

深入浅出 JNA—快速调用原生函数

By 沈东良（网名：良少）

Blog: <http://blog.csdn.net/shendl>

2009/7/20

本文原名《使用 JNA 方便地调用原生函数》发表于 2009 年 3 月的“程序员”杂志上。感谢程序员杂志的许可，使这篇文章能够成为免费的电子版，发布于网络上。

程序员杂志发表此文时，略有裁剪，因此本文比程序员上的文章内容更多。

JNA 的 API 参考手册和最新版本的 pdf 文档，可以在如下地址下载：

<http://code.google.com/p/shendl/downloads/list>

目录

深入浅出 JNA—快速调用原生函数.....	1
为什么需要 JNA.....	2
JNA 介绍.....	2
JNA 实现原理.....	2
JNA 调用原生函数.....	3
例子 1 使用 JNA 调用原生函数.....	3
使用 JNA 调用原生函数的模式.....	3
Java 和原生代码的类型映射.....	4
Java—C 和操作系统数据类型的对应表.....	4
JNA 支持常见的数据类型的映射.....	4
跨平台、跨语言调用原则：.....	5
JNA 模拟结构体.....	5
例 2 使用 JNA 调用使用 Struct 的 C 函数.....	5
Structure 说明.....	6
JNA 模拟复杂结构体.....	7
例 3 结构体内部可以包含结构体对象的数组.....	7
例 4 结构体内部可以包含结构体对象的指针的数组.....	7
原生代码调用 Java 代码.....	8
例 5 通过回调函数实现原生代码调用 Java 代码.....	8
JNA 回调函数说明.....	9
JNA 模拟指针.....	9
例 6 使用 PointerByReference 模拟指向指针的指针.....	11
例 7 使用 Pointer 和 PointerByReference 模拟指针.....	11
Pointer 类详解.....	12
结语.....	13

为什么需要 JNA

和许多解释执行的语言一样，Java 提供了调用原生函数的机制，以加强 Java 平台的能力。Java™ Native Interface (JNI)就是 Java 调用原生函数的机制。

事实上，很多 Java 核心代码内部就是使用 JNI 实现的。这些 Java 功能实际上是通过原生函数提供的。

但是，使用 JNI 对 Java 开发者来说简直是一场噩梦。

如果你已经有了原生函数，使用 JNI，你必须使用 C 语言再编写一个动态链接库，这个动态链接库的唯一功能就是使用 Java 能够理解的 C 代码来调用目标原生函数。

这个没什么实际用途的动态链接库的编写过程令人沮丧。同时编写 Java 和 C 代码使开发难度大大增加。

因此，在 Java 开发社区中，人们一直都视 JNI 为禁地，轻易不愿涉足。

缺少原生函数的协助使 Java 的使用范围大大缩小。

反观.NET 阵营，其 P/Invoke 技术调用原生函数非常方便，不需要编写一行 C 代码，只需要写 Annotation 就可以快速调用原生函数。因此，与硬件有关的很多开发领域都被 .NET 所占据。

JNA 介绍

JNA(Java Native Access)框架是一个开源的 Java 框架，是 SUN 公司主导开发的，建立在经典的 JNI 的基础之上的一个框架。

JNA 项目地址：<https://jna.dev.java.net/>

JNA 使 Java 调用原生函数就像.NET 上的 P/Invoke 一样方便、快捷。

JNA 的功能和 P/Invoke 类似，但编写方法与 P/Invoke 截然不同。JNA 没有使用 Annotation，而是通过编写一般的 Java 代码来实现。

P/Invoke 是.NET 平台的机制。而 JNA 是 Java 平台上的一个开源类库，和其他类库没有什么区别。只需要在 classpath 下加入 jna.jar 包，就可以使用 JNA。

JNA 使 Java 平台可以方便地调用原生函数，这大大扩展了 Java 平台的整合能力。

JNA 实现原理

JNI 是 Java 调用原生函数唯一的机制。JNA 也是建立在 JNI 技术之上的。它简化了 Java

调用原生函数的过程。

JNA 提供了一个动态的 C 语言编写的转发器，可以自动实现 Java 和 C 的数据类型映射。你不再需要编写那个烦人的 C 动态链接库。

当然，这也意味着，使用 JNA 技术比使用 JNI 技术调用动态链接库会有些微的性能损失。可能速度会降低几倍。但对于绝大部分项目来说，影响不大。

JNA 调用原生函数

让我们先看一个 JNA 调用原生函数的例子。

例子 1 使用 JNA 调用原生函数

假设我们有一个动态链接库，发布了这样一个 C 函数：

```
void say(wchar_t* pValue){
    std::wcout.imbue(std::locale("chs"));
    std::wcout<<L"原生函数说: "<<pValue<<std::endl;
}
```

