Openssl 目录名以及功能描述

目录名	功能描述
Crypto	存放 OpenSSL 所有加密算法源码文件和相关标注如 X.509 源码文件，是 OpenSSL 中最重要的目录，包含了 OpenSSL 密码算法库的所有内容。
SSL	存放 OpenSSL 中 SSL 协议各个版本和 TLS 1.0 协议源码文件，包含了 OpenSSL 协议库的所有内容。
Apps	存放 OpenSSL 中所有应用程序源码文件，如 CA、X509 等应用程序的源文件就存放在这里。
Doc	存放了 OpenSSL 中所有的使用说明文档，包含三个部分：应用程序说明文档、加密算法库 API 说明文档以及 SSL 协议 API 说明文档。
Demos	存放了一些基于 OpenSSL 的应用程序例子，这些例子一般都很简单，演示怎么使用 OpenSSL 其中的一个功能。
Include	存放了使用 OpenSSL 的库时需要的头文件。
Test	存放了 OpenSSL 自身功能测试程序的源码文件

OpenSSL 算法目录

openSSL 的算法目录 Crypto 目录包含了 OpenSSL 密码算法库的所有源代码文件，是 OpenSSL 中最重要的目录之一。

OpenSSL 的密码算法库包含了 OpenSSL 中所有密码算法、密钥管理和证书管理相关标准的实现，在 Windows 下编译后的库文件名为 libeay32.lib，在 Linux 下编译后生产的库文件名为 libcrypto.a。Crypto 目录下包含了众多的子目录，这些目录大多数以相关的算法或标准名称的简写命名。

当然,并非所有这些目录存放的源文件都是密码算法和标准，有些是 OpenSSL 本身的一些相关功能文件，如 BIO、DSO 和 EVP 等。

Crypto 子目录列表

目录名	功能描述
Aes	对称算法，美国新的对称加密算法标准 AES 算法源码。
Bf	对称算法，Blowfish 对称加密算法源码。
Cast	对称算法，CAST 对称加密算法源码。
Des	对称算法，包括了 DES 和 3DES 对称加密算法源码。
Idea	对称算法，IDEA 对称加密算法源码。
Rc2	对称算法，RC2 对称加密算法源码。
Rc4	对称算法，RC4 对称加密算法源码
Rc5	对称算法，RC5 对称加密算法源码。
Dh	非对称算法，DH 非对称密钥交换算法源码。
Dsa	非对称算法，DSA 非对称算法源码，用于数字签名。
Ec	非对称算法，EC 椭圆曲线算法源码。
Rsa	非对称算法，RSA 非对称加密算法源码，既可以用于密钥交换，也可以用于数字签名。
Md2	信息摘要算法，MD2 信息摘要算法源码。
Md5	信息摘要算法，MD5 信息摘要算法源码。
Mdc2	信息摘要算法，MDC2 信息摘要算法源码。
Sha	信息摘要算法，SHA 信息摘要算法源码，包括了 SHA1 算法。
Ripemd	信息摘要算法，RIPEMD-160 信息摘要算法源码。
Comp	数据压缩算法数据压缩算法的函数接口，目前没有压缩算法，只是定义了一些空的接口函数
Asn1	PKI 相关标准 ASN.1 标准实现源码，只实现了 PKI 相关的部分，不是完全实现。包括 DER 编解码等功能。
Ocsp	PKI 相关标准 OCSP（在线证书服务协议）实现源码。
Pem	PKI 相关标准 PEM 标准实现源码，包括了 PEM 的编解码功能。
Pkcs7	PKI 相关标准 PKCS#7 标准实现源码。PKCS#7 是实现加密信息封装的标准，包括了证书封装的标准和加密数据的封装标准。
Pkcs12	PKI 相关标准 PKCS#12 标准实现源码。包括了 PKCS#12 文件的编解码功能。PKCS#12 是一种常用的证书和密钥封装格式。
X509	PKI 相关标准 X.509 标准的实现源码。包括了 X.509 的编解码功能，证书管理功能等。X509v3 PKI 相关标准 X.509 第三版扩展功能的实现源码

Krb5	其它标准支持支持 Kerberos 协议的一些接口函数和结构定义
Hmac	其它标准支持 HMAC 标准的支持结构和函数源源代码。
Lhash	其它标准支持动态 HASH 表结构和函数源代码
Bio	自定义 OpenSSL 自身定义的一种抽象 IO 接口，封装了各种平台的几乎所有 IO 接口，如文件、内存、缓存、标准输入输出以及 Socket 等等。
Bn	自定义 OpenSSL 实现大数管理的结构及其函数。
Buffer	自定义 OpenSSL 自定义的缓冲区结构体。
Conf	自定义 OpenSSL 自定义的管理配置结构和函数。
Dso	自定义 OpenSSL 自定义的加载动态库的管理函数接口。如使用 Engine 机制就用到了这些函数提供的功能。
Engine	自定义 OpenSSL 自定义的 Engine 机制源代码。Engine 机制运行 OpenSSL 使用第三方提供的软件密码算法库或者硬件加密设备进行数据加密等运算。相当于 Windows 平台的 CSP 机制。
Err	自定义 OpenSSL 自定义的错误信息处理机制。
Evp	自定义 OpenSSL 定义的一组高层算法封装函数，包括了对称加密算法封装、非对称加密算法封装、签名验证算法封装以及信息摘要算法封装，类似

PKCS#11 提供的接口标准。

Objects	自定义 OpenSSL 管理各种数据对象的定义和函数。事实上，Objects 的 OID 是根据 ASN.1 的标准进行命名的，不完全是 OpenSSL 自定义的结构。
Rand	自定义 OpenSSL 的安全随机数产生函数和管理函数。
Stack	自定义定义了 OpenSSL 中 STACK 结构和相关管理函数。
Threads	自定义 OpenSSL 处理线程的一些机制。
Txt_db	自定义 OpenSSL 提供的文本证书库的管理机制。
Ui	自定义 OpenSSL 定义的一下用户接口交换函数。
Perlasm	自定义编译的时候需要用到的一些 Perl 辅助配置文件。

☆对称加密算法

OpenSSL 一共提供了 8 种对称加密算法，其中 7 种是分组加密算法，仅有的一种流加密算法是 RC4。这 7 种分组加密算法分别是 AES、DES、Blowfish、CAST、IDEA、RC2、RC5，都支持电子密码本模式（ECB）、加密分组链接模式（CBC）、加密反馈模式（CFB）和输出反馈模式（OFB）四种常用的分组密码加密模式。其中，AES 使用的加密反馈模式（CFB）和输出反馈模式（OFB）分组长度是 128 位，其它算法使用的则是 64 位。事实上，DES 算法里面不仅仅是常用的 DES 算法，还支持三个密钥和两个密钥 3DES 算法。

虽然每种加密算法都定义了自己的接口函数，但是 OpenSSL 还使用 EVP 封装了所有的对称加密算法，使得各种对成加密算法能够使用统一的 API 接口 EVP_Encrypt 和 EVP_Decrypt 进行数据的加密和解密，大大提供了代码的可重用性能。

☆非对称加密算法

OpenSSL 一共实现了 4 种非对称加密算法，包括 DH 算法、RSA 算法、DSA 算法和椭圆曲线算法（EC）。DH 算法一般用户密钥交换。RSA 算法既可以用于密钥交换，也可以用于数字签名，当然，如果你能够忍受其缓慢的速度，那么也可以用于数据加密。DSA 算法则一般只用于数字签名。

跟对称加密算法相似，OpenSSL 也使用 EVP 技术对不同功能的非对称加密算法进行封装，提供了统一的 API 接口。如果使用非对称加密算法进行密钥交换或者密钥加密，则使用 EVP_Seal 和 EVP_Open 进行加密和解密；如果使用非对称加密算法进行数字签名，则使用 EVP_Sign 和 EVP_Verify 进行签名和验证。

☆信息摘要算法

OpenSSL 实现了 5 种信息摘要算法，分别是 MD2、MD5、MDC2、SHA（SHA1）和 RIPEMD。SHA 算法事实上包括了 SHA 和 SHA1 两种信息摘要算法，此外，OpenSSL 还实现了 DSS 标准中规定的两种信息摘要算法 DSS 和 DSS1。

OpenSSL 采用 EVP_Digest 接口作为信息摘要算法统一的 EVP 接口，对所有信息摘要算法进行了封装，提供了代码的重用性。

☆密钥和证书管理

密钥和证书管理是 PKI 的一个重要组成部分，OpenSSL 为之提供了丰富的功能，支持多种标准。

首先，OpenSSL 实现了 ASN.1 的证书和密钥相关标准，提供了对证书、公钥、私钥、证书请求以及 CRL 等数据对象的 DER、PEM 和 BASE64 的编解码功能。OpenSSL 提供了产生各种公开密钥对和对称密钥的方法、函数和应用程序，同时提供了对公钥和私钥的 DER 编解码功能。并实现了私钥的 PKCS#12 和 PKCS#8 的编解码功能。OpenSSL 在标准中提供了对私钥的加密保护功能，使得密钥可以安全地进行存储和分发。

在此基础上，OpenSSL 实现了对证书的 X.509 标准编解码、PKCS#12 格式的编解码以及 PKCS#7 的编解码功能。并提供了一种文本数据库，支持证书的管理功能，包括证书密钥产生、请求产生、证书签发、吊销和验证等功能。

事实上，OpenSSL 提供的 CA 应用程序就是一个小型的证书管理中心（CA），实现了证书签发的整个流程和证书管理的大部分机制。

Engine 机制

Engine 机制的出现是在 OpenSSL 的 0.9.6 版的事情，开始的时候是将普通版本跟支持 Engine 的版本分开的，到了 OpenSSL 的 0.9.7 版，Engine 机制集成到了 OpenSSL 的内核中，成为了 OpenSSL 不可缺少的一部分。

Engine 机制目的是为了使 OpenSSL 能够透明地使用第三方提供的软件加密库或者硬件加密设备进行加密。OpenSSL 的 Engine 机制成功地达到了这个目的，这使得 OpenSSL 已经不仅仅使一个加密库，而是提供了一个通用地加密接口，能够与绝大部分加密库或者加密设备协调工作。当然，要使特定加密库或加密设备更 OpenSSL 协调工作，需要写少量的接口代码，但是这样的工作量并不大，虽然还是需要一点密码学的知识。Engine 机制的功能跟 Windows 提供的 CSP 功能目标是基本相同的。

BIO 机制

BIO 机制是 OpenSSL 提供的一种高层 IO 接口，该接口封装了几乎所有类型的 IO 接口，如内存访问、文件访问以及 Socket 等。这使得代码的重用性大幅度提高，OpenSSL 提供 API 的复杂性也降低了很多。

本文来自 CSDN 博客，转载请标明出处：

http://blog.csdn.net/kafeiwu2003/archive/2009/02/24/3931612.aspx

openssl 之 BIO 系列之 01~25,EVP 系列之 01~20

作者：gdwzh 时间:2002-01-02 11:56 出处:互联网 责编:MyFAQ

作者：DragonKing Mail:wzhah@263.net 发布于：http://gdwzh.126.com openssl 专业论坛

摘要：openssl 之 BIO 系列之 1---抽象的 IO 接口

BIO-抽象的 IO 接口

其实包含了很多种接口，用通用的函数接口，主要控制在 BIO_METHOD 中的不通实现函数控制，我初步估计了一下，大概有 14 种，包括 6 种 filter 型和 8 种 source/sink 型。

BIO 是在底层覆盖了许多类型 I/O 接口细节的一种应用接口，如果你在程序中使用 BIO，那么就可以和 SSL 连接、非加密的网络连接以及文件 IO 进行透明的连接。

有两种不通的 BIO 接口，一种是 **source/sink 型**，一种是 **fileter 型**的。

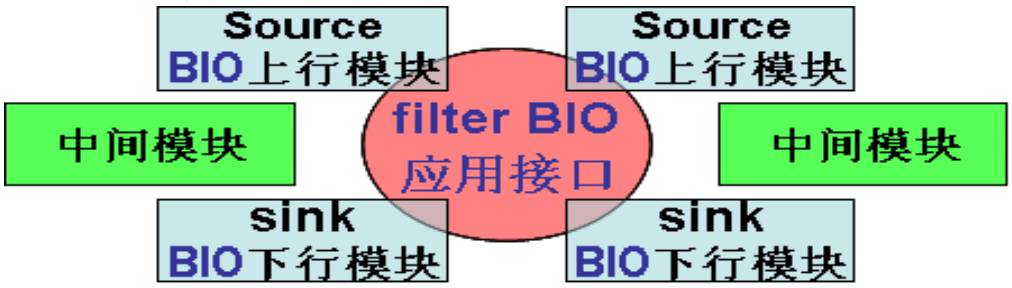
顾名思义，**source/sink 类型**的 BIO 是**数据源**或**数据目标**（我不知道 sink 该怎么翻译，据水木 liaojzh 说，一般是 destination（目标、宿）的同义词，大家自己理解吧，呵呵），例如，sokect BIO 和文件 BIO。

source/sink 直译的意思是"源"和"漏"。我更喜欢称呼它们"**上行模块**"和"**下行模块**"。其中：

本级"上行模块"负责结果数据的输出以及与下一级的"下行模块"接驳。

本级"下行模块"负责原始数据的输入以及与上一级的"上行模块"接驳。

而 **filter BIO** 就是把数据从一个 BIO 转换到另外一个 BIO 或**应用接口**，在转换过程中，这些数据可以不修改（如信息摘要 BIO)，也可以进行转换。例如在加密 BIO 中，如果写操作，数据就会被加密，如果是读操作，数据就会被解密。



BIO 可以连接在一起成为一个 BIO 链（单个的 BIO 就是一个环节的 BIO 链的特例），如下是 BIO 的结构定义，可以看到它有上下环节的：

```
struct 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 */BIO 下联
 struct bio_st *prev_bio; /* used by filter BIOs */BIO 上联
 int references;
 unsigned long num_read;
 unsigned long num_write;
 CRYPTO_EX_DATA ex_data;
 };
```

一个 BIO 链通常**包括一个 source BIO** 和**一个或多个 filter BIO**，**数据从第一个 BIO 读出或写入,然后经过一系列 BIO 变化到输出**（通常是一个 source/sink BIO）。

注：这是根据 openssl 的 BIO.pod 翻译和根据我自己的理解添加的，以后我会慢慢将 BIO 的细节说出来，希望大家一起努力。

openssIBIO 系列之 2---BIO 结构和 BIO 相关文件介绍

BIO 的结构定义和相关项解析如下：

（包含在 bio.h 文件中，其主文件为 bio_lib.c）

```
typedef struct bio_st BIO;
struct bio_st
 {
 BIO_METHOD *method;//BIO 方法结构，是决定 BIO 类型和行为的重要参数，各种 BIO 的不同之处主要也正在于此项。
 /* bio, mode, argp, argi, argl, ret */
 long (*callback)(struct bio_st *,int,const char *,int, long,long);//BIO 回调函数
 char *cb_arg; /* first argument for the callback * //回调函数的第一个参量
 int init;//初始化标志，初始化了为 1，否则为 0
 int shutdown;//BIO 开关标志，如果为 1，则处于关闭状态，如果为 0，则处于打开的状态。
 int flags; /* extra storage */
 int retry_reason;
 int num;
 void *ptr;
 struct bio_st *next_bio; /* used by filter BIOs */BIO 下联
 struct bio_st *prev_bio; /* used by filter BIOs */BIO 上联
 int references;
 unsigned long num_read; //读出的数据长度
 unsigned long num_write;//写入的数据长度
 CRYPTO_EX_DATA ex_data;
 };
```

在 BIO 的所用成员中，method 可以说是最关键的一个成员，它决定了 BIO 的类型，可以看到，在声明一个新的 BIO 结构时，总是使用下面的声明：

```
BIO* BIO_new(BIO_METHOD *type);
```

在源代码可以看出，BIO_new 函数除了给一些初始变量赋值外，主要就是把 type 中的各个变量赋值给 BIO 结构中的 method 成员。

一般来说，上述 type 参数是以一个类型生成函数的形式提供的，如生成一个 mem 型的 BIO 结构，就使用下面的语句：

```
BIO *mem = BIO_new(BIO_s_mem());
```

这样的函数有以下一些：

【source/sink 型】

- BIO_s_accept():是一个封装了类似 TCP/IP socket Accept 规则的接口，并且使 TCP/IP 操作对于 BIO 接口是透明的。
- BIO_s_bio():封装了一个 BIO 对，数据从其中一个 BIO 写入，从另外一个 BIO 读出
- BIO_s_connect():是一个封装了类似 TCP/IP socket Connect 规则的接口，并且使 TCP/IP 操作对于 BIO 接口是透明的
- BIO_s_fd():是一个封装了文件描述符的 BIO 接口，提供类似文件读写操作的功能
- BIO_s_file():封装了标准的文件接口的 BIO，包括标志的输入输出设备如 stdin 等
- BIO_s_mem():封装了内存操作的 BIO 接口，包括了对内存的读写操作
- BIO_s_null():返回空的 sink 型 BIO 接口，写入这种接口的所有数据读被丢弃，读的时候总是返回 EOF
- BIO_s_socket():封装了 socket 接口的 BIO 类型

【filter 型】

- BIO_f_base64()：封装了 base64 编码方法的 BIO,写的时候进行编码，读的时候解码
- BIO_f_buffer()：封装了缓冲区操作的 BIO，写入该接口的数据一般是准备传入下一个 BIO 接口的，从该接口读出的数据一般也是从另一个 BIO 传过来的。
- BIO_f_cipher()：封装了加解密方法的 BIO，写的时候加密，读的时候解密
- BIO_f_md()：封装了信息摘要方法的 BIO，通过该接口读写的数据都是已经经过摘要的。
- BIO_f_null()：一个不作任何事情的 BIO，对它的操作都简单传到下一个 BIO 去了，相当于不存在。
- BIO_f_ssl()：封装了 openssl 的 SSL 协议的 BIO 类型，也就是为 SSL 协议增加了一些 BIO 操作方法。

上述各种类型的函数正是构成 BIO 强大功能的基本单元，所以，要了解 BIO 的各种结构和功能，也就应该了解这些函数类型相关的操作函数。

所有这些源文件，都基本上包含于/crypto/bio/目录下的同名.c 文件（大部分是同名的）中。

在 BIO_METHOD 里面，定义了一组行为函数，上述不通类型的 BIO_METHOD 行为函数的定义是不同的，其结构如下（以非 16 位系统为例）：

```
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;
```

在 BIO 的成员中，callback 也是比较重要的，它能够用于程序调试用或者自定义改变 BIO 的行为。详细会在以后相关的部分介绍。

BIO 的很多操作，都是 BIO_ctrl 系列函数根据不通参数组成的宏定义来完成的。所以要了解 BIO 的行为，了解 BIO_ctrl 系列函数以及其各个参数的意义也是很重要的。

【BIO 目录文件的简要说明】

bio.h：　　主定义的头文件，包括了很多通用的宏的定义。
bio_lib.c　　主要的 BIO 操作定义文件，是比较上层的函数了。
bss_*系列：　　是 soruce/sink 型 BIO 具体的操作实现文件
bf_*系列：　　是 filter 型 BIO 具体的操作实现文件
bio_err.c：　　是错误信息处理文件
bio_cb.c：　　是 callback 函数的相关文件
b_print.c：　　是信息输出的处理函数文件
b_socket.c：　　是 Socket 连接的一些相关信息处理文件
b_dump.c：　　是对内存内容的存储操作处理

由于时间和能力有限，这个概述就写到这儿了，以后的文章主要根据 openssl 的帮助文档，并结合源代码做一些分析。

openssl 之 X509 系列之 3---证书请求的 IO 函数

声明：本文档根据 openssl 源代码、SSLeay 文档以及其它相关材料写成，转载需经本站授权，商业用途不得转载.本人能力有限，如有问题欢迎交流与指正。

【输入输出函数】

这些函数有两类：一类是将 X509_REQ 信息在文件或 BIO 抽象层上输入输出，另一类是在控制台上将 X509_REQ 信息进行显示。它们的函数定义如下：

```
X509_REQ *d2i_X509_REQ_fp(FILE *fp,X509_REQ **req);
int i2d_X509_REQ_fp(FILE *fp,X509_REQ *req);
X509_REQ *d2i_X509_REQ_bio(BIO *bp,X509_REQ **req);
int i2d_X509_REQ_bio(BIO *bp,X509_REQ *req);
int X509_REQ_print_ex(BIO *bp, X509_REQ *x, unsigned long nmflag, unsigned long cflag);
int X509_REQ_print(BIO *bp,X509_REQ *req);
int X509_REQ_print_fp(FILE *fp, X509_REQ *x)
```

【d2i_X509_REQ_fp】

将证书请求从文件中读入并转化成 X509_REQ 内部结构。

【i2d_X509_REQ_fp】

将 X509_REQ 对象进行 DER 编码输出,并写入 fp 指定的文件中。

【d2i_X509_REQ_bio】

功能与 d2i_X509_REQ_fp 相同，只是读的时候从 BIO 抽象层上读，你可以将它与文件相关联就可以了。

【i2d_X509_REQ_bio】

功能与 i2d_X509_REQ_fp 相同，只是写的时候从 BIO 抽象层上写，你可以将它与文件或者内存 BIO 相关联就可以输出了。

【X509_REQ_print】

将 X509_REQ 在 BIO 上输出，但输入是可以读的，比如 Subject=XXX 等。其实底层就是调用 X509_REQ_print_ex 来实现的。

【X509_REQ_print_ex】

这个函数与 X509_REQ_print 的区别是可以用标志去控制输出，nmflags 用于控制显示方式，cflag 用于控制哪些不显示，可以按自己的需要进行定制。它们的定义在 x509.h 里。

具体如下：

```
#define X509_FLAG_COMPAT 0
#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)
```

【X509_REQ_print_fp】

其实这个函数就是将可读的结果保存在文件里，内存就是生成一个 BIO 对象 BIO_new(BIO_s_file()，然后再将文件句柄传给他 BIO_set_fp(b,fp,BIO_NOCLOSE)，再调用 X509_REQ_print 函数进行输出。这几个 print 函数,具体实现在 crypto/asn1/t_req.c 中。

opensslBIO 系列之 4---BIO 控制函数介绍

BIO 控制函数介绍---根据 openssl doc/crypto/bio/bio_ctrl.pod 翻译和自己的理解写成

BIO 控制函数有许多，并且不同的 BIO 类型还有不同的控制函数，这里只简单介绍一些通用的 BIO 控制函数，至于某种类型 BIO 的特定控制函数，则参考后续的文件。

BIO 的通用控制函数有以下几种，其声明如下（openssl/bio.h）：

```
long BIO_ctrl(BIO *bp,int cmd,long larg,void *parg);
long BIO_callback_ctrl(BIO *b, int cmd, void (*fp)(struct bio_st *, int, const char *, int, long, long));
char * BIO_ptr_ctrl(BIO *bp,int cmd,long larg);
long BIO_int_ctrl(BIO *bp,int cmd,long larg,int iarg);
int BIO_reset(BIO *b);
int BIO_seek(BIO *b, int ofs);
int BIO_tell(BIO *b);
int BIO_flush(BIO *b);
int BIO_eof(BIO *b);
int BIO_set_close(BIO *b,long flag);
int BIO_get_close(BIO *b);
int BIO_pending(BIO *b);
int BIO_wpending(BIO *b);
size_t BIO_ctrl_pending(BIO *b);
size_t BIO_ctrl_wpending(BIO *b);
```

其实，在这些函数中，除了

BIO_ctrl,

BIO_callback_ctrl,
BIO_ptr_ctrl,
BIO_int_ctrl,
BIO_ctrl_pending,
BIO_ctrl_wpending 是真正的函数外，其它都是宏定义，而且，在这些函数中，除了 BIO_ctrl,BIO_callback_ctrl，其它基本上都是简单的 BIO_ctrl 输入不同的参数的调用。
下面就一个一个介绍这些函数。

【BIO_ctrl】

从上面的叙述可以知道，BIO_ctrl 是整个控制函数中最基本的函数，它支持不同的命令输入，从而产生不同的功能，由此，它也就衍生了许多其它函数，作为一个比较地层的控制函数，一般来说用户并不需要直接调用它，因为在它之上已经使用宏定义和函数调用的形式建造了许多直接面向用户的函数。

filter 型的 BIO 没有定义 BIO_ctrl 功能，如果对他们调用这个函数，他们就简单的把命令传到 BIO 链中的下一个 BIO。也就是说，通常可以不用直接调用一个 BIO 的 BIO_ctrl 函数，只需要在它所在的 BIO 链上调用该函数，那么 BIO 链就会自动将该调用函数传到相应的 BIO 上去。这样可能就会导致一些意想不到的结果，比如，在目前的 filter 型 BIO 中没有实现 BIO_seek() 函数（大家待会就会明白 BIO_seek 就是 BIO_ctrl 的简单宏定义），但如果在这个 BIO 链上的末尾是一个文件或文件描述符型 BIO，那么这个调用也会返回成功的结果。

对于 source/sink 型 BIO 来说，如果他们不认得 BIO_ctrl 所定义的操作，那么就返回 0。

【BIO_callback_ctrl】

这个函数是这组控制函数中唯一一个不是通过调用 BIO_ctrl 建立起来的，它有自己的实现函数，而且跟 BIO_ctrl 毫不相干。跟 BIO_ctrl 一样，它也是比较底层的控制函数，在它上面也定义了一些直接面向用户的控制函数，一般来说，用户不需要直接调用该函数。

需要说明的是，该函数和 BIO_ctrl 函数为了实现不同类型 BIO 具有不同的 BIO_ctrl 控制功能，他们的操作基本上都是由各个 BIO 的 callback 函数来定义的。这是不同的 BIO 能灵活实现不同功能的根本所在。

【BIO_ptr_ctrl 和 BIO_int_ctrl】

这两个函数都是简单的调用了 BIO_ctrl 函数，不同的是，后者是输入了四个参数并传入到 BIO_ctrl 函数中，简单返回了调用 BIO_ctrl 返回的返回值；而前者只输入了三个参数，最后一个 BIO_ctrl 参数是作为输出参数并作为返回值的。

【BIO_reset】

该函数是 BIO_ctrl 的宏定义函数，为了大家对 BIO_ctrl 的宏定义函数有一个感性的认识，我把这个宏定义写出来，如下：

```
#define BIO_reset(b) (int)BIO_ctrl(b,BIO_CTRL_RESET,0,NULL)
```

这就是 BIO_ctrl 的典型宏定义方式，它通过这种方式产生了大量的控制函数。顾名思义，BIO_reset 函数只是简单的将 BIO 的状态设回到初始化的时候的状态，比如文件 BIO，调用该函数就是将文件指针指向文件开始位置。

一般来说，调用成功的时候该函数返回 1，失败的时候返回 0 或－1；但是文件 BIO 是一个例外，成功调用的时候返回 0，失败的时候返回－1。

【BIO_seek】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下：

```
#define BIO_seek(b,ofs) (int)BIO_ctrl(b,BIO_C_FILE_SEEK,ofs,NULL)
```

该函数将文件相关的 BIO（文件和文件描述符类型）的文件指针知道距离开始位置 ofs(输入参数)字节的位置上。调用成功的时候，返回文件的位置指针，否则返回－1；但是文件 BIO 例外，成功的时候返回 0，失败的时候返回－1。

【BIO_tell】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下：

```
#define BIO_tell(b) (int)BIO_ctrl(b,BIO_C_FILE_TELL,0,NULL)
```

该函数返回了文件相关 BIO 的当前文件指针位置。跟 BIO_seek 一样，调用成功的时候，

返回文件的位置指针，否则返回－1；但是文件 BIO 例外，成功的时候返回 0，失败的时候返回－1。

【BIO_flush】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下：

#define BIO_flush(b) (int)BIO_ctrl(b,BIO_CTRL_FLUSH,0,NULL)

该函数用来将 BIO 内部缓冲区的数据都写出去，有些时候，也用于为了根据 EOF 查看是否还有数据可以写。调用成功的时候该函数返回 1，失败的时候返回 0 或－1。之所以失败的时候返回 0 或者－1，是为了标志该操作是否需要稍后以跟 BIO_write()相同的方式重试。这时候，应该调用 BIO_should_retry()函数，当然，正常的情况下该函数的调用应该是失败的。

【BIO_eof】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下

#define BIO_eof(b) (int)BIO_ctrl(b,BIO_CTRL_EOF,0,NULL)

如果 BIO 读到 EOF，该函数返回 1，至于 EOF 的具体定义，根据 BIO 的类型各不相同。如果没有读到 EOF，该函数返回 0.

