

Elanix 项目综合报告（一）

Elanix 中的二进制跨平台构件加载 与运行机制研究

作 者	高 靖
相关人员	陈志成 苏杭 高靖 杨昕 石磊
部 门	Elanix 开发小组
时 间	2006-3-16
版 本	1.0.0.0
修改记录	说明：本综合报告主要根据参与 Elanix 项目的高靖的硕士研究生论文整理而成。

目 录

目 录	I
第 1 章 引 言	5
1.1 课题目的和意义	5
1.2 构件跨平台技术研究现状	5
1.3 报告的组织结构	8
第 2 章 相关技术	10
2.1 “和欣”操作系统	10
2.1.1 “和欣”操作系统概述	10
2.1.2 “和欣”操作系统特点	11
2.2 CAR构件技术及运行平台	11
2.2.1 CAR构件	11
2.2.2 “和欣”CAR构件运行平台	12
2.3 Wine虚拟机	14
2.3.1 Wine加载模型	14
2.3.2 Wine特点分析	16
2.4 可执行文件分析：PE和ELF	17
2.4.1 PE文件格式	17
2.4.2 ELF文件格式	18
2.4.3 PE与ELF文件格式比较	19
2.5 元数据	20
2.5.1 元数据定义	20
2.5.2 CAR构件元数据	21
第 3 章 Elanix结构设计 with 实现	23
3.1 引论	23
3.2 Elanix —— 开发环境建立 with 维护	23
3.3 Elanix系统结构	28
3.3.1 Elanix系统结构	28
3.3.2 Elanix实现思路	29

第 4 章 Elanix加载器	30
4.1 Elanix加载器设计原理	30
4.1.1 加载器理论依据	30
4.1.2 运行模式的转变	34
4.1.3 Elanix加载器设计	35
4.1.4 加载器的地址空间分配方式	36
4.1.5 Elanix地址空间分布	40
4.1.6 小结	41
4.2 Elanix加载器实现	42
4.2.1 PE格式模块的解析和加载	42
4.2.2 模块加载和选择	43
4.2.3 模块加载方式	45
4.3 Elanix加载器模块管理和地址空间管理	48
4.3.1 模块数据结构定义	48
4.3.2 模块管理	50
4.3.3 地址空间管理	50
4.4 Elanix加载器对图形、网络、文件模块的支持	52
4.4.1 Elastos对外部模块支持	52
4.4.2 本地模块模式	52
4.4.3 内建模块模式	53
4.4.4 Elanix瀑布式模块加载	54
4.5 Elanix加载器加载实例及流程	55
第 5 章 自描述构件加载与管理	57
5.1 构件加载	57
5.2 进程内构件加载过程	58
5.2.1 进程内CAR构件	58
5.2.2 Elanix加载进程内CAR构件	59
5.3 进程外CAR构件互操作	61
5.3.1 Elastos2.0 进程外CAR构件互操作	61
5.3.2 Elanix进程外CAR构件互操作	62
5.3.3 Elanix Server 性能分析	64
5.4 CAR构件元数据解析	65
5.4.1 CAR构件元数据	65
5.4.2 CAR构件元数据解析	67
5.5 CAR构件加载版本管理	68
5.5.1 Elastos2.0 中CAR构件版本管理	68

5.5.2 Elanix中CAR构件版本管理	69
5.6 Elanix中CAR构件加载安全认证	70
5.7 构件加载实例及流程	71
第6章 总结和展望	73
6.1 总结	73
6.2 工作展望	74
参 考 文 献	75

第1章 引言

1.1 课题目的和意义

本课题基于Elastos2.0操作系统^[1]，目的是在Linux上直接加载二进制格式的Elastos2.0应用程序和CAR构件^[2]。同时，在Linux上提供Elastos2.0内核的功能，如构件的跨进程访问、互斥等操作，最终实现将未经修改的二进制格式Elastos2.0应用程序和CAR构件加载到Linux上运行，扩展Elastos2.0应用程序和CAR构件的跨平台特性。

Elastos2.0操作系统和CAR构件技术借鉴了Java虚拟机思想，但没有采取中间代码的形式，而是通过良好的体系架构，为跨平台提供支持。Elastos2.0借助C++语言、机器指令实现了高效率，属于低资源要求的虚拟机，其可以与当前主流的软件技术互补、互操作。

目前，Elastos2.0应用程序及CAR构件已经可以成功的运行在Elastos和Windows操作系统上，这为Elastos2.0应用程序和CAR构件向更多操作系统平台移植打下了坚实的基础。

本课题选择了Linux为Elastos2.0应用程序跨平台运行的研究对象。这是因为，Linux作为Open Software的代表，已经受到越来越多的重视，中国信息产业部已经将开发Linux放到了一个战略的高度。因此，将Elastos2.0应用程序和CAR构件加载到Linux上运行，成为Elastos技术发展的重要任务之一，有了对Linux的支持，也才能更全面的体现Elastos2.0架构跨平台特性。

1.2 构件跨平台技术研究现状

软件系统的复杂性不断增长，为了能地缩短软件产品的开发周期，提高软件产品质量，产生了集软件复用、分布式对象计算、企业级应用开发等技术为一体的基于构件的软件开发(CBSD, Component Based Software Development)。构件实际是些小的二进制可执行程序，他们可以给应用程序、操作系统以及其他构件提供服务。开发出来的构件可以被连接起来形成应用程序或构件系统，并且构件可以在运行时刻、在不重新链接或编译应用程序的情况下被卸载或替换。构件跨平台技术也是如火如荼，许多公司都推出了不同的构件标准，支持不同

的系统平台之间的构件互操作。

目前主流的构件技术包括：微软.net构件^[3]、COM构件^[4]、J2EE的EJB^[5]、Elastos的CAR构件等。尽管构件技术已经在Windows上得到了广泛的应用，但在Linux/Unix系统上，由于没有制定相应的构件规范标准。绝大部分的Linux应用程序仍然采用传统的基于动态链接库的架构。下面介绍几种主流的构件技术及其跨平台特性。

COM 和 .NET

COM 是微软的组件对象模型。这是构造 ActiveX 组件（可以让开发者同样重视其它 COM 组件的能力）和 OLE（对象连接和嵌入）的核心技术。COM 组件来自于微软的组件对象模型。COM 组件可以是 ActiveX 控件 (OCX)、ActiveX DLL 或 In Process Servers (DLL)，也可以是 ActiveX EXE 或 Out of Process Servers (EXE)。实际上，基于已有的功能，任何 COM 组件都可以从适当的环境中获取并使用，例如 Basic，VC++，IIS 等等。

Microsoft.NET 是 Microsoft 基于 XML 的 Web 服务平台。XML Web services 允许应用程序通过 Internet 进行通讯和共享数据，而不管所采用的是哪种操作系统、设备或编程语言。Microsoft .NET 平台并将这些服务集成在一起之所需。.Net 架构采用了由 .Net 平台、XML、SOAP 和 UDDI 组成的四层应用程序开发和运行平台，使得开发多语言、跨平台、可复用的软件成为可能。C# 是 .Net 平台上重点支持的面向构件的新型开发语言。它具有支持构件开发、面向对象、类型安全；能进行版本控制等特点。

EJB

为了推动基于 Java 的服务器端应用开发，Sun 于是在 1999 年底推出了 Java2 技术及相关的 J2EE 规范，J2EE 的目标是：提供平台无关的、可移植的、支持并发访问和安全的，完全基于 Java 的开发服务器端构件的标准。

企业级 JavaBean (EJB) 组件就是在分布式环境中运行的 JavaBean 组件。是 Sun 推出的基于 Java 的服务器端构件规范 J2EE 的一部分。EJB 组件在 Java Application Servers 中运行，适用于分布式的、可扩展的商业逻辑配置。它基于 Java 语言，提供了基于 Java 二进制字节代码的重用方式。JavaBean 组件一般是客户端组件，在 Java 虚拟机上运行。而设计企业级 JavaBean 是为了使其运行在

分布环境中的应用服务器上，因此服从环境的需求、内容和限制。从实现的角度，JavaBean 通常为用户提供界面功能和客户端的商业逻辑。EJB 则提供分布的、中间层的商业逻辑。因此可根据实际情况确定是创建/使用 JavaBean 组件还是 EJB 组件。从分布式计算的角度，EJB 像 CORBA 一样，提供了分布式技术的基础。提供了对象之间的通讯手段。从 Internet 技术应用的角度，EJB 和 Servlet, JSP 一起成为新一代应用服务器的技术标准，EJB 中的 Bean 可以分为会话 Bean 和实体 Bean，前者维护会话，后者处理事务，而 Servlet 负责与客户端通信，访问 EJB，并把结果通过 JSP 产生页面传回客户端。J2EE 的优点是，服务器市场的主流还是大型机和 UNIX 平台，这意味着以 Java 开发构件，能够做到"Write once, run anywhere"，开发的应用可以配置到包括 Windows 平台在内的任何服务器环境中去。

CORBA

CORBA(Common Object Request Broker Architecture)是一组标准，用来定义“分布式对象系统”，由 OMG(Object Management Group)作为发起和标准制定单位。OMG 由 700 多家公司和单位组成，几乎包括了所有有影响的公司。CORBA 的目的是定义一套协议，符合这个协议的对象可以互相交互，不论它们是用什么样的语言写的，不论它们运行于什么样的机器和操作系统。

COBRA 标准主要分为 3 个层次：对象请求代理、公共对象服务和公共设施。最底层是对象请求代理 ORB，规定了分布对象的定义（接口）和语言映射，实现对象间的通讯和互操作，是分布对象系统中的'软总线'；在 ORB 之上定义了很多公共服务，可以提供诸如并发服务、名字服务、事务(交易)服务、安全服务等各种各样的服务；最上层的公共设施则定义了组件框架，提供可直接为业务对象使用的服务，规定业务对象有效协作所需的协定规则。目前，CORBA 兼容的分布计算产品层出不穷，其中有中间件厂商的 ORB 产品，如 BEAM3, IBM Component Broker，有分布对象厂商推出的产品，如 IONAObix 和 OOCObacus 等。

CORBA CCM(CORBA Component Model)技术，是在支持 POA (Portable Object Adapter, 便携式对象适配器)的 CORBA 规范(版本 2.3 以后)基础上，结合 EJB 当前规范的基础上发展起来的。CORBA 构件模型，是 OMG 组织制定的一个用于开发和配置分布式应用的服务器端中间件模型规范，它主要包括如下

三项内容：第一、抽象构件模型，用以描述服务器端构件结构及构件间互操作的结构；第二、构件容器结构，用以提供通用的构件运行和管理环境，并支持对安全、事务、持久状态等系统服务的集成；第三、构件的配置和打包规范，CCM 使用打包技术来管理构件的二进制、多语言版本的可执行代码和配置信息，并制定了构件包的具体内容和基于 XML 的文档内容标准。

总之，CORBA 的特点是大而全，互操作性和开放性非常好。CORBA 的缺点是庞大而复杂，并且技术和标准的更新相对较慢，CORBA 规范从 1.0 升级到 2.0 所花的时间非常短，而再往上的版本的发布就相对十分缓慢了。在具体的应用中使用不是很多。

上面介绍的几种常见的构件技术与标准，反映了当前构件技术发展的趋势。每种技术都有其优缺点，也都有成功应用的范例。Elastos2.0 和 CAR 构件在架构设计和发展上借鉴了各技术的长处的同时，采用中间件技术，融合 SOA（Service-Oriented Architecture，面向服务架构）、AOP（Aspect Oriented Programming，面向方向编程）等新特色，这为 Elastos2.0 成为面向下一代的高性能网络操作系统奠定了基础。

1.3 报告的组织结构

本文从构件跨平台研究的技术角度入手，分析了近年来流行的构件跨平台技术，介绍了 Elastos 虚拟机跨平台的机制，其上的 CAR 构件技术，以及与 Elastos 和 Linux 相关的技术背景。在此基础之上，引入 Elanix——Linux 上的 Elastos2.0 虚拟操作系统。论文详细介绍了 Elanix 环境的建立，Elanix 加载器，以及 CAR 构件在 Elanix 上的加载等重要技术的实现。

论文第二章介绍了 WINE^[6] 虚拟机的实现机制和特点，分析了 WINE 虚拟机的模型和体系架构。指出了 WINE 作为跨平台技术方案的优势和其局限性。分析了 Elastos 上可执行文件格式（PE 格式），Linux 上可执行文件格式（ELF 文件），并对两种文件格式进行了比较。还介绍了元数据的定义，以及 Elastos2.0 上元数据的使用和其所包含的内容，这是整篇文章的技术基础。

论文第三章介绍了 Elanix 的系统结构，其内容包括：Elanix 开发环境的建立和维护方式、Elanix 系统架构和 Elanix 的实现思路。

论文第四章重点介绍了 Elanix 加载器的实现，包括：加载器的设计原理、

加载器是如何实现 PE 文件在 Linux 上加载的、加载器对于加载模块的管理和对地址空间的管理、加载器对外部模块的支持、例举了在 Elanix 上加载 socket 通信的 Elastos2.0 应用程序。

第五章介绍了 Elanix 对 CAR 构件加载的技术实现，内容有：Elanix 上构件加载的模式、针对进程内构件和进程外构件加载进行了分析、还分析了 CAR 构件在加载过程中的版本控制、安全认证等问题、最后例举了 CAR 构件加载的实例。

文章最后进行了总结，同时也对下阶段 Elanix 项目的所面临的技术挑战和发展目标进行了阐述。

第2章 相关技术

2.1 “和欣”操作系统

2.1.1 “和欣”操作系统概述

“和欣”（英文名：Elastos）32 位嵌入式实时操作系统就是由科泰世纪科技有限公司历时两年半开发出来的，其目的是为应用软件提供基础平台。“和欣”操作系统集成了第三代网络的核心技术，为 WEB 服务在各种嵌入式设备上的实现提供了技术平台。32 位嵌入式操作系统。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。“和欣”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。

完全面向构件技术的操作系统。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。如图 2.1 所示。

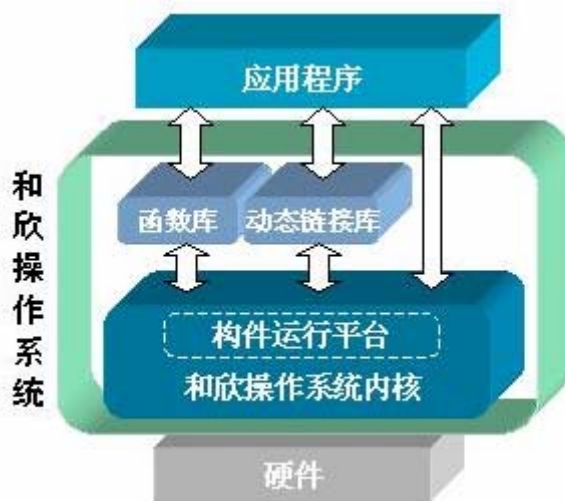


图 2.1 “和欣”操作系统架构图

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。

2.1.2 “和欣”操作系统特点

“和欣”操作系统的最大特点就是：

- (1) 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；
- (2) 用构件技术实现了“灵活”的操作系统；

这是“和欣”操作系统区别于其他嵌入式操作系统产品的最大优势。

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。

“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的技术。

2.2 CAR 构件技术及运行平台

2.2.1 CAR 构件

CAR (Component Assembly Runtime) 构件技术是面向构件编程的编程模型，它规定了一组构件间相互调用的标准，使得二进制构件能够自描述，能够在运行时动态链接。

CAR 兼容微软的 COM。但是和微软 COM 相比，CAR 删除了 COM 中过时的约定，禁止用户定义 COM 的非自描述接口；完备了构件及其接口的自描述功能，实现了对 COM 的扩展；对 COM 的用户界面进行了简化包装，易学易用。

从上面的定义中，我们可以说 CAR 是微软 COM 的一个子集，同时又对微软的 COM 进行了扩展，在“和欣”SDK 工具的支持下，使得高深难懂的构件编程技术很容易被 C/C++ 程序员理解并掌握。CAR 中的“ez”源自与英文单词“easy”，恰如其分地反映了这一特点。

CAR 构件的编程思想是“和欣”技术的精髓，它贯穿于整个技术体系的实现中。

2.2.2 “和欣”CAR 构件运行平台

“和欣”CAR 构件运行平台提供了一套符合 CAR 规范的系统服务构件及支持构件相关编程的 API 函数，实现并支持系统构件及用户构件相互调用的机制，为 CAR 构件提供了编程运行环境。“和欣”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“和欣”操作系统中，“和欣”构件运行平台与“和欣灵活内核”共同构成了完整的操作系统。

在 Windows 2000、WinCE、Linux 等其他操作系统上，“和欣”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“和欣”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。

图 2.2 直观地显示了“和欣”构件运行平台在 Windows 2000/XP、“和欣”操作系统中的位置。

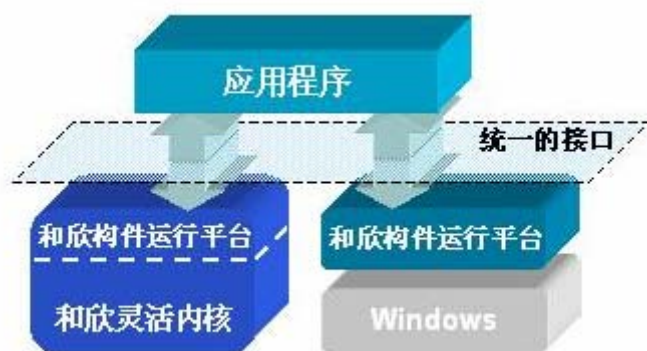


图 2.2 “和欣”构件运行平台

从“和欣”构件运行平台的定义，我们知道该平台为 CAR 提供了运行环境。

从这个意义上，我们说的 CAR 技术也可以理解为在运行环境中对 CAR 规范提供支持的程序集合。

从编程的角度看，“和欣”构件运行平台提供了一套系统服务构件及系统 API（应用程序编程接口），这些是在该平台上开发应用程序的基础。

“和欣”操作系统提供的其他构件库也是通过这些系统服务构件及系统 API 实现的。系统提供的这些构件库为应用编程开发提供了方便，主要有：

图形系统构件库；

设备驱动构件库；

文件系统构件库；

网络系统构件库。

从“和欣”构件运行平台来看，这些构件和应用程序的构件是处于同样的地位。用户可以开发性能更好或者更符合需求的文件系统、网络系统等构件库，替换原来系统提供的构件库，也可以开发并建立自己的应用程序构件库。图 2.3 显示出“和欣”构件运行平台的功能及其与构件库、应用程序的关系。

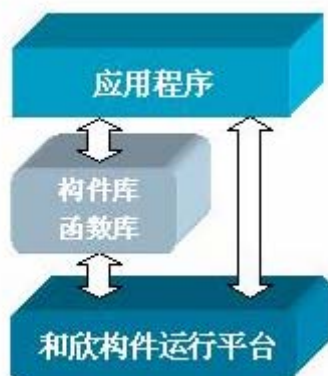


图 2.3 “和欣”构件平台与构件库和应用程序的关系

总之，构件运行平台为 CAR 构件提供了对程序员完全透明的运行环境，构件可以运行在不同地址空间，不同环境，甚至跨网络。构件运行平台自动为构件运行提供支持，配置必要的网络协议、针对不同的输入输出设备的协议。程序员不必过多地去关心诸如网络协议转换及构件运行控制等与其他构件互操作时的协调问题，只需专注于自己需要解决的程序算法的实现。从而可以从繁杂庞大的应用环境体系中解放出来，大大提高编程的效率。

“和欣”构件运行平台直接运行二进制构件，而不是像 JAVA 和 .NET 那样

通过虚拟机在运行程序时解释执行中间代码。因此，与其他面向构件编程的系统相比，具有资源消耗小，运行效率高的优点。

2.3 Wine 虚拟机

2.3.1 Wine 加载模型

Wine 是 “Wine Is Not an Emulator” 的缩写，这意味着，Wine 并不是一个虚拟机，即它不是通过模拟 CPU 来工作的。Wine 的目的是通过软件模拟的方式，将 Windows 的 API 以 Linux 系统调用实现。这样，没有 Windows 的系统支持，Windows 应用程序仍然可以加载到 Linux 上运行。是一个在 X 和 UNIX 之上的，Windows 3.x 和 Windows APIs 的实现。它是一个 Windows 兼容层，用通俗的话说，就是一个 Windows 模拟器，这个层即提供了一个用来从 Windows 源进出到 UNIX 的开发工具包(Winelib)，也提供了一个程序加载器，该加载器允许不用任何修改 Windows 3.1/95/NT 的二进制文件，就可以运行在 Intel Unix 及其衍生版本下。

Wine 的体系架构如图 2.4。其包括三个部分：

第一部分：Windows 应用程序或 DLL，即图 2.4 中上面白色的部分，这些程序是未经修改的 Windows 可执行文件，其文件格式为 PE 格式。

第二部分：Wine 的实现部分，即图 2.4 中间灰色的部分，其也是 Windows 程序在 Linux 上运行的 Wine 支持层，正是这部分的存在，Windows 程序才能在 Linux 上完成与 Windows 上相同的功能和操作。从图中可以看出，Wine 在 Linux 上实现了 Windows 程序必须依赖的三个 DLL：USER32.DLL，GUI.DLL，KERNEL.DLL。在这部分的左侧，还有 Wine Server，其主要是用来封装了 Windows 内核的功能，如进程间通信，同步，进程/线程管理等。Wine Server 是个独立的 Linux 进程，其采取 Socket 的方式与 Wine 加载的 Windows 应用程序进行通信。

第三部分：Linux 库函数的支持，即图 2.4 下面白色的部分，这部分提供 Linux 的系统调用，是任何运行在 Linux 上的应用程序的基础，Wine 也不例外，Wine 利用 Linux 的系统功能，为上层的 Windows 应用程序提供系统服务，从而实现在 Linux 上对 Windows 应用程序的支持。

