JavaCard 应用程序开发三部曲

来源：天极网

内容简介：

Java Card 技术适用于智能卡和其他高度专业化设备的 Java 平台，这些设备的内存和处理能力都比 J2ME 设备的要求更加苛刻。

智能卡在个人信息安全方面有很大用处。它们可用于添加验证和安全访问到需要高级别安全的信息系统。保存在智能卡上的信息是便携的。使用 Java Card 技术，你可能随身携带保存在一个小型并且安全的媒介上的重要并且敏感的个人信息，比如你的病历、信用卡号或者电子现金余额。

基础篇

1、什么是智能卡？

智能卡不是新鲜事物。它们在二十年前在欧洲就以记忆卡片的形式推出了，用于保存关键的电话信息，以减少盗打付费电话的可能。

智能卡技术是 ISO 国际标准组织的连接技术委员会 1(JTC1)和国际电子委员会(IEC)定义并控制的一种行业标准。1987 年推出的 ISO/IEC 7816 国际标准系列在 2003 年推出了它的最新的升级版本，定义了智能卡的各个方面，包括物理特征、物理接触界面、电子信号和传输协议、命令、安全体系、应用程序标识符和公用数据元素等。

智能卡是一个包含嵌入集成电路(IC)的塑料卡片，类似于一张信用卡。当用作 SIM 卡时，这个塑料卡片很小，正好能放入手机中。智能卡设计时就极注重高度安全性，窜改一点点内容都会导致毁坏它包含的信息。

在智能卡使用的某些领域，它们只是仅仅提供受保护的非易失性存储。更高级的智能卡还有微处理器和内存，用于安全的处理和储存，并且可以用于使用公共密钥或者共享密钥算法的安全应用程序。智能卡上的非易失性存储是最宝贵的资源，可用于保存秘钥和数字证书。一些智能卡有单独的加密协处理器，支持象 RSA、AEC 和(3)DES 这样的算法。

智能卡不包含电池，只有在和读卡机连接的时候才被激活。当它被连接时，在执行一个复位序列之后，卡片处于非激活状态，等待接收来自客户端（主机）应用程序的命令请求。

智能卡可以分为可接触和非可接触。可接触智能卡通过读卡器和智能卡的 8 个触点物理接触来通讯并工作，而非可接触智能卡依靠在小于 2 英尺的一般距离之内的射频信号通讯。非接触智能卡的射频通信基于类似于用于保存反盗窃和记录清单的射频标识符(RFID)标记的技术。图 1 描述了可接触和非可接触智能卡：

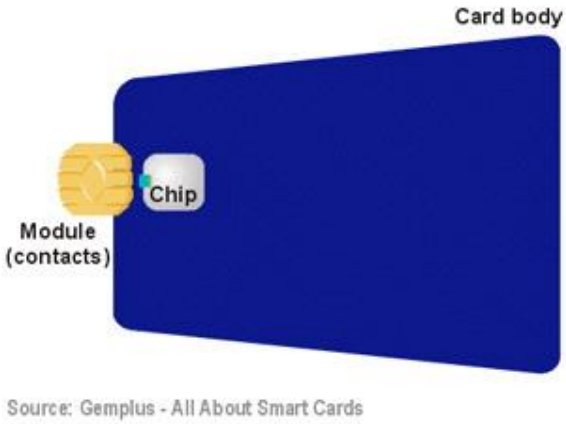


图 1a.接触式智能卡

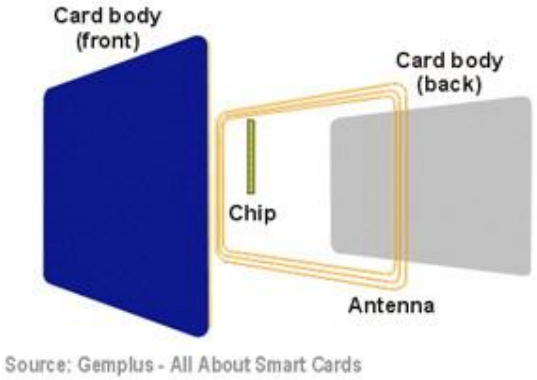


图 1b.非接触式智能卡

Java Card 技术还存在除了智能卡之外的其它的形态，例如智能按钮和 USB 令牌，这两种如图 2 所示。这些的功能和智能卡差不多，例如用于验证用户或者传送敏感信息。智能按钮包含一块电池而且是基于可接触模式，而 USB 令牌则可以直接插入个人计算机的 USB 端口，而不需要任何可接触或者非可接触读卡器。这两种类型的 Java Card 具有与智能卡相同的编程能力并且具有防篡改能力。

图 2a. 带有 Java 功能的智能纽扣

图 2b. 带有 Java 功能的 USB 令牌

请参阅 What is a Smart Card? http://java.sun.com/products/javacard/smartcards.htm 获取更详细的信息。

2、JavaCard 规范

多年以前，Sun 微系统公司实现了智能卡和类似的资源约束设备的潜能，并且定义了一组 Java 技术子集规范来为它们创建应用程序，Java Card 小应用程序。支持这些规范的设备称为 Java Card 平台。在一个 Java Card 平台上，来自不同的供应商的多个应用程序可以安全地共存。

一个典型的 Java Card 设备有一个 8 或 16 位的运行在 3.7MHz 的中央处理器，带有 1K 的 RAM 和多于 16K 的非易失性存储器(可编程只读存储器或者闪存）。高性能的智能卡带有单独的处理器和加密芯片，以及用于加密的内存，并且有一些还带有 32 位的中央处理器。

Java Card 技术规范目前是 2.2 版，由三部分组成：

- Java Card 虚拟机规范，定义了用于智能卡的 Java 程序语言的一个子集和虚拟机。
- Java Card 运行时环境规范，进一步定义了用于基于 Java 的智能卡的运行期行为。
- Java Card 应用编程接口规范，定义了用于智能卡应用程序核心框架和扩展 Java 程序包和

类。

Sun 还提供了 Java Card 开发工具箱(JCDK) http://java.sun.com/products/javacard/，包含了 Java Card 运行期环境和 Java Card 虚拟机的引用实现，和其它帮助开发 Java Card 小应用程序的工具。本文的第二部分将详细讲述 JCDK。

Java Card 技术和 J2ME 平台

让我们比较一下 Java Card 和 J2ME 平台技术：

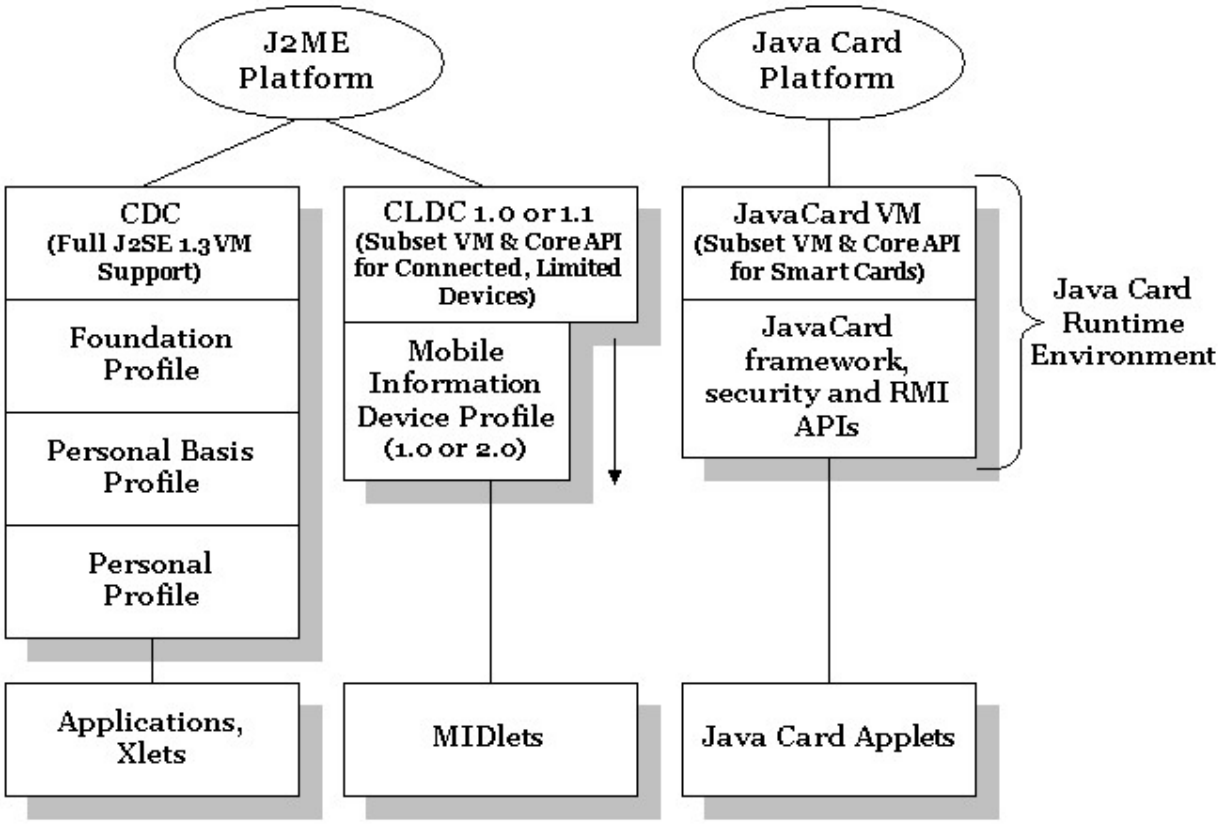


图. Java Card 技术和 J2ME 平台

CDC 和 CLDC 配置以及它们相关的简表是 J2ME 平台的一部分，而 Java Card 是一个单独创建来用于智能卡环境的平台。

3、元素

完整的 Java Card 应用程序由一个后端应用程序和系统、一个主机（卡外）应用程序、一个接口设备（读卡器）和卡上小应用程序、用户证书和支持软件组成。所有的这些元素共同组成一个安全的端到端应用程序：

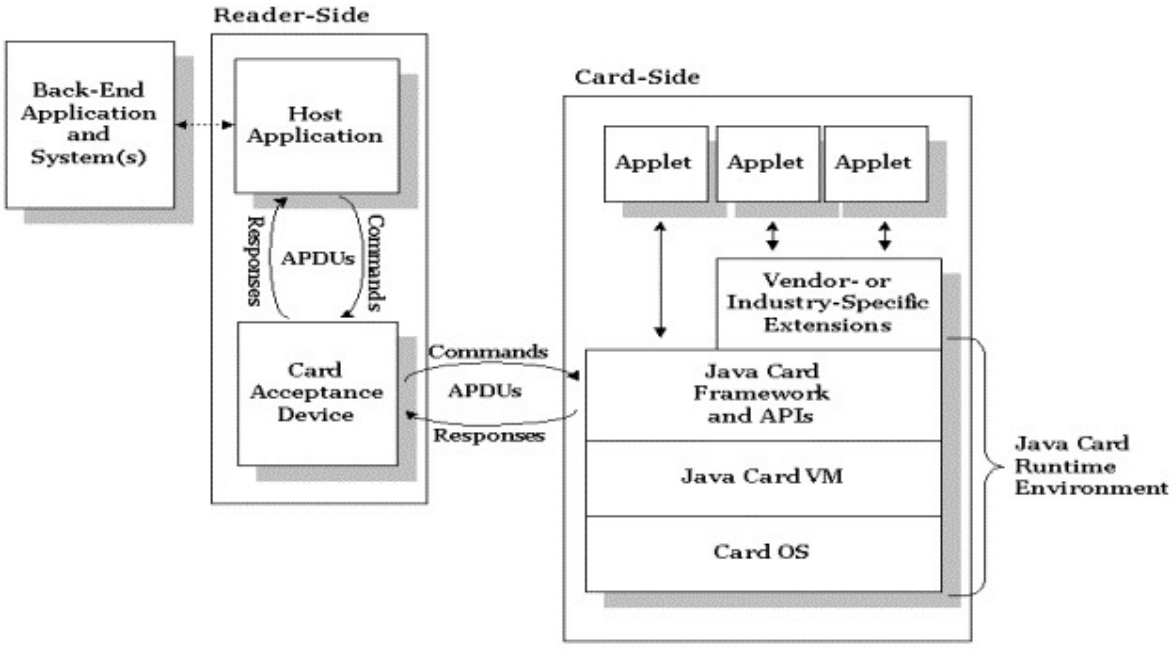


图 1. Java Card 应用程序的体系结构

一个典型的 Java Card 应用程序不是孤立的，而是包含卡端、读取端和后端元素。让我们更详细的讲述一下每个元素。

后端应用程序和系统

后端应用程序提供了支持卡上 Java 小应用程序的服务。 例如，一个后端应用程序可以提供到安全系统和卡上的证书的连接，提供强大的安全性。在一个电子付款系统中，后端应用程序可以提供到信用卡及其他付款信息的访问。

读取端主应用程序

主应用程序存在于一个例如个人计算机这样的台式机或者终端、电子付款终端、手机或者一个安全子系统中。

主应用程序处理用户、Java Card 小应用程序和供应商的后端应用程序之间的通讯。

传统的读取端应用程序是使用 C 编写的。近来 J2ME 技术的广泛普及有望使用 Java 实现主应用程序；例如，它可以在一台支持 MIDP 和安全信赖服务应用编程接口（Security and Trust Services API）手机上运行。

智能卡供应商一般不仅提供开发工具箱，而且提供支持读取端应用程序和 Java Card 小应用程序的应用程序编程接口。例如 OpenCard Framework http://www.opencard.org/，就是一个基于 Java 的应用程序编程接口集，隐藏了来自不同供应商的读取器的一些细节，并且提供了 Java Card 远程方法调用分布式对象模型和安全信任服务应用编程接口（SATSA），我在本文后面一部分讨论它们。

读取端卡片接受设备

卡片接受设备（CAD）是处于主应用程序和 Java Card 设备之间的接口设备。一个 CAD 为卡片提供电力，以及与之进行电子或者射频通信。一个 CAD 可能是一个使用串行端口附于台式计算机的读卡器，或者可能被整合到终端内，例如饭店或者加油站内的电子付款终端。接口设备从主应用程序到卡片转送应用程序协议数据单元(Application Protocol Data Unit，简称 APDU）命令（在后面讨论），并且从卡片向主应用程序转送响应。一些 CAD 有用于输入个人识别号码的键盘，有的可能还有显示屏。

卡片端小应用程序和环境

Java Card 平台是一个多应用程序环境。在图 4 中我们可以看到，卡片上可能存在一个或多个 Java Card 小应用程序，还有支持软件--卡片的操作系统和 Java Card 运行时环境（JCRE）一起。JCRE 由 Java Card 虚拟机、Java Card Framework 和应用程序编程接口以及一些扩展应用程序编程接口组成。

