Elanix 项目综合报告（一）

Elanix 虚拟操作系统和构件技术的研究与应用

作　　者	苏　杭
相关人员	陈志成　苏杭　高峭　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 项目的苏杭的硕士研究生论文整理而成。

目 录

第1章　引　言

1.1 国内外研究背景

上个世纪八十年代以来，构件技术迅猛发展。这为大规模软件协作开发、软件标准化、软件共享和软件运行安全机制提供了坚实的基础。构件技术与以往编程技术最大的不同点在于程序模块之间的关系不再是静态的，而是动态的。也就是说用构件技术将程序拆分各个可以独立部署的模块，在程序运行时可以动态加载和动态替换。构件技术的一个目标是构件具有平台无关性。一个开发好的构件，理想情况下，应该可以在任意操作系统，软件平台上独立部署，正常运行。但是构件技术最初并不能达到这个目标，比如说 COM，就是与 Windows 操作系统紧密联系。二进制的 COM 构件起初只能在 Windows 平台上运行。但是随着虚拟机技术的发展，构件技术进入了中间件时代。

虚拟机技术的主要特点就是屏蔽系统环境，为程序的运行提供一个虚拟的环境。虚拟机技术有许多的应用领域，比如虚拟机技术可以解决系统更新换代，向下兼容问题，可以用来研究检测病毒等等。但是在中间件技术领域，虚拟机的作用是为构件提供一个虚拟的运行环境，让构件具有跨平台的性能。目前流行的 Java 编程技术和.Net 编程技术都是虚拟机与构件技术相组合的典范。

Java 编程技术有自己的虚拟机和编程语言。Java 语言是与Java 虚拟机的规定紧密相关的[1]。Java 虚拟机可以运行在Windows、Linux 等多种操作系统和硬件平台上。而用Java 语言开发的应用程序和JavaBeans 构件都运行在Java 虚拟机之上[2]。程序员在开发Java 程序时，不需要关心具体运行平台的差异性。这是因为Java 虚拟机已经屏蔽了不同平台之间的差异性，提供了统一的虚拟运行环境[3]。正是由于Java 虚拟机给Java 应用程序及Java 构件提供的这种跨平台的特性，才使得Java 技术迅速在网络编程领域普及开来。网络中存在着大量异质的系统，如果使用传统的编程技术，必须考虑各种不同的操作系统、硬件环境、协议，开发分布式程序将会非常复杂，而且开发出来的程序的维护和复用都很困难。如果使用Java 编程技术，在统一的Java 虚拟机平台上，程序员可以开发出简单、可复用的程序和构件。这极大的促进了Java 程序和Java 构件的发展。

同样的，微软的.Net 技术也有自己的虚拟机和编程语言。.Net平台是以公共

语言运行时(Common Language Runtime ,CLR) 为核心[4]。不同编程语言的源程序在.Net平台上首先经编译器被翻译为Microsoft 中间语言(简称IL) 。这是一组可以有效地转换为本机代码且独立于CPU 的指令。最后将包含IL 代码的EXE或DLL 文件保存到磁盘上。CLR在可以执行中间语言(IL) 代码之前，将通过实时编译器(J IT ,Just - In - Time) 转换为本机代码(本机代码是指运行于J IT编译器所在的同一计算机结构上的CPU 特定的代码) 。所以.Net平台也可以屏蔽操作系统、硬件环境等等因素，为程序和构件提供统一的运行环境[5]。与Java相比，.Net支持多种编程语言，可以跨语言集成。

1.2 课题的目的和意义

本课题研究虚拟操作系统技术[6]以及在虚拟操作系统环境中的构件技术，以文件加载、API转接、内核模拟为主要特征，结合Elastos操作系统和CAR构件技术特点，在Linux操作系统上设计并实现了一个虚拟操作系统Elanix。

Elanix虚拟操作系统的主要目的为CAR（Component Assembly Runtime）构件运行在Linux上提供虚拟统一的运行平台。CAR 构件技术作为新一代中间件技术之一，必然要实现网络环境下，跨平台部署，分布式运行的功能。但是目前CAR 构件主要是在Elastos 上运行，同时在Windows 上也可以运行。Elanix虚拟操作系统的实现将使得CAR 构件能在Elastos、Windows、Linux 三种操作系统上运行。这将促进CAR 构件标准的推广。Elanix虚拟操作系统使CAR 构件标准更适用于网络环境下的编程需求。普适计算要求程序运行具有游牧式的特征，即用户和计算均可以按需自由移动[7]。Elanix虚拟操作系统允许Elastos应用程序和CAR构件可以在多个操作系统间自由移动，为普适计算下CAR构件运行特征（如点击运行、按需下载）提供支撑平台。

从 Linux 操作系统角度来看，Elanix 虚拟操作系统的实现为 Linux 带来的一种新的构件规范即 CAR 构件规范。如同 Java 与 Linux 的结合为 Linux 操作系统带来了先进的软件框架和大量应用程序，CAR 构件技术与 Linux 的结合也必将丰富 Linux 的应用。考虑到 CAR 构件技术的特性更加适用于嵌入式领域，Elanix 虚拟操作系统的实现对于 CAR 构件技术在嵌入式 Linux 上推广有借鉴意义。同时对于 Linux 服务器来说，Elanix 虚拟操作系统也扮演着 CAR 构件运行平台的

先行者的角色。

此外，由于使用 CAR 标准和 Elastos API 开发的应用程序，可以在不做任何修改、不重新编译的情况下，运行在 Elastos、Windows、Linux 三种操作系统上。虚拟操作系统技术缩小了应用软件平台间移植的周期，提高的软件复用性，节省了人力、财力。

虚拟操作系统是虚拟机技术的一部分，所以 Elanix 操作系统也是对虚拟机技术一次新的尝试，在二进制代码级上，实现跨 Elastos、Windows、Linux 三种操作系统的功能。这也将为虚拟机技术发展积累经验。

1.3 报告各部分的主要内容

本文所涉及的主要工作 Elanix 虚拟操作系统属于 863 项目“面向普适计算环境的构件化操作系统研究”的组成部分。作者参与了 Elastos 操作系统和 CAR 构件技术的研发，完成了 Elastos 操作系统移植工作（包括从 x86 平台移植到 ARM 平台，VMWare 虚拟机上运行等）。Elanix 虚拟操作系统是本文介绍的重点，作者负责整个 Elanix 虚拟操作系统的构架设计，设计并实现了 API 转换层和内核对象服务模块两大部分，参与了加载器，Elanix 图形系统、Elanix 文件系统和 Elanix 面向对象编程 AOP 机制的设计工作。为了确保 Elanix 虚拟操作系统项目的质量，作者还搭建了一整套 Elanix 开发环境和测试环境。

论文第一章是研究背景与内容组织的相关介绍。

第二章主要介绍虚拟操作系统的概念和构件技术的发展。具体分析了当今典型虚拟操作系统 Wine 的技术和应用，并对构件技术，特别是 CAR 构件技术进行了详细说明。

第三章主要根据 Elanix 虚拟操作系统需要实现的目标，设计了 Elanix 虚拟操作系统的体系结构，并对 Elanix 中的主要模块 PE 加载器、API 转换层、内核对象服务模块一一给出了说明。

第四章主要研究如何在 Elanix 虚拟操作系统环境中实现对 CAR 构件技术的支持。其中包括的关键技术有：四层通信层次、以类为单位的列集散集机制、命名服务、远程对象生命周期管理等。

第五章主要介绍了 Elanix 虚拟操作系统的开发与测试方法。在对比了 Wine 与 Elanix 之间结构差别后，对两者进行了性能测试比较，并分析了测试结果。

第六章对全文进行了总结和前景展望。

第2章 虚拟操作系统与构件技术

2.1 虚拟操作系统

2.1.1 虚拟机的定义

简单地说，虚拟机是就是一台假想的计算机，它能执行特定的指令集，它使得程序的执行就好像是在一台实际的计算机上执行一样[8]。我们说“虚拟”二字，有着两方面的含义：

(1) 运行一定规则的描述语言的机器并不一定是一台真实的以该语言为机器代码的计算机，比如 JAVA 想做到跨平台兼容，那么每一种支持 JAVA 运行的计算机都要运行一个解释环境，这就是 JAVA 虚拟机；

(2) 另一个含义是运行对应规则描述语言的机器并不是该描述语言的原设计机器，这种情况也称为仿真环境。

虚拟机的概念包含有许多层次的内容。严格意义上的虚拟机指将一序列标准的指令代码转换成为特定的机器语言的程序。由这样的定义出发，虚拟机好像是指编译/汇编器，实际上汇编语言就是最初的虚拟机。通过汇编器的作用它使得好像是汇编语言直接在物理机器上直接运行一样，从而构成了一台“汇编语言虚拟机”，只不过我们不这样叫而已。

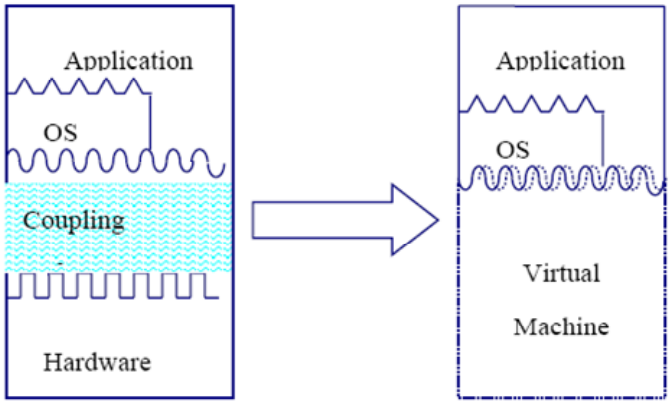

图 2.1 虚拟机原理

虚拟机的主要应用是使编写的运用程序能够适应多个不同的硬件平台，使程序员不必考虑各个硬件平台的差异。从而使编写的程序具有跨平台的特点，

增加了程序的通用性。虚拟机可以用图 2.1 表示。

虚拟软件系统放置在底层硬件和应用程序之间，它将应用程序的指令转换成为底层硬件的 ISA（Instruction Set Architecture），从应用程序的观点看来它好像直接运行在它所支持的硬件之上。也就是直接运行在由虚拟软件形成的虚拟机上。

2.1.2 虚拟机的分类

所有的计算机系统都是由三个主要的部分组成的：硬件、操作系统、应用程序。这三个主要的系统部件堆叠在一起的层次反映了他们之间的直接交互关系。操作系统由于具有管理硬件资源的特权，所以应用程序只需通过系统调用直接操作硬件，而不必自己去直接管理与操纵硬件资源。