【BIO_set_close】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下：

#define BIO_set_close(b,c) (int)BIO_ctrl(b,BIO_CTRL_SET_CLOSE,(c),NULL)

该函数设置 BIO 的关闭标志，该标志可以为 BIO_CLOSE 或 BIO_NOCLOSE。一般来说，该标志是为了指示在 source/sink 型 BIO 释放该 BIO 的时候是否关闭其下层的 I/O 流。该函数总是返回 1。

【BIO_get_close】

该函数也是 BIO_ctrl 的宏定义函数，其定义如下：

#define BIO_get_close(b) (int)BIO_ctrl(b,BIO_CTRL_GET_CLOSE,0,NULL)

该函数读取 BIO 的关闭标志，返回 BIO_CLOSE 或 BIO_NOCLOSE。

【BIO_pending、BIO_wpending、BIO_ctrl_pending 和 BIO_ctrl_wpending】

这些函数都是用来得到 BIO 中读缓存或写缓存中字符的数目的，返回相应缓存中字符的数目。

前面两个函数也是 BIO_ctrl 的宏定义函数，其定义如下：

#define BIO_pending(b) (int)BIO_ctrl(b,BIO_CTRL_PENDING,0,NULL)

#define BIO_wpending(b) (int)BIO_ctrl(b,BIO_CTRL_WPENDING,0,NULL)

后两个函数功能跟他们是一样的，只不过他们是通过调用 BIO_ctrl 函数实现的，而不是宏定义。此外，前面两个函数返回的是 int 型，而后面两个函数返回的是 size_t 型。

需要注意的是，BIO_pending 和 BIO_wpending 并不是在所有情况下都能很可靠地得到缓存数据的数量，比如在文件 BIO 中，有些数据可能在文件内部结构的缓存中是有效的，但是不可能简单的在 BIO 中得到这些数据的数量。而在有些类型 BIO 中，这两个函数可能还不支持。基于此，openssl 作者本身也建议一般不要使用这两个函数，而是使用后面两个，除非你对你所做的操作非常清楚和了解。

openssl 之 BIO 系列之 5---CallBack 函数及其控制

CallBack 函数及其控制---根据 openssl doc/crypto/bio/bio_set_callback.pod 翻译和自己的理解写成

通过前面的介绍大家已经知道，BIO 的 callback 函数是非常重要的，是实现 BIO 多态性的一个关键因素之一，BIO 提供的 callback 控制系列函数有五个，其实都是一些宏定义，下面是它的声明和定义（openssl/bio.h）：

#define BIO_set_callback(b,cb) ((b)->callback=(cb))

#define BIO_get_callback(b) ((b)->callback)

#define BIO_set_callback_arg(b,arg) ((b)->cb_arg=(char *)(arg))

#define BIO_get_callback_arg(b) ((b)->cb_arg)

其中，callback 函数本身的声明如下：

typedef long callback(BIO *b, int oper, const char *argp,

int argi, long argl, long retvalue);

此外，还有一个用于调试目的的函数，其实声明如下：

long BIO_debug_callback(BIO *bio,int cmd,const char *argp,int argi,long argl,long ret);

如果要看具体的例子，那么在文件 crypto/bio/bio_cb.c 的函数 BIO_debug_callback() 本身就是一个非常好的例子。

下面，我们从 callback 函数本身开始分别简单介绍这些函数的作用。

【callback】

callback 函数在 BIO 中非常重要，许多控制功能都是要通过 callbank 函数协助完成的，比如 BIO 要执行释放的操作 BIO_free，那么其实它是先调用 callback 函数设置下面的操作将是释放操作（控制码:BIO_CB_FREE），然后才调用别的相关函数执行真正的操作，在后面我们会列出这些控制功能函数，并简单说明 callback 函数是怎么在这些功能的实现中使用的。现在，我先简单介绍 callback 函数的各个参数：

(参数名字参看说明的函数的声明)

参数---b

这是 callback 函数的输入参数，也就是 callback 函数对应的 BIO

参数---oper

设置 BIO 将要执行的操作，有些操作，callback 函数将被调用两次，一次实在实际操作之前，一次实在实际操作之后，在后面的调用的时候，一般是将 oper 和 BIO_CB_RETURN 相或操作后作为参数的。也就是说，后一次调用的时候 oper 参数应该使用 oper|BIO_CB_RETURN。

参数---argp, argi,argl

这些参数根据 oper 定义的操作的不同而不一样，是在相应操作中要用到的参数。

参数---retvalue

这是默认的 callback 函数返回值，也就睡说，如果没有提供 BIO 没有提供相应的 callback 函数，那么就会返回这个值。真正的返回值是 callback 函数本身提供的。如果在实际的操作之前调用 callback 函数，并且这时候 retvalue 参数设置为 1，如果 callback 的函数返回值无效，那么对 callback 函数的调用就会导致程序立刻返回，BIO 的操作就不会执行。

一般情况下，callback 函数在执行完后都应该返回 retvalue 的值，除非该操作有特别的目的要修改这个返回值。

下面简单列出我们比较熟悉的一些跟 **callback 函数相关的 BIO 函数使用 callback 函数的情况：**

1.BIO_free(b)

在执行该操作之前，调用了 callback(b, BIO_CB_FREE, NULL, 0L, 0L, 1L)

2.BIO_read(b,out,outl)

在执行该操作之前，调用了 callback(b, BIO_CB_READ, out, outl, 0L, 1L),之后调用了 callback(b, BIO_CB_READ|BIO_CB_RETURN, out, outl, 0L,retvalue)，大家可以看到，这就是我们上面说明过的情况，即两次调用 callback 的操作，后面一次 oper 的参数需要或上 BIO_CB_RETURN。

3.BIO_write(b,in,inl)

在执行该操作之前，调用了 callback(b, BIO_CB_WRITE, in, inl, 0L, 1L),之后调用了 callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, inl, 0L, retvalue)

4.BIO_gets(b,out,outl)

在执行该操作之前，调用了 callback(b, BIO_CB_GETS, out, outl, 0L, 1L),之后调用了 callback(b, BIO_CB_GETS|BIO_CB_RETURN, out, outl, 0L, retvalue)

5.BIO_puts(b, in)

在执行该操作之前，调用了 callback(b, BIO_CB_WRITE, in, 0, 0L, 1L) ,之后调用了 callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, 0, 0L,retvalue)

6.BIO_ctrl(BIO *b, int cmd, long larg, void *parg)

在执行该操作之前，调用了 callback(b,BIO_CB_CTRL,parg,cmd,larg,1L), 之后调用了 callback(b,BIO_CB_CTRL|BIO_CB_RETURN,parg,cmd, larg,ret)

【BIO_set_callback 和 BIO_get_callback】

这两个函数用于设置和返回 BIO 中的 callback 函数，它们都是宏定义，根据前面的叙述我们已经知道，callback 函数在许多高层的操作中都使用了，因为它能用于调试跟踪的目的或更改 BIO 的操作，具有很大的灵活性，所以这两个函数也就有用武之地了。

【BIO_set_callback_arg 和 IO_get_callback_arg】

顾名思义，这两个函数用了设置和得到 callback 函数中的参数。

【BIO_debug_callback】

这是一个标准的调试信息输出函数，它把相关 BIO 执行的所有操作信息都打印输出到制定的地方。如果 callback 参数没有指定输出这些信息的 BIO 口，那么就会默认使用 stderr 作为信息输出端口。

openssl 之 BIO 系列之 6---BIO 的 IO 操作函数

BIO 的 IO 操作函数---根据 openssl doc/crypto/bio/bio_read.pod 翻译和自己的理解写成

这些函数是 BIO 的基本读写操作函数，包括四个，他们的定义如下（openssl/bio.h）：

```
int BIO_read(BIO *b, void *buf, int len);
int BIO_gets(BIO *b,char *buf, int size);
int BIO_write(BIO *b, const void *buf, int len);
int BIO_puts(BIO *b,const char *buf);
```

【BIO_read】

从 BIO 接口中读出指定数量字节 len 的数据并存储到 buf 中。成功就返回真正读出的数据的长度，失败返回 0 或－1，如果该 BIO 没有实现本函数则返回－2。

【BIO_gets】

该函数从 BIO 中读取一行长度最大为 size 的数据。通常情况下，该函数会以最大长度限制读取一行数据，但是也有例外，比如 digest 型的 BIO，该函数会计算并返回整个 digest 信息。此外，有些 BIO 可能不支持这个函数。成功就返回真正读出的数据的长度，失败返回 0 或－1，如果该 BIO 没有实现本函数则返回－2。需要注意的时，如果相应的 BIO 不支持这个函数，那么对该函数的调用可能导致 BIO 链自动增加一个 buffer 型的 BIO。

【BIO_write】

往 BIO 中写入长度为 len 的数据。成功就返回真正写入的数据的长度，失败返回 0 或－1，如果该 BIO 没有实现本函数则返回－2。

【BIO_puts】

往 BIO 中写入一个以 NULL 为结束符的字符串，成功就返回真正写入的数据的长度，失败返回 0 或－1，如果该 BIO 没有实现本函数则返回－2。

需要注意的是，返回指为 0 或－1 的时候并不一定就是发生了错误。在非阻塞型的 source/sink 型或其它一些特定类型的 BIO 中，这仅仅代表目前没有数据可以读取，需要稍后再进行该操作。

有时候，你可能会使用了阻塞类型的 sokect 使用的一些系统调用技术（如 select,poll,equivalent）来决定 BIO 中是否有有效的数据被 read 函数读取，但建议不要在阻塞型的接口中使用这些技术，因为这样的情况下如果调用 BIO_read 就会导致在底层的 IO 中多次调用 read 函数，从而导致端口阻塞。建议 select（或 equivalent）应该和非阻塞型的 IO 一起使用，可以在失败之后能够重新读取该 IO，而不是阻塞住了。

关于 BIO 的 IO 操作为什么会失败以及怎么处理这些情况请参加 BIO_should_retry()函数的说明文档。

openssl 之 BIO 系列之 7---BIO 链的操作

BIO 链的操作---根据 openssl doc/crypto/bio/bio_push.pod 翻译和自己的理解写成

我在介绍 BIO 结构的时候说过，BIO 结构其实是一个链式结构，单个 BIO 是只有一个环节的 BIO 链的特例，那么我们怎么构造或在一个 BIO 链中增加一个 BIO，怎么从一个 BIO 链中删除一个 BIO 呢，那么本节就是专门讲述这个问题的。

其实，在 openssl 中，针对 BIO 链的操作还是很简单的，仅仅包括两个函数（openssl/bio.h）：

```
BIO * BIO_push (BIO *b,BIO *append);
BIO * BIO_pop (BIO *b);
```

【BIO_push】

该函数把参数中名为 append 的 BIO 附加到名为 b 的 BIO 上，并返回 b。其实，openssl 作者本身也认识到，BIO_push 的函数名字可能会导致误会，因为 BIO_push 函数其实只是将两个 BIO 连接起来，而不是 Push 的功能，应该是 join 才对。

我们举几个简单的例子说明 BIO_push 的作用，假设 md1、md2 是 digest 类型的 BIO，b64 是 Base64 类型的 BIO，而 f 是 file 类型的 BIO，那么如果执行操作

BIO_push(b64, f);

那么就会形成一个 b64-f 的链。然后再执行下面的操作：

BIO_push(md2, b64);

BIO_push(md1, md2);

那么就会形成 md1-md2-b64-f 的 BIO 链，大家可以看到，在构造完一个 BIO 后，头一个 BIO 就代表了整个 BIO 链，这根链表的概念几乎是一样的。

这时候，任何写往 md1 的数据都会经过 md1,md2 的摘要（或说 hush 运算），然后经过 base64 编码，最后写入文件 f。可以看到，构造一条好的 BIO 链后，操作是非常方便的，你不用再关心具体的事情了，整个 BIO 链会自动将数据进行指定操作的系列处理。

需要注意的是，如果是读操作，那么数据会从相反的方向传递和处理，对于上面的 BIO 链，数据会从 f 文件读出，然后经过 base64 解码，然后经过 md1,md2 编码，最后读出。

【BIO_pop】

该函数把名为 b 的 BIO 从一个 BIO 链中移除并返回下一个 BIO，如果没有下一个 BIO，那么就返回 NULL。被移除的 BIO 就成为一个单个的 BIO，跟原来的 BIO 链就没有关系了，这样你可以把它释放或连接到另一个 BIO 上去。可以看到，如果是单个 BIO 的时候，该操作是没有任何意义的。

如果你执行操作：

BIO_pop(md2);

那么返回值将为 b64,而 md2 从上述的链中移除，形成一个新的 md1-b64-f 的 BIO 链，对于数据操作来说，还是往 md1 读写，没有什么变化，但是底层处理过程已经发生变化了，这就是封装与透明的概念。可以看到，虽然 BIO_pop 参数只是一个 BIO，但该操作直接的后果会对该 BIO 所在的链产生影响，所以，当 BIO 所在的链不一样的时候，其结果是不一样的。

此外：BIO_push 和 BIO_pop 操作还可能导致其它一些附加的结果，一些相关的 BIO 可能会调用一些控制操作，这些具体的细节因为各个类型的 BIO 不一样，在他们各自的说明中会有说明。

openssl 之 BIO 系列之 8---读写出错控制

读写出错控制---根据 openssl doc/crypto/bio/bio_should_retry.pod 翻译和自己的理解写成

当 BIO_read 或 BIO_write 函数调用出错的时候，BIO 本身提供了一组出错原因的诊断函数，他们定义如下（openssl/bio.h）：

```
#define BIO_should_read(a) ((a)->flags & BIO_FLAGS_READ)
#define BIO_should_write(a) ((a)->flags & BIO_FLAGS_WRITE)
#define BIO_should_io_special(a) ((a)->flags & BIO_FLAGS_IO_SPECIAL)
#define BIO_retry_type(a) ((a)->flags & BIO_FLAGS_RWS)
#define BIO_should_retry(a) ((a)->flags & BIO_FLAGS_SHOULD_RETRY)
#define BIO_FLAGS_READ 0x01
#define BIO_FLAGS_WRITE 0x02
#define BIO_FLAGS_IO_SPECIAL 0x04
#define BIO_FLAGS_RWS (BIO_FLAGS_READ|BIO_FLAGS_WRITE|BIO_FLAGS_IO_SPECIAL)
#define BIO_FLAGS_SHOULD_RETRY 0x08
BIO * BIO_get_retry_BIO(BIO *bio, int *reason);
int BIO_get_retry_reason(BIO *bio);
```

因为这些函数是用于决定为什么 BIO 在读写数据的时候不能读出或写入数据，所以他们一般也是在执行 BIO_read 或 BIO_write 操作之后被调用的。

【BIO_should_retry 】

如果读写出错的情况是要求程序稍后重试，那么该函数返回 true.如果该函数返回 false，这时候判定错误情况就要根据 BIO 的类型和 BIO 操作的返回值来确定了。比如，如果对 socket 类型的 BIO 调用 BIO_read 操作并且返回值为 0，此时 BIO_should_retry 返回 false 就说明 socket 连接已经关闭了。而如果是 file 类型的 BIO 出现这样的情况，那说明就是读到文件 eof 了。有

些类型 BIO 还会提供更多的出错信息，具体情况参见各自的说明。

如果 BIO 下层 I/O 结构是阻塞模式的，那么几乎所有（SSL 类型 BIO 例外）BIO 类型都不会返回重试的情况（就是说调用 BIO_should_retry 不会返回 true），因为这时候对下层 I/O 的调用根本不会进行。所以建议如果你的应用程序能够判定该类型 BIO 在执行 IO 操作后不会出现重试的情况时，就不要调用 BIO_should_retry 函数。file 类型 BIO 就是这样的一个典型例子。

SSL 类型的 BIO 是上述规则的唯一例外，也就是说，既便在阻塞型的 I/O 结构中，如果在调用 BIO_read 的时候发生了握手的过程，它也能会返回重试要求（调用 BIO_should_retry 返回 true）。在这种情况下，应用程序可以立刻重新执行失败的 I/O 操作，或者在底层的 I/O 结构中设置为 SSL_MODE_AUTO_RETRY，那么就可以避免出现这种失败的情况。

如果应用程序在非阻塞型 BIO 中调用 IO 操作失败后立刻重试，那么可能导致效率很低，因为在数据允许读取或有效之前，调用会重复返回失败结果。所以，正常的应用应该是等到需要的条件满足之后，程序才执行相关的调用，至于具体怎么做，就跟底层的 IO 结构有关了。例如，如果一个底层 IO 是一个 soket，并且 BIO_should_retry 返回 true,那么可以调用 select()来等待数据有效之后再重试 IO 操作。在一个线程中，可以使用一个 select()来处理多个非阻塞型的 BIO，不过，这时候执行效率可能出现非常低的情况，比如如果其中一个延时很长的 SSL 类型 BIO 在握手的时候就会导致这种情况。

在阻塞型的 IO 结构中，对数据的读取操作可能会导致无限期的阻塞，其情况跟系统的 IO 结构函数有关。我们当然不期望出现这种情况，解决的办法之一是尽量使用非阻塞型的 IO 结构和使用 select 函数（或 equivalent）来设置等待时间。

【BIO_should_read】

该函数返回 true 如果导致 IO 操作失败的原因是 BIO 此时要读数据。

【BIO_should_write】

该函数返回 true 如果导致 IO 操作失败的原因是 BIO 此时要写数据。

【BIO_should_io_special】

该函数返回 true 如果导致 IO 操作失败的原因是特殊的（也就是读写之外的原因）

【BIO_get_retry_reason】

返回失败的原因，其代码包括 BIO_FLAGS_READ, BIO_FLAGS_WRITE 和 BIO_FLAGS_IO_SPECIAL。目前的 BIO 类型只返回其中之一。如果输入的 BIO 是产生特殊出错情况的 BIO，那么该函数返回错误的原因代码，就跟 BIO_get_retry_BIO()返回的 reason 一样。

【BIO_get_retry_BIO】

该函数给出特殊情况错误的简短原因，它返回出错的 BIO，如果 reason 不是设置为 NULL，它会包含错误代码，错误码的含义以及下一步应该采取的处理措施应该根据发生这种情况下各种 BIO 的类型而定。

openssl 之 BIO 系列之 9---BIO 对的创建和应用

BIO 对的创建和应用---根据 openssl doc/crypto/bio/bio_new_bio_pair.pod 翻译和自己的理解写成

BIO 对是 BIO 中专门创建的一对缓存 BIO，要创建 BIO 对，调用下面定义的函数（openssl\bio.h）：

int BIO_new_bio_pair(BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2);

这个函数调用成功后返回 1，这时候 bio1 和 bio2 都是有效的了；否则就返回 0，而 bio1 和 bio2 就会设为 NULL，这是后可以检测出错堆栈以得到更多错误信息。

这个 BIO 对创建之后，它的两端都能作为数据缓冲的输入和输出。典型的应用是它一端和 SSL 的 IO 连接，而另一端则被应用控制，这样，应用程序就不需要直接和网络连接打交道了。

这两个 BIO 对的功能是完全对称的，它们的缓冲区的大小由参数 writebuf1 和 writebuf2 决定，如果给定的大小是 0，那么该函数就会使用缺省的缓存大小。BIO_new_bio_pair 不会检查 bio1 和 bio2 是否真的指向其它 BIO，bio1 和 bio2 的值都被重写，但是在此之前不会调用 BIO_free()函数。所以，在使用 bio1 和 bio2 之前，必须自己保证这两个变量是空的 BIO，否则可能造成内存泄漏。

值得注意的是，虽然这两个 BIO 是一对的和一起创建的，但是却必须分别释放。之所以这

样做，是有其重要原因的，因为有些 SSL 函数，如 SSL_set_bio 或 BIO_free 会隐含调用 BIO_free 函数，所以这时候另一端的 BIO 就只能单独释放了。

为了让大家对 BIO 对的应用模型有一个感性的认识，下面举一个简单的例子说明问题。

BIO 对能给提供应用程序中对网络处理的完全控制能力，程序可以对根据需要调用 soket 的 select()函数，同时却可以避免直接处理 SSL 接口。下面是使用 BIO_new_bio_pair 的简单代码模型：

```
BIO *internal_bio, *network_bio;
...
BIO_new_bio_pair(internal_bio, 0, network_bio, 0);
SSL_set_bio(ssl, internal_bio);
SSL_operations();
...
application | TLS-engine
| |
+----------> SSL_operations()
| /\ ||
| || \/
| BIO-pair (internal_bio)
+----------< BIO-pair (network_bio)
| |
socket |
...
SSL_free(ssl); /* 隐式释放 internal_bio */
BIO_free(network_bio); /* 显式释放 network_bio*/
...
```

因为 BIO 对只会简单的缓存数据，而不会直接涉及到连接，所以它看起来就想非阻塞型的接口，如果写缓存满了或读缓存空的时候，调用 IO 函数就会立刻返回。也就是说，应用程序必须自己对写缓存执行 flush 操作或对读缓存执行 fill 操作。可以使用前面介绍过的 BIO_ctrl_pending 函数看看是否有数据在缓存里面并需要传输到网络上去；为了下面的 SSL_operation 能够正确执行，可以调用 BIO_ctrl_get_read_request 函数，以决定需要在写缓存写入多少数据。上面两个函数可以保证正确的 SSL 操作的进行。

需要注意的是，SSL_operation 的调用可能会出现返回 ERROR_SSL_WANT_READ 值，但这时候写缓存却还有数据的情况，所以应用程序不能简单的根据这个错误代码进行判断，而必须保证写缓存以及执行过 flush 操作了，否则就会造成死锁现象，因为另一端可能知道等到有数据了才会继续进行下面的操作。

openssl 之 BIO 系列之 10---BIO 链的定位操作

BIO 链的定位操作---根据 openssl doc/crypto/bio/bio_find_type.pod 翻译和自己的理解写成

前面的一篇文章讲过 BIO 链的构造方法，这里讲的是在一个 BIO 链中，怎么查找一个特定的 BIO，怎么遍历 BIO 链中的每一个 BIO，这组函数定义如下（openssl/bio.h）：

```
BIO * BIO_find_type(BIO *b,int bio_type);
BIO * BIO_next(BIO *b);
#define BIO_method_type(b) ((b)->method->type)
```

可以看到，这组函数中有两个是真正的函数，另一个则是宏定义，其中，bio_type 的值定义如下：

```
#define BIO_TYPE_NONE 0
#define BIO_TYPE_MEM (1|0x0400)
#define BIO_TYPE_FILE (2|0x0400)
#define BIO_TYPE_FD (4|0x0400|0x0100)
#define BIO_TYPE_SOCKET (5|0x0400|0x0100)
#define BIO_TYPE_NULL (6|0x0400)
#define BIO_TYPE_SSL (7|0x0200)
#define BIO_TYPE_MD (8|0x0200)
```

```
#define BIO_TYPE_BUFFER (9|0x0200)
#define BIO_TYPE_CIPHER (10|0x0200)
#define BIO_TYPE_BASE64 (11|0x0200)
#define BIO_TYPE_CONNECT (12|0x0400|0x0100)
#define BIO_TYPE_ACCEPT (13|0x0400|0x0100)
#define BIO_TYPE_PROXY_CLIENT (14|0x0200)
#define BIO_TYPE_PROXY_SERVER (15|0x0200)
#define BIO_TYPE_NBIO_TEST (16|0x0200)
#define BIO_TYPE_NULL_FILTER (17|0x0200)
#define BIO_TYPE_BER (18|0x0200)
#define BIO_TYPE_BIO (19|0x0400)
#define BIO_TYPE_DESCRIPTOR 0x0100
#define BIO_TYPE_FILTER 0x0200
#define BIO_TYPE_SOURCE_SINK 0x0400
```

可以看到，这些定义大部分都是根据各种 BIO 类型来命名的，但并不是跟现有的 BIO 类型是一一对应的，在以后的文章里，我会对这些 BIO 类型一一进行介绍，现在大家只要有一个概念就可以了。

【BIO_find_type】

该函数在给定的 BIO 链中根据特定的 BIO 类型 bio_type 进行搜索，搜索的起始位置就是 b。如果给定的类型是一个特定的实现类型，那么就会搜索一个给类型的 BIO；如果只是一个总体的类型定义，如 BIO_TYPE_SOURCE_SINK（就是 sourc/sink 类型的 BIO），那么属于这种类型的最先找到的 BIO 就是符合条件的。在找到符合的 BIO 后，BIO_find_type 返回该 BIO，否则返回 NULL。需要注意的是，如果你使用的 0.9.5a 以前版本，如果给输入参数 b 赋值为 NULL，可能引起异常错误！