所有的Java Card小应用程序扩展Applet基本类，并且必须实现install()和process()方法；JCRE在安装小应用程序的时候调用 install()，并且在每次有一个进入的用于小应用程序的 APDU 的时候调用 process()。

Java Card 小应用程序在被装载的时候实例化，并且在断电的时候保持运行。Java Card 小应用程序起一个服务器的作用，并且是无源的。在一张卡片被加电以后，每个小应用程序都保持非运行的状态直到它被选择，在此时可能会做初始化。小应用程序只有在一个 APDU 被发送给它以后才被激活。一个小应用程序如何激活（被选择）在"一个 Java Card 小应用程序的生命周期"一节中描述。

与 Java Card 小应用程序通讯（访问智能卡）

你可以使用两种模型中的任何一种来在一个主应用程序和一个 Java Card 小应用程序之间通信。第一个模型是基本消息传送模型，第二种是基于 Java Card 远程方法调用（JCRMI），这是 J2SE RMI 分布式对象模型的一个子集。此外，SATSA 通过一个基于更加抽象的应用编程接口的普通连接框架（Generic Connection Framework，简称 GCF）应用编程接口，让你要么使用消息传递要么使用 JCRMI 来访问智能卡。

4、消息传递模型

图 1 中说明的消息传递模型是所有 Java Card 通信的基础。它的核心就是应用程序协议数据单元（APDU），CAD 和 Java Card 框架之间交换的一个逻辑数据包。JavaCard 框架接收任何 CAD 发送进来的 APDU 命令并且传送到相应的小应用程序中。小应用程序处理 APDU 命令，然后返回一个响应 APDU。那些 APDU 遵守国际标准规格 ISO/IEC 7816 - 3 和 7816 - 4。

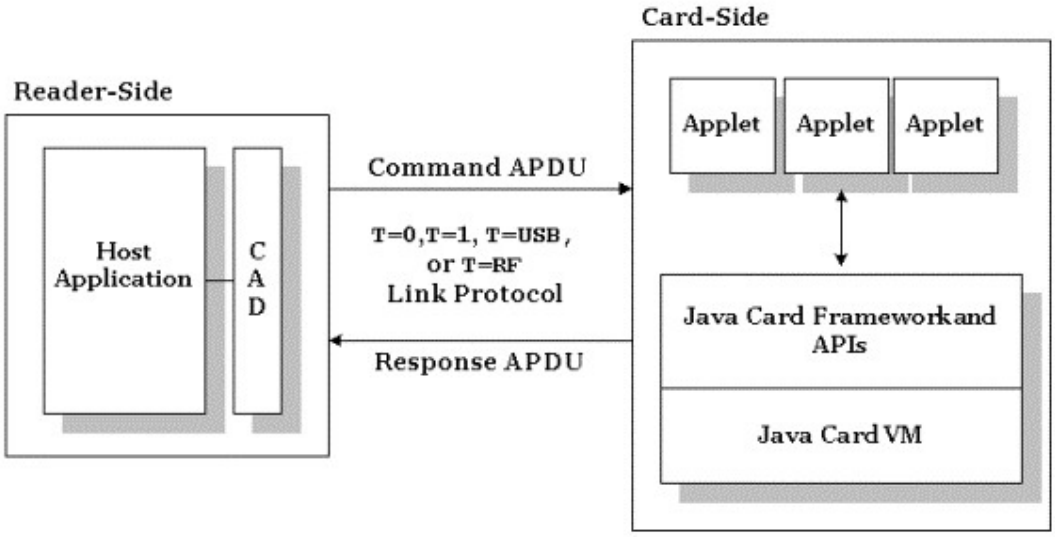


图 1 使用消息传递模型通讯

读卡器和卡之间的通信通常基于下面两种连接协议的一种，面向字节的 T = 0，或者面向数据块的 T = 1。还可能会用到被称为 T = USB 和 T = RF 的替换协议。JCRE APDU 类向应用程序隐藏了一些协议细节，但不是全部，因为 T = 0 协议相当的复杂。

1.APDU 命令

一个 APDU 命令的结构由它的第一个字节的值控制，大部分情况下看上去如下所示：

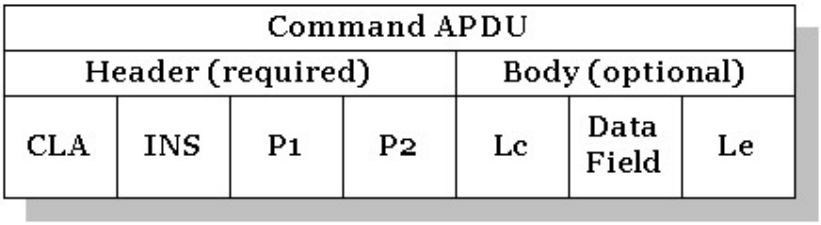


图 2、APDU 命令

一个 APDU 命令有一个必须有的头和一个可选的体，包含：

• CLA（1 字节）：这个必要的字段识别指令的一个特定应用程序类。有效的 CLA 值在 ISO 7816

- 4 规范中定义：

表格 1、ISO 7816 CLA 值

CLA 值	指令类
0x0n, 0x1n	ISO 7816 - 4 卡指令，比如文件存取和安全操作
20 to 0x7F	保留
0x8n or 0x9n	你可以用作你的特定的应用程序指令的 ISO/IEC 7816 - 4 格式，根据标准解释' X '
0xAn	特定的应用程序或者供应商的指令
B0 to CF	你可以用作特定应用程序的 ISO/IEC 7816 - 4 格式
D0 to FE	特定的应用程序或者供应商的指令
FF	保留给协议类型选择

• 理论上，你可以使用所有的 CLA 值 0x80 或者更高值来用于特定应用程序指令，但是在许多现在的 Java Card 实现中，只有黑体显示的是实际认可的。

• INS（1 字节）：这个必需的字段指明 CLA 字段中标示的指令类中的一个特定指令。ISO 7816 - 4 标准指定用于访问卡上的数据的基本指令，当它根据在像标准中定义的卡上的文件系统那样结构化的时候。附加功能已经在这个标准中的其它地方说明，其中一些是安全功能。表 2 中是一个 ISO 7816 指令的列表。只有当使用一个相应的 CLA 字节值时，你才可以根据标准定义你自己的特定应用程序的 INS 值，。

表格 2、当 CLA = 0x 时的 ISO 7816 - 4 INS 值

INS 值	命令描述
0E	Erase Binary
20	Verify

70	Manage Channel
82	External Authenticate
84	Get Challenge
88	Internal Authenticate
A4	Select File
B0	Read Binary
B2	Read Record(s)
C0	Get Response
C2	Envelope
CA	Get Data
D0	Write Binary
D2	Write Record
D6	Update Binary
DA	Put Data
DC	Update Record
E2	Append Record

- P1（1 字节）：这个必需的字段定义指令参数 1。你可以使用这个字段来检验 INS 字段，或者用于输入数据。
- P2（1 字节）：这个必需的字段定义指令参数 2.你可以使用这个字段来检验 INS 字段，或者用于输入数据。
- Lc（1 字节）：这个可选的字段是命令的数据字段的字节数。
- 数据字段（可变的，字节 Lc 数）：这个可选的字段保存命令数据。

• Le（1 字节）：这个可选的字段指定在期望响应的数据字段中的极限字节数。

取决于命令数据的存在与否以及相应是否必须，命令 APDU 有四种变化。只有在你使用协议 T = 0 时，你才需要关心这些变化：

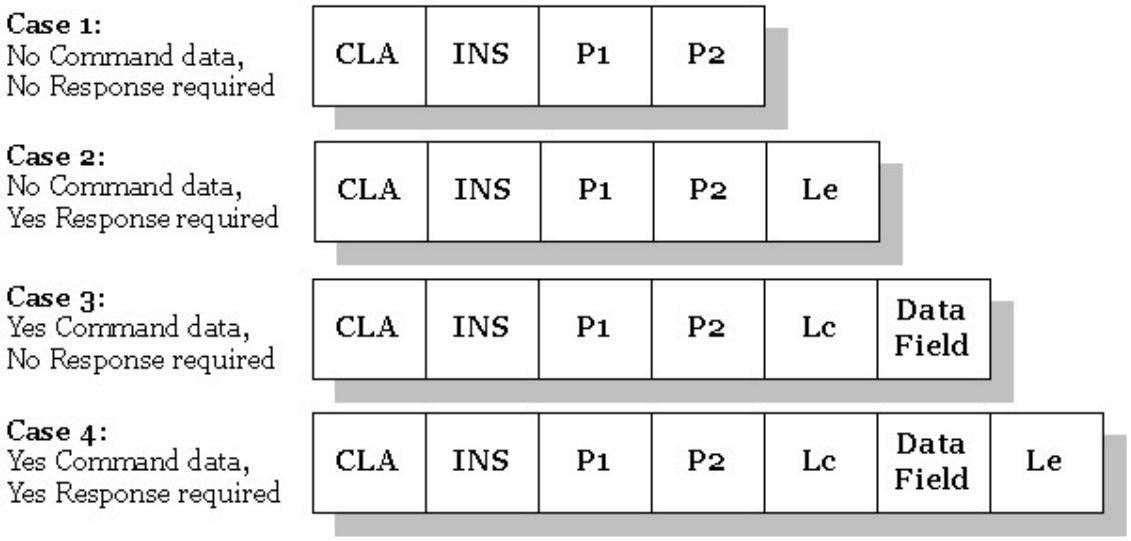


图 3、APDU 命令的四个可能的结构

一个典型的应用程序将以不同的结构方式使用不同的 APDU 命令。

2、响应 APDU

响应 APDU 的格式很简单的：

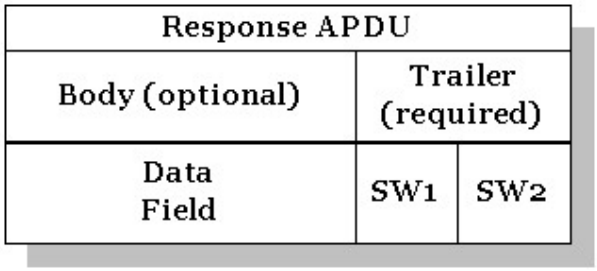


图 4、响应 APDU

和一个 APDU 命令相似，响应 APDU 有可选择的和必要的字段：

• 数据字段（可变长度，由 APDU 命令中的 Le 确定）：这个可选择的字段包含小应用程序返回的数据。

• SW1（1 字节）：这个必要的字段是状态字 1。

• SW2（1 字节）：这个必要的字段是状态字 2。

这些状态字的值在 ISO 7816 - 4 规范中定义：

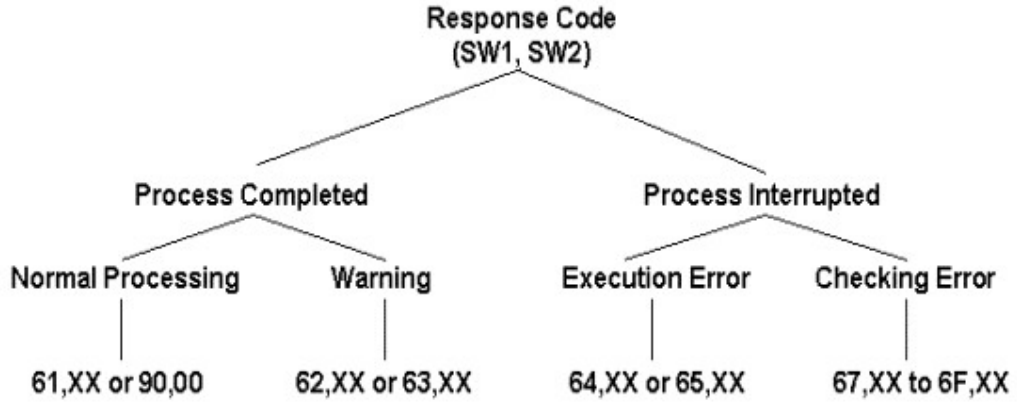


图 5、响应状态码

Java Card 框架应用编程接口中的 ISO7816 Java 接口定义了许多常数来帮助规范返回错误代码。

3、过程 APDU

每当有一个进入的 APDU 用于所选择的小应用程序，JCRE 就调用小应用程序的 process ()方法，把进入的 APDU 作为一个参数传送。这个小应用程序必须解析 APDU 命令，处理数据、生成一个响应 APDU，然后把控制权返回给 JCRE。

RMI(JCRMI)通讯模型

第二种通信模型依靠 J2SE RMI 分布式对象模型的一个子集。

在 RMI 模型中，一个服务器应用程序创建并生成可访问的远程对象，并且一个客户应用程序获

得到远程对象的远程引用，然后调用它们的远程方法。在 JCRMI 中，Java Card 小应用程序是服务器，而主应用程序是客户端。

JCRMI 由类 RMIService 提供到扩展程序包 javacardx.rmi 中。JCRMI 消息被封装到传入 RMIService 方法的 APDU 对象中，换句话说，JCRMI 提供了一个基于 APDU 消息传递模型的分布式对象模型机制，通过这个机制服务器和客户端通信，来回传送方法信息、参数和返回值。

5、虚拟机技术

Java Card 虚拟机(JCVM)规范定义了 Java 程序设计语言的一个子集和一个用于智能卡的兼容 Java 的虚拟机，包括二进制数据表示和文件格式，以及 JCVM 指令集。

用于 Java Card 平台的虚拟机是两部分实现，一部分在卡外，一部分运行在卡本身。卡上的 Java Card 虚拟机解释字节码、管理类和对象等等。外部 Java 虚拟机部分是一个开发工具，一般称为 Java Card 转换工具，装载、检验和进一步地准备卡片小应用程序 Java 类，用于在卡上执行。转换工具输出的是一个 Converted Applet(CAP)文件，这是一个包含一个 Java 程序包中所有类的文件。转换程序检验类是否遵循 Java Card 规范。

JCVM 只支持 Java 程序设计语言的一个有限的子集，然而它保留了许多熟悉的特性，包括对象、继承、程序包、动态对象创建、虚拟方法、接口和异常。JCVM 规范放弃了对许多语言元素的支持，因为这些语言元素可能会用掉很多智能卡本来就很有限的内存：

表格 1、Java Card 语言限制的摘要信息

语言特性	动态类装载、安全管理（java.lang.securitymanager）、线程、对象克隆和某些方面的程序包访问控制不支持。
关键字	不支持 native、synchronized、transient、volatile、strictfp。
类型	不支持 char、double、float 和 long,也不支持多维数组。对 int 的支持是可选的。

