
ET199 超级多功能锁用户手册 身份认证篇

V1.0 版

版权所有© 2007-2013 北京坚石诚信科技有限公司
<http://www.jansh.com.cn>

坚石诚信科技有限公司

软件开发协议

坚石诚信科技有限公司（以下简称坚石）的所有产品，包括但不限于：开发工具包，磁盘，光盘，硬件设备和文档，以及未来的所有订单都受本协议的制约。

1. 许可使用

您可以将本软件合并、连接到您的计算机程序中，但其目的只是如使用手册中描述的那样保护您的程序或进行网络身份认证。您可以以备份为目的复制合理数量的拷贝。

2. 禁止使用

除在条款 1 中特别允许的之外，不得复制、反向工程、反汇编、反编译、修改、增加、改进软件、硬件和产品的其它部分。禁止对软件和产品的任何部分进行反向工程，禁止推导软件的源代码。禁止使用产品中的磁盘或光盘来传播、存储非本产品的原始内容的任何信息或由坚石提供的产品的任何升级。禁止将软件放在公共服务器上传播。

3. 有限担保

坚石保证在自产品发给您之日起的 12 个月内，在正常的使用情况下，硬件和软件存储介质没有重大的工艺和材料上的缺陷。

4. 修理限度

当根据本协议提出索赔时，坚石唯一的责任就是根据实际情况，免费进行替换或维修。坚石对被替换下来的任何产品部件都享有所有权。

保修索赔单必须在担保期内写好，在发生故障 14 天内连同令人信服的证据交给坚石诚信科技公司。往返运费需由客户承担。

除了在本协议中保证的担保之外，坚石诚信科技公司不再提供特别的或隐含的担保，也不再对本协议中所描述的产品负其它责任，包括它们的质量，性能和对某一特定目的的适应性。

5. 责任限度

不管因为什么原因，不管是因合同中的规定还是由于刑事的原因，包括疏忽的原因，

而使您及任何一方受到了损失，由我方产品所造成的损失或该产品是起诉的原因或与起诉有间接关系，坚石诚信科技公司对您及任何一方所承担的全部责任不超出您购买该产品所支付的货款。在任何情况下，坚石诚信科技公司对于由于您不履行责任所导致的损失，或对于数据、利润、储蓄或其它的后续的和偶然的损失，即使坚石诚信科技公司被建议有这种损失的可能性，或您根据第 3 方的索赔而提出的任何索赔均不负责任。

6. 协议终止

当您不能遵守本协议所规定的条款时，将终止您的许可和本协议。但条款 2，3，4，5 将继续有效。

北京坚石诚信科技有限公司

地址：北京市海淀区学清路 9 号汇智大厦 B 座二层

邮编：100192

电话：010—82730011（总机）

传真：010—82737938

网址：<http://www.jansh.com.cn>

目 录

快速入门及注意事项	1
第一章 ET199 简介	2
1.1 关于 ET199 超级多功能锁	2
1.2 软件保护	2
1.3 身份认证	3
1.4 ET199 支持的平台	3
第二章 身份认证注意事项	4
2.1 ET199 硬件上的优越性	4
2.2 ET199 管理员 PIN (SO PIN) 和用户 PIN (User PIN)	4
2.3 公有区和私有区	4
2.4 ET199 中间件	5
第三章 ET199 中间件的安装和卸载	6
3.1 准备安装 ET199	6
3.2 安装 ET199 运行库	6
3.3 卸载 ET199 运行库	9
第四章 ET199PKI 工具	13
4.1 ET199 用户版管理工具	13
4.1.1 未插入 Token 的界面	13
4.1.2 插入 Token 的界面	13
4.1.3 管理工具的按钮功能	14
4.2 ET199 管理员版管理工具	27
4.3 ET199 配置工具	34
4.4 ET199 初始化设置工具	36
4.4.1 初始化为空锁	36
4.4.2 初始化为 PKI 格式	37
第五章 常用功能需求举例	39
5.1 PKCS#11 接口	39
5.1.1 打开和关闭硬件	39
5.1.2 获取硬件信息	42
5.1.3 验证 SO PIN 和 User PIN	43
5.1.4 初始化	43
5.1.5 检测硬件插拔	48
5.1.6 得到 PIN 码的重试次数和修改 TokenName	51
5.1.7 读写数据	55
5.1.8 对称加解密	60

5.1.9 RSA 操作	66
5.1.10 低级初始化	79
5.1.11 解锁 USER PIN	79
5.1.12 修改 USER PIN 和 SO PIN	80
5.2 CAPI 接口	81
5.2.1 枚举 ET199 硬件中的证书	81
5.2.2 使用 CAPI 接口验证用户 PIN (User PIN)	87
5.2.3 签名和验签	88
5.2.4 RSA 加解密	96
附录 A	100

快速入门及注意事项

- ET199硬件有三种状态：空锁，加密锁（加密锁格式），身份认证锁（PKI格式）。
 - 出厂默认为加密锁格式，当需要使用身份认证功能时，需要使用《ET199超级多功能锁用户手册—加密锁篇》中5.2.2节中介绍的工具，将ET199格式化为空锁，然后再使用PKI工具（见4.2节）进行PKI格式化。
 - 将PKI格式的ET199恢复为加密锁时，需要先使用PKI设置工具（见4.2.1节）格式化为空锁，然后再使用《ET199超级多功能锁用户手册—加密锁篇》中5.2.2节中介绍的工具，将ET199格式化为加密锁。
- 如果您需要使用ET199多功能锁进行开发或者测试请与我公司联系。开发需要的相关资料 and 软件接口请从我公司网站上下载。网址：<http://www.jansh.com.cn>
- ET199多功能锁是USB接口的HID设备，支持Win98SE以上的Windows操作系统，包括32位和64位系统，不需要安装额外的驱动程序。
- ET199多功能锁具有64位全球唯一硬件序列号，及软件加密功能和身份认证功能于一身，适用于软件保护和安全系统身份认证。
- ET199工作温度：0℃—70℃。硬件擦写次数10万次，读没有限制。
- 在使用ET199PKI功能前，需要先安装PKI运行时环境（即安装redist\cn 目录下的ET199-SimpChinese.exe），安装时需要有操作系统的管理员权限。
- 当管理员PIN锁死时，可以使用PKI设置工具重新初始化（见4.2.2节）。

第一章 ET199 简介

1.1 关于 ET199 超级多功能锁

ET199超级多功能锁（以下简称ET199）是世界上第一款使用智能卡硬件，将软件保护功能和身份认证功能合二为一的产品。ET199采用无驱设计，功能强大，价格实在，能够使用在各种领域和众多的用途中，达到一锁多能的目的。

1.2 软件保护

在软件加密方面，ET199采用了多种先进的关键加密技术，是目前软件保护领域中最安全的保护产品。

- 硬件上的安全性

很多软件被盗版都是由于硬件被复制，不能抵御破解攻击造成的。一个高强度的加密锁首先应有牢固的硬件基础。ET199采用了安全强度最高的智能卡芯片，硬件不能被复制，多重安全级别，并且集成了16位CPU，8KRAM，64K存储空间等模块。ET199在具有如此强大和安全功能的同时，改进了生产工艺，极大降低了生产成本，从而使广大的软件厂商不必再花费高额的加密锁成本就能够使用上智能卡型加密锁，从根本上提高了软件的加密强度。

- 硬件上的兼容性

ET199采用无驱设计，使用高速HID协议，在WIN98二版以上的操作系统中不需要安装驱动程序，彻底解决了由于驱动安装而给软件开发商带来的各种各样的问题。无驱的同时，ET199使用高速协议，对应用软件没有任何速度上的影响。正因为ET199在硬件上卓越的兼容性，使用ET199几乎不需要任何维护，大大节省了维护成本。

- 先进的软件保护

ET199就像一个小型计算机，能够完成众多以前只能在PC上完成的功能，如复杂的浮点运算。软件开发商可以将自己的程序代码转移到ET199中运行，计算机中没有任何程序的痕迹，同时ET199内部的代码任何人也获取不到。软件开发商发行的是一个不完整的软件，只有与ET199结合，软件才能够正确执行。由于硬件上的安全性，可以认为ET199是不可破解的。

1.3 身份认证

在身份认证方面，ET199可以做为数字证书的安全载体，敏感数据都被安全地保存在ET199的安全存储区域中，未授权用户是无法接触到这些信息的。数据的签名和加密操作全部在ET199内部完成，私钥从生成的时刻起就一直保存其中，可有效的杜绝黑客程序的攻击。ET199的安全性还在于其使用的加密算法都是被广泛公开，业界公认的，经受了多年考验的标准算法。同时，一流的芯片封装工艺也保证了芯片内数据的安全性。

ET199能够硬件产生512，1024和2048位的RSA密钥对，硬件实现RSA的各种运算。其采用16位的CPU，结合EnterSafe中间件，使用高速无驱的通讯技术，能够十分迅速的完成各种PKI应用的操作。

ET199 提供符合业界广泛认可的 PKCS#11 和 Microsoft CryptoAPI 两种标准的接口。任何兼容这两种接口的应用程序，都可以立即集成 ET199 进行使用。同时，ET199 也针对多个第三方的软件产品进行了兼容性优化。此外，ET199 内置大容量的安全存储器，可以同时存储多个数字证书和用户私钥及其他数据。也就是说，多个 PKI 应用程序可以共用同一个 ET199。

1.4 ET199 支持的平台

- Windows 98 SE
- Windows Me
- Windows 2000
- Windows XP
- Windows 2003
- Windows Vista
- Windows 2008
- Windows 7
- Windows 8

包括各版本补丁，以及相关的 32 位和 64 位系统

第二章 身份认证注意事项

在您使用 ET199 进行身份认证时，请务必认真阅读下面的注意事项，这些都是您在进行身份认证时将了解和用到的。另外您还可以参见 EnterSafe 提供的丰富的 PKI 文档，包括：PKI 基础知识、密码学体系、使用 CAPI 和 PKCS#11 接口开发、CAPI 应用、Netscape 应用、WORD 应用等。

2.1 ET199 硬件上的优越性

- ET199 采用高强度的智能卡安全芯片，硬件不可复制。
- ET199 的用户空间为 64K。
- 硬件擦写次数 10 万次，保存 10 年。
- 工作温度：0℃—70℃
- 64 位全球唯一硬件序列号。
- ET199 采用高速无驱设计，将高速和应用简便集成到一体。
- 硬件实现 512、1024、2048 位的 RSA 运算功能。
- ET199 采用加密的 USB 端口通讯，加密算法在硬件中，保证了传输数据的安全性。

2.2 ET199 管理员 PIN (SO PIN) 和用户 PIN (User PIN)

ET199有3种状态：匿名，用户PIN验证通过，管理员PIN验证通过

- 管理员PIN（以下简称SO PIN）：验证管理员PIN通过后，可以对ET199进行PKI初始化和解锁用户PIN。
- 用户PIN（以下简称User PIN）：验证用户PIN通过后，可以访问写在私有区中的数据和使用RSA私钥进行运算。

如果设置了管理员PIN和用户PIN的重试次数。当使用者连续输入错误的次数达到了最大限制的次数，就会被锁死。用户PIN被锁死后可以使用管理员进行解锁，当管理员PIN被锁死后，只能重新进行PKI初始化。

2.3 公有区和私有区

经过PKI初始化后，ET199中64K用户空间分为公有区和私有区。

- 公有区：存放一般数据的区域，不需要进行用户PIN验证就可以读写。一般数

字证书中的证书数据和公钥数据存放在这里

- 私有区：存放保密数据的区域，需要进行用户PIN验证才可以读写。签章数据，数值证书的私钥的属性信息存储在这里。

注：私钥不是存储在这两个区域中的。

2.4 ET199 中间件

在使用 ET199 进行身份认证时需要先安装中间件，中间件是进行 PKI 身份认证时需要的运行环境，包括 MS CAPI 接口和 RSA PKCS#11 接口。

第三章 ET199 中间件的安装和卸载

本章包含以下主题：

- 准备安装 ET199
- 安装 ET199 运行库
- 卸载 ET199 运行库

3.1 准备安装 ET199

在开始安装 ET199 运行库之前，请确定满足以下要求：

- 操作系统为以上列出的操作系统
- 主机上带有至少一个 USB 端口
- 计算机的 BIOS 支持 USB 设备，并且在 CMOS 设置中将 USB 支持功能打开
- USB 设备延长线或 USB Hub（可选）
- ET199 Token

3.2 安装 ET199 运行库

使用ET199之前,您必须安装ET199运行库。执行redist\cn\ET199-SimpChinese.exe后，出现如图 1所示的欢迎界面窗口：

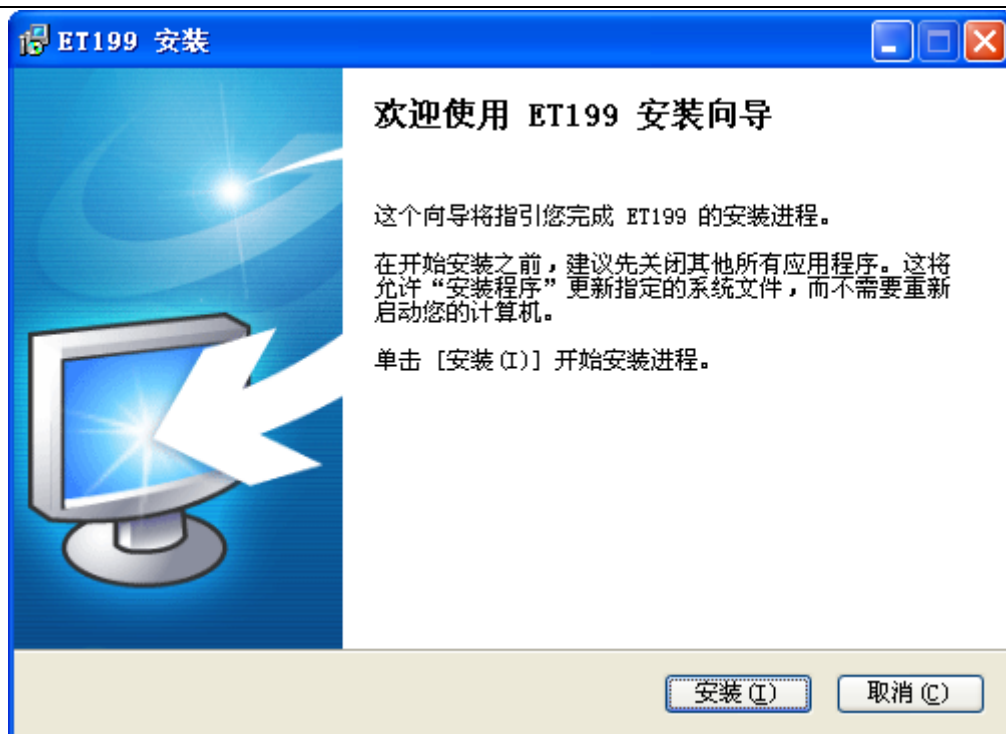


图1 欢迎界面窗口

在欢迎界面中，点击“安装”即进入正在安装界面，如图 2所示：

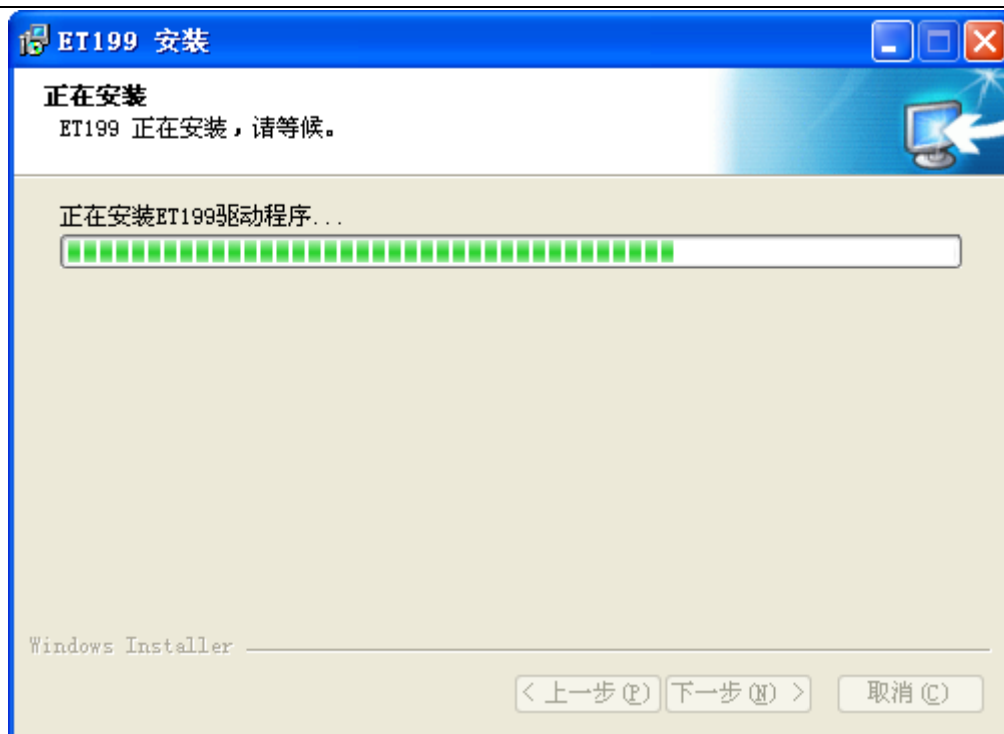


图2 正在安装界面

ET199 安装完成以后，出现图 3所示的窗口，如这时遇到一些杀毒等软件弹出的提示框，应选择允许：

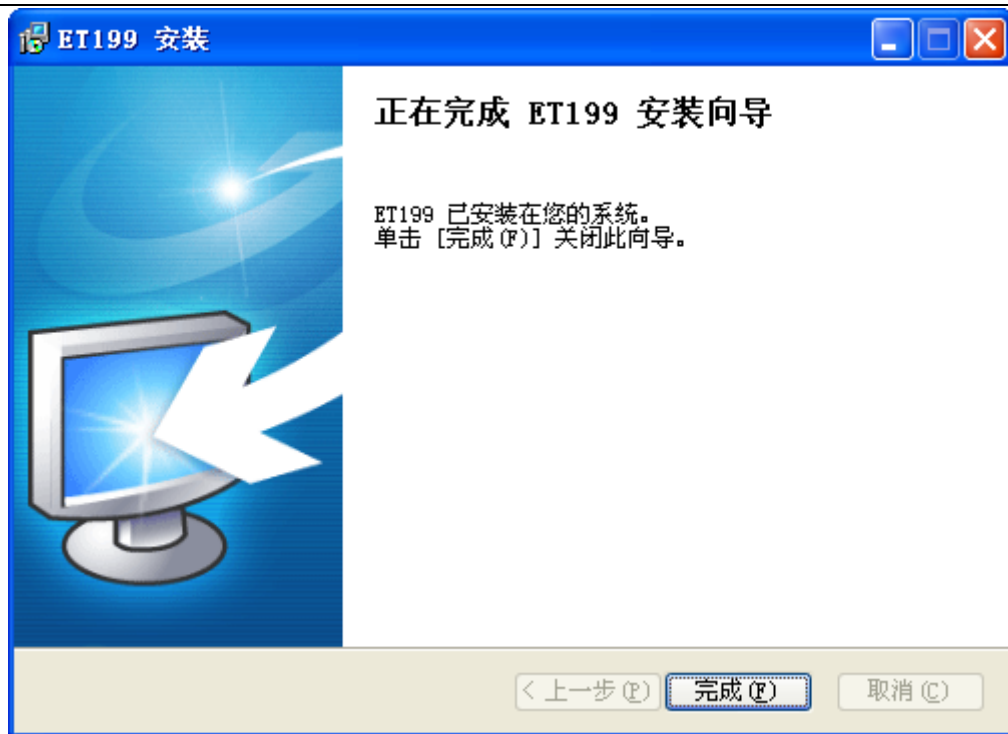


图3 完成安装界面

点击“完成”，就完成了安装。

3.3 卸载 ET199 运行库

安装了 ET199 运行库之后，您可以通过以下方式卸载：

- 使用控制面板中的“添加/删除程序”卸载

使用“开始”→“设置”→“控制面板”的打开控制面板，然后双击“添加/删除程序”打开“添加/删除程序 属性”对话框，在“安装/卸载”的列表中选中“ET199 (仅用做移除)”项，然后单击“更改/删除”按钮。