【BIO_next】

该函数顾名思义，是返回当前 BIO 所在的 BIO 链中的下一个 BIO，所以，它可以用来遍历整个 BIO 链，并且可以跟 BIO_find_type 函数结合起来，在整个 BIO 链中找出所有特定类型的 BIO。这个函数是在 0.9.6 版本新加的，以前的版本要使用这个功能，只能使用 bio->next_bio 来定位了。

【BIO_method_type】

该函数返回给定的 BIO 的类型。

下面给出一个在一个 BIO 链中找出所有 digest 类型 BIO 的例子：

```
BIO *btmp;
btmp = in_bio; /* in_bio 是被搜索的 BIO 链 */
do {
btmp = BIO_find_type(btmp, BIO_TYPE_MD);
if(btmp == NULL) break; /* 如果没有找到*/
/* btmp 是一个 digest 类型的 BIO，做些你需要做的处理 ...*/
...
btmp = BIO_next(btmp);
} while(btmp);
```

到此为止，就已经基本写完了 BIO 的基础知识方面的东西，下面的文章将开始对每一个具体的 BIO 类型进行介绍，我想有了前面的这些铺垫和知识，后面的就轻松多了。请大家继续关注 http://gdwzh.126.com 的 openssl 专业论坛！

openssl 之 BIO 系列之 11---文件(file)类型 BIO

文件(file)类型 BIO---根据 openssl doc/crypto/bio/bio_s_file.pod 翻译和自己的理解写成

前面我们已经介绍了很多 BIO 的基本构造和操作，现在，我们开始针对每一个类型 BIO 进行进一步的介绍，这些介绍都是基本基于 openssl 的帮助文档的，我尽可能加入自己的一些理解，理清思路。在开始这部分之前，我一直在想该从哪一种类型的 BIO 开始比较合适，因为这些 BIO 类型之间有些还是有相互联系的，比如 BIO_s_bio 型就和 BIO_f_ssl 有关系，最后，考虑到大家对文件操作都比较熟悉，而且该类型 BIO 比较独立，我决定从这个 BIO 开始介绍，

随后逐步介绍其它 source/sink 型 BIO，然后再介绍 filter 型 BIO。

文件(file)类型 BIO 的相关函数和定义如下（openssl\bio.h）：

```
BIO_METHOD * BIO_s_file(void);
BIO *BIO_new_file(const char *filename, const char *mode);
BIO *BIO_new_fp(FILE *stream, int flags);
BIO_set_fp(BIO *b,FILE *fp, int flags);
BIO_get_fp(BIO *b,FILE **fpp);
int BIO_read_filename(BIO *b, char *name)
int BIO_write_filename(BIO *b, char *name)
int BIO_append_filename(BIO *b, char *name)
int BIO_rw_filename(BIO *b, char *name)
```

下面逐一介绍它们的作用和用法。

【BIO_s_file】

经过前面的介绍，大家应该对这种类型的函数结构很熟悉了，他们就是生成 BIO 类型的基本构造函数，BIO_s_file 返回 file 类型的 BIO，file 类型的 BIO 封装了一个标准的文件结构，它是一种 source/sink 型 BIO。file 类型的 BIO_METHOD 结构定义如下：

```
static BIO_METHOD methods_filep=
 {
 BIO_TYPE_FILE,
 "FILE pointer",
 file_write,
 file_read,
 file_puts,
 file_gets,
 file_ctrl,
 file_new,
 file_free,
 NULL,
 };
```

可以看到，file 类型的 BIO 定义了 7 个函数，这些函数的实现都在 Crypto\bio\bss_file.c 里面，大家如果要了解该类型 BIO 的函数实现，可以参考该文件。事实上，BIO_s_file 只是简单返回一个 file 类型的 BIO_METHOD 的结构的指针，其函数实现如下：

```
BIO_METHOD *BIO_s_file(void)
 {
 return(&methods_filep);
 }
```

其实，从这个结构可以略见 BIO 的实现的一班，即 BIO 的所有动作都是根据它的 BIO_METHOD 的类型（第一个参数）来决定它的动作和行为的，从而实现 BIO 对各种类型的多态实现。

在 file 类型中，使用前面介绍过的 BIO_read 和 BIO_write 对底层的 file 数据流进行读写操作，file 类型 BIO 是支持 BIO_gets 和 BIO_puts 函数的（大家如果忘了这些函数的作用，请参考前面的《BIO 系列之 6---BIO 的 IO 操作函数》）。

BIO_flush 函数在 file 类型 BIO 中只是简单调用了 API 函数 fflush。

BIO_reset 函数则将文件指针重新指向文件的开始位置，它调用 fseek(stream,0,0)函数实现该功能。

BIO_seek 函数将文件指针位置至于所定义的位置 ofs 上（从文件开头开始计算的偏移 ofs），它调用了文件的操作函数 fseek(stream,ofs,0)，是一个宏定义形式的函数，需要注意的是，因为该函数调用了 fseek 函数，所以成功的时候返回 0，失败的时候返回－1，这是跟标准 BIO_seek 函数定义不一样的，因为标准的定义是成功返回 1，失败返回非正值。

BIO_eof 调用了 feof 函数。

如果在 BIO 结构中设置了 BIO_CLOSE 的标志，则在 BIO 释放的时候会自动调用 fclose 函

数。

【BIO_new_file】

该函数根据给定的 mode 类型创建了一个文件 BIO，mode 参数的函数跟 fopen 函数中 mode 参数的含义是一样的。返回的 BIO 设置了 BIO_CLOSE 标志。调用成功返回一个 BIO，否则返回 NULL。

事实上，该函数先调用了 fopen 函数打开一个文件，然后调用 BIO_new 函数创建一个 file 类型 BIO，最后调用函数 BIO_set_fp 函数给 BIO 结构跟相关的 file 帮定。

【BIO_new_fp】

用文件描述符创建一个 file 类型 BIO，参数 Flags 可以为 BIO_CLOSE,BIO_NOCLOSE（关闭标志）以及 BIO_FP_TEXT（将文件设置为文本模式，默认的是二进制模式，该选项只对 Win32 平台有效）。事实上，该函数调用 BIO_new 函数创建一个 file 类型 BIO，然后调用函数 BIO_set_fp 函数给 BIO 结构跟相关的 file 帮定。需要注意的是，如果下层封装的是 stdout,stdin 和 stderr，他们因为跟一般的是不关闭的，所以应该设置 BIO_NOCLOSE 标志。调用成功返回一个 BIO，否则返回 NULL。

【BIO_set_fp】

该函数将 BIO 跟文件描述符 fp 帮定在一起，其参数 flags 的含义跟 BIO_new_fp 是一样的。该函数是一个宏定义函数。调用成功返回 1，否则返回 0，不过目前的实现是从来不会出现失败情况的。

【BIO_get_fp】

该函数返回 file 类型 BIO 中文件描述符，也是一个宏定义。调用成功返回 1，否则返回 0，不过目前的实现是从来不会出现失败情况的。

【BIO_tell】

返回位置指针的值。是一个宏定义函数。

【BIO_read_filename, BIO_write_filename, BIO_append_filename， BIO_rw_filename】

这四个函数分别设置 BIO 的读文件名，写文件名，附加文件名以及读写的文件名。他们都是一些宏定义函数。调用成功返回 1，否则返回 0。

从上面各函数的介绍可以看出，因为 BIO 调用了底层的各种操作函数，所以，如果底层函数的调用有任何异常，都会反映在 BIO 的调用上。

下面举几个 BIO 文件类型操作的简单例子：

1.最简单的实例程序

```
BIO *bio_out;
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_printf(bio_out, "Hello World\n");
```

2.上述例子的另一种实现方法

```
BIO *bio_out;
bio_out = BIO_new(BIO_s_file());
if(bio_out == NULL) /*  出错*/
if(!BIO_set_fp(bio_out, stdout, BIO_NOCLOSE)) /*  出错则将文件流定向到标准输出*/
BIO_printf(bio_out, "Hello World\n");
```

3.写文件操作

```
BIO *out;
out = BIO_new_file("filename.txt", "w");
if(!out) /*出错  */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

4.上述例子的另一种实现方法

```
BIO *out;
out = BIO_new(BIO_s_file());
if(out == NULL) /* Error ... */
if(!BIO_write_filename(out, "filename.txt")) /* Error ... */
```

```
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

openssl 之 BIO 系列之 12---文件描述符(fd)类型 BIO

文件描述符(fd)类型 BIO---根据 openssl doc\crypto\bio_s_fd.pod 翻译和自己的理解写成

文件描述符类型 BIO 也是一个 source/sink 型的 BIO,它定义了以下一些类型的函数（openssl\bio.h）：

```
BIO_METHOD * BIO_s_fd(void);
#define BIO_set_fd(b,fd,c) BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c) BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)
BIO *BIO_new_fd(int fd, int close_flag);
```

有一点需要说明的是，虽然存在 bss_fd.c 文件，但是关于 fd 类型的 BIO 的实现函数，并非真正在 bss_fd.c 里面，而是在 bss_sock.c 里面，bss_fd.c 这是简单包含了 bss_sock.c 文件，所以大家要找实现函数，应该到 bss_sock.c 里面找。

【BIO_s_fd】

该函数返回一个文件描述符类型的 BIO_METHOD 结构，它封装了文件描述符类型的一些规则，如 read()和 write()函数等。fd 类型的 BIO_METHOD 结构如下：

```
static BIO_METHOD methods_fdp=
 {
 BIO_TYPE_FD,"file descriptor",
 fd_write,
 fd_read,
 fd_puts,
 NULL, /* fd_gets, */
 fd_ctrl,
 fd_new,
 fd_free,
 NULL,
 };
```

可见，跟 file 类型 BIO 相比，它没有实现 gets 的方法。下面对一些同样的 BIO 操作函数作些简单说明：

BIO_read 和 BIO_write 对底层的文件描述符结构进行读写操作。这两个函数的一些行为取决于他们所在的平台的文件描述符的读写函数的行为，如果底层的文件描述符是非阻塞型的，那么他们基本上是跟我们前面介绍过得 BIO 的 IO 操作函数一样的。请参看前面的文章和资料。socket 是一类特殊的描述符，不应该使用文件描述符类型的 BIO 来封装它，而应该使用专门的 socke 类型 BIO，在以后我们会进行介绍。

BIO_puts 是支持的，但是 BIO_gets 在本类型描述符中是不支持的。

如果设置了关闭标志，那么当 BIO 被释放的时候底层的文件描述符就会被关闭。

BIO_reset 调用 lseek(fd,0,0)函数，使文件指针指向开始的位置。调用成功返回 0，失败返回－1。

BIO_seek 调用了 lseek(fd,ofs,0)函数，设置文件指针的位置到从文件头偏移 ofs 的位置，成功返回文件指针的位置，失败返回－1。

BIO_tell 返回目前文件指针的位置，它其实调用了 lseek(fd,0,1)函数，失败返回－1。

【BIO_set_fd】

该函数将 BIO 的底层文件描述符设置为 fd，关闭标志也同时做了设置，其含义与文件类型 BIO 相应的含义一样。返回 1。

【BIO_get_fd】返回相应 BIO 的底层文件描述符，存于参数 c，不过，同时也作为返回值返回。c 应该为 int *类型的指针。如果 BIO 没有初始化，调用该函数将失败，返回－1。

【BIO_new_fd】

创建并返回一个底层描述符为 fd，关闭标志为 close_flag 的文件描述符类型的 BIO。其实，该函数依次调用了 BIO_s_fd、BIO_new 和 BIO_set_fd 完成了该功能。该函数如果调用失败返

回 NULL。

下面是一个简单的例子：

```
BIO *out;
out = BIO_new_fd(fileno(stdout), BIO_NOCLOSE);
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

openssl 之 BIO 系列之 13---Socket 类型 BIO

Socket 类型 BIO---根据 openssl doc\crypto\bio_s_socket.pod 翻译和自己的理解写成

Socket 类型的 BIO 也是一种 source/sink 型 BIO，封装了 Socket 的 IO 操作，它相关的一些函数定义如下（openssl\bio.h）：

```
BIO_METHOD * BIO_s_socket(void);
#define BIO_set_fd(b,fd,c) BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c) BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)
BIO *BIO_new_socket(int sock, int close_flag);
```

前面我们在介绍 fd 类型 BIO 的时候曾经说过，它的函数的实现文件跟 Soket 类型的 BIO 其实是放在一起的，都在文件 bss_socket.c 里面，从这些定义我们就可以知道，之所以这样做，是因为这两种类型的 BIO 实现的函数基本是相同的，并且具有很多的共性。

【BIO_s_socket】

该函数返回一个 Socket 类型的 BIO_METHOD 结构，BIO_METHOD 结构的定义如下：

```
static BIO_METHOD methods_sockp=
 {
 BIO_TYPE_SOCKET,
 "socket",
 sock_write,
 sock_read,
 sock_puts,
 NULL, /* sock_gets, */
 sock_ctrl,
 sock_new,
 sock_free,
 NULL,
 };
```

可以看到，它跟 fd 类型 BIO 在实现的动作上基本上是一样的。只不过是前缀名和类型字段的名称不一样。其实在象 Linux 这样的系统里，Socket 类型跟 fd 类型是一样，他们是可以通用的，但是，为什么要分开来实现呢，那是因为有些系统如 windows 系统，socket 跟文件描述符是不一样的，所以，为了平台的兼容性，openssl 就将这两类分开来了。

BIO_read 和 BIO_write 对底层的 Socket 结构进行读写操作。

BIO_puts 是支持的，但是 BIO_gets 在 Socket 类型 BIO 中是不支持的，大家如果看源代码就可以知道，虽然 BIO_gets 在 Socket 类型是不支持的，但是如果调用该函数，不会出现异常，只会返回-1 的出错信息。

如果设置了关闭标志，那么当 BIO 被释放的时候底层的 Socket 连接就会被关闭。

【BIO_set_fd】

该函数将 Socket 描述符 fd 设置为 BIO 的底层 IO 结构，同时可以设置关闭标志 c。该函数返回 1。

【BIO_get_fd】

该函数返回指定 BIO 的 Socket 描述符，如果 c 参数不是 NULL，那么就将该描述符存在参数 c 里面，当然，Socket 描述符同时也作为返回值，如果 BIO 没有初始化则调用失败，返回－1。

【BIO_new_socket】

该函数根据给定的参数返回一个 socket 类型的 BIO，成功返回该 BIO 指针，失败返回 NULL。其实，该函数依次调用了 BIO_s_socket，BIO_new 和 BIO_set_fd 实现它的功能。

openssl 之 BIO 系列之 14---源类型的 NULL 型 BIO

source/sink 型 BIO---根据 openssl doc\crypto\bio_s_null.pod 翻译和自己的理解写成

这是一个空的 source/sink 型 BIO，写到这个 BIO 的数据都被丢掉了，从这里执行读操作也总是返回 EOF。该 BIO 非常简单，其相关函数的定义如下(openssl\bio.h)：

```
BIO_METHOD * BIO_s_null(void);
```

其相关的源文件实现函数在 bss_null.c 里面。

【BIO_s_null】

该函数返回一个 NULL 型的 BIO_METHOD 结构，该结构定义如下：

```
static BIO_METHOD null_method=
 {
 BIO_TYPE_NULL,
 "NULL",
 null_write,
 null_read,
 null_puts,
 null_gets,
 null_ctrl,
 null_new,
 null_free,
 NULL,
 };
```

从结构上看，这个类型的 BIO 实现了不少的函数，但是，仔细看看源文件，就会发现所有这些函数都只是简单返回 0、1 或者输入数据的长度，而不作任何事情。熟悉 Linux 系统的技术人员可能知道，这跟 Linux 系统的/dev/null 设备的行为是一样的。

一般来说，在 openssl 里面，这种类型的 BIO 是置放在 BIO 链的末尾的，比如在应用程序中，如果你要将一些数据通过 filter 型的 BIO digest 进行摘要算法，但不需要把它送往任何地方，又因为一个 BIO 链要求以 source/sink 型 BIO 开始或结束，所以这时候就可以在 BIO 链的末尾添加一个 source/sink 型的 NUll 类型 BIO 来实现这个功能。

openssl 之 BIO 系列之 15---内存(mem)类型 BIO

mem 类型 BIO---根据 openssl doc\crypto\bio_s_mem.pod 翻译和自己的理解写成

内存（mem）类型 BIO 所定义的相关系列函数如下（openssl\bio.h）：

```
BIO_METHOD * BIO_s_mem(void);
BIO_set_mem_eof_return(BIO *b,int v)
long BIO_get_mem_data(BIO *b, char **pp)
BIO_set_mem_buf(BIO *b,BUF_MEM *bm,int c)
BIO_get_mem_ptr(BIO *b,BUF_MEM **pp)
BIO *BIO_new_mem_buf(void *buf, int len);
```

内存型 BIO 是 source/sink 型 BIO，它使用内存作为它的 I/O。写进该类型 BIO 的数据被存储在 BUF_MEM 结构中，该结构被定义为适合存储数据的一种结构，其结构定义如下：

```
typedef struct buf_mem_st
{
 int length;    /* current number of bytes */
 char *data;
 int max; /* size of buffer */
} BUF_MEM;
```

可见，该结构定义了内存数据长度，数据存储空间以及最大长度三个变量来表述一段内存存储数据。但值得注意的是，内存型 BIO 的内存是可以无限扩大的，也就是说，不过你往里面写多少数据，都能成功执行。

一般来说，任何写入内存型 BIO 的数据都能被读出，除非该内存型 BIO 是只读类型的，那么，这时候如果对只读的内存型 BIO 执行读操作，那么相关数据就会从该 BIO 删除掉（其实没有删掉，只是指针往后面移动，访问不了了，如果调用 BIO_reset 就可以再访问）。

【BIO_s_mem】

该函数返回一个内存型的 BIO_METHOD 结构，期定义如下：

```
static BIO_METHOD mem_method=
{
 BIO_TYPE_MEM,
 "memory buffer",
 mem_write,
 mem_read,
 mem_puts,
 mem_gets,
 mem_ctrl,
 mem_new,
 mem_free,
 NULL,
};
```

BIO_write 和 BIO_read 函数是支持的。对内存型 BIO 执行写操作总是成功的，因为内存型 BIO 的内存能够无限扩大。任何一个对可读写的内存型 BIO 的读操作都会在使用内部拷贝操作从 BIO 里面删除该段数据，这样一来，如果 BIO 里面有大量的数据，而读的却只是很小的一些片断，那么会导致操作非常慢。使用只读的内存型 BIO 避免了这个问题。在使用的时候，如果内存型 BIO 必须使用可读写的，那么可以加一个 Buffer 型 BIO 到 BIO 链中，这可以使操作速度更快。在以后的版本（该文档所述版本为 0.9.6a）,可能会优化速度操作的问题。

BIO_gets 和 BIO_puts 操作在该类型 BIO 是支持的。

如果设置了 BIO_CLOSE 标志，那么内存型 BIO 被释放的时候其底层的 BUF_MEM 型 BIO 也同时被释放。

BIO_reset 函数被调用时，如果该 BIO 是可读写的，那么该 BIO 所有数据都会被清空；如果该 BIO 是只读的，那么该操作只会简单将指针指向原始位置，里面的数据可以再读。该文档所述版本的（0.9.6a）的内存型 BIO 对读写模式的 BIO 没有提供一个可以将指针重置但不破坏原有数据的方法，在以后的版本可能会增加的。

BIO_eof 返回 true，表明只时候 BIO 里面没有可读数据。

BIO_ctrl_pending 返回目前 BIO 里面存储的数据的字节(byte)数。

【BIO_set_mem_eof_return】

该函数设置一个没有数据的内存型 BIO 的执行读动作的行为。如果参数 v 是 0，那么该空的内存型 BIO 就会返回 EOF，也就是说，这时候返回为 0，如果调用 BIO_should_retry 就会返回 false。如果参数 v 为非零，那么就会返回 v，并且同时会设置重试标志，也就是说此时调用 BIO_read_retry 会返回 true。为了跟正常的返回正值避免混淆，v 应该设置为负值，典型来说是－1。

【BIO_get_mem_data】

该函数是一个宏定义函数，它将参数 pp 的指针指向内存型 BIO 的数据开始处，返回所有有效的数据。

【BIO_set_mem_buf】

该函数将参数 bm 所代表的 BUF_MEM 结构作为该 BIO 的底层结构，并把关闭标志设置为参数 c，c 可以是 BIO_CLOSE 或 BIO_NOCLOSE，该函数也是一个宏定义。

【BIO_get_mem_ptr】

该函数也是一个宏定义函数，它将底层的 BUF_MEM 结构放在指针 pp 中。

【BIO_new_mem_buf】

该函数创建一个内存型 BIO，其数据为 buf 里面长度为 len 的数据（单位为 byte），如果参数 len 是－1，那么就默认 buf 是以 null 结束的，使用 strlen 解决长度。这时候 BIO 被设置为只读的，不能执行写操作。它用于数据需要存储在一块静态内存中并以 BIO 形式存在的时候。所需要的数据是直接从内存中读取的，而不是先要执行拷贝操作（读写方式的内存 BIO 就是要先拷贝），这也就要求这块内存是只读的，不能改变，一直维持到 BIO 被释放。

【例子】

1.创建一个内存型 BIO 并写入数据

```
BIO *mem = BIO_new(BIO_s_mem());
BIO_puts(mem, "Hello World\n");
```

2.创建一个只读的内存型 BIO

```
char data[] = "Hello World";
BIO *mem;
mem = BIO_new_mem_buf(data, -1);
```

3.把一个 BUF_MEM 结构从一个 BIO 中取出并释放该 BIO

```
BUF_MEM *bptr;
BIO_get_mem_ptr(mem, &bptr);
BIO_set_close(mem, BIO_NOCLOSE); /* BIO_free()  不释放 BUF_MEM 结构  */
BIO_free(mem);
```

openssl 之 BIO 系列之 16---BIO 对(pair)类型 BIO

BIO 对(pair)类型 BIO---根据 openssl doc\crypto\bio_s_bio.pod 翻译和自己的理解写成

前面我们已经介绍过 BIO 对的概念，其实更进一步，BIO 对也是作为一种 source/sink 类型的 BIO 来处理的，也就是说，BIO 里面还提供了一种专门的 BIO_METHO 方法来处理 BIO 对的各种操作。BIO 对类型的 BIO 各种相关的函数定义如下(openssl\bio.h)：

```
BIO_METHOD *BIO_s_bio(void);
#define BIO_make_bio_pair(b1,b2) (int)BIO_ctrl(b1,BIO_C_MAKE_BIO_PAIR,0,b2)
#define BIO_destroy_bio_pair(b) (int)BIO_ctrl(b,BIO_C_DESTROY_BIO_PAIR,0,NULL)
#define BIO_shutdown_wr(b) (int)BIO_ctrl(b, BIO_C_SHUTDOWN_WR, 0, NULL)
#define BIO_set_write_buf_size(b,size) (int)BIO_ctrl(b,BIO_C_SET_WRITE_BUF_SIZE,size,NULL)
#define BIO_get_write_buf_size(b,size) (size_t)BIO_ctrl(b,BIO_C_GET_WRITE_BUF_SIZE,size,NULL)
int BIO_new_bio_pair(BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2);
#define BIO_get_write_guarantee(b) (int)BIO_ctrl(b,BIO_C_GET_WRITE_GUARANTEE,0,NULL)
size_t BIO_ctrl_get_write_guarantee(BIO *b);
#define BIO_get_read_request(b) (int)BIO_ctrl(b,BIO_C_GET_READ_REQUEST,0,NULL)
size_t BIO_ctrl_get_read_request(BIO *b);
int BIO_ctrl_reset_read_request(BIO *b);
```

可以看到，这些函数中大多数是宏定义函数并且都是基于 BIO_ctrl 函数的。

BIO 对类型的 BIO 是一对 source/sink 型的 BIO，数据通常是从一个 BIO 缓冲写入，从另一个 BIO 读出。其实，从源代码（bss_bio.c）可以看出，所谓的 BIO 对只是将两个 BIO 的终端输出（BIO 结构中参数 peer 的 ptr 成员）相互设置为对方，从而形成一种对称的结构，如下：

```
bio1->peer->ptr=bio2
bio2->peer->ptr=bio1
数据流向 1(写 bio1,读 bio2)：--->bio1--->bio2--->
数据流行 2(写 bio2,读 bio1)：--->bio2--->bio1--->
```

因为没有提供内部数据结构的内存锁结构(lock)，所以，一般来说这个 BIO 对的两个 BIO 都必须在一个线程下使用。因为 BIO 链通常是以一个 source/sink BIO 结束的，所以就可以实现应用程序通过控制 BIO 对的一个 BIO 从而控制整个 BIO 链的数据处理。其实，也就相当于 BIO 对给应用程序提供了一个处理整个 BIO 链的入口。上次我们说 BIO 对的时候就说过，BIO 对的一个典型应用就是在应用程序里面控制 TLS/SSL 的 I/O 接口，一般来说，在应用程序想在 TLS/SSL 中使用非标准的传输方法或者不适合使用标准的 socket 方法的时候就可以采用这样的方法来实现。

前面提过，BIO 对释放的时候，需要分别释放两个 BIO，如果在使用 BIO_free 或者 BIO_free_all 释放了其中一个 BIO 的时候，另一个 BIO 就也必须要释放。

当 BIO 对使用在双向应用程序的时候，如 TLS/SSL，一定要对写缓冲区里面的数据执行 flush

操作。当然，也可以通过在 BIO 对中的另一个 BIO 调用 BIO_pending 函数，如果有数据在缓冲区中，那么就将它们读出并发送到底层的传输通道中区。为了使请求或 BIO_should_read 函数调用成功（为 true），在执行任何正常的操作（如 select）之前，都必须这样做才行。

下面举一个例子说明执行 flush 操作的重要性：

考虑在 TLS/SSL 握手过程中，采用了 BIO_write 函数发送了数据，相应的操作应该使 BIO_read。BIO_write 操作成功执行并将数据写入到写缓冲区中。BIO_read 调用开始会失败，BIO_should_retry 返回 true。如果此时对写缓冲区不执行 flush 操作，那么 BIO_read 调用永远不会成功，因为底层传输通道会一直等待直到数据有效（但数据却在写缓冲区里，没有传到底层通道）。

【BIO_s_bio】

该函数返回一个 BIO 对类型的 BIO_METHOD，其定义如下：

