同济大学
TONGJI UNIVERSITY

硕士学位论文

基于构件技术的新型 Java 虚拟机本地调用机制的研究

（智能手机嵌入式软件平台研发及产业化，课题编号：2009ZX01039-002-002）

姓　　名：王建民

学　　号：0820080232

所在院系：电子信息与工程学院计算机系

学科门类：工学

学科专业：计算机软件与理论

指导教师：陈榕

副指导教师：顾伟楠

二〇一一年五月

同濟大學
TONGJI UNIVERSITY

A dissertation submitted to

Tongji University in conformity with the requirements for

the degree of Master of Engineering

The Research of New Native Calling Mechanism of Java Virtual Machine Based on Component Technology

Candidate: Jianmin Wang

Student Number: 0820080232

School/Department: School of Electronics and Information Engineering

Discipline: Engineering

Major: Computer Software and Theory

Supervisor: Rong Chen

May, 2011

基于构件技术的新型JAVA虚拟机本地调用机制的研究

王建民　同济大学

学位论文版权使用授权书

本人完全了解同济大学关于收集、保存、使用学位论文的规定，同意如下各项内容：按照学校要求提交学位论文的印刷本和电子版本；学校有权保存学位论文的印刷本和电子版，并采用影印、缩印、扫描、数字化或其它手段保存论文；学校有权提供目录检索以及提供本学位论文全文或者部分的阅览服务；学校有权按有关规定向国家有关部门或者机构送交论文的复印件和电子版；在不以赢利为目的的前提下，学校可以适当复制论文的部分或全部内容用于学术活动。

学位论文作者签名：

年　　月　　日

同济大学学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或者没有公开发表的作品的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：

年　　月　　日

摘要

近几年来，随着互联网和移动设备的快速发展，智能手机已逐渐成为人们身边必不可少的信息获取、交流的工具，也成为软件行业新的发展热点。在众多智能手机平台中，Android 手机平台由于其全面的开放性取得了快速的发展，逐渐成为全球最大的手机平台。Android 系统的应用框架与 GUI 库都是采用 Java 语言实现，程序开发语言也以 Java 语言为主。因此，如何进一步提高 Java 语言的执行效率对 Android 手机平台的发展具有重要意义。

CAR（Component Assembly Runtime）构件技术是面向构件的编程模型，它表现为一组编程规范，规定了构件间相互调用的标准，使得二进制构件能够自描述，能够在运行时动态链接。CAR 构件的运行环境是 Elastos 中间件平台，具有跨平台的能力，可以运行于包括 Android 在内的多个移动平台。CAR 构件具有与本地代码相当的执行效率，并且具备良好的跨平台性。因此在构件技术的前提下，如果用一个 CAR 构件代替某个 Java 构件，并且保证整个程序的完整性、不影响其他构件的执行、调用，那么可以使得该构件具有与本地代码相当的执行效率。因此本文提出了基于构件技术的 Java 虚拟机本地调用机制来实现这种方式。

本文首先介绍了 Java 虚拟机本地调用机制的工作原理，分析了目前虚拟机调用本地功能机制的优缺点，提出了以构件技术为基础的 JNC（Java Native Component，Java 本地构件）机制，作为实现 CAR 构件替换 Java 构件的基础。本文还探讨了 Java 本地构件机制的设计目标，研究了构件加载机制、生命周期管理和相互调用机制。最后，本文提出了 Elastos 平台上的 Java 本地构件机制的实现，并进一步探讨了 JNC 机制在 Elastos 平台中担任的角色。

关键字：CAR 构件，中间件，构件技术，虚拟机，Java

ABSTRACT

With the rapid development of Internet and mobile devices in recent years, such as iPhone, iPad and Android Phone, smart phone plays an indispensable role in our life, which gets a lot of attention recently in the software industry. Among lots of smart phone runtime platform, Android platform has made rapid development due to its full openness, and become largest smart phone platform in the world. Java program language in Android is used to implement application frameworks and GUI library, and is executed in a faster virtual machine, named Dalvik. However, the performance of Java language still has potential to improve.

CAR (Component Assembly Runtime) technology is a component-oriented programming model, appearing a set of programming rules and criterion of components calling each other. With self-describing information, CAR component can be loaded dynamically while running. Elastos middleware is the runtime environment of component, which can run on kinds of systems, including Windows, Linux, WinCE, Android, Symbian and so on. CAR component has about the same performance as native code, and the character of cross platform, so it is a effective way to improve performance of Java language that CAR component replace Java component to implement the same function.

The principle of Java virtual machine and its native calling function is introduces firstly, and then the strengths and weaknesses of native function calling mechanism is analyzed in current Java virtual machine. A new mechanism named Java Native Component, JNC for short, is presented to implement that Java components can be substituted by CAR components. The purpose of JNC mechanism desine is discussed in this paper, and the lifecycle management and interaction mechanism of components in JNC also is discussed. Finally, The design and implement of JNC mechanism on Elastos Middleware is proposed, and the role of JNC mechanism is probed into further in Elastos platform.

Key Words: CAR component, Middleware, Component Technology, Virtual Machine, Java

目录

第 1 章 引言

1.1 背景与研究意义

自从手机进入人们的生活以来，尤其是近几年智能手机的快速发展，手机逐渐成为人们新的获取信息的方式。而且这几年互联网的快速发展，社交网络、微博、团购等网络新事物的快速兴起，使得人们的生活越来越离不开互联网。方便智能的手机与互联网结合形成的移动互联网，满足了人们随时随地获取网络信息的需求。因此，智能手机平台即移动终端平台也逐渐成为软件行业、乃至互联网行业新的热点。

不同于个人电脑平台的单一性，移动终端平台在 iPhone 手机、Android 手机的相继问世后开始向多样化发展，目前市场中包括 Windows Mobile(Windows Phone 7)、iOS、Android、Symbian、BlackBerry、WebOS 等各种平台。由于平台之间差异巨大，促使程序开发者对跨平台的需求极为强烈。而随着平台开发规模的增加，移动终端平台对通过构件技术降低软件成本的需求也越来越明显。无论是对服务器软件、终端软件还是嵌入式开发，构件技术一直是软件工程中降低开发成本、提高开发效率的重要途径。

Elastos 中间件平台是一个构件运行平台，它支撑着 CAR（Component Assembly Runtime）[1]构件技术。CAR 构件技术是总结了面向对象和面向构件编程技术的基础上，为了支持下一代网络应用软件的开发而发明的。Elastos 的功能模块及其支撑的软件都是采用 CAR 构件进行拼装的，CAR 构件就像工厂里生产的零部件，其目的就是实现软件的工厂化生产。CAR 构件内部采用 C/C++编写，携带元数据信息，元数据通过反射机制参与构件组装计算，生成的代码直接以目标平台的二进制代码运行，能够达到 C/C++的运行效率。

Java 语言[33]是一种面向对象的、跨平台、安全性高的开发语言，目前也是程序员数目最多的一种编程语言。谷歌推广的开源工程的 Google Web Toolkit（GWT）[21]就是通过 Java 语言快速构建和维护复杂而又高性能的 JavaScript 前端应用程序，从而降低了开发难度，这也体现的 Java 语言作为开发语言的优势。Java 语言具有良好的跨平台性能，无论是大型机、还是 ARM 处理器，都有 Java 虚拟机的存在。通过 OSGi 框架[22]的支撑，Java 构件技术充分的发挥了 Java 语言的动态性，而 OSGi 框架本身也开始成为 Java 规范的一部分。

因此，无论是对于 Elastos 平台的开发还是 CAR 构件技术的推广，本文的研究都具有十分重要的意义。基于本文的研究，促使 Java 语言成为 Elastos 平台程

序开发语言之一，有利于 Elastos 平台程序开发者的工作；完善了 Java 虚拟机构件化的运行；完善了 CAR 构件技术对 Java 语言的支持，对 CAR 构件与 Java 语言构件之间的相互调用也提供了可实行的方案；通过对 Java 语言与 CAR 构件的深入分析、交互，可以进行发现两种成熟环境在不同方面的对比，有利于 CAR 构件技术的自身完善。

1.2 国内外研究现状

尽管针对于 Java 构件与 CAR 构件相互调用国内外还没有较多的研究工作，但对于 Java 语言与本地语言进行交互的工作除了 Java 规范里提到的 JNI[3]（Java Native Interface）机制，还有很多开源工程尝试针对 JNI 机制进行改进，如 JNA[6]（Java Native Access）、JNative[7]、NativeCall[8]、JNIWrapper 等，这些方案中有些比较成熟，有一些还只是尝试，但都各有其特点。在此对这些方案进行简单介绍。

1.2.1 JNI（Java Native Interface）

JNI 方式是 Java 规范中对 Java 使用本地代码规定的一种调用方式。基本原理是将 Java 类中的具有 native 属性的方法，与加载的本地库（dll 文件或 so 文件）中相应的方法加载时进行绑定，以便在此 native 方式执行时自动跳转至本地方法继续执行，并对方法参数和返回值进行相应的参数转换。

JNI 机制使得开发者一方面可以充分利用 Java 平台提供的功能，另一方面也可以使用其他语言编写的代码。作为 Java 虚拟机实现的一部分，JNI 提供了双向的操作，既允许 Java 代码调用本地代码，也使得本地代码可以对 Java 虚拟机进行操作。图 1.1 表示了 JNI 在 Java 虚拟机中的角色[4]。

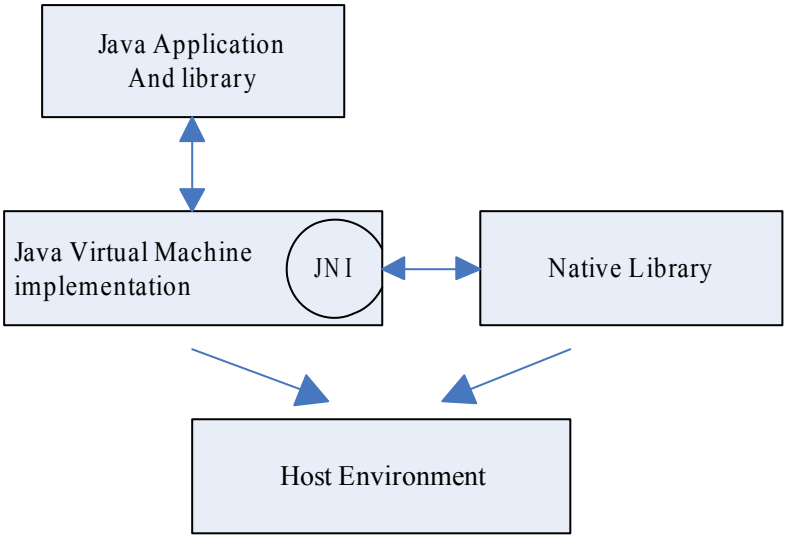

图 1.1 虚拟机中 JNI 的角色

这种方式的优势是对所有 Java 虚拟机都使用，而缺点是使用十分麻烦，需要本地方法与类方法名称相对应，并且不具有动态加载、卸载的能力，对于参数转换要求较高，需要额外生成头文件、编译操作，不适合应用到构件的动态的相互调用。

1.2.2 JNA（Java Native Access）

JNA 工程是对 JNI 方式的改进，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 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 库使用一个名为 libffi 的小型的本地库来动态的调用本地代码，JNA 库允许动态的加载本地库并将方法指针进行绑定，使用 libffi 库进行方法调用，而不需要 JNI 方式中的静态绑定、头文件等工作，并且提供了 Java 接口使得在 Java 程序中可以使用 C 代码中常用的结构体、指针的能力[6]。

JNA 的缺点在于需要对要加载的动态链接库文件进行封装后才能正常装载，而且需要在 Java 接口中提供对动态链接库文件中的函数与结构的描述，才能实现 Java 接口与 native function 的映射，不够便捷。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，这也使得 JNA 技术比 JNI 技术调用动态链接库会有些微的性能损失，甚至可能速度会降低几倍。

1.2.3 JNative

JNative 相比于 JNA 工程而言，是一个比较小的开源工程，实现的功能也不足，其主要特点是对结构体映射、函数调用以及回调的支持，并且支持多线程操

作。

与 JNA 工程相同的是都采用专门的指针类来映射本地代码的指针操作。结构体采用 AbstractBasicData 抽象类来表示，其映射基本思想是通过已知结构占用内存的大小以及变量各自占用内存的大小，以及默认提供的基本数据类型的转换来实现对结构体数据的获取与设置。使用方式是定义继承自 AbstractBasicData 的子类，并且增加与本地代码中结构体相同的变量，实现包括 getValueFromPointer()、getSizeOf()、createPointer()等方法，并且需要自己实现针对结构体变量的获取器、设置器。

总体来说，相比于 JNA 需要额外封装动态链接库文件，JNative 的机制更为便捷，只需要 Java 端的封装。但需要实现的方法过多，只适用于简单程序实现，不便于应用到大量的开发工作中。

1.2.4 NativeCall

不同于 JNA 项目和 JNI 项目的目标是对 JNI 机制的改进更便捷的使用本地代码，NativeCall 项目的目标是不需要开发者编写本地代码而调用系统函数。当前 NativeCall 只支持 Windows XP 系统。

1.3 论文主要工作

论文的主要内容是对基于 Elastos 平台的 Java 虚拟机的构件化工作，包括构件化的设计以及构件化的运行。主要研究内容包括：

1. 研究并提出 CAR 构件与 Java 构件的相似性，以及如何改造 Java 虚拟机促使两种构件的等价，即 JNC（Java Native Component，Java 本地构件）机制；
2. 研究如何采用 JNC 技术对 Java 虚拟机基础类库实现构件化；
3. 分析跨语言构件相互继承的机制；
4. 探讨 Elastos 平台中 Java 构件的角色；

1.4 章节安排

本论文主要包括六章内容，各部分主要内容如下：

第一章引言。简要介绍了选题的背景和意义，所用技术的研究和发展现状，以及论文的主要内容。

第二章相关技术介绍。主要包括论文涉及的构件技术、CAR 构件技术、Java

虚拟机以及 OSGi 框架的介绍。

第三章虚拟机设计。主要是分析设计了 JNC 机制以及基于 JNC 技术的构件化的 Java 虚拟机，分析了 JNC 机制中跨语言构件之间的继承关系。

第四章虚拟机构件化实现。探讨了 JNC 机制的具体实现以及基础类库构件化实现的相关问题，还探讨了 JNC 机制以及 Java 语言在 Elastos 平台中的角色。

第五章总结与展望。总结了本文研究成果，并对以后工作进行了展望。

第2章 相关技术介绍

2.1 CAR构件技术

CAR（即Component Assembly Runtime）构件技术是新一代面向构件的编程模型，是基于“软件即零件”的编程思想而产生，其定义是“表现为一组编程规范，规定了构件间相互调用的标准，包括构件、类、对象、接口等定义与访问构件对象的规定，使得二进制构件能够自描述，能够在运行时动态链接”[13]。从其字面意思来讲，“CAR就是在运行时对软件构件进行组装并最终完成预计功能的一种软件技术”[13]。在CAR构件技术中，构件就是零件，零件与零件组装而形成大的部件，部件加上外壳即成为了产品，这也是CAR构件技术的基本思想。

程序开发经历了从面向结构编程、面向对象编程到面向构件编程、面向服务编程的发展。面向结构编程是通过函数实现了代码的服用。面向对象程序设计思想是采用对象来表示程序的各个部分，并通过对象间的继承和多态提高软件的复用性。通过对软件模块的封装，使其相对独立，从而使复杂的问题简单化。面向对象程序设计强调的是对象的封装，但模块（对象）之间的关系在编译的时候被固定，模块之间的关系是静态的，在程序运行时不可改变模块之间的关系，就是说在运行时不能换用零件[16]。

CAR构件技术通过二进制的封装以及动态链接技术解决软件的动态升级和软件的动态替换问题。面向构件技术对一组类的组合进行封装，并代表完成一个或多个功能的特定服务，同时为用户提供多个接口。整个构件隐藏了具体的实现，只用接口提供服务。

CAR构件技术主要解决的问题有：不同来源的构件实现互操作、构件升级不会影响其他的构件、构件独立于编程语言、构件运行环境的透明性。

2.1.1 CAR发展简史

CAR技术在很大程度上借鉴了微软COM（Component Object Model，构件对象模型）的思想，最初CAR是兼容COM的，但是和COM相比，CAR删除了COM中过时的约定，禁止用户定义COM的非自描述接口；完备了构件及其接口的自描述功能，实现了对COM的扩展；对COM的用户界面进行了简化包装，可以说CAR是微软COM的一个子集，同时又对微软的COM进行了扩展，在Elastos SDK工具的支持下，使得高深难懂的构件编程技术很容易被C/C++程序员理解并掌

握，因此最初称之为ezCOM，其中“ez”源自与英文单词“easy”，恰如其分地反映了这一特点。首先编写一个.cdl文件，CDL即构件定义语言，对应于微软的IDL（接口定义语言），然后将它转换成微软的.idl文件，最后用MIDL（微软的IDL编译器）进行编译生成相应的代码。

目前的CAR技术已经不再保持与COM兼容，也不再使用微软的MIDL编译器，而是使用自己的工具carc，lube和cppvan。例如，首先编写一个.car文件，定义构件模块中的构件类、接口以及接口方法等信息，使用emake命令，实际上是调用编译器carc.exe，编译生成代码框架，填写完实现代码后再使用z命令编译生成.dll文件，该文件的资源段中包含了元数据，而COM是没有的，这个编译生成代码框架的过程以及如何填写实现代码在后面的章节中有详细的说明与示例。利用Elastos IDE工具将使得用户对这些复杂问题的把握变得容易。

CAR技术在发展过程中也在一直变化着，例如对于创建对象，ObjInstantiate、New、EzCreateObject、EzCreateInstance和NewInContext都是创建对象时使用的。其中EzCreateInstance已经不再使用了；EzCreateObject就是原来的EzCreateInstance，现在用EzCreateObject以指定的CLSID创建一个未初始化的类对象；EzCreateObjectEx用于在远程的机器上创建一个指定类的对象；ObjInstantiate原来叫做Instantiate，最早叫NEW_COMPONENT。New其实是个简化版的EzCreateObject，简化了CLSID，InterfaceId，DomainInfo。并且通过重载，允许使用带有多个参数的New创建对象。该方法用来在同一Domain创建一个构件对象，实际上是对EzCreateObject的一层封装，现在开发人员应当尽量避免使用EzCreateObject，而用New方法来创建一个对象；NewInContext方法也是用来创建一个构件对象，但用户可以指定该构件对象的语境，即context由参数pDomainInfo来指定，现在已经实现的CAR支持pDomainInfo的值为：CTX_SAME_DOMAIN, CTX_DIFF_DOMAIN。

