同济大学
TONGJI UNIVERSITY

硕士学位论文

基于 Dalvik 的复合对象技术的研究与实现

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

姓　　名：盛泽昀

学　　号：0820080231

所在院系：电子信息与工程学院计算机系

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

副指导教师：顾伟楠

二〇一一年三月

同濟大學
TONGJI UNIVERSITY

A dissertation submitted to

Tongji University in conformity with the requirements for

the degree of Master of Science

The Research and Implemention of Debugging Technology for Compound Object in Dalvik

(Smart phones and embedded software platform for industrial R & D
Grant No: 2009ZX01039-002-002)

Candidate: Zeyun Sheng

Student Number: 0820080231

School/Department: School of Electronics and Information Engineering

Discipline: Computer Science and Technology

Major: Computer Software and Theory

Supervisor: Rong Chen

March, 2011

基于Dalvik的复合对象调试技术的研究与实现

盛泽昀 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

Android 手机操作系统的应用是基于 Java 语言开发的，它的类 Java 虚拟机 Dalvik 提供了应用的运行时环境。由于 CAR 对象具有 Java 对象的同样特性、同样生命周期及运行时上下文，因此在 Android 计算环境中引入 CAR 构件技术，使它成为 Android 计算环境的一部分，这就使得应用开发者在开发阶段就优先考虑 Android 程序优化，用 C/C++的 native 化程序技术，实现特别的计算要求。

结合 Java 与 CAR 构件技术，人们提出了 Java 与 CAR 混合编程技术，通过把 Java 对象与 CAR 对象复合，产生了一个逻辑对象的两个操作面：Java 语言面、C++语言面，这对提高软件执行效率，满足嵌入式系统计算需要，有相当大的优势。在这样的一个计算环境内，传统的 Java 调试技术具有一定的局限性，因为 Java 无法“理解”复合在其对象上的 CAR 对象。面向 C++的调试技术则过于重视指令和内存细节，无法从对象级、接口级把握发生在虚拟机内部的计算行为。

当 Java 对象与 CAR 对象混合起来，如何调试这些 CAR 对象成为一新技术点。本文提出一种快照与 C/C++调试相结合的复合对象调试技术，对 CAR 运行时环境，运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议把信息传递给调试器，在进入 CAR 构件前插入单步中断，从而让 C/C++调试器得到介入的机会，实现 CAR 构件的二进制调试。本技术基于 Java 调试体系 JPDA，通过对虚拟机内部调试体系 JPDA 的实现，完成对程序的快照调试。最后在不改变 Java 调试器的情况下，使它具有了调试复合对象的能力。

本文首先论述了在 Elastos 平台上的一种 Java 和 CAR 混合编程技术，达到让 Java 虚拟机上运行 CAR 构件目的的同时，利用 CAR 构件提高 Java 程序的运行速度而不破坏 Java 的编程模型。在这种混合对象编程模型中提出一种复合对象的调试技术。利用 Java 自身的调试体系 JPDA，通过 JPDA 中的 JDWP 协议，实现在 Java 程序的 IDE 环境下查看和调试 Java 虚拟机上运行 CAR 构件的详细情况，包括各种接口的扇入扇出，参数，回调函数等。同时也能查看 CAR 构件中注册的 Java 的回调函数，真正实现复合对象的相互调试。

本课题所做的工作已在 Android 手机开发中得到具体应用，能快速有效的调试 Java 和 CAR 混合编程的程序，取得了较好的工程效果。

关键词：Dalvik，虚拟机，CAR，JDWP，快照，调试

ABSTRACT

All applications are based on the Java language in Android mobile operating system. Java Virtual Machine Dalvik of Android provides a runtime environment for all applications. Because CAR's objects have the same characteristics of Java objects, the same life cycle and runtime operational context, the Android's computing environment has added CAR component technology, making the CAR component technology become part of Android's computing environments. it is useful for application developers who can consider Android optimization in the development stage, using C/C++ language's native of technology to meet particularly computing requirements.

Combination of Java and CAR component technology, we propose Java and CAR mixed programming technology, and producing a logical object of the two operation sides through compounding the Java object and CAR object: Java language side, C++ language side. The technology is considerable benefits to improve the efficiency of software and meet the computing needs of embedded system. In such a computing environment, the tradition of Java debugging techniques encounter great limitations, because Java can not "understand" the CAR object which has been compounded with itself .However, Debugging for C++ technology is too much emphasis on instruction and memory details, and can't take place in the calculation behavior of the internal of the virtual machine from the object level, interface level.

When Java object and CAR object have been mixed, how to debug the CAR objects becomes a new technology point. This paper presents compound objects debugging technology for combination of a snapshot and the C/C++ debugging, this technology can have a snapshot of CAR runtime environment, runtime operational context and snapshot of CAR components themselves. Entering a single-step interrupt before inserting CAR component, in order to the C/C++ debugger to get involved in the opportunity to achieve CAR component of the binary debugging through the Java debugging protocol to pass information to the debugger .The technology which is based on Java debugging system JPDA achieves the program debugging, by extending the implementation of JPDA in the virtual machine. This technology has not changed the Java debugger; however it has the ability to debug compound objects.

The paper first discusses a mixture of programming of Java and CAR in the Elastos platform, in order to allow Java virtual machine to run CAR components, meantime, use CAR components to improve the running speed of Java without destroying Java's programming model. Based on this mixed-object programming model, we present a debugging technology of compound objects. Aims to use JDWP in Java's debugging system JPDA under IDE environment to view and debug details of CAR members which run Java virtual machine, including fan-in-fan-out interfaces, parameters, and various callback functions. It also can view the CAR members which have been registered Java's callback function. The real implementation have completely finished for debugging each other between compound objects.

The compound objects debugging technology in this paper has been applied in the development of specific applications in Android OS. It can quickly and efficiently debug Java and CAR mixed programming. In fact, the actual use effect is very well.

Key Words: Dalvik，Virtual machine，CAR，JDWP，Snapshot，Debug

目录

第 1 章 绪论

1.1 背景

在最近几年里，移动通信和互联网成为当今世界发展最快、市场潜力最大、前景最诱人的两大业务。它们的增长速度都是任何预测家未曾预料到的。迄今，全球移动用户已超过 15 亿，互联网用户也已逾 7 亿。中国移动通信用户总数超过 3.6 亿，互联网用户总数则超过 1 亿。这一历史上从来没有过的高速增长现象反映了随着时代与技术的进步，人类对移动性和信息的需求急剧上升。越来越多的人希望在移动的过程中高速地接入互联网，获取急需的信息，完成需要做的事。所以，现在出现的移动与互联网相结合的趋势是历史的必然。目前，移动互联网正逐渐渗透到人们生活、工作的各个领域，短信、下载铃图、移动音乐、手机游戏、视频应用、手机支付、位置服务等丰富多彩的移动互联网应用迅猛发展，正在深刻改变信息时代的社会生活，移动互联网经过几年的曲折前行，终于迎来了新的发展高潮。同时随着 3G 的成熟与普及，移动网络宽带化、IP 化，以及手机终端的智能化，使得手机上的相关应用越来越丰富。在用户需求与网络条件逐步完善的情况下，如此庞大的移动应用市场已经成为最炙手可热的市场，使得 Apple、Nokia、 Google、微软、移动等通信及 IT 行业巨头相继向手机应用市场延伸。

Android 是 Google 开发的基于 Linux 平台的开源手机操作系统。它包括操作系统、用户界面和应用程序 —— 移动电话工作所需的全部软件，而且不存在任何以往阻碍移动产业创新的专有权障碍。Google 与开放手机联盟合作开发了 Android，这个联盟由包括中国移动、摩托罗拉、高通、宏达电和 T-Mobile 在内的 30 多家技术和无线应用的领军企业组成。Google 通过与运营商、设备制造商、开发商和其他有关各方结成深层次的合作伙伴关系，希望借助建立标准化、开放式的移动电话软件平台，在移动产业内形成一个开放式的生态系统。

Android 以 Java 为编程语言，使接口到功能，都有层出不穷的变化，其中 Activity 等同于 J2ME 的 MIDlet，一个 Activity 类（class）负责创建视窗（window），一个活动中的 Activity 就是在 foreground（前景）模式，背景运行的程序叫做 Service。两者之间通过由 ServiceConnection 和 AIDL 连结，达到复数程序同时运行的效果。如果运行中的 Activity 全部画面被其他 Activity 取代时，该 Activity 便被停止（stopped），甚至被系统清除（kill）。

View 等同于 J2ME 的 Displayable，程序人员可以通过 View 类与“XML

layout”将 UI 放置在视窗上，Android 1.5 的版本可以利用 View 打造出所谓的 Widgets，其实 Widget 只是 View 的一种，所以可以使用 xml 来设计 layout，HTC 的 Android Hero 手机即含有大量的 widget。至于 ViewGroup 是各种 layout 的基础抽象类（abstract class），ViewGroup 之内还可以有 ViewGroup。View 的构造函数不需要再 Activity 中调用，但是 Displayable 的是必须的，在 Activity 中，要通过 findViewById()来从 XML 中取得 View，Android 的 View 类的显示很大程度上是从 XML 中读取的。View 与事件（event）息息相关，两者之间通过 Listener 结合在一起，每一个 View 都可以注册一个 event listener，例如：当 View 要处理用户触碰（touch）的事件时，就要向 Android 框架注册 View.OnClickListener。另外还有 Image 等同于 J2ME 的 BitMap。

如今，我们看到的是一个繁荣而庞大的软件产业。但是依然存在着两大问题：一是编写程序仍然需要很多时间；二是编写出的程序在运行时仍然会出现意料外的行为。而且后一个问题的表现形式越来越多，可能突然报告一个错误，可能给出一个看似正确却并非需要的结果，可能自作聪明地自动执行一大堆无法取消的操作，可能忽略用户的命令，可能长时间没有反应，可能直接崩溃或者永远僵死在那里.....而且总是可能有无法预料的其他意外情况出现。调试时定位程序中的错误并修正其错误的过程，在软件开发环境中，调试器是软件开发必不可少的工具，通用计算机，无论是巨型机，大型机，工作站还是 PC 机，都配置有适合系统特点的调试工具。在嵌入式系统软件开发环境中，调试尤显其重要性，如何“导出”调试信息也是软硬件开发者必须考虑的一个问题。嵌入式系统不能运行一个本地调试器，实现对其中程序的调试只能是通过串口通信，网络通信等方式来完成调试器与运行目标程序的嵌入式设备的交互，这就要求被调试程序中要有实现这种橡胶护的功能模块。

1.2 研究意义

Elastos 操作系统是面向嵌入设备[4]的，和 Wince，Android 等一样都是智能手机的操作系统。智能手机除了具备手机的通话功能外，还具备了 PDA 的大部分功能，特别是个人信息管理以及基于无线数据通信的浏览器，GPS 和电子邮件功能。智能手机为用户提供了足够的屏幕尺寸和带宽，既方便随身携带，又为软件运行和内容服务提供了广阔的舞台。

在 Elastos 中，Component Assembly 包含了这两层含义：（1）软件零件，特指“目标代码单元”。在 CAR 编程规范中就是 DLL，也可以是 JAVA 或 C#中

的目标代码文件；（2）软件部件，是软件零件的集合。一般是个“半成品”，通过 XML 或角本包装成为“产品”，也可以是个“产品”。软件部件不但包含一组 DLLs（也可以是单个 DLL），还包含了数字签名、下载压缩包、元数据信息等打包之后的信息，类似于 JAVA 里面的 JAR 文件、Windows 里面的 CAB 文件等。

Elastos 中 CAR 的含义是“基于 CPU 指令集的软件零部件运行单元”，简单理解就是“软件零部件运行单元”。

Android 手机操作系统的所有应用都是基于 Java 语言的，它的类 Java 虚拟机 Dalvik 提供了所有应用的运行时环境。Dalvik 是一个面向 Linux 作为嵌入式操作系统设计的虚拟机。把 CAR 构件技术引入 Android 计算环境，成为 Android 计算环境的一部分，CAR 的对象具有 Java 对象的同样特性、同样的生命周期及运行上下文，这使得应用程序开发者在开发阶段就可以考虑 Android 程序优化，用 C/C++这样的 native 化程序技术，实现特别的计算要求。当 Java 对象与 CAR 对象混合起来，如何调试这些 CAR 对象呢？

所谓程序调试，是将编制的程序投入实际运行前，用手工或编译程序等方法进行测试，修正语法错误和逻辑错误的过程。传统意义上的调试都是单一对象的，调试技术具有语言单一性，不能跨语言调试。对于复合对象类（Java&CAR 对象）语言中的调试技术，不管是在 PC 上还是移动设备中的调试技术都相当贫瘠。这也就需要产生一种新的调试模型，这种模型能解决先前调试技术无法跨语言调试的缺陷，能在复合对象中兼顾两种语言对象的调试，同时能解决软件开发新的发展过程中产生的新阶段的瓶颈。

本文提出一种快照与 C/C++调试相结合的调试技术，对 CAR 运行环境，运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议把信息传递给调试器，在进入 CAR 构件前插入单步中断，从而让 C/C++调试器得到介入的机会，实现 CAR 构件的二进制调试。

1.3 本文所做的工作

本文首先介绍了 Java&CAR 程序设计技术，然后详细介绍了 CAR 构件快照技术，最后通过扩充 JAVA 调试技术的 JPDA 的实现，支持 CAR 快照的传输，并介入 C/C++调试器，实现复合对象的调试。

Java&CAR 调试技术是指在调试器（如 Eclipse）端，对 CAR 构件运行数据环境、线程、回调事件等进行监测、动态调整的技术。它基于 JDWP 协议，通过扩展 JDWP 在 Dalvik 中的调试事件的实现，达到调试 CAR 构件的目的。在 Java&CAR 的视角下，CAR 构件不仅是二进制的 C/C++程序，更是具有自己独

立计算特性的计算体。

可以监测与管理的 CAR 运行状态信息有：

- CARed java object，这些实例的内存、宿主 Java 类名等。
- Dalvik&CAR 中，实现回调业务的 IApplet 信息。其中的事件队列中的事件情况。
- CAR 构件运行时的数据环境，指 CAR 加载模块（module）的情况、所占内存情况、singleton 实例创建情况等。

Java&CAR 调试技术并不在机器语言级调试 CAR 构件，只是快照（snapshot）式对 CAR 构件及其运行实例进行监测。通过这些监测结果，有助于对系统架构及程序实现进行优化。

1.4 论文结构

本文从复合对象的概念出发，结合 3G 时代手机应用的发展背景，阐述了复合对象调试模型对手机应用的发展所带来的意义；描述了复合对象的调试相比传统应用所具有的特点和优势；紧接着介绍了 Elastos 操作系统和该操作系统上的 CAR 构件技术；接着介绍了基于 JAVA 的调试技术 JDWP，以及在该技术下，实现的 CAR 构件的快照技术等；最后结合上述背景条件，提出了一种基于 JAVA&CAR 复合对象调试模型，阐述了在 JPDA 整个框架模型中的扩充以便支持 CAR 快照的传输，最后完成了快照（snapshot）式对 CAR 构件及其运行实例进行监测的全面实现。

第一章：绪论

介绍了手机应用背景，复合对象调试模型建立的意义和本文所做的工作。

第二章：Java&CAR 程序设计技术

介绍了 Elastos 操作系统和 CAR 构件技术，Dalvik 在 CAR 构件运行时中的应用研究，以及 Java&CAR 的 CAR 构件的详细介绍。

第三章：JPDA 的介绍

介绍了 Java 调试体系 JPDA，并详细介绍了 JDWP 的工作原理，包括协议分析，包结构，实现和处理机制。

第四章：CAR 构件快照

介绍了复合对象，方法信息，线程快照和几个特别对象的快照。

第五章：JPDA 的实现和 C/C++调试器的介入

详细介绍了在 JPDA 调试体系中的具体扩充，包括在 debugger.c 和 jdwphandler.c 的具体实现，以便实现 CAR 构件快照的传输。对 CAR 运行环境，

运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议 JDWP 把调试信息传递给调试器（如 Eclipse），实现 JAVA 语言级调试。

第六章：部分实现与应用

本章介绍的是在实际应用中的具体实现情况，就具体在开发环境中如何实现，首先介绍的是 android 中 Dalvik 的具体运作，接着介绍 Dalvik 如何运行 Java&CAR 的复合对象，接着在 Eclipse 的 debugger 环境下，查看如何调试 Java&CAR 复合对象，如何获得 CAR 构件的快照。

第七章：总结

总结本论文的研究工作，并对以后的研究工作提出了展望。

第 2 章 Java&CAR 程序设计技术

2.1 Elastos 操作系统概述

2.1.1 Elastos 操作系统

“和欣”操作系统（英文名称为“Elastos”）[4]是构件化的网络嵌入式操作系统，具有多进程、多线程、抢占式、多优先级任务调度等特性。目前，Elastos 已经可以在包括 PC、ARM、MIPS 等多种体系架构上运行。Elastos 提供的功能模块全部基于 CAR（Component Assembly Runtime）构件技术，这是 Elastos 操作系统的精髓。CAR 构件技术规定了构件间相互调用的标准，每个 CAR 构件都包含自描述信息，可以在运行时动态裁剪组装。CAR 构件技术贯穿于整个 Elastos 操作系统技术体系中。

从传统的操作系统体系结构的角度来看，和欣操作系统可以看成是由微内核、构件支持模块、系统服务器组成的。

微内核：主要可分为 4 大部分：硬件抽象层（对硬件的抽象描述，为该层之上的软件模块提供统一的接口）；内存管理（规范化的内存管理接口，虚拟内存管理）；任务管理（进程管理的基本支持，支持多进程，多线程）；进程间通信（实现进程间通信的机制，是构件技术的基础设施）。

构件支持模块：提供了对 CAR 构件的支持，实现了构件运行环境。构件支持模块并不是独立于微内核单独存在的，微内核中的进程间通讯部分为其提供了必要的支持功能。

系统服务器：在微内核体系结构的操作系统中，文件系统、设备驱动、网络支持等系统服务是由系统服务器提供的。在和欣操作系统中，系统服务器都是以动态链接库的形式存在。

同时，和欣操作系统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。

和欣操作系统的最大特点也是它最大的优势：

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。

在新一代互联网应用中，越来越多的嵌入式产品需要支持 Web 服务，而 Web

服务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。这是新一代互联网应用的需要，是必然的发展方向。和欣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服务的技术。

实现 Web 服务的关键技术之一是面向构件、中间件的编程技术。Web 服务提供的软件服务就是可执行的功能模块，就是构件。构件是包含了对其功能的自描述信息的程序模块。和欣操作系统支持在网络环境下动态查找、动态链接构件，为 Web 服务提供了支撑平台。

和欣操作系统可广泛应用于信息家电、工业控制、传统工业改造、国防、商业电子等领域，已经开发了 PDA/掌上电脑、数控机床、工业远程监控设备、医疗仪器等应用。

2.1.2 CAR构件技术

Elastos 提供的基本系统服务包括：基本 CAR 函数，与 CAR 构件相关的调用；命名服务，提供注册命名服务功能；内存管理，虚拟内存的管理功能；进程，进程创建、退出和获取；线程，线程的操作；线程局部存储，在线程范围内保存某些变量和值；同步，与同步相关的变量处理和方法；其它系统服务，如获取系统统计信息等功能；错误返回机制，设置或获取错误报告信息；可执行文件和模块，获取或装载当前进程的映像模块；内存共享，创建共享内存；堆内存，在堆上分配或释放内存等；CAR 反射函数，在构件对象中获取模块信息。