- 从开始菜单中卸载

选择“开始”→“程序”→“EnterSafe”→“ET199”→“卸载”。

以上两种方式都会启动 ET199 的解除安装向导，如图 4所示：

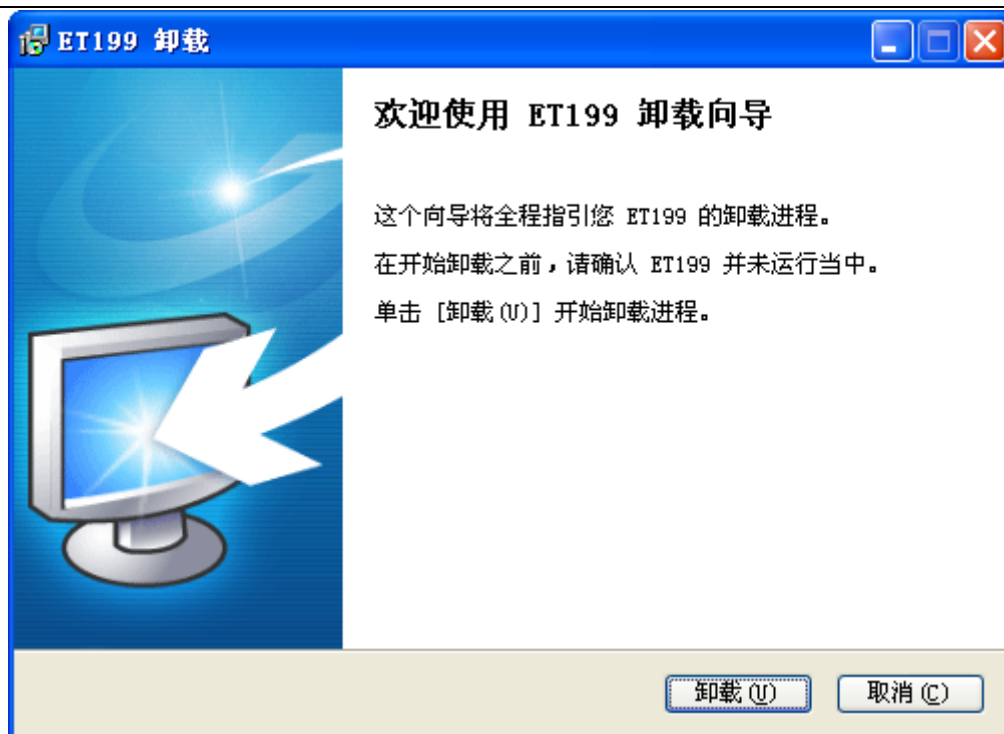


图4 解除安装向导界面

单击“卸载”按钮，解除安装向导会自动完成卸载操作，卸载过程中出现如图 5 的窗口：

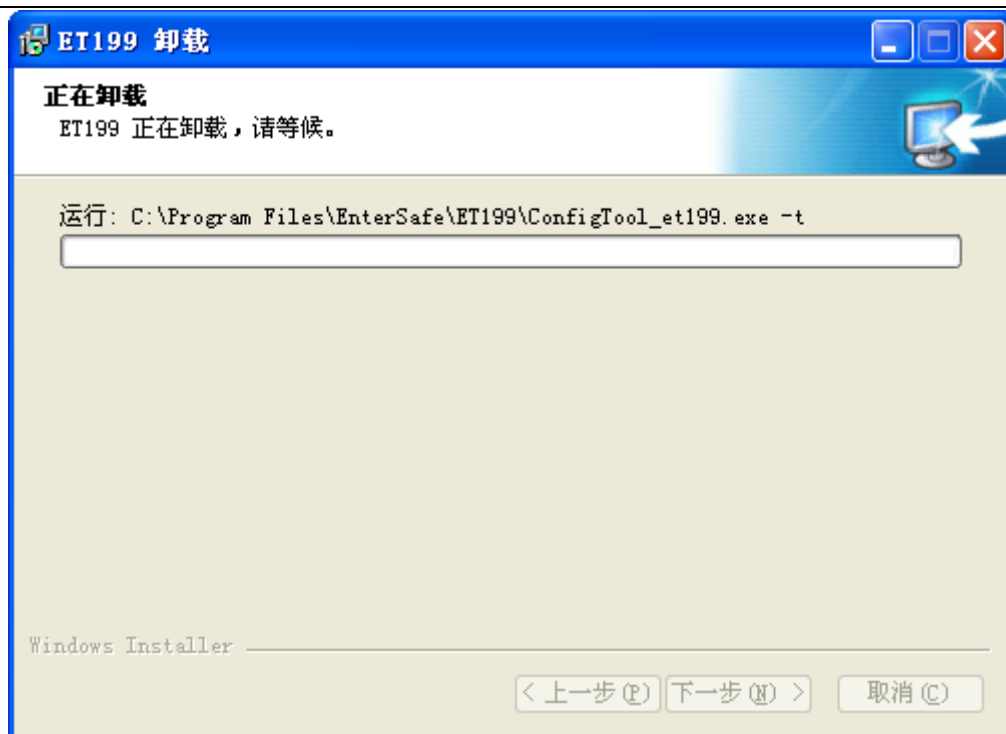


图5 卸载过程界面

卸载完成以后，出现如图 6 所示的窗口：

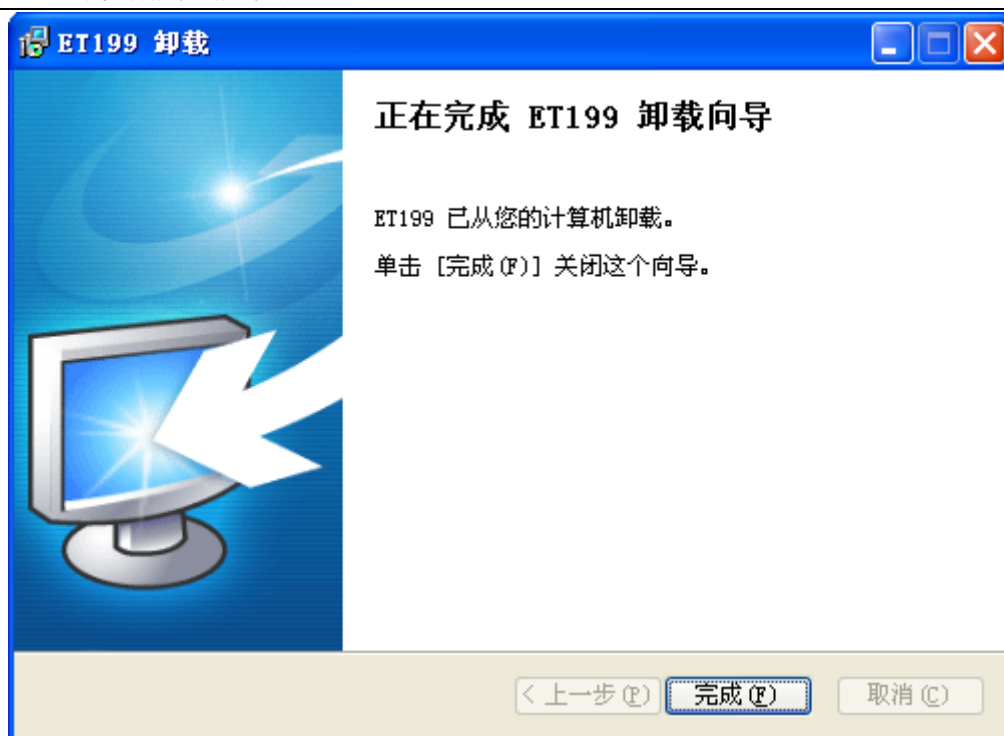


图6 完成卸载界面

单击“完成”按钮系统会从您的计算机上解除 ET199 的安装，并关闭解除安装向导。

第四章 ET199PKI 工具

4.1 ET199 用户版管理工具

注意：在使用本工具前，需要先安装 PKI 运行时环境（即安装 redist\cn 目录下的 ET199-SimpChinese.exe），安装时需要有操作系统的管理员权限。另外，在使用 ET199 之前必须将 ET199 格式化成 PKI 格式。

4.1.1 未插入 Token 的界面

可以在“开始”→“程序”→“EnterSafe”→“ET199”中找到管理工具的快捷方式，点击管理工具的快捷方式启动管理工具，出现界面如图 7 所示：

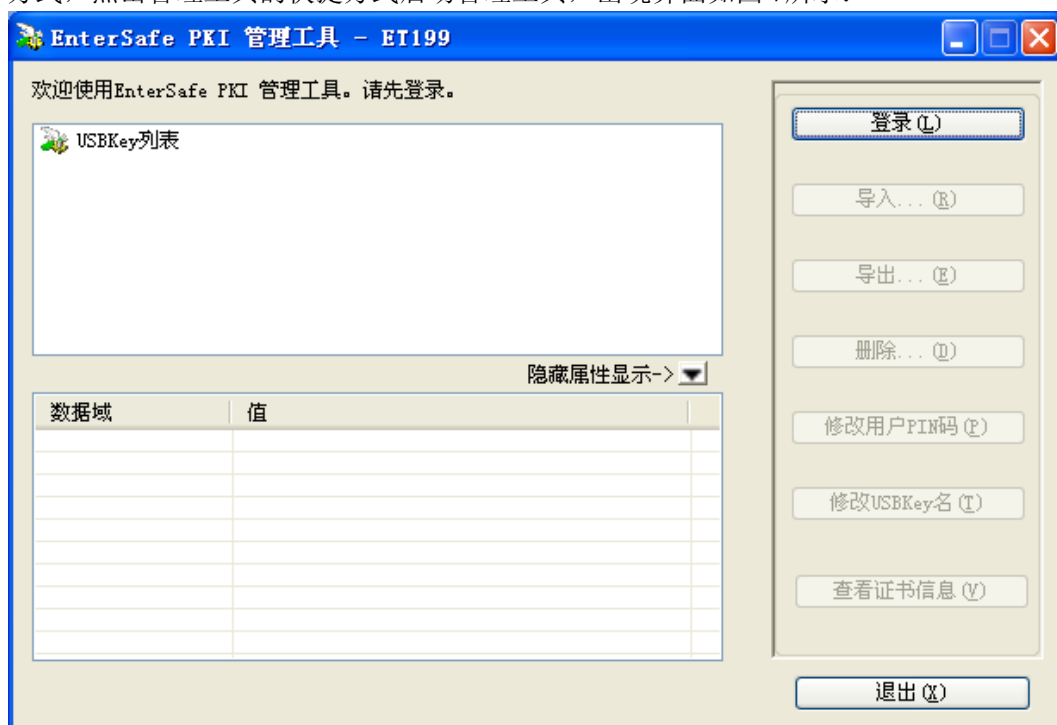


图7 未插入 Token 的界面

4.1.2 插入 Token 的界面

现在在您计算机的 USB 接口中插入一个名称为“ET199”的 Token，那么管理工具就能自动识别出这个 Token 的基本信息，并且会呈现出如图 8所示的界面：

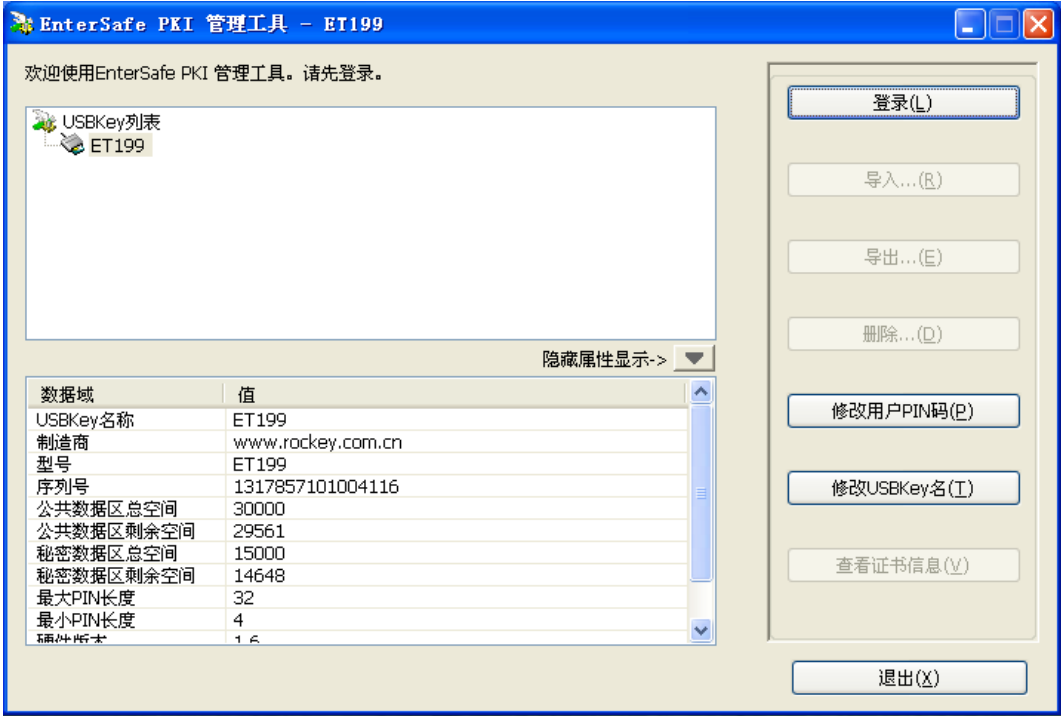


图8 插入 Token 的界面

4.1.3 管理工具的按钮功能

管理工具的按钮包括：“登录”、“导入”、“导出”、“删除”、“修改用户 PIN 码”、“修改 USBKey 名”、“查看证书信息”和“退出”，如图 8右侧所示的按钮。

● 登录

在管理工具主界面上点击“登录”按钮，弹出 PIN 码输入框，如图 9所示：

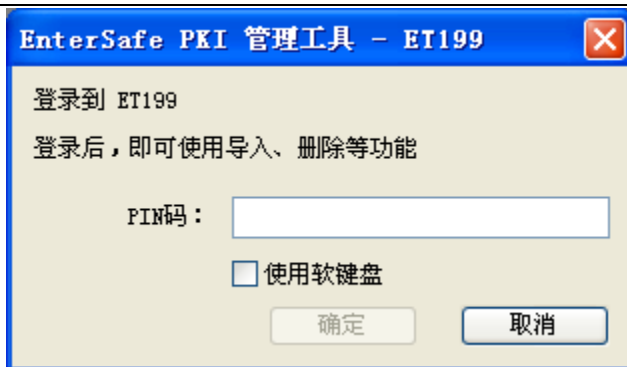


图9 PIN 码输入框

在此用户可以通过勾选“使用软键盘”复选框来使用软键盘输入 PIN 码，以避免木马程序对用户输入的 PIN 码的监控，如下图所示：

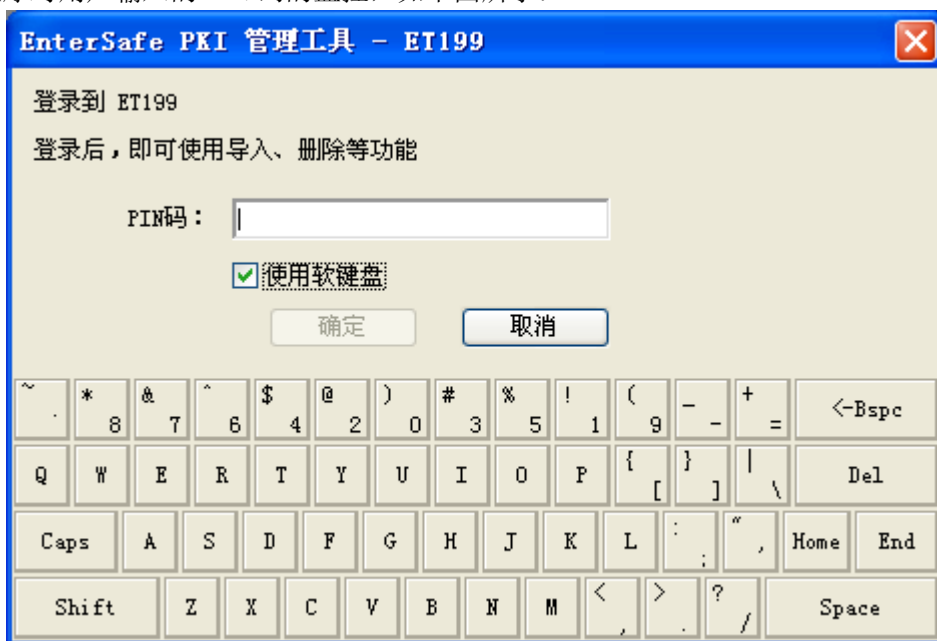


图 10 PIN 码输入框—使用软键盘

注意：选择“使用软键盘”后，物理键盘的键盘输入将被禁用。另外，只有 Windows2000 以上的操作系统支持软键盘功能，Windows Me 和 Windows98 没有此功能。

用户输入正确 PIN 码后，点击“确定”按钮，进入如下图所示的 Token 界面，在界面的上半部显示令牌列表，用户可以点选树型列表内的项目，界面的下部会显示用户点选项目的相应属性值。用户可以通过点击“隐藏属性显示”按钮来隐藏属性显示。用户登录后，不仅可以查看 Token 中的公有数据的信息，还可以查看到 Token 里私有数据的信息。用户登录后，“登录”按钮显示为“登出”，您可以点击“登出”按钮，安全登出 ET199。

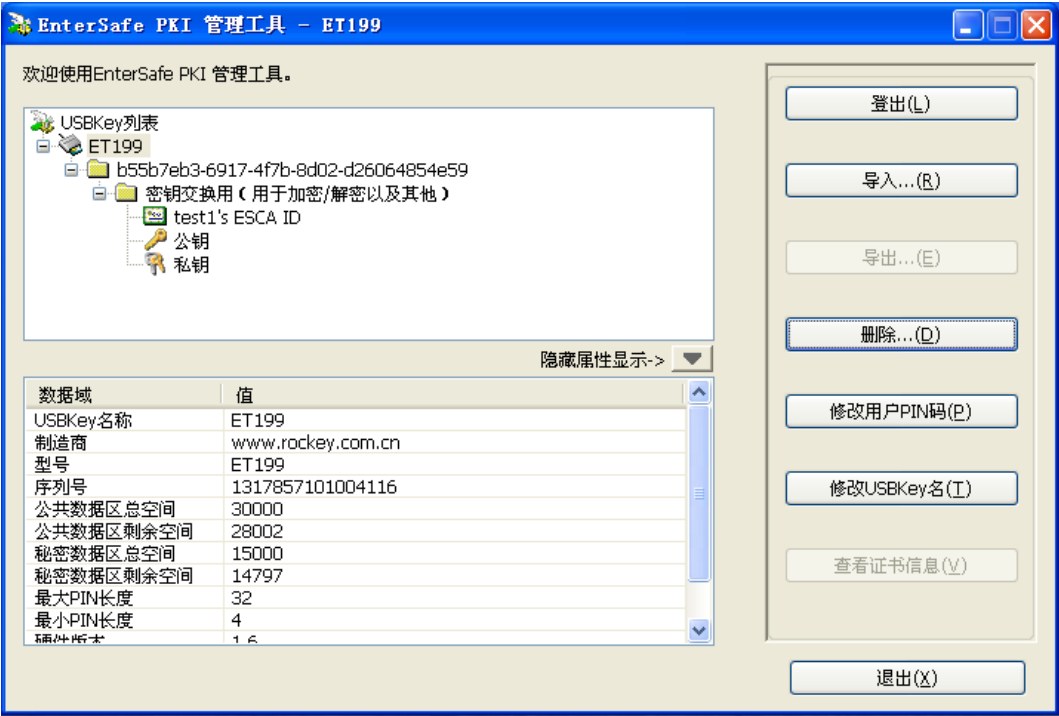


图 11 登录后的界面

如果用户输入的 PIN 码错误，则会弹出下图所示的提示框，提示您的 PIN 码输入错误，点击“是”按钮返回登录对话框，继续登录，点击“否”按钮，退出登录。

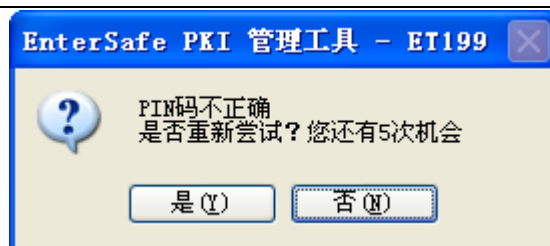


图 12 PIN 码输入错误

注意：ET199 对用户 PIN 码的误输入次数有限制，如果您连续累计 6 次错误输入 PIN 码，USBKey 将被锁定，锁定后您将不能对 USBKey 做任何操作。需要使用管理员工具进行解锁。

● 修改用户 PIN 码

您还可以对 USBKey 的 PIN 码进行修改，插入 USBKey 后点击管理工具主页面上的“修改用户 PIN 码”按钮，弹出如下图所示修改 PIN 码对话框，分别输入原 PIN 码和新 PIN 码并确认新 PIN 码后点击“确定”按钮，即可完成用户 PIN 码的修改。

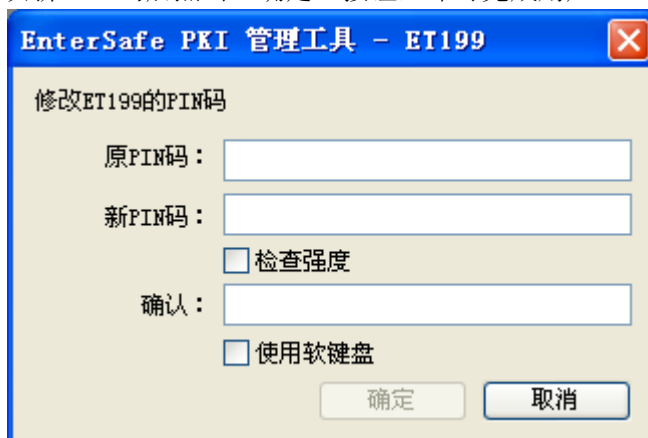


图 13 修改用户 PIN 码

用户可以通过选择使用软键盘来避免木马程序对用户设置的 PIN 码的监控，如果用户选择使用“使用软键盘”，修改用户 PIN 码对话框如下图所示：

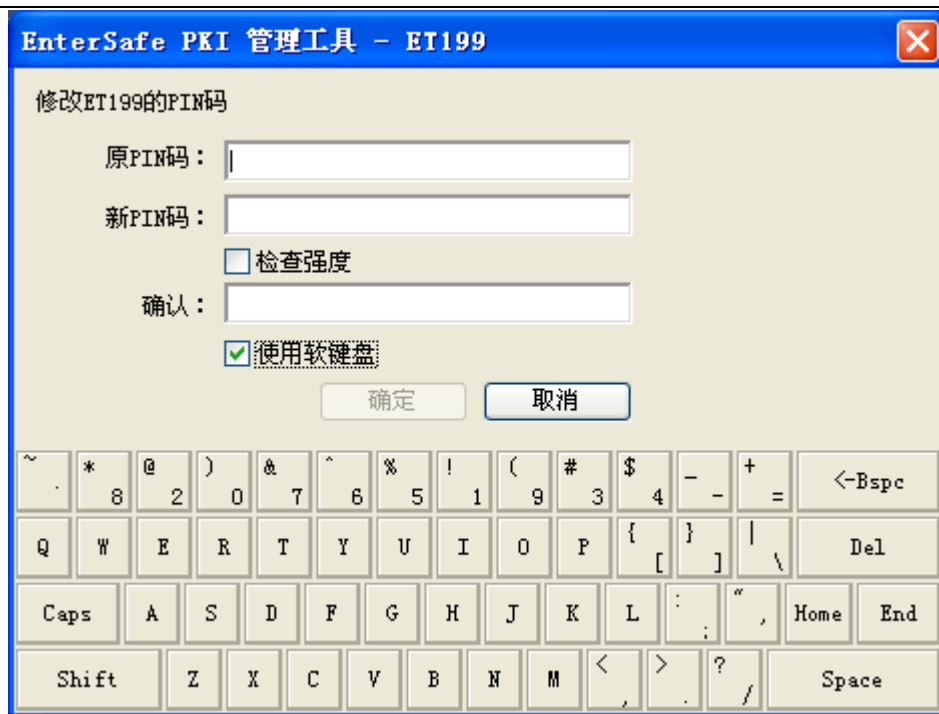


图 14 修改用户 PIN 码—使用软键盘

用户还可以选择“检查强度”来检查设置的 PIN 码的安全强度，当设置的 PIN 码强度较低时管理工具会进行提示，如下图所示的红色圆点，其内有“低”字表示设置的 PIN 码强度较低。