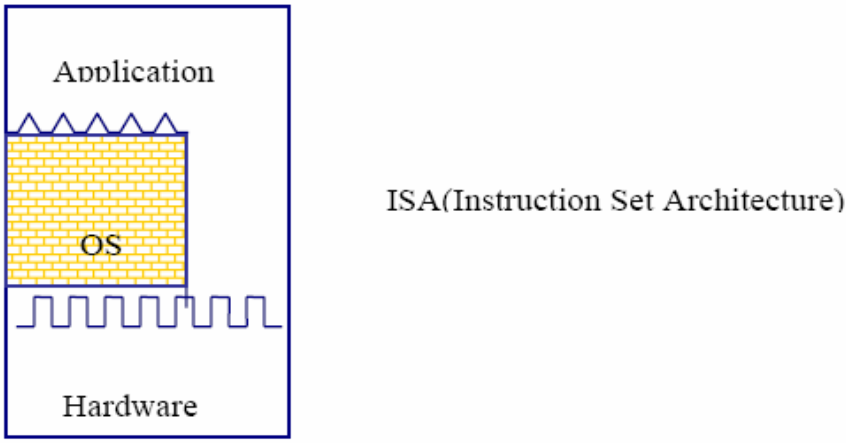

图 2.2 计算机系统层次示意图

随着计算机技术的发展这种分层模型的运用形式也产生了一定的问题。由于每一种运用只能在适当的“组合”条件下才能正常运行，所以产生了软、硬件的兼容问题。在某种 ISA（Instruction Set Architecture）下编译的软件不能在其他 ISA 下运行。例如，能在苹果 Macintosh 机上运行的软件不能在基于 Intel 处理器的计算机上运行，所以当 Macintosh 计算机早已使用 GUI 好几年后，基于 Intel 处理器的 PC 机还在等待 Windows 3.0 的推出。

另一方面在底层硬件结构相同的情况下，由于应用程序对操作系统的依赖，所以不同操作系统下的应用程序也存在兼容性的问题。例如，Windows 下的应

用程序不能在 Linux 中直接加载运行。所以，我们在 Linux 中我们还暂不能使用金山词霸。

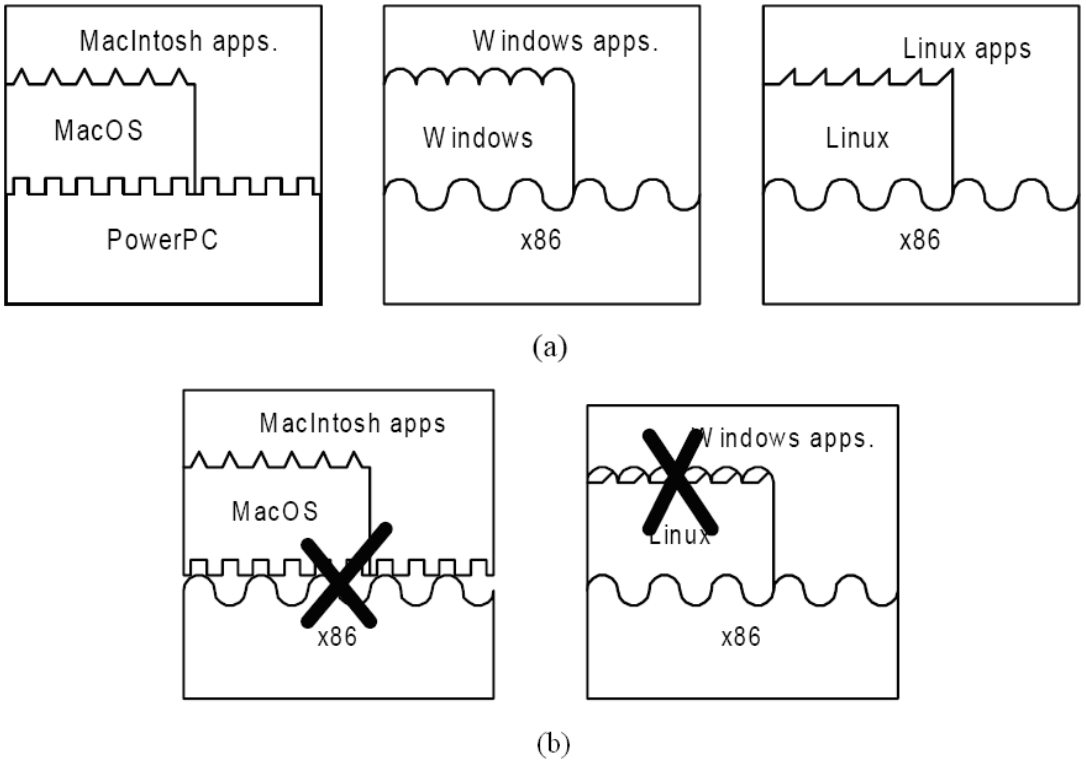

图 2.3 主流计算机系统兼容性示意图

由上所述，根据虚拟的层次我们可以将虚拟机分成为三类：

- ✧ Virtual Machine 实现对硬件结构的虚拟
- ✧ Virtual Operation System 实现对操作系统的虚拟
- ✧ Virtual Machine and Virtual Operation System 两者兼有

Virtual Machine 通常被称为仿真机（Emulation），它是一种为二进制代码呈现原始环境的技术，使得程序就好像在原来的目标环境中运行一样。仿真机主要致力于用软件模拟出计算机的体系结构，包括硬件单元模型，如 CPU、系统芯片（DMA、时钟、中断控制器、总线控制器）。所有这些软件单元集合起来成为虚拟机。或者说仿真机更注重所虚拟对象的硬件部分，仿真各个硬件部分而组成一台虚拟的计算机。仿真机的工作是从二进制代码中读取每一条指令，然后以某种方式执行它，将虚拟机设置为与原始计算机相似的状态。由于仿真机可能需要多个时钟周期才能完成被仿真对象的一个时钟周期（也就是说需要

多条指令才能仿真出对象的一条指令），仿真程序的运行必定比宿主计算机慢得多。

虚拟操作系统（Virtual Operation System）只是对操作系统进行抽象，在新的操作系统环境中虚拟出原来操作系统的环境，使得应用程序可以在不用改动代码的情况下，在新的操作系统上运行。虚拟操作系统的虚拟可以是源代码级别的，也可以是二进制代码级别的。在源代码级别实现的虚拟操作系统上，可以通过连上系统调用转换库重新编译应用程序，使得编译出来的应用程序可以在新的操作系统上运行。Cygwin 和Mingw[9] 都是这种类型虚拟机的代表。在二进制代码级别上实现的虚拟操作系统，必须实现一个加载器，用于加载不同的可执行文件格式。同时截获程序运行时的系统调用，将其转换为当前操作系统上的系统调用。WINE、LINE 和Elanix都是这类虚拟机。

更为常见的一类虚拟机对硬件体系和操作系统都进行了虚拟。这种虚拟机通常用翻译器实现，所以也常被称为翻译机（Translation）。翻译机是将二进制代码转换为另一种语言，使得转换后的语义与源语义严格相同的技术。翻译机主要关注于使用的形式语言。它包括将源二进制可执行文件解析成为相应的语义元素。这样的二进制元素能够被翻译和汇编成为目标机器上的可执行文件。Java 虚拟机就是一种翻译机。它实现了对操作系统的虚拟，在 Java 虚拟机上可以直接运行应用程序。同时 Java 虚拟机又是对一个假象的硬件体系结构的虚拟，它有自己的寄存器组和堆栈。并且 Sun 公司已经推出 Java 芯片，这些芯片可以说是 Java 虚拟机的硬件实现，Java 芯片用来直接运行 Java 程序。

2.1.3 虚拟操作系统Wine

著名的开源项目WINE是一种虚拟操作系统，这一点可以从WINE的名称上就可以看出。WINE可以看成是“Wine Is Not an Emulator”的首字母递归缩写，也可以看成是“Windows Emulator”的缩写[10]。这两种说法都是正确的，前者说明Wine不是一个硬件虚拟机，它没有虚拟一个CPU。用户不可以在Wine上安装Windows操作系统或者Windows设备驱动程序。但是Wine实现了Windows API函数可以用于把Windows应用程序移植Unix上。后者说明了对于Windows应用程序来说Wine看起来行为就像Windows一样。所以说Wine是虚拟操作系统中的典型

代表。

因为Windows应用程序不能运行在Linux上，而Linux应用程序也不能运行在Windows上。所以用户常常在一台安装两种不同的操作系统用于运行两种应用程序。但是运行不同操作系统上的应用程序时，必须重新启动计算机进入另一个操作系统。Wine提供了一种更为方便的方法，可以让Windows应用程序和Linux本地应用程序都能运行在类Unix操作系统上[11]。目前Wine虚拟操作系统的体系构架如图 2.4 所示：

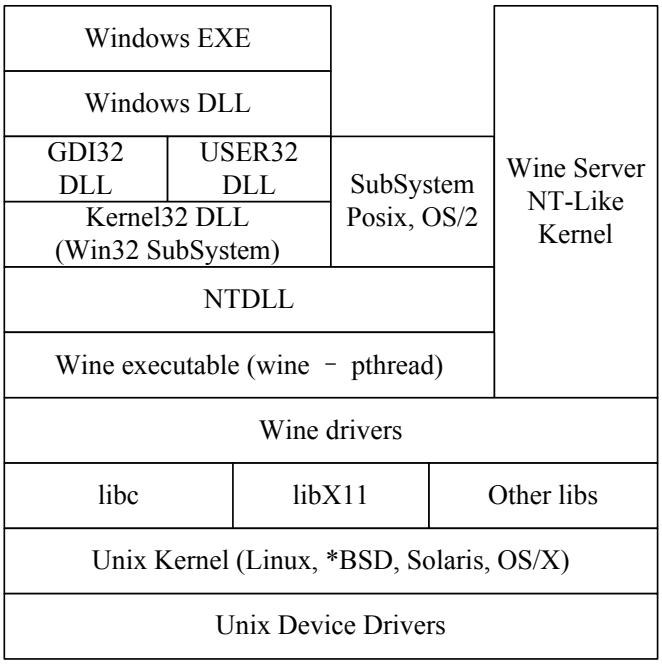

图 2.4 Wine 体系构架图

2.2 构件技术介绍

在信息时代，新的技术革命正在改变我们日常生活的面貌，而这场技术革命的核心是计算机软件系统。在面向对象技术给解决软件危机带来曙光之时，分布式网络计算的巨大压力又给软件开发提出了许多新的难题，使软件开发仍处于高风险状态。新的分布式网络计算要求软件实现跨空间、跨时间、跨设备、跨用户的共享，导致软件在规模、复杂度、功能上的极大增长，迫使软件要向异构协同工作、各层次上集成、可反复重用的工业化道路上前进。为适应软件的这种需求，新的软件开发模式必须支持分布式计算、浏览器/服务器结构、模

