计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2015.20131166
52(1):141-155,2015

基于跨语言对象迁移策略的复合本地对象模型

黄玉坤[1,3] 陈 榕[1,2] 裴喜龙[1] 曹 璟[2]
[1](同济大学电子与信息工程学院 上海 200092)
[2](上海科泰世纪科技有限公司 上海 201203)
[3](江西财经大学信息管理学院 南昌 330029)
(yukun.huang.jx@gmail.com)

A Compound Native Object Model Based on the Strategy of Cross-Language Object Migration

Huang Yukun[1,3], Chen Rong[1,2], Pei Xilong[1], and Cao Jing[2]
[1](*Institute of Electronic and Information Engineering, Tongji University, Shanghai* 200092)
[2](*Shanghai Kortide Century Technology, LTD, Shanghai* 201203)
[3](*School of Information Technology, Jiangxi University of Finance and Economics, Nanchang* 330029)

Abstract The Java native interface (JNI) mechanism, which is designed to handle the interaction between native code and Java code, is currently widely utilized to develop mobile applications. However, JNI is observed hardly from perfection in two points: on one hand, the overhead of invoking functions of JNI interfaces heavily affects programs' runtime performance; on the other hand, the complexity of the JNI's programming specification prevents the integration and reusing of third party native components in Java code. To solve these problems, a new strategy is advised to migrate objects between Java components and native components by injecting necessary information of native objects into high-level objects. Guided by this strategy, a model of compound native objects (CNO) is proposed to integrate a Java object and a native object into a compound object which share same metadata maintained by Java class objects. Therefore the CNO model could literally reduce the overheads for the time saving of data type conversions, and lessen down the programming burden of the bridging code. A prototype of the CNO model is implemented on the basis of the Dalvik virtual machine such that Java could reuse third-party components in a dynamical and efficient way. Experiments show that the CNO model outweighs JNI in better performance of accessing native methods.

Key words Java native interface (JNI); cross-language; native component; metadata; Dalvik virtual machine

摘 要 Java 本地调用接口(Java native interface, JNI)机制被广泛应用在移动应用开发领域. JNI 机制中 JNI 接口函数被用于在本地代码中解析和转换 Java 端的数据类型和 Java 对象. 然而,JNI 接口函数的调用开销影响了程序运行的效率,其复杂的使用规范也是集成与复用第三方本地组件时的主要障碍. 提出一种基于跨语言对象迁移策略的复合本地对象模型,能够实现有效减少本地调用程序中的 JNI 接口函数调用开销和有效利用已有本地组件的目的. 详细讨论了复合本地对象的语言特性及其具体实现,并给出跨语言对象迁移规范以及开发实例. 在 Dalvik 虚拟机中实现了该模型,通过实验证明该策略和模型能够有效改善 JNI 机制的不足.

收稿日期:2013-08-11;修回日期:2014-05-12
基金项目:国家科技重大专项基金项目(2009ZX03004-005);"核高基"国家科技重大专项基金项目(2009ZX01039-002-002)

关键词 Java 本地调用接口；跨语言；本地组件；元数据；Dalvik 虚拟机

中图法分类号 TP301

近年来，Android 系统已成为目前最流行的移动智能设备操作系统，越来越多的开发者基于 Android 系统开发应用程序，其内嵌的 Dalvik 虚拟机[1]提供了 Java 程序的运行环境. 为提升程序运行效率和更好地利用硬件平台特性，Java 程序存在调用 C/C++等语言编写的本地代码的需求，因此 Java 本地调用接口(Java native interface，JNI)机制被广泛应用在移动应用开发领域. Android NDK (native development kit)[2-3]在 Dalvik 虚拟机中提供了 JNI 机制，使得 Android 上的应用程序一方面可以通过将性能敏感的业务用本地语言实现来提高程序运行效率，另一方面也可以使用已有的针对特定平台和特殊设备编写的本地库来提高程序的用户体验.

然而，使用 JNI 机制并不一定意味着程序的运行效率肯定会得到改善[4]. 使用 JNI 机制本身要付出一定的开销代价. 这些开销一方面来自于从 Java 代码中调用本地方法时，需要通过虚拟机内部的 JNI 机制对本地方法进行查找和绑定；另一方面来自于从本地代码中访问 Java 端的数据和对象时必须通过 JNI 接口函数进行转换. 当本地代码要频繁或者大量访问 Java 端数据和对象时，这种开销会变得非常显著[5-6]. 如果用本地代码实现关键业务所得的效率提升不能抵消以上 JNI 本身造成的开销，利用 JNI 实现的应用程序的运行效率就会降低.

其次，目前已经存在很多专门为移动智能设备开发的用 C/C++实现的通用基础类库和软件构件. 利用 JNI 技术框架实现遗留系统的迁移[7-8]，复用已有的本地库代码可以达到快速开发，减少开发成本的目的. 但是，使用 JNI 复用第三方软件构件的开发难度较大[9-10]. 要实现 Java 程序调用已有的本地构件中的方法需要使用 C 语言编写与本地库适配的桥接代码，并在桥接代码中通过 JNI 接口函数频繁地访问 Java 对象和进行大量的参数数据类型转换工作[11-12]. JNI 接口函数的复杂使用规范一直以来都是影响 Java 程序利用第三方构件中原生函数的开发效率的瓶颈，也是 Java 程序员不愿意复用已有本地代码的最大障碍. 同时，桥接代码的存在也使得 Java 代码与第三方组件的本地代码紧密地耦合在一起，使得 Java 程序丧失了其跨平台的能力[13].

因此，对于 Android 系统而言，研究如何提升本地调用程序的运行效率、有效集成和复用第三方组件的方法具有重要的意义和实用价值.

通过分析 JNI 机制的调用过程不难看出，JNI 程序的主要开销是由于跨语言对象之间的交互和数据类型转换时必须使用 JNI 接口函数造成的. 然而，JNI 机制无法从根本上避免 JNI 接口函数调用. 同时在本地方法中使用 JNI 接口函数也是 JNI 程序无法方便地跨平台移植的根本原因. 我们从跨语言对象的对象级迁移策略入手，提出了复合本地对象(compound native object，CNO)模型. 使用 CNO 模型提供的方法开发本地方法调用程序，不仅能极大地减少甚至避免在本地调用程序中使用 JNI 接口函数，提升本地调用程序的运行效率，而且还能在本地调用程序中有效集成和复用第三方组件，并很好地解决跨平台移植问题.

1 跨语言对象迁移策略

JNI 提供了一组接口函数用来访问全局引用、局部引用和 Java 对象中的成员字段. 但是通过不透明的引用和 JNI 接口函数来访问 Java 对象的开销要比直接访问 C 数据结构的开销高得多. 当本地代码需要访问和操作 Java 对象时，本地代码必须通过 JNI 提供的各种函数来操作虚拟机中的 Java 对象. 而每次操作 Java 对象的方法和字段都必须经过 2～3 次的 JNI 函数调用[9,14]. 当本地方法传递的参数含有大量复杂数据类型，如数组、字符串和对象类型时，本地方法中将大量使用 JNI 接口函数来解析这些数据类型和对象，这种开销将高得不可接受[15].

在提高 JNI 程序执行效率方面，大量的研究围绕 JNI 技术本身的开销提出了解决方案. 传统的改进策略是尽量减少在 JNI 方法调用、本地代码访问 Java 对象、传递数据和转换数据类型的过程中所产生的开销. 一些改进方法是函数内联[16]、缓存字段或方法的 ID 和钉住对象[17]，或者通过改进即时编译器[14]和虚拟机运行时环境[18]. 这些方法虽然有效，但是都忽视了从对象迁移的角度减少 JNI 交互的可能性.

JNI 机制体现了将计算能力向本地迁移的思想，它将 Java 对象中的成员方法声明为本地方法，然后用本地代码将其实现. 然而 JNI 机制并没有考

虑到计算资源向本地迁移的问题,作为有可能被本地代码加工和处理的数据仍然保留在 Java 域,这使得本地方法为了使用 Java 域中的数据而不得不通过 JNI 接口函数向上访问. Liang[4] 认为本地代码对 Java 对象的成员变量的访问和调用 Java 对象的方法所花费的时间很大程度上取决于 JNI 接口函数的多次调用的开销. 那么将频繁参与本地计算的 Java 对象中的数据也向本地迁移,则可以尽量减少本地代码通过 JNI 接口函数调用次数以及减少跨语言访问对象数据造成的接口转换代价. 我们把这种 Java 对象的数据(计算资源)和方法(计算能力)向本地迁移的策略称为跨语言对象本地化迁移.

要实现跨语言对象的迁移有几个关键问题需要考虑,包括跨语言对象的迁移形式、跨语言对象的耦合方法以及跨语言对象的元数据融合.

