Java 卡应用系统的开发

目　　录

第 1 章 引言

1.1 Java卡

Java 卡(Java Card)就是能够运行 Java 程序的智能卡(Smart Card)或 IC 卡(Integrated Circuit Card)。Java 卡的硬件包括：CPU、ROM、RAM 和 EEPROM。

在 1995 年推出 Java 后，Sun 公司专门成立了 JavaSoft 部门，以推广 Java 在各个领域内的应用。Schlumberger（斯伦贝谢）于 1996 年 10 月提出了 Java Card 1.0，它只有 API 协议。随后 Schlumberger 和 Gemplus（金普斯）倡议建立一个促进在智能卡领域应用 Java 的机构，1997 年 4 月，Java Card 论坛正式宣布成立。Java Card 论坛的成立目的就是根据厂商的意见来完善这个 Java Card API 规范（2.0 版即由此论坛提出并获得通过），使之最终成为多应用智能卡的首选编程语言，在智能卡领域确立 Java 的权威地位。

所谓 Java Card 规范，主要是定义了 Java 作为一种独立于平台的编程技术在智能卡上的应用。从 Java Card 2.0 开始，Java 卡规范包括了应用编程接口(Application Programming Interface, API)、Java 虚拟机（Java Virtual Machine, JVN）和 Java Card 运行环境（Java Card Runtime Environment，JCRE)。其中，API 规范指定了 Java Card 的核心类库；虚拟机规范议指定了适合于 Java Card 的虚拟机的特点；JCRE 规范指定了更细致的行为，例如内存管理、安全措施等。目前，Java 卡规范 2.2 已经发布。

除了 Java Card 规范外，还可以从网上免费下载 Java Card 开发包（Development Kit），这个开发包提供了一套完整的测试、转换工具，以及一些 Java Card 的应用。

一张 Java 卡在出厂时就，就加载了以下软件：卡操作系统（Card Operating System）、Java Card 虚拟机，API 类库和可选的 Java 卡应用（Applet）。随后，在初始化和个人化时，再向 EEPROM 中写入新的 Java 卡应用和数据。和 PC 上的 Java 虚拟机不同，Java Card 上的虚拟机处于永远运行的状态，没有电源被理解为无限的时钟周期，而虚拟机所创建的对象一般被写入到 EEPROM 中。

1.2 Java卡的程序员视图

对于开发 Java 卡应用系统的程序员来说，Java 卡上可见的对象有两类：卡管理器（CardManager）和 Java 卡应用（Applet）。如图 1.1。

卡管理器（CardManager）又称为发卡商安全域（Card Issuer's Security Domain），一张 Java 卡上可以有多个安全域，一个安全域中可以有多个 Applet。卡管理器最常见的安全域。Java 卡应用则是根据实际需求开发的 Java 程序。

在应用系统中，制卡程序与卡管理器交互，在通过认证后，可以将包含 Applet 的包下载到 Java 卡上，并进行安装，生成 Applet 的实例，可以删除 Applet 的实例和 Applet 包，还可以读取卡管理器保存的状态数据，并设置卡管理器的状态。制卡程序还向 Applet 实例写入初始数据或个人化数据。

在应用系统中，终端应用程序与 Applet 交互，在通过认证后，读写 Applet 中保存的数据。

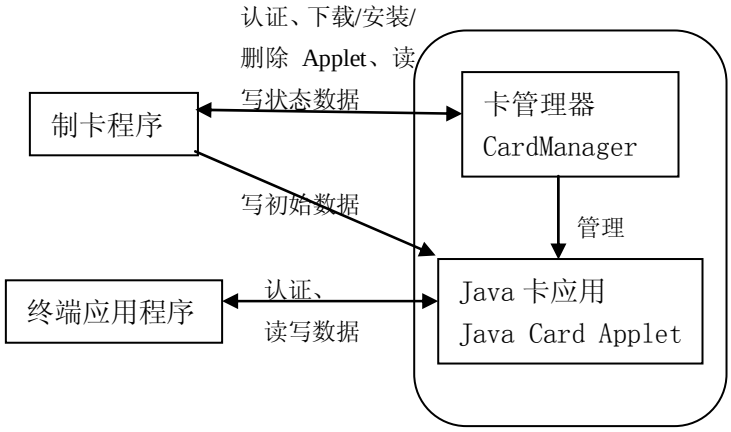

图 1.1　Java 卡的程序员视图

Java 卡应用对于应用程序员可见的标识就是它的应用标识符 AID（Application IDentifier），AID 长度为 5—16 字节。卡管理器也有一个 AID，一般为 0xA0000000030000 或 0xA000000003000000，用 ASCII 串表示为" A0000000030000"或" A000000003000000"。卡管理器可以认为是一个特殊的 Java 卡应用。

Java 卡应用（包括卡管理器）对于应用程序员可见的接口，就是 Java 卡应用与主机通信时使用的应用协议数据单元（Application Protocol Data Unit, APDU）。Java 卡 Applet 与主机的接口的必须支持 JCRE 的 select 命令 APDU 和具体应用的 APDU。Select 命令的 APDU 必须遵照 Java 卡标准，具体应用 APDU 的结构必须遵照 ISO7816。

根据 ISO7816-4，一个 APDU 由两个结构组成：主机发送给智能卡的命令 APDU（Command APDU, C_APDU），智能卡响应主机的响应 APDU（Response APDU, R_APDU）。

C-APDU 格式如下表：

命令头（必有）				命令体（可选）		
CLA	INS	P1	P2	Lc	Data	Le

其中，命令头有 4 个字节：CLA 为 APDU 类别；INS 表示要执行的指令；P1 和 P2 表示指令的参数；命令体可选部分，长度不固定（>=0 ）：Lc 为数据段的长度，单位为字节；Le 为主机希望智能卡响应的字节数目。

对于命令类型字节 CLA，ISO 标准类型命令为 0x00，行业专用类型命令为 0x80，自行开发的应用的专有类型命令可以采用 0xA0、0xC0 等。如果命令需要附加报文认证码（Message Authentication Code, MAC）或对数据部分加密，则对应的 CLA 的分别为 0x04、0x84、0xA4 和 0xC4 等。

R-APDU 格式如下表：

响应头可选部分	响应体必有部分	
Le 字节的数据段	SW1	SW2

其中，SW1 和 SW2 称为状态字，SW1 为高 8 位，Sw2 为低 8 位。

这样，有 4 种类型的 C-APDU 和 R-APDU，如下表所示：

C-APDU	R-APDU
CLA INS P1 P2	SW1 SW2
CLA INS P1 P2 Le	Le 字节数据 SW1 SW2
CLA INS P1 P2 Lc Lc 字节数据	SW1 SW2
CLA INS P1 P2 Lc Lc 字节数据 Le	Le 字节数据 SW1 SW2

对于支持 Java 卡规范 2.2 及以上的 Java 卡，终端应用程序可以将 Java 卡应用看成是一个远程对象，使用远程方法调用 RMI 来于 Java 卡应用交互。由于终端应用可能还要与一般的 CPU 卡交互，APDU 对于 Java 卡和非 Java 卡都是相同的，因此，下面我们不考虑以 RMI 方式与 Java 卡应用交互。

1.3 Java卡应用系统的开发

1.3.1 应用系统体系结构

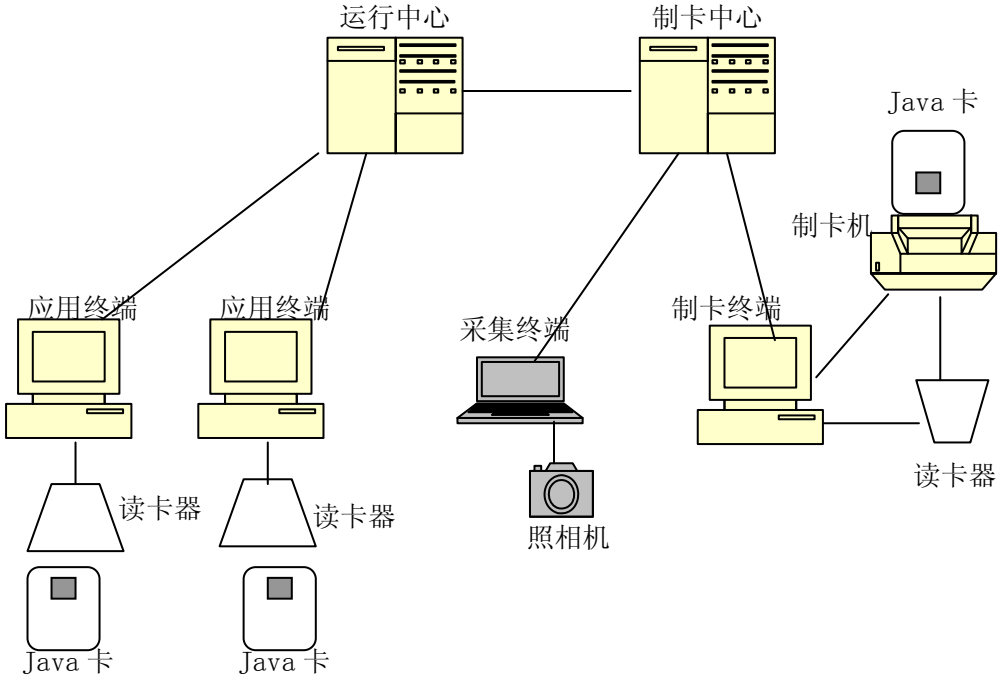

图 1.2　简单的 Java 卡应用系统结构

一个简单的 Java 卡应用系统如图 1.2 所示。应用系统划分为两大部分：制卡和运行。制卡中心向运行中心传输已经制作好的 Java 卡信息，运行中心向制卡中心传输因为丢失和损坏而需要重新制卡的 Java 卡信息。运行中心还维护有关黑名单和灰名单等信息。

在制卡部分，采集终端用于采集 Java 卡用户的基本信息，然后，将用户的基本信息传输到制卡中心。制卡机中放置有空白的 Java 卡，制卡机与制卡终端可以直接连接，以打印卡面信息，制卡机也通过读写器与制卡终端间接连接，这样，制卡终端首先将 Java 卡应用下载到制卡机中的 Java 卡上，然后，将用户的个人信息写入 Java 卡应用中。制作完毕的 Java 卡发放给用户使用。

用户持 Java 卡在应用终端进行各种业务交易，以及进行 Java 卡的挂失和解挂等。

由于制卡中心、运行中心、终端应用和制卡应用都与具体的系统相关，下面，将主要介绍所有应用系统的共同部分，包括：密钥系统、Java 卡应用、终端应用读写 Java 卡的接口、制卡应用读写 Java 卡的接口。

1.3.2 有关规范

Java 卡应用系统中涉及的有关规范有：（1）Java 卡规范。要求 Java 卡遵循 Java 卡规范，保证了应用系统中的 Applet 可以在不同厂商的 Java 卡上运行，而不必修改。（2）OpenPlatform/GlobalPlatform 规范。这个规范主要规定了卡管理器的行为。要求卡管理器遵循这个规范，保证了用语下载和安装 Applet 的制卡程序可以与不同厂商的 Java 卡上的卡

管理器交互，而不必做修改。（3）PC/SC 规范。这个规范规定了访问读卡器的接口。要求读卡器遵循这个规范，保证了制卡程序和终端应用程序可以以一致的方式，访问不同厂商的读卡器。

1.3.3 简单的密钥系统

当制作完毕的 Java 卡发放给用户后，Java 卡就游离于应用系统之外，Java 卡应用的数据就存在着被非法读写的可能，同时，也可能有伪造的 Java 卡进入应用系统，因此，需要提供保护 Java 卡应用的数据的手段，以及鉴别伪造的 Java 卡的手段。常规的手段是采用密钥验证。

所谓密钥验证，就是要求终端应用和 Java 卡应用两者对于密钥有相同的知识。对于采用对称密钥技术（即加密和解密数据的密钥是相同的）的系统，如 DES（data encryption standard），要求终端应用和 Java 卡应用有相同的密钥。一个 Java 卡应用，至少有两个密钥，一个读密钥，一个写密钥，只有通过读密钥验证后，终端应用才能读 Java 卡应用的数据，只有通过写密钥验证后，终端应用才能写或读 Java 卡应用的数据。

在一个系统中，Java 卡的数量一般为 10^5 数量级，一张 Java 卡上可以存在几十个应用，一个 Java 卡的读密钥和写密钥不相同，同一个 Java 可应用的读/写密钥在不同的 Java 卡上也不相同。因此，终端应用必须知道 10^6 数量级的密钥。显然，这些密钥不可能保存在应用终端上，也不可能保存在运行中心。这还没有考虑新增加的 Java 卡和作废的 Java 卡，以及新增加的 Java 卡应用和作废的 Java 卡应用。

一个简单的解决方案是，设计一个系统的根密钥，为每个 Java 卡应用，设计两个分散参数，称为读/写分散参数，系统根密钥与读/写分散参数作用，形成各个应用的读/写根密钥。对于每张 Java 卡，可以取一个分散参数，例如，卡号，各个应用的读/写根密钥与卡号作用，形成每张 Java 卡上的各个应用的读/写密钥。这样，终端应用只需要知道若干应用的读/写根密钥，再临时读取用户的卡号，就可以计算出某个 Java 卡的某个 Java 卡应用的读/写密钥。

当然，应用的读/写根密钥也不能由终端应用保存，一个解决方案是保存在一个 Java 卡应用中，这个 Java 卡应用存在于一张 Java 卡上，这张 Java 卡称为机具卡，这个 Java 卡应用称为机具卡应用。这样，当应用终端需要读写 Java 卡应用的数据时，用户的 Java 卡（称为用户卡）和机具卡都必须同时插入应用终端的两个读卡器中，才能进行读写密钥的验证。通常，一个系统中可能有 10^5 数量级的机具卡分散在不同地理位置，每张机具卡上有一个机具卡应用，一个机具卡应用只有部分应用的读/写根密钥，因为终端应用一般不会读写所有 Java 卡应用的数据。

在制作用户卡时，需要计算用户卡上各个 Java 卡应用的读/写密钥。保存所有应用的读/写根密钥的 Java 卡应用称为发行卡应用，发行卡应用存在于一张发行卡中，它根据用户的卡号，计算出各个 Java 卡应用的读/写密钥，写入 Java 卡应用中。这样，当制卡应用需要制作用户卡时，空白用户卡和发行卡都必须同时插入制卡终端的两个读卡器中，才能进行制卡。通常，一个系统中最多可能 10 数个的发行卡，并且是地理上集中使用。

当然，最后会有一个根密钥 Java 卡应用，它保存有系统的根密钥，并根据各个应用的读/写分散参数，生成应用的读/写根密钥，注入到机具卡和发行卡中。根密钥应用存在于一张根密钥卡中，整个应用系统只有一张根密钥卡。

这样，应用系统中存在 4 种不同作用的 Java 卡，它们之间的关系如图 1.3 所示。在制作发行卡和机具卡时，需要根密钥卡来计算它们需要的应用根密钥。在制作用户卡时，需要发行卡来计算用户卡中的应用密钥。由于整个应用系统只有一张根密钥卡，因此，根密钥卡

采用人工制作，利用 Java 卡厂商提供的工具，进行根密钥应用的下载、安装，以及数据的初始化。

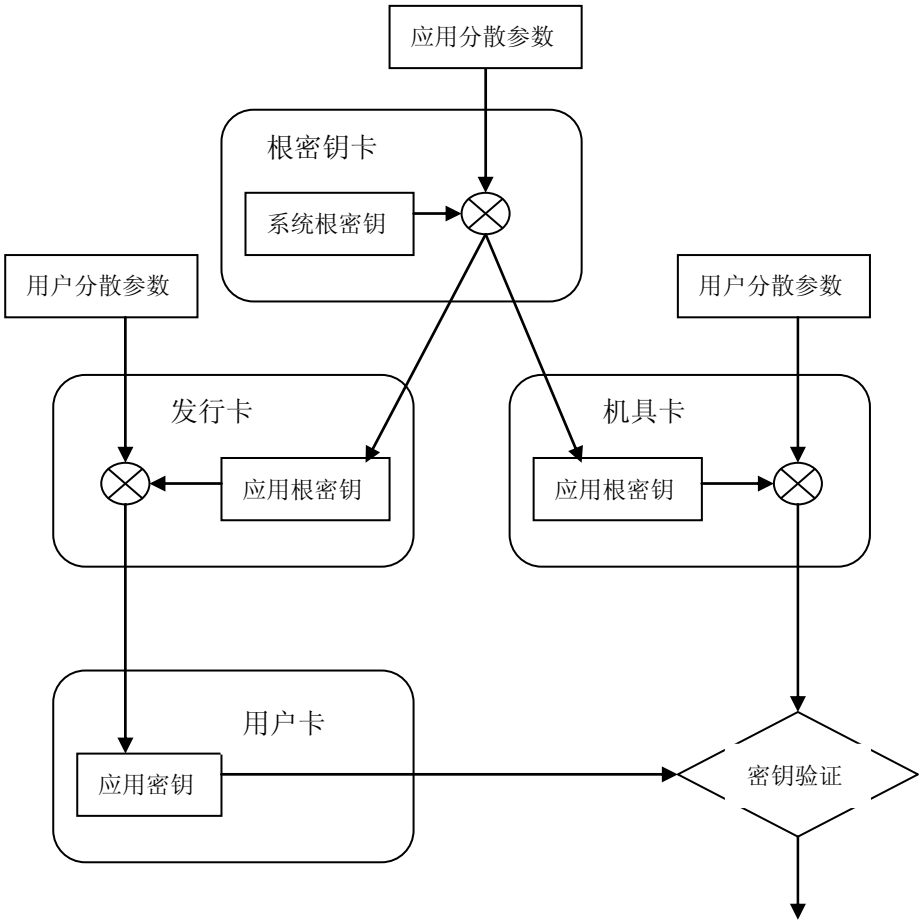

图 1.3　4 种卡之间的关系

当终端应用验证 Java 卡是否为伪造的过程，称为内部认证（Internal Authentication），内部认证的过程如图 1.4。当 Java 卡应用验证终端应用是否可以读写其数据的过程，称为外部认证（External Authentication），外部认证的过程如图 1.5。可以看出，内部认证和外部认证是对称的。如果采用特殊的方式，内部认证和外部认证可以结合在一起，我们将在后面介绍。认证完成后，就认为在终端应用与制卡应用之间，建立了一个通道。

卡管理器是一个特殊的 Java 卡应用，它也有 3 个密钥来保护 Java 卡上的 Applet 包和实例，分别称为认证密钥（Authentication Key）、计算报文认证码（Message Authentication Code, MAC）的密钥（MAC Key）和密钥加密用的密钥（Key Encryption Key, KEK）。只有通过密钥验证后，制卡程序才能向 Java 卡下载 Applet 包，并安装 Applet 实例，以及删除 Applet 实例和包。在卡制造商交付的空白 Java 卡中，卡管理器的密钥是已知的，在制卡完成后，必须更改这 3 个密钥。卡管理器新密钥的根密钥也采用系统的根密钥和一个分散参数来计算，每张 Java 卡上的卡管理器的密钥则采用 Java 卡的序列号作为分散参数，与卡管理器的根密钥混合而成。

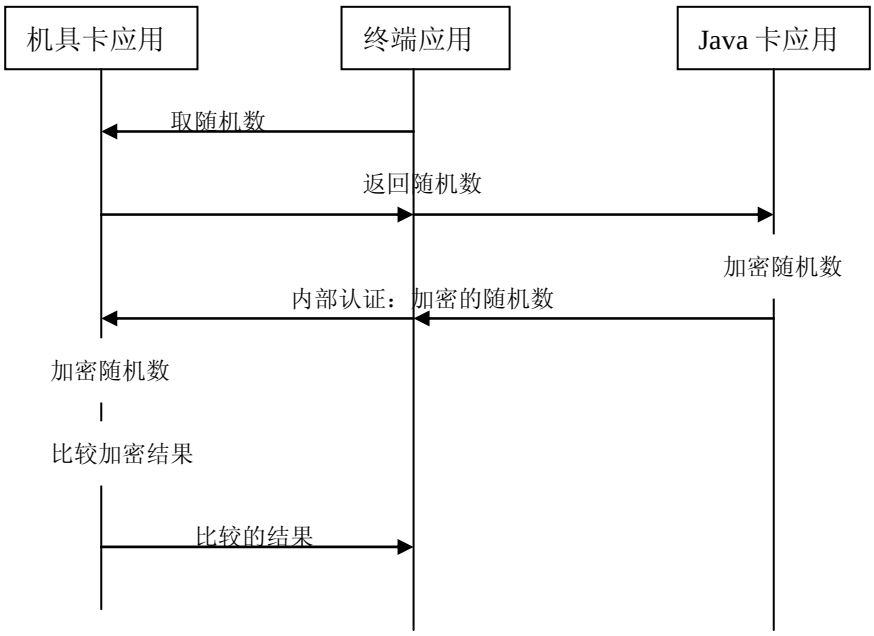

图 1.4　　内部认证流程

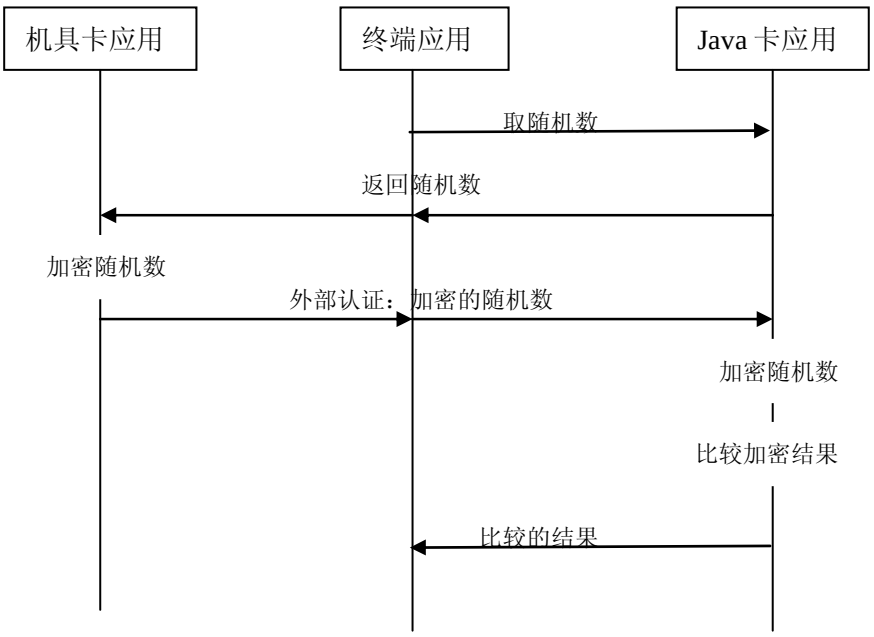

图 1.5　　外部认证流程

系统根密钥应用和发行卡应用需要有计算卡管理器密钥的功能，但由于与卡管理器交互还需要其它的功能，因此，单独设立一个称为发行管理的应用，它存在于系统根密钥卡上和发行卡上，分别向根密钥应用和发行卡应用取得制卡根密钥，根据 Java 卡的序列号，计算卡管理器的新密钥，并实现其它功能。

1.3.4 Java卡应用

一个 Java 卡应用就是一个 Applet 对象。这个对象的主要数据成员有两类：密钥和信息。密钥用于控制信息的读写，密钥一般用字节数组 byte[]存储。信息则存储在布尔 bool 变量或数组、字节 byte 变量或数组、短整数 short 变量或数组中，信息按照其应用关联性，划分为多个信息域，每个信息域由一组密钥控制读写。