2.1.2 CAR 构件示例

一个 CAR 构件中可能包含有类、接口、结构体、枚举类型、结构体以及常量等，图 2.1 是 CAR 构件的示意图。

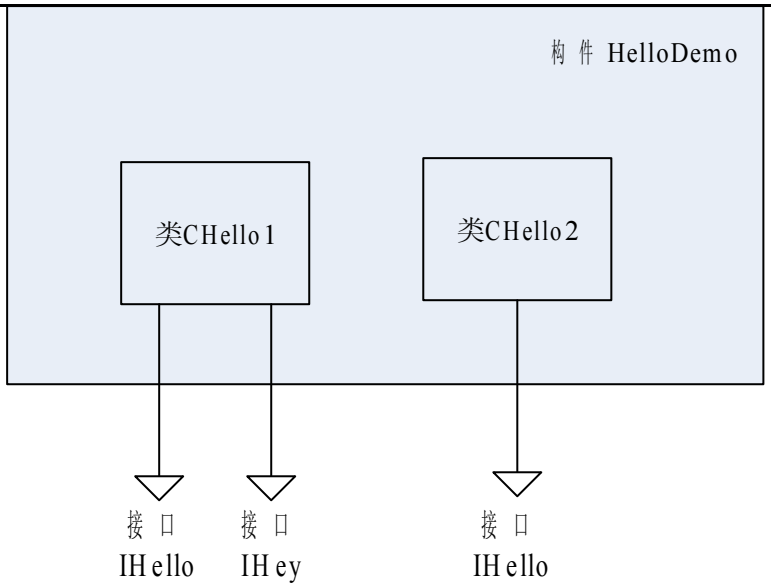

图 2.1 构件 HelloDemo 示意图

构件 HelloDemo 的示例定义如下：

```
module              //构件 HelloDemo.car
{
    //接口 IHello
    interface IHello {
        Hello([in] Int32 i);    //方法
    }

    //接口 IHey
    interface IHey {
        Hey([in] Int32 i, [out] AString* char);
    }

   //类 CHello1
    class CHello1 {
        interface IHello;
        interface IHey;
    }

   //类 CHello2
    class CHello2 {
        interface IHello;
    }
}
```

2.1.3 CAR 自描述数据

CAR 构件技术最初的设计借鉴了微软的 COM（Component Object Model，构件对象模型）[24]的思想，并且在最初设计时保持了对 COM 的兼容，但随着 CAR 技术本身的发展，逐渐放弃了对 COM 的兼容，而更关注如何独立地完善本身设计。

与 COM 一个很大的不同就是 CAR 构件都具有自描述数据[18]，即“元数据”（metadata）。COM 构件虽然也提供了元数据，但同时也支持不包含元数据的构件，并且 COM 构件的元数据是保存在系统所在的注册表中，而非构件本身中，这就丢失了元数据本来的含义。

CAR 构件的元数据是 CAR 文件经过 CAR 编译器生成的，元数据与构件的实现代码一起被打包到构件模块文件中。元数据记录了构件接口及构件类的定义信息，是构件自描述的基础。

2.1.3.1 自描述数据类型的必要性

在传统的应用编程习惯中，编程者如果需要声明一个存储 1000 个字节的缓存空间，通常就简单定义为：

```
#define BUFLENGTH 1000
Byte    buf[BUFLENGTH];
```

开发者在使用该缓存空间时，通常关心的是 buf 中实际参与计算的内容，却很少注意 buf 的自我描述性。在网络计算中，一个没有特征的数据可能增加服务的不必要的负担。对于上面的例子来说，该 buf 所带信息太少。在将这段数据传递给某个远程服务接口的方法时，为防止内存溢出，必须附带上 buf 的容量。例如：

```
void foo (
        Byte    *pBuf,
        Int32    capacity     );
```

如果这部分内存 buf 有部分内容正在被其他服务使用，而在当前服务中又不希望被覆盖，那么接口方法声明时还需要加入关于已经使用的参数进行描述：

```
void foo (
        Byte    *pBuf,
        Int32    capacity,
    Int32    used     );
```

其中 used 参数表示使用了的字节。我们并不认为这种接口方法的定义是成功的，因为让服务端花费多余的处理来识别后两个参数是资源的一种浪费。而出现这种接口方法的定义，主要原因在于传统的操作系统对于这种常见的参数传递习惯没有定义一种合适的数据类型来处理它。尤其在面向网络的应用程序中，数据应该是自描述的。

2.1.3.2 CAR 语言

CAR 构件编写者在写一个自己的构件时，第一件事情就是要写一个 CAR 文件，构件编写者使用 CAR 语言在这个文件里描述自己的构件接口，接口方法以及实现接口的构件类等等，CAR 工具会根据这个 CAR 文件为构件编写者生成代码框架以及其他构件运行时所需要的代码，构件编写者只需关心自己接口方法的实现。

下面请看 hello 构件的 CAR 文件（hello.car）的内容：

```
//hello.car 文件
module
{
    interface IHello {
        Hello();
    }

    class CHello {
        interface IHello;
    }
}
```

上面的 CAR 文件描述了 hello 构件模块有一个构件类 CHello， 一个接口 IHello，接口方法有 Hello(), 这个方法没有参数，构件类实现了 IHello 等等信息。

类似于微软的 ODL（Object Definition Language）文件，CAR 文件描述了一个构件里所包含的构件类的组织信息（如构件类的排列顺序、包含的接口）、各个接口的信息（如接口的种类、包含的接口方法）、各个方法的信息（如接口方法参数的种类、排列顺序等）以及接口和构件对象的标识等信息。

有关 CAR 构件描述语言(也就是 CAR 语言)的具体介绍，请参看第四章到第九章相关内容。

2.1.3.3 CAR 构件元数据

元数据(metadata)，是描述数据的数据(data about data)，首先元数据是一种数据,是对数据的抽象，它主要描述了数据的类型信息。

普通的源文件（c 或者 c++语言）经过编译器的编译产生二进制的文件，但在编译时编译器只提取了 CPU 执行所需的信息，忽略了数据的类型信息。比如一个指针，单看编译完之后的二进制代码或汇编已不能区分它是整型或是 char 型了，如果是指向字符串的指针，字符串的长度也无从知晓。这部分类型信息就属于我们所说的元数据信息。

CAR 构件以接口方式向外提供服务，构件接口需要元数据来描述才能让其他使用构件服务的用户使用。构件为了让接口与实现无关，从而保持了接口的不变性，使得动态升级成为可能；并且使用 vptr 结构将接口的内部实现隐藏起来,

由接口的元数据来描述接口的函数布局和函数参数属性。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不同构件之间的调用才成为可能，构件的远程化，进程间通讯，自动生成 Proxy 和 Stub 及自动 Marshalling、Unmarshalling 才能正确地进行。

CAR 构件的元数据是 CAR 文件经过 CAR 编译器生成的，元数据与构件的实现代码一起被打包到构件模块文件中。元数据记录了构件接口及构件类的定义信息，是构件自描述的基础。

在 CAR 里，ClassInfo 被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。同时 ClassInfo 也是自动生成构件源程序的基础。

在目前的 CAR 构件开发环境下 ClassInfo 以两种形式存在：一种是与构件的实现代码一起被打包到构件模块文件中，用于列集和散集用的；另一种是以单独的文件形式存在，存放在目标目录中，最终会被打包到 DLL 的资源段里，该文件的后缀名为 cls，如 hello.car 将会生成 hello.cls。这个 cls 文件和前者相比，就是它详细描述了构件的各种信息，而前者是一个简化了 ClassInfo，如它没有接口和方法名称等信息。cls 文件就是 CAR 文件的二进制版本。由于前者只是用于 CAR 构件库的实现，接下来我们要介绍的是后一种 ClassInfo, 这是用户需要关心的。

CAR 构件的自描述信息主要包含类信息(ClassInfo)和导入信息(ImportInfo)两种。ClassInfo 被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；与此相反，ImportInfo 则描述该构件运行时需要用到的别的服务性构件的信息。

2.1.3.4 ClassInfo 构成

对于每个 CAR 构件模块，ClassInfo 主要包括三大部分：构件模块信息、所有的构件类信息以及所有的接口信息。

我们可通过如图 2.2 说明构件模块信息的主要构成：

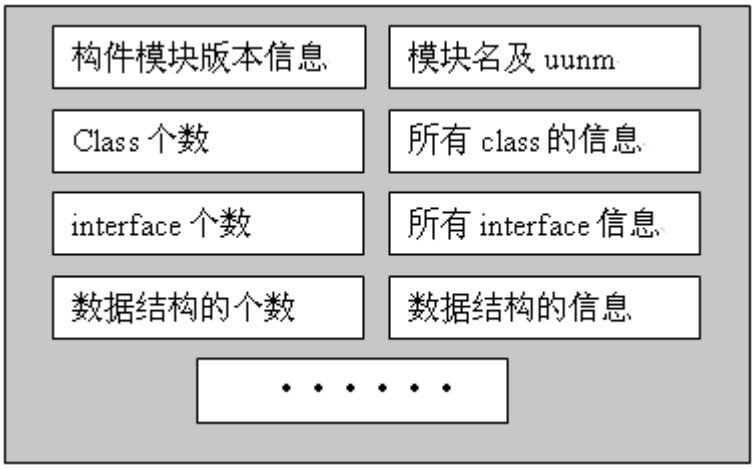

图 2.2 模块信息主要构成

对于每个构件类信息构成，可如图 2.3 实例：

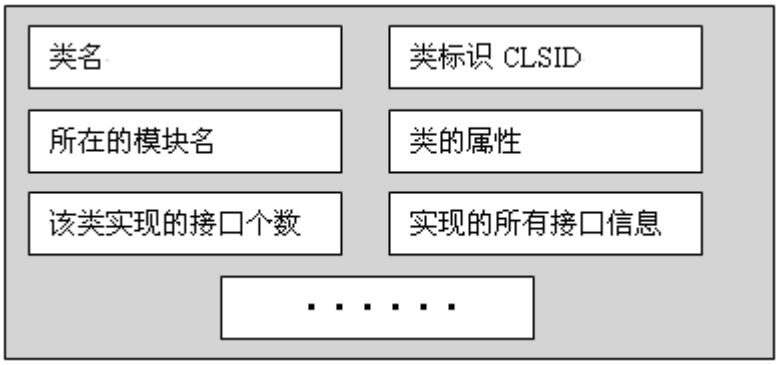

图 2.3 类信息主要构成

这里要注意的是，类信息里有所在的模块名，这是因为该类可能是其它模块的构件类。

对于每个接口信息，其主要构成如图 2.4 所示：

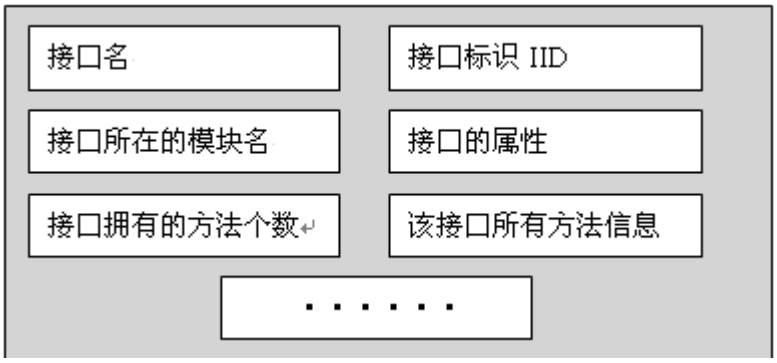

图 2.3 接口信息主要构成

接口也可以是另外模块定义，所以接口信息也记录了接口所在的模块名。

其中每个方法的信息结构如图 2.5 所示：

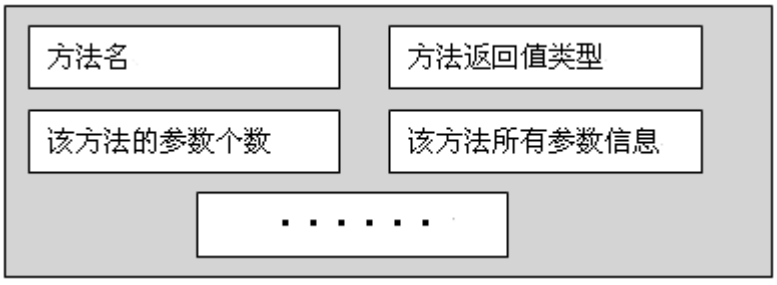

图 2.5 方法信息主要构成

对于方法的每个参数有参数名及参数属性等主要构成。参数属性描述了该参数是输入参数还是输出参数或者是否为输入输出参数。

另外需要说明的是，每个接口的方法信息不包括 IObject 方法的信息，因为它是所有接口的基接口，没有必要包含在每个接口信息里面。

2.1.3.5 自描述数据类型在 CAR 构件开发中的重要性

C/C++定义的标准数据类型中的只有一部分符合自描述标准，但上面列出的表格里的 CAR 基本数据类型都属于自描述数据类型。

基础自描述数据类型在传统开发中并不能很好的体现它的优势，因为在传统的单道程序或“客户/服务器”（C/S）二层体系结构设计中，对数据是否自描述没有太多要求，它可以通过用户的自我约定及额外的参数传递来解决这个问题，而且对于二层体系结构来说它在资源上的消耗是微乎其微的。

但在网络技术迅猛发展的今天，“客户/中间件/服务器”三层乃至所谓的多层体系结构、中间件技术、Grid 网络计算等新概念新技术层出不穷，传统的操作系统已不能很好的适应 WEB 服务的要求，而基于构件技术的 Elastos 正是为适应这种新形式而研发出来的新一代操作系统。我们知道，在中间件的应用开发中，构件接口参数的列集（Marshaling）和散集（UnMarshaling）起着关键性的作用，除了整型和布尔型这类的简单类型能被顺利处理外，其他部分的复杂类型则将消耗系统的很大一部分资源用于处理传递参数的列集和散集。而定义出一套基础自描述数据类型将使我们在以下方面获利：可以通过有限的参数传递，得到理想的数据信息；能有效地降低服务构件的负载，并能快速响应客户构件的应用请求；能有效地减少数据的二义性，避免发生人为的不必要的计算错误；满足构件兼容性的要求。

2.1.4 CAR 构件的反射调用

正如 2.1.2 节所介绍的，由于 CAR 构件自身带有“说明书”，因此可以通过反射机制动态获取 CAR 构件的信息。

对于一个已生成的 CAR 构件 HelloDemo.dll，可通过以下代码实现构件的动态加载、对象的动态创建以及方法的动态调用：：

```
    IHello * pHello = NULL;                          /* 定义接口指针，IHello 在构件 RefTest
中提供实现 */
    IModuleInfo * pModuleInfo = NULL;               /* 定义构件接口 */
    IClassInfo * pClassInfo = NULL;                 /* 定义类接口 */
    PObject pObj = NULL;                            /* 定义类实例指针 */
    IMethodInfo * pMethodInfo = NULL;               /* 定义方法接口 */
    IArgumentList * pArgumentList = NULL;           /* 定义方法参数列表接口 */
```

```
    ECode ec;                                          /* 定义函数返回值变量 */

    ec = _CModuleInfo_Acquire(L"RefTest.dll", &pModuleInfo);  /*查询构件 RefTest*/
    if (FAILED(ec))                                        /* 查询失败的情况 */
        goto Exit;

    ec = pModuleInfo->GetClassInfo("CHello", &pClassInfo);     /* 得到构件中的类 Chello
接口*/
    if (FAILED(ec))                                       /* GetClassInfo 执行失败的
情况 */
        goto Exit;

    ec = pClassInfo->CreateObject(&pObj);                  /* 产生类 Chello 实例 */
    if (FAILED(ec))                                        /* CreateObject 执行失败的
情况 */
        goto Exit;

    ec = pClassInfo->GetMethodInfo("test", &pMethodInfo);     /* 得到方法 test 的接口*/
    if (FAILED(ec))                                       /* GetMethodInfo 执行失败的
情况 */
        goto Exit;

    ec = pMethodInfo->CreateArgumentList(&pArgumentList);    /* 生成方法列表*/
    if (FAILED(ec))                                    /* CreateArgumentList 执行失败
的情况 */
        goto Exit;

    ec = pArgumentList->SetInputArgumentOfInt32(0, 100);      /* 根据序号设置参数值*/
    if (FAILED(ec))                                  /* SetInputArgumentOfInt32 执行失
败的情况 */
        goto Exit;

    ec = pMethodInfo->Invoke(pObj, pArgumentList);             /* 完成动态调用 test 方法
*/
    if (FAILED(ec))                                        /* Invoke 执行失败的情况
*/
        goto Exit;
```

因此，可以通过对虚拟机进行改造，实现对于其他构件（无论是 Java 构件还是 CAR 构件）在依赖某个 CAR 构件时进行动态的加载和执行。下一节将介绍如何动态判断需要加载的 CAR 构件。

2.1.5 CAR 技术对软件工程的作用

CAR的重要特点就是上文所介绍的：构件的相互操作性；软件升级的独立性；编程语言的独立性；进程运行透明度。

对于软件开发企业来说，采用CAR构件技术具有如下几方面的重要意义：

（1）CAR的开发工具自动实现构件的封装，简化了构件编程的复杂性，有利于构件化编程技术的推广；

（2）CAR构件技术是一个实现软件工厂化生产的先进技术，可以大大提升企业的开发技术水平，提高软件生产的效率和软件产品的质量；

（3）CAR构件技术为建立软件工厂化生产的软件标准提供了参考，有利于建立企业内、行业内的软件标准和构件库。

在实际的编程应用中，CAR 技术可以使程序员在以下几个方面得到受益：

1、 易学易用

基于 COM 的构件化编程技术是大型软件工程化开发的重要手段。微软 Windows 2000 的软件全部是用 COM 实现的。但是微软 COM 的繁琐的构件描述体系令人望而生畏。CAR 的开发环境 ElastosSDK 提供了结构简洁的构件描述语言和自动生成辅助工具等，使得 C++程序员可以很快地掌握 CAR 编程技术。

2、 可以动态加载构件

在网络时代，软件构件就相当于零件，零件可以随时装配。CAR 技术实现了构件动态加载，使用户可以随时从网络得到最新功能的构件。

3、 采用第三方软件丰富系统功能

CAR 技术的软件互操作性，保证了系统开发人员可以利用第三方开发的，符合 CAR 规范的构件，共享软件资源，缩短产品开发周期。同时用户也可以通过动态加载第三方软件扩展系统的功能。

4、 软件复用

软件复用是软件工程长期追求的目标，CAR 技术提供了构件的标准，二进制构件可以被不同的应用程序使用，使软件构件真正能够成为"工业零件"。充分利用"久经考验"的软件零件，避免重复性开发，是提高软件生产效率和软件产品质量的关键。

5、 系统升级

传统软件的系统升级是一个令软件系统管理员头痛的工程问题，一个大型软件系统常常是"牵一发而动全身"，单个功能的升级可能会导致整个系统需要重新调试。CAR 技术的软件升级独立性，可以圆满地解决系统升级问题，个别构件的更新不会影响整个系统。