```
static BIO_METHOD methods_biop =
{ BIO_TYPE_BIO,
  "BIO pair",
  bio_write,
  bio_read,
  bio_puts,
  NULL /*  没有定义  bio_gets */,
  bio_ctrl,
  bio_new,
  bio_free,
  NULL /*  没有定义  bio_callback_ctrl */
};
```

从定义中可以看到，该类型的 BIO 不支持 BIO_gets 的功能。

BIO_read 函数从缓冲 BIO 中读取数据，如果没有数据，则发出一个重试请求。

BIO_write 函数往缓冲 BIO 中写入数据，如果缓冲区已满，则发出一个重试请求。

BIO_ctrl_pending 和 BIO_ctrl_wpending 函数可以用来查看在读或写缓冲区里面有效的数据的数量。

BIO_reset 函数将写缓冲区里面的数据清除。

【BIO_make_bio_pair】

该函数将两个单独的 BIO 连接起来成为一个 BIO 对。

【BIO_destroy_pair】

该函数跟上面的函数相反，它将两个连接起来的 BIO 对拆开；如果一个 BIO 对中的任何一个 BIO 被释放，该操作会自动执行。

【BIO_shutdown_wr】

该函数关闭 BIO 对的其中一个 BIO，一个 BIO 被关闭后，针对该 BIO 的任何写操作都会返回错误。从另一个 BIO 读数据的时候要么返回剩余的有效数据，要么返回 EOF。

【BIO_set_write_buf_size】

该函数设置 BIO 的缓冲区大小。如果该 BIO 的缓存区大小没有初始化，那么就会使用默认的值，大小为 17k，这对于一个 TLS 记录来说是足够大的了。

【BIO_get_write_buf_size】

该函数返回写缓冲区的大小。

【BIO_new_bio_pair】

该函数我们在前面的《BIO 系列之 9---BIO 对的创建和应用》中已经做了详细的介绍，其实，它是调用了 BIO_new,BIO_make_bio_pair 和 BIO_set_write_buf_size 函数来创建一对 BIO 对的。如果两个缓冲区长度的参数都为零，那么就会使用默认的缓冲区长度。

【BIO_get_write_guarantee 和 BIO_ctrl_get_write_guarantee】

这两个函数返回当前能够写入 BIO 的数据的最大长度。如果往 BIO 写入的数据长度比该函数返回的数据长度大，那么 BIO_write 返回的写入数据长度会小于要求写入的数据，如果缓冲

区已经满了，则会发出一个重试的请求。这两个函数的唯一不同之处是一个使用函数实现的，一个是使用宏定义实现的。

【BIO_get_read_request 和 BIO_ctrl_get_read_request】

这两个函数返回要求发送的数据的长度，这通常是在对该 BIO 对的另一个 BIO 执行读操作时因为缓冲区数据为空导致失败时发出的请求。所以，这通常用来表明现在应该写入多少数据才能使接下来的读操作能够成功执行，这在 TLS/SSL 应用程序中是非常有用的，因为对于这个协议来说，读取的数据长度比缓冲区的数据长度通常要有意义的多。如果在读操作成功之后调用这两个函数会返回 0，如果在调用该函数之前有新的数据写入（不管是部分还是全部满足需要读取的数据的要求），那么调用该函数也会返回 0。理所当然，该函数返回的数据长度肯定不会大于 BIO_get_write_guarantee 函数返回的数据长度。

【BIO_ctrl_reset_read_request】

该函数就是把 BIO_get_read_request 要返回值设置为 0。

openssl 之 BIO 系列之 17---连接（connect）类型 BIO

连接(connect)类型 BIO---根据 openssl doc\crypto\bio_s_connect.pod 翻译和自己的理解写成

该类型的 BIO 封装了 socket 的 Connect 方法，它使得编程的时候可以使用统一的 BIO 规则进行 socket 的 connect 连接的操作和数据的发送接受，而不用关心具体平台的 Socket 的 connect 方法的区别。其相关定义的一些函数如下(openssl\bio.h)：

```
BIO_METHOD * BIO_s_connect(void);
#define BIO_set_conn_hostname(b,name) BIO_ctrl(b,BIO_C_SET_CONNECT,0,(char *)name)
#define BIO_set_conn_port(b,port) BIO_ctrl(b,BIO_C_SET_CONNECT,1,(char *)port)
#define BIO_set_conn_ip(b,ip) BIO_ctrl(b,BIO_C_SET_CONNECT,2,(char *)ip)
#define BIO_set_conn_int_port(b,port) BIO_ctrl(b,BIO_C_SET_CONNECT,3,(char *)port)
#define BIO_get_conn_hostname(b) BIO_ptr_ctrl(b,BIO_C_GET_CONNECT,0)
#define BIO_get_conn_port(b) BIO_ptr_ctrl(b,BIO_C_GET_CONNECT,1)
#define BIO_get_conn_ip(b,ip) BIO_ptr_ctrl(b,BIO_C_SET_CONNECT,2)
#define BIO_get_conn_int_port(b,port) BIO_int_ctrl(b,BIO_C_SET_CONNECT,3,port)
#define BIO_set_nbio(b,n) BIO_ctrl(b,BIO_C_SET_NBIO,(n),NULL)
#define BIO_do_connect(b) BIO_do_handshake(b)
BIO *BIO_new_connect(char *str)
```

【BIO_s_connect】

该函数返回一个 connect 类型的 BIO_METHOD 结构，该结构定义如下：

```
static BIO_METHOD methods_connectp=
{
 BIO_TYPE_CONNECT,
 "socket connect",
 conn_write,
 conn_read,
 conn_puts,
 NULL, /* connect_gets, */
 conn_ctrl,
 conn_new,
 conn_free,
 conn_callback_ctrl,
};
```

事实上，为了维护一个 Socket 结构，openssl 里面还定义了一个 BIO_CONNECT 结构来维护底层 socket 的地址信息以及状态信息，不过，通过封装，我们一般是不用直接接触该结构的，在此也就不再多做介绍，感兴趣可以参看文件 bss_conn.c 里面的定义和函数。

BIO_read 和 BIO_write 的操作调用底层的连接的 IO 操作来完成。如果在服务器地址和端口设置正确，但连接没有建立的时候调用读写操作函数，那么会先进行连接的建立操作，然后再进行读写操作。

BIO_puts 操作是支持的，但是 BIO_gets 操作不支持，这在该类型 BIO 的 BIO_METHOD 结构定义中就可以看出来。

如果关闭标志设置了，那么在 BIO 被释放的时候，任何活动的连接和 socket 都会被关闭。

BIO_reset 方法被调用的时候，连接（connect）类型的 BIO 的任何活动连接都会被关闭，从而回到可以重新跟同样的主机建立连接的状态。

BIO_get_fd 函数返回连接类型的 BIO 的底层 socket，当参数 c 不是 NULL 的时候，就将该 socket 赋值给 c，当然，socket 也作为返回值。c 参数应该为 int*类型。如果 BIO 没有初始化，则返回－1。

【BIO_set_conn_hostname】

该函数使用字符串设置主机名，该主机名也可以为 IP 地址的形式，还可以包括端口号，如 hostname:port，hostname/any/other/path 和 hostname:port/any/other/path 也是可以的。返回 1。

【BIO_set_conn_port】

该函数设置主机的端口号。该端口号的形式可以为数字的形式，也可以为字符串类似"http"的形式。如果使用字符串形式，首先会使用 getservbyname 函数搜索其相关的端口，如果没有搜索到，那么就会使用一张缺省的名字端口解释表，目前该表列出的字符串有：http, telnet, socks, https, ssl, ftp, gopher 和 wais.返回 1。

需要注意的是：如果端口名已经作为主机名的一部分设置了，那么它就会覆盖 BIO_set_conn_port 函数设置的端口值。有的时候（如有些应用可能不希望用固定的端口连接）可能不方便，这时候可以通过检测输入主机名的字符串中的":"字符，报错或截取字符串来避免这种情况。

【BIO_set_conn_ip】

该函数使用二进制的模式设置 IP 地址。返回 1。

【BIO_set_conn_int_port】

该函数以整数形式设置主机端口号，参数应该为 int*的形式。返回 1。

【BIO_get_conn_hostname】

该函数返回连接类型 BIO 的主机名，如果 BIO 以及初始化，但是没有设置主机名，那么返回 NULL。返回值因为是一个内部指针，所有不能更改它的值。

【BIO_get_conn_port】

该函数返回字符串类型的端口信息。如果没有设置，就返回 NULL。

【BIO_get_conn_ip】

该函数返回二进制形式的 IP 地址。如果没有设置，返回为全 0。

【BIO_get_conn_int_port】

该函数返回整数形式的端口号，如果没有设置，则返回 0。

上述四个函数的返回值在连接操作完成之后会被更新。而在此之前，返回值都是应用程序自己设置的。

【BIO_set_nbio】

设置 I/O 的非阻塞标志。如果参数 n 为 0，则 I/O 设置为阻塞模式；如果 n 为 1，则 I/O 设置为非阻塞模式。缺省的模式是阻塞模式。应该在连接建立之前调用本函数，因为非阻塞模式的 I/O 是在连接过程中设置的。返回值恒为 1。

注意：

如果是阻塞模式的 I/O，执行 IO 操作时（如读写），如果返回负值，说明就产生了错误的情况，如果返回值是 0，一般来说表明连接已经关闭。

如果设置为非阻塞模式，那么发出重试的请求就是很正常的事情了。

【BIO_do_connect】

该函数进行给定 BIO 的连接操作，如果连接成功，返回 1，否则返回 0 或负值。在非阻塞模式的时候，如果调用失败了，可以调用 BIO_should_retry 函数以决定是否需要重试。

一般来说，应用程序不需要调用本函数，只有在希望将连接过程跟其它 IO 处理过程独立开来的时候，才需要调用本函数。

在初始化连接的过程的时候，如果返回值失败的原因为 BIO_RR_CONNECT，调用 BIO_should_io_special 返回值可能也为 true。如果出现这种情况，说明连接过程被阻塞住了，应用程序应该使用正常的方法进行处理，直到底层的 socket 连接上了再重试。

【BIO_new_connect】

该函数创建并返回一个连接类型的 BIO，其实，它调用了 BIO_s_connect、BIO_new 已经 BIO_set_conn_hostname 函数完成了整个操作。成功则返回一个 BIO，否则返回 NULL。

【例子】

这是一个连接到本地 Web 服务器的例子，返回一页的信息并把该信息复制到标准输出设备。

```
BIO *cbio, *out;
int len;
char tmpbuf[1024];
ERR_load_crypto_strings();
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");
ERR_print_errors_fp(stderr);
/* whatever ... */
}
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);
```

openssl 之 BIO 系列之 18---接受(accept)类型 BIO

接受(accept)类型 BIO---根据 openssl doc\crypto\bio_s_accept.pod 翻译和自己的理解写成

接受（accept）类型的 BIO 跟连接（connect）类型 BIO 是相对应的，它封装了 Socket 的 accept 方法及其相关的一些操作，使得能够对不同的平台使用同一的函数进行操作。其定义的相关函数如下(openssl\bio.h)：

```
BIO_METHOD * BIO_s_accept(void);
#define BIO_set_accept_port(b,name) BIO_ctrl(b,BIO_C_SET_ACCEPT,0,(char *)name)
#define BIO_get_accept_port(b) BIO_ptr_ctrl(b,BIO_C_GET_ACCEPT,0)
BIO *BIO_new_accept(char *host_port);
#define BIO_set_nbio_accept(b,n) BIO_ctrl(b,BIO_C_SET_ACCEPT,1,(n)?"a":NULL)
#define BIO_set_accept_bios(b,bio) BIO_ctrl(b,BIO_C_SET_ACCEPT,2,(char *)bio)
#define BIO_set_bind_mode(b,mode) BIO_ctrl(b,BIO_C_SET_BIND_MODE,mode,NULL)
#define BIO_get_bind_mode(b,mode) BIO_ctrl(b,BIO_C_GET_BIND_MODE,0,NULL)
#define BIO_BIND_NORMAL 0
#define BIO_BIND_REUSEADDR_IF_UNUSED 1
#define BIO_BIND_REUSEADDR 2
#define BIO_do_accept(b) BIO_do_handshake(b)
```

【BIO_s_accept】

该函数返回一个接受（Accept）类型的 BIO_METHOD 结构，其定义如下：

```
static BIO_METHOD methods_acceptp=
 {
 BIO_TYPE_ACCEPT,
 "socket accept",
 acpt_write,
 acpt_read,
 acpt_puts,
```

```
NULL, /* connect_gets, */
acpt_ctrl,
acpt_new,
acpt_free,
NULL,
};
```

通过该结构我们可以一目了然的知道该方法支持什么 I/O 操作，在本类型中，BIO_gets 的操作是不支持的。跟连接（connect）类型 BIO 一样，openssl 也定义了一个维护接受 Socket 信息的结构，在此我也不再详细介绍该结构，大家参考 bss_acpt.c 文件。

本类型的 BIO 对各种平台的 TCP/IP 的 accept 做了封装，所以在使用的时候就可以同一的使用 BIO 的规则进行操作，而不用担心因为不同的平台带来程序改写或增加移植的工作量，这也是 BIO 很重要的一个目的。

BIO_read 和 BIO_write 函数操作调用了底层平台连接的 I/O 相关操作，如果这时候没有连接建立，端口设置正确，那么该 BIO 就会等待连接的建立。事实上，当一个连接建立的时候，一个新的 socket 类型 BIO 就会被创建并附加到 BIO 链中，形成 accept->socket 的 BIO 结构，所以这时候对初始化了的接受 socket 进行 IO 操作就会导致它处于等待连接建立的状态。当一个接受类型的 BIO 在 BIO 链的末尾的时候，在处理 I/O 调用之前它会先等待一个连接的建立；如果不是在末尾，那么它简单的把 I/O 调用传到下一个 BIO。

如果接受（accept）类型 BIO 的关闭标志设置了，那么当 BIO 被释放的时候，该 BIO 链上任何活动的连接和 socket 都会被关闭。

BIO_get_fd 和 BIO_set_fd 可以用来取得和设置该连接的 socket 描述符，详细情况参看《BIO 系列之 12---描述符(fd)类型 BIO》。

【BIO_set_accept_port】

该函数使用字串名来设置接受端口。其形式为“host:port”，“host”是要使用的接口，“port”是端口。这两部分都可以为“*”，这时候表示可以使用任意接口和端口。这里的端口的字符串含义跟连接（connect）类型 BIO 里面定义的一样，可以为数字形式的，可以使用 getservbyname 函数去匹配得到，还可以使用字符串的表，请参看连接型 BIO 说明的相关部分。

【BIO_new_accept】

该函数将 BIO_new 和 BIO_set_accept_port 函数放在一个函数里面调用，创建一个新的接受（accept）类型的 BIO。

【BIO_set_nbio_accept】

该函数设置接受 socket 是否为阻塞模式（缺省），如果参数 n 为 0，为阻塞模式，n 为 1 则为非阻塞模式。

【BIO_set_accept_bios】

该函数用来设置一个 BIO 链，当接受到一个连接的时候，这个设置好的 BIO 链就会被复制或附加到整个 BIO 链上来。有的时候这中处理方法是非常好的，比如，如果每个连接都需要一个缓冲区或 SSL 类型的 BIO，这时候使用本函数就省了很多麻烦了。需要注意的是，在调用本函数后，这个设置的链上的 BIO 不能自己释放，在接受（accept）BIO 被释放后，它们会自动释放。

当该函数调用的时候，其设置的 BIO 链位于接受 BIO 和 socket 类型的 BIO 之间，即形成：accept->otherbios->socket 的新的 BIO 链。

【BIO_set_bind_mode 和 BIO_get_bind_mode】

这两个函数用来设置和取得目前的绑定模式。如果设置为 BIO_BIND_NORMAL（缺省），那么另外一个 socket 就不能绑定到同一个端口。如果设置为 BIO_BIND_REUSEADDR ，那么另外的 socket 可以绑定到同一个端口。如果设置为 BIO_BIND_REUSEADDR_IF_UNUSED，那么首先会尝试以 BIO_BIND_NORMAL 的模式绑定到端口，如果失败了而且端口没有使用，那么就会使用 BIO_BIND_REUSEADDR 模式绑定到端口。

【BIO_do_accept】

该函数有两个功能，当它在接受（accept）BIO 设置好之后第一被调用的时候，它会创建一

个接受 socket 并把它跟地址绑定；第二次被调用的时候，它会等待连接的到来。

【注意】

如果服务器端希望可以处理多个连接的情况（通常都是这样的），那么接受 BIO 必须能够用来处理以后的连接，这时候可以这样做：

等待到一个连接后，调用

connection=BIO_pop(accept)

这样，cnnection 会包含一个最近建立的连接的 BIO，accept 就再次成为一个独立的 BIO，可以用来处理其它连接了。如果没有其它连接建立，那么就可以使用 BIO_free 释放该 BIO。

当然，如果只有一个连接需要处理，也可以使用连接 BIO 本身来进行 I/O 操作。但是一般来说不推荐这样做，因为这时候该接受 BIO 依然处于接受其它连接建立的状态。这可以使用上述的方法解决。

【例子】

这个例子在 4444 端口接受了两个连接，发送信息到每个连接上然后释放他们。

```
BIO *abio, *cbio, *cbio2;
ERR_load_crypto_strings();
abio = BIO_new_accept("4444");
/* 首先调用 BIO_accept 启动接受 BIO */
if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error setting up accept\n");
ERR_print_errors_fp(stderr);
exit(0);
}
/* 等待连接建立*/
if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error accepting connection\n");
ERR_print_errors_fp(stderr);
exit(0);
}
fprintf(stderr, "Connection 1 established\n");
/* 返回连接的 BIO*/
cbio = BIO_pop(abio);
BIO_puts(cbio, "Connection 1: Sending out Data on initial connection\n");
fprintf(stderr, "Sent out data on connection 1\n");
/* 等待另一个连接的建立 */
if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error accepting connection\n");
ERR_print_errors_fp(stderr);
exit(0);
}
fprintf(stderr, "Connection 2 established\n");
/* 关闭连接 BIO，不再接受连接的建立 */
cbio2 = BIO_pop(abio);
BIO_free(abio);
BIO_puts(cbio2, "Connection 2: Sending out Data on second\n");
fprintf(stderr, "Sent out data on connection 2\n");
BIO_puts(cbio, "Connection 1: Second connection established\n");
/* 关闭这两个连接 */
BIO_free(cbio);
BIO_free(cbio2);
```

openssl 之 BIO 系列之 19---Filter 类型的 NULL 型 BIO

Fileter 类型的 NULL 型 BIO---根据 openssl doc\crypto\bio_f_null.pod 翻译和自己的理解写成

前面我们已经介绍完 source/sink 型的 BIO 了，以后的 BIO 系列文章将开始介绍过滤(filter)

类型的 BIO。那么首先介绍的是一个非常简单的 BIO 类型——NULL 型 filter BIO，其定义如下（openssl\bio.h）：

BIO_METHOD * BIO_f_null(void);

在本类型中，只有这个函数是定义了的，该函数返回一个 NULL 型的过滤 BIO_METHOD 结构，NULL 过滤型 BIO 是一个不作任何事情的 BIO。针对该类型 BIO 的任何调用都会被简单传递中 BIO 链中的下一个 BIO 中去，也就相当于该 BIO 是不存在的一样。所以，一般来说，该类型的 BIO 用处不大。

openssl 之 BIO 系列之 20---缓冲（buffer）类型 BIO

缓冲（buffer）类型 BIO---根据 openssl doc\crypto\bio_f_buffer.pod 翻译和自己的理解写成

缓冲（buffer）类型 BIO 是一种过滤（filter）型的 BIO，其相关的一些函数定义如下（openssl\bio.h）：

```
BIO_METHOD * BIO_f_buffer(void);
#define BIO_get_buffer_num_lines(b) BIO_ctrl(b,BIO_C_GET_BUFF_NUM_LINES,0,NULL)
#define BIO_set_read_buffer_size(b,size) BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,0)
#define BIO_set_write_buffer_size(b,size) BIO_int_ctrl(b,BIO_C_SET_BUFF_SIZE,size,1)
#define BIO_set_buffer_size(b,size) BIO_ctrl(b,BIO_C_SET_BUFF_SIZE,size,NULL)
#define BIO_set_buffer_read_data(b,buf,num) BIO_ctrl(b,BIO_C_SET_BUFF_READ_DATA,num,buf)
```

【BIO_f_buffer】

该函数返回一个 Buffer 类型的 BIO_METHOD 结构，该结构定义如下（bf_buff.c）：

```
static BIO_METHOD methods_buffer=
{
BIO_TYPE_BUFFER,
"buffer",
buffer_write,
buffer_read,
buffer_puts,
buffer_gets,
buffer_ctrl,
buffer_new,
buffer_free,
buffer_callback_ctrl,
};
```

由结构定义可见，该类型 BIO 支持所有 BIO 的 I/O 函数。写入缓冲(buffer)BIO 的数据存储在缓冲区里面，定期写入到 BIO 链的下一个 BIO 中，事实上，只有缓冲区已满或者调用了 BIO_flush 函数时，数据才会写入下面的 BIO，所以，当任何存储在缓冲区的数据需要写入的时候（如在使用 BIO_pop 函数从 BIO 链中删除一个 buffer 类型 BIO 之前），必须使用 BIO_flush 函数，如果 BIO 链的末尾是一个非阻塞型的 BIO，有时候调用 BIO_flush 可能出现失败，需要重试的情况。从该类型 BIO 读取数据时，数据从下一个 BIO 填充到该 BIO 的内部缓冲区中，然后再读出来。该类型 BIO 支持 BIO_gets 和 BIO_puts 方法，事实上，BIO_gets 函数是通过在下一个 BIO 的 BIO_read 函数来实现的，所以，如果一个 BIO 不支持 BIO_gets 方法（如 SSL 类型的 BIO），可以通过预先附加一个 buffer 类型 BIO 来实现 BIO_gets 的功能。

BIO_reset 被调用的时候，该类型 BIO 里面的所有数据都会被清空。

【BIO_get_buffer_num_lines】

返回缓冲区中目前数据的的行数。

【 BIO_set_read_buffer_size、BIO_set_write_buffer_size 和 BIO_set_buffer_size】

这三个函数分别设置缓冲类型 BIO 的读、写或者读写缓冲区的大小。初始的缓冲区大小由宏定义 DEFAULT_BUFFER_SIZE 决定，默认的是 1024。如果设置的缓冲区大小小于 DEFAULT_BUFFER_SIZE，那么就会被忽略，也就是说缓冲区大小会保持为 DEFAULT_BUFFER_SIZE 所定义的大小。当重新设置缓冲区大小时，里面的数据会全部被清

空。成功执行返回 1，否则返回 0。

【BIO_set_buffer_read_data】

该函数清空缓冲区原有的数据，并使用 num 个 buf 中的数据填充该缓冲区，如果 num 的大小大于目前的缓冲区设定大小，那么缓冲区就会自动扩大。成功设置返回 1，否则返回 0。

openssl 之 BIO 系列之 21---Base64 类型的 BIO

Base64 类型 BIO---根据 openssl doc\crypto\bio_f_base64.pod 翻译和自己的理解写成

该类型为过滤（filter）类型 BIO，其定义如下（openssl\bio.h,openssl\evp.h）：

```
BIO_METHOD * BIO_f_base64(void);
```

【BIO_f_base64】

该函数返回一个 Base64 类型的 BIO_METHOD 结构，该结构定义如下（evp\bio_b64.c）：

```
static BIO_METHOD methods_b64=
{
 BIO_TYPE_BASE64,
 "base64 encoding",
 b64_write,
 b64_read,
 NULL, /* b64_puts, */
 NULL, /* b64_gets, */
 b64_ctrl,
 b64_new,
 b64_free,
 b64_callback_ctrl,
};
```

应该注意的是，该类型的 BIO 其定义并不在 bio 目录下，而是在 evp 目录下。

当往该 BIO 写入数据时，数据被 Base64 编码，当从该 BIO 读数据时，数据被 Base64 解码。该 BIO 不支持 BIO_gets 和 BIO_puts 的功能。

BIO_flush 在该类型 BIO 被调用的时候，表示需要写入的数据已经写完，用来把最后的一段数据写入到 BIO 里面去。

【BIO_set_flags】

该函数可以用来设置标记 BIO_FLAGS_BASE64_NO_NL，该标记设置后，将把所有数据编码成为一行或者说期望所有数据都在一行上。需要注意的是，由于 base64 编码本身格式的原因，不能准确可靠的决定编码后的数据块的结束位置，大家使用的时候自己需要注意数据的长度问题。

【例子】

下面的程序将字符串"Hello World\n"进行 base64 编码并写入到标准输出设备。

```
 BIO *bio, *b64;
 char message[] = "Hello World \n";

 b64 = BIO_new(BIO_f_base64());
 bio = BIO_new_fp(stdout, BIO_NOCLOSE);
 bio = BIO_push(b64, bio);
 BIO_write(bio, message, strlen(message));
 BIO_flush(bio);
 BIO_free_all(bio);
```

下面的程序将 base64 编码的数据从标准输入设备读出并将解码数据输出到标准输出设备：

```
 BIO *bio, *b64, bio_out;
 char inbuf[512];
 int inlen;
 char message[] = "Hello World \n";

 b64 = BIO_new(BIO_f_base64());
 bio = BIO_new_fp(stdin, BIO_NOCLOSE);
```

```
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
bio = BIO_push(b64, bio);
while((inlen = BIO_read(bio, inbuf, strlen(message))) > 0)
BIO_write(bio_out, inbuf, inlen);
BIO_free_all(bio);
```

openssl 之 BIO 系列之 22---Cipher 类型的 BIO

Cipher 类型 BIO---根据 openssl doc\crypto\bio_f_cipher.pod 翻译和自己的理解写成

该类型为过滤（filter）类型 BIO，其定义如下（openssl\bio.h,openssl\evp.h）：

```
BIO_METHOD * BIO_f_cipher(void);
void BIO_set_cipher(BIO *b,const EVP_CIPHER *cipher,
unsigned char *key, unsigned char *iv, int enc);
int BIO_get_cipher_status(BIO *b)
int BIO_get_cipher_ctx(BIO *b, EVP_CIPHER_CTX **pctx)
```

【BIO_f_cipher】

该函数返回 cipher 类型的 BIO_METHOD 结构，其结构定义如下（evp\bio_enc.c）：