它需要传入一个 Unicode 编码的字符数组。然后在控制台上打印出一段中文字符。

为了调用这个原生函数，使用 JNA，我们需要编写这样的 Java 代码：

```
public interface TestDll1 extends Library {
    TestDll1 INSTANCE = (TestDll1)Native.loadLibrary("TestDll1", TestDll1.class);
    public void say(WString value);
}
```

这里，如果动态链接库是以 stdcall 方式输出函数，那么就继承 StdCallLibrary。

然后就可以像普通的 Java 程序那样调用这个接口：

```
public static void main(String[] args) {
    TestDll1.INSTANCE.say(new WString("Hello World!"));
    System.out.println("Java 输出。");
}
```

执行，可以看到控制台下如下输出：

原生函数说: Hello World!

Java 输出。

使用 JNA 调用原生函数的模式

JNA 不使用 native 关键字。

JNI 使用 native 关键字，使用一个个 Java 方法来代表外部的原生函数。

而 JNA 使用一个 Java 接口来代表一个动态链接库发布的所有函数。

对于不需要的原生函数，你可以不在 Java 接口中声明 Java 方法原型。

如果使用 JNI，你需要使用 System.loadLibrary 方法，把我们专为 JNI 编写的动态链接库载入进来。这个动态链接库实际上是我们真正需要的动态链接库的代理。

上例中使用 JNA 类库的 Native 类的 loadLibrary 方法，是直接把我们需要的动态链接库

载入进来。使用 JNA，我们不需要编写作为代理的动态链接库，不需要编写一行原生代码。

上面的 JNA 代码使用了单例，接口的静态变量返回的是接口的唯一实例，这个 Java 对象是 JNA 通过反射动态创建的。通过这个对象，我们可以调用动态链接库发布的函数。

Java 和原生代码的类型映射

跨平台、跨语言调用的最大难点，就是不同语言之间数据类型不一致造成的问题。绝大部分跨平台调用的失败，都是这个问题造成的。

JNA 使用的数据类型是 Java 的数据类型。而原生函数中使用的数据类型是原生函数的编程语言使用的数据类型。可能是 C,Delphi,汇编等语言的数据类型。因此，不一致是在所难免的。

JNA 提供了 Java 和原生代码的类型映射。

Java—C 和操作系统数据类型的对应表

Java 类型	C 类型	原生表现
boolean	int	32 位整数 (可定制)
byte	char	8 位整数
char	wchar_t	平台依赖
short	short	16 位整数
int	int	32 位整数
long	long long, __int64	64 位整数
float	float	32 位浮点数
double	double	64 位浮点数
Buffer Pointer	pointer	平台依赖(32 或 64 位指针)
<T>[] (基本类型的数组)	pointer array	32 或 64 位指针(参数/返回值) 邻接内存(结构体成员)

JNA 支持常见的数据类型的映射

Java 类型	C 类型	原生表现
String	char*	\0 结束的数组 (native encoding or jna. encoding)
WString	wchar_t*	\0 结束的数组(unicode)
String[]	char**	\0 结束的数组的数组

WString[]	wchar_t**	\0 结束的宽字符数组的数组
Structure	struct* struct	指向结构体的指针 (参数或返回值) (或者明确指定是结构体指针) 结构体(结构体的成员) (或者明确指定是结构体)
Union	union	等同于结构体
Structure[]	struct[]	结构体的数组, 邻接内存
Callback	<T> (*fp)()	Java 函数指针或原生函数指针
NativeMapped	varies	依赖于定义
NativeLong	long	平台依赖(32 或 64 位整数)
PointerType	pointer	和 Pointer 相同