块化和构件化集成，使软件类似于硬件一样，可用不同的标准构件拼装而成。具体地说可实现下列几点要求：

- ✧ 提供一种手段，使应用软件可用预先编好的、功能明确的产品部件定制而成, 并可用不同版本的部件实现应用的扩展和更新。
- ✧ 利用模块化方法，将复杂的难以维护的系统分解为互相独立、协同工作的部件，并努力使这些部件可反复重用。
- ✧ 突破时间、空间及不同硬件设备的限制，利用客户和软件之间统一的接口实现跨平台的互操作。

为满足上述要求，软件构件技术出现了。而构件重用的目标是达到需求、分析、设计、编码、测试的重用。从此，一种影响软件产业发展的新的软件开发方法诞生了。

2.2.1 软件复用

从抽象程度来看，面向对象技术已达到了类级重用（代码重用），它以类为封装的单位。这样的重用粒度还太小，不足以解决异构互操作和效率更高的重用。构件将抽象的程度提到一个更高的层次，它是对一组类的组合进行封装，并代表完成一个或多个功能的特定服务，也为用户提供了多个接口。整个构件隐藏了具体的实现，只用接口提供服务。这样，在不同层次上，构件均可以将底层的多个逻辑组合成高层次上的粒度更大的新构件，甚至直接封装到一个系统，使模块的重用从代码级、对象级、架构级到系统级都可能实现，从而使软件像硬件一样，能任人装配定制而成的梦想得以实现。

软件复用[12]（或软件重用）是指充分利用过去软件开发中积累的成果、知识和经验，去开发新的软件系统，使人们在新系统的开发中着重于解决出现的新问题、满足新需求，从而避免或减少软件开发中的重复劳动。 软件复用可分为产品式复用和生成式复用。产品式复用是指对软件开发中中间制品（程序代码，各阶段中的文档或模型，测试用例等）的复用，其实现途径关键是将可复用的程序代码组装（或集成）而生成软件应用系统，因此产品式复用亦称组装式复用；生成式复用主要是将软件的需求进行规约化（或形式化）描述，然后利用可复用的应用程序生成器自动或半自动

地生成所需的软件系统。目前组装式复用是软件复用的主流方式。 软件复用使人们在软件开发中不必“重新发明轮子”或“一切从零开始”，提高了软件生产率和质量，缩短开发周期，降低开发成本。软件的重用中没有材料的消耗，而且软件通过多次重用后其质量和可靠性越来越高。据统计，软件系 统的开发中若复用程度达到 50%，则其生产率提高 40%，开发成本降低约 40%， 软件出错率降低近 50%。 软件工程专家 Bohem 认为，近十年来软件复用已成为解决软件危机、提高软件生产率和质量的最有效、最具潜力的手段。

软件构件（也称软件组件）是软件系统内可标识的、符合某种标准要求的构成成分，类似于传统工业中的零部件。广义上讲，构件可以是需求分析、设计、代码、测试用例、文档或软件开发过程中的其它产品。狭义来说，一般指对外提供一组规约化接口的、符合一定标准的、可替换的软件系统的程序模块。本文中的构件定义指后者。软件构件有两个特征：

(1) 有用性，指构件完成的功能是有用的，也就是其功能可出现在很多应用软件中。

(2) 易用性，指构件要有很好的包装，能很方便地使用它。一般来讲，构件的包装要符合一定的标准。

总结来说，软件构件是软件复用的基本单元。软件构件技术使得软件人员在应用开发时可以使用其他人的劳动成果，为软件产业进行大规模专业化分工与合作形成了前提。一般来说，要实现构件技术必须具备下列几个条件：

- ✧ 有标准软件体系结构，保证构件间通信协议统一，实现同步和异步操作控制，突破本地空间限制，充分利用网络环境；
- ✧ 构件有标准接口，保证系统可分解成多个功能独立的单元，用构件组装而成；构件独立于编程语言；
- ✧ 构件提供版本兼容，来实现应用系统的扩展和更新。

2.2.2 CAR构件技术

CAR（Component Assembly Runtime）构件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[13]。CAR兼容微软的COM标准，但和COM相比，它删除了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（即元数据），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[14]。

CAR技术是在总结面向对象编程、面向构件编程技术的发展历史和经验，为更好地支持面向以Web服务为代表的下一代网络应用软件开发而发明的。CAR很大程度地借鉴了COM技术[15]，保持了和COM的兼容性，同时对COM进行了重要的扩展。

为了在资源有限的嵌入式系统中实现面向中间件编程技术，同时又能得到C/C++的运行效率，CAR 没有使用 JAVA 和.NET 的基于中间代码-虚拟机的机制，而是采用了用 C++编程，用和欣 SDK 提供的工具直接生成运行于和欣构件运行平台的二进制代码的机制。用 C++编程实现构件技术，使得更多的程序员能够充分运用自己熟悉的编程语言知识和开发经验，很容易掌握面向构件、中间件编程技术。在不同操作系统上实现的和欣构件运行平台，可以使 CAR 构件二进制代码可以实现跨操作系统平台兼容。

CAR 的重要特点时构件的相互操作性；软件升级的独立性；编程语言的独立性；进程运行透明度。在实际的编程应用中，CAR 技术可以使程序员得到以下几个方面的受益：

- ✧ 易学易用：基于 COM 的构件化编程技术是大型软件工程化开发的重要手段。微软 Windows 2000 的软件全部是用 COM 实现的。但是微软 COM 的繁琐的构件描述体系令人望而生畏。CAR 的开发环境和欣 SDK 提供了结构简洁的构件描述语言和自动生成辅助工具等，使得 C++程序员可以很快地掌握 CAR 编程技术。
- ✧ 可以动态加载构件：在网络时代，软件构件就相当于零件，零件可以随时装配。CAR 技术实现了构件动态加载，使用户可以随时从网络得到最新功能的构件。
- ✧ 采用第三方软件丰富系统功能：CAR 技术的软件互操作性，保证了系统开发人员可以利用第三方开发的，符合 CAR 规范的构件，共享软件资源，缩短产品开发周期。同时用户也可以通过动态加载第三方软件扩展系统的功

能。

- ✧ 软件复用：软件复用是软件工程长期追求的目标，CAR 技术提供了构件的标准，二进制构件可以被不同的应用程序使用，使软件构件真正能够成为"工业零件"。充分利用"久经考验"的软件零件，避免重复性开发，是提高软件生产效率和软件产品质量的关键。
- ✧ 系统升级：传统软件的系统升级是一个令软件系统管理员头痛的工程问题，一个大型软件系统常常是"牵一发而动全身"，单个功能的升级可能会导致整个系统需要重新调试。CAR 技术的软件升级独立性，可以圆满地解决系统升级问题，个别构件的更新不会影响整个系统。
- ✧ 实现软件工厂化生产：上述几个特点，都是软件零件工厂化生产的必要条件。构件化软件设计思想规范了工程化、工厂化的软件设计方法，提供了明晰可靠的软件接口标准，使软件构件可以像工业零件一样生产制造，零件可用于各种不同的设备上。
- ✧ 提高系统的可靠性、容错性：由于构件运行环境可控制，可以避免因个别构件的崩溃而波及到整个系统，提高系统的可靠性。同时，系统可以自动重新启动运行中意外停止的构件，实现系统的容错。
- ✧ 有效地实现系统安全性：系统可根据构件的自描述信息自动生成代理构件，通过代理构件进行安全控制，可以有效地实现对不同来源的构件实行访问权限控制、监听、备份容错、通信加密、自动更换通信协议等等安全保护措施。

第3章 Elanix虚拟操作系统研究

3.1 本章引论

Elanix 是一种虚拟操作系统。正如第二章所述，虚拟操作系统只是对操作系统进行抽象，在新的操作系统环境中虚拟出原来操作系统的环境，使得应用程序可以在不用改动代码的情况下，在新的操作系统上运行。如图 3.1 所示，Elanix 是二进制级别的虚拟操作系统，可以让编译生成后的和欣操作系统（Elastos）的应用程序直接运行的 Linux 操作系统上。

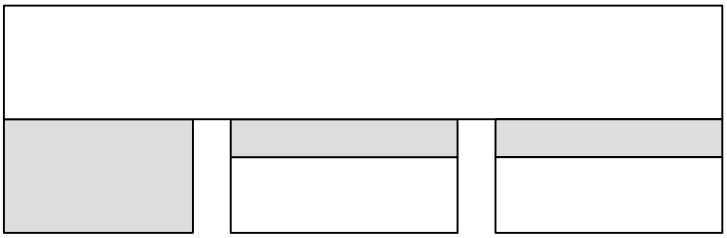

图 3.1 Elastos 应用程序跨平台示意图

和欣操作系统是 32 位嵌入式操作系统，它基于微内核，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。和欣操作系统体积小、速度快，适合网络时代的嵌入式信息设备[13]。和欣操作系统式完全面向构件技术的操作系统，它提供的功能模块全部基于CAR构件技术，因此是可拆卸的构件，应用系统可以按照需要裁减组装而成，或在运行时动态加载必要的构件。

本章主要说明了Elanix虚拟操作系统的整体结构，并依次说明了PE加载器、API 函数转换层和 Server 模块。其中主要对 Elanix Server 模块中使用的各个技术进行了分析讨论。

3.2 Elanix整体结构

Elanix 虚拟操作系统的目标是在 Linux 上虚拟出 Elastos 操作系统环境，让基于 CAR 构件技术的 Elastos 应用程序可以在 Linux 上运行。如果没有 Elanix，

Elastos 应用程序不能直接在 Linux 上运行。这是因为：

首先 Elastos 应用程序和 CAR 构件是 PE 文件格式，而 Linux 操作系统并不支持 PE 文件格式，Linux 操作系统上的应用程序和动态链接库是 ELF 格式；

其次 Linux 操作系统没有提供 Elastos 应用程序在运行过程中调用的 Elastos API 函数；

最后 Linux 操作系统也没有提供 Elastos 应用程序在运行过程中使用的内核对象服务；

内核对象是 Elastos 操作系统的特点之一。与传统操作系统不同，Elastos 内核已经完全构件化，内核的各个功能模块是按照 CAR 构件规范建立，都是 CAR 构件对象。Elastos 内核向 Elastos 应用程序提供的大部分系统服务是以 CAR 构件规范接口的形式提供的。只有少量系统服务是以 API 函数形式提供。当 Elastos 应用程序运行到调用 Elastos API 函数和内核对象服务的时候，如果运行环境没有提供这些函数和对象或者提供的函数和对象的语法语义与真实的函数和对象不符，那么程序就会运行出错。