```
static BIO_METHOD methods_enc=
{
BIO_TYPE_CIPHER,"cipher",
enc_write,
enc_read,
NULL, /* enc_puts, */
NULL, /* enc_gets, */
enc_ctrl,
enc_new,
enc_free,
enc_callback_ctrl,
};
```

该类型的 BIO 将写入该 BIO 的数据加密，从该 BIO 读数据时数据被解密，它事实上封装了 EVP_CipherInit、EVP_CipherUpdate、EVP_CipherFinal 三种方法。它不支持 BIO_puts 和 BIO_gets 的方法，如果要使用这两个方法，可以通过在前面附加一个 buffer 类型的 BIO 来实现，这在前面我们介绍过。

跟 base64 型 BIO 相似，当调用 BIO_flush 函数时，表明所有数据都已经通过该类型 BIO 加密了，用来将最后的一段数据通过该 BIO 进行加密。在进行加密的时候，必须调用 BIO_flush 函数来把最后的数据通过 BIO 进行加密，否则最后的数据会在解密的时候出现失败的情况。当从一个加密类型的 BIO 读取数据时，当读到最后一段数据时，会通过检测 EOF 自动检测到数据结束标志并自动将这段数据解密。

【BIO_set_cipher】

该函数设置该 BIO 的加密算法，数据使用参数 key 为加密密钥，参数 iv 作为加密的 IV（初始化向量）。如果 enc 设置为 1，则为加密，enc 设置为 0，则为解密。该函数不返回值。

【BIO_get_cipher_status】

该函数是一个 BIO_ctrl 的宏，用来检测解密是否成功执行。因为在解密的时候（执行读操作的时候），如果最后一段数据发生错误，会返回 0，而遇到 EOF 成功完成操作后也会返回 0，所以必须调用本函数确定解密操作是否成功执行了。解密成功返回 1，否则返回 0。

【BIO_get_cipher_ctx】

该函数也是 BIO_ctrl 的一个宏定义函数，它返回 BIO 的内部加密体制。返回的加密体制可以使用标准的加密规则进行设置。这在 BIO_set_cipher 函数的灵活性不能适应应用程序的需要的时候是很有用的。该函数总是返回 1。

openssl 之 BIO 系列之 23---MD 类型的 BIO

MD 类型 BIO---根据 openssl doc\crypto\bio_f_md.pod 翻译和自己的理解写成

该类型为过滤（filter）类型 BIO，其定义如下（openssl\bio.h,openssl\evp.h）：

```
BIO_METHOD * BIO_f_md(void);
```

```
int BIO_set_md(BIO *b,EVP_MD *md);
int BIO_get_md(BIO *b,EVP_MD **mdp);
int BIO_get_md_ctx(BIO *b,EVP_MD_CTX **mdcp);
```

跟 Cipher 类型一样，该类型的一些定义和实现文件是在 evp\bio_md.c 里面，而不是在 bio 目录下。大家要看源文件，请参看这个文件。

【BIO_f_md】

该函数返回一个 MD 类型的 BIO_METHOD 结构，其定义如下：

```
static BIO_METHOD methods_md=
{
BIO_TYPE_MD,"message digest",
md_write,
md_read,
NULL, /* md_puts, */
md_gets,
md_ctrl,
md_new,
md_free,
md_callback_ctrl,
};
```

MD 类型 BIO 对通过它的任何数据都进行摘要操作（digest)，事实上，该类型 BIO 封装了 EVP_DigestInit、EVP_DigestUpdate 和 EVP_DigestFinal 三个函数的功能和行为。该类型 BIO 是完全对称的，也就是说，不管是读数据（BIO_read)还是写数据（BIO_write），都进行相同的摘要操作。

BIO_gets 函数执行的时候，如果给定的 size 参数足够大，可以完成摘要（digest）计算，那么就会返回摘要值。BIO_puts 函数是不支持的，如果需要支持该函数，可以在前面附加一个 buffer 类型的 BIO。

BIO_reset 函数重新初始化一个摘要类型的 BIO，事实上，它是简单重新调用了 EVP_DigestInit 函数进行初始化。

注意，在从一个摘要 BIO 里面读取完摘要信息之后，在重新使用该 BIO 之前，必须调用 BIO_reset 或 BIO_set_md 重新初始化该 BIO 才行。

【BIO_set_md】

该函数是一个 BIO_ctrl 函数的宏定义函数，它使用参数 md 设置给定 BIO 的摘要算法。该函数必须在执行读写操作之前调用，用来初始化一个摘要类型的 BIO。调用成功返回 1，否则返回 0。

【BIO_get_md】

该函数也是 BIO_ctrl 函数一个宏定义。它返回 BIO 摘要方法的指针到 mdp 参数里面。调用成功返回 1，否则返回 0。

【BIO_get_md_ctx】

该函数返回摘要 BIO 的方法结构到 mdcp 参数里面。该结构可以作为参数使用在 EVP_DigestFinal、EVP_SignFinal 和 EVP_VerifyFinal 函数里，这增加了灵活性。因为该函数返回的结构是一个 BIO 内部的结构，所以对该结构的任何改变操作都会影响到相应的 BIO，并且如果该 BIO 释放了，该结构指针也就无效了。调用成功返回 1，否则返回 0。

【例子】

1.下列的例子创建一个包含 SHA1 和 MD5 类型摘要 BIO 的 BIO 链，并将数据"Hello World"通过它们进行摘要操作。

```
BIO *bio, *mdtmp;
char message[] = "Hello World";
bio = BIO_new(BIO_s_null());
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
// 使用 BIO_push 在 BIO 链前面增加一个 sink 类型的 BIO，作为 BIO 链开始的标志
```

```
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
/* 注意，现在 mdtmp 变量已经没有用了*/
BIO_write(bio, message, strlen(message));//因为最后一个 BIO 是 null 型的 BIO，所以数据实际上已经自动被丢弃了。
```

2.下面的例子演示了从摘要类型 BIO 读数据的过程：

```
BIO *bio, *mdtmp;
char buf[1024];
int rdlen;
bio = BIO_new_file(file, "rb");
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
do {
rdlen = BIO_read(bio, buf, sizeof(buf));
/* 可以在这里面加入处理数据的代码 */
} while(rdlen > 0);
```

3.下面的例子从一个 BIO 链中读取摘要数据并输出。可以跟上面的例子一起使用。

```
BIO *mdtmp;
unsigned char mdbuf[EVP_MAX_MD_SIZE];
int mdlen;
int i;
mdtmp = bio; /* 这里假设 BIO 已经设置好了*/
do {
EVP_MD *md;
mdtmp = BIO_find_type(mdtmp, BIO_TYPE_MD);
if(!mdtmp) break;
BIO_get_md(mdtmp, &md);
printf("%s digest", OBJ_nid2sn(EVP_MD_type(md)));
mdlen = BIO_gets(mdtmp, mdbuf, EVP_MAX_MD_SIZE);
for(i = 0; i < mdlen; i++) printf(":%02X", mdbuf[i]);
printf("\n");
mdtmp = BIO_next(mdtmp);
} while(mdtmp);
BIO_free_all(bio);
```

openssl 之 BIO 系列之 24---SSL 类型的 BIO

SSL 类型的 BIO---根据 openssl doc\crypto\bio_f_ssl.pod 翻译和自己的理解写成

从名字就可以看出，这是一个非常重要的 BIO 类型，它封装了 openssl 里面的 ssl 规则和函数，相当于提供了一个使用 SSL 很好的有效工具，一个很好的助手。其定义（openssl\bio.h,openssl\ssl.h）如下：

```
BIO_METHOD *BIO_f_ssl(void);
#define BIO_set_ssl(b,ssl,c) BIO_ctrl(b,BIO_C_SET_SSL,c,(char *)ssl)
#define BIO_get_ssl(b,sslp) BIO_ctrl(b,BIO_C_GET_SSL,0,(char *)sslp)
#define BIO_set_ssl_mode(b,client) BIO_ctrl(b,BIO_C_SSL_MODE,client,NULL)
#define BIO_set_ssl_renegotiate_bytes(b,num) BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_BYTES,num,NULL);
#define BIO_set_ssl_renegotiate_timeout(b,seconds) BIO_ctrl(b,BIO_C_SET_SSL_RENEGOTIATE_TIMEOUT,seconds,NULL);
```

```
#define                                    BIO_get_num_renegotiates(b)
BIO_ctrl(b,BIO_C_SET_SSL_NUM_RENEGOTIATES,0,NULL);
BIO *BIO_new_ssl(SSL_CTX *ctx,int client);
BIO *BIO_new_ssl_connect(SSL_CTX *ctx);
BIO *BIO_new_buffer_ssl_connect(SSL_CTX *ctx);
int BIO_ssl_copy_session_id(BIO *to,BIO *from);
void BIO_ssl_shutdown(BIO *bio);
#define BIO_do_handshake(b) BIO_ctrl(b,BIO_C_DO_STATE_MACHINE,0,NULL)
```

该类型 BIO 的实现文件在 ssl\bio_ssl.c 里面，大家可以参看这个文件得到详细的函数实现信息。

【BIO_f_ssl】

该函数返回一个 SSL 类型的 BIO_METHOD 结构，其定义如下：

```
static BIO_METHOD methods_sslp=
{
BIO_TYPE_SSL,"ssl",
ssl_write,
ssl_read,
ssl_puts,
NULL, /* ssl_gets, */
ssl_ctrl,
ssl_new,
ssl_free,
ssl_callback_ctrl,
};
```

可见，SSL 类型 BIO 不支持 BIO_gets 的功能。

BIO_read 和 BIO_write 函数调用的时候，SSL 类型的 BIO 会使用 SSL 协议进行底层的 I/O 操作。如果此时 SSL 连接并没有建立，那么就会在调用第一个 IO 函数的时候先进行连接的建立。

如果使用 BIO_push 将一个 BIO 附加到一个 SSL 类型的 BIO 上，那么 SSL 类型的 BIO 读写数据的时候，它会被自动调用。

BIO_reset 调用的时候，会调用 SSL_shutdown 函数关闭目前所有处于连接状态的 SSL，然后再对下一个 BIO 调用 BIO_reset，这功能一般就是将底层的传输连接断开。调用完成之后，SSL 类型的 BIO 就处于初始的接受或连接状态。

如果设置了 BIO 关闭标志，那么 SSL 类型 BIO 释放的时候，内部的 SSL 结构也会被 SSL_free 函数释放。

【BIO_set_ssl】

该函数设置 SSL 类型 BIO 的内部 ssl 指针指向 ssl，同时使用参数 c 设置了关闭标志。

【BIO_get_ssl】

该函数返回 SSL 类型 BIO 的内部的 SSL 结构指针，得到该指针后，可以使用标志的 SSL 函数对它进行操作。

【BIO_set_ssl_mode】

该函数设置 SSL 的工作模式，如果参数 client 是 1，那么 SSL 工作模式为客户端模式，如果 client 为 0，那么 SSL 工作模式为服务器模式。

【BIO_set_ssl_renegotiate_bytes】

该函数设置需要重新进行 session 协商的读写数据的长度为 num。当设置完成后，在没读写的数据一共到达 num 字节后，SSL 连接就会自动重新进行 session 协商，这可以加强 SSL 连接的安全性。参数 num 最少为 512 字节。

【BIO_set_ssl_renegotiate_timeout】

该函数跟上述函数一样都是为了加强 SSL 连接的安全性的。不同的是，该函数采用的参数是时间。该函数设置重新进行 session 协商的时间，其单位是秒。当 SSL session 连接建立的时间到达其设置的时间时，连接就会自动重新进行 session 协商。

【BIO_get_num_renegotiates】

该函数返回 SSL 连接在因为字节限制或时间限制导致 session 重新协商之前总共读写的数据长度。

【BIO_new_ssl】

该函数使用 ctx 参数所代表的 SSL_CTX 结构创建一个 SSL 类型的 BIO，如果参数 client 为非零值，就使用客户端模式。

【BIO_new_ssl_connect】

该函数创建一个包含 SSL 类型 BIO 的新 BIO 链，并在后面附加了一个连接类型的 BIO。

方便而且有趣的是，因为在 filter 类型的 BIO 里，如果是该 BIO 不知道（没有实现）BIO_ctrl 操作，它会自动把该操作传到下一个 BIO 进行调用，所以我们可以在调用本函数得到 BIO 上直接调用 BIO_set_host 函数来设置服务器名字和端口，而不需要先找到连接 BIO。

【BIO_new_buffer_ssl_connect】

创建一个包含 buffer 型的 BIO，一个 SSL 类型的 BIO 以及一个连接类型的 BIO。

【BIO_ssl_copy_session_id】

该函数将 BIO 链 from 的 SSL Session ID 拷贝到 BIO 链 to 中。事实上，它是通过查找到两个 BIO 链中的 SSL 类型 BIO，然后调用 SSL_copy_session_id 来完成操作的。

【BIO_ssl_shutdown】

该函数关闭一个 BIO 链中的 SSL 连接。事实上，该函数通过查找到该 BIO 链中的 SSL 类型 BIO，然后调用 SSL_shutdown 函数关闭其内部的 SSL 指针。

【BIO_do_handshake】

该函数在相关的 BIO 上启动 SSL 握手过程并建立 SSL 连接。连接成功建立返回 1，否则返回 0 或负值，如果连接 BIO 是非阻塞型的 BIO，此时可以调用 BIO_should_retry 函数以决定释放需要重试。如果调用该函数的时候 SSL 连接已经建立了，那么该函数不会做任何事情。一般情况下，应用程序不需要直接调用本函数，除非你希望将握手过程跟其它 IO 操作分离开来。

需要注意的是，如果底层是阻塞型（openssl 帮助文档写的是非阻塞型,non blocking,但是根据上下文意思已经 BIO 的其它性质，我个人认为是阻塞型，blocking 才是正确的）的 BIO，在一些意外的情况 SSL 类型 BIO 下也会发出意外的重试请求，如在执行 BIO_read 操作的时候如果启动了 session 重新协商的过程就会发生这种情况。在 0.9.6 和以后的版本，可以通过 SSL 的标志 SSL_AUTO_RETRY 将该类行为禁止，这样设置之后，使用阻塞型传输的 SSL 类型 BIO 就永远不会发出重试的请求。

【例子】

1.一个 SSL/TLS 客户端的例子，完成从一个 SSL/TLS 服务器返回一个页面的功能。其中 IO 操作的方法跟连接类型 BIO 里面的例子是相同的。

```
BIO *sbio, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;
ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();
//如果系统平台不支持自动进行随机数种子的设置，这里应该进行设置(seed PRNG)
ctx = SSL_CTX_new(SSLv23_client_method());
//通常应该在这里设置一些验证路径和模式等，因为这里没有设置，所以该例子可以跟使用任意 CA 签发证书的任意服务器建立连接
sbio = BIO_new_ssl_connect(ctx);
BIO_get_ssl(sbio, &ssl);
if(!ssl) {
fprintf(stderr, "Can't locate SSL pointer\n");
}
```

```
/* 不需要任何重试请求*/
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
//这里你可以添加对 SSL 的其它一些设置
BIO_set_conn_hostname(sbio, "localhost:https");
out = BIO_new_fp(stdout, BIO_NOCLOSE);
if(BIO_do_connect(sbio) <= 0) {
fprintf(stderr, "Error connecting to server\n");
ERR_print_errors_fp(stderr);
}
if(BIO_do_handshake(sbio) <= 0) {
fprintf(stderr, "Error establishing SSL connection\n");
ERR_print_errors_fp(stderr);
}
/* 这里可以添加检测 SSL 连接的代码，得到一些连接信息*/
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);
```

2.一个简单的服务器的例子。它使用了 buffer 类型的 BIO，从而可以使用 BIO_gets 从一个 SSL 类型的 BIO 读取数据。它创建了一个包含客户端请求的随机 web 页，并把请求信息输出到标准输出设备。

```
BIO *sbio, *bbio, *acpt, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;
ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();
//可能需要进行随机数的种子处理（seed PRNG）
ctx = SSL_CTX_new(SSLv23_server_method());
if (!SSL_CTX_use_certificate_file(ctx,"server.pem",SSL_FILETYPE_PEM)
|| !SSL_CTX_use_PrivateKey_file(ctx,"server.pem",SSL_FILETYPE_PEM)
|| !SSL_CTX_check_private_key(ctx)) {
fprintf(stderr, "Error setting up SSL_CTX\n");
ERR_print_errors_fp(stderr);
return 0;
}
//可以在这里设置验证路径，DH 和 DSA 算法的临时密钥回调函数等等
/* 创建一个新的服务器模式的 SSL 类型 BIO*/
sbio=BIO_new_ssl(ctx,0);
BIO_get_ssl(sbio, &ssl);
if(!ssl) {
fprintf(stderr, "Can't locate SSL pointer\n");
}
/* 不需要任何重试请求 */
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
/* 创建一个 Buffer 类型 BIO */
bbio = BIO_new(BIO_f_buffer());
```

```
/* 加到 BIO 链上*/
sbio = BIO_push(bbio, sbio);
acpt=BIO_new_accept("4433");
/*
```

当一个新连接建立的时候，我们可以将 sbio 链自动插入到连接所在的 BIO 链中去。这时候，这个 BIO 链(sbio)就被 accept 类型 BIO 吞并了，并且当 accept 类型 BIO 释放的时候，它会自动被释放。

```
*/
BIO_set_accept_bios(acpt,sbio);
out = BIO_new_fp(stdout, BIO_NOCLOSE);
/* 设置 accept BIO */
if(BIO_do_accept(acpt) <= 0) {
fprintf(stderr, "Error setting up accept BIO\n");
ERR_print_errors_fp(stderr);
return 0;
}
/* 等待连接的建立 */
if(BIO_do_accept(acpt) <= 0) {
fprintf(stderr, "Error in connection\n");
ERR_print_errors_fp(stderr);
return 0;
}
/*
因为我们只想处理一个连接，所以可以删除和释放 accept BIO 了
*/
sbio = BIO_pop(acpt);
BIO_free_all(acpt);
if(BIO_do_handshake(sbio) <= 0) {
fprintf(stderr, "Error in SSL handshake\n");
ERR_print_errors_fp(stderr);
return 0;
}
BIO_puts(sbio, "HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n");
BIO_puts(sbio, "<pre>\r\nConnection Established\r\nRequest headers:\r\n");
BIO_puts(sbio, "------------------------------------------------\r\n");
for(;;) {
len = BIO_gets(sbio, tmpbuf, 1024);
if(len <= 0) break;
BIO_write(sbio, tmpbuf, len);
BIO_write(out, tmpbuf, len);
/* 查找请求头的结束标准空白行*/
if((tmpbuf[0] == '\r') || (tmpbuf[0] == '\n')) break;
}
BIO_puts(sbio, "------------------------------------------------\r\n");
BIO_puts(sbio, "</pre>\r\n");
/* 因为使用了 buffer 类型的 BIO，我们最好调用 BIO_flush 函数 */
BIO_flush(sbio);
BIO_free_all(sbio);
```

openssl 之 BIO 系列之 25---结束语

（作者：DragonKing, Mail: wzhah@263.net ,发布于：http://gdwzh.126.com 之 openssl 专业论坛）

经过半个月左右，终于将 BIO 的结构和各个分支基本介绍完了，BIO 是一个很好的思想，具备了**基本的面向对象的思想**，也是跨平台实现的一个范例。

如果大家耐心看完了这个系列就可以发现，BIO 基本几乎封装了**除了证书处理外的 openssl**

所有的功能，包括加密库以及 SSL/TLS 协议。当然，它们都只是在 openssl 其它功能之上封装搭建起来的，但却方便了不少。对于一般的编程应用人员来说，从 BIO 开始使用 openssl 的 API 功能或者是一个不错的选择，因为通过封装，BIO 的 I/O 函数是有限的，掌握和使用起来相对简单容易。

因为时间和精力有限，很多东西我并没有进行测试，只是根据源代码和文档为基础写成的，肯定免不了有错误的地方，希望大家指正，一定要指正，否则就误人子弟了。

下面的要介绍的系列，将会是 EVP 或 SSL 方面的，如果有特别的要求，请提出来：）我以大家的意见为准，因为写这个东西，也是网站刚开的时候一些网友的建议。没有人看，我写了也没有用。

再次感谢大家的支持！

希望大家继续支持！

OpenSSL 中对称加密算法的统一接口

msn: yfydz_no1@hotmail.com
来源：http://yfydz.cublog.cn

1. 前言

OpenSSL 是一个开源的 SSL 实现，其中集成了多种加密算法。OpenSSL 将各算法的各自独特地方封装在内部，对外则使用了统一的算法接口，因此外部应用只需指定使用何种算法，就可以用相同的方法调用加解密函数而不用考虑其差异。这种算法统一封装的方式在其他很多软件中都采用，也给算法扩充提供了方便。

以下代码来自 OpenSSL-0.9.7b。

2. EVP 接口

2.1 数据结构

Openssl/crypto/evp 目录下定义各种算法的接口源文件，这些文件要作的事就是要填写描述算法的 EVP_CIPHER 结构，每个算法都有一个 **EVP_CIPHER 结构进行描述**：

Openssl/crypto/evp/evp.h

```
struct evp_cipher_st
 {
 int nid;
 int block_size;
 int key_len;    /* Default value for variable length ciphers */
 int iv_len;
 unsigned long flags; /* Various flags */
 int (*init)(EVP_CIPHER_CTX *ctx, const unsigned char *key,const unsigned char *iv, int enc);
/* init key */ /*初始化密钥* /
 int (*do_cipher)(EVP_CIPHER_CTX *ctx, unsigned char *out,const unsigned char *in, unsigned int inl);/* encrypt/decrypt data */ /*加密/解密数据* /
 int (*cleanup)(EVP_CIPHER_CTX *); /* cleanup ctx */ /*清除环磷酰胺* /
 int ctx_size;  /* how big ctx->cipher_data needs to be */ /*多大环磷酰胺，“密码数据需要* /
 int (*set_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *); /* Populate a ASN1_TYPE with parameters */ /*填充带参数的 ASN1 类型* 1 /
 int (*get_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *); /* Get parameters from a ASN1_TYPE */ /*从 1 的 ASN1 类型*参数/
 int (*ctrl)(EVP_CIPHER_CTX *, int type, int arg, void *ptr); /* Miscellaneous operations */ /* *杂项业务/
 void *app_data;  /* Application data */ /*应用程序数据* /
 } /* EVP_CIPHER */; /*执行副总裁加密法* /;
typedef struct evp_cipher_st EVP_CIPHER;
```

nid：　　算法的 ID 号，在 include/openssl/object.h 中定义；
block_size：加解密的分组长度
key_len：　密钥长度
iv_len：　　初始向量长度
flags：　　标志
(* init)：　初始化函数，提供密钥，IV 向量，算法上下文 CTX，加密还是解密
(* do_cipher)：　加解密函数，提供算法上下文 CTX，输出数据，输入数据和输入数据长度
(* clean_up)：　资源释放
ctx_size：　各算法相关数据大小,实际就是各算法的密钥数据
(*set_asn1_parameters)：设置 asn1 参数
(*get_asn1_parameters)：获取 asn1 参数

(*ctrl)：　其他控制操作

app_data：算法相关数据

另一个重要的数据结构是描述加密算法的**上下文结构 EVP_CIPHER_CTX**，这个结构是进入算法前由系统根据指定的算法提供的：

```
struct evp_cipher_ctx_st
 {
 const EVP_CIPHER *cipher;
 ENGINE *engine; /* functional reference if 'cipher' is ENGINE-provided */
 int encrypt;    /* encrypt or decrypt */
 int buf_len;    /* number we have left */
 unsigned char    oiv[EVP_MAX_IV_LENGTH]; /* original iv */
 unsigned char    iv[EVP_MAX_IV_LENGTH]; /* working iv */
 unsigned char buf[EVP_MAX_BLOCK_LENGTH];/* saved partial block */
 int num;        /* used by cfb/ofb mode */
 void *app_data;    /* application stuff */
 int key_len;    /* May change for variable length cipher */
 unsigned long flags; /* Various flags */
 void *cipher_data; /* per EVP data */
 int final_used;
 int block_mask;
 unsigned char final[EVP_MAX_BLOCK_LENGTH];/* possible final block */
 } /* EVP_CIPHER_CTX */;
typedef struct evp_cipher_ctx_st EVP_CIPHER_CTX;
```

参数为：

cipher：算法指针

engine：加解密引擎

encrypt：加密或解密

buf_len：剩余空间

oiv：原始的初始向量

iv：当前的初始向量

buf：保存的部分块数据

num：cfb/ofb 方式时的数据数量

app_data：应用相关数据

key_len：密钥长度

flags：标志

cipher_data：各算法相关部分，主要是各算法的 key 等

final_used：

block_mask：块的掩码

final：最后的分组块

1.2.2 算法接口

每种算法就是要填写各自的 EVP_CIPHER 结构，以 RC4 为例，定义了两个 RC4 的 EVP_CIPHER 结构，只是密钥长度不同，一个是 128 位(16 字节密钥)，一个是 40 位(5 字节密钥)，而算法都一样:

```
#ifndef OPENSSL_NO_RC4
#include <stdio.h>
#include "cryptlib.h"
#include <openssl/evp.h>
#include <openssl/objects.h>
#include <openssl/rc4.h>
/* FIXME: surely this is available elsewhere? */
#define EVP_RC4_KEY_SIZE    16
```

```
    //这个结构是各加密算法独有的,各算法的各自不同//也就是 EVP_CIPHER_CTX 结构中cipher_data//
    typedef struct
          {
          RC4_KEY ks; /* working key */
          } EVP_RC4_KEY;
    #define data(ctx) ((EVP_RC4_KEY *)(ctx)->cipher_data)
    static int rc4_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key,
        const unsigned char *iv,int enc);
    static int rc4_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out,
        const unsigned char *in, unsigned int inl);
    static const EVP_CIPHER r4_cipher=
      {
      NID_rc4,
      1,EVP_RC4_KEY_SIZE,0,
      EVP_CIPH_VARIABLE_LENGTH,
      rc4_init_key,
      rc4_cipher,
      NULL,
      sizeof(EVP_RC4_KEY),
      NULL,
      NULL,
      NULL
      };
    static const EVP_CIPHER r4_40_cipher=
      {
      NID_rc4_40,
      1,5 /* 40 bit */,0,
      EVP_CIPH_VARIABLE_LENGTH,
      rc4_init_key,
      rc4_cipher,
      NULL,
      sizeof(EVP_RC4_KEY),
      NULL,
      NULL,
      NULL
      };
    // 返回算法结构指针
    const EVP_CIPHER *EVP_rc4(void)
      {
      return(&r4_cipher);
      }
    const EVP_CIPHER *EVP_rc4_40(void)
      {
      return(&r4_40_cipher);
      }
    // 密钥初始化函数
    // ctx：加解密上下文；key：密钥字符串；iv：初始化向量；enc：加密还是解密
    static int rc4_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key,
        const unsigned char *iv, int enc)
      {
    // RC4 算法设置密钥函数
    // 一般来说，加解密时算法里用的密钥并不是用户输入的密码字符串本身，因为算法使用的密钥长度要求是固定的，通常为 64 位或 128 位，而用户自己定义的密码长度则不确定，所以一般都都要对用户输入的
```

密码进行变换，映射到一个固定长度密钥上，然后算法再使用该密钥加密，所以算法中用的密钥和用户的密码一般是不同的