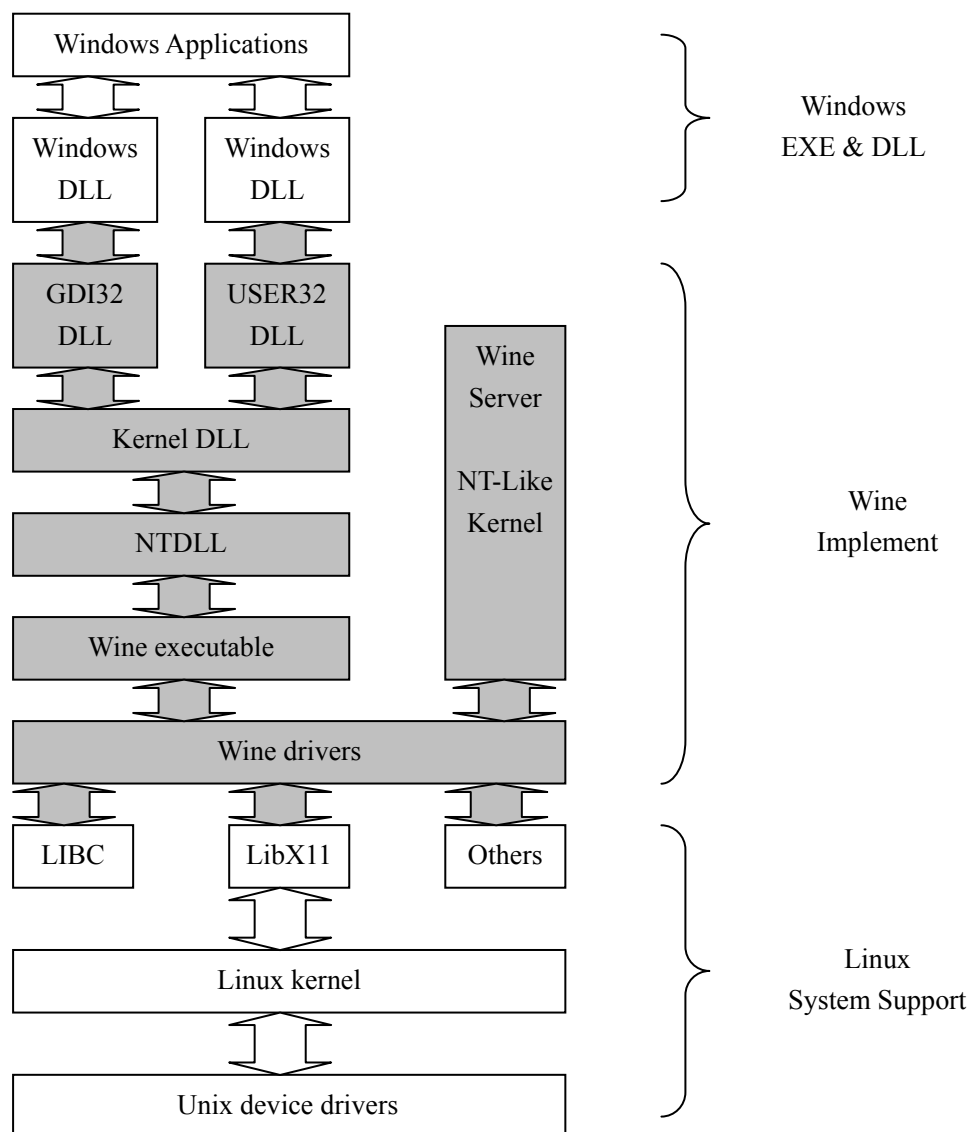


图 2.4 Wine 的体系架构

Wine 是一个建立在在 Windows 应用程序和 Linux 系统之间的一个 Windows 兼容层，其提供了 Windows API。可以将 Wine 看做是 Windows 在 Linux 上的模拟模拟环境或模拟器。Wine 利用 Linux 的系统调用实现了对 Windows 应用程序的支持，并在 Linux 下提供了一个 Windows 应用程序加载器,该加载器允许不经任何修改 Windows 3.1/95/NT 的二进制文件运行在 Linux 及其衍生版本下。Wine 可以工作在绝大多数的 Linux 版本下，而不需要 Microsoft Windows 提供任何支持，同时其还是一个开源项目，因此可以选择的实现部分功能。除此以外，它

可以随意地使用本地系统的 DLLs，只要这些 DLL 是可以被利用的。Wine 的发布是完全开源代码的，并且是免费发行的。目前 Wine 仍在发展阶段，仅能执行少部份的 Windows 软体，大部份的软体仍然无法正常执行。

2.3.2 Wine 特点分析

尽管 Wine 并不愿意其他人将其称为虚拟机，但是将其看作是 Linux/UNIX 操作系统上运行 Windows 应用程序的虚拟机并不为过。Wine 可以将 Windows 应用程序不作任何修改就可以运行在 Linux/UNIX 上。但是与 Java 虚拟机不同在于：Java 虚拟机是对 CPU 的仿真，而 Wine 则是对操作系统的仿真。因此，也可以将 Wine 看作是在 Linux 操作系统下执行部分 Windows 应用程序的工具。

作为工具，Wine 在 Linux 和 UNIX 之上，实现了 Windows 3.x 和 Windows APIs。因此，Wine 具有以下特点：

首先，Wine 提供了一个 Windows 兼容层，这个层提供了一个用来从 Windows 源（Windows 源代码）进入到 Linux 的开发工具包(Winelib)，Winelib 是一个开发工具包，它允许你在 Linux 系统上编译 Windows 应用程序。

其次，Wine 提供了一个程序加载器，该加载器允许加载不用任何修改 Windows 3.1/95/NT 的二进制文件；

第三，Wine 自身维护了一套注册表机制，Wine 注册表机制和 Windows 是一样的。Windows 注册表通常都包含大量默认值，他们中的一部分是系统运行所必须的。Wine 为了运行 Windows 的应用程序，因此也必须要保证操作系统注册机制的一致性。

第四，Wine Server 在 Wine 中是最重要的了，因为其他进程都要通过 Wine Server 来和 Linux/UNIX 操作系统内核进行交互。那么 Wine Server 在 Wine 中具体起到什么功能？简单的说，它提供了 Inter-Process Communication (IPC)，同步和进程/线程管理。

第五，Wine 有四个核心的动态连接库模块，他们分别是 User、Kernel、GDI 和 NTDLL。这几个模块上层提供了所有的 Windows 内核 API 接口函数，满足了应用程序跨平台的要求。同时它为 Linux 内核提供了系统调用的机制。通过 Wine Server，可以将客户进程的 Windows 系统调用转换为 Linux 的系统调用。

第六，Wine 是一个完全由免费代码组成的，Wine 的发布是完全开源代码的，并且是免费发行的，因此可供研究和学习。

上面的六点是 Wine 的特色，也是 Wine 的吸引力所在。Wine 扩展了虚拟机的定义，为 Windows 应用程序跨平台运行探索了一条新路，对本课题的研究也提供了大量可借鉴的经验和技术。

2.4 可执行文件分析：PE 和 ELF

2.4.1 PE 文件格式

PE^[7]文件格式（Portable Executable）是从Windows NT 3.1 引入的一种新可执行文件格式。PE文件格式在制定过程中主要参照了UNIX操作系统所通用的COFF规范，同时为了保证与旧版本MS-DOS及Windows操作系统的兼容，PE文件格式保留了MS-DOS中常见的MZ头部，因此PE文件最开始的两个字符是MZ，而不是PE。PE文件被称为是可移植的，并非是指PE格式的可执行文件可以在不同的平台（如x86、Alpha、MIPS等等）上执行，而是指不同平台上加载该PE文件的加载器无需经过完全重写就可以达到加载PE文件的目的。

PE 文件的结构有点杂乱，这主要是由于历史原因造成的，图 2.5 中描述了 PE 文件的一般格式。可以将其分为四部分，用不同的颜色加以区分，其中前两部分是所有 PE 程序所必须的具备的，架构次序也是固定的，各部分的作用如下：

第一部分，即淡黄色的部分，是 DOS 残留部分，其并没有太多实际的用处，只是为了与原来的 DOS 的 MZ 文件格式保持一致，并可以在非支持 PE 文件的老版本的 DOS 系统上执行；

第二部分，即淡蓝色的部分是 PE 文件真正的文件头，虽然其中一部分被称为 PE 可选头部，但实际上是 PE 文件必不可少的一部分；

第三部分，是 PE 文件所包含内容的索引，其具体内容随着 PE 文件的不同会有差别；

第四部分是 PE 文件真正保存程序和数据的部分，因此是 PE 文件的核心，前面的三个部分都是在为这部分服务并提供这部分的相关信息。



图 2.5 PE 文件结构

2.4.2 ELF 文件格式

ELF^[8]是Executable and linking Format, 也称为可执行连接格式。ELF是UNIX系统实验室(USL)作为应用程序二进制接口(Application Binary Interface(ABI)而开发和发布的。在SVR4 和Solaris 2.x上, 都做为可执行文件默认的二进制格式。ELF比a.out和COFF更强大更灵活,结合一些适当的工具, 程序员使用ELF就可以在运行时控制程序的流程。工具接口标准委员会(TIS)选择了正在发展中的ELF标准作为工作在 32 位INTEL体系上不同操作系统之间可移植的二进制文件格式。Linux上默认的可执行文件格式格式为ELF。ELF结构如图 2.6 所示。

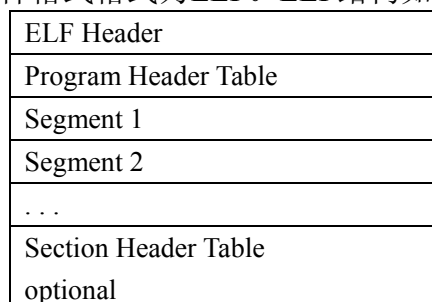


图 2.6 ELF 完整的结构图

每个 ELF 头在文件的开始, 都保存了路线图(road map), 其中描述了该文件的组织情况, 这部分也称为 ELF header 头。ELF 文件的头部是一个 52 个字节大小的 elf32_hdr 结构, 其包括两个数据成员:

```
Elf32_Off e_phoff;
```

```
Elf32_Off e_shoff;
```

它们分别代表了 program header 与 section header 在 ELF 文件中的偏移量。ELF 文件头包含了描述文件结构的重要信息。在文件头的指引下我们能够找到文件的程序头表（program header table）和 section 头表（section header table）。进而形成对 ELF 文件的有效索引和管理。如查找符号表，重定位信息等等。

2.4.3 PE 与 ELF 文件格式比较

ELF 和 PE 都是目前广泛使用的可执行文件格式，但 ELF 文件相对 PE 文件来说，结构更清晰，更简单，这是因为 ELF 不用考虑太多与历史版本兼容的问题。抛开两种标准的差别，无论 PE 格式还是 ELF 格式，其目的都是保持应用程序的基本执行信息，以提供给加载器所必需的连接、设置信息，当应用程序加载器（PE 或 ELF 加载器）把应用程序（PE 或 ELF 格式可执行程序）加载到相应的地址空间后，PE 格式或 ELF 格式对于系统运行平台来说就不再具有意义，因为加载后的程序或数据并没有 PE 或 ELF 的区别，加载器的使命也宣告结束。而真正影响被加载应用程序能否正常运行的因素是，其机器指令的格式，是否遵循相同的堆栈约定，是否有合理的程序逻辑等文件格式无关的其它方面。

具体说来，Elastos 上编译生成的 PE 格式可执行文件和 Linux 上编译生成的 ELF 格式可执行文件的区别表现在以下几个方面：

（1）对文件头的定义和节（section）表的描述上，每个节中的内容也略有差别，但这些差别都是形式上的，其本质都是为了将可执行文件映射到相应的地址空间。

（2）ELF 加载起始地址 0x8000000 (128M)，其下的 128M 空间没有使用，而 PE 默认加载地址是 0x400000 (4M) 开始的地址，因此 0x400000 到 0x8000000 之间存在 124M 的空间没有被 Linux 使用。

（3）Linux 中共享库的加载地址为 0x40000000 (1G) 开始的地址，Elastos 上动态库的加载地址为 0x10000000 (256M) 开始的地址，两地址间同样存在未使用的地址空间。

（4）Elastos 在加载 CAR 构件过程中与普通的 PE 格式文件还有些不同：在加载 CAR 构件时，需要解析 CAR 构件所包含的自描述信息（元数据），并将这些信息保存，以便于其他进程对该构件的访问。

图 2.7 比较了 Linux 和 Elastos 应用程序加载后的地址分布情况。

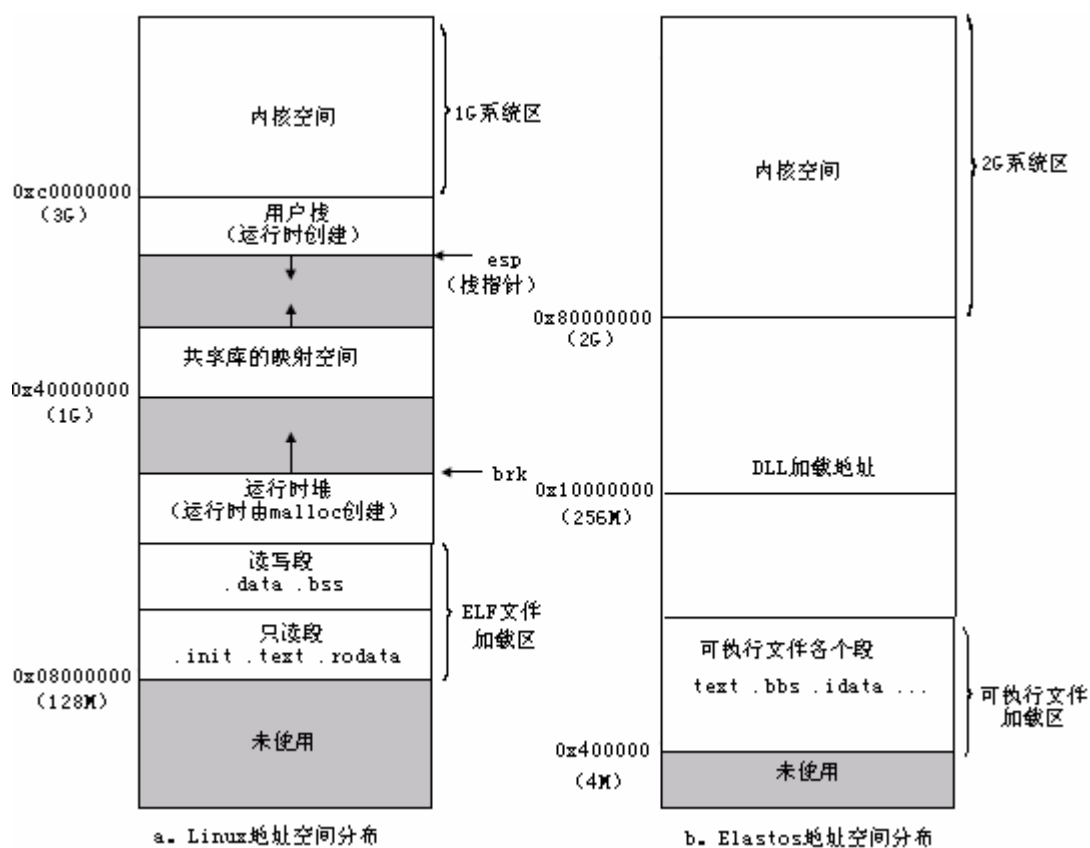


图 2.7 linux 和 Elastos 地址空间分布

从图 2.7 中可以看出，ELF 格式应用程序加载与 Elastos 应用程序加载后的地址空间分布并没有太多的冲突，而且 Linux 上有 3G 的用户地址空间，可以考虑将 Elastos 应用程序加载到 Linux 未使用的地址空间中，这样就解决了 Elastos2.0 应用程序地址空间问题，本文后面在介绍的 Elanix 加载器时，还会详细探讨这个问题。

2.5 元数据

2.5.1 元数据定义

普通的源文件（c 或者 c++语言）经过编译器的编译产生二进制的文件，其中通常不包含数据的类型信息。比如一个指针，单看编译完之后的二进制代码或汇编已不能区分它是整型或是 char 型了，如果是指向字符串的指针，字符串

的长度也无从知晓。这部分类型信息就属于我们所说的元数据信息。

元数据^[9](metadata)，即描述数据的数据(data about data)，这可以从两方面理解，一方面数据也是一种数据，另一方面数据还是对数据的一种抽象，它主要描述了数据的类型信息。

构件技术也需要应该有相应的元数据来描述。一般说来，由于构件主要通过暴露一组接口来提供对外的服务，所以构件对象首先需要向外提供的是接口的元数据。接口元数据往往包括了描述这些对象支持的接口列表，各个接口中的函数布局，和所有函数的函数参数属性等信息。除了接口的元数据外，构件元数据还需要包括其他一些信息，如构件的版本，构件全局标识符，构件依赖的运行环境，同其他构件的依赖关系等等。

Microsoft 的 COM 技术、OMG COBRA 中都使用元数据作为构件交互过程中的不可缺少的加载依据。

2.5.2 CAR 构件元数据

CAR 构件也需要元数据以提供 CAR 构件版本，服务等方面的信息。CAR 构件以接口方式向外提供服务，因此需要元数据来描述接口信息，这样其才能被其他使用构件服务的用户所使用。CAR 构件为了让接口与实现无关，保持了接口的不变性，使得动态升级成为可能；并且通过使用 vptr 结构将接口的内部实现隐藏起来，由接口的元数据来描述接口的函数布局 and 函数参数属性。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不同构件之间的调用才成为可能，也只有这样 CAR 构件的远程化，进程间通讯，自动生成 Proxy 和 Stub 及自动 Marshalling、Unmarshalling 等机制才能正确地进行。

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

在 CAR 里，ClassInfo 被作为构件程序的元数据信息，用于描述构件导出的接口及方法列表。同时 ClassInfo 也是自动生成构件源程序的基础。

在 CAR 构件平台中，元数据主要以两种形式组织和存储：CTL (Component Type Library) 元数据和 ClassInfo 元数据。

CTL 元数据：该类元数据保存了构件开发者使用构件描述语言在 CAR 文件中描述的所有信息。由于这些信息比较多，而且当中的相当一部分内容不会被经

常使用，因此这部分内容是在压缩打包后放到 DLL 中，这个压缩及打包到 DLL 的过程都是由 CAR 开发环境提供的 `carc.exe` 自动完成的。

ClassInfo 元数据：这类元数据记录了 CAR 文件经常需要被 CAR 构件运行平台使用到的部分元数据，其内容是 CTL 元数据的子集，具体说来 ClassInfo 包括以下信息：

构件 DLL 中实现的构件类（class）数目、接口（interface）数目及其列表。

构件 DLL 中每个构件类实现的接口数目和列表、及该构件对象对应的 CLSID。

构件 DLL 中每个接口实现的方法数目和列表，及该接口对应的 IID。

构件 DLL 中每个接口方法的参数数目和参数类型列表。

为了方便使用，ClassInfo 元数据一静态数组的形式存储于 DLL 的静态数据区，每次加载时被加载到构件所在的地址空间。

综上所述，CAR 构件使用的元数据以两种形式存在：一种是以单独的文件形式存在，存放在目标目录中，最终会被打包到 DLL 的资源段里，该文件的后缀名为 `cls`，如 `hello.car` 将会生成 `hello.cls`；另一种是与构件的实现代码一起被打包到构件模块文件中，这部分元数据是第一种元数据的子集，其保存在 CAR 构件的静态数据区，以方便列集和散集使用。

第3章 Elanix 结构设计与实现

3.1 引论

Elastos操作系统架构建立的依据之一就是要兼顾跨平台的需求，而将Elastos上的应用程序加载到Linux、Windows等系统上运行是Elastos操作系统跨平台特性的具体体现，也是Elastos技术特色的重要方面。

Elanix是在Linux上的一个Elastos2.0虚拟平台，在其上Elastos2.0的应用程序可以不经任何修改、以二进制形式在Linux系统上运行。Elanix的开发借鉴了Wine的体系架构，但采取了更加灵活的策略，并针对Elastos系统的特点，从效率，实现等角度考虑，以更加动态的机制增加对Elastos2.0应用程序运行的支持。

就整个Elanix虚拟平台而言，Elanix不仅是Elastos2.0程序的运行平台，而且还是Elastos2.0应用程序的开发平台以及Elanix虚拟机的开发环境。Elanix提供了开发Elastos2.0 应用程序所需的自动代码生成工具，并建立了相似的开发环境、测试环境、以及文档管理。Elanix参照Elastos2.0 NTOS（在Windows上的Elastos2.0虚拟操作系统）、Wine（在Linux上的Windows虚拟操作系统）、Linux和Elastos2.0的特点，分析了不同系统之间的交换问题，并在针对性试验基础之上，设计了Elasnix的体系架构。本章就上面这些方面内容进行分析介绍。

3.2 Elanix — 开发环境建立与维护

Elanix作为Linux上对Elastos2.0模拟，其需要建立一个类似Elastos2.0的开发、测试环境、以及文档管理，建立这些环境和保留相关文档的好处是：提供便利的开发测试环境，提高开发效率，建立有效的文档管理机制，促进项目参与者的交流，也为项目新成员提供技术指南。Elanix开发环境包含以下几个部分：

（一）Elasnix 开发环境

环境中的工具和命令

Elanix的开发环境是参照Elastos2.0建立的，其主要包括：进入虚拟环境和相应的环境设置，文件的配置选项，编译生成可执行文件的命令，目标代码目录

与源代码目录之间的对应关系，设置环境默认路径（包括：库路径，可执行代码路径，头文件路径等）。

Elanix环境中工具和命令交织在一起。工具都是程序，都是通过命令来调用。命令却不一定是程序，有些命令只是shell长命令的缩写。目前Elanix环境中的主要工具及命令的清单如下：

命令	功能	帮助命令
pd ,	在当前目录及目标镜像目录之间切换	pd ?
emake	在当前目录下编译	emake -?
addevsdir	批量在 CVS 服务器上添加目录及文件	addevsdir
delcvmdir	批量从 CVS 服务器上删除目录及文件	delcvmdir
treedir	显示指定目录的树型结构图	treedir -?
gnuplot	GNU 的画图工具	gnuplot -h
z	emake 简称	alias
za	emake all 简称，编译所有 src 下目录	alias
zc	emake clean 简称，删除目标镜像目录	alias
zv	emake -v 简称，编译时显示详细信息	alias
z clobber	清除当前版本的所有目标代码与 za 配合使用	无
src	跳转到源程序目录	alias
obj	跳转到目标程序目录	alias
ela	跳转到 src/elastos 目录	alias
tst	跳转到 src/testing 目录	alias
updatetags	更新项目中所有的源文件的 tags	updatetags -?

