1_string_st ASN1_INTEGER;

另外，还用相同的数据结构 asn1_string_st 定义了：

ASN1_ENUMERATED;

ASN1_BIT_STRING;

ASN1_OCTET_STRING;

ASN1_PRINTABLESTRING;

ASN1_T61STRING;

ASN1_IA5STRING；
ASN1_GENERALSTRING；
ASN1_UNIVERSALSTRING；
ASN1_BMPSTRING；
ASN1_UTCTIME；
ASN1_TIME；
ASN1_GENERALIZEDTIME；
ASN1_VISIBLESTRING；
ASN1_UTF8STRING；
ASN1_TYPE;
这些都是定义基本数据结构的必要元素。

对于每种类型，均有四种最基本的函数：new、free、i2d 和 d2i。其中 new 函数用于生成一个新的数据结构；free 用于释放该结构； i2d 用于将该内部数据结构转换成 DER 编码；d2i 用于将 DER 编码转换成内部数据结构。另外，大部分类型都有 set 和 get 函数，用于给内部数据结构赋值和从中取值。以 ASN1_INTEGER 为例，它有如下基本函数：

```
ASN1_INTEGER ASN1_INTEGER_new(void);
void *ASN1_INTEGER_free(ASN1_INTEGER *a);
ASN1_INTEGER   *d2i_ASN1_INTEGER(ASN1_INTEGER **a,
                                 unsigned char **in,long len);
int    i2d_ASN1_INTEGER(ASN1_INTEGER *a,unsigned char **out);
long ASN1_INTEGER_get(ASN1_INTEGER *a)
int ASN1_INTEGER_set(ASN1_INTEGER *a, long v);
```

前面的四个函数由 DECLARE_ASN1_FUNCTIONS(ASN1_INTEGER)声明，并由 IMPLEMENT_ASN1_FUNCTIONS(ASN1_INTEGER)实现。

采用 ASN.1 定义的复杂的结构都是由基本的类型构造的，因此可以用这些基本的数据来实现对复杂结构的编码。

13.5 用 openssl 的 ASN.1 库 DER 编解码

当采用 Openssl 的 ASN.1 库编码一个 asn.1 定义的结构的时候，需要采用如下步骤：

1) 用 ASN.1 语法定义内部数据结构，并声明函数；

所谓内部数据结构，指的是 Openssl 中用基本的数据类型按照 ASN.1 语法定义的其他的数据结构，这种数据结构可以方便的用于编解码。

以 x509v4 中的证书有效期为例，证书有效期定义如下：

```
AttCertValidityPeriod   ::= SEQUENCE
{
notBeforeTime    GeneralizedTime,
notAfterTime     GeneralizedTime
}
```

所以我们可以定义相应的内部数据结构，如下：

```
typedef   struct      X509V4_VALID_st
{
ASN1_GENERALIZEDTIME *notBefore;
```

```
ASN1_GENERALIZEDTIME *notAfter;
}X509V4_VALID;
DECLARE_ASN1_FUNCTIONS(X509V4_VALID)
```

其中最后一行用于定义四个函数：

```
X509V4_VALID *X509V4_VALID_new(void);
void *X509V4_VALID_free(X509V4_VALID *a);
X509V4_VALID *d2i_ASN1_INTEGER(X509V4_VALID **a,unsigned char **in,long len);
int i2d_ X509V4_VALID (X509V4_VALID *a,unsigned char **out);
```

2) 实现内部数据结构的四个基本函数

实现内部数据结构的基本函数，是通过一系列的宏来实现的。定义的模式如下，以属性证书有效期为例，如下：

```
/* X509V4_VALID */
ASN1_SEQUENCE(X509V4_VALID) =
{
ASN1_SIMPLE(X509V4_VALID, notBefore, ASN1_GENERALIZEDTIME),
ASN1_SIMPLE(X509V4_VALID, notAfter, ASN1_GENERALIZEDTIME)
} ASN1_SEQUENCE_END(X509V4_VALID)
IMPLEMENT_ASN1_FUNCTIONS(X509V4_VALID)
```

这样通过宏就实现了一个 asn.1 定义结构的最基本的四个函数。

本例有五个宏，采用什么样的宏，与数据结构的 asn.1 定义相关。

13.6 Openssl 的 ASN.1 宏

Openssl 中的 ASN.1 宏用来定义某种内部数据结构以及这种结构如何编码，部分宏定义说明如下：

1) DECLARE_ASN1_FUNCTIONS
   用于声明一个内部数据结构的四个基本函数，一般可以在头文件中定义。
2) IMPLEMENT_ASN1_FUNCTIONS
   用于实现一个数据结构的四个基本函数。
3） ASN1_SEQUENCE
   用于 SEQUENCE，表明下面的编码是一个 SEQUENCE。
4) ASN1_CHOICE
   表明下面的编码是选择其中一项，为 CHOICE 类型。
5） ASN1_SIMPLE
   用于简单类型或结构类型，并且是必须项。
6) ASN1_OPT
   用于可选项，表明 asn.1 语法中，本项是可选的。
7) ASN1_EXP_OPT
   用于显示标记，表明 asn.1 语法中，本项是显示类型，并且是可选的；
8) ASN1_EXP
   用于显示标记，表明 asn.1 语法中，本项是显示标记。
9） ASN1_IMP_SEQUENCE_OF_OPT
   用于隐示标记，表明 asn.1 语法中，本项是一个 SEQUENCE 序列，为隐

示类型，并且是可选的。

10) ASN1_IMP_OPT
用于隐示标记，表明 asn.1 语法中，本项是隐示类型，并且是可选的。

11) ASN1_IMP
用于隐示标记，表明 asn.1 语法中，本项是隐示类型。

12) ASN1_SEQUENCE_END
用于 SEQUENCE 结束。

13) ASN1_CHOICE_END
用于结束 CHOICE 类型。

13.7 ASN1 常用函数

ASN1 的基本的数据类型一般都有如下函数：new、free、i2d、d2i、i2a、a2i、print、set、get、cmp 和 dup。其中 new、free、i2d、d2i 函数通过宏定义实现。new 函数用于分配空间，生成 ASN1 数据结构;free 用于释放空间；i2d 函数将 ASN1 数据结构转换为 DER 编码；d2i 将 DER 编码转换为 ASN1 数据结构，i2a 将内部结构转换为 ASCII 码，a2i 将 ASCII 码转换为内部数据结构。set 函数用于设置 ASN1 类型的值，get 函数用于获取 ASN1 类型值；print 将 ASN1 类型打印；cmp 用于比较 ASN1 数据结构；dup 函数进行数据结构的拷贝。

常用的函数有：

1） int a2d_ASN1_OBJECT(unsigned char *out, int olen, const char *buf, int num)

计算 OID 的 DER 编码，比如将 2.99999.3 形式转换为内存形式。示例：

```
#include <openssl/asn1.h>
void main()
{
    const          char      oid[]={"2.99999.3"};
    int                          i;
    unsigned char      *buf;

    i=a2d_ASN1_OBJECT(NULL,0,oid,-1);
    if (i <= 0)
        return;
    buf=(unsigned char *)malloc(sizeof(unsigned char)*i);
    i=a2d_ASN1_OBJECT(buf,i,oid,-1);
    free(buf);
    return;
}
```

输出结果：buf 内存值为：86 8D 6F 03

2) int a2i_ASN1_INTEGER(BIO *bp,ASN1_INTEGER *bs,char *buf,int size)

将 bp 中的 ASC 码转换为 ASN1_INTEGER,buf 存放 BIO 中的 ASC 码。示例如下：

```
#include <openssl/asn1.h>
int   main()
{
    BIO                *bp;
    ASN1_INTEGER   *i;
```

```
    unsigned char  buf[50];
    int                 size,len;

    bp=BIO_new(BIO_s_mem());
    len=BIO_write(bp,"0FAB08BBDDEECC",14);
    size=50;
    i=ASN1_INTEGER_new();
    a2i_ASN1_INTEGER(bp,i,buf,size);
    BIO_free(bp);
    ASN1_INTEGER_free(i);
    return 0;
}
```

3）int a2i_ASN1_STRING(BIO *bp,ASN1_STRING *bs,char *buf,int size)

将 ASCII 码转换为 ASN1_STRING，示例：

```
#include <openssl/asn1.h>
int   main()
{
    BIO                 *bp;
    ASN1_STRING         *str;
    unsigned char  buf[50];
    int                 size,len;

    bp=BIO_new(BIO_s_mem());
    len=BIO_write(bp,"B2E2CAD4",8);
    size=50;
    str=ASN1_STRING_new();
    a2i_ASN1_STRING(bp,str,buf,size);
    BIO_free(bp);
    ASN1_STRING_free(str);
    return 0;
}
```

转换后 str->data 的前四个字节即变成"测试"。

4）unsigned char *asc2uni(const char *asc, int asclen, unsigned char **uni, int *unilen)

将 ASCII 码转换为 unicode，示例：

```
#include <stdio.h>
#include <openssl/crypto.h>
int   main()
{
    unsigned char  asc[50]={"B2E2CAD4"};
    unsigned char  uni[50],*p,*q;
    int                 ascLen,unlen;

    ascLen=strlen(asc);
    q=asc2uni(asc,ascLen,NULL,&unlen);
```

```
    OPENSSL_free(q);
    return 0;
}
```

5）int ASN1_BIT_STRING_get_bit(ASN1_BIT_STRING *a, int n)

本函数根据 n 获取其比特位上的值，示例：

```
#include <openssl/asn1.h>
int     main()
{
    int     ret,i,n;
    ASN1_BIT_STRING *a;

    a=ASN1_BIT_STRING_new();
    ASN1_BIT_STRING_set(a,"ab",2);
    for(i=0;i<2*8;i++)
    {
        ret=ASN1_BIT_STRING_get_bit(a,i);
        printf("%d",ret);
    }
    ASN1_BIT_STRING_free(a);
    return 0;
}
```

程序输出：0110000101100010

说明：a 中”ab”的二进制既是 0110000101100010。

6）ASN1_BIT_STRING_set

设置 ASN1_BIT_STRING 的值，它调用了 ASN1_STRING_set 函数；

7）void *ASN1_d2i_bio(void *(*xnew)(void), d2i_of_void *d2i, BIO *in, void **x)

对 bio 的数据 DER 解码，xnew 无意义，d2i 为 DER 解码函数，in 为 bio 数据，x 为数据类型，返回值为解码后的结果。如果 x 分配了内存，x 所指向的地址与返回值一致。示例如下：

```
#include <stdio.h>
#include <openssl/asn1.h>
#include <openssl/x509v3.h>
#include <openssl/bio.h>
int   main()
{
    BIO     *in;
    X509    **out=NULL,*x;

    in=BIO_new_file("a.cer","r");
    out=(X509 **)malloc(sizeof(X509 *));
    *out=NULL;
    x=ASN1_d2i_bio(NULL,(d2i_of_void *)d2i_X509,in,out);
    X509_free(x);
    free(out);
```

```
    return 0;
}
```

8）void *ASN1_d2i_fp(void *(*xnew)(void), d2i_of_void *d2i, FILE *in, void **x)

将 in 指向的文件进行 DER 解码，其内部调用了 ASN1_d2i_bi 函数，用法与 ASN1_d2i_bi 类似。

9）int ASN1_digest(i2d_of_void *i2d, const EVP_MD *type,
char *data,unsigned char *md, unsigned int *len)

ASN1 数据类型签名。将 data 指针指向的 ASN1 数据类型用 i2d 函数进行 DER 编码，然后用 type 指定的摘要方法进行计算，结果存放在 md 中，结果的长度由 len 表示。

10）int ASN1_i2d_bio(i2d_of_void *i2d, BIO *out, unsigned char *x)

将 ASN1 数据结构 DER 编码，并将结果写入 bio。示例如下：

```
#include <openssl/asn1.h>
#include <openssl/bio.h>
int        main()
{
    int                    ret;
    BIO                    *out;
    ASN1_INTEGER       *a;

    out=BIO_new_file("int.cer","w");
    a=ASN1_INTEGER_new();
    ASN1_INTEGER_set(a,(long)100);
    ret=ASN1_i2d_bio(i2d_ASN1_INTEGER,out,a);
    BIO_free(out);
    return 0;
}
```

本程序将 ASN1_INTEGER 类型装换为 DER 编码并写入文件。int.cer 的内容如下：

02 01 64 （十六进制）。

11）int ASN1_i2d_fp(i2d_of_void *i2d, FILE *out, void *x)

将 ASN1 数据结构 DER 编码并写入 FILE，此函数调用了 ASN1_i2d_bio。

12）void *ASN1_dup(i2d_of_void *i2d, d2i_of_void *d2i, char *x)

ASN1 数据复制。x 为 ASN1 内部数据结构，本函数先将 x 通过 i2d 将它变成 DER 编码，然后用 d2i 再 DER 解码，并返回解码结果。

13) ASN1_ENUMERATED_set

设置 ASN1_ENUMERATED 的值。

14) ASN1_ENUMERATED_get

获取 ASN1_ENUMERATED 的值；示例如下：

```
clude <openssl/asn1.h>
int   main()
{
    long                   ret;
    ASN1_ENUMERATED       *a;

    a=ASN1_ENUMERATED_new();
```

```
    ASN1_ENUMERATED_set(a,(long)155);
    ret=ASN1_ENUMERATED_get(a);
    printf("%ld\n",ret);
    return 0;
}
```

15）BIGNUM *ASN1_ENUMERATED_to_BN(ASN1_ENUMERATED *ai, BIGNUM *bn)

将 ASN1_ENUMERATED 类型转换为 BN 大数类型。此函数调用 BN_bin2bn 函数获取 bn，如果 ai->type 表明它是负数，再调用 BN_set_negative 设置 bn 成负数。示例如下：

```
#include <openssl/asn1.h>
int   main()
{
    long                ret;
    ASN1_ENUMERATED     *a;
    BIGNUM              *bn;

    a=ASN1_ENUMERATED_new();
    ASN1_ENUMERATED_set(a,(long)155);
    ret=ASN1_ENUMERATED_get(a);
    bn=BN_new();
    bn=ASN1_ENUMERATED_to_BN(a,bn);
    BN_free(bn);
    ASN1_ENUMERATED_free(a);
    return 0;
}
```

如果 ASN1_ENUMERATED_to_BN 的第二个参数为 NULL,bn 将在内部分配空间。

16）int ASN1_GENERALIZEDTIME_check(ASN1_GENERALIZEDTIME *a)

检查输入参数是不是合法的 ASN1_GENERALIZEDTIME 类型。

17）int ASN1_parse_dump(BIO *bp, const unsigned char *pp, long len, int indent, int dump)

本函数用于将 pp 和 len 指明的 DER 编码值写在 BIO 中，其中 indent 和 dump 用于设置打印的格式。indent 用来设置打印出来当列之间空格个数，ident 越小，打印内容越紧凑。dump 表明当 asn1 单元为 BIT STRING 或 OCTET STRING 时，打印内容的字节数。示例如下：

```
#include <openssl/bio.h>
#include <openssl/asn1.h>
int   main()
{
    int         ret,len,indent,dump;
    BIO         *bp;
    char *pp,buf[5000];
    FILE        *fp;
    bp=BIO_new(BIO_s_file());
    BIO_set_fp(bp,stdout,BIO_NOCLOSE);
    fp=fopen("der.cer","rb");
```

```
    len=fread(buf,1,5000,fp);
    fclose(fp);
    pp=buf;
    indent=7;
    dump=11;
    ret=ASN1_parse_dump(bp,pp,len,indent,dump);
    BIO_free(bp);
    return 0;
}
```

其中 der.cer 为一个 DER 编码的文件，比如一个数字证书。

18）int ASN1_sign(i2d_of_void *i2d, X509_ALGOR *algor1, X509_ALGOR *algor2, ASN1_BIT_STRING *signature, char *data, EVP_PKEY *pkey, const EVP_MD *type)

对 ASN1 数据类型签名。i2d 为 ASN1 数据的 DER 方法，signature 用于存放签名结果，data 为 ASN1 数据指针，pkey 指明签名密钥，type 为摘要算法，algor1 和 algor2 无用，可全为 NULL。签名时，先将 ASN1 数据 DER 编码，然后摘要，最后签名运算。

在 x509.h 中有很多 ASN1 数据类型的签名都通过此函数来定义，有 X509_sign、X509_REQ_sign、X509_CRL_sign、NETSCAPE_SPKI_sign 等。示例如下：

```
#include <openssl/asn1.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>
int   main()
{
    int                 ret;
    ASN1_INTEGER  *a;
    EVP_MD              *md;
    EVP_PKEY        *pkey;
    char            *data;
    ASN1_BIT_STRING    *signature=NULL;
    RSA                 *r;
    int                 i,bits=1024;
    unsigned long e=RSA_3;
    BIGNUM              *bne;

    bne=BN_new();
    ret=BN_set_word(bne,e);
    r=RSA_new();
    ret=RSA_generate_key_ex(r,bits,bne,NULL);
    if(ret!=1)
    {
        printf("RSA_generate_key_ex err!\n");
        return -1;
    }
    pkey=EVP_PKEY_new();
    EVP_PKEY_assign_RSA(pkey,r);
```

```
    a=ASN1_INTEGER_new();
    ASN1_INTEGER_set(a,100);
    md=EVP_md5();
    data=(char *)a;
    signature=ASN1_BIT_STRING_new();
    ret=ASN1_sign(i2d_ASN1_INTEGER,NULL,NULL,signature,data,pkey,md);
    printf("signature len : %d\n",ret);
    EVP_PKEY_free(pkey);
    ASN1_INTEGER_free(a);
    free(signature);
    return 0;
}
```

}本例将 ASN1_INTEGER 整数签名。

19）ASN1_STRING *ASN1_STRING_dup(ASN1_STRING *str)

ASN1_STRING 类型拷贝。内部申请空间，需要用户调用 ASN1_STRING_free 释放该空间。

20）int ASN1_STRING_cmp(ASN1_STRING *a, ASN1_STRING *b)

ASN1_STRING 比较。ossl_typ.h 中绝大多数 ASN1 基本类型都定义为 ASN1_STRING，所以，此函数比较通用。示例如下：

```
#include <openssl/asn1.h>
int    main()
{
    int    ret;
    ASN1_STRING     *a,*b,*c;
    a=ASN1_STRING_new();
    b=ASN1_STRING_new();
    ASN1_STRING_set(a,"abc",3);
    ASN1_STRING_set(b,"def",3);
    ret=ASN1_STRING_cmp(a,b);
    printf("%d\n",ret);
    c=ASN1_STRING_dup(a);
    ret=ASN1_STRING_cmp(a,c);
    printf("%d\n",ret);
    ASN1_STRING_free(a);
    ASN1_STRING_free(b);
    ASN1_STRING_free(c);
    return 0;
}
```

21）unsigned char * ASN1_STRING_data(ASN1_STRING *x)

获取 ASN1_STRING 数据存放地址，即 ASN1_STRING 数据结构中 data 地址。本函数由宏实现。

22）int ASN1_STRING_set(ASN1_STRING *str, const void *_data, int len)

设置 ASN1 字符串类型的值。str 为 ASN1_STRING 地址，_data 为设置值的首地址，len 为被设置值的长度。示例如下：

```
    ASN1_STRING     *str=NULL;
```

str=ASN1_STRING_new();

ASN1_STRING_set(str,”abc”,3);

此示例生成的 ASN1_STRING 类型为 OCTET_STRING。其他的 ASN1_STRING 类型也能用此函数设置，如下：

```
ASN1_PRINTABLESTRING      *str=NULL;
str=ASN1_PRINTABLESTRING_new();
ASN1_STRING_set(str,”abc”,3);
```

23）ASN1_STRING_TABLE *ASN1_STRING_TABLE_get(int nid)

根据 nid 来查找 ASN1_STRING_TABLE 表。此函数先查找标准表 tbl_standard，再查找扩展表 stable。ASN1_STRING_TABLE 数据结构在 asn1.h 中定义，它用于约束 ASN1_STRING_set_by_NID 函数生成的 ASN1_STRING 类型。

```
typedef struct asn1_string_table_st {
    int nid;
    long minsize;
    long maxsize;
    unsigned long mask;
    unsigned long flags;
} ASN1_STRING_TABLE;
```

其中 nid 表示对象 id，minsize 表示此 nid 值的最小长度，maxsize 表示此 nid 值的最大长度，mask 为此 nid 可以采用的 ASN1_STRING 类型：B_ASN1_BMPSTRING、B_ASN1_UTF8STRING、B_ASN1_T61STRING 和 B_ASN1_UTF8STRING，flags 用于标记是否为扩展或是否已有 mask。

24）ASN1_STRING *ASN1_STRING_set_by_NID(ASN1_STRING **out, const unsigned char *in, int inlen, int inform, int nid)

根据 nid 和输入值获取对应的 ASN1_STIRNG 类型。out 为输出，in 为输入数据，inlen 为其长度，inform 为输入数据的类型，可以的值有：MBSTRING_BMP、MBSTRING_UNIV、MBSTRING_UTF8、MBSTRING_ASC，nid 为数字证书中常用的 nid，在 a_strnid.c 中由全局变量 tbl_standard 定义，可以的值有：NID_commonName、NID_countryName 、NID_localityName 、NID_stateOrProvinceName 、NID_organizationName 、NID_organizationalUnitName 、NID_pkcs9_emailAddress 、NID_pkcs9_unstructuredName 、NID_pkcs9_challengePassword 、NID_pkcs9_unstructuredAddress 、NID_givenName 、NID_surname 、NID_initials 、NID_serialNumber 、NID_friendlyName 、NID_name 、NID_dnQualifier 、NID_domainComponent 和 NID_ms_csp_name。生成的 ASN1_STRING 类型可以为：ASN1_T61STRING 、ASN1_IA5STRING 、ASN1_PRINTABLESTRING 、ASN1_BMPSTRING、ASN1_UNIVERSALSTRING 和 ASN1_UTF8STRING。

示例 1：

```
#include <stdio.h>
#include <openssl/asn1.h>
#include <openssl/obj_mac.h>
int   main()
{
    int         inlen,nid,inform,len;
    char in[100],out[100],*p;
```

```
    ASN1_STRING    *a;
    FILE      *fp;

    /* 汉字“赵”的UTF8值,可以用UltraEdit获取*/
    memset(&in[0],0xEF,1);
    memset(&in[1],0xBB,1);
    memset(&in[2],0xBF,1);
    memset(&in[3],0xE8,1);
    memset(&in[4],0xB5,1);
    memset(&in[5],0xB5,1);
    inlen=6;
    inform=MBSTRING_UTF8;
    nid=NID_commonName;
    /* 如果调用下面两个函数，生成的ASN1_STRING类型将是ASN1_UTF8而不是
ASN1_BMPSTRING    */
    ASN1_STRING_set_default_mask(B_ASN1_UTF8STRING);
    ret=ASN1_STRING_set_default_mask_asc("utf8only");
    if(ret!=1)
    {
        printf("ASN1_STRING_set_default_mask_asc err.\n");
        return 0;
    }
    a=ASN1_STRING_set_by_NID(NULL,in,inlen,inform,nid);
    p=out;
    len=i2d_ASN1_BMPSTRING(a,&p);
    fp=fopen("a.cer","w");
    fwrite(out,1,len,fp);
    fclose(fp);
    ASN1_STRING_free(a);
    return 0;
}
```

本例根据 UTF8 编码的汉字获取 nid 为 NID_commonName 的 ASN1_STRING 类型，其结果是一个 ASN1_BMPSTRING 类型。

示例 2：

```
#include <stdio.h>
#include <openssl/asn1.h>
#include <openssl/obj_mac.h>
int   main()
{
    int         inlen,nid,inform,len;
    char in[100],out[100],*p;
    ASN1_STRING    *a;
    FILE      *fp;
```

```
    strcpy(in,"ab");
    inlen=2;
    inform=MBSTRING_ASC;
    nid=NID_commonName;
    /* 设置生成的ASN1_STRING类型 */
    ASN1_STRING_set_default_mask(B_ASN1_UTF8STRING);
    a=ASN1_STRING_set_by_NID(NULL,in,inlen,inform,nid);
    switch(a->type)
    {
    case V_ASN1_T61STRING:
        printf("V_ASN1_T61STRING\n");
        break;
    case V_ASN1_IA5STRING:
        printf("V_ASN1_IA5STRING\n");
        break;
    case V_ASN1_PRINTABLESTRING:
        printf("V_ASN1_PRINTABLESTRING\n");
        break;
    case V_ASN1_BMPSTRING:
        printf("V_ASN1_BMPSTRING\n");
        break;
    case V_ASN1_UNIVERSALSTRING:
        printf("V_ASN1_UNIVERSALSTRING\n");
        break;
    case V_ASN1_UTF8STRING:
        printf("V_ASN1_UTF8STRING\n");
        break;
    default:
        printf("err");
        break;
    }
    p=out;
    len=i2d_ASN1_bytes(a,&p,a->type,V_ASN1_UNIVERSAL);
    fp=fopen("a.cer","w");
    fwrite(out,1,len,fp);
    fclose(fp);
    ASN1_STRING_free(a);
    getchar();
    return 0;
}
```

25）void ASN1_STRING_set_default_mask(unsigned long mask)

设置 ASN1_STRING_set_by_NID 函数返回的 ASN1_STRING 类型。mask 可以取如下值： B_ASN1_BMPSTRING、B_ASN1_UTF8STRING、B_ASN1_T61STRING 和 B_ASN1_UTF8STRING。

26）int ASN1_STRING_set_default_mask_asc(char *p)

设置 ASN1_STRING_set_by_NID 函数返回的 ASN1_STRING 类型。字符串 p 可以的值有：nombstr、pkix、utf8only 和 default，如果设置为 default，则相当于没有调用本函数。

27）int ASN1_STRING_TABLE_add(int nid, long minsize, long maxsize, unsigned long mask, unsigned long flags)

添加扩展的 ASN1_STRING_TABLE 项。说明：a_strnid.c 中定义了基本的 ASN1_STRING_TABLE 项，如果用户要添加新的 ASN1_STRING_TABLE 项，需要调此次函数。Openssl 源代码中有好几处都有这种用法，Openssl 定义标准的某种表，并且提供扩展函数供用户去扩充。

示例：ASN1_STRING_TABLE_add（NID_yourNID,1,100, DIRSTRING_TYPE,0）。

28）void ASN1_STRING_TABLE_cleanup(void)

清除用户自建的扩展 ASN1_STRING_TABLE 表。

29）int i2a_ASN1_INTEGER(BIO *bp, ASN1_INTEGER *a)

将整数转换成为 ASCII 码,放在 BIO 中。示例如下：

```
#include <openssl/asn1.h>
int   main()
{
      ASN1_INTEGER   *i;
      long              v;
      BIO                    *bp;

      printf("输入 v 的值:\n");
      scanf("%ld",&v);
      i=ASN1_INTEGER_new();
      ASN1_INTEGER_set(i,v);
      bp=BIO_new(BIO_s_file());
      BIO_set_fp(bp,stdout,BIO_NOCLOSE);
      i2a_ASN1_INTEGER(bp,i);
      BIO_free(bp);
      ASN1_INTEGER_free(i);
      printf("\n");
      return 0;
}
```

输出:

```
输入 v 的值:
15
0F
```

30）int i2a_ASN1_STRING(BIO *bp, ASN1_STRING *a, int type)

type 不起作用，将 ASN1_STRING 转换为 ASCII 码.。示例如下：

```
#include <openssl/asn1.h>
#include <openssl/asn1t.h>
int   main()
{
```

```
    ASN1_STRING         *a;
    BIO                 *bp;

    a=ASN1_STRING_new();
    ASN1_STRING_set(a,"测试",4);
    bp=BIO_new(BIO_s_file());
    BIO_set_fp(bp,stdout,BIO_NOCLOSE);
    i2a_ASN1_STRING(bp,a,1);
    BIO_free(bp);
    ASN1_STRING_free(a);
    printf("\n");
    return 0;
}
```

输出结果:

```
B2E2CAD4
```

31）OBJ_bsearch

用于从排序好的数据结构地址数组中用二分法查找数据。示例如下：

```
#include <openssl/objects.h>
typedef  struct    Student_st
{
    int        age;
}Student;
int  cmp_func(const void *a,const void *b)
{
    Student *x,*y;
    x=*(Student **)a;
    y=*(Student **)b;
    return x->age-y->age;
}
int  main()
{
    int                 ret,num,size;
    ASN1_OBJECT         *obj=NULL;
    char            **addr,*p;
    Student             a[6],**sort,**x;

    a[0].age=3;
    a[1].age=56;
    a[2].age=5;
    a[3].age=1;
    a[4].age=3;
    a[5].age=6;
    sort=(Student **)malloc(6*sizeof(Student *));
    sort[0]=&a[0];
```

```
    sort[1]=&a[1];
    sort[2]=&a[2];
    sort[3]=&a[3];
    sort[4]=&a[4];
    sort[5]=&a[5];

    qsort(sort,6,sizeof(Student *),cmp_func);
    obj=OBJ_nid2obj(NID_rsa);
    ret=OBJ_add_object(obj);
    if(ret==NID_undef)
    {
        printf("err");
    }
    else
    {
        printf("ok\n");
    }
    p=&a[4];
    addr=OBJ_bsearch(&p,(char *)sort,6,sizeof(Student *),cmp_func);
    x=(Student **)addr;
    printf("%d == %d\n",a[4].age,(*x)->age);
    return 0;
}
```

32）OBJ_create

根据 oid 以及名称信息生成一个内部的 object，示例：

nid=OBJ_create("1.2.3.44","testSn","testLn")。

33）OBJ_NAME_add

OBJ_NAME_cleanup

OBJ_NAME_get

OBJ_NAME_init

OBJ_NAME_remove

OBJ_NAME_new_index

OBJ_NAME_do_all

OBJ_NAME_do_all_sorted

OBJ_NAME 函数用于根据名字获取对称算法或者摘要算法，主要涉及到函数有:

```
int EVP_add_cipher(const EVP_CIPHER *c);
int EVP_add_digest(const EVP_MD *md);
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
const EVP_MD *EVP_get_digestbyname(const char *name);
void EVP_cleanup(void);
```

这些函数在 evp/names.c 中实现，他们调用了 OBJ_NAME 函数。

EVP_add_cipher 和 EVP_add_digest 函数调用 OBJ_NAME_init 和 OBJ_NAME_add 函数，将 EVP_CIPHER 和 EVP_MD 信息放入哈希表，EVP_get_cipherbyname 和 EVP_get_digestbyname 函数调用 OBJ_NAME_get 函数从哈希表中查询需要的信息，

EVP_cleanup 函数清除存放到 EVP_CIPHER 和 EVP_MD 信息。另外，程序可以通过调用 OpenSSL_add_all_ciphers 和 OpenSSL_add_all_digests 函数将所有的对称算法和摘要算法放入哈希表。

34）int　　OBJ_new_nid(int num)

此函数将内部的 new_nid 加 num，返回原 nid。

35）const char *OBJ_nid2ln(int n)

根据 nide 得到对象的描诉。

36）OBJ_nid2obj

根据 nid 得到对象。

37）const char *OBJ_nid2sn(int n)

根据 nid 得到对象的 sn(简称)。

38）int OBJ_obj2nid(const ASN1_OBJECT *a)

根据对象获取其 nid；

39）OBJ_obj2txt

根据对象获取对象说明或者 nid，示例：

```
#include <openssl/asn1.h>
int   main()
{
      char buf[100];
      int        buf_len=100;
      ASN1_OBJECT *a;

      a=OBJ_nid2obj(65);
      OBJ_obj2txt(buf,buf_len,a,0);
      printf("%s\n",buf);
      OBJ_obj2txt(buf,buf_len,a,1);
      printf("%s\n",buf);
      return 0;
}
```

输出结果：

```
sha1WithRSAEncryption
1.2.840.113549.1.1.5
```

40）int OBJ_sn2nid(const char *s)

根据对象别名称获取 nid

41）OBJ_txt2nid

根据 sn 或者 ln 获取对象的 nid。

42）OBJ_txt2obj

根据 sn 或者 ln 得到对象。

13.8　属性证书编码

对属性证书（x509v4）编码

以下是采用 Openssl 的 asn.1 库对属性证书编/解码的源代码：

```
/* x509v4.h */
```

```
/* valid time */
typedef    struct      X509V4_VALID_st
{
      ASN1_GENERALIZEDTIME      *notBefore;
      ASN1_GENERALIZEDTIME      *notAfter;
}X509V4_VALID;
DECLARE_ASN1_FUNCTIONS(X509V4_VALID)

/* issuer */
typedef    struct      ISSUERSERIAL_st
{
      GENERAL_NAMES     *issuer;
      ASN1_INTEGER   *subjectSN;
      ASN1_BIT_STRING     *issuerUID;
}ISSUERSERIAL;
DECLARE_ASN1_FUNCTIONS(ISSUERSERIAL)

/* objdigest */
typedef    struct      OBJDIGEST_st
{
      ASN1_ENUMERATED *digestType;
      ASN1_OBJECT           *otherType;
      X509_ALGOR            *digestAlg;
      ASN1_BIT_STRING     *digestBit;
}OBJDIGEST;
DECLARE_ASN1_FUNCTIONS(OBJDIGEST)

/* holder */
typedef    struct      ACHOLDER_st
{
      ISSUERSERIAL *baseCertificateID;
      GENERAL_NAMES     *entityName;
      OBJDIGEST         *objDigest;
}ACHOLDER;
DECLARE_ASN1_FUNCTIONS(ACHOLDER)

/* version 2 form */
typedef    struct      V2FORM_st
{
      GENERAL_NAMES     *entityName;
      ISSUERSERIAL *baseCertificateID;
      OBJDIGEST         *objDigest;
}V2FORM;
DECLARE_ASN1_FUNCTIONS(V2FORM)
```

```
typedef struct ACISSUER_st
{
	int type;
	union
	{
		V2FORM		*v2Form;
	}form;
} ACISSUER;
DECLARE_ASN1_FUNCTIONS(ACISSUER)

/* X509V4_CINF */
typedef	struct	X509V4_CINF_st
{
ASN1_INTEGER			*version;
ACHOLDER			*holder;
ACISSUER			*issuer;
X509_ALGOR			  *signature;
ASN1_INTEGER		*serialNumber;
X509V4_VALID			*valid;
STACK_OF(X509_ATTRIBUTE) *attributes;
ASN1_BIT_STRING			*issuerUID;
STACK_OF(X509_EXTENSION *extensions;
}X509V4_CINF;
DECLARE_ASN1_FUNCTIONS(X509V4_CINF)

/* x509v4 */
typedef	struct	X509V4_st
{
	X509V4_CINF		*cert_info;
	X509_ALGOR		*sig_alg;
	ASN1_BIT_STRING	*signature;
}X509V4;
DECLARE_ASN1_FUNCTIONS(X509V4)

/* x509v4.c */
/* ACISSUER */
ASN1_CHOICE(ACISSUER) = {
	ASN1_IMP(ACISSUER, form.v2Form, V2FORM,0)
} ASN1_CHOICE_END(ACISSUER)
IMPLEMENT_ASN1_FUNCTIONS(ACISSUER)

/* ACHOLDER */
ASN1_SEQUENCE(ACHOLDER) = {
```

```
        ASN1_IMP_OPT(ACHOLDER, baseCertificateID, ISSUERSERIAL,0),
        ASN1_IMP_SEQUENCE_OF_OPT(ACHOLDER, entityName,
GENERAL_NAME,1),
        ASN1_IMP_OPT(ACHOLDER, objDigest, OBJDIGEST,2)
    } ASN1_SEQUENCE_END(ACHOLDER)
    IMPLEMENT_ASN1_FUNCTIONS(ACHOLDER)

    /* V2FORM */
    ASN1_SEQUENCE(V2FORM) = {
        ASN1_SEQUENCE_OF_OPT(V2FORM, entityName, GENERAL_NAME),
        ASN1_IMP_OPT(V2FORM, baseCertificateID, ISSUERSERIAL,0),
        ASN1_IMP_OPT(V2FORM, objDigest, OBJDIGEST,1)
    } ASN1_SEQUENCE_END(V2FORM)
    IMPLEMENT_ASN1_FUNCTIONS(V2FORM)

    /* ISSUERSERIAL */
    ASN1_SEQUENCE(ISSUERSERIAL) = {
        ASN1_SIMPLE(ISSUERSERIAL, issuer,GENERAL_NAMES),
        ASN1_SIMPLE(ISSUERSERIAL, subjectSN, ASN1_INTEGER),
        ASN1_OPT(ISSUERSERIAL, issuerUID,ASN1_BIT_STRING)
    } ASN1_SEQUENCE_END(ISSUERSERIAL)
    IMPLEMENT_ASN1_FUNCTIONS(ISSUERSERIAL)

    /* OBJDIGEST */
    ASN1_SEQUENCE(OBJDIGEST) = {
        ASN1_SIMPLE(OBJDIGEST, digestType, ASN1_ENUMERATED),
        ASN1_OPT(OBJDIGEST, otherType, ASN1_OBJECT),
        ASN1_SIMPLE(OBJDIGEST, digestAlg, X509_ALGOR),
        ASN1_SIMPLE(OBJDIGEST, digestBit, ASN1_BIT_STRING)
    } ASN1_SEQUENCE_END(OBJDIGEST)
    IMPLEMENT_ASN1_FUNCTIONS(OBJDIGEST)

    /* X509V4_VALID */
    ASN1_SEQUENCE(X509V4_VALID) = {
        ASN1_SIMPLE(X509V4_VALID, notBefore, ASN1_GENERALIZEDTIME),
        ASN1_SIMPLE(X509V4_VALID, notAfter, ASN1_GENERALIZEDTIME)
    } ASN1_SEQUENCE_END(X509V4_VALID)
    IMPLEMENT_ASN1_FUNCTIONS(X509V4_VALID)

    /* X509V4_CINF */
    ASN1_SEQUENCE(X509V4_CINF) = {
    ASN1_SIMPLE(X509V4_CINF,version, ASN1_INTEGER),
    ASN1_SIMPLE(X509V4_CINF, holder, ACHOLDER),
        ASN1_SIMPLE(X509V4_CINF, issuer, ACISSUER),
```

```
        ASN1_SIMPLE(X509V4_CINF, signature, X509_ALGOR),
        ASN1_SIMPLE(X509V4_CINF, serialNumber, ASN1_INTEGER),
        ASN1_SIMPLE(X509V4_CINF, valid, X509V4_VALID),
        ASN1_SEQUENCE_OF(X509V4_CINF, attributes, X509_ATTRIBUTE),
        ASN1_OPT(X509V4_CINF, issuerUID, ASN1_BIT_STRING),
        ASN1_SEQUENCE_OF_OPT(X509V4_CINF, extensions, X509_EXTENSION)
} ASN1_SEQUENCE_END(X509V4_CINF)
IMPLEMENT_ASN1_FUNCTIONS(X509V4_CINF)

ASN1_SEQUENCE(X509V4) = {
        ASN1_SIMPLE(X509V4, cert_info, X509V4_CINF),
        ASN1_SIMPLE(X509V4, sig_alg, X509_ALGOR),
        ASN1_SIMPLE(X509V4, signature, ASN1_BIT_STRING)
} ASN1_SEQUENCE_END(X509V4)
```

第十四章　错误处理

14.1 概述

程序设计时，一般通过函数的返回值来判断是否调用成功。设计良好的函数以及好的错误处理能帮助调用者快速找到错误原因。错误处理应该尽可能多的包含各种信息，包括：

- 错误码；
- 出错文件以及行号；
- 错误原因；
- 出错函数；
- 出错库；
- 出错模块与类别信息；
- 错误堆栈信息等。

并且，出错信息最好能支持多种输出。可以是输出在标准输出上，也可以是文件等形式。

14.2 数据结构

openssl 中，通过 unsigned long 类型来存放错误信息。它包含三部分内容：库代码、函数代码以及错误原因代码。其中，库代码在 crypto/err.h 中定义，函数代码以及错误原因代码由各个功能模块定义（同类代码不能与其他的重复，也不能超过一定的大小）。比如 err.h 中为 BIO 定义如下库代码：

```
/* library */
#define ERR_LIB_BIO        32
```

而 crypto/bio.h 中定义了如下函数和错误原因代号：

```
/* Function codes. */
#define BIO_F_ACPT_STATE                    100
/* Reason codes. */
#define BIO_R_ACCEPT_ERROR                  100
```

错误信息通过上述三部分通过计算得到，并且根据此信息能提取各个代码。计算函数在 err.h 中定义如下：

```
#define ERR_PACK(l,f,r)          (((((unsigned long)l)&0xffL)*0x1000000)| \
                  ((((unsigned long)f)&0xfffL)*0x1000)| \
                  ((((unsigned long)r)&0xfffL)))
#define ERR_GET_LIB(l)           (int)((((unsigned long)l)>>24L)&0xffL)
#define ERR_GET_FUNC(l)          (int)((((unsigned long)l)>>12L)&0xfffL)
#define ERR_GET_REASON(l)        (int)((l)&0xfffL)
```

可以看出，库的个数不能大于 255（0xff），函数个数和错误原因不能大于 4095（0xfff）。除非计算出来的值与已有的值没有冲突。

主要数据结构有两个，定义在 crypto/err/err.h 中，如下：

1）ERR_STRING_DATA

```
typedef struct ERR_string_data_st
```

```
{
unsigned long error;
const char *string;
} ERR_STRING_DATA;
```

该数据结构的内容由各个功能模块来设置。其中，error 用来存放错误信息（由库代码、函数代码以及错误原因代码计算得来），string 用来存放文本信息，可以是函数名也可以是错误原因。以 crypto/bio_err.c 为例，它定义了两个全局表，分别用来存放函数信息和错误信息：

```
#define ERR_FUNC(func) ERR_PACK(ERR_LIB_BIO,func,0)
#define ERR_REASON(reason) ERR_PACK(ERR_LIB_BIO,0,reason)
static ERR_STRING_DATA BIO_str_functs[]=
{
    {ERR_FUNC(BIO_F_ACPT_STATE), "ACPT_STATE"},
    ……
}
static ERR_STRING_DATA BIO_str_reasons[]=
{
    {ERR_REASON(BIO_R_ACCEPT_ERROR)          ,"accept error"},
    {ERR_REASON(BIO_R_BAD_FOPEN_MODE)        ,"bad fopen mode"},
    ……
}
```

这两个表通过 ERR_load_BIO_strings 函数来添加到错误信息哈希表中去。为了便于查找，所有模块的错误信息存放在一个全局哈希表中，在 crypto/err.c 中实现。

2）ERR_STATE

```
typedef struct err_state_st
{
unsigned long pid;
int err_flags[ERR_NUM_ERRORS];
unsigned long err_buffer[ERR_NUM_ERRORS];
char *err_data[ERR_NUM_ERRORS];
int err_data_flags[ERR_NUM_ERRORS];
const char *err_file[ERR_NUM_ERRORS];
int err_line[ERR_NUM_ERRORS];
int top,bottom;
} ERR_STATE;
```

该结构用于存放和获取错误信息。由于可能会有多层函数调用(错误堆栈)，这些信息都是一个数组。每个数组代表了一层函数的错误信息。各项意义如下：

pid：当前线程 id。

err_buffer[i]：第 i 层错误码，包含库、函数以及错误原因信息。

err_data[i]：存放第 i 层操作信息。

err_data_flags[i]：存放 err_data[i]相关的标记；比如为 ERR_TXT_MALLOCED 时，表名 err_data[i]中的数据是动态分配内存的，需要释放；为 ERR_TXT_STRING 表名 err_data[i]中的数据是一个字符串，可以用来打印。

err_file[i]：第 i 层错误的文件名。

err_line[i]：第 i 层错误的行号。

top 和 bottom：用于指明 ERR_STATE 的使用状态。top 对应与最后一个错误（错误堆栈的最上层），bottom 对应第一个错误（错误堆栈的最底层）。

当用户需要扩展 openssl 的模块时，可以仿照其他已有模块来实现自己的错误处理。

14.3 主要函数

1) ERR_add_error_data

在本层错误的 err_data 元素中添加说明信息。该函数一般由各个模块调用，比如可以用它说明什么操作导致了错误。

2） ERR_clear_error

清除所有的错误信息。如果不清楚所有错误信息，可能会有其他无关错误遗留在 ERR_STATE 表中。

3） ERR_error_string/ ERR_error_string_n

根据错误码获取具体的错误信息，包括出错的库、出错的函数以及错误原因。

4) ERR_free_strings

释放错误信息哈希表；通常在最后调用。

5) ERR_func_error_string

根据错误号，获取出错的函数信息。

6） ERR_get_err_state_table

获取存放错误的哈希表。

7） ERR_get_error

获取第一个错误号。

8) ERR_get_error_line

根据错误号，获取错误的行号。

9) ERR_get_error_line_data

根据错误号，获取出错信息。

10) ERR_get_implementation

获取错误处理函数，与哈希表操作相关。

11）ERR_get_state

获取 ERR_STATE 表。

12）ERR_lib_error_string

根据错误号，获取是哪个库出错。

13）ERR_load_strings

加载错误信息，由各个模块调用。

14）ERR_load_ASN1_strings

ERR_load_BIO_strings

ERR_load_BN_strings

ERR_load_BUF_strings

ERR_load_COMP_strings

ERR_load_CONF_strings

ERR_load_CRYPTO_strings

ERR_load_crypto_strings

ERR_load_DH_strings

ERR_load_DSA_strings
ERR_load_DSO_strings
ERR_load_EC_strings
ERR_load_ENGINE_strings
ERR_load_ERR_strings
ERR_load_EVP_strings
ERR_load_OBJ_strings
ERR_load_OCSP_strings
ERR_load_PEM_strings
ERR_load_PKCS12_strings
ERR_load_PKCS7_strings
ERR_load_RAND_strings
ERR_load_RSA_strings
ERR_load_UI_strings
ERR_load_X509_strings
ERR_load_X509V3_strings
各个模块实现的，加载各自错误信息。

15）ERR_peek_error
获取第一个错误号。

16）ERR_peek_error_line
获取第一个错误的出错行。

17）ERR_peek_error_line_data
获取第一个错误的行数和错误信息。

18）ERR_peek_last_error
获取最后一个错误号。

19）ERR_peek_last_error_line
获取最后一个错误的行号。

20）ERR_peek_last_error_line_data
获取最后一个错误的行号和错误信息。

21）ERR_print_errors
将错误信息输出到 bio 中。

22）ERR_print_errors_cb
根据用户设置的回调函数来打印错误信息。

23）ERR_print_errors_fp
将错误打印到 FILE 中。

24) ERR_put_error
将错误信息存放到 ERR_STATE 表中 top 指定的错误堆栈(最后的错误)。

25) ERR_reason_error_string
根据错误号得到错误原因。

26) ERR_remove_state
删除线程相关的错误信息。

27) ERR_set_error_data
将错误信息存放到 ERR_STATE 表中 top 指定的错误堆栈(最后的错误)。

28) ERR_unload_strings

从错误哈希表中删除相关信息。

14.4 编程示例

```
#include <openssl/err.h>
#include <openssl/bn.h>
int   mycb(const char *a,size_t b,void *c)
{
      printf("my print : %s\n",a);
      return 0;
}

int   main()
{
      BIO                  *berr;
      unsigned long  err;
      const       char *file,*data,*efunc,*elib,*ereason,*p;
      int                  line,flags;
      char              estring[500];
      FILE                 *fp;

      /*
      ERR_load_crypto_strings();
      */
      ERR_load_BIO_strings();
      ERR_clear_error();
      berr=BIO_new(BIO_s_file());
      BIO_set_fp(berr,stdout,BIO_NOCLOSE);
      BIO_new_file("no.exist","r");
      err=ERR_peek_last_error();
      err=ERR_peek_last_error_line(&file,&line);
      printf("ERR_peek_last_error_line err : %ld,file : %s,line: %d\n",err,file,line);
      err=ERR_peek_last_error_line_data(&file,&line,&data,&flags);
printf("ERR_peek_last_error_line_data err: %ld,file :%s,line :%d,data :%s\n",err,file,line,data);
      err=ERR_peek_error();
      printf("ERR_peek_error err: %ld\n",err);
      err=ERR_peek_error_line(&file,&line);
      printf("ERR_peek_error_line err : %ld,file : %s,line: %d\n",err,file,line);
      err=ERR_peek_error_line_data(&file,&line,&data,&flags);
   printf("ERR_peek_error_line_data err : %ld,file :%s,line :%d,data :%s\n",err,file,line,data);
      err = ERR_get_error_line_data(&file,&line,&data,&flags);
   printf("ERR_get_error_line_data err : %ld,file :%s,line :%d,data :%s\n",err,file,line,data);
      if(err!=0)
      {
```

```
            p=ERR_lib_error_string(err);
            printf("ERR_lib_error_string : %s\n",p);
        }
        err=ERR_get_error();
        if(err!=0)
        {
            printf("ERR_get_error err : %ld\n",err);
            efunc=ERR_func_error_string(err);
            printf("err func : %s\n",efunc);
            elib=ERR_lib_error_string(err);
            printf("err lib : %s\n",efunc);
            ereason=ERR_reason_error_string(err);
            printf("err reason : %s\n",efunc);
            efunc=ERR_func_error_string(err);
            printf("err func : %s\n",efunc);
            elib=ERR_lib_error_string(err);
            printf("err lib : %s\n",efunc);
            ereason=ERR_reason_error_string(err);
            printf("err reason : %s\n",efunc);
            ERR_error_string(err,estring);
            printf("ERR_error_string : %s\n",estring);

            ERR_error_string_n(err,estring,sizeof(estring));
            printf("ERR_error_string_n : %s\n",estring);
        }
        err=ERR_get_error_line(&file,&line);
        printf("err file :%s , err line : %d\n",file,line);
        ERR_print_errors(berr);
        BIO_new_file("no.exist2","r");
        fp=fopen("err.log","w");
        ERR_print_errors_fp(fp);
        fclose(fp);
        BIO_new_file("no.exist3","r");
        ERR_print_errors_cb(mycb,NULL);
    ERR_put_error(ERR_LIB_BN,BN_F_BNRAND,BN_R_BIGNUM_TOO_LONG,__FILE__,
line);
        ERR_print_errors(berr);
        ERR_load_BN_strings();
ERR_put_error(ERR_LIB_BN,BN_F_BNRAND,BN_R_BIGNUM_TOO_LONG,__FILE__,line)
;
        ERR_print_errors(berr);
ERR_put_error(ERR_LIB_BN,BN_F_BNRAND,BN_R_BIGNUM_TOO_LONG,__FILE__,line)
;
        ERR_set_error_data("set date test!\n",ERR_TXT_STRING);
```

```
        err=ERR_set_mark();
        ERR_print_errors(berr);
        ERR_free_strings();
        BIO_free(berr);
        return 0;
}
```

