Visual J++ 6.0 中调用本地 Win32 API 技术

杨少波　（中国科学院计算技术研究所　100080）

摘要：本文主要介绍在 Visual J++ 6.0 开发平台中，利用 Java 语言和 WFC 类库开发 Windows 应用程序及 Web 应用时，内嵌调用本地 Windows 系统的 Win32 API 函数的技术机制、编程要点并给出应用示例。

关键词：Java 语言　J/Direct Call Builder　Win32 API 函数

一、引言

Microsoft 公司新近推出的 Visual J++ 6.0 Java 语言程序集成开发平台，为 Windows 应用程序的开发开辟了另一新途径。程序员借助于 Java 语言并通过双向快速应用程序开发 RAD 工具快速地构造出 Windows 应用程序及组件。这是目前较优的基于数据库驱动、客户服务器模式的高性能 Windows GUI 应用程序和 Web 解决方案的开发工具。

其 J/Direct Call Builder 本地代码向导工具增强了 Java 语言对 Windows 系统本地 Win32API 函数的支持，使访问本地 API 函数变得容易和直接；它是从 Java 语言代码中直接调用 Windows 系统 Win32 API 函数的一种工具，利用它可以访问所有的 Win32 系统 API 函数。其实现机制是：一个本地 Win32 API 函数可以在同一个源程序中被声明和调用，所需的数据类型转换都由 Java 虚拟机自动完成。

二、利用向导工具内嵌 Win32 API 函数

利用此向导工具可以指导程序员如何实现在 Java 程序代码中调用 Win32 API 函数。操作过程如下：

(1)在 Visual J++ 6.0 集成 IDE 开发平台下的 View 菜单中，选择 Other Windows 子菜单，然后选择 J/Direct Call Builder 子菜单；(2)此时显示出在 Advapi32.dll、Gdi32.dll、Kernel32.dll、Shell.dll、Spoolss.dll、User32.dll、Winmm.dll 等动态链接库文件中定义出的各个 Win32 API 函数；(3)在 Target 栏中选择要包括对 API 调用的类(缺省未给定时的类名为 Win32 类名)，并选择所需要调用的 API 函数原型定义；(4)选择 Copy to Target 按钮(或双击所选择的内容)，J/Direct Call Builder 向导工具自动插入此选中的 API 函数所需的函数原型声明信息。

```
class UseWinAPI
{ public static void main(String args[])
    { MessageBox(null,"Win32 API Called!",
        "Win32 API in Java",0);
    }
/** @dll.import("USER32") */ //API 函数原型声明信息
private static native int MessageBox(int hwndOwner,
    String text,String title,int fuStyle);
}
```

这样可在同一程序的各个类成员函数中使用此 Win32 API 函数。但由于 Java 语言的一些特殊性及 Windows API 函数的复杂性，在调用中还应针对不同应用情况加以相应特殊处理。下文试图介绍这些方面的编程要点并给出应用示例。

三、涉及指针及结构体数据类型的处理

由于 Java 语言不提供对指针及结构体数据类型的支持，但在许多 Win32 API 函数中涉及到大量的指针或结构体数据类型的形参。因此在使用这些 Win32 API 函数时便需要作一些技术处理后方可实现正确调用。

1.本地 API 函数调用中的指针类型的形参

实现的主要措施是在 Java 程序中定义出只包含一个元素的数组来代替指针参数，如 Win32 API 中查询获取磁盘可用容量的函数原型定义如下：

```
BOOL GetDiskFreeSpace(LPCTSTR szRootPathName,
DWORD *lpSectorsPerCluster,
    DWORD *lpBytesPerCluster, DWORD *
lpFreeClusters, DWORD *lpClusters);
```