类和接口	不支持除了 Object 和 Throwable 以外的 Java 核心应用编程接口类和接口（java.io、java.lang、java.util），并且 Object 和 Throwable 的大部分方法不可用。
异常	一些 Exception 和 Error 子类被省去，因为它们封装的异常和错误不可能在 Java Card 平台上出现。

还有程序模型限制。例如一个装载库类不能再扩展到卡上；它隐含地成为 final 类型。

为了符合存储限制，JCVM 规范额外定义了许多程序属性的约束。表格 4 JCVM 资源限制总结。注意这些约束中许多对于 Java Card 开发者来说是很明白的。

表格 2、Java Card 虚拟机约束的摘要信息

程序包	一个程序包可以引用 128 个其他的程序包
	一个完全合乎要求的程序包名限于 255 字节以内。 注意字符大小取决于字符编码。
	一个完全合乎要求的程序包名限于 255 字节以内。
类	一个类最多可以直接或者间接地实现 15 个接口。
	一个接口最多可以继承于 14 个接口。
	一个程序包如果包含小应用程序（一个小应用程序程序包），它最多可以有 256 个静态方法；如果没有小应用程序（库程序包），它最多只能有 255 个静态方法。
	一个类最多可以实现 128 个 public 或者 protected 实例方法。

在 Java Card 虚拟机中，象在 J2SE 虚拟机中一样，class 文件是核心，但是 JCVM 规范定义了两种其他文件格式来进一步使平台独立，转换小应用程序（Converted Applet，CAP）和导出（Export）格式，这将后面的文章中讲述。

虚拟机的生命周期

JCVM 的生命周期与卡片本身的生命周期一致：在卡片制造并测试之后至发行到持卡人手中的一段时间内它就开始了生命周期，当卡片丢失或者毁坏的时候它的生命周期也就结束了。 卡片没有

电力的时候 JCVM 也不会停止，因为它的状态被保存在卡片的非易失性存储器中。启动 JCVM 初始化 JCRE 并且创建所有的 JCRE 框架对象，这些在 JCVM 的整个生命周期都是运转着的。JCVM 启动之后，与卡片所有的相互作用原则上都是被卡片上的某个小应用程序控制。 当卡片没电的时候，保存在 RAM 中的任何数据都会丢失，但是保存在永久性存储器中的任何状态都被保留。当再次加电以后，虚拟机又再次激活，这时虚拟机和对象的状态被恢复，并且重新开始执行等待进一步地输入。

6、应用编程接口

Java Card 应用编程接口规范定义了传统的 Java 程序设计语言应用编程接口的一个小的子集--甚至小于 J2ME 的 CLDC。不支持字符串也不支持多线程。没有象 Boolean 和 Integer 这样的包装类，也没有 Class 和 System 类。

除 Java 核心类的小子集以外，Java Card 框架还定义了它自己的特定支持 Java Card 应用程序的核心类。这些包含在下面的程序包中：

- java.io 定义了一个异常类，基本的 IOException 类，来完成 RMI 异常层次。除此之外，没有包含其他传统的 java.io 类。
- java.lang 定义了 Object 和 Throwable 类，但是没有 J2SE 中那么多方法。它还定义了许多异常类：Exception 基本类，各种运行时间异常和 CardException。除此之外，没有包含其他传统的 java.lang 类。
- java.rmi 定义了 Remote 接口和 RemoteException 类。 除此之外，没有包含其他传统的 java.rmi 类。 对远程方法调用（Remote Method Invocation，RMI）的支持被包含来简化的移植并整合到使用 Java Card 技术的设备中。
- javacard.framework 定义了组成核心 Java Card 框架的接口，类和异常。 它定义了重要的概念，例如个人识别号（Personal Identification Number，PIN），应用程序协议数据单元（Application Protocol Data Unit，APDU），Java Card 小应用程序 Applet，Java Card System（JCSystem）和

一个 utility 类。 它还定义了各种 ISO7816 常数和各种 Java Card 特定的异常。 表格 5 总结了这些程序包的内容：

Table 5. 表格 Java Card v2.2 javacard.framework

接口	ISO7816 定义与 ISO 7816-3 和 ISO 7816-4 相关的常数。
	MultiSelectable 识别可以支持并发选择的小应用程序。
	个人识别号码（PIN）描述一个被用于安全（验证）目的的个人识别号。
	Shareable 识别一个共享对象。能通过小应用程序防火墙的对象必须实现这个接口。
类	AID 定义了一个遵循 ISO7816-5 与应用程序提供者关联的 Application 标识符；一个小应用程序必备的属性。
	APDU 定义了一个遵循 ISO7816-4 的应用程序协议数据单元，是小应用程序(卡上）和主应用程序（卡外）之间使用的通信格式。
	小应用程序定义了一个 Java Card 应用程序。所有的小应用程序必须扩展这个抽象类。
	JCSystem 提供了控制小应用程序生命周期、资源和事务管理，和小应用程序内部对象共享和对象删除的方法。
	OwnerPIN 是 PIN 接口的一个实现。
	Util 提供用于操作数组和各种 short 的方法，包括 arrayCompare()、arrayCopy()、arrayCopyNonAtomic()、arrayFillNonAtomic()、getShort()、makeShort()、setShort()。
异常	定义了各种的 Java Card 虚拟机异常类：APDUException、CardException、CardRuntimeException、ISOException、PINException、SystemException、TransactionException、UserException。

javacard.framework.service 定义了用于服务的接口、类和异常。 服务处理 APDU 格式的进入的命令。 表格 6 总结了框架服务应用编程接口：

表格 6. javacard.framework.service

接口	Service，基本的服务接口，定义了 processCommand()、processDataIn()和 processDataOut()方法。
	RemoteService 是一个普通 Service，提供到卡上的服务的远程处理。
	SecurityService 扩展了 Service 基本接口，并且提供了查询当前安全状况的方法，包括 isAuthenticated ()、isChannelSecure ()和 isCommandSecure ()。
类	BasicService 是一个服务的默认实现；它提供帮助方法来处理 APDU 和服务协作。
	Dispatcher 维护一个服务的注册。如果你想委托一个 APDU 的处理到几个服务上，你可以使用一个 dispatcher。 一个 dispatcher 可以使用 process ()方法完整的处理一个 APDU，或者使用 dispatch ()方法把它发送到几个服务上让其处理。
异常	ServiceException 一个服务相关的异常

javacard.security 定义了用于 Java Card 安全框架的类和接口。 Java Card 规范定义了一个强健的安全应用编程接口，包括各种型式的私钥和公钥及其算法、用于计算循环码校验（CRCs）的方法、消息摘要和签名：

表格 7. javacard.security

接口	普通的基本接口 Key，PrivateKey、PublicKey 和 SecretKey，以及描述各种类型安全密钥和算法的子接口：AESKey、DESKey、DSAKey、DSAPrivateKey、DSAPublicKey、ECKey、ECPrivateKey、ECPublicKey、RSAPrivateCrtKey、RSAPrivateKey、RSAPublicKey
类	Checksum：用于循环冗余码校验算法抽象基本类
	KeyAgreement：用于秘钥约定算法的基本类
	KeyBuilder：秘钥-对象工厂
	KeyPair：一个保存一对秘钥的容器，一个私钥一个公钥

	MessageDigest：用于散列算法的基本类
	RandomData：用于生成随机数的基本类
	Signature：用于签名算法的基本抽象类
异常	CryptoException：与加密有关异常，比如不支持的算法或者未初始化的秘钥。

javacardx.crypto 是一个扩展程序包，定义了接口 KeyEncryption 和 Cypher 类，都在自己的程序包中，便于控制导出。使用 KeyEncryption 来解密一个使用加密算法的输入秘钥。 Cypher 是所有的密码必须实现的基本抽象类。

CardRemoteObject 定义两个方法 export()和 unexport()，允许或者禁止从卡外到对象的远程访问。RMIService 扩展了 BasicService，并且实现 RemoteService 来处理 RMI 请求。

安全和信任服务应用编程接口(SATSA)

定义在 JSR177 中的 SATSA，指定一个提供用于 J2ME 的安全和信任应用编程接口的可选程序包。客户端应用编程接口提供了到通过一个安全元素（例如一张智能卡）提供的服务的访问，包括敏感信息的安全存储与检索，以及加密和验证服务。

SATSA 利用定义在 CLDC 1.0 版本中的普通连接框架(GCF)来提供到消息传递和 JCRMI 通信模型的更抽象的接口。为了支持信息传送，SATSA 定义了 APDU：URL 模式和 APDUConnection，并且为了支持 JCRMI，它定义了 JCRMI：模式和 JavaCardRMIConnection。

SATSA 有下面的程序包组成：

java.rmi 定义了 Java2 标准版 java.rmi 程序包的一个子集，特别是 Remote 和 RemoteException。

javacard.framework 定义了一个远程方法可能抛出的标准 Java Card 应用编程接口异常：CardRuntimeException、ISOException、APDUException、CardException、PINException、SystemException、TransactionException 和 UserException。

javacard.framework.service 定义了远程的方法可能抛出的一个标准的 Java Card 应用编程接口服务异常：ServiceException。

javacard.security 定义了一个远程方法可能抛出的标准的 Java Card 应用编程接口与加密相关

的异常：CryptoException。

javax.microedition.io 定义了两个连接子接口，APDUConnection 用于基于 APDU 协议的智能卡的访问，JavaCardRMIConnection 用于 Java Card RMI 协议。

javax.microedition.jcrmi 定义了 Java Card RMI stub 编译程序生成的 stub 使用的类和接口。

javax.microedition.pki 定义了用于用户证书基本管理的类。

javax.microedition.securityservice 定义了用于生成应用程序级别的数字签名的类。

Java Card 运行时环境

JCRE 规范定义了 Java Card 虚拟机的生命周期，小应用程序生命周期，小应用程序如何被选择并相互隔离，事务和对象持久性和共享。这 JCRE 提供一个平台无关的接口到卡片的操作系统提供的服务。它由 Java Card 虚拟机、Java Card 应用编程接口和任何特定供应商的扩展组成：

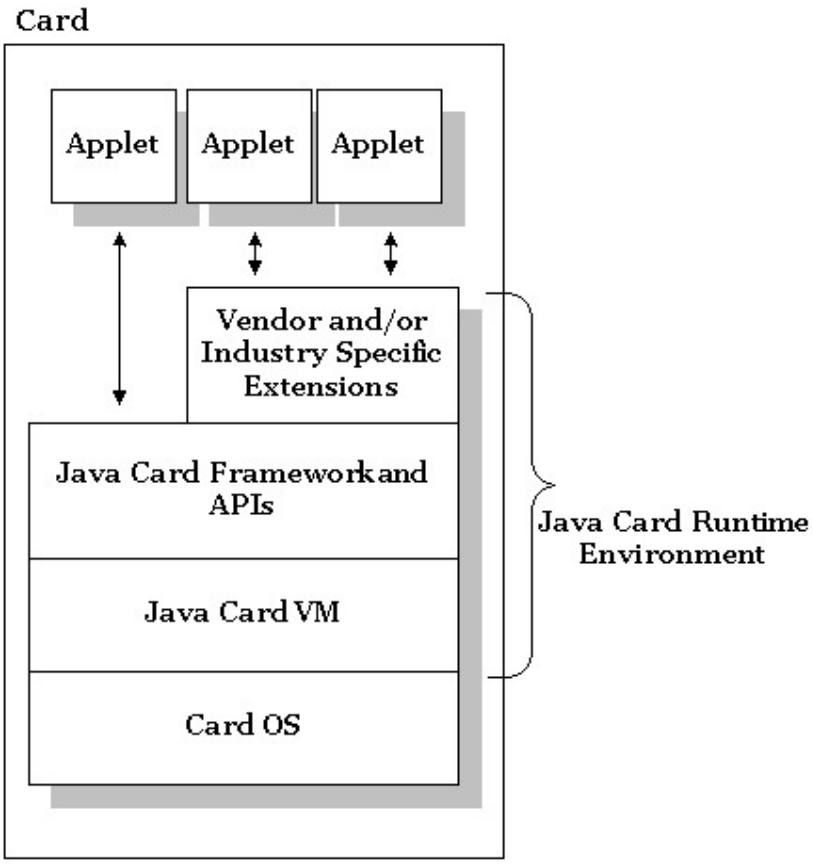


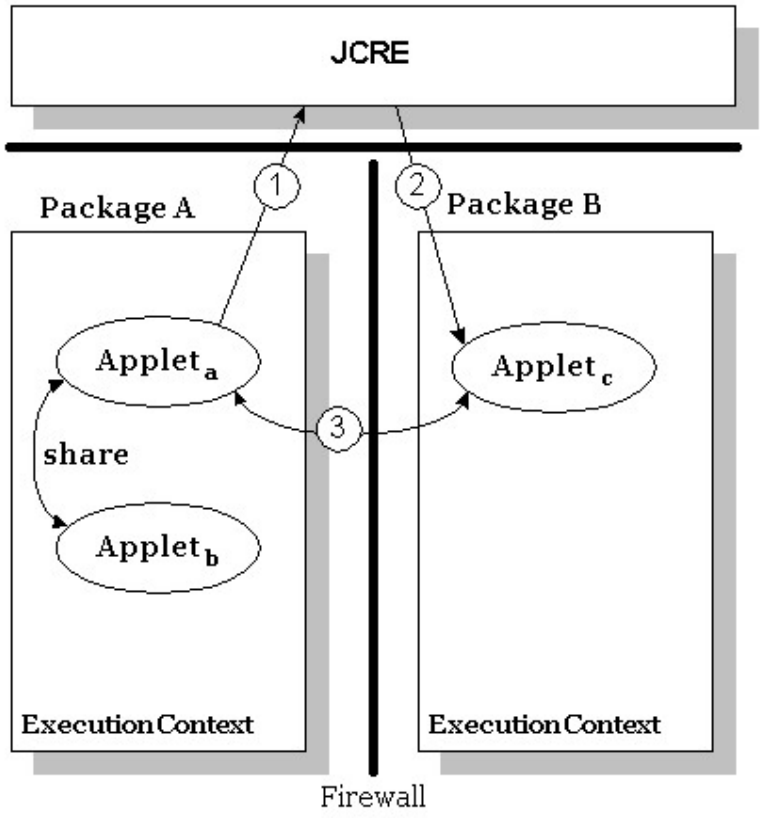
图 Java Card 体系结构和运行时环境

7、小应用程序

Java Card 平台是一个安全的多应用环境-许多来自不同供应商的不同的小应用程序可以在同一张卡片上安全地共存。每个小应用程序被指派给一个执行上下文，这个上下文控制到分配给它的对象的访问。