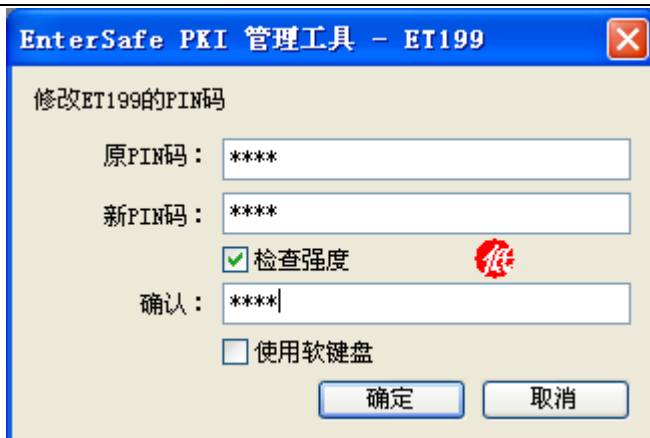


图 15 用户 PIN 码强度低

当设置的 PIN 码强度较强时，会显示如下图所示的提示：

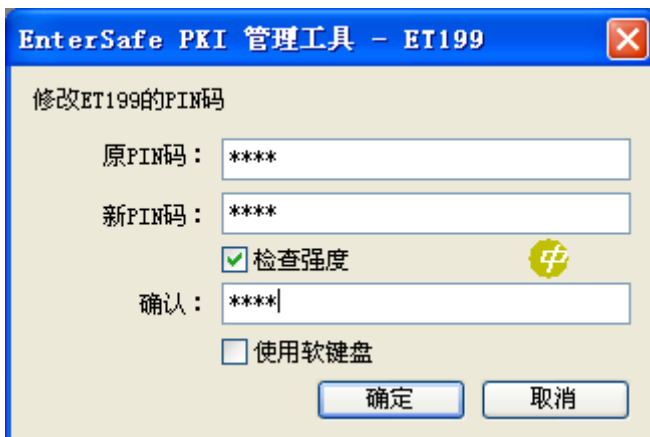


图 16 用户 PIN 码强度中

建议您在设置时，PIN 码中尽量同时包含大小写字母、数字和特殊符号，并且在容易记忆的前提下尽量设置较长的 PIN 码，如下图所示：

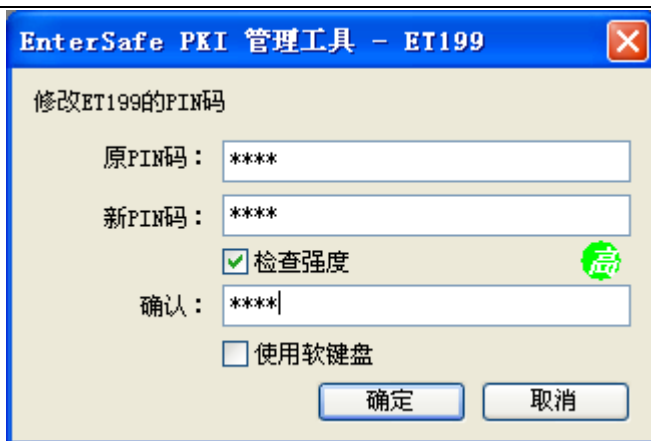


图 17 用户 PIN 码强度高

在“原 PIN 码”栏内和“新 PIN 码”以及“确认”栏内输入相应的 PIN 码，点击“确定”按钮，弹出如下图所示的 PIN 码修改成功对话框，点击“确定”按钮，完成用户 PIN 的修改。

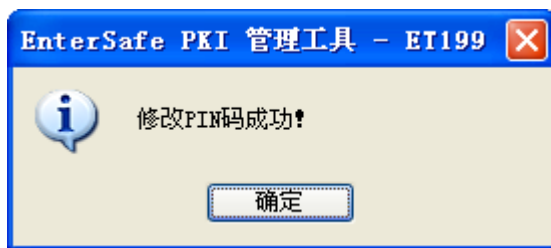


图 18 修改用户 PIN 码完成

● 查看证书信息

在 USBKey 列表中点击容器（文件夹图标）左侧的“+”或双击图标以显示容器内的内容；点击证书图标左侧的“+”以显示证书包含的公钥私钥对。选中证书，此时“查看证书信息”按钮变为可用，如下图所示：

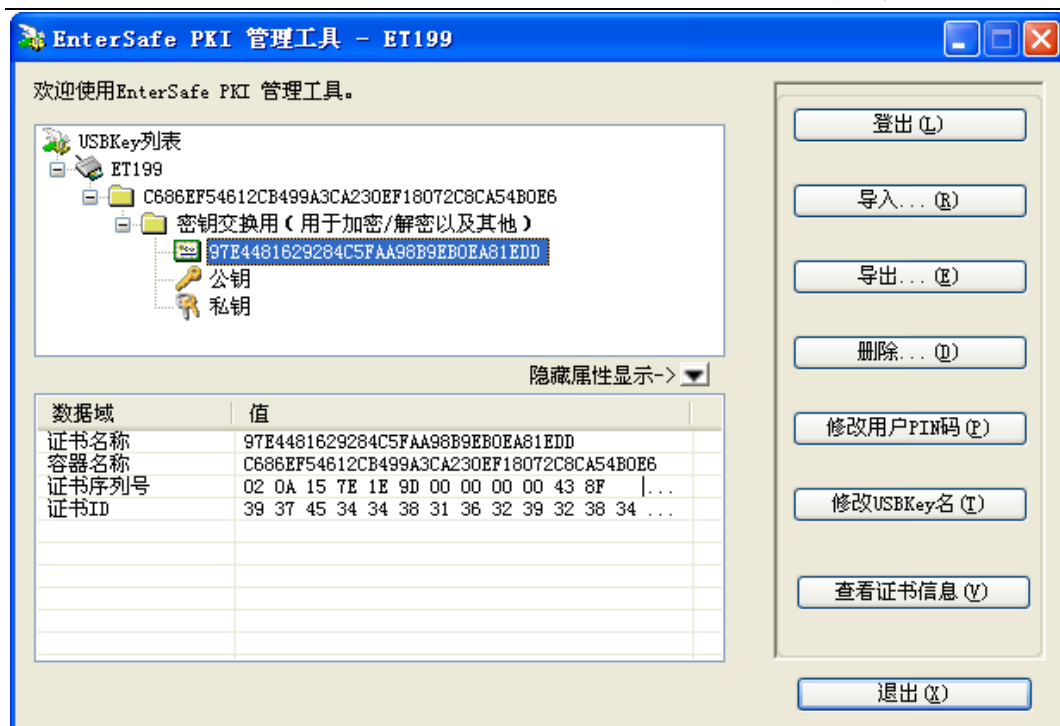


图 19 查看证书信息界面

点击“查看证书信息”按钮或双击证书图标，弹出“查看证书信息”对话框，用户可以点选“常规”、“详细信息”和“路径”选项卡来查看证书的信息，如下图所示：

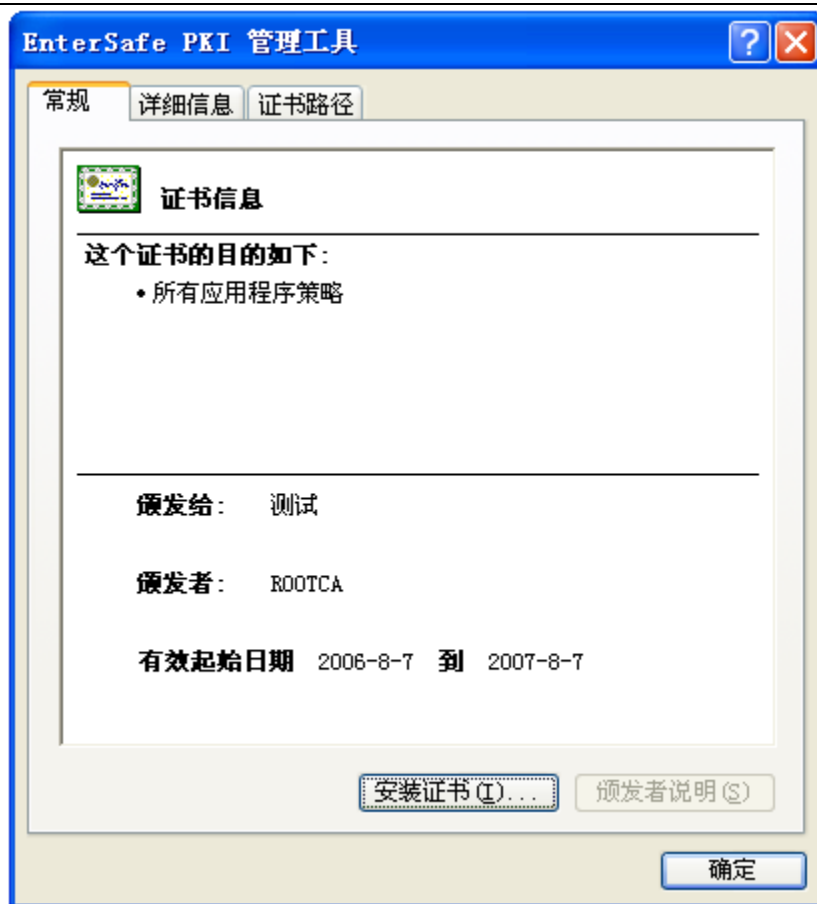


图 20 查看证书对话框

● 导入证书

目前 ET199 支持的证书类型包括：P12、PFX、P7B、CRT 和 CER 五种类型，其中 P12 和 PFX 类型的证书含有公私钥对，P7B、CRT 和 CER 类型的证书不包含公私钥对。下面以导入 PFX、CER 和 P7B 类型的证书为例进行说明。

(1) 导入 PFX 证书

在管理工具主页面点击“导入”按钮，显示如下图所示的界面，点击“浏览”按钮，选择要导入的 PFX 证书的路径，如证书设置了访问密码，还需在“证书访问密码”文本框中输入密码。由于 PFX 证书中通常除了含有公私钥对对应的用户证书外，还可能包含一个完整的证书链，用户可以选择“只导入文件中的用户证书”，这样只导入 PFX

证书中的公私钥对对应的用户证书，或选择“导入文件中的全部证书”，这样可以导入带有证书链的全部证书。用户可以新建一个容器来存储导入的证书，也可以使用已有容器，由于 PFX 类型的证书含有公私钥对，所以既可以用来交换也可以用于签名，用户选择一个证书的用途，完成上述设置后点击“确定”按钮即可完成证书的导入。

注意：同一个容器中只能保存一份签名证书和一份交换证书，如果将一个证书导入已经存在的容器内，并且容器内原有证书与新导入的证书用途相同，则管理工具会提示用户替换原有证书。

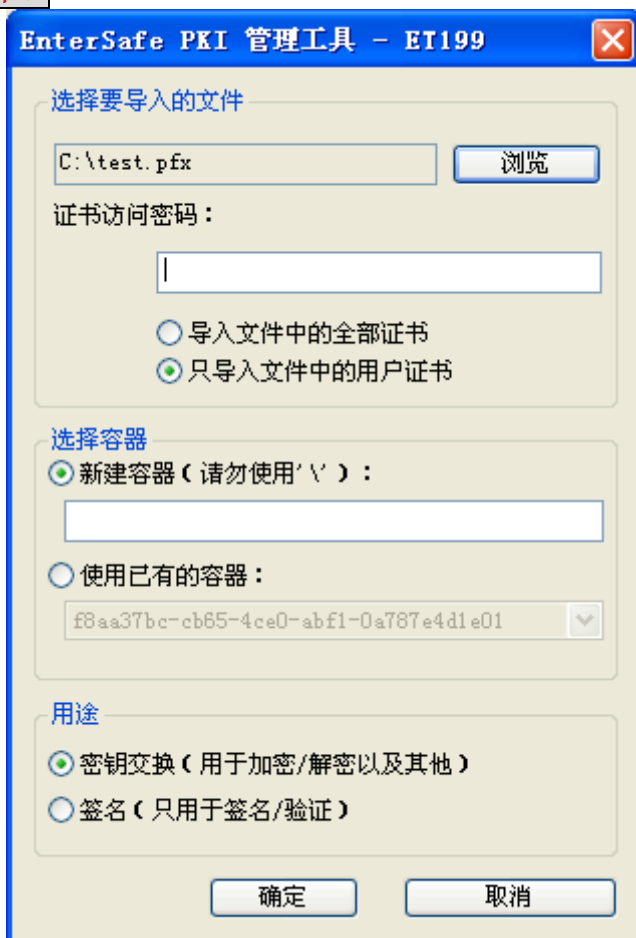


图 21 PFX 证书导入界面

(2) 导入 CER 证书

在管理工具主页面点击“导入”按钮，显示如下图所示的界面，点击“浏览”按钮，

选择要导入 CER 证书的路径，用户需新建一个容器来存储导入的证书，由于 CER 类型的证书不包含公私钥对，只能用于交换，所以“证书用途”部分的单选按钮为不可选状态，完成上述设置后点击“确定”按钮即可完成证书的导入。

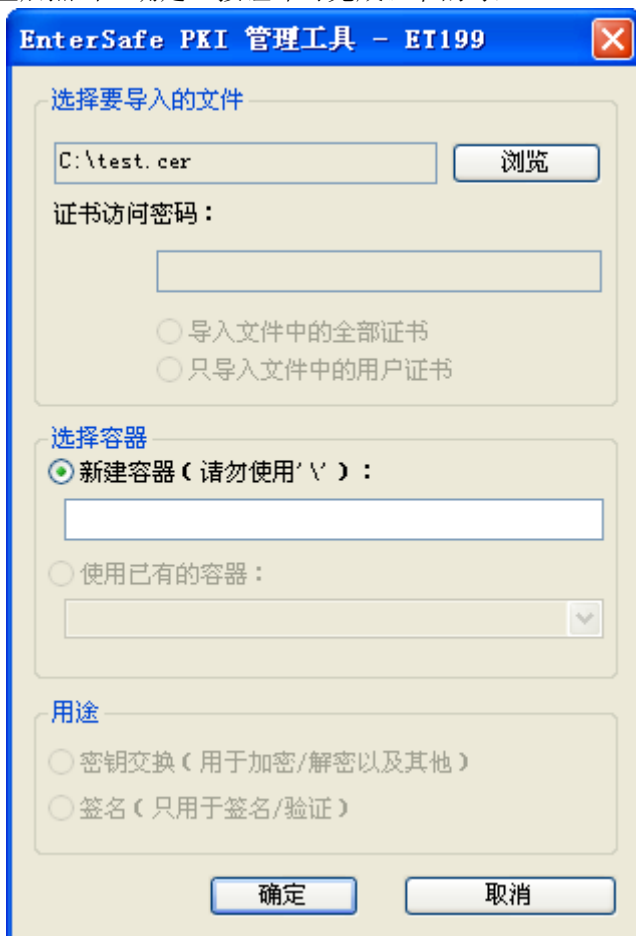


图 22 CER 证书导入界面

(3) 导入 P7B 证书

在管理工具主页面点击“导入”按钮，显示如下图所示的界面，点击“浏览”按钮，选择要导入 P7B 证书的路径，用户需新建一个容器来存储导入的证书，由于 P7B 类型的证书不包含公私钥对，只能用于交换，所以“证书用途”部分的单选按钮为不可选状态，完成上述设置后点击“确定”按钮即可完成证书的导入。

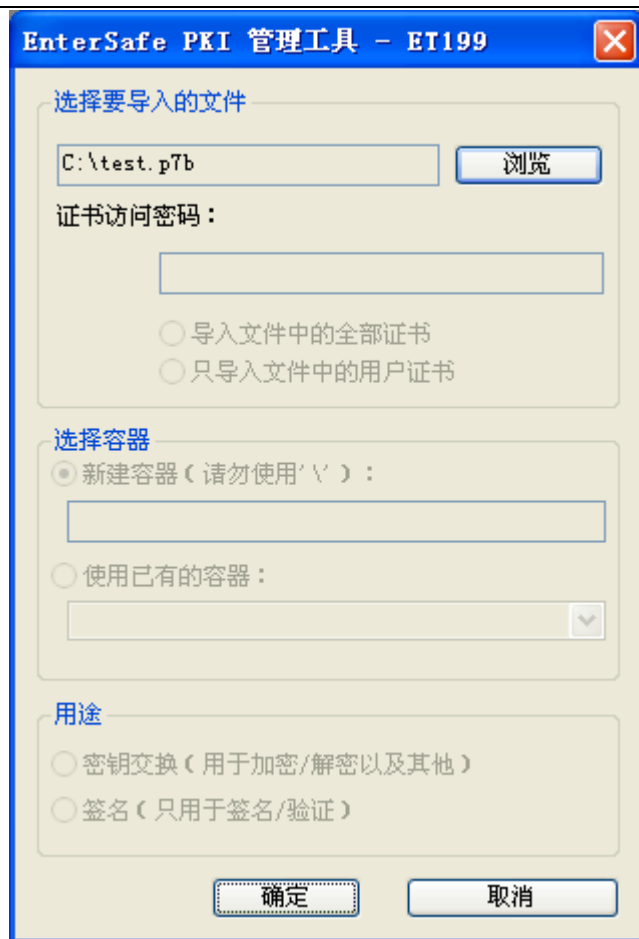


图 23 P7B 证书导入界面

● 导出证书

用户可以使用管理工具将 USBKey 中的证书导出并保存成文件，导出证书的方法如下：在管理工具主页面的树形列表中选择要导出的证书，并点击“导出”按钮，弹出选择导出路径对话框，选择要保存证书文件的路径并设置证书文件的名称，如下图所示：



图 24 选择证书导出路径对话框

点击“保存”按钮，如果导出成功则弹出如下图所示的对话框：

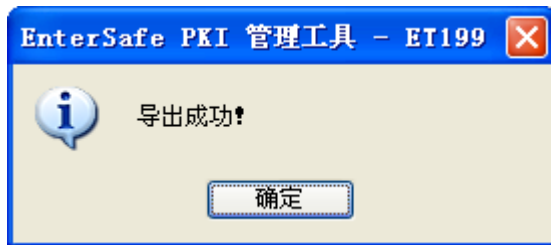


图 25 导出成功对话框

注意：管理工具只能导出证书，证书对应的公私钥对无法导出。

● 删除

在管理工具主页面的树形列表中选择要删除的容器名，并点击“删除”按钮，弹出如下图所示的对话框：

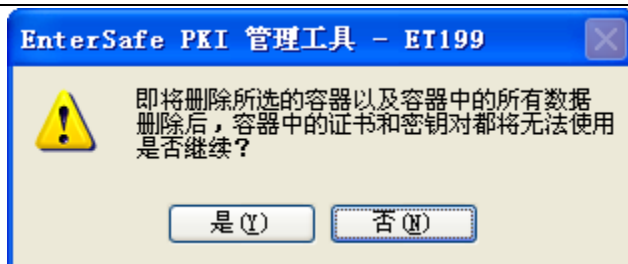


图 26 删除证书提示

点击“是”按钮，确认删除。用户可以用同样的方法删除 ET199 内的密钥或证书，您只需将鼠标点击在树形列表中您要删除的密钥或证书上，再点击“删除”按钮就可删除相应的密钥或证书。**如果您点击树形列表中的“ET199”令牌名，再点击“删除”按钮，您将删除 ET199 中所有容器及容器内的证书和密钥。**

● 修改令牌名

一般情况下 USBKey 都是以序列号来相互区分的，但是序列号不直观而且不容易记，所以在 ET199 中以 USBKey 名称来标记 USBKey。USBKey 名可以根据自己的喜好任意命名。

在管理工具主页面点击“修改 USBKey 名”按钮，弹出如下图所示的对话框：

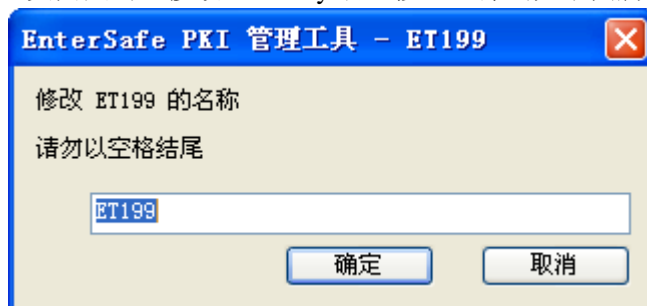


图 27 修改令牌名

在文本框内输入令牌的名称后点击“确定”按钮，完成令牌名的修改。**注意：令牌名称最大不能超过 32 个字符，其中一个汉字占 3 个字符。**

4.2 ET199 管理员版管理工具

注意：在使用本工具前，需要先安装 PKI 运行时环境（即安装 redist\cn 目录下的 ET199-SimpChinese.exe），安装时需要有操作系统的管理员权限。

管理员版管理工具包括用户版管理工具全部的功能，相应功能的使用方法与用户版管理工具的使用方法相同。除此之外，还包括“解锁”，“初始化”和“修改 SO PIN”的功能。管理员版管理工具的界面与用户版管理工具的界面稍有不同，在管理工具主界面的右侧按钮区有一个翻页按钮，点击这个翻页按钮能够使按钮进行前后换页，在下图中用红色圈出：