第十五章　摘要与 HMAC

15.1 概述

摘要函数用于将任意数据通过计算获取唯一对应值，而这个值的长度比较短。它是一种多对一的关系。理论上，这个短的值就对应于原来的数据。这个过程是不可逆的，即不能通过摘要值来计算原始数据。摘要在信息安全中有非常重要的作用。很多网络应用都通过摘要计算来存放口令。摘要是安全协议中不可或却的要素，特别是身份认证与签名。用户需要对数据进行签名时，不可能对大的数据进行运算，这样会严重影响性能。如果只对摘要结果进行计算，则会提供运算速度。常用摘要算法有：sha、sha1、sha256 以及 md5 等。其他还有 md4、md2、mdc2 以及 ripemd160 等。

15.2 openssl 摘要实现

openssl 摘要实现的源码位于 crypto 目录下的各个子目录下，如下所示：

- crypto/ripemd：ripemd 摘要实现(包括汇编代码)及其测试程序；
- crypto/md2：md2 摘要实现及其测试程序；
- crypto/mdc2：mdc2 摘要实现及其测试程序；
- crypto/md4：md4 摘要实现及其测试程序；
- crypto/md5：md5 摘要实现及其测试程序；
- crypto/sha：sha、sha1、sha256、sha512 实现及其测试程序(包含汇编源码)。

上述各种摘要源码在 openssl 中都是底层的函数，相对独立，能单独提取出来，而不必包含 openssl 的 libcrypto 库(因为这个库一般比较大)。

15.3 函数说明

所有的摘要算法都有如下几个函数：

1) XXX_Init
   XXX 为具体的摘要算法名称，该函数初始化上下问，用于多数据摘要。
2) XXX_Update
   XXX 为具体的摘要算法名称，进行摘要计算，该函数可运行多次，对多个数据摘要。
3) XXX_Final
   XXX 为具体的摘要算法名称，进行摘要计算，该函数与 1)和 2）一起用。
4) XXX
   对一个数据进行摘要。该函数由上述 1）2）和 3）实现，只是 XXX_Update 只调用一次。对应源码为 XXX_one.c。

这些函数的测试程序，可参考各个目录下对应的测试程序源码。

15.4 编程示例

以下示例了 MD2、MD4、MD5、SHA 和 SHA1 函数的使用方法：

```
#include <stdio.h>
#include <string.h>
#include <openssl/md2.h>
#include <openssl/md4.h>
#include <openssl/md5.h>
#include <openssl/sha.h>

int   main()
{
      unsigned char  in[]="3dsferyewyrtetegvbzVEgarhaggavxcv";
      unsigned char  out[20];
      size_t                  n;
      int                     i;

      n=strlen((const char*)in);
      #ifdef            OPENSSL_NO_MDC2
      printf("默认 openssl 安装配置无 MDC2\n");
      #else
      MDC2(in,n,out);
      printf("MDC2 digest result :\n");
      for(i=0;i<16;i++)
            printf("%x ",out[i]);
      #endif
      RIPEMD160(in,n,out);
      printf("RIPEMD160 digest result :\n");
      for(i=0;i<20;i++)
            printf("%x ",out[i]);
      MD2(in,n,out);
      printf("MD2 digest result :\n");
      for(i=0;i<16;i++)
            printf("%x ",out[i]);

      MD4(in,n,out);
      printf("\n\nMD4 digest result :\n");
      for(i=0;i<16;i++)
            printf("%x ",out[i]);

      MD5(in,n,out);
      printf("\n\nMD5 digest result :\n");
      for(i=0;i<16;i++)
            printf("%x ",out[i]);
```

```
    SHA(in,n,out);
    printf("\n\nSHA digest result :\n");
    for(i=0;i<20;i++)
        printf("%x ",out[i]);

    SHA1(in,n,out);
    printf("\n\nSHA1 digest result :\n");
    for(i=0;i<20;i++)
        printf("%x ",out[i]);

    SHA256(in,n,out);
    printf("\n\nSHA256 digest result :\n");
    for(i=0;i<32;i++)
        printf("%x ",out[i]);

    SHA512(in,n,out);
    printf("\n\nSHA512 digest result :\n");
    for(i=0;i<64;i++)
        printf("%x ",out[i]);
    printf("\n");
    return 0;
}
```

以上示例中演示了各种摘要计算函数的使用方法。对输入数据 in 进行摘要计算，结果存放在 out 缓冲区中。其中：

mdc2、md4 和 md5 摘要结果为 16 字节，128 比特；

ripemd160、sha 和 sha1 摘要结果为 20 字节，160bit；

sha256 摘要结果为 32 字节，256bit；

sha512 摘要结果为 64 字节，512bit。

15.5 HMAC

HMAC 用于保护消息的完整性，它采用摘要算法对消息、填充以及秘密密钥进行混合运算。在消息传输时，用户不仅传送消息本身，还传送 HMAC 值。接收方接收数据后也进行 HMAC 运算，再比对 MAC 值是否一致。由于秘密密钥只有发送方和接收方才有，其他人不可能伪造假的 HMAC 值，从而能够知道消息是否被篡改。

ssl 协议中用 HMAC 来保护发送消息，并且 ssl 客户端和服务端的 HMAC 密钥是不同的，即对于双方都有一个读 MAC 保护密钥和写 MAC 保护密钥。

HMAC 的实现在 crypto/hmac/hmac.c 中，如下：

```
unsigned char *HMAC(const EVP_MD *evp_md, const void *key, int key_len,
        const unsigned char *d, size_t n, unsigned char *md,
        unsigned int *md_len)
{
    HMAC_CTX c;
```

```
        static unsigned char m[EVP_MAX_MD_SIZE];

        if (md == NULL) md=m;
        HMAC_CTX_init(&c);
        HMAC_Init(&c,key,key_len,evp_md);
        HMAC_Update(&c,d,n);
        HMAC_Final(&c,md,md_len);
        HMAC_CTX_cleanup(&c);
        return(md);
    }
```

evp_md 指明 HMAC 使用的摘要算法；
key 为秘密密钥指针地址；
key_len 为秘密密钥的长度；
d 为需要做 HMAC 运算的数据指针地址；
n 为 d 的长度；
md 用于存放 HMAC 值；
md_len 为 HMAC 值的长度。

第十六章　数据压缩

16.1 简介

数据压缩是将原有数据通过某种压缩算法计算得到相对数据量小的过程。这种过程是可逆的，即能通过压缩后的数据恢复出原数据。数据压缩能够节省存储空间，减轻网络负载。

在即需要加密又需要压缩的情况下，必须先压缩再加密，次序不能颠倒。因为加密后的数据是一个无序的数据，对它进行数据压缩，效果不大。

SSL 协议本身支持压缩算法，Openssl 实现也支持压缩算法。它实现了一个空的压缩算法(crypto/comp/c_rle.c)并支持 zlib 压缩算法（crypto/comp/ c_zlib.c）。openssl 中用户可实现自己的压缩算法。

当 openssl 在有 zlib 库的平台下安装时，需要有 zlib 或者 zlib-dynamic 选项。比如：

./config　zlib

./config　zlib-dynamic

16.2 数据结构

Openssl 通过函数地址来抽象数据压缩。主要数据结构如下：

1) COMP_METHOD

该数据结构定义了具体压缩/解压函数，这些函数可由用户自己实现。

```
typedef struct comp_method_st
{
    int type;
    const char *name;
    int (*init)(COMP_CTX *ctx);
    void (*finish)(COMP_CTX *ctx);
    int (*compress)(COMP_CTX *ctx,unsigned char *out, unsigned int olen,
    unsigned char *in, unsigned int ilen);
    int (*expand)(COMP_CTX *ctx,unsigned char *out, unsigned int olen,
    unsigned char *in, unsigned int ilen);
    long (*ctrl)(void);
    long (*callback_ctrl)(void);
} COMP_METHOD;
```

各项意义如下：

type：压缩算法的 nid；

name：压缩算法的名字；

init：初始化函数；

finish：结束操作；

commpress：具体的压缩算法，本函数必须实现；

expand：具体的解压算法，本函数必须实现；

ctrl 和 callback_ctrl：控制函数与回调控制函数，用于内部控制。

通过 COMP_METHOD，Openssl 能调用用户自己实现的压缩算法。只要用户实现了 COMP_METHOD 中的各个函数(主要是 compress 和 expand 函数)。

Openssl 压缩源码位于 crypto/comp 目录下。它实现了一个空压缩算法和 zlib 压缩算法。其中空压缩算法由 openssl 自己实现，只是简单的拷贝数据。而 zlib 算法，openssl 实现了基于其接口的 COMP_METHOD，需要 zlib 库支持(/usr/lib/libz.a，/usr/lib/libz.so)。

2) comp_ctx

该结构用于存放压缩/解压中的上下文数据，主要供 crypto/comp/comp_lib.c 使用。

```
struct comp_ctx_st
{
    COMP_METHOD *meth;
    unsigned long compress_in;
    unsigned long compress_out;
    unsigned long expand_in;
    unsigned long expand_out;
    CRYPTO_EX_DATA    ex_data;
};
```

各项意义如下：

meth：COMP_METHOD 结构，一个 comp_ctx 通过它指明了一种具体的压缩算法；

compress_in：被压缩数据总字节数；

compress_out：压缩数据(结果)总字节数；

expand_in：被解压数据总字节数；

expand_out：解压数据（结果）总字节数；

ex_data：供用户使用的扩展数据，用于存放用户自定义的信息。

16.3 函数说明

1) COMP_rle

返回 openssl 实现的空压缩算法，返回值为一个 COMP_METHOD。

2) COMP_zlib

返回基于 zlib 库的 COMP_METHOD。

3） COMP_CTX_new

初始化上下文，输入参数为 COMP_METHOD。

4） COMP_compress_block

压缩计算。

5） COMP_expand_block

解压计算。

16.4 openssl 中压缩算法协商

Openssl 中的压缩算法的协商与加密套件一样，都是由客户端在 client hello 消息中指定一个算法列表，而由服务端决定选取其中的一种，并通过 server hello 消息来通知客户端。

16.5 编程示例

```
#include <string.h>
#include <openssl/comp.h>
int    main()
{
       COMP_CTX          *ctx;
       int               len,olen=100,ilen=50,i,total=0;
       unsigned char  in[50],out[100];
       unsigned char  expend[200];

#ifdef     _WIN32
       ctx=COMP_CTX_new(COMP_rle());
#else
       /* for linux */
       ctx=COMP_CTX_new(COMP_zlib());
#endif
       for(i=0;i<50;i++)
              memset(&in[i],i,1);
       total=COMP_compress_block(ctx,out,olen,in,50);
       len=COMP_expand_block(ctx,expend,200,out,total);
       COMP_CTX_free(ctx);
       return 0;
}
```

第十七章 RSA

17.1 RSA 介绍

RSA 算法是一个广泛使用的公钥算法。其密钥包括公钥和私钥。它能用于数字签名、身份认证以及密钥交换。RSA 密钥长度一般使用 1024 位或者更高。RSA 密钥信息主要包括[1]：

- n：模数
- e：公钥指数
- d：私钥指数
- p：最初的大素数
- q：最初的大素数
- dmp1：e*dmp1 = 1 (mod (p-1))
- dmq1：e*dmq1 = 1 (mod (q-1))
- iqmp：q*iqmp = 1 (mod p)

其中，公钥为 n 和 e；私钥为 n 和 d。在实际应用中，公钥加密一般用来协商密钥；私钥加密一般用来签名。

17.2 openssl 的 RSA 实现

Openssl 的 RSA 实现源码在 crypto/rsa 目录下。它实现了 RSA PKCS1 标准。主要源码如下：

1） rsa.h
定义 RSA 数据结构以及 RSA_METHOD，定义了 RSA 的各种函数。

2) rsa_asn1.c
实现了 RSA 密钥的 DER 编码和解码，包括公钥和私钥。

3） rsa_chk.c
RSA 密钥检查。

4） rsa_eay.c
Openssl 实现的一种 RSA_METHOD，作为其默认的一种 RSA 计算实现方式。此文件未实现 rsa_sign、rsa_verify 和 rsa_keygen 回调函数。

5）rsa_err.c
RSA 错误处理。

6）rsa_gen.c
RSA 密钥生成，如果 RSA_METHOD 中的 rsa_keygen 回调函数不为空，则调用它，否则调用其内部实现。

7）rsa_lib.c
主要实现了 RSA 运算的四个函数(公钥/私钥，加密/解密)，它们都调用了 RSA_METHOD 中相应都回调函数。

8）rsa_none.c
实现了一种填充和去填充。

9）rsa_null.c

实现了一种空的 RSA_METHOD。

10) rsa_oaep.c

实现了 oaep 填充与去填充。

11）rsa_pk1.

实现了 pkcs1 填充与去填充。

12）rsa_sign.c

实现了 RSA 的签名和验签。

13）rsa_ssl.c

实现了 ssl 填充。

14）rsa_x931.c

实现了一种填充和去填充。

17.3 RSA 签名与验证过程

RSA 签名过程如下：

1) 对用户数据进行摘要；

2） 构造 X509_SIG 结构并 DER 编码，其中包括了摘要算法以及摘要结果。

3） 对 2）的结果进行填充，填满 RSA 密钥长度字节数。比如 1024 位 RSA 密钥必须填满 128 字节。具体的填充方式由用户指定。

4） 对 3）的结果用 RSA 私钥加密。

RSA_eay_private_encrypt 函数实现了 3）和 4）过程。

RSA 验签过程是上述过程的逆过程，如下：

1) 对数据用 RSA 公钥解密，得到签名过程中 2）的结果。

2) 去除 1）结果的填充。

3) 从 2）的结果中得到摘要算法，以及摘要结果。

4) 将原数据根据 3）中得到摘要算法进行摘要计算。

5） 比较 4）与签名过程中 1）的结果。

RSA_eay_public_decrypt 实现了 1）和 2）过程。

17.4 数据结构

RSA 主要数据结构定义在 crypto/rsa/rsa.h 中：

17.4.1 RSA_METHOD

```
struct rsa_meth_st
{
const char      *name;
int (*rsa_pub_enc)(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding);
int (*rsa_pub_dec)(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int
```

padding);

int (*rsa_priv_enc)(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding);

int (*rsa_priv_dec)(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding);

/* 其他函数 */

int (*rsa_sign)(int type,const unsigned char *m, unsigned int m_length,unsigned char *sigret, unsigned int *siglen, const RSA *rsa);

int (*rsa_verify)(int dtype,const unsigned char *m, unsigned int m_length,unsigned char *sigbuf, unsigned int siglen, const RSA *rsa);

int (*rsa_keygen)(RSA *rsa, int bits, BIGNUM *e, BN_GENCB *cb);

};

主要项说明：

name：RSA_METHOD 名称；

rsa_pub_enc：公钥加密函数，padding 为其填充方式，输入数据不能太长，否则无法填充；

rsa_pub_dec：公钥解密函数，padding 为其去除填充的方式，输入数据长度为 RSA 密钥长度的字节数；

rsa_priv_enc：私钥加密函数，padding 为其填充方式，输入数据长度不能太长，否则无法填充；

rsa_priv_dec：私钥解密函数，padding 为其去除填充的方式，输入数据长度为 RSA 密钥长度的字节数；

rsa_sign：签名函数；

rsa_verify：验签函数；

rsa_keygen：RSA 密钥对生成函数。

用户可实现自己的 RSA_METHOD 来替换 openssl 提供的默认方法。

17.4.2 RSA

RSA 数据结构中包含了公/私钥信息（如果仅有 n 和 e，则表明是公钥），定义如下：

```
struct rsa_st
{
/* 其他 */
const RSA_METHOD *meth;
ENGINE *engine;
BIGNUM *n;
BIGNUM *e;
BIGNUM *d;
BIGNUM *p;
BIGNUM *q;
BIGNUM *dmp1;
BIGNUM *dmq1;
BIGNUM *iqmp;
CRYPTO_EX_DATA ex_data;
```

```
int references;
/* 其他数据项 */
};
```

各项意义：

meth：RSA_METHOD 结构，指明了本 RSA 密钥的各种运算函数地址；

engine：硬件引擎；

n，e，d，p，q，dmp1，dmq1，iqmp：RSA 密钥的各个值；

ex_data：扩展数据结构，用于存放用户数据；

references：RSA 结构引用数。

17.5 主要函数

1） RSA_check_key

检查 RSA 密钥。

2）RSA_new

生成一个 RSA 密钥结构，并采用默认的 rsa_pkcs1_eay_meth RSA_METHOD 方法。

3）RSA_free

释放 RSA 结构。

4) RSA *RSA_generate_key(int bits, unsigned long e_value,
void (*callback)(int,int,void *), void *cb_arg)

生成 RSA 密钥，bits 是模数比特数，e_value 是公钥指数 e，callback 回调函数由用户实现，用于干预密钥生成过程中的一些运算，可为空。

5） RSA_get_default_method

获取默认的 RSA_METHOD，为 rsa_pkcs1_eay_meth。

6） RSA_get_ex_data

获取扩展数据。

7） RSA_get_method

获取 RSA 结构的 RSA_METHOD。

8） RSA_padding_add_none
RSA_padding_add_PKCS1_OAEP
RSA_padding_add_PKCS1_type_1（私钥加密的填充）
RSA_padding_add_PKCS1_type_2（公钥加密的填充）
RSA_padding_add_SSLv23
各种填充方式函数。

9） RSA_padding_check_none
RSA_padding_check_PKCS1_OAEP
RSA_padding_check_PKCS1_type_1
RSA_padding_check_PKCS1_type_2
RSA_padding_check_SSLv23
RSA_PKCS1_SSLeay
各种去除填充函数。

10）int RSA_print(BIO *bp, const RSA *x, int off)

将 RSA 信息输出到 BIO 中，off 为输出信息在 BIO 中的偏移量，比如是屏幕 BIO，则表示打印信息的位置离左边屏幕边缘的距离。

11）int DSA_print_fp(FILE *fp, const DSA *x, int off)
将 RSA 信息输出到 FILE 中，off 为输出偏移量。
12）RSA_public_decrypt
RSA 公钥解密。
13）RSA_public_encrypt
RSA 公钥加密。
14）RSA_set_default_method/ RSA_set_method
设置 RSA 结构中的 method，当用户实现了一个 RSA_METHOD 时，调用此函数来设置，使 RSA 运算采用用户的方法。
15）RSA_set_ex_data
设置扩展数据。
16）RSA_sign
RSA 签名。
17）RSA_sign_ASN1_OCTET_STRING
另外一种 RSA 签名，不涉及摘要算法，它将输入数据作为 ASN1_OCTET_STRING 进行 DER 编码，然后直接调用 RSA_private_encrypt 进行计算。
18）RSA_size
获取 RSA 密钥长度字节数。
19）RSA_up_ref
给 RSA 密钥增加一个引用。
20）RSA_verify
RSA 验证。
21）RSA_verify_ASN1_OCTET_STRING
另一种 RSA 验证，不涉及摘要算法，与 RSA_sign_ASN1_OCTET_STRING 对应。
22）RSAPrivateKey_asn1_meth
获取 RSA 私钥的 ASN1_METHOD，包括 i2d、d2i、new 和 free 函数地址。
23）RSAPrivateKey_dup
复制 RSA 私钥。
24）RSAPublicKey_dup
复制 RSA 公钥。

17.6 编程示例

17.6.1 密钥生成

```
#include <openssl/rsa.h>
int   main()
{
      RSA                 *r;
      int                 bits=512,ret;
      unsigned long e=RSA_3;
      BIGNUM                  *bne;
```

```
    r=RSA_generate_key(bits,e,NULL,NULL);
    RSA_print_fp(stdout,r,11);
    RSA_free(r);
    bne=BN_new();
    ret=BN_set_word(bne,e);
    r=RSA_new();
    ret=RSA_generate_key_ex(r,bits,bne,NULL);
    if(ret!=1)
    {
        printf("RSA_generate_key_ex err!\n");
        return -1;
    }
    RSA_free(r);
    return 0;
}
```

说明：

调用 RSA_generate_key 和 RSA_generate_key_ex 函数生成 RSA 密钥，

调用 RSA_print_fp 打印密钥信息。

输出：

```
Private-Key: (512 bit)
modulus:
    00:d0:93:40:10:21:dd:c2:0b:6a:24:f1:b1:d5:b5:
    77:79:ed:a9:a4:10:66:6e:88:d6:9b:0b:4c:91:7f:
    23:6f:8f:0d:9e:9a:b6:7c:f9:47:fc:20:c2:12:e4:
    b4:d7:ab:66:3e:73:d7:78:00:e6:5c:98:35:29:69:
    c2:9b:c7:e2:c3
publicExponent: 3 (0x3)
privateExponent:
    00:8b:0c:d5:60:16:93:d6:b2:46:c3:4b:cb:e3:ce:
    4f:a6:9e:71:18:0a:ee:f4:5b:39:bc:b2:33:0b:aa:
    17:9f:b3:7e:f0:0f:2a:24:b6:e4:73:40:ba:a0:65:
    d3:19:0f:c5:b5:4f:59:51:e2:df:9c:83:47:da:8d:
    84:0f:26:df:1b
prime1:
    00:f7:4c:fb:ed:32:a6:74:5c:2d:6c:c1:c5:fe:3a:
    59:27:6a:53:5d:3e:73:49:f9:17:df:43:79:d4:d0:
    46:2f:0d
prime2:
    00:d7:e9:88:0a:13:40:7c:f3:12:3d:60:85:f9:f7:
    ba:96:44:29:74:3e:b9:4c:f8:bb:6a:1e:1b:a7:b4:
    c7:65:0f
exponent1:
    00:a4:dd:fd:48:cc:6e:f8:3d:73:9d:d6:83:fe:d1:
```

```
        90:c4:f1:8c:e8:d4:4c:db:fb:65:3f:82:51:38:8a:
        d9:74:b3
exponent2:
        00:8f:f1:05:5c:0c:d5:a8:a2:0c:28:eb:03:fb:fa:
        7c:64:2d:70:f8:29:d0:dd:fb:27:9c:14:12:6f:cd:
        da:43:5f
coefficient:
        00:d3:fa:ea:a0:21:7e:8a:e1:ab:c7:fd:e9:3d:cb:
        5d:10:96:17:69:75:cd:71:d5:e5:07:26:93:e8:35:
        ca:e3:49
```

17.6.2 RSA 加解密运算

```
#include <openssl/rsa.h>
#include <openssl/sha.h>
int    main()
{
        RSA                 *r;
        int                 bits=1024,ret,len,flen,padding,i;
        unsigned long  e=RSA_3;
        BIGNUM                  *bne;
        unsigned char  *key,*p;
        BIO                 *b;
        unsigned char  from[500],to[500],out[500];

        bne=BN_new();
        ret=BN_set_word(bne,e);
        r=RSA_new();
        ret=RSA_generate_key_ex(r,bits,bne,NULL);
        if(ret!=1)
        {
                printf("RSA_generate_key_ex err!\n");
                return -1;
        }
        /*  私钥 i2d */

        b=BIO_new(BIO_s_mem());
        ret=i2d_RSAPrivateKey_bio(b,r);
        key=malloc(1024);
        len=BIO_read(b,key,1024);
        BIO_free(b);
        b=BIO_new_file("rsa.key","w");
        ret=i2d_RSAPrivateKey_bio(b,r);
        BIO_free(b);
```

```
/*  私钥 d2i */
/*  公钥 i2d */
/*  公钥 d2i */
/*  私钥加密  */
flen=RSA_size(r);
printf("please select private enc padding : \n");
printf("1.RSA_PKCS1_PADDING\n");
printf("3.RSA_NO_PADDING\n");
printf("5.RSA_X931_PADDING\n");
scanf("%d",&padding);
if(padding==RSA_PKCS1_PADDING)
	flen-=11;
else if(padding==RSA_X931_PADDING)
	flen-=2;
else if(padding==RSA_NO_PADDING)
	flen=flen;
else
{
	printf("rsa not surport !\n");
	return -1;
}
for(i=0;i<flen;i++)
	memset(&from[i],i,1);
len=RSA_private_encrypt(flen,from,to,r,padding);
if(len<=0)
{
	printf("RSA_private_encrypt err!\n");
	return -1;
}
len=RSA_public_decrypt(len,to,out,r,padding);
if(len<=0)
{
	printf("RSA_public_decrypt err!\n");
	return -1;
}
if(memcmp(from,out,flen))
{
	printf("err!\n");
	return -1;
}
/* */
printf("please select public enc padding : \n");
printf("1.RSA_PKCS1_PADDING\n");
```

```
    printf("2.RSA_SSLV23_PADDING\n");
    printf("3.RSA_NO_PADDING\n");
    printf("4.RSA_PKCS1_OAEP_PADDING\n");
    scanf("%d",&padding);
    flen=RSA_size(r);
    if(padding==RSA_PKCS1_PADDING)
        flen-=11;
    else if(padding==RSA_SSLV23_PADDING)
        flen-=11;
    else if(padding==RSA_X931_PADDING)
        flen-=2;
    else if(padding==RSA_NO_PADDING)
        flen=flen;
    else if(padding==RSA_PKCS1_OAEP_PADDING)
        flen=flen-2 * SHA_DIGEST_LENGTH-2 ;
    else
    {
        printf("rsa not surport !\n");
        return -1;
    }
    for(i=0;i<flen;i++)
        memset(&from[i],i+1,1);
    len=RSA_public_encrypt(flen,from,to,r,padding);
    if(len<=0)
    {
        printf("RSA_public_encrypt err!\n");
        return -1;
    }
    len=RSA_private_decrypt(len,to,out,r,padding);
    if(len<=0)
    {
        printf("RSA_private_decrypt err!\n");
        return -1;
    }
    if(memcmp(from,out,flen))
    {
        printf("err!\n");
        return -1;
    }
    printf("test ok!\n");
    RSA_free(r);
    return 0;
}
```

上述程序中当采用公钥 RSA_SSLV23_PADDING 加密，用私钥 RSA_SSLV23_PADDING

解密时会报错，原因是 openssl 源代码错误：

rsa_ssl.c 函数 RSA_padding_check_SSLv23 有：

```
if (k == -1)     /* err */
{
RSAerr(RSA_F_RSA_PADDING_CHECK_SSLV23,RSA_R_SSLV3_ROLLBACK_ATTA
CK);
return (-1);
}
```

修改为 k!=-1 即可。

各种 padding 对输入数据长度的要求：

私钥加密：

RSA_PKCS1_PADDING	RSA_size-11
RSA_NO_PADDING	RSA_size-0
RSA_X931_PADDING	RSA_size-2

公钥加密

RSA_PKCS1_PADDING	RSA_size-11
RSA_SSLV23_PADDING	RSA_size-11
RSA_X931_PADDING	RSA_size-2
RSA_NO_PADDING	RSA_size-0
RSA_PKCS1_OAEP_PADDING	RSA_size-2 * SHA_DIGEST_LENGTH-2

17.6.3 签名与验证

签名运算：

```
#include <string.h>
#include <openssl/objects.h>
#include <openssl/rsa.h>

int   main()
{
int                  ret;
RSA                  *r;
int                  i,bits=1024,signlen,datalen,alg,nid;
unsigned long e=RSA_3;
BIGNUM                  *bne;
unsigned char  data[100],signret[200];

bne=BN_new();
ret=BN_set_word(bne,e);
r=RSA_new();
ret=RSA_generate_key_ex(r,bits,bne,NULL);
if(ret!=1)
{
      printf("RSA_generate_key_ex err!\n");
```

```
        return -1;
}

for(i=0;i<100;i++)
        memset(&data[i],i+1,1);
printf("please select digest alg: \n");
printf("1.NID_md5\n");
printf("2.NID_sha\n");
printf("3.NID_sha1\n");
printf("4.NID_md5_sha1\n");
scanf("%d",&alg);
if(alg==1)
{
        datalen=55;
        nid=NID_md5;
}
else if(alg==2)
{
        datalen=55;
        nid=NID_sha;
}
else if(alg==3)
{
        datalen=55;
        nid=NID_sha1;
}
else if(alg==4)
{
        datalen=36;
        nid=NID_md5_sha1;
}
ret=RSA_sign(nid,data,datalen,signret,&signlen,r);
if(ret!=1)
{
        printf("RSA_sign err!\n");
        RSA_free(r);
        return -1;
}
ret=RSA_verify(nid,data,datalen,signret,signlen,r);
if(ret!=1)
{
        printf("RSA_verify err!\n");
        RSA_free(r);
        return -1;
```

```
        }
        RSA_free(r);
        printf("test ok!\n");
        return 0;
    }
```

注意：本示例并不是真正的数据签名示例，因为没有做摘要计算。

ret=RSA_sign(nid,data,datalen,signret,&signlen,r)将需要运算的数据放入 X509_ALGOR 数据结构并将其 DER 编码，对编码结果做 RSA_PKCS1_PADDING 再进行私钥加密。

被签名数据应该是摘要之后的数据，而本例没有先做摘要，直接将数据拿去做运算。因此 datalen 不能太长，要保证 RSA_PKCS1_PADDING 私钥加密运算时输入数据的长度限制。

ret=RSA_verify(nid,data,datalen,signret,signlen,r)用来验证签名。

参考文献：

[1] PKCS #1 v2.1: RSA Cryptography Standard

第十八章 DSA

18.1 DSA 简介

Digital Signature Algorithm (DSA)算法是一种公钥算法。其密钥由如下部分组成：

1）p

一个大素数，长度为 L(64 的整数倍)比特。

2）q

一个 160 比特素数。

3）g

$g=h^{(p-1)/q} \bmod p$，其中 h 小于 p-1。

4）x

小于 q。

5) y

$y=g^x \bmod p$。

其中 x 为私钥，y 为公钥。p、q 和 g 是公开信息(openssl 中称为密钥参数)。

DSA 签名包括两部分，如下：

$r = (g^k \bmod p) \bmod q$

$s = (k^{-1} (H(m) + xr)) \bmod q$

其中，H(m)为摘要算法；

DSA 验签如下：

$w = s^{-1} \bmod q$

$u1 = (H(m) * w) \bmod q$

$u2 = (rw) \bmod q$

$v = ((g^{u1} * y^{u2}) \bmod p) \bmod q$

如果 v=r，则验证通过。

18.2 openssl 的 DSA 实现

Openssl 的 DSA 实现源码在 crypto/dsa 目录下。主要源码如下：

1) dsa_asn1.c

DSA 密钥参数(p、q 和 g)、DSA 公钥（pub_key、p、q 和 g）以及 DSA 私钥(priv_key、pub_key、p、q 和 g)的 DER 编解码实现。

2）dsa_depr.c

生成 DSA 密钥参数。

3）dsa_err.c

DSA 错误处理。

4）dsa_gen.c

生成 DSA 密钥参数。

5）dsa_key.c

根据 DSA 中的密钥参数产生公钥和私钥。

6）dsa_lib.c

实现了 DSA 通用的一些函数。

7）dsa_ossl.c

实现了一个 DSA_METHOD，该 DSA_METHOD 为 openssl 默认的 DSA 方法，主要实现了如下三个回调函数：dsa_do_sign（签名）、dsa_sign_setup（根据密钥参数生成公私钥）和 dsa_do_verify（验签）。

8）dsa_sign.c

实现了 DSA 签名和根据密钥参数生成公私钥。

9）dsa_vrf.c

实现了 DSA 验签。

18.3 DSA 数据结构

DSA 数据结构定义在 crypto/dsa.h 中，如下所示：

1) DSA_SIG

签名值数据结构

```
typedef struct DSA_SIG_st
{
    BIGNUM *r;
    BIGNUM *s;
} DSA_SIG;
```

签名结果包括两部分，都是大数。

2） DSA_METHOD

```
struct dsa_method
{
    const char *name;
    DSA_SIG * (*dsa_do_sign)(const unsigned char *dgst, int dlen, DSA *dsa);
    int (*dsa_sign_setup)(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp, BIGNUM **rp);
    int (*dsa_do_verify)(const unsigned char *dgst, int dgst_len, DSA_SIG *sig, DSA *dsa);
    /* 其他 */
    int (*dsa_paramgen)(DSA *dsa, int bits,unsigned char *seed, int seed_len, int *counter_ret, unsigned long *h_ret,BN_GENCB *cb);
    int (*dsa_keygen)(DSA *dsa);
};
```

本结构是一个函数集合，DSA 的各种计算都通过它来实现。drypto/dsa_ossl.c 中实现了一个默认的 DSA_METHOD。如果用户实现了自己的 DSA_METHOD，通过调用 DSA_set_default_method 或 DSA_set_method，用户可以让 openssl 采用自己的 DSA 计算函数。

主要项意义如下：

name：DSA_METHOD 的名字；

dsa_do_sign：签名算法函数；

dsa_sign_setup：根据密钥参数生成公私钥的函数；

dsa_do_verify：签名验证函数；
dsa_paramgen：生成密钥参数函数；
dsa_keygen：生成公私钥函数。

18.4 主要函数

1） DSA_do_sign
数据签名。
2) DSA_do_verify
签名验证。
3) DSA_dup_DH
将 DSA 密钥转换为 DH 密钥。
4） DSA_new
生成一个 DSA 数据结构，一般情况下，DSA_METHOD 采用默认的 openssl_dsa_meth 方法。
5) DSA_free
释放 DSA 数据结构。
6) DSA_generate_key
根据密钥参数生成公私钥。
7) DSA_generate_parameters
生成密钥参数。
8） DSA_get_default_method
获取默认的 DSA_METHOD。
9) DSA_get_ex_data
获取扩展数据。
10) DSA_new_method
生成一个 DSA 结构。
11）DSA_OpenSSL
获取 openssl_dsa_meth 方法。
12）DSA_print
将 DSA 密钥信息输出到 BIO 中。
13）DSA_print_fp
将 DSA 密钥信息输出到 FILE 中。
14）DSA_set_default_method
设置默认的 DSA_METHOD。
15) DSA_set_ex_data
设置扩展数据。
16）DSA_set_method
获取当前 DSA 的 DSA_METHOD 方法。
17）DSA_SIG_new
生成一个 DSA_SIG 签名值结构。
18）DSA_SIG_free
释放 DSA_SIG 结构。
19）DSA_sign

DSA 签名。

20）DSA_sign_setup

根据密钥参数生成公私钥。

21）DSA_size

获取 DSA 密钥长度的字节数。

22）DSA_up_ref

给 DSA 结构添加一个引用。

23）DSA_verify

签名验证。

24）DSAparams_print

将 DSA 密钥参数输出到 bio。

25）DSAparams_print_fp

将 DSA 密钥参数输出到 FILE。

18.5 编程示例

18.5.1 密钥生成

```
#include <string.h>
#include <openssl/dsa.h>

int   main()
{
    DSA                         *d;
    int                         ret,i;
    unsigned  char seed[20];
    int                         counter=2;
    unsigned  long h;

    d=DSA_new();
    for(i=0;i<20;i++)
        memset(seed+i,i,1);
    //ret=DSA_generate_parameters_ex(d, 512,seed, 20, &counter,&h,NULL);
    /*  生成密钥参数  */
    ret=DSA_generate_parameters_ex(d, 512,NULL,0,NULL,NULL,NULL);
    if(ret!=1)
    {
        DSA_free(d);
        return -1;
    }
    /*  生成密钥  */
    ret=DSA_generate_key(d);
    if(ret!=1)
```

```
    {
        DSA_free(d);
        return -1;
    }
    DSA_print_fp(stdout,d,0);
    DSA_free(d);
    return 0;
}
```

输出：

```
priv:
    35:8f:e6:50:e7:03:3b:5b:ba:ef:0a:c4:bd:92:e8:
    74:9c:e5:57:6d
pub:
    41:ea:ff:ac:e4:d0:e0:53:2e:cf:f0:c2:34:93:9c:
    bc:b3:d2:f7:50:5e:e3:76:e7:25:b6:43:ed:ac:7b:
    c0:31:7d:ea:50:92:ee:2e:34:38:fa:2d:a6:03:0c:
    4f:f5:89:4b:4b:30:ab:e2:e8:4d:e4:77:f7:e9:4f:
    60:88:2e:2a
P:
    00:ab:8d:e8:b8:be:d1:89:e0:24:6d:4b:4e:cd:43:
    9d:22:36:00:6a:d7:dd:f2:2c:cd:ce:69:9e:5f:87:
    b4:6e:76:5f:e6:ef:74:7c:3b:11:5d:60:50:db:ce:
    00:7e:ea:1e:a9:94:69:69:8b:e1:fc:7f:2a:ca:c2:
    f0:e5:f8:63:c1
Q:
    00:f8:68:d5:d5:4b:85:e6:a7:4f:98:08:bc:00:e2:
    34:2e:94:cd:31:43
G:
    00:8c:1a:09:06:a7:63:4b:cb:e0:c2:85:79:9f:12:
    9d:ac:a7:34:3c:eb:9b:ab:4b:fe:54:c1:22:ff:49:
    ec:17:d1:38:77:f5:2e:85:f7:80:d1:ac:4c:1a:96:
    a1:88:a5:90:66:31:ed:6f:0b:00:f7:2e:df:79:6b:
    95:97:c4:8a:95
```

18.5.2 签名与验证

```
#include <string.h>
#include <openssl/objects.h>
#include <openssl/dsa.h>

int    main()
{
    int                 ret;
    DSA                 *d;
```

```
int                     i,bits=1024,signlen,datalen,alg,nid;
unsigned char  data[100],signret[200];

d=DSA_new();
ret=DSA_generate_parameters_ex(d,512,NULL,0,NULL,NULL,NULL);
if(ret!=1)
{
    DSA_free(d);
    return -1;
}
ret=DSA_generate_key(d);
if(ret!=1)
{
    DSA_free(d);
    return -1;
}
for(i=0;i<100;i++)
    memset(&data[i],i+1,1);
printf("please select digest alg: \n");
printf("1.NID_md5\n");
printf("2.NID_sha\n");
printf("3.NID_sha1\n");
printf("4.NID_md5_sha1\n");
scanf("%d",&alg);
if(alg==1)
{
    datalen=20;
    nid=NID_md5;
}
else if(alg==2)
{
    datalen=20;
    nid=NID_sha;
}
else if(alg==3)
{
    datalen=20;
    nid=NID_sha1;
}
else if(alg==4)
{
    datalen=20;
    nid=NID_md5_sha1;
}
```

```
    ret=DSA_sign(nid,data,datalen,signret,&signlen,d);
    if(ret!=1)
    {
        printf("DSA_sign err!\n");
        DSA_free(d);
        return -1;
    }
    ret=DSA_verify(nid,data,datalen,signret,signlen,d);
    if(ret!=1)
    {
        printf("DSA_verify err!\n");
        DSA_free(d);
        return -1;
    }
    DSA_free(d);
    printf("test ok!\n");
    return 0;
}
```

第十九章 DH

19.1 DH 算法介绍

DH 算法是 W.Diffie 和 M.Hellman 提出的。此算法是最早的公钥算法。它实质是一个通信双方进行密钥协商的协议：两个实体中的任何一个使用自己的私钥和另一实体的公钥，得到一个对称密钥，这一对称密钥其它实体都计算不出来。DH 算法的安全性基于有限域上计算离散对数的困难性。离散对数的研究现状表明：所使用的 DH 密钥至少需要 1024 位，才能保证有足够的中、长期安全。

首先，发送方和接收方设置相同的大数数 n 和 g，这两个数不是保密的，他们可以通过非安全通道来协商这两个素数；

接着，他们用下面的方法协商密钥：

发送方选择一个大随机整数 x，计算 X= g^x mod n ，发送 X 给接收者；

接收方选择一个大随机整数 y，计算 Y = g^y mod n ，发送 Y 给发送方；

双方计算密钥：发送方密钥为 k1=Y^x mod n，接收方密钥为 k2=X^y mod n。

其中 k1=k2=g^(xy) mod n。

其他人可以知道 n、g、X 和 Y，但是他们不能计算出密钥，除非他们能恢复 x 和 y。DH 算法不能抵御中间人攻击，中间人可以伪造假的 X 和 Y 分别发送给双方来获取他们的秘密密钥，所以需要保证 X 和 Y 的来源合法性。

19.2 openssl 的 DH 实现

Openssl 的 DH 实现在 crypto/dh 目录中。各个源码如下：

1） dh.h

定义了 DH 密钥数据结构以及各种函数。

2) dh_asn1.c

DH 密钥参数的 DER 编解码实现。

3） dh_lib.c

实现了通用的 DH 函数。

4） dh_gen.c

实现了生成 DH 密钥参数。

5） dh_key.c

实现 openssl 提供的默认的 DH_METHOD，实现了根据密钥参数生成 DH 公私钥，以及根据 DH 公钥(一方)以及 DH 私钥(另一方)来生成一个共享密钥，用于密钥交换。

6）dh_err.c

实现了 DH 错误处理。

7） dh_check.c

实现了 DH 密钥检查。

19.3 数据结构

DH 数据结构定义在 crypto/dh/dh.h 中，主要包含两项，如下：

1) DH_METHOD

```
struct dh_method
{
    const char *name;
    int (*generate_key)(DH *dh);
    int (*compute_key)(unsigned char *key,const BIGNUM *pub_key,DH *dh);
    int (*bn_mod_exp)(const DH *dh, BIGNUM *r, const BIGNUM *a,
            const BIGNUM *p, const BIGNUM *m, BN_CTX *ctx,
            BN_MONT_CTX *m_ctx);
    int (*init)(DH *dh);
    int (*finish)(DH *dh);
    int flags;
    char *app_data;
    int (*generate_params)(DH *dh, int prime_len, int generator, BN_GENCB *cb);
};
```

DH_METHOD 指明了一个 DH 密钥所有的计算方法函数。用户可以实现自己的 DH_METHOD 来替换 openssl 提供默认方法。各项意义如下：

name：DH_METHOD 方法名称。

generate_key：生成 DH 公私钥的函数。

compute_key：根据对方公钥和己方 DH 密钥来生成共享密钥的函数。

bn_mod_exp：大数模运算函数，如果用户实现了它，生成 DH 密钥时，将采用用户实现的该回调函数。用于干预 DH 密钥生成。

init：初始化函数。

finish：结束函数。

flags：用于记录标记。

app_data：用于存放应用数据。

generate_params：生成 DH 密钥参数的回调函数，生成的密钥参数是可以公开的。

2) DH

```
struct dh_st
{
    /* 其他 */
    BIGNUM *p;
    BIGNUM *g;
    long length; /* optional */
    BIGNUM *pub_key;
    BIGNUM *priv_key;
    int references;
    CRYPTO_EX_DATA ex_data;
    const DH_METHOD *meth;
    ENGINE *engine;
```

```
    /* 其他 */
};
```

p、g、length：DH 密钥参数；
pub_key：DH 公钥；
priv_key：DH 私钥；
references：引用；
ex_data：扩展数据；
meth：DH_METHOD，本 DH 密钥的各种计算方法，明确指明了 DH 的各种运算方式；
engine：硬件引擎。

19.4 主要函数

1） DH_new
生成 DH 数据结构，其 DH_METHOD 采用 openssl 默认提供的。
2） DH_free
释放 DH 数据结构。
3） DH_generate_parameters
生成 DH 密钥参数。
4） DH_generate_key
生成 DH 公私钥。
5） DH_compute_key
计算共享密钥，用于数据交换。
6） DH_check
检查 DH 密钥。
7） DH_get_default_method
获取默认的 DH_METHOD，该方法是可以由用户设置的。
8） DH_get_ex_data
获取 DH 结构中的扩展数据。
9) DH_new_method
生成 DH 数据结构。
10）DH_OpenSSL
获取 openssl 提供的 DH_METHOD。
11）DH_set_default_method
设置默认的 DH_METHOD 方法，当用户实现了自己的 DH_METHOD 时，可调用本函数来设置，控制 DH 各种计算。
12）DH_set_ex_data
获取扩展数据。
13）DH_set_method
替换已有的 DH_METHOD。
14）DH_size
获取 DH 密钥长度的字节数。
15) DH_up_ref
增加 DH 结构的一个引用。
16）DHparams_print

将 DH 密钥参数输出到 bio 中。

17) DHparams_print_fp

将 DH 密钥参数输出到 FILE 中。

19.5 编程示例

```
#include <openssl/dh.h>
#include <memory.h>
int   main()
{
    DH      *d1,*d2;
    BIO     *b;
    int     ret,size,i,len1,len2;
    char sharekey1[128],sharekey2[128];

    /* 构造 DH 数据结构 */
    d1=DH_new();
    d2=DH_new();
    /* 生成 d1 的密钥参数，该密钥参数是可以公开的 */
    ret=DH_generate_parameters_ex(d1,64,DH_GENERATOR_2,NULL);
    if(ret!=1)
    {
        printf("DH_generate_parameters_ex err!\n");
        return -1;
    }
    /* 检查密钥参数 */
    ret=DH_check(d1,&i);
    if(ret!=1)
    {
        printf("DH_check err!\n");
        if(i&DH_CHECK_P_NOT_PRIME)
            printf("p value is not prime\n");
        if(i&DH_CHECK_P_NOT_SAFE_PRIME)
            printf("p value is not a safe prime\n");
        if (i&DH_UNABLE_TO_CHECK_GENERATOR)
            printf("unable to check the generator value\n");
        if (i&DH_NOT_SUITABLE_GENERATOR)
            printf("the g value is not a generator\n");
    }
    printf("DH parameters appear to be ok.\n");
    /* 密钥大小 */
    size=DH_size(d1);
    printf("DH key1 size : %d\n",size);
    /* 生成公私钥 */
```

```
ret=DH_generate_key(d1);
if(ret!=1)
{
	printf("DH_generate_key err!\n");
	return -1;
}
/* p 和 g 为公开的密钥参数，因此可以拷贝 */
d2->p=BN_dup(d1->p);
d2->g=BN_dup(d1->g);
/* 生成公私钥,用于测试生成共享密钥 */
ret=DH_generate_key(d2);
if(ret!=1)
{
	printf("DH_generate_key err!\n");
	return -1;
}
/* 检查公钥 */
ret=DH_check_pub_key(d1,d1->pub_key,&i);
if(ret!=1)
{
	if (i&DH_CHECK_PUBKEY_TOO_SMALL)
		printf("pub key too small \n");
	if (i&DH_CHECK_PUBKEY_TOO_LARGE)
		printf("pub key too large \n");
}
/* 计算共享密钥 */
len1=DH_compute_key(sharekey1,d2->pub_key,d1);
len2=DH_compute_key(sharekey2,d1->pub_key,d2);
if(len1!=len2)
{
	printf("生成共享密钥失败 1\n");
	return -1;
}
if(memcmp(sharekey1,sharekey2,len1)!=0)
{
	printf("生成共享密钥失败 2\n");
	return -1;
}
printf("生成共享密钥成功\n");
b=BIO_new(BIO_s_file());
BIO_set_fp(b,stdout,BIO_NOCLOSE);
/* 打印密钥 */
DHparams_print(b,d1);
BIO_free(b);
```

```
    DH_free(d1);
    DH_free(d2);
    return 0;
}
```

本例主要演示了生成 DH 密钥以及密钥交换函数。

第二十章 椭圆曲线

20.1 ECC 介绍

椭圆曲线(ECC)算法是一种公钥算法，它比流行的 RSA 算法有很多优点：

1）安全性能更高， 如 160 位 ECC 与 1024 位 RSA、DSA 有相同的安全强度。

2）计算量小，处理速度快， 在私钥的处理速度上（解密和签名），ECC 比 RSA、DSA 快得多。

3）存储空间占用小 ECC 的密钥尺寸和系统参数与 RSA、DSA 相比要小得多，所以占用的存储空间小得多。

4）带宽要求低。

20.2 openssl 的 ECC 实现

Openssl 实现了 ECC 算法。ECC 算法系列包括三部分：ECC 算法(crypto/ec)、椭圆曲线数字签名算法 ECDSA (crypto/ecdsa)以及椭圆曲线密钥交换算法 ECDH(crypto/dh)。

研究椭圆曲线需要注意的有：

1) 密钥数据结构

主要是公钥和私钥数据结构。椭圆曲线密钥数据结构如下，定义在 crypto/ec_lcl.h 中，对用户是透明的。

```
struct ec_key_st
{
    int             version;
    EC_GROUP   *group;
    EC_POINT    *pub_key;
    BIGNUM      *priv_key;
    /* 其他项 */
}
```

2) 密钥生成

对照公钥和私钥的表示方法，非对称算法不同有各自的密钥生成过程。椭圆曲线的密钥生成实现在 crytpo/ec/ec_key.c 中。Openssl 中，椭圆曲线密钥生成时，首先用户需要选取一种椭圆曲线(openssl 的 crypto/ec_curve.c 中内置实现了 67 种，调用 EC_get_builtin_curves 获取该列表)，然后根据选择的椭圆曲线计算密钥生成参数 group，最后根据密钥参数 group 来生公私钥。

3） 签名值数据结构

非对称算法不同，签名的结果表示也不一样。与 DSA 签名值一样，ECDSA 的签名结果表示为两项。ECDSA 的签名结果数据结构定义在 crypto/ecdsa/ecdsa.h 中，如下：

```
typedef struct ECDSA_SIG_st
{
    BIGNUM *r;
```

```
    BIGNUM *s;
} ECDSA_SIG;
```