一个执行上下文和另一个执行上下文之间的界限经常被称为小应用程序防火墙(applet firewall)。它是 Java 沙箱安全概念的一个 Java Card 运行时间改进本，联合类装入器 java.ClassLoader 和访问控制器、 java.AccessController 的功能。Java Card 防火墙创建了一个虚拟堆，这样一个对象只能访问存在于相同的防火墙内的（公共的）方法和数据。 一个防火墙可能包含许多小应用程序及其他对象，比如公共的秘钥。一个 Java Card 执行上下文目前作用域是程序包。当每个对象被创建的时候，它被指派去执行调用程序的上下文。

Java Card 平台支持跨防火墙的安全对象共用。图表 12 描述小应用程序隔离和对象共用：



图表 1. 小应用程序防火墙和对象共用

典型的流程，如图表 12 中的描述：

请求通过调用系统的 JCSystem.getAppletShareableInterfaceObject ()方法访问 Appletc 的共享接口。

由于 Appleta，JCRE 通过调用小应用程序的 getShareableInterfaceObject ()方法来要求 Appletc 的可共享的接口。

如果 Appletc 允许共用，Appleta 将获得一个 Appletc 的共享对象的引用。Appleta 现在就可以访问 Appletc 了。 Appleta 将拥有它创建的任意对象，即使是那些定义在 Appletc 的。

在同一个执行上下文中的小应用程序默认情况下能够相互访问，所以 Appleta 和 Appletb 不需要遵循这个程序来共享对象。

管理内存和对象

在一个 Java Card 设备中，内存是最重要的资源。 在一些 Java Card 中，实现一个垃圾收集程序可能不能使用。当一个对象被创建的时候，对象和它的内容被保存在非易失性存储器中，使之可在会话之间使用。在某些情况下，应用程序数据不需要持久--它是暂时的或者瞬变的（transient）。为了减少智能卡的持久性内存的消耗，并且最大化它的生命周期，我们要尽可能的经常以 transient 更新数据。

Java Card 技术不支持关键字 transient。取而代之，Java Card 应用编程接口（javacard.framework.JCSystem）定义了三个方法，允许你在运行时间创建 transient 数据，还定义了一个方法让你检查一个对象是否是 transient 的：

```
static byte[] makeTransientByteArray(short length, byte event)
static Object makeTransientObjectArray(short length, byte event)
static short[] makeTransientShortArray(short length, byte event)
static byte isTransient(java.lang.Object theObj)
```

你可以创建一个瞬变的字节或者 short 基本数据类型的数组，你也可以创建一个瞬变 Object。但是记住下面用于瞬变数据的行为：

一个瞬变对象的状态在会话之间不能持久保存。 注意内容（不是对象本身）是什么是瞬变的。和任何其他的 Java 语言对象一样，一个瞬变对象只要它被引用就一直存在。

当一个事件例如卡片复位或者小应用程序取消选择发生的时候，一个瞬变对象的内容可能重置为字段的缺省值（0、false 或者 null）。

因为安全的理由，瞬变对象的字段不被保存在持久内存中。

对瞬变对象字段的更新不是原子性的，不会受事务的影响。

在一个 Java Card 环境中，数组和基本类型应在对象声明中声明，并且你应该最小化对象实例化，以利于对象重用。实例化对象在小应用程序生命周期中只有一次，最好在小应用程序的初始化阶段，在小应用程序生命周期中只被调用一次的 install()方法中。

为了促进对象的重用，对象应该保持在小应用程序的生命周期的范围内或引用中，并且它们的状态（成员变量的值）在重用之前根据情况重置。 因为垃圾收集程序并不总是可用的，一个应用程序可能从不回收分配给不太占用内存的对象的存储器。

持久的事务

JCRE 支持原子事务，原子事务安全地更新一个或多个持久对象。如果发生掉电或者程序错误等情况，事务将保护数据的完整性。 事务是在系统级支持的，通过下面的方法：

```
JCSystem.beginTransaction()
JCSystem.commitTransaction()
JCSystem.abortTransaction()
```

在一个为许多事务模型所共用的模式中，一个 Java Card 事务以对 beginTransaction()的调用开始，以对 commitTransaction()或者 abortTransaction()的调用结束。 让我们来看看使用这些应用程序编程接口的代码片断：

```
...
```

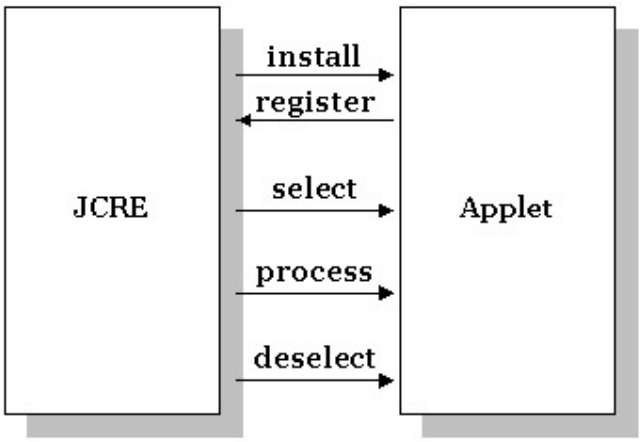
```
private short balance;
...
JCSystem.beginTransaction();
balance = (short)(balance + creditAmount);
JCSystem.commitTransaction();
...
```

实例变量 balance 的更新是为了保证一个原子操作。 如果一个程序错误或者电力重置等事件发生，这个事务就会保证前面的 balance 值余额被恢复。

JCRE 不支持嵌套事务。

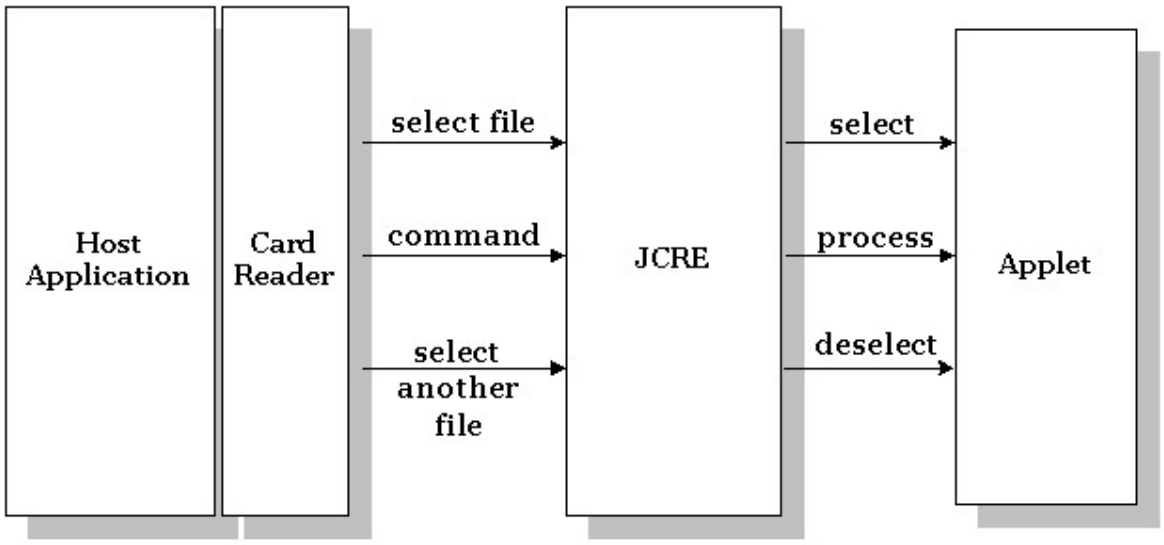
8、生存周期

卡片上的每个小应用程序由一个 Application 标识符（AID）唯一标识。 定义在 ISO 7816 - 5 中的 AID 是一段 5 到 16 字节之间的序列。 所有的小应用程序必须扩展 Applet 抽象基本类，这个类定义了 JCRE 使用的方法来控制小应用程序的生存周期，如图 10 概括：



图表 1. Java Card 小应用程序生命周期方法

小应用程序生存周期在小应用程序被下载到卡片中并且 JCRE 调用小应用程序的 static Applet.install ()方法的时候开始，并且小应用程序通过调用 Applet.register ()在 JCRE 中注册。 一旦小应用程序被安装并且注册，它处于未选择的状态，可以进行选择并且处理 APDU。 图表 11.总结小应用程序方法的操作。



图表 2、使用 Java Card 小应用程序方法

当处在未选择的状态的时候，小应用程序是非激活状态。当主应用程序要求 JCRE 选择一个卡片中特定的小应用程序的时候（通过指示读卡器发送一个 SELECT APDU 或者 MANAGE CHANNEL APDU），一个小应用程序被选择进行 APDU 处理。为了通知这个小应用程序主应用程序已经选择了它，JCRE 调用它的 select()方法；小应用程序一般执行相应的初始化来为进行 APDU 处理做准备。

一旦选择，JCRE 传送输入的 APDU 命令到小应用程序，通过调用它的 process()方法来进行处理。JCRE 捕捉任何小应用程序没能捕捉的异常。

当主应用程序告诉 JCRE 选择另一个小应用程序的时候，前一个小应用程序取消选择。 JCRE 通知活动的小应用程序，它已经通过调用它的 deselect()方法被取消了选择，小应用程序回到不活动的未经选择的状态。

Java Card 会话和逻辑通道

卡片会话是卡片被加电并且和读卡器交换 APDU 的一段时间。

Java Card 2.2 支持逻辑通道（logical channels）的概念，允许最多智能卡中的 16 个应用程序会话同时开启，每个逻辑通道一个会话。因为卡片中的 APDU 的处理不能中断，并且每个 APDU 包含一个到逻辑通道（在 CLA 字节）的引用，变动的 APDU 可以拟同步地访问卡片上的许多小应用程序。你可以设计一个小应用程序被多次选择；也就是说，每次和一个以上逻辑通道通信。多选的小应用程序必须实现 javacard.framework.MultiSelectable 接口和相应方法。

在一些卡片部署中，一个默认小应用程序可以被定义为在卡片复位以后被自动地选择，用于在基本逻辑通道（通路 0）上通信。Java Card 2.2 允许你定义默认小应用程序，但是不指定的如何做；其机理由厂家特定。

JavaCard 小应用程序

1、简介

当创建一个 Java Card 应用程序的时候的典型的步骤是：

1、编写 Java 源代码。

2、编译你的源代码。

3、把类文件改变为一个 Converted Applet（CAP）文件。

4、检验这个 CAP 是否有效；这个步骤是可选的。

5、安装这个 CAP 文件。

当用 Java 程序设计语言开发传统的程序的时候，头两个步骤是相同的：编写.java 文件并且把它们编译成.class 文件。可是，一旦你已经创建 Java Card 类文件，过程会变化的。

Java Card 虚拟机(JCVM)被分成卡外虚拟机和卡内虚拟机。这个分解移除了昂贵的卡外操作，并且考虑到了在卡本身上的小的内存空间，但是它导致在开发 Java Card 应用程序的时候的额外步骤。

在 Java Card 类可以被导入一个 Java Card 设备之前，他们必须被转化成标准的 CAP 文件格式，然后选择性地检验：

- 转化必然伴有把每个 Java 程序包变换到一个 CAP 文件中，在一个程序包中包含类和接口的联合二进制表示法。转化是一个卡外操作。
- 验证是一个可选择的过程，来确认 CAP 文件的结构、有效的字节码子集和程序包内依赖性。你可能想在你使用的第三方供应商程序包上进行验证，或者如果你的转换工具来自一个第三方供应商。验证一般来说是一个卡外操作，但是一些卡片产品可能包括一个机载的检验器。

一旦检验，CAP 文件就即将安装在 Java Card 设备上了。

2、Sun JavaCard Development 工具箱

Sun JavaCard Development 工具箱

你可以使用 Sun JavaCard 开发工具箱编写 JavaCard 小应用程序，并且甚至可以不使用一个智能卡或者读卡器来测试它们。这个工具箱包括所有你开发和测试所需要的 Java Card 小应用程序的基本工具：

1、 Java Card Workstation Development Environment (JCWDE)，一个便利的易于使用的 JavaCard 模拟工具，允许开发者直接执行类文件，而不要转化和安装 CAP 文件。JCWDE 可以和调试程序和 IDE 整合。

从这个开发工具箱的 2.2.1 版本开始，JCWDE 支持 Java Card RMI (JCRMI)。注意 JCWDE 不是一个成熟的 Java Card 模拟器。它不支持许多 JCRE 特性，例如包安装、小应用程序实例创建、防火墙和事务。请参阅这个开发工具箱的用户指南获取更多信息。

2、C 语言 Java Card 运行时环境(C-JCRE)，一个使用 C 语言编写的可执行参考实现。C-JCRE 是一个 Java Card 应用程序编程接口、虚拟机和运行时环境完全兼容的实现。它能让一个开发者在一个工作站环境中精确地测试小应用程序的行为。

C-JCRE 有一些限制：它在一个卡片会话期间支持多达八个可以返回的引用，多达 16 个可以同时被导出的远程对象，8 个远程方法中的数组类型参数，32 个支持的 Java 程序包和 16 个 Java Card 小应用程序。想要获得这些限制条件，请参阅 Java Card 开发工具箱用户指南。

3、 JavaCard 转化工具，用于生成 CAP 文件。

4、 JavaCard 检验，用于选择性地核对 CAP 和导出文件的有效性。

5、 一个发送和接收应用程序协议数据单元(Application Protocol Data Units，APDUs)的 APDU 工具（apdutool）。这样你就可以在 Java Card 小应用程序测试期间发送 APDU。你可以 apdutool 读取的脚本文件，发送 APDUs 到 C-JCRE 或者 JCWDE 中。

6、 一个 capdump 工具，用于转出 CAP 的内容，和一个打印 EXP 文件的 exp2text。

7、 一个 scriptgen 工具，转换 CAP 文件为 APDU 脚本文件。这个工具还被认为是卡外安装程

序。

8、 支持库（用于 Java Card 应用编程接口的类文件和导出文件）文档和范例。

当 Sun JavaCard 开发工具箱允许你编写和测试 Java Card 小应用程序的时候，部署一个现实的端对端的智能卡应用程序需要开发工具箱中没有包含的工具，例如利用了终端应用程序编程接口，如 OpenCard 和 Global Platform 应用程序编程接口。它可能还需要利用例如 Subscriber Identification Module (用户识别模块，SIM)工具包这样的工具来帮助你管理 SIM。