Elanix环境是仿照Elastos环境建立的。因此大部分命令，工具以及路径设置方式与Elastos2.0中相同，但由于Linux与Windows系统上的差别，两者还是存在一些不同，主要表现在以下方面：

1. 可以通过文本方式进入Elanix开发环境，而且是新建了一个Bash，如果使用exit命令，可以退回到原来的Linux环境中。
2. Elanix环境中目录的快捷方式加上参数无效。比如在Elastos环境下，可以用“build misc”进入D:\Elastos\build\misc，但在Elanix环境下，只能用build命令，进入~/Elanix/build目录，再使用“cd misc”才能达到同样效果。这是由于Linux alias的限制造成的。
3. Elanix环境下路径是以"/"分隔，Elastos环境下则是用"\\"分隔。
4. 目前Elanix环境支持开发Linux可执行程序、静态库、动态链接库和内核

模块四种。在sources文件中Linux的静态库的TARGET_TYPE为a，而Elastos中静态库的TARGET_TYPE为lib。Linux的动态链接库的TARGET_TYPE为so，而Elastos中动态链接库的TARGET_TYPE为dll。

5. Elanix上不仅在/build/tools目录下建立了pd.sh脚本，并且在alias.pub中建立了快捷命令pd=". pd.sh"。这是由于Bash脚本运行和bat脚本运行方式不同决定的。只有使用". pd.sh"才能使得pd.sh中对当前目录的更改，反应到当前Bash环境中。Elanix上的pd采用","字符来表示镜像目录，这是因为Elastos20中使用的"~"、""字符在Linux Bash中都有特定的含义，所以不得不修改为其它字符。

6. ElanixDDK和SDK都是放在一个目录下的。而Elastos20中DDK和SDK分为两个目录。

7. Elanix中.so动态链接库放在LIB目录下而Elastos20中.dll动态链接库放在BIN目录中，这是因为.so动态链接库不仅在运行时被使用，而且编译时也要使用，不像.dll动态链接库只在运行时使用，编译时使用的是对应的lib文件。Elanix环境将LIB目录添加到LD_LIBRARY_PATH环境变量中，这样Linux程序运行时可以找到LIB目录中的.so动态链接库。

Elanix环境中程序编译的配置文件

目前Elanix环境支持多种程序的编译，包括elf可执行文件，.a静态库，.so动态链接库，.S汇编源文件。一般情况来说，在Elanix环境中编译程序都要写dirs文件和sources文件。dirs文件中写明了该目录下还需要编译的子目录，sources文件中写明了该目录下的源文件如何编译。一个较为复杂的编写elf可执行文件的sources文件示例如下：

```
TARGET_NAME= elanix
TARGET_TYPE= exe

C_FLAGS= -D_GNUC -D_x86 -D_USECOMPTTR
LINK_FLAGS=-ldl
SPECIAL_TARGET= $(SYSTEM_BIN_PATH)/$(TARGET)

SOURCES= \
    elanix.cpp \
    loader.cpp \

LIBRARIES= \
    $(TARGET_LIB_PATH)/coapi.a \
```

TARGET_NAME和TARGET_TYPE指明了要生成的目标，C_FLAGS是编译时添加的参数，LINK_FLAGS=-ldl是链接时添加的参数，SPECIAL_TARGET是生成目标后还需要完成的工作，通常与makefile.inc文件配合使用，SOURCES列出了要编译的源文件，LIBRARIES列出了要使用的库。

（二）Elanix测试环境Dailybuild

Elanix项目是由Elanix项目小组几位同学合作开发的，因此项目开发过程中不可避免的需要碰到不同模块之间的协调、交互的问题，按照以往的经验，每个模块都由一个同学负责，在项目的中后期，再统一集成测试，这种开发和测试方式带来的一个显著问题就是，将不同模块之间的协作测试放在了项目的最后，这样如果前期的某些BUG或模块的设计在单个模块测试时不能表现出来，但在集成测试的过程中确是不可克服的，这样会明显的拖延项目的进度，也给开发人员带来了巨大的压力，因此在开发过程中实时地监控系统的开发进度，将是非常关键的，为此，参照Elastos2.0，Elanix也建立了类似的DailyBuild测试环境。该测试环境的目标就是：建立一个自动化的贯穿整个过程的测试系统，用来编译、测试、采集测试数据、进行数据分析并且给出分析报告，可以及时发现问题并更准确地暴露问题所在，提高每天工作的效率。而且根据这份报告，可以制定出相应的补救措施与工作安排，尽早解决问题。

DailyBuild顾名思义，Daily就是每天的意思，Build除了编译之外，还延伸到测试、测试数据收集、测试数据分析和生成报告，这整个过程都是自动化的。Daily Build实际上还可与CVS相结合的，可以更方便地根据不同的需要去选择不同的条件进行Daily Build。

目前实现的功能是简单的回归测试和代码行统计，以及使用gnuplot生成图表。自动测试的基本流程如下表：

测试步骤	步骤说明	对应代码片断
代码更新	删除 Daily Build 服务器上有关的所有源代码，以保证当前 Daily Build 服务器的环境是干净的	ELASTOS_ROOT=\$HOME'/ElaLinux' rm -fr \$ELASTOS_ROOT'/obj' rm -fr \$ELASTOS_ROOT'/src' rm -fr \$ELASTOS_ROOT'/build'

	把 CVS 服务器上的最新代码更新到 Daily Build 服务器上, 以保证当前代码是最新的	cd \$ELASTOS_ROOT cvs update -P -d . >> /dev/null
编译最新代码	设置编译环境为 release 环境, 并编译所有为 release 版本	. elalinux_rls.sh > /dev/null za && echo 'Success' echo 'Failed'
	设置编译环境为 debug 环境, 并编译所有为 debug 版本	. elalinux_dbg.sh > /dev/null za && echo 'Success' echo 'Failed'
运行测试程序	在所有编译完成后, 对特定程序开始分别测试, 每个程序的自动测试一般都包括执行测试程序, 和比较执行结果两步。	. _runtest.sh > \$DATAPATH'/phrase1_step1.dat' diff \$DATAPATH'/phrase1_step1.dat' \$REFLECTPATH'/phrase1_step1.out' && echo 'Success' echo 'Failed'
代码统计并生成图表	统计 src 目录下所有的文件的代码行数(除 CVS) 并与当前日期一起作为一个记录添加到名为 statistics.txt 的文本文件里。	#count code lines #count records of statistics.txt #count the date paste \$DATAPATH'/tmpdate' \$DATAPATH'/tmpwc' paste \$DATAPATH'/tmpnum' - >> \$ELASTOS_ROOT'/dailybuild/statistics.txt'
	使用 gnuplot 根据 statistics.txt 中的统计数据生成图表	plot "tmpstatistics" using 2:3 with linespoints notitle

(三) Elanix文档资料

文档是Elanix项目中的重要一环, 文档不仅仅作为同学对自己负责部分的总结, 还起到了和其他同学交流的基础, 还可作为未来参与本项目同学的入门指南。Elanix项目中对文档的采取和开发同样的重视态度, 对每个模块的开发和测试都有详细的文档说明和相关技术背景分析。下表列出了Elanix文档的层次目录。

目录	说明
dailybuild	包含 DailyBuild 相关的技术文档
elanix_env	设置 Elanix 环境的相关文档

elastos	与 Elastos2.0 相关的技术文档
loader	Elanix 加载器相关的技术文档
project	项目相关文档，包括项目管理，工具的生成等
roadmap	Elanix 路线图文档
server	Elanix Server 相关文档
wine	Wine 分析的相关文档

3.3 Elanix 系统结构

3.3.1 Elanix 系统结构

Elanix系统结构的建立是在结合Elastos2.0自身的特点，参照Wine架构设计的基础上建立起来的。其分为三个部分：

第一部分是Elastos2.0应用程序（包括EXE和DLL）以及CAR构件部分。这部分是Elastos2.0应用程序以二进制源码被加载到Linux上的部分，因而也是Elanix项目的主体运行部分。

第二部分是PE Loader和Elastos.so部分。这部分实现对Elastos2.0应用程序的加载，以及通过Elastos.so实现Elastos2.0应用程序的系统调用，因而这部分是Elanix的加载和系统支持部分。

第三部分是Elanix Server部分。这部分实现对Elastos2.0内核的支持，使得Elastos2.0应用程序可以在Linux上获得Elastos2.0内核提供的服务，如CAR构件的跨进程调用，创建新的进程等等。

Elanix框架结构如图3.1所示。

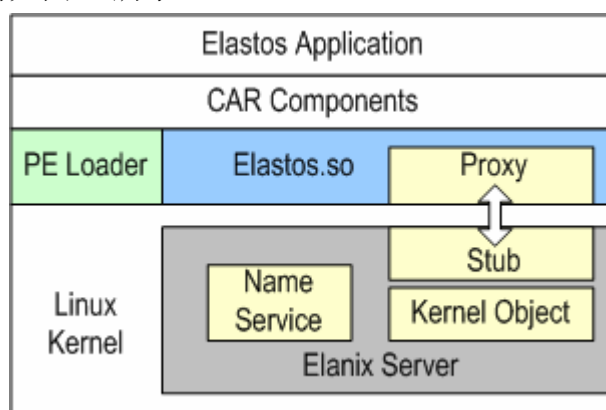


图 3.1 Elanix 框架结构

从图3.1中可以看出，Elanix是建立在Elastos2.0应用程序和Linux系统调用之

间的虚拟层，而Elastos2.0内核的功能则在Linux内核中实现，以提供更高的效率及调用更底层的系统服务。

3.3.2 Elanix 实现思路

Elanix 加载 Elastos2.0 应用程序的实现思路是：

首先，加载部分：在 Linux 上，每个 Elanix 进程对应一个 Linux 进程，Linux 进程为 Elanix 加载的应用程序提供地址空间，同时该进程还负责 Elastos2.0 应用程序的加载。因此，每个 Linux 进程启动的时候都会有一个 PE Loader，该 PE loader 将会把 Elastos.so，指定的 Elastos2.0 的 PE 可执行文件，以及相关的其它模块如 DLL 或外部模块加载到该进程自身的地址空间内。

其次，系统调用部分：Elastos2.0 程序在执行过程中，如果调用 Elastos2.0 API，将会调用 elastos.so 中同名的函数。Elastos.so 将会综合使用 Elanix Server 和 Linux 系统调用完成该 API 请求。有关 CAR 构件方面的机制也将通过 Elastos.so, Elanix Server 和 Linux 系统调用三方协作完成的。

第三，外部模块部分：一些 Elastos2.0 应用程序会调用其他的 dll 或外部模块，如 elacrt.dll，elacmd.dll。这些 dll 如果遇到系统调用，则系统调用都是通过 elastos.so 调用 Linux 的 API 实现的，因此这些 dll 不必以.so 共享库的方式存在，而直接以加载普通 PE 文件的方式将其加载，从而节省了不必要的工作量。但也有些模块需要以 so 的方式实现，如，elasocket.dll，因为这类的 dll 与 Linux 上所提供的功能类似，通过 so 封装可以提高 Elastos2.0 应用程序在 Linux 上的稳定性，同时对这些 dll 的包装实现方便，因此也是 Elanix 采取的一种模块支持策略。具体采取那种方式需要根据具体情况决定。

第四，Elastos 内核部分：在 Elanix 上内核模块称为 Elanix Server，其是一个 Linux 内核模块，因此它处于 Linux 内核空间，并可以为所有 Elanix 进程提供内核服务，如进程间通信，信号量，同步互斥等。

上面简要介绍了四个部分的实现思路，它们是 Elanix 中的主要部分，也是实现 Elanix 虚拟平台的关键技术，本文后面的章节中会详细介绍实现的过程和依据。

第4章 Elanix 加载器

4.1 Elanix 加载器设计原理

4.1.1 加载器理论依据

Elanix的设计目标是将Elastos2.0应用程序以二进制的PE文件格式加载到Linux上运行，即二进制文件的跨平台执行，这与一般的源程序级，或中间代码级的跨平台技术有很大的不同。其好处不需修改可执行程序，避免重新编译和连接源代码的时间。但是，Linux上可执行文件格式为ELF，而Elastos2.0应用程序格式则为PE，Linux不支持PE文件的加载，因此必须在Linux上实现PE文件加载器，这样才能实现PE文件的加载。至于Elastos2.0应用程序能否在Linux环境下运行，则需要考虑一些系统相关的问题。一旦这些问题的得到解决，那么对于将Elastos2.0应用程序加载到Linux地址空间运行的技术而言，其在实现上具有可行性。同时，对这些问题的解决也是加载器设计的理论基础和设计依据。下面就相关的问题分别进行讨论：

一、Elastos2.0应用程序与Linux应用程序的机器码格式兼容性

可以肯定，两者的机器码指令集兼容并不是障碍，因为Elastos2.0可以作为X86体系架构计算机的操作系统，并且在其上可以允许加载运行基于X86架构的Elastos2.0应用程序。这样基于X86架构的Elastos2.0应用程序和基于X86架构的Linux上的应用程序，在指令编码上并没有不同，其指令集格式都为IA32（Intel Architecture 32-bit）。通过对用objdump等工具导出应用程序的机器码分析，Elastos和Linux应用程序所对应的汇编代码和机器码格式是完全一致的，这样就为Elastos应用程序在Linux上运行提供了指令级的支持。

二、字节顺序的一致性

有关字节顺序的一致性问题的，同样可以从基于X86架构这个条件中找到答案。字节顺序是指存储器中的字节如何排列以及顺序关系。通常，多字节的数据类型对象被存储在连续的字节序列中，对象的地址为该序列中最小的地址，但对

象的数值根据在地址空间中顺序的不同,分为大端法(big endian)和小端法(little endian), X86架构的处理器,都是采用小端法,因此,该问题仅与处理器架构有关,与操作系统及应用程序没有关系,因此,在这个问题也同样可以获得了一致的答案。

三、寻址方式与地址空间隔离

首先,在寻址方式上, Linux采取平面寻址方式(flat addressing),在这种方式下,程序员可以将整个用户的地址空间看作是个大数组,并可以通过指针进行访问。Elastos2.0上采取相同的寻址方式,尽管两者可访问的用户空间大小不同,但这并不影响双方在合法地址空间中的相互寻址。事实上, Linux为Elastos2.0应用程序提供了更大范围的地址空间,因为Linux的用户空间大小为3GB,而Elastos2.0用户空间为2GB,而加载到Linux地址空间的Elastos2.0应用程序不但可以访问已有的2GB地址空间,还可以通过linux支持,访问Linux上多出的1GB地址空间,因此从理论上讲,将Elastos2.0应用程序加载到在Linux地址空间并没有太多大小的限制。总之,两者相同的寻址方式,为Elastos2.0应用程序与Linux系统的互操作,提供了基础。

其次,在地址空间隔离方面, Elastos2.0应用程序加载到Linux用户地址空间,其不能直接访问Elastos2.0内核空间,即0x80000000以上的地址空间,而Linux内核空间为0xC0000000开始的地址空间,因此通过Elastos2.0应用程序访问Linux内核空间的可能性更小,按规范编写的Elastos2.0程序是不会直接访问内核地址的。这样将Elastos2.0应用程序加载到Linux用户地址空间运行,并与Linux上其它应用程序隔离,确保了加载的安全性。这样,即使Elastos2.0应用程序发生错误,其也不会对Linux内核或其它进程产生影响。

四、Elastos2.0上的数据类型和Linux上的数据类型的匹配

数据类型是否匹配,主要为了确保在Elastos2.0应用程序调用Linux系统函数或Linux应用程序调用Elastos2.0提供的函数过程中,是否能保持参数入栈和出栈的一致性,这也是不同系统间函数相互调用首先要考虑的问题。数据类型匹配与否,主要表现在两个系统中相同数据类型的字符宽度是否一致。表4.1比较了两个系统间主要数据类型的字符宽度。从表中可以看出, wchar_t在两个系统上所占的字符宽度并不相同,因此一种变通的方法,就是在Linux定义新类型

Wchar_t，使其与Elastos上的wchar_t所占字符宽度相同，这样就解决了数据类型字符宽度不匹配的问题，因此第四个问题也获得了解决。

类型	Elastos	Linux
char	1	1
short int	2	2
int	4	4
long int	4	4
void *	4	4
float	4	4
double	8	8
wchar_t	2	4

表4.1 Linux和Elastos2.0常见类型字符宽度比较

五、过程调用中的调用格式、以及对栈的维护

对于这个问题，分为两个问题分别加以讨论：

首先要解决的是在函数调用过程中，堆栈的使用方式如何约定。这种调用约定(Calling convention)包括以下内容：函数参数的压栈顺序，由调用者还是被调用者把参数弹出栈，以及产生函数修饰名的方法。常见的调用约定方式包括以下几种：_cdecl，_stdcall，_fastcall等，Elastos上的函数默认调用方式为_stdcall，因此为了保持与Elastos上的一致，基于Linux系统的Elanix加载器，在开发过程中对所涉及的函数也采取相同的约定，即_stdcall，这样就可以保证Elanix加载器中的函数接口与Elastos2.0应用程序的函数接口保持调用关系的一致性，为两系统函数之间的相互操作提供基础。

其次，要解决的问题是：Elastos2.0应用程序运行在Linux系统中，栈由谁来维护？一般说来，基于X86体系结构的应用程序都是通过程序栈来支持函数调用的，栈是用来传递参数，保存返回信息，保存寄存器信息，以及保存过程局部变量的地址空间。因此，在X86体系结构下，包含两个与栈相关的寄存器，%ebp（帧指针寄存器），%esp（栈指针寄存器）。栈指针是可以移动的，而帧是信息获取的参照地址，帧栈在过程调用过程中的结构如图4.1。无论在Linux还是在Elastos2.0上，过程的调用都遵循帧栈结构，尽管Linux用户栈的起始地址和Elastos2.0用户栈的起始地址并不相同，但这并不影响Elastos2.0应用程序中的过

程在调用过错中适用Linux用户空间的栈，因为通过反汇编可以看到，在过程调用的栈的操作都是通过帧指针寄存器%ebp和栈指针寄存器%esp进行的。因此在第五个问题的第二个方面也获得了解决。

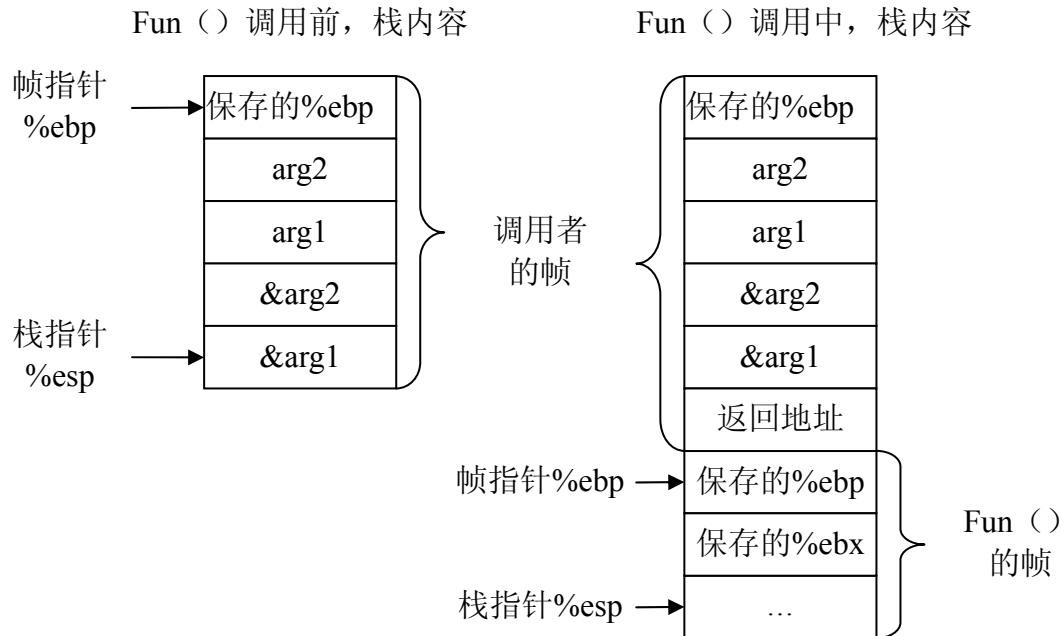


图 4.1 X86 帧栈结构示意图

六、如何在Linux上分配Elastos2.0中应用程序需要动态获得的地址空间

这个问题的解决，需要涉及到Elastos2.0应用程序和Linux系统调用交互的问题，Elastos2.0程序，不可避免的要用到new，malloc等内存分配函数，因此，设计到如何在Linux系统中为Elastos2.0应用程序分配地址空间，这个问题的解决并不困难，因为所有需要系统支持的函数都是通过Elastos.dll提供的API实现的，如new需要用到Elastos.dll中的EzMemAlloc()函数，因此只要在Linux上实现相同功能的API并供Elastos2.0应用程序使用即可，关于地址空间分配的问题，后面还会详细讨论。

解决了上面的几个问题，奠定了将一个Elastos2.0应用程序加载到Linux上并运行的理论基础，对这些问题的解答，也是Elastos2.0程序在Linux上运行的依据。特别是对于那些不涉及到任何系统调用的Elastos2.0应用程序来说，要想使其正

常运行在Linux系统上，只需将PE文件加载到Linux地址空间，找到相应的入口函数点，并跳转到其中运行，Elastos2.0程序就可以象其它的Linux应用程序一样运行在Linux系统上了。

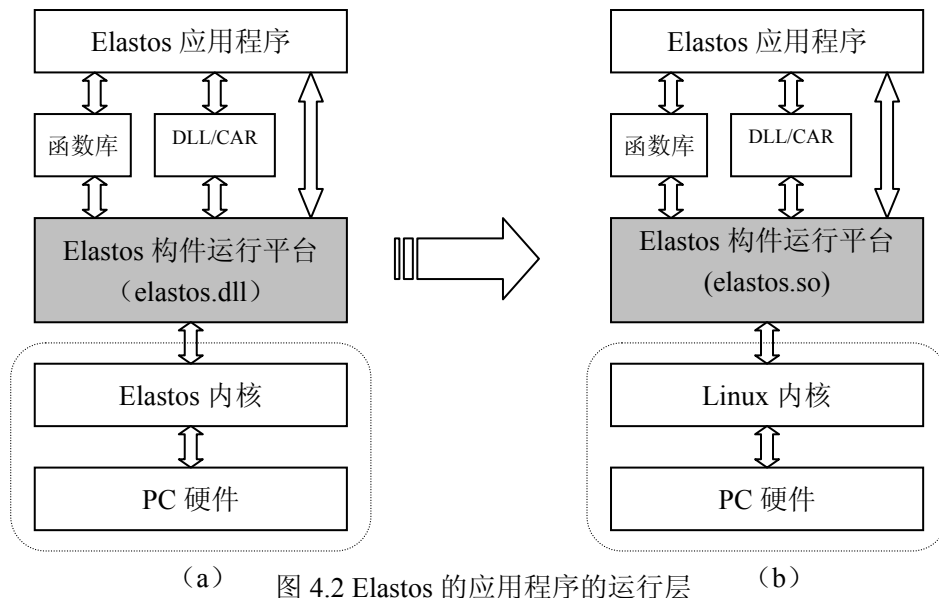
4.1.2 运行模式的转变

对于 Elastos2.0 应用程序来说，仅仅符合 X86 体系架构和机器码兼容，并不能保证其在 Linux 上顺利运行，这是因为 Elastos 应用程序还需要系统调用支持，正如上一节中第六个问题所提到的那样，需要系统支持的 elastos2.0 API 必须经过 Linux 相关系统调用的重新包装或实现，才能为 Elastos2.0 应用程序在 Linux 上的运行提供必要的支持。Elastos2.0 在设计之初就考虑到系统跨平台的特性，因此 Elastos2.0 系统对底层的硬件进行了封装，所有的 Elastos 系统调用被封装在 Elastos.dll 中。通过分析 Elastos 层次结构（如图 2.1），也可以看出，不仅 Elastos2.0 应用程序还有 Elastos2.0 上的 DLL/CAR 构件的系统调用都是通过 Elastos.dll 完成的。Elastos.dll 做为应用程序和系统（包括操作系统和硬件体系架构）之间的虚拟层，具有承上启下的功能。Elastos.dll 之上的应用程序和 DLL/CAR 构件，可以以源码的方式或二进制的方式，移植到不同的体系结构中；而在 Elastos.dll 之下则屏蔽了底层系统调用的差别。Elanix 加载器在完成对 Elastos2.0 应用程序和 CAR 构件加载的同时，还要在 Linux 系统中实现这个中间层，即在 Elanix 运行环境中实现一个与 Elastos.dll 中功能相同的 Linux 函数库，以提供 Elastos 应用程序所需的系统调用。