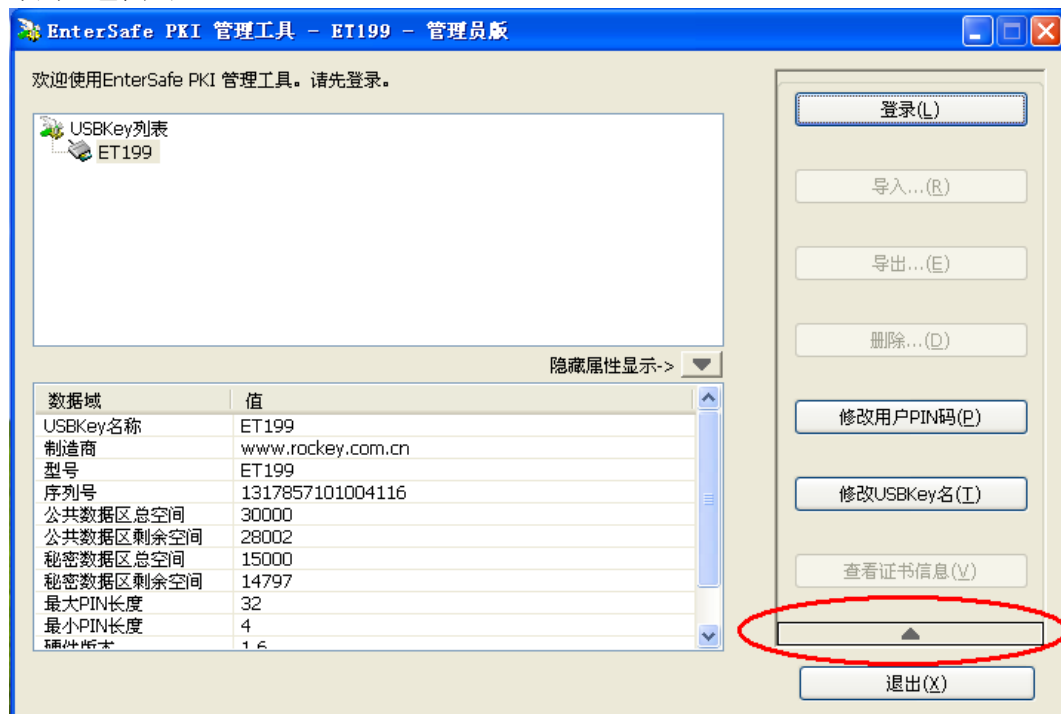


图 28 管理员版管理工具主页面 I

点击翻页按钮，显示如下图所示的界面：

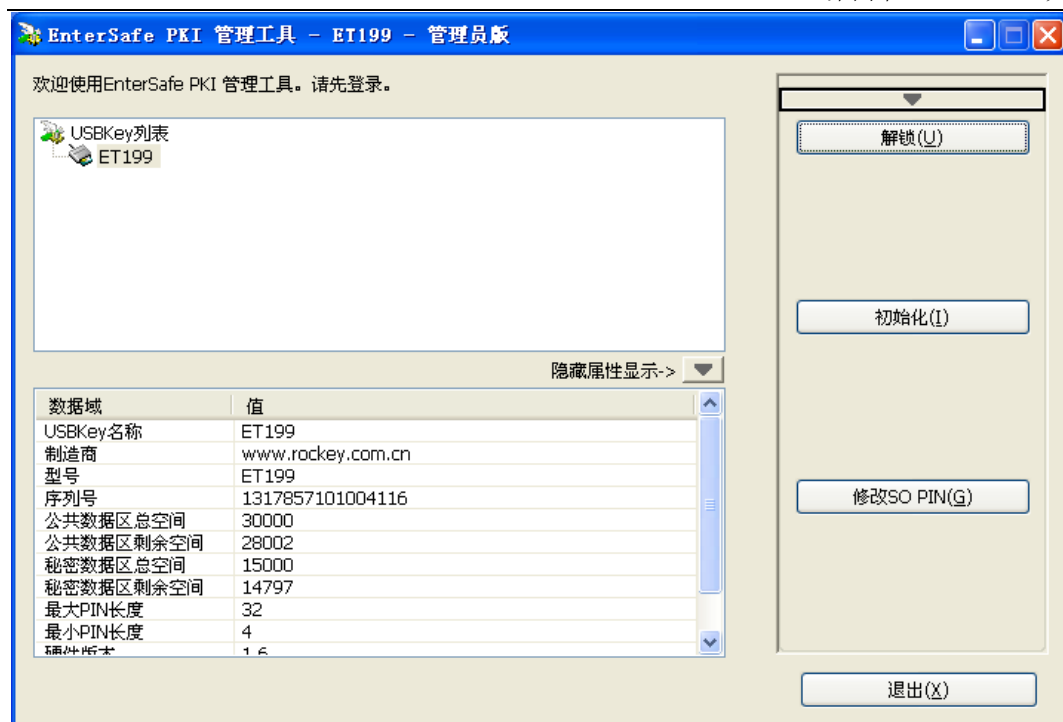


图 29 管理员版管理工具主页面 II

● 解锁

利用管理员版管理工具可以对由于误操作而锁定的 USBKey 进行解锁，解锁的方法如下：在图 29 所示的管理工具主界面上点击“解锁”按钮，将弹出解锁对话框，如下图所示：

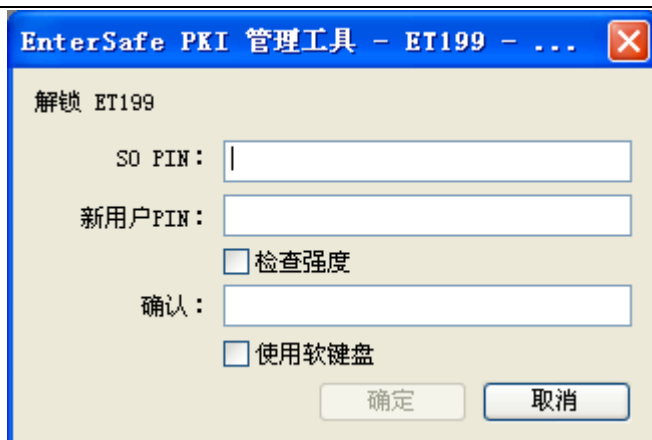


图 30 解锁对话框

用户可以通过选择使用“使用软键盘”来避免木马程序对用户输入和设置的 PIN 码的监控，如果用户选择使用“使用软键盘”，解锁 USBKey 的对话框如下图所示：

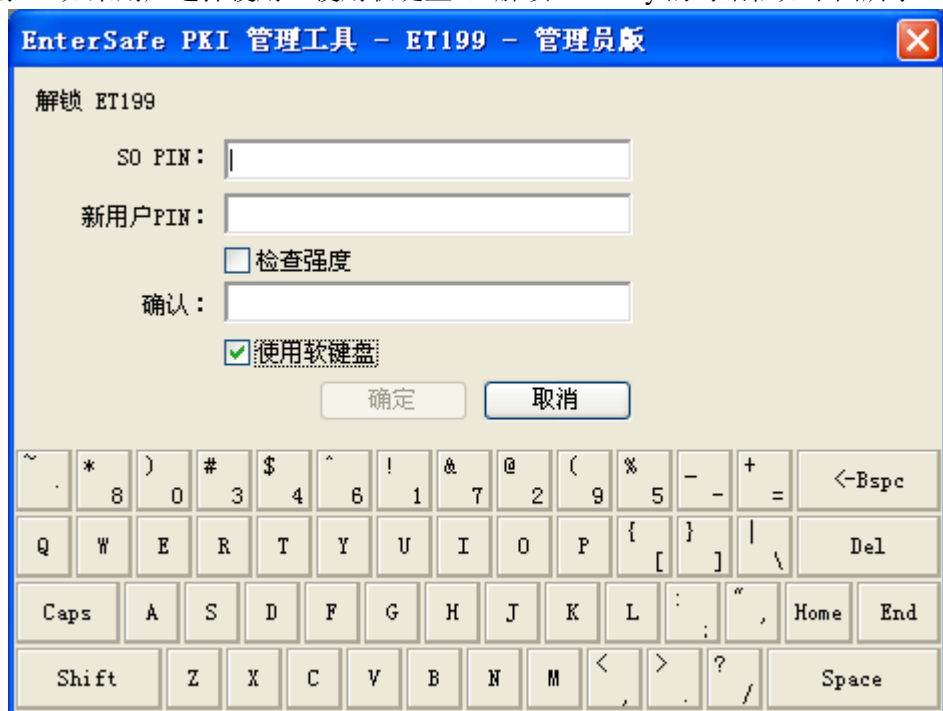


图 31 解锁对话框—软键盘

用户还可以选择“检查强度”来检查设置的 PIN 码的安全强度，具体使用方法与“修改用户 PIN 码”中的“检查强度”相同。

输入 SO PIN，设置新用户 PIN 并确认，然后点击“确定”按钮，解锁成功将弹出如下图所示的对话框：

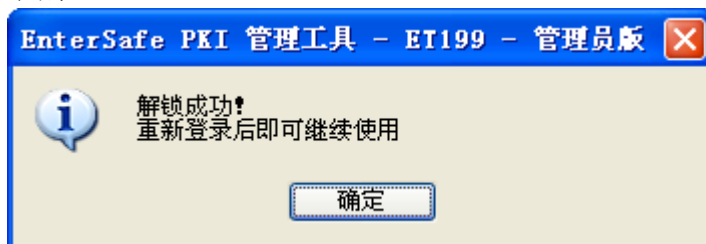
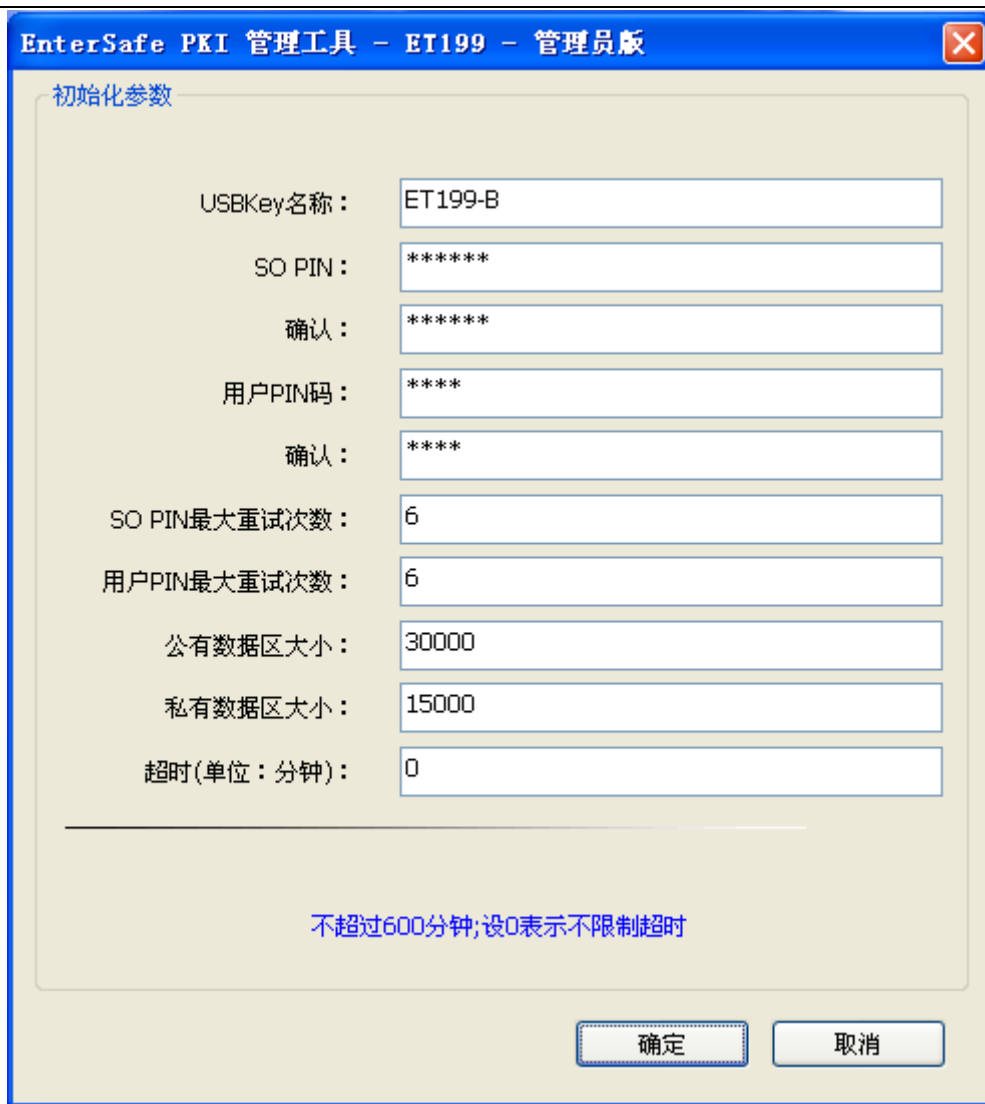


图 32 解锁成功对话框

● 初始化

在图 29 所示的管理工具主界面上点击“初始化”按钮，将弹出初始化 USBKey 对话框，如下图所示：



EnterSafe PKI 管理工具 - ET199 - 管理员版

初始化参数

USBKey名称：	ET199-B
SO PIN：	*****
确认：	*****
用户PIN码：	****
确认：	****
SO PIN最大重试次数：	6
用户PIN最大重试次数：	6
公有数据区大小：	30000
私有数据区大小：	15000
超时(单位：分钟)：	0

不超过600分钟;设0表示不限制超时

确定 取消

图 33 初始化对话框

超时设置表明在前一次验证后，然后没有对 ET199 进行操作，当没有操作的时间超过设定的时间后，ET199 的状态恢复为没有验证的状态。**注意：当“超时”设置为“0”时，表明不对操作进行超时限制。**

设置好各初始化参数后，点击“确定”按钮，此时会弹出提示对话框，让用户确认进行初始化操作，如下图所示：

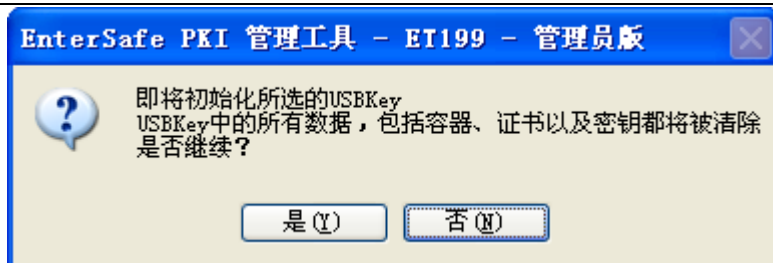


图 34 确认初始化对话框

点击“是”按钮，开始对 USBKey 的初始化操作，初始化成功后会弹出初始化成功对话框，如下图所示：

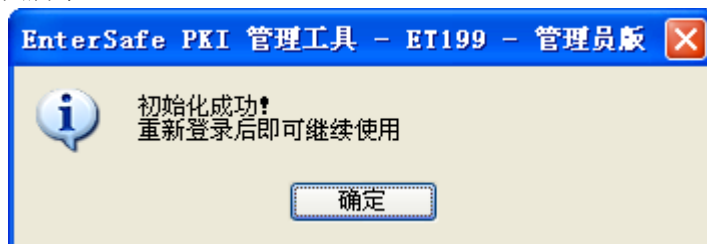


图 35 初始化成功对话框

● 修改 SO PIN

在图 29 所示的管理工具主界面上点击“修改 SO PIN”按钮，将弹出修改 SO PIN 对话框。如下图所示：

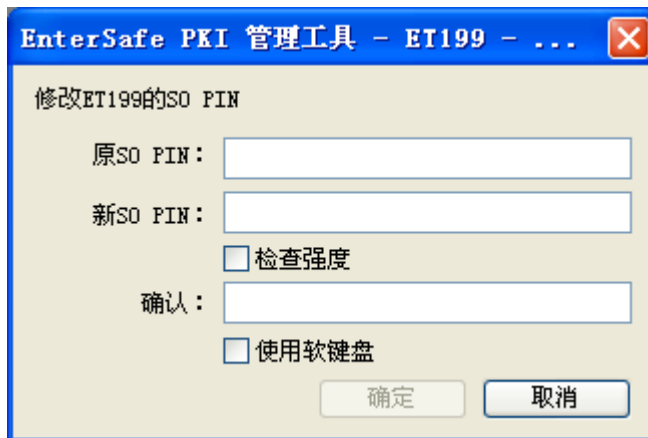


图 36 修改 SO PIN 对话框

用户可以通过选择使用“使用软键盘”来避免木马程序对用户设置的 SO PIN 的监

控，如果用户选择使用“使用软键盘”，修改 SO PIN 对话框如下图所示：

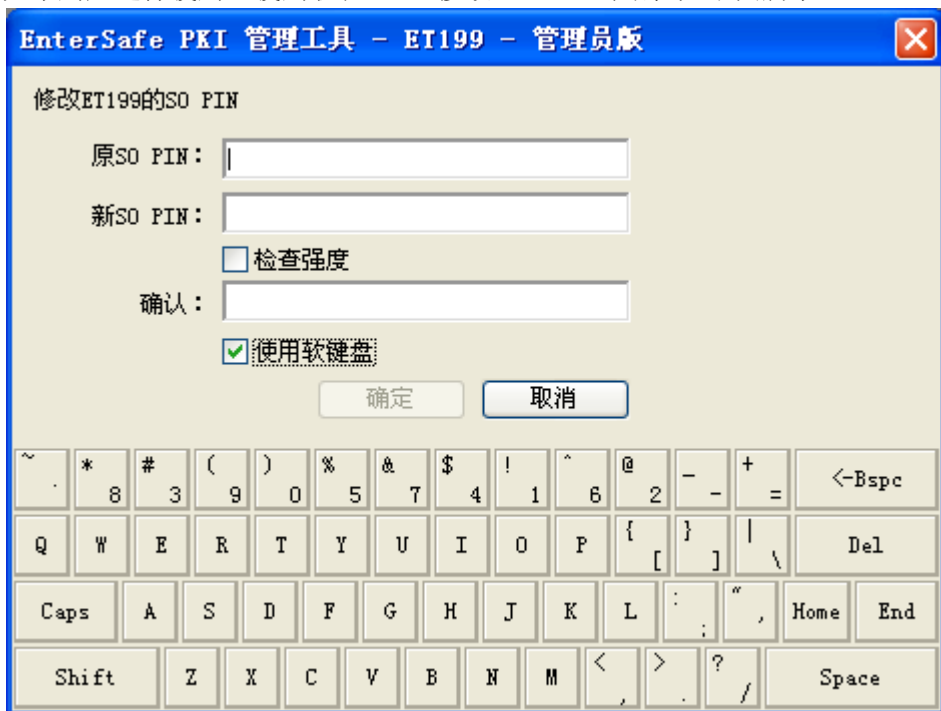


图 37 修改 SO PIN—使用软键盘

用户还可以选择“检查强度”来检查您设置的 SO PIN 的安全强度，具体使用方法与“修改用户 PIN 码”中的“检查强度”相同。

填入原 SO PIN、新 SO PIN，并确认新 SO PIN，然后点击“确定”按钮，修改 SO PIN 成功将弹出修改 SO PIN 成功对话框，如下图所示：

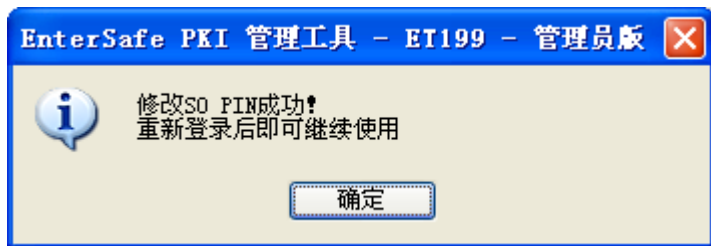


图 38 修改 SO PIN 成功

4.3 ET199 配置工具

注意：在使用本工具前，需要先安装 PKI 运行时环境（即安装 redist\cn 目录下的 ET199-SimpChinese.exe），安装时需要有操作系统的管理员权限。

可以在“开始”→“所有程序”→“EnterSafe”→“ET199”中找到配置工具的快捷方式，点击配置工具的快捷方式启动配置工具，出现界面如下图所示：

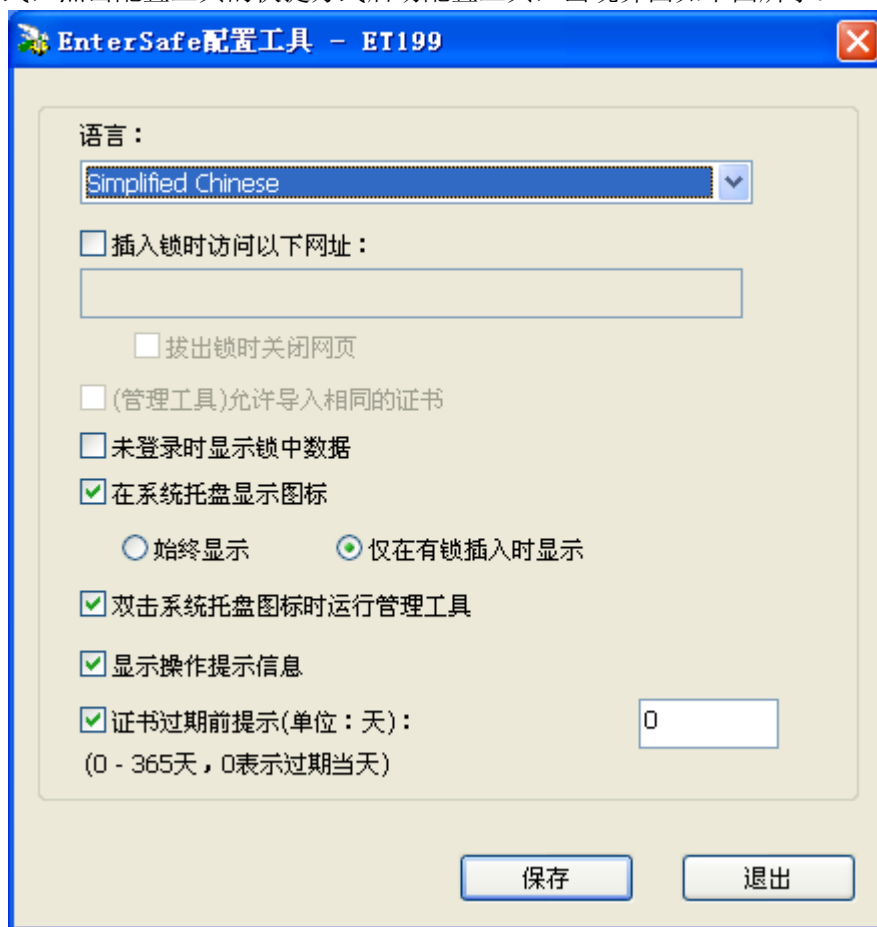


图 39 配置工具的界面

注意：在 Windows Vista/Windows 2008/Windows 7 下的配置工具对话框没有“拔出锁时关闭网页”的选项。

在上图中，您可以看到，通过配置工具可以进行语言、插入锁访问指定网址、托盘图标等多项设置，当您选中某项功能后点击“保存”按钮，会提示您配置成功信息，如下图所示：

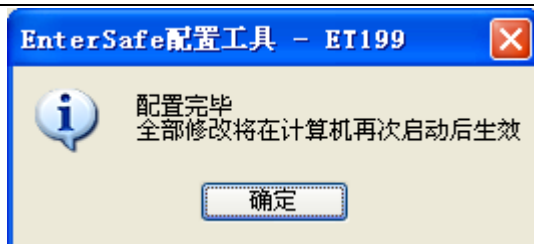


图 40 配置成功对话框

在您使用配置工具进行功能配置后，建议您重新启动计算机，以便使配置生效。

4.4 ET199 初始化设置工具

注意：在使用本工具前，需要先安装 PKI 运行时环境（即安装 `redist\cn` 目录下的 `ET199-SimpChinese.exe`），安装时需要有操作系统的管理员权限。