跨平台、跨语言调用原则：

尽量使用基本、简单的数据类型；

尽量少跨平台、跨语言传递数据！

如果有复杂的数据类型需要在 Java 和原生函数中传递，那么我们就必须在 Java 中模拟大量复杂的原生类型。这将大大增加实现的难度，甚至无法实现。

如果在 Java 和原生函数间存在大量的数据传递，那么一方面，性能会有很大的损失。更为重要的是，Java 调用原生函数时，会把数据固定在内存中，这样原生函数才可以访问这些 Java 数据。这些数据，JVM 的 GC 不能管理，会造成内存碎片。

如果你需要调用的动态链接库中，有复杂的数据类型和庞大的跨平台数据传递。那么你应该另外写一些原生函数，把需要传递的数据类型简化，把需要传递的数据量简化。

JNA 模拟结构体

在原生代码中，结构体是经常使用的复杂数据类型。这里我们研究一下怎样使用 JNA 模拟结构体。

例 2 使用 JNA 调用使用 Struct 的 C 函数

假设我们现在有这样一个 C 语言结构体

```
struct UserStruct{
    long id;
    wchar_t* name;
    int age;
};
```

使用上述结构体的函数

```
#define MYLIBAPI extern "C" __declspec( dllexport )
```

MYLIBAPI void sayUser(UserStruct* pUserStruct);

对应的 Java 程序中，在例 1 的 接口中添加下列代码：

```
public static class UserStruct extends Structure{
    public NativeLong id;
    public WString name;
    public int age;
    public static class ByReference extends UserStruct
    implements Structure.ByReference { }
    public static class ByValue extends UserStruct implements
    Structure.ByValue
    { }
}

public void sayUser(UserStruct.ByReference struct);
```

Java 中的调用代码：

```
UserStruct userStruct=new UserStruct ();
    userStruct.id=new NativeLong(100);
    userStruct.age=30;
    userStruct.name=new WString("奥巴马");
TestDll1.INSTANCE.sayUser(userStruct);
```

Structure 说明

现在，我们就在 Java 中实现了对 C 语言的结构体的模拟。

这里，我们继承了 Structure 类，用这个类来模拟 C 语言的结构体。

必须注意，Structure 子类中的公共字段的顺序，必须与 C 语言中的结构的顺序一致。否则会报错！

因为，Java 调用动态链接库中的 C 函数，实际上就是一段内存作为函数的参数传递给 C 函数。

动态链接库以为这个参数就是 C 语言传过来的参数。

同时，C 语言的结构体是一个严格的规范，它定义了内存的次序。因此，JNA 中模拟的结构体的变量顺序绝对不能错。

如果一个 Struct 有 2 个 int 变量。 Int a, int b

如果 JNA 中的次序和 C 中的次序相反，那么不会报错，但是数据将会被传递到错误的字段中去。

Structure 类代表了一个原生结构体。当 Structure 对象作为一个函数的参数或者返回值传递时，它代表结构体指针。当它被用在另一个结构体内部作为一个字段时，它代表结构体本身。

另外，Structure 类有两个内部接口 Structure.ByReference 和 Structure.ByValue。这两个接口仅仅是标记，如果一个类实现 Structure.ByReference 接口，就表示这个类代表结构体指针。如果一个类实现 Structure.ByValue 接口，就表示这个类代表结构体本身。

使用这两个接口的实现类，可以明确定义我们的 Structure 实例表示的是结构体的指针还是结构体本身。

上面的例子中，由于 Structure 实例作为函数的参数使用，因此是结构体指针。所以这里直接使用了 UserStruct userStruct=new UserStruct ();

也可以使用 `UserStruct userStruct=new UserStruct.ByReference ();`
明确指出 `userStruct` 对象是结构体指针而不是结构体本身。

JNA 模拟复杂结构体

C 语言最主要的数据类型就是结构体。结构体可以内部可以嵌套结构体，这使它可以模拟任何类型的对象。

JNA 也可以模拟这类复杂的结构体。

例 3 结构体内部可以包含结构体对象的数组

```
struct CompanyStruct{  
    long id;  
    wchar_t* name;  
    UserStruct users[100];  
    int count;  
};
```

JNA 中可以这样模拟：

```
public static class CompanyStruct extends Structure{  
    public NativeLong id;  
    public WString name;  
    public UserStruct.ByValue[] users=new  
UserStruct.ByValue[100];  
    public int count;  
}
```

