```
DWORD                cbData;
// 第一步，使用缺省 CSP，打开“MY”密钥库
hCertStore = CertOpenSystemStore(NULL, _T(“MY”));
if (hCertStore == NULL) {
    // 提示错误信息并退出
    TCHAR *errMsg = _T("不能打开证书库: MY");
    MessageBox(buf, "错误信息", MB_OK|MB_ICONERROR);
    return FALSE;
}
// 第二步，使用 CertEnumCertificatesInStore 从打开的证书库中获得证书
//获取库中第一个证书时，需要把 pCertContext 置为 NULL
pCertContext = NULL;
while(pCertContext= CertEnumCertificatesInStore(hCertStore, pCertContext)) {
    // 第三步，获取证书通用名
    cbData = CertGetNameString(pCertContext,
        CERT_NAME_SIMPLE_DISPLAY_TYPE, 0, NULL, pszNameString, 256);
    if (cbData > 1 && cbData < 256) {
        printf("\n 发现证书：%s \n",pszNameString);
    }
}
// 第四步，关闭证书库句柄
CertCloseStore(hCertStore, 0);
return TRUE;
```

2. 复制证书句柄

```
PCCERT_CONTEXT   pDupCertContext = NULL;
//  复制证书对象指针
if (pDupCertContext = CertDuplicateCertificateContext(pCertContext)) {
        printf("成功复制证书对象指针\n");
        // 使用复制的指针...
        CertFreeCertificateContext(pDupCertContext); // 释放复制的指针
}
else {
        printf("复制指针错误\n");
        exit(1);
}
```

12.1.3 使用私钥

12.1.3.1 函数说明

1. 通过证书获得私钥

```
BOOL WINAPI CryptAcquireCertificatePrivateKey(
        PCCERT_CONTEXT pCertContext,
        DWORD          dwFlags,
        void*          pvReserved,
```

```
    HCRYPTPROV_OR_NCRYPT_KEY_HANDLE* phCryptProv,
    DWORD*          pdwKeySpec,
    BOOL*           pfCallerFreeProvOrN);
```

参数：

pCertContext：包含 CERT_CONTEXT 证书内容的指针。

dwFlags：

CRYPT_ACQUIRE_CACHE_FLAG：如果一个句柄已经缓存，则返回相同的句柄；否则，获取一个新的句柄和缓存使用证书的 CERT_KEY_CONTEXT_PROP_ID 属性。若设置此标志，pfCallerFreeProvOrNCryptKey 变量接收 FALSE 时调用应用程序不能释放句柄。

CRYPT_ACQUIRE_COMPARE_KEY_FLAG：比较证书中的公钥与 CSP 返回的公钥，如果密钥不匹配，获取失败，错误代码设置为 NTE_BAD_PUBLIC_KEY。如果返回缓存句柄，则不比较公钥。

CRYPT_ACQUIRE_NO_HEALING：此功能将不会尝试重新创建证书上下文中的 CERT_KEY_PROV_INFO_PROP_ID 属性，即使此属性不能被检索到。

CRYPT_ACQUIRE_SILENT_FLAG：此环境中 CSP 显示任何用户界面。如果 CSP 必须显示 UI 以进行操作，导致调用失败，错误代码设置为 NTE_SILENT_CONTEXT。

CRYPT_ACQUIRE_USE_PROV_INFO_FLAG：使用该证书的 CERT_KEY_PROV_INFO_PROP_ID 属性来判断是否应该缓存。如果以前的调用中，CRYPT_KEY_PROV_INFO 结构的 dwFlags 成员包含了 CERT_SET_KEY_CONTEXT_PROP，此函数只使用缓存。

pvReserved：保留，为 NULL。

phCryptProv：输出值，CSP 服务提供者句柄。

pdwKeySpec：输出值，密钥类型，签名密钥或加密密钥。

pfCallerFreeProv：输出值，表明是否释放输出提供者句柄 hCryptProv。

说明：对于指定证书上下文得到一个提供者句柄和私钥类型。

2．创建哈希对象

```
BOOL WINAPI CryptCreateHash(
        HCRYPTPROV hProv,
        ALG_ID Algid,
        HCRYPTKEY hKey,
        DWORD dwFlags,
        HCRYPTHASH* phHash);
```