4.4.1 初始化为空锁

在使用 ET199 加密锁功能前，如果 ET199 为 PKI 格式，需要先将 ET199 初始化为空锁。运行 `DevFormat.exe` 工具，鼠标左键单击左边的“初始化为空锁”按钮，出现如下界面：

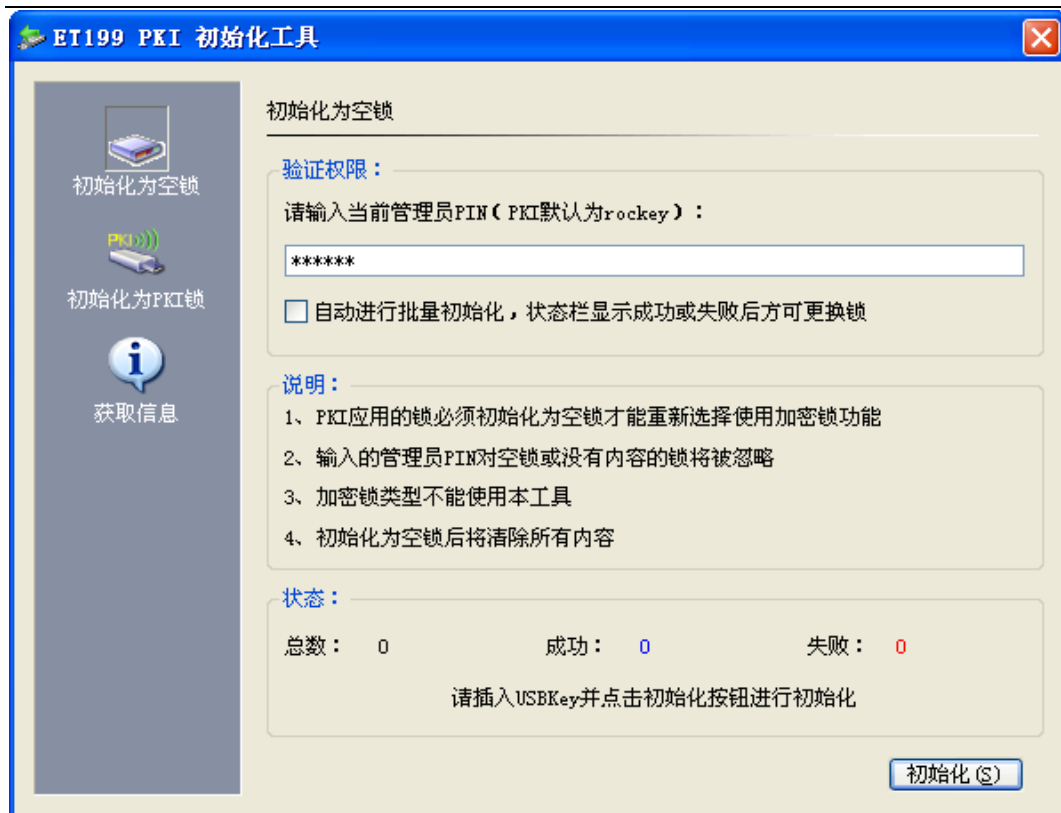


图 41 设置为空锁

在右侧输入管理员 PIN（见 2.2 节），鼠标左键单击“初始化”按钮进行初始化。选择“自动进行批量初始化”复选框后，会进行批量初始化。**注意：如果管理员 PIN 锁死了，则没有办法初始化为空锁，只能退回给我公司处理，重新生产。**

初始空锁成功后，使用加密锁设置工具（见《ET199 超级多功能锁用户手册—加密锁篇》中 5.2.2 节）初始化为加密锁。

4.4.2 初始化为 PKI 格式

如果要使用 ET199 的 PKI 功能，需要先使用本工具将 ET199 格式化为 PKI 格式。在格式化前，要保证 ET199 为空锁状态。如果 ET199 是加密锁格式，请先使用加密锁设置工具（见《ET199 超级多功能锁用户手册—加密锁篇》中 5.2.2 节）初始化为空锁。

鼠标左键单击左边的“初始化为 PKI 锁”按钮，出现如下界面：

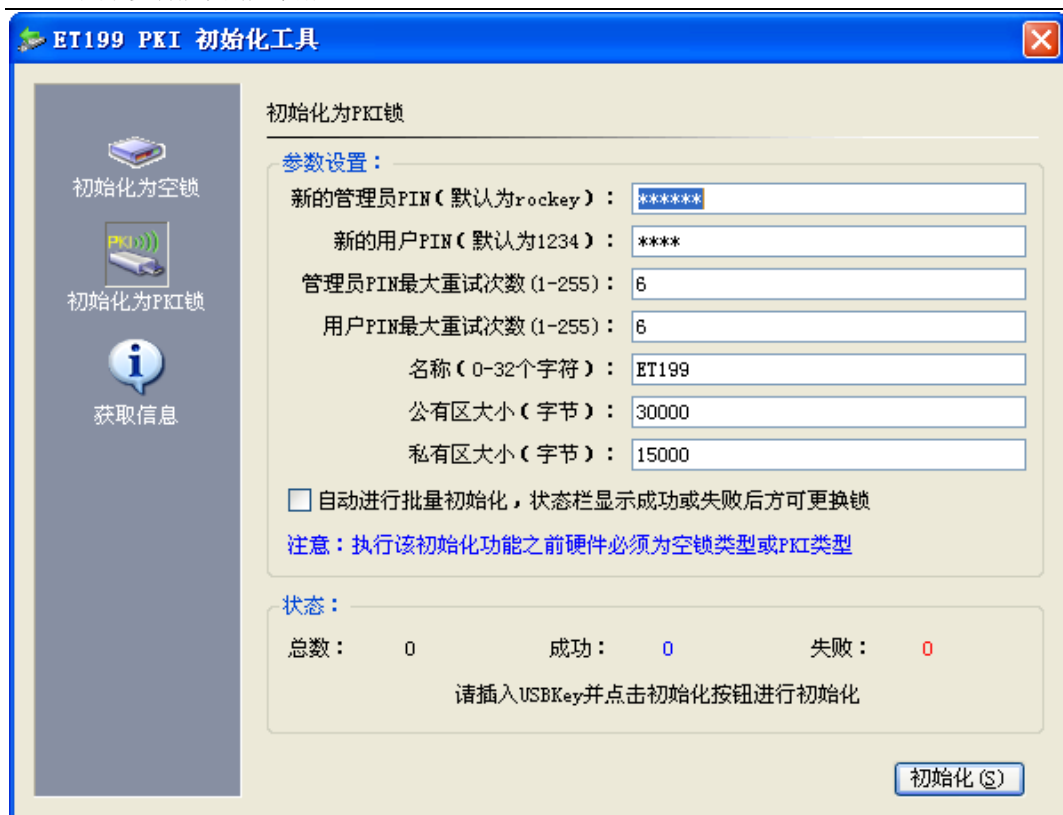


图 42 设置为 PKI 锁

填入相关信息，需要注意：

- 管理员 PIN（SO PIN）和用户 PIN（User PIN）长度：4—32
- 管理员 PIN（SO PIN）和用户 PIN（User PIN）重试次数：1—255 次
为 1—254 时：重试次数
为 255 时：无重试次数限制
- 名称为 ET199 设备的 Label 名称，可以通过 PKCS#11 接口获取（参见 5.1.2 节）
- 公有区空间大小和私有区空间大小可以调整，但公私有区的总空间不能超过 45K。每个空间最小为 1 字节。

鼠标左键单击“初始化”按钮进行初始化。选择“自动进行批量初始化”复选框后，会进行批量初始化。

第五章 常用功能需求举例

本章将要介绍使用 PKCS#11 和 CAPI 进行开发时需要注意的事项以及经常使用到的功能。如：读写数据，监测硬件插拔，导入证书等。关于 PKCS#11 和 CAPI 的说明您可以参见 RSA 公司的 PKCS#11 开发文档和微软的 MSDN，也可以参见由 EnterSafe 提供的说明文档。

5.1 PKCS#11 接口

在使用 PKCS#11 进行开发时，需要把相关的头文件（include\pkcs11 目录下）拷贝到工程目录下，如果您希望在工程中使用 lib，也需要把 PKCS#11 的 lib 库（lib\et199csp11.lib）拷贝到工程目录下，并加入到工程中。在阅读本节内容时，建议结合具体的示例程序来熟悉 PKCS#11 接口。示例程序可以从我公司的网站上下载。

5.1.1 打开和关闭硬件

在使用 PKCS#11 接口进行编程时需要注意，**一个进程中只能调用一次 C_Initialize 函数和 C_Finalize 函数，不能频繁的调用。**打开和关闭设备的过程，见下面的代码：

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h" //PKCS#11 的头文件，由 RSA 公司定义

//定义获取 Cryptoki 库中函数指针表的函数指针
typedef CK_RV (* C_GETFUNCTIONLISTPROC) (CK_FUNCTION_LIST_PTR_PTR);

int main(int argc, char* argv[])
{
    C_GETFUNCTIONLISTPROC pGetFunctionList = NULL;
    CK_FUNCTION_LIST_PTR ET199 = NULL;
    HINSTANCE hDll = NULL;

    CK_RV ck_rv = 0;
    CK_ULONG ulCount = 0;
    CK_SLOT_ID_PTR pSlot = NULL;
    CK_SESSION_HANDLE hSession = NULL;
```

```
//加载 PKCS#11 库
hDll = LoadLibrary("et199csp11.dll");
if(hDll == NULL)
{
    printf("Loadlibrary Error!\n");
    return -1;
}

//获得一个指向 Cryptoki 库中函数指针表的指针
pGetFunctionList =
    (C_GETFUNCTIONLISTPROC)GetProcAddress(hDll, "C_GetFunctionList");
if(pGetFunctionList == NULL)
{
    printf("Get Function List Error!\n");
    return -1;
}

pGetFunctionList(&ET199);
if(ET199 == NULL)
{
    printf("Get Function List Pointer Error!\n");
    return -1;
}

//C_Initialize, 初始化环境
ck_rv = ET199->C_Initialize(NULL_PTR);
if(ck_rv != CKR_OK)
{
    printf("C_Initialize Error! 0x%08x\n", ck_rv);
    return -1;
}

//获得计算机系统中的 Token 槽列表
ck_rv = ET199->C_GetSlotList(TRUE, NULL, &ulCount);
if(ck_rv != CKR_OK)
{
    printf("C_GetSlotList 1 Error! 0x%08x\n", ck_rv);
}
```



```

        return -1;
    }

    if(ulCount == 0)
    {
        printf("No ET199 Found!\n", ck_rv);
        return -1;
    }

    pSlot = new CK_SLOT_ID[ulCount];
    ck_rv = ET199->C_GetSlotList(TRUE, pSlot, &ulCount);
    if(ck_rv != CKR_OK)
    {
        printf("C_GetSlotList 2 Error! 0x%08x\n", ck_rv);
        goto clean;
    }

    //打开 Token 槽列表中 pSlot[0]的 ET199, 即为第一个 ET199
    //打开后的会话句柄保留在 hSession 中
    //以后就可以使用这个句柄对 ET199 进行操作了
    ck_rv = ET199->C_OpenSession(pSlot[0], CKF_RW_SESSION|CKF_SERIAL_SESSION,
    NULL, NULL, &hSession);
    if(ck_rv != CKR_OK)
    {
        printf("C_OpenSession Error! 0x%08x\n", ck_rv);
        goto clean;
    }
    else
    {
        printf("Open ET199 OK!\n");
    }

    //其他操作.....

    //关闭会话句柄 hSession
    ck_rv = ET199->C_CloseSession(hSession);

clean:

```

```
//关闭环境
ck_rv = ET199->C_Finalize(NULL_PTR);
if(ck_rv != CKR_OK)
{
    printf("C_Finalize Error! 0x%08x\n", ck_rv);
}

//释放 DLL
if(hDll)
{
    FreeLibrary(hDll);
}

//释放内存
if(pSlot != NULL)
{
    delete [] pSlot;
    pSlot = NULL;
}

return 0;
}
```

在编程时,只能调用一次 C_Initialize 函数和 C_Finalize 函数,不能频繁的调用。也不建议多次调用 C_GetSlotList 函数,因为每次调用都会遍历一遍系统中的 Token 槽列表,耽误时间。可以多次调用 C_OpenSession 和 C_CloseSession 函数,即当一组操作完后调用 C_CloseSession 函数,当需要其他操作时再次调用 C_OpenSession 函数。

5.1.2 获取硬件信息

使用 C_GetTokenInfo 函数可以获取到硬件的相关信息,包括:硬件序列号,公私有区大小,Token Label 等信息,这些信息都是存储在 CK_TOKEN_INFO 结构中。

```
CK_TOKEN_INFO tokenInfo={0};
ck_rv = ET199->C_GetTokenInfo(pSlot[0], &tokenInfo);
if(ck_rv != CKR_OK)
{
    printf("C_GetTokenInfo Error! 0x%08x\n", ck_rv);
}
```

```

    goto clean;
}

```

在 5.1.1 的示例中“获得计算机系统中的 Token 槽列表”后，“打开 ET199Session 会话”前，就可以调用上面的函数获得硬件的相关信息。如：ET199 硬件序列号在 tokenInfo.serialNumber 中。

5.1.3 验证 SO PIN 和 User PIN

调用 C_Login 函数就可以验证 SO PIN 和 User PIN，第二个参数进行区分，见下面的代码：

```

BYTE pbSOPIN[] = "rockey";
DWORD dwSOPINLen = strlen((const char*)pbSOPIN);
BYTE pbUserPIN[] = "1234";
DWORD dwUserPINLen = strlen((const char*)pbUserPIN);

//验证 SO PIN，第二个参数为 CKU_SO
ck_rv = ET199->C_Login(hSession, CKU_SO,
                      (CK_UTF8CHAR_PTR)pbSOPIN, dwSOPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}

//验证 User PIN，第二个参数为 CKU_USER
ck_rv = ET199->C_Login(hSession, CKU_USER,
                      (CK_UTF8CHAR_PTR)pbUserPIN, dwUserPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}

```

5.1.4 初始化

使用 C_InitToken 函数可以将 ET199 进行初始化，可以将硬件中的内容全部清空，

重新设置 User PIN。在调用函数时需要输入正确的 SO PIN，而且不能打开 Session 会话，初始后还需要调用 C_InitPIN 函数设置 User PIN。见下面的代码：

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"          //PKCS#11 的头文件，由 RSA 公司定义

//定义获取 Cryptoki 库中函数指针表的函数指针
typedef CK_RV (* C_GETFUNCTIONLISTPROC) (CK_FUNCTION_LIST_PTR_PTR);

int main(int argc, char* argv[])
{
    C_GETFUNCTIONLISTPROC pGetFunctionList = NULL;
    CK_FUNCTION_LIST_PTR ET199 = NULL;
    HINSTANCE hDll = NULL;

    CK_RV ck_rv = 0;
    CK_ULONG ulCount = 0;
    CK_SLOT_ID_PTR pSlot = NULL;
    CK_SESSION_HANDLE hSession = NULL;

    //SOPIN 和 UserPIN
    BYTE pbSOPIN[] = "rockey";
    DWORD dwSOPINLen = strlen((const char*)pbSOPIN);
    BYTE pbUserPIN[] = "1234";
    DWORD dwUserPINLen = strlen((const char*)pbUserPIN);
    BYTE pbTokenName[] = "ET199";

    //加载 PKCS#11 库
    hDll = LoadLibrary("et199csp11.dll");
    if(hDll == NULL)
    {
        printf("Loadlibrary Error!\n");
        return -1;
    }

    //获得一个指向 Cryptoki 库中函数指针表的指针
```

```
pGetFunctionList =
    (C_GETFUNCTIONLISTPROC)GetProcAddress(hDll, "C_GetFunctionList")
;
if(pGetFunctionList == NULL)
{
    printf("Get Function List Error!\n");
    return -1;
}

pGetFunctionList(&ET199);
if(ET199 == NULL)
{
    printf("Get Function List Pointer Error!\n");
    return -1;
}

//C_Initialize, 初始化环境
ck_rv = ET199->C_Initialize(NULL_PTR);
if(ck_rv != CKR_OK)
{
    printf("C_Initialize Error! 0x%08x\n", ck_rv);
    return -1;
}

//获得计算机系统上的 Token 槽列表
ck_rv = ET199->C_GetSlotList(TRUE, NULL, &ulCount);
if(ck_rv != CKR_OK)
{
    printf("C_GetSlotList 1 Error! 0x%08x\n", ck_rv);
    return -1;
}

if(ulCount == 0)
{
    printf("No ET199 Found!\n", ck_rv);
    return -1;
}
```

```
pSlot = new CK_SLOT_ID[ulCount];
ck_rv = ET199->C_GetSlotList(TRUE, pSlot, &ulCount);
if(ck_rv != CKR_OK)
{
    printf("C_GetSlotList 2 Error! 0x%08x\n", ck_rv);
    goto clean;
}

//初始化 ET199
ck_rv = ET199->C_InitToken(pSlot[0], (CK_UTF8CHAR_PTR)pbSOPIN, dwSOPINLen,
                          (CK_UTF8CHAR_PTR)pbTokenName);
if(ck_rv != CKR_OK)
{
    printf("C_InitToken Error! 0x%08x\n", ck_rv);
    goto clean;
}

//打开 Session 会话，得到会话句柄 hSession
ck_rv = ET199->C_OpenSession(pSlot[0], CKF_RW_SESSION|CKF_SERIAL_SESSION,
NULL, NULL, &hSession);
if(ck_rv != CKR_OK)
{
    printf("C_OpenSession Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Open ET199 OK!\n");
}

//以管理员身份登陆，即验证 SO PIN
ck_rv = ET199->C_Login(hSession, CKU_SO,
                      (CK_UTF8CHAR_PTR)pbSOPIN, dwSOPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}
```

```
//设置新的用户 PIN, 即设置 User PIN
ck_rv = ET199->C_InitPIN(hSession,
                        (CK_UTF8CHAR_PTR)pbUserPIN, dwUserPINLen);
if(ck_rv != CKR_OK)
{
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}

//登出
ck_rv = ET199->C_Logout(hSession);
if(ck_rv != CKR_OK)
{
    printf("C_Logout Error! 0x%08x\n", ck_rv);
    goto clean;
}

//关闭会话句柄 hSession
ck_rv = ET199->C_CloseSession(hSession);
printf("Init ET199 OK!\n");

clean:
//关闭环境
ck_rv = ET199->C_Finalize(NULL_PTR);
if(ck_rv != CKR_OK)
{
    printf("C_Finalize Error! 0x%08x\n", ck_rv);
}

//释放 DLL
if(hDll)
{
    FreeLibrary(hDll);
}

//释放内存
if(pSlot != NULL)
```

```
{
    delete [] pSlot;
    pSlot = NULL;
}

return 0;
}
```

5.1.5 检测硬件插拔

使用 C_WaitForSlotEvent 函数来检测 ET199 硬件的插拔。在检测插拔时需要使用 C_GetSlotInfo 函数来得到 Token 列表槽的信息，见下面的代码

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"          //PKCS#11 的头文件，由 RSA 公司定义

//定义获取 Cryptoki 库中函数指针表的函数指针
typedef CK_RV (* C_GETFUNCTIONLISTPROC) (CK_FUNCTION_LIST_PTR_PTR);

int main(int argc, char* argv[])
{
    C_GETFUNCTIONLISTPROC pGetFunctionList = NULL;
    CK_FUNCTION_LIST_PTR ET199 = NULL;
    HINSTANCE hDll = NULL;

    CK_RV ck_rv = 0;
    CK_ULONG ulCount = 0;
    CK_SLOT_ID_PTR pSlot = NULL;
    CK_SLOT_INFO si = {0};

    //加载 PKCS#11 库
    hDll = LoadLibrary("et199csp11.dll");
    if(hDll == NULL)
    {
        printf("Loadlibrary Error!\n");
    }
}
```



```
        return -1;
    }

    //获得一个指向 Cryptoki 库中函数指针表的指针
    pGetFunctionList =
        (C_GETFUNCTIONLISTPROC)GetProcAddress(hDll, "C_GetFunctionList");
    if(pGetFunctionList == NULL)
    {
        printf("Get Function List Error!\n");
        return -1;
    }

    pGetFunctionList(&ET199);
    if(ET199 == NULL)
    {
        printf("Get Function List Pointer Error!\n");
        return -1;
    }

    //C_Initialize, 初始化环境
    ck_rv = ET199->C_Initialize(NULL_PTR);
    if(ck_rv != CKR_OK)
    {
        printf("C_Initialize Error! 0x%08x\n", ck_rv);
        return -1;
    }

