同济大学
TONGJI UNIVERSITY

硕士学位论文

Android 中 JAVA 虚拟机模型研究与优化

姓　　名：陈卫伍

学　　号：0820080233

所在院系：电子与信息工程学院计算机系

学科门类：计算机科学与技术

学科专业：计算机软件与理论

指导教师：陈榕 教授

副指导教师：顾伟楠 教授

二〇一一年一月

同濟大學
TONGJI UNIVERSITY

A dissertation submitted to

Tongji University in conformity with the requirements for

the degree of Master of Computer Software and Theory

The Research and Optimization of JAVA Virtual Machine on Android

(Supported by the Natural 863 Plan, Grant No.2001AA113400)

Candidate: Weiwu Chen

Student Number: 0820080233

School/Department: School of Electronics and Information Engineering

Discipline: Computer Science and Technology

Major: Computer Software and Theory

Supervisor: Prof. Rong Chen

Associate Supervisor: Prof. Weinan Gu

January, 2010

ANDROID中JAVA虚拟机模型研究与优化　陈卫伍　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

Java 编程语言通过不同平台的 Java 虚拟机实现，能够实现跨平台，是事实上的网络语言。作为最流行的编程语言之一，Java 还具有简单，面向对象，稳定，多线程，动态性等优点。正因为如此，Google 推出的 Android 操作系统特别实现了一个针对嵌入式环境的 Java 虚拟机 Dalvik 来支持 Java 编程。

但是 Java 程序需要编译成字节码然后通过虚拟机解释执行，运行速度远远不如编译执行的二进制代码。虽然有一些办法可以提高 Java 程序的运行速度，例如使用本地方法来代替 Java 方法的 JNI 技术，即时编译技术以及本地编译技术，但是这些技术要么丧失了 Java 程序可以跨平台的特性，要么需要大量的额外内存，都在存在各自的局限性。

另一方面科泰世纪公司的 CAR（Component Assembly Runtime）构件技术是一种二进制组件，可以达到本地程序的运行效率，同时它通过 Elastos 构件运行环境运行在多种平台之上，实现了跨平台。同时其还具有自描述、动态加载、动态组装、二进制继承等特性，CAR 构件的这些性质为基于 CAR 构件来优化 Java 虚拟机模型提供了契机。

本文在 Java 虚拟机和 CAR 构件技术的基础上，提出了基于 CAR 构件对 Java 虚拟机模型的一种优化模型,并且在 Dalvik 上实现了这种模型，在不改变 Java 编程模型的情况下，让 Java 程序语言和 CAR 构件实现混合编程，能明显减小代码尺寸，提高程序运行速度，并且程序的安全性方面得到增强。

本文完成了课题的研究目标，研究成果已经在上海科泰世纪科技有限公司的产品化开发中得到应用。

关键词：Java 虚拟机，CAR，Android，Dalvik

ABSTRACT

Java programming language, which can be platform-cross based on different platform has different implement of Java virtual machine, is the language of the network. As one of the most popular programming language, Java has the features of simple, object-oriented, stable, multithreading and dynamic. Because of this, Google has specially implemented a Java virtual machine named Dalvik on Andorid system, which is aimed at embeded environment, to support Java programming.

Java program needs to be compiled into bytecode, and then be interpreted on virtual machine, so the performance can't be able to compete with native language, which is complied and executed on real machine. There are some methods to improve the efficiency of Java program, such as Java native interface technique, instantaneous compiling technique and native compilation technique, but these technique let the Java lose the feature of cross-platform or need a large number of extra memory, and has limitation of its own.

On the other hand, CAR (Component Assembly Runtime), which is developed by Kortide Corporation, is a kind of binary component and has native program efficiency. CAR via Elastos CAR runtime environment could run on different platform, and CAR has the features of self-describe, dynamic load, dynamic assembly, binary inherited. These features proved an opportunity for us to use CAR optimize Java virtual machine.

In this paper, based on Java virtual machine and CAR technology, a model use CAR to optimize Java machine was propsed, and implemented this model on Dalvik virtual machine. Without change Java programming model, realized the Java and CAR mixed programming, which can reduce the size of code obviously, improve the operating speed of program and enhance the security of application.

This paper accomplishes the research objectives and its results are already applied in Kortide Corporation's products.

Key Words: Java virtual machine, CAR, Android, Dalvik

目录

第 1 章 绪论

1.1 背景

Java 是由 Sun Microsystems 公司于 1995 年 5 月推出的 Java 程序设计语言和 Java 平台的总称。由于其简单、安全、高效及跨平台等特性，使其一诞生就受到人们的推崇，并且经久不衰。作为现在最流行的语言之一，Java 语言在网络，服务器，嵌入式操作系统等领域发挥着其重要作用。Google 于 2007 年推出的 Android 平台也决定采用 Java 语言作为其操作系统的编程语言，并且定制了特别适合嵌入式平台的 Java 虚拟机——Dalvik。

不同于编译执行的 C++语言，Java 首先要将源代码编译成字节码，然后在 Java 虚拟机上解释执行字节码。通过在不同平台实现 Java 虚拟机，从而实现了“一次编译，到处执行”的 Java 跨平台特性。这种解释执行方式降低了 Java 程序的运行效率。

另一方面，科泰公司发明的 CAR（Component Assembly Runtime）构件技术是直接采用 C++语言编程生成的二进制组件，没有采用类似 Java 的基于中间代码和虚拟机的机制，因此它的运行效率非常高。同时其运行于可以跨平台的 Elastos 平台之上，具有自描述、动态加载、动态组装、二进制继承等特性。CAR 构件的这些特性使得 Java 虚拟机调用 CAR 构件成为可能。

1.2 课题的目的和意义

Java 程序运行效率问题一直为人所诟病，而现有的提高 Java 程序的技术手段都不能完美解决这个问题。CAR 构件优化 Java 虚拟机模型的提出能够直接提高 Java 程序运行效率，并且不改变 Java 程序跨平台，可移植等特性。对 Java 程序的编程语义并不会产生太大改变，同时 CAR 构件编程模型中的一些优秀特性对 Java 编程模型是一个强大的补充。

1.3 本人所作的工作

本文详细分析了 Java 虚拟机技术和 CAR 构件技术，并且基于这两种技术提出了基于 CAR 构件优化 Java 虚拟机的软件模型。并且在 Dalvik 虚拟机上实现了这个软件模型，最后给出了新模型的编程实例和性能分析。

1.4 论文的组织结构

本文介绍了 Java 虚拟机，Android 系统，Dalvik 虚拟机，CAR 构件系统，重点是描述了 CAR 构件优化 Java 虚拟机的软件模型，并且基于 Dalvik 对该软件模型的实现。以下是本文的内容安排：

第一章绪论介绍了本文的研究背景、提出 Dalvik 调用 CAR 构件编程模型的目的和意义。

第二章介绍和分析了本文的相关的理论和技术基础，包括 Elastos 操作系统、CAR 构件技术、Java 虚拟机，Android 操作系统以及 Dalvik 虚拟机等。

第三章对当前 Java 虚拟机模型存在的一些弱点，以及基于 CAR 构件优化 Java 虚拟机模型的可行性和一些技术难点进行分析，最后提出了 CAR 构件优化 Java 虚拟机的模型，。

第四章在描述了在 Dalvik 上如何实现 CAR 构件优化 Java 虚拟机的模型。

第五章演示了 Dalvik 调用 CAR 构件编程的例子，并对其进行了性能上的比较和分析。

第六章对本文的工作进行了总结，并对该软件模型的发展进行了展望。

在文章的最后，列出了本文参考的一些文献和资料。

第 2 章 相关理论和技术研究

2.1 Java 虚拟机

2.1.1 Java 虚拟机简介

Java 虚拟机（Java Virtual Machine，简称 JVM），是可以运行 Java 代码的抽象计算机。Java 虚拟机规范定义了 Java 虚拟机的各种特性，当提到某个具体平台的 Java 虚拟机时，实际上是指在那个平台上对 Java 虚拟机规范的一个实现。Java 虚拟机是 Java 程序的运行环境，由于 Java 虚拟机规范的定义是平台独立的，所以编译后的 Java 程序可以运行在各种平台之上而不需要重新编译，从而实现了 Java 的“一次编写，到处运行”（write once, run anywhere）目标。

Java 虚拟机规范定义了每个 Java 虚拟机都必须实现的特性，但是并不具体制定某个特性应该如何实现，而是把这一决定权交给特性平台的 Java 虚拟机开发者。可以通过不同的方式（软件或硬件）实现 Java 虚拟机，每个 Java 虚拟机包括方法区，堆，Java 栈，程序计数器和本地方法栈五部分，见图 2.1 所示。

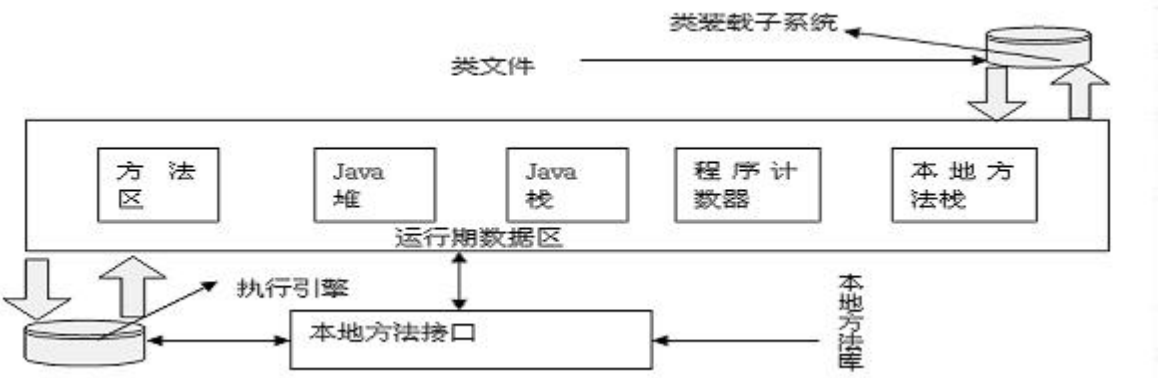

图 2.1 Java 虚拟机体系结构图

Java 虚拟机首先通过类装载子系统把中间字节码（.class 文件）装载进来，同时解析其类信息，并把他们放到方法域中。当对象创建时在堆上分配空间，当对象销毁时垃圾回收机制会将空间回收。每个 Java 线程创建时,分别为它们创建程序计数器和栈，其中程序计数器指向下一条指令，Java 栈则用作方法的调用。本地方法的调用依靠本地方法栈实现。执行引擎用于解释 Java 字节码，每个字节码定义为执行某种操作。

2.1.2 Java 程序执行过程

图 2.2 展示的是一个 Java 程序从源文件到最后执行的全过程。Java 程序源文件通过编译器产生可以在 Java 虚拟机上执行的中间字节码，在 Java 虚拟机上执行主要会经过装载，连接和解释执行三个步骤。

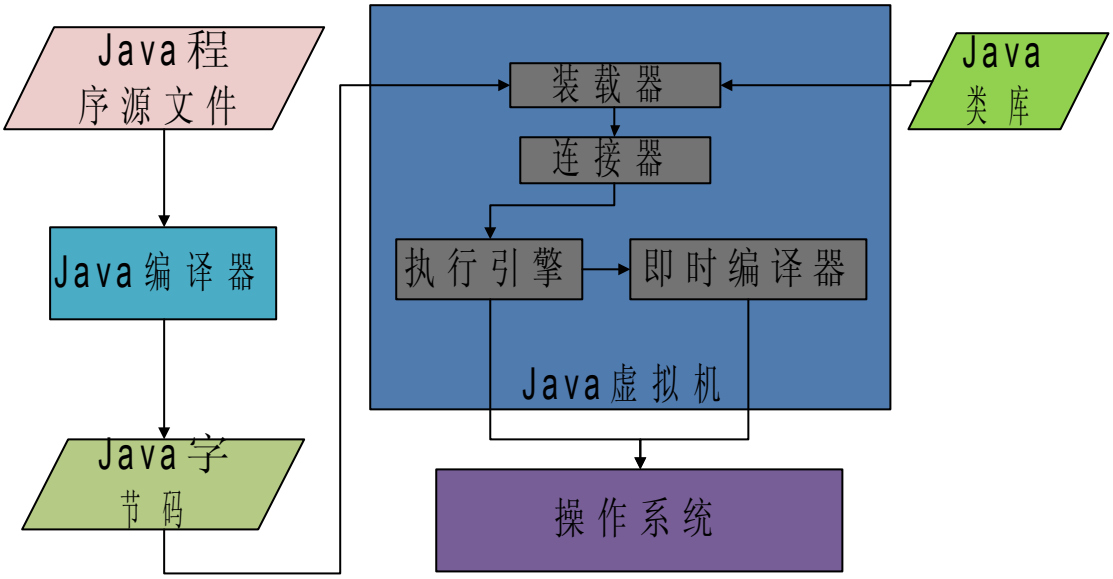

图 2.2 Java 程序运行流程

（1）装载 Java 类文件

Java 虚拟机中的类装载子系统负责装载 class 文件，可以从程序或 Java API 中装载 class 文件，其中 Java API 只有程序需要的那些类才会被装载。存在两种类装载器：引导类装载器和用户自定义类装载器。引导类装载器是虚拟机实现的一部分，它以特定的方式装载 Java 类，一般用于加载 Java API 中的类。而用户自定义装载器是由用户实现的类装载器，能够使用自定义的方式来装载类。

具体来说，虚拟机装载 Java 类文件的过程就是类装载子系统把 class 文件做为二进制数据流读入 Java 虚拟机，并解析这个二进制数据流，让 class 文件类所包含的信息转化为内部数据结构，同时创建 java.lang.Class 类的实例来表示该类文件。

（2）连接

连接又包含三个步骤——验证、准备和解析。“验证”步骤保证装载的 class 文件内容有正确的结构并且适于 Java 虚拟机的使用，而“准备”步骤则负责为该类变量分配空间，并将其初始化为默认值，“解析”步骤负责把类型中的符号引用转换为直接引用。

（3）解释执行

Java 虚拟机中有一个执行引擎来进行 Java 字节码的解释执行。Java 虚拟机规范定义了每个 Java 字节码应该处理什么，执行引擎就是按照这个规范来解释执行 Java 字节码。

Java 方法编译后形成一个字节码流，这个字节码流实际上是由 Java 虚拟机的指令序列构成的。Java 虚拟机指令和汇编指令一样，每条指令包含一个单字节的操作码，后面跟随 0 个或多个操作数。Java 虚拟机中运行的每个线程的执行引擎根据程序计数器所指向的内存空间取得操作码，如果操作码有操作数，取得它的操作数。执行操作码和跟随操作数规定的动作，然后再取得下一个操作码。这个过程将一直持续，直到从初始方法返回。

2.1.3 Java 虚拟机优化技术

最简单的解释字节码执行效率很低，在性能上大大落后于本地编译的 C++程序。在 Java 的发展过程中，很多技术被应用来提高 Java 程序的运行速度。

（1）JNI 技术

Java 的执行效率比本地程序慢的根本原因是需要通过虚拟机来解释执行中间字节码，而本地程序可以直接在机器上执行。所以提高 Java 程序执行效率最直接的办法就是在 Java 程序中用本地方法来代替 Java 方法，本地方法不需要经过执行引擎，而是直接送给操作系统执行。

在 Java 虚拟机规范中定义了 JNI（Java Native Interface）本地编程界面规范，它允许在虚拟机里运行的 Java 代码与 C，C++或汇编代码编写的程序或库进行互操作。使用 JNI 技术，可以使得 Java 程序代码的执行效率接近本地程序代码的水准。但是这种技术丧失了 Java 程序拥有的跨平台特性，同时它只能调用 C/C++语言中的简单函数，并不适合大规模编程。

（2）即时编译技术

即时编译技术（Just In Time, JIT），是虚拟机中常用到的优化执行字节码技术，能提高 Java 程序的执行速度。在虚拟机中有一个即时编译器，在虚拟机解释执行 Java 字节码的过程中，将第一次被执行的字节码编译成本地代码，并将其缓存。这样当相同的方法再次被调用时就可以执行本地代码，而不需要解释执行。很显然，执行本地代码要比解释执行 Java 字节码效率高。但是这种方式存在一个问题，由于对每条字节码都进行编译会造成编译过程负担过重，有可能使得程序执行速度反而降低了。解决的方法是只对经常执行的字节码进行编译，而对其他的代码不做编译，继续由虚拟机解释执行。

在这种方法里，虚拟机在开始的时候解释执行字节码，并记录使用最频繁的代码段，并把其编译成本地代码。这种策略之下的 Java 虚拟机只需要编译大概 10%~20%对性能有影响的代码，而能是虚拟机在 80%~90%的时间在执行被优化的代码，运行速度大大提高了。使用 JIT 技术需要等到程序运行一段时间才能真正跑起来，而且会导致代码大小增长 4~6 倍，需要很大的额外内存来保存编译生成的本地代码。

（3）提前编译技术

提前编译技术（Ahead OfTime, AOT）的基本思想是：在 Java 程序执行之前，生成 Java 方法的本地代码，以便在程序运行时直接使用本地代码。这样可以避免 JIT 编译器在运行时的性能消耗和内存消耗。虽然 AOT 技术能提高 Java 程序的性能，但是它牺牲了程序的平台无关性和代码质量，因为经过 AOT 得到的本地代码不具备 Java 程序的动态行为，也不具有关于加载类和类层次结构信息。