图 1 显示了这个工具包的目录结构（Windows 版本），以及包含开发工具的 bin 目录的内容。

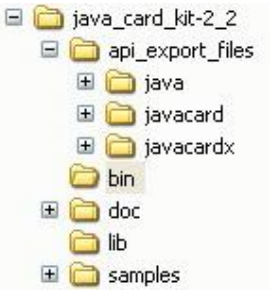


Figure 1a. Development Kit Directory Structure

Figure 1b. Contents of bin directory

现在让我们在看一次 Java Card 开发步骤，这次使用 Sun Java Card Development 工具箱：

1.使用你喜爱的编辑器或者 IDE 编写 Java 源程序。

2.使用你喜爱的编译程序或者 IDE 编译 Java 源程序。

3.选择性地，使用 JCWDE 模拟器测试你的 Java Card 小应用程序。重申一下，JCWDE 不是一个成熟的 Java Card 模拟器。

4.使用工具包的 bin 目录下的转换程序把类文件转化成一个 Converted Applet (转化过的小应用程序，CAP)文件。注意，除类文件之外，另一个输入到这个转换工具中的文件是导出文件，提供了关于你的应用程序导入的（引用）的程序包的信息。这些是还被装载到卡片中的程序包。导出文件还是转换工具的一个输出。

5.选择性地，检验 CAP 的有效性。这一步包括使用 verifycap 脚本来验证 CAP 文件的有效性，使用 verifyexp 来验证导出文件，并且使用 verifyrev 来检验程序包修正之间的二进制兼容性。工具 verifycap、verifyexp 和 verifyrev 脚本全部都可在 bin 目录中得到。

6.安装 CAP 文件。使用 scriptgen 工具转换 CAP 文件为一个（安装）APDU 脚本文件。然后使用 apdutool 发送脚本文件（安装 APDU 命令和 CAP 文件）到 Java Card 设备上的 C-JCRE 或者一个 JCRE。JCRE 保存 CAP 文件到卡片的内存中。

下面的图总结了这些步骤。注意每个 Java Card 供应商提供它自己的工具，但是这些用于开发一个 Java Card 小应用程序的步骤在开发工具箱之间通常是相同的：

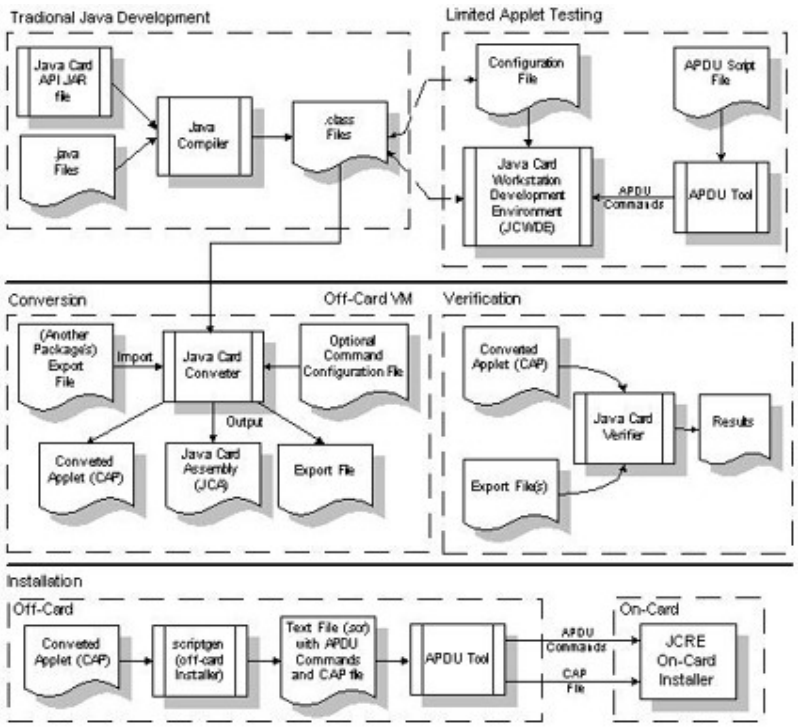


Figure 2. Java Card Development Steps(click for larger image)

3、小应用程序结构

Sun 提供了两个模型用来设计 JavaCard 应用程序（javacard.framework.Applet）：传统的 JavaCard API 和 JavaCard Remote Method Invocation（Java Card 远程方法调用，JCRMI）编程接口。我们可以

使用其中任何一个来编写 Java Card 小应用程序，开发 Java Card 小应用程序是一个两步的过程：

1.定义负责主应用程序和小应用程序之间接口的命令和响应 APDU。

2.编写 Java Card 小应用程序本身

JavaCard 小应用程序结构

首先，让我们看一下 Java Card 小应用程序的结构。

列表 1 说明了一个典型的 JavaCard 小应用程序是如何构造的：

```
import javacard.framework.*

...

public class MyApplet extends Applet {

// Definitions of APDU-related instruction codes

...

MyApplet() {...} // Constructor

// Life-cycle methods

install() {...}

select() {...}

deselect() {...}

process() {...}

// Private methods

...

}
```

列表 1.一个 JavaCard 小应用程序的结构

一个 JavaCard 小应用程序通常定义它的 APDU 相关指令、它的构造器，然后是 Java Card 小

应用程序的生命周期方法：install ()、select ()、deselect ()和 process ()。最后，它定义任何合适的私有方法。

4、定义 APDU 指令

定义 APDU 指令

不同的 Java Card 应用程序有不同的接口（APDU）需求。一个信用卡小应用程序可能支持验证 PIN 号码的方法，产生信用和借记事务，并且核对帐目余额。一个健康保险小应用程序可能提供访问健康保险信息、保险总额限制、医生、病人信息等等信息的权限。你定义的精确的 APDU 全依赖你的应用程序需求。

举例来说，让我们亲身感受一下如何开发经典的 Wallet 信用卡示例。你可以在 Sun Java Card Development 工具箱的 samples 目录下得到这个及其他示例的完整的代码。

我们将开始定义一个 APDU 命令来查询保存在 Java Card 设备上的当前余额数。注意，在一个实际信用卡应用程序中，我们还将定义信用并且借记命令。我们将分配我们的 Get Balance APDU 一个 0x80 指令类和一个 0x30 指令。Get Balance APDU 不需要任何指令参数或者数据区，并且预期的响应由包含余额的两个字节组成。下一个表格描述 Get Balance APDU 命令：

表 1 - Get Balance APDU 命令

Name	CLA	INS	P1	P2	Lc	Data Field	Le (size of response)
Get Balance	0x80	0x30	0	0	N/A	N/A	2

虽然 Get Balance 命令未定义输入数据，但是有一些命令 APDU 将定义输入数据。举例来说，让我们定义验证从卡片读取器中传递来的 PIN 号码的 Verify PIN APDU 命令。下一个表格定义 Verify APDU：

表格 2- Verify APDU 命令

Name	CLA	INS	P1	P2	Lc	Data Field	Le (size of response)
Verify PIN	0x80	0x20	0	0	PIN Len	PIN Value	N/A

注意 Le 字段，响应的大小是 N/A。这是因为没有到 Verify PIN 的应用程序特定响应；成功或者失败通过响应 APDU 中的状态字标明。

为了简化 APDU 过程，javacard.framework.ISO7816 接口定义了许多常数，我们可以用来从 process ()方法传送到小应用程序中的输入缓冲器中检索各个的 APDU 字段：

```
...
byte cla = buf[ISO7816.OFFSET_CLA];
byte ins = buf[ISO7816.OFFSET_INS];
byte p1 = buf[ISO7816.OFFSET_P1];
byte p2 = buf[ISO7816.OFFSET_P2];
byte lc = buf[ISO7816.OFFSET_LC];
...
// Get APDU data, by copying lc bytes from OFFSET_CDATA, into
// reusable buffer databuf.
Util.arrayCopy(buf, ISO7816.OFFSET_CDATA, databuf, 0, lc);
...
```

列表 2、使用 ISO-7816-4 常数

现在我们将定义用于 Get Balance 和 Verify 命令的类(CLA)和指令(INS)，Get Balance 响应的大小，以及在如果 PIN 验证失败后的出错返回代码。

```
...
// MyApplet APDU definitions
final static byte MyAPPLET_CLA = (byte)0x80;
```

```
final static byte VERIFY_INS = (byte)0x20;

final static byte GET_BALANCE_INS = (byte) 0x30;

final static short GET_BALANCE_RESPONSE_SZ = 2;

// Exception (return code) if PIN verify fails.

final static short SW_PINVERIFY_FAILED = (short)0x6900;

...
```

列表 3、小应用程序的 APDU 定义

接下来，让我们定义小应用程序构造器和生命循环方法。

5、构造器

构造器

定义一个初始化这个对象的状态的私有构造器。这个构造器被从 install()方法调用；换句话说，构造器只在小应用程序的生命周期期间被调用：

```
/**

* Private Constructor.

*/

private MyApplet() {

super();



// ... Allocate all objects needed during the applet's

// lifetime.

ownerPin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);

...

// Register this applet instance with the JCRE.
```

```
register();
}
```

列表 4、小应用程序构造器

在这个示例中，我们使用一个 javacard.framework.OwnerPIN，一个描述个人识别号码的对象；这个对象将存在于 Java Card 小应用程序的一生。回忆一下本文第一部分中的"管理内存和对象"，在一个 Java Card 环境中，数组和基本类型将在对象声明中被声明，而且你应该最小化对象实例，以利于对象重用。在小应用程序生命周期期间，以创建对象一次。做到这点的一个简易的方法是在构造器中创建对象，并且从 install()方法中调用这个构造器-- install()本身在小应用程序生命周期中只被调用一次。为了利于再使用，对象应该保持在范围中或者适当的引用中，用于小应用程序的生命周期，并且它们的成员的值在再使用之前适当的重置。因为一个垃圾收集程序并不总是可用，一个应用程序可能从不回收被分配给对象的存储空间。

install ()方法

JCRE 在安装过程期间调用 install()。你必须覆盖这个从 javacard.framework.Applet 类继承来的方法，并且你的 install ()方法必须实例化这个小应用程序，如下：

```
/**
* Installs the Applet. Creates an instance of MyApplet. The
* JCRE calls this static method during applet installation.
* @param bArray install parameter array.
* @param bOffset where install data begins.
* @param bLength install parameter data length.
* @throw ISOException if the install method fails.
*/
public static void install(byte[] bArray, short bOffset, byte bLength)
throws ISOException {
```

```
// Instantiate MyApplet
new MyApplet();
...
}
```

列表 5、install ()小应用程序生命周期方法

install ()方法必须直接或者间接地调用 register ()方法来完成安装；如果这步失败将导致安装失败。在我们的范例中，构造器调用 register()。

select()方法

JCRE 调用 select()来通知已经被选作 APDU 过程的小应用程序。你不必实现这个方法，除非你想提供会话初始化或者个性化。select()方法必须返回 true 来指明它即将处理进入的 APDU，或者返回 false 来拒绝选择。javacard.framework.Applet 类的默认实现返回 true。

```
/**
* Called by the JCRE to inform this applet that it has been
* selected. Perform any initialization that may be required to
* process APDU commands. This method returns a boolean to
* indicate whether it is ready to accept incoming APDU commands
* via its process() method.
* @return If this method returns false, it indicates to the JCRE
* that this Applet declines to be selected.
*/
public boolean select() {
// Perform any applet-specific session initialization.
return true;
}
```

列表 6、select()小应用程序生命周期方法

deselect()方法

JCRE 调用 deselect()来通知小应用程序，它已经被取消选定了。你不必实现这个方法，除非你想提供会话清除。javacard.framework.Applet 类的默认实现什么都不做。

```
/**
* Called by the JCRE to inform this currently selected applet
* it is being deselected on this logical channel. Performs
* the session cleanup.
*/
public void deselect() {
// Perform appropriate cleanup.
ownerPin.reset();
}
```

列表 7、deselect()小应用程序生命周期方法

在我们的示例中，我们重置了 PIN（个人识别号码）。

process()方法--感受 APDU 的全过程

一旦一个小应用程序已经被选择，它将准备接收命令 APDUs，如在本文第一部分中"Java Card 小应用程序的生命周期"描写的。

回想一下被从主机端（客户端）应用程序发送到卡片的 APDU 命令，如下面的说明：

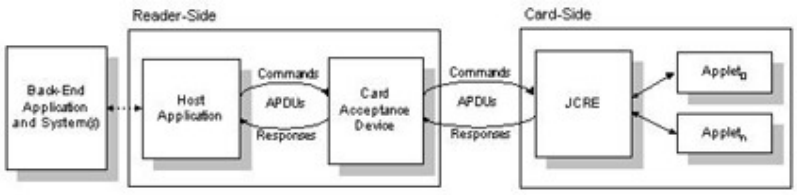


Figure 3. APDU 指令和响应流程

每次 JCRE 接收一个 APDU 命令(通过卡片读取器从主应用程序,或者如果使用 Sun Java Card Development 工具箱就通过 apdutool），它调用小应用程序的 process()方法，把输入命令当作一个参数传送给它（APDU 命令输入缓冲中的参数）。process()方法然后：

1.摘录 APDU CLA 和 INS 字段

2.检索应用程序特定的 P1、P2 和数据字段

3.处理 APDU 数据

4.生成并发送一个响应

5.优雅地返回，或者抛出相应的 ISO 异常

在此时，JCRE 发送合适的状态字回到主应用程序，通过读卡器。

列表 8 显示一个样本 process()方法。

```
/**
* Called by the JCRE to process an incoming APDU command. An
* applet is expected to perform the action requested and return
* response data if any to the terminal.
*
* Upon normal return from this method the JCRE sends the ISO-
* 7816-4-defined success status (90 00) in the APDU response. If
* this method throws an ISOException the JCRE sends the
* associated reason code as the response status instead.
```

```
* @param apdu is the incoming APDU.

* @throw ISOException if the process method fails.

*/

public void process(APDU apdu) throws ISOException {



// Get the incoming APDU buffer.

byte[] buffer = apdu.getBuffer();



// Get the CLA; mask out the logical-channel info.

buffer[ISO7816.OFFSET_CLA] =

(byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);



// If INS is Select, return - no need to process select

// here.

if ((buffer[ISO7816.OFFSET_CLA] == 0) &&

(buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)) )

return;



// If unrecognized class, return "unsupported class."

if (buffer[ISO7816.OFFSET_CLA] != MyAPPLET_CLA)

ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);



// Process (application-specific) APDU commands aimed at

// MyApplet.
```

```
switch (buffer[ISO7816.OFFSET_INS]) {


case VERIFY_INS:
verify(apdu);
break;


case GET_BALANCE_INS:
getBalance(apdu);
break;


default:
ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
break;
}
}
```