在 Linux 上实现 Elastos.dll 所具备的功能，最自然的想法就是在 Linux 上实现具有相同功能的动态连接库。在 Linux 上与 Elastos2.0 的 DLL 模块最相似的就是 so 模块了。so 模块也可以看成是 Linux 下的动态连接库，其可以通过 Linux 提供的 `dlopen()`，`dlsym()`，`dlclose()` 等系统函数加载 so 模块，查找 so 中所实现函数的地址，以及卸载 so 模块。用 Elastos.so 取代 Elastos.dll，这样就实现了 Linux 上对 Elastos.dll API 的支持。

在 Linux 系统中，有了 Elastos.so 模块的存在，通过 Elanix 加载器的加载和设置，将 Linux 系统调用转换为 Elastos2.0 所需的系统调用，为 Elastos2.0 应用程序在 Linux 系统上运行提供了底层的支持。这样 Elanix 中应用程序的运行结构就变成了图 4.2（b）所示。图 4.2 中（a）与（b）灰色的部分隔离了应用程序以及 CAR 构件对底层系统的访问。当 Elanix 加载器将应用程序、DLL/CAR 以

及 Linux 上实现的 Elastos 系统调用函数库加载到各自的地址空间后, Linux 中就实现与 Elastos2.0 相同的应用程序及构件运行环境, 这就保证了 Elastos 应用程序或 CAR 构件在 Elanix 上的正确执行。



(a) 图 4.2 Elastos 的应用程序的运行层 (b)

综上所述, “和欣”操作系统通过 Elastso.dll 来屏蔽底层的系统调用。Elanix 通过 DLL 到 SO 的模式转变这种模式, 确保了 Elastot2.0 应用程序可以以二进制的格式架载、运行在 Linux 上。这一模式还同样适用于 Elastos2.0 应用程序以二进制格式加载到 Widnows 操作系统上, 不过这时在 Windows 上是用新的 Elastos.dll (用 Windows 系统调用实现) 实现原来 Elastos.dll 的功能。

4.1.3 Elanix 加载器设计

通过上面两部分的讨论, 为 Elastos2.0 应用程序加载到 Linux 地址空间运行, 不仅仅提供了理论基础, 还解决了如何通过 Linux 系统为 Elastos2.0 应用程序提供系统调用的问题。因此, 两部分结合, 就解决了 Elastos2.0 程序在 Linux 上运行的理论和技术问题。但如何将 Elastos2.0 应用程序加载到 Linux 的地址空间呢? 这就需要有一个加载器将 Elastos2.0 应用程序加载到 Linux 地址空间, 该加载器可以分析加载 PE 格式的可执行文件或动态连接库。实现 PE 加载器是必需的, 这是因为 Linux 默认的可执行文件的加载格式是 ELF, Linux 并不支持对 PE 文件的加载, 尽管有些工具可以将 PE 文件格式转换为 ELF 文件格式, 但这种格式上的转变技术还不成熟, 另一个更重要的原因是这种格式的转变并没有解决系统调用问题, 因

此方案是行不通的。Elastos2.0应用程序是PE格式的，为了在Linux上运行Elastos2.0 应用程序，必须在Linux上实现PE文件加载器，即Elanix加载器。

Elanix加载器的实现目标为：

(1) 在Linux环境下编写PE格式的加载程序（Elanix加载器），该加载器是一个ELF格式的可执行文件，其可在Linux环境下加载Elastos2.0应用程序、动态连接库以及CAR构件。

(2) 被加载的Elastos2.0 EXE或DLL被映射到加载器所在的用户空间，加载器负责对分配空间的管理和加载模块的管理。

(3) Elanix加载在完成加载任务的同时，也为被加载的模块提供所需的地址空间，因此可以将Elanix加载器看作是Elastos2.0应用程序的载体。

(4) Elanix加载器在加载Elastos2.0模块过程中会涉PE文件的解析，代码段，数据段等的加载，外部模块的查找设置，输入输出表的读取会设置，以及重定位等。Elanix加载器还要负责加载与Elastos2.0相关的Linux模块的加载，如Ealstos.so。

(5) 当加载完成后，Elanix加载器将控制权交给Elastos2.0应用行程序的入口函数，至此加载器的使命结束。

(6) Elastos2.0应用程序开始运行，直到程序结束，退出应用程序的同时，退出Elanix加载器，释放加载器所占用的地址空间。

Elanix加载器的结构如图4.3。

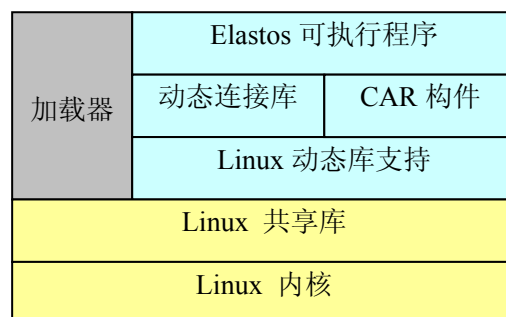


图 4.3 加载器结构图

4.1.4 加载器的地址空间分配方式

Elanix加载器的目的之一就是在Linux上能够运行指定的Elastos2.0程序，那

么一个Elastos2.0应用程序要运行，就必须加载到Linux地址空间中。前面已经提到Elastos2.0应用程序被加载到Elanix加载器所在的地址空间，因此内存的映射也必须以Elanix加载器的内存分配策略为依托。在应用程序中分配内存最常见的就是利用new, malloc等系统函数来从堆中分配内存，还有一种就是利用mmap来分配大块的内存，mmap还有个好处是可以指定分配内存的地址。对应于上面的内存分配策略，PE文件的加载也可以采用两种方式。

第一种方式是利用new, malloc等函数，在堆中分配一段空间，再将Elastos2.0应用程序的执行代码拷贝到分配的地址空间中，最后将跳转到堆中执行代码，程序执行后返回PE加载程序。如图4.4。

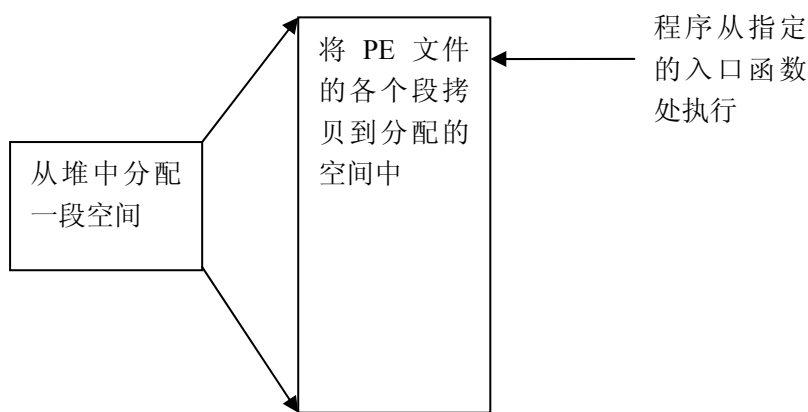


图 4.4 堆分配方式

这种方式的好处是内存分配策略简单，理论上可以分配所有未分配的用户地址空间，而且源代码的可移植性好。但此方式需要考虑两个问题，一是每次分配的空间的虚拟地址可能不相同，二是连续的两次内存的分配，可能会出现两次分配区域不连续的情况，这些问题对Elastos2.0应用程序的加载会产生直接的影响，其中最大的问题是寻址的问题，因为PE文件生成后，其指令中包含的地址，一般都是绝对地址（默认加载基地址+偏移量），如果没有在默认加载基地址加载相应的块，这里就需要考虑重定位的问题，这样第一的问题似乎好解决些。但对于第二个问题就比较麻烦了，如果各个段被分配到完全相互独立的不连续空间中，重定位的过程将相当繁琐。还有个需要考虑的问题，在Elastos2.0中有些环境数据是保存在0x400000以下的地址空间的，这是malloc或new是无法实现的，因为这些地址空间在Linux上一般是不会被分配的。因此，

采取这种方式，对于加载Elastos2.0应用程序并不可行。

第二种方式是利用mmap。mmap在Linux上主要作用是文件映射。Linux中的传统文件访问方式是，首先用open系统调用打开文件，然后使用read，write以及lseek等调用进行顺序或者随机的I/O。这种方式是非常低效的，每一次I/O操作都需要一次系统调用。而如果能够通过一定的机制将页面映射到进程的地址空间中，也就是说首先通过简单的产生某些内存管理数据结构完成映射的创建。当进程访问页面时产生一个缺页中断，内核将页面读入内存并且更新页表指向该页面，这就是mmap所实现的功能。mmap在文件映射方面，给用户提供了措施，似的用户将文件映射到自己地址空间的某个部分，使用简单的内存访问指令读写文件。

mmap的调用格式：

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

其中start是映射地址，length是映射长度，如果flags的MAP_FIXED不被置位，则该参数通常被忽略，而查找进程地址空间中第一个长度符合的空闲区域；Fd是映射文件的文件句柄，offset是映射文件中的偏移地址；prot是映射保护权限，可以是PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE, flags则是指映射类型，可以是MAP_FIXED, MAP_PRIVATE, MAP_SHARED，该参数必须被指定为MAP_PRIVATE和MAP_SHARED其中之一，MAP_PRIVATE是创建一个写时拷贝映射(copy-on-write)，也就是说如果有多个进程同时映射到一个文件上，映射建立时只是共享同样的存储页面，但是如果某进程企图修改页面内容，则复制一个副本给该进程私用，它的任何修改对其它进程都不可见。而MAP_SHARED则无论修改与否都使用同一副本，任何进程对页面的修改对其它进程都是可见的。

Elanix加载器利用mmap，将PE文件的各个段加载到其相应的虚拟地址空间，然后从PE文件的函数入口点开始执行，执行结束后返回PE加载程序。mmap还有个好处是可以从指定的地址分配空间，Elanix加载器能够顺利进行，正是依借住了mmap函数在地址空间分配的这个优势。在讨论具体方式之前，先分析linux的ELF和Elastos2.0上PE可执行文件默认的加载地址的不同。如图4.5所示：

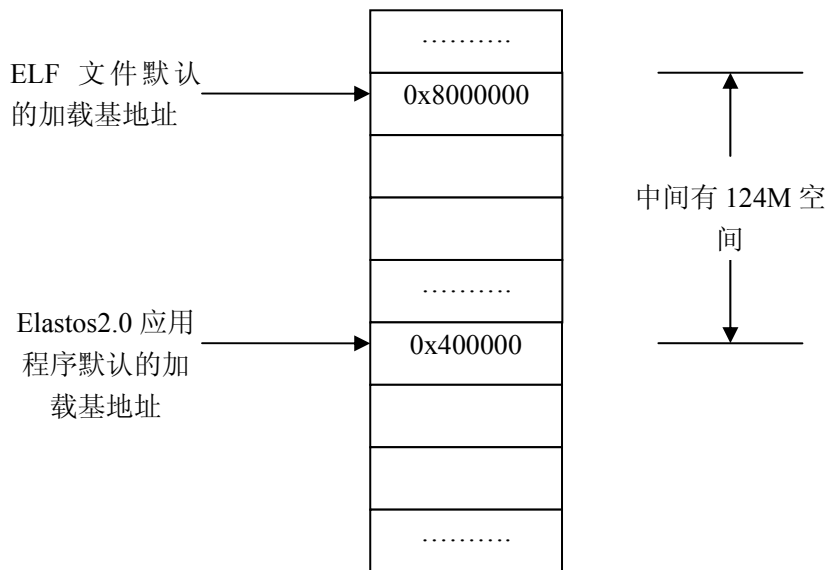


图 4.5 Elastos2.0 与 Linux 应用程序默认加载基地址

从图4.5可以看出，默认的Elastos2.0应用程序加载基地址与ELF文件的默认加载基地址之间有124M空间的虚拟内存没有使用（尽管在Elastos2.0应用程序加载地址之下的4M空间也没有利用，但这4M是传统保留区，只提供给系统相关的DLL使用，因此也可以不计算在内），Elanix加载器中就是利用了这124M未分配的内存，再通过使用mmap函数，从而实现了Elastos2.0编译出的Elastos2.0应用程序加载。其实如果应用程序不是很大，这124兆空间是足够的，而且这124M空间只是用来加载EXE的，对于DLL、CAR构件等，并不加载到这段地址空间。总的说来，采用mmap方式可以获得与Elastos2.0应用程序最为相似的地址空间布局。第二种方式的加载思路如图4.6。

图4.6例举了一个简单的Elastos2.0应用程序（该应用程序没有用到任何系统调用，也不涉及到任何外部依赖的DLL或so）的加载，其只有两个段：代码段和数据段，默认的加载基地址为0x400000，每个段大小为0x1000。加载后其在linux下的内存映象和Elastos2.0下的内存映象是相同的。因此，理论上加载后就可以从代码段中指定的入口点开始执行代码，同时由于加载文件的地址和默认值相同，无需考虑重定位的处理，因此加载完毕后，在执行的过程中可以访问已加载的所有段的内容，而无论其访问的地址是在代码段还是数据段。

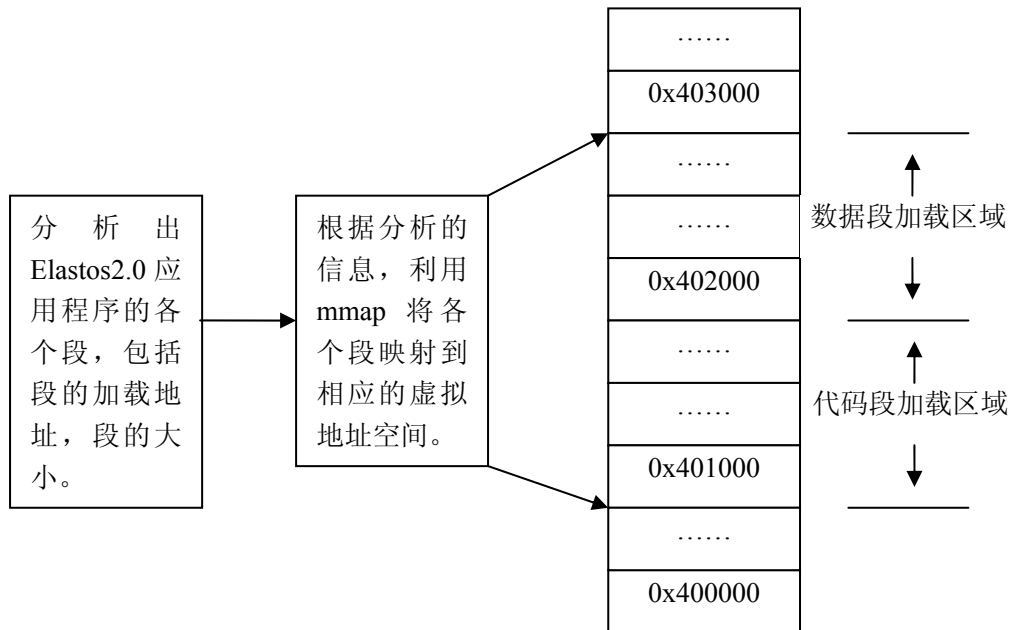


图 4.6 mmap 方式

采用mmap方式加载Elastos2.0可执行程序，可以避免重定位的处理，这在实际应用中是十分重要的，因为有些Elastos2.0应用程序为了减少文件长度，常常省略重定位信息，这样如果采用new或malloc来分配内存，就会遇到无法重定位的问题。mmap则不存在该问题。但采用mmap方式也有个问题需要考虑，即对于比较大的程序（如大于124M）将难以加载。但就Elastos2.0来说，其是针对嵌入式设备的，因此加载如此大的应用程序也比较少见，因为一般应用程序都需要DLL支持，而DLL则加载到其他的地址空间，因此对于一般的Elastos2.0 EXE程序来说，这124M空间是够用的。

4.1.5 Elanix 地址空间分布

通过上节的分析可以看出，Elanix加载器利用mmap来映射或预留地址空间，并将Elastos2.0应用程序映射到Elanix加载器本身所在的地址空间。但Elanix应用程序涉及到很多模块，除了将EXE程序加载到0x400000开始的地址空间，其还需要加载如DLL，CAR构件等其他PE模块。这样就涉及到Elanix加载器中地址空间如何分配的问题。为解决地址分配问题，结合Linux用户地址空间的分布，Elanix加载器采取如下三原则进行地址空间分配或映射：

一、一般将exe文件加载到从0x400000开始的地址空间（因为有些Elastos2.0

应用程序为了减小文件的大小，省去了重定位信息，这种情况下只能将其加载到0x400000开始的地址空间）。

二、对于DLL或CAR构件则可加载到任意可用的地址空间，在Elanix中将DLL/CAR的加载起始地址设为0x15000000，加载的具体地址，根据模块的大小逐渐累加，有关地址分配在后面还会详细介绍。

三、对于非PE格式的模块，如Elastos.so则通过Elanix加载器调用Linux系统函数加载到相应的地址空间。对于这些模块的加载，其加载地址不受Elanix加载器控制。

Elanix加载器加载Elastos应用程序和相应的DLL后，其地址空间分布如图4.7所示，其中灰色部分是加载Elastos应用程序和DLL/CAR文件的用户地址空间。

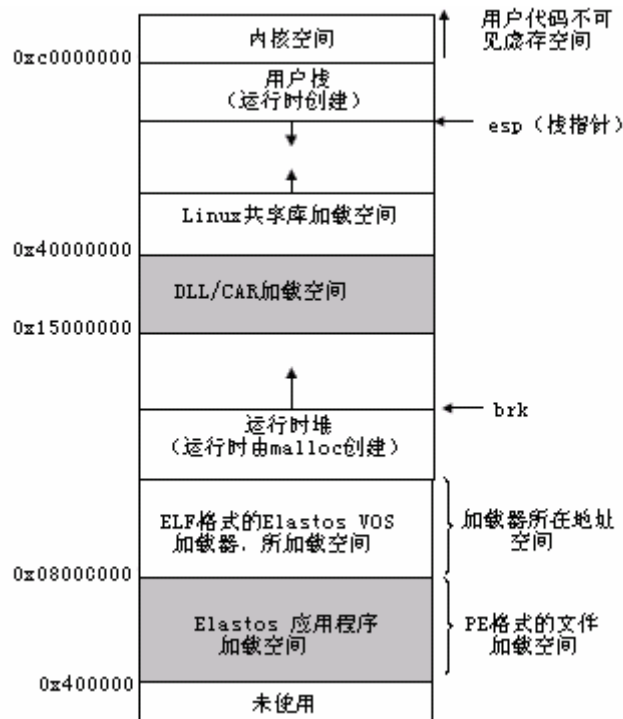


图4.7 Elanix加载器加载Elastos应用程序和DLL/CAR构件后地址空间的分布

4.1.6 小结

本节对Elanix加载器的设计原理，模式的转变，设计目标和架构，内存分配策略和地址空间分配进行了详细的阐述，为Elanix加载器的实现提供了理论依据和技术保障。

4.2 Elanix 加载器实现

在上一节中讨论了Elanix加载器设计过程中所解决的技术问题和加载的理论依据，并对地址空间分配原则已经进行了详细的分析。本节将探讨Elanix加载器具体的实现，采用的具体技术，对外部模块的支持模式，以及Elanix加载器加载的Elastos2.0应用程序实例。

4.2.1 PE 格式模块的解析和加载

加载Elastos2.0应用程序，首先需要解决的是解析Elastos2.0应用程序，也就是解析和加载PE格式的二进制模块，这是将Elastos2.0应用程序加载到Linux上运行的基础。在第二章，已经对PE文件格式有了详细的解释，如何将现有的PE文件加载到Linux地址空间，使Elanix加载器的一个主要任务。

PE模块中有很多节，其中最关键的段是：.txt，.data，.bss，这三个段是加载后模块允许所必须的，当然还有些符号表相关的段以及资源段等，这些段在程序链接和元数据获取时候还是有意义的，因此也需要加载以提高加载速度。在加载.bss段的时候尤其要注意，因为.bss段是未初始化数据段，该段在PE模块中的内容长度为0，但加载后的该段长度不为0，因此Elanix加载时需要预留出该段所需的空間。图4.8是Elanix加载器加载PE模块的流程图。

图4.8中灰色的方框是加载PE模块中各section映射到Linux地址空间的部分。在这部分中，通过mmap分配每个section所需的地址空间，然后将该section的内容拷贝到分配的地址空间，这样就完成了该section的加载，依次读取PE模块的每个section，并将其拷贝到对应的地址空间。由于采用mmap方式分配映射地址空间，因此可以指定各section的加载地址，各section的加载地址，可以通过如下公式计算得出。

$$\text{LoadAddr} = \text{imageBase} + \text{section.VirtualAddress};$$

其中LoadAddr为section应该映射的起始地址，imageBase是PE模块加载后的起始地址，section.VirtualAddress是被加载的section相对模块加载地址的偏移量。

由此可以看出，PE模块中真正需要加载到Linux地址空间的是PE模块所包含的各个section的内容，PE文件的头部和各种section表，只是为了帮助在加载时定位PE模块中section的位置，加载完毕后，这些PE头部所包含的信息也就不再需