2.2 Dalvik 虚拟机

2.2.1 Android 操作系统

Android 是 Google 公司于 2007 年推出的基于 Linux 平台的开源手机操作系统，包括操作系统、中间件和应用软件。Android 手机平台技术由被称为开放手机联盟（Open Handset Alliance）的全球性联盟组织推广，该联盟将会支持 Google 发布的 Android 手机操作系统或应用软件。图 2.3 展示了 Android 的体系架构。

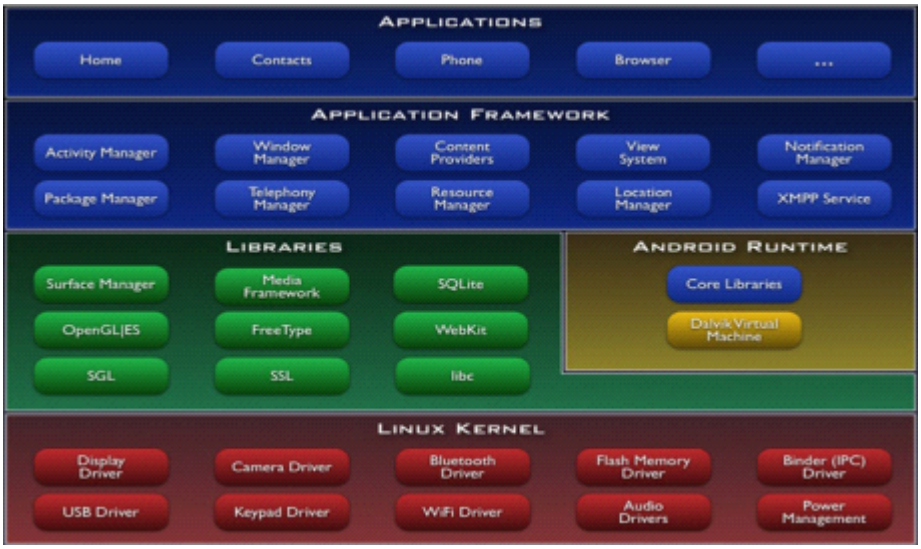

图 2.3 Android 体系架构

Android 可以自上而下分为四个层次，应用程序（Application），应用程序框架（Application Framework），库和 Android 运行环境（Libraries & Android Runtime），操作系统内核（Linux Kernel）。其中 Android 运行环境主要是指的 Dalvik 虚拟机， 应用程序，应用程序框架以及 Dalvik 的核心库都是 Java 代码编写的，它们都运行在 Dalvik 之上。Dalvik 在 Android 系统中起到了承上启下的作用，所以它性能的好坏直接决定了 Android 系统性能的好坏。

2.2.2 Dalvik 虚拟机简介

相对于 PC 来说，Android 操作系统所处的嵌入式运行平台具有较低频率的 CPU、较少的内存、不提供内外存 swap 机制以及电池供电等特点。为了能在这种功耗要求严格、内存和处理都非常有限的嵌入式平台高效地运行应用程序，Dalvik 引入了很多优化和改进的手段。以下是 Dalvik 具有的一些特性：

- 工作在低功耗的 ARM 设备上；
- 运行在 Linux 内核之上；
- 基于寄存器的虚拟机；
- 有自己的独特的指令集（不同于标准 Java 的二进制字节码）；
- 有自己的可执行文件格式（不同于标准 Java 的 Class 文件格式）；
- 支持 J2ME 的 CLDC API；
- 支持多线程。

Dalvik 接受的可执行文件是.dex(Dalvik Executable)文件格式，这种格式是对传统的.class 文件格式的优化，能很好地降低文件存储的内存需求。与传统的 Java 虚拟机不同，Dalvik 是基于寄存器的。相对于基于栈的虚拟机实现，基于寄存器的虚拟机在运行应用程序时的性能有本质的提高，并且能对字节码验证和提前优化提供更好的支持。Dalvik 在设让上要尽可能的降低功耗，减少内存使用来适应 Android 系统。从基于寄存器设计，使用独立的指令集和文件格式来看，它似乎又不像一个传统的 Java 虚拟机。但是它通过移植 Java SE 的实现 Apache Harmony 支持了 Java 编程，所以说 Dalvik 是一个适合嵌入式设备的 Java 虚拟机。

2.2.3 dex 文件格式

Android 中可以使用 dx 工具将 Java 源代码经过编译后生成 class 文件格式或 jar 文件格式转换为 dex 文件格式，也即 dex 格式的中间字节码，这是 Dalvik 接受的可执行文件格式。

如图 2.4 所示展示了 dex 文件的组成。dex 文件包括文件头，字符串列表，

类型列表，方法原型列表，域列表，方法列表，类列表，数据域几个部分。

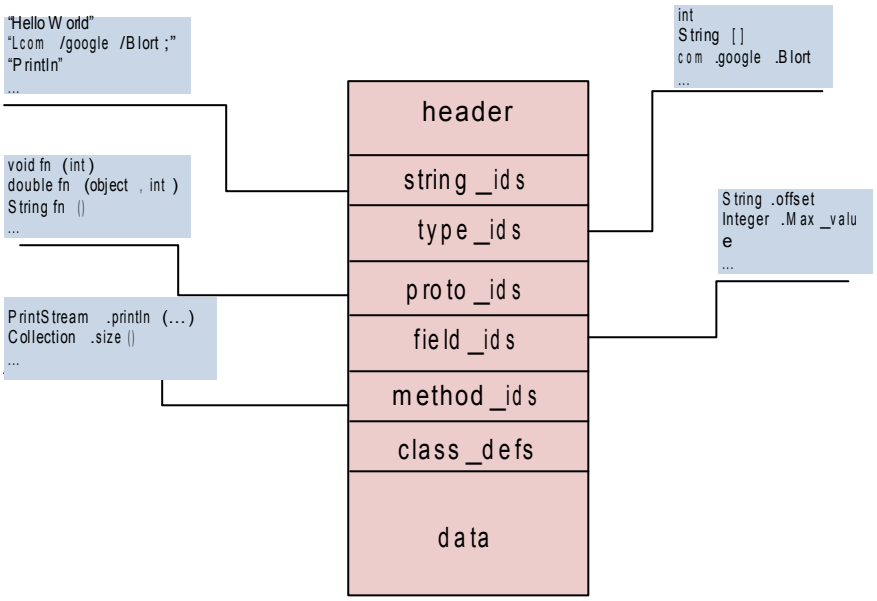

图 2.4 dex 文件的组成

其中文件头用于标识这个 dex 文件，包括文件的长度，所采取的编码格式，校检码以及接下来各个部分的大小以及绝对位置。字符串列表存储了文件中各个部分所涉及的字符串的长度和偏移位置，这些字符串包括字符串常量，类名称，方法名称，变量名称等等。类列表，方法列表和域列表分别包含在 dex 文件中所有的类引用，方法引用和变量引用。方法原型列表包含所有在 dex 文件中的方法原型。数据域包含了所有以上列表支持的数据。其实我们可以根据各个部分功能的不同把一个 dex 文件分为头部，索引和数据三个部分，头部即是文件头，索引包括字符串列表，类型列表，方法原型列表，域列表，方法列表，类列表几个部分，数据代表数据域部分。头部可以找到索引的位置和数目，可以找到数据区的起始位置，索引可以快速定位到某个元素，而数据是实际存储信息的地方。

使用 dx 工具将 class 文件格式或 jar 文件格式转换为 dex 文件，是将原来分散在各个文件中各个部分分别合并，形成 dex 文件的各个部分。这样的方式可以大大减少冗余，举例来说，如果在两个 class 文件中都出现“hello”这个字符串，转化为 dex 文件个时候，“hello”字符串就只出现一次。表 2.1 是对两中格式下的系统库，浏览器和闹钟应用程序所产生文件大小的一个对比。

表 2.1 dex 文件格式和 jar 文件格式的大小比较

对比项	未经压缩的 jar 文件		压缩的 jar 文件		未经压缩的 dex 文件	
	字节数	百分比	字节数	百分比	字节数	百分比
通用系统库	21445320	100	10662048	50	10311972	48
浏览器程序	470312	100	232065	49	209248	44
闹钟程序	119200	100	61658	52	53020	44

未经压缩的 dex 文件格式比压缩过的 jar 文件格式还要小，这对于 Java 程序运行在内存有限的嵌入式环境下是非常有帮助的。

2.2.4 中间字节码

中间字节码是 Java 源文件经过编译得到的，每条字节码由操作和操作数组成，定义了一个特定的操作，它是 Java 的“汇编语言”。相对于 Java 虚拟机规范中所定义的 Java 字节码格式，Dalvik 使用自定义的 dex 字节码格式。下面通过一小段程序在经过编译后生成的两个格式的字节码来进行分析。

如下是一个函数，它初始化两个局部变量，让对这两个局部变量做运算后存储到第三个局部变量。

```
public static void foo() {
    int a = 1;
    int b = 2;
    int c = (a + b) * 5;
}
```

生成的相应 Java 字节码如下：

```
偏移量        字节码        助记符
    0:          04        iconst_1
    1:          3B        istore_0
    2:          05        iconst_2
    3:          3C        istore_1
    4:          1A        iload_0
    5:          1B        iload_1
    6:          60        iadd
    7:          08        iconst_5
    8:          68        imul
    9:          3D        istore_2
    10:         B1        return
```

偏移量表示字节码在内存中的相对位置，一般来说一个字节码需要一个字节来保存。现有 Java 虚拟机规范中定义了 201 个不同的字节码，每个字节码用一个从 0 开始的编号来表示，并且为每条字节码定义了相应助记符。例如使用 04 来代表 iconst_1 的字节码，它定义的行为是将数字 1 入栈中。

前面提到过，规范定义的 Java 虚拟机是基于栈来计算的，虚拟机有一个程序计数器指向当前所执行的字节码，执行完后指向下一条字节码，依次循环，直到返回。初始时程序计数器指向字节码 iconst_1，下面我们就按照虚拟机的方式执行这段字节码，分析虚拟机每步所做的动作。开始时 iconst_1 实际上是把 1 放到栈顶，然后程序计数器指向 istore_0，而 istore_0 是将栈顶的值取出来放在局部变量 0 中，现在局部变量 0 的值为 1。类似的 iconst_2，istore_1 是把值 2 存储在局部变量 1 中。iload_0，iload_1 将分别局部变量 0 和局部变量 1 的值放到栈上，所以现在栈顶存放了两个值 1 和 2。iadd 所做的动作是将栈顶的两个数取出来并相加，结果入栈，现在栈顶是 3。iconst_5 将 5 入栈，imul 取出栈顶的两个数并

相乘，结果入栈，现在栈顶是 15。istore_2 将结果存入局部变量 2 中，也即我们程序中所定义的变量 c 中。最后 return，方法返回。

可以看到，采用 Java 字节码格式，虚拟机需要 10 个字节的空间来保存这个方法的 Java 字节码，运行时需要 9 次读操作，10 次写操作。

另外我们来看看采用 dex 字节码格式的情况，生成的相应 dex 字节码如下：

```
偏移量          字节码          助记符
0:              1210            const/4      v0, #int 1
1:              1221            const/4      v1, #int 2
2:              B010            add-int/2addr v0, v1
3:              DA00 0005       mul-int/lit8  v0, v0, #int 5
5:              0E00            return-void
```

dex 字节码也定义了约 255 条的字节码，每条字节码代表一个行为，并且用一个编号来代替。例如 12 定义为为一个给定的寄存器赋值，这点和 Java 字节码是不同的，dex 字节码是基于寄存器来进行计算。所以 1210 表示的是给寄存器 0 赋值为 1。

同样，我们也按照 Dalvik 执行这段字节码的过程，分析每一步虚拟机做了什么操作。const/4　v0, #int 1 和 const/4　v1, #int 2 分别给寄存器 0 和寄存器 1 赋值为 1 和 2。add-int/2addr　v0, v1 将两个寄存器的值相加，并将结果赋给寄存器 0，现在寄存器 0 中的值为 3。mul-int/lit8　v0, v0, #int 5 将寄存器 0 的值乘以 5，将结果保存在寄存器 0 中，现在寄存器 0 中的值为 15，也即程序中变量 c 的值为 15。return-void，方法返回。

这个方法基于 dex 文件格式需要的存储空间为 5 个字节，dalvik 执行这一段字节码会进行 4 次读操作，4 次些操作。存储空间和读写次数都大大优于 Java 字节码格式，并且由于读写寄存器会比读写栈或内存效率更高，所以 Dalvik 的效率是优于一般的 Java 虚拟机的。

但是这并不能说明基于寄存器的虚拟机比基于栈的虚拟机更好，因为基于寄存器的方式会假定目标机器上有多少个寄存器，这样就达不到虚拟机的平台无关要求了，因为有些平台可能并不存在虚拟机假定的那么多寄存器。这也是为什么规范中定义的字节码实现是基于栈的，而不是基于寄存器的。但是对于 Android 来说，因为它主要针对的目标机器是嵌入式的 ARM 芯片，所以 Dalvik 采用基于寄存器的实现是合理的。

2.3 CAR 构件技术

2.3.1 软件编程技术的发展

80 年代以来，目标指向型软件编程技术有了很大的发展，为大规模的软件协同开发以及软件标准化、软件共享、软件运行安全机制等提供了理论基础。其发展可以大致分为以下几个阶段。

1) 面向对象编程

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

象编程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。其代表是 C++语言所代表的面向对象编程。

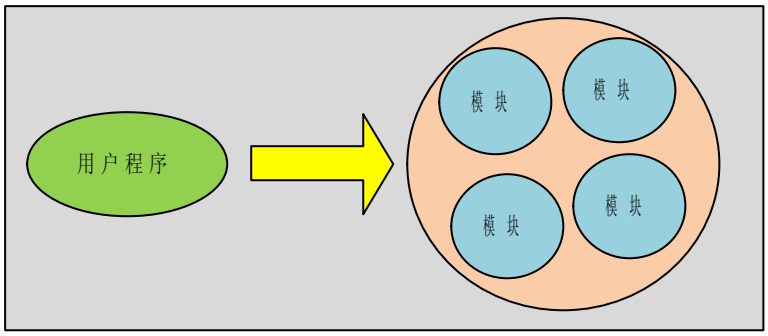

图 2.5　面向对象编程的运行模型，模块之间的关系固定

2) 面向构件编程

为了解决不同软件开发商提供的构件模块（软件对象）可以相互操作使用，构件之间的连接和调用要通过标准的协议来完成。构件化编程模型强调协议标准，需要提供各厂商都能遵守的协议体系。就像公制螺丝的标准一样，所有符合标准的螺丝和螺母都可以相互装配。构件化编程模型建立在面向对象技术的基础之上，是完全面向对象的，提供了动态构造部件模块（运行中可以构造部件）的机制。构件在运行时动态装入，是可换的。其代表是 COM 技术。

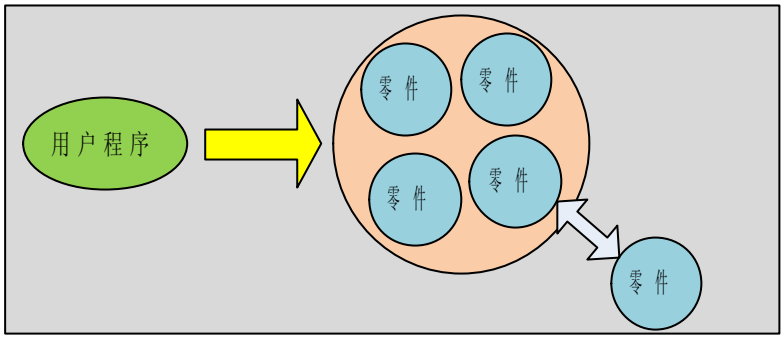

图 2.6　构件化程序的运行模型，运行时零件可替换

3) 面向中间件编程

由于因特网的普及，构件可来自于网络，系统要解决自动下载，安全等问题。因此，系统中需要根据构件的自描述信息自动生成构件的运行环境，生成代理构件即中间件，通过系统自动生成的中间件对构件的运行状态进行干预或控制，或自动提供针对不同网络协议、输入输出设备的服务（即运行环境）。中间件编程更加强调构件的自描述和构件运行环境的透明性，是网络时代编程的重要技术。其代表是 CAR、JAVA 和.NET（C#语言）。

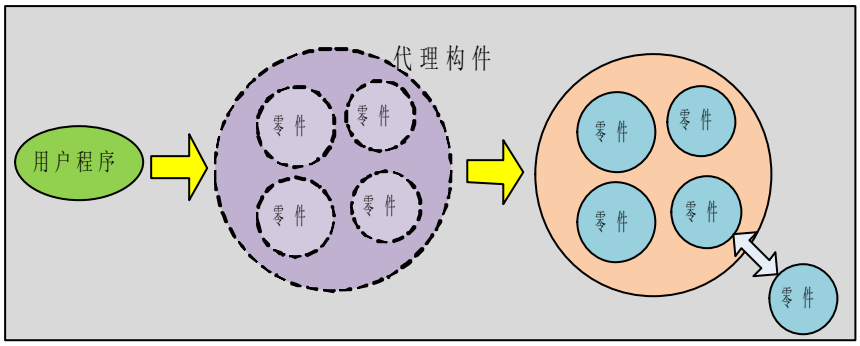

图 2.7　中间件运行环境的模型，动态生成代理构件

在这样的发展过程中，人们逐步深化了对大规模软件开发所需的科学模型、网络环境下软件运行必要机制的理解，使软件技术达到了更高的境界，实现了：

- 构件的相互操作性。不同软件开发商开发的具有独特功能的构件，可以确保与其他人开发的构件实现互操作。
- 软件升级的独立性。实现在对某一个构件进行升级时不会影响到系统中的其他构件。
- 编程语言的独立性。不同的编程语言实现的构件之间可以实现互操作。
- 构件运行环境的透明性。提供一个简单、统一的编程模型，使得构件可以在进程内、跨进程甚至于跨网络运行。同时提供系统运行的安全、保护机制。