列表 8、process()小应用程序生命周期方法

我们的 process()方法调用 getBalance()和 verify()方法。列表 9 显示 getBalance ()方法，处理 get balance APDU 并且返回保存在卡片中的余额。

```
/**
* Retrieves and returns the balance stored in this card.
* @param apdu is the incoming APDU.
*/
private void getBalance(APDU apdu) {
```

```
// Get the incoming APDU buffer.
byte[] buffer = apdu.getBuffer();

// Set the data transfer direction to outbound and obtain
// the expected length of response (Le).
short le = apdu.setOutgoing();

// If the expected size is incorrect, send a wrong-length
// status word.
if (le != GET_BALANCE_RESPONSE_SZ)
ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

// Set the actual number of bytes in the response data field.
apdu.setOutgoingLength((byte)GET_BALANCE_RESPONSE_SZ);

// Set the response data field; split the balance into 2
// separate bytes.
buffer[0] = (byte)(balance >> 8);
buffer[1] = (byte)(balance & 0xFF);

// Send the 2-byte balance starting at the offset in the APDU
// buffer.
apdu.sendBytes((short)0, (short)GET_BALANCE_RESPONSE_SZ);
```

```
}
```

列表 9、处理 Get Balance APDU

getBalance ()方法通过调用 APDU.getBuffer ()方法取得一个引用到 APDU 缓冲。在返回响应(当前余额）之前，小应用程序必须设置 JCRE 模式通过调用 APDU.setOutgoing()方法来发送，方便地返回期望的响应大小。我们还必须设置响应数据字段中的字节的实际数字，通过调用 APDU.setOutgoingLenth()。APDU 缓冲中的响应事实上通过调用 APDU.sendBytes ()发送。

小应用程序不直接发送返回码（状态字)；一旦小应用程序调用 APDU.setOutgoing ()并且提供任何请求的信息，JCRE 注意这个状态字。状态字的值依靠 process()方法如何使返回到 JCRE 来变化。如果所有的已经正常运行，JCRE 将返回 9000，指明无错。你的小应用程序可以通过抛出一个定义在 ISO7816 接口中的异常返回一个错误代码。 在列表 9 中，如果期望响应的大小不正确，方法 getBalance()抛出一个 ISO7816.SW_WRONG_LENGTH 代码。对于有效的状态码值，请参阅 ISO7816 接口的定义，或者回到本文第一部分的"响应 APDU"。

现在让我们看看在列表 10 中的 verify()方法。因为我们定义的验证 PIN APDU 命令包含数据，verify()方法必须调用 APDU.setIncomingAndReceive ()方法，设置 JCRE 为接收模式，然后接收输入数据。

```
/**
* Validates (verifies) the Owner's PIN number.
* @param apdu is the incoming APDU.
*/
private void verify(APDU apdu) {


// Get the incoming APDU buffer.
byte[] buffer = apdu.getBuffer();
```

```
// Get the PIN data.
byte bytesRead = (byte)apdu.setIncomingAndReceive();

// Check/verify the PIN number. Read bytesRead number of PIN
// bytes into the APDU buffer at the offset
// ISO7816.OFFSET_CDATA.
if (ownerPin.check(buffer, ISO7816.OFFSET_CDATA, byteRead)
== false )
ISOException.throwIt(SW_PINVERIFY_FAILED);
}
```

列表 10、处理验证 APDU

这个方法通过调用 APDU.getBuffer() 取得一个到 APDU 缓冲的引用，调用 APDU.setIncomingAndReceive()来接收命令数据，从输入的 APDU 缓冲中取得 PIN 数据，并且验证 PIN。一个验证失败导致状态码 6900 被发送回主应用程序。

有时输入的数据比填充到 APDU 缓冲中的数据要多，并且小应用程序必须大块的读取数据知道没有数据可以读取。在此情况下，我们必须首先调用 APDU.setIncomingAndReceive()，然后重复地调用 APDU.receiveBytes()，直到不再有数据可用。列表 11 显示如何读取大量输入数据。

```
...
byte[] buffer = apdu.getBuffer();
short bytes_left = (short) buffer[ISO.OFFSET_LC];
short readCount = apdu.setIncomingAndReceive();
while (bytes_left > 0) {
```

```
// Process received data in buffer; copy chunk to temp buf.
Util.arrayCopy(buffer, ISO.OFFSET_CDATA, tbuf, 0, readCount);
bytes_left -= readCount;
// Get more data
readCount = apdu.receiveBytes(ISO.OFFSET_CDDATA);
}
...
```

列表 11、读取大量输入数据

由于每个大块被读取，小应用程序可以把它添加到另一个缓冲中，否则仅仅处理它。

6、JavaCard RMI 模型介绍

Java Card 小应用程序编程的第二个模型是 JavaCard RMI (JCRMI)，严格的说它是 J2SE RMI 分布对象模型的缩小版。

这是一个以对象为中心的模型，根据这个模型你在上一节看到的 APDU 通信和句柄将被抽象化；取而代之的是你处理对象。这简化了编程和集成基于 Java Card 技术的设备。

在 RMI 模型中，一个服务器应用程序创建并生成可访问的远程对象，并且一个客户应用程序获得到服务器的远程对象的远程引用，然后调用它们的远程方法。在 JCRMI 中，Java Card 小应用程序是服务器，而主应用程序是客户端。

JavaCard RMI 简介

两个程序包提供了 JavaCard RMI 支持：

1、java.rmi 定义了 Java 2 标准版 java.rmi 程序包的一个子集。它定义了 Remote 接口和 RemoteException 类。除此之外，没有包含其他传统的 java.rmi 类。

2、Javacard.framework.service 定义了 Java Card 小应用程序服务类，包括 RMI 服务类

CardRemoteObject 和 RMIService。

3、Class CardRemoteObject 定义了两个方法启动和禁止卡片外的对象的远程访问。类 RMIService 处理 RMI 请求（转化输入的命令 APDU 为远程方法调用）。

编写一个 JCRMI 应用程序类似于编写一个典型的基于 RMI 的应用程序：

1.定义远程类的行为为一个接口。

2.编写远程类的服务器实现，和支持类。

3.编写一个使用远程服务的客户机程序和支持类。

注意 JCRMI 没有改变小应用程序的基本结构或者生命周期，你不久就会看到。

7、远程接口

远程接口

创建一个远程服务中的第一步是定义它的可见行为。远程接口定义你的小应用程序提供的服务。和在标准的 J2SE RMI 中一样，所有的 Java Card RMI 远程接口必须扩展 java.rmi.Remote 接口。为了说明，这里有一个远程接口，揭示一个取得保存在卡片里的余额的方法：

```
import java.rmi.*;
import javacard.framework.*;

public interface MyRemoteInterface extends Remote {
...
public short getBalance() throws RemoteException;
...
// A complete credit card application would also define other
// methods such as credit() and debit() methods.
```

```
...
}
```

列表 1、远程接口

MyRemoteInterface 定义这个远程方法，在我们的示例中一个 getBalance ()方法，来取得保存在智能卡中的余额。注意，除了 Java Card 特定的导入之外，这个远程接口看起来完全象一个标准的 RMI 远程接口。

服务器实现

下一步是实现服务器的行为。服务器实现包含 Java Card 小应用程序，在所有的你已经定义的远程接口当中的实现，和任何与你的应用程序相关的特定类。

8、小应用程序

Java Card 小应用程序

Java Card 小应用程序是 JCRMI 服务器，并且是主机（客户端）应用程序可用的远程对象的所有者。一个典型的 Java Card RMI 小应用程序的结构在下面的图表中说明：

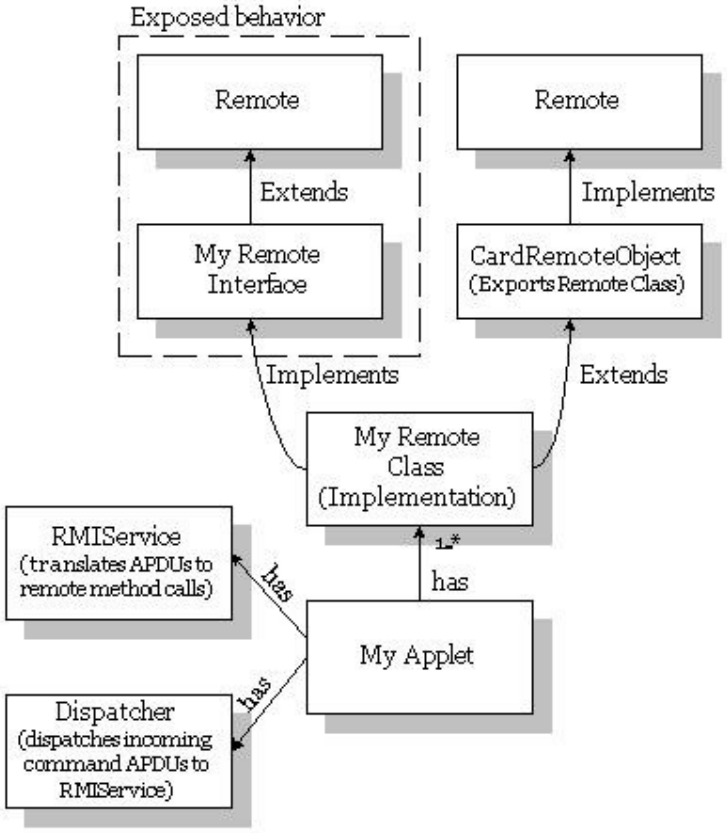


Figure 4. 典型的 JavaCard RMI 应用程序结构

当和明确地处理 APDU 消息的小应用程序相比，基于 JCRMI 的小应用程序更像一个对象容器。如图 4 所示，基于 JCRMI 的小应用程序有一个或多个远程对象，一个 APDU Dispatcher 和一个接收 APDU 并且把它们转化为远程方法调用的 RMIService。Java Card 远程类可以扩展 CardRemoteObject 类，自动导出对象，使之可用于远程使用。

JCRMI 小应用程序必须扩展 javacard.framework.Applet，遵循标准的小应用程序结构，并且定义适当的生命周期方法。它必须安装并且登记本身，并且分配 APDU。下面这一代码片断说明一个基于 JCRMI 的小应用程序的典型结构：

```
public class MyApplet extends javacard.framework.Applet {



private Dispatcher disp;
```

```
private RemoteService serv;

private Remote myRemoteInterface;



/**

* Construct the applet. Here instantiate the remote

* implementation(s), the APDU Dispatcher, and the

* RMIService. Before returning, register the applet.

*/

public MyApplet () {

// Create the implementation for my applet.

myRemoteInterface = new MyRemoteInterfaceImpl();

// Create a new Dispatcher that can hold a maximum of 1

// service, the RMIService.

disp = new Dispatcher((short)1);

// Create the RMIService

serv = new RMIService(myRemoteInterface);

disp.addService(serv, Dispatcher.PROCESS_COMMAND);

// Complete the registration process

register();

}

...
```

小应用程序创建一个 Dispatcher 和一个处理输入的 JCRMI APDU 的 RMIService。

```
...
/**
* Installs the Applet. Creates an instance of MyApplet.
* The JCRE calls this static method during applet
* installation.
* @param bArray install parameter array.
* @param bOffset where install data begins.
* @param bLength install parameter data length.
*/
public static void install(byte[] aid, short s, byte b) {
new MyApplet();
}
```

在 JavaCard 环境中，JVM 的生命周期是物理卡片的生命周期，而不是提供垃圾收集器的所有的 Java Card 实现的，所以你需要最小化内存分配。在安装时间创建对象，这样内存只被分配给它们一次。

```
/**
* Called by the JCRE to process an incoming APDU command. An
* applet is expected to perform the action requested and
* return response data, if any.
*
* This JCRMI version of the applet dispatches remote
* invocation APDUs by invoking the Dispatcher.
*
```

```
* Upon normal return from this method the JCRE sends the ISO-
* 7816-4-defined success status (90 00) in the APDU response.
* If this method throws an ISOException, the JCRE sends the
* associated reason code as the response status instead.
* @param apdu is the incoming APDU.
* @throw ISOException if the install method fails.
*/
public void process(APDU apdu) throws ISOException {
// Dispatch the incoming command APDU to the RMIService.
disp.process(apdu);
}
```

代码列表 13.Java Card RMI 小应用程序

小应用程序的 process()方法接收一个 APDU 命令并且把它发送到 RMIService，RMIService 通过把它转化为一个 RMI 调用和后续响应处理这条命令

9、实现远程对象

实现远程对象

实现一个 JCRMI 远程对象类似于实现标准的 J2SE RMI 远程对象。主要区别是在 JCRMI 中你的远程对象有扩展 CardRemoteObject 的选择（除了实现你的远程接口之外）。

CardRemoteObject 定义两个方法 export()和 unexport()，分别允许或者禁止从卡外到对象的访问。通过扩展 CardRemoteObject，你自动地导出你的远程对象所有的方法。如果你决定不扩展 CardRemoteObject，你将要负责通过调用 CardRemoteObject.export()导出它们。

```
import java.rmi.RemoteException;
```

```
import javacard.framework.service.CardRemoteObject;
import javacard.framework.Util;
import javacard.framework.UserException;
/**
* Provides the implementation for MyRemoteInterface.
*/
public class MyRemoteImpl extends CardRemoteObject implements MyRemoteInterface {
/** The balance. */
private short balance = 0;

/**
* The Constructor invokes the superclass constructor,
* which exports this remote implementation.
*/
public MyRemoteImpl() {
super(); // make this remote object visible
}

/**
* This method returns the balance.
* @return the stored balance.
* @throws RemoteException if a JCRMI exception is
* encountered
*/
```

```
public short getBalance() throws RemoteException {
return balance;
}

// Other methods
...
}
```

列表14.远程对象实现

10、完整的 Java Card RMI 应用程序流程

一个完整的 Java Card RMI 应用程序的流程

让我们概述一个 JCRMI 应用程序的流程。客户端（主机）应用程序通过传递 RMI APDU 到卡上的 JCRE 来产生 RMI 调用，依次转送这些 APDU 到相应的 JCRMI 小应用程序。这个小应用程序分配接收的 APDU 到 RMIService，依次处理 APDU 并且转化它为一个 RMI 调用。一个 JCRMI 小应用程序的典型流程在下面说明：