要了。

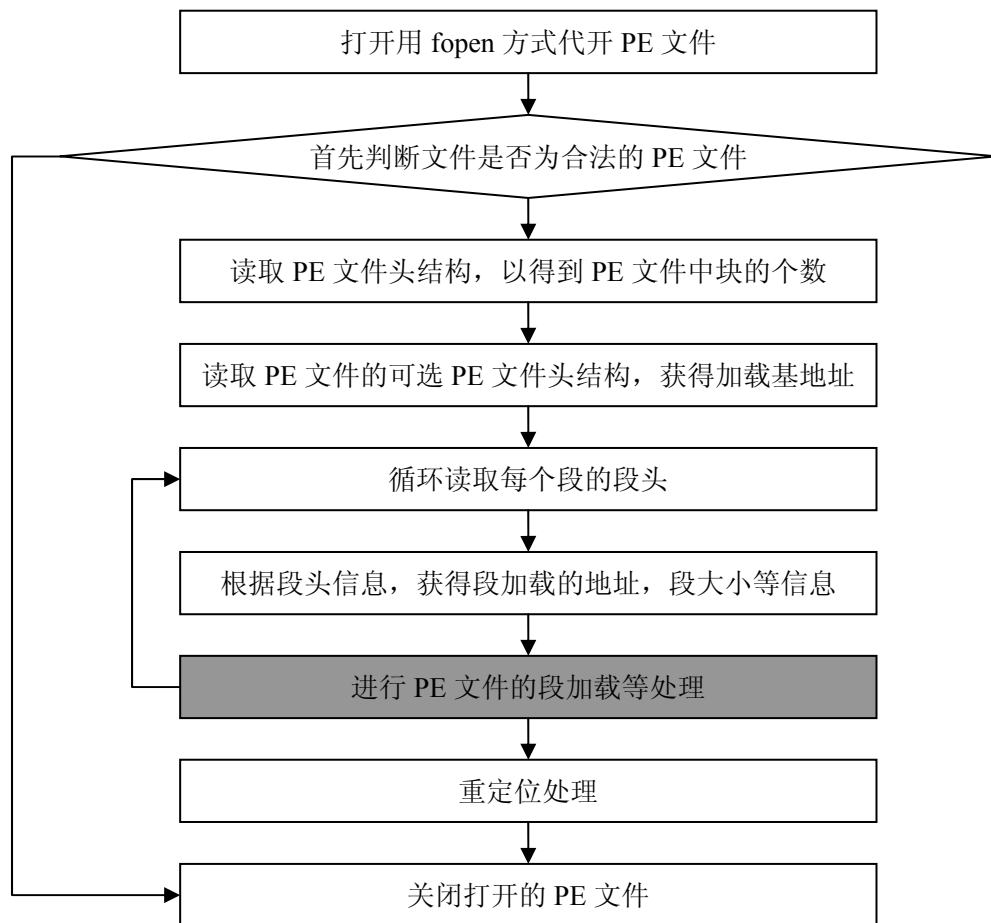


图4.8 PE段加载

4.2.2 模块加载和选择

Elanix加载器加载Elastos2.0应用程序，并不仅仅是将模块中的各个section映射到相应的内存地址就完成任务了，其还必须加载应用程序所依赖的其它模块，并根据各模块之间的关系进行相应的设置，只有这样，Elastos2.0应用程序才能在Linux上顺利运行。

Elanix将所有加载的模块分为两类：本地模块（Native Module）和内建模块（Build-in Module）。两者的定义如下：

本地模块（Native Module）：是指Elastos2.0上生成的模块，包括EXE和DLL，其好处是可以保持与Elastos2.0的完全兼容，一般是以“.dll”或“.exe”结尾。

内建模块（Build-in Module）：该模块是指Elastos2.0需要的，但由于系统调

用的原因，不能以本地模块的方式加载到Linux系统上的模块。这种模块在Elanix加载器中以“.so”结尾。为实现这种模块，必须在Linux上开发与Elastos2.0上模块相同的API，然后通过前面提到的模式转换，达到在Linux上支持Elastos2.0应用程序调用的目的。

在Elanix加载器加载所需模块的过程中，通过加载路径的选择，来找到对应的模块。模块加载规则如下：

首先，加载ealstos.so模块，并将其加入到模块链表中，这是因为：

第一，elastos.so为Elastos2.0应用程序运行、Elastos内核服务提供支持的基础平台，基本上所有的Elastos2.0应用程序都会用到该模块所提供的API；

第二，在elastos.so中的dllmain函数进行一些Elanix server相关的操作和设置，这些操作必须在应用程序加载之前完成；

第三，elastos.so 完成了一些环境变量的设置，如应用程序参数设置。因此将其作为第一个加载的模块，以在Linux上建立Elastos2.0应用程序的运行平台。

其次，是加载应用程序，应用程序加载到内存中后，分析是否有其他模块需要加载（根据模块的依赖关系）。加载过程如下：

（一）如果有外部模块需要加载，则先在已加载模块链表中查找该模块是否已经加载，如已经加载，则直接使用，找到相应的函数入口地址，设置相应的输入函数表；

（二）如有外部模块需要加载，但该模块在已加载模块链表中没有找到，则按照图 4.9 种的规则查找需要加载模块的路径（so 或 dll）。从图中的路径搜索可以看出 elanix 加载器的加载顺序是 dll 文件优先，即本地模块（Native DLL）优先，当前路径优先。这样做也出于对从移植的工作量，毕竟加载本地模块（dll 或 exe）比实现相同功能的内建模块（本地模块在 Linux 上的实现）要简单的多。找到模块加载路径后，进行加载，加载后找到所需的函数入口地址，完成相应的输入函数表设置，并将其添加到已加载模块链表中。

（三）EXE 或 DLL 所需的所有外部模块加载成功后才会设置 EXE 或 DLL 的外部输入函数表的指针，外部函数输入表是程序能否正常运行的重要保证。

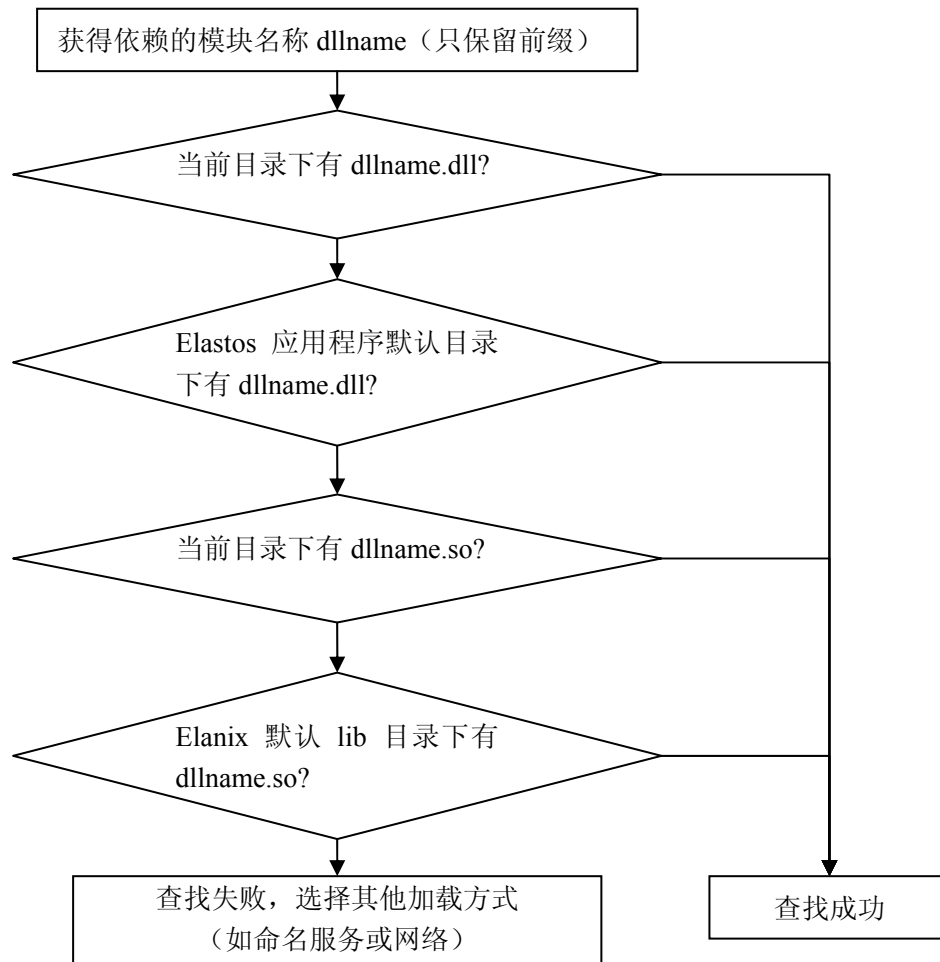


图 4.9 模块加载路径选择

4.2.3 模块加载方式

利用上面的模块路径搜索方式，从优先级不同的路径下查找对应的模块，找到后，即可进行应用程序的加载。加载的过程根据模块的性质采取不同的策略：对于PE格式的EXE或DLL文件，可以通过Elanix提供的函数LoadPEModule实现；对于ELF格式的SO文件可以采用Linux系统调用dlopen实现。PE模块的加载稍微复杂一些，因为这里涉及到地址空间影射，重定位，输入输出函数表的设置和读取，以及.bss段的初始化工作。可以看出PE模块装载的过程是个递归、嵌套的过程，这是因为应用程序或DLL模块在加载前，并不知道其所需的外部依赖模块，只有在加载该模块后，通过动态的分析其所需外部依赖模块名称，

再根据模块路径的查找规则，才能找到对应的模块并进行加载。总的模块加载过程如图4.10所示：

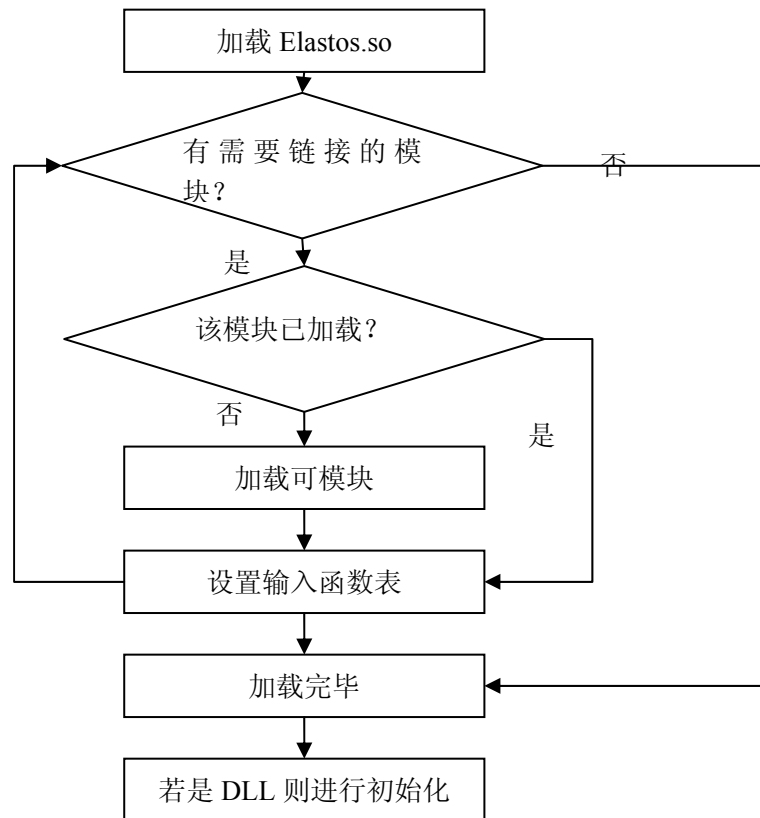


图 4.10 模块加载流程

模块加载的关键函数 `LoadModule()` 实现代码如图4.11。其中的函数 `UpdateIATAccordingIID()` 完成了递归调用模块加载函数的功能。函数 `LoadModule()` 是Elanix加载器实现的关键函数，其不仅仅用于EXE和普通DLL的加载，其还适用于CAR构件以及SO模块的加载。

```

BOOL LoadModule(char* ModulePath,...)
{
    ...
    MODULENODE *pModuleNode = new MODULENODE();
    MODULEINFO *pModuleInfo = new MODULEINFO();
    ReadModuleInfo(pModuleInfo); //读取模块信息
    Module_type = ImageModuleType (pModuleInfo); //获取模块类型
    if(Module_type == PE_MODULE) { //判读是否为合法的 PE 文件
        start = pModuleInfo->ldBase; //模块的加载起始地址
        for(i=0;i<pModuleInfo->section_num;i++){
            //将模块中各个 section 影射到对应的地址空间
            mmap((void*)start, pModuleInfo->addr, SizeOfRawData,
                FLAG, MAP_ANONYMOUS, -1, 0);
            //计算下一个 Section 开始的地址值
            start = start + SizeOfRawData;
        }
        //重定位
        DoRelocation(pModuleInfo);
        //设置输入函数表
        UpdateIATAccordingIID(pModuleInfo);
        //初始化 DLL
        if(pModuleInfo->type == TYPE_DLL)
            InitDll(pModuleInfo);
        pModuleNode->info = pModuleInfo;
        InsertNewModule(pModule);
    }
    else if(Module_type == SO_MODULE) { //判读是否为合法的 SO 文件
        pModuleNode->info = dlopen(ModulePath,RTLD_LAZY);
        InsertNewModule(pModule);
    }
    else{
        printf("Can't support module types!!\n");
        return ERROR;
    }
    return OK;
}

```

图 4.11 加载器模块加载代码示例

4.3 Elanix 加载器模块管理和地址空间管理

4.3.1 模块数据结构定义

Elanix对加载的模块进行了管理，模块管理是必要的：首先通过模块管理，可以记录已经加载模块的信息；其次，通过模块管理可以快速查找已加载的模块；第三，模块的管理对于应用程序的动态加载运行有重要的意义，如CAR构件的加载。Elanix采取链表方式来管理模块，每个加载的PE模块（EXE及DLL）或SO模块，都会在链表中添加相应的链表项，在加载新模块时，首先在链表中查找是否有模块已经加载，如果已经加载，则找到对应的链表项，并返回；否则，进行加载。所有的加载模块都保存在一个加载模块指针的全局链表中，链表和链表项的定义如下：

```
typedef struct _MODULEINFO {
    IMAGE_DOS_HEADER idh; //PE 模块 DOS 头部
    IMAGE_FILE_HEADER ifh; //PE 模块的 PE 头部
    IMAGE_OPTIONAL_HEADER32 ioh; //PE 模块的 PE 可选头部
    DWORD ldBase; //PE 模块在 Elanix 上的加载起始地址
    DWORD Memory_Size; //PE 模块占用的地址空间大小
} MODULEINFO, *PMODULEINFO;

typedef struct MODULENODE{
    char modulename[NAME_SIZE]; //模块名称
    DWORD moduletype; //模块类别
    INT count; //模块引用计数
    union{
        PMODULEINFO pModuleInfo; //PE 模块的加载信息
        void* pLib; //ELF 格式的 SO 模块指针
    }
} MODULENODE, * MODULENODE;

typedef struct _MODULE_LIST{
    PMODULENODE node;
    _MODULE_LIST* next;
} MODULE_LIST, *PMODULE_LIST;
```

上面是Elanix中对加载后模块的信息记录，以一个典型的Elastos应用程序HelloWorld.exe来说，其需要由Loader加载的模块包括以下三个：HelloWorld.exe，elacrt.exe，elastos.so。这三个模块在链表中应该具有如图4.12所示的关系。

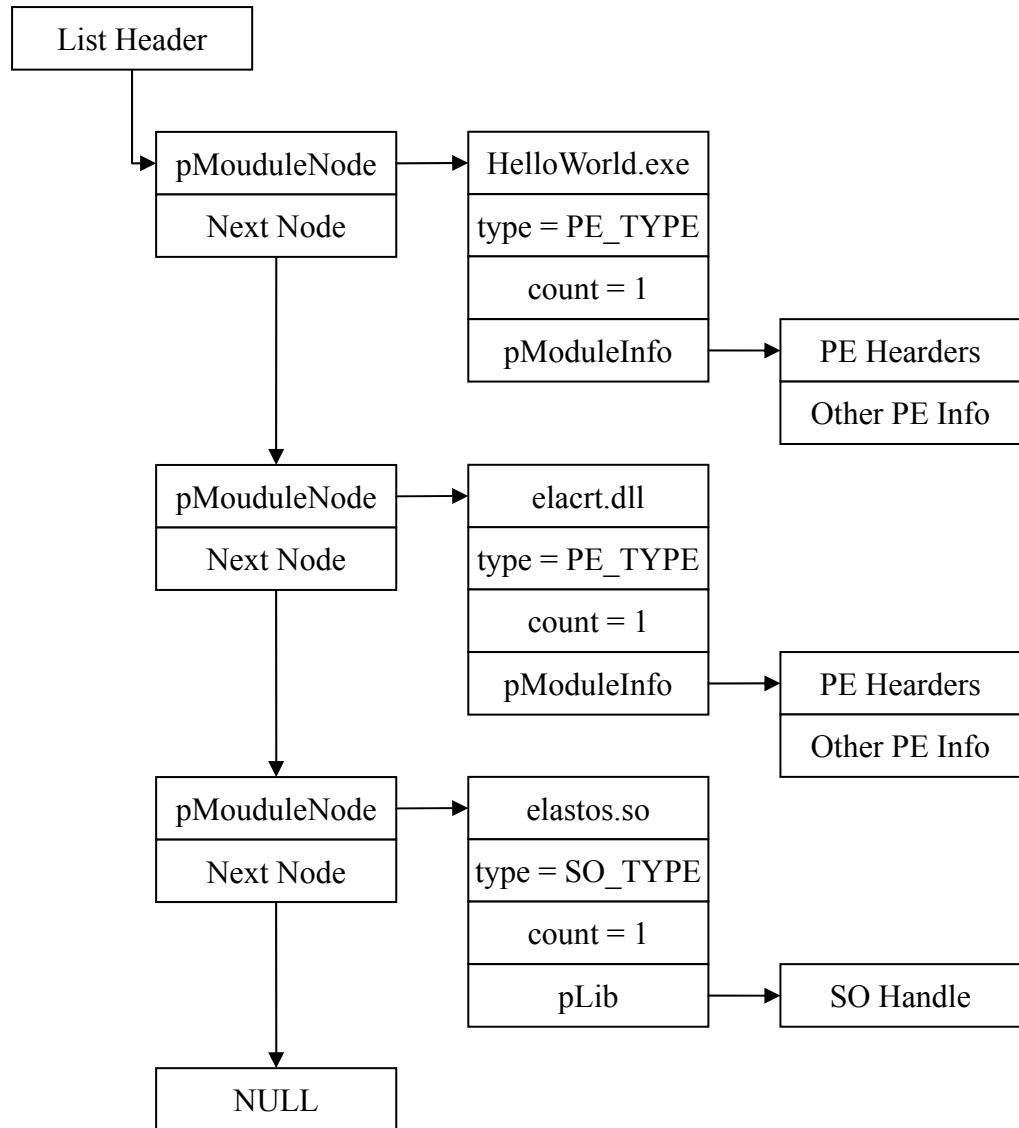


图4.12 Elanix加载后的模块链表

链表中的模块，对于ELF格式的SO模块而言，其只需保留通过dlopen系统调用得到的SO模块动态库的指针即可，而PE模块加载后保留信息要多一些，这是为了其他模块在加载相同模块时，这些信息可以减少不必要的读写磁盘模块的时间，从而提高模块的加载速度。

4.3.2 模块管理

Elanix加载器加载应用程序时是以模块（包括so，dll和exe）为单位进行加载的，每个被Elanix加载器加载模块信息都被保存在一个全局模块链表中，该模块链表的作用就是纪录所加载模块，快速定位模块，为其他需要加载的模块提供服务，因此有必要对加载后的模块链表进行维护。模块链表的维护包括增加模块，查找模块，模块生命周期管理，以及删除模块。模块在链表中的顺序并不重要。下面是模块链表常用操作：

增加模块：增加模块时，只需将加载的模块插入到链表开始即可，同时将引用计数设为1。此操作的时间复杂度为 $O(1)$ 。

查找模块：查找模块时根据模块名称在链表中查找，这里的模块名称是指不包含后缀的名称（如elastos.dll对应为elastos），找到后将查找到的模块的引用计数加1，并返回模块指针；没有找到，则返回空指针。此操作的时间复杂度为 $O(n)$ 。

模块生命周期管理：模块加载后则通过引用计数来控制模块的生命周期，当引用计数为0时，删除该模块所分配的地址空间，同时从链表中删除对应的链表节点。可以通过手动或自动地方式维护各模块的生命周期。此操作的时间复杂度为 $O(n)$ 。

删除模块：当需要删除被加载模块的时候，并不是立刻删除其对应的节点或分配的地址空间，而是首先将其引用计数减1，再判断是否为0，如果不为0，则该节点继续保存在链表中；如果为0，现将其所引用的其他模块的引用计数同时减1，然后，再通过生命周期管理的相关操作，将引用计数为0的模块彻底删除，这里的彻底指的是，链表节点的删除和地址空间的释放。

4.3.3 地址空间管理

Elanix对每个加载的PE格式的Elastos2.0应用程序，利用内存分配链表来管理地址空间。内存链表的作用是记录用户地址空间的使用情况，确定新加载模块的加载位置，并将回收的模块进行回收，以节省或重新利用地址空间。在地址空间的管理上Elanix参照了伙伴系统^[10]的分配和回收策略。Elanix上对于每个Elastos2.0应用程序，维护一个内存链表，该链表记录了已分配和未分配的内存块地址和大小。初始的时候，已分配表为空，未分配表有两个部分，其规则如下：

1, 第一块未分配内存区域是针对EXE模块而言, 块的起始地址设为0x400000, 大小为0x400000到0x7000000的之间的空间, 这是因为, 有些EXE程序没有重定位信息, 因此, 如果加载到0x400000以外的地方, 则不能正常运行, 而0x8000000是ELF程序默然的加载地址, 为了保险起见, 将EXE模块的上限设为0x7000000以避免冲突, 对于绝大部分Elastos应用程序来说, 这样的设计完全满足需要。

2, 第二块未分配内存区域是针对DLL模块而言, 该块的起始地址为0x15000000, 大小为从0x15000000到0x30000000的之间的地址空间, 每加载一个模块就根据加载模块所占用的空间, 修改内存分配表, 下次分配, 从上次分配结束后的地址开始, 以0x1000000取整, 连续分配。

3, 当通过Elanix加载器加载的模块被删除时, 两个链表会做相应的调整, 未分配的内存区的链表还会根据情况, 合并相连的两个或几个内存区域。

图4.13演示了HelloWorld.exe加载到内存后, 内存分配表的状态。

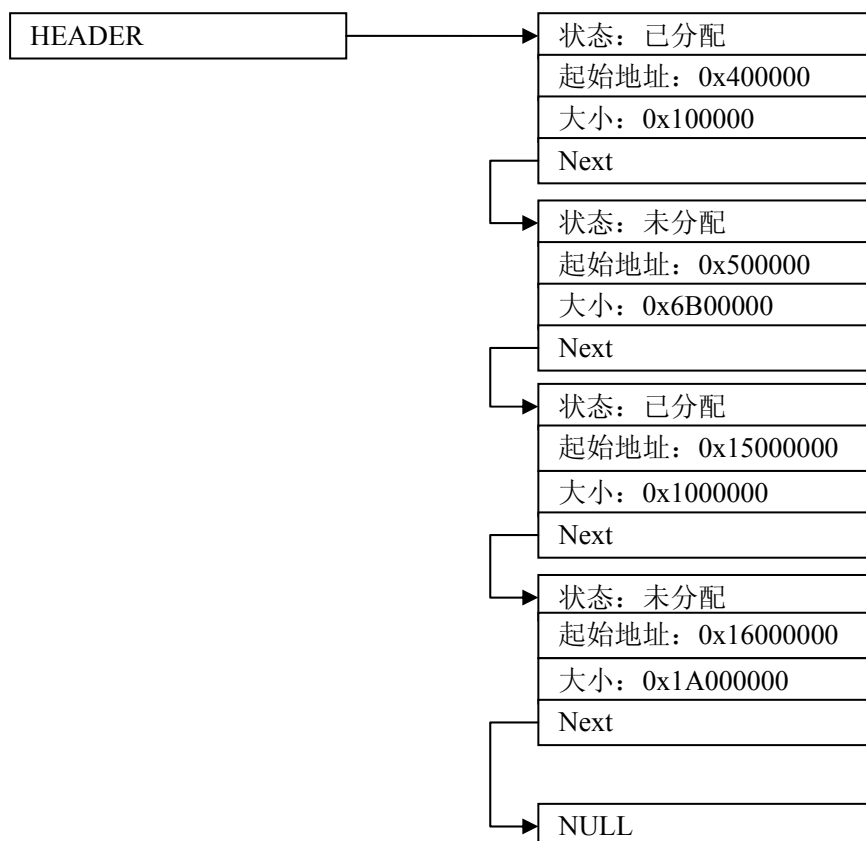


图 4.13 Elanix 地址空间分配链表

4.4 Elanix 加载器对图形、网络、文件模块的支持

Elastos2.0应用程序在运行过程中不可避免地需要设备模块的支持，如网络，图形，文件等模块。在Elastos2.0中这些模块也是通过Elastos.dll获得系统支持。本节将就在Elanix环境下如何支持Elastos2.0外部模块进行讨论，首先，先分析了Elastos下对外部模块的支持，而后结合Linux特点，设计了两种Elastos2.0外部支持模块在Elanix上加载的模式，最后综合两种模式的长处，提出Elanix外部模块的瀑布式加载模式。