    //获得计算机系统上的 Token 槽列表
    ck_rv = ET199->C_GetSlotList(TRUE, NULL, &ulCount);
    if(ck_rv != CKR_OK)
    {
        printf("C_GetSlotList 1 Error! 0x%08x\n", ck_rv);
        return -1;
    }

    if(ulCount == 0)
    {
        printf("No ET199 Found!\n", ck_rv);
    }
}
```

```
        return -1;
    }

    pSlot = new CK_SLOT_ID[ulCount];
    ck_rv = ET199->C_GetSlotList(TRUE, pSlot, &ulCount);
    if(ck_rv != CKR_OK)
    {
        printf("C_GetSlotList 2 Error! 0x%08x\n", ck_rv);
        goto clean;
    }

    while(TRUE)
    {
        //获得事件消息
        ck_rv = ET199->C_WaitForSlotEvent(CKF_DONT_BLOCK,
                                           &pSlot[0], NULL_PTR);
        if(ck_rv == CKR_OK)
        {
            CK_SLOT_INFO si = {0};
            //得到 Token 槽列表信息
            ck_rv = ET199->C_GetSlotInfo(pSlot[0], &si);
            if(ck_rv == CKR_OK)
            {
                //判断 ET199 是否插入
                if(si.flags & CKF_TOKEN_PRESENT)
                {
                    printf("ET199 Insert\n");
                }
                else
                {
                    printf("ET199 Remove\n");
                }
            }
        }
    }
}
```

```

    }

clean:
    //关闭环境
    ck_rv = ET199->C_Finalize(NULL_PTR);
    if(ck_rv != CKR_OK)
    {
        printf("C_Finalize Error! 0x%08x\n", ck_rv);
    }

    //释放 DLL
    if(hDll)
    {
        FreeLibrary(hDll);
    }

    //释放内存
    if(pSlot != NULL)
    {
        delete [] pSlot;
        pSlot = NULL;
    }

    return 0;
}

```

5.1.6 得到 PIN 码的重试次数和修改 TokenName

标准的 PKCS#11 中没有这样的接口，这时需要利用我们提供的扩展 PKCS#11 来获取 PIN 的重试次数和修改 TokenName。在初始化时也可以设置 TokenName，但需要 SO PIN，而在这里可以直接修改 TokenName。在工程中加入 auxiliary.h 头文件。

```

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h" //PKCS#11 的头文件，由 RSA 公司定义
#include "auxiliary.h" //扩展 PKCS#11 库的头文件

```

```
//定义获取 Cryptoki 库中函数指针表的函数指针
typedef CK_RV (* C_GETFUNCTIONLISTPROC) (CK_FUNCTION_LIST_PTR_PTR);

//定义获取扩展库中函数指针表的函数指针
typedef
CK_DECLARE_FUNCTION_POINTER(CK_RV, EP_GetAuxFunctionList)
(
    AUX_FUNC_LIST_PTR_PTR pAuxFunc
);

int main(int argc, char* argv[])
{
    C_GETFUNCTIONLISTPROC pGetFunctionList = NULL;
    EP_GetAuxFunctionList pE_GetAuxFunctionList = NULL_PTR;
    CK_FUNCTION_LIST_PTR ET199 = NULL;
    AUX_FUNC_LIST_PTR ET199AUX = NULL;
    HINSTANCE hDll = NULL;

    CK_RV ck_rv = 0;
    CK_ULONG ulCount = 0;
    CK_SLOT_ID_PTR pSlot = NULL;
    BYTE pbTokenName[] = "ET199";

    //PIN 信息结构
    AUX_PIN_INFO pininfo={0};

    //加载 PKCS#11 库
    hDll = LoadLibrary("et199csp11.dll");
    if(hDll == NULL)
    {
        printf("Loadlibrary Error!\n");
        return -1;
    }

    //获得一个指向 Cryptoki 库中函数指针表的指针
    pGetFunctionList =
        (C_GETFUNCTIONLISTPROC)GetProcAddress(hDll, "C_GetFunctionList");
```

```
if(pGetFunctionList == NULL)
{
    printf("Get Function List Error!\n");
    return -1;
}

pGetFunctionList(&ET199);
if(ET199 == NULL)
{
    printf("Get Function List Pointer Error!\n");
    return -1;
}

//获得一个指向扩展库中函数指针表的指针
pE_GetAuxFunctionList =
    (EP_GetAuxFunctionList)GetProcAddress(hDll, "E_GetAuxFunctionList");
if(NULL == pE_GetAuxFunctionList)
{
    printf("GetProcAddress Aux Error\n");
    return -1;
}

pE_GetAuxFunctionList(&ET199AUX);
if(NULL == ET199AUX)
{
    printf("Get Token PTR Error\n");
    return -1;
}

//C_Initialize, 初始化环境
ck_rv = ET199->C_Initialize(NULL_PTR);
if(ck_rv != CKR_OK)
{
    printf("C_Initialize Error! 0x%08x\n", ck_rv);
    return -1;
}

//获得计算机系统上的 Token 槽列表
```

```
ck_rv = ET199->C_GetSlotList(TRUE, NULL, &ulCount);
if(ck_rv != CKR_OK)
{
    printf("C_GetSlotList 1 Error! 0x%08x\n", ck_rv);
    return -1;
}

if(ulCount == 0)
{
    printf("No ET199 Found!\n", ck_rv);
    return -1;
}

pSlot = new CK_SLOT_ID[ulCount];
ck_rv = ET199->C_GetSlotList(TRUE, pSlot, &ulCount);
if(ck_rv != CKR_OK)
{
    printf("C_GetSlotList 2 Error! 0x%08x\n", ck_rv);
    goto clean;
}

//设置 TokenName
ck_rv = ((EP_SetTokenLabel)(ET199AUX->pFunc[EP_SET_TOKEN_LABEL]))(
    pSlot[0], CKU_USER, NULL, 0, (CK_CHAR_PTR)pbTokenName);
if(ck_rv != CKR_OK)
{
    printf("P_SET_TOKEN_LABEL Error! 0x%08x\n", ck_rv);
    goto clean;
}

//获得 User PIN 的当前重试次数和最大重试次数
ck_rv = ((EP_GetPinInfo)(ET199AUX->pFunc[EP_GET_PIN_INFO]))(
    pSlot[0], &pininfo);
if(ck_rv != CKR_OK)
{
    printf("EP_GET_PIN_INFO Error! 0x%08x\n", ck_rv);
    goto clean;
}
```

```

printf("UserPinCurCounter: %02x\n", pininfo.bUserPinCurCounter);
printf("UserPinMaxRetries: %02x\n", pininfo.bUserPinMaxRetries);

clean:
    //关闭环境
    ck_rv = ET199->C_Finalize(NULL_PTR);
    if(ck_rv != CKR_OK)
    {
        printf("C_Finalize Error! 0x%08x\n", ck_rv);
    }

    //释放 DLL
    if(hDll)
    {
        FreeLibrary(hDll);
    }

    //释放内存
    if(pSlot != NULL)
    {
        delete [] pSlot;
        pSlot = NULL;
    }

    return 0;
}

```

5.1.7 读写数据

在使用 PKCS#11 接口进行数据读写时，需要先设置一个数据模版，这个模版中定义了数据的属性，包括：数据类型，是否在硬件中，在公有区还是私有区，数据内容等。模版设置好后，就可以按模版创建和查找数据对象了。创建或者查找数据对象后会得到一个对象句柄，使用这个句柄就可以读写数据。见下面的代码：

- **创建数据对象**

```
CK_RV ck_rv;
```

```
//数据类型：普通数据
CK_OBJECT_CLASS objClass = CKO_DATA;
CK_BBOOL bTrue = TRUE;
CK_BBOOL bFalse = FALSE;
//数据对象名称
CK_BYTE pbLabel[] = "test";
//数据对象句柄
CK_OBJECT_HANDLE hObject = NULL;

//数据对象模版
CK_ATTRIBUTE pDataTemplate[]={

    //数据类型
    {CKA_CLASS, &objClass, sizeof(CK_OBJECT_CLASS)},

    //存储在 ET199 硬件中，设为 TRUE
    {CKA_TOKEN, &bTrue, sizeof(CK_BBOOL)},

    //是否存储在私有区里，
    //为 TRUE，在私有区创建，需要验证 User PIN 才能读写
    //为 FALSE，在公有区创建，不需要验证 User PIN 就可以读写
    {CKA_PRIVATE, &bFalse, sizeof(CK_BBOOL)},

    //数据的标识
    {CKA_LABEL, pbLabel, sizeof(pbLabel)},

    //数据内容，在创建时设为 NULL
    {CKA_VALUE, NULL, 0},
};

//创建数据对象，第三个参数 5 表示模版中有五项
//创建对象的句柄在 hObject 中
ck_rv = ET199->C_CreateObject(hSession, pDataTemplate, 5, &hObject);
if(ck_rv != CKR_OK)
{
    printf("C_CreateObject Error! 0x%08x\n", ck_rv);
    goto clean;
}
```



```

}
else
{
    printf("Create Data OK!\n");
}

```

● 查找数据对象

查找数据对象分为三步：C_FindObjectsInit, C_FindObjects和FindObjectsFinal。是按数据模版的项来查找数据的。

```
CK_ULONG ulFindObjectCount = 0;
```

```
//查找数据对象
```

```
//按照模板的前四项查找，即数据对象，硬件里面，私有区和标签符合的
```

```
ck_rv = ET199->C_FindObjectsInit(hSession, pDataTemplate, 4);
```

```
if(ck_rv != CKR_OK)
```

```

{
    printf("C_FindObjectsInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

```

```
//将查找到的数据对象的句柄放到 hObject 中
```

```
//第 3 个参数见 PKCS#11 的接口文档
```

```
//当为 0 时，查找所有符合标准的数据对象
```

```
//当为 1 时，查找一个符合标准的数据对象
```

```
hObject = 0;
```

```
ck_rv = ET199->C_FindObjects(hSession, &hObject, 1, &ulFindObjectCount);
```

```
if(ck_rv != CKR_OK)
```

```

{
    printf("C_FindObjects Error! 0x%08x\n", ck_rv);
    goto clean;
}

```

```
ck_rv = ET199->C_FindObjectsFinal(hSession);
```

```
if(ck_rv != CKR_OK)
```

```

{
    printf("C_FindObjectsFinal Error! 0x%08x\n", ck_rv);
    goto clean;
}

```

```
}

if(ulFindObjectCount == 0)
{
    printf("Not Find Data To Read!\n");
    goto clean;
}
```

● 写入数据

在写入数据前要先查找数据对象，获得数据对象的句柄。由于数据模版中一些属性是不能写的，这要看具体的 PKCS#11 库中的实现方法，为了做到兼容，建议在写数据时再声明一个新的模版，只包括数据内容一项。

```
CK_BYTE pbData [] = "Hello World!";
DWORD dwDataLen = sizeof(pbData);

//写入数据模版， pbData 和 dwDataLen 是写入数据的内容和长度
CK_ATTRIBUTE pWriteTemplate[]={
    {CKA_VALUE, (void*)pbData, dwDataLen},
}

//写入数据
ck_rv = ET199->C_SetAttributeValue(hSession, hObject, pWriteTemplate, 1);
if(ck_rv != CKR_OK)
{
    printf("C_SetAttributeValue Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Write Data OK!\n");
}
```

● 读取数据

与写入数据一样，在读取数据时也要先查找数据对象，同时建议声明一个读取数据的模版。

```
CK_BYTE_PTR pbReadData = NULL;
```

```

DWORD dwReadDataLen =0;

//模版中为 NULL
CK_ATTRIBUTE pReadTemplate[]={
    {CKA_VALUE, NULL, 0},
}

//读取数据对象
//第一次调用先得到数据长度
ck_rv = ET199->C_GetAttributeValue(hSession, hObject, pReadTemplate, 1);
if(ck_rv != CKR_OK)
{
    printf("C_GetAttributeValue 1 Error! 0x%08x\n", ck_rv);
    goto clean;
}

//分配空间，多分配一个字节补 0，为字符串结尾
dwReadDataLen = pReadTemplate[0].ulValueLen;
pbReadData = new CK_BYTE[dwReadDataLen + 1];
memset(pbReadData, 0, dwReadDataLen + 1);

pReadTemplate[0].pValue = pbReadData;
pReadTemplate[0].ulValueLen = dwReadDataLen;

ck_rv = ET199->C_GetAttributeValue(hSession, hObject, pReadTemplate, 1);
if(ck_rv != CKR_OK)
{
    printf("C_GetAttributeValue 2 Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Read Data OK! Len: %d Data: %s\n", dwReadDataLen, pbReadData);
}

```

● 删除数据

删除数据前要先查找数据，得到数据对象句柄。

```
ck_rv = ET199->C_DestroyObject(hSession, hObject);
```

```

if(ck_rv != CKR_OK)
{
    printf("C_DestroyObject Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Delete Data OK!\n");
}

```

5.1.8 对称加解密

对称加解密算法的密钥是相同的，即加密和解密使用的密钥是同一个。密钥的长度也是固定的，如 DES：8 个字节，3DES：16 个字节或者 24 字节。加密的数据的长度也要是固定的，一般为 8 的倍数。目前流行的对称加解密算法有 DES，3DES 等。

ECB 和 CBC 是加解密中常用的两种模式。在对称加解密中，数据是按块来进行的，DES 和 3DES 都是 8 个字节为一块。ECB 模式下，块与块之间没有关系，是各自独立的。CBC 模式下，块与块之间是有关系的，前块的加密结果与后块做异或，然后再进行加密，以此类推。

在对称加解密时，数据是分块进行的，DES 和 3DES 都是 8 个字节为一块。这时如果加密的数据不是 8 的倍数，就需要进行补齐。补齐的规则可以由开发商自己来设定。常见的方式为，少几个字节就补上该数字。如：原数据为 0x11 0x22 0x33 三个字节，这时缺少 5 个字节，那么后面都补数字 5，即：0x11 0x22 0x33 0x05 0x05 0x05 0x05 0x05，这时解密后读取最后一个字节，然后去掉相应的 5 个字节，即得到原文。如果原文正好是 8 的倍数，那么就多补 8 个字节，都赋值为 0x08，这样解密后，最后一个字节是 0x08，去掉 8 个字节后就是原文。下面以 3DES 算法来说明，**注意在使用对称加解密时都需要**

先验证 User PIN。

● 创建对称密钥

```

//User PIN,
BYTE pbUserPIN[] = "1234";
DWORD dwUserPINLen = strlen((const char*)pbUserPIN);

//密钥名称
BYTE pKeyName [] = "testkey";
DWORD dwKeyNameLen = strlen((const char*)pKeyName);

```

```
//密钥内容, 3DES 为 24 字节
CK_BYTE pbKey[] = "123456781234567812345678";
DWORD dwKeyLen = strlen((const char*)pbKey);

//数据对象类型, 为密钥类型对象
CK_OBJECT_CLASS objClass = CKO_SECRET_KEY;
//密钥类型, 为 3DES 密钥
CK_KEY_TYPE keyType = CKK_DES3;
//密钥对象模版
CK_ATTRIBUTE pSymKeyTemplate[] = {

    //数据对象类型
    {CKA_CLASS, &objClass, sizeof(CK_OBJECT_CLASS)},
    //密钥类型
    {CKA_KEY_TYPE, &keyType, sizeof(CK_KEY_TYPE)},
    //存储在硬件中
    {CKA_TOKEN, &bTrue, sizeof(CK_BBOOL)},
    //密钥名称
    {CKA_LABEL, (CK_UTF8CHAR*)&pKeyName, dwKeyNameLen},
    //存储在私有区, 需要 User PIN 验证后才可以使用, 对称密钥是读不出来的
    {CKA_PRIVATE, &bTrue, sizeof(CK_BBOOL)},
    //密钥可以加密
    {CKA_ENCRYPT, &bTrue, sizeof(CK_BBOOL)},
    //密钥可以解密
    {CKA_DECRYPT, &bTrue, sizeof(CK_BBOOL)},
    //密钥内容
    {CKA_VALUE, pbKey, dwKeyLen},
    //密钥长度
    {CKA_VALUE_LEN, &dwKeyLen, sizeof(CK_ULONG)},
};

CK_OBJECT_HANDLE hObject = NULL;

//使用前需先验证 User PIN
ck_rv = ET199->C_Login(hSession, CKU_USER, pbUserPIN, dwUserPINLen);
if(ck_rv != CKR_OK)
{
```

```
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Verify User PIN OK!\n");
}

//创建 3DES 对称密钥, 第 3 个参数表示模版中有 9 项
ck_rv = ET199->C_CreateObject(hSession, pSymKeyTemplate, 9, &hObject);
if(ck_rv != CKR_OK)
{
    printf("C_CreateObject Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Create SysKey Object OK!\n");
}
```

● 查找密钥对象

在使用 3DES 密钥进行对称加解密时, 需要先查找密钥对象, 得到密钥的句柄。

```
DWORD dwObjectCount = 0;

//按模版的前 4 项查找密钥, 即按密钥名称查找
ck_rv = ET199->C_FindObjectsInit(hSession, pSymKeyTemplate, 4);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjectsInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

ck_rv = ET199->C_FindObjects(hSession, &hObject, 1, &dwObjectCount);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjects Error! 0x%08x\n", ck_rv);
    goto clean;
}
```

```
}
```

```
ck_rv = ET199->C_FindObjectsFinal(hSession);
if(dwObjectCount <= 0)
    goto clean;
```

● 加密 (3DES)

```
//源数据
```

```
BYTE pbPlainData[] = "hello world!";
DWORD dwPlainDataLen = strlen((const char*)pbPlainData);
```

```
//加密时使用的数据
```

```
BYTE* pbPlainDataIN;
DWORD dwPlainDataLenIN = 0;
```

```
//加密后的数据
```

```
BYTE* pbCipherData = NULL;
DWORD dwCipherDataLen = 0;
```

```
//块大小
```

```
BYTE dwBlockSize = 8;
DWORD dwP = 0, dwQ = 0;
```

```
//加密机制
```

```
CK_MECHANISM ckMechanism = {0, NULL_PTR, 0};
CK_BYTE iv[8] = {0};
ckMechanism.mechanism = CKM_DES3_ECB;    //以 3DES 的 ECB 方式
ckMechanism.pParameter = iv;
ckMechanism.ulParameterLen = 8;
```

```
//填充数据。
```

```
dwP = dwPlainDataLen / dwBlockSize;
dwQ = dwPlainDataLen % dwBlockSize;
dwPlainDataLenIN = (dwP + 1) * dwBlockSize;
pbPlainDataIN = new CK_BYTE[dwPlainDataLenIN];
```

```
memset(pbPlainDataIN, 0, dwPlainDataLenIN);
```

```
memcpy(pbPlainDataIN, pbPlainData, dwPlainDataLen);
memset(pbPlainDataIN + dwP * dwBlockSize + dwQ,
        dwBlockSize - dwQ, dwBlockSize - dwQ);

//加密初始化
ck_rv = ET199->C_EncryptInit(hSession, &ckMechanism, hObject);
if(ck_rv != CKR_OK)
{
    printf("C_EncryptInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

//第一次调用, 得到加密后的数据长度
ck_rv = ET199->C_Encrypt(hSession, pbPlainDataIN, dwPlainDataLenIN,
        NULL, &dwCipherDataLen);
if(ck_rv != CKR_OK)
{
    printf("C_Encrypt Error! 0x%08x\n", ck_rv);
    goto clean;
}

//分配空间
pbCipherData = new CK_BYTE[dwCipherDataLen];

//第二次调用, 得到加密后的数据
ck_rv = ET199->C_Encrypt(hSession, pbPlainDataIN, dwPlainDataLenIN,
        pbCipherData, &dwCipherDataLen);
if(ck_rv != CKR_OK)
{
    printf("C_Encrypt Error! 0x%08x\n", ck_rv);
    goto clean;
}

//打印出加密后的数据
printf("CipherDataLen:%d", dwCipherDataLen);
for(i=0; i<dwCipherDataLen; ++i)
{
    if(i%16 == 0)
```



```

        printf("\n");
        printf("%02x ", pbCipherData[i]);
    }
    printf("\n");

```

● 解密 (3DES)

```

//临时数据
BYTE* pbTemp = NULL;
DWORD dwTempLen = 0;

//解密后的数据
BYTE* pbPlainDataOut;
DWORD dwPlainDataLenOut = 0;

//解密初始化
ck_rv = ET199->C_DecryptInit(hSession, &ckMechanism, hObject);
if(ck_rv != CKR_OK)
{
    printf("C_DecryptInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

//第一次调用得到解密后数据的长度
ck_rv = ET199->C_Decrypt(hSession, pbCipherData, dwCipherDataLen,
                        NULL, &dwTempLen);
if(ck_rv != CKR_OK)
{
    printf("C_Decrypt Error! 0x%08x\n", ck_rv);
    goto clean;
}

pbTemp = new BYTE[dwTempLen];

ck_rv = ET199->C_Decrypt(hSession, pbCipherData, dwCipherDataLen,
                        pbTemp, &dwTempLen);
if(ck_rv != CKR_OK)
{

```

```

    printf("C_Decrypt Error! 0x%08x\n", ck_rv);
    goto clean;
}

//去掉补齐, 得到原文的数据
dwPlainDataLenOut = dwTempLen - pbTemp[dwTempLen - 1];
pbPlainDataOut = new BYTE[dwPlainDataLenOut + 1];
memset(pbPlainDataOut, 0, dwPlainDataLenOut + 1);
memcpy(pbPlainDataOut, pbTemp, dwPlainDataLenOut);

printf("PlainDataLen:%d  Value:%s\n", dwPlainDataLenOut, pbPlainDataOut);

```

5.1.9 RSA 操作

非对称加解密的加密密钥和解密密钥是不同的, 分为公钥和私钥。公钥是公开的密钥, 可以散发给其他人; 私钥是秘密的, 需要拥有者本人自己掌握, 不能透露给任何人。公钥和私钥是匹配的, 通常称为公私钥对。一般使用公钥进行加密, 即私钥的主人将与之匹配的公钥散发给其他人, 其他人使用这个公钥进行加密, 将加密后的数据发送给私钥主人, 这时只有与这个公钥匹配的私钥才能正确解密数据。目前流行的非对称加解密算法有 RSA 算法等。RSA 密钥对通常分为 512 位, 1024 位和 2048 位, 位数越高, 加密强度越强, 但运算速度也越慢。ET199 采用高性能智能卡芯片, 在硬件中支持 512 位, 1024 位和 2048 位 RSA 运算。

数字签名的功能是防止数据在传输中被篡改。整个功能是使用 RSA 密钥对运算来实现的。数字签名的过程为:

- (1) 先将数据使用散列算法进行散列, 当然也可以直接使用明文数据, 但如果明文数据比较大, RSA 运算将会比较慢。一般都采用先将明文数据散列的方法, 得到一个固定长度的数据。
- (2) 将 (1) 中的散列后的数据使用私钥进行签名操作, 将原文数据和签名的结果发送给接收者。
- (3) 接收者使用与 (1) 中相同的散列算法, 将原文数据进行散列, 然后使用与私钥匹配的公钥, 把散列的结果与签名后的结果进行验签, 如果验证签名成功表示数据在传输过程中没有被篡改。

● 产生 RSA 密钥对

使用 C_GenerateKeyPair 函数来产生 RSA 密钥对, 需要先定义公钥模版和私钥模版,

代码如下：

