# OpenSSL 与网络信息安全
# ——基础、结构和指令

王志海　童新海　沈寒辉　编著

清 华 大 学 出 版 社
北京交通大学出版社
· 北京 ·

## 内 容 简 介

本书结合 OpenSSL 的结构和应用指令，对密码算法、公钥基础设施、数字证书和密码应用协议等内容进行了全面的具体阐述。本书试图通过对 OpenSSL 的具体介绍，一方面让读者能够熟悉和掌握 OpenSSL 这个强大的工具库，另一方面更希望读者能够在实践中深入理解密码学理论、思想及其相关应用的实质。

本书共分 12 章。第 1 章对密码学理论、密码学相关应用和本书的情况作了一个概要的说明；第 2～3 章主要介绍密码学的基本概念、密码技术的基本实现、对称加密算法、公开密钥算法和单向散列函数算法等密码学知识；第 4～6 章介绍 OpenSSL 的结构、编译和安装方法及其使用的基本概念；第 7～12 章是本书的重点，详细介绍了 OpenSSL 的应用程序（指令）的使用方法和各项参数的意义。

本书可以作为密码技术设计研发人员的参考书籍，在校的本科学生、研究生入门级的密码学和信息安全技术方面的参考书籍，还可以作为信息安全培训和密码技术培训教材。



图书在版编目（**CIP**）数据

OpenSSL 与网络信息安全：基础、结构和指令/王志海，童新海，沈寒辉编著. —北京：清华大学出版社；北京交通大学出版社，2007.4
ISBN 978－7－81123－006－2

Ⅰ. O…　Ⅱ. ①王…　②童…　③沈…　Ⅲ. 计算机网络-安全技术　Ⅳ. TP393.08

中国版本图书馆 CIP 数据核字（2007）第 051086 号

责任编辑：刘　洵
出版发行：清 华 大 学 出 版 社　邮编：100084　电话：010－62776969　http：//www.tup.com.cn
　　北京交通大学出版社　邮编：100044　电话：010－51686414　http：//press.bjtu.edu.cn
印 刷 者：北京瑞达方舟印务有限公司
经　　销：全国新华书店
开　　本：185×260　印张：16.25　字数：362 千字
版　　次：2007 年 4 月第 1 版　2007 年 4 月第 1 次印刷
书　　号：ISBN 978－7－81123－006－2/TP・345
印　　数：1～4 000 册　定价：26.00 元

# 前　言

近几年来，密码学得到了越来越广泛的应用和关注，例如新出现的内网安全、SSL VPN、网上银行安全和数据保密等新安全热点领域，都离不开密码技术。密码技术为什么能够在短短几年内得到如此强烈的关注，究其背景，主要在两个方面：一是随着信息化程度的提高，对信息安全的控制越来越细致，甚至已经深入到文件级的数据单元，简单的安全控制措施已经不能满足要求，而密码技术由于其先天的完整理论体系备受青睐；二是由于近年来国内外接触密码技术的研发设计人员越来越多，密码技术的应用有了一定的群众基础。

在中国乃至世界密码技术的推广应用中，OpenSSL 的功劳是不可忽视的，目前涉及密码技术的产品，相当大一部分直接或者部分引用了 OpenSSL 的类库或者例程。对于密码技术的研发设计人员，没有听说过 OpenSSL 的更是少之又少。目前国内外和密码技术、公钥基础设施（PKI）相关的资料，大部分偏重理论、数学算法和协议介绍，与实际应用尚有一段差距，不利于普通研发设计人员对密码技术的理解和应用。鉴于此，笔者自 2002 年创办 www. OpenSSL. cn 以来，便一直试图结合 OpenSSL 本身丰富的内容和应用，对密码技术进行理论结合实际的疏理，试图编写一本简单易懂的密码技术入门资料。

《OpenSSL 与网络信息安全——基础、结构和指令》一书从分析介绍 OpenSSL 本身的体系结构、设计思想和应用程序（指令）入手，让读者逐步领会密码学、公钥基础设施（PKI）及相关应用协议等方面的理论内涵、设计思想、应用场景及实现方法等，能够将编程开发工作和密码学理论联系起来，从而融会贯通。阅读本书，要想取得上述的效果，需要从两方面着手：一是提前阅读密码学理论和公钥基础设施方面的资料；二是阅读本书的时候，一定要结合 OpenSSL 的实际环境，多动手。

本书完成历经近四年，非常缓慢，除了个人事务繁多的因素外，更主要的是由于其中很多细节之处，都需要试验和编程来进行仔细的考究。在本书的编写过程中，虽然笔者力求做到仔细，但是错误肯定还是难以避免，希望同行们在阅读的过程中发现后能够及时在 www. OpenSSL. cn 网站上更正，假若能告知我们，那就更不胜感谢！

王志海

2007. 4

# 目　录

# 第 1 章 概 述

## 1.1 信 息 安 全

信息安全是本书要解决的主要问题，本节将介绍信息安全的基本概念和本书将要涉及的信息安全的内容。

### 1.1.1 信息安全概念

信息安全是自古以来就存在的概念，比如以前为了保证传递书信的保密性，使用腊封或其他方式将书信封装在信封内；还有使用暗号口令确认接受信息的人的身份等方法。需要注意的是，信息安全技术是跟信息的载体形式和传送媒介密切相关的，信息载体的变化和信息传送媒介的变化必然会导致信息安全技术的变化发展。

在过去的二十多年中，信息技术取得了令人惊异的发展，越来越多的有价值的信息和资料以数字信息存放在计算机等数字信息存储设备中。与此同时，信息共享技术也获得了巨大的突破，以 Internet 的发展为代表，短短的时间内，从美国军方的一个专用网络发展到联系着全世界千千万万人的庞大信息网络。这些客观的变化导致对信息安全的要求发生了重大的变化。

随着信息数字化及计算机应用的发展，对存储在计算机中的文件和其他数字信息的保护需求成为了一种必然，尤其对一个能够通过公共网络进入的共享系统来说，这种需求显得尤为迫切。针对这种需求，目前发展起来的技术有防病毒技术和防火墙技术，等等。有些文献将这些保护数据、阻挡非法数据访问的技术统称为计算机安全技术或系统安全技术。

信息安全技术的另外一个重要变化是由网络和通信设施的产生和应用引起的。这些网络和通信设施用于在用户的各种终端及计算机之间传输数据信息，这种传输过程很容易受到非法窃听等攻击，这就需要对在网络中传输的数据采取安全的保护措施。针对这种需求发展起来的技术有 VPN、SSL 等。有些文献将这种类型的技术统称为网络安全技术。

事实上，因为现在的绝大部分数据终端设备（包括计算机）基本上都是跟网络相联的，所以无论是计算机安全、系统安全还是网络安全，都不是完全相互独立的。而且，这些名词由于其笼统性，反而有概念不清和误导的可能。所以，本书更愿意用具体一点的技术名词来说明不同的信息安全技术。

本书关注的是使用基于密码学原理来进行数据信息保护的技术，尤其偏向于利用该技

术保护在网络中传输的数据。对于防火墙、防病毒及入侵检测（IDS）等技术涉及的安全问题和解决方案，本书不作介绍。在本书后面的章节中，如果没有特别指明，信息安全的范围仅仅包括使用基于密码学原理来进行数据信息保护的安全技术。

### 1.1.2 信息安全内容

正如 Bruce Schneier[2] 所说，安全问题就如一条链子一样，必须保证每一个环节的安全才能达到使整个链子具有安全性。所以，在解决任何一个实际的或抽象的系统的安全问题之前，都应该首先分析其可能存在的安全缺陷，进而采取相应的安全措施。为了使读者了解本书涉及的领域和需要解决的问题，下面介绍一下与本书内容相关的安全问题，并举一些相关的安全漏洞的例子，加深读者的理解。

**(1)** 机密性

机密性是指保护信息免受主动的非法窃取、阅读等攻击。在信息数字化之前，信息的机密性是依靠严格的管理制度和强大的物理手段来实现的，如戒备森严的房子和难以破坏的保险箱。对于独立的设备（没有联网的计算机）中的数字信息，当然也可以依赖传统的保密手段，但是对于一般的共享系统或联网的系统来说，传统的方法就显得难以胜任，必须采用新的针对数字信息的手段。

机密性涉及的内容是多方面的，主要包括内容的机密性和信息量的机密性。

内容的机密性是很容易理解的，就是确保数字信息的内容不被没有授权的人访问。内容的机密性既可以针对计算机中的一个重要文件，也可以针对网络中传输的一些数据。对于计算机中的一个文件内容的保护显得可能简单一点，最简单的处理方法是只需要采用足够强大的加密算法将文件的内容加密即可。对于网络上传输的信息的保护，需要考虑的情况就会多一点，其中之一就是可能需要考虑对数据做不同层次的保护。比如，对一般的计算机之间的通信，可能只是选择其中重要的数据信息进行保护；而对于机密性要求非常高的办公室之间的两台计算机，可能会对它们之间传输的所有数据都进行保护。

信息量的机密性源于网络传输中通信量分析技术的产生，但本书认为不仅仅限于网络通信量的分析，在本地的计算机系统中，一样存在类似的安全危险，本书将它们统称为信息量的机密性。采用通信量分析进行攻击要求攻击者能够在通信设施上监听到通信的源地址和目的地址、通信频度、通信的数据长度、通信的时长等特征。对于本地计算机系统中的普通加密保护的文件，一样可以通过获取文件的长度信息、修改时间来获得有用的信息。更进一步，对一些结构化的文件，可以通过分析其各个结构的长度信息等来获得更多有用的信息，比如对加密数据库中各个字段长度的分析就可能得到大量的有用信息。信息量的分析还可以针对计算机系统中运行的程序，比如对加密程序的攻击就已经成功破解了 RSA 私钥。信息量的机密性在以前远远没有得到重视，但在今后的时间里，随着其相关攻击手段和事件的增加，必然会得到更大的发展。

**(2)** 完整性

完整性是指确保信息没有被修改，也可以防止假冒的信息。对于一个计算机中存储的信息来说，完整性的概念就是确保信息在存储的过程中没有被非法进行修改或替换。

对于网络信息来说，情况就复杂多了，主要分成面向连接和无连接两种情况。对于面向连接的情况来说，完整性是针对信息流的服务，它需要确保接收到的信息和发送的信息

是一样的，没有被篡改、插入、重排序、重复或者延迟，同时也要确保通信结束后数据在网络上的销毁。所以，面向连接的完整性服务不仅仅可以确保消息没有被非法篡改，还可以在一定程度上防止拒绝服务攻击（DOS）。对于无连接的网络信息传输来说，完整性的内容跟计算机系统中的文件对象完整性含义是一样的，即保护信息没有被篡改或替换。

**(3)** 鉴别

鉴别是指确认访问者的身份或消息的来源，防止冒充他人的行为发生。对于计算机中存储的信息来说，鉴别的功能就是确保访问者的身份，如最简单的是使用口令和密码来确保访问者的身份，安全一点的解决方案是通过 USB Key、IC 卡或其他形式的令牌进行身份鉴别。

对于网络传输中的信息来说，鉴别所需要确认的对象有很多种，可能是消息传输操作的用户，也可能是特定的应用程序，也可能是特定的 IP 地址，有时候可能是这些特征的综合，本书统称这些为实体。也就是说，在网络传输信息的过程中，鉴别的功能首先是确保通信双方的两个实体都是可信的，都是它们所宣称的实体；其次，鉴别还要确保在信息传输的过程中防止第三方假冒两个合格方的任何一方来达到未授权接收信息或发送信息的目的。

**(4)** 抗抵赖

抗抵赖是指保证消息的制造者或发出者不能在事后否认他制作或发出的消息。对于计算机中存储的信息，抗抵赖的功能就是确保文件在被合法授权用户修改后该用户不能否认自己做过这样的修改。这在传统的书面文件中，可以使用手写签名来解决这个问题，相应地，对于数字信息，可以使用数字签名来达到相同的目的。

对于网络信息传输的过程而言，抗抵赖的功能要求接收方能够验证消息的发送方，同时要求发送方能够验证消息的接收方，并能够在发生争议后向第三方（比如法庭）证明消息的发送方或接收方。

**(5)** 攻击举例

为了进一步说明上述四种信息安全措施的必要性，下面举一些有针对性的攻击例子，加深读者的理解。

Susn 使用 Mail 通过 Internet 向 Tom 传送一个带有机密信息的文件，但是由于没有使用安全措施保护该传送的信息，与 Susn 在同一公司局域网内工作的怀有不良目的 Jim 通过使用 Sniffiner 监听了 Susn 发送 Mail 的整个过程，因为 Internet 上的信息是明文传送的，所以 Jim 成功获取了这个带有机密信息的文件。这是一个针对机密性攻击的例子。

Susn 是公司的总裁，她想给表现不错的 Eric 加 2 000 元工资，起草了一份电子文件，叫秘书 Anna 发给财务部。Anna 是 Jim 的女朋友，而 Jim 正好跟 Eric 在同一个部门，于是 Anna 就将电子文件的名字 Eric 改成了 Jim，然后发给财务部，后果可想而知。这是针对完整性攻击的例子。

计算机 Server 上存有公司重要的客户资料，只有公司的客户部部门经理 Tom 有权限访问这些重要资料。Jim 是该公司一个员工，他准备跳槽，为了获得更多的客户资料，他一直想访问 Server 上的客户资料。有一天 Jim 无意中获得了 Tom 的账号和密码（偷看的），然后就连接上 Server，告诉 Server 是 Tom 要访问客户资料，并按 Server 的要求输入了密码，顺利取得了想要的客户资料。这是针对鉴别攻击的例子。

Jim 在一个电子商务网站购买了一台笔记本电脑，并通过网络告诉银行从自己的账户

给电子商务网站转账。Jim 在取得笔记本电脑后，发现自己其实并不需要笔记本电脑，但是退货已经不可能，于是就打算抵赖，他跟银行说自己其实并没有购买笔记本，是别人冒充了他，银行要负责任。如果银行没有一套针对这种情况的措施和方法，情况就会很麻烦。这是针对抗抵赖攻击的一个例子。

上述只是举了一些攻击的简单例子，事实上，实际的情况可能复杂得多，涉及的攻击可能也是多方面的。这些攻击采取的措施都是基于密码学原理的，下面本书将要介绍密码学的基本知识。

## 1.2 密 码 学

本节将概要地介绍密码学的功能、内容和应用，使读者对密码学有一个初步的了解。

### 1.2.1 密码学作用

最原始的密码学的作用是进行信息保密，即解决前面所述的机密性问题。但现代发展起来的密码学的作用已经远远超出了这个范围，它可以用来解决包括机密性、完整性、鉴别及抗抵赖相关的各种难题，囊括了本书前面介绍的各种安全问题。

机密性问题是密码学最早关心的问题，也是核心的问题。目前针对机密性问题，密码学提出了各种算法，主要分为对称加密算法和公开密钥算法。不管是什么算法，都是为了在各种复杂和苛刻的条件下实现信息保密的功能。对称加密算法主要适用于通信双方已经共享了秘密的密钥的情况，而公开密钥算法则适用于通信双方没有共享的密钥的情况。目前常用的对称加密算法有 DES，3DES，AES，等等，常用的公开密钥算法有 RSA、DH 算法，等等。

完整性问题在现代密码学中也得到了较好的解决，主要是采用了散列函数和数字签名相结合的算法。散列函数是一种单向映射函数，是密码学中确保数据完整性的核心算法，目前常用的有 MD5，SHA 和 SHA1 算法，等等。

鉴别问题和抗抵赖问题在密码学中的解决不仅仅依赖密码算法本身，还依赖一套严格执行的密码协议或网络协议。这些协议以密码算法为基础，达到了鉴别和抗抵赖的功能。虽然设计一个好的密码协议并非如想像的那么容易，但目前密码协议还是种类繁多，我们熟悉的有 Kerberos，SKID 等，网络协议有 SSL，SET 等。这些协议基本上都具备了鉴别和抗抵赖的功能。

### 1.2.2 密码学内容

目前现代密码学基本可以分为两大部分：密码算法和密码协议。密码协议是在密码算法的基础上实现的，但其完成的功能比单一的密码算法所能完成的功能丰富得多。

密码算法根据其完成的功能可以分为对称加密算法、公开密钥算法、数字签名算法及信息摘要（散列函数）算法。对称加密算法主要完成了明文数据到密文数据的转换功能，其加密密钥和解密密钥是相同的。公开密钥算法完成的功能跟对称加密算法是相同的，但是其加密密钥和解密密钥不相同。数字签名算法的功能是实现鉴别和抗抵赖的功能，其具体的实现有时候可以采用对称加密算法或公开密钥算法实现，也有专门设计用于数字签名

的算法。信息摘要算法主要是实现将大量的信息不可逆映射成一段定长或较短的信息而基本保持其独特性。所谓独特性，也就是说不同的信息经过相同的信息摘要算法映射后得到的结果应该是不同的。

密码协议是基于密码学的协议，它包含了某种密码算法，但通常不仅仅是为了加密，而是为了更加复杂的特定目的设计的。参与协议的各方可能是各种各样的人，可能是相互信任的朋友，也可能是敌人，他们的目的可能是为了共享秘密、确定相互身份或者共同签署合同。在密码协议中使用密码算法一般来说是为了防止和发现窃听、攻击和欺骗等。

### 1.2.3　密码学应用

随着网络尤其是 Internet 的发展，密码学的应用已经越来越广泛。目前主要的应用领域有电子商务、电子政务、私人邮件、Web 安全访问及虚拟专用网（VPN），等等。

电子商务是密码学迄今为止应用最成功的领域之一，主要涉及网络交易中的保密性、身份鉴别、完整性及抗抵赖等功能。例如用户在某电子商务网站使用信用卡进行购物时，必须保证整个过程是保密的而且是安全的，对于电子商务网站和银行来说，也必须确保用户的身份是可信的，并且能够向第三方证明用户确实执行了相关的操作。目前电子商务中成功的协议之一是 SET 协议。

电子政务及办公自动化系统的使用已越来越广泛，与之密切相关的安全性也出现了需求。比如公文的签发，需要数字签名，用户权限的控制需要进行身份确认，等等。

电子邮件是普通网络用户使用最为广泛的 Internet 工具，因为邮件涉及个人隐私等机密信息，所以保护电子邮件的安全早就成为了一个热点话题。目前，安全邮件系统 PGP 取得了很大的成功。PGP 不仅仅实现了邮件信息在网络传输中的机密性，而且能够建立用户之间的信任关系，确认用户的身份，并保证邮件信息的完整性。

Web 已经成了信息发布的一条重要途径，一些企业可能希望一些重要的资料文件不被未经授权的用户访问，另一方面，用户为了安全可能也需要验证服务器的身份。SSL 协议是解决这种需要的一个成功协议之一。

虚拟专用网技术（VPN）是解决大型的地理位置分布分散的公司或机构的低成本联网方案，该技术使用密码算法和密码协议通过 Internet 建立起分支机构之间的虚拟专用网络，使得各分支机构之间能够安全地进行通信。

## 1.3　公钥基础设施

公钥基础设施（PKI）是近年来受到非常多关注的一项技术，本节介绍公钥基础设施的基本概念及其组件，并介绍数字证书的基本作用。

### 1.3.1　公钥基础设施的必要性

公钥基础设施（PKI）是一种基于公开密钥算法的安全基础标准，它提供了一个框架，在这个框架内建立了可以创建鉴定和验证过程需要的身份和相关信任关系，建立了可以管理的公开密钥加密系统。

虽然公钥基础设施是建立在密码学的公开密钥算法的基础上，但其解决了仅仅依靠密

码学算法所无法解决的问题。公开密钥算法的密钥对能够实现对特定密钥对的鉴别和验证，但是无法建立将特定密钥对跟具体的个人身份联系起来的可信任关系。让我们来看一个利用公开密钥算法进行数字签名的例子。

创建一个数字签名，首先需要对文件的内容使用散列函数生成一个散列值以确保信息不被修改。然后就使用签名者的私钥对得到的散列值或摘要进行加密。数字签名的验证过程要求重新生成文档的散列值，再使用签名者的公钥对用签名者私钥加密的散列值进行解密，得到最初的散列值，然后比较这两个散列值。如果两个散列值一致，那么就认为签名通过。但是关键的问题是怎么确认那对先用于加密散列值，然后又用于解密散列值的密钥对确实是属于所声称的签名者的。对于一个聪明的入侵者来说，因为公钥是公开的，他可以通过某种方法用自己的公钥替换那个用于标识某个特定签名者的公钥，公钥被发布后，他就可以用自己的私钥生成一个能够通过上述验证过程的数字签名。所以，虽然拥有了公开密钥算法，但是算法本身并不足以确定一个可信人的身份。

公钥基础设施正是为了建立这种信任关系而产生的，它的主要目的就是建立可信任的数字身份，将特定密钥对和特定的人或实体联系起来，建立这种联系的主要形式就是颁发可信任的数字证书（或者叫电子证书）。

### 1.3.2 数字证书

对于初学者来说，数字证书的功能可能是难以理解的，事实上，你完全可以将数字证书跟你常用的身份证作对比。不过数字证书是你（或一个实体）在数字世界标识自己身份的证书罢了。表 1-1 显示了数字证书跟居民身份证项目的一些对应关系，虽然这些对应不一定非常贴切，但显示了数字证书的真实作用。

表 **1-1** 数字证书跟居民身份证项目的一些对应关系

| 居民身份证项目 | 数字证书项目 |
| --- | --- |
| 颁发机构是公安局 | 颁发机构是验证中心 |
| 公安局印章 | 验证中心电子签名 |
| 唯一的居民身份证号码 | 唯一的证书序列号 |
| 持有人姓名 | 证书拥有实体名称 |

事实上，在使用上，数字证书跟现实中的身份证也有很多相似的地方，比如在银行开户时，银行需要你出示身份证并验证身份证的真假和是否确实是你自己的身份证；在数字世界也一样，当你在电子商务网站购物时，出示你的数字证书，电子商务网站会验证你的证书是否可信并确定是否确实是你自己的证书。

身份证的颁发需要公安局等权威机构执行一系列的材料核对和验证工作，并需要居委会等机构参与身份证的验证和管理工作。数字证书的颁发也一样，首先就需要一个权威的大家都信任的机构，此外，还需要其他一些设施或机构参与数字证书的管理，这些数字世界设施的集合就是我们所说的公钥基础设施（PKI）。

### 1.3.3 公钥基础设施的组件

公钥基础设施并不是一个单一的设施，它由一系列的组件组成以完成其特定的功能。

在一个 PKI 的作用域中，并非下面介绍的所有设施都是必须的，有些设施如 RA 可能并不需要。

**(1) 验证机构 (CA)**

CA 可以说是 PKI 系统中的核心机构，负责确认身份和创建数字证书，建立一个身份和一对密钥之间的联系。作为一个程序员或技术人员，通常可能将 CA 跟证书签发服务器(或证书签发应用程序) 等同起来，事实上并非如此。CA 是一个软硬件和服务的集合，包括了人、操作流程、操作规程和验证策略、支持的软硬件及环境。一个成功的 CA 必须制定一些规则，使申请者和证书用户确信该 CA 所确认的身份适用于自己的目的并且是可信任的。一个 CA 是否能够获得成功，可能更重要的是在于其管理因素而不是技术因素。

**(2) 注册机构 (RA)**

RA 负责证书申请人的资料登记和初始的身份鉴别，还可能需要接受证书用户提出的证书撤销等其他服务。事实上，RA 是一个可选的组件，在很多时候，它所负责的功能并不需要独立出来，而是可以成为证书服务器的一部分。一般来说，RA 最主要的职责就是接受申请人的申请请求，确认申请人的身份，然后将确认了身份的申请请求递交给 CA。

**(3) 证书服务器**

证书服务器是负责根据注册过程中提供的信息生成证书的计算机或服务程序。证书服务器将用户的公钥和其他一些信息形成证书结构并用 CA 的私钥进行签名，从而生成正式的数字证书。事实上，证书服务器还可能要完成其他一些功能，比如证书的存放、发布及吊销等操作。技术人员在谈到 CA 时，通常就是指证书服务器。

**(4) 证书库**

任何证书在使用之前，必须将证书及其相应的公钥公布出去。证书库就是存储可以公开发布的证书的设施。通常以目录的形式组成 PKI 的证书库，如 X.500 目录或者 LDAP 目录。目前最为常见的是 LDAP 目录协议，它由一组对目录中定位信息的方法和协议描述组成。

**(5) 证书验证**

当证书用户收到一个证书的时候，需要对这个证书进行验证。证书验证的项目通常包括：

- 验证证书的签名者以确认是否信任该证书；
- 检测证书的有效期，确验证书是否有效；
- 确验证书没有被签发它的 CA 撤销；
- 检测证书预期用途跟 CA 在该证书中指定的策略是否相符合。

证书的验证过程通常是对证书链的验证，这通常要执行多个上述项目的循环验证已得到最终验证结果。证书的验证可以由客户端的验证程序执行，也可以提供专门的验证服务，客户端可以通过使用这种服务来完成验证。

**(6) 密钥恢复服务**

密钥对是确保证书能够顺利签发和正常使用的基本前提，密钥对的产生形式是多种多样的，既可以是在客户端的软件如 OpenSSL 的应用程序或 IE 浏览器的密钥存储器中产生，也可以是在 USB Key 或 Smart Card 这样的物理设备中产生，有时候也可能是在一个集中的密钥产生服务器中产生。无论哪种情况，都需要密钥恢复服务，使得加密密钥可以

存档，并且在它们丢失后能够恢复。设想如果 Susn 是公司的一个高层主管，她使用自己的密钥加密了众多的重要文件，如果有一天她在出差时不小心飞机失事，从而丢失了密钥，这时候如果不能恢复密钥，后果是可怕的。有时候，执法部门也会要求提供加密密钥。所以必须提供密钥恢复机制。

(7) 时间服务器

因为每台计算机的时间都是可以设置并且不尽相同的，但是证书的验证和签发等都需要一个统一、可信和单调增加的时间。这就使得时间服务器的存在有了价值。时间服务器可以为 PKI 作用域中的各个应用程序和 PKI 组件提供数字式时间戳，从而确保 PKI 域能够可信正确地运行。

(8) 签名服务器

许多 PKI 的应用程序和服务需要执行数字签名的操作，比如对文件的签名和对交易信息的签名，等等。可能很多应用程序本身不具备签名的功能，那么这时候可以使用签名服务器提供的服务，集中为各个应用程序生成签名。通常，签名服务器还提供验证签名的服务。

## 1.4 安 全 协 议

安全协议是密码学在计算机网络应用中的具体形式，几乎绝大部分的密码算法最后都要以安全协议的形式得到应用。本节介绍通用的网络模型，以及建立于其上的各种安全协议，重点介绍 SSL 协议。

### 1.4.1 网络模型和安全协议类型

虽然 ISO 网络模型是最标准的网络模型，但是，目前得到普遍应用的是 TCP/IP 网络模型。在本书中我们的讨论采用的是从 TCP/IP 网络模型中抽象出来的 5 层网络模型，其结构和对应的协议层如图 1-1 所示。

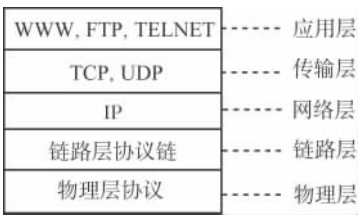


图 1-1 网络模型

目前比较成功的安全协议，基本上是在上面三层即应用层、传输层和网络层实现。下面我们介绍几个具有代表性的协议。

(1) 应用层安全协议 **PGP**

PGP（Pretty Good Privacy）协议是由 Philip Zimmerann 设计的免费保密电子邮件程序。严格来说，PGP 程序不能说是一个网络协议，但它确实是主要用于在网络上安全传递电子邮件。该程序能够保证电子邮件网上传输的安全性，具备了邮件加密、对邮件发送人的身份进行鉴别、对邮件进行数字签名和完整性验证等强大的功能。最新版本的 PGP

采用了多种加密算法以供用户选择，比如对称加密算法采用了 IDEA 和 3DES 等，公开密钥算法采用了 RSA 和 DH 等，信息摘要算法采用了 MD5 和 SHA1 等。

在使用 PGP 的时候，用户之间首先要建立信任关系，这种信任关系的建立主要由用户自己确定。用户可以通过对相互公开密钥的签名建立一个 PGP 用户的互联组。比如 Tom 认识 Jim，Tom 将自己的公钥传给 Jim 之后，Jim 在 Tom 的公钥上签名并传回给 Tom，Tom 保存这份签名的公钥。当 Tom 想和 Eric 通信时，就将有 Tom 签名的自己的公钥复制给 Eric，Eric 可能以前已经有 Jim 的公钥备份并信任 Jim 签名的其他用户的公钥，所以当它用 Jim 公钥验证该签名有效时，就接受 Tom，这样 Jim 就将 Tom 介绍给了 Eric。

**(2)** 传输层安全协议 **SSL**

SSL（Security Socket Layer）是由 Netscape 公司提出的一种安全协议，目前其版本已经发展到 SSL v3，标准化版本为 TLS（Transport Layer Security）。SSL 协议目前广泛地使用在 WWW 协议中，主流的 Web 服务器和浏览器都支持 SSL 协议，但是 SSL 协议绝不仅仅针对于 WWW 协议，它可以应用于传输层上的所有服务，在 1.4.2 节我们会更加详细地介绍 SSL 协议。

**(3)** 网络层安全协议 **VPN**

虚拟专用网（VPN）技术确切地说不是一个协议，目前实现 VPN 的技术多种多样，这里仅仅指使用 IPSec 隧道方式建立起来的 VPN。PGP 保护的是特定 Mail 应用的数据，SSL 协议如果有必要，可以建立两个特定主机之间的部分应用程序的安全通信信道，而 VPN 做的工作是保护任何两个子网或主机之间传输的数据的安全。VPN 能够加密和鉴别在网络层的所有通信量，VPN 提供了在公用网和 Internet 上建立安全通信信道的能力。

VPN 使用的典型例子是假如有一个包含多个地理上分散的局域网的机构，这些局域网内部运行着各种不安全的协议，为了管理上的需要，该机构希望将所有这些局域网连接起来，显然，如果采用租用专线的方式花费将会非常昂贵，但是普通的 Internet 连接显然是不安全的。这时候 VPN 就是一个很好的解决方案，实施 VPN 方案的各个局域网物理上通过 Internet 相互连接在一起，但是所有从局域网进入 Internet 的信息包都要经过 VPN 设备的鉴别和加密，所有从 Internet 如局域网进入的数据也都由 VPN 设备进行鉴别和解密。这样，对于 Internet 上局域网外的其他用户来说，这些局域网就是难以进入的。对于局域网的用户和应用程序来说，VPN 的这些操作是透明的，跟另一个局域网的用户建立连接和与本地局域网某个用户建立连接是一样方便的。

## 1.4.2　SSL 协议

SSL 协议是本书重点介绍的协议，所以有必要在进入本书的正题之前对 SSL 协议做更多的了解。SSL 协议最初是由 Netscape 公司提出的，并且已经在其第三版的基础上形成了 Internet 标准化版本 TLS。SSL 的提出最初是为了 Web 的安全性，但是 SSL 协议不仅仅能够解决 Web 的安全性问题，这对一般的技术人员和程序员来说是一个好消息，因为 Web 的安全性很大程度上掌握在像 Microsoft 这些巨型公司中。

SSL 协议被设计成使用 TCP 协议提供端到端的安全服务，实际上，SSL 由一组协议组成，而不是单个协议。SSL 的协议栈如图 1-2 所示。

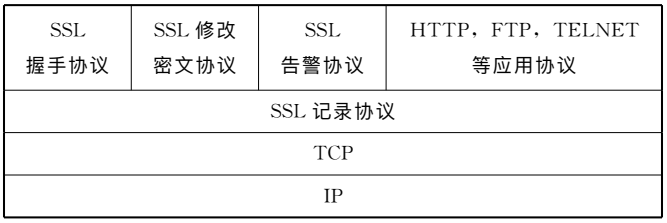


图 1-2 SSL 协议栈模型

SSL 记录协议为不同的高层协议提供安全服务，HTTP、FTP 等高层应用协议都可以在 SSL 协议上运行。SSL 握手协议、SSL 修改密文协议和 SSL 告警协议也是 SSL 协议的一部分，它们的作用是用来管理与 SSL 有关的交换。

SSL 协议中有两个重要的概念，即连接和会话。连接是指两台主机之间提供特定类型服务的传输，是点对点的关系。一般来说，连接是短暂的，每一个连接都与一个会话相关联。会话是客户和服务器之间的关联，会话是通过握手协议进行创建的。会话是加密安全参数的一个集合，包含了比如加密算法、临时加密密钥和初始向量等。会话可以被多个连接所共享，这样可以避免为每个连接重新进行安全参数的协商而花费昂贵的时间代价。任何一对服务器和客户之间可以存在多个安全的 SSL 连接，这些连接可以共享一个会话，也可以共享不同的会话。理论上说，一对服务器和客户之间也可以存在多个会话，但是由于这样付出的代价相当高，所以一般来说不支持这种做法。

SSL 记录协议为 SSL 连接提供了机密性和报文完整性两种服务。机密性和报文完整性所需要的密钥都是在握手协议中协商提供的。记录协议接收到传输的应用报文后，将数据分成可管理的块，进行数据压缩（可选）、应用 MAC、加密和增加首部，然后使用 TCP 报文传输。记录层接收到底层发来的数据后，进行解密、验证、解压和重新排序组合，然后交给上层的应用协议。

SSL 修改密文协议是一个最简单的 SSL 相关协议，它只有一个报文，报文由值为 1 的单个字节组成。这个协议的唯一作用就是将挂起状态复制到当前状态，改变连接将要使用的密文族。

SSL 告警协议是将 SSL 有关的告警信息传送给通信的对方实体。SSL 告警协议跟其他使用 SSL 的应用协议（如 HTTP 协议）一样，报文按照当前的状态被压缩和加密。

SSL 握手协议是 SSL 协议中最复杂的协议。服务器和客户端使用这个协议相互鉴别对方的身份，协商加密算法和 MAC 算法，以及在 SSL 记录协议中加密数据的加密密钥和初始向量。握手协议是建立 SSL 连接首先应该执行的协议，必须在传输任何数据之前完成。SSL 握手协议由一系列报文组成，根据功能基本上可以分成四个阶段。

第一阶段是建立安全能力。所谓安全能力是指将要在通信中使用的加密算法、数字签名算法、密钥交换算法、MAC 算法，以及其他一些记录协议需要使用的必要参数如初始向量，等等。这个阶段由两个参数相同的报文组成，一个是 Client_hello 报文，一个是 Server_hello 报文，协议的发起由客户端执行。这个阶段完成后，就完成了安全能力的建立。

第二个阶段是服务器鉴别和密钥交换。如果服务器需要被鉴别，这个阶段将从服务器给客户端发送自己的证书开始执行。这个阶段，服务器可能向客户端发送的信息包括证书

或证书链报文、密钥交换报文、客户证书请求报文及证书完成报文。除了最后一个报文，并非所有报文都是必须的。很多情况下，可能只要发送其中的一两个报文即可完成这个握手阶段。

第三个阶段是客户鉴别和密钥交换。收到服务器证书完成报文后，客户端首先验证服务器是否提供合法的证书，检测服务器的参数是否可以接受，如果这些都满足条件，客户端就向服务器发送客户证书报文（或者无证书告警信息）、客户密钥交换报文和证书验证报文中的一个或多个。除了客户密钥交换报文，其他两个报文在某些情况下不是必需的。

第四阶段是完成握手阶段。这个阶段完成安全连接的建立。首先是客户通过修改密文协议发送改变算法定义报文，将挂起的算法族定义复制到当前的算法族定义。然后客户立刻接着发送在新的算法和密钥加密下的完成报文。服务器对这两个报文的响应是发送自己的改变算法定义报文，将挂起状态复制到当前状态，并发送完成报文。到此为止，握手协议完成，客户端和服务器建立了安全连接，应用层协议可以使用 SSL 连接进行安全的数据通信了。

## 1.5　OpenSSL

介绍 OpenSSL 是本书的重点，本节将对 OpenSSL 做一个总的概述，介绍 OpenSSL 的简史和 OpenSSL 的组成，并讨论其优缺点。

### 1.5.1　OpenSSL 简史

上面我们简单介绍了 SSL 协议，与所有的协议一样，都只是一些规则而已，真正要应用，必须将协议规则转换成代码，对于任何个人和组织来说，这都是一项艰巨的任务，尤其是 SSL 协议这样一种涉及诸多专业知识的协议。目前，主流的 Web 服务器和浏览器都支持 SSL 协议，但由于这些商业产品的源代码不是开放的，使得对这些商业产品的信任和研究都比较落后。

OpenSSL 是一个开放源代码的 SSL 协议的产品实现，它采用 C 语言作为开发语言，具备了跨系统的性能。OpenSSL 项目最早由加拿大人 Eric A. Yang 和 Tim J. Hudson 开发，现在由 OpenSSL 项目小组负责改进和开发，这个小组是由全球的一些技术精湛的志愿技术人员组成，他们的劳动都是无偿的，在此我们应该向他们表示崇高的敬意。

OpenSSL 最早的版本在 1995 年发布，1998 年后开始由 OpenSSL 项目组维护和开发。当前最新的版本是 0.9.7b 版本，完全实现了对 SSLv1，SSLv2，SSLv3 和 TLS 的支持。OpenSSL 的源代码库可以从 OpenSSL 的官方网站 www.openssl.org 自由下载，并可以免费用于任何商业或非商业的目的。由于采用 C 语言开发，OpenSSL 的源代码库具有良好的跨平台性能，支持 Linux，UNIX，Windows，MAC 和 VMS 等多种平台。目前，OpenSSL已经得到了广泛的应用，许多类型的软件中的安全部分都使用了 OpenSSL 的库，如 VOIP 的 OpenH323 协议、Apache 服务器、Linux 安全模块，等等。我们有理由预期，OpenSSL 和其所倡导的开放源代码的思想必将被众多的支持者发扬光大。

### 1.5.2 OpenSSL 的组成

虽然 OpenSSL 使用 SSL 作为其名字的重要组成部分，但其实现的功能却远远超出了 SSL 协议本身。OpenSSL 事实上包括了三部分：SSL 协议库、密码算法库和应用程序。

SSL 协议库部分完全实现和封装了 SSL 协议的三个版本和 TLS 协议，SSL 协议库的实现是在密码算法库的基础上完成的。使用该库，你完全可以建立一个 SSL 服务器和 SSL 客户端。该部分在 Linux 下编译会形成一个明文 libssl.a 的库，在 Windows 下则是名为 ssleay32.lib 的库。

OpenSSL 密码算法库是一个强大完整的密码算法库，它是 OpenSSL 的基础部分，也是很值得一般密码安全技术人员研究的部分，它实现了目前大部分主流的密码算法和标准。主要包括公开密钥算法、对称加密算法、散列函数算法、X.509 数字证书标准、PKCS12、PKCS7 等标准。事实上，OpenSSL 的 SSL 协议部分和应用程序部分都是基于这个库开发的。目前，这个库除了可以使用本身的缺省算法外，在 0.9.6 版本之后，还提供了 Engine 机制，用于将如加密卡这样外部的加密算法实现集成到 OpenSSL 中。密码算法库在 Linux 下编译后其库文件名称为 libcrypto.a，在 Windows 下编译后其库文件为libeay32.lib。

应用程序部分是 OpenSSL 最生动的部分，也是 OpenSSL 使用入门部分。该部分基于上述的密码算法库和 SSL 协议库实现了很多实用和范例性的应用程序，覆盖了众多的密码学应用。主要包括了各种算法的加密程序和各种类型密钥的产生程序（如 rsa，md5，enc，等等）、证书签发和验证程序（如 ca，x509，crl 等）、SSL 连接测试程序（如 S_client 和S_server等），以及其他的标准应用程序（如 Pkcs12 和 Smime 等）。在某些时候，不需要做二次开发，仅仅使用这些应用程序便能满足我们的应用要求，比如采用 Ca 程序就能基本上实现一个小型的 CA 功能。这些应用程序同时也是使用 OpenSSL加密算法库和 SSL 协议库的优秀例子，比如 ca，req 和 x509 程序就是使用 OpenSSL的库开发一个 CA 中心服务器的优秀例子，又如 S_client 和 S_server 程序就是利用 SSL 协议库建立 SSL 安全连接的优秀例子。对于初学者来说，研读这些应用程序的源代码通常是最好的入门途径。

### 1.5.3 OpenSSL 的优缺点

除了 OpenSSL 之外，技术开发人员还有其他的一些密码算法库或 SSL 函数库可选。Wei Dai 写的 Crypto++就是著名的开放源代码算法库之一，该库使用 C++语言作为开发语言，由于采用面向对象的 C++语言，对于初学者来说可能更容易理解和接受。但是该库仅仅实现了一些常用的密码算法，而没有实现如 X.509 标准和 SSL 协议，其功能仅仅是 OpenSSL 的一个子集。此外，Microsoft 也提供了一个密码库 CryptoAPI，跟几乎大部分的 Microsoft 产品一样，CryptoAPI 不是开放源代码的，并且，它只能在 Windows 平台上使用，对于开发者来说是一个黑匣子，很多技术人员不喜欢这点。

与其他的一些同类型密码库相比，OpenSSL 具有以下优点：

- 采用 C 语言开发，支持多种操作系统，可移植性好；
- 功能全面，支持大部分主流密码算法、相关标准协议和 SSL 协议；

● 开放源代码，可信任，能够根据自己的需要进行修改，对技术人员有借鉴和研究的价值；

● 具备应用程序，既能直接使用，也可以方便地进行二次开发；

● 免费使用，能够用于商业和非商业目的。

当然，OpenSSL 也有如下一些缺点：

● 采用非面向对象的 C 语言开发，对于初学者来说有一定的困难，也不利于代码的剥离；

● 文档不全面，增加了使用的困难。

总的来说，OpenSSL 是一个非常优秀的软件包，很值得密码安全技术人员研究和使用，这也是写本书的目的所在。

## 1.6　本 书 概 要

本书偏重于基础，是 OpenSSL 的入门书籍，其章节基本是按照基本理论、基本结构到具体应用的顺序进行安排。

第一部分是密码学基础，包括第 2～3 章。该部分主要介绍了密码学的基本概念、密码技术的基本实现、对称加密算法、公开密钥算法和单向散列函数算法等密码学知识。如果读者对密码学已经有相当的了解，可以跳过这部分。

第二部分是 OpenSSL 结构，包括第 4～6 章。该部分介绍了 OpenSSL 的结构、编译和安装方法及其使用的基本概念。

第三部分是 OpenSSL 指令，包括第 7～12 章。该部分详细介绍了 OpenSSL 的应用程序指令的使用方法和各项参数的意义，是本书的重点。

其实，本来还应该花些时间对公钥基础设施（PKI）和 SSL 协议进行一些介绍的，但是考虑到目前市场上类似的介绍书籍已经不少，所以就略过了。

## 1.7　推 荐 资 料

由于 OpenSSL 是基于密码学和网络的，所以需要比较多的基础知识，本书对这些基础知识的介绍并不完备，所以向读者推荐一些参考资料，以便读者更深入地学习。

(1) 推荐书籍

●《计算机网络》，Andrew S. Tanenbaum 著，清华大学出版社，第 4 版。该书是计算机网络经典和集大成的教材，系统地阐述了计算机网络的原理、结构及相关的问题。

●《应用密码学——协议、算法和 C 源程序》，Bruece Schneier 著，机械工业出版社。该书是密码学的经典著作，对近年来密码学的研究成果做了全景式的概括。该书力图从概念上解释密码学，抛开了烦琐的数学公式。

●《密码编码学与网络安全：原理与实践》，William Stallings 著，电子工业出版社，第 3 版。该书对密码学的基本理论做了系统的介绍，并详细介绍了密码学的应用。

●《公钥基础设施（PKI）——实现和管理电子安全》，Andrew Nash 等著，清华大学出版社。该书全面介绍了公钥基础设施的概念，并对其实现、应用和发展前景作了介绍。

(2) 推荐网络资源

- http://www.openssl.org，OpenSSL 官方网站，可以自由下载 OpenSSL 的最新版本和以前的各个版本，并有 FAQ 和其他 OpenSSL 相关网站的连接。
- http://www.openssl.cn，国内唯一的 OpenSSL 中文专业站点，也是本书的支持站点，除了丰富的中文资料和例程下载之外，还有众多 OpenSSL 技术人员参与交流。
- http://www.google.com，强大的搜索引擎，你可以输入关键字如 OpenSSL 和 PKI 等查找你需要的资料。

# 第 2 章

# 密码学基本概念

## 2.1 密码学作用

密码学主要涉及信息加密、鉴别、完整性和抗抵赖的内容，本节将介绍信息加密、鉴别、完整性和抗抵赖的基本概念。

### 2.1.1 信息加密

我们称没有加密的信息为明文。在出现计算机密码学之前，在计算机和其他数字存储设备中保存的数字信息都是以明文的形式存在的，在诸如像 Internet 这样的网络上传输的信息也都是以明文信息存在的。这带来了诸多安全隐患，许多重要的文件可能被轻而易举地窃取，尤其当我们通过网络传输文件时，面临的危险是不可预测的。所以，有必要使用某种方法伪装明文以隐藏真正消息，这个伪装的过程称为加密，加密后得到的信息称为密文。一般来说，加密的过程是可逆的，也就是说，密文在需要的时候可以转换成明文，这个过程称为解密。加密和解密的过程如图 2-1 所示。


图 2-1　加密和解密过程

我们通常愿意用简单的公式表示这个过程，下面先介绍一些通用的约定符号。一般来说可以使用 $M$ 和 $P$ 表示明文，这里的明文其实是抽象的信息，可以是一些比特位的组合、文本文件、图形、语音或者视频，等等。其实对于计算机来说，所有的信息都是二进制数据，都是 0 或 1 的比特位的组合。所以明文其实就是指待加密的信息。

密文一般使用 $C$ 表示，事实上，在计算机上它也是二进制的数据。在一般情况下，密文的长度跟明文的长度相同或者稍微长一点。

加密函数一般使用 $E$ 来表示，解密函数则使用 $D$ 来表示。这样，加密的过程可以用如下公式表示：

$$C=E(M)$$

解密的过程则可以表示为

$$M=D(C)$$

一个正确的密码算法应该能够正确地恢复初始的明文，所以加密和解密函数必须满足

下列公式

$$M=D(E(M))$$

事实上，所有的密码算法基本上都执行上述公式描述的过程，只是函数 $E$ 和 $D$ 有区别，而这个区别，正是不同密码算法的关键区别所在。之所以要研究各种不同的加密算法，原因是多方面的，但是总是有人试图要从密文中使用各种手段（不是使用 $D$ 函数，因为他一般来说没有这个条件）恢复出明文是其中主要的原因。我们称这种行为为密码分析，自然这种技术叫做密码分析技术；而设计密码算法的技术则称为密码编码技术。至此我们知道事实上密码学可以分为密码编码学和密码分析学，这两种对抗技术的存在正是导致了现在那么多繁杂的密码算法存在的主要原因。

### 2.1.2 鉴别

现在密码技术已经远远不止于提供信息加密功能，它还能提供鉴别、完整性及抗抵赖的功能。鉴别能力就是指消息的接收者应该能够确认消息的发送者确实为发送者本身，入侵者不可能伪装成他人。

鉴别在日常生活中是常见的，比如你到银行，银行的营业员通过你出示的身份证上的照片和你的面容对照来鉴别你的身份。在更严格的机构，比如国家安全局的某个机要室，可能会采用你指纹或 DNA 样本来鉴别你的身份。鉴别之所以如此常见，是因为鉴别是确保交易或交流安全进行的根本前提和手段，任何人或机构都不会信任一个无法鉴别其身份真假的个人或实体，更别说进行交易了。

在数字世界鉴别也一样重要，其概念和要求跟现实世界是一致的，只是其鉴别的对象、执行对象及鉴别方法可能会有所区别。

数字世界鉴别的对象和执行的对象是多种多样的，可能是一个人，也可能是一个应用程序、一台计算机或其他数字终端，我们统称这些对象为实体。无论实体是什么，鉴别的目的是相同的，就是确认发送消息的实体是其所宣称的实体本身。比如一台计算机 Client 要跟另外一台计算机 Server 通信，Client 向 Server 宣称它是 Client，那么为了安全起见，Server 会要求它向自己证明它是 Client。

在现实世界中，我们有多种认为一定程度上可靠的鉴别方法，在面对面的交流中，可以使用权威机构发放的证件（如身份证）跟本人对照；在书面文件中，可以使用手写签名、公章等达到鉴别的目的。在数字世界中，情况要复杂得多，主要的原因是因为数字世界缺乏像现实世界那样的权威，到后面章节，读者会更加深刻地理解这一点。数字世界中最简单和普遍的鉴别方案应该就是用户名和口令的方案，不过密码学提供了数字世界中更好的鉴别方案，这通常通过一套密码协议完成。常用的鉴别协议是基于公开密钥算法和对称加密算法的，通过这种方式，信息接收者能够鉴别信息确实是由信息发送者所宣称的实体发出的。事实上，这里鉴别的也不是实体本身，而是使用公开密钥对代替了实体本身，这导致很多问题存在，比如如果代表某人的公私密钥对被其他人所获取，那就可以冒充该人的身份。为了解决这个问题，产生了很多不同形式的安全设备和协议，如智能卡和 USB Key，等等，这些设备没有一个真正解决了这个问题，只是降低了受攻击的风险而已。

### 2.1.3　完整性

完整性用来确保信息没有被修改，也可以防止假冒的信息。没有接触过密码学的人对于完整性的概念可能比较陌生，因为在现实的世界中，完整性并不是一个很引人注目的事情，这与完整性在现实世界中的必要性不是特别明显是有关的。一个印刷在书面上的文件，要想通过修改其上面的文字或者数字来破坏其完整性是不容易的，你可以涂抹文件上面的文字，但可能很容易就被发现了。现实世界中也有需要完整性的例子，比如一个由多个页面组成的合同文本，我们可能会通过骑缝章来保证其完整性。

相对于现实世界而言，存储在计算机中的数字信息的完整性受到破坏的风险就大大增加了。一个存储在计算机中重要的文本文件，可能被其他人恶意修改了其中一个重要的数字而你没有发现就发出去了，甚至可能整个文件都给替换了而你没有再看一眼就发出去了。在网络传输中，完整性面临的风险就更大，这种风险有两种，一种是恶意攻击，一种是偶尔的事故，无论哪种，都可能造成无法弥补的损失。在网络中，恶意攻击者可以监听并截获你的信息包，然后修改或替换其中的信息，再发给接收方，这样能够不知不觉地达到其目的。网络是一个物理设备，虽然其出错的可能性极低，但还是有可能发生的，如果你在给某个商家转账的过程中其中的付款数字在网络传输时发生了错误而没有发现，后果是严重的。总之，在数字世界中，为了能够安全地存储信息和传输信息，完整性是必不可少的。

目前完整性的解决方案主要是基于单向散列函数和加密算法。单向散列函数能够将一个大的文件映射成一段小的信息码，并且具有不同的文件映射成的信息码相同的概率极小的性质。通常我们将会将原始信息使用单向散列函数处理得到一段信息码，然后将其加密，跟文件一起保存。如果有人更改了文件，那么当我们再次使用该文件时，先使用同样的单向散列函数得到信息码，然后用自己的密钥解密原来生成的信息码跟新得到的信息码对比，那么就会发现不一样，就可以发现文件已经被修改。完整性一般是用来确保信息没有被修改，但是在发现信息被修改后，怎么进行恢复也是一件有趣的事情，不过本书不研究这个问题。

### 2.1.4　抗抵赖

这个世界总是有那么一些不地道和不诚实的人，他们通常为了达到自己的目的否认自己曾经做过的事情或说过的话，这种行为我们称为抵赖。为了制止这种令人厌烦甚至会带来损失的行为，我们通常会采取防范措施，最典型的就是签字画押和按手指纹等，这种措施称为抗抵赖。

在数字世界中，通常存在这样的行为。比如一个人可能在网上某个网站购买了一台摄像机，是通过网上银行进行转账付款的。在收到摄像机并收藏好后，他却跟银行说他没有购买过该商品，也没有要求银行支付该款项。面对这种抵赖行为，银行必须有应对的措施和证据，能够向法院或其他第三方证明该人确实执行了转账的操作。由此可见，抗抵赖的基本特点是能够向第三方证明消息发送者确实发送了该消息。

目前，密码学已经提供了技术上可以抗抵赖的解决方案，就是数字签名技术。数字签名技术主要建立在公开密钥算法的基础上，并且需要公钥基础设施的支持才能够运作。由

于数字签名技术其实只是将特定的密钥对跟某个实体联系起来，所以必须具备一个权威的验证机构，这是一个难点。此外，虽然技术上可以证明数字签名具有抗抵赖的功能，但是要在实际中使用，更重要的问题是还需要法律的支持。目前在发达的国家已经有了数字签名法案，承认数字签名的法律意义，我国也颁布了数字签名的法案，但所有这些都还只是刚刚起步，还需要做很多的努力。

## 2.2 密码数学

密码学是数学的一个分支，其各种应用也是建立在数学的基础上的，这一节将简单介绍一些密码学算法常用的数学运算。对于大多数读者来说，这一节可能是枯燥无味的，如果你不想深入了解密码算法的原理，可以先跳过这一节。

### 2.2.1 素数

如果等式 $a=mb$ 成立，其中 $a$，$b$ 和 $m$ 都为整数，当 $b\neq 0$ 时，称 $b$ 能整除 $a$，用符号 $b|a$ 表示。此时，称 $b$ 为 $a$ 的一个因子。

如果整数 $p$ 是大于 1 且因子仅为 $\pm 1$ 和 $\pm p$ 时，称 $p$ 为素数。

整数 $a$ 和 $b$ 互素，是指它们之间没有共同的素数因子。即 $a$ 和 $b$ 的最大公因子为 1。确定一个大数的素数因子是不容易做到的。

### 2.2.2 模运算

模运算是许多密码算法都使用的基本数学运算，这一节将对模运算的性质作基本的介绍。如果读者需要了解更详细的性质，可以参考相关的数学书籍。

给定任意正整数 $n$ 和整数 $a$，总可以找到整数 $q$ 和非负整数 $r$，使得

$$a=qn+r,\ 0\leqslant r<n$$

成立，其中 $q$ 为小于等于 $a/n$ 的整数，称为 $a$ 除以 $n$ 的商，$r$ 称为余数。

模运算符用 mod 表示。设 $a$ 是一个整数，$n$ 是一个正整数，则定义 $a \bmod n$ 为 $a$ 除以 $n$ 的余数。所以对任意整数 $a$，可以表示为

$$a=[a/n]\times n+(a \bmod n)$$

其中，$[x]$ 表示小于或等于 $x$ 的最大整数。

例如 $13 \bmod 7=5$，而 $-13 \bmod 7=1$

如果 $(a \bmod n)=(b \bmod n)$，则称 $a$ 和 $b$ 模 $n$ 同余，记做 $a\equiv b \bmod n$。特别的，如果有 $a\equiv 0 \bmod n$，则 $n \mid a$。

模运算符具有如下性质：

- 如果 $n|(a-b)$，则 $a\equiv b \bmod n$；
- $(a \bmod n)=(b \bmod n)$ 等价于 $a\equiv b \bmod n$；
- $a\equiv b \bmod n$ 等价于 $b\equiv a \bmod n$；
- 如果 $a\equiv b \bmod n$ 且 $b\equiv c \bmod n$，则有 $a\equiv c \bmod n$。

事实上，模操作是将所有整数都映射到有限整数集合 {0，1，2，…，$n-1$} 中，在该集合内，一样可以进行算术运算，这就是模运算。下面列出模运算的一些性质。

基本性质：

- $[(a \bmod n)+(b \bmod n)] \bmod n=(a+b) \bmod n$
- $[(a \bmod n)-(b \bmod n)] \bmod n=(a-b) \bmod n$
- $[(a \bmod n)\times(b \bmod n)] \bmod n=(a\times b) \bmod n$

交换律：

- $(a+b) \bmod n=(b+a) \bmod n$
- $(a\times b) \bmod n=(b\times a) \bmod n$

结合律：

- $[(a+b)+c] \bmod n=[a+(b+c)] \bmod n$
- $[(a\times b)\times c] \bmod n=[a\times(b\times c)] \bmod n$

分配律：

- $[a\times(b+c)] \bmod n=[(a\times b)+(a\times c)] \bmod n$

恒等式：

- $(0+a) \bmod n=a \bmod n$
- $(1\times a) \bmod n=a \bmod n$

需要注意，在模运算中，下列等式不能恒成立：

$$(a\times b)\equiv(a\times c) \bmod n，\text{则 } b\equiv c \bmod n$$

如果附加 $a$ 与 $n$ 互素的条件，则等式成立。

### 2.2.3　数学定理

这一节介绍几个基础的定理，这些定理通常是后面要叙述的加密算法的基础。

**(1)** 费马定理

如果 $p$ 是素数，$a$ 是不能被 $p$ 整除的正整数，则有：

$$a^{p-1}\equiv 1 \bmod p$$

**(2)** 欧拉定理

欧拉函数 $\Phi(n)$ 是表示小于 $n$ 且与 $n$ 互素的正整数的个数。对于任意素数 $p$，有：

$$\Phi(p)=p-1$$

对两个不同的素数 $p$ 和 $q$，令 $n=pq$，则有：

$$\Phi(n)=\Phi(pq)=\Phi(p)\Phi(q)=(p-1)(q-1)$$

欧拉定理表述为对于任何互素的整数 $a$ 和 $n$，有：

$$a^{\Phi(n)}\equiv 1 \bmod n \text{ 或者 } a^{\Phi(n)+1}\equiv a \bmod n$$

那么根据我们前面的定理，就有：

$$a^{\Phi(n)+1}=a^{(p-1)(q-1)+1}\equiv a \bmod n$$

熟悉 RSA 算法的读者可能都知道，这就是 RSA 算法的来源根据。

### 2.2.4 异或运算

异或运算是计算机密码算法经常采用的运算方法，它是对比特位的操作，在 C 语言中使用的操作符是“^”，我们在公式中使用 xor 作为其标记。异或运算其实很简单，其基本性质如下：

$$0 \text{ xor } 0=0$$
$$0 \text{ xor } 1=1$$
$$1 \text{ xor } 0=1$$
$$1 \text{ xor } 1=0$$
$$a \text{ xor } a=0$$
$$a \text{ xor } b \text{ xor } b=a$$

我们在后面的章节中会看到，许多算法都使用了异或操作以完成加密功能。

### 2.2.5 随机数

好的随机数是生成安全的密钥的基础，这是密码学中随机数之所以如此重要的原因。目前在大多数系统和编译器中都嵌入了随机数发生器，简单的调用这些函数就可以产生随机数。但是，这些随机数发生器产生的随机数一般来说是非常差的随机数，它们根本不能产生真正的随机数，更别说随机序列了，对于一般简单应用程序来说，这或许已经足够了，但是对于密码学来说远远不能保证安全性。

事实上，在计算机上不可能产生真正的随机序列，因为计算机的状态数总是有限的，所以其必然存在周期性，周期性的东西在一定程度上都是可预测的，既然是可预测，那么也就不是真正意义上的随机数。要产生真正的随机序列，需要找到真正随机的输入序列，这在计算机中是不可能找到的。解决这个问题的折中办法是使用伪随机序列。所谓伪随机序列是指虽然该随机序列不是真正的随机序列，但是由于序列的周期足够长，使得在实际应用中使用到的相对很短的子序列看起来跟真正的随机序列没有太大的区别。也就是说，如果有一个周期为 2 亿的随机数产生器，而一般来说使用的位数只是几千位以下的量级，那么对于这几千位来说，可能跟一个真正的随机数产生器的输出看起来没有太大的区别，那么这个随机数序列就可以称为伪随机数序列。所谓没有太大的区别，事实上是一个难以确定的概念，需要通过试验来测试。比如它们产生 0 和 1 的数目应该大约相同，长度为 1 的游程大略占 1/2，长度为 2 的游程大约占 1/4，等等。0 和 1 的游程分布应该是相同的，它们的长度应该是不可压缩的，等等。也就是说，一个伪随机数发生器，应该能够通过所有我们能找到的随机性统计检验，这样看起来是随机的。

有许多文献提出了各种各样的伪随机数发生器和随机数检验方法，它们当然都是周期性的，但是一般来说，只要周期大于 $2^{256}$ 位，就基本可以使用。但对于密码学应用来说，仅仅是伪随机数是不够的，还应该具备更严格的要求，这种随机序列称为密码学意义上安全的随机序列，它们应该具备以下性质：

- 是伪随机序列，即看起来是随机的，能够通过各种已知的随机性检验；
- 随机序列是不可预测的，给出产生序列的算法或硬件和所有以前产生的随机序列

的全部知识，也不能通过计算预测下一位是什么。

对于一般的密码学应用来说，密码学意义上安全的随机序列就可以满足要求了，并不必苛求真正的随机数。事实上，真正的随机数是难以评估的，它相比于密码学意义上安全的随机数序列，还应该附加不能重复产生的条件，也就是说给定完全相同的输入不会产生相同的输出。但是这个条件的评估是难以实施的。所以，对于我们的应用来说，一个密码学意义上安全的随机数产生器就足够了。

### 2.2.6　大数

大数是密码学书籍中经常使用的概念，但是却没有明确的定义，这似乎是一个很有趣的事情。为了让读者建立一个基本的大数的概念，下面列出了一些自然界的物理量来说明大数，见表 2-1。

表 **2-1**　大数列举

| 物理意义描述 | 大　数 |
|---|---|
| 宇宙的年龄 | $2^{34}$ |
| 行星中的原子数 | $2^{170}$ |
| 太阳中的原子数 | $2^{190}$ |
| 银河系中的原子数 | $2^{223}$ |
| 宇宙中的原子数 | $2^{256}$ |
| 宇宙的生命期 | $2^{37}$ 年 |
| 宇宙的体积 | $2^{280}$ 平方厘米 |
| 下一个冰川年代时间 | $2^{14}$ 年 |
| 行星的年龄 | $2^{30}$ 年 |

## 2.3　密 码 算 法

密码算法就是用于加密和解密的函数，本节介绍密码算法的基本概念、种类、评估标准和应用，使读者对密码算法有一个初步的认识。

### 2.3.1　算法基础

密码学的具体应用形式一般来说都表现为密码算法。密码算法一般由一个加密函数和一个解密函数组成。

密码算法的目的主要就是为了保密信息，如果算法的保密性是基于算法本身的，这种算法称为有限制算法，如果算法的保密性不是基于算法本身，而是基于密钥的，那么属于无限制算法。

有限制算法在历史上曾经广为使用，但是其有很多局限性，根据现代密码学定义的保

密标准，它们已经远远不能满足需要。有限制算法有如下的缺点。

- 大的或成员经常变化的组织不能使用该算法，因为一旦一个成员离开这个组织，为了安全的需要，其他成员就必须更换新的算法。如果有人故意或无意曝露了这个算法，组织的所有人也必须更换该算法。
- 有限制算法不可能进行标准化和质量控制，质量得不到保证。因为每个组织都必须有自己唯一的算法，并且由于保密性的需要，不能购买流行的软件或硬件产品，否则窃听者也可以购买这些产品并学习使用该算法，所以组织用户只能自己编写和实现算法。这样不但成本很高，而且如果组织没有密码学家，也没有办法评估算法的安全性。

尽管有这些缺陷，还是有很多应用采用这种算法，比如 GSM 的加密算法。这一方面可能由于用户没有认识到这些缺陷，另一方面则是因为用户可能不在乎这些问题。

现代密码学解决这个问题的办法是使用了密钥。密钥一般用 K 表示。密钥可以由很多数里面的任意一个或几个值组成，我们称某类型密钥的可能值的集合为密钥空间。加密和解密都使用相同密钥的算法一般称为对称加密算法，如 DES 算法。在这种密码算法中，加密和解密运算都使用相同的密钥 K。其加密和解密函数分别为

$$C=E_K(M)$$
$$M=D_K(C)$$

为了能够正确恢复初始明文信息，需要满足下面的性质：

$$D_K(E_K(M))=M$$

图 2-2 是这种加密算法的流程示意图。


图 2-2　对称加密算法的加密和解密过程

与对称加密算法不同的是，有些算法使用的加密密钥跟解密密钥并不相同，这种加密算法大多数称为非对称加密算法（如果其加密密钥和解密密钥能相互推算出来，就还称为对称加密算法），如 RSA 算法。我们使用 Ke 表示加密密钥，Kd 表示解密密钥，则加密函数、解密函数和算法应该满足的如下等式：

$$C=E_{Ke}(M)$$
$$M=D_{Kd}(C)$$
$$D_{Kd}(E_{Ke}(M))=M$$

其加密和解密流程如图 2-3 所示。


图 2-3　非对称加密算法的加密和解密过程

无论是对称加密算法还是非对称加密算法都是基于密钥的安全性，而不是密码算法自身的安全性。也就是说，密码算法本身是公开的，可以接受分析，也可以生产使用同样算法的大量产品。窃听者可能知道你的算法，但是因为不知道你的密钥，所以他同样不可能阅读你的信息。

### 2.3.2　对称加密算法

对称加密算法又称为传统密码算法、单密钥算法或秘密密钥算法。事实上，它的定义是加密密钥能够从解密密钥中推算出来，解密密钥也能够从加密密钥中推算出来的算法。也就是说，其加密密钥和解密密钥不一定是相同的，只是可以相互推算出来而已。当然，对于大多数对称加密算法来说，加密密钥和解密密钥是相同的。对称加密算法要求通信双方在开始通信之前，要首先商定一个用于加密和解密的密钥。算法的安全性就依赖于这个密钥，如果这个密钥被泄漏了，就意味着通信可能不再安全。要保证通信中信息的安全，就必须保证密钥的安全。

根据密码算法的加密方式不一样，对称加密算法又可以分成流加密算法和块加密算法。

流加密算法又称为序列加密算法或序列密码，它每次只对明文中的单个位或单个字节进行加密操作。这种算法的优点是能够实时进行数据传输和解密，缺点是抗攻击能力比较弱。

块加密算法又称为分组加密算法或分组密码，它每次对明文中的一组数据位进行加密操作。现在使用的分组加密算法典型的分组长度是 64 位，因为经过密码学家们证明，这个长度大到足以防止破译攻击，而又小到足以方便使用。块加密算法的优点是抗攻击能力好，但是实时性稍微差一点。

### 2.3.3　非对称加密算法

非对称加密算法也叫公开密钥算法，是现代密码学取得的最大成就之一，也是密码学在近 20 年来能够快速发展和推广应用的主要原因之一。公开密钥算法中加密密钥跟解密密钥不一样，并且解密密钥理论上很难根据加密密钥推算出来。非对称加密算法的加密密钥是可以公开的，理论上任何人都可以获得这个公开的加密密钥并加密数据。但是，使用公开的加密密钥加密的信息只有相应的解密密钥才能解开，而这个解密密钥一般是不公开的。在非对称加密算法系统中，加密密钥也叫作公开密钥或公钥，而解密密钥则称为私人密钥或私钥。

非对称加密算法的这种性质使得其用途非常广泛，目前来说，非对称加密算法一般有两种典型的应用：普通的数据加密和数字签名。

在普通的数据加密应用中，使用公开密钥进行加密，而使用私人密钥进行解密。使用公开密钥算法加密数据的好处是通信双方不需要事先协商加密密钥，减少了密钥泄漏的风险，也扩大了通信对象的范围。理论上说，所有的人都可以使用公开密钥算法轻而易举地建立安全的通信信道。用 Kpu 表示公开密钥，Kpr 表示私人密钥，则可以表示如下：

$$C = E_{\mathrm{Kpu}}(M)$$
$$M = D_{\mathrm{Kpr}}(C)$$
$$D_{\mathrm{Kpr}}(E_{\mathrm{Kpu}}(M)) = M$$

公开密钥算法的另一个巨大贡献就是解决了数字签名的难题。在数字签名应用中，私人密钥用于加密，而公开密钥用于解密。因为一般情况下只有密钥对的持有人才会拥有私人密钥，所以如果用该人密钥对的公开密钥能够正确解开一个加密的数据，那就可以证明该数据是密钥对的持有人签发（加密）的。这就是公开密钥算法应用于数字签名的基本原理。数字签名的运算可以表示如下：

$$C = E_{\mathrm{Kpr}}(M)$$
$$M = D_{\mathrm{Kpu}}(C)$$
$$D_{\mathrm{Kpu}}(E_{\mathrm{Kpr}}(M)) = M$$

### 2.3.4 算法安全性

之所以要考虑算法的安全性，根本的原因在于总有人想要攻击和破译算法。事实上，攻击和破译算法也已经成了一门专门的学问，称为密码分析。密码分析是指在不知道密钥的情况下，得出明文或密钥的技术。密码分析针对的对象可能是密钥，也可能是密码算法本身。常用的密码分析方法有以下一些。

- 唯密文攻击。密码分析者通过各种途径得到了使用同一算法加密的密文，密码分析者通过分析这些密文得出部分或全部明文消息，有时候也能够得到加密密钥，这样就可以用来解密别的密文。这种情况下一般要求有较多的密文作为攻击资料。
- 已知明文攻击。密码分析者在幸运的情况下，不但得到了密文，而且得到了这些密文相应的明文。他们通过密文和明文的对照推导出加密的密钥或者设计一种新的算法，使得使用相同密钥和算法加密的信息都能够被解密。
- 选择明文攻击。密码分析者不但可以得到密文和相应的明文，并且在必要的时候可以选择被加密的明文，这样，密码分析者能够获取更多的信息用来获得加密的密钥或者设计一种新的算法，使得使用相同密钥和算法加密的信息都能够被解密。
- 选择密文攻击。密码分析者可以选择不同的被加密的密文，而且能够得到相应的明文，从而从这些信息中推导出密钥或设计新的解密算法。
- 选择密钥攻击。事实上密码分析者并不能选择加密密钥，这里指他可能具有各种密钥之间相互关系的一些专业知识，从而用这些专业知识达到破译目的。
- 社会工程。事实上，很多情况下这是最有效的方法，就是通过欺骗、威胁、贿赂、监视和其他非法手段对密钥持有人进行攻击，直到得到密钥为止。

上述的所有攻击手段（除了最后一个）都可以作为我们评估一个算法安全性的参考依据。事实上，没有一个算法可以宣称是绝对安全的，各种算法有不同层次的安全性。对算法安全性的评估没有统一的标准，但基本上可以从以下几个方面考虑。

- 价值代价。如果破译算法的代价大于加密数据的价值，算法可以算是安全的。
- 时间代价。如果破译算法所需要的时间比加密数据需要保密的时间更长，算法可以算是安全的。

- 可能性。如果使用该密钥加密的数据量远远小于破译该算法需要的数据量，算法可以算是安全的。

根据破译算法程度的不同，可以将算法安全性分为五个等级，下面的安全性由低到高列出。

- 全部破译。能得到密钥，使得所有使用该密钥加密的密文都能被恢复成明文。
- 全盘推导。通过分析，可以找到一个替代算法，在不知道密钥的情况下恢复出所有该密钥加密的明文。
- 局部推导。能够从得到的密文中计算出相应的明文。
- 信息推导。通过分析可以得到部分有关密钥或明文的信息。
- 不能破译。无论给出多少密文信息，都不能恢复出明文信息。事实上，只要使用蛮力攻击，几乎所有实际的算法都是可以破译的（除了一次一乱密码）。

在实际使用中，我们只要考虑一个算法在计算上的安全性就可以了。也就是说，当一个算法用当前所能得到的所有资源都不能破译的时候，就可以说这个算法是安全的。破译算法的难易程度可以用攻击算法的复杂性来衡量。攻击算法的复杂性从下面三个方面衡量。

- 数据复杂性。即攻击时需要的数据量。
- 计算复杂性。完成攻击计算所需要的时间或计算量。
- 存储量需求。进行攻击所需要的最大存储空间。

一般来说，攻击的复杂性值等同于这三个因素中最小的值。

复杂性一般用数量级来表示，比如一个算法的计算复杂性是 $2^{128}$，那么破译这个算法就需要 $2^{128}$ 次运算。这可能用最好的计算机也要花费 $10^{15}$ 年以上的时间才能完成计算，这在实际中是不可行的，那么该算法也就是安全的。当然，现在的计算机速度快速发展，目前看起来不可能破解的算法过几年或几十年可能就能轻而易举地被破解。但是没有必要担心，只要你使用的算法在最近的几年或几十年是安全的，那么就是一个计算上安全的算法。以后计算能力增强了，自然会出现更强大的加密算法。

## 2.4　密码通信协议

密码学在网络中的应用总是通过通信协议的形式实现的，在本节，将对通信协议中的基本概念和基本组件进行介绍，为以后进一步介绍密码通信协议打下基础。

### 2.4.1　基于密码学的安全通信

信息数据在一般网络中的传输都是明文形式，很容易受到窃听、截获及更改等攻击，这在很多情况下是通信双方不希望出现的。谈恋爱的情侣不希望自己的悄悄话被偷听，通过网上进行电子交易的用户也不希望自己的账号和密码泄漏出去。总之，安全通信是大多数人和组织都追求的目标。

目前来说，基于密码学的安全通信技术总的来说有两种，一种基于对称加密算法，一种基于公开密钥算法。下面将分别介绍这两种方法。

### 1. 基于对称加密算法的安全通信

一般来说，对称加密算法的特点就是在加密和解密中需要使用相同的密钥。用户 Tom 和 Jim 使用对称加密算法进行安全通信的步骤通常如下：

① Tom 和 Jim 通过网络协商将要使用的加密算法类型；

② Tom 和 Jim 通过网络协商将要使用的加密和解密密钥和初始向量（有些算法需要）；

③ Tom 使用协商好的密钥和算法加密要发送的信息；

④ Tom 通过网络发送加密后的信息给 Jim；

⑤ Jim 收到 Tom 发来的信息，使用协商好的算法和密钥解密信息得到明文。

对这样的一个简单的基于对称加密算法的网络协议，对于一个攻击者来说，有许多可以攻击的地方。最基本的就是唯密文攻击，也就是说攻击者可以在 Tom 通过网络给 Jim 发送加密后的信息的时候监听他们的通信过程，取得一些密文信息，然后进行分析攻击。事实上，对于一个网络传输过程来说，无论什么算法或协议都无法避免这种攻击。抵抗这种攻击的方法不在于协议，而在于选择一个好的算法和一个足够长的密钥，也可以通过经常更换加密密钥来增加抗攻击能力。

但事实上对这样一个网络通信协议，聪明的攻击者根本不需要这么费劲进行唯密文攻击，它可以设法从一开始就监听整个通信过程。因为该协议使用对称加密算法，一般来说，在完成协商加密密钥之前，通信双方都必须使用明文进行通信，这包括协商加密算法和加密密钥。这样，攻击者也同样可以获知加密算法和加密密钥，剩下的事情就是监听整个通信过程，使用获取的加密算法和密钥简单解密其中传输的信息，甚至在某种情况下攻击者还可以主动一点，截获信息，然后解密成明文信息，接着进行更改或使用新的信息后做相同的加密，再发给对方以达到自己的特定目的。解决基于对称加密算法的安全通信协议这个问题有一种基本方法，就是在进行安全通信之前先通过其他手段交换（或协商）密钥，比如通过软盘复制等非网络手段。但是基于对称加密算法的安全网络协议必须秘密地交换密钥这一特点给协议的应用带来了很多的限制。

总结起来，基于对称加密算法的安全网络通信协议存在以下问题。

- 密钥必须秘密地进行分配，这通常只能采用物理的传输手段实现，对于自动化程度高和应用广泛的加密系统来说，这样做的代价将是不可接受的。
- 一旦加密密钥丢失，那么获得密钥的攻击者就有能力解密通信中的任何数据，并且可以假装成其中任何一方发送虚假信息而不被发觉。
- 随着用户的增多，需要的密钥个数会成平方级迅速增长。在安全性要求高的系统中，用户两两之间进行通信使用的密钥都应该是不同的，这样如果 $n$ 个用户就需要 $n(n-1)/2$ 个密钥。这样，随着用户数的增多，密钥将可能会多到不可忍受的地步。这可以通过每几个用户为一组使用一个共同的密钥来减少密钥数，但这是需要以牺牲安全性为代价的。

基于对称加密算法的安全通信协议存在的这些问题是由于对称加密算法本身的性质决定的，公开密钥算法提供了更好的解决方案。

### 2. 基于公开密钥算法的安全通信

第一个公开密钥算法是 DH 算法，在 1976 年由 Whitfield Diffie 和 Martin Hellman 提

出。正如目前所看到的情况一样，公开密钥算法的提出对于密码学来说是革命性的，也解决了上述基于对称加密算法的通信协议无法解决的问题。回顾前面介绍过的公开密钥算法的内容，可以知道，公开密钥算法的主要特点就是加密和解密密钥不相同，由加密密钥推导不出解密密钥而且加密密钥是可以公开的。下面是 Tom 和 Jim 使用基于公开密钥算法的安全通信协议进行通信的步骤：

① Tom 和 Jim 通过网络协商要使用的公开密钥算法；

② Jim 将自己的公钥（加密密钥）通过网络传送给 Tom；

③ Tom 使用 Jim 的公钥加密要发送的消息；

④ Tom 将加密后的消息通过网络传送给 Jim；

⑤ Jim 使用自己的私钥（解密密钥）解密收到的 Tom 发送的消息。

这样，基于对称加密算法通信协议中存在的问题就轻而易举地解决了。即使攻击者一直在网络中窃听了所有消息，但是由于 Jim 的解密密钥不用通过网络进行传输，而加密密钥不能解密加密的信息，攻击者将不能获得任何有用的信息。事实上，在实际应用中，通信双方甚至不用相互发送公钥，因为可以从一个公钥数据库中获取任何人的公钥，这样即使原来不认识的人也能轻而易举地建立安全的通信信道。

虽然上述过程解决了对称加密算法存在的问题，但是公开密钥算法也有其缺点，概括起来有如下两个。

- 公开密钥算法比对称加密算法慢。一般来说，对称加密算法加密信息的速度是公开密钥算法加密同样的信息的速度的近千倍。虽然有理由认为我们所拥有的计算机的速度会增加以减轻速度慢的影响，但是我们所需要的带宽也会不断增加。
- 公开密钥算法抵抗选择明文攻击的能力较低。这是因为公开密钥算法的加密密钥是公开的，攻击者可以尝试加密各种明文以跟获取的密文相对照来确定明文。如果明文的取值范围是有限的，这种攻击就很有效。而对称加密算法因为密钥都是秘密的，所以不容易受到这种攻击。

在现在大多数实用的安全通信协议中，综合了对称加密算法和公开密钥算法的优点，同时使用这两种算法来完成通信过程。在这种通信协议中，一般使用公开密钥算法保护和协商会话密钥。这里的会话密钥事实上一般就是对称加密算法要使用的加密密钥，用来加密真正要传输的数据信息。我们称这种既使用公开密钥算法，又使用对称加密算法的安全通信协议为混合密码通信系统或混合密码系统。Tom 跟 Jim 使用混合密码通信系统进行安全通信的步骤如下：

① Tom 和 Jim 通过网络协商要使用的公开密钥算法和对称加密算法；

② Jim 通过网络将自己的公开密钥（加密密钥）发给 Tom；

③ Tom 产生一个会话密钥，并使用 Jim 的公开密钥使用协商好的公开密钥算法加密该会话密钥；

④ Tom 通过网络传输使用 Jim 公开密钥加密后的会话密钥给 Jim；

⑤ Jim 使用自己的私人密钥（解密密钥）解密出 Tom 发过来的会话密钥；

⑥ Tom 使用会话密钥作为加密密钥通过协商好的对称加密算法加密要发送的信息；

⑦ Tom 通过网络将使用会话密钥加密的信息发送给 Jim；

⑧Jim 使用会话密钥作为解密密钥通过协商好的对称加密算法解密 Tom 发送过来的

信息。

混合密码通信系统将公开密钥算法应用于密钥分配解决了对称加密算法无法解决的密钥管理难题。在该系统中，即使攻击者从头到尾一直监听整个通信过程，他也难以轻而易举地获取加密密钥（会话密钥）以获取有用的信息。事实上，为了防止会话密钥泄漏，通常会话密钥在使用之后就立刻销毁，并且在通信过程中根据需要进行更换，增加了会话密钥的安全性。当然，如果私人密钥泄漏了，那么对整个密码系统来说将是灾难性的，但是因为它不用通过网络传输，并且只有在加密会话密钥的时候才使用，其泄漏的风险还是比较小的。

### 2.4.2 单向散列函数

单向散列函数又称为信息摘要函数或压缩函数，是现代密码学的重要函数类型，也是许多密码通信协议中使用到的函数。这节将简单介绍单向散列函数的性质和应用。

单向散列函数是单向函数的特殊形式。所谓单向函数，是指计算很容易，求逆却非常困难或实际操作上不可能的函数。也就是说假如函数 $f$ 是单向函数，对于已知的任意数 $x$，我们可以轻易地得到值 $f(x)$，但是反过来，如果知道 $f(x)$，想要得到 $x$ 是非常困难的或几乎不可能的。

与单向散列函数相关的另外一个概念是散列函数。所谓散列函数，是指可以将输入长度可变的字符串映射成一个固定长度的字符串的函数，这个固定长度的字符串称为散列值。事实上，散列函数是一个多对一的映射函数，理论上不能保证散列值相同的两个字符串是相同的。

单向散列函数结合了单向函数和散列函数的特点，是指在一个方向上工作的散列函数。也就是说，从输入值得到散列值很容易，但使散列值等于某一个值却是非常困难的事情。好的单向散列函数应该具备无冲突的特点。所谓无冲突，并非真正意义上的无冲突，它实际上是指两个不同的输入产生相同散列值的可能性非常小。一般来说，一个好的单向散列函数如果其输入值有一位改变，那么将引起其散列值一半数据位的改变。所以，已知一个散列值，要找到一个输入值，使得其散列值等于已知散列值在计算上是不可能的。

单向散列函数在密码学中的应用非常广泛，基本上来说可以分为不加密散列值和加密散列值的应用。不加密的散列值一般用作文件指纹或数据指纹。在很多情况下，某人可能声称他拥有跟你一样的一份机密文件或数据，需要你跟他合作。他不想把文件或数据传给你，但是需要向你证明他的话是真的，这时候他可以给你发送该文件的散列值，然后你将自己计算的散列值跟发送过来的散列值作比较，如果相同，就基本上可以肯定他也拥有该文件。

加密的散列值一般称为消息鉴别码（MAC），它的功能跟单向散列函数基本一样，只是只有拥有解密密钥的人才能验证该信息。单向散列函数还应用在数字签名中以提高运算的速度。

### 2.4.3 数字签名

#### 1. 数字签名的基本问题

对于签名我们不会陌生，我们经常会被要求或主动要求在文件、合同及协议等资料上

签名，当然，这种签名是手写签名。有时候替换的签名手段是加盖印章，这个印章一般是通过权威机构验证的，比如公司印章是工商局验证的。签名在现实生活中之所以如此常用，是因为它具备以下的性质。

- 签名值得信任。如果签名者对一份文件签名，他应该是经过慎重的考虑，同意文件的内容才签名的，这使得接受文件的人有理由信任该签名。事实上，这是对签名这一操作的共同约定。
- 签名不能伪造和抵赖。通过字体或印章，可以证明是签字者本身而不是别人在该文件上进行了签名。这样一则可以防止别人伪造签名，一则可以防止签名者事后抵赖。
- 签名不能重用。签名是跟被签名的文件在一起的，一般来说，很难或可以看作不能将该签名转移到别的文件上去作为别的文件的签名。
- 被签名的文件不能改变。文件被签名后，文件的内容一般来说是不能改变的，比如对多页的文件，我们可以采用骑缝签名或印章的形式保证文件不被改变。

事实上，上述的描述只是我们对签名所实现功能的期待，真正的签名对上述的任何一条性质都不能说是完全满足的。比如可以伪造签名和改变签名后的文件内容，但是这些实现起来都有一定的困难，并且很容易被发现。所以我们愿意使用签名这种形式，因为它至少一定程度上增加了欺骗和抵赖等恶意行为的难度。

对于存储在计算机等数字设备中的数字文件，签名所能提供的那些功能和性质更是非常必要的，因为数字文件易于被修改却不易被发现。我们称计算机中使用的实现签名的相应功能和性质的技术为数字签名技术。

但是要在数字世界中使用数字签名，却存在两大困难，一个是法律上的困难，一个是技术上的困难。在现实世界中，签名或印章是受到法律承认和保护的，我们可以很信任地使用签名。但在数字世界，数字签名对于政府机构和公众来说，都还是陌生的概念，更别说取得法律支持了。没有法律的支持，数字签名就没有太大的意义。近来情况慢慢好转，在美国，已经有相应的数字签名法案，在中国数字签名法案也已经开始试行，但是这些都还远远不够。不过本书更关心的还是数字签名的技术实现问题。

因为数字文件的内容和信息容易被复制和修改的原因，数字签名实现起来比现实世界面临更多的困难。

- 难以实现信任关系。因为数字签名不像手写签名，它只能表述为数字信息，怎么将特定的数字信息跟特定的个人建立可信任的联系是一个首先要解决的问题。
- 难以防止伪造，它不能简单地在文件上签字，这样数字信息很容易被复制到别的文件上使用。
- 难以保证被签名的文件内容不被修改，因为数字文件很容易修改而不留痕迹。

幸运的是，现代密码学基本上解决了这些难题，使得数字签名成为可能，下面介绍密码学的各种数字签名解决方案。

### 2. 基于对称加密算法的数字签名

基于对称加密算法的数字签名方案需要一个第三方的参与，这个第三方是作为签名者和验证者都信任的仲裁者存在的。一个数字签名系统起作用的范围称为域，仲裁者能够与域内的任何一个用户进行通信，而且跟各个用户都分别共享着不同的密钥。假如 Jim 要验

证 Tom 的签名，仲裁者为 Server，Server 跟 Jim 的共享密钥是 $K_{Jim}$，跟 Tom 的共享密钥是 $K_{Tom}$，则整个过程如下。

① Tom 首先用 $K_{Tom}$加密要发送给 Jim 的消息，并发送给 Server。

② Server 收到 Tom 发送的消息后，使用 $K_{Tom}$解密该消息。

③ Server 将解密的消息、Tom 加密的消息及自己收到 Tom 信息的声明一起使用 $K_{Jim}$加密并发送给 Jim。

④ Jim 用 $K_{Jim}$解密 Server 发来的消息，就可以得到 Tom 的消息和 Tom 加密的消息，并得到 Server 的证明消息来自 Tom 的声明。

事实上，通过分析可以看到，这个协议基本上达到了跟手写签名同样的效果，其特点如下。

- 签名是值得信任的。因为 Server 是一个公正的和可信任的仲裁者，而且因为只有它跟 Tom 共享私钥，所以它能够确定发送消息的人就是 Tom。所以 Server 的声明对于 Jim 来说就是可信的，能够证明确实是 Tom 发送的消息。
- 签名不能伪造和抵赖。因为只有 Tom 和 Server 知道 $K_{Tom}$，所以别人不可能使用 $K_{Tom}$加密消息，也就不能伪造 Tom 的签名。Server 是大家都信任的，不会伪造签名。因为别人不能伪造签名，而且 Server 会记录和证明 Tom 发送过该消息，所以 Tom 不能抵赖自己的签名。
- 签名不能重用。如果 Jim 怀有恶意，想把 Server 对 Tom 发送的消息的证明信息附加到另外一个对他有利的消息上，Tom 可以申请 Server 仲裁。Server 会要求 Jim 提供消息和他收到的 Tom 加密的消息，然后使用他跟 Tom 共享的私钥加密 Jim 提供的消息，再和 Jim 提供的 Tom 加密的消息对比，就可以发现 Jim 的欺骗行为。Jim 没有 Tom 的私钥，不可能伪造一个跟 Server 使用 Tom 私钥加密假信息得到的信息相同的加密信息。
- 签名后的文件不能修改。因为文件信息的改变会导致加密后信息的改变，所以使用上面相同的办法，可以防止 Jim 恶意修改 Tom 签名后的信息。