2.2.2 CAR 构件技术的产生

CAR 技术就是在总结面向对象编程、面向构件编程技术的发展历史和经验，为更好地支持面向以 Web Service（WEB 服务）为代表的下一代网络应用软件开发而发明的。为了在资源有限的嵌入式系统中实现面向中间件编程技术，同时又能得到 C/C++的运行效率，CAR 没有使用 JAVA 和.NET 的基于中间代码——虚拟机的机制，而是采用了用 C++编程，用 Elastos SDK 提供的工具直接生成运行于“CAR 构件运行平台”的二进制代码的机制。用 C++编程实现构件技术，使得更多的程序员能够充分运用自己熟悉的编程语言知识和开发经验，很容易掌握面向构件、中间件编程的技术。在不同操作系统上实现的 CAR 构件运行平台，可以使 CAR 构件的二进制代码实现跨操作系统平台兼容。

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。这样，在不同层次上，构件均可以将底层多个逻辑组合成高层次上的粒度更大的新构件。构件之间通过约定的接口进行数据交换和信息传递，构件的位置是相互透明的，可以在同一个用户进程空间，也可以在不同的用户进程空间，甚至在不同的机器上，而且不同的构件可以用不同的语言编写，只要它们符合事先约定的构件规范。

另外，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软件开发技术水平，提高软件生产效率和软件产品质量；软件工厂化生产需要有零件的标准，CAR 构件技术为建立软件标准提供了参考，有利于建立企业、行业的软件标准和构件库。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。

为了避免使用“中间件”这个有不同语义解释的词汇造成概念上的混淆，我们简单地将 CAR 技术统称为 CAR 构件技术。

2.2.3 回调机制

普通函数调用是指应用程序调用由系统或其它程序实现的函数，而回调恰好相反，是指应用程序自己实现某些函数，用于被系统 DLL 或其它程序调用。这种函数称为回调函数，其一般用于截获消息、获取系统信息或处理异步事件。事件回调机制是 CAR 技术的重要模型之一，包括异步回调（callbacks）和同步回调(delegates)两种。

如图 2.8 所示，在回调机制中，抛出回调事件的一方称为服务器端(server)，响应回调事件的一方称为客户端(client)。客户端对于服务器端感兴趣的事件注册回调函数后，当服务器端发生该事件时，其会向与其注册过的客户端广播回调事件通知，客户端在接收到此通知后会执行对应的回调函数。值得指出的是 CAR 构件的回调依赖于 CAR 构件的 Applet 机制，Applet 机制会在下一小节做详细讨论，在这里只关注 CAR 构件如何进行回调的注册以及触发等行为。

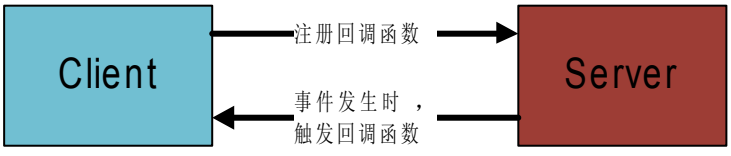

图 2.8 回调模型

使用 CAR 构件的回调模型编程时，我们需要编写两个 CAR 构件，服务器端 CAR 构件和客户端 CAR 构件。服务器端的 CAR 构件完成回调接口及事件的定义，并触发回调事件，客户端 CAR 构件则完成回调事件的实现，注册，响应和注销。例如考虑一个实际的情景，假设要帮助一座大楼实现火灾自动报警装置，

当一座大楼着火了会发出警告通知，大楼中的工作人员听到消息后便纷纷逃出大楼，消防队员听到通知就立刻冲入大楼救火。下面的步骤展示了如何在 CAR 构件编程中使用回调编程模型来实现这个情景。

4) 定义回调接口及回调事件

可以在服务器端的 CAR 构件，即图 2.8 所示的 Server 端构件定义回调接口，回调接口中定义的函数即为回调函数。定义回调接口的语法为 callbacks JXXX，JXXX 为回调接口的名称。

大楼 CBuilding 发生火灾 CatchFire()后，向外界抛出回调事件通知，不同的人在接收到这个消息后会有不同的反应，职员 Employee 会纷纷逃出大楼 RunAway()；消防员 Fireman 会赶去救火 RushIntoFire()。如下所示为 CAR 构件的定义文件 Building.car，

```
module
{
    interface IBuilding {
        CatchFire();
    }

    callbacks JFireAlarmRing { //定义异步回调接口
        EmployeeEvent();        //定义异步回调事件
        FiremanEvent();         //定义异步回调事件
    }

    class CBuilding {
        interface IBuilding;
        callbacks JFireAlarmRing;      //声明异步回调接口
    }
}
```

在上例中，服务器端为构件类 CBuilding 定义了 JFireAlarmRing 接口，并在接口中声明了 EmployeeEvent()和 FiremanEvent()回调方法，分别代表发生火灾事件时不同人员触发的回调方法。CBuilding 可以在普通接口方法 CatchFire()中抛对于不同人员抛出不同回调事件。

5) 激发回调事件

可以在普通接口方法中激发类中定义的回调事件，激发回调事件的语法为：Callback::XXX(XXX 为回调事件名)。例如在 CatchFire()函数中激发 EmployeeEvent 和 FireManEvent 两个回调事件。

```
ECode CBuilding::OnFire()
{
    CConsole::WriteLine("The building is on fire!");
    Callback::EmployeeEvent();
```

```
        Callback:: FireManEvent();
        return NOERROR;
    }
```

6) 实现回调函数

在客户端需要为所关心的回调事件实现对应的回调函数。例如对于公司雇员，当发生火警铃声响起时，逃离办公司。定义回调函数 EmployeeRunAway 如下，当触发 EmployeeEvent 回调事件时，调用此函数。

```
ECode EmployeeRunAway(PVoid,PInterface pSender)
{
    printf("The employee are running away.....");
    return NOERROR;
}
```

7) 注册回调函数

在客户端将某个服务器端的回调事件与处理该事件的回调函数一一对应的过程称为回调函数的注册。注册回调函数的方法是：在某个线程或者 ElastosMain 里调用 AddXXXCallback。例如为 EmployeeEvent 事件绑定回调函数 EmployeeRunAway 如下所示。

```
    CBuilding::AddEmployeeEventCallback(pBuilding,&EmployeeRunAway, NULL);
```

其中，EmployeeEvent 是回调事件，它指明用户所关心的事件；pBuilding 是构件对象的接口指针，它指明用户所关心的是谁的事件；EmployeeRunAway 是回调函数，如果发生用户关心的事件，pBuilding 就会调用这个函数。

8) 响应回调函数

用户程序通过指针调用普通接口方法，随即便触发了定义在该方法中的回调事件，客户端会响应该事件。

```
.....
pBuilding->CatchFire();
.....
```

因为在 CatchFire 方法中触发了 EmployeeEvent 和 FiremanEvent 事件，所以当调用 CatchFire 方法时，会调用注册在这两个事件上的回调函数。

9) 注销回调事件

在客户端通过 RemoveXXXCallback（XXX 代表回调事件名）函数来注销曾注册的回调事件。这个函数的参数和 AddXXXCallback 函数参数一样，在此不做赘述。

2.2.4 Applet 机制

进程是操作系统进行资源分配的基本单元，每一个应用程序被加载到单独的

进程中，拥有一个独立的进程空间。这样，进程作为应用程序之间一个独立而安全的边界在很大程度上提高了运行安全，以进程作为应用程序的边界的缺点是降低了性能。随着运行进程数量的增加，操作系统要频繁地在各个进程之间切换，每次都要保存和恢复进程上下文、换进换出内存页面等等，这种代价是显而易见的。

为了减少系统开销，提高运行效率，可以将各个应用程序加载到同一个进程空间里。这需要某种机制能够在同一个进程空间里来为不同的应用程序划分边界，保证应用程序不会互相干扰。但现有的进程和线程模型没有提供这种能力。

CAR 构件技术的 Applet 机制解决了这一问题。Applet 是逻辑上应用程序的载体，当将应用程序实现为一个 Applet 后，该程序运行时就具有了明确的边界。不同的 Applet 可以运行在同一个进程空间中，但是却彼此独立，一个 Applet 不能直接访问另一个 Applet 的内存空间。每个 Applet 也可以在单独的进程空间中运行。以 Applet 来实现应用程序的好处是具有逻辑上的独立性，同时实现上避免了进程间的切换，减小了系统开销。 图 2.11 展示了进程，线程和 Applet 的关系。

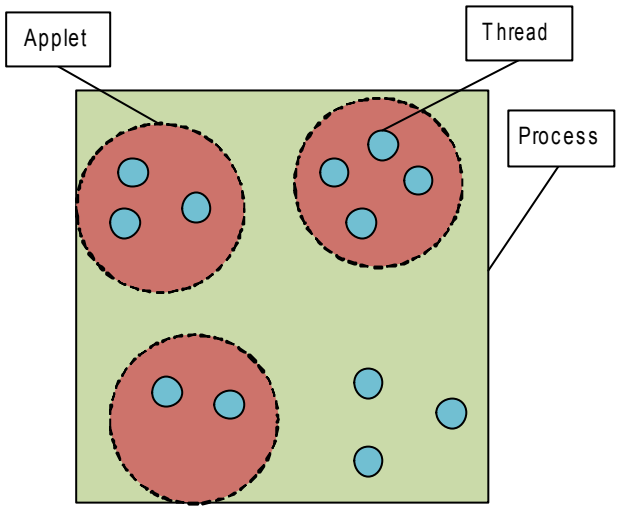

图 2.11 进程、线程、Applet 之间的关系

如果进程可以被看作由各种资源、访问/执行权限、边界和线程组成的一个运行环境，那么 Applet 就可以看作是进程的子集，进程内的 Applet 分享着进程的资源、继承若干访问权限、包含一个或多个专属线程以及 Applet 之间的边界。

在 2.2.3 节中曾提到过 CAR 构件的回调机制十分依赖于 Applet 机制，这是因为，CAR 构件注册事件和触发回调方法都是由 Applet 的消息线程来完成的。当 Elastos 的一个程序启动时，它将创建一个 Applet 主线程作为消息处理线程。当注册回调事件时，会在将这个事件标记在消息处理线程的线程局部存储区(TLS, Thread Local Storage)上。当触发回调事件时，该事件会被抛到消息队列中，最终

由消息处理线程完成实际的调用。当用户想结束 Applet 时，可以调用 IApplet::Finish 方法，这个方法会通知 Applet 的回调事件处理线程，使之结束回调事件的处理，然后 Applet 的主线程开始清理资源，准备退出。如图 2.12 是 Applet 的消息循环线程的流程图。

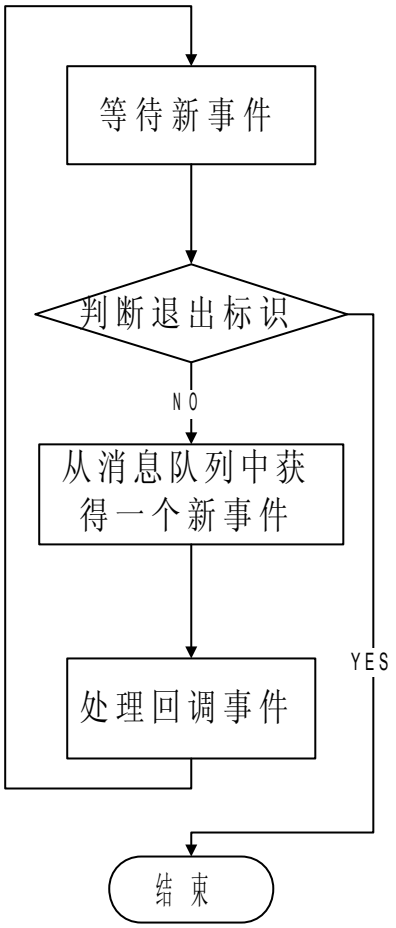

图 2.12 Applet 消息循环线程流程图

无论谁创建 Applet，Applet 都使用自己的消息处理线程。Applet 内部创建的子线程也都使用该 Applet 的消息处理线程。它所注册的回调和它的子线程所注册的回调都由 Applet 的回调线程处理。所有 Applet 的子线程都继承 Applet 的回调线程。

2.2.5 反射机制

了解 CAR 构件的反射机制之前，首先需要了解元数据。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。我们说 CAR 构件是有自描述性质的，就是构件能够描述自己的信息结构，这是通过元数据来实现的。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它使用构件服务的用户使用，元

数据描述的就是服务和调用之间的关系。

设计 CAR 反射的根本目的是为了在应用中动态发现类型信息，动态调用构件方法。在 CAR 中，接口即类型，只不过这种类型是一般类型的抽象，是“类型的类型”。以 IClassInfo 为例，在 CAR 系统中有与之对应的 CClassInfo 来描述该接口，当客户端产生一个用户自定义类（比如 Myclass，它属于 IClassInfo）实例时，CAR 系统会自动生成 CClassInfo 对象实例来表述 Myclass 对象。我们把 CClassInfo 对象称为类对象，它类似于 JAVA 中的 Class 对象，反射机制正是通过类对象中封装的方法对类型实例进行操作。

CAR 构件的反射接口主要包括 IModuleInfo、IInterfaceInfo 、IClassInfo、IMethodInfo、IStructInfo 等，它们分别可以用来表示表示 CAR 构件的模块信息，接口信息，类信息，方法信息和数据结构信息。下面就对这些接口做简单的说明来介绍 CAR 构件的反射机制。

当客户端加载构件后，首先需要“了解”构件的“内容”，有哪些类，有哪些接口，接口提供什么方法等。根据这些应用中的需求，IModuleInfo 设计为如图 2.7 所示的结构，

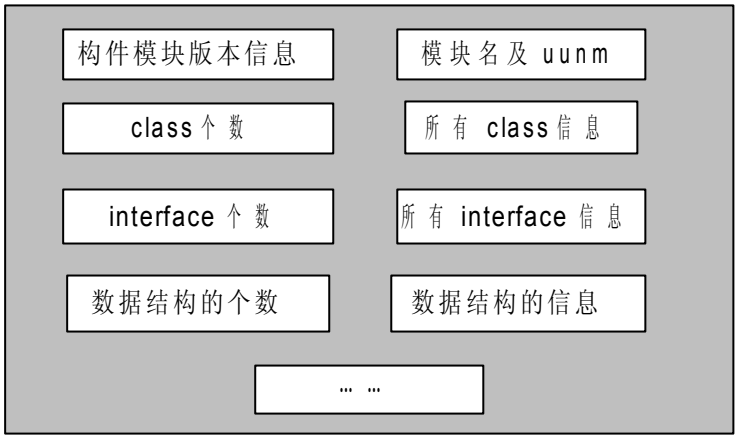

图 2.9 模块信息的主要构成

实际上由模块信息 IModuleInfo，我们可以查询得到模块中所有 class、interface 以及数据结构的信息。对于每个类信息 IClassInfo，其主要构成如图 2.8 所示，接口信息（IInterfaceInfo）也有类似的结构：

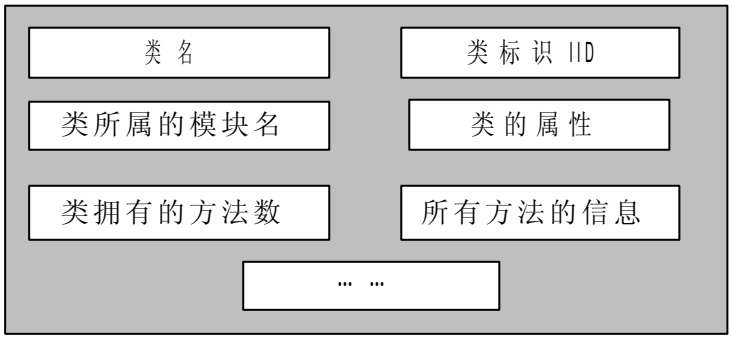

图 2.10 类信息的主要构成

通过类信息可以得到类中所有方法的信息，通过方法信息（IMethodInfo）可以得到方法的参数列表，并且可以调用这个方法。这样通过 CAR 构件的反射能够完成对 CAR 构件方法的动态调用。

2.4 Elastos 操作系统

2.4.1 Elastos 操作系统简介

Elastos 操作系统是国家 863 项目自主知识产权的 32 位嵌入式操作系统。该操作系统基于微内核，具有多进程、多线程、抢占式、基于线程的多优先级任务调度等特性。提供 FAT 兼容的文件系统，可以从软盘、硬盘、FLASH ROM 启动，也可以通过网络启动。Elastos 操作系统体积小，速度快，适合网络时代的绝大部分嵌入式信息设备。