参数：

hProv：指定容器的 CSP 模块句柄。

Algid：哈希算法的标识符，支持 CALG_MD5、CALG_SHA1、CALG_SHA_256、CALG_SHA_384、CALG_SHA_512。

hKey：如果哈希算法是密钥哈希，如 HMAC 或 MAC 算法，就用此密钥句柄传递密钥。对于非密钥算法，此参数为 NULL。

dwFlags：保留。必须为 0。

phHash：输出参数，哈希对象的句柄。

说明：此函数初始化哈希数据流。它创建并返回一个 CSP 哈希对象的句柄，此句柄由 CryptHashData 来调用。

3．计算数据哈希值

```
BOOL WINAPI CryptHashData(
        HCRYPTHASH hHash,
        BYTE* pbData,
        DWORD dwDataLen,
        DWORD dwFlags);
```

参数：

hHash：哈希对象句柄。

pbData：指向要计算哈希值的数据指针。

dwDataLen：数据长度。

dwFlags：对 CRYPT_USERDATA 标志，所有微软 CSP 都忽略此参数。如果所有其他 CSP 不忽略此参数，且置此参数，CSP 提示用户直接输入数据。

说明：此函数把一段数据加入到指定的哈希对象中。

4．销毁哈希对象句柄

```
BOOL WINAPI CryptDestroyHash(
        HCRYPTHASH hHash);
```

参数：

hHash：哈希对象句柄。

说明：此函数销毁由参数指定的哈希对象。当一个哈希对象被销毁后，它对程序来说将不可用。

5．计算数据签名值

```
BOOL WINAPI CryptSignHash(
        HCRYPTHASH hHash,
        DWORD dwKeySpec,
        LPCTSTR sDescription,
        DWORD dwFlags,
        BYTE* pbSignature,
        DWORD* pdwSigLen);
```

参数：

hHash：哈希对象句柄。

dwKeySpec：CSP 容器使用的密钥类型，签名时使用 AT_SIGNATURE。

sDescription：不再使用，必须置为 NULL。

dwFlags：一般为 0，若与 RSA 密钥提供者一起使用，可设置为 CRYPT_NOHASHOID 时，表示 HASH 的 OID 不加入到公钥加密中。在签名时，OID 总会加到签名值中。

pbSignature：输出值，存放签名后的结果字节串。

pdwSigLen：输出值，存放签名值的长度。

说明：计算数据签名值。

12.1.3.2　示例程序

私钥存放在 CSP 的容器中，在使用私钥前，需要先确定私钥的存放位置，然后打开 CSP 句柄。在获得私钥句柄后，就可以进行签名操作。

确定私钥的位置有两种方式，一是通过查找到的证书定位私钥，二是通过密钥容器定位私钥。应用系统一般使用证书定位私钥方式，CA 系统一般通过容器方式定位私钥。

1．通过证书句柄获得私钥句柄

```
HCRYPTPROV hCryptProv;
HCRYPTKEY   hPrivateKey;
BOOL          bFreeProv;
DWORD         dwKeySpec;
PCCERT_CONTEXT pCertContext = ...; // 已完成赋值，如通过参数方式获得
// 第一步，通过证书句柄获得 CSP 提供者信息
BOOL rc = CryptAcquireCertificatePrivateKey(
    pCertContext,   CRYPT_ACQUIRE_USE_PROV_INFO_FLAG,
    NULL, // reserved
    &hCryptProv,   &dwKeySpec,   &bFreeProv);
if (!rc) {
    // 处理错误，handleError("CryptAcquireCertificatePrivateKey");
    return rc;
}
```