其中有许多指针类型参数定义，调用时，应在 Java 程序中相应地定义出只有一个元素的数组，然后将数组

© 中国科学院软件研究所 http://www.c-s-a.org.cn

名作为实参传给该 Win32 API 函数的指针形参,实现函数调用。

```
class UseWinAPI
{ int sectorsPerCluster[ ] = {0}; int bytesPerCluster[ ] = {0};
    int freeCluster[ ] = {0}; int clusters[ ] = {0};
    public static void main(String args[ ])
    { GetDiskFreeSpace ( " C: \ ", sectorsPerCluster, bytesPerCluster,
      freeCluster, clusters);
    System. out. println( " Sector/Cluster = " + sectorsPerCluster[0]);
    System. out. println ( " bytes/Cluster = " + bytesPerCluster[0]);
//.....其他代码;
    }
/ * * @dll. import("KERNEL32") * /
private static native boolean GetDiskFreeSpace(String rootName,
    int PSectorsPerCluster[ ], int PBytesPerCluster[ ],
    int PFreeCluster[ ], int PClusters[ ]);
}
```

2. 本地 API 函数调用中有关结构体类型的形参

对此类 Win32 API 函数,可采用 @dll. struct()编译指示(Compile Directive)指明一个对应结构体各成员的 Java 类,该类中只包含成员数据,其数据类型与原结构体成员相对应(对某些数据类型的结构体成员还应进行转换),如 Win32 API 中的 SYSTEMTIME 结构体的使用。

原 Win32 API 中的 SYSTEMTIME 定义:

```
typedef struct { WORD wYear; WORD wMonth; WORD wDayofWeek; WORD wDay;
    WORD wHour; WORD wMinute; WORD wSecond; WORD wMilliSeconds;
    } SYSTEMTIME;
```

在 Visual J + + 6.0 中使用此结构体的方法如下:

(1)定义出一个 Java 类

```
/ * * @dll. struct() * /
class SYSTEMTIME
{ public short wYear; public short wMonth; Public short wDayofWeek;
    public short wDay; public short wHour; public short wMinute;
    public short wSecond; public short wMilliSeconds;
}
```

(2)使用此结构体

```
class UseWinAPI
    { public static void main(String args[ ])
    { SYSTEMTIME systemTime = new SYSTEMTIME();
    GetSystemTime(systemTime);
    System. out. println("Year is" + systemTime. wYear);
    System. out. println ( " Month is" + systemTime. wMonth);
//.....其他代码;
    }
/ * * @dll. import("KERNEL32") * /
private static native void GetSystemTime(SYSTEMTIME pst);
}
```

3. 本地 API 函数调用中有关内嵌型结构体

对每一个结构体类分别采用 @dll. struct()编译指示指明,并分别给出对应的类定义,在父结构体中应创建出其内嵌的子结构体对象实例,才能使用内嵌子结构体成员数据,否则会出现空对象错误。如 Win32 API 中的 MSG 结构体的使用。

原 Win32 API 中的定义:

```
typedef struct{ LONG x; LONG y; } Point;
typedef struct { int hwnd; int message; int wParam; int lParam;
    int time; Point pt; //内嵌有另一个结构体
    }MSG;
```

在 Visual J + + 6.0 中使用此结构体的方法如下:

(1)定义出一个 Java 类

```
/ * * @dll. struct() * /
class Point {}
    public int x; public int y;
    }
/ * * @dll. struct() * /
class MSG {
    public int hWnd; public int message; public int wParam; public int time;
    public int lParam; public POint pt = new Point();
    }
```

(2)使用此结构体

```
class UseWinAPI
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
{ public static void main(String args[])
    { MSG myMsg = new MSG();
    myMsg.time = 0; //使用此父结构体成员数据
    myMsg.pt.x = 0; //使用此内嵌型结构体成员数据
    myMsg.pt.y = 0;
    }
}
```

4. 结构体中包含有固定长度的数组数据成员

采用在 @dll.structmap([Type = FIXEDARRAY, Size = 4])编译指示中指明为固定长度的数组成员，同样也应给出结构体形参对应的类成员定义。

结构体的定义：struct EmbeddedArrays

```
{ BYTE b[4]; CHAR c[5];
};
```

在 Visual J++ 6.0 中使用此种结构体的方法如下：

(1)定义出一个 Java 类

```
/** @dll.struct() */
class EmbeddedArrays
{ /** @dll.structmap([type = FIXEDARRAY, size = 4]) */
    public byte b[];
    /** @dll.structmap([type = FIXEDARRAY, size = 5]) */
    public char c[];
}
```