6、 实现软件工厂化生产

上述几个特点，都是软件零件工厂化生产的必要条件。构件化软件设计思想

规范了工程化、工厂化的软件设计方法，提供了明晰可靠的软件接口标准，使软件构件可以像工业零件一样生产制造，零件可用于各种不同的设备上。

7、 提高系统的可靠性、容错性

由于构件运行环境可控制，可以避免因个别构件的崩溃而波及到整个系统，提高系统的可靠性。同时，系统可以自动重新启动运行中意外停止的构件，实现系统的容错。

8、 有效地实现系统安全性

系统可根据构件的自描述信息自动生成代理构件，通过代理构件进行安全控制，可以有效地实现对不同来源的构件实行访问权限控制、监听、备份容错、通信加密、自动更换通信协议等等安全保护措施。

2.2 Elastos 中间件平台

Elastos 是一种新的计算平台，它是在设计中把网络统一考虑进去了的操作系统，简化了在高度分布式 Internet 环境中的应用程序开发。Elastos 旨在实现下列目标： 提供一个一致的面向对象的编程环境，而无论对象代码是在本地存储和执行，还是在本地执行但在 Internet 上分布，或者是在远程执行的。提供一个将软件部署和版本控制冲突最小化的代码执行环境。 提供一个保证代码（包括由未知的或不完全受信任的第三方创建的代码）安全执行的代码执行环境。提供一个可消除脚本环境或解释环境的性能问题的代码执行环境。

CAR 构件运行时环境是 Elastos 的重要组成部分，在不造成歧义的情况下，我们认为 Elastos 就是 CAR 构件运行时。

“Elastos 构件运行平台”提供了一套符合 CAR 规范的系统服务构件及支持构件相关编程的 API 函数，实现并支持系统构件及用户构件相互调用的机制，为 CAR 构件提供了编程运行环境[13]。Elastos 构件运行平台在不同操作系统上有不同的实现，符合 CAR 编程规范的应用程序通过该平台实现二进制级跨操作系统平台兼容。在 Windows 2000、WinCE、Linux 等其他操作系统上，Elastos 构件运行平台屏蔽了底层传统操作系统的具体特征，实现了一个构件化的虚拟操作系统。在 Elastos 构件运行平台上开发的应用程序，可以不经修改、不损失太多效率、以相同的二进制代码形式，运行于传统操作系统之上。CAR 构件技术主要解决的问题有：不同来源的构件实现互操作，构件升级不会影响其他的构件，构件独立于编程语言，构件运行环境的透明性。

同时，Elastos 构件运行平台提供的功能模块全部基于 CAR 构件技术，是可拆卸的构件，应用系统可以按照需要剪裁组装，或在运行时动态加载必要的构件，

还可以用自己开发的构件替换已有模块。

2.1 Elastos 平台的主要特点有

（1）引入 CAR 构件技术，为移动平台系统的软件开发的工程化、工厂化提供基础。Elastos 平台的构件化软件设计思想规范了工程化、工厂化的软件设计方法，提供了明晰可靠的软件接口标准，使软件构件可以像工业零件一样生产制造，零件可用于各种不同的设备上，这点优势将对移动平台领域软件的工程化开发产生很大影响。

（2）支持动态加载和升级构件。在网络时代，需要将软件构件视作零件一样，随时进行装配，以满足用户的各种计算需求，可以说动态加载构件是必要的功能。尤其是随着 iPhone、Android 新一代智能手机的出现，用户需要能够动态的通过网络获取需要的程序。CAR 技术实现了构件动态加载，满足用户的这些需求。同时因为构件可动态加载，软件的升级也变得更加简便，开发商也不需要再为了添加了部分功能而向用户重新发布整套软件，不再“牵一发而动全省”，而只需升级个别的构件即可解决软件的升级问题。

（3）支持软件重用。软件重用一直是软件工程追求的目标，CAR 技术提供了构件的标准，二进制构件可以被不同的应用软件使用，使软件构件真正能够成为工业零件。移动平台软件开发商可以充分利用此优势来建立自己的构件库，在不同开发阶段开发的软件构件，其成果很容易被以后的开发所共享，充分利用经过考验的软件零件，避免重复性开发，使得系列产品的开发更加容易，新产品开发周期也得以缩短，可以提高软件生产效率和软件产品的质量，保护软件开发的投资。

（4）跨平台兼容的特性使得软件移植的风险被降低。基于 Elastos 平台开发的应用软件具有跨平台的特性，可运行在 Windows、Linux 等 PC 平台上，也可运行在 Windows Mobile、Android、Symbian 等系统上，这一特点既可以大大节约软件移植的费用，也可以避免因移植而带来的其它隐患。

（5）功能完备的开发环境和便利的开发工具将节约开发时间。这些环境和工具将帮助程序员学习和掌握先进的构件编程技术，可以在开发环境下开发调试应用软件，与硬件研制工作同时进行，缩短产品研制周期。

2.2 CAR 构件技术在 Elastos 中的作用

CAR 技术由操作系统内核来实现，可以充分利用内核中的线程调度、跨进程通讯、软件装卸、服务定位等设施对 CAR 构件提供高效、可靠的服务。同时内核本身的程序实现也可因利用 CAR 技术而变得更加模块化，从而加强对内核的软件工程管理。

Elastos 操作系统正是基于这样的思路实现的。Elastos 中的操作系统内核、

Elastos 构件运行平台提供的构件库，都是用 CAR 技术实现的。内核与 CAR 技术运行环境的紧密结合，为 Elastos 的“灵活内核”体系结构提供有力的支持，高效率地实现了全面面向构件技术的新一代操作系统。

虽然 CAR 技术会增加内核代码量，但脱开应用一味强调内核大小并没有意义，CAR 技术引入内核将会大大减少各种应用软件与操作系统的总体资源开销。在 Elastos 构件运行平台上直接运行二进制构件，这也符合对运行效率、实时性有严格要求的嵌入式系统的工业要求。二进制代码就是实际的 CPU 指令流，其所需的执行时间是可计算的，因此，系统运行时间是可预知的（predictable），这是目前存在的其他虚拟机系统所不能及的。

2.3 Java 虚拟机

2.3.1 Java 虚拟机简述

Java 编程语言是一种典型的面向对象语言[20]，它的语法类似于 C 语言和 C++，但去掉了 C、C++语言中一些混淆的、不安全的特性，如多重继承、指针等。Java 平台的设计是用来支持多主机平台和允许软件构件的安全递送[33]。

Java 虚拟机是 Java 平台的基石，使得 Java 平台不依赖于硬件和操作系统而存在，使得 Java 程序具有较小的目标代码尺寸，并且能够保证用户不受恶意程序的攻击。Java 虚拟机是一个抽象的计算机，与真实的计算机相同，它具有指令集、运行时管理内存的功能。Java 虚拟机操作的是一种特殊的二进制文件——字节码(class)文件，虚拟机并不知道 Java 语言的语法，只是通过执行 class 文件中的栈操作等虚拟机指令来实现程序的运行，class 文件是通过对 Java 程序源文件编译而产生，类似于目标代码文件。正因为如此，Java 虚拟机也可以用来作为其他语言的解释器，只需要将其他语言编译成 class 文件，如 JRuby、JPython、Groovy 等较受欢迎的动态编程语言。

Java 规范并未对 Java 虚拟机如何实现有任何规定，也没规定 Java 虚拟机运行的硬件或操作系统的要求。在 Java 语言诞生的十几年来，各种平台中都有 Java 虚拟机的身影，小至 ARM 芯片，大至大型服务器，无论是平台差异性较大的手机平台，还是应用最广泛的个人电脑平台，都有相应的 Java 虚拟机的实现，这也使得 Java 程序能够具有最广泛的应用领域，也实现了“一次编译，处处运行”的口号。

2.3.2 Java 虚拟机的体系结构

在 Java 虚拟机规范中，一个虚拟机实例分为类装载器子系统、内存区、数据类型以及指令这几个部分组成的。这些部分共同展示了虚拟机的内部抽象体系结构。规范中并非通过此规定虚拟机内部的体系结构的实现方式，而是用于定义对于虚拟机外部实现的特征。每个虚拟机都有两种机制，一个是装载需要加载的类(类或是接口)，叫做类装载子系统；另一个负责执行包含在类中的指令，叫做运行引擎。每个 JVM 又包括方法区、堆、Java 栈、程序计数器和本地方法栈这五个部分，这几个部分和类装载机制与运行引擎机制一起组成的体系结构图为：

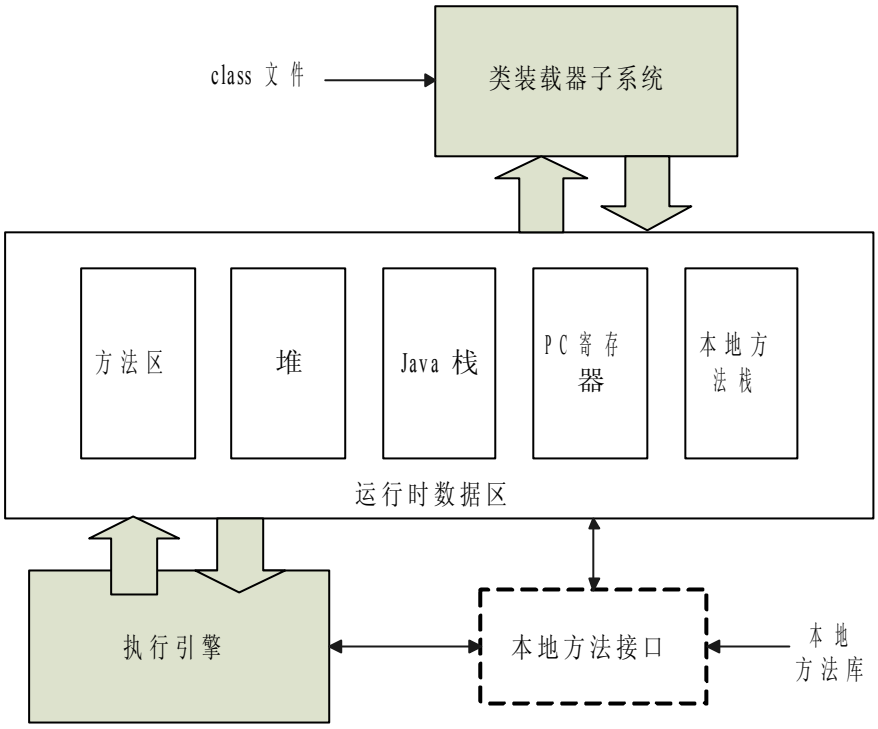

图 2.6 Java 虚拟机的内部体系结构

执行引擎处于 JVM 的核心位置，执行引擎就相当于运行 Java 字节码的“CPU”，但并不是通过硬件实现的，而是用软件实现的。在 Java 虚拟机规范中，它的行为是由指令集所决定的。尽管对于每条指令，规范很详细地说明了当 JVM 执行字节码遇到指令时，它的实现应该做什么。Java 虚拟机支持大约 248 个字节码。每个字节码执行一种基本的 CPU 运算，例如，把一个整数加到寄存器，子程序转移等。Java 指令集相当于 Java 程序的汇编语言，Java 指令集中的指令包含一个单字节的操作符，用于指定要执行的操作，还有 0 个或多个操作数，提供操作所需的参数或数据。许多指令没有操作数，仅由一个单字节的操作符构成。

对于本地方法接口，实现 JVM 并不要求一定要有它的支持，甚至可以完全没有。Sun 公司实现 Java 本地接口(JNI)是出于可移植性的考虑，当然也可以根据需要设计出其它的本地调用机制来代替 JNI 机制。但是这些设计与实现是比较复杂的事情，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

掉。

Java 虚拟机中的运行时数据区主要分为五部分：

（1）PC 寄存器

Java 虚拟机支持许多线程的同时运行，对于每一个线程都会有自己的 PC 寄存器，也叫做程序计数器。当线程执行 Java 方法的时候，PC 寄存器中的内容总是下一个将被执行指令的“地址”。这个地址可能是一个本地指针，也可能是字节码中指令的偏移量。但是若线程执行的是一个本地的方法，那么程序计数器的值就不会被定义。

（2）Java 栈

每个虚拟机线程创建时都会同时创建一个 Java 栈，虚拟机只会直接对 Java 栈做两种操作：压栈或出栈。栈的容量可以设为固定值或者可动态扩展。

Java 栈中由许多栈帧组成，一个栈帧包含一个 Java 方法调用的状态。栈帧由三部分组成：局部变量区、运行环境区、操作数区。局部变量区和操作数区的大小要视对应的方法而定，都在 Java 文件编译时确定了，并写入 class 文件中。而运行环境区的大小依赖于具体的实现。每当线程调用一个 Java 方法时，虚拟机都会在该线程的 Java 栈中压入一个新帧，即压栈。

Java 方法完成时就会执行出栈操作，完成的方式有两种。一种是通过 return 语句返回，是正常返回；另一种是通过抛出异常而异常中止的。

（3）堆

Java 的堆是一个运行时数据区，Java 程序创建的所有类实例或数组都放在同一堆中。对于一个 Java 虚拟机实例都只存在一个堆空间，因此所有线程都共享这个堆。又因为每一个 Java 程序都独占一个 Java 实例，因而每个 Java 程序都有自己私有的堆空间。但同一个 Java 程序的多个线程则共享同一个堆空间，这时需要考虑多线程访问对象的同步问题了。

堆的管理是虚拟机负责的，不给程序员显式释放对象的能力，虚拟机自己负责决定如何以及何时释放不再被运行的程序引用的对象所占据的内存。这个任务的执行者就是垃圾收集器。Java 不规定具体使用的垃圾回收算法，只规定了虚拟机实现必须“以某种方式”管理自己的堆空间。这具体的垃圾回收算法，都是由虚拟机的设计者根据他们的目标、考虑所受的限制、用自己的能力去决定什么才是最好的技术。因为对象的引用可能很多地方都存在，如 Java 栈、堆、方法区、本地方法，所以垃圾收集技术的使用在很大的程度上影响着运行时数据区地设计。

（4）方法区

Java 虚拟机中，被装载类型的信息存储在一个逻辑上成为方法区的内存中。

当虚拟机装在某个类型时，它使用类装载器定位相应的 class 文件，然后读入这个 class 文件并将它传输到虚拟机中。紧接着，虚拟机提取其中的类型信息并将这些信息存储到方法区。而具体虚拟机是如何存储这些信息的是由具体的设计实现者来决定的。

由于所有线程都共享方法区，因此它们对方法区数据的访问必须设计为是线程安全的。比如，如果两个线程要访问一个类，而这个类还没有被加载，那么此时应该只有一个线程去加载这个类，而另一个线程则只能等待。

方法区的大小不必是固定的，虚拟机可根据需要动态调整，可以在一个堆中自由分配。方法区也可以被垃圾收集，因为虚拟机允许通过用户定义的类装载器动态扩展 Java 程序，因此一些类也会成为程序“不再引用”的类。当某个类成为“不再引用”的类时，Java 虚拟机可以卸载这个类（垃圾收集），从而是方法区占据的内存保持最小。

Java 方法区与传统语言中的编译后代码或是 Unix 进程中的正文段类似。它保存方法代码（编译后的 java 代码）和符号表。在当前的 Java 实现中，方法代码不包括在垃圾回收堆中，但计划在将来的版本中实现。每个类文件包含了一个 Java 类或一个 Java 界面的编译后的代码。可以说类文件是 Java 语言的执行代码文件。为了保证类文件的平台无关性，Java 虚拟机规范中对类文件的格式也作了详细的说明。

（5）本地方法栈

当一个线程调用本地方法时，它就不再受到虚拟机关于结构和安全限制方面的约束，它既可以访问虚拟机的运行期数据区，也可以使用本地处理器以及任何类型的栈。例如，本地栈是一个 C 语言的栈，那么当 C 程序调用 C 函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。在实现 Java 虚拟机时，本地方法接口使用的是 C 语言的模型栈，那么它的本地方法栈的调度与使用则完全与 C 语言的栈相同。

任何本地方法接口都会使用某种本地房方法栈。当线程调用 Java 方法时，虚拟机会创建一个新的栈帧并压入 Java 栈。然而当它调用的是本地方法时，虚拟机会保持 Java 栈不变，不再在线程的 Java 栈中压入新的帧，虚拟机只是简单地动态链接并直接调用指定的本地方法。可以把这看作是虚拟机利用本地方法来动态扩展自己。

很可能，本地方法接口需要回调 Java 虚拟机中的 Java 方法，在这种情况下，线程会保存本地方法栈的状态并进入到另一个 Java 栈中，如图 2.7 所示：

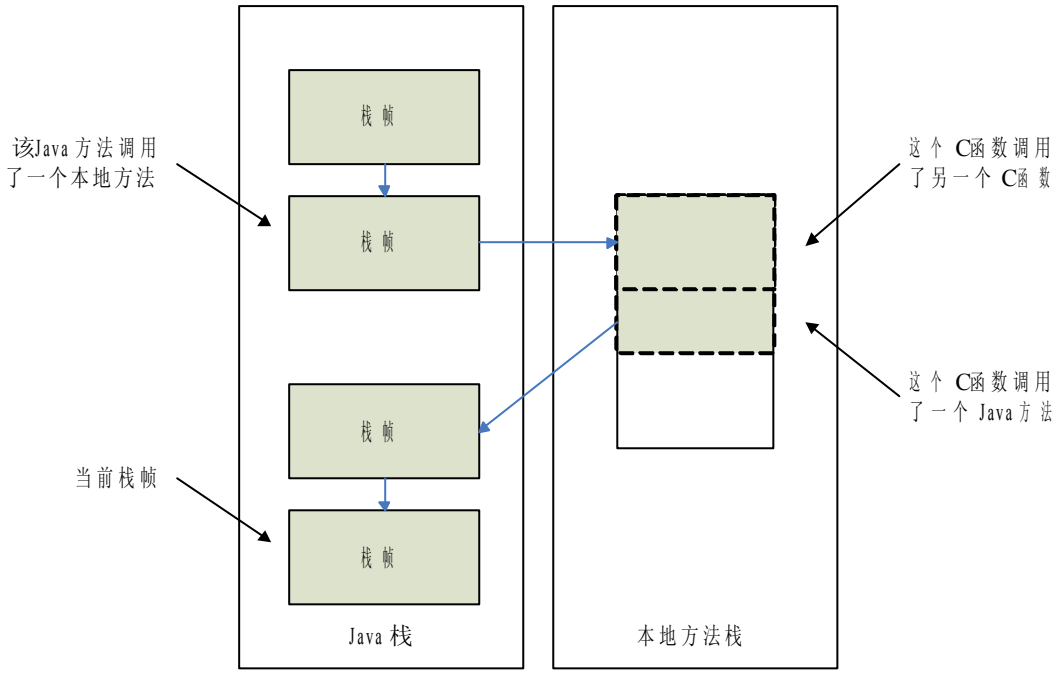

图 2.7 一个线程调用 Java 方法和本地方法时的栈

2.3.3 Dalvik 虚拟机

Dalvik 虚拟机是 Google 公司自主研发的代号为 Dalvik 的 Java 虚拟机技术，是其 Android 系统运行的基石。Android 系统未采用其他开源虚拟机实现的原因一个是避开 Sun 公司 Java 授权问题，另一个是尽可能的加快在移动设备中 Java 程序的执行效率。Dalvik 虚拟机对 Java 虚拟机的设计进行了较大幅度的优化。首先是对字节码文件的优化，采用了对 class 文件优化后产生的 dex 文件[25]。其次，不同于大多数的 Java 虚拟机采用基于栈的架构，Dalvik 虚拟机采用基于寄存器的架构，这种方式减少了操作一次数据所需的指令数。尽管，目标代码文件以及虚拟机指令集与其他 Java 虚拟机差别很大，但 Java 虚拟机基础类库 API 依然与 Java 规范保持兼容。