通过上面的分析可以看到，这个基于对称加密算法的数字签名方案在理论上是可行的。但是由于这个方案是完全基于可信任的仲裁者，在实际应用中会产生许多问题，主要存在如下的缺点。

- 对仲裁者 Server 处理能力要求高。因为所有需要进行数字签名的域内用户都必须和 Server 通信，在一个用户数量比较大的系统中，对 Server 的处理能力的要求将是非常高的，这往往会成为通信带宽的瓶颈。
- 信任关系过度集中于 Server。在该数字签名系统中，Server 与用户共享了所有的私钥，是整个域的信任基础。所以要求 Server 必须是高度安全的，不能出现任何错误，所有私钥都不能泄漏，程序也必须保证是安全的。而且一旦出现任何安全问题，不但会影响以后的文件，以前所有签署的文件都可能要作废。在网络上建立这样一个高度安全的第三方在实际操作上是不容易的。

### 3. 基于公开密钥算法的数字签名

公开密钥算法用于数据加密时，一般来说加密密钥是公开的，而解密密钥是保密的，并且具有根据加密密钥不能推算出解密密钥的特点。当公开密钥算法用于数字签名时，要求就有所不同。因为数字签名是要求所有的人都能够验证的，所以要求用于数字签名的公

开密钥算法的解密密钥是公开的，而用于签名的加密密钥是保密的。为了安全和防止签名伪造，同时要求根据解密密钥不能推算出加密密钥。并非所有公开密钥算法都能满足用于数字签名的这些要求，目前常用于数字签名的算法有 RSA 和 DSA 等。事实上，用于数字签名的公开密钥算法有些也不能用于数据加密，如 DSA 算法就是这样一个专门用于数字签名的公开密钥算法。

使用公开密钥算法的数字签名方案简单而有效，其基本的步骤如下。

① Tom 使用自己的私人密钥加密文件，从而实现对文件的签名并传送给 Jim。

② Jim 用 Tom 的公开密钥解密文件，从而实现签名的验证。

这个签名方案非常简单，不需要第三方的参与，并且它也可以满足我们对数字签名期待的性质和功能。

- 签名值得信任。当 Jim 使用 Tom 的公钥解密信息成功时，他就可以确定是 Tom 签发了这个文件，因为别人不可能拥有 Tom 的私钥。
- 签名不能伪造和抵赖。因为只有 Tom 知道自己的私钥，所以别人不可能伪造他的签名。同时，Tom 也不能抵赖自己的签名，Jim 可以在必要的时候向第三者出示 Tom 签名的文件并使用 Tom 的公钥验证。
- 签名不能重用。因为私钥对整个文件进行加密操作，使用另外的文件再次签名会产生不同的签名值，这样很容易就可以戳穿 Jim 的骗局。
- 签名后的文件不能改变。使用上述的相同方法，就可以发现文件是否改变。

因为公开密钥算法加密数据的速度一般比较慢，尤其对于大的文件，这种速度有时候是不能忍受的，为了解决这个问题，实际中通常采用单向散列函数和公开密钥算法相结合的数字签名解决方案。其基本步骤如下。

① Tom 对要签名的文件使用单向散列函数，然后使用自己的私钥将得到的散列值加密，最后将文件和加密后的散列值发送给 Jim。

② Jim 收到 Tom 发来的文件和加密的散列值后，用同样的单向散列函数算法对该文件作处理得到新的散列值，然后使用 Tom 的公钥将 Tom 发送过来的加密散列值解密，并与自己计算得到的散列值对比，如果一致，就证明文件是 Tom 做了签名的。

这样的一个协议不但提高了计算速度，还带来很多其他好处。第一个好处是文件和签名的保存可以分开，只要记住它们的对应关系就可以了，这给管理带来了很大的方便。第二个好处是减少了签名系统尤其是签名接收者需要的存储空间，很多时候，比如中央数据库，可能只要保存文件的签名信息而不需要保存庞大的文件，当发生争执的时候，中央数据库只要通过文件散列值就可以解决争端。

### 4. 数字签名的更多知识

虽然使用公开密钥算法能够很好地解决数字签名的基本难题，但在一些要求更高的情况下，可能还需要提供更严格的解决方案。对这些问题，本节只是简单地介绍，要更详细地了解有关方面的细节，读者可以查阅更加理论性的密码学书籍。

前面我们讨论数字签名的抗抵赖功能时，基于一个隐性前提，即私钥跟签名者个人是绝对等同的。但是事实上却不是如此，毕竟私钥与其代表的人是可以分离的两个不同的实体，数字签名技术其实是没有办法解决这个问题的。利用该问题签发者可以对自己的签名进行抵赖。签名者对文件签名后，故意把文件在公共的场所公布，这样他就可以以私钥丢

失有人冒充自己为理由否认执行了上述文件的签名操作。目前没有能够完全解决这个问题的措施或技术，但是有些能够增加限制的措施，比如将私钥隐藏在不能读出的硬件模块中，用于密码用途的智能卡就是这样的设备。采用时间标记也可以一定程度上限制这种抵赖行为，至少可以确保以前旧的签名依然有效。事实上，时间标记在实际的数字签名应用中经常采用，它可以防止签名重用和信息重放等攻击。

多重签名也是实际应用中可能经常遇到的问题，一般都使用单向散列函数，多个签名者可以分别对文件的散列值进行签名，然后再一起发给接收者。如果不使用单向散列函数，问题会麻烦得多。一种可能的方案是签名者分别对原来的文件签名，这样得到的签名文件内容将是原来的 $n$ 倍，$n$ 等于签名者的个数；另一种可能方案是后一个签名者在前一个签名者的签名基础上继续进行签名，直到所有签名者都完成签名，但是这样同样带来一个问题，即验证的时候必须先验证最后一个签名者的签名，然后往前类推，这带来了应用上的限制。

## 2.5 密钥交换协议

在实际的密码应用协议中，通常要生成一个临时的密钥用于对信息进行加密，这个密钥称为会话密钥。会话密钥一般只在通信期间使用，有时候一个通信会话还会使用多个会话密钥以增加安全性，而这些会话密钥在通信完成之后都会被销毁。怎么安全地完成通信双方对会话密钥的协商是非常重要的问题，通常解决这个问题的协议就称为密钥交换协议。

### 2.5.1 基于对称加密算法的密钥交换协议

基于对称加密算法的密钥交换协议需要可信任的仲裁者的存在，作用域内任何一个使用这个协议的用户都分别跟仲裁者预先共享一个私钥。假设仲裁者名字为 Server，Tom 和 Jim 执行的密钥交换协议步骤如下。

① Tom 向 Server 申请一个跟 Jim 进行安全通信的密钥。

② Server 产生一个会话密钥，分别使用 Tom 的私钥和 Jim 的私钥加密成两个副本，并发送给 Tom。

③ Tom 收到 Server 返回的信息后，解密自己的会话密钥信息，同时将使用 Jim 私钥加密的会话密钥发送给 Jim。

④ Jim 将收到的信息解密得到会话密钥，Tom 跟 Jim 可以使用会话密钥进行安全通信。

跟基于对称加密算法的数字签名协议一样，该协议的缺点是需要第三方作为仲裁者，整个体系的安全基于仲裁者的安全，如果仲裁者受到攻击并被控制，那么攻击者就能取得所有私钥并随意监视网络中传输的数据。此外，仲裁者可能成为通信的瓶颈，因为系统中所有要求建立安全通信的用户都需要跟仲裁者联系。基于这些原因，安全性要求高或者用户数量较多的组织就不适合使用该方案。

### 2.5.2　基于公开密钥算法的密钥交换协议

我们在前面讨论基于公开密钥算法的安全通信协议的时候已经介绍过使用公开密钥算法进行密钥交换的协议。事实上，这也是公开密钥算法的主要用途之一。这里只作简单的回顾。使用公开密钥算法进行密钥交换的协议的简单步骤如下。

① Tom 产生会话密钥，使用 Jim 的公钥加密该会话密钥并发给 Jim。

② Jim 使用自己的私钥解密收到的消息得到会话密钥，双方使用该会话密钥进行安全通信。

这里忽略了 Tom 是怎么得到 Jim 的公钥的步骤，因为公钥是公开的，所以有很多方法可以得到 Jim 的公钥，可以在执行上述两个步骤之前由 Jim 发给 Tom，也可以由 Tom 从公开的公开密钥数据库中取得，甚至有时候还可以从第三方取得 Jim 的公钥。可以看到，公开密钥算法是很适于解决密钥交换和分发问题的。

### 2.5.3　高级密钥交换协议

#### 1. 中间人攻击

公开密钥交换算法在理论上很好地解决了密钥分发和交换的问题，但事实上，由于数据在网络中传输，聪明的攻击者有更多的机会发起攻击，中间人攻击就是其中一种。中间人攻击者不同于被动的窃听者，他主动发起攻击，并具备截取通信双方的网络信息，同时使用新信息替换这些信息的能力。这是可以实现的，比如他可以伪装成一个路由服务器，最新版本的 dnsniff 就具备使用中间人攻击技术攻击 SSL 协议的功能。中间人攻击的步骤一般如下。

① Tom 将自己的公钥传给 Jim，攻击者截获 Tom 的公钥并用自己的公钥代替 Tom 的公钥发给 Jim。

② Jim 将自己的公钥传给 Tom，攻击者截获 Jim 的公钥并用自己的公钥代替 Jim 的公钥发给 Tom。

③ Tom 和 Jim 开始相互用“对方”的公钥加密信息并通过网络传输给对方，事实上，经过上面两步，使用的公钥（加密密钥）实际上都是攻击者的，所以攻击者只要简单地监视两者的通信，并用自己的私钥解密双方发送的消息就可以得到全部信息，甚至可以更改信息。

有时候可能 Tom 和 Jim 不会相互交换密钥，他们可能会从一个公钥数据库中取得对方的公钥，这时候攻击者的攻击目标首先就是公钥数据库，如果他能够用自己的公钥取代通信双方的公钥，他也能够成功。

如果通信双方早就相互拥有对方的公钥，攻击者就不能进行中间人攻击，但因为这种情况毕竟是特殊的，所以中间人攻击应该受到重视。中间人攻击之所以能够实施，是因为在上述的通信协议中通信双方没有办法发觉已经或正在遭受攻击。

#### 2. 连锁密钥交换协议

连锁密钥交换协议是解决中间人攻击的一个巧妙的协议，当然不是一个能够完全防止中间人攻击的协议，只是增加了中间人攻击的难度。它基于这样一种技术，即拥有加密消

息的一半是没有意义的，不能进行解密。这种技术是存在的，最简单的例子就是分组加密算法，因为分组加密算法中消息的解密依赖初始向量，后面部分消息的解密要依靠前面的消息，所以可以先发送后面部分的消息，然后再发送前面部分的消息，这样，即使先得到消息的后面部分也是不能解密信息的。连锁密钥交换协议的基本步骤如下。

① Tom 和 Jim 相互交换自己的公钥。

② Tom 用 Jim 的公钥加密要发送的消息，并发送加密后消息的一半给 Jim。

③ Jim 用 Tom 的公钥加密要发送的消息，并发送加密后消息的一半给 Tom。

④ Tom 在收到 Jim 的第一部分消息后，将加密消息的另一半发送给 Jim。

⑤ Jim 在收到 Tom 的另一半消息后，将自己加密消息的另一半发送给 Tom，并将收到的两部分消息合起来，用自己的私钥解密消息。

⑥ Tom 收到 Jim 的另一半消息后，将两部分消息合在一起，然后用自己的私钥解密。

连锁协议最重要的特点是，双方发送消息的时候是相互连锁的，也就是说，通信一方只有收到另一方发送的消息的一部分后才会继续协议，否则就会等待或终止协议。这样的信息发送连锁，使得中间的攻击者没有办法进行窃听式的中间人攻击。下面我们详细分析一下加入中间人攻击会出现的情况。

攻击者可以在通信中截获通信双方的公钥，并用自己的公钥替代通信双方的公开密钥。但是当攻击者在第②步截获到消息的时候，虽然该消息是用他的公钥加密的，但是因为只有一半，所以至少暂时没有办法解密，同时因为协议是连锁的，所以他这时候只有两个选择，要么伪造一个假消息给 Jim，要么终止攻击行为，我们假设他有勇气继续碰运气。在第③步的时候，他存在同样的困难选择，只好再伪造一个假消息。这样，当他在后续的两步收到可以解密的全部消息时，他已经没有办法修改以前伪造的消息。当然，这样他还可以用伪造的消息假装跟通信双方继续进行交谈，但是这样显然比窃听获取有用的信息更加困难。

连锁密钥协议虽然增加了中间人攻击的难度，但并没有从根本上杜绝中间人攻击行为，使用证书的密钥交换协议能更好地解决这一问题。

### 3. 使用证书的密钥交换协议

分析基本的基于公开密钥算法的密钥交换协议之所以容易受到中间人攻击，是因为公开密钥没有跟个人的身份证明联系在一起，所以攻击者可以随意地用任何公开密钥替代通信双方的公开密钥。假设我们使得公开密钥跟身份证明是紧紧联系在一起的，那么攻击者就难以通过简单的替换公开密钥的方式冒充通信任一方。数字证书或者说数字签名技术可以将公开密钥跟某个人的身份证明关联在一起。

在这种高级密钥交换协议中，需要一个可信任的第三方，这个第三方能够验证任何想通过验证的用户，他是绝对可信的。他用自己的私钥对一个用户的公钥和用户个人信息进行签名，形成一个数字证书。可信任的第三方自己的公钥是任何用户都可以取得的，这样任何用户都可以通过验证这个第三方的数字签名来确定通信对方发送过来的公钥是不是真的是他自己的公钥。这样，攻击者想要冒充第三方就很困难，因为他没有可信任的第三方的私钥，没有办法伪造他的签名。

当然，这种使用证书的密钥交换协议很依赖可信任的第三方，如果可信任的第三方受到攻击，整个通信系统的安全性将面临威胁，但是这种攻击显然比中间人攻击难得多。事

实上，即便攻击者取得了对第三方私钥的控制权，也难以对以前签发的公钥证书进行更方便的攻击。

### 2.5.4　不需要密钥交换协议的安全通信

这里所说的不需要密钥交换协议的安全通信，并非真的不需要进行密钥的协商，只是指在信息传输之前，可以不进行专门的密钥交换步骤，而是直接将密钥和传输信息一起发送给对方。这样的协议在只发送简单的消息而不需要太多信息交换的时候显得特别有效。下面是其基本的步骤。

① Tom 产生随机会话密钥，并用会话密钥加密要发送的信息。

② Tom 使用 Jim 的公钥加密会话密钥。Jim 的公钥可以从公钥数据库中取得，或者 Tom 已经存有 Jim 的公钥。

③ Tom 将使用会话密钥加密的信息和使用 Jim 公钥加密的会话密钥通过网络传送给 Jim。

④ Jim 收到 Tom 发送的信息，先用自己的私钥解密加密的会话密钥。

⑤ Jim 使用解密得到的会话密钥解密加密的信息。

可以看到，这种协议中，如果 Tom 要给 Jim 发送一个信息、文件或者 E-mail，只要发送一次就可以了，而不用建立 TCP 连接等待 Jim 的响应。这在许多实际应用中是很好的方式。比如在消息广播中，接收者可能是很多人，发送者不一定期待接收者的回应，就可以采用下面的没有密钥交换协议的安全通信协议。

① 发送者产生一个会话密钥，并使用该会话密钥加密要发送的信息。

② 发送者使用 Tom、Jim 或更多的用户的公钥分别加密会话密钥。

③ 发送者将加密的信息和使用各个接收者的公钥加密的会话密钥分别发给 Tom 和 Jim 等人。

④ Tom 和 Jim 等人将收到的信息是用自己的公钥加密的会话密钥和使用会话密钥加密的信息，他们首先使用自己的私钥解密加密的会话密钥，然后使用会话密钥解密加密的信息。

## 2.6　鉴 别 协 议

鉴别是指确定一个人的身份，即确定一个人是否是他所宣称的身份。在现实世界中，鉴别是非常重要的，这是进行安全交易的基础。比如在银行，营业员需要鉴别你是否是你出示的证据所代表的人。在数字世界，鉴别一样重要，并且从一开始就提出了各种解决措施。

### 2.6.1　基于口令的鉴别协议

口令是最常用和最常见的鉴别协议。当登录一台重要的计算机时，它会要求输入用户名和密码，用户名代表你的身份，口令起鉴别作用，如果你是该用户名所代表的人，你就应该能够输入正确的密码。

最简单的口令鉴别系统是在服务器上或者验证者本身存有你的口令，然后将你输入的

口令跟数据库或文件中的口令对比，如果一致，那么就通过鉴别。事实上这有很大的安全隐患，如果有办法取得该口令数据库或文件的读取权限，那么就可以轻而易举地冒充每一个用户，使得鉴别系统失效。由于一般鉴别应用程序运行的主机都是联网的，所以存在遭受攻击的诸多危险，更有甚者，如果管理员想要冒充这样一个鉴别系统中的用户，将是非常容易的。

为了增加口令文件或口令数据库的安全性，我们需要密码学的帮助。目前来说，有两种基本的方法：一种是将口令采用对称加密算法进行加密后在数据库保存，然后在鉴别的时候对用户输入的口令作相同的运算，如果得出的值跟数据库保存的值一致，那么鉴别就通过；一种是使用单向散列函数替代对称加密函数，其他过程都基本一样。这两种鉴别协议都可以表述如下。

① Tom 将自己的用户名和密码传送给计算机 Server。

② Server 将密码或更多的其他信息（比如用户名）使用特定的算法进行计算。

③ Server 根据 Tom 用户名查找数据库中相应的字段数据与计算结果比较，如果相同，鉴别通过。

可以看到，在该改进的鉴别协议中，服务器保存的不再是明文的口令，而是密文值，这样即使服务器遭受攻击，也使获取用户口令的难度增大了。

但是，基于口令的鉴别依然是非常脆弱的，其特点总的来说有如下一些。

- 容易受到字典攻击。因为一般用户为了口令方便易记，会采用有意义或常用的字符串作为口令，这样，即便服务器存放的是口令的加密值（或散列值）而不是明文，攻击者也可以通过常用口令列表的字典式攻击有效获取大部分用户的口令。
- 容易遭受网络窃听的攻击。在上述的口令鉴别协议中，口令是以明文的形式在网络中传输的，这样非常容易被有意的攻击者获取。
- 口令容易泄漏。一般用户为了使用方便，会选取简单而且长度短的口令，这样非常容易泄漏。一个有意的攻击者可能只要偷偷在你身后观测几分钟就能得到你的口令。
- 口令持有人的身份容易冒充。攻击者只要取得某个用户的口令，就可以完全欺骗基于口令的鉴别系统，冒充用户的身份。

### 2.6.2 基于公开密钥算法的鉴别协议

为了克服基于口令的鉴别协议的缺点，在鉴别协议中采用公开密钥算法是一个不错的选择。在基于公开密钥算法的鉴别系统中，主机只保存各个用户的公开密钥，每个用户保存自己的私人密钥。这里使用的公开密钥算法是用于签名的，公开密钥用来解密信息，可以对任何人公开，也不能推算出用来加密的私人密钥，所以这不会带来安全性问题。基本的协议步骤如下。

① Tom 向鉴别主机 Server 发出鉴别请求。

② Server 产生一个随机数，并发送给 Tom。

③ Tom 用自己的私人密钥加密收到的随机数和自己的名字，然后发送给 Server。

④ Server 收到 Tom 的消息后，查找 Tom 在数据库中的公开密钥，并使用该公开密钥解密 Tom 发送来的消息，如果得到的随机数跟 Server 原来发出去的相同，鉴别通过。

上述这个协议是 Server 对 Tom 进行了鉴别，如果 Tom 同样需要对 Server 的身份进

行鉴别，也可以采用类似的步骤。

这个协议中，没有人能够冒充 Tom，因为只有 Tom 有自己的私人密钥，如果没有私人密钥，基本上不可能通过 Server 的鉴别。

因为使用了公开密钥算法，在这个协议中，系统安全是基于私人密钥安全的基础上的。私人密钥不在网络上传输，所以，任何试图通过网络窃听进行的攻击都不太可能，除了使用唯密文攻击或强力破解法。

私人密钥是一长串难以记住的随机数或字符串，一般不从键盘输入，窃听者难以像小偷似地躲在背后获得密钥。当然，私人密钥能否安全存放就成了这个鉴别系统的一个重要的问题。为了解决这个问题，有很多改进的方案。一种方案是将私人密钥封装在不能读取出密钥数据的模块中，比如智能卡和 USB Key，这种模块本身具有计算和存储功能。用户每次登录系统进行鉴别的时候都需要将这些硬件设备接入到计算机中，也就是说，攻击者只有得到了用户的这个标识性的硬件（通常会称为令牌），才可能对鉴别系统进行欺骗性的攻击。实际的系统中，还会在硬件本身加入口令验证的功能，这样攻击者只有同时取得这个令牌和口令才能对鉴别系统发起攻击。增加了令牌的鉴别协议的基本步骤如下。

① Tom 在自己使用的计算机中接入自己的令牌 Smart，然后向鉴别主机 Server 发出鉴别请求。

② Server 产生一个随机数，并发送给 Tom。

③ Tom 向存储私人密钥的设备 Smart 发送 Server 送来的随机数并输入口令。

④ Smart 设备验证 Tom 口令是否正确，如果正确，使用自己的计算设备在内部用 Tom 的私人密钥加密收到的随机数和 Tom 的用户名，然后发送给 Server（或者先交给 Tom，Tom 再发送给 Server）。

⑤ Server 收到 Tom 的消息后，查找 Tom 在数据库中的公开密钥，并使用该公开密钥解密 Tom 发送来的消息，如果得到的随机数跟 Server 原来发出去的相同，鉴别通过。

有时候，为了更高的安全性，需要防止聪明的攻击者对协议发起攻击，就需要采用下面更复杂的方式。

① Tom 产生一个随机数，并使用自己的私人密钥进行加密计算，将结果发送给验证主机 Server。

② Server 产生一个随机数，并发送给 Tom。

③ Tom 根据自己产生的随机数和收到的随机数用他的私人密钥进行加密计算，将结果发送给 Server。

④ Server 从数据库中查找到 Tom 的公开密钥，并将收到的所有信息进行解密，并核对信息是否一致，如果一致，则鉴别通过。

上述协议保证了发起鉴别请求和通过鉴别的用户是同一个人，不可能是别人冒充的，从而阻止了一些针对协议的攻击。

在上述的协议中也可以加入智能卡等令牌设备以增加私人密钥的安全性。

### 2.6.3　基于对称加密算法的鉴别协议

对称加密算法一样可以构造鉴别协议，事实上，在前面介绍的基于口令的鉴别协议中，就可以使用对称加密算法。基于对称加密算法的协议相比于基于公开密钥算法的协议

有许多的缺点，比如难以防范中间人攻击，所以并不是一个好的协议，下面只是对这种协议常用的基本形式作简单的介绍。在协议执行之前，要求用户和服务器之间有一个共享的密钥，基于对称加密算法的鉴别协议的一般步骤如下。

① Tom 向 Server 发起鉴别请求。

② Server 产生一个随机数 RNDa，并发送给 Tom。

③ Tom 收到 Server 的随机数 RNDa，接着自己产生另一个随机数 RNDb，将用户名、RNDa 和 RNDb 用设定的算法和预先共享的密钥加密，然后将得到的秘密值和随机数 RNDb 一起发给 Server。

④ Server 将用户名、RNDa 和收到的 RNDb 一起用设定的算法和预先共享的密钥加密，比较计算得到的秘密值和收到的秘密值是否一致，如果一致，Tom 的身份鉴别就通过了。

如果 Tom 需要鉴别 Server 的身份，也就是说要求一个双向鉴别的协议，那么可以随后执行下面两个步骤。

⑤ Server 将自己的用户名和 RNDb 用设定的算法和预先共享的密钥加密，将得到的秘密值发送给 Tom。

⑥ Tom 收到秘密值后，将 Server 用户名和 RNDb 用设定的算法和预先共享的密钥加密，并将得到的秘密值和收到的秘密值比较，如果一致，那么就可以认为 Server 的身份通过鉴别。

### 2.6.4 信息鉴别

我们在前面介绍的鉴别协议本质上都是对用户身份进行鉴别的协议。在实际应用中，还经常需要对信息进行鉴别，也就是说，信息接收者需要对收到的信息进行鉴别，以确定该信息是可信的，确实是由信息所声称的发送者发送的。

可以使用基于公开密钥算法的数字签名技术实现对信息的鉴别，其基本的鉴别过程如下。

① Tom 对要发送的信息用单向散列函数计算，并对得到的秘密值使用自己的私人密钥加密，然后将信息和签名的秘密值发送给 Server。

② Server 接收到信息后，对信息使用同样的单向散列函数作计算，然后用 Tom 的公开密钥解密签名的秘密值，将计算值和解密值比较，如果一致，证明信息是可信的，完成信息的鉴别。

对称加密算法也可以实现对信息的鉴别。比如当 Server 从 Tom 那里接收到用它们共享的密钥加密的信息时，它就可以认为收到的信息确实是来自 Tom 的，因为没有别人知道它们的共享密钥。但基于对称加密算法的信息鉴别存在向第三方证明的问题。也就是说，Server 没有办法向第三方证明信息确实是来自 Tom 的，因为只有它们两个人共享密钥。事实上，第三方理论上可以相信信息是来自通信的双方，但是没有办法确定是哪一方，因为它们使用的是共同的密钥。

密码学上定义了一种专门用于信息鉴别的函数，叫着信息鉴别码（MAC），一般来说它是带有密钥的单向散列函数。这可以使用单向散列函数和普通的加密算法组合实现，也可由专门用于 MAC 的密码算法实现。

## 2.7　实际应用的混合协议

前面根据使用的基本密码算法和功能的不同分门别类地介绍了各种协议和功能，但是，在实际应用中，一个协议要考虑的问题通常是各种问题的综合，要实现的功能也是多方面的。所以，实际应用中的协议通常是本章前面介绍的各种技术的部分或全部的综合，本节将对实际的协议作一些介绍，让读者理解这些技术是怎么综合在一起的。

### 2.7.1　Yahalom 协议

这个协议是基于对称加密算法的简单协议，它需要可信任第三方仲裁者的存在，假设 Tom 和 Jim 要使用这个协议进行通信，仲裁者为 Server，那么协议步骤如下。

① Tom 产生一个随机数 RNDt，与自己的名字一起发送给 Jim。

② Jim 收到 Tom 的信息后，也产生一个随机数 RNDj，然后将 Tom 的名字、RNDt 和 RNDj 一起用它和仲裁者 Server 共享的密钥加密，并将得到的加密值和自己的名字一起发送给 Server。

③ Server 收到 Jim 的消息后，首先用它和 Jim 共享的密钥解密信息，接着产生随机会话密钥，然后根据接收到的信息和会话密钥生成两个信息：第一个信息由 Jim 的名字、随机会话密钥、Jim 的随机数和 Tom 的随机数组成，并使用 Server 和 Tom 共享的密钥加密；第二个信息由 Tom 的名字和会话密钥组成，并使用 Server 和 Jim 共享的密钥加密。Server 完成这两个消息的构造后，都发送给 Tom。

④ Tom 收到这两个消息后，使用它与 Server 共享的密钥解密第一个消息，并比较 RNDt 值跟自己原来发送的值是否相同。如果相同，Tom 使用解密得到的会话密钥加密 RNDj，并将得到的加密值和从 Server 接收到的没有解密的第二个消息一起发送给 Jim。

⑤ Jim 收到 Tom 的信息后，用自己与 Server 共享的密钥解密 Server 构造（通过 Tom 发送）的信息得到会话密钥，使用会话密钥解密得到 RNDj 并确认得到的 RNDj 跟自己原来产生的是否一致，如果一致，协议完成。通信双方即完成了身份鉴别，也完成了会话密钥的交换，可以开始进行安全的通信。

### 2.7.2　Kerberos 协议

Kerberos 是目前在操作系统中使用广泛的协议，这个协议也是基于对称加密算法和可信任仲裁者的。在这个协议中，使用了时间标志，假设了所有用户的时钟都跟可信任仲裁者的时钟相同，这样可以防止重放攻击。这种协议的基本步骤如下。

① Tom 将自己的名字和 Jim 的名字发送给仲裁者 Server。

② Server 接收到 Tom 的消息后，产生一个会话密钥，然后构造两个消息：一个消息是由时间标记、使用寿命、会话密钥和 Tom 的名字组成，并使用 Server 与 Jim 的共享密钥加密；另一个消息是由时间标记、使用寿命、会话密钥和 Jim 的名字组成，并使用 Server 与 Tom 的共享密钥加密。完成两个消息的构造后，Server 将两个消息发送给 Tom。

③ Tom 接收到 Server 返回的消息后，首先用它与 Server 的共享密钥解密收到的第二

个消息，提取会话密钥。接着使用会话密钥加密自己的名字和时间标记，发送给 Jim，同时也将 Server 产生的使用 Jim 的密钥加密的信息转发给 Jim。

④ Jim 接收到信息后，首先用自己与 Server 的共享密钥解密得到会话密钥，然后用会话密钥解密得到时间标志，如果确认得到的时间标记跟当前系统时间相差不是太远，那么 Jim 可以确定 Tom 的身份，并接着对时间标记加 1 的信息使用会话密钥加密，发送给 Tom。然后它们就可以使用这个会话密钥进行安全的通信。

上面的步骤只是简单地描述了 Kerberos 协议的工作流程，实际的 Kerberos 协议要复杂一些，比如对申请不同的服务也有特别的信息标记，等等。Kerberos 协议最大的特点是加入了时间标记以防止重放攻击，但是这需要不同系统时间之间的精确同步，而这通常会出现问题。

### 2.7.3 Neuman-Stubblebine 协议

Kerberos 协议在协议中加入时间标记是防止重放攻击不错的措施，但是，由于其采用的时间是不同系统的时间，所以需要系统时间同步，这通常很难做到，也很容易出问题。Neuman-Stubblebine 协议解决了这个问题，它使得时间只是一个系统的时间，而不需要进行同步。该协议也是基于对称加密算法的协议，可以看作 Yahalom 协议的改进版本，其基本步骤如下。

① Tom 产生一个随机数 RNDt，与自己的名字一起发送给 Jim。

② Jim 收到 Tom 的信息后，也产生一个随机数 RNDj，然后将 Tom 的名字、RNDt 和一个时间标记一起用他和仲裁者 Server 共享的密钥加密，并将得到的加密值、自己的名字和 RNDj 一起发送给 Server。

③ Server 收到 Jim 的消息后，首先用它和 Jim 共享的密钥解密信息，接着产生随机会话密钥，然后根据接收到的信息和会话密钥生成两个信息：第一个信息由 Jim 的名字、随机会话密钥、Jim 产生的时间标记和 Tom 的随机数组成，并使用 Server 和 Tom 共享的密钥加密；第二个信息由 Tom 的名字和会话密钥和时间标记组成，并使用 Server 和 Jim 共享的密钥加密。Server 完成这两个消息的构造后，把这两个消息和 RNDj 都发送给 Tom。

④ Tom 收到这两个消息后，使用它与 Server 共享的密钥解密第一个消息，并比较 RNDt 值跟自己原来发送的值是否一样。如果相同，Tom 使用解密得到的会话密钥加密 RNDj，并将得到的加密值和从 Server 接受到的没有解密的第二个消息一起发送给 Jim。

⑤ Jim 收到 Tom 的信息后，用自己与 Server 共享的密钥解密 Server 构造（通过 Tom 发送）的信息得到会话密钥，并确认得到的 RNDj 和时间标记跟自己原来产生的是一致的，并且时间跟目前系统时间的差值保持在一个范围内。如果这些条件满足，协议完成。通信双方即完成了身份鉴别，也完成了会话密钥的交换，可以开始进行安全的通信。

在这个协议中，Jim 只要验证自己产生的时间标记，所以不存在时间同步的问题，这样就可以解决 Kerberos 协议中由于时间不同步带来的协议漏洞。

### 2.7.4 分布式鉴别安全协议

前面介绍的协议都是基于对称加密算法的，下面介绍基于公开密钥算法的一些实际应

用协议。

分布式鉴别安全协议（DASS）是一种同时使用了公开密钥算法和对称加密算法的协议，提供双向的鉴别和密钥交换。在这种协议中，也需要可信任第三方仲裁者的参与，它保存了所有用户的公开密钥。下面是基本的协议流程。

① Tom 想跟 Jim 通信的时候，发送 Jim 的名字给仲裁者 Server。

② Server 收到消息后，查找数据库中 Jim 的公开密钥，并使用自己的私人密钥对 Jim 的名字和公开密钥签名，然后将公开密钥和签名一起发送给 Tom。

③ Tom 收到 Server 返回的信息后，首先用 Server 的公开密钥验证它对 Jim 公开密钥的签名，以确保收到的公开密钥确实是 Jim 的。然后，Tom 产生一个会话密钥和一对临时的公开密钥和私人密钥对，用会话密钥加密时间标记，用跟 Server 保存的公开密钥相对应的私人密钥对生命周期、自己的名字和临时公开密钥签名。之后，用 Jim 的公开密钥加密会话密钥，并用临时私人密钥签名。最后将所有这些信息发送给 Jim。

④ Jim 收到 Tom 的消息后，首先给 Server 发送一个仅由 Tom 名字组成的信息。

⑤ Server 收到消息后，查找数据库中 Tom 的公开密钥，并使用自己的私人密钥对 Tom 的名字和公开密钥签名，然后将公开密钥和签名一起发送给 Jim。

⑥ Jim 收到 Server 返回的信息后，首先用 Server 的公开密钥验证它对 Tom 公开密钥的签名，以确保收到的公开密钥确实是 Tom 的。然后它使用从 Server 得到的 Tom 的公钥验证 Tom 发来的信息的签名并解密得到 Tom 的临时公开密钥，用这个临时公开密钥验证签名并随后用自己的私人密钥解密得到会话密钥。最后用得到的会话密钥解密加密的时间标记以确保这是当前的信息，防止重放攻击。

⑦ 如果 Tom 需要继续对 Jim 进行鉴别，那么 Jim 就用临时会话密钥加密一个新的时间标记并发送给 Tom。

⑧ Tom 用会话密钥解密这个消息以确保是当前的消息。

可以看到，事实上，DASS 协议的时间标记也是不同系统的时间，需要以时间的同步为前提，这也给协议带来一定的漏洞。

在 DASS 协议中，在协议中还使用到了通信的第三方，事实上，有些协议可以在通信的时候不涉及第三方，这种协议一般使用数字证书，这是很好的一种方式。SSL 协议的数字证书验证方式就采用了这样的协议。当然，这种协议的前提是需要一个可信的证书签发机构。

有许多这样的基于公开密钥算法的协议，它们都是利用公开密钥算法的特点，并使用了对称加密算法、单向散列函数、数字签名等密码算法。有兴趣的读者可以查阅相关的文献和资料。

## 2.8　本 章 小 结

本章首先介绍密码学的基本功能和设计目标，对加密、鉴别、完整性和抗抵赖的概念进行了详细形象的解释。本章还介绍了一些与密码学相关的基础数学知识，了解这些基本知识，对后面章节的密码算法的理解会很有好处。

本章还介绍了密码算法的基本分类，对算法的历史、对称加密算法和非对称加密算法

的定义和性质做了概括性的阐述，并简单介绍了算法安全性的考虑因素。

本章还对密码通信协议的基础知识和各种基于密码学的通信协议作了介绍。描述了各种基于对称加密算法和非对称加密算法的密钥交换协议，以及鉴别协议的详细工作方式和考虑的问题。并对单向散列函数和数字签名等重要概念及其应用作了介绍。

最后，本章介绍了一些实际的应用协议，让读者对本章所介绍的内容在实际中的应用模型有一个形象的了解。

# 第 3 章

# 密码实现技术

## 3.1 密钥管理技术

一个好的密码通信系统依赖的不是算法的机密性，而是密钥的机密性。所以，在一个实用的密码通信系统中，安全地产生、保存、分发和使用密钥是整个系统安全性得到保证的前提。通常一个系统的密码算法和通信协议都设计得很好，但是攻击者仍然可以轻而易举地通过密钥管理攻入系统，而不必费尽心机去攻击算法和协议。

密钥管理之所以存在很多困难，因为它涉及的不仅仅是纯技术的问题，密钥的持有人可能为了别的利益或目的主动泄漏密钥，这是技术上几乎没有办法防止的。

本节将对密钥管理可能涉及的各个问题作简单的介绍，更多的是概念，而不是技术。

### 3.1.1 密钥生成

密钥生成是使用密码算法首先面临的问题，因为无论是对称加密算法还是公开密钥算法，都离不开密钥。密钥生成主要面临两个问题。

- 如何安全地生成密钥。即如何生成可信任的密钥，保证用户得到的密钥是安全的，生成密钥的机器或程序是可信的。
- 如何生成安全的密钥。安全的密钥没有统一准确的定义，但一般来说是指密钥抗猜测和抗穷举等针对密钥攻击的能力。涉及密钥长度和密钥强弱的问题。

#### 1. 安全地生成密钥

对于一个公开密钥系统，公开密钥和私人密钥对一般来说是由一个程序生成的，这个程序可以运行在使用者自己的计算机内，也可以运行在另一个可信任第三方的计算机内。如果运行在自己的计算机内，一般来说，安全地产生密钥是没有问题的，除非该计算机已经被恶意的攻击者控制了。但是并非所有情况下都希望公开密钥和私人密钥对在自己的计算机内产生，这有两个可考虑的因素。

- 个人生成的密钥不一定能保证密钥的强度和安全性。每个人对密钥重要性和密钥性质的理解不一样，你很难确保自己生成的密钥是安全的，而对于一个安全性非常重要的组织来说，这是不能允许的。
- 对于一个组织来说，密钥一般来说需要进行集中备份。一个公司或组织希望能够备份每个员工的密钥，它不会希望如果一个员工突然出意外而再也没有办法阅读和处理这个员工负责的资料和数据，它也不会希望它无法对公司的一个员工所作的事情没有任何办

法监控。

基于上述这些原因，很多时候会要求使用一个服务器集中生成密钥，然后再将密钥分发给密钥持有人。那么如何确保服务器的安全以确保安全地生成密钥就显得非常重要。攻击者可以有很多办法攻击这样一个密钥集中生成的系统，如果他取得了服务器的控制权，那么所有的密钥都将是不安全的。攻击者还可以通过 IP 欺骗使密钥使用者相信它产生的密钥是可信的，更有甚者，攻击者还可能是服务器的管理员本身，所有这些，都是密钥生成需要考虑的安全问题。

对于对称加密算法，存在同样的问题，比如通常在基于对称加密算法的协议中，会要求仲裁者生成会话密钥，这也跟公开密钥算法存在一样的安全性问题。

2. 生成安全的密钥

密码算法的安全性依赖于密钥的安全性，如果一个通信系统使用的密钥是弱密钥，那么整个密码通信系统都是脆弱的。

密钥的强弱首先跟密钥的长度有关系。一般来说，密钥越长，密钥的安全性就越强。这对于穷举攻击来说是最明显的。比如一个 8 位的密钥，那么穷举攻击的次数就是 $2^8$ (256)，这对于攻击者来说，可能轻而易举就可以计算出来；而一个 32 位的密钥，需要的计算次数可能就是 $2^{32}$，对于攻击者来说，这需要的时间代价就会大得多。但是并非密钥越长越好，因为密钥越长，你所花费的时间代价就越高，密钥长度的选择总是一个折中的结果，既要保证安全性，又要考虑时间代价。相同长度的对称加密算法跟公开密钥算法抗穷举攻击的能力不一样，因为公开密钥算法是基于大数分解的难度的。表 3-1 列出了一些攻击难度相同的对称加密算法和公开密钥算法。

表 **3-1** 对于穷举攻击难度相同的对称密钥长度和公开密钥长度

| 对称密钥长度 | 公开密钥长度 |
|---|---|
| 56 位 | 384 位 |
| 64 位 | 512 位 |
| 80 位 | 768 位 |
| 112 位 | 1 792 位 |
| 128 位 | 2 304 位 |

对于相同长度的密钥，也存在强密钥和弱密钥的区别。通常用户偏向于选择弱密钥，弱密钥主要是指那些易于猜测的密钥，比如用户的账号信息、用户生日和用户其他个人信息，还可能包括常用的单词和短语，等等。这种密钥，对于抵抗字典攻击是非常脆弱的。

解决这种弱密钥的一个办法是做一些限制，强迫用户选择一些安全性稍高的密钥。比如要求用户选择的密钥在 8 个字符以上，包含的字符应该有数字、大写字母、小写字母和特殊字符。

另一种做法是随机生成密钥。必须保证随机源具有可靠的随机性，要么从真正的物理随机源生成，要么从安全的伪随机发生器中生成。对于许多加密算法来说，都存在特定的

弱密钥，这些弱密钥相对于其他密钥来说会更加脆弱。通常这些密钥的比率很小，也可以通过对生成的密钥进行检测来发现。比如在 DES 算法的 $2^{56}$ 个密钥中，仅仅存在 16 个弱密钥，比率非常低。对于公开密钥算法来说，随机生成密钥会困难一点，因为公开密钥算法的密钥需要满足一些附加的数学要求，比如要求是素数等。大随机素数的产生也是密码学专门研究的一个领域。

使用基于单向散列函数的通行短语替代单词产生密钥也是一个很好的选择。在这种方法中，可以使用一个长的容易记忆的句子替代单词作为输入，然后使用单向散列函数算法将该句子转换成一个固定长度的密钥，这样如果句子足够长，产生的密钥就可以看作是随机的。这种方法既容易记忆，又保证了密钥的强度。图 3-1 所示是一个使用时间作为随机源生成密钥的框架图。

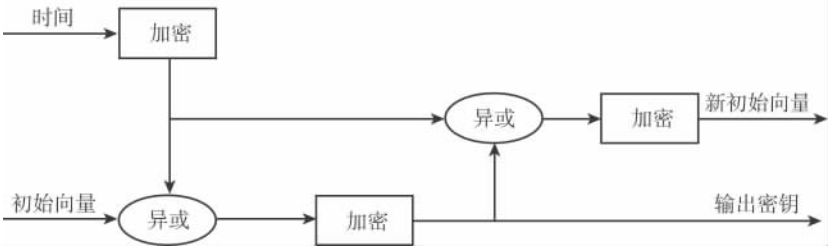


图 3-1　随机密钥生成举例

### 3.1.2　密钥分发

通信双方要进行安全的通信，首先需要协商密钥，尤其对于对称加密算法，双方必须先取得同样的加密密钥（会话密钥），无论这个密钥怎么产生，都需要进行密钥信息的交换。对于对称加密算法来说，这存在很多问题。最安全的手段当然是物理手段，你可以将密钥复制到软盘上，然后坐车送给你要通信的接收方。但是这显然很不划算，有时候甚至是很困难的，比如两个人在不同的国家。或者可以通过网络传输，但是因为在此之前你们没有办法建立安全信道，密钥只能通过明文在网络上传输，这就非常容易泄漏，而密钥一旦泄漏了，利用该密钥建立起来的安全通信信道就一点也不安全。当然，有些时候可以将密钥分成多个部分从不同的信道甚至在不同的时间发送，这样可以减少受攻击的危险。

对密钥进行分类是一种折中的方案，在这种方案中，将密钥分为密钥加密密钥和数据加密密钥。密钥加密密钥是用来加密数据密钥的密钥，它使用相对比较少，只在每次开始建立通信信道的时候需要使用。这种密钥一般只能通过手动分发，比如采用智能卡和 USB Key，等等。有了这种密钥加密密钥，就可以在每次通信信道建立的时候用它来加密数据密钥从而建立安全通信信道。

公开密钥算法更好地解决了这个问题，因为公开密钥算法的公开密钥是可以公开的，所以在网络上可以明文传输，从而可以利用公开密钥交换加密的会话密钥。但是公开密钥算法的这种情况是基于假设私人密钥是不需要传输的，很多时候并非如此。私人密钥和公开密钥对可能并不是由用户自己产生，而是由一个服务器集中产生，这时候私人密钥的传输会产生和对称密钥相同的问题，这通常也用手动分发的方法解决。

### 3.1.3 密钥验证

密钥验证包括两方面的内容：一是验证密钥的发送方是可信的；二是验证密钥确实是正确和完整的。

密钥通常由另一方发送过来，但是怎么验证发送密钥的人确实是可信的，怎么肯定密钥确实是由所声称的发送人发送的呢？这一点很重要，如果不能肯定这一点而使用了攻击者发送过来的密钥，那么安全性就无从谈起。可以使用数字签名解决这个问题，即密钥发送人使用自己的私人密钥对发送的密钥进行签名，接收方通过验证签名确定密钥的可信性。

密钥一般来说是通过网络传输的，网络传输不可能保证百分之百的正确，但是密钥必须百分之百的正确，否则根本不可能进行正常的通信。有一些校验办法可以利用，比如可以在发送密钥之前使用密钥加密一个常量，然后将加密常量跟密钥一起传输，接收到密钥后，使用密钥解密该常量，如果跟发送的常量一致，说明密钥是正确的。

更好的办法是利用单向散列函数的性质，先对密钥作单向散列函数运算，并将得到的散列值与密钥一起发送，在接收端，对密钥作同样的散列运算，如果一致，肯定接收到的密钥是正确的。因为单向散列函数对数据的微小变化十分敏感，所以这是比较可靠的。

### 3.1.4 密钥使用

使用密钥总的来说有两种方式，一种是使用软件进行加密，一种是使用硬件进行加密。软件加密是较常使用而不安全的方式。软件加密通常使用了整个系统的资源，包括内存和硬盘，在使用加密程序进行加密的时候，如果系统中有其他恶意程序（比如恶意木马程序）存在，那么可以很容易将你的密钥窃取。即便你安全地完成了加密，也可能由于密钥无意中留在硬盘中而泄漏。总之，使用软件在计算机中进行加密很危险。目前有一种有效的针对 RSA 的时间分析攻击方法，就是利用软件监视在计算机中程序运行的状态来进行攻击的。

利用硬件进行加密是一个更安全的方案，密钥一般保存在硬件中，硬件自身具有计算功能，能够在内部完成数据的加密和解密，所以密钥不用读出到计算机内存中，这减少了受到攻击的风险。这种硬件加密跟计算机协作的流程一般如下。

① 用户将硬件验证口令、数据及要进行操作的指令送入硬件接口。

② 硬件在内部比较口令是否正确，如果正确，则将要接收的数据使用内部的密钥进行指定的计算（一般是加密或者解密）。

③ 硬件将计算结果返回给用户。

注意，在这个流程中，第②步是完全在硬件内部完成的，不利用外部的任何资源，所以不会泄漏出密钥信息。事实上，在这种硬件的设计中，密钥一般是不能从硬件读出的，这确保了密钥的安全性。并且读者应该注意到，使用该硬件还要输入口令，这也增加了安全性，使得即便硬件丢失了，攻击者也不能轻易使用硬件进行冒充，还需要花费一定时间破解口令。

### 3.1.5　密钥存储

除了临时的会话密钥，其他无论是什么类型的密钥总是要以某种方式保存下来，以便以后再次使用。

密钥存储最常用的方式或许是存储在人的大脑中，比如一些通过口令得到密钥的系统，我们一般把口令存储在脑中。当然，有些人对自己的记忆力不是那么确定，他们将密钥记在工作本或便笺上，然而这种方法应该避免，因为这大大增加了密钥泄漏的可能性。

通常密钥会存储在文件中，存入文件中的密钥最好加密保存，比如采用 DES 算法进行加密，这样会一定程度上增加密钥的安全性。许多基于密码的系统就是这样实现的，比如 Microsoft 的 IE，对导入 IE 中的私人密钥就增加了加密存放的功能，PKCS＃12 标准也规定了私人密钥需要加密。

更安全的方式是存储在硬件中或者存储器中。一种是没有计算功能的存储器，密钥存放在里面，可以方便携带和接入计算机，比如现在的 IC 卡或者 USB 接口的 FLASH，同样，存放在里面的密钥最好也是经过加密后存放。这样，只要存储密钥的硬件还在你手中，就不太可能丢失和泄漏。如果存储密钥的硬件丢失了，拿到密钥硬件的攻击者没有你的密钥保护口令，一时也难以利用密钥，这可以使你有一定时间进行处理，比如更换密钥。IC 卡或 FLASH 中的密钥是可以很容易复制出来的，这是一个缺点，增加安全性的办法是使用密钥的时候同时要求提供存储硬件的特征。存储在智能卡这种设备中是更安全的，因为这种设备不需要将密钥读出卡外，即便在使用的时候也不需要，有些产品的密钥就是在内部产生、内部存储及内部使用，不能读出，大大增加了密钥的安全性。

存储密钥的同时要考虑的还有备份密钥的问题。很多情况下需要对密钥进行备份，比如一个机构中的职员使用的密钥，大多数都会要求备份，以便在出现意外的情况下也能够使用或访问该职员所保存的公司资料。备份的一般方式是在组织中集中进行，但是需要建立一个可信和安全的机制，以防止备份密钥的泄漏。比如使用多个密钥对备份的密钥进行加密，而每个密钥掌握在不同的人手中。

### 3.1.6　密钥销毁

一般来说，密钥都有一定的使用寿命，通常也会给密钥设定有效期，这是因为密钥使用的时间越长，其泄漏和受攻击的危险就越大。比如密钥使用时间长了，使用该密钥加密的信息就会比较多，那么就更容易针对该密钥进行唯密文分析。密钥加密密钥有效期一般比较长，比如数字证书里面的密钥，一般会使用一年或多年；而数据加密密钥有效期一般很短，比如只有几分钟，甚至几秒钟。

无论密钥有效期长短，到了有效期之后，我们都要更换新的密钥，同时销毁旧的密钥。销毁旧密钥事实上很重要，这样可以避免攻击者用旧密钥解密以前他截取到的信息。必须尽可能地做到彻底销毁密钥，因为密钥攻击者通常会很有耐心，他们会从内存中、密钥存储过的硬盘空间中、废弃不用的 IC 卡中甚至废纸堆中寻找密钥。

### 3.1.7　公钥管理

相对于需要保密的密钥来说，公钥管理显得轻松一些，因为公钥是可以公开的。事实

上，从哪里获取公钥一般是不重要的，只要保证该公钥是可信任的。获取公钥可能的途径有以下一些。

- 从通信的对方获取。
- 从公开的公开密钥数据库获取。
- 从通信对方的朋友处获取。
- 从自己已有的数据库中获取。

但是除了最后一个途径可信程度比较大之外，其他途径都需要考虑这样一个问题：获取的公钥是否可信，即获取的公钥是不是确实是该公钥宣称所代表的人使用的。一般来说，比较好的解决方法是通过可信任第三方的数字签名来确认这个问题，目前比较系统化和标准的应用方法就是数字证书。

数字证书（又称为电子证书）是由一个可信任的第三方签发的，除了公钥外，一般包括使用人的姓名、地址、邮件和其他私人信息，然后可信任的第三方用自己的私人密钥对所有这些信息签名。这样可以防止攻击者替换公钥的行为，获得证书的用户只要使用可信任第三方的公钥验证可信任第三方对证书的签名即可以确定证书是否可信。

事实上在数字证书这种系统中，所有的信任关系都建立在签发证书的可信任第三方之上，所以这个可信任第三方的建立和安全就显得异常重要。通常签发数字证书的可信任第三方机构称为证书验证中心（CA）。CA 的管理是一个复杂的问题，涉及管理体制和技术上的诸多问题，当然，仅仅是技术还是好办的，至少对于密码技术人员来说不是一个难以克服的问题。

为了建立一个通用的可信任的 CA 系统，通常模拟现实的社会采用逐级验证的方法来实现。也就是说，最高级的 CA（根 CA）验证下一级的 CA，下一级的 CA 再验证其下属的 CA，最末端的 CA 直接验证用户。这样，就构成了一个所谓的证书链。证书链中的根 CA 是自信任的，也就是说，对它的信任应该是无条件的，那么谁能承担这样一种角色就是一个系统值得慎重考虑的问题。

CA 对公钥的管理是集中式的，事实上，还有分布式的方式，并且这种方式在 PGP 中取得了成功。考虑这样一种情况，你是公司的老板，需要招聘一位财务人员，因为财务人员涉及公司的诸多秘密，必须要是可信任的。你没有办法通过中介结构或者招聘来获得对应聘者的这种信任，如果有一个很好的朋友向你推荐这样一位应聘者，你基本上就在一定程度上获得了对他的信任。这就是 PGP 的所谓分布式公开密钥管理，实际上就是在已有的信任关系的基础上不断延伸和加强的一种链式或树状信任关系。对于私人的应用来说，这是不错的一种方式。

## 3.2 分组加密模式

对称加密算法按其加密数据的方式一般来说可以分成两种类型：分组加密和序列加密。分组加密又称为块加密，是将要处理的数据分成固定的长度，然后在这固定长度的数据上使用密码算法进行计算。序列加密模式又称为流加密方式，是对要处理的数据按位（或字节）逐个进行加密处理。

密码算法的加密模式结构通常很简单，由一些基本的加密算法、反馈流程及简单运算

组成。通常密码算法的加密模式应该考虑下列的问题。

- 算法加密模式的安全性。虽然加密模式不会对密码算法本身的安全性产生影响，但是一个差的算法加密模式可能会增加攻击者攻击成功的机会，甚至提供机会。例如，如果相同的明文加密后总是产生相同的密文，就很容易被攻击者利用。
- 算法加密模式的容错性。加密的数据通常要经过网络传输，网络传输中出现差错是很可能的，这时候就需要考虑算法的容错性。有些算法会导致错误扩散，其中一位出错而导致多位甚至多个字节不能正常解密。
- 算法加密模式的效率。算法加密模式的选择，对加密效率会有影响，一般来说，要基本保证加密模式的效率不会比算法本身的效率低过多。
- 算法加密模式的实时性。不同的算法加密模式实时性不同，有的算法可以一边加密一边实时传输和解密，有些则需要等到一个数据块全部接收到了才能进行解密。在不同的应用中，对实时性的要求是不一样的。

常用的分组加密模式有四种，分别是：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）。下面重点介绍这四种加密模式，对其他的加密模式仅作简单的概括。

### 3.2.1　电子密码本模式

电子密码本模式（ECB）是最简单的分组加密模式，也是最能体现“分组”概念的加密模式。它将加密的数据分成若干组，每组的大小跟加密密钥长度相同，然后每组都用相同的密钥进行加密。比如 DES 算法，使用一个 64 位的密钥，如果采用该模式加密，就是将要加密的数据分成每组 64 位的数据，如果最后一组不够 64 位，那么就补齐为 64 位，然后每组数据都采用 DES 算法的 64 位密钥进行加密。图 3-2 是电子密码本模式的流程图，图中 P0、P1 是明文分组，C0、C1 是相应的密文分组。

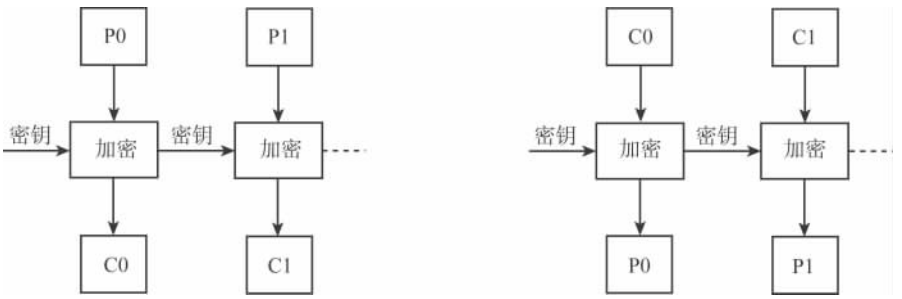


图 3-2　电子密码本模式加密和解密流程

比如对“My name is DragonKing”这句话，就可以 8 个字符（64 位）为一分组，形成图 3-3 这样的分组，最后一个分组不足 8 个字符，可以采用其他数据填充补齐。

| My name | is Drago | nKing |
|---|---|---|

图 3-3　使用 64 位分组电子密码本模式的实例

电子密码本加密模式的每个分组的加密是相互独立的，相互之间没有关系，可以不必按顺序进行，各分组也可以并行进行加密和解密。这在某些应用方式下是非常有用的，比如数据库加密。如果使用电子密码本模式对数据库中的每个记录进行加密，那么数据库的存取就会非常方便，每个记录都可以独立进行加密和解密的存取、添加或者删除等操作，而且可以进行并行的处理以提高速度。

电子密码本模式加密的数据，如果密文数据出错了，解密的时候会影响整个明文分组，可能导致整个明文分组不能正确地解密，但是不会影响其他明文分组。电子密码本模式对密文数据的增减是十分敏感的，如果密文数据中插入或丢失了一位数据，那么随后的整个密文序列都将不能正确地解密，这称为同步错误。所以，使用电子密码本模式的时候，必须确保密文分组的完整性。

采用电子密码本模式加密，相同的明文采用相同的密钥加密总是得到相同的密文，对于加密大量数据的密钥来说，这非常容易受到攻击。大量信息的情况下会有很多数据重复，那么如果攻击者知道了一段明文对应的密文，无论那段密文在哪儿出现，他都能解密该密文。更加严重的是，如果电子密码本模式加密的是一种有固定结构的信息，那么攻击者甚至可以不通过破译密钥就能达到自己的目的。

假设一个公司给员工账号发放工资的信息结构如图 3-4 所示，该信息是由公司通过网络发往银行的，采用了电子密码本模式加密。假设 Jim 是该公司的一个员工，通过大量信息的收集，他完全可以知道自己的姓名和账号对应的帧，并把自己姓名的密文（第 1 个到第 6.5 个分组）存储下来。做了这些准备，Jim 就可以偷梁换柱了。最简单的方法是 Jim 监视公司的网络，特别是在发工资的那些日子，如果发现转账信息，可以偷偷把他上头高级主管的信息的后面 2.5 个分组跟自己信息的后面 2.5 个分组交换，这样他会得到比自己多得多的工资。如果他足够大胆或准备赚一把就逃的话，他可以把所有人的转账信息的前面 6.5 个分组都换成自己的信息分组，这样他就可以有机会到银行取走所有人的工资然后逃走。

| 信息结构 | 姓名 | | | | 账号 | | | 金额 | |
|---|---|---|---|---|---|---|---|---|---|
| 分组号码 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

图 3-4　电子密码本模式加密的信息帧结构

公司可以通过经常更换加密密钥增加这种攻击难度，但是不能根本解决问题。还有一种办法就是利用单向散列函数的性质增加 MAC 功能。

综上所述，电子密码本模式有以下特点。

- 每次加密的数据长度固定。
- 各个分组相互独立，可以并行加密和解密。
- 相同的明文使用相同的密钥总是产生相同的密文。
- 一个位的错误只对所在的明文块产生影响，但是增加或删除一个位会导致其后整个密文序列没有办法正确解密。

### 3.2.2　加密分组链接模式

加密分组链接模式（CBC）可以解决电子密码本模式容易受到分组替换攻击的问题。

加密分组链接模式首先也是将明文分成固定长度的分组，然后将前面一个加密分组输出的密文与下一个要加密的明文分组进行异或操作计算，将计算结果再用密钥进行加密得到密文。第一明文分组加密的时候，因为前面没有加密的密文，所以需要一个初始化向量（IV）。跟电子密码本模式不一样，通过链接关系，使得密文跟明文不再是一一对应的关系，破解起来更困难，而且克服了只要简单调换密文分组就可能达到目的的攻击。加密分组链接模式的流程如图 3-5 所示，其中 P0、P1 是明文分组，C0、C1 是密文分组，IV 是初始向量。

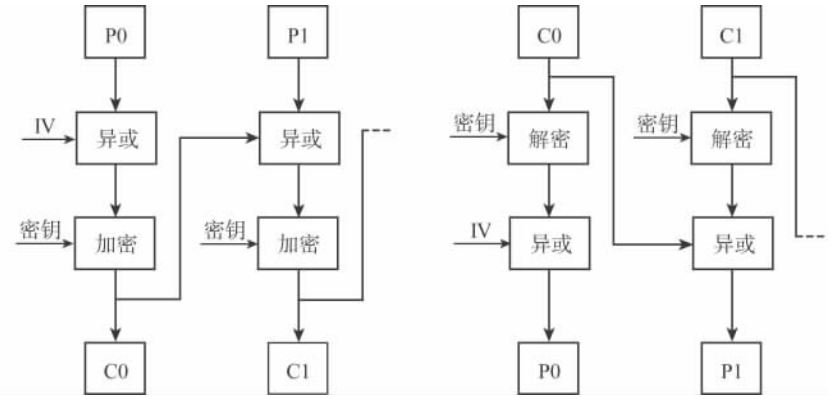


图 3-5　加密分组链接模式加密和解密流程

加密分组链接模式的过程用数学公式表示如下：

$$C_i = E_K(P_i \oplus C_{i-1})$$
$$P_i = D_K(C_i) \oplus C_{i-1}$$

加密分组链接模式之所以能够使得对相同明文采用相同的密钥加密能够得到不同的密文，是因为该模式在进行解密之前，将明文跟一段长度相同的数据进行了异或操作。对于一般的分组来说，与明文异或操作的数据来自前一个密文输出，但是对于第一个明文分组来说，它的前面并没有密文输出可以利用，所以必须给定一段跟分组等长度的数据，称为初始向量（IV）。初始向量一般随机产生，也可以由时间标记产生，解密使用的初始向量必须跟加密的时候使用的初始向量一致，否则不可能得到正确的解密结果。使用不同的初始向量，相同的明文使用相同的密钥会产生完全不同的密文，这使得攻击者对密文的分析更加困难；而使用相同的初始化向量，相同的明文使用相同的密钥还是会产生相同的密文，这是我们不愿意看到的结果，所以一般应该避免重复利用初始向量。

初始向量在加密分组链接模式中没有任何保密价值，因为除第一个分组外，加密分组链接中其他分组用来跟明文进行异或的数据都是前面输出的密文分组，而这个密文分组显然是可以在网络上简单获取的，所以，即便你对初始向量进行保密，也仅仅对第一个分组有一点点意义，而对其他大部分分组来说没有任何意义。

加密分组链接模式的缺点之一是会造成错误扩散。对于明文错误来说，性质并不是很严重，因为经过加密和解密来说恢复的明文还只是发送错误的地方有错误，不会扩散到别

的地方。而且，明文发生错误的可能性较小。

密文发生错误的可能性更大，因为密文要经过有许多不可确定因素的网络的传播。在加密分组链接模式中，如果密文有一位发生了错误，那么其后的一个分组中相应的位会发生错误，该分组的明文不能恢复，但是错误分组的第二个分组之后的分组不会受这个错误的影响，这称为加密分组链接模式的自恢复功能。

跟电子密码本模式一样，加密分组链接模式也不能恢复同步错误。也就是说，如果密文中增加或删除了一位，那么该位以后的所有密文的解密都将难以得到正确的明文结果。所以使用加密分组链接和电子密码本模式都需要确保分组链接的完整性。

加密分组链接模式虽然对抵抗分组重复攻击和分组替换攻击有效，但是很容易受到干扰性的攻击。比如攻击者可以在密文之后附加一些信息，而接收者没有办法发现这是附加的信息，虽然可能是乱码，但有时候可能导致其他重大的安全问题。攻击者还可以利用加密分组链接错误扩散的特点进行破坏性的攻击，比如通过改变一个分组的一位从而控制下面一个分组相应位的变化，甚至通过删除或增加一位密文数据从而使得明文无法恢复，等等。此外，如果使用相同的初始化变量和加密密钥加密的信息数据量太大，一样会给攻击者提供大量的信息，这也是需要注意的。

综上所述，加密分组链接具有以下的一些特点。

- 每次加密的数据长度固定。
- 当相同的明文使用相同的密钥和初始向量的时候 CBC 模式总是产生相同的密文。
- 链接操作使得密文分组要依赖当前和以前处理过的明文分组，密文分组顺序不能进行重新排列，也不能进行并行操作。
- 可以使用不同的初始化向量来避免相同的明文产生相同的密文，能一定程度上抵抗字典攻击等密文分析。
- 一位发生错误后，会对当前及后一个分组的明文产生错误。增加或删除一个位会导致其后整个密文序列没有办法正确解密。
- 不能实时解密，必须等到 8 个字节都接收到之后才能开始解密，否则得不到正确的结果。

### 3.2.3 加密反馈模式

在加密分组链接模式下，必须等整个分组的数据接收完之后才能进行解密，不能实时解密，这在很多网络应用中是不适合的。加密反馈模式（CFB）正是了为了适应这种要求作的改进。

加密反馈模式通过引入移位寄存器来克服加密分组链接模式不能实时解密的困难。图 3-6 所示是带 64 位移位寄存器的加密反馈模式的加密和解密流程图。图中 C2、C3 及 P10 等都是一个字节（8 位）的数据，所以能够实现字符的实时加密和解密，不用再等到 8 个字节都接收到之后再进行解密。图中是在进行第 10 个字节数据的加密和解密过程，在该过程中，先从移位寄存器取 8 个字节的数据（C2 到 C9）用密钥进行加密，然后取加密数据最左边的一个字节跟输入的明文 P10 进行异或操作，得到的值作为输出密文 C10，同时将 C10 送入到移位寄存器中。

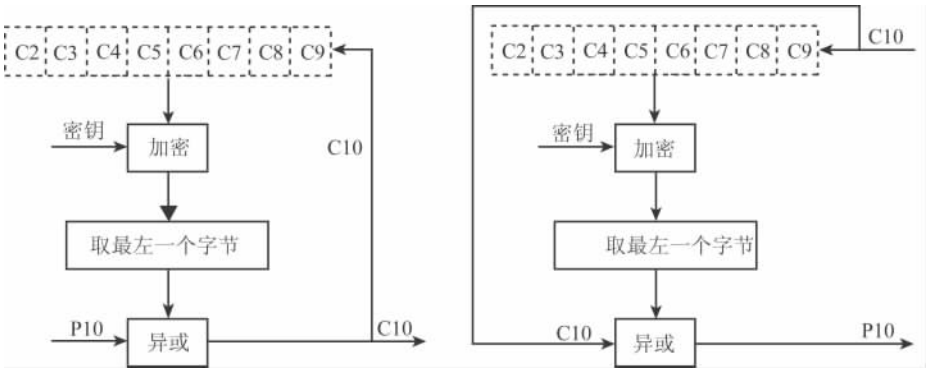


图 3-6　加密反馈模式的加密和解密流程

加密反馈模式用数学公式可以表示为：

$$C_i = P_i \oplus E_K(C_{i-1})$$

$$P_i = C_i \oplus E_K(C_{i-1})$$

跟加密分组链接模式一样，加密反馈模式的密文也跟以前所有的明文有关系，具有相关联的关系。在刚开始的时候，移位寄存器也要使用初始向量填充。初始向量同样没有必要保密，其原因跟加密分组链接模式是一样的。对不同的消息，初始向量应更换，以保证在使用相同的密钥加密信息的时候不会产生重复的密文，比如可以使用不断增大的索引号码作为初始向量。

加密反馈模式同样存在错误扩散的问题。明文错误不会有太大的影响，如果明文其中一位出错了，解密后它也只会影响明文错误的那一位。

但是密文错误影响就会严重得多。在如图 3-6 所示的加密反馈模式中，如果密文中有一位发生了错误，那么当前字节的密文及随后 8 个字节的密文的解密都会受到影响，可能会导致无法正确解密。因为在这 9 个字节的解密计算都使用到了该错误的密文。但是在这 9 个字节之后，密文就能自动恢复到正常解密的状态。

对于同步错误，加密反馈模式同样具有自恢复的功能，如果密文数据中增加或删除一位，在这一位进入到寄存器的时候，会影响其所在的 8 个字节，然后移出寄存器，密文就恢复了正确解密。

基于加密反馈模式的错误扩散性能，攻击者一样可以通过更改密文某些位使得密文解密成另外一些消息。

综上所述，加密反馈模式具有以下特点。

- 每次加密的数据不超过 64 位。
- 当使用相同的密钥和初始向量的时候，相同明文使用 CFB 模式加密输出相同的密文。
- 链接操作的方法使得密文数据依赖当前和以前所有的数据，所以数据都应该按顺序组织在一起，不能进行并行计算操作。
- 可以使用不同的初始变量使相同的明文产生不同的密文，防止字典攻击等密文分析行为。
- CFB 模式的强度依赖于密钥的长度，强度最大的情况是每次加密的数据长度和密

钥长度相同的情况。

- 当每次加密的数据长度的取值比较小的时候，相同的明文一般需要更多的循环来完成加密，这可能会导致过大的开销。
- 每次加密数据的位数应该为 8 的整数倍。
- 一旦某位数据出错，会影响到目前和其后一些字节的加密数据的正确解密，但是对同步错误具有自恢复功能。
- 数据可以实时传输，每接收到一位都可以随即进行解密。

### 3.2.4 输出反馈模式

输出反馈模式（OFB）跟加密反馈模式相似，只是输入寄存器的数据不太一样。这种方式输入寄存器的数据从加密算法输出的分组的前 8 位数据取得（对于加密数据长度为 8 位的输出反馈模式）。图 3-7 所示是输出分组长度为 64 位，每次加密输出为 8 位的输出反馈模式流程图。图中，S2、S3、P10 和 C10 等都是一个字节。图中是在进行第 10 个字节数据的加密和解密过程，在该过程中，先从移位寄存器取 8 个字节的数据（S2 到 S9）用密钥进行加密，然后取加密数据最左边的一个字节 S10 跟输入的明文 P10 进行异或操作，得到的值作为输出密文 C10，同时将 S10 送入到移位寄存器中。

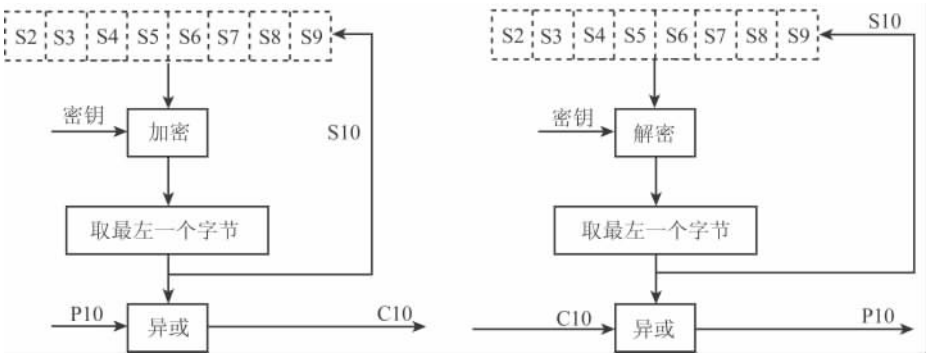


图 3-7 输出反馈模式的加密和解密流程

输出反馈模式用数学公式表示如下：

$$C_i = P_i \oplus S_i，\text{其中 } S_i = E_K(S_{i-1})$$

$$P_i = C_i \oplus S_i，\text{其中 } S_i = E_K(S_{i-1})$$

这里 $S_i$ 是一个内部状态，跟密文和明文都是不相关的。因为输出反馈模式使用的反馈变量是内部变量，所以有时候也就叫内部反馈模式。

输出反馈模式也需要使用初始向量对寄存器进行初始化，初始化向量也不需要保密。

输出反馈模式不存在错误扩散的问题，如果密文的一位发生了错误，那么只会影响其对应明文的一个位，而不会影响别的位。对于同步错误，输出反馈模式没有自恢复功能，所以一旦增加或删除了密文的一位，该位后面的所有密文都将不能正确解密。

输出反馈模式的密文与前后数据的相关性小，所以安全性相对于加密分组链接模式和加密反馈模式比较弱。输出反馈模式使用密文序列跟明文异或产生明文，而密文序列由密钥本身产生，这肯定具有周期性，在使用相同密钥加密的数据到达一定数量时，会产生重复，这就会大大降低算法的安全性。

综上所述，输出反馈模式的特点如下。

- 每次加密的数据不超过 64 位。
- 当使用相同的密钥和初始向量的时候，相同明文使用 OFB 模式加密输出相同的密文。要注意的是，在 OFB 模式下相同的密钥和初始向量产生相同的密钥流，所以，为了安全原因，一个特定的初始向量对一个给定的密钥应该只使用一次。
- 因为没有使用链接操作，数据相关性小，所以使得 OFB 模式更容易受到攻击。
- 可以使用不同的初始变量产生不同的密钥流，从而使得相同的明文使用相同密钥产生不同的密文。
- 当每次加密的数据长度取值比较小的时候，相同的明文一般需要更多的循环来完成加密，这可能会导致过大的开销。
- 每次加密数据的位数应该为 8 的整数倍。
- OFB 模式不会进行错误传播，某位密文发生错误，只会影响该位对应的明文，而不会影响别的位。
- OFB 模式不是自同步的，如果加密和解密两个操作失去同步，那么系统需要重新初始化。
- 每次重新同步的时候，应该使用不同的初始向量。这样可以避免产生相同的密文，从而避免“已知明文”攻击。

### 3.2.5　三重分组加密模式

许多分组加密算法，仅仅使用前面介绍的一些分组加密模式可能安全性不够，这是由于分组长度的限制导致的，比如 DES 算法，密钥长度一般来说就是 64 位，容易受到穷举攻击。为了解决这个问题，提出了一些新的算法，但是，也有基于现有的算法的新分组加密模式。三重分组加密模式就是在这种情况下提出的，OpenSSL 常用的有三重电子密码本模式（3ECB）和三重加密分组链接模式（3CBC）。

三重分组加密模式使用了不止一个密钥，对明文分组进行了基于基本分组加密模式的加密、解密和加密操作。目前来说，有两种应用方式：一种是使用两个密钥，即第一个密钥和第三个密钥相同；还有一种就是使用三个不同的密钥。图 3-8 所示是使用两个密钥的三重分组加密模式的加密和解密流程图；图 3-9 所示是使用了三个不同密钥的三重分组加密模式的加密和解密流程图。

使用两个密钥的三重分组加密模式的过程用数学公式表述如下：

$$C = E_{K1}(D_{K2}(E_{K1}(P)))$$
$$P = D_{K1}(E_{K2}(D_{K1}(C)))$$

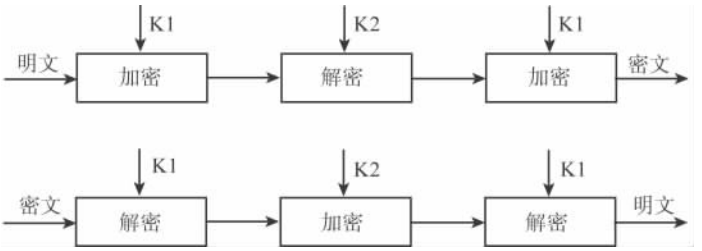


图 3-8　使用两个密钥的三重分组加密模式

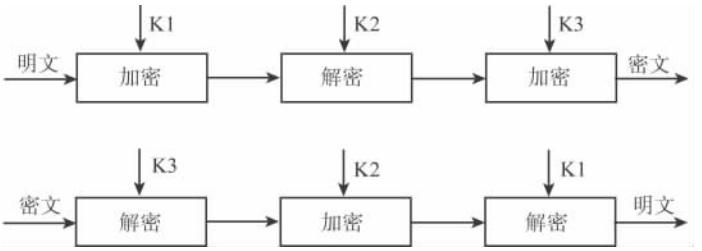


图 3-9　使用三个密钥的三重分组加密模式

使用三个密钥的三重分组加密模式的过程用数学公式表述如下：

$$C=E_{K3}(D_{K2}(E_{K1}(P)))$$
$$P=D_{K1}(E_{K2}(D_{K3}(C)))$$

三重加密分组模式具有如下特点。

- 加密过程为使用 K1 加密，然后使用 K2 解密，最后使用 K3 加密完成加密过程。
- 加密过程使用了电子密码本模式、加密分组链接模式等其他基本分组加密模式，但是密钥增加到了 168 位，虽然目前理论上可以使得有效密钥长度降到 112 位，相对于 56 位的密钥来说，安全性还是得到了很大的提高。
- 如果前两个密钥是相同的，那么就相当于使用一个密钥加密了一次。
- 如果第一个和第三个密钥是相同的，那么密钥长度事实上是 112 位，有些攻击方法可以使得有效密钥的长度降为比 56 位稍多的长度，但需要很大的内存空间。
- 如果三个密钥都是相同的，那么就跟普通的分组加密模式效果相同了。
- 三重分组加密模式的基本特点跟其使用的基本分组加密模式特点基本一致。

### 3.2.6　其他分组加密模式

#### 1. 计数器模式

在输出反馈模式中，移位寄存器的输入是从加密输出分组中最左端的值取出的。计数器模式跟这种模式不同的是移位寄存器的输出是一个计数器，每一个分组完成加密后，计数器要增加某一个常数。计数器模式跟输出反馈模式一样没有错误扩散的特点，但是也同样没有同步错误恢复功能。

#### 2. 分组链接模式

分组链接模式将分组密码算法的输入与前面所有密文分组的异或值相异或。分组链接

模式也需要一个初始向量，具有密文错误扩散的性质。分组链接模式使用数学公式表示如下：

$$C_i = E_K(P_i \oplus F_i);$$
$$F_{i+1} = F_i \oplus C_i$$
$$P_i = F_i \oplus D_K(C_i);$$
$$F_{i+1} = F_i \oplus C_i$$

3. 其他模式

还有很多其他的分组加密模式，比如扩散密码分组链接模式、带校验和的密码分组链接，等等，其基本原理跟前面介绍的分组加密模式相似，感兴趣的读者可以查看相关的文献以获得更详细的内容。

### 3.2.7　数据填充方法

电子密码本模式和加密分组链接模式的分组算法都要求加密输入的分组是固定长度的，但是大多数输入明文可能都不是分组长度的整数倍，也就是说，最后一个分组一般来说是不足一个分组长度的。为了使分组加密算法能够正常工作，通常使用填充技术对最后一个分组进行填充以使该分组正好为一个分组的固定长度。

进行数据填充有很多方法，最简单的就是使用规则的数据进行填充，比如使用全 0 或者全 1。当然，你也可以使用其他自己喜欢的任何模式。这里介绍一种在 OpenSSL 中使用的填充方法。

这种填充的基本思想是使用该分组不够的字节的数作为值填充各个要填充的字节，这样，在解密恢复的时候，检查最后一个字节的值就知道要删除的填充字节的数目。但是为了使这种方法能正确工作，即便最后一个分组是完整的，也要增加一个分组作为填充。例如分组长度位是 8 字节（64 位），最后一个分组长度为 5 个字节，那么需要填充的字节是 3 个字节，那么就使用数值 3 填充最后 3 个字节，如图 3-10（a）所示。图 3-10（b）则是分组长度也是 8 字节（64 位），但是最后一个分组长度是完整的填充示意图，填充的字节长度就是 8 个字节，数值为 8。

上述方法有一个缺点，就是经过填充后，密文长度跟明文长度不一样，这在很多应用中是非常不方便的。为此提出了一种称为密文挪用的分组填充办法，基本思想是从前一个输出的密文分组中截取出一段跟要填充的数据长度相同的数据 $C'$ 填充最后一个不完整的分组 $P_n$，然后该分组加密后作为倒数第二个分组 $C_{n-1}$ 传输，而 $P_{n-1}$ 加密出来的密文分组截取后剩余的一段数据（长度跟明文最后一个不完整的分组 $P_n$ 长度相同）作为最后一个不完整的密文分组 $C_n$ 传输。解密的时候，先解密 $C_{n-1}$ 得到不完整明文分组 $P_n$ 和加密出来的密文分组的后一段 $C'$，然后将 $C_n$ 和 $C'$ 再次合成 $P_{n-1}$ 对应的密文，再进行解密得到 $P_{n-1}$。这样，明文长度和密文长度就保持一致。图 3-11 和图 3-12 分别是电子密码本模式和加密分组链接模式使用密文挪用方法填充最后一个分组的示意图。

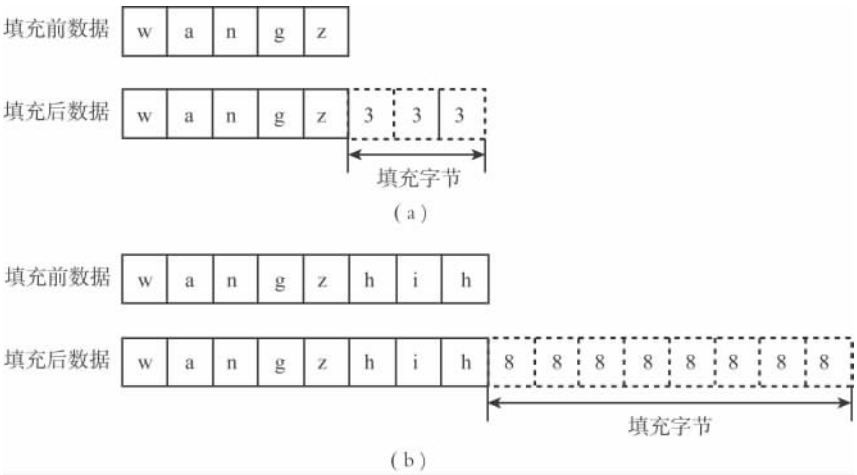


图 3-10　分组加密算法数据填充示意图

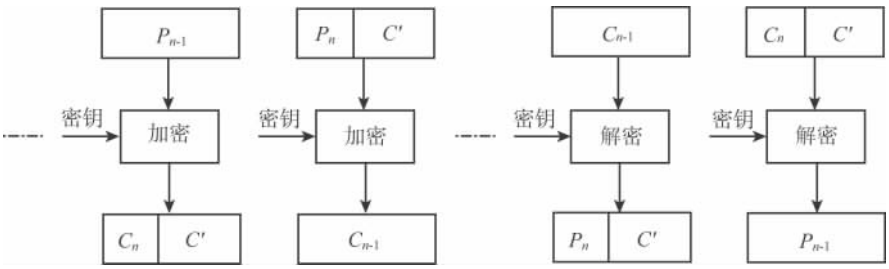


图 3-11　使用密文挪用填充方法的电子密码本模式

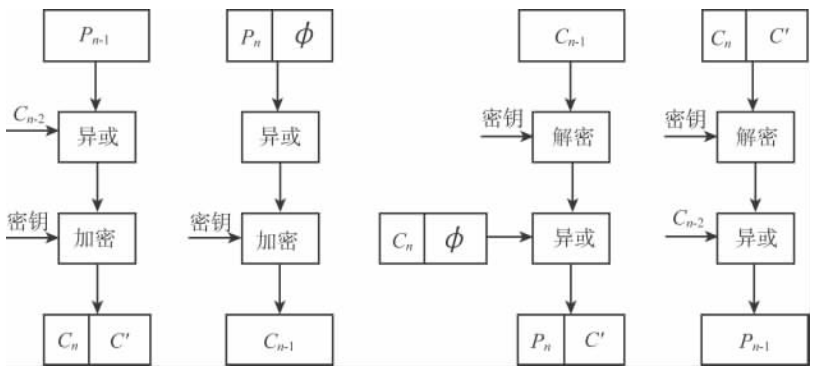


图 3-12　使用密文挪用填充方法的加密分组链接模式

## 3.3　序列加密模式

分组加密模式每次对一组数据进行加密处理，而序列加密模式则不同，它每次对一位明文进行简单运算（一般是异或运算）从而得到密文。一个序列加密模式结构一般由密钥序列发生器、运算单元、明文输入和密文输出组成。图 3-13 是序列加密模式的加密和解密流程图。假设密钥序列发生器的输出位序列为 $K_1$，$K_2$，$K_3$……，明文输入为 $P_1$，

$P_2$，$P_3$……，而密文输出用 $C_1$，$C_2$，$C_3$……表示，则序列加密模式可以用数学公式表示如下：

$$C_i = P_i \oplus K_i$$
$$P_i = C_i \oplus K_i$$

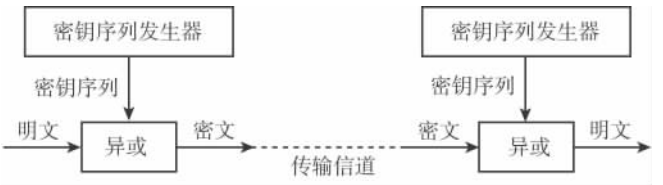


图 3-13　通用序列加密模式模型

序列加密模式的安全性完全依赖密钥序列的安全性，也就是说，依赖密钥序列发生器的安全性。假如密钥序列输出总是为 0，那么相当于没有加密，该加密系统就没有任何存在的意义，如果密钥序列是完全随机不重复的，那么该加密系统的安全性相当于一次一乱密码本方式，几乎是不可破译的。当然，实际上很难构造出输出完全随机的密钥序列发生器。

密钥序列发生器的输出虽然应该看起来是随机的，但必须是确定的，也就是说应该是可以控制的，否则如果加密跟解密的输出密钥序列不一样，就不可能进行正确的数据解密。

密钥序列发生器的输出不能每次都相同，否则就非常容易破密，没有任何安全性。在一个密钥序列发生器每次输出都是一样的情况下，攻击者如果有机会获得一份明文跟密文对应的数据，他就可以恢复出密钥，从而可以用该密钥来读取任何使用该序列加密算法加密的数据，不仅仅是以后的数据，也可以包括以前有办法截取到的数据。所以，为了使密钥序列发生器每次输出不一样，我们需要设置其状态，使得其开始输出的状态都不同，这个初始化的向量就是密钥。这样就增加了攻击者攻击的难度，如果他有办法获得一个密钥，只要更换密钥，他就只好获得新的密钥，但是很可能也是徒劳。

一个密钥序列发生器理论上可以抽象为三部分：内部状态、状态转移函数及输出函数，如图 3-14 所示。内部状态描述了密钥序列发生器当前的内部状态，对于一个同样结构

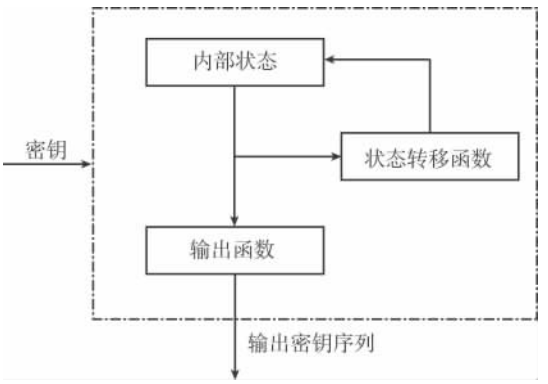


图 3-14　密钥序列发生器的内部结构示意图

的密钥序列发生器，相同的内部状态应该输出相同的密钥序列，否则就不可能进行正确的加密和解密。状态转移函数描述了从当前状态转移到下一个状态的规则，从而产生一个新的状态。输出函数对内部状态进行计算等处理，从而输出密钥序列。在实际应用中，这些组件的划分有时候并非那么明确。对于密钥序列发生器来说，密钥可以看作是对其内部状态进行初始化或重置的向量。

### 3.3.1 自同步序列加密模式

自同步序列加密模式又称为密文自动密钥，它的主要思想是密钥序列跟以前一定数量的密文位相关，即密钥序列是以前一定数量密文位的函数。图 3-15 是自同步序列加密模式的加密和解密框架图。