4.4.1 Elastos 对外部模块支持

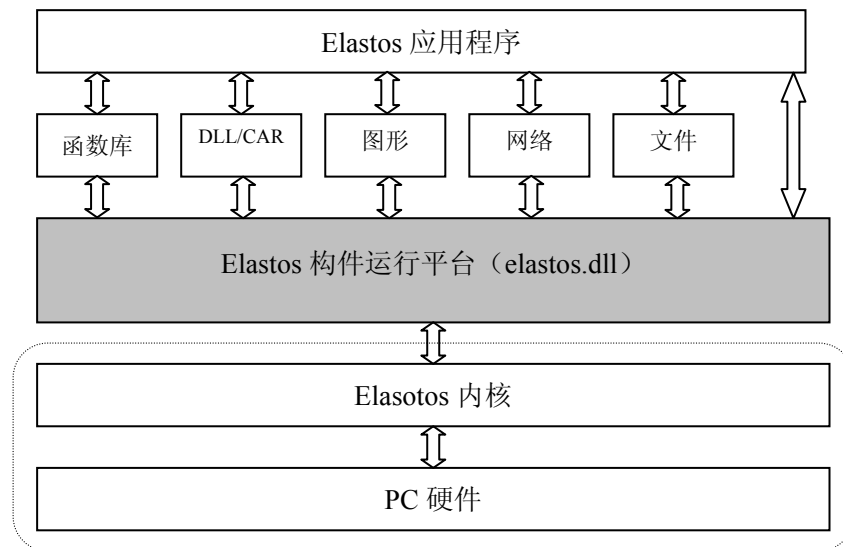


图 4.14 Elastos 上加载外部模块

Elastos对外部模块的支持如图4.14所示，Elastos上加载的外部模块也是通过Elastos.dll获得系统支持的。因此，Elanix模块对外部模块的支持，可以采取两种方式：本地模块（Native Module）模式和内建模块（Build-in Module）模式。

4.4.2 本地模块模式

第一种方式：即按普通DLL的方式加载外部模块。采取与Elastos相同的体系架构，将Elastos上外部模块直接通过Elanix加载器加载到Linux地址空间，由于Elastos的外部模块大都是DLL形式存在得，因此，采用本方式将这些外部模块加

载到Linux地址空间，同加载一般的DLL没有太多的区别，只需将Elastos上对应的模块支持文件（DLL），拷贝到Elanix环境的指定路径下即可。采用这种方式的好处是，简单，快速，体系结构上保持同Elastos的高度一致，但这样做也有缺点：一，对于移植同硬件关联较大的模块，还存在一些问题；二，对于在Elastos构架平台中尚未支持的外部设备或一些系统服务来说，这种方式也不可行，例如Elastos2.0的图形系统，其对图形硬件，消息机制，以及同步互斥等都有需求，因此在这种情况下，此方案对于图形系统来说并不是最佳方案。图4.15显示了这种方式下的Elanix的结构层次。

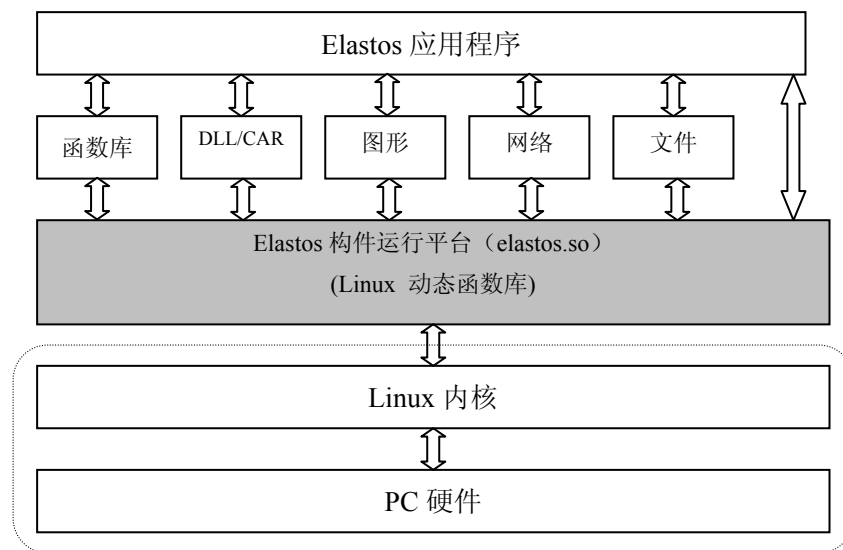


图 4.15 普通 DLL 模式实现外部模块加载

4.4.3 内建模块模式

第二种方式：即以SO方式实现对外部模块的支持，这种方式下，模块不再是以DLL的方式加载到Elanix系统架构中，而是以SO的方式实现与Elastos2.0中外部设备模块相同的功能，从而在Elanix上实现对Elastos外部模块的支持。在这种方式下，必须在Linux上重写Elastos对应的外部模块，并拥有相同的输出函数名称，以及相同的函数参数列表，保证其与Elastos2.0上的函数接口完全兼容。这种方式的好处是：可以利用Linux的系统调用，在更高层次上实现对底层的封装，并提供了Linux系统级的稳定性；还可以避免由于第一种方式中由于Elanix对Elastos2.0内核支持不足而造成的无法加载问题。当然本方案也有缺点：最明显

的缺点，就是需要进行大量的移植工作，这些移植工作包括，接口的实现，头文件的引入等，做了一些重复劳动，同时，采用本方式，打破了原来Elastos的系统架构，部分破坏了Elanix的系统完整性。图4.16显示了此方式下Elanix的结构层次。

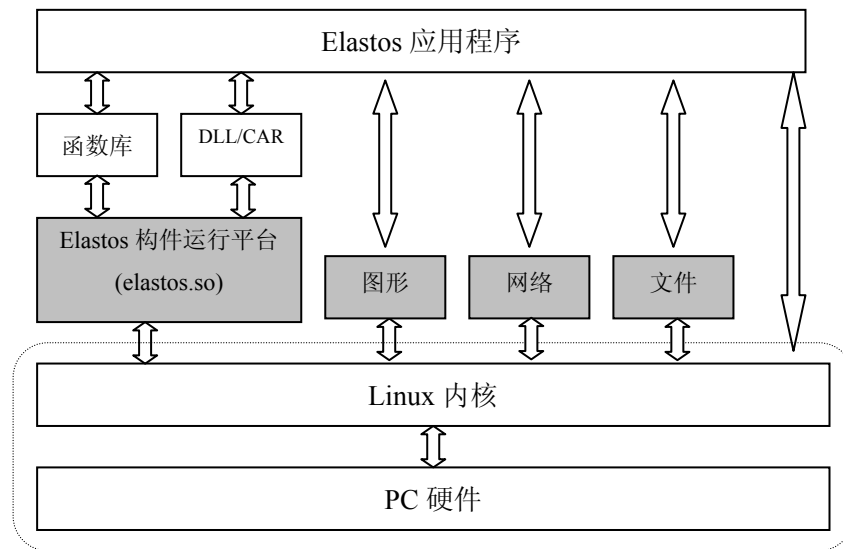


图 4.16 SO 模式实现外部模块加载

4.4.4 Elanix 瀑布式模块加载

目前Elanix针对不同的外部模块采取不同的加载策略，对于那些系统调用简单的模块，或者在Elastos2.0上的设备描述与Linux相近的外部模块采取直接加载的方式予以支持，如文件系统。而对于系统调用复杂，或Elastos2.0上底层对设备描述与Linux上差别较大的情况下，则采用第二种方式予以支持，如网络 and 图形部分。就目前加载的外部模块看来，其架构如图4.17所示。采取瀑布式加载方式的好处是可以根据情况对外部模块的实现方式做出选择，对模块的加载模式的多样性进行了有益的尝试。

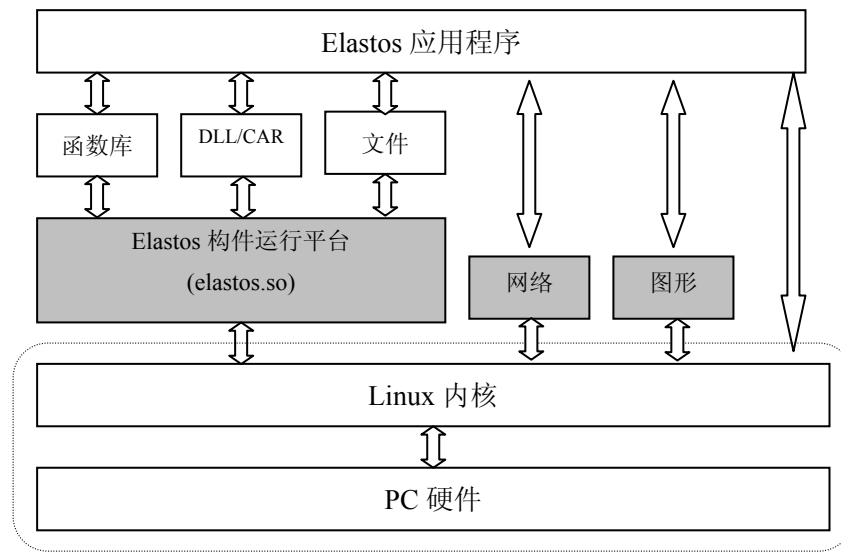


图 4.17 瀑布模式实现外部模块加载

4.5 Elanix 加载器加载实例及流程

下面的例子演示加载 Elastos2.0 上关于 Socket 通信的例子程序，包括 Server 部分（Tcpsvr.exe）和 Client 部分（Tcpcli.exe）。Server 和 Client 在加载过程中需要 Elastos2.0 的网络模块（elsocket.dll）的支持，还需要 C 运行时库的支持（elacrt.dll），其中网络模块是通过 elsocket.so 实现的，而 C 运行时库是直接将 elacrt.dll 作为本地模块加载到应用程序的地址空间。Server 模块和 Client 模块加载后的关系如图 4.18 所示。程序的运行流程如下：

首先，在 Server 端建立 Socket（输入命令“\$elanix tcpsvr.exe 1999”），并在指定的端口（本例中 1999）监听客户请求，当 Server 端受到 Client 端的服务请求后，发送信息，显示服务请求方的 IP 地址（本例中为 192.168.5.188），关闭 Socket。

其次，在 Client 端，其建立与 Server 的 Socket 连接（输入命令“\$elanix tcpcli.exe 192.168.5.188 1999”），获得并显示 Server 发过来的信息（“Hello!Are you fine?”），Socket 连接结束。

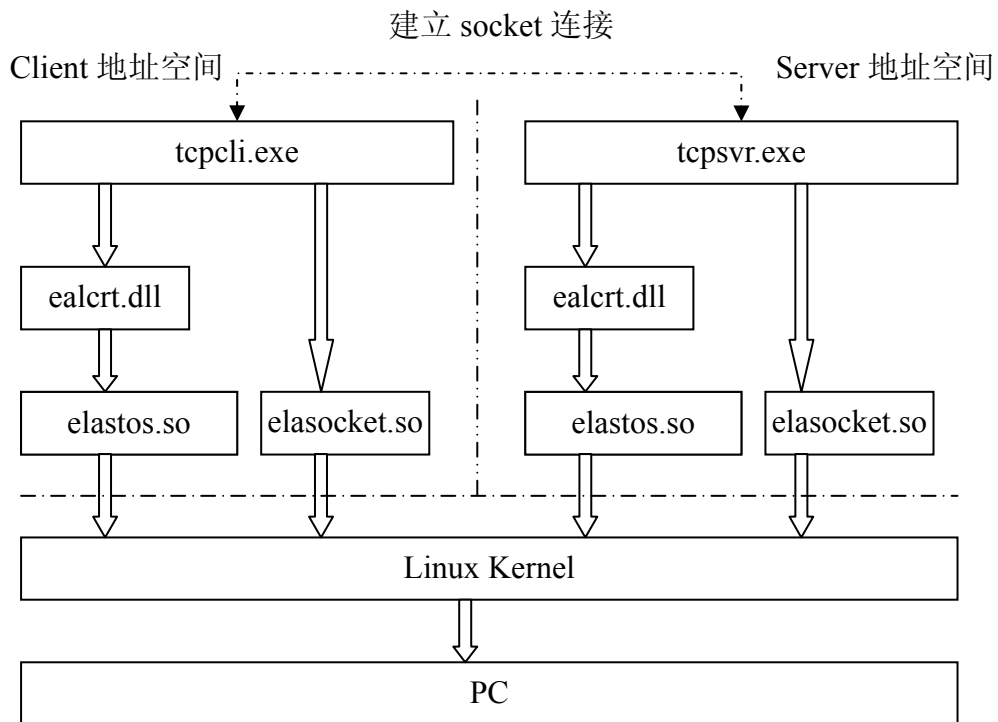


图 4.18 Elanix 上加载的建立 Socket 通信的示例

```
root@~/Elanix/elasamples/roadmap
File Edit View Terminal Go Help
[root@roadmap]# elanix tcpsvr.exe 1999
Server get connection from 192.168.5.188
OK!!
[root@roadmap]#
```

图 4.19 Socket Server 端运行结果

```
root@~/Elanix/elasamples/roadmap
File Edit View Terminal Go Help
[root@roadmap]# elanix tcpcli.exe 192.168.5.188 1999
I have received:Hello! Are You Fine?
[root@OSSUHANG roadmap]#
```

图 4.20 Socket Client 端运行结果

第 5 章 自描述构件加载与管理

5.1 构件加载

Elastos2.0 应用程序经常需要 CAR 构件支持，在第四章中介绍了 Elanix 加载器，其为加载 CAR 构件奠定了基础。因为 CAR 构件是以 DLL 形式存在的，也是符合 PE 规范的。理论上，将 CAR 构件加载到 Linux 地址空间和普通 PE 格式的 DLL 或 EXE 模块并没有本质的区别，也需要经过：各个 section 到地址空间的映射，重定位，输入输出表的设置和读取等步骤。尽管 CAR 构件是以 DLL 形式存在的，但 CAR 构件的使用和一般的 DLL 比起来确是大相径庭。其不同表现在以下几个方面：

一、CAR 构件加载的时机与非 CAR 构件 DLL 不同。非 CAR 构件 DLL 加载是在 Elastos2.0 应用程序加载运行之前，加载器通过分析非 CAR 构件 DLL 的输出函数表，定位到应用程序所需的外部函数依赖地址，并设置相应的应用程序的输入函数表，由此可以看出，非 CAR 构件 DLL 必须在 Elastos2.0 应用程序加载前完成，而不管应用程序是否用到了该 DLL，否则，无法找到 DLL 相应的输出函数地址，加载会失败。因此对于非 CAR 构件支持的 DLL，其加载时机相对应用程序而言是个同步的过程。而 CAR 构件的加载则完全不同，其并不是在应用程序加载之前加载，而是等到应用程序需要用到 CAR 构件时，才由加载器将其加载到相应的地址空间，因此，CAR 构件的加载是个异步的过程，如果应用程序根本没有用到 CAR 构件，则根本不需要加载改 CAR 构件。

二、CAR 构件的使用与非 CAR 构件 DLL 不同。非 CAR 构件 DLL 的使用时，通过 DLL 中相应的输出函数表，找到相应的函数入口地址，并将找到的地址填写到应用程序对应输入函数表的表项，从而实现 DLL 函数在应用程序运行前的查找并使用。而 CAR 构件是以接口方式对外提供服务的，分析 CAR 构件的输出函数表可以看到，每个 CAR 构件只包含下面两个输出函数：

`DllCanUnloadNow()`

`DllGetClassObject();`

`DllCanUnloadNow()` 用来决定是否可以卸载加载的 CAR 构件；
`DllGetClassObjec()` 用来创建 CAR 构件对象的类厂对象，通过类厂对象可以创建

类对象，并最终供应用程序使用。

第三点不同是 CAR 构件不仅可以作为单纯的 CAR 构件加载，还可以将其注册到 Elastos2.0 内核，以为其它应用程序提供服务，这在 Elastos2.0 上称之为命名服务。

第四点不同是，CAR 构件包含自描述信息——元数据。CAR 构件的元数据包括三大部分信息：构件模块信息、所有的构件类信息以及所有的接口信息。这些信息，为 CAR 构件的加载提供必要的认证信息，使得 CAR 构件的加载更为安全可靠。

第五，Elanix 加载器对 CAR 构件的加载只是将其加载到 Linux 用户地址空间，真正控制管理 CAR 构件的则是 Elastos.so 和 Elanix Server。因此 CAR 构件的加载更多程度上注重 Elastos.so 虚拟层，即 Elastos.so 上对 CAR 构件的支持，以及 Elanix Server 对进程间 CAR 调用的支持。大部分 CAR 构件的加载、调用、运行是通过 Elastos.so 提供的 API 完成的。

从上面几点可以看出，CAR 构件的加载比一般的 DLL 加载更加复杂，其涉及的层面也更多。本章就从几个方面探讨 Elanix 在 CAR 构件加载中所做的研究及所获得的成果。

5.2 进程内构件加载过程

5.2.1 进程内 CAR 构件

在 Elastos2.0 中，进程内 CAR 构件与客户程序在同一个进程地址空间里。CAR 构件并不直接输出其所包含对象的接口指针，客户程序必须借助 Elastos.dll 提供的 API 加载并获得 CAR 构件对象的接口指针。一旦，客户程序得到的接口指针，其就得到了指向 CAR 构件的接口 vtable，vtable 包含了接口的所有接口方法，这样客户程序可以直接调用 CAR 构件对象所提供的接口方法。可以看出，进程内 CAR 构件和客户程序的通信效率非常高，因为构件和客户程序都在同一进程空间内。当然，由于 CAR 构件直接运行在客户进程中，构件程序的严重错误有可能导致客户进程的崩溃，因此构件程序的稳定性以及在加载合适的构件版本，对客户程序的稳定运行意义重大。Elastos2.0 进程内构件示意图如图 5.1 所示。

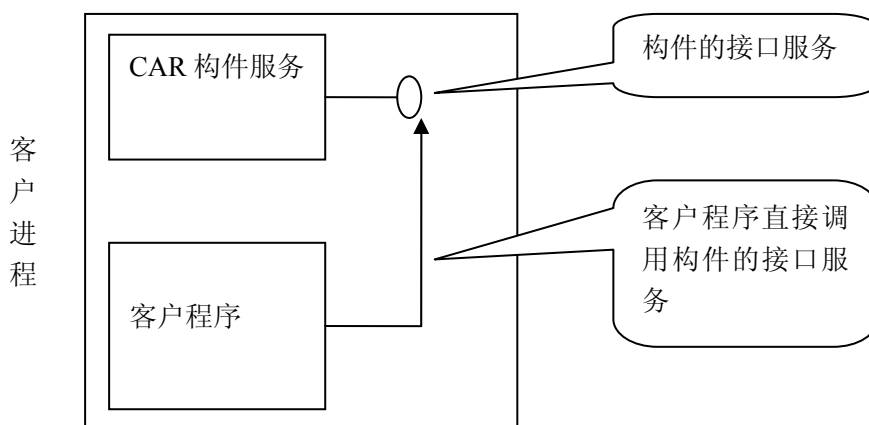


图 5.1 Elastos2.0 进程内构件

5.2.2 Elanix 加载进程内 CAR 构件

正如前面所述, CAR 构件并不直接对外提供对外的接口指针或构件的方法, 应用程序必须调用 Elastos.dll/Ealstos.so 中相应的 API, 创建所需的 CAR 构件对象, 获得对象指针, 这样才能享受 CAR 构件提供的服务。在 Elanix 上, CAR 构件也是以二进制格式加载到 Linux 地址空间的, 由于 CAR 构件在 Elastos2.0 上是 PE 格式的 DLL 文件, 因此 CAR 构件的加载同样需要通过 Elanix 加载器完成, 创建 CAR 构件的过程则同 Elastos2.0 上并没有太大的区别。

CAR 构件对象不能直接通过 new 生成, 而是由 CAR 构件对象所对应的类厂生成。因此为了创建 CAR 构件的对象, 必须先创建该对象所对应的类厂, 在通过类厂创建构件对象。

那么如何创建类厂对象呢? 答案就是 CAR 构件所在 DLL 中的输出函数之一: DllGetClassObject。每个 CAR 构件 DLL 都会包含该输出函数。DllGetClassObject 函数的作用是根据传入参数的 CAR 构件的 CLSID 创建相应的 CAR 构件类厂对象, 获得了类厂对象。有了类厂对象, 就可以通过其创建 CAR 构件对象实例。CAR 构件对象创建成功后, 将构件对象的指针 (即构件接口指针) 返回给应用程序, 这样应用程序就可以通过 CAR 构件对象的接口指针, 正常使用其所提供的功能了。