1.1 跨语言对象的迁移形式

1) 从 Java 对象迁移到本地对象

从 Java 对象迁移到本地对象是指将 Java 对象中的成员变量和成员方法都迁移到本地对象中实现. 但是,这并不意味着将 Java 对象从原来的 Java 对象系统中完全迁移出去. Java 对象被完全移除会对原来的 Java 程序的结构和实现产生较大的影响. Java 程序的重构会增加迁移的成本. 为了使 Java 对象的迁移不引起 Java 程序的重构,在 Java 对象的计算资源和计算能力被迁移到本地对象的过程中,Java 对象仍然要保留在原来的 Java 对象系统. 这样,在对 Java 对象本地化迁移的过程中产生了两个对象:一个是真正承载原来 Java 对象计算能力和计算资源的本地实体对象,一个是仍然保留在 Java 对象系统中的空壳对象,它被用来代理原来 Java 对象的角色和行为,因此称它为代理对象.

2) 从本地对象迁移到 Java 对象

从本地对象迁移到 Java 对象是指将本地对象中的成员变量和成员方法通过 Java 代理对象传递到 Java 对象系统中去. 即本地对象的计算资源和计算能力被传递到 Java 对象系统,本地对象本身并没有任何变化,但必须为本地对象在 Java 对象系统中构造一个 Java 代理对象,并通过代理对象与 Java 对象进行交互.

为了使 Java 代理对象能实现其代理功能,Java 代理对象需要与本地实体对象以某种方式耦合,使得其他 Java 对象可以通过 Java 代理对象访问本地实体对象中的方法和数据. 本地实体对象中的方法和数据在本地计算环境被使用和修改后,Java 代理对象能获取其新的状态.

1.2 跨语言对象的耦合方法

JNI 机制通过桥接代码实现 Java 代码与本地代码在方法级别上的耦合,桥接代码也被称为 JNI 层. 为了减少开发人员编写 JNI 代码的工作量,Ciacca 等人[19] 提出了共享残根(shared stubs)技术来简化编写桥接代码的封装类,但是其开发难度和开发效率不太理想. 而 Java-COM Bridge[20],JNA (Java native access)[21],Jawin[22],以及 SWIG(simplified wrapper and interface generator)[23-24] 等技术均采用了隐藏 JNI 层的策略,从如何自动生成桥接代码入手,提供桥接代码自动生成工具以及桥接代码的运行环境[25-26]. 但是无论提供何种 JNI 层隐藏技术,其隐藏的对象仅仅是面向开发者,虽然可以减少开发人员的工作量,但是都不能解决桥接代码中 JNI 接口函数调用开销的问题.

基于跨语言对象迁移的策略,本文尝试在对象级别上实现 Java 对象和本地实体对象的耦合. 选择在对象级别上进行耦合,主要考虑到以下几个方面:1)Java 代码与本地代码之间的交互更符合面向对象的编程模型;2)可以在被耦合的对象内部实现跨语言之间的基本数据类型以及引用数据类型的转换,从而消除 JNI 层;3)可以利用 Java 的动态特性、安全特性和垃圾回收(garbage collector, GC)机制来管理本地实体对象;4)Java 对象与本地实体对象的耦合只与类和接口的定义相关,而与 Java 语言和本地语言的具体实现代码无关. 这在解决了跨语言程序的跨平台和可移植性问题的同时,也使得 Java 程序开发者和本地程序开发者可以分别在自己擅长的领域中开发程序中的一部分.

在跨语言对象系统中,在对象级别上进行耦合并不是一个最新的想法. Java JNI bridge[27] 和基于 .NET 平台的 Bridge2Java[28] 利用桥接器技术;Java-XCOM[29] 和 JaCOb[30] 等利用中间件和 Web 服务技术,以及 Microsoft Java VM[31] 利用虚拟机技术等都提供了在对象和接口级别上解决多语言的互操作能力. JaCOb 使用的是基于中间件的 Web 服务技术实现 Java 对象与本地 COM 对象的耦合[32];Bridge2Java 是通过代理生成器读取 typelib 文件,将 COM 服务器对象的 Dispatch 接口、方法和属性转换成 Java 代理的对象和方法;Microsoft Java 虚拟机则是在虚拟机内部同时提供 Java 和 COM 两种运行环境,并为 Java 或 COM 对象生成对应环境下的 Wrapper 对象. 但是如前所述,桥接器、中间

件、Web 服务等技术存在效率问题,而基于 COM 技术的 Bridge2Java,Microsoft Java VM,WIN8 平台并不适合 Android 系统的生态环境.

如何将 Java 对象和本地对象在运行时耦合起来,并在耦合过程中消除 JNI 层,是实现 Java 对象的本地对象之间耦合的关键问题.同时,Java 代理对象与本地实体对象的耦合方式也决定了迁移方案的优劣和迁移过程的难易程度.本文提出的针对 Android 平台的跨语言对象耦合方法同时借鉴了 Bridge2Java 和 Microsoft Java 虚拟机的思路.一方面通过代理生成器将本地对象的接口、方法转换成 Java 代理的对象和方法,另一方面在 Dalvik 虚拟机中同时提供 Java 和本地对象的 Runtime 环境.区别在于两点:一是本地实体对象的实现不是基于 COM 技术和 Microsoft 平台,而是更适合嵌入式设备的 CAR 构件技术和 Elastos 平台;二是不同于 Microsoft 完整地开发一个 Microsoft JavaVM,本文仅对 Dalvik VM 的 JNI 框架进行扩展,实现在 Dalvik 虚拟机内部完成 Java 对象和本地实体对象的耦合.从对象在虚拟机内部的内存结构入手直接将 Java 对象和本地对象耦合成一个复合对象.这种基于虚拟机的跨语言对象的耦合方式需要对象元数据和反射机制的支持.

1.3 跨语言对象的元数据融合

由于在虚拟机内部一切对象都是用 C 语言数据结构表示,所以将 Java 对象和 C/C++对象耦合成一个复合对象是可能的.但是 Java 对象系统与本地实体对象系统的异构性,使得 Java 对象与本地实体对象的复合存在困难.其异构性体现在 Java 程序与 C++程序在代码执行方式、内存管理、元数据和反射能力等方面.

编程语言的元数据(metadata)主要描述了数据的类型信息.我们设想的 Java 对象与本地 C++对象的耦合方式是:将代理对象和实体对象的元数据表述融合起来,通过元数据将 Java 对象中的数据成员、接口和方法与本地实体对象中的相应目标绑定.然而实现元数据融合的前提是 Java 对象能够在运行时通过反射机制获得本地 C++对象的元数据信息,并且可以通过这些元数据和反射机制访问和管理 C++对象.C++对象不具有元数据和反射能力,使得我们无法实现元数据的融合.因此我们需要一个自带元数据的本地实体对象,并且在虚拟机的运行时中提供本地实体对象的元数据反射能力.

虽然 COM 也支持元数据和反射,但是其元数据和反射机制与 Java 的元数据和反射机制相比存在很大差异[33]:一是 COM 构件的元数据存储在注册表中;二是 COM 的反射机制是通过 IDispatch 接口和自动化机制实现的;三是 COM 的反射机制与 Java 的反射机制相比,在原理上和信息量上相差甚远[34].

我们选择 CAR(component assembly runtime)构件技术作为实现本地实体对象的编程规范.CAR 是一种轻量级的二进制构件标准,支持用 C/C++语言开发构件.CAR 构件的运行时由 Elastos OS 提供[35-36].CAR 构件的元数据是由 CAR 的接口定义文件经过 CAR 编译器生成的.ModuleInfo,ClassInfo,InterfaceInfo 和 MethodInfo 作为构件程序的元数据信息,用于描述构件导出的接口及方法列表.它们被存储在 CAR 构件模块的资源段中,随着构件模块一起加载到运行时环境,并通过 CAR 构件反射机制读取这些元数据信息[37].

选择 CAR 构件技术作为本地实体对象系统的开发技术的原因除了 CAR 构件技术的元数据和反射机制与 Java 的元数据和反射机制在设计上、效率上和能力上有一定程度的相似性和可比性之外,还有 CAR 构件技术和 Elastos 构件系统非常适合在资源受限的移动嵌入式设备上部署和运行各种服务和应用程序.还有一个重要的因素是,整个 Elastos 运行时环境可以以动态链接库的形式非常简单方便地嵌入到 Dalvik 运行时环境中.这使得在 Android 系统中可以用很低的代价嵌入 Elastos 运行时环境并使用 CAR 构件.在 CAR 构件技术的支撑下,我们设计了一种复合本地对象模型,并采用一种元数据注入方法实现 Java 对象与本地 CAR 对象元数据的融合.

2 复合本地对象模型

本文提出一种复合本地对象(compound native object,CNO)模型,实现 Java 对象与本地 C++/CAR 对象的耦合,进而实现 Java 构件对象与 CAR 本地构件对象之间的相互迁移.