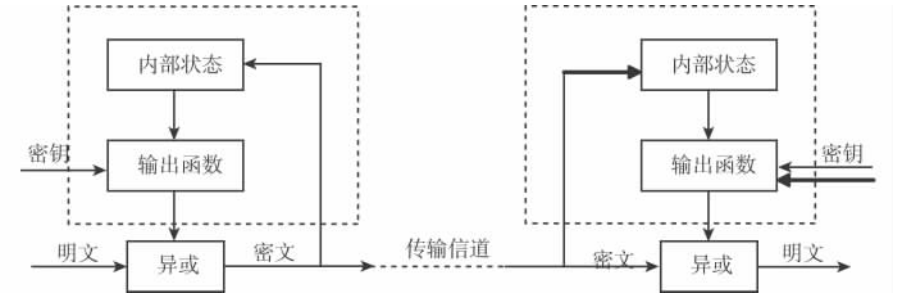


图 3-15 自同步序列加密模式加密和解密流程

在图 3-15 所示的自同步序列加密模式中，内部状态就是由以前密文的 $n$ 位组成，而内部状态中的 $n$ 位密文作为输出函数的输入，所以，输出的密钥序列的复杂性和安全性完全依赖于输出函数。输出函数通常可以有各种各样的形式，比如可以使用分组加密算法或单向散列函数算法为主体构成。因为在自同步序列加密模式中，解密端的内部状态在从开始到收到 $n$ 位密文之后，自动跟密文序列进行同步的解密，所以称为自同步序列加密模式。

在自同步序列加密模式中，不需要进行人工的同步，一般来说，可以在开始的 $n$ 位（假设内部状态是 $n$ 位密文的函数）之前，可以随便传送一些信息，这些解密一般是不正确的，在收到 $n$ 位密文之后，就开始实现同步，能够进行信息的正确解密了。这种能够自动进行解密密钥同步的功能在某些应用中是很有用的。

自同步序列加密模式事实上跟分组加密模式中的加密反馈模式很相似，同样存在错误扩散问题。也就是说，如果有一位密文发生了错误，那么随后的 $n$ 位密文的解密都不能正确进行。

在密钥没有更换的情况下，自同步序列加密模式很容易受到回放攻击，攻击者可以将记录下的密文重新发送给接收者，因为解密器能够自动进行同步，所以接收端会重复接收到正确的解密信息。这在一些电子交易应用中可能给攻击者带来巨大的利益。

综上所述，自同步序列加密模式的特点如下。

- 内部状态是前面 $n$ 位固定长度密文的函数。
- 算法的安全性依赖于输出函数的复杂性和安全性。

- 会产生密文错误扩散，密文中一位发生错误，会导致其后 $n$ 位密文不能正确解密。
- 解密的时候具有密钥序列和密文自动同步的功能，具有同步错误自恢复功能。
- 明文长度和密文长度相同。
- 容易受到回放攻击。

### 3.3.2　同步序列加密模式

同步序列加密模式又称为密钥自动密钥加密，它的特点是密钥序列的产生跟处理的消息是不相关，相互独立的。图 3-16 是同步序列加密模式的加密和解密流程。

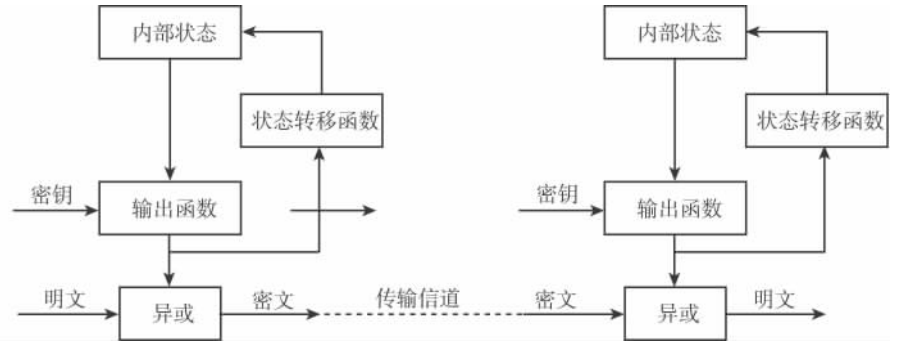


图 3-16　同步序列加密模式的加密和解密流程

在加密端，密钥序列发生器在使用密钥进行初始化后开始产生密钥序列，在解密端，密钥序列发生器使用同样的密钥进行状态初始化从而产生同样的密钥序列，这样就能够开始进行正确的加密和解密。一般来说，加密端和解密端的同步都是通过相同的密钥进行状态初始化设置实现的。由于在加密和解密端必须产生相同的密钥，所以事实上同步序列加密模式的密钥发生器产生的密钥必须是确定的，必须保证这些输出密钥序列在相当长的一个范围内不能重复，否则安全性将得不到保证。当然，最好是不重复，但这基本上是不可能做到的。

如果在解密过程中同步状态被破坏了，比如增加或删除了一个密文位或者密钥序列发生器跳过了一个周期，那么必须重新进行同步，否则数据不可能进行正确的解密。所以同步序列加密模式对同步错误是没有办法恢复的。

同步序列加密模式不会产生密文错误扩散，如果一个密文位发生了错误，它只会影响该位的正确解密，而不会对其他任何密文位的解密产生影响。

因为同步序列加密模式不会对密钥和密文进行自动同步，所以能够一定程度上防止直接针对密文的一些攻击。比如在密文中插入或删除一位很容易被发现，这时候因为失去同步，解密的可能都是乱码。在自同步序列加密模式下采用的简单回放攻击，对同步序列加密模式也是基本无效的。

综上所述，同步序列加密模式的特点如下。

- 密钥序列的产生跟明文和密文消息都无关。
- 不会产生密文错误扩散，但是没有同步错误自动恢复功能。
- 密文和明文长度相同。

## 3.4 加密模式选择

前面介绍了两大类加密算法模式：分组加密模式和序列加密模式。它们之间最大的区别在于分组加密模式每次对一组数据进行加密运算处理，而序列加密模式则逐位对数据进行加密运算处理。事实上，在实际设计应用的算法中，并没有那么严格的区别，它们有可能是相互结合的。比如输出反馈分组加密模式，就可以看作分组加密算法和序列加密模式的结合。事实上，该序列加密模式的输出函数由分组加密算法和一个位选择器组成。表 3-2 列出了分组加密模式和序列加密模式的一些性能上的对比。

分组加密模式和序列加密模式各有优势。分组加密模式便于软件实现，可以每次对一组数据进行处理，避免耗时特别大的位操作，也更能阐释计算机目前常用的分组概念。序列加密模式更适于硬件操作，因为硬件很适合逐位进行计算操作，这对算法的硬件实现是非常有利的。

表 3-2 分组加密模式和序列加密模式性能对比

| | 分组加密模式 | 序列加密模式 |
|---|---|---|
| 安全性能 | 密文之间相关性大，抵抗密文分析能力强，安全性主要依赖加密算法本身 | 密文之间相关性小，安全性主要依赖于密钥序列发生器 |
| 容错性能 | 除了输出反馈模式外，都会产生密文错误扩散；加密分组反馈模式具有同步错误恢复功能 | 自同步序列加密模式会产生密文错误扩散，但具有同步错误恢复功能；同步序列加密模式不会产生错误扩散，但没有同步错误恢复能力 |
| 算法效率 | 明文长度跟密文长度不一定相同，适于软件实现 | 明文长度跟密文长度相同，适用于硬件实现 |
| 实时性能 | 电子密码本模式和加密分组链接模式不能实时解密 | 可以实时解密 |

分组加密模式是目前比较常用的加密模式，各种分组加密模式在性能上各有不同。如果仅仅考虑算法实现的复杂性和数据处理速度，那么选用电子密码本模式是比较好的选择。但是，电子密码本模式也是最容易受到分组重复及密文分析等攻击的分组加密算法，一般不建议使用该加密模式。

加密分组链接模式是适用于大多数文件加密的模式，它的安全性能相对不错，虽然会产生密文错误扩散并且不能进行同步错误恢复，但如果是在本地存储而不通过网络传输，同步错误发生的概率是很小的，所以加密分组链接模式是加密存储文件的首选加密模式。

需要实时解密的系统，比如需要对单个字符进行处理的系统可以选择加密反馈模式，但是该模式会产生密文错误扩散，所以在对错误扩散敏感的系统中是不能使用的。

对容易发生密文错误的系统，可以选择输出反馈模式，因为该模式不会产生密文错误扩散，所以可以降低错误率。但是该模式安全性比较差。表 3-3 是各种分组加密算法性能的比较。

表 3-3　分组加密算法性能比较

| | 电子密码本 | 加密分组链接 | 加密反馈 | 输出反馈 |
|---|---|---|---|---|
| 安全性能 | 相同明文产生相同密文，分组可被替换 | 相同明文可以产生不同密文，分组之间具有相关性，替换困难 | 相同明文可以产生不同密文，分组之间具有相关性，替换困难 | 相同明文可以产生不同密文，分组之间具有相关性，替换困难 |
| 容错性能 | 产生错误扩散，同步错误不能恢复 | 产生错误扩散，同步错误不能恢复 | 产生错误扩散，同步错误可以恢复 | 不产生错误扩散，同步错误不能恢复 |
| 算法效率 | 不会降低分组加密算法速度，可并行计算，密文长度大于明文长度 | 不会降低分组加密算法速度，不可并行计算，密文长度大于明文长度 | 不会降低分组加密算法速度，不可并行计算，密文长度等于明文长度 | 不会降低分组加密算法速度，不可并行计算，密文长度等于明文长度 |
| 实时性能 | 不能实时解密 | 不能实时解密 | 能实时解密 | 能实时解密 |

## 3.5　加密算法应用

加密算法的应用是非常广泛的。首先就是对在通信网络中传输的数据进行加密，防止第三方窃听甚至进行数据篡改。根据对通信网络中传输的数据的保密性要求的不同，加密算法具体应用模式也不尽相同。还有一种常见的加密算法的应用就是对存储数据进行加密，比如对文件或者数据库进行加密，这样对存储的数据进行保护，防止没有授权的非法数据访问和破坏。

此外，在使用加密算法的时候，我们可能还经常面临一些选择，比如使用公开密钥算法还是对称密钥算法，使用硬件加密还是软件加密等问题，本节也将对这些问题作一些简要的介绍。

### 3.5.1　传输数据加密

根据加密设备在网络模型中放置位置的不同，我们可以将传输数据的加密分为物理信道加密、链路加密、终端加密、会话加密及应用加密。

#### 1. 物理信道加密

物理信道加密方式的加密设备或程序放置在链路层和物理层之间，也就是说，所有经过物理设备的数据都将被加密保护，无论是数据还是上层的协议信息。这种数据保护方式非常有利于使用硬件加密设备，因为一般的物理接口都是工业标准化的。但是，因为该方式对所有数据包括路由信息都进行了加密，所以该方式只适合使用在专有线路或者同类型线路上，任何中间的路由在处理信息之前都必须先进行加密。

就安全性来说，这种数据保护方式是非常有效的，因为它对物理信道中传输的所有信息都进行了加密，不但隐藏了数据，而且将所有可能的通信信息都保护起来，攻击者不能得到通信源和目的地址，也不能得到通信的时长和数据量。因为物理信道总是在传输一系列的数据位，没有终止的时候。

但是，物理信道加密的方式也存在问题。因为它是对所有数据进行加密，所以网络节点上的路由器都需要对所有数据进行解密和加密，工作量非常大。此外，因为是基于物理链接的加密，所以必须保证网络上所有的节点都是可信和安全的，如果网络上有任何一个节点有安全问题，那么就会泄漏大量信息。但是，对于一个大的网络来说，通常很难做到

保证所有节点都是安全和可信的。

2. 链路加密

链路加密方式的加密设备放置在链路层和网络层（IP）之间，它将通过 IP 传输的所有数据（包括原始的 IP 头部）都进行加密保护，然后添加新的 IP 头发往设定的目标地址。基于 IPSec 隧道方式的 VPN 技术就是其实现的例子。跟物理信道加密方式不一样，该方式运行于链路层协议之上，并将所有基于 IP 传输的数据发往特定的目标地址，这构成了一个逻辑意义上的私有网络，将公共网络的数据和这个逻辑私有网络的数据分离开来，不能互通，这就是所谓的虚拟专用网技术。

相对于物理信道加密方式，由于使用链路加密方式的数据一般在公共网络中进行传输，所以给攻击者透露了更多的通信信息，比如 IP 网关目的地址和源地址，通信的数据量等。但是因为通过两个网关的数据可能是许多实际终端数据无序的综合，所以攻击者很难获取有用的信息，从而起到了很好的通信量保密作用。同时，链路加密方式也需要考虑密钥管理等问题。

基于链路加密方式的最热门的应用就是 VPN 技术，该技术通过在公共网络（如 Internet）中构建一个虚拟的专用网络，从而为具有很多分支机构的组织和公司提供了廉价的专用网络替代解决方案。

3. 终端加密

终端加密的加密设备放置在网络层（IP）和传输层（TCP、UDP 等传输协议）之间，它对传输层的数据和协议进行了加密保护，然后发往指定的目标终端。目标终端接收到该信息之后，同样需要对 IP 层传送上来的数据进行解密，然后再交给传输层相应的协议模块进行处理。跟链路加密方式不一样，终端加密方式对 IP 首部不进行保护，所以适合于端对端的通信，能够保护两个终端之间所有通过 IP 进行传输的数据。IPSec 的传输方式就是终端加密方式实现的例子之一。

终端加密的方式只对两个终端之间传输的数据进行加密，很容易受到基于信息量分析的安全攻击。此外，同样需要考虑密钥分发等密钥管理问题。

4. 会话加密

会话加密的加密设备放置在传输层和应用协议（如 HTTP，FTP 等）之间，它对上层特定应用协议的数据进行加密保护，然后再交给传输层处理。SSL（Secure Socket Layer）协议就是会话加密方式实现的著名例子。事实上，会话加密存在很多灵活的形式，比如可以只对基于 TCP 之上的数据进行加密保护，也可以只对基于 UDP 上的数据进行保护，甚至可以只对基于 TCP 之上的 HTTP 协议进行单独的保护等。虽然理论上可以实现对所有基于特定传输协议（如 TCP）之上的应用协议进行统一的保护，但是目前来说，更多的是针对特定应用协议的保护，如针对 HTTP 协议的保护。

会话加密通常是对两个终端之间的一个会话链接进行加密保护，所以能够给攻击者透露很多通信量方面的信息，如通信的数据量、通话时长，等等，容易受到通信量分析攻击。此外，密钥管理同样是需要考虑的问题。

会话加密的优点是灵活性好，可移植性好，不需要更改系统底层的协议。设计、开发及投入应用的周期相对比较短，并且能够适应特别行业客户的需要。比如有的用户可能只

对 HTTP 协议的数据保密感兴趣，那么它就完全没有必要采用 VPN 技术，因为这种技术可能还会妨碍他对其他协议的使用。

5. 应用加密

应用加密的加密设备或程序是放置在应用网络协议之上的，它对真正的信息数据进行加密保护，而对应用协议信息不作任何保护，是一种最上层的数据加密保护方式。PGP (Pretty Good Privacy) 程序是该加密模式实现的例子之一，该程序对要通过电子邮件发送的信息进行加密保护，然后再使用电子邮件的相应协议发送出去。

应用加密方式具有非常大的灵活性，如果愿意，可以在任何应用程序之中嵌入应用加密程序，从而实现对数据的保护。同样，如果需要通过网络进行传输，应用加密模式也面临密钥管理的一系列问题。其通信量的泄漏跟会话加密方式基本相同。

### 3.5.2　存储数据加密

加密算法还经常被用于加密存储的数据，比如对文件、数据库或驱动器进行加密。存储数据的加密跟通信数据的加密有很多不同点，比如密文的保存时间、密文的数据量及密钥的保存时间，等等。归纳起来，存储数据的加密通常有以下特点。

- 密文和密钥存储的时间长，需要安全可靠的方式保管密钥。
- 因为针对存储设备，要求快速进行数据的加密和解密，甚至需要采用硬件加密设备或者特殊算法。
- 对于数据库这样的加密应用，因为数据库每个字段的长度可能比加密算法的数据块长度小，所以会导致密文长度比明文长度大，这样会导致需要更多的存储空间。
- 密钥管理更加复杂，因为不同的人可能需要存取不同的文件或者同一个数据库的不同部分。因为加密的数据量太大，不能使用一个单独的密钥加密所有数据，否则密钥很容易受到唯密文攻击。
- 安全管理非常重要。你必须保证你加密的数据相对应的明文不会在别的地方被攻击者取得，否则很容易受到已知明文攻击。还必须保证在阅读解密后的内容之后彻底将解密的内容销毁，否则也容易受到攻击。

单个文件的数据加密非常容易解决，进行统一的加密就可以了。如果文件过大，为了安全性，可以使用一个长密钥的不同部分对文件的不同部分进行加密；也可以使用多个子密钥对文件的不同部分进行加密，然后使用一个主密钥加密这些子密钥，这样可以降低字典攻击成功的危险。

对于数据库这样的存储方式，虽然对整个数据库进行统一的加密将是简单的，但是会带来一系列问题。首先是对于大型数据库，如果只需要读取其中一条记录的一个字段的数据，却需要花大量的时间对整个数据库进行解密，这是很不可行的。其次，对于数据库字段的不同部分，可能有不同的权限控制，这样使用相同的密钥显然是实现不了的。解决的办法是对不同的字段使用不同的密钥进行加密，但是这样很容易受到分组重放攻击，此外，还需要设计产生不同密钥的方法。

经常会涉及对驱动器加密的问题。针对驱动器的加密具有数据量大的特点，所以一般使用主密钥加密数据加密密钥的密钥管理方式。事实上，可以有两种不同的驱动器数据加密方式：驱动器级和文件级。文件级加密方式是针对每个文件分别进行加密，一般每个文

件都使用不同的加密密钥，所有这些加密密钥都使用一个主密钥进行保护。这样在使用某个文件时，需要对这个文件进行解密、使用，然后再加密。驱动器级的加密方式对整个逻辑驱动器进行加密，但是涉及驱动器安装、文件扇区管理及磁盘数据随机存取等问题，所以实现起来比较复杂。这种情况下，启动驱动器前会要求用户输入一个口令，通过这个口令产生一个主密钥，然后用这个主密钥解密真正的解密密钥。

### 3.5.3 公开密钥算法和对称密钥算法

公开密钥算法自产生以来，围绕其与对称密钥算法的优缺点讨论就一直存在。事实上，这两种算法解决问题的重点是不一样的。对称加密算法主要被设计用于数据的加密，而公开密钥算法主要用于解决密钥分发和管理的问题。

就加密数据的安全性而言，选用适当的算法和密钥长度，公开密钥算法和对称加密算法都能达到相同的效果。但是由于公开密钥加密算法加密速度慢，所以一般不使用公开密钥算法进行大量数据的加密，而是使用它作为密钥管理和分发的协议。一个成功的现代密码学系统，应该是同时使用公开密钥算法和对称密钥算法的混合密码系统，从而能够最大程度地发挥这两种不同类型加密算法的优点。

### 3.5.4 硬件加密和软件加密

#### 1. 硬件加密

军事部门等保密性要求高的部门一般都使用硬件加密。因为硬件加密相比于软件加密具有很多优点。

- 速度优势。加密算法的运行很多不是普通计算机所能完成的，所以在计算机上运行效率会非常低，而使用专门针对某种特定算法设计的硬件却能大大加快计算速度。
- 安全性。对于软件加密算法或者程序，可以通过程序跟踪攻击更改算法而却能不让任何人发现。但是硬件加密设备进行了严密的封装，包括防篡改、防电子辐射等设备，使得硬件的安全性更有保障。
- 容易使用。硬件加密设备一般是做成了标准的硬件接口，不需要什么密码知识就能简单地将设备连接到计算机或其他设备中进行使用。而如果使用软件，则需要多得多的知识。

虽然硬件加密有这么多优点，但是这些优点是针对用户来说的。对于开发者来说，开发一个硬件加密设备所需要的成本可能会比开发一个软件加密程序高得多，需要的时间也长得多。这也是国内目前硬件加密设备比较少的重要原因。

#### 2. 软件加密

软件可以实现任何加密算法，其缺点是速度慢、开销大和容易受到篡改；优点是灵活性好、可移植性强及容易升级。软件加密程序很容易得到，所以个人用户可以轻松地使用软件加密保护自己的文件和数据。加密软件开发的周期一般来说也比较短，不需要开发公司投入大量的前期资金。这就是软件加密在国内流行的重要原因。

## 3.6　本 章 小 结

本章首先介绍了密钥管理技术，包括密钥的生成、分发、验证、使用、存储及销毁，并对公钥的管理作了特别的介绍。使得读者对密钥在实际使用中碰到的问题和解决办法具备了初步的认识。

接着本章介绍了分组加密算法的几种基本加密模式，包括电子密码本模式、加密分组链接模式、加密反馈模式和输出反馈模式，并对此衍生出的一些加密模式如三重分组加密模式也作了简要的介绍。此外，还介绍了分组加密模式经常会涉及的数据填充的几种基本方法。

本章对序列加密算法的两种主要序列加密模式进行了详细的介绍，包括自同步序列加密模式和同步序列加密模式。并对这两种模式的优缺点进行了比较。

最后，本章对使用加密算法时需要考虑的一些问题作了简单的阐述，比如加密算法应用的不同方式和目的、加密模式的选择和对比等。

通过阅读本章，读者应该对加密算法在实际实现中可能需要考虑的问题和解决办法有一个基本的认识。

# 第 4 章

# OpenSSL 概述

## 4.1 OpenSSL 背景

本节将对 OpenSSL 的背景作一个简要的介绍，并介绍一些其他类似的软件开发包，从而让读者在对比中得出 OpenSSL 的特点。

### 4.1.1 OpenSSL 简介

前面我们介绍过众多的密码算法、公钥基础设施标准及 SSL 协议，或许这些有趣的功能会让你产生实现所有这些算法和标准的想法。果真如此，在对你表示敬佩的同时，还是忍不住提醒你：这是一个令人望而生畏的过程。这个工作不再是简单的读懂几本密码学专著和协议文档那么简单，而是要理解所有这些算法、标准和协议文档的每一个细节，并用你可能很熟悉的 C 语言字符一个一个去实现这些定义和过程。我们不知道你将需要多少时间来完成这项有趣而可怕的工作，但肯定不是一年两年的问题。我们相信，上面的话肯定不会让你绝望，事实上，自从你拿到本书开始，就不会对这样的话绝望，因为至少你知道有 OpenSSL 这个工具可以使你逃避这个可怕的工作。

我们首先应该感谢 Eric A. Young 和 Tim J. Hudson，他们自 1995 年就开始编写后来具有巨大影响的 OpenSSL 软件包，更令我们高兴的是，这是一个没有太多限制的开放源代码的软件包，这使得我们可以利用这个软件包做很多事情。Eric A. Young 和 Tim J. Hudson 是加拿大人，后来由于写 OpenSSL 功成名就之后就到大公司里赚大钱去了。1998 年，OpenSSL 项目组接管了 OpenSSL 的开发工作，并推出了 OpenSSL 的 0.9.1 版。到目前为止，OpenSSL 的算法已经非常完善，对 SSL2.0，SSL3.0 及 TLS1.0 都支持。OpenSSL 目前最新的版本是 0.9.7b 版。

OpenSSL 采用 C 语言作为开发语言，这使得 OpenSSL 具有优秀的跨平台性能，这对于广大技术人员来说是一件非常美妙的事情，可以在不同的平台使用同样熟悉的东西。OpenSSL 支持 Linux，Windows，BSD，Mac，VMS 等平台，这使得 OpenSSL 具有广泛的适用性。不过，对于目前新成长起来的 C++程序员，可能对于 C 语言的代码不是很习惯，但习惯 C 语言总比使用 C++重新写一个跟 OpenSSL 相同功能的软件包轻松不少。

### 4.1.2 其他密码算法开发包

事实上，任何一个普通的密码学应用开发人员可能都不需要亲自写烦琐难懂的密码算

法和实现那些标准制定者们制定出来的可怕标准，因为这个计算机世界里除了 OpenSSL，还有很多可选择的密码算法开发包。这些开发包有些是诸如 Microsoft 这种公司提供的，有些则是由像 Eirc A. Young 这样的无私贡献者提供的。这些开发包的功能和优缺点不尽相同，下面我们对几个常见的开发包作简单介绍。

我们首先应该想到的就是著名的 Microsoft 提供的密码算法库 CryptoAPI。CryptoAPI 当前最新的版本是 CryptAPI2.0，包含了各种密码算法、密钥管理及证书管理功能，功能相当完善和强大。事实上，Windows 平台基于密码学的安全结构基本上是以 CryptoAPI 为基础的，如支持 SSL，TLS 及 PCT 协议的 SChannel 就是基于 CryptoAPI 之上的，而 IE 也是与 CryptoAPI 间接相关。Windows 的 CryptoAPI 事实上是定义了一个 Windows 平台通用的接口标准，它使用密码支持组件（CSP）提供底层真正的加密操作。CSP 机制使得 CryptoAPI 能够非常灵活，可以方便地使用第三方提供的加密库或硬件设备。事实上，为了跟在 Microsoft 的后面混口饭吃，很多智能卡（Smart Card）厂商都提供了 CSP 接口。CryptoAPI 虽然有这么多的方便之处，但 Microsoft 的产品缺点却是人所共知的。首先 CryptoAPI 是基于 Windows 平台的，不像 OpenSSL 那样有良好的跨平台性能，如果你想在 Linux 下进行密码学应用的开发，这段话你算是白看了；其次 CryptoAPI 不提供源代码，对于你来说，CryptoAPI 是一个黑箱子，根本没有办法知道它里面执行了什么动作，对于一个充满好奇的程序员来说，实在是一件很郁闷的事情，而对于密码实现技术的研究人员来说，CrytoAPI 的意义也极其有限；最后，CrytoAPI 是美国公司的产品，对于像中国这样的国家来说，密码出口产品的算法受到很大限制，对于许多领域的安全来说，这是一个不能接受的事实。

David 的力作 Crypto++也是一个不错的开放源代码的密码算法包，它使用 C++写成，对于 C++程序员来说，这是一个好消息，甚至可能马上兴高采烈地丢掉本书。但是不要高兴得太早，因为 Crypto++仅仅提供了密码算法的实现，而对密钥管理封装和证书管理等相关功能并没有提供很好的支持，如果你需要这些功能，建议你最好还是将本书后面的章节看下去。

如果你是 Java 程序员，还可以选择 JSSE 的密码算法开发包，这是一个功能和 OpenSSL 几乎相当的开发包，区别的最大不同之处在于它是用 Java 写成的。关于安全产品使用 Java 与 C/C++开发哪一个比较好的问题一直是一个争论的焦点，事实上作为一个技术人员，你可能最后根本不会受这些争论的影响，而是取决于自己对这两种语言的偏爱。

## 4.2　OpenSSL 结构

本书作者第一次从 OpenSSL 的官方网站下载了 OpenSSL 开发包并解压到硬盘上的时候，面对 OpenSSL 目录里面的众多目录和文件，感觉是满头雾水，不知道该如何下手。相信你或多或少有这种不愉快的感觉，本节将对 OpenSSL 的这些目录结构作一个大概的介绍，这对于刚开始了解 OpenSSL 是必需的。

### 4.2.1 OpenSSL 总体结构

OpenSSL 整个软件包大概可以分成三个主要的功能部分：密码算法库、SSL 协议库及应用程序。OpenSSL 的目录结构自然也是围绕这三个功能部分进行规划的。

首先，我们注意到 OpenSSL 的根目录下有不少文件，这些文件包含 OpenSSL 各个平台下编译安装的说明文档、编译安装的配置文件及 OpenSSL 本身版本变化的一些说明文档。诸如 INSTALL. * 这样名称的文件，都是安装编译说明文件，后缀名是平台的名称。比如 INSTALL. W32，就是 Windows 平台的 OpenSSL 安装编译说明文件。只有 Linux 的安装编译说明文件是不带后缀的，就是 INSTALL，由此可见 OpenSSL 肯定是出身 Linux 家族了。表 4-1 列出了系统平台和对应安装文件的关系。其他一些文件的作用根据其文件名就可以知道个大概，如果你刚刚接触 OpenSSL，这些文件还是值得一读的。

表 **4-1** **OpenSSL** 安装说明文件名和相应的平台

| 文件名 | 系统平台简单描述 |
|---|---|
| INSTALL | Linux 等 UNIX 平台 |
| INSTALL. W32 | Windows 平台，包括 Windows 98，Windows 2000，Windows NT 和 Windows XP 等 |
| INSTALL. WCE | WinCE 平台 |
| INSTALL. MacOS | 苹果计算机的操作平台 MacOS |
| INSTALL. OS2 | OS/2 操作平台 |
| INSTALL. VMS | VMS 平台，一种在 Windows 系统下的虚拟操作系统 |
| INSTALL. DJGPP | DJGPP 平台，一种在 Windows 系统下的虚拟操作系统 |

接下来该逐一看看那些目录了，这是 OpenSSL 真正的精华所在。并非所有这些子目录都是很重要的，我们在后面对那些重要的子目录将作一个详细的介绍，这里先对一些不是很重要的目录作一个简述。MacOS，MS，OS/2 及 VMS 这几个目录，包含了在不同的平台编译时的环境变量配置文件，在安装编译完成之后，这几个目录就没有作用了。bugs，certs，perl，shlib，times，tools 及 utils 目录都是一些辅助的目录，里面包含的文件对于我们使用 OpenSSL 进行工作并没有很多的帮助，所以可以不作深究。当然，这些目录中的文件在编译的时候起的作用可能是不可或缺的，但是这并非我们关注的焦点。

Crypto 目录是 OpenSSL 所有密码算法和一些 PKI 相关标准源码存放的目录，也是 OpenSSL 最重要的一个目录。SSL 目录是 SSL 协议各个版本的实现源码存放的目录。Doc 目录是 OpenSSL 使用的说明文档存放的目录，这个目录对于 OpenSSL 使用者来说具有“芝麻开门”的作用。Apps 目录存放了 OpenSSL 所有应用程序的源代码文件，也是研究 OpenSSL API 的很好的例子。Demos 目录就是一些乐意奉献的人写的 OpenSSL 应用的例子，在你开始使用 OpenSSL 进行工作之前，可以看看这个目录，或许会有所帮助。Include 目录是使用 OpenSSL 的库进行编程的时候可能需要使用到的一些头文件。Test 目录是 OpenSSL 一些自身功能测试源程序所在的地方。表 4-2 是一些重要目录的名称和其对应的功能描述。

表 4-2　OpenSSL 部分目录的功能说明

| 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准如 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 |

如果你在 Windows 平台下将 OpenSSL 编译成功后，还会增加三个新的目录：Inc32，Out32dll，Tmp32dll。Inc32 目录和 Include 目录相似，存放的是 Windows 平台下使用 OpenSSL 进行编程需要包含的头文件。Out32dll 则存放了 OpenSSL 编译成功后的可执行应用程序、链接库 LIB 文件和动态 DLL 文件。Tmp32dll 则是在编译过程中存放 OBJ 等临时文件的目录。

### 4.2.2　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 下编程后的库文件名为 libeay32.lib，在 Linux 下编译后生成的库文件名为 libcrypto.a。Crypto 目录下包含了众多的子目录，这些子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。

为了让读者对这些目录有一个总体的了解，表 4-3 列出了 Crypto 目录主要的子目录的功能和简单说明。

表 4-3　Crypto 子目录列表

| 目录名称 | 目录类型 | 内容或功能描述 |
| --- | --- | --- |
| Aes | 对称加密算法 | 美国新的对称加密算法标准 AES 算法源码 |
| Bf | 对称加密算法 | Blowfish 对称加密算法源码 |
| Cast | 对称加密算法 | CAST 对称加密算法源码 |
| Des | 对称加密算法 | 包括了 DES 和 3DES 对称加密算法源码 |
| Idea | 对称加密算法 | IDEA 对称加密算法源码 |
| Rc2 | 对称加密算法 | RC2 对称加密算法源码 |

续表

| 目录名称 | 目录类型 | 内容或功能描述 |
| --- | --- | --- |
| Rc4 | 对称加密算法 | RC4 对称加密算法源码 |
| Rc5 | 对称加密算法 | RC5 对称加密算法源码 |
| Dh | 非对称加密算法 | DH 非对称密钥交换算法源码 |
| Dsa | 非对称加密算法 | DSA 非对称加密算法源码，用于数字签名 |
| Ec | 非对称加密算法 | EC 椭圆曲线算法源码 |
| Rsa | 非对称加密算法 | 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 | 其他标准支持 | 支持 Kerberos 协议的一些接口函数和结构定义 |
| Hmac | 其他标准支持 | HMAC 标准的支持结构和函数源码 |
| Lhash | 其他标准支持 | 动态 HASH 表结构和函数源码 |
| 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 自定义的加载动态链接库的管理函数接口。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 |

续表

| 目录名称 | 目录类型 | 内容或功能描述 |
| --- | --- | --- |
| 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 |

通过阅读表 4-3，对 OpenSSL 密码算法库提供的功能也就有了一个大概的了解。事实上，在 Crypto 根目录下还有一些文件，也是密码算法库的组成部分之一，但是由于不是很主要的部分，这里就不再一一介绍。对 OpenSSL 提供的密码算法大概总结一下：对称加密算法 8 种，非对称加密算法 4 种，信息摘要算法 5 种。

### 4.2.3　OpenSSL 文档目录

OpenSSL 的文档目录 Doc 对于刚刚接触 OpenSSL 的人来说应该是一个最值得留恋的地方，从这里开始漫长的 OpenSSL 之旅是一个很好的选择。OpenSSL 提供的文档并不全面，甚至可以说是非常有限，而且不能及时跟着版本更新。但是对于初学者来说，这些文档能解决不少问题，事实上这也不能责怪 OpenSSL 的开发者，因为对于一个酷爱编写代码的优秀技术人员来说，写文档实在是一件枯燥无比的工作。当然，这些文档都是使用英文撰写的，如果你在读本书，你可能就是因为不太喜欢英文。

OpenSSL 的文档使用 Perl 文档格式保存，为“. pod”文件，对于一些用户，比如 Windows 平台的用户，阅读这样的文档可能存在一些麻烦，因为直接用写字板或者其他阅读器打开这些文档会使格式显得凌乱。可以使用 Perl 工具 pod2text 或者 pod2html 指令将文档转换成 txt 文本格式或者 html 格式以方便阅读。

OpenSSL 的文档主要分为三部分：应用程序说明文档、密码算法库 API 文档及 SSL 协议库 API 文档，分别对应 Doc 根目录下的三个子目录 Apps，Crypto 和 SSL。应用程序说明文档目录（Apps）包含了大部分 OpenSSL 应用程序的使用和参数说明，并有部分例子。密码算法库 API 文档（Crypto）则包含了部分 OpenSSL 密码算法库的 API 的使用说明，可惜不是很全面。SSL 协议库 API 文档包含了 OpenSSL 实现的 SSL 协议和 TLS 协议的大部分 API 使用说明，该部分由于变动比较缓慢，所以文档相对全面一些。此外，目前版本的 OpenSSL 文档目录下还有一个 Howto 子目录，现在只有一个文件，内容是关于证书的一些问题，希望后续的版本能够丰富这个 Howto 目录。OpenSSL 文档目录的根目录下还有两个综述性质的文档 ssleay. txt 和 openssl. txt，也是值得一看的。

## 4.3 OpenSSL 功能

作为一个基于密码学的安全开发包，OpenSSL 提供的功能相当强大和全面，囊括了主要的密码算法、常用的密钥和证书封装管理功能及 SSL 协议，并提供了丰富的应用程序供测试或其他目的使用。

### 4.3.1 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，其他算法使用的则是 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的 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，使得各种对称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，大大提高了代码的可重用性能。

### 4.3.2 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 算法一般用于密钥交换。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。

### 4.3.3 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，提高了代码的重用性。当然，跟对称加密算法和非对称加密算法不一样，信息摘要算法是不可逆的，不需要一个解密的逆函数。

### 4.3.4 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。

### 4.3.5　SSL 和 TLS 协议

虽然已经有众多的软件实现了 SSL 的功能，但是 OpenSSL 里面实现的 SSL 协议能够让我们对 SSL 协议有一个更加清楚的认识，因为至少存在两点：一是 OpenSSL 实现的 SSL 协议是开放源代码的，我们可以研究 SSL 协议实现的每一个细节；二是 OpenSSL 实现的 SSL 协议是纯粹的 SSL 协议，没有跟其他协议（如 HTTP）协议结合在一起。由于 SSL 协议现在经常跟 HTTP 协议在一起应用形成 HTTPS 协议，所以很多人误认为 SSL 协议就是为了保护 Web 安全性的，这实在是一个很大的误解！

OpenSSL 实现了 SSL 协议的 SSLv2 和 SSLv3，支持了其中绝大部分的算法协议。OpenSSL 也实现了 TLSv1.0，TLS 是 SSLv3 的标准化版，虽然区别不大，但毕竟有很多细节不尽相同。OpenSSL 除了提供使用 SSL 协议和 TLS 协议的 API 接口函数之外，还提供了两个不错的应用程序 S_Client 和 S_Server。S_Client 用来模拟 SSL 客户端，可以用来测试 SSL 服务器，比如 IIS 和带 mod_ssl 的 Apache 等；而 S_Server 模拟了一个 SSL 服务器，可以用来测试 SSL 客户端，比如 IE 和 Netscape 等。事实上，由于是开放源代码的，S_Client 和 S_Server 程序的源代码还是很好的 OpenSSL 的 SSL 接口 API 使用例子。

### 4.3.6　应用程序

OpenSSL 的应用程序已经成为了 OpenSSL 一个重要的组成部分，其重要性恐怕是 OpenSSL 的开发者开始没有想到的。现在 OpenSSL 的应用中，很多都是基于 OpenSSL 的应用程序而不是其 API 的，如 OpenCA，就是完全使用 OpenSSL 的应用程序实现的。OpenSSL 的应用程序是基于 OpenSSL 的密码算法库和 SSL 协议库写成的，所以也是一些非常好的 OpenSSL 的 API 使用范例，读懂所有这些范例，你对 OpenSSL 的 API 使用了解就比较全面了，当然，这也是一项锻炼你的意志力的工作。

OpenSSL 的应用程序提供了相对全面的功能，在相当多的人看来，OpenSSL 已经为自己做好了一切，不需要再做更多的开发工作了，所以，他们也把这些应用程序称为 OpenSSL 的指令。OpenSSL 的应用程序主要包括密钥生成、证书管理、格式转换、数据加密和签名、SSL 测试，以及其他辅助配置功能。表 4-4 是 OpenSSL-0.9.7 版本的指令列表。在表 4-4 中，根据指令的性质，对指令进行了归类，这些归类不一定科学，但是对

于本书的描述来说，会有帮助。这些类型包括：对称密钥指令、非对称密钥指令、信息摘要和签名指令、证书签发和管理指令、标准转换指令、SSL测试指令及其他指令。

表 4-4 OpenSSL-0.9.7 指令列表

| 指　令 | 类　型 | 功能说明 |
|---|---|---|
| asn1parse | 其他 | 对 ASN.1 编码的文件或字符串进行解析，比如对证书文件，可以使用该指令对其进行解释，它会将其中每一个数据对象打印处理 |
| ca | 证书签发和管理 | 该指令是一个功能强大的指令，模拟了一个小型 CA 的功能，并跟 OpenSSL 提供的文本数据库联系起来作为证书数据库。该指令具有证书签发、验证、吊销等功能 |
| ciphers | 其他 | 该指令可以列出不同的协议支持的算法体系 |
| crl | 证书签发和管理 | 该指令可以对吊销证书列表（CRL）文件进行文本解析和验证 |
| crl2pkcs7 | 格式转换 | 该指令可以将 CRL 和多个证书封装成一个 PKCS#7 格式的证书文件 |
| dgst | 信息摘要和签名 | 该指令可以使用不同的信息摘要算法对输入的信息进行信息摘要操作，并且可以对摘要的信息进行签名或者验证 |
| dh | 非对称密钥 | 生成和处理 DH 密钥参数文件，dhparam 已经集成了该指令的功能，一般不使用该指令了 |
| dhparam | 非对称密钥 | 用于生成 DH 密钥参数文件、解析 DH 密钥参数文件及格式转换等。在新版本中，这个指令集成了 dh 和 gendh 指令的功能，而 dh 和 gendh 指令在未来的版本中可能会消失或者用作其他用途 |
| dsa | 非对称密钥 | 该指令用于对 DSA 密钥的格式转换及信息输出处理，并且可以对 DSA 密钥进行加密 |
| dsaparam | 非对称密钥 | 该指令用于生成和处理 DSA 参数文件，并且可以用于生成 DSA 密钥 |
| enc | 对称密钥 | 该指令可以使用 OpenSSL 支持的各种对称加密算法对给定的数据或者文件进行加密或者解密 |
| engine | 其他 | 该指令显示 OpenSSL 支持的 Engine 接口列表，并可以测试 OpenSSL 支持的 engine 接口是否有效 |
| errstr | 其他 | 根据给定错误代码显示相应的错误信息 |
| gendh | 非对称密钥 | 生成 DH 密钥，该功能已经集成到 dhparam 指令中 |
| gendsa | 非对称密钥 | 根据 DSA 参数文件生成一个 DSA 私钥，可以采用不同的算法对 DSA 私钥加密保护 |
| genrsa | 非对称密钥 | 生成 RSA 私钥，可以同时对 RSA 私钥进行加密保护 |
| nseq | 格式转换 | 将普通 X.509 证书转换成 Netscape 格式的证书，也可以将 Netscape 证书转换成 X.509 证书 |
| ocsp | 其他 | 是一个实现了在线证书状态协议（OCSP）的指令工具，可以对证书的有效性进行验证等 OCSP 任务操作 |
| passwd | 其他 | 根据给定的口令通过 HASH 算法生成密钥 |
| pkcs12 | 格式转换 | 该指令可以将 X.509 证书和 PEM 编码的私钥封装成 PKCS#12 格式的证书，也可以将 PKCS#12 格式的证书转换成 X.509 证书和私钥 |

续表

| 指　令 | 类　型 | 功能说明 |
|---|---|---|
| pkcs7 | 格式转换 | 该指令将 PKCS#7 格式的文件转换成普通的 X.509 格式的证书或 CRL |
| pkcs8 | 格式转换 | 该指令可以将私钥加密转换成 PKCS#8 格式或者将 PKCS#8 格式私钥转换成普通的 PEM 或 DER 编码私钥 |
| rand | 其他 | 该指令可以产生一系列的伪随机数比特，并保存在文件中 |
| req | 证书签发和管理 | 该指令生成证书标准的请求文件，而且可以生成自签名的根证书 |
| rsa | 非对称密钥 | 该指令对 RSA 密钥进行格式转换和文本解析输出等处理，格式转换的时候可以对密钥进行加密 |
| rsautl | 非对称密钥 | 该指令采用 RSA 算法对输入数据进行签名、验证、加密和解密等操作 |
| s_client | SSL 测试 | 该指令模拟一个 SSL 客户端，可以对支持 SSL 的服务器进行测试和调试操作 |
| s_server | SSL 测试 | 该指令模拟一个 SSL 服务器，可以对支持 SSL 的浏览器（如 IE）进行测试和链接操作 |
| s_time | SSL 测试 | 该指令可以用来测试建立一个 SSL 链接的时间 |
| sess_id | SSL 测试 | 该指令可以处理经编码保存下来的 SSL Session 结构并可以根据选项打印出其中的信息 |
| smime | 其他 | 该指令可以用来对 S/MIME 邮件进行加密、解密、签名和验证等操作 |
| speed | 其他 | 该指令测试算法的速度，如果有硬件加密设备，也可以测试硬件加密设备的速度 |
| spkac | 其他 | 该指令用来处理 Nestcape 的签名公钥和挑战文件（SPKAC），可以验证 SPKAC的签名，打印信息，也可以用来生成 SPKAC 文件 |
| verify | 证书签发和管理 | 该指令用来验证证书或者证书链的合法性 |
| version | 其他 | 该指令用来输出 OpenSSL 的版本信息 |
| x509 | 证书签发和管理 | 该指令用来显示证书内容及签发新的证书 |

在上述列出的指令中，有些指令，如 dgst，事实上包括了很多指令，如 sha，md5，等等，直接使用这些算法名字作为指令也是一样的效果的。如下面两个指令执行结果是一样的。

- 采用 dgst 指令进行信息摘要操作：

```
OpenSSL>dgst -md5 x509.o
MD5(x509.o) = 079a8fe71c51fe606d58d0c8117ac1f6
```

- 采用 md5 指令进行信息摘要操作：

```
OpenSSL>md5 x509.o
MD5(x509.o) = 079a8fe71c51fe606d58d0c8117ac1f6
```

这样的指令还有 Enc，它也是可以采用不同的指令形式执行相同的操作，如下面两个指令的执行结果是相同的。

- 采用 enc 指令执行加密操作：

```
OpenSSL>enc -des -in x509.o -out x5091.txt
enter des-cbc encryption password:
```

```
Verifying- enter des-cbc encryption password:
```

- 直接使用 des 指令执行加密操作：

```
OpenSSL>des -in x509.o -out x5092.txt
enter des -cbc encryption password:
Verifying -enter des -cbc encryption password:
```

从上面的指令运行结果可以看到，它们事实上都是采用 DES-CBC 方式对文件进行加密。

一般来说，执行 OpenSSL 的应用程序是从 OpenSSL 指令开始，该指令事实上提供了一个进入所有其他指令的接口。开始进入该平台，可以在 OpenSSL 执行程序目录下输入：

```
[dragonking@ann425 apps]$./openssl
OpenSSL>
```

在出现的提示符下输入相应指令即可。

如果在 Windows 平台下，你使用“MS Dev Studio workspace for OpenSSL”对 OpenSSL 源码进行编译，你可以得到独立的上述介绍的 OpenSSL 大部分应用程序，所以可以直接使用其名称直接运行，而不用先运行 OpenSSL.exe 指令，这跟在 OpenSSL 指令提供的平台下执行是一样的。事实上，OpenSSL 指令本身就是调用其他指令实现的。如在 Windows 平台上下面指令执行结果是相同的。

- 启动 OpenSSL 指令后再执行指令程序：

```
G:\openssl\openssl-0.9.7\out32dll>openssl
OpenSSL>dgst -md5.rnd
MD5(.rnd) = a53cf1c25f6acde6c55b2eaf4a9e3a9f
```

- 直接执行指令程序：

```
G:\openssl\openssl-0.9.7\out32dll>dgst -md5.rnd
MD5(.rnd) = a53cf1c25f6acde6c55b2eaf4a9e3a9f
```

### 4.3.7 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 机制的目的是为了使 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 已经不仅仅是一个加密库，而是提供了一个通用的加密接口，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协调地工作，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。

目前，OpenSSL 的 0.9.7 版本支持的内嵌第三方加密设备有 8 种，包括：CryptoSwift，nCipher，Atalla，Nuron，UBSEC，Aep，SureWare 及 IBM 4758 CCA 的硬件加密设备。现在还出现了支持 PKCS#11 接口的 Engine 接口，支持微软 CryptoAPI 的接口也有人进行开发。当然，所有上述 Engine 接口支持不一定很全面，比如，可能支持其中一两种公开密钥算法。表 4-5 是 OpenSSL-0.9.7 版本支持的硬件及其对应的简要描述名称，

这个简要描述名称在很多时候是要使用的，如编程或执行指令的时候，该简要名称是大小写敏感的，目前一般都是采用小写字符。

表 4-5　OpenSSL 支持的 Engine 接口

| 简要名称 | Engine 接口描述 |
| --- | --- |
| dynamic | 动态加载 Engine 设备的接口 |
| cswift | CryptoSwift 的硬件加密设备 Engine 支持 |
| chil | nChipher 硬件加密设备 Engine 支持 |
| atalla | Atalla 硬件加密设备 Engine 支持 |
| nuron | Nuron 硬件加密设备 Engine 支持 |
| ubsec | UBSEC 硬件加密设备 Engine 支持 |
| aep | Aep 硬件加密设备 Engine 支持 |
| sureware | SureWare 硬件加密设备 Engine 支持 |
| 4758cca | IBM 4758 CCA 硬件加密设备 Engine 支持 |

### 4.3.8　辅助功能

前面介绍了 OpenSSL 的大部分功能，如果你第一次接触 OpenSSL 而又对密码学技术有所了解，这些功能相信会让你激动不已，如果你愿意，OpenSSL 其实还有一些能让你高兴的地方。

首先最值的一提的就是 OpenSSL 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、文件访问及 Socket 等。这使得代码的重用性大幅度提高，OpenSSL 提供的 API 的复杂性也降低了很多。前面介绍的 EVP 封装也提高了 OpenSSL 代码的可重用性。

从前面应用程序的介绍就可以得知，OpenSSL 对于随机数的生成和管理也提供了一整套的解决方法和 API 支持函数。随机数的好坏是决定一个密钥是否安全的重要前提，OpenSSL 提供了这么一个解决方案，虽然不一定在所有的应用中都能令人满意。

OpenSSL 还提供了其他的一些辅助功能，如从口令生成密钥的 API，证书签发和管理中的配置文件机制，等等。如果你有足够的耐心，将会在深入使用 OpenSSL 的过程中慢慢发现很多这样的小功能，让你不断有新的惊喜。

## 4.4　OpenSSL 应用

OpenSSL 的应用一般可以分为两种不同的方式：基于 OpenSSL 指令的应用和基于 OpenSSL 加密库和协议库的应用。前者更容易一些，而后者需要做的工作更多一些。当然，这些应用不一定是截然分开的，你当然可以两种都用一点，比如使用 SSL 协议的 API，但是证书可以使用 OpenSSL 的指令签发。

### 4.4.1　基于 OpenSSL 指令的应用

基于 OpenSSL 指令的应用很容易，只要安装好了 OpenSSL，就可以开始使用了。最简单的应用就是使用 req，ca 及 x509 指令来签发一个证书了。该证书可以用于各种目的，

比如很多 Apache 服务器就是使用 OpenSSL 的指令生成的证书作为服务器证书。此外，作为测试目的，使用 OpenSSL 指令生成一个证书也是不错的选择。

对 OpenSSL 指令的集大成者应该是 OpenCA 了，它是一个完全基于 OpenSSL 的指令开发的一个证书验证中心（CA）程序，没有使用任何 OpenSSL 的 API。当然，它不仅仅使用了 OpenSSL 的一个软件包，它还使用了诸如 OpenLDAP 等软件包，OpenCA 从一个侧面上证明了 OpenSSL 指令的作用是很全面的。

### 4.4.2 基于 OpenSSL 函数的应用

基于 OpenSSL 函数库的应用相比与基于 OpenSSL 指令的应用开发的工作量会大很多，但这并不意味着其应用会更少。事实上，基于 OpenSSL 的应用大部分是这种方式的，这种方式使得开发者能够根据自己的需求灵活地进行选择，而不必受 OpenSSL 有限的指令的限制。

Mod_ssl 是一个基于 OpenSSL 指令应用的范例，该例中 Apache 和 mod_ssl 结合起来实现 HTTPS 服务，成功地在 Linux 和 Windows 等平台运行。Stunnel 是另外一个成功应用 OpenSSL 函数库的程序，该程序提供一种包含客户端和服务器的安全代理服务器功能，事实上就是在客户端和服务器使用 OpenSSL 建立一条安全的 SSL 通道。

除了这些广为人知的项目外，有很多公司的商业密码产品都是基于 OpenSSL 的函数库开发的，在 OpenSSL 提供了 Engine 机制后，这种势头会更加明显。当然，出于各种目的，相当一部分公司在自己的产品中不会声明这一点。

## 4.5 OpenSSL 授权

OpenSSL 的授权问题很多人并不重视，或者，你也不愿意浪费宝贵的时间在这一节上面。但是，阅读本书的人基本上都是以通过自己的知识在产品中的实现来体现自己的价值的，所以努力维护知识产权，从根本上还是维护了我们自己的利益。

OpenSSL 的授权由两部分组成：OpenSSL 授权和原始的 SSLeay 授权。当然，这两种授权大同小异，基本上是没有限制的，只要在产品中加入或者保留 OpenSSL 和 SSLeay 的版权声明就行。详细的授权内容请参考 OpenSSL 根目录下的 LICENSE。

## 4.6 本 章 小 结

本章对 OpenSSL 作了一个初步的介绍，给读者全面展示了 OpenSSL 的历史、结构、功能和应用。

本章首先介绍了 OpenSSL 的背景，并将 OpenSSL 和其他同类的加密算法库做了比较。

本章还介绍了 OpenSSL 的结构，对 OpenSSL 的目录进行了总体的阐述，让读者对 OpenSSL 的结构有了一个全面的认识。

接着重点介绍了 OpenSSL 的功能，对 OpenSSL 的功能作了一个快速的浏览，展现了 OpenSSL 的魅力所在。

最后，本章还对 OpenSSL 的应用现状和实例做了简单的介绍，并对 OpenSSL 的授权做了强调。

# 第 5 章

# OpenSSL 编译和安装

## 5.1 概　　述

因为是基于 C 语言开发的，所以 OpenSSL 具有优秀的跨平台性能，在许多平台上都可以进行编译和安装。目前来说，OpenSSL-0.9.7 版本官方测试支持的有 UNIX（Linux），DOS（需 djgpp 支持），Windows，OpenVMS，MacOS，OS/2 和 WinCE 等 7 种平台。事实上，还可以将 OpenSSL 成功移植到其他平台，如 VxWorks、Tandem（Compad Non-Stop 核心）、Sony 的 NEWS-OS、Hurd 等。当然，在这些平台的移植需要做的工作可能需要多一点，至少，它们的说明文档不是很全面。

如果你使用的平台是 Linux 平台，那么你将是幸运的，因为 OpenSSL 在 Linux 系统上的编译和安装已经做得非常好，只要按照说明，一般都不会出什么问题。而对于其他平台的用户，比如 Windows 用户，编译和安装 Linux 还是一件比较具有挑战性的工作。当然，并非所有 OpenSSL 用户都要花时间编译和安装 Linux，事实上，对于 Windows 用户来说，完全可以到“中国 OpenSSL 专业论坛”下载已经编译好的 OpenSSL 执行版本使用，这样可以节省很多时间。如果你要对 OpenSSL 的代码进行改写并且要进一步研究 OpenSSL，那么编译和安装的过程将是不可避免的，而且，完成之后你会发现，对 OpenSSL 的了解又深入了一步，至少，它对你来说不再是一个“黑匣子”了。

下载 OpenSSL 源码后，在使用之前要经过两个阶段：编译和安装，这两个阶段是分开的。首先要做的是编译，也就是说，将 OpenSSL 的源代码编译链接成可执行文件、静态链接库和动态链接库；然后进行安装，也就是将 OpenSSL 编译好的执行程序、静态链接库和动态链接库复制到特定的系统目录，在某些系统中，可能还需要设置环境变量等。一般来说，前者所占的工作量比较大，也容易出问题。

OpenSSL 进行编译的时候，首先面临的是选择编译器的问题，在某些系统中，有多种编译器可以选择，比如 Windows 系统，可以选择 Visual C++，也可以选择 Borland C++ Builder，甚至还可以选择在 Mingw32 或者 Cygwin 中的 GNU C 编译器进行编译。这取决于用户的软件配置情况和偏好。理论上，所有符合 ANSI C 标准的编译器都可以用来编译 OpenSSL，但是具体的设置可能不太一样。

OpenSSL 编译好后，一般生成：应用程序（指令）、静态链接库（跟头文件一起使用）和动态链接库。如果你仅仅需要使用 OpenSSL 的指令，那么应用程序部分就能够满足你的要求，也就是编译后名为 openssl 的程序，在 Windows 下名字为 openssl.exe。如

果你需要使用 OpenSSL 的 API 进行编程，那么使用静态链接库和头文件就可以，当然，如果你要使用动态链接库，也是可以的。不过，动态链接库目前对于 OpenSSL 来说还处于测试阶段，所以在默认情况下不是所有系统编译完后都会自动生成动态链接库。目前，在 Windows 平台使用 VC 编译提供了自动生成动态链接库的编译脚本，而使用 Borland C++ Builder 则没有提供生成动态链接库的编译脚本。如果在 Linux 平台想生成动态链接库，也需要增加额外的选项。

无论什么平台，使用 OpenSSL 编译一般需要经过三个步骤：

- 使用 Configure 批处理指令，根据平台环境、选用的编译器及其他参数设置编译的环境变量；
- 通过使用批处理指令来根据上面设置好的环境变量生成编译脚本文件；
- 使用特定的编译指令进行编译链接。

在上述三个步骤中，第一个步骤使用的 Configure 批处理指令是一个复杂的脚本处理文件，我们在下一节将对它作专门的介绍。

## 5.2 Configure 脚本指令

很多人可能已经使用 OpenSSL 有一段时间了，但可能对 OpenSSL 根目录下的 Configure文件所知甚少，其实，这是 OpenSSL 的许多秘密入口之一。

### 5.2.1 Configure 功能概述

OpenSSL 之所以具有优秀的跨平台性能，除了前面提到的采用 C 语言编写的条件之外，还有就是因为 OpenSSL 具有一个强大的具有平台适应性的 Configure 脚本文件。Configure 指令使得 OpenSSL 可以适应多种不同的系统平台和多达几十种不同的编译器。Configure 指令还可以使 OpenSSL 在编译的时候具备组件的选择功能，比如可以选择支持或者不支持某种算法或者协议等，这使得 OpenSSL 具有很大的灵活性，比如在空间有限的嵌入式系统移植中就具备独特的优势。Configure 指令还有其他一些灵活的功能，比如选择 OpenSSL 安装目录、是否支持汇编编译（通常为了加快速度）、是否支持硬件或者某种特定的硬件、编译的时候是否生成动态链接库、是否支持动态加载其他动态链接库，等等。

Configure 指令事实上根据配置选项，重新改写和生成了 opensslv. h，opensslconf. h，Makefile. ssl 三个文件，从而达到配置的目的。在 Windows 下，还会生成错误信息提示文件 buildinf. h，该文件在 Crypto 根目录下。

### 5.2.2 Configure 使用方式

Configure 的使用一般有两种方式，一种是直接的方式，比如在 Windows 系统中直接调用下列指令：

```
Perl Configure VC-WIN32
```

当然，上面的是简单的方式，事实上，Configure 可以带很多参数，比如要在编译的版本中去掉调用其他动态链接库的功能，就可以使用下面的 Configure 指令：

```
Perl Configure VC-WIN32 no-dso
```

还有一种就是间接的方式，如在 Linux 系统中，就是通过运行 config 指令来调用 Configure 指令的。事实上，在这种情况下，config 指令自动提取了一些环境变量作为 Configure 的参数执行，比如自动识别系统的内核版本和编译器并设定参数等。

### 5.2.3　Configure 参数介绍

Configure 有许多参数，本书只对一些常用的参数进行介绍，对于不常用的参数，读者可以自己通过阅读 Configure 文件获得相应的知识。相信如果你能使用本书没有介绍的参数，那么你已经具备了轻松读懂 OpenSSL 任何文件的能力了。

Configure 的参数形式如下：

```
Configure[no -<cipher> … ][-Dxxx][-lxxx][-Lxxx][-fxxx][-Kxxx][no -hw -xxx|no -hw]
[[no -]threads][[no-]shared][[no-]zlib|zlib-dynamic][no-asm][no-dso][no-krb5][386][- -
prefix = DIR][- -openssldir = OPENSSLDIR][- -with-xxx[ = vvv]][- -test-sanity][- -install__
prefix = PDIR]
```

**（1）openssldir**

参数形式：- -openssldir＝OPENSSLDIR

该参数设置 OpenSSL 的安装目录为 OPENSSLDIR，默认情况下的目录是/usr/local/ssl，如果参数- -prefix 给定了，那么默认情况就是 DIR/ssl，其中 DIR 是由- -prefix 给定的路径参数。如果要设置 OpenSSL 默认的安装目录为 C:/OpenSSL，则应该为：

```
perl Configure VC-WIN32- -openssldir = C:/OpenSSL
```

注意，这里一定要使用“/”作为目录分隔符，这是因为 C 语言里面“\”有特殊作用导致的。

**（2）prefix**

参数形式：- -prefix＝DIR

该参数设置 OpenSSL 的 Lib，Include 和 Bin 目录的前缀目录，默认情况下是 OPENSSLDIR 目录。如果 prefix 和 openssldir 参数都设置了，那么其目录就为 DIR/OPENSSLDIR。例如执行下面的指令配置：

```
perl Configure VC-WIN32-openssldir = OpenSSL-prefix = C:/Local
```

这时候，编译完成后 OpenSSL 默认的安装目录就是 C:/Local/OpenSSL 目录了。与 openssldir 选项设置相同，这里也一定要使用“/”作为目录分隔符。

OpenSSL 还会把- -openssldir 参数值和- -prefix 参数值重复部分去掉，比如执行下面指令：

```
perl Configure VC-WIN32-openssldir = C:/Local/OpenSSL-prefix = C:/Local
```

那么其结果跟上述指令的结果依然是一样的。

使用 prefix 或者 openssldir 设置目录位置选项后，可以选择 OpenSSL 的安装目录，达到改变 OpenSSL 库中默认的路径“usr/local/ssl”为自己需要设定的路径的目的，从而避免在 Windows 平台出现的默认目录问题，方便了使用。

**（3）install_prefix**

参数形式：- -install_prefix＝PDIR

该参数在执行安装任务的时候有效，默认是空的，如果设置了，在 Linux 系统下，它会将安装文件复制到 PDIR/DIR 目录下，这里 DIR 是指使用- -prefix 和- -openssldir 指令设置的 OpenSSL 默认安装路径。使用该参数并不会改变程序里面本身默认的安装路径。

**（4）with-xxx**

参数形式：- -with-xxx=DIR

这个形式的参数目前只有针对 Kerberos 5 协议的功能，一般都是- -with-krb5-xxx 的形式，用来设定 Kerberos 5 协议相关的一些设置。表 5-1 是其几种不同的形式及相应的说明。

表 **5-1** **with-xxx** 形式的参数

| 参数形式 | 参数值 | 功能说明 | 例　子 |
|---|---|---|---|
| - -with-krb5-dir | 目录路径 | 设定 Kerberos 5 协议文件所在的目录，比如如果设定目录为 KDIR，那么 Kerberos 5 协议的 lib 目录和 include目录就应该为 KDIR/lib 和 KDIR/include | - -with-krb5-dir=/usr/local/krb5 |
| - -with-krb5-lib | 目录路径 | 设定 Kerberos 5 协议 lib 文件所在路径。默认的是 KRB5_DIR/lib | - -with-krb5-lib=/usr/local/krb5/lib |
| - -with-krb5-include | 目录路径 | 设定 Kerberos 5 协议头文件所在路径。默认的是 KRB5_DIR/include | - -with-krb5-include =/usr/local/krb5/include |
| - -with-krb5-flavor | MIT 或 Heimdal | 设定使用什么方式的 Kerberos 5 协议，目前支持 MIT 和 Heimdal 两种 | - -with-krb5-flavor=MIT |

**（5）no-hw-xxx 和 no-hw**

参数形式：no-hw 或者 no-hw-xxx

这两个参数用来设定 OpenSSL 模块中是否需要去掉硬件支持。如果使用 no-hw 选项，则所有通过 Engine 接口的硬件加密设备都不会编译进 OpenSSL 的可执行模块中。如果使用 no-hw-xxx，则可以指定不将特定的硬件加密设备编译到 OpenSSL 的可执行模块中。其中 xxx 是硬件加密设备的简要描述名称。比如如果要指定不将 nuron 的硬件加密设备编译到可执行模块，则执行下面的 Configure 指令：

```
perl Configure VC-WIN32 no-hw-nuron
```

如果要去掉所有硬件加密设备支持，则执行下面的 Configure 指令：

```
perl Configure VC-WIN32 no-hw
```

**（6）threads 和 no-threads**

参数形式：threads 或 no-threads

这两个参数用来指明 OpenSSL 的编译库是否支持多线程。默认情况下是支持的。如要用 VC 在 Windows 下编译不支持多线程的 OpenSSL 可执行模块，则执行下面的 Configure指令：

```
perl Configure VC-WIN32 no-threads
```

**（7）shared 和 no-shared**

参数形式：shared 或 no-shared

这两个参数用来指明是否生成动态链接库，但该选项并非对所有平台的编译器都有

效，比如对 Windows 平台的 Borland C++ Builder 编译器就无效。

**（8）zlib，no-zlib 和 zlib-dynamic**

参数形式：zlib，no-zlib 或 zlib-dynamic

zlib 是一种支持数据压缩算法的 API 库，这里可以用 zlib 或者 no-zlib 选择是否将 zlib 的库编译进 OpenSSL 代码中。如果想 OpenSSL 动态调用 zlib 库，则可以使用 zlib-dynamic选项。

**（9）no -asm**

参数形式：no-asm

使用该参数将在编译的过程中不使用汇编编译器。使用汇编编译器能加快编译的速度。

**（10）no -dso**

参数形式：no-dso

使用该参数后，OpenSSL 将去掉支持动态调用其他动态链接库的接口功能，如果使用 OpenSSL 动态接口，那么只是简单返回 NULL。

**（11）no -krb5**

参数形式：no-krb5

使用该参数后，OpenSSL 不会将任何 Kerberos 5 协议的代码和库支持编译到 OpenSSL可执行模块中。

**（12）no -＜cipher＞**

参数形式：no-xxx

该参数用来去掉 OpenSSL 库中特定的加密算法支持。xxx 为算法的简要名称，比如 DES，MD5，RSA 等，也可以为 ssl，ssl2，ssl3 或者 tls1 等。例如要去掉 OpenSSL 中 DES 算法的支持，则执行下面指令：

```
perl Configure VC-WIN32 no-DES
```

**（13）386**

参数形式：386

该选项用来编译适合于 80386 平台的 OpenSSL 代码，默认的代码运行速度更快，但仅仅适合于 486 以上的平台运行。

**（14）其他**

- -test-sanity 选项是用来在代码中生成一些测试代码的选项，仅仅用于调试目的。

其他的选项如-Dxxx，-lxxx，-Lxxx，-fxxx，-Kxxx 等是直接传递到编译器的选项，用来允许你定义额外的宏定义、选用附加的链接库、链接库目录或者其他额外的编译选项等。

## 5.3　基于 Linux 系统的编译和安装

OpenSSL 与 Linux（UNIX）系统有着天然的联系，因为 OpenSSL 开始就是基于 Linux 系统开发的，这也是大多数优秀开放源代码软件的共同点。基于这点，OpenSSL 在 Linux（UNIX）下的编译是最容易和最自动化的，其编译说明文档也是最全面的。

### 5.3.1 Linux 系统编译安装概述

OpenSSL 在 Linux 下的编译安装最终要达到两个目的：首先是编译和链接源代码生成应用程序（指令）、静态链接库和动态链接库，其次是根据环境变量或者 config 指令的配置将 OpenSSL 的应用程序、静态链接库、动态链接库及其他一些配置文件复制到系统指定的安装目录中。所以，在 Linux 系统下编译完 OpenSSL 后，我们会得到一个可执行程序 openssl，两个链接库 libcrypto.a 和 libssl.a，如果你选择编译为动态链接库的模式，还会得到另外两个有用的动态链接库 libcrypto.so.0.9.7 和 libssl.so.0.9.7 文件。此外，在使用 OpenSSL 编程的时候，你还需要用到 include 目录下的头文件。

虽然有各种不同的 Linux 或者 UNIX 系统，但是 OpenSSL 项目组的工作人员做了大量的工作，使得在大部分 Linux 或者 UNIX 系统中都能很轻松地进行 OpenSSL 的编译和安装。这些工作主要体现在 config 文件和前面介绍过的 Configure 文件的脚本指令上。如果你的系统是 OpenSSL 支持的 UNIX 系统（一般都是支持的，如果需要确定，可以查看 Configure 文件），那么可以使用 config 指令自动配置安装环境，当然，如果你有兴趣并且对你的系统充分了解，你也可以像在 Windows 系统中一样使用 Configure 指令手动配置安装环境。

一般来说，在 Linux 下安装 OpenSSL，都需要经过下面几个步骤：

- 使用 ./config 或者 ./Configure 指令配置环境，生成编译安装脚本；
- 使用 make 指令执行编译任务；
- 使用 make test 指令测试编译好的程序；
- 使用 make install 指令安装 OpenSSL 到指定的目录。

在上述的四个步骤中，第二个步骤需要花费的时间是最长的，一般需要几十分钟，具体时间跟你的计算机速度有关。

### 5.3.2 准备安装 OpenSSL

如果你已经下定决心要进入到 OpenSSL 的世界，首先欢迎你成为 OpenSSL 的使用者，也欢迎你成为“中国 OpenSSL 专业论坛”的常客。接下来的任务，就是下载一个 OpenSSL 源码包的最新版本，可以到 OpenSSL 的官方网站 www.OpenSSL.org 下载，也可以到中国 OpenSSL 专业论坛 www.OpenSSL.cn 下载。假如你下载的是 openssl-0.9.7 版本，那么执行下面指令将 OpenSSL 解压：

```
tar-xzvf openssl-0.9.7.tar.gz
```

之后你就会得到一个名为 openssl-0.9.7 的目录，进入这个目录，是不是发现里面的很多名称在第 4 章已经介绍过？这就是你要执行安装指令的根目录了。

当然，不要着急，在开始正式执行安装指令之前，你还必须确保你的系统具备下述这些条件：

- 安装了 make 程序；
- 安装了 Perl 5；
- 安装了一个 ANSI C 编译器，一般默认是 gcc；
- 具备了开发环境，包括具备了必须的静态函数库和头文件；

- 所用的 UNIX 系统是 OpenSSL 所支持的。

当然，如果你使用的是主流的 Linux 系统或者 UNIX 系统，这些条件一般都是满足的。如果确定你具备了这些条件，那么就可以开始编译和安装 OpenSSL 了。

### 5.3.3　OpenSSL 编译安装步骤

**(1)** 快速安装

如果对 OpenSSL 的安装没有特殊的要求或者刚刚接触 OpenSSL，也就是说，你能够允许使用 OpenSSL 的默认安装模式，那么事情就变得非常简单：你只要按顺序执行下面的指令，那么就一切 OK 了。

① ./config

② make

③ make test

④ make install

需要注意的是，在执行上述最后一个指令时，你必须确保你具有 root 权限，否则安装可能会出错。成功执行上述这些指令后，在你执行指令的根目录下，你就会发现名为 libcrypto.a 和 libssl.a 的库文件，然后，如果进入到 apps 子目录中，有一个 openssl 可执行程序，运行下面指令：

```
./openssl
```

如果出现了下面的提示符：

```
OpenSSL>
```

那么恭喜你，OpenSSL 已经安装成功了，可以开始享用你的劳动成果了。事实上，OpenSSL 所有使用的东西都安装在默认的路径/usr/local/ssl 目录下面，进入该目录，首先会发现一个 openssl.cnf 文件，这就是以后经常会用到的 OpenSSL 配置文件，主要是用于证书生成和管理方面的。Bin 子目录存放了 OpenSSL 可执行程序，也就是指令；Include目录存放了使用 OpenSSL 开发的时候需要的头文件；Lib 目录存放了 libcrypto.a 和 libssl.a 两个库文件；Man 目录是 OpenSSL 的使用文档。

**(2)** 根据系统配置 **OpenSSL** 编译选项

很多时候我们会有自己的想法，在安装 OpenSSL 的时候也不例外，比如，你可能想将 OpenSSL 安装到别的目录而不是默认的/usr/local/ssl 目录。OpenSSL 的开发者能够理解你的这种想法，所以也提供了实现你这种想法的途径。通过 config 指令或者 Configure 指令，你可以定制你的 OpenSSL 套餐，当然，既然要个性化，当然就需要了解一些规则。

- 使用 config 指令自动配置 OpenSSL 系统编译选项。

config 指令是首先的选择，它能够自动识别你的系统内核和一些配置，比如编译器，但是又给了你不少灵活的选项。config 指令执行的方式如下：

```
./config[option]
```

OpenSSL 的 0.9.7 版本中有效的选项在表 5-2 中列了出来，你会发现，所有的选项都是前面介绍过的 Configure 选项中存在的，不过选项减少了一些。这很正常，因为 config 指令就是调用了 Configure，不过它首先自动收集了一些系统的信息作为调用 Configure 的参数。所以，config 指令所有这些参数的格式和用法也和 Configure 同名参数的格式和

用法是相同的。

表 5-2 config 指令的选项

| 选　项 | 功　能 |
| --- | --- |
| - -prefix | 设置 OpenSSL 的 Lib，Include 和 Bin 目录的前缀目录 |
| - -openssldir | 设置 OpenSSL 的 Lib，Include 和 Bin 目录的上级目录 |
| no-threads 和 threads | 设置编译后的 OpenSSL 是否支持线程 |
| no-zlib，zlib 和 zlib-dynamic | 设置 OpenSSL 是否编译压缩算法库 zlib 及编译的方式等 |
| no-shared 和 shared | 设置 OpenSSL 是否编译成动态链接库 |
| no-asm | 使用该选项将不使用汇编编译器 |
| no-<cipher> | 设置不支持的算法 |
| 386 | 编译适用于 80386 芯片的 OpenSSL 代码 |
| -Dxxx，-lxxx，-Lxxx，-fxxx，-Kxxx | 给编译器传递宏定义、静态链接库、静态链接库目录及其他附加编译选项的参数 |

事实上，config 指令相比于 Configure 指令的唯一不同之处就是它要自动收集系统的信息或者编译器的信息，如果你对 config 指令的能力有怀疑，那么可以输入下面的测试指令：

```
./config-t
```

在本文作者的平台下，它的输出如下：

```
[root@ann425 openssl-0.9.7]#./config-t
Operating system: i586-whatever-linux2
Configuring for linux-pentium
/usr/bin/perl./Configure linux-pentium
```

可以看到，config 指令输出了系统平台的信息，并准备使用 ./Configure linux-pemtium参数调用 Configure 指令。

如果你要改变默认的安装目录，那么就以下面的形式执行 config 指令：

```
./config- -prefix=/usr/local- -openssldir=/usr/local/openssl
```

因为 OpenSSL 的动态链接库事实上还处于测试阶段，所以 OpenSSL 在 Linux 系统中默认的情况下是以静态链接库的方式编译的。如果要在 Linux 下将 OpenSSL 编译成动态链接库的形式，那么在执行 config 指令的时候采用下面的指令：

```
./config shared
```

使用上述指令在执行 make 指令后，在 OpenSSL 源码包根目录下面会生成 libcrypto.so.0.9.7 和 libssl.so.0.9.7 两个动态链接库，执行 make install 指令之后，在/usr/local/ssl/lib 指令下面也有这两个库。但是，要正确执行 OpenSSL，最好将这两个动态链接库复制到系统/lib 目录下。

- 使用 Configure 指令手动配置系统编译选项。

有时候你可能并不希望 OpenSSL 自动配制系统编译选项，比如，你想使用某个特定

的编译器编译 OpenSSL 的时候。这时候你可以使用 Configure 脚本指令手动输入系统配置参数。OpenSSL 支持相当多的平台、硬件和编译器的混合编译体制，你肯定会想知道 OpenSSL 能够支持那些组合，那么只要简单输入下面的指令就能得知：

```
./Configure
```

可以看到，输出的选项非常多，它们一般表明了系统平台、硬件和编译器的信息。使用 Configure 手动配置系统参数的时候，你输入的选项应该是其中之一。例如在 Intel 奔腾 166 安装 RedHat Linux 系统的计算机上，就应该输入下面的选项：

```
./Configure linux-pentium[option]
```

option 是前面介绍过的 Configure 指令的其他参数选项，可以根据自己的需要填写。

如果你真的那么不幸使用的平台是 OpenSSL 不支持的，那么你就要自己修改 Configure脚本文件了。不过，只要那个平台支持 ASNI C 编译器，问题总是能解决的。Configure 在 Linux 平台最后以 Makefile.org 为原始文件生成 Makefile.ssl 文件，同时也以/crypto/opensslconf.h.in 文件为原本，在其上增加一些宏定义形成/crypto/openssl-conf.h 文件。

**(3) 编译链接 OpenSSL 源代码**

在使用 config 或者 Configure 指令配置好 OpenSSL 之后，只要简单输入 make 指令就可以进行 OpenSSL 的编译和链接了，这个过程通常要持续几十分钟。编译完成后，在 OpenSSL 源代码的根目录会生成 libcrypto.a 和 libssl.a 两个静态链接库，如果你使用了动态链接库编译选项，那么同时还会得到 libcrypto.so.0.9.7 和 libssl.so.0.9.7 两个文件，而 OpenSSL 的应用程序（指令）则在 apps 目录下。

编译 OpenSSL 的过程如果出现错误，一般来说都不会是 OpenSSL 代码包的问题，比如可能是你的系统本身的头文件不全导致的。有些时候，使用汇编编译器会导致错误，这时候可以使用 no-asm 选项重新配置编译试试。如果编译不通过，OpenSSL 一般会给出错误信息，可以通过详细分析这些信息来获得解决的办法，当然，到“中国 OpenSSL 专业论坛”提问是一个解决你的问题的很快的方法。

**(4) 测试编译后的 OpenSSL**

如果编译已经成功完成，那么接下来就可以运行 make test 指令测试 OpenSSL 是否确实已经安装好了。如果有问题需要重新编译，可以执行下列指令：

```
make clean
make
```

**(5) 安装 OpenSSL**

在测试通过之后，就可以放心地进行 OpenSSL 的安装了，安装的目录是你使用--prefix或者--openssldir 设定的目录或者默认的/usr/local/ssl 目录。安装执行下面的指令：

```
make install
```

执行安装指令后，如果指定的安装目录不存在，安装程序首先会创建该目录。然后，安装程序在该目录下创建表 5-3 所示的目录。

表 5-3 安装程序创建的目录

| 目录名 | 功 能 |
|---|---|
| Certs | 初始化的时候是空目录，用于存放证书文件的默认目录 |
| Misc | 存放一些脚本指令文件 |
| Man | 存放 OpenSSL 的说明文档 |
| Private | 初始化的时候是空目录，用于存放私钥文件的默认目录 |
| Lib | 存放 OpenSSL 静态链接库和动态链接库 |
| Bin | 存放 OpenSSL 的应用程序 |
| Include | 存放使用 OpenSSL 的静态链接库的时候需要的头文件，以 openssl 目录存放 |

注意，在使用 OpenSSL 进行编程的时候，头文件包含应该使用以下形式：

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

而不是：

```
#include<ssl.h>
```

这样可以避免跟其他同名的头文件产生冲突。

如果你想将 OpenSSL 安装目录下的程序复制到别的目录，那么你可以使用下面的指令：

```
make INSTALL_PREFIX=/tmp/openssl install
```

注意，这样虽然能改变安装文件的路径，但是并不会改变 OpenSSL 应用程序使用--prefix和--openssldir 参数设置的安装路径或者默认的路径。这个功能的启用，仅仅是为了方便可执行程序打包等其他用途而已。比如，我们执行了下面的系列指令：

```
./config
make
make test
make INSTALL_PREFIX=/tmp/package
```

那么，执行完后，OpenSSL 的安装后的文件是在/tmp/package/usr/local/ssl 目录下，但是如果运行 openssl 应用指令，查找默认的 openssl.cnf 的文件路径依然是/usr/local/ssl 目录，而不是/tmp/package/usr/local/ssl 目录。事实上，在 Configure 指令中也存在设定这个属性的参数，就是--install_prefix，其效果是一样的。

## 5.4 基于 Windows 系统的编译和安装

OpenSSL 在 Windows 系统下的编译安装比起 Linux 系统来说，问题会多一点，不过，既然 OpenSSL 可以在 Windows 系统下使用，成功对于你来说就是迟早的问题，所以不用担心。

### 5.4.1 OpenSSL 在 Windows 系统编译概述

OpenSSL 可以在所有的 Windows 系统中编译成功，而且方法都是一样的。当然，16 位的操作系统跟 32 位的 Windows 操作系统编译方法有一些区别。本节只介绍在 32 位操

作系统如 Windows 98、Windows 2000 和 Windows XP 之上的 OpenSSL 编译方法。16 位 Windows 操作系统的编译方法可以参照 32 位 Windows 系统的方法和 OpenSSL 的安装文件做简单的配置修改就可以。

在 Windows 下进行 OpenSSL 的编译，目前来说有四种官方提供的方法：Visual C++，Borland C，Mingw32 和 Cygwin。其中后两种方法事实上都是使用了 GNU C 编译器进行编译。此外，本书还将介绍一种使用第三方的软件包进行编译的方法，也就是 Andrew写的 VC OpenSSL 编译工程。

在进行 OpenSSL 安装之前，一般来说，必须确保你的 Windows 操作系统中已经安装 Windows 版本的 ActiveState Perl，除非你使用 Cygwin 进行编译，否则编译的时候 Perl 是一定需要的，Perl 可以从 http://www.activestate.com/ActivePerl 下载。此外，你还必须具备以下软件之一：

- Visual C++；
- Borland C 或者 Borland C++ Builder；
- GNU C 编译器（Mingw32 或者 Cygwin）。

如果你想提高编译的速度，节省时间，那么可以安装汇编编译器，使用 OpenSSL 的汇编功能。目前来说，OpenSSL 支持以下两种汇编编译器：

- Microsoft 的汇编编译器 MASM；
- 免费的 Netwide 汇编编译器 NASM。

Microsoft 的 MASM 一般会随 VC 发布，或者随一些 DDK 发布，但是，它的名称不一定是一样的，一般为 xxxxml.exe 和 xxxxml1.err 的形式，下载安装后要将它们的名称改为 ml.exe 和 ml.err，否则会导致编译失败。NASM 是免费获取的，可执行文件名称是 nasmw.exe，可以从很多网上下载。

在 Windows 下编译 OpenSSL，一般需要下面这些步骤（使用 GNU C 编译器除外）：

① 使用 perl Configure [option] 指令配置编译变量和选项；

② 使用 ms \ do_ms 等类似指令生成编译脚本文件；

③ 使用编译器进行代码的编译链接。

可以看到，在 Windows 系统下，OpenSSL 并没有提供安装程序，这需要使用者自己配置使用环境。

### 5.4.2　使用 VC 编译 OpenSSL

使用 VC 进行 OpenSSL 编译之前先要安装 Perl，Perl 在这里的作用主要是为了执行 Configure 脚本文件，执行完后，它就不再起作用，在 OpenSSL 的使用过程中也不需要使用 Perl（除非你要使用其提供的文档格式转换程序如 pod2text 等）。

使用 VC 编译 OpenSSL，首先需要打开指令行环境，单击 Windows 左下角【开始】，然后单击【运行】，输入“cmd”，就出现了一个文本输入界面，这就是 Windows 的指令行环境。编译 OpenSSL 的所有操作都应该在同一个这样的环境中进行。

在进入 OpenSSL 的编译环境之前，你首先需要配置好指令行环境的 VC 环境变量，这通常可以通过在指令行环境中执行 VC/Bin 目录下的 vcvars32.bat 文件来实现，如下：

```
D:\Program Files\Microsoft Visual Studio\VC98\Bin>vcvars32
```

执行这个指令的目的主要是配置好 VC 的路径等系统环境变量，使得使用下面的指令编译的时候能正确定位指令程序的位置。执行上述指令后，如果提示信息里没有错误信息，则表示顺利执行。如果有错误信息，则必须找到出错的原因并解决，比如，有的系统可能会提示环境变量空间不足的错误，这时候就需要进行调整。如果有问题，可以到“中国 OpenSSL 专业论坛”寻找答案或提问，一般都能得到解决。在做完以上这些准备工作之后，就可以进入 OpenSSL 的安装程序了。需要注意的是，以下所有操作还是要在同一个指令行环境下执行，否则前面所做的工作就无用了。OpenSSL 的 VC 编译程序如下：

① 使用 perl Configure VC-WIN32 [option] 指令配置编译参数和选项；

② 使用 do_ms、do_masm 或者 do_nasm 批处理指令生成编译脚本文件；

③ 使用 nmake 指令进行代码编译链接。

**(1)** 配置编译参数

配置编译参数是进行 OpenSSL 编译的第一步，通常来说，这一步可以确定系统的环境，使用什么编译器、默认安装路径及其他一些选项等。如果你不需要对 OpenSSL 默认的配置进行改变，那么使用 VC 编译 OpenSSL，只需要执行下面的指令：

```
perl Configure VC-WIN32
```

但是，在 Windows 下使用这个指令，存在一些问题。首先就是默认路径的问题，OpenSSL 默认的安装路径是“/usr/local/ssl”，所以如果调用 OpenSSL 的指令行或者使用到相关函数而没有指定配置等相关文件路径，会使用默认路径，这样，在 Windows 平台下就容易导致错误，比如执行 openssl req 指令，如果不指定 openssl.cnf 文件的路径，就会报错。为了解决这个问题，可以在执行 Configure 指令的时候设定 Windows 形式的路径，如下：

```
perl Configure VC-WIN32- -prefix = C:- -openssldir = OpenSSL
```

这样，OpenSSL 的默认安装路径就是“C:/OpenSSL”目录，比如 openssl.cnf 文件的默认路径就是“C:/OpenSSL/openssl.cnf”。只要编译好之后将 OpenSSL 的相关使用文件复制到“C:/OpenSSL”目录，就可以避免经常出现的文件路径出错问题。

OpenSSL 包含了大量的算法，在某些应用情况下，你可能只需要使用其中的某些算法，这时候，为了降低 OpenSSL 编译后的动态链接库占用的空间，可以使用 Configure 指令将不需要的算法去掉。例如如果要去掉对 DES 算法的支持，则执行如下配置指令：

```
perl Configure VC-WIN32 no-DES
```

Configure 指令可以同时去掉多个算法，比如，你要去掉 DES 算法，同时还要去掉 SSLv1 支持的所有算法，那么可以使用下面的配置指令：

```
perl Configure VC-WIN32 no-DES no-ssl1
```

如果你不需要硬件的支持，那么可以使用下面的配置指令：

```
perl Configure VC-WIN32 no-hw
```

某些情况下，你的 OpenSSL 库可能只会使用某种特定的加密硬件，那么其他硬件支持就是多余的，那么你可以使用下面的配置指令进行配置：

```
perl Configure VC-WIN32 no-hw-xxx
```

其中，xxx 是 OpenSSL 中硬件的名称简写。

(2) 生成批处理文件

在使用 Configure 脚本配置好编译参数后，就可以使用批处理指令生成编译脚本。生成编译脚本根据采用编译器的不同通常使用不同的批处理文件，目前来说使用 VC 编译的时候有三种选择：do_ms，do_masm 和 do_nasm。事实上，这三个指令任何一个都是用来创建一系列编译脚本文件，即 .mak 脚本的。

do_ms 批处理指令的批处理文件是 ms 目录下的 do_ms.bat 文件。使用该批处理指令，编译过程中只使用了 VC 的编译器，而不会使用任何汇编编译器，一般来说，这样的编译速度会比较慢，通常要几十分钟。不过，因为没有使用其他汇编编译器，所以出错的机会就少了，也不用考虑配置汇编编译器的环境变量等问题。对于使用后面两种方法编译不过去的读者来说，do_ms 指令是一种选择。执行该指令的方法是在 OpenSSL 根目录下执行如下指令：

```
ms\do_ms
```

如果你有微软的汇编编译器 MASM，那么就可以使用 do_masm 批处理指令，该批处理指令的文件是 ms 目录下的 do_masm.bat。需要注意的是，OpenSSL 编译脚本默认的 masm 编译器名称是 ml.exe，默认错误处理文件是 ml.err，所以你需要将名为 xxxxml.exe 和 xxxxml.err 的文件更改为 ml.exe 和 ml.err 文件，然后配置 PATH 环境变量，使得在当前指令行环境中能定位 ml.exe 文件。使用 MASM 编译器后，能够大大提高 OpenSSL 的编译速度，执行该指令的方法是在 OpenSSL 根目录下运行下面指令：

```
ms\do_masm
```

使用汇编编译器提高 OpenSSL 编译速度的另一个选择是使用 NASM 编译器，该编译器可以免费获得。这时候你需要使用的编译指令是 do_nasm，该指令对应的批处理文件是 ms 目录下的 do_nasm.bat 文件。使用该编译指令，先要下载 masmw.exe 程序，并设定好该程序的 PATH 环境变量，确保编译的时候能够定位 masmw.exe 程序，然后在 OpenSSL 的根目录下执行如下指令：

```
ms\do_nasm
```