进程内 CAR 构件加载的工作流程如图 5.2。

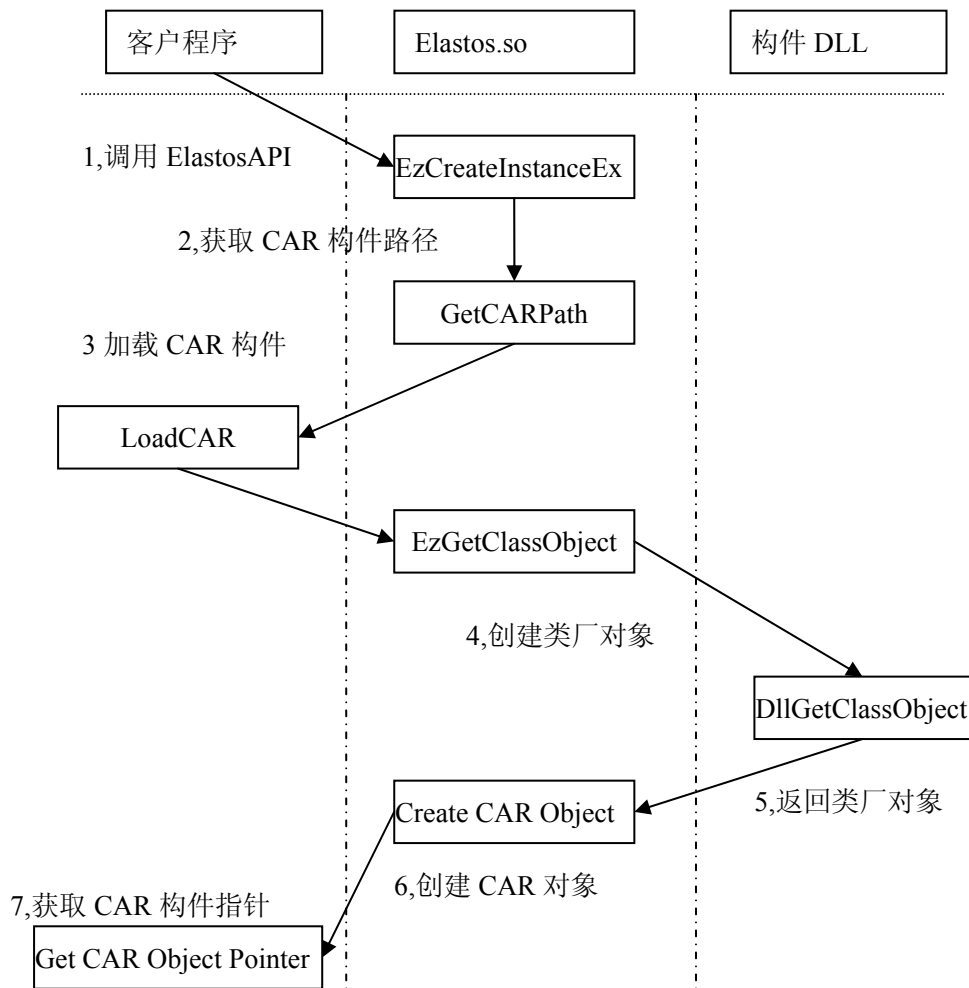


图 5.2 进程内 CAR 构件加载

从图 5.2 中可以看出，Elanix 上 CAR 构件对象的生成需要通过 Elastos.so 的 EzCreateInstanceEx 函数来创建，具体过程如下：

首先，EzCreateInstanceEx 函数内部首先根据传入的 REF EZCLS ID 参数，获取 CAR 构件的名称，通过默认路径搜索找到构件模块的加载路径。

其次，获得 CAR 构件路径后，从 Elastos.so 返回应用程序以调用 Elanix 加载器，并将 CAR 构件加载到用户的地址空间，加载完毕后，Elanix 加载器将控制权交给 Elastos.so 的 EzCreateInstanceEx 函数。

第三，EzCreateInstanceEx 函数继续进行 CAR 构件的创建工作。加载 CAR 构件后就可以获得 CAR 构件的输出函数：DllGetClassObject()，调用该函数即可创建 CAR 构件对应的类厂对象。

第四，创建好类厂对象后，就可以通过类厂对象创建相应的类对象，类对象创建成功后，就可以将类对象的指针返回给客户程序，同时退出 EzCreateInstanceEx 函数。

第五，客户构件根据 EzCreateInstanceEx 函数的返回值获得所需的 CAR 接口指针，调用构件接口函数，以完成相应的功能调用。

通过上面五步，CAR 构件就可以通过 Elanix 虚拟平台加载到与应用程序同样的进程地址空间。

5.3 进程外 CAR 构件互操作

5.3.1 Elastos2.0 进程外 CAR 构件互操作

Elastos2.0 中，进程外 CAR 构件与客户程序在不同的进程地址空间里，这样的 CAR 构件也称为远程构件服务。客户程序和远程构件服务在通信时，必须要跨越进程边界，这就需要解决下面两个问题：

- 1) 一个进程怎样调用另外一个进程里面的方法。
- 2) 参数如何从一个进程传替到另外一个进程。

先简单介绍 COM 是如何实现的。对于第一个问题，COM 采用了 LPC 和 RPC 来解决进程间通信问题，LPC 用于实现同一机器上的不同进程通信，RPC 用于实现不同机器上的进程通信。对于第二个问题，COM 采用列集\散集来解决进程间的参数传替问题。对于构件服务和客户程序，LPC 和 RPC 解决的是代理 DLL 和存根 DLL 之间的通信问题。

在 Elastos2.0 上，CAR 通过“和欣”内核实现存根和代理的通信，以解决不同进程间 CAR 构件调用的问题。对于第二个问题，CAR 构件运行平台自动解决 CAR 构件的列集\散集。

进程外 CAR 构件在和客户程序的通信时，CAR 构件运行环境会在客户进程创建代理对象，会为 CAR 构件服务启动一个服务进程并创建存根对象，对于客户进程而言，它只是和本地的代理对象通信，进程间通信的流程如下：

- 1). 客户端用户的一次 CAR 远程调用会转发到代理对象上
- 2). 代理对象将数据打包(列集), 并传给内核
- 3). 内核将打包的信息传给存根对象
- 4). 存根对象将数据解包(散集), 调用真正的构件服务接口函数
- 5). 服务构件接口函数完成调用, 并返回
- 6). 存根对象调用返回信息打包(列集), 并返回到内核
- 7). 内核将服务端的返回信息传递给客户端
- 8). 代理对象将数据解包(散集), 获得调用返回信息。整个远程构件方法调用过程完成。

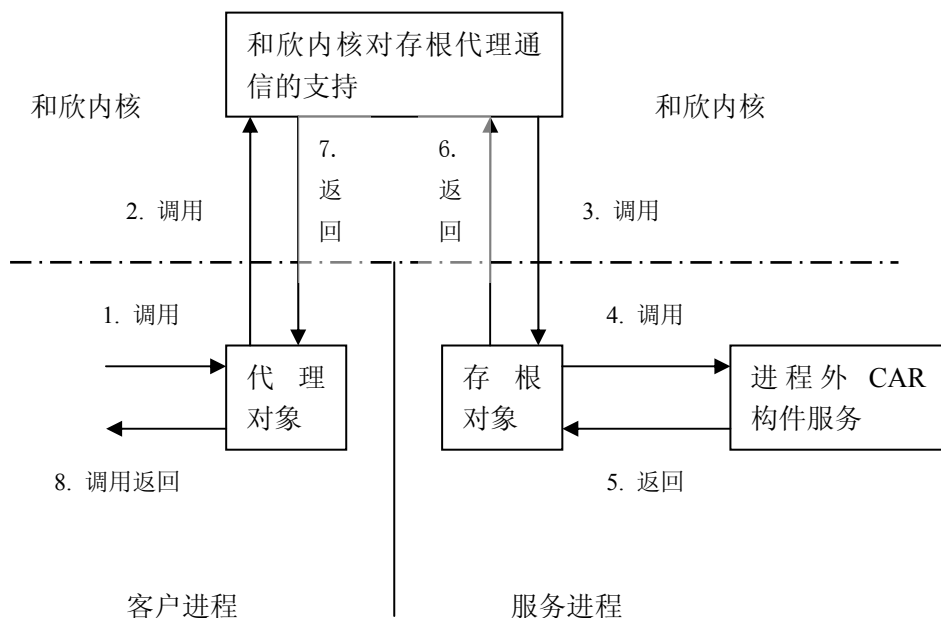


图 5.3 进程外 CAR 构件和客户程序

5.3.2 Elanix 进程外 CAR 构件互操作

Elanix 进程外 CAR 构件的互操作性的主要功能是由 Elanix Server 提供的, 而 Tunnel 是 Elanix 上加载的 Elastos2.0 应用程序与 Elanix Server 通信的通道。Elanix Server 主要划分为两部分: Tunnel (图 5.4 蓝色部分) 和 Major (图 5.4 中黄色部分)。

Elanix Server 被封装为一个 Linux 伪驱动。其在 Elanix 环境初始化的时

候被加载到 Linux 内核，其可以通过 Tunnel 模块，以字符设备文件的模式与用户程序通信，实现用户程序对 Elastos 内核的访问，并获得 Elastos 内核的服务。

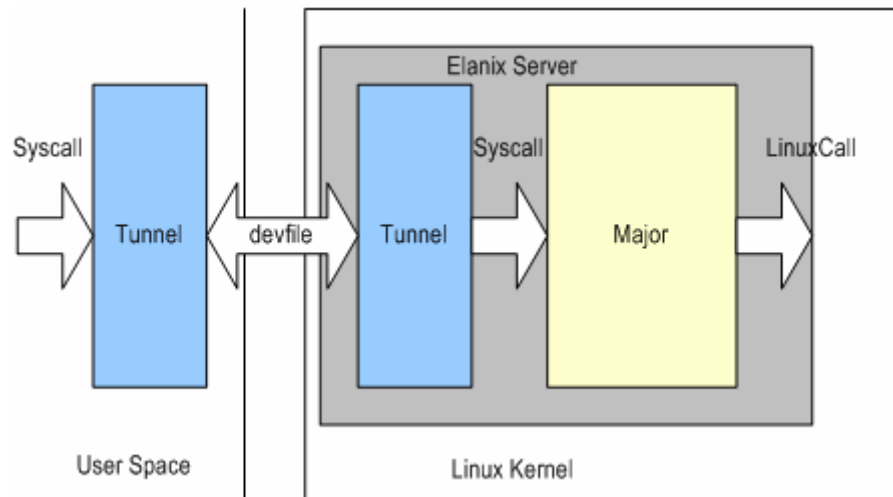


图 5.4 Elanix 上 Tunnel 和 Elanix Server 的关系

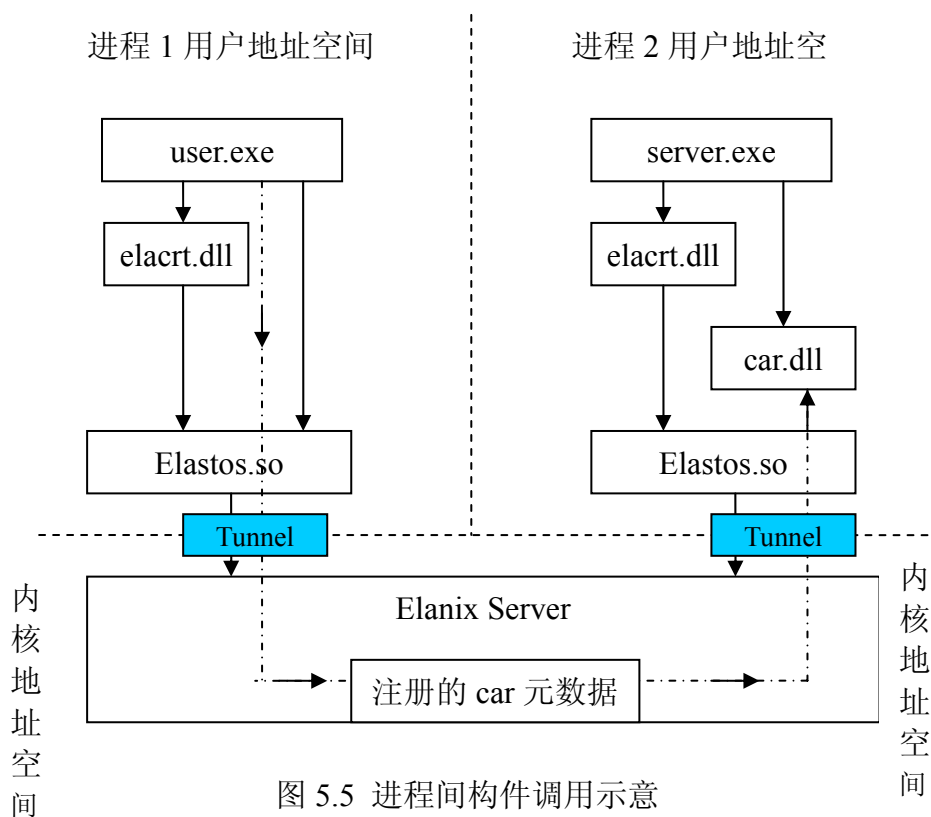


图 5.5 进程间构件调用示意

图 5.5 中有两个进程, user.exe 和 server.exe。其中 user.exe 进程需要访问 server.exe 地址空间中加载的 CAR 构件(car.dll)的功能。首先 server.exe 通过 Elastos.so 中 API EzRegisterService 函数向 Elanix Server 注册接口, 在注册接口前用户必须加载所需注册的 CAR 构件(car.dll), 并创建该构件的一个实例, 并将其做为参数传给该函数 EzRegisterService。在注册之前, 还必须保证已经获取该组件的元数据。由于一般的用户 CAR 构件都是以 DLL 形式存在的, 因此在加载 CAR 构件时, Elanix 会将被加载构件的元数据注册到 Elanix Server, 这样, 当下次需要调用同样构件时, 可用省去加载构件的开销。

user.exe, 即服务使用者, 则通过 Elastos.so 中的 API EzFindService 函数获得服务的接口, 从而获得服务。然后通过获得的接口进行完成相应的服务, 或者将该接口服务传递给其它使用者。

当 server.exe 决定要取消该项服务的时候, 它会通过 EzUnregisterService 函数注销该服务, 当服务被注销后, 其它用户如果通过 EzFindService 函数试图获取该服务都会失败。但是在服务提供者调用 EzUnregisterService 之前获得了该服务的接口的用户, 可以继续使用该服务, 除非该服务所在的进程已经退出。Elanix 上跨进程调用的 CAR 构件, 在调用过程中还涉及到参数和接口的列集和散集。

5.3.3 Elanix Server 性能分析

Elanix Server 是通过伪驱动实现与 Linux 上加载的 Elastos2.0 应用程序进行通信的, 这不同于 Wine, Wine 是通过 Socket 和其上加载的 Windows 应用程序通信的。两种方案各有千秋, Wine 方案的优点是: 实现简单, 可移植性好, 受 Linux 系统升级的影响小; 缺点是: 速度慢, 实时性较差, 需要在两个进程之间交换数据。Elanix Server 实现方式的优点是: 由于集成在 Linux 内核其响应速度快, 实时性好。但不足是: 实现较 Socket 方式复杂, 受 Linux 系统升级的影响大。Elanix 就两种方式的用户程序请求的响应时间进行了测试, 测试前在 Wine Server 和 Elanix Server 中各定义了一个新的空系统调用 NoOperation, 该系统调用请求不做任何处理, 直接返回空值。

测试硬件环境为: CPU 是 AMD 1.5G、内存 256M, 硬盘 80G。

测试软件环境为: 操作系统是 Red Hat Linux 9.0, 内核版本是 2.4.20-8, 编译器是 gcc 3.2.2, Wine 的版本为 20041019。

分别测试 Wine Server 和 Elanix Server 对用户程序一百万次、二百万次、三百万次请求响应时间。得到测试结果如下表所示。

测试次数	1 百万次	2 百万次	3 百万次	平均单次
Elanix Server 响应时间	0.579352 (ms)	1.158849 (ms)	1.737585 (ms)	0.579324 (μs)
Wine Server 响应时间	10.28146 (ms)	21.01983 (ms)	31.63047 (ms)	10.44497 (μs)

结果显示 Wine Server 的响应时间明显要大于 Elanix Server。这是因为：一，Elanix Server 是一个处于 Linux 内核空间的内核模块，响应用户程序请求不需要切换进程上下文；Wine Server 作为一个独立的用户进程，每次响应用户程序的请求，都必须在两个进程上下文间来回切换；二，用户程序与 Elanix Server 之间是通过设备文件互连，这种连接方式要快于 Wine Server 与用户程序之间的 Socket 连接方式。

由于 Elanix 的设计目的就是在 Linux 上实现和 Elastos2.0 应用程序在 Elastos2.0 上相似的速度，因此 Elanix Server 的选用内核模块作为通信机制。

5.4 CAR 构件元数据解析

5.4.1 CAR 构件元数据

每个 CAR 构件都包含元数据，元数据包含以下信息：构件模块信息、所有的构件类信息以及所有的接口信息，图 5.6 说明了 CAR 构件元数据信息的组成情况。

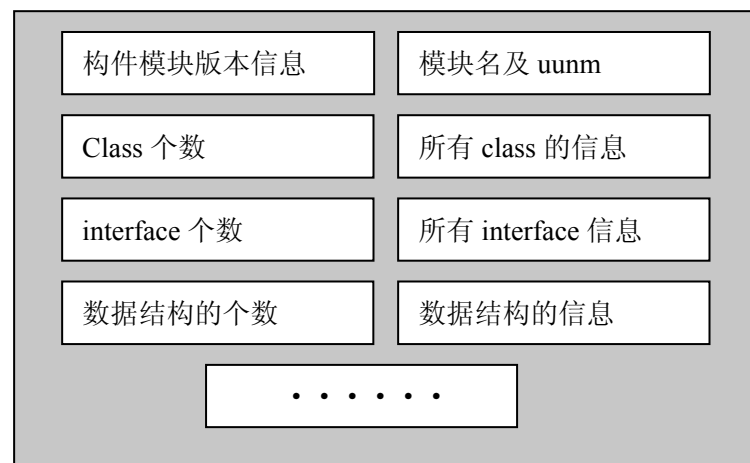


图 5.6 模块信息主要

对于每个 CAR 构件中的类信息，其包含的内容如图 5.7 所示。

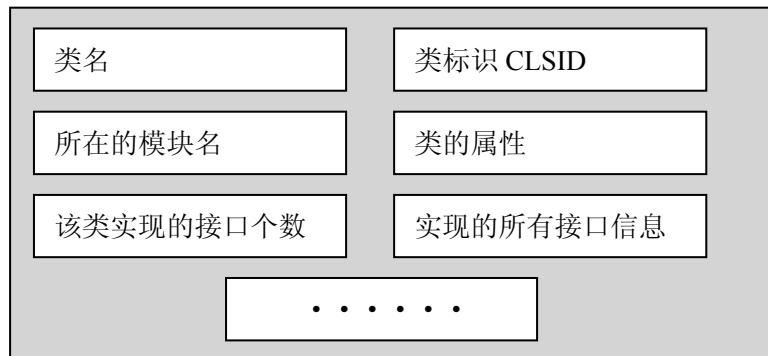


图 5.7 类信息主要构成

类信息里有所在的模块名，这是因为该类可能是其它模块的构件类；对于每个接口信息，其主要构成如图 5.8。

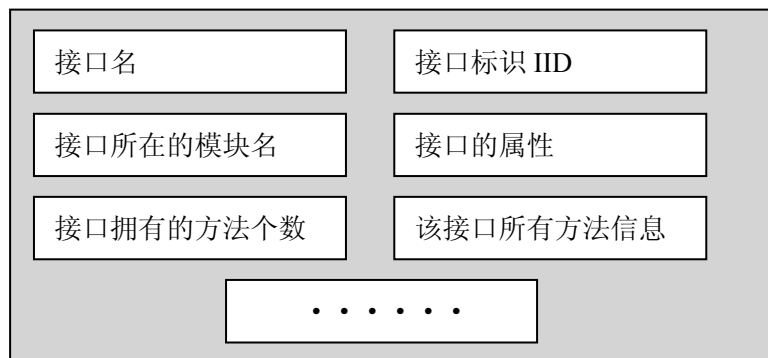


图 5.8 接口信息主要构成

接口也可以是另外模块定义，所以接口信息也记录了接口所在的模块名其中每个方法的信息结构如图 5.9。

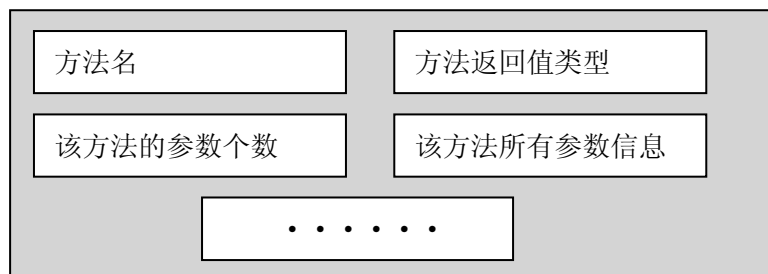


图 5.9 方法信息主要构成

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

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

从上面的分析可以看出，CAR 元数据提供丰富的构件自描述信息，其中的构件名称和版本信息，这在构件加载过程中十分重要，尤其是版本信息，其实构件加载中版本控制的依据。类和接口的元数据信息则为在 Elanix 上实现命名服务提供了支持。

5.4.2 CAR 构件元数据解析

CAR 构件库提供了几个 API 来得到上面提到的接口。

```
STDAPI EzGetModuleMetadata(  
    /* [in] */ POBJECT pObject,  
    /* [out] */ PMMETADATA *ppModuleMetadata);  
  
STDAPI EzGetClassMetadata(  
    /* [in] */ POBJECT pObject,  
    /* [out] */ PCMETADATA *ppClassMetadata);  
  
STDAPI EzGetInterfaceMetadata(  
    /* [in] */ POBJECT pObject,  
    /* [out] */ PIMETADATA *ppIMetadata);
```

在 Elanix 上也要实现对 CAR 构件解析的 API。上面几个 API 是通过 Elastos.so 提供的。其中 EzGetModuleMetadata 的功能是获得 CAR 文件的元数据；EzGetClassMetadata 的功能是获得模块的中类的元数据信息；EzGetInterfaceMetadata 的功能是获得模块中类接口的信息。

这些元数据的获取是保存在 CAR 构件数据段中的，因此加载 CAR 构件后，就可以获得 ClassInfo 元数据。按照 ClassInfo 元数据的保存格式，就可以获得关于构件版本，类，接口以及参数等的元数据。

这些元数据的解析获得对下面要讨论的 CAR 构件加载的过程中的控制有重要意义。

5.5 CAR 构件加载版本管理

5.5.1 Elastos2.0 中 CAR 构件版本管理

Elastos2.0 中对 CAR 构件加载有严格的限制，只有安全，可靠的构件版本才能被加载，因此 Elastos2.0 在加载 CAR 构件时会对加载的构件做出判断，鉴别将被加载的 CAR 构件是否与当前系统的版本兼容。Elastos2.0 是从无到有一步一步发展起来的，但发展过程中不可避免的要对系统更新升级，同样 CAR 构件的发展也是同步进行的，有些在 Elastos 早期版本上开发的 CAR 构件版本在新 Elastos 版本下可能无法正常运行，加载这样的 CAR 构件会引发错误，严重的还可能会引发系统错误。因此，构件加载中的控制是 Elastos 重点考虑的安全因素。

Elastos2.0 通过对对构件版本进行控制，从而加强构件加载的安全性。Elastos2.0 中，CAR 构件的版本采取四级编次方法，即：

格 式：	主版本号	次版本号	生成编号	修改编号
范 例：	2	0	8	3

以上版本号表示为：2.0.8.3。

主版本号标识一次重大的改变，构件的功能有比较大的改变或升级；次版本号标示构件的内容有一些改变或扩展，或者构件的某项功能有所改变或完善；生成编号标识构件的生成批次，如版本编译的次数；修改编号标示在两个生成编号之间，也即在构件的下一编译之前，对 BUG 的修改后得到的版本。2.0.8.3 表示 2.0.8 版本的第 3 次修改版。