Java 卡应用的结构如下：

```
public class MyApplet extends Applet [implements MyInterface]
{
    //常量定义
    ……
    //变量定义
    ……
    //私有方法
    ……
    //扩展 Applet 的方法

    //在安装 Applet 时，由 JCRE 调用。
    //bArray：存放安装参数的数组
    //bOffset：安装参数在数组中的起始位置
    //bLength：安装参数字节数目
    public static void install (byte[] bArray, short bOffset, byte bLength)
    {
    }

    //当 Java 卡 Applet 被选中时，由 JCRE 调用。
    //Java 卡 Applet 可以定义 select（）完成初始化，否则，JCRE 调用父类的 select（）。
    public boolean select（）
    {
    }

    //当 Java 卡 Applet 被放弃时，由 JCRE 调用。
    //Java 卡 Applet 可以定义 deselect（）完成清除，
    //否则，JCRE 调用父类的 deselect（）。
    public boolean deselect（）
    {
    }

    //返回一个可共享的接口对象 SIO（Shareable Interface Object）。
    //对于作为服务类的 Java 卡 Applet，必须实现这个方法。
    public Shareable getShareableInterfaceObject (AID clientAID, byte parameter)
    {
    }
```

```
    //Java 卡 Applet 必须实现这个方法，解释每一个 APDU 命令，并调用相关方法执行。
    //apdu：传递给 Java 卡应用的 C-APDU
    public abstract void process（APDU apdu）
    {
    }
}
```

Java 卡 Applet 在被选择（select)后，如果收到 APDU 指令，process（)方法就会被调用，所有的有关 APDU 指令的处理都会在 process（)方法中被执行。一般需要执行以下五个过程来处理 APDU 指令：

（1）获得 APDU 指令的缓冲（buffer)：用 getBuffer（） 方法来得到 APDU 指令的缓冲，现在，在缓冲中只包含指令信息（CLA，INS，P1，P2，Lc)。

（2）获取 APDU 指令中的数据：如果 APDU 指令中包含数据信息，那么必须用 setIncomingAndReceive（）方法来获取数据域中的信息。这是因为数据信息并不是必须的，有些 APDU 指令是没有数据信息的，所以在处理 APDU 指令时，一般先得到指令（INS)，根据指令判断这一指令是否有数据信息，若有，则使用 setIncomingAndReceive（） 方法，将数据信息读取到缓冲中，并可从缓冲的第六个字节起来获取数据信息；若无，则不调用 setIncomingAndReceive（）方法。

（3）分析处理 APDU 数据：分析处理 APDU 数据是指 Applet 收到数据后，对数据进行分析，并按照一定的流程来处理数据的过程。

（4）返回数据：Applet 处理完数据后，向终端返回状态字与数据的过程。先用 setOutgoing（）方法来设置数据传送的方向，表明 Applet 要输出数据，setOutgoing（）方法则会返回 Applet 能够回传数据的长度 Le。然后，Applet 用 setOutgoingLength（）方法告诉终端将实际回传数据的长度。最后，Applet 将要回传的数据设置到 APDU 的缓冲（buffer）中，用 sendBytes（）方法来传送数据。

（5）返回状态字：当 process（）方法正常执行后，Applet 会给终端自动发出 0x9000 状态字，表明 APDU 指令操作成功。如果在 process（）方法执行过程中，Applet 发现了一些错误，可以用 ISOException.throwIt（short reason） 方法来向终端发出一些特定的状态字，来告知终端发生了什么样的错误。

1.3.5 终端应用接口

从上面可以知道，终端应用与 Java 卡应用的交互比较复杂，除了用户卡外，还需要机具卡的参与。为此，我们可以创建一个接口，将常见的终端应用与 Java 卡应用的交互封装在这个接口中，以简化终端应用的编程。实际接口采用 VC++的动态链接库实现，动态链接库提供一组函数，供采用不同的编程语言开发的终端应用调用。

1.3.6 制卡应用接口

制卡应用在制卡时，与 Java 卡应用的交互也比较复杂。在制作用户卡时，需要用户卡和发行卡的参与；在制作发行卡和机具卡时，还需要根密钥卡的参数。为此，我们也可以创建一个接口，将常见的制卡应用与 Java 卡应用的交互封装在这个接口中，以简化制卡应用的编

程。实际接口采用 VC++的动态链接库实现，动态链接库提供一组函数，供采用不同的编程语言开发的终端应用调用。

第 2 章　Java卡应用的开发

2.1　应用概述

现在，我们来开发 3 个 Java 卡应用。

第 1 个应用 Applet00 的数据有：6 字节的用户密码，一个 16 字节的信息域存放 Java 卡编号。它的密钥包括：主控密钥、验卡密钥、密码重装密钥。通过主控密钥验证后，才能更新这 3 个密钥，主控密钥还作为这 3 个密钥的传输加密密钥；为简单起见，主控密钥先固定在程序中，以后通过主控密钥验证后，再更新。验卡密钥用于加密主机传送来的数据，供主机验证卡的有效性。通过密码重装密钥验证后，可以将用户的密码重新设置为初始密码。读信息域的数据不需要先通过密钥验证。

第 2 个应用 Applet01 的数据分为 4 个信息域，它们的长度分别为 256、128、1024、1024 字节，都是二进制格式，数据含义由主机解释。它的密钥包括：主控密钥、读密钥、写密钥。主控密钥的作用同前。通过读密钥验证后，可以读各个信息域的数据。通过写密钥验证后，可以读写各个信息域的数据。

第 3 个应用 Applet02 的数据分为 4 个信息域，它们的长度分别为 300、200、400、4000 字节，都是二进制格式，数据含义由主机解释，一些信息域划分为记录，记录的长度和记录内各字段的含义也由主机解释。它的密钥包括：主控密钥、读密钥、写密钥一、写密钥而。主控密钥和读密钥的作用同前。通过写密钥二验证后，可以读写第一个信息域的数据。通过写密钥一验证后，可以读写其它信息域的数据。

在制作用户卡时，需要向各个应用写入（更新）密钥，并向各个信息域写入数据。这时，不需要通过任何密钥验证。在制作完成后，更新密钥和读写数据时，必须先通过相关的密钥验证。

2.2　确定应用的AID

一张 Java 卡上可以有多个 Applet 类，每个 Applet 类可以有多个实例对象，各个实例对象通过应用标识（AID，Application Identifier） 相互区分。根据 ISO7816，一个 AID 包括两个部分：5 个字节的资源标识（RID，Resource Identifier）、0--11 字节的特性标识扩展（PIX，Property Identifier eXtension）。

我们给第 1 个应用 Applet00 赋予一个 AID=0x41:0x50:0x50:0x30:0x30，即字符串“APP00”，它的第 1 个实例的 AID 为：0x41:0x50:0x50:0x30:0x30:0x01。

同样。我们分别给第 2、3 个应用 Applet01 和 Applet02 赋予 AID=0x41:0x50:0x50:0x30:0x31、 AID=0x41:0x50:0x50:0x30:0x32，即字符串“APP01”、“APP02”，它们的第 1 个实例的 AID 分别为：0x41:0x50:0x50:0x30:0x31:0x01、0x41:0x50:0x50:0x30:0x32:0x01。

2.3　定义应用与主机的接口

应用与主机的接口就是就是 Java 卡应用与主机通信时使用的应用协议数据单元（Application Protocol Data Unit, APDU），也就是 Java 卡应用的指令集。在设计 Java

卡应用的指令集时，可以采用现有的一些指令集，如《中国金融集成电路（IC）卡卡规范》中的指令集。如果不能满足需要，可以自行设计应用专有指令。

2.3.1 Applet00 的指令集

（1）选择应用

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x04	0x00	0x06	0x41:0x50:0x50:0x30:0x30:0x01	无

响应 APDU

数据域	状态字	状态字含义
	0x9000	成功
	0x6A82	应用不存在

（2）完成个人化

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xEF	0x00	0x00	0x00	无	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功

（3） 取随机数

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x84	0x00	0x00	无	无	0x08

响应 APDU

数据域	状态字	状态字含义
随机数	0x9000	成功
	0x6700	长度错误

（4）验证密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x82	密钥编号	0x00	0x08	加密后的随机数	无

P1=0x00：主控密钥。

P1=0x02：密码重装密钥。验证通过后，密码为缺省密码。

响应 APDU

数据域	状态字	状态字含义

无	0x9000	成功
	0x6709	没有密码
	0x670E	没有随机数
	0x670A	重装密码失败
	0x6715	没有传输密钥
	0x6722	主控制密钥无效

（5）验证验卡密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x88	0x01	0x00	0x08	随机数	0x08

响应 APDU

数据域	状态字	状态字含义
加密后的随机数	0x9000	成功
	0x6708	没有主控密钥

（6）更新应用密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xD4	密钥编号	0x00	0x18	密钥	无

P1=0x00：主控密钥。

P1=0x01：验卡密钥。

P1=0x02：密码重装密钥。

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6704	失败

（7）选择信息域

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x00	0x00	0x02	文件标识	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6A82	失败

（8）读信息域数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le

0x00	0xB0	偏　移（高）	偏　移（低）	0x00	无	长度

响应 APDU

数据域	状态字	状态字含义
数据	0x9000	成功
	0x6706	超过文件长度
	0x670F	读失败

（9）更新信息域数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xD6	偏　移（高）	偏　移（低）	长度	数据	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功 SW_NO_ERROR
	0x6706	超过文件长度
	0x6707	写失败

（10）验证密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x20	0x00	0x00	0x06	密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x670B	密码无效

（11）修改密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0x5E	0x00	0x00	0x0C	旧密码新密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x670B	旧密码无效
	0x670C	修改失败

2.3.2 Applet01 和Applet02 的指令集

（1）选择应用

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x04	0x00	0x06	AID	无

AID=0x41:0x50:0x50:0x30:0x31:0x01，应用 2。

AID=0x41:0x50:0x50:0x30:0x32:0x01，应用 3。

响应 APDU

数据域	状态字	状态字含义
	0x9000	成功
	0x6A82	应用不存在

（2）完成个人化

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xEF	0x00	0x00	0x00	无	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功

（3） 取随机数

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x84	0x00	0x00	无	无	0x08

响应 APDU

数据域	状态字	状态字含义
随机数	0x9000	成功
	0x6700	长度错误

（4）验证密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x82	密钥编号	密钥编号	0x08	加密后的随机数	无

P1=0x00：主控密钥。

P1=0x01：读密钥。

P1=0x02：写密钥。P2=0x00：写密钥一；P2=0x01：写密钥二。

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x670E	没有随机数
	0x670A	重装密码失败
	0x6715	没有传输密钥

（5）更新应用密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xD4	密钥编号	0x00	0x18	密钥	无

P1=0x00：主控密钥。

P1=0x01：读密钥。

P1=0x02：写密钥。

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6704	失败

（6）选择信息域

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x00	0x00	0x02	文件标识	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6A82	失败

（7）读信息域数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xB0	偏　移（高）	偏　移（低）	0x00	无	长度

响应 APDU

数据域	状态字	状态字含义
数据	0x9000	成功
	0x6706	超过文件长度
	0x670F	读失败

（8）更新信息域数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xD6	偏　移（高）	偏　移（低）	长度	数据	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6706	超过文件长度
	0x6707	写失败

（9）读信息域记录

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xB2	记录号	0x00	0x00	无	长度

响应 APDU

数据域	状态字	状态字含义
数据	0x9000	成功
	0x6706	超过文件长度
	0x670F	读失败

（10）更新信息域记录

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xDC	记录号	0x00	长度	数据	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6706	超过文件长度
	0x6707	写失败

2.4　应用设计

由上可知，每个 Applet 主要管理两类数据：二进制文件和读写控制密钥；对文件的操作主要有：选择、读、写，对密钥的操作主要是更新和验证。我们可以得到如图 2.1 的简单对象模型。

每个 Applet 由一个或多个类组成。在我们的应用中，每个 Applet 的数据存储包括：二进制文件、访问控制密钥，因此，每一部分数据用一个类来管理。我们修改图 2.1 的对象模型，设计了 4 个类：（1）文件类 FileService：管理单个二进制文件的读写，只有一个方法 fileReadWrite（），对应图 2.1 中 File。（2）密钥管理类 KeyService：管理应用的读写控制密钥和主控密钥的更新和验证，主要方法包括：updateXXXKey（）、verifyXXXKey（）、getState（），对应图 2.1 中 Key；（3）数据服务类 DataService：管理多个 FileService

的类，它不创建文件对象，需要使用前面的三种类的对象，主要方法包括：selectFile（）、readFile（）、updateFile（）。

Applet 的结构如图 2.2 示，在这个设计中，Applet 是一个集合（Collect），负责创建有关对象，DataService 是一个容器（Container）。

其中，KeyService 是三个类的总称，因为各个应用的读写控制密钥的功能不完全一样，我们首先设计了一个基本的 KeyService0，它适用于应用 Applet01，这是多数应用的模式。一些应用（如 Applet02）有多个读写控制密钥，需要扩展 KeyService0，得到 KeyService2，KeyService2 重载了 KeyService 的部分方法。一些应用（如 Applet00）的读写控制密钥的功能与 KeyService0 不同，需要 KeyService0 作为服务类，并重新定义部分方法，得到 KeyService1。如图 2.3 示。

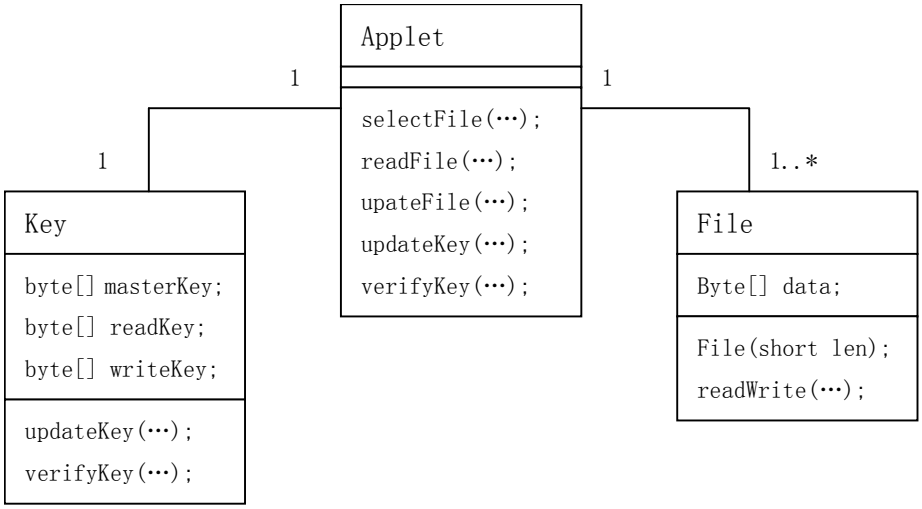

图 2.1　对象模型

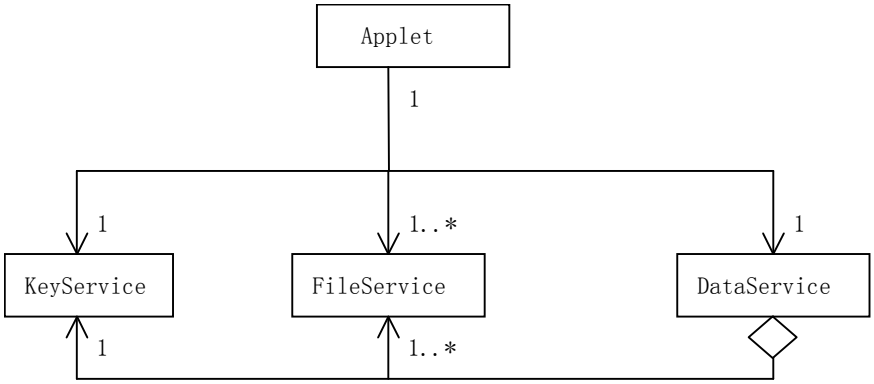

图 2.2 Applet 结构

图 2.3 三个 KeyService 之间的关系

图 2.4 以 Applet 的 process（）处理“读文件”指令的序列图说明了各个对象之间的相互作用。

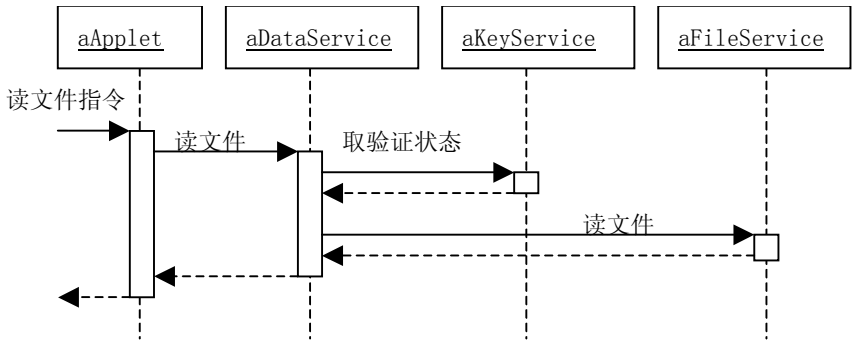

图 2.4　Applet 读文件

每个 Applet 的结构相同，它们都共享 FileService、KeyService 和 DataService，这 3 个被共享的类组成一个服务包，各个 Applet 各自组成一个包，如图 2.5 所示。

applets 程序组织：

app　应用包

Applet00 第 1 个应用的包

Applet00.java 第 1 个应用

Applet01 第 2 个应用的包

Applet01.java 第 2 个应用

Applet02 第 3 个应用的包

Applet02.java 第 2 个应用

services 服务包

CommonService.java　常数定义、读 APDU 缓冲全部数据。

DataService.java 管理多个文件：选择、读、写。

FileService.java 单个文件读写。

KeyService.java 三个密钥（主控、读、写）的管理：更新与验证。

KeyService1.java 三个密钥（主控、验卡、密码重装）的管理：更新与验证，用于 Applet00。

KeyService2.java 四个密钥（主控、读、写一、写二）的管理：更新与验证，用于 Applet02。

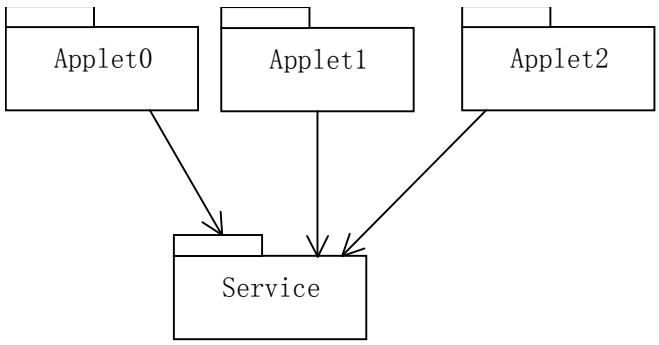

图 2.5　包之间关系

2.5 编程实现

由于 Java 卡上的计算和存储资源有限，Java 卡 Applet 的编程与 Java 程序的编程不大一样，并且需要采取一些优化措施。根据项目的实践，我们总结出以下一些 Java 卡 Applet 的编程要点。

2.5.1 错误检测

Java 卡 Applet 中，相当大一部分代码用于检测非法的或者错误格式的命令上。一个没有检测到的错误可能将卡锁住，或者导致重要数据丢失。一般，Applet 在执行 APDU 命令所要求的任务之前，要按照该 Applet 的需求，确认该命令：APDU 命令被该 Applet 支持；APDU 命令格式正确；APDU 命令满足安全性或其它内部条件。Applet 还要检测任务是否被正确执行。如果检测到错误，Applet 在正常情况下停止处理，产生一个 ISOException 异常，如果 Applet 不处理这个异常，则 JCRE 会捕获到，并提取状态字返回给主机。

2.5.2 内存使用

在程序中，应尽量使用 APDU 缓冲来保存中间结果和进行参数传递，APDU 缓冲是 RAM，速度快，大小约 200 多字节，与具体的 Java 卡相关。

Java 卡的 RAM 堆栈大小只有 100 多字节，因此，方法调用嵌套不能太深。不要使用递归调用。

2.5.3 内存泄露

在编程时，应该认为 Java 卡没有垃圾回收机制，一个 Applet 需要的对象或空间必须在其构造器中分配完毕，Applet 构造器可以调用其它类的构造器，任何类只能在构造器中分配空间（Applet 在调用 register（）以后不能再分配空间）。这样，如果由于内存或其它原因，安装失败，分配给 Applet 的空间可以回收。

由于在卡的使用过程中，用户可能在中途将卡拔出，deselect（）可能不被调用，因此不要假设 deselect（）中的清除是完整的。

为了减少存储量，可以采用对象共享和复用。对象的共享即共享接口对象机制。对象复用的规则是通过向一个对象实例的成员变量写入新值，即采用单件（Singleton）设计模式，被共享的对象只创建一次，每次需要使用它时，就将其成员变量（一般是 static）用新值覆盖。对于只有 static final 基本数据类型成员和 static 方法的共享类，不必实例化。

在对象共享和复用中，被共享和复用的 Applet 或对象，与共享或复用它们的 Applet 或对象一般在不同的 CAP（Converted APplet)文件中，具有不同的上下文，参数只能是基本类型，能作为参数的数组只能是 APDU 缓冲，因此，需要数组作为参数时，应将数组复制的 APDU 缓冲中，从共享接口或对象返回后，再从 APDU 缓冲复制到需要的位置。如果在多个 Applet 之间复用要实例化的单件模式对象，删除一个 Applet，可能会影响另一个 Applet，因此，只在一个 Applet 内使用这种方式。

2.5.4 性能

类层次问题；在设计应用的类层次结构时，需要考虑卡上有限的资源。类的继承层次能增加 Applet 的灵活性，但也会增大存储开销；如果采用紧凑结构，减少存储开销，但会限制代码共享、调试和升级。因此，必须平衡需求和设计。建议每个 Applet 的类不超过 10 个，类继承不超过 3 层。

临时数组：采用临时（transcient）数组来保存中间结果或频繁更新的数据。Java 卡的存储器有 3 类：EEPROM（相当于磁盘）、ROM 和 RAM（用于存储方法的参数、方法中定义的局部变量、方法返回值等）。Applet 的成员变量以及用 new 分配的对象存储在 EEPROM 上，具有持久性，对它们的更新也保证了原子性，但是写入时间比较长。为提高速度，可以使用临时数组，临时数组的数据存储在 RAM 上，没有持久性（掉电后，分配的空间仍存在，但数据内容不确定），对它们的更新也没有原子性。临时数组用于存储中间结果，并且，可以使用事务机制来保证更新临时数据的原子性。由于 Java 卡的 RAM 有限（一般只有 200 多字节），分配后也不会回收，需要权衡，小心使用。

使用本地方法（如 makeShort（）等），提高计算速度。

加密解密：除非特别需要，卡上的数据不要使用复杂的加密和解密方法（如 RSA），以免占用过多的时间。

第 3 章　密钥应用的开发

3.1　密钥应用概述

在我们的密钥系统中，需要生成一个系统根密钥，每个 Java 卡应用有 3 个分散参数，分散参数与系统根密钥混合，形成 Java 卡应用的 3 个根密钥。每个 Java 卡应用根密钥与用户卡号混合，形成每个 Java 卡应用的主控和读写密钥。制卡应用也是一个特殊的应用，它的分散参数是 Java 卡的序列号，它只需要制卡根密钥，自行形成自己的密钥。

我们的第 1 个密钥应用是系统根密钥应用 MakterApp，它产生一个 16 字节随机数，作为系统根密钥，以后不再改变。它可以接收输入的 Java 卡应用的分散参数，分散参数以后也不再改变。我们可以 MasterApp 从可以导出 Java 卡应用的根密钥，这需要一个传输密钥。MasterApp 还要产生制卡应用的根密钥。