2.1.3 Elastos的回调机制

在 CAR 构件技术中，回调机制实现了用户程序与构件间的双向函数调用，即不仅用户程序能通过接口调用构件提供的服务，构件也能通过回调接口向外抛出事件，调用用户程序实现的函数

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。

服务器端与客户端之间的关系可以是一对多，也可以是多对一。即一个服务器端可以向多个客户端抛出相同或不同的回调事件；而一个客户端也可接收来自不同服务器端抛出的多个回调事件。

CAR 的回调具有广播与异步的特征。广播意味著在服务器端抛出的回调事件可被多个客户端响应；异步一方面意味着服务器端在抛出了回调事件后不用等待客户端的回复，而可以继续其它的操作；另一方面，客户端在接受到回调事件通知后不用立刻去执行这个回调事件，而是将其放入自己的消息队列，等待执行。

CAR 事件回调机制适用于并发运行的环境，其预期解决的问题是：当服务器端和客户端之间的依赖关系比较弱，服务器端只想广播一下自己的状态信息，而客户端也只要知道服务器端发生过什么，彼此间的通信没有实时性的要求。

例如，如果两个人通过邮政系统通信或者打电话，就不太适合用我们的事件回调机制来实现。邮政信件是两个对象之间的点对点通信，但 CAR 的回调是可以广播的，可以被 N 个人注册和接收到，隐私全曝光了。如果是打电话的话，因为回调不具有实时性，对方隔个十分八分再回一句话，你一定难以忍受。

再以进度条更新的应用为例，下载一个文件时要显示下载进度，下载是由一个底层构件实现的，显示是由一个 UI 程序实现的，一般做法是：底层构件每次收到数据就发事件给 UI，UI 计算百分比更新显示。但可能 UI 处理回调的线程很繁忙，可能底层已经下载了 50%但 UI 才显示了 10%，或者由于更新进度条的消息太多，当用户想 cancel 的时候，这个 cancel 事件被排在后面，当执行 cancel 时，已经下载完了等等。

面对这种应用，首先要考虑 UI 设计要求是什么？实时显示真实下载状态？是否要显示整个下载过程？是否能接受进度条不准确或不及时？

如果要求实时性，要求进度条绝对反映真实状态，那绝对不应该用 CAR 的事件回调机制，要想办法直接同步调用。

如果可以接受进度条不准确，只是给用户一个心理安慰，那完全可以使用我们的回调机制，并且合并更新进度条的事件避免大量垃圾事件堵塞消息队列，这样用户看到的进度条可能会跳跃着变化，也可能由于下载太快直接就是 100%。还可以提高 cancel 事件的优先级，使其尽快被执行来中止下载。

整个 Callback 过程的实现可以分成三个部分：Server，Sink Object，Client。

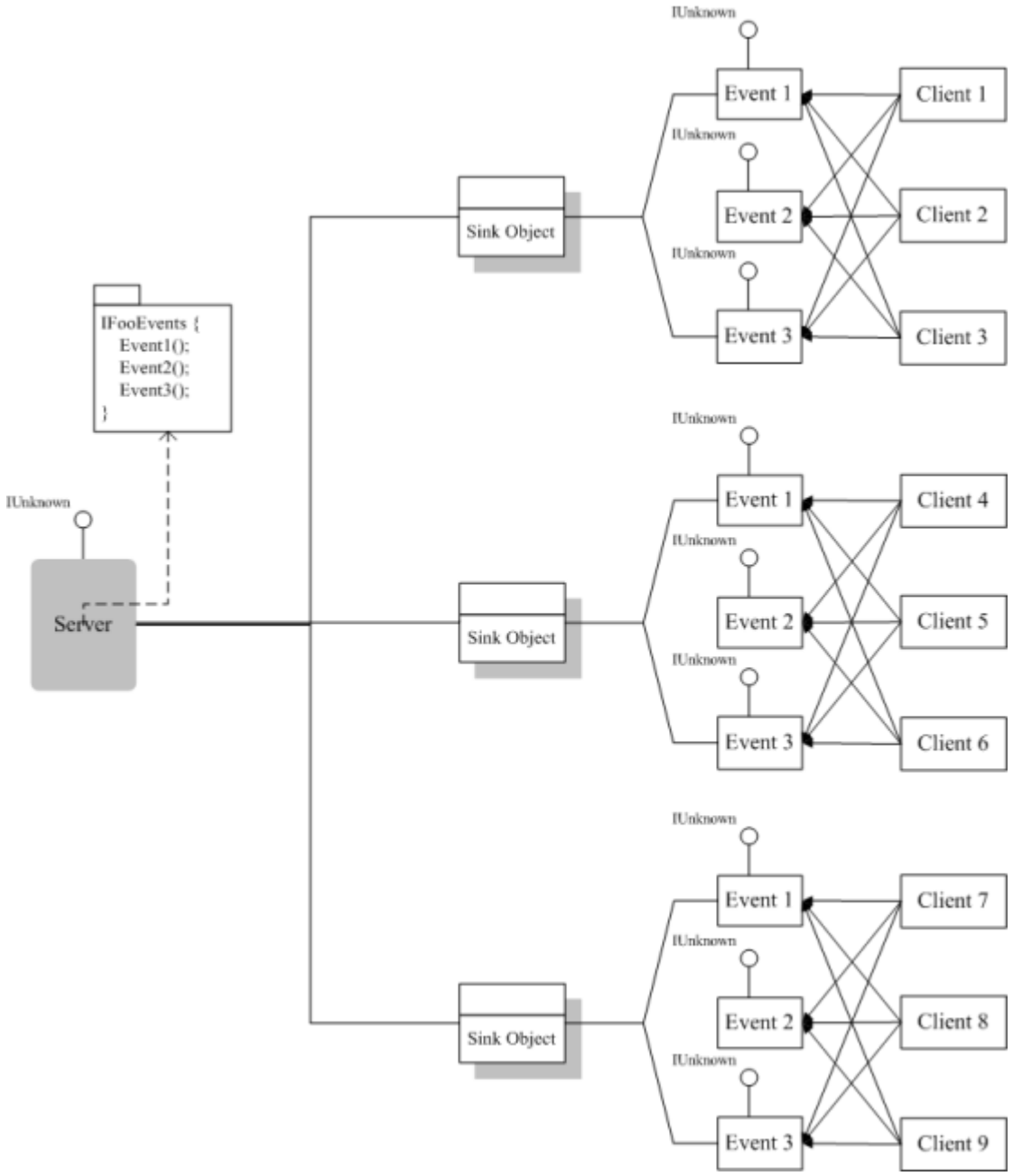

图 2.1　callback 机制的结构

Server 端负责激发事件；Client 负责处理事件；那么 SinkObject 呢？负责关联 Server 与 Client。Server 可能会激发很多事件，但不是每个事件都是同一个 client 关心的；同样,Client 也可能同时关心 Server 的不同事件。事件与 Client 是多对多的关系，一个 Client 可以关心多个事件，一个事件也可以发送给多个 Client，至于哪些 Client 关心哪些事件，这个关联关系就是通过 SinkObject 来记录和管理。

下面我们来看一下注册回调和激发事件的时序图

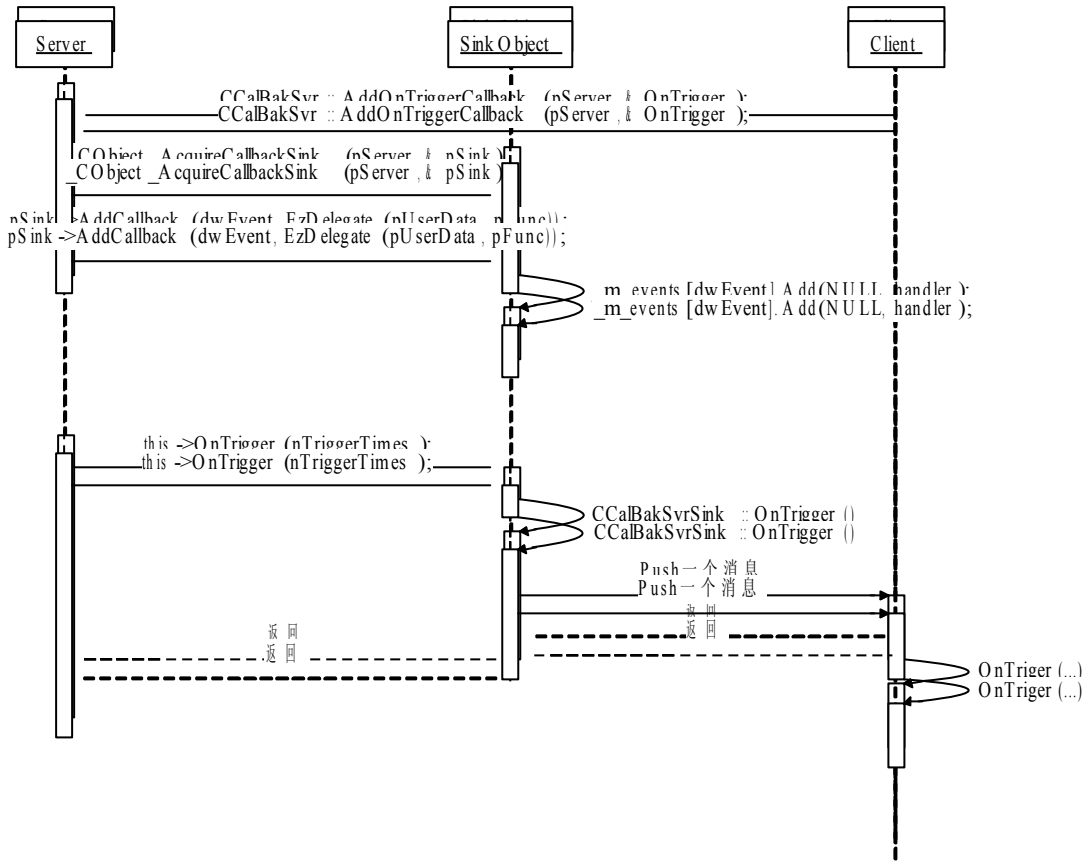

图 2.2 一次回调过程的时序图

（1）注册回调

AddXXXCallback 是“万恶之源”，它会传递四要素信息（Server 对象、事件、Client、回调函数）给目标 Server，Server 端则 Acquire 一个 Sink Object，然后通过其接口方法增加上一条注册信息，记录在 Sink Object 的数组_m_events 里。

（2）激发事件

当 Server 端激发一个事件时，会调用 Sink Object 对应的函数方法，再由 Sink Object 查找注册信息的数组，找到对应的 Client，向其 push 一条消息，然后立即返回，不会等待 Client，这条消息同样包含着四要素信息（Server 对象、事件、Client、回调函数），Client 收到以后解析消息，调用回调函数

ECode OnTrigger(PVoid pUserData, ICalBakSvr *pSender, int nTriggerTimes)

其中 pSender 即为 Server 对象句柄，表明是谁激发的事件。

注意：这里要特别提醒一下，Sink Object 发送一条消息给 Client 后就立即返回，不会等待 Client 回调函数完成，不保证 Client 调用 OnTriger 时 Server 的状

态。之所以如此是避免由于 Client 代码的问题影响 Server 端，如果要等 Client 处理完再返回，倘若 OnTriger 里面有一个死循环，Server 的代码就永远也执行不下去了。Server 和 Client 很可能是完全不同的两个厂商。所以 Server 不能做这种假设和信任。我们即将支持内核事件的注册回调，如果由于用户代码问题而阻塞内核进程，这更是不能容许的。

2.1.4 Elastos 可执行文件规范

在 Elastos 中，可执行文件的格式可以是.exe 或者具有主函数的.dll。对于.exe 文件，主函数的形式可能是以下三种：

1）ECode __cdecl ElastosMain(const WStringArray& args);

ElastosMain 的返回值即为进程退出码。

示例 1：

```
#include <elastos.h>

ECode ElastosMain(const WStringArray& args)
{
    CConsole::WriteLine(L”Hello World!”);

    return NOERROR;
}
```

2）int __cdecl _wmain(int argc, wchar_t * wargv[]);

示例 2：

```
#include <stdio.h>

int wmain( int argc, wchar_t *argv[])
{
wprintf(L"Hello World!\n");
return 0;
}
```

3）int __cdecl main(int argc, char * argv[]);

示例 3：

```
#include <stdio.h>

int main( int argc, char *argv[])
```

```
{
    printf("Hello World!\n");
    return 0;
}
```

其中 ElastosMain()是 Elastos 平台默认首选的可执行文件的主函数，_wmain()和 main()是 POSIX 标准兼容的主函数。

具有 main 属性的 CAR 构件可以被独立的加载运行，如：

```
[main] class CFooBar {...}
```

对应会生成包含

```
ECode CFooBar::Main(const WStringArray& args);
```

的 C++代码。这个函数和 ElastosMain 的原型一致，行为也一致。参数表存在一些差异，第一个参数为该 DLL 的 PATH，而非宿主进程的可执行文件 PATH。

可以使用 SuperExe 来执行 CAR 构件 dll，必须有带 main 属性的 class 的 dll 才能被 SuperExe 执行。其他 CAR 构件只能由其他进程以 dll 的形式调用。

示例：4

CFooBar car

```
module CFooBar.dll
{
    [main]
    class CFooBar {
    }
}
```

CFooBar.cpp

```
ECode CFooBar::Main(const WStringArray& args)
{
WStringBuf_<20> wstrBuf;
wstrBuf.Copy(L"HelloWorld!\n");
    CConsole::WriteLine(wstrBuf);
    return NOERROR;
}
```

2.1.5 多任务操作系统

在计算机中，多任务指的是多个程序共享处理器等计算资源，在单 CPU 的计算机上，某个时刻，只有一个任务正在执行，即 CPU 正在这个任务的指令。

多任务要解决的问题就是如何调度多个任务，指明什么时候该执行哪个任务，什么时候该让下一个正在等待的任务执行。把 CPU 从一个任务指派到另一个任务的动作叫做上下文切换，当上下文切换频率很高的时候，多个任务看起来就像是在并行执行。即使是在拥有多个处理器的计算机上，多任务机制使得并行的任务数目可以大于处理器个数。

不同的操作系统采用不同的任务调度策略，这些操作系统可以分为这么几类：

多道程序（Multiprogramming）系统，在这中系统中，一个任务一直执行，直到主动或者被动的放弃 CPU，主动的情况有等待一个外部事件，比如从磁带读取数据。这种机制是为了提高 CPU 的使用效率。

分时系统(time-sharing)，这种系统的设计目标是让多个用户同时交互式的使用同一个系统，它和多道程序系统相似的一点是强调多个任务并行执行，不同点是它更强调用户和系统之间的交互性，这种系统通常配备多个终端，用户通过终端和计算机进行交互。

实时(real-time)系统，这种系统强调系统对外部事件的及时反应。当一个外部事件发生的时候，系统保证某个任务在一定时间内获得使用 CPU 的权力，实时系统通常被用来控制工业机器人等实时性要求较高的领域。

分时这个词已经不太常用了，它正在被多任务这个词取代，因为多个用户同时地交互式的使用一个计算机的情况越来越少了，取而代之的是个人计算机和工作站。

多道程序

早期，CPU 时间比较昂贵，而外部部件的时间相对便宜，当一个程序访问外部硬件的，然后外部硬件还在处理数据的时候，CPU 就停止执行程序指令了，这种方式很浪费 CPU 时间，效率低下。

为了解决这个问题，上世纪 60 年代，人们提出了多道程序的思想。多个程序被同时加载到内存，第一个程序先执行，当它访问速度相对慢的外部设备的时候，这个程序被暂停，状态被保存下来，第二个程序开始执行。当所有程序执行完毕，整个计算过程结束。多道程序系统不保证一个程序会及时执行，如果某个程序不访问外设，并且执行几个小时，那么其他程序就没有机会执行，由于当时没有多个用户在终端等待结果，这也没有什么问题。用户们将一摞纸带交给管理员，然后几个小时后再回来取结果。多道程序系统减少了用户总的等待时间，提高了计算吞吐量。

协作式多任务/分时系统

后来计算机从批处理模式进化到交互模式，这个时候多道程序系统就不再适

用了，每个用户都希望自己的程序（看起来像是）独占计算机，分时技术使得这种愿望得以实现，当然了，由于计算机在并行的处理多个用户的请求，所以速度上比起单独处理一个用户的请求要慢一些。

在早期的多任务系统上，多个应用在设计的时候就考虑到了相互协作，应用程序在运行的过程中会主动的将 CPU 让给其他应用，这种机制就是所谓的协作式多任务。虽然现在不常见了，不过这种机制当时被微软和苹果公司的操作系统采用，用来支持多应用的并行执行。

由于协作式多任务系统依赖于程序主动的让出 CPU，如果某个程序设计设计有问题，一直不让出 CPU，则其他程序就没机会执行，系统就将瘫痪。所以每个程序在安装到服务器之前要经过仔细检查，否则当某个服务器端的程序行为异常的时候，整个系统将变慢或者干脆瘫痪。

抢占式的多任务/分时系统

抢占式的多任务系统通常能保证每个程序都有几乎执行，因为操作系统可以强制一个程序放弃 CPU。这种系统还能快速的处理外部事件，比如数据输入，这种事件可能需要及时的引起某个进程的注意。

这种系统充分利用了计算机硬件的能力，抢占式的处理多个进程。1969 年，抢占式的多任务就在 Unix 上实现了，现在它已经成了 Unix 和类 Unix 系统（包括 Linux， Soloris， BSD 和其他衍生版本）的标准。

在任意时刻，进程可能处于一下两个状态之一：正在等待外部输入或者输出；这在使用 CPU。在早期的系统之中，程序需要通过轮询查询外部请求，这个时候，系统做不了其他事情，后来有了中断机制和抢占式调度，正在请求 I/O 的进程将被阻塞，系统让其他进程先执行，当外部数据到达的时候将触发一个中断，被阻塞的进程进入就绪状态，等到下一次调度的时候，这个进程就有机会执行了。

实时系统

设计多任务的另一个目的是为了在单处理器的计算机上实现实时系统，一个处理器要及时处理多个不相关的外部事件。实时系统采用了分层次的中断机制，并采用了进程优先级，保证重要的任务获得较多的处理器时间。

多线程

多任务显著的提高了计算机的吞吐量，程序员可以用一组相互协作的进程实现自己的应用程序（比如一个进程接受输入，一个处理数据，一个将结果保存到硬盘），于是进程之间的数据交互问题就凸显了出来。

于是人们提出了线程，进程之间数据交互的最有效办法就是共享地址空间，进程对于程序之间的隔离过于极端，线程算是对这种极端的适度纠正。线程可以被视为运行在同一个地址空间的进程，由于线程切换的时候不涉及地址空间的切

换，所以线程也被称为轻进程。

线程之间的调度是抢占式的，有的操作系统提供了一种叫纤程的机制，纤程采用的是协作式调度。在没有提供纤程的系统上，应用程序通过反复调用工作方法（worker functions）实现自己的纤程。纤程比线程更轻，也更容易编程，但是在多处理器的计算机上，线程比纤程更有优势。

2.2 Dalvik 在 CAR 构件运行时中的应用研究

2.2.1 支持 CAR 构件的 Dalvik 虚拟机

2.2.1.1 Dalvik 介绍

Android 是 Google 于 2007 年推出的一个开放源码的手机操作系统。它基于 Linux 操作系统，并为了使其更适应嵌入式设备的使用环境，作了许多改进和创新。Android 的上层应用使用的是 Java 语言，所以 Java 虚拟机是 Android 操作系统承上启下的基础。因此，Google 精心设计了自己的 Java 虚拟机——Dalvik。

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，不提供缓存机制以及电池供电等特点。不同于传统的 Java 虚拟机，Dalvik 必须通过一些创新来适应这样的运行环境。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