```

CK_BBOOL bTrue = TRUE;
CK_ULONG ulModulusBits = 1024;           //1024 位 RSA 密钥
CK_BYTE subject[] = "Sample RSA Key Pair"; //密钥名称
CK_ULONG keyType = CKK_RSA;

CK_OBJECT_HANDLE hPubKey = 0;             //公钥句柄
CK_OBJECT_CLASS pubClass = CKO_PUBLIC_KEY; //公钥类型
//公钥模版
CK_ATTRIBUTE pubTemplate[] =
{
    //数据对象类型：公钥
    {CKA_CLASS,      &pubClass,      sizeof(pubClass)},
    //密钥类型：CKK_RSA
    {CKA_KEY_TYPE,   &keyType,       sizeof(keyType)},
    //密钥名称："Sample RSA Key Pair"
    {CKA_SUBJECT,    subject,         sizeof(subject)},
    //密钥位数：1024
    {CKA_MODULUS_BITS, &ulModulusBits, sizeof(ulModulusBits)},
    //是否支持加密：支持
    {CKA_ENCRYPT,      &bTrue,         sizeof(bTrue)},
    //是否在 Token 中：是
    {CKA_TOKEN,        &bTrue,         sizeof(bTrue)},
};

CK_OBJECT_HANDLE hPriKey = 0;             //私钥句柄
CK_OBJECT_CLASS priClass = CKO_PRIVATE_KEY; //私钥类型
//私钥模版
CK_ATTRIBUTE priTemplate[] = {
    //数据对象类型：私钥
    {CKA_CLASS,      &priClass,      sizeof(priClass)},
    //密钥类型：CKK_RSA
    {CKA_KEY_TYPE,   &keyType,       sizeof(keyType)},
    //密钥名称："Sample RSA Key Pair"
    {CKA_SUBJECT,    subject,         sizeof(subject)},
    //是否支持解密：支持
    {CKA_DECRYPT,      &bTrue,         sizeof(bTrue)},
};

```

```

        //私钥的属性放在私有区，私钥的内容是放在其他安全区域中的
        {CKA_PRIVATE,      &bTrue,      sizeof(bTrue)},
        //是否在 Token 中：是
        {CKA_TOKEN,       &bTrue,       sizeof(bTrue)},
    };
    //算法机制
    CK_MECHANISM keyGenMechanism = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0};

    //验证 UserPIN
    ck_rv = ET199->C_Login(hSession, CKU_USER,
                          (CK_UTF8CHAR_PTR)pbUserPIN, dwUserPINLen);
    if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
    {
        printf("C_Login Error! 0x%08x\n", ck_rv);
        goto clean;
    }

    //产生 1024 位 RSA 密钥对
    ck_rv = ET199->C_GenerateKeyPair(hSession, &keyGenMechanism,
                                    pubTemplate, sizeof(pubTemplate)/sizeof(CK_ATTRIBUTE),
                                    priTemplate, sizeof(priTemplate)/sizeof(CK_ATTRIBUTE),
                                    &hPubKey, &hPriKey);
    if(ck_rv != CKR_OK)
    {
        printf("C_GenerateKeyPair Error! 0x%08x\n", ck_rv);
        goto clean;
    }

```

● 公钥加密

加密前先要使用 C_FindObjects 函数查找公钥对象，使用公钥时不需要进行 User PIN 验证。在加密时，做了填充，每次的加密结果都不同，但都能解出正确的原文。代码如下：

```

#define KEYMODULUSLEN 128    //1024 位 RSA 密钥的 MODULUS 为 128 字节
#define KEYPADLEN      12    //补齐字节

//块数
int iBlockNum = 0;

```

```
//原文内容
BYTE pbPlainData[] = "Hello World!";
//原文长度
DWORD dwPlainDataLen = strlen((const char*)pbPlainData);
//密文数据指针
BYTE* pbCipherData = NULL;
//密文数据长度
DWORD dwCipherDataLen = 0;

CK_BBOOL bTrue = TRUE;
CK_BBOOL bFalse = FALSE;

CK_OBJECT_HANDLE hObject = NULL;
CK_ULONG ulObjectCount = 0;

CK_MECHANISM ckMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
CK_OBJECT_CLASS objClassPub = CKO_PUBLIC_KEY;
//公钥模版
CK_ATTRIBUTE pPubKeyTemplate[] = {
    {CKA_CLASS, &objClassPub, sizeof(CKO_PUBLIC_KEY)},
    {CKA_ENCRYPT, &bTrue, sizeof(CK_BBOOL)},
    {CKA_TOKEN, &bTrue, sizeof(CK_BBOOL)},
    {CKA_PRIVATE, &bFalse, sizeof(CK_BBOOL)},
    {CKA_LABEL, NULL, 0},
};

//查找公钥对象初始化，按照公钥模板的前四项查找
ck_rv = ET199->C_FindObjectsInit(hSession, pPubKeyTemplate, 4);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjectsInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

hObject = 0; //将句柄初始
//查找公钥对象
ck_rv = ET199->C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
```

```
if(ck_rv != CKR_OK)
{
    printf("C_FindObjects Error! 0x%08x\n", ck_rv);
    goto clean;
}

//查找公钥对象结束
ck_rv = ET199->C_FindObjectsFinal(hSession);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjectsFinal Error! 0x%08x\n", ck_rv);
    goto clean;
}

//使用公钥加密, 可以查看 MSDN,
//1024 位的公钥加密时 128 字节为一块, 但有 12 个字节作为 PAD
iBlockNum = (dwPlainDataLen / (KEYMODULUSLEN - KEYPADLEN)) + 1;
dwCipherDataLen = iBlockNum * KEYMODULUSLEN;
pbCipherData = new BYTE[dwCipherDataLen];

//公钥加密初始化
ck_rv = ET199->C_EncryptInit(hSession, &ckMechanism, hObject);
if (ck_rv != CKR_OK)
{
    printf("C_EncryptInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

//公钥加密
ck_rv = ET199->C_Encrypt(hSession, pbPlainData, dwPlainDataLen,
pbCipherData, &dwCipherDataLen);
if (ck_rv != CKR_OK)
{
    printf("C_Encrypt Error! 0x%08x\n", ck_rv);
    goto clean;
}

//打印加密结果
```

```
printf("CipherDataLen:%d", dwCipherDataLen);
for(i=0; i<dwCipherDataLen; ++i)
{
    if(i%16 == 0)
        printf("\n");
    printf("%02x ", pbCipherData[i]);
}
```

● 私钥解密

在使用私钥解密数据时，需要先验证 User PIN，然后使用 C_FindObjects 函数查找私钥对象。使用上面加密后的密文 pbCipherData 进行解密，由于填充的原因，虽然密文不同，但都能解出正确的原文。代码如下：

```
//临时数据
BYTE* pbTemp = NULL;
DWORD dwTempLen = 0;

CK_BBOOL bTrue = true;
CK_BBOOL bFalse = false;
CK_MECHANISM ckMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
CK_OBJECT_HANDLE hObject = NULL;
CK_ULONG ulObjectCount = 0;

CK_OBJECT_CLASS objClassPri = CKO_PRIVATE_KEY;
CK_ATTRIBUTE pPriKeyTemplate[] = {
    {CKA_CLASS, &objClassPri, sizeof(CKO_PRIVATE_KEY)},
    {CKA_DECRYPT, &bTrue, sizeof(CK_BBOOL)},
    {CKA_TOKEN, &bTrue, sizeof(CK_BBOOL)},
    {CKA_PRIVATE, &bTrue, sizeof(CK_BBOOL)},
    {CKA_LABEL, NULL, 0},
};

//使用私钥解密
//使用私钥必须验证 User PIN
ck_rv = ET199->C_Login(hSession,
    CKU_USER, (unsigned char*)"1234", strlen("1234"));
if(ck_rv != CKR_OK)
{
```

```
    printf("C_Login Error! 0x%08x\n", ck_rv);
    goto clean;
}
else
{
    printf("Verify User PIN OK!\n");
}

//查找私钥对象初始化，按私钥模版的前四项查找
ck_rv = ET199->C_FindObjectsInit(hSession, pPriKeyTemplate, 4);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjectsInit Error! 0x%08x\n", ck_rv);
    goto clean;
}

hObject = 0; //将句柄初始
//查找私钥对象
ck_rv = ET199->C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjects Error! 0x%08x\n", ck_rv);
    goto clean;
}

//查找私钥对象结束
ck_rv = ET199->C_FindObjectsFinal(hSession);
if(ck_rv != CKR_OK)
{
    printf("C_FindObjectsFinal Error! 0x%08x\n", ck_rv);
    goto clean;
}

//私钥解密初始化
ck_rv = ET199->C_DecryptInit(hSession, &ckMechanism, hObject);
if (ck_rv != CKR_OK)
{
    printf("C_DecryptInit Error! 0x%08x\n", ck_rv);
```



```

        goto clean;
    }

    pbTemp = new BYTE[dwCipherDataLen];
    memset(pbTemp, 0, dwCipherDataLen);

    //私钥解密
    //将上面加密后的数据 pbCipherData 进行解密
    //dwCipherDataLen 必须为 128 的倍数 (RSA 为 1024 位时)
    ck_rv = ET199->C_Decrypt(hSession, pbCipherData, dwCipherDataLen,
                             pbTemp, &dwTempLen);

    //打印解密结果
    printf("PlainDataLen:%d  Value:%s\n", dwTempLen, pbTemp);

```

● 使用私钥签名

使用私钥进行签名前要先验证 User PIN, 然后使用 C_FindObjects 函数查找私钥对象。签名前要先对数据进行 MD5 或者 SHA1 散列操作, 然后将散列后的值根据 PKCS 的标准进行填充, 再进行签名操作。使用 PKCS#11 接口签名的结果与使用 CAPI 接口签名的结果在顺序上是相反的。见下面的代码:

```

CK_BBOOL bTrue = TRUE;
CK_ULONG ulObjectCount = 0;
CK_OBJECT_HANDLE hPriObject = NULL;
CK_OBJECT_CLASS priObject = CKO_PRIVATE_KEY;
//私钥模版
CK_ATTRIBUTE priTemplate[] = {
    {CKA_CLASS, &priObject, sizeof(priObject)},
    {CKA_TOKEN, &bTrue, sizeof(bTrue)},
    {CKA_PRIVATE, &bTrue, sizeof(bTrue)},
};

//散列算法机制, 这里使用 SHA1 散列算法
CK_MECHANISM digestMechan[] = {CKM_SHA_1, NULL_PTR, 0};
//签名数据
CK_BYTE pbMsg[] = "123456";
//签名数据长度
CK_ULONG cbMsg = strlen((const char*)pbMsg);

```

```
//散列结果
CK_BYTE_PTR pbMsgSHA1 = NULL;
//散列结果长度
CK_ULONG cbMsgSHA1 = 0;

//MD5 填充的数据
CK_BYTE TMD5[34] = {
    0x30, 0x20, 0x30, 0x0c, 0x06, 0x08, 0x2a, 0x86,
    0x48, 0x86, 0xf7, 0x0d, 0x02, 0x05, 0x05, 0x00,
    0x04, 0x10};

//SHA1 填充的数据
CK_BYTE TSHA1[35] = {
    0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0e,
    0x03, 0x02, 0x1a, 0x05, 0x00, 0x04, 0x14};

//算法机制
CK_MECHANISM ckMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
//签名结果
CK_BYTE_PTR pbSignedData = NULL;
//签名结果长度
CK_ULONG cbSignedData = 0;

//验证 PIN, 因为要使用私钥签名
ck_rv = ET199->C_Login(hSession, CKU_USER, (unsigned char*)"1234", 4);
if(CKR_OK != ck_rv)
{
    printf("C_Login Error: %08x\n", ck_rv);
    goto clean;
}

//查找私钥对象初始化
ck_rv = ET199->C_FindObjectsInit(hSession, priTemplate, 3);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjectsInit Pri Error: %08x\n", ck_rv);
    goto clean;
}
```

```
//查找私钥对象
ck_rv = ET199->C_FindObjects(hSession, &hPriObject, 1, &ulObjectCount);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjects Pri Error: %08x\n", ck_rv);
    goto clean;
}

//查找私钥对象结束
ck_rv = ET199->C_FindObjectsFinal(hSession);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjectsFinal Pri Error: %08x\n", ck_rv);
    goto clean;
}

//散列初始化
ck_rv = ET199->C_DigestInit(hSession, digestMechan);
if(CKR_OK != ck_rv)
{
    printf("C_DigestInit Error: %08x\n", ck_rv);
    goto clean;
}

//将原文数据进行 SHA1 散列，第一次先得到散列后的数据长度
ck_rv = ET199->C_Digest(hSession, pbMsg, cbMsg, NULL, &cbMsgSHA1);
if(CKR_OK != ck_rv)
{
    printf("C_Digest 1 Error: %08x\n", ck_rv);
    goto clean;
}

//分配空间
pbMsgSHA1 = new BYTE[cbMsgSHA1];
memset(pbMsgSHA1, 0, cbMsgSHA1);

//将原文数据进行 SHA1 散列，第二次得到散列后的数据结果
```

```
ck_rv = ET199->C_Digest(hSession, pbMsg, cbMsg, pbMsgSHA1, &cbMsgSHA1);
if(CKR_OK != ck_rv)
{
    printf("C_Digest 2 Error: %08x\n", ck_rv);
    goto clean;
}

//将散列后的数据填充到 TSHA1 中, 前 15 个字节为 PKCS 标准中规定的固定值
memcpy(TSHA1+15, pbMsgSHA1, cbMsgSHA1);

//使用私钥签名初始化
ck_rv = ET199->C_SignInit(hSession, &ckMechanism, hPriObject);
if(CKR_OK != ck_rv)
{
    printf("C_SignInit Error: %08x\n", ck_rv);
    goto clean;
}

//使用私钥签名, 第一次得到签名后的数据长度
ck_rv = ET199->C_Sign(hSession, TSHA1, 35, NULL, &cbSignedData);
if(CKR_OK != ck_rv)
{
    printf("C_Sign 1 Error: %08x\n", ck_rv);
    goto clean;
}

//分配空间
pbSignedData = new CK_BYTE[cbSignedData];
memset(pbSignedData, 0, cbSignedData);

//使用私钥签名, 第二次得到签名后的数据结果
ck_rv = ET199->C_Sign(hSession, TSHA1, 35, pbSignedData, &cbSignedData);
if(CKR_OK != ck_rv)
{
    printf("C_Sign 2 Error: %08x\n", ck_rv);
    goto clean;
}
```

```
//打印签名结果
for(CK_ULONG i=0; i<cbSignedData; ++i)
{
    if(i%16 == 0)
        printf("\n");
    printf("%02x ", pbSignedData[i]);
}
printf("\n");
```

● 使用公钥验证签名

使用公钥验签时，先要使用 C_FindObjects 函数查找公钥对象。验证签名时不需要验证 User PIN。代码如下：

```
CK_ULONG ulTokenCount = 0;

//查找公钥对象，用于验签
CK_OBJECT_CLASS pubObject = CKO_PUBLIC_KEY;
CK_OBJECT_HANDLE hPubObject = NULL;
//公钥模版
CK_ATTRIBUTE pubTemplate[] = {
    {CKA_CLASS, &pubObject, sizeof(pubObject)},
    {CKA_TOKEN, &bTrue, sizeof(bTrue)},
    {CKA_MODULUS, NULL, 0},
    {CKA_PUBLIC_EXPONENT, NULL, 0}
};

//算法机制
CK_MECHANISM ckMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};

//查找公钥对象初始化
ck_rv = ET199->C_FindObjectsInit(hSession, pubTemplate, 2);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjectsInit Pub Error: %08x\n", ck_rv);
    goto clean;
}

//查找公钥对象
```

```
ck_rv = ET199->C_FindObjects(hSession, &hPubObject, 1, &ulObjectCount);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjects Pub Error: %08x\n", ck_rv);
    goto clean;
}

//查找公钥对象结束
ck_rv = ET199->C_FindObjectsFinal(hSession);
if(CKR_OK != ck_rv)
{
    printf("C_FindObjectsFinal Pub Error: %08x\n", ck_rv);
    goto clean;
}

//使用公钥验证签名初始化
ck_rv = ET199->C_VerifyInit(hSession, &ckMechanism, hPubObject);
if(CKR_OK != ck_rv)
{
    printf("C_VerifyInit Pub Error: %08x\n", ck_rv);
    return -1;
}

//使用公钥验证签名
//pbSignedData 是上面签名的数据结果
//cbSignedData 必须为 128 的倍数 (RSA 为 1024 位时)
ck_rv = ET199->C_Verify(hSession, TSHA1, 35, pbSignedData, cbSignedData);
if(CKR_OK != ck_rv)
{
    printf("C_Verify Pub Error: %08x\n", ck_rv);
    return -1;
}
else
{
    printf("Verified OK!\n");
}
```

5.1.10 低级初始化

可以使用 ET199 提供的辅助 PKCS#11 接口 (auxiliary.h) 来完成低级初始化, 即 PKI 设置工具中初始化为 PKI 格式的功能 (见 4.2.2 节)。通过低级初始化可以将管理员 PIN (SO PIN) 锁死的硬件重新初始化为 PKI 格式。见 samples\PKCS11\VC\FormatKey 目录中的 VC 工程。

其中的初始结构为:

```
AUX_INIT_TOKEN_LOWLEVL_PKI param = {0};
param.strTokenName      //Label 名称
param.strSOPin          //管理员 PIN (SO PIN)
param.strUserPin        //用户 PIN (User PIN);
param.ucSOMaxPinEC      //管理员 PIN 最大重试次数
param.ucUserMaxPinEC    //用户 PIN 最大重试次数
param.nRSAKeyPairCount  //RSA 密钥对最大数, 一般为 9
param.ulPubSize          //公有区大小
param.ulPrvSize          //私有区大小
param.nDSAKeyPairCount  //密钥对最大数目, 一般为 9
```

取值范围的注意事项参见 4.2.2 节的说明。

5.1.11 解锁 USER PIN

可以使用 PKCS#11 接口中的 C_InitPIN 来解锁 USER PIN, 解锁操作前需要先验证 SO PIN。

//以 SO PIN 登录

```
ck_rv = ET199->C_Login(hSession, CKU_SO, (CK_UTF8CHAR_PTR)pbSoPIN, ulSoPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error: %08x\n", ck_rv);
    return -1;
}
```

//重新设置 USER PIN

```
ck_rv = ET199->C_InitPIN(hSession, (CK_UTF8CHAR_PTR)pbUserPIN, ulUserPINLen);
if(ck_rv != CKR_OK)
{
    printf("InitPIN Error: %08x\n", ck_rv);
}
```

```
    return -1;
}
//退出 S0 PIN 登录状态
ck_rv = ET199->C_Logout(hSession);
```

5.1.12 修改 USER PIN 和 S0 PIN

可以使用 PKCS#11 接口中的 C_SetPIN 来修改 USER PIN 和 S0 PIN。修改前需要先使用 C_Login 接口来验证 USER PIN 和 S0 PIN。

```
//修改 S0 PIN
ck_rv = ET199->C_Login(hSession, CKU_S0,
                        (CK_UTF8CHAR_PTR)pbOldSoPIN, ulOldSoPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error: %08x\n", ck_rv);
    return -1;
}

ck_rv = ET199->C_SetPIN(hSession,
                        (CK_UTF8CHAR_PTR)pbOldSoPIN, ulOldSoPINLen,
                        (CK_UTF8CHAR_PTR)pbNewSoPIN, ulNewSoPINLen);
if(ck_rv != CKR_OK)
{
    printf("C_SetPIN Error: %08x\n", ck_rv);
    return -1;
}