第 2 个密钥应用是应用根密钥应用 IssueApp。它从系统根密钥应用得到所有 Java 卡应用的根密钥，这些根密钥不再改变，这需要一个传输密钥。它接收输入的用户卡号，计算用户卡上 Java 卡应用的密钥。可以从 IssueApp 导出用户应用的密钥，这也需要一个传输密钥。应用根密钥应用也需要保持制卡应用的根密钥。

第 3 个密钥应用是机具应用 DeviceApp，它从系统根密钥应用得到某些应用的根密钥，这些根密钥不再改变，这需要一个传输密钥。它接收输入的用户卡号，计算用户应用的密钥。根据计算的用户密钥，与用户应用交互。

采用密码来保护各个密钥应用的使用，累计密码错试次数超过移动数目后，密钥应用被关闭。

为简单起见，传输密钥固定在程序中，以后不变（也可以在制卡时写入，以后通过密码验证后，再更改）。

发行管理应用 BatchApp 存在于系统根密钥应用卡上，或存在于应用根密钥应用卡上。系统根密钥应用和应用根密钥应用需要提供一个接口，以便向发行管理应用提供卡管理器的根密钥。

3.2　数据的加密和MAC的计算

在密钥应用中，密钥通常作为数据传输，为了保证数据的隐秘性，必须根据加密密钥进行加密，有时为了保证数据的完整性，还需要根据签名密钥计算命令的报文认证码（Message Authentication Code, MAC）。

Java 卡中，最低强度的密钥方法是数据加密标准(DES)。数据加密标准是对称密码学算法，加密和解密数据需要使用相同的密钥。最简单的形式是使用 8 字节密钥加密 8 字节数据块，能够使用相同的 8 字节密钥来解密，得到原始的明文。在 Java 卡应用中，所有加密和解密算法都需要使用双长度密钥的 3 级 DES（DES3ENC 或 DES3DEC）的安全性。图 3.1 定义了 3 级 DES 加密需要的方法。3 级 DES 解密是这个运算的逆。

当然随着 Java 卡技术的进步，可以采用强度更高的非对称的公开密钥技术来进行数据加密和计算 MAC。

Java 卡的加密方式主要有密码机块链接和电子码本两类。

3.2.1 密码机块链接

密码机块链接（Cypher Block Chaining，CBC）是一种通过加密机制链接多个8字节块的方法。

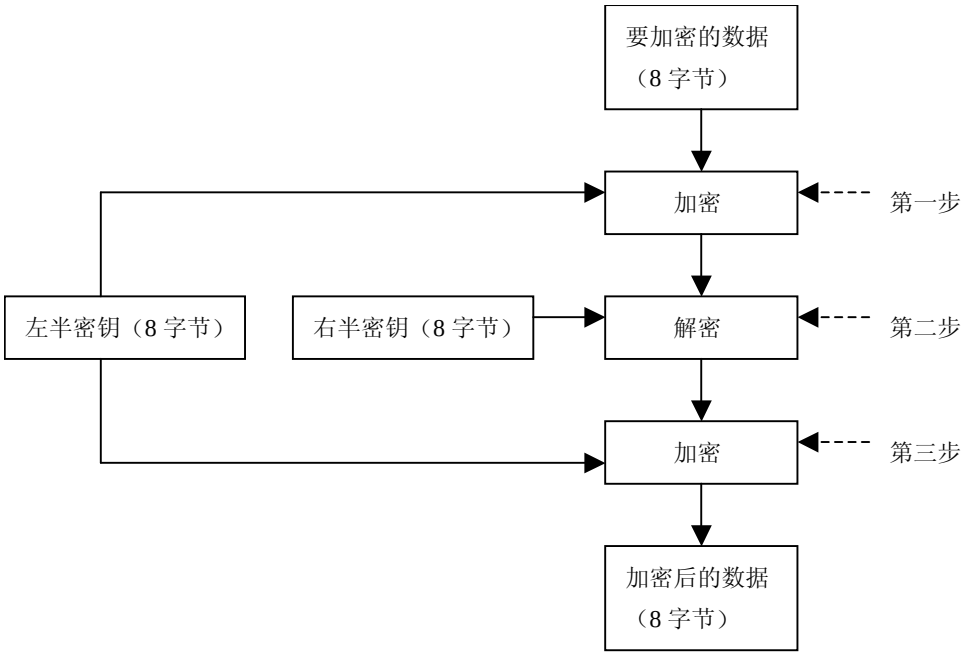

图 3.1　3 级 DES 加密（DES3ENC）

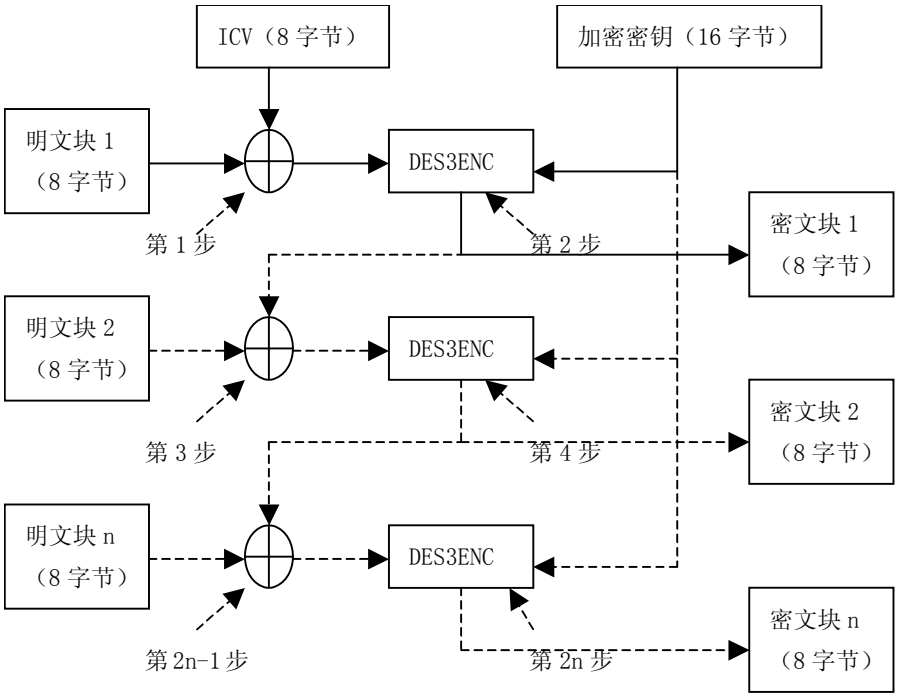

图 3.2　链接式数据加密

（1）链接式数据加密

加密数据（命令消息的数据字段）时，每个8字节块的加密结果成为整个加密结果的一部分，并用于下一次加密运算的输入。在这种加密模式中，初始链接向量（Initial Chaining Vector，ICV）总是8字节的二进制0。数据解密则是使用解密（DES3DEC）运算来代替每个加密（DES3ENC）运算的逆运算。如图3.2。

(2) 链接式签名/MAC计算（全3级DES）

链接式MAC计算时，每次加密结果用作下次加密的输入，仅保留最后的结果。最后的结果是数据的签名（MAC或认证密文），签名通常与数据一起传输。初始链接向量ICV取决于完成的签名运算。签名的验证需要相同的运算，以及一个比较步骤。如图3.3。

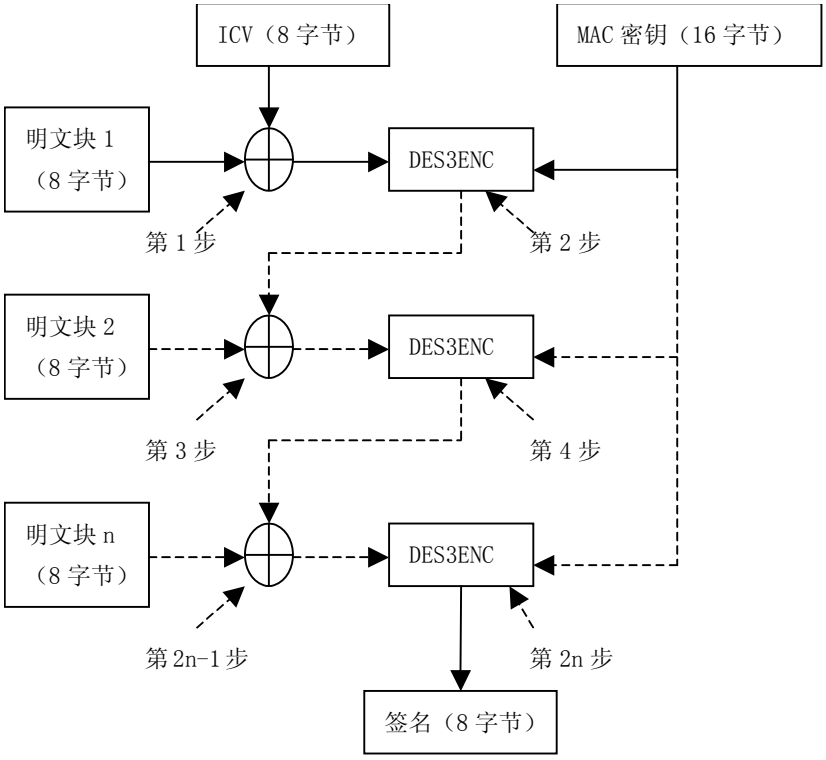

图 3.3 链接式 MAC 计算方法（全 3 级

3.2.2 电子码本加密

电子码本（Electronic Code Book，ECB）加密是一种加密一个或多个8字节块数据（通常是密钥）的方法。加密时，每个8字节块加密的结果成为整个加密结果的一部分。数据的解密是精确的逆运算，用解密运算DES3DEC代替加密运算DES3ENC。如图3.4。

3.2.3 计算和验证APDU命令的MAC

APDU命令的MAC应用于整个要传输到卡上的APDU命令，包括命令消息的头部和数据字段（不包括Le）。MAC的产生和验证需要使用MAC会晤密钥、初始链接向量以及全3级DES签名方法。

在计算MAC前，需要对APDU命令进行填充，并修改命令头部。

APDU命令头部的修改规则为：（1）命令消息长度Lc必须增加8字节，以包括命令消息中数据字段里的MAC。（2）必须修改类别字节，以指出这个APDU包括安全消息传递，这需要将命令类别字节的b3设置为1（命令类别字节CLA的8位分别为：b8、b7、b6、b5、b4、b3、b2、b1，b7是最高位，b1是最低位）。例如，0x00和0x80命令类型被分别修改为0x04和0x84。

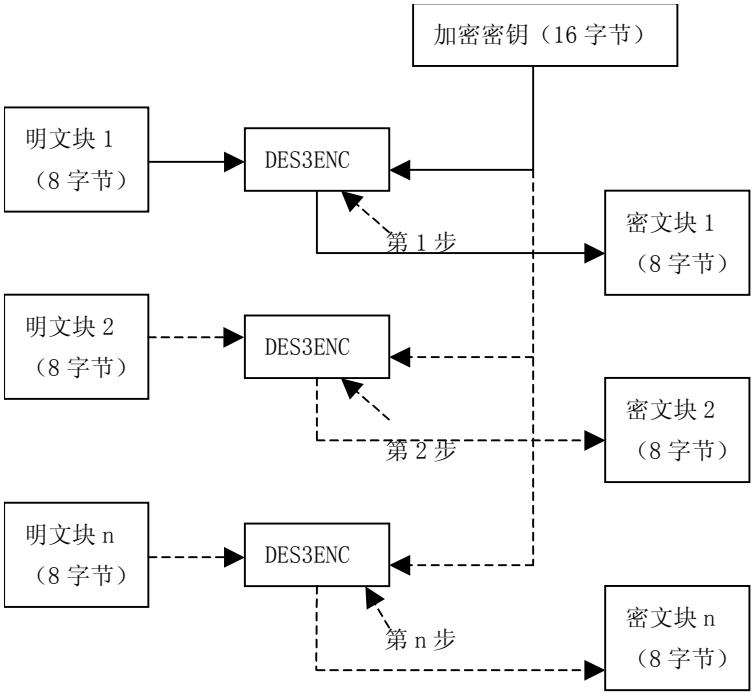

图 3.4　电子码本数据加密

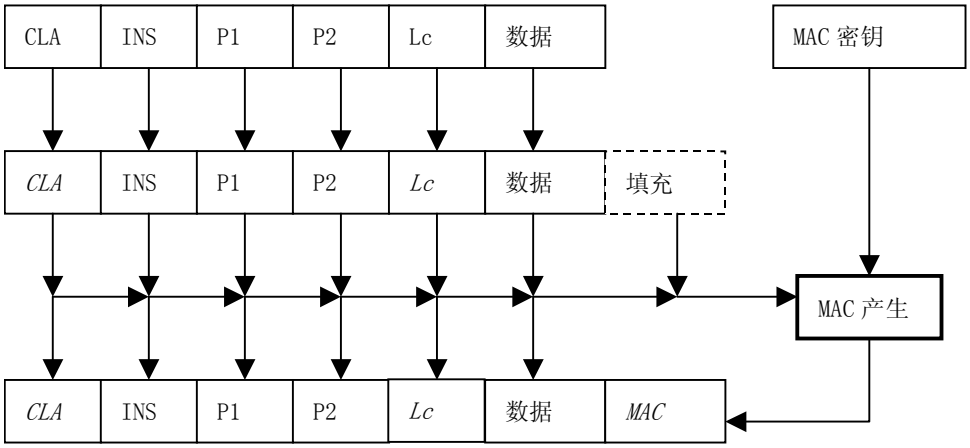

图 3.5　MAC 的计算

计算MAC时的填充规则要应用于包含命令消息中APDU命令头和数据字段的数据块，规则是：（1）数据块右边附加字节0x80。（2）如果步骤（1）得到的结果数据块长度是8的倍数，就不需要进一步填充。（3）否则，在数据块右边附加一个或多个字节0x00，直到数据块长度为8的倍数为止。

在计算命令序列中各个命令的MAC时，使用初始链接向量ICV来链接各个命令，以保证命令序列完整性。计算命令序列中第一条命令的MAC时，ICV的值为8字节的0x00，计算命令序

列中其它命令的MAC时，ICV的值是上一条命令的MAC，即使上一条命令执行失败，也是这样。

MAC的计算如图3.5。

卡上应用为了验证MAC，必须对数据完成相同的填充机制，并使用与主机（卡外实体）相同的ICV和MAC会晤密钥，来验证MAC。卡上应用必须保留验证后的MAC，用作随后MAC验证的ICV（无论接收到的APDU是否成功执行，都必须这样。即，不能丢弃验证后的MAC，必须保留刚验证过的MAC值）。

3.2.4 APDU数据字段的加密和解密

如果需要保证数据的隐秘性，主机可以将需要传输到卡上的命令消息中的明文数据字段加密，数据加密不包括命令消息的头部和MAC，只包括数据字段中的数据。命令消息中数据的加密需要使用加密会晤密钥和链式数据加密方法。

加密数据前，必须填充数据，与计算MAC不同，填充的字节成为数据字段的一部分，这就要求进一步修改命令中的Lc值。

数据加密的填充规则为：（1）原始明文数据字段的长度被附加在左边，成为命令数据的一部分。（2）如果步骤（1）得到的数据字段的长度为8的倍数，就不需要进一步填充。（3）否则，在数据字段右边附加一个字节0x80。（4）如果步骤（3）得到的数据字段的长度为8的倍数，就不需要进一步填充。（5）否则，在数据字段右边增加一个或多个字节0x00，直到数据字段长度为8的倍数为止。

必须将为完成以上填充而附加到数据字段的字节数目增加到Lc，附加的字节包括强制的长度字节、可选的0x80和任何可选的0x00。

填充完成后，就可以使用加密密钥对填充后的数据字段加密，加密按8字节块逐块进行，每次加密的结果成为命令消息中加密后数据的一部分。数据的加密如图3.6。

同时需要计算MAC和数据加密时，应该先计算MAC，再进行数据加密，

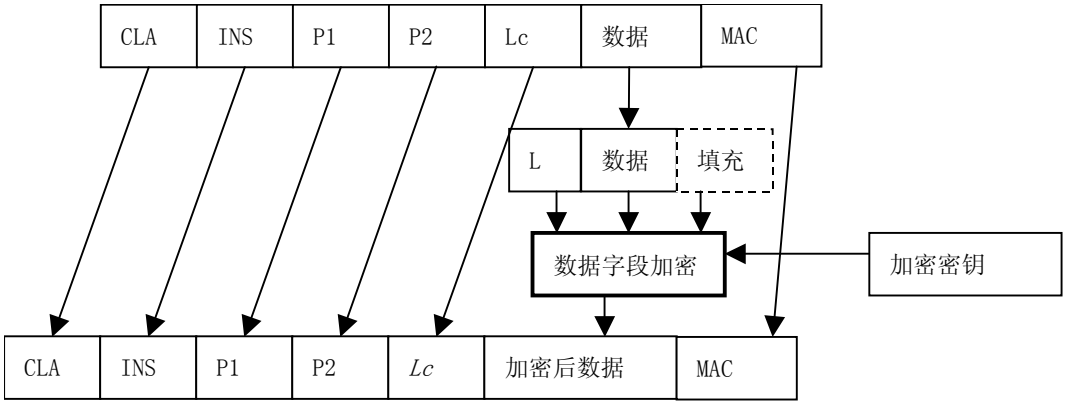

图 3.6　数据的加密

数据加密中，初始链接向量ICV和链接向量仅用于链接目前被加密的数据的各块，因此，数据加密的ICV总是8字节的0x00。

卡上应用在接收到经过数据加密的命令后，首先要解密命令消息，解密使用8字节的0x00作为ICV，并使用与主机（卡外实体）相同的加密会晤密钥。在解密后，必须先删除填充，并修改Lc以反映数据的真实长度，真实长度是原始明文数据长度加上可能有的MAC的长度。在数据解密后，如果需要，应该验证MAC。

3.3 密钥应用的AID和接口

3.3.1 应用的AID

系统根密钥应用 MasterApp 的 AID=0x4B:0x45:0x59:0x4D:0x43，即字符串“KEYMC ”，它的第 1 个实例的 AID 为：0x4B:0x45:0x59:0x4D:0x43:0x01。

应用根密钥应用 IssueApp 的 AID=0x4B:0x45:0x59:0x49:0x43，即字符串“KEYIC ”，它的第 1 个实例的 AID 为：0x4B:0x45:0x59:0x49:0x43:0x01。

机具应用 DeviceApp 的 AID=0x4B:0x45:0x59:0x44:0x43，即字符串“KEYDC ”，它的第 1 个实例的 AID 为：0x4B:0x45:0x59:0x44:0x43:0x01。

3.3.2 系统根密钥应用的指令集

（1）选择应用

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x04	0x00	0x06	0x4B:0x45:0x59:0x4D:0x43:0x01	无

响应 APDU

数据域	状态字	状态字含义
	0x9000	成功
	0x6A82	应用不存在

（2）验证密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x20	0x00	0x00	0x06	密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（3）修改密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0x5E	0x00	0x00	0x0C	旧密码新密码	无

响应 APDU

数据域	状态字	状态字含义

无	0x9000	成功 SW_NO_ERROR
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（5）更新传输密钥数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xF1	0x00	0x00	0x18	密钥密文数据 1+16+7	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6701	需要 PIN 验证
	0x6702	修改传输密钥失败
	0x670B	没有传输密钥

（6）导入分散参数

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF0	应用编码	密钥编码	0x10	分散参数	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6701	需要 PIN 验证
	0x670A	分散参数已经存在
	0x6703	密钥类型错误

（7）导出应用根密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF3	应用编码	密钥编码	0x00	无	0x18

响应 APDU

数据域	状态字	状态字含义
导出密钥	0x9000	成功
	0x6701	需要 PIN 验证
	0x6709	没有分散参数
	0x6703	密钥类型错误
	0x670B	没有传输密钥
	0x6707	没有系统根密钥

3.3.3 应用根密钥应用的指令集

（1）选择应用

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x04	0x00	0x06	0x4B:0x45:0x59:0x49:0x43:0x01	无

响应 APDU

数据域	状态字	状态字含义
	0x9000	成功
	0x6A82	应用不存在

（2）验证密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x20	0x00	0x00	0x06	密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（3）修改密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0x5E	0x00	0x00	0x0C	旧密码新密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功 SW_NO_ERROR
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（5）更新传输密钥数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xF1	0x00	0x00	0x18	密钥密文数据 1+16+7	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功

	0x6721	需要 PIN 验证
	0x6722	修改传输密钥失败
	0x672A	没有传输密钥

（6）导入应用根密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xF2	应用编码	密钥编码	0x18	密钥	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6721	需要 PIN 验证
	0x672A	没有传输密钥
	0x6723	密钥类型错误
	0x6724	导入密钥失败

（7）导入分散参数

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF3	0x00	0x00	0x10	分散参数	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6721	需要 PIN 验证
	0x6725	还没有导入应用根密钥

（8）导出应用密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF4	应用编码	密钥编码	0x00	无	0x18

响应 APDU

数据域	状态字	状态字含义
导出密钥	0x9000	成功
	0x6721	需要 PIN 验证
	0x6726	没有分散参数
	0x6723	密钥类型错误
	0x672A	没有传输密钥
	0x6725	还没有导入应用根密钥

3.3.4 机具应用的指令集

（1）选择应用

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0xA4	0x04	0x00	0x06	0x4B:0x45:0x59:0x44:0x43:0x01	无

响应 APDU

数据域	状态字	状态字含义
	0x9000	成功
	0x6A82	应用不存在

（2）验证密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x20	0x00	0x00	0x06	密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（3）修改密码

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0x5E	0x00	0x00	0x0C	旧密码新密码	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功 SW_NO_ERROR
	0x68xx	密码验证失败，剩余 xx 次

注：剩余 0 次后，应用被锁住。

（5）更新传输密钥数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xF1	0x00	0x00	0x18	密钥密文数据 1+16+7	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功

	0x6741	需要 PIN 验证
	0x6742	修改传输密钥失败
	0x674D	没有传输密钥

（6）导入应用根密钥

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x84	0xF2	应用编码	密钥编码	0x18	密钥	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6741	需要 PIN 验证
	0x674D	没有传输密钥
	0x6743	密钥类型错误
	0x6745	导入密钥失败

（7）设置用户卡号（导入分散参数）

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF7	0x00	0x00	0x10	分散参数	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6741	需要 PIN 验证

（8）取随机数

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x84	0x00	0x00	0x00	无	0x08

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6700	长度错误
	0x6741	需要 PIN 验证

（8）验证用户卡

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x00	0x82	0x01	0x00	0x08	数据	无

响应 APDU

数据域	状态字	状态字含义
无	0x9000	成功
	0x6700	长度错误
	0x6741	需要 PIN 验证
	0x674B	无分散参数
	0x6744	无随机数
	0x6746	无验卡根密钥
	0x674C	用户卡无效

（9）加密数据

命令 APDU

CLA	INS	P1	P2	Lc	数据域	Le
0x80	0xF9	应用编码	密钥编码 0	0x08	数据	0x08

响应 APDU

数据域	状态字	状态字含义
加密的数据	0x9000	成功
	0x6700	长度错误
	0x6741	需要 PIN 验证
	0x674B	无分散参数
	0x6746	无应用根密钥
	0x674C	用户卡无效