```
RC4_set_key(&data(ctx)->ks,EVP_CIPHER_CTX_key_length(ctx),
  key);
  return 1;
  }
// 加解密处理函数
// ctx：加解密上下文；out：输出数据；in：输入数据；inl：输入数据长度
static int rc4_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out,
    const unsigned char *in, unsigned int inl)
  {
  RC4(&data(ctx)->ks,inl,in,out);
  return 1;
  }
#endif
```

对于定义好 EVP_CIPHER 结构的加解密算法，最后通过 EVP_add_cipher()函数添加到系统算法链表中：

```
openssl/evp/c_alle.c
void OpenSSL_add_all_ciphers(void)
{
.....
#ifndef OPENSSL_NO_RC4
  EVP_add_cipher(EVP_rc4());
  EVP_add_cipher(EVP_rc4_40());
#endif
.....
```

以后外部函数要调用 RC4 算法，实际就是找到 RC4 的 EVP_CIPHER 结构指针，进行初始化后进行加解密处理：

如 openssl 命令：

```
openssl enc –rc4 –in aaa.txt –out aaa.enc
```

执行顺序如下：

openssl/apps/openssl.c

main() -> do_cmd()->MAIN() (apps/enc.c) -> EVP_get_cipherbyname(), EVP_BytesToKey(), BIO_set_cipher(),

BIO_set_cipher -> EVP_CipherInit_ex -> ctx->cipher->init

而加密算法作为 BIO 的一个环节被 BIO_push 到 BIO 中，这样就通过直接 BIO 读写操作就调用了相应的加解密算法。

3. 块加密算法定义宏

对于块加密算法，如 AES、CAST、Blowfish 等，其加解密函数都是类似的，openssl 更是定义了一系列宏来简化算法的定义，如对于 Blowfish 算法接口：

```
/* openssl/evp/e_bf.c */
#ifndef OPENSSL_NO_BF
#include <stdio.h>
#include "cryptlib.h"
#include <openssl/evp.h>
#include "evp_locl.h"
#include <openssl/objects.h>
#include <openssl/blowfish.h>
static int bf_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key, const unsigned char *iv, int enc);
typedef struct
  {
  BF_KEY ks;
```

```
} EVP_BF_KEY;
#define data(ctx) EVP_C_DATA(EVP_BF_KEY,ctx)
//定义块加密算法的宏
IMPLEMENT_BLOCK_CIPHER(bf, ks, BF, EVP_BF_KEY, NID_bf, 8, 16, 8, 64,
    EVP_CIPH_VARIABLE_LENGTH, bf_init_key, NULL,
    EVP_CIPHER_set_asn1_iv, EVP_CIPHER_get_asn1_iv, NULL)
// 显式只需要定义一个初始化密钥函数就可以了
static int bf_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key,
      const unsigned char *iv, int enc)
 {
 BF_set_key(&data(ctx)->ks,EVP_CIPHER_CTX_key_length(ctx),key);
 return 1;
 }
#endif
```

在 openssl/evp/evp_locl.h 中该宏定义为：

```
#define IMPLEMENT_BLOCK_CIPHER(cname, ksched, cprefix, kstruct, nid, \
block_size, key_len, iv_len, cbits, \
flags, init_key, \
cleanup, set_asn1, get_asn1, ctrl) \
 BLOCK_CIPHER_all_funcs(cname, cprefix, cbits, kstruct, ksched) \
 BLOCK_CIPHER_defs(cname, kstruct, nid, block_size, key_len, iv_len, \
cbits, flags, init_key, cleanup, set_asn1, \
get_asn1, ctrl)
```

1) 宏 BLOCK_CIPHER_all_funcs 用来定义加解密函数，定义为：

```
#define BLOCK_CIPHER_all_funcs(cname, cprefix, cbits, kstruct, ksched) \
 BLOCK_CIPHER_func_cbc(cname, cprefix, kstruct, ksched) \
 BLOCK_CIPHER_func_cfb(cname, cprefix, cbits, kstruct, ksched) \
 BLOCK_CIPHER_func_ecb(cname, cprefix, kstruct, ksched) \
 BLOCK_CIPHER_func_ofb(cname, cprefix, cbits, kstruct, ksched)
```

其中 BLOCK_CIPHER_func_cbc 定义为：

```
#define BLOCK_CIPHER_func_cbc(cname, cprefix, kstruct, ksched) \
static int cname##_cbc_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out, const unsigned char *in, unsigned int inl) \
{\
 cprefix##_cbc_encrypt(in, out, (long)inl, &((kstruct *)ctx->cipher_data)->ksched, ctx->iv, ctx->encrypt);\
 return 1;\
}
```

此处注意一个#define 的技巧：可以用#define 替换变量或函数名，参数在打头或结尾处时，可分别在参数后或参数前用##；参数在中间，参数开头结尾都用##；

这样对于 blowfish 定义来说，上面这个宏实际定义这样一个函数：

```
static int bf_cbc_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out, const unsigned char *in, unsigned int inl)
{
 bf_cbc_encrypt(in, out, (long)inl, &((kstruct *)ctx->cipher_data)->ksched, ctx->iv, ctx->encrypt);
 return 1;
}
```

其他几个宏定义类似，定义不同模式的 bf 加密算法，包括 cbc，cfb，ecb，ofb。

2) 宏 BLOCK_CIPHER_defs 用于定义算法结构，定义为：

```
#define BLOCK_CIPHER_defs(cname, kstruct, \
nid, block_size, key_len, iv_len, cbits, flags, \
init_key, cleanup, set_asn1, get_asn1, ctrl) \
```

```
BLOCK_CIPHER_def_cbc(cname, kstruct, nid, block_size, key_len, iv_len, flags, \
init_key, cleanup, set_asn1, get_asn1, ctrl) \
BLOCK_CIPHER_def_cfb(cname, kstruct, nid, key_len, iv_len, cbits, \
flags, init_key, cleanup, set_asn1, get_asn1, ctrl) \
BLOCK_CIPHER_def_ofb(cname, kstruct, nid, key_len, iv_len, cbits, \
flags, init_key, cleanup, set_asn1, get_asn1, ctrl) \
BLOCK_CIPHER_def_ecb(cname, kstruct, nid, block_size, key_len, iv_len, flags, \
init_key, cleanup, set_asn1, get_asn1, ctrl)
```

其中 BLOCK_CIPHER_def_cbc 定义为：

```
#define BLOCK_CIPHER_def_cbc(cname, kstruct, nid, block_size, key_len, \
iv_len, flags, init_key, cleanup, set_asn1, \
get_asn1, ctrl) \
BLOCK_CIPHER_def1(cname, cbc, cbc, CBC, kstruct, nid, block_size, key_len, \
iv_len, flags, init_key, cleanup, set_asn1, get_asn1, ctrl)
```

而 BLOCK_CIPHER_def1 定义为：

```
#define BLOCK_CIPHER_def1(cname, nmode, mode, MODE, kstruct, nid, block_size, \
key_len, iv_len, flags, init_key, cleanup, \
set_asn1, get_asn1, ctrl) \
static const EVP_CIPHER cname##_##mode = { \
  nid##_##nmode, block_size, key_len, iv_len, \
  flags | EVP_CIPH_##MODE##_MODE, \
  init_key, \
  cname##_##mode##_cipher, \
  cleanup, \
  sizeof(kstruct), \
  set_asn1, get_asn1,\
  ctrl, \
  NULL \
}; \
const EVP_CIPHER *EVP_##cname##_##mode(void) { return &cname##_##mode; }
```

所以宏 BLOCK_CIPHER_def_cbc 展开后就是 blowfish 的 cbc 算法结构定义：

```
static const EVP_CIPHER bf_cbc = {
  NID_bf_cbc, 8, 16, 8,
  EVP_CIPH_VARIABLE_LENGTH | EVP_CIPH_CBC_MODE,
  bf_init_key,
  bf_cbc_cipher,
  NULL,
  sizeof(EVP_BF_KEY),
  EVP_CIPHER_set_asn1_iv, EVP_CIPHER_get_asn1_iv,
  NULL,
  NULL
};
const EVP_CIPHER *EVP_bf_cbc(void) { return &bf_cbc; }
```

其他几个宏定义类似，定义不同模式的 bf 加密算法的 EVP_CIPHER 数据结构，包括 cbc，cfb，ecb，ofb。

4. 结论

openssl 使用了统一的 EVP_CIPHER 算法结构，很好地封装了各种对称加密算法，实现了算法的对象化。

5. 附录

cbc，cfb，ecb，ofb 等并不是新的加密算法，而是对加密算法的应用模式。

ECB：Electronic Code Book，电子密码本模式，最基本的加密模式，也就是通常理解的加密，相同的明文将永远加密成相同的密文，无初始向量，容易受到密码本重放攻击，一般情况下很少用。

CBC：Cipher Block Chaining，密码分组链接，明文被加密前要与前面的密文进行异或运算后再加密，因此只要选择不同的初始向量，相同的密文加密后会形成不同的密文，这是目前应用最广泛的模式。CBC 加密后的密文是上下文相关的，但明文的错误不会传递到后续分组，但如果一个分组丢失，后面的分组将全部作废(同步错误)。

CFB：Cipher FeedBack，密码反馈，类似于自同步序列密码，分组加密后，按 8 位分组将密文和明文进行移位异或后得到输出同时反馈回移位寄存器，优点最小可以按字节进行加解密，也可以是 n 位的，CFB 也是上下文相关的，CFB 模式下，明文的一个错误会影响后面的密文(错误扩散)。

OFB：Output Feedback，输出反馈，将分组密码作为同步序列密码运行，和 CFB 相似，不过 OFB 用的是前一个 n 位密文输出分组反馈回移位寄存器，OFB 没有错误扩散问题。

---

openssl 之 EVP 系列之 01-20

openssl 之 EVP 系列之 1---算法封装---根据 openssl doc\crypto\EVP.pod 翻译和自己的理解写成（作者：DragonKing，Mail：wzhah@263.net，发布于：http://openssl.126.com 之 openssl 专业论坛，版本：openssl-0.9.7）

EVP 系列的函数定义包含在"evp.h"里面，这是一系列封装了 openssl 加密库里面所有算法的函数。通过这样的统一的封装，使得只需要在初始化参数的时候做很少的改变，就可以使用相同的代码但采用不同的加密算法进行数据的加密和解密。

EVP 系列函数主要**封装了三大类型的算法**，要支持全部这些算法，请调用 OpenSSL_add_all_algorithms 函数，下面分别就其结构作一个简单的介绍。

【公开密钥算法】

函数名称：EVP_Seal*...*，EVP_Open*...*

功能描述：该系列函数封装提供了公开密钥算法的加密和解密功能，实现了电子信封的功能。

相关文件：p_seal.c,p_open.c

【数字签名算法】

函数名称：EVP_Sign*...*，EVP_Verify*...*

功能描述：该系列函数封装提供了数字签名算法和功能。

相关文件：p_sign.c,p_verify.c

【对称加密算法】

函数名称：EVP_Encrypt*...*

功能描述：该系列函数封装提供了对称加密算法的功能。

相关文件：evp_enc.c,p_enc.c,p_dec.c,e_*.c

【信息摘要算法】

函数名称：EVP_Digest*...*

功能描述：该系列函数封装实现了多种信息摘要算法。

相关文件：digest.c,m_*.c

【信息编码算法】

函数名称：EVP_Encode*...*

功能描述：该系列函数封装实现了 ASCII 码与二进制码之间的转换函数和功能。

相关文件：encode.c

注意：自从出现 engin 版本以后，所有对称加密算法和摘要算法可以用 ENGINE 模块实现的算法代替。如果 ENGINE 模块实现的对称加密和信息摘要函数被注册为缺省的实现算法，那么当使用各种 EVP 函数时，软件编译的时候会自动将该实现模块连接进去。

openssl 之 EVP 系列之 2---对称加密算法概述

---根据 openssl doc\crypto\EVP_EncryptInit.pod 和 doc\ssleay.txt cipher.doc 部分翻译和自己的理解写成（作者：DragonKing，Mail：wzhah@263.net，发布于：http://gdwzh.126.com 之 openssl 专业论坛，版本：openssl-0.9.7）

对称加密算法封装的函数系列名字是以 EVP_Encrypt*...*开头的，其实，这些函数只是简单调用了 EVP_Cipher*...*系列的同名函数，换一个名字可能是为了更好的区别和理解。除了实现

了对称加密算法外，EVP_Encrypt*...*系列还对块加密算法提供了缓冲功能。以后我们可能会更多使用 EVP_Cipher 的术语，因为它是真正的实现结构。

EVP_Cipher*...*得以实现的一个基本结构是下面定义的一个算法结构，它定义了 EVP_Cipher 系列函数应该采用什么算法进行数据处理，其定义如下（evp.h）：

```
typedef 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 *);
 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); /* Miscellaneous operations */
 void *app_data;
}EVP_CIPHER;
```

下面对这个结构的部分成员的含义作一些解释：

nid——是算法类型的 nid 识别号，openssl 里面每个对象都有一个内部唯一的识别 ID

block_size——是每次加密的数据块的长度，以字节为单位

key_len——各种不同算法缺省的密钥长度

iv_len——初始化向量的长度

init——算法结构初始化函数，可以设置为加密模式还是解密模式

do_cipher——进行数据加密或解密的函数

cleanup——释放 EVP_CIPHER_CTX 结构里面的数据和设置。

ctx_size——设定 ctx->cipher_data 数据的长度

set_asn1_parameters——在 EVP_CIPHER_CTX 结构中通过参数设置一个 ASN1_TYPE

get_asn1_parameters——从一个 ASN1_TYPE 中取得参数

ctrl——其它各种操作函数

app_data——应用数据

通过定义这样一个指向这个结构的指针，你就可以在连接程序的时候只连接自己使用的算法；而如果你是通过一个整数来指明应该使用什么算法的话，会导致所有算法的代码都被连接到代码中。通过这样一个结构，还可以自己增加新的算法。

在这个基础上，每个 EVP_Cipher*...*函数都维护着一个指向一个 EVP_CIPHER_CTX 结构的指针。

```
typedef 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];
 int num;
 void *app_data;
 int key_len;
```

```
unsigned long flags;
void *cipher_data;
int final_used;
int block_mask;
unsigned char final[EVP_MAX_BLOCK_LENGTH];
} EVP_CIPHER_CTX;
```

下面对这个结构部分成员做简单的解释：

cipher——是该结构相关的一个 EVP_CIPHER 算法结构

engine——如果加密算法是 ENGINE 提供的，那么该成员保存了相关的函数接口

encrypt——加密或解密的标志

buf_len——该结构缓冲区里面当前的数据长度

oiv——初始的初始化向量

iv——工作时候使用的初始化向量

buf——保存下来的部分需要数据

num——在 cfb/ofb 模式的时候指定块长度

app_data——应用程序要处理数据

key_len——密钥长度，算法不一样长度也不一样

cipher_data——加密后的数据

上述两个结构是 EVP_Cipher(EVP_Encrypt)系列的两个基本结构，它们的其它一些列函数都是以这两个结构为基础实现了。文件 evp\evp_enc.c 是最高层的封装实现，各种加密的算法的封装在 p_enc.c 里面实现，解密算法的封装在 p_dec.c 里面实现，而各个 e_*.c 文件则是真正实现了各种算法的加解密功能，当然它们其实也是一些封装函数，真正的算法实现在各个算法同名目录里面的文件实现。

openssl 之 EVP 系列之 3---EVP_Encrypt 支持的对称加密算法列表

---根据 openssl doc\crypto\EVP_EncryptInit.pod 和 doc\ssleay.txt cipher.doc 部分翻译和自己的理解写成(作者：DragonKing Mail：wzhah@263.net 发布于：http://openssl.126.com 之 openssl 专业论坛 版本：openssl-0.9.7)

openssl 对称加密算法的格式都以函数形式提供，其实该函数返回一个该算法的结构体，其形式一般如下：

EVP_CIPHER* EVP_*(void)

在 openssl 中，所有提供的对称加密算法长度都是固定的，有特别说明的除外。下面对这些算法进行分类的介绍，首先介绍一下算法中使用的通用标志的含义。

【通用标志】

ecb——电子密码本（Electronic Code Book)加密方式

cbc——加密块链接（Cipher Block Chaining）加密方式

cfb——64 位加密反馈（Cipher Feedback）加密方式

ofb——64 位输出反馈（Output Feedback）加密方式

ede——该加密算法采用了加密、解密、加密的方式，第一个密钥和最后一个密钥是相同的

ede3——该加密算法采用了加密、解密、加密的方式，但是三个密钥都不相同

【NULL 算法】

函数：EVP_enc_null()

说明：该算法不作任何事情，也就是没有进行加密处理

【DES 算法】

函数：EVP_des_cbc(void), EVP_des_ecb(void), EVP_des_cfb(void), EVP_des_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 DES 算法

【使用两个密钥的 3DES 算法】

函数：

EVP_des_ede_cbc(void),

EVP_des_ede(),

EVP_des_ede_ofb(void),
EVP_des_ede_cfb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 3DES 算法，算法的第一个密钥和最后一个密钥相同，事实上就只需要两个密钥

【使用三个密钥的 3DES 算法】

函数：

EVP_des_ede3_cbc(void),
EVP_des_ede3(),
EVP_des_ede3_ofb(void),
EVP_des_ede3_cfb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 3DES 算法，算法的三个密钥都不相同

【DESX 算法】

函数：EVP_desx_cbc(void)

说明：CBC 方式 DESX 算法

【RC4 算法】

函数：EVP_rc4(void)

说明：RC4 流加密算法。该算法的密钥长度可以改变，缺省是 128 位。

【40 位 RC4 算法】

函数：EVP_rc4_40(void)

说明：密钥长度 40 位的 RC4 流加密算法。该函数可以使用 EVP_rc4 和 EVP_CIPHER_CTX_set_key_length 函数代替。

【IDEA 算法】

函数：

EVP_idea_cbc(),
EVP_idea_ecb(void),
EVP_idea_cfb(void),
EVP_idea_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 IDEA 算法。

【RC2 算法】

函数：EVP_rc2_cbc(void), EVP_rc2_ecb(void), EVP_rc2_cfb(void), EVP_rc2_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 RC2 算法，该算法的密钥长度是可变的，可以通过设置有效密钥长度或有效密钥位来设置参数来改变。缺省的是 128 位。

【定长的两种 RC2 算法】

函数：

EVP_rc2_40_cbc(void),
EVP_rc2_64_cbc(void)

说明：分别是 40 位和 64 位 CBC 模式的 RC2 算法。

【Blowfish 算法】

函数：

EVP_bf_cbc(void),
EVP_bf_ecb(void),
EVP_bf_cfb(void),
EVP_bf_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 Blowfish 算法，该算法的密钥长度是可变的

【CAST 算法】

函数：

EVP_cast5_cbc(void),

EVP_cast5_ecb(void),
EVP_cast5_cfb(void),
EVP_cast5_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 CAST 算法，该算法的密钥长度是可变的

【RC5 算法】

函数：

EVP_rc5_32_12_16_cbc(void),
EVP_rc5_32_12_16_ecb(void),
EVP_rc5_32_12_16_cfb(void),
EVP_rc5_32_12_16_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 RC5 算法，该算法的密钥长度可以根据参数“number of rounds”（算法中一个数据块被加密的次数）来设置，缺省的是 128 位密钥，加密次数为 12 次。目前来说，由于 RC5 算法本身实现代码的限制，加密次数只能设置为 8、12 或 16。

【128 位 AES 算法】

函数：

EVP_aes_128_ecb(void)，
EVP_aes_128_cbc(void)，
PEVP_aes_128_cfb(void)，
EVP_aes_128_ofb(void)
说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 128 位 AES 算法

【192 位 AES 算法】

函数：
EVP_aes_192_ecb(void)，
EVP_aes_192_cbc(void)，
PEVP_aes_192_cfb(void)，
EVP_aes_192_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 192 位 AES 算法

【256 位 AES 算法】

函数：
EVP_aes_256_ecb(void)，
EVP_aes_256_cbc(void)，
PEVP_aes_256_cfb(void)，
EVP_aes_256_ofb(void)

说明：分别是 CBC 方式、ECB 方式、CFB 方式以及 OFB 方式的 256 位 AES 算法

上述的算法是 0.9.7 版本支持的所有对称加密算法，关于算法的详细情况，请参看该算法的资料了或本系列后续的文章。

openssl 之 EVP 系列之 4---EVP_Encrypt 系列函数详解(一)

---根据 openssl doc\crypto\EVP_EncryptInit.pod 和 doc\ssleay.txt cipher.doc 部分翻译和自己的理解写成（作者：DragonKing，Mail：wzhah@263.net，发布于：http://openssl.126.com 之 openssl 专业论坛，版本：openssl-0.9.7）

EVP_Cipher 系列包含了很多函数，我将他们大概分成两部分来介绍，一部分是基本函数系列，就是本文要介绍的，另一个部分是设置函数系列，将在后面的文章进行介绍。**基本系列函数主要是进行基本的加密和解密操作的函数，他们的定义如下**（openssl\evp.h）：

```
int EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
```

```
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);
int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
ENGINE *impl, unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv);
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);
int EVP_DecryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv);
int EVP_DecryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);
int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
unsigned char *key, unsigned char *iv, int enc);
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm,
int *outl);
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
```

其实在这里列出的函数虽然很多，但是大部分是功能重复的，有的是旧的版本支持的函数，新的版本中已经可以不再使用了。事实上，函数 EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_CipherInit 以及 EVP_CipherFinal 在新代码中不应该继续使用，他们保留下来只是为了兼容以前的代码。

在新的代码中，应该使用 EVP_EncryptInit_ex、EVP_EncryptFinal_ex、EVP_DecryptInit_ex、EVP_DecryptFinal_ex、EVP_CipherInit_ex 以及 EVP_CipherFinal_ex 函数，因为它们可以在每次调用完算法后，不用重新释放和分配已有 EVP_CIPHER_CTX 结构的内存的情况下重用该结构，方便很多。下面我们分别对这些函数进行介绍。

【EVP_CIPHER_CTX_init】

该函数初始化一个 EVP_CIPHER_CTX 结构体，只有初始化后该结构体才能在下面介绍的函数中使用。操作成功返回 1，否则返回 0。

【EVP_EncryptInit_ex】

该函数采用 ENGINE 参数 impl 的算法来设置并初始化加密结构体。其中，参数 ctx 必须在调用本函数之前已经进行了初始化。参数 type 通常通过函数类型来提供参数，如 EVP_des_cbc 函数的形式，即我们上一章中介绍的对称加密算法的类型。如果参数 impl 为 NULL，那么就会使用缺省的实现算法。参数 key 是用来加密的对称密钥，iv 参数是初始化向量（如果需要的话）。在算法中真正使用的密钥长度和初始化密钥长度是根据算法来决定的。在调用该函数进行初始化的时候，除了参数 type 之外，所有其它参数可以设置为 NULL，留到以后调用其它函数的时候再提供，这时候参数 type 就设置为 NULL 就可以了。在缺省的加密参数不合适的时候，可以这样处理。操作成功返回 1，否则返回 0。

【EVP_EncryptUpdate】

该函数执行对数据的加密。该函数加密从参数 in 输入的长度为 inl 的数据，并将加密好的数据写入到参数 out 里面去。可以通过反复调用该函数来处理一个连续的数据块。写入到 out 的数据数量是由已经加密的数据的对齐关系决定的，理论上来说，从 0 到(inl+cipher_block_size-1)

的任何一个数字都有可能（单位是字节），所以输出的参数 out 要有足够的空间存储数据。写入到 out 中的实际数据长度保存在 outl 参数中。操作成功返回 1，否则返回 0。

【EVP_EncryptFinal_ex】

该函数处理最后（Final）的一段数据。在函数在 padding 功能打开的时候（缺省）才有效，这时候，它将剩余的最后的所有数据进行加密处理。该算法使用标志的块 padding 方式（AKA PKCS padding）。加密后的数据写入到参数 out 里面，参数 out 的长度至少应该能够一个加密块。写入的数据长度信息输入到 outl 参数里面。该函数调用后，表示所有数据都加密完了，不应该再调用 EVP_EncryptUpdate 函数。如果没有设置 padding 功能，那么本函数不会加密任何数据，如果还有剩余的数据，那么就会返回错误信息，也就是说，这时候数据总长度不是块长度的整数倍。操作成功返回 1，否则返回 0。

PKCS padding 标准是这样定义的，在被加密的数据后面加上 n 个值为 n 的字节，使得加密后的数据长度为加密块长度的整数倍。无论在什么情况下，都是要加上 padding 的，也就是说，如果被加密的数据已经是块长度的整数倍，那么这时候 n 就应该等于块长度。比如，如果块长度是 9，要加密的数据长度是 11，那么 5 个值为 5 的字节就应该增加在数据的后面。

【EVP_DecryptInit_ex, EVP_DecryptUpdate 和 EVP_DecryptFinal_ex】

这三个函数是上面三个函数相应的解密函数。这些函数的参数要求基本上都跟上面相应的加密函数相同。如果 padding 功能打开了，EVP_DecryptFinal 会检测最后一段数据的格式，如果格式不正确，该函数会返回错误代码。此外，如果打开了 padding 功能，EVP_DecryptUpdate 函数的参数 out 的长度应该至少为（inl+cipher_block_size）字节；但是，如果加密块的长度为 1，则其长度为 inl 字节就足够了。三个函数都是操作成功返回 1，否则返回 0。

需要注意的是，虽然在 padding 功能开启的情况下，解密操作提供了错误检测功能，但是该功能并不能检测输入的数据或密钥是否正确，所以即便一个随机的数据块也可能无错的完成该函数的调用。如果 padding 功能关闭了，那么当解密数据长度是块长度的整数倍时，操作总是返回成功的结果。

【EVP_CipherInit_ex, EVP_CipherUpdate 和 EVP_CipherFinal_ex】

事实上，上面介绍的函数都是调用这三个函数实现的，它们是更底层的函数。完成了数据的加密和解密功能。他们根据参数 enc 决定执行加密还是解密操作，如果 enc 为 1，则加密；如果 enc 为 0，则解密；如果 enc 是－1，则不改变数据。三个函数都是操作成功返回 1，否则返回 0。

【EVP_CIPHER_CTX_cleanup】

该函数清除一个 EVP_CIPHER_CTX 结构中的所有信息并释放该结构占用的所有内存。在使用上述的函数完成一个加密算法过程后应该调用该函数，这样可以避免一些敏感信息遗留在内存造成安全隐犯。操作成功返回 1，否则返回 0。

【EVP_EncryptInit, EVP_DecryptInit 和 EVP_CipherInit】

这三个函数的功能分别跟函数 EVP_EncryptInit_ex, EVP_DecryptInit_ex 和 EVP_CipherInit_ex 功能相同，只是他们的 ctx 参数不需要进行初始化，并且使用缺省的算法库。三个函数都是操作成功返回 1，否则返回 0。

【EVP_EncryptFinal, EVP_DecryptFinal 和 EVP_CipherFinal】

这三个函数分别跟函数 EVP_EncryptFinal_ex, EVP_DecryptFinal_ex 以及 EVP_CipherFinal_ex 函数功能相同，不过，他们的参数 ctx 会在调用后自动释放。三个函数都是操作成功返回 1，否则返回 0。

openssl 之 EVP 系列之 5---EVP_Encrypt 系列函数详解(二)

---根据 openssl doc\crypto\EVP_EncryptInit.pod 和 doc\ssleay.txt cipher.doc 部分翻译和自己的理解写成（作者：DragonKing, Mail: wzhah@263.net ，发布于：http://gdwzh.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

前面的文章我们介绍了 EVP_ENcrypt 系列函数的基本部分，本文将介绍他们的一些扩充部分，即一些参数设置和其它辅助的函数，其定义如下（openssl\evp.h）：

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
int EVP_CIPHER_CTX_set_key_length(EVP_CIPHER_CTX *x, int keylen);
```

```
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))
#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
#define EVP_CIPHER_nid(e) ((e)->nid)
#define EVP_CIPHER_block_size(e) ((e)->block_size)
#define EVP_CIPHER_key_length(e) ((e)->key_len)
#define EVP_CIPHER_iv_length(e) ((e)->iv_len)
#define EVP_CIPHER_flags(e) ((e)->flags)
#define EVP_CIPHER_mode(e) ((e)->flags) & EVP_CIPH_MODE)
int EVP_CIPHER_type(const EVP_CIPHER *ctx);
#define EVP_CIPHER_CTX_cipher(e) ((e)->cipher)
#define EVP_CIPHER_CTX_nid(e) ((e)->cipher->nid)
#define EVP_CIPHER_CTX_block_size(e) ((e)->cipher->block_size)
#define EVP_CIPHER_CTX_key_length(e) ((e)->key_len)
#define EVP_CIPHER_CTX_iv_length(e) ((e)->cipher->iv_len)
#define EVP_CIPHER_CTX_get_app_data(e) ((e)->app_data)
#define EVP_CIPHER_CTX_set_app_data(e,d) ((e)->app_data=(char *)(d))
#define EVP_CIPHER_CTX_type(c) EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_flags(e) ((e)->cipher->flags)
#define EVP_CIPHER_CTX_mode(e) ((e)->cipher->flags & EVP_CIPH_MODE)
int EVP_CIPHER_param_to_asn1(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
int EVP_CIPHER_asn1_to_param(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
```