由此可知 Elanix 想要实现目标，必须解决三个基本问题。

1) 如何将 PE 文件格式的 Elastos 应用程序和 CAR 构件加载到 Linux 进程空间并运行；

2) 如何截获并实现 Elastos 应用程序中调用的 Elastos API 函数；

3) 如何截获并实现 Elastos 应用程序中调用的 Elastos 内核对象服务；

如图 3.1 所示，Elanix 主体上由 PE 加载器、API 函数转换层、内核对象服务三个模块组成。

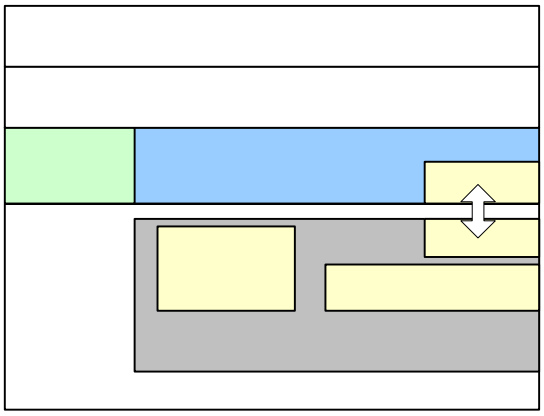

图 3.2 Elanix 结构图

Elanix 的 PE 可执行程序加载器在每个 Elanix 进程启动的时候，会把由参数指定的 Elastos 应用程序和 Elastos.so 加载到 Elanix 进程用户空间中。这样就解决了第一个基本问题。

另外 Elanix 提供了一种 Built-In DLL 技术，能够把以 Linux 上.so 文件形式存在的动态链接库，加载到用户进程空间中，提供 Elastos DLL 相同的功能——为 Elastos 应用程序提供 API。Elastos.so 就是一个 Built-In DLL，它包含了与 Elastos API 同名函数，通过调用 Linux 系统调用和 Elanix 内核对象服务实现了 Elastos API 函数的转换。Elanix 内核对象服务（Elanix Server）模块，通过 Elastos.so，向 Elastos 应用程序提供内核对象服务。Elastos.so 和 Elanix Server 两者联合解决了后两个基本问题。

图 3.3 是 Elastos 结构图，与上图相比，可以看出 Elanix 结构大体与 Elastos 相同，但各模块皆有不同。Elanix 加载器模块是在用户空间中，而 Elastos 加载器模块是在内核空间中。Elanix 用 Elastos.so 取代了 Elastos 中的 Elastos.dll 实现 API 函数转换。Elanix 的内核对象服务是通过 Linux 内核中的 Elanix Server 模块提供的，而 Elastos 的内核对象服务是直接通过 Elastos 内核提供的。

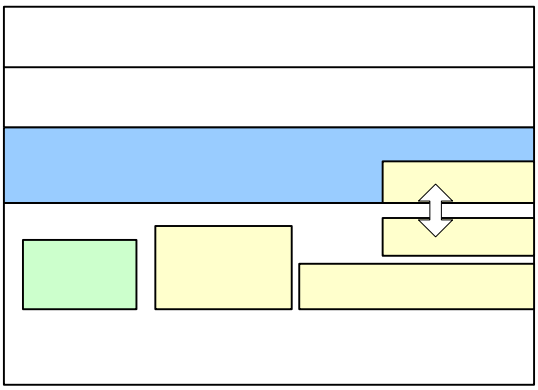

图 3.3 Elastos 结构图

总之，Elanix 通过 PE 加载器、API 函数转换层、内核对象服务三个模块在 Linux 操作系统上虚拟出 Elastos 操作系统相同的环境，使得 Elastos 应用程序和 CAR 构件可以在 Linux 操作系统上直接运行。

3.3 PE程序加载器

Elastos应用程序和CAR构件所采用的二进制文件格式为PE[16]（Portable Executable）格式，与Windows上可执行文件格式相同。然而，Linux系统支持最广泛的可执行文件格式为ELF(Executable and Link Format)[17]，它不支持PE文件格式。因此为了使Elastos应用程序能在Elanix中运行，必须在Elanix中设计实现支持PE格式的文件加载器。Elanix加载器的功能是加载运行Elastos SDK上开发的应用程序和自描述CAR构件。加载过程不仅包括将可执行文件映射到Linux地址空间中，还需要考虑地址空间的分配策略，动态库和CAR构件加载，系统调用实现等方面的问题。PE格式和ELF格式加载的区别在于：

(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) 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，并将这些信息保存，以便于其他进程对该构件的访问。

(3) 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。

(4) Linux 中共享库的加载地址为 0x40000000（1G）开始的地址，Elastos 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。

由于可执行文件格式不同，Elastos应用程序无法在Linux下直接加载运行，但是Elastos应用程序和Linux下的ELF程序之间仍存在一些共同点：两者都是运行在PC机上；使用的机器指令集都是基于IA-32[18]（Intel Architecture 32-bit）；过程调用都是遵循以栈帧[18]（Stack Frame）结构来进行的；最为重要的是，ELF格式和PE格式文件的代码段并没有显著的不同，通过反汇编命令可以看到，两者的汇编指令完全兼容。指令级的兼容性为执行Elastos应用程序提供了运行的基础。基于此，Elanix加载器的设计原理为：

(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 Elastos 应用程序、动态连接库以及 CAR 构件；

(2) 当加载完成后，进行必要的外部函数地址的设置，然后加载器将控制权交给 Elastos 可执行程序的入口函数，至此加载器的使命结束；

(3) Elastos 应用程序开始运行，直到程序结束。

将 Elastos 应用程序映射到地址空间，需要通过 Linux 系统调用分配所需的地址空间。在 Linux 分配地址空间主要有两种方式：一种是利用 new，malloc 等比较常见的系统函数来从堆中分配地址空间；另一种是通过 mmap 来分配或预留地址空间。Elanix 加载器采用第二种方式分配地址空间，其好处是可以指定分配空间的起始地址，从而可以在指定地址加载 Elastos 应用程序。一般将 exe 文件加载到从 0x400000 开始的地址空间（以防缺少重定位信息），而 DLL 则可加载到任意可用的地址空间。

3.4 API函数转换层

Elanix 中 API 函数转换层主要有三个功能：一个是截获 Elastos 应用程序运行中对 Elastos 系统 API 函数的调用；一个是实现这些 API 函数的功能保证与原有的 Elastos 系统 API 函数语义相同；最后是要调用 Elanix Server 向 Elastos 应用程序提供内核对象服务

3.4.1 实现API函数

截获 API 函数调用的工作实际上是在加载 Elastos 应用程序时完成的。Elastos 应用程序对 API 函数的调用实际上就是调用动态链接库 Elastos.dll 中的函数。Elastos.dll 在编译生成时，还有一个配合的 Elastos.lib 静态库文件。该静态库是用于链接生成 Elastos 应用程序时使用的。Elastos 应用程序中必然有调用 API 函数的指令，而 Elastos.lib 静态库中提供了这些 API 函数，这样 Elastos 应用程序才可以正确编译链接生成。但 Elastos.lib 中的 API 函数体内都只有一条跳转语句，API 函数的真正实现还是在 Elastos.dll 动态链接库中。如 Elastos.lib 的

API 函数 EzCreateProcess 的汇编代码如下所示：

_EzCreateProcess:

004010C0: FF 25 8C 52 42 00 jmp dword ptr [_imp_EzCreateProcess]

这些 API 函数的跳转地址都存储在 PE 文件的 IAT（Import Address Table）表中。所以 PE 可执行文件加载器在加载 Elastos 应用程序的过程中，只需要把 IAT 表中的值修改为 Elastos.so 中同名 API 函数的地址值（Elastos.so 中函数地址值可以通过 dlsym 函数得到），就可以在 Elastos 应用程序运行过程截获 API 函数调用，在 Elastos.so 中处理 API 函数。

Elanix 虚拟操作系统实现的 API 函数可以分类如表 3.1。由于不同类别的 API 函数功能不同，所以采用的实现方法也不同。

表 3.1 Elanix API 函数分类

类别	说明	示例
字符串及内存操作	向应用程序提供基本常用的字符串及内存操作	EzAllocString 复制创建一个新的字符串 EzTaskMemAlloc 分配一块内存 ……
基本系统服务	向应用程序提供最为基本的系统服务	EzDebugPrint 输出打印信息 EzGetSystemTime 得到当前系统时间 ……
CAR 构件相关服务	CAR 构件加载创建等等。	EzGetObjectFactory 得到指定 CAR 构件类厂 EzCreateInstance 加载并创建 CAR 构件对象 ……
内核对象服务	使用 Elanix Server 提供的内核对象向应用程序提供常用服务	EzFindService 获取指定名字的服务接口 EzCreateProcess 根据文件名创建新进程 EzCreateSharedMemory 创建共享内存 ……

“字符串及内存操作”类和“基本系统服务”类中 API 函数的功能较为单一而且不需要访问所有 Elanix 进程共享的数据，所以可以通过调用 Linux 上用

户态系统函数以实现其功能。例如 EzAllocString 函数代码如图 3.4 所示，主要是通过调用 Linux 上 glibc 库中的的 malloc，wcslen，wcscpy 三个函数实现的。

```
STDAPI_(BSTR) EzAllocString(const OLECHAR * pStr)
{
    if (pStr == NULL)    return NULL;
     UINT byteLen = sizeof(OLECHAR) * wcslen((const Wchar_t *) pStr);
    BSTR bstr = (BSTR) malloc(byteLen + 6);
    if (bstr != NULL) {
        *((UINT*)bstr) = byteLen;
        bstr = (BSTR) ((UINT*)bstr + 1);
        wcscpy(bstr, (Wchar_t *)pStr);
    }
    return bstr;
}
```

图 3.4 API 函数 EzAllocString 的实现代码

“与 CAR 构件相关服务”类的 Elanix API 函数会使用 PE 可执行文件加载器以便加载 CAR 构件。“内核对象服务”类的 Elanix API 函数主要是以 C 函数的形式封装了 Elanix Server 提供的内核对象。比如 EzCreateProcess 函数的代码如图 3.5，主要是通过调用 Kernel 和 Process 内核对象接口实现了功能。

```
STDAPI EzCreateProcess(EzStr esName,EzStr esArg, IProcess **ppIProcess)
{
    HRESULT hr;
    IProcess *pIProcess;
    hr = GetIKernel()->CreateProcess(&pIProcess);
    if (FAILED(hr)) return hr;
    hr = pIProcess->Start(esName, esArg);
    if (SUCCEEDED(hr) && ppIProcess) {
        *ppIProcess = pIProcess; }
    else {
         pIProcess->Release();}
    return hr;
}
```