//修改 USER PIN
ck_rv = ET199->C_Login(hSession, CKU_USER,
                        (CK_UTF8CHAR_PTR)pbOldUserPIN, ulOldUserPINLen);
if(!((ck_rv == CKR_OK) || (ck_rv == CKR_USER_ALREADY_LOGGED_IN)))
{
    printf("C_Login Error: %08x\n", ck_rv);
    return -1;
}

ck_rv = ET199->C_SetPIN(hSession,
```



```

        (CK_UTF8CHAR_PTR)pbOldUserPIN, ulOldUserPINLen,
        (CK_UTF8CHAR_PTR)pbNewUserPIN, ulNewUserPINLen);
if(ck_rv != CKR_OK)
{
    printf("C_SetPIN Error: %08x\n", ck_rv);
    return -1;
}

```

5.2 CAPI 接口

在使用微软的 CAPI 接口进行开发时应注意，有些定义在微软每年发布的 PSDK 中，因此需要先安装 PSDK。下面的连接是 PSDK2003 的下载地址：

<http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdk-full.htm>

下载并安装后，需要在 VC 中进行配置。在这里我们假设把 PSDK 安装到 C:\Program Files\Microsoft SDK 目录下，在 VC6 中配置为：

(1)Tools->Options->Directories 下的 Show directories for 中，include files 下面新增一个 C:\PROGRAM FILES\MICROSOFT SDK\INCLUDE，然后把这个新增的项提到最上面。

(2)Tools->Options->Directories 下的 Show directories for 中，Library files 下面新增一个 C:\PROGRAM FILES\MICROSOFT SDK\LIB，然后把这个新增的项提到最上面。

使用微软的 CAPI 进行编程时，先要知道一些概念。

- 容器：在 CAPI 的接口中有容器的概念，数字证书（证书，公钥和私钥）都是放在容器中的，每个容器有不同的名称，一个容器中只能存放一张签名证书和一张加密证书。如不指定容器名称，ET199 会使用一个默认的 GUID 作为容器名。
- 签名证书：只用来进行签名操作的证书，不能进行加解密操作。RSA 密钥类型为 AT_SIGNATURE。
- 加密证书：可以进行签名和加解密操作。RSA 密钥类型为 AT_KEYEXCHANGE。

在工程中需要加入 wincrypt.h 头文件（#include <wincrypt.h>），并加入 Crypt32.lib 的库。

另外 CAPI 接口不能完成所有 PKCS#11 接口所完成的功能，如：修改 PIN 码，读写数据等，您的应用中要用到这些功能就必须使用 PKCS#11 接口来实现。

5.2.1 枚举 ET199 硬件中的证书

存储在 ET199 中的证书都是使用 EnterSafe 提供的中间件产生或者导入的，ET199 的 CSP 名称为：“EnterSafe ET199 CSP V1.0”。枚举 ET199 硬件中的证书代码如下：

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <wincrypt.h>

#define ET199_CSP_NAME "EnterSafe ET199 CSP v1.0"

int main(int argc, char* argv[])
{
//CSP 句柄
    HCRYPTPROV hTokenProv = NULL;
//密钥句柄
    HCRYPTKEY hKeyCAPI = NULL;

//存放容器名称的数组，容器名最大只能 260 字节
    BYTE pbCertContainerName[9][260] = {0};
    DWORD dwCertNumber = 0;
    DWORD dwCertContainerNameLen = 260;

//证书数据
    DWORD dwCertLen = 0;
    BYTE* pbCert = NULL;
    PCCERT_CONTEXT pCertContext = NULL;

//证书名称
    DWORD dwCertNameLen = 0;
    BYTE* pbCertName = NULL;

//证书序列号
    DWORD dwCertSNLen = 0;
    BYTE* pbCertSN = NULL;

    DWORD i=0, j=0;

//获得 CSP 句柄。第二个参数给 NULL，先枚举容器
    if(!CryptAcquireContext(&hTokenProv, NULL,
        ET199_CSP_NAME, PROV_RSA_FULL, NULL))
```

```
{
    printf("CryptAcquireContext Error:0x%08x\n", GetLastError());
    return -1;
}

//枚举第一个容器，最后一个参数 CRYPT_FIRST
if(CryptGetProvParam(
    hTokenProv,
    PP_ENUMCONTAINERS,
    pbCertContainerName[dwCertNumber],
    &dwCertContainerNameLen,
    CRYPT_FIRST))
{
    //打印枚举到的第一个容器名
    printf("Container %d Name: %s\n",
        dwCertNumber, pbCertContainerName[dwCertNumber]);

    if(dwCertContainerNameLen != 0)
        dwCertNumber ++;

    //枚举其它的容器，最后一个参数为 0
    while(CryptGetProvParam(
        hTokenProv,
        PP_ENUMCONTAINERS,
        pbCertContainerName[dwCertNumber],
        &dwCertContainerNameLen,
        0))
    {
        //打印枚举到的容器
        printf("Container %d Name: %s\n",
            dwCertNumber, pbCertContainerName[dwCertNumber]);

        if(dwCertContainerNameLen != 0)
            dwCertNumber ++;
    }
}
```

```
//打印 ET199 中有几个容器
printf("Certificate Number In ET199: %d\n", dwCertNumber);

//释放 CSP 句柄
CryptReleaseContext(hTokenProv, 0);

if(dwCertNumber == 0)
{
    printf("No Certificate In ET199!\n");
}

//循环获得 ET199 中每个证书的信息
for(i=0; i<dwCertNumber; ++i)
{
    //按容器获得 CSP 句柄。第二个参数给容器名
    if(!CryptAcquireContext(&hTokenProv,
        (const char*)pbCertContainerName[i],
        ET199_CSP_NAME, PROV_RSA_FULL, NULL))
    {
        printf("CryptAcquireContext Error:0x%08x\n", GetLastError());
        return -1;
    }

    //获得加密密钥句柄。第二个参数给 AT_KEYEXCHANGE
    //如果一个容器中既有加密密钥也有签名密钥，请自行处理
    if(!CryptGetUserKey(hTokenProv, AT_KEYEXCHANGE, &hKeyCAPI))
    {
        printf("CryptGetUserKey Error:0x%08x\n", GetLastError());
        return -1;
    }

    //第一次调用获取证书数据长度
    if(!CryptGetKeyParam(hKeyCAPI, KP_CERTIFICATE, NULL, &dwCertLen, 0))
    {
        printf("CryptGetKeyParam 1 Error:0x%08x\n", GetLastError());
        return -1;
    }
}
```

```
//分配空间
pbCert = new BYTE[dwCertLen];
memset(pbCert, 0, dwCertLen);

//第二次调用获取证书数据
if(!CryptGetKeyParam(hKeyCAPI, KP_CERTIFICATE,
                    pbCert, &dwCertLen, 0))
{
    printf("CryptGetKeyParam 2 Error:0x%08x\n", GetLastError());
    return -1;
}

//根据证书数据创建 CERT_CONTEXT 结构
pCertContext = CertCreateCertificateContext(
    PKCS_7_ASN_ENCODING | X509_ASN_ENCODING,
    pbCert,
    dwCertLen);

if(pCertContext == NULL)
{
    printf("CertCreateCertificateContext Error:0x%08x\n",
        GetLastError());
    return -1;
}

//获得证书名称
dwCertNameLen = CertGetNameString(pCertContext,
    CERT_NAME_SIMPLE_DISPLAY_TYPE, 0, NULL, NULL, 0);

pbCertName = new BYTE[dwCertNameLen+1];
memset(pbCertName, 0, dwCertNameLen+1);

CertGetNameString(pCertContext, CERT_NAME_SIMPLE_DISPLAY_TYPE,
    0, NULL, (char *)pbCertName, dwCertNameLen);

printf("\nCert %d Name: %s\n", i, pbCertName);

//获得证书 SN
```

```
dwCertSNLen = pCertContext->pCertInfo->SerialNumber.cbData;
pbCertSN = new BYTE[dwCertSNLen];
memset(pbCertSN, 0, dwCertSNLen);
memcpy(pbCertSN, pCertContext->pCertInfo->SerialNumber.pbData,
        dwCertSNLen);

printf("Cert %d SN:\n", i);
for(j=0; j<dwCertSNLen; ++j)
{
    printf("%02x ", pbCertSN[j]);
}
printf("\n");

//释放 CSP 句柄
CryptReleaseContext(hTokenProv, 0);

//释放内存
if(pbCert != NULL)
{
    delete [] pbCert;
    pbCert = NULL;
}

if(pbCertName != NULL)
{
    delete [] pbCertName;
    pbCertName = NULL;
}

if(pbCertSN != NULL)
{
    delete [] pbCertSN;
    pbCertSN = NULL;
}
}

return 0;
}
```

5.2.2 使用 CAPI 接口验证用户 PIN (User PIN)

在使用 CAPI 的接口进行私钥签名或者私钥解密这些涉及到私钥的操作时，需要先验证用户 PIN (User PIN)，这时 ET199 会弹出一个输入用户 PIN 码的对话框，如下图所示：

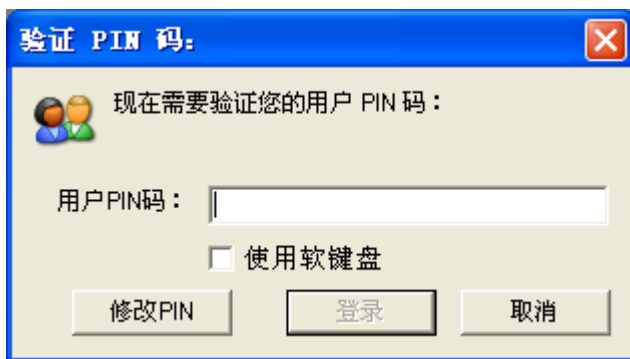


图 22 验证用户 PIN 码对话框

如果开发商不希望弹出对话框，可以使用 CAPI 接口进行用户 PIN 的验证。**用户 PIN 的验证状态是按进程来划分的，当要在同一个进程中恢复为没有验证过的状态，也需要使用下面的方法。**代码如下：

```
HCRYPTPROV hTokenProv = NULL;
BYTE pbUserPIN[] = "12341";

//获得 CSP 句柄
if(!CryptAcquireContext(&hTokenProv, NULL,
    ET199_CSP_NAME, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
{
    printf("CryptAcquireContext Error:0x%08x\n", GetLastError());
    return -1;
}

//设置成没有验证过的状态，第三个参数给 NULL
if(!CryptSetProvParam(hTokenProv, PP_SIGNATURE_PIN, NULL, 0))
{
    printf("CryptSetProvParam Error:0x%08x\n", GetLastError());
    return -1;
}
```

```
}

//验证用户 PIN，第三个参数给用户 PIN
if(!CryptSetProvParam(hTokenProv, PP_SIGNATURE_PIN, pbUserPIN, 0))
{
    printf("CryptSetProvParam Error:0x%08x\n", GetLastError());
    return -1;
}

//释放 CSP 句柄
CryptReleaseContext(hTokenProv, 0);
```

5.2.3 签名和验签

在进行签名和验签时，要保证 ET199 中至少有一对 RSA 密钥对。代码如下：

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <wincrypt.h>

//定义 CSP 名称
#define TEST_CSP_NAME    "EnterSafe ET199 CSP v1.0"
//容器名称
#define CONTAINER_NAME   "CSPKeyTest"

//错误处理函数
void MyHandleError(char *s)
{
    fprintf(stderr, "An error occurred in running the program. \n");
    fprintf(stderr, "%s\n", s);
    fprintf(stderr, "Error number %x.\n", GetLastError());
    fprintf(stderr, "Program terminating. \n");
    exit(1);
}

void main(void)
```



```

{
    HCRYPTPROV hProv;           //CSP 句柄
    BYTE *pbBuffer= (BYTE *)"123456"; //被签名的数据
    DWORD dwBufferLen = strlen((char *)pbBuffer); //被签名的数据长度
    HCRYPTHASH hHash;           //HASH 对象句柄
    HCRYPTKEY hKey;              //密钥句柄
    BYTE *pbKeyBlob;             //公钥数据, BLOB 结构
    BYTE *pbSignature;           //数字签名
    DWORD dwSigLen;              //签名长度
    DWORD dwBlobLen;             //公钥长度, BLOB 结构的长度
    LPTSTR szDescription = "";   //签名时的 Description 参数

    // 获得 CSP 句柄
    printf("The following phase of this program is signature.\n\n");
    if(CryptAcquireContext(
        &hProv,
        NULL,
        TEST_CSP_NAME,
        PROV_RSA_FULL,
        0))
    {
        printf("CSP context acquired.\n");
    }
    else//如获取失败, 则密钥容器不存在, 创建一个新的容器
    {
        if(CryptAcquireContext(
            &hProv,
            NULL,
            TEST_CSP_NAME,
            PROV_RSA_FULL,
            CRYPT_NEWKEYSET)) //创建新的密钥容器
        {
            printf("A new key container has been created.\n");
        }
        else
        {
            MyHandleError("Error during CryptAcquireContext.");
        }
    }
}

```

```
}

// 从密钥容器中取数字签名用的密钥
if(CryptGetUserKey(
    hProv,
    AT_SIGNATURE,    //这里也可以使用 AT_KEYEXCHANGE
    &hKey))
{
    printf("The signature key has been acquired. \n");
}
else//如果取密钥失败，则创建新的密钥，
{
    if(GetLastError() == NTE_BAD_KEYSET)
    {
        if(CryptGenKey(
            hProv,          //CSP 句柄
            AT_SIGNATURE,   //创建的密钥对类型为 AT_SIGNATURE
            0,              //key 类型，这里用默认值
            &hKey))         //创建成功返回新创建的密钥对的句柄
        {
            printf("Created a signature key pair.\n");
        }
        else
        {
            MyHandleError("Error occurred creating a
                           signature key.\n");
        }
    }
    else
    {
        MyHandleError("Error during CryptGetUserKey for signkey.");
    }
}
```

//这里导出公钥，在验证签名时使用。程序的后半部为验签的过程
//应用时需要将原文，签名结果和公钥发给对方进行验签

//**ET199 的 CSP 只能导出公钥，不能导出私钥**

```
if(CryptExportKey(
    hKey,          //密钥句柄
    NULL,
    PUBLICKEYBLOB, //导出公钥
    0,
    NULL,
    &dwBlobLen))   //得到公钥的大小
{
    printf("Size of the BLOB for the public key determined. \n");
}
else
{
    MyHandleError("Error computing BLOB length.");
}

// 为存储公钥的缓冲区分配内存。
if(pbKeyBlob = (BYTE*)malloc(dwBlobLen))
{
    printf("Memory has been allocated for the BLOB. \n");
}
else
{
    MyHandleError("Out of memory. \n");
}

// 真正导出公钥数据
if(CryptExportKey(
    hKey,
    NULL,
    PUBLICKEYBLOB,
    0,
    pbKeyBlob,     //公钥数据，为 BLOB 结构
    &dwBlobLen))
{
    printf("Contents have been written to the BLOB. \n");
}
else
{

```

```
        MyHandleError("Error during CryptExportKey.");
    }

    //在签名前先要进行 HASH 散列操作。创建 HASH 对象
    if(CryptCreateHash(
        hProv,          //CSP 句柄
        CALG_SHA1,      //使用 SHA1 散列算法，也可以使用 CALG_MD5,
        0,
        0,
        &hHash))        //HASH 对象句柄
    {
        printf("Hash object created. \n");
    }
    else
    {
        MyHandleError("Error during CryptCreateHash.");
    }

    //对被签名数据进行 HASH 运算
    if(CryptHashData(
        hHash,          //HASH 对象句柄
        pbBuffer,       //被签名数据 (“123456”)
        dwBufferLen,    //被签名数据长度
        0))
    {
        printf("The data buffer has been hashed. \n");
    }
    else
    {
        MyHandleError("Error during CryptHashData.");
    }

    //使用私钥对经过 HASH 后的数据进行签名
    //这时会弹出输入用户 PIN (User PIN) 的对话框
    dwSigLen= 0;
    if(CryptSignHash(
        hHash,          //HASH 对象句柄
        AT_SIGNATURE,   //密钥类型，可以是 AT_KEYEXCHANGE
```

```
        szDescription,          //这个参数没用
        0,
        NULL,
        &dwSigLen))              //得到数字签名大小
    {
        printf("Signature length %d found.\n", dwSigLen);
    }
    else
    {
        MyHandleError("Error during CryptSignHash.");
    }

    //为数字签名缓冲区分配内存
    if(pbSignature = (BYTE *)malloc(dwSigLen))
    {
        printf("Memory allocated for the signature.\n");
    }
    else
    {
        MyHandleError("Out of memory.");
    }

    //得到数字签名
    if(CryptSignHash(
        hHash,                    //HASH 对象句柄, 里面包含被签名数据经过 HASH 后的数据
        AT_SIGNATURE,            //密钥类型, 可以是 AT_KEYEXCHANGE
        szDescription,
        0,
        pbSignature,              //这里将返回数字签名
        &dwSigLen))
    {
        printf("pbSignature is the hash signature.\n");
    }
    else
    {
        MyHandleError("Error during CryptSignHash.");
    }
}
```

```

//打印签名结果
printf("\n");
for(DWORD i=0; i<dwSigLen; ++i)
{
    if(i%16 == 0)
        printf("\n");
    printf("%02x ",pbSignature[i]);
}
printf("\n");

//释放 HASH 对象句柄
if(hHash)
    CryptDestroyHash(hHash);

printf("The hash object has been destroyed.\n");
printf("The signing phase of this program is completed.\n\n");

/*****

```

上面的代码为签名过程。 下面的代码是接收者使用的,这里为了说明方便就把它放在一个文件里了。pbBuffer, pbSignature, szDescription, pbKeyBlob, 还有他们的长度在这里直接使用了, 应该是接收者从文件或其他地方读取的。pbBuffer 里保存的是被签名的数据, 所以认证时的内容必须跟签名时的一样 这里使用的 CSP 句柄也没有重新创建

```

*****/

printf("The following phase of this program is verify signature.\n\n");

//使用公钥进行验签。先把公钥数据导入生成公钥句柄
HCRYPTKEY hPubKey;
if(CryptImportKey(
    hProv,          //CSP 句柄
    pbKeyBlob,      //公钥数据, BLOB 结构
    dwBlobLen,      //公钥长度
    0,
    0,
    &hPubKey))      //公钥句柄

```

```
{
    printf("The key has been imported.\n");
}
else
{
    MyHandleError("Public key import failed.");
}

//先对原文做散列运算，验签时使用。创建哈希对象
if(CryptCreateHash(
    hProv,          //CSP 句柄
    CALG_SHA1,     //这里使用 SHA1 散列算法，与签名时的一致
    0,
    0,
    &hHash))       //HASH 对象句柄
{
    printf("The hash object has been recreated. \n");
}
else
{
    MyHandleError("Error during CryptCreateHash.");
}

//将原文做 HASH 计算
if(CryptHashData(
    hHash,          //HASH 对象句柄
    pbBuffer,       //原文（“123456”）
    dwBufferLen,    //原文长度
    0))
{
    printf("The new hash has been created.\n");
}
else
{
    MyHandleError("Error during CryptHashData.");
}
```

//验证数字签名，即原文 HASH 后的数据，签名后的数据和公钥进行验证操作

```
    if(CryptVerifySignature(
        hHash,           //HASH 对象句柄, 里面包含原文 HASH 后的数据
        pbSignature,     //签名后的数据
        dwSigLen,        //签名后数据的程度
        hPubKey,         //公钥句柄
        szDescription,   //没有用
        0))
    {
        printf("The signature has been verified.\n");
    }
    else
    {
        printf("Signature not validated!\n");
    }

    //释放内存
    if(pbSignature)
        free(pbSignature);

    if(pbKeyBlob)
        free(pbKeyBlob);

    //释放 HASH 对象句柄
    if(hHash)
        CryptDestroyHash(hHash);

    //释放 CSP 句柄
    if(hProv)
        CryptReleaseContext(hProv, 0);

    printf("\n\nPress any key to exit...\n");
    getch();
}
```

5.2.4 RSA 加解密


```
HCRYPTPROV hTokenProv = NULL;           //CSP 句柄
HCRYPTKEY hKeyCAPI = NULL;              //密钥句柄

BYTE pbData[] = "Hello World!";         //待加密数据
DWORD dwDataLen = sizeof(pbData);       //待加密数据长度（为 13）

BYTE* pbEncryptedData = NULL;            //加密后数据
DWORD dwEncryptedLen = dwDataLen;
DWORD dwTemp = 0;

//获得 CSP 句柄
if(!CryptAcquireContext(&hTokenProv, NULL,
    ET199_CSP_NAME, PROV_RSA_FULL, NULL))
{
    printf("CryptAcquireContext Error:0x%08x\n", GetLastError());
    return -1;
}

//获得加密密钥句柄
if(!CryptGetUserKey(hTokenProv, AT_KEYEXCHANGE, &hKeyCAPI))
{
    printf("CryptGetUserKey Error:0x%08x\n", GetLastError());
    return -1;
}

//第一次调用，获得加密后的数据长度
if(!CryptEncrypt(
    hKeyCAPI,           //密钥句柄
    0,
    TRUE,               //当待加密数据只有一块或者是最后一块时设为 TRUE
    0,
    NULL,               //输入为待加密数据，返回为密文。第一次调用给 NULL
    &dwEncryptedLen,     //输入为待加密数据的长度，返回为加密后数据的长度
    dwDataLen))         //缓冲区大小
{
    printf("CryptEncrypt 1 Error:0x%08x\n", GetLastError());
    return -1;
}
```

```
//分配空间
pbEncryptedData = new BYTE[dwEncryptedLen];
memset(pbEncryptedData, 0, dwEncryptedLen);

//将待加密的数据拷贝到分配的空间中
memcpy(pbEncryptedData, pbData, dwDataLen);

//设置第六个参数，待加密的数据长度（为 13）
//设置第七个参数，缓冲区的大小（为 128）
dwTemp = dwEncryptedLen;
dwEncryptedLen = dwDataLen;
dwDataLen = dwTemp;

//第二次调用，得到加密后的密文
if(!CryptEncrypt(hKeyCAPI, 0, TRUE, 0,
                pbEncryptedData, &dwEncryptedLen, dwDataLen))
{
    printf("CryptEncrypt 2 Error:0x%08x\n", GetLastError());
    return -1;
}

//打印加密结果
printf("Cipher Text:\n");
for(i=0; i<dwEncryptedLen; ++i)
{
    printf("%02X ", pbEncryptedData[i]);

    if((i+1) % 16 == 0)
        printf("\n");
}

//解密，第六个参数一定要是 128 的倍数（RSA 为 1024 位时）
if(!CryptDecrypt(
    hKeyCAPI,          //密钥句柄
    0,
    TRUE,              //当待加密数据只有一块或者是最后一块时设为 TRUE
    0,
```

```
        pbEncryptedData, //输入为加密后的密文，输出为解密后的原文
        &dwEncryptedLen)) //输入为加密后密文长度，输出为解密后原文长度
    {
        printf("CryptDecrypt Error:0x%08x\n", GetLastError());
        return -1;
    }

//打印原文
printf("\nPlain Text:\n%s\n", pbEncryptedData);

//释放 CSP 句柄
CryptReleaseContext(hTokenProv, 0);

//释放内存
if(pbEncryptedData != NULL)
{
    delete [] pbEncryptedData;
    pbEncryptedData = NULL;
}
```

附录 A

支持的操作系统	Windows 98SE/ME/2000/XP/2003/Vista/2008/Windows7/Windows8
证书和标准	PKCS # 11 v2.11, MS CAPI, PC/SC, X.509 v3 证书存储, SSL v3, IPsec, 兼容 ISO 7816
处理器	16 位
存储空间	64K
内置安全算法	RSA, DES, 3DES, MD5, SHA-1
芯片安全水平	安全加密的数据存储
功率	< 250 mW
工作温度	0 ~ 70° C
存放温度	- 40 ~ 85° C
湿度	0 ~ 100% 不结露
接口类型	A 类 USB
外壳	一次性, 防水, 硬塑料外壳
数据存储年限	室温下数据保持时间最少 10 年
写次数	最少擦写次数 10 万次