4) 签名与验签

对照签名结果，研究其是如何生成的。crypto/ecdsa/ ecs_sign.c 实现了签名算法，crypto/ecdsa/ ecs_vrf.c 实现了验签。

5） 密钥交换

研究其密钥交换是如何进行的；crypto/ecdh/ech_ossl.c 实现了密钥交换算法。

20.3 主要函数

1） EC_get_builtin_curves
获取椭圆曲线列表。

2） EC_GROUP_new_by_curve_name
根据指定的椭圆曲线来生成密钥参数。

3） int EC_KEY_generate_key
根据密钥参数生成 ECC 公私钥。

4） int EC_KEY_check_key
检查 ECC 密钥。

5） int ECDSA_size
获取 ECC 密钥大小字节数。

6） ECDSA_sign
签名，返回 1 表示成功。

7） ECDSA_verify
验签，返回 1 表示合法。

8） EC_KEY_get0_public_key
获取公钥。

9） EC_KEY_get0_private_key
获取私钥。

10）ECDH_compute_key
生成共享密钥

20.4 编程示例

下面的例子生成两对 ECC 密钥，并用它做签名和验签，并生成共享密钥。

```
#include <string.h>
#include <stdio.h>
#include <openssl/ec.h>
#include <openssl/ecdsa.h>
#include <openssl/objects.h>
#include <openssl/err.h>

int  main()
{
```

```
EC_KEY              *key1,*key2;
EC_POINT            *pubkey1,*pubkey2;
EC_GROUP            *group1,*group2;
int                 ret,nid,size,i,sig_len;
unsigned char  *signature,digest[20];
BIO                 *berr;
EC_builtin_curve    *curves;
int                       crv_len;
char              shareKey1[128],shareKey2[128];
int                 len1,len2;

/*  构造 EC_KEY 数据结构  */
key1=EC_KEY_new();
if(key1==NULL)
{
    printf("EC_KEY_new err!\n");
    return -1;
}
key2=EC_KEY_new();
if(key2==NULL)
{
    printf("EC_KEY_new err!\n");
    return -1;
}
/*  获取实现的椭圆曲线个数  */
crv_len = EC_get_builtin_curves(NULL, 0);
curves = (EC_builtin_curve *)malloc(sizeof(EC_builtin_curve) * crv_len);
/*  获取椭圆曲线列表  */
EC_get_builtin_curves(curves, crv_len);
/*
nid=curves[0].nid;会有错误，原因是密钥太短
*/
/*  选取一种椭圆曲线  */
nid=curves[25].nid;
/*  根据选择的椭圆曲线生成密钥参数 group */
group1=EC_GROUP_new_by_curve_name(nid);
if(group1==NULL)
{
    printf("EC_GROUP_new_by_curve_name err!\n");
    return -1;
}
group2=EC_GROUP_new_by_curve_name(nid);
if(group1==NULL)
{
```

```
        printf("EC_GROUP_new_by_curve_name err!\n");
        return -1;
}
/* 设置密钥参数 */
ret=EC_KEY_set_group(key1,group1);
if(ret!=1)
{
        printf("EC_KEY_set_group err.\n");
        return -1;
}
ret=EC_KEY_set_group(key2,group2);
if(ret!=1)
{
        printf("EC_KEY_set_group err.\n");
        return -1;
}
/* 生成密钥 */
ret=EC_KEY_generate_key(key1);
if(ret!=1)
{
        printf("EC_KEY_generate_key err.\n");
        return -1;
}
ret=EC_KEY_generate_key(key2);
if(ret!=1)
{
        printf("EC_KEY_generate_key err.\n");
        return -1;
}
/* 检查密钥 */
ret=EC_KEY_check_key(key1);
if(ret!=1)
{
        printf("check key err.\n");
        return -1;
}
/* 获取密钥大小 */
size=ECDSA_size(key1);
printf("size %d \n",size);
for(i=0;i<20;i++)
        memset(&digest[i],i+1,1);
signature=malloc(size);
ERR_load_crypto_strings();
berr=BIO_new(BIO_s_file());
```

```
    BIO_set_fp(berr,stdout,BIO_NOCLOSE);
    /* 签名数据，本例未做摘要，可将 digest 中的数据看作是 sha1 摘要结果 */
    ret=ECDSA_sign(0,digest,20,signature,&sig_len,key1);
    if(ret!=1)
    {
        ERR_print_errors(berr);
        printf("sign err!\n");
        return -1;
    }
    /* 验证签名 */
    ret=ECDSA_verify(0,digest,20,signature,sig_len,key1);
    if(ret!=1)
    {
        ERR_print_errors(berr);
        printf("ECDSA_verify err!\n");
        return -1;
    }
    /* 获取对方公钥，不能直接引用 */
    pubkey2 = EC_KEY_get0_public_key(key2);
    /* 生成一方的共享密钥 */
    len1=ECDH_compute_key(shareKey1, 128, pubkey2, key1, NULL);
    pubkey1 = EC_KEY_get0_public_key(key1);
    /* 生成另一方共享密钥 */
    len2=ECDH_compute_key(shareKey2, 128, pubkey1, key2, NULL);
    if(len1!=len2)
    {
        printf("err\n");
    }
    else
    {
        ret=memcmp(shareKey1,shareKey2,len1);
        if(ret==0)
            printf("生成共享密钥成功\n");
        else
            printf("生成共享密钥失败\n");
    }
    printf("test ok!\n");
    BIO_free(berr);
    EC_KEY_free(key1);
    EC_KEY_free(key2);
    free(signature);
    free(curves);
    return 0;
}
```

第二十一章 EVP

21.1 EVP 简介

Openssl EVP(high-level cryptographic functions[1])提供了丰富的密码学中的各种函数。Openssl 中实现了各种对称算法、摘要算法以及签名/验签算法。EVP 函数将这些具体的算法进行了封装。

EVP 主要封装了如下功能函数：

1）实现了 base64 编解码 BIO；

2）实现了加解密 BIO；

3）实现了摘要 BIO；

4）实现了 reliable BIO；

5）封装了摘要算法；

6）封装了对称加解密算法；

7）封装了非对称密钥的加密(公钥)、解密(私钥)、签名与验证以及辅助函数；

7）基于口令的加密(PBE)；

8）对称密钥处理；

9）数字信封：数字信封用对方的公钥加密对称密钥，数据则用此对称密钥加密。发送给对方时，同时发送对称密钥密文和数据密文。接收方首先用自己的私钥解密密钥密文，得到对称密钥，然后用它解密数据。

10）其他辅助函数。

21.2 数据结构

EVP 数据结构定义在 crypto/evp.h 中，如下所示：

21.2.1 EVP_PKEY

```
struct evp_pkey_st
{
    int references;
    union
    {
        char *ptr;
#ifndef OPENSSL_NO_RSA
        struct rsa_st *rsa;    /* RSA */
#endif
#ifndef OPENSSL_NO_DSA
        struct dsa_st *dsa;   /* DSA */
#endif
#ifndef OPENSSL_NO_DH
```

```
                struct dh_st *dh;     /* DH */
#endif
#ifndef OPENSSL_NO_EC
                struct ec_key_st *ec;      /* ECC */
#endif
            } pkey;
        STACK_OF(X509_ATTRIBUTE) *attributes; /* [ 0 ] */
};
```

该结构用来存放非对称密钥信息，可以是 RSA、DSA、DH 或 ECC 密钥。其中，ptr 用来存放密钥结构地址，attributes 堆栈用来存放密钥属性。

21.2.2 EVP_MD

```
struct env_md_st
{
    int type;
    int pkey_type;
    int md_size;
    unsigned long flags;
    int (*init)(EVP_MD_CTX *ctx);
    int (*update)(EVP_MD_CTX *ctx,const void *data,size_t count);
    int (*final)(EVP_MD_CTX *ctx,unsigned char *md);
    int (*copy)(EVP_MD_CTX *to,const EVP_MD_CTX *from);
    int (*cleanup)(EVP_MD_CTX *ctx);
    int (*sign)(int type, const unsigned char *m, unsigned int m_length,
          unsigned char *sigret, unsigned int *siglen, void *key);
    int (*verify)(int type, const unsigned char *m, unsigned int m_length,
            const unsigned char *sigbuf, unsigned int siglen,
            void *key);
  int required_pkey_type[5];
  int block_size;
  int ctx_size; /* how big does the ctx->md_data need to be */
  } ;
```

该结构用来存放摘要算法信息、非对称算法类型以及各种计算函数。主要各项意义如下：

type：摘要类型，一般是摘要算法 NID；

pkey_type：公钥类型，一般是签名算法 NID；

md_size：摘要值大小，为字节数；

flags：用于设置标记；

init：摘要算法初始化函数；

update：多次摘要函数；

final：摘要完结函数；

copy：摘要上下文结构复制函数；

cleanup：清除摘要上下文函数；

sign：签名函数，其中 key 为非对称密钥结构地址；
verify：摘要函数，其中 key 为非对称密钥结构地址。
openssl 对于各种摘要算法实现了上述结构，各个源码位于 cypto/evp 目录下，文件名以 m_开头。Openssl 通过这些结构来封装了各个摘要相关的运算。

21.2.3 EVP_CIPHER

```
struct evp_cipher_st
    {
    int nid;
    int block_size;
    int key_len;
    int iv_len;
    unsigned long flags;
    int (*init)(EVP_CIPHER_CTX *ctx, const unsigned char *key,
            const unsigned char *iv, int enc);
    int (*do_cipher)(EVP_CIPHER_CTX *ctx, unsigned char *out,
             const unsigned char *in, unsigned int inl);
    int (*cleanup)(EVP_CIPHER_CTX *); /* cleanup ctx */
    int ctx_size;
    int (*set_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
    int (*get_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
    int (*ctrl)(EVP_CIPHER_CTX *, int type, int arg, void *ptr);
    void *app_data;
    } ;
```

该结构用来存放对称加密相关的信息以及算法。主要各项意义如下：
nid：对称算法 nid；
block_size：对称算法每次加解密的字节数；
key_len：对称算法的密钥长度字节数；
iv_len：对称算法的填充长度；
flags：用于标记；
init：加密初始化函数，用来初始化 ctx，key 为对称密钥值，iv 为初始化向量，enc 用于指明是要加密还是解密，这些信息存放在 ctx 中；
do_cipher：对称运算函数，用于加密或解密；
cleanup：清除上下文函数；
set_asn1_parameters：设置上下文参数函数；
get_asn1_parameters：获取上下文参数函数；
ctrl：控制函数；
app_data：用于存放应用数据。
openssl 对于各种对称算法实现了上述结构，各个源码位于 cypto/evp 目录下，文件名以 e_开头。Openssl 通过这些结构来封装了对称算法相关的运算。

21.2.4 EVP_CIPHER_CTX

```
struct evp_cipher_ctx_st
{
    const EVP_CIPHER *cipher;
    ENGINE *engine;
    int encrypt;
    int buf_len;
    unsigned char  oiv[EVP_MAX_IV_LENGTH];
    unsigned char  iv[EVP_MAX_IV_LENGTH];
    unsigned char      buf[EVP_MAX_BLOCK_LENGTH];

  /* 其他 */
  unsigned char final[EVP_MAX_BLOCK_LENGTH];
  } ;
```

对称算法上下文结构，此结构主要用来维护加解密状态，存放中间以及最后结果。因为加密或解密时，当数据很多时，可能会用到 Update 函数，并且每次加密或解密的输入数据长度任意的，并不一定是对称算法 block_size 的整数倍，所以需要用该结构来存放中间未加密的数据。主要项意义如下：

cipher：指明对称算法；

engine：硬件引擎；

encrypt：是加密还是解密；非 0 为加密，0 为解密；

buf 和 buf_len：指明还有多少数据未进行运算；

oiv：原始初始化向量；

iv：当前的初始化向量；

final：存放最终结果，一般与 Final 函数对应。

21.3 源码结构

evp 源码位于 crypto/evp 目录，可以分为如下几类：

1) 全局函数

主要包括 c_allc.c、c_alld.c、c_all.c 以及 names.c。他们加载 openssl 支持的所有的对称算法和摘要算法，放入到哈希表中。实现了 OpenSSL_add_all_digests、OpenSSL_add_all_ciphers 以及 OpenSSL_add_all_algorithms(调用了前两个函数)函数。在进行计算时，用户也可以单独加载摘要函数（EVP_add_digest）和对称计算函数（EVP_add_cipher）。

2) BIO 扩充

包括 bio_b64.c、bio_enc.c、bio_md.c 和 bio_ok.c，各自实现了 BIO_METHOD 方法，分别用于 base64 编解码、对称加解密以及摘要。

3) 摘要算法 EVP 封装

由 digest.c 实现，实现过程中调用了对应摘要算法的回调函数。各个摘要算法提供了自己的 EVP_MD 静态结构，对应源码为 m_xxx.c。

4） 对称算法 EVP 封装

由 evp_enc.c 实现，实现过程调用了具体对称算法函数，实现了 Update 操作。各种对称算法都提供了一个 EVP_CIPHER 静态结构，对应源码为 e_xxx.c。需要注意的是，e_xxx.c 中不提供完整的加解密运算，它只提供基本的对于一个 block_size 数据的计算，完整的计算由 evp_enc.c 来实现。当用户想添加一个自己的对称算法时，可以参考 e_xxx.c 的实现方式。一般用户至少需要实现如下功能：

- 构造一个新的静态的 EVP_CIPHER 结构；
- 实现 EVP_CIPHER 结构中的 init 函数，该函数用于设置 iv，设置加解密标记、以及根据外送密钥生成自己的内部密钥；
- 实现 do_cipher 函数，该函数仅对 block_size 字节的数据进行对称运算；
- 实现 cleanup 函数，该函数主要用于清除内存中的密钥信息。

5) 非对称算法 EVP 封装

主要是以 p_开头的文件。其中，p_enc.c 封装了公钥加密；p_dec.c 封装了私钥解密；p_lib.c 实现一些辅助函数；p_sign.c 封装了签名函数；p_verify.c 封装了验签函数；p_seal.c 封装了数字信封；p_open.c 封装了解数字信封。

6） 基于口令的加密

包括 p5_crpt2.c、p5_crpt.c 和 evp_pbe.c。

21.4 摘要函数

典型的摘要函数主要有：

1） EVP_md5

返回 md5 的 EVP_MD。

2) EVP_sha1

返回 sha1 的 EVP_MD。

3) EVP_sha256

返回 sha256 的 EVP_MD。

4） EVP_DigestInit

摘要初使化函数，需要有 EVP_MD 作为输入参数。

5） EVP_DigestUpdate 和 EVP_DigestInit_ex

摘要 Update 函数，用于进行多次摘要。

6） EVP_DigestFinal 和 EVP_DigestFinal_ex

摘要 Final 函数，用户得到最终结果。

7） EVP_Digest

对一个数据进行摘要，它依次调用了上述三个函数。

21.5 对称加解密函数

典型的加解密函数主要有：

1） EVP_CIPHER_CTX_init

初始化对称计算上下文。

2） EVP_CIPHER_CTX_cleanup

清除对称算法上下文数据，它调用用户提供的销毁函数销清除存中的内部密钥以及其他数据。

3） EVP_des_ede3_ecb
返回一个 EVP_CIPHER；

4) EVP_EncryptInit 和 EVP_EncryptInit_ex
加密初始化函数，本函数调用具体算法的 init 回调函数，将外送密钥 key 转换为内部密钥形式，将初始化向量 iv 拷贝到 ctx 结构中。

5） EVP_EncryptUpdate
加密函数，用于多次计算，它调用了具体算法的 do_cipher 回调函数。

6） EVP_EncryptFinal 和 EVP_EncryptFinal_ex
获取加密结果，函数可能涉及填充，它调用了具体算法的 do_cipher 回调函数。

7） EVP_DecryptInit 和 EVP_DecryptInit_ex
解密初始化函数。

8） EVP_DecryptUpdate
解密函数，用于多次计算，它调用了具体算法的 do_cipher 回调函数。

9） EVP_DecryptFinal 和 EVP_DecryptFinal_ex
获取解密结果，函数可能涉及去填充，它调用了具体算法的 do_cipher 回调函数。

10）EVP_BytesToKey
计算密钥函数，它根据算法类型、摘要算法、salt 以及输入数据计算出一个对称密钥和初始化向量 iv。

11）PKCS5_PBE_keyivgen 和 PKCS5_v2_PBE_keyivgen
实现了 PKCS5 基于口令生成密钥和初始化向量的算法。

12）PKCS5_PBE_add
加载所有 openssl 实现的基于口令生成密钥的算法。

13）EVP_PBE_alg_add
添加一个 PBE 算法。

21.6 非对称函数

典型的非对称函数有：

1） EVP_PKEY_encrypt
公钥加密。

2) EVP_PKEY_decrypt
私钥解密。

3) EVP_PKEY_assign
设置 EVP_PKEY 中具体的密钥结构，使它代表该密钥。

4) EVP_PKEY_assign_RSA/ EVP_PKEY_set1_RSA
设置 EVP_PKEY 中的 RSA 密钥结构，使它代表该 RSA 密钥。

5) EVP_PKEY_get1_RSA
获取 EVP_PKEY 的 RSA 密钥结构。

6) EVP_SignFinal
签名操作，输入参数必须有私钥(EVP_PKEY)。

7) EVP_VerifyFinal
验证签名，输入参数必须有公钥(EVP_PKEY)。

8) int EVP_OpenInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,const unsigned

char *ek, int ekl, const unsigned char *iv,EVP_PKEY *priv)

解数字信封初始化操作，type为对称加密算法，ek为密钥密文，ekl为密钥密文长度，iv为填充值，priv为用户私钥。

9) EVP_OpenUpdate

做解密运算。

10) EVP_OpenFinal

做解密运算，解开数字信封。

11) int EVP_SealInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, unsigned char **ek,int *ekl, unsigned char *iv, EVP_PKEY **pubk, int npubk)

type为对称算法，ek数组用来存放多个公钥对密钥加密的结果，ekl用于存放ek数组中每个密钥密文的长度，iv为填充值，pubk数组用来存放多个公钥，npubk为公钥个数，本函数用多个公钥分别加密密钥，并做加密初始化。

12）EVP_SealUpdate

做加密运算。

EVP_SealFinal

做加密运算，制作数字信封。

21.7 BASE64 编解码函数

1) EVP_EncodeInit

BASE64 编码初始化。

2) EVP_EncodeUpdate

BASE64 编码，可多次调用。

3) EVP_EncodeFinal

BASE64 编码，并获取最终结果。

4) EVP_DecodeInit

BASE64 解码初始化。

5) EVP_DecodeUpdate

输入数据长度不能大于 80 字节。BASE64 解码可多次调用，注意，本函数的输入数据不能太长。

6) EVP_DecodeFinal

BASE64 解码，并获取最终结果。

7）EVP_EncodeBlock

BASE64 编码函数，本函数可单独调用。

8）EVP_DecodeBlock

BASE64 解码，本函数可单独调用，对输入数据长度无要求。

21.8 其他函数

1） EVP_add_cipher

将对称算法加入到全局变量，以供调用。

2） EVP_add_digest

将摘要算法加入到全局变量中，以供调用。

3) EVP_CIPHER_CTX_ctrl
对称算法控制函数，它调用了用户实现的 ctrl 回调函数。

4) EVP_CIPHER_CTX_set_key_length
当对称算法密钥长度为可变长时，设置对称算法的密钥长度。

5) EVP_CIPHER_CTX_set_padding
设置对称算法的填充，对称算法有时候会涉及填充。加密分组长度大于一时，用户输入数据不是加密分组的整数倍时，会涉及到填充。填充在最后一个分组来完成，openssl 分组填充时，如果有 n 个填充，则将最后一个分组用 n 来填满。

6) EVP_CIPHER_get_asn1_iv
获取原始 iv，存放在 ASN1_TYPE 结构中。

7) EVP_CIPHER_param_to_asn1
设置对称算法参数，参数存放在 ASN1_TYPE 类型中，它调用用户实现的回调函数 set_asn1_parameters 来实现。

8) EVP_CIPHER_type
获取对称算法的类型。

9） EVP_CipherInit/EVP_CipherInit_ex
对称算法计算(加/解密)初始化函数，_ex 函数多了硬件 enginge 参数，EVP_EncryptInit 和 EVP_DecryptInit 函数也调用本函数。

10) EVP_CipherUpdate
对称计算（加/解密）函数，它调用了 EVP_EncryptUpdate 和 EVP_DecryptUpdate 函数。

11）EVP_CipherFinal/EVP_CipherFinal_ex
对称计算(加/解)函数，调用了 EVP_EncryptFinal（_ex）和 EVP_DecryptFinal(_ex)；本函数主要用来处理最后加密分组，可能会有对称计算。

12）EVP_cleanup
清除加载的各种算法，包括对称算法、摘要算法以及 PBE 算法，并清除这些算法相关的哈希表的内容。

13) EVP_get_cipherbyname
根据字串名字来获取一种对称算法(EVP_CIPHER)，本函数查询对称算法哈希表。

14) EVP_get_digestbyname
根据字串获取摘要算法(EVP_MD)，本函数查询摘要算法哈希表。

15) EVP_get_pw_prompt
获取口令提示信息字符串.

16）int EVP_PBE_CipherInit(ASN1_OBJECT *pbe_obj, const char *pass, int passlen,
ASN1_TYPE *param, EVP_CIPHER_CTX *ctx, int en_de)
PBE 初始化函数。本函数用口令生成对称算法的密钥和初始化向量，并作加/解密初始化操作。本函数再加上后续的 EVP_CipherUpdate 以及 EVP_CipherFinal_ex 构成一个完整的加密过程（可参考 crypto/p12_decr.c 的 PKCS12_pbe_crypt 函数）.

17) EVP_PBE_cleanup
删除所有的 PBE 信息，释放全局堆栈中的信息.

18）EVP_PKEY *EVP_PKCS82PKEY(PKCS8_PRIV_KEY_INFO *p8)
将 PKCS8_PRIV_KEY_INFO(x509.h 中定义)中保存的私钥转换为 EVP_PKEY 结构。

19) EVP_PKEY2PKCS8/EVP_PKEY2PKCS8_broken
将 EVP_PKEY 结构中的私钥转换为 PKCS8_PRIV_KEY_INFO 数据结构存储。

20) EVP_PKEY_bits
非对称密钥大小，为比特数。

21) EVP_PKEY_cmp_parameters
比较非对称密钥的密钥参数，用于 DSA 和 ECC 密钥。

22）EVP_PKEY_copy_parameters
拷贝非对称密钥的密钥参数，用于 DSA 和 ECC 密钥。

23）EVP_PKEY_free
释放非对称密钥数据结构。

24) EVP_PKEY_get1_DH/EVP_PKEY_set1_DH
获取/设置 EVP_PKEY 中的 DH 密钥。

25) EVP_PKEY_get1_DSA/EVP_PKEY_set1_DSA
获取/设置 EVP_PKEY 中的 DSA 密钥。

26）EVP_PKEY_get1_RSA/EVP_PKEY_set1_RSA
获取/设置 EVP_PKEY 中结构中的 RSA 结构密钥。

27) EVP_PKEY_missing_parameters
检查非对称密钥参数是否齐全，用于 DSA 和 ECC 密钥。

28) EVP_PKEY_new
生成一个 EVP_PKEY 结构。

29) EVP_PKEY_size
获取非对称密钥的字节大小。

30) EVP_PKEY_type
获取 EVP_PKEY 中表示的非对称密钥的类型。

31）int EVP_read_pw_string(char *buf,int length,const char *prompt,int verify)
获取用户输入的口令；buf 用来存放用户输入的口令，length 为 buf 长度，prompt 为提示给用户的信息，如果为空，它采用内置的提示信息，verify 为 0 时，不要求验证用户输入的口令，否则回要求用户输入两遍。返回 0 表示成功。

32) EVP_set_pw_prompt
设置内置的提示信息，用于需要用户输入口令的场合。

21.9 对称加密过程

对称加密过程如下：

1） EVP_EncryptInit：
设置 buf_len 为 0，表明临时缓冲区 buf 没有数据。

2） EVP_EncryptUpdate：
ctx 结构中的 buf 缓冲区用于存放上次 EVP_EncryptUpdate 遗留下来的未加密的数据，buf_len 指明其长度。如果 buf_len 为 0，加密的时候先加密输入数据的整数倍，将余下的数据拷贝到 buf 缓冲区。如果 buf_len 不为 0，先加密 buf 里面的数据和输入数据的一部分（凑足一个分组的长度），然后用上面的方法加密，输出结果是加过密的数据。

3） EVP_ EncryptFinal
加密 ctx 的 buf 中余下的数据，如果长度不够一个分组（分组长度不为 1），则

填充，然后再加密，输出结果。

总之，加密大块数据（比如一个大的文件，多出调用 EVP_EncryptUpdate）的结果等效于将所有的数据一次性读入内存进行加密的结果。加密和解密时每次计算的数据块的大小不影响其运算结果。

21.10 编程示例

1）示例 1

```
#include <string.h>
#include <openssl/evp.h>
int    main()
{
    int                    ret,which=1;
    EVP_CIPHER_CTX         ctx;
    const EVP_CIPHER       *cipher;
    unsigned char          key[24],iv[8],in[100],out[108],de[100];
    int                    i,len,inl,outl,total=0;

    for(i=0;i<24;i++)
    {
        memset(&key[i],i,1);
    }
    for(i=0;i<8;i++)
    {
        memset(&iv[i],i,1);
    }
    for(i=0;i<100;i++)
    {
        memset(&in[i],i,1);
    }
    EVP_CIPHER_CTX_init(&ctx);
    printf("please select :\n");
    printf("1: EVP_des_ede3_ofb\n");
    printf("2: EVP_des_ede3_cbc\n");
    scanf("%d",&which);
    if(which==1)
        cipher=EVP_des_ede3_ofb();
    else
        cipher=EVP_des_ede3_cbc();
    ret=EVP_EncryptInit_ex(&ctx,cipher,NULL,key,iv);
    if(ret!=1)
    {
        printf("EVP_EncryptInit_ex err1!\n");
```

```
        return -1;
    }
    inl=50;
    len=0;
    EVP_EncryptUpdate(&ctx,out+len,&outl,in,inl);
    len+=outl;
    EVP_EncryptUpdate(&ctx,out+len,&outl,in+50,inl);
    len+=outl;
    EVP_EncryptFinal_ex(&ctx,out+len,&outl);
    len+=outl;
    printf("加密结果长度：%d\n",len);
    /* 解密 */
    EVP_CIPHER_CTX_cleanup(&ctx);
    EVP_CIPHER_CTX_init(&ctx);
    ret=EVP_DecryptInit_ex(&ctx,cipher,NULL,key,iv);
    if(ret!=1)
    {
        printf("EVP_DecryptInit_ex err1!\n");
        return -1;
    }
    total=0;
    EVP_DecryptUpdate(&ctx,de+total,&outl,out,44);
    total+=outl;
    EVP_DecryptUpdate(&ctx,de+total,&outl,out+44,len-44);
    total+=outl;
    ret=EVP_DecryptFinal_ex(&ctx,de+total,&outl);
    total+=outl;
    if(ret!=1)
    {
        EVP_CIPHER_CTX_cleanup(&ctx);
        printf("EVP_DecryptFinal_ex err\n");
        return -1;
    }
    if((total!=100) || (memcmp(de,in,100)))
    {
        printf("err!\n");
        return -1;
    }
    EVP_CIPHER_CTX_cleanup(&ctx);
    printf("test ok!\n");
    return 0;
}
```

输出结果如下：

```
please select :
```

```
1: EVP_des_ede3_ofb
2: EVP_des_ede3_cbc
1
加密结果长度：100
test ok!
please select :
1: EVP_des_ede3_ofb
2: EVP_des_ede3_cbc
2
加密结果长度：104
test ok!
```

2）示例 2

```
#include <string.h>
#include <openssl/evp.h>
int    main()
{
       int                     cnid,ret,i,msize,mtype;
       int                     mpktype,cbsize,mnid,mbsize;
       const EVP_CIPHER        *type;
       const EVP_MD            *md;
       int                     datal,count,keyl,ivl;
       unsigned char       salt[20],data[100],*key,*iv;
       const       char        *cname,*mname;

       type=EVP_des_ecb();
       cnid=EVP_CIPHER_nid(type);
       cname=EVP_CIPHER_name(type);
       cbsize=EVP_CIPHER_block_size(type);
       printf("encrypto nid : %d\n",cnid);
       printf("encrypto name: %s\n",cname);
       printf("encrypto bock size : %d\n",cbsize);
       md=EVP_md5();
       mtype=EVP_MD_type(md);
       mnid=EVP_MD_nid(md);
       mname=EVP_MD_name(md);
       mpktype=EVP_MD_pkey_type(md);
       msize=EVP_MD_size(md);
       mbsize=EVP_MD_block_size(md);
       printf("md info : \n");
       printf("md type    : %d\n",mtype);
       printf("md nid    : %d\n",mnid);
       printf("md name : %s\n",mname);
       printf("md pkey type : %d\n",mpktype);
       printf("md size : %d\n",msize);
```

```
        printf("md block size : %d\n",mbsize);

        keyl=EVP_CIPHER_key_length(type);
        key=(unsigned char *)malloc(keyl);
        ivl=EVP_CIPHER_iv_length(type);
        iv=(unsigned char *)malloc(ivl);
        for(i=0;i<100;i++)
            memset(&data[i],i,1);
        for(i=0;i<20;i++)
            memset(&salt[i],i,1);
        datal=100;
        count=2;
        ret=EVP_BytesToKey(type,md,salt,data,datal,count,key,iv);
        printf("generate key value: \n");
        for(i=0;i<keyl;i++)
            printf("%x ",*(key+i));
        printf("\n");
        printf("generate iv value: \n");
        for(i=0;i<ivl;i++)
            printf("%x ",*(iv+i));
        printf("\n");
        return 0;
    }
```

EVP_BytesToKey 函数通过 salt 以及 data 数据来生成所需要的 key 和 iv。

输出：

```
encrypto nid : 29
encrypto name: DES-ECB
encrypto bock size : 8
md info :
md type    : 4
md nid    : 4
md name : MD5
md pkey type : 8
md size : 16
md block size : 64
generate key value:
54 0 b1 24 18 42 8d dd
generate iv value:
ba 7d c3 97 a0 c9 e0 70
```

3) 示例 3

```
#include <openssl/evp.h>
#include <openssl/rsa.h>
int  main()
{
```

```
    int                 ret,inlen,outlen=0;
    unsigned long e=RSA_3;
    char            data[100],out[500];
    EVP_MD_CTX          md_ctx,md_ctx2;
    EVP_PKEY        *pkey;
    RSA                 *rkey;
    BIGNUM              *bne;

    /* 待签名数据*/
    strcpy(data,"openssl 编程作者：赵春平");
    inlen=strlen(data);
    /* 生成 RSA 密钥*/
    bne=BN_new();
    ret=BN_set_word(bne,e);
    rkey=RSA_new();
    ret=RSA_generate_key_ex(rkey,1024,bne,NULL);
    if(ret!=1)   goto err;
    pkey=EVP_PKEY_new();
    EVP_PKEY_assign_RSA(pkey,rkey);
    /* 初始化*/
    EVP_MD_CTX_init(&md_ctx);
    ret=EVP_SignInit_ex(&md_ctx,EVP_md5(), NULL);
    if(ret!=1) goto err;
    ret=EVP_SignUpdate(&md_ctx,data,inlen);
    if(ret!=1) goto err;
    ret=EVP_SignFinal(&md_ctx,out,&outlen,pkey);
    /* 验证签名*/
    EVP_MD_CTX_init(&md_ctx2);
    ret=EVP_VerifyInit_ex(&md_ctx2,EVP_md5(), NULL);
    if(ret!=1) goto err;
    ret=EVP_VerifyUpdate(&md_ctx2,data,inlen);
    if(ret!=1) goto err;
    ret=EVP_VerifyFinal(&md_ctx2,out,outlen,pkey);
    if(ret==1)
        printf("验证成功\n");
    else
        printf("验证错误\n");
err:
    RSA_free(rkey);
    BN_free(bne);
    return 0;
}
```

4）示例 4

```
#include <openssl/evp.h>
#include <openssl/rsa.h>
int   main()
{
      int                  ret,ekl[2],npubk,inl,outl,total=0,total2=0;
      unsigned long  e=RSA_3;
      char             *ek[2],iv[8],in[100],out[500],de[500];
      EVP_CIPHER_CTX     ctx,ctx2;
      EVP_CIPHER       *type;
      EVP_PKEY         *pubkey[2];
      RSA                  *rkey;
      BIGNUM                   *bne;

      /*  生成 RSA 密钥*/
      bne=BN_new();
      ret=BN_set_word(bne,e);
      rkey=RSA_new();
      ret=RSA_generate_key_ex(rkey,1024,bne,NULL);
      pubkey[0]=EVP_PKEY_new();
      EVP_PKEY_assign_RSA(pubkey[0],rkey);
      type=EVP_des_cbc();
      npubk=1;
      EVP_CIPHER_CTX_init(&ctx);
      ek[0]=malloc(500);
      ek[1]=malloc(500);
      ret=EVP_SealInit(&ctx,type,ek,ekl,iv,pubkey,1);  /*  只有一个公钥*/
      if(ret!=1) goto err;
      strcpy(in,"openssl  编程");
      inl=strlen(in);
      ret=EVP_SealUpdate(&ctx,out,&outl,in,inl);
      if(ret!=1) goto err;
      total+=outl;
      ret=EVP_SealFinal(&ctx,out+outl,&outl);
      if(ret!=1) goto err;
      total+=outl;

      memset(de,0,500);
      EVP_CIPHER_CTX_init(&ctx2);
      ret=EVP_OpenInit(&ctx2,EVP_des_cbc(),ek[0],ekl[0],iv,pubkey[0]);
      if(ret!=1) goto err;
      ret=EVP_OpenUpdate(&ctx2,de,&outl,out,total);
      total2+=outl;
      ret=EVP_OpenFinal(&ctx2,de+outl,&outl);
      total2+=outl;
```

```
        de[total2]=0;
        printf("%s\n",de);
err:
        free(ek[0]);
        free(ek[1]);
        EVP_PKEY_free(pubkey[0]);
        BN_free(bne);
        getchar();
        return 0;
}
```

输出结果：openssl 编程

参考文献：

[1] http://www.openssl.org/docs/crypto/evp.html#NAME

第二十二章 PEM 格式

22.1 PEM 概述

Openssl 使用 PEM（Privacy Enhanced Mail）格式来存放各种信息，它是 openssl 默认采用的信息存放方式。Openssl 中的 PEM 文件一般包含如下信息：

1） 内容类型

表明本文件存放的是什么信息内容，它的形式为“-------BEGIN XXXX ------”，与结尾的“------END XXXX------”对应。

2) 头信息

表明数据是如果被处理后存放，openssl 中用的最多的是加密信息，比如加密算法以及初始化向量 iv。

3) 信息体

为 BASE64 编码的数据。

举例如下：

-----BEGIN RSA PRIVATE KEY-----

Proc-Type: 4,ENCRYPTED

DEK-Info: DES-EDE3-CBC,9CFD51EC6654FCC3

g2UP/2EvYyhHKAKafwABPrQybsxnepPXQxpP9qkaihV3k0uYJ2Q9qD/nSV2AG9Slqp0HBom
nYS35NSB1bmMb+oGD5vareO7Bt+XZgFv0FINCclTBsFOmZwqs/m95Af+BBkCvNCct+ngM+
UWB2N8jXYnbDMvZGyI3ma+Sfcf3vX7gyPOEXgr5D5NgwwNyu/LtQZvM4k2f7xn7VcAFGm
mtvAXvqVrhEvk55XR0plkc+nOqYXbwLjYMO5LSLFNAtETm9aw0nYMD0Zx+s+8tJdtPq+Ifu
3g9UZkvh2KpEg7he8Z8vaV7lpHiTjmpgkKpx9wKUCHnJq8U3cNcYdRvCWNf4T2jYLSS4kxd
K2p50KjH8xcfWXVkU2CK9NQGlh18TmPueZOkSEHf76KTE9DWKAo7mNmcByTziyofe5qK
htqkYYVBbaCFC0+pKTak4EuLgznt6j87ktuXDXFc+50DnWi1FtQN3LuQH5htl7autzaxCvenfG
QByIh7gxCygBVCJdWca3xE1H0SbRV6LbtjeB/NdCvwgJsRLBXXkjU2TKy/ljsG29xHP2xzlvOt
ATxq1zMMwMKt7kJMFpgSTIbxgUeqzgGbR7VMBmWSF4bBNnGDkOQ0WLJhVq9OMbzpB
zmGJqHn3XjZ2SPXF4xhC7ZhAMxDsFs35P4lPLDH/ycLTcLtUmVZJzvPvzh2r56iTiU28f/rMnH
n1xQ92Cf+62VgECI6CwTotMeM0EfGdCQCiWjeqrzH9qy8+VN3Q2xIlUZj7ibO59YO1A5zVxp
KcQRamwyIy/IYTPr2c2wLfsTZPBt6mD4=

-----END RSA PRIVATE KEY-----

本例是作者生成的一个 RSA 密钥，以 PEM 格式加密存放，采用了 openssl 默认的对称加密算法。其中，“-----BEGIN RSA PRIVATE KEY-----”表明了本文件是一个 RSA 私钥；DES-EDE3-CB 为对称加密算法，9CFD51EC6654FCC3 为对称算法初始化向量 iv。

22.2 openssl 的 PEM 实现

Openssl 的 PEM 模块实现位于 crypto/pem 目录下，并且还依赖于 openssl 的 ASN1 模块。Openssl 支持的 PEM 类型在 crypto/pem/pem.h 中定义如下：

```
#define PEM_STRING_X509_OLD             "X509 CERTIFICATE"
```

```
#define PEM_STRING_X509              "CERTIFICATE"
#define PEM_STRING_X509_PAIR         "CERTIFICATE PAIR"
#define PEM_STRING_X509_TRUSTED      "TRUSTED CERTIFICATE"
#define PEM_STRING_X509_REQ_OLD      "NEW CERTIFICATE REQUEST"
#define PEM_STRING_X509_REQ          "CERTIFICATE REQUEST"
#define PEM_STRING_X509_CRL          "X509 CRL"
#define PEM_STRING_EVP_PKEY          "ANY PRIVATE KEY"
#define PEM_STRING_PUBLIC            "PUBLIC KEY"
#define PEM_STRING_RSA               "RSA PRIVATE KEY"
#define PEM_STRING_RSA_PUBLIC        "RSA PUBLIC KEY"
#define PEM_STRING_DSA               "DSA PRIVATE KEY"
#define PEM_STRING_DSA_PUBLIC        "DSA PUBLIC KEY"
#define PEM_STRING_PKCS7             "PKCS7"
#define PEM_STRING_PKCS8             "ENCRYPTED PRIVATE KEY"
#define PEM_STRING_PKCS8INF          "PRIVATE KEY"
#define PEM_STRING_DHPARAMS          "DH PARAMETERS"
#define PEM_STRING_SSL_SESSION       "SSL SESSION PARAMETERS"
#define PEM_STRING_DSAPARAMS         "DSA PARAMETERS"
#define PEM_STRING_ECDSA_PUBLIC      "ECDSA PUBLIC KEY"
#define PEM_STRING_ECPARAMETERS      "EC PARAMETERS"
#define PEM_STRING_ECPRIVATEKEY      "EC PRIVATE KEY"
```

Openssl 生成 PEM 格式文件的大致过程如下：

1) 将各种数据 DER 编码；
2) 将 1）中的数据进行加密处理（如果需要）；
3) 根据类型以及是否加密，构造 PEM 头；
4) 将 2）中的数据进行 BASE64 编码，放入 PEM 文件。

Openssl 各个类型的 PEM 处理函数主要是 write 和 read 函数。write 函数用于生成 PEM 格式的文件，而 read 函数主要用于读取 PEM 格式的文件。各种类型的调用类似。

22.3 PEM 函数

PEM 函数定义在 crypto/pem.h 中。函数比较简单，主要的函数有：

1) PEM_write_XXXX/PEM_write_bio_XXXX
   将 XXXX 代表的信息类型写入到文件/bio 中。
2) PEM_read_XXXX/PEM_read_bio_XXXX
   从文件/bio 中读取 PEM 的 XXXX 代表类型的信息。
   XXXX 可用代表的有：SSL_SESSION、X509、X509_REQ、X509_AUX、X509_CRL、RSAPrivateKey、RSAPublicKey、DSAPrivateKey、PrivateKey、PKCS7、DHparams、NETSCAPE_CERT_SEQUENCE、PKCS8PrivateKey、DSAPrivateKey、DSA_PUBKEY、DSAparams、ECPKParameters、ECPrivateKey、EC_PUBKEY 等。
3) PEM_ASN1_read/PEM_ASN1_read_bio
   比较底层的 PEM 读取函数，2）中的函数都调用了这两个函数。
4) PEM_ASN1_write/PEM_ASN1_write_bio
   比较底层的 PEM 读取函数，1)中的函数都调用了这两个函数。

5） PEM_read_bio

读取 PEM 文件的各个部分，包括文件类型、头信息以及消息体(base64 解码后的结果）。

6） PEM_get_EVP_CIPHER_INFO

根据头信息获取对称算法，并加载初始化向量 iv。

7) PEM_do_header

根据对称算法，解密数据。

8) PEM_bytes_read_bio

获取 PEM 数据，得到的结果为一个 DER 编码的明文数据，该函数先后调用了 5)、 6）和 7）函数。

22.4 编程示例

1）示例 1

```
#include <openssl/pem.h>
#include <openssl/evp.h>

int   mycb(char *buf,int num,int a,char *key)
{
      if(key)
            strcpy(buf,key);
      else
      {
            if(a==1)
                  printf("请输入加密密码:\n");
            else
                  printf("请输入解密密码:\n");
            scanf("%s",buf);
      }
      return strlen(buf);
}

int   main()
{
      int                  ret;
      BIO                  *out,*in;
      RSA                  *r,*read;
      int                  i,bits=512;
      unsigned long e=RSA_3;
      BIGNUM                     *bne;
      const EVP_CIPHER *enc=NULL;

      bne=BN_new();
      ret=BN_set_word(bne,e);
```

```
    r=RSA_new();
    ret=RSA_generate_key_ex(r,bits,bne,NULL);
    if(ret!=1)
    {
        printf("RSA_generate_key_ex err!\n");
        return -1;
    }
    enc=EVP_des_ede3_ofb();
    out=BIO_new_file("pri.pem","w");
//  ret=PEM_write_bio_RSAPrivateKey(out,r,enc,NULL,0,mycb,"123456");
//  ret=PEM_write_bio_RSAPrivateKey(out,r,enc,NULL,0,NULL,"123456");
    ret=PEM_write_bio_RSAPrivateKey(out,r,enc,NULL,0,mycb,NULL);
    if(ret!=1)
    {
        RSA_free(r);
        BIO_free(out);
        return -1;
    }
    BIO_flush(out);
    BIO_free(out);
    out=BIO_new_file("pub.pem","w");
    ret=PEM_write_bio_RSAPublicKey(out,r);
    if(ret!=1)
    {
        RSA_free(r);
        BIO_free(out);
        return -1;
    }
    BIO_flush(out);
    BIO_free(out);
    OpenSSL_add_all_algorithms();
    in=BIO_new_file("pri.pem","rb");
    read=RSA_new();
//  read=PEM_read_bio_RSAPublicKey(in,&read,NULL,NULL);
//  read=PEM_read_bio_RSAPrivateKey(in,&read,mycb,"123456");
//  read=PEM_read_bio_RSAPrivateKey(in,&read,NULL,"123456");
    read=PEM_read_bio_RSAPrivateKey(in,&read,mycb,NULL);
    if(read->d!=NULL)
        printf("test ok!\n");
    else
        printf("err!\n");
    RSA_free(read);
    BIO_free(in);
    return 0;
```

```
}
```

输出：

```
请输入加密密码:
123456
请输入解密密码:
123456
test ok!
```

本示例生成 RSA 密钥，并将私钥写入成 PMI 格式写入文件；然后再读取。主要需要注意的是回调函数的用法。用户可以采用默认的方式，也可以自己写。采用默认方式时，回调函数设为 NULL，否则设置为用户实现调回调函数地址。另外，最后一个参数如果为空，将需要用户输入口令，否则采用参数所表示的口令。

2）示例 2

```
#include <openssl/pem.h>
#include <openssl/bio.h>
int   main()
{

    BIO                 *bp;
    char            *name=NULL,*header=NULL;
    unsigned char  *data=NULL;
    int                 len,ret,ret2;
    EVP_CIPHER_INFO    cipher;

    OpenSSL_add_all_algorithms();
    bp=BIO_new_file("server2.pem","r");
    while(1)
    {
        ret2=PEM_read_bio(bp,&name,&header,&data,&len);
        if(ret2==0)
            break;
        if(strlen(header)>0)
        {
            ret=PEM_get_EVP_CIPHER_INFO(header,&cipher);
            ret=PEM_do_header(&cipher,data,&len,NULL,NULL);
            if(ret==0)
            {
                printf("PEM_do_header err!\n");
                return -1;
            }
        }
        OPENSSL_free(name);
        OPENSSL_free(header);
        OPENSSL_free(data);
    }
```

```
        printf("test ok.\n");
        BIO_free(bp);
        return 0;
    }
```

说明：

本例 server2.pem 的内容如下：

```
-----BEGIN CERTIFICATE-----
MIIB6TCCAVICAQYwDQYJKoZIhvcNAQEEBQAwWzELMAkGA1UEBhMCQVUxEzARBg
NVBAgTClF1ZWVuc2xhbmQxGjAYBgNVBAoTEUNyeXB0U29mdCBQdHkgTHRkMRswGQ
YDVQQDExJUZXN0IENBICgxMDI0IGJpdCkwHhcNMDAxMDE2MjIzMTAzWhcNMDMwM
TE0MjIzMTAzWjBjMQswCQYDVQQGEwJBVTETMBEGA1UECBMKUXVlZW5zbGFuZDE
aMBgGA1UEChMRQ3J5cHRTb2Z0IFB0eSBMdGQxIzAhBgNVBAMTGlNlcnZlciB0ZXN0IG
NlcnQgKDUxMiBiaXQpMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJ+zw4Qnlf8SMVIP
Fe9GEcStgOY2Ww/dgNdhjeD8ckUJNP5VZkVDTGiXav6ooKXfX3j/7tdkuD8Ey2//Kv7+ue0CA
wEAATANBgkqhkiG9w0BAQQFAAOBgQCT0grFQeZaqYb5EYfk20XixZV4GmyAbXMftG1E
o7qGiMhYzRwGNWxEYojf5PZkYZXvSqZ/ZXHXa4g59jK/rJNnaVGMk+xIX8mxQvlV0n5O9P
Iha5BX5teZnkHKgL8aKKLKW1BK7YTngsfSzzaeame5iKfzitAE+OjGF+PFKbwX8Q==
```

```
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,8FDB648C1260EDDA

CPdURB7aZqM5vgDzZoim/qtoLi5PdrrJol9LrH7CNqJfr9kZfmiexZrE4pV738Hh
UBoidqT8moxzDtuBP54FaVri1IJgbuTZPiNbLn00pVcodHdZrrttrjy1eWLlFmN/QcCRQhIo
Row+f1AhYGhsOhVH+m4fRb8P9KXpPbEDYVcG0R0EQq6ejdmhS0vV+YXGmghBSGH12i3
OfRJXC0TXvazORsT322jiVdEmajND6+DpAtmMmn6JTYm2RKwgFr9vPWv9cRQaMP1yrrBC
tMiSINS4mGieN1sE1IvZLhn+/QDNfS4NxgnMfFjSl26TiNd/m29ZNoeDDXEcc6HXhoS/PiT+zP
Bq7t23hmAroqTVehV9YkFsgr71okOTBwlYMbFJ9goC87HYjJo4t0q9IY53GCuoI1Mont3Wm9I
8QlWh2tRq5uraDlSq7U6Z8fwvC2O+wFF+PhRJrgD+4cBETSQJhj7ZVrjJ8cxCbtGcE/QiZTmmy
Y3sirTlUnIwpKtlfOa9pwBaoL5hKk9ZYa8L1ZCKKMoB6pZw4N9OajVkMUtLiOv3cwIdZk4OI
FSSm+pSfcfUdG45a1IQGLoqvt9svckz1sOUhuu5zDPIQUYrHFn3arqUO0zCPVWPMm9oeYOk
B2WCz/OiNhTFynyX0r+Hd3XeT26lgFLfnCkZlXiW/UQXqXQFSjC5sWd5XJ1+1ZgAdXq0L5q
v/vAIrfryNNZHRFxC8QDDI504OA1AHDkHuH9NO9Ur8U0z7qrsUAf5OnMRUK//QV11En5o/
pWcZKD0SVGS03+FVqMhtTsWKzsil5CLAfMbOWUw+/1k1A==
```

```
-----END RSA PRIVATE KEY-----
```

PEM_read_bio 函数可以循环读取文件中的内容。

PEM_do_header 用于解密数据，之前必须调用函数 OpenSSL_add_all_algorithms。

PEM_do_header 解密后的数据放在 data 中，长度由 len 表示，len 即是输入参数又是输出参数。

name、header 和 data 等用 OPENSSL_free 释放内存。

第二十三章 Engine

23.1 Engine 概述

Openssl 硬件引擎(Engine）能够使用户比较容易地将自己的硬件加入到 openssl 中去，替换其提供的软件算法。一个 Engine 提供了密码计算中各种计算方法的集合，它用于控制 openssl 的各种密码计算。

23.2 Engine 支持的原理

Openssl 中的许多数据结构不仅包含数据本身，还包含各种操作，并且这些操作是可替换的。Openssl 中这些结构集合一般叫做 XXX_METHOD，有 DSO_METHOD、DSA_METHOD、EC_METHOD、ECDH_METHOD、ECDSA_METHOD、DH_METHOD、RAND_METHOD、RSA_METHOD、EVP_CIPHER 和 EVP_MD 等。以 RSA 结构为例(crypto/rsa/rsa.h)，RSA 结构不仅包含了大数 n、e、d 和 p 等等数据项目，还包含一个 RSA_METHOD 回调函数集合。该方法给出了 RSA 各种运算函数。

对于各种数据类型，要进行计算必须至少有一个可用的方法(XXX_METHOD)。因此，openssl 对各种类型都提供了默认的计算方法(软算法)。如果用户实现了自己的 XXX_METHOD，那么就能替换 openssl 提供的方法，各种计算由用户自己控制。硬件 Engine 就是这种原理。根据需要，一个硬件 Engine 可实现自己的 RAND_METHOD、RSA_METHOD、EVP_CIPHER、DSA_METHOD、DH_METHOD、ECDH_METHOD 和 EVP_MD 等，来替换对应软算法的 METHOD。

23.3 Engine 数据结构

Engine 数据结构定义在 crypto/engine/eng_int.h 文件中，是对用户透明的数据结构，如下：

```
struct engine_st
{
    const char *id;
    const char *name;
    const RSA_METHOD *rsa_meth;
    const DSA_METHOD *dsa_meth;
    const DH_METHOD *dh_meth;
    const ECDH_METHOD *ecdh_meth;
    const ECDSA_METHOD *ecdsa_meth;
    const RAND_METHOD *rand_meth;
    const STORE_METHOD *store_meth;
    ENGINE_CIPHERS_PTR ciphers;
    ENGINE_DIGESTS_PTR digests;
    ENGINE_GEN_INT_FUNC_PTRdestroy;
```

```
    ENGINE_GEN_INT_FUNC_PTR init;
    ENGINE_GEN_INT_FUNC_PTR finish;
    ENGINE_CTRL_FUNC_PTR ctrl;
    ENGINE_LOAD_KEY_PTR load_privkey;
    ENGINE_LOAD_KEY_PTR load_pubkey;
    /* 其他项 */
    CRYPTO_EX_DATA ex_data;
    struct engine_st *prev;
    struct engine_st *next;
};
```

本结构包含大量的运算集合函数(包括各种 METHOD)供用户来实现。各项意义如下：

id：Engine 标识；

name：Engine 的名字；

rsa_meth：RSA 方法集合；

dsa_meth：DSA 方法集合；

dh_meth：DH 方法集合；

ecdh_meth：ECDH 方法结合；

ecdsa_meth：ECDSA 方法集合；

rand_meth：随机数方法集合；

store_meth：存储方法集合；

ciphers：对称算法选取函数。硬件一般会支持多种对称算法，该回调函数用来从用户实现的多个对称算法中根据某种条件(一般是算法 nid)来选择其中的一种；

digests：摘要算法选取函数。该回调函数用来从用户实现的多个摘要算法中根据某种条件(一般是算法 nid)来选择其中的一种；

destroy：销毁引擎函数；

init：初始化引擎函数；

finish：完成回调函数；

ctrl：控制函数；

load_privkey：加载私钥函数；

load_pubkey：加载公钥函数；

ex_data：扩展数据结构，可用来存放用户数据；

prev/next：用于构建 Engine 链表，openssl 中的硬件 Engine 可能不止一个。

上述这些函数，用户根据应用的需求来实现其中的一种或多种。

23.4 openssl 的 Engine 源码

Openssl 的 Engine 源码分为四类：

1） 核心实现

在 crypto/engine 目录下，是其核心实现。当同时有多个硬件 Engine 时，openssl 分别为 cipher 对称算法(tb_cipher.c)、dh 算法(tb_dh.c)、digest 摘要算法(tb_digest.c)、dsa 算法(tb_dsa.c)、ecdh 算法(tb_ecdh.c)、ecdsa 算法(tb_ecdsa.c)、rand 随机数算法(tb_rand.c)、rsa 算法(tb_rsa.c)和存储方式(tb_store.c)维护一个哈希表。所有用户实现的硬件 Engine 都注册在这些全局的哈希表中。同时，用户使用的时候，能够指定各种算法默认的硬件 Engine。