图 3.5 API 函数 EzCreateProcess 实现代码

3.4.2 转接内核对象服务

Elastos 应用程序在运行过程中往往会调用各种各样的内核对象（包括驱动对象），Elanix API 函数仅仅封装了常用的一小部分内核对象方法。Elastos 应用程序运行过程中也可能会调用其它进程提供的 CAR 对象服务。无论是内核对象还是远程 CAR 对象，对于调用者 Elastos 应用程序来说调用方式都是一致的。调用者可以通过两种方式得到服务：

(1) 通过 Elanix Server 提供的命名服务，使用 EzFindService 函数查找已注册过的名字，得到 CAR 对象接口指针。

(2) 通过先前调用过的函数或者对象方法的输出参数，得到新的 CAR 对象接口指针。

无论何种方式取得接口指针后，调用接口的方法如同调用进程内的 CAR 对象方法一样。之所以能达到如此功能主要是因为 API 转换层会为被调用的内核对象和进程外 CAR 构件在本进程用户空间内自动生成代理对象。如果本进程空间内的 CAR 对象需要被其它进程或者内核使用，那么 API 转换层将会在注册该 CAR 对象为命名服务的时候为该 CAR 对象在本进程用户空间内创建存根对象。除此以外，API 转换层还完成了标准列集散集的全过程，确保了远程 CAR 对象方法调用和内核对象方法调用的顺利进行。而这些对于 Elastos 应用程序来说都是透明的。

有关具体代理与存根，列集散集等远程 CAR 构件通信机制将在第 4 章有详细的说明。

3.5 内核对象服务模块

Elanix 内核对象服务模块（Server）是 Linux 上 Elanix 虚拟操作系统的核心部件之一。Elanix Server 处于 Linux 内核中，向用户态的 elastos.so 提供 Elastos 系统调用（SysCall）。elastos.so 将把这些 Syscall 函数封装为 Elastos API，向用户程序提供服务。从这个意义上来说，Syscall 是比 API 更底层的系统接口。

表 3.2 SysCall 函数分类

分类	SysCall	说明
命名服务及列集散集相关	SysRegisterService	在命名服务上注册服务，此 oid 对应的 object 必须已经在内核中了
	SysRegister	注册 Object
	SysUnregister	删除 Object
	SysRegisterCommon	命名服务上的通用服务注册函数
	SysResolveCommon	命名服务上的通用服务查找函数
	SysUnregisterCommon	命名服务上的通用服务注销函数
	SysInvoke	列集散集中数据传递的函数
	SysRegisterClsInfo	注册元数据到共享链表中
	SysAttachServer	注册 Object 到 Import 链表上
	SysDetachServer	从 Import 链表上注销 Object
事件相关	SysCreateEvent	创建事件
	SysDestroyEvent	删除事件
	SysNotifyEvent	通知事件
	SysWaitEvent	等待事件
系统功能相关	SysReboot	重启系统
	SysExitThread	退出线程（如果是主线程则退出进程）
	SysPrint	打印输出

因为 Elastos 与传统操作系统不同，它不仅仅提供普通的 C 语言函数形式的系统调用，而且提供符合 CAR 构件标准的内核对象服务。用户得到内核对象（代理对象）的接口指针后，就可以调用内核对象提供的不同服务。可以说 Elanix Server 提供的系统调用比传统操作系统内核更复杂。Elastos 内核向 Elastos 应用程序提供的大部分系统服务是内核对象的接口，只有少量系统服务是直接以 API 函数形式提供。Elanix Server 作为 Elastos 内核在 Linux 上的模拟实现，同样也向 Elastos 应用程序提供内核对象服务。

表 3.3 Elanix Server 中主要内核对象及其方法

对 象	说 明	接口中的方法
进程对象	代表 Elanix 进程，包含多个线程对象	Start, Kill, GetThreads, ...
线程对象	代表 Elanix 线程，执行单位	Start, Suspend, Resume, Abort, ...
共享内存对象	代表可以被多个进程共享的一段内存	Attach, Detach
互斥对象	代表互斥信号量	Lock, TryLock, Unlock
模块对象	代表内存中载入的一段代码或数据	GetEntryPoint, GetName, …

与传统通过 API 函数提供操作系统内核功能相比，内核对象服务有以下优点：

(1) 符合面向对象和面向构件的程序设计思想。应用程序编写更方便、结构更清晰。

(2) 都有完备的元数据，实现自描述，可以动态加载/卸载，这包括硬件驱动即插即用。

(3) 具有动态可配置性，当环境发生变化时，可以对内核对象服务做相关调整

3.5.1 Linux模块与字符设备文件

Elanix Server 中的内核对象主要负责进程、线程管理、共享内存、进程间通信等工作。这些内核对象中的数据和方法需要被多个进程共享，所以 Elanix Server 实现及其与用户程序之间的通信方式可以考虑以下四种设计方案：

(1) 申请一块所有进程共享的内存，将内核对象放入其中。各个进程可以直接调用共享内存中的内核对象。

(2) 在一个单独的 Linux 进程中实现内核对象，其它进程通过 IPC 或 Socket 与其通信。

(3) 修改Linux内核源码[19]，在Linux内核中实现内核对象，重新编译Linux内核。用户程序可以通过系统调用得到内核对象服务。

(4) Elanix Server使用Linux模块作为载体[20]，实现内核对象。用户程序可

以通过设备文件[21]得到内核对象服务。

经过比较，作者选择用第四种方案设计并实现了Elanix Server通信机制。Linux中模块是一种目标对象文件，是一组已经编译好而且已经链接成可执行文件的程序，它们是核心的一部分，但是并没有编译到核心里面去。模块可以在系统启动时加载到系统中，也可以在系统运行的任何时刻加载；在不需要时，可以将模块动态卸载[22]。内核模块的动态装载特性可以把内核映像文件保持在最小从而节省内存，使用新的模块时不必重新编译内核，只要把新的模块编译后装载进系统，使系统具有更高的灵活性，同时对于模块技术的一个重要应用——设备驱动程序来说，虽然通常依赖于某些特殊的硬件，但它不依赖于某个固定的硬件平台，使其具有平台无关性[23]。与其它三种方案相比，使用Linux模块作为Elanix Server的载体和设备文件的通信方式有以下优点：

(1) Linux 模块运行在 Linux 内核空间，由于用户空间的程序不可以直接访问内核空间的程序和数据，这样提高了 Elanix Server 的安全性和可靠性。共享内存虽然使用起来方便，但这种方式并不可靠，一个应用程序的误操作就有可能导致整个 Elanix 系统崩溃。

(2) 基于设备文件的通信方式，其通信效率要高于 IPC 和 Socket 等进程间通信方式。

(3) 与修改 Linux 内核源码方法相比，使用 Linux 模块作为载体，不需要重新编译 Linux 内核，便于 Elanix Server 的发布和升级，提高了 Elanix Server 灵活性和在不同版本 Linux 内核上的可移植性。

内核对象服务模块在底层是通过 SysCall 函数向 API 转换层和用户程序提供内核对象服务的。因为 SysCall 函数实现在内核空间中，而 SysCall 函数的调用者处于用户空间。所以必须提供一种途径使得 SysCall 的调用、参数和返回值可以在用户空间与内核空间之间穿行。

Elastos 操作系统与其它操作系统一样通过软中断由用户程序调用内核的 SysCall 函数。因为 Elanix 是建立在 Linux 之上的虚拟操作系统，且 Elanix Server 是 Linux 模块不能修改 Linux 的系统调用表，所以 Elanix 采用了字符设备文件的方式传送 Elanix 程序。 Elanix 定义了完整的通信协议保证用户请求和服务应答可以通过设备文件传输。其关键部分代码如下所示：

```
//Protocal.h 中定义 SysNo
enum SysNo{
    RegisterService,
    UnRegister,
    Invoke,
    …
    NB_SYSNO,
};
```

```
//Elanix Server 中定义 SysCall 函数表
typedef void (*handler) (union request* preq,
            union reply* prep);
static const handler
    SysCallTable[NB_SYSNO] ={
    (handler) SysRegisterService,
    (handler) SysUnRegister,
    (handler) SysInvoke,
    ….
};
```

图 3.6 SysNo 与 SysCall 函数表的定义

Protocal.h 头文件中定义了通信双方共用的数据结构包括函数请求号 SysNo 枚举类型，函数输入 Request 联合体、输出参数 Reply 联合体等等。在 Elanix Server 中定义了 SysCall 函数指针数组，注意 SysCallTable 中函数指针的排列顺序必须与 SysNo 中定义的一致。

数据结构定义完毕后，对于不同 SysCall 调用，Request 联合体和 Reply 联合体会赋于不同结构类型和值。然后用户程序中就可以通过读写设备文件发送函数请求、得到函数结果。Elanix Server 会接受传入的数据，通过已经定义好的函数指针数组 SysCallTable 解析并分发，调用对应的 SysCall 函数完成请求。最后，当用户程序调用 read 函数的时候，Elanix Server 会将 Reply 中的结果传给用户程序。

```
//用户程序中读写设备文件
if(open("/dev/server",
    O_RDWR) != -1){
    write(fd, pRequest,
        sizeof(*pRequest));
    read(fd, pReply,
        sizeof(*pReply));
    fclose(fd);
}
```

```
//Elanix Server 中分发 SysCall 请求
enum SysNo sysno = request_ptr->sysno;
if(sysno < NB_SYSNO)
    SysCallTable[sysno] (request_ptr,
        reply_ptr);
```

图 3.7 读写设备文件与调用 SysCall 函数的关键代码

3.5.2 Linux模块中使用C++编程

Elanix 中的内核对象都是按照 CAR 构件规范编写的，所以每个内核对象都需要先写接口描述文件.CAR，然后在用 CAR 文件编译器生成 C++代码框架，最后在 C++代码框架中填写实现。CAR 构件接口最后都是用 C++虚基类来实现的。比如所有接口共同的父接口 IUnknown 就是用 C++语言表示如图 3.8 所示：

```
Class IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        /* [in] */ REFIID riid,
        /* [out] */ void **ppv) = 0;
    virtual ULONG STDMETHODCALLTYPE AddRef( void) = 0;
    virtual ULONG STDMETHODCALLTYPE Release( void) = 0;
};
```

图 3.8 使用 C++语言来表达 IUnknown 接口

由上可知内核对象用 C++编程实现更为方便。但是 Elanix Server 是一个 Linux 驱动模块。Linux 驱动模块处于 Linux 内核空间，Linux 内核都是采用 C 语言编写，一般 Linux 驱动模块也是用 C 语言编写。