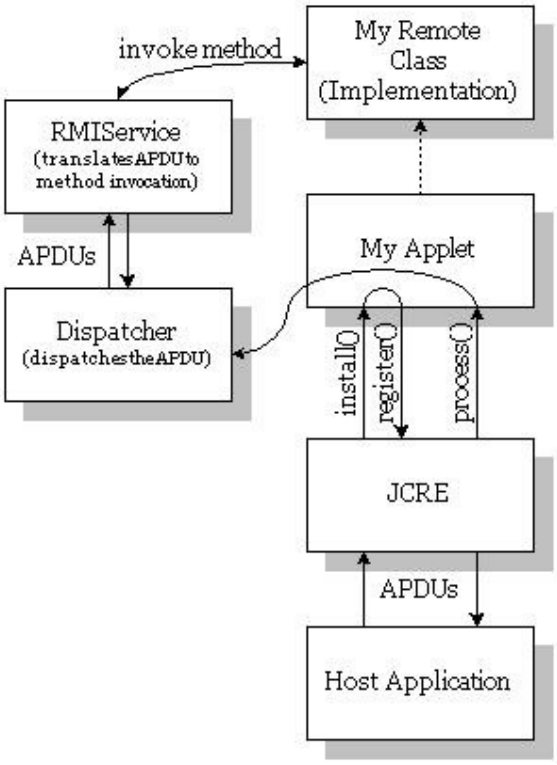


Figure 5. 基于 JavaCard RMI 模型的应用程序流程

简言之，JCRMI 提供一个基于 APDU 的消息传递模型的分布式对象模型机制。JCRMI 消息被封装到传送到 RMIService 的 APDU 消息中，负责解码 APDU 命令，并且转化这些命名到方法调用和响应。这允许服务器和客户端通信，来回传送方法信息、参数和返回值。

JavaCard 主应用程序开发教程

1、JavaCard 应用程序的组成元素

JavaCard 应用程序的组成元素

JavaCard 应用程序不是独立的，而是一个端对端的应用程序的一部分：

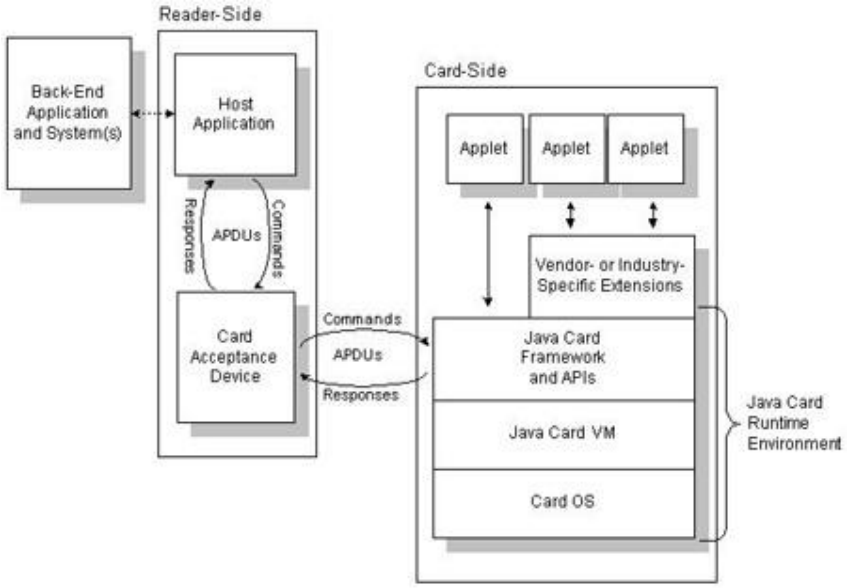


Figure 1.JavaCard 应用程序的典型组成

一个典型的 JavaCard 应用程序由以下部分组成：

1、提供访问例如保存在数据库中的安全或者电子付款信息的 back-office 服务的后端应用程序。后端应用程序如何开发超出了本文的范围。

2、在卡外，驻留在一个卡片终端上，主应用程序使用许多用于卡片访问的接口之一来访问智能卡上的小应用程序，例如 Java Card RMI、OpenCard Framework 应用编程接口或者 Security and Trust Services 应用编程接口(SATSA)。

3、读卡器，卡片终端或者卡片接收设备，提供主应用程序和卡上小应用程序之间的物理接口。

4、卡上的是 Java Card 小应用程序和 Java Card 框架。注意，在访问小应用程序之前，主应用程序必须提供证书并且验证自己。

编写一个主应用程序-访问你的小应用程序

客户端上的主应用程序处理用户、JavaCard 小应用程序和供应商的后端应用程序之间的通讯。主程序访问你的小应用程序提供的服务。它存在于终端或者卡片接收设备上，例如一个工作站、一个售货点(POS)终端、一个手提电话或者一个机顶盒。回想一下一个主机应用程序和小应用程序使用 ISO - 7816 APDU 命令经由读卡器或终端进行交互。

传统的读卡端应用程序使用 C 语言编写，但是主机程序可以使用 Java 程序语言或者其他语言编写，只要它能够与小应用程序交换有效的 ISO - 7816 APDU 命令。

现在部署的大部分的手提电话整合一个智能卡阅读器访问捆绑在它上面的 SIM 卡。使用即将到来的 JSR 177、用于 J2ME 的安全和信任服务应用编程接口(SATSA)和 J2ME 设备的广泛采用，我们可以想象有许多主应用程序将使用移动设备上的 Java 技术编写。SATSA 的意图是启动一个运行在基于 J2ME 的设备上的 Java Card 主应用程序。JSR 177 目前处在 JCP 团体审查阶段。

当你编写客户端应用程序的时候，有三个主要的应用程序编程接口可用：OpenCard Framework、JavaCard RMI Client 应用编程接口和安全与信任服务应用编程接口(SATSA)。我们将依次看看这些应用程序编程接口。

2、OpenCard 框架介绍

OpenCard 框架介绍

智能卡供应商一般不仅提供开发工具箱，而且提供支持读取端应用程序和 JavaCard 小应用程序的应用程序编程接口。许多供应商支持 OpenCard 框架(OCF)，这是一套基于 Java 的应用程序编程接口，把一些来自不同的供应商的与读卡器交互的细节隐藏起来。

OpenCard 联盟是一群推动 OpenCard 框架定义与采用的公司，目前 OpenCard 框架是 1.2 版本。OCF 的目标是提供给主机端应用程序的开发者跨不同的卡片读取器供应商工作的应用编程接口。

OCF 为你定义许多标准卡服务。其中两个是 FileAccessCardService 和 SignatureCardService。一个特殊的类型是 ApplicationManagerCardService，提供了在卡上安装、注册和删除小应用程序的生命周期管理方法。

当编写一个主机端基于 OCF 的应用程序时，你基本上要把它分离成两个部分：

1、和终端或者读取器交互的主应用程序对象（初始化 OCF，等待卡片插入并且终止 OCF），并且能够显露高级的卡片访问方法，例如 getBalance ()。

2、一个实现实际的低级通道管理和 APDU 输入/输出的小应用程序代理。当把 APDU 细节从应用程序中隐藏起来的时候，这个代理（Proxy）设计模式允许你显露一个面向对象接口。

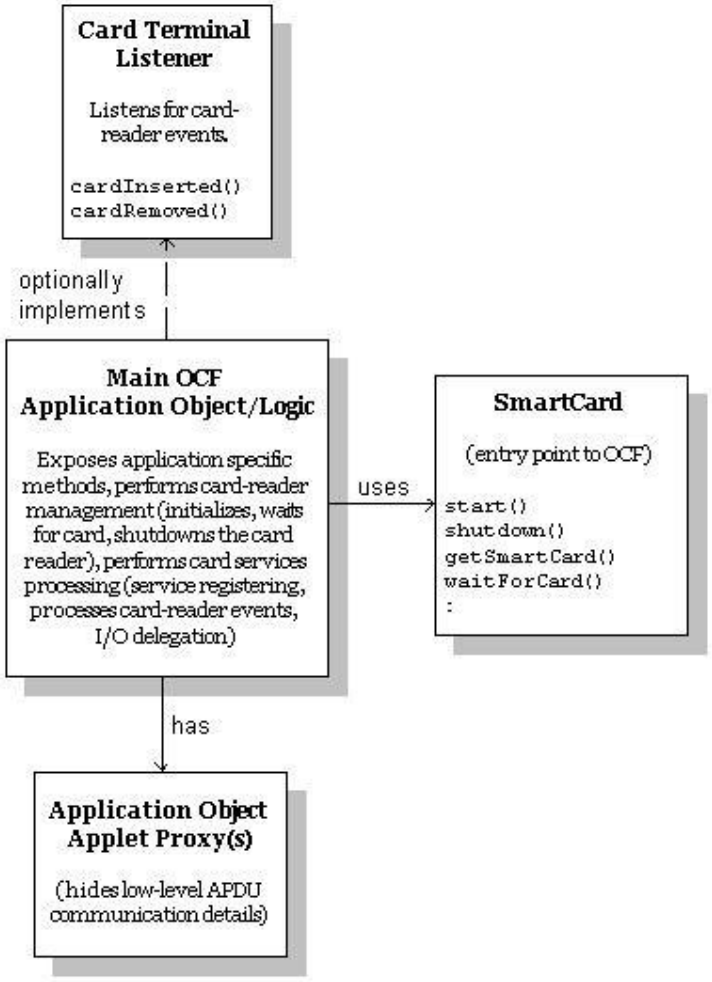


Figure 2. OCF 应用程序的结构

总之，一个典型的 OCF 应用程序具有一个或多个 main 对象，都是在主机上创建，可能再它们的自己执行的线程上。这些 main 应用程序对象显露了高级的特定应用程序调用，这些最终都被委托给小应用程序代理。它们使用 SmartCard 对象，这是应用程序到 OCF 的入口点，启动应用程序来初始化并且关闭 OCF，并且等待卡片被插入。main 对象可以实现一个 CTListener，你不久将看到，这个监听者提供了诸如卡片插入和拔出等事件的异步标志信息。

你可以使用一个同步或者异步模型编写你的应用程序。

在同步模型中，你的主应用程序初始化 OCF，然后等待卡片被插入。然后它执行你的 main 应用程序逻辑，并且在完成的时候关闭 OCF：

```
...
try {
   // Initialize OCF
   SmartCard.start();
   // Wait for a smart card
   CardRequest cr = new CardRequest(CardRequest.NEWCARD, null,
   OCFCardAccessor.class);
   SmartCard myCard = SmartCard.waitForCard(cr);


   // Main client work is done here...
   ...


} catch (Exception e){
   // Handle exception
} finally {
   try {
```

```
   // Shut down OCF
   SmartCard.shutdown();
   } catch (Exception e) {
      e.printStackTrace();
}
}
...
```

列表 1.一个同步 OCF 应用程序的典型结构

如果你喜欢使用异步的途径，你的类必须实现 CTListener 接口，并且，在初始化阶段，注册它自己用于诸如插入和拔出等卡片终端事件的通知。下面的应用程序骨架以初始化 OCF 和注册监听者开始，然后定义了用于有效事件的回调方法。

```
public class MyHostSideApp implements CTListener
...
public MyHostSideApp() {
   try {
      // Initialize the framework
      SmartCard.start ();
      // Register this as a Card Terminal Event Listener
      CardTerminalRegistry.getRegistry().addCTListener(this);
   } catch (Exception e) {
      // handle error...
   }
}
```

```
public void cardInserted(CardTerminalEvent ctEvent) {
...
}

public void cardRemoved(CardTerminalEvent ctEvent) {
...
}
...
}
```

列表 2、一个异步 OCF 应用程序的典型结构

当一张卡片被插入时，运行时间调用 cardInserted ()方法，并且当卡片被拔出时，运行时间调用 cardRemoved()方法。在下面的代码中，插入卡片初始化小应用程序代理的创建，并且拔出卡片触发小应用程序代理的清除。代码列表还说明了信用卡余额请求代理。

```
import opencard.core.event.CTListener;
import opencard.core.event.CardTerminalEvent;
import opencard.core.service.SmartCard;
import opencard.core.service.CardService;
...

public class MyHostSideApp implements CTListener
{
   public void MyHostSideApp() {
```

```
   try {
      // Initialize the framework
      SmartCard.start ();
      // Register this as a Card Terminal Event Listener
      CardTerminalRegistry.getRegistry().addCTListener(this);
   } catch (Exception e) {
   // Handle error.
   ...
   }
}

/**
* Card insertion event. Get new card and card service
* @param ctEvent The card insertion event.
*/

public void cardInserted(CardTerminalEvent ctEvent) {
   try {
      // Get a SmartCard object
      card = SmartCard.getSmartCard(ctEvent);
      // Get the card proxy instance.
      myCardProxy = (MyCardProxy)
      card.getCardService(MyCardProxy.class, true);
   } catch (Exception e) {
```

```
    e.printStackTrace();
    }
}

/**
* Card removal event. Invalidate card and card service.
* @param ctEvent The card removal event.
*/
public synchronized void cardRemoved(CardTerminalEvent ctEvent) {
    card = null;
    myCardProxy = null;
    // Initialize the framework
    SmartCard.shutdown();
}

/**
* Get balance from the smart card.
*/
public int getBalance() {
    try {
        // Get mutex to prevent other Card Services from modifying
        // data. Delegate the call to the applet proxy.
        card.beginMutex();
        return Integer.parseInt(myCardProxy.getBalance());
```

```
    } catch (Throwable e) {
    return 0;
    } finally {
        // End mutual exclusion
        card.endMutex();
    }
}
...
}
```

列表 3.一个增强的基于监听者的 OCF 应用程序

接下来是小应用程序代理的选节。OCF 应用程序委托服务调用到小应用程序代理，实现复杂的 APDU 管理模块：

```
public class MyCardProxy extends AppletProxy {
    // My APDU definitions.
    final static byte MyAPPLET_CLA = (byte)0x80;
    final static byte VERIFY_INS = (byte)0x20;
    final static byte GET_BALANCE_INS = (byte) 0x30;
    final static short GET_BALANCE_RESPONSE_SZ = 2;
    protected final static int OK = 0x9000;
    final static short SW_PINVERIFY_FAILED = (short)0x6900;

    /**
    * Reusable command APDU for getting an information
```

```
* entry field.

*/



private CommandAPDU getBalanceAPDU = new CommandAPDU(14);



...



/** Application identifier of the BusinessCard applet */



private static final ApplicationID MY_CARD_AID =

   new ApplicationID(new byte[] { (byte)0xD4,

      (byte)0x55,

      (byte)0x00,

      (byte)0x00,

      (byte)0x22,

      (byte)0x00,

      (byte)0x00,

      (byte)0x00,

      (byte)0xFF});

/**

   * Create a MyCardProxy instance.

   *

   * @param scheduler The Scheduler from which channels

   * have to be obtained.
```

```
    * @param card The SmartCard object to which this
    * service belongs.
    * @param blocking Currently not used.
    *
    * @throws opencard.core.service.CardServiceException
    * Thrown when instantiation fails.
*/

protected void initialize(CardServiceScheduler scheduler,
    SmartCard card, boolean blocking)
throws CardServiceException {
    super.initialize(MY_CARD_AID, scheduler, card, blocking);
    try {
        // Allocate the card channel. This gives us
        // exclusive access to the card until we release the
        // channel.
        allocateCardChannel();

        // Get the Card State.
        ...

    } finally {
    releaseCardChannel();
}
```

```
}


  /**
     * Gets the balance.
     * @return The balance.
  */


  public String getBalance()
     throws CardServiceInvalidCredentialException,
     CardServiceOperationFailedException,
     CardServiceInvalidParameterException,
     CardServiceUnexpectedResponseException,
     CardServiceException,
     CardTerminalException {
        try {
           allocateCardChannel();


           // Set up the command APDU and send it to the card.
           getBalanceAPDU.setLength(0);
           getBalanceAPDU.append(MyAPPLET_CLA); // Class
           getBalanceAPDU.append(GET_BALANCE_INS); // Instr'n
           getBalanceAPDU.append((byte) 0x00); // P1
           getBalanceAPDU.append((byte) 0x00); // P2
           getBalanceAPDU.append((byte) 0x00); // Lc
```

```
        getBalanceAPDU.append((byte) 0x00); // Le


        // Send command APDU and check the response.
        ResponseAPDU response =
        sendCommandAPDU(getCardChannel(), MY_CARD_AID, getBalanceAPDU);
        switch (response.sw() & 0xFFFF) {
          case OK :
            return new String(response.data());
          default :
            throw new
            CardServiceUnexpectedResponseException("RC=" + response.sw());
        }
      } finally {
      releaseCardChannel();
    }
  }


  ...
}
```