2） 内置硬件 Engine
源码位于 engines 目录，实现了一些硬件 Engine。
3) 范例
源码位于 demos/engines 目录下，供用户学习参考。
4） 分散于其他各个运算模块用于支持 Engine
各个运算模块都支持 Engine，当提供了 Engine 时，将会采用 Engine 中的算法。

23.5 Engine 函数

主要函数如下：
1) ENGINE_add
将 Engine 加入全局到链表中。
2) ENGINE_by_id
根据 id 来获取 Engine。
3） ENGINE_cleanup
清除所有 Engine 数据。
4） const EVP_CIPHER *ENGINE_get_cipher(ENGINE *e, int nid)
根据指定的硬件 Engine 以及对称算法的 nid，获取 Engine 实现的对应的 EVP_CIPHER，用于对称计算。
5） ENGINE_get_cipher_engine
根据对称算法 nid 来获取 Engine。
6） ENGINE_get_ciphers/ENGINE_set_ciphers
获取/设置指定 Engine 的对称算法选取函数地址，该函数用于从 Engine 中选择一种对称算法。
7) ENGINE_get_ctrl_function
获取 Engine 的控制函数地址。
8） const DH_METHOD *ENGINE_get_DH(const ENGINE *e)
获取 Engine 的 DH_METHOD。
9） const EVP_MD *ENGINE_get_digest(ENGINE *e, int nid)
根据 Engine 和摘要算法 nid 来获取 Engine 中实现的摘要方法 EVP_MD。
10) ENGINE *ENGINE_get_digest_engine(int nid)
根据摘要算法 nid 来获取 Engine。
11）ENGINE_get_digests/ENGINE_set_digests
获取/设置指定 Engine 的摘要算法选取函数地址，该函数用于从 Engine 中选择一种摘要算法。
12) const DSA_METHOD *ENGINE_get_DSA(const ENGINE *e)
获取 Engine 的 DSA 方法。
13) int ENGINE_register_XXX(ENGINE *e)
注册函数，将某一个 Engine 添加到对应方法的哈希表中。
14) void ENGINE_unregister_XXX(ENGINE *e)
将某一个 Engine 从对应的哈希表中删除。
15) void ENGINE_register_all_XXX(void)
将所有的 Engine 注册到对应方法的哈希表中。
16）ENGINE_set_default_XXXX

设置某 Engine 为对应 XXXX 方法的默认 Engine。

17) ENGINE_get_default_XXXX
获取 XXXX 方法的默认 Engine。

18）ENGINE_load_XXXX
加载某种 Engine。

19) ENGINE_get_RAND/ENGINE_set_RAND
获取/设置 Engine 的随机数方法。

20) ENGINE_get_RSA/ENGINE_set_RSA
获取/设置 Engine 的 RSA 方法。

21) ENGINE_get_first/ENGINE_get_next/ENGINE_get_prev/ENGINE_get_last
Engine 链表操作函数。

22）ENGINE_set_name/ENGINE_get_name
设置/获取 Engine 名字。

23）ENGINE_set_id/ENGINE_get_id
设置/获取 Engine 的 id。

24) int ENGINE_set_default(ENGINE *e, unsigned int flags)
根据 flags 将 e 设置为各种方法的默认 Engine。

25) ENGINE_set_XXX_function
设置 Engine 中 XXX 对应的函数。

26) ENGINE_get_XXX_function
获取 Engine 中 XXX 对应的函数。

27) ENGINE_ctrl
Engine 控制函数。

28) ENGINE_get_ex_data/ENGINE_set_ex_data
获取/设置 Engine 的扩展数据。

29）ENGINE_init/ENGINE_finish
Engine 初始化/结束。

ENGINE_up_ref
给 Engine 增加一个引用。

ENGINE_new/ENGINE_free
生成/释放一个 Engine 数据结构。

ENGINE_register_complete
将给定的 Engine，对于每个方法都注册一遍。

ENGINE_register_all_complete
将所有的 Engine，对于每个方法都注册一遍。

23.6 实现 Engine 示例

以下的示例演示了采用 Engine 机制，来改变 openssl 的各种运算行为。实现的 Engine 方法有：随机数方法、对称算法、摘要算法以及 RSA 运算算法。其中，RSA 计算中，密钥 ID 存放在 Engine 的扩展数据结构中。

```
#include <openssl/rsa.h>
#include <openssl/rand.h>
#include <openssl/engine.h>
```

```
static int hw_get_random_bytes(unsigned char* buf, int num)
{
	int	i;

	printf("call hw_get_random_bytes\n");
	for(i=0;i<num;i++)
		memset(buf++,i,1);
	return 1;
}
/*  生成 RSA 密钥对  */
static	int	genrete_rsa_key(RSA *rsa, int bits, BIGNUM *e, BN_GENCB *cb)
{
	printf("genrete_rsa_key \n");
	return 1;
}
/* RSA 公钥加密  */
int rsa_pub_enc(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding)
{
	printf("call rsa_pub_enc \n");
	return 1;
}
/*RSA 公钥解密  */
int rsa_pub_dec(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding)
{
	printf("call rsa_pub_enc \n");
	return 1;
}
/* RSA 私钥加密  */
int rsa_priv_enc(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding)
{
	char *keyid;

	/*  获取私钥 id */
	keyid=(char *)ENGINE_get_ex_data(rsa->engine,0);
	printf("call rsa_pub_dec \n");
	printf("use key id :%d \n",keyid);
	return 1;
}
/* RSA 私钥解密  */
int rsa_priv_dec(int flen,const unsigned char *from,unsigned char *to,RSA *rsa,int padding)
{
	printf("call rsa_priv_dec \n");
	return 1;
```

```
}
/* RSA 算法  */
RSA_METHOD hw_rsa =
{
        "hw cipher",
        rsa_pub_enc,
        rsa_pub_dec,
        rsa_priv_enc,
        rsa_priv_dec,
        NULL,
        NULL,
        NULL,
        NULL,
        RSA_FLAG_SIGN_VER,
        NULL,
        NULL,
        NULL,
        genrete_rsa_key
        };
/*  随机数方法  */
static RAND_METHOD hw_rand =
        {
        NULL,
        hw_get_random_bytes,
        NULL,
        NULL,
        NULL,
        NULL,
        };
/* Engine 的 id */
static const char *engine_hw_id = "ID_hw";
/* Engine 的名字  */
static const char *engine_hw_name = "hwTest";
static int hw_init(ENGINE *e)
{
        printf("call hw_init\n");
        return 1;
}

static int hw_destroy(ENGINE *e)
{
        printf("call hw_destroy\n");
        return 1;
}
```

```
static int hw_finish(ENGINE *e)
{
	printf("call hw_finish\n");
	return 0;
}

static EVP_PKEY *hw_load_privkey(ENGINE* e, const char* key_id,
		UI_METHOD *ui_method, void *callback_data)
{
	/*  将密钥 id 放在 ENGINE 的扩展数据中  */
	int    index;

	printf("call hw_load_privkey\n");
	index=0;
	ENGINE_set_ex_data(e, index, (char *)key_id);
	return NULL;
}

#define    HW_SET_RSA_PRIVATE_KEY   1
/*  实现自己的控制函数  */
static int hw_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)(void))
{
	switch(cmd)
	{
		case HW_SET_RSA_PRIVATE_KEY:
			hw_load_privkey(e,p,NULL,NULL);
			break;
		default:
			printf("err.\n");
			return -1;
	}
	return 0;
	}

static EVP_PKEY *hw_load_pubkey(ENGINE* e, const char* key_id,
		UI_METHOD *ui_method, void *callback_data)
{
	printf("call hw_load_pubkey\n");
	return NULL;
}

static const ENGINE_CMD_DEFN      hw_cmd_defns[] = {
	{ENGINE_CMD_BASE,
```

```
        "SO_PATH",
        "Specifies the path to the 'hw' shared library",
        ENGINE_CMD_FLAG_STRING},
    {0, NULL, NULL, 0}
    };

static int hw_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key,
                const unsigned char *iv, int enc)
    {
    return 1;
    }

static int hw_cipher_enc(EVP_CIPHER_CTX *ctx, unsigned char *out,
                    const unsigned char *in, unsigned int inl)
    {
    memcpy(out,in,inl);
    return 1;
    }

#include <openssl/objects.h>
/*  定义自己的 des_ecb 硬件算法*/
static const EVP_CIPHER EVP_hw_c=
    {
    NID_des_ecb,
    1,8,0,
    8,
    hw_init_key,
    hw_cipher_enc,
    NULL,
    1,
    NULL,
    NULL,
    NULL,
    NULL
    };

const EVP_CIPHER *EVP_hw_cipher(void)
    {
    return(&EVP_hw_c);
    }

/*  选择对称计算函数  */
static int cipher_nids[] =
    { NID_des_ecb, NID_des_ede3_cbc, 0 };
```

```
static int hw_ciphers(ENGINE *e, const EVP_CIPHER **cipher,      const int **nids, int nid)
{
      if(cipher==NULL)
      {
            *nids = cipher_nids;
            return (sizeof(cipher_nids)-1)/sizeof(cipher_nids[0]);
      }
      switch (nid)
      {
            case NID_des_ecb:
            *cipher = EVP_hw_ciphe()r;
            break;
            //其他对称函数
      }
      return 1;
}
static int init(EVP_MD_CTX *ctx)
{
      printf("call md init\n");
      return 1;
}

static int update(EVP_MD_CTX *ctx,const void *data,size_t count)
{
      printf("call md update\n");
      return 1;
}

static int final(EVP_MD_CTX *ctx,unsigned char *md)
{
      int    i;

      printf("call md final\n");
      for(i=0;i<20;i++)
      memset(md++,i,1);
      return 1;
}

int mySign(int type, const unsigned char *m, unsigned int m_length,
                  unsigned char *sigret, unsigned int *siglen, void *key)
{
      RSA*k;
      int    keyid;
```

```
	k=(RSA *)key;
	/* 获取硬件中的私钥 ID，进行计算 */
	keyid=ENGINE_get_ex_data(k->engine,0);
	printf("call mySign\n");
	printf("use key id is %d\n",keyid);
	return 1;
}

int myVerify(int type, const unsigned char *m, unsigned int m_length,
		const unsigned char *sigbuf, unsigned int siglen,
		void *key)
{
	printf("call myVerify\n");
	return 1;
}

static int digest_nids[] =
	{ NID_sha1, NID_md5, 0 };
/* 实现的 sha1 摘要算法 */
static const EVP_MD hw_newmd=
	{
	NID_sha1,
	NID_sha1WithRSAEncryption,
	SHA_DIGEST_LENGTH,
	0,
	init,
	update,
	final,
	NULL,
	NULL,
	mySign,  /* sign */
	myVerify,/* verify */
	//sizeof(EVP_MD *)+sizeof(SHA_CTX),
	6
	};

static EVP_MD *   EVP_hw_md()
{
	return (&hw_newmd);
}
/* 选择摘要算法的函数 */
static int hw_md(ENGINE *e, const EVP_MD **digest,const int **nids, int nid)
{
	if(digest==NULL)
```

```
    {
        *nids = digest_nids;
        return (sizeof(digest_nids)-1)/sizeof(digest_nids[0]);
    }
    switch (nid)
    {
        case NID_sha1:
            *digest = EVP_hw_md();
            break;
            //其他摘要函数

    }
    return 1;
}
static int bind_helper(ENGINE *e)
{
    int    ret;

    ret=ENGINE_set_id(e, engine_hw_id);
    if(ret!=1)
    {
        printf("ENGINE_set_id failed\n");
        return 0;
    }
    ret=ENGINE_set_name(e, engine_hw_name);
    if(ret!=1)
    {
        printf("ENGINE_set_name failed\n");
        return 0;
    }
    ret=ENGINE_set_RSA(e, &hw_rsa);
    if(ret!=1)
    {
        printf("ENGINE_set_RSA failed\n");
        return 0;
    }
    ret=ENGINE_set_RAND(e, &hw_rand);
    if(ret!=1)
    {
        printf("ENGINE_set_RAND failed\n");
        return 0;
    }
    ret=ENGINE_set_destroy_function(e, hw_destroy);
    if(ret!=1)
```

```
{
	printf("ENGINE_set_destroy_function failed\n");
	return 0;
}
ret=ENGINE_set_init_function(e, hw_init);
if(ret!=1)
{
	printf("ENGINE_set_init_function failed\n");
	return 0;
}
ret=ENGINE_set_finish_function(e, hw_finish);
if(ret!=1)
{
	printf("ENGINE_set_finish_function failed\n");
	return 0;
}
ret=ENGINE_set_ctrl_function(e, hw_ctrl);
if(ret!=1)
{
	printf("ENGINE_set_ctrl_function failed\n");
	return 0;
}
ret=ENGINE_set_load_privkey_function(e, hw_load_privkey);
if(ret!=1)
{
	printf("ENGINE_set_load_privkey_function failed\n");
	return 0;
}
ret=ENGINE_set_load_pubkey_function(e, hw_load_pubkey);
if(ret!=1)
{
	printf("ENGINE_set_load_pubkey_function failed\n");
	return 0;
}
ret=ENGINE_set_cmd_defns(e, hw_cmd_defns);
if(ret!=1)
{
	printf("ENGINE_set_cmd_defns failed\n");
	return 0;
}
ret=ENGINE_set_ciphers(e,hw_ciphers);
if(ret!=1)
{
	printf("ENGINE_set_ciphers failed\n");
```

```
        return 0;
    }
    ret=ENGINE_set_digests(e,hw_md);
    if(ret!=1)
    {
        printf("ENGINE_set_digests failed\n");
        return 0;
    }
    return 1;
    }

static ENGINE *engine_hwcipher(void)
    {
    ENGINE *ret = ENGINE_new();
    if(!ret)
        return NULL;
    if(!bind_helper(ret))
        {
        ENGINE_free(ret);
        return NULL;
        }
    return ret;
    }

void ENGINE_load_hwcipher()
    {
    ENGINE *e_hw = engine_hwcipher();
    if (!e_hw) return;
    ENGINE_add(e_hw);
    ENGINE_free(e_hw);
    ERR_clear_error();
}

#define    HW_set_private_keyID(a)     func(e,a,0,(void *)1,NULL)

#include <openssl/engine.h>
#include <openssl/evp.h>

int    main()
{
    ENGINE        *e;
    RSA_METHOD    *meth;
    int                ret,num=20,i;
    char        buf[20],*name;
```

```
EVP_CIPHER*cipher;
EVP_MD              *md;
EVP_MD_CTX                  mctx,md_ctx;
EVP_CIPHER_CTX ciph_ctx,dciph_ctx;
unsigned char  key[8],iv[8];
unsigned char  in[50],out[100],dd[60];
int                         inl,outl,total,dtotal;
RSA                         *rkey;
RSA_METHOD          *rsa_m;
EVP_PKEY    *ek,*pkey;
ENGINE_CTRL_FUNC_PTR       func;

OpenSSL_add_all_algorithms();
ENGINE_load_hwcipher();

e=ENGINE_by_id("ID_hw");
name = (char *)ENGINE_get_name(e);
printf("engine name :%s \n",name);
/*  随机数生成  */
ret=RAND_set_rand_engine(e);
if(ret!=1)
{
      printf("RAND_set_rand_engine err\n");
      return -1;
}
ret=ENGINE_set_default_RAND(e);
if(ret!=1)
{
      printf("ENGINE_set_default_RAND err\n");
      return -1;
}
ret=RAND_bytes((unsigned char *)buf,num);
/*  对称加密  */
for(i=0;i<8;i++)
      memset(&key[i],i,1);
EVP_CIPHER_CTX_init(&ciph_ctx);
/*  采用 Engine 对称算法  */
cipher=EVP_des_ecb();
ret=EVP_EncryptInit_ex(&ciph_ctx,cipher,e,key,iv);
if(ret!=1)
{
      printf("EVP_EncryptInit_ex err\n");
      return -1;
}
```

```
strcpy((char *)in,"zcpsssssssssssss");
inl=strlen((const char *)in);
total=0;
ret=EVP_EncryptUpdate(&ciph_ctx,out,&outl,in,inl);
if(ret!=1)
{
	printf("EVP_EncryptUpdate err\n");
	return -1;
}
total+=outl;
ret=EVP_EncryptFinal(&ciph_ctx,out+total,&outl);
if(ret!=1)
{
	printf("EVP_EncryptFinal err\n");
	return -1;
}
total+=outl;
/*  解密  */
dtotal=0;
EVP_CIPHER_CTX_init(&dciph_ctx);
ret=EVP_DecryptInit_ex(&dciph_ctx,cipher,e,key,iv);
if(ret!=1)
{
	printf("EVP_DecryptInit_ex err\n");
	return -1;
}
ret=EVP_DecryptUpdate(&dciph_ctx,dd,&outl,out,total);
if(ret!=1)
{
	printf("EVP_DecryptUpdate err\n");
	return -1;
}
dtotal+=outl;
ret=EVP_DecryptFinal(&dciph_ctx,dd+dtotal,&outl);
if(ret!=1)
{
	printf("EVP_DecryptFinal err\n");
	return -1;
}
dtotal+=outl;

/* Engine 摘要  */
EVP_MD_CTX_init(&mctx);
md=EVP_sha1();
```

```
ret=EVP_DigestInit_ex(&mctx,md,e);
if(ret!=1)
{
	printf("EVP_DigestInit_ex err.\n");
	return -1;
}
ret=EVP_DigestUpdate(&mctx,in,inl);
if(ret!=1)
{
	printf("EVP_DigestInit_ex err.\n");
	return -1;
}
ret=EVP_DigestFinal(&mctx,out,(unsigned int *)&outl);
if(ret!=1)
{
	printf("EVP_DigestInit_ex err.\n");
	return -1;
}
func=ENGINE_get_ctrl_function(e);
/*  设置计算私钥 ID */
HW_set_private_keyID(1);
rkey=RSA_new_method(e);
pkey=EVP_PKEY_new();
EVP_PKEY_set1_RSA(pkey,rkey);

EVP_MD_CTX_init(&md_ctx);
ret=EVP_SignInit_ex(&md_ctx,EVP_sha1(),e);
if(ret!=1)
{
	printf("EVP_SignInit_ex err\n");
	return -1;
}
ret=EVP_SignUpdate(&md_ctx,in,inl);
if(ret!=1)
{
	printf("EVP_SignUpdate err\n");
	return -1;
}
ret=EVP_SignFinal(&md_ctx,out,(unsigned int *)&outl,pkey);
if(ret!=1)
{
	printf("EVP_SignFinal err\n");
	return -1;
}
```

```
    /* 私钥加密 */
    RSA_private_encrypt(inl,in,out,rkey,1);
    /* 公钥解密 */
    /* 公钥加密 */
    /* 私钥解密 */
    printf("all test ok.\n");
    ENGINE_free(e);
    ENGINE_finish(e);
    return 0;
}
```

读者可以跟踪调试上述示例来研究各种细节。

第二十四章 通用数据结构

24.1 通用数据结构

本文中的通用数据结构主要指的是证书相关的各个数据结构。它们主要用在数字证书申请、数字证书和 CRL 中。主要包括如下数据结构：

- X509_ALGOR，X509 算法；
- X509_VAL，有效时间；
- X509_PUBKEY，X509 公钥；
- X509_SIG，X509 摘要或者签名值；
- X509_NAME_ENTRY，X509 中的一项名称；
- X509_NAME，X509 名称集合；
- X509_EXTENSION，X509 扩展项；
- X509_ATTRIBUTE，X509 属性；
- GENERAL_NAME，通用名称。

通过 openssl 提供的 ASN1 库，这些数据结构都是可以进行 DER 编解码的。用户主要需要了解它们各项的意义、对它们的编解码以及对它们的 set 和 get 操作。

24.2 X509_ALGOR

该数据结构用来表示算法，它定义在 crypto/x509/x509.h 中，如下：

```
struct X509_algor_st
{
    ASN1_OBJECT *algorithm;
    ASN1_TYPE *parameter;
}
```

包含两项：

algorithm：ASN1_OBJECT 类型，表明了是何种算法；

parameter：ASN1_TYPE 类型，代表该算法需要的参数。ASN1_TYPE 类型可以存放任意数据。

该结构的 DER 编解码接口在 crypto/asn1/x_algor.c 中由 ASN1 宏来实现，其中 parameter 是可选的。该结构相关的函数为：new(生成数据结构)、free(释放数据结构)、i2d(将它转换为 DER 编码)、d2i(由 DER 编码转换为该结构)和 dup(拷贝)。

编程示例如下：

```
#include <string.h>
#include <openssl/x509.h>
int   main()
{
    FILE      *fp;
    char *buf,*p;
    char data[]={"12345678"},read[1024];
```

```
    int          len;
    X509_ALGOR      *alg=NULL,*alg2=NULL,*alg3=NULL;

    /* new 函数 */
    alg=X509_ALGOR_new();
    /* 构造内容 */
    alg->algorithm=OBJ_nid2obj(NID_sha256);
    alg->parameter=ASN1_TYPE_new();
    ASN1_TYPE_set_octetstring(alg->parameter,data,strlen(data));
    /* i2d 函数 */
    len=i2d_X509_ALGOR(alg,NULL);
    p=buf=malloc(len);
    len=i2d_X509_ALGOR(alg,&p);
    /* 写入文件 */
    fp=fopen("alg.cer","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    /* 读文件 */
    fp=fopen("alg.cer","rb");
    len=fread(read,1,1024,fp);
    fclose(fp);
    p=read;
    /* d2i 函数 */
    d2i_X509_ALGOR(&alg2,&p,len);
    if(alg2==NULL)
    {
        printf("err\n");
    }
    /* dup 函数 */
    alg3=X509_ALGOR_dup(alg);
    /* free 函数 */
    X509_ALGOR_free(alg);
    if(alg2)
        X509_ALGOR_free(alg2);
    X509_ALGOR_free(alg3);
    free(buf);
    return 0;
}
```

24.3 X509_VAL

该数据结构用来表示有效时间，定义在 crypto/x509/x509.h 中，如下：

```
typedef struct X509_val_st
{
```

```
    ASN1_TIME *notBefore;
    ASN1_TIME *notAfter;
} X509_VAL;
```

包含两项：

notBefore：生效日期；

notAfter：失效日期；

该结构的 DER 编解码通过宏在 crypto/asn1/x_val.c 中。包括是个函数：new、free、i2d 和 d2i。

编程示例如下：

```
    #include <string.h>
    #include <openssl/x509.h>
    int   main()
    {
        FILE      *fp;
        char *buf,*p;
        char read[1024];
        int         len;
        X509_VAL    *val=NULL,*val2=NULL;
        time_t          t;

        /* new  函数  */
        val=X509_VAL_new();
        /*  构造内容  */
        t=time(0);
        ASN1_TIME_set(val->notBefore,t);
        ASN1_TIME_set(val->notAfter,t+1000);
        /* i2d  函数  */
        len=i2d_X509_VAL(val,NULL);
        p=buf=malloc(len);
        len=i2d_X509_VAL(val,&p);
        /*  写入文件  */
        fp=fopen("val.cer","wb");
        fwrite(buf,1,len,fp);
        fclose(fp);
        /*  读文件  */
        fp=fopen("val.cer","rb");
        len=fread(read,1,1024,fp);
        fclose(fp);
        p=read;
        /* d2i  函数  */
        d2i_X509_VAL(&val2,&p,len);
        if(val2==NULL)
        {
            printf("err\n");
```

```
    }
    /* free 函数 */
    X509_VAL_free(val);
    if(val2)
    X509_VAL_free(val2);
    free(buf);
    return 0;
}
```

24.4 X509_SIG

该结构用来存放摘要或者签名值，定义在 crypto/x509/x509.h 中，如下：

```
typedef struct X509_sig_st
{
    X509_ALGOR *algor;
    ASN1_OCTET_STRING *digest;
} X509_SIG;
```

其中，algor 为算法，digest 用于存放摘要或者签名值。对数据进行签名时，要先对数据摘要，摘要的结果要通过本结构进行 DER 编码，然后才能用私钥进行计算，此时 digest 中存放的就是摘要值。

本结构的 DER 编码通过 ASN1 宏在 crypto/asn1/x_sig.c 中实现，包括 new、free、i2d 和 d2i 函数。

用于签名的摘要 DER 编码示例如下：

```
#include <string.h>
#include <openssl/x509.h>
int   main()
{
    X509_SIG                *sig;
    unsigned char    data[50]={"123456789"};
    unsigned char    dgst[20];
    int                     len;
    unsigned char    *buf,*p;
    FILE                    *fp;

    SHA1(data,strlen(data),dgst);
    sig=X509_SIG_new();
    /* sig->algor->algorithm 不是动态分配的，所有不需要释放
    ASN1_OBJECT_free(sig->algor->algorithm);    */
    sig->algor->algorithm=OBJ_nid2obj(NID_sha1);
    ASN1_OCTET_STRING_set(sig->digest,dgst,20);
    len=i2d_X509_SIG(sig,NULL);
    p=buf=malloc(len);
    len=i2d_X509_SIG(sig,&p);
    fp=fopen("sig.cer","wb");
```

```
    fwrite(buf,1,len,fp);
    fclose(fp);
    free(buf);
    X509_SIG_free(sig);
    return 0;
}
```

24.5 X509_NAME_ENTRY

该数据结构代表了一个名称，数据结构在 crypto/x509/x509.h 中定义如下：

```
typedef struct X509_name_entry_st
{
    ASN1_OBJECT *object;
    ASN1_STRING *value;
    int set;
    int size;
} X509_NAME_ENTRY;
```

每个 X509_NAME_ENTRY 对应于一个证书中的 C、OU 和 O 等实体名称，其中 object 表明了实体的类型是 C 还是 OU 等；value 表明了该实体的内容，这两项用于 DER 编解码。该结构的 DER 编解码在 crypto/asn1/x_name.c 中由宏实现，包括 new、free、i2d、d2i 和 dup 函数。

24.6 X509_NAME

该结构是一个名称集合，在 crypto/x509/x509.h 中定义如下：

```
struct X509_name_st
{
    STACK_OF(X509_NAME_ENTRY) *entries;
    int modified;
#ifndef OPENSSL_NO_BUFFER
    BUF_MEM *bytes;
#else
    char *bytes;
#endif
    unsigned long hash;
    }
```

它主要包含了 X509_NAME_ENTRY 堆栈信息，bytes 用于存放 DER 编码值，hash 为该结构的摘要计算值。该结构的 DER 编解码在 crypto/asn1/x_name.c 中由宏实现。

主要函数：

1) int X509_NAME_add_entry(X509_NAME *name, X509_NAME_ENTRY *ne, int loc, int set)

   将一个 X509_NAME_ENTRY 放入 X509_NAME 的堆栈中，在堆栈中的位置由 loc 指定。

2) int X509_NAME_add_entry_by_NID(X509_NAME *name, int nid, int type,
unsigned char *bytes, int len, int loc, int set)

根据 nid 在 X509_NAME 的 X509_NAME_ENTRY 堆栈中添加一项；bytes 为要添加项的值，type 指明了 types 的 ASN1 类型，loc 为堆栈中的位置；根据 nid 能够获取 ASN1_OBJECT(OBJ_nid2obj 函数)。

3) X509_NAME_add_entry_by_OBJ

与 2）类似，只是要添加的项由 ASN1_OBJECT 来表示。

4）X509_NAME_add_entry_by_txt

与 2）类似，只是要添加的项由字符串来表示，根据 txt 能获取 ASN1_OBJECT(OBJ_txt2obj 函数)。

5） X509_NAME_ENTRY 509_NAME_ENTRY_create_by_NID(X509_NAME_ENTRY **ne, int nid, int type, unsigned char *bytes, int len)

根据 nid 来生成一个 X509_NAME_ENTRY，bytes 为要添加项的值，type 指明了 types 的 ASN1 类型。

6）X509_NAME_ENTRY_create_by_OBJ

与 5）类似，生成的项由 ASN1_OBJECT 来表示。

7）X509_NAME_ENTRY_create_by_txt

与 5）类似，生成的项有字符串来表示。

8）int X509_NAME_get_text_by_NID(X509_NAME *name, int nid, char *buf, int len)

根据 NID 来获取值，结果存放在 buf 中。

9）X509_NAME_get_text_by_OBJ

根据 ASN1_OBJECT 来获取值。

10) int X509_NAME_get_index_by_OBJ(X509_NAME *name,
ASN1_OBJECT *obj, int lastpos)

根据 ASN1_OBJECT 获取 NAME_ENTRY 在堆栈中的位置。

11）X509_NAME_get_index_by_NID

根据 NID 获取 X509_NAME_ENTRY 在堆栈中的位置。

12）X509_NAME_cmp

名字比较。

13）X509_NAME_delete_entry

从堆栈中删除一个指定位置的 X509_NAME_ENTRY，并将它返回。

14）X509_NAME_digest

根据指定的算法，对 X509_NAME 做摘要计算。

15）X509_NAME_dup

名字拷贝。

16）X509_NAME_entry_count

获取 X509_NAME 的 X509_NAME_ENTRY 堆栈中元素个数。

17）X509_NAME_ENTRY_dup

X509_NAME_ENTRY 拷贝。

18）X509_NAME_ENTRY_get/set_data

获取/设置一项名称的值；set 函数还需指明值的 ASN1 类型。

19）X509_NAME_ENTRY_get/set_object

获取/设置一项名称的 ASN1_OBJECT。

20）X509_NAME_get_entry

根据指定堆栈位置获取一个 X509_NAME_ENTRY。

21）X509_NAME_hash

摘要计算，该结果是对 MD5 的结果处理后的值。

22）char *X509_NAME_oneline(X509_NAME *a, char *buf, int len)

将 a 表示的名字变成：/OU=z/CN=的形式放在 buf 中，返回 buf 首地址。

23) X509_NAME_print/ X509_NAME_print_ex

打印 X509_NAME 到 bio 中。

24) X509_NAME_print_ex_fp

打印 X509_NAME 到 FILE 中。

25) int X509_NAME_set(X509_NAME **xn, X509_NAME *name)

通过 dup 函数，设置*xn 的值为 name。

编程示例：

```
#include <string.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
int   main()
{
    X509                *x;
    BIO                 *b,*out;
    int                 ret,len,position,count;
    unsigned int    mdl;
    unsigned char  md[20];
    char            buf[1024],*bufp,bytes[500];
    const     EVP_MD      *type;
    X509_NAME       *xname,*xn;
    unsigned long hv=0;
    FILE                *fp;
    ASN1_OBJECT         *obj;
    X509_NAME_ENTRY  *entry,*c=NULL,*c1;
    ASN1_STRING         *str;

    /* cert.cer 为 PEM 格式的数字证书  */
    b=BIO_new_file("b64cert.cer","r");
    if(b==NULL)
    {
        printf("can not open b64cert.cer!\n");
        return -1;
    }
    x=PEM_read_bio_X509(b,NULL,NULL,NULL);
    /* X509_NAME  函数  */

    /* X509_get_issuer_name,返回指针地址  */
    xname=X509_get_issuer_name(x);
```

```
        /* X509_get_subject_name,返回指针地址 */
        xname=X509_get_subject_name(x);

        /* X509_NAME_hash,将 X509_NAME 数据结构中缓存的 DER 编码值(放在
bytes 中)MD5,其结果再做运算，注意 xname->hash 此时的值无意义 */
        hv=X509_NAME_hash(xname);

        /* X509_NAME_print */
        out=BIO_new(BIO_s_file());
        BIO_set_fp(out,stdout,BIO_NOCLOSE);
        X509_NAME_print(out,xname,0);
        printf("\n");

        /* X509_NAME_print_ex_fp */
        fp=stdout;
        X509_NAME_print_ex_fp(fp,xname,0,XN_FLAG_SEP_MULTILINE);
        printf("\n\n");

        /* X509_NAME_print_ex,XN_FLAG_SEP_MULTILINE表明个值打印时占一行
*/
        X509_NAME_print_ex(out,xname,0,XN_FLAG_SEP_MULTILINE);
        printf("\n");

        /* X509_NAME_get_text_by_NID */
        len=1024;
        ret=X509_NAME_get_text_by_NID(xname,NID_commonName,buf,len);
        printf("commonName : %s\n\n",buf);

        /* X509_NAME_get_text_by_OBJ */
        len=1024;
        obj=OBJ_nid2obj(NID_commonName);
        memset(buf,0,1024);
        ret=X509_NAME_get_text_by_OBJ(xname,obj,buf,len);
        printf("commonName : %s\n\n",buf);

        /* X509_NAME_get_index_by_NID */
        position=X509_NAME_get_index_by_NID(xname,NID_commonName,-1);
        entry=X509_NAME_get_entry(xname,position);
        printf("entry value : %s\n",entry->value->data);

        /* X509_NAME_ENTRY_get_data */
        str=X509_NAME_ENTRY_get_data(entry);

        /* X509_NAME_ENTRY_get_object */
```

```
obj=X509_NAME_ENTRY_get_object(entry);

/* X509_NAME_entry_count */
count=X509_NAME_entry_count(xname);
/* X509_NAME_get_index_by_OBJ */
len=1024;
memset(buf,0,1024);
position=X509_NAME_get_index_by_OBJ(xname,obj,-1);
entry=X509_NAME_get_entry(xname,position);
printf("entry value : %s\n",entry->value->data);

/* X509_NAME_digest */
type=EVP_sha1();
ret=X509_NAME_digest(x->cert_info->subject,type,md,&mdl);
if(ret!=1)
{
	printf("X509_NAME_digest err.\n");
	BIO_free(b);
	X509_free(x);
	return -1;
}
/* X509_name_cmp */
ret=X509_name_cmp(x->cert_info->subject,x->cert_info->issuer);
if(ret==0)
{
	printf("subject == issuer\n");
}
else
{
	printf("subject != issuer\n");
}
/*    X509_NAME_oneline    */
len=1024;
bufp=X509_NAME_oneline(x->cert_info->subject,buf,len);
if(bufp==NULL)
{
	printf("X509_NAME_oneline err\n");
}
else
{
	printf("%s\n",buf);
}

/*  构造 X509_NAME */
```

```
        xn=X509_NAME_new();
        strcpy(bytes,"openssl");
        len=strlen(bytes);
        /* X509_NAME_add_entry_by_txt */

    ret=X509_NAME_add_entry_by_txt(xn,"commonName",V_ASN1_UTF8STRING,bytes,len
,0,-1);
        if(ret!=1)
        {
            printf("X509_NAME_add_entry_by_txt err.\n");
        }
        /* X509_NAME_add_entry_by_NID */
        strcpy(bytes,"china");
        len=strlen(bytes);

    ret=X509_NAME_add_entry_by_txt(xn,LN_countryName,V_ASN1_UTF8STRING,bytes,le
n,0,-1);
        if(ret!=1)
        {
            printf("X509_NAME_add_entry_by_txt err.\n");
        }

        /* X509_NAME_add_entry_by_OBJ */
        strcpy(bytes,"myou");
        len=strlen(bytes);
        obj=OBJ_nid2obj(NID_organizationalUnitName);

    ret=X509_NAME_add_entry_by_OBJ(xn,obj,V_ASN1_UTF8STRING,bytes,len,0,-1);
        if(ret!=1)
        {
            printf("X509_NAME_add_entry_by_OBJ err.\n");
        }

        /* X509_NAME_ENTRY_create_by_NID */
        strcpy(bytes,"myo");
        len=strlen(bytes);

    c=X509_NAME_ENTRY_create_by_NID(&c,NID_organizationName,V_ASN1_UTF8STR
ING,bytes,len);

        /* X509_NAME_add_entry */
        ret=X509_NAME_add_entry(xn,c,1,-1);
        if(ret!=1)
        {
```

```
        printf("X509_NAME_add_entry_by_OBJ err.\n");
    }
    /*    X509_NAME_ENTRY_set_object */
    obj=OBJ_nid2obj(NID_localityName);
    c1=X509_NAME_ENTRY_new();
    ret=X509_NAME_ENTRY_set_object(c1,obj);
    if(ret!=1)
    {
        printf("X509_NAME_ENTRY_set_object err.\n");
    }
    strcpy(bytes,"mylocal");
    len=strlen(bytes);
    /* X509_NAME_ENTRY_set_data */
    ret=X509_NAME_ENTRY_set_data(c1,V_ASN1_UTF8STRING,bytes,len);
    if(ret!=1)
    {
        printf("X509_NAME_ENTRY_set_data err.\n");
    }
    ret=X509_NAME_add_entry(xn,c,2,-1);
    if(ret!=1)
    {
        printf("X509_NAME_add_entry_by_OBJ err.\n");
    }
    c1=X509_NAME_delete_entry(xn,2);
    /* X509_NAME_set */
    BIO_free(b);
    X509_free(x);
    return 0;
}
```

24.7 X509_EXTENSION

本结构用于存放各种扩展项信息。

1）结构定义

数字证书扩展项，定义在 crypto/x509/x509.h 中，如下：

```
typedef struct X509_extension_st
{
    ASN1_OBJECT *object;
    ASN1_BOOLEAN critical;
    ASN1_OCTET_STRING *value;
} X509_EXTENSION;
```

其中 object 指明是哪种扩展项；critical 指明是否为关键扩展项，为 0xFF 时为关键扩展项，-1 为非关键扩展项；value 为 DER 编码的具体扩展项的值。该结构的 DER 编解码在 crypto/asn1/x_exten.c 中由宏实现，包括 new、free、i2d、d2i 和 dup 函数。扩展项的 DER 编

解码可直接采用 i2d 和 d2i 来完成，也可用采用 openssl 提供的其他函数。

2）通过 X509V3_EXT_METHOD 进行 DER 编解码

Openssl 通过 X509V3_EXT_METHOD 来实现对扩展项的编解码。X509V3_EXT_METHOD 定义在 crypto/x509v3/x509v3.h 中，如下：

```
struct v3_ext_method
{
    int ext_nid;
    int ext_flags;
    ASN1_ITEM_EXP *it;
    X509V3_EXT_NEW ext_new;
    X509V3_EXT_FREE ext_free;
    X509V3_EXT_D2I d2i;
    X509V3_EXT_I2D i2d;
    X509V3_EXT_I2S i2s;
    X509V3_EXT_S2I s2i;
    X509V3_EXT_I2V i2v;
    X509V3_EXT_V2I v2i;
    X509V3_EXT_I2R i2r;
    X509V3_EXT_R2I r2i;
    void *usr_data;
};
typedef struct v3_ext_method X509V3_EXT_METHOD;
```

该结构以 ext_nid 表示是何种扩展项，以 it、d2i 和 i2d 函数来指明来它的 DER 编解码函数。这样，只要知道了 ext_nid，就能够对数据进行 DER 编解码。Openssl 对于每个支持的扩展项都实现了上述数据结构，这些文件都在 crypto/x509v3 目录下：

- v3_akey.c：权威密钥标识，实现了 AUTHORITY_KEYID 的 DER 编解码和 X509V3_EXT_METHOD；
- v3_alt.c：颁发者别名，实现了 GENERAL_NAMES 的 509V3_EXT_METHOD；
- v3_bcons.c：基本约束，实现了 BASIC_CONSTRAINTS 的 DER 编解码和 509V3_EXT_METHOD；
- v3_cpols.c：证书策略，实现了 CERTIFICATEPOLICIES 的 DER 编解码和 509V3_EXT_METHOD；
- v3_crld.c：CRL 发布点，实现了 CRL_DIST_POINTS 的 DER 编解码和 509V3_EXT_METHOD；
- v3_enum.c：证书撤销原因，实现了其 509V3_EXT_METHOD；
- v3_extku.c：扩展密钥用法，实现了 EXTENDED_KEY_USAGE 的 DER 编解码，扩展密钥和 ocsp_accresp 的 509V3_EXT_METHOD；
- v3_info.c：权威信息获取，实现了 AUTHORITY_INFO_ACCESS 的 DER 编解码，v3_info 和 v3_sinfo 两个 509V3_EXT_METHOD；
- v3_int.c：实现了 v3_crl_num、v3_delta_crl 和 v3_inhibit_anyp(继承任何策略)的 509V3_EXT_METHOD；
- v3_ncons.c：名字约束，实现了 NAME_CONSTRAINTS 的 DER 编解码和它的 509V3_EXT_METHOD；
- v3_ocsp.c：实现了 OCSP 相关的多个扩展项的 509V3_EXT_METHOD；

- v3_pci.c：实现了代理证书扩展项的 509V3_EXT_METHOD；
- v3_pcons.c：策略约束，实现了 POLICY_CONSTRAINTS 的 DER 编解码和 509V3_EXT_METHOD；
- v3_pku.c：密钥有效期，实现了 PKEY_USAGE_PERIOD 的 DER 编解码和它的 509V3_EXT_METHOD；
- v3_pmaps.c：策略映射，实现了 POLICY_MAPPINGS 的 DER 编解码和它的 509V3_EXT_METHOD；
- v3_skey.c：主体密钥标识，实现了该扩展项的 509V3_EXT_METHOD；
- v3_sxnet.c：实现了 SXNET 的 DER 编解码和它的 509V3_EXT_METHOD。

openssl 为 509V3_EXT_METHOD 维护了两个表供调用者查找和使用。一个表定义在 crypto/x509v3/ext_dat.h 中，如下：

```
static X509V3_EXT_METHOD *standard_exts[] = {
&v3_nscert,
&v3_ns_ia5_list[0],
&v3_ns_ia5_list[1],
/* 其他 */
&v3_policy_mappings,
&v3_inhibit_anyp
};
```

该表是一个全局表。另外一个表在 crypto/x509v3/v3_lib.c 中，是一个全局的 X509V3_EXT_METHOD 堆栈，定义如下：

```
static STACK_OF(X509V3_EXT_METHOD) *ext_list = NULL;
```

当用户其他扩展的时候，可以实现自己的 X509V3_EXT_METHOD，并调用 X509V3_EXT_add 函数放入堆栈。

当用户根据扩展项的 nid 查找对应的 X509V3_EXT_METHOD 时，首先查找 standard_exts，然后在查找 ext_list。找到后，用户就能根据 X509V3_EXT_METHOD 中的各种方法来处理扩展项(比如，DER 编解码)。

将具体的扩展项数据结构(不是指 X509_EXTENSION 而是一个具体扩展项，比如 NAME_CONSTRAINTS)合成 X509_EXTENSION 时，可以采用如下函数：

```
X509_EXTENSION *X509V3_EXT_i2d(int ext_nid, int crit, void *ext_struc)
```

其中 ext_nid 指明了是那种扩展项，crit 表明是否为关键扩展项，ext_struc 为具体扩展项数据结构地址(比如 NAME_CONSTRAINTS 的地址)，返回值为一个已经构造好的 X509_EXTENSION。该函数首先根据 ext_nid 查表来获取具体扩展项的的 X509V3_EXT_METHOD，然后根据 X509V3_EXT_METHOD 中的 it 或者 i2d 函数将具体扩展项(比如 NAME_CONSTRAINTS)进行 DER 编码，最后再调用 X509_EXTENSION_create_by_NID 来生成一个扩展项并返回。

从 X509_EXTENSION 中提取出具体扩展项的数据结构可以采用如下函数：

```
void *X509V3_EXT_d2i(X509_EXTENSION *ext)
```

该函数首先根据 X509_EXTENSION 来获取是那种扩展项，并查找 X509V3_EXT_METHOD 表，然后根据对应的 d2i 函数解码 X509_EXTENSION-> value 中的 DER 编码数据，生成具体的扩展项数据结构并返回。

上述两个函数是具体扩展项和 X509_EXTENSION 相互转化最基本的函数，很多函数都基于它们。

主要函数：

- X509V3_EXT_add：在扩展 X509V3_EXT_METHOD 表 ext_list 中添加一个方法。
- X509V3_EXT_get_nid：根据 nid 来查找 X509V3_EXT_METHOD。
- X509V3_EXT_get：根据扩展项来查找 X509V3_EXT_METHOD，它调用了 X509V3_EXT_get_nid
- X509V3_EXT_add_alias：添加一个 X509V3_EXT_METHOD，使具有相同方法的 X509V3_EXT_METHOD 有不同的扩展项 nid。
- X509V3_get_d2i：从扩展项堆栈中查找具体的扩展项，并返回具体扩展项数据结构地址。
- X509V3_EXT_print：打印单个扩展项。
- int X509V3_add1_i2d(STACK_OF(X509_EXTENSION) **x, int nid, void *value,int crit, unsigned long flags)。

往扩展项堆栈中添加一个具体的扩展项 value，该具体的扩展项是其数据结构地址，添加扩展项时，根据输入参数 flags 可以处理扩展项冲突。flags 可以的值定义在 x509v3.h 中，如下：

```
#define X509V3_ADD_DEFAULT          0L
#define X509V3_ADD_APPEND       1L
#define X509V3_ADD_REPLACE          2L
#define X509V3_ADD_REPLACE_EXISTING  3L
#define X509V3_ADD_KEEP_EXISTING  4L
#define X509V3_ADD_DELETE       5L
#define X509V3_ADD_SILENT       0x10
```

由于 flags 值的不同，本函数的操作可以有如下情况：

a) 扩展项堆栈中没有 nid 对应的扩展项，此时如果 flags 为 X509V3_ADD_REPLACE_EXISTING 或 X509V3_ADD_DELETE 则报错：无此扩展项；

b) 扩展项堆栈中有 nid 对应的扩展项，如果 flags 为 X509V3_ADD_KEEP_EXISTING，成功返回；如果 flags 是 X509V3_ADD_DEFAULT 报错，表明此扩展项已经存在；如果 flags 是 X509V3_ADD_DELETE，则删除这个扩展项；如果 flags 是 X509V3_ADD_REPLACE_EXISTING，则替换此扩展项。

编程示例 1：

调用函数 X509_EXTENSION_create_by_NID 和 X509_EXTENSION_create_by_OBJ 生成扩展项，并调用 X509_EXTENSION_get_object、X509_EXTENSION_get_data 和 X509_EXTENSION_get_critical 获取扩展项信息；调用 X509_EXTENSION_set_object、X509_EXTENSION_set_critical 和 X509_EXTENSION_set_data 设置扩展项信息。这种构造扩展项的方法是比较烦琐的方法。

```
#include <openssl/x509.h>
#include <openssl/x509v3.h>
#include <openssl/objects.h>
int main()
{
    X509_EXTENSION      *ext=NULL;      /* 必须=NULL */
    ASN1_OCTET_STRING   *data,*data2;
    time_t              t;
    PKEY_USAGE_PERIOD   *period,*period2;
```

```
        int                     len,ret,buflen=100;
        unsigned char       *p,*der,*der2;
        ASN1_OBJECT             *obj=NULL;
        char                buf[100];
        BIO                     *b;

        /* 构造内部数据结构 */
        period=PKEY_USAGE_PERIOD_new();
        t=1;
        /* 从时间 1970 年 1 月 1 日 0 点 0 分 0 秒往后算时间，t=1 表示 1 秒*/
        period->notBefore=ASN1_GENERALIZEDTIME_set(period->notBefore,t);
        t=100;
        period->notAfter=ASN1_GENERALIZEDTIME_set(period->notAfter,t);
        /* der 编码 */
        len=i2d_PKEY_USAGE_PERIOD(period,NULL);
        der=(unsigned char *)malloc(len);
        p=der;
        len=i2d_PKEY_USAGE_PERIOD(period,&p);

        data=ASN1_OCTET_STRING_new();
        ASN1_OCTET_STRING_set(data,der,len);
    #if 1

X509_EXTENSION_create_by_NID(&ext,NID_private_key_usage_period,1,data);
    #else
            obj=OBJ_nid2obj(NID_private_key_usage_period);
            X509_EXTENSION_create_by_OBJ(&ext,obj,1,data);
    #endif
        /* get 函数*/
        obj=X509_EXTENSION_get_object(ext);
        OBJ_obj2txt(buf,buflen,obj,0);
        printf("extions obj : %s\n",buf);
        data=X509_EXTENSION_get_data(ext);
        b=BIO_new(BIO_s_file());
        BIO_set_fp(b,stdout,BIO_NOCLOSE);
        ASN1_STRING_print(b,data);
        ret=X509_EXTENSION_get_critical(ext);
        if(ret==1)
        {
            printf("关键扩展项\n");
        }
        else
        {
            printf("非关键扩展项\n");
```

```
        }
        /* set 函数 */
        ret=X509_EXTENSION_set_object(ext,obj);
        if(ret!=1)
        {
            printf("X509_EXTENSION_set_object err\n");
        }
        ret=X509_EXTENSION_set_critical(ext,0); /* 设置为非关键扩展 */
        if(ret!=1)
        {
            printf("X509_EXTENSION_set_critical err\n");
        }
        period2=PKEY_USAGE_PERIOD_new();
        t=(2006-1970)*365*24*3600;
        period2->notBefore=ASN1_GENERALIZEDTIME_set(period2->notBefore,t);
        t=t+10*365*24*3600;
        period2->notAfter=ASN1_GENERALIZEDTIME_set(period2->notAfter,t);
        /* der 编码 */
        len=i2d_PKEY_USAGE_PERIOD(period2,NULL);
        der2=(unsigned char *)malloc(len);
        p=der2;
        len=i2d_PKEY_USAGE_PERIOD(period2,&p);
        data2=ASN1_OCTET_STRING_new();
        ASN1_OCTET_STRING_set(data2,der2,len);
        ret=X509_EXTENSION_set_data(ext,data2);          /* 设置新的时间段 */
        if(ret!=1)
        {
            printf("X509_EXTENSION_set_data err\n");
        }
        PKEY_USAGE_PERIOD_free(period);
        PKEY_USAGE_PERIOD_free(period2);
        free(der);
        free(der2);
        ASN1_OCTET_STRING_free(data);
        ASN1_OCTET_STRING_free(data2);
        X509_EXTENSION_free(ext);
        return 0;
    }
```