2.1 CNO 模型

定义 1. CNO 对象是一个在虚拟机内部将本地实体对象与 Java 代理对象耦合在一起形成的跨语言复合对象.

可以认为 CNO 对象是一个具有 Java 对象外壳的 CAR 本地实体对象,对于 Java 域的其他 Java 对

象来说,CNO 对象与其他普通 Java 对象的交互行为是一样的,但 CNO 对象对消息的反应和处理都是通过内部的 CAR 本地实体对象完成. CNO 由两个既相互独立又相互紧密联系的对象组成:处于 Java 域的 Java 代理对象和处于本地域的 CAR 本地实体对象.

Java 代理对象和 CAR 本地实体对象的相互独立性体现在:

1) Java 代理对象和 CAR 本地实体对象在内存中分配的空间分别处于 Java 堆和本地堆,但这两个空间都属于同一个线程.

2) Java 代理对象和 CAR 本地实体对象各自遵守自己实现语言的编程规范、对象行为、语言特性.

Java 代理对象和 CAR 本地实体对象的紧密联系性体现在:

1) Java 代理对象是 CAR 本地实体对象在 Java 域中的代理,它在 Java 域中以代理的角色为普通 Java 对象提供 CAR 对象的本地计算资源和计算能力.

2) CAR 本地实体对象依赖于 Java 代理对象存在,它的生命周期受到 Java 代理对象的控制.

CNO 对象模型不仅仅在方法调用和数据访问上实现跨语言互操作,还可以实现跨语言的继承和垃圾收集等比较高级的语言设施.

2.2 CNO 的继承

从软件开发角度来看,继承允许程序员复用已有代码. 从对象系统的角度来看,继承允许一个父类将其状态和行为传递给子类. 传统的继承都是应用于单一语言环境,子类继承父类的所有非私有的属性和方法. CNO 对象的继承包含 3 种情况:

1) 一个 CNO 类可以作为一个 Java 类被其他 Java 类继承,继承中的虚/实方法引用规则遵循 Java 原则,如图 1(a)所示.

一个 CNO 类可以在 Java 域中派生子类,比如 FooBar 类继承了 CNO 类 Foo,则 FooBar 的对象也自动得到 Foo 对象中的 CAR 本地构件对象. 同时子类 FooBar 可以继承父类 Foo 的方法(也就是 native 方法).

2) 一个 CNO 类内部的 CAR 构件类可以作为子类继承自其他 CAR 构件类,其父类又是另一个 CAR 构件类的子类等等. 在这条继承链上所有的构件类都将成为这个最初的构件类的父类,并使得 Java 代理对象可以通过与其对应的 CAR 本地构件对象访问整个构件对象的继承链,如图 1(b)所示.

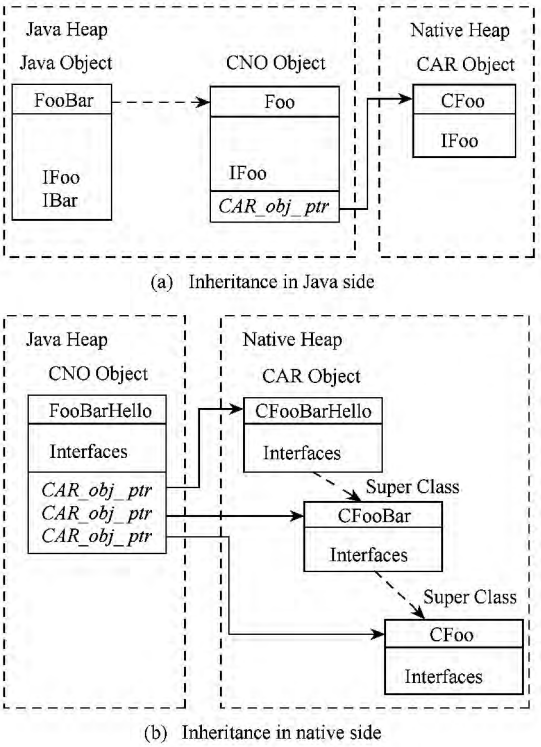


Fig. 1 Inheritance mechanism of CNO classes.

图 1 CNO 类的继承机制

需要特别注意的是,第 2 种继承情况是通过构件聚合实现的. CAR 有和 Java 语义上相同的基于类的继承和接口继承,不过实现继承的内部机制是 CAR 特有的二进制继承机制[38],它是一种以黑盒复用的方式直接使用被继承的 CAR 对象的二进制可执行代码,子类和父类以聚合的方式组成一个新的类.

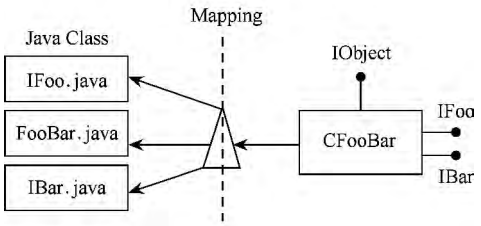
3) 通过多个 CAR 构件类实现一个 Java 类的多个本地化接口. 即定义 Java 类的多个接口为本地接口,并且每个接口用一个 CAR 构件类来实现,则 CNO 复合对象内部将按照接口继承链的顺序,依次生成多个 CAR 本地实体对象. 其继承图示与图 1(b)类似.

2.3 CNO 的迁移映射关系和耦合方式

在 CNO 模型中,Java 代理对象和 CAR 本地实体对象之间并不一定是一对一的迁移映射关系. 除了上述由于 CNO 的继承,造成一个 Java 代理对象可能对应于多个 CAR 本地实体对象之外. 也可能一个 CAR 本地实体对象对应多个 Java 代理对象. 当 CAR 本地对象迁移到 Java 代理对象时,一个 CAR 本地实体对象可能实现了多个接口,为了降低 CNO 对象的复杂度,我们将 CAR 本地实体对象类中包含的多个接口分开,分别将它们映射成独立的 Java 代理对象,如图 2 所示:

```
FooBar. car
module{
        interface IFoo{
            foo([in] Int32 i,
                [in] Int32 j,
                [out] Int32 * o);}
        interface IBar{
            bar([out] String *str);}
        class CFooBar{
            interface IFoo;
            interface IBar;}
}
```

(a) CFooBar component class



(b) Mapping between CAR class and Java proxy classes

Fig. 2 CFooBar component class and its corresponding Java proxy classes.

图 2 CFooBar 构件类及其对应的 3 个 Java 代理类

在实现 CAR 本地构件对象向 Java 构件对象迁移的过程中，Java 代理类和 CAR 本地类的迁移映射关系是：对于 CAR 构件模块中的每一个类、以及类所包含的接口都对应一个 Java 代理类. 以 CFooBar 的 CAR 构件类为例，CFooBar 是一个本地类，它实现了 IFoo 和 IBar 两个接口. 图 2(a)是它的模块接口描述文件. 图 2(b)将一个 CFooBar 构件类解析后映射成 3 个 Java 代理类：FooBar. java，IFoo. java 和 IBar. java.

这 3 个 Java 代理类都与 CFooBar 本地类复合形成如图 3 所示的 CNO 复合本地类. 3 个 CNO 类的区别在于 FooBar. java 代理类负责实例化 CFooBar 本地类，并负责 CFooBar 本地类的生命周期，CFooBar 本地实体对象将它的 IObject 接口的引用注入 FooBar. java 代理对象；而 IFoo. java 和 IBar. java 类不负责实例化 CFooBar 本地实体对象. 在实例化 IFoo. java 和 IBar. java 类对象时，CFooBar 本地实体对象已经被 FooBar. java 代理对象实例化. 不过他们需要获得 CFooBar 本地实体对象的 IFoo 和 IBar 接口的引用，实现 CFooBar 本地实体对象的注入.

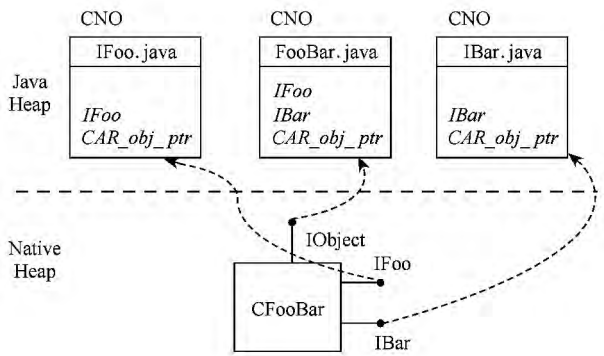


Fig. 3 Coupling between Java object and CAR object.

图 3 3 种 Java 代理对象与 CAR 本地对象的耦合方式

由于在面向构件编程中，外部客户都是通过接口使用构件对象，接口是引用对象的唯一方式，因此，我们将 CAR 构件对象的接口也映射为一个独立的 Java 代理对象. 除了在接口描述文件中显式定义的 IFoo 和 IBar 接口之外，每个 CAR 构件对象还有一个隐藏的 IObject 接口. 所有的 CAR 构件类都继承了该接口，它类似于 COM 构件类的 IUnknown 接口.