Dalvik 虚拟机非常适合在移动终端上使用，相对于在桌面系统和服务器系统运行的虚拟机而言，它不需要很快的 CPU 速度和大量的内存空间。根据 Google 的测算，64M 的 RAM 已经能够令系统正常运转了。其中 24M 被用于底层系统的初始化和启动，另外 20M 被用于高层启动高层服务。当然，随着系统服务的增多和应用功能的扩展，其所消耗的内存也势必越来越大。

Dalvik 虚拟机相比于传统的 Java 虚拟机主要具有如下几个特征：

（1）特有的字节码文件

一个应用中会定义很多类，编译完成后即会有很多相应的 CLASS 文件，CLASS 文件间会有不少冗余的信息；而 DEX 文件格式会把所有的 CLASS 文件

内容整合到一个文件中，这样，除了减少整体的文件尺寸，I/O 操作，也提高了类的查找速度。同时还增加了新的操作码的支持。原来每个类文件中的常量池，在 DEX 文件中由一个常量池来管理。DEX 文件结构尽量简洁，使用等长的指令，借以提高解析速度，并且尽量扩大只读结构的大小，借以提高跨进程的数据共享。

（2）基于寄存器的架构

相对于基于堆栈的虚拟机实现，基于寄存器的虚拟机实现虽然在硬件通用性上要差一些，但是它在代码的执行效率上却更胜一筹。一般来讲，虚拟机中指令的解释执行时间主要花在以下三个方面： 分发指令、访问运算数、执行运算。其中“分发指令”这个环节对性能的影响最大。在基于寄存器的虚拟机里，可以更为有效的减少冗余指令的分发和减少内存的读写访问。虽然 Dalvik 虚拟机并没有使用目前流行的虚拟机技术，如 JIT，但是根据 Google 的报告，这个功能的缺失并没有另 Dalvik 虚拟机在性能上有所损失。我们也同时相信，Dalvik 虚拟机的性能还有进一步提高的空间。

（3）应用具有独立的进程空间

每一个 Android 应用都运行在一个 Dalvik 虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制，内存分配和管理，Mutex 等等都是依赖底层操作系统而实现的。所有 Android 应用的线程都对应一个 Linux 线程，虚拟机因而可以更多的依赖操作系统的线程调度和管理机制。不同的应用在不同的进程空间里运行，加之对不同来源的应用都使用不同的 Linux 用户来运行，可以最大程度的保护应用的安全和独立运行。Zygote 是一个虚拟机进程，同时也是一个虚拟机实例的孵化器，每当系统要求执行一个 Android 应用程序，Zygote 就会 FORK 出一个子进程来执行该应用程序。这样做的好处显而易见：Zygote 进程是在系统启动时产生的，它会完成虚拟机的初始化，库的加载，预置类库的加载和初始化等等操作，而在系统需要一个新的虚拟机实例时，Zygote 通过复制自身，最快速的提供个系统。另外，对于一些只读的系统库，所有虚拟机实例都和 Zygote 共享一块内存区域，大大节省了内存开销。

2.4 OSGi 框架

OSGi（Open Services Gateway initiative）是面向 Java 编程语言的动态模型系统和服务平台，实现了一个完整的、动态的组件模型[22]。OSGi 最初是针对家庭高速网关服务而提出的，现在应用领域包括移动手机、Eclipse IDE 开源工程、汽车工业、建筑自动化、网格计算、应用服务器等等。

OSGi 框架中构件的名称是 Bundle（束），一个 Bundle 代表一个独立功能单

元，具有某一特定功能、可以独立运行、动态安装卸载等性质。OSGi 规范中规定了 Bundle 可以远程安装、启动、停止、更新、卸载等动态操作，并且针对 Bundle 中的包、类都有详细的规定。框架中的服务注册机制允许 Bundle 可以动态的发现新服务的加入或移除，并且动态的进行调整。

目前，软件的复杂性正在快速的提高，而这个主要是由于缩短的生产周期、增加的功能需求以及同类产品的大量重复而导致的，这也大幅度的增加了软件生产成本，是发生软件危机的主要原因。而且，当程序员更新系统中某一功能模块时，而这一功能模块已经成为系统运行的重要部分，因此如何保证在系统运行时安全的对关键模块进行动态更新是很重要的。而 OSGi 框架就是充分利用 Java 语言的动态性，并且借助于 Java 语言执行的平台无关性，使得 OSGi 框架具有很广阔的应用领域

OSGi 框架提供了一个标准化的程序（Bundle）运行环境，这个框架中具有四层[22]，如图 2.8：

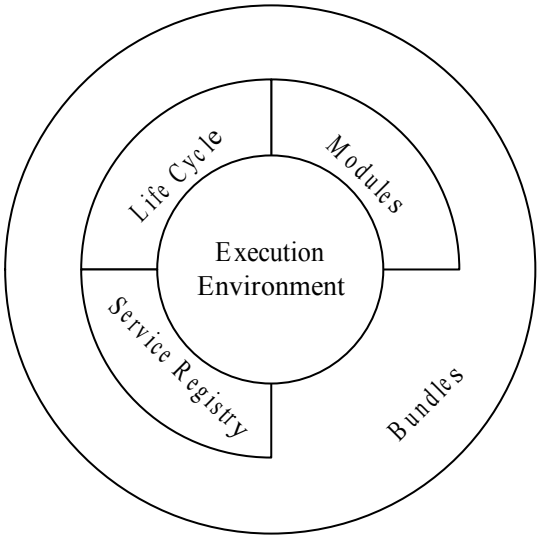

图 2.8 OSGi 框架

L0 层：运行环境层（Execution Environment）。运行环境是指 Java 规范中规定的运行环境，如 J2SE，CDC，CLDE，MIDP 等都是有效的运行环境。根据 OSGi 规范，OSGi 平台也具有一个最小的运行环境集合用来执行 Bundles。所谓 Bundle，在 OSGi 规范中是指模块化单元。在 OSGi 服务框架中，Bundle 是唯一可部署的 Java 应用程序实体。一个 Bundle 由 java 的类和其他资源组成，可以为终端用户提供功能。Bundle 和普通的 Java 工程唯一不同的是需要在 MANIFEST.MF 中编写 Bundle 的元数据信息。所谓 Bundle 的元数据，就是指关于 Bundle 的所有信息在 MANIFEST.MF 中进行的描述，这些信息中包含有 Bundle 的名称、描述、开发商、类路径、导入的包和导出的包、导入的服务和导出的服

务等。在 Bundle 启动后，安装在 OSGi 服务平台的其它 Bundle 便可以使用其所提供的功能和服务。

L1 层：模块层（Modules）。模块层定义了类加载策略和针对 Java 语言的模块模型。模块层对多个 Bundle 之间共享、隐藏 Java 包有严格的规定。一些基于 Java 的工程，如 JBoss、NetBeans，可以通过自定义类加载器来创建自有的面向对象层来进行包装、部署等工作。在模块层中最重要的是提供独立的类加载机制，保证各个 Bundle 之间的隔离与共享，OSGi 框架中的类加载流程如图 2.9 所示。

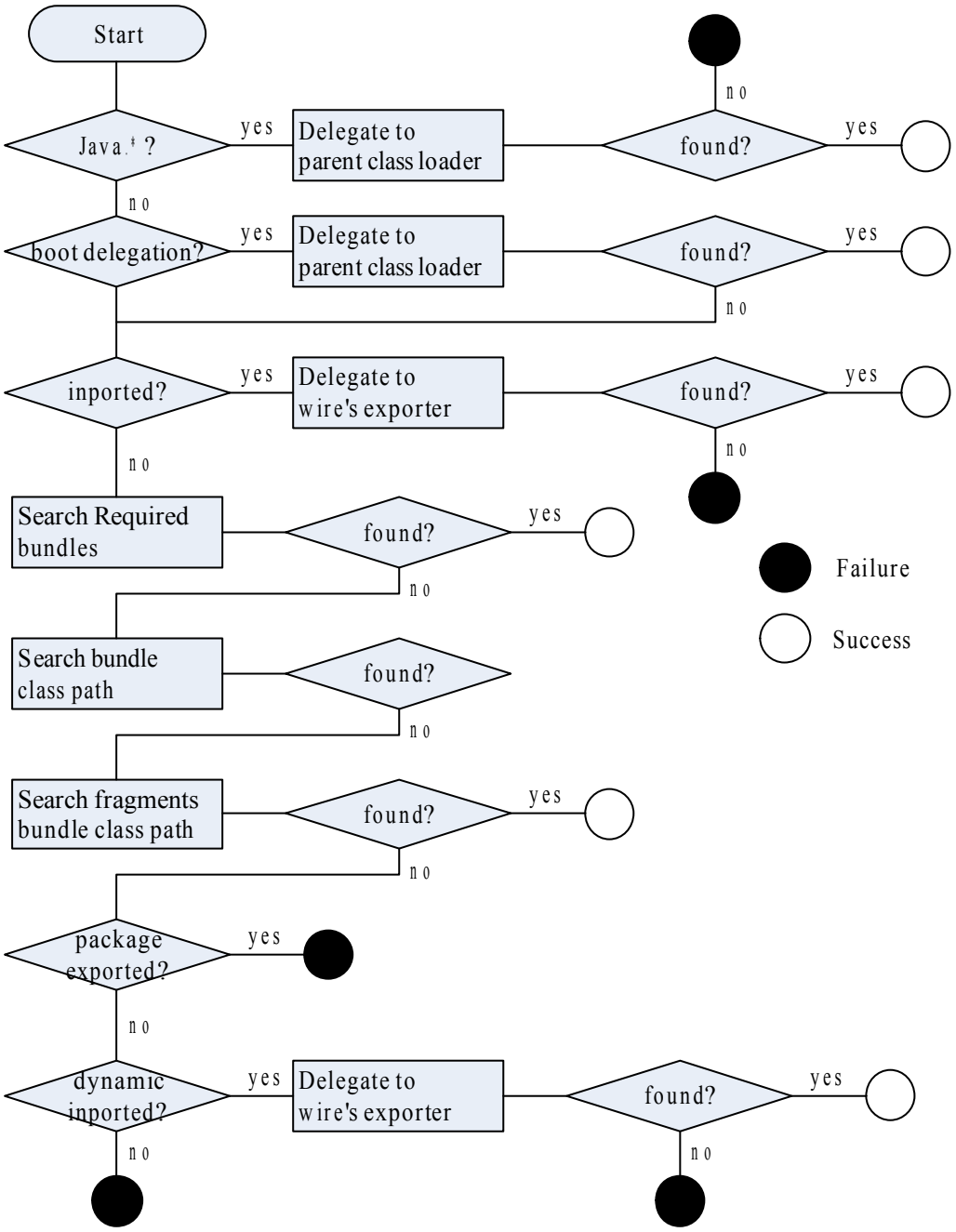

图 2.9 OSGi 框架类加载流程图

L2 层：生命周期层（Life Cycle）。生命周期层管理 Bundles 动态的安装、启

动、停止、更新和卸载，提供 API 来在运行时对 Bundles 进行管理。生命周期层通过安全机制进行完全的保护，保证程序避免被病毒侵入。

Bundle 可能具有以下几种状态：

(1)INSTALLED：成功安装 Bundle。

(2)RESOLVED：Bundle 依赖的所有类都已准备好。这个状态标志着 bundle 已经是启动就绪或者是已经停止。

(3)STARTING：正在启动 Bundle，还未完成启动。

(4)ACTIVE：Bundle 已经启动完毕，正在运行中。

(5)STOPPING：正在停止 Bundle，还未完成停止。

(6)UNINSTALLED：Bundle 已经卸载完毕，不能进入其他状态。

这几种状态之间的转换图如图 2.10 所示。

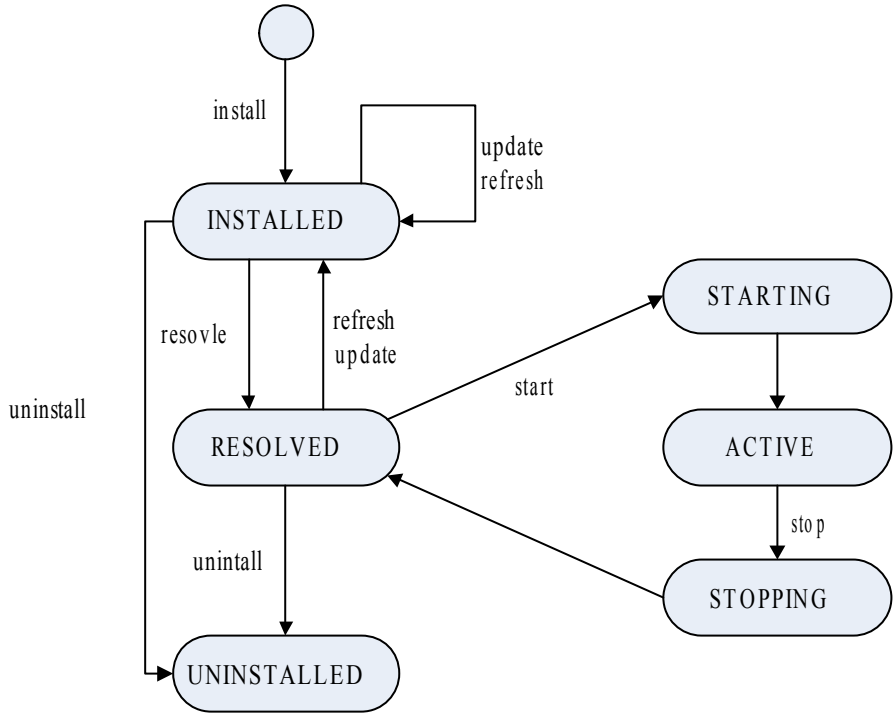

图 2.10 Bundle 状态转换图

L3 层：服务层（Service Registry）。服务层向 Java Bundle 开发者提供了一个动态的、简洁的、一致的编程模型，通过 Java 接口将服务说明与服务实现进行分离，来保证服务注册、注销具有运行时动态性。该层提供了一个综合的模式来进行 Bundles 之间的对象共享，提供了一种优秀的方式保证 Bundles 之间数据的传输。一个服务（Service）就是一个通过服务登记来注册到一个或者多个 Java 接口下的 Java 对象。Bundle 可以注册服务，查找服务，接收注册服务的状态改变信息。

OSGi 框架目前已经有很多成功的商业实例。最著名的是两个：Eclipse，BMW

汽车的应用控制系统。

Eclipse 作为 Java 业界成功的 IDE project，在 3.0 以前的版本它采用的是自己设计的一套插件体系结构，在整个业界被认为非常成功的一种设计，但 Eclipse 在 3.0 版本时却转为采用 OSGI 作为其插件体系结构。这是由于 Eclipse 的插件体系结构和 OSGI 的思想非常的耦合，都强调微核、系统插件、应用插件的概念，因为 OSGI 的规范性以及 OSGI 对于插件体系结构更为完整的定义，所以采用了 Eclipse 这么快的转变。Eclipse 采用 OSGI 作为其插件体系结构后启动效率具有明显的提升，同时也使得可以在运行时对插件进行管理，更明显的提升是插件的开发更加的规范，从而可以使用很多已有的 OSGI 插件。这也带来了良好的插件系统的体验以及插件系统的开发经验。

宝马汽车的应用控制系统采用 OSGi 作为其底层架构最初是很令人惊讶的，因为很多人都认为 Java 的效率不高，不可能用于汽车这样的应用控制系统上。在 EclipseCon 2006 会议上 BMW 采用 OSGi 得到了证实，介绍了 BMW 汽车的应用控制系统，这套系统主要用来控制汽车上的音箱、灯光等设备，总共由 1000 多个 Bundle 构成，但 BMW 汽车的应用控制系统启动时间却只需要 3.5 秒，这也说明 Java 语言的效率在很多时候都是可以满足需求的。

第 3 章 JNC 机制的研究

本章首先介绍 Java 虚拟机规范中规定的 JNI 机制的设计，然后给出本地构件调用机制（JNC）的定义及其设计目标，最后对设计思路以及设计中需要关注的问题进行探讨。

3.1 对现有机制的分析

本小节主要对当前的 Java 虚拟机本地调用机制进行简要的分析。

JNI 机制不仅仅是 Java 虚拟机调用本地代码的机制，也是 Java 虚拟机自身运行依赖的基础。

3.1.1 设计目的

首先，提供二进制兼容能力，对于同一个宿主平台上的不同的 Java 虚拟机实现能够对相同的本地二进制库进行调用，因此，JNI 机制的设计不涉及 Java 虚拟机的具体实现。

其次，保证效率。对于本地库的调用，除了对于本地功能的需求，另一个重要原因是满足部分对时间要求高的程序，因此 JNI 机制的设计要避免对效率的额外消耗。

最后，全面能力。Java 虚拟机规定的 API 很难覆盖所有的本地能力，因此通过 JNI 机制，使得 Java 虚拟机能够通过这种方式满足程序的所有功能需求。

3.1.2 加载本地库

在 Java 程序调用本地方法之前，虚拟机必须首先加载包含本地方法实现的本地库。

本地库都是通过类加载器（classloader）进行加载的，通过 System.loadLibrary() 方法来实现加载，参数是本地库的名字。不同的平台上本地库的名字会被进行相应的扩展，例如，名为“native”的本地库，在 Linux 平台上会被对应于 libnative.so 文件，在 Windows 平台上会被对应于 native.dll 文件。虚拟机在加载本地库时会在本地平台库文件路径下查找相应的文件。库文件路径也可通过平台的环境变量进行相应的设置。

本地库对应的类加载器是通过如下几步进行确定的：

首先，确定调用 System.loadLibrary()方法的区域块；

其次，找到此区域块所在的类；

最后，这个类对应的加载器就是要找的类加载器。

例如，下面的例子中 foo 本地库就是通过类 C 的加载器进行加载的。

```
class C {
    static {
        System.loadLibrary("foo");
    }
}
```

3.1.3 链接本地方法

虚拟机在本地方法第一次被调用前进行链接。链接本地方法的步骤是：首先，确定本地方法对应的类的加载器；其次，在类加载器对应的本地库集合中查找与本地方法对应的本地函数；最后，对本地方法建立内部数据结构，使得所有对本地方法的调用都直接跳转至本地函数。