从 Dalvik 的特性来说，它尽可能的降低功耗，减少内存来适应 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。

2.2.1.2 虚拟机的 CAR 扩展

把类（Class）实例化为实例(Object)，并对 Object 进行生命周期管理、组装、运行上下文（Frame）管理，对字节码表示的方法进行解释执行，这是 Java 虚拟机的基本功能。CAR 构件也有类与实例的概念，它的实例就是一个包装了的 C++对象，运行环境就是一个普通 C/C++程序的上下文环境。把两者结合，让 CAR 起到 JAVA 领域中的 AOT（Ahead Of Translation）的作用，结合两个对象系统的各自长处，满足移动设备的计算要求。Dalvik 是一个开源 Java 虚拟机，它支持了 Android 操作系统移动设备的所有应用的运行。通过对 Dalvik 的改造，使得 Android 支持用 CAR 技术来写 Java 类，在保持接口语义的完整性的同时，提高了程序执行效率、减小了内存使用，实现了对 Android 的深度定制。

Java 是跨平台语言，可以做到“一次编写，随处运行”。它这种特性依赖的是 Java 虚拟机（JVM）的平台相关性，即 Java 虚拟机的实现满足了 Java 虚拟机规范，但是都是依赖具体平台的。由于 Java 程序需要 Java 虚拟机来解释执行，在性能上必定不如可以编译执行的本地代码。CAR 是一种依附于 Elastos 操作系统平台的二进制组件，具有自描述性，二进制继承等特性。本文结合 CAR 和 Java 虚拟机技术，提出了一种 Java 和 CAR 混合编程的技术，旨在利用 CAR 构件提高 Java 程序的运行速度而不破坏 Java 的内部结构，所依赖的虚拟机平台是 Android 上的 Dalvik 虚拟机。

2.2.1.3 虚拟机的 CAR 扩展互操作与 JNI 的比较

JNI（Java Native Interface）是 Java 的本地编程界面规范，它允许在虚拟机里运行的 Java 代码可以和 C，C++或汇编代码编写的程序或库进行互操作。

当使用 Java 编程时，程序中会产生一些调用本地代码的需求，例如出现以下的情况时，就必须使用 JNI 来调用本地代码。

- Java的标准库并不支持某个平台的特性。
- 已经有一个库是用本地代码编写的，同时程序又想调用这个库。
- 在一些地方，程序对执行时间有严格的要求，而Java语言并不能满足这一需求。

使用 JNI 技术，可以很好的解决 Java 语言和 C 语言之间的调用问题。但是这种技术只能调用 C 语言中的简单函数，同时也丧失了 Java 语言的跨平台性。从编程角度来说，数据类型的转换需要程序员自己来掌握，繁琐且容易出错。

本文描述的对虚拟机的CAR 扩展技术是基于JNI 技术之上实现的一个技术。

CAR 构件扩展技术保证了 CAR 语义的完整性而提高了 Java 程序的运行速度，同时由于 CAR 构件是运行于 Elastos 平台上的，这样对利用 CAR 扩展实现的 Java 程序的可移植性基本没有影响。另外 CAR 构件扩展技术在虚拟机内部实现了 Java 和 CAR 数据类型的转换，而不需要程序员来考虑，程序员使用 Java 调用 CAR 构件就好像调用一个普通的 Java 类一样简单。

2.2.2 Java 和 CAR 构件相互操作的设计和实现

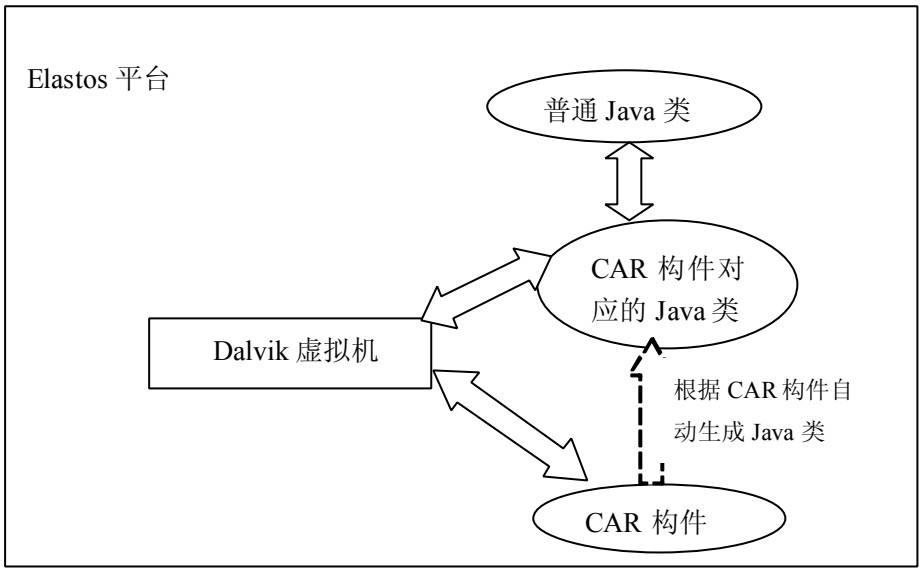

图 2.3 Java 和 CAR 构件互操作模型

如图 2.3 所示是 Java 和 CAR 构件的互操作模型。Java 和 CAR 构件互操作时，需要根据 CAR 构件生成一个 Java 类。因为对于每一个 CAR 构件来说，都具有一个描述自身信息的接口文件，这个文件描述了相应的 CAR 构件具有的可供调用的类名称和函数原型。可以根据这个接口文件生成具有同样类名称和函数原型的 Java 类，在这个类中函数都被声明为 native 类型的，同时在类的头部利用 Java 的 Annotation 属性来声明这些类是 CAR 的对应 Java 类。这样在加载这个 Java 类时，虚拟机可以知道这是一个 CAR 构件对应的 Java 类。虚拟机对这个类除了像普通 Java 类一样处理，同时还加载相应的 CAR 构件，查询并存储相应 CAR 中类的类信息。在这里加载 CAR 构件的机制依赖于 CAR 的运行环境，即 Elastos 平台。在普通 Java 类调用 CAR 对应 Java 类时，生成相应的 CAR 对象，而在调用这个类中的函数时系统会去调用 CAR 构件中的函数。这样就实现了 Java 和 CAR 构件的互操作。

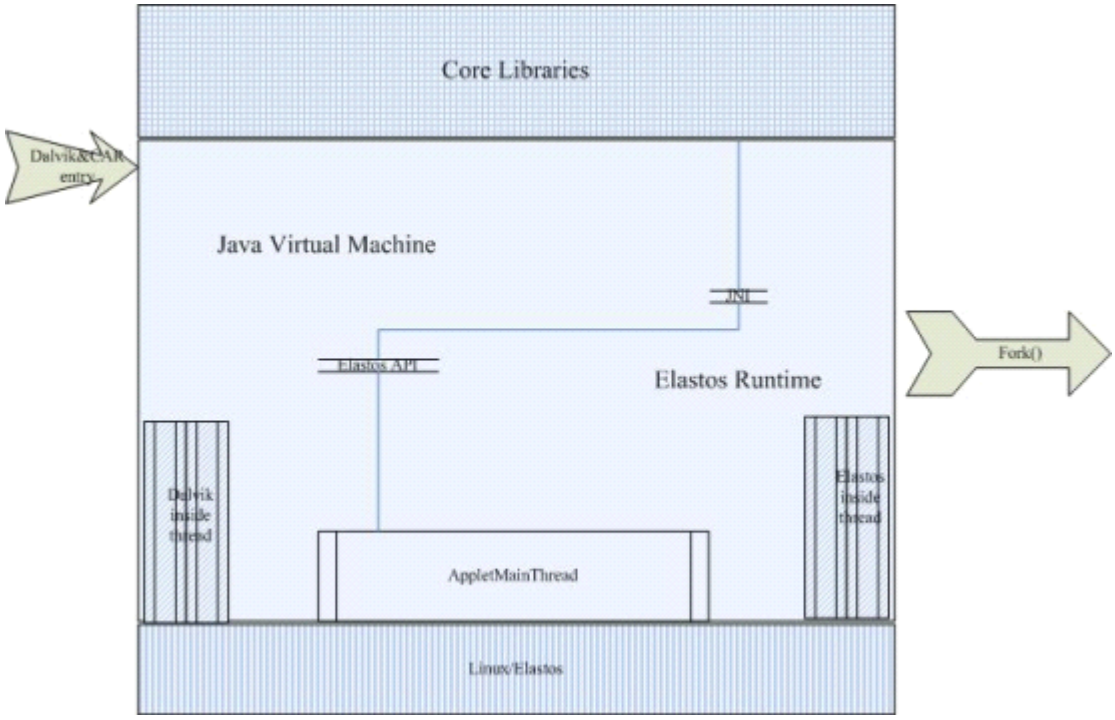

图 2.4 Dalvik&CAR 虚拟机架构示意图

图 2.4 显示了 Dalvik 虚拟机与 Elastos 运行时的架构，底层的操作系统分别为 Linux 与 Elastos，同时 JVM 和 Elastos 都保留自己的内部线程，Elastos Runtime 与 JVM 通过 JNI 和 Elastos API 进行互操作。

2.2.2.1 应用开发支持

由于我们的目标是要普通的 Java 类可以调用 CAR 构件，并且不能破坏 Java 的语法。也就是是说当在普通 Java 类中新建一个 CAR 构件对象就好像新建一般的 Java 对象一样，调用 CAR 构件中的函数就好像调用一般 Java 对象中的函数一样。要达到这个目的我们必须根据 CAR 构件生成一个 Java 类，这个类具有和 CAR 构件同样的语义。如下所示，我们根据 CAR 构件的接口文件生成相应的 Java 类文件（这个可以自己做的 CAR2Java 工具来自动化完成）：

例如对于有一个实现两个整数相加的 CAR 构件方法来说，它对应的 Java 类是这样的：

```
package com.elastos.runtime;
import    dalvik.annotation.CAR;
@CAR(ElastosModule="Calculate.dll", ElastosClass="CCalculate")
class Calculate{
```

```
    native int add(int a, int b);
}
```

在这里我们自定义了一个 Java 的 annotation 类@CAR 来标识这个 Java 类是一个 CAR 构件对应的类。这个 annotation 类具有两个属性 ElastosModule 和 ElastosClass，分别指定 CAR 构件的名称和 CAR 类的名称。我们在根据 CAR 构件构造 Java 类时，使用@CAR 来修饰 Java 类，并指明其对应的 CAR 构件的名称和 CAR 类的名称。有了这个属性，在虚拟机加载之时就可以知道这个类对应的 CAR 构件和类了。

2.2.2.2 对 Dalvik 进行 CAR 扩展

(1)利用反射机制得到 CAR 构件的类信息和方法信息

每当虚拟机加载一个标有@CAR 属性的 java 类，我们就知道这是一个 CAR 构件对应的 Java 类。我们在保存这个类信息的结构体中设置一个标识标明它是 CAR 构件对应的 Java 类。同时利用 Elastos 中 CAR 构件的反射机制加载对应的 CAR 构件，并得到 CAR 构件中的类信息和方法信息。

具体过程是根据 ElastosModule 属性得到 CAR 构件的名称，根据名称加载 CAR 构件，并得到它的构件信息 ImoduleInfo。然后根据 ElastosClass 属性得到类的名称，从 IModeInfo 中反射得到其类信息 IClassInfo，将其保存在 Java 类信息的结构体中。

而在加载 Java 类的本地函数时，如果发现所属的类是 CAR 构件对应的 Java 类，我们将反射 CAR 的函数信息。函数信息 IMethodInfo 的反射过程和类信息的反射过程类似，通过加载类时保存的类信息 IClassInfo，并结合函数名字，可以反射得到 CAR 构件中函数信息 IMethodInfo。设置好 CAR 函数信息指针后，我们得到 CAR 函数的地址指针，将其保存在 Java 方法信息的结构中，并设置其结构中的 nativeFunc 成员为的 dvmCallCARJNIMethod。

(2)CAR 对象的生命周期

当创建一个 CAR 构件对应的 Java 类对象时，我们也创建一个 CAR 构件的对象。新创建的 CAR 对象的生命周期和对应的 Java 对象的生命周期是相同的。即它在 Java 对象创建时被创建，在 Java 对象销毁时被销毁，并且在 Java 对象被克隆时，也对应克隆出 CAR 对象。

对于具有 CAR 构件的 Java 类对象，我们为其多分配 sizeof(u4)的空间来存储新建 CAR 构件的指针,这样新建的 CAR 构件对象就依附在 Java 对象之上了。当要使用这个 CAR 对象时，只要取出 Java 对象之后的 4 字节的内容，强制转换

成对应的指针即可。

(3)Java 调用 CAR 构件

通过前面的工作我们设置 dvmCallCARJNIMethod 为方法结构中的 naticeFunc, dvmCallCARJNIMethod 是我们自己调用 CAR 构件方法的桥接方法，每当调用 CAR 构件对应的 Java 类中的方法时都会触发这个方法。在这个方法中我们实现调用相应 CAR 构件对象上的相应方法，并实现了 Java 类型和 CAR 构件类型的相互转换。

Java 参数类型和 CAR 参数类型对应关系表如表 2.1 所示：

表 2.1　Java 类型和对应的 CAR 类型

Java类型	CAR类型	描述
boolean	Boolean	8位整数
byte	Byte	8位无符号整数
char	Char	字符类型
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
float	Float	32位IEEE浮点数
double	Double	64位IEEE浮点数
string	String	字符串对象

2.3　Java&CAR 中的 CAR 构件

2.3.1　Java Class 中内建 CAR Class

由于我们的目标是要普通的 Java 类可以调用 CAR 构件，并且不能破坏 Java 的语法。也就是说当在普通 Java 类中新建一个 CAR 构件对象就好像新建一般的 Java 对象一样，调用 CAR 构件中的函数就好像调用一般 Java 对象中的函数一样。要达到这个目的我们必须根据 CAR 构件生成一个 Java 类，这个类具有和 CAR 构件同样的语义。

例如对于有一个实现两个整数相加的 CAR 构件方法来说，它的接口文件表下所示：

```
//Calculate.CAR
module
{
```

```
    interface IAdd
    {
        add([in] Int32 i, [in] Int32 j, [out] Int32* o);
    }
    class CCalculate
    {
        interface IAdd;
    }
}
```

我们根据 CAR 构件的接口文件生成相应的 Java 类文件（这个过程可以使用 CAR2Java 工具来自动完成）：对应的 Java 类文件如下所示：

```
package com.elastos.runtime;
import    dalvik.annotation.CAR;
@CAR(ElastosModule="Calculate.dll",                ElastosClass="CCalculate"                ,
CARSingletonClass="staticCCalculate")
class Calculate{
    native int  add(int a, int b);
    static native int setMark(int k);
}
```

在这里我们通过一个 Java 的 Annotation 类@CAR 来标识这个 Java 类内置有一个 CAR 构件类。Annotation 类@CAR 定义如下所示：

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

这个注释类具有两个属性 ElastosModule 和 ElastosClass，分别指定 CAR 构件的名称和 CAR 类的名称。我们在根据 CAR 构件构造 Java 类时，使用@CAR 来修饰 Java 类，并指明其对应的 CAR 构件的名称和 CAR 类的名称。有了这个

属性，在虚拟机加载时就可以知道这个类对应的 CAR 构件和类了。

如果一个接口的名字是“Ijava”开头，表示这个接口中定义的方法，是用来指示 java 的当前类中的同名方法的 CAR 的元数据，这些同名方法一定不是 native 的，如果是 native 的，则这个规则不起作用。

示例如下：

```
interface IjavaBuilding {
    CatchFire([in] Int32 i, [in] Int32 j, [out] Int32 *o);
    /*  具体写 c 程序的封装函数时，可以这样写：
      JValue j;
      static jmethodID methodID = 0;
      if (methodID == 0)
          methodID = GetMethodID(env, class, "CatchFire", NULL);
      o = env->CallNonVirtualIntMethodV(jniEnv, jniObj, /*clazz*/ *(int *)obj,
methodID, i, j);
     */
}
```

上例中，CatchFire 方法的 Java 方法在被后面的例程调用时，将知道它的参数等元信息，通过这些元信息，Java&CAR 把参数中，需要转换的，做自动转换。如 CAR 中的 ArrayOf、BufferOf、enum 等，转为 Java 对应的类的对实例。

只有有 Java 的 CAR 包装信息的 Java 函数，在下列例程中，Java&CAR 才会自动做参数转换：

Call<type>Method 例程

Call<type>MethodA 例程

Call<type>MethodV 例程

CallNonvirtual<type>Method 例程

CallNonvirtual<type>MethodA 例程

CallNonvirtual<type>MethodV 例程

2.3.2 Java Class 中内建 CAR Interface

CAR 的接口 Interface 是附属在 CAR 类（Class）的一个概念，它是对一个对象实例的可操作的功能的集合。CAR 的接口，全部是方法（Method、API）组合的，没有属性（Property）、事件（Event）等说法。

我们通过一个 Java 的 Annotation 类@CAR(ElastosInterface)来标识这个 Java 类内置有一个 CAR 构件接口。Annotation 类@CAR 定义如下所示：

```
package dalvik.annotation;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ElastosInterface {
    String ElastosModule();
    String ElastosInterface();
}
```

这个注释类具有两个属性 ElastosModule 和 ElastosClass，分别指定 CAR 构件的名称和 CAR 类的名称。我们在根据 CAR 构件构造 Java 类时，使用@CAR 来修饰 Java 类，并指明其对应的 CAR 构件的名称和 CAR 类的名称。有了这个属性，在虚拟机加载时就可以知道这个类对应的 CAR 构件和类了。

注意：

```
@CAR(ElastosModule="Calculate.dll", ElastosClass="CCalculate")
@CAR(ElastosModule="Calculate.dll", ElastosInterface="ICalculate")
```

上面这种表述中，ElastosClass 与 ElastosInterface 的识别顺序是 ElastosClass 优先，而且每个类上附属的这个定义只能有一个。

一个 Java 类继承链上，只能有一个 ElastosInterface 定义。

$\908-CARTestInterface\src\CPV.java

```
import dalvik.annotation.ElastosClass;
@ElastosClass(ElastosModule="PV.dll", ElastosClass="CPV")
public class CPV{
    public native IV getIV();
    public native void IPPrint();
    public native IP getIP();
    public native void IVPrint();
}
```

$\908-CARTestInterface\src\IP.java

```
import dalvik.annotation.ElastosInterface;
@ElastosInterface(ElastosModule="PV.dll", ElastosInterface="IP")
public class IP {
    public native IV getIV();
    public native void   IPPrint();
```

```
}
```

$\908-CARTestInterface\src\IV.java

```
import dalvik.annotation.ElastosInterface;


@ElastosInterface(ElastosModule="PV.dll", ElastosInterface="IV")
public class IV {
	public native IP getIP();
	public native void   IVPrint();
}
```

$\908-CARTestInterface\src\Main.java

```
import dalvik.CAR.CARCallbackFunc;
class Main{
	public static void main(String[] args)
	{
		System.out.printf("test start here \n");
		CPV pv = new CPV();
		pv.IPPrint();
		pv.IVPrint();
		IP m_IP = pv.getIP();
		//pv.getIP();
		System.out.printf("test getIP \n");
//		IV m_IV = pv.getIV();
		m_IP.IPPrint();
//		m_IV.IVPrint();
		System.out.println("test end here");
	}
}
```