3.4 密钥应用设计

密钥应用的结构比较简单。我们将各个密钥应用的公共部分独立出来，形成一个公共服务类 CommonService，用于数据加密解密和 MAC 的计算。

由于 MasterKey 和 IssueCard 应用需要向制卡应用提供制卡应用的根密钥，我们采用接口服务类设计模式，来实现这种功能。MasterKey 和 IssueCard 应用作为服务类，制卡应用作为客户类。为此，采用 Java 卡的共享接口对象（Shareable Interface Object, SIO）机制来实现，它的对象结构如图 3.7 所示。首先，定义一个服务接口 BatchKeyInterface，作为服务类的服务接口，客户 Applet 根据服务类 Applet 的应用标识 AID 向 JCRE 请求服务类 Applet，然后调用服务类 Applet 的方法，访问服务类提供的接口。

applets 程序组织：

key　应用包

MasterCard 系统根密钥应用的包

MasterApp 系统根密钥应用

IssueCard 应用根密钥应用的包

IssueApp 应用根密钥应用

DeviceCard 机具应用的包

DeviceApp 机具应用

BatchCard 发行管理应用的包

BatchApp 发行管理应用

Interfaces 接口包

BatchKeyInterfaces 接口

Services 公共服务包

CommonService 公共服务类

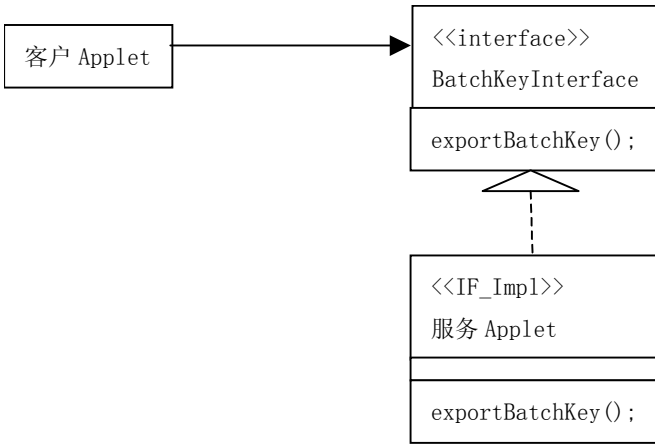

图 3.7　共享接口对象

各个应用的密钥编码采用 2 个字节。第 1 个字节为应用编码。Java 卡应用 Applet00、Applet01、Applet02 的编码分别为 0x00、0x01、0x02。第 2 个字节为应用密钥编码，分散前，主控密钥（卡管理器根密钥）为 0x01，读密钥（验卡密钥）为 0x02，写密钥为 0x03，写密钥二为 0x04。分散后，主控密钥（卡管理器根密钥）为 0x11，读密钥（验卡密钥）为 0x12，写密钥为 0x13，写密钥二为 0x14。这样，0x0102 就表示 Applet01 的读密钥。

第 4 章　卡管理器

4.1　卡管理器概述

卡管理器是Java卡上的一个特殊的应用，它负责维护Java卡及其内容的总体安全和管理，由于卡管理器在整个卡中起监督员作用，它的生命周期可认为是卡的生命周期，卡管理器生命周期的结束与卡生命周期的结束等价。

卡管理器在Java卡的注册表中保存和维护卡生命周期状态信息，并管理需要的状态转换，以响应APDU命令。

卡管理器有多组密钥，每组密钥有3个密钥，分别称为认证（加密）密钥（Authentication Key）、报文认证码（Message Athentication Code, MAC）计算密钥（MAC Key）和密钥加密密钥（Key Encryption Key, KEK）。在任何时候，可以让卡管理器选择一组密钥，用于与主机的交互。

卡管理器称为发卡商安全域，发卡商知道卡管理器使用的当前密钥组中的密钥，因而可以通过卡管理器的认证，并与卡管理器交互。发卡商可以通过卡管理器来创建其它安全域，可以将安全域想象为卡管理器对象类的一个实例，安全域也有自己的密钥组。对于具有某种权限安全域，主机在通过认证后，可以通过这个安全域来修改安全域中的内容（如Applet包和Applet实例）。

主机（外部实体）与卡管理器交互时，首先需要通过认证，通过认证后，主机和卡管理器之间就建立了一条安全通道，安全通道的级别在认证过程中指定。安全通道的级别有：（1）一般通道（CLEAR）。主机发送给卡管理器的命令APDU不用计算MAC，数据段也没有加密。（2）MAC通道。主机发送给卡管理器的命令APDU必须用MAC Key计算MAC，但数据段不用加密。（3）MAC加密通道。主机发送给卡管理器的命令APDU必须用MAC Key计算MAC，数据段也必须用加密密钥进行加密。

这3类通道的安全强度为：MAC加密通道 > MAC通道 > 一般通道。安全强度越高，计算时间就越长。可以根据实际需要，选择适当的通道类型。

卡管理器的行为由OpenPlatform/GlobalPlatform规范规定。

4.1.1　卡管理周期的状态

卡管理器在其生命周期中，可以处于某个状态。卡生命周期状态包括：（1）OP_READY。（2）INITIALIZED。（3）SECURED。（4）CM_LOCKED。（5）TERMINATED。

我们可以在卡生命周期的任何点上终止卡。生命周期状态OP_READY和INITIALIZED用于卡生命周期的发行前阶段，生命周期状态SECURED、CM_LOCKED和TERMINATED用于卡生命周期的发行后阶段。

4.1.1.1　OP_READY

在不可逆的卡生命周期状态OP_READY，运行时间环境的所有基本功能都是可用的，卡管理器作为被选择的应用，已经就绪，可以接收、执行和响应APDU。

卡在OP_READY状态具有的功能包括：（1）包括在不可变持久存储器中的可执行下载文

件是可用的。（2）运行时间环境为执行准备就绪。（3）卡管理器作为“被选择的应用”行动。（4）卡管理器中有一个初始化密钥可以使用。

在这个生命周期状态，可以下载安全域及其密钥集合，以保证应用提供商的密钥与发卡商的密钥分隔。此外，可以将应用下载到卡的存储器中，并安装任何应用。如果在这个阶段还有可用的个人化信息，就可以将这些信息写入应用，将应用个人化。

4.1.1.2 INITIALIZED

不可逆的卡生命周期状态INITIALIZED是一个管理性的卡状态，它的严格定义取决于制造商、发行商和/或实现。这个状态通常表示：卡上已经存在一些初始数据（例如，密钥和卡管理器数据），但卡还没有准备就绪、发行到卡用户手中。

发卡商在认证后，才能启动从OP_READY到INITIALIZED的迁移。

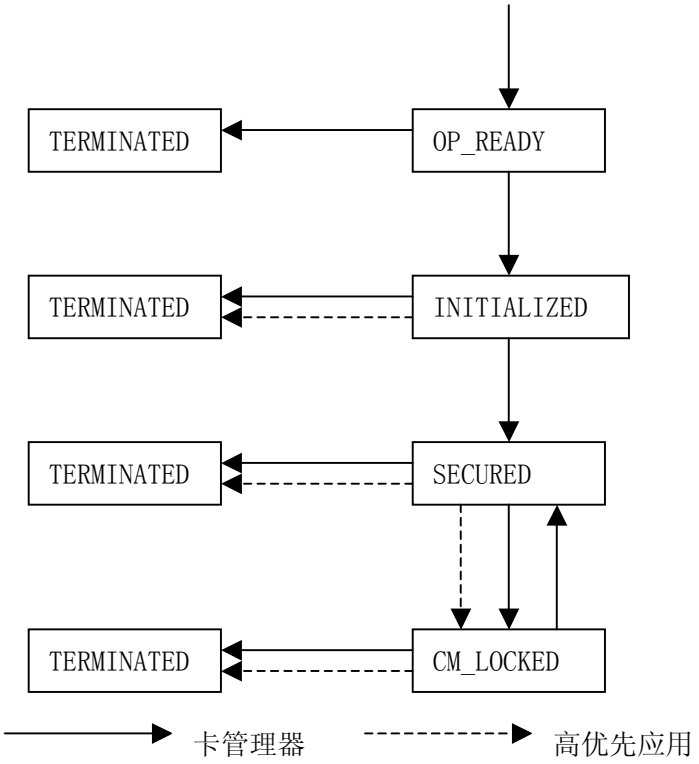

图 4.1 卡管理器生命周期状态迁移

4.1.1.3 SECURED

不可逆的卡管理器生命周期状态SECURED是发行期间常规的、用于运行的卡生命周期状态，这个状态是实施发卡商的行为和卡发行后行为（诸如应用下载、安装和激活）有关的安全策略的卡管理器的指示符。

在SECURED状态，卡具有如下功能：（1）卡管理器具有发挥其全部功能所必需的密钥集合和安全元素。（2）可以通过卡管理器来执行对发卡商初始化的卡内容的改变。（3）可以经过卡管理器来完成对属于发卡商的应用在发行后的个人化。（4）激活的安全域具有发挥其全部功能所必需的密钥集合和安全元素。（5）可以通过具有委派管理优先权的安全域来

执行对应用提供商初始化的卡内容的改变。（6）可以经过安全域来完成对属于应用提供商的应用在发行后的个人化。

发卡商在认证后，才能启动从INITIALIZED或CM_LOCKED到SECURED的迁移。

4.1.1.4 CM_LOCKED

可逆状态CM_LOCKED用于通知卡管理器暂时关闭卡上除卡管理器以外的所有应用，这个状态为发卡商提供了标识内部或外部安全威胁，同时将卡上的应用功能关闭的能力。

将卡管理器设置为这个状态时，表示除了由发卡商控制的卡管理器外，不再有其它功能有效。卡管理器本身、卡上具有相关优先权的应用、或一个经过认证的发卡商可以启动从SECURED到CM_LOCKED的迁移。

通过与卡通信，发卡商在决定威胁不再存在或严重性有限时，可以将卡复位到以前的运行状态。

4.1.1.5 TERMINATED

卡管理器被设置为生命周期状态TERMINATED时，将永久关闭包括卡管理器本身的所有应用的功能。创建这个状态是为具有优先权的实体提供一个机制，由于检测到一个严重威胁或卡要作废等原因，来从逻辑上“销毁”一张卡。

卡管理器状态TERMINATED是不可逆的，标志着卡生命周期和卡的结束（卡将不会对任何通信作出响应）。

卡管理器本身、卡上具有相关优先权的应用、或一个通过认证的发卡商可以启动从任何以前状态到TERMINATED的迁移。

4.1.2 卡管理器生命周期状态迁移

图4.1给出了卡管理器的生命周期迁移图，这可以认为是一个具有逆向状态迁移或跳过状态迁移概率的顺序过程。

4.2 卡管理器的外部接口

卡管理器的外部接口就是它能接收的主机或外部实体发送给它的命令APDU。下表给出了卡管理器能接收的命令的概要。

CLA	INS	命令
80/84	E4	DELETE
80/84	CA	GET DATA
80/84	F2	GET STATUS
80/84	E6	INSTALL
80/84	E8	LOAD
80/84	D8	PUT KEY

00	A4	SELECT
80/84	F0	SET STATUS

4.2.1 信息编码

可执行下载文件生命周期状态用1字节编码，如下表所示。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0	0	0	0	0	LOGICALLY_DELETED
0	0	0	0	0	0	0	1	LOADED

Applet应用生命周期状态用1字节编码，如下表所示。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0	0	0	0	0	LOGICALLY_DELETED
0	0	0	0	0	0	1	1	INSTALLED
0	0	0	0	0	1	1	1	SELECTABLE
0	0	0	0	1	1	1	1	PERSONLIZED
0	1	1	1	1	1	1	1	BLOCKED
1	1	1	1	1	1	1	1	LOCKED

卡管理器生命周期状态用1字节编码，如下表所示。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0	0	0	0	1	OP_READY
0	0	0	0	0	1	1	1	INITIALIZED
0	0	0	0	1	1	1	1	SECURED
0	1	1	1	1	1	1	1	CM_LOCKED
1	1	1	1	1	1	1	1	TERMINATED

应用的优先权用1字节编码，如下表所示。

B8	B7	B6	B5	B4	B3	B2	B1	含义
1								安全域
	1							数据认证模式DAP验证
		1						委派管理
			1					卡管理器封锁优先权
				1				卡终止优先权
					1			缺省被选择
						1		改变PIN优先权
							1	强制DAP验证

一个应用可以支持一个或多个应用优先权，各位的含义如下：（1）b8=1表示应用是一个没有委派管理的安全域（与b6互斥）。（2）b7=1表示应用是一个具有数据认证模式DAP验证能力的安全域（与b1互斥）。（3）b6=1表示应用是一个具有委派管理优先权的安全域（与b8互斥）。（4）b5=1表示应用具有卡管理器封锁优先权。（5）b4=1表示应用具有卡终止优先权。（6）b3=1表示应用具有缺省选择优先权。（7）b2=1表示应用具有改变PIN优先

权。（8）b1=1表示应用是一个具有强制数据认证模式DAP验证能力的安全域（与b7互斥）。

下表描述的任何命令都返回的错误条件。

SW1	SW2	含义
64	00	没有特定诊断
67	00	Lc长度错误
69	82	安全条件不满足
69	85	使用条件不满足
6A	86	P1或P2错误
6D	00	指令无效
6E	00	指令类别无效

所有卡管理器接收的命令的指令类别字节遵从ISO 7816-4标准，并根据下表编码。

类别	含义
00	ISO类型命令
80	专用命令
84	具有安全消息传递的专用命令

对象（如Applet包和Applet实例）的编码采用TLV表示，TLV表示：标签Tag、长度Length和值Value，这是一种表示对象的方式，标签指示对象的类型，长度指示对象值的字节数目，值指示对象值的字节串。

4.2.2 命令详解

4.2.2.1 DELETE命令

DELETE命令用于删除一个唯一标识的对象，对象可以是可执行下载文件、应用或其它项目，但是，为了删除一个对象，这个对象必须由被选择的应用唯一标识。

DELETE命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	E4	DELETE
P1	xx	参见参考控制参数P1
P2	00	必须为00
Lc	xx	数据字段长度
Data	xxxx	TLV编码的对象标识符
Le	00	

参考控制参数P1用于指示随后是附加的DELETE命令，或最后一条DELETE命令已经到达。

B8	B7	B6	B5	B4	B3	B2	B1	含义

0								最后一条DELETE命令
1								随后还有DELETE命令
	0	0	0	0	0	0	0	保留未来使用

参考控制参数P2总是为00。

数据字段应该包含要删除的对象的TLV编码名字。删除可执行下载文件或应用应该使用ISO应用标识符标签“4F”来指明。下表给出了DELETE命令的一般结构。

出现	长度（字节数目）	名字
强制	1—2	标签
强制	1	对象标识符长度
强制	1—n	对象标识符

在卡管理器处理DELETE时，将返回一个字节的00，指示没有附加的响应数据。

命令的成功执行用“90”“00”编码，IC卡可能返回下列警告条件。

SW1	SW2	含义
62	00	应用已经逻辑删除

IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
65	81	存储器失败
69	85	引用的的数据不能删除
6A	88	没找到应用的数据
6A	82	没找到应用
6A	80	命令数据中值不正确

4.2.2.2 GET DATA命令

GET DATA命令用于查询单个数据对象，P1和P2编码用于定义特定的数据对象.

GET DATA命令消息根据下表编码:

代码	值	含义
CLA	80/84	Open Platform命令
INS	CA	GET DATA
P1	xx	00或标签值高字节
P2	xx	标签值低字节
Lc	00	
Data	无	
Le	00	

参数P1和P2定义要读取的数据对象的标签.

命令消息的数据字段是空的。

响应消息的数据字段包含指向命令消息的P1和P2参数的TLV编码的数据对象。

命令的成功执行用“90”“00”编码。
IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
6A	88	没找到应用的数据

4.2.2.3 GET STATUS命令

GET STATUS命令用于根据给定的匹配/搜索准则，查询卡管理器、可执行下载文件和应用相关的生命周期状态信息。GET STATUS是SET STATUS的互补命令。

GET STATUS命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	F2	GET STATUS
P1	xx	参考控制参数P1
P2	xx	参考控制参数P2
Lc	xx	数据字段长度
Data	xxxx	搜索准则
Le	00	

参考控制参数P1用于选择将要包括在响应卡元素的某个子集。参考控制参数P1根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
1								指示卡管理器
	1							指示应用
		1						指示可执行下载文件
			0	0	0	0	0	保留未来使用

P1的下列组合是可以接受的：（1）80—仅卡管理器。（2）40—仅应用。（3）20—仅可执行下载文件。（4）E0—卡管理器、应用和可执行下载文件。（5）C0—卡管理器和应用。（6）60—应用和可执行下载文件。（7）A0—卡管理器和可执行下载文件。

在这个上下文中，安全域作为应用考虑，通过应用优先权字节与卡上其它应用区分。

参考控制参数P2根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0	0	0	0		保留未来使用
							0	找出第一个或全部对象
							1	找出下一个对象

查询卡管理器的状态时，找出下一个对象没有意义。

命令消息中的数据字段包含TLV编码的搜索条件，可以使用标签“9F70”（应用生命周期状态），必须使用“4F”（应用AID）。生命周期搜索条件是根据生命周期状态字节的位映射。

为搜索所有满足根据参考控制参数P1选择准则的对象，必须在命令消息中出现搜索准则“4F”“00”（即，所有匹配AID条目）。

为搜索可以选择的Visa应用，参考控制参数必须等于“40”，命令消息中必须出现搜索条件 ‘4F’、‘05’、‘A0’、‘00’、‘00’、‘00’、‘03’、‘9F70’、‘01’、‘07’。

根据命令数据字段的搜索准则和参考控制参数的选择准则，响应消息中可能返回多个下列数据结构。

长度	值	含义
1	xx	AID长度
1—n	xxxxx	卡管理器、可执行下载文件或应用的AID
1	xx	卡管理器、可执行下载文件或应用的生命周期状态
1	xx	应用优先权

卡管理器、应用和可执行下载文件生命周期状态、应用优先权根据前面的编码规则进行编码。

命令的成功执行用“90”“00”编码，IC卡可能返回下列警告条件。

SW1	SW2	含义
63	10	还有更多的数据

IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
6A	88	没找到应用的数据
6A	80	命令数据中值不正确

4.2.2.4 INSTALL命令

安装一个应用或安全域需要需用几个不同的卡上功能。INSTALL命令用于指示卡管理器，应该完成应用安装过程中的哪一步的安装。

INSTALL命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	E6	INSTALL
P1	xx	参考控制参数P1
P2	xx	00
Lc	xx	数据字段长度
Data	xxxx	安装数据
Le	00	

INSTALL命令的参考控制参数P1如下编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0					保留未来使用
				1				用于使之可选择
					1			用于安装
						1		用于下载
							0	保留未来使用

各个位的用法如下：（1）b4=1表示要安装或已经安装的应用将要成为可选择的。（2）b3=1表示将要在卡上安装一个应用。（3）b2=1表示将要下载一个下载文件。

可以使用一条INSTALL命令安装一个应用并使之成为可选择的，即将b4和b3都设置为1。

INSTALL命令的安全控制参数P2保留未来使用，编码为“00”。

命令消息的数据字段包含LV编码的数据，LV编码的数据表示中没有分隔符号。

INSTALL[下载]的数据字段：下载文件时使用的INSTALL的APDU（P1的b2设置为1）的数据字段如下表。

出现	长度（字节数目）	名字
强制	1	下载文件AID的长度
强制	5—16	下载文件AID
强制	1	安全域AID的长度
条件的	0—16	安全域AID
强制	1	下载文件数据认证模式DAP的长度
条件的	0—n	下载文件数据认证模式DAP
强制	1	下载参数字段长度
条件的	0—n	下载参数字段
强制	1	下载标记长度
条件的	0—n	下载标记（下载数据认证模式DAP）

下载文件数据认证模式DAP(hash)和下载标记对于委派管理是强制的，如果没有使用委派管理，下载文件数据认证模式DAP和下载标记不应出现。

INSTALL[安装]的数据字段：应用安装时使用的INSTALL的APDU（P1的b3设置为1）的数据字段如下表。

出现	长度（字节数目）	名字
强制	1	下载文件AID的长度
强制	5—16	下载文件AID
强制	1	下载文件中AID的长度
强制	5—16	下载文件中AID
强制	1	应用实例AID的长度
强制	5—16	应用实例AID
强制	1	应用优先权长度
强制	1	应用优先权

强制	1	安装参数长度
强制	2—n	安装参数
强制	1	安装标记长度
条件的	0—n	安装标记（安装数据认证模式DAP）

安装标记对于委派管理是强制的，如果没有使用委派管理，安装标记就不应出现。

Open Platform卡使用实例AID来指示选择被安装的应用时使用的AID。

Open Platform卡使用应用特定的安装参数字段（由标签“C9”标识）来使应用知道其安装参数，如果应用不需要其特定的参数，则对应的长度为“00”，这个标签和为系统特定参数定义的标签要求安装方法参数字段至少包含值‘9F’、‘00’。

Open Platform卡目前要求一字节的应用优先权，并根据前面的编码规则进行编码，如果只安装一个应用（即，不用相同的INSTALL命令使之成为可选择的），除了“缺省被选择”优先权外，可以使用其它任何优先权。

应该让应用知道实例AID、应用优先权和应用特定的参数，对于实例AID和应用优先权，应该让应用知道其长度和值，对于应用特定参数，TLV对象的长度和值应该让应用知道（即，丢弃标签“C9”）。

INSTALL[设置为可选择]的数据字段：将安装后的应用设置为可选择时使用的INSTALL的APDU（P1的b4设置为1或b4和b3设置为1）的数据字段如下表。

出现	长度（字节数目）	名字
强制	1	下载文件AID的长度=00
不出现		
强制	1	下载文件中AID的长度=00
不出现		
强制	1	应用实例AID的长度
强制	5—16	应用实例AID
强制	1	应用优先权长度
强制	1	应用优先权
强制	1	安装参数长度=00
不出现		安装参数
强制	1	安装标记长度
条件的	0—n	安装标记（安装数据认证模式DAP）

下载和安装参数字段是TLV结构的值，由可选的系统特定的和应用特定的（安装参数）值组成。可以使用下列标签：

标签	长度	值（名字）
C9	可变	应用特定的参数
EF	可变	系统特定的参数
C6	2	非易失性代码空间限制
C7	2	易失性数据空间限制
C8	2	非易失性数据空间限制

卡管理器处理INSTALL时，返回一字节的00，表示没有附加的响应数据。

命令的成功执行用“90”“00”编码。IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
65	81	存储器失败
6A	88	没找到应用的数据
6A	84	存储器空间不够
6A	80	命令数据中值不正确

4.2.2.5 LOAD命令

LOAD命令定义下载应用时LOAD命令数据字段中传输的下载文件结构，但并没有定义IC卡对下载文件的内部处理和存储。

将下载文件分段为便于传输的较小的块，就可以使用多条LOAD命令将一个下载文件逐块传输到卡上。每条LOAD命令从00开始编号，LOAD命令的编号应该严格顺序地按1递增，并通知卡最后一块数据的到达。

接收到整个下载文件后，卡将对下载文件执行必要的内部处理，并执行LOAD命令前面的INSTALL[下载]命令指出的附加处理。

LOAD命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	E8	LOAD
P1	xx	参考控制参数P1
P2	xx	块号
Lc	xx	数据字段长度
Data	xxxx	安装数据
Le	00	

LOAD命令的参考控制参数P1如下编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0								还有更多的块
1								最后一块
	0	0	0	0	0	0	0	保留未来使用

参考控制参数P2包含块号，块号顺序地从00编码到FF。

命令消息的数据字段包含下载文件的一部分。完整的Open Platform下载文件结构如下。

标签	长度	值（名字）
E2	可变	数据认证模式DAP块
4F	5—6	DAP的ID（安全域的AID）
C3	可变	数据认证模式DAP
C4	可变	下载文件数据块