Elastos 的功能模块全部是可拆卸的构件，可按需动态裁剪或运行时动态加载构件。 Elastos 提供了一套完整的开放的系统服务构件及系统 API、功能全面的动态链接构件库、函数库。

Elastos 区别于其他嵌入式操作系统的特点是：

1、面向构件技术，在操作系统层直接提供了对构件运行环境的支持；

2、开放的“软件总线”及“灵活内核”体系结构；

3、为资源有限的嵌入式系统有效地支持网络服务（Web Service）提供了高效率的“傻瓜化”运行平台；

4、基于 Elastos 技术，可实现软件“用户零维护”、“瞬间启动”，各类应用软件无需安装，可“点击运行”、“滚动下载”，不同厂家的软件以目标代码形式实现“无缝链接”，支持应用软件跨平台运行；

5、可以有效地支持移动计算、网络计算、普适计算等代表信息技术发展的新兴领域，为其提供与其内在的本质和特点相适应的有效的编程模型和底层软件平台解决方案；

6、体积小，高效率，面向网络时代的嵌入式信息设备应用；

7、提供 Win2000/XP 仿真技术，支持二进制代码兼容；

8、2.5G/3G 手机主要是数据业务终端。传统手机的核心技术“无线通讯模块”变成了系统组成的配件，提供软件服务是手机厂家及移动运营商市场竞争核心手段之一，这正是 Elastos 手机操作系统及 CAR 技术的优势；

9、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；

10、易于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 规范构件，共享软件资源，缩短产品开发周期（Time to Market）。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；

11、支持软件复用，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；

12、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；

13、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；

14、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；有效地构筑系统安全性，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。

在新一代因特网应用中，越来越多的嵌入式产品需要支持 Web Service，而 Web Service 的提供一定是基于构件的。在这种应用中，用户通过网络获得服务程序，这个程序一定是带有自描述信息的构件，本地系统能够为这个程序建立运行环境，自动加载运行。这是新一代因特网应用的需要，是必然的发展方向。Elastos 操作系统就是应这种需要而开发，率先在面向嵌入式系统应用的操作系统中实现了面向构件的技术。

2.4.2 CAR 构件与 Elastos 操作系统的关系

Elastos 操作系统的最大特点就是：全面面向构件技术，在操作系统层提供了对构件运行环境的支持；用构件技术实现了"灵活"的操作系统。操作系统提供

的功能模块全部基于 CAR（Component Assembly Runtime）构件技术，因此是可拆卸的构件，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。

CAR 构件的运行是依赖于 Elastos 操作系统的，我们将 Elastos 操作系统中支持 CAR 构件的运行环境称之为 Elastos 构件运行平台。CAR 可以通过 Elastos 平台运行在不同的操作系统之上，从而实现了跨平台。

第 3 章 基于 CAR 构件优化 Java 虚拟机模型分析

3.1 现有 Java 虚拟机模型的弱点

3.1.1 执行速度

和可以编译执行本地程序（例如 C++）相比，Java 程序的执行速度比较低，这是 Java 在采用虚拟机技术获得平台无关特性的同时必然要付出的代价。事实上，在 Java 的发展过程中，为了提高 Java 程序的执行速度，不断有新的优化技术被提出来，例如在小节 2.2.1 中提到的虚拟机优化技术。但是这些技术都存在各自的局限性，只适合某种特定的场合或者以牺牲 Java 的平台无关性和动态性为代价。

3.1.2 中间字节码的代码安全问题

Java 源文件编译后的 class 文件是特定于 Java 虚拟机的二进制文件格式。它的格式是有严格定义的，并且是于 Java 虚拟机所在平台无关的。在静态链接的可执行程序中，类，方法或字段的引用只是直接的指针或者偏移量，而且静态链接的二进制文件优化程度非常高，还通常缺失符号信息，这使得逆向编译二进制文件相当困难。而在 Java class 文件中，类之间通过字符串来表示所引用的类，引用方法时会指明方法的名字和原型（返回类型，参数个数和类型），而对引用的字段则有详细的描述来说明字段类型和字段名字。所以对于一个 class 文件，想要把它逆向编译为 Java 源文件相当容易。假如商业公司利用 Java 来制作他们的产品的话，竞争者可以很轻易地通过逆向编译来获得源代码从而从中获得竞争优势。

3.2 CAR 构件优化 Java 虚拟机的可行性研究

3.2.1 CAR 构件运行环境

CAR 构件的运行是依赖 Elastos 构件运行平台的。Elastos 构件运行平台提供了一套符合 CAR 规范的系统服务构件相关编程的 API 函数，实现并支持系统构建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。

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。Elastos 构件运行平台的这种跨平台特性和 Java 虚拟机要求的跨平台特性极其吻合，这就使得 CAR 构件技术引入到 Java 虚拟机中变得相当合理。

3.2.2 CAR 构件对 Java 虚拟机的优化

由二进制代码组成的 CAR 构件在 Java 虚拟机中能够直接得到接近本地代码的运行效率。这样就从根本上解决了 Java 程序运行效率不高的缺点，而不需要额外的内存消耗。

同时由于 Elastos 构件运行平台的跨平台特性，CAR 构件的引入不会对 Java 程序的跨平台特性有任何影响。相反，CAR 构件的引入可以使编程人员从 CAR 构件编程模型中获益。例如 CAR 构件的动态组装，回调机制，面向方面编程等 CAR 构件编程模型中的优秀特性是对现有 Java 编程模型的巨大补充。

另外 CAR 构件和本地程序并无二样，所以想对它进行逆向工程得到程序源代码变得和反编译本地程序一样困难，从而改变了 Java 程序对知识产权的保护有心无力的现状。

3.2.3 引入 CAR 构件存在的技术难点

由于Java编程模型和CAR构件编程模型实际上是两种完全不同的编程模型，它们之间的实现混合编程肯定会碰到许多技术问题。

（1） CAR 构件的加载

碰到的第一个要解决的问题就是CAR构件如何加载到Java虚拟机之中，这是 CAR 构件优化 Java 虚拟机模型的起点。只有首先将 CAR 构件正确加载到 Java 虚拟机中，我们才能实现 CAR 优化 Java 虚拟机模型的各种想法。解决这个问题的方案是首先将 Elastos 构件运行平台引入到 Java 虚拟机之中（这个由于 Elastos 构件运行平台的跨平台特性，实现并不困难），然后通过 CAR 构件的反射机制将 CAR 构件加载进来。

（2） 调用 CAR 构件

在 CAR 构件加载进来以后，如何实现 Java 程序和 CAR 构件之间的相互操作是首要关注的问题。为了引入不使得 Java 编程模型有很大的变动，我们

根据 CAR 构件构造一些 Java 类，这些 Java 类能够正确反映 CAR 构件内部的类，方法和变量，它们实际上是 CAR 构件在 Java 程序的一个代理。为了讲述简便起见，将这些由 CAR 构件生成的 Java 类称之为代理类。在 Java 虚拟机加载 CAR 构件的同时也加载这些代理类，运行代理类时虚拟机会操作 CAR 构件。所以当 Java 程序创建这些代理类的对象时，实际上是创建了相应的 CAR 构件对象，这是 CAR 构件对象在 Java 程序端的代理对象。当调用这些代理类中的方法时，实际上就是调用了 CAR 构件的方法，从而实现了 Java 对 CAR 构件的调用。并且 Java 程序本身并不知道它在操作一个 CAR 构件，它的行为方式和操作一个普通的 Java 类并无二异，这样我们的 CAR 构件优化虚拟机模型实际上并没有改变 Java 语言编程模型。

（3） 参数问题

语言之间混合编程，参数的传递是肯定要遇到的问题。CAR 构件编程模型的参数类型和 Java 语言的参数类型并不相同，这需要我们仔细研究两种编程模型的参数类型，并对它们进行转化。即将当 Java 程序调用 CAR 构件时将 Java 的参数类型转化为 CAR 构件对应的类型，当 CAR 构件返回时将 CAR 构件的参数类型转化为 Java 对应的类型。

（4） 内存管理

Java 语言和 CAR 构件有不同的内存管理模型，当他们混合编程时，如果没有统一的内存管理机制的话，很容易造成内存泄露。而内存泄露对于大型软件来说是不可容忍的，所以这是想要使 CAR 构件优化 Java 虚拟机模型在实际应用中广泛使用必须要解决的问题。

那么现在让我们来看看两种编程模型各自的内存管理机制，从而找出统一内存管理机制的办法。Java 通过垃圾回收机制来统一管理内存，所以它并不需要程序员关心内存。CAR 构件编程模型是通过引用计数的方式对 CAR 构件分配的内存进行管理。前面我们提到 CAR 构件对象实际上在 Java 程序端会有一个代理对象和它相对应，我们可以利用这点来设计我们的内存统一管理模型。即当代理对象销毁时（依靠垃圾回收机制），我们销毁相应的 CAR 构件对象（引用计数减为零）。

3.3 CAR 构件优化 Java 虚拟机的模型

通过前面的分析，我们知道 CAR 构件优化 Java 虚拟机是可行的，并且是有意义的。如图 3.1 是 CAR 优化 Java 虚拟机的模型，Elastos 构件运行平台作为 Java 虚拟机的一部分提供 CAR 构件运行环境支持。CAR 构件通过一个代理类暴露给

Java 程序调用，产生 Java 对象的同时也生成代理类对象，Java 通过对代理类对象的控制也可以控制 CAR 对象。虚拟机负责对它们进行统一的管理。

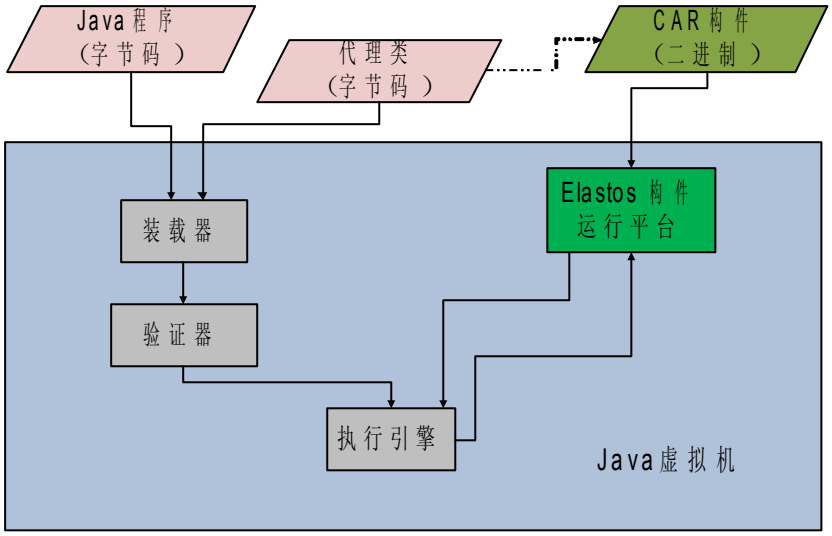

图 3.1 CAR 构件优化 Java 虚拟机模型

图 3.2 展示了 CAR 优化 Java 虚拟机后 Java 应用程序、CAR 构件、Elastos 构件运行平台、Java 虚拟机各个部分的关系。

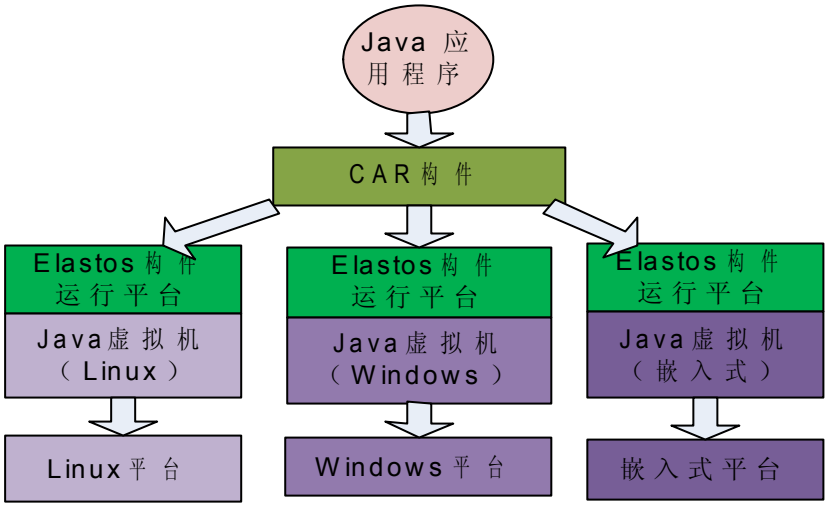

图 3.2 Java 程序和 CAR 构件的关系

在不同的平台上，无论是 Linux，Windows 还是嵌入式平台都有各自不同的 Java 虚拟机实现，Java 应用程序都能够不做修改地运行在这些平台之上。同样 Elastos 构件运行平台屏蔽了底层操作系统的具体特征，可以让 CAR 构件运行在不同的平台之上。在 CAR 构件优化 Java 虚拟机的模型中，Java 应用程序调用可以在各种平台之上调用 CAR 构件，在大大提高程序运行效率的同时不丧失 Java 程序的跨平台特性。

第 4 章 CAR 构件优化 Dalvik 实现

在分析了 CAR 优化 Java 虚拟机模型的基础上，本章主要论述此模型在 Android 上的一个实现，即利用 CAR 构件来优化 Android 上的 Java 虚拟机——Dalvik。

4.1 Dalvik 运行 Java 程序原理

在用CAR构件优化Dalvik虚拟机之前，我们需要首先弄明白Dalvik运行 Java 程序的原理，分析各个关键步骤，这样才能设计如何将 CAR 构件引入到 Dalvik 中以及引入 CAR 构件的时机，做到有的放矢。

根据 2.1.2 节的介绍，Java 虚拟机的本质实际上是加载中间字节码文件并解释执行中间字节码，Dalvik 也不例外，虽然它接受的输入是自定义的.dex 文件格式而不是一般的.class 文件格式。Dalvik 运行一个普通的 Java 程序，大致可以分为 3 个过程，首先创建一个虚拟机的实例，然后加载 Java 程序的.dex 文件格式的中间字节码，最后找到 DEX 字节码中的 main 方法开始解释执行，直到程序运行完毕。下面就依次对这三个步骤进行详细说明以了解 Dalvik 运行 Java 程序的内部机制。

4.1.1 创建虚拟机实例

Dalvik 通过 JNI 函数 JNI_CreateJavaVM 来创建一个 Java 虚拟机。在这个函数中会完成很多初始化的工作，为虚拟机下一步加载字节码和解释执行做好准备。这一步所作的工作基本如下：

- 分析从命令行所传入的参数，并作相关设置
- 初始化堆和垃圾回收机制
- 初始化线程
- 加载系统基础类库
- 初始化并注册本地函数库

在虚拟机实例创建好以后，虚拟机的加载字节码系统和执行引擎系统处于可工作状态。

4.1.2 加载 dex 字节码

在加载 dex 字节码这一步中，首先 Dalvik 将 dex 文件映射到内存中来，这样操作 dex 文件只需要操作内存，而不需要对文件进行操作，提高了效率。需要指出的是，dex 文件的结构是紧凑的，为了进一步提高性能，Dalvik 在第一次解析 dex 文件时首先对它进行了优化，优化主要的工作是

- 对虚函数的调用，用虚表中的方法索来代表虚函数
- 对域的方问，用域的索引来代表域
- 对于少数的会频繁调用的函数，如 String.length()，使用内联函数来代替
- 删除没有被调用的函数
- 增加可以提前计算的数据

所以在 Dalvik 运行 Java 程序时，会生成一个以.odex 结尾的优化文件，有研究表明优化后的文件大小有所增加，可以达到源文件的 1~4 倍。

由于加载某一个类时，都是通过类的名字来寻找，所以虚拟机根据类的名字建立一个哈希表来存储已经找到的类信息，这样下次查找相同的类时可以快速定位。在加载类的具体内容时，虚拟机首先为每个类创建一个 ClassObject 对象，以下是这个 ClassObject 对象的数据结构。

```
struct ClassObject {
    Object              obj;
    u4                  instanceData[CLASS_FIELD_SLOTS];
    const char*         descriptor;
    char*               descriptorAlloc;
    u4                  accessFlags;
    u4                  serialNumber;
    DvmDex*             pDvmDex;
    ClassStatus         status;
    ClassObject*        verifyErrorClass;
    u4                  initThreadId;
    size_t              objectSize;
    ClassObject*        elementClass;
    ClassObject*        arrayClass;
    int                 arrayDim;
    PrimitiveType       primitiveType;
    ClassObject*        super;
    Object*             classLoader;
    InitiatingLoaderList initiatingLoaderList;
    int                 interfaceCount;
    ClassObject**       interfaces;
    int                 directMethodCount;
```

```
    Method*             directMethods;
    int                 virtualMethodCount;
    Method*             virtualMethods;
    int                 vtableCount;
    Method**            vtable;
    int                 iftableCount;
    InterfaceEntry*     iftable;
    int                 ifviPoolCount;
    int*                ifviPool;
    int                 sfieldCount;
    StaticField*        sfields;
    int                 ifieldCount;
    int                 ifieldRefCount;
    InstField*          ifields
    const char*         sourceFile;
};
```