上述三种方法中，每次编译只能选择其中一种，后面两种都采用了汇编编译器，提高了编译速度，但是需要配置的环境也复杂一些。而 do_ms 方式则因为没有采用汇编编译器编译，编译速度比较慢，但是比较简单易行。无论采用哪种方法，编译后的 OpenSSL 使用方法和效率都是一样的，没有什么区别，它们仅仅影响编译的过程。

(3) 代码编译

完成上面的所有步骤后，就可以安心地进行 Windows 下编译 OpenSSL 的最后一步了。不过，这一步虽然执行指令比较简单，但是一般出错都会体现在这一个步骤中。这些错误一般都是前面的不正确配置导致的，你需要耐心地检查以确保前面的步骤正确执行。

前面执行完 do_ms，do_masm 或者 do_nasm 指令后，通常会在 ms 目录下生成一系列编译脚本文件，当然，在 Windows 下，我们仅仅关心其中的两个脚本文件：nt.mak 和 ntdll.mak。

如果我们需要编译后的 OpenSSL 库是支持动态 DLL 形式的，那么应该使用 ntdll.mak文件进行编译。这样，编译完成后我们会得到四个与 OpenSSL 的 API 库有关的文件：ssleay32.lib、libeay32.lib、ssleay32.dll 和 libeay32.dll。执行的编译指令形式

如下：

```
nmake-f ms\ntdll.mak
```

如果你不希望以动态链接库的形式使用 OpenSSL，那么可以使用 nt.mak 文件进行编译。这样，编译后使用 OpenSSL 的时候，会直接将代码链接进你的程序里面。执行的编译指令如下：

```
nmake-f ms\nt.mak
```

在使用 nmake 指令编译的时候，可能会遇到一些错误从而导致编译不成功，这时候首先应该确认前面的配置步骤你已经正确执行，如果还是有问题，可以看 OpenSSL 本身的 FAQ 和 INSTALL.W32 文件里面的一些建议。最后，问题如果还没有解决，可以参加“中国 OpenSSL 论坛”的讨论。

### 5.4.3 使用 BC 编译 OpenSSL

除了 Visual C++，你还可以选用 Borland C 的系列产品来编译 OpenSSL，目前有效的版本包括 Borland C++ Builder 3，Borland C++ Builder 4 和 Borland C++ Builder 5。跟使用 Visual C++编译一样，使用 Borland C++ Builder 同样需要在 Windows 的指令行环境进行编译工作，进入指令行环境的方法请参考 VC 编译部分。

使用 Borland C++ Builder 系列编译 OpenSSL 的时候，需要 NASM 编译器，所以你必须先下载该编译器，并且在 PATH 环境变量中设置好 nasmw.exe 文件的路径。其次，你也需要设定 Borland C++ Builder 编译器的执行程序路径的 PATH 环境变量。如果你使用的是 Borland C++ Builder 5 版本，那么路径环境一般在安装的时候已经设定，不需要做路径设置工作；但是如果你使用的是 Borland C++ Builder 3 或者 Borland C++ Builder 4，那么你就需要在指令行环境中设置好它们的编译程序所在的路径，一般是其 Bin 目录。

完成上面的工作之后，在同一个指令行环境中，你就可以进入编译 OpenSSL 的旅程了，使用 Borland C++ Builder 编译 OpenSSL 的一般步骤如下。

① 如果是 Borland C++ Builder 5，运行 perl Configure BC-32 [option] 指令；如果是 Borland C++ Builde r3 或者 Borland C++ Builder 4，则设置编译执行文件路径和 Bin 目录的 PATH 环境变量。

② 如果是 Borland C++ Builder 5，运行批处理指令 do_nasm；如果是 Borland C++ Builder 3 或者 Borland C++ Builder 4 则运行 bcb4。

③ 执行 make-f ms \ bcb.mak 指令编译链接 OpenSSL 代码。

因为使用 Borland C++ Builder 编译 OpenSSL 的模式目前只提供了产生静态链接库的形式，如果想要使用动态链接库，那么需要在完成上面第②个步骤后自己修改 bcb.mak 文件。所以，使用 Borland C++ Builder 编译完成后，生成的库文件仅仅是两个静态链接库文件 libeay32.lib 和 ssleay32.lib，而没有动态链接库文件。

下面，我们主要介绍使用 Borland C++ Builder 5 进行 OpenSSL 编译的过程。

**(1)** 配置编译参数

如果你使用的是 Borland C++ Builder 5，那么可以跟使用 VC 一样使用 Configure 脚本指令配置 OpenSSL 编译程序的选项，如果使用的是过时的 Borland C++ Builder 3 或者 Borland C++ Builder 4，那么你只能修改 bcb4.bat 文件以达到你定制的目的了。

一般情况下，如果你不需要对 OpenSSL 默认的配置做任何修改或者你想尽快使用 Windows 下的 OpenSSL，那么你只需要简单执行下面的指令：

```
perl Configure BC-32
```

但是这样编译的 OpenSSL 在 Windows 平台上使用起来可能不是特别方便。首先就是默认安装路径问题，默认的 Linux 下的“/usr/local/ssl”路径会导致许多程序默认选项不能正确执行，这时候你可以更改默认的 OpenSSL 路径为 Windows 的风格，如执行下面的指令：

```
perl Configure BC-32- -prefix = C:- -openssldir = OpenSSL
```

这样，默认的 OpenSSL 安装路径就是“C:/OpenSSL”了，比如，默认的 openssl. cnf 路径就是“C:/OpenSSL/openssl. cnf”，只要将 OpenSSL 编译好的文件复制到该目录下就能够正确执行。

此外，跟使用 VC 进行编译一样，你同样可以选择是否不支持某些算法或者不支持某些硬件以降低编译后库文件的大小，这些指令形式如下：

```
perl Configure BC-32 no-DES
perl Configure BC-32 no-DES no-ssl1
perl Configure BC-32 no-hw
perl Configure BC-32 no-hw-xxx
```

参数的意义和详细使用方法可以参考 VC 编译 OpenSSL 的相关章节介绍，这里不再一一复述。

**(2)** 生成批处理文件

跟使用 Visual C++编译 OpenSSL 不同，使用 Borland C++ Builder 编译 OpenSSL 并没有那么多的灵活选择。对于使用 Borland C++ Builder 5 版本的读者来说，应该使用以下的批处理指令生成编译脚本文件：

```
ms\do_nasm
```

对于使用 Borland C++ Builder 3 或者 Borland C++ Builder 4 的读者来说，则执行下面的批处理文件：

```
ms\bcb4
```

**(3)** 代码编译

完成上面的工作之后，简单执行下面的指令就可以完成编译链接 OpenSSL 的工作：

```
make-f ms\bcb. mak
```

编译完成后，可以得到编译好的 OpenSSL 的应用程序、两个 OpenSSL 的静态链接库。当然，编译也会因为前面配置或者协调的原因失败。如果出现这种情况，请首先参考 OpenSSL 本身的 FAQ 和 INSTALL. W32 文件，如果还解决不了问题，那么请到本书的支持站点“中国 OpenSSL 专业论坛”参与讨论，会得到很多 OpenSSL 使用者的热心帮助。

### 5.4.4　使用 VC6OSSL 编译 OpenSSL

**(1) VC6OSSL 介绍**

VC6OSSL 全称为“Developer Studio Workspace and Project files for openssl-0. 9. 7”，

是 Andrew Gray 做的工作，该工程可以从网站 http://www.iconsinc.com/~agray/ossldev 下载。为了简单起见，我们以后简称该项目工具包为 VC6OSSL。VC6OSSL 将 OpenSSL 所有的可执行程序和代码封装在一个 VC6 的 Workspace 里面，每个可执行的应用程序和链接库都是一个独立的项目，使用者可以使用熟悉的 VC6 项目编译 OpenSSL。

使用 VC6OSSL 完成 OpenSSL 编译后，会将 OpenSSL 每个应用程序（指令）分别生成 Windows 下独立的可执行文件，比如 CA 指令，会生成 ca.exe 文件。这给使用者增加了更大的灵活性。而且，因为 VC 编译工程里面是将每一个可执行指令都作为一个单独的项目组织的，所以你可以对单个指令在 VC 下进行 Debug 调试，对于编程人员来说，这也是一个深入了解 OpenSSL 的很好工具。

VC6OSSL 支持的版本包括 0.9.7a 版和 0.9.6 的各个版本。其中 0.9.6 各个版本都提供了普通编译、MASM 和 NASM 三种编译方式，而 0.9.7a 目前仅仅提供了普通的不带汇编的编译方式。

VC6OSSL 编译完成后，将在 out32dll 目录下生成 DEBUG 和 RELEASE 两个目录，分别是 Debug 版和 Release 版的 OpenSSL 应用程序和链接库。

**(2) 使用 VC6OSSL**

VC6OSSL 本身并没有提供 OpenSSL 的源文件，所以使用 VC6OSSL 之前首先需要下载 OpenSSL 源码包并解压。接着，将 VC6OSSL 解压到 OpenSSL 源码包的根目录，这将会在 OpenSSL 根目录下创建一个子目录 MSVC097，如果是 0.9.7 以前的本版，创建的目录名称是 MSVC。

跟使用其他 Windows 平台工具编译 OpenSSL 的方法一样，你首先也需要安装 Perl，然后进入到指令行环境执行 OpenSSL 的编译参数设置工作。关于这些步骤的详细细节，可以参考 VC 编译 OpenSSL 的章节部分。

首先，在指令行环境中执行 OpenSSL 的 Configure 脚本指令，方式如下：

```
perl Configure VC-WIN32
```

当然，在上述指令中，一样也可以根据你的要求进行设定，比如默认路径等问题。

接着，使用 OpenSSL 的批处理文件生成编译脚本文件。一般来说，如果你不使用汇编编译器（VC6OSSL 的 0.9.7 版本目前只提供非汇编方式），那么可以执行下述的指令：

```
ms\do_ms
```

如果你使用的 OpenSSL 的版本是 0.9.6 的，那么还可以下载对应 VC6OSSL 版本的 MASM 或者 NASM。如果使用的 VC6OSSL 是 MASM 版本的，那么执行下述指令：

```
ms\do_masm
```

如果使用的 VC6OSSL 是 NASM 版本的，则执行：

```
ms\do_nasm
```

需要注意的是，如果使用 MASM 汇编方式，你需要确保 MASM 的可执行文件和错误处理文件的名字已经更改为 ml.exe 和 ml.err，并已经在系统的 PATH 环境变量中设置了 MASM 可执行程序的路径。如果使用 NASM 汇编方式，需要下载 NASM 的可执行程序 nasmw.exe 并在系统的 PATH 环境变量中设置该程序所在的路径。

完成上述的指令后，在 OpenSSL 根目录下执行下面的指令：

```
perl msvc097\doinc.pl
```

如果是 0.9.7 以前的版本，则执行：

```
perl msvc\doinc.pl
```

该指令会创建编译后输出文件需要的目录，以及将一些头文件复制到指定的地方等操作，从而为使用 VC6OSSL 编译 OpenSSL 做好准备工作。

最后，你打开 OpenSSL 根目录下的 MSVC097（或者 MSVC）目录，然后双击 Openssl.dsw 文件开启 VC6 的编译环境，选择“批构建”（Batch Build），就可以等待 OpenSSL 编译的成功结果了。

### 5.4.5　其他在 Windows 系统中编译 OpenSSL 的方法

如果你有足够的理由不愿意使用上面介绍的三种方法在 Windows 平台安装OpenSSL，那么还有两种方法可以供你选择：Mingw32 和 Cygwin。这两种方法都是使用 GNU C 编译器进行编译。

**（1）使用 Mingw32 编译 OpenSSL**

首先，需要下载 Mingw32，并将软件包解压缩到某个目录。然后开启 Windows 的指令行工作环境，进入到 Mingw32 所在的目录，运行以下指令设置 Mingw32 的 PATH 环境变量：

```
Mingw32
```

然后，进入到 OpenSSL 的根目录，执行：

```
ms\mingw32
```

如果出现错误，可是试试非汇编方式：

```
ms\mingw32 no-asm
```

如果这个指令成功完成，那么将会生成 libcrypto.a 和 libssl.a 两个静态链接库。如果你要使用 DLL 的动态链接库，那么就应该使用 libeay32.a 和 ssleay32.a 两个静态链接库作为链接库，它们对应的动态链接库是 libeay32.dll 和 ssleay32.dll。

**（2）使用 Cygwin 环境编译 OpenSSL**

Cygwin 是一个在 Windows 9x，Windows NT，Windows 2000 及 Windows XP 平台虚拟一个 Linux 运行环境的软件。这样，可以像在 Linux 下一样在 Cygwin 环境中编译 OpenSSL 软件包。执行的指令步骤如下：

① ./config

② make

③ make test

④ make install

当然，在进行编译之前，也要确保在 Cygwin 中安装了 Perl，否则也会导致脚本指令不能正确执行。

### 5.4.6　安装和使用 OpenSSL

OpenSSL 在 Windows 下没有提供安装的方法，所以，就其本身来说，并没有安装的概念，但是为了方便使用 OpenSSL，对其目录做规范的整理也是必要的。OpenSSL 编译后，使用 OpenSSL 一般有两种方式，即使用 OpenSSL 的指令行方式和使用 OpenSSL 的

链接库进行编程。所以，我们在设置 OpenSSL 在 Windows 下使用的目录时，需要考虑这两种应用的情况。事实上对 OpenSSL 目录的安排并没有统一的标准，这里只是给读者提供一些可能的建议。

对 OpenSSL 的目录安排最好在编译 OpenSSL 之前就选定，这主要是为了在编译的时候设定 OpenSSL 的安装目录以使 OpenSSL 在 Windows 下的使用更为方便。例如，如果你想编译后的 OpenSSL 主要目录放置在“C：\ OpenSSL”目录下，那么就可以通过 Configure 的-prefix 和-openssldir 参数设定 OpenSSL 默认的安装目录，这样在编译后，就可以将 OpenSSL 的应用程序和文件的根目录设定为“C：\ OpenSSL”。编译完后，需要做的安装工作如下：

- 建立“C：\ OpenSSL”目录，并建立“bin”，“lib”，“include”目录；
- 将 apps 目录下的 openssl. cnf 文件复制到 C：\ OpenSSL 目录下；
- 将 out32dll 目录下的 openssl. exe，libeay32. dll 和 ssleay32. dll 复制到 C：\ OpenSSL \ bin 目录下；
- 将 out32dll 目录下的 libeay32. lib 和 ssleay32. lib 复制到 C：\ OpenSSL \ lib 目录下；
- 将 inc32 目录下的 openssl 目录复制到 C：\ OpenSSL \ include 目录下。

做完这些工作后，就可以在 C：\ OpenSSL \ bin 目录下使用 openssl. exe 提供的 OpenSSL 指令了。当然，如果你要使用 CA 指令的所有功能，即模拟一个小型 CA，那么还需要在 openssl. exe 文件所在的目录下按要求建立相应的目录结构（如 demoCA）和配置相应的文件，这将在 CA 指令使用的相关章节作详细的介绍。如果要使用 OpenSSL 的库进行编程，还需要在编程环境（比如 VC）中增加 C：\ OpenSSL \ include 目录为头文件目录，增加C：\ OpenSSL \ lib目录为默认的链接库目录。当然，你可能还需要将两个 DLL 头文件复制到应用程序所在的目录。

事实上，为了更方便地使用 OpenSSL 进行编程，可以通过下面的步骤来配置Windows 环境：

- 将 libeay32. dll 和 ssleay32. dll 复制到系统所在目录，如 system32；
- 将 libeay32. lib 和 ssleay32. lib 复制到编程工具默认库文件所在的目录，如 VC 的 . . \ VC98 \ lib 目录；
- 将头文件目录 openssl 复制到编程工具默认头文件所在的目录，如 VC 的 . . \ VC98 \ Include 目录。

这样，不用做任何修改，即能使用 OpenSSL 进行编程。

如果你在编译 OpenSSL 的时候没有使用编译选项改变和设定 OpenSSL 默认的安装目录，那么 OpenSSL 编译完成后，你同样可以使用上面的方法安装和设置 OpenSSL 的使用目录，但是需要在使用 OpenSSL 的应用程序（指令）之前设定 OPENSSL_CONF 环境变量为 openssl. cnf 文件所在的路径，如 openssl. cnf 在 C：\ OpenSSL 目录下，那么应该设定：

```
OPENSSL_CONF = C：\ OpenSSL \ openssl. cnf
```

有两种设定方法，一种是在运行 OpenSSL 的指令行环境中直接输入：

```
D:\OpenSSL\bin>set OPENSSL_CONF = D：\ OpenSSL \ openssl. cnf
```

这样，在当前指令行环境下就可以正常使用 OpenSSL 的指令而不必每个指令都使用-config 选项指定 openssl.cnf 的路径。但是，这种方法在当前指令行环境关闭后设置即无效，再使用新的指令行环境使用 OpenSSL 指令时还需要重新设置。为了避免重复设置，可以用【我的电脑】|【属性】|【高级】|【环境变量】|【新建】来设置 OPENSSL_CONF 变量，这样就可以一劳永逸。可以在指令行环境输入下面指令就显示 OPENSSL_CONF 环境变量的值。

```
D:\OpenSSL\bin> set OPENSSL
OPENSSL_CONF = D: \ OpenSSL \ openssl.cnf
```

如果你在编译的时候没有设定和改变默认安装目录，也没有按上述方法配置 OPENSSL_CONF 环境变量，那么在使用一些指令的时候（如 req，ca 等）就需要使用-config 选项指定 openssl.cnf 文件的完整路径，否则程序可能没有办法正确执行。

使用 OpenSSL 的链接库进行编程的时候，必须使用动态的多线程库（MultiThreaded DLL）作为链接库，否则可能导致 OpenSSL 在执行某些 I/O 接口操作的时候产生异常。这是由于 OpenSSL 的内存分配机制可能的混乱导致的，解决的办法除了上述的之外，还可以在应用程序使用 OpenSSL 的任何库函数之前调用 CRYPTO_malloc_init（）函数进行初始化。

## 5.5　基于其他系统的编译和安装

OpenSSL 使用 C 语言开发，使得其具有很强的平台移植性，可以移植到大部分支持 C 语言的平台。部分系统的编译提供了相关的文档，文档名字格式基本上是 INSTALL.xxx，其中 xxx 为系统平台的简称，需要的时候可以参考这些文件。输入下列指令可以查看 OpenSSL 目前支持的编译平台和方式。

```
perl Configure
```

事实上，即便是在 OpenSSL 目前不支持的平台上，在满足一定条件的时候也是可以进行 OpenSSL 的移植的，不过，这需要更多经验和耐心。

## 5.6　本 章 小 结

本章首先介绍了 OpenSSL 编译和安装的总体架构，并介绍了配置指令 Configure 的详细功能和用法。

接着，针对 OpenSSL 在 Linux 平台的编译做了详细的讲解和分析，使得读者具备了在 Linux 下灵活编译和配置 OpenSSL 的能力。

本章还详细介绍了使用 Visual C++，Borland C++ Builder 及 VC6OSSL 在 Windows 平台编译 OpenSSL 的过程、方法和注意事项。

最后，本章介绍了在 Windows 系统下安装和使用 OpenSSL 的方法和注意事项。

# 第 6 章

# OpenSSL 基本概念

## 6.1 配 置 文 件

OpenSSL 的许多应用程序和函数使用配置文件来获取默认的信息和选项，所以，要使用 OpenSSL，尤其是使用 OpenSSL 的指令，就必须先了解 OpenSSL 的配置文件的概念和设置方法。

### 6.1.1 配置文件概述

(1) 配置文件结构和语法

配置文件是 OpenSSL 的一个基础结构组件，OpenSSL 使用一组称为 OpenSSL CONF 的函数来读取 OpenSSL 配置文件的信息。OpenSSL 提供的主配置文件是 openssl.cnf，它集成了 OpenSSL 所要使用的配置文件选项的大部分内容。此外，OpenSSL 还提供了其他一些部分的配置文件，用于专门配置证书请求或者 X.509 v3 证书的扩展项。

OpenSSL 的配置文件使用字段的概念分成不同的部分。一般来说，每个字段的开始使用［Section_Name］来标识，直到下一个字段开始或者到达文件结尾为一个字段的结束。字段的名字可以由字符、数字或者下划线组成。

需要注意的是，OpenSSL 的第一个字段是比较特殊的，它不像其他的字段有名字和类似［Section_Name］的开头。OpenSSL 第一个字段被设置为 OpenSSL 的默认字段，当 OpenSSL CONF 函数要读取配置文件数据的时候，首先会根据字段的名字去查找相应的字段，如果没有找到，则会定位到默认的字段，也即第一个字段。

每个字段包含一组不定数量的变量名和值域数据对，它们的形式如下：

变量名＝变量值

变量名可以由字符、数字和标点符号组成（如“.”、“,”和“_”等）。等号后面数据构成了变量的值，变量值域的结束以当前行结束为标志，并且会自动忽略前面和后面的空格。比如变量式子：

```
Name = Value
```

上式 Value 前面的空格被忽略，Name 的值还是 Value。

变量值域可支持由“\”或者需要用其他引用符号声明的特殊字符，如“\”符号需要这样声明：“\”。此外，变量值域还支持在行末尾使用“\”作为连接符号来使得可以使用多行数据作为一个值域。同样，变量值域还能支持和识别诸如“\n”、“\r”等这样

的特殊制表符。例如，可使用双引号保持字符串前面和后面的空格不被忽略等，下面是一些使用特殊字符的例子。

```
space = " There are spaces "
```

这里，space 的变量值就是 " There are spaces " 而不是 " There are spaces "。可以看到，使用双引号后前后的空格被保留了。又如：

```
longvar = begin\
middle\
end
```

这里，longvar 变量值为 "beigin middle end"，这是因为使用了连接符号 " \ " 的效果。又如：

```
special = including\character\n
```

这里的 special 变量值为 "including \ character \ n"，包含了特殊字符 " \ " 和回车换行符号 " \ n"。

变量值域还支持其他变量值的展开。使用 " $ 变量名" 或者 " ${变量名}" 的形式，可以展开当前字段名为 "变量名" 的变量的值。例如下面的一组定义：

```
dir = ./demoCA
certs = $dir/certs
```

则 certs 的值相当于：

```
certs = ./demoCA/certs
```

如要展开其他字段的变量值，则可以使用 " $ 字段名::变量名" 或者 " ${字段名::变量名}" 的形式展开，如下面的一组定义：

```
[Section1]
home = /home
[Section2]
dir = ${Section1::home}/dragonking
```

则 dir 相当于下式的定义：

```
dir = /home/dragonking
```

OpenSSL 的配置文件将环境变量的存储环境映射为名为 "EVP" 的一个虚拟字段，你可以使用类似于 " $ EVP::变量名" 或者 " ${EVP::变量名}" 的形式来获取系统环境中定义的变量的值，如下面的式子获取了 Windows 环境中默认的系统盘的路径：

```
default = $EVP::HOMEDRIVE
```

其结果在本书作者的 Windows XP 系统中相当于：

```
default = C:
```

在使用环境变量展开的时候，如果要展开的变量不存在，就可能导致错误，比如在 Windows 系统中，就不存在 HOME 这个环境变量，那么诸如 " $ EVP::HOME/OpenSSL" 的路径展开就会导致系统的错误，为了避免这个问题，需要采取预先定义默认值的办法来解决。例如上述的例子中，可以先定义 HOME 的默认值，如果 EVP 中不存在该变量，则采用默认值，下面是一个解决的例子：

```
HOME = ./
```

```
Dir = $EVP::HOME/OpenSSL
```

如果 EVP 中存在 HOME 这个变量，则使用 EVP 中的 HOME 变量展开；如果 EVP 中不存在 HOME 这个变量，则 Dir 值域使用默认的 HOME 值展开为“./OpenSSL”。需要注意的是，默认值一定要在变量展开之前定义，否则无效。

OpenSSL 的变量值域还支持字段嵌入，也就是说，变量值可以为同文件的另一个字段名。例如下面的例子：

```
[Main_Section]
varA = begin
varB = Sub_Section
[Sub_Section]
Ba = A
Bb = B
```

如果在同一个字段中有多个相同名字的变量，那么除了最后一个，前面的同名变量都会被忽略。

在 OpenSSL 的配置文件中，“#”为注释符号，与其同一行且位于“#”后面的所有数据都被视为注释语句。

**(2) openssl. cnf 的内容**

openssl. cnf 是 OpenSSL 提供的主配置文件，它的结构和语法完全遵循上述介绍的 OpenSSL 配置文件的结构和语法。一般来说，它位于 OpenSSL 软件包的 apps 目录下，读者可以使用普通的文本浏览工具打开和编辑该文件。

目前 openssl. cnf 用于 req、ca 和 x509 等指令中，所有使用 openssl. cnf 的指令，都是通过-config 选项来指定配置文件，如果你不指定，默认的就是 openssl. cnf 文件。当然，并非任何时候都能成功找到 openssl. cnf 文件，这取决于你是否合理编译和安装了 OpenSSL。openssl. cnf 文件的内容包括了三大部分：默认的文件配置、证书请求配置及证书签发配置。事实上，OpenSSL 的配置文件应用远远不止于此，你甚至可以在使用 OpenSSL 函数库的时候使用这种方便的配置机制。在后面的章节中，你还会看到如何使用配置文件灵活配置 X. 509 证书的扩展项。

## 6.1.2 配置文件中的通用变量配置

openssl. cnf 文件配置了一些可能用到的通用变量，这些变量数量很少，基本上都是位于 openssl. cnf 文件的主字段部分即默认字段部分。

**(1) 随机数默认文件**

在 openssl. cnf 文件的默认字段部分，读者可以看到有如下的语句：

```
HOME = .
RANDFILE = $ENV::HOME/. rnd
```

RANDFILE 变量是用来设置可能用到的随机数文件，这个文件通常用作生成随机数的种子文件。读者通过本书前面的章节的介绍应该知道，随机数在密码学中具有重要的意义，通常，随机数的好坏可以对密钥的安全性产生很大的影响。大家可以看到，在 RANDFILE 变量之前还定义了 HOME 变量，这主要就是为了防止环境变量 ENV 中不存

在 HOME 变量（如 Windows 系统）的时候可能导致的错误。

**(2)** 扩展对象定义

有时候可能要使用一些扩展对象定义，比如在使用 X.509 证书的扩展项的时候，这些扩展项的对象可能在 OpenSSL 中并没有进行定义，那么就要对扩展对象进行定义。对扩展对象的定义通常包含三个部分：对象数字形式的 OID、对象简称和对象详细描述。OpenSSL 以两种方式定义扩展对象：扩展对象文件方式和在配置文件中使用特定字段方式。

扩展对象文件的指定使用 OpenSSL 默认字段中的 oid_file 变量，形如下面的格式：

```
oid_file = $ENV::HOME/.oid
```

打开 openssl.cnf 文件可以看到，在默认的 openssl.cnf 文件中，该语句是被注释掉的。其中的 .oid 即为扩展对象定义文件。扩展对象定义文件每一行定义一个对象，格式一般如下：

对象数字 ID　［空格］　对象简称　［空格］　对象详细描述

下面就是一个 X.509 v3 扩展项的扩展对象定义文件的实例：

```
2.99999.1       SET.ex1       SET x509v3 extension 1
2.99999.2       SET.ex2       SET x509v3 extension 2
2.99999.3       SET.ex3       SET x509v3 extension 3
2.99999.4       SET.ex4       SET x509v3 extension 4
```

扩展对象还可以在配置文件中的一个字段中定义，这个特定的字段通过 oid_section 变量来指定，变量值为字段名。其格式如下：

```
oid_section = new_section
```

扩展对象字段的变量对通常由扩展对象简称和扩展对象 OID 组成，在这种方式中，扩展对象详细描述跟扩展对象简称是相同的，下面是一个扩展对象使用字段定义的例子：

```
[new_section]
oid_ext1 = 2.99999.1
oid_ext2 = 2.99999.2
```

读者从默认的 openssl.cnf 中可以看到，其没有定义任何扩展对象，如果要使用扩展对象，读者可以参照上面的方法自己定义新的对象。

### 6.1.3　配置文件中的证书请求配置

**(1)** 证书请求主配置字段

在申请证书之前，通常要生成符合 PKCS ＃10 标准的证书请求封装格式。OpenSSL 的 req 指令实现了产生证书请求的功能，其相关选项的信息就采用 OpenSSL 配置文件来进行设置。OpenSSL 中证书请求的配置分成几个字段，包括一个基本字段和几个附属的字段。

OpenSSL 配置文件中名为 req 的字段是证书配置请求的基本字段，也是使用证书请求配置的入口，而其他附属字段都是以这个字段作为引导的。目前，默认 openssl.cnf 文件证书请求配置字段已有的选项如下。

```
[req]
default_bits = 1024
default_keyfile = privkey.pem
```

```
distinguished_name = req_distinguished_name
attributes = req_attributes
x509_extensions = v3_ca
input_password = secret
output_password = secret
string_mask = nombstr
req_extensions = v3_req
```

上述选项并非所有都使用了，比如input_password和output_password就已经被注释。因为req指令还可以生成RSA密钥对，所以default_bits选项指定了默认生成密钥的长度，default_keyfile则指定了默认的私钥输出保存文件。input_password和output_password分别提供了默认的输入私钥保护口令和输出私钥保护口令。string_mask选项则是设定了证书请求的信息字段的字符串类型。distinguished_name，attributes，x509_extensions和req_extensions的值域都是字段名，指定了包含相应信息的字段名字。事实上，与证书请求相关的配置文件选项远远不止这些，表6-1列出了OpenSSL配置文件中目前支持的证书请求参数及其意义，这些参数基本上跟req指令中同名的参数选项作用是相同的，通常如果req指令设置了同名选项，则配置文件的默认值将不会被使用，否则就使用配置文件的默认值。

**表6-1 OpenSSL配置文件中的证书请求参数**

| 参数名 | 参数值域 | 参数作用描述 |
|---|---|---|
| default_bits | 一般是2的次方，比如512，1 024或2 048等 | 生成的证书中的密钥对的长度 |
| default_keyfile | 文件名 | 生成私钥的输出保存文件 |
| input_password | 字符串 | 当要读取输入私钥文件时候的口令 |
| output_password | 字符串 | 保存输出私钥文件的口令 |
| oid_file | 文件名 | 定义扩展对象的文件名 |
| oid_section | 字段名 | 定义扩展对象的字段名 |
| RANDFILE | 文件名 | 随机数种子文件 |
| encrypt_key | yes或no | 如果该参数值设置为no，那么生成的私钥将不会采用加密保护 |
| default_md | md5，sha1，md2或mdc2 | 定义签名使用的信息摘要算法，默认的是MD5 |
| string_mask | 字符类型描述名，比如BMPString等 | 为一些字段指定特定字符串类型，比如证书请求中的城市和组织名称等字段就可以使用这样的方法指定，这使得证书可以支持如汉字这样的多字节字符 |
| | default | 定义了包含PrintableString，T61String和BMPString三种类型的字符串，是默认值 |
| | pkix | 定义了包括PrintableString和BMPString两种字符串类型 |
| | utf8only | 只使用UTF8String类型 |
| | nombstr | 仅使用PrintableString和T61String，不使用多字节的字符串类型如UTF8String和BMPString等 |
| req_extensions | 字段名 | 指定了证书请求扩展字段名，这个字段定义了要加入到证书请求中的一系列扩展项 |

续表

| 参数名 | 参数值域 | 参数作用描述 |
| --- | --- | --- |
| x509_extensions | 字段名 | 指定了生成自签名证书时要使用的证书扩展项字段，该字段定义了一系列要加入到证书中的扩展项 |
| prompt | yes 或 no | 如果该值设定为 no，那么 req 指令不会再要求用户输入一些证书字段的信息，而是直接从配置文件读取该字段的值。相应的，distinguished_name 和 attributes 的格式也要做相应的改变 |
| utf8 | yes 或 no | 如果设置为 yes，那么任何证书域要输入的值都被解析成 UTF8 格式的字符串，而默认的情况下使用的是 ASCII 编码。这就要求所有输入值，不管从指令行界面输入还是从配置文件输入，都应该是 UTF8 格式的，适当地使用该选项能够使证书支持诸如汉字这样的多字节字符 |
| attributes | 字段名 | 指定了定义证书请求属性的字段名，这个字段定义了证书请求的一些属性，典型的如验证密码（challengePassword）和别名（unstructuredName），在 OpenSSL 的证书签发工具中，这些属性是没有用的，但是有些 CA 会要求这些属性。该字段定义变量的格式跟 distinguished_name 字段的格式是一样的，都会受到 promppt 的值的影响 |
| distinguished_name | 字段名 | 定义了输入用户信息选项的“特征名称”字段名，该字段包含了多项用户信息项，其格式跟 attributes 一样，都会受到 prompt 的值影响 |

从表 6-1 可以看到，虽然证书请求字段支持 oid_file，oid_section 及 RANDFILE 等参数，但是默认的 openssl.cnf 文件并没有这些选项，根据使用 OpenSSL 配置文件的规则，配置文件的读取函数将使用主字段也就是配置文件默认字段中同名的参数。

**(2)** 特征名称字段

特征名称字段由 req 字段中的 distinguished_name 指定，它包含了用户的主要信息，如国家、省份、城市、组织及名字等信息，如果两个证书的特征名称字段所有参数值都相同，一般就可以看成是同一个用户，所以该字段所有信息的总和称为“特征名称”。

特征名称字段允许的变量名可以是任何对象的简称或详细描述名称。有些对象在 OpenSSL 里面已经编译进去，比如 commonName，countryName，localityName，organizationName，organizationUnitName 和 stateOrPrivinceName。此外，emailAddress，name，surname，givenName，initials 和 dnQualifier 等也在 OpenSSL 里面做了定义。特征名称字段还可以包含自定义的扩展对象名，这些扩展对象可以在扩展对象文件（oid_file）或者扩展对象字段（oid_section）中定义，一般来说，扩展对象的值类型都被默认为是 DirectoryString。

根据证书请求配置字段中变量 prompt 的值的变化，特征名称字段变量定义的格式有两种。如果 prompt 的值设为 no，那么特征字段中每个条目只是简单包含了变量名和对应的变量值，通常有如下形式：

```
CN = My Name
```

```
OU = My Organization
emailAddress = someone@somewhere.org
```

在这种情况下，不会要求用户再输入这些对象的域值，而只是简单地将配置文件中相应的值复制过去。

相反，如果 prompt 没有定义或者其值没有设置为 no，那么特征名称字段一个变量域的属性就会包含输入信息的属性，一般包含了类似下面的格式：

```
fieldName = "prompt"
fieldName_default = "default field value"
fieldName_min = 2
fieldName_max = 4
```

其中，fieldName 是对象的简称或者详细描述，如 CN 或者 commonName。"prompt" 是要求用户输入变量值的时候显示的提示信息；fieldName_default 的值定义的是默认的变量值；fieldName_min 和 fieldName_max 定义了变量值字符串最小的长度和最大的长度。事实上，这四个项目除了第一个，其他都是可选的。对一个变量来说，用户如果没有输入任何信息，会使用默认的变量值；如果定义了默认变量值但是用户输入了 "." 号，该变量域就会被忽略和取消，不会被包含在证书请求中；如果用户没有输入任何信息，也没有在配置文件定义默认值，那么该变量域也会被忽略和取消。

输入的变量值字符串的长度必须在配置文件定义的最小长度和最大长度之间，这给一些变量值做了额外的限制，比如国家（C），其值一般就定义为 2 个字符串。那么如果输入的值不是两个字符串长度，就会被拒绝接受。

有些变量，比如 organizationName，在特征名称字段可能会不止一个，但是，OpenSSL 的配置文件只会加载同名变量的最后一个变量。为了避免这种情况可能导致的数据丢失，OpenSSL 规定了在特征名称字段中，如果字符后面跟着点号（"."），那么该字符就被忽略。比如 a. fieldName 和 b. fieldName 在 OpenSSL 中都认为是名为 fieldName 的对象。这样，就可以将相同变量名出现多次的情况区别开来，而不会因为相同被 OpenSSL 配置文件的读取函数忽略，从而保证多个变量值的情况能够被正确输入到证书请求的特征名称中。例如对于有两个 organizationName 变量的情况，可以使用如下的格式：

```
0.organizationName
1.organizationName
```

**(3)** 证书请求属性字段

证书请求属性字段由 req 字段中的 attributes 变量指定，它定义了一些在 CA 签发证书的时候可能用到的属性值，现在典型的应用字段是 challengePassword 和 unstructuredName。challengePassword 一般是在用户获取 CA 签发的证书的时候验证用户时使用的，而 unstructuredName 则是一个可选的别名。这些属性目前对于 OpenSSL 的应用程序来说都被忽略了，但是有些 CA 是要求这些属性的。

证书请求属性字段的格式跟特征名称字段的格式相同，也会受到 prompt 的值的影响，使用方法可以参考特征名称字段的格式介绍。

**(4)** 证书请求扩展字段

证书请求扩展字段由 req 字段中的 req_extensions 变量指定，它定义了要加入到证书请求中的扩展项。该字段的格式跟证书扩展字段的格式基本是一致的，目前主要用来定义 X.509 v3 的扩展项。关于 X.509 v3 扩展项的配置文件格式将会在本书后续的章节中作详细的介绍。

### 6.1.4　配置文件中的证书签发配置

**(1) CA** 主配置字段

配置文件中 CA 主配置字段设置了 OpenSSL 的 CA 指令默认的选项值，对 CA 指令的行为有很大的影响。不过，CA 主配置字段并不是 OpenSSL 配置文件中名为“ca”的字段，ca 字段只是通过 default_ca 变量给 OpenSSL 应用程序提供了 CA 主配置字段的入口，即告诉 CA 指令配置文件中哪个字段是用作 CA 主配置字段，比如在默认的 openssl.cnf 文件中，CA 主配置字段就是 CA_default 字段。当然，应用程序（如 CA 指令）也可以通过自己的选项-name 来选择 CA 主配置字段，这在后面的指令应用章节中将作更详细的介绍。

CA 主配置字段的信息主要包括 CA 指令配置文件、CA 签发证书的限制和策略，以及指定 CA 扩展项字段。一个相对比较完整的 CA 主配置字段看起来如下。

```
[CA_default]
dir = ./demoCA
certs = $dir/certs
crl_dir = $dir/crl
database = $dir/index.txt
new_certs_dir = $dir/newcerts
certificate = $dir/cacert.pem
serial = $dir/serial
crl = $dir/crl.pem
private_key = $dir/private/cakey.pem
RANDFILE = $dir/private/.rand
x509_extensions = usr_cert
name_opt = ca_default
cert_opt = ca_default
copy_extensions = copy
crl_extensions = crl_ext
default_days = 365
default_crl_days = 30
default_md = md5
preserve = no
policy = policy_anything
```

一个实际的 CA 主配置字段，不一定要出现所有这些变量。这些变量的作用跟同名应

用指令的选项的作用经常是一样的，但是指令选项的优先级一般来说比配置文件的要高。表6-2是目前支持的 CA 主配置字段参数。

表 6-2 OpenSSL 配置文件中的 CA 主配置字段参数

| 参数名 | 参数值域 | 参数作用描述 |
|---|---|---|
| oid_file | 文件名 | 定义扩展对象的文件名 |
| oid_section | 字段名 | 定义扩展对象的字段名 |
| new_certs_dir | 目录名 | 存放 CA 指令签发生成新证书的目录，该参数是必须提供的 |
| certificate | 文件名 | 存放 CA 证书的文件，该参数是必须提供的 |
| private_key | 文件名 | 存放 CA 私钥的文件，该参数是必须提供的 |
| RANDFILE | 文件名 | 用于读写随机数种子的文件，也可以是一个 EGD socket |
| default_days | 正整数 | 签发证书的有效期，以天为单位 |
| default_startdate | 时间格式 | 证书生效的日期，格式为 YYMMDDHHNNSSZ，如果该选项没有设置，则使用当前时间作为生效时间 |
| default_enddate | 时间格式 | 证书失效的日期，格式为 YYMMDDHHNNSSZ，该选项和 default_days 选项至少应该有一个 |
| default_crl_hours | 正整数 | 从当前 CRL 到下次 CRL 发布以小时为单位的时间间隔，在生成 CRL 的时候，该选项和 default_crl_days 参数必须有一个 |
| default_crl_days | 正整数 | 从当前 CRL 到下次 CRL 发布以天为单位的时间间隔，在生成 CRL 的时候，该选项和 default_crl_hours 参数必须有一个 |
| default_md | md5、md2、mdc2 和 sha1 | 签发证书时采用的信息摘要算法，选定的信息摘要算法也用于 CRL 的签发 |
| database | 文件名 | OpenSSL 定义的用于已签发证书的文本数据库文件，这个文件通常初始化的时候是空文件，但是却是必须提供的参数 |
| serialfile | 文件名 | 证书签发时使用的序列号参考文件，该文件的序列号是以十六进制的格式存放的，该文件必须提供并且应该包含一个有效的序列号 |
| x509_extensions | 字段名 | 指定 X.509 扩展项定义字段，如果 OpenSSL 配置文件中没有该字段，那么首先会到默认字段查找扩展项，如果没有提供任何扩展项，那么 OpenSSL 的 ca 指令会生成一个 X.509 v1 证书而不是 X.509 v3 证书 |
| crl_extensions | 字段名 | 生成 CRL 时加入的扩展项定义字段，如果没有提供 CRL 扩展项，那么生成的 CRL 就是 v1 版本的 |
| preserve | no 或 yes | 通常情况下，证书签发的特种名称（DN）域的各个参数的顺序是跟证书匹配策略的参数顺序一致的，但是，如果 preserve 参数值为 yes，那么各个参数的顺序就保持跟证书请求的一致 |
| email_in_dn | no 或 yes | 如果该值设置为 no，那么证书请求中的 E-mail 信息会被从证书 DN 域删除，但是可以作为扩展项存在。如果没有设置该参数，那么默认是将 E-mail 信息放置在 DN 域中的 |
| msie_hack | no 或 yes | 这是为了兼容老版本的微软 IE 而设置的参数，现在基本废弃不用了 |
| policy | 字段名 | 指定用于证书请求 DN 域匹配策略的字段，该字段的策略决定 CA 要求和处理证书请求提供的 DN 域各个参数值的规则 |

续表

| 参数名 | 参数值域 | 参数作用描述 |
| --- | --- | --- |
| nameopt | 证书 DN 域显示方式选项，其可选值跟 x509 指令 -nameopt 选项是相同的 | 指定当需要用户确认签发证书的时候显示的可读证书 DN 域显示方式，详细的请参考后面介绍 x509 指令的章节 |
| certopt | 证书域显示选项，其可选值跟 x509 指令-certopt 选项是相同的，但是 no_signame 和 no_sigdump 总是被默认设置 | 指定当需用用户确认签发证书的时候显示的证书域的方式，详细请参考后面介绍 x509 指令的章节 |
| copy_extensions | none，copy 或 copyall | 该参数决定了证书请求中的扩展项信息是否加入到证书扩展项中，如果该项没有定义或者设置为 none，那么所有证书请求的扩展项都会被忽略；如果该项值为 copy，那么如果证书扩展项中没有的项目就会被复制到证书中；如果值为 copyall，那么证书请求扩展项中所有选项都会被复制到证书扩展项中，如果在证书扩展项中已经存在该项，那么证书扩展项的项目会首先被删除，然后再从证书请求中复制过来 |

表 6-2 中，oid_file 和 oid_section 参数一般都不会出现在 CA 主配置字段中，也就是说，一般都使用配置文件默认字段中的 oid_file 和 oid_section 的值。

default_startdate 和 default_enddate 的格式是 YYMMDDHHMMSSZ，是 UTCTime 结构，表示的是标准的格林威治时间。各个位的意义如表 6-3 所示。比如 2003 年 8 月 6 日上午 9 时 30 分 44 秒，则表示为：030806093044Z。

**表 6-3　UTCTime 时间格式**

| 形式参数 | 代表意义 | 例　　子 |
| --- | --- | --- |
| YY | 两位数的年份 | 比如 1999 年，设置为 99 |
| MM | 月份 | 比如 8 月，设置为 08 |
| DD | 日期 | 比如 23 号，设置为 23 |
| HH | 小时 | 比如 13 点，设置为 13 |
| NN | 分钟 | 比如 50 分，设置为 50 |
| SS | 秒 | 比如 24 秒，设置为 24 |
| Z | 字符结束符号 | |

**(2) 请求信息匹配策略字段**

证书请求信息的匹配策略字段由 CA 主配置字段的 policy 参数指定。该字段包含了一组跟证书特征名称（DN）域相关的变量。变量的名称是 DN 域对象的名称，变量值可能为 optoinal，supplied 或 match。下面是一个匹配策略字段的实例：

```
[policy_match]
countryName = match
stateOrProvinceName = match
```

```
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional
```

如果在匹配字段中一个变量的值是 match，那么，该变量在证书请求中的值必须跟 CA 证书相应的变量值完全相同，否则就会被拒绝签发，比如上述的 conutryName 的匹配策略是 match，那么如果 CA 证书的 countryName（C）是“CN”，那么证书请求中的 countryName 就也必须要求是“CN”。如果在匹配策略字段中一个变量的值是 supplied，那么证书请求中必须提供该变量值，否则也会被拒绝签发，但是变量值可以不同。如果在匹配策略字段中一个变量的值是 optional，那么证书请求中可以提供该变量，也可以没有该变量。

一般来说，如果证书请求 DN 域中出现了证书请求匹配策略中没有的变量，签发证书的时候会被自动删除。但是，如果使用了 preserve 选项并设置其值为 yes，或者在 CA 指令中使用了-preserveDN 选项，则匹配策略中没有的变量也会被保留下来。

**(3)** 扩展项字段

CA 主字段中包含了两个指向扩展项字段的变量：x509_extensions 和 crl_extensions，它们都指向一个包括了 X509 v3 证书扩展项信息的字段，从格式和本质上来说，它们都是统一的，都是为了添加 X509 v3 证书扩展项功能的。与此相同的还有证书请求配置中的 req_extensions变量指向的证书请求扩展字段。本书将在下一节详细介绍这种扩展项字段的格式和支持的变量。

### 6.1.5 配置文件中 X.509 v3 证书扩展项

#### 1. 扩展项字段应用概述

X.509 v3 证书已经得到了广泛应用，其最大的特色就是增加了扩展项的支持。在 OpenSSL 中，除了可以在指令中使用参数来指定扩展项的配置，更通常使用的是在配置文件中配置扩展项的各个参数。

X.509 v3 扩展项不仅仅可以添加到证书中，也可以添加到证书请求和证书吊销列表（CRL）中。在 OpenSSL 配置文件中，针对不同的应用，有不同的 X.509 v3 扩展项字段的入口。如果要在使用 ca 指令的时候添加证书扩展项字段，那么应该使用 CA 主配置字段的 x509_extensions 变量；如果要在证书请求中添加证书扩展项字段，那么就应该使用证书请求配置主字段的 req_extensions 变量；如果要在 CRL 中添加证书扩展项，那么就应该使用 CA 主配置字段的 crl_extensions 变量。此外，OpenSSL 的 x509 指令使用的证书扩展项字段是以配置文件默认字段的 extensions 变量为入口的，而证书请求配置主字段的 x509_extensions 变量则为使用 req 指令签发自签名证书提供了扩展项字段的入口。

#### 2. 扩展项字段变量格式

扩展项字段的变量格式是相同的，都具备如下的基本格式：

```
extension_name = [critical,] extension_options
```

上述的 critical 是可选的，一旦变量标记为 critical，即标记该选项为关键选项，那么

要求就会非常严格。理论上，任何客户端一旦不能理解标记了 critical 的扩展项就应该将该证书视为无效，所以在使用 critical 的时候要非常谨慎。有些不规范的软件甚至会将任何包含了 critical 标记的证书都视为无效。

扩展项有三种基本的类型：字符串扩展项、多值扩展项和原始扩展项。

字符串扩展项很简单，它的值域仅仅包含一个字符串作为值或者作为行为方式的描述，比如：

```
nsComment = "This is a Comment"
```

多值扩展项有两种方式：短型和长型。短型的多值扩展项值域是一系列名字和对应值的列表，中间用逗号隔开，而名字和值中间使用“:”隔开，比如：

```
basicConstraints = critical,CA:true,pathlen:1
```

长型的多值扩展项允许将真正的值域放置在单独的字段中，比如上述的变量，也可以写成如下的形式：

```
basicConstraints = critical,@bs_section
[bs_section]
CA=true
pathlen=1
```

短型的多值扩展项和长型的多值扩展项是完全等价的，使用任何一种都可以。但是，有时候值域里面一个变量会出现两次，这种情况下对长型的多值扩展项就需要做一些处理。比如下面的多值扩展项变量：

```
subjectAltName = email:dragonking@here,email:dragonking@there
```

读者可以看到，值域中 email 变量出现了两次，如果用长型的多值扩展项方式表示，就不能简单把它们列举成：

```
subjectAltName = @alt_section
[alt_section]
email = dragonking@here
email = dragonking@there
```

如果以上述的方式表示，那么 alt_section 字段的第一个 email 变量就会被 OpenSSL 的配置文件读取函数忽略，因为根据 OpenSSL 的规则，同一字段出现同名变量，那么就只有最后一个有效。所以为了避免这种请求，通常我们采取下面的形式：

```
subjectAltName = @alt_section
[alt_section]
email.0 = dragonking@here
email.1 = dragonking@there
```

这样，我们就能保证同名变量的所有信息都能正确加入到证书相应的域中。

原始扩展项顾名思义，它采用了原始的数据（如对象的 OID）而不是可读数据来表示变量名或变量值，该方式也支持多个值，中间可以用“,”分开。这种方式的使用应该要非常谨慎，否则就可能导致无效的变量对象声明。比如：

```
certificatePolicies = 1.2.4.5,1.1.3.4
```

甚至还可以在变量域中使用 DER 编码的特定对象 OID，这需要通过 DER 标识来声

明，比如下面的例子：

```
basicConstraints = critical,DER:00:01:02:03
```

DER 后面的数据是经过 DER 编码并用十六进制表示的对象 OID，相信没有多少人愿意使用这种难懂的方法，除了不得已而为之。

### 3. 扩展项字段变量种类

X.509 v3 证书的扩展项的主要目的是为了能够对证书和其相关私钥的用途和使用方法做出限制，从而提高证书的可管理性。下面将以分类的方式来介绍目前支持的扩展项变量，这些变量名称如果是“ns××××”的形式，则一般是 Netscape 定义的扩展项，而不是这个形式的则一般是 PKIX 定义的扩展项。表 6-4 列出了目前 OpenSSL 配置文件支持的扩展项。事实上，这里列出的扩展项只是 OpenSSL 支持的在配置文件中设置的选项，并不是 OpenSSL 支持的扩展项的全部，有些选项 OpenSSL 是支持显示的，但是并不能在配置文件中设置，比如私钥使用周期、CRL 序列号和 CRL 说明等。

**表 6-4 OpenSSL 支持的扩展项**

| 参数名称 | 参数值域 | 定义机构 | 参数说明 |
|---|---|---|---|
| nsBaseUrl | URL 字符串 | Netscape | Netscape 基本 URL |
| nsRevocationUrl | URL 字符串 | Netscape | Netscape 吊销 URL |
| nsCaRevocationUrl | URL 字符串 | Netscape | Netscape CA 吊销 URL |
| nsRenewalUrl | URL 字符串 | Netscape | Netscape 更新 URL |
| nsCaPolicyUrl | URL 字符串 | Netscape | Netscape CA 策略 URL |
| nsSslServerName | 字符串 | Netscape | Netscape SSL 服务器名称 |
| nsComment | 字符串 | Netscape | 证书注释说明 |
| nsCertType | client | Netscape | 定义证书的用途，其值可以为值域中的参数一个或者多个，各个值的意义详见表 6-5 |
| | server | | |
| | email | | |
| | objsign | | |
| | reserved | | |
| | sslCA | | |
| | emailCA | | |
| | objCA | | |
| keyUsage | DigitalSignature<br>nonRepudiation<br>keyEncipherment<br>dataEncipherment<br>keyAgreement<br>keyCertSign<br>cRLSign<br>encipherOnly<br>decipherOnly | PKIX | 定义证书中的密钥的用途，其值可以为值域中参数的一个或多个，参数的意义详见表 6-6 |
| basicConstraints | CA：TRUE 或 CA：FALSE<br>pathlen：n | PKIX | 定义证书基本限制扩展项，使得可以正确识别 CA 证书并对其路径长度作限制，参数域中的“n”为非负整数，表示证书路径长度 |

续表

| 参数名称 | 参数值域 | 定义机构 | 参数说明 |
|---|---|---|---|
| extendedKeyUsage | 用户自己定义的有特定意义的对象简称或对象数字形式 OID | PKIX<br>Netscape<br>Microsoft | 用户自定义的密钥应用信息，目前 PKIX、Microsoft 和 Netscape 都定义了一些，详见表 6-7 |
| subjectKeyIdentifier | “hash”或者用户使用十六进制字符串定义的主体标识 | PKIX | 主体密钥标识，如果值为 hash，则采用 PKIX 规定的方法自动生成主题标识 |
| authorityKeyIdentifier | keyid［:always］<br>issure［:always］ | PKIX | 验证机构密钥标识，一般包括签发 keyID、签发机构和 CA 证书序列号 |
| subjectAltName | email，URI，DNS<br>IP，RID，email：copy | PKIX | 主体别名，可以以更灵活的方式命名证书 |
| issureAltName | email，URI，DNS<br>IP，RID，issure：copy | PKIX | 颁发者主体别名 |
| authorityInfoAccess | accessOID：location | PKIX | location 为 subjectAltName 域中除了 email：copy 之外的所有值之一，accessOID 理论上可以为所有有效对象标识 |
| crlDistribution-Points | URL：location | PKIX | 验证证书吊销状态的 CRL 发布站点，其中 location 为网址 |
| certificatePolicies | 对象标识 OID | PKIX | 证书策略扩展，设置证书策略扩展标识符和限定符列表 |

**（1）描述字符串型的扩展项**

这种类型的扩展项变量直接使用一个字符串给变量赋值，这样的扩展项变量包括：nsBaseUrl，nsRevocationUrl，nsCaRevocationUrl，nsRenewalUrl，nsCaPolicyUrl，nsSslServerName和 nsComment。这些扩展项从名字可以知道，基本上是给出了一些跟证书相关的 URL 的信息，这些扩展项都是 Netscape 定义的。不过，目前很多证书实际上并没有使用这些选项。下面是一个这种扩展项在配置文件中的例子：

```
nsComment = “DragonKing Generated Certificate”
```

**（2）证书类型（nsCertType）扩展项**

nsCertType 是 Netscape 定义的指定证书用途的扩展项，它可选的值包括：client，server，email，objsign，reserved，sslCA，emailCA 和 objCA。nsCertType 值域可以为这些值的一个或者多个。这些值所代表的意义如表 6-5 所示。

**表 6-5　nsCertType 定义值**

| 参数值 | 说　明 |
|---|---|
| client | 用于客户端的证书 |
| server | 用于服务器的证书 |
| email | 用户 E-mail 安全协议的证书 |
| objsign | 用于签名的证书 |
| sslCA | 用于签发 SSL 用户证书的 CA 证书 |
| emailCA | 用于签发 E-mail 用户证书的 CA 证书 |
| objCA | 用于签发用户签名证书的 CA 证书 |
| reserved | 保留值 |

例如，如果要签发一个只用来签名的用户证书，那么 nsCertType 设置应该如下：

```
nsCertType = objsign
```

如果要签发的是一个普通用户证书，并且是用于服务器端的，可以将 nsCertType 的值设置如下：

```
nsCertType = objsign,email,server
```

如果你要使用的是一个 CA 证书，并且该 CA 证书可以用来签发任何用途的证书，那么你的定义应该如下：

```
nsCertType = objCA,emailCA,sslCA
```

**（3）密钥用途（keyUsage）扩展项**

keyUsage 是 PKIX 定义的用于限制证书中密钥用途的扩展项，事实上，它跟 nsCertType 的功能基本上是相同的，都是为了限制证书用途。keyUsage 可选值包括：digitalSignature，nonRepudiation，keyEncipherment，dataEncipherment，keyAgreement，keyCertSign，cRLSign，encipherOnly 和 decipherOnly。keyUsage 的值可以为上述值的一个或者多个，值域中各个参数的意义详见表 6-6。

**表 6-6 keyUsage 定义值**

| 参数值 | 说　明 |
|---|---|
| digitalSignature | 用于数字签名的密钥 |
| nonRepudiation | 用于提供不可否认服务的密钥，一般是第三方公正机构使用的密钥 |
| keyEncipherment | 用于对其他密钥或相关安全信息进行加密，可以用于密钥安全传输 |
| dataEncipherment | 用于对用户数据进行加密的密钥 |
| keyAgreement | 用于商定或建立进一步操作需要的密钥的密钥 |
| keyCertSign | 用于验证证书签名的密钥，也就是 CA 的公钥，这只有在 CA 证书中才有效 |
| cRLSign | 用于验证 CA 对 CRL 签名的密钥 |
| encipherOnly | 密钥仅用于加密 |
| decipherOnly | 密钥仅用于解密 |

例如，如果要指定证书中的密钥只能用来进行加密操作，并且标记该扩展项为关键扩展项，则设置应该如下：

```
keyUsage = critical,keyEncipherment,encipherOnly
```

**（4）基本限制（basicConstraints）扩展项**

basicConstraints 是 PKIX 定义的证书扩展项，在 OpenSSL 配置文件中，它是一个多值型扩展项，它的值域包括两部分：CA 参数和 pathlen 参数。

CA 参数的值为 TURE 或者 FALSE，如果 CA 参数值为 TRUE，指明该证书是一个 CA 证书，即可以用来签发别的证书的证书；如果 CA 的值为 FALSE，那么表示证书是一个最终用户证书，不能作为 CA 证书。下面的配置表明是一个 CA 证书：

```
basicConstraints = CA:TRUE
```

pathlen 参数值是整数，只对 CA 证书有效，也就是说，只有 CA 值为 TRUE 的时候才有效。它指明了从该 CA 开始下面还可以出现多少级 CA 证书。所以，如果你设定一个

CA 证书的 pathlen 为 0，设置如下：

```
basicConstraints = CA:TRUE,pathlen = 0
```

那么表明该 CA 证书只能签发最终用户证书而不能签发更低级别的 CA 证书。

PKIX 规定该扩展项必须标记为关键选项，当然，你理论上也可以不这样做。有效的 CA 证书一般应该标记为 CA 值为 TRUE，而最终用户证书 CA 值一定不能设置为 TRUE，否则就很容易导致混乱。下面的例子是一个标记该扩展项为关键扩展项的最终用户配置例子：

```
basicConstraints = critical,CA:FALSE
```

**（5）扩展密钥（extendedKeyUsage）扩展项**

extendedKeyUsage 允许用户添加额外的密钥应用信息，用户可将自己定义的对象和相应的信息添加到 extendedKeyUsage 列表中。extenedKeyUsage 的值域是多个代表特定意义的数据对象的列表，可以为对象的简称，也可以为对象的数字形式 OID。

目前 PKIX，Netscape 和 Microsoft 都定义了自己的扩展密钥扩展对象，在表 6-7 中列出了 OpenSSL 目前支持的扩展对象。下面是两个使用 extendedKeyUsage 扩展项的例子：

```
extendedKeyUsage = critical,codeSigning,1.2.3.4
extendedKeyUsage = nsSGC,msSGC
```

**表 6-7　extendedKeyUsage 参数值**

| 参数值 | 定义机构 | 参数说明 |
|---|---|---|
| serverAuth | PKIX | 用于 SSL/TLS Web 服务器验证 |
| clientAuth | PKIX | 用于 SSL/TLS Web 客户端验证 |
| codeSigning | PKIX | 用于代码签名 |
| emailProtection | PKIX | 用于 S/MIME 协议 E-mail 保护 |
| TimeStamping | PKIX | 用于可信时间戳 |
| msCodeInd | Microsoft | 用于微软特有代码签名（验证码） |
| msCodeCom | Microsoft | 用于微软商业代码签名（验证码） |
| msCTLSign | Microsoft | 用于微软信任列表签名 |
| msSGC | Microsoft | 用于微软服务器登录加密 |
| msEFS | Microsoft | 用于微软加密文件系统 |
| nsSGC | Netscape | 用于 Netsacpe 服务器登录加密 |

**（6）主体密钥标识（subjectKeyIdentifier）扩展项**

主体密钥标识用于在证书主体拥有多个密钥集的时候指定密钥属于哪个密钥集。subjectKeyIdentifier 参数值目前有两种，一种是以十六进制字符串的方式直接给定主体密钥标识，这种方式现在基本上不使用了；另外一种是给定值 hash，这样，就会根据 PKIX 的规定生成合适的主体密钥标识，如下面的例子：

```
subjectKeyIdentifier = hash
```

**（7）验证机构密钥标识（authorityKeyIdentifier）扩展项**

验证机构密钥标识用于构造证书链的时候标识签发机构的证书和密钥，证书中的验证

机构密钥标识包括三个部分：密钥 ID（keyID）、验证机构 DN 和 CA 证书序列号。KeyID 在验证机构使用多对密钥的时候能起到区别的作用。

在 OpenSSL 中，authorityKeyIdentifier 扩展项的参数值域有两个：keyid 和 issure，两个参数都可选的取值为“always”。下面是一些应用形式：

```
authorityKeyIdentifier = keyid,issure:always
authorityKeyIdentifier = keyid:always
```

如果给定 keyid，那么 CA 签发证书的时候就会复制 CA 证书的主体密钥标识到新签发的证书中，如果 keyid 取值 always，则复制失败的时候就拒绝签发证书。

给定 issure 将告诉 CA 签发证书的时候复制 CA 证书的 DN 和序列号到新签发证书中，一般来说，虽然给定了 issure 值，但是只有 keyid 出显但复制主体密钥失败或者 keyid 没有给定的情况下才会执行 CA 证书的 DN 和序列号的复制操作。如果 issure 给定了 always 值，则不论在什么请求下都执行其定义操作。

**（8）主体别名（subjectAltName）扩展项**

主体别名为证书提供了形式更加灵活的命名方式，理论上可以包含 IP 地址、URL、电子邮件及域名等信息作为主体别名。目前，就 OpenSSL 来说，在配置文件中支持的值包括：email（E-mail 地址），URL（全球资源定位地址），DNS（DNS 域名），RID（已注册对象标识）和 IP（IP 地址）。下面是一些 OpenSSL 配置文件中赋值的例子：

```
subjectAltName = email:my@ OpenSSL. cn,URI:http://www. OpenSSL. cn/
subjectAltName = email:my@ OpenSSL. cn,RID:1. 2. 3. 4
```

此外，email 参数还有一个特殊的值“copy”，如果设置了该值，就会自动把证书主体中包含的 E-mail 地址复制到证书扩展项中。其使用方式如下：

```
subjectAltName = email:copy,email:my@OpenSSL. cn,URI:http://www. OpenSSL. cn/
```

**（9）颁发者别名（issuerAltName）扩展项**

颁发者别名扩展项为颁发者的证书提供了不同形式的命名方式，它采用的形式跟上述的主体别名基本一致。OpenSSL 配置文件中的 issureAltName 支持 subjectAltName 的所有参数，但是不支持 email 参数的 copy 值。issureAltName 还支持 issure：copy 选项，如果该选项设置了，那么证书颁发者主体别名中所有别名都会被复制到新签发证书的颁发者别名扩展项中。下面是使用 issure：copy 选项的例子：

```
issuerAltName = issuer:copy
```

**（10）验证机构信息处理（authorityInfoAccess）扩展项**

验证机构信息处理扩展项给出了如何处理跟 CA 相关的信息的详细细节。在OpenSSL 配置文件中，其基本格式如下：

```
accessOID;location
```

location 可以是主体别名中除了 email：copy 参数外的所有形式。accessOID 理论上可以为任何有效的对象标识，但是，目前有具体意义的仅仅是 OCSP 和 caIssures。OCSP 表示在指定的 location 中作为 OCSP 响应服务器；caIssures 则表示 CA 的地址。下面是两个应用例子：

```
authorityInfoAccess = OCSP;URL:http://www. OpenSSL. cn/
authorityInfoAccess = caIssuers;URL:http://www. OpenSSL. cn/ca. html
```

**（11）CRL 分布点（crlDistributionPoints）扩展项**

CRL 分布点扩展项用于指明用户为了验证证书吊销状态而需要查找 CRL 信息的发布站点。目前来说，OpenSSL 的配置文件仅支持 CRL 分布点中的 URL 参数，而对于 PKIX 中规定的 cRLIssuer 和撤销原因暂时还不支持。crlDistributionPoints 可以有多个 URL 参数，是一个多值型的扩展项。下面是 crlDistributionPoints 扩展项在 OpenSSL 配置文件中使用的两个例子：

```
crlDistributionPoints = URL:http://www.OpenSSL.cn/ca.crl
crlDistributionPoints = URL:http://OpenSSL.cn/my.crl,URL:http://OpenSSL.cn/ca.crl
```

**（12）证书策略（certificatePolicies）扩展项**

证书策略扩展项定义了证书应用的策略，比如用于证书验证的策略。证书策略扩展项目前使用并不多，支持的软件也很少。在 OpenSSL 配置文件中，certificatePolicies 是一种原始型的扩展项，其值域一般来说是原始的对象数字标识的列表，中间可以用逗号分开。下面是一个简单的例子：

```
certificatePolicies = 1.2.4.5,1.1.3.4
```

证书扩展策略除了支持对象标识符，还支持限定符，但是，在 OpenSSL 中，如果要使用限定符，那么就不能直接在 certificatePolicies 的值域中添加，而要使用单独的字段，看起来如下面的形式：

```
certificatePolicies = 1.2.4.5,1.1.3.4,@policy
```

目前来说，OpenSSL 配置文件支持的证书扩展策略限定符有 CPS（验证操作规范）和 userNotice。CPS 给定了验证操作规范的 URL 地址，而 userNotice 则包含了通知用户的一些扩展信息。

一个包含了证书扩展策略限定符的字段必须使用 policyIdentifier 参数指明对象标识（OID）。通常来说，CPS 参数使用下面的表达形式：

```
CPS.nnn = value
```

其中，nnn 为非负整数，这样可以允许加入多个 CPS 地址。

userNotice 限定符本身又包含了 explicitText，organization 和 noticeNumbers 三个限定符选项，所以也要使用单独的 userNotice 限定符字段来给赋值，通常一个 userNotice 的值这样表示：

```
userNotice = @section
```

其中 section 为 userNotice 的限定符字段。explicitText 和 organization 的值都是文本形式的，而 noticeNumbers 的值则是一系列数字的列表。需要注意的是，organization 和 noticeNumbers 两个值在 OpenSSL 配置文件中是必须同时出现的，不能单独出现其中一个而没有给出另一个。下面是一个相对完整的证书策略扩展字段的例子：

```
certificatePolicies = ia5org,1.2.3.4,1.5.6.7.8,@polsect
[polsect]
policyIdentifier = 1.3.5.8
CPS.1 = "http://www.OpenSSL.cn/"
CPS.2 = "http://OpenSSL.cn/"
```

```
userNotice.1 = @notice
[notice]
explicitText = "Welcome to visit www.OpenSSL.cn"
organization = "www.OpenSSL.cn"
noticeNumbers = 1,2,3,4
```

在微软的产品中，要正确显示证书策略扩展项，必须在 certificatePolicies 的值域一开始就声明“ia5Org”选项，如上例所示。

## 6.2 文件编码格式

OpenSSL 中虽然使用 PEM 作为基本的文件编码格式，但是，由于不同的对象其封装的标准格式不太一样，所以经常会导致读者产生迷惑。此外，OpenSSL 也支持 DER 编码和过时的 Netscape 编码格式（NET）。

### 6.2.1 数据编码格式

OpenSSL 的编码是基于 ASN.1 标准的，ASN.1 全称为 Abstract Syntax Notation One，是一种描述数字对象的方法和标准。ASN.1 是一种结构化的数字对象描述语言，它包括了两部分，分别为数据描述语言（ISO 8824）和数据编码规则（ISO 8825）。ASN.1 的数据描述语言标准允许用户自定义基本数据类型，并可以通过简单的数据类型组成更复杂的数据类型。比如一个复杂的数据对象，如 X.509 证书，就是在其他一些数据类型上定义的，而其他数据类型又是在更基本的数据类型上建立的，直到回溯到定义的最基本的数据类型。

ASN.1 提供了多种数据编码方法，包括了 BER，DER，PER 和 XER 等。这些编码方法规定了将数字对象转换成应用程序能够处理、保存和网络传输的二进制编码形式的一组规则。目前经常被采用的是 BER（Basic Encode Rules）编码，但是 BER 编码具有编码不唯一的性质，也就是说，一个相同的对象通过 BER 编码可能会产生几种不同的编码数据。所以，在 OpenSSL 和其他密码学相关软件中经常使用 BER 的一个子集 DER（Distinguished Encoding Rules）。对于每一个 ASN.1 对象，使用 DER 编码得出的二进制编码数据是唯一的。

PEM 编码全称是 Privacy Enhanced Mail，是一种保密邮件的编码标准。通常来说，对信息的编码过程基本如下。

① 信息转换为 ASCII 码或其他编码方式，比如采用 DER 编码。

② 使用对称加密算法加密经过编码的信息。

③ 使用 BASE64 对加密后的信息进行编码。

④ 使用一些头定义对信息进行封装，主要包含了进行正确解码需要的信息，头定义的格式形式如下：

```
Proc-Type,4:ENCRYPTED
DEK-Info:cipher-name,ivec
```

其中，第一个头信息标注了该文件是否进行了加密，该头信息可能的值包括

ENCRYPTED（信息已经加密和签名），MIC-ONLY（信息经过数字签名但没有加密），MIC-CLEAR（信息经过数字签名但是没有加密，也没有进行编码，可使用非 PEM 格式阅读），以及 CLEAR；第二个头信息标注了加密的算法及对称加密块算法使用的初始向量。

⑤ 在这些信息的前面加上如下形式头标注信息：

—BEGIN PRIVACY-ENHANCED MESSAGE—

在这些信息的后面加上如下形式尾标注信息：

—END PRIVACY-ENHANCED MESSAGE—

OpenSSL 的 PEM 编码基本上是基于 DER 编码之上的，也就是说，它在上述第一步采用的是 DER 编码，所以，从本质上来说，OpenSSL 的 PEM 编码就是在 DER 编码基础上进行 BASE64 编码，然后添加一些头尾信息组成的。读者可以做一个实验，将 OpenSSL签发的 PEM 证书文件转换成 DER 编码（二进制），导入到 Windows IE 中，然后再从 Windows IE 以 BASE64 编码的 DER 格式导出到另一个文件中，比较原始的 PEM 格式文件跟新导出的 DER 文件，读者会发现其实是一样的。事实上，OpenSSL 也是将以这种方式导出的文件当成是 PEM 格式的文件进行解码的。

需要说明的是，OpenSSL 的指令有对特定类型的文件（比如 X.509 证书文件）进行 PEM 和 DER 格式转换的方法。由于 OpenSSL 里面的 DER 格式是指没有经过 BASE64 编码的 DER 格式，所以事实上就是在二进制的 DER 和 BASE64 编码的 DER（PEM）格式之间进行转换。

OpenSSL 还支持一种老式的 Netscape 编码格式（NET），由于这种格式现在基本上废弃不用了，这里不再作更多的介绍。

### 6.2.2　证书编码

#### 1. X.509 证书

数据编码格式提供了封装数据的基本方法，但是，对于具体的数据对象，比如证书，根据其包含的内容的不同，还有更具体的表达方式。

目前总的来说有三种常用的证书编码格式：X.509 证书、PKCS＃12 证书和 PKCS＃7 证书。X.509 证书是最经常使用的证书，它仅包含了公钥信息而没有私钥信息，是可以公开进行发布的，所以 X.509 证书对象一般都不需要加密。一个 OpenSSL 签发的经过 PEM 编码的 X.509 证书看起来通常有如下的格式：

……（省略可读解释信息）……

—BEGIN CERTIFICATE—

MIIDtzCCAyCgAwIBAgIBAjANBgkqhkiG9w0BAQQFADCBmDELMAkGA1UEBhMCQ04x
EDAOBgNVBAgTB0JlaUppbmcxFzAVBgNVBAoTDnd3dy5PcGVuU1NMLmNuMRYwFAYD
VQQLEw1NYW5hZ2VyIEdyb3VwMSAwHgYDVQQDExdXZWJDQSBvZiB3d3cuT3BlblNT
TC5jbjEkMCIGCSqGSIb3DQEJARYVRHJhZ29uS2luZ0BPcGVuU1NMLmNuMB4XDTAz
MDgwMTA3NDI0M1oXDTEzMDcyOTA3NDI0M1owgYIxCzAJBgNVBAYTAkNOMRAwDgYD
VQQIEwdCZWlKaW5nMRcwFQYDVQQKEw53d3cuT3BlblNTTC5jbjENMAsGA1UECxME
VXNlcjETMBEGA1UEAxMKRHJhZ29uS2luZzEkMCIGCSqGSIb3DQEJARYVRHJhZ29u

S2luZ0BPcGVuU1NMLmNuMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDPe5Ud
/suMvgGS9 + xte6b50rFby9SNw92PoInhFOv6F5gi3Lyh1mKsZNg3sNz11kSVhNhm
RHddVsK7xTfUT + PYgfIRcugh7HDPIurvuSU2x7GdZ0CYUw1EIm0MPsKUKN7P8hIS
yGgh + wmrvyNNkAhZ50iF7LG2nh/YFLsrukxWbQIDAQABo4IBIzCCAR8wCQYDVR0T
BAIwADAsBglghkgBhvhCAQ0EHxYdT3BlblNTTCBHZW5lcmF0ZWQgQ2VydGlmaWNh
dGUwHQYDVR0OBBYEFOI4xUwj1D6GBxo7vQw8vhH3OI3wMIHEBgNVHSMEgbwwgbmA
FNqjb5alIpv2vqGPv/Fyfwzc8xKaoYGdpIGaMIGXMQswCQYDVQQGEwJDTjEQMA4G
A1UECBMHQmVpamluZzEQMA4GA1UEBxMHSGFpZGlhbjEXMBUGA1UEChMOd3d3Lk9w
ZW5TU0wuY24xEDAOBgNVBAsTB01hbmFnZXIxEzARBgNVBAMTCkRyYWdvbktpbmcx
JDAiBgkqhkiG9w0BCQEWFURyYWdvbktpbmdAT3BlblNTTC5jboIBATANBgkqhkiG
9w0BAQQFAAOBgQBCqv4 + 8ERyCbtD2FULIGIOM1xEHeQwvT6nHI + 1Z9AcwWSL9uF7
iIdS7ZdknwPFNEgHjuSltN6MZOmrXboJyVRm8IfzSQ9JB/cnPx4VpQdMWIUJC43q
wBOa5wKWIN9AyJbrKFLOqwFjAhdZN1qh9SFCC56cNWj1m5dYO + YKiQoXeQ = =
—END CERTIFICATE—

上述信息中，在“—BEGIN CERTIFICATE—”和“—END CERTIFICATE—”之间就是 PEM 编码的 X.509 证书。PEM 编码的数据信息总是包含在一对头和尾组成的标识符之间，对于 X.509 证书来说，除了上述形式的头尾格式，还可能出现以下两种不同的标识符：

—BEGIN X.509 CERTIFICATE—

—END X.509 CERTIFICATE—

或者

—BEGIN TRUSTED CERTIFICATE—

—END TRUSTED CERTIFICATE—

以上三种形式，OpenSSL 都可以进行正确的处理和解码。

读者可能还会在 OpenSSL 实际签发的证书文件中最前面发现很多可读的证书明文解释，事实上，这些只是增加了证书文件的可读性，并不代表真正的数据。在其他软件中，比如 Windows 软件，可能并不支持这些额外的明文信息，所以，先要将“—BEGIN CERTIFICATE—”之前的所有可读信息去掉。删除这些信息的方法很多，最简单的是手动删除。此外，也可以使用 x509 指令，进行 PEM 到 PEM 的证书格式转换，就可以去除明文可读信息。

在 Windows 平台，X.509 证书文件的后缀名经常是 der，cer 或者 crt，都是可以被自动识别的。当然，对于 OpenSSL 来说，后缀名是没有实际意义的。

OpenSSL 的指令 x509 提供了对 X.509 证书进行格式转换的方法，可以在 DER，PEM 和 NET 三种格式之间进行转换。

### 2. PKCS#12 证书

PKCS#12 证书不同于 X.509 证书，它可以包含一个或多个证书，并且还可以包含证书对应的私钥。OpenSSL 对 PKCS#12 证书提供了支持，并且提供了专门用于处理 PKCS#12 证书的指令 pkcs12。pkcs12 指令可以将普通的 X.509 证书和私钥封装成 PKCS#12 证书，也可以将 PKCS#12 证书转换成 X.509 证书，并提取出相应的私钥。

一般来说，PKCS＃12 证书的私钥是经过加密的，密钥由用户提供的口令产生。所以，无论在使用 pkcs12 指令编码或者解码 PKCS＃12 证书的时候，一般都会要求用户输入密钥口令。

PKCS＃12 证书文件在 Windows 平台和 Mozzila 中支持的后缀名是 p12 或者 pfx。如果要在 IE 或者 Mozzila 中正确使用自己的证书，那么一般来说都要求转换成包含公钥和私钥的 PKCS＃12 证书导入到相关软件中。

3．PKCS＃7 证书

PKCS＃7 可以封装一个或多个 X.509 证书或者 PKCS＃6 证书（PKCS＃6 是一种证书格式，但是并不经常使用），并且可以包含 CRL 信息。PKCS＃7 证书也不包含私钥信息，但是跟 X.509 证书不同的是，它可以包含多个证书，这就使得 PKCS＃7 可以将验证证书需要的整个证书链上的证书都包含进来，从而方便证书的发布和正确使用。

OpenSSL 提供了对 PKCS＃7 格式文件的支持，并提供了 crl2pkcs7 和 pkcs7 两个指令来生成和处理 PKCS＃7 文件。可以使用他们在 X.509 证书和 PKCS＃7 证书之间进行转换和处理。

PKCS＃7 文件在 Windows 平台的合法后缀名是 p7b。

### 6.2.3　密钥编码

OpenSSL 有多种形式的密钥，很多情况下，需要把这些密钥保存下来。OpenSSL 提供了 PEM 和 DER 两种编码方式对这些要保存的密钥进行编码，而且都提供了相关指令可以使用户在这两种格式之间进行转换。

DER 编码存储的密钥文件是不可读的，如果你乐意用文本编辑器打开它，那么将看到一些难以理解的符号，然而这很正常，因为这是一个二进制编码的文件。PEM 则不一样，它要友好得多，因为 PEM 经过 BASE64 编码。用文本编辑器打开 PEM 编码的密钥文件，可以看到跟证书类似，它们真正的编码都包含在类似于：

```
—BEGIN XXXXX—
—END XXXXX—
```

这样一对符号内，使用某些 OpenSSL 指令的一些选项后，你也许会在这对符号之前看到一些明文的信息，这些信息是编码内容的明文解释，并不是实际的数据，你要关注和使用的就是上述特殊符号内包含的内容。

密钥总的来说有两种，一种是可以公开的（比如公开密钥对的公钥），一种是不能公开的（比如公开密钥对的私钥）。那么，反映在编码上，有的密钥文件就会需要加密，有的就不需要加密。一个经过加密的 PEM 编码密钥文件在上述的符号内会增加一些头信息，这就是在前面介绍的 PEM 编码的头信息，这时候就派上用场了。这些头信息主要是为密钥解密的时候提供有用的信息，包括标记密钥加密状态、使用的加密算法及初始向量（对于块加密算法来说）。下面是一个经过加密的 RSA 私钥的 PEM 编码文件的例子：

```
—BEGIN RSA PRIVATE KEY—
Proc-Type:4,ENCRYPTED
DEK-Info:DES-EDE3-CBC,86B0167E005535D2
```

```
tN + eost4VSWpEMASx2sxoi9RCm/ibrbGLsiYLFkZ4APFXsP9Xt5CtISUdGH9FMku
REjJByDI2ry421PNOsutEnxrgS9AZb25U/JNpCb2JMgtWEGFKhj460jRbU4eZqVl
6/Zt1bCYPnp + qRAnrL3OCdEjqohavw + scdVrmek3WQ6ucACl72H340polphMj7Zs
feowZJESkRhbitUEyl6lJts1mmvimjhLDkGraqvAXUag2HfJ4h8i + 8WWoTNhLW9c
N/5t3f + 9mmBXpAAb9Tb6qypl7p7NcEZf91AWnlOODfqVwzfCl + k0MWNPG2hF4wxH
gDJpWBiqnpJqGbanCZMYXAECf + VukLvR0/7nOUEd6VBo0DFeOCrIU2h1e6 + GebsY
YGHZjzrHjSPbFJo6bl8n7SojDadmugEA + YLXj9zILNlGjtKfzb4x0wqxq9dJXO3s
OmOYyOlFdcAb11CE9LEoNjI5bWP/fq8FRXYn7OcvvkZFfeIQFQ4s1UJbeLm + rSU3
7SLt + I0yenO7kh3kV10H8stIh874Dps4vZBDZRBYgffcK2tC6vd2Fpb9Wfm + 0Y3k
tvxbJCKoSxObIBlFxHGGlPpnNoIUhEKmOPKBCE6VAE3h3Fh9B4sVe20ABS58qEsa
w6KrDjyNAGA6hecq6x8IFROurNYYjQx3gvTVUocqHGetZcu7d4MDl7UgDwfXaRTD
sNJ58PEDiu3BU + fnB1p3B2lV7lQPwECexjQ0IGLDSa5/XOTOFj043KGthZ9Mh9y4
AwXLxgzdwSROIm4JPyjSt + U1/ye6fDQKYHZwDxJq/6YEGu0o7avxNA = =
—END RSA PRIVATE KEY—
```

可以看到，上述的 PEM 编码的是 RSA 私钥（RSA PRIVATE KEY），该密钥经过了加密（ENCRYPTED），使用的加密算法是 3DES 的 CBC 方式（DES-EDE3-CBC），使用的 IV 向量是“86B0167E005535D2”。

OpenSSL 的指令提供了对密钥加密的功能，并提供了多种可选的加密算法，这些算法都是对称加密算法，比如 DES 和 DES3 等。OpenSSL 的指令通常使用要求输入口令的方式来生成用来加密密钥数据的密钥，这里的口令并非直接用来作为加密的密钥，而是根据这个口令使用一系列 HASH 操作来生成一个用于加密密钥数据的密钥。在读取这类加密的密钥文件的时候，OpenSSL 的指令同样会要求你输入用于解密的口令，当然，这个口令跟你加密密钥时输入的口令应该是一致的。

OpenSSL 对某些类型的密钥也提供了一些标准的封装格式，如 PKCS＃8 和 PKCS＃12 格式。PKCS＃8 是用于封装需要保密的密钥的规范格式，OpenSSL 提供了 pkcs8 的指令来处理这类格式的文件。事实上，PKCS＃12 对密钥的封装就是采用了 PKCS＃8 的格式。

OpenSSL 有各种各样的密钥类型需要采用上述方法进行编码，比如 RSA 的公钥和私钥、DSA 的公钥和私钥、DH 密钥参数，等等，它们的形式基本上是一样的。当然，无论采用什么加密方式对密钥进行加密保存，以文件形式存储的密钥其安全性总是不可靠的，更值得信赖和推荐的方式是将密钥存储在不可导出的密钥设备中，比如 USB Key、Smart Card 和加密机等硬件设备中，这些设备内部能够完成加密操作，所以不需要将密钥取出，这增加了密钥的安全性。

### 6.2.4 其他编码

证书吊销列表（CRL）是用户验证证书的重要参考资料，它主要包含了无效的证书列表，告诉用户哪些证书是已经吊销或无效的。当然，如果没有 CRL，或许你可以选择另一种验证方式，那就是使用在线证书服务协议（OCSP）。但不幸的是，OCSP 并不总是能够使用，比如你的网络有时候可能不能接通 OCSP 服务器，那么你就要自行解决这个问题

了。CRL 是一种相对方便和独立的解决方案，只要获取了 CA 中心提供的在有效期内的 CRL，基本上就能对证书的有效性进行验证。

OpenSSL 提供 CRL 文件的生成、解释及格式转换等。PEM 格式编码的 CRL 一般包含在如下的一对符号内：

—BEGIN X.509 CRL—

—END X.509 CRL—

当然，还可以使用 DER 格式保存 CRL。OpenSSL 的指令提供了在 PEM 和 DER 之间对 CRL 进行格式转换的功能。此外，你也可以把 CRL 和证书一起封装成 PKCS＃7 格式，使用的指令是 crl2pkcs7。

OpenSSL 还包含了其他一些可以使用 PEM 编码的有趣对象，当然，你可能永远不需要使用这些对象。SSL 协议的 Session 对象是其中之一，它可以被编码成 PEM 格式，然后你可以使用 OpenSSL 的指令将它解释成可以查看的明文。Session 对象处理的指令是 sess_id。

OpenSSL 还支持对 Netscape 证书序列格式（NSEQ）的 PEM 编码和解释，相应的指令是 nseq。

## 6.3　文本数据库

### 6.3.1　应用概述

文本数据库是跟 OpenSSL 的 CA 应用程序紧密结合在一起的，它以文本的方式记录 CA 已经签发的证书的状态和摘要信息。这些状态信息可以用于跟证书库相关的一些操作，比如使用 ca 指令生成 CRL 主要就是读取这个文本数据库的信息作为参考。

文本数据库就是普通的文本文件，在使用之前，需要先建立一个空文件作为文本数据库，名字和后缀名都是无所谓的。ca 指令使用的文本数据库文件可以在配置文件中指定，相关参数是 CA 主配置字段的 database 选项。默认的文本数据库文件是 index.txt 文件，在 OpenSSL 根目录的 apps/demoCA 目录下，该文件已经包含了 OpenSSL 提供的演示证书的一些信息。当然，如果你要使用自己的 CA 证书，那么该文件应该首先清空。上述的初始工作，包括文本数据库文件的建立等 CA 目录结构的构造，可以手动完成，也可以使用 OpenSSL 的 Perl 脚本 CA.pl 完成。

OpenSSL 提供了管理和读写文本数据库的接口函数，它们都在 crypto/txt_db 目录下，函数不多，很容易理解和使用，当然，这是基于你阅读下面的一节——即你对文本数据库的结构已经有所了解的基础上。

### 6.3.2　数据结构

OpenSSL 的文本数据库结构非常简单，每一行代表一条完整的记录，每条记录对应一个已经签发的证书。一般来说，每条记录包括六个可选的字段，分别是：证书状态、证书生效时间、证书到期时间、证书序列号、证书存放路径及证书的特征名称（DN）值。其中，证书生效时间是可选项，有些记录是可以不填写的。下面是文本数据库中一条完整

记录：

```
R 031105233205Z 951009233205Z 01 certs/00000001 /CN = DragonKing
```

证书状态字段是文本数据库记录中的关键字段，也是第一个字段，ca 指令产生 CRL 列表就是根据证书中的状态信息作为依据的。状态字段的可选值有三个，分别是 V（Valid），R（Revoked）和 E（Expire），代表有效、吊销和过期三种证书状态。依次可知，上述记录对应的证书状态是已经吊销了（R）。

第二个字段是证书到期时间字段，保存的时间已经换算成格林威治时间。时间格式在前面的章节已经介绍过，这里不再重复。在上述的例子中，证书到期的时间是 03 年 11 月 5 日 23 时 32 分 5 秒。

第三个字段是证书生效时间，格式同证书到期时间的格式。证书生效时间是可选的字段，在文本数据库的记录中不一定要存在，比如你签发证书的时候没有指定生效时间而是使用默认的当前时间作为生效时间，那么在文本数据库的相应记录中就不会存在该项。