【EVP_CIPHER_CTX_set_padding】

该函数设置是否采用 padding 功能.在算法缺省的情况下，是使用标准的块 padding 功能的，并且在解密的时候会自动接测 padding 并将它删除。如果将参数 pad 设置为 0，则 padding 功能就会被禁止，那么在加密和解密的时候，数据应该为加密块长度的整数倍，否则就会出错。函数恒返回 1。

【EVP_CIPHER_CTX_set_key_length】

该函数进行加密算法结构 EVP_CIPHER_CTX 密钥长度的设置。如果算法是一个密钥长度固定的算法，那么如果设置的密钥长度跟它固定的长度不一致，就会产生错误。

【EVP_get_cipherbyname, EVP_get_cipherbynid 和 EVP_get_cipherbyobj】

这三个函数都根据给定的参数返回一个 EVP_CIPHER 结构，不同的是给定的参数分别是算法名称、算法的 NID 和一个 ASN1_OBJECT 结构。具体的算法名称、NID 以及 ASN1_OBJECT 结构请参看 object\boject.h 文件的定义。

【EVP_CIPHER_nid 和 EVP_CIPHER_CTX_nid】

这两个函数返回 EVP_CIPHER 或 EVP_CIPHER_CTX 结构内部的算法的 NID。返回的 NID 值只是一个内部存储的值，并不一定真的有相应的 OBJECT 定义。

【EVP_CIPHER_key_length 和 EVP_CIPHER_CTX_key_length】

这两个函数返回 EVP_CIPHER 或 EVP_CIPHER_CTX 结构内部的算法的密钥长度。常量 EVP_MAX_KEY_LENGTH 定义了所有算法最长的密钥长度。需要注意的是，对于 EVP_CIPHER_key_length 函数来说，对特定的一种算法密钥长度是不变的，但是 EVP_CIPHER_CTX_key_length 函数对同一个算法密钥长度却是可变的。

【EVP_CIPHER_iv_length 和 EVP_CIPHER_CTX_iv_length】

这两个函数返回 EVP_CIPHER 或 EVP_CIPHER_CTX 结构内部的算法的初始化向量长度。如果算法不使用 IV，那么就会返回 0。常量 EVP_MAX_IV_LENGTH 定义了所有算法最长的 IV 长度

【EVP_CIPHER_block_size 和 EVP_CIPHER_CTX_block_size】

这两个函数返回 EVP_CIPHER 或 EVP_CIPHER_CTX 结构内部的算法的加密块长度。常量 EVP_MAX_IV_LENGTH 也是所有算法最长的块长度。

【EVP_CIPHER_type 和 EVP_CIPHER_CTX_type】

这两个函数返回 EVP_CIPHER 或 EVP_CIPHER_CTX 结构内部的算法的类型。该类型的值是算法的 NID，一般来说，NID 忽略了算法的一些参数，如 40 位和 129 位 RC2 算法的 NID 是相同的。如果算法没有相应定义的 NID 或者不是 ASN1 所支持的，那么本函数就会返回 NID_undef。

【EVP_CIPHER_CTX_cipher】

该函数返回 EVP_CIPHER_CTX 结构里面的 EVP_CIPHER 结构。

【EVP_CIPHER_mode 和 EVP_CIPHER_CTX_mode】

这两个函数返回相应结构算法的块加密模式，包括 EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE 和 EVP_CIPH_OFB_MODE；如果算法是流加密算法，那么就返回 EVP_CIPH_STREAM_CIPHER 。

【EVP_CIPHER_param_to_asn1】

该函数设置算法结构的参数，一般来说设置的值包括了所有参数和一个 IV 值。如果算法有 IV，那么调用该函数时 IV 是必须设置的。该函数必须在所设置的算法结构使用之前（如调用 EVP_EncryptUpdate 和 EVP_DecryptUpdate 函数之前）调用。如果 ASN1 不支持该算法，那么调用该函数将导致失败。操作成功返回 1，否则返回 0。

【EVP_CIPHER_asn1_to_param】

该函数给用算法结构里面的值设置参数 type 的结构。其设置的内容由具体的算法决定。如在 RC2 算法中，它会设置 IV 和有效密钥长度。本函数应该在算法结构的基本算法类型已经设置了但是密钥还没有设置之前调用。例如，调用 EVP_CipherInit 函数的时候使用参数 IV，并将 key 设置位 NULL，然后就应该调用本函数，最后再调用 EVP_CipherInit，这时候除了 key 设置位 NULL 外所有参数都应该设置。当 ASN1 不支持不支持该算法或者有参数不能设置的时候(如 RC2 的有效密钥长度不支持)，该函数调用就会失败。操作成功返回 1，否则返回 0。

【EVP_CIPHER_CTX_ctrl】

该函数可以设置不同算法的特定的参数。目前只有 RC2 算法的有效密钥长度和 RC5 算法的加密次数（rounds)可以进行设置。

BTW：我自己感觉都写的有一点慢了，知道大家想知道怎么用来编程，但是，先把这么多函数介绍清楚了，下面看起来就会轻松多了，下一篇就将介绍 EVP_Encrypt*...*系列函数的编程框架，并举几个例子。

openssl 之 EVP 系列之 6---EVP_Encrypt 系列函数编程架构及例子

---根据 openssl doc\crypto\EVP_EncryptInit.pod 和 doc\ssleay.txt cipher.doc 部分翻译和自己的理解写成(作者：DragonKing, Mail：wzhah@263.net ，发布于：http://openssl.126.com 版本：openssl-0.9.7)

在前面的两篇文章，已经对 EVP_Encrypt*...*系列函数做了详细的介绍，本章将说明该系列函数通用的应用架构，并举几个函数应用例子。

【应用架构】

一般来说，EVP_Encrypt*...*系列函数的应用架构如下所描述(假设加密算法为 3DES):

1.定义一些必须的变量

```
char key[EVP_MAX_KEY_LENGTH];
char iv[EVP_MAX_IV_LENGTH];
EVP_CIPHER_CTX ctx;
unsigned char out[512+8];
int outl;
```

2.给变量 key 和 iv 赋值，这里使用了函数 EVP_BytesToKey，该函数从输入密码产生了密钥 key 和初始化向量 iv，该函数将在后面做介绍。如果可以有别的办法设定 key 和 iv，该函数的调用不是必须的

```
EVP_BytesToKey(EVP_des_ede3_cbc,EVP_md5,NULL,passwd,strlen(passwd),key,iv);
```

3.初始加密算法结构 EVP_CIPHER_CTX

```
EVP_EncryptInit_ex(&ctx, EVP_des_ede3_cbc(), NULL, key, iv);
```

4.进行数据的加密操作

```
while (....)
{
 EVP_EncryptUpdate(ctx,out,&outl,in,512);
}
```

一般来说采用了循环的结构进行处理，每次循环加密数据为 512 字节，密文输出到 out，out 和 int 应该是指向不相同的内存的。

5.结束加密，输出最后的一段 512 字节的数据

EVP_EncryptFinal_ex(&ctx, out, &outl)

该函数会进行加密的检测，如果加密过程有误，一般会检查出来。

说明：加密跟上述过程是一样的，只不过要使用 EVP_Decrypt*...*系列函数。

【例子】

1.取得算法 RC5 使用的循环次数（round）

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC5_ROUNDS, 0, &nrounds);
```

2.取得算法 RC2 的有效密钥长度

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC2_KEY_BITS, 0, &key_bits);
```

3.设置算法 RC5 使用的循环次数（round）

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC5_ROUNDS, nrounds, NULL);
```

4.设置算法 RC2 的有效密钥长度

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC2_KEY_BITS, key_bits, NULL);
```

5.使用 Blowfish 算法加密一个字符串

```
int do_crypt(char *outfile)
{
unsigned char outbuf[1024];
int outlen, tmplen;
//其实 key 和 iv 一般应该从别的地方得到，这里固定了至少作为演示使用的
unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
unsigned char iv[] = {1,2,3,4,5,6,7,8};
char intext[] = "Some Crypto Text";
EVP_CIPHER_CTX ctx;
FILE *out;
EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit_ex(&ctx, EVP_bf_cbc(), NULL, key, iv);
if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, intext, strlen(intext)))
{
/* 出错处理*/
return 0;
}
//注意，传入给下面函数的输出缓存参数必须注意不能覆盖了原来的加密输出的数据
if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
{
/* 出错处理*/
return 0;
}
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
```

//注意，保存到文件的时候要使用二进制模式打开文件，因为密文数据是二进制的，而且，不能采用 strlen 函数，因为密文字符串不是以 NULL（0）为结束的字符串

```
out = fopen(outfile, "wb");
```

```
fwrite(outbuf, 1, outlen, out);
fclose(out);
return 1;
}
```

上面举的例子加密的密文可以使用 openssl 提供的应用程序 cipher.exe 来解密，命令如下：

openssl bf -in cipher.bin -K 000102030405060708090A0B0C0D0E0F -iv 0102030405060708 -d

6.使用文件 I/O 和 80 位密钥 RC2 算法的通用加密解密函数实例，该函数可以进行加密，也可以进行解密，由参数 do_encrypt 决定，该参数为 1 时则执行加密，为 0 时则执行解密。

```
int do_crypt(FILE *in, FILE *out, int do_encrypt)
{
/*为输出缓冲设置足够的附加空间*/
inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
int inlen, outlen;
//其实 key 和 iv 一般应该从别的地方得到，这里固定了至少作为演示使用的
unsigned char key[] = "0123456789";
unsigned char iv[] = "12345678";
//这时候不进行 key 和 IV 的设置，因为我们还要修改参数
EVP_CIPHER_CTX_init(&ctx);
EVP_CipherInit_ex(&ctx, EVP_rc2(), NULL, NULL, NULL, do_encrypt);
EVP_CIPHER_CTX_set_key_length(&ctx, 10);
//完成参数设置，进行 key 和 IV 的设置
EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);
for(;;)
{
inlen = fread(inbuf, 1, 1024, in);
if(inlen <= 0) break;
if(!EVP_CipherUpdate(&ctx, outbuf, &outlen, inbuf, inlen))
{
/*出错处理 */
return 0;
}
fwrite(outbuf, 1, outlen, out);
}
if(!EVP_CipherFinal_ex(&ctx, outbuf, &outlen))
{
/* 出错处理*/
return 0;
}
fwrite(outbuf, 1, outlen, out);
EVP_CIPHER_CTX_cleanup(&ctx);
return 1;
}
```

openssl 之 EVP 系列之 7---信息摘要算法结构概述

---根据 openssl doc\crypto\EVP_DigestInit.pod 翻译和自己的理解写成（作者：DragonKing, Mail: wzhah@263.net ，发布于：http://openssl.126.com 之 openssl 专业论坛，版本：openssl-0.9.7）

该系列函数封装了 openssl 加密库所有的信息摘要算法，通过这种 EVP 封装，当使用不同的信息摘要算法时，只需要对初始化参数修改一下就可以了，其它代码可以完全一样。这些算法包括 MD2、MD5 以及 SHA 等算法。

【EVP_MD 结构介绍】

所有的算法都维护着下面定义的结构的一个指针，在此基础上实现了算法的功能。该结构 EVP_MD 如下：

```
typedef 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,unsigned long 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 (*verify)();
 int required_pkey_type[5]; /*EVP_PKEY_xxx */
 int block_size;
 int ctx_size;
} EVP_MD;
```

下面对该结构体的部分参数解释：

type——信息摘要算法的 NID 标识

pkey_type——是信息摘要-签名算法体制的相应 NID 标识，如 NID_shaWithRSAEncryption

md_size——是信息摘要算法生成的信息摘要的长度，如 SHA 算法是 SHA_DIGEST_LENGTH，该值是 20

init——指向一个特定信息摘要算法的初始化函数，如对于 SHA 算法，指针指向 SHA_Init

update——指向一个真正计算摘要值的函数，例如 SHA 算法就是指向 SHA_Update

final——指向一个信息摘要值计算之后要调用的函数，该函数完成最后的一块数据的处理工作。例如 SHA 算法就是指向 SHA_Final.

copy——指向一个可以在两个 EVP_MD_CTX 结构之间拷贝参数值的函数

required_pkey_type——指向一个用来签名的算法 EVP_PKEY 的类型，如 SHA 算法就指向 EVP_PKEY_RSA_method

block_size——一个用来进行信息摘要的输入块的的长度（单位是字节），如 SHA 算法就是 SHA_CBLOCK

ctx_size——是 CTX 结构的长度，在 SHA 算法里面应该就是 sizeof(EVP_MD*)+sizeof(SHA_CTX)

如果你要增加新的算法，那么可以定义这个结构，并进行必要的一直，然后就可以使用通用的函数了。跟 EVP_CIPHER 系列函数一样，使用这个封装技术，就可以在使用一种摘要算法时，比如 MD5，在连接程序的时候就只连接 MD5 的代码。如果使用证书来标识算法，那么就会导致所有其它的信息摘要算法代码都连接到程序中去了。

【EVP_MD_CTX 结构介绍】

在调用函数的时候，一般来说需要传入上面说的 type 的参数和下面所定义的一个 CTX 结构，该结构 EVP_MD_CTX 定义如下：

```
typedef struct env_md_ctx_st
{
 const EVP_MD *digest;
 ENGINE *engine;
 unsigned long flags;
 void *md_data;
}EVP_MD_CTX ;
```

该结构的成员解释如下：

digest——指向上面介绍的 EVP_MD 结构的指针

engine——如果算法由 ENGINE 提供，该指针指向该 ENGINE

md_data——信息摘要数据

【支持的信息摘要算法】

EVP_md_null(void)
EVP_md2(void)
EVP_md4(void)
EVP_md5(void)
EVP_sha(void)
EVP_sha1(void)
EVP_dss(void)
EVP_dss1(void)
EVP_mdc2(void)
EVP_ripemd160(void)

openssl 之 EVP 系列之 8---EVP_Digest 系列函数详解

---根据 openssl doc\crypto\EVP_DigestInit.pod 翻译和自己的理解写成

（作者：DragonKing，Mail：wzhah@263.net，发布于：http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

EVP_Digest 系列提供了与 EVP_Encrypt 系列相似的函数，定义如下（openssl/evp.h）：

```
void EVP_MD_CTX_init(EVP_MD_CTX *ctx);
EVP_MD_CTX *EVP_MD_CTX_create(void);
int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md,
unsigned int *s);
int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx);
void EVP_MD_CTX_destroy(EVP_MD_CTX *ctx);
int EVP_MD_CTX_copy_ex(EVP_MD_CTX *out,const EVP_MD_CTX *in);
int EVP_DigestInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_DigestFinal(EVP_MD_CTX *ctx, unsigned char *md,
unsigned int *s);
int EVP_MD_CTX_copy(EVP_MD_CTX *out,EVP_MD_CTX *in);
#define EVP_MAX_MD_SIZE (16+20) /* The SSLv3 md5+sha1 type */
#define EVP_MD_type(e) ((e)->type)
#define EVP_MD_pkey_type(e) ((e)->pkey_type)
#define EVP_MD_size(e) ((e)->md_size)
#define EVP_MD_block_size(e) ((e)->block_size)
#define EVP_MD_CTX_md(e) (e)->digest)
#define EVP_MD_CTX_size(e) EVP_MD_size((e)->digest)
#define EVP_MD_CTX_block_size(e) EVP_MD_block_size((e)->digest)
#define EVP_MD_CTX_type(e) EVP_MD_type((e)->digest)
```

【EVP_MD_CTX_init】

该函数初始化一个 EVP_MD_CTX 结构。

【EVP_MD_CTX_create】

该函数创建一个 EVP_MD_CTX 结构，分配内存并进行初始化，返回该结构。

【EVP_DigestInit_ex】

该函数使用参数 impl 所指向的 ENGINE 设置该信息摘要结构体，参数 ctx 在调用本函数之前必须经过初始化。参数 type 一般是使用象 EVP_sha1 这样的函数的返回值。如果 impl 为 NULL，那么就会使用缺省实现的信息摘要函数。大多数应用程序里面 impl 是设置为 NULL 的。操作成功返回 1，否则返回 0。

【EVP_DigestUpdate】

该函数将参数 d 中的 cnt 字节数据进行信息摘要到 ctx 结构中去，该函数可以被调用多次，用以对更多的数据进行信息摘要。操作成功返回 1，否则返回 0。

【EVP_DigestFinal_ex】

本函数将 ctx 结构中的摘要信息数据返回到参数 md 中，如果参数 s 不是 NULL，那么摘要

数据的长度（字节）就会被写入到参数 s 中，大多数情况瞎，写入的值是 EVP_MAX_MD_SIZE。在调用本函数后，不能使用相同的 ctx 结构调用 EVP_DigestUpdate 再进行数据的信息摘要操作，但是如果调用 EVP_DigestInit_ex 函数重新初始化后可以进行新的信息摘要操作。操作成功返回 1，否则返回 0。

【EVP_MD_CTX_cleanup】

清除一个信息摘要结构，该函数应该在一个信息摘要结构使用后不再需要的时候调用。

【EVP_MD_CTX_destroy】

清除信息摘要结构并释放所有分配的内存空间，只有使用 EVP_MD_CTX_create 函数创建的信息摘要结构才能使用该函数进行释放。

【EVP_MD_CTX_copy_ex】

该函数可以用来将信息摘要数据从 in 结构拷贝到 out 结构中。如果有大量的数据需要进行信息摘要，而且这些数据只有最后几个字节不同的时候，使用该函数就显得特别有用，节省时间。其中，out 结构必须在调用本函数之前进行初始化。操作成功返回 1，否则返回 0。

【EVP_DigestInit】

该函数功能跟 EVP_DigestInit_ex 函数相同，但是 ctx 参数可以不用初始化，而且该函数只使用缺省实现的算法。

【EVP_DigestFinal】

该函数功能跟 EVP_DigestFinal_ex 函数相同，但是 ctx 结构会自动清除。一般来说，现在新的程序应该使用 EVP_DigestInit_ex 和 EVP_DigestFinal_ex 函数，因为这些函数可以在使用完一个 EVP_MD_CTX 结构后，不用重新声明和初始化该结构就能使用它进行新的数据处理，而且新的带_ex 的函数也可以使用非缺省的实现算法库。

【EVP_MD_CTX_copy】

该函数跟 EVP_MD_CTX_copy_ex 函数功能相同，但是 out 参数可以不用初始化。

【EVP_MD_size 和 EVP_MD_CTX_size】

这两个函数返回结构里面摘要信息的长度。

【EVP_MD_block_size 和 EVP_MD_CTX_block_size】

这两个函数返回摘要信息块处理的长度。

【EVP_MD_type 和 EVP_MD_CTX_type】

这两个函数返回信息摘要结构算法的 NID。例如，EVP_MD_type(EVP_sha1())返回 NID_sha1。该函数通常在设置 ASN1 OID 的时候使用。如果算法不存在，返回 NID_undef。

【EVP_MD_CTX_md】

该函数返回给定 EVP_MD_CTX 结构里面的 EVP_MD 结构。

【EVP_MD_pkey_type】

该函数返回信息摘要结构里面公钥签名算法的 NID。例如，如果 EVP_sha1 是使用 RSA 签名算法，那么就会返回 NID_sha1WithRSAEncryption。

【EVP_md2、EVP_md5、EVP_sha、EVP_sha1、EVP_mdc2 和 EVP_ripemd160】

这些函数返回相应名字的 EVP_MD 结构，它们都使用 RSA 算法作为签名算法。在新的程序里，一般推荐使用 sha1 算法。

【EVP_dss 和 EVP_dss1】

这两个函数返回的 EVP_MD 结构分别使用 sha 和 sha1 信息摘要算法，但是签名算法使用 DSS（DSA）。

【EVP_md_null】

该函数返回的信息摘要结构不作任何事情，返回的摘要信息长度为 0。

【EVP_get_digestbyname、EVP_get_digestbynid 和 EVP_get_digestbyobj】

这三个函数分别根据给定的算法名称、算法 NID 以及 ASN1_OBJECT 结构返回一个相应的 EVP_MD 算法结构。摘要算法在使用之前必须进行初始化，如使用 Openssl_add_all_digests 进行初始化。如果调用不成功，返回 NULL。

openssl 之 EVP 系列之 10---EVP_Sign 系列函数介绍

---根据 openssl doc\crypto\EVP_SignInit.pod 翻译

（作者： DragonKing, Mail: wzhah@263.net ，发布于： http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

EVP_Sign 系列函数使用的基础结构跟信息摘要算法使用的基础结构是一样的，而且，其前面的两个操作步骤初始化和数据操作（信息摘要）也跟信息摘要算法是一样的，唯一不一样的是最后一步操作，本系列函数做了签名的工作，而信息摘要系列函数当然就只是简单的处理完摘要信息了事了。其实这是很容易理解的事情，因为签名算法就是在信息摘要之后用私钥进行签名的过程。本系列函数定义的如下(openssl\evp.h)：

```
int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_SignUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_SignFinal(EVP_MD_CTX *ctx,unsigned char *sig,unsigned int *s, EVP_PKEY *pkey);
void EVP_SignInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_PKEY_size(EVP_PKEY *pkey);
```

【EVP_SignInit_ex】

该函数是一个宏定义函数，其实际定义如下：

#define EVP_SignInit_ex(a,b,c) EVP_DigestInit_ex(a,b,c)

可见，该函数跟前面叙述的 EVP_DigestInit_ex 的功能和使用方法是一样的，都是使用 ENGINE 参数 impl 所代表的实现函数功能来设置结构 ctx。在调用本函数前，参数 ctx 一定要经过 EVP_MD_CTX_init 函数初始化。详细使用方法参看前面的文章介绍。成功返回 1，失败返回 0。

【EVP_SignUpdate】

该函数也是一个宏定义函数，其实际定义如下：

#define EVP_SignUpdate(a,b,c) EVP_DigestUpdate(a,b,c)

该函数使用方法和功能也跟前面介绍的 EVP_DigestUpdate 函数一样，将一个 cnt 字节的数据经过信息摘要运算存储到结构 ctx 中，该函数可以在一个相同的 ctx 中调用多次来实现对更多数据的信息摘要工作。成功返回 1，失败返回 0。

【EVP_SignFinal】

该函数跟前面两个函数不同，这是签名系列函数跟信息摘要函数开始不同的地方，其实，该函数是将签名操作的信息摘要结构 ctx 拷贝一份，然后调用 EVP_DigestFinal_ex 完成信息摘要工作，然后开始对摘要信息用私钥 pkey 进行签名,并将签名信息保存在参数 sig 里面。如果参数 s 不为 NULL，那么就会将签名信息数据的长度（单位字节）保存在该参数中，通常写入的数据是 EVP_PKEY_size(key)。

因为操作的时候是拷贝了一份 ctx，所以，原来的 ctx 结构还可以继续使用 EVP_SignUpdate 和 EVP_SignFinal 函数来完成更多信息的签名工作。不过，最后一定要使用 EVP_MD_CTX_cleanup 函数清除和释放 ctx 结构，否则就会造成内存泄漏。

此外，当使用 DSA 私钥签名的时候，一定要对产生的随机数进行种子播种工作（seeded)，否则操作就会失败。RSA 算法则不一定需要这样做。至于使用的签名算法跟摘要算法的关系，在 EVP_Digest 系列中已经有详细说明，这里不再重复。