在进行本地方法与本地库中的本地函数进行比对时，比对的方式是通过名称。本地函数的名字应该包括如下几项：前缀“Java_”，完整的类名，下划线“_”，方法名。对于重载函数最后应该还包括“__”和编码的参数类型。

例如，类 Test 中的方法 func 对应的本地函数名应该为“Java_com_company_test_Test_func”。

```
package com.company.test;
Class Test {
    public native int func();
}
```

3.1.4 JNIEnv 接口指针

对于每一个被调用的本地函数，第一个参数都是 JNIEnv 接口指针，指向被预先设定好的函数表。函数表中包括 Java 语言的一些基本操作，包括创建对象、调用方法、获取、设置对象属性值、释放对象等，以便于本地库对 Java 程序的功能调用。

采用接口指针的方式，而不是采用固定的函数入口的方式的原因[4]主要是：

第一，接口指针的方式使得本地库不需要与 Java 虚拟机的具体实现进行链接；

第二，由于不使用固定的函数入口，因此 Java 虚拟机可以在不同的情况下提供不同函数表。例如，调试虚拟机时提供一类包括完整的参数检验的函数；而

程序正常运行时就提供效率最优的一组函数；

第三，函数表的方式便于在未来对 JNIEnv 的接口函数进行扩展。

3.1.5 数据传递

对于基本数据类型，如 int、char、boolean 等，在虚拟机与本地代码之间采用副本的方式进行传递。

对于其他复杂数据，即对象，采用引用传递。每一个对象的引用中包含指向对象的指针，但这个指针对本地代码是不透明的，无法直接使用的。不直接传递指针而是传递一个引用给本地代码，使得虚拟机能够更灵活的管理 Java 对象，如图 3.1 所示。

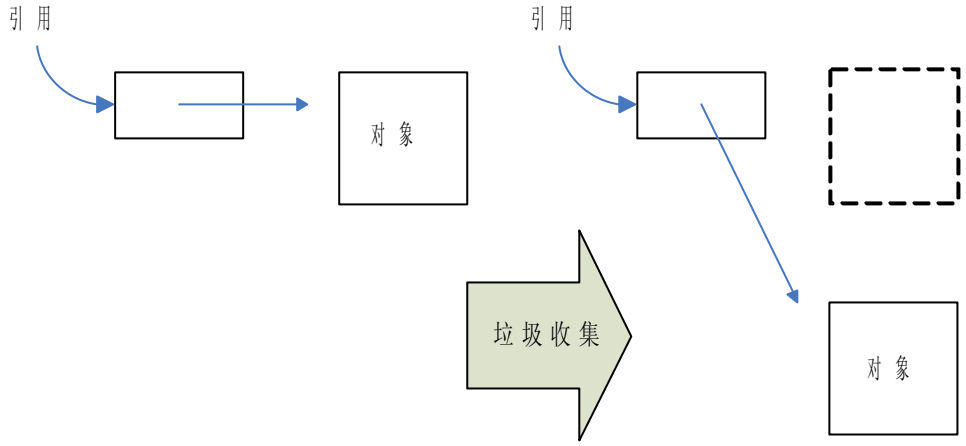

图 3.1 垃圾收集后本地代码重定位对象

采用引用的方式可以避免本地代码对 Java 对象的直接依赖，也避免本地代码对 Java 虚拟机所管理的内存进行直接操作而导致程序异常退出。

3.1.6 异常处理

JNI 的异常处理机制是指本地代码通过 JNIEnv 接口对 Java 虚拟机的操作引起的异常，不负责对本地代码一般错误的检查，因为对本地代码产生的所有可能的错误进行检查会较大的降低本地调用的执行效率。

不同于普通 Java 程序对异常处理是即时的，本地代码引起的 Java 虚拟机抛出的异常是被挂起在当前线程中，当此次 JNI 函数调用结束时再对挂起的异常进行处理。对于本地代码来讲，发生异常时，可以采用两种操作：一是立即返回，由 Java 虚拟机对本地方法抛出的异常进行处理；二是通过调用 ExceptionClear 清空异常，然后转入自己定义的异常处理代码。

3.2 本地构件调用机制的定义

正如前一章所言，CAR 构件是“基于 CPU 指令集的软件零部件运行单元”，具有与 C/C++程序相同的执行效率，并且具有元数据，可以执行反射操作。CAR 构件的执行环境 Elastos 具有包括 Windows、Linux、Android 等多个平台版本，因此，CAR 构件具有一定跨平台的能力。

对于构件技术来说，构件应该具有语言无关性，即对于一种构件技术，应规定的是构件与构件相互调用的规则以及管理的框架，但对具体构件的实现语言应该没有特殊规定。因此，当构件技术持续发展时，用更高级、动态的语言编写的构件会逐渐用效率更高、低级语言编写的构件替代，并且不影响其他构件的运行。而在发展过程中，因为不同语言具有的相应特性，相应的构件会扮演不同的角色。如 C 语言编写的构件，具有良好的效率以及本地能力，但生产成本较高，相比于脚本语言生产周期也较长，因而作为本地功能或计算强度高的模块更为合适；而脚本语言编写的构件，如 Lua、JavaScript 等，具有很强的动态性，容易编写，但效率不高，更应该作为构件组装者的角色；使用 Java 语言的程序员数目最多，当前 Java 程序也十分庞大，又具有良好的平台无关性，因此支持 Java 语言编写的构件对于开发者和构件技术的推广都有很强的意义。

对于 CAR 构件和 Java 构件来说，都具有跨平台的能力，都具有反射、动态性、可扩展性等性质。对于 JNI、JNA 机制来说，向 Java 开发者提供了对本地库进行函数级调用的能力，其中一个是静态调用，一个可以动态调用。但对于一个构件系统，尤其是 CAR 构件技术，需要的是 Java 构件与 CAR 构件的等价替代，形成与语言无关的构件生态系统。

因此，本文提出了本地构件调用机制作为解决这种需求的方案。

在 Elastos 平台上，采用平台中的 Bonsai[19]和 Java 虚拟机[17]为各自的 CAR 构件和 Java 构件的执行环境，通过这两种构件的相互组装、相互调用而形成一个应用程序的构建过程及其运行过程，完成这种过程的机制我们称之为本地构件调用机制（Java Native Component），简称 JNC 机制。

通过 JNC 机制加载的 CAR 构件（或者是 Java 构件），称之为 JNC 构件。其中，根据 CAR 构件与 Java 构件的特性及其当前发展现状，本文所述的 JNC 机制更注重 Java 构件对 CAR 构件的加载，因此，下文所述 JNC 构件若不额外说明，则主要是指通过 JNC 机制加载的 CAR 构件。

3.3 设计目标

JNC 机制属于一种“桥接”机制，基础目的是实现 Java 构件与 CAR 构件便捷的相互调用。在此基础之上，完成构件的统一管理、完成两种构件的相互组装，达到 Java 构件与 CAR 构件之间能够相互等价替换的能力。

JNC 机制的目标可以细分为以下几点：

3.3.1 动态性

作为构件技术中的一种机制，对构件的基本特性之一动态性的支持是必需的要求。构件技术的动态性不仅体现在加载、卸载等过程，还体现在接口的动态适配、服务提供者的动态查找等。

而对于 JNC 机制，这里的动态性主要是指 Java 构件（或 CAR 构件）按需动态加载 CAR 构件（或 Java 构件）。也就是说，在 JNC 机制下，当 Java 构件（或 CAR 构件）加载或运行过程中，其依赖的 CAR 构件（或 Java 构件）能够动态的安装、启动，在安装、启动后能够被 Java 构件（或 CAR 构件）即时调用。这个过程可分为加载、调用、卸载。加载、卸载过程与单一构件执行相比，需要增加对构件的初始化操作，以便于跨语言的调用。而调用过程中，则需要参数类型的转换、对象的封装、数据的同步等。

3.3.2 基本数据类型的支持

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。这类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。

对于 Java 语言中没有对应的一些 CAR 数据类型，如包括 DataTime 结构体类型，可以采用在 Java 端封装默认的结构体类，提供默认的操作方法，来完成 Java 端对这类数据类型的调用。对于 Buffer、Array 这类在 Java、CAR 两端具有不同操作方法的类型，也可采用类似的方式。

这些包括参数转换在内的一些方式的使用原则是在保证不损耗数据精度的前提下尽量提高调用效率。

3.3.3 数据同步

这里的数据一致性是指在 Java 语言执行环境与 CAR 构件执行环境中同时存

在的同一对象的数据及其副本是否能保持同步。

在构件技术中，一个构件对外提供的应该是接口，而应该避免直接提供数据，因此在 JNC 机制中，不应存在同一份数据在两个语言构件中都存在副本。

但实际操作中，一个构件在另一个语言环境中被加载调用时，由于需要一定的封装性的操作，因此参数传递时可能需要通过转换操作来形成数据的可用。尤其是对基本类型的操作，在转换操作中容易产生数据的同步问题。例如，Java 构件调用 CAR 构件的一个方法，参数所用内存由 CAR 构件（或 Java 构件）分配，而该数据由 Java 构件（或 CAR 构件）来操作、改变，再通过另一个调用返回给 CAR 构件（或 Java 构件）。这个过程中，由于必要的参数转换，同一个基本类型的数据在两个语言环境下都存在内存。若不保持同步，该数据同时被另一个构件调用，就可能会导致数据的错误。

首先，构件应避免将自己的数据直接暴露给其他构件使用，这一方面不利于数据的保护，另一方面也不利于构件自身对内存的管理。其次，构件间的相互调用，基本类型数据应采用值传递的方式，而避免采用指针传递的方式。而对于需要大量数据共享的构件间调用，应采用专门的类型如 Buffer、Array 等使用，而非直接采用内存的共享。

3.3.4 完善的内存管理

JNC 机制中内存的管理主要是指 JNC 构件加载及释放时，对其所占内存的管理，尤其是针对于跨语言环境而附加的桥接工作所用内存的管理。Java 语言具有垃圾收集机制，不需要开发者主动的去管理内存，而 CAR 构件是使用 C++管理内存的策略，由程序员负责内存的分配、释放。因此，对于 JNC 构件，当 Java 构件不需要其按需加载的 JNC 构件（CAR 构件时），如何将 CAR 构件的释放与 Java 语言的垃圾收集机制结合起来是解决问题的关键。

3.3.5 CAR 基本特性的支持

CAR 构件具有较多自有的基础特性，例如事件回调机制、Applet 机制、面向方面编程、反射、构件继承、同步异步调用机制等。尤其是同步异步调用、回调、反射、构件继承，这些是 CAR 构件的基本特性，而与 Java 语言自有的语法、语义又不完全相同，如何保证对这些 CAR 构件基本特性的支持，也是 JNC 机制是否满足需求的一个标准。

3.3.6 安全的执行环境

构件技术既要保证构件的可扩展性，也要保证构件执行的安全性，避免恶意代码的侵入。通过引入 Java 语言中对 Jar 包、Class 文件的验证机制，可以加强在构件加载过程中的安全机制。在建立 Java 语言与 CAR 构件的桥接环境过程中，要尽量限制将 CAR 构件的操作、数据直接暴露给 Java 执行环境，避免引入新的安全隐患。

3.3.7 统一的管理

JNC 机制应该成为对 Java 构件、CAR 构件形成统一管理框架的基础。在良好的桥接机制的保证下，对于同一个平台的跨语言构件应该进行统一的构件管理，构件的加载、注册服务、合并、消息传递、生命周期的管理等。

3.3.8 等价性

等价性是指，相同功能的 CAR 构件与 Java 构件能够在保证功能实现的前提下进行相互替换。满足等价性的前提是前五个设计目标的满足，并且需要在统一的管理框架在，才可能满足等价替换的特性。

3.3.9 快速的开发部署

作为基于 CAR 构件技术的附加机制，JNC 机制本身不应引入过多的桥接代码而导致 JNC 构件开发、部署步骤的繁琐。在满足动态加载、功能需求的前提下，应尽量避免构件尺寸的扩大、内存占用的增加、执行效率的降低。无论是对程序开发人员、还是部署人员，都应降低工作量，增强部署的自动化。

3.3.10 跨平台运行

Java 语言一直以优秀的平台无关性受到广大程序员的欢迎，CAR 构件的运行平台 Elastos 中间件也具有良好的跨平台能力。JNC 机制依赖对 Elastos 平台中 Java 虚拟机的改造，这个虚拟机来自于对 Android 系统上 Dalvik 虚拟机的移植。由于 Dalvik 虚拟机依赖于较多包括 fork、signal 等 Linux 机制，因此完善 Java 虚拟机移植也影响着 JNC 机制的稳定运行和跨平台的能力。

以上十个目标中，前五个目标是对 JNC 机制具体调用设计的要求，其他五个是在前者基础上对 JNC 机制的综合目标。

3.4 框架设计

对于 Java 构件与 CAR 构件存在于两种语言环境中，本文通过代理类来实现 Java 构件对 CAR 构件的依赖关系及调用。

图 3.2 是 JNC 机制总体示意图。

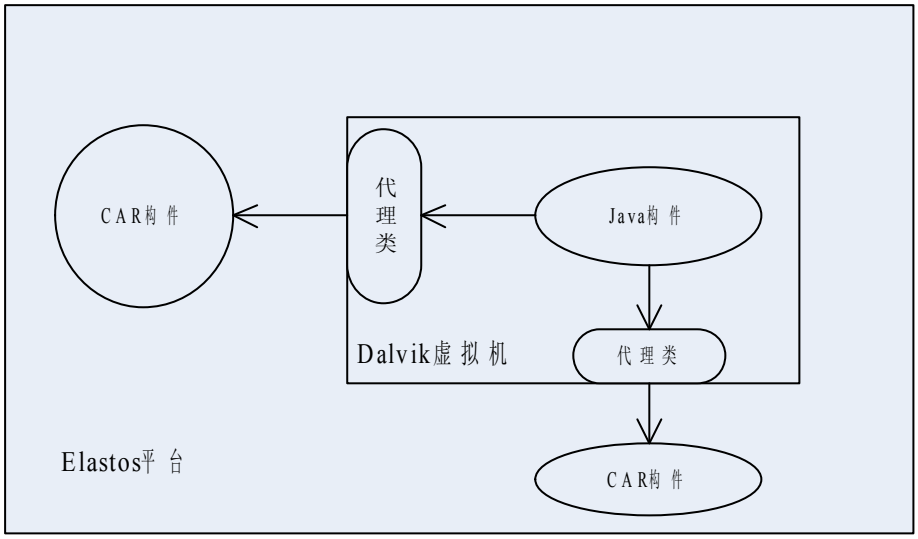

图 3.2 JNC 机制示意图

3.4.1 构件加载机制

同名代理类是指通过构建工具自动生成的与 CAR 构件中的类相对应的 Java 类，在 Java 虚拟机中起到 CAR 构件的代理作用。

那么如何区分 Java 普通类与 CAR 构件的代理类？

在 Java 规范 1.5 引入了 Annotation 机制，通过给类增加@注释，将程序中的类、方法、属性和元数据联系起来，并且将元数据存储的类文件中。Java 规范默认提供三个标准的 Annotation，@Deprecated 表示该方法不再被使用，@Override 表示该方法覆盖父类方法，@SuppressWarnings 表示该方法中的警告被忽略。

Annotation 是一种元数据，可以通过反射的方式获取，而又不影响程序的语言，因此，本文采用 Annotation 方式来修饰 CAR 构件的代理类。对于每一个 CAR 构件类，都将通过自动构建工具生成包含有 Annotation 元数据的代理类，来供 Java 构件对其进行调用。

在虚拟机执行过程中，当加载包含有 CAR Annotation 的信息时，将自动从系统构件库目录加载相应的 CAR 构件 HelloDemo.dll，并将 Java 代理类 CHello1 的类信息中加入 CAR 构件 HelloDemo 中 CHello1 的 IClassInfo 引用。

当虚拟机创建 Java 代理类 CHello1 的对象 JavaObj 时，会同时创建 CAR 构件中 CHello1 的对象 CARObj，并关联起来，同时将 Java 对象 JavaObj 的 native 方法与 CARObj 的对应方法关联起来，由此形成 Java 端程序对 CAR 构件的调用。

详细的加载过程如图 3.3 所示。

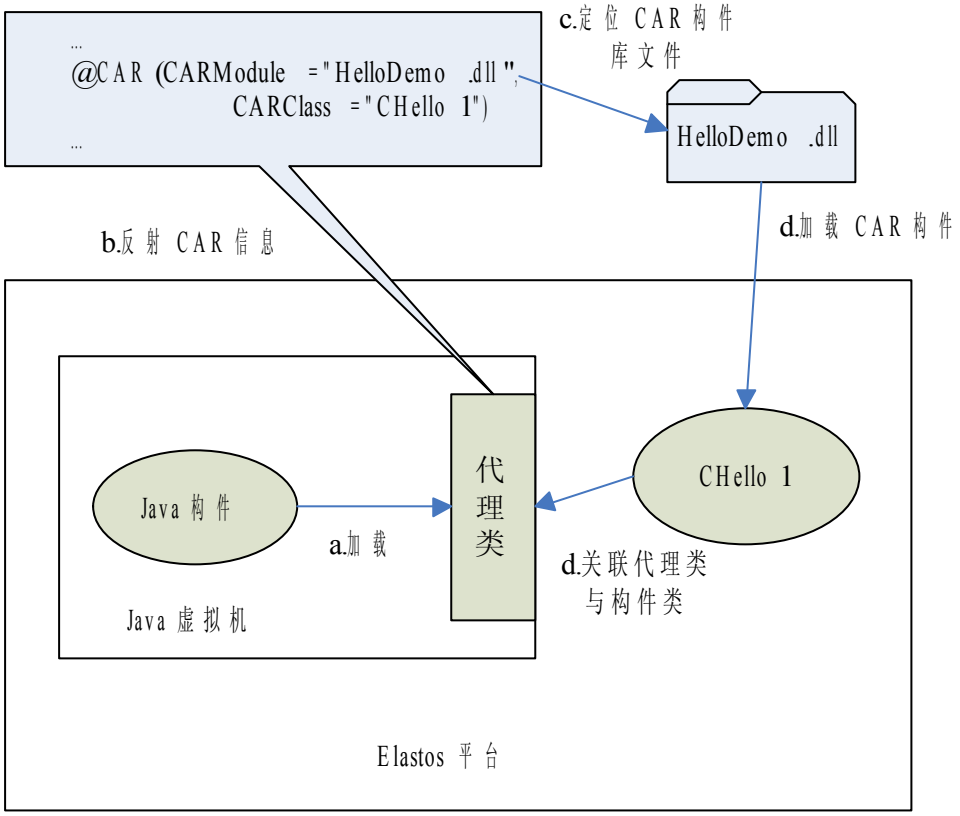

图 3.3 JNC 构件加载过程

3.4.1.1 生成同名代理类

代理类与构件类对应关系如下：

1. 对于构件中的每一个类、接口都生成相应的 Java 代理类。

2. 对于生成的每一个 Java 代理类，都属于 com.elastos.HelloDemo 包，都具有 CAR 注释

```
@CAR(CARModule="HelloDemo.dll", CARClass="CHello2")
```