除非下载文件传输到的卡包含具有强制数据认证模式DAP验证优先权的安全域，在下载

文件中，数据认证模式DAP块是可选的。一个下载文件可以包含多个数据认证模式DAP块。

上面定义的长度必须根据BER编码，即，所有小于128字节的长度用一字节编码，128到255的长度值用2字节编码等。

卡管理器在处理最后一条LOAD命令，会返回一字节的00，表示没有附加的响应数据。

命令的成功执行用“90”“00”编码。IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
65	81	存储器失败
6A	86	P1/P2不正确
6A	84	存储器空间不够
69	85	使用条件不满足

4.2.2.6 PUT KEY命令

PUT KEY命令用于：（1）替换现存密钥集合版本中的一个或多个密钥。（2）用新密钥集合版本替换现存密钥集合版本。（3）增加包含一个或多个密钥的新密钥集合版本。

密钥由其密钥集合版本及其密钥索引的组合唯一标识。卡管理器或安全域可以有多个密钥集合版本，给定密钥集合版本中可以有多个密钥。

PUT KEY命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	D8	PUT KEY
P1	xx	参考控制参数P1
P2	xx	参考控制参数P2
Lc	xx	数据字段长度
Data	xxxx	密钥数据
Le	00	

参考控制参数P1用于定义目前密钥集合版本，以及本命令随后是否有更多的PUT KEY命令。目前密钥集合版本标识了一个已经在卡上的密钥集合版本，密钥集合值为00时，表示增加新的密钥集合版本（新的密钥集合版本在命令消息的数据字段中指示）。密钥集合版本编码从01到7F。

PUT KEY命令消息的参考控制参数P1根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0								最后一条（或仅此一条）命令
1								还有更多的PUT KEY命令
	x	x	x	x	X	x	x	当前密钥集合版本

参考控制参数P2用于定义密钥索引，以及数据字段中是否包含多个密钥。如果有一个密钥包含在命令消息数据字段中，P2指示这个密钥的密钥索引，如果有多个密钥包含在命令消息数据字段中，P2指示第一个密钥的密钥索引，数据字段中后面的密钥的索引随这个索引递

增。密钥索引编码从01到7F。

PUT KEY命令消息的参考控制参数P2根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0								单个密钥
1								多个密钥
	x	x	x	x	X	x	x	密钥索引

新密钥集合版本
密钥集合数据字段（隐含密钥索引 P2+0）
密钥集合数据字段（隐含密钥索引 P2+1）
密钥集合数据字段（隐含密钥索引 P2+2）

图 4.2 密钥数据字段的编码

命令消息数据字段包含新的密钥集合版本号，以及一个或多个密钥集合数据字段。新的密钥集合版本指示要在卡上创建的新的密钥集合（当前密钥集合版本在P1=00中）或替代当前密钥集合版本的版本（当前密钥集合版本在P1!=00中）替换密钥时，提交给卡的新密钥的格式必须与卡上已有密钥的格式一样，例如，不能改变现存密钥的大小。使用这个命令下载或替换密钥或私钥时，密钥值应该加密，加密所用的密钥和算法应该根据当前上下文隐含推导出来。

如果数据字段包含多个密钥，则这些密钥都属于相同的密钥集合版本，数据字段中密钥的顺序反映了密钥索引顺序。图4.2说明了数据字段的编码，可以选择下载第二个或第三个密钥。

密钥集合数据字段结构如下表。

长度	含义
1	密钥的算法ID
1—n	密钥长度
可变	密钥数据值
0—n	密钥校验值长度
可变	密钥校验值（如果出现）

响应消息的数据字段包含密钥集合版本的明文，如果有的话，还包括命令消息数据字段中出现的密钥校验值。个人化服务器可以使用这个返回的密钥集合版本和密钥校验值来验证密钥集合版本的正确下载。

命令的成功执行用“90”“00”编码。IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
65	81	存储器失败
6A	88	引用的数据没找到
6A	84	存储器空间不够
94	84	算法不支持

94	85	密钥校验值不正确

4.2.2.7 SELECT命令

SELECT命令用于选择一个应用，卡管理器只处理指示“按名选择”的SELECT命令，所有其它选择被传递给被选择的应用。

SELECT命令消息根据下表编码。

代码	值	含义
CLA	00	一般命令
INS	A4	SELECT
P1	xx	参考控制参数P1
P2	xx	参考控制参数P2
Lc	xx	AID长度
Data	xxxx	要选择的应用或文件的AID
Le	00	

SELECT命令消息的参考控制参数P1根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
					1			根据名字选择

SELECT命令消息的参考控制参数P2根据下表编码。

B8	B7	B6	B5	B4	B3	B2	B1	含义
						0	0	第一次或唯一一次出现
						1	0	下一次出现

命令的数据字段中包含要选择的应用的AID。对于卡管理器，为“A0000000030000”或“A000000003000000”。

响应数据由被选择的应用或文件的特定信息组成。下表定义了卡管理器和安全域的文件控制信息编码。

标签	描述	出现
6F	文件控制信息（FCI模板）	强制
84	应用/文件AID	强制
A5	专有（Proprietary）数据	强制
9F6E	应用产品生命周期数据	强制
9F65	命令消息中数据字段最大长度	强制

命令的成功执行用“90”“00”编码。选择卡管理器时，IC卡可能返回下列警告条件。

SW1	SW2	含义
62	83	卡生命周期状态为CM_LOCKED

IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
6A	81	不支持的功能，例如卡生命周期状态为CM_LOCKED
6A	82	应用或文件不存在

4.2.2.8 SET STATUS命令

SET STATUS命令用于修改卡或当前被选择的应用的生命周期状态。在卡管理器的生命周期状态SECURED，只能根据发卡商的有关安全通信的策略来使用该命令。

SET STATUS命令消息根据下表编码。

代码	值	含义
CLA	80/84	Open Platform命令
INS	F0	SET STATUS
P1	xx	状态类型参数
P2	xx	状态控制参数
Lc	xx	数据字段长度
Data	xxxx	应用的AID
Le	00	

SET STATUS命令的状态类型参数根据下表编码，以指示状态的改变是否应该应用于卡管理器或应用，在这个上下文中，安全域作为应用考虑，其它状态类型是应用特定的或保留给未来使用。

B8	B7	B6	B5	B4	B3	B2	B1	含义
1								卡管理器
	1							应用
		0	0	0	0	0	0	保留给未来使用

状态控制参数P2根据前面的编码规则来编码。

数据字段应该包括卡管理器或应用的AID。

响应消息中没有数据字段。

命令的成功执行用“90”“00”编码。IC卡可能返回下列附加的错误条件。

SW1	SW2	含义
6A	88	引用的数据不存在
6A	80	命令数据的值不正确

4.3 安全通道的建立

4.3.1 安全通道概述

安全通道在卡和卡外实体之间提供了一个安全的通信通道。安全通道提供的3级安全性是：（1）相互认证。卡和卡外实体各自证明它们具有同一秘密的知识。（2）完整性和认证。靠保证从卡外实体接收到的数据实际上来自认证后的卡外实体，顺序正确并没被修改。（3）隐秘。从卡外实体传输到卡上的数据对于未认证后的实体是不可见的。

安全通道的使用范围和规则如下：

（1）所有应用，包括卡管理器和安全域。（1a）选择应用（SELECT命令）从不使用安全通道，选择过程由于其自身的本质，总是在安全通道外部。（1b）安全通道的发起总是从卡和卡外实体之间的相互认证开始。（1c）安全通道发起后的安全消息传递的安全级别在发起安全通道期间定义。（1d）如果是向卡传送密钥，无论在整个APDU消息中是否使用隐秘和/或完整性，数据字段中的密钥必须使用附加层次的隐秘进行安全保证。（1e）在所有与安全通道相关的密码学功能中，要使用数据加密标准DES，特别是3级倍DES。（1f）在安全域中使用的所有密钥必须是提供112位加密强度的16字节（128位）密钥。

（2）卡管理器。（2a）查询基本的卡和/或发卡商信息(GET DATA)从部需要使用安全通道。GET DATA命令应该在安全通道内部或外部执行。（2b）所有其它卡管理器APDU命令或者涉及在安全通道发起之中，或者必须在安全通道内使用。（2c）在卡到达安全状态之前（即，OP_READY或INITIALIZED状态时），卡假设其处于安全环境中，卡和卡外实体必须至少相互认证一次，以打开安全通道。（2d）一旦达到安全状态（即，SECURE或CM_LOCKED状态），卡总是假设其处于不安全的环境中，卡和卡外实体必须相互认证，所有从卡外实体到卡的通信必须包含完整性校验。

4.3.2 相互认证

相互认证通过发起安全通道的过程来达到，并向卡和卡外提供保证：它们都在与认证后的实体通信。这意味着，两个实体具有对相同秘密信息（卡静态密钥）的知识。如果相互认证中任何一步失败，就必须重新开始认证过程，即产生新挑战和会晤密钥。

（1）安全通道总是由卡外实体通过向卡传递一个“主机”挑战（对本次会晤唯一的随机数）来发起（参见INITIALIZE UPDATE命令）。（2）卡接收到挑战后，产生自己的“卡”挑战（又是对本次会晤唯一的随机数）。（3）卡利用主机挑战、卡挑战和内部静态密钥，创建新的秘密数据（会晤密钥），并使用新创建的会晤密钥之一，产生第一个密码值（卡密文）（参见后面）。（4）卡密文和卡挑战，以及其它数据被传输到卡外实体。（5）由于卡外实体应该知道卡用于产生卡密文的同样信息，它就应该能产生相同的密文，并完成比较，就能认证卡。（6）卡外实体现在使用类似的过程，创建第二个密码值（主机密文），传送回卡（参见EXTERNAL AUTHENTICATE命令）。（8）由于卡应该知道卡外实体用于产生主机密文的同样信息，它就应该能产生相同的密文，并完成比较，就能认证卡外实体。

图4.3中的流程是卡和卡外实体之间相互认证的例子，这个流程说明了发生在卡管理器和卡外实体之间的相互认证。

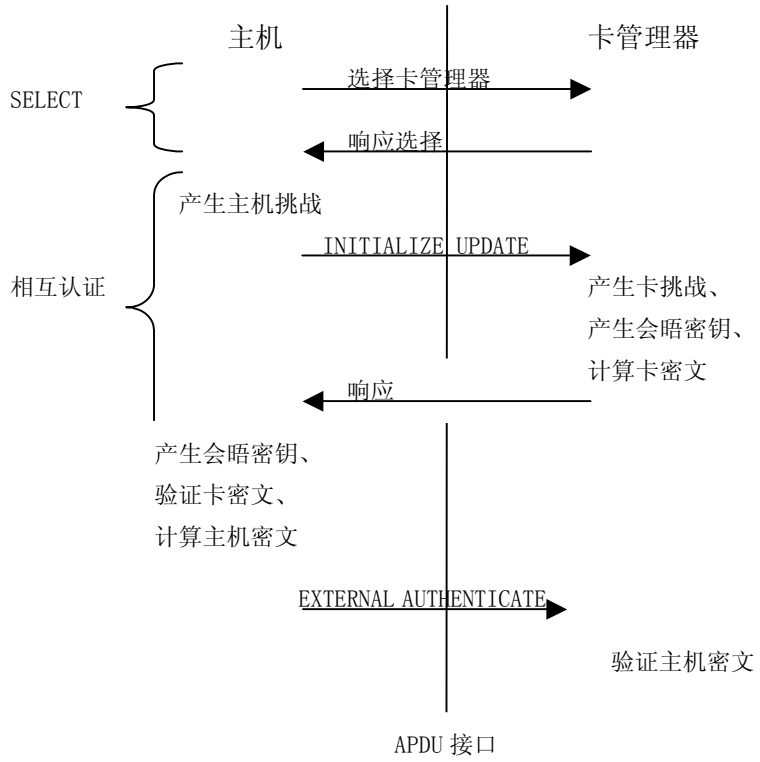

图 4.3　相互认证流程(卡管理器)

4.3.3　安全消息传递

安全消息传递允许卡外实体在命令传输到卡上之前，对APDU命令的组成增加隐秘性和/或完整性。

（1）消息完整性

消息完整性为卡提供一种手段，来验证从通过认证的卡外实体接收到的命令没有被修改，以及验证从通过认证的卡外实体接收到的命令是由认证后的卡外实体按顺序传送的。

这个级别的完整性是通过应用多个链接的DES运算到APDU命令的头部和数据字段，产生一个消息认证码MAC来实现的（使用了相互认证过程中产生的会晤密钥）。

卡接收到包含MAC的消息后，使用相同的会晤密钥，完成相同的运算，将内部产生的MAC与从卡外实体接收到的MAC比较，保证整个命令的完整性。（如果对消息数据应用了隐秘，MAC仅用于明文数据）。

传输到卡上的命令序列的完整性是通过使用来自当前命令作为下一个命令的初始链接向量（Initial Chaining Vector，ICV）来实现的，这向卡保证，所有命令按顺序接收到。

（2）消息数据隐秘性

数据隐秘提供了一种手段，在跨过开放网络上传输的数据免于被拦截后进行暴露分析的担心。这个级别的安全性不能应用于作为单独安全消息传递系统的消息，仅能与消息完整性一起使用。

在要传输到卡上的命令消息的数据字段间，应用多级链接DES运算（使用相互认证过程中产生的会晤密钥），可以达到这个级别的隐秘。

多个消息之间隐秘性没有联系，即，消息之间没有链式要求。

4.3.4 安全通道中密钥的生成

每次发起一个安全通道时，就产生DES会晤密钥，并在相互认证过程中使用。如果安全级别指出需要安全的消息传递，这些相同的会晤密钥可以用于随后的命令。

产生会晤密钥是要保证有不同的密钥集合用于每次安全通信，尽管这对于密钥加密运算并不是必须的，但它对于认证运算、MAC的产生和验证、命令消息的加密和解密都很重要。因此，很有必要仅从静态加密和MAC密钥中创建会晤密钥。

使用静态加密和MAC密钥、随机的主机和卡挑战来创建DES会晤密钥。创建会晤密钥涉及3个步骤：（1）产生会晤密钥导出数据（相同的导出数据用于创建加密和MAC会晤密钥）。图4.4。（2）创建加密会晤密钥。图4.5。（3）创建MAC会晤密钥。图4.6。

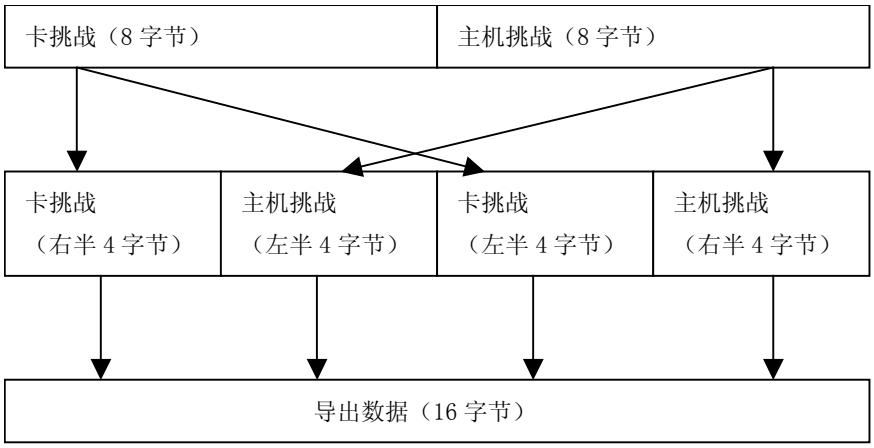

图 4.4　产生导出数据

用于产生这些密钥的DES运算总是电子码本ECB模式的3级DES运算。

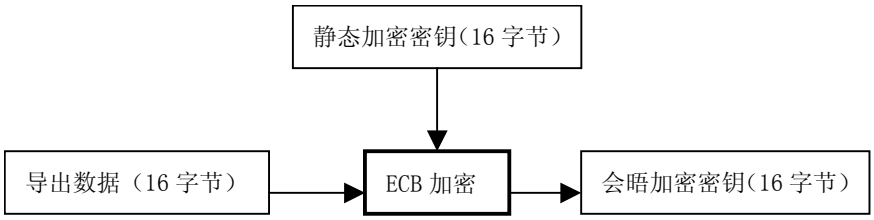

图 4.5　创建会晤加密密钥

4.3.5 认证密文

卡和卡外实体（主机）都产生认证密文。卡外实体验证卡密文，卡验证主机密文。认证密文的产生和验证要使用加密会晤密钥和第3章中描述的签名方法。

（1）卡认证密文

卡密文的产生和认证是通过将8字节主机挑战和8字节卡挑战串接，得到16字节的块。

将相同的填充规则应用于APDU的MAC功能，数据必须进一步用的8字节块（80 00 00 00 00 00 00 00）填充。

签名方法使用加密会晤密钥和8字节的0x00作为初始链接向量ICV，应用于这个24字节的块，得到的8字节结果就是卡签名。

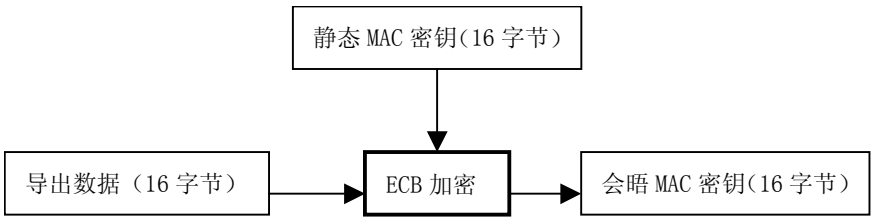

图 4.6　创建会晤 MAC 密钥

（2）主机认证密文

主机密文的产生和认证是通过将8字节主机挑战和8字节卡挑战串接，得到16字节的块。

将相同的填充规则应用于APDU的MAC功能，数据必须进一步用的8字节块（80 00 00 00 00 00 00 00）填充。

签名方法使用加密会晤密钥和8字节的0x00作为初始链接向量ICV，应用于这个24字节的块，得到的8字节结果就是主机签名。

4.3.6　认证命令

4.3.6.1　INITIALIZEUPDATE命令

初始化一条安全通道，允许卡被主机认证，计算用于 MAC 计算和随后 APDU 中命令加密的会话密钥。

INITIALIZEUPDATE命令消息根据下表编码。

字段	内容
CLA	80
INS	50
P1	发卡商安全域的密钥集合版本。00=使用当前密钥集合版本，这是在使用初始化密钥集合时唯一允许的值
P2	选择初始化安全通道的密钥的密钥集合版本
Lc	08
数据	主机随机数
Le	1C

参数 P1 指示安全通道密钥集合中的密钥，这个密钥作为在选择初始化安全通道的密钥时的起始点。

值	描述

00	密钥索引由应用隐含知道
01	用作认证和加密的第一个会话密钥，要使用安全通道密钥集合中的第一个密钥计算出来。用于 MAC 计算的第二个会话密钥，要使用密钥集合中的第二个密钥计算出来。用于密钥加密的第三个会话密钥，要从密钥集合中的第三个密钥计算出来。
02	第一个会话密钥使用密钥集合中的第二个密钥计算出来，第二个会话密钥使用密钥集合中的第三个密钥计算出来，第三个会话密钥使用密钥集合中的第一个密钥计算出来
03	第一个会话密钥使用密钥集合中的第三个密钥计算出来，第二个会话密钥使用密钥集合中的第一个密钥计算出来，第三个会话密钥使用密钥集合中的第二个密钥计算出来。

所有其它值均保留给将来使用。

响应数据字段包含卡标识数据、卡随机数和卡认证数据。

长度	描述
0A	卡标识
01	密钥集合版本（如果 P1!=0，则等于 P1）
01	起始密钥索引（如果 P2!=0，则等于 P2）
08	卡随机数
08	主机认证卡使用的卡密文

如果在 INITIALIZEUPDATE 命令前的 SELECT 命令规定了安全域，则卡标识为：

字节	描述
1—2	安全域 AID 的最后两个字节
3—4	IC 制造数据
5—8	IC 序列号
9—10	IC 批标识符

否则为：

字节	描述
1—2	卡管理器 AID 的最后两个字节
3—4	IC 制造数据
5—8	IC 序列号
9—10	IC 批标识符

响应状态：

状态	含义
6700	数据长度错误
6985	使用条件不满足
6A86	P1 和/或 P2 不正确
6E00	CLA 不正确
9000	成功执行

4.3.6.2 EXTERNALAUTHENTICATE命令

证明卡和外部设备都拥有相同的密钥，这是用于发起安全通道的命令序列中的一部分。要求在当前会话前，必须已经发布了 INITIALIZEUPDATE 命令。

EXTERNALAUTHENTICATE命令消息根据下表编码。

字段	内容
CLA	84
INS	82
P1	安全控制参数
P2	00----FF
Lc	10
数据	认证数据（主机密文），随后是 MAC
Le	无

参数P1用于设置EXTERNALAUTHENTICATE命令以后和安全通道中所有安全消息传递命令的安全级别。

B8	B7	B6	B5	B4	B3	B2	B1	含义
0	0	0	0	0	0			保留
						1	1	命令数据的 MAC 插入和加密
						0	1	MAC 插入
						0	0	明文模式，仅在 OP_READY 和 INITIALIZED 阶段使用

响应状态：

状态	含义
6300	认证失败（主机密文没有通过验证）
6700	Lc 参数长度错误
6985	使用条件不满足（没有 INITIALIZEUPDATE）
6A86	P1 和/或 P2 不正确
6A88	MAC 没有通过验证
6E00	CLA 不正确
9000	成功执行（对象被物理删除）

第 5 章　发行管理应用的开发

5.1　发行管理应用的功能

发行管理应用的主要功能是根据制卡程序的要求，生成能够发送给用户卡的卡管理器的命令，由制卡程序发送给卡管理器，以便与卡管理器交互，进行Applet的下载、安装和删除等。

在从Java卡制造商得到空白的Java时，卡制造商提供了每张卡的卡管理器的当前密钥集合的编号，以及当前密钥集合中的3个密钥的值。制卡程序将这3个密钥下载到发行管理应用中，作为初始静态密钥集合，发行管理应用根据初始静态密钥，生成制卡程序与用户卡的卡管理器交互的命令，进行发行制卡，即进行Applet的下载和安装。

发行制卡完成后，发行管理应用从应用根密钥应用取得卡管理器根密钥，再从用户卡的卡管理器取得Java卡的序列号，作为分散参数，得到用户卡的卡管理器的分散后密钥，再用这3个分散后密钥替换用户卡的卡管理器的当前密钥，并将用户卡的卡管理器切换到SECURED状态。这时，用户卡进入发行后阶段。

在用户卡进入发行后阶段之后，如果需要删除废弃的Applet实例或包，或者下载安装新的Applet包或实例，即发行后制卡。这时，发行管理应用从应用根密钥应用取得卡管理器根密钥，再从用户卡的卡管理器取得Java卡的序列号，作为分散参数，得到用户卡的卡管理器的分散后密钥。这样，发行管理应用仍能生成制卡程序与用户卡的卡管理器交互的命令，进行发行后制卡。

发行制卡和发行后制卡的流程分别如图5.1和图5.2。

5.2　密钥的计算

5.2.1　导出会晤密钥

每次在制卡应用（称为主机或外部实体）与卡管理器之间建立一个安全通道时，就在制卡应用与卡管理器之间建立了一个会晤，一个会晤有3个密钥，这3个密钥的生命周期为会晤的生命周期。会晤从INITIALIZE_UPDATE开始，到下一个INITIALIZE_UPDATE结束，或到卡被拔出读卡器时结束。

在GET_EXTERNAL_AUTHENTICATE中，发行管理应用需要计算会晤密钥，使用卡管理器的当前密钥集合（初始静态密钥集合或分散后的密钥集合），以及用户卡卡管理器产生的随机数和发行管理应用产生的随机数，计算得到用于安全通道通信的3个密钥。如图5.3。