成员 serialNumber 是由虚拟机制定的唯一类序列号，initThreadId 是初始化这个类对象的线程号，classLoader 表示加载这个类的类加载器，status 表示类对象的状态，可以有如下的状态，

```
typedef enum ClassStatus {
    CLASS_ERROR             = -1,
    CLASS_NOTREADY          = 0,
    CLASS_LOADED            = 1,
    CLASS_PREPARED          = 2,    /* part of linking */
    CLASS_RESOLVED          = 3,    /* part of linking */
    CLASS_VERIFYING         = 4,    /* in the process of being verified */
    CLASS_VERIFIED          = 5,    /* logically part of linking; done pre-init */
    CLASS_INITIALIZING      = 6,    /* class init in progress */
    CLASS_INITIALIZED       = 7,    /* ready to go */
} ClassStatus;
```

分别表示了这个类对象的各个状态，其中只有 CLASS_INITIALIZED 状态才表示这个类对象是可以使用的。serialNumber、initThreadId、status、classLoader 这些成员变量在创建的时候就可以设置。还有一些成员变量需要读取 dex 文件中的类描述信息才能设置。例如 descriptor（类描述符），accessFlags（访问标识），objectSize（对象的大小）、（super）父类等信息可以直接在 dex 文件中获取。interfaces（接口）、directMethods（方法）、virtualMethods（抽象方法），ifields（域），sfields（静态域）这些成员变量需要根据它们的个数分配相应的内存，并且从中读取相应的信息。Method 这个数据结构用来代表一个方法，在下节会详细分析这个数据结构。当这个 classObject 对象的成员都设置完毕就完成了从 dex 中对一个类对象的装载。

4.1.3 解释执行 dex 字节码

在从 dex 文件加载完字节码后，虚拟机在类中寻找 main 方法，这个方法必须存在而且是静态的，公共的，否则虚拟机直接退出。这也是为什么 Java 程序中总要定义一个静态公共 main 方法的原因，它是解释执行 dex 字节码的入口。

在详细描述解释执行的过程之前，我们来先看看 Dalvik 中对一个方法是如何存储的。

```
struct Method {
    ClassObject*     clazz;
    u4               accessFlags;
    u2               methodIndex;
    u2               registersSize;
    u2               outsSize;
    u2               insSize;
    const char*      name;
    DexProto         prototype;
    const char*      shorty;
    const u2*        insns;
    int              jniArgInfo;
    DalvikBridgeFunc nativeFunc;
    const RegisterMap* registerMap;
};
```

clazz 表示这个方法所属的类，accessFlags 代表的是这个方法的访问属性，这在规范中有详细的定义。例如如果这个方法具有 public 修饰符，那么它的 accessFlags 最后一位为 1，如果方法具有 native 修饰符，accessFlags 倒数第 9 为为 1。name 表示方法的名字，prototype 表示方法的原型。insns 是最重要的成员，它指向了代表方法的中间字节码。值得注意的是，当一个方法具有 native 修饰符，也即这个方法是一个本地方法时，当第一次调用这个方法时，虚拟机会在设定好的库路径中寻找这个方法对应的本地函数，并将本地函数的指针保存在 insns 中，这样再次调用此方法时不需要再次定位本地函数，而是直接使用 insns 作为本地函数指针。而 nativeFunc 作为虚拟机和本地函数之间的调用界面，每次 Java 虚拟机执行本地方法时都调用 nativeFunc 指针所指向的函数，在这个函数中完成对本地函数（insns 所指向的函数）的调用。我们的 CAR 构件优化 Dalvik 会使用到这一特性。

所以当虚拟机解释执行一个方法时，总是找到这个方法 insns 取得字节码来开始解释执行。之后解释执行字节码的过程和我们在 2.2.4 节中分析的过程一致，图 4.1 是 Dalvik 执行 dex 字节码的流程图。

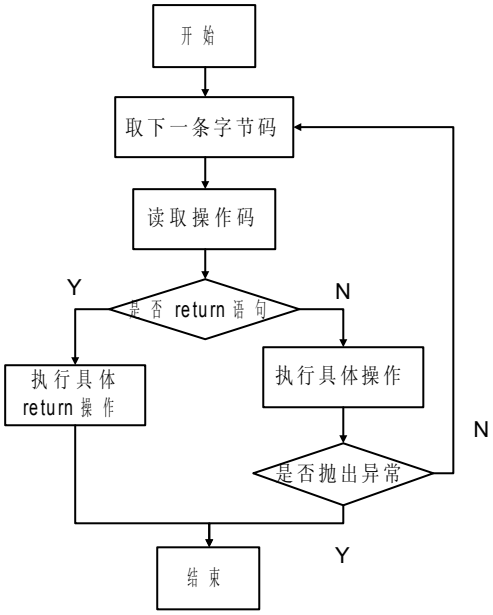

图 4.1 Dalvik 执行 dex 字节码

执行引擎本质上是一个大循环，不断地从字节码序列中取出下一个字节码来执行。其中的执行具体操作就是解释字节码的部分，可以有不同的实现。为了提高效率，Dalvik 采用汇编指令来执行。即每个字节码都对应一段汇编语言代码，这段汇编语言程序完成了字节码指定的行为。执行引擎根据不同的字节码跳到对应的汇编代码进行执行，执行完毕后再取得下一个字节码，依次循环，直到有异常抛出或者遇到 return 语句。

以上是执行一个方法，当遇到方法调用时（方法调用也是字节码指令），会把调用方法的类信息，方法索引值保存起来，然后将调用参数传递给被调用方法执行。当被调用方法返回时，会从发生调用的地方继续执行调用的方法。

4.2 在 Dalvik 中引入 Elastos 构件运行环境

在分析了 Dalvik 的运行机制后，接下来我们实现如何使用 CAR 构件优化 Dalvik 虚拟机。

要想 CAR 构件能够运行在虚拟机之上，必须先把 Elastos 构件运行环境引入进来。Elastos 构件运行环境对于 CAR 构件的支持实际上是表现为一组 API，调用这组 API 很够实现加载 CAR 构件，反射 CAR 构件，实现 CAR 构件的回调等各种功能。在 3.2.1 节提到，Elastos 构件运行环境是跨平台的，它在不同的操作系统上有不同实现，所以我们只需要在 Elastos 开发平台编译出符合 Android 平台的 Elastos 构件运行环境。

编译出的 Elastos 构件运行环境表现为一组动态库，分别是 elastos.eco，ElCRuntime.eco，elfloader.dll，JniBridge.so，KD.dll，zlib.so。可以通过 adb 工具

将这些动态库放入 Android 的文件系统中。然后在 Dalvik 启动时通过 dlopen 函数打开这些动态库，将这些动态库加载到进程中。为了将动态库中的 API 暴露出来，可以通过 dlsym 得到动态库中各种 API 函数的地址。通过 dlsym 得到的函数地址，我们可以按照 API 函数的名字构造一套相同的 API 函数提供给 CAR 构件使用。举例来说，对于 Elastos 构件运行环境中的反射 CAR 构件的 API 函数 _CReflector_AcquireModuleInfo，我们通过如下方式得到。

首先定义一个函数指针来保存 Elastos 构件运行环境库中的 API 函数地址，

```
Static _ELASTOS ECode __cdecl
        (*_CReflector_AcquireModuleInfo_Internal)(Elastos::String name, IModuleInfo**);
```

在 ElapiInitialize 函数中加载 Elastos 构件运行环境库，这个函数是在创建虚拟机实例的时候被调用的，也即加载 ELastos 构件运行环境库的时机是创建虚拟机实例的时候。

```
bool ElapiInitialize()
{
    ……
    handle_elfloader = dlopen("/data/data/com.elastos.runtime/elastos/elfloader.dll",
                                    RTLD_LAZY);
    ……
    void* (*pLoadElfModule)(const char*, const char*, int);
    pLoadElfModule = (void* (*)(const char*, const char*, int))dlsym(
                                    handle_elfloader, "__elfloader_LoadElfModule");
    ……
    elHandle = pLoadElfModule("/data/data/com.elastos.runtime/elastos",
                            "elastos.eco", RTLD_LAZY);
    ……
    handle_ElCRuntime = pLoadElfModule("/data/data/com.elastos.runtime/elastos",
                                    "ElCRuntime.eco", RTLD_LAZY);
    ……
    _CReflector_AcquireModuleInfo_Internal = (_ELASTOS ECode __cdecl
                    (*)(Elastos::String, IModuleInfo**))pGetElfModuleSymbol
                    (elHandle, "__elastos__CReflector_AcquireModuleInfo");
    ……
}
```

如果加载成功，elHandle 中保存的就是动态链接库的句柄，而函数指针 _CReflector_AcquireModuleInfo_Internal 保存的是 Elastos 构件运行环境中对应 API 的函数地址。我们将动态库句柄和函数地址都保存下来，这样每个虚拟机只需要执行依次动态库的加载和函数查找。利用从动态库中得到的函数地址，可以在 Dalvik 虚拟机中实现自己的 Elastos 构件运行 API 函数，从而得到了 CAR 构件的运行环境。

```
_ELASTOS ECode __cdecl _CReflector_AcquireModuleInfo(Elastos::String name,
                                        IModuleInfo **ppModuleInfo)
{
    return _CReflector_AcquireModuleInfo_Internal(name, ppModuleInfo);
}
```

这样在 Dalvik 中我们就有了 CAR 构件需要的 Elastos 构件运行环境了。

4.3 Dalvik 调用 CAR 构件

4.3.1 根据 CAR 构件产生代理类

根据 CAR 优化虚拟机模型的设计，为了不改变 Java 编程模型，我们通过生成一个 Java 类来作为 CAR 构件在 Java 端的代理。这个代理类应该能完全反映 CAR 构件所包含的信息，这样虚拟机在处理代理类时，也能对 CAR 构件做相应的处理。

如下是一个 CAR 构件的定义文件 CCalculate.car，它有一个类 CCalculate，类包含一个 IAdd 接口，具有一个 Add 方法实现两个整型数相加，结果返回一个整型。

```
module
{
    Interface IAdd
    {
        Add([in] Int32 I, [in]Int32 j, [out]Int32* o)
    }
    Class CCalculate
    {
        Interface IAdd
    }
}
```

接下来是根据这个 CAR 构件生成对应的 Java 代理类，这一步已经实现用工具 car2java 可以自动化生成。即 car2java 接受一个 CAR 构件作为输入，输出这个 CAR 构件对应的 Java 代理类文件。例如上述 CAR 构件生成的 Java 代理类文件 CCalculate.java 如下所示。

```
import dalvik.annotation.ElastosClass;
@ElastosClass(Module="CCalculate.eco", Class="CCalculate")
class CCalculate{
    native static int Add(int i, int j);
}
```

其中类名,方法名与 CAR 构件中定义的一致，这样方便虚拟机在操作代理类

时操作 CAR 构件。由于这个类是个代理类，所以它的方法的实现是在 CAR 构件中，方法有 native 修饰符修饰，这样可以通过编译而不会报错。而方法中传的参数，可以看到 Java 语言的 int 类型和 CAR 构件中的 Int32 类型相对应，这点后面还会详细介绍。而 ElastosClass 是位于 Dalvik 核心库的一个注释类，它定义如下

```
package dalvik.annotation;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ElastosClass {
    String Module();
    String Class() default "";
    String SingletonClass() default "";
}
```

利用这个注释类可以指定 Java 代理类所对应的 CAR 构件以及 CAR 构件中类的名字，这样虚拟机在加载 Java 代理类时可以加载对应的 CAR 构件，并从 CAR 构件中得到类的信息。

4.3.2 加载代理类时操作 CAR 构件

当 Java 程序调用代理类时，虚拟机首先会加载此代理类，我们在这里改变虚拟机的行为并做相应的处理。loadClassFromDex0 函数是每个类从字节码文件中加载到虚拟机都要经过的，所以我们在这里判断加载的类是否是 CAR 对象代理类，判断的依据就是类中是否含有 ElastosClass 注释。

```
static ClassObject* loadClassFromDex0(DvmDex* pDvmDex,
        const DexClassDef* pClassDef, const DexClassDataHeader* pHeader,
        const u1* pEncodedData, Object* classLoader)
{
    ClassObject* newClass = NULL;
    ……
    if (gDvm.shouldCARWorking) {
        if (dvmHasCARClassAnnotation(newClass)){
            SET_CLASS_FLAG(newClass, CLASS_CAR_CLASS);
            if(!dvmSetCARClassInfo(newClass))
                LOGW("Set class info failed %s", descriptor);
        } else if (dvmHasCARInterfaceAnnotation(newClass)) {
            SET_CLASS_FLAG(newClass, CLASS_CAR_INTERFACE);
            if (!dvmSetCARInterfaceInfo(newClass))
                LOGW("Set class info failed %s", descriptor);
        }
```

```
        }
        ……
    }
```

正如 4.1.2 节所说明的一样，加载一个类文件就是创建一个 ClassObject 对象，在这里是创建对象是 newClass。dvmHasCARClassAnnotation 判断一个类文件中是否包含 ElastosClass 的注释类，如果为真，说明正在加载的类是一个 CAR 构件代理类。通过 SET_CLASS_FLAG 为其访问属性添加 CLASS_CAR_CLASS 标识，也即

```
        newClass->accessFlags |= CLASS_CAR_CLASS
```

这样这个代理类的 ClassObject 对象就具有 CLASS_CAR_CLASS 标识了，这个标识使代理类和其他的普通 Java 类分开。

如果某个类是一个代理类，那么我们通过 dvmSetCARClassInfo 这一函数加载相应的 CAR 构件，反射出 CAR 构件，得到对应的类信息并且将其保存起来。

```
bool dvmSetCARClassInfo(/*const*/ ClassObject* clazz)
{
    char* CARModule = dvmGetCARClassAnnotationValue(clazz, (char*)"Module");
    char* CARClass = dvmGetCARClassAnnotationValue(clazz, (char*)"Class");
    ……
    IModuleInfo *pModuleInfo;
    ……
    pModuleInfo = (IModuleInfo*)findHashCARModuleInfo(cachedName,
                                kAddRefCount, NULL);
    if (pModuleInfo == NULL) {
        ec = _CReflector_AcquireModuleInfo(cachedName, &pModuleInfo);
        if (FAILED(ec)) {
            ……
        } else {
            if (addHashCARModuleInfo(cachedName, pModuleInfo) == NULL) {
                LOGD("add hash _CReflector_AcquireModuleInfo Failed");
            }
        }
    }
    dvmHashTableUnlock(gDvm.nativeLibs);
    ……
```

这是加载 CAR 构件的部分，首先从代理类的注释中得到这个代理类对应的 CAR 构件名称和类名称。为了避免重复加载 CAR 构件，将已加载的 CAR 构件信息保存在一个系统的哈希表 gDvm.nativeLibs 中。当每次要加载新的 CAR 构件时首先根据 CAR 构件的名称去查找这个表，如果找到，说明当前要加载的 CAR 构件已经在之前加载过了，可以直接使用，不需要再重新加载。如果哈希表中不存在要加载的 CAR 构件，那么现在加载，并把加载后的信息保存在哈希表中。

加载 CAR 构件的函数_CReflector_AcquireModuleInfo 是前面已经提到的 Elastos 运行环境中的 API，它接受一个 CAR 构件的名称和一个 IModuleInfo 类型的指针，如果加载 CAR 构件成功，IModuleInfo 保存的就是这个 CAR 构件的信息。实际上保存在 gDvm.nativeLibs 哈希表中的信息正是每个 CAR 构件的 IModuleInfo，通过它能够得到 CAR 构件的全部信息。接下来需要得到 CAR 构件中的类信息，并将其保存在 ClassObject 结构体中。

```
        ……
        IClassInfo *pClassInfo;
        if (CARClass != NULL && *CARClass != '\0') {
            ec = pModuleInfo->GetClassInfo(CARClass, &pClassInfo);
            if (FAILED(ec)) {
                LOGD("GetClassInfo Failed");
                return false;
            }
            clazz->pClassInfo = (u4)pClassInfo;
            ……
        } else {
            clazz->pClassInfo = NULL;
        }
        ……
        return true;
    }
```

以上是 dvmSetCARClassInfo 的下半部分，pModuleInfo 是之前加载 CAR 构件得到的 IMuduleInfo 指针，CARClass 是在代理类中通过注释类指定的 CAR 构件中的对应类名称。将类名称做为参数，通过 IMuduleInfo 的 GetClassInfo 方法可以得到 CAR 构件类的类信息 IClassInfo。然后是将得到的 IClassInfo 类信息保存在 ClassObject 对象中，为此，我们在 ClassObject 的结构体中添加了一个新的成员变量 pClassInfo 用来保存 IClassInfo。

```
    struct ClassObject {
        Object          obj;                    /* MUST be first item */
        u4              instanceData[CLASS_FIELD_SLOTS];
        ……
        //Elastos: inside CAR support, this is the IClassInfo which is Reflected from CARClass.
        //For no CAR class, this will be NULL.
        u4              pClassInfo;
    };
```

pClassInfo 添加在 ClassObject 结构体的末尾，对于代理类的对象，这个成员是代理类对应 CAR 构件类的 IClassInfo，而对于普通的 Java 类对象，这个成员的值为 NULL。这样虚拟机完成了在加载代理类的同时加载相应的 CAR 构件并