我们把 FooBar. java 称为本地构件类代理对象，而把 IFoo. java 和 IBar. java 类称为本地构件接口代理对象. 3 个 Java 代理类 FooBar. java，IFoo. java 和 IBar. java 的定义如图 4 所示：

```
FooBar. java
package com. elastos. cno. foobar;
class FooBar{
    private String strField=" Elastos";
    native int foo(int a,int b);
    native String bar();
}
IFoo. java
package com. elastos. cno. foobar;
class IFoo{
    native int foo(int a,int b);
}
IBar. java
package com. elastos. cno. foobar;
class IBar{
    native String bar();
}
```

Fig. 4 Definition of Java proxy classes.

图 4 Java 代理类的定义

2.4 CNO 复合本地对象的元数据与定义规范

CNO 复合本地对象模型的元数据包含两个层次：1)编译时的元数据；2)运行时的元数据. 编译时的元数据是指如何让程序员和编译程序认识和理解一个 CNO 类所需的元数据. 编译时元数据包括 CNO 类的类型标识、本地实体类所在的构件模块名称、类名称或接口名称. 编译时元数据在虚拟机解释执行的过程中也将起作用. 比如，在加载一个 CNO

类时,虚拟机需要通过 Java 类中的 Annotation 来获取本地 CAR 对象与 Java 代理对象的映射关系,并根据构件模块名称、类名称或接口名称自动从系统构件库目录加载相应的 CAR 构件的模块文件和构件类. 而运行时的元数据是指虚拟机在运行时实现 CNO 对象内部耦合所需的元数据. 运行时的元数据将在第 3 节 CNO 复合本地对象在 Dalvik 虚拟机上的实现中讨论.

本文采用 Annotation 方式在 Java 代理对象中写入编译时元数据,使其成为一个 CNO 类. Java 规范中的 Annotation 是与一个程序元素相关联信息或者元数据的标注. 它不影响 Java 程序的执行,但是对例如编译器警告或者对文档生成器等辅助工具产生影响. 通过给类增加@注释,可以将程序中的类、方法、属性和元数据联系起来,并且将元数据存储在类文件中通过反射的方式获取[39].

CNO 类的定义规范:

1) 通过 Annotation 类@ElastosClass 来标识一个 Java 代理类,并说明对应的 CAR 构件类所在的 CAR 构件模块名称以及类的名称,则该 Java 类是一个本地构件类代理类,该 CNO 对象是复合本地类对象. @ElastosClass 的书写规范为 @ElastosClass(*Module*="模块名称",*Class*="构件类名称").

2) 通过 Annotation 类@ElastosInterface 来标识一个 Java 代理类,并说明对应的 CAR 构件接口所在的 CAR 构件模块名称以及接口名称,则该 Java 类是一个本地构件接口代理类,该 CNO 对象是复合本地接口对象. @ElastosInterface 的书写规范为 @ElastosInterface(*Module*="模块名称",*Interface*="接口名称").

以 FooBar 构件模块为例,图 4 已给出 FooBar 构件对应的 3 个 Java 代理类 FooBar.java,IFoo.java 和 IBar.java 的定义,现在我们可以用以上 Annotation 类注释这 3 个 Java 代理类,使其成为 CNO 类,如图 5 所示.

```
FooBar.java
package com.elastos.cno.foobar;
import dalvik.annotation.CAR;
@ElastosClass(Module=" FooBar",Class=" CFooBar")
class FooBar{
    private String strField=" Elastos";
    native int foo(int a,int b);
    native String bar();}
IFoo.java
package com.elastos.cno.foobar;
import dalvik.annotation.CAR;
@ElastosInterface(Module=" FooBar",Interface=" IFoo")
class IFoo{
    native int foo(int a,int b);}
IBar.java
package com.elastos.cno.foobar;
import dalvik.annotation.CAR;
@ElastosInterface(Module=" FooBar",Interface=" IBar")
class IBar{
    native String bar();}
```

Fig. 5 Definition of CNO classes.

图 5 CNO 类的定义

2.5 CNO 与 JNI 实现本地调用程序代码比较

为比较 JNI 及 CNO 实现 Java 程序与本地代码互调在编程规范上的不同,我们设计了一个示例程序,该程序包含了两个 Java 类(Foo 和 FooJniDemo). 其中,Foo 类定义了一个 native 方法 modifyFoo,该方法接收 3 种数据类型的参数并返回一个 Foo 类的新对象. FooJniDemo 类是 Foo 类的用户类,负责实例化 Foo 对象并调用 Foo 类的本地方法 modifyFoo. 另外,FooJniDemo 类还定义了一个普通 Java 方法 *modifyFooCallback*. 在 Foo 类的 *modifyFoo* 方法的本地实现中还包含调用 *modifyFooCallback* 方法的语句,即在本地代码中反向回调 Java 方法. 图 6 给出了 JNI 与 CNO 在 Java 端的代码实现,图 7 给出了 JNI 与 CNO 在本地端的代码实现,JNI 与 CNO 在具体实现上的不同之处在图 6、图 7 中作了标记.

比较图 6 左右两边代码,两者区别主要体现在用线框标注的 4 处地方:1)基于 CNO 实现的本地调用程序必须加载 Elastos 运行时,并引入有关 CNO 元数据声明的 Annotaion 标注类,即必须在 CNO 实现代码的 1 号线框中声明以上 2 条语句;2)由于实现了 CNO 机制的 Dalvik 虚拟机可以根据 CNO 类的 Annotaion 自动加载本地对象,所以在 CNO 实现代码中不再需要显式地加载本地库;3)由于 Foo 类包含了本地方法,所以 Foo 类需要改写成一个 CNO 类,即为 Foo 类增加 Annotaion 标注语句;4)为了能在 CNO 本地对象中调用 Java 对象的方法,我们使用了回调函数注册技术,如 CNO 实现代码的 4 号线框中语句.

比较图 7 两边代码容易看出,在 JNI 机制下,本地端方法所要实现的语句无论是在数量上还是在复杂程度上都要比 CNO 方法实现的代码多且复杂. 图 7 右边的 1~3 号线框分别是访问 Java 对象的静态字段、创建 Java 对象和回调 Java 方法. 而在 CNO

```
JNI Sample
Foo.java
package com.elastos.foojni


public class Foo{
   /*Declarations of the fields*/
   public native Foo modifyFoo(int i,String str,int[] arr);
}
FooJniDemo.java
package com.elastos.foojni
public class FooJniDemo{
   /*Declarations*/
   private void modifyFooCallback() {
      System.out.println("Callback from native.!");
   }
   public static void main(String args[]) {
      FooJniDemo demo = new FooJniDemo();
      ⋮
      foo = new Foo(18,arr1);
      newfoo = foo.modifyFoo(1,"Good-bye!",arr2);
   }
   static {
         System.loadLibrary("foojni");   2
   }
}
```

```
CNO Sample
Foo.java
package com.elastos.foojni
package com.elastos.runtime;
import dalvik.annotation.ElastosClass;   1
@ElastosClass(Module="FooJni.dll", Class="CFoo")   3
public class Foo{
   /*Declarations of the fields*/
   public native Foo modifyFoo(int i,String str,int[] arr);
}
FooJniDemo.java
package com.elastos.foojni
public class FooJniDemo{
   /*Declarations*/
   private void modifyFooCallback() {
      System.out.println("Callback from native!");
   }
   public static void main(String args[]) {
      FooJniDemo demo = new FooJniDemo();
      ⋮
      foo = new Foo(18, arr1);
      ElastosCallbackFunc.addCallbackHandler(
       foo,"modifyFooEvent",   4
       "FooJniDemo","modifyFooCallback","()V");
      newfoo = foo.modifyFoo(1,"Good-bye!",arr2);
   }
}
```

Fig. 6 Sample codes of JNI and CNO in Java side.