PV.CAR

```
module
{
	interface IP;
	interface IV;
	interface IP {
		getIV([out] IV** pIV);
```

```
        IPPrint();
    }
    interface IV {
        getIP([out] IP** pIP);
        IVPrint();
    }
    class CPV{
        interface IP;
        interface IV;
    }
}
```

注意：

如果一个 CAR interface 作为输出参数（[out]属性参数），有两种定义方式：

方式一：

java/lang/Object obj = Foo();

这种定义方式中，必须有一个与它名字匹配的 Java 类。名字匹配的规则是："Wrapper"＋CAR interface 名字。

obj 其实是一个 Java 类：

```
package sameWithMethod;
class WrapperInterfaceName
{
};
```

sameWithMethod 就是提供了 Foo 这个 native 函数定义的那个包的名字。

方式二：

myPackage/MyObject myobj = Foo();

myPackage/MyObject 是一个内置了 ElastosInterface ifcInterface 的 Java 类。

上面的两种定义方式中，CAR 的 Foo 具有类似这样的定义：

ECode Foo(Interface *ifc);

2.3.3 Java&CAR属性(Property)

在 C#中，类，结构或者通过 get 和 set 存储方法得到私有域的接口被叫做属性。下面代码显示了动物类物种的属性，这个动物类能使用物种的私有变量。

类似于 C#中：

```
public class Animal{private string species;
```

```
public class Animal
{
    private string species;
    public string Species
    {
        get
        {
            return species;
        }
        set
        {
            species = value;
        }
    }
}
```

2.3.3.1 动态 Property

Java&CAR 提供 Property 机制。定义动态（非 static）的 property 有两种形式：

形式一：

```
@CARMember;
```

形式二：

```
@CARMember(Setter=setFunction, Getter=getFunction);
```

第二种方式中，Setter 或 Getter 也可以省略。

缺省的 Setter 或 Getter 的函数名为：setProperty 名/getProperty 名，即 Property 名前加“set”或“get”。没有声明 Setter 或 Getter 函数，Java&CAR 将用缺省名代替。

如果在相应的 CAR 类（当前 Java 类）中找不到 Setter 或 Getter 函数，则造成该属性的只读（只提供了 Getter 函数）或只写（只提供了 Setter 函数）。

```
@CAR(ElastosModule,  ElastosClass)
public class Animal
{
    @CARMember(Setter="CARSetterFunction", Getter="CARSetterFunction")
    private string species;
}
```

Getter/Setter 函数的定义规范是：

```
//vm/oo/Object.h
/*
 * Generic field header.  We pass this around when we want a generic Field
 * pointer (e.g. for reflection stuff).  Testing the accessFlags for
 * ACC_STATIC allows a proper up-cast.
 */
struct Field {
    ClassObject*    clazz;          /* class in which the field is declared */
    const char*     name;
    const char*     signature;      /* e.g. "I", "[C", "Landroid/os/Debug;" */
    u4              accessFlags;
#ifdef PROFILE_FIELD_ACCESS
    u4              gets;
    u4              puts;
#endif
};
/*
 * Instance field.
 */
struct InstField {
    Field           field;          /* MUST be first item */
    /*
     * This field indicates the byte offset from the beginning of the
     * (Object *) to the actual instance data; e.g., byteOffset==0 is
     * the same as the object pointer (bug!), and byteOffset==4 is 4
     * bytes farther.
     */
    int             byteOffset;
    CARGetter       setter;
    CARSetter       getter;
};

typedef cdecl void (*CARGetter)(void *thisCAR, void *InstField, void *data);
```

```
typedef cdecl void (*CARSetter)(void *thisCAR, void *InstField, void *data);
```

注意：

CAR 提供 Getter/Setter 的函数，可以调用 JNI 来完成自己的功能，当然也能够向 Dalvik 抛出异常。

Java 的 abstract class 上也可以有 CARMember 标识，因为 CARMember 的定义上，有@Inherited 属性。

```
package dalvik.annotation;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Inherited
public @interface CARMember {
    String Setter() default "";
    String Getter() default "";
    String ClassDescriptor() default "";
}
```

2.3.3.2 静态 Property

静态（static）是全虚拟机共享的，它存储于属于类（Class）的空间中。在 Java&CAR 中，它是通过 CAR 的 singleton 机制实现的。

Java&CAR 提供 Property 机制。定义静态（static）的 property 语法形式：

- @CARMember(Setter=setFunction, Getter=getFunction, ClassDescriptor=LclassDescriptor;)
- Setter 或 Getter 可以省略，ClassDescriptor 是一种类似于“Ljava/lang/InternalError;”格式的字符串。
- 缺省的 Setter 或 Getter 的函数名为：setProperty 名/getProperty 名，即 Property 名前加“set”或“get”。没有声明 Setter 或 Getter 函数，Java&CAR 将用缺省名代替。

如果在相应的 CAR 类中找不到 Setter 或 Getter 函数，则造成该属性的只读（只提供了 Getter 函数）或只写（只提供了 Setter 函数）。这里的“相应的 CAR 类”由 ClassDescriptor 属性描述，如果当前 Property 没有提供 ClassDescriptor 信息，则由当前类的 CARSingletonClass 提供。

```
@CAR(ElastosModule,  ElastosClass)
```

```
public class Animal
{
    @CARMember(Setter="CARSetterFunction",          Getter="CARSetterFunction",
ClassDescriptor=”Lcom/kortide/StaticProperty;”)
    static String species;
}
```

第 3 章 JPDA 介绍

3.1 JPDA

JPDA（Java Platform Debugger Architecture）是 Java 平台调试体系结构的缩写，通过 JPDA 提供的 API，开发人员可以方便灵活的搭建 Java 调试应用程序。 JPDA 主要由三个部分组成：Java 虚拟机工具接口（JVMTI），Java 调试线协议（JDWP），以及 Java 调试接口（JDI）。

JPDA 定义了一个完整独立的体系，它由三个相对独立的层次共同组成，而且规定了它们三者之间的交互方式，或者说定义了它们通信的接口。这三个层次由低到高分别是 Java 虚拟机工具接口（JVMTI），Java 调试线协议（JDWP）以及 Java 调试接口（JDI）。这三个模块把调试过程分解成几个很自然的概念：调试者（debugger）和被调试者（debuggee），以及他们中间的通信器。被调试者运行于我们想调试的 Java 虚拟机之上，它可以通过 JVMTI 这个标准接口，监控当前虚拟机的信息；调试者定义了用户可使用的调试接口，通过这些接口，用户可以对被调试虚拟机发送调试命令，同时调试者接受并显示调试结果。在调试者和被调试着之间，调试命令和调试结果，都是通过 JDWP 的通讯协议传输的。所有的命令被封装成 JDWP 命令包，通过传输层发送给被调试者，被调试者接收到 JDWP 命令包后，解析这个命令并转化为 JVMTI 的调用，在被调试者上运行。类似的，JVMTI 的运行结果，被格式化成 JDWP 数据包，发送给调试者并返回给 JDI 调用。而调试器开发人员就是通过 JDI 得到数据，发出指令。

3.1.1 Java 虚拟机工具接口（JVMTI）

JVMTI（Java Virtual Machine Tool Interface）即指 Java 虚拟机工具接口，它是一套由虚拟机直接提供的 native 接口，它处于整个 JPDA 体系的最底层，所有调试功能本质上都需要通过 JVMTI 来提供。通过这些接口，开发人员不仅调试在该虚拟机上运行的 Java 程序，还能查看它们运行的状态，设置回调函数，控制某些环境变量，从而优化程序性能。JVMTI 的前身是 JVMDI 和 JVMPI，它们原来分别被用于提供调试 Java 程序以及 Java 程序调节性能的功能。在 J2SE 5.0 之后 JDK 取代了 JVMDI 和 JVMPI 这两套接口，JVMDI 在最新的 Java SE 6 中已经不提供支持，而 JVMPI 也计划在 Java SE 7 后被彻底取代。

3.1.2 Java 调试线协议（JDWP）

JDWP（Java Debug Wire Protocol）是一个为 Java 调试而设计的一个通讯交互协议，它定义了调试器和被调试程序之间传递的信息的格式。在 JPDA 体系中，作为前端（front-end）的调试者（debugger）进程和后端（back-end）的被调试程序（debuggee）进程之间的交互数据的格式就是由 JDWP 来描述的，它详细完整地定义了请求命令、回应数据和错误代码，保证了前端和后端的 JVMTI 和 JDI 的通信通畅。比如在 Sun 公司提供的实现中，它提供了一个名为 jdwp.dll（jdwp.so）的动态链接库文件，这个动态库文件实现了一个 Agent，它会负责解析前端发出的请求或者命令，并将其转化为 JVMTI 调用，然后将 JVMTI 函数的返回值封装成 JDWP 数据发还给后端。

另外，需要注意的是 JDWP 本身并不包括传输层的实现，传输层需要独立实现，但是 JDWP 包括了和传输层交互的严格的定义，就是说，JDWP 协议虽然不规定通过什么方式传输数据，但是它规定了我们传送的数据的格式。在 Sun 公司提供的 JDK 中，在传输层上，它提供了 socket 方式，以及在 Windows 上的 shared memory 方式。当然，传输层本身就是本机内进程间通信方式和远端通信方式。

3.1.3 Java 调试接口（JDI）

JDI（Java Debug Interface）是三个模块中最高层的接口，在多数的 JDK 中，它是由 Java 语言实现的。 JDI 由针对前端定义的接口组成，通过它，调试工具开发人员就能通过前端虚拟机上的调试器来远程操控后端虚拟机上被调试程序的运行，JDI 不仅能帮助开发人员格式化 JDWP 数据，而且还能为 JDWP 数据传输提供队列、缓存等优化服务。

表 3.1. JPDA 层次比较

模板	层次	编程语言	作用
JVMTI	底层	C	获取及控制当前虚拟机状态
JDWP	中介层	C	定义 JVMTI 和 JDI 交互的数据格式
JDI	高层	Java	提供 Java API 来远程控制被调试虚拟机

每一个虚拟机都应该实现 JVMTI 接口，但是 JDWP 和 JDI 本身与虚拟机并非是不可分的，这三个层之间是通过标准所定义的交互的接口和协议联系起来的，因此它们可以被独立替换或取代，但不会影响到整体调试工具的开发和使用。因此，开发和使用自己的 JDWP 和 JDI 接口实现是可能的。

3.1.4 Java 调试接口的特点

Java 语言是第一个使用虚拟机概念的流行的编程语言，正是因为虚拟机的存在，使很多事情变得简单而轻松，掌握了虚拟机，就掌握了内存分配、线程管理、即时优化等等运行态。同样的，Java 调试的本质，就是和虚拟机打交道，通过操作虚拟机来达到观察调试我们自己代码的目的。这个特点决定了 Java 调试接口和以前其他编程语言的巨大区别。

以 C/C++ 的调试为例，目前比较流行的调试工具是 GDB 和微软的 Visual Studio 自带的 debugger，在这种 debugger 中，首先，我们必须编译一个“debug”模式的程序，这个会比实际的 release 模式程序大很多。其次，在调试过程中，debugger 将会深层接入程序的运行，掌握和控制运行态的一些信息，并将这些信息及时返回。这种介入对运行的效率和内存占用都有一定的需求。基于这些需求，这些 Debugger 本身事实上是提供了，或者说，创建和管理了一个运行态，因此他们的程序算法比较复杂，个头都比较大。对于远端的调试，GDB 也没有很好的默认实现，当然，C/C++ 在这方面也没有特别大的需求。

而 Java 则不同，由于 Java 的运行态已经被虚拟机所很好地管理，因此作为 Java 的 Debugger 无需再自己创造一个可控的运行态，而仅仅需要去操作虚拟机就可以了。 Java 的 JPDA 就是一套为调试和优化服务的虚拟机的操作工具，其中，JVMTI 是整合在虚拟机中的接口，JDWP 是一个通讯层，而 JDI 是前端为开发人员准备好的工具和运行库。

从构架上说，我们可以把 JPDA 看作成是一个 C/S 体系结构的应用，在这个构架下，我们可以方便地通过网络，在任意的地点调试另外一个虚拟机上的程序，这个就很好地解决了部署和测试的问题，尤其满足解决了很多网络时代中的开发应用的需求。前端和后端的分离，也方便用户开发适合于自己的调试工具。

从效率上看，由于 Java 程序本身就是编译成字节码，运行在虚拟机上的，因此调试前后的程序、内存占用都不会有大变化（仅仅是启动一个 JDWP 所需要的内存），任意程度都可以很好地调试，非常方便。而 JPDA 构架下的几个组成部分，JDWP 和 JDI 都比较小，主要的工作可以让虚拟机自己完成。

从灵活性上，Java 调试工具是建立在强大的虚拟机上的，因此，很多前沿的应用，比如动态编译运行，字节码的实时替换等等，都可以通过对虚拟机的改进而得到实现。随着虚拟机技术的逐步发展和深入，各种不同种类，不同应用领域中虚拟机的出现，各种强大的功能的加入，给我们的调试工具也带来很多新的应用。

总而言之，一个先天的，可控的运行态给 Java 的调试工作，给 Java 调试

接口带来了极大的优势和便利。通过 JPDA 这个标准，我们可以从虚拟机中得到我们所需要的信息，完成我们所希望的操作，更好地开发我们的程序。

3.2 JDWP 协议介绍

3.2.1 JDWP 协议介绍

这里首先要说明一下 debugger 和 Dalvik vm。Dalvik vm 中运行着我们希望要调试的程序，它与一般运行的 Java 虚拟机没有什么区别，只是在启动时加载了 Agent JDWP 从而具备了调试功能。而 debugger 就是我们熟知的调试器，它向运行中的 target vm 发送命令来获取 target vm 运行时的状态和控制 Java 程序的执行。Debugger 和 target vm 分别在各自的进程中运行，他们之间的通信协议就是 JDWP。

JDWP 与其他许多协议不同，它仅仅定义了数据传输的格式，但并没有指定具体的传输方式。这就意味着一个 JDWP 的实现可以不需要做任何修改就正常工作在不同的传输方式上（在 JDWP 传输接口中会做详细介绍）。

JDWP 是语言无关的。理论上我们可以选用任意语言实现 JDWP。然而我们注意到，在 JDWP 的两端分别是 target vm 和 debugger。Target vm 端，JDWP 模块必须以 Agent library 的形式在 Java 虚拟机启动时加载，并且它必须通过 Java 虚拟机提供的 JVMTI 接口实现各种 debug 的功能，所以必须使用 C/C++ 语言编写。而 debugger 端就没有这样的限制，可以使用任意语言编写，只要遵守 JDWP 规范即可。

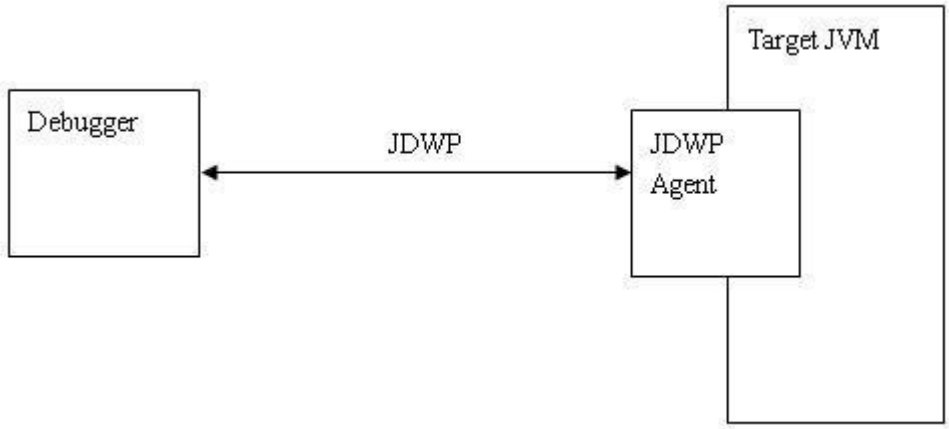

图 3.1. JDWP agent 在调试中扮演的角色

3.2.2 协议分析

JDWP 大致分为两个阶段：握手和应答。握手是在传输层连接建立完成后，做的第一件事：

Debugger 发送 14 bytes 的字符串“JDWP-Handshake”到 target Java 虚拟机

Target Java 虚拟机回复“JDWP-Handshake”

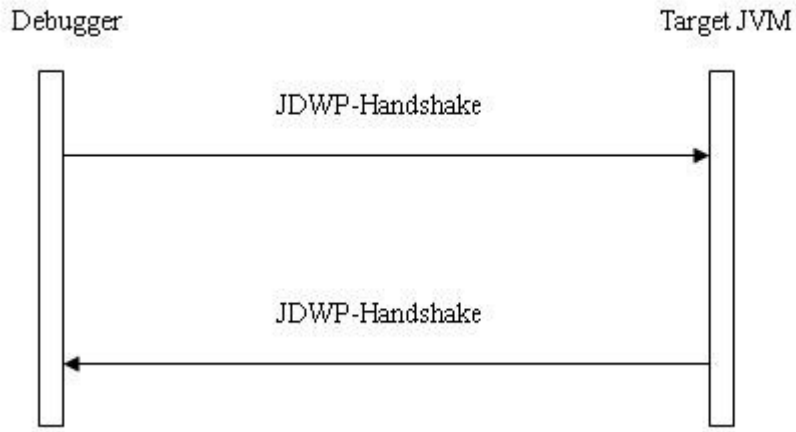

图 3.2. JDWP 的握手协议

握手完成，debugger 就可以向 target Java 虚拟机发送命令了。JDWP 是通过命令（command）和回复（reply）进行通信的，这与 HTTP 有些相似。JDWP 本身是无状态的，因此对 command 出现的顺序并不受限制。

JDWP 有两种基本的包（packet）类型：命令包（command packet）和回复包（reply packet）。

Debugger 和 target Java 虚拟机都有可能发送 command packet。Debugger 通过发送 command packet 获取 target Java 虚拟机的信息以及控制程序的执行。Target Java 虚拟机通过发送 command packet 通知 debugger 某些事件的发生，如到达断点或是产生异常。

Reply packet 是用来回复 command packet 该命令是否执行成功，如果成功 reply packet 还有可能包含 command packet 请求的数据，比如当前的线程信息或者变量的值。从 target Java 虚拟机发送的事件消息是不需要回复的。还有一点需要注意的是，JDWP 是异步的：command packet 的发送方不需要等待接收到 reply packet 就可以继续发送下一个 command packet。

3.2.3 Packet 的结构

Packet 分为包头（header）和数据（data）两部分组成。包头部分的结构和长度是固定，而数据部分的长度是可变的，具体内容视 packet 的内容而定。Command packet 和 reply packet 的包头长度相同，都是 11 个 bytes，这样更有利于传输层的抽象和实现。

如 3.3 为 Command packet 的 header 的结构

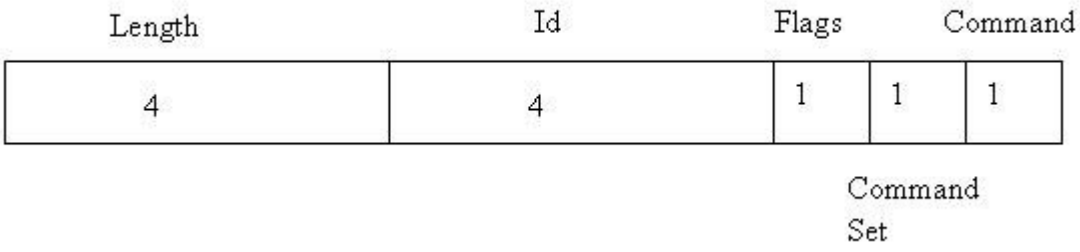

图 3.3 JDWP command packet 结构

- Length 是整个 packet 的长度，包括 length 部分。因为包头的长度是固定的 11 bytes，所以如果一个 command packet 没有数据部分，则 length 的值就是 11。
- Id 是一个唯一值，用来标记和识别 reply 所属的 command。Reply packet 与它所回复的 command packet 具有相同的 Id，异步的消息就是通过 Id 来配对识别的。
- Flags 目前对于 command packet 值始终是 0。
- Command Set 相当于一个 command 的分组，一些功能相近的 command 被分在同一个 Command Set 中。Command Set 的值被划分为 3 个部分：
- 0-63：从 debugger 发往 target Java 虚拟机的命令
- 64 - 127： 从 target Java 虚拟机发往 debugger 的命令
- 128 - 256： 预留的自定义和扩展命令