反射得到 CAR 构件中对应类信息的过程。

4.3.3 产生代理类对象时生成 CAR 构件对象

代理类只是 CAR 构件在 Java 端的一个代理，所以当普通的 Java 程序调用代理类，创建一个代理类对象时，虚拟机要产生对应的 CAR 构件对象。虚拟机都是通过 dvmAllocObject 来创建新的 Java 对象的，所以我们对这个函数进行修改。当创建的是一个代理类的对象时，我们除了创建这个代理对象，同时还创建对应的 CAR 构件对象，并且把它保存起来。

```
Object* dvmAllocObject(ClassObject* clazz, int flags)
{
    Object*         newObj;
    int             layerCount;
    ClassObject* clz;
    ClassObject* clz2;
    u4*              pCARObject;
    ……
    for (clz = clazz;  clz->super != NULL;  clz = clz->super) {
        if (IS_CLASS_FLAG_SET(clz,
                    CLASS_CAR_CLASS|CLASS_CAR_INTERFACE))
            break;
    }
    if (IS_CLASS_FLAG_SET(clz, CLASS_CAR_CLASS)) {
        for (layerCount = 1, clz2 = clz->super;
                clz2->super != NULL;  clz2 = clz2->super) {
            if (IS_CLASS_FLAG_SET(clz2, CLASS_CAR_CLASS))
                layerCount++;
        }
        newObj = dvmMalloc(clazz->objectSize + sizeof(u4)*layerCount, flags);
        for (layerCount = 0, clz2 = clz;  clz2->super != NULL;  clz2 = clz2->super) {
            if (IS_CLASS_FLAG_SET(clz2, CLASS_CAR_CLASS)) {
                u4 pCARObject = dvmCreateCARClassObject(clazz, newObj);
                ……
                *(u4 *)(((char *)newObj) + clazz->objectSize
                            + sizeof(u4)*layerCount) = pCARObject;
                layerCount++;
            }
        }
    }
    ……
    return newObj;
}
```

在这个函数中，首先通过查看要创建对象或要创建对象的父类是否有设置 CLASS_CAR_CLASS 访问标识，如果设置了这个标识的话，说明要创建的是一个代理类对象。对于代理类对象，为其多分配额外的空间来存放 CAR 构件的对象。也即新创建的 CAR 构件对象是附加在代理对象之上的，这样可以很快速地通过代理对象找到对应的 CAR 构件对象，也方便了 CAR 构件对象的管理。dvmCreateCARClassObject 是根据代理对象创建 CAR 构件对象的函数，实现如下。

```
u4 dvmCreateCARClassObject(const ClassObject *clazz, const Object *object)
{
    IClassInfo          *pClassInfo;
    PInterface           pObject;
    ECode                ec;
    IConstructorInfo *pConstructorInfo;
    pClassInfo = (IClassInfo*)(clazz->pClassInfo);
    if (clazz->pConstructorInfo == (u4)NULL) {
        ec = pClassInfo->CreateObject(&pObject);
    } else {
        ……
    }
    return (u4)pObject;
}
```

因为之前已经为代理类的 ClassObject 设置了其 pClassInfo 指向 CAR 构件对象的类信息，所以这里可以直接取得代理类对应的构件类的信息。然后通过 IClassInfo 的 CreateObject 函数可以新建一个 CAR 构件对象，保存在 pObject 当中。新建的 CAR 构件对象又附加在代理类对象之后额外分配的内存中，这样我们就根据代理类对象产生了对应的 CAR 构件对象。

4.3.4 为代理类对象设置 CAR 构件方法信息

当 Java 调用代理类中的方法时，由于代理类中的方法没有实现，具有 native 修饰符，所以它在第一次调用的时候会触发 dvmResolveNativeMethod 函数。这个函数是为 Java 本地方法在动态库中寻找对应的本地函数，这样在下一次调用相同本地方法是可以直接调用动态库中的本地函数，而不需要重新查找一遍，提高了效率。我们在这个函数中做一些修改，如果 Java 本地方法来自代理类，那么反射得到 CAR 构件中对应方法的信息，并将其保存起来。

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

```
ClassObject *clazz = method->clazz;
void            *func;
……
if(IS_CLASS_FLAG_SET(clazz, CLASS_CAR_CLASS|CLASS_CAR_INTERFACE))
{
    if (!dvmSetCARMethodInfo(method)) {
        if (method->pCARMethodInfo == 1)
            goto A_NORMAL_NATIVE_METHOD;
            dvmThrowException("Ljava/lang/UnsatisfiedLinkError;",
                    method->name);
            return;
        }
            ……
```

首先还是通过判断 Java 对象中有没有 CLASS_CAR_CLASS 标识来判断当前对象是否是一个代理类对象。对于代理类对象，得到对应 CAR 构件方法的信息，并将其保存在方法的结构体中。为此，特意在方法的结构体中增加了一个 CARMethodInfo 的结构体用来保存 CAR 构件方法的信息。

```
typedef struct CARMethodInfo{
    //Elastos: inside CAR support, this is the IMethodInfo which is Reflect from
    //CARClass.   For no CAR class, this will be NULL.
    u4              pMethodInfo;
    u2              interfaceIndex;
    short           idxOutParam;   //CAR [out] parameter index(position)
    u4*             regTypes;      //this is the types based on the method signature
    u4*             regExtraData; //the ExtraData
    bool            needCopy;
}CARMethodInfo;

struct Method {
    /* the class we are a part of */
    ClassObject*       clazz;
    ……
    CARMethodInfo* pCARMethodInfo;
};
```

和修改 ClassObject 结构体一样，也是在结构体的末尾添加一个新成员变量 CARMethodInfo。这个成员变量会保存 CAR 构件的方法信息 IMethodInfo 和方法参数信息。dvmSetCARMethodInfo 是通过反射的 CAR 构件中方法的信息，并将其保存在 Method 结构体中的 CARMethodInfo 成员。

```
bool dvmSetCARMethodInfo(Method* meth)
{
```

```
ClassObject         *clazz = meth->clazz;
IMethodInfo         *pMethodInfo = NULL;
IClassInfo          *pClassInfo = NULL;
ECode                ec = -1;
Int32                paramCount;
Int32                i, k, outParamNum;
……
CARMethodInfo       *p;
char                 *methodName;
methodName = dvmGetCARMethodAnnotationValue(meth, (char*)"value");
if (methodName == NULL || *methodName == '\0')
    methodName = (char*)meth->name;
    if (IS_CLASS_FLAG_SET(clazz, CLASS_CAR_CLASS)) {
        pClassInfo = (IClassInfo*)(clazz->pClassInfo);
        if (pClassInfo != NULL) {
            ec = GetMethodInfoAndInterfaceIndex(pClassInfo,
                                methodName, &interfaceIndex, &pMethodInfo);
        }
    } else if (IS_CLASS_FLAG_SET(clazz, CLASS_CAR_INTERFACE)){
        ……
    }
    ……
    p = (CARMethodInfo*)dvmLinearAlloc(clazz->classLoader,
                                sizeof(CARMethodInfo));
    p->pMethodInfo = (u4)pMethodInfo;
    p->regTypes = (u4*)dvmLinearAlloc(clazz->classLoader,
                                (meth->insSize+1)*sizeof(u4)); //+1 for outs
    p->regExtraData = (u4*)dvmLinearAlloc(clazz->classLoader,
                                (meth->insSize+1)*sizeof(u4)); // +1 for outs
    paramDesc = (const char**)_alloca((meth->insSize+1)*sizeof(char *));
    p->needCopy = false;
    p->interfaceIndex = interfaceIndex-1;
    meth->pCARMethodInfo = p;
    int actualArgs = 0;
    ……
    return true;
}
```

从方法的注释中得到 CAR 构件对应的方法的名称，如果为空，则默认 CAR 构件中的方法和代理类中方法的名称是一致的。然后根据方法中的名称可以从 IClassInfo 的方法信息 IMethodInfo。最后从 Java 的堆上分配内存存放 IMethodInfo 和参数信息，将其赋值给代理类方法的数据结构中。在这里要指出的是，在调用第一次调用 CAR 构件方法时，为每个方法反射一次参数的信息，并且将其保存

在方法结构体中，这样以后再次调用时可以直接使用这个参数信息，而不需要查询，提高了 Java 和 CAR 构件之间方法调用时参数传递的效率。关于这一点会在参数传递和转换部分做详细介绍。

4.3.5 调用 CAR 构件中的方法

接下来继续执行 dvmResolveNativeMethod，要做的事情是取得对应的 CAR 构件对象，进而找到函数地址。

```
        Object *object = (Object*)args[0];
        int         i;
        ClassObject *clz;
        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);
        ……
    if (pCARObject != NULL)
        func = dvmGetCARMethodAddr(method, pCARObject);
    else
        func = NULL;
    ……
```

在这里 args[0]是 Java 的对象，而从对象后的额外分配内存中可以取得 CAR 构件对象。然后调用 dvmGetCARMethodAddr 方法从这个对象中取得代理类本地方法对应的 CAR 构件函数的地址。

```
void *dvmGetCARMethodAddr(Method* meth, u4 *pCARObject)
{
    VObject         *vobj = (VObject*)(pCARObject);
    CMethodInfo *pMethodInfo;
    UInt32             index;
    pMethodInfo = (CMethodInfo*)meth->pCARMethodInfo->pMethodInfo;
    index = pMethodInfo->m_uIndex;
    meth->pCARMethodInfo->interfaceIndex);
    vobj += meth->pCARMethodInfo->interfaceIndex;
    return vobj->vtab->methods[METHOD_INDEX(index)];
}
```

在详细研究了 CAR 构件对象的组成结构后，我们发现可以通过虚表来得到函数的地址。通过之前得到的 IMethodInfo 信息可以获得函数在对象的虚表中的索引，而虚表存放的正是函数的实际地址。

再次回到 dvmResolveNativeMethod 函数，真正进入虚拟机调用 CAR 构件的函数部分。

```
                ……..
                if(func != NULL) {
                    if (dvmIsSynchronizedMethod(method))
                        dvmSetNativeFunc(method,
                                dvmCallSynchronizedCARJNIMethod, (const u2*)func);
                    else
                        dvmSetNativeFunc(method,
                                dvmCallCARJNIMethod, (const u2*)func);
                    dvmCallCARJNIMethod(args, pResult, method, self);
                    return;
                } else {
                    …..
                }
            }
        ……
        }
```

通过 dvmSetNativeFunc 函数将方法结构体中的 nativeFunc 设置为 dvmCallCARJNIMethod，而将是将 CAR 构件方法的地址赋值给 insns 成员。这样当 Java 程序每次调用代理类中的本地方法时都会触发虚拟机调用 dvmCallCARJNIMethod 方法，在这个方法中完成对 CAR 构件方法的调用。

```
void dvmCallCARJNIMethod(const u4* args, JValue* pResult,
                const Method* method, Thread* self)
{
    ……
    u4          *pCARObject;
    int         insSize = method->insSize;
    if (dvmIsStaticMethod(method)) {
        ……
    } else {
        Object* object = (Object*)args[0];
        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*)*(u4*)(((char*)object)
                                + method->clazz->objectSize + sizeof(u4)*i);
        insSize--;
        args++;
    }
```

```
            if (!method->pCARMethodInfo->needCopy){
                ec = dvmPlatformCARInvoke(
                        pCARObject+method->pCARMethodInfo->interfaceIndex,
                        method->jniArgInfo, insSize, args, method->shorty,
                        (void*)method->insns, pResult);
                if (ec) {
                    char    buf[256];
                    dvmThrowException("Ljava/lang/IllegalAccessError;", buf);
                }
            } else {
                ……
                ec = dvmPlatformCARInvoke(
                        pCARObject+method->pCARMethodInfo->interfaceIndex,
                        method->jniArgInfo, insSize, argsCopy+1, method->shorty,
                        (void*)method->insns, pResult);
                if (ec) {
                    dvmThrowException("Ljava/lang/IllegalAccessError;", buf);
                }
                ……
            }
        }
    }
```

在调用 CAR 构件函数是会有两种情况出现，一种需要参数转换，另外一种不需要参数转换。对于不需要参数转换的情况，直接调用 dvmPlatformCARInvoke 函数来触发 CAR 构件方法。需要参数转换的情况先将 Java 类型的参数转换为 CAR 构件对应的参数类型，将参数组织好再调用 dvmPlatformCARInvoke 函数，返回时再将参数从 CAR 构件类型的参数转换为 Java 类型。

dvmPlatformCARInvoke 是一个跟平台相关的汇编语言函数，它负责触发 CAR 构件函数。将调用 CAR 构件方法需要的参数压入栈中，然后跳转到 CAR 构件函数地址执行。

```
dvmPlatformCARInvoke:
    .fnstart
    mov         ip, sp                              @ ip<- original stack pointer
    .save {r4, r5, r6, r7, r8, r9, ip, lr}
    stmfd     sp!, {r4, r5, r6, r7, r8, r9, ip, lr}
    mov         r4, ip                              @ r4<- original stack pointer
    …….
    @ Expand the stack by the specified amount.    We want to extract the
    @ count of double-words from r1, multiply it by 8, and subtract that
    @ from the stack pointer.
    and       ip, r1, #0x0f000000               @ ip<- doube-words required
    sub       sp, sp, ip, lsr #21               @ shift right 24, then left 3
```

```
mov       r9, sp                          @ r9<- sp    (arg copy dest)
mov       r8, r3                           @r8 hold the argv
mov       r7, r2
mov       r6, r1
.Lfast_copy_loop:
     @ if (--argc < 0) goto invoke
     subs       r7, r7, #1
     bmi        .Lcopy_done                      @ NOTE: expects original argv in r9
.Lfast_copy_loop2:
     @ Get pad flag into carry bit.    If it's set, we don't pull a value
     @ out of argv.
     movs       r6, r6, lsr #1
     ldrcc      ip, [r8], #4                  @ ip = *r8++ (pull from argv)
     strcc      ip, [r9], #4                  @ *r9r6, r2++ = ip  (write to stack)
     bcc        .Lfast_copy_loop
     add        r9, r9, #4                     @ if pad, just advance ip  without store
     b          .Lfast_copy_loop2               @ don't adjust argc after writing pad
.Lcopy_done:
     ……
     ldrne      ip, [r4, #8]                  @ ip = out
     strne      ip, [r9], #4                  @ *r9++ = ip (write to stack)
     @ call the method
     ldr        ip, [r4, #4]                  @ func
     blx        ip
     ……
.fnend
```

这是一段 ARM 汇编程序，在 ARM 汇编中，方法之间利用 r0~r3 寄存器来传参数，多余的参数放在栈上。所以当 dvmPlatform 被调用时，r0 存放的是 CAR 构件的对象，r1 和 r2 存放的是参数的描述信息和个数，CAR 构件方法需要的参数在 r3 寄存器中，方法的签名存放在栈顶，方法的地址在相对于栈顶 4 个偏移量的位置，而返回结果 pResult 在相对于栈顶 8 个偏移量的位置。调用 CAR 构件方法时，使 r0 寄存器存放 CAR 构件的对象，这是根据 C++函数的调用规则，第一个参数总是为 this 指针。r1~r3 寄存器存放真正的参数，当 r1~r3 放不下时放到寄存器上。然后利用 blx 指令跳转到 CAR 构件方法的地址，CAR 方法被调用，当调用返回时。r0~r1 存放的是返回值，将这个值保存到 pResult。pResult 最后会返回到 dvmCallCARJNIMethod，这样就完成了一次 Java 对 CAR 构件函数的调用。

4.4 参数的传递和转换

CAR 构件编程模型和 Java 编程模型有不同的参数类型，当在 Java 方法调用 CAR 构件方法以及从 CAR 构件方法返回到 Java 方法时会发生参数传递和转换，目的是使两种语言的参数类型能够互相匹配，从而实现方法的正确调用和返回。

在 Java 编程语言中，参数可以分为基本类型和引用类型，基本类型又包括 boolean，byte，short，int，long，char，float 和 double 八种类型。详细描述如表 4.1 所示。

表 4.1 Java 参数类型

数据类型			描述
基本类型		boolean	只有 true 和 false 两个值，虚拟机内部通常用 int 或 byte 来表示，整数零表示 false，所有非零值表示 true
	整数类型	byte	8 位有符号整数，取值范围（-2^7~2^7-1）
		short	16 位有符号整数，取值范围（-2^{15}~2^{15}-1）
		int	32 位有符号整数，取值范围（-2^{31}~2^{31}-1）
		long	64 位有符号整数，取值范围（-2^{63}~2^{63}-1）
		char	16 位无符号整数，取值范围（0~2^{16}-1）
	浮点类型	float	32 位标准单精度浮点数，IEEE754 标准定义
		double	64 位标准双精度浮点数，IEEE754 标准定义
引用类型		reference	对某对象的引用，位于堆中

CAR 构件中的定义的数据类型如表 4.2 所示。

表 4.2 CAR 支持的数据类型

CAR 的数据类型	描述
Int8	8 位有符号整数
Byte	8 位无符号整数
Int16	16 位有符号整数
Int32	32 位有符号整数
Int64	64 位有符号整数
Float	32 位 IEEE 浮点数
Double	64 位 IEEE 浮点数
Boolean	字符类型