图 6 JNI 与 CNO 在 Java 端的代码实现

```
JNI Sample
foojni.c
Jobject java_com_elastos_foojni_foo_modifyFoo(JNIEnv
* env, object obj, int i,String str,int[] arr)
{
   jfieldID strFieldId = env->GetStaticFieldID(clazz,
"stringField", "Ljava/lang/String;");   1
   jstring jstr_foo =
(jstring) env->GetStaticObjectField(clazz, strFieldId);
   const char *str_foo = (jstr_foo==0) ? 0 :
env->GetStringUTFChars(jstr_foo, 0);
   str_newfoo = str_foo + str;
   ⋮
   env->ReleaseStringUTFChars(jstr_foo, str_foo);
   …
   jclass newcls =(*env)->FindClass(env,
"com/elastos/foojni/foo");   2
   jmethodID mid_init = (*env)->GetMethodID(env,
newcls, "<init>", "()V");
   jobject newfoo = (*env)->NewObjectA(env, newcls,
mid_init, 0);
   ⋮
   jclass cls = (*env)->GetObjectClass(env, obj);   3
   jmethodID mid_callback = (*env)->GetMethodID(env,
cls, "modifyFooCallback", "()V");
   (*env)->CallVoidMethod(env, obj, mid_callback);
}
```

```
CNO Sample
CFoo.cpp
String CFoo::s_stringField("Hello,world!");
   ⋮
ECode CFoo::modifyFoo( /* [in] */ Int32 i, /* [in] */
const String& str, /* [in] */ const ArrayOf<Int32> &
array, /* [out, callee] */ IFoo ** ppNewFoo)
{
   ⋮
   s_stringField += str;
   …
   ECode ec = CFoo::New(ni, *narray, &pNewFoo);
   if (FAILED(ec)) return ec;
   *ppNewFoo = pNewFoo;
   ⋮
   Callback::modifyFooEvent();
   return NOERROR;
}


ECode CFoo::constructor(
   /* [in] */ Int32 intField,
   /* [in] */ const ArrayOf<Int32> &arrayField)
{
   m_intField = intField;
   m_intArray = arrayField.Clone();
   return NOERROR;
}
```

Fig. 7 Sample codes of JNI and CNO in native side.

图 7 JNI 与 CNO 在 native 端的代码实现

本地代码中访问一个 Java 对象的静态字段实际上只要直接访问该 Java 对象对应的本地 CAR 对象的静态成员变量. 对于创建一个 Java 对象，在 CNO 本地代码中只需调用 CFoo 类的 New 方法. 在 CNO 本地代码中回调 Java 方法也只需通过一条 Callback::*modifyFooEvent*()；语句触发 CAR 对象的回调事件，Elastos 运行时会自动将 CNO 示例程序的 Java 端代码中注册的回调函数信息传递给 Dalvik 虚拟机，并通过虚拟机调用 Java 方法.

3 CNO 对象模型在 Dalvik 虚拟机中的实现

在跨语言边界的对象软件系统中，CNO 对象可以很自然地与 Java 和 C++两种语言边界内的对象

进行交互,并遵循各自的编程规范.我们认为虚拟机可以成为这个跨语言对象生态系统的管理者.通过虚拟机在对象层级把所有的对象系统融合,负责所有对象(普通 Java 对象、CNO 对象、普通 CAR 对象)的生命周期管理,为整个跨语言对象生态系统提供运行环境.

3.1 CNO 对象的运行时元数据

CNO 对象运行时元数据的融合是实现 Java 对象与本地 CAR 对象耦合的基础.CNO 对象的运行时元数据包括 Java 代理对象的元数据和本地实体对象的元数据.Java 代理对象的元数据实际上就是 Java 对象的类对象 ClassObject;而本地实体对象的元数据包括本地实体对象的构件模块元数据、类元数据、接口元数据和方法元数据.CNO 对象运行时元数据融合的方式是:在 Java 代理对象的 ClassObject 结构中注入本地实体对象运行时元数据.

1) 本地对象运行时元数据信息注入 Java 类对象

元数据融合的方法是指当实例化一个 CNO 对象时,将 CAR 本地类的元数据信息和方法元数据信息注入 Java 代理对象的 ClassObject 的相关数据结构.每当虚拟机加载一个标有@ElastosXxx 形式的 Anotation 元数据的 Java 类时,会在 Java 代理对象的 ClassObject 的 *accessFlags* 中设置一个标识,标明它是一个 CNO 本地类复合对象还是一个 CNO 本地接口复合对象.同时利用 CAR 构件的反射机制加载对应的 CAR 本地类,并得到 CAR 构件中的类信息和方法信息.具体过程是根据@ElastosClass 元数据中的 *Module* 属性得到 CAR 构件模块的名称,根据名称加载 CAR 构件模块,并得到它的构件信息 IModuleInfo.然后根据@ElastosClass 元数据中的 Class 属性得到类的名称,并从 IModuleInfo 中反射得到其类信息 IClassInfo,将其保存在 ClassObject 类的结构体中.虚拟机将通过这些注入的本地对象运行时元数据将 Java 代理类与 CAR 构件类中的对应结构关联起来.为了实现 CAR 本地类的元数据信息注入,我们修改了 ClassObject 的内存结构,如图 8 所示.

首先,本文在 ClossObject 结构中增加了 3 个 *accessFlags* 值用来表示该类是 CAR 本地类的 Java 代理类.当 *accessFlags* 的值为 CLASS__CAR__CLASS 时表示该 Java 代理类对应的是一个 CAR 构件本地类;当 *accessFlags* 的值为 CLASS_CAR__INTERFACE 时表示该 Java 代理类对应的是一个

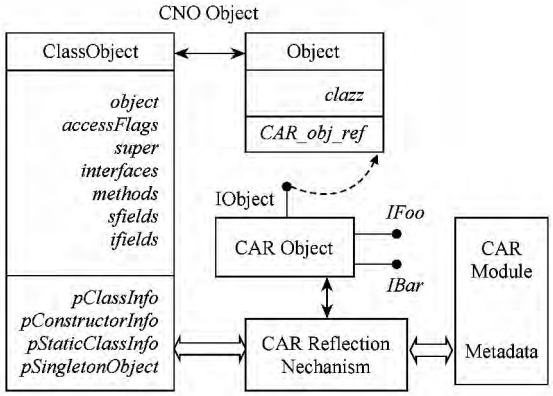


Fig. 8 Module and class's metadata injection.

图 8 模块与类元数据注入

CAR 构件本地接口;当 *accessFlags* 的值为 CLASS__CAR_NEEDCLEAN 时表示需要清理该 Java 代理对象内部的 CAR 构件对象,该标记用于管理 CAR 构件本地实体对象的垃圾回收.其次,在 ClassObject 结构体最后位置增加了 4 个指针,用于指向 CAR 构件的相关元数据反射对象指针,分别是构件类信息反射对象指针 *pClassInfo*、构造函数信息反射对象指针 *pConstructorInfo*、静态类信息反射对象指针 *pStaticClassInfo* 以及类的唯一实例对象指针 *pCARSingletonObject*.

2) CAR 对象方法元数据信息注入 Java 类对象

CNO 对象中的 CAR 方法相对于 Java 来说是后期延迟动态绑定的.虚拟机在 Java 本地方法首次被调用前进行链接,将其绑定到 CAR 本地方法.CNO 对象绑定本地方法的步骤是:在加载复合对象类的过程中,对本地方法建立内部数据结构,将 CAR 本地实体对象的方法元数据注入 Java 代理类的本地方法结构体.若方法是首次被调用,此时 CNO 对象的 ClassObject 的 Method 结构体中 *nativeFunc* 指针指向函数 *dvmResolveNativeMethod*(),其主要作用是通过解析注入的方法元数据对 CAR 构件方法进行解析,找到该方法对应的构件类,并获得相应的构件方法的实际地址,并将其保存到该 Method 结构体中的 *nativeFunc* 成员中,使得所有对这个 Java 本地方法的调用都直接跳转至本地实体对象的相应函数.由于 *nativeFunc* 指针改为指向 CAR 构件本地方法的实际地址,因此之后每次调用该本地方法都跳过方法解析,直接进入方法执行.

本地方法元数据 IMethodInfo 的反射过程和类信息的反射过程类似,通过加载类时保存的类元数据反射接口 IClassInfo,并结合方法名称反射得到方法元数据的反射接口 IMethodInfo.通过 IMethodInfo

可以得到方法的参数列表和数据类型信息. 为了在CNO对象的ClassObject对象中注入CAR构件对象方法元数据,需要定义一个新的结构体CARMethodInfo用来存储CAR构件中的方法元数据,如图9所示:

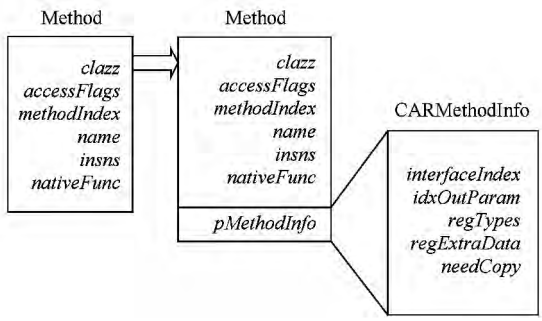


Fig. 9 Native method's metadata injection.

图9 本地方法元数据注入

图9中,*pMethodInfo*指向CAR本地实体对象的方法信息的反射接口,在通过CAR反射机制得到IMethodInfo的指针后,将其保存在这个指针中;*interfaceIndex*指示当前方法是CAR本地类实现的第几个接口,可以根据*interfaceIndex*定位当前方法所在的接口的*vtable*指针;*idxOutParam*指示当前方法的第几个参数开始对应的是CAR本地方法的输出参数;*regTypes*指示参数的类型;*regExtraData*指示参数的附加类型,通常用于表示ArrayOf〈Type〉或者BufferOf〈Type〉中的Type类型;*needCopy*表示传递的参数需要拷贝. 最后,在ClassObject的Method结构体中还增加了一个指针*pCARMethodInfo*指向CARMethodInfo.