第四个字段是证书序列号，以十六进制的格式表示。证书库中的每个证书的序列号都是不同的。

第五个字段是证书存储路径，指明了生成的证书存储的具体位置。该目录选项有时候可能由于 ca 指令某些参数使用的原因导致没有办法填写，这时候就标记为“unknown”，但是，这不会影响证书库的正常使用。如果你在使用 ca 指令的时候已经指定了证书输出文件，那么就会标记路径字段为“unknown”。

第六个字段也是一个重要的字段，就是记录对应的证书的特征名称（DN），也即是证书中的主题名。在 OpenSSL 的文本证书库中，ca 指令要求必须保证签发的证书具有不同的 DN 值，否则就不能成功签发证书，这是因为 OpenSSL 将 DN 值作为文本数据库一个索引。当然，这种限制跟实际应用的情况可能会有所不同，比如，有些情况下同一个人可能要申请多个证书用于不同的目的，E-mail 或者 Web 访问，这就会引起矛盾。

如果证书吊销的时候使用了吊销原因的选项（ca 指令的 crl_reason 选项），那么在被吊销的证书相应的记录中，还可能会添加一到两个字段注明吊销原因，下面就是两条使用了吊销原因的记录：

```
R 130906152018Z 030909155742Z,holdInstruction,holdInstructionCallIssuer
01 unknown/C = CN/ST = BeiJing/O = www. OpenSSL. cn/CN = DragonKing
R 130906160955Z 030909161037Z,superseded 02 unknown
/C = CN/ST = BeiJing/O = www. OpenSSL. cn/CN = WangZhiHai
```

文本数据库的记录生成之后，一般来说，以后可能修改的字段就是证书状态字段，当证书被吊销后，证书状态字段会发生改变。对于过期的证书，文本数据库并不会自动更新，需要使用 ca 指令的-updatedb 选项进行更新，一般来说，在生成 CRL 之前，都应该使用该选项进行更新。

## 6.4 序列号文件

序列号文件是 ca 指令签发证书的时候的依据文件之一，它从该文件读取当前签发的证书的序列号并将序列号文件中的序列号加 1，这样，就可以确保证书的序论号是递增

的，不会重复。序列号文件也是一个文本文件，里面仅仅简单包含了一个十六进制表示的数字。在使用 CA 之前，需要先创建该文件并初始化，工作其实很简单，创建一个空文件，用文本编辑器打开填上一个十六进制的数字，比如你想序列号从零开始，那么就填“00”，然后保存退出即可。当然，如果你的序列号文件跟 CA 程序所使用的配置文件（通常是 openssl.cnf）指定的序列号文件不一样，你还要修改配置文件的相关参数，该参数是 CA 主配置字段的 serial 选项。

事实上，上述的初始化工作虽然可以手动完成，但是也可以依靠脚本 CA.pl 完成。此外，x509 也提供了创建序列号文件的机制。

## 6.5　随机数文件

无论使用 OpenSSL 的指令还是其 API，随机数文件都是会经常碰到的一个概念。大部分密码算法的安全性都跟随机数的好坏相关，所以一个成功的密码学应用软件，对随机数的处理是不能随便的。OpenSSL 虽然没有提供很完美的随机数生成程序，但是也提供了一些相关机制。因为采用的随机数生成 API 基本上都是相同的，为了生成不同的随机数，也就是增强随机数的随机性，就需要采用不同的随机数种子。OpenSSL 的随机数文件基本上就是提供随机数种子的文件。

无论什么文件，只要有一定的内容，都可以作为 OpenSSL 应用指令的随机数文件，因为 OpenSSL 是将之当成二进制文件处理的，所以什么内容并不重要，虽然可能所带来的安全性可能有区别。这里介绍随机数文件的意义在于告诉读者：无论什么文件，都可以作为 OpenSSL 指令要求的随机数文件。

事实上，如果你没有提供随机数文件，OpenSSL 指令也会从屏幕的状态取得随机数种子。默认的随机数文件可以在 OpenSSL 配置文件的 CA 主配置字段的 randfile 选项中指定。如果你在参数中没有输入随机数文件，那么 OpenSSL 指令一般会首先根据该配置文件查找默认的随机数文件，如果不存在，再采用从屏幕状态获取随机数种子的方法。

## 6.6　口令输入方式

虽然口令的安全性很值得担忧，但是口令在 OpenSSL 中是经常使用的，这是没有办法替代的一种简易的保护数据的方法。OpenSSL 中使用口令的地方很多，比如密钥的加密和解密，等等。OpenSSL 的指令提供了多种灵活的口令输入方法，但是，正是因为方法太多了，反而令很多使用者觉得迷惑不解。

在 OpenSSL 的指令中，通常通过“-passin”或者“-passout”的参数来输入口令，这些参数的形式有 5 种，分别对应 5 种不同的口令获取途径。

### 6.6.1　提示输入

最简单的，也就是默认的，是从指令行界面提示输入口令。一般来说，如果你没有使用“-passin”或者“-passout”指明口令获取方式，如果输入文件或指令要求输入口令，那么就会采用这种方式。这种方式的应用很简单，只要根据提示输入口令即可，但是一般

会要求验证一次。这种输入方法的好处是口令不会在屏幕上以明文显示出来，在一定程度上可以防止恶意的旁人的偷窥。当然它也有不方便的地方，比如没有办法在程序使用脚步自动批处理运行的时候使用，因为它需要交互，这样自然限制了其自动性能。虽然这种方式是默认的，但是如果你非要用“-passin”指定这种方式，那么可以输入如下面形式的参数：

```
-passin stdin
```

### 6.6.2 直接输入

直接输入口令的方式可能是最受欢迎的，也是最容易想像的方式，它可以直接在指令参数中输入指令。但是因为 OpenSSL 灵活的口令获取方式，使得使用者通常被这种容易想像方式迷惑。如果你要使用这种方式，应该使用下面的形式：

```
-passin pass:12345678
```

这可能跟你想像的很不一样，你可能只是简单的想到应该如此：

```
-passin 12345678
```

但这肯定会使得 OpenSSL 的指令给你返回错误使用参数的提示。事实上你只需要做多一点点，在口令前面输入“pass:”5 个字符。这种方式因为口令是以明文输入的，在屏幕上可见，所以使用的时候一定先要转过头四周看看，确保没有人想盗取你的口令才能进行。

### 6.6.3 环境变量输入

有时候可能需要将口令保存起来重复利用，而你又不太想记住这个复杂的口令，那么可以选择将它存放在环境变量中，使用的时候只要记住环境变量名就可以。对于测试来说，这是一个好方法，这可以使得你所有需要口令的文件都使用相同的口令，免除你痛苦地回忆口令。这种方式的使用首先要求你设置好环境变量，然后输入如下的形式：

```
-passin env:passwdvar
```

其中的 passwdvar 就是环境变量名。当然，这种方式的安全性不怎么样，因为环境变量是所有使用相同计算机的人都能看到的，并且是以明文存放。

### 6.6.4 文件输入

OpenSSL 还提供了从文件获取口令的方式，这种方式指定获取口令的文件名，然后将文件名的第一行作为口令。如果你同时使用了“-passin”和“-passout”参数并且都指定了同一个文件作为口令文件，那么指令就会将第一行作为-passin 的口令，而第二行作为-passout 的口令。这种方式的使用形式如下：

```
-passin file:filename
```

这里的 filename 是文件名，当然，必要的时候，需要包含路径。事实上，提供的 filename 不一定就是文件，可以是设备或其他符合文件 I/O 操作规范的命名管道。

### 6.6.5 描述符输入

通用的描述符也可以作为一种获取口令的来源，其使用方式如下：

-passin fd:number

这里的 number 是文件描述句柄的编号。这种方式在 Windows 下的指令行界面下使用似乎并不方便，但是在脚步文件中使用还是可以的。而在 Unix 系统下，使用就方便得多，比如管道就可以作为一种文件描述符输入。

## 6.7　本 章 小 结

本章对 OpenSSL 指令可能使用到的一些通用概念作了详细的解释和介绍，为阅读下面章节打下了基础。

首先介绍了 OpenSSL 中具有重要地位的配置文件，包括配置文件的结构、使用规则及各种应用方式，等等。在介绍配置文件的同时，也对 OpenSSL 指令涉及的一些基本概念作了解释。

接下来介绍了 OpenSSL 的各种文件编码方式，并在此同时介绍了 OpenSSL 支持的各种证书、密钥和 CRL 的封装标准。

本章还对 OpenSSL 的文本数据库应用和结构作了解释和介绍，对序列号文件和随机数文件的概念也作了阐释。

最后，本章介绍了 OpenSSL 提供的各种灵活的口令输入和获取方式，排除读者对 OpenSSL 口令输入方式的迷惑。

# 第 7 章

# 对称加密算法指令

## 7.1 对称加密算法指令概述

OpenSSL 的加密算法库提供了丰富的对称加密算法，一般来说，使用 OpenSSL 对称加密算法有两种方式，一种是使用 API 函数的方式，一种是使用 OpenSSL 提供的对称加密算法指令方式。本书将介绍对称加密算法的指令方式。

OpenSSL 的对称加密算法指令主要用来对数据进行加密和解密处理，输入输出的方式主要是文件，当然，也可以是默认的标准输入输出设备。OpenSSL 基本上为所有其支持的对称加密算法都提供了指令方式的应用，这些应用指令的名字基本上都是以对称加密算法本身的名字加上位数、加密模式或其他属性组合而成。比如 DES 算法的 CBC 模式，其对应的指令就是 des-cbc。要查找最新版本的 OpenSSL 支持哪些对称加密算法指令，只要在 OpenSSL 应用程序提示符下输入“-help”，就可以看到在输出信息最后部分（Cipher commands）有很多我们熟悉的算法名称，这些就是 OpenSSL 支持的对称加密算法指令。

OpenSSL 还将所有的对称加密算法指令集成在一个指令程序中，那就是 enc 指令。enc 指令是 OpenSSL 中对称加密算法指令的集大成者，用户可以在它的一个参数中指定使用哪种加密算法和模式，从而统一了对称加密算法指令的入口。enc 指令的格式跟单独一个对称加密算法指令的格式是基本一样的，但是增加了一个算法类型的选择参数选项。比如，下面两种形式的指令是等价的：

```
OpenSSL>enc -des-cbc-in pln.txt -out enc.txt -pass pass:12345678
OpenSSL>des-cbc-in pln.txt -out enc.txt -pass pass:12345678
```

OpenSSL 的对称加密算法指令还可以增加 BASE64 编解码的功能，也就是说，可以在数据加密后将数据进行 BASE64 编码后输出保存，然后在解密的时候先将数据进行 BASE64 解码后再进行解密。这样，对于加密后数据的保存和处理就更加方便，因为 BASE64 编码将加密后的很多不可见字符都编码成可见的字符了。当然，如果你愿意，也可以单独使用 BASE64 编码的功能，OpenSSL 为此也提供了专门的指令，其使用形式跟其他对称加密算法指令是一致的。

OpenSSL 的对称加密算法指令还可以作为使用第三方加密库（通常是硬件加密设备）的应用接口，也就是说，它可以使用第三方的加密库作为完成加密操作的真正设备。这是通过 OpenSSL 的 Engine 机制实现的。当然，使用第三方加密库的前提是你已经安装了第三方的加密设备并成功通过 Engine 机制加载到了 OpenSSL 中。

不过，对 OpenSSL 的所有指令，包括对称加密算法指令，你不能期望过高，这些指令虽然功能强大，但是并没有支持所有可能的加密算法和模式，只是提供了固定的部分功能。比如对称加密算法指令就不支持 76 位的 RC2 加解密或者 84 位的 RC4 加解密等功能。如果你想使用这些灵活的加密模式和算法，那么就需要对 OpenSSL 进行进一步的深入研究，即使用 OpenSSL 的 API。

## 7.2　对称加密算法指令种类

OpenSSL 的对称加密算法指令支持多种算法，每种算法又根据密钥长度和加密模式的不同分成多个指令。此外，enc 指令支持的对称加密算法跟独立对称加密算法虽然大部分一样，但是并非是一一对应的关系，在下面将会将这些区别和对应关系都一一列出。在本书的介绍中，将以 enc 指令支持的算法种类为主线，表 7-1 列出了 enc 指令支持的对称加密算法类型。

**表 7-1　enc 指令支持的对称加密算法**

| 算法描述 | enc 指令参数 | 相应的独立指令 |
|---|---|---|
| AES 算法 128 位 CBC 模式 | -aes-128-cbc | aes-128-cbc |
| | -aes128 或-AES128 | |
| AES 算法 128 位 CFB 模式 | -aes-128-cfb | 无 |
| AES 算法 128 位 ECB 模式 | -aes-128-ecb | aes-128-ecb |
| AES 算法 128 位 OFB 模式 | -aes-128-ofb | 无 |
| AES 算法 192 位 CBC 模式 | -aes-192-cbc | aes-192-cbc |
| | -aes192 或-AES192 | |
| AES 算法 192 位 CFB 模式 | -aes-192-cfb | 无 |
| AES 算法 192 位 ECB 模式 | -aes-192-ecb | aes-192-ecb |
| AES 算法 192 位 OFB 模式 | -aes-192-ofb | 无 |
| AES 算法 256 位 CBC 模式 | -aes-256-cbc | aes-256-cbc |
| | -aes256 或-AES256 | |
| AES 算法 256 位 CFB 模式 | -aes-256-cfb | 无 |
| AES 算法 256 位 ECB 模式 | -aes-256-ecb | aes-256-ecb |
| AES 算法 256 位 OFB 模式 | -aes-256-ofb | 无 |
| Blowfish 算法 CBC 模式 | -bf-cbc | bf |
| | -bf 或-BF | bf-cbc |
| | -blowfish | |
| Blowfish 算法 CFB 模式 | -bf-cfb | bf-cfb |
| Blowfish 算法 ECB 模式 | -bf-ecb | bf-ecb |
| Blowfish 算法 OFB 模式 | -bf-ofb | bf-ofb |
| CAST5 算法的 CBC 模式 | -cast5-cbc | cast5-cbc |
| | -cast-cbc 或-CAST-cbc | cast-cbc |
| | -cast 或-CAST | cast |
| CAST5 算法 CFB 模式 | -cast5-cfb | cast5-cfb |

续表

| 算法描述 | enc 指令参数 | 相应的独立指令 |
| --- | --- | --- |
| CAST5 算法 ECB 模式 | -cast5-ecb | cast5-ecb |
| CAST5 算法 OFB 模式 | -cast5-ofb | cast5-ofb |
| DES 算法 CBC 模式 | -des-cbc | des-cbc |
| | -des 或-DES | des |
| DES 算法 CFB 模式 | -des-cfb | des-cfb |
| DES 算法 ECB 模式 | -des-ecb | des-ecb |
| DES 算法 OFB 模式 | -des-ofb | des-ofb |
| 两个密钥的三重 DES 算法 CBC 模式 | -des-ede-cbc | des-ede-cbc |
| 两个密钥的三重 DES 算法 CFB 模式 | -des-ede-cfb | des-ede-cfb |
| 两个密钥的三重 DES 算法 ECB 模式 | -des-ede | des-ede |
| 两个密钥的三重 DES 算法 OFB 模式 | -des-ede-ofb | des-ede-ofb |
| 三个密钥的三重 DES 算法 CBC 模式 | -des-ede3-cbc | des-ede3-cbc |
| | -des3 或-DES3 | des3 |
| 三个密钥的三重 DES 算法 CFB 模式 | -des-ede3-cfb | des-ede3-cfb |
| 三个密钥的三重 DES 算法 ECB 模式 | -des-ede3 | des-ede3 |
| 三个密钥的三重 DES 算法 OFB 模式 | -des-ede3-ofb | des-ede3-ofb |
| DESX 算法 CBC 模式 | -desx-cbc | desx |
| | -dex 或-DESX | |
| IDEA 算法 CBC 模式 | -idea-cbc | idea-cbc |
| | -idea 或-IDEA | idea |
| IDEA 算法 CFB 模式 | -idea-cfb | idea-cfb |
| IDEA 算法 ECB 模式 | -idea-ecb | idea-ecb |
| IDEA 算法 OFB 模式 | -idea-ofb | idea-ofb |
| RC2 算法 128 位 CBC 模式 | -rc2-cbc | rc2-cbc |
| | -rc2 或-RC2 | rc2 |
| RC2 算法 128 位 CFB 模式 | -rc2-cfb | rc2-cfb |
| RC2 算法 128 位 ECB 模式 | -rc2-ecb | rc2-ecb |
| RC2 算法 128 位 OFB 模式 | -rc2-ofb | rc2-ofb |
| RC2 算法 40 位 CBC 模式 | -rc2-40-cbc | rc2-40-cbc |
| RC2 算法 64 位 CBC 模式 | -rc2-64-cbc | rc2-64-cbc |
| RC4 算法 128 位 | -rc4 | rc4 |
| RC4 算法 40 位 | -rc4-40 | rc4-40 |
| RC5 算法 CBC 模式 | -rc5-cbc | rc5-cbc |
| | -rc5 或-RC5 | rc5 |
| RC5 算法 CFB 模式 | -rc5-cfb | rc5-cfb |
| RC5 算法 ECB 模式 | -rc5-ecb | rc5-ecb |
| RC5 算法 OFB 模式 | -rc5-ofb | rc5-ofb |
| 不使用加密算法 | -none | 无 |
| | 或者“缺省” | |

从表 7-1 可知，不包括不加密的情况在内，OpenSSL 的 enc 指令一共提供 49 中可选的对称加密算法模式，对于一般的文件加密来说，这应该是能够满足了。

对于大部分块加密对称加密算法，OpenSSL 都提供了 CBC、CFB、ECB 和 OFB 四种加密模式。需要注意的是，对于使用两个密钥和三个密钥 EDE 方式的三重 DES 算法，其 ECB 模式的指令名称并没有明确标出，很容易跟 CBC 模式混同，因为块加密算法使用最多的是 CBC 方式。

如果要单独使用 BASE64 编码而不进行数据的加密和解密，那么只要简单地忽略算法类型参数并加上“-a”或者“-base64”选项就可以了，当然，你如果不愿意忽略算法类型，也可以输入“-none”选项。下面三种形式是等价的，仅使用 BASE64 编码的指令：

```
OpenSSL>enc -base64[other_options…]
OpenSSL>enc -none -base64 [other_options…]
OpenSSL>base64 [other_options…]
```

从表 7-1 还可以看出，对于绝大部分对称加密算法来说，都有 enc 参数式的指令和单独的指令两种方式，它们都是等价的，至于选择哪一种，主要取决于用户本身的偏好。不过，enc 指令可能会更快地支持新的对称加密算法，而单独的对称加密算法指令就不一定能得到及时的更新。这跟 enc 实现的机制是有关系的，因为 enc 使用的是 OpenSSL 内部定义的对称加密算法简称作为类型输入参数，只要是 OpenSSL 定义的对称加密算法模式，enc 程序都能通过接口 API 自动支持，不需要因为增加了新的对称加密算法而进行 enc 应用程序的修改。

上面提供了种类繁多的对称加密算法，它们的加密强度和所带来的安全性能也不一样，对于初学者来说，如果你觉得要保护的文件比较重要，而且不是很大的文件，那么推荐使用强度尽可能大的加密算法，比如 3DES 的 CBC 方式。对于小文件来说，这些强度大的算法并不耗费多少时间，而它带来的安全性却大大增加了。

## 7.3　对称加密算法指令参数

相对于其他指令来说，对称加密算法的指令参数比较少，使用起来也相对容易。对称加密算法指令的参数形式如下：

```
OpenSSL>enc -ciphername|none[-in filename][-out filename][-pass arg][-e][-d][-a|-base64][-A][-k password][-kfile filename][-K key][-iv IV][-p][-P][-bufsize number][-nopad][-debug][-engine e][-salt|-nosalt][-S salt]
OpenSSL> ciphername[-in filename][-out filename][-pass arg][-e][-d][-a|-base64][-A][-k password][-kfile filename][-K key][-iv IV][-p][-P][-bufsize number][-nopad][-debug][-engine e][-salt|-nosalt][-S salt]
```

对于大部分对称加密算法指令来说，上述两种方式都是可行而且等同的。

**（1）ciphername 选项**

ciphername 是在表 7-1 中第二列和第三列中出现的参数值之一。如果你选择第一种指令形式，那么就需要选择表 7-1 第二列的参数形式；如果你选择的是后一种独立对称加密算法指令方式，那就应该使用表 7-1 第三列的参数形式。对于第一种对称加密算法指令使

用方式来说，如果你不需要进行任何加密或解密处理，那么可以输入-none 选项替代 ciphername，当然，你也可以简单忽略 ciphername 选项，其效果是一样的。

**（2）in 和 out 选项**

这两个选项分别用于指令输入和输出文件，对于加密操作来说，输入的应该是明文文件，也就是要加密的文件，输出的是密文文件，即经过加密的文件；对于解密操作来说，输入的是经过加密的文件，而输出的是恢复的明文文件。输入和输出文件名本身是没有任何限制的，只要符合习惯的文件名形式即可。需要注意的是，一般不要试图编辑和改变加密后的文件，那样做可能引起文件的部分甚至全部内容不能进行正确的解密！默认的输入和输出文件是标准输入和输出设备，对于 Windows 来说，就是指令行界面。需要注意的是，如果提供的输出文件名是已经存在的文件，那么程序会首先将该文件内容清空！

**（3）口令输入选项 pass，k 和 kfile**

如果你不愿意输入繁杂的用来加密数据的密钥和初始向量，那么可以使用从口令中提取密钥和初始向量的方法。OpenSSL 对称加密算法指令中输入口令的目的正在于此，事实上，OpenSSL 中几乎所有输入的口令都是用作提取密钥的材料，而不是直接用作加密的密钥。pass 选项提供了最灵活的口令输入方式，输入的源可以是标准输入设备、指令行直接输入、提示输入、文件、环境变量和文件描述符，具体的格式介绍读者可以参考 7.4 节的应用实例。k 选项和 kfile 选项都是为了兼容以前的版本而保留的，它们的功能目前都可以使用 pass 选项来实现。k 选项后面输入的参数是口令字符。kfile 选项后面输入的参数是作为口令的文件名，当然，必要的时候应该也提供路径。OpenSSL 的口令文件以第一行作为输入口令。

**（4）e 和 d 选项**

e 和 d 选项分别表示执行加密操作（encryption）和解密操作（decryption），两个参数是互斥的，不能同时出现，但是可以同时不出现，这时候就执行默认的操作，即执行加密操作。这两个选项都不带参数。

**（5）base64，a 和 A 选项**

选项 a 和 base64 的作用是相同的，就是将文件进行 BASE64 的编解码操作。对于加密操作来说，就在数据加密之后进行 BASE64 编码；对于解密操作来说，就在解密操作之前执行 BASE64 解码。A 选项跟 a 或者 base64 选项一起使用才能生效，如果出示了 A 选项，则程序将努力将所有加密数据作为一行进行 BASE64 编码，而不是按照文件原有的换行格式进行 BASE64 编码。需要注意的是，在文件比较大的时候，A 选项经常会导致指令执行失败。上述三个选项都不带参数。

**（6）K 和 IV 选项**

如果你使用这两个选项，意味着你不相信 OpenSSL 从口令提取加密密钥和初始向量的方法，而使用自己直接提供的加密密钥和初始向量，那么这时候你就不再需要使用口令选项。K 选项后面的参数是加密密钥，是以十六进制的方式表示，长度不能超过 64 个字符。IV 选项只有在分组加密算法的某些模式才需要，其参数也是以十六进制的方式表示，长度不能超过 32 个字符。如果输入的参数不是十六进制的字符，那么程序就会报错。上述选项参数输入的长度如果不够，就在后面补零替代。

**（7）salt，nosalt 和 S 选项**

salt 选项指明在从口令提取密钥的过程中使用盐值，这可以增强被加密数据的安全性，事实上，即便不出示此选项，默认指令也会使该选项生效。nosalt 选项跟 salt 选项相反，告诉指令在密钥提取的过程中不使用盐值，一般来说，如果不是特别地为了跟 OpenSSL 一些老版本兼容，不要做这样降低安全性的事情。S 选项后面的输入参数是十六进制编码的真正使用的盐值，其长度不能超过 16 个字符。所有这些选项只有跟口令输入选项一起使用才会生效，如果你使用 K 和 IV 选项方式输入加入密钥，那么这些选项将不会产生任何作用。

**（8）engine 选项**

该选项的参数是 OpenSSL 支持的 Engine 的名称，一般是以字符串表示，比如“cswift”，“nuron”和“pkcs11”等。如果你不知道目前你的 OpenSSL 支持的 Engine 特征字符串，可以输入下列指令，其中有效的 Engine 都会提示出来：

```
OpenSSL>engine -t
```

如果你使用了有效的 Engine，那么你所执行的加密操作实际上都是在第三方加密设备中执行，而不再是使用 OpenSSL 密码库的软件算法。

**（9）p 和 P 选项**

p 选项打印出对称加密算法真正使用的加密密钥和初始向量，输出的格式是十六进制的形式。如果出示了 P 选项，则程序在打印出加密密钥和初始向量后就立刻退出，而不执行真正的加密或解密操作。

**（10）nopad，bufsize 和 debug 选项**

这几个选项之间其实并没有联系，这里之所以放在一起，只是为了节省篇幅。nopad 选项指定不使用默认的 PKCS#5 标准的补丁方式，如果这样做，那么就要确保在块加密算法中输入的数据是加密块长度的整数倍，比如使用 DES 算法，就要保证输入的数据是 8 字节的整数倍。

bufsize 选项的参数指定了读写文件的 I/O 缓存，指定的数字以字节为单位，也可以在数字后面加“k”表示是以 1 024 字节为单位。比如，下面的格式都是可以被接受的：

```
-bufsize 256
-bufsize 8k
```

如果你指定的缓存小于 80 字节，指令程序会自动将它增加为 80 字节，这是进行 BASE64 一行编码所需要的最小长度。

使用 debug 选项后，OpenSSL 的对称加密算法指令会将整个执行过程中 I/O 操作相关的 BIO 列出来。这主要是为了调试目的而使用，对于一般用户来说，该选项很少使用。

## 7.4　应 用 实 例

本节将给出对称加密算法指令的一些具体实例，在下面的例子中，“pln. txt”表示明文文件，“enc. txt”表示经过加密的密文文件，而“rcv. txt”表示恢复的明文文件。

### 7.4.1　不进行加密操作的应用

下面指令将文件 pln. txt 的内容复制到文件 enc. txt 中：

```
OpenSSL>enc -none -in pln. txt -out enc. txt
```

下面的两个指令是等价的，它们对 pln. txt 的文件内容进行 BASE64 编码之后将其存储到文件 enc. txt 中：

```
OpenSSL>enc -base64 -in pln. txt -out enc. txt
OpenSSL>base64 -in pln. txt -out enc. txt
```

要对上述指令编码的 enc. txt 文件进行 BASE64 解码，可以使用下面的指令之一：

```
OpenSSL>enc -a -in enc. txt -out rcv. txt-d
OpenSSL>base64 -in enc. txt -out rcv. txt -d
```

在上述的编码指令中增加“-A”选项将会试图把所有字符编码成一行，不过，处理大文件的时候该选项可能会失败。

### 7.4.2 加密和解密文件的应用

使用 3DES 的 CBC 模式加密 pln. txt 文件并保存到 enc. txt 文件中，下面的两个指令是等价的：

```
OpenSSL>enc -des-ede3-cbc -in pln. txt -out enc. txt -k 12345678
OpenSSL>des-ede3-cbc -in pln. txt -out enc. txt -k 12345678
```

相应的解密指令如下：

```
OpenSSL>enc -des-ede3-cbc -in enc. txt -out rcv. txt -k 12345678-d
```

如果需要对加密后的密文再进行 BASE64 编码，那么可以采用下面形式的指令进行加密和解密：

```
OpenSSL>enc -des-ede3-cbc -in pln. txt -out enc. txt -k 12345678 -e -a
OpenSSL>enc -des-ede3-cbc -in enc. txt -out rcv. txt -k 12345678 -d -a
```

如果你安装了有效的硬件加密设备并且已经通过 Engine 机制加载到 OpenSSL 中，则可以使用硬件加密设备进行数据的加解密，下面的两个指令分别使用一个支持 PKCS#11 接口的硬件加密机进行加密和解密操作：

```
OpenSSL> enc -des-ede3-cbc -in pln. txt -out enc. txt -k 12345678 -e -a -engine
pkcs11
OpenSSL> enc -des-ede3-cbc -in enc. txt -out rcv. txt -k 12345678 -d -a -engine
pkcs11
```

如果使用的第三方的密码库或加密设备算法能够跟 OpenSSL 的软件加密库兼容，那么可以使用第三方加密设备加密，而使用 OpenSSL 软件库进行解密，或者反之。这可以用来测试第三方加密库的算法实现是否规范。

### 7.4.3 多种口令和密钥输入方式应用

OpenSSL 对称加密算法加密数据的时候需要使用密钥，对于块加密算法的某些模式，还需要初始向量，既可以直接输入加密密钥和初始向量，也可以通过口令来提取加密密钥和初始向量。就口令输入来说，也有多种不同的方式，本节将给出多种输入口令和密钥的例子。

下面是 DES 算法 CBC 模式指令直接使用加密密钥和初始向量的例子：

```
OpenSSL>enc -des-cbc -in pln. txt -out enc. txt -K FE01A8B -iv 09BC6DEA
```

通过直接输入的口令提取密钥和初始变量：

```
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass pass:12345678
```

从标准设备（指令行）输入口令提取密钥和初始变量，在需要输入的时候会有提示信息出现：

```
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass stdin
```

从环境变量输入口令提取密钥和初始变量，这首先需要进行环境变量的设置：

```
C:\>set mypass = 1234567
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass env:mypass
```

从文件输入口令提取密钥和初始变量，这首先需要建立文件并在文件的第一行输入使用的口令：

```
C:\>echo 12345678 passfile.txt
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass file:passfile.txt
```

默认的情况下使用口令提取加密密钥和初始向量的时候，会使用一个盐值。可以通过参数选择不使用盐值或者直接设定使用的盐值而不是随机产生，下面分别是不使用盐值和直接使用一个指定盐值的指令：

```
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass pass:12345678 -nosalt
OpenSSL>enc -des-cbc -in pln.txt -out enc.txt -pass pass:12345678 -S A243BC
```

## 7.5　本 章 小 结

本章对 OpenSSL 的对称加密算法指令进行了介绍，给读者展示了 OpenSSL 指令对加密算法的强大支持。

首先，本章介绍了 OpenSSL 对称加密算法指令的总体情况和功能，介绍了 OpenSSL 对称加密算法指令的两种不同使用方式。

接着本章详细解释了 OpenSSL 对称加密算法指令支持的对称加密算法类型及其在指令中的使用方法和特征名称。

本章还对 OpenSSL 对称加密算法指令的参数做了详细的介绍，包括其使用条件、功能和形式等。

最后本章给出了一些具体的指令实例，帮助读者更形象地理解这些指令使用方法。事实上，读者在阅读本章及后续章节的时候，结合 OpenSSL 的具体环境进行操作，理解会更加深入，收益也会更大。

# 第 8 章

# 非对称加密算法指令

## 8.1 非对称加密算法指令概述

非对称加密算法也称为公开密钥算法，其解决了对称加密算法密钥需要预分配的难题，使得现代密码学的研究和应用取得了重大发展。非对称加密算法的基本特点如下：

- 加密密钥和解密密钥不相同；
- 密钥对中的一个密钥可以公开（称为公开密钥）；
- 根据公开密钥很难推算出私人密钥。

根据非对称加密算法的这些特点，我们可以使用非对称加密算法进行数字签名、密钥交换及数据加密等。但是，由于非对称加密算法的加密速度相对于对称加密算法来说慢很多，所以一般不直接用于大量数据的加密，而是用于数据加密密钥的交换和数字签名。

并非所有非对称加密算法都可以同时用于密钥交换和数字签名，根据特点不一样，有的只能用于密钥交换，有的只能用于数字签名，而有的可能两者都可以。数字签名和密钥交换对公开密钥算法的要求有一定的区别，主要有如下两点。

- 密钥交换算法使用公开密钥进行加密，使用私人密钥进行解密；而数字签名算法则使用私人密钥进行加密，使用公开密钥进行解密。
- 密钥交换算法要求从加密密钥（公开密钥）很难推算出解密密钥（私人密钥）；而数字签名算法则要求从解密密钥（公开密钥）很难推算出加密密钥（私人密钥）。

目前来说，常用的非对称加密算法有 RSA、DH 和 DSA 三种。其中 RSA 既可以用于密钥交换，也可以用于数字签名，因为它能够同时满足上述两种不同的要求。DH 算法一般来说只能用于密钥交换，而 DSA 算法则是专用于数字签名的算法。

OpenSSL 对于上述三种方法都支持，在指令里面也提供了相当丰富的应用，这些应用包括 DH 密钥参数的生成和解析，DSA 密钥参数的生成和解析，DSA 密钥生成，RSA 密钥生成和使用 RSA 密钥进行加密和解密。表 8-1 是目前 OpenSSL 支持的非对称加密算法指令。

表 8-1　OpenSSL 的非对称加密算法指令

| 指　令 | 种　类 | 功能描述 |
|---|---|---|
| genrsa | RSA 算法 | 生成并输出一个 RSA 私钥 |
| rsa | RSA 算法 | 处理 RSA 密钥的格式转换等问题 |
| rsautl | RSA 算法 | 使用 RSA 密钥进行加密、解密、签名和验证等运算 |
| gendh | DH 算法 | 生成 DH 密钥参数 |
| dhparam | DH 算法 | 用于生成和处理 DH 密钥参数 |
| dh | DH 算法 | 处理 DH 密钥参数的格式转换等 |
| dsaparam | DSA 算法 | 生成和处理 DSA 密钥参数，并可以生成 DSA 密钥 |
| dsa | DSA 算法 | 处理 DSA 密钥的格式转换和解释等 |
| gendsa | DSA 算法 | 根据 DSA 参数文件生成一个 DSA 密钥 |

事实上，除了上述这些指令外，还有一些指令也提供了生成非对称加密算法密钥和使用非对称加密算法的功能，比如 req 指令就提供了生成 RSA 密钥的功能，而 dgst 指令也提供了使用 RSA 密钥进行数字签名和验证的功能。这些功能我们会在相关指令中再作进一步介绍，而本章仅对表 8-1 中列出的指令进行介绍。

## 8.2　RSA 算法指令

### 8.2.1　RSA 算法特点和 RSA 指令概述

RSA 算法是最常使用的非对称加密算法，该算法具有很好的安全性，并且经过了长期的分析和考验。此外，该算法是一个同时具备签名和密钥交换特性的公开密钥算法，所以在许多应用和协议中都会使用到该算法。

根据 RSA 算法的特点，目前来说 RSA 算法的应用一般就包括两方面：密钥交换和数字签名。事实上，RSA 算法也可以用来进行数据的加密，但是由于其加密速度相对于对称加密算法来说非常缓慢，所以一般不用于大量数据的加密。事实上，无论用于密钥交换还是数字签名，本质上都是使用 RSA 的公钥或者私钥进行数据加密的过程。

RSA 密钥加密的特点是输入数据不能超过 RSA 密钥的长度（根据不同的补齐方式能够加密的数据长度不一样），而输出数据长度总是与 RSA 密钥长度相同，所以对于量大的数据，一般需要预先进行数据分割。解密的时候，RSA 算法每次输入的数据的长度总是跟密钥长度相同，比如 1 024 位的 RSA 密钥，每次执行解密操作的时候输入的数据长度总是1 024位（128 字节）的，如果有多个 1 024 位的密文块，那就分别解密，然后再将所有这些数据链接起来即可得到明文。

对于用于数字签名的 RSA 算法来说，通常是跟信息摘要函数一起使用，这样就可以一次完成加密，比如对于一个大文件，可以先使用 SHA1 或者 MD5 算法转换成 160 位或者 128 位的摘要信息，然后再使用 RSA 算法进行加密从而完成该文件的数字签名过程。在验证签名的时候，使用同样的信息摘要算法对该文件进行运算得到信息摘要值，然后使用 RSA 算法（通常是公钥）进行解密得到签名方的信息摘要值，对比新计算的信息摘要

值和解密得到的信息摘要值，如果一致，则签名验证成功。

在 OpenSSL 中，提供了 rsautl 指令来完成上述的 RSA 算法加密和解密数据的功能，也提供了使用 RSA 算法进行数字签名和验证的功能。此外，dgst 指令也提供了使用 RSA 算法结合信息摘要算法进行数字签名和验证的功能，但是该指令的详细使用方法本章暂时不作介绍，而在信息摘要算法相关的章节中作介绍。

无论什么时候，使用 RSA 算法的首要条件是有一对 RSA 密钥，包括一个公钥和一个私钥。为了生成一个安全的 RSA 密钥对，应用程序要花一定的时间完成，而这时间在实时通信中是难以忍受的，所以一般要预先生成一对密钥。此外，RSA 用于数字证书等 PKI 应用的时候，还具有身份标识的功能，跟具体的实体是一一对应的，这也要求预先产生一个在一定时期内固定不变的密钥。

OpenSSL 提供了生成安全 RSA 密钥对的指令 genrsa，该指令可以产生 RSA 密钥对，而且长度不受限制。相应地，OpenSSL 还提供了对 RSA 密钥各项参数进行解释、格式转换及其他相关的加密和解密等 RSA 密钥管理的指令 rsa。使用这两个 RSA 密钥指令，基本可以完成生成 RSA 密钥和使用 RSA 密钥需要做的一些预处理工作。OpenSSL的 req 指令也提供了 RSA 密钥的生成和封装功能，我们将在证书管理相关的章节介绍该指令。

### 8.2.2 生成 RSA 密钥

**(1) genrsa 指令格式**

使用 RSA 算法的第一步就是生成一个 RSA 密钥，那么首先让我们来看看 OpenSSL 提供的这个密钥生成指令 genrsa 如何使用，并且如何生成我们想要的 RSA 密钥。genrsa 指令的格式如下：

```
OpenSSL>genrsa[-out filename][-passout arg][- des][- des3][- idea][- aes128]
[- aes192][- aes256][- f4][- 3][- rand file(s)][- engine engine_id] [numbits]
```

从上面的指令格式和参数-f4 和-3 可以基本看出，genrsa 指令能够生成基于不同指数(exponent)值的 RSA 密钥，并使用不同的对称加密算法对输出的密钥进行保护。此外，还可以使用第三方加密软件或硬件设备而不是 OpenSSL 本身的库生成 RSA 密钥。

**(2) 输出文件选项 out**

out 选项指定了密钥输出的文件名，输出的密钥是 RSA 密钥对的私钥。RSA 密钥都是成对的，也就是说，应该包含一个公钥和一个私钥，其中公钥包含了模数 n 和指数 e，私钥包含了 d。但在 OpenSSL 中或通常的私钥结构中，会把公钥的参数包含在一起，所以在 OpenSSL 的 genrsa 指令中，仅输出了存储私钥的一个文件，然后在需要使用公钥的时候再从私钥文件中读取公钥参数 n 和 e。如果想得到仅包含公钥参数的公钥结构文件，那么可以使用 rsa 指令从私钥文件中提取并进行结构转换。对于 OpenSSL 来说文件名和其后缀都是没有特别含义的，你可以取任意名字，当然，习惯的用法是以格式来指令后缀，比如“pem”。

**(3) 口令输入选项 passout**

passout 选项指定了输出密钥文件的加密保护口令，其格式跟 OpenSSL 通用的口令输入格式相同，输入源可以是指令输入，也可以是文件、提示输入、环境变量等，详细的参

考本书 6.6 节内容。如果没有出示该参数并且在指令中使用了加密算法选项，那么程序将会在指令行界面提示输入口令。如果没有使用加密算法选项，那么密钥实际上不进行加密，那么 passout 选项将被简单地忽略。最后再次提醒一下，除非是测试，否则建议你提供的口令一定要足够复杂，长度也越长越好，当然，你自己需要能够记住。

**(4) 加密算法类型选项 des，des3，idea，aes128，aes192 和 aes256**

由于 RSA 私钥的安全对 RSA 算法的安全性起到关键的作用，所以，如果你非要把 RSA 私钥保存在不怎么安全的计算机存储器或者内存中，那么一般建议对该输出密钥进行加密。OpenSSL 考虑到了这种尴尬的安全需求，提供了多种对称加密算法对输出私钥进行加密，这些对称加密算法强度不一样，包括：DES、DES3、IDEA、128 位 AES、192 位 AES 和 256 位 AES 算法。你可以根据自己的需要选择其中一种进行加密。加密模式都是 CBC 方式，加密需要的密钥和初始向量都是从提供的口令中提取出来的。使用了加密选项之后，就必须提供口令输入，如果没有显式使用 passout 提供口令，应用程序也会在指令行界面提示输入口令。

**(5) 指数选项 f4 和 3**

RSA 算法的指数 e 是一个公开的参数，属于公钥参数中的一部分，目前来说，常用的值有 3、17 和 65 537，OpenSSL 指令目前仅支持 3 和 65 537，而在很多硬件设备中，常常仅支持 65 537 一种。如果你想使用 65 537 作为指数值，那么使用 f4 选项或者干脆就不使用该类选项，因为该类选项的默认值就是使用 65 537；如果你要使用 3 作为指数值，那么可以使用“-3”选项。

**(6) 密钥长度选项 numbits**

无论什么算法，一般来说密钥的长度可以影响其加密的强度，密钥越长，那么加密强度越大，数据也就越安全。当然，正如无数安全文献和读物所强调的那样，安全是一个相对的概念，我们只能根据具体的需求确定一个折中的安全方案，具体到这里来说，就是选择一个合适的密钥长度。密钥越长，虽然其安全性越高，但是其耗费的时间也就越长，这不仅仅体现在使用密钥进行加密的时候，也体现在生成密钥的时候，如果使用 genrsa 指令生成一个8 192位的 RSA 密钥对，其时间可能是你不能忍受的，不信？马上你就试试看。一般来说，现在 RSA 密钥使用的长度一般是 1 024 位，这对于保护一般的数据长度是可以的，更短的（比如 512 位）其安全性能就要受到怀疑。2 048 位在安全性要求较高的场合也是一个可以的选择，比如你的 RSA 密钥是用在 CA 根证书里面，那么就有必要采用比 1 024 位更长的密钥。基本的原则是：密钥使用的周期越长，使用的场合安全性要求越高，那么密钥要求的长度也就越长。

**(7) 其他选项**

使用 engine 选项可以使用第三方加密库或者硬件设备而不是 OpenSSL 算法库生成 RSA 密钥对。由于 RSA 密钥的私钥安全性具有重要意义，所以怎么保护其私钥都不过分。通常来说，目前采用的方法是采用硬件设备来生成和存放 RSA 密钥，这样的设备包括 Smart Card、USB Key 和加密机等。这些硬件设备通常具备自己的算法处理芯片，具备 RSA 算法的加密、解密、签名和验证等功能，使得 RSA 私钥可以永远不用导出硬件设备却能够正常使用。这种方法相对于保存在计算机硬盘中的文件存储方法要安全得多。engine 选项后面的参数是 engine ID，这个 ID 在编写 engine 接口的时候提供，一般是一个

描述性质的字符串。要正确使用 engine 选项生成 RSA 密钥对，必须首先保证你正常加载了 engine 设备和其相关的驱动。可使用 engine -t 指令测试目前有效的 engine 设备并获取其 ID 名。

随机数文件选项 rand 提供了产生随机数的参考种子文件，这可以进一步增加生成的 RSA 密钥的随机性，从而增加其安全性。如果没有提供该选项，那么程序会从标准输出设备的状态读取信息作为随机数种子。随机数文件可以为任意类型的文件。如果使用了有效的 Engine 设备，而且该设备支持随机数产生等函数，那么 genrsa 指令会调用 Engine 中的随机数产生函数产生随机数，随机数文件能否起作用依具体的 Engine 接口情况而定。

**(8) 应用实例**

生成一个 1 024 位的 RSA 密钥，并输出到 rsaprivkey. pem 文件中，对输出密钥不进行加密，其指令如下：

```
OpenSSL>genrsa -out rsaprivkey. pem 1024
```

下面的指令生成一个 1 024 位的密钥，并采用 DES3 算法进行加密，口令直接从指令行方式提供，输出密钥的文件为 rsaprivkey. pem：

```
OpenSSL>genrsa - out rsaprivkey. pem -des3 -passout pass:xY * &2wG 1024
```

下面的指令生成一个 1 024 位的 RSA 密钥，采用 readme. txt 文件作为随机数文件，采用具备 pkcs11 接口的硬件设备产生 RSA 密钥，而不是采用 OpenSSL 的算法库产生 RSA 密钥。密钥输出到 rsaprivkey. pem，不采用加密措施。因为事实上这里输出的密钥仅有公钥参数，真正的私钥参数 d 保存在硬件设备里面，并没有导出，所以没必要对所谓的“私钥”文件做保护。其指令如下：

```
OpenSSL>genrsa -out rsaprivkey. pem -engine pkcs11 -rand readme. txt 1024
```

一般来说，如果硬件设备本身具有随机数产生功能，那么“- rand”选项就不是很必要，Engine 接口内部函数可能已经调用硬件本身的随机数产生功能产生生成密钥必须的随机数，其随机性一般来说比软件提供的随机性更好。

### 8.2.3 管理 RSA 密钥

**(1) rsa 指令格式**

RSA 密钥保存在文件中后，在使用的时候，我们通常会对密钥进行一些处理，比如重新设置加密口令或者加密算法，从私钥中输出公钥参数或者进行格式转换，等等。OpenSSL 提供了 rsa 指令来帮助用户完成这些可能需要的密钥管理功能。当然，这里所谓的密钥管理功能跟我们通常说的密钥管理功能并不完全相同，通常说的密钥管理功能主要考虑密钥的安全存放和传输等问题，远远比这复杂得多。rsa 指令的格式如下：

```
OpenSSL>rsa[-inform fmt][-outform fmt][-in filename][-passin arg][-out filename]
[-passout arg][-sgckey][-des][-des3][-idea][-aes128][-aes192][-aes256][-text][-noout]
[-modulus][-check][-pubin][-pubout][-engine engine_id]
```

需要指出的是，rsa 指令能够处理的密钥不仅仅是保存在文件中的 RSA 私钥，还可以是通过 engine 调用的存储在硬件中的私钥。

**(2) 密钥格式选项 inform 和 outform**

早期的 OpenSSL 版本的密钥格式非常有限，只提供了 PEM、DER 和 NET 三种类

型，而现在的版本提供的密钥类型则丰富得多，并且考虑了使用 Engine 提供的硬件设备的密钥的功能。表 8-2 列出了 OpenSSL- 0.9.7 版本支持的密钥格式。

rsa 指令并没有对表 8-2 中列出的所有格式提供支持。如果输入的是 RSA 私钥，支持的格式包括 DER 编码格式、PEM 编码格式、PKCS＃12 编码格式、Netscape 编码格式、旧版的 IIS SGC 编码格式及 ENGINE 格式。如果输入的是 RSA 公钥，则比 RSA 私钥少支持一种 PKCS＃12 格式，这是因为 PKCS＃12 格式一般用来封装证书和私钥，而不直接用于封装公钥。

**表 8-2　OpenSSL 指令支持的密钥格式**

| 数据格式 | OpenSSL 定义 | 指令字符串参数 | 其他说明 |
|---|---|---|---|
| DER 编码 | FORMAT_ASN1 | d | |
| PEM 编码 | FORMAT_PEM | p | |
| 文本格式 | FORMAT_TEXT | t | |
| NET 编码 | FORMAT_NETSCAPE | n | Netscape 编码格式 |
| SGC 编码 | FORMAT_IISSGC | | 旧版本 IIS 采用的 SGC 密钥兼容格式 |
| PKCS＃12 编码 | FORMAT_PKCS12 | pkcs12、p12 或者 1 | |
| ENGINE 格式 | FORMAT_ENGINE | e | 使用第三方加密库者硬件中存储的密钥 |

在表 8-2 中，指令字符串参数是只在 inform 或 outform 选项后面填写的参数，它们一般对大小写不敏感，比如对于 DER 编码，可以指定为“- inform D”或者“- inform d”，它们是相同的。

如果指定密钥的输入格式为 ENGINE，那么密钥输入文件（in 选项指定的文件）的内容及其含义依具体的 Engine 接口代码而定，可能是一个 ID 值，也可能是一个公钥文件，甚至可能毫无意义。

rsa 指令支持的密钥输出格式（outform 选项指定）比较少，仅包含 DER 编码格式、PEM 编码格式、NET 编码格式及 SGC 编码格式。其中，SGC 编码格式的指定需要使用-sgckey选项。

**（3）输入和输出文件选项 in 和 out**

in 选项指定了存储输入密钥的文件名，输入文件默认情况下应该是 RSA 私钥文件，但是如果使用了-pubin 选项，那么输入文件就应该是包含 RSA 公钥的文件。文件的具体格式可以使用-inform 选项指定。如果在 inform 选项中指定输入的文件格式是 ENGINE 格式并且使用了-engine 选项，那么输入文件的内容和意义视具体的 Engine 接口代码而定。

out 选项指定了存储的输出文件名，默认情况下输出的内容应该是 RSA 私钥，但是如果使用了-pubout 选项或者输入为公钥，则输出内容是 RSA 公钥。如果使用了 noout 选项，那么本选项将被简单忽略。

**（4）输入和输出口令选项 passin 和 passout**

由于 RSA 私钥的重要性，所以一般都使用了加密保护，加密保护使用的密钥和初始

向量是从用户提供的口令中提取出来的，所以在使用 RSA 密钥的时候也需要提供相同的口令进行密钥的解密。passin 选项指定了获取 RSA 私钥解密口令的源，而 passout 选项则指定了输出 RSA 私钥时进行加密的口令的源。口令源的形式是多样的，可以是直接输入，也可以是从文件、环境变量、标准输入输出等获取，详细请参考相关章节。

如果输入的是公钥或者输出的是公钥，那么 passout 选项将被简单忽略。如果输入的密钥文件需要口令进行解密而没有使用 passin 选项提供口令，那么程序会在指令行提示用户输入解密口令。

(5) 加密算法类型选项

跟 genrsa 指令一样，rsa 指令也提供了六种常用的分组加密算法用于加密 RSA 私钥，这些算法包括 DES、DES3、IDEA、128 位 AES、192 位 AES 和 256 位 AES 算法。如果输入使用了这些加密选项之一但是却没有使用 passout 提供加密口令，那么将会在指令行提示输入加密口令。如果输入的是公钥或者输出的是公钥文件，那么该加密选项将会被忽略。如果使用 engine 选项指定了有效的 Engine 设备，而 Engine 设备正好支持选定的加密算法，那么将采用 Engine 设备提供的加密算法加密生成密钥，而不再是使用 OpenSSL 算法库进行密钥的加密。

(6) 信息输出选项 **text**、**noout** 和 **modulus**

使用 text 选项将会以明文的形式输出密钥各个参数的值，使用 modulus 选项则专门输出模数值。如果使用了 noout 选项，那么就不会输出任何密钥到文件中，即便使用了 out 选项指定了输出文件也会被忽略。

(7) 密钥输入和输出类型选项 **pubin** 和 **pubout**

默认情况下，输入的文件应该是 RSA 私钥文件，如果要输入 RSA 公钥文件，则应该使用-pubin 选项，该选项没有参数。如果输入的是 RSA 私钥，而你希望输出一个相对应的 RSA 公钥，那么使用 pubout 选项就可以实现该目的。使用 pubout 输出公钥时，任何跟密钥加密有关的选项都会被忽略，如 passout 和 des 等，因为 RSA 公钥是可以公开的，一般情况下不需要做任何的保护。

(8) 使用第三方加密库或硬件密钥

如果你要使用硬件里的或者第三方加密库的密钥而不是 OpenSSL 本身的密钥处理函数，那么可以使用 engine 选项指定要使用的第三方加密库或者硬件。engine 选项后面的参数是 Engine 的 ID，通常是一个简短的描述字符串，由相应的 Engine 接口代码决定。

使用 Engine 选项后，rsa 指令中两个部分可能受到影响。首先是输入密钥，如果在 inform 中也指定了输入密钥格式是 ENGINE 格式，那么指令将试图从 Engine 设备中加载密钥而不是直接从文件中读取密钥，当然，这时候 in 选项提供的文件可能包含了一些加载密钥需要的有用信息，也可能是 Engine 设备里面 RSA 私钥对应的公钥参数。其次，如果选定的密钥加密算法在 Engine 中支持，那么将会使用 Engine 中的密钥加密算法加密密钥，而不再是使用 OpenSSL 默认的算法库加密 RSA 私钥。

(9) 其他选项

check 选项使用后，指令将会检测输入 RSA 密钥的正确性和一致性。

如果输入或者输出的密钥是 SGC 格式，那么应该使用 sgckey 选项进行格式的转换和调整。SGC 选项目前一般不使用，它是一种经过修改的 NET 格式，用于旧版本的 IIS 服

务器。

**(10)** 应用实例

将一个加密的 PEM 编码的 RSA 私钥转换成一个不加密的 PEM 编码的 RSA 私钥，虽然这样做的必要性很值得怀疑，但是我们还是给出如下指令实现你的好奇心：

```
OpenSSL>rsa -in privkey. pem -passin pass:12345678 -out privkeypln. pem
```

如果某一天你突然觉得原来使用 DES3 加密的 RSA 私钥保存不够安全，而想要使用 256 位 AES 算法加密的时候，可以使用如下指令：

```
OpenSSL>rsa -in privkey. pem -passin pass:12345678 -out privkeyehc. pem -passout pass:3#r%W8^ -aes256
```

把一个 PKCS#12 格式编码的 RSA 私钥转换成 PEM 编码的 RSA 私钥，并进行加密，其指令如下：

```
OpenSSL>rsa -in privkey. p12 -inform p12 -passin pass:12345678 -out privkeye-hc. pem-passout pass:3#r%W8^ -aes256
```

如果对 RSA 私钥里面的那些参数充满好奇而想看一眼，那么可使用下面的指令实现你的梦想，当然，事实上你看到将是一堆数字，或许没有任何意义：

```
OpenSSL>rsa -in privkey. pem -passin pass:12345678 -text -noout
```

Engine 设备里的私钥是不能导出来的，但是公钥参数可以导出来看看，我们用下面的指令把一个 pkcs11 里面的私钥对应的公钥参数输出到一个文件中：

```
OpenSSL>rsa -in pubkey. pem -inform e -engine pkcs11 -pubout -out pubkey. pkcs11
```

事实上，这里的 pubkey. pem 里面保存的是一个公钥文件，我们使用这个公钥文件的参数查找 Engine 设备里面相应的私钥，然后再导出这个私钥的公钥参数到文件里面。

如果你非常在乎加密速度，那么可以使用 Engine 里面的硬件加密算法解密和加密你要输出的 RSA 私钥文件：

```
OpenSSL>rsa -engine pkcs11 -in privkey. pem -passin pass:12345678 -out privkeye-hc. pem -passout pass:3#r%W8^ -des3
```

最后，让我们输入一个 RSA 公钥而不是 RSA 私钥文件，并打印出其模数值：

```
OpenSSL>rsa -in pubkey. pem -pubin -modulus -noout
```

### 8.2.4　使用 RSA 密钥

**(1) rsautl** 指令格式

RSA 密钥目前的用途主要包括密钥交换和数字签名，但是，RSA 密钥本身也是可以用来直接加密数据的，不过相对于对称加密算法就显得非常缓慢。事实上，不管是密钥交换还是数据加密，都是使用 RSA 公钥加密，使用 RSA 私钥解密，其过程是相同的；而数字签名则相反，使用 RSA 私钥加密，使用 RSA 公钥解密。

无论是使用 RSA 公钥还是私钥进行加密操作，RSA 每次能够加密的数据长度不能超过 RSA 密钥长度，并且根据具体采用的补齐方法不同输入的加密数据最大长度也不一样，而输出长度则总是跟 RSA 密钥长度相同。RSA 的不同补齐方法对应的数据输入输出长度限制如表 8-3 所示，其中的参数字符串是 rsautl 指令中选择该种补齐方式的参数形式。

表 8-3 RSA 算法加密时输入和输出数据长度

| 数据补齐方式 | 输入数据长度 | 输出数据长度 | 参数字符串 |
|---|---|---|---|
| PKCS#1 v1.5 | 少于（密钥长度－11）字节 | 同密钥长度 | -pkcs |
| PKCS#1 OAEP | 少于（密钥长度－41）字节 | 同密钥长度 | -oaep |
| PKCS# v1.5 for SSLv23 | 少于（密钥长度－11）字节 | 同密钥长度 | -ssl |
| 不使用补齐方式 | 同密钥长度 | 同密钥长度 | -raw |

OpenSSL 提供了 rsautl 指令来帮助用户使用 RSA 密钥完成数据的加密、解密、签名和验证等功能。但是，由于 rsautl 指令的签名功能没有使用信息摘要算法，所以不能用来实现对文件或者其他大量的数据体进行真正意义上的数字签名，而只能对符合表 8-3 输入数据长度条件的小量数据进行数字签名操作，也就是 RSA 私钥加密操作。

OpenSSL 的 rsautl 指令的格式如下：

```
OpenSSL> rsautl [-in file] [-out file] [-inkey file] [-keyform fmt] [-pubin]
[-certin] [-sign] [-verify] [-encrypt] [-decrypt] [-pkcs] [-ssl] [-raw] [-oaep] [-hexdump]
[-asn1parse] [-rev] [-engine engine_id]
```

**（2）数据输入和输出文件选项 in 和 out**

in 选项指定了存储输入数据的文件选项，需要注意的是，如果执行的是加密操作（sign 或者 encrypt），那么输入数据的长度一定要符合表 8-3 对输入数据长度的要求，否则 rsautl 指令将不能成功执行。如果是解密操作（verify 或者 decrypt），那么输入数据长度总是跟 RSA 密钥长度相同。

out 选项指定了接收输出数据的文件选项，对于 rsautl 指令来说，如果是加密操作（sign 或者 encrypt），那么输出数据长度总是跟密钥长度相同，比如对于 1 024 位的 RSA 密钥，输出数据长度就总是 128 字节（1 024 位）。

默认的输入输出设备是标准输入和输出设备。

**（3）密钥输入和格式选项 inkey、keyform、pubin 和 certin**

rsautl 指令通过 inkey 选项指定加密或解密操作中要使用的密钥的存储文件。默认情况下，无论是上述四种操作中的哪一种，输入的密钥都应该为 PEM 格式的 RSA 私钥。实际上，因为 OpenSSL 的 PEM 编码，RSA 私钥里面包含了公钥参数，所以即便在执行公钥加密或者解密操作时输入的是 RSA 私钥，那么也能够正确执行，因为指令程序会自动使用 RSA 私钥里面的公钥参数进行相应的公钥操作。

RSA 密钥的输入格式可以是多样的，默认的是 PEM 编码格式，其具体的格式可以通过 keyform 选项来指定，keyform 选项支持的参数可以参考表 8-2。目前来说，如果输入的是 RSA 私钥，rsautl 指令支持的格式包括 DER 编码格式、PEM 编码格式、PKCS#12 编码格式、Netscape 编码格式、旧版的 IIS SGC 编码格式及 ENGINE 密钥格式。如果输入的是 RSA 公钥，则比 RSA 私钥少支持一种 PKCS#12 格式。如果 keyform 指定输入的密钥是 ENGINE 格式（- keyfrom e），那么 inkey 指定的密钥文件内容及意义要根据具体的 Engine 接口而定，可能是一个密钥 ID 字符串，可能是一个公钥文件，也可能是没有任何意义的文件。

如果你在执行公钥解密（verify）操作或者公钥加密（encrypt）操作，那么可以从

inkey 参数输入一个公钥而不是私钥，使用 pubin 选项就可以告诉指令程序将要输入的文件是公钥文件。

如果你希望使用一个证书里面的公钥文件，那么可以使用- certin 选项告诉指令程序 inkey 选项指定将要输入的文件是一个证书文件。同样，证书文件的格式可以通过 keyform 选项来指定，目前支持的证书格式包括 DER 编码、PEM 编码、NET 编码和 PKCS＃12编码格式四种。

**（4）操作类型选项 sign、verify、encrypt 和 decrypt**

rsautl 指令提供了四种 RSA 算法的操作。使用 sign 选项表示执行数字签名操作，事实上，就是使用 RSA 的私钥进行加密操作。使用 verify 选项表示执行数字签名的验证操作，也就是使用 RSA 的公钥进行解密操作。使用 encrypt 选项表示执行数据加密操作，也就是使用 RSA 公钥进行加密操作。使用 decrypt 选项表示执行数据解密操作，也就是使用 RSA 私钥进行解密操作。需要说明的是，这里的数字签名操作只是简单的 RSA 私钥加密操作，没有结合信息摘要算法，所以事实上不能真正对文件等比较大的实际数据进行签名操作。

**（5）数据补齐方式选项 pkcs、ssl、oaep 和 raw**

所谓数据补齐方式是指在利用 RSA 密钥进行数据加密的时候，总是要求被加密数据长度跟 RSA 密钥长度相同，但是输入数据通常不是正好跟 RSA 密钥长度相同，所以需要将不同长度的被加密数据补齐成跟密钥长度相同的数据然后再进行加密；当然，如果数据超出 RSA 密钥长度，则需要自己预先分割。OpenSSL 目前支持的补齐方式包括 PKCS＃1 v1.5 规定的方式、SSL 协议规定的补齐方式、OAEP 补齐方式和不使用补齐的方式。默认的数据补齐方式是 PKCS＃1 v1.5。

PKCS＃1 v1.5 补齐方式是目前使用最广泛的补齐方式，如果要用 PKCS＃1 v1.5 补齐方式，那么可以在指令中使用- pkcs 选项。SSL 协议规定的补齐方式是 PKCS＃1 v1.5 的修订版，本质上区别不大，使用- ssl 选项调用该种方式。这两种基于 PKCS＃1 v1.5 的补齐方式都要求每次加密输入的实际数据长度小于密钥长度－11 字节。

OAEP 方式是 PKCS＃1 v2.0 规定的新补齐方式，对于新的应用程序，一般来说推荐这种新的方式。要用这种方式，在指令中使用-oaep 选项。该种补齐方式要求每次加密输入的数据长度不能超过密钥长度－41 字节的长度。

最后，如果你不愿意使用任何补齐方式，那么-raw 选项是你的选择。但是，如不使用补齐方式，数据长度必须跟 RSA 密钥长度相同。

需要注意的是，如果执行的是签名操作（RSA 私钥加密），那么只能使用 PKCS＃1 v1.5 的补齐方式或者不使用补齐方式。

**（6）engine 选项**

与其他 OpenSSL 指令一样，rsautl 指令也提供了对 Engine 的支持，可以通过 Engine 机制使用第三方算法库或者硬件设备而不是 OpenSSL 本身的 RSA 算法进行加密、解密、签名和验证信息。

engine 选项后面应该给出有数的 Engine 设备 ID，该 ID 一般是一个简短的描述字符串，由 Engine 接口代码决定。在该选项给定了有效的 Engine 接口设备之后，rsautl有两个方面会受到影响。首先是 RSA 密钥的获取会受到影响。如果在使用了 engine 选项后，

keyform 中指定了密钥的输入格式是 Engine 格式，那么指令将试图从 Engine 设备中获取加密密钥。这时候 inkey 选项指定的密钥文件内容就根据具体的 Engine 接口要求而定。

其次，如果使用了 engine 选项，而且 Engine 设备支持 RSA 的公钥加密、私钥解密、私钥加密和公钥解密的函数的全部或其中一部分，那么就会启用 Engine 设备执行加密或者解密的操作，而不是使用 OpenSSL 的 RSA 算法。当然，如果 Engine 不支持或者只支持其中的一部分加密操作，那么不支持的部分就会调用 OpenSSL 算法库本身的 RSA 加密操作。

(7) 其他选项

如果你希望输出数据的十六进制的解析信息，那么可以使用 hexdump 选项来实现。使用 asn1parse 可以看到使用 ASN.1 编码（事实上一般是 DER 编码）的数据对象的明文解析和对应的编码数据，该选项在跟 verify 选项一起使用的时候比较有用，可以用来查看验证的签名的信息。上述两个选项一般来说在执行解密操作的时候意义更大一些，否则只是一些看起来没有意义的乱码。

有时候你可能想把输入的数据颠倒顺序后再进行加密或解密操作，比如把“123456789”变成“987654321”，那么可以使用 rev 选项实现你这个奇怪的想法。也就是说，如果你使用了 rev 选项，虽然你输入的数据是“123456789”，事实上你加密的数据是“987654321”。

(8) 应用实例

在下面的所有例子中，plain.txt 文件表示要加密的明文数据文件，enc.txt 表示已经经过加密的密文数据文件，repln.txt 表示解密得到的明文数据文件。

首先，让我们简单加密和解密一个数据文件：

```
OpenSSL>rsautl -in plain.txt -out enc.txt -inkey pubkey.pem -pubin -encrypt
OpenSSL>rsautl -in enc.txt -out repln.txt -inkey privkey.pem -decrypt
```

如果你有一个 PKCS#12 格式的证书，并且里面包含了证书相应的 RSA 私钥，那么你可以采用下面的指令对数据进行签名和验证：

```
OpenSSL>rsautl -in plain.txt -out enc.txt -inkey cert.pfx -keyform pkcs12 -sign
OpenSSL>rsautl -in enc.txt -out repln.txt -inkey cert.pfx -keyform pkcs12 -certin
-verify
```

需要注意的是，使用 PKCS#12 格式的 RSA 私钥加密的时候，会在指令执行的时候提示输入使用口令，用户需要输入正确的口令才能使用 RSA 私钥。在其他需要使用被加密的私钥的 rsautl 指令时，情况也是一样的。

很幸运，我们有一个 PKCS#11 接口的 Engine，并且它支持 RSA 公钥加密和私钥解密操作，那么我们使用下面的指令来执行 RSA 密钥的加密和解密操作。这时候 inkey 指定的文件里面保护的事实上是一个公钥，Engine 根据这个公钥的参数查找其在Engine硬件中相应的公钥和私钥对象：

```
OpenSSL>rsautl -in plain.txt -out enc.txt -inkey pubkey.pem -keyform e -engine
pkcs11 -encrypt
OpenSSL>rsautl -in enc.txt -out repln.txt -inkey pubkey.pem -keyform e -engine
pkcs11 -decrypt
```

让我们看看十六进制编码的数据到底是怎么回事吧：

```
OpenSSL>rsautl -in enc.txt -inkey pubkey.pem -pubin -verify -hexdump
```

你是否产生了想把输入数据顺序颠倒后再进行加密的想法？那么试试下面的指令：

```
OpenSSL>rsautl -in plain.txt -rev -out enc.txt -inkey pubkey.pem -pubin -encrypt
```

# 8.3　DH 算法指令

## 8.3.1　DH 算法和指令概述

DH 算法（Diffie-Hellman）是最早提出的一种用于密钥交换的公开密钥算法，该算法目前已经广泛应用于各种安全协议。SSL 协议同样支持 DH 算法，所以 OpenSSL 也提供了对该算法的支持。

DH 算法使用之前需要通信双方预先共享两个参数，本原元 g 和模 n，这两个参数对算法的安全性有很关键的影响，所以一般需要预先生成这两个参数并检测其安全性能。我们统称这两个参数为 DH 算法参数。OpenSSL 提供了生成这些必要参数和管理这些参数的指令 dhparam、gendh 和 dh。

跟 RSA 算法不一样，DH 算法本身不定义和进行加密和解密的操作，它只是提供了一种安全交换或者说生成共同的数据加密密钥（会话密钥）的方法。这也意味着 DH 算法的用途仅仅是用于密钥交换，而没有任何别的意义。相应的，OpenSSL 也没有提供利用 DH 算法进行加密和解密操作的指令，甚至没有提供生成 DH 密钥的指令，虽然这事实上是值得考虑的事情。OpenSSL 仅提供了生成 DH 算法参数的指令和其相关管理指令。

事实上，虽然 OpenSSL 目前的版本还保留了三个 DH 算法相关指令，但是 dhparam 指令已经集成了 gendh 和 dh 两个指令的所有功能，在后续的版本中，gendh 和 dh 指令很可能被取消或者赋予新的功能定义。但在本书，还会对这两个指令作详细的介绍。

## 8.3.2　生成 DH 算法参数

**（1）gendh 指令格式**

DH 算法参数包括本原元 g 和模 n，OpenSSL 提供的指令 gendh 和 dhparam 都可以生成 DH 参数，并可以经过编码保存在文件中。gendh 指令的功能目前已经集成在 dhparam 指令中，但是 dhparam 指令还集成了 dh 指令的所有功能，在这部分，我们只介绍 gendh 指令。首先来看看 gendh 指令的格式：

```
OpenSSL>gendh[-out file][-2|-5][-engine engine_id][-rand file][numbits]
```

**（2）输出文件选项 out**

out 选项指定了 DH 算法参数输出和保存的文件名，可以是标准输出设备，比如 Windows 下就是当前指令行界面。gendh 指令没有输入选项。使用 gendh 指令输出的 DH 算法参数都是 PEM 编码的。

**（3）本原元 g 选项 2 和 5**

DH 算法参数指令的主要目的是产生公共模数 n，而本原元 g 是指定的，目前常用的本原元有 2 和 5，虽然 3 也偶尔被使用，但是 OpenSSL 并没有提供支持。默认情况下使

用 2 作为本原元。

**(4) engine 选项**

产生 DH 算法参数的时候 engine 选项的影响体现在两个方面，DH 算法产生函数方面和随机数生成函数方面。如果 engine 指定的设备有效并且支持 DH 算法参数的产生，那么将会调用 Engine 设备提供的 DH 参数生成函数而不再使用 OpenSSL 函数库本身的 DH 参数产生函数。同时，如果 Engine 设备支持随机数生成函数，那么在产生 DH 算法参数需要的随机数的时候，调用的系列随机数函数将直接使用 Engine 设备提供的相应函数，这时候，rand 选项指定的随机数种子文件是否有意义就依具体的 Engine 接口而定。

**(5) 随机数文件选项 rand**

rand 选项指定了产生随机数时使用的随机数种子文件，该文件一般来说可以为任意类型的文件。如果没有指定，指令会从其他途径获取必要的随机数种子。如果使用了 engine选项并且该 Engine 接口支持随机数产生函数，那么该 rand 选项指定随机数文件的意义及是否被真正使用要根据具体的 Engine 接口而定。

**(6) 密钥长度选项 numbits**

DH 密钥长度依据 DH 算法参数的长度而定，所以，生成的 DH 算法参数的长度决定了 DH 密钥的长度。一般来说，现在 512 位的 DH 密钥是可以信任的，当然，如果你愿意，也可以采用更长的密钥。密钥越长，生成 DH 参数的时间越长，而安全性也越高。

**(7) 应用实例**

生成一个本原元为 2，长度为 512 为的 DH 算法参数并保存在 dh512.pem 文件里面：

```
OpenSSL>gendh -out dh 512.pem 512
```

使用支持 PKCS#11 接口的 Engine 设备生成 DH 算法参数，该 DH 参数长度为 1 024 位，有效指令如下：

```
OpenSSL>gendh -out dh1024.pem-engine pkcs11 1024
```

### 8.3.3 管理 DH 算法参数

**(1) dh 指令格式**

我们已经可以生成任何想要的 DH 算法参数并保存在某个文件里面，这个文件不需要保密，因为它们本来就是公开的参数。但并非说我们就可以不关心这个保存了 DH 算法参数的文件了，为了使用它，我们有时候可能需要做一些管理工作，比如格式转换、安全性测试及转换成 C 编码等操作。OpenSSL 的 dh 指令提供了满足这些常见要求的功能。

首先来看看 dh 指令的格式：

```
OpenSSL>dh[-inform fmt][-outform fmt][-in infile][-out outfile][-engine engine_id][-check][-C][-text][-noout]
```

**(2) 输入和输出格式选项 inform 和 outform**

inform 和 outform 选项指定 DH 参数编码输入和输出文件的格式，默认情况下是 PEM 编码。目前来说，支持的格式包括 PEM 编码和 DER 编码两种格式，格式的相应参数见表 8-2。如果输出信息不是编码的 DH 参数，比如- text 和-C 选项的输出，则不会受到 outform 格式选项的影响。

**(3) 输入和输出文件选项 in 和 out**

in 选项指定了输入的 DH 参数文件，该文件应该保存了 PEM 编码或者 DER 编码格式的 DH 参数，如果你是使用 gendh 指令生成的 DH 参数文件，那么肯定就是 PEM 编码的。如果是 DER 编码的 DH 文件，则需要在 inform 中指定其格式。

out 选项指定了输出文件，包括输出格式转换后 DH 参数和 text 选项输出的明文解析信息。

**(4) DH 参数检测选项 check**

DH 参数文件存放一段时间之后，你如果对该文件产生了怀疑，可以使用 check 选项对其中的 DH 参数进行检查。check 选项提供的检查包含四个方面：模数是否正确，模数是否安全，本原元 g 是否正确及本原元是否合适。如果检查有问题，指令会输出提示信息。

**(5) 输出 C 语言代码选项 C**

DH 算法参数在使用的时候，既可以从文件读入，也可以直接集成在代码里面。OpenSSL 提供了从 DH 文件参数转换成相应的 C 语言代码的方法，就是使用 C 选项。使用该选项后，输出三部分信息：模数数组、本原元 g 数组及 getdh 函数。需要注意的是，C 语言的输出代码只输出到标准输出设备，不会输出到 out 选项指定的文件中。下面是一个使用 C 选项后输出的一个 512 位 DH 算法参数 C 代码的例子，是不是觉得很有意思？

```
static unsigned char dh512_p [] = {
        0xBD, 0x14, 0x06, 0xDB, 0x1C, 0x4F, 0x11, 0x90, 0x79, 0x5D, 0x9F, 0xF1,
    0x9D, 0xBB, 0xDF, 0xD7, 0xF2, 0x57, 0x87, 0x80, 0xFE, 0x12, 0x83, 0x9E,
    0xB4, 0xE0, 0xDE, 0xD7, 0x34, 0x8A, 0x00, 0xA4, 0xB2, 0xF7, 0x19, 0x92,
    0xF3, 0x4C, 0xFA, 0xD1, 0xDD, 0xEE, 0xBC, 0x00, 0x28, 0xA5, 0x99, 0x74,
    0xAA, 0x02, 0xAD, 0x4F, 0x48, 0x6F, 0x7A, 0x2D, 0x53, 0xC9, 0x2D, 0xC7,
    0x3C, 0x30, 0x2A, 0x63,
    };
static unsigned char dh512_g[] = {
    0x02,
    };
DH *get_dh512()
    {
    DH *dh;
    if ((dh = DH_new()) == NULL)return(NULL);
    dh->p = BN_bin2bn(dh512_p,sizeof(dh512_p),NULL);
    dh->g = BN_bin2bn(dh512_g,sizeof(dh512_g),NULL);
    if((dh->p == NULL)||(dh->g == NULL))
            return(NULL);
    return(dh);
    }
```

**(6)** 其他选项

engine 选项指定了要使用的 Engine 设备，如果该 Engine 接口支持在使用 dh 指令中要使用到的某个函数或者操作，那么就会使用 Engine 设备而不是 OpenSSL 本身算法库的函数。

text 选项告诉指令输出 DH 算法参数的明文解析信息，这包括模数 n 和本原元 g 的十六进制编码数据。

noout 选项的使用会忽略输出选项 out，不输出 DH 编码参数到 out 选项指定文件或标准输出设备。

**(7)** 应用实例

格式转换是可能经常碰到的事情，使用下面的指令可轻松将 PEM 编码的 DH 参数文件转换成 DER 编码的 DH 参数文件：

```
OpenSSL>dh -in dh512.pem -out dh512.der -outform d
```

检测 DH 参数文件保存的 DH 参数是否正确，因为我们不需要它输出 DH 参数，所以使用 noout 选项免得浪费存储资源：

```
OpenSSL>dh -in dh512.pem -noout -check
```

输出 DH 算法参数的明文解释，但愿你能够看出一点门道，因为那都是十六进制编码的数据：

```
OpenSSL>dh -in dh512.pem -noout -text
```

如果你是一个使用 OpenSSL 的 API 进行编程的程序员，那么下面的指令可以减轻你写 DH 算法参数获取代码的负担：

```
OpenSSL>dh -in dh512.pem -noout -C
```

### 8.3.4 更丰富和综合的 DH 算法参数指令

**(1) dhparam** 指令格式

新版本（至少在当前的 openssl-0.9.7 的版本）的 dhparam 指令已经集成了 gendh 指令和 dh 指令的所有功能，而 dh 指令和 gendh 指令在以后的版本中很可能不再被支持，所以，如果你习惯了 gendh 指令和 dh 指令的使用方式，那么现在赶紧转移到 dhparam 指令上来。先看看 dhparam 指令的格式：

```
OpenSSL> dhparam [-inform fmt] [-outform fmt] [-in filename] [-out filename]
[-dsaparam] [-noout] [-text] [-C] [-2] [-5] [-check] [-engine engine_id] [-rand file]
[numbits]
```

可以看到，新东西太少了，除了一个 dsaparam 选项外，其他选项都很熟悉，如果你已经熟悉了 gendh 指令和 dh 指令的选项，那么对这个指令的大部分选项也是非常熟悉的。

**(2) DSA** 风格的 **DH** 参数

dsaparam 选项是 dhparam 指令比 gendh 指令和 dh 指令唯一多出来的选项，该选项告诉指令生成一个 DSA 风格的 DH 参数，而不是使用典型的 DH 参数生成方式。

所谓 DSA 风格的 DH 参数是利用 DSA 参数类型跟 DH 参数具有相似性的特点，先生成一个 DSA 参数，然后将其转换为 DH 参数。DSA 参数生成相对于 DH 参数来说速度更快，而其达到相同安全性能需要的密钥长度更短一些，所以使用 DH 算法的时候效率就显

得更高。但也是有代价的，使用 DSA 风格的 DH 算法参数，最好为每一个应用生成一个算法参数，否则容易受到一种称为“小群”（small-subgroup）的方法的攻击。

使用了 dsaparam 选项之后，如果使用了输入文件，那么输入文件格式都被视作 DSA 参数文件，dhparam 指令将会把 DSA 参数转换成 DH 参数。输出则是一个 DSA 风格的 DH 参数文件。

**（3）** 还有什么不同?

还有什么不同么？我们不太想得出来了，如果非说有，那么就是你在生成 DH 参数后并保存到文件的时候，可以选择是保存成 PEM 格式还是 DER 格式，而不再受到 gendh 指令限制只能保存成 PEM 格式，然后再使用 dh 指令才能转换成 DER 格式。

**（4）** 应用实例

使用 dhparam 指令替代 gendh 指令生成一个 DH 密钥参数，使用 5 作为本原元，输出格式为 DER 格式，指令如下：

```
OpenSSL>dhparam -outform d -out dh512.pem -5 512
```

让我们看看怎么生成一个 DSA 风格的 DH 密钥参数，试试下面这个指令，是不是速度快很多?

```
OpenSSL>dhparam -dsaparam -out dh512.pem 512
```

## 8.4　DSA 算法指令

### 8.4.1　DSA 算法和 DSA 指令概述

DSA 算法是美国的国家标准数字签名算法，它只有数字签名的功能，而没有密钥交换的功能，虽然通过一些简单的改变也可以实现密钥交换的功能，但那显然已经不能称为标准的 DSA 算法。前面说过，所谓数字签名，也就是使用私钥进行加密操作，使用公钥进行解密操作，并且算法应该具备从公钥很难推算出私钥的特点。DSA 算法作为一种数字签名算法，显然应该满足这个特点。

要使用 DSA 算法，第一步当然要生成一对 DSA 密钥，但是通常在生成 DSA 密钥之前，需要生成一个 DSA 密钥参数，然后根据这个参数来生成真正的 DSA 密钥。DSA 密钥参数决定了 DSA 密钥的长度。OpenSSL 提供了三个相关的 DSA 指令来满足不同的需求。首先是 dsaparam 指令，该指令主要用来生成 DSA 密钥参数，并提供了一些格式转换、C 代码生成等其他类似于 dhparam 指令的功能。一组 DSA 参数可以用来生成多个不同的 DSA 密钥，而不是仅仅对应于一个 DSA 密钥。gendsa 指令用来从现有的 DSA 参数中生成 DSA 密钥，使用相同的 DSA 参数可以生成不同的 DSA 密钥，不过这需要 rand 选项指定的随机数种子文件配合。同样，为了提供一些可能需要的 DSA 密钥管理功能，如格式转换、保护口令更改等，OpenSSL 提供了 dsa 指令。

显而易见，DSA 密钥参数跟 DH 算法的密钥参数意义是不一样的，使用方式也是不一样的，DSA 算法在使用之前就应该跟 RSA 算法一样先生成一对 DSA 密钥。那么为什么要将生成 DSA 密钥参数跟生成 DSA 密钥区分开来呢？首先，因为要使用一个 DSA 密钥，必须预先共享其参数 p、q 和 g；其次，因为生成一组 DSA 密钥参数所耗费的时间比

较多，而一组 DSA 密钥参数可以用来生成许多组 DSA 密钥，所以分开来就可以避免每次生成一对 DSA 密钥都要重新生成 DSA 密钥参数，耗费大量的时间。

OpenSSL 没有提供类似 rsautl 指令的专门 DSA 算法使用指令，但是可以在 dgst 指令中使用 DSA 算法和密钥进行数字签名和验证的操作。这基本上满足了 DSA 算法使用的要求。关于 DSA 密钥在 dgst 指令中的使用细节，我们将在介绍 dgst 指令的相关章节给予介绍。

### 8.4.2 生成和管理 DSA 密钥参数

**（1）dsaparam 指令格式**

OpenSSL 提供的生成和管理 DSA 密钥参数的指令 dsaparam 的功能非常类似于 dhparam指令，其提供的选项形式也非常相似。生成一组 DSA 密钥参数不仅仅可以用来生成不同的 DSA 密钥，还可以用于生成 DH 密钥参数，这在前面的 DH 密钥参数生成相关章节已经介绍过了。

dsaparam 指令格式如下：

```
OpenSSL> dsaparam [-inform fmt] [-outform fmt] [-in filename] [-out filename]
[-noout] [-text] [-C] [-rand file(s)] [-engine engine_id] [-genkey] [numbits]
```

很明显，跟 dhparam 选项形式非常相似，但是为了保持本章节一定的独立性，还是有必要对这些参数选项做简单的介绍。

**（2）输入和输出格式选项 inform 和 outform**

inform 和 outform 选项指定了输入 DSA 密钥参数和输出 DSA 密钥参数或者 DSA 密钥的编码格式，目前支持的格式有 PEM 和 DER 两种，默认的是 PEM 格式。其参数的具体形式请参考表 8-2。

**（3）输入和输出文件选项 in 和 out**

in 选项指定了输入文件名，一般来说，如果使用了 in 选项，表示是从已有的 DSA 密钥参数中生成新的 DSA 密钥或者对密钥参数进行格式转换等管理操作。如果要生成新的 DSA 密钥参数或者利用新生成的密钥参数生成 DSA 密钥，那么一般不需要使用 in 选项。

out 选项指定了输出文件名，输出的信息可能是 DSA 密钥参数、DSA 密钥和 text 选项给出的明文解析信息。输出的 DSA 密钥参数和 DSA 密钥的编码格式由 outform 选项指定。如果输出的是 DSA 密钥，那么其形式是 DSA 私钥的形式，但是包含了 DSA 公钥参数。

**（4）engine 选项**

如果使用了有效的 Engine 设备并提供该选项，而且在生成 DSA 密钥参数的过程中执行的操作或者函数在 Engine 中支持，比如大数操作、随机数生成及信息摘要函数等，那么就会采用 Engine 设备而不是 OpenSSL 默认的算法库的函数进行实际的操作和运算。同样，engine_id 是一个简短的描述型字符串，由 Engine 接口决定。

**（5）C 语言代码输出选项**

某些时候，我们可能需要将 DSA 密钥参数写入到 C 语言的代码中去，OpenSSL 提供了从一组 DSA 密钥参数转换成代码的方法，就是在 dsaparam 指令中使用 C 选项。下面是使用 dsaparam 指令的 C 选项生成的一段 DSA 密钥参数的 C 语言代码。

```
static unsigned char dsa512_p[]={
```

```
    0xA8,0xA3,0x18,0x31,0x7B,0x4A,0x7A,0x31,0x6C,0xA0,0xC0,0x3E,
    0xB5,0xD8,0x7F,0x8F,0x86,0x4F,0x2E,0x3B,0xAD,0xB9,0x78,0x1A,
    0x62,0xCB,0xEC,0x5F,0x5D,0x1D,0x5E,0xE2,0x53,0x89,0xB5,0xDF,
    0x7D,0xD8,0x20,0x5E,0x17,0x6D,0x5C,0x79,0x35,0xBF,0xFF,0xCD,
    0x7D,0x78,0x6B,0xE0,0xF1,0xE9,0x27,0xF9,0x27,0x47,0x9D,0xCB,
    0x9D,0x50,0xC4,0x59,
    };
static unsigned char dsa512_q[]={
    0x89,0x4D,0x0F,0x8F,0xBE,0x7C,0xDD,0x8B,0xB3,0xA9,0x33,0x42,
    0xCF,0x6F,0xB2,0x62,0x95,0x9D,0x98,0x85,
    };
static unsigned char dsa512_g[]={
    0x7D,0x1A,0x20,0x8D,0x8B,0x22,0xB0,0x10,0x69,0x55,0x4F,0xEE,
    0x57,0xFC,0x12,0x8B,0xEA,0xEE,0x29,0x8E,0xB6,0x20,0x84,0x51,
    0x7D,0x4C,0xD3,0xEE,0xB8,0x70,0x82,0x5F,0x84,0xE3,0x1E,0xEC,
    0x6B,0x7A,0xDA,0xDA,0x57,0x44,0xD7,0x26,0x55,0x89,0x60,0xA2,
    0x93,0x18,0x06,0x5B,0x3A,0xE1,0x4F,0x7A,0xF8,0x76,0x79,0x42,
    0xC8,0xED,0xF4,0x7D,
    };
DSA *get_dsa512()
    {
    DSA *dsa;
    if((dsa=DSA_new()) == NULL) return(NULL);
    dsa->p=BN_bin2bn(dsa512_p,sizeof(dsa512_p),NULL);
    dsa->q=BN_bin2bn(dsa512_q,sizeof(dsa512_q),NULL);
    dsa->g=BN_bin2bn(dsa512_g, sizeof(dsa512_g),NULL);
    if((dsa->p==NULL)||(dsa->q==NULL)||(dsa->g==NULL))
    {DSA_free(dsa);return(NULL);}
    return(dsa);
    }
```

**(6) 直接生成 DSA 密钥选项 genkey**

虽然 dsaparam 指令通常用来生成 DSA 密钥参数，但是如果你想直接生成一个 DSA 密钥而不是 DSA 密钥参数，那么可以使用 genkey 选项。该选项使得输出到 out 选项指定的输出文件的是一个 PEM 编码或者 DER 编码的 DSA 私钥。但是，该私钥是没有进行加密的，如果要安全地存放，必须使用下面将要介绍的 dsa 指令进行加密保存。使用该选项后，输出的虽然只是一个 DSA 私钥，但是私钥结构里面已经包含了 DSA 公钥的参数，所以，如果要得到相应的 DSA 公钥，那么只要从这个 DSA 私钥里面提取相应的参数就可以了，dsa 指令可以帮助你从一个 DSA 私钥得到相应的 DSA 公钥。

**(7) DSA 密钥参数长度选项 numbits**

DSA 密钥参数决定了用其产生的 DSA 密钥的长度，一般来说，512 位的 DSA 密钥只能提供很短期的安全性能，所以一般建议使用 1 024 位的 DSA 密钥。