列表 4、一个小应用程序代理示例

你可以在 OpenCard Framework 1.2 Programmer ' s Guide（http://www.opencard.org/docs/pguide/PGuide.html）中获得更多的关于 OCF 使用的消息。还引用了 OpenCard Framework 引用实现（http://www.opencard.org/index-download.shtml）中的示例文件。

本文中未涉及，但是值得一提的是称为 Global Platform（http://www.globalplatform.org）的主机端应用程序编程接口。Global Platform Card Committee 提供了一组补充 Java Card 应用程序编程接口的卡片应用程序编程接口，提供了卡片管理特性。这些规范都在 2.1.1 版本中。

3、RMI 客户端编程接口

JavaCard RMI 客户端应用编程接口

前面你已经学到了如何编写一个基于 JCRMI 的小应用程序。如果你的 Java Card 小应用程序基于 JCRMI，你可以使用 Java Card RMI 客户端应用编程接口编写一个主应用程序，访问智能卡中保存的小应用程序对象。

为了智能卡管理和访问，JCRMI 客户端应用编程接口需要一个卡片终端和诸如刚刚描述的 OpenCard Framework 这样的服务应用编程接口。

当我们把这两个应用程序编程接口放在一起，我们得到一个非常简单、非常完整的面向对象编程模型，有以下几个优点：

1、不必知道智能卡和读卡器的细节

2、不必知道低级的 APDU 通讯

3、便于设计和维护代码，缩短开发时间

JCRMI 客户端应用编程接口在下面的程序包中定义：

1 com.sun.javacard.javax.smartcard.rmiclient 包含核心 JCRMI 客户端应用编程接口。它定义：

1）JCRMI 代码程序用来访问智能卡的 CardAccessor 接口。

2）用于 JCRMI 程序生成实现的基本类 CardObjectFactory 类。这个类的实例与一个 Java Card 小应用程序选择的会话有关。

3）客户端应用程序使用的 JavaCardRMIConnect 类，用于初始化一个 JCRMI 会话并且获得初始的远程引用。

4）许多 Java Card 异常子类，例如 APDUExceptionSubclass、CardExceptionSubclass、

CardRuntimeExceptionSubclass、CryptoExceptionSubclass、ISOExceptionSubclass、PINExceptionSubclass、PINException、ServiceExceptionSubclass、SystemExceptionSubclass、TransactionExceptionSubclass 和 UserExceptionSubclass。

2 javacard.framework 定义了许多客户端上的许多可以被再次抛出的 Java Card 异常：APDUException、CardException、CardRuntimeException、ISOException、PINException、SystemException、TransactionException 和 UserException。

3 javacard.framework.service 定义了 ServiceException，描述与服务框架有关的异常。

4 javacard.security 定义了描述一个有关加密异常的 CryptoException。

4、生成 RMI 客户端程序

生成 RMI 客户端程序

你可以使用标准的 Java RMI 编译程序（rmic）生成客户端程序。你必须使用下面格式的命令运行 rmic，用于你的小应用程序中的每个远程类：

rmic -v1.2 -classpath path -d output_dir class_name

在这里：

1 -v1.2 是一个 Java Card RMI 客户端框架所需要的标志。

2 - classpath 路径标明到远程类的路径。

3 output_dir 是存放结果程序的目录。

4 class_name 是远程类的名称。

然而，推荐生成 RMI 客户端程序的方法是使用 J2SE SDK 1.3 中的动态代理生成机制。如果当你选择 JCRMI 小应用程序的时候使用 CardObjectFactory 子类型 JCCardProxyFactory 的话，JavaCard RMI 客户端应用编程接口的 2.2 版本将为你自动生成程序，你不必再生成程序。这个方法在列表 5 中说明。

用法限制

因为 Java Card 是一个有限制的运行时环境，我们可以发现 JCRMI 的限制。Java Card 不支持序列化，并且 JCRMI 参数和返回值也有限制：

1、每个到远程方法的参数必须是 Java Card 支持的类型之一，不包括 char、double、float、long 或多维数组。对于 int 的支持是自选的。

2、任何远程方法的返回值必须是支持的类型之一，或者 void，或者一个远程接口类型。

JCRMI 客户应用程序

一个 JCRMI 客户应用程序类似你前面看到的 OCF 主应用程序，因为 JCRMI 客户端应用编程接口依靠 OCF 用于卡片管理和通讯。

下面的代码片断首先初始化 OCF，并且等待智能卡插入。它然后创建一个 OCFCardAccessor 实现，用于把我们的 JCRMI 连接到卡片上，如果必要，客户端程序动态生成，小应用程序被选中，我们取得远程引用，最后我们产生到 getBalance()的我们的远程调用：

```
...
try {
  // Initialize OCF
  SmartCard.start();

  // Wait for a smart card
  CardRequest cr = new CardRequest(CardRequest.NEWCARD, null,OCFCardAccessor.class);
  SmartCard myCard = SmartCard.waitForCard(cr);

  // Get an OCFCardAccessor for Java Card RMI
  CardAccessor ca = (CardAccessor)
  myCard.getCardService(OCFCardAccessor.class, true);
```

```
    // Create a Java Card RMI instance
    JavaCardRMIConnect jcRMI = new JavaCardRMIConnect(ca);

    // Create a Java Card Proxy Factory that is used for dynamic
    // proxy generation.
    CardObjectFactory factory = new JCCardProxyFactory(ca);
    // select the Java Card applet
    jcRMI.selectApplet(MY_APPLET_AID, factory);

    // Get the initial reference
    MyRemoteInterface myRemoteInterface = (MyRemoteInterface) jcRMI.getInitialReference();
    if(myRemoteInterface == null) {
       throw new
       Exception("Received null instead of the initial ref");
    }

    // Invoke the remote getBalance() method
    short balance = myRemoteInterface.getBalance();
}
catch(UserException e) {
    // Handle exception
    ...
}
catch (Exception e){
```

```
    // Handle exception
    ...
} finally {
    // Clean up
    try{
        SmartCard.shutdown();
    }catch (Exception e){
        System.out.println(e);
    }
}
```

列表 5、示例 JCRMI 客户端

如你所见，你必须让代码简化。

5、用于 J2ME 的安全和信任服务编程接口

用于 J2ME 的安全和信任服务编程接口

SATSA 是一套用于 J2ME 的新的可选程序包，定义一个客户端应用编程接口来访问安全元素：例如智能卡这样的设备。在本节，我将仅仅介绍 SATSA 的通讯应用编程接口；本文中将不涉及 SATSA PKI 和加密应用程序编程接口。

SATSA 通讯应用编程接口被分解成下面几部分：

1 SATSA-APDU 定义一个应用编程接口，用于和遵循 ISO-7816-4 的智能卡通讯。这个可选程序包由单独的 javax.microedition.io.APDUConnection 程序包组成。

2 SATSA-JCRMI 定义一个 Java Card RMI 客户端应用编程接口。这个可选程序包由下面的 Java 程序包组成：

a）javax.microedition.io.JavaCardRMIConnection

b) javax.microedition.jcrmi.RemoteRef

c) javax.microedition.jcrmi.RemoteStub

d) java.rmi.Remote

e) java.rmi.RemoteException

f) javacard.framework.service.ServiceException

g) javacard.framework.CardRuntimeException

h) javacard.framework.ISOException

i) javacard.framework.APDUException

j) javacard.framework.CardException

k) javacard.framework.PINException

l) javacard.framework.SystemException

m) javacard.framework.TransactionException

n) javacard.framework.UserException

o) javacard.security.CryptoException

SATSA 把 J2ME 和 Java Card 平台紧密的结合在一起。SATSA 使用 CLDC 1.0 Generic Connection Framework (GCF)用于在基于 J2ME 的设备和一个如下面所图解的智能卡之间通讯：

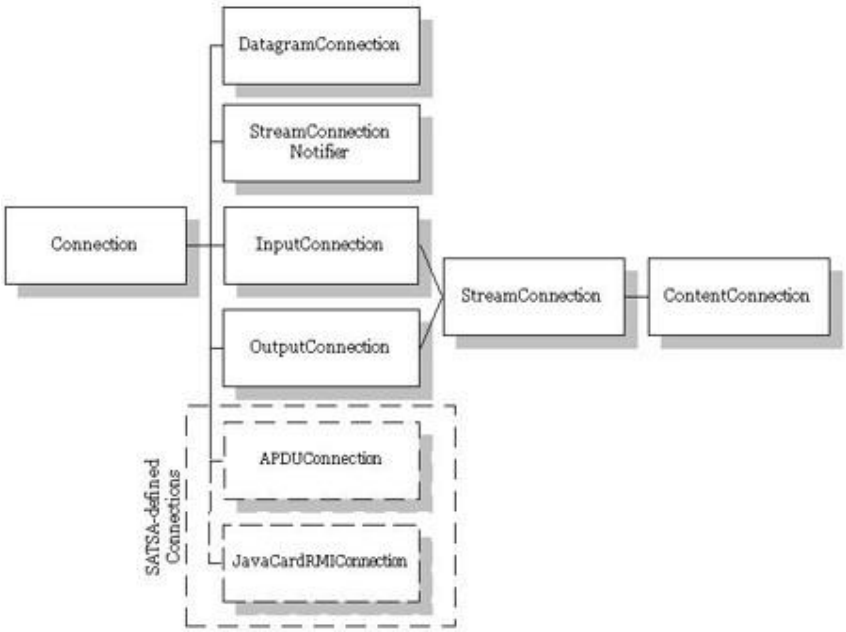


Figure 3. 普通连接和 SATSA 连接

因为 SATSA 基于 GCF，开发使用手机上的智能卡的 MIDlet 相对不容易，但是对于 J2ME 开发者来说很熟悉。

SATSA 考虑到用于 Java Card 应用程序的两个总体变成模型：APDU-消息传递模型和 Java Card RMI 面向对象分布式模型。SATSA 为了这些模型中的每一个定义了一个新的 GCF 连接类型：

1 APDUConnection 允许一个 J2ME 应用程序使用 ISO-7816 APDU 协议来与智能卡应用程序交换 APDU。

2 JavaCardRMIConnection 允许一个 J2ME 应用程序使用 Java Card RMI 来调用智能卡上的远程方法。

6、指定 SATSA 连接类型

指定 SATSA 连接类型

所有的 GCF 连接都是使用 Connector.open()方法创建。Connector.open()的一个参数是指明创建的连接类型的 URL。CLDC GCF 使用下面的格式定义这个 URL 为一个字符串：

scheme:[target][params]

在这里：

1 scheme 是创建的连接类型（和使用的协议）。

2 target 一般是某种网络地址。

3 params 是可选参数，通过分号隔开。

对于 SATSA，URL 的格式是：

protocol:[slotID]; AID

在这里：

1 protocol 要么是 apdu，用于基于 APDU 的连接，或者是 jcrmi，用于基于 JCRMI 的连接。

2 slotID 是指明卡片插入的插孔号。slotID 字段是可选的；默认值为 0。

3 AID 是用于智能卡应用程序的应用程序标识符。AID 是一个由句号分隔开的 5 到 16 个十六进制字节值得字符串；例如，" A0.0.0.67.4.7.1F.3.2C.3"。

7、使用 APDUConnection

使用一个 APDUConnection

一个 APDUConnection 定义方法，允许我们使用 GCF 与遵循 ISO-7816 的卡片通讯。它定义三个方法：

1 enterPIN()提示用户输入一个个人识别号码。

2 exchangeAPDU()与智能卡应用程序交换 APDU。这个调用直到一个响应从智能卡返回的时候（或者处理被中断）才会阻断。

3 getATR()返回智能卡发送的 Answer To Reset(ATR)消息，作为重置操作的响应。

下面的代码片断显示如何打开一个 APDUConnection，如何关闭它，以及如何交换一个命令 APDU 并且接收一个响应 APDU：

```
...
try {
   // Create an APDUConnection
   String url = "apdu:0;AID=A1.0.0.67.4.7.1F.3.2C.5";
   APDUConnection ac = (APDUConnection) Connector.open(url);

   // Send a command APDU and receive a response APDU
   byte[] responseAPDU = ac.exchangeAPDU(commandAPDU);
   ...

   // Close connection.
   ac.close();
} catch(IOException e){
   ...
}
...
```

列表 6、使用 SATSA-APDU

SATSA 使 APDU 通讯简单化。注意，这个命令和响应 APDU 的格式和你在本系列第二部分《JavaCard 小应用程序开发教程》看到的相同，告诉你如何编写一个基于 APDU 消息传送的 JavaCard 小应用程序。

8、JavaCardRMIConnection

使用一个 JavaCardRMIConnection

一个 JavaCardRMIConnection 定义方法，允许我们使用 Java Card RMI 程序设计模型。JavaCardRMIConnection 定义方法 getInitialReference()，返回初始远程引用的程序对象。

```
...
try {
   // Create a JavaCardRMIConnection
   String url = "jcrmi:0;AID=A0.0.0.67.4.7.1F.3.2C.3";
   JavaCardRMIConnection jc = (JavaCardRMIConnection)
   Connector.open(url);
   MyRemoteObject robj = (MyRemoteObject)
   jc.getInitialReference();
   ...
   short balance = robj.getBalance();
   ...
   // Close connection
   jc.close();
} catch (Exception e) {
   ...
}
...
```

列表 7、使用 SATSA-JCRMI

采用上面所说的 SATSA-JCRMI 应用编程接口，允许我们调用本系列第二部分(《JavaCard 小应用程序开发教程》)定义的 getBalance()方法，向你说明如何编写一个基于 RMI 的 JavaCard 小应用程序。

<OVER>