5.2.2　计算发行管理应用的密文

在GET_EXTERNAL_AUTHENTICATE中，发行管理应用需要计算自己的密文，供用户卡卡管理器验证。计算过程如图5.4。8字节的填充是ISO9797M2填充，即：0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00。

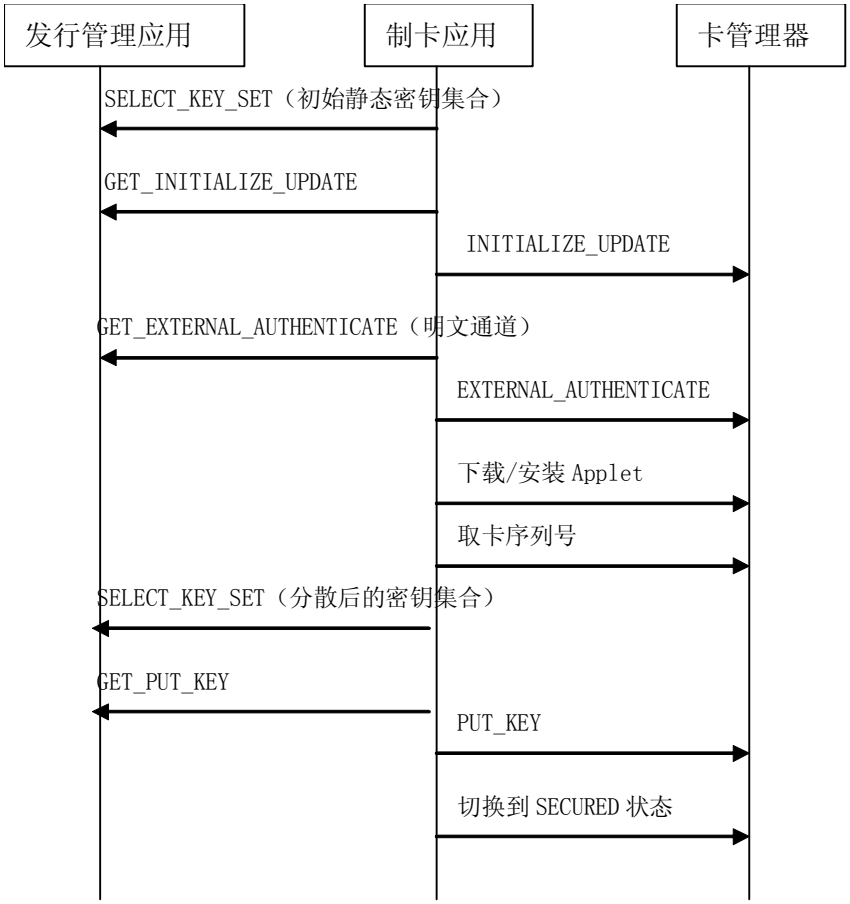

图 5.1　发行制卡

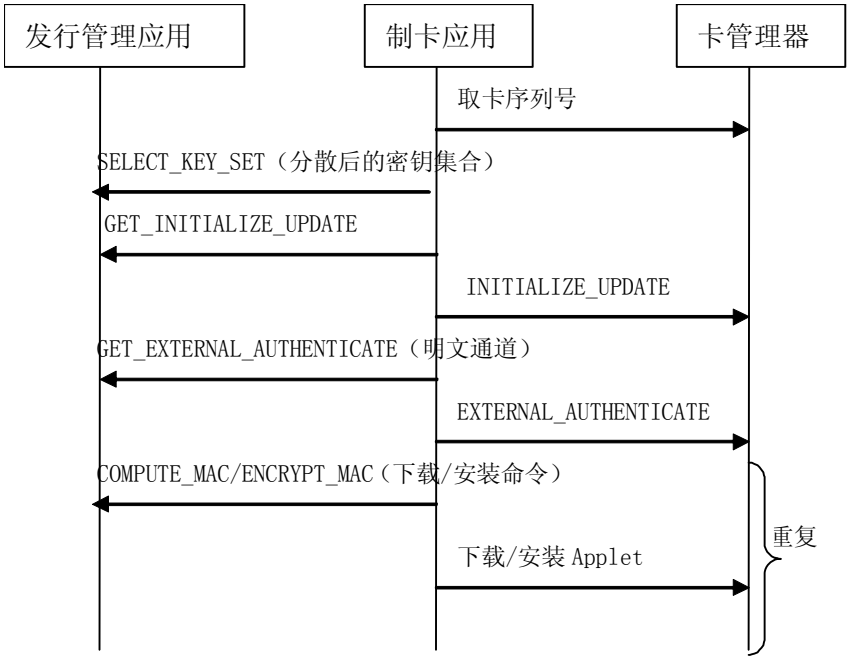

图 5.2　发行后制卡

5.2.3 验证用户卡卡管理器的密文

在GET_EXTERNAL_AUTHENTICATE中，发行管理应用需要验证用户卡卡管理器的密文。验证过程如图5.5。8字节的填充是ISO9797M2填充，即：0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00。

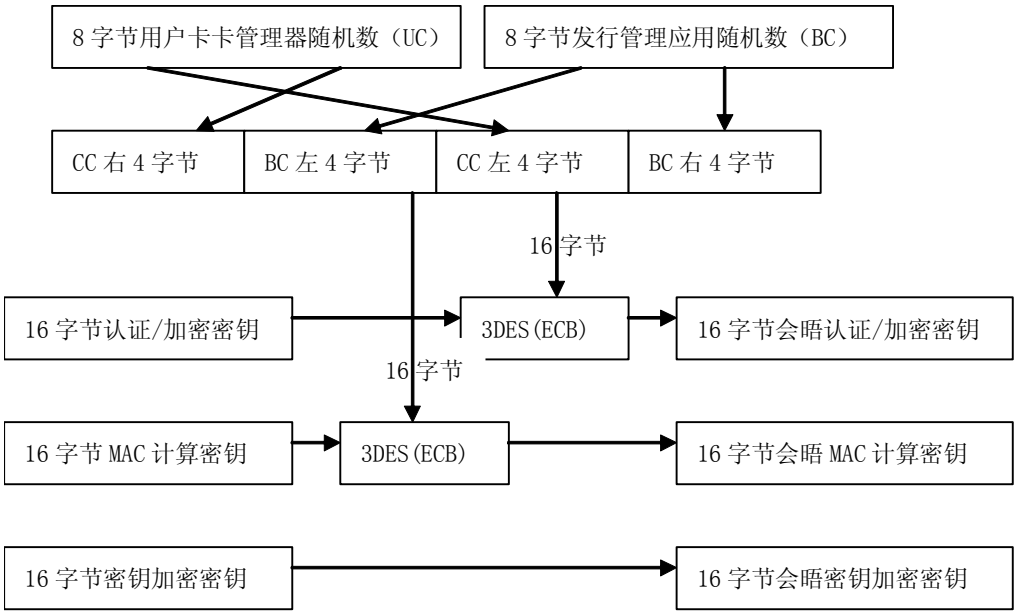

图 5.3 会晤密钥的计算

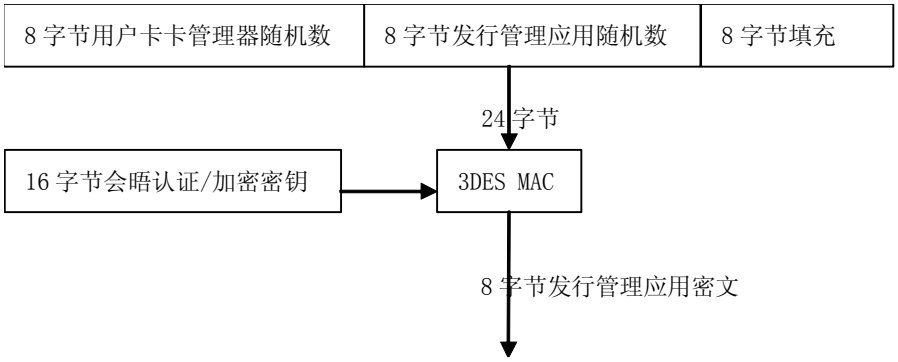

图 5.4 计算发行管理应用的密文

5.2.4 计算分散后密钥

发行管理应用需要计算卡管理器分散后的密钥，计算需要6字节的卡序列号，填充0xCC和0x33成为8字节的分散参数，与卡管理器根密钥混合。如图5.6。

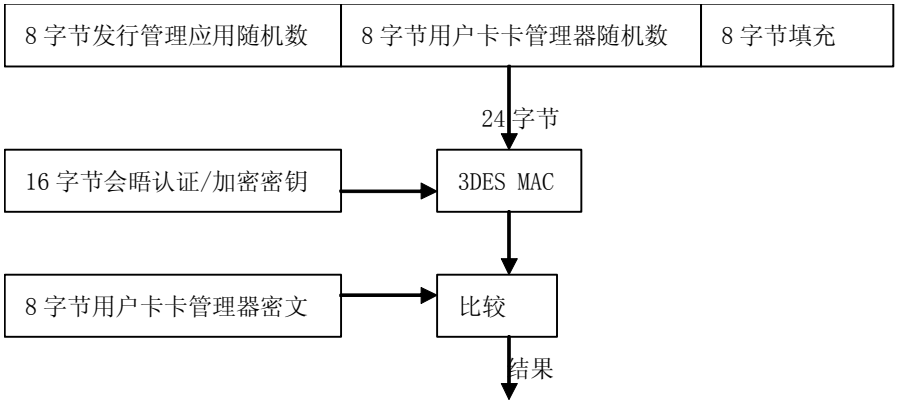

图 5.5 验证用户卡卡管理器的密文

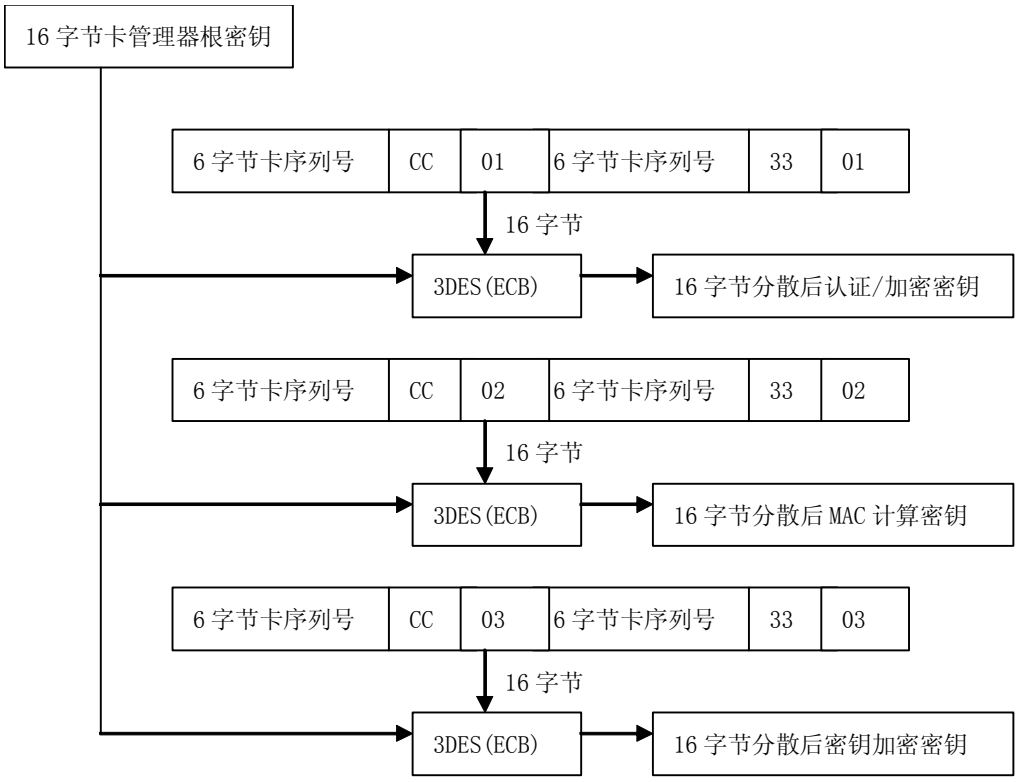

图 5.6 计算卡管理器分散后的密钥

5.2.5 密钥加密和密钥校验值计算

在发行制卡阶段，发行管理应用计算出的用户卡卡管理器的新密钥需要下载到用户卡上，在下载前，需要用密钥加密密钥对新密钥加密，并计算校验值。这个过程如图5.7。

5.2.6 计算命令的MAC

在发行后制卡阶段，所有发向用户卡卡管理器的命令都必须附加MAC。MAC的计算如图5.8。填充采用ISO9797M2填充：首先附加0x80，如果这时填充后命令长度为8的倍数，就不再填充，否则，继续附加0x00，直到命令的长度为8的倍数为止。

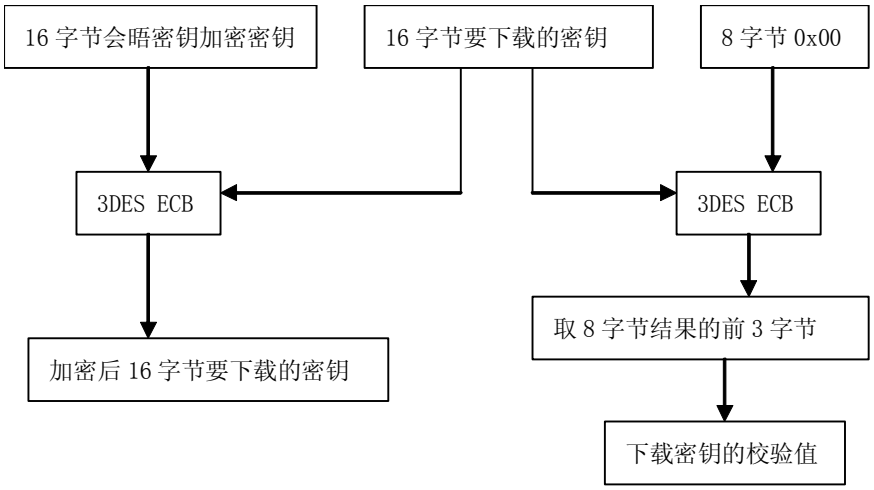

图 5.7 密钥的加密和校验值的计算

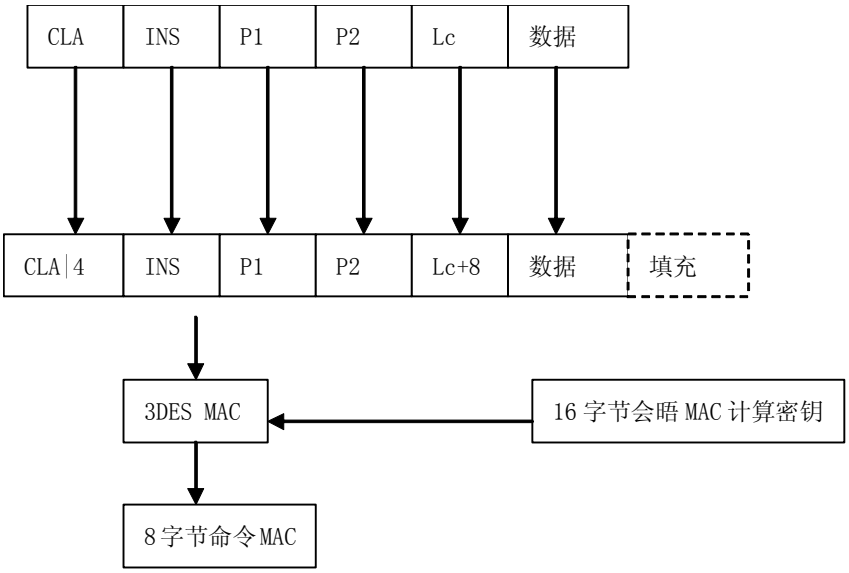

图 5.8 计算命令的 MAC

5.2.7 计算命令的MAC并加密命令的数据

在发行后制卡阶段，所有发向用户卡卡管理器的命令都必须附加MAC，还可能对命令的数据进行加密。这时，应该先计算MAC，再进行数据加密。MAC的计算如图5.8。数据的加密如图5.9。填充采用ISO9797M2填充：首先附加0x80，如果这时填充后命令长度为8的倍数，

就不再填充，否则，继续附加0x00，直到命令的长度为8的倍数为止。

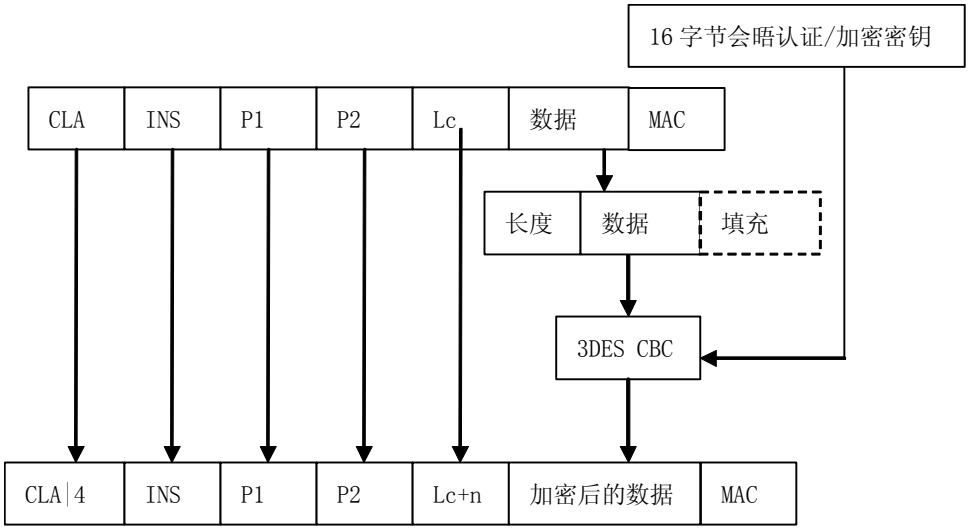

图 5.9　加密命令的数据

5.3　发行管理应用的AID

系统根密钥应用 MasterApp 的 AID=0x4B:0x45:0x59:0x42:0x43，即字符串“KEYBC ”，它在根密钥卡上的实例的 AID 为：0x4B:0x45:0x59:0x42:0x43:0x01，它在发行卡上的实例的 AID 为：0x4B:0x45:0x59:0x42:0x43:0x02。

5.4　接口设计

5.4.1　PUT_KEY

本命令必须在安全通道中使用，它的安全级别必须与EXTERNAL_AUTHENTICATE命令中定义的安全级别匹配。

命令消息格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0xD8	0xXX	0x00	0xXX	密钥数据	无
	PUT_KEY	当前密钥集合	未使用	密钥数据长度		

P1=0x01：初始静态密钥（集合）
P1=0x02：分散后密钥（集合）

数据格式：

字段号	长度	内容
1	0x01	密钥算法标识符，0x81=ECB模式
2	0x01	密钥长度，0x10
3	可变	密钥值（已经使用密钥加密密钥进行了加密）
4	0x01	密钥校验值长度，总是为0x03
5	0x03	密钥校验值（空消息使用密钥加密后的前3字节）

发行管理应用在接收到命令后，根据P1：（1）选择初始静态密钥（集合）；或者（2）根据给定的分散参数，使用分散算法，计算最后主密钥，并选择。然后返回。

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6581	内存错误
0x6985	条件不满足（安全通道没有打开）
0x6a80	数据字段内容不正确
0x6a84	内存不够
0x6a88	没有找到引用的数据（指定的密钥集合版本不存在）
0x9000	正确
0x9484	算法不支持
0x9485	密钥值不正确

5.4.2 SELECT_KEY

本命令选择一个用于打开安全通道的密钥集合，或者选择要下载到子卡中的密钥集合。

命令消息格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x3A	0xXX	0x00	0xXX	分散参数	无
	SELECT_KEY_SET	要选择的密钥集合	未使用	数据长度		

P1=0x01：初始静态密钥（集合），这时，没有数据字段。
P1=0x02：分散后密钥（集合），这时，必须有数据字段。

数据格式：

字段号	长度	内容
1	0x06	分散参数

发行管理应用在接收到命令后，根据P1：（1）选择初始静态密钥（集合）；或者（2）根据给定的分散参数，使用分散算法，计算最后主密钥，并选择。然后返回。

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6700	Lc不正确
0x9000	正确

5.4.3 GET_INITIALIZE_UPDATE

本命令构造一个INITIALIZE_UPDATE命令，这个命令将被发送到用户卡卡管理器，用于打开到用户卡卡管理器的安全通道。

命令格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x36	0xXX	0x00	无	无	0x0D
	GET_INITIALIZE_UPDATE	密钥集合版本号	未使用			响应数据长度

返回的数据格式：

字段号	长度	内容
1	0x01	0x80，CLA
2	0x01	0x50, INS
3	0x01	0xXX, P1，密钥版本号
4	0x01	0x00, P2
5	0x01	0x08, Lc
6	0x08	发行管理应用的随机数

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6c0d	Le不正确
0x9000	正确

5.4.4 GET_EXTERNAL_AUTHENTICATE

本命令构造一个EXTERNAL_AUTHENTICATE命令，这个命令将被发送到用户卡卡管理器，用于打开到用户卡卡管理器的安全通道。在调用这个命令前，必须使用SELECT_KEY_SET命令，选择用户卡卡管理器上的密钥集合，并已经使用GET_INITIALIZE_UPDATE初始化了到用户卡卡管理器的安全通道。

命令格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x38	0xXX	0x00	0x1C	子卡的响应	0x15
	GET_EXTERNAL_AUTHENTICATE	安全级别	未使用	数据长度		响应数据长度

P1=0x00：无安全消息
P1=0x01：MAC安全消息
P1=0x03：MAC和加密的安全消息

数据格式：

字段号	长度	内容
1	0x0A	密钥分散参数
2	0x02	密钥信息数据
3	0x08	用户卡卡管理器随机数
4	0x08	用户卡卡管理器密文

发行管理应用在接收到命令后，从选择的密钥集合，计算安全通道会话密钥，检查用户卡卡管理器密文。然后使用发行管理应用密文，构造发送到子卡的EXTERNAL_AUTHENTICATE命令，并计算命令的MAC。

返回的数据格式：

字段号	长度	内容
1	0x01	0x84：CLA
2	0x01	0x82：INS
3	0x01	0xXX：P1
4	0x01	0x00：P2
5	0x01	0x10：数据长度
6	0x08	发行管理应用密文
7	0x08	命令的MAC

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6700	Lc不正确
0x6c15	Le不正确
0x6985	命令顺序错误
0x6a80	数据字段错误
0x9000	正确

5.4.5 GET_PUT_KEY

本命令构造一个PUT_KEY命令，这个命令将被发送到用户卡卡管理器，用于下载发行管理应用的密钥集合。在调用这个命令前，必须使用SELECT_KEY_SET命令，选择了要下载到用户卡卡管理器上的密钥集合，并已经使用GET_INITIALIZE_UPDATE和GET_EXTERNAL_AUTHENTICATE初始化并建立了到用户卡卡管理器的安全通道。

命令格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x3C	0xXX	0x00	无		0x48
	GET_PUT_KEY	当前密钥集合版本号	新的密钥集合版本号			响应数据长度

发行管理应用在接收到命令后，使用会话密钥加密选择的密钥集合中的3个密钥，计算这3个密钥的校验值。

返回的数据格式：

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6cXX	Le不正确
0x6985	命令顺序错误
0x9000	正确

5.4.5 COMPUTE_MAC

本命令构造一个包含MAC的、将被发送到用户卡卡管理器的命令。在调用这个命令前，必须使用GET_INITIALIZE_UPDATE和GET_EXTERNAL_AUTHENTICATE初始化并建立了到用户卡卡管理器的安全通道。