并且导入相应的 dalvik.annotation.CAR 或 dalvik.annotation.CARInterface 注释类。

3. 对于构件中的每一个类，若其实现了构件中的一个接口，那么此类对应的 Java 代理类应包含 extends 关键字，并继承自此接口对应的代理类。

4. 对于构件中的每一个类、接口中的方法，生成的 Java 代理类都具有相应的 native 同名方法，并且参数类型、返回值具有一一对应关系。

例如，对于 2.1 节定义的构件类 HelloDemo 来说，其中类 CHello1 通过自动构建工具生成的代理类的定义如下：

```
package com.elastos.HelloDemo;

import dalvik.annotation.CAR;
```

```
@CAR(CARModule="HelloDemo.dll", CARClass="CHello1")
public class CHello1 implements IHello, IHey{
    public native CHello1();

    public native void Hello(int i);

    public native String Hey(int i);

}
```

部分类型对应关系如下表：

表 3.1 CAR 类型与 Java 类型对应关系表

CAR 类型	Java 类型
AChar	char
WChar	char
Byte	byte
Boolean	boolean
Int16	short
Int32	int
Int64	long
AString	String
WString	String
Float	float
Double	double
Interface	相应的 Java 代理类
Enum	相应的 Java 代理类

3.4.1.2 加载构件类

传统的 Java 虚拟机加载 Java 类时主要分为三步：装载、连接、初始化。其中，装载就是把二进制形式的 Java 类型读入 Java 虚拟机中；而连接就是把这种意境读入虚拟机的二进制型式的类型数据合并到虚拟机的运行时状态中去，又分为验证、准备、解析三个阶段；初始化就是把类变量赋以适当的初始值。

对于本文中的新型 Java 虚拟机，加载构件类的过程针对于 Java 端主要是两个部分：装载、连接，由于代理类中并不包含有类变量，因此没有初始化这个阶段。装载阶段的工作与传统的 Java 虚拟机加载过程相同，即将二进制形式的 Java 代理类读入 Java 虚拟机中；连接阶段则负责加载 CAR 构件类、CAR 构件类与

代理类的绑定，以及将代理类合并到虚拟机的运行时状态中去，也可分为验证、加载、关联、解析四个阶段。

验证阶段中，相比于传统虚拟机中对 Java 类的验证，这里的验证主要包括两个方面：CAR 构件类自身合法性的验证，以及代理类与构件类的一致性验证。

加载阶段中，主要是指对 CAR 构件以及其依赖的其他构件的加载过程。本文暂不考虑 CAR 构件依赖于 Java 构件的情况。在 Elastos 环境下，CAR 构件的加载过程中，若其依赖于其他构件，则 Elastos 平台会自动将其加载，因此对于 Java 虚拟机来讲，不需要额外考虑多级加载的问题。

关联阶段中，主要工作一个是将 CAR 构件中类、接口的引用与代理类在虚拟机中的类实例关联起来，关联方式采用增加虚拟机内类实例的属性来保存对 CAR 构件中类型的引用。另一个是将类、接口中的方法引用与代理类中的本地方法关联起来，一方面增加虚拟机内方法实例的属性来保存对 CAR 构件中方法的引用，另一方面将虚拟机中对应方法的结构体的 nativeFunc 指针指向 dvmCallCARJNIMethod 方法。

解析阶段中与传统的 Java 类的解析相同，主要是将生成好的代理类的类实例加入至虚拟机的运行时状态中。

3.4.1.3 实例化构件类

传统的 Java 虚拟机类的实例化主要是两个操作：分配内存、赋初始值。对所有在对象的类中和它的超类中声明的变量都要分配内存。

而在本文中的新兴 Java 虚拟机中，除了传统的类实例化外，主要是代理类的实例化。由于代理类都是自动生成的，根据 3.3.1 节的生成规则可知，代理类的超类（除了 Object 类）都是 CAR 构件的代理类，在 Java 运行环境中都没有声明变量，因此分配内存以及赋初始值的操作主要由 CAR 构件执行环境 Elastos 平台进行管理。而 Java 虚拟机需要做的是在实例化代理类时，同时实例化 CAR 构件类，并将 CAR 构件实例化出来的对象引用保存在代理类实例中，如果 CAR 构件类有超类，则将超类的实例也创建出来，并将引用保存至代理类实例中。

3.4.2 构件互调机制

对于两个构件之间的相互调用，有正调、回调之分，也有同步、异步之分，共有四种情况。CAR 构件技术中，默认通过 callback、delegate 等关键字实现了正调、回调、同步、异步机制。而对于 Java 语言来讲，并没有默认的回调或异步机制。

正调回调机制、同步异步机制等属于程序框架提供的能力，同一种开发语言中不同的框架也可采用不同的方式来实现互调机制。

3.4.2.1 回调机制

回调是一种双向调用模式，是指被调用方在接口被调用后也会调用对方的接口。也就是模块 A 调用模块 B，而模块 B 中又存在调用模块 A 中的一个函数 c 的代码，图 15.1 所示的形式就叫回调，其中的函数 c 就是回调函数。

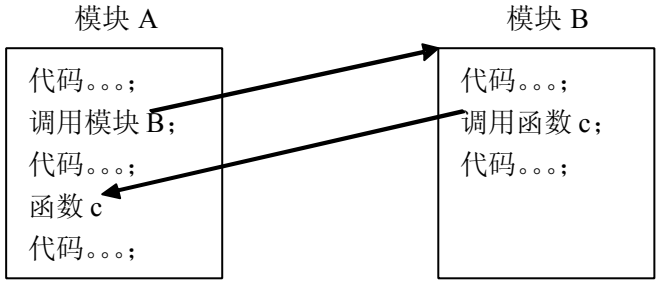

图 3.4 回调示意图

在 Java 开发中，使用最普遍的回调机制是通过接口或抽象类实现，这一点在 Android 程序开发中有很好的体现。

例如，Android 程序中进行界面开发时，需要实现对 Click 事件进行回调的功能。首先，开发人员需要构建实现 OnClickListener 接口的 MyListener 类，并实现其方法 onClick()。获取 MyListener 类的实例化对象 listener，作为回调对象，调用 View. setOnClickListener()方法进行回调注册。此时当发生 click 事件时，就会自动回调开发人员实现 listener 对象的方法 onClick()。这是一个通过接口或抽象类实现回调的典型例子，当然这种方式下若实现异步回调则需要结合 Android 系统中 Handler 来实现。

还有通过反射实现回调的例子，如 WebOS 框架。采用哈希表存储回调方法与应答值的映射。当注册回调时，将回调方法的引用与唯一生成的对应的应答值映射存储在哈希表中，并将应答值发送给回调事件发起者。当回调事件发起时，会发送带有应答值的消息至注册回调的线程的消息队列。当该线程解析消息队列获取到此消息时，线程通过应答值从哈希表中获取回调方法的引用并执行。

相比于前者，利用反射机制效率较低，需要额外的哈希表存储空间，还需要线程对消息队列的循环解析。但前者需要开发人员建立实现接口，实例化对象，开发不够便捷。

3.4.2.2 同步异步机制

对于异步机制，很多框架都是采用消息队列来实现，如 Android 系统中的 Handler、WebOS 中的 MessageThread、CAR 构件技术等。同步机制也常常利用异步机制来实现。正常的异步机制是线程 A 通过向线程 B 发送消息来实现调用，

在线程 B 回复之前，线程 A 就可以继续执行。那么实现同步就是线程 A 继续等待直到获得线程 B 的回复。

尽管都是采用消息队列来实现异步机制，但具体实现在对消息的处理上有所不同。Android 系统中的对消息的处理采用 switch 语句来实现，而 WebOS 中采用哈希表来实现。

3.4.3 构件对象的生命周期管理

对象的生命周期是指对象在内存中存活的时间段，通常包括三个阶段：对象的创建、对象的使用以及对象的销毁。这一节主要对 JNC 构件对象的创建和销毁进行分析。

3.4.3.1 对象的创建

对于一个 JNC 构件，创建对象时会产生代理类的代理对象、对应的构件对象以及父类的构件对象。这三类对象与普通 Java 构件创建对象的时机应该基本相同。

而对于 Singleton 构件类，唯一实例的创建时机可以在类加载时，也可以在第一次调用方法时。若选在类加载时，可能会发生还未被调用构件就被注销了。因此，在第一次调用方法时更为妥当。

3.4.3.2 对象的销毁

代理对象的生命周期与其他 Java 构件对象的生命周期是一致的。因此对于 JNC 机制来讲，首先需要关注的是代理对象的生命周期与对应的 CAR 构件对象（包括父类的构件对象）是否保持一致，尤其是对象的销毁。

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，若要保持代理对象与 CAR 构件对象的生命周期一致，就需要在代理对象被垃圾回收时，同时对 CAR 构件对象执行 Release 操作。

3.5 跨语言构件继承关系

本节将探讨采用 3.4 节设计的 JNC 机制是否满足构件之间的面向对象关系。

JNC 机制的初衷是采用 CAR 构件类替代部分 Java 类，以达到 Java 对象与 CAR 对象的无缝交互，进而提高 Java 程序的执行效率，由此也形成了跨语言环境的面向对象系统。面向对象系统中，继承是三个基本特性之一，如果一个类要合并另一类的特性，则这个类是其继承者[10]。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。而在不同语言环境下的两个类相互继承，就要求子类创建实例时同步在另一个语言环境中创建父类的实例，才能达到子类对象继承并正常调用父类的方法。因此，在跨语言环境中的继承机制采用方法继承，所谓方法继承是指子类不继承父类的属性，即父类没有非私有属性，只继承父类的方法或动作。采用方法继承可以避免由于子类实例与父类实例共同存在可能导致的数据不一致的问题。

3.5.1 现有继承机制概述

3.5.1.1 C++语言中的继承、多态

C++中继承的语法是"class B : public A"，B 类可以调用 A 类中的非私有成员方法。C++中多态是通过 virtual 关键字实现的，声明为 virtual 的方法可以被子类覆盖，但对于声明为 virtual 的方法并不是被真正覆盖。

代码示例如下：

```
class A{
public:
    void print_A(){
        printf("This is A.");
    }

    void print(){
        printf("This is A.");
    }

    virtual void v_print(){
        printf("This is A.");
    }
}

class B : public A{
public:
    void print(){
        printf("This is B.");
    }

    virtual void v_print(){
        printf("This is B.");
    }
}
```

执行以下代码：

```
B* bb = new B();
```

```
bb->print_A();
bb->print();
bb->v_print();
((A*) bb)->print();
((A*) bb)->v_print();
```

结果是：

```
This is A.
This is B.
This is B.
This is A.
This is B.
```

从结果可以看出，没有被声明为 virtual 的方法 print()，并未被覆盖，子类对象的指正转换为父类指针后调用的依然是父类的方法，而声明为 viirtual 的方法 print_v()则已被子类覆盖，通过最后一个语句调用可以看出。

3.5.1.2 Java 语言中的继承、多态

相比于 C++中采用 virtual 关键字来实现多态机制，Java 中多态隐含在继承实现中，容易让人混淆继承与多态的关系。

继承的语法是”class B extends A”，B 类可以调用 A 类中的非私有成员函数。若 B 类中实现了与 A 同名的非私有成员方法，则是实现了多态机制中的重载或覆盖，若未实现同名的非私有成员方法，则仅仅是继承。

代码示例如下：

```
public class A{
    public int i = 1;
    public void print(){
        System.out.println("This is A." + this.i);
    }
}

public class B extends A{
    public int i = 2;
    public void print(){
        System.out.println("This is B." + this.i);
    }
}
```

执行以下代码：

```
B b = new B();
b.print();
System.out.println("i:" + b.i);
A a = b;
a.print();
```

```
System.out.println("i:" + a.i);
```

执行结果是：

```
This is B.2
i:2
This is B.2
i:1
```

可以看到，即使改变了子类对象 b 的引用，调用 print()方法，依然是调用的子类的方法，因而实现了多态。值得注意的是 Java 里成员变量是没有多态的概念的。

Java 中增加了 super 关键字来调用父类的方法，与 this 不同，这只是通知编译器调用父类方法的关键字，而 this 是对象的引用指针。C++中是采用（父类名字::方法名）的方式来调用父类方法。

3.5.1.3 CAR 构件中的继承、多态

不同于 C++和 Java，CAR 对象的可重用性表现在 CAR 对象的包容和聚合，而不是通过继承的方式。CAR 对象中的继承是用来实现多态的，是通过 CAR 对象具有的虚接口来实现多态，类似于 C++中通过 virtual 函数来实现多态。

代码示例如下：

```
module Elastos:Sample:Animal.dll
{
    interface ISay {
      Say();
    }

    interface IAnimal {
      SetName(String name);
      GetName(StringBuf<50> name);
    }

    class CAnimal {
      virtual interface ISay;          //定义虚接口
      interface IAnimal;
    }

    class CDog inherits CAnimal {          //声明继承关系
      interface ISay;                  //继承基类的虚接口
    }

    class CCat inherits CAnimal {          //声明继承关系
      interface ISay;                  //继承基类的虚接口
    }
```

```
}
```

具体函数实现暂略去，执行代码如下：

```
ec = CCat::New(&pCat);
if (FAILED(ec)) {
    CConsole::WriteLine("Can't create CCat\n");
    return ec;
}

pAnimal = IAnimal::Probe(pDog);
pAnimal->SetName(L"Dog ");
pDog->Say();
pAnimal = IAnimal::Probe(pCat);
pAnimal->SetName(L"Cat ");
pCat->Say();
```

执行结果如下：

```
Dog say : Wang Wang~~~
Cat say : Miao Miao~~~
```

CAR 对象中继承的关键字是 inherits，继承的基本要求是父类定义中含有虚接口，即 virtual interface 的定义，父类、子类都具有虚接口 ISay 的实现。示例中，子类 CDog 和 CCat 可以调用父类 CAnimal 的 SetName 和 GetName 两个方法，并分别各自覆盖了 ISay 这个虚接口的方法，实现了多态的机制。

3.5.2 跨语言的方法继承机制

继承、多态两个概念在不同的面向对象系统中有不同的解释，在 Java 语言和 CAR 构件这两个对象系统中也有不同的使用方式。Java 语言以面向对象为中心，继承用来实现可重用性和扩展性（多态），多态继承与普通继承语法上无明显区别；CAR 构件的设计以构件为中心，不同于其他面向对象语言的设计，构件的可重用性不依赖继承实现，而是采用包容和聚合实现，构件之间的继承关系用来实现多态，通过虚接口实现。

在 JNC 机制下，对于具有继承关系的三个类：Java 类与两个具有继承关系的 CAR 构件类的继承关系如图 3.5 所示：

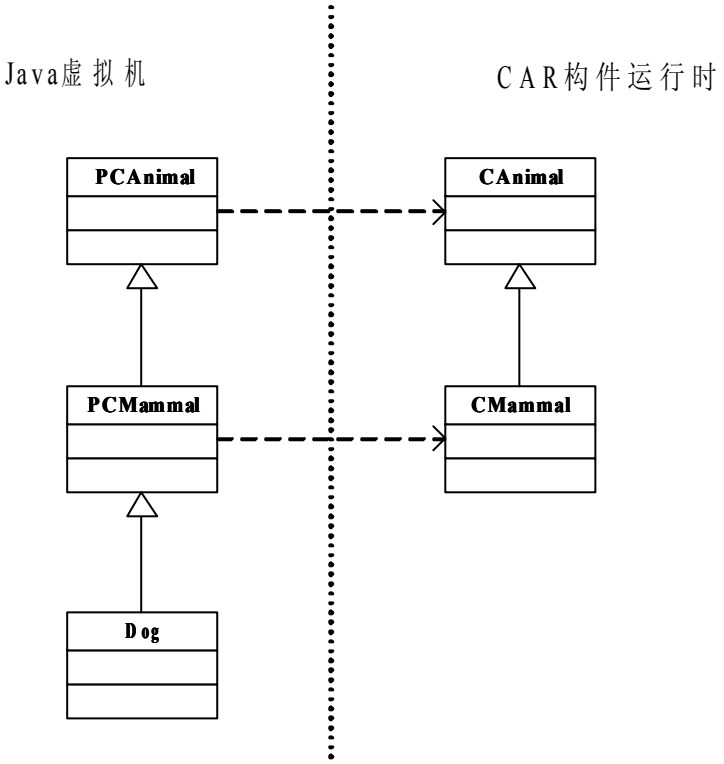

图 3.5 跨语言构件类继承关系示例

类 PCAnimal、类 PCMammal（为便于区分，用以 P 开头代表是代理类，实际生成代码中类名中不包含 P）分别是类 CAnimal 与类 CMammal 在 Java 环境中的代理类，在 Java 环境中通过创建代理类对象以及对此对象的使用来达到对 CAR 构件的调用的目的，其实现机制是在创建代理类 PCAnimal 的实例 panimal 时，同时创建类 CAnimal 的实例 animal，并将 panimal 的方法执行与 animal 方法映射起来，由此形成了代理关系，在 Java 语言环境下执行代理实例的动作就会执行构件实例的动作。

继承类的实例化原理与代理类的实现机制相同，在创建子类 Dog 的实例 dog 时，同时创建其所有是代理类的父类对应的 CAR 构件实例，并将子类实例 dog 从父类（CAR 构件类）继承而来的动作与 CAR 构件实例映射，由此形成继承机制。如图 3.6 所示：

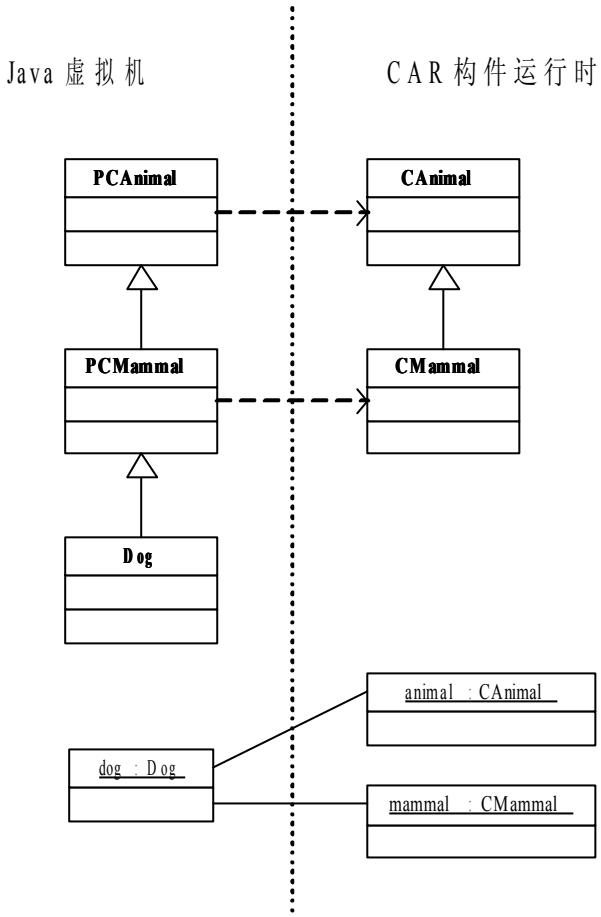

图 3.6 跨语言构件继承实现示意图

第 4 章 JNC 机制的实现

实现主要分为四部分，包括生成同名代理类、加载构件类、实例化构件类、执行构件方法。

4.1 生成同名代理类

通过自动生成工具 CAR2Java 实现，程序主体包括三个文件：carc.exe、cls2java.exe、elasys.cls（carc.exe 执行时依赖此文件）。默认运行平台是 Windows 平台，在 Linux 平台下，请在命令前加 wine 执行。