图 3.4 为 Reply packet 的 header 的结构：

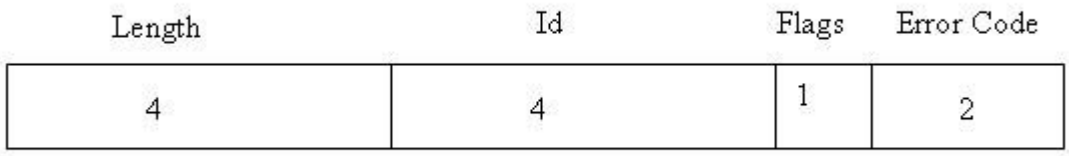

图 3.4. JDWP reply packet 结构

- Length、Id 作用与 command packet 中的一样。
- Flags 目前对于 reply packet 值始终是 0x80。我们可以通过 Flags 的值来判断接收到的 packet 是 command 还是 reply。
- Error Code 用来表示被回复的命令是否被正确执行了。零表示正确，非零表示执行错误。
- Data 的内容和结构依据不同的 command 和 reply 都有所不同。比如请求一个对象成员变量值的 command，它的 data 中就包含该对象的 id 和成员变量的 id。而 reply 中则包含该成员变量的值。

JDWP 传输接口（Java Debug Wire Protocol Transport Interface）前面提到 JDWP 的定义是与传输层独立的，但如何使 JDWP 能够无缝的使用不同的传输实现，而又无需修改 JDWP 本身的代码？ JDWP 传输接口（Java Debug Wire Protocol Transport Interface）为我们解决了这个问题。

JDWP 传输接口定义了一系列的方法用来定义 JDWP 与传输层实现之间的交互方式。首先传输层的必须以动态链接库的方式实现，并且暴露一系列的标准接口供 JDWP 使用。与 JNI 和 JVMTI 类似，访问传输层也需要一个环境指针（jdwpTransport），通过这个指针可以访问传输层提供的所有方法。

当 JDWP agent 被 Java 虚拟机加载后，JDWP 会根据参数去加载指定的传输层实现（Sun 的 JDK 在 Windows 提供 socket 和 share memory 两种传输方式，而在 Linux 上只有 socket 方式）。传输层实现的动态链接库实现必须暴露 jdwpTransport_OnLoad 接口，JDWP agent 在加载传输层动态链接库后会调用该接口进行传输层的初始化。接口定义如下：

```
JNIEXPORT jint JNICALL
jdwpTransport_OnLoad(JavaVM *jvm,
jdwpTransportCallback *callback,
jint version,
jdwpTransportEnv** env);
```

callback 参数指向一个内存管理的函数表，传输层用它来进行内存的分配和释放，结构定义如下：

```
typedef struct jdwpTransportCallback {
void* (*alloc)(jint numBytes);
void (*free)(void *buffer);
} jdwpTransportCallback;
```

env 参数是环境指针，指向的函数表由传输层初始化。

JDWP 传输层定义的接口主要分为两类：连接管理和 I/O 操作。

3.2.4 连接管理

连接管理接口主要负责连接的建立和关闭。一个连接为 JDWP 和 debugger 提供了可靠的数据流。Packet 被接收的顺序严格的按照被写入连接的顺序。

连接的建立是双向的，即 JDWP 可以主动去连接 debugger 或者 JDWP 等待 debugger 的连接。对于主动去连接 debugger，需要调用方法 Attach，定义如下：

```
jdwpTransportError
Attach(jdwpTransportEnv* env, const char* address,
jlong attachTimeout, jlong handshakeTimeout)
```

在连接建立后，会立即进行握手操作，确保对方也在使用 JDWP。因此方法参数中分别指定了 attch 和握手的超时时间。

address 参数因传输层的实现不同而有不同的格式。对于 socket，address 是主机地址；对于 share memory 则是共享内存的名称。

JDWP 等待 debugger 连接的方式，首先需要调用 StartListening 方法，定义如下：

```
jdwpTransportError
StartListening(jdwpTransportEnv* env, const char* address,
char** actualAddress)
```

该方法将使 JDWP 处于监听状态，随后调用 Accept 方法接收连接：

```
jdwpTransportError
Accept(jdwpTransportEnv* env, jlong acceptTimeout, jlong
handshakeTimeout)
```

与 Attach 方法类似，在连接建立后，会立即进行握手操作。

3.2.5 I/O 操作

I/O 操作接口主要是负责从传输层读写 packet。有 ReadPacket 和 WritePacket 两个方法：

```
jdwpTransportError
ReadPacket(jdwpTransportEnv* env, jdwpPacket* packet)
jdwpTransportError
WritePacket(jdwpTransportEnv* env, const jdwpPacket* packet)
```

参数 packet 是要被读写的 packet，其结构 jdwpPacket 与我们开始提到的 JDWP packet 结构一致，定义如下：

```
typedef struct{
jint len; // packet length
jint id; // packet id
jbyte flags; // value is 0
jbyte cmdSet; // command set
jbyte cmd; // command in specific command set
jbyte *data; // data carried by packet
} jdwpCmdPacket;
typedef struct {
jint len; // packet length
jint id; // packet id
jbyte flags; // value 0x80
jshort errorCode; // error code
jbyte *data; // data carried by packet
} jdwpReplyPacket;

typedef struct jdwpPacket {
union {
jdwpCmdPacket cmd;
jdwpReplyPacket reply;
} type;
} jdwpPacket;
```

3.2.6 JDWP 的命令实现机制

下面将通过讲解一个 JDWP 命令的实例来介绍 JDWP 命令的实现机制。JDWP 作为一种协议，它的作用就在于充当了调试器与 Java 虚拟机的沟通桥梁。通俗点讲，调试器在调试过程中需要不断向 Java 虚拟机查询各种信息，那么 JDWP 就规定了查询的具体方式。

在 Java 6.0 中，JDWP 包含了 18 组命令集 合，其中每个命令集合又包含了若干条命令。那么这些命令是如何实现的呢？下面我们先来看一个最简单的 VirtualMachine（命令集合 1）的 Version 命令，以此来剖析其中的实现细节。

因为 JDWP 在整个 JPDA 框架中处于相对底层的位置。

```
CommandPacket packet = new CommandPacket(
JDWPCommands.VirtualMachineCommandSet.CommandSetID,
JDWPCommands.VirtualMachineCommandSet.VersionCommand);
ReplyPacket reply = debuggeeWrapper.vmMirror.performCommand(packet);
String description = reply.getNextValueAsString();
int jdwpMajor = reply.getNextValueAsInt();
int jdwpMinor = reply.getNextValueAsInt();
String vmVersion = reply.getNextValueAsString();
String vmName = reply.getNextValueAsString();
logWriter.println("description\t= " + description);
logWriter.println("jdwpMajor\t= " + jdwpMajor);
logWriter.println("jdwpMinor\t= " + jdwpMinor);
logWriter.println("vmVersion\t= " + vmVersion);
logWriter.println("vmName\t\t= " + vmName);
```

首先，我们会创建一个 VirtualMachine 的 Version 命令的命令包实例 packet。该命令包主要就是配置了两个参数：CommandSetID 和 VersionComamnd，它们的值均为 1。表明执行的命令是属于命令集合 1 的命令 1，即 VirtualMachine 的 Version 命令。

然后在 performCommand 方法中发送了该命令并收到了 JDWP 的回复包 reply。通过解析 reply，得到了该命令的回复信息。

```
description = Java 虚拟机 version 1.6.0 (IBM J9 VM, J2RE 1.6.0 IBM J9 2.4 Windows XP
x86-32
jvmwi3260sr5-20090519_35743 (JIT enabled, AOT enabled)
J9VM - 20090519_035743_lHdSMr
```

```
JIT - r9_20090518_2017
GC - 20090417_AA, 2.4)
jdwpMajor = 1
jdwpMinor = 6
vmVersion = 1.6.0
vmName = IBM J9 VM
```

测试用例的执行结果显示，我们通过该命令获得了 Java 虚拟机的版本信息，这正是 VirtualMachine 的 Version 命令的作用。

JDWP 接收到的是调试器发送的命令包，返回的就是反馈信息的回复包。模拟的调试器会发送 VirtualMachine 的 Version 命令。JDWP 在执行完该命令后就向调试器返回 Java 虚拟机的版本信息。

返回信息的包内容同样是在 JDWP Spec 里面规定的。Spec 中的描述如下（测试用例中的回复包解析就是参照这个规定的):

表 3.2 JDWP Spec 的规定

类型	名称	说明
string	description	VM version 的文字描述信息
int	jdwpMajor	JDWP 主板本号
int	jdwpMinor	JDWP 次版本号
string	vmVersion	VM JRE 版本，也就是 java. version 属性值
string	vmName	VM 的名称，也就是 java. vm. name 属性值

下面以 Apache Harmony 的 JDWP 为例，介绍 JDWP 内部是如何处理接收到的命令并返回回复包的实现架构。

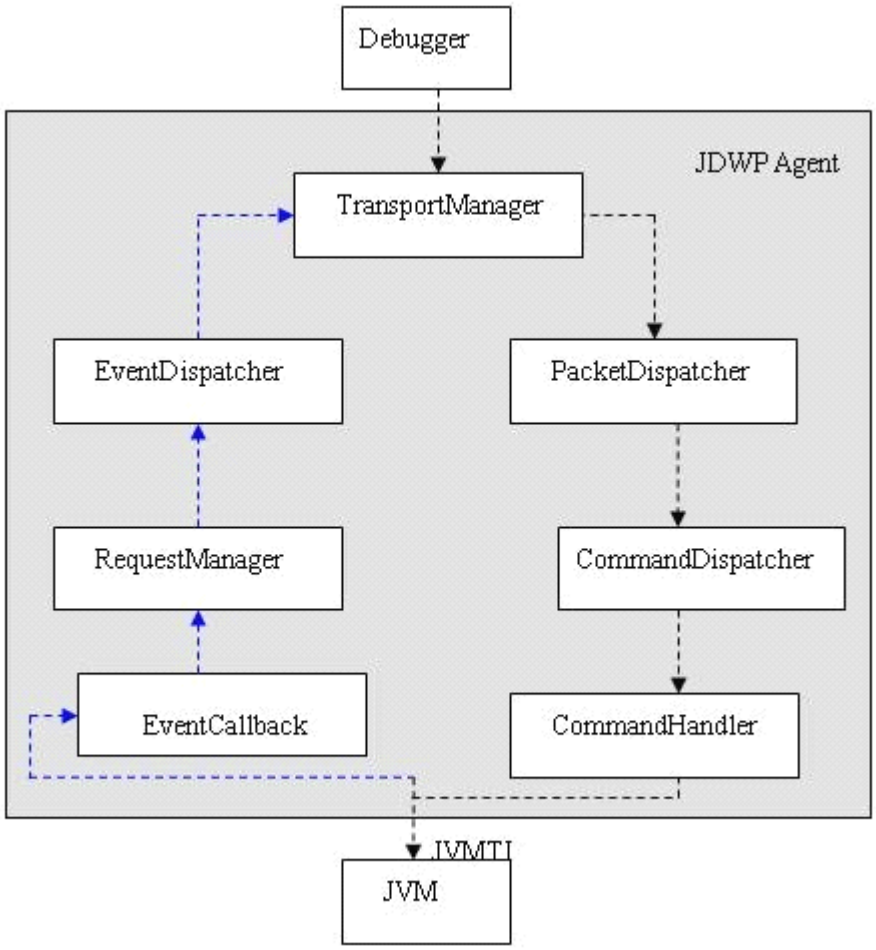

图 3.5. JDWP 架构图

如图 3.5 所示，JDWP 接收和发送的包都会经过 TransportManager 进行处理。JDWP 的应用层与传输层是独立的，就在于 TransportManager 调用的是 JDWP 传输接口（Java Debug Wire Protocol Transport Interface），所以无需关心底层网络的具体传输实现。TransportManager 的主要作用就是充当 JDWP 与外界通讯的数据包的中转站，负责将 JDWP 的命令包在接收后进行解析或是对回复包在发送前进行打包，从而使 JDWP 能够专注于应用层的实现。

对于收到的命令包，TransportManager 处理后会转给 PacketDispatcher，进一步封装后会继续转到 CommandDispatcher。然后，CommandDispatcher 会根据命令中提供的命令组号和命令号创建一个具体的 CommandHandler 来处理 JDWP 命令。

其中，CommandHandler 才是真正执行 JDWP 命令的类。我们会为每个 JDWP 命令都定义一个相对应的 CommandHandler 的子类，当接收到某个命令时，就会创建处理该命令的 CommandHandler 的子类的实例来作具体的处理。

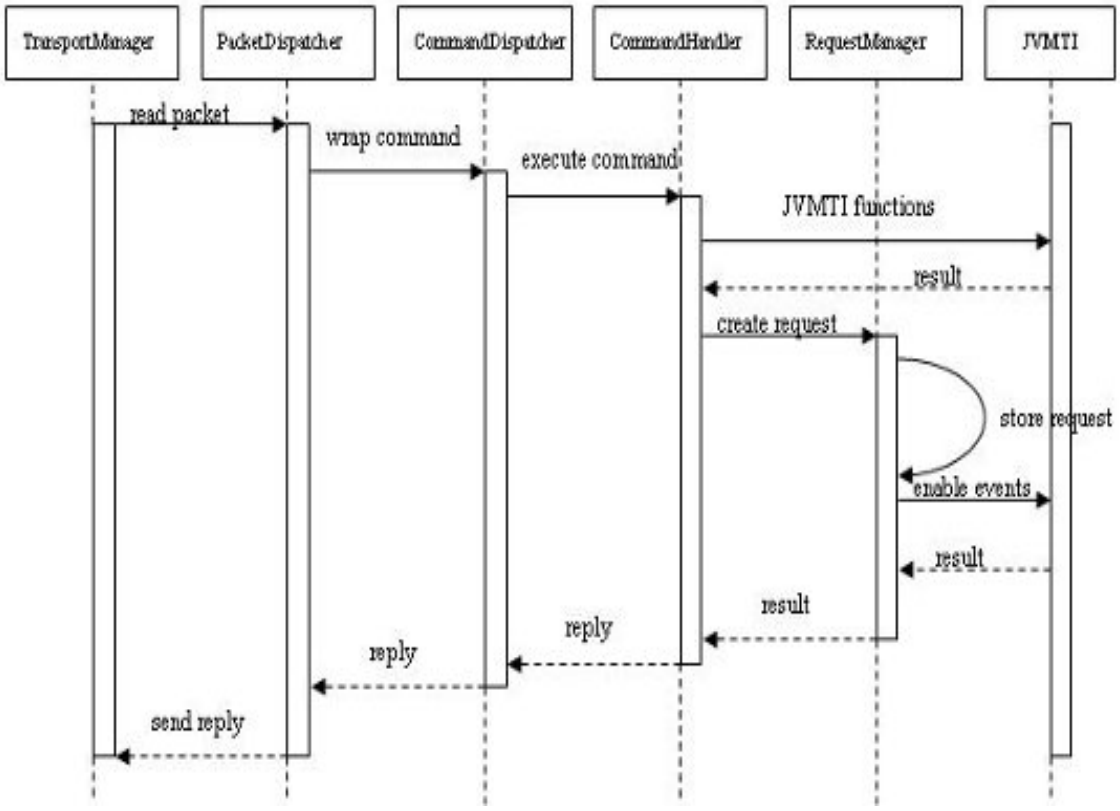

图 3.6. JDWP 命令处理流程

3.2.7 单线程执行的命令

图 3.6 就是一个命令的处理流程图。可以看到，对于一个可以直接在该线程中完成的命令（我们称为单线程执行的命令），一般其内部会调用 JVMTI 方法和 JNI 方法来真正对 Java 虚拟机进行操作。

例如，VirtualMachine 的 Version 命令中，对于 vmVersion 和 vmName 属性，我们可以通过 JNI 来调用 Java 方法 System.getProperty 来获取。然后，JDWP 将回复包中所需要的结果封装到包中后交由 TransportManager 来进行后续操作。

多线程执行的命令

对于一些较为复杂的命令，是无法在 CommandHandler 子类的处理线程中完成的。例如，ClassType 的 InvokeMethod 命令，它会要求在指定的某个线程中执行一个静态方法。显然，CommandHandler 子类的当前线程并不是所要求的线程。

这时，JDWP 线程会先把这个请求先放到一个列表中，然后等待，直到所要

求的线程执行完那个静态方法后，再把结果返回给调试器。

3.2.8 JDWP 的事件处理机制

前面介绍的 VirtualMachine 的 Version 命令过程非常简单，就是一个查询和信息返回的过程。在实际调试过程中，一个 JDI 的命令往往会有数条这类简单的查询命令参与，而且会涉及到很多更为复杂的命令。要了解更为复杂的 JDWP 命令实现机制，就必须介绍 JDWP 的事件处理机制。

在 Java 虚拟机中，我们会接触到许多事件，例如 VM 的初始化，类的装载，异常的发生，断点的触发等等。那么这些事件调试器是如何通过 JDWP 来获知的呢？下面，我们通过介绍在调试过程中断点的触发是如何实现的，来为大家揭示其中的实现机制。

在这里，任意调试一段 Java 程序，并在某一行中加入断点。然后，执行到该断点，此时所有 Java 线程都处于 suspend 状态。这是很常见的断点触发过程。为了记录在此过程中 JDWP 的行为，使用一个开启了 trace 信息的 JDWP。虽然这并不是一个复杂的操作，但整个 trace 信息也有几千行。

可见，作为相对底层的 JDWP，其实际处理的命令要比想象的多许多。为了介绍 JDWP 的事件处理机制，其中比较重要的一些 trace 信息：

```
[RequestManager.cpp:601] AddRequest: event=BREAKPOINT[2], req=48, modCount=1,
policy=1
[RequestManager.cpp:791] GenerateEvents: event #0: kind=BREAKPOINT, req=48
[RequestManager.cpp:1543] HandleBreakpoint: BREAKPOINT events: count=1,
suspendPolicy=1,
location=0

[RequestManager.cpp:1575] HandleBreakpoint: post set of 1
[EventDispatcher.cpp:415] PostEventSet -- wait for release on event: thread=4185A5A0,
name=(null), eventKind=2
[EventDispatcher.cpp:309] SuspendOnEvent -- send event set: id=3, policy=1
[EventDispatcher.cpp:334] SuspendOnEvent -- wait for thread on event: thread=4185A5A0,
name=(null)
[EventDispatcher.cpp:349] SuspendOnEvent -- suspend thread on event: thread=4185A5A0,
name=(null)
[EventDispatcher.cpp:360] SuspendOnEvent -- release thread on event: thread=4185A5A0,
name=(null)
```

首先，调试器需要发起一个断点的请求，这是通过 JDWP 的 Set 命令完成的。在 trace 中，看到 AddRequest 就是做了这件事。可以清楚的发现，调试器请求的是一个断点信息（event=BREAKPOINT[2]）。