Elastos2.0 对所加载构件版本采取如下规则

规则一：主版本兼容规则

主版本号被认定为是具有兼容性的版本，对于通常情况的应用程序而言，尽管所引用的构件可能已经升级或改变，但只要主版本号相同，就可以正确的加载运行。

规则二：默认最新规则

对于某应用程序，如果其所引用的构件已经有所升级或改变，“和欣”操作系统将在保证主版本号相同的情况下，默认加载次版本号、生成编号、修改编号均为最新的构件。

规则三：版本全同规则

如果在运用第一条和第二条规则之后，系统仍不能正确加载相应的构件，或者加载之后应用程序不能正确运行，以至出现某些功能上的错误，此时版本管理器会运用版本全同规则，也即在本地或指定的远程 URL 处寻找四级版本号完全相同的构件，如果存在，则加载此版本的构件

5.5.2 Elanix 中 CAR 构件版本管理

Elanix 中采取同 Elastos2.0 相同的构件版本策略。在 Elanix 环境中设置当前 Elanix 所支持加载的 CAR 构件版本号，加载 CAR 构件前首先通过解析元数据获得 CAR 构件的版本信息，再通过相应的 Elastos2.0 上的构件版本规则，对构件的版本进行判断，符合规则的予以加载，否则不加载相应的构件。构件加载的流程如图 5.10 所示。

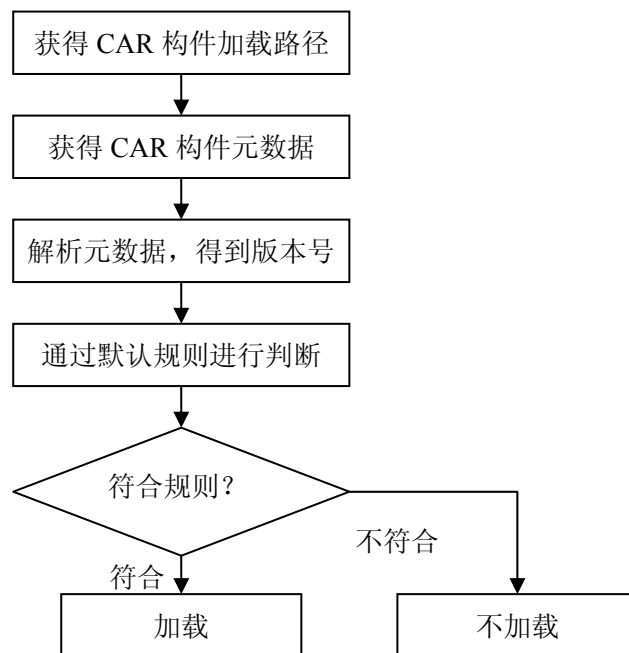


图 5.10 Elanix 构件加载中版本控制

Elanix 中 CAR 构件的版本控制主要解决了 Elanix 在加载 CAR 构件过程中对版本的控制，以决定所将要加载的 CAR 构件是被当前 Elanix 所支持，减少应版本不一致而引起的加载或运行错误。

5.6 Elanix 中 CAR 构件加载安全认证

构件的安全性已经是构件加载面临的主要问题，构件版本的控制可以解决构件加载过程中构件与系统兼容性的问题，但对于非法构件的加载只靠版本控制是不够的，因此有必要设计一种加载模式，使得CAR构件的加载与认证机制向结合，为CAR构件的加载提供安全保证。可以通过PKI^[11]机制实现对CAR构件加载过程中的认证。PKI，即公开密钥体系结构，最著名的符合PKI机制的加密算法包括RSA^[11]，Plhlig-Hellman，Rabin，McEliece等。PKI机制的基本思想是用一对密钥（公钥和私钥）对数据进行加密和解密，如通过公钥加密的数据，只有通过私钥才能解密，反之亦然。可以通过发放证书的方式发布公钥，而私钥一般不对外公布。

为了保证 CAR 构件加载的安全性，可用通过对可靠的 CAR 构件开发人员给以认证证书，其中包含了加密 CAR 构件安全信息所需的公钥，有了加密公钥，CAR 构件就可以将加密后的信息保存在元数据或资源段中，这些信息包括构件的加载环境，使用范围和有效期等。而私钥则虽保存在 Elastos 系统或 Elanix 系统中，加载 CAR 构件的时候首先通过元数据获得加密后的构件信息，并用私钥进行解密，解密后的信息如果符合要求则再行版本管理等控制，以决定是否加载，否则不予加载，因此包含构件认证后的构件加载过程如图 5.11。

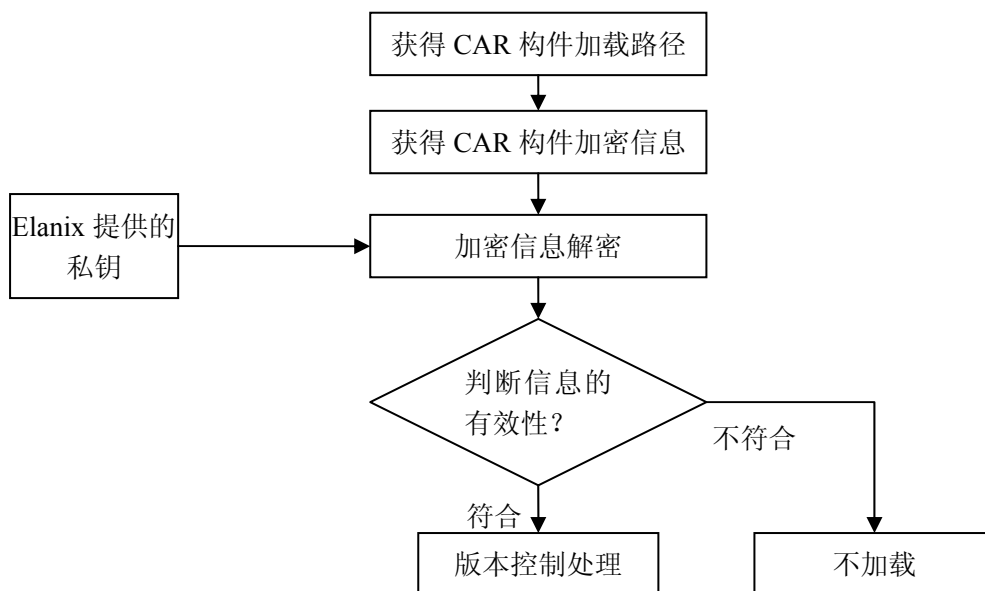


图 5.11 Elanix 构件加载中安全认证

目前 CAR 构件认证的方案还比较简单，并没有完全实现整个 PKI 机制，这主要是因为完善的 PKI 机制详细而且复杂，在 Elastos 发展初期，采取这种策略，不利于 CAR 构件的发展和普及，因此上面的解决方案即可以一定程度的构件认证功能，也为以后建立更完善的 CAR 构件认证机制提供了基础。

5.7 构件加载实例及流程

猜数字是经典的游戏，在 Elastos2.0 中有猜数字游戏的源码，其分为两个部分，主程序部分和 CAR 构件部分，在 Elastos2.0 上编译后，可生成符合 X86 体系架构的二进制可执行文件，分别对应为 (guess.exe 和 guess.dll)。主程序部分主要是控制输入输出，如读如键盘录入的数字，将猜测的结果以可读文字的形式输出到显示器，以及开始新游戏或退出；CAR 构件部分完成猜数字的逻辑部分，其功能包括：产生被猜测的数字，判断录入数字的有效性，将用户猜测的数字与答案进行比较并返回比较结果。两部分的依赖关系如图 5.12。在 Elanix 上通过 Elanix 加载器加载猜数字游戏相关的可执行文件并加载 CAR 构件。对于调用 CAR 构件方式可采用两种方式，即进程内和进程间，进程内构件在加载过程中需要考虑构件的版本控制，进程间调用 CAR 构件时必需先将 guess.dll 在 Elanix Server 中注册。具体的加载方式与前面介绍相同。

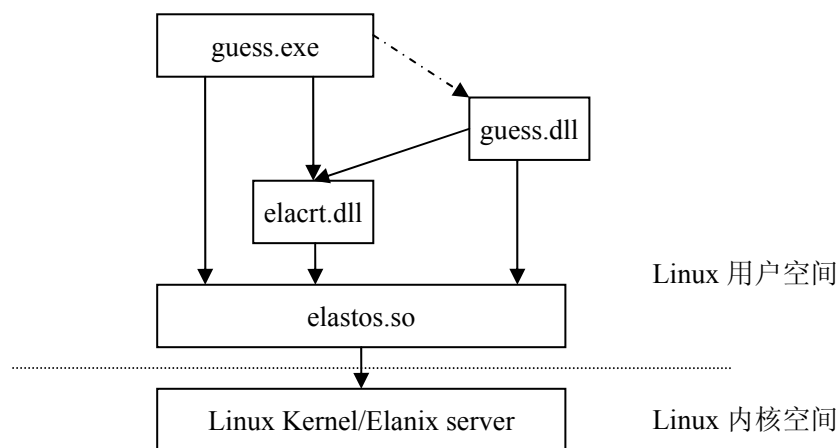


图 5.12 猜数字游戏之间的模块依赖关系

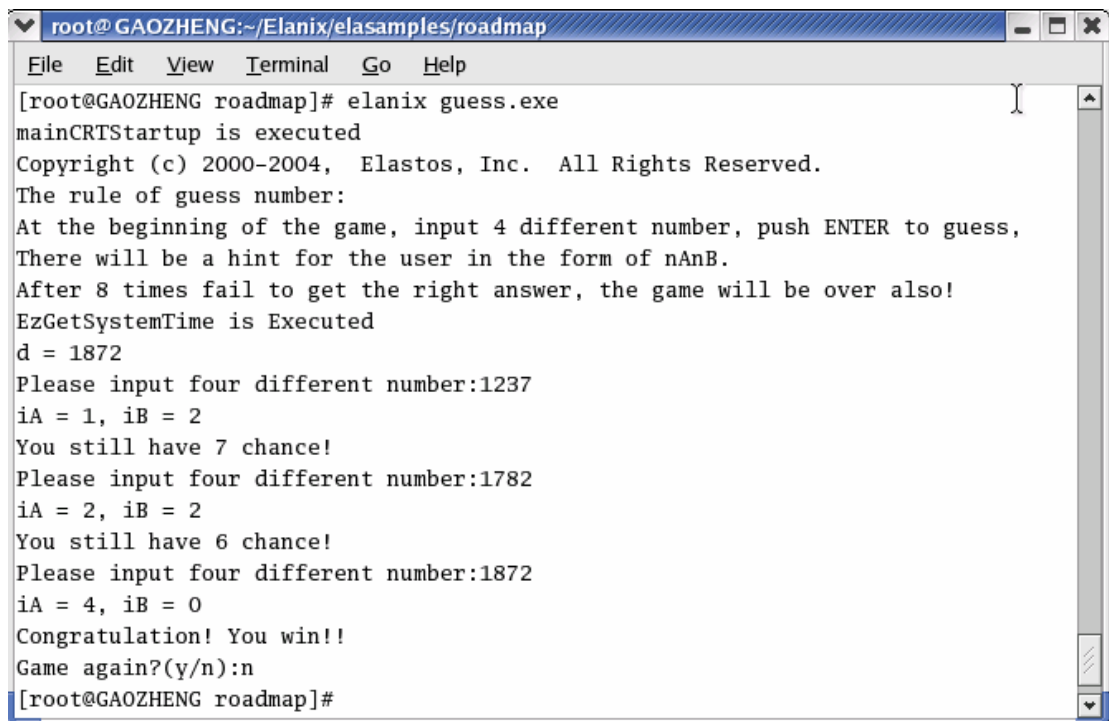
猜数字游戏的运行流程如下：

首先，通过 Elanix 加载器加载猜数字游戏应用程序（输入命令 “\$elanix guess.exe”）。

第二，猜数字游戏在加载过程中会使用 CAR 构件（“guess.dll”）提供的功能，如产生被猜的数字，判断用户的输入与结果是否相同，生成用户提示信息等功能。因此在用到 CAR 构件功能时，elanix 会加载（进程内）或查找（进程外）guess.dll，以获得 CAR 构件接口指针供应用程序使用。

第三，猜数字游戏中的主程序部分负责读取用户的输入（所猜的数字）和输出（提示信息）部分，而 CAR 构件则负责游戏的逻辑部分。

猜数字游戏的运行结果如图 5.13 所示。



```
root@GAOZHENG:~/Elanix/elasamples/roadmap
File Edit View Terminal Go Help
[root@GAOZHENG roadmap]# elanix guess.exe
mainCRTStartup is executed
Copyright (c) 2000-2004, Elastos, Inc. All Rights Reserved.
The rule of guess number:
At the beginning of the game, input 4 different number, push ENTER to guess,
There will be a hint for the user in the form of nAnB.
After 8 times fail to get the right answer, the game will be over also!
EzGetSystemTime is Executed
d = 1872
Please input four different number:1237
iA = 1, iB = 2
You still have 7 chance!
Please input four different number:1782
iA = 2, iB = 2
You still have 6 chance!
Please input four different number:1872
iA = 4, iB = 0
Congratulation! You win!!
Game again?(y/n):n
[root@GAOZHENG roadmap]#
```

图 5.13 猜数字游戏运行结果

第 6 章 总结和展望

6.1 总结

本文主要的研究内容是将 Elastos2.0 上运行的、二进制格式可执行程序 and CAR 构件，加载到 Linux 操作系统上运行的相关机制，具体包括以下内容：

(1) 综述了构件技术和跨平台技术，介绍了国内外研究背景，阐述了课题的研究意义。文章分析了 Elastos2.0 操作系统的框架结构，以及 CAR 构件的技术特点，并就构件的二进制级别兼容进行了深入剖析。

(2) 在分析 Linux 应用程序加载的过程和地址空间分布的基础上，结合 Elastos2.0 体系架构的特点，以及 Elastos2.0 应用程序加载的地址空间分布，将两个系统上的应用程序有机的结合起来，对 Linux 地址空间进行了新的划分。同时，利用 Linux 上的动态连接库应用程序的特点，对加载到 Linux 上的 Elastos2.0 应用程序的运行模式进行了改变——从 DLL 到 SO 的模式转变，并最终设计并实现了 Elastos 应用程序在 Linux 上的运行机制。

(3) 研究了如何将 PE 文件加载到 Linux 地址空间的方式，设计并实现了将 Elastos 二进制应用程序加载到 Linux 地址空间上的 PE 文件加载器。实现过程中涉及到：PE 文件的解析、不同模块之间关系的设定、加载模块的管理和对地址空间的管理、加载器对外部模块的支持、以及瀑布式的外部模块的加载模型，为在 Linux 上支持 Elastos2.0 CAR 构件奠定了坚实的基础。

(4) 研究了 CAR 构件加载的机制，基于 Elanix PE 加载器，设计并实现了在 Linux 上加载二进制 CAR 构件的加载方式，其中包括进程内 CAR 构件加载和进程外 CAR 构件加载；还研究了对加载过程中的 CAR 构件版本控制问题，并根据版本控制规则解决 CAR 构件版本的一致性；对构件的网络安全性也进行了探讨，并提出了有安全认证的构件加载模型。

通过对上述问题的研究，为 Elastos 应用程序和 CAR 构件的跨平台运行奠定了技术和理论的基础，也为未来构件跨平台技术的研究提供了借鉴和参考。

6.2 工作展望

在我们的坚持和努力下，Linux 上 Elastos2.0 的虚拟环境已经取得阶段性成果，其上不但可以加载普通的 Elastos2.0 应用程序和 CAR 构件，通过 Elanix Server 还可以进行进程间的 CAR 构件加载或调用。目前 Elanix 上支持的外部模块包括：网络模块和文件模块。应用与测试表明：我们设计实现的 Elanix 虚拟操作系统是一条切实可行的技术方案，也取得了丰硕的成果，现在项目还在进行当中，下一步的工作主要包括：

(1) 实现加载 Elastos2.0 上的图形模块，使得 Elastos2.0 的图形应用程序可以加载到 Linux 上运行。

(2) 完善 CAR 构件加载的安全认证机制，提高 CAR 构件加载的安全性。建立相应的网络环境下的 CAR 构件查找机制，以支持网络环境下的 CAR 构件加载和点击运行。

(3) 完善 Elanix Server 对内核对象的支持，提供信号量等同步互斥机制，使得 Elanix Server 成为 Elastos2.0 Kernel 在 Linux 上的完整实现。

参 考 文 献

- [1] “和欣 2.0” 资料大全, 北京科泰世纪科技有限公司, 2004.10.
- [2] CAR 技术简介, 科泰世纪科技有限公司, 2004, 11.
- [3] Don Box,Chris Sells, Essential .NET Volume 1: The Common Language Runtime, Pearson Educaton, 2003.
- [4] Don Box, Essential COM. 3rd Edition. Addison-Wesley. January 1998.
- [5] Enterprise JavaBeans TM Specification,version2.1. <http://java.sun.com/products/ejb>.
- [6] Uwe Bonnes, Jonathan Buzzard, Zoran Dzelajlija et al. Wine Developer's Guide, <http://www.winehq.org/site/docs/wine-devel/index>, 2004, 11.
- [7] Microsoft Portable Executable and Common Object File Format Specification, 1999.
- [8] Executable and Linking Format Spec v1.2, TIS Committee, 1995.
- [9] 陈榕, 苏翼鹏, 杜永文等。构件自描述封装方法及运行的方法, 中国专利, 1514361, 2004.07.21.
- [10] William Stallings. Operating Systems Internals and Design Principles, Fourth Edition, Prentice-Hall, inc. 2001.
- [11] Bruce Schneier. Applied Cryptography Protocols, algorithms, and source code in C, John Wiley & Sons, Inc, 1996.
- [12] OMG.CORBA Components version 3.0 . 2002.
- [13] Wolfgang Emmerich,et, al, Component Technologies : JavaBeans, COM, CORBA, RMI, EJB and the CORBA Component Model. Proc. of the 24th International Conference of tware Engineering, 2002.
- [14] Java TM 2 Platform Enterprise Edition Specification version1.4. <http://java.sun.com/j2ee/docs.html>.
- [15] SUN Microsystems. Java 2 Platform, Standard Edition Guide. <http://java.sun.com/>
- [16] 潘爱民, COM 原理与应用, 清华大学出版社, 1999.ISBN 7-302-02268-2.
- [17] Microsoft Corporation. .Net Framework Guide. <http://msdn.microsoft.com/library/>
- [18] McCune, Mike. Integrating Linux and Windows. Prentice-Hall. December 2000.
- [19] Nadelson, Mark/Hagan, Tom. Making UNIX and Windows NT Talk: Object-Oriented Inter-PlatformCommunication. Miller Freeman / R&D Books. December 1999.
- [20] Jason Pritchard. COM and CORBA Side by Side: architectures, strategies, and implementations, Addison Wesley Longman, Inc, 1999.

- [21] Gregory Brill. Applying COM+, New Riders Publishing, 2001.
- [22] 韦乐平, 薛君敖, 孟洛明等, OMG.CORBA 系统结构、原理与规范, 北京:电子工业出版社, 2000.
- [23] Brent Rector, Chris Sells. ATL Internals, Addison Wesley Longman, Inc, 2001.
- [24] J. E. Smith, Ravi Nair. Virtual Machines: Architectures, Implementations and Applications. Morgan Kaufmann Publishers, 2004.
- [25] Molay, Bruce. Understanding Unix/Linux Programming: A Guide to Theory and Practice. Prentice-Hall. November 2002.
- [26] Wine Weekly Newsletter. <http://www.winehq.com/site?news=archive>, 2005, 1.
- [27] De Goyeneche, J.-M. De Sousa. Loadable Kernel Modules, E.A.F. Software, IEEE. Volume 16, Issue 1, Jan.-Feb, 1999, P65 -71.
- [28] Peter Jay Salzman, Ori Pomerantz. The Linux Kernel Module Programming Guide. <http://tldp.org/LDP/lkmpg/2.6/html/>, 2005,1.
- [29] Randal E. Bryant, David O'Hallaron. Computer Systems A programmer's Perspective, Prentice Hall, Inc, 2003.
- [30] Michael Franz. Dynamic Linking of Software Components, Computer, March, 1997, P74-81.
- [31] OMG. Common Object Request Broker Architecture : Core Specification version 3.0[S].2002.
- [32] Ji HyunLee, Cheol JungYoo, etal. Analysis of Object Interaction during the Enterprise JavaBeans Lifecycle Using Formal Specification Technique [J].ACMSIGPLANNotices,2002. Addison Wesley,1997.
- [33] 高靖, 苏杭, 陈志成, 王小鸽, Elanix 虚拟操作系统中加载器的设计与实现, 计算机工程与设计, 录用待刊.
- [34] 苏杭, 高靖, 陈志成, 王小鸽, Elanix 内核对象服务通信机制的设计与实现, 计算机工程与设计, 录用待刊.
- [35] 胡玉杰, 李善平, Windows 程序在 Linux 上的运行, 计算机工程, 2003 年 7 月 P169-170.
- [36] 陈志成, CAR 构件的点击运行机制, 2004.10.
- [37] 陈志成, CAR 构件的版本管理方案, 2004.10.
- [38] 《CAR 命名服务机制》, 北京科泰世纪科技有限公司, 2004.10.
- [39] Søren Peter Nielsen, Kjetil Meidell Andenæs, Kevin P. Smith, COM Together — with Domino.
- [40] Abbott, Doug. Linux and Embedded and Real-time Applications. Newnes. March 2003.
- [41] Eddon, Guy/Eddon, Henry. Inside COM+ Base Services. Microsoft Press. October 1999.
- [42] Elanix 小组. Elanix 项目开发文档, 清华大学操作系统与中间件研究中心, 2005.10.

- [43] EFL 组件规范, 清华大学操作系统与中间件研究中心, 2005.
- [44] Baker, Sean. CORBA Distributed Objects: Using Orbix. Addison-Wesley. November 1997.
- [45] Silberschatz, A., Galvin, P. Operating System Concepts. Addison-Wesley, 1998.
- [46] 张斌, 高波。Linux 网络编程, 清华大学出版社, 2000.
- [47] Larry L.Peterson, Bruce S.Davie. Computer Networks: A systems Approach Second Edition, Morgan Kaufmann Publishers, Inc, 2000.
- [48] 罗云彬。Windows 环境下 32 位汇编语言程序设计。电子工业出版社, 2002。
- [49] 尤晋元, 史美林等。Windows 操作系统原理。机械工业出版社, 2001。
- [50] 彭礼孝。驱动程序开发起步与进阶。人民邮电出版社, 2000。
- [51] 李安渝等。Web Services 技术与实现, 国防工业出版社, 2003。