4.1.1 工具使用方法

1． 对于模板文件(.car)，需要两步。

A. 用 carc 工具编译校验 car 文件并生成样式库文件(.cls)。

```
carc –c {name}.cls {name}.car
```

B. 用 cls2java 工具依据样式库文件(.cls)生成 Java 文件(.java)。

```
cls2java {name}.cls
```

2． 对于样式库文件(.cls)或构件(.dll)，只需要一步，

```
cls2java {name}[.cls|.dll]
```

4.1.2 工具实现原理

CLS 文件是 CAR 文件压缩后的样式库，里面描述了整个构件的接口、类、方法、结构、枚举等的定义。有了 cls 文件，就可以得到该构件的接口，函数的调用方法，甚至生成源码框架。构件(.dll)中在生成时已加入 cls，cls 文件就相当于构件的说明书。

通过从 CLS 读取到整个构件的定义 CLSModule，将构件中的类、接口、方法、结构体、枚举等类型与 Java 中的按 CAR 调用 Java 的原则进行转换，进而生成 Java 文件。

这里以生成类作为举例说明实现原理：

遍历 CLSModule 中的类数组 ppClassDir，每一类生成对应的 Java 类，此 Java 类与原类名称相同，Java 类的 annotation 格式为 @CAR(Module=”{ModuleName}.dll”, CARClass=”{ClassName}”)，Java 类实现所有原类实现的

接口，包含的方法与原类方法、构造方法的合集相对应，构造方法名称与类名相同，构造方法没有返回值，所有方法的参数为原类方法中的类型为 in 的所有参数，非构造方法返回值与原接口方法第一个 out 类型的参数类型相同，其他 out 类型的参数将被忽略。生成的代理类实例如图 4.1 所示：

```
// CHello_1.java ①
package com.elastos.HelloDemo; ②

import dalvik.annotation.CAR; ③

@CAR (CARModule = "HelloDemo.dll", CARClass = "CHello_1") ④ ⑤
public class CHello_1 implements IHello, IHey { ⑥
  public native CHello_1(); ⑦

  public native void Hello(int i); ⑧

  public native String Hey(int i);

}
```

图 4.1 代理类生成过程示例

其中，

① 代理类名称，与其对应的构件类名称相同；

② 代理类所在包路径，都采用“com.elastos.构件名”的格式；

③ 为了支持 CAR Annotation，需要导入 dalvik.annotation.CAR 注释类，是 JNC 机制为了提供 CAR Annotation 的支持而增加的类；

④ CAR 注释中包含 CARModule 项，其值应设为代理类对应的构件类所在构件的文件名；

⑤ CAR 注释中包含 CARClass 项，其值应设为代理类对应的构件类名称，若生成的 Interface，则由 CARInterface 项表示，值为对应的构件接口名称；

⑥ 对于构件类实现的接口，在代理类中都需同样加入 implements 关键字，并加上构件接口对应的同名代理接口；

⑦ 对于构造方法，默认均为 public，都是 native 方法，名称设为与代理类名称相同，参数类型根据表 3.1 所示的对应关系进行转换，参数名称都可通过反射从 CAR 构件中获取；

⑧ 普通方法与构造方法相同，方法名也可通过反射从 CAR 构件中获取。

以上八步就是代理类生成的基本步骤，对于静态类（CAR 构件中成为 singleton）、静态方法等，也都会进行相应的设置。

4.2 加载构件类

4.2.1 类结构体定义

由于引入了构件代理类，因此对 vm\oo\Object.h 文件中定义的结构体进行了相应的修改。

首先增加了三个 ClassFlag 用来表示该类是 CAR 构件的代理类。

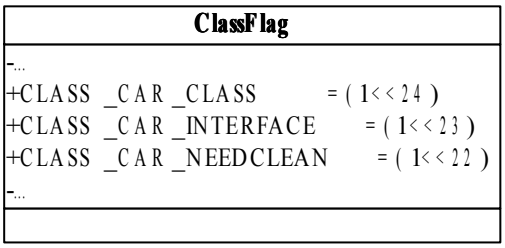

图 4.2 枚举类型 ClassFlag 增加项

其中 CLASS_CAR_CLASS 表示代理类对应的是一个 CAR 构件类，CLASS_CAR_INTERFACE 表示代理类对应的是一个 CAR 构件接口，CLASS_CAR_NEEDCLEAN 代理类对应的一个需要清理的 CAR 构件接口。

其次，对虚拟机中表示类实例的结构体 ClassObject 进行了修改，在结构体最后位置增加了四个指针，用于指向 CAR 构件的相关引用，分别是构件类引用、构造函数引用、静态类引用以及类的唯一实例的引用。

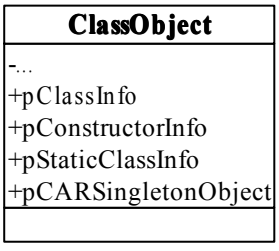

图 4.3 结构体 ClassObject 增加项

其中，pClassInfo（pStaticClassInfo）指向代理类对应的构件类（静态构件类）信息引用，pConstructorInfo 指向代理类对应的构件类的构造函数信息引用，若构件类是静态类，则 pCARSingletonObject 指向其唯一实例。

增加一个结构体 CARMethodInfo 用来表示 CAR 构件中的方法。

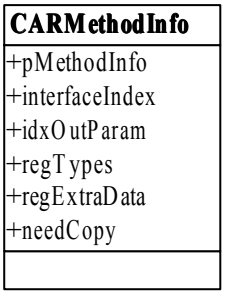

图 4.4 结构体 CARMethodInfo

对方法结构体 Method，增加了一个指针指向 CAR 构件方法结构体。

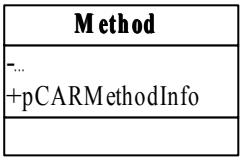

图 4.5 结构体 Method 增加项

4.2.2 加载关联阶段

加载构件类的过程可细分为五个阶段装载、验证、加载、关联、解析。其中装载阶段、解析阶段与传统的 Java 类加载过程相同，验证阶段主要是对 CAR 构件的合法性验证，此处主要介绍加载、关联两个阶段的实现。

传统的 Java 类加载是通过 vm\oo\Class.c 文件中 findClassNoInit()函数来实现的，其函数定义如下：

```
static ClassObject* findClassNoInit(const char* descriptor, Object* loader,
    DvmDex* pDvmDex)
```

当虚拟机发现一个类尚未加载时，将通过调用 findClassNoInit()函数，在系统路径下查找所有包中是否包含此类，若找到，则加载相应的 dex 文件，并调用 vm\oo\Class.c 文件中的函数 loadClassFromDex()，从 dex 文件中加载来至内存中。详细调用函数调用情况如图 4.6 所示：

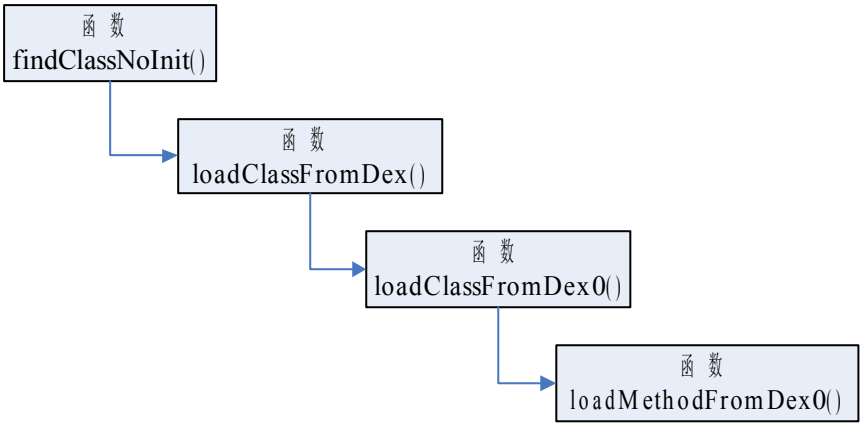

图 4.6 传统类加载函数调用图

对于构件类加载过程中，当调用到 loadClassFromDex0()函数时，代理类的字节码文件已经读入虚拟机内存中，即装载过程已经结束。

此时进入连接中的加载阶段，此时增加判断此类是否是代理类的语句，并相应的进行 CAR 构件加载操作。插入的判断语句流程图如图 4.7：

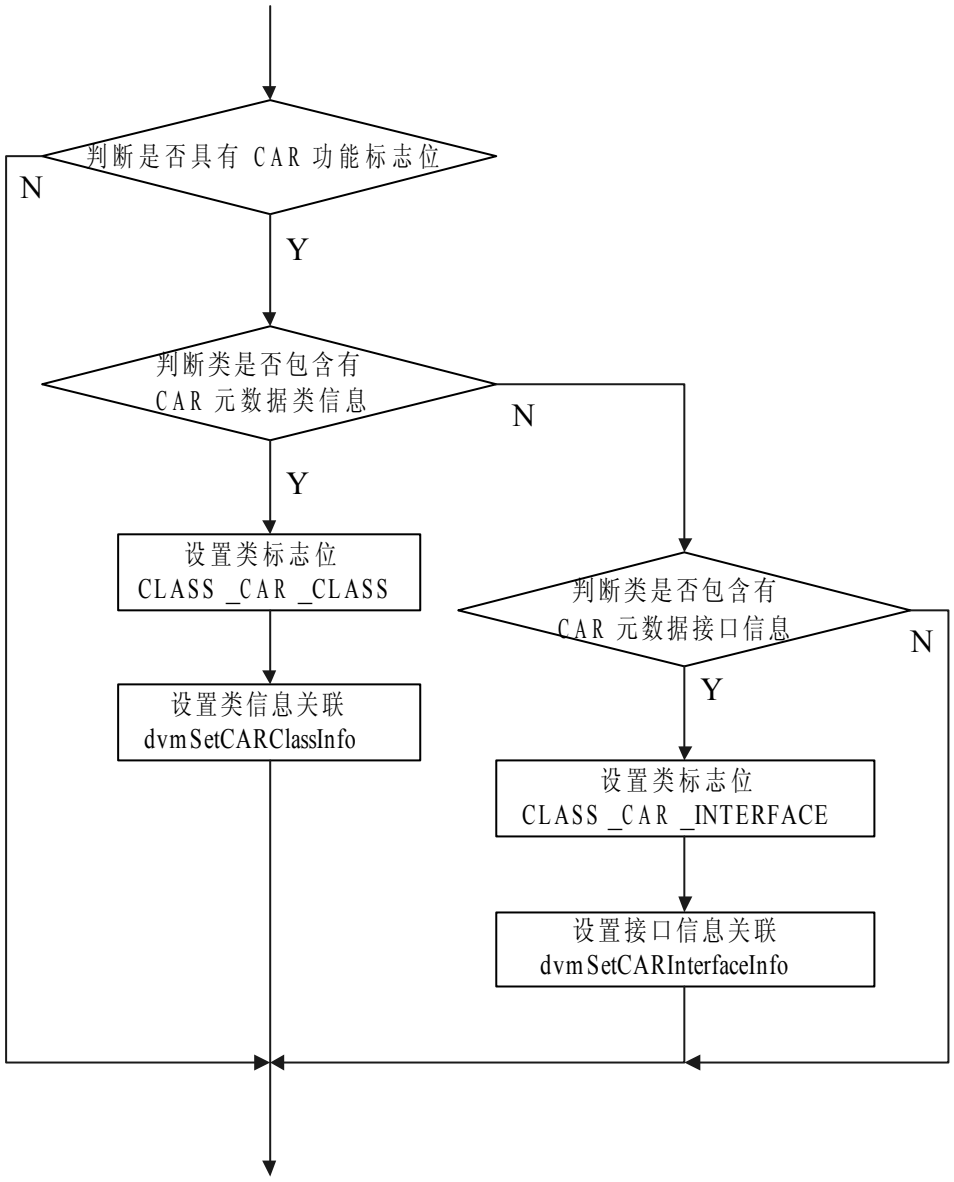

图 4.7 加载代理类判断语句流程图

从图 4.7 中可知，若此类是 CAR 构件类或接口对应的代理类，则执行函数 dvmSetCARClassInfo()或 dvmSetCARInterfaceInfo()进行 CAR 构件类的加载、关联操作。

dvmSetCARClassInfo()的主要实现如下：

首先从代理类中读取相应注释的信息，即获取其对应的构件名及类名。

```
char* CARModule = dvmGetCARClassAnnotationValue(clazz, (char*)"Module");
char* CARClass = dvmGetCARClassAnnotationValue(clazz, (char*)"Class");
```

然后，从虚拟机加载本地库的哈希表中查找是否相应的 CAR 构件已经加载，若没加载则加载构件对应的 DLL 文件。

```
dvmHashTableLock(gDvm.nativeLibs);
```

```
    pModuleInfo = (IModuleInfo*)findHashCARModuleInfo(CARModule, kAddRefCount,
NULL);
    if (pModuleInfo == NULL) {
        ec = _CReflector_AcquireModuleInfo(CARModule, &pModuleInfo);
        if (FAILED(ec)) {
            LOGD("_CReflector_AcquireModuleInfo Failed");
            dvmHashTableUnlock(gDvm.nativeLibs);
            return false;
        }
        if (addHashCARModuleInfo(CARModule, pModuleInfo) == NULL) {
            LOGD("add hash _CReflector_AcquireModuleInfo Failed");
            pModuleInfo->Release();
        }
    }
    dvmHashTableUnlock(gDvm.nativeLibs);
```

加载构件成功后，进入关联阶段。

从构件中 IModuleInfo 中读取出对应代理类的类信息指针 pClassInfo 和构造函数信息指针 pConstructorInfo，并将两个指针保存至虚拟机类实例中的 pClassInfo、pConstructorInfo 变量中。代码如下：

```
    if (CARClass != NULL && *CARClass != '\0') {
        ec = pModuleInfo->GetClassInfo(CARClass, &pClassInfo);
        LOGD("pClassInfo %p", pClassInfo);
        if (FAILED(ec)) {
            LOGD("GetClassInfo Failed");
            return false;
        }

        clazz->pClassInfo = (u4)pClassInfo;
        ec = pClassInfo->GetConstructorInfoByParamCount(3, (IConstructorInfo
**)&(clazz->pConstructorInfo));
        if (FAILED(ec)) {
            clazz->pConstructorInfo = NULL;
        }
    } else {
        clazz->pClassInfo = NULL;
    }
```

若代理类对应的构件类是只有唯一实例的类，即 Singleton，则同时还要创建这个唯一实例并将其指针保存至类实例中的 pCARSingletonObject 变量中。

关联类信息后，接下来对代理类中的每一个方法与 CAR 构件类的方法进行关联。

首先通过 loadMethodFromDex()函数中的语句进行本地函数关联，设置方法

结构体的 nativeFunc 指针指向 dvmResolveNativeMethod。

最后通过执行 dvmSetCARmethodInfoForJavaMethod()函数将 CAR 构件类中方法信息指针与代理类相应的方法进行关联，即将 CAR 构件的方法信息指针保存在代理类对应的方法结构体的 pCARMethodInfo 变量中。

4.3 实例化构件类

虚拟机实例化类的操作是通过 vm\alloc\Alloc.c 文件中的 dvmAllocObject()方法完成的，其定义如下：

```
Object* dvmAllocObject(ClassObject* clazz, int flags)
```

正如节 3.3.3 所说明的，对于构件类实例化的执行最重要的是对其所有超类都建立相应的构件对象，并保存在代理类实例化的对象结构体中。

首先，查找该构件类的超类的数目，

然后，根据计算出来的超类数目分配相应的空间，

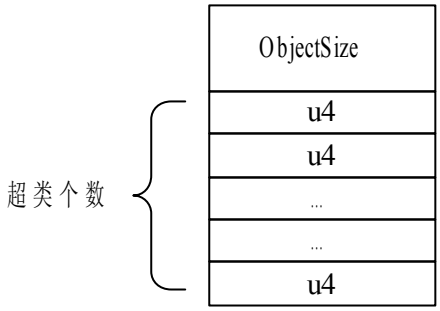

图 4.8 对象分配内存大小示意图

最后对每一个超类创建实例化相应的对象，并将指针保存在 newObj 对象结构体中

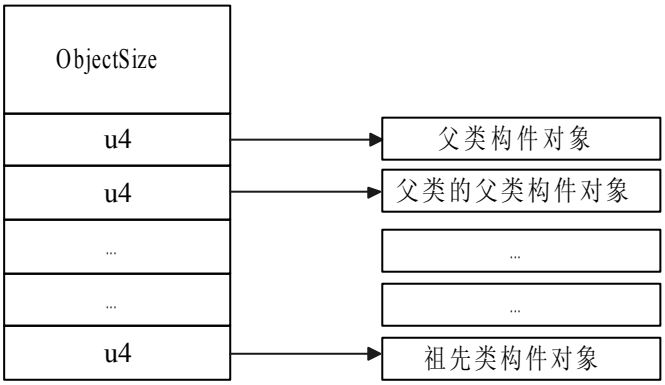

图 4.9 子类对象与父类对象示意图

4.4 调用构件方法

Java 虚拟机的方法调用是通过 vm\interp\Stack.c 文件中的 dvmCallMethod() 系列方法实现的，对于本地方法调用则总是调用方法结构体中 nativeFunc 函数指针指向的函数。

若方法是第一次被调用，参考第 4.1.2.2 节可知，此时 nativeFunc 指针指向 vm\Native.c 文件中的 dvmResolveNativeMethod() 函数，其定义如下：

```
void dvmResolveNativeMethod(const u4* args, JValue* pResult,
    const Method* method, Thread* self)
```

对于 CAR 构件类的方法调用，dvmResolveNativeMethod() 函数主要操作是对 CAR 构件方法进行解析，其执行过程如下：

由于代理类对应的构件来可能有多个超类，所以需要首先找到该方法对应的构件类，并获得其在代理类实例中保存的构件对象指针。

```
Object* object = (Object*)args[0];
int i;
ClassObject* clz;
//find my CAR object
for (i = 0, clz = object->clazz;  clz != method->clazz && clz != NULL;  clz = clz->super)
{
    if (IS_CLASS_FLAG_SET(clz, CLASS_CAR_CLASS))
        i++;
}
pCARObject = *(u4*)(((char*)object)+clazz->objectSize + sizeof(u4)*i);
```