Linux模块是经过编译但是尚未链接的目标代码（.o）文件，可以在系统运行时动态地“安装”到内核中。这种动态安装既可以由特权用户进行，也可以由内核在有需要时自动地启动[24]。在内核源码中可以规定允许导出内核的一些符号，如函数入口、全局变量等等，使得Linux模块程序将调用这些内核函数或者访问这些全局变量。对这些符号的引用是在模块加载的时候链接解决的，所以加载的过程也就包含了链接的过程。

如上所述，如果使用 C++编写 Linux 模块，存在着两个困难：

(1) Linux 模块必须包含 Linux 内核源码的头文件。而这些头文件中的函数声明是 C 语言风格的，C++程序中包含了这些头文件就会编译出错。

(2) 既便编译顺利通过，也无法加载到 Linux 内核。C++程序中用到的许多 C++特性都需要底层 C++函数库的支持。上文提到 Linux 模块加载的过程就是链接的过程，因为没有所需的函数，所以使用 insmod 命令加载 Linux 模块时会出现符号无法解析的错误。

一种解决的方法是根据 C++对象模型使用 C 语言实现等价的功能。这种方法全部用 C 语言实现了 Elanix 内核对象服务。虽然实现简便，但是代码复杂，不易理解和维护。如图 3.9 所示，Elanix Server 中通过分开编译 C 与 C++代码、链接部分 C++支持库和使用 extern ”C”连接 C 语言和 C++语言的方法仍然使用 C++语言编写内核对象。

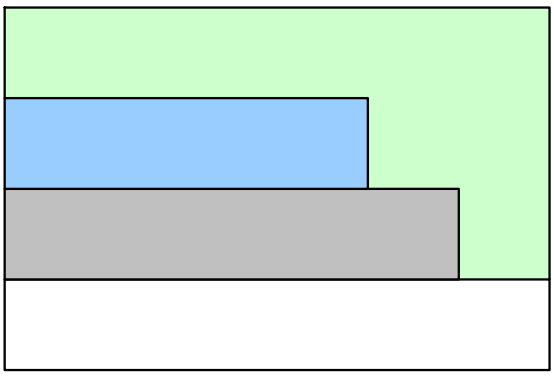

图 3.9 Elanix Server 支持 C++语言层次图

首先 Elanix Server 将 Linux 模块实现代码和内核对象服务实现模块代码分开实现。前者用 C 语言包含 Linux 内核源码头文件实现，生成 module.o 目标文件；后者用 C++语言单独实现，生成 major.a 静态库。这就解决的第一个问题。

然后Elanix Server生成时会链接C++语言支持库libsupc++.a静态库。该静态库提供了C++特性所需要了这些C++ ABI(Application Binary Interface)，比如C++中使用的new、delete操作符，就由libsupc++.a提供。但是libsupc++.a还要调用C++ Base ABI：unwinding库和malloc，free函数。可以通过Linux内核中的kmalloc和kfree函数实现malloc和free提供给libsupc++.a。而Uwinding库提供的是C++ Exception处理机制的底层支持[25]。在Linux内核实现C++ Exception处理机制较为复杂，需要修改Linux内核源码[26]。因为Elanix Server中的内核对象只需要用到C++类和虚拟函数，不需要C++ Exception处理机制，而且不能修改Linux内核源

内核对象、
（C+
C++支持库
libsupc++.a unw
Elanix Server内
（C语言

码，所以可以简单的编写一个文件unwind.c将Uwinding库中的函数都实现为空函数，生成目标文件unwind.o。

最后我们把 major.o、libsupc++.a、module.o 和 unwind.o 四者用 ld 链接器生成一个 elaserver.o 目标文件，此时就可以用 insmod elaserver.o 命令将 C++编写的内核对象加入 linux 内核。

3.5.3 内核对象封装

Elanix 建立在 Linux 操作系统之上，Elanix Server 不能象 Elastos 那样自己实现内核对象的方法，而是用与 Elastos 类似的逻辑结构封装 Linux 内核功能提供与 Elastos 相同的内核对象服务。

在 Elanix Sever 初始化过程中，一个总的 Kernel 对象将在命名服务（Naming Service）上注册为“ElastosKernel”服务。当用户程序调用内核对象服务时候，先用 EzFindService 函数查找“ElastosKernel”服务得到 IKernel 接口指针，再得到各个内核对象的接口指针，然后就可以调用各个内核对象的方法。例如，启动新进程执行 Example 程序的代码如图 3.10：

```
EzFindService(EZCSTR("ElastosKernel"), (IUnknown **)&pIKernel);
pIKernel->CreateProcess(&pIProcess);
pIProcess->Start(EZCSTR(“Example”), NULL);
```

图 3.10 调用进程内核对象示例代码

如图 3.11 所示，Elanix Server 调用 Linux 内核函数实现了与 Elastos 相同功能的内核对象，最后这些对象都被汇总到 Kernel 对象中。

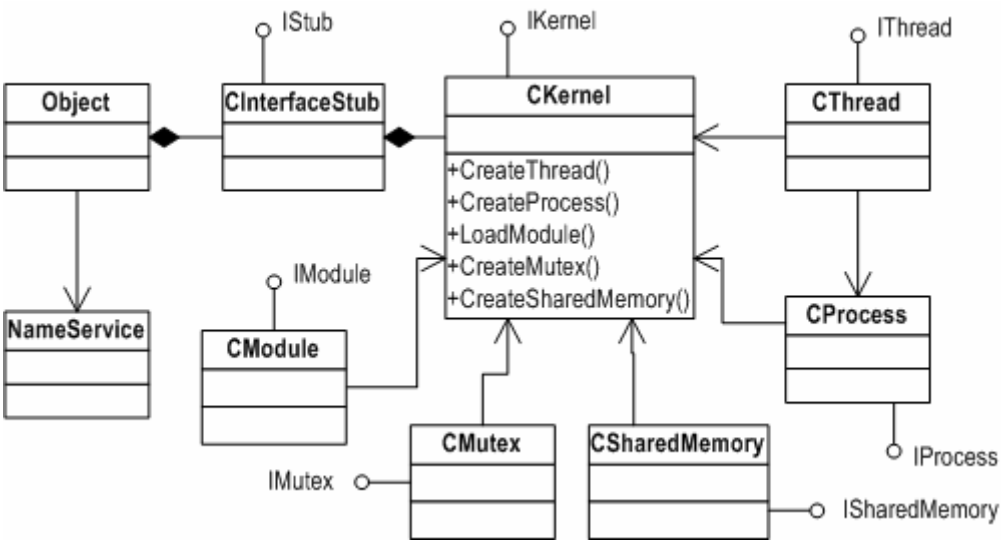

图 3.11 主要内核对象及命名服务结构图

Elanix Server 将 Linux 内核中各个功能模块封装为与 Elastos 内核中相同的内核对象。因为 Elastos 内核与 Linux 内核内部数据结构差别很大，所以不可能一对一的把 Linux 内核中的数据结构和函数封装为 Elanix Server 中的内核对象。下面以进程对象和线程对象部分属性和方法为例，说明封装原理。

```
class CProcess : public IProcess{
public:
    STDMETHODIMP QueryInterface(REFIID riid, void ** ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetId( /* [out] */ INT * pPid);
    STDMETHODIMP GetThreads(
        /* [out] */ IObjectEnumerator **ppThreads);
    ......
public:
    AS                  *m_pAS;
    DLinkNode           m_threadList;
    .....
}
```

图 3.12 进程对象部分源码

Elasot 内核与 Linux 内核线程进程管理的既有联系也有区别。CProcess 类开

头的三个成员函数 QueryInterface、AddRef、Release 是由 CAR 构件标准规定。Elanix 进程对象和线程对象是一对多的关系。每个进程对象中有一个地址空间（AS）对象的指针，该地址空间（AS）对象负责对该进程的地址空间进行管理。每个进程对象中有一个线程对象的链表 m_threadList，该链表存储了该进程中的线程对象。

```
class Thread : public DLinkNode, public IThread {
public:
    STDMETHODIMP QueryInterface( REFIID riid, void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetId( /* [out] */ int * pTid);
    ......
public:
    uint_t    m_uState;
    ......
}
```

图 3.13 Elanix 线程对象部分源码

Thread 类也是按照 CAR 构件标准实现了 IThread 接口。每个线程对象有一个成员变量 m_uState 标识了该线程的状态。与 Elastos 不同，Linux 内核用进程控制块 task_struct 结构存放了 Linux 进程的信息。在 Linux 内核中只有进程和轻量级进程，没有线程的概念。Linux 的轻量级线程可以共享地址空间和资源，提供对用户程序线程的支持。所以 Elanix Server 将 Linux 进程和轻量级进程中有关执行的信息转换到一一对应的线程对象中。比如线程对象的成员变量 m_uState 将由 Linux 进程的 task_struct 结构中描述该进程状态的 state 字段值决定，线程对象的 TID 将与 Linux 进程的 PID 一一对应。

同时 Elanix Server 为单独的 Linux 进程或者一组共享地址空间和资源的 Linux 轻量级进程，创建一个进程对象，该进程对象的 PID 将由 Elanix Server 独立实现并维护与 Linux 内核无关。该进程对象的线程列表将保存单独 Linux 进程或轻量级进程组对应的线程对象。同时将 Linux 单独进程或轻量级进程组的空间和资源信息转换到对应的进程对象中去。比如进程对象的地址空间对象指针

m_pAS 将指向一个根据 Linux 进程控制块 task_struct 中描述内存布局 mm_struct 结构新建立的地址空间对象。

总之，Elanix Server 通过自己独立实现和引用 Linux 内核相关数据及函数的方法，将 Linux 内核功能模块封装为与 Elastos 相同的内核对象。

第4章 Elanix中的CAR构件技术

4.1 本章引论

CAR 构件技术是 Elastos 操作系统最大的特色，CAR 构件与 Elastos 操作系统融为一体。Elanix 虚拟操作系统必然要实现对 CAR 构件全面支持。Elanix 虚拟操作系统实现了以下 CAR 构件支撑技术：

- 进程内加载 CAR 构件
- 命名服务机制
- 基于元数据的自动列集散集机制
- 构件生命周期管理

CAR 构件中的远程对象服务和内核对象服务因为客户端程序不能直接访问服务端的对象，所以采用了相似的技术，比如命名服务，列集散集，生命周期管理等等。所以在说明这些技术的时候，不会特意指出是针对内核对象服务还是远程对象服务，在一些实现细节上的不同会特别说明。

4.2 CAR构件分类与通信

在Elanix虚拟操作系统上根据客户程序和CAR构件服务程序所在的进程地址空间，可分为进程内 CAR 构件，内核空间的 CAR 构件，进程外 CAR 构件，这三种构件导致了客户和构件服务之间的通信方式不同。

4.2.1 进程内CAR构件及其加载

进程内 CAR 构件与客户程序在同一个进程用户地址空间里，客户程序可以直接调用接口的方法，因此进程内 CAR 构件和客户程序的通信效率非常高，与此同时，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非常重要。