**(8) 其他选项**

rand 选项跟其他指令同名选项一样，指定了一个随机数种子文件，默认情况下指令也会从其他可能的途径自动获取。

text 选项告诉指令输出 DSA 密钥参数的明文解析，为了满足大家对 text 选项的好奇心，我们看看下面使用了 text 选项的 DSA 密钥参数输出结果：

```
DSA-Parameters:(512 bit)
    p:
        00:a8:a3:18:31:7b:4a:7a:31:6c:a0:c0:3e:b5:d8:
        7f:8f:86:4f:2e:3b:ad:b9:78:1a:62:cb:ec:5f:5d:
        1d:5e:e2:53:89:b5:df:7d:d8:20:5e:17:6d:5c:79:
        35:bf:ff:cd:7d:78:6b:e0:f1:e9:27:f9:27:47:9d:
        cb:9d:50:c4:59
    q:
        00:89:4d:0f:8f:be:7c:dd:8b:b3:a9:33:42:cf:6f:
        b2:62:95:9d:98:85
    g:
        7d:1a:20:8d:8b:22:b0:10:69:55:4f:ee:57:fc:12:
        8b:ea:ee:29:8e:b6:20:84:51:7d:4c:d3:ee:b8:70:
        82:5f:84:e3:1e:ec:6b:7a:da:da:57:44:d7:26:55:
        89:60:a2:93:18:06:5b:3a:e1:4f:7a:f8:76:79:42:
        c8:ed:f4:7d
```

noout 选项告诉指令不用输出 DSA 密钥参数或者 DSA 密钥到 out 选项指定的文件或者标准输出设备中（通常是指令行界面）。一般来说，在你想测试和查看 DSA 密钥参数的时候，为了避免还要到相应的目录中去删除一个无意义的文件的时候会使用这个选项。

**(9) 应用实例**

简单地生成一个 512 位的 DSA 密钥参数，并以 PEM 编码的格式输出到文件中：

```
OpenSSL>dsaparam -out dsa512.pem 512
```

将上面指令生成的 DSA 密钥参数转换成 DER 编码的格式并保存到新的文件中：

```
OpenSSL>dsaparam -in dsa512.pem -out dsa512.der -outform d
```

下面的指令将一个 512 位的 DSA 密钥参数文件转换成一段 C 语言代码，并输出到指令行界面中：

```
OpenSSL>dsaparam -C -noout -in dsa512.pem -inform p
```

使用 Engine 设备直接产生一个 DSA 私钥，并以 PEM 编码的格式保存到文件中：

```
OpenSSL>dsaparam -engine pkcs11 -outform p -genkey -out dsa1024.pem 1024
```

是不是想看看 DSA 密钥参数里面到底有些什么东西？那么用下面的指令吧：

```
OpenSSL>dsaparam -text -noout -in dsa512.pem -inform p
```

### 8.4.3　生成 DSA 密钥

**（1）gendsa 指令格式**

DSA 密钥是在 DSA 密钥参数的基础上产生的，一对 DSA 密钥可以包含三个部分：DSA 密钥参数（p、q 和 g）、DSA 私钥和 DSA 公钥。DSA 密钥参数是公开的，甚至可以为一组网络用户所共享，即这组网络用户使用相同的 DSA 密钥参数产生各自的 DSA 密钥对。

dsaparam 指令为我们生成了 DSA 密钥参数，紧接着我们就要使用 OpenSSL 提供的 gendsa 指令生成真正要使用的 DSA 密钥对。虽然说是密钥对，但是 gendsa 指令只输出 DSA 私钥，这是因为 DSA 私钥里面已经包含了 DSA 公钥的所有参数，所以如果需要使用相应的 DSA 公钥，那么可以通过 DSA 私钥来获得。dsa 指令提供了从 DSA 私钥输出相应 DSA 公钥的功能。

gendsa 指令的格式如下：

```
OpenSSL> gendsa[-out filename][-passout arg][-des][-des3][-idea][-aes128]
[-aes192][-aes256][-rand file][-engine-engine_id] [paramfile]
```

**（2）输出文件选项 out**

out 选项指定了保存生成的 DSA 密钥的文件，如果没有使用 out 选项，那么 DSA 密钥将会输出到标准输出设备，一般就是当前指令行界面。输出密钥的编码格式 OpenSSL 没有提供可以选择的余地，只能是 PEM 编码。

**（3）输出 DSA 私钥保护口令 passout**

如果输出的 DSA 私钥是保存在文件里面，那么对私钥进行加密保护就显得非常必要。跟其他密钥保护的方式一样，OpenSSL 提供了基于口令的加密保护方式。passout 指定了获取加密口令的源和方法，口令可以从指令参数、文件、环境变量等获得，详细方式和方法请参考相关章节的内容。passout 选项只有在指定了 DSA 私钥使用加密选项之后才会有效，否则该选项将被简单忽略。

**（4）密钥加密算法选项**

对 DSA 私钥进行保护的重要性我们已经反复强调多次，现在就不用说为什么了。加密算法选项指定了使用什么对称加密算法来对 DSA 私钥进行加密，可选的算法有 6 种：DES、DES3、IDEA、128 位 AES、192 位 AES 和 256 位 AES。这些算法使用的加密密钥和初始变量都经过特定的算法从提供的口令中获取。如果使用了上述加密算法选项之一但却没有使用 passout 选项指定加密口令，那么指令会在指令行界面提示用户输入口令。如果没有使用上述任意一种加密算法，那么对 DSA 密钥将不会进行加密，这当然很危险！

**（5）engine 选项**

engine 选项似乎紧紧跟随每一个 OpenSSL 指令，在这里也不例外。在 gendsa 指令里使用 engine 选项指定有效的 Engine 之后，受影响最显著的地方就是加密密钥的算法，也就是说，如果加密的密钥算法在指定的 Engine 接口中支持，那么指令就会调用 Engine 设备对 DSA 密钥进行加密。其次，产生随机数的操作也可能会在 Engine 设备中进行，只要 Engine 接口支持这些相应的操作。

**(6) DSA 密钥参数选项**

该选项指定用于生成 DSA 密钥的 DSA 密钥参数文件，DSA 密钥的长度就取决于文件里面的 DSA 密钥参数。遗憾的是对于该文件的格式你没有选择，只有输入 PEM 编码的 DSA 密钥参数才能够正确运行 gendsa 指令。如果你拥有的是一个 DER 编码的 DSA 密钥参数，那么请使用 dsaparam 指令先将其转换成 PEM 编码的 DSA 密钥参数。

**(7) 随机数选项**

生成 DSA 密钥的过程同样需要随机数的参与，既然有随机数参与，我们就需要随机数种子文件，rand 选项一直就充当这样的功能。当然，如果你不提供该文件，那么指令也会想办法从其他可能的途径获取随机数种子。在使用了 Engine 的情况下，甚至硬件设备有可能自己能够支持产生随机数种子的操作。

**(8) 应用实例**

首先还是看看最简单的例子，从一个 512 位的 DSA 密钥参数文件生成一个 DSA 密钥并以不加密方式保存在文件中：

```
OpenSSL>gendsa -out dsa512.key dsa512.pem
```

上述的做法显然是不妥当的，让我们选用 256 位的 AES 算法对输出 DSA 密钥进行加密后再保存，口令直接从指令参数输入：

```
OpenSSL>gendsa -out dsa512.key -aes256 -passout pass:12345678 dsa512.pem
```

### 8.4.4 管理 DSA 密钥

**(1) dsa 指令格式**

用户经常会有各种奇怪的需求和想法，尤其是为了加强安全性，对 DSA 密钥也一样。并非说生成一个 DSA 密钥就可以高枕无忧了，可能某一天睡了个午觉之后，因为做了一个恶梦，突然觉得用来加密的 DSA 私钥的口令不再安全了，那么需要立刻更换这个保护口令。显然，gendsa 指令不能满足要求，不能随便使用一个新的 DSA 密钥对，幸好 OpenSSL 考虑到了这种需求，提供了 dsa 指令。当然，dsa 指令能做得更多一点。

```
OpenSSL> dsa [-inform fmt] [-outform fmt] [-in filename] [-passin arg] [-out filename] [-passout arg] [-cipher_name] [-engine-engine_id] [-text] [-noout] [-modulus] [-pubin] [-pubout]
```

**(2) 输入和输出格式选项 inform 和 outform**

inform 和 outform 选项分别指定了输入 DSA 密钥和输出 DSA 密钥的编码格式，目前支持的格式包括 PEM 编码和 DER 编码两种。

**(3) 输入和输出密钥类型选项 pubin 和 pubout**

默认情况下，输入和输出的密钥都应该是 DSA 私钥，但是，有时候我们可能需要从一个输入的 DSA 私钥里面获取一个相应的 DSA 公钥发送给签名验证方，那么就可以选择 pubout 选项来输出一个 DSA 公钥。在某些情况下，我们甚至可能输入一个 DSA 公钥，对它的信息进行解释或者进行格式转换。因为 DSA 公钥不需要加密，所以如果使用了 pubin 或者 pubout 选项，相应的 passin 和 passout 选项指定的口令也会被忽略，输出的公钥不会被加密。

**(4)** 输入和输出文件选项 **in** 和 **out**

in 指定了输入 DSA 密钥的保存文件，默认情况下输入的文件应该保存一个 PEM 编码或者 DER 编码的 DSA 私钥。如果使用了 pubin 选项，则指令认为输入文件保存的是一个没有加密的 DSA 公钥。密钥的编码格式默认是 PEM，如果是 DER 格式，则需要使用 inform 选项指定。

out 选项指定了输出 DSA 密钥的保存文件，如果输入的是 DSA 私钥，默认情况下输出的也是 DSA 私钥，但是如果使用了 pubout 选项，则输出的将是一个 DSA 公钥。如果输入是一个 DSA 公钥，那么输出也是一个 DSA 公钥。输出公钥的时候不会对密钥进行加密操作。默认情况下输出的密钥编码格式是 PEM，如果需要输出密钥编码格式为 DER 的密钥，那么可以使用 outform 选项指定。如果使用了 text 和 modulus 选项，还会在 out 指定的文件中输入明文解析信息。使用了 noout 选项将不会输出编码密钥数据。

**(5)** 输入和输出口令选项 **passin** 和 **passout**

passin 选项指定了解密 DSA 私钥需要的口令的方式和源，口令可以从多种渠道获取，详细的参考 6.6 节的介绍。如果输入了 DSA 私钥而没有使用 passin 选项，那么指令会从指令行界面提示输入解密密钥数据需要的口令。如果使用了 pubin 选项，那么 passin 选项会被忽略。

passout 选项指定加密输出的 DSA 私钥需要的口令的方式和源，口令的详细使用方法和格式参考 6.6 节的介绍。如果选择了加密算法但是没有使用 passout 选项，那么指令会从指令行界面提示用户输入保护密钥的口令。如果输入的是 DSA 公钥（使用 pubin 选项）或者输出的是 DSA 公钥（使用 pubout 选项），那么 passout 选项就会被忽略。

**(6)** 加密算法选项

dsa 指令提供了更加丰富的密钥加密算法，理论上，所有 OpenSSL 支持的对称加密算法都可以在这里用于加密 DSA 私钥。使用的方式是直接输入该对称加密算法的合法名称，具体可以参考表 7-1 中 enc 指令参数字段。

**(7)** **engine** 选项

engine 选项指定使用 Engine 设备中支持的对称加密算法替代 OpenSSL 算法库中的加密算法对 DSA 私钥进行加密或者解密。例如，如果 Engine 设备支持 DES3 算法，而且你指定了使用 DES3 算法对 DSA 密钥进行加密，那么指令就会调用 Engine 设备中相应的 DES3 算法对 DSA 私钥进行加密。

**(8)** 解析信息输出选项

使用 text 选项会输出 DSA 密钥各项参数的明文解析信息，包括 DSA 密钥的三个基本参数、公钥参数和私钥参数。

使用 modulus 选项会输出 DSA 公钥的参数信息。而使用 noout 选项则指令不会输出编码的密钥数据。

**(9)** 应用实例

进行一下格式转换，将一个 PEM 编码的 DSA 私钥转换成一个 DER 编码的 DSA 私钥：

```
OpenSSL> dsa -in dsakey.pem -passin pass:111111 -des-ede3-cbc -out dsakey.der
-outform d -passout pass:111111
```

更换 DSA 私钥的保护口令：

```
OpenSSL> dsa -in dsakey.pem -passin pass:111111 -des-ede3-cbc -out dsakeyn.pem -passout pass:123456
```

从一个 DSA 私钥得到其相应的 DSA 公钥，发给那个需要验证签名的文件的人：

```
OpenSSL>dsa -in dsakey.pem -passin pass:111111 -out dsapubkey.pem -pubout
```

输入一个 DSA 公钥文件，并查看其明文解析信息：

```
OpenSSL>dsa -in dsapubkey.pem -pubin -noout -text
```

使用 Engine 设备的 3DES 算法 ECB 模式加密一个 DSA 私钥，并同时输出其公钥参数的解析信息：

```
OpenSSL> dsa -in dsakey.pem -passin pass:111111 -modulus -engine pkcs11 -des-ede3 -out dsakeyn.pem -passout pass:123456
```

## 8.5 本章小结

本章介绍了 OpenSSL 支持的三种非对称加密算法的性质和其相关指令的使用，为读者进入 OpenSSL 指令丰富的领域打开了又一扇门。

本章首先介绍了使用最为广泛的 RSA 密钥算法的密钥生成、管理和使用方法及其应该注意的细节。

接着本章介绍了最早的公开密钥算法 DH 算法，DH 算法是一种专门用于密钥交换的算法，本章对其密钥参数生成和密钥参数管理需要注意的问题和方法作了介绍。

最后本章介绍了一种专门用于数字签名的非对称加密算法 DSA，DSA 是美国国家标准数字签名算法，本章介绍了其基本性质和使用方法、DSA 密钥参数和 DSA 密钥的生成和管理等。

通过本章的学习，应该对非对称加密算法的性质和使用模型有一个清晰的了解，并对这三种常用的非对称加密算法 RSA、DH 和 DSA 的性质有深入的理解。

# 第 9 章

# 信息摘要和数字签名指令

## 9.1 信息摘要算法和数字签名

信息摘要算法是现代密码学算法中不可缺少的一部分，与对称加密算法和非对称加密算法不同，它不是一种可逆的操作，经过它进行处理的数据，输出数据长度一般来说总是固定的，并且理论上很难从输出恢复出输入。信息摘要算法还具有对输入数据变化非常敏感的性质，也就是说，即便输入数据变化非常小，比如只有一个标点符号，那么经过信息摘要算法得到的输出数据的变化表现是明显的。信息摘要算法的另外一个特点是从不同输入得到相同输出的概率非常低，这样可以让在不同输入的情况下为了得到相同输出而进行的攻击难度很大。基于上述这些性质，信息摘要算法一般被用于保证数据完整性。简单地说，就是在原来数据量相对比较大的文件跟一段固定长度、数据量比较小的数据（通常称为摘要信息）之间建立一种特定的几乎是一一对应的联系，然后通过验证这段摘要信息的不变性来保证其对应的大文件的不变性，即完整性。

数字签名操作一般采用非对称加密算法（公开密钥算法），其实质是使用非对称加密算法密钥对的私钥对数据进行加密，而数字签名的验证操作则是使用公钥对数据进行解密操作，然后比较得到的原始文件跟解密得到的文件信息是否一致，如果一致，则认为数字签名有效，从而确认文件的有效性。上述的流程对于一个小文件来说不会有太大的问题，但是对于一个非常大的文件，比如一本书稿来说，几乎就是难以忍受的，因为非对称加密算法加密和解密的速度很慢，你必须花费大量的时间等待签名和验证过程的完成。

信息摘要算法跟非对称加密算法的结合可以解决上述的难题，一个实用的数字签名操作流程如下：

① 对要签名的原始文件 File 做信息摘要操作得到摘要信息 $M_F$；

② 使用私钥对 $M_F$ 进行加密得到 $S_F$；

③ $S_F$ 就是原始文件的签名信息，可跟文件一起保存或者发送给接收人。

对上述数字签名的验证流程如下：

① 验证者接收到 File 和 $S_F$ 后，首先对文件 File 采用相同的信息摘要算法得到摘要信息 $M_{Fn}$；

② 使用公钥对 $S_F$ 解密得到 $M_{Fo}$；

③ 比较 $M_{Fo}$和 $M_{Fn}$，如果相同，则验证成功，证书文件 File 没有更改，并且数字签名 $S_F$ 有效。

上述签名和验证过程中，只需要使用非对称加密算法对简短的摘要信息进行加密和解密操作，相比于原始的文件（假设原始文件很大）数据量大大缩小，从而避免了对大量数据进行私钥加密和公钥解密的操作，同时又能通过信息摘要算法保证数据的完整性。

OpenSSL 目前支持的信息摘要算法包括 SHA1、SHA、MD5、MD4、MD2、MDC2 和 RIPEMD160。OpenSSL 对所有这些信息摘要算法都提供了单独的同名指令，此外，还使用了统一的指令 dgst 集成了所有支持的信息摘要算法的操作。

因为信息摘要算法通常跟数字签名一起使用，所以，在所有的信息摘要算法指令中，OpenSSL 还支持使用 RSA 或者 DSA 密钥对摘要信息进行数字签名和验证过程，也就是说，所有这些信息摘要指令都能够完成上面介绍的数字签名的流程和数字签名验证的流程，这对于实际的应用具有重大的意义。表 9-1 列出了 OpenSSL 支持的信息摘要指令。

**表 9-1 OpenSSL 信息摘要指令**

| 指 令 | 指令功能描述 |
|---|---|
| sha | SHA 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| sha1 | SHA1 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| md5 | MD5 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| md4 | MD4 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| md2 | MD2 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| mdc2 | MDC2 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| rmd160 | RIPEMD160 信息摘要算法指令，包括使用 RSA 密钥进行数字签名和验证的功能 |
| dgst | 集成了所有信息摘要算法指令的功能，也包括使用 RSA 密钥结合各种信息摘要算法进行数字签名和验证的功能。此外，dgst 算法还支持 DSA 数字签名和验证操作，如果使用 DSA 数字签名，那么就必须将信息摘要算法选项设置为 dss1 |

dgst 指令的格式和其他信息摘要算法指令的格式基本上是保持一致的，下面是使用 sha1 算法进行信息摘要操作的两个指令：

```
OpenSSL>sha1 -c -hex pln.txt
OpenSSL>dgst -sha1 -c -hex pln.txt
```

如果你执行了上述两条指令，会发现它们的输出结果是一样的，因为它们采用了相同的信息摘要算法 SHA1。仔细比较还会发现，这两个指令除了 dgst 多了一个选择算法的选项外，其他选项也是一致的。dgst 指令跟其他单独的信息摘要指令的不同之处还在于它支持使用 DSA 密钥进行数字签名和验证操作。

因为 dgst 和其他各种信息摘要算法的指令格式基本一致，所以在本章的介绍中，将以 dgst 指令为主线，在必要的时候给出其他信息摘要算法指令的说明。

## 9.2 指 令 格 式

前面已经说过，dgst 指令格式跟其他信息摘要算法指令格式非常相似，下面是 dgst 指令的格式：

OpenSS＞ dgst [-dgst__cipher] [-c] [-d] [-hex] [-binary] [-out filename] [-sign filename] [-verify filename] [-prverify filename] [-keyform fmt] [-engine id] [-rand file(s)] [-signature filename] [file…]

而其他信息摘要算法指令通用的格式如下：

OpenSS＞ dgst__cipher [-c] [-d] [-hex] [-binary] [-out filename] [-sign filename] [-verify filename] [-prverify filename] [-keyform fmt] [-engine id] [-rand file(s)] [-signature filename] [file…]

上述指令中，dgst_cipher 是 OpenSSL 支持的各种信息摘要算法的名称，跟 dgst 指令中的 dgst_cipher 选项是基本一致的，但是在单独指令中，不支持 dss1 指令的操作，其具体名称如表 9-2 所示。

根据表 9-2，我们如果要使用 MD5 信息摘要算法对一个文件进行信息摘要操作，则下面两个指令是等价的：

OpenSSL＞md5 -c -hex pln. txt

OpenSSL＞dgst -md5 -c -hex pln. txt

但是如果要使用 DSS 规定的信息摘要算法，则只能使用指令：

OpenSSL＞dgst -dss1 -c -hex pln. txt

而下面的指令则是无效的：

OpenSSL＞dss1 -c -hex pln. txt

表 **9-2**　**dgst** 指令中的信息摘要选项算法类型参数名称

| 算法类型参数 | 同名单独指令 | 简要说明 |
|---|---|---|
| sha | sha | SHA 信息摘要算法，摘要信息位数是 160 位 |
| sha1 | sha1 | SHA1 信息摘要算法，摘要信息位数是 160 位 |
| md5 | md5 | MD5 信息摘要算法，摘要信息位数是 128 位 |
| md4 | md4 | MD4 信息摘要算法，摘要信息位数是 128 位 |
| md2 | md2 | MD2 信息摘要算法，摘要信息位数是 128 位 |
| mdc2 | mdc2 | MDC2 信息摘要算法，摘要信息位数是 128 位 |
| rmd160 | rmd160 | 160 位 RIPEMD 信息摘要算法，摘要信息位数是 160 位 |
| ripemd160 | | |
| ripemd | | |
| dss1 | 无 | DSS 签名标准指定的信息摘要算法，通常跟 DSA 签名算法一起使用。摘要信息位数是 160 位 |

表 9-2 中，算法类型参数字段的值不区分大小写，比如，sha 和 SHA 的效果是一样的，DSS1 和 dss1 的效果也是一样的。

从表 9-2 还可以看出，摘要信息的位数一般都远远小于我们常用的公开密钥算法的密钥位数（1 024），所以对摘要信息的签名可以一次完成，与直接对原始文件签名相比大大提高了签名速度和验证速度。

## 9.3 指令选项说明

因为 dgst 指令跟单独的信息摘要算法指令的选项基本相同，所以本节介绍的选项基本上是通用的，具体不同的地方将会作出说明。

### 9.3.1 信息摘要算法选项

对于 dgst 指令来说，可采用的信息摘要算法有 8 种，具体参数可以参考表 9-2。对于单独的信息摘要算法来说，该选项是不存在的，因为其指令本身就已经限定了使用的信息摘要算法。

### 9.3.2 输出文件选项 out

输出文件选项 out 指定了输出信息的文件，输出的信息包括摘要信息、验证成功与否的信息及数字签名信息等。输出信息的格式根据使用的输出格式选项的不一样而不同。默认情况下使用的输出文件是标准输出设备，一般是当前指令行界面。

### 9.3.3 输入文件选项 files

该选项一般放在各个选项的最后，直接输入要进行信息摘要操作、数字签名操作或者验证签名操作的文件即可。dgst 指令和其他单独的信息摘要指令在这里可以同时对多个文件进行信息摘要操作，但是对于数字签名和签名验证操作则只能每次针对一个文件，否则就可能导致验证失败。需要注意的是，无论是进行信息摘要操作、数字签名操作还是验证签名操作，这里输入的都应该是原来的明文文件。在信息摘要操作和数字签名操作中，该文件信息被用来作为信息摘要函数的输入，得到的摘要信息被保存或者进一步经过私钥签名后保存；而在签名验证过程中，该文件信息也被作为信息摘要函数的输入，其结果跟用公钥解密的数字签名信息进行对比，如果一致，则验证通过，否则验证就失败。

### 9.3.4 数字签名选项

dgst 指令和其他单独的信息摘要指令仅提供了一个跟签名有关的选项 sign，该选项的参数存储了一个 RSA 或者 DSA 私钥的文件。如果是 RSA 私钥，那么说明将要使用的数字签名算法是 RSA 算法，如果使用的是 DSA 私钥，那么说明将要使用的算法是 DSA 数字签名算法。DSA 数字签名算法只能在 dgst 指令中使用，并且要同时使用 dss1 作为算法选项参数。输入私钥的编码格式由 keyform 选项指定，其格式可以是多种多样的。

### 9.3.5 数字签名验证选项

dgst 指令和其他单独信息摘要指令提供了三个跟数字签名验证有关的选项 signature、verify 和 prverify。

signatrue 指定了保存要进行验证的签名信息的文件，通常来说，为了正确验证，应该是一个二进制编码的文件，十六进制编码的文件可能导致不能正确验证。这需要在进行数字签名的时候不使用 hex 选项。

verify 选项跟 prverify 选项不能同时使用。使用 verify 选项表示将要输入的用于验证数字签名的密钥是一个公钥；如果使用 prverify 选项，则表示将要输入的文件保存的是一个私钥，指令将从这个私钥中读取其公钥参数进行数字签名的验证，之所以能够这么做，是因为私钥结构里面通常保存了所用公钥参数。输入密钥的编码格式由 keyform 选项决定。

### 9.3.6　输入密钥格式选项 keyform

keyform 选项告诉指令输入用于签名或者验证的密钥属于什么编码格式。目前来说，如果输入的是私钥（使用 sign 选项签名或者 prverify 选项验证的时候），支持的格式包括 DER 编码格式、PEM 编码格式、PKCS＃12 编码格式、Netscape 编码格式、旧版的 IIS SGC 编码格式及 ENGINE 密钥格式。如果输入的是公钥（使用 verify 选项验证的时候），则比私钥少支持一种 PKCS＃12 格式，这是因为 PKCS＃12 格式一般用来封装证书和私钥，而不直接用于封装公钥。keyform 后面跟随的参数和具体意义可以参考表 8-2。如果 keyform 指定的密钥格式是 ENGINE 格式（-keyform e），那么输入的密钥文件的内容和意义需要根据具体 Engine 接口而定，可能只是一个特定的字符串 ID，可能是一个公钥，也可能是毫无用处的内容。

### 9.3.7　engine 选项

engine 选项指定了替代 OpenSSL 默认算法库的 Engine 设备。对 dgst 指令和其他信息摘要算法指令来说，使用 Engine 选项可能会对几个方面产生影响。首先是信息摘要算法，如果指定的 Engine 接口有效并且支持该选定的信息摘要算法，那么 Engine 设备里面的信息摘要算法就会被启用。其次是数字签名和验证算法，如果指定 Engine 设备有效并且支持使用的数字签名和验证算法，那么 Engine 设备的数字签名和验证算法就会被启用。第三个是随机数产生及相关的操作，如果 Engine 设备支持随机数产生和其他相关操作，那么这些 Engine 操作也会被启用以替代 OpenSSL 本身的函数。

### 9.3.8　输出格式选项

为了看起来方便或者其他原因，可能需要对输出的摘要信息和数字签名信息做一些格式调整，比如将二进制格式转换成十六进制等。OpenSSL 提供了一些此类选项。

hex 选项可以将输出的摘要信息或者签名信息进行十六进制编码后输出，但是验证的时候可能导致验证失败。

c 选项只有跟 hex 选项一起使用才有效，它在每个十六进制数据编码之间增加一个分隔符“:”，这样便于查看。默认的摘要信息和签名信息是以二进制的形式输出的，如果要显式表示使用二进制输出，可以使用 binary 选项。

d 选项使用后指令会将 BIO 读取文件的操作信息打印出来，一般是为了调试的目的才会使用这个选项。

### 9.3.9　随机数文件选项

在 dgst 或其他信息摘要指令操作的过程中，有时候需要用到随机数，那么同样需要

一个随机数种子文件，这可以通过 rand 选项指定。事实上，在所有指令的 rand 选项中，指定的随机数文件可以不止一个，而可以是多个，文件之间根据应用系统的不同采用不同的符号进行连接。比如 Windows 系统可以使用“;”作为连接符号，OpenVMS 系统采用“,”作为连接符号，而其他系统则采用“:”作为连接符号。例如下面是 Windows 系统中使用多个随机数文件的形式：

```
OpenSSL>dgst -md5 -rand file1;file2;file3 pln.txt
```

如果不使用 rand 选项，指令会从其他可用资源获取随机数种子。

## 9.4 使用信息摘要指令进行数字签名和验证

数字签名和其验证是最经常使用的密码学应用技术之一，OpenSSL 的 dgst 指令提供一个方便的工具用于实现这个目的，本节重点介绍怎么使用这个工具。

### 9.4.1 执行数字签名

**(1)** 数字签名流程概述

如果我们要对一个文件进行数字签名，需要做的事情包括：

① 产生一个 RSA 或者 DSA 密钥对；

② 使用某种信息摘要算法对文件进行信息摘要操作得到摘要信息；

③ 使用 RSA 或 DSA 私钥对摘要信息进行加密完成签名操作；

④ 将 RSA 或 DSA 私钥对应的公钥、文件和签名信息一起保存或者发送给接收方。

在本节我们将使用 OpenSSL 的指令工具完成 RSA 和 DSA 数字签名的整个流程。

**(2)** 执行 **RSA** 数字签名

① 首先生成一个 RSA 密钥，并加密保存，执行以下指令：

```
OpenSSL>genrsa -out rsaprivkey.pem -passout pass:111111 -des3 1024
```

② 从生成的 RSA 私钥中导出一个相应的 RSA 公钥并保存：

```
OpenSSL> rsa -in rsaprivkey.pem -passin pass:111111 -out rsapubkey.pem -pubout
```

③ 虽然理论上信息摘要操作和私钥加密操作是分开执行的，但是 dgst 指令提供了集成在一起的功能，所以我们可以使用一个指令完成。当然，由于采用 RSA 算法进行数字签名操作，我们有多种信息摘要算法可选，目前来说，一般推荐使用 SHA1 算法，那么我们执行以下的指令完成信息摘要和私钥加密的操作：

```
OpenSSL>dgst -sha1 -sign rsaprivkey.pem -out sgn.txt file.doc
```

上面操作中的 file.doc 就是要进行签名的文件。还可以使用下面的指令得到相同的效果：

```
OpenSSL>sha1 -sign rsaprivkey.pem -out sgn.txt file.doc
```

④ 将 sgn.txt、file.doc 和 rsapubkey.pem 一起保存或者发送给接收方，完成整个 RSA 数字签名流程。

**(3)** 执行 **DSA** 数字签名

① 首先生成一组 DSA 密钥参数，可以用来生成以后可能需要的许多 DSA 密钥，执行下面指令：

OpenSSL>dsaparam -out dsaparam.pem 1024

② 根据这组 DSA 密钥参数生成一个 DSA 私钥，并加密保存在文件中：

OpenSSL>gendsa -out dsaprivkey.pem -des3 -passout pass:111111 dsaparam.pem

③ 根据 DSA 私钥导出 DSA 公钥并保存，执行下面指令：

OpenSSL>dsa -in dsaprivkey.pem -passin pass:111111 -out dsapubkey.pem -pubout

④ 接着使用 dgst 指令对文件进行信息摘要操作和数字签名操作，由于没有单独的支持 DSA 算法的信息摘要指令，我们别无选择，只能使用 dgst 指令，并且，指令的信息摘要算法选项只能使用 dss1 参数：

OpenSSL>dgst -dss1 -sign dsaprivkey.pem -out sgn.txt file.doc

⑤ 将 sgn.txt、file.doc 和 dsapubkey.pem 一起保存或者发送给接收方，完成整个 DSA 数字签名流程。

### 9.4.2　验证数字签名

**(1)** 验证数字签名流程概述

在验证一个文件的数字签名以前，我们通常拥有以下信息：

- 原始文件 file.doc；
- 文件数字签名信息 sgn.txt；
- 签名私钥对应的公钥。

验证一个文件数字签名的流程通常如下：

① 对原始文件采用跟签名时相同的信息摘要算法对原始文件 file.doc 做信息摘要操作得到摘要信息 M1；

② 使用公钥解密 sgn.txt 里面的签名信息得到解密后的数据 M2；

③ 对比 M1 和 M2，如果一致则签名验证通过，可以确保两点，即原始文件没有被更改和数字签名有效。

**(2)** 验证 **RSA** 数字签名

相比于数字签名过程需要大量的前期准备工作，验证 RSA 数字签名则显得简单得多，只需要一个指令便可见分晓，当然，在进行验证之前，你首先必须确定数字签名过程中使用的是什么信息摘要算法，并在指令中明确指定：

OpenSSL>dgst -sha1 -verify rsapubkey.pem -signature sgn.txt file.doc

如果验证通过，那么会给出成功的提示信息，否则就给出签名验证失败的提示信息。如果你知道签名使用的信息摘要算法，比如，你知道签名算法使用的是 SHA1，那么可以使用下面的指令：

OpenSSL>sha1 -verify rsapubkey.pem -signature sgn.txt file.doc

**(3)** 验证 **DSA** 数字签名

DSA 数字签名的验证只能使用 dgst 指令，这是因为没有单独支持 DSA 算法的信息摘要指令，执行的指令如下：

OpenSSL>dgst -dss1 -verify dsapubkey.pem -signature sgn.txt file.doc

验证如果成功，将会给出签名验证成功的提示信息。

## 9.5 信息摘要指令应用实例

这里给出一些使用 dgst 和其他信息摘要指令的例子，由于对数字签名和验证已经在 9.4 节给出了比较丰富的例子，所以本节将不再进一步给出相同的例子。

使用 RIPEMD160 生成一个文件的摘要信息，并保存在文件 sgn.txt 中：

```
OpenSSL>dgst -rmd -out sgn.txt pln.txt
```

或者使用下面的指令：

```
OpenSSL>rmd160 -out sgn.txt pln.txt
```

是不是觉得二进制的数据实在没有办法看，那么把它转换成十六进制编码的信息摘要数据吧：

```
OpenSSL>dgst -rmd -hex -out sgn.txt pln.txt
```

或者使用下面的指令：

```
OpenSSL>rmd160 -hex -out sgn.txt pln.txt
```

PKCS#11 接口 Engine 提供了 SHA1 算法，可以使用：

```
OpenSSL>dgst -sha1 -engine pkcs11 -out sgn.txt pln.txt
```

或者使用下面的指令：

```
OpenSSL>sha1 -engine pkcs11 -out sgn.txt pln.txt
```

使用一个 PKCS#12 格式的 RSA 私钥来签名一个文件，指令如下：

```
OpenSSL>dgst -sha1 -keyform pkcs12 -sign rsakey.pfx -out sgn.txt pln.txt
```

或者下面的指令也是一样的：

```
OpenSSL>sha1 -keyform pkcs12 -sign rsakey.pfx -out sgn.txt pln.txt
```

使用多个随机数文件能够提供足够多的随机数种子，看看下面指令：

```
OpenSSL>dgst -dss1 -keyform p -sign dsakey.pem -out sgn.txt -rand file1;file2;
file3 pln.txt
```

## 9.6 本章小结

本章对信息摘要算法和数字签名算法作了简单介绍，并重点介绍了 OpenSSL 相关的信息摘要指令和签名指令。

在本章的开头，首先介绍了信息摘要算法的性质及其在数字签名中的应用，并对 OpenSSL 提供的相关指令的功能作了简要的分析。

然后对 dgst 等信息摘要相关指令的格式和功能作了详细的分析介绍，对每个选项都作了必要的说明。

接着本章对如何使用 OpenSSL 的指令完成一个文件的数字签名和验证流程作了具体的分析和演示，让读者对数字签名和验证有进一步感性的认识。

最后本章给出了一些 OpenSSL 信息摘要指令应用的例子。

通过阅读本章，应该对信息摘要算法的性质和应用模型有深刻的理解，并能够掌握数字签名和验证的具体操作流程和应用特点。对 OpenSSL 的 dgst 等指令能够完成的功能有初步的印象。

# 第 10 章

# 证书和 CA 指令

## 10.1 证书和 CA 功能概述

在前面我们已经对数字证书和 PKI 的基本概念作了详细的介绍，但是在本章为了帮助大家更接近现实的应用，对一些概念有必要再作一些进一步的阐述。

### 10.1.1 为什么需要证书?

在第 9 章我们领略了使用 OpenSSL 的公开密钥算法和信息摘要算法指令完成数字签名和签名验证的整个过程。显而易见，我们做到了利用一个公钥验证其相应私钥签名的文件或者数据，但是仅此而已，我们没有办法知道这个私钥的持有人的真实身份，任何人都可以生成一对甚至无数对公开密钥对，密钥对跟特定的实体之间没有任何必然的联系。数字证书正是为了建立实体跟密钥对之间的联系而存在，证书验证中心 CA 充当了确认特定实体跟密钥对之间关系的确认人，并且通过用自己的私钥对这些确认的信息和公钥一起签名来保证其可信性和不可改变性。这里的前提是，CA 是所有用户都信任的。

所以在第 9 章数字签名的流程中，事实上在生成密钥对之后，用户首先需要把自己的信息和公钥一起交给 CA 验证，申请一个属于自己的并得到 CA 和其他用户认可的证书，然后才能正式使用证书相应的私钥对进行数字签名，也只有在这样的前提下，其数字签名才能具有实质上的意义。

### 10.1.2 证书生命周期

#### 1. 证书申请

所谓证书生命周期是指从证书申请到证书被吊销的整个过程，这中间涉及证书申请、颁发、使用和其他各个管理方面的问题。首先面临的问题当然是证书申请。既然证书的主要目的是为了建立特定密钥对和特定实体之间的联系，那么必须先生成一对密钥对，密钥对可以由各种公开密钥算法密钥组成，不过，目前来说，常用的是 RSA 密钥和 DSA 密钥。使用何种公开密钥算法作为证书中的密钥，根据证书的使用目的和具体的技术情况而定，比如，对于用于密钥交换目的的证书，就不能使用 DSA 密钥，而 RSA 密钥则是可能的选择。

生成密钥对可以使用 OpenSSL 的 genrsa 指令和 gendsa 指令，也可以使用本章将要介绍的 req 指令。生成密钥对之后，就需要填写用户自己的信息，并跟自己的公钥一起交

给 CA 进行验证，幸运的话（通常会很幸运），就可以很快拿到签发好的属于自己的证书。但对于具体的操作或者技术来说，用户需要填写哪些信息及怎么格式化这些信息是一个需要考虑的问题，PKCS#10 标准规定了这些相关的格式。其基本思想是用户根据要求填写这些信息，然后将这些信息和用户产生的公钥一起用相应的私钥签名发给 CA 验证，这可以使得 CA 确认这些信息是申请的用户自己发送的，因为 CA 可以把证书请求中的公钥拿出来验证这个证书请求的签名。OpenSSL 提供了 req 指令帮助我们填写和生成这个格式复杂的证书请求，当然，根据不同 CA 的要求，你可能需要更改 OpenSSL 的配置文件（默认是 openssl.cnf）中证书请求相关的部分配置。完成这些工作，你就可以把证书请求交给 CA，休息几分钟等待 CA 签发你的证书（当然，很多 CA 不让你有休息的机会，比如 OpenSSL 的 CA 指令，执行操作只要几秒到几百毫秒）。

2. 证书颁发

首先看看证书颁发要考虑什么问题，第一步就是验证证书请求上的签名是否正确，也就是说，确保公钥对应的私钥就在申请者手中并且申请信息是正确的没有被更改的。然后就要仔细审查证书请求里的用户信息，这工作一般由 CA 管理人员完成，管理人员应该依据管理规范（一个正规的 CA 运营商应该具备的基本条件）对这些用户信息进行审查和核实，必要的时候甚至需要亲自查访用户等。

CA 可能对某些信息还有特殊要求，比如要求国家名字必须跟 CA 本身设定的一样，或者要求省份跟 CA 设置的一样，这些要求通常不需要人工审查，计算机程序可以比我们做得更好。OpenSSL 通过设置 OpenSSL 配置文件（默认是 openssl.cnf）的匹配策略来对这些字段提出不同的要求，这种策略通常有三种：匹配、支持和可选。匹配也就是要求申请填写的信息跟 CA 设置信息必须一致，支持是指必须填写这项申请信息，可选就灵活多了，可有可无。

如果通过了所有这些严格的审查措施，那么 CA 就可以给你签发证书了，如果 CA 是建立在 OpenSSL 指令的基础上，那么其使用的签发指令可能是 ca，也可能是 x509，但更可能是 ca 指令，因为它本身就是 OpenSSL 的一个模拟 CA。

3. 证书验证

有了一个证书后你就可以在数字世界中必要的时候随时出示这个证书证明你的身份。但是拿到你证书的人当然不会看一眼就信任你的证书，它需要仔细验证你的证书，需要验证的信息包括：CA 的数字签名、证书的有效期、证书是否被吊销及其他一些可能的限制选项。

在证书验证过程中出现的一个问题是，这个需要验证你的证书的用户是否具备验证你的证书的能力？让我们首先来看看现实世界中使用一个身份证的情况，当你向银行出示你的身份证的时候，营业员可能只要简单看看证件的盖章和其他防伪标记就通过你的验证了。但是某些时候，比如签订一份非常重要的合同的时候，对方可能为了确认你的身份证的真伪，亲自拿到发证机关进行验证。这是两种不同的验证模式，数字世界中也是一样的。

如果用户自己具备验证能力和验证所需要的材料，就可以自己验证，比如使用 OpenSSL 的 verify 程序完成验证过程。如果用户自己没有能力或者没有足够的资料进行

验证，那么可以到 CA 或者 CA 指定的机构验证证书，在线证书服务协议（OCSP）规定了相应的规则，OpenSSL 也提供了 ocsp 指令完成相应的功能。

需要说明的是，上面的流程只是完成了对一个数字证书合法性的验证，而并没有验证声称拥有这个证书人的真实身份。要进行这样的验证，需要具体的协议细节，但不管什么协议，验证一个实体是否确实是其生成的证书对应的用户都是基于证书内容里面包含的公钥和相应的私钥的基础上的。

4．证书吊销

在某些情况下，比如一个公司员工离开了原来的公司，那么他所拥有的证书也应该及时被撤销，以防止该员工可能利用原有的权限获取非法的信息。证书撤销的操作从技术上来说包括两个方面：一是从 CA 的证书数据库中删除被吊销的证书；二是对外公布被撤销的证书信息，如序列号等，具体来说就是生成和公布证书吊销列表 CRL。如果你使用的是 OpenSSL 的 ca 指令作为 CA 中心服务程序，那么可以使用 ca 指令的 revoke 选项和 gencrl 选项实现这些证书吊销的操作。

5．证书过期

证书的使用有一定的期限，这是为了确保证书安全性和有效性的需要。证书过期后，CA 需要更新证书库中已经过期的证书的状态，比如对过期证书归档和将其从有效证书库中删除等操作。如果使用 OpenSSL 的 ca 指令，那么需要使用 updatedb 选项更新证书库中证书的标记状态。

### 10.1.3　证书的封装类型

1．X.509 证书

X.509 证书包含的内容主要是用户信息、证书序列号、签发者、有效期、公钥、其他信息及 CA 的数字签名，这些在公钥基础设施部分已经作了介绍。我们需要确认的是，X.509 只包含了一个公钥，而没有这个公钥对应的私钥的任何信息。公钥是可以公开的，所以 X.509 证书一般也是随意公开的，任何人都可以获取你的 X.509 证书，然后使用它来跟你通信。比如在 SSL 协议中，在服务器要跟你建立安全信道的时候，就会给你发送它自己的 X.509 证书，以便证明自己的身份，这是必要的，也是安全的，它不用担心你会利用它的 X.509 冒充它欺骗其他用户，因为你没有 X.509 证书里面公钥相应的私钥，达不到这样的目的。X.509 证书适用于一般的证书应用模式。

Windows 平台对 X.509 证书是认可的，如果要在 Windows 平台查看和管理 X.509 证书，需要将 X.509 证书文件后缀名改成 cer、der 或者 crt 之一。OpenSSL 从来不会从文件名称这些表面上的东西判断一个证书的格式，它对 X.509 证书的支持是基于其文件内容的，一般来说，PEM 编码和 DER 编码都是可以被接受的。并且，OpenSSL 本身的 ca 指令颁发的证书就是 X.509 标准格式的。

2．PKCS#12 证书

如果你的证书是使用 OpenSSL 或其他产品申请的，而不是使用微软自身的 KeyStore 生成密钥对并在线申请的，那么要在 IE 或 IIS 服务器上使用该证书，你就得想办法把证书和相应的私钥关联起来，并让微软的软件认可它们之间的关联关系。这时候你需要使用

的就不仅仅是一个 X.509 证书，而是 X.509 证书和它相应的私钥。

还有另外一种情况。一般来说，证书相应的私钥应该保存在硬件设备中，如 Smart Card 和 USB Key 以确保其安全性，计算机存储总是让人不放心。但是很多时候由于条件限制或者安全性要求不足以让单位的决策者掏出这么一笔钱买这些设备，那么就必须把证书和其相应的私钥保存在内存中，这就对证书和私钥的存放提出了复杂的要求。你可能需要在不同的计算机上使用相同的证书，当然，也需要相应的私钥，如果把它们保存在不同的文件中，有可能因为弄错了多个证书和私钥之间的对应关系而导致没有办法使用，所以最好是把它们保存在一起。

PKCS#12 格式证书就是为了适应上述的这些需求产生的，它将证书和其相应的私钥封装在一起。当然，证书和私钥需要的安全性是不一样的，证书可以公开，所以不需要加密保存；而私钥的安全性非常重要，PKCS#12 采用了 PKCS#8 的私钥封装格式对私钥进行了基于口令的加密，虽然这种安全性很值得怀疑，但是毕竟有了一些保护。

如果你把证书和私钥封装成 PKCS#12 格式的证书，你就可以直接把它导入到微软的平台使用（比如 IE），当然，最好之前把文件后缀名改成 pfx 或者 p12。

OpenSSL 的所有指令几乎都接受 PKCS#12 的格式，并且不计较你用什么文件名或者后缀名。同时，OpenSSL 还给出了将 X.509 证书和其私钥转换成 PKCS#12 格式证书的指令 pkcs12，该指令也能实现反向的功能，即将一个 PKCS#12 的证书转换成一个 X.509 证书和一个私钥。

### 3. PKCS#7 证书

如果你使用的是一个根 CA 颁发的证书，验证不会有太大的问题，你只要简单把你和你的 CA 证书发送给验证方就可以了，并且通常来说，验证方甚至已经有了跟你相同的 CA 证书，验证可能轻而易举完成了。但是如果给你颁发证书的 CA 不是根 CA 情况就显得复杂得多，看看图 10-1 吧，你和你的验证用户可能隶属于不同的 CA，但是这些不同的 CA 可能属于相同的根 CA，那么验证就需要更多的信息，不仅仅需要你的 CA 证书，还需要签发你的 CA 证书的上级 CA 证书，直到上溯到根 CA 证书，也就是说，验证的时候，验证方会需要一个完整的证书链。所谓证书链，就是一个用户证书和一系列与其证书相关的 CA 证书的有序集合。所以，考虑到用户这些需求，为了使证书用户能够正确地使用自己的证书，CA 在给用户颁发证书的时候，不仅仅要给用户发放用户自己的证书，还可能需要把证书链中的所有 CA 证书都给用户。

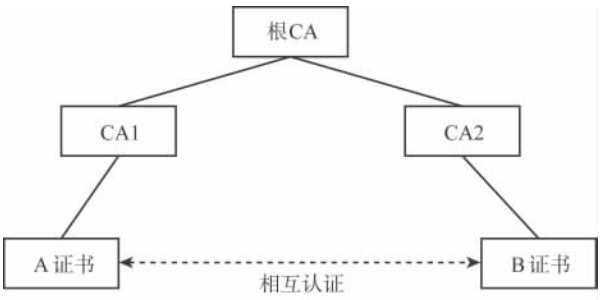


图 10-1 多级 CA 系统及其验证

上面所说的验证过程对于一般情形来说还太理想了，事实上，一个 CA 或多或少会有一些由于各种原因被吊销的证书（除非刚刚建立的 CA），这些证书可能依然在有效期内，用户验证证书通常可能不使用 OCSP 协议（至少该协议不总是可行的），那么为了排除这些已经被吊销的证书，就需要通知用户那些证书已经无效，这一般通过证书吊销列表（CRL）来实现。CRL 是公开的，可以自由下载，但是很多时候验证方可能要求用户给它提供合法的 CRL。

这样一来，用户需要给验证方提供的东西就非常多了：自己的证书、证书链上所有的 CA 证书和 CRL。这么多东西如果一个一个地发送给接收方，他可能会烦死，因为他必须一一鉴别你发送过来的是什么东西，以及考虑怎么保存这些信息。为了避免上述所有这些尴尬的情况，PKCS＃7 标准的证书格式出现了。

PKCS＃7 标准很简单：可以包含多个证书和 CRL。这样就可以解决上述的问题了，在验证的时候，把自己的证书、相关证书链上的 CA 证书和 CRL 封装成一个 PKCS＃7 格式证书发送给验证方就可以了。

微软对 PKCS＃7 证书的支持依赖文件后缀名 p7，OpenSSL 对 PKCS＃7 协议的支持有限，但是提供了将普通用户证书和 CRL 封装成 PKCS＃7 证书的指令 crl2pkcs7，同时也提供了 pkcs7 指令来处理 PKCS＃7 证书的一些相关问题。

表 10-1 列出了上述三种证书的基本情况。事实上，还有其他一些适应不同需要的证书封装格式，但因为使用得不多，这里不再介绍。

表 **10-1**　三种不同封装格式的证书

| 证书类型 | 证书内容 | 适用范围 | OpenSSL 支持 | Windows 后缀 |
|---|---|---|---|---|
| X.509 | 用户信息、公钥、序列号、CA 签名和其他证书信息 | 用户和被验证用户处于同一个 CA 域内的验证模型，用户不需要考虑私钥存储的问题 | 生成、使用和其他管理都支持 | der<br>cer<br>crt |
| PKCS＃12 证书 | X.509 证书和其相应私钥 | 需要同时使用 X.509 证书和其私钥的应用，比如导入证书到 IE 或 Netscape 浏览器 | 生成、使用和其他管理都基本支持 | pfx<br>p12 |
| PKCS＃7 证书 | 多个 X.509 证书和 CRL | 需要同时使用多个用户证书和 CRL 的验证过程或其他应用 | 生成和管理支持 | p7 |

## 10.1.4　使用证书

### 1. 证书应用模型

使用数字证书的目的只有一个，在数字世界建立一个跟现实世界完全相类似的信任模型。数字证书的作用很简单，几乎就是跟现实世界的证书（比如身份证）作用一一对应的。现实世界中的身份证是怎么使用的呢？我们下面的介绍会将数字世界对应于现实世界的名字在括号中标出。首先，公安局（CA）颁发了你的身份证（数字证书），然后你要别人确认你的身份的时候，你拿出你的身份证，对方如果相信自己具备验证身份证真伪的能

力，那么他就会通过查看公安局盖章（CA 签名）、你的资料信息（用户主体信息）和其他信息来确认身份证是公安局颁发的还是伪造的。当然，如果这些都通过了，他可能还拿出一份公安局公布的无效身份证的资料（CRL）对照一下，看看你的身份证号码（证书序列号）是不是已经名列其中了。如果你是一个守法的人，这样的验证可能很顺利就通过了。

但是也有可能用户觉得事关重大，根本就不相信自己现有的能力和资料能够确保身份证的合法性，那么他就需要把身份证拿到公安局或者公安局指定的某个代理机构（OCSP）确认身份证的真伪，该机构显然拥有更丰富的经验和信息来确认身份证的真伪。

上面介绍的两种验证模型，也是数字世界的两种验证模型，一种是需要 CA 或其指定机构参与的，一种是不需要 CA 参与的，如图 10-2 所示。

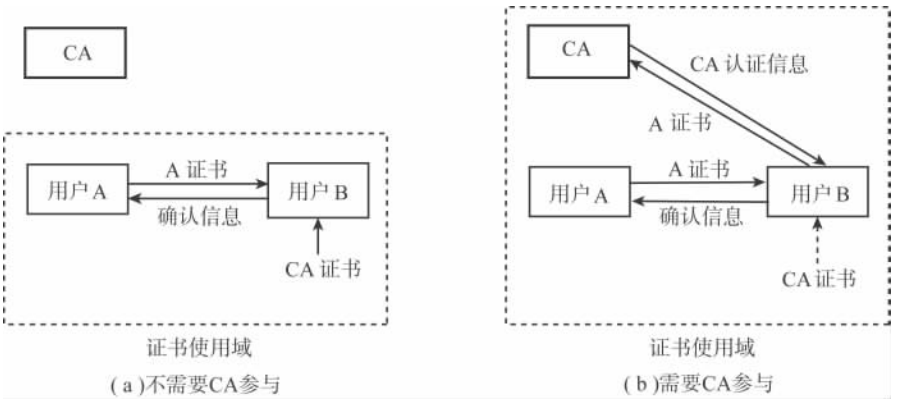


图 10-2　两种不同的证书验证模型

对于不需要 CA 参与的证书应用模型，验证用户 B 应该具备 CA 证书（或证书链）、CRL 这些基本的材料和验证应用程序。对于需要 CA 参与的证书应用模型，用户 B 几乎可以不拥有任何资料，它只要可以理解 CA 提供的在线验证协议就行了，现在通常使用的是 OCSP 协议。

2. 证书链

我们回头看看图 10-1，用户 A 和用户 B 的证书是不同 CA 签发的，分别是 CA1 和 CA2，那么如果 A 发给用户 B 自己的证书并要求进行验证，那么 B 应该怎么做呢？如果 B 没有 CA1 的证书，那么 B 就没有办法确认用户 A 证书中签名的合法性，验证过程显然没有办法进行下去。同样，要验证 CA1 的证书，B 也需要根 CA 的证书，以便提取其中的公钥验证其对 CA1 的签名。所以，为了确保 B 能正确验证自己的证书，A 必须给 B 发送三个证书：自己的证书、CA1 证书和根 CA 证书。这三个证书就构成了一个完整的证书链。

B 接收到 A 发送过来的整个证书链之后，B 进行验证的整个操作流程如下：

① 从 CA1 提取公钥，验证用户证书 A 的合法性和有效性；

② 从根 CA 提取公钥，验证 CA1 证书的合法性和有效性；

③ 利用根 CA 的公钥验证根 CA 证书的合法性和有效性（因为它是一个自签名根证书）；

④ B 查找自己的信任证书库（通常是一些根 CA 的列表），看看上述通过验证的根 CA 是否在信任列表中，如果在，那么验证通过，否则验证不通过。

一般来说，证书应用程序都应该支持证书链的构造和验证，用户只要将构造证书链需要的证书提供给应用程序就可以了。比如 OpenSSL 的指令 verify，你只需要将构建一个完整证书链需要的证书都存放在一个文件（CAfile）或者一个目录（CApath）中，并把这个相关的信息通过指令选项输入，verify 指令就会自动完成整个证书链的构造和验证过程。

3. 用户身份确认

上面我们已经顺利通过一个证书的验证了，实质上，我们仅仅是对证书本身的合法性和有效性进行了验证，对于用户是否确实拥有这个证书还需要进一步确认。

需要再次强调的是，仅有证书是没有多大意义的。千万不要以为自己得到了管理员的一个数字证书就可以窃喜不已，事实上它毫无用处，在实际的应用协议中，在通过对你出示的证书进行验证之后，还要验证你是否确实拥有该证书。过程很简单，提取证书中的公钥，使用该公钥加密一个随机数，然后发送给你，并要求你解密后返回该随机数。如果你确实是证书的主人，那么你拥有相应的私钥，你可以解密发送过来的信息并得到随机数，然后返回给对方证明你的身份，对方比较本地保存的随机数和你返回的随机数，如果一致，证明你能够正确解密证书上公钥加密的数据，确认你拥有证书相应的私钥。而你如果是冒充的用户，那么这时候你就束手无策了，你没有办法正确解密得到那个随机数，从而也就狼狈地暴露了你的身份。

4. 更多问题

虽然我们有了一个合法的数字证书，但证书在具体应用环境或者协议中的使用要考虑的问题还可能有许多。首先面临的就是证书格式问题，对于具体的应用程序来说，对证书的编码格式可能有不同的限制和支持，比如一个使用非微软本身密钥产生器申请的证书需要在 IE 中使用，首先就必须将证书和密钥封装成 PKCS＃12 格式才能导入到微软的机制中，而 OpenSSL 则不存在这样的限制。

其次可能还有证书本身的一些扩展字段问题，有些 CA 中心自己扩展了一些字段，但是这些字段在别的应用程序中并不一定能得到很好的支持，这样有可能就会导致验证失败等问题，尤其是该扩展字段标记为关键扩展之后。

这样的问题要是啰嗦起来，数不胜数，如果想在碰到这些问题的时候能够迅速解决，那么最主要的就是对整个密码学技术和 PKI 应用有一个清晰的概念和认识。

### 10.1.5　CA 的建立

建立一个投入实际应用的 CA 是一个非常复杂的问题，要考虑的不仅仅是技术上的问题，更多的是管理上的规范性问题。对于管理问题，不是本书关注的话题，不作更多的讨论。这里要说的是，怎么从技术上建立一个 CA 服务器。

前面我们介绍了证书的各个方面，都是假设 CA 已经存在，但是事实上，可能你并不能找到一个合适的 CA，甚至，你的任务就是建立一个 CA。这时候该做些什么？

事实上你有很多选择。微软平台自身就附带了一个可选的 CA 服务器程序，该程序能

够实现申请证书和发放证书等功能。此外，OpenCA 也是一个选择。你还可以选择一个营利性的公共 CA 来建立你的证书服务体系。不过无论如何，在本书我们要讨论的重点可能是 OpenSSL 提供的 ca 指令。我并非推荐大家使用 OpenSSL 的 ca 指令，但它确实有很多值得我们使用和研究的地方。首先它是开放源代码的，这非常有好处，也适应国家对网络安全产品的要求；其次它是免费获得的，任何人都可以拥有，不存在太多的版权限制；再次，其功能也还不错，通过几次的改进，其功能已经远远超出了当时作为一个演示 ca 程序的范畴。

无论你使用哪种 CA 应用程序，要使一个 CA 服务器开始运作，首先要为 CA 自身生成一对密钥和一个证书。因为 CA 的证书安全性非常重要，所以其密钥的长度应该比普通的证书密钥长度长一些，比如采用 2 048 位的 RSA 密钥。如果 CA 是一个根 CA，那么其证书就是一个自签名证书，即使用自己的私钥对证书内容（包括自己的公钥）进行签名。如果不是一个根 CA，那么应该向上级 CA 申请证书。OpenSSL 的 req 指令可以提供生成证书请求和自签名根证书的功能，在后面的章节中将会作详细的介绍。CA 拥有了自己的证书和私钥之后，CA 应用程序一般来说就可以开始运行了。

### 10.1.6 OpenSSL 证书和 CA 指令概览

在本章前面的内容介绍了证书和 CA 的一些内容和操作，看起来非常烦琐和复杂，但是请放心，OpenSSL 提供的指令工具基本上都能解决上述的问题，甚至更多。表 10-2 列出了目前 OpenSSL 支持的与证书和 CA 相关的指令。

**表 10-2 OpenSSL 支持的证书和 CA 指令**

| 指　令 | 功能描述 |
|---|---|
| req | 根据给定的密钥对或者新生成的密钥对按 OpenSSL 配置文件指定的要求生成符合 pkcs＃10 的证书请求。此外，还支持生成自签名根证书的功能 |
| ca. pl | 这是一个 Perl 脚本指令，需要 Perl 的支持。它使用简单的参数形式封装了 ca，verify 和 pkcs12 等指令的一些功能，并支持创建一个适用于 ca 指令运行的目录结构和环境 |
| ca | 该指令模拟了一个 ca 服务器的功能和操作，包括签发证书、吊销证书、产生 CRL 等证书相关管理操作 |
| x509 | 该指令是一个显示 X. 509 证书内容和签发证书的工具，具备了使用特定私钥和证书进行证书签发的功能 |
| verify | 验证证书的程序 |
| ocsp | 支持在线证书验证协议的指令，可以处理 OCSP 协议的一些标准格式信息操作，生成 OCSP 请求发送给其他 OCSP 服务器，甚至可以自己模拟一个 OCSP 服务程序 |
| crl | 处理和显示 CRL 文件信息 |
| crl2pkcs7 | 将 CRL 和其他证书封装成 PKCS＃7 证书 |
| pkcs12 | 将 X. 509 证书和其相应私钥封装成 PKCS＃12 证书，或者将 PKCS＃12 证书转换成 X. 509 证书 |
| pkcs7 | 处理 PKCS＃7 的证书，并可以将之转换成普通格式的证书 |

在本章后面的部分，将对这些指令的使用作详细的介绍。事实上，OpenSSL 还提供了其他一些跟证书相关的指令，但是由于使用不多，我们将在本书其他章节介绍。

## 10.2　申 请 证 书

申请证书包含了许多步骤，如生成密钥对、填写用户信息、签名，等等。此外，根据申请证书类型的不同，还需要对不同的字段做不同的限制，这些都使得申请证书不像想像起来那么简单，但无论如何，本节还是试图让你学会怎么用 req 指令尽量生成你想要的任意类型证书。整个过程我们都会结合 OpenSSL 的指令进行介绍，但是你学到的应该不仅仅是指令操作本身，更多的应该是对 PKI 应用中一个实际的证书申请流程的本质的体会，从而加深对 PKI 概念的理解。

### 10.2.1　req 指令介绍

#### 1. 功能概述和指令格式

req 指令一般来说应该是提供给证书申请用户的工具，用来生成证书请求以便交给 CA 验证和签发证书。但是，OpenSSL 的 req 指令的功能远比这样的要求强大得多，它不仅可以生成 RSA 密钥、DSA 密钥，以及将它们封装成证书请求，还可以对现有的证书请求进行签名验证、格式转换及信息修改等。下面将通过对 req 指令的各个参数的介绍来了解其全面的功能。首先来看看 req 指令的格式：

```
OpenSSL> req [-inform fmt] [-outform fmt] [-in filename] [-passin arg] [-out filename] [-passout arg] [-text] [-pubkey] [-noout] [-verify] [-modulus] [-new] [-rand file(s)] [-newkey rsa:bits] [-newkey dsa:file] [-nodes] [-key filename] [-keyform fmt] [-keyout filename] [-digest] [-config filename] [-subj arg] [-x509] [-days n] [-set_serial n] [-asn1-kludge|-no-asn1-kludge] [-newhdr] [-extensions section] [-reqexts section] [-utf8] [-nameopt nopt] [-reqopt ropt] [-batch] [-verbose] [-engine id] [-subject]
```

相信这 36 个选项一定让你眼花缭乱，头痛不止。不过，其实不用着急，常用的选项其实也不多，很多选项对于你来说可能永远也不会使用，你只要根据自己的需要选取其中部分即可。req 指令的许多选项跟 OpenSSL 配置文件（默认为 openssl.cnf）的证书请求部分是紧密相关的，如果在指令中有些选项需要指定的参数没有给定，那么 req 指令就会从配置文件中相应的字段去查找设置值。此外，有相当部分的证书请求属性都是由配置文件的相关部分决定的。在本节的介绍中，会指出相应的证书请求配置部分内容。

#### 2. 输入和输出格式选项 inform，outform 和 keyform

相信如果读者按顺序阅读了本书前面的章节，这几个选项的名字应该非常熟悉了。但是因为具体的指令支持的格式不尽相同，谨慎起见，仍然需要作一些简要的介绍。inform 选项指定了选项 in 指定的输入证书请求文件的编码格式，目前来说，支持的编码格式有两种：PEM 编码和 DER 编码，默认为 PEM 编码。in 选项指定的文件输入内容是一个已经存在的符合 PKCS#10 标准的证书请求，在这里输入的目的一般是为了验证和显示其

中的一些内容。

outform 选项指定了输出选项 out 输出的证书请求或自签名证书的编码格式，目前支持的编码格式包括 PEM 编码和 DER 编码，默认为 PEM 编码。需要注意的是，该选项指定的格式不影响 keyout 选项的密钥编码格式和 pubkey 输出的公钥编码格式。

keyform 选项指定了密钥输入选项 key 指定的密钥编码格式，目前来说，可以接受的编码格式包括 PEM 编码格式、DER 编码格式、PKCS＃12 编码格式、Netscape 编码格式、旧版本的 IIS SGC 编码格式及 Engine 格式。各种格式对应的参数名称请参考表 8-2。

3. 输入和输出文件选项 in，out，key 和 keyout

in 选项指定了保存证书请求的文件名，默认是标准输入。如果启用了该选项，那么指令将根据一个已经存在的证书请求生成自签名根证书或者显示其中的一些内容，但是该证书请求的内容基本上是不能进行变更的。如果是要根据现有的证书请求生成自签名根证书（使用 x509 选项），那么要求同时使用 key 选项指定证书请求里面的公钥相应的私钥以便对自签名根证书进行签名。如果使用了 new 选项或者 newkey 选项，那么 in 选项指定的证书请求文件将被忽略。in 选项输入的证书请求的编码可以是 PEM 格式也可以是 DER 格式，由 inform 选项指定。

out 选项指定了输出文件名，默认是标准输出。输出文件可以保存包括证书请求、自签名证书、编码公钥，以及由 text、subject、modulus 等选项指定的明文输出信息。其中，证书请求和自签名证书的输出编码格式可以为 DER 编码或者 PEM 编码，由 outform 选项指定。公钥（如果使用了 pubkey 选项输出）的编码格式固定为 PEM 格式。

key 选项指定了输入私钥的文件，该文件内的私钥编码格式由 keyform 指定。需要注意的是，如果 keyform 指定的编码格式是 ENGINE 格式，那么 key 指定的文件内容要根据具体的 Engine 接口而定，可能是一个编码公钥，也可能是一个字符串 ID，甚至可能是毫无意义的内容。密钥类型可以是 RSA 私钥，也可以是 DSA 私钥。

keyout 选项指定了新生成的私钥的输出保存文件，如果没有指定，那么默认情况下会从 OpenSSL 配置文件中的 req 字段的 default_keyfile 选项的参数中获取输出私钥文件名（openssl. cnf 文件默认值是 privkey. pem）。仅在使用了 newkey 或 new 选项导致生成新密钥对的时候该选项才有效，如果使用了已有的私钥（key 选项指定），那么该选项就会被忽略。该选项输出的私钥编码格式固定为 PEM 编码。输出到 keyout 选项指定文件的私钥一般都会被加密保护，加密的算法是 DES3 的 CBC 模式。

4. 输入和输出口令选项 passin 和 passout

passin 选项指定了读取 key 选项指定的私钥所需要的解密口令，输入口令的方式可以是多样的，具体的使用方法请参考 6. 6 节的详细介绍。如果没有使用该选项，而 key 选项指定的私钥又要求提供解密口令，那么指令会在指令行提示用户输入口令。

passout 选项指定了使用 keyout 选项输出私钥时使用的加密口令，输入口令的方式和具体参数参考本书 6. 6 节。如果使用了 nodes 选项，则表示不对私钥进行加密，那么该选项指定的口令无效。如果该选项没有使用，并且输出了私钥，也没有使用 nodes 选项，那么指令会要求用户从指令行界面输入加密口令。

5. 指令操作类型 new，newkey，nodes 和 x509

new 选项告诉指令执行生成新的证书请求操作，这时候，in 选项指定的输入文件会被忽略。使用 new 选项后，如果没有使用 key 选项指定私钥用于生成证书请求，那么就会根据 newkey 选项的参数生成新的密钥对。默认情况下，如果指令既没有使用 key 选项，也没有使用 newkey 选项，那么指令就会生成一个 RSA 密钥，其长度由 OpenSSL 配置文件中 req 字段的 default_bits 选项的参数决定。

newkey 选项让指令生成一个新的密钥对，该选项只有在没有使用 key 选项的时候才有效。newkey 选项有两项功能：指定生成的密钥类型和密钥长度。newkey 选项的参数有如下两种形式：

```
rsa:numbits
dsa:file
```

其中，rsa 代表生成 RSA 密钥作为证书请求的密钥对，dsa 则表示生成 DSA 密钥对作为证书请求的密钥对。numbits 表示要生成 RSA 密钥的长度，一般来说使用 1 024 位作为普通证书请求是合适的。file 是存储了 DSA 密钥参数的文件名，DSA 密钥的长度由该文件存储的 DSA 参数决定。

nodes 选项告诉指令不对新生成的私钥进行加密保存，这样，passout 选项将被简单忽略。如果没有使用 nodes 选项，并且生成了新的私钥，那么私钥输出到 keyout 指定的文件中时将会被以 DES3 的 CBC 模式加密。

x509 选项让指令生成一个自签名根证书而不是输出一个证书请求。所谓自签名根证书，就是指用一对密钥对的私钥对自己相应的公钥生成的证书请求进行签名而颁发证书，这样，证书申请人和签发人都是同一个，所以称为自签名根证书。如果根证书不是用于根 CA，那么一般来说只有测试上的意义。在 req 指令中，既可以直接生成一个新的自签名根证书，也可以根据现有的证书请求和其相应私钥生成自签名根证书。如果是根据现有证书请求生成自签名根证书，那么一定要 key 选项提供相应的私钥指令才能执行成功。

6. 属性设置选项 digest，subj，days 和 set_serial

digest 选项指定了生成证书请求或者自签名根证书时使用的信息摘要算法，该信息摘要算法一般在生成数字签名的时候使用。其支持的信息摘要算法参数请参考表 9-2。需要注意的是，如果使用的是 RSA 算法，理论上除了 DSS 数字签名标准指定的信息摘要算法（dss1）不能使用以外，其他信息摘要算法都能够用于生成数字签名并且 req 指令也能够正确执行，但是，由于 RIPEMD 算法在标准中没有得到支持，很可能导致某些 CA 和应用程序不能正确使用。如果使用的密钥是 DSA 密钥，那么 digest 的任何设定都会被忽略，事实上，这时 digest 的值就固定为 DSS 规定的信息摘要算法。默认的 digest 算法是 MD5。

subj 选项直接从指令行指定了证书请求的主体名称，如果没有出示该选项，那么 req 指令将会根据配置文件特征名称字段的设定提示用户输入必要的信息，比如国家代号（通常是两个字符）、省份、单位名称，等等，详细参考第 6.1.3 节。如果使用 subj 选项，那么这些提示信息就不会再出现。subj 选项的参数格式如下：

```
/type0 = value0/type1 = value1/type2 = value2……
```

参数应该以“/”开头，每个值直接使用“/”作为分隔符号。“type *”应该是 OpenSSL 内部支持的数据对象名称。比如下面是一个例子：

/C = CN/ST = BeiJing/CN = DragonKing

days 选项设定了生成的自签名根证书的有效期，单位是天。该选项只有在使用了 x509 选项生成自签名根证书的时候才有效。默认值是 30 天。

set_serial 选项指定了生成的自签名根证书的序列号，默认情况下生成的自签名根证书序列号是 0。该选项也只有在生成自签名根证书的时候有效。

7. 配置文件选项 config，extensions 和 reqexts

config 选项指定了 req 指令在生成证书请求的时候使用 OpenSSL 配置文件，该文件的最好例子是 openssl. cnf，一般来说，openssl. cnf 也是默认的 req 指令配置文件。但是，如果在 Windows 环境中使用并且在编译的时候没有指定正确的路径，那么该默认配置文件很可能因为路径不正确而不能不到，这时候使用 config 选项就显得非常必要了。当然，config 选项并非指定配置文件的唯一方法，你还可以使用 OPENSSL_CONF 或者 SSLEAY_CONF环境变量指定默认的配置文件。关于配置文件更详细的说明，请参考第 6 章。

extensions 选项指定了生成自签名根证书的时候使用的扩展字段，其参数为 OpenSSL 配置文件中某个字段名（比如 openssl. cnf 文件中的 v3_ca 字段）。该字段一般规定了一些证书的扩展项信息，比如证书的用途，是否能够作为 CA 证书等性质。

reqexts 选项指定了生成证书请求时使用的扩展字段，该字段参数也是配置文件中的某个字段名，比如 openssl. cnf 文件中的 v3_req 字段，该字段也是证书请求中设置的要求申请的证书的一些使用限制信息。其内容跟证书扩展项内容基本一致。

extensions 选项和 reqexts 选项使得在同一个配置文件中可以存在多个不同的字段用于生成不同用途和目的的证书，比如，申请一个 CA 证书和用户证书的证书请求扩展字段设置是不一样的，那么可以在配置文件中将两种不同的证书请求设置分别写成两个不同的字段，在生成证书请求的时候根据需要使用 reqexts 选项指定不同的字段即可达到不同的目的。

8. 属性格式化选项 asn1-kludge，newhdr 和 utf8

一般情况下，如果证书请求的某个属性没有填写，那么对应的证书请求字段会填写一个空的值。但是某些不完全标准的 CA 不能正确处理这些空值，它们的要求是：如果没有填写任何值，那么该属性字段就不应该出现在证书请求中。对于这些 CA，你可能就需要使用 asn1-kludge 选项实现这个目的。

newhdr 选项使用后将会在输出的 PEM 编码的证书请求开始和结束行增加“NEW”标记字符串，这是为了跟 Netscape 和其他一些证书服务器兼容设置的选项。使用 newhdr 和不用 newhdr 选项的证书请求格式如下。

不使用 newhdr 选项：

—BEGIN CERTIFICATE REQUEST—

—END CERTIFICATE REQUEST—

使用 newhdr 选项：

—BEGIN NEW CERTIFICATE REQUEST—

—END NEW CERTIFICATE REQUEST—

utf8 选项使用之后，将对输入的信息采用 UTF8 编码，而不是默认的 ASCII 编码。UTF8 编码对于中文来说具有很重要的作用。

9. engine 选项

engine 选项指定后，所有指定 Engine 支持的操作都将使用 Engine 指定的加密库或者硬件设备进行，而不再是使用 OpenSSL 默认的算法库。对于 req 指令来说，可能使用 Engine 设备的操作包括 RSA 或者 DSA 密钥产生、签名时使用的信息摘要算法、签名时使用的私钥加密算法、签名验证操作的公钥解密操作、随机数产生和对私钥进行加密的 DES3 算法。如果 engine 选项指定了有效的 Engine 设备，那么指令中任何该 Engine 设备支持的上述操作都会使用 Engine 设备的操作流程而不再使用 OpenSSL 算法库本身提供的函数。

10. 输出内容选项 text，reqopt，pubkey，noout 和 subject

text 选项让指令输出证书请求或者自签名根证书内容的明文解析，默认情况下，它将输出所有可能输出的内容，但是如果使用了 reqopt 选项，其输出内容就取决于 reqopt 选项。

reqopt 选项用于指定 text 选项输出的内容，其参数可以是多个，每个之间使用“,”作为连接符号。reqopt 支持的参数如表 10-3 所示。

表 **10-3**　**reqopt** 选项支持的参数

| 参　数 | 参数意义 |
|---|---|
| compatible | 默认的参数，输出所有内容的明文解析信息 |
| ca_default | 输出内容跟相当于同时使用了 no_header、no_version、no_issuer、no_pubkey、no_sigdump 和 no_signame 参数 |
| no_header | 输出明文信息不使用“Data:”和“Certificate:”前缀 |
| no_version | 不输出版本信息 |
| no_serial | 不输出序列号信息 |
| no_signame | 不输出签名算法名称信息 |
| no_validity | 不输出有效期信息 |
| no_subject | 不输出主体名信息 |
| no_issuer | 不输出签发者信息 |
| no_pubkey | 不输出公钥信息 |
| no_extensions | 不输出证书扩展项信息 |
| no_sigdump | 不输出签名数据 |
| no_aux | 不输出证书信任信息 |
| no_attributes | 不输出扩展属性信息 |
| ext_default | 输出不支持的证书扩展信息 |
| ext_error | 如果不支持扩展信息，则输出错误信息 |
| ext_parse | 对不支持的扩展信息进行 ASN.1 编码的对象解析 |
| ext_dump | 采用十六进制编码输出不支持的扩展信息。 |

对于证书请求来说，上表中的很多内容没有，对于这种情况，指令会自然忽略，相关的参数也就不会产生任何作用。reqopt 选项只有在跟 text 选项一起使用的时候才有效。例如我们要使输出自签名证书的文本信息不包含序列号和公钥信息，则可以使用下面的指令：

```
OpenSSL>req-x509 -newkey rsa:1024 -text -reqopt no_serial,no_pubkey-noout
```

pubkey 选项使用后，指令将会输出 PEM 编码的公钥到 out 选项指定的文件中。默认情况下只输出私钥到 keyout 指定的文件，并不输出公钥。

noout 选项使用后，指令将不会输出编码的证书请求或者自签名根证书到 out 选项指定的文件中，该选项一般用来测试指令或者查看证书请求的信息。

subject 选项告诉指令输出主体名信息，即使已经使用 text 选项输出了主体名信息，该选项的启用也会让指令重复输出一遍主体名信息。

### 11. 输出字符编码选项 nameopt

nameopt 选项指定了如何显示主体名称和签发者名称，主要用于显示名称中不同编码的内容，比如 UTF8 编码等。如果没有使用该选项，那么默认使用的是表 10-4 中的“oneline”参数。该选项可以同时使用多个参数，每个参数之间使用“,”作为连接符。此外，每个参数之前还可以使用“-”表示不使用该参数定义的显示方式。表 10-4 是 nameopt 选项支持的所有参数。

**表 10-4 nameopt 选项支持的参数**

| 参　数 | 参数意义 |
|---|---|
| compat | 使用旧的格式显示，使用该参数与没有使用 nameopt 选项的效果是相同的 |
| RFC 2253 | 使用与 RFC 2253 规定兼容的格式显示主体名称和签发者名称，其效果跟同时使用 esc_2253，esc_ctrl，esc_msb，utf8，dump_der，dump_nostr，dump_unknown，sep_comma_plus，dn_rev 和 sname 相同 |
| oneline | 一种比 RFC 2253 可读性更强的单行显示格式，相当于同时使用 esc_2253，esc_ctrl，esc_msb，utf8，dump_der，dump_nostr，use_quote，sep_comma_plus_spc，spc_eq 和 sname 参数 |
| multiline | 一种多行显示格式，相当于同时使用了 esc_ctrl，esc_msb，sep_multiline，spc_eq，lname 和 align 参数 |
| esc_2253 | 对 RFC 2253 规定的特殊字符（，＋”＜＞；）做特殊的显示处理。除了上述括号中的字符，对于在字符串开头的“＃”，以及在字符串开头和结尾的空格符也需要做特殊显示处理 |
| esc_ctrl | 对控制符进行特殊显示，控制符是指 ASCⅡ值小于 0x20 的字符和删除符（0x7f），对于这些字符，使用“\XX”的方式显示，XX 是字符值十六进制的编码表示 |
| esc_msb | 对 MSB 集中的字符串使用特殊显示方式，MSB 集合是指所有 ASCⅡ值大于 127 的字符集合 |
| use_quote | 对某些需要特殊显示处理的字符使用双引号将这个字符括起来，如果不使用该选项，那么所有要特殊显示的字符都会使用“\” |

续表

| 参　数 | 参数意义 |
| --- | --- |
| utf8 | 该参数告诉指令首先将所有要显示的字符转换成 UTF8 格式。如果你使用的是支持 UTF8 字符的终端，那么使用该选项有可能让你能够正确显示多字节的字符，比如中文。如果没有使该参数，那么所有值大于 127 的多字节字符都会使用“\UXXXX”（16 位）或者“\WXXXXXXXX”（32 位）表示。如果该参数前面使用“-”禁止其功能后，那么所有 UTF8 字符都会被首先转换成自己代表的字符显示 |
| ignore_type | 不输出字符串的类型 |
| show_type | 在字符串开头显示字符串的类型，比如 UTF8 字符串，那么就显示“UTF8STRING:”。这对于调试目的是有一些意义的 |
| dump_der | 该参数使用后，任何需要使用十六进制编码的值域都将被进行 DER 编码然后以十六进制显示。否则的话就直接显示每个字节的内容 |
| dump_nostr | 使用十六进制编码显示非字符类型的字符串，如 OCTET STRING 类型，如果该参数没有设置，那么将会认为每个字节代表一个字符进行显示 |
| dump_all | 对所有域都使用十六进制编码方式进行显示 |
| dump_unknown | 对所有没有定义的 OID 值域使用十六进制编码显示 |
| sep_comma_plus | 显示的时候每个值域之间使用逗号（,）分隔开 |
| sep_comma_plus_space | 显示的时候每个值域之间使用逗号和空格（,）分隔开 |
| sep_semi_plus_space | 显示的时候每个值域之间使用分号和空格（;）分隔开 |
| sep_multiline | 使用多行方式显示，每个值域一行 |
| dn_rev | 对值域显示的顺序进行反向重排 |
| nofname | 不显示值域的名称，比如默认情况下显示为“C=CN”，使用该参数后，显示为“CN” |
| sname | 使用值域简短名称显示 |
| lname | 使用值域的长名称显示 |
| oid | 使用值域的 OID 显示 |
| align | 多行显示的时候进行对齐 |
| spc_eq | 在显示的值域和值之间的等号左右增加一个空格（=） |

需要注意的是，nameopt 选项只影响使用 text 和 subject 选项输出的主体名称和签发者名称的显示方式，对其他项目的显示方式没有影响。上述的功能对于正确显示中文信息来说具有很重要的意义。前面说过，可以同时使用多个参数，下面是一个例子：

```
OpenSSL>req -x509 -newkey rsa:1024 -noout-text -nameopt oneline,-utf8,lname
```

上述指令指定在显示自签名证书的主体名称和签发者名称时，使用单行显示方式，但是禁止在显示之前进行 UTF8 格式的转换，并且域名称使用长名称进行显示。

### 12. 证书请求数字签名验证选项 verify

使用 verify 选项的指令将对证书请求的数字签名进行验证操作，并给出失败或者成功的提示信息。其验证的过程是从证书请求里面提取公钥，然后使用该公钥对证书请求的数字签名进行验证。

13. 其他选项 rand，batch 和 verbose

rand 选项指定了生成密钥对或者其他一些操作需要的随机数种子文件。如果没有使用 rand 选项，指令会从其他可能得到的资源取得随机数种子数据。

使用 batch 选项将不再提示用户输入生成证书请求需要的用户信息，而是直接从 OpenSSL 配置文件特征名称字段读取默认值，如果没有默认值，则不填写该值域。一般来说，在你做测试的时候使用该选项会提高效率。

verbose 选项让指令输出执行的各个操作的详细信息，一般来说，该选项也仅对调试程序有意义。

14. 简单的例子

虽然后面我们还要针对 req 指令的具体应用作一些介绍，但是首先让我们来看几个简单的 req 指令应用例子，建立一个初步的印象。

下面的指令生成一个新的证书请求文件，因为没有私钥，所以也要求指令直接生成一个新的 1 024 位私钥用于该证书请求。证书请求输出到 req. pem 文件中，而私钥则输出到 privkey. pem 中，私钥的加密口令采用“12345678”。指令如下：

```
OpenSSL>req -new -newkey rsa:1024 -keyout privkey.pem -passout pass:12345678
-out req.pem
```

下面的指令生成一个自签名的根证书而不是证书请求：

```
OpenSSL>req -x509 -newkey rsa:1024 -keyout privkey.pem -passout pass:12345678
-out selfsign.cer
```

下面的指令对一个已经签发的证书请求签名进行验证：

```
OpenSSL>req -in req.pem -verify -noout
```

## 10.2.2 生成证书密钥

1. 证书请求中使用 RSA 密钥

RSA 密钥是目前证书中最经常使用的密钥类型，这一部分是因为其既可以支持密钥交换，又可以支持数字签名的特性；更重要的是 RSA 算法是一种经过充分考验和密码分析的算法，其安全性得到了公认。

在使用 req 指令生成证书请求时，我们可以采用两种方法生成 RSA 密钥对：一种是使用 genrsa 指令预先生成密钥对，一种是直接在 req 指令中生成密钥对。从本质上来说，这两种方法没有区别，但是，使用第一种方式能够更灵活地处理密钥的保存和格式等问题。

我们首先看下面的指令，它使用 req 指令直接生成了一个新的 1 024 位 RSA 密钥用于新的证书请求中：

```
OpenSSL> req -new -newkey rsa:1024 -keyout privkey.pem -out req.pem -passout
pass:12345678
```

上述指令对生成的 RSA 私钥进行了加密，加密的算法自己是没有选择余地的，只能是 DES3-CBC 模式，当然，你可以使用 rsa 指令对密钥使用的加密方法和口令进行重新的

更改。此外，私钥输出的编码格式也没有选择的余地，只能是 PEM 编码。这些限制可能很多时候不能满足你的要求。

为了避免上面这些限制，你可以使用 genrsa 指令先生成需要的 RSA 密钥对，然后再使用 req 指令生成证书请求。事实上，在实际应用中，你的 RSA 密钥对甚至可能不是使用 OpenSSL 的指令工具生成的，但这没有关系，只要你的密钥封装格式是 req 指令中 keyform 选项支持的，那么就能够使用 req 指令生成证书请求。下面我们来看怎么使用 genrsa 指令和 req 指令生成一个证书请求。

虽然说 DES3-CBC 方式的安全性一般还是值得信赖的，但我们更愿意使用 256 位的 AES 算法保护我们的 RSA 私钥，毕竟对于私钥来说，怎么提高它的安全性都不为过。显然，req 指令满足不了我们的要求，那么可以使用 genrsa 指令：

```
OpenSSL>genrsa -aes256 -passout pass:12345678 -out rsakey.pem 1024
```

然后，使用 req 指令生成证书请求以便发给 CA 签发证书：

```
OpenSSL>req -new -key rsakey.pem -passin pass:12345678 -out req.pem
```

### 2. 证书请求中使用 DSA 密钥

DSA 密钥是专门用于数字签名的密钥，当你要申请一个用于数字签名的证书的时候就可能需要使用这种类型的密钥。跟 RSA 密钥产生不同，DSA 密钥一般通过其预先生成的 DSA 密钥参数来生成 DSA 密钥。req 指令提供了根据 DSA 密钥参数生成 DSA 密钥的功能，而 DSA 密钥的长度则由给定的 DSA 密钥参数决定。当然，你也可以根据给定的 DSA 密钥对生成证书请求。

下面我们介绍利用 req 指令本身生成 DSA 密钥的功能。由于 req 指令要求输入一组 DSA 密钥参数来生成 DSA 密钥，所以我们必须首先使用 dsaparam 指令生成 DSA 密钥参数。需要注意的是，req 指令只能接受 PEM 编码的 DSA 密钥参数。

首先生成 DSA 密钥参数的指令如下：

```
OpenSSL>dsaparam -out dsaparam.pem 1024
```

下面生成使用 DSA 密钥的证书请求文件，并将生成的 DSA 密钥保存在 dsakey.pem 文件中：

```
OpenSSL>req -new -newkey dsa:dsaparam.pem -keyout dsakey.pem -out req.pem
-passout pass:12345678
```

跟使用 req 指令生成 RSA 密钥对相同的是，输出 DSA 私钥的加密算法只能是 DES3-CBC 模式，编码方式也只能是 PEM 方式。所以为了灵活和方便，你可能愿意使用 gendsa 指令来生成 DSA 密钥，然后再使用 req 指令生成证书请求文件。

首先我们利用刚才生成的 DSA 密钥参数文件生成一对 DSA 密钥：

```
OpenSSL>gendsa -out dsakey.pem -aes256 dsaparam.pem
```

接着，我们使用 req 指令生成一个使用上述 DSA 密钥的证书请求：

```
OpenSSL>req -new -key dsakey.pem -passin pass:12345678 -out req.pem
```

再次强调，上面我们在生成 RSA 或者 DSA 密钥对的时候，都只输出了私钥文件，这是因为私钥文件结构中已经包含了相应的公钥参数，req 指令使用私钥的时候，事实上主要是从其中提取出相应的公钥，并封装在 PKCS＃10 的证书请求结构中，然后采用私钥对整个证书请求结构签名从而完成操作。

### 3. 谁来生成密钥对?

谁来生成密钥对?这是一个永远不可能找到统一答案的问题。

一般来说,申请一个证书的时候,为了保证私钥确实是证书主体名所代表的实体自己唯一拥有,那么要求密钥对必须是该实体自己在本地产生,这样就可以避免密钥被第三方(包括 CA)知道。但是现实并非如此简单,如果你使用的是一个用于密钥交换的证书,那么当一个用户或者你自己采用你的密钥对的公钥对一个加密数据的密钥进行保护,然后使用该数据加密密钥加密一个重要文件,假设这个文件是公司重要的文件,而只有你才有备份,这种情况就非常危险。一旦你丢失了自己的私钥或者由于你的突然消失导致私钥再也不能恢复,那么该文件就可能永远不能被恢复了。对于一个组织或公司来说,这种情况显然是不能允许的,这就导致了对密钥备份功能的要求。如果考虑到这点,为了方便管理,通常就由 CA 或者 RA 来生成用户的密钥对,并在发放证书的时候将证书和密钥对一起发送给用户。但这显然违背了私钥是唯一被主人拥有的假设,这样你可能觉得非常没有安全感,因为掌握备份密钥库的人可能随时可以查看你的任何资料和通信内容,但是你没有任何办法。

对于用于数字签名的证书来说情况会好一点,因为数字签名不需要对整个文件进行加密,它不存在数据恢复的问题。即便私钥丢失或者破坏了,也只是不能使用它继续签名其他文件而已,换一个就是了。所以用于数字签名的密钥对不需要备份功能,一般可以由用户自己产生。从法律角度而不是保密角度来看,数字签名的唯一性显得尤为重要。

由于上述的这些原因,现在很多系统都使用所谓双证书的概念,即一个证书用于密钥交换,一个证书用于数字签名。用于密钥交换的证书其密钥对一般由 CA 或者 RA 代替用户产生;而用于数字签名的证书的密钥对则由用户自己产生。事实上,这样的做法只是在法律上得到了保证,而数据保密性能则很值得怀疑,尤其是个人的隐私问题。

具体到技术上,如果采用的是用户自身生成密钥对,那么 req 指令的功能应该由用户来使用。但是如果是 CA 或者 RA 生成密钥对,req 指令跟用户基本就没有关系了,事实上这时候 req 指令的功能应该由 RA 或者 CA 执行,而用户只要简单地把自己的信息交给 RA 或者 CA 就可以了。自己是轻松了,但是,CA 真的那么值得信任?