编程示例 2：

通过 X509V3_EXT_METHOD 来构造扩展项，简单。

```
    #include <openssl/x509v3.h>
    int     main()
    {
      X509_EXTENSION          *ext=NULL;
```

```
    STACK_OF(X509_EXTENSION)*exts=NULL;
    time_t                  t;
    PKEY_USAGE_PERIOD     *period;
    int                     ret;

    /* 构造内部数据结构 */
    period=PKEY_USAGE_PERIOD_new();
    t=1;
    /* 从时间 1970 年 1 月 1 日 0 点 0 分 0 秒往后算时间，t=1 表示 1 秒*/
    period->notBefore=ASN1_GENERALIZEDTIME_set(period->notBefore,t);
    t=100;
    period->notAfter=ASN1_GENERALIZEDTIME_set(period->notAfter,t);
    /* 根据具体的扩展项构造一个 X509_EXTENSION */
    ext=X509V3_EXT_i2d(NID_private_key_usage_period,1, period);
    /* 根据具体的扩展项构造一个 X509_EXTENSION 堆栈*/
    ret=X509V3_add1_i2d(&exts,NID_private_key_usage_period,
  period,1,X509V3_ADD_DEFAULT);
    X509_EXTENSION_free(ext);
    sk_X509_EXTENSION_pop_free(exts,X509_EXTENSION_free);
    return 0;
  }
```

24.8 X509_ATTRIBUTE

该数据结构用来存放属性信息，定义在 crypto/x509/x509.h 中，如下：

```
typedef struct x509_attributes_st
{
    ASN1_OBJECT *object;
    int single;
    union
    {
        char        *ptr;
        STACK_OF(ASN1_TYPE) *set;
        ASN1_TYPE  *single;
    } value;
} X509_ATTRIBUTE;
```

X509_ATTRIBUTE 可以存放任意类型的数据，object 指明了属性的类型，single 用于表示 value 的类型是 ASN1_TYPE(为 0)还是 ASN1_TYPE 堆栈(为 1)。ASN1_TYPE 可以存放任意 ASN1 类型数据。

该结构的 DER 编解码在 crypto/asn1/x_attrib.c 中由宏实现，实现了 new、free、i2d、d2i 和 dup 函数。

主要函数：

1) int X509_ATTRIBUTE_count(X509_ATTRIBUTE *attr)
   获取属性中 ASN1_TYPE 的个数。

2）X509_ATTRIBUTE *X509_ATTRIBUTE_create(int nid, int atrtype, void *value)

生成一个属性。id 用来生成 ASN1_OBJECT，指明是哪种属性，atrtype 为 ASN1 类型，value 为值，用来设置 set 堆栈。

3）X509_ATTRIBUTE *X509_ATTRIBUTE_create_by_OBJ(X509_ATTRIBUTE **attr,
const ASN1_OBJECT *obj, int atrtype, const void *data, int len)

生成一个属性，obj 指明了属性类型，atrtype 为 ASN1 类型，data 和 len 指明了需要设置的值。

4）X509_ATTRIBUTE_create_by_NID

同上，属性类型由 nid 指定。

5）X509_ATTRIBUTE_create_by_txt

同上，属性类型由 ASN1_OBJECT 的名字指定。

6）X509_ATTRIBUTE_dup

拷贝函数。

7）ASN1_TYPE *X509_ATTRIBUTE_get0_type(X509_ATTRIBUTE *attr, int idx)

获取属性中由堆栈位置 idx 指定的 ASN1_TYPE，如果属性不是 set 集合则返回 value. single。

8）void *X509_ATTRIBUTE_get0_data(X509_ATTRIBUTE *attr, int idx,
int atrtype, void *data)

由 idx 和 ASN1 类型 atrtype 来获取 value. ptr。

9）X509_ATTRIBUTE_get0_object

获取属性类型信息。

10) X509_ATTRIBUTE_set1_data

设置属性信息。

11) X509_ATTRIBUTE_set1_object

设置属性类别。

12）STACK_OF(X509_ATTRIBUTE) *X509at_add1_attr(
STACK_OF(X509_ATTRIBUTE) **x, X509_ATTRIBUTE *attr)

性堆栈中添加一个属性，返回属性堆栈；如果*x 为 NULL，则生成一个新的堆栈。

13) STACK_OF(X509_ATTRIBUTE) 509at_add1_attr_by_NID(
STACK_OF(X509_ATTRIBUTE) **x,int nid,
int type,const unsigned char *bytes, int len)

往属性堆栈中添加一个属性，添加属性的类型由 nid 指定，bytes 为属性值，len 为其长度，type 指明了 bytes 的 ASN1 类型。

14）X509at_add1_attr_by_OBJ

同上，属性类型由 ASN1_OBJECT 指定。

15）X509at_add1_attr_by_txt

同上，属性类型由属性名指定。

24.9 GENERAL_NAME

本结构用来表示通用名称，它定义在 crypto/x509v3/x509v3.h 中，如下：

```
typedef struct GENERAL_NAME_st
{
```

```
#define GEN_OTHERNAME      0
#define GEN_EMAIL    1
#define GEN_DNS      2
#define GEN_X400     3
#define GEN_DIRNAME     4
#define GEN_EDIPARTY     5
#define GEN_URI      6
#define GEN_IPADD    7
#define GEN_RID      8
int type;
union {
    char *ptr;
    OTHERNAME *otherName;            /*  其他名称  */
    ASN1_IA5STRING *rfc822Name;      /*  符合 rfc822 标准的名称  */
    ASN1_IA5STRING *dNSName;         /* DNS 名称  */
    ASN1_TYPE *x400Address;
    X509_NAME *directoryName;        /*  目录名  */
    EDIPARTYNAME *ediPartyName;
    ASN1_IA5STRING *uniformResourceIdentifier;       /* URI 名称  */
    ASN1_OCTET_STRING *iPAddress;                    /* IP 地址名称  */
    ASN1_OBJECT *registeredID;
    /* Old names */
    ASN1_OCTET_STRING *ip;
    X509_NAME *dirn;
    ASN1_IA5STRING *ia5;
    ASN1_OBJECT *rid; /* registeredID */
    ASN1_TYPE *other; /* x400Address */
} d;
} GENERAL_NAME;
typedef STACK_OF(GENERAL_NAME) GENERAL_NAMES；
```

GENERAL_NAMES 为 GENERAL_NAME 堆栈。

GENERAL_NAME 可以包含各种各样的名称。type 用来表示该结构采用了什么样的名称，从 0~8 分别对应 otherName、rfc822Name、dNSName、x400Address、directoryName、ediPartyName、uniformResourceIdentifier、iPAddress 和 registeredID。ptr 用来存放通用的各种名称的地址。当 type 为某一类型时，数据存放在 d 中对应的项。比如，当 type 为 GEN_DIRNAME 时，数据存放在 directoryName 中。openssl 中最常用的是 X509_NAME。

上述两种结构的 DER 编码在 crypto/x509v3/v3_genn.c 中由宏实现，实现了 new、free、i2d 和 d2i 四个函数。

编程示例：

```
#include <openssl/x509v3.h>
int   main()
{
    GENERAL_NAME      *gn;
```

```
    GENERAL_NAMES    *gns;
    char             *buf,*p;
    int                   len;
    FILE                  *fp;

    gns=sk_GENERAL_NAME_new_null();
    /* new 函数 */
    gn=GENERAL_NAME_new();
    /* 设置 gn 的值为一个 rfc822Name */
    gn->type=GEN_EMAIL;
    gn->d.rfc822Name=ASN1_STRING_new();
    ASN1_STRING_set(gn->d.rfc822Name,"forxy@126.com",13);

    len=i2d_GENERAL_NAME(gn,NULL);
    p=buf=malloc(len);
    len=i2d_GENERAL_NAME(gn,&p);
    /* 生成的 gn1.cer 并不是一个完整的 DER 编码文件，不能用 ASN1view 工具查看 */
    fp=fopen("gn1.cer","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    free(buf);
    sk_GENERAL_NAME_push(gns,gn);

    /* new 函数 */
    gn=GENERAL_NAME_new();
    /* 设置 gn 的值为一个 GEN_DIRNAME */
    gn->type=GEN_DIRNAME;
    gn->d.directoryName=X509_NAME_new();
    X509_NAME_add_entry_by_txt(gn->d.directoryName,LN_commonName,V_ASN1_U
TF8STRING,"forxy",5,0,-1);
    /* i2d 函数 */
    len=i2d_GENERAL_NAME(gn,NULL);
    p=buf=malloc(len);
    len=i2d_GENERAL_NAME(gn,&p);
    /* 生成的 gn2.cer 并不是一个完整的 DER 编码文件，不能用 ASN1view 工具查看 */
    fp=fopen("gn2.cer","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    free(buf);
    sk_GENERAL_NAME_push(gns,gn);

    /* GENERAL_NAMES 的 i2d 函数 */
    len=i2d_GENERAL_NAMES(gns,NULL);
    p=buf=malloc(len);
```

```
    len=i2d_GENERAL_NAMES(gns,&p);
    /* 生成完整的 DER 编码文件 */
    fp=fopen("gns.cer","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    free(buf);
    sk_GENERAL_NAME_pop_free(gns,GENERAL_NAME_free);
    return      0;
}
```

第二十五章 证书申请

25.1 证书申请介绍

生成 X509 数字证书前，一般先由用户提交证书申请文件，然后由 CA 来签发证书。大致过程如下：

1) 用户生成自己的公私钥对；
2) 构造自己的证书申请文件，符合 PKCS#10 标准。该文件主要包括了用户信息、公钥以及一些可选的属性信息，并用自己的私钥给该内容签名；
3) 用户将证书申请文件提交给 CA；
4) CA 验证签名，提取用户信息，并加上其他信息（比如颁发者等信息），用 CA 的私钥签发数字证书；

X509 证书申请的格式标准为 pkcs#10 和 rfc2314。

25.2 数据结构

根据 PKCS#10，openssl 的 X509 数字证书申请结构定义在 crypto/x509.h 中，如下所示，主要由两部分组成：

1）X509_REQ_INFO

```
typedef struct X509_req_info_st
{
    ASN1_ENCODING enc;
    ASN1_INTEGER *version;
    X509_NAME *subject;
    X509_PUBKEY *pubkey;
    STACK_OF(X509_ATTRIBUTE) *attributes;
} X509_REQ_INFO;
```

该结构为证书申请信息主体，其中 version 表示版本，subject 为申请者信息，pubkey 为申请者公钥信息，attributes 为可选的属性信息。该结构的 DER 编码接口在 crytpo/asn1/x_req.c 中由宏实现，实现了 new、free、i2d 和 d2i 函数。

2）X509_REQ

```
typedef struct X509_req_st
{
    X509_REQ_INFO *req_info;
    X509_ALGOR *sig_alg;
    ASN1_BIT_STRING *signature;
    int references;
} X509_REQ;
```

该结构为证书申请信息，req_info 为信息主体，sig_alg 为签名算法，signature 为签名值(申请者对 req_info 的 DER 编码值用自己的私钥签名)。该结构的 DER 编码接口在 crytpo/asn1/x_req.c 中由宏实现，实现了 new、free、i2d 和 d2i 函数。

25.3 主要函数

1） int X509_REQ_add1_attr(X509_REQ *req, X509_ATTRIBUTE *attr)
添加一个属性到 req 的属性堆栈中。

2） int X509_REQ_add1_attr_by_NID(X509_REQ *req,int nid,
int type,const unsigned char *bytes, int len)
添加一个属性到 req 的属性堆栈中，nid 指明了属性类型，bytes 为属性值，len 为其长度，type 为属性值的 ASN1 类型。

3） X509_REQ_add1_attr_by_OBJ
同上，属性类型由 ASN1_OBJECT 指定。

4） X509_REQ_add1_attr_by_txt
同上，属性类型由属性名指定。

5） int X509_REQ_add_extensions_nid(
X509_REQ *req, STACK_OF(X509_EXTENSION) *exts,int nid)
添加一个属性到 req 的属性堆栈中，将 exts 扩展项集合作为一个属性加入，nid 指明了加入的是哪种属性；该函数将 X509_EXTENSION 堆栈 DER 编码，编码后的值作为属性值。

6） X509_REQ_add_extensions
调用了 5)，只是 nid 指定为 NID_ext_req。

7） X509_REQ_delete_attr
从属性堆栈中删除指定位置的属性。

8） X509_REQ_digest
根据指定的摘要算法，对 X509_REQ 结构做摘要计算。

9） X509_REQ_dup
拷贝函数，返回一个 X509_REQ，返回的 X509_REQ 需要调用 X509_REQ_free 释放空间。

10）int X509_REQ_extension_nid(int req_nid)
判断 req_nid 是否为 NID_ext_req、NID_ms_ext_req 或其他由用户设置的 NID，如果是返回 1，否则返回 0。

11）STACK_OF(X509_EXTENSION) *X509_REQ_get_extensions(X509_REQ *req)
获取 X509_REQ 中的属性信息，并将属性信息转换为 X509_EXTENSION 堆栈。该函数从 X509_REQ 的属性堆栈中查找包含合法的 nid 类型的属性(见 X509_REQ_get_extension_nids 函数说明)，如果找到一个，则将属性值通过 DER 解码转换为扩展项堆栈。

12）X509_REQ_get1_email
获取证书申请中申请者的邮件地址信息，信息来自 X509_NAME *subject 和 STACK_OF(X509_ATTRIBUTE) *attributes，返回一个堆栈。

13）X509_REQ_get_attr
根据指定位置，获取属性堆栈中的一个属性。

14）int X509_REQ_get_attr_by_NID(const X509_REQ *req, int nid, int lastpos)
根据属性 nid，从 req 的属性堆栈中查找对应属性，并返回。查找堆栈时，从 lastpos 位置开始查找。

15）X509_REQ_get_attr_by_OBJ

同上，根据 ASN1_OBJECT 来查找属性。

16）X509_REQ_get_attr_count

属性堆栈中属性的个数。

17）X509_REQ_get_extension_nids/ X509_REQ_set_extension_nids

获取证书申请合法扩展项列表，默认情况下，该列表在 x509/x509_req.c 中定义如下：

static int ext_nid_list[] = { NID_ext_req, NID_ms_ext_req, NID_undef}；

static int *ext_nids = ext_nid_list;

本函数返回 ext_nids；

通过 X509_REQ_set_extension_nids 函数，用户可用定义自己的证书申请扩展项列，表，该函数的输入参数是一个 nid 列表。调用 X509_REQ_set_extension_nids 时，将 ext_nids 修改为用户输入参数，不再是默认的 ext_nid_list。

18）X509_REQ_get_pubkey

获取公钥。

19）X509_REQ_print

将证书申请信息输出到 BIO 中。

20）int X509_REQ_print_ex(BIO *bp, X509_REQ *x,

unsigned long nmflags, unsigned long cflag)

将证书申请信息输出到 BIO 中，输出的内容通过 cflag 进行过滤，其值定义在 x509.h 中，如下：

```
#define    X509_FLAG_NO_HEADER        1L
#define    X509_FLAG_NO_VERSION           (1L << 1)
#define    X509_FLAG_NO_SERIAL        (1L << 2)
#define    X509_FLAG_NO_SIGNAME           (1L << 3)
#define    X509_FLAG_NO_ISSUER        (1L << 4)
#define    X509_FLAG_NO_VALIDITY          (1L << 5)
#define    X509_FLAG_NO_SUBJECT           (1L << 6)
#define    X509_FLAG_NO_PUBKEY        (1L << 7)
#define    X509_FLAG_NO_EXTENSIONS        (1L << 8)
#define    X509_FLAG_NO_SIGDUMP           (1L << 9)
#define    X509_FLAG_NO_AUX       (1L << 10)
#define    X509_FLAG_NO_ATTRIBUTES        (1L << 11)
```

21）X509_REQ_print_fp

将证书申请消息输出到 FILE 中。

22）X509_REQ *X509_to_X509_REQ(X509 *x, EVP_PKEY *pkey, const EVP_MD *md)

根据证书信息，申请者私钥以及摘要算法生成证书请求。x 为数字证书，pkey 为申请人的私钥信息，md 为摘要算法，pkey 和 md 用于给证书申请签名。

23）X509 *X509_REQ_to_X509(X509_REQ *r, int days, EVP_PKEY *pkey)

根据 X509_REQ 生成一个数字证书并返回，days 指明其失效期，pkey 为外送私钥，用于签名，返回数字证书。此函数无多大用处，由于没有指明颁发者，生成的数字证书颁发者就是 X509_REQ 中的申请人，并且证书的摘要固定用的是 md5 算法，另外，没有处理证书扩展项。

24）int X509_REQ_set_pubkey(X509_REQ *x, EVP_PKEY *pkey)

设置证书请求的公钥。

25）int X509_REQ_set_subject_name(X509_REQ *x, X509_NAME *name)
设置证书请求的者的名称，此函数调用 X509_NAME_set 函数来实现。
26）int X509_REQ_set_version(X509_REQ *x, long version)
设置证书请求信息的版本，此函数调用 ASN1_INTEGER_set 函数来完成。

25.4 编程示例

25.4.1 生成证书请求文件

```
#include <string.h>
#include <openssl/x509.h>
#include <openssl/rsa.h>

int   main()
{
      X509_REQ          *req;
      int               ret;
      long              version;
      X509_NAME         *name;
      EVP_PKEY          *pkey;
      RSA               *rsa;
      X509_NAME_ENTRY   *entry=NULL;
      char              bytes[100],mdout[20];
      int               len,mdlen;
      int               bits=512;
      unsigned long     e=RSA_3;
      unsigned char     *der,*p;
      FILE              *fp;
      const EVP_MD      *md;
      X509              *x509;
      BIO               *b;
      STACK_OF(X509_EXTENSION) *exts;

      req=X509_REQ_new();
      version=1;
      ret=X509_REQ_set_version(req,version);
      name=X509_NAME_new();
      strcpy(bytes,"openssl");
      len=strlen(bytes);

   entry=X509_NAME_ENTRY_create_by_txt(&entry,"commonName",V_ASN1_UTF8STRI
NG,(unsigned char *)bytes,len);
      X509_NAME_add_entry(name,entry,0,-1);
```

```
        strcpy(bytes,"bj");
        len=strlen(bytes);

    entry=X509_NAME_ENTRY_create_by_txt(&entry,"countryName",V_ASN1_UTF8STRIN
G,bytes,len);
        X509_NAME_add_entry(name,entry,1,-1);

        /* subject name */
        ret=X509_REQ_set_subject_name(req,name);
        /* pub key */
        pkey=EVP_PKEY_new();
        rsa=RSA_generate_key(bits,e,NULL,NULL);
        EVP_PKEY_assign_RSA(pkey,rsa);
        ret=X509_REQ_set_pubkey(req,pkey);
        /* attribute */
        strcpy(bytes,"test");
        len=strlen(bytes);

    ret=X509_REQ_add1_attr_by_txt(req,"organizationName",V_ASN1_UTF8STRING,bytes,le
n);
        strcpy(bytes,"ttt");
        len=strlen(bytes);

    ret=X509_REQ_add1_attr_by_txt(req,"organizationalUnitName",V_ASN1_UTF8STRING,b
ytes,len);
        md=EVP_sha1();
        ret=X509_REQ_digest(req,md,mdout,&mdlen);
        ret=X509_REQ_sign(req,pkey,md);
        if(!ret)
        {
            printf("sign err!\n");
            X509_REQ_free(req);
            return -1;
        }
        /*  写入文件 PEM 格式  */
        b=BIO_new_file("certreq.txt","w");
        PEM_write_bio_X509_REQ(b,req,NULL,NULL);
        BIO_free(b);
        /* DER 编码  */
        len=i2d_X509_REQ(req,NULL);
        der=malloc(len);
        p=der;
        len=i2d_X509_REQ(req,&p);
        OpenSSL_add_all_algorithms();
```

```
    ret=X509_REQ_verify(req,pkey);
    if(ret<0)
    {
        printf("verify err.\n");
    }
    fp=fopen("certreq2.txt","wb");
    fwrite(der,1,len,fp);
    fclose(fp);
    free(der);
    X509_REQ_free(req);
    return 0;
}
```

本例用于生成一个证书请求文件，并测试了 X509_REQ_verify 和 X509_REQ_digest 等函数。

25.4.2 解码证书请求文件

```
#include <openssl/pem.h>
int   main()
{
    BIO                 *in;
    X509_REQ            *req=NULL,**req2=NULL;
    FILE                *fp;
    unsigned char  buf[1024],*p;
    int                 len;

    in=BIO_new_file("certreq.txt","r");
    req=PEM_read_bio_X509_REQ(in,NULL,NULL,NULL);
    if(req==NULL)
    {
        printf("DER 解码错误!\n");
    }
    else
    {
        printf("DER 解码成功!\n");
    }
    fp=fopen("certreq2.txt","r");
    len=fread(buf,1,1024,fp);
    fclose(fp);
    p=buf;
    req2=(X509_REQ **)malloc(sizeof(X509_REQ *));
    d2i_X509_REQ(req2,&p,len);
    if(*req2==NULL)
    {
```

```
        printf("DER 解码错误!\n");
    }
    else
    {
        printf("DER 解码成功!\n");
    }
    X509_REQ_free(*req2);
    free(req2);
    return 0;
}
```

其中 certreq.txt 是 PEM 格式的证书请求文件，certreq2.txt 为 DER 编码格式。

第二十六章 X509 数字证书

26.1 X509 数字证书

数字证书是将用户(或其他实体)身份与公钥绑定的信息载体。一个合法的数字证书不仅要符合 X509 格式规范，还必须有 CA 的签名。用户不仅有自己的数字证书，还必须有对应的私钥。

X509v3 数字证书主要包含的内容有[1]：证书版本、证书序列号、签名算法、颁发者信息、有效时间、持有者信息、公钥信息、颁发者 ID、持有者 ID 和扩展项。

26.2 opessl 实现

openssl 实现了标准的 x509v3 数字证书，其源码在 crypto/x509 和 crypto/x509v3 中。其中 x509 目录实现了数字证书以及证书申请相关的各种函数，包括了 X509 和 X509_REQ 结构的设置、读取、打印和比较；数字证书的验证、摘要；各种公钥的导入导出等功能。x509v3 目录主要实现了数字证书扩展项相关的函数。

26.3 X509 数据结构

该结构定义在 crypto/x509.h 中，如下：

```
typedef struct x509_cinf_st
{
    ASN1_INTEGER *version;                  /* 版本 */
    ASN1_INTEGER *serialNumber;             /* 序列号    */
    X509_ALGOR *signature;                  /* 签名算法 */
    X509_NAME *issuer;                      /* 颁发者    */
    X509_VAL *validity;                     /* 有效时间 */
    X509_NAME *subject;                     /* 持有者    */
    X509_PUBKEY *key;                       /* 公钥 */
    ASN1_BIT_STRING *issuerUID;             /* 颁发者唯一标识    */
    ASN1_BIT_STRING *subjectUID;            /* 持有者唯一标识    */
    STACK_OF(X509_EXTENSION) *extensions;   /* 扩展项    */
} X509_CINF;
```

本结构是数字证书的信息主体；

```
struct x509_st
{
    X509_CINF *cert_info;
    X509_ALGOR *sig_alg;
    ASN1_BIT_STRING *signature;
    int valid;
    int references;
```

```
        char *name;
        CRYPTO_EX_DATA ex_data;
        long ex_pathlen;
        long ex_pcpathlen;
        unsigned long ex_flags;
        unsigned long ex_kusage;
        unsigned long ex_xkusage;
        unsigned long ex_nscert;
        ASN1_OCTET_STRING *skid;
        struct AUTHORITY_KEYID_st *akid;
        X509_POLICY_CACHE *policy_cache;
#ifndef OPENSSL_NO_SHA
        unsigned char sha1_hash[SHA_DIGEST_LENGTH];
#endif
        X509_CERT_AUX *aux;
        };
```

该结构表示了一个完整的数字证书。各项意义如下：

cert_info：证书主体信息；

sig_alg：签名算法；

signature：签名值，存放 CA 对该证书采用 sig_alg 算法签名的结果；

valid：是否是合法证书，1 为合法，0 为未知；

references：引用次数，被引用一次则加一；

name：证书持有者信息，内容形式为/C=CN/O=ourinfo……，该内容在调用 d2i_X509 的过程中，通过回调函数 x509_cb(crypto/asn1/x_x509.c)调用 X509_NAME_oneline 来设置；

ex_data：扩展数据结构，用于存放用户自定义的信息；

扩展项信息，用于证书验证。下面的扩展项信息由 crypto/x509v3/v3_purp.c 中的 x509v3_cache_extensions 函数设置：

ex_pathlen：证书路径长度，对应扩展项为 NID_basic_constraints；

ex_flags：通过“与”计算存放各种标记；

ex_kusage：密钥用法，对应扩展项为 NID_key_usage；

ex_xkusage：扩展密钥用法，对应扩展项为 NID_ext_key_usage；

ex_nscert：Netscape 证书类型，对应扩展项为 NID_netscape_cert_type；

skid：主体密钥标识，对应扩展项为 NID_subject_key_identifier；

akid：颁发者密钥标识，对应扩展项为 NID_authority_key_identifier；

policy_cache：各种策略缓存，crypto/x509v3/pcy_cache.c 中由函数 policy_cache_create 设置，对应的策略为 NID_policy_constraints、NID_certificate_policies、NID_policy_mappings 和 NID_inhibit_any_policy（见 policy_cache_new 和 policy_cache_set 函数）；

sha1_hash：存放证书的 sha1 摘要值；

aux：辅助信息；

上述两个结构的 DER 编解码接口由宏在 crypto/asn1/x_x509.c 中实现，包括各自的 new、free、i2d 和 d2i 函数。

DER 解码编程示例如下：

```
#include <openssl/x509.h>
```

```
int   main()
{
      X509              *x;
      FILE              *fp;
      unsigned char  buf[5000],*p;
      int               len,ret;
      BIO               *b;

      /* cert.cer 为 DER 编码的数字证书
      用户如果是 windows 系统，可以从 IE 中导出一个 x509v3 的数字证书作为解析目
      标
      */
      fp=fopen("cert.cer","rb");
      if(!fp) return -1;
      len=fread(buf,1,5000,fp);
      fclose(fp);

      p=buf;
      x=X509_new();
      d2i_X509(&x,(const unsigned char **)&p,len);
      b=BIO_new(BIO_s_file());
      BIO_set_fp(b,stdout,BIO_NOCLOSE);
      ret=X509_print(b,x);
      BIO_free(b);
      X509_free(x);
      return 0;
}
```

程序输出：

```
Certificate:
Data:
      Version: 3 (0x2)
      Serial Number:
            06:37:6c:00:aa:00:64:8a:11:cf:b8:d4:aa:5c:35:f4
      Signature Algorithm: md5WithRSAEncryption
      Issuer: CN=Root Agency
      Validity
            Not Before: May 28 22:02:59 1996 GMT
            Not After : Dec 31 23:59:59 2039 GMT
      Subject: CN=Root Agency
      Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                  Modulus (512 bit):
                        00:81:55:22:b9:8a:a4:6f:ed:d6:e7:d9:66:0f:55:
```

```
                bc:d7:cd:d5:bc:4e:40:02:21:a2:b1:f7:87:30:85:
                5e:d2:f2:44:b9:dc:9b:75:b6:fb:46:5f:42:b6:9d:
                23:36:0b:de:54:0f:cd:bd:1f:99:2a:10:58:11:cb:
                40:cb:b5:a7:41
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        commonName:
            .GFor Testing Purposes Only Sample Software Publishing Credentials Agency
        2.5.29.1:
            0>.....-...O..a!..dc..0.1.0...U....Root Agency...7l...d.....\5.
Signature Algorithm: md5WithRSAEncryption
    2d:2e:3e:7b:89:42:89:3f:a8:21:17:fa:f0:f5:c3:95:db:62:
    69:5b:c9:dc:c1:b3:fa:f0:c4:6f:6f:64:9a:bd:e7:1b:25:68:
    72:83:67:bd:56:b0:8d:01:bd:2a:f7:cc:4b:bd:87:a5:ba:87:
    20:4c:42:11:41:ad:10:17:3b:8c
```

上述示例解码 DER 编码的数字证书，X509_print 用于打印数字证书信息。

如果需要解码 PEM 格式的证书，如下例：

```
#include <openssl/x509.h>
#include <openssl/pem.h>
int   main()
{
    X509            *x;
    BIO             *b;

    /* cert.cer 为 PEM 格式的数字证书 */
    b=BIO_new_file("b64cert.cer","r");
    x=PEM_read_bio_X509(b,NULL,NULL,NULL);
    BIO_free(b);
    X509_free(x);
    return 0;
}
```

上例得到 X509 数据结构。

26.4 X509_TRUST 与 X509_CERT_AUX

1） X509_TRUST

该结构定义在 crypto/x509v3/x509v3.h 中，如下：

```
typedef struct x509_trust_st
{
    int trust;
    int flags;
    int (*check_trust)(struct x509_trust_st *, X509 *, int);
    char *name;
    int arg1;
```

```
        void *arg2;
    } X509_TRUST;
```

信任检查数据结构，本结构用来检查数字证书是否是受信任的，其主要的函数实现在x509/x509_trs.c 中。其主要项为回调函数 check_trust，该函数用于判断证书是受信任的。

Openssl 在 x509_trs.c 中维护了两个表，标准表和扩展表，用于判断特定 NID 的信任情况。如下：

标准表：

```
static X509_TRUST trstandard[] = {
{X509_TRUST_COMPAT, 0, trust_compat, "compatible", 0, NULL},
{X509_TRUST_SSL_CLIENT, 0, trust_1oidany, "SSL Client", NID_client_auth, NULL},
{X509_TRUST_SSL_SERVER, 0, trust_1oidany, "SSL Server", NID_server_auth, NULL},
{X509_TRUST_EMAIL, 0, trust_1oidany, "S/MIME email", NID_email_protect, NULL},
{X509_TRUST_OBJECT_SIGN, 0, trust_1oidany, "Object Signer", NID_code_sign, NULL},
{X509_TRUST_OCSP_SIGN, 0, trust_1oid, "OCSP responder", NID_OCSP_sign, NULL},
{X509_TRUST_OCSP_REQUEST, 0, trust_1oid, "OCSP request", NID_ad_OCSP, NULL}
};
```

扩展表：

```
static STACK_OF(X509_TRUST) *trtable = NULL;
```

扩展表通过 X509_TRUST_add 函数来添加。当用户需要对某个 NID 做判断时，查找这两个表，然后通过 check_trust 得到结果。

2）X509_CERT_AUX

该结构定义在 x509.h 中，如下：

```
typedef struct x509_cert_aux_st
{
    STACK_OF(ASN1_OBJECT) *trust;
    STACK_OF(ASN1_OBJECT) *reject;
    ASN1_UTF8STRING *alias;
    ASN1_OCTET_STRING *keyid;
    STACK_OF(X509_ALGOR) *other;
} X509_CERT_AUX;
```

该结构是 X509 的一项，用于决定一个证书是否受信任。trust 堆栈中存放了受信任的ASN1_OBJECT，reject 堆栈中存放了应该拒绝的 ASN1_OBJECT。trust 堆栈通过X509_add1_trust_object 函数来存放一个可信的 ASN1_OBJECT，reject 堆栈通过X509_add1_reject_object 来存放一个应该拒绝的 ASN1_OBJECT。这两个堆栈在x509/x509_trs.c 的 obj_trust 函数中使用。obj_trust 函数是默认的 check_trust 函数。

上述两个结构在证书验证中的作用如下：

- 在 X509 结构中构造 X509_CERT_AUX；
- 调用 X509_add1_trust_object 和 X509_add1_reject_object，将受信任和要拒绝的ASN1_OBJECT 添加到 X509_CERT_AUX 的两个堆栈中；
- 验证证书时，如果要验证某个 ASN1_OBJECT 是否受信任，查表找到相应的check_trust，进行计算。如果对应的项在标准表 trstandard 中，除了X509_TRUST_COMPAT(检查证书用途)都会调用 obj_trust 函数。

26.5 X509_PURPOSE

该结构用于检查证书用途，它定义在 x509v3.h 中，如下：

```
typedef struct x509_purpose_st
{
    int purpose;
    int trust;
    int flags;
    int (*check_purpose)(const struct x509_purpose_st *,const X509 *, int);
    char *name;
    char *sname;
    void *usr_data;
} X509_PURPOSE;
```

purpose 为证书用途 ID，check_purpose 为检查证书用途函数。基本的用途 ID 在 x509v3.h 中定义，如下：

```
#define X509_PURPOSE_SSL_CLIENT          1
#define X509_PURPOSE_SSL_SERVER          2
#define X509_PURPOSE_NS_SSL_SERVER       3
#define X509_PURPOSE_SMIME_SIGN          4
#define X509_PURPOSE_SMIME_ENCRYPT       5
#define X509_PURPOSE_CRL_SIGN            6
#define X509_PURPOSE_ANY                 7
#define X509_PURPOSE_OCSP_HELPER         8
```

Openssl 在 x509v3/v3_purp.c 中维护了两个表，用来检查各种证书用途。如下：

标准表：

```
static X509_PURPOSE xstandard[] = {
    {X509_PURPOSE_SSL_CLIENT,          X509_TRUST_SSL_CLIENT,          0,
check_purpose_ssl_client, "SSL client", "sslclient", NULL},
    {X509_PURPOSE_SSL_SERVER,          X509_TRUST_SSL_SERVER,          0,
check_purpose_ssl_server, "SSL server", "sslserver", NULL},
    {X509_PURPOSE_NS_SSL_SERVER,       X509_TRUST_SSL_SERVER,          0,
check_purpose_ns_ssl_server, "Netscape SSL server", "nssslserver", NULL},
    {X509_PURPOSE_SMIME_SIGN,          X509_TRUST_EMAIL,               0,
check_purpose_smime_sign, "S/MIME signing", "smimesign", NULL},
    {X509_PURPOSE_SMIME_ENCRYPT,       X509_TRUST_EMAIL,               0,
check_purpose_smime_encrypt, "S/MIME encryption", "smimeencrypt", NULL},
    {X509_PURPOSE_CRL_SIGN, X509_TRUST_COMPAT, 0, check_purpose_crl_sign,
"CRL signing", "crlsign", NULL},
    {X509_PURPOSE_ANY, X509_TRUST_DEFAULT, 0, no_check, "Any Purpose",
"any", NULL},
    {X509_PURPOSE_OCSP_HELPER, X509_TRUST_COMPAT, 0, ocsp_helper, "OCSP
helper", "ocsphelper", NULL},
};
```

扩展表：

static STACK_OF(X509_PURPOSE) *xptable = NULL;

扩展表由用户通过 X509_PURPOSE_add 函数来添加。

当用户需要检查某个证书用途时，先查表，找到对应的 X509_PURPOSE，然后调用其 check_purpose 函数来判断证书用途是否合法。

检查证书用途的函数为 int X509_check_purpose(X509 *x, int id, int ca)，该函数用于检查证书的用途。x 为待检查待证书，id 为证书用途 NID，ca 表明 x 是否是 ca 证书。

基本用法如下：

```
#include <openssl/x509.h>
#include <openssl/x509v3.h>
int   main()
{
      X509          *x=0;
      int           id,len,ret;
      FILE          *fp;
      unsigned char  buf[5000],*p;

      fp=fopen("root.cer","rb");
      len=fread(buf,1,5000,fp);
      fclose(fp);

      p=buf;
      d2i_X509(&x,&p,len);
      id=X509_PURPOSE_OCSP_HELPER;
      ret=X509_check_purpose(x,id,0);
      if(ret==1)
      {
            printf("purpose check ok!\n");
      }
      else
      {
            printf("purpose check failed!\n");
      }
      X509_free(x);
      return 0;
}
```

如果输入的 id 小于 0，不做任何检查，只是证书的各个扩展项信息写入到将 X509 数据结构中。

另外本函数支持其他用途的验证，示例如下：

```
#include <openssl/x509v3.h>
int   cb_check_tsa(X509_PURPOSE *purpose,const X509 *x,int isCA)
{
      int         flag;
```

```
        printf("------------my check!----------------\n");
        /*   针对 x 添加判断函数  */
        flag=*((int *)(purpose->usr_data));
        if(flag)
            return 1;  /*  由此功能  */
        else
            return 0;  /*  无此功能  */
    }
    int   main()
    {
        X509                *x=0;
        int                 id,len,ret;
        FILE                *fp;
        unsigned char  buf[5000],*p;
        int                 tsaFlag;

        tsaFlag=1;
        ret=X509_PURPOSE_add(1000,1000,0,cb_check_tsa,"tsa","checkTsa",&tsaFlag);
        fp=fopen("root.cer","rb");
        len=fread(buf,1,5000,fp);
        fclose(fp);
        p=buf;
        d2i_X509(&x,&p,len);
        id=1000;
        ret=X509_check_purpose(x,id,0);
        if(ret==1)
        {
            printf("purpose check ok!\n");
        }
        else
        {
            printf("purpose check failed!\n");
        }
        X509_free(x);
        return 0;
    }
```

本程序通过调用函数 X509_PURPOSE_add 添加一个 X509_PURPOSE 内部数据结构，然后再验证证书是否有此用途。用户所要实现的为 X509_PURPOSE 中的回调函数，在此回调函数中，用户根据证书信息来判断证书是否有此用途。如果用户还需要其他的信息才能作出判断，可以另外获取 X509_PURPOSE 数据结构中的 usr_data。usr_data 为一个 void 指针类型。用户可在调用 X509_PURPOSE_add 函数时将它写入对应的 X509_PURPOSE 数据结构(上例中的 tsaFlag)。

26.6 主要函数

1） X509_STORE_add_cert
将证书添加到 X509_STORE 中。

2) X509_STORE_add_crl
将 crl 添加到 X509_STORE 中。

3) void X509_STORE_set_flags(X509_STORE *ctx, long flags)
将 flags 赋值给 ctx 里面的 flags，表明了验证证书时需要验证哪些项。

4） X509_TRUST_set_default
设置默认的 X509_TRUST 检查函数。

5） int X509_verify(X509 *a, EVP_PKEY *r)
验证证书的签名。

6） X509_verify_cert
验证证书，用法可参考 apps/verify.c。

7） X509_verify_cert_error_string
根据错误号，获取错误信息。

8） X509_add1_ext_i2d
根据具体的扩展项数据结构添加一个扩展项。

9） X509_add_ext
X509_EXTENSION 堆栈中，在指定位置添加一项。

10）X509_ALGOR_dup
算法拷贝。

11）X509_alias_get0/X509_alias_set1
获取/设置别名。

12）X509_asn1_meth
获取 X509 的 ASN1_METHOD，包括 new、free、i2d 和 d2i 函数。

13）X509_certificate_type
获取证书和公钥类型。

14）int X509_check_issued(X509 *issuer, X509 *subject);
检查 subject 证书是否由 issuer 颁发，如果是则返回 X509_V_OK，即 0。

15）X509_check_private_key
检查私钥与证书中的公钥是否匹配，匹配返回 1。

16）X509_cmp
证书比较。

17) int X509_cmp_current_time(ASN1_TIME *s)
将 s 与当前时间进行比较，返回值小于 0 则 s 早于当前时间，大于 0 则 s 晚与当前时间。

18）int X509_cmp_time(ASN1_TIME *ctm, time_t *cmp_time)
如果 ctm 时间在 cmp_time 之后，则返回值大于 0。

19) X509_delete_ext
删除扩展项堆栈中指定位置的扩展项。

20）X509_digest
根据指定的摘要算法对 X509 结构做摘要。

20) X509_dup

拷贝函数。

21）X509_find_by_issuer_and_serial

根据颁发者的 X509_NAME 名称和证书序列号，在 X509 堆栈中查找对应的证书并返回。

22) X509_find_by_subject

从证书堆栈中根据持有者名字查询证书，并返回。

23）X509_get0_pubkey_bitstr

获取 X509 结构中的 DER 编码的公钥信息。

24）X509_load_cert_crl_file

加载证书和 crl，用于验证证书。

25）X509_PURPOSE_get0

根据 X509_PURPOSE 的位置获取对应的 X509_PURPOSE。

26）X509_PURPOSE_get0_name

获取 X509_PURPOSE 的名字。

27）X509_PURPOSE_get0_sname

获取 X509_PURPOSE 的别名。

28）X509_PURPOSE_get_by_id

根据证书用途 ID 获取 X509_PURPOSE 在当前数组(xstandard)或堆栈(xptable)中的位置，如果没有返回-1。

29）X509_PURPOSE_get_by_sname

根据别名获取对应的 X509_PURPOSE 在数组或堆栈中的位置。

30）X509_PURPOSE_get_count

获取所有的 X509_PURPOSE 个数，包括标准的和用户动态添加的。

31）X509_PURPOSE_get_id

获取 X509_PURPOSE 的 ID。

32) int X509_PURPOSE_set(int *p, int purpose)

检查是否有 purpose 标识的 X509_PURPOSE，并将 purpose 值写入 p。

33) STACK_OF(X509_EXTENSION) X509v3_add_ext

(STACK_OF(X509_EXTENSION) **x, X509_EXTENSION *ex, int loc)

添加扩展项,堆栈操作，将 ex 表示的扩展项根据 loc 指定的位置插入到 X509_EXTENSION 堆栈中。

34) X509v3_delete_ext

堆栈操作，去除指定位置的扩展项。

35）int X509V3_EXT_print(BIO *out, X509_EXTENSION *ext,

unsigned long flag, int indent)

本函数用于打印单个扩展项，out 为 BIO 类型的输出对象,ext 为扩展项,flag 表明不支持扩展项的处理方式,indent 表明输出时第一列的位置。

flag 的值在 x509v3.h 中定义，可以有：

- #define X509V3_EXT_DEFAULT 0

  打印 DER 编码内容，调用 M_ASN1_OCTET_STRING_print。

- #define X509V3_EXT_ERROR_UNKNOWN (1L << 16)

  打印一行语句。

- #define X509V3_EXT_PARSE_UNKNOWN (2L << 16)

  分析扩展项的 DER 编码，并打印。

- #define X509V3_EXT_DUMP_UNKNOWN (3L << 16)

  打印出 DER 编码的内容，调用 BIO_dump_indent。

36）int X509V3_extensions_print(BIO *bp, char *title,
STACK_OF(X509_EXTENSION) *exts, unsigned long flag, int indent)

本函数将堆栈中的所有扩展项打印，参数意义同上。

37) int X509v3_get_ext_by_critical(const STACK_OF(X509_EXTENSION) *sk, int crit, int lastpos)

获取扩展项在堆栈中的位置，crit 表面扩展项是否关键，lastpos 为指定堆栈搜索起始位置。此函数从给定的 lastpos 开始搜索扩展项堆栈，找到与 crit 匹配的扩展项后，返回其位置，如果找不到扩展项，返回-1。

38）int X509v3_get_ext_by_NID(const STACK_OF(X509_EXTENSION) *x, int nid, int lastpos)

获取扩展项在其堆栈中的位置，此函数根据扩展项标识 nid 以及堆栈搜索的起始进行搜索，如果找到，返回它在堆栈中的位置，如果没找到，返回-1。

39) int X509v3_get_ext_by_OBJ(const STACK_OF(X509_EXTENSION) *sk, ASN1_OBJECT *obj, int lastpos)

功能同上。

40）X509_EXTENSION *X509v3_get_ext(const STACK_OF(X509_EXTENSION) *x, int loc)

获取扩展项，loc 为扩展项在堆栈 x 中的位置，如果不成功，返回 NULL。

41）int X509v3_get_ext_count(const STACK_OF(X509_EXTENSION) *x)

获取扩展项的个数，此函数调用堆栈操作 sk_X509_EXTENSION_num(x)来获取扩展项的个数。

42）STACK_OF(CONF_VALUE) * X509V3_get_section(X509V3_CTX *ctx, char *section)

获取配置信息，section 为配置信息中的“段”信息。比如有配置信息如下：

[CA]

Name1=A

Name2=B

则 section 应是”CA”，返回的信息为它包含的内容信息。

43）char * X509V3_get_string(X509V3_CTX *ctx, char *name, char *section)

根据段和属性获取值，比如有如下配置信息：

[CA]

Name1=A

Name2=B

调用此函数时 name 为”Name1”,sectionwei “CA”，则返回值为”A”。

44）int X509V3_get_value_bool(CONF_VALUE *value, int *asn1_bool)

判断配置信息的布尔值，如果 value 表示的值为 true、TRUE、y、Y、yes、YES，*asn1_bool 的值设为 xff，并返回 1，如果为 false、FALSE、n、N、NO、no, *asn1_bool 设置为 0，并返回 1。此函数调用不成功时返回 0。

45) int X509V3_get_value_int(CONF_VALUE *value, ASN1_INTEGER **aint)

将 value 中的值转换为 ASN1_INTEGER 类型，结果存放在**aint 中，函数调用成功返回 1，否则返回 0。

46）STACK_OF(CONF_VALUE) *X509V3_parse_list(const char *line)

分析配置信息的一行数据，返回结果。

26.7 证书验证

26.7.1 证书验证项

数字证书验证中，主要考察的项有：

i. 有效期，看证书是否已经失效；
ii. 签名，用颁发者的公钥来验证签名；
iii. 证书用途；
iv. 名字比较，证书中的颁发者信息应与颁发者证书的持有者信息一致；
v. 扩展项约束；

26.7.2 Openssl 中的证书验证

Openssl 中的证书验证比较复杂，实现源码在 x509/x509_vfy.c 中，主要有两个函数：X509_verify_cert 和 internal_verify。X509_verify_cert 主要将所有的证书信息进行排序，构造出一个有序的证书链，然后调用 internal_verify 函数来验证证书。internal_verify 是 openssl 提供的一个内置的验证证书链的函数。如果用户通过 X509_STORE_set_verify_func 函数设置了 X509_STORE_CTX 的 verify 函数，将调用用户实现的 verify 函数而不会调用 internal_verify。

如何用 openssl 函数验证证书，用户可以参考 apps/verify.c。

参考文献

[1] rfc2459，Internet X.509 Public Key Infrastructure Certificate and CRL Profile

第二十七章 OCSP

27.1 概述

在线证书状态协议（OCSP，Online Certificate Status Protocol，rfc2560）用于实时表明证书状态。OCSP 客户端通过查询 OCSP 服务来确定一个证书的状态。OCSP 可以通过 HTTP 协议来实现。rfc2560 定义了 OCSP 客户端和服务端的消息格式。

27.2 openssl 实现

openssl 在 crypto/ocsp 目录实现了 ocsp 模块，包括客户端和服务端各种函数。主要源码如下：

- ocsp_asn.c：ocsp 消息的 DER 编解码实现，包括基本的 new、free、i2d 和 d2i 函数；
- ocsp_cl.c：ocsp 客户端函数实现，主要用于生成 ocsp 请求；
- ocsp_srv.c：ocsp 服务端思想，主要用于生成 ocsp 响应；
- ocsp_err.c：ocsp 错误处理；
- ocsp_ext.c：ocsp 扩展项处理；
- ocsp_ht.c：基于 HTTP 协议通信的 OCSP 实现；
- ocsp_lib.c：通用库实现；
- ocsp_prn：打印 OCSP 信息；
- ocsp_vfy：验证 ocsp 请求和响应；
- ocsp.h：定义了 ocsp 请求和响应的各种数据结构和用户接口。

27.3 主要函数

1） d2i_OCSP_REQUEST_bio
将 bio 中的 DER 编码的数据转换为 OCSP_REQUEST 数据结构。

2） d2i_OCSP_RESPONSE_bio
将 bio 中的 DER 编码的数据转换为 OCSP_RESPONSE 数据结构。

3） i2d_OCSP_RESPONSE_bio
将 OCSP_RESPONSE 数据结构 DER 编码，并输出到 BIO 中。

4） i2d_OCSP_REQUEST_bio
将 OCSP_REQUEST 数据结构 DER 编码，并输出到 BIO 中。

5） PEM_read_bio_OCSP_REQUEST
读取 PEM 格式的 OCSP_REQUEST 信息，返回其数据结构。

6） PEM_read_bio_OCSP_RESPONSE
读取 PEM 格式的 OCSP_RESPONSE 信息，返回其数据结构。

7） PEM_write_bio_OCSP_REQUEST
将 OCSP_REQUEST 结构写成 PEM 格式。

8） PEM_write_bio_OCSP_RESPONSE
将 OCSP_RESPONSE 结构写成 PEM 格式。

9） OCSP_REQUEST_sign

本函数由宏来定义，它用于给 OCSP_REQUEST 数据结构签名。签名的对象为 DER 编码的 OCSP_REQINFO 信息，签名算法为 OCSP_SIGNATURE 指定的的算法，签名私钥以及摘要算法由输入参数指定。

10）int OCSP_request_sign(OCSP_REQUEST *req,
X509 *signer,
EVP_PKEY *key,
const EVP_MD *dgst,
STACK_OF(X509) *certs,
unsigned long flags)

本函数用于给 OCSP 请求消息签名，通过 OCSP_REQUEST_sign 函数进行签名，将 signer 持有者信息写入 req，如果 flags 不为 OCSP_NOCERTS，将 certs 信息写入 req。

11）OCSP_BASICRESP_sign

对 OCSP_BASICRESP 结构进行签名，签名结果放在 OCSP_BASICRESP 的 signature 中，摘要算法由输入参数指定。

12）OCSP_REQUEST_verify

验证 ocsp 请求签名，公钥由输入参数指定。

13）OCSP_BASICRESP_verify

验证 ocsp 响应签名，公钥由输入参数指定。

14）OCSP_request_verify

验证 ocsp 响应，该函数做全面的验证，包括签名、证书目的以及证书链等。

15）int OCSP_basic_sign(OCSP_BASICRESP *brsp,X509 *signer, EVP_PKEY *key,
const EVP_MD *dgst,STACK_OF(X509) *certs, unsigned long flags)

本函数用输入参数 signer、key、dgst、certs 和 flags 来填充 brsp 数据结构，并对 brsp 结构签名，成功返回 1，否则返回 0。

16) int OCSP_check_validity(ASN1_GENERALIZEDTIME *thisupd, ASN1_GENERALIZEDTIME *nextupd, long nsec, long maxsec)

时间检查计算，合法返回 1，thisupd 为本次更新时间，nextupd 为下次更新时间。thisupd 和 nextupd 由响应服务生成，他们被传给请求者。请求者收到响应之后需要验证 ocsp 消息的时间有效性。要求如下：

- 本次更新时间不能比当前时间提前太多，提前时间不能大于 nsec，比如 ocsp 服务器多时间比请求者系统时间快很多，导致 thisupd 错误非法；
- 本次更新时间不能晚于当前时间太多，否则 ocsp 消息失效，晚的时间不能大于 maxsec；
- 下次更新时间不能晚于当前时间太多，晚多时间不大于 nsec(由于下一条规则限制，也不能大于 maxsec)；
- 下次更新时间必须大于本次更新时间。

总之，本次更新时间和下次更新时间必须在以当前时间为中心的一个窗口内。

17）OCSP_CERTID_dup

复制函数。

18）OCSP_CERTSTATUS_dup

复制函数。

19）OCSP_ONEREQ *OCSP_request_add0_id(OCSP_REQUEST *req,

OCSP_CERTID *cid)

本函数用于往请求消息中添加一个证书 ID；它将一个 OCSP_CERTID 信息存入 OCSP_REQUEST 结构，返回内部生成的 OCSP_ONEREQ 指针。根据 cid 构造一个 OCSP_ONEREQ 信息，并将此信息放入 req 请求消息的堆栈。