这里，必须给 `users` 字段赋值，否则不会分配 100 个 `UserStruct` 结构体的内存，这样 JNA 中的内存大小和原生代码中结构体的内存大小不一致，调用就会失败。

例 4 结构体内部可以包含结构体对象的指针的数组

```
struct CompanyStruct2{  
    long id;  
    wchar_t* name;  
    UserStruct* users[100];  
    int count;  
};
```

JNA 中可以这样模拟：

```
public static class CompanyStruct2 extends Structure{  
    public NativeLong id;  
    public WString name;  
    public UserStruct.ByReference[] users=new  
UserStruct.ByReference[100];
```

```

        public int count;
    }

    测试代码:
    CompanyStruct2.ByReference                                companyStruct2=new
    CompanyStruct2.ByReference();
        companyStruct2.id=new NativeLong(2);
        companyStruct2.name=new WString("Yahoo");
        companyStruct2.count=10;
        UserStruct.ByReference                                pUserStruct=new
    UserStruct.ByReference();
        pUserStruct.id=new NativeLong(90);
        pUserStruct.age=99;
        pUserStruct.name=new WString("杨致远");
    // pUserStruct.write();
        for(int i=0;i<companyStruct2.count;i++){
            companyStruct2.users[i]=pUserStruct;
        }
    TestDll11.INSTANCE.sayCompany2(companyStruct2);

```

执行测试代码，报错了。这是怎么回事？

考察 JNI 技术，我们发现 Java 调用原生函数时，会把传递给原生函数的 Java 数据固定在内存中，这样原生函数才可以访问这些 Java 数据。对于没有固定住的 Java 对象，GC 可以删除它，也可以移动它在内存中的位置，以使堆上的内存连续。如果原生函数访问没有被固定住的 Java 对象，就会导致调用失败。

固定住哪些 java 对象，是 JVM 根据原生函数调用自动判断的。而上面的 CompanyStruct2 结构体中的一个字段是 UserStruct 对象指针的数组，因此，JVM 在执行时只是固定住了 CompanyStruct2 对象的内存，而没有固定住 users 字段引用的 UserStruct 数组。因此，造成了错误。

我们需要把 users 字段引用的 UserStruct 数组的所有成员也全部固定住，禁止 GC 移动或者删除。

如果我们执行了 pUserStruct.write();这段代码，那么就可以成功执行上述代码。

Structure 类的 write()方法会把结构体的所有字段固定住，使原生函数可以访问。

原生代码调用 Java 代码

JNI 技术是双向的，既可以从 Java 代码中调用原生函数，也可以从原生函数中直接创建 Java 虚拟机，并调用 Java 代码。

但是，这样做要写大量 C 代码，对于广大 Java 程序员来说是很头疼的。

使用 JNA，我们就可以不写一行 C 代码，照样实现原生代码调用 Java 代码！

JNA 可以模拟函数指针，通过函数指针，就可以实现在原生代码中调用 Java 函数。

让我们先看一个模拟函数指针的 JNA 例子：

例 5 通过回调函数实现原生代码调用 **Java** 代码

```
int getValue(int (*fp)(int left,int right),int left,int right){
    return fp(left,right);
}
```

C 函数中通过函数指针调用外部传入的函数，执行任务。

JNA 中这样模拟函数指针：

```
public static interface Fp extends Callback {
    int invoke(int left,int right);
}
```

C 函数用如下 Java 方法声明代表：

```
public int getValue(Fp fp,int left,int right);
```

现在，我们有了代表函数指针 `int (*fp)(int left,int right)` 的接口 `Fp`，但是还没有 `Fp` 的实现类。

```
public static class FpAdd implements Fp{
    @Override
    public int invoke(int left, int right) {
        return left+right;
    }
}
```

JNA 回调函数说明

原生函数可以通过函数指针实现函数回调，调用外部函数来执行任务。这就是策略模式。

JNA 可以方便地模拟函数指针，把 Java 函数作为函数指针传递给原生函数，实现在原生代码中调用 Java 代码。

JNA 模拟指针

JNA 可以模拟原生代码中的指针。Java 和原生代码的类型映射表中的指针映射是这样的：

Java 类型	C 类型	原生表现
Buffer Pointer	pointer	平台依赖(32 或 64 位指针)
<T>[] (基本类型的数组)	pointer array	32 或 64 位指针(参数/返回值) 邻接内存(结构体成员)
PointerType	pointer	和 Pointer 相同