## 10.2.3 在证书请求中增加扩展项

要真正用好 req 指令提供的功能,仅仅了解 req 指令的选项和参数是不够的,还需要对 req 指令所依赖的 OpenSSL 配置文件中的 req 指令相关字段有比较深入的了解。一下子要全部了解是不可能的,下面的几个部分我们将结合一些具体的功能来给读者展示怎么使用 OpenSSL 配置文件中的证书请求相关字段实现一些扩展的功能。首先我们看看怎么在证书请求中增加一个自己定义的扩展项。

在某些特殊情况下,可能需要在证书中扩展一些标记以便实现一些特定的功能,比如,你可能需要在给你公司员工发放的证书中增加一项权限的功能,这可能给会给管理带来很大的方便,尤其对于有多个服务器的分布式系统中,这种证书尤其有用。下面我们就通过往证书的特征名称字段中添加一个扩展的字段实现存取用户的权限信息。

首先请打开 OpenSSL 默认的配置文件 openssl.cnf,然后确保其中的下列语句没有被注释掉:

```
oid_section = new_oids
```

然后找到 new_oids 字段，在该字段添加下列语句：

```
[new_oids]
……
UserRights = 2.5.4.555
……
```

上面定义了一个新的 OID 对象，其简短名称和长名称都是 UserRights，而 OID 则为 2.5.4.255。下面，我们就要在证书请求的特征名称字段添加我们新增加的这个 OID 对象，使得证书主体名称中能够增加一项代表用户权限的值域。为了使得管理方便，我们限定该项的值只能是两个字符，默认的值是“US”（User），表示普通用户权限。下面是往 req_distinguished_name 字段添加的语句：

```
[req_distinguished_name]
……
UserRights = User Rights
UserRights_max = 2
UserRights_min = 2
UserRights_default = US
……
```

设置完上面这些内容，我们就可以使用这个配置文件作为 req 指令的配置文件，进行新证书请求的签发。你会发现在生成证书请求时提示输入信息中多了一项提示内容，即“User Rights”选项，用户就可以根据自己对不同权限的命名填写内容了。如果你用 text 选项则显示的证书请求主体名称形式如下：

```
Subject:C = CN,ST = BeiJing,O = www.OpenSSL.cn,CN = Wangzh,UserRights = US
```

上面的流程给读者展示了怎么添加一个扩展项内容到证书请求中，这对于增加证书的功能非常有好处。如果证书应用的系统是一个分布式的系统，那么这些选项对于分布式的权限控制等会产生非常大的好处。当然，要在最后生成的证书中增加这些扩展项，CA 的配置文件相应的一些选项也需要做一些更改，我们会在 ca 指令部分对此作进一步介绍。

### 10.2.4　申请用户证书

证书按照其在证书链中的位置分类，一般可以分为两种：终端用户证书和 CA 证书。所谓 CA 证书，是指该证书可以用于签发别的用户证书，也就是说，它可以有下级的证书。终端用户证书则用于具体应用程序或协议中，它不能用来签发别的证书，在证书链中，它处于最末端。

对于一般用户来说，申请的证书都是终端用户证书。对于生成证书请求的 req 指令来说，生成的证书请求是用于申请一个终端用户证书还是 CA 证书是由 OpenSSL 配置文件的证书请求 X.509 v3 扩展项字段来决定的，openssl.cnf 文件中的默认字段名是 req_v3。为了使 req_v3 扩展字段生效，首先必须确保配置文件 req 字段中的下面语句没有被注释掉：

```
req_extensions = v3_req
```

接着，我们修改 v3_req 字段中的 basicConstraints 选项为下面的形式：

```
[v3_req]
……
basicConstraints = CA:FALSE
……
```

然后保存，利用这个配置文件作为 req 指令的配置文件，其生成的证书请求就是一个申请终端用户证书的证书请求，如果你使用 text 选项输出明文信息，你会发现在“Requested Extensions”部分出现下面的信息：

```
X509v3 Basic Constraints:
    CA:FALSE
```

上述的 CA：FALSE 值即表示该证书请求不是 CA 证书请求，而是普通的终端用户证书请求。当然，证书请求的这个扩展字段最后能起到什么作用，还得取决于签发证书的 CA 的具体策略。

### 10.2.5 申请 CA 证书

如果你负责建立一个分公司的 CA 服务器，你可能就面临申请一个 CA 证书这样的任务，当然，你并不需要签发这个证书，你只需要生成一个申请 CA 证书的请求发给你的上级 CA（比如总公司的 CA）然后等待它的签发就可以了。但是，怎么生成一个 CA 证书请求而不是普通的终端用户证书请求呢？诀窍很简单，只需要修改 OpenSSL 配置文件中的一个配置选项即可。事实上，申请一个 CA 证书需要涉及的配置文件内容跟申请一个终端用户证书涉及的配置文件内容是相同的，只是最后我们需要更改 v3_req 字段中 basicConstraints 选项为下面的形式：

```
[v3_req]
……
basicConstraints = CA:TRUE
……
```

然后保存，利用这个配置文件作为 req 指令的配置文件，其生成的证书请求就是一个申请 CA 证书的证书请求，如果你使用 text 选项输出明文信息，你会发现在“Requested Extensions”部分出现下面的信息：

```
X509v3 Basic Constraints:
    CA:TRUE
```

上述的 CA：TRUE 值即表示该证书请求是 CA 证书请求。同样，证书请求的这个扩展字段最后能起到什么作用，还得取决于签发证书的上级 CA 的具体策略。

## 10.3 建立 CA

细心的读者会发现，到目前为止，我们前面的工作可能毫无意义，因为我们可能还没有一个可用的 CA 服务器。不要着急，我们现在就花几分钟建立一个 CA 服务器，有了 OpenSSL，建立一个 CA 服务器非常简单！注意，我们这里谈的是一个 CA 服务器，而不是一个 CA，这两者是有区别的。CA 服务器只是技术上的基础程序，而 CA 则是一个集

技术和管理于一体的膨大机构。

### 10.3.1　CA 服务器的基本功能

CA 服务器是一个应用程序，它从技术上实现了符合 PKI 和 X.509 等相关标准的证书签发和管理功能。总的来说，其基本功能应该包括以下几个方面：接受申请证书的请求、审核证书请求、签发证书、发布证书、吊销证书、生成和发布证书吊销列表（CRL）及证书库的管理。

1. 接受证书请求

当然，这是首先必备的功能，一个 CA 应用程序必须能够从各种途径接受用户的证书请求，至少需要一个这样的途径和接口。一般来说，证书请求的提交归纳起来有三种：在线实时提交、在线非实时提交和本地提交。在线实时提交是指提供一个应用接口（一般是个服务端口），用户可以通过支持 CA 服务器规定的证书请求协议发送证书请求信息，CA 服务器能够实时响应提交的请求并作出是否提交成功的回复，如果 CA 服务器是自动签发证书并且你的申请符合设定的条件，用户就有可能快速得到签发好的证书。这种方式的实现一般采用 C/S 的模式，需要客户端软件的支持。

在线非实时提交一般通过 Web 方式实现，用户可以通过 Web 页面把证书请求提交到服务器指定的一个目录或者数据库中，然后，CA 服务器程序或其管理员可以定期或者不定期检测数据库，如果发现有新的证书请求，则起动证书请求审核和签发流程。这种方式的实现一般采用 B/S 结构。

本地提交方式即用户到 CA 服务器所在地或者 RA 服务器所在地提交或填写自己的资料和申请证书的请求，然后交给 CA 服务器管理员处理。

一般来说，如果你的 CA 实现的是密钥对由用户自己生成的方式，那么一般接受的证书请求都是符合 PKCS#10 标准的。如果不是这样，即 CA 服务器号称可以替代用户生成“安全”的密钥对的方式，那么接受的资料很可能就是一些文本资料，至于具体的格式，就根据自己的需要决定了。

上述三种接受证书请求的模式各有其不同的应用环境。在线实时接受申请模式，实现起来工作量会比较大，并且需要客户端软件的支持；但是也有好处，在某些必须使用数字证书的客户端软件中可以集成证书自动申请功能，这样就简化了用户使用客户端软件时候的配置操作。而基于 Web 方式的在线非实时提交方式实现起来显然比较容易。本地提交方式能够提供对用户身份进行严格确认的机会，这对于证书的安全应用能够提供很好的保障。

2. 审核证书请求

很难说审核证书请求仅仅是一个技术上的问题，事实上，审核过程要求得更多的是管理的策略。当然，你可以通过 CA 服务器程序将你的管理策略实现，比如必须要求用户提交的证书请求中国家名称跟规定的相一致，等等。当然，仅仅这些是不够的，严肃对待一个证书请求的态度是验证严格审核其各个条目的信息，并在有必要的时候跟用户本人进行核实。这就至少要求 CA 服务器能够向管理员显示出一个证书请求的所有详细内容，这就是 CA 服务器对审核证书请求的基本要求，很简单，不是吗？当然，CA 服务器可以实现

得更多一点，比如对一些固定字段设置匹配策略等以增加自动化处理的能力。

3. 签发证书

如果通过了证书请求的审核过程，那么签发证书就是 CA 服务器马上要执行的操作。该操作实际上是将证书请求的信息和 CA 服务器自己的部分信息按照 X.509 的标准进行格式化，然后 CA 服务器使用自己的私钥对所有这些信息进行数字签名，然后形成一个 X.509 标准的证书。接着，当然要将证书保存在证书库中，并对证书库进行更新。

4. 发布证书

证书签发好后，就需要将证书发布出去。发布对象有两种：一种是申请证书的用户本人，必须让他能够及时获得自己的证书；一种是其他用户，可以向他们提供获得所有用户证书的途径。一般来说，可以通过 Web 方式或 LDAP 目录发布证书，当然，对于申请证书的用户本人，也可以通过 Mail 或其他途径发送。

在某些情况下，比如是 CA 服务器生成证书密钥对的情况下，还要考虑私钥的传输问题，这一般来说需要高度考虑其安全性，最好采用硬件设备来传送私钥。如果非要考虑用软件的方式来保存私钥，那么一般需要对私钥进行加密保存，最简单的方式，就是将证书和私钥一起封装成 PKCS#12 的证书然后通过可能的途径发送给用户本人。

5. 吊销证书

有些情况下需要取消一个没有过期的证书的有效性，那么这就需要 CA 服务器执行吊销该证书的操作，这是确保整个 CA 证书域安全性的必备条件。吊销证书的操作一般就是更新证书数据库，发布新的 CRL 等，目的就是为了尽可能让所有用户都知道该证书已经无效。

6. 生成和发布 CRL

虽然所有已经吊销的证书在 CA 有效的证书数据库中都已经不存在，但是用户在很多时候并不能实时地访问和查询 CA 数据库，当然也就不能就此终止所有基于证书的应用服务。这时候就需要利用 CA 服务器发布的证书吊销列表（CRL）来进行证书的验证。CA 服务器必须能够定期更新和发布 CRL，以便用户能够获得准确的 CRL 信息。

7. 证书库管理

CA 服务器的证书库存储了 CA 签发的所有有效证书，有时候，甚至也包括无效的证书和证书相应的私钥，这就要求 CA 服务器能够对证书进行有效的管理。例如对证书状态的更新，无效和过期的证书应该删除或者做相应的标记状态，等等。

### 10.3.2 CA 服务器的基本要素

要建立和运行一个 CA 服务器，那么至少应该具备以下一些要素：

- 一个具备 10.3.1 节所描述功能的 CA 应用程序；
- 一个 CA 证书和一个其相应的私钥，可能是自签名的根证书，也可能是向另一个 CA 申请的证书；
- 证书数据库，用于存储证书（有时候还包括私钥）。

CA 应用程序的选择有很多，如果你不介意微软的神秘代码可能隐含的安全性问题，那么可以用 Windows 系统的 CA 服务器程序。你还可以选择 OpenCA 提供的 CA 服务器，

当然，它事实上就是 OpenSSL 的封装版本。如果你对 OpenSSL 感兴趣，你还可以使用 OpenSSL 的 ca 指令作为你的 CA 应用程序，这是一个不错的选择，当然，可能会有一些缺陷。最后，如果你有足够的勇气，就自己开发一个 CA 应用程序吧，事实上，有了 OpenSSL，这项工作并非像想像的那么困难。而在本书，我们将详细介绍怎么使用 OpenSSL 的 ca 指令建立一个基本可用的 CA 服务器。

CA 证书和私钥的产生根据不同的应用而定，如果你的 CA 服务器是自成体系，并不需要一个上级 CA 的支持，那么可以使用 OpenSSL 的 req 指令生成一个自签名根证书作为 CA 服务器的证书即可。如果你的 CA 服务器是上级 CA 的一个下级 CA，那么你需要生成一个 CA 证书请求并发送给上级 CA，然后等待上级 CA 签发你的 CA 证书。

证书数据库的选择是多样的，你可以使用任何像 SQLServer，Oracle 和 MySql 这样的数据库，也可以使用自定义的文本数据库，甚至可以使用特定的目录结构等。这需要根据自己的需要而定。如果你选择现有的 CA 应用程序，你可能没有这么多选择，比如 OpenSSL 的 ca 指令，就仅仅使用文本数据库。

具备了上述这些基本的要素，你的 CA 服务器就可以启动并运行签发证书的服务了。

### 10.3.3　OpenSSL 的模拟 CA 服务器结构

#### 1. 默认的 CA 目录结构

上面介绍了各种 CA 应用服务器，事实上本书的目的只有一个：就是介绍基于 OpenSSL 的 ca 指令的证书服务器。当然，并非仅仅想告诉你 OpenSSL 的 CA 服务器是如何配置和使用的，而是希望通过上面的介绍，让你深刻理解 CA 服务器的工作流程和操作过程。

OpenSSL 的 ca 指令是一个模拟的 CA 服务器程序，它实现了 10.3.1 节描述的 CA 服务器的基本功能。ca 指令的正常运行依赖于一个目录结构和一个 OpenSSL 配置文件，OpenSSL 配置文件我们已经相对熟悉了，本节主要对这个目录结构进行介绍。图 10-3 显示了整个 CA 目录的结构。

图 10-3 中，粗线条框显示的是目录，而细线条框显示的则是文件。图中，certs 和 crl 目录是使用 ca.pl 程序创建生成的，目前来说，ca 指令并没有使用这两个目录，可以忽略。

newcerts 目录存放新签发的证书文件，证书文件名是序列号，后缀名是 pem，即输出的证书默认都是 PEM 编码的。比如签发的证书如果其序列号是 01，那么其保存在 newcerts 目录的证书文件名就是“01.pem”。

private 目录目前仅对于存放 CA 证书相应的私钥文件有意义，其默认私钥文件名为 cakey.pem。每次使用 CA 指令如果没有指定特别的密钥，那么就会使用该密钥签发证书。

demoCA 根目录下的 cacert.pem 文件就是 PEM 编码的 CA 证书文件，里面保存了 CA 的证书。如果没有使用 ca 指令的选项指定别的证书文件，那么每次使用 ca 指令签发证书的时候就使用该证书。

demoCA 根目录下的 index.txt 文件是 ca 指令相应的文本证书数据库，index.txt 文

件保存了已经签发的证书的信息和状态，关于该文件的具体结构请参考第 6 章相关内容。

demoCA 根目录下的 serial 文件是 ca 指令决定签发证书的序列号所依赖的文件，其格式和内容请参考第 6 章相关内容。

在使用 ca 指令签发证书后，你还会发现在根目录下还会产生 serial.old 和 index.txt.old 文件，这是上次签发证书时 serial 和 index.txt 文件的备份文件，一般来说不用理会它们。

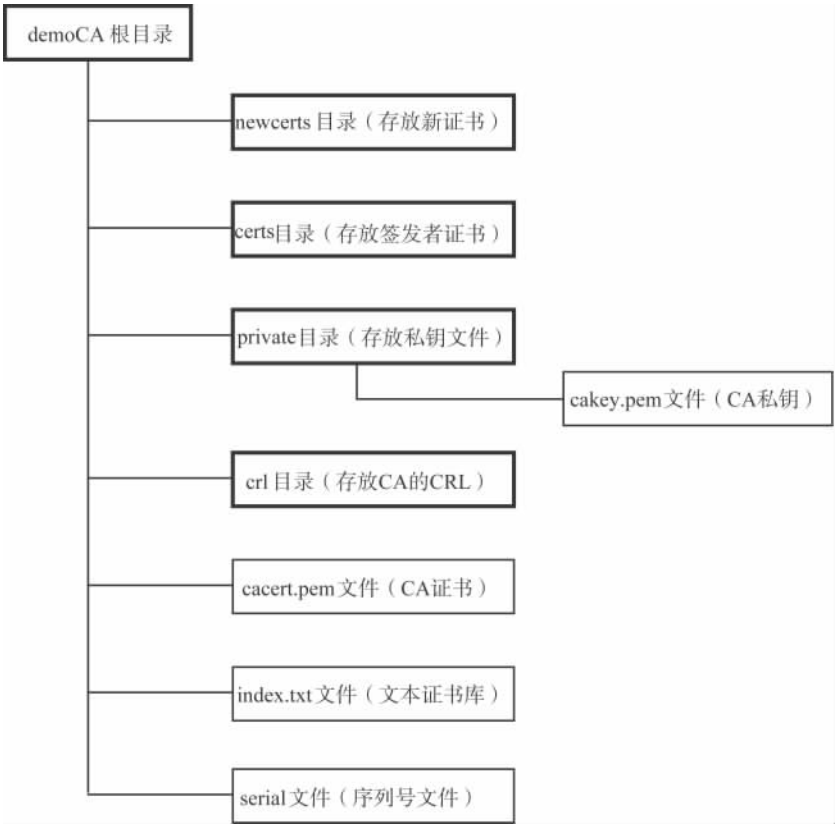


图 10-3 OpenSSL 的 CA 目录结构

## 2. OpenSSL 配置文件

事实上，上面所说的所有目录结构都是可以改变的，它们都由 OpenSSL 配置文件的 CA 字段所决定，让我们来看看默认的 openssl.cnf 文件的这些相关配置选项：

```
[CA_default]
dir = ./demoCA
certs = $dir/certs
crl_dir = $dir/crl
database = $dir/index.txt
new_certs_dir = $dir/newcerts
certificate = $dir/cacert.pem
serial = $dir/serial
```

```
crl = $dir/crl.pem
private_key = $dir/private/cakey.pem
RANDFILE = $dir/private/.rand
```

如果你愿意，你可以通过更改上述这些选项的值来更改 ca 指令使用的目录结构和文件。OpenSSL 的配置文件在 ca 指令中的作用不仅仅是如此，还有更强大的功能，读者可以参考第 6 章对 OpenSSL 配置文件的介绍，本书后面的章节也会做进一步的解释。事实上，在使用 ca 指令时，指定特定的 OpenSSL 配置文件也是必备的条件之一。

为了方便起见，本书后面的章节在介绍 ca 指令时，都以默认的目录结构为准。

### 10.3.4　建立基于 OpenSSL 的 CA 服务器

#### 1. 手动创建一个 CA 目录结构

假设我们已经拥有了 ca 指令程序和配置文件，为了使 ca 指令能够正确运行，首先就是按前面介绍的 CA 目录结构的要求创建这样一个 CA 目录结构和相应的文件。如果你愿意自己来一步一步建立这么一个目录，那么可以按照下面的步骤进行。

① 在 openssl 可执行程序（Windows 下是 openssl.exe）所在目录下创建一个名为“demoCA”的目录。

② 在 demoCA 目录下创建 newcerts，private，crl 和 certs 子目录。

③ 在 demoCA 目录下创建一个空的名为 index.txt 的文本文件。

④ 在 demoCA 目录下创建一个空的名为 serial 的文件，用文本编辑器打开（vi 或者记事本），在文件中填入 01，保存退出。

⑤ 将 CA 证书文件转换成 PEM 格式，复制到 demoCA 根目录下，并重新命名为 cacert.pem。

⑥ 将 CA 私钥文件转换成 PEM 格式，复制到 demoCA/private 目录下，并重新命名为 cakey.pem。

经过上述两个步骤，你就建立了一个 ca 程序能够使用的 CA 目录结构。如果你的 CA 使用上级 CA 签发的证书，那么你可能需要将证书和私钥文件转换成 PEM 编码的格式以完成第⑤和第⑥步的配置。如果你使用的是自签名根证书，那么可以使用下面的 req 指令生成一个自签名根证书和私钥：

```
OpenSSL>req -x509 -newkey rsa:2048 -keyout cakey.pem -out cacert.pem
```

#### 2. 自动创建一个 CA 目录结构

手动创建一个 CA 目录的过程是很烦琐的，OpenSSL 开发者能够理解大家的这种心情，提供了一个基于 Perl 的脚本帮助大家自动创建一个 CA 目录结构。前提是，你正确安装了 Perl 软件。

首先你将 OpenSSL 的 apps 目录下的 ca.pl 文件复制到你想要放置 CA 目录结构的目录。接着，在指令行界面进入到 ca.pl 所在的目录，运行下面的指令：

```
C:\OpenSSL>perl ca.pl -newca
```

接着指令会提示你输入已有的 CA 证书文件名或者简单的按 Enter 键创建一个自签名的根证书。按提示执行完后，你会发现一个完整的 CA 目录结构已经创建，是不是轻松很

多？如果你需要用已经申请到的 CA 证书作为这个 CA 的证书，那么可以将你的 CA 证书和相应的私钥分别转换成 PEM 格式，并替代 cacert. pem 文件和 cakey. pem 文件即可。

3. 指定 OpenSSL 配置文件

OpenSSL 的 ca 指令必须使用 OpenSSL 的配置文件才能够正确运行。所以我们首先必须给它指定一个配置文件。在 Unix 系统或 Linux 系统下你只要正确编译和安装了 OpenSSL，这都不会成为问题，因为 ca 指令默认的 OpenSSL 配置文件路径是“/usr/local/ssl/openssl. cnf”。但是，如果是在 Windows 平台，就会产生问题。

首先，如果在 Windows 平台编译 OpenSSL 的时候指定了 Windows 风格的安装路径，那么很好，你只需要将 OpenSSL 的配置文件复制到该目录并命名为 openssl. cnf 文件就可以了。

如果你在 Windows 编译时没有配置安装路径，那么你可以用下面的两种方法之一达到你的目的。第一种方法是使用环境变量。即设置 OPENSSL_CONF 或者 SSLEAY_CONF 环境变量为 OpenSSL 配置文件的绝对或相对 openssl. exe 文件的路径名。设置环境变量的方法可以参考前面的章节，最简单的就是在指令行环境下输入：

```
C:\OpenSSL>set OPENSSL_CONF = C:\OpenSSL\openssl. cnf
```

还有一种可能比较烦琐的方法，就是每次使用 CA 指令时都使用 config 选项指定 OpenSSL 的配置文件路径。

上述的方法你愿意用哪种？这取决于你自己的偏好。

## 10.4 CA 操作

在本章的前面我们介绍过 CA 服务器的基本功能，一个 CA 执行的操作也就围绕这些基本功能展开。在本节，我们将以介绍 OpenSSL 的 ca 指令为例子，演示一个 CA 应该执行哪些操作。同样，读者学习到的不应该仅仅是 OpenSSL 的 ca 指令操作，更多的应该是对 CA 服务器程序的功能更深入的理解，从而能够举一反三。本节除了详细介绍 ca 指令的选项以外，还将结合一些具体的实例展示 ca 指令的功能。

### 10.4.1 ca 指令介绍

1. 功能概述和指令格式

ca 指令模拟了一个完整的 CA 服务器，它的功能不仅仅包括签发用户证书，还包括吊销证书、产生 CRL 及更新证书库等管理操作。下面是 ca 指令的选项和格式：

```
OpenSSL>ca[-verbose][-config filename][-name section][-gencrl][-revoke file]
[-crl_reason reason][-crl_hold instruction][-crl_compromise time][-crl_CA_compro-
mise time] [-subj arg] [-crldays days] [-crlhours hours] [-crlexts section]
[-startdate date] [-enddate date] [-days arg] [-md arg] [-policy arg] [-keyfile
arg] [-keyform fmt] [-key arg] [-passin arg] [-cert file] [-in file] [-out file]
[-notext] [-outdir dir] [-infiles] [-spkac file] [-ss_cert file][-preserveDN][-no-
emailDN][-batch][-msie_hack][-extensions section][-extfile file][-status cert_ser-
```

ial][-updatedb][-engine e_id]

这里一共有 39 个选项，在下面的对这些选项的介绍中，将尽可能地对它们进行分类介绍，以便读者理解和查阅。

2. 证书配置文件选项 config，name，extensions，extfile 和 policy

ca 指令跟 OpenSSL 的配置文件关系非常紧密，ca 指令的证书请求值域匹配策略、扩展项参数及其他默认的参数设置都是依赖于 OpenSSL 配置文件的相关字段。ca 指令除了会自动寻找和使用默认的 OpenSSL 配置文件和字段外，还运行用户在指令中指定的配置文件及特定的字段。

config 选项指定将要使用的配置文件。在默认情况下，如果不使用 config 选项，ca 指令首先会寻找环境变量 OPENSSL_CONF 或者 SSLEAY_CONF 定义的文件名，如果这两个环境变量都没有定义，那么就使用 OpenSSL 安装时默认的路径。这个默认的路径通常情况下是“/usr/local/ssl/openssl.cnf”，当然，你可以在安装 OpenSSL 的时候使用 configure 脚本指令更改这个默认的安装目录。比如通常在 Windows 下编译完 OpenSSL 后，可执行文件 openssl.exe 通常在目录 out32dll 下，而其配置文件则在 apps 目录下，为了在 out32dll 目录下能够正确使用 OpenSSL 的 ca 指令，可以使用 config 选项：

```
OpenSSL>ca -in req.pem -config../apps/openssl.cnf -out cert.pem
```

在默认的情况下，ca 指令使用的 CA 主配置字段由配置文件中 ca 字段的 default_ca 选项的参数值决定。其形式如：

```
default_ca = CA_default
```

当然，某些情况下你可能想使用在同一配置文件下（由 config 选项指定或默认得到的配置文件值）另外一个字段作为 CA 的主配置字段，那么 name 选项就可以实现你这个目的。name 选项的参数是字段名，比如上面的“CA_default”，指定的字段名应该是已经在使用的配置文件中定义好了的。例如在下面的名为 openssl.cnf 的配置文件中，定义了一个新的名为 My_CA 的 CA 主配置字段：

```
……
[My_CA]
dir = ./MyCA
certs = $dir/certs
……
```

为了使用这个字段作为 ca 指令的主配置字段，就应该使用类似下面的指令：

```
OpenSSL>ca -in req.pem -name My_CA -out cert.pem
```

policy 选项让指令使用其参数指定的字段名作为签发证书请求时的匹配策略字段（关于匹配字段的选项定义请参考第 6 章的介绍），其参数是配置文件中定义的一个字段名称。事实上，在 OpenSSL 提供的 openssl.cnf 文件中，就已经定义了两个匹配策略字段：policy_match 和 policy_anything。默认情况下，ca 指令通过 CA 主配置字段的下列选项采用 policy_match 字段作为匹配策略：

```
policy = policy_match
```

如果你不想更改 openssl.cnf，但想使用 policy_anything 字段作为匹配策略字段，则可以使用下面的指令实现你的要求：

```
OpenSSL>ca -in req.pem -policy policy_anything -out cert.pem
```

假设你的 CA 需要管理不同的用户群，而每个用户群需要自己特定的匹配策略，比如每个用户群组织名称都不同但是要求同一用户群的用户组织名称必须相同，那么就可以在配置文件中预先定义不同的匹配策略字段，然后根据具体用户使用不同的匹配策略字段来签发证书。policy 选项在这种情况下可以大显身手。

extensions 和 extfile 两个选项用来选定签发证书时候使用的 X.509 v3 扩展项定义字段。extensions 的参数是字段名，用来选定配置文件中用于定义 X.509 v3 扩展项的特定字段，默认的字段所在文件跟 config 选项选定的配置文件或者默认的配置文件是同一个文件。但是某些情况下你可能希望使用一个专用的文件用来保存 X.509 v3 扩展项信息，那么这时候可以使用 extfile 选项指定这个文件。如果仅仅使用了 extfile 选项指定额外的 X.509 v3 扩展项配置文件，而没有使用 extensions 选项，那么具体使用 extfile 指定的文件中那个字段，由默认配置文件中 CA 主配置字段的 x509_extensions 选项决定，类似如下的格式：

```
x509_extensions = usr_cert
```

如果 extensions 和 extfile 两个选项都没有使用，那么默认的情况下就会使用 config 选项指定或者默认配置文件的 CA 主配置字段中 x509_extensions 选项参数指定的字段。例如在 openssl.cnf 文件中指定的就是上面所描述的 usr_cert 字段。假设我们在 my_config.cnf 文件中定义了一个 X.509 v3 扩展项字段如下：

```
[x509_v3]
……
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer:always
basicConstraints = CA:true
……
```

那么为了使用这个扩展项字段，使用类似下面的指令：

```
OpenSSL>ca -in req.pem -extensions x509_v3 -extfile my_config.cnf -out cert.pem
```

使用好 extensions 和 extfile 两个选项能够使 ca 指令的运用更加灵活。例如，你管理的用户可能其证书用途各不相同，这可以通过在 X.509 v3 的扩展项中来定义。为了方便签发各种不同用途的证书，如果你不使用 extensions 选项，或许你需要不断烦琐地更改配置文件的 X.509 v3 扩展项字段（默认是 usr_cert），相信这样下去，很快你就会产生辞职的想法。不过，如果你会使用 extensions 选项，事情就不一样了，你可以根据需要预先编辑好几个不同的 x509_v3 扩展项字段，然后根据签发证书的需要使用 extensions 和 extfile 选项选用不同的字段，从而实现不同权限的证书的签发。需要注意的是，如果你提供了有效的 X.509 v3 扩展项字段（不管通过指令选项提供还是配置文件选项提供），即便字段是空的，指令也会生成一个 X.509 v3 版本的证书。但是如果没有提供有效的 X.509 v3 扩展项，那么就生成 X.509 v1 证书。

3. 输入和输出选项 in，infiles，out 和 outdir

in 选项的参数是一个可以包含路径的文件名，该文件保存的应该是一个 PEM 编码的 PKCS#10 格式的证书请求。事实上，in 选项仅用于证书请求的输入，而不能输入其他类

型的内容，也就是说，如果你使用了 in 选项，那么表示你要执行的操作就是签发一个证书。

infiles 选项的参数是一系列包含 PEM 编码证书请求的文件，跟 in 选项唯一不同的是，它可以一下子输入多个证书请求文件，而不是一个。该选项总是作为指令的最后一个选项，其后面的所有参数都被认为是证书请求文件。

out 选项指定了输出签发好的证书或者新生成的 CRL 的文件。此外，如果没有使用 notext 选项，那么证书的明文信息也会输出到 out 选项指定的文件中，这在大多数 Windows 平台上会导致证书文件不能被正确解释，解决的最简单办法是使用 notext 选项禁止这些明文信息的输出。不过，在较新的 Windows 系统，如 Windows Server 2003，这个问题得到了解决。out 选项指定的文件中输出的证书或者 CRL 都是 PEM 编码的。

outdir 选项指定了新生成证书的输出目录，在默认情况下，新生成证书总是输出到 newcerts 目录，并且是以序列号加“pem”后缀成为一个完整的证书文件名。这些证书都采用了 PEM 编码。如果使用 outdir 选项指定了新的目录，那么新生成的证书文件就会输出到这个新指定的目录。需要注意的是，不管 out 选项是否给出，都不会对输出到 outdir 或者其默认目录的证书文件产生影响。

4. 特殊的证书请求输入选项 ss_cert 和 spkac

前面介绍的两种证书请求的输入方式，即使用 in 或者 infiles 选项。它们的区别是输入单个证书请求文件还是输入多个证书请求文件。而它们的共同点则是输入的都是 PEM 编码的符合 PKCS#10 标准的证书请求。OpenSSL 还提供了其他两种输入证书请求的方法，即 ss_cert 选项和 spkac 选项支持的方法。

ss_cert 选项允许用户输入一个自签名的证书作为申请证书的证书请求，ca 指令将会从这个自签名证书中提取用户信息和公钥用于签发最后的用户证书。也就是说，你可以使用 req 指令生成一个自签名证书而不是证书请求，然后用这个自签名证书来向上级 CA 提供信息申请你的证书。

spkac 选项允许用户输入一个 SPKAC 格式的文件作为申请证书请求的资料，spkac 是 Netscape 规定的一种格式，该格式包含了一个签名的公钥和其他附加用户信息，具体可以参考本书 spkac 指令的相关介绍。需要注意的是，使用 spkac 作为证书请求输入时，输出的证书编码是 DER 格式，而不是 PEM 格式的。

5. 证书有效期选项 startdate，days 和 enddate

这三个都是用来设置证书有效日期的选项，分别是设置证书的生效时间、证书的到期时间及有效时间。

startdate 设置证书的生效时间，其参数格式是“YYMMDDHHMMSSZ”，每两位代表一个时间变量，分别是“年月日时分秒”。比如要定义签发的证书生效时间是 2003 年 9 月 10 日 12 时 24 分 36 秒（格林威治时间），那么 startdate 选项的参数就应该为“030910122436Z”。证书生效时间除了可以在 ca 指令的 startdate 选项指定外，也可以在配置文件的 CA 主配置字段的 default_startdate 选项指定，其参数格式跟 startdate 选项参数格式一致。如果没有设置 startdate 选项，也没有在配置文件中定义 default_startdate 选项，那么证书生效时间就是签发证书的时间。

days 选项设置证书的有效天数。所谓有效天数，也就是从证书生效的时间到证书到期的时间之间的天数。

enddate 选项的参数格式跟 startdate 选项一样，用来设置证书的到期时间。同样，你也可以通过配置文件 CA 主配置字段中的 default_enddate 选项设定证书到期时间。如果设置了 enddate 选项，那么 days 选项将会被自动忽略。如果没有设置证书到期时间，那么就会根据 days 选项设置的有效时间来决定证书到期时间。如果指令中也没有使用 days 选项，那么指令就会从 CA 主配置字段读取 default_days 选项的参数来决定证书有效期。

6. 证书内容选项 subj，preserveDN，noemailDN，md 和 msie_hack

一般情况下，签发证书的时候，证书的主体名称（subject name，也称为特征名称）由证书请求中的主体名称决定。不过，有时候用户自己填写的资料可能并不能让你这个挑剔的 CA 管理员满意，那么你可以使用 subj 选项重新填写用户的证书主体名称。subj 选项的参数格式应该如下：

```
/type0 = value0/type1 = value1.....
```

其中，“type * ”必须是已经由 OpenSSL 内部定义或者在配置文件中定义了的数据对象。

preserveDN 选项使指令在签发证书的时候让证书主体名称内的各项内容顺序跟证书请求中的顺序保持一致。而在默认情况下，证书主体名称内的各个选项顺序是按照配置文件中的证书匹配策略字段的选项顺序进行排列的。这个选项对于老版本的 IE 证书管理器是必要的，而现在则显得很不必要了。

一般来说，证书的主体名称里面可以包含 E-mail 项目，但是，因为 E-mail 可能是一个经常变动的项目，所以如果不把它放在证书主体名称中而是放在主体别名中会更好一点。noemailDN 选项可以帮你做到这一点，当证书请求中有 E-mail 这一项，但是你不希望在证书主体名称中出现 E-mail 时，可以使用这个选项来去除。如果你还使用了证书主体别名，并使用了“email：move”或者“email：copy”策略（详细选项使用方法请参考第 6 章），那么使用该选项后指令会自动将证书请求中的 E-mail 项目移动到证书主体别名中。

7. CA 证书和私钥选项 cert，keyfile，keyform，passin 和 key

在签发证书的时候，需要使用 CA 的证书和私钥对用户的证书请求进行签发证书操作。默认情况下，证书和私钥文件都是通过配置文件的 CA 主配置字段的 certificate 和 private_key 两个选项决定的。不过，OpenSSL 的 ca 指令提供了 cert 和 keyfile 选项让用户能够选择另外的证书和私钥文件作为签发证书的 CA 证书和私钥文件。或许你很难想像这种特殊的需要，为什么不使用预先设置好的证书和私钥文件呢？但是，至少在测试的时候，我们需要这些灵活的选项。事实上，这些选项不仅仅可以用于测试，因为 CA 的指令不仅仅用于签发证书，还可以用于吊销证书和发布 CRL，某些时候，发布 CRL 的角色跟 CA 的角色可以是不同的，即需要使用不同的证书来完成证书签发工作和生成 CRL 的工作，但是它们都使用同一个 ca 指令，所以这些选项就能派上用场了。

cert 选项的参数是一个可以包括路径的文件名，文件里面是一个 PEM 编码的 X.509 证书文件。

keyfile 选项的参数也是一个可以包括路径的文件名，不过，该文件里面可以包含的私钥格式就比证书文件丰富多了。目前来说，支持的编码格式包括 PEM 编码格式、DER 编码格式、PKCS＃12 编码格式、Netscape 编码格式、旧版本的 IIS SGC 编码格式及 ENGINE格式。当然，OpenSSL 指令不能自动识别这些格式，必须使用 keyform 选项指定某种具体的格式，各种格式对应的参数名称请参考表 8-2。

passin 选项给定了读取私钥文件的时候需要提供的口令，当然，跟我们所熟悉的一样，提供口令的方式可以是多样的，比如从指令直接输入、从环境变量获取或者从文件获取，等等。

key 选项是一个老选项，其作用是直接从指令输入一个口令，其功能可以被“-passin pass:”选项所替代。

8. 证书吊销选项 revoke，crl_reason，crl_hold，crl_compromise 和 crl_CA_compromise

经过上面那些指令选项的介绍，如果你是一个勇于实践的人，你一定已经使用 OpenSSL 的 ca 指令签发了不少乱七八糟的证书，这些证书基本上是没有任意意义的，不是么？那么就让我们开始吊销它们，这可能是它们存在的唯一意义了。

revoke 选项让你能够轻松地吊销一个证书，其参数就是你需要吊销的证书文件名，revoke 选项吊销的证书应该是 PEM 编码的 X.509 证书。

当然，仅仅这样吊销可能还满足不了你的管理需求，某些情况下，你甚至希望告诉所有用户为什么要吊销这个证书以证明你是正确的。那么，使用 crl_reason 选项吧，crl_reason 选项的参数是 CRL v2 版规定的一些吊销理由，它们如表 10-5 所示。

**表 10-5　CRL 吊销参数**

| 吊销参数 | 代表意义 |
|---|---|
| unspecified | 没有定义 |
| keyCompromise | 私钥泄密。证书主体的私钥被认为已经泄密 |
| CACompromise | CA 私钥泄密。CA 的私钥被认为已经泄密，一般在吊销 CA 证书的时候使用这个参数 |
| AffiliationChanged | 关系改变。证书中的主体名或者其他信息已经改变 |
| superseded | 已被取代。证书已经被新的证书取代 |
| cessationOfOperation | 操作停止。证书已经不再需要 |
| certificateHold | 证书冻结。证书被有效冻结，如果使用了这个原因码，还可以使用冻结指示代码扩展（下面介绍的 crl_hold 选项的参数） |
| removeFromCRL | 从 CRL 删除。仅用于增量 CRL，表示将该证书已经过期或者从冻结状态释放。OpenSSL暂时没有实现增量 CRL 的功能 |

上述的吊销参数对大小写是不敏感的。如果在吊销操作的时候使用了这些吊销参数，那么生成的 CRL 就是 v2 版本的，如果所有吊销证书操作都没有使用任何参数，那么生成的 CRL 将是 v1 版本的。

当你希望使用 certificateHold 码作为吊销一个证书的理由时，还可以使用冻结指示代码扩展，这可以通过使用 crl_hold 参数来添加。目前，crl_hold 支持的参数包括 holdIn-

structionNone，holdInstructionCallIssuer 和 holdInstructionReject。crl_hold 选项使用后，设置证书吊销码为 certificateHold，并且用其参数设置冻结指示代码扩展项。

crl_compromise 选项使用后，将证书吊销原因设定为 keyCompromise，其参数表示私钥泄密的时间。它的参数是符合“YYYYMMDDHHMMSSZ”格式的时间。其中“YYYY”代表 4 位的年份，而其他位跟 startdate 选项的参数代表的意义一样的。

crl_CA_compromise 选项使用后，将证书吊销原因设定为 CACompromise，其参数表示私钥泄密的时间。它的参数格式跟 crl_compromise 格式一致。

上述的 crl_reason，crl_hold，crl_compromise 和 crl_CA_compromise 只要同时使用一项即可，如果同时使用了多项，那么在指令中最靠后的一个将是有效的，而前面的都视为无效。

### 9. CRL 生成选项 gencrl，crldays，crlhours 和 crlexts

执行完证书吊销操作后，证书库中相应的证书记录会更新，但是，对于离线的应用验证程序而言，还是没有办法知道被吊销的证书的序列号。为了让所有证书用户及时得到吊销证书的消息，CA 必须及时生成和发布 CRL。

gencrl 选项用于生成一个 CRL，并将生成的 CRL 使用 PEM 编码输出到 out 选项指定的文件，如果没有使用 out 选项，则输出到标准输出设备。

为了保证 CRL 能及时反映证书库的更新信息，CRL 也必须不断定期或者不定期更新，最好的方法就是，当发布一个 CRL 的时候，同时限定该 CRL 的有效期，那么到了有效期之后，用户就必须下载新的 CRL 来使用。CRL 更新的间隔时间可以根据需要选择，如果是非常重要的证书应用系统，并且用户数量比较大，那么单位时间内执行吊销操作的可能性就大，这时候可以将 CRL 的更新时间设置得短一点。反之，如果系统用户数量不多，证书应用系统的安全性也不是要求特别高，那么更新时间可以设置得稍微长一点。crldays 和 crlhours 选项跟 gencrl 选项一起使用以设置 CRL 的更新时间。默认情况下，如果没有使用 crldays 选项或者 crlhours 选项，指令会从配置文件的 CA 主配置字段的 default_crl_days 或者 default_crl_hours 选项读取默认值。crldays 以“天”为单位设置 CRL 的有效期，而 crlhours 则以小时为单位设置 CRL 有效期，这两个选项可以一起使用，比如你要设置 CRL 的有效期为 5 天 12 个小时，那么可以设置 crldays 的参数为 5，而 crlhours 为 12。

crlexts 指定了生成 CRL 的时候使用的扩展项定义字段，其参数为字段名称。CRL 扩展项字段是在 OpenSSL 配置文件中定义的。如果定义了 CRL 扩展项字段，即便该字段是空的，生成的 CRL 也将是 v2 版本的。如果没有提供 CRL 扩展项字段，则将生成 v1 版的 CRL。

### 10. 证书管理选项 status 和 updatedb

status 选项用来查看证书库中指定证书的状态，比如是否有效、吊销或者过期等。该选项的参数是证书序列号。

updatedb 选项用来更新文本数据库的证书状态，它主要用来更新证书库中已经过期的证书的状态信息。通常情况下，即便证书已经过了有效期，如果不使用这个选项，OpenSSL 的 ca 指令也不会自动更新证书库中相应证书条目的状态，所以需要使用这个选

项来自动更新证书库的状态信息。

11. engine 选项

engine 选项告诉 ca 指令尽可能使用 Engine 设备提供的加密函数和算法替代 OpenSSL 本身的函数。对于 ca 指令来说，可能替代的算法包括：签名时使用的信息摘要算法、签名时使用的私钥加密算法、验证证书请求中的数字签名时使用的公钥解密算法、随机数函数及私钥数据。

engine 选项对于使用硬件加密设备中的私钥作为 CA 的私钥具有重要的意义，因为硬件中的私钥一般是不能导出的，从而能够增加 CA 私钥的安全性。当然，这需要相应 Engine接口的支持。

12. notext，batch 和 verbose 选项

使用 ca 指令签发完一个证书的时候，你是不是经常发现大部分版本的 Windows 都不能正确解释你的 PEM 编码的证书文件？事实上这是因为 OpenSSL 的 ca 指令在签发证书的时候，同时将证书的明文解析信息输出到了保存编码证书的文件前面，从而导致这些不怎么标准的 Windows 程序解码失败（Windows Server 2003 已经解决了这个问题）。解决这个问题很简单，在签发证书的时候使用 notext 选项，那么指令就不会输出明文信息到证书文件。

使用 batch 选项后 ca 指令以批处理方式工作，在这种方式下，ca 指令不提示用户输入任何信息而直接签发所有输入的证书请求。

使用 verbose 选项将会输出更详细的一些操作过程信息，你如果对 ca 指令的操作过程感兴趣，试试这个选项好了。

13. 简单的例子

这里给出一些简单的指令例子，让大家对指令的选项能够进一步加深认识。当然，要真正掌握 ca 指令，还是要自己去操作，多做实验。

下面是一个签发证书的简单例子，为了让 Windows 能够正确解码签发的证书，我们使用 notext 选项。

```
OpenSSL>ca -in req.pem -out cert.cer -notext
```

下面我们将同时签发两个证书请求，并限定证书的有效期为 2003 年 9 月 9 日到 2004 年 10 月 10 日，我们没有使用输出文件，默认的证书将输出到 ./demoCA/newcerts 目录下。

```
OpenSSL>ca -notext -startdate 030909000000Z -enddate 041010000000Z -infiles req1.pem req2.pem
```

下面使用 Engine 中的私钥签发证书的例子，指令中提供的私钥文件事实上是一个公钥文件。

```
OpenSSL > ca -in req.pem -out cert.cer -notext -cert pkcs11cert.pem -keyfile pkcs11key.pem -keyform e -engine pkcs11
```

下面的指令吊销一个证书，并指明吊销的原因是因为证书私钥泄漏。

```
OpenSSL>ca -revoke cert.pem -crl_reason keyCompromise
```

如果你需要在吊销证书的时候具体指明私钥泄漏的时间，那么可以使用下面的指令：

```
OpenSSL>ca -revoke cert.pem -crl_compromise 20030915000000Z
```

证书吊销完了，下面就是生成一个CRL，我们设定CRL更新的时间为7天7小时。指令如下：

```
OpenSSL>ca -gencrl -crldays 7 -crlhours 7 -out crl.crl
```

查询序列号为1的证书的状态：

```
OpenSSL>ca -status 1
```

更新文本证书数据库：

```
OpenSSL>ca -updatedb
```

### 10.4.2 在证书中增加扩展项

在10.2.3节，介绍了怎么在证书请求中增加自定义的扩展项。但是，证书请求中增加了扩展项，并不等于签发的证书就会自动增加扩展项。事实上，从严格意义上说，证书请求应该是用户生成的申请文件，也就是说，其所有内容基本上都是代表了用户的意愿，并不能由此推断CA的策略。所以，为了使证书中能够顺利增加10.2.3节的扩展项信息，我们还需要在CA指令的配置文件上做一些工作。

你可能已经可以想像得到，我们的证书扩展项工作同样需要在OpenSSL的配置文件中完成。首先，请确认你已经按照10.2.3节在OpenSSL配置文件openssl.cnf中增加了扩展项的OID，即以下信息：

```
oid_section = new_oids
[new_oids]
……
UserRights = 2.5.4.555
……
```

注意，如果证书请求指令（req）使用的配置文件跟证书签发指令（ca）使用的配置文件不相同，必须确保上面的信息在两个配置文件中都存在并且一致。

下面，找到配置文件中的CA策略配置字段，如在openssl.cnf文件中是policy_match或者policy_anything字段，在其中增加新增扩展项：

```
[policy_anything]
countryName = match
stateOrProvinceName = optional
……
UserRights = supplied
```

当然，跟其他DN字段信息一样，新增扩展项的匹配策略也可以根据CA的需要调整为optional或者match等。

完成上面的配置之后，保存配置文件，并使用该配置文件签发证书，就可以生成增加了扩展项的证书。当然，上述配置方法使得扩展项信息是增加在证书主体名称中的。

### 10.4.3 签发用户证书

在10.2.4节，介绍了怎么申请一个终端用户证书，即怎么使用OpenSSL的req指令生成一个终端用户证书请求。同样，签发什么类型的证书事实上是由CA决定的，也就是

说，即便生成了一个有效的终端用户证书请求，但是如果接受申请的 CA 不签发终端用户证书，或者，甚至因为错误配置导致签发了 CA 证书，都是可能发生的。

可以通过在 OpenSSL 配置文件中的 X.509 v3 扩展项字段告诉 ca 指令签发用户证书。ca 指令在使用默认的 openssl.cnf 作为配置文件的情况下，使用的 X.509 v3 扩展项字段是 user_cert 字段。在 openssl.cnf 文件中找到该字段，确保 basicConstraints 选项的参数如下：

```
basicConstraints = CA:FALSE
```

当然，默认情况下一般就是如此。使用该配置文件签发的证书就是一个终端用户证书。如果你使用的是自己定义的另外的 X.509 v3 扩展字段，那么也只需要保证上述的 basicConstraints 选项值为 CA：FALSE，就可以成功签发一个终端用户证书了。使用自定义的字段的方法就是在 ca 指令中使用 extensions 选项，当然，也可以通过修改配置文件中 CA 主配置字段的 x509_extensions 选项的值来实现。

事实上，当 OpenSSL 的 ca 指令使用默认的 openssl.cnf 作为配置文件的时候，其签发的证书总是终端用户证书，你可能并不需要做更多的工作来实现这个功能，更关心的可能是 10.4.4 讲述的内容。

### 10.4.4　签发下级 CA 证书

如果你正在为一个大型的跨区域企业或者政府部门建立一个 CA 系统，那么你可能需要建立一个多级的 CA 体系，这就涉及签发下级 CA 证书的问题。

从技术上来说，一个 CA 证书和一个用户证书并没有本质区别，它们的数据项是基本相同的，格式也是相同的，一个下级 CA 证书的合法性也是通过上级 CA 的数字签名来保证和证明的。事实上，区别普通终端用户证书和 CA 证书的唯一标记就是 X.509 v3 扩展项字段中的选项 basicConstraints。如果该选项值为 CA：FALSE，则表示证书是普通的终端用户证书；如果该选项值是 CA：TRUE，则表示该证书是 CA 证书。

在 OpenSSL 的默认配置文件 openssl.cnf 文件中已经存在一个签发 CA 证书的 X.509 v3扩展字段 v3_ca。将该字段的 basicConstraints 设置为如下格式：

```
basicConstraints = CA:TRUE
```

上面的设置无误后，如果要签发 CA 证书，则使用 extensions 选项指定 X.509 v3 扩展项字段为 v3_ca，如下形式：

```
OpenSSL>ca[其他选项]-extensions v3_ca
```

当然，如果你的配置文件专门用于签发 CA 证书，那么更方便的办法是直接在 CA 主配置字段将 x509_extensions 选项的值设置如下：

```
x509_extensions = v3_ca
```

当然，你还可以使用自己定义的 X.509 v3 扩展项字段，其使用方法是相同的。

通过上面这些介绍，你就可以使用 OpenSSL 的指令顺利签发一个下级 CA 证书，从而建立整个证书域的分级管理，对于大型的企业或者机构来说，这非常方便。

### 10.4.5　建立一个多级 CA

#### 1. 任务概述

经过前面章节的介绍，我们已经知道该怎么申请证书、建立 CA、签发 CA 证书和普

通用户证书，本节将以一个实际的例子完成 CA 操作部分的介绍。我们的任务是建立如图 10-4 所示的 CA 体系结构，并签发两个终端用户证书。

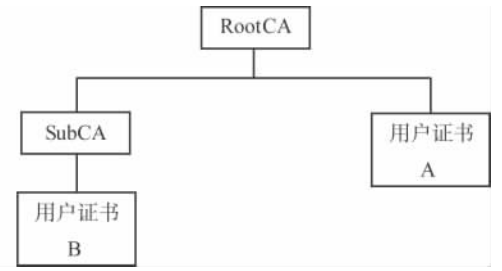


图 10-4　多级 CA 证书结构例子

使用 OpenSSL 建立如图 10-4 这样的一个 CA 结构需要完成的技术工作包括：

① 使用 req 指令生成一个自签名根证书作为 RootCA 的证书，配置 RootCA 的配置文件；

② 使用 req 指令生成一个 CA 证书请求，并使用 RootCA 的证书签发该请求形成 SubCA 的证书，配置 SubCA 的配置文件；

③ 使用 req 指令生成一个证书请求，并使用 SubCA 的证书签发该请求形成用户证书 B；

④ 使用 req 指令生成一个证书请求，并使用 RootCA 的证书签发该请求形成用户证书 A。

在完成这些步骤的整个过程中，使用的 OpenSSL 指令只有 req 和 ca 两个，并且参数形式基本是相同的。此外，还需要配置 OpenSSL 的配置文件。通过整个流程，读者不仅仅要进一步熟悉 req 和 ca 指令的应用，更重要的是对整个 CA 体制架构有一个更加深刻和感性的认识。所有这些操作都是基于 Windows 平台的，在 Linux 平台，过程类似，甚至更简单一些。

当然，在构建实际应用的系统时，还有更加复杂的管理工作需要完成，但是，这不是本书所关心的问题。下面让我们开始这个有趣的过程。

2. 建立 RootCA

建立 RootCA 技术上抽象的任务有三部分：建立一个 CA 目录结构、生成一个自签名根证书和修改配置文件。

为了管理方便，我们选定 RootCA 的根目录在 C 盘。首先，在 C 盘根目录创建一个名为“RootCA”的目录。从编译好的 OpenSSL 目录中复制“openssl.exe”程序，“openssl.cnf”和 ca.pl 文件到“C:\RootCA”目录下。打开 Windows 的指令行界面，进入到“C:\RootCA”目录，执行下面的指令：

```
C:\RootCA>perl ca.pl -newca
```

根据其提示步骤完成整个操作流程，就可以成功在 C:\RootCA 目录下创建一个 demoCA目录。执行 ca.pl 脚本指令的过程中，会提示产生自签名的根证书，可以忽略。下面我们使用 req 指令来生成一个使用 2 048 位密钥的自签名根证书作为 RootCA 的证书。

首先执行如下指令启动 OpenSSL.exe 程序进入 OpenSSL 的运行环境。

```
C:\RootCA>openssl
```

然后执行下面的指令：

```
OpenSSL>req -x509 -newkey rsa:2048 -out rootca_cert.pem -keyout rootca_key.pem
-config C:\RootCA\openssl.cnf
```

然后，将文件 rootca_cert.pem 重命名为 cacert.pem，复制到“C:\RootCA\demoCA”目录下取代原来的文件；将文件 rootca_key.pem 重命名为 cakey.pem，复制到“C:\RootCA\demoCA\private”目录下取代原来的同名文件。这就完成了 RootCA 证书和其相应私钥的设置。

为了使 RootCA 能够同时签发普通用户证书和下级 CA 证书，很多时候我们还需要对 OpenSSL 配置文件（C:\RootCA\openssl.cnf）进行一些配置，但是，由于 OpenSSL 默认的配置文件本身已经具有了两个不同的 X.509 v3 扩展字段用于签发用户证书（usr_cert）和签发 CA 证书（v3_ca），所以在这里不需要再做更多的配置。当然，在实际的应用中，可能还需要对其他一些扩展项和匹配策略等配置信息进行修改。

到此为止，我们已经成功建立了一个 RootCA，是不是很简单？

### 3. 建立 SubCA

SubCA 是 RootCA 的下级 CA，它的证书的合法性需要由 RootCA 签名来证明。建立一个 SubCA 的简要步骤如下：

① 建立 SubCA 证书目录；

② 生成 SubCA 密钥对和证书请求；

③ 使用 RootCA 签发 SubCA 的证书；

④ 设置 SubCA 的证书和私钥文件；

⑤ 设置 SubCA 的配置文件。

SubCA 跟 RootCA 是两个不同的 CA，在实际情况下，它们一般不在同一个计算机内运行，但是，大家都应该明白，这里只是一个演示操作，所以我们选定在 C 盘根目录下建立一个名为“SubCA”的目录作为 SubCA 的主目录。

然后，跟建立 RootCA 一样，从编译好的 OpenSSL 目录中复制“openssl.exe”、“openssl.cnf”和 ca.pl 文件到 C:\SubCA 目录下。打开 Windows 的指令行界面，进入到 C:\SubCA 目录，执行下面的指令：

```
C:\SubCA>perl ca.pl -newca
```

根据其提示步骤完成整个操作流程，就可以成功在“C:\SubCA”目录下创建一个 demoCA 目录。这样，SubCA 的目录结构就建好了，但是 SubCA 还没有合法的经过 RootCA 签名的证书和相应的密钥，紧接下面的步骤就是完成这个任务。

SubCA 的证书跟普通证书是基本相同的，除了其 X.509 v3 扩展项的某些选项的限制有一些区别。首先，为了向 RootCA 申请一个 CA 证书，需要生成一个密钥对和证书请求。因为要申请的是一个 CA 证书，所以证书请求的 X.509 v3 扩展项跟普通证书请求有区别，我们通过修改 OpenSSL 的配置文件来达到这一目的。用文本编辑器打开 C:\SubCA\openssl.cnf 文件，修改其中 v3_req 字段的 basicConstraints 选项为下面的值：

```
[v3_req]
```

……

basicConstraints = CA:TRUE

……

保存修改的 openssl. cnf 文件并退出，就可以在 C:\ SubCA 目录执行下面的指令以生成 SubCA 的密钥对和证书请求：

OpenSSL> req -new -newkey rsa:2048 -out subca_req. pem -keyout subca_key. pem -config C:\SubCA\openssl. cnf

这个指令生成了一个 2 048 位的密钥对作为 SubCA 的密钥，并同时生成了一个相应的证书请求。私钥保存在文件 subca_key. pem 中，而证书请求保存在 subca_req. pem 中。为了获得合法的 CA 证书，需要将证书请求提交给 RootCA 签发，提交的途径是多样的，可以通过网络连接提交，也可以通过远程数据库提交，而在这里，我们只需要将 subca_req. pem 文件复制到 C:\ RootCA 目录下就可以了。

然后，就要启动 RootCA 签发 SubCA 提交的证书请求了，指令行界面下在 C:\ RootCA 目录下启动 openssl. exe 程序，并执行下面的签发证书指令：

OpenSSL>ca -in subca_req. pem -out subca_cert. pem -notext -extensions v3_ca-config C:\RootCA\openssl. cnf

指令执行完后，SubCA 的证书就已经成功签发了，下面进行 SubCA 证书和私钥的配置，执行下面的指令即可：

C:\RootCA>cp C:\RootCA\subca_cert. pem C:\SubCA\demoCA\cacert. pem

C:\RootCA>cp C:\SubCA\subca_key. pem C:\SubCA\demoCA\private\cakey. pem

上述的操作只是简单地使用新签发的证书和其相应私钥替换原来 demoCA 目录下的证书文件和私钥文件，从而完成 SubCA 证书和私钥的设置。

在图 10-4 所示的结构中，SubCA 并没有下级的 CA，所以在 OpenSSL 配置文件中只要设定一个签发普通用户证书的 X. 509 v3 扩展字段即可。幸好，OpenSSL 默认的配置文件已经设置好，我们不需要费神再去配置它。

RootCA 和 SubCA 都已经配置好了，现在可以签发用户证书了。

4. 签发用户证书 A

图 10-4 的结构显示，用户证书 A 是 RootCA 签发的普通用户证书，所以我们首先在指令行界面下到 C:\ RootCA 目录下，启动 openssl. exe 程序。生成一个用户证书事实上包含三个部分：生成密钥对、生成证书请求及签发证书。在这里，我们使用 req 指令完成第一和第二个部分，使用 ca 指令完成第三部分。

普通用户证书的密钥对一般使用 1 024 位密钥就可以了，可使用下面的指令生成密钥对和证书请求：

OpenSSL> req -new -newkey rsa:1024 -out userA_req. pem -keyout userA_key. pem -config C:\RootCA\openssl. cnf

根据配置文件“C:\ RootCA \ openssl. cnf”的配置，ca 指令默认 X. 509 v3 扩展项使用的是用于签发普通用户证书的 usr_cert 字段，所以我们执行下面的 ca 指令即可签发用户证书 A：

OpenSSL>ca -in userA_req. pem -out userA_cert. pem -notext -config C:\RootCA\

openssl.cnf

执行完上述指令后，用户 A 就拥有了一个合法的 RootCA 签发的证书 A 和相应的私钥，分别保存在 C:\RootCA\userA_cert.pem 和 C:\RootCA\userA_key.pem 文件中。

5. 签发用户证书 B

用户证书 B 的申请和签发跟用户证书 A 的申请和签发基本相同，只是，因为用户证书 B 是 SubCA 签发的证书，所以应该在指令行界面下进入“C:\SubCA”目录并启动 openssl.exe 程序。然后，执行下面的两条指令完成用户证书 B 的申请和签发：

```
OpenSSL>req -new -newkey rsa:1024 -out userB_req.pem -keyout userB_key.pem
-config C:\SubCA\openssl.cnf
OpenSSL>ca -in userB_req.pem -out userB_cert.pem -notext -config C:\SubCA\
openssl.cnf
```

执行完上述两条指令，用户 B 就拥有了一个合法的 SubCA 签发的证书 B 和相应的私钥，分别保存在 C:\SubCA\userB_cert.pem 和 C:\SubCA\userB_key.pem 文件中。

到此为止，我们就完成了图 10-4 显示的 CA 目录结构的构造任务。整个过程使用的指令非常简单，只有 req 和 ca 两个，希望读者能从这整个流程中加深对 CA 和证书的体系结构的理解，那么本节的目的就达到了。

## 10.5　使 用 证 书

本节的名称或许并不是定义得很准确，事实上，在本节，我们想说的应该是使用证书前可能需要准备的一些工作，比如证书的格式转换等。当然，也可能需要查看证书的一些信息。总之，离证书的真正应用还有一些距离，证书的真正应用应该是在具体协议和软件中的使用和配置，而这些，显然不是本书要涉及的内容，虽然我们在后面介绍 SSL 指令的时候会提到一些。

### 10.5.1　X.509 证书

阅读到这里，你对 X.509 证书应该已经耳熟能详，也知道其大概有些什么内容。但是，具体如何，你或许还没有概念，让我们跟着 OpenSSL 的指令来拨开你心中最后的迷雾。

OpenSSL 的 x509 指令可以说是一个功能强大的 X.509 证书处理指令，通过该指令的介绍和使用，你对 X.509 的证书会有一个相当感性的认识，下面我们就从这个小指令入手来了解 X.509 证书。

1. x509 指令功能概述和格式

x509 指令是一个功能强大的指令，甚至超出了其名称所涉及的范围，它既可以以各种方式显示一个证书的内容，也可以对一个证书的格式进行各种有趣的转换，甚至可以签发证书等。

下面是 x509 指令的格式和选项：

OpenSSL > x509 [-inform arg] [-outform arg] [-keyform arg] [-CAform are] [-CAkeyform arg][-in arg][-out arg][-passin arg][-serial][-hash][-subject][-issuer] [-email][-purpose][-startdate][-enddate][-dates][-modulus][-fingerprint][-pubkey] [-alias][-noout][-ocspid][-trustout][-clrtrust][-clrreject][-addtrust arg][-addreject arg][-setalias arg][-days arg][-checkend arg][-signkey arg][-x509toreq][-req] [-CA arg][-CAkey arg][-CAcreateserial][-CAserial arg] [-set_serial n] [-text][-C] [-md2|md5|sha1|mdc2][-extfile arg][-extensions arg][-clrext][-nameopt arg][-engine e] [-certopt arg]

这些选项多达 49 个，更让人兴奋的是，事实上，还有很多子选项在这里没有一一列出，由此可见该指令的功能之强。下面对这些选项进行简单的分类介绍。

2. 输入和输出格式选项 inform，outform，keyform，CAform 和 CAkeyform

或许你根本就不看这段内容，因为经过前面反复的介绍，这些选项对你来说已经是非常熟悉了。不过，介绍一下，还是一本完整的书籍应该做到的。

inform 选项指定了 in 选项所指定的输入文件的格式，输入的文件可能是证书，也可能是证书请求文件（当使用了 req 选项的时候）。无论哪种文件，该选项可以接受的格式包括 PEM、DER 编码和老式的 Netscape 的编码格式。CAform 跟 inform 参数的意义相类似，不过它指定的是 CA 选项所指定的输入文件的格式。

outform 则指定了 out 选项输出的文件格式，但是，并非所有的内容都会受到这个选项的影响，仅输出文件是证书或者证书请求文件的时候会受到该选项的影响。目前，支持的格式同样包括 PEM、DER 和 NET 格式，具体的参数和意义参考表 8-2。

keyform 选项指定了 signkey 选项指定的输入私钥文件的格式，支持 PEM、DER 和 ENGINE 三种格式，其中 ENGINE 是指通过第三方驱动设备提供的私钥，具体参考表 8-2。CAkeyform 的参数意义跟 keyform 相同，不过，它相关的文件是选项 Cakey 所指定的文件。

3. 输入和输出选项 in，out，CA 和 CAkey

in 选项指定了输入的证书文件的路径，如果使用了 req 选项，则输入文件将被视为是一个证书请求文件而不是证书文件。文件的编码格式由 inform 参数决定。默认情况下输入是标准输入设备。

out 选项指定了接受输出的文件的路径，默认情况下，输出目标是标准输出设备。如果输出的是证书或者证书请求，则其输出格式由 outform 参数决定。

CA 选项指定了在签发证书或者转换证书格式的时候需要的 CA 证书文件路径。其格式由 CAform 参数指定。

CAkey 选项则指定了签发证书或者转换证书格式的时候需要使用的 CA 证书相对应的私钥文件路径，如果使用的私钥是 Engine 提供的，则是一个 Engine 的索引文件，如公钥文件等，根据具体的 Engine 私钥索引方法具体决定。普通的私钥文件格式由 CAkeyform 参数决定。

4. 输入口令选项 passin

passin 参数给出了当使用 signkey 输入私钥文件时需要提供的口令字符串。在使用

CA 证书的私钥签发证书的时候，则给出的是使用 CAkey 选项输入私钥文件时需要提供的口令字符串。一般来说，signkey 选项与 CAkey 选项是不会同时出现的。输入口令的方式可以是多样的，具体的使用方法请参考 6.6 节的详细介绍。如果没有使用该选项，而指定的私钥又要求提供解密口令，那么指令会在指令行提示用户输入口令。

5. 输出显示内容选项 serial，hash，subject，issuer，email，purpose，modulus，fingerprint，ocspid，startdate，enddate，dates，pubkey，C 和 noout

这是一些烦琐的证书显示选项，虽然如此，了解它们能够加深对数字证书内容本身的认识。

serial 选项显示证书的序列号。

hash 选项显示证书中的 HASH 值，内容是长度为 8 位的数字和字母，目前只是支持 PEM 格式的证书。

subject 选项显示证书的所有者信息。

issuer 选项显示证书的签发者信息。

email 选项显示证书中的 E-mail 地址。

purpose 选项显示证书的签发用途。

modulus 选项显示证书 RSA 密钥的模数。

fingerprint 选项显示证书的指纹信息。

ocspid 选项显示证书使用者和公钥的 OCSP 摘要值。

startdate 和 enddate 选项分别显示证书的生效时间和到期时间，而使用 dates 选项则同时显示上述两种时间。

pubkey 选项显示证书里面的公钥。

C 选项输出证书的 C 语言编码格式。

noout 选项屏蔽输出编码格式的证书或者证书请求。

6. 输出证书更加灵活的内容的选项 text 和 certopt

打开一个编码的证书文件，即便你是高级的 PKI 程序员，也不会明白这些乱七八糟的字符到底是什么意思。所以，将这些编码转换成能够读懂的信息就显得很必要，如果你使用 text 选项，你就能达到这个目的，看到所有证书的内容。

当然，很多时候，你可能并不会对 text 选项输出的所有内容都感兴趣，那么 certopt 选项可以帮助你选择所需要的东西。certopt 选项一般只有出现 text 选项的时候才使用，它的参数值很多，可以同时提供一个或者多个，如果是多个，它们之间要使用分号分开。certopt 选项还可以在指令中多次使用，这可以让你更随意地更改你的想法。certopt 的参数具体意义详见表 10-3，但参数 no_attributes 在 certopt 选项中是无效的。

7. 证书信任设置选项 trustout，addtrust，clrtrust，addreject，clrreject，setalias 和 alias

这些选项都是对可信证书扩展字段进行设置的选项。事实上，所谓的可信证书就是普通的证书加上一些扩展字段，这些字段定义了该证书可以用来做什么和不能用来做什么，并定义了证书的别名。

一般来说，在进行证书验证的过程中，证书链中至少有一个证书应该是“可信证书”，否则就不能完成验证过程。默认情况下，必须在本地保存一个可信证书，并且该可信证书应该是一个根 CA 证书，这种情况下，任何一个结束于该可信根 CA 的证书链上的证书可以用于指定用途。事实上，现在 OpenSSL 并没有要求那么严格，如果一个根证书没有可信选项，那么它就默认该证书是一个可以用于任何用途的可信证书。

目前 OpenSSL 的版本仅在根 CA 上支持可信选项设置，从而实现在最终的 CA 上统一控制证书的用途，比如允许 CA 颁发的证书用于 SSL 客户端验证或者 SSL 服务器验证等。当然，OpenSSL 的开发组承诺在以后的版本中将支持在所有证书中设置可信选项，而不仅仅限于根证书。

下面回到 x509 指令的显示世界中来，让我们看看这些选项的具体含义和用法。

trustout 选项是最容易明白的一个选项了，它会在输出的证书中加上可信的标记，从而使其成为一个可信证书。x509 指令可以接受的输入包括可信证书和普通的 X.509 证书，但是，如果没有 trustout 选项，默认情况下输出的证书都会把可信选项去掉，成为普通的 X.509 证书。当然，如果你通过其他选项设置或者修改了证书的可信选项字段，那么会自动输出一个可信证书。

设置证书的信任选项在 OpenSSL 的指令中显得很简单，使用 addtrust 选项就可以设置可信证书的用途，你可以在这里使用任何有效的参数，但是目前真正使用的参数仅仅包括 clientAuth，serverAuth 和 emailProtection。其中，clientAuth 参数表示用于 SSL 客户端验证，serverAuth 参数表示用于 SSL 服务器验证，而 emailProtection 参数则用于 S/MIME 邮件的保护。相反，clrtrust 则将清除证书中所有可信选项的用途设置。

相反，如果你想禁止证书用于某种用途，那么可以使用 addreject 选项，该选项的参数内容和 addtrust 相同。而使用 clrreject 则可以清除所有设置好的禁止用途选项。

我们猜你不会喜欢用一个证书的序列号来记忆或者标识一个证书，虽然实际的计算机系统是如此的，但毕竟人跟计算机还存在一些区别。那么我们的证书设计者充分利用了以人为本的设计理念，提供了一种定义证书的名称的方法，即证书的别名，setalias 选项就是为了让你拥有这种自由。证书别名通常是一个好记的字符串，比如“OpenSSL's Certificate”，等等，看着比“0x00cd0e”愉快多了是不是？alias 则事实上只是一个输出选项，使用该选项，输出的内容中就会包括证书的别名了。

### 8. 证书签发转换选项 signkey，clrext，req，x509toreq，days，set_serial，CAserial，CAcreateserial，extfile 和 extensions

前面介绍过 ca 指令，它是一个小型的模拟 CA，可以用来签发证书等。在 OpenSSL 中，我们还有更多的选择可以实现相同的功能，x509 指令就提供了签发证书的功能，这一节我们就来看看这些功能。当然，OpenSSL 好意提供的这些灵活选择给初学者带来很多疑惑，徒增很多混乱，但是，考虑到 OpenSSL 这些应用指令的初衷本身也是实验和演示性质的，也就能够理解了。

首先来看看 signkey 选项，这个选项指定了输入的保存私钥的文件路径。这个选项使用后，会将输入的证书文件或者证书请求文件转换成一个自签名的证书。如果输入的是证书文件，那么 x509 指令会执行下面的操作：

① 使用证书所有者名称替代证书签发者名称；

② 更改证书有效期，其中生效期从当前时间开始，结束日期取决于 days 选项，而其他扩展项和内容都会保留，如果你想清除这些过时的扩展项，那么就需要使用 clrext 选项；

③ 使用 signkey 提供的私钥重新签发证书。

但是如果在使用 signkey 的时候同时使用了 x509toreq 选项，过程则与上面不一样，signkey 提供的私钥将会被用来对生成的证书请求进行签名。

如果输入的是证书请求文件，事情就很简单，签发一个自签名证书即可，签发者名称当然跟证书持有人名称一致。

通过上面 signkey 的介绍，clrext 选项的功能大家已经有所了解，事实上 clrext 选项仅当 x509 指令通过一个证书创建另一个证书的时候有意义。也就是说，仅当输入是证书，并且使用 signkey 或者 CA 选项创建一个新的证书的时候，clrext 选项会将原来证书中的扩展项全部清除，否则，默认情况下所有扩展项都会被保留下来形成新证书的一部分。

一般情况下，x509 指令会认为输入选项 in 输入的是一个证书，如果你想输入的是一个证书请求，那么使用 req 选项就是通知 x509 指令的唯一途径。

有些情况下，一些特殊需要总是会出现的，比如将一个证书转换成证书请求，这时候 x509toreq 选项就能帮助你达到这个目的。这功能看起来或许有些奇怪，但是在证书到期续签的时候可能就有些用处了。使用这个选项的时候，需要使用 signkey 选项提供相应的私钥在生产证书请求的时候进行数字签名。

签发证书的时候需要指定有效期，days 选项就定义了有效期的长短，默认情况下，x509 指令证书签发的有效期是 30 天。

我们知道证书都有一个序列号，这个序列号就像你的身份证号码一样独特而重要，ca 指令是通过文本数据库中的 serial 文件来保存确定的。而在 x509 指令中，则可以有多种方法确定一个将要生成的证书的序列号。首先最直接的是可以使用 set_serial 选项来直接指定序列号，该方法优先级是最高的，如果使用了该方法，后面的两种方法即便在指令中使用了也会被忽略。其次是可以使用 CAserial 选项指定序列号文件来提供序列号，事实上，所谓序列号文件事实上是一个仅包含一个十六进制正整数的文件。使用 CAserial 选项后，指令会自动将该序列号文件中的整数加 1 并保存回文件，默认情况下，x509 指令的序列号文件名称为证书文件名称加上 .srl 后缀，比如输出的证书文件名为 cert.pem，那么指令会试图从文件 cert.srl 中获取序列号。最后，如果你这个文件也没有（其实你真的可以自己用文本编辑器编一个），就使用 CAcreateserial 选项生成一个证书序列号文件，该序列号文件保存的是 2，而所签发的证书序列号是 1，这时候你可以打开这个序列号文件好好研究一下了。

有时候在证书中需要加入扩展项，这时候可以使用 extfile 选项，extfile 选项指定的是一个扩展项配置文件，具体内容格式可以参考第 6 章的说明。指定扩展项定义文件之后，可能还需要使用 extensions 指定使用的扩展字段，因为一个扩展项配置文件中可能存在多个扩展字段，如果不使用该选项，那么需要加入的扩展项应该定义在文件的默认字段（未命名字段）或者默认字段中应该包含一个名为 extensions 的变量来指明扩展项所在的字段。

9. 证书显示名称格式选项 nameopt

选项 nameopt 的具体参数的含义在介绍 req 指令时已经使用表 10-4 说明，这里通过介绍怎么使用 OpenSSL 指令来显示中文证书来让读者理解这些选项的妙用。

经常有人会怀疑，OpenSSL 是不是不能正确显示中文证书，事实上，并非如此。利用 nameopt 的参数，完全可以显示所有人和签发者的中文信息，包括名称、单位和城市，等等。首先，我们需要一个能够输入中文的控制台终端，比如普通的中文 Windows 指令行界面。在该控制台可以按照正常的 req 指令生成一个证书，其用户信息用中文填写完成，从而得到一个中文的证书。如果你直接使用下面的指令，

```
OpenSSL>x509 -in cncert.pem -subject -out text.txt
```

结果很让人沮丧，因为你看到的将是晦涩难懂的输出：

```
subject = /C = CN/ST = \xB1\xB1\xBE\xA9\xD6\xD0\xB9\xFA/L = \xB1\xB1\xBE\xA9/O = \xBA\xA3\xB5\xED/OU = OpenSSL/CN = \xCD\xF5\xD6\xBE\xBA\xA3/emailAddress = wangzhihai@opnessl.cn
```

但是如果使用下面的指令，

```
OpenSSL>x509 -in cncert.pem -subject -nameopt RFC2253,-utf8,-esc_msb
```

结果就可能让你心满意足，甚至兴奋异常，因为你看到了所输入的中文信息：

```
subject = emailAddress = wangzhihai@opnessl.cn,CN = 王志海,OU = OpenSSL,O = 海淀,L = 北京,ST = 北京中国,C = CN
```

对比上面的指令可以看出，后面的指令只是多了几个 nameopt 选项，就实现了我们显示中文的目标，很简单不是吗？相信这时候你会开始仔细研读表 10-4 了，多做试验，收获会更加丰富。

10. 验证证书是否过期选项 checkend

这个选项非常简单实用，目的就是验证输入的证书是否在有效期内。如果本选项返回 0，说明输入的证书没有过期，如果返回是 1，则说明证书过期了。

11. 信息摘要算法选择选项 md5，md2，sha1 和 mdc2

对于这些名词，大家应该已经非常熟悉了，它们是进行信息摘要时可能用到的四种算法，在前面章节有对于这几种算法的详细介绍。信息摘要算法在证书中主要是用于 CA 颁发证书签名的时候，具体到 x509 指令来说，如果信息摘要算法变化了，对于前面介绍的-fingerprint，-signkey 和-CA 等证书信息的输入输出选项的结果都会有影响。在 x509 指令里，如果没有特别指定，默认使用的是 MD5 算法。由于 DSA 算法固定使用 SHA1 算法作为数字签名的信息摘要算法，所以如果你使用 DSA 密钥进行数字签名，则 SHA1 的使用将是固定不变的，这些选项也就没有任何使用的必要。

12. engine 选项

engine 选项告诉 x509 指令尽可能使用 Engine 设备提供的加密函数和算法替代 OpenSSL 本身的函数。对于 x509 指令来说，可能替代的算法包括：签名时使用的信息摘要算法、签名时使用的私钥加密算法、验证证书请求中的数字签名时使用的公钥解密算法、随机数函数及私钥数据等。

13. 简单的例子

x509 指令的选项如此之多，这里就不一一举例。下面给出一些简单常用的指令例子，让大家对指令的选项能够进一步加深认识。当然，大家可以根据自己的兴趣运用 x509 指令对证书进行操作，多做实验对于掌握指令是很有帮助的。

首先，我们来查看一个证书的内容（这里我们假设已经生成了证书，生成证书的步骤本章的前面部分有详细介绍），我们将其直接输出到屏幕上：

```
OpenSSL>x509 -in demoCA\cert.pem -noout -text
```

显示证书序列号：

```
OpenSSL>x509 -in demoCA\cert.pem -noout -serial
```

显示证书 HASH 值：

```
OpenSSL>x509 -in demoCA\cert.pem -noout -hash
```

显示 MD5 指纹信息：

```
OpenSSL>x509 -in demoCA\cert.pem -noout -fingerprint
```

如果想转换证书格式，可以使用下面指令（从 PEM 转到 DER 格式）：

```
OpenSSL>x509 -in demoCA\cert.pem -inform PEM -out demoCA\cert.der -outform DER
```

如果你想把一个证书换个根 CA，又不想重新生成一个证书请求，那么就可以用下面这个指令把证书转换为证书请求：

```
OpenSSL>x509 -x509toreq -in demoCA\cert.pem -out req.pem -signkey key.pem
```

x509 指令的相关介绍到此告一段落，但是，事实上要用好 x509 指令，有更多的东西需要去挖掘，尤其是证书扩展项和其在不同的应用模式下的具体内容。如果你需要进一步挖掘 x509 指令的含义，研读 OpenSSL 的帮助文档和源代码是一个不错的办法，此外，多做实验这也是一个非常有效的办法。

## 10.5.2　CRL

CA 签发的证书都是有有效期限制的，证书到了有效期就会自动作废并需要重新颁发，但难道所有证书都是到了有效期之后才会无效的？答案显然是否定的。如果证书的持有人离开组织，或者其私钥已泄露，或者如果其他一些与安全相关的事件规定它不再需要将证书视为“有效”，则 CA 的管理员必须吊销证书。当证书被 CA 吊销时，它将被添加到该 CA 的证书吊销名单中，CA 中心必须定期发布更新吊销列表（CRL），以保证所有 CA 相关的应用都能及时得知证书的有效性信息。所谓 CRL，事实上就是一系列已经被吊销的证书序列号组成的文件，并且为了确保该文件的真实性，该文件经过了 CA 的合法签名。

一般来说，CRL 都具备有效期限制，其有效期由 CA 决定，可能是一天、一周或者一个月等，当然，有效期越长，其跟实际情况相差的可能性越大。

CRL 的颁发和公布都是由 CA 定期进行的，发布的方式是多种多样的，可以是在网上发布或者通过 E-mail 发布，无关紧要。CRL 的使用者是所有使用该 CA 签发证书的系统和用户，他们需要随时获取最新的 CRL，并在进行证书验证的时候查看该 CRL，确定接收到的证书不在该 CRL 列表内（如果在该列表内，则证书就是无效的了）。

### 1. crl 指令功能概述和格式

OpenSSL 提供了 crl 指令用于显示、处理和验证 CRL 文件信息，而发布 CRL 文件则在前面介绍的 CA 指令中可以实现。

下面是 crl 指令的格式和选项：

```
Openssl > crl [-inform PEM | DER] [-outform PEM | DER] [-text] [-in filename] [-out filename][-noout][-hash][-issuer][-lastupdate][-nextupdate][-CAfile file][-CApath dir][-nameopt nopt]
```

### 2. 输入和输出格式选项 inform 和 outform

inform 和 outform 选项指定了输入和输出文件格式，相对应于 in 和 out 选项，可以接受的格式包括 PEM 和 DER 编码格式，默认接受的格式是 PEM。

### 3. 输入和输出选项 in 和 out

in 选项指定了输入的 CRL 文件的路径，文件的编码格式由 inform 参数决定。默认情况下输入的是标准输入设备。

out 选项指定了接受输出的文件的路径，默认情况下，输出目标是标准输出设备。其输出格式由 outform 参数决定。

### 4. 输出显示内容选项 hash，issuer，noout，text，lastupdate，nextupdate 和 date

hash 选项显示 CRL 文件中 CA 签名的 HASH 值。

issuer 选项显示 CRL 文件签发者信息，一般来说，就是 CA 的信息。

noout 选项要求指令不输出编码 CRL 的内容。

text 选项要求指令输出 CRL 文件的可理解的形式，也就是说，能够让你一看就懂的形式。

lastupdate 和 nextupdate 选项分别显示 CRL 文件的最近更新时间和下次更新时间，而使用 date 选项则同时显示上述两种时间。事实上，所谓下次更新时间，也就是本 CRL 文件失效的时间。

### 5. 输出字符编码选项 nameopt

nameopt 选项指定了如何显示主体名称和签发者名称，主要用于显示名称中不同编码的内容。没有使用该选项时，默认使用的是“oneline”参数。该选项可以同时使用多个参数，每个参数间使用“,”分开即可。具体的参数可以参照表 10-4。

### 6. CA 文件和目录选项 CAfile 和 CApath

CAfile 指定一个 CA 证书文件，该 CA 证书用于核实 CRL 中的签名是否正确和有效。

CApath 则告诉指令从一个目录中查找有效的 CA 证书，此目录必须是标准证书目录，所谓标准目录，即 OpenSSL 要求的基于文件 HASH 值为文件名排序的一个目录，详见 x509 指令的介绍。

### 7. 简单的例子

把 CRL 文件的格式从 PEM 转化成 DER：

```
OpenSSL>crl -in crl.pem -outform DER -out crl.der
```

输出 DER 格式编码的证书吊销列表的文本：

```
OpenSSL>crl -in crl.der -text -noout
```

### 10.5.3　PKCS#12 证书

如果你已经阅读完本章的前面部分，相信对于 PKCS#12 证书已经有了一定的了解，它的内容并不复杂，而且由于 Microsoft 的强大，我们必须正视它在我们现实生活中存在的份量。事实上，除了 Microsoft 的产品，Netscape 的浏览器也需要 PKCS#12 格式证书的支持。

基于此，OpenSSL 除了在一般的指令中对 PKCS#12 证书提供支持，还开发了专门用于处理 PKCS#12 证书的指令 pkcs12，用于 PCKS#12 证书的生成、解释和验证等。

#### 1. pkcs12 指令功能概述和格式

pkcs12 指令功能总体来说可以分成创建 PKCS#12 证书和解释 PKCS#12 证书。创建 PKCS#12 证书即将普通的 X.509 证书和私钥封装成 PKCS#12 证书；解释证书可以将 PKCS#12 证书转换成 X.509 证书，并提取出相应的私钥，并可以输出其他必要的信息。

一般来说，PKCS#12 证书的私钥是经过加密的，密钥由用户提供的口令产生。所以，在使用 pkcs12 指令编码或者解码 PKCS#12 证书的时候，一般都会要求用户输入密钥口令。

下面是 pkcs12 指令的格式和选项：

```
OpenSSL > pkcs12 [-export] [-chain] [-inkey filename] [-certfile filename] [-CApath arg] [-CAfile filename] [-name name] [-caname name] [-in filename] [-out filename] [-noout] [-nomacver] [-nocerts] [-clcerts] [-cacerts] [-nokeys] [-info] [-des] [-des3] [-idea] [-aes128|-aes192|-aes256] [-nodes] [-noiter] [-maciter] [-twopass] [-descert] [-certpbe alg] [-keypbe alg] [-keyex] [-keysig] [-password arg] [-passin arg] [-passout arg] [-engine e] [-rand file(s)]
```