在 JDWP 的实现中，这一过程表现为：在 Set 命令中会生成一个具体的 request, JDWP 的 RequestManager 会记录这个 request（request 中会包含一些过滤条件，当事件发生时 RequestManager 会过滤掉不符合预先设定条件的事件），并通过 JVMTI 的 SetEventNotificationMode 方法使这个事件触发生效（否则事件发生时 Java 虚拟机不会报告）。

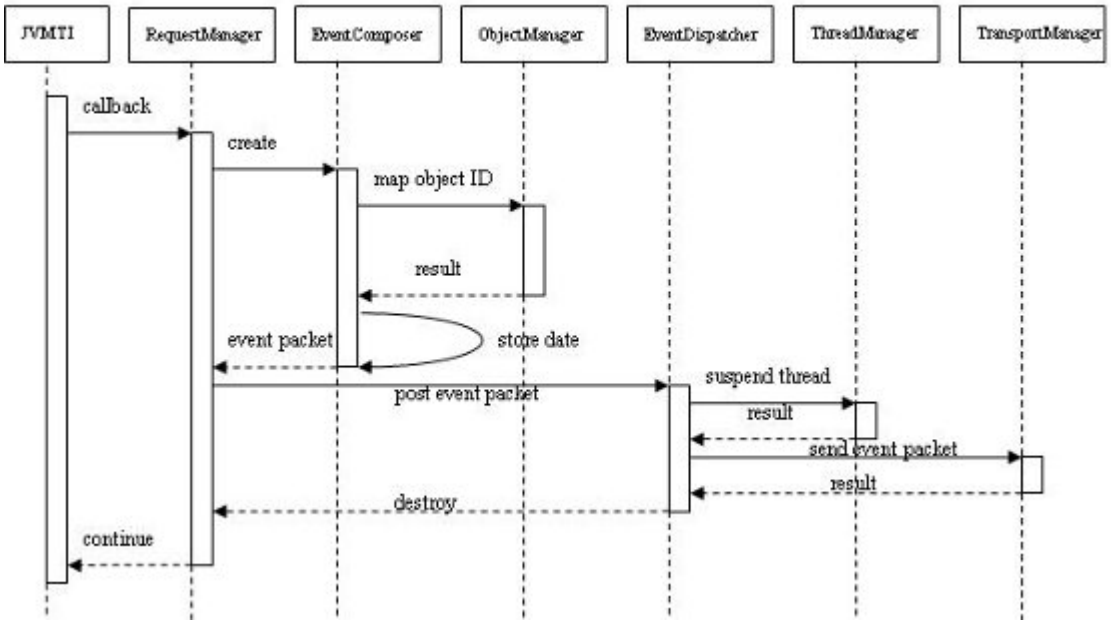

图 7. JDWP 事件处理流程

当断点发生时，Java 虚拟机就会调用 JDWP 中预先定义好的处理该事件的回调函数。在 trace 中，HandleBreakpoint 就是先开始在 JDWP 中定义好的处理断点信息的回调函数。它的作用就是要生成一个 JDWP 端所描述的断点事件来告知调试器（Java 虚拟机只是触发了一个 JVMTI 的消息）。

由于断点的事件在调试器申请时就要求所有 Java 线程在断点触发时被 suspend，那这一步由谁来完成呢？这里要谈到一个细节问题，HandleBreakpoint 作为一个回调函数，其执行线程其实就是断点触发的 Java 线程。

显然，不应该由它来负责 suspend 所有 Java 线程。就是要把该断点触发信息返回给调试器。如果我们先返回信息，然后 suspend 所有 Java 线程，这就无法保证在调试器收到信息时所有 Java 线程已经被 suspend。

反之，先 Suspend 了所有 Java 线程，谁来负责发送信息给调试器呢？

为了解决这个问题，通过 JDWP 的 EventDispatcher 线程来帮我们 suspend 线程和发送信息。实现的过程是，我们让触发断点的 Java 线程来 PostEventSet（trace 中可以看到），把生成的 JDWP 事件放到一个队列中，然后就开始等待。由 EventDispatcher 线程来负责从队列中取出 JDWP 事件，并根据事件中的设定，来 suspend 所要求的 Java 线程并发送出该事件。

在这里，我们在事件触发的 Java 线程和 EventDispatcher 线程之间添加了一个同步机制，当事件发送出去后，事件触发的 Java 线程会把 JDWP 中的该事件删除，到这里，整个 JDWP 事件处理就完成了。

第 4 章 CAR 构件快照

4.1 复合对象的快照

通过创建一个自定义的 elastos.CarDebugger.CARObjDesc 对象来描述复合对象的状态,包括:(这些信息通过调用 IClassInfo 接口函数获得)

```
String     name_ElastosClass;          //实现 Java 类的 CAR 类名字
String     name_JavaClass;         //由 CAR 类实现的 Java 类的名字
String     ID;                     //CAR 类的 ID
int        CARObjSize;             //CAR 对象占内存的大小
int        CARObjAddr;                 //CAR 对象的首地址
byte[]     MemDump;                //CAR 对象的前若干个字节
Class      ElastosModuleInfo;          //CAR 类所在的 CAR 构件名字
Class      CMethodInfo;
boolean    IsSingleton;            //CAR 类是否是单例类
int        count_Method;           //CAR 类中的方法个数
int        count_Aspect;           //CAR 类中的方面个数
int        count_Aggregatee;       //CAR 类中的聚合个数
int        count_Interface;        //CAR 类中的接口个数
int        count_Constructor;      //CAR 类中的构造函数个数
int        count_CallbackInterface; //CAR 类中的回调接口个数
int        count_CallbackMethod;   //CAR 类中的回调函数个数
```

4.2 方法信息的快照

Dalvik 存在一个方法调用栈,每调用一个方法的时候就会创建一个栈帧并压入方法调用栈,所以通过创建一个 elastos.CARDebugger.MethodDesc 对象描述当前执行方法的栈帧信息,包括:

获得当前线程(Thread)中的当前栈帧的指针 curFrame,取出当前方法栈帧的信息(包括当前方法的 Method 结构体)，放入 elastos.CARDebugger.MethodDesc,并在 Debugger 端显示出来

栈帧 Frame 构成:

```
-------------------------------------------------------------------------------------------
    Return          返回值
    Params          参数
    Locals          局部变量
    Break frame     上一帧的辅助信息
void*     prevFame;          //调用方法的栈帧指针
          const u2* savedPc;             //调用方法的程序计数器
          const Method*method;  //当前方法的 Method 结构体指针
          union{                         //当前方法的程序计数器
               Object** localRefTop;
               Const u2*      currentPc;
          }
    Regular frame 当前帧的辅助信息
void*     prevFame;          //调用方法的栈帧指针
          const u2* savedPc;             //调用方法的程序计数器
          const Method*method;  //当前方法的 Method 结构体指针
          union{                         //当前方法的程序计数器
               Object** localRefTop;
               Const u2*      currentPc;
          }
    Args
```

4.3 线程快照

创建一个 elastos.CARDebugger.ThreadObjDesc 对象来描述当前线程的信息,包括通过 dvmThreadSelf()函数取到当前线程的数据结构指针(Thread*),然后一次取出线程的各种信息,放入 elastos.CARDebugger.ThreadObjDesc 中,并在 Debugger 端显示出来。

```
Thread{
          U4   threadId;                              //线程 ID
          ThreadStatus   status;                      //线程状态
          Int   suspendCount;                    //挂起次数
```

```
        Int    dbgSuspendCount;                    //Debug 挂起次数
        BoolisSuspended;                           //是否挂起
        Pthread_thandle;                       //
        pid_t       systemTid;                     //
u1*  interpStackStart;                         //
const u1* interpStackEnd;                      //
int    interpStackSize;                        //
bool statckOverflowed;                         //
void*       curFrame;                          //当前方法调用栈帧
Object*   exception;                           //
Object*   threadObj;                           //
JNIEnv*  jniEnv;                               //JNI 环境变量指针
ReferenceTable      jniLocalRefTable;          //
ReferenceTable      jniMonitorRefTable;        //
ReferenceTable      jniMonitorRefTable;        //
Object*   classLoaderOverride;                 //
Monitor* waitMonitor;                          //
Boolinterrupted;                               //
BoolthrowingOOME;                              //
Struct Thread*prev;                        //
Struct Thread*next;                    //
DebugInvokeReq    invokeReq;                   //
Struct LockedObjectData*     pLockedObjects;   //
Int    allocLimit;                             //
BoolcpuClockBaseSet;                           //
U8   cpuClockBase;                         //
AllocProfStateallocProf;                       //
U4   stackCrc;                                 //
Void*       appletThreadId;                    //
U4   pArgumentList;                            //
}
```

4.4 几个特别对象（Callbacks/Delegates,MsgLoop Thread)的快照

4.4.1 Callbacks 对象

Callbacks 对象用于处理异步事件(event),包含了若干个 Callback 方法的信息,通过创建一个 elastos.CARDebugger.CallbackMethod 对象数组来进行描述,每个 CallbackMethod 对象包括:

String involvedClass;　　//Callback 方法所在的类名

String methodName;//Callback 方法的名字

当要调试一个 Callbacks 对象的时候,就会创建一个 CallbackMethod 对象数组,分别设置数组中每一个 CallbackMethod 对象的信息

4.4.2 Delegates 对象

Delegates 对象与 Callbacks 对象类似,只是用于处理同步事件(errand),包含了若干个 Delegate 方法的信息,通过创建一个 elastos.CARDebugger.DelegateMethod 对象数组进行描述,每个 DelegateMethod 对象包括:

String involvedClass;　　//Delegate 方法所在的类名

String methodName;//Delegate 方法的名字

当要调试一个 Delegates 对象的时候,就会创建一个 DelegateMethod 对象数组,分别设置数组中每一个 DelegateMethod 对象的信息

4.4.3 MsgLoopThread 对象

启动 Dalvik 时会创建一个消息循环线程(MsgLoopThread)用于处理回调,MsgLoopThread 含有一个线程专有数据 ICallbackContext*(CallbackContext 接口指针),可用于访问 CallbackContext 接口函数:

```
PostCallbackEvent()
SendCallbackEvent()
RequestToFinish()
CancelAllPendingCallbacks()
CancelPendingCallbacks()
CancelCallbackEvents()
HandleCallbackEvents()
```

```
CancelAllCallbackEvents()
```

在 Java 对象里面定义了一个 ICallbackContext 域来描述 ICalbackContext 的指针。

第 5 章 JPDA 的实现和 C/C++调试器的介入

本章讨论在 DalvikVM 端和 JDWP 做相应的扩展，以便能在 IDE 端（Eclipse）检测到 Java 类调用 CAR 构件中，对 CAR 运行环境，运行上下文及 CAR 构件本身进行快照的传输以及 CAR 构件调用注册到 Java 中的回调函数时，返回的回调函数的快照。

5.1 复合对象的快照传输

5.1.1 修改 dalvik/VM 下的 debugger.c

(1)原始 Java 类

```
public class SetGetClass1 {
    public int a1;
    public int b1;
    public int c1;
    public double d1;
    SetGetClass1()
    {
        a1=11;
        b1=12;
        c1=13;
        d1=14.2324;
    }
}
```

(2)Java 程序中创建对象

```
SetGetClass1 test1=new SetGetClass1();
```

在 Eclipse 的调试器下，看到的都是可以 java 类中的各种参数的具体情况。

接着通过改动里面 dalvik/VM 中的代码，使得在 Eclipse 的调试器下可看到 CAR 中的详细信息。

改动 dalvik/VM 下的 debugger.c:输出类 SetGetClass1 的域信息时,增加一个域。

```
void dvmDbgOutputAllFields(RefTypeId refTypeId, bool withGeneric,
    ExpandBuf* pReply)
{
    static const u1 genericSignature[1] = "";
    ClassObject* clazz;
    Field* field;
    u4 declared;
    int i;
    clazz = refTypeIdToClassObject(refTypeId);
    assert(clazz != NULL);
//------------------------------------------------------------------------
//如果是 SetGetClass1 类,则 SetGetClass1 类声明的域数加一
    if(strcmp(clazz->descriptor,"Lcom/kortide/SetGetClass1;")==0)
    {
        declared = clazz->sfieldCount + clazz->ifieldCount+1;
    }
    else{
        declared = clazz->sfieldCount + clazz->ifieldCount;
    }
//------------------------------------------------------------------------
    expandBufAdd4BE(pReply, declared);
    int accessflag;
    u1 *p;
    for (i = 0; i < clazz->sfieldCount; i++) {
        field = (Field*) &clazz->sfields[i];
        expandBufAddFieldId(pReply, fieldToFieldId(field));
        expandBufAddUtf8String(pReply, (const u1*) field->name);
        expandBufAddUtf8String(pReply, (const u1*) field->signature);
        if (withGeneric)
            expandBufAddUtf8String(pReply, genericSignature);
        expandBufAdd4BE(pReply, field->accessFlags);
    }
    for (i = 0; i < clazz->ifieldCount; i++) {
        field = (Field*) &clazz->ifields[i];
```

```
            expandBufAddFieldId(pReply, fieldToFieldId(field));
            expandBufAddUtf8String(pReply, (const u1*) field->name);
            expandBufAddUtf8String(pReply, (const u1*) field->signature);
            if (withGeneric)
                expandBufAddUtf8String(pReply, genericSignature);
            expandBufAdd4BE(pReply, field->accessFlags);

        if (i == 0) {
            p = field->signature;
            accessflag = field->accessFlags;
        }
        }
//---------------------------------------------------------------------------
// 如果是 SetGetClass1 类，则增加一个域，域信息包括
FieldId:1111,FieldName:"testFieldName",FieldSignature:'I',FieldAccessFlag
        if(strcmp(clazz->descriptor,"Lcom/kortide/SetGetClass1;")==0)
        {
            expandBufAddFieldId(pReply, 1111);
            expandBufAddUtf8String(pReply, "testFieldName");
            expandBufAddUtf8String(pReply, p);
            if(withGeneric)
                expandBufAddUtf8String(pReply, genericSignature);
            expandBufAdd4BE(pReply, accessflag);
        }
//---------------------------------------------------------------------
}
```

5.1.2 修改 dalvik/VM/JDWP jdwphandler.c

改动 dalvik/VM/JDWP 的 javahandler.c：当 fieldID==2222 时，Eclipse 发送 JDWP 的 GetValues 请求时,DalvikVM 增加一个新域，并创建对象填充新域。

```
static JdwpError handleOR_GetValues(JdwpState* state,
        const u1* buf, int dataLen, ExpandBuf* pReply)
{
```

```
    ObjectId objectId;
    u4 numFields;
    int i;
    objectId = dvmReadObjectId(&buf);
    numFields = read4BE(&buf);
    LOGV("    Req for %d fields from objectId=0x%llx\n", numFields, objectId);
    expandBufAdd4BE(pReply, numFields);
    for (i = 0; i < (int) numFields; i++) {
        FieldId fieldId;
        u1 fieldTag;
        int width;
        u1* ptr;
        const char* fieldName;
        fieldId = dvmReadFieldId(&buf);
//-------------------------------------------------------------------------
      if(fieldId==2222)
    {LOGD("------GetFieldValue
ClassObject:SetGetClass1,FieldID:%d,FieldName:CCarObject",fieldId);
        int intVal = 5678;
        fieldTag=JT_INT;
        fieldTag=JT_OBJECT;
        expandBufAdd1(pReply, fieldTag);
        width = dvmDbgGetTagWidth(fieldTag);
        ptr=expandBufAddSpace(pReply,width);
        //set4BE(ptr, intVal);
//创建一个 java 对象填充新域。
    Thread* self=dvmThreadSelf();
    jclass cls=dvmFindClass("Lcom/kortide/CCarObject;", tonyloader);
    jobject CCarObjectObj = (jobject) dvmAllocObject(cls, ALLOC_DEFAULT);
    ReferenceTable* pRef = &self->jniLocalRefTable;
    if (!dvmAddToReferenceTable(pRef, (Object*)CCarObjectObj)) {
            dvmDumpReferenceTable(pRef, "JNI local");
            dvmDumpThread(self, false);
            dvmAbort();
```

```
            } else {}
            dvmSetObjectId(ptr, (ObjectId)CCarObjectObj);
            return ERR_NONE;
        }
//-------------------------------------------------------------------
            fieldTag = dvmDbgGetFieldTag(objectId, fieldId);
            width = dvmDbgGetTagWidth(fieldTag);
            LOGV("        --> fieldId %x --> tag '%c'(%d)\n",
                fieldId, fieldTag, width);
            expandBufAdd1(pReply, fieldTag);
            ptr = expandBufAddSpace(pReply, width);
            dvmDbgGetFieldValue(objectId, fieldId, ptr, width);
        }
        return ERR_NONE;
}
```

在 Eclipse 的 debug 窗口下，看到的都是 CARObjDesc 对象的各种参数的具体情况，如图 5.1 所示。

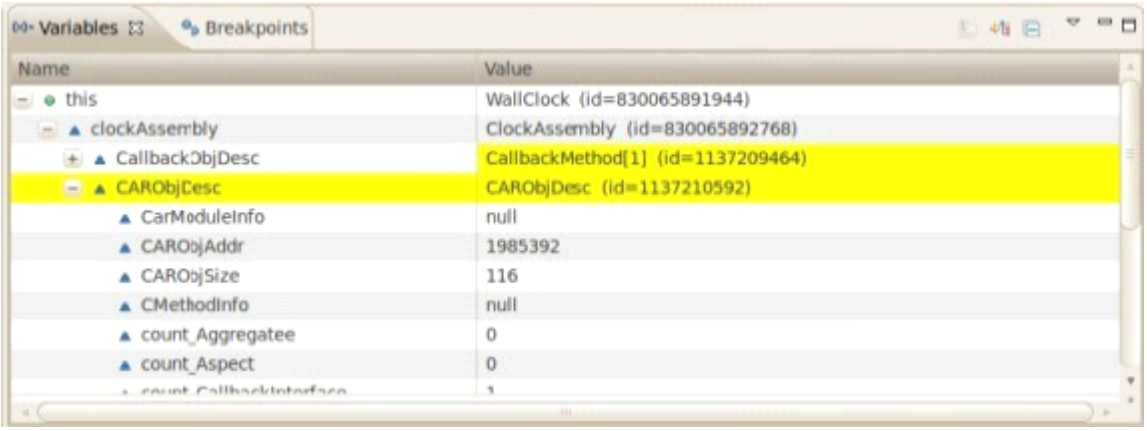

图 5.1 复合对象中 CAR 存在的情况

如图 5.1 所示 CARObjDesc 中的各种参数 ElastosModuleInfo，该对象的地址空间 CARObjAddr 为 1985392，该对象的大小 CARObjSize 为 116 等等。这些参数都对 CAR 运行环境，运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议把信息传递给调试器，真正实现复合对象的调试。

5.2 注册 callback 到 Java 类中

在 CAR 构件运行的程序中，当 CAR 构件的事件触发时，调用 java 中注册的回调函数时，在 JPDA 的系统中，通过在 JdwpHandler.c 文件的 handleOR_GetValueys 函数中分别增加两个对象来描述触发 CAR 构件中注册到 JAVA 的回调函数，一个对象用来描述 CAR 对象本身，另一个对象用来描述该注册的 Callback 对象。