原生代码中的数组，可以使用 JNA 中对应类型的数组来表示。

原生代码中的指针，可以使用 `Pointer` 类型，或者 `PointerType` 类型及它们的子类型来模拟。

`Pointer` 代表原生代码中的指针。其属性 `peer` 就是原生代码中指针的地址。

我们不可以直接创建 `Pointer` 对象，但可以用它表示原生函数中的任何指针。

`Pointer` 类有 2 个子类：`Function`，`Memory`。

`Function` 类代表原生函数的指针，可以通过 `invoke(Class, Object[], Map)` 这一系列的方法调用原生函数。

`Memory` 类代表的是堆中的一段内存，它也是我们可以创建的 `Pointer` 子类。创建一个 `Memory` 类的实例，就是在原生代码的内存区中分配一块指定大小的内存。这块内存会在 GC 释放这个 Java 对象时被释放。`Memory` 类在指针模拟中会被经常用到。

`PointerType` 类代表的是一个类型安全的指针。`ByReference` 类是 `PointerType` 类的子类。`ByReference` 类代表指向堆内存的指针。`ByReference` 类非常简单。

```
public abstract class ByReference extends PointerType {
    protected ByReference(int dataSize) {
        setPointer(new Memory(dataSize));
    }
}
```

`ByReference` 类有很多子类，这些类都非常有用。

`ByteByReference`，`DoubleByReference`，`FloatByReference`，`IntByReference`，`LongByReference`，`NativeLongByReference`，`PointerByReference`，`ShortByReference`，`W32API.HANDLEByReference`，`X11.AtomByReference`，`X11.WindowByReference`

`ByteByReference` 等类顾名思义，就是指向原生代码中的字节数据的指针。

`PointerByReference` 类表示指向指针的指针。

在 JNA 中模拟指针，最常用到的就是 `Pointer` 类和 `PointerByReference` 类。`Pointer` 类代表指向任何东西的指针，`PointerByReference` 类表示指向指针的指针。`Pointer` 类更加通用，事实上 `PointerByReference` 类内部也持有 `Pointer` 类的实例。

`PointerByReference` 类可以嵌套使用，它所指向的指针，本身可能也是指向指针的指针。`PointerByReference` 类的源代码：

```
public class PointerByReference extends ByReference {
    public PointerByReference() {
        this(null);
    }

    public PointerByReference(Pointer value) {
        super(Pointer.SIZE);
        setValue(value);
    }

    public void setValue(Pointer value) {
        getPointer().setPointer(0, value);
    }

    public Pointer getValue() {
        return getPointer().getPointer(0);
    }
}
```

可以看到，`PointerByReference` 类的构造器做了如下工作：

- 1, 首先在堆中分配一个指针大小的内存, 并用一个 `Pointer` 对象代表。
`PointerByReference` 类的实例持有这个 `Pointer` 对象。
- 2, 然后, 这个堆上新创建的指针的值被设置为传入的参数的地址, 也就是指向传入的 `Pointer` 对象。这样, 新创建的 `Pointer` 对象就是指针的指针。

例 6 使用 **PointerByReference** 模拟指向指针的指针

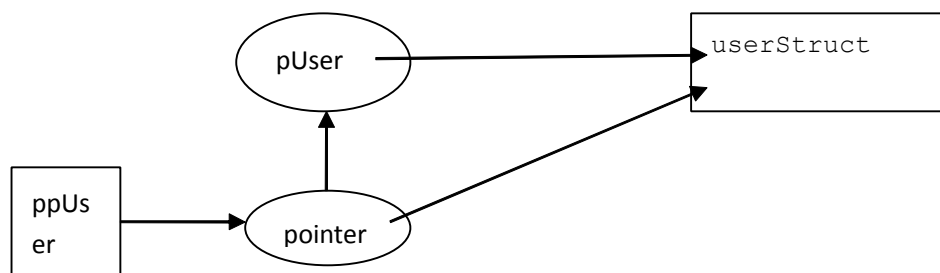
假设我们有一个结构体 `UserStruct` 的实例 `userStruct`, 现在又有了一个指向 `userStruct` 对象的指针 `pUser`。