通过获得的构件对象指针获取相应的构件方法指针，这通过 dvmGetCARMethodAddr() 函数实现，其实现体是：

```
void *dvmGetCARMethodAddr(Method* meth, u4 *pCARObject)
{
    LOGD("dvmGetCARMethodAddr");
    VObject* vobj = (VObject*)(pCARObject);
    CMethodInfo *pMethodInfo;
    pMethodInfo = (CMethodInfo*)meth->pCARMethodInfo->pMethodInfo;
    UInt32 index = pMethodInfo->m_uIndex;
    vobj += meth->pCARMethodInfo->interfaceIndex;

    void* pMethodAddr = vobj->vtab->methods[METHOD_INDEX(index)];
    return pMethodAddr;
}
```

最后，将该方法结构体 Method 的 nativeFunc 指针指向 dvmCallCARJNIMethod() 函数或 dvmCallSynchronizedCARJNIMethod() 函数，并调用 dvmCallCARJNIMethod() 方法继续执行此构件方法。由于 nativeFunc 指针改

为指向 dvmCallCARJNIMethod()函数，因此之后每次调用此构件方法都跳过方法解析，直接进入方法执行。

dvmCallCARJNIMethod()函数的主要工作是调用 Elastos 平台接口执行构件方法，并对返回值进行数据转换。

4.5 基础类库的构件化

类库（Base Libraries）是指 Java 语言提供的已经具有标准实现的类的集合。基础类库提供了包括基本数据类型、网络、数学计算、文件处理、日志、并发编程、安全性等常用功能。这些功能都是以类的形式进行加载，以类的 API 的形式提供给用户使用。对于涉及到本地功能的 API，将通过 JNI 的方式进行实现。

Java 虚拟机中类库可分为核心类库和基础类库两类。核心类库主要是指包 Lang 和包 Util 中的类的集合，该类库提供了基本数据类型、日期、文件操作、反射、集合等虚拟机运行依赖的核心类，是满足 Java 程序运行的基本条件。基础类库是指 Java 规范中规定的不包含再核心类库中的类的集合[17]。Dalvik 虚拟机的结构图[17]显示了对 Java 基础类库的支持情况，如图 4.10：

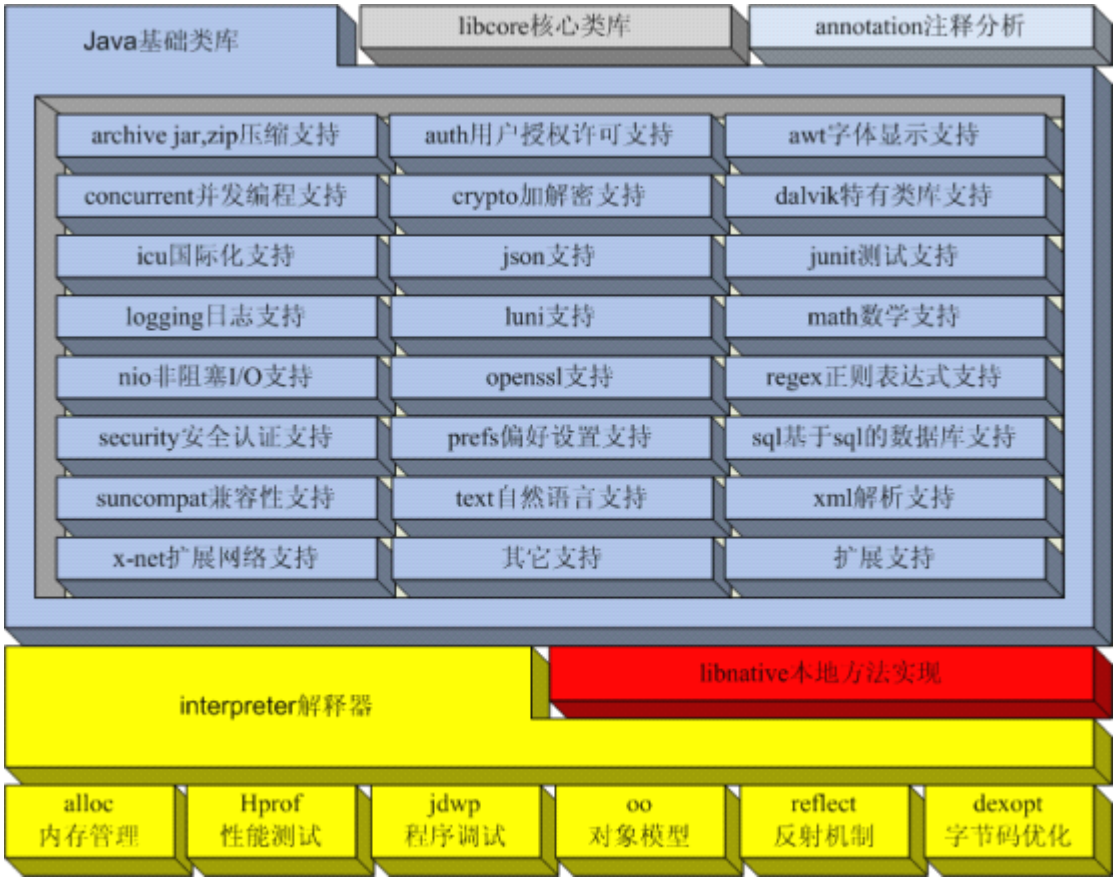

图 4.10 Dalvik 结构图

由图 4.10 可知，Dalvik 虚拟机中共提供了二十多个基础类库功能包，但并非所有的功能包都依赖本地功能。依赖本地功能的包主要是 archive、dalvik、icu、luni-kernel、luni、nio、openssl、sql、x-net、xml 包。这些包的主要功能是：

表 4.1 Java 虚拟机中依赖本地功能的包列表

功能包名	功能简述
archive	提供对 Jar 文件的读写操作，以及文件压缩、解压缩操作
dalvik	提供 Dalvik 虚拟机特有的功能，包括特有的注释、字节码等。
icu	提供国际化支持
luni-kernel luni	提供核心类库的支持
nio	提供数据缓存流以及输入输出通道的支持
openssl	提供数据的安全加密支持
sql	提供基本的数据库支持
x-net	提供网络开发的支持
xml	提供 xml 解析相关功能的支持

基础类库的构件化是指对于依赖本地功能的基础类库采用 JNC 机制实现，即对于目前采用 JNI 实现的基础类库，改为采用 JNC 机制实现，这样可以使得虚拟机自身功能按需加载。如网络、数据库、xml 解析等功能，并不是所有构件都依赖这些功能的实现，因此可以将其通过 JNC 机制包装成构件，使得这些功能按需加载。而对于 luni 核心类库以及 icu 这些必要的功能包封装成构件则能提供虚拟机的动态更新核心模块的能力。

4.6 JNC 机制的角色

4.6.1 JNC 机制的角色

Elastos 是由构件相互组装而成的平台，通过 Dalvik 虚拟机以及 Webkit 浏览器的引入[17]，增加了对 Java 语言和 JavaScript 脚本语言的支持，也同时对 Java 构件提供了支持。而 JNC 机制的提出，则提供了 CAR 构件与 Java 构件相互调用的能力。

如果把构件比喻成零件，那么构件与构件之间的组装则是通过接口来完成。由于两个构件的接口不一定完全相符，因此就需要接口适配层来完成接口的转换。JNC 机制就是这种接口适配层。

JNC 机制通过将 CAR 构件接口转换为 Java 构件的接口，并保证了 CAR 构

件在 Java 环境中的正常运行，进而提供了 CAR 构件与 Java 构件相互组装的能力。

4.6.2 Java 语言的角色

当前在 Elastos 平台中主要有三类编程语言，CAR（C\C++语言）、Java 语言、脚本语言（如 Lua、XmlGlue、JavaScript 等）。这三种语言也代表着当前主流的开发语言类型。

CAR（C\C++语言）属于编译型静态语言，如 C\C++、ObjectC、Pascal 等。最大特点是生成目标代码、具有很高的执行效率，但平台一致性差。

Java 语言属于解释型语言，尽管需要编译，但不生成机器目标代码，而是生成字节码文件通过虚拟机来解释执行。由于采用垃圾收集机制，使得开发人员可以不必关注对内存的管理，因此提高了开发效率。同时还具有很高的平台无关性和安全性，这些特点使得 Java 语言成为目前使用人数最多的编程语言。尽管通过采用“即时编译”（just-in-time compiler）机制，Java 语言的执行速度有了很大提高，但 Java 语言的性能依然大大落后于本地编译的 C++程序。随着最近虚拟机技术的发展，如适应性优化等技术，Java 程序的总体执行速度已经能够与本地编译的 C++程序相媲美[12]。

前两种语言都是需要进行编译，而大部分脚本语言都是动态语言，采用解释执行。这几年来，脚本语言由于其非常高的灵活性广受关注，尤其是 Javascript。随着 Web2.0 时代的来临，受互联网发展的推动，作为唯一能够直接操作网页对象的语言，JavaScript 有了很广泛的应用。而且随着前 Palm 公司的 WebOS 手机操作系统的推出，也证明了 JavaScript 语言可以成为系统框架的承担者。

这三类语言各有自己的特点，对于构件技术平台 Elastos，主要由两部分构成，一个功能构件，另一个是构件组装者。对于功能构件来讲，要求的是功能完善、执行效率高，这种角色 CAR 构件更为适合。而对于构件组装者，要求的是具有较高的灵活、小巧的特点，相比较而言，脚本语言更为适合这种角色。那么既然“构件+脚本”已经具有良好的能力，那么作为中间者的 Java 语言可以起到什么样的作用呢？

本文认为，Java 语言在发展中对 Elastos 平台起到的角色主要有三个方面：

1. 脚本语言的执行载体。由于 Java 语言具有非常广泛的平台移植能力、并且执行效率也在逐渐提高，因而很多新型的动态语言都选择 Java 语言来实现其解释器，其中尤其以 JRuby、JPython、Groovy 为代表。因此，提供了对 Java 语言的支持也就提供了对很多动态语言的支持，并且随着 JNC 机制的提出，使得这些动态语言也便于对 CAR 构件的组装。

2. 开发语言。较好的开发体验、较高的开发效率也是 Java 语言的一大特点。

动态语言尽管十分灵活，语法规则较宽松，但这也使得很多程序逻辑问题只有在程序执行时才可以发现，增加了测试成本和测试难度。因此，如果使用 Java 作为开发语言、脚本作为执行语言的思路，可以很好的改进脚本语言开发效率地下的问题。GWT(Google Web Toolkit)的出现正是解决了这个问题，这种思路也同样可以是用在 Elastos 平台之上。

3. 开发者范围。由于 Java 语言具有数目最庞大的程序开发者，因而对于任何一个平台都不应该忽视 Java 语言的存在。一个平台上开发者的数目也影响着平台程序的数目和用户的体验，进而影响平台的成功。这一点在苹果公司的 iOS 平台体现的最为明显。App Store 的火热与 iPhone、iPad 的火热是相互影响的。在一定程度上，平台的成功不仅仅是对用户的争取，尤其在最初阶段，是对开发者的争取。

第 5 章 总结与展望

经过一年多的工作和努力，Java 虚拟机与构件技术的结合终于有了目前阶段性的成果。在这期间，总共完成了以下几个工作：

1）完善 Elastos 平台上 Java 虚拟机的实现，在此基础上顺利移植了 OSGi 框架 Felix；

2）Java 构件框架 OSGi 的顺利移植推动了对 CAR 构件与 Java 构件互调的设计工作，命名为“Java Native Component”；

3）在 Elastos 平台上根据之前的设计对 Java 虚拟机进行了改造，完成了 JNC 机制的实现；

4）实现了自动生成代理类的 CAR2Java 工具，而且完善了 JNC 机制中基本数据类型的转换以及 CAR 构件垃圾收集的功能。

笔者参与了 JNC 机制的设计与实现工作，也参与了前期 Java 虚拟机的移植与完善的工作。通过阅读很多技术资料、文档、源代码以及与同事的讨论，形成了对 JNC 机制的具体设计思路，并参与编写、修改大量源代码，举行了数次讲座和讨论会，进而最终完成了 JNC 机制的实现。

但目前 JNC 机制还不够完善，主要有以下几个方面：

1）对 CAR 构件特性的支持还不足，包括回调、消息传递、聚合等功能还不够完善。

2）基础类库的构件化工作设计还不够完善，对于虚拟机核心类库的构件化还没有完整的设计。

3）对 CAR 构件、Java 构件共同组成的跨语言构件生态环境未提供统一的构件管理框架。

4）对 CAR 构件调用 Java 构件的支持还不足。

以上几个方面在未来的工作中还要继续开展，对于 JNC 机制的应用还要进行推广。

总之，Java 虚拟机的构件化课题已经取得了阶段性的成果，达到了预定的目标，但是，还有大量后续的研究和工作需要展开，希望本人的工作能够对移动设备平台的软件构件技术的发展有一定的贡献。

致谢

在本课题的研究、开展过程中以及整个研究生学习阶段，我首先要感谢我的导师陈榕教授。他在平时的工作中的言传身教、做事的态度，让我受益匪浅。不仅让我对自己的研究方向有了清醒的认识，也让我对整个行业以及社会工作有了更深入的理解，更让我懂得了做人做事要有坚持、认真的态度。

我还要感谢裴喜龙老师、顾伟楠老师，从进入同济校门开始一直到现在，老师一直对我谆谆教导，不仅仅亲身教给了我很多学习、开发的经验，也给予了很多参与项目学习实践的机会，让我在实习期间学习能力、工作能力上有了很大的提高。老师在我参与项目期间对我的肯定让我更加有信心面对困难。同时，在本课题的开展和论文的撰写过程中，两位老师花费大量时间悉心指导，给予了我很大的帮助。

我还要真诚感谢顾伟涛、马骁夫、陈国栋等好朋友在这两年多的时间里在学习、生活上对我的帮助，感谢周毅敏、徐凡、蒋章恺学长对于我提出的问题耐心的解答。感谢沈洁在我经历困难时对我的安慰与鼓励，以及在六年多的大学成长过程中对我的影响。

最后，感谢陈卫伍、盛泽昀等同届同学，和他们一起学习讨论的过程很愉快，也让我收获很多。真诚感谢共同参与项目的各位同事，大家的共同努力、合作才能取得项目的成功，也使我的课题能够顺利完成。

2011 年 3 月

参考文献

[1] Chen Rong. The application of middleware technology in embedded OS. in:6th Workshop on Embedded System, In conjunction with the ICYCS, Hangzhou, P R China, 2001
[2] Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, 2007
[3] Rob Gordon. Essential JNI: Java Native Interface, Ph/Ptr Essential Series, 1998
[4] Liang, S. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, Reading, 1999
[5] Jason Baker , Wilson C. Hsieh, Maya: multiple-dispatch syntax extension in Java, Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, June 17-19, 2002, Berlin, Germany
[6] Toddfast, The JNA Project: Java Native Access, URL: https://jna.dev.java.net/, January 2011
[7] Marc Denty, The JNative Project: Java to native interface, URL: http://jnative.sf.net, 2006
[8] Johann Burkard, The NativeCall Project, URL: http://nativecall.sourceforge.net, July 9th, 2004
[9] B. Meyer, Object-Oriented Software Construction[M], Prentice Hall International, 1988.
[10] Ian Joyner, A Critique of C++ and Programming Language Trends of the 1990s[R], Unisys – ACUS, 1996.
[11] 陈榕, 刘艺平. 技术报告: 基于构件、中间件的因特网操作系统及跨操作系统的构件、中间件运行平台(863 课题技术鉴定文件), 2003
[12] Bill Venners，深入 java 虚拟机(第 2 版)，曹晓钢，蒋靖．机械工业出版社，2003
[13] 上海科泰世纪科技有限公司．Elastos 资料大全．上海：科泰世纪有限公司，2010
[14] 潘爱民．COM 原理与应用．清华大学出版社, 1999
[15] 陈榕.构件自描述封装方法及运行的方法．中国专利：1514361．2004-07-21
[16] 张久安. CAR 构件与 Java 构件的互调机制研究与开发：[硕士学位论文]. 上海：同济大学
[17] 周毅敏. Java 虚拟机的移植与基于 CAR 构件的二次开发：[硕士学位论文]. 上海：同济大学
[18] 陈果. 基于 CAR 构件的元数据计算的研究：[硕士学位论文]. 上海：同济大学
[19] 陈俞飞. CAR 构件虚拟机(Bonsai)的研究与实现：[硕士学位论文]. 上海：同济大学
[20] 李如豹，刚冬梅等译. Java2 核心技术卷 1:基础知识(原书第 7 版) [M].北京:机械工业出版社.2006
[21] Google, The GWT Project: Google Web Toolkit (GWT 2.1), URL: http://code.google.com/webtoolkit/, December 2010
[22] The OSGi Alliance. OSGi Service Platform Core Specification. Version 4.2. June 2009
[23] P Dobrev et al., "Device and Service Discovery inHome Networks with OSGi", IEEE Commun. Mag.Vol. 40, No. 8, Aug. 2002, pp 86-92
[24] Microsoft Corporation. The Component Object Model Specification. URL:http://www.microsoft.com/com/resources/comdocs.asp
[25] Google Inc. Android Documentation［EB/OL］. http://code.google.com/intl/en/android/

Documentation.html 2007
[26] Sun Microsystems Inc. Java ME Technology API Documentation ［EB/OL］. http://java.sun.com/javame/reference/apis.jsp 2007
[27] The Java™ Virtual Machine Specification. http://java.sun.com/docs/books/jvms/
[28] Patrick Chan, Doug Kramer, Rosanna Lee. The Java Class Libraries Volume 1: java.io, java.lang, java.math, java.net, java.text, java.util, 2nd edition., Addison-Wesley Longman Publishing Co., Inc., 1998
[29] Liang Zaoqing. Process View of Reflection Mechanism for Reuse Software Architecture. 武汉大学学报：自然科学英文版, 2007, Vol.12(3)：431-436
[30] 王立冬，张凯. Java 虚拟机分析. 北京理工大学学报，2002 年 2 月，第 22 卷，第 1 期: 60-63
[31] 余法红．Java 环境下 COM 的调用研究与实现. 计算机时代，2007．Vol．6：39-40
[32] 姚昱旻 刘卫国．Android 的架构与应用开发研究［J］．计算机系统应用，2008 17(11) : 110-112

个人简历、在读期间发表的学术论文与研究成果

个人简历：

王建民，男，1987 年 1 月生。

2008 年 6 月毕业于南京审计学院 计算机科学与技术专业 获学士学位。

2008 年 9 月考入同济大学电子与信息工程学院计算机软件与理论专业攻读硕士学位。

发表论文：

[1] 王建民，陈卫伍，陈榕.基于构件技术的 Widget 本地扩展模型及实现.电脑知识与技术.2010 年 10 月

研究成果：

参与核高基重大专项："智能手机嵌入式软件平台研发及产业化"的研发并取得一定成绩。