3.2 CNO对象的实例化

在本文改进的Dalvik虚拟机中,除了支持传统的Java类实例化外,还支持CNO对象类的实例化. 我们将CNO对象视为一个完整的对象,构成这个CNO对象的Java代理构件对象和CAR本地构件对象同生共死,其生命周期与普通Java对象一样包括3个阶段:对象的实例化、对象的使用以及对象的销毁. CNO对象与普通Java对象的实例化时机基本相同.

在不考虑继承的情况下,CNO对象中参与复合的过程只涉及两个对象,Java代理对象和CAR本地实体对象. 这时需要在Java代理对象的实例对象后面多分配一个4 B大小的对象引用单元来存储CAR对象实例引用.

在考虑继承的情况下,参与复合的过程则涉及多个对象,包括1个Java代理对象和1组与该Java代理对象相关的多个CAR本地实体对象. 如果本地构件类继承自其他本地构件类,则需要在CNO对象的实例化过程中为CNO对象分配内存空间. 首先计算其本地构件类所有父类的数目;然后根据计算出来的父类数目在Java代理对象中分配相应的对象引用单元来存储CAR对象实例引用;最后对每一个父类实例化相应的CAR本地实体对象,并将指针保存在Java代理对象为其分配的对象引用单元之中. 当要使用Java代理对象中第n层上的父类CAR对象时,只要取出Java对象之后的第$4\times n$个字节的内容,强制转换成对应的指针即可.

3.3 CNO对象的销毁与垃圾收集

CNO对象的生命周期与普通Java对象的生命周期是一致的. 不同于Java对象采用垃圾收集机制完全隐性的方式进行管理,CAR构件对象采用引用计数的方式对对象的生命周期进行管理. 因此,为了保持代理对象与CAR构件对象的生命周期一致,我们强制规定Java代理对象与其对应的本地构件对象之间的引用计数为一. 虚拟机会在GC堆内存不足时以Mark-Sweep机制进行GC对象回收. 因此当GC回收对象时,同步释放整个CNO对象. 通过这种销毁策略,可以利用Java的GC机制管理本地实体对象在内存中的生命周期,但是需要确保垃圾回收器不会将那些正在被本地方法调用的对象释放.

Dalvik的垃圾回收机制采用了多种算法,其中一种算法是Copy,使用Copy算法回收垃圾时,Java对象可能会被移动位置. CAR对象在初始化时会记录其对应的Java代理对象的地址,一旦这个Java对象被移动CAR对象应该更新这条记录,否则用旧的值去调用目标Java对象的方法必然出错. 所以,在Java对象被移动时需要让虚拟机通知一下附属其上的CAR对象.

3.4 数据类型转换

CNO对象将JNI这样的跨语言调用和参数数据类型转换隐藏在复合对象的内部,由虚拟机在运行时完成,使其对开发者透明. 因此,我们必须在Dalvik虚拟机内部的本地方法调用过程中增加本地方法的参数和返回值数据类型的转换机制.

表1为Java与CAR的数据类型映射表. 对于CAR构件技术中部分基本数据类型,如Boolean, Byte, Int16, Int32, Int64, Float, Double, Char等,Java语言中都有相应的基本类型对应,如boolean, byte, short, int, long, float, double, char等. 但对于

CAR 构件中另一部分数据类型,如 Int8,UInt16,UInt32,UInt64 等,Java 语言并没有完全的对应类型.这种类型可以在保证无精度损失的前提下采用更高精度的 Java 类型表示,并在参数转换时进行相应的调整.

Table 1 Data Types of Java and CAR

表 1 Java 与 CAR 的数据类型

Java	CAR	Description
boolean	Boolean	8 b Integer
byte	Byte	8 b Signed Integer
short	Int16	16 b Signed Short
int	Int32	32 b Signed Integer
long	Int64	64 b Signed Long
char	Char16	16 b Unsigned Integer
float	Float	32 b IEEE Float
double	Double	64 b IEEE Float
Object	Interface	Object
String	String	String
StringBuffer	StringBuf	String Buffer
array[]	ArrayOf	Array
array[]	BufferOf	Array

String,StringBuffer 类型在 Java 中作为引用类型分别和 CAR 构件中的内置类型 String 和 StringBuf 相对应.另外,Java 中的数组类型和 CAR 构件中的 ArrayOf 或 BufferOf 相对应.在转换过程中对于引用类型或者其他复杂类型的转换,需要利用 CAR 的反射机制.比如 Java 中的一维数组类型参数 int[],在转换成 CAR 对应的数组类型 ArrayOf〈Int32〉时,需要通过 CAR 的反射机制获得 ArrayOf 对象类,并通过类厂创建一个 ArrayOf〈Int32〉对象.对于 Java 语言中没有对应类型的一部分 CAR 数据类型,如 DataTime 结构体类型,可以在 Java 端定义类似的结构体类完成 Java 端对这类数据类型的调用.

4 实验及评估

为了验证本文所提出 CNO 对象模型和跨语言对象系统的可行性,我们在 Gingerbread(Android2.3.7)系统的 Dalvik 虚拟机基础上进行扩展,实现了一个支持 CNO 对象模型的虚拟机.本节我们先对 CNO 的性能进行分析,然后通过几个测试用例,验证采用 CNO 对象模型实现的跨语言本地调用程序在执行效率、集成与复用第三方构件和跨平台性上是否比传统 JNI 程序更加高效.

4.1 CNO 性能分析

本文提出 CNO 模型的目的是为了使 Java 程序可以有效利用第三方本地构件,并提高本地调用程序的运行效率.然而 CNO 为程序运行效率的提升付出了一部分内存空间的代价:1)为了支持 CAR 对象,需要在 Andriod 系统中加载 Elastos 运行时,这将额外占用数兆的内存空间;2)在 CNO 对象中管理本地对象的元数据也增加了内存消耗;3)Java 对象中指向 CAR 对象的指针也是额外的内存开销,但是由于构件编程的聚合特性,CAR 对象的继承链不会太长,所以指针所占开销十分有限.

在效率上,CNO 模型对本地方法调用在不同执行阶段的执行效率的影响不同,且对不同数据类型参数和返回值的本地方法的整体执行性能提升也有差异.考虑到使用复杂程序难以对这样的与数据类型有关的执行效率提升或下降的代码直接定位,所以本文在后面的实验中分别对 int,String,array 和 object 这 4 种典型的数据类型的本地方法的执行效率进行数据采集和分析.

CNO 模型对整个虚拟机系统和应用程序在不同运行阶段的性能影响也是不同的.比如在 CNO 模式下,在系统启动阶段、程序初始化阶段、第 1 次本地方法调用等阶段会产生额外时间开销.但是这些一次性开销所占时间是可接受的,在优化较复杂的程序时,相比在执行本地方法阶段的效率提升,这些时间开销一般可以忽略.

4.2 实验平台

实验系统平台搭建在基于 Intel® Core™ i3-3110M CPU 2.40 GHz 处理器和 2 GB 内存的 Ubuntu10.04 操作系统之上,原型系统实现基于 Gingerbread 和 Elastos3.0.实验程序运行在基于 Gingerbread 版本的 Android 虚拟设备模拟器(AVD emulator)之上,目标 API 是 level 10.

4.3 测试用例与实验方法

我们准备了以下 4 组测试用例:

1) 简单数据类型作为参数的本地方法调用,功能是进行 1 到 n 的累加和的数值计算,返回数值类型.本地方法原型为 public native int *Sum*(int n);

2) String 类型作为参数的本地方法调用,功能是将输入的两个字符串连接起来,返回 String 类型,本地方法原型为 public native String *Strcat*(String a,String b);

3) 数组类型作为参数的本地方法调用,功能是

两个数组相加得到一个新数组，返回数组类型，本地方法原型为 public native int[] *ArrayAdd*(int[]*a*, int[]*b*)；

4）用户自定义对象作为参数的本地方法调用，功能是在本地访问和处理输入对象参数中的成员，并构造一个新的用户自定义对象，返回用户自定义对象类型，本地方法原型为 public native MyObject *GetMyObject*(MyObject *obj*)；

对于每一个测试用例需要采集的实验数据包括 2 个时间段：1）Java 端从调用一个本地方法开始到本地方法结束后返回所消耗的时间；2）从进入本地代码开始到结束返回前所消耗的时间. 为了保证实验数据的有效性，我们对每个测试用例进行 6 组实验结果的统计. 这 6 组实验结果是对每个用例分别运行 100，200，400，600，800，1 000 次所得到的耗时均值. 为了与 JNI 技术进行比较，我们同样也基于 Gingerbread 和 Android NDK R8b 开发了相应的测试用例.