进程内的 CAR 构件也称为本地服务，一个接口要想只作为本地接口，那么

在定义此接口时，在 CAR 文件里为本接口加上 local 属性即可。

构件的装载过程也就是构件对象的创建过程，客户端通过调用 EzCreateInstance 得到对象接口指针，这个函数声明如下：

```
STDAPI EzCreateInstance(
  /* [in] */ REFEZCLSID rclsid,
  /* [in] */ DWORD dwContext,
  /* [in] */ REFIID riid,
  /* [out] */ POBJECT *ppObj)
```

rclsid 指定构件类的 EZCLSID, dwContext 指定对象环境参数，如果使用 CTX_SAME_DOMAIN，则创建进程内 CAR 构件对象。riid 指定要创建的对象接口 IID, ppObj 返回对象接口指针。

EzCreateInstance 创建进程内构件对象的过程如下：

(1) Elanix 根据 EZCLSID 的 uunm 将指定 CAR 构件动态链接库装载到客户进程

(2) CAR 构件库得到 DllGetClassObject 函数的内存地址

(3) EzGetObjectFactory 调用 DllGetClassObject 函数得到对象工厂

(4) 对象工厂调用 CreateInstance 创建指定 CAR 构件对象，并用 QueryInterface 方法查询出返回的接口指针

Elanix 虚拟操作系统不支持在客户进程中直接创建进程外的 CAR 构件对象。必须先由服务进程创建该 CAR 构件对象后，并注册为命名服务后。客户进程才可以通过命名服务得到进程外 CAR 构件接口。而处于内核空间的 CAR 构件对象，即内核对象，其代码在编译时就已经存在于 Elanix Server 中。所以当 insmod 命令加载 elaserver.mo 模块时，内核对象代码也就被载入 Linux 内核空间。命名服务以及 SysCall 函数会根据用户程序的请求，在内核空间创建相应的内核对象已提供服务。

4.2.2 远程CAR构件及其通信方式

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。客户程序不可以直接访问另一地址空间内的 CAR 构件接

口，所以客户程序和远程构件服务在通信时，必须要跨越进程边界。同样用户程序访问内核对象服务时，也不可以直接访问收到保护的内核空间内的内核对象接口。

Elanix 规定了四层通信机制，主要通过代理和存根实现了客户程序与远程构件服务和内核对象服务之间的交互。客户程序在使用远程构件服务和内核对象服务的时候，实际上是在使用本进程用户地址空间中的代理，所以使用方法与调用进程内 CAR 构件对象完全相同。

表 4.1 Elanix 远程构件服务与内核对象服务通信各层含义

空间 / 层次	客户进程用户空间	服务进程空间或内核空间
第四层 主体层	客户程序主体，请求调用远程对象服务或内核对象服务。	远程 CAR 对象主体实现远程对象服务或内核对象主体实现内核对象服务
第三层 接口层	以 CAR 构件接口的形式向用户程序提供远程对象服务或内核对象服务。	远程 CAR 对象接口或是内核对象接口
第二层 列集层	根据元数据生成标准代理对象，将对象函数调用信息打包传递。 接收存根对象返回结果。	根据元数据，存根解包代理传来的信息，调用对应的远程对象服务或内核对象服务。发送结果给代理对象。
第一层 传输层	将请求从用户空间传递到内核空间或服务进程用户空间。	分发请求，将结果从用户空间或内核空间传递到客户进程用户空间。

如表 4.1 所示，为了保证向 Elastos 客户程序提供远程对象服务和内核对象服务。定义了 Elanix 中客户程序与远程对象或内核对象的通信层次。每个层次提供服务帮助上个层次之间通信，且每个层次对于上个层次来说都是透明的，可替换的。比如第一层传输层，远程对象服务会采用消息队列的方式传递数据，而内核对象服务会采用字符设备文件的方式传递数据，但这两种方式对于第二层列集层的实现来说没有任何影响。

4.3 命名服务机制

命名服务机制属于CAR构件技术的一部分，类似于CORBA的命名服务[27]，CAR构件技术通过命名服务机制提供一种发布，获取，使用CAR构件的方法。远程对象服务和内核对象服务都是通过命名服务机制向客户程序发布服务的。命名服务是一种以字符串为标识的服务。服务程序可以通过 操作系统API函数EzRegisterService向操作系统注册自己的服务接口，而服务的使用者（即客户端程序）则可以通过API函数EzFindService来获取指定的服务接口。

命名服务通过简单，友好的系统 API 函数，为系统服务以及用户组件提供了一套完整的 CAR 构件使用流程，具有良好的扩展性，并支持基于组件的动态更新和升级。

4.3.1 命名服务设计原则

命名服务完成了将服务与字符串绑定的功能，用户可以通过相应的字符串获得该服务组件对象指针。其设计原则如下：

(1) 用户和内核都可以创建一个命名服务，其它用户可以透明的获得该用户指针，而无需考虑其所在位置空间的不同。

(2) 命名服务 server 端获得的组件指针代表的可以是其 server 进程内空间创建并运行的组件，亦可以是 Elastos 操作系统内核提供的以接口形式提供的内核功能，也可以是 server 端所获得的其它进程所实现的组件服务。

(3) 用户一旦通过命名服务获得某个组件服务，其与组件服务交互的过程不再需要与命名服务相关联。

(4) 相对于命名服务所获得的组件服务，命名服务亦是一个普通的组件使用者，提供命名服务的程序可以选择在一个合适的时间点释放该组件服务指针。所有适应于 CAR 构件生命周期管理的操作和语义亦适应于命名服务。

(5) 命名服务的生命周期由命名服务创建者所控制，但使用命名服务的程序可以通过系统事件，或者进程间通讯给服务创建者相关信息，命名服务创建者可以通过对服务使用情况的分析来选择退出时机。

(6) 相关组件服务退出，命名服务不再有效，其它再试图通过命名服务获取相关组件服务的程序将返回错误。

(7) 相关组件服务动态更新或者升级，不影响其它通过命名服务获取该服务的程序，其依然可以无需改动，无需重新编译地正常工作。

4.3.2 命名服务机制的实现

命名服务包含两部分，一部分指用户获得某个组件接口指针（可以是用户自己的组件，亦可以是用户远程获得的组件指针）后，通过 EzRegisterService API 函数向内核注册一个命名服务，用户选择合适的时间注销命名服务，并释放资源，被称为这部分内容为命名服务服务器端；用户通过 EzFindService API 函数获得相应的组件服务并应用，被称为命名服务客户端，工作流程如图 4.1。

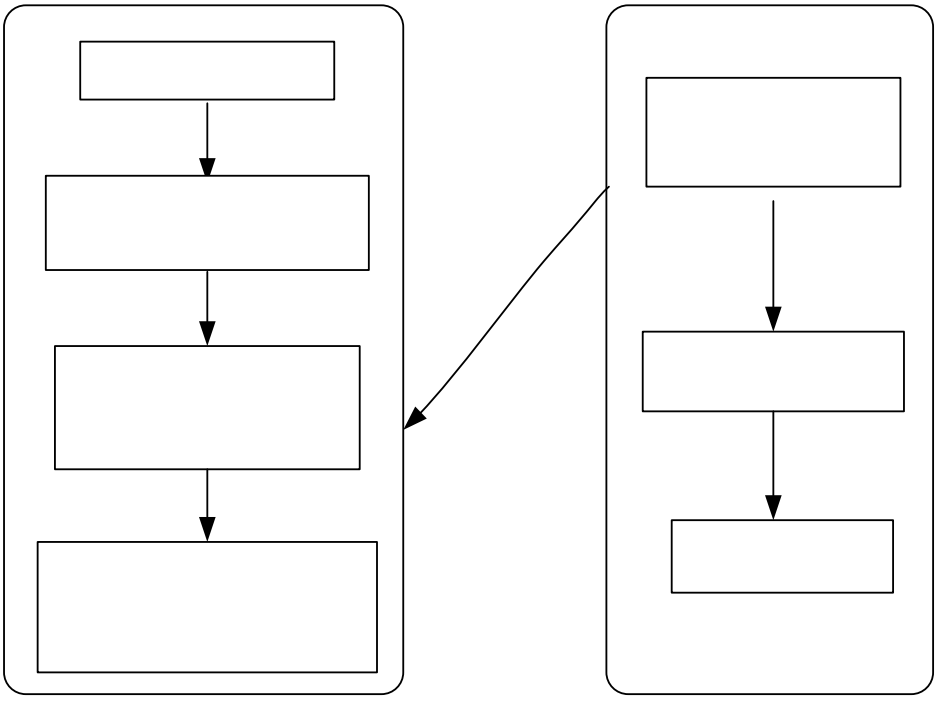

图 4.1 命名服务机制的工作流程图

命名服务机制的实质是将一个组件和指定的字符串绑定的过程，组件使用者可以远程通过字符串查询该组件，并获得组件服务。命名服务本身即可以作为一个单独的构件存在，亦可以做为处于内核的 Elanix Server 的一部分。目前 Elanix 虚拟操作系统采取的策略是将其做为 Elanix Server 的一部分，以提高效率。

远程 CAR 组件在创建的时候会向 Elanix Server 注册相关信息，并建立存根，称之为 Stub，远程用户获得组件指针的时候在自己进程空间建立代理，称之为 proxy。

远程 CAR 组件在 Elanix Server 注册的信息代表了该组件对象的存在，一个内核对象 Object 代表某个组件的注册信息，通过这些信息可以找到相关建立代理所必须的信息，另一方面，亦可通过这些信息找到该远程组件以及组件服务相关信息。

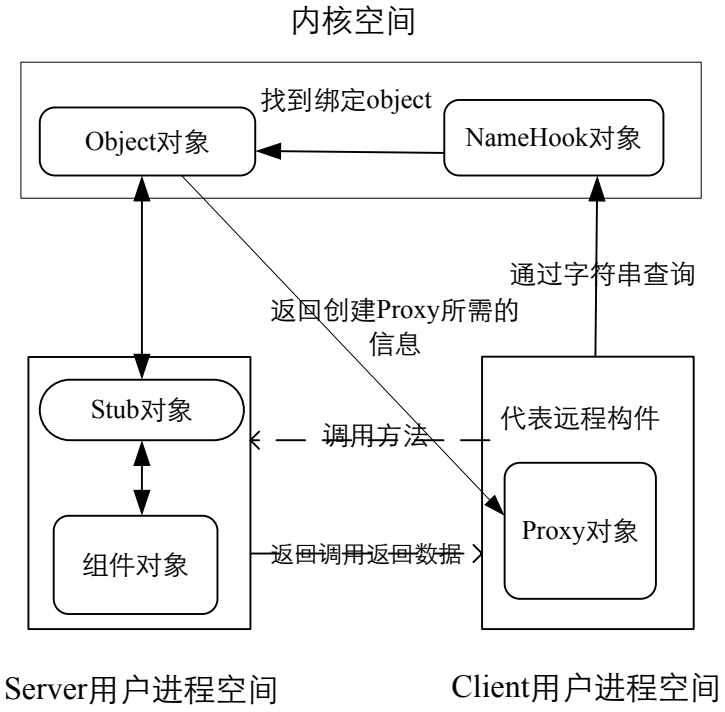

图 4.2 远程构件的命名服务

简单的通过命名服务机制获得远程构件服务的如图 4.2，命名服务的实现主要是通过在 Elanix Server 创建代表一个命名服务对象 NameHook， 然后将该对