现在有两个描述对象 CARObjDesc(描述 CAR 对象)和 CallbackObjDesc(描述 Callbacks 对象)位于 JdwpHandler.c 文件的 handleOR_GetValueys 函数。

```
    Field*  field = (Field*)fieldId;、
      if(field->signature[0]=='L')
      {
ClassObject* fieldClazz = dvmFindClass(field->signature,field->clazz->classLoader);
                              if(strcmp(fieldClazz->descriptor,"Ljava/lang/Callbacks;")==0   ||
\                                                    (fieldClazz->super!=NULL     &&
strcmp(fieldClazz->super->descriptor,"Ljava/lang/Callbacks;")==0)) {
            //如果这个域的类或父类是 Callbacks
            ArrayObject* CallbackObjDesc_Obj;
            Object* Callback_Obj = NULL;
     Callback_Obj = dvmGetFieldObject(Java_Obj,((InstField*)field)->byteOffset);//得到这
个 Callbacks 对象
            u4      *CallbackMethods;
            u4      *thisObjs;
            u4       icount;
            //Callbacks 对象中回调函数的数组
                  CallbackMethods = (const u4*)(((ArrayObject*)dvmGetFieldObject((const
Object*)Callback_Obj, gDvm.offJavaLangDelegate_value))->contents); //Callbacks 对象中
对象数组
                        thisObjs = (const u4*)(((ArrayObject*)dvmGetFieldObject((const
Object*)Callback_Obj,gDvm.offJavaLangDelegate_thisObj))->contents);
            //Callbacks 对象中回调函数的个数
icount = dvmGetFieldInt((const Object*)Callback_Obj, gDvm.offJavaLangDelegate_count);
            //创建一个 CallbackMethod 对象数组(CallbackMethod 对象用来描述回调函
数)
```

```
                                        ClassObject*        classArrayClass        =
dvmFindArrayClass("[Lelastos/CarDebugger/CallbackMethod;", NULL);
                                                            CallbackObjDesc_Obj    =
dvmAllocArrayByClass(classArrayClass,icount,ALLOC_DEFAULT);
            fieldTag=JT_ARRAY;
           expandBufAdd1(pReply, fieldTag);
           width = dvmDbgGetTagWidth(fieldTag);
           ptr=expandBufAddSpace(pReply,width);
                                                                                jclass
clzCallbackMethod=(*pJNIEnv)->FindClass(pJNIEnv,"elastos/CarDebugger/CallbackMetho
d");
                        jmethodID  methodID_init  =  (*pJNIEnv)->GetMethodID(pJNIEnv,
clzCallbackMethod, "<init>", "()V");
                    jmethodID  methodID_setClazz  =  (*pJNIEnv)->GetMethodID(pJNIEnv,
clzCallbackMethod, "setClazz", "(Ljava/lang/String;)V");
                    jmethodID  methodID_setName  =  (*pJNIEnv)->GetMethodID(pJNIEnv,
clzCallbackMethod, "setName", "(Ljava/lang/String;)V");
           Object* CallbackMethod_Obj = NULL;
            jstring jstr = NULL;   //遍历 Callbacks 对象中的数组,取得每一个回调函数的
信息,并放入一个 CallbackMethod 对象
           int i;
           for (i = 0; i < icount; i++)
{
               Object* obj = (Object*)(thisObjs[i]);
                   if (CallbackMethods[i] == 0) {
                   if (obj != 0) {
                                    CallbackMethod_Obj                              =
(*pJNIEnv)->NewObject(pJNIEnv,clzCallbackMethod,methodID_init);
jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, obj->clazz->descriptor);
  (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,methodID_setClazz,
jstr);
                        char buf[256];
  jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, strcat(strcpy(buf, "On"), field->name));
                        (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,
```

```
methodID_setName,jstr);
                    } else {   //obj == 0
                                                                CallbackMethod_Obj   =
(*pJNIEnv)->NewObject(pJNIEnv,clzCallbackMethod,methodID_init);
 jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, Callback_Obj->clazz->descriptor);
                        (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,
methodID_setClazz,jstr);
                       char buf[256];
                        jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, strcat(strcpy(buf, "On"),
field->name));
                        (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,
methodID_setName,jstr);
                    }
                } else {   //CallbackMethods[i] != 0
                    if (obj != 0) {
                                    CallbackMethod_Obj                              =
(*pJNIEnv)->NewObject(pJNIEnv,clzCallbackMethod,methodID_init);
 jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, obj->clazz->descriptor);
  (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,methodID_setClazz,j
str);
jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, ((Method*)CallbackMethods[i])->name);
                        (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,
methodID_setName,jstr);
                    } else {   //obj == 0
                                                    CallbackMethod_Obj                  =
(*pJNIEnv)->NewObject(pJNIEnv,clzCallbackMethod,methodID_init);
                                                jstr  =  (*pJNIEnv)->NewStringUTF(pJNIEnv,
((Method*)CallbackMethods[i])->clazz->descriptor);
    (*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,methodID_setClaz
z,jstr);
 jstr = (*pJNIEnv)->NewStringUTF(pJNIEnv, ((Method*)CallbackMethods[i])->name);
(*pJNIEnv)->CallVoidMethod(pJNIEnv,(jobject)CallbackMethod_Obj,methodID_setName,j
str);
                    }
```

```
                }
                CallbackObjDesc_Obj->contents[i] = CallbackMethod_Obj;
            }
            dvmSetObjectId(ptr, (ObjectId)CallbackObjDesc_Obj);
            return ERR_NONE;
        } else if (strcmp(fieldClazz->descriptor,"Ljava/lang/Delegates;")==0 || \
                                                    (fieldClazz->super!=NULL
&strcmp(fieldClazz->super->descriptor,"Ljava/lang/Delegates;")==0)) {
                    }
    }
```

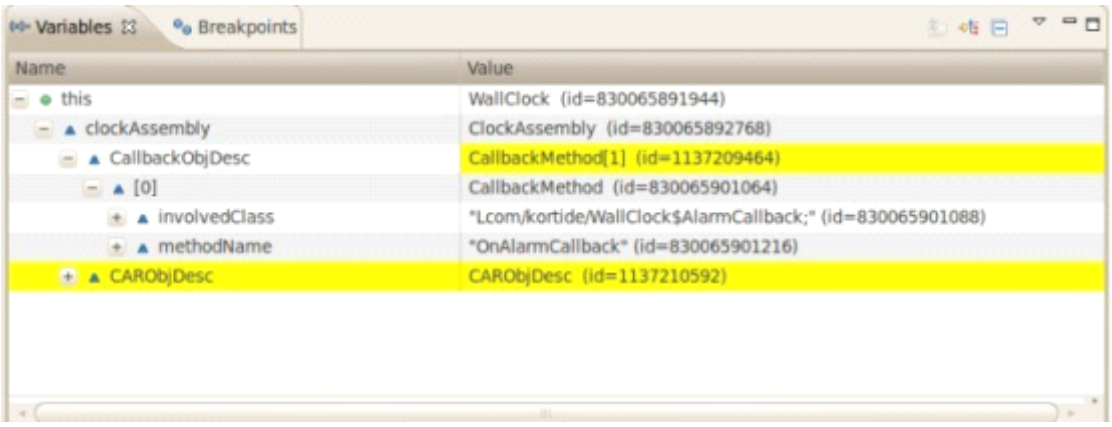

图 5.2 描述注册到 java 中的 callback 对象的快照

图 5.2 的 CallbackObjDesc 对象快照，包括 CallbackObjDesc 对象中的方法（CallbackMethod），涉及到的类（WallClock$AlarmCallback），方法名（onAlarmCallback）都做了相应的快照信息的描述。

5.3 C/C++调试器的介入

5.3.1 gdb 介绍

Linux 包含了一个叫 gdb 的 GNU 调试程序。gdb 是一个用来调试 C 和 C++ 程序的强力调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况. 以下是 gdb 所提供的一些功能:

- 设置断点；
- 监视程序变量的值；

- 程序的单步执行；
- 修改变量的值。

在命令行上键入 gdb 并按回车键就可以运行 gdb 了, gdb 将被启动并且你将在屏幕上看到类似的内容:

```
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to
change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

在可以使用 gdb 调试程序之前，必须使用 -g 选项编译源文件。也就是说，写了下面的一个程序，名字命名为 test.c++, 编译时使用如下语句：

#g++ -o test -g test.c++

#./test

test.c++ 源程序如下：

```
#include <iostream.h>
static char buff [256];
static char* string;
int main ()
{
printf ("Please input a string: ");
gets (string);
printf (" Your string is: %s ", string);
}
```

运行后，程序出现了错误，就可以使用 gdb 来查错了，方法如下：

```
#gdb test
```

(gdb)run 运行 test（二进制）程序，如果在编译时没有加上参数-g 选项，也可以通过如下语句来达到相同的效果

```
#gdb
(gdb)file test
```

如果想查看源程序的部分代码，可以用 list 命令来实现，如：

```
（gdb）list
```

可在 makefile 中如下定义 CFLAGS 变量：

```
CFLAGS = -g
```

运行 gdb 调试程序时通常使用如下的命令：

```
gdb progname
```

在 gdb 提示符下按回车健将重复上一个命令.

表 5.1 gdb 命令描述

命 令	描 述
file FILE	装入想要调试的可执行文件.
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
q(quit)	终止 gdb
watch	使你能监视一个变量的值而不管它何时被改变
make	使你能不退出 gdb 就可以重新产生可执行文件
shell	使你能不离开 gdb 就执行 UNIX shell 命令
break Num	在指定的行上设置断点
bt	显示所有的调用栈帧。该命令可用来显示函数的调用顺序
clear	删除设置在特定源文件、特定行上的断点。其用法为：clear FILENAME:NUM
continue	继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行时
display EXPR	每次程序停止后显示表达式的值。表达式由程序定义的变量组成。
help NAME	显示指定命令的帮助信息
info break	显示当前断点清单，包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示所有的函数名称
info local	显示当函数中的局部变量信息
info prog	显示被调试程序的执行状态
info var	显示所有的全局和静态变量名称
make	在不退出 gdb 的情况下运行 make 工具

print EXPR	显示表达式 EXPR 的值

5.3.2 运行 gcc/egcs

GCC 是 GNU 的 C 和 C++编译器。实际上，GCC 能够编译三种语言：C、C++ 和 Object C（C 语言的一种面向对象扩展）。利用 GCC 命令可同时编译并连接 C 和 C++ 源程序。

如果有两个或少数几个 C 源文件，也可以方便地利用 GCC 编译、连接并生成可执行文件。

GCC 可同时用来编译 C 程序和 C++ 程序。一般来说，C 编译器通过源文件的后缀名来判断是 C 程序还是 C++ 程序。在 Linux 中，C 源文件的后缀名为 .c，而 C++ 源文件的后缀名为 .C 或 .cpp。但是，GCC 命令只能编译 C++ 源文件，而不能自动和 C++ 程序使用的库连接。因此，通常使用 g++命令来完完成 C++ 程序的编译和连接，该程序会自动调用 GCC 实现编译。

表 5.2 gcc/egcs 的主要选项

gcc 命令的常用选项选项	解释
-ansi	只支持 ANSI 标准的 C 语法。这一选项将禁止 GNU C 的某些特色，例如 asm 或 typeof 关键词
-c	只编译并生成目标文件
-DMACRO	以字符串“1”定义 MACRO 宏
-DMACRO=DEFN	以字符串“DEFN”定义 MACRO 宏
-E	只运行 C 预编译器
-g	生成调试信息，GNU 调试器可利用该信息
-IDIRECTORY	指定额外的头文件搜索路径 DIRECTORY
-LDIRECTORY	指定额外的函数库搜索路径 DIRECTORY
-lLIBRARY	连接时搜索指定的函数库 LIBRARY
-m486	针对 486 进行代码优化
-o FILE	生成指定的输出文件。用在生成可执行文件时
-O0	不进行优化处理
-O	或 -O1 优化生成代码

-O2	进一步优化
-O3	比 -O2 更进一步优化，包括 inline 函数
-shared	生成共享目标文件。通常用在建立共享库时
-static	禁止使用共享连接
-UMACRO	取消对 MACRO 宏的定义
-w	不生成任何警告信息
-Wall	生成所有警告信息

第 6 章 部分实现与应用

本章介绍的是在实际应用中的具体实现情况，就具体在开发环境中如何实现，首先介绍的是 android 中 Dalvik 的具体运作，接着介绍 Dalvik 如何运行 Java&CAR 的复合对象，接着在 Eclipse 的 debugger 环境下，查看如何调试 Java&CAR 复合对象，如何获得 CAR 构件的快照。

6.1 Android 中的 Dalvik

每一个 Android 应用在底层都会对应一个独立的 Dalvik 虚拟机实例，其代码在虚拟机的解释下得以执行。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。

应用程序包（APK）被发布到手机上后，运行前会对其中的 DEX 文件进行优化，优化后的文件被保存到缓存区域（优化后的格式被称为 DEY），虚拟机会直接执行该文件。如果应用包文件不发生变化，DEY 文件不会被重新生成。如图 6.1 所示。

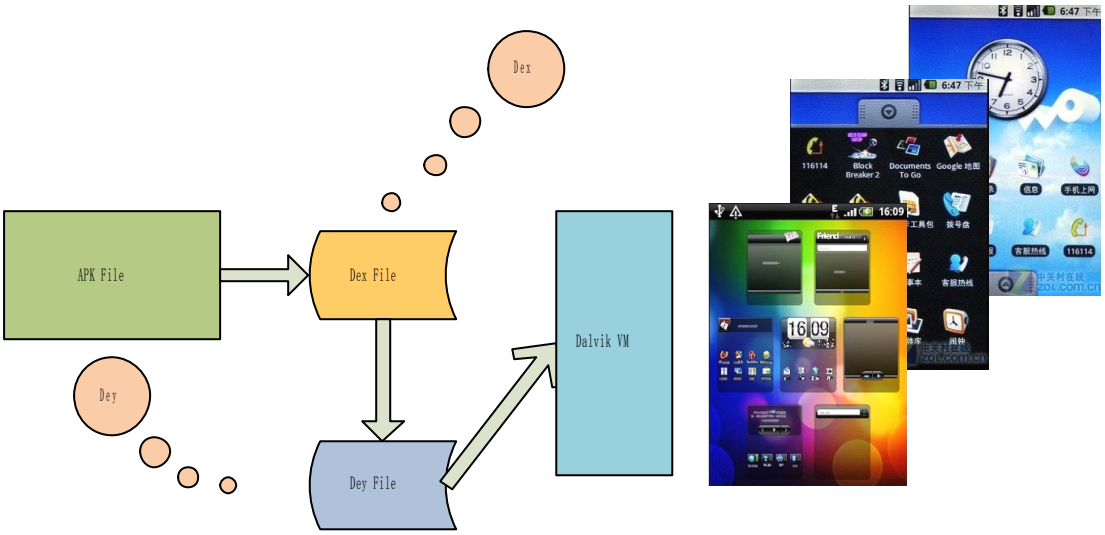

图 6.1 DEX 优化

Android 应用开发和 Dalvik 虚拟机 Android 应用所使用的编程语言是 Java 语言，和 Java SE 一样，编译时使用 Sun JDK 将 Java 源程序编程成标准的 Java 字节码文件（.class 文件），而后通过工具软件 DX 把所有的字节码文件转成 DEX 文件（classes.dex）。最后使用 Android 打包工具（aapt）将 DEX 文件，资源文件以及 AndroidManifest.xml 文件（二进制格式）组合成一个应用程序包（APK）。应用程序包可以被发布到手机上运行。如图 6.2 所示。

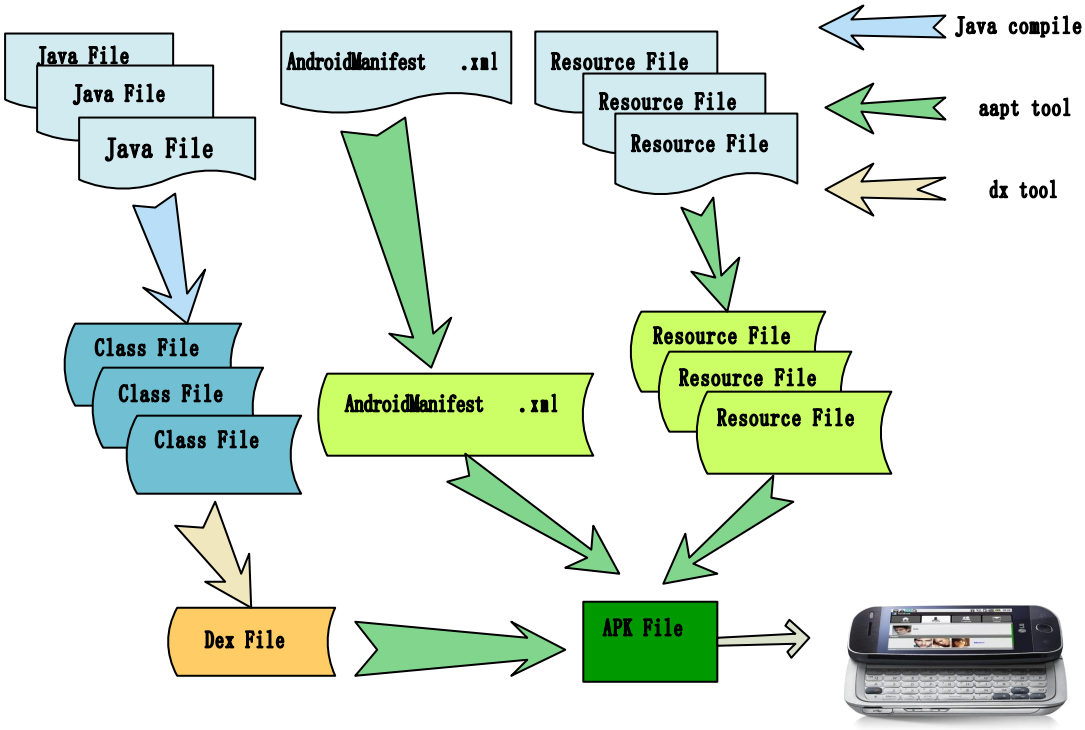

图 6.2 APK 打包过程

6.1.1 Dalvik中运行 Java&CAR

本具体应用是在 Android 模拟器上跑一个 CallbackJDWP 的应用，首先在 Eclipse 创建一个基本的 Android 应用程序，称为 CallbackJDWP。创建这个应用程序之后，将调试和运行它，其中应用程序的项目名为 CallbackDebug。

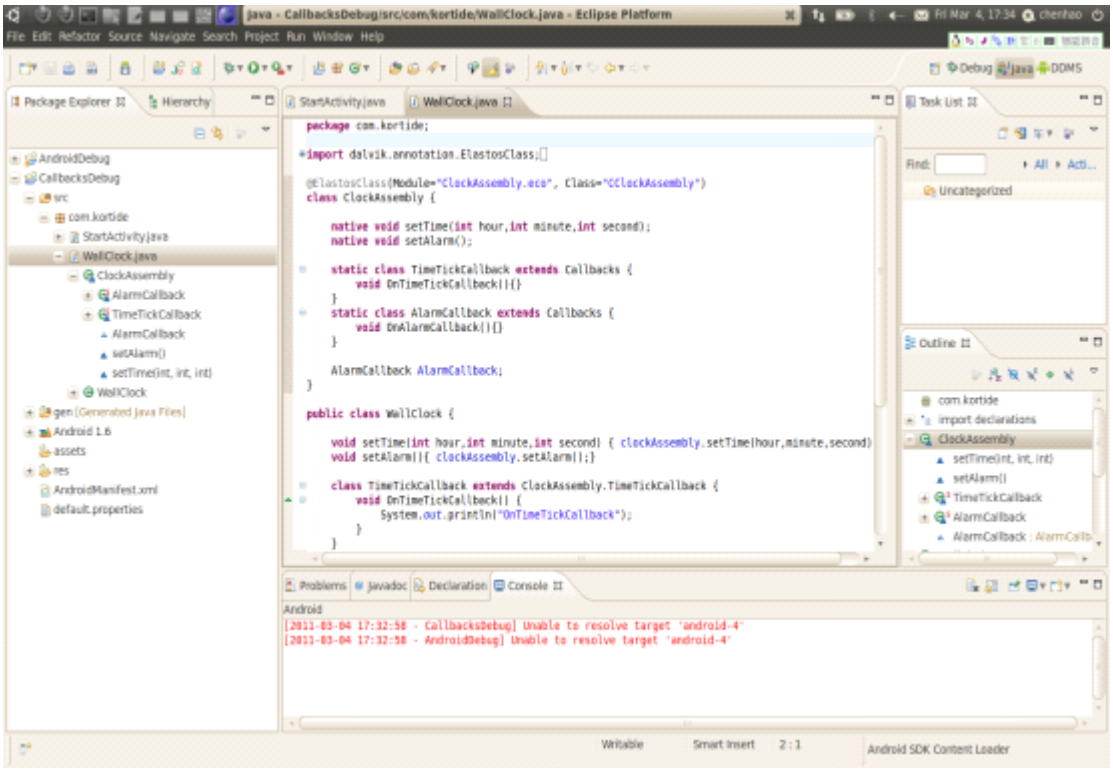

图 6.3 在 Eclipse 中的 CallbackDebug 的各透视窗口

具体代码：