20）int OCSP_request_set1_name(OCSP_REQUEST *req, X509_NAME *nm)

本函数用于设置消息请求者的名字。

21）int OCSP_request_add1_cert(OCSP_REQUEST *req, X509 *cert)

本函数往消息请求中添加一个证书。此证书信息放在 OCSP_REQUEST 结构的一个堆栈中，并将此证书结构的引用加 1。

22）int OCSP_response_status(OCSP_RESPONSE *resp)

本函数获取 OCSP 响应状态。

23）OCSP_BASICRESP *OCSP_response_get1_basic(OCSP_RESPONSE *resp)

本函数从响应数据结构中获取 OCSP_BASICRESP 信息。

24）int OCSP_resp_count(OCSP_BASICRESP *bs)

本函数获取响应消息中包含的证书状态的个数。

25）OCSP_SINGLERESP *OCSP_resp_get0(OCSP_BASICRESP *bs, int idx);

给定单个响应的序号，从堆栈中取出。

26）int OCSP_resp_find(OCSP_BASICRESP *bs, OCSP_CERTID *id, int last)

根据 ocsp 证书 ID 查询对应的响应在堆栈中的位置，last 为搜索堆栈时的起始位置，如果小于 0，从 0 开始。

27）int OCSP_single_get0_status(OCSP_SINGLERESP *single, int *reason,

ASN1_GENERALIZEDTIME **revtime,

ASN1_GENERALIZEDTIME **thisupd,

ASN1_GENERALIZEDTIME **nextupd)

获取单个证书的状态，返回值为其状态，ocsp.h 中定义如下：

#define V_OCSP_CERTSTATUS_GOOD 0

#define V_OCSP_CERTSTATUS_REVOKED 1

#define V_OCSP_CERTSTATUS_UNKNOWN 2

如果证书被撤销，并且 reason 和 revtime 参数不为空，将撤销原因以及撤销时间返回。并且对于这个证书给出 thisUpdate 和 nextUpdate。

28）int OCSP_resp_find_status(OCSP_BASICRESP *bs, OCSP_CERTID *id, int *status,

int *reason,

ASN1_GENERALIZEDTIME **revtime,

ASN1_GENERALIZEDTIME **thisupd,

ASN1_GENERALIZEDTIME **nextupd);

功能同 OCSP_single_get0_status 函数，id 为 OCSP 证书 ID，它依次调用 OCSP_resp_find、OCSP_resp_get0 和OCSP_single_get0_status 函数，其中 status 为返回的证书状态。

29）int OCSP_request_add1_nonce(OCSP_REQUEST *req, unsigned char *val, int len)

添加 nonce 扩展项,val 和 len 表明了 nonce 值,如果 val 为空,则内部生成长度为 len 的随机数作为 nonce。

30）int OCSP_basic_add1_nonce(OCSP_BASICRESP *resp, unsigned char *val, int len)

功能同上。

31）int OCSP_check_nonce(OCSP_REQUEST *req, OCSP_BASICRESP *bs)

检测 nonce，用于防止重放攻击；检查请求和响应的 nonce 扩展项，看他们是否相等，OCSP 服务端应当将请求中的 nonce 拷贝到响应中。如果请求和响应中的 nonce 扩展项都存在，比较 nonce 值，如果不相等，返回错误，或者，请求中有 nonce，而响应中没有 nonce，也返回错误。验证正确时返回值大于 0。

32）int OCSP_copy_nonce(OCSP_BASICRESP *resp, OCSP_REQUEST *req)

将请求中都 nonce 拷贝到响应中。

33）X509_EXTENSION *OCSP_crlID_new(char *url, long *n, char *tim)

根据 crl 的 url，crl 个数以及生成 crl 的时间生成 X509_EXTENSION 扩展项。

34）X509_EXTENSION *OCSP_accept_responses_new(char **oids)

根据多个 oid 的名字生成扩展项，其中 oids 指针数组，以 NULL 结尾。本函数由客户端调用，告诉服务端它所要的端响应的类型，参考 rfc2560 对于 AcceptableResponses 扩展项的说明。

35）X509_EXTENSION *OCSP_archive_cutoff_new(char* tim)

生成单个证书的 Archive Cutoff 扩展项，某已被撤销的证书的 Archive Cutoff 时间为本次 OCSP 生效时间(producedAt)减去被撤销时的时间。可以将它看作已撤销了多长时间。

36）X509_EXTENSION *OCSP_url_svcloc_new(X509_NAME* issuer, char **urls);

根据颁发者名字和一个或多个 url 生成扩展项。扩展项内容为 AuthorityInfoAccess。urls 为指针数组，以 NULL 结束。

37）OCSP_CERTID *OCSP_cert_to_id(const EVP_MD *dgst, X509 *subject, X509 *issuer)

根据摘要算法、持有者证书和颁发者证书生成 OCSP_CERTID 数据结构。

38）OCSP_CERTID *OCSP_cert_id_new(const EVP_MD *dgst,
X509_NAME *issuerName,
ASN1_BIT_STRING* issuerKey,
ASN1_INTEGER *serialNumber)；

本函数根据摘要算法、颁发者名字、颁发者公钥 DER 编码以及证书持有者的证书序列号生成 OCSP_CERTID；奇怪的是 serialNumber 可以为空，无法标识需要查询状态证书。

39）int OCSP_id_issuer_cmp(OCSP_CERTID *a, OCSP_CERTID *b)

比较 OCSP_CERTID，如果相等返回 0，不相等返回非 0。本函数不比较证书序列号。

40）int OCSP_id_cmp(OCSP_CERTID *a, OCSP_CERTID *b)

比较 OCSP_CERTID，如果相等返回 0，不相等返回非 0。本函数比较所有项，包括证书序列号。

41）int OCSP_parse_url(char *url, char **phost, char **pport, char **ppath, int *pssl);

分析 url，获取主机、端口、路径和协议(http 还是 https)等信息。

42）char *OCSP_response_status_str(long s)

根据 OCSP 响应码获取响应状态信息。

43）char *OCSP_cert_status_str(long s)

根据证书状态码获取证书状态信息。

44）char *OCSP_crl_reason_str(long s)

根据状态码获取证书撤销原因。

45）int OCSP_REQUEST_print(BIO *bp, OCSP_REQUEST* o, unsigned long flags)

将 OCSP 请求 OCSP_REQUEST 的信息输出到 bp 中,flags 表明不支持到扩展项

的处理方式，参考 X509V3_extensions_print 以及 X509V3_EXT_print 函数。

46）int OCSP_RESPONSE_print(BIO *bp, OCSP_RESPONSE* o, unsigned long flags)

将 OCSP 请求 OCSP_RESPONSE 的信息输出到 bp 中,flags 表明不支持到扩展项到处理方式,参考 X509V3_extensions_print 以及 X509V3_EXT_print 函数。

47）int OCSP_request_onereq_count(OCSP_REQUEST *req)

获取 OCSP 请求中请求列表的个数,即多少个证书状态需要查询。

48）OCSP_ONEREQ *OCSP_request_onereq_get0(OCSP_REQUEST *req, int i)

根据在堆栈中到位置获取 OCSP_ONEREQ,OCSP_ONEREQ 包含了单个证书的信息。

49）OCSP_CERTID *OCSP_onereq_get0_id(OCSP_ONEREQ *one)

获取 OCSP_ONEREQ 中到证书 ID 信息。

50）int OCSP_id_get0_info(ASN1_OCTET_STRING **piNameHash,
ASN1_OBJECT **pmd,ASN1_OCTET_STRING **pikeyHash,
ASN1_INTEGER **pserial, OCSP_CERTID *cid)

从 cid 中获取颁发者名字摘要值、摘要算法、颁发者公钥摘要值以及持有者证书序列号,成功返回 1,否则为 0。

51) int OCSP_request_is_signed(OCSP_REQUEST *req)

判断请求是否已签名，如果已签名返回 1,否则返回 0。

52）OCSP_RESPONSE *OCSP_response_create(int status, OCSP_BASICRESP *bs)

生成 OCSP 响应数据，status 为响应状态，bs 为响应的具体内容。

53）OCSP_SINGLERESP *OCSP_basic_add1_status(OCSP_BASICRESP *rsp,
OCSP_CERTID *cid,
int status, int reason,
ASN1_TIME *revtime,
ASN1_TIME *thisupd, ASN1_TIME *nextupd);

根据输入参数证书 ID、证书状态、撤销原因、撤销时间、本次 OCSP 时间以及下次 OCSP 时间生成一个单一证书的状态信息，将此状态信息放入 rsp 的堆栈中，并返回此状态信息。

54）int OCSP_basic_add1_cert(OCSP_BASICRESP *resp, X509 *cert)

添加一个证书到响应信息中。

55）ASN1_STRING *ASN1_STRING_encode(ASN1_STRING *s, i2d_of_void *i2d,
void *data, STACK_OF(ASN1_OBJECT) *sk)

本函数将数据进行 DER 编码,编码后的结果放在 ASN1_STRING 中，并返回此 ASN1_STRING。其中，s 为要设置的 ASN1_STRING，i2d 为输入数据的 i2d 方法,data 为输入数据结构，sk 为输入对象堆栈。如果 data 不为空，则 DER 编码 data 指向的数据结构；如果 data 为空，sk 不为空，则 DER 编码 sk 堆栈表示的内容。

56）int OCSP_REQUEST_get_ext_count(OCSP_REQUEST *x)

获取 OCSP_REQUEST 结构中 tbsRequest 成员的扩展项的个数。

57）int OCSP_REQUEST_get_ext_by_NID(OCSP_REQUEST *x, int nid, int lastpos)

根据对象 nid 获取扩展项在 x->tbsRequest->requestExtensions 中的位置。

58）int OCSP_REQUEST_get_ext_by_OBJ(OCSP_REQUEST *x,
ASN1_OBJECT *obj, int lastpos)

获取对象在 x->tbsRequest->requestExtensions 中的位置。

59）int OCSP_REQUEST_get_ext_by_critical(OCSP_REQUEST *x, int crit, int lastpos)

根据是否关键 crit 以及堆栈搜索基准 lastpos 获取 x->tbsRequest->requestExtensions 中扩展项的位置。

60）X509_EXTENSION *OCSP_REQUEST_get_ext(OCSP_REQUEST *x, int loc)
根据扩展项在堆栈中的位置获取扩展项。

61）X509_EXTENSION *OCSP_REQUEST_delete_ext(OCSP_REQUEST *x, int loc)
根据扩展项在堆栈中的位置删除扩展项。

62）void *OCSP_REQUEST_get1_ext_d2i(OCSP_REQUEST *x, int nid, int *crit, int *idx)
根据扩展项 nid 获取扩展项信息，其中返回值为扩展项数据结构的指针地址，crit 返回是否时关键扩展，idx 表明它在堆栈中的位置。

63）int OCSP_REQUEST_add1_ext_i2d(OCSP_REQUEST *x, int nid, void *value, int crit,
unsigned long flags)
将具体的扩展项添加到 x 中，成功则返回 1。其中，nid 表明是什么扩展项，crit 表明是否是关键扩展，value 是具体扩展项数据结构的地址，flags 表明了何种操作，参考函数 X509V3_add1_i2d。

64）int OCSP_REQUEST_add_ext(OCSP_REQUEST *x, X509_EXTENSION *ex, int loc)
将扩展项添加到 x->tbsRequest->requestExtensions 堆栈中,loc 表示堆栈位置。

65）int OCSP_basic_verify(OCSP_BASICRESP *bs, STACK_OF(X509) *certs,
X509_STORE *st, unsigned long flags)
验证 OCSP 响应消息,成功返回 1。验证内容有：验证 OCSP 签名、验证签名者证书、检查每个证书状态信息的颁发者是否是相同、检查颁发者证书的扩展密钥用法中是否支持 OCSP 签名。

27.4 编程示例

ocsp 的编程主要是生成 ocsp 请求、解析 ocsp 请求、生成 ocsp 响应、解析 ocsp 响应得到结果以及消息的签名和验证。客户端可用 ocsp_cl.c 中提供的函数，服务端可用 ocsp_srv.c 中提供的函数。典型的应用程序请参考 apps/ocsp.c。

第二十八章 CRL

28.1 CRL 介绍

证书撤销列表(Certificate Revocation List，简称 CRL)，是一种包含撤销的证书列表的签名数据结构。CRL 是证书撤销状态的公布形式，CRL 就像信用卡的黑名单，用于公布某些数字证书不再有效。

CRL 是一种离线的证书状态信息。它以一定的周期进行更新。CRL 可以分为完全 CRL 和增量 CRL。在完全 CRL 中包含了所有的被撤销证书信息，增量 CRL 由一系列的 CRL 来表明被撤销的证书信息，它每次发布的 CRL 是对前面发布 CRL 的增量扩充。

基本的 CRL 信息有：被撤销证书序列号、撤销时间、撤销原因、签名者以及 CRL 签名等信息。

基于 CRL 的验证是一种不严格的证书认证。CRL 能证明在 CRL 中被撤销的证书是无效的。但是，它不能给出不在 CRL 中的证书的状态。如果执行严格的认证，需要采用在线方式进行认证，即 OCSP 认证。

28.2 数据结构

Openssl 中的 crl 数据结构定义在 crypto/x509/x509.h 中。

1） X509_REVOKED

```
typedef struct X509_revoked_st
{
    ASN1_INTEGER *serialNumber;
    ASN1_TIME *revocationDate;
    STACK_OF(X509_EXTENSION) *extensions;
    int sequence;
} X509_REVOKED;
```

本结构用于存放一个被撤销证书的信息，各项意义如下：

serialNumber：被撤销证书的序列号；

revocationDate：撤销时间；

extensions：扩展项，可选；

sequence：顺序号，用于排序，表示当前被撤销证书信息在 crl 中的顺序。

2）X509_CRL_INFO

```
typedef struct X509_crl_info_st
{
    ASN1_INTEGER *version;
    X509_ALGOR *sig_alg;
    X509_NAME *issuer;
    ASN1_TIME *lastUpdate;
    ASN1_TIME *nextUpdate;
    STACK_OF(X509_REVOKED) *revoked;
```

```
    STACK_OF(X509_EXTENSION)  *extensions;
    ASN1_ENCODING enc;
} X509_CRL_INFO;
```

crl 信息主体，各项意义如下：

version：crl 版本；

sig_alg：crl 签名法；

issuer：签发者信息；

lastUpdate：上次更新时间；

nextUpdate：下次更新时间；

revoked：被撤销证书信息；

extensions：扩展项，可选。

3) X509_CRL

```
struct X509_crl_st
{
    X509_CRL_INFO *crl;
    X509_ALGOR *sig_alg;
    ASN1_BIT_STRING *signature;
    int references;
} ;
```

完整 crl 数据结构，各项意义如下：

crl：crl 信息主体；

sig_alg：签名算法，与 X509_CRL_INFO 中的一致；

signature：签名值；

references：引用。

上述三个结构的 DER 编解码通过宏在 crypto/asn1/x_crl.c 中实现，包括 new、free、i2d 和 d2i 函数。

28.3 CRL 函数

CRL 函数主要是 set 和 get 函数，如下：

1） int X509_CRL_add0_revoked(X509_CRL *crl, X509_REVOKED *rev)
添加一个被撤销证书的信息。

2） int X509_CRL_print(BIO *bp,X509_CRL *x)
打印 crl 内容到 BIO 中。

3） int X509_CRL_print_fp(FILE *fp, X509_CRL *x)
将 crl 的内容输出到 fp 中，此函数调用了 X509_CRL_print。

4） int X509_CRL_set_issuer_name(X509_CRL *x, X509_NAME *name)
设置 crl 的颁发者。

5) int X509_CRL_set_lastUpdate(X509_CRL *x, ASN1_TIME *tm)
设置 crl 上次发布时间。

6) int X509_CRL_set_nextUpdate(X509_CRL *x, ASN1_TIME *tm)
设置 crl 下次发布时间。

7) int X509_CRL_set_version(X509_CRL *x, long version)
设置 crl 版本。

8） int X509_CRL_sign(X509_CRL *x, EVP_PKEY *pkey, const EVP_MD *md)
对 crl 进行签名，pkey 为私钥，md 为摘要算法，结果存放在 x-> signature 中。

9） int X509_CRL_sort(X509_CRL *c)
根据证书序列号对 crl 排序，此函数实现采用了堆栈排序，堆栈的比较函数为 X509_REVOKED_cmp(crypto/asn1/x_crl.c)。

10）int X509_CRL_add1_ext_i2d(X509_CRL *x, int nid, void *value, int crit, unsigned long flags)
添加 CRL 扩展，nid 为要添加的扩展标识，value 为被添加的具体扩展项的内部数据结构地址，crit 表明是否为关键扩展，flags 表明何种操作。此函数调用 X509V3_add1_i2d 函数。

11）int X509_CRL_add_ext(X509_CRL *x, X509_EXTENSION *ex, int loc)
添加扩展项到指定堆栈位置，此函数调用 X509v3_add_ext，进行堆栈插入操作。

12）int X509_CRL_cmp(const X509_CRL *a, const X509_CRL *b)
CRL 比较，此函数调用 X509_NAME_cmp，只比较颁发者的名字是否相同。

13）X509_EXTENSION *X509_CRL_delete_ext(X509_CRL *x, int loc)
删除 CRL 扩展项堆栈中的某一项，loc 指定被删除项在堆栈中的位置。

14）int X509_CRL_digest(const X509_CRL *data, const EVP_MD *type,
unsigned char *md, unsigned int *len)
CRL 摘要，本函数对 X509_CRL 进行摘要，type 指定摘要算法，摘要结果存放在 md 中，len 表明摘要结果长度。

15）X509_CRL_dup
CRL 数据拷贝，此函数通过宏来实现。大部分 ASN1 类型数据都有 dup 函数，它们的实现方式比较简单：将对象 DER 编码，然后再解码，这样就实现了 ASN1 数据的复制。

16) void *X509_CRL_get_ext_d2i(X509_CRL *x, int nid, int *crit, int *idx)
CRL 中的获取扩展项，此函数用于获取 crl 中指定扩展项的内部数据结构，返回值为具体的扩展项数据结构地址，nid 为扩展项标识，它调用了 X509V3_get_d2i 函数。

17）int X509_CRL_get_ext_by_critical(X509_CRL *x, int crit, int lastpos)
获取扩展项在其堆栈中的位置，crit 为扩展项是否关键标识，lastpos 为堆栈搜索起始位置。此函数调用了 X509v3_get_ext_by_critical。

18）int X509_CRL_get_ext_by_NID(X509_CRL *x, int nid, int lastpos)
获取扩展项在其堆栈中的位置，nid 为扩展项标识，lastpos 为搜索起始位置。如果找到此扩展项，返回其在堆栈中的位置。

19) int X509_CRL_get_ext_by_OBJ(X509_CRL *x, ASN1_OBJECT *obj, int lastpos)
同上。

20）int X509_CRL_get_ext_count(X509_CRL *x)
获取 crl 中扩展项的个数。

21) int X509_CRL_verify(X509_CRL *a, EVP_PKEY *r)
验证 CRL。EVP_PKEY 结构 r 中需要给出公钥。

28.4 编程示例

下面的例子用来生成一个 crl 文件。

```
#include <openssl/x509.h>
int	main()
{
	int			ret,len;
	unsigned char	*buf,*p;
	unsigned long	e=RSA_3;
	FILE			*fp;
	time_t			t;
	X509_NAME		*issuer;
	ASN1_TIME		*lastUpdate,*nextUpdate,*rvTime;
	X509_CRL		*crl=NULL;
	X509_REVOKED	*revoked;
	EVP_PKEY		*pkey;
	ASN1_INTEGER	*serial;
	RSA			*r;
	BIGNUM			*bne;
	BIO			*bp;

	/*  生成密钥*/
	bne=BN_new();
	ret=BN_set_word(bne,e);
	r=RSA_new();
	ret=RSA_generate_key_ex(r,1024,bne,NULL);
	if(ret!=1)
	{
		printf("RSA_generate_key_ex err!\n");
		return -1;
	}
	pkey=EVP_PKEY_new();
	EVP_PKEY_assign_RSA(pkey,r);
	crl=X509_CRL_new();
	/*  设置版本*/
	ret=X509_CRL_set_version(crl,3);
	/*  设置颁发者*/
	issuer=X509_NAME_new();
	ret=X509_NAME_add_entry_by_NID(issuer,NID_commonName,V_ASN1_PRINTAB
LESTRING,"CRL issuer",10,-1,0);
	ret=X509_CRL_set_issuer_name(crl,issuer);
	/*  设置上次发布时间*/
	lastUpdate=ASN1_TIME_new();
	t=time(NULL);
	ASN1_TIME_set(lastUpdate,t);
	ret=X509_CRL_set_lastUpdate(crl,lastUpdate);
	/*  设置下次发布时间*/
```

```
    nextUpdate=ASN1_TIME_new();
    t=time(NULL);
    ASN1_TIME_set(nextUpdate,t+1000);
    ret=X509_CRL_set_nextUpdate(crl,nextUpdate);
    /* 添加被撤销证书序列号*/
    revoked=X509_REVOKED_new();
    serial=ASN1_INTEGER_new();
    ret=ASN1_INTEGER_set(serial,1000);
    ret=X509_REVOKED_set_serialNumber(revoked,serial);
    rvTime=ASN1_TIME_new();
    t=time(NULL);
    ASN1_TIME_set(rvTime,t+2000);
    ret=X509_CRL_set_nextUpdate(crl,rvTime);
    ret=X509_REVOKED_set_revocationDate(revoked,rvTime);
    ret=X509_CRL_add0_revoked(crl,revoked);
    /* 排序*/
    ret=X509_CRL_sort(crl);
    /* 签名*/
    ret=X509_CRL_sign(crl,pkey,EVP_md5());
    /* 写入文件*/
    bp=BIO_new(BIO_s_file());
    BIO_set_fp(bp,stdout,BIO_NOCLOSE);
    X509_CRL_print(bp,crl);
    len=i2d_X509_CRL(crl,NULL);
    buf=malloc(len+10);
    p=buf;
    len=i2d_X509_CRL(crl,&p);
    fp=fopen("crl.crl","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    BIO_free(bp);
    X509_CRL_free(crl);
    free(buf);
    getchar();
    return 0;
}
```

第二十九章 PKCS7

29.1 概述

加密消息语法（pkcs7），是各种消息存放的格式标准。这些消息包括：数据、签名数据、数字信封、签名数字信封、摘要数据和加密数据。

29.2 数据结构

Openssl 的 pkcs7 实现在 crypto/pkcs7 目录下。pkcs7 的各种消息数据结构和函数在 crypto/pkcs7/pkcs7.h 中定义，主要数据结构如下：

```
typedef struct pkcs7_st
{
    /* 其他项 */
    ASN1_OBJECT *type;
    union
    {
        char *ptr;
        /* NID_pkcs7_data */
        ASN1_OCTET_STRING *data;
        /* NID_pkcs7_signed */
        PKCS7_SIGNED *sign;
        /* NID_pkcs7_enveloped */
        PKCS7_ENVELOPE *enveloped;
        /* NID_pkcs7_signedAndEnveloped */
        PKCS7_SIGN_ENVELOPE *signed_and_enveloped;
        /* NID_pkcs7_digest */
        PKCS7_DIGEST *digest;
        /* NID_pkcs7_encrypted */
        PKCS7_ENCRYPT *encrypted;
        /* Anything else */
        ASN1_TYPE *other;
        } d;
    } PKCS7;
```

其中 type 用于表示是何种类型的 pkcs7 消息，data、sign、enveloped、signed_and_enveloped、digest 和 ncrypted 对于了 6 种不同的具体消息。oher 用于存放任意数据类型（也可以是 pkcs7 结构），所以，本结构可以是一个嵌套的数据结构。

pkcs7 各种类型数据结构的 DER 编解码通过宏在 crypto/pkcs7/pk7_asn1.c 中实现，包括 new、free、i2d 和 d2i 函数。

29.3 函数

1) PKCS7_add_attrib_smimecap
给 PKCS7_SIGNER_INFO 添加 NID_SMIMECapabilities 属性。

2） int PKCS7_add_attribute(PKCS7_SIGNER_INFO *p7si, int nid, int atrtype,void *value)
给 PKCS7_SIGNER_INFO 添加属性，nid 为属性类型，value 为属性的 ASN1 数据结构，atrtype 为 value 的 ASN1 类型。

3） int PKCS7_add_certificate(PKCS7 *p7, X509 *x509)
将证书添加到 PKCS7 对应消息的证书堆栈中，只对 NID_pkcs7_signed 和 NID_pkcs7_signedAndEnveloped 两种类型有效。

4） PKCS7_add_crl
将 crl 添加到 PKCS7 对应消息的 crl 堆栈中，只对 NID_pkcs7_signed 和 NID_pkcs7_signedAndEnveloped 两种类型有效。

5） PKCS7_add_recipient/ PKCS7_add_recipient_info
添加接收者信息。

6）PKCS7_add_signer
添加一个签名者信息。

7） KCS7_add_signed_attribute
给 PKCS7_SIGNER_INFO 添加属性。

8） PKCS7_cert_from_signer_info
从 pkcs7 消息中根据颁发者和证书序列号获取证书。

9） PKCS7_ctrl
控制函数。

10）PKCS7_dataDecode
解析输入的 pkcs7 消息，将结果存入 BIO 链表并返回。

11）PKCS7_dataInit/PKCS7_dataFinal
解析输入的 pkcs7 消息，将结果存入 BIO。

12）PKCS7_dataVerify
验证 pkcs7 数据。

13）PKCS7_sign
签名 pkcs7 消息。

14) PKCS7_verify
验证 pkcs7 消息。

15）PKCS7_set_type
设置 pkcs7 消息类型。

16）PKCS7_dup
拷贝 pkcs7 结构。

29.4 消息编解码

PKCS7 编码时调用函数 i2d_PKCS7，在调用此函数之前，需要填充其内部数据结构。PKCS7 解码时调用函数 d2i_PKCS7 获取内部数据结构。

下面是一些编码的示例。

29.4.1 data

```
/* pkcs7    data */
#include <string.h>
#include <openssl/pkcs7.h>
#include <openssl/objects.h>

int    main()
{
       PKCS7                   *p7;
       int                     len;
       char                buf[1000],*der,*p;
       FILE                    *fp;

       p7=PKCS7_new();
       PKCS7_set_type(p7,NID_pkcs7_data);
       strcpy(buf,"pkcs7 data !\n");
       len=strlen(buf);
       ASN1_OCTET_STRING_set(p7->d.data,(const unsigned char *)buf,len);

       len=i2d_PKCS7(p7,NULL);
       der=(char *)malloc(len);
       p=der;
       len=i2d_PKCS7(p7,(unsigned char **)&p);
       fp=fopen("p7_data.cer","wb");
       fwrite(der,1,len,fp);
       fclose(fp);
       PKCS7_free(p7);
       free(der);
       return 0;
}
```

本例用于生成 data 类型的 pkcs7 消息。

29.4.2 signed data

```
#include <openssl/pem.h>
#include <openssl/pkcs7.h>
#include <openssl/objects.h>
#include <openssl/x509.h>
int    main()
{
```

```
    PKCS7                   *p7;
    int                     len;
    unsigned char       *der,*p;
    FILE                    *fp;
    X509                    *x;
    BIO                     *in;
    X509_ALGOR                  *md;
    PKCS7_SIGNER_INFO*si;

    p7=PKCS7_new();
    PKCS7_set_type(p7,NID_pkcs7_signed);
    p7->d.sign->cert=sk_X509_new_null();
    in=BIO_new_file("b64cert.cer","r");
    x=PEM_read_bio_X509(in,NULL,NULL,NULL);
    sk_X509_push(p7->d.sign->cert,x);
    md=X509_ALGOR_new();
    md->algorithm=OBJ_nid2obj(NID_md5);
    sk_X509_ALGOR_push(p7->d.sign->md_algs,md);
    si=PKCS7_SIGNER_INFO_new();
    ASN1_INTEGER_set(si->version,2);
    ASN1_INTEGER_set(si->issuer_and_serial->serial,333);
    sk_PKCS7_SIGNER_INFO_push(p7->d.sign->signer_info,si);
    len=i2d_PKCS7(p7,NULL);
    der=(unsigned char *)malloc(len);
    p=der;
    len=i2d_PKCS7(p7,&p);
    fp=fopen("p7_sign.cer","wb");
    fwrite(der,1,len,fp);
    fclose(fp);
    free(der);
    PKCS7_free(p7);
    return 0;
}
```

本例用于生成 signed 类型的 pkcs7 消息。

29.4.3 enveloped

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>
#include <openssl/x509.h>
int   main()
{
    PKCS7                   *p7;
    int                     len;
```

```
    char                *der,*p;
    FILE                *fp;
    PKCS7_RECIP_INFO    *inf;

    p7=PKCS7_new();
    PKCS7_set_type(p7,NID_pkcs7_enveloped);
    ASN1_INTEGER_set(p7->d.enveloped->version,3);
    inf=PKCS7_RECIP_INFO_new();
    ASN1_INTEGER_set(inf->version,4);
    ASN1_INTEGER_set(inf->issuer_and_serial->serial,888888);
    inf->key_enc_algor->algorithm=OBJ_nid2obj(NID_des_ede3_cbc);
    ASN1_OCTET_STRING_set(inf->enc_key,(const unsigned char *)"key
info....",12);
    sk_PKCS7_RECIP_INFO_push(p7->d.enveloped->recipientinfo,inf);

  p7->d.enveloped->enc_data->algorithm->algorithm=OBJ_nid2obj(NID_des_ede3_cbc);
    p7->d.enveloped->enc_data->enc_data=ASN1_OCTET_STRING_new();
    ASN1_OCTET_STRING_set(p7->d.enveloped->enc_data->enc_data,(const
unsigned char *)"info....",8);
    len=i2d_PKCS7(p7,NULL);
    der=(char *)malloc(len);
    p=der;
    len=i2d_PKCS7(p7,(unsigned char **)&p);
    fp=fopen("p7_evveloped.cer","wb");
    fwrite(der,1,len,fp);
    fclose(fp);
    PKCS7_free(p7);
    free(der);
    return 0;
}
```

本例用于生成 enveloped 类型的 pkcs7 消息。

29.4.4 signed_and_enveloped

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>

int  main()
{
    PKCS7               *p7;
    int                 len;
    char            *der,*p;
    FILE                *fp;
```

```
    p7=PKCS7_new();
    PKCS7_set_type(p7,NID_pkcs7_signedAndEnveloped);
    len=i2d_PKCS7(p7,NULL);
    der=(char *)malloc(len);
    p=der;
    len=i2d_PKCS7(p7,(unsigned char **)&p);
    fp=fopen("p7_singAndEnv.cer","wb");
    fwrite(der,1,len,fp);
    fclose(fp);
    PKCS7_free(p7);
    free(der);
    return 0;
}
```

本例用于生成 signedAndEnveloped 类型的 pkcs7 消息,不过省略了数据结构的填充。

29.4.5 digest

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>
#include <openssl/pem.h>
int  main()
{
    PKCS7               *p7;
    int                 ret;
    BIO                 *b;

    p7=PKCS7_new();
    ret=PKCS7_set_type(p7,NID_pkcs7_digest);
    b=BIO_new_file("p7Digest.pem","w");
    PEM_write_bio_PKCS7(b,p7);
    BIO_free(b);
    PKCS7_free(p7);
    return 0;
}
```

本例用于生成 digest 类型的 pkcs7 消息,并以 PEM 格式存储。

29.4.6 encrypted

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>
#include <openssl/x509.h>
int  main()
{
```

```
    PKCS7                   *p7;
    int                     ret,len;
    char                *der,*p;
    FILE                    *fp;

    p7=PKCS7_new();
    ret=PKCS7_set_type(p7,NID_pkcs7_encrypted);
    ASN1_INTEGER_set(p7->d.encrypted->version,3);

p7->d.encrypted->enc_data->algorithm->algorithm=OBJ_nid2obj(NID_des_ede3_cbc);
    p7->d.encrypted->enc_data->enc_data=ASN1_OCTET_STRING_new();
    ASN1_OCTET_STRING_set(p7->d.encrypted->enc_data->enc_data,(const
unsigned char *)"3434",4);
    len=i2d_PKCS7(p7,NULL);
    der=(char *)malloc(len);
    p=der;
    len=i2d_PKCS7(p7,(unsigned char **)&p);
    fp=fopen("p7_enc.cer","wb");
    fwrite(der,1,len,fp);
    fclose(fp);
    PKCS7_free(p7);
    free(der);
    return 0;
}
```

本例用于生成 encrypted 类型的 pkcs7 消息。

29.4.7 读取 PEM

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>
#include <openssl/pem.h>
int   main()
{
    BIO       *b;
    PKCS7   *p7;

    b=BIO_new_file("p7Digest.pem","r");
    p7=PEM_read_bio_PKCS7(b,NULL,NULL,NULL);
    BIO_free(b);
    PKCS7_free(p7);
    return 0;
}
```

本例用于读取 PEM 格式的 PKCS7 数据。

29.4.8 解码 pkcs7

```
#include <openssl/pkcs7.h>
#include <openssl/objects.h>

int   main()
{
      PKCS7                  *p7=NULL;
      int                    ret,len;
      char                buf[1000],*p,name[1000];
      FILE                   *fp;


      fp=fopen("p7_sign.cer","rb");
      len=fread(buf,1,1000,fp);
      fclose(fp);

      p=buf;
      d2i_PKCS7(&p7,(const unsigned char **)&p,len);
      ret=OBJ_obj2txt(name,1000,p7->type,0);
      printf("type : %s \n",name);
      PKCS7_free(p7);
      return 0;
}
```

本例解码 DER 格式的 PKCS7 消息。

第三十章　PKCS12

30.1 概述

pkcs12 (个人数字证书标准)用于存放用户证书、crl、用户私钥以及证书链。pkcs12 中的私钥是加密存放的。

30.2 openss 实现

openssl 的 pkcs12 实现在 crypto/pkcs12 目录，有如下源码：

- p12_add.c：处理 PKCS12_SAFEBAG，PKCS12_SAFEBAG 用于存放证书和私钥相关的信息；
- p12_attr.c：属性处理；
- p12_crt：生成一个完整的 pkcs12；
- p12_init.c：构造一个 pkcs12 数据结构；
- p12_kiss.c：解析 pkcs12 结构，获取证书和私钥等信息；
- p12_npas：设置新口令；
- p12_p8e.c：加密处理用户私钥(pkcs8 格式)；
- p12_p8d.c：解密出用户私钥；
- pk12err.c：错误处理；
- p12_asn.c：pkcs12 各个数据结构的 DER 编解码实现；
- p12_crpt.c：pkcs12 的 pbe(基于口令的加密)函数；
- p12_decr.c.c：pkcs12 的 pbe 解密；
- p12_key.c：根据用户口令生成对称密钥；
- p12_mutl.c：pkcs12 的 MAC 信息处理；
- p12_utl.c：一些通用的函数。

30.3 数据结构

数据结构定义在 crypto/pkcs12/pkcs12.h 中，如下所示：

1）PKCS12_MAC_DATA

```
typedef struct
{
    X509_SIG *dinfo;
    ASN1_OCTET_STRING *salt;
    ASN1_INTEGER *iter;
} PKCS12_MAC_DATA;
```

该结构用于存放 pkcs12 中的 MAC 信息，防止他人篡改。xinfo 用于存放 MAC 值和摘要算法，salt 和 iter 用于根据口令来生成对称密钥(pbe)。

2）PKCS12

```
typedef struct
```

```
{
    ASN1_INTEGER *version;
    PKCS12_MAC_DATA *mac;
    PKCS7 *authsafes;
} PKCS12;
```

pkcs12 数据结构，version 为版本，mac 用于存放 MAC 信息以及对称密钥相关的信息 authsafes 为 pkcs7 结构，用于存放的证书、crl 以及私钥等各种信息。

3）PKCS12_BAGS

```
typedef struct pkcs12_bag_st
{
    ASN1_OBJECT *type;
    union
    {
        ASN1_OCTET_STRING *x509cert;
        ASN1_OCTET_STRING *x509crl;
        ASN1_OCTET_STRING *octet;
        ASN1_IA5STRING *sdsicert;
        ASN1_TYPE *other;
    }value;
} PKCS12_BAGS;
```

该结构用于存放各种实体对象。

4）PKCS12_SAFEBAG

```
typedef struct
{
    ASN1_OBJECT *type;
    union
    {
        struct pkcs12_bag_st *bag;
        struct pkcs8_priv_key_info_st    *keybag;
        X509_SIG *shkeybag;
        STACK_OF(PKCS12_SAFEBAG) *safes;
        ASN1_TYPE *other;
    }value;
    STACK_OF(X509_ATTRIBUTE) *attrib;
} PKCS12_SAFEBAG;
```

该结构用于存放各种证书、crl 和私钥数据。

上述两种结构与 pkcs7 数据结构的相互转化可参考 p12_add.c。在使用中，用户根据证书、私钥以及 crl 等信息来构造 PKCS12_SAFEBAG 数据结构，然后将这些结构转化为 pkcs12 中的 pkcs7 结构。

30.4 函数

1) int PKCS12_gen_mac(PKCS12 *p12, const char *pass, int passlen, unsigned char *mac, unsigned int *maclen)

生成 MAC 值，pass 为用户口令，passlen 为口令长度，mac 和 maclen 用于存放 MAC 值。当 p12 中 pkcs7 为数据类型时，本函数有效。

2） int PKCS12_verify_mac(PKCS12 *p12, const char *pass, int passlen)

验证 pkcs12 的 MAC，pass 为用户口令，passlen 为口令长度。PKCS12 的 MAC 值存放在 p12-> mac-> dinfo->digest 中。本函数根据 pass 和 passlen 调用 PKCS12_gen_mac 生成一个 MAC 值，与 p12 中已有的值进行比较。

3） PKCS12_create

成 PKCS12 数据结构。

4） PKCS12_parse

解析 PKCS12，得到私钥和证书等信息。

5) PKCS12_key_gen_asc/PKCS12_key_gen_uni

生成 pkcs12 密钥，输入口令为 ASCII 码/UNICODE。

6）unsigned char * PKCS12_pbe_crypt(X509_ALGOR *algor, const char *pass,
int passlen, unsigned char *in, int inlen, unsigned char **data,
int *datalen, int en_de)

PKCS12 加解密，algor 为对称算法，pass 为口令，passlen 为口令长度，in 为输入数据，inlen 为输入数据长度，data 和 datalen 用于存放结果，en_de 用于指明时加密还是解密。

7） PKCS7 *PKCS12_pack_p7data(STACK_OF(PKCS12_SAFEBAG) *sk)

打包 PKCS12_SAFEBAG 堆栈，生成 PKCS7 数据结构并返回。

8） PKCS12_unpack_p7data

上面函数的逆过程。

9) PKCS12_pack_p7encdata

将 PKCS12_SAFEBAG 堆栈根据 pbe 算法、口令和 salt 加密，生成 pkcs7 并返回。

10）PKCS12_unpack_p7encdata

上述过程的逆过程。

11) int PKCS12_newpass(PKCS12 *p12, char *oldpass, char *newpass)

替换 pkcs12 的口令。

12）PKCS12_setup_mac

设置 pkcs12 的 MAC 数据结构。

13）PKCS12_set_mac

设置 pkcs12 的 MAC 信息。

14）PKCS12_pack_authsafes

将 pkcs7 堆栈信息打包到 pkcs12 中。

15）PKCS12_unpack_authsafes

上面函数的逆过程，从 pkcs12 中解出 pkcs7 堆栈，并返回。

16）PKCS12 *PKCS12_init(int mode)

生成一个 pkcs12 数据结构，mode 的值必须为 NID_pkcs7_data，即 pkcs12 中的 pkcs7 类型必须是 data 类型。

17）PKCS12_PBE_add

加载各种 pbe 算法。

18）PKCS12_PBE_keyivgen

根据口令生成对称密钥，并做加解密初始化。

19）PKCS12_item_pack_safebag

将输入的数据打包为 PKCS12_SAFEBAG 并返回。

20）PKCS12_x5092certbag

将证书打包为 PKCS12_SAFEBAG 并返回。

21）PKCS12_certbag2x509

上述过程的逆过程。

22）PKCS12_x509crl2certbag

将 crl 打包为 PKCS12_SAFEBAG 并返回。

23）PKCS12_certbag2x509crl

上述过程的逆过程。

24）PKCS12_item_i2d_encrypt

将数据结构 DER 编码，然后加密，数据存放在 ASN1_OCTET_STRING 中并返回。

24）PKCS12_item_decrypt_d2i

上面函数的逆过程，解密输入数据，然后 DER 解码出数据结构，并返回。

25）int PKCS12_add_friendlyname_uni(PKCS12_SAFEBAG *bag,
const unsigned char *name, int namelen)

给 PKCS12_SAFEBAG 添加一个属性，属性类型为 NID_friendlyName，name 为 unicode 编码。

26）int PKCS12_add_friendlyname_asc(PKCS12_SAFEBAG *bag, const char *name,
int namelen)

给 PKCS12_SAFEBAG 添加一个属性，属性类型为 NID_friendlyName，name 为 ASCII 码。

27) PKCS12_get_friendlyname

上面函数的逆过程，返回一个 ASCII 码值。

28）PKCS12_add_CSPName_asc

给 PKCS12_SAFEBAG 添加一个 NID_ms_csp_name 属性，输入参数为 ASCII 码。

29）PKCS12_add_localkeyid

给 PKCS12_SAFEBAG 添加一个 NID_localKeyID 属性。

30）PKCS12_MAKE_SHKEYBAG

将 pkcs8 密钥转化为 PKCS12_SAFEBAG。

30）PKCS8_PRIV_KEY_INFO *
PKCS12_decrypt_skey(PKCS12_SAFEBAG *bag, const char *pass, int passlen)

上面函数的逆过程，从 bag 中提取 pkcs8 密钥信息。

30.5 编程示例

1）pkcs12 解码

之一：

```
#include <string.h>
#include <openssl/pkcs12.h>
int   X509_ALGOR_print(BIO *bp,X509_ALGOR *signature)
{
    int                 nid;
    unsigned char  *p;
```

```
    PBEPARAM        *pbe=NULL;

    nid=OBJ_obj2nid(signature->algorithm);
    switch(nid)
    {
        case NID_md5WithRSAEncryption:
                printf("md5WithRSAEncryption");
                break;
        case NID_sha1WithRSAEncryption:
                printf("sha1WithRSAEncryption");
                break;
        case NID_rsaEncryption:
                printf("rsaEncryption");
                break;
        case NID_sha1:
                printf("sha1");
                break;
        case NID_pbe_WithSHA1And3_Key_TripleDES_CBC:
                printf("NID_pbe_WithSHA1And3_Key_TripleDES_CBC");
                break;
        default:
                printf("unknown signature.");
                break;
    }
    if(signature->parameter!=NULL)
    {
        if(nid==NID_pbe_WithSHA1And3_Key_TripleDES_CBC)
        {
            printf("算法参数:\n");
            p=signature->parameter->value.sequence->data;
            d2i_PBEPARAM(&pbe,&p,signature->parameter->value.sequence->length);
            printf("salt : \n");
            i2a_ASN1_INTEGER(bp,pbe->salt);
            printf("\n");
            printf("iter : %d\n",ASN1_INTEGER_get(pbe->iter));
        }
    }
    printf("\n");
    return 0;
}

void X509_SIG_print(BIO *bp,X509_SIG *a)
{
    if(a->algor!=NULL)
```

```
        {
            printf("算法:\n");
            X509_ALGOR_print(bp,a->algor);
        }
        if(a->digest!=NULL)
        {
            printf("摘要:\n");
            i2a_ASN1_STRING(bp,a->digest,1);
        }
}

void PKCS12_SAFEBAG_print(BIO *bp,PKCS12_SAFEBAG *bag)
{
        int                     nid,attrnum,certl,len=50,k,n,x;
        unsigned char  *p,buf[50];
        PBEPARAM        *pbe=NULL;
        X509_ATTRIBUTE      *attr;
        ASN1_TYPE       *type;
        X509                    *cert=NULL;

        nid=OBJ_obj2nid(bag->type);
        if((nid==NID_pkcs8ShroudedKeyBag)||
(nid==NID_pbe_WithSHA1And3_Key_TripleDES_CBC)) /* pkcs 8 */
        {
            nid=OBJ_obj2nid(bag->value.shkeybag->algor->algorithm);
            if(nid==NID_pbe_WithSHA1And3_Key_TripleDES_CBC)
            {
                /* alg */
                X509_SIG_print(bp,bag->value.shkeybag);
            }
        }
        else if(nid==NID_certBag)
        {
            nid=OBJ_obj2nid(bag->value.bag->type);
            if(nid==NID_x509Certificate)
            {
                p=bag->value.bag->value.x509cert->data;
                certl=bag->value.bag->value.x509cert->length;
                d2i_X509(&cert,&p,certl);
                if(cert!=NULL)
                {
                    X509_print(bp,cert);
                }
```

```
            }
        }
        printf("attris : \n");
        attrnum=sk_X509_ATTRIBUTE_num(bag->attrib);
        for(k=0;k<attrnum;k++)
        {
                attr=sk_X509_ATTRIBUTE_value(bag->attrib,k);
                nid=OBJ_obj2nid(attr->object);
                OBJ_obj2txt(buf,len,attr->object,1);
                printf("object : %s,nid is %d\n",buf,nid);
                if(attr->single==0) /* set */
                {
                        n=sk_ASN1_TYPE_num(attr->value.set);
                        for(x=0;x<n;x++)
                        {
                                type=sk_ASN1_TYPE_value(attr->value.set,x);
                                if((type->type!=V_ASN1_SEQUENCE)                &&
(type->type!=V_ASN1_SET))
                                {
                                    if(type->type==V_ASN1_OCTET_STRING)
                                        i2a_ASN1_INTEGER(bp,type->value.octet_string);
                                    else
                                        ASN1_STRING_print(bp,(ASN1_STRING
*)type->value.ptr);
                                }
                        }
                }
            printf("\n");
        }
}

int    main()
{
        FILE                *fp;
        PKCS12              *p12=NULL;
        PKCS7               *p7=NULL,*one;
        unsigned char  buf[10000],*p;
        int                 len,i,num,j,count,ret;
        STACK_OF(PKCS7) *p7s;
        STACK_OF(PKCS12_SAFEBAG) *bags;
        PKCS12_SAFEBAG     *bag;
        PBEPARAM           *pbe=0;
        BIO                 *bp;
        char              pass[100];
```

```
int                 passlen;
X509                *cert=NULL;
STACK_OF(X509)*ca=NULL;
EVP_PKEY            *pkey=NULL;

fp=fopen("timeserver.pfx","rb");
len=fread(buf,1,10000,fp);
fclose(fp);

OpenSSL_add_all_algorithms();
bp=BIO_new(BIO_s_file());
BIO_set_fp(bp,stdout,BIO_NOCLOSE);
p=buf;
d2i_PKCS12(&p12,&p,len);
printf("input password : \n");
scanf("%s",pass);
ret=PKCS12_parse(p12,pass,&pkey,&cert,&ca);
if(ret!=1)
{
    printf("err\n");
    return 0;
}
/*  私钥写入文件  */
p=buf;
len=i2d_PrivateKey(pkey,&p);
fp=fopen("prikey.cer","wb");
fwrite(buf,1,len,fp);
fclose(fp);
/*  修改密码  */
ret=PKCS12_newpass(p12,pass,"test");
fp=fopen("newpass.pfx","wb");
ret=i2d_PKCS12_fp(fp,p12);
fclose(fp);
/* version */
printf("version : %d\n",ASN1_INTEGER_get(p12->version));
/*    PKCS12_MAC_DATA */
printf("PKCS12_MAC_DATA sig :\n");
X509_SIG_print(bp,p12->mac->dinfo);
printf("salt : \n");
i2a_ASN1_STRING(bp,p12->mac->salt,1);
printf("iter : %d\n",ASN1_INTEGER_get(p12->mac->iter));
/* p7s */
p7s=PKCS12_unpack_authsafes(p12);
num=sk_PKCS7_num(p7s);
```

```
    for(i=0;i<num;i++)
    {
        one=sk_PKCS7_value(p7s,i);
        if(PKCS7_type_is_data(one))
        {
            bags = PKCS12_unpack_p7data(one);
            count=sk_PKCS12_SAFEBAG_num(bags);
            for(j=0;j<count;j++)
            {
                bag=sk_PKCS12_SAFEBAG_value(bags,j);
                PKCS12_SAFEBAG_print(bp,bag);
            }
        }
        else if(PKCS7_type_is_encrypted(one))
        {
        back:
            printf("\ninput password :\n");
            scanf("%s",pass);
            passlen=strlen(pass);
            bags = PKCS12_unpack_p7encdata(one,pass,passlen);
            if(bags==NULL)
                goto back;
            printf("passwod is :%s\n",pass);
            count=sk_PKCS12_SAFEBAG_num(bags);
            for(j=0;j<count;j++)
            {
                bag=sk_PKCS12_SAFEBAG_value(bags,j);
                PKCS12_SAFEBAG_print(bp,bag);
            }
        }

    }
    BIO_free(bp);
    sk_PKCS7_pop_free(p7s,PKCS7_free);
    PKCS12_free(p12);
    return 0;
}
```

之二：采用 PKCS12_parse 函数，下面的例子用于解析 pkcs12 文件，获取证书，以及 RSA 密钥信息。

```
int     p12_parse
(
char *p12,int p12Len,char *pass,char *cert,int *certlen,
char *n,int *nlen,
char *e,int *elen,
```

```
char *d,int *dlen,
char *p,int *plen,
char *q,int      *qlen,
char *dmp1,int *dmp1len,
char *dmq1,int *dmq1len,
char *iqmp,int *iqmplen
)
{
    int             ret=0,certl;
    char        *pp=NULL,*certp=NULL,*derCert=NULL;
    BIO             *bp=NULL;
    PKCS12      *PK12=NULL;
    EVP_PKEY    *pkey=NULL;
    X509            *cc=NULL;

    OpenSSL_add_all_algorithms();
    pp=p12;
    d2i_PKCS12(&PK12,&pp,p12Len);
    if(PK12==NULL)
    {
        printf("d2i_PKCS12 err\n");
        return -1;
    }
    ret=PKCS12_parse(PK12,pass,&pkey,&cc,NULL);
    if(ret!=1)
    {
        printf("PKCS12_parse err\n");
        return -1;
    }
    /* cert */
    certl=i2d_X509(cc,NULL);
    certp=(char *)malloc(certl+10);
    derCert=certp;
    certl=i2d_X509(cc,&certp);
    memcpy(cert,derCert,certl);
    *certlen=certl;
    free(derCert);
    /* n */
    *nlen=BN_bn2bin(pkey->pkey.rsa->n,n);
    /* e */
    *elen=BN_bn2bin(pkey->pkey.rsa->e,e);
    /* d */
    *dlen=BN_bn2bin(pkey->pkey.rsa->d,d);
    /* p */
```