(2)使用此结构体

```
class UseWinAPI
{ public static void main(String args[])
    { EmbeddedArrays arrayObj = new EmbeddedArrays();
    arrayObj.b[0] = 1;
    arrayObj.c[4] = 'A';
    }
}
```

四、涉及回调函数及动态加载 DLL 库文件

1. 回调函数的处理技术

在 Windows 系统中有许多涉及到回调函数的 Win32 API 函数，但目前 Java 语言及常规的开发系统中并未直接提供对回调函数的支持。如何使用这些 Win32 API 函数来增强应用程序的性能？这在 Visual J++ 6.0 的 Java 应用程序中借助于 J/Direct Call Builder 也能够实现。

首先在程序开始处引用 com.ms.dll.Callback 类，并声明所要采用的涉及回调函数的 Win32 API 函数原型(本例选用 EnumWindows())。

```
import com.ms.dll.Callback;
/** @dll.import("USER32") */
static native boolean EnumWindows (Callback wndenumProc, int lParam);
```

其次定义一个 Callback 类的派生类，并在其中定义出一个非静态的名为 callback()成员函数，在函数体中加入程序员的回调函数功能操作代码，因为此函数即为用户的回调函数，完成用户所希望的功能。

```
class EnumWindowsProc extends Callback
{ public boolean callback(int hwnd, int lParam)
    { //...回调函数功能操作代码
    return true;
    }
//.....其他成员函数代码;
}
```

最后在自己类的成员函数代码中，使用该 Win32 API 函数，调用时传给前面定义出的 Callback 类的派生类的对象实例作为回调函数的参数。

```
class UseWinAPI
{ public static void main(String args[])
    { boolean result = EnumWindows (new EnumWindowsProc(),0);
//.....其他代码;
    }
}
```

2. 动态加载 DLL 库文件

在 Visual C++ 中，程序员可利用 LoadLibrary()加载动态链接库文件；然后用 GetProcAddress()获取所要调用的 Win32 API 函数指针，利用此函数指针调用该 DLL 文件中的输出函数；最后采用 FreeLibrary()释放此 DLL 程序。而这在 Visual J++ 6.0 集成 IDE 开发平台中则要采用如下方法来实现。

首先在 Java 代码中声明所要使用的涉及 DLL 程序操作的本地 Win32 API 函数。

```
/** @dll.import("KERNEL32", auto) */
public native static int LoadLibrary(String lpLibFileName);
/** @dll.import("KERNEL32", auto) */
public native static boolean FreeLibrary(int hLibModule);
/** @dll.import("KERNEL32", ansi) */
public native static int GetProcAddress(int hModule, String
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

lpProcName);

其次利用 Msjava.dll 再输出声明一个名为 call 的函数,采用与本地 Win32 API 函数相同的方式声明它。

```
/ * * @dll.import("msjava") * /
public native static boolean call(int funcptr, String argument);
```

最后调用 DLL 程序中的指定的输出函数(本例为"boolean myFunction(LPCSTR text)"),然后采用 FreeLibrary()释放此 DLL 程序。

```
class UseWinAPI
{ public static void main(String args[])
    { int hModule = LoadLibrary("DLL 文件名");
    int funcAddr = GetProcAddress(hModule," myFunction");
    boolead result = call(funcAddr,"Hello, Friends!");
//"Hello, Friends!"为传给 DLL 中的输出函数 myFunction()的实参
    FreeLibrary(hModule);
//.....其他代码;
    }
}
```

参考文献

[1] 杨少波,Java Applet 调用 ActiveX 控件的技术,计算机系统应用,1998 年第 5 期

[2] 杨少波,Java 程序设计中的数据交换和通信技术,计算机系统应用,1998 年第 12 期

(来稿时间:1998 年 12 月)

© 中国科学院软件研究所 http://www.c-s-a.org.cn