本函数操作成功返回 1，否则返回 0。

【EVP_SignInit】

本函数也是一个宏定义函数，其定义如下：

#define EVP_SignInit(a,b) EVP_DigestInit(a,b)

所以其功能和用法跟前面介绍的 EVP_DigestInit 函数完全一样，使用缺省实现的算法初始化算法结构 ctx。

【EVP_PKEY_size】

本函数返回一个签名信息的最大长度（单位字节）。实际签名信息的长度则由上述的函数 EVP_SignFinal 返回，有可能比这小。

上述所有函数发生错误，可以使用 ERR_get_error 函数获得错误码。

openssl 之 EVP 系列之 11---EVP_Verify 系列函数介绍

---根据 openssl doc\crypto\EVP_VerifyInit.pod 翻译和自己的理解写成

（作者:DragonKing, Mail: wzhah@263.net , 发布于:http://openssl.126.com 之 openssl 专业论坛,版本: openssl-0.9.7）

跟 EVP_Sign 系列函数一样，EVP_Verify 系列函数的前两步（初始化和信息摘要处理）跟信息摘要算法是一样的，因为签名验证的过程就是先对信息进行信息摘要，然后再将发来的摘要信息解密后进行比较的过程，其定义如下（openssl\evp.h）：

```
int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_VerifyUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_VerifyFinal(EVP_MD_CTX *ctx,unsigned char *sigbuf, unsigned int siglen,EVP_PKEY *pkey);
int EVP_VerifyInit(EVP_MD_CTX *ctx, const EVP_MD *type);
```

【EVP_VerifyInit_ex】

该函数是一个宏定义函数，其实际定义如下：

#define EVP_VerifyInit_ex(a,b,c) EVP_DigestInit_ex(a,b,c)

所以，其功能和使用方法跟前面介绍的 EVP_DigestInit_ex 函数是一样的。该函数使用参数 impl 所提供的算法库对验证结构 ctx 进行设置。在调用本函数之前，参数 ctx 必须经过调用 EVP_MD_CTX_init 进行初始化。成功返回 1，失败返回 0。

【EVP_VerifyUpdate】

该函数也是一个宏定义函数，其实际定义如下：

#define EVP_VerifyUpdate(a,b,c) EVP_DigestUpdate(a,b,c)

所以，其功能和使用方法跟前面介绍的 EVP_DigestUpdate 函数是相同的。该函数将参数 d 中的 cnt 字节数据经过信息摘要计算后保存到 ctx 中，该函数可以进行多次调用，以处理更多的数据。成功调用返回 1，失败返回 0。

【EVP_VerifyFinal】

该函数使用公钥 pkey 和 ctx 结构里面的信息验证 sigbuf 里面的数据的签名。事实上，该函数先调用 EVP_MD_CTX_copy_ex 函数将原来的 ctx 拷贝一份，然后调用 EVP_DigestFinal_ex 函数完成拷贝的 ctx 的信息摘要计算，最后才使用公钥进行签名的验证工作。

因为该函数实际上处理的是原来 ctx 函数的一个拷贝，所以原来的 ctx 结构还可以调用 EVP_VerifyUpdate 和 EVP_VerifyFinal 函数进行更多的数据处理和签名验证工作。

在使用完之后，ctx 必须使用 EVP_MD_CTX_cleanup 函数释放内存，否则就会导致内存泄漏。

此外，至于信息摘要算法和签名算法的关联的关系，请参照信息摘要算法部分的说明。

该函数调用成功返回 1，失败则返回 0 或－1。

【EVP_VerifyInit】

该函数使用缺省的实现算法对 ctx 结构进行初始化。也是一个宏定义函数，其定义如下：

#define EVP_VerifyInit(a,b) EVP_DigestInit(a,b)

所以跟 EVP_DigestInit 函数功能和用法是一样的。

openssl 之 EVP 系列之 12---EVP_Seal 系列函数介绍

---根据 openssl doc\crypto\EVP_SealInit.pod 翻译和自己的理解写成

（作者： DragonKing， Mail: wzhah@263.net ，发布于： http://openssl.126.com 之 openssl 专业论坛,版本: openssl-0.9.7）

改系列函数是相当于完成一个电子信封的功能，它产生一个随机密钥，然后使用一个公钥对改密钥进行封装，数据可以使用随机密钥进行加密。

信封加密在进行大量数据传输的时候是必须经常要用到的，因为公开密钥算法的加解密速度很慢，但对称算法就快多了。所以一般用公开密钥算法进行加密密钥的传输，而真正进行数据加密则使用对称加密算法。

其定义的函数如下（openssl\evp.h）：

```
int EVP_SealInit(EVP_CIPHER_CTX *ctx, EVP_CIPHER *type, unsigned char **ek,
```

int *ekl, unsigned char *iv,EVP_PKEY **pubk, int npubk);
int EVP_SealUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_SealFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);

【EVP_SealInit】

该函数初始化一个加密算法结构 EVP_CIPHER_CTX，采用了指定的加密算法，使用一个随机密钥和初始化向量 IV。事实上，该函数调用 EVP_EncryptInit_ex 函数两次完成了 ctx 结构的初始化工作。参数 type 是算法类型，跟签名介绍过的是一样的，为 EVP_des_cbc 类型的函数。随机私钥被一个或多个公钥加密，这就允许秘钥被公钥相应的私钥解密。参数 ek 是一个缓存序列，可以存放多个被公钥加密后的密钥的信息，所以每个缓存空间都应该足够大，比如 ek[i] 的缓存空间就必须为 EVP_PKEY_size(pubk[i])那么大。每个被加密密钥的长度保存在数字 ekl 中。参数 pubk 是一个公钥陈列，可以包含多个公钥。函数成功执行返回 npubk，失败返回 0。

因为该函数的密钥是随机产生的，随意在调用该函数之前，必须对随机数播种（seeded）。

使用的公钥必须是 RSA，因为在 openssl 里面这是唯一支持密钥传输的公钥算法。

跟 EVP_EncryptInit 函数一样，本函数也可以分为两次调用，第一次调用的时候要将参数 npubk 设为 0，第二调用的时候就应该将参数 type 设为 NULL。

【EVP_SealUpdate】

该函数是一个宏定义函数，其实际定义如下：

#define EVP_SealUpdate(a,b,c,d,e) EVP_EncryptUpdate(a,b,c,d,e)

由此可见，其完成的功能和使用方法跟 EVP_EncryptUpdate 函数是一样的。细节参看前面介绍的文章。成功执行返回 1，否则返回 0。

【EVP_SealFinal】

该函数简单调用了 EVP_EncryptFinal_ex 完成其功能，所以其完成的功能和使用参数也跟 EVP_EncryptFinal_ex 函数一样，细节请参考相关文章。唯一不一样的是，该函数还调用 EVP_EncryptInit_ex(ctx,NULL,NULL,NULL,NULL)函数对 ctx 结构再次进行了初始化。成功返回 1，否则返回 0。

openssl 之 EVP 系列之 13---EVP_Open 系列函数介绍

---根据 openssl doc\crypto\EVP_OpenInit.pod 翻译和自己的理解写成

（作者：DragonKing，Mail：wzhah@263.net，发布于：http://openssl.cn 之 openssl 专业论坛,版本：openssl-0.9.7）

本系列函数相对于 EVP_Seal 系列函数，是进行信封加密的。它将公钥加密了的密钥加密出来，然后进行数据的解密。其定义的函数如下（openssl\evp.h）：

int EVP_OpenInit(EVP_CIPHER_CTX *ctx,EVP_CIPHER *type,unsigned char *ek,
int ekl,unsigned char *iv,EVP_PKEY *priv);
int EVP_OpenUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl, unsigned char *in, int inl);
int EVP_OpenFinal(EVP_CIPHER_CTX *ctx, unsigned char *out,
int *outl);

【EVP_OpenInit】

该函数初始化一个用来加密数据的 ctx 结构。它使用参数 priv 的私钥解密参数 ek 里面长度为 ekl 字节的加密密钥。参数 iv 是初始化向量。如果参数 type 设定的加密算法长度是可变的，那么密钥长度就会被设置为解密得到的密钥的长度；如果加密算法长度是固定的，那么得到的解密密钥的长度就必须跟固定算法长度相同才行。成功执行返回密钥的长度，否则返回 0。

跟函数 EVP_DecryptInit 一样，该函数也可以分成多次调用，首次调用应该将参数 priv 设置为 NULL，再次调用的时候应该将 type 设置为 NULL。

【EVP_OpenUpdate】

该函数是一个宏定义函数，其实际定义如下：

#define EVP_OpenUpdate(a,b,c,d,e) EVP_DecryptUpdate(a,b,c,d,e)

所以，其功能和使用方法跟前面介绍过的 EVP_DecryptUpdate 相同，请参考相应的文章。

成功执行返回 1，否则返回 0。

【EVP_OpenFinal】

事实上，该函数调用 EVP_DecryptFinal_ex 完成了其功能，所以其使用方法跟功能跟函数 EVP_DecryptFinal_ex 是一样的，参考该函数说明就可以。唯一不同的是，本函数还调用 EVP_DecryptInit_ex(ctx,NULL,NULL,NULL,NULL)再次进行了初始化工作。成功执行返回 1，否则返回 0。

作者 Blog：http://blog.csdn.net/gdwzh/

openssl 之 EVP 系列之 14---EVP_Encode 系列函数介绍

根据自己的理解写成（作者：DragonKing，Mail：wzhah@263.net，发布于：http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

该系列函数主要对数据进行 BASE64 编码,为此,它定义了一个简单的 EVP_ENCODE_CTX 结构类型，由于该结构比较简单，主要包括了长度信息以及数据缓存，大家可以参考 evp.h，这里不再做介绍。

EVP_Encode 系列函数定义如下（openssl\evp.h）：

void EVP_EncodeInit(EVP_ENCODE_CTX *ctx);

void EVP_EncodeUpdate(EVP_ENCODE_CTX *ctx,unsigned char *out,int *outl,unsigned char *in,int inl);

void EVP_EncodeFinal(EVP_ENCODE_CTX *ctx,unsigned char *out,int *outl);

int EVP_EncodeBlock(unsigned char *t, const unsigned char *f, int n);

【EVP_EncodeInit】该函数初始化一个用来进行 base64 编码的结构，事实上，该函数只是简单设置了结构里面几个常量的长度。

【EVP_EncodeUpdate】该函数将参数 in 里面的 inl 自己数据拷贝到结构体 ctx 里面，如果结构体里面有数据，就同时将结构体里面的数据进行 BASE64 编码并输出到参数 out 指向的缓存里面，输出数据的长度保存在 outl 里面。注意，在第一次调用本函数的时候，虽然往结构体里面拷贝数据了，但是结构体 ctx 里面开始是没有输入数据存在并且输入数据长度不超出 ctx 内部存储数据的最长限制，就不会有任何数据被进行 BASE64 编码，也就是说，不会有任何数据输出；但是如果输入数据长度比内部存储的数据长，那么就会输出部分经过 BASE64 编码的数据。数据输出总是在下一层输入前完成的。

【EVP_EncodeFinal】该函数将结构体 ctx 里面剩余数据进行 BASE64 编码并写入到参数 out 里面去，输出数据的长度保存在 outl 里面。

【EVP_EncodeBlock】该函数将参数 f 里面的字符串里面的 n 个字节的字符串进行 BASE64 编码并输出到参数 t 里面。返回数据的字节长度。事实上，在函数 EVP_EncodeUpdate 和 EVP_EncodeFinal 里面就调用了该函数完成 BASE64 编码功能。

openssl 之 EVP 系列之 15---EVP_Decode 系列函数介绍

---根据自己的理解写成

（作者:DragonKing, Mail: wzhah@263.net , 发布于:http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

本系列函数与 EVP_Encode 系列函数相对，对数据进行 BASE64 解码。其定义的函数如下（openssl\evp.h）：

```
 void EVP_DecodeInit(EVP_ENCODE_CTX *ctx);
 int EVP_DecodeUpdate(EVP_ENCODE_CTX *ctx,unsigned char *out,int *outl,u
nsigned char *in, int inl);
 int EVP_DecodeFinal(EVP_ENCODE_CTX *ctx, unsigned char *out, int *outl)
;
 int EVP_DecodeBlock(unsigned char *t, const unsigned char *f, int n);
```

【EVP_DecodeInit】

该函数初始化一个用来进行 BASE64 解码的数据结构。

【EVP_DecodeUpdate】

该函数将参数 in 里面 inl 字节的数据拷贝到结构体 ctx 里面。如果结构体里面已经有数据，那么这些数据就会先进行 BASE64 解码，然后输出到参数 out 指向的内存中，输出的字节数保

存在参数 outl 里面。输入数据为满行的数据时，返回为 1；如果输入数据是最后一行数据的时候，返回 0；返回－1 则表明出错了。

【EVP_DecodeFinal】

该函数将结构体 ctx 里面剩余的数据进行 BASE64 解码并输出到参数 out 指向的内存中，输出数据长度为 outl 字节。成功返回 1，否则返回－1。

【EVP_DecodeBlock】

该函数将字符串 f 中的 n 字节数据进行 BASE64 解码，并输出到 out 指向的内存中，输出数据长度为 outl。成功返回解码的数据长度，返回返回－1。

openssl 之 EVP 系列之 16---EVP_PKEY 系列函数详解(-)

---根据自己的理解写成

（作者:DragonKing, Mail: wzhah@263.net , 发布于:http://openssl.126.com 之 openssl 专业论坛, 版本: openssl-0.9.7）

EVP_PKEY 系列函数定义了一个密钥管理的结构体，其定义如下（openssl\evp.h）：

```
typedef struct evp_pkey_st
{
 int type;
 int save_type;
 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
 } pkey;
 int save_parameters;
 STACK_OF(X509_ATTRIBUTE) *attributes; /* [ 0 ] */
}EVP_PKEY;
```

该密钥结构既可以用来存储公钥，也可以用来存储私钥。该结构其实是一个在 openssl 中很通用的结构，在许多算法中使用了该结构作为存储体。基于该结构定义的 EVP_PKEY 系列函数有如下几个常用的：

```
EVP_PKEY * EVP_PKEY_new(void);
void EVP_PKEY_free(EVP_PKEY *pkey);
int EVP_PKEY_type(int type);
int EVP_PKEY_bits(EVP_PKEY *pkey);
int EVP_PKEY_size(EVP_PKEY *pkey);
```

这些函数的实现主要在文件 evp\p_lib.c 里面，大家如果要看源代码，就可以看该文件了。下面对这些函数分别进行简单的介绍。

【EVP_PKEY_new】

本函数生成并返回一个 EVP_PKEY 结构体，并对该结构体元素进行初始化，如果成功调用返回该结构体的指针，否则返回 NULL。因为该函数返回的只是一个空结构体，所以还要使用下面的一些相关函数进行设置，如 EVP_PKEY_set1_RSA 之类的函数。

【EVP_PKEY_free】

该函数用来释放 EVP_PKEY 结构里面指针指向的内存空间。

【EVP_PKEY_type】

该函数返回输入结构密钥的类型，目前之支持 RSA（EVP_PKEY_RSA）、DSA（EVP_PKEY_DSA）以及 DH（EVP_PKEY_DH）类型的密钥，如果为其它类型，则返回 NID_undef。典型使用方式是 EVP_PKEY_type(pkey->type)。

【EVP_PKEY_size】

该函数返回 EVP_PKEY 结构中密钥的字节长度，只对 RSA 和 DSA 类型的密钥有效，如果输入的参数为 NULL 或其它密钥类型，则返回 0。

【EVP_PKEY_bits】

该函数返回 EVP_PKEY 结构中密钥的比特长度，只对 RSA 和 DSA 类型的密钥有效，如果输入的参数没有初始化或其它密钥类型，则返回 0。

openssl 之 EVP 系列之 17---EVP_PKEY 系列函数详解(二)

---根据自己的理解写成

（作者:DragonKing, Mail: wzhah@263.net , 发布于:http://openssl.126.com 之 openssl 专业论坛, 版本: openssl-0.9.7）

接着前面介绍的 EVP_PKEY 系列函数，我们这章介绍的函数如下（openssl\evp.h）：

```
int EVP_PKEY_assign(EVP_PKEY *pkey,int type,char *key);
int EVP_PKEY_assign_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_assign_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_assign_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_set1_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_set1_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_set1_DH(EVP_PKEY *pkey,DH *key);
RSA *EVP_PKEY_get1_RSA(EVP_PKEY *pkey);
DSA *EVP_PKEY_get1_DSA(EVP_PKEY *pkey);
DH *EVP_PKEY_get1_DH(EVP_PKEY *pkey);
EC_KEY *EVP_PKEY_get1_EC_KEY(EVP_PKEY *pkey); int EVP_PKEY_copy_parameters (EVP_PKEY *to,EVP_PKEY *from);
int EVP_PKEY_missing_parameters(EVP_PKEY *pkey);
int EVP_PKEY_save_parameters(EVP_PKEY *pkey,int mode);
int EVP_PKEY_cmp_parameters(EVP_PKEY *a,EVP_PKEY *b);
int EVP_PKEY_decrypt(unsigned char *dec_key,unsigned char *enc_key,int
enc_key_len,EVP_PKEY *private_key);
int EVP_PKEY_encrypt(unsigned char *enc_key,unsigned char *key,int key_
len,EVP_PKEY *pub_key);
```

【EVP_PKEY_assign】

该函数通过类型参数 type 和密钥值 key 给初始化了的 EVP_PKEY 结构 pkey 设置相应的参数，参数 key 的内存是直接跟 pkey 结构相关在一起的，所以 key 的内存在 pkey 使用期间不能改变，pkey 被释放的时候，key 也会被自动释放。当 pkey 不为 NULL 的时候，返回 1，否则返回 0。

【EVP_PKEY_assign_RSA、EVP_PKEY_assign_DSA 和 EVP_PKEY_assign_DH】

这三个函数都是上面的函数 EVP_PKEY_assign 函数的宏定义函数，分别完成 RSA、DSA 以及 DH 类型密钥的 pkey 结构初始工作。当 pkey 不为 NULL 的时候，返回 1，否则返回 0。

【EVP_PKEY_set1_RSA、EVP_PKEY_set1_DSA 和 EVP_PKEY_set1_DH】

这三个函数跟上面函数的功能是一样的，只不过用这些函数设置的 pkey 释放的时候，key 内存不会被释放。openssl 其实在内部做了一中类似于 COM 接口的引用技术，计算 key 被引用的次数，从而实现了虽然公用内存，但是却能保持内存不被释放的功能。当 pkey 不为 NULL 的时候，返回 1，否则返回 0。

【EVP_PKEY_get1_RSA、EVP_PKEY_get1_DSA 和 EVP_PKEY_get1_DH】

这三个函数返回 pkey 结构相应类型的密钥指针，如果 pkey 结构里面的密钥类型跟函数类型不一致，则返回 NULL。

【EVP_PKEY_copy_parameters】

该函数将参数 from 结构体的参数拷贝到参数 to 结构体中，如果两个结构体的密钥类型不

一样，则不会拷贝任何东西，产生错误提示，返回 0；如果 from 中没有参数，则产生错误提示返回 0。成功执行返回 1。该函数只对 DSA 类型的密钥有效，其它类型密钥不会执行任何操作。

【EVP_PKEY_missing_parameters】

如果结构 pkey 中的参数 p、q 或 g 中有一个没有设置，那么该函数就会返回 1，否则就返回 0。本函数只对 DSA 类型密钥有效。其它类型密钥都返回 0。

【EVP_PKEY_save_parameters】

该函数也只对 DSA 类型密钥有效，如果密钥为 DSA 类型，那么参数 pkey->save_param eters 就会被设置为参数 mode 的值。如果要重用该密钥的参数，mode 应该设置为 1。唯一用到本函数功能的好象就是 X509_PUBKEY_set 函数。成功调用返回 pkey 参数原来的保存标志，其它类型密钥则返回 0。

【EVP_PKEY_cmp_parameters】

该函数也仅仅对 DSA 类型密钥有效，如果两个密钥的参数 p、q 和 g 都相同，则返回 1，否则返回 0。如果两个密钥之中有一个或两个都不为 DSA 密钥，则返回－1。

【EVP_PKEY_encrypt】

该函数使用公钥 pubkey 将密钥 key 加密并保存在 enc_key 里面，返回输出值的长度。

enc_key 分配的内存必须保证能够容下输出的数据。key 参数的长度又参数 key_len 指定。

本函数在 EVP_SealInit 中被调用了，并且在 PEM_SealInit 中也被间接调用了。如果公钥类型不是 RSA，那么本函数返回－1。

【EVP_PKEY_encrypt】

跟上述函数相对应，该函数将参数 ek 里面的加密密钥用私钥 priv 解密并输出到参数 key 中，返回输出结果的长度。key 的长度也必须足够大，能够容下输出结果。ek 的长度由参数 ekl 指定。如果私钥不是 RSA 类型，那么本函数返回－1。

【说明】

此外，ASN1 还提供了下列 4 个函数，在这里不再介绍，等到介绍 ASN1 函数再介绍这些函数。

```
EVP_PKEY * d2i_PublicKey(int type,EVP_PKEY **a, unsigned char **pp,long length);
int i2d_PublicKey(EVP_PKEY *a, unsigned char **pp);
EVP_PKEY * d2i_PrivateKey(int type,EVP_PKEY **a, unsigned char **pp,long length);
EVP_PKEY * d2i_AutoPrivateKey(EVP_PKEY **a, unsigned char **pp, long length);
int i2d_PrivateKey(EVP_PKEY *a, unsigned char **pp);
```

openssl 之 EVP 系列之 18---EVP_BytesToKey 函数介绍

---根据 doc\crypto\EVP_BytesToKey.pod 翻译和自己的理解写成

（作者：DragonKing, Mail: wzhah@263.net ,发布于：http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

该函数定义如下（openssl\evp.h）：

```
int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md,const unsign ed char
*salt, const unsigned char *data,int datal, int count, unsigned char  *key,unsigned char *iv);
```

该函数实现的功能是通过密码口令产生一个加密密钥和初始化向量 IV。参数 types 是要使用该密钥和 IV 的加密算法，类型位 EVP_CIPHER。参数 md 是要使用的信息摘要算法。

参数 salt 是在提取密钥和 IV 的时候要使用的盐值，它应该指向一个 8 字节的缓存或者 NULL，如果是 NULL，就不使用盐值。参数 data 是包含了 datal 个字节数据的缓存，这些数据用来提取密钥数据。参数 count 指定了迭代的次数。提取出来的密钥和 IV 向量分别保存在参数 key 和 iv 里面。函数返回生成的密钥的长度。

本函数使用的提取密钥算法是将产生的 D_1,D_2,...,D_i,...等数据串连起来，一直计算到数据长度满足了密钥和 IV 的长度。其中 D_i 定义如下：

```
D_i = HASH^count(D_(i-1) || data || salt)
```

这里的符号||表示数据串连，也就是连接起来。初始的 D_0 是空的，也就是说长度为 0。HASH 是代表使用的信息摘要算法，HASH^1(data) 就是 HASH(data)，HASH^2(data) 是 HASH(HASH(data)),其它以此类推。串连出来的数据前面的用来做密钥，后面的用来做 IV 值。

该函数一个典型的应用就是根据参数 data 中的口令来为一个加密算法产生加密密钥。增加参数 count 会使算法的执行速度下降，但是却使得使用大量备选密码强力破解的黑客行为难度大大增加。

如果密钥和 IV 的长度比摘要算法的长度短并且使用了 MD5 算法，那么该函数使用的算法就跟 PKCS#5 v1.5 标志是兼容的。否则，就会使用非标准的扩展方法来产生其它数据。

新的应用程序建议应该使用更标准的算法如 PKCS#5 v2.0 来提取加密密钥。

openssl 之 EVP 系列之 19---EVP 提供对口令管理的函数介绍

---根据自己的理解写成

（作者：DragonKing, Mail: wzhah@263.net ,发布于：http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

EVP 提供的对口令管理的函数有三个，定义如下（openssl\evp.h）：

```
int EVP_read_pw_string(char *buf,int length,const char *prompt,int verify);
void EVP_set_pw_prompt(char *prompt);
char * EVP_get_pw_prompt(void);
```

【EVP_read_pw_string】

该函数从标准输入终端读入一个口令数据，其中，读入的参数保存在 buf 里面，参数的长度保存在 length 里面。prompt 参数里面保存的是输出到显示终端的提示语句；如果 verify 参数设置为 1，那么口令就会被要求输入两次，如果两次输入相同，才会成功返回 0，否则就会返回错误 1，返回－1 表示发生了系统错误。

值得注意的是，由于历史原因，在 openssl 里面，该函数的核心代码是在 DES 算法里面实现的，所以如果将 DES 算法禁止了，该函数的调用也会失败。

【EVP_set_pw_prompt】

该函数设置在调用 EVP_read_pw_string 函数的时候缺省的 prompt 参数的值，也就是说如果调用 EVP_read_pw_string 函数的时候 prompt 参数为 NULL，那就会使用本函数设置的缺省的值。需要注意的是，本函数设置的缺省值其实是保存在一个静态的数组变量里面，所以如果在 Win16 平台使用的时候，可能会出现莫名其妙的现象；在使用多线程程序的时候也必须小心。

【EVP_get_pw_prompt】

该函数返回一个指向缺省 prompt 字符串（一个静态变量）的指针，如果该参数没有设置，那么就返回 NULL。

openssl 之 EVP 系列之 20---结束语

（作者：DragonKing, Mail: wzhah@263.net ,发布于：http://openssl.126.com 之 openssl 专业论坛,版本：openssl-0.9.7）

openssl 提供的 EVP 系列函数，是对 openssl 所有各种加密算法做了一个高层的封装，包括了以下几种：

1.对称加密算法(EVP_Encrypt)
2.信息摘要（HASH）算法(EVP_Digest)
3.签名算法(EVP_Sign 和 EVP_Verify)
4.公开密钥算法(EVP_Seal 和 EVP_Open)
5.BASE64 编码算法(EVP_Encode 和 EVP_Decode)

此外，为了提供更加完善的功能，以便实际应用的需要，openssl 的 EVP 系列还提供了以下几种辅助功能:

1.不同类型密钥结构的管理(EVP_PKEY)
2.加密密钥和 IV 值提取功能(EVP_BytesToKey)
3.口令获取和管理功能

如果能够对 EVP 系列函数有一个透彻的了解，那么我们基本上就能的心应手地使用 openssl 提供的加密函数的功能了。

EVP 系列的介绍已经完成，其中肯定有不少错误和不清楚的地方，希望大家指针。