跟 OpenSSL 的其他指令一样，这些繁多的参数让人望而生畏。不过，这些选项中常用的并不是很多，有些可能对你来说永远也用不着，除非你刻意去使用它们。比如，当你需要把 PKCS#12 证书转化成 X.509 证书时，仅仅用-in 和-out 选项就可以完成了。如果你想把 X.509 证书和其私钥封装成 PKCS#12 证书，再增加-export 一个选项即可轻松完成。当然，如果你需要更加具体复杂的要求，你需要了解的指令选项也会多一些。

在下面的介绍中，会先介绍一些在创建 PKCS#12 证书和解释 PKCS#12 证书时都需要用到的选项，然后介绍解释 PKCS#12 证书用到的一些选项，最后介绍创建 PKCS#12 证书需要用到的选项。

#### 2. 输入和输出文件选项 in 和 out

默认情况下，输入和输出指定的都是标准输入输出设备。

在解释 PKCS#12 证书的时候，即没有使用-export 选项的时候，in 选项的参数默认情况下是一个 PKCS#12 格式的文件，它将被解释为单独的证书和私钥，并总是以 PEM

格式写入到 out 选项指定的输出文件中。

在创建 PKCS#12 证书的时候，out 选项参数输出的是一个 PKCS#12 格式的证书文件。in 选项输入的文件是包含一系列证书和私钥的文件。这些证书和私钥必须是 PEM 编码的，它们在该文件中的顺序不受限制，但要求至少有一个私钥和其相对应的证书存在。如果存在其他附加的证书，则这些证书都会被封装到输出的 PKCS#12 证书文件中。

3. 输入和输出口令选项 passin，passout，password

passin 选项指定了使用输入文件的时候需要的口令源。如果是解释 PKCS#12 证书，则是该证书文件解密口令源；如果是创建 PKCS#12 证书，则是读取私钥文件时候可能需要的口令源。

passout 选项则指定了输出文件时进行加密的口令源。如果是解释 PKCS#12 证书，则是输出私钥加密口令源；如果是创建 PKCS#12 证书，则是加密 PKCS#12 证书文件时可能需要的口令源。

password 选项提供的总是 PKCS#12 证书的加密口令源，即在解释 PKCS#12 证书时，其参数为读取 PKCS#12 证书需要的口令源；在创建 PKCS#12 证书的时候，其参数为加密 PKCS#12 证书需要的口令源。

口令源的形式多种多样，可以是直接输入，也可以是从文件、环境变量、标准输入输出等获取，具体见本书 6.6 节。

4. 输出内容选项 noout，clcerts，cacerts，nocerts，nokeys 和 info

这些选项都是在解释 PKCS#12 证书时使用的。

noout 选项告诉指令不要输出任何私钥和证书信息到输出文件中。

clcerts 选项告诉指令只输出客户证书（不是 CA 证书）到输出文件中。

cacerts 选项告诉指令只输出 CA 证书到输出文件中。

nocerts 选项告诉指令不要输出任何证书到输出文件中。

nokeys 选项告诉指令不要输出私钥到输出文件中。

info 选项告诉指令输出一些其他 PKCS#12 证书文件相关的信息，如构成、算法及其迭代次数等。

5. 加密算法选项 des，des3，idea，aes128，aes192，aes256 和 nodes

这些选项仅在解释 PKCS#12 证书的时候使用。

des，des3，idea，aes128，aes192 和 aes256 选项用于指定在输出私钥时，对私钥进行加密使用的算法。默认情况下，pkcs12 指令会使用 des3 作为加密算法。

nodes 选项则告诉指令对私钥不进行加密处理。

6. nomacver 和 twopass

这两个选项使用较少，也仅在解释 PKCS#12 证书的时候使用。

nomacver 选项告诉指令在读取 PKCS#12 证书之前，不需要对文件进行 MAC 的完整性验证。

twopass 选项告诉指令提示输入完整性验证和加密需要的密码源。大多数 PKCS#12 软件实现都假设这两个密码是一样的，不需要使用这个选项。

7. 创建 PKCS#12 证书选项 export

该选项在创建 PKCS#12 证书的时候必须使用，告诉指令这时候需要创建一个 PKCS#12 证书，而不再是解释一个已有的 PKCS#12 证书。

8. 证书内容输入选项 inkey 和 certfile

这两个选项在创建 PKCS#12 证书的时候使用。

inkey 选项指定一个包含了私钥的文件，指令将从这个文件中读取创建 PKCS#12 证书需要的私钥。如果该选项没有使用，则在 in 参数指定的输入文件中必须包含一个私钥。

certfile 选项指定向创建的 pkcs#12 证书中封装额外的证书，这些证书存储在该选项的参数文件中，可以是一个或者多个。

9. 证书别名选项 name 和 caname

这两个选项在创建 PKCS#12 证书的时候使用。

name 选项指定了证书和私钥的一个好记的别名，该名称通常显示在需要导入该 PKCS#12 证书文件的软件中的证书列表中。

caname 选项用于为其他证书指定一个好记的别名，该选项可以使用多次以为多个证书指定名称，名称的顺序跟证书出现的顺序保持一致即可。该选项一般仅在 IE 里面有效，在 Netscape 软件中则通常被忽视。

10. 证书加密算法选项 descert，keypbe 和 certpbe

这三个选项在创建 PKCS#12 证书的时候使用。

descert 选项告诉指令使用 3DES 算法加密证书，在默认情况下，指令使用 3DES 算法加密私钥，但是使用 40 位的 RC2 算法加密证书。

keypbe 和 certpbe 选项分别指定对密钥和证书进行加密的算法。虽然，所有的 PKCS#5v1.5 或者 PKCS#12 算法都是支持的，但是 OpenSSL 还是建议你只用 PKCS#12 标准中的加密算法。这些算法包括 DES，3DES，RC2 和 RC4 等。这些参数的形式诸如：

```
-keypbe PBE-SHA1-RC2-40
```

支持的具体算法和其参数形式见表 11-1。

11. 密钥用途选项 keyex 和 keysig

这两个选项在创建 PKCS#12 证书的时候使用，而且仅在 Microsoft 的 IE 和其他 Microsoft 软件中才有效。

keyex 选项指定密钥和证书用于密钥交换，keysig 则指定密钥和证书用于数字签名。如果仅使用 keysig 选项，则密钥仅能用于进行签字。仅用于签名的密钥可以用来进行 S/MIME 签名、验证码（ActiveX 控制签名）和 SSL 客户端验证。需要注意的是，由于 OpenSSL 本身存在的 BUG，只有 IE5.0 和其后的版本才能够支持仅用于签名用途的密钥进行 SSL 客户端验证。

12. 迭代次数选项 nomaciter，noiter 和 maciter

这几个选项在创建 PKCS#12 证书的时候使用，而且仅在你需要兼容 Microsoft 的 IE4.0 的时候需要。

因为算法通常使用普通口令产生密钥，为了降低受到字典攻击的可能性，可以将口令

进行多次迭代从而产生复杂的口令。这通常以重复密码算法中的一部分步骤和降低速度为代价。MAC虽然一般用于验证文件的完整性，但因为私钥和证书也是使用与其相同的口令，所以MAC同样存在被攻击的可能性。通常情况下，MAC和算法的迭代次数都设置为2 048次，如果使用了nomaciter和noiter选项，迭代次数将分别降为1。显然，这两个选项大大降低了文件的安全性，所以，除非不得已，建议不要使用这两个选项。除了IE4.0不支持MAC迭代外，大部分软件都支持多次MAC和算法迭代。

maciter目前来说意义不大，指明需要使用MAC迭代，目前版本这已经是默认的，只是为了兼容以前的版本该选项才没有取消。

13. 其他选项chain和rand

这两个选项同样在创建PKCS#12证书的时候使用。

chain选项告诉指令要建立一个完整的用户证书相关的证书链保存在PKCS#12文件中。指令会从一个标准的CA证书容器中寻找建立证书链需要的证书，如果失败，则视为发生致命性错误。这个功能当然比较特殊，不过，很多时候，验证方确实需要被验证方提供完整的证书链来进行验证操作，尤其当多级CA存在的情况下，这种要求就是常见的了。

随机数文件选项rand提供了产生随机数的参考种子文件。如果没有提供该选项，那么程序会从标准输出设备的状态读取信息作为随机数种子。随机数文件可以为任意类型的文件。rand选项可以支持多个文件，用分隔符隔开即可，Windows平台使用“;”分隔，OpenVMS系统使用“,”分隔，其他系统都使用“;”分隔。

14. 简单的例子

下面给出一些简单的例子，帮助大家进一步熟悉pkcs12指令的常用选项。当然要掌握pkcs12的全部指令，还需要自己进行更多的实践和操作。

首先，我们要来创建一个PKCS#12证书，而且还可以根据需要为它起个容易记的名字。假设已经生成了X.509证书及其私钥。

```
OpenSSL>pkcs12 -export -in Mycert.pem -inkey MyKey.pem -out file.p12 -name"MyCert"
```

如果你觉得封装了一个证书还满足不了需求，那么你可以添加额外的证书，指令如下：

```
OpenSSL>pkcs12 -export -in Mycert.pem -inkey MyKey.pem -out file.p12 -name"MyCert"-certfile Mycert1.pem
```

有了PKCS#12证书，你可以查看它的结构、使用的算法等信息，以及反复加密的次数：

```
OpenSSL>pkcs12 -in file.p12 -info -noout
```

当然如果你想再重新看一遍证书及其私钥的信息，而且不想从源文件去看，或者说源文件已经丢失，那么分解PKCS#12证书可以满足你的要求：

```
OpenSSL>pkcs12 -in file.p12 -out file.pem
```

如果你只是需要客户证书信息，你可以单独把它输出到一个文件中去：

```
OpenSSL>pkcs12 -in file.p12 -clcerts -out file.pem
```

在输出证书私钥时，你更希望是没有加密的私钥，虽然不安全，但是仍然可以操作：

```
OpenSSL>pkcs12 -in file.p12 -out file.pem -nodes
```

### 10.5.4　PKCS#7 证书

通过前面介绍 PKCS#7 标准证书出现的背景，相信你对使用 PKCS#7 证书的好处有了一定的了解。PKCS#7 格式的证书可以封装 X.509 标准证书、相关证书链上的 CA 证书和 CRL，这样就可以直接把 PKCS#7 证书发给验证方验证，免去了把以上的验证内容一个一个发给接收方的烦琐了。

微软对 PKCS#7 证书的支持依赖文件后缀名 p7，当然，OpenSSL 则是从来不关心后缀名的。OpenSSL 对 PKCS#7 协议提供了将普通用户证书、CA 证书和 CRL 封装成 PKCS#7 格式证书的指令 crl2pkcs7，此外还提供了 pkcs7 指令来解释 PKCS#7 格式证书，下面就具体介绍一下 crl2pkcs7 和 pkcs7 指令。

#### 1. crl2pkcs7 指令功能概述和指令格式

crl2pkcs7 指令名不符实，并不仅仅是把 CRL 转换成 PKCS#7 证书。其实，在 OpenSSL 中，该指令是唯一可以生产 PKCS#7 证书的指令，主要用于把 CRL 和一个或多个证书封装成一个 PKCS#7 格式的证书。并且，CRL 是一个可供选择项，即 PKCS#7 证书里面不一定需要一个 CRL。下面具体介绍该指令的用法。

crl2pkcs7 指令的格式如下：

```
OpenSSL>crl2pkcs7[-inform PEM|DER][-outform PEM|DER][-in filename][-out filename]
[-certfile filename][-nocrl]
```

看完 crl2pkcs7 指令的格式，相信你对掌握此指令将信心十足了，因为相比于前面的指令，这个指令实在是太简单了。不过，下面仍要对指令中具体的选项进行介绍。

**(1) 输入和输出格式选项 inform 和 outform**

inform 选项指出了输入的 CRL 文件的编码格式。目前只支持两种格式：PEM 和 DER。

outform 选项指出了输出的 PKCS#7 文件的编码格式。目前支持 PEM 和 DER 两种格式。

**(2) 输入和输出选项 in，out 和 certfile**

in 选项指定了存储输入 CRL 的文件名。此选项如果不指定，默认采用标准输入。

out 选项指定了存储输出 PKCS#7 证书的文件名，默认采用标准输出。

certfile 选项指定的文件可以包含一个或多个证书，文件中的证书将会全部被封装到 PKCS#7 证书中，这个选项也可以多次使用以加入多个证书。需要注意的是，所有的证书都应该是 PEM 格式编码的。

**(3) 不包含 CRL 的 PKCS#7 证书选项 nocrl**

nocrl 选项指定在输出文件中不包含 CRL 文件。一般情况下，在输出的 PKCS#7 文件中包含 CRL。指定此选项后，操作过程中将不会去读输入文件中的 CRL。

**(4) 简单的例子**

crl2pkcs7 指令最常用的用法是用来生成 PKCS#7 证书。下面就是生成 PKCS#7 证书的简单例子。

首先生成一个包含证书及CRL的PKCS#7证书：

```
OpenSSL>crl2pkcs7 -in crl.pem -certfile cert.pem -out p7.pem
```

接下来生成一个DER格式的PKCS#7证书，其中包含了多个不同的证书，不包含CRL：

```
OpenSSL> crl2pkcs7 -nocrl -certfile newcert.pem -certfile demoCA/cacert.pem
-outform DER -out p7.der
```

通常，你看到的一个PEM格式的PKCS#7证书的首行和末行格式为：

—BEGIN PKCS7—

—END PKCS7—

为了与一些CA兼容，PKCS#7证书首行和末行也可以写成：

—BEGIN CERTIFICATE—

—END CERTIFICATE—

2. pkcs7指令功能概述和指令格式

好了，现在我们通过crl2pkcs7指令已经可以生成一个有意义的PKCS#7证书了，有些时候，你或许想看看这个包含多个X.509证书和CRL信息的文件里面的内容，OpenSSL提供pkcs7指令满足你的这种愿望。pkcs7指令的格式如下：

```
OpenSSL>pkcs7[-inform arg][-outform arg][-in arg][-out arg] [-print_certs]
[-text][-noout][-engine e]
```

**(1) 输入和输出格式选项 inform 和 outform**

inform和outform选项指定了in和out选项相关文件的格式，可以是PEM或者DER格式。

**(2) 输入和输出选项 in 和 out**

in选项指定了输入的证书文件的路径，文件的编码格式由inform参数决定。默认情况下输入的是标准输入设备。

out选项指定了接受输出的文件的路径，文件的编码格式由outform参数决定。默认情况下，输出目标是标准输出设备。

**(3) 输出显示内容选项 ptint_certs，text 和 noout**

print_certs选项要求输出该PKCS#7证书文件内的所有证书和CRL，该选项逐行输出所有证书的主题名和签名者。

text选项要求输出证书的所有内容。

noout选项要求不要输出编码结构的PKCS#7证书。

**(4) engine 选项**

engine选项指定后，所指定的Engine（通过唯一的id指定）会使用Engine设备指定的库或者硬件设备相关接口来处理本指令的一些操作。

**(5) 简单的例子**

pkcs7指令使用起来比较容易，选项少且也都是较常见的。下面就举两个简单的例子，让读者对该指令有一个更清晰的认识。

把PKCS#7证书从PEM格式转化成DER格式：

```
OpenSSL>pkcs7 -in file.pem -outform DER -out file.der
```

输出文件中所有的证书：

```
OpenSSL>pkcs7 -in file.pem -print_certs -out certs.pem
```

# 10.6　验 证 证 书

通过前面章节的介绍，我们对证书的格式和证书的生成都已经有了一定的了解，相信这时候你明白了一个道理：证书不过是一个数字化世界的身份证而已，不是很复杂的东西。那么，当你拿到一个数字证书的时候，或者要使用一个数字证书的时候，首先想到的当然是验证这个证书的真伪，这便是证书的验证。在这一节，我们将着重介绍如何验证一个证书。

## 10.6.1　验证证书的过程

假设你是本书的忠实读者并阅读了前面所有的内容，对于证书的结构，就已经不再陌生。证书的结构中的关键内容包括：序列号、公钥、用户名称、签发者、CA 签名和其他一些附属信息等。证书验证过程就是依赖于这些信息和公钥对应的私钥进行。通常的证书验证过程包括以下要点：

① 确认证书内容是正确的和完整的，没有被篡改，CA 签名是正确的；

② 确认证书是有效的，在有效期内并且没有被吊销（CRL 中没有该证书序列号）；

③ 确认 CA 证书是可以被信任的证书，如果 CA 证书不是根证书，还要继续对 CA 证书进行验证；

④ 通过与用户的交互，基于证书中的公钥和公开密钥算法确认用户的身份。

其中，确认用户身份的过程是基于公开密钥算法的，基本过程如下：

① 被验证方发送自己的用户证书给验证方；

② 验证方提取证书中的公钥，并产生一个随机数 Rp，使用该公钥加密该随机数得到加密的随机数 Re 并发给被验证方；

③ 被验证方使用自己的私钥解密 Re 得到 Rep 并发回给验证方；

④ 验证方对比 Rp 和 Rep，如果一致，则验证通过，否则，验证不通过。

从上述的过程可以知道，因为证书中公钥对应的私钥仅真正的证书所有者才可能持有（理论上是这样的，当然现实的系统和社会中不尽如此），所以如果被验证方能够正确解密使用该证书上的公钥加密的信息，即可认为被验证方确实是证书的持有人，也就可以确定其身份。

无论何种类型的证书，其验证过程都是基本一致的，当然，根据具体的情况，其验证协议和要求提供的材料可能有一些区别。关于证书验证的更多情况，也可以参考前面 10.1.4 节内容。不过，在这一节，我们关心的是验证证书的有效性，而不对证书持有人的身份真实性进行验证。

OpenSSL 提供了证书验证的一些指令，包括普通验证指令和一种标准的 OCSP 协议处理指令。了解这些指令的用途，对于理解证书验证的过程是很有帮助的，下面就让我们再度进入到具体的指令中去。

## 10.6.2 verify 指令介绍

### 1. 功能概述和指令格式

verify 指令提供了证书或者说证书链的验证功能，下面是 verify 指令的选项和格式：

OpenSSL>verify[-CApath directory][-CAfile file][-purpose purpose][-untrusted file][-issuer_checks][-crl_check][-crl_check_all][-ignore_critical][-verbose][-engine e][certificates]

在具体介绍 verify 指令的参数之前，先不要着急，让我们整体看看这个指令是按照什么样的顺序进行证书验证的。

当然，verify 指令的验证过程跟我们前面描述的验证过程基本是一致的，唯一存在的区别是：无论在验证过程中遇到什么错误，虽然通常情况下这时候应该算作验证失败，应该停止验证过程了，但是我们 OpenSSL 中的 verify 指令还是不辞劳苦地将整个验证过程执行完。这样的好处是可以告诉用户在整个证书链的验证过程中存在的所有问题，而不需要用户一个一个问题进行逐步排除。

为了执行验证过程，verify 指令首先做的事情是试图从提供的所有证书里建立一个证书链，证书链从提供的用户证书开始，而结尾应该是一个根 CA 证书。证书链建立的规则是寻找当前证书的签发者证书，如果一个证书的签发者就是其自己，那么该证书就是根证书。如果 verify 指令不能成功建立一个证书链，那么就认为验证失败。

寻找当前证书的签发者证书这个过程本身也是一个复杂的过程。在 OpenSSL 早期版本中（0.9.5a），第一个找到的持有者名称跟当前证书签发者名称一致的证书，即被认为是当前证书的签发者证书。而在 0.9.6 和后面的版本中，所有找到的持有者名称与当前证书签发者名称一致的证书，都被认为是可能的签发者证书，需要进行进一步的验证来确定。也就是说，除了前面的条件，当前证书的密钥标识字段内容必须跟候选签发者证书的密钥标识、签发者和序列号等相匹配，此外，候选签发者证书的密钥用途扩展字段也应该包含允许签发证书的功能。

建立证书链的时候，程序首先查找不信任列表，如果找不到当前证书的签发者证书，则开始在信任列表中查找证书。而根 CA 证书总是在信任列表中被找到的。如果需要验证的证书本身就是一个根证书，那么必须在信任证书列表中找到一个完全一致的证书才能认为是验证通过。

建立一个完整证书链后，verify 程序开始检查每一个非信任证书的扩展项是否跟 purpose 选项提供的参数内容相一致。当然，如果 purpose 选项没有出现，则在本步骤 verify 指令将不进行任何操作。需要验证的证书或者“叶子”证书其扩展项必须跟要求验证的 purpose 参数一致，其他 CA 证书也必须是有效的证书。至于 purpose 的具体内容和含义，请参考下面将要介绍的本指令的 purpose 参数介绍。

完成上述步骤后，verify 开始检查根 CA 证书的信任设置，以确定其是否支持 purpose 的要求。OpenSSL 为了兼容以前的版本，假定如果 CA 证书没有任何信任设置，则默认为支持任何用途。

如果你使用的是 0.9.7 以后的版本，很幸运，OpenSSL 将支持 CRL 验证的操作，这时候 verify 将根据 CRL 检查整个证书链的每个证书是否有效，即没有被管理员吊销。

最后，verify 程序将检查整个证书链上所有证书的有效期，根据当前的系统时间进行检测。需要注意的是，各个证书的数字签名也是在这个最后的时刻进行检验的。

通过上面所有的验证步骤，证书验证才算成功，否则，任何一步的失败，都意味着证书验证不通过。

2. CA 证书选项 CApath 和 CAfile

CApath 选项用来指定我们信任的 CA 的证书存放目录。但是这些证书的名称应该是下面这样的格式：xxxxxxxx.0（xxxxxxxx 代表证书的哈希值，可以参看 x509 指令的 -hash选项）。

CAfile 选项用来指定我们信任的 CA 证书文件，其中可以包含多个 CA 证书。目前只支持 PEM 格式的文件。

3. 非信任证书选项 untrusted

untrusted 选项用来指定非信任证书文件，同样，该文件可以包含多个 PEM 格式的证书。所谓非信任的证书文件，一般是指在构造证书链的时候需要使用的 CA 证书，但一般都不是根证书。你千万不要以为这是不被信任的证书。

通常，CRL 也可以通过这个选项指定的文件输入到 verify 验证过程中。

4. 证书用途选项 purpose

此选项主要验证证书的扩展字段，显示证书的用途。如果没有设置此选项，那么 verify 将不会对证书链进行验证操作。目前，purpose 支持的参数包括：sslclient，sslserver，nssslserver，smimesign 和 smimeencrypt。更多的关于本参数的内容请参考本书第 6.1 节内容。

5. 验证选项 issuer_checks，crl_check，crl_check_all 和 ignore_critical

issuer_checks 选项告诉指令输出在查找签发者证书过程中的详细信息，从而可以知道每个候选签发者证书被拒绝的原因。

crl_check 选项告诉指令检查被验证证书的 CRL，如果不幸 CRL 中存在被验证证书的序列号，则证书将不能通过验证。CRL 通过 untrusted 选项的参数输入。

crl_check_all 选项告诉指令检查证书链中所有证书的 CRL，即不仅仅要对被验证证书是否被吊销进行验证，还要对整个证书链中其他证书是否被吊销进行验证。

ignore_critical 选项告诉指令可以忽略一些比较少用的扩展项，即如果该扩展项指令没有处理的话，则忽略。

6. 如何输入被验证的证书?

在 verify 指令中，在指令所有参数最后面输入要验证的证书文件名即可指定需要验证的证书。如果有多个证书文件需要验证，可以以空格为间隔一起列在后面，但是这些证书都应该是 PEM 格式的数字证书。如果不提供证书文件，则指令将试图从标准输入中读取被验证的证书。

使用 verbose 选项，指令将把验证过程中的所有详细过程输出。

engine 选项用来指定可以用于替代 OpenSSL 的函数库的其他算法库或者硬件，详见前面章节。

7. 诊断证书验证结果

如果验证操作有问题，OpenSSL 提供了一些输出来让我们弄明白到底出现了什么问

题，虽然这些信息看起来有些难懂，不过还是有用的。

诊断信息输出格式如下：

```
server.pem:/C = AU/ST = Queensland/O = CryptSoft Pty Ltd/CN = Test CA(1024 bit)
error 24 at 1 depth lookup:invalid CA certificate
```

第一行说明哪个证书文件出问题，后面是其证书的主题。

第二行说明错误号，验证的深度，并给出错误的解释，虽然这些解释不甚明了。其中，验证深度是从 0 开始计算的。

表 10-6 给出了错误号及其描述的详细说明。需要注意的是，有的错误虽然有定义，但没用使用，这里用 unused 标识。

表 **10-6** **verify** 指令验证错误诊断代码

| 序号 | 错误代码 | 错误说明 |
|---|---|---|
| 1 | X509_V_OK | 验证操作通过 |
| 2 | X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT | 找不到当前证书的签发者证书 |
| 3 | X509_V_ERR_UNABLE_TO_GET_CRL | 没找到有效的 CRL |
| 4 | X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE | 无法解密数字证书签名，这可能意味着证书签名跟期望的不一致，仅对 RSA 算法有意义 |
| 5 | X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE | 无法解密 CRL 的签名 |
| 6 | X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY | 无法提取签发者的公钥 |
| 7 | X509_V_ERR_CERT_SIGNATURE_FAILURE | 证书中的数字签名无效 |
| 8 | X509_V_ERR_CRL_SIGNATURE_FAILURE | CRL 中的数字签名无效 |
| 9 | X509_V_ERR_CERT_NOT_YET_VALID | 证书还没有到生效期 |
| 10 | X509_V_ERR_CRL_NOT_YET_VALID | CRL 还没有到生效期 |
| 11 | X509_V_ERR_CERT_HAS_EXPIRED | 证书已经过期 |
| 12 | X509_V_ERR_CRL_HAS_EXPIRED | CRL 已经过期 |
| 13 | X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD | 证书中生效时间格式不正确 |
| 14 | X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD | 证书中到期时间格式不正确 |
| 15 | X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD | CRL 中最近更新时间格式不正确 |
| 16 | X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD | CRL 中下次更新时间格式不正确 |
| 17 | X509_V_ERR_OUT_OF_MEM | 内存溢出（理论上不应该发生） |
| 18 | X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT | 被验证证书是自签名证书，但是信任列表中不存在该证书 |
| 19 | X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN | 证书链中的非根证书都能够找到，但是根证书不在信任证书列表中 |
| 20 | X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY | 不能找到需要的签发者证书，这通常意味着提供的信任证书列表不完整 |

续表

| 序号 | 错误代码 | 错误说明 |
|---|---|---|
| 21 | X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE | 不能验证最终证书，这通常是因为证书链中只有一个被验证的证书，没有根证书 |
| 22 | X509_V_ERR_CERT_CHAIN_TOO_LONG（unused） | 证书链太长 |
| 23 | X509_V_ERR_CERT_REVOKED | 证书已经被吊销 |
| 24 | X509_V_ERR_INVALID_CA | 无效的 CA 证书，可能因为不是 CA 证书，也可能因为它的扩展项跟验证目标不一致 |
| 25 | X509_V_ERR_PATH_LENGTH_EXCEEDED | 超出证书路径长度限制 |
| 26 | X509_V_ERR_INVALID_PURPOSE | 被验证的证书不能用于指定的使用目的 |
| 27 | X509_V_ERR_CERT_UNTRUSTED | 根 CA 证书信任设置没有指定需要验证的用途 |
| 28 | X509_V_ERR_CERT_REJECTED | 根 CA 证书信任设置选项表明不支持需要验证的通途 |
| 29 | X509_V_ERR_SUBJECT_ISSUER_MISMATCH | 当前候选证书名称和要查找的签发者证书名称不匹配，这个错误只在使用了-check_issuer选项才会显示出来 |
| 30 | X509_V_ERR_AKID_SKID_MISMATCH | 当前候选证书授权密钥和证书的主题密钥标识不匹配，这个错误只在使用了-check_issuer选项才会显示出来 |
| 31 | X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH | 当前候选证书的序列号和当前证书验证密钥标识内容不一致，这个错误只在使用了-check_issuer选项才会显示出来 |
| 32 | X509_V_ERR_KEYUSAGE_NO_CERTSIGN | 当前候选签发者证书的密钥用途中不包括签发证书功能 |
| 33 | X509_V_ERR_APPLICATION_VERIFICATION（unused） | 应用程序自定义的错误码 |

8. 简单的例子

verify 指定的选项应该是较少的，用起来也非常简单。只需要按上述的各个选项的操作进行。下面是证书 cert. pem 验证的例子，此证书用根证书 cacert. pem 生成：

```
OpenSSL>verify -CApath c:\Rootca -CAfile ca\cacert. pem -verbose cert1. pem
```

## 10. 6. 3　在线证书状态服务协议指令 ocsp

或许看到这节的内容，你禁不住要抱怨：上面的 verify 指令已经将证书验证过程说得非常明白了，为什么还要在证书验证方面喋喋不休？这是因为，在许多情况下，需要验证证书真实性的人（当然也可能计算机或者程序）并不能完成证书验证的过程，也就是说，必须提供一种手段或者服务，能够帮助完成证书验证的需要。这就是为什么 OCSP 协议（Online Certificate Status Protocol），即在线证书状态协议会出现在我们面前的原因。

OCSP 协议使得普通证书应用程序可以通过服务方式知道一个特定证书的（吊销）状态，从而使得普通程序不需要了解证书验证的细节和实现方式。

OCSP 接受一个客户端发来的证书验证请求信息（通常是以 HTTP 协议发出），然后

OCSP查询证书库的状态。当然，用户请求里面只是包含了证书的一些关键信息，而并不需要一个完整的证书。

根据结果，OCSP响应器一般返回给客户端的状态信息包括三种：

① 良好，说明请求证书是正确的，而且没有被吊销；

② 吊销，说明证书已经被吊销；

③ 未知，证书不在该OCSP响应器的范围内。

为了防止可能发生的欺骗，每一个OCSP返回的回应信息都有签名，而签名者必须是用户所信任的，也就是说，用户还必须对每一个回应信息进行验证。在OpenSSL实现上，这个签名者可能是以下三种之一。

① 被验证证书的CA。即回应信息签名是用验证证书的CA证书签发的。

② 被验证证书CA特殊授权的OCSP响应器。即使用OCSP本身的证书签发回应信息，OCSP的证书跟被验证证书是同一个CA颁发的，并且OCSP证书的用途被指明具有OCSP签名的权限。

③ 被信任的OCSP响应器。即使用OCSP本身的证书签发回应信息，OCSP证书本身的CA的根CA在信任设置里面必须是明确指明是可以用于OCSP签名的，或者是用户明确定义可以信任的OCSP响应器。

可见，OpenSSL的实现跟OCSP协议规定的是基本相符合的，虽然在第三种方式上不完全一致。OpenSSL对OCSP回应信息的验证也是通过基于OCSP证书建立一个证书链来进行的，其验证过程可以参考10.6.2节的内容。关于OCSP更多的详细内容在RFC2560中有明确定义。

### 1. 功能概述和指令格式

OpenSSL提供的指令总是超出人们的预期，这次看到的ocsp指令也不例外。它几乎可以完成大部分OCSP协议通常的功能。该指令既可以用来输出OCSP请求和应答内容，也可以创建OCSP请求并发送给一个OCSP服务器，甚至，它还可以作为一个OCSP服务器运行。下面是ocsp指令格式和其参数：

```
OpenSSL>ocsp[-out file][-issuer file][-cert file][-serial n][-signer file]
[-signkey file][-sign_certs file][-no_certs][-req_text][-resp_text][-text]
[-reqout file][-respout file][-reqin file][-respin file][-nonce][-no_nonce]
[-url URL][-host host:n][-path][-CApath dir][-CAfile file][-VAfile file][-validity_
period n][-status_age n][-noverify][-verify_certs file][-trust_other][-no_intern]
[-no_signature_verify][-no_cert_verify][-no_chain][-no_cert_checks][-port num]
[-index file][-CA file][-rsigner file][-rkey file][-rother file][-resp_no_certs]
[-nmin n][-ndays n][-resp_key_id][-nrequest n]
```

很可怕，又是一堆长长的参数！不过，到了这里，你应该对OpenSSL的特点有些了解了，看起来很多而已，常用的不过是几个，所以，不要害怕。ocsp指令里大部分选项都是用于测试和调试目的的，一般情况下，CAfile，Capath和VAfile才是要经常使用的选项。

ocsp使用的时候可以作为客户端或者服务器两种模式，下面我们对其参数的具体介绍也分为这两部分来进行。区分ocsp指令在服务器模式还是在客户端模式工作的标志就

是是否使用 index 选项：如果使用 index 选项，就是在服务器模式下工作，否则就是在客户端模式下工作。

2. 输入和输出选项 out，reqout，respout，reqin 和 respin

out 选项指定通用的输出文件，可以在服务器模式或者客户端模式使用，默认情况下使用标准输出设备。一般来说，只有可读写的信息会在 out 参数指定的文件中输出，如提示信息、文本的请求或者响应解释信息等。

reqout 选项告诉 ocsp 指令输出 OCSP 请求信息并且存储在参数指定的文件中。respout 则告诉指令输出 OCSP 响应并存储到该参数指定的文件中。上述选项输出的请求和响应都是以 DER 格式存储的。这两个选项同样都可以用于客户端和服务器模式。

reqin 和 respin 选项分别告诉指令从这两个选项指定的文件中读取 OCSP 请求或者 OCSP 响应。如果创建请求或者响应的选项出现，如 cert，serial 或者 host 选项生效，那么这两个选项将会被自动忽略。reqin 选项在服务器模式和客户端模式都可能使用，而 respin 选项则仅在客户端模式有效。

3. 输出内容选项 req_text，resp_text 和 text

req_text 选项和 resp_text 选项分别告诉指令输出证书请求或者证书响应的文本信息，如果使用 text 选项，则指令将同时输出请求和响应文本信息。这些输出会保存在 out 指定的文件中或者标准输出设备中。这些选项在服务器和客户端模式都可以使用。

4. 证书相关选项 issuer，cert 和 serial

issuer 选项指定存储要验证证书的签发者证书的文件，该证书必须是 PEM 格式的。该选项可以重复使用多次以选择多个签发者证书。该选项在创建 OCSP 请求的时候是必须的。

cert 选项指定要验证的证书，即加到 OCSP 请求中的证书。签发该证书的 CA 证书必须在 issuer 选项中指定，否则指令就不能正常执行。

serial 跟 cert 选项的功能一样，用于向 OCSP 请求中增加证书，不同的是，该选项使用证书序列号来标识要加入的证书。该序列号默认是以十进制表示，如果在参数前增加了 0x，则认为是十六进制。如果你要输入负数序列号（似乎是很奇怪的需求），只要在输入参数前面增加“－”号作为标识即可。

5. 请求相关选项 signer，signkey，nonce 和 no_nonce

如果需要对 OCSP 请求进行签名，则可以利用 signer 选项来指定签名的证书，其相应的私钥则可以在 signkey 选项指定的文件中输入。如果没有使用 signkey 选项，则 ocsp 指令将从 signer 选项指定的文件中读取私钥。如果 signer 和 signkey 都没有使用，那么 OCSP 请求就不会被加密。

为了区分每个请求，OCSP 定义了所谓的 nonce，事实上，它是一个使用摘要算法生成的跟时间相关的随机数，为了防止重放攻击等行为的发生。使用 reqin 输入已有 OCSP 请求的时候，默认情况下是不会向请求中增加 nonce 的，如果选择了 nonce 选项，则将强制向 OCSP 请求中增加 nonce 扩展项。相反，当使用 ocsp 指令创建新的请求的时候（使用 cert，serail 或者 host 选项等），默认情况下会在请求中增加 nonce 扩展项，如果不想使用 nonce 扩展项，则可以使用 no_nonce 选项去除请求中可能增加的 nonce 扩展项。

6. OCSP 响应器地址选项 host，path 和 url

OpenSSL 提供给我们的 ocsp 指令不仅仅可以处理一些 OCSP 请求或者响应信息，还可以实时模拟一个 OCSP 客户端，即时产生 OCSP 请求，发送给指定的 OCSP 响应器并得到返回的响应信息。

OCSP 响应器一般提供基于 HTTP 或者 HTTPS 的服务，其定位一般都是通过 IP 地址、端口和 HTTP 文件路径来进行。host 选项参数指定了 OCSP 响应器的地址和服务端口，一般是 host：port 的方式。而 path 选项则指定了 HTTP 文件的路径名，默认情况下使用“/”。

url 选项是另外一种指定 OCSP 响应器地址的方式，使用 HTTP 或者 HTTPS 请求的格式。事实上，在 OpenSSL 的 ocsp 指令中，url 参数也是被解释成 host 和 path 参数进行实际连接的建立的。

7. 响应验证证书选项 CAfile，CApath，verify_certs，trust_other 和VAfile

CAfile 和 CApath 选项大家已经非常熟悉，它们用来指定 CA 证书文件或者一个标准的存放 CA 证书文件的目录，这里输入的 CA 文件用于验证响应信息的签名。

很多 OCSP 响应器在返回的响应信息中不会加入完整的 OCSP 证书，那么这使得客户端在验证响应信息的时候就会有一定麻烦。为了兼容这种情况，verify_certs 选项可以指定增加一个或者多个证书，在验证 OCSP 响应时候可以查找使用这些附加的证书。

trust_other 选项告诉指令，verify_certs 指定的证书都是可信任的，不需要对这些证书进行进一步的验证工作，这时候，指令不会执行完整的证书验证过程，这对于不能够建立完整证书链的情况就非常有帮助。

VAfile 选项的功能是 verify_certs 和 trust_other 选项的综合，该选项指定的证书都是默认为可信任的附加证书，不需要进行进一步验证。

8. 响应验证策略选项 noverify，no_signature_verify，no_cert_verify，no_cert_checks，no_intern，no_chain，validity_period 和 status_age

noverify 选项告诉指令对 OCSP 响应签名和 nonce 字段不进行任何验证，该选项事实上放弃了所有对 OCSP 响应的验证。no_signature_verify 选项告诉指令不用验证 OCSP 响应中的签名信息，即便签名无效。no_cert_verify 选项告诉指令不用验证响应签名者的证书，也就是说，任何人签名的响应都是有效的。no_cert_checks 选项则告诉指令对响应签名证书的状态不用做任何附加的检查，即如果证书已经被吊销或者过期，也是不会被发现的。上述的选项因为都在不同程度降低了响应验证的强度和响应的安全性控制，所以理论上一般仅在测试的时候使用上述选项。

no_intern 选项告诉指令忽略 OCSP 响应中包含的证书信息，在这种情况下，必须使用 verify_certs 或者 VAfile 选项来增加额外的 OCSP 响应者证书以保证验证响应签名过程的顺利完成。no_chain 告诉指令不要使用响应中的证书作为附加的非信任 CA 证书，也就是说，不要让响应中的证书增加到验证过程中建立的证书链中去，这确保了验证过程中使用的证书完全是由用户来控制。

每个证书响应都会包括一个有效期，这个有效期由一个 notBefore 字段和一个可选的 notAfter 字段组成，当前的时间必须在这两个字段规定的时间之间，响应才有效。但是，这个时间间隔有时候是非常短的，可能只有几秒，而实际系统的客户端和服务器的时间可能不同步，存在误差。为了避免这种情况导致响应失效，OpenSSL 提供了 validity_period 和 status_age 选项来设定可以允许的时间误差值，其参数单位是秒。其中，validity_period 选项适用于 notAfter 扩展项存在的情况下，其设定的是一个误差秒数值，默认值是 5 分钟。如果 notAfter 扩展项不存在，这一般意味着该响应只是即时有效，这时候可以使用 status_age 选项来设定有效期，即 notBefore 时间比当前早的时间不应该超过 status_age 选项设定的值。默认情况下，这项检查是不会执行的，即响应在 notBefore 时间后总是有效。

### 9. OCSP 服务器选项 index 和 CA

上面介绍的选项大部分是用于处理 OCSP 请求、OCSP 响应或者模拟客户端，事实上，ocsp 还可以用于模拟一个 OCSP 响应器（服务器）。

index 选项是否在指令中出现是区分 ocsp 指令运行在服务器还是客户端的标记，如果使用了 index 选项，则表明指令已经运行在 OCSP 服务器（或者说响应器）的模式下。显然，OCSP 响应器就是为了处理 OCSP 请求的，在 OpenSSL 的 ocsp 指令中，OCSP 请求的输入方式是灵活多样的：第一种是指令行方式，即直接在 ocsp 指令行中输入，使用 issuer 和 serial 参数产生请求信息；第二种是采用文件输入方式，即使用 reqin 选项指定一个要处理的 OCSP 请求；第三种方式是采用其他 OCSP 客户端输入，即定义好 OCSP 运行的服务端口 port 或者服务 URL 即可。

index 选项跟我们在 ca 指令中看到的 index 选项相同，指定的是 OpenSSL 提供的模拟 CA 中的证书库文件，该文件包含了该 CA 所有颁发的证书信息和其（吊销）状态。

在服务器运行模式下，CA 选项是必须使用的，它指定了 index 证书库相应的 CA 证书文件。

### 10. OCSP 响应签发者选项 rsigner，rkey 和 resp_key_id

OCSP 响应必须是经过签名的，所以在服务器模式下，rsigner 选项是务必要使用的，它指定了用于对 OCSP 响应签名的证书。rkey 则指定了 rsigner 证书选项相对应的私钥，如果 rkey 选项在指令中没有出现，那么在服务器模式下，指令会从 rsigner 指定的文件中读取用于响应签名的私钥。

OCSP 响应被签名后，还需要在响应中标识用于签名的私钥，默认情况下使用的是证书主题名。如果使用 resp_key_id 选项，则将使用私钥 ID 标识 OCSP 响应信息签名私钥。

### 11. OCSP 响应内容选项 rother，resp_no_certs，nmin 和 ndays

如果想在返回给客户端的响应信息中包含其他附加证书信息（可能用于客户端验证响应签名时使用），那么可以使用 rother 选项指定存储这些附加证书的文件。相反，如果不想在响应信息中放入任何证书（包括签发响应的证书），那么可以使用 resp_no_certs 选项。

nmin 和 ndays 选项用来填充响应信息中的 nextUpdate 字段，该字段告诉客户端下一次吊销列表更新的时间将会在多长时间之后，nmin 和 ndays 分别以分钟和日为单位进行定义。如果在 ocsp 指令中这两个选项都没有使用，那么 nextUpdate 字段就会从响应信息

中删除，这意味着吊销列表随时都会进行更新。

12. OCSP服务器运行方式选项port和nrequest

这两个参数仅在ocsp指令以服务器方式运行且请求是从OCSP客户端输入的情况下使用。其中，port选项定义了OCSP服务器的服务端口，而nrequest选项则告诉指令在接受和处理该参数指定数量的OCSP请求后退出运行状态。

事实上，还有其他一些参数没有一一介绍，比如engine选项，其意义跟其他指令中的同名选项是一样的，即使用第三方密码设备或者密码库来执行一些运算操作。读者可以根据指令的使用和源代码的阅读来进一步了解这些指令的更多参数。

13. 简单的例子

下面从OpenSSL帮助文档中选出一些例子，进一步帮助读者理解OCSP协议和OpenSSL提供的ocsp指令的运行方式方法。

创建一个OCSP请求，并把它写入指定文件中去：

```
OpenSSL>ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem -reqout req.der
```

发送一个查询到OCSP响应器，此响应器的服务URL为http://ocsp.myhost.com/，保存返回的响应到一个文件中，并把它以文本格式打印出来：

```
OpenSSL> ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem -url http://ocsp.myhost.com/ -resp_text -respout resp.der
```

读一个OCSP响应并以文件格式打印出来：

```
OpenSSL>ocsp -respin resp.der-text
```

OCSP服务器使用8888端口服务，按照标准CA配置并使用独立的响应者证书，所有的响应和请求以文本方式写入指定的输出文件：

```
OpenSSL>ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem -text -out log.txt
```

OCSP Server使用8888端口服务，按照标准CA配置并使用独立的响应者证书，但在处理完一个OCSP请求后立即退出服务：

```
OpenSSL>ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem -nrequest 1
```

从指令行即时产生一个OCSP请求，并实时到OCSP服务器查询其状态信息：

```
OpenSSL>ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem -issuer demoCA/cacert.pem -serial 1
```

从文件读入一个已经存在的OCSP请求，发送到OCSP服务器查询状态，并把返回的响应写入另一个文件中：

```
OpenSSL>ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem -reqin req.der -respout resp.der
```

## 10.7 本章小结

本章对证书和CA相关的知识作了简单的介绍，并结合OpenSSL的指令，对证书生命周期中涉及的问题和操作进行了详细的介绍。

在本章开头首先概括性地介绍了证书和 CA 的功能与基本概念，使读者在基本概念上有一个总体的把握。

接着，本章遵循整个证书生命周期的流程，首先对证书申请过程中涉及的密钥产生和证书请求相关的各种问题作了介绍，并具体介绍了 OpenSSL 提供的指令操作方法。通过这一部分阅读，读者对证书中的密钥，如何申请不同用途的证书，以及使用 OpenSSL 指令完成这些任务都应该有了比较好的掌握。

本章还介绍了建立一个 CA 和运行一个 CA 需要考虑的各种技术问题和需求，并结合 OpenSSL 提供的指令示范了如何根据需求建立 CA，并签发各种用途的用户证书。

然后，本章还介绍了如何使用 OpenSSL 指令去操作几种常用证书，比如说对 X.509，PKCS#12，PKCS#7 证书具体生成和使用的介绍，让读者对这几种常见到的证书有感性而全面的认识。

最后本章对证书的验证过程和应用方式进行了介绍，并结合 OpenSSL 中的指令，对不同应用方式的证书验证过程作了详细的介绍，让读者掌握 OpenSSL 证书验证指令的同时，深入了解证书的应用模式。

# 第 11 章 OpenSSL 的标准转换指令

## 11.1 标准转换指令概述

阅读完本书前面的章节，你对 OpenSSL 最痛苦的感受之一可能就是其繁多复杂的各种文件编码格式、证书格式和密钥格式等。事实上，并非 OpenSSL 开发者想要将数字世界弄得如此令人头疼，只是由于各种原因，数字世界存在各种不同的标准，为了尽量兼容这些不同的标准，OpenSSL 开发者才相应地在其指令和代码中支持如此多的格式。这是伟大的行为，但同时也是令 OpenSSL 初学者痛苦的根源之一。

要弄清楚这些不同标准之间的区别已然不容易，更为可怕的是，很多时候你还面临处理这些不同标准之间相互兼容和转换的问题，比如将 X.509 证书转换成 PKCS#12 格式证书就是常见的任务。事实上，在 OpenSSL 的很多指令中提供了不同格式和标准之间的转换功能，读者对这些标准转换的操作也有了一定的了解，本章将针对 OpenSSL 支持的其他一些标准进行介绍，阐述这些标准和其他标准之间的关系，并结合 OpenSSL 提供的工具指令介绍这些标准之间的转换方法。

因为前面的章节已经对大部分的标准和转换指令做了介绍，本章将集中介绍 pkcs8 和 nseq 指令的使用及其相关的标准。

## 11.2 PKCS#8 标准和指令

### 11.2.1 PKCS#8 标准简介

相对于其他标准，PKCS#8 标准算是一个非常简单的标准了，它主要用于封装私钥和其他相关的属性信息。一般来说，PKCS#8 格式的私钥都是被加密的，支持 PKCS#5 和 PKCS#12 标准定义的算法，当然，私钥也可以不加密。PKCS#8 标准一方面可以增强私钥的安全性，另一方面也为用户提供了一种简单的确立信任关系的方式，这主要是基于私钥特别名称和最高层可信者的权威公钥等属性信息。

### 11.2.2 pkcs8 指令介绍

#### 1. 功能概述和指令格式

OpenSSL 提供了 pkcs8 指令来执行 PKCS#8 标准相关的工作。它可以把私钥转化为

PKCS＃8 的格式，也可以把 PKCS＃8 格式的私钥转换成其他存储标准的格式。OpenSSL 的 pkcs8 指令对加密和非加密格式的私钥都能进行处理，其支持的加密算法有 PKCS＃5（版本 1.5 和 2.0）和 PKCS＃12 两种标准的定义。

OpenSSL 提供的经过 PEM 编码的 PKCS＃8 标准的文件，分为加密和非加密两种方式。加密的 PKCS＃8 密钥标识如下：

—BEGIN ENCRYPTED PRIVATE KEY—

—END ENCRYPTED PRIVATE KEY—

非加密的 PKCS＃8 密钥标识如下：

—BEGIN PRIVATE KEY—

—END PRIVATE KEY—

这些标识符中间就是 PEM 编码的 PKCS＃8 文件内容。

pkcs8 指令格式如下：

```
OpenSSL>pkcs8[-topk8][-inform PEM|DER][-outform PEM|DER][-in filename][-passin arg][-out filename][-passout arg][-noiter][-nocrypt][-nooct][-embed][-nsdb][-v2 alg][-v1 alg][-engine id]
```

显然，选项很少，而且大部分我们都已经非常熟悉，这是很值得庆幸的事情，不是吗？

2. PKCS＃8 文件生成选项 topk8

默认情况下，pkcs8 指令会要求输入一个 PKCS＃8 格式的密钥，并将其进行处理后转换成普通格式的密钥输出并存储，这种情况一般用于处理或者显示一个已经存在的 PKCS＃8 密钥的内容。如果使用了 topk8 选项，则 pkcs8 指令会将输入的一个普通格式的密钥转换成 PKCS＃8 标准封装的密钥并输出。

3. 输入输出选项 in，out，inform 和 outform

in 选项指定了输入密钥的文件名，默认情况下使用标准输入。如果输入的密钥是经过加密的，那么在读取密钥信息的过程中会提示输入加密口令。当然，如果你已经使用 passin选项提供了加密口令，该提示信息就不会再出现。

out 选项指定了存储输出密钥信息的文件名，默认情况下是标准输出。如果选择了加密选项而又没有使用 passout 选项指定加密口令时，在输出密钥时会提示输入加密口令。输出文件与输入文件在同一个文件夹下时不可以重名，否则生成的文件会在没有读原密钥文件之前覆盖原文件，出现读密钥错误。

inform 和 outform 选项分别指定了输入和输出文件的编码格式，支持的格式包括 PEM 和 DER 两种。

4. 加密口令输入选项 passin 和 passout

由于私钥的重要性，所以一般都使用了加密保护，加密保护使用的密钥和初始向量是从用户提供的口令中提取出来的，所以在使用密钥的时候也需要提供相同的口令进行密钥的解密。passin 和 passout 选项分别指定了输入和输出密钥时进行加解密的口令源。关于口令源的具体使用方法和形式请参考本书 6.6 节。

如果输入的密钥文件需要口令进行解密而却没有使用 passin 选项提供口令，那么程

序会在指令行提示用户输入解密口令。

5. 私钥格式选项 nocrypt，nooct，embed 和 nsdb

使用 pkcs8 指令时，通常情况下输入或是产生的 PKCS＃8 格式的密钥都是使用口令基于某种算法加密的私钥信息结构。如果使用 nocrypt 选项，则告诉指令其输入和产生的私钥都将是不经加密的私钥信息结构。私钥不进行加密，显然安全性会大大降低，所以除非万不得已，一般不推荐使用此选项。

nooct 选项用于产生一种非标准格式的 RSA 私钥，以兼容一些软件。RSA 私钥一般明确要求被封装到一个 OCTET STRING 字符串内。但是有些软件只能识别私钥自身，对附加的 OCTET STRING 字符串不能识别。在这种情况下，就可以使用 nooct 选项去掉 OCTET STRING 字符串。

embed 选项用于产生一种非标准格式的 DSA 密钥。在这种格式中，DSA 参数内嵌到 PrivateKey 结构中。

nsdb 选项则用于产生一种兼容 Netscape 私钥数据库格式的 DSA 密钥。

6. 密钥加密算法选项 v2 和 v1

v2 选项用于选择 PKCS＃5 v2.0 中的所有算法。一般情况下，PKCS＃8 结构的私钥是使用 pbeWithMD5AndDES-CBC 这种口令加密算法加密的。虽然这种算法只使用了 56bit 的 DES 加密，但已经是 PKCS＃5 v1.5 中加密强度最高的算法。使用 v2 选项则可以选择 PKCS＃5 v2.0 中更高强度的算法，这些算法有 168 bit 的 3DES 或者 128 bit 的 RC2 等。目前的软件支持 PKCS＃5 v2.0 的不多，但是如果你使用的范围都局限于 OpenSSL，那么就一点问题都没有。

v2 选项有效的参数包括 des，des3 和 rc2，OpenSSL 推荐读者使用 DES3 这种加密算法。

v1 选项用于选择 PKCS＃5 v1.5 或者 PKCS＃12 中定义的算法，事实上，也包含了部分 PKCS＃5 v2.0 中定义的算法。表 11-1 列出了 v1 选项可以使用的参数及其含义。

表 **11-1** **pkcs8** 指令 **v1** 参数查询表

| 参　数 | 算法描述 | 相关标准 |
|---|---|---|
| PBE-MD2-DES | 56 位 DES 算法，口令迭代算法是 MD2 | PKCS＃5 v1.5 |
| PBE-MD5-DES | 56 位 DES 算法，口令迭代算法是 MD5 | PKCS＃5 v1.5 |
| PBE-SHA1-RC2-64 | 64 位 RC2 算法，口令迭代算法是 SHA1 | PKCS＃5 v2.0 |
| PBE-MD2-RC2-64 | 64 位 RC2 算法，口令迭代算法是 MD2 | PKCS＃5 v2.0 |
| PBE-MD5-RC2-64 | 64 位 RC2 算法，口令迭代算法是 MD5 | PKCS＃5 v2.0 |
| PBE-SHA1-DES | 56 位 DES 算法，口令迭代算法是 SHA1 | PKCS＃5 v2.0 |
| PBE-SHA1-RC4-128 | 128 位 RC4 算法，口令迭代算法是 SHA1 | PKCS＃12 |
| PBE-SHA1-RC4-40 | 40 位 RC4 算法，口令迭代算法是 SHA1 | PKCS＃12 |
| PBE-SHA1-3DES | 168 位 3DES 算法，口令迭代算法是 SHA1 | PKCS＃12 |
| PBE-SHA1-2DES | 112 位 2DES 算法，口令迭代算法是 SHA1 | PKCS＃12 |
| PBE-SHA1-RC2-128 | 128 位 RC2 算法，口令迭代算法是 SHA1 | PKCS＃12 |
| PBE-SHA1-RC2-40 | 40 位 RC2 算法，口令迭代算法是 SHA1 | PKCS＃12 |

7. engine 选项

如果 engine 选项指定了有效的 Engine 设备，那么指令中任何该 Engine 设备支持的上述操作都会使用 Engine 设备的操作流程而不再使用 OpenSSL 算法库本身提供的函数。对于 pkcs8 指令来说，可以使用 engine 设备的操作主要包括 RSA 或者 DSA 密钥产生和对私钥进行加解密的算法。

8. 简单的例子

读完上面的部分，你肯定已经确信 pkcs8 是一个简单的指令了，虽然如此，下面还是要再举一些常用的例子。

首先我们直接把一个普通的私钥转化为 PKCS＃8 格式的密钥（默认使用 DES 算法加密）：

```
OpenSSL>pkcs8 -in key.pem -topk8 -out enckey.pem
```

非常简单就把一个普通的私钥转化为 PKCS＃8 格式的密钥，不是吗？下面使用 PKCS＃5 v2.0 中的 DES3 加密算法转换密钥格式：

```
OpenSSL>pkcs8 -in key.pem -topk8 -v2 des3 -out enckey.pem
```

使用 PKCS＃12 提供的算法进行密钥格式转换（使用 DES3 加密算法）：

```
OpenSSL>pkcs8 -in key.pem -topk8 -out enckey.pem -v1 PBE-SHA1-3DES
```

把一个 DER 格式的未经加密的 PKCS＃8 结构的私钥转换成普通格式：

```
OpenSSL>pkcs8 -inform DER -nocrypt -in key.der -out key.pem
```

## 11.3　Netscape 证书标准

### 11.3.1　Netscape 证书标准简介

为了方便下载一系列的证书，尤其对于为了验证要下载一个证书链的情况，Netscape 提供了一种名为 Netscape 证书序列（Netscape Certificate Sequence）的格式来封装一系列证书（事实上，里面采用了一个 PKCS＃7 格式来封装证书），以便能够一次性地下载或者传输多个数字证书。所以，某些时候，Netscape 证书序列可以替代 PKCS＃7 的作用，用来打包一系列证书。

Netscape 证书序列虽然不一定能够得到微软的支持，但是在其他一些开源软件和 Linux 软件中却得到了广泛的支持，我们也因此随时会面临要把普通的一些证书封装成 Netscape 证书序列的要求和任务。OpenSSL 显然也考虑到了这些可能出现的问题，于是提供了本节要介绍的 nseq 指令。

### 11.3.2　nseq 指令介绍

1. 功能概述和指令格式

nseq 指令的任务简单而明确：将普通 X.509 证书封装成 Netscape 证书序列，或者将 Netscape 证书序列转换成普通的 X.509 证书。这个指令的格式简单，也容易掌握，其格式为：

```
OpenSSL>nseq[-in filename][-out filename][-toseq]
```

2. 输入和输出文件选项 in 和 out

in 和 out 选项分别指定了输入和输出的文件名，默认是标准的输入和输出设备。

3. 转化格式选项 toseq

使用 toseq 选项告诉指令执行将一个或者多个普通 X.509 证书转换成一个 Netscape 证书序列的操作。如果不使用该选项，即默认情况下，输入的应该是一个 Netscape 证书序列文件，而输出的是一个或者多个普通 X.509 证书，当然，它们都存储在一个输出文件中。

4. 简单的例子

下面给出几个简单的例子。首先介绍如何从一个 Netscape 证书序列得到普通的证书：

```
OpenSSL>nseq -in nseq.pem -out certs.pem
```

从普通证书创建一个 Netscape 证书序列：

```
OpenSSL>nseq -in certs.pem -toseq -out nseq.pem
```

## 11.4 本章小结

本章在前面章节的基础上，对 OpenSSL 涉及的标准和格式转换背景、指令及其使用情况做了总结和补充。事实上，本章对于一般的读者来说，可能并不需要涉及。

本章首先对 PKCS#8 标准进行了简单的介绍，并结合 pkcs8 指令，让读者对 PKCS#8 及其在 OpenSSL 中相对应的指令有了一个基本的了解，加深读者对密钥管理重要性和方法的认识。

最后，结合 nseq 指令，本章还对 Netscape 的证书序列进行了介绍。

# 第 12 章

# OpenSSL 的 SSL 相关指令

## 12.1 再谈 SSL 与 OpenSSL

如果你认为 OpenSSL 就是为了实现 SSL 协议而存在的，那么很可能对我们已经颇有怨言，因为直到本章，才进入 OpenSSL 中 SSL 协议的相关内容。不过正如本书前面所说，OpenSSL 之博大，并非仅仅局限于 SSL 协议本身。事实上，由于 SSL 协议已经是密码学和 PKI 技术中非常具体的一个应用协议，为了实现它，OpenSSL 在密码学基础应用和 PKI 技术的基础实现上做了大量的工作，才逐渐形成和奠定了 OpenSSL 在密码学应用和 PKI 技术开发中的重要基础软件包地位。

当然，OpenSSL 的初衷，就是实现 SSL 协议，而且是以开放源代码的方式发展，所以 OpenSSL 对 SSL 协议的支持，是全面而且具体的，包括了协议的实现、协议的应用和协议的开发调试等多个方面。本章将主要从应用层面上来对 SSL 协议进行阐述，这仅是 OpenSSL 提供的 SSL 支持的一部分内容，OpenSSL 还提供了基于 SSL 开发 API 等更多具有重要价值的资料，但并不在本书的阐述范围之内。

前面的章节已经对 SSL 协议进行了充分的介绍，本章将主要结合 OpenSSL 提供的几个 SSL 相关指令，对 SSL 协议的运行环境、实施和测试进一步进行阐述，使你脱离 RFC 那些看起来似乎抽象绵长的文档，实实在在感受一下 SSL 协议的实质。

OpenSSL 提供了四个直接的 SSL 相关指令：s_client，s_server，s_time 和 sess_id，这些指令可以模拟 SSL 客户端和服务器端的多种动作和环境，用于测试和分析现有的 SSL 服务器或者客户端的运行状况，也可以将这些源代码作为实现 SSL 客户端或者服务器的参考。

## 12.2 SSL 服务器分析

SSL 服务器已经无处不在，如 IIS 的 HTTPS 协议，就是 SSL 服务器的实现之一。SSL 服务器到底在干什么？SSL 服务器对客户端的请求是怎么处理的？很显然，如果你刚开始接触 SSL 协议，对于分析这些 SSL 服务器以了解其中运行的过程会非常感兴趣。最有效和直接的方法，就是模拟一个客户端的动作，去连接 SSL 服务器，这样对 SSL 服务器的响应行为和运行就将一清二楚。OpenSSL 显然考虑了我们这些共同的需要，提供 s_client 和 s_time 指令模拟一个 SSL 客户端，让你非常轻松就可以对 SSL 服务器进行

测试。

### 12.2.1 用 s_client 指令模拟 SSL 客户端

#### 1. 功能概述和指令格式

s_client 指令模拟了一个通用的 SSL/TLS 客户端。事实上，为了调试的目的，它比一般的 SSL 客户端实现复杂得多。你可以用 s_client 连接各种 SSL 服务器，如果你要连接一个 HTTPS 服务器（HTTS 服务端口是 443），输入下面的口令就可以：

```
OpenSSL>s_client -connect 192.168.0.1：443
```

一旦 s_client 和某个 SSL 服务器的连接建立成功，那么服务器所有传送过来的信息都会被显示出来，而客户端的所有键盘输入也会被发送给服务器。

下面通过对 s_client 指令各个参数的介绍来全面了解该指令的功能。首先看看s_client 指令的格式：

```
OpenSSL> s_client[-connect host:port >][-host host][-port n][-verify
depth][-cert filename][-key filename][-capath directory][-CAfile filename]
[-crl_check_all][-crl_check][-reconnect][-pause][-showcerts][-debug][-nbio_
test][-state][-nbio][-crlf][-ign_eof][-quiet]
[-ssl2][-ssl3][-tls1][-no_ssl2][-no_ssl3][-no_tls1][-bugs][-cipher cipher-
list][-serverpref][-rand file][-starttls prot]
```

#### 2. 服务地址选项 connect，host 和 port

作为一个 SSL 模拟客户端，要运行的时候，首先当然需要指定 SSL 服务器的地址和服务端口。connect 选项指定了连接的主机地址和服务端口，其参数形式如下：

```
-connect host:port
```

其中，host 可以为 IP 地址，也可以为 URL；port 为服务端口。

事实上，你还可以使用两个很少使用的 host 和 port 选项来分别指定服务地址和服务端口，如下面的指令：

```
OpenSSL>s_client -host 192.168.0.1 -port 443
```

如果上述选项都没有使用，那么默认建立的连接为本地主机的 4433 端口。

#### 3. 客户端证书选项 cert 和 key

cert 选项指定客户端要使用的证书文件，该文件包含的证书仅在 SSL 服务器明确提出要求验证客户端证书的时候会被使用，否则该证书没有任何用处，cert 选项也完全可以省略。

key 选项则指定了客户端证书相应的私钥文件，如果使用了 cert 选项而没有使用该选项，那么指令会试图从 cert 指定的证书存储文件中读取私钥。

当服务器要求进行客户端证书验证的时候，最常见到的错误就是声称客户端没有用户证书或者说提供的证书列表文件是空的（当然，你很有信心，你提供的证书文件绝对是包含一个证书的）。这种情况出现的原因，一般是因为服务器在要求客户端证书验证的时候发送过来的信任 CA 证书列表里面不包含签发你所提供的客户证书的 CA 证书。使用 s_client指令，你可以查看到 SSL 服务器支持的 CA 列表。不过，有些 SSL 服务器只会在

客户端访问某些页面的时候才要求进行客户端证书验证，这时候，为了看到其支持的 CA 证书列表，可以使用 prexit 选项将 SSL 连接过程中的所有信息打印处理。

4. 验证服务器选项 verify，CApath，CAfile，crl_check 和 crl_check_all

SSL 支持双向的证书验证，也就是说，不仅仅服务器可以验证客户端的身份，客户端也可以验证服务器的身份。事实上，在目前的运用中，更常见的是客户端对服务器身份的验证。使用 verify 选项后，将启动 SSL 模拟客户端对 SSL 服务器证书的验证过程，同时，verify 的参数设定了证书验证的深度，超过此深度，即视为证书验证失败。

所谓证书验证深度，就是证书链（这里指服务器证书的证书链）的长度，即从根 CA 到服务器证书的层数，如果签发服务器证书的 CA 本身就是根 CA，那么证书链长度就为 1。

需要注意的是，由于 s_client 仅仅是一个 SSL 客户端模拟程序，为了能充分显示出 SSL 连接过程中所有可能出现的问题，当验证服务器证书的过程中碰到错误或者应当视为验证失败的情况下，s_client 程序也不会中断连接，而是会继续工作下去，当然，它会将所有错误信息忠实地输出来。所以，s_client 程序永远不会因为验证服务器证书失败而终止运行。

CApath 选项指定了存放用于验证服务器证书时使用的候选信任 CA 证书标准目录。在需要建立客户端的证书链时，该目录存放的证书也可能被使用。

CAfile 选项则指定了一个存放一系列信任 CA 证书的文件，这些证书在验证服务器证书的时候需要使用。此外，当需要建立客户端的证书链时，这个文件中存放的 CA 证书也可能被使用。

crl_check 选项告诉指令要同时检查服务证书在 CRL 列表中的状态，crl_check_all 则告诉指令要检查整个服务器证书链中每一个证书在 CRL 列表中的状态。CRL 由 cert 指令指定的文件输入。

5. 显示信息和状态选项 showcerts，prexit，state，msg，debug 和 quiet

默认情况下，s_client 指令仅输出服务器的证书，如果使用了 showcerts 选项，则将显示服务器的整个证书链。在你验证服务器证书碰到问题时，可以使用该选项来诊断是否是证书原因。

使用 prexit 选项后，无论 SSL 连接是否建立成功，当程序退出时都会试图输出所有会话（Session）信息。而一般情况下，仅在连接建立成功的时候，s_client 选项才会输出会话信息。在以下几种 SSL 连接曾经成功建立的情况下，prexit 选项对诊断 SSL 连接状态非常有用：

① 使用的加密算法需要重新协商；

② 因为需要对客户端证书进行验证导致 SSL 连接建立失败；

③ 在访问特定 URL 的时候要求进行客户端证书验证导致连接建立失败。

当然，prexit 选项输出的信息有时候不一定非常准确，因为有时候 SSL 连接可能从来就没有成功建立过，这时候输出的会话信息自然就没有任何意义。

state 选项告诉指令输出 SSL 连接会话（Session）状态。

msg 选项告诉指令以十六进制编码的格式输出所有协议相关的信息。

debug 选项告诉指令输出所有的调试信息，并将所有的通信数据以十六进制编码的格式输出。

quiet 选项告诉指令不要输出所有会话和证书信息，同时，该选项会默认打开 ign_eof 这个选项。

6. socket 连接方式选项 nbio_test 和 nbio

nbio_test 选项告诉指令对非阻塞 SSL 通信 socket 进行更多的测试检查。

nbio 选项告诉指令使用非阻塞 socket 进行 SSL 通信。

7. 连接动作选项 pause，reconnect，starttls 和 ign_eof

使用 pause 选项后，指令在每次读或者写操作之后都会暂停一秒钟。

reconnect 选项主要用于测试服务器会话缓存功能是否有效，使用该指令后，s_client 会使用同一个会话（Session）反复连接相同的 SSL 服务器 5 次。

有些应用协议支持 TLS 协议，但是必须在建立协议连接之后输入一个 STARTTLS 指令才能开始建立 TLS 连接的动作，s_client 指令为了支持这种协议形式，增加了starttls 选项。starttls 选项的参数是要建立连接的协议代码，目前仅支持“smth”协议，相信以后会慢慢完善和增加的（最新的版本已经增加了 pop3 协议）。

ign_eof 选项告诉指令当输入文件到达文件尾（遇到文件结束符）时保持连接。如果不启用 ign_eof 选项（或者 quiet 选项），s_client 指令跟服务器的连接建立后就处于交换模式，这时候还可以输入几个简单的指令来调整连接的状态，目前支持的指令包括：

① R，会话重新协商，即在行开始第一个字母输入“R”的时候，SSL 当前的会话进行重新协商，采用新的会话；

② Q，断开连接，即在行开始第一个字母输入“Q”的时候，SSL 当前连接立即断开，结束 s_client 指令。事实上，当输入到达文件尾（文件结束符）的时候，一样会断开连接。

8. 协议版本选项 ssl2，ssl3，tls1，no_ssl2，no_ssl3 和 no_tls1

这些选项告诉指令选择用什么版本的协议建立 SSL 连接。默认的情况下，SSL 客户端支持所有版本协议，包括 SSL 2.0，SSL 3.0 和 TLS 1.0，并自动选择协商合适的协议建立连接。但是有些 SSL 服务器的实现并不规范，所以不一定能支持所有协议，并且会要求一定要不支持某种协议（如 TLS 1.0）才能够建立连接。

使用 ssl2，ssl3 或者 tls1 选项是告诉指令只使用指定的版本协议，放弃其他协议。如果这几个选项同时使用，那么只有最后一个选项才会有效。

no_ssl2，no_ssl3 和 no_tls1 选项则告诉指令放弃某种指定的协议。

9. 密码算法选项 cipher 和 serverpref

SSL 客户端和服务器建立连接的时候，需要协商一组密码算法，用户通信加密和数据完整性保证等，通常情况下，服务器会从客户端发送过来的所有支持的密码算法组中选择第一组（服务器当然也必须支持该组算法）作为双方通信使用的算法。有时候可能你希望使用某组特定的算法，这时候就可以使用 cipher 选项来指定。关于 cipher 参数的具体信息，请参考本书相关章节的内容。

serverpref 选项仅对 SSL 2.0 协议有效，它告诉指令使用 SSL 服务器指定的算法来建

立连接。

10. 其他选项 bugs，crlf，rand 和 engine

SSL/TLS 有几个众所周知的实现上的 BUG，s_client 指令提供 bugs 选项来适应存在这些 BUG 的服务器。

crlf 选项把终端输入的换行回车符转化成/r/n 发送给 SSL 服务器，某些服务器有这样的特殊需求。

rand 选项指定了用于产生随机数种子的文件，这跟其他指令的同名选项是一样的，这里不再多作介绍。

engine 选项则告诉指令使用指定的 Engine 设备，参数是 Engine 设备的 ID 号。

11. 简单的例子

下面的例子将与招商银行的信用卡网站建立 SSL 连接，并显示出其所使用的证书链：

```
OpenSSL>s_client -connect creditcard.cmbchina.com:443 -showcerts
```

你可以看到，招商银行用的是 VeriSign 签发的数字证书，并且该证书号称是广东深圳的。

### 12.2.2　SSL 服务器性能测试指令 s_time

1. 功能概述和指令格式

OpenSSL 提供了另外一个与 s_client 相似的 SSL 客户端模拟程序 s_time，不同的是，该程序主要用来测试 SSL 服务器的性能。s_time 程序通过从服务器下载一个页面，通过测试其传输的数据和花费的时间来测试 SSL 服务器性能，其测试的项目包括：

① 给定时间段内建立的连接数量；

② 传输的数据量；

③ 计算每个连接平均花费时间。

下面是 s_time 指令的选项和格式：

```
OpenSSL>s_time[-connect host:port][-www page][-cert filename][-key filename]
[-CApath directory][-CAfile filename][-reuse][-new][-verify depth][-nbio][-time
seconds] [-ssl2][-ssl3][-bugs][-cipher cipherlist]
```

你或许已经发现，这个指令的选项绝大部分都跟 s_client 指令相同，那么我们也不再花费时间，只对其中在 s_client 指令中没有出现的选项进行详细的介绍，其他选项请你翻阅 12.2.1 节内容。

2. 网页内容选项 www

www 选项的参数指定了要从 SSL 服务器上获取（GET 动作）的网页名称，例如“default.asp”这样的网页文件名。如果该选项没有定义，那么 s_time 指令只是完成握手建立连接，但是不传输任何应用数据。

3. 连接会话选项 new 和 reuse

new 选项告诉 s_time 指令每次建立新 SSL 连接的时候都使用新的会话（Session）ID。

reuse 选项则告诉 s_time 指令每次建立新 SSL 连接的时候使用相同的会话（Session）

ID。这可以用来测试 SSL 服务器的会话缓存功能的工作状态是否正常。

如果上述两个选项都没有使用，那么指令会默认将两个选项都打开，并交替执行这两个选项定义的内容。也就是说在建立新 SSL 连接的时候，指令一会儿用新的会话 ID，一会儿重用上次会话 ID。

4. 连接时间选项 time

选项 time 告诉指令统计在该选项参数指定的时间内（单位是秒）s_time 和 SSL 服务器能够建立连接并从服务器传输指定数据（如果 www 指定了页面）的次数。当然，最后测试结果出来的时间不一定正好是指定的时间，因为指令必须在等待一个完整的连接动作完成后才能够结束并给出结果。

5. 简单的例子

我们只举一个简单的例子来说明 s_time 指令的应用，该例子从招商银行的信用卡网站请求一个名为 default.asp 的网页，并要求测试 10 秒钟内连接（包括数据传输）的次数，指定使用 RC4-MD5 算法簇：

```
OpenSSL>s_time -connect creditcard.cmbchina.com:443 -www default.asp -time 10
-cipher RC4-MD5
```

## 12.3 SSL 客户端分析

我们可能需要对现有的 SSL 客户端或者我们自己正在开发的 SSL 客户端进行测试。为此，OpenSSL 同样很负责任地提供了 s_server 指令，用于模拟一个通用的而且负责的 SSL 服务器，目的就是为了能够对 SSL 客户端进行测试。

1. 功能概述和指令格式

s_server 指令不仅仅模拟了一个完整通用的 SSL 服务器，并且还在此基础上模拟了一个 Web 服务器，提供 Web 服务。下面是 s_server 指令的选项和格式：

```
OpenSSL s_server[-accept port][-context id][-verify depth][-Verify depth]
[-cert filename][-key keyfile][-dcert filename][-dkey keyfile][-dhparam filename]
[-nbio][-nbio_test][-crlf][-debug][-msg][-state][-CApath directory][-CAfile
filename][-nocert][-cipher cipherlist][-quiet][-no_tmp_rsa][-ssl2][-ssl3][-tls1]
[-no_ssl2][-no_ssl3][-no_tls1][-no_dhe][-bugs][-hack][-www][-WWW][HTTP][-engine
id][-id_prefix arg][-rand file(s)]
```

这些选项有 36 个之多，其中有相当一部分选项的名称和含义与 s_client 中的同名选项相同，因此，在下面的选项介绍中，我们将选择介绍在 s_client 中没出现过或者含义不尽相同的选项进行介绍，其他没有介绍的选项，读者可以在 12.2.1 节中参考同名的选项说明。

2. 监听端口选项 accept

accept 选项指定 SSL 服务端口，在默认的情况下指令将使用 4433 端口。

3. 证书和私钥选项 cert，key，dcert，dkey 和 nocert

cert 选项指定服务器使用的证书。大部分服务器算法簇的运行需要数字证书，而且有

些还需要特定类型公钥的证书。例如在 DSS 算法簇中就需要一个包含了 DSS（DSA）密钥的证书。cert 选项如果没有定义，默认情况下将使用 server. pem 文件中的证书。

key 选项指定存储私钥的文件，该私钥和 cert 选项指定的证书应该是相对应的，如果本选项没有出现，那么 cert 选项定义的证书文件会被认为同时也是存储私钥的文件。

正如前面所说，在 SSL 协议的算法簇中，不同的算法可能需要不同密钥类型的证书来进行服务器验证。例如，RSA 算法簇需要 RSA 密钥证书来完成 SSL 连接的建立，而 DSA 算法簇则需要 DSA 密钥证书来完成 SSL 连接的建立过程。所以，为了使得 SSL 服务器能够兼容各种不同客户端的需要，可能需要在服务器端提供不同类型的证书以备选用，这就是下面 dcert 和 dkey 选项出现的原因。

dcert 和 dkey 这两个选项用于在 cert 跟 key 选项之外再指定一个附加的证书文件和私钥文件，它们的用法跟 cert 和 key 选项是相同的，不过，dcert 和 dkey 选项没有默认值。因为我们的服务器需要支持不同的组合密码，而不同的组合密码需要不同的证书和私钥文件。不同的组合密码在这里主要是指组合密码里面的不对称加密算法不同。比如基于 RSA 的组合密码需要的是 RSA 的私钥文件和证书；而基于 DSA 算法的，则需要的是 DSA 的私钥文件和证书，使用这两个选项，就使得我们的服务器同时支持两种不同的组合密码成为可能。

nocert 选项告诉服务器不使用证书进行 SSL 连接的建立，这样的直接后果就是造成需要使用证书的 SSL 算法簇不能使用，即 SSL 服务器将仅支持匿名的 DH 算法进行 SSL 连接建立。不过，目前有些浏览器（如 IE 和 Netscape）仅支持基于 RSA 算法证书的 SSL 算法簇，如果 SSL 服务器不支持 RSA 算法，则将导致 SSL 连接建立失败。

4. DH 参数选项 dhparam 和 no_dhe

选项 dhparam 指定一个 DH 参数文件。DH 密码簇在使用的时候需要一组 DH 参数来产生私钥。如果没有指定 DH 参数文件，当使用 DH 算法的时候，那么服务器首先会尝试从服务器证书文件里面获得 DH 参数；如果证书文件里面没有 DH 参数，那么 s_server 指令程序代码中的一组固定 DH 参数将会被使用。

no_dhe 选项告诉 s_server 指令禁止一切的 DH 参数的调用，此时，DH 算法簇将不会在 SSL 连接中被使用。

5. 证书验证选项 verify，Verify，CApath 和 CAfile

选项 verify 要求客户端发送证书进行验证，同时其参数设置了客户端证书链最大的验证深度。但是，对于客户端来说，这仅是一个可选的动作，客户端如果不发送证书给服务器进行验证，也可以顺利建立 SSL 连接。而如果使用 Verify 选项表示客户端必须发送一个证书进行验证，否则 SSL 握手将失败。

选项 CApath 指定了服务器存放可信任证书文件的目录，这些证书文件在对客户端证书验证的时候需要使用，此外，在要求建立服务器证书链的时候，目录中的这些证书也有可能被使用。

选项 CAfile 指定一个存储证书的文件，该文件包含了服务器信任的 CA 证书，跟 CApath 一样，这些证书在进行客户端证书验证的时候需要使用，在建立服务器证书链的时候也可能被使用。此外，客户端进行 SSL 握手要求客户端发送用户证书时，这些证书

将作为服务器可接受的信任 CA 列表发送给客户端。

如果服务器端发送一个空的信任 CA 列表给客户端（比如没有定义 CAfile 文件），虽然理论上应该是违反协议规定的，但很多 SSL 客户端认为这时候服务器能够接受任何 CA 签发的证书，这在调试程序的时候可能反而会有些帮助。

### 6. Web 服务器模拟选项 www，WWW 和 HTTP

使用 www 选项后，当 SSL 客户端连接上 SSL 服务器的时候，将接收到服务器发回的一些状态信息。这些状态信息包括连接使用的算法和会话参数等，并且被格式化成 HTML 的格式，所以，经常可以被 Web 浏览器 SSL 客户端接受和解释。

使用 WWW 选项服务器端将模拟一个简单的 Web 服务器。指令程序所在的目录将被解释成网页的当前目录。例如如果客户端的 URL 请求是 https：//myhost/page. html，服务器将把 ./page. html 发回给客户端。

HTTP 选项跟 WWW 选项的功能基本上是一样的，唯一的区别是，使用 HTTP 选项后，当前目录中所有被请求的文件，都假定是符合 HTTP 格式的文件。而选用 WWW 选项则 s_server 在发送被请求的文件给客户端之前，会根据后缀名判断被发送的文件是普通文件还是 HTTP 文件，并告知客户端。

如果上述三个选项都没有使用，当一个 SSL 客户端成功连接到服务器后，客户端所发送过来的任何数据都会在服务器端显示出来，服务器端任何键盘输入也都会被发送给客户端。此时，一些单个的特殊字符在指令行可以输入，以达到一些特殊的目的，支持的字符指令及其含义如表 12-1 所示。

表 **12-1　s_server** 支持的单个字符指令表

| 字母指令 | 含义说明 |
|---|---|
| q | 中断当前连接，但不关闭服务程序，仍可以接受新连接 |
| Q | 中断当前连接并退出服务程序 |
| r | 重新进行 SSL 会话协商 |
| R | 重新进行 SSL 会话协商，并要求客户端发送证书 |
| P | 服务器端向 TCP 层直接发送一些明文，这将导致客户端认为服务器没有按协议进行通信而断开连接 |
| S | 打印出会话缓冲区的状态信息 |

### 7. 其他选项 context，no_tmp_rsa，id_prefix 和 hack

context 选项用于设置 SSL 结构体的 ID，可以设置为任何值，如果不设置，将使用默认值。

某些算法簇会要求使用一个临时的 RSA 密钥，no_tmp_rsa 选项将禁止临时 RSA 密钥的产生。

id_prefix 选项指定 SSL/TLS 所有会话 ID 的前缀。这个功能在测试多个 SSL 服务器的时候会非常有用（比如通过 SSL 代理的方式），每个服务器都会产生特定范围的会话 ID，方便识别和调试。

hack 选项是为了兼容早期版本的 Netscape 浏览器，一般来说使用很少。

8. 简单的例子

这里仅举一两个非常简单的例子，关于 s_server 各个指令更深入的掌握，需要读者自己动手去做实验。下面的例子必须在 s_server 程序的目录下放一个 server.pem 的文件，文件应该存有一个服务器数字证书和其相应的私钥，否则将出错。该程序模拟了一个 SSL 服务器和 Web 服务器：

```
OpenSSL>s_server -accept 443 -www
```

下面的例子不需要 server.pem 文件的支持，但因为使用 nocert 选项禁止了 RSA 相关算法簇的使用，仅支持匿名 DH 算法簇，普通的 IE 浏览器连接将不能成功：

```
OpenSSL>s_server -accept 443 -www -nocert
```

## 12.4　SSL 会话过程深入分析

在使用 s_client 或者 s_server 指令的 debug 选项进行调试的时候，通常会看到其中包括一段诸如下面的内容：

```
—BEGIN SSL SESSION PARAMETERS—
MHUCAQECAgMABAIABAQgAvtzcmvTViXDDkzhYpvL89olKiMEC81xgaEzPsYGAI0E
MFHm3EcrAul6XMHo9fxUNaoBTrhXMHuox9A6GiGwrbP/v++hLyYcf0M2WBPhgu+6
kqEGAgRDRV3WogQCAgEspAYEBAEAAAA=
—END SSL SESSION PARAMETERS—
```

这就是 SSL 会话（Session）结构的 PEM 编码，你可能看不懂上述这些不知所云的 PEM 编码内容。但是，分析一个 SSL 连接中所有会话的实际内容对于我们分析 SSL 连接及调试程序都是非常有用的，所以 OpenSSL 提供了一个翻译工具，帮助我们读懂上面的内容。

所以，上面的方法，就是获取编码 SSL 会话结构的一种途径和需要之一。

1. 功能概述和指令格式

OpenSSL 提供的 sess_id 指令就是用于处理保存下来的 SSL 会话（Session）编码结构，根据指令选项要求打印出其中的信息。sess_id 一般作为一个调试工具使用，并需要用到 SSL 协议的相关知识，这听起来有些难度，不过，幸运的是，很少用户会用到这个指令。其指令格式为：

```
OpenSSL> sess_id[-inform arg][-outform arg][-in filename][-out filename]
[-text][-noout][-cert][-context ID]
```

2. 输入和输出格式选项 inform 和 outform

跟其他指令一样，这两个选项分别指定了输入和输出格式，其中，outform 指定的也是输出编码会话（Session）结构的格式，不会影响其他可读内容的输出格式。目前支持的格式包括 DER 和 PEM，默认是 PEM 格式。

PEM 格式的 SSL 会话包含在一个固定结构中，第一行和最后一行格式如下：

```
—BEGIN SSL SESSION PARAMETERS—
—END SSL SESSION PARAMETERS—
```

3. 输入和输出文件选项 in 和 out

in 和 out 选项指定了输入和输出文件名，输入文件一般存储的是一个编码 SSL 会话结构，输出文件则输出 SSL 会话信息（也包括 SSL 的编码会话结构）。默认情况下是标准输入和输出设备。

4. 输出内容选项 text，noout 和 cert

选项 text 将告诉指令对 SSL 会话结构进行完全的明文解释，输出所有可能的内容，其输出的信息典型如下面所示：

```
SSL-Session:
Protocol:TLSv1
Cipher:0016
Session-ID:871E62626C554CE95488823752CBD5F3673A3EF3DCE9C67BD916C809914B40ED
Session-ID-ctx:01000000
Master-Key:A7CEFC571974BE02CAC305269DC59F76EA9F0B180CB6642697A68251F2D2BB57E51DBBB
4C7885573192AE9AEE220FACD
Key-Arg:None
Start Time:948459261
Timeout:300 (sec)
Verify return code 0 (ok)
```

这些字段的含义如表 12-2 所示。

表 12-2 SSL 会话字段含义

| 字段标识 | 含　义 |
|---|---|
| Protocol | 使用的 SSL 协议版本 |
| Cipher | 会话使用的算法簇，详细代码含义需要查看 SSL 和 TLS 相关协议 |
| Session-ID | 十六进制编码的会话 ID |
| Session-ID-ctx | 十六进制编码的会话 ID 结构内容 |
| Master-Key | SSL 会话主密钥 |
| Key-Arg | 密钥参数，仅在 SSL 2.0 有用 |
| Start Time | 会话开始时间，以整数的方式标识 |
| Timeout | 会话闲置生存时间，以秒为单位 |
| Verify return code | 验证客户端证书的返回码 |

当 SSL 会话结构中包含证书时，cert 选项告诉指令输出该证书结构，如果同时还使用 text 选项，那么将输出证书的可读形式。

一般情况下，sess_id 指令最后都会输出 SSL 会话的编码结构，使用 noout 选项后，将只输出 text 选项或者 cert 选项要求输出的内容，而不再输出 SSL 会话的编码结构。

5. 其他选项 context

context 选项用来指定一个特殊的 ID 替换编码里面的 ID 作为输出的会话 ID，这个 ID

可以为任意字符串。

6. 简单的例子

下面的指令解析出指定的 SSL 会话结构信息，并且要求不输出编码的 SSL 会话结构：

```
OpenSSL>sess_id -in sess.pem -text -noout
```

## 12.5　本 章 小 结

本章围绕 OpenSSL 提供的几个 SSL 指令进行了详细的介绍，一方面加深了读者对 SSL 协议的感性认识，另一方面也让读者掌握了如何调试和测试一个 SSL 服务器或者客户端程序。

本章首先结合 s_client 和 s_time 指令讲述了如何测试一个 SSL 服务器，并且对 SSL 客户端应该具备的通用特性作了介绍。

然后，本章结合 s_server 指令，讲述了如何利用 s_server 测试 SSL 客户端，并且对 SSL 服务器和客户端的特点做了进一步阐述。

最后，本章介绍了 OpenSSL 提供的一个分析 SSL 会话结构的小工具 sess_id，对 SSL 的会话结构进一步进行剖析，加深了读者对 SSL 协议实现的理解。

# 参 考 文 献


[1] TANENBAUM A S. 计算机网络. 4 版. 北京：清华大学出版社，2004.

[2] SCHNEIER B. 应用密码学：协议、算法和 C 源程序. 北京：机械工业出版社，2000.

[3] STALLINGS W. 密码编码学与网络安全：原理与实践. 3 版. 北京：电子工业出版社，2004.

[4] NASH A. 公钥基础设施（PKI）：实现和管理电子安全. 北京：清华大学出版社，2002.