4.4 实验结果与分析

1）Java 端调用本地方法

表 2 分别列出了基于 JNI 方法和基于 CNO 方法实现的 4 个测试用例循环执行 100，200，400，600，800 和 1 000 次所使用的时间（单位为 ms）. 表格中的数值出现小数点是由于我们对每组循环都做了 20 次采样后取其平均值. 从结果来看，对于 *Sum* 这样以简单数据类型作为参数和返回值的本地方法，基于 JNI 方法的实现在 Java 端执行所用的时间与基于 CNO 方法的实现在 Java 端执行所用时间相差不大. 基于 CNO 方法的实现所用的时间要稍微多一点，但最大差距也不到 0.5 ms. 这是因为 JNI 方法对应 int 这样的简单数据类型不需要转换，其运算操作也非常简单；而 CNO 虽然也不需要对 int 这样的简单数据类型进行转换，但是在本地方法调用和返回时还要对其数据类型进行判断，所以执行时间稍有增加. 而对于 *ArrayAdd*，*Strcat* 这样以数组和字符串数据类型作为参数和返回值的本地方法，基于 CNO 方法的实现其执行时间相比 JNI 方法有明显的减少. 比如都是在循环 400 次的情况下，基于 JNI 方法的 *ArrayAdd* 用了 30.50 ms，而基于 CNO 方法的 *ArrayAdd* 只用了 13.92 ms. 执行时间优化更明显的是以对象为参数和返回值的 *GetMyObject* 函数. 在循环 1 000 次的情况下，基于 JNI 方法的 *GetMyObject* 用了 1 354.90 ms，而基于 CNO 方法的 *GetMyObject* 只用了 423 ms. 从以上数据可以看出，对于 String，array，Object 这样的复杂数据类型参数和返回值的本地方法，CNO 模型对它们的执行性能提升效果逐渐明显起来，这与我们提出 CNO 模型的优化策略所期望的结果是一致的.

Table 2 Elapsed Time of Calling Native Method in Java

表 2 Java 端调用 Native 方法所用时间 ms

Native Method	100 Loops		200 Loops		400 Loops		600 Loops		800 Loops		1 000 Loops	
	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO
Sum	0.20	0.21	0.38	0.75	1.00	1.10	1.22	1.24	1.71	1.78	1.80	1.85
ArrayAdd	7.85	0.66	17.00	8.44	30.50	13.92	46.50	28.77	53.30	41.92	86.10	78.60
Strcat	15.12	1.21	23.60	9.06	35.70	28.50	56.43	25.96	95.70	49.12	136.26	85.00
GetMyObject	148.10	15.70	259.12	36.82	477.37	91.40	873.03	172.02	1 139.78	309.60	1 354.90	423.00

2）在 C/C++中执行本地方法

我们进一步测试了 JNI 程序和 CNO 程序的 4 组测试用例在本地 C/C++端循环执行 100，200，400，600，800 和 1 000 次执行时所用时间. 表 3 给出了本地执行时间的实验结果，可以看出无论是 JNI 还是 CNO 实现的 *Sum* 方法其在本地的执行时间基本持平. 基于 CNO 方法实现的 *ArrayAdd*，*Strcat* 方法在本地执行时间只有相应 JNI 方法的本地执行时间的 1/2 左右. 基于 CNO 方法实现的 *GetMyObject* 方法循环执行 1 000 次只用了 53.42 ms，而 JNI 实现循环 1 000 次所用的 1 129.14 ms 是它的近 21 倍. 产生如此大差距的原因在于：在基于 CNO 模型实现的 *GetMyObject* 方法中，其参数所传递的对象本身也是一个 CNO 对象，在本地有其对应的 CAR 对象实现. *GetMyObject* 方法在访问该参数对象时，实际访问和操作的是 CAR 对象（也即是 C++对象），因此在 *GetMyObject* 方法中不需要使用 JNI 接口方法从参数对象中解析数据. 在返回对象时，*GetMyObject* 方法返回的也是一个 CNO 对象，不需要直接构造 Java 对象，所以本地执行时间才有了如此大的改善.

Table 3 Elapsed Time of Running Native Method in Native

表 3 在本地执行 Native 方法 ms

Native Method	100 Loops		200 Loops		400 Loops		600 Loops		800 Loops		1 000 Loops	
	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO	JNI	CNO
Sum	0.12	0.13	0.27	0.28	0.58	0.52	0.87	0.99	1.15	1.33	1.56	1.65
ArrayAdd	7.77	0.36	12.94	4.20	20.09	14.38	33.45	16.48	40.82	20.15	70.63	26.45
Strcat	6.16	0.40	17.02	4.82	33.84	16.28	54.84	20.26	98.93	24.19	127.34	29.77
GetMyObject	145.62	12.11	251.60	18.43	442.18	27.65	787.20	42.89	1054.20	51.34	1129.14	53.42

3）本地方法调用开销

表 4 为调用本地方法时的调用和返回过程所花的开销,通过 4 个本地方法在 Java 端和 C/C++端执行一次所花的时间均值的差得到. 这虽然不是一个精确的结果,但可以看出基于 CNO 模型的本地方法调用和返回所花的时间要比基于 JNI 方法所用时间要多一些. 前 3 组本地方法差距仅 0.8～5 μs,而 *GetMyObject* 方法的差距为 64 μs. 产生多付出的调用开销的原因在于我们需要在本地方法绑定时通过反射注入 CAR 对象元数据,并绑定 CAR 对象的相应方法. 与 CNO 本地方法在本地执行所获得的性能优化相比,这些开销是完全可以接受的.

Table 4 Average Elapsed Time and Invoking Cost of Native Method

表 4 Native 方法平均执行时间及调用开销 μs

Native Method	From Java		In Native		Invoking Cost	
	JNI	CNO	JNI	CNO	JNI	CNO
Sum	2.06	2.45	1.40	1.04	0.66	1.41
Array Add	78.33	43.76	61.66	23.28	16.67	20.48
Strcat	118.06	53.06	95.62	27.10	22.45	25.96
GetMyObject	1 367.45	277.72	1 233.09	79.40	134.36	198.32

4）集成第三方本地库的效率

一个 CNO 程序分为 Java 程序和 CAR 构件模块两部分,CAR 构件模块并没有像 JNI 程序的本地库那样被打包在 apk 中,而是在运行时动态装载、动态绑定. 这使得 CNO 程序在 CAR 构件接口定义保持不变的情况下,可以动态升级和替换本地 CAR 构件而不需要重新编译 Java 端代码. 这有利于 CNO 程序的移植.

在代码尺寸上,基于 CNO 模型开发本地调用程序与基于 JNI 方法开发本地调用程序相比,两者在 Java 端的代码基本是一样的. 开发者完全可以按照开发 JNI 程序的规范书写 CNO 程序 Java 端的代码. 两种开发方法主要差别在于本地程序的编写,JNI 程序要求开发者使用 JNI 规范实现本地方法或者实现调用本地库的桥接代码. CNO 程序要求开发者使用 CAR 构件技术直接开发本地构件,无需任何桥接代码. 所以在代码尺寸上,基于 CNO 模型开发的本地调用程序要更小一点.

在编译后的程序大小的比较上,基于 CNO 模型开发本地调用程序也有很好的表现. 本文实验所设计的 CNO 方法实现的例子程序,其编译后所得的 Java 端程序大小为 184 KB,CAR 构件模块编译后的大小为36.6 KB,两者相加共 220.6 KB. 而基于 JNI 方法开发的本地调用程序编译后的应用程序大小为 536 KB. 因此,CNO 程序相比 JNI 程序更加节省存储空间和程序运行时的内存空间,对开发移动智能设备上的应用更为有利.

5 总结及未来工作

针对如何提升本地调用程序的运行效率、减少本地调用程序尺寸和解决本地调用程序的跨平台问题,本文提出了基于跨语言构件对象迁移策略的 CNO 对象模型,探讨了跨语言构件对象迁移策略的主要思想和实现方法,并采用元数据注入的方法在 Dalvik 虚拟机内实现了 CNO 对象模型. 本文进行了基于 CNO 对象模型的本地调用程序运行效率的实验,并与 JNI 本地调用运行效率进行比较,结果反映出基于 CNO 对象模型开发本地调用程序可以有效提高程序运行效率,减小程序尺寸和可方便地跨

平台.当然，目前实现的基于 Dalvik 虚拟机的原型系统对 CNO 对象模型的支持还不够完善，仍然有大量值得关注和研究的问题，如支持成员变量元数据的注入、实现 CNO 对象模型的回调机制，以及基于 CNO 对象模型的调试方法和调试工具等问题，这些都是我们下一步的研究方向.