```
    *plen=BN_bn2bin(pkey->pkey.rsa->p,p);
    /* q */
    *qlen=BN_bn2bin(pkey->pkey.rsa->q,q);
    /* dmp1 */
    *dmp1len=BN_bn2bin(pkey->pkey.rsa->dmp1,dmp1);
    /* dmq1 */
    *dmq1len=BN_bn2bin(pkey->pkey.rsa->dmq1,dmq1);
    /* iqmp */
    *iqmplen=BN_bn2bin(pkey->pkey.rsa->iqmp,iqmp);

    PKCS12_free(PK12);
    OPENSSL_free(PK12);
    return 0;
}
```

2) 生成 pkcs12 证书

之一：

```
#include <openssl/pkcs12.h>
#include <openssl/pkcs7.h>
int main()
{
    int                         ret,len,key_usage,iter,key_nid;
    PKCS12                      *p12;
    PKCS7                       *p7;
    STACK_OF(PKCS7)                 *safes;
    STACK_OF(PKCS12_SAFEBAG)        *bags;
    PKCS12_SAFEBAG                  *bag;
    FILE                        *fp;
    unsigned char           *buf,*p,tmp[5000];
    X509                        *cert=NULL;
    EVP_PKEY                    *pkey=NULL;

    OpenSSL_add_all_algorithms();
    p12=PKCS12_init(NID_pkcs7_data);
    /*
    p12->mac=PKCS12_MAC_DATA_new();
    p12->mac->dinfo->algor->algorithm=OBJ_nid2obj(NID_sha1);
    ASN1_STRING_set(p12->mac->dinfo->digest,"aaa",3);
    ASN1_STRING_set(p12->mac->salt,"test",4);
    p12->mac->iter=ASN1_INTEGER_new();
    ASN1_INTEGER_set(p12->mac->iter,3);
    */
    /* pkcs7 */
    bags=sk_PKCS12_SAFEBAG_new_null();
    fp=fopen("time.cer","rb");
```

```
    len=fread(tmp,1,5000,fp);
    fclose(fp);
    p=tmp;
    /* cert */
    d2i_X509(&cert,&p,len);
    bag=PKCS12_x5092certbag(cert);
    sk_PKCS12_SAFEBAG_push(bags,bag);
    /* private key */
    fp=fopen("prikey.cer","rb");
    len=fread(tmp,1,5000,fp);
    fclose(fp);
    p=tmp;
    pkey=d2i_PrivateKey(EVP_PKEY_RSA,NULL,&p,len);
    PKCS12_add_key(&bags,pkey,KEY_EX,PKCS12_DEFAULT_ITER,NID_pbe_WithSHA1
And3_Key_TripleDES_CBC,"openssl");
    p7=PKCS12_pack_p7data(bags);
    safes=sk_PKCS7_new_null();
    sk_PKCS7_push(safes,p7);
    ret=PKCS12_pack_authsafes(p12,safes);
    len=i2d_PKCS12(p12,NULL);
    buf=p=malloc(len);
    len=i2d_PKCS12(p12,&p);
    fp=fopen("myp12.pfx","wb");
    fwrite(buf,1,len,fp);
    fclose(fp);
    printf("ok\n");
    return 0;
}
```

之二：采用 PKCS12_create 函数：

```
#include <openssl/pkcs12.h>
#include <openssl/pkcs7.h>
int   main()
{
    int                         ret,len,key_usage,iter,key_nid;
    PKCS12                      *p12;
    PKCS7                       *p7;
    STACK_OF(PKCS7)                 *safes;
    STACK_OF(PKCS12_SAFEBAG)    *bags;
    PKCS12_SAFEBAG                  *bag;
    FILE                        *fp;
    unsigned char           *buf,*p,tmp[5000];
    X509                        *cert=NULL;
    EVP_PKEY                    *pkey=NULL;
```

```
        OpenSSL_add_all_algorithms();
        fp=fopen("time.cer","rb");
        len=fread(tmp,1,5000,fp);
        fclose(fp);
        p=tmp;
        /* cert */
        d2i_X509(&cert,&p,len);
        /* private key */
        fp=fopen("prikey.cer","rb");
        len=fread(tmp,1,5000,fp);
        fclose(fp);
        p=tmp;
        pkey=d2i_PrivateKey(EVP_PKEY_RSA,NULL,&p,len);
        p12=PKCS12_create("ossl","friend
name",pkey,cert,NULL,NID_pbe_WithSHA1And3_Key_TripleDES_CBC,
                NID_pbe_WithSHA1And40BitRC2_CBC,PKCS12_DEFAULT_ITER,
                                        -1,KEY_EX);
        len=i2d_PKCS12(p12,NULL);
        buf=p=malloc(len);
        len=i2d_PKCS12(p12,&p);
        fp=fopen("myp12.pfx","wb");
        fwrite(buf,1,len,fp);
        fclose(fp);
        printf("ok\n");
        return 0;
}
```

第三十一章 SSL 实现

31.1 概述

SSL 协议最先由 netscape 公司提出，包括 sslv2 和 sslv3 两个版本。当前形成标准的为了 tls 协议(rfc2246 规范)和 DTLS（rfc4347，用于支持 UDP 协议）。sslv3 和 tls 协议大致一样，只是有一些细微的差别。实际应用中，用的最多的为 sslv3。

SSL 协议能够保证通信双方的信道安全。它能提供数据加密、身份认证以及消息完整性保护，另外 SSL 协议还支持数据压缩。

SSL 协议通过客户端和服务端握手来协商各种算法和密钥。

31.2 openssl 实现

SSL 协议源码位于 ssl 目录下。它实现了 sslv2、sslv3、TLS 以及 DTLS（Datagram TLS，基于 UDP 的 TLS 实现）。ssl 实现中，对于每个协议，都有客户端实现(XXX_clnt.c)、服务端实现(XXX_srvr.c)、加密实现(XXX_enc.c)、记录协议实现(XXX_pkt.c)、METHOD 方法(XXX_meth.c)、客户端服务端都用到的握手方法实现(XXX_both.c)，以及对外提供的函数实现(XXX_lib.c)，比较有规律。

31.3 建立 SSL 测试环境

为了对 SSL 协议有大致的了解，我们可以通过 openssl 命令来建立一个 SSL 测试环境。

1） 建立自己的 CA

在 openssl 安装目录的 misc 目录下（或者在 apps 目录下），运行脚本：./CA.sh -newca（Windows 环境下运行：perl ca.pl –newca），出现提示符时，直接回车。 运行完毕后会生成一个 demonCA 的目录，里面包含了 ca 证书及其私钥。

2） 生成客户端和服务端证书申请：

openssl req -newkey rsa:1024 -out req1.pem -keyout sslclientkey.pem

openssl req -newkey rsa:1024 -out req2.pem -keyout sslserverkey.pem

3) 签发客户端和服务端证书

openssl ca -in req1.pem -out sslclientcert.pem

openssl ca -in req2.pem -out sslservercert.pem

4) 运行 ssl 服务端和客户端：

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -CAfile demoCA/cacert.pem -ssl3

openssl s_client -ssl3 -CAfile demoCA/cacert.pem

运行客户端程序后，如果正确，会打印类似如下内容:

```
SSL-Session:
  Protocol  : SSLv3
   Cipher    : DHE-RSA-AES256-SHA
      Session-ID:
A729F5845CBFFBA68B27F701A6BD9D411627FA5BDC780264131EE966D1DFD6F
```

5

Session-ID-ctx:

Master-Key:

B00EEBD68165197BF033605F348A91676E872EB48487990D8BC77022578EECC0A9789CD1F929E6A9EA259F9F9F3F9DFA

Key-Arg : None

Start Time: 1164077175

Timeout : 7200 (sec)

Verify return code: 0 (ok)

此时，输入数据然后回车，服务端会显示出来。

命令的其他选项：

a) 验证客户端证书

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -CAfile demoCA/cacert.pem -ssl3 -Verify 1

openssl s_client -ssl3 -CAfile demoCA/cacert.pem -cert sslclientcert.pem -key sslclientkey.pem

b) 指定加密套件

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -CAfile demoCA/cacert.pem -ssl3 -Verify 1

openssl s_client -ssl3 -CAfile demoCA/cacert.pem -cert sslclientcert.pem -key sslclientkey.pem -cipher AES256-SHA

其中 AES256-SHA 可用根据 openssl ciphers 命令获取，s_server 也可用指明加密套件：

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -CAfile demoCA/cacert.pem -ssl3 -Verify 1 -cipher AES256-SHA

c) 指定私钥加密口令

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -CAfile demoCA/cacert.pem -ssl3 -Verify 3 -cipher AES256-SHA -pass pass:123456

openssl s_client -ssl3 -CAfile demoCA/cacert.pem -cert sslclientcert.pem -key sslclientkey.pem -pass pass:123456

用参数 pass 给出私钥保护口令来源：

-pass file:1.txt (1.txt 的内容为加密口令 123456)；

-pass env:envname (环境变量)；

-pass fd:fdname ；

-pass stdin。

比如：

openssl s_client -ssl3 -CAfile demoCA/cacert.pem -cert sslclientcert.pem -key sslclientkey.pem -pass stdin

然后输入口令 123456 即可。

31.4 数据结构

ssl 的主要数据结构定义在 ssl.h 中。主要的数据结构有 SSL_CTX、SSL 和 SSL_SESSION。SSL_CTX 数据结构主要用于 SSL 握手前的环境准备，设置 CA 文件和目录、设置 SSL 握手中的证书文件和私钥、设置协议版本以及其他一些 SSL 握手时的选项。SSL 数据结构主要用于 SSL 握手以及传送应用数据。SSL_SESSION 中保存了主密钥、session id、读写加解密

钥、读写 MAC 密钥等信息。SSL_CTX 中缓存了所有 SSL_SESSION 信息，SSL 中包含 SSL_CTX。一般 SSL_CTX 的初始化在程序最开始调用，然后再生成 SSL 数据结构。由于 SSL_CTX 中缓存了所有的 SESSION，新生成的 SSL 结构又包含 SSL_CTX 数据，所以通过 SSL 数据结构能查找以前用过的 SESSION id，实现 SESSION 重用。

31.5 加密套件

一个加密套件指明了 SSL 握手阶段和通信阶段所应该采用的各种算法。这些算法包括：认证算法、密钥交换算法、对称算法和摘要算法等。

在握手初始化的时候，双方都会导入各自所认可的多种加密套件。在握手阶段，由服务端选择其中的一种加密套件。

OpenSSL 的 ciphers 命令可以列出所有的加密套件。openssl 的加密套件在 s3_lib.c 的 ssl3_ciphers 数组中定义。比如有：

```
/* Cipher 05 */
    {
    1,
    SSL3_TXT_RSA_RC4_128_SHA,
    SSL3_CK_RSA_RC4_128_SHA,
    SSL_kRSA|SSL_aRSA|SSL_RC4   |SSL_SHA1|SSL_SSLV3,
    SSL_NOT_EXP|SSL_MEDIUM,
    0,
    128,
    128,
    SSL_ALL_CIPHERS,
    SSL_ALL_STRENGTHS,
    }
```

其中 1 表示是合法的加密套件；SSL3_TXT_RSA_RC4_128_SHA 为加密套件的名字，SSL3_CK_RSA_RC4_128_SHA 为加密套件 ID，SSL_kRSA|SSL_aRSA|SSL_RC4 |SSL_SHA1|SSL_SSLV3 表明了各种算法，其中密钥交换采用 RSA 算法（SSL_kRSA），认证采用 RSA 算法（SSL_aRSA），对称加密算法采用 RC4 算法(SSL_RC4)，摘要采用 SHA1，采用 SSL 协议第三版本，SSL_NOT_EXP|SSL_MEDIUM 表明算法的强度。

在客户端和服务器端建立安全连接之前，双方都必须指定适合自己的加密套件。加密套件的选择可以通过组合的字符串来控制。

字符串的形式举例：ALL:!ADH:RC4+RSA:+SSLv2:@STRENGTH。

Openssl定义了4中选择符号：“＋”，“－”，“!”，“@”。其中，“＋”表示取交集；“－”表示临时删除一个算法；“!”表示永久删除一个算法；“@“表示了排序方法。

多个描述之间可以用“:”、“,”、“ ”、“;”来分开。选择加密套件的时候按照从左到的顺序构成双向链表，存放与内存中。

ALL:!ADH:RC4+RSA:+SSLv2:@STRENGTH表示的意义是：首先选择所有的加密套件（不包含eNULL，即空对称加密算法），然后在得到的双向链表之中去掉身份验证采用DH的加密套件；加入包含RC4算法并将包含RSA的加密套件放在双向链表的尾部；再将支持SSLV2的加密套件放在尾部；最后得到的结果按照安全强度进行排序。

SSL建立链接之前，客户端和服务器端用openssl函数来设置自己支持的加密套件。主要的函数有：

int SSL_set_cipher_list(SSL *s,const char *str)；

int SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str)；

比如只设置一种加密套件：

```
int    ret=SSL_set_cipher_list(ssl,"RC4-MD5");
```

如果服务端只设置了一种加密套件，那么客户端要么接受要么返回错误。加密套件的选择是由服务端做出的。

31.6 密钥信息

ssl 中的密钥相关信息包括：预主密钥、主密钥、读解密密钥及其 iv、写加密密钥及其 iv、读 MAC 密钥、写 MAC 密钥。

1) 预主密钥

预主密钥是主密钥的计算来源。它由客户端生成，采用服务端的公钥加密发送给服务端。

以 sslv3 为例，预主密钥的生成在源代码 s3_clnt.c 的 ssl3_send_client_key_exchange 函数中，有源码如下：

```
tmp_buf[0]=s->client_version>>8;
tmp_buf[1]=s->client_version&0xff;
if (RAND_bytes(&(tmp_buf[2]),sizeof tmp_buf-2) <= 0)
      goto err;
s->session->master_key_length=sizeof tmp_buf;
……
n=RSA_public_encrypt(sizeof tmp_buf,tmp_buf,p,rsa,RSA_PKCS1_PADDING);
```

此处，tmp_buf 中存放的就是预主密钥。

2) 主密钥

主密钥分别由客户端和服务端根据预主密钥、客户端随机数和服务端随机数来生成，他们的主密钥是相同的。主密钥用于生成各种密钥信息，它存放在 SESSION 数据结构中。由于协议版本不同，生成方式也不同。sslv3 的源代码中，它通过 ssl3_generate_master_secret 函数生成，tlsv1 中它通过 tls1_generate_master_secret 函数来生成。

3) 对称密钥和 MAC 密钥

对称密钥（包括 IV）和读写 MAC 密钥通过主密钥、客户端随机数和服务端随机数来生成。sslv3 源代码中，它们在 ssl3_generate_key_block 中生成，在 ssl3_change_cipher_state 中分配。

31.7 SESSION

当客户端和服务端在握手中新建了 session，服务端生成一个 session ID，通过哈希表缓存 SESSION 信息，并通过 server hello 消息发送给客户端。此 ID 是一个随机数，SSL v2 版本时长度为 16 字节，SSLv3 和 TLSv1 长度为 32 字节。此 ID 与安全无关，但是在服务端必须是唯一的。当需要 session 重用时，客户端发送包含 session id 的 clientHello 消息（无 sesion 重用时，此值为空）给服务端，服务端可用根据此 ID 来查

询缓存。session 重用可以免去诸多 SSL 握手交互，特别是客户端的公钥加密和服务端的私钥解密所带来的性能开销。session 的默认超时时间为 60*5+4 秒，5 分钟。

session 相关函数有：

1) int SSL_has_matching_session_id(const SSL *ssl, const unsigned char * id,unsigned int id_len)

SSL 中查询 session id，id 和 id_len 为输入的要查询的 session id，查询哈希表 ssl->ctx->sessions，如果匹配，返回 1，否则返回 0。

2） int ssl_get_new_session(SSL *s, int session)

生成 ssl 用的 session，此函数可用被服务端或客户端调用，当服务端调用时，传入参数 session 为 1，生成新的 session；当客户端调用时，传入参数 session 为 0，只是简单的将 session id 的长度设为 0。

3） int ssl_get_prev_session(SSL *s, unsigned char *session_id, int len)

获取以前用过的 session id，用于服务端 session 重用，本函数由服务端调用，session_id 为输入 senssion ID 首地址，len 为其长度，如果返回 1，表明要 session 重用；返回 0，表示没有找到；返回-1 表示错误。

4） int SSL_set_session(SSL *s, SSL_SESSION *session)

设置 session，本函数用于客户端，用于设置 session 信息；如果输入参数 session 为空值，它将置空 s->session；如果不为空，它将输入信息作为 session 信息。

5） void SSL_CTX_flush_sessions(SSL_CTX *s, long t)

清除超时的 SESSION，输入参数 t 指定一个时间，如果 t=0,则清除所有 SESSION，一般用 time(NULL)取当前时间。此函数调用了哈希表函数 lh_doall_arg 来处理每一个 SESSION 数据。

6） int ssl_clear_bad_session(SSL *s)

清除无效 SESSION。

31.8 多线程支持

编写 openssl 多线程程序时，需要设置两个回调函数：

CRYPTO_set_id_callback((unsigned long (*)())pthreads_thread_id);

CRYPTO_set_locking_callback((void (*)())pthreads_locking_callback);

对于多线程程序的写法，读者可以参考 crypto/threads/mttest.c，也可以查考下面的例子。

31.9 编程示例

本示例用多线程实现了一个 ssl 服务端和一个客户端。

服务端代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <errno.h>
#ifndef  _WIN32
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#else
#include <winsock2.h>
#include <windows.h>
#endif
#include "pthread.h"
#include <openssl/rsa.h>
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#define CERTF "certs/sslservercert.pem"
#define KEYF  "certs/sslserverkey.pem"
#define  CAFILE  "certs/cacert.pem"
pthread_mutex_t   mlock=PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t *lock_cs;
static long *lock_count;
#define CHK_NULL(x) if ((x)==NULL) { printf("null\n"); }
#define CHK_ERR(err,s) if ((err)==-1) { printf(" -1 \n"); }
#define CHK_SSL(err) if ((err)==-1) {   printf(" -1 \n");}
#define  CAFILE  "certs/cacert.pem"

int  verify_callback_server(int ok, X509_STORE_CTX *ctx)
{
        printf("verify_callback_server \n");
        return ok;
}

int  SSL_CTX_use_PrivateKey_file_pass(SSL_CTX *ctx,char *filename,char *pass)
{
    EVP_PKEY   *pkey=NULL;
    BIO        *key=NULL;

    key=BIO_new(BIO_s_file());
    BIO_read_filename(key,filename);
    pkey=PEM_read_bio_PrivateKey(key,NULL,NULL,pass);
    if(pkey==NULL)
    {
        printf("PEM_read_bio_PrivateKey err");
        return -1;
```

```
    }
    if (SSL_CTX_use_PrivateKey(ctx,pkey) <= 0)
    {
        printf("SSL_CTX_use_PrivateKey err\n");
        return -1;
    }
    BIO_free(key);
    return 1;
}

static int s_server_verify=SSL_VERIFY_NONE;
void * thread_main(void *arg)
{
    SOCKET s,AcceptSocket;
    WORD wVersionRequested;
    WSADATA wsaData;
    struct sockaddr_in  service;
    int    err;
    size_t          client_len;
    SSL_CTX             *ctx;
    SSL     *ssl;
    X509            *client_cert;
    char        *str;
    char        buf[1024];
    SSL_METHOD      *meth;

    ssl=(SSL *)arg;
    s=SSL_get_fd(ssl);
    err = SSL_accept (ssl);
    if(err<0)
    {
        printf("ssl accerr\n");
        return ;
    }
    printf ("SSL connection using %s\n", SSL_get_cipher (ssl));
    client_cert = SSL_get_peer_certificate (ssl);
    if (client_cert != NULL)
    {
            printf ("Client certificate:\n");
            str = X509_NAME_oneline (X509_get_subject_name (client_cert), 0, 0);
            CHK_NULL(str);
            printf ("\t subject: %s\n", str);
            OPENSSL_free (str);
            str = X509_NAME_oneline (X509_get_issuer_name    (client_cert), 0, 0);
```

```
                CHK_NULL(str);
                printf ("\t issuer: %s\n", str);
                OPENSSL_free (str);
                X509_free (client_cert);
        }
        else
                printf ("Client does not have certificate.\n");
        memset(buf,0,1024);
        err = SSL_read (ssl, buf, sizeof(buf) - 1);
        if(err<0)
        {
                printf("ssl read err\n");
                closesocket(s);
                return;
        }
        printf("get : %s\n",buf);
#if 0
        buf[err] = '\0';
        err = SSL_write (ssl, "I hear you.", strlen("I hear you."));    CHK_SSL(err);
#endif
        SSL_free (ssl);
        closesocket(s);
}

pthread_t pthreads_thread_id(void)
{
        pthread_t ret;

        ret=pthread_self();
        return(ret);
}

void pthreads_locking_callback(int mode, int type, char *file,
                int line)
{
        if (mode & CRYPTO_LOCK)
                {
                pthread_mutex_lock(&(lock_cs[type]));
                lock_count[type]++;
                }
        else
                {
                pthread_mutex_unlock(&(lock_cs[type]));
                }
```

```
}

int main ()
{
	int		err;
	int		i;
	SOCKET	s,AcceptSocket;
	WORD		wVersionRequested;
	WSADATA		wsaData;
	struct sockaddr_in	service;
	pthread_t pid;
	size_t		client_len;
	SSL_CTX		*ctx;
	SSL		*ssl;
	X509		*client_cert;
	char	*str;
	char	buf[1024];
	SSL_METHOD	*meth;

	SSL_load_error_strings();
	SSLeay_add_ssl_algorithms();
	meth = SSLv3_server_method();
	ctx = SSL_CTX_new (meth);
	if (!ctx)
	{
		ERR_print_errors_fp(stderr);
		exit(2);
	}
	if ((!SSL_CTX_load_verify_locations(ctx,CAFILE,NULL)) ||
			(!SSL_CTX_set_default_verify_paths(ctx)))
	{
		printf("err\n");
		exit(1);
	}
	if (SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM) <= 0)
	{
		ERR_print_errors_fp(stderr);
		exit(3);
	}
	if (SSL_CTX_use_PrivateKey_file_pass(ctx, KEYF, "123456") <= 0)
	{
		ERR_print_errors_fp(stderr);
		exit(4);
	}
```

```
if (!SSL_CTX_check_private_key(ctx))
{
        fprintf(stderr,"Private key does not match the certificate public key\n");
        exit(5);
}
s_server_verify=SSL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT|
                                        SSL_VERIFY_CLIENT_ONCE;
SSL_CTX_set_verify(ctx,s_server_verify,verify_callback_server);
SSL_CTX_set_client_CA_list(ctx,SSL_load_client_CA_file(CAFILE));
wVersionRequested = MAKEWORD( 2, 2 );
err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 )
{
    printf("err\n");
    return -1;
}
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(s<0) return -1;
service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr("127.0.0.1");
service.sin_port = htons(1111);
if (bind( s, (SOCKADDR*) &service, sizeof(service)) == SOCKET_ERROR)
{
    printf("bind() failed.\n");
    closesocket(s);
    return -1;
}
if (listen( s, 1 ) == SOCKET_ERROR)
    printf("Error listening on socket.\n");

printf("recv .....\n");
lock_cs=OPENSSL_malloc(CRYPTO_num_locks() * sizeof(pthread_mutex_t));
lock_count=OPENSSL_malloc(CRYPTO_num_locks() * sizeof(long));
for (i=0; i<CRYPTO_num_locks(); i++)
{
    lock_count[i]=0;
    pthread_mutex_init(&(lock_cs[i]),NULL);
}
CRYPTO_set_id_callback((unsigned long (*)())pthreads_thread_id);
CRYPTO_set_locking_callback((void (*)())pthreads_locking_callback);
while(1)
{
    struct timeval tv;
    fd_set fdset;
```

```
            tv.tv_sec = 1;
            tv.tv_usec = 0;
            FD_ZERO(&fdset);
            FD_SET(s, &fdset);
            select(s+1, &fdset, NULL, NULL, (struct timeval *)&tv);
            if(FD_ISSET(s, &fdset))
            {
                AcceptSocket=accept(s, NULL,NULL);
                ssl = SSL_new (ctx);
                CHK_NULL(ssl);
                err=SSL_set_fd (ssl, AcceptSocket);
                if(err>0)
                {
                    err=pthread_create(&pid,NULL,&thread_main,(void *)ssl);
                    pthread_detach(pid);
                }
                else
                    continue;
            }
        }
        SSL_CTX_free (ctx);
        return 0;
}
```

客户端代码如下：

```
#include <stdio.h>
#include <memory.h>
#include <errno.h>
#ifndef    _WIN32
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#else
#include <windows.h>
#endif
#include "pthread.h"
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#define    MAX_T 1000
```

```
#define    CLIENTCERT       "certs/sslclientcert.pem"
#define    CLIENTKEY  "certs/sslclientkey.pem"
#define    CAFILE         "certs/cacert.pem"
static pthread_mutex_t *lock_cs;
static long *lock_count;

pthread_t pthreads_thread_id(void)
{
      pthread_t ret;

      ret=pthread_self();
      return(ret);
}

void pthreads_locking_callback(int mode, int type, char *file,
             int line)
{
      if (mode & CRYPTO_LOCK)
            {
            pthread_mutex_lock(&(lock_cs[type]));
            lock_count[type]++;
            }
      else
            {
            pthread_mutex_unlock(&(lock_cs[type]));
            }
}

int   verify_callback(int ok, X509_STORE_CTX *ctx)
{
      printf("verify_callback\n");
      return ok;
}

int   SSL_CTX_use_PrivateKey_file_pass(SSL_CTX *ctx,char *filename,char *pass)
{
      EVP_PKEY    *pkey=NULL;
      BIO               *key=NULL;

      key=BIO_new(BIO_s_file());
      BIO_read_filename(key,filename);
      pkey=PEM_read_bio_PrivateKey(key,NULL,NULL,pass);
      if(pkey==NULL)
      {
```

```
            printf("PEM_read_bio_PrivateKey err");
            return -1;
        }
        if (SSL_CTX_use_PrivateKey(ctx,pkey) <= 0)
        {
            printf("SSL_CTX_use_PrivateKey err\n");
            return -1;
        }
        BIO_free(key);
        return 1;
}

void *thread_main(void *arg)
{
        int         err,buflen,read;
        int         sd;
        SSL_CTX             *ctx=(SSL_CTX *)arg;
        struct              sockaddr_in dest_sin;
        SOCKET      sock;
        PHOSTENT    phe;
        WORD        wVersionRequested;
        WSADATA             wsaData;
        SSL             *ssl;
        X509            *server_cert;
        char        *str;
        char        buf [1024];
        SSL_METHOD      *meth;
        FILE            *fp;

        wVersionRequested = MAKEWORD( 2, 2 );
        err = WSAStartup( wVersionRequested, &wsaData );
        if ( err != 0 )
        {
            printf("WSAStartup err\n");
            return -1;
        }
        sock = socket(AF_INET, SOCK_STREAM, 0);
        dest_sin.sin_family = AF_INET;
        dest_sin.sin_addr.s_addr = inet_addr( "127.0.0.1" );
        dest_sin.sin_port = htons( 1111 );

again:
        err=connect( sock,(PSOCKADDR) &dest_sin, sizeof( dest_sin));
        if(err<0)
```

```
        {
            Sleep(1);
            goto again;
        }
        ssl = SSL_new (ctx);
        if(ssl==NULL)
        {
            printf("ss new err\n");
            return ;
        }
        SSL_set_fd(ssl,sock);
        err = SSL_connect (ssl);
        if(err<0)
        {
            printf("SSL_connect err\n");
            return;
        }
        printf ("SSL connection using %s\n", SSL_get_cipher (ssl));
        server_cert = SSL_get_peer_certificate (ssl);
        printf ("Server certificate:\n");
        str = X509_NAME_oneline (X509_get_subject_name (server_cert),0,0);
        printf ("\t subject: %s\n", str);
        OPENSSL_free (str);
        str = X509_NAME_oneline (X509_get_issuer_name    (server_cert),0,0);
        printf ("\t issuer: %s\n", str);
        OPENSSL_free (str);
        X509_free (server_cert);
        err = SSL_write (ssl, "Hello World!", strlen("Hello World!"));
        if(err<0)
        {
            printf("ssl write err\n");
            return ;
        }
#if 0
        memset(buf,0,ONE_BUF_SIZE);
        err = SSL_read (ssl, buf, sizeof(buf) - 1);
        if(err<0)
        {
            printf("ssl read err\n");
            return ;
        }
        buf[err] = '\0';
        printf ("Got %d chars:'%s'\n", err, buf);
#endif
```

```
    SSL_shutdown (ssl);    /* send SSL/TLS close_notify */
    SSL_free (ssl);
    closesocket(sock);
}

int   main ()
{
    int         err,buflen,read;
    int         sd;

    struct              sockaddr_in dest_sin;
    SOCKET sock;
    PHOSTENT phe;
    WORD wVersionRequested;
    WSADATA wsaData;
    SSL_CTX             *ctx;
    SSL       *ssl;
    X509              *server_cert;
    char      *str;
    char      buf [1024];
    SSL_METHOD      *meth;
    int         i;
    pthread_t pid[MAX_T];

    SSLeay_add_ssl_algorithms();
    meth = SSLv3_client_method();
    SSL_load_error_strings();
    ctx = SSL_CTX_new (meth);
    if(ctx==NULL)
    {
        printf("ssl ctx new eer\n");
        return -1;
    }

    if (SSL_CTX_use_certificate_file(ctx, CLIENTCERT, SSL_FILETYPE_PEM) <= 0)
    {
        ERR_print_errors_fp(stderr);
        exit(3);
    }
    if (SSL_CTX_use_PrivateKey_file_pass(ctx, CLIENTKEY, "123456") <= 0)
    {
         ERR_print_errors_fp(stderr);
         exit(4);
     }
```

```
    lock_cs=OPENSSL_malloc(CRYPTO_num_locks() * sizeof(pthread_mutex_t));
    lock_count=OPENSSL_malloc(CRYPTO_num_locks() * sizeof(long));
    for (i=0; i<CRYPTO_num_locks(); i++)
    {
        lock_count[i]=0;
        pthread_mutex_init(&(lock_cs[i]),NULL);
    }
    CRYPTO_set_id_callback((unsigned long (*)())pthreads_thread_id);
    CRYPTO_set_locking_callback((void (*)())pthreads_locking_callback);
    for(i=0;i<MAX_T;i++)
    {
        err=pthread_create(&(pid[i]),NULL,&thread_main,(void *)ctx);
        if(err!=0)
        {
            printf("pthread_create err\n");
            continue;
        }
    }
    for (i=0; i<MAX_T; i++)
    {
        pthread_join(pid[i],NULL);
    }
    SSL_CTX_free (ctx);
    printf("test ok\n");
    return 0;
}
```

上述程序在 windows 下运行成功，采用了 windows 下的开源 pthread 库。

需要注意的是，如果多线程用 openssl,需要设置两个回调函数

CRYPTO_set_id_callback((unsigned long (*)())pthreads_thread_id);

CRYPTO_set_locking_callback((void (*)())pthreads_locking_callback);

31.10 函数

1） SSL_accept

对应于 socket 函数 accept，该函数在服务端调用，用来进行 SSL 握手。

2） int SSL_add_client_CA(SSL *ssl,X509 *x)

添加客户端 CA 名。

3） const char *SSL_alert_desc_string_long(int value)

根据错误号得到错误原因。

4） SSL_check_private_key

检查 SSL 结构中的私钥。

5） SSL_CIPHER_description

获取 SSL 加密套件描述。

6） SSL_CIPHER_get_bits

获取加密套件中对称算法的加密长度。

7） SSL_CIPHER_get_name
得到加密套件的名字。

8） SSL_CIPHER_get_version
根据加密套件获取 SSL 协议版本。

9） SSL_clear
清除 SSL 结构。

10) SSL_connect
对应于 socket 函数 connect，该函数在客户端调用，用来进行 SSL 握手。

11) SSL_CTX_add_client_CA
给 SSL_CTX 添加客户端 CA。

12) int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c)
往 SSL_CTX 添加 session。

13) SSL_CTX_check_private_key
检查私钥。

14) SSL_CTX_free
释放 SSL_CTX 空间。

15) long SSL_CTX_get_timeout(const SSL_CTX *s)
获取超时时间。

16) SSL_CTX_get_verify_callback
获取证书验证回调函数。

17) SSL_CTX_get_verify_depth
获取证书验证深度。

18）SSL_CTX_get_verify_mode
获取验证方式，这些值在 ssl.h 中定义如下：

```
#define SSL_VERIFY_NONE                 0x00
#define SSL_VERIFY_PEER                 0x01
#define SSL_VERIFY_FAIL_IF_NO_PEER_CERT 0x02
#define SSL_VERIFY_CLIENT_ONCE          0x04
```

19）SSL_get_current_cipher
获取当前的加密套件。

20）SSL_get_fd
获取链接句柄。

21）SSL_get_peer_certificate
获取对方证书。

22）XXX_client/server_method
获取各个版本的客户端和服务端的 SSL 方法。

23）SSL_read
读取数据。

24) SSL_write
发送数据。

25）SSL_set_fd
设置 SSL 的链接句柄。

26）SSL_get_current_compression

获取当前的压缩算法的 COMP_METHOD。

27）SSL_get_current_expansion

获取当前的解压算法的 COMP_METHOD。

28）SSL_COMP_get_name

获取压缩/解压算法的名称。

29）SSL_CTX_set/get_ex_data

设置/读取用户扩展数据。

30）SSL_dup

复制函数。

31）SSL_get_default_timeout

获取默认超时时间。

32）SSL_do_handshake

进行 ssl 握手。

第三十二章 Openssl 命令

32.1 概述

Openssl 命令源码位于 apps 目录下，编译的最终结果为 openssl（windows 下为 openssl.exe）。用户可用运行 openssl 命令来进行各种操作。

32.2 asn1parse

asn1parse 命令是一种用来诊断 ASN.1 结构的工具，也能用于从 ASN1.1 数据中提取数据。

用法：

openssl asn1parse [-inform PEM|DER] [-in filename] [-out filename] [-noout] [-offset number] [-length number] [-i] [-oid filename] [-strparse offset] [-genstr string] [-genconf file]

选项：

-inform PEM|DER

输入数据的格式为 DER 还是 PEM，默认为 PEM 格式。

-in filename

输入文件名,默认为标准输入。

-out filename

输出文件名，默认为标准输出，给定一个 PEM 文件，采用此选项可用生成一个 DER 编码的文件。

-noout

无输出打印。

-offset number

数据分析字节偏移量，分析数据时，不一定从头开始分析，可用指定偏移量，默认从头开始分析。

-length number

分析数据的长度，默认的长度为整个数据的长度；

-i

标记实体，加上此选项后，输出会有缩进，将一个 ASN1 实体下的其他对象缩进显示。此选项非默认选项，加上此选项后，显示更易看懂。

-dump

显示十六进制数据。非默认选项。

-dlimit number

与-dump 不同，-dump 显示所有的数据，而此选项只能显示由 number 指定数目的十六进制数据。

-oid file

指定外部的 oid 文件。

-strparse offset

此选项也用于从一个偏移量开始来分析数据，不过，与-offset 不一样。-offset

分析偏移量之后的所有数据，而-strparse 只用于分析一段数据，并且这种数据必须是 SET 或者 SEQUENCE，它只分析本 SET 或者 SEQUENCE 范围的数据。
使用示例：输入文件为一个证书的 PEM 格式文件，文件名为 server.pem，各种命令如下：

```
openssl    asn1parse c:\serverr.pem
openssl    asn1parse –in c:\server.pem –inform pem
```

上面的输出内容如下：

```
    0:d=0    hl=4 l= 489 cons: SEQUENCE
    4:d=1    hl=4 l= 338 cons: SEQUENCE
    8:d=2    hl=2 l=     1 prim: INTEGER                 :06
   11:d=2    hl=2 l=    13 cons: SEQUENCE
   13:d=3    hl=2 l=     9 prim: OBJECT                  :md5WithRSAEncryption
   24:d=3    hl=2 l=     0 prim: NULL
   26:d=2    hl=2 l=    91 cons: SEQUENCE
   28:d=3    hl=2 l=    11 cons: SET
   30:d=4    hl=2 l=     9 cons: SEQUENCE
   32:d=5    hl=2 l=     3 prim: OBJECT                  :countryName
   37:d=5    hl=2 l=     2 prim: PRINTABLESTRING     :AU
   41:d=3    hl=2 l=    19 cons: SET
   43:d=4    hl=2 l=    17 cons: SEQUENCE
   45:d=5    hl=2 l=     3 prim: OBJECT                  :stateOrProvinceName
   50:d=5    hl=2 l=    10 prim: PRINTABLESTRING     :Queensland
   62:d=3    hl=2 l=    26 cons: SET
   64:d=4    hl=2 l=    24 cons: SEQUENCE
……
```

以其中的一行进行说明：

```
13:d=3    hl=2 l=     9 prim: OBJECT                  :md5WithRSAEncryption
```

13 表示偏移量；d=3 表示此项的深度；hl=2 表示 asn1 头长度；l=9 表示内容长度；prim:OBJECT 表示 ASN1 类型；md5WithRSAEncryption 表示 oid。
示例如下：

```
openssl    asn1parse –in c:\server.pem –out c:\server.cer
```

此命令除了显示上面内容外，并生成一个 der 编码的文件。

```
openssl    asn1parse –in c:\server.pem –i
```

此命令显示上面的内容，但是有缩进。

```
openssl    asn1parse –in c:\server.pem –i –offset 26
```

此命令从偏移量 26 开始分析，到结束。注意，26 从前面命令的结果得到。

```
openssl    asn1parse –in c:\server.pem –i –offset 13 –length 11
```

此命令从偏移量 13 进行分析，分析长度为 11

```
openssl    asn1parse –in c:\server.pem –i –dump
```

分析时，显示 BIT STRING 等的十六进制数据；

```
openssl    asn1parse –in c:\server.pem –i –dlimit 10
```

分析时，显示 BIT SRING 的前 10 个十六进制数据。

```
openssl    asn1parse –in c:\server.pem –i –strparse 11
```

此令分析一个 SEQUENCE。

openssl asn1parse –in c:\server.pem –i –strparse 11 –offset 2 –length 11

根据偏移量和长度分析。

32.3 dgst

dgst 用于数据摘要。

用法：

openssl dgst [-md5|-md4|-md2|-sha1|-sha|-mdc2|-ripemd160|-dss1] [-c] [-d] [-hex] [-binary] [-out filename] [-sign filename] [-passin arg] [-verify filename] [-prverify filename] [-signature filename] [file...]

选项：

-d

打印调试信息。

-sign privatekeyfile

用 privatekeyfile 中的私钥签名。

-verify publickeyfile

用 publickeyfile 中的公钥验证签名。

-prverify privatekeyfile

用 privatekeyfile 中的私钥验证签名。

-keyform PEM | ENGINE

密钥格式，PEM 格式或者采用 Engine。

-hex

显示 ASCII 编码的十六进制结果，默认选项。

-binary

显示二进制数据。

-engine e

采用引擎 e 来运算。

-md5

默认选项，用 md5 进行摘要。

-md4

用 md4 摘要。

-md2

用 md2 摘要。

-sha1

用 sha1 摘要。

-sha

用 sha 摘要。

-sha256

用-sha256 摘要。

-sha512

用 sha512 摘要。

-mdc2

用 mdc2 摘要。

-ripemd160

用 ripemd160 摘要。

示例：

openssl dgst c:\server.pem

运行此命令后文件的 md5 值摘要结果会在屏幕打印出来，此结果为摘要结果转换为 ASCII 码后的值：

MD5(c:\server.cer)= 4ace36445f5ab4bbcc2b9dd55e2f0e3a

openssl dgst –binary c:\server.pem

结果为二进制乱码。

openssl dgst –hex –c c:\server.pem

结果由：分开，如下：

MD5(c:\server.cer)= 4a:ce:36:44:5f:5a:b4:bb:cc:2b:9d:d5:5e:2f:0e:3a

openssl dgst –sign privatekey.pem –sha1 –keyform PEM –c c:\server.pem

将文件用 sha1 摘要，并用 privatekey.pem 中的私钥签名。

32.4 gendh

此命令用于生成 DH 参数。

选项：

-out file

输出结果到 file 指定的文件；如果不指定，结果显示在屏幕屏幕上；

-2

将 2 作为生成值，此为默认值；

-5

将 5 作为生成值；

-rand

指定随机数文件；

-engine e

采用 Engine 生成；

示例：

openssl gendh

openssl gendh -5 -out dh.pem 1024

32.5 passwd

生成各种口令密文。

用法：

openssl passwd [-crypt] [-1] [-apr1] [-salt string] [-in file] [-stdin] [-noverify] [-quiet] [-table] {password}

选项：

-crypt

默认选项，生成标准的 unix 口令密文。

-1

md5 口令密文。

-apr1

Apache md5 口令密文。

-salt string

加入由 string 指定的 salt。

-in file

输入的口令文件，默认从 stdin 中读取。

-stdin

默认选项，从 stdin 读取口令。

-noverify

用户输入口令时，不验证。

-quiet

无警告。

-table

用户输入的口令和结果用缩进隔开。

-reverse

用户输入的口令和结果用缩进隔开，输出内容颠倒顺序。

示例：

(1) openssl passwd

(2) openssl passwd -1

(3) openssl passwd -1 –noverify

(4) openssl passwd –table –reverse -noverify

32.6 rand

生成随机数。

用法：

openssl rand [-out file] [-rand file(s)] [-base64] num

选项：

-out file

结果输出到 file 中。

-engine e

采用 engine 来生成随机数。

-rand file

指定随机数种子文件。

-base64

输出结果为 BASE64 编码数据。

num

随机数长度。

示例：

(1) openssl rand –base64 100

(2) openssl rand –base64 –out myr.dat 100

32.7 genrsa

生成 RSA 密钥。

用法：

openssl genrsa [-out filename] [-passout arg] [-des] [-des3] [-idea] [-f4] [-3] [-rand file(s)] [-engine id] [numbits]

选项：

-des

以 des cbc 模式加密密钥；

-des3

以 3des cbc 模式加密密钥；

-idea

以 idea cbc 模式加密密钥；

-aes128, -aes192, -aes256

cbc 模式加密密钥；

-out file

输出文件；

-f4

指定 E 为 0x1001；

-3

指定 E 为 3；

-engine e

指定 engine 来生成 RSA 密钥；

-rand file

指定随机数种子文件；

numbits

密钥长度，如果不指定默认为 512。

示例：

openssl genrsa -des3 –out prikey.pem –f4 1024

32.8 req

req 命令主要用于生成和处理 PKCS#10 证书请求。

用法：

openssl req [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out filename] [-passout arg] [-text] [-pubkey] [-noout] [-verify] [-modulus] [-new] [-rand file(s)] [-newkey rsa:bits] [-newkey dsa:file] [-nodes] [-key filename] [-keyform PEM|DER] [-keyout filename] [-[md5|sha1|md2|mdc2]] [-config filename] [-subj arg] [-multivalue-rdn] [-x509] [-days n] [-set_serial n] [-asn1-kludge] [-newhdr] [-extensions section] [-reqexts section] [-utf8] [-nameopt] [-batch] [-verbose] [-engine id]

选项：

-out

指定输出文件名。

-outform DER|PEM

指定输出格式。

-newkey rsa:bits

用于生成新的 rsa 密钥以及证书请求。如果用户不知道生成的私钥文件名称，默认采用 privkey.pem，生成的证书请求。如果用户不指定输出文件(-out)，则将证书请求文件打印在屏幕上。生成的私钥文件可以用-keyout 来指定。生成过程中需要用户输入私钥的保护口令以及证书申请中的一些信息。

-new

生成新的证书请求以及私钥，默认为 1024 比特。

-rand

指定随机数种子文件，比如有随机数文件 rand.dat，用户输入：-rand file:rand.dat。

-config file

指定证书请求模板文件，默认采用 openssl.cnf，需另行指定时用此选项。配置的写法可以参考 openssl.cnf，其中有关于生成证书请求的设置。

-subj arg

用于指定生成的证书请求的用户信息，或者处理证书请求时用指定参数替换。生成证书请求时，如果不指定此选项，程序会提示用户来输入各个用户信息，包括国名、组织等信息，如果采用此选择，则不需要用户输入了。比如：-subj /CN=china/OU=test/O=abc/CN=forxy，注意这里等属性必须大写。

-multivalue-rdn

当采用-subj arg 选项时，允许多值 rdn 名，比如 arg 参数写作：/CN=china/OU=test/O=abc/UID=123456+CN=forxy。

-reqexts ..

设置证书请求的扩展项，被设置的扩展项覆盖配置文件所指定的扩展项。

-utf8

输入字符为 utf8 编码，默认输入为 ASCII 编码。

-batch

不询问用户任何信息（私钥口令除外），采用此选项生成证书请求时，不询问证书请求当各种信息。

-noout

不输出证书请求。

-newhdr

在生成的 PME 证书请求文件的头尾添加“NEW”，有些软件和 CA 需要此项。

-engine e

指定硬件引擎。

-keyout

指定生成的私钥文件名称。

示例：

openssl req –new

openssl req –new –config myconfig.cnf

openssl req –subj /CN=cn/O=test/OU=abc/CN=forxy

openssl req -newkey rsa:1024

openssl req -newkey rsa:1024 -out myreq.pem –keyout myprivatekey.pem

openssl req -newkey rsa:1024 -out myreq.pem -keyout myprivatekey.pem -outform

DER

-subject

输出证书请求者信息。

-modulus

输出证书请求的模数。

示例：openssl req -in myreq.pem -modulus –subject。

-pubkey

获取证书请求中的公钥信息。

示例：

openssl req -in myreq.pem -pubkey -out pubkey.pem

-in filename

输入的证书请求文件。

-text

打印证书请求或自签名证书信息。

-verify

验证证书请求。

示例：

openssl req -in zcp.pem -verify

-inform DER|PEM

指定输入的格式是 DEM 还是 DER。

-x509

生成自签名证书。

-extensions ..