象与与该命名服务绑定的组件对象在 Server 的注册信息 Object 相绑定。NameHook，Object，Stub 对象在系统里面存在一一对应的关系，这样通过字符串就可以找到相应的 NameHook，通过 NameHook 则可以发现相应组件对象的 Object，通过 Object 又可获得足够建立 Proxy 的信息。

上述过程即 EzFindService API 函数的实现流程，而对于注册一个命名服务，则是通过 API 函数 EzRegisterService 完成的，该 API 函数将一个字符串与一个组件接口相绑定，实际上这里的组件接口也可能是一个远程组件接口代理，这样就分成两种情况做不同的处理：

接口是一个远程组件接口代理，则通过接口代理，找到相应的 Object， 然后创建命名对象 NameHook, 将二者关联起来。

接口是一个本地对象接口指针，则创建或者找到相应的存根 stub，然后向内核注册相关信息，并生成 Object 对象，创建命名对象 NameHook, 将二者关联起来。

调用 EzUnregisterService 注销命名服务，其实是取消 NameHook 对象与相应 Object 对象的关联，以及销毁 NameHook 的过程。 在取消关联的同时会释放相关组件指针，并不再允许用户通过命名服务获得组件服务。

鉴于命名服务和关联组件可能属于不同用户所构建，则关联组件有可能提早被其所属进程退出，或者发生异常，提前退出，Server 则会检测到这种情况的发生， 销毁 NameHook 以及 Object 对象以防止该组件指针再被用户利用。而以往获得的组件指针将会全部失效，在调用过程中 Server 会返回一个错误值，以通知用户。

4.3.3 命名服务机制的优点

用户程序通过字符串获取相关服务，而服务则可由系统，以及其它用户程序提供，这样将服务和服务使用者隔离的方式，减小了代码的耦合性，极大的增强了代码的可扩展性，安全性，同时也最大的发挥了系统的功能。

从扩展性来看，用户可以通过在保持接口定义不变的情况下，修改服务程序代码，升级服务程序；另一方面用户亦可以通过提供新的接口，来扩展新的功能，新的用户可以利用新的接口，而旧的用户则不会产生影响，其代码可以

不经修改，不经重新编译，正常运行。

进一步讨论命名服务机制的扩展性，用户通过 CAR 构件方式实现的程序，都可以通过命名服务机制的方式，提供给其它远程用户。

从安全性来看，用户可以通过将一个信任度不高的服务启动在一个单独的进程中，并通过命名服务机制获得，这样就通过进程地址空间机制，隔离了服务与用户，同时服务之间的数据交换可以经过系统的构件平台数据交互机制的检测。

命名服务机制允许将系统提供的各种服务接口，比如进程，线程，module，同步对象等与对应的字符串绑定，其它进程可以通过命名服务机制非常方便的获得该进程，从而能很方便的实现进程间通讯，扩展了系统的功能。

命名服务机制以简单的三个 API 函数，提供对服务组件和字符串的绑定，到获取，到注销的整套机制，用非常简约而且容易理解的方式提供给用户， 体现了软件设计里面简单即是美的设计理念。

Elanix 虚拟操作系统利用命名服务机制来传递共享内存指针以及一些同步对象指针，从而实现跨进程的进程间通讯。

4.4 基于元数据的自动列集散集机制

元数据(metadata)，是描述数据的数据(data about data)。为了使得构件运行良好并实现构件间的互操作，构件模型必须提供元数据已暴露/获得构件接口定义以及其它的构件特征的途径，并需要对这些途径进行标准化[28]。

事实上，构件技术出现伊始，各构件模型中就已经分别有专门的技术来满足上述需求了。微软的构件对象模型COM（Component Object Model）提供了Type Library以及相关API来暴露构件接口定义[29]，.NET中则使用编译托管代码后与MSIL（Microsoft Intermediate Language，微软中介语言）一同产生的metadata来描述“它所关联的托管代码”所定义的类型[30]；CORBA[31]的接口定义语言（IDL[32]，Interface Definition Language）有良好的接口映射机制，可将接口定义信息应用于解决异构的操作系统、开发商和编程语言之间的互操作问题；Sun公司的EJB（Enterprise JavaBeans）[33]使用了Java所提供的语言级反射技术来在运行时获得一个类的所有方法的信息。构件元数据能够提供对如接口定义等构件

静态结构以及其它动态信息的形式化描述及其标准化访问途径，极大地推动了构件化系统地整合和维护过程，使得复用构件成为了构造大规模软件系统的关键技术[34]。

4.4.1 CAR构件元数据

CAR 构件以接口方式向外提供服务，构件接口需要元数据来描述才能让其他使用构件服务的用户使用。构件为了让接口与实现无关，从而保持了接口的不变性，使得动态升级成为可能；并且将接口的内部实现隐藏起来，由接口的元数据来描述接口的函数布局和函数参数属性。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不同构件之间的调用才成为可能，构件的远程化，进程间通讯，自动生成代理和存根及自动列集散集才能正确地进行。

CAR构件编写者在写一个自己的构件时，第一件事情就是要写一个CAR文件，构件编写者使用CAR语言在这个文件里描述自己的构件接口，接口方法以及实现接口的构件类等等，CAR工具会根据这个CAR文件为构件编写者生成代码框架以及其他构件运行时所需要的代码，构件编写者只需关心自己接口方法的实现。类似于微软的ODL（Object Definition Language）[35]文件，CAR文件描述了一个构件里所包含的构件类的组织信息以及接口和构件对象的标识等信息。

CAR构件的元数据是CAR文件经过CAR编译器生成的二进制格式信息。当然元数据可以用XML[36]等形式描述，但使用二进制格式更高效。元数据与构件的实现代码一起被打包到构件模块文件中。元数据记录了构件接口及构件类的定义信息，是构件自描述的基础。在CAR里，ClassInfo被作为构件程序的元数据信息，用于描述构件导出的接口及方法列表。同时ClassInfo也是自动生成构件源程序的基础。

在目前的 CAR 构件开发环境下 ClassInfo 以两种形式存在：一种是与构件的实现代码一起被打包到构件模块文件中，用于列集和散集用的；另一种是以单独的文件形式存在，存放在目标目录中，最终会被打包到 DLL 的资源段里，该文件的后缀名为 cls。这个 cls 文件和前者相比，就是它详细描述了构件的各种信息，而前者是一个简化了 ClassInfo，如它没有接口和方法名称等信息。cls 文件

就是 CAR 文件的二进制版本。由于前者只是用于 CAR 构件库的实现，接下来要说明的是后一种 ClassInfo, 这是用户需要关心的。

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

表 4.2 CAR 构件元数据内容清单

构件模块信息	构件类信息	接口信息	方法信息
构件模块版本信息 模块名称以及 uunm 模块包含类的个数 所有构件类的信息 模块包含接口个数 所有接口的信息 数据结构的个数 数据结构的信息	构件类的名称 构件名的标识 构件类所在模块 构件类的属性 实现的接口个数 所有接口的信息	该接口的名称 接口标识 IID 接口所在模块 接口的属性 拥有的方法个数 所有方法信息	方法名称 方法返回值类型 方法参数的个数 所有参数信息

对于方法的每个参数有参数名及参数属性构成。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 IUnknown 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。

4.4.2 以类为单位的代理存根机制

在通常的构件技术中，用户调用是以接口为单位。而构件的实现，则是以类为单位，一个类可以包含多个接口。从远程接口的自动列集\散集的实现来看，即可以选择以类为单位进行列集散集，又可以选择以接口为单位进行列集散集。Elanix 虚拟操作系统选择以类为单位进行远程接口的代理存根机制。这里有两个方面的原因。

一方面对于同一实现类的构件接口，用户常常通过 QuerryInterface 而获得。如果采用以接口为单位的方式进行自动列集散集，就要远程获取数据，重新建立存根代理，并返回相应的接口代理给用户。而远程的数据获取，尤其分布式

环境的情况下非常耗时。这样就促使尽量少的跨远程获取数据，每次获取都尽量带够足够多的信息，以类为单位就符合这种理念。当用户通过 QueryInterface 获取接口的时候，所以的处理都在客户端进行。

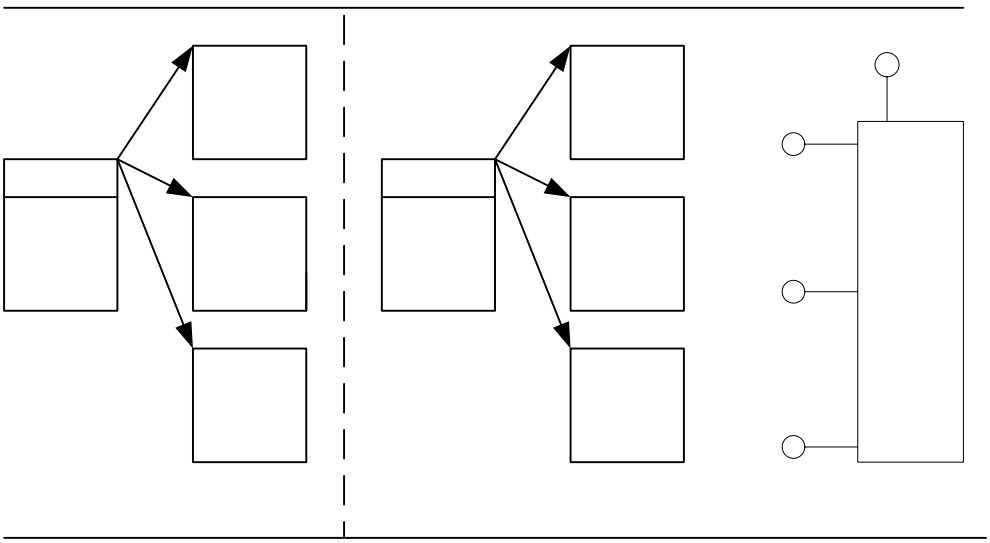