为了得到 `UserStruct**` 指针在 Java 中的对等体, 我们可以执行如下代码:

```
PointerByReference ppUser=new PointerByReference(pUser);
```

这会在堆中创建一个指针 `pointer`, 然后把 `pUser` 指针的地址复制到 `pointer` 对象中, 这样 `pointer` 也就是指向 `pUser` 的指针。`Pointer` 对象就是代表 `UserStruct**` 类型的指针。可以使用 `ppUser.getPointer()` 方法返回 `pointer` 对象。

我们在 Java 和原生代码的类型映射表中曾经指出, `PointerType` 和 `Pointer` 类型相同, 都可以表示指针。`PointerByReference` 类是 `PointerType` 类的子类, 因此, `ppUser` 对象也可以代表 `UserStruct**` 类型的指针。



例 7 使用 `Pointer` 和 **PointerByReference** 模拟指针

下面, 给大家展示一个完整的例子, 展示如何使用 `Pointer` 和 `PointerByReference` 类型模拟各类原生指针。

C 代码:

```
void sayUser(UserStruct* pUserStruct) {
    std::wcout.imbue(std::locale("chs"));
    std::wcout<<L"ID:"<<pUserStruct->id<<std::endl;
    std::wcout<<L"姓名:"<<pUserStruct->name<<std::endl;
    std::wcout<<L"年龄:"<<pUserStruct->age<<std::endl;
}

void sayUser2(UserStruct** ppUserStruct) {
    //UserStruct** ppUserStruct=*pppUserStruct;
    UserStruct* pUserStruct=*ppUserStruct;
```

```

    sayUser(pUserStruct);
}

void sayUser3(UserStruct*** pppUserStruct) {
    //UserStruct** ppUserStruct=*pppUserStruct;
    UserStruct** ppUserStruct=*pppUserStruct;
    sayUser2(ppUserStruct);
}

```

然后发布这 3 个函数。

JNA 中模拟:

在接口中添加方法:

```

public void sayUser(UserStruct.ByReference struct);
public void sayUser2(PointerByReference ppUserStruct);
public void sayUser3(Pointer pppUserStruct);

```

JNA 中调用:

```

UserStruct pUserStruct2=new UserStruct();
pUserStruct2.id=new NativeLong(90);
pUserStruct2.age=99;
pUserStruct2.name=new WString("乔布斯");
pUserStruct2.write();
Pointer pPointer=pUserStruct2.getPointer();
PointerByReference ppUserStruct=new
PointerByReference(pPointer);
System.out.println("使用ppUserStruct!!!!");
TestDll1. INSTANCE.sayUser2(ppUserStruct);
System.out.println("使用pppUserStruct!!!!");
PointerByReference pppUserStruct=new
PointerByReference(ppUserStruct.getPointer());
TestDll1. INSTANCE.sayUser3(pppUserStruct.getPointer());

```

可以看到, 我们能够使用 Pointer 或者 PointerByReference 来表示指向指针的指针。
sayUser3 中, 我们使用了 PointerByReference 类的 getPointer()方法返回了代表 UserStruct*** 类型的指针。

事实上, 如果 `public void sayUser3(Pointer pppUserStruct);` 定义成 `public void sayUser3(PointerByReference pppUserStruct);` 也是可以的, 只是调用时提供的参数变为 pppUserStruct 对象本身即可。

通过使用 Pointer 和 PointerByReference 类, 我们可以模拟任何原生代码的指针。

Pointer 类详解

setPointer()方法相当于 `pTr2=&ptr1;`

setLong()方法相当于 `ptr2=&long;`

getPointer(0)相当于 `(void*) *ptr2;`

取指针指向的值, 返回的还是指针。

`getLong(0)`相当于 `(long)*ptr2;`
取指针指向的值，返回的是 `long` 类型的数据。

结语

JNA 打破了 Java 和原生代码原本泾渭分明的界限，实现了 Java 和原生代码的强强联合，在各自擅长的领域分工合作，快速解决问题。

Java 可以方便地利用原生代码的优势：执行速度快，可以直接操作硬件，机器码不容易被破解等。

原生代码可以通过回调 Java 函数，利用 Java 的优势：开发效率高，自动内存管理，跨平台，类库丰富，网络功能强大，支持多种脚本语言等。

JNA 为 Java 开发者打开了一扇通向广袤的原生代码世界的大门。