2．私钥签名

```
BYTE   *data=..., sig[256]; //设置初始数据
DWORD   dlen, buflen, dwKeySpec = AT_SIGNATURE;
HCRYPTPROV *hCryptProv
HCRYPTHASH hHash = 0;
BOOL rc = FALSE;
DWORD siglen = buflen;
// 第一步，使用 MD5 算法创建哈希对象
rc = CryptCreateHash(hCryptProv,CALG_MD5 , 0, 0, &hHash);
if (!rc)   return 0;
// 第二步，计算签名原文数据 HASH 值
rc = CryptHashData(hHash, data, dlen, 0);
if(!rc) {
    CryptDestroyHash(hHash); // 如果出现错误，释放哈希对象句柄
    return 0;
}
// 因为是签名，使用私钥类型 AT_SIGNATURE;在 CryptSignHash()中，如果没有置 CRYPT_
// NOHASHOID 标记，自动在 HASH 结果前加 OID 后再使用私钥加密
// 第三步，执行签名操作
rc = CryptSignHash(hHash, dwKeySpec, NULL, 0, sig, &siglen);
```

```
// 释放哈希对象句柄
CryptDestroyHash(hHash);
if (!rc) return 0;
// 第四步，需要对签名结果字节流进行反序，以便与其他密码库签名结果互通
// reverseBuffer(sig, siglen);
return siglen;
```

12.2 PKCS#11

12.2.1 PKCS#11 简介

PKCS#11 是 RSA 公司定义的安全编程接口，其目的是屏蔽密码设备的差异性，向应用层提供一套统一的编程接口。现有很多密码设备提供了 PKCS#11 编程接口，包括密码服务器、智能密码密钥、密码卡、IC 卡等硬件设备。当然也可以使用软件实现，如火狐浏览器（firefox）内嵌了 PKCS#11 安全模块，用于 SSL 通道建立和密码运算。

12.2.1.1 Cryptoki

PKCS#11 编程接口称作 Cryptoki，是 cryptographic token interface（密码令牌接口）的缩写。它遵循一种基于对象的简单方法，提出技术独立（支持各种各样的设备）和资源共享（多个应用程序可访问多个设备）的目标，应用程序以逻辑视图方式把所有密码设备当作一种密码令牌。应用层不关心密码令牌是如何实现的，它只要调用标准的接口，即可完成密码运算。

Cryptoki 的通用模型如图 12-2 所示。

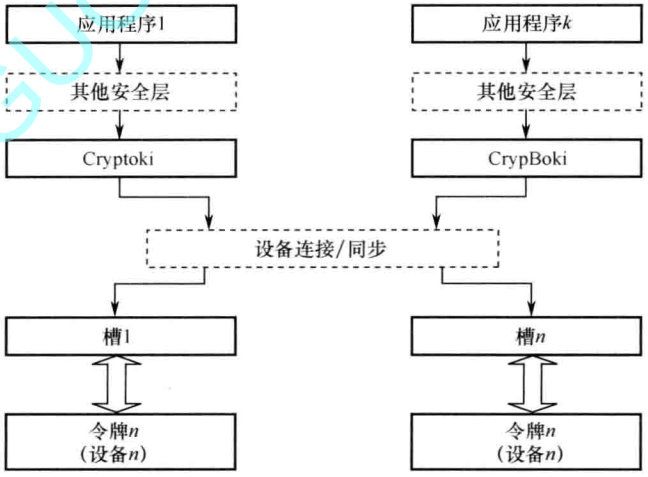

图 12-2　通用 Cryptoki 模型

在系统中，密码设备通过槽（slot）进行运行，这些槽可能是一个物理读卡器，Cryptoki 为这些设备提供操作接口，若一台密码设备存在于读卡器中，一个令牌就存在于该槽中。由于 Cryptoki 提供槽和令牌的逻辑视图，所以槽和令牌可能有其他的物理交互。多个槽可能共享一个读卡器，一个系统可以有多个槽，应用程序能连接到这些槽的任何一个或全部