命令格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x3E	0x00	0x00	0xXX	数据	0xXX
	COMPUTE_MAC	未使用	未使用	数据长度		响应数据长度

数据格式：

字段号	长度	内容
1	0x01	CLA,其中b3=0

2	0x01	INS
3	0x01	P1
4	0x01	P2
5	0x01	Lc，不包括MAC长度
6	可变	数据

发行管理应用在接收到命令后，计算数据字段的MAC，附加在数据后面返回。

返回数据格式：

字段号	长度	内容
1	0x01	CLA,其中b3=0
2	0x01	INS
3	0x01	P1
4	0x01	P2
5	0x01	Lc，包括MAC长度
6	可变	数据
7	0x08	MAC

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6cXX	Le不正确
0x6985	命令顺序错误
0x9000	正确

5.4.6 ENCRYPT_MAC

本命令构造一个数据已经加密、包含MAC的、将被发送到用户卡卡管理器的命令。在调用这个命令前,必须使用GET_INITIALIZE_UPDATE和GET_EXTERNAL_AUTHENTICATE初始化并建立了到用户卡卡管理器的安全通道。

命令格式：

CLA	INS	P1	P2	Lc	DataField	Le
0x80	0x3E	0x00	0x00	0xXX	数据	0xXX
	COMPUTE_MAC	未使用	未使用	数据长度		响应数据长度

数据格式：

字段号	长度	内容
1	0x01	CLA,其中b3=0
2	0x01	INS
3	0x01	P1

4	0x01	P2
5	0x01	Lc，不包括MAC长度
6	可变	数据

发行管理应用在接收到命令后，计算数据字段的MAC，加密数据字段中的数据字段，将MAC附加在数据后面返回。

返回数据格式：

字段号	长度	内容
1	0x01	CLA,其中b3=0
2	0x01	INS
3	0x01	P1
4	0x01	P2
5	0x01	Lc，包括MAC长度和加密填充长度
6	可变	数据
7	0x08	MAC

返回状态字：

0x6e00	CLA不正确
0x6a86	P1/P2不正确
0x6cXX	Le不正确
0x6985	命令顺序错误
0x9000	正确

第 6 章　PC/SC接口编程

随着智能卡的广泛应用，为解决计算机与各种读卡器之间的互操作性问题，人们提出了 PC/SC（Personal Computer/Smart Card）规范，PC/SC 规范作为读卡器和卡与计算机之间有一个标准接口，实现不同生产商的卡和读卡器之间的互操作性，其独立于设备的 API 使得应用程序开发人员不必考虑当前实现形式和将来实现形式之间的差异，并避免了由于基本硬件改变而引起的应用程序变更，从而降低了软件开发成本。

Microsoft 在其 Platform SDK 中实现了 PC/SC，作为连接智能卡读卡器与计算机的一个标准模型，提供了独立于设备的 API，并与 Windows 平台集成。因此，我们可以用 PC/SC 接口来访问智能卡。

6.1　PC/SC概述

PC/SC 接口包含 30 多个以 Scard 为前缀的函数，所有函数的原型都在 winscard.h 中声明，应用程序需要包含 winscard.lib，所有函数的正常返回值都是 SCARD_S_SUCCESS。在这 30 多个函数中，常用的函数只有几个，与智能卡的访问流程（图 6.1）对应，下面将详细介绍这些常用函数。

6.2　PC/SC的主要函数

6.2.1　建立资源管理器的上下文

函数 ScardEstablishContext()用于建立将在其中进行设备数据库操作的资源管理器上下文（范围）。

函数原型：LONG SCardEstablishContext(DWORD dwScope, LPCVOID pvReserved1, LPCVOID pvReserved2, LPSCARDCONTEXT phContext);

各个参数的含义：

（1）dwScope：输入类型；表示资源管理器上下文范围，取值为：SCARD_SCOPE_USER（在用户域中完成设备数据库操作）、SCARD_SCOPE_SYSTEM（在系统域中完成设备数据库操作）。要求应用程序具有相应的操作权限。

（2）pvReserved1：输入类型；保留，必须为 NULL。

（3）pvReserved2：输入类型；保留，必须为 NULL。

（4）phContext：输出类型；建立的资源管理器上下文的句柄。

下面是建立资源管理器上下文的代码：

```
SCARDCONTEXT        hSC;
LONG                lReturn;
lReturn = SCardEstablishContext(SCARD_SCOPE_USER, NULL, NULL, &hSC);
if ( lReturn!=SCARD_S_SUCCESS )
    printf("Failed SCardEstablishContext\n");
```

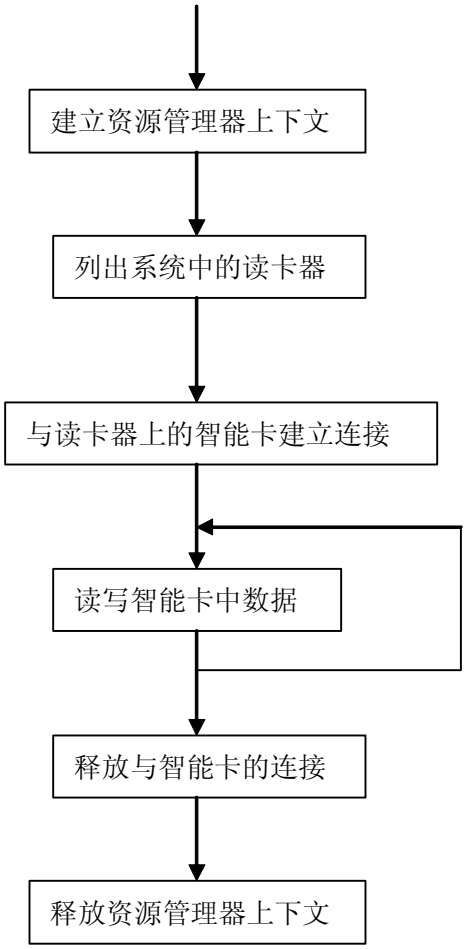

图 6.1 PC/SC 智能卡的访问流程

6.2.2 获得系统中安装的读卡器列表

函数 ScardListReaders()可以列出系统中安装的读卡器的名字。

函数原型：LONG SCardListReaders(SCARDCONTEXT hContext, LPCTSTR mszGroups, LPTSTR mszReaders, LPDWORD pcchReaders);

各个参数的含义：

（1）hContext：输入类型；ScardEstablishContext()建立的资源管理器上下文的句柄，不能为 NULL。

（2）mszGroups：输入类型；读卡器组名，为 NULL 时，表示列出所有读卡器。

（3）mszReaders：输出类型；系统中安装的读卡器的名字，各个名字之间用’\0’分隔，最后一个名字后面为两个连续的’\0’。

（4）pcchReaders：输入输出类型；mszReaders 的长度。

系统中可能安装多个读卡器，因此，需要保存各个读卡器的名字，以便以后与需要的读卡器建立连接。

下面是获得系统中安装的读卡器列表的代码：

```
char            mszReaders[1024];
LPTSTR          pReader, pReaderName[2];
```

```
DWORD           dwLen=sizeof(mzsReaders);
int             nReaders=0;
lReturn = SCardListReaders(hSC, NULL, (LPTSTR)mszReaders, &dwLen);
if ( lReturn==SCARD_S_SUCCESS )
{
    pReader = (LPTSTR)pmszReaders;
    while (*pReader !='\0'  )
    {
        if ( nReaders<2 ) //使用系统中前 2 个读卡器
            pReaderName[nReaders++]=pReader;
        printf("Reader: %S\n", pReader );
        //下一个读卡器名
        pReader = pReader + strlen(pReader) + 1;
    }
}
```

6.2.3 与读卡器（智能卡）连接

函数 ScardConnect()在应用程序与读卡器上的智能卡之间建立一个连接。

函数原型：LONG SCardConnect(SCARDCONTEXT hContext, LPCTSTR szReader, DWORD dwShareMode, DWORD dwPreferredProtocols, LPSCARDHANDLE phCard, LPDWORD pdwActiveProtocol);

各个参数的含义：

（1）hContext：输入类型；ScardEstablishContext()建立的资源管理器上下文的句柄。

（2）szReader：输入类型；包含智能卡的读卡器名称（读卡器名称由 ScardListReaders()给出）。

（3）dwShareMode：输入类型；应用程序对智能卡的操作方式，SCARD_SHARE_SHARED（多个应用共享同一个智能卡）、SCARD_SHARE_EXCLUSIVE（应用独占智能卡）、SCARD_SHARE_DIRECT（应用将智能卡作为私有用途，直接操纵智能卡，不允许其它应用访问智能卡）。

（4）dwPreferredProtocols：输入类型；连接使用的协议，SCARD_PROTOCOL_T0（使用 T=0 协议）、SCARD_PROTOCOL_T1（使用 T=1 协议）。

（5）phCard：输出类型；与智能卡连接的句柄。

（6）PdwActiveProtocol：输出类型；实际使用的协议。

下面是与智能卡建立连接的代码：

```
SCARDHANDLE     hCardHandle[2];
DWORD           dwAP;
lReturn = SCardConnect( hContext, pReaderName[0],    SCARD_SHARE_SHARED,
    SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1, &hCardHandle[0], &dwAP );
if ( lReturn!=SCARD_S_SUCCESS )
{
    printf("Failed SCardConnect\n");
    exit(1);
}
```

与智能卡建立连接后，就可以向智能卡发送指令，与其交换数据了。

6.2.4 向智能卡发送指令

函数 ScardTransmit()向智能卡发送指令，并接受返回的数据。

函数原型：LONG SCardTransmit(SCARDHANDLE hCard, LPCSCARD_IO_REQUEST pioSendPci, LPCBYTE pbSendBuffer, DWORD cbSendLength, LPSCARD_IO_REQUEST pioRecvPci, LPBYTE pbRecvBuffer, LPDWORD pcbRecvLength);

各个参数的含义：

（1）hCard：输入类型；与智能卡连接的句柄。

（2）pioSendPci：输入类型；指令的协议头结构的指针，由 SCARD_IO_REQUEST 结构定义。后面是使用的协议的协议控制信息。一般使用系统定义的结构，SCARD_PCI_T0（T=0 协议）、 SCARD_PCI_T1（T=1 协议）、SCARD_PCI_RAW（原始协议）。

（3）pbSendBuffer：输入类型；要发送到智能卡的数据的指针。

（4）cbSendLength：输入类型；pbSendBuffer 的字节数目。

（5）pioRecvPci：输入输出类型；指令协议头结构的指针，后面是使用的协议的协议控制信息，如果不返回协议控制信息，可以为 NULL。

（6）pbRecvBuffer：输入输出类型；从智能卡返回的数据的指针。

（7）pcbRecvLength：输入输出类型；pbRecvBuffer 的大小和实际大小。

对于 T=0 协议，收发缓冲的用法如下：

（a）向智能卡发送数据：要向智能卡发送 n>0 字节数据时，pbSendBuffer 前 4 字节分别为 T=0 的 CLA、INS、P1、P2，第 5 字节是 n，随后是 n 字节的数据；cbSendLength 值为 n+5(4 字节头+1 字节 Lc+n 字节数据)。PbRecvBuffer 将接收 SW1、SW2 状态码;pcbRecvLength 值在调用时至少为 2，返回后为 2。

```
BYTE        recvBuffer[260];
int         sendSize, recvSize;
BTYE        sw1, sw2;
BYTE    select_mf[]={0xC0, 0xA4, 0x00, 0x00, 0x02, 0x3F, 0x00};
sendSize=7;
recvSize=sizeof(recvBuffer);
lReturn = SCardTransmit(hCardHandle[0], SCARD_PCI_T0, select_mf, sendSize,
    NULL, recvBuffer, &recvSize);
if ( lReturn != SCARD_S_SUCCESS )
{
    printf("Failed SCardTransmit\n");
    exit(1);
}
//返回的数据，recvSize=2
sw1=recvBuffer[recvSize-2];
sw2=recvBuffer[recvSize-1];
```

（b）从智能卡接收数据：为从智能卡接收 n>0 字节数据，pbSendBuffer 前 4 字节分别为 T=0 的 CLA、INS、P1、P2，第 5 字节是 n（即 Le），如果从智能卡接收 256 字节，则第 5

字节为 0；cbSendLength 值为 5（4 字节头+1 字节 Le）。PbRecvBuffer 将接收智能卡返回的 n 字节，随后是 SW1、SW2 状态码；pcbRecvLength 的值在调用时至少为 n+2，返回后为 n+2。

```
BYTE        get_challenge[]={0x00, 0x84, 0x00, 0x00, 0x08};
sendSize=5;
recvSize=sizeof(recvBuffer);
lReturn = SCardTransmit(hCardHandle[0], SCARD_PCI_T0, get_challenge,
    sendSize, NULL, recvBuffer, &recvSize);
if ( lReturn != SCARD_S_SUCCESS )
{
    printf("Failed SCardTransmit\n");
    exit(1);
}
//返回的数据, recvSize=10
sw1=recvBuffer[recvSize-2];
sw2=recvBuffer[recvSize-1];
//data=recvBuffer[0]----recvBuffer[7]
```

（c）向智能卡发送没有数据交换的命令：应用程序既不向智能卡发送数据，也不从智能卡接收数据，pbSendBuffer 前 4 字节分别为 T=0 的 CLA、INS、P1、P2，不发送 P3；cbSendLength 值必须为 4。PbRecvBuffer 从智能卡接收 SW1、SW2 状态码；pcbRecvLength 值在调用时至少为 2，返回后为 2。

```
BYTE    set_flag[]={0x80, 0xFE, 0x00, 0x00};
sendSize=4;
recvSize=sizeof(recvBuffer);
lReturn = SCardTransmit(hCardHandle[0], SCARD_PCI_T0, set_flag, sendSize,
    NULL, recvBuffer, &recvSize);
if ( lReturn != SCARD_S_SUCCESS )
{
    printf("Failed SCardTransmit\n");
    exit(1);
}
//返回的数据，recvSize=2
sw1=recvBuffer[recvSize-2];
sw2=recvBuffer[recvSize-1];
```

（d）向智能卡发送具有双向数据交换的命令：T=0 协议中，应用程序不能同时向智能卡发送数据，并从智能卡接收数据，即发送到智能卡的指令中，不能同时有 Lc 和 Le。这只能分两步实现：向智能卡发送数据，接收智能卡返回的状态码，其中，SW2 是智能卡将要返回的数据字节数目；从智能卡接收数据（指令为 0x00、0xC0、0x00、0x00、Le）。

```
BYTE    get_response={0x00, 0xc0, 0x00, 0x00, 0x00};
BYTE    internal_auth[]={0x00, 0x88, 0x00, 0x00, 0x08, 0x01, 0x02, 0x03, 0x04,
0x05, 0x06, 0x07, 0x08};
sendSize=13;
recvSize=sizeof(recvBuffer);
```

```
lReturn = SCardTransmit(hCardHandle[0], SCARD_PCI_T0, internal_auth,
    sendSize, NULL, recvBuffer, &recvSize);
if ( lReturn != SCARD_S_SUCCESS )
{
    printf("Failed SCardTransmit\n");
    exit(1);
}
//返回的数据，recvSize=2
sw1=recvBuffer[recvSize-2];
sw2=recvBuffer[recvSize-1];
if ( sw1!=0x61 )
{
    printf("Failed Command\n");
    exit(1);
}
get_response[4]=sw2;
sendSize=5;
recvSize=sizeof(recvBuffer);
lReturn = SCardTransmit(hCardHandle[0], SCARD_PCI_T0, get_response,
    sendSize, NULL, recvBuffer, &recvSize);
if ( lReturn != SCARD_S_SUCCESS )
{
    printf("Failed SCardTransmit\n");
    exit(1);
}
//返回的数据，recvSize=10
sw1=recvBuffer[recvSize-2];
sw2=recvBuffer[recvSize-1];
//data=recvBuffer[0]----recvBuffer[7]
```

6.2.5 断开与读卡器（智能卡）的连接

在与智能卡的数据交换完成后，可以使用函数 ScardDisconnect()终止应用与智能卡之间的连接。

函数原型：LONG SCardDisconnect(SCARDHANDLE hCard, DWORD dwDisposition);

各个参数的含义：

（1）hCard：输入类型；与智能卡连接的句柄。

（2）dwDisposition：输入类型；断开连接时，对智能卡的操作，SCARD_LEAVE_CARD（不做任何操作）、SCARD_RESET_CARD（复位智能卡）、SCARD_UNPOWER_CARD（给智能卡掉电）、SCARD_EJECT_CARD（弹出智能卡）。

下面是断开与智能卡连接的代码：

```
lReturn = SCardDisconnect(hCardHandle[0], SCARD_LEAVE_CARD);
if ( lReturn != SCARD_S_SUCCESS )
{
```

```
        printf("Failed SCardDisconnect\n");
        exit(1);
    }
```

6.2.6 释放资源管理上下文

在应用程序终止前时，应该调用函数 ScardReleaseContext()释放资源管理器的上下文。

函数原型：LONG SCardReleaseContext(SCARDCONTEXT hContext);

各个参数含义：

（1）hContext：输入类型；ScardEstablishContext()建立的资源管理器上下文的句柄，不能为 NULL。

下面是释放资源管理上下文的代码：

```
lReturn = SCardReleaseContext(hSC);
if ( lReturn!=SCARD_S_SUCCESS )
    printf("Failed SCardReleaseContext\n");
```

6.3 接口程序中的PC/SC接口

在制卡接口和应用接口中，只有 Reader.cpp 和 JavaCommand.cpp 与 PC/SC 接口有关。

Reader.cpp 直接与 PC/SC 交互，它的主要函数有：

ReaderContext()用于建立上下文和取读卡器列表，如果没有读卡器，则返回错误；如果只有一个读卡器，则认为是一个双卡座读卡器；如果读卡起大于等于 2 个，则只取前 2 个读卡器，并认为它们是单卡座的。

ReaderConnect()用于建立与读卡器的连接。如果是两个读卡器，则分别建立连接。如果是一个双卡座读卡器，则分别与两个卡座建立连接，这时通过选择卡座并向每个卡座发送复位命令来实现的。

ReaderSelect(int reader)用于设置编号为 reader 的读卡器/座为当前读卡器，由于只有两个读卡器/卡座，reader 取值为 0 或 1。对于双卡座情况，通过 ScardControl()来选择当前卡座。

ReaderCommand(BYTE *command, DWORD length)用于将长度为 length 的命令 command 发送到读卡器上，记录结果状态和返回的数据到缓冲中，留待调用者以后使用。它只向 PC/SC 接口发送命令并接收响应，并不知道命令的类型，由它的调用者负责命令的生成和响应的处理。

JavaCommand.cpp 则负责形成各种命令 APDU，并根据命令的类型，对响应进行处理。它的代表性函数有：

JavaCmdWriteBinaryFile(int offset, int length, BYTE *data)需要向卡应用发送数据。它首先构造一条读数据命令到 bpCommandBuffer 中，然后通过 ReaderCommand()将命令发送到卡上，最后通过 ReaderCardState()接收缓冲中的状态。

JavaCmdReadBinaryFile(int offset, int length, BYTE *data)需要接收 Java 卡应用返回的数据。它首先构造一条写数据命令到 bpCommandBuffer 中，然后通过 ReaderCommand()将命令发送到卡上，最后通过 ReaderCardData()接收缓冲中的数据。

JavaCmdPersonlized()用于设置应用的个人化标志，与卡上应用没有数据交换。它首先

构造一条命令到 bpCommandBuffer 中，然后通过 ReaderCommand()将命令发送到卡上，最后通过 ReaderCardState()接收缓冲中的状态。

JavaCmdInternalAuthenticate(int keyType, int length, BYTE *dataIn, BYTE *dataOut)用于内部认证，即机具卡检验用户卡的合法性，它与卡上应用有双向的数据交换。它首先构造一条命令到 bpCommandBuffer 中;然后通过 ReaderCommand()将命令发送到卡上，由于采用 T=0 协议，不能同时发送 Lc 和 Le，只发送了 Lc，没有发送 Le；再通过 ReaderCardState()接收缓冲中的状态，判断返回数据的长度，最后通过 JavaCmdGetResponse()接收缓冲中的数据。

第 7 章　终端应用接口

7.1　接口概述

7.1.1　接口函数

终端应用接口的函数包括以下函数：

```
//定位机具卡和用户卡、对用户卡进行验证
//pPin是机具卡应用的密码
long ConnectCards(const char *pPin);

//选择应用并验证读写密钥
//pAid是ASCII串形式的应用标识符
//rwFlag是选择应用的方式：读或写
long SelectApplication(const char *pAid, short rwFlag);

//选择应用中的信息域
//pId是ASCII串形式的信息域标识符
long SelectRegion(const char *pId);

//读取应用中当前信息域若干字节的数据，读取的数据同卡内存储的方式相同
//nOffset是数据在信息域中偏移量
//nLen是要读取的数据字节数目
//pData存放读取到的数据
long ReadBinary(short nOffset, short nLen, char *pData);

//将若干字节的数据写入当前应用中当前信息域，写入的数据同卡内存储的方式相同
//nOffset是数据在信息域中偏移量
//nLen是要写入的数据字节数目
//pData存放要写入的数据
long UpdateBinary(short nOffset, short nLen, char *pData);

//读取当前应用中当前信息域的一条记录的数据，读取的数据同卡内存储的方式相同
//nNum是记录号
//nLen是记录长度
//pData存放读取到的数据
long ReadRecord(short nNum, short nLen, char *pData);

//将一条记录的数据写入当前应用中当前信息域，写入的数据同卡内存储的方式相同
//nNum是记录号
```

```
//nLen是记录长度
//pData存放要写入的数据
long UpdateRecord(short nNum, short nLen, char *pData);

//读取当前应用中当前信息域的一个字段，读取的数据要转换为ASCII串
//nOffset是数据在信息域中偏移量
//nLen是要写入的数据字节数目，按Java卡中的存储格式计算
//pData存放读取到的数据
//nFormat是字段在Java卡中的存储格式
long GetInfo(short nOffset, short nLen, char *pData, short nFormat);

//将一个字段写入当前应用中当前信息域，
//要写入的数据是ASCII串，应转换为卡上的存储格式
//nOffset是数据在信息域中偏移量
//nLen是要写入的数据字节数目，按Java卡中的存储格式计算
//pData存放写到的数据
//nFormat是字段在Java卡中的存储格式
long PutInfo(short nOffset, short nLen, char *pData, short nFormat);

//读取当前应用中当前信息域的一条记录的一个字段，读取的数据要转换为ASCII串
//nRecord是记录号
//nRecLen是记录长度
//nOffset是字段在记录中偏移量
//nLen是要读取的字段字节数目，按Java卡中的存储格式计算
//pData存放读取到的数据
//nFormat是字段在Java卡中的存储格式
long GetRecordInfo(short nRecord, short nRecLen, short nOffset, short nLen, char
*pData, short nFormat);

//将一个字段写入当前应用中当前信息域的一条记录中，
//要写入的数据是ASCII串，应转换为卡上的存储格式
//nRecord是记录号
//nRecLen是记录长度
//nOffset是字段在记录中偏移量
//nLen是要写入的字段字节数目，按Java卡中的存储格式计算
//pData存放写入的数据
//nFormat是字段在Java卡中的存储格式
long PutRecordInfo(short nRecord, short nRecLen, short nOffset, short nLen, char
*pData, short nFormat);

//验证用户卡密码
//pCardPin是卡密码
long VerifyPin(char *pCardPin);
```

```
//修改用户卡密码
//pOldPin是旧的卡密码
//pNewPins是新的卡密码
long ChangePin(char *pOldPin, char *pNewPin);

//重装用户卡密码
//pNewPin是新的卡密码
long ReLoadPin(char *pNewPin);

//返回接口错误状态
long GetAppError();

//释放读卡器
long ReleaseReader();
```