AChar	窄字符类型
WChar	宽字符类型
AString	指向一个常量窄字符串（8 位字符串）的指针
WString	指向一个常量宽字符串（16 位字符串）的指针
AStringBuf	存储用户窄字符串的缓冲区数据结构
WStringBuf	存储用户宽字符串的缓冲区数据结构
BufferOf	一种具有自描述功能的 T 类型数组，操作对象是 T 类型的数据块
ArrayOf	一种具有自描述功能的 T 类型数组
MemoryBuf	存储用户数据块的缓冲区数据结构
struct	结构体
ECode	该数据项是 32 位整数，标准的返回类型
enum	枚举类型
IInterface	接口指针，包括系统接口和自定义接口
EMuid	Modul 唯一 ID
EGuid	类标示符，用于唯一标示一个 CAR 类
EIID	接口标示符，用于唯一标示一个特定的 CAR 接口

经过对两个编程语言数据类型的仔细研究，我们发现可以对各个数据类型做如表 4.3 的对应即可满足我们虚拟机支持 CAR 构件的参数转换要求。

表 4.3 Java 和 CAR 构件数据类型对应关系

Java 类型	CAR 类型	描述
boolean	Boolean	8 位整数
byte	Int8	8 位有符号整数
short	Int16	16 位有符号整数
int	Int32	32 位有符号整数
long	Int64	64 位有符号整数
char	Char16	16 位无符号整数
float	Float	32 位 IEEE 浮点数
double	Double	64 位 IEEE 浮点数
Object	Iterface	对象
String	String	字符串对象
StringBuffer	StringBuf	存储字符串的缓冲区数据结构
[](数组对象)	ArrayOf 或 BufferOf	数组对象

值得指出的是，String，StringBuffer 在 Java 中做为引用类型分别和 CAR 构件中的内置类型 String 和 StringBuf 相对应。另外 Java 中的数组结构和 CAR 构件中的 ArrayOf 或 BufferOf 相对应，即当传入一个 Java 的数组时，根据数组类的数据创建 CAR 构件的 ArrayOf 或 BufferOf 类型的对象，创建哪种对象这是根

据 CAR 构件定义的类型来决定的。

4.5 CAR 构件回调模型的实现

回调编程模型是 CAR 构件非常优秀的编程模型，所以我们在 CAR 构件优化虚拟机中应该支持这一模型。

在虚拟机中的实现中，由于客户端是 Java 程序，相对于普通的回调模型，我们采用如图 4.2 的代理回调模型来支持 CAR 构件的回调。

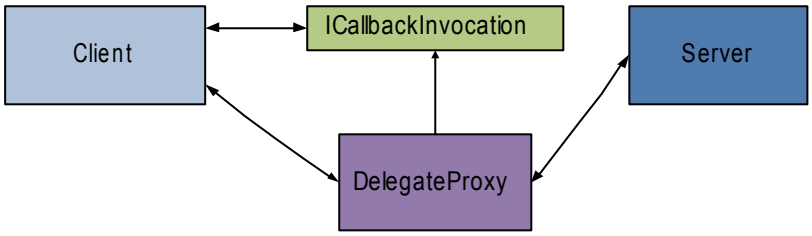

图 4.2 代理回调模型

在这个模型中，Client 是 Java 程序，Server 端是 CAR 构件。客户端和构件并不直接对话，而是通过中间层 DelegateProxy 来进行联系。Java 端注册回调事件不是直接注册在 CAR 构件之上，而是注册在中间层 DelegateProxy 之上，它是 Server 的一个代理。注册回调代理后，构件就可以触发该代理。进而调用 ICallbackInvocation 中的 Invoke 方法。也即当构件发生回调事件是，它会通过代理调用 ICallbackInvocation 的 Invoke 方法，在这个 Invoke 方法中我们实现对 Java 注册的回调方法的调用。

通过前面的分析，CAR 构件的回调事件只能注册 Applet 线程之上，而不能注册在普通的线程上。每个 Elastos 程序都会启动一个 Applet 消息循环线程来进行回调函数的处理，实际上这个消息循环线程是 CAR 构件回调机制需要的环境。如果想要在 Dalvik 中实现 CAR 构件的回调编程模型，首先需要将这个环境引入进来。为了达到这个目的，我们在 Dalvik 中也启动一个类似 Applet 消息守护进程。在 Dalvik 启动时，我们启动一个消息线程，它的入口函数为 MsgLoopTHread，这个线程能够提供 CAR 构件回调需要的环境。

```
static void* MsgLoopThread(void *vptr_args)
{
    Thread    *self;
    ECode       ec;
    JavaVMAttachArgs attachArgs;
    attachArgs.version = JNI_VERSION_1_2;
    attachArgs.name = "MsgLoopThread";
    attachArgs.group = (jobject)dvmGetSystemThreadGroup();
    dvmAttachCurrentThread(&attachArgs,true);
```

```
        self = dvmThreadSelf();
        dvmChangeStatus(self, THREAD_VMWAIT);
        IInterface *pOrgCallbackContext;
        IInterface *m_pCallbackContext;
        _Impl_CallbackSink_InitCallbackContext(&m_pCallbackContext);
        pOrgCallbackContext = (PInterface)pthread_getspecific(TL_CALLBACK_SLOT);
        if (NULL != pOrgCallbackContext) {
            pOrgCallbackContext->Release();
        }
        pthread_setspecific(TL_CALLBACK_SLOT, m_pCallbackContext);
        pAppletObj = m_pCallbackContext;
        gDvm.pCallbackInvocation =
                (unsigned int)AcquireCallbackInvocation(self->jniEnv);
        if ((void*)gDvm.pCallbackInvocation == NULL)
            LOGD("AcquireCallbackInvocation error!");
        pthread_mutex_lock(&appletMutex);
        pthread_cond_signal(&appletCond);
        pthread_mutex_unlock(&appletMutex);
        ec = _Impl_CallbackSink_TryToHandleEvents(m_pCallbackContext);
        dvmChangeStatus(self, THREAD_RUNNING);
        dvmDetachCurrentThread();
        return NULL;
    }
```

pAppletObj 是一个句柄，客户端利用这个句柄可以向中间层 DelegateProxy 注册注册回调事件。_Impl_CallbackSink_TryToHandleEvents 会不断循环，如果服务器端激发了客户端注册的回调事件，则调用客户端相应的回调函数。这样就提供了 CAR 构件回调所需要的环境，接下来的事情是考虑如何在 Java 端注册回调事件，以及如何处理回调。

为了在 Java 端可以按照 Java 编程模型的方式注册回调，我们在系统库中增加注册回调的函数，这样在编程中 Java 注册回调就可以通过系统的 API 来实现。

```
package dalvik.CAR;
public class CARCallbackFunc{
    native public static void addCallbackHandler(Object obj, String callbackMethodName,
            String className, String methodName, String signature);
    native public static void addCallbackHandlerNoSignatire(Object obj,
            String callbackMethodName, String className, String methodName);
    native public static void removeCallbackHandler(Object obj,
            String callbackMethodName, String className,
            String methodName, String signature);
    native public static void removeAllCallbackHandler(Object obj);
    native public static int GetJavaMethodNoSignature(String className,
            String methodName);
```

```
    native public static int GetJavaMethodNoSignature(Object obj, String methodName);
    native public static int GetJavaMethodInSameClass(String methodName);
    native public static int Object2Int(Object obj);
}
```

其中 addCallbackHandler 是向一个 CAR 构件对象中注册回调函数，参数 obj 是 CAR 构件对象的代理对象，callbackMethodName 是要注册的回调事件名称，className，methodName，signature 分别表示的是要注册的回调函数的所属类，名称和原型。

addCallbackHandler 是虚拟机内部的本地方法，它的实现如下。

```
static void Dalvik_dalvik_CAR_CARCallbackFunc_addCallbackHandler(const u4* args,
        JValue* pResult)
{
    Object* CARObj = (Object*)args[0];
    StringObject* nameObj;
    nameObj = (StringObject*)args[1];
    char* callbackMethodName = dvmCreateCstrFromString(nameObj);
    nameObj = (StringObject*)args[2];
    ClassObject* clazz;
    if (CARObj->clazz->classLoader != NULL) {
        clazz = dvmFindClassByName(nameObj, CARObj->clazz->classLoader, true);
    } else {
        clazz = dvmFindClassByName(nameObj, dvmGetSystemClassLoader(), true);
    }
    nameObj = (StringObject*)args[3];
    char* methodName = dvmCreateCstrFromString(nameObj);
    nameObj = (StringObject*)args[4];
    char* methodDescript = dvmCreateCstrFromString(nameObj);
    Method* method = dvmFindDirectMethodByDescriptor(clazz,
                                    methodName, methodDescript);
    dvmAddCallbackHandler(CARObj, callbackMethodName, clazz, method);
    free(callbackMethodName);
    free(methodName);
    free(methodDescript);
}
```

在这个函数中分别从参数中取得要注册回调函数的类名，函数名后，找到相应的类和函数，然后通过 dvmAddCallbackHandler 将回调函数注册到 DelegateProxy 上的相应事件。

```
bool dvmAddCallbackHandler(Object* CARObject, char* callbackMethodName,
                ClassObject* clazz, Method* method)
{
    ClassObject    *clz;
    int                i;
```

```
    ICallbackInvocation *pCallbackInvocation;
    pCallbackInvocation = (ICallbackInvocation *)gDvm.pCallbackInvocation;
    if (pCallbackInvocation == NULL)
        pCallbackInvocation = AcquireCallbackInvocation(dvmThreadSelf()->jniEnv);
    IDelegateProxy *piDelegateProxy = NULL;
    ICallbackMethodInfo *pCallbackInfo = NULL;
    for (i = 0, clz = CARObject->clazz;  clz != method->clazz && clz != NULL;
            clz = clz->super) {
        if (IS_CLASS_FLAG_SET(clz, CLASS_CAR_CLASS)) {
            if (IsCallback(callbackMethodName,
                    (IClassInfo*)clz->pClassInfo, &pCallbackInfo))
                    break;
                i++;
            }
        }
        if (clz == NULL)
            return  false;
    ECode ec = pCallbackInfo->CreateDelegateProxy((void*)clazz,
                    (void*)method, pCallbackInvocation, &piDelegateProxy);
    if (FAILED(ec)) {
        pCallbackInfo->Release();
        return false;
    }
    EventHandler handler;
    piDelegateProxy->GetDelegate(&handler);
    pCallbackInfo->AddCallback((IInterface*)(*(u4*)(((char*)CARObject)
                    +CARObject->clazz->objectSize+sizeof(u4)*i)), handler);
    return true;
}
```

这个函数将创建 DelegateProxy，并将方法注册在 DelegateProxy 对应的事件上，这样就完成了将 Java 方法注册到回调代理。

另一方面，当 CAR 构件中触发某个事件时，代理回调会触发 ICallbackInvocation 中的 Invoke 方法，在这个方法中我们调用注册的 Java 回调方法。

```
ECode CJavaCallbackInvocation::Invoke(
    /* [in] */ PVoid pTargetObject,
    /* [in] */ PVoid pTargetMethod,
    /* [in] */ ICallbackArgumentList * pCallbackArgumentList)
{
    ECode ec;
    ……
    ec = OrganizeParameters(pICallbackMethodInfo,
```

```
                    pCallbackArgumentList, args, paramCount);
        ……
        JValue    rt;
        Thread *self = dvmThreadSelf();
        dvmChangeStatus(self, THREAD_RUNNING);
        dvmCallMethodA(self, (Method*)pTargetMethod, (Object*)pTargetObject, &rt, args);
        dvmChangeStatus(self, THREAD_NATIVE);
        pICallbackMethodInfo->Release();
        return NOERROR;
    }
```

在这个函数中首先从 pCallbackArgumentList 反射出调用 Java 回调方法需要的参数，并将参数的格式转换成 Java 可以接受的格式。最后通过 dvmCallMethodA 可以调用 Java 回调方法，除了调用参数，它需要的参数是 Java 方法和 Java 对象，而这两个参数都是之前 Java 客户端注册在回调代理上的，可以直接使用。

4.6 内存统一管理机制

由于 Java 虚拟机使用垃圾自动回收机制，Java 对象的内存我们不需要关心。所以内存统一管理机制实际上是要保证我们在虚拟机中引入的 CAR 构件不会造成虚拟机的内存泄露，而这个实现需要依靠 Java 虚拟机的垃圾自动回收机制。

虚拟机中引入 CAR 构件主要有 2 个地方产生内存需要我们去做释放，

（1） 引入 CAR 构件时会在代理类的类对象和方法对象上分配相应的内存来保存相应的 CAR 构件信息，所以当代理类被卸载时，需要释放这部分的内存。

（2） CAR 构件的代理对象被创建时会创建相应的 CAR 构件对象，那么当代理对象被垃圾回收时，CAR 构件对象也要相应的析构。

对于（1），当虚拟机要卸载某个类时会调用 dvmFreeClassInnards，所以在这个类中释放在类对象和方法对象为 CAR 构件存放信息的内存。

```
void dvmFreeClassInnards(ClassObject* clazz)
{
        void *tp;
        int i;
        if (IS_CLASS_FLAG_SET(clazz, CLASS_CAR_CLASS
                |CLASS_CAR_INTERFACE))
        {
                dvmReleaseCARClassInfo(clazz);
        }
        ……
        if (clazz->directMethods != NULL) {
```

```
            ……
            for (i = 0; i < directMethodCount; i++) {
                freeMethodInnards(&directMethods[i]);
            }
            dvmLinearFree(clazz->classLoader, directMethods);
        }
        ……
    }
```

首先判断被卸载的类是否是一个代理类，如果是一个代理类，那么调用 dvmReleaseCARClassInfo 来释放类对象上为 CAR 构件分配的内存。

```
    void dvmReleaseCARClassInfo(ClassObject* clazz)
    {
        if ((void*)clazz->pClassInfo != NULL) {
            IClassInfo *pClassInfo = (IClassInfo*)clazz->pClassInfo;
            clazz->pClassInfo = NULL;
            pClassInfo->Release();
        }
        if ((void*)clazz->pStaticClassInfo != NULL) {
            IClassInfo *pStaticClassInfo = (IClassInfo*)clazz->pStaticClassInfo;
            clazz->pStaticClassInfo = NULL;
            pStaticClassInfo->Release();
        }
    }
```

而在 freeMethodInnards 中释放方法对象中为 CAR 构件分配的内存。

```
    static void freeMethodInnards(Method* meth)
    {
        ClassObject* clazz = meth->clazz;
        if (IS_CLASS_FLAG_SET(clazz, CLASS_CAR_CLASS
                |CLASS_CAR_INTERFACE)) {
            if (meth->pCARMethodInfo != NULL)
            {
                dvmReleaseCARMethodInfo(meth);
                if (meth->pCARMethodInfo->regTypes != NULL){
                    void* regTypes = meth->pCARMethodInfo->regTypes;
                    meth->pCARMethodInfo->regTypes = NULL;
                    dvmLinearFree(clazz->classLoader, regTypes);
                }
                if (meth->pCARMethodInfo->regExtraData != NULL){
                    void* regExtraData = meth->pCARMethodInfo->regExtraData;
                    meth->pCARMethodInfo->regExtraData = NULL;
                    dvmLinearFree(clazz->classLoader, regExtraData);
                }
                void* pCARMethodInfo = meth->pCARMethodInfo;
```

```
                meth->pCARMethodInfo = NULL;
                dvmLinearFree(clazz->classLoader, pCARMethodInfo);
            }
        }
    }
```

这样，当代理类被虚拟机卸载时，附加在类对象和方法对象 CAR 构件信息相应也被释放。

对于新建 CAR 构件对象的内存，还是利用虚拟机的垃圾回收机制，当代理类对象被释放时，释放相应 CAR 构件的对象。

```
void dvmReleaseTrackedAlloc(Object* obj, Thread* self)
{
    ClassObject* clz;
    u4                pCARObject;
    int               layerCount = 0;
    ……
    for (clz = obj->clazz;  clz->super != NULL;  clz = clz->super) {
        if (IS_CLASS_FLAG_SET(clz, CLASS_CAR_CLASS)) {
            pCARObject = *(u4 *)(((char *)obj)
                    + obj->clazz->objectSize + sizeof(u4)*layerCount);
            if (pCARObject == (u4)NULL) {
                if (!dvmIsCARClassGeneric(clz)) {
                    dvmAbort();
                }
            }
            dvmReleaseCARClassObject(pCARObject);
            layerCount++;
        } else if (IS_CLASS_FLAG_SET(obj->clazz, CLASS_CAR_INTERFACE)) {
            pCARObject = *(u4 *)(((char *)obj) + obj->clazz->objectSize + sizeof(u4));
            if (pCARObject != (u4)NULL) {
                dvmReleaseCARClassObject(pCARObject);
            }
            break;
        }
    }
    ……
}
```

当 Java 对象被垃圾回收时，虚拟机会调用 dvmReleaseTrackedAlloc 来完成最后的清理工作，我们在这里对 CAR 构件对象进行释放。假如回收的对象是一个代理类对象，可以从这个代理类对象后面的额外内存中取得它对应的 CAR 构件对象，并且调用 dvmReleaseCARClassObject 进行释放。

```
void dvmReleaseCARClassObject(const u4 pObject)
{
```

```
        if ((void*)pObject != NULL )
            ((PInterface)pObject)->Release();
    }
```