```
package com.kortide;


import dalvik.annotation.ElastosClass;
import java.lang.Callbacks;


@ElastosClass(Module="ClockAssembly.eco", Class="CClockAssembly")
class ClockAssembly {


    native void setTime(int hour,int minute,int second);
    native void setAlarm();
```

```
    static class TimeTickCallback extends Callbacks {
        void OnTimeTickCallback(){}
    }
    static class AlarmCallback extends Callbacks {
        void OnAlarmCallback(){}
    }

    AlarmCallback AlarmCallback;
}

public class WallClock {

    void setTime(int hour,int minute,int second)
{ clockAssembly.setTime(hour,minute,second); }
    void setAlarm(){ clockAssembly.setAlarm();}

    class TimeTickCallback extends ClockAssembly.TimeTickCallback {
        void OnTimeTickCallback() {
            System.out.println("OnTimeTickCallback");
        }
    }
    class AlarmCallback extends ClockAssembly.AlarmCallback {
        void OnAlarmCallback() {
            System.out.println("OnAlarmCallback");
        }
    }

    WallClock(){
        clockAssembly = new ClockAssembly();

        clockAssembly.AlarmCallback = new AlarmCallback();
    }

    ClockAssembly clockAssembly;
```

```
}
```

在这个源代码片段中，这里我们自定义了一个 Java 的 annotation 类@ ElastosClass 来标识这个 Java 类是一个 CAR 构件对应的类。这个 annotation 类具有两个属性 Module 和 Class，分别指定 CAR 构件的名称和 CAR 类的名称。我们在根据 CAR 构件构造 Java 类时，使用@ ElastosClass 来修饰 Java 类，并指明其对应的 CAR 构件的名称和 CAR 类的名称。有了这个属性，在虚拟机加载之时就可以知道这个类对应的 CAR 构件和类了。

应用程序成功地编译了，运行这个示例应用程序了。在 Eclipse 中选择 Open Run Dialog 或工具栏上的快捷按钮。打开一个对话框，可以在这里创建启动配置。选择 Android Application 选项并单击 New 的图标。指定配置名称。并选择 Emulator 选项卡，根据需要指定模拟器设置。可以保持默认设置。选择 Run 运行示例应用程序。如图 6.4 所示

图 6. 4 在 android 模拟器应用列表中的 CallbackJDWP

图 6.5 运行 CallbackJDWP 应用

6.2 Eclipse 调试 Java&CAR 程序

在 Eclipse 中，选择 Window > Open Perspective > Other。在出现的对话框中选择 Debug。这会在 Eclipse 中打开一个新的透视图。

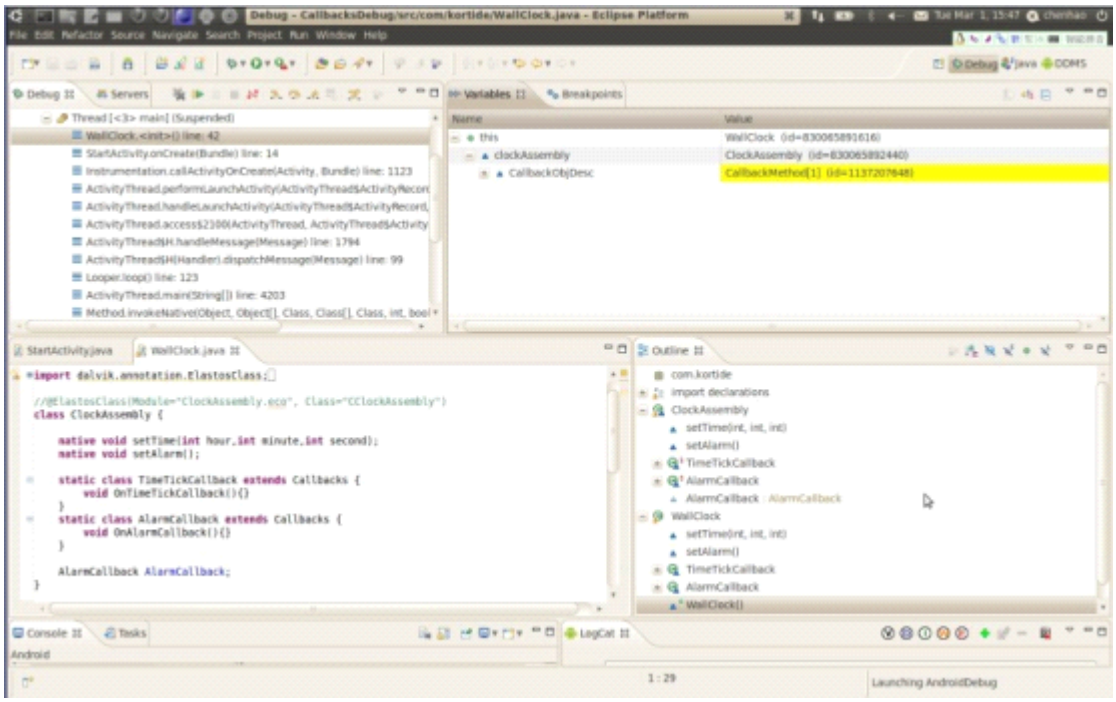

图 6.6 仅是 Java 程序的调试信息

图 6.6 显示的仅是 Java 程序的调试信息，在该图中并无 CAR 构件的快照信息。

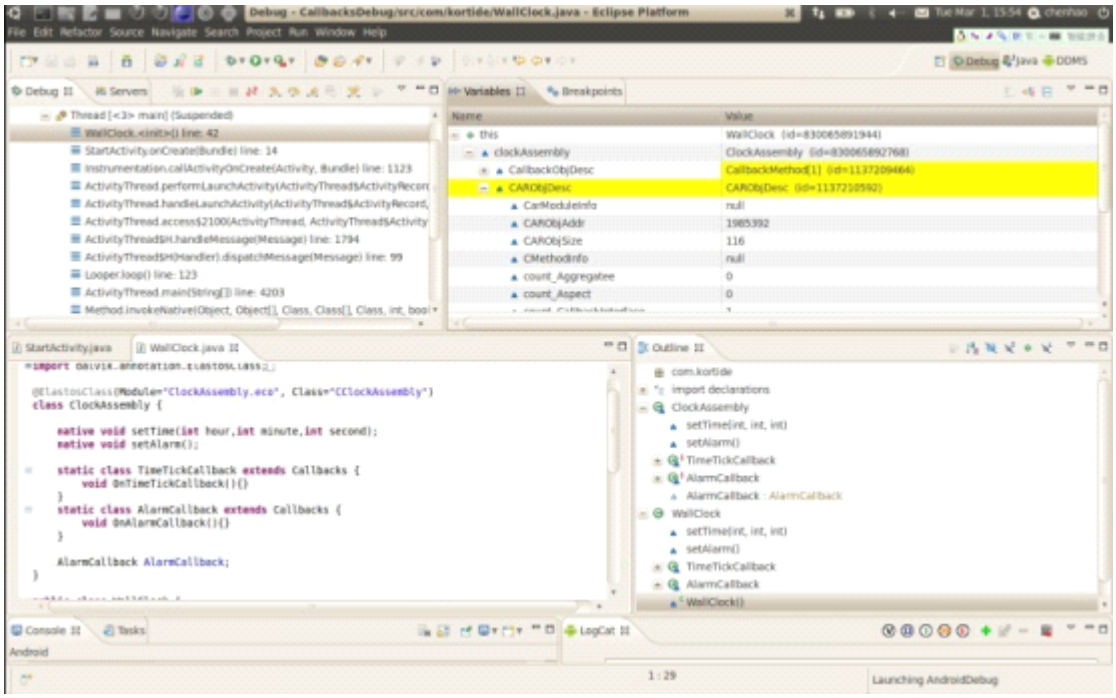

图 6.7 Eclipse 下 CAR 构件的快照信息

图 6.7 显示的是通过对 JPDA 端的实现，Eclipse 的调试窗口下，可以看到 CARObjDesc 对象的快照信息，包括该对象的地址位置，对象中的方法等，各种扇入扇出的方法

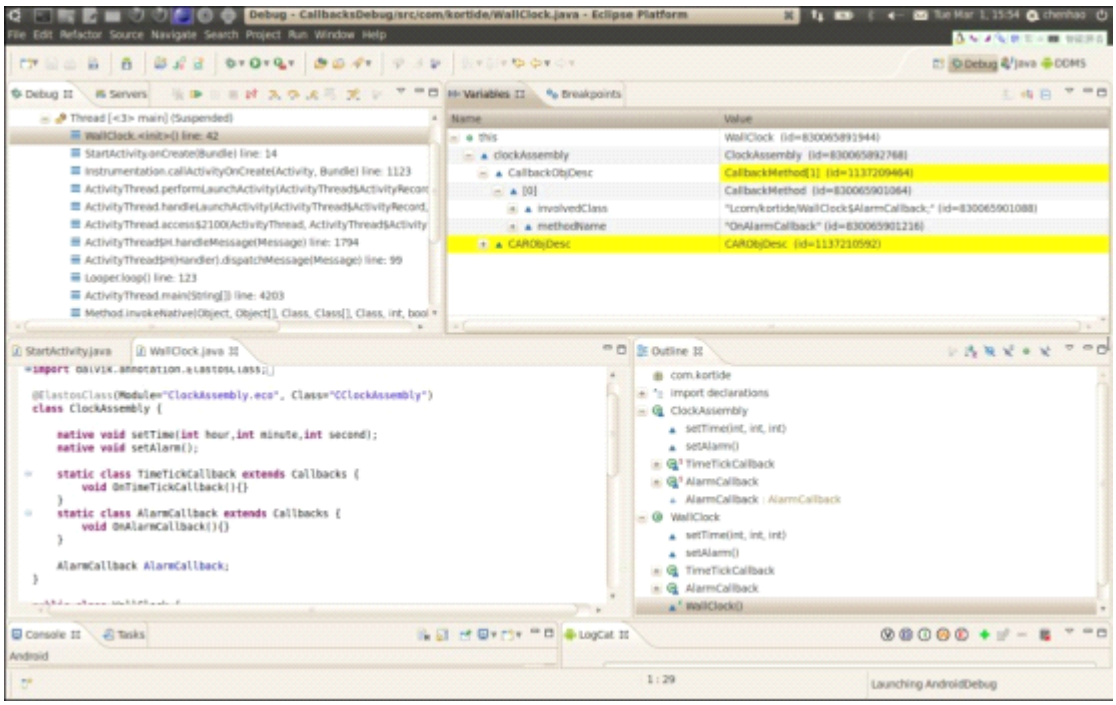

图 6.8 CAR 构件调用注册到 Java 中回调函数的快照信息

图 6.8 显示的是 CAR 构件调用中的某一事件被触发，并该触发的事件需要使用注册到 Java 中回调函数的情况，这是 Eclipse 的调试窗口下，可以通过 Java 调试协议 JDWP 返回该回调函数的快照信息。

第 7 章 总结

7.1 工作总结

本文基于 Elastos 操作系统，提出了复合对象调试模型的概念，为应用程序的调制提出了一种新的调试模型，同时也给当前基于单一对象的调试模型提出了一种解决方法，简化了应用程序复合对象的调试，提供了了更为合理调试方式。本文讨论了实现本文提出一种快照与 C/C++调试相结合的调试技术，对 CAR 运行环境，运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议把信息传递给调试器，在进入 CAR 构件前插入单步中断，从而让 C/C++调试器得到介入的机会，实现 CAR 构件的二进制调试。构件中的关键技术，给出了其部分实现，

这些工作主要包括：

1 结合 Java 与 CAR 构件技术，人们提出了 Java 与 CAR 混合编程技术，通过把 Java 对象与 CAR 对象复合，提供一个逻辑对象的两个操作面：Java 语言面、C++语言面。

2.提出一种快照与 C/C++调试相结合的复合对象调试技术，对 CAR 运行环境，运行上下文及 CAR 构件本身进行快照，通过 Java 的调试协议 JDWP 把调试信息传递给调试器（如 Eclipse），实现 JAVA 语言级调试。在进入 CAR 构件前插入单步中断，从而让 C/C++调试器得到介入的机会，实现 CAR 构件的二进制调试。

3.利用 JAVA 自身的调试体系 JPDA，通过 JPDA 中的 JDWP 协议，能在 JAVA 程序的 IDE 环境下查看和调试 JAVA 虚拟机上运行 CAR 构件详细情况，包括各种接口的扇入扇出、参数、各种回调函数。

4.当 CAR 构件中注册了 Java 的回调函数，当触发 CAR 构件中的事件，应用会调用 Java 中的回调函数，在 JPDA 端实现该回调函数的快照的详细信息。

基于上面的部分实现，如果要给出一个切实可用的 JAVA&CAR 的复合对象调试机制，还有很多后续工作要做：

1. 线程调试的实现,涉及到动态控制虚拟机的都还没实现,如单线程，多线程的实现部分。

2. JDWP 的调试协议中的事件处理机制还未在复合对象调试技术中实现

7.2 工作展望

在软件开发环境中，调试器是软件开发必不可少的工具，通用计算机，无论是巨型机，大型机，工作站还是 PC 机，都配置有适合系统特点的调试工具。在嵌入式系统软件开发环境中，调试尤显其重要性。

软件工程技术的发展经历了结构化开发和面向对象开发的两个阶段。对于不同的开发阶段，软件调试技术也随着软件工程技术的发展不断更新，结构化调试和面向对象调试技术也相继产生。随着复合对象这一新概念的提出，调试技术的局限性也逐渐显示出来，如一般只能调试一种语言，调试技术具有语言单一性，不能跨语言调试。本文提出的这种基于 Java&CAR 的调试模型，在一个范围内解决了 JAVA 与 C/C++之间的基于 Dalvik 的调试模型，能在复合类对象中兼顾两种语言对象的调试，但对于复合对象的调试技术的研究和实现，，仅是最基本的出发点，只是实现了运行基本的静态显示调试信息，而没有完全实现静动态的调试技术。因此，功能不够完善，仍需后续工作来完成。比如尽实现复合对象的动态调试，包括线程，事件的具体调试仍需后续工作。

致谢

衷心导师感谢陈榕教授，他对于计算机科学的深刻见解和严谨务实的态度将使我终生受益。

感谢顾伟楠老师，他对我的论文写作进行了细心的指导，提出了很多宝贵的意见。

在科泰世纪实习的一年多时间中，裴喜龙老师给了我热情的指导和帮助，他教给我很多学习，工作和为人的智慧，不胜感激。

感谢科泰世纪的同事，他们给了我很多的帮助，让我慢慢熟悉了公司的软件开发环境，了解软件开发的流程。

感谢徐凡，陈俞飞，周平东等师兄，通过和他们的交流，我学到的不少东西。

感谢袁轶，陈为伍，王建民，申波，李代立，王保卫，谢立丹同门，通过他们的帮助与交流，让我在专业学习上有了质的提高。

感谢师弟陈灏，在我的专业论文上给了莫大的帮助，以至于论文能顺利有效的完成。

参考文献

[1] Robert Charles Metzger.Debugging by Thinking 电子工业出版社,2004

[2]The Dalvik Virtual Machine Open Source Project. http://www.dalvikvm.com/

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

[4]Kortide.Website.http://www.kortide.com.cn

[5]Specification of Java Debug Wire Protocol

[6]Specification of Java Debug Wire Protocol Interface

[7] Google Inc. Android Documentation. http://code.google.com/intl/en/android/Documentation.html

[8] Chien-Wei Chang, Chun-Yu Lin, Chung-Ta King, Yi-Fan Chung, Shau-Yin Tseng Implementation of JVM tool interface on Dalvik virtual machine VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on

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

[10] Yunhe Shi, David Gregg, Andrew Beatty. Virtual Machine Showdown: Stack Versus Registers, ACM Transactions on Architecture and Code Optimization, 2008, Vol.4(2):154~163

[11] http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html/

[12] Bhupinder S. Mongia , Vijay K. Madisetti Reliable Real-Time Applications on Android OS 2010

[13] Microtsoft Corporation and Digital Equipment Corporation.The Component Object Model Specification.Draft version 0.9. http://www.microsoft.com/com,1995.

[14] 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

[15] Weiquan Zhao, Jian Chen. CoOWA: A Component Oriented Web Application Model. The 31st International Conference on Technology of Object-Oriented Language and Systems, Sep 1999

[16] 虚拟机. http://zh.wikipedia.org/wiki/虚擬機器

17] 陈为伍，王建民，陈榕 Dalvik 在 CAR 构件运行时的研究 2010 年 12 月

[18] 周毅敏;陈榕;;Dalvik 虚拟机进程模型分析[J];计算机技术与发展;2010 年 02 期

[19]

[20] 上海科泰世纪科技有限公司．Elastos 智能手机软件整体解决方案白皮书．2006

[21] 张银奎.软件调试. 电子工业出版社,2008

[22] 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

[23] 周毅敏，陈榕.Java 虚拟机的移植与基于 CAR 构件的二次开发 2010

[24]毛德操，胡希明．Linux 内核源代码情景分析．浙江大学出版社，2001

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

[26] 王开谭．Java 中反射机制浅析及应用. 电脑知识及技术, 2007．Vol．1(1)：265-266

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

[28] 陈闳中．Linux 在嵌入式操作系统中的应用［J］. 同济大学学报，2001 05(014) : 564-566

[29] 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, 2003.
[30] 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
[31] 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)：
[32] 郑炜，陈榕，苏翼鹏，殷人昆. CAR 构件编程技术中的自描述特性. 计算机工程与应用. 2005.09
[33] 郑坤. “和欣”操作系统跨平台虚拟机研究与实现. 硕士毕业论文. 2007
[34] 陆刚. Linux 平台上和欣虚拟机的研究与实现. 硕士毕业论文. 2008
[35]洪岷生等. C 语言及其开发工具:调试器及运行库.上下册厦门大学出版社,1998.5
[36]上海科泰世纪科技有限公司．Elastos CAR 构件与编程模型技术文档．2006

个人简历、在读期间发表的学术论文与研究成果

个人简历：

盛泽昀，女，1985 年 8 月生。

2004 年 7 月毕业重庆师范大学 电子信息科学与技术专业 获学士学位。

2008 年 9 月入同济大学读硕士研究生。

已发表论文：

[1] 盛泽昀. 基于移动设备的卡片式应用程序界面开发. 电脑知识与技术，2010-10 第 28 期

……