7.1.2 接口的使用方法

在使用接口时，首先调用 ConnectCards(const char *pPin)定位读卡器，参数是机具卡应用的密码。

随后调用SelectApplication(const char *pAid, short rwFlag)，选择卡上的应用，参数1是应用的AID，以ASCII串的形式给出，例如“415050303001”，参数2为读/写方式，表示以后对应用中数据的操作方式。再调用SelectRegion(const char *pId)选择要操作的信息域，参数是信息域表示，以ASCII串的形式给出，例如“05”。现在，可以调用读写函数读写信息域中的数据了。这个过程可反复进行。

在操作过程中，如果出现错误，可以调用GetAppError()，返回接口错误状态。

操作完毕后，调用ReleaseReader()释放读卡器。

接口函数提供了数据转换的功能。对于在Java卡上以二进制、BCD码表示的数值，允许终端应用以ASCII串的格式传入。在读取Java上的数据时，接口可以将以二进制、BCD码表示的数值，转换为ASCII串，传递给终端应用。

7.2 接口的设计与实现

应用接口的实现代码主要在javacard_app.cpp中。最主要的函数是ConnectCards(const char *pPin)和SelectApplication(const char *pAid, short rwFlag)。

应用接口中，存储了Java卡上的应用的AID（ASCII串格式），以及各个应用的密钥编码（二进制格式）。

应用接口总是假设系统中有两个单卡座读卡器，或一个双卡座读卡器，双卡座读卡器的两个卡座作为两个读卡器处理。两个读卡器编号为0、1。接口将命令发送到的读卡器称为当前读卡器。Reader.cpp中记录有当前读卡器的编号，并能改变当前读卡器的编号，这样，接口就能将命令发送到适当的读卡器上。在这两个读卡器中，插有机具卡的读卡器称为主读卡器majorReader，插有用户卡的读卡器称为次读卡器minorReader。

在ConnectReaders()中，首先建立读卡器上下文，连接读卡器；然后依次对各个读卡器进行操作，检查读卡器的卡上是否有机具卡应用（接口以ASCII字符串格式，存储了机具应

用的AID），如果某个读卡器上有机具卡应用，这个读卡器就是主读卡器，另一个读卡器为次读卡器，并记录主、次读卡器的编号，以后，接口就可以向机具卡和用户卡分别发命令。

接口中存储了Java卡的卡管理器的AID（ASCII字符串格式），据此判断用户卡的类型（Java卡或一般的CPU卡）。

ConnectReaders()在确定了用户卡的类型后，读出Applet00中的用户卡号，机具卡应用利用用户卡号，与应用根密钥分散，得到卡上应用的密钥。

随后，ConnectReaders()验证用户卡，由机具应用产生一个随机数，发送到用户卡上，Applet00将这个随机数加密后返回（即内部认证），机具应用也将随机数加密，比较两个加密的结果是否一致。

在SelectApplication()中，首先选择指定的应用，然后根据读写要求，与指定的应用验证权限，即外部认证。由Java卡应用产生一个随机数，返回给接口，接口发送给机具应用加密，然后返回给Java卡应用，Java卡应用也将随机数加密，比较两个加密结果是否一致。

其它接口函数就相对简单，一般是直接产生命令，发送到用户卡上。

第 8 章　制卡应用接口

8.1　Java卡Applet的生成

Java 卡 Applet 生成大致可分为以下几个步骤：

8.1.1　编辑Java卡Applet源代码

可以选用自己其喜爱的任何编辑工具，如 UltraEdit、VJ++、Visual Café、Eclipse 等，来编辑 Java 卡 Applet 源代码。

8.1.2　编译Java卡Applet源代码

编写好 Java 卡 Applet 源代码后，可以用 Java 编译器生成 class 文件。Java 卡 Applet 的编译过程与普通的 Java 程序编译过程没什么区别，编译器可以使用 Sun 公司的 Java 开发工具包（Java Development Kit, JDK）中的 javac.exe。

编译时，输入文件是 Java 卡 Applet 源代码(*.java)和一些 Java 卡的包 (package)，其中，Java 卡的包保存在 api21.jar 文件中，api21.jar 在 Java 卡开发包中。输出文件是 class 文件，位于 Java 卡 Applet 源代码(*.java)同一目录下。

编译命令行为：〈编译执行程序的路径和程序名〉〈可选参数〉〈Java 源文件〉

在编译 Java 卡 Applet 源代码时，只用“-g”参数，而不用“-O”参数，因为用“-g”参数时，可以在 class 文件中产生“LocalVariableTable”属性，转换程序(converter)要使用这个属性。如果同时使用了“-O”参数，就不会产生“LocalVariableTable”属性。这样转换时就会出错。

编译命令的例子：

```
set JC21_HOME=C:\java_card_kit-2_1_2
set JAVA_HOME=c:\jdk1.3

c:\jdk1.3\bin\javac -g -classpath c:\java_card_kit-2_1_2\lib\api21.jar;c:\javacard\applets Services\*.java
@call c:\java_card_kit-2_1_2\bin\converter -config Services.opt
copy Services\javacard\Services.cap c:\javacard\applets\cap\app\Services.cap
@call C:\java_card_kit-2_1_2\bin\scriptgen -o Services.scr Services\javacard\Services.cap

c:\jdk1.3\bin\javac -g -classpath c:\java_card_kit-2_1_2\lib\api21.jar;c:\javacard\applets Applet00\*.java
@call c:\java_card_kit-2_1_2\bin\converter -config Applet00.opt
copy Applet00\javacard\Applet00.cap c:\javacard\applets\cap\app
@call C:\java_card_kit-2_1_2\bin\scriptgen -o Applet00.scr Applet00\javacard\Applet00.cap
```

如上命令行中的 c:\jdk1.3\bin\javac 部分就完成了 Services.java 和 Applet00.java 的编译，同时在编译时用到了 api21.jar 中的一些 class 文件。

8.1.3 生成Cap(Converted APplet) 文件

Java 卡的运行环境并不能识别 Java 的 class 文件，因此，必须将包含 Applet 的 class 文件转换成 Cap 文件，Cap 文件就是可以被装载到 Java 卡上的 Applet。转换工作由 Java 卡转换器(converter)完成。

转换器(converter) 是由 Java 卡开发包（Java Card Development Kit）提供的字节代码工具。它将 class 文件转换成一些输出文件。转换时，输入文件是：由编译器生成的 class 文件；输出文件是：Cap 文件、Export 文件、JCA 文件，它们的后缀分别是：*.cap、*.exp、 *.jca，文件名与输入文件一致，它们位于 Java 卡 Applet 源代码目录下的一个叫 Javacard 的子目录中。

Java 卡开发包提供了一个批处理文件：converter.bat，进行文件转换。命令行的一些可选参数如下：

-classdir： 项目的根目录

-exportpath： 一些转换时要用到的 Exp 文件的父目录，

-d：输出的路径，它指明的是根目录

-applet [AID][classname]:指明缺省 Applet 的 AID,和含 Install() 方法的 class 文件名

-out [CAP][EXP][JCA]：说明要转换器生成什么文件，一般默认为生成 CAP 和 EXP 文件

-nobanner ：信息使用标准输出

包(package) 名：要被转换的包名

包 AID：5 到 16 十进制，十六进制或八进制数，表明 Applet 的 AID

版本：用户自定义的版本号

命令行的一些可选参数可以放在一个后缀为 opt 的文件中。

命令行的例子如下：

```
@call c:\java_card_kit-2_1_2\bin\converter -config Services.opt
copy Services\javacard\Services.cap c:\javacard\applets\cap\app\Services.cap
```

文件 Services.opt 的内容如下（#后面是注释，不是文件的内容）：

```
-classdir    C:\javacard\applets   #项目的根目录
-exportpath C:\java_card_kit-2_1_2\api21_export_files\;c:\javacard\applets   #转换时要用到的 Exp 文件的父目录
-out         CAP EXP #要转换器生成 CAP 和 EXP 文件
app.Services #包名称
0x41:0x50:0x50:0x53:0x56:0x43 #包的 AID
1.0 #版本
```

文件 Applet00.opt 的内容如下（#后面是注释，不是文件的内容）：

```
-classdir    C:\javacard\applets   #项目的根目录
-exportpath C:\java_card_kit-2_1_2\api21_export_files\;c:\javacard\applets #转换
```

时要用到的 Exp 文件的父目录
-out　　CAP EXP　#要转换器生成 CAP 和 EXP 文件
-applet　　0x41:0x50:0x50:0x30:0x30:0x01 app.Applet00.Applet00 #Applet 的 AID 和名称
app.Applet00　#包名称
0x41:0x50:0x50:0x30:0x30　#包的 AID
1.0　#版本

8.1.4　产生脚本文件

生成的 cap 文件不能直接下载，需要使用 Java 卡开发包提供的脚本生成器 scriptgen，将 cap 文件转换为 scr 文件。转换命令行的例子如下：

@call　C:\java_card_kit-2_1_2\bin\scriptgen　-o　Services.scr Services\javacard\Services.cap

8.2　Applet的下载和安装指令的生成

当 Applet 的 scr 文件生成后，我们就可以进行 Applet 的下载和安装。在这一过程中，需要使用 PC 机，读卡器和控制读卡器的软件。下载软件以 scr 文件为输入，形成下载指令，将 scr 文件装载到 Java 卡上。本小节的具体实现在 javaload.cpp 中。

8.2.1　转换scr文件

Scr 文件是文本文件，包含了下载 Applet 的指令，但这些指令不是 OpenPlatform 的指令，我们需要自行开发软件，以 scr 文件为输入，提取其中的数据，生成 OpenPlatform 的下载指令，发向 Java 卡的 CardManager。转换 scr 文件的具体方法如下：

逐行读入 scr 文件，提取以 0x80、0xB4 开始的有效行，有效行的例子如下："0x80 0xB4 0x01 0x00 0x14 0x01 0x00 0x11 0xDE 0xCA 0xFF 0xED 0x01 0x02 0x02 0x00 0x01 0x07 0xB7 0xFE 0xCE 0xF1 0x00 0x00 0x00 0x7F;"，行中第 5 个字节 0x14 是本行（有效）数据的字节数目（本行有 20 个有效字节），最后一个字节是 0x7F，这个字节没有用处，不在有效字节中。

提取行中的有效字节，得到 0x01 0x00 0x11 0xDE 0xCA 0xFF 0xED 0x01 0x02 0x02 0x00 0x01 0x07 0xB7 0xFE 0xCE 0xF1 0x00 0x00 0x00。将有效字节转换为二进制形式，得到 20 个字节。这 20 个字节可以作为下载指令的数据。

将整个 scr 文件的全部有效字节安装在文件中的顺序，保存到一个字节数组中后，再分成若干块进行下载，每块的字节数目最多可为 255 字节。

转换的程序在JavaLoad.cpp中。

8.2.2　下载Applet

下载时，首先指定 Applet 的包标识 PID(Package IDentifier)，指令为：0x80/0x84、0xE6、0x02、0x00、Lc、Data，其中 0x02 表示 Load 下载，Lc 为 Applet 的 PID 字节数目，

Data 为 Applet 的 PID。

实际下载 Applet 的指令为：0x80/0x84、0xE8、P1、块号、Lc、Data，其中，P1=0x00，表示是第一块，P1=0x80 表示是最后一块；块号从 0 开始，最大块号为 255；Lc 是数据的字节数目，Data 是数据。

下载的第一条指令比较特殊，需要在实际的 Applet 前，插入一个长度指示。如果第一块的数据是 250 字节，根据 scr 文件有效字节的数目不同，Lc 可能为 252、253、254。生成第一条下载指令的规则为：

（1）如果 scr 文件有效长度小于 128，则第一条下载指令为：0x80/0x84、0xE8、P1、0x00、Data 长度+2、0xC4、scr 文件有效长度、Data；

（2）如果 scr 文件有效长度大于等于 128 并且小于 256，则第一条下载指令为：0x80/0x84、0xE8、P1、0x00、Data 长度+3、0xC4、0x81、scr 文件有效长度、Data；

（3）如果 scr 文件有效长度大于等于 25632768，则第一条下载指令为：0x80/0x84、0xE8、P1、0x00、Data 长度+4、0xC4、0x81、scr 文件有效长度高 8 位、scr 文件有效长度低 8 位、Data。

最后一条下载指令的 P1=0x80，Lc 是最后一块的实际长度。

8.2.3 安装Applet

安装（实例化）Applet 的指令为：指令为：0x80/0x84、0xE6、0x0C、0x00、Lc、Data，其中，0x0C 表示 Install 安装，并设置 Applet 状态为可选择的（selectable）；Lc 为 Data 的长度；缺省安装时，Data 的格式为：Applet 的 PID 字节数目、Applet 的 PID、Applet 的 AID(Applet IDentifier)字节数目、Applet 的 AID、Applet 实例的 AID 字节数目、Applet 实例的 AID（可以与 Applet 的 AID 相同）、0x01、0x00、0x02、0xC9、0x00、0x00。

8.3 制卡接口的设计与实现

8.3.1 接口函数

制卡接口包括下列函数：

```
//删除Java卡上的Applet包或实例
//batchAID是发行管理应用的标识，ASCII串。
//batchKey是卡管理器的3个密钥，ASCII串，无间隔。
//cardManagerAID是卡管理器的标识，ASCII串。
//objects是要删除的对象数目
//ids是要删除的对象的标识的集合，ASCII串，用';'分隔，
//     Applet实例的AID应该放在Applet包(CAP)标识的前面。
//flag是与卡管理器的建立通道类型
long DeleteObjects(char *batchAID, char *batchKey, char *cardManagerAID, short
objects, char *ids, short flag);

//向Java卡下载Applet，并安装Applet实例
```

```
//batchAID是发行管理应用的标识，ASCII串。
//batchKey是卡管理器的3个密钥，ASCII串，无间隔。
//cardManagerAID是卡管理器的标识，ASCII串。
//files是CAP文件（Applet包）的数目。
//ids是要下载的对象的标识的集合，ASCII串，用';'分隔Applet包，
//     用','分隔Applet包内的包AID和实例AID，
//     Applet实例的AID应该放在Applet包(CAP)标识的后面。
//flag是与卡管理器的建立通道类型
long LoadApplets(char *batchAID, char *batchKey, char *cardManagerAID, short files,
char *ids, short flag);

//列出Java卡上的CAP包或Applet实例
//batchAID是发行管理应用的标识，ASCII串。
//batchKey是卡管理器的3个密钥，ASCII串，无间隔。
//cardManagerAID是卡管理器的标识，ASCII串。
//ids保存返回的AID标识，标识之间用“;”隔离。
//     AID都是ASCII串。Applet的AID放在所有包的AID的前面。
//flag是与卡管理器的建立通道类型
long ListObjects(char *batchAID, char *batchKey, char *cardManagerAID, char *ids,
short flag);

//向发行卡写数据
//masterAID是根密钥卡应用的AID，ASCII串
//masterPIN是根密钥卡密码，ASCII串
//issueAID是发行卡应用的AID，ASCII串
//issuePIN是发行卡应用的密码，ASCII串
//keyIndex是应用根密钥的编号，ASCII串，编号之间之间用';'隔开
//keyCount是应用根密钥的数目
//keyLength是应用根密钥的长度
long IssueCardData(char *masterAID, char *masterPIN, char *issueAID, char *issuePIN,
char *keyIndex, short keyCount, short keyLength);

//向机具卡写数据
//masterAID是根密钥卡应用的AID，ASCII串
//masterPIN是根密钥卡应用的密码，ASCII串
//deviceAID是机具卡应用的AID，ASCII串
//devicePIN是机具卡应用的密码，ASCII串
//keyIndex是应用根密钥的编号，ASCII串，编号之间之间用';'隔开
//keyCount是应用根密钥的数目
//keyLength是应用根密钥的长度
long DeviceCardData(char *masterAID, char *masterPIN, char *deviceAID, char
*devicePIN, char *keyIndex, short keyCount, short keyLength);

//向用户卡写数据
```

```
//issueAID是发行卡应用的AID，ASCII串
//issuePIN发行卡应用密码，ASCII串
//userAID：用户卡应用的AID，ASCII串
//keyIndex是应用密钥的编号，ASCII串，编号之间之间用';'隔开
//keyCount是应用密钥的数目
//keyLength是应用密钥的长度
//sn是用应用根密钥计算应用密钥的分散参数，16字节卡号
//snLength是分散参数的长度（十六进制数组字节数目）
//files是数据文件的文件数目
//lengths是各个文件数据长度，ASCII串，长度之间用';'隔开
//data是各个数据文件的数据，没有间隔
//flag是是否进行个人化的标志
long UserCardData(char *issueAID, char *issuePIN, char *userAID, char *keyIndex,
short keyCount, short keyLength, char *sn, short snLength, short files, char
*lengths, char *data, short flag);

//返回接口程序状态
long GetAppError();

//释放读卡器
long ReleaseReader();
```

8.3.2 接口的使用方法

如果要向用户卡下载安装 Applet，可以直接调用 LoadApplets(char *batchAID, char *batchKey, char *cardManagerAID, short files, char *ids, short flag)，然后调用 UserCardData(char *issueAID, char *issuePIN, char *userAID, char *keyIndex, short keyCount, short keyLength, char *sn, short snLength, short files, char *lengths, char *data, short flag)向应用加载数据。在操作过程中，如果出现错误，可以调用 GetAppError()，返回接口错误状态。操作完毕后，调用 ReleaseReader()释放读卡器。

如果要向机具卡下载安装 Applet，可以直接调用 LoadApplets(char *batchAID, char *batchKey, char *cardManagerAID, short files, char *ids, short flag)，然后调用 DeviceCardData(char *masterAID, char *masterPIN, char *deviceAID, char *devicePIN, char *keyIndex, short keyCount, short keyLength)向机具应用加载密钥。在操作过程中，如果出现错误，可以调用 GetAppError()，返回接口错误状态。操作完毕后，调用 ReleaseReader()释放读卡器。

如果要向发行卡下载安装 Applet，可以直接调用 LoadApplets(char *batchAID, char *batchKey, char *cardManagerAID, short files, char *ids, short flag)，然后调用 IssueCardData(char *masterAID, char *masterPIN, char *issueAID, char *issuePIN, char *keyIndex, short keyCount, short keyLength)向应用根密钥应用加载密钥。在操作过程中，如果出现错误，可以调用 GetAppError()，返回接口错误状态。操作完毕后，调用 ReleaseReader()释放读卡器。

LoadApplets()中参数的含义：

batchAID是发行管理应用的标识，ASCII串，例如“4B4559424301”或“4B4559424302”。

batchKey是卡制造厂商提供的卡管理器的3个初始密钥，ASCII串，例如，“404142434445464748494A4B4C4D4E4F”，密钥之间无间隔，一个密钥十六进制16字节，或者32个ASCII字符，3个密钥共十六进制48字节，或96个ASCII字符，如果只有32个ASCII字符，则表示3个密钥的值相同。这个参数仅在发行制卡阶段有用，发行后制卡时，可以为任意字符串。

cardManagerAID是卡管理器的标识，ASCII串，例如“A0000000030000”，或“A000000003000000”。

files是CAP文件（Applet包）的数目，例如4。

ids是下载的对象的标识的集合，ASCII串，用';'分隔Applet包，用','分隔Applet包内的包AID和实例AID，Applet实例的AID应该放在Applet包(CAP)标识的后面，例如，“services.cap,415050535643;applet00,41505030300,415050303001;applet01.cap,4150503031,415050303101;applet02.cap,4150503032,415050303201”。这里，第一个包是公共服务包，因而没有Applet实例。

flag是与卡管理器的建立通道类型。通道类型有3种，加上区分发行制卡和发行后制卡，通道有5种。发行制卡时，可以为：0，一般通道；2，MAC通道；4，加密MAC通道。发行后制卡时，可以为：1，MAC通道；3，加密MAC通道。

UserCardData()中参数的含义：

issueAID是发行卡应用的AID，ASCII串，例如“4B4559494301”。

issuePIN发行卡应用密码，ASCII串，例如“123456”。

userAID是用户卡应用的AID，ASCII串， 例如“415050303001”。

keyIndex是应用密钥的编号，ASCII串，编号之间之间用';'隔开，例如“0111;0112;0113”，表示需要下载3个密钥到applet01。

keyCount是应用密钥的数目，例如，3。

keyLength是应用密钥的长度，目前，均为16。

sn是用应用根密钥计算应用密钥的分散参数，这是16字节卡号，即16字节的ASCII串（卡号本身就是ASCII字符）。

snLength是分散参数的长度，目前是16。

files是应用的信息域/数据文件数目，例如，Applet01有4个信息域，就为4。

lengths是各个信息域的长度，ASCII串，长度之间用';'隔开，例如，对于Applet01，为“256;128;1024;1024”。

data是各个信息域的数据，没有间隔，为二进制格式。

flag是是否进行个人化的标志，目前是1。

DeviceCardData()和 IssueCardData()中参数的含义：

masterAID是根密钥卡应用的AID，ASCII串，例如“4B45594D4301”。

masterPIN是根密钥卡应用的密码，ASCII串，例如“123456”。

deviceAID/issueAID是机具卡/发行卡应用的AID，ASCII串，例如“4B4559444301”/“4B4559494301”。

devicePIN/issuePIN是机具卡/发行卡应用的密码，ASCII串，例如“123456”

keyIndex是应用根密钥的编号，ASCII串，编号之间之间用';'隔开。例如，“0011;0012;0013;0111;0112;0113;0211;0212;0213;0214”表示下载全部应用的根密钥。

keyCount是应用根密钥的数目。

keyLength是应用根密钥的长度，目前为16。

8.3.3 制卡接口的实现

制卡接口的实现代码主要在javacard_make.cpp中。

制卡接口总是假设系统中有两个单卡座读卡器，或一个双卡座读卡器，双卡座读卡器的两个卡座作为两个读卡器处理。两个读卡器编号为0、1。接口将命令发送到的读卡器称为当前读卡器。Reader.cpp中记录有当前读卡器的编号，并能改变当前读卡器的编号，这样，接口就能将命令发送到适当的读卡器上。在制作机具卡或发行卡时，插有根密钥卡的读卡器称为主读卡器majorReader，另一个读卡器称为次读卡器minorReader。在制作用户卡时，插有发行卡的读卡器称为主读卡器majorReader，另一个读卡器称为次读卡器minorReader。

在 LoadApplets()中，首先根据 flag，建立与用户卡的卡管理器之间的通道，通道的建立过程参见第 4 章。然后，从 ids 中分离出 Applet 包的文件名，和 Apllet 包的 AID，进行下载，如果还能分离出 Applet 实例的 AID，就进行安装。下载安装完后，如果有必要，就改变用户卡的卡管理器的密钥集合，并将用户卡的卡管理器切换到 SECURED 状态。

在 DeleteObjects()中，首先根据 flag，建立与用户卡的卡管理器之间的通道，通道的建立过程参见第 4 章。然后，从 ids 中分离出 Applet 实例的 AID 和 Apllet 包的 AID，进行删除。

在 ListObjects()中，首先根据 flag，建立与用户卡的卡管理器之间的通道，通道的建立过程参见第 4 章。然后，分别取用户卡上的剩余空间、卡管理器的状态、Applet 包的 AID、Applet 实例的 AID。

在以上过程中，如果不是一般通道，发送到用户卡卡管理器的命令，需要先由发行管理应用计算 MAC,可能还要将命令中的数据加密,在发送到用户卡上。这由 WriteToChildCard()来实现。

向 Java 卡应用加载数据或密钥时，直接与卡应用交互，不需要卡管理器的参与。密钥的分散由根密钥卡或发行卡应用完成。