这样，代理类对象回收时 CAR 构件对象也相应回收，CAR 构件对象的生命周期和代理类对象的生命周期是一致的。

经过内存检测工具检测，CAR 构件优化 Dalvik 虚拟机的实现没有内存泄露的情况，所以我们对于内存管理的考虑是周全的。

第 5 章 CAR 构件优化虚拟机模型编程实例和性能分析

在本章中，将通过几个简单的例子来展示如何在 CAR 构件优化虚拟机模型上进行编程，并且通过性能测试来分析此模型的运行效率。

5.1 CAR 构件优化虚拟机模型编程实例

5.1.1 简单的计算器例子

在这个例子中，我们在 CAR 构件中完成整数的加，减，乘，除四则运算，而在 Java 程序中使用 CAR 构件的函数来完成计算。CAR 构件的描述文件 Calculate.car 如下：

```
module
{
    interface ICalculate
    {
        Add([in] Int32 x, [in] Int32 y, [out] Int32* o);
        Sub([in] Int32 x, [in] Int32 y, [out] Int32* o);
        Div([in] Int32 x, [in] Int32 y, [out] Int32* o);
        Mul([in] Int32 x, [in] Int32 y, [out] Int32* o);
    }
    class CCalculate
    {
        interface ICalculate;
    }
}
```

可以看到，这个 CAR 构件有一个 CCalculate 类具有 ICalculate 接口，而接口则包括 Add，Sub，Div，Mul 四个函数。在 Elastos 编译环境中，通过 Calculate.car 自动生成 CAR 构件的源文件 CCalculate.cpp，实现具体的函数功能。往其中填充代码如下：

```
#include "CCalculate.h"
ECode CCalculate::Add(
        /* [in] */ Int32 x,
        /* [in] */ Int32 y,
        /* [out] */ Int32 * pO)
{
    *pO = x+y;
    return NOERROR;
}
```

```
ECode CCalculate::Sub(
        /* [in] */ Int32 x,
        /* [in] */ Int32 y,
        /* [out] */ Int32 * pO)
{
        *pO = x-y;
        return NOERROR;
}
ECode CCalculate::Div(
        /* [in] */ Int32 x,
        /* [in] */ Int32 y,
        /* [out] */ Int32 * pO)
{
        *pO = x/y;
        return NOERROR;
}
ECode CCalculate::Mul(
        /* [in] */ Int32 x,
        /* [in] */ Int32 y,
        /* [out] */ Int32 * pO)
{
        *pO = x*y;
        return NOERROR;
}
```

最后通过 Elastos 编译环境可以生成 CAR 构件 Calculate.eco，这个 CAR 构件提供整数的运算函数。同时通过 CAR 构件的描述文件也可以自动生成 CAR 构件的代理 Java 类文件。

```
import dalvik.annotation.ElastosClass;
@ElastosClass(Module="Calculate.eco", Class="CCalculate")
class CCalculate{
        native static int Add(int x, int y);
        native static int Sub(int x, int y);
        native static int Div(int x, int y);
        native static int Mul(int x, int y);
}
```

其中 ElastosClass 注释类指明了这个代理类所对应的 CAR 构件时 Calculate.eco，而对应的类是 CAR 构件中的 CCalculate 类。我们编写一个小的 Java 程序来测试这个 CAR 构件是否能真正被 Java 所调用

```
public class Main{
        public static void main(String[] args)
        {
                System.out.println("test start here\n");
```

```
        CCalculate mCalculate = new CCalculate();
        int i = mCalculate.Add(2,4);
        System.out.println(i);
    }
}
```

运行这个 Java 程序可以看到 Java 传入参数 2 和 4，CAR 构件接受来自 Java 的参数，调用 Add 方法，计算出结果为 6 并且返回给 Java。屏幕输出值为 6，这样就实现了 Java 对 CAR 构件的调用。

5.1.2 带有回调事件的编程例子

当虚拟机引入 CAR 构件后，CAR 构件中的回调编程模型同样也能在 Java 编程模型中运用，这对 Java 编程模型是一个补充。下面这个例子来解释如何在 Java 程序中使用 CAR 构件的回调编程模型。

首先定义 CAR 构件的描述文件 FooBarDemo.car

```
module
{
    interface IFoo {
        Foo();
    }
    interface IBar {
        Bar();
    }
    callbacks JFooEvent {
        FooEvent();
    }
    class CFooBar {
        interface IFoo;
        interface IBar;
        callbacks JFooEvent;
    }
}
```

这个 CAR 构件有两个方法，分别是 Foo 和 Bar，并且有一个 FooEvent 的回调事件。同样的通过 Elastos 编程环境可以生成 CAR 构件的框架，在 CFooBar.cpp 中实现 CAR 构件的具体功能。

```
#include "CFooBar.h"
#include "stdio.h"
ECode CFooBar::Foo()
{
    printf("Foo\n");
```

```
        Callback::FooEvent();
        return NOERROR;
    }
    ECode CFooBar::Bar()
    {
        printf("Bar\n");
        return NOERROR;
    }
```

Foo 和 Bar 都是打印一条语句，并且在 Foo 方法中触发 FooEvent 回调事件。所以当程序注册了 FooEvent 回调事件后，每当调用 Foo 方法时就会触发相应的回调。在 Java 端，生成 CAR 构件的代理 Java 类如下。

```
    import dalvik.annotation.ElastosClass;
    @ElastosClass(Module="FooBarDemo.eco", Class="CFooBar")
    class CFooBarDemo{
        native void Foo();
        native void Bar();
    }
```

Java 程序可以向 CAR 构件注册自己的回调，并且调用 CAR 构件的方法。

```
    import dalvik.CAR.CARCallbackFunc;
    public class Main{
        public void Callback()
        {
            System.out.println("Callback from Java!\n");
        }
        public static void main(String[] args)
        {
            System.out.println("test start here\n");

            CFooBarDemo mFooBarDemo = new CFooBarDemo();
            CARCallbackFunc.addCallbackHandler(mFooBarDemo,
                                        "FooEvent", "Main", "Callback", "()V");
            mFooBarDemo.Foo();
            mFooBarDemo.Bar();
        }
    }
```

首先生成 CAR 构件的对象，然后向 CAR 构件对象注册 FooEvent 事件的回调函数为 Callback。addCallbackHandler 接受的参数是要注册的 CAR 构件的对象，要注册的事件名称，回调方法所属的类，名称和原型。最后分别调用 Foo 和 Bar 方法，结果如下

```
    Foo
    Callback from Java!
```

Bar

而 Foo 方法触发了回调事件，也即在执行 CAR 构件的 Foo 方法时，调用了 FooEvent 的 Java 回调方法 Callback。

5.2 性能测试和分析

通过前面的分析，CAR 构件优化虚拟机模型得到的一个直接的好处就是程序运行效率能够得到提高。整型和浮点型数的加、减、乘、除运算能够很好的反映出程序的运行效率，所以我们在相同的环境上测试使用 CAR 优化模型和没有使用 CAR 优化模型的程序进行运算所需要的时间，以分析 CAR 构件优化模型对传统 Java 编程模型性能的提高。

实验环境如下：

- 富士康 F902 手机
- Android1.6 操作系统

表 5.1 是使用 CAR 优化模型和 Java 传统编程模型程序中整型数的运算效率，分别进行两个整数的加减乘除运算十万次。为了得到可靠的数值，进行 10 组数据的对比。

表 5.1 整数的运算效率（单位：ms）

时间 / 次序		加法运算时间（一千万次）	减法运算时间（一千万次）	乘法运算时间（一千万次）	除法运算时间（一千万次）
1	优化模型	4.18ms	2.52ms	2.58ms	8.91ms
	普通模型	9.80ms	8.97ms	8.99ms	14.42ms
2	优化模型	4.49ms	2.71ms	2.51ms	8.82ms
	普通模型	9.48ms	8.87ms	8.78ms	14.48ms
3	优化模型	4.62ms	2.56ms	2.91ms	8.54ms
	普通模型	8.90ms	9.30ms	9.01ms	13.78ms
4	优化模型	4.71ms	2.94ms	5.75ms	8.64ms
	普通模型	8.97ms	9.09ms	9.30ms	13.99ms
5	优化模型	4.68ms	3.25ms	2.73ms	9.83ms
	普通模型	8.89ms	9.18ms	9.20ms	14.31ms
6	优化模型	4.63ms	2.71 ms	2.76 ms	8.41 ms
	普通模型	9.92 ms	9.19 ms	9.08 ms	14.23 ms
7	优化模型	4.61 ms	2.73 ms	2.53 ms	8.71 ms
	普通模型	11.2 ms	8.79 ms	9.19 ms	14.05 ms

8	优化模型	4.46 ms	2.61 ms	2.61 ms	8.71 ms
	普通模型	8.87 ms	9.18 ms	8.47 ms	15.28 ms
9	优化模型	4.19 ms	2.84 ms	2.61 ms	8.47 ms
	普通模型	9.57 ms	9.02 ms	9.29 ms	16.59 ms
10	优化模型	5.01 ms	2.72 ms	2.58 ms	8.73 ms
	普通模型	9.88 ms	10.34 ms	9.89 ms	14.19 ms

可以看到优化模型的运算效率比普通模型的运算效率要高，图 5.1 通过更直观的方式反映了两种模型的性能。

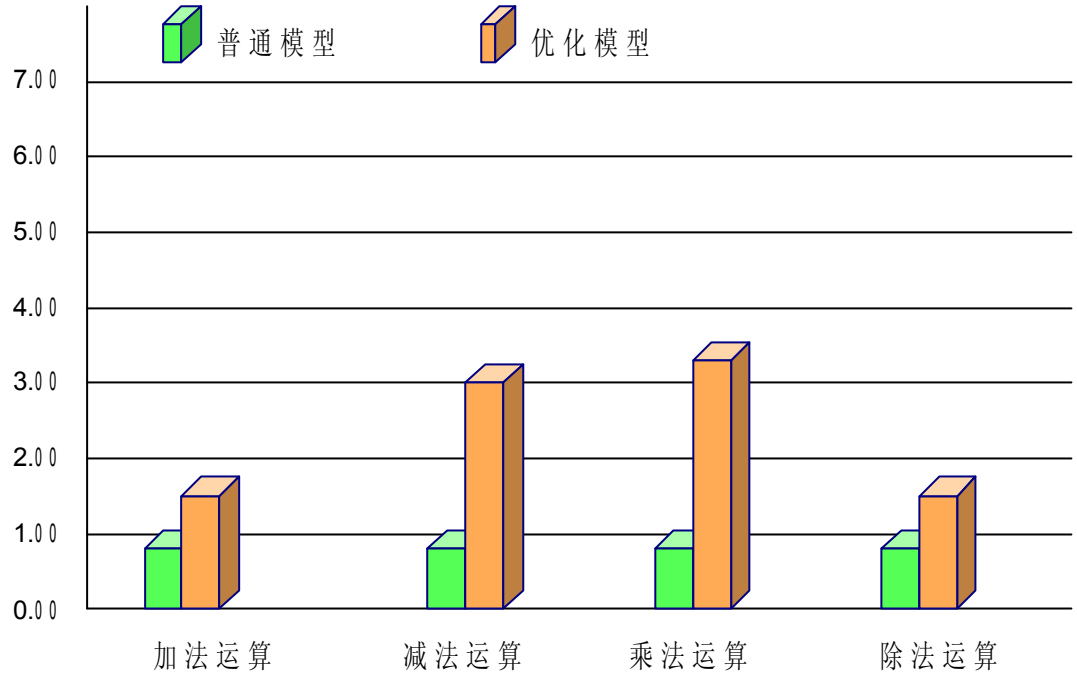

图 5.1 普通模型和优化模型性能比较

可以看到优化模型在加法和除法运算上是普通模型的 1.7~1.8 倍，而在减法和乘法运算效率上能达到 3.1~3.2 倍之多。可见我们的 CAR 构件优化 Java 虚拟机模型确实是大大提高了程序的运行效率。

第 6 章 总结和展望

本文首先详细研究 Java 虚拟机，Dalvik 和 CAR 构件技术。对 Java 虚拟机的组成结构，运行机制进行了简单的介绍，对当前优化 Java 虚拟机的几种技术进行了研究，并指出了各种技术存在的局限性。Google 的 Dalvik 虚拟机是 Java 虚拟机的一种，并且有自己独特的实现方式，使得 Dalvik 特别适合嵌入式的环境来执行 Java 程序。而 CAR 构件技术是科泰世纪自主研发的二进制技术，它通过 Elastos 构件运行环境可以实现跨平台，并且具有自描述、动态加载、动态组装、二进制继承等特性。

在本文的第三章我们提出了一个 CAR 构件优化 Java 虚拟机的模型，并且对优化模型的可行性以及遇到的问题进行了详细探讨。由于 CAR 构件是二进制的代码，所以能够大大提高程序的运行效率。同时由于 CAR 构件运行在 Elastos 构件运行环境中，可以实现跨平台，这样不会破坏 Java 程序的跨平台特性。接着在第四章中，我们在 Dalvik 上实现了这个优化模型，对其详细实现进行了描述，并且对于其中的关键问题参数传递，回调模型，内存管理等进行了详细讨论。最后给出了 CAR 构件优化 Java 虚拟机模型的编程实例，通过一个简单的程序对优化模型的性能进行了测试，并和普通模型进行对比分析。

本文已经完成了既定的研究目标，并且取得了预期的研究成果，展望未来的研究工作，主要有以下两个方面：

（1） Java 编程模型中的类继承关系如何利用 CAR 构件实现需要进一步的分析研究。

（2） CAR 构件中其他一些优秀的编程模型，如面向方面编程模型，如何在优化模型中实现。

致谢

在论文完成之际，我要衷心感谢所有关心和帮助过我的老师和同学们。

衷心感谢导师陈榕教授对我的精心指导以及在硕士期间给予的帮助和照顾。导师渊博的专业知识，精益求精的工作作风，追求真理十年如一日的学术态度深深地影响了我，这是我成长道路上一笔宝贵的财富。

衷心感谢顾伟楠教授在学习和论文上对我无私帮助与教诲，并使我逐渐掌握了如何进行科学研究的方法。

衷心感谢裴禧龙老师对我无微不至的关怀，老师不仅在学术上给了我许多指导，并且言传身教地教会我很多做人的道理，他的谆谆教诲永远都是我一生前进的动力。

衷心感谢一起渡过硕士生涯的同门们，无论在生活上还是学习上都给予我很多无私的帮助。

最后我要感谢我的父母，虽然远在异乡求学，但是却时刻能感受到温暖。您们对我的关心和支持是我在沪求学 7 年来最坚强的后盾，也将是我坚持追求的勇气和力量的源泉。作为您们的儿子，我感到无比的骄傲和自豪。

参考文献

[1] Java Virtual Machine, http://en.wikipedia.org/wiki/JVM，2009

[2] The JavaTM Virtual Machine Specification. http://java.sun.com/docs/books/jvms/

[3] Java Native Interface. http://java.sun.com/docs/books/jni/ 1997

[4] Rob Gordon. Essential JNI: Java Native Interface, Ph/Ptr Essential Series, 1998

[5] Android，http://www.android.com/，2008

[6] Dalvik Virtual Machine. http://www.dalvikvm.com/, 2009

[7] Joshua Engel, Programming for the Java™ Virtual Machine, Addison Wesley, 1999

[8] Yunhe Shi, David Gregg, M. Anton Ertl.Virtual Machine Showdown: Stack Versus Registers. ACM Transactions on Architecture and Code Optimization (TACO), Jan 2008, Vol.4 (4)

[9] Robert Stark, Joachim Schmid, Egon Borger Java and the Java Virtual Machine, 2001

[10] Sheng Liang, Gilad Bracha, Dynamic class loading in the Java virtual machine 1998

[11] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee. LaTTe: a Java VM just-in-time compiler with fast and efficient register allocation, 1999

[12] Yunhe Shi, Kevin Casey, M.Anton Ertl, David Gregg, Virtual machine showdown: Stack versus registers, 2008

[13] Sun Microsystems, Inc. Specification:Kni documentation. http://java.sun.com/javame/reference/docs/kni/html/

[14] Elastos 资料大全.上海科泰世纪科技有限公司，2008 年 10 月

[15] Bill Venners，深入 java 虚拟机(第 2 版)，曹晓钢，蒋靖．机械工业出版社，2003

[16] 探矽工作室．深入嵌入式 Java 虚拟机．中国铁道出版社，2003

[17] 周毅敏 Java 虚拟机的移植与基于 CAR 构件的二次开发．硕士毕业论文. 2009

[18] 闫伟．Java 虚拟机即时编译器的一种实现原理．西北工业大学计算机学院，2007．Vol. 28(5)：58-60

[19] 熊波.构件自描述封装及运行方法.江西理工大学学报.2007．Vol．1：35-38

[20] 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

[21] 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

[22] 陈卫伍，王建民，陈榕. Dalvik 在 CAR 构件运行时钟的研究与实现，电脑知识与技术, 2010.

[23] 谌宁, 覃征. 基于嵌入式 Java 虚拟机的垃圾回收算法, 计算机应用, 2005.

个人简历、在学期间发表的学术论文与研究成果

个人简历：

陈卫伍，男，1986 年 11 月 9 日出生。

2008 年 7 月毕业于同济大学电子与信息工程学院计算机科学与技术专业 获学士学位。

2008 年 9 月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研究生。

已发表论文：

陈卫伍，王建民，陈榕. Dalvik 在 CAR 构件运行时中的应用研究.

电脑知识与技术，2010 年 11 月.