参 考 文 献

[1] Bornstein D. Dalvik VM internals [C] //Proc of the Google I/O Developer Conf. San Francisco: Google, 2008: 1-58

[2] Developers A. Android NDK [EB/OL]. [2013-08-01] http://developer.android.com/sdk/ndk/index.html

[3] Ratabouil S. Android NDK Beginner's Guide [M]. Birmingham: Packt Publishing, 2012: 5-22

[4] Liang S. The Java TM Native Interface: Programmer's Guide and Specification [M]. Reading, MA: Addison-Wesley, 1999: 6-23

[5] Gilmore S, Shkaravska O. Estimating the cost of native method calls for resource-bounded functional programming languages [J]. Electronic Notes in Theoretical Computer Science, 2006, 151(3): 27-45

[6] Lin Chengmin, Lin J H, Dow C R, et al. Benchmark dalvik and native code for android system [C] //Proc of the 2nd Int Conf on Innovations in Bio-inspired Computing and Applications. Piscataway, NJ: IEEE, 2011: 320-323

[7] Canfora G, Fasolino A R, Frattolillo G, et al. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures [J]. Journal of Systems and Software, 2008, 81(4): 463-80

[8] Chiang C C. Wrapping legacy systems for use in heterogeneous computing environments [J]. Information and Software Technology, 2001, 43(8): 497-507

[9] Bubak M, Kurzyniec D, Luszczek P. Convenient use of legacy software in Java with Janet package [J]. Future Generation Computer Systems, 2001, 17(8): 987-997

[10] Korsholm S, Jean P. The Java legacy interface [C] //Proc of the 5th Int Workshop on Java Technologies for Real-time and Embedded Systems. New York: ACM. 2007: 187-195

[11] Shi Xuelin, Zhang Zhaoqing, Wu Chenggang. Mapping Cobol data to Java type system with functional equivalence [J]. Journal of Computer Research and Development, 2006, 2(43): 336-342 (in Chinese)
(石学林，张兆庆，武成岗. Cobol 到 Java 翻译中的数据类型转换方法[J]. 计算机研究与发展，2006，43(2)：336-342)

[12] Ren Junwei, Lin Dongdai. Research of platform independent programming using JNI technology [J]. Application Research of Computers, 2005, 7(22): 180-184 (in Chinese)
(任俊伟，林东岱. JNI 技术实现跨平台开发的研究[J]. 计算机应用研究，2005，22(7)：180-184)

[13] Chen Hui, Chen Yiyun, Wu Ping, et al. A typed low-level language used in Java virtual machine [J]. Journal of Computer Research and Development, 2006, 1(43): 15-22 (in Chinese)
(陈晖，陈意云，吴萍，等. 一种用于 Java 虚拟机的类型化低级语言[J]. 计算机研究与发展，2006，43(1)：15-22)

[14] Newhall T, Miller B P. Performance measurement of dynamically compiled Java executions [C] //Proc of the 1999 ACM Conf on Java Grande. New York: ACM, 1999: 42-50

[15] Kim Y J, Cho S J, Kim K J, et al. Benchmarking Java application using JNI and native C application on Android [C] //Proc of the 12th Int Conf on Control, Automation and Systems (ICCAS). Piscataway, NJ: IEEE, 2012: 284-287

[16] Stepanian L, Brown A D, Kielstra A, et al. Inlining java native calls at runtime [C] //Proc of the 1st ACM/USENIX Int Conf on Virtual Execution Environments. New York: ACM. 2005: 121-131

[17] Lee Yannhang, Chandrian P, Li Bo. Efficient Java native interface for Android based mobile devices [C] //Proc of the 10th Int Conf on Trust, Security and Privacy in Computing and Communications. Piscataway, NJ: IEEE, 2011: 1202-1209

[18] Breg F, Polychronopoulos C D. Java virtual machine support for object serialization [J]. Concurrency and Computation Practice and Experience, 2003, 15(3/4/5): 263-275

[19] Ciacca F, Flamia S, Trevisti G. Process, apparatus and system for executing mhp applications: Germany, PCT/IT2005/000, 719 [P]. 2005-12-07

[20] Alder D. The Jacob project: A Java-COM bridge, Version 1.8, 1999-2004 [EB/OL]. (2004-08-17) [2013-05-21]. http://danadler.com/jacob/

[21] JNA-users Google group. Java native access (JNA) [EB/OL]. (2011-11-09) [2013-06-30]. https://java.net/projects/jna

[22] Jawin H S. An open source interoperability solution [EB/OL]. (2001-11-14)[2013-06-30]. http://www.onjava.com/pub/a/onjava/2001/11/14/jawin.html

[23] Beazley D M. SWIG: An easy to use tool for integrating scripting languages with C and C++[C] //Proc of the 4th USENIX Tcl/Tk Workshop. Berkeley: USENIX Association, 1996: 15-15

[24] Cottom T L. Using swig to bind C++ to python [J]. Computing in Science & Engineering, 2003, 5(2): 88-97

[25] Li J, Moore K. Enabling rapid feature deployment on embedded platforms with JeCOM bridge [G] //LNCS 3291: OTM Confederated Int Conf COOPIS, DOA, and ODBASE. Berlin: Springer, 2004: 1482-1501

[26] Hirzel M, Grimm R. Jeannie: Granting Java native interface developers their wishes [J]. Acm Sigplan Notices, 2007, 42(10): 19-38

[27] Chen Miaobo, Goldenberg S, Srinivas S, et al. Java JNI Bridge: A framework for mixed native ISA execution [C] //Proc of the 4th Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2006: 65-75

[28] IBM alphaWorks. Bridge2Java [EB/OL]. (2003-02-14) [2009-03-15]. http://www.alphaworks.ibm.com/tech/bridge2java

[29] Mutlu A, Ege M. Java-XCOM component integration on Linux operating system [C] //Proc of the 29th Conf on EUROMICRO. Piscataway, NJ: IEEE, 2003: 70-75

[30] Hummer W, Wolf W, Hahn C. Accessing COM-based applications from Java using WebServices [C] //Proc of the 1st Int Conf on Web Services. Athens, Georgia: CSREA Press, 2003: 487-490

[31] Fernande A. Microsoft Java Virtual Machine [M]. British Columbia, Canada: Spire, 2012: 10-88

[32] Li Rui, Li Yonggang. Study on calling COM automation components base on JACOB from Java [J]. Control & Automation, 2007, 15(23): 168-170 (in Chinese)
(李瑞, 李永刚. JAVA 中基于 JACOB 的 COM 组件调用研究[J]. 微计算机信息, 2007, 23(15): 168-170)

[33] Raj G S. A detailed comparison of CORBA, DCOM and Java/RMI [EB/OL]. 1998 [2013-05-17]. http://ece842.com/S13/readings/raj2003.pdf

[34] Obasanjo D. A comparison of Microsoft's C# programming language to Sun Microsystem's Java programming language [EB/OL]. 2002 [2013-05-17]. http://www.25hoursaday.com/CsharpVsJava.html

[35] Yang Xiangke, Chen Rong. Research of component device driver model based on Elastos [J]. Computer Technology and Development, 2008, 18(12): 25-31 (in Chinese)
(杨向科, 陈榕. 基于 Elastos 的构件化驱动编程模型的研究[J]. 计算机技术与发展, 2008, 18(12): 25-31)

[36] Jiang Zhanggai, Chen Rong. Native extension strategy of WebKit based on CAR components [J]. Journal of Computer Applications, 2009, 29(z2): 195-197 (in Chinese)
(蒋章概, 陈榕. 基于 CAR 构件的 WebKit 本地扩展策略[J]. 计算机应用, 2009, 29(增刊 2): 195-197)

[37] Zheng Wei, Chen Rong, Su Yipeng, et al. CAR component oriented programming and its self-description property [J]. Computer Engineering and Applications, 2005, 9(41): 95-98 (in Chinese)
(郑炜, 陈榕, 苏翼鹏, 等. CAR 构件编程技术中的自描述特性[J]. 计算机工程与应用, 2005, 41(9): 95-98)

[38] He Jianli, Chen Rong, Gu Weinan. A new method for component reuse [C] //Proc of the 2nd Int Conf on Computer Science and Information Technology. Piscataway, NJ: IEEE, 2009: 304-307

[39] Cunningham H, Maynard D, Tablan, V. JAPE: A Java annotation patterns engine, CS-00-10 [R]. South Yorkshire, England: Department of Computer Science, University of Sheffield, 2000

Hang Yukun, born in 1978. PhD candidate in Tongji University. Student member of China Computer Federation. Her main research interests include embedded operating system, mobile computing, component oriented technology.

Chen Rong, born in 1957. Professor and PhD supervisor in Tongji University. Member of China Computer Federation. His main research interests include embedded operating system, component oriented technology (chen.rong@kortide.com).

Pei Xilong, born in 1967. Engineer in Tongji University. His main research interests include operating system, software engineering (Pei_xilong@tongji.edu.cn).

Cao Jing, born in 1980. PhD of the Southeast University. His main research interests include programming languages, program analysis (chao.jing@kortide.com).