设置证书扩展项，设置的扩展项优先于配置文件指定的扩展项。

-set_serial

设置生成证书的证书序列号，比如 -set_serial 100 或 -set_serial 0x100

-[md5|md4|md2|sha1|mdc2]

生成自签名证书时，指定摘要算法。

-passin

用户将私钥的保护口令写入一个文件，采用此选项指定此文件，可以免去用户输入口令的操作。比如用户将口令写入文件“pwd.txt”，输入的参数为：-passin file:pwd.txt。

-days

指定自签名证书的有效期限。

示例：

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem -days 300

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem -days 300 -text

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem -days 300 -text -md5

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem -days 300 -text -md5 –set_serial 0x100

openssl req -in myreq.pem -x509 -key myprivatekey.pem -out mycert.pem -days 300

-text -md5 –passin file:pwd.txt

这里的 myreq.pem 为 PEM 格式的文件，可以用-inform 指定其格式。

-out filename

要输出的文件名。

-text

将 CSR 文件里的内容以可读方式打印出来。

-noout

不要打印 CSR 文件的编码版本信息。

-modulus

将 CSR 里面的包含的公共米要的系数打印出来。

-verify

检验请求文件里的签名信息。

示例：

生成 ECC 证书请求：

openssl ecparam -genkey -name secp160r1 -out ec160.pem

openssl req -newkey ec:ec160.pem

注意，如果由 ecparam 中的 -name 指定的密钥长度太短，将不能生成请求。因为 md5 或者 sha1 等的摘要长度对它来说太长了。

32.9 x509

X509 命令是一个多用途的证书工具。它可以显示证书信息、转换证书格式、签名证书请求以及改变证书的信任设置等。

用法：

openssl x509 [-inform DER|PEM|NET] [-outform DER|PEM|NET] [-keyform DER|PEM] [-CAform DER|PEM] [-CAkeyform DER|PEM] [-in filename] [-out filename] [-serial] [-hash] [-subject_hash] [-issuer_hash] [-subject] [-issuer] [-nameopt option] [-email] [-startdate] [-enddate] [-purpose] [-dates] [-modulus] [-fingerprint] [-alias] [-noout] [-trustout] [-clrtrust] [-clrreject] [-addtrust arg] [-addreject arg] [-setalias arg] [-days arg] [-set_serial n] [-signkey filename] [-x509toreq] [-req] [-CA filename] [-CAkey filename] [-CAcreateserial] [-CAserial filename] [-text] [-C] [-md2|-md5|-sha1|-mdc2] [-clrext] [-extfile filename] [-extensions section] [-engine id]

选项：

-inform DER|PEM|NET

指定输入文件的格式，默认为 PEM 格式。

-outform DER|PEM|NET

指定输出文件格式，默认为 PEM 格式。

-keyform

指定私钥文件格式，默认为 PEM 格式。

-CAform

指定 CA 文件格式，默认为 PEM 格式。

-CAkeyform

指定 CA 私钥文件格式，默认为 PEM 格式。

-in filename

指定输入文件名。

-out filename

指定输出文件名。

-passin

指定私钥保护密钥来源，参考 req 说明，比如：-passin file:pwd.txt。

-serial

显示证书的序列号。

-subject_hash

显示持有者的摘要值。

-issuer_hash

显示颁发者的摘要值。

-hash

显示证书持有者的摘要值，同-subject_hash。

-subject

显示证书持有者 DN。

-issuer

显示证书颁发者 DN。

-email

显示 email 地址。

-enddate

显示证书到期时间。

-startdate

显示证书的起始有效时间。

-purpose

显示证书用途。

-dates

显示证书的有效期。

-modulus

显示公钥模数。

-pubkey

输出公钥。

-fingerprint

打印证书微缩图。

-alias

显示证书别名。

-noout

不显示信息。

-ocspid

显示持有者和公钥的 OCSP 摘要值。

-trustout

输出可信任证书。

-clrtrust

清除证书附加项里所有有关用途允许的内容。

-clrreject

清除证书附加项里所有有关用途禁止的内容。

-addtrust arg

添加证书附加项里所有有关用途允许的内容。

-addreject arg

添加证书附加项里所有有关用途禁止的内容。

-setalias arg

设置证书别名。

-days arg

设置证书有效期。

-checkend arg

显示证书在给定的 arg 秒后是否还有效。

-signkey filename

指定自签名私钥文件。

-x509toreq

根据证书来生成证书请求，需要指定签名私钥，如：

openssl x509 -in ca.pem -x509toreq -signkey key.pem

-req

输入为证书请求，需要进行处理。

-CA arg

设置 CA 文件，必须为 PEM 格式。

-CAkey arg

设置 CA 私钥文件，必须为 PEM 格式。

-CAcreateserial

如果序证书列号文件，则生成。

-CAserial arg

由 arg 指定序列号文件。

-set_serial

设置证书序列号。

-text

打印证书信息。

-C

用 C 语言格式显示信息。

-md2|-md5|-sha1|-mdc2

指定使用的摘要算法，缺省为 MD5。

-extfile filename

指定包含证书扩展项的文件名，如果没有，那么生成的证书将没有任何扩展项。

-clrext

删除证书所有的扩展项。当一个证书由另外一个证书生成时，可用此项。

-nameopt option

指定打印名字时采用的格式。

-engine e

采用硬件引擎 e。

-certopt arg

当采用-text 显示时，设置是否打印哪些内容，arg 可用是：compatible、no_header、no_version、no_extensions 和 ext_parse 等等，详细信息请参考 x509 命令的帮助文档。

示例：

openssl x509 -in cert.pem -noout -subject -nameopt RFC2253

openssl x509 -in cert.pem -inform PEM -out cert.der -outform DER

openssl x509 -req -in req.pem -extfile openssl.cnf -extensions v3_usr -CA cacert.pem -CAkey key.pem –Cacreateserial

32.10 version

version 命令用来打印版本以及 openssl 其他各种信息。

用法：

version -[avbofp]

选项：

-a

打印所有信息。

-v

打印当前 openssl 的版本信息。

-b

打印当前版本的 openssl 是什么时候编译完成的。

-o

建立库时的各种与加密算法和机器字节有关的信息。

-f

编译 openssl 的编译选项。

-p

平台信息。

32.11 speed

speed 命令用于测试库的性能。

用法：

openssl speed [-engine id] [md2] [mdc2] [md5] [hmac] [sha1] [rmd160] [idea-cbc] [rc2-cbc] [rc5-cbc] [bf-cbc] [des-cbc] [des-ede3] [rc4] [rsa512] [rsa1024] [rsa2048] [rsa4096] [dsa512] [dsa1024] [dsa2048] [idea] [rc2] [des] [rsa] [blowfish]

选项：

-engine id

设置硬件引擎 id。

-elapsed

测量采用实时时间，不是所用 CPU 时间，两者时间差异较大。

-mr

生成机器可读显示。

-multi n

并行允许 n 个测试。

示例：

openssl speed md5

32.12 sess_id

sess_id 为 SSL/TLS 协议的 session 处理工具。

用法：

openssl sess_id [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-out filename] [-text] [-noout] [-context ID]

选项：

-inform DER|PEM

指定输入格式是 DER 还是 PEM；

-outform DER|PEM

指定输出格式是 DER 还是 PEM；

-in filename

session 信息的文件名；

-out filename

输出 session 信息的文件名；

-text

打印信息；

-cert

打印数字证书；

如果用户需要分析 session 信息，需要有一个 session 文件，用户可在程序中将 SSL_SESSION 写入文件，然后用本命令来分析。

32.13 s_server

s_server 是 openssl 提供的一个 SSL 服务程序。使用此程序前，需要生成各种证书，可参考：第 31 章中第建立 SSL 测试环境一节。本命令可以用来测试 ssl 客户端，比如各种浏览器的 https 协议支持。

用法：

openssl s_server [-accept port] [-context id] [-verify depth] [-Verify depth] [-cert filename] [-key keyfile] [-dcert filename] [-dkey key-file] [-dhparam filename] [-nbio] [-nbio_test] [-crlf] [-debug] [-msg] [-state] [-CApath directory] [-CAfile filename] [-nocert] [-cipher cipherlist] [-quiet] [-no_tmp_rsa] [-ssl2] [-ssl3] [-tls1] [-no_ssl2] [-no_ssl3] [-no_tls1] [-no_dhe] [-bugs] [-hack] [-www] [-WWW] [-HTTP] [-engine id] [-rand file(s)]

选项：

-accept arg

监听的 TCP 端口，缺省为 4433。

-context arg

设置 ssl 上下文，不设置时采用缺省值。

-cert certname

服务使用的证书文件名。

-certform arg

证书文件格式，默认为 PEM。

-keyform arg

私钥文件格式，默认为 PEM。

-pass arg

私钥保护口令来源。

-msg

打印协议内容。

-timeout

设置超时。

-key keyfile

服务使用的私钥文件，由-cert 指定的文件既可以包含证书，也可用包含私钥，此时，就不需要此选项。

-no_tmp_rsa

不生成临时 RSA 密钥。

-verify depth

设置证书验证深度。

-Verify arg

如果设置了此项为 1，服务端必须验证客户端身份。

-CApath path

设置信任 CA 文件所在路径，此路径中的 ca 文件名采用特殊的形式：xxx.0。其中 xxx 为 CA 证书持有者的哈希值，可通过 x509 -hash 命令获得。

-CAfile file

指定 CA 证书文件。

-state

打印 SSL 握手状态。

-debug

打印更多的信息。

-nbio

不采用 BIO。

-quiet

不打印输出信息。

-ssl2, -ssl3, -tls1

只采用某一种协。;

-no_ssl2, -no_ssl3, -no_tls1

不采用某种协议。

-www

返回给用户一个网页，内容为 SSL 握手的一些内容。

-WWW -HTTP

将某个文件作为网页发回客户端，例如 client 的 URL 请求是 https://myhost/page.html ，则把 ./page.html 发回给 client。如果不设置-www、-WWW 、-HTTP,客户端在终端输入任何字符，服务端都会响应同样的字符给客户

端。

-rand file:file:...

设置随机数种子文件，SSL 协议握手中会生成随机数，比如 clienthello 和 serverhello 消息。

-crlf

将用户在终端输入的换行回车转化成/r/n。

连接命令，这些输入不是程序运行选项，在程序运行过程中输入，如下：

q

中断当前连接，但不关闭服务。

Q

中断当前连接，退出程序。

r

重新协商。

R

重新协商，并且要求客户端证书。

P

在 TCP 层直接送一些明文，造成客户端握手错误并断开连接。

S

打印缓存的 SESSION 信息。

32.14 s_client

s_client 为一个 SSL/TLS 客户端程序，与 s_server 对应，它不仅能与 s_server 进行通信，也能与任何使用 ssl 协议的其他服务程序进行通信。

用法：

openssl s_client [-connect host:port>] [-verify depth] [-cert filename]
[-key filename] [-CApath directory] [-CAfile filename] [-reconnect]
[-pause] [-showcerts] [-debug] [-msg] [-nbio_test] [-state] [-nbio]
[-crlf] [-ign_eof] [-quiet] [-ssl2] [-ssl3] [-tls1] [-no_ssl2]
[-no_ssl3] [-no_tls1] [-bugs] [-cipher cipherlist] [-engine id] [-rand file(s)]

选项：

-host host

设置服务地址.

-port port

设置服务端口，默认为 4433。

-connect host:port

设置服务地址和端口。

-verify depth

设置证书验证深度。

-cert arg

设置握手采用的证书。

-certform arg

设置证书格式，默认为 PEM。

-key arg

指定客户端私钥文件名，私钥可以与证书存放同一个文件中，这样，只需要-cert选项就可以了，不需要本选项。

-keyform arg

私钥格式，默认为 PEM。

-pass arg

私钥保护口令来源，比如：-pass file:pwd.txt，将私钥保护口令存放在一个文件中，通过此选项来指定，不需要用户来输入口令。

-CApath arg

设置信任 CA 文件所在路径，此路径中的 ca 文件名采用特殊的形式：xxx.0，其中 xxx 为 CA 证书持有者的哈希值，它通过 x509 -hash 命令获得。

-CAfile arg

指定 CA 文件名。

-reconnect

重新连接，进行 session 重用。

-pause

每当读写数据时，sleep 1 秒。

-showcerts

显示证书链。

-debug

额外输出信息。

-msg

打印协议消息。

-nbio_test

更多协议测试。

-state

打印 SSL 状态。

-nbio

不采用 BIO。

-quiet

不显示客户端数据。

-ssl2、-ssl3、-tls1、-dtls1

指定客户端协议。

-no_tls1/-no_ssl3/-no_ssl2

不采用某协议。

-bugs

兼容老版本服务端的中的 bug。

-cipher

指定加密套件。

-starttls protocol

protocol 可以为 smtp 或 pop3，用于邮件安全传输。

-rand file:file:...

设置随机数种子文件，SSL 协议握手中会生成随机数，比如 clienthello 和 serverhello 消息中的随机数。

-crlf

将用户在终端输入的换行回车转化成/r/n。

32.15 rsa

Rsa 命令用于处理 RSA 密钥、格式转换和打印信息。

用法：

openssl rsa [-inform PEM|NET|DER] [-outform PEM|NET|DER] [-in filename] [-passin arg] [-out filename] [-passout arg] [-sgckey] [-des] [-des3] [-idea] [-text] [-noout] [-modulus] [-check] [-pubin] [-pubout] [-engine id]

选项：

-inform DER|PEM|NET

指定输入的格式，NET 格式是与老的 Netscape 服务以及微软的 IIS 兼容的一种不太安全的格式。

-outform DER|PEM|NET

指定输出格式。

-in filename

输入文件名。

-passin arg

私钥保护密钥来源，比如：-passin file:pwd.txt。

-out filename

输出的文件名。

-des|-des3|-idea

指定私钥保护加密算法。

-text

打印密钥信息。

-noout

不打印任何信息。

-modulus

打印密钥模数。

-pubin

表明输入文件为公钥，默认的输入文件是私钥。

-pubout

表明输出文件为公钥。

-check

检查 RSA 私钥。

-engine id

指明硬件引擎。

示例：

生成明文私钥文件：

openssl genrsa -out key.pem

转换为 DER 编码：

openssl rsa -in key.pem -outform der -out key.der

将明文私钥文件转换为密码保护：

openssl rsa -inform der -in key.der -des3 -out enckey.pem

将公钥写入文件：
openssl rsa -in key.pem -pubout -out pubkey.pem
打印公钥信息：
openssl rsa -pubin -in pubkey.pem –text -modulus
显示私钥信息，保护密钥写在 pwd.txt 中
openssl rsa -in enckey.pem –passin file:pwd.txt

32.16 pkcs7

pkcs7 命令用于处理 DER 或者 PEM 格式的 pkcs#7 文件。
用法：
openssl pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-out filename] [-print_certs] [-text] [-noout] [-engine id]
选项：
-inform DER|PEM
输入文件格式，默认为 PEM 格式。
-outform DER|PEM
输出文件格式，默认为 PEM 格式。
-in filename
输入文件名，默认为标准输入。
-out filename
输出文件名, 默认为标准输出。
-print_certs
打印证书或 CRL 信息，在一行中打印出持有者和颁发者。
-text
打印证书相信信息。
-noout
不打印信息。
-engine id
指定硬件引擎。
示例：
把一个 PKCS#7 文件从 PEM 格式转换成 DER 格式：
openssl pkcs7 -in file.pem -outform DER -out file.der
打印文件所有证书
openssl pkcs7 -in file.pem -print_certs -out certs.pem

32.17 dsaparam

dsaparam 命令用于生成和操作 dsa 参数。
用法：
openssl dsaparam [-inform DER|PEM] [-outform DER|PEM] [-in filename] [-out filename] [-noout] [-text] [-C] [-rand file(s)] [-genkey] [-engine id] [numbits]
选项：

-inform DER|PEM
输入文件格式。
-outform DER|PME
输出文件格式。
-in filename
输入文件名。
-out filename
输出文件名。
-nout
不打印输出信息。
-text
打印内容信息。
-C
以 C 语言格式打印信息。
-rand file(s)
指定随机数种子文件，多个文件用冒号分开。
-genkey
生成 dsa 密钥。
-engine id
指定硬件引擎。
number
生成密钥时指定密钥大小。
示例：
生成 DSA 密钥：
openssl dsaparam -genkey 512 -out dsa.pem
打印密钥信息：
openssl dsaparam -in dsa.pem -text
openssl dsaparam -in dsa.pem -C

32.18 gendsa

gendsa 根据 DSA 密钥参数生成 DSA 密钥，dsa 密钥参数可用 dsaparam 命令生成。
用法：
openssl gendsa [-out filename] [-des] [-des3] [-idea] [-rand file(s)] [-engine id] [paramfile]
选项：
-out filename
指定输出文件。
-des|-des3|-idea|-aes128|-aes192|-aes256
指定私钥口令保护算法，如果不指定，私钥将被明文存放。
-rand file(s)
指定随机数种子文件，多个文件用冒号分开。
-engine id
指定硬件引擎。
paramfile

指定使用的 DSA 密钥参数文件。

示例：

生成 DSA 参数：

openssl dsaparam -genkey 512 -out dsaparam.pem

生成 DSA 密钥：

openssl gendsa -des3 -out encdsa.pem dsaparam.pem

32.19 enc

enc 为对称加解密工具，还可以进行 base64 编码转换。

用法：

openssl enc -ciphername [-in filename] [-out filename] [-pass arg] [-e] [-d] [-a] [-A] [-k password] [-kfile filename] [-K key] [-iv IV] [-p] [-P] [-bufsize number] [-nopad] [-debug]

选项：

-ciphername

对称算法名字，此命令有两种适用方式：-ciphername 方式或者省略 enc 直接用 ciphername。比如，用 des3 加密文件 a.txt：

openssl enc -des3 -e -in a.txt -out b.txt

openssl des3 -e -in a.txt -out b.txt

-in filename

输入文件，默认为标准输入。

-out filename

输出文件，默认为标准输出。

-pass arg

输入文件如果有密码保护，指定密码来源。

-e

进行加密操作，默认操作。

-d

进行解密操作。

-a

当进行加解密时，它只对数据进行运算，有时需要进行 base64 转换。设置此选项后，加密结果进行 base64 编码；解密前先进行 base64 解码。

-A

默认情况下，base64 编码结果在文件中是多行的。如果要将生成的结果在文件中只有一行，需设置此选项；解密时，必须采用同样的设置，否则读取数据时会出错。

-k password

指定加密口令，不设置此项时，程序会提示用户输入口令。

-kfile filename

指定口令存放的文件。

-K key

输入口令是 16 进制的。

-iv IV

初始化向量，为 16 进制。

比如：openss des-cbc -in a.txt -out b.txt -a -A -K 1111 -iv 2222

-p

打印出使用的 salt、口令以及初始化向量 IV。

-P

打印使用的 salt、口令以及 IV，不做加密和解密操作。

-bufsize number

设置 I/O 操作的缓冲区大小，因为一个文件可能很大，每次读取的数据是有限的。

-debug

打印调试信息。

进行 base64 编码时，将 base64 也看作一种对称算法。

32.20 ciphers

显示支持的加密套件。

用法：

openssl ciphers [-v] [-ssl2] [-ssl3] [-tls1] [cipherlist]

选项：

-v

详细列出所有加密套件。包括 ssl 版本、密钥交换算法、身份验证算法、对称算法、摘要算法以及该算法是否可以出口。

-ssl3

只列出 SSLv3 使用的加密套件。

-ssl2

只列出 SSLv2 使用的加密套件。

-tls1

只列出 TLSv1 使用的加密套件。

cipherlist

此项为一个规则字符串，用此项能列出所有符合规则的加密套件，如果不加-v 选项，它只显示各个套件名字；

示例：

openssl ciphers -v 'ALL:eNULL'

openssl ciphers -v '3DES:+RSA'

32.21 CA

ca 命令是一个小型 CA 系统。它能签发证书请求和生成 CRL。它维护一个已签发证书状态的文本数据库。

用法：

openssl ca [-verbose] [-config filename] [-name section] [-gencrl]
[-revoke file] [-crl_reason reason] [-crl_hold instruction] [-crl_com
promise time] [-crl_CA_compromise time] [-subj arg] [-crldays days]
[-crlhours hours] [-crlexts section] [-startdate date] [-enddate date]

[-days arg] [-md arg] [-policy arg] [-keyfile arg] [-key arg] [-passin arg] [-cert file] [-in file] [-out file] [-notext] [-outdir dir] [-infiles] [-spkac file] [-ss_cert file] [-preserveDN] [-noemailDN] [-batch] [-msie_hack] [-extensions section] [-extfile section] [-engine id] B[-utf8] [-multivalue-rdn]

选项：

-verbose

打印附加信息。

-config

指定配置文件，此配置文件中包含了证书存放路径、私钥和生成证书控制等信息。如果默认安装 openssl，配置文件在/usr/local/ssl/路径下。我们可以先用 apps 目录下的 CA.sh 或者 CA.pl 脚本来 建立环境：sh CA.sh -newca,输入后回车就会生成一个 demonCA 的目录。

-name section

替换配置文件指定的 default_ca 所表示的内容。比如有 openssl.cnf 配置如下：

```
[ ca ]
default_ca      = CA_default
[ CA_default ]
dir             = ./demoCA
certs           = $dir/certs
crl_dir         = $dir/crl
database        = $dir/index.txt

[ my_defaultCA ]
dir             = ./demoCA1
certs           = $dir/certs
crl_dir         = $dir/crl
database        = $dir/index.txt
```

此时用户也可以采用选项来指定 default_ca 的值： -name my_defaultCA;

-gencrl

生成 CRL 文件。

-revoke file

撤销证书，file 文件中包含了证书。

-crl_reason reason

设置 CRLv2 撤销原因，原因可以为：unspecified、keyCompromise、CACompromise、affiliationChanged、superseded、cessationOfOperation、certificateHold 和 removeFromCRL。这些原因区分大小写。

-crl_hold instruction

当 crl 撤销原因为 certificateHold 时(证书挂起)，采用此项来指定用户行为。instruction 的值可以是：holdInstructionNone、holdInstructionCallIssuer 和 holdInstructionReject。比如用选项： -crl_hold holdInstructionReject 时，指明用户必须拒绝挂起的证书。

-crl_compromise time

当 crl 撤销原因为 keyCompromise 时(密钥泄露),设置密钥泄露时间 time。Time

采用通用时间格式：YYYYMMDDHHMMSSZ。

-crl_CA_compromise time

当 crl 撤销原因为 CACompromise 时(CA 被破坏),设置其时间，格式同-crl_compromise time。

-subj arg

持有者参数，如/CN=cn/O=test/OU=t/cn=forxy，忽略空格已经\后的字符。

-crldays days

设置下次 CRL 发布时间，days 为下次发布时间距现在的天数。

-crlhours hours

设置下次 CRL 发布时间，hours 为下次发布时间距现在的小时数。

-crlexts section

指定 CRL 扩展项。section 为配置文件中的段，如果不提供 crl 扩展项段，则生成第一版本的 crl，如果提供，则生成第二版本的 crl。

-startdate date

设置证书生效起始时间，采用 UTCTime 格式：YYMMDDHHMMSSZ。

-enddate date

设置证书失效时间，采用 UTCTime 格式：YYMMDDHHMMSSZ。

-days arg

设置证书有效期，arg 为天数。

-md arg

设置摘要算法：md5、sha、sha1 或 mdc2。

-policy arg

指定 CA 策略，arg 为配置文件中的策略段，比如配置文件有如下信息：

```
[ ca ]
policy                  = policy_match
[ policy_match ]
countryName             = match
stateOrProvinceName     = match
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
[ policy_anything ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

此时，采用的是 policy_match 策略(由 policy=policy_match 指定)，用户可以设置采用 policy_anything： -policy policy_anything。

-keyfile arg

指定签发证书的私钥文件。

-key arg

指定私钥解密口令。

-passin arg

指定私钥口令来源。

-cert file

指定 CA 文件。

-in file

输入的证书请求文件。

-out file

输出文件名。

-notext

在证书文件中，不输出文本格式的证书信息。

-outdir dir

设置输出路径。

-infiles ...

处理多个证书请求文件，此选项必须放在最后，此选项后的多个输入都被当作是证书请求文件。

-ss_cert file

指定需要由 CA 签发的自签名证书。

-preserveDN

证书中的 DN 顺序由配置文件来决定，如果设置此选项，则证书中 DN 的顺序与请求文件一致。

-noemailDN

如果证书请求者 DN 中包含邮件项，生成的证书也将会在持有者 DN 中包含。但是，较好的方式是将它放入到扩展项(altName)中去，如果设置了此选项，则进行这种操作。

-batch

批处理，不询问用户信息。

-msie_hack

支持很老的 IE 证书请求。

-extensions section

如果没有通过-extfile 选项指定扩展项信息，section 为配置文件中与扩展项有关的段，签发证书时添加 section 指定的扩展项(默认采用 x509_extensions)，如果不指定扩展，将生成第一版本的数字证书。

-engine id

指定硬件引擎。

-utf8

表明任何输入都必须是 utf8 编码(用户的终端输入和配置文件),默认为 ASCII 编码。

-multivalue-rdn

当采用 -subj 参数时，支持多值 RDN，比如：DC=org/DC=OpenSSL/DC=users/UID=123456+CN=John Doe。

示例：下面所有命令在 apps 目录下运行：

1） 建 CA

在 apps 目录下

sh ca.sh -newca 生成新 CA，遇到提示，直接回车；

2) 生成证书请求

openssl req -new -out req.pem -keyout key.pem

openssl req -new -out req2.pem -keyout key2.pem

3) 签发证书

openssl ca -config /usr/local/ssl/openssl.cnf -name CA_default -days 365 -md sha1 -policy policy_anything -cert demoCA/cacert.pem -in req.pem -out cert1.pem -preserveDN -noemailDN -subj /CN=CN/O=JS/OU=WX/cn=myname -extensions myexts

openssl.cnf 中相关内容如下：

[myexts]

basicConstraints=CA:FALSE

sComment = "OpenSSL Generated Certificate test"

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid,issuer

openssl ca -cert demoCA/cacert.pem -in req2.pem -out cert2.pem

4) 撤销一个证书

openssl ca -revoke cert2.pem

5) 生成 crl，设置原因、挂起处理方法

openssl ca -gencrl -out crl.crl

openssl ca -gencrl -crl_reason keyCompromise -crl_compromise 20010101030303Z -crl_hold holdInstructionReject -crl_CA_compromise 20020101030303Z -crldays 10 -out crl2.crl

生成一个 crl 时需要一个 crlnumber，它是一个文本文件，内容为数字，比如：03。

32.22 verify

证书验证工具、

用法：

openssl verify [-CApath directory] [-CAfile file] [-purpose purpose] [-untrusted file] [-help] [-issuer_checks] [-verbose] [-crl_check] [-engine e] [certificates]

选项

-CApath directory

信任的 CA 证书存放目录，它们的文件名为 xxxx.0，其中 xxxx 为其证书持有者的摘要值，通过 openssl x509 -hash -in cacert1.pem 可以获取。

-CAfile file

CA 证书，当其格式为 PEM 格式时，里面可以有多个 CA 证书。

-untrusted file

不信任的 CA 的证书，一个文件中可有多个不信任 CA 证书。

-purpose purpose

证书的用途，如果不设置此选项，则不会验证证书链。purpose 的值可以是：sslclient、sslserver、nssslserver、smimesign 和 smimeencrypt。

-help

打印帮助信息。

-verbose

打印详细信息。

-issuer_checks

打印被验证书与 CA 证书间的关系。

-crl_check

验证 CRL，可以将 CRL 内容写在 CAfile 指定的 PEM 文件中。

certificates

待验证的证书。

举例：

上一节，我们制作了两个证书：cert1.pem 和 cert2.pem，并撤销了 cert2.pem，生成了一个 crl 文件。在此基础上，我们将 crl 文件的内容拷贝到 demoCA/cacert.pem 的结尾，然后做如下验证命令：

openssl verify -CAfile demoCA/cacert.pem -verbose -purpose sslclient -crl_check cert1.pem cert2.pem

会有如下信息：

Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

cert1.pem: OK

cert2.pem: /C=CN/ST=JS/O=WX/OU=JN/CN=test2/emailAddress=test22@a.net

error 23 at 0 depth lookup:certificate revoked

出错信息用户请参考 verify 文档。

32.23 rsatul

rsautl 为 RSA 工具。本指令能够使用 RSA 算法签名，验证身份， 加密/解密数据。

用法：

openssl rsautl [-in file] [-out file] [-inkey file] [-pubin] [-certin] [-sign] [-verify] [-encrypt] [-decrypt] [-pkcs] [-ssl] [-raw] [-hexdump] [-engine e] [-passin arg]

选项：

-in filename

指定输入文件名，缺省为标准输入。

-out filename

指定输入文件名， 缺省为标准输出。

-inkey file

输入私钥文件名。

-pubin

表明我们输入的是一个公钥文件，默认输入为私钥文件。

-certin

表明我们输入的是一个证书文件。

-sign

给输入的数据签名。

-verify

对输入的数据进行签名。

-encrypt

用公钥对输入数据加密。

-decrypt

用私钥对输入数据解密。

-pkcs, -oaep, -ssl, -raw

指定填充方式，上述四个值分别代表：PKCS#1.5(默认值)、 PKCS#1OAEP、SSLv2 以及不填充。

-hexdump

用十六进制输出数据。

-engine e

指定硬件引擎。

-passin arg

指定私钥保护口令的来源，比如：-passin file:pwd.txt。

举例：

生成 RSA 密钥：

openssl genrsa -des3 -out prikey.pem

分离出公钥：

openssl rsa -in prikey.pem -pubout -out pubkey.pem

对文件签名：

openssl rsautl -sign -inkey prikey.pem -in a.txt -hexdump，文件 a.txt 的内容不能太长；

openssl rsautl -sign -inkey prikey.pem -in a.txt -out sig.dat

验证签名：

openssl rsautl -verify -inkey prikey.pem -in sig.dat，验证成功后打印出 a.txt 的内容；

公钥加密：

openssl rsautl -encrypt -pubin -inkey pubkey.pem -in a.txt -out b.txt

私钥解密：

openssl rsautl -decrypt -inkey prikey.pem -in b.txt

用证书中的公钥加密：

openssl rsautl -encrypt -certin -inkey cert1.pem -in a.txt

32.24 crl

crl 工具，用于处里 PME 或 DER 格式的 CRL 文件。

用法：

openssl crl [-inform PEM|DER] [-outform PEM|DER] [-text] [-in filename] [-out filename] [-noout] [-hash] [-issuer] [-lastupdate] [-nextupdate] [-CAfile file] [-CApath dir]

选项：

-inform PEM|DER

输入文件格式，默认为 PEM 格式。

-outform PEM|DER

输出文件格式，默认为 PEM 格式。

-text

打印信息。

-in filename

指定输入文件名，默认为标准输入。

-out filename

指定输出文件名，默认为标准输出。

-noout

不打印 CRL 文件内容。

-hash

打印值。

-issuer

打印颁发者 DN。

-lastupdate

上次发布时间。

-nextupdate

下次发布时间。

-CAfile file

指定 CA 文件。

-CApath dir

指定多个 CA 文件路径，每个 CA 文件的文件名为 XXXX.0，XXXX 为其持有者摘要值。

示例：

请先参考 CA 一节来生成一个 CRL 文件，再做如下操作：

openssl crl -in crl.crl -text -issuer -hash -lastupdate –nextupdate 显示 CRL 信息；

验证 CRL：

openssl crl -in crl.crl -CAfile demoCA/cacert.pem –noout

输出结果：

verify OK

下面通过指定 CA 文件路径来验证；

在 demoCA 目录下建立一个目录：CAfiles

openssl x509 -in demoCA/cacert.pem -hash 得到如下值：(比如)

86cc3989

在 CAfiles 下建立一个 86cc3989.0 文件，内容为 demoCA/cacert.pem 的内容

验证 CRL：

openssl crl -in crl.crl -CApath demoCA/CAfiles –noout

32.25 crl2pkcs7

本命令根据 CRL 或证书来生成 pkcs#7 消息。

用法：

openssl crl2pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-out filename] [-certfile filename] [-nocrl]

选项：

-inform PME|DER

CRL 输入格式，默认为 PEM 格式。

-outform PME|DER

pkcs#7 输出格式，默认为 PEM 格式。

-in filename

指定 CRL 文件，不设置此项则从标准输入中获取。

-out filename

指定输出文件，不设置此项则输入到标准输出。

-certfile filename

指定证书文件，PEM 格式的证书文件可以包含多个证书，此选项可以多次使用。

-nocrl

不处理 crl。一般情况下，输出文件中包含 crl 信息，设置此选项时，读取时忽略 CRL 信息，生成的信息不保护 CRL 信息。

示例：

openssl crl2pkcs7 -in crl.crl -out crlpkcs7.pem

openssl crl2pkcs7 -in crl.crl -certfile demoCA/ca cert.pem -out crlcertpkcs7.pem

openssl crl2pkcs7 -in crl.crl -certfile demoCA/ca cert.pem -out certpkcs7.pem –nocrl

上面生成的三个 pkcs7 文件包含的内容是不同的，crlpkcs7.pem 只有 crl 信息；crlcertpkcs7.pem 既有 crl 信息又有证书信息；certpkcs7.pem 只有证书信息。

所以，不要被 crl2pkcs7 名字所迷惑，以为它只能将 crl 转换为 pkcs7 格式的信息。

32.26 errstr

本命令用于查询错误代码。

用法：

openssl errstr [-stats] <errno>

选项：

-stats

打印哈希表状态。

errno

错误号。

举例：

用户输入：

openssl req -config no.txt

有如下错误信息：

2220:error:02001002:system library:

openssl errstr 02001002

openssl errstr -stats 02001002

32.27 ocsp

在线证书状态工具。

用法：

openssl ocsp [-out file] [-issuer file] [-cert file] [-serial num] [-signer file] [-signkey file] [-sign_other file] [-no_certs] [-req_text] [-resp_text] [-text] [-reqout file] [-respout file] [-reqin file] [-respin file] [-nonce] [-no_nonce] [-url URL] [-host host:n] [-path] [-CApath dir] [-CAfile file] [-VAfile file] [-validity_period n] [-status_age n] [-noverify] [-verify_other file] [-trust_other] [-no_intern] [-no_signature_verify] [-no_cert_verify] [-no_chain] [-no_cert_checks] [-port num] [-index file] [-CA file] [-rsigner file] [-rkey file] [-rother file] [-resp_no_certs] [-nmin n] [-ndays n] [-resp_key_id] [-nrequest n]

选项：

-out file

指定输出文件，默认为标准输出。

-issuer file

指定当前颁发者证书，此选项可以用多次，file 中的证书必须是 PEM 格式的。

-cert file

将 file 指定的证书添加到 OCSP 请求中去。

-serial num

将数字证书序列号添加到 OCSP 请求中去，num 为证书序列号，0x 开始表示是十六进制数据，否则是十进制数据，num 可以是负数，前面用-表示。

-signer file, -signkey file

OCSP 请求签名时，分别指定证书和私钥；如果只设置-signer 选项，私钥和证书都从-signer 指定的文件中读取；如果不设置这两项，OCSP 请求将不会被签名。

-sign_other filename

签名的请求中添加其他证书。

-no_certs

签名的请求中不添加任何证书。

-req_text

打印 OCSP 请求信息。

-resp_text

打印 OCSP 响应信息。

-text

打印 OCSP 请求或者响应信息。

-reqout file

指定 DER 编码的 OCSP 请求输出文件。

-respout file

指定 DER 编码的 OCSP 响应输出文件。

-reqin file

指定输入的 DER 编码的 OCSP 请求文件。

-respin file

指定输入的 DER 编码的 OCSP 响应文件。

-nonce，-no_nonce

设置或不设置 OCSP 中的 nonce 扩展。

-url URL

指定 OCSP 服务的 URL。

-host host:n

发送 OCSP 请求给服务，host 为地址或域名 n 为端口号。

-path

OCSP 请求所用的路径。

-CApath dir

可信 CA 文件目录，CA 文件名请参考其他章节说明。

-CAfile file

可信 CA 文件，file 可以包含多个 CA 证书。

-VAfile file

指定受信任的 OCSP 服务的证书，file 可以包含多个证书；等价于-verify_certs 和-trust_other 选项。

-validity_period n

设置 OCSP 响应中可接受的时间误差，n 以秒为单位。默认可接受时间误差为 5 秒，OCSP 认证中有关时间的说明请参考 OCSP 一章。

-status_age n

如果 OCSP 响应中没用提供响应的失效时间，则说明马上可以获取到新的响应信息；此时需要检查起始时间是否比当前时间晚 n 秒；默认情况不做此操作。

-noverify

不验证 OCSP 响应的签名和 nonce。

-verify_other file

设置其他用于搜索 OCSP 响应者证书的文件。

-trust_other

由-verify_other 指定的文件中包含了响应者的证书，用此选项时，不对响应者证书做额外的验证。当不能获取响应者证书的证书链或其根 CA 时，可用此选项，以保证验证能通过，即：使用了此选项后，verify_other 所指定的 OCSP 服务者证书是可以信任的，即使那些证书有问题。

-no_intern

不搜索 OCSP 响应者的证书，采用此选项时，OCSP 响应者的证书必须在-verify_certs 或-VAfile 中指定。

-no_signature_verify

不验证响应者的签名，用于测试。

-no_cert_verify

不验证响应者的证书，用于测试。

-no_chain

不验证响应者证书链。

-no_cert_checks

不验证响应者证书，不检查响应者是否有权来发布 OCSP 响应，用于测试。

-port num

OCSP 服务端口。

-index file

指定证书状态索引文件。

-CA file

指定 CA 证书。

-rsigner file

指定用于签发 OCSP 响应的证书。

-rkey file

指定用于签发 OCSP 响应的私钥文件。

-rother file

将其他证书添加到 OCSP 响应中。

-resp_no_certs

OCSP 响应中不包含证书。

-nmin n

距离下次更新时间，n 以分钟为单位。

-ndays n

距离下次更新时间，n 以天为单位。

-resp_key_id

用响应者的私钥 ID 来标记 OCSP 响应，默认为响应者证书的持有者。

-nrequest n

OCSP 服务最大响应个数，默认无限制。

举例：

1）请先用 req 和 ca 命令生成 OCSP 服务证书和私钥，下面的 OCSP 服务证书为 ocspservercert.pem，OCSP 服务签名私钥为 ocspserverkey.pem

2）生成 OCSP 请求：

openssl ocsp -issuer demoCA/cacert.pem -cert cert.pem -cert -cert2.pem -reqout ocspreq.der

3）打印 OCSP 请求信息：

openssl ocsp -reqin ocspreq.der -text

4）启动 OCSP 服务：

openssl ocsp -ndays 1 -index demoCA/index.txt -port 3904 -CA demoCA/cacert.pem -text -rkey ocspserverkey.pem -rsigner ocspservercert.pem

5）请求 OCSP 响应：

openssl ocsp -issuer demoCA/cacert.pem -url http://127.0.0.1:3904 -reqin ocspreq.der -VAfile ocspservercert.pem -respout resp.der

打印如下信息：

Response verify OK

或者：openssl ocsp -issuer demoCA/cacert.pem -url http://127.0.0.1:3904 -cert cert.pem -cert cert2.pem -VAfile ocspservercert.pem -respout resp.der

打印如下信息：

Response verify OK

cert.pem: unknown

This Update: Mar 9 16:50:12 2007 GMT

Next Update: Mar 10 16:50:12 2007 GMT

cert2.pem: revoked

This Update: Mar 9 16:50:12 2007 GMT

Next Update: Mar 10 16:50:12 2007 GMT

Revocation Time: Mar 9 13:56:51 2007 GMT

5） 根据响应的文件来验证：

openssl ocsp -respin resp.der -VAfile ocspserverc ert.pem -text

32.28 pkcs12

pkcs12 文件工具，能生成和分析 pkcs12 文件。

用法：

openssl pkcs12 [-export] [-chain] [-inkey filename] [-certfile filename] [-CApath arg] [-CAfile arg] [-name name] [-caname name] [-in filename] [-out filename] [-noout] [-nomacver] [-nocerts] [-clcerts] [-cacerts] [-nokeys] [-info] [-des] [-des3] [-aes128] [-aes192] [-aes256] [-idea] [-nodes] [-noiter] [-maciter] [-twopass] [-descert] [-certpbe alg] [-keypbe alg] [-keyex] [-keysig] [-password arg] [-passin arg] [-passout arg] [-rand file(s)] [-engine e]

选项：

-export

输出 pkcs12 文件。

-chain

添加证书链。

-inkey filename

指定私钥文件，如果不用此选项，私钥必须在-in filename 中指定。

-certfile filename

添加 filename 中所有的文件。

-CApath arg

指定 CA 文件目录。

-CApath arg

指定 CA 文件。

-name name

指定证书和私钥的友好名。

-caname name

指定 CA 友好名，可以多次使用此选项。

-in filename

指定私钥和证书读取的文件，必须为 PEM 格式。

-out filename

指定输出的 pkcs12 文件，默认为标准输出。

-noout

不输出信息。

-nomacver

读取文件时不验证 MAC。

-nocerts

不输出证书。

-clcerts

只输出客户证书，不包含 CA 证书。

-cacerts

只输出 CA 证书，不包含 CA 证书。

-nokeys

不输出私钥。

-info

输出 pkcs12 结构信息。

-des3，-aes128 ，-aes192，[-aes256，[-idea

私钥加密算法；。

-nodes

不对私钥加密。

-noiter

不多次加密。

-maciter

加强完整性保护，多次计算 MAC。

-twopass

需要用户分别指定 MAC 口令和加密口令。

-descert

用 3DES 加密 pkcs12 文件，默认为 RC2-40。

-certpbe alg

指定证书加密算法，默认为 RC2-40。

-keypbe alg

指定私钥加密算法，默认为 3DES。

-keyex

设置私钥只能用于密钥交换。

-keysig

设置私钥只能用于签名。

-password arg

指定导入导出口令来源。

-passin arg

输入文件保护口令来源。

-passout arg

指定所有输出私钥保护口令来源。

-rand file(s)

指定随机数种子文件，多个文件间用分隔符分开，windows 用“;”，OpenVMS 用“,“，其他系统用“:”。

-engine e

指定硬件引擎。

举例：

1）生成 pkcs12 文件，但不包含 CA 证书：

openssl pkcs12 -export -inkey ocspserverkey.pem -in ocspservercert.pem -out ocspserverpkcs12.pfx

2） 生成 pcs12 文件，包含 CA 证书：

openssl pkcs12 -export -inkey ocspserverkey.pem -in ocspservercert.pem -CAfile demoCA/cacert.pem -chain -out ocsp1.pfx

3） 将 pcks12 中的信息分离出来，写入文件：

openssl pkcs12 –in ocsp1.pfx -out certandkey.pem

4） 显示 pkcs12 信息：

openssl pkcs12 –in ocsp1.pfx -info

32.29 pkcs8

pkcs8 格式的私钥转换工具。

用法：

openssl pkcs8 [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out filename] [-passout arg] [-topk8] [-noiter] [-nocrypt] [-nooct] [-embed] [-nsdb] [-v2 alg] [-v1 alg] [-engine id]

选项：

-inform PEM|DER

输入文件格式。

-outform PEM|DER

输出文件格式。

-in filename

输入文件。

-passin arg

输入文件口令保护来源。

-out filename

指定输出文件。

-passout arg

输出文件口令保护来源。

-topk8

输出 pkcs8 文件。

-noiter

MAC 保护计算次数为 1。

-nocrypt

加密输入文件，输出的文件不被加密。

-nooct

不采用八位组表示私钥。

-embed

采用嵌入式 DSA 参数格式。

-nsdb

采用 Netscape DB 的 DSA 格式。

-v2 alg

采用 PKCS#5 v2.0，并指定加密算法，可以是 des、des3 和 rc2，推荐 des3。

-v1 alg

采用 PKCS#5 v1.5 或 pkcs12，并指定加密算法，可采用算法包括：

PBE-MD2-DES 、 PBE-MD5-DES 、 PBE-SHA1-RC2-64 、 PBE-MD2-RC2-64 、 PBE-MD5-RC2-64、PBE-SHA1-DES、PBE-SHA1-RC4-128、PBE-SHA1-RC4-40、 PBE-SHA1-3DES、PBE-SHA1-2DES、PBE-SHA1-RC2-128 和 PBE-SHA1-RC2-40。

-engine i

指定硬件引擎。

示例：

1） 将私钥文件转换为 pkcs8 文件：

openssl pkcs8 -in ocspserverkey.pem -topk8 -out ocspkcs8key.pem

2） pkcs8 中的私钥以明文存放：

openssl pkcs8 -in ocspserverkey.pem -topk8 -nocrypt -out ocspkcs8key.pem

32.30 s_time

s_time 是 openss 提供的 SSL/TLS 性能测试工具，用于测试 SSL/TSL 服务。

用法：

openssl s_time [-connect host:port] [-www page] [-cert filename] [-key filename] [-CApath directory] [-CAfile filename] [-reuse] [-new] [-verify depth] [-nbio] [-time seconds] [-ssl2] [-ssl3] [-bugs] [-cipher cipherlist]

用法：

-connect host:port

指定服务，默认为本机的 4433 端口。

-www page

指定获取的 web 网页。

-cert filename

指定证书。

-key filename

指定私钥。

-CApath directory

指定 CA 文件目录。

-CAfile filename

指定 CA 文件。

-reuse

session 重用。

-new

新建链接。

-verify depth

设置验证深度。

-nbio

不采用 BIO。

-time seconds

指定搜集数据的秒数，默认 30 秒。

-ssl2，-ssl3

采用的 SSL 协议。

-bugs

开启 SSL bug 兼容。

-cipher cipherlist

指定加密套件。

示例：

1） 启动 s_server 服务：

openssl s_server -cert sslservercert.pem -key sslserverkey.pem -ssl3

2) 启动 s_time

openssl s_time -cert sslclientcert.pem -key sslclientkey.pem -CAfile demoCA/cacert.pem -ssl3

32.31 dhparam 和 dh

Dhparam 为 dh 参数操作和生成工具。dh 命令与 dhparam 用法大致一致，下面只给出了 dhparam 的说明。

用法：

openssl dhparam [-inform DER|PEM] [-outform DER|PEM] [-in filename] [-out filename] [-dsaparam] [-noout] [-check] [-text] [-C] [-2] [-5] [-rand file(s)] [-engine id] [numbits]

选项：

-inform DER|PEM

输入文件格式，DER 或者 PEM 格式。

-outform DER|PEM

输出格式。

-in filename

读取 DH 参数的文件，默认为标准输入。

-out filename

dh 参数输出文件，默认为标准输出。

-dsaparam

生成 DSA 参数，并转换为 DH 格式。

-noout

不输出信息。

-text

打印信息。

-check

检查 dh 参数。

-C

以 C 语言风格打印信息。

-2，-5

指定 2 或 5 为发生器，默认为 2，如果指定这些项，输入 DH 参数文件将被忽略，自动生成 DH 参数。

-rand files

指定随机数种子文件。

-engine id

指定硬件引擎。

numbit

指定素数 bit 数，默认为 512。

示例：

1) openssl dhparam –out dhparam.pem -text 512

生成内容如下：

Diffie-Hellman-Parameters: (512 bit)

prime:

00:8f:18:1b:4f:7a:74:e1:89:42:e6:99:0f:15:4e:
72:ad:ca:7b:fb:68:ef:85:7b:16:a8:5b:85:01:82:
dd:db:57:1f:c5:86:89:fa:16:10:6e:d0:05:2b:15:
e2:87:98:0e:53:f2:c8:18:f9:5b:7e:4d:ce:9b:6d:
3f:23:11:52:63

generator: 2 (0x2)

-----BEGIN DH PARAMETERS-----

MEYCQQCPGBtPenThiULmmQ8VTnKtynv7aO+FexaoW4UBgt3bVx/Fhon6FhBu0AUr
FeKHmA5T8sgY+Vt+Tc6bbT8jEVJjAgEC

-----END DH PARAMETERS-----

2) 检查生成的 DH 参数

openssl dhparam -in dhparam.pem -text -check

32.32 ecparam

椭圆曲线密钥参数生成及操作。

用法：

openssl ecparam [-inform DER|PEM] [-outform DER|PEM] [-in filename] [-out filename] [-noout] [-text] [-C] [-check] [-name arg] [-list_curve] [-conv_form arg] [-param_enc arg] [-no_seed] [-rand file(s)] [-genkey] [-engine id]

用法：

-inform DER|PEM

输入文件格式。

-outform DER|PEM

输出文件格式。

-in filename

输入文件。

-out filename

输出文件。

-noout

不打印信息。

-text

打印信息。

-C

以 C 语言风格打印信息。

-check

检查参数。

-name arg

采用短名字。

-list_curves

打印所有可用的短名字。

-conv_form arg

指定信息存放方式，可以是 compressed、uncompressed 或者 hybrid，默认为 compressed。

-param_enc arg

指定参数编码方法，可以是 named_curve 和 explicit，默认为 named_curve。

-no_seed

如果-param_enc 指定编码方式为 explicit，不采用随机数种子。

-rand file(s)

指定随机数种子。

-genkey

生成密钥。

-engine id

指定硬件引擎。

示例：

openssl ecparam -list_curves
openssl ecparam -name secp112r1 -genkey –text
openssl ecparam -genkey -name secp160r1 -out ec160.pem
openssl req -newkey ec:ec160.pem

32.33 ec

椭圆曲线密钥处理工具。
用法：

openssl ec [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out filename] [-passout arg] [-des] [-des3] [-idea] [-text] [-noout] [-param_out] [-pubin] [-pubout] [-conv_form arg] [-param_enc arg] [-engine id]

选项：

-inform PEM|DER
输入文件格式。
-outform PEM|DER
输出文件格式。
-in filename
输入文件名。
-passin arg
私钥保护口令来源。
-out filename
输出文件名。
-passout arg
输出文件保护口令来源。
-des，-des3，-idea
私钥保护算法。
-noout
不输出信息。
-param_out
输出参数。
-pubin
输入的是公钥。
-pubout
输出公钥。
-conv_form arg
指定信息存放方式，可以是 compressed、uncompressed 或者 hybrid，默认为 compressed。
-param_enc arg
指定参数编码方法，可以是 named_curve 和 explicit，默认为 named_curve。
-engine id
指定硬件引擎。

示例：

1) 生成 ec 私钥

openssl ecparam -genkey -name secp112r1 -out eckey.pem -text

2) 转换为 DER 编码

openssl ec -outform der -in eckey.pem -out eckey.der

3) 给私钥进行口令保护

openssl ec -in eckey.pem -des -out enceckey.pem

4) 将公钥写入文件

openssl ec -in eckey.pem -pubout -out ecpubkey.pem

5） 显示密钥信息

openssl ec -in eckey.pem –text

openssl ec -in ecpubkey.pem -pubin –text

6） 转换为 pkcs8 格式

openssl pkcs8 -topk8 -in eckey.pem -out eckeypk8.pem

32.34 dsa

dsa 命令用于处理 DSA 密钥、格式转换和打印信息。

用法：

openssl dsa [-inform PEM|DER] [-outform PEM|DER] [-in filename]
[-passin arg] [-out filename] [-passout arg] [-des] [-des3] [-idea]
[-text] [-noout] [-modulus] [-engine id]

选项：

-inform

输入 dsa 密钥格式，PEM 或 DER。

-outform

输出文件格式，PEM 或 DER。

-in filename

输入的 DSA 密钥文件名。

-passin arg

指定私钥包含口令存放方式。比如用户将私钥的保护口令写入一个文件，采用此选项指定此文件，可以免去用户输入口令的操作。比如用户将口令写入文件“pwd.txt”，输入的参数为：-passin file:pwd.txt。

-out filename

指定输出文件名。

-passout arg

输出文件口令保护存放方式。

-des -des3 -idea

指定私钥保护加密算法。

-text

打印所有信息。

-noout

不打印信息。

-modulus

打印公钥信息。

-engine id

指定引擎。

示例：

1) 生成 dsa 参数文件

openssl dsaparam -out dsaparam.pem 1024

2) 根据 dsa 参数文件生成 dsa 密钥

openssl gendsa -out dsakey.pem dsaparam.pem

3) 将 PME 密钥转换为 DER 密钥

openssl dsa -in dsakey.pem -outform DER -out dsakeyder.pem

4) 打印公钥信息

openssl dsa -in dsakey.pem –modulus

5) 打印所有信息

openssl dsa -in dsakey.pem –text

6) 将 dsa 密钥加密存放

openssl dsa -in dsakey.pem -des -out enckey.pem

32.35 nseq

本命令用于多个证书与 netscape 证书序列间相互转化。

用法：openssl nseq [-in filename] [-out filename] [-toseq]

选项：

-in filename

输入文件名。

-out filename

输出文件名。

-toseq

含此项时将多个证书转化为 netscape 证书序列，否则将 netscape 证书序列转化为多个证书。

示例：

1) 将多个证书写成一个文件

cat newcert.pem > 1.pem

cat cacert.pem >> 1.pem

2) 将多个证书转化为 netscape 证书序列

openssl nseq -in 1.pem -toseq -out 2.pem

3) 将 netscape 证书序列转化为多个证书

openssl nseq -in 2.pem -out 3.pem

32.36 prime

检查一个数是否为素数。示例如下：

openssl prime 79

openssl prime -hex 4F

32.37 smime

S/MIME 工具，用于处理 S/MIME 邮件，它能加密、解密、签名和验证 S/MIME 消息。

用法：

openssl smime [-encrypt] [-decrypt] [-sign] [-verify] [-pk7out] [-des]
[-des3] [-rc2-40] [-rc2-64] [-rc2-128] [-in file] [-certfile file]
[-signer file] [-recip file] [-inform SMIME|PEM|DER] [-passin arg]
[-inkey file] [-out file] [-outform SMIME|PEM|DER] [-content file] [-to
addr] [-from ad] [-subject s] [-text] [-rand file(s)] [cert.pem]...

主要选项：

-encrypt
加密数据。

-decrypt
解密数据。

-sign
签名数据。

-verify
验证数据。

-in
输入文件名。

-out
输出文件名。

-pk7out
输出 pkcs7 格式的文件。

-des -des3 -rc2-40 –rc2-60 –rc2-128
对称算法。

-signer file
指定签名者证书。

-recip file
指定接收者证书。

-inform
输入文件格式。

-passin arg
私钥保护口令来源。

-inkey file
私钥文件。

-outform
输出文件格式。

示例：

1) 用对方的证书来加密消息
openssl smime -encrypt -in mail.pem -out enced.pem newcert.pem
openssl smime -encrypt -in mail.pem -out enced.pem -des newcert.pem

2） 用私钥解密消息
openssl smime -decrypt -in enced.pem -out mymail.pem -inkey newkey.pem

openssl smime -decrypt -in enced.pem -out mymail.pem -inkey newkey.pem -des

3）用自己的私钥签名数据

openssl smime -sign -in mail.pem -out signedmail.pem -inkey newkey.pem -signer newcert.pem

4) 验证签名

openssl smime -verify -in signedmail.pem -CAfile newcert.pem -signer newcert.pem

此处 newcert 是一个自签名证书，如果不是自签名证书用如下命令：

openssl smime -verify -in signedmail.pem -CAfile demoCA/cacert.pem -signer newcert2.pem

5) 将数据转化为 pkcs7 格式

openssl smime -pk7out -in signedmail.pem -out p7.pem

后记

终于写完个大概，希望大家批评指正并完善。可以给我发邮件：forxy@126.com。最好加个标题：openssl 编程。

书中有部分文字是来自网上的，比如密码学的介绍以及部分 openssl 命令。其他都是本人手打。希望这本书对您有所帮助。

最后我送大家几句话：宁静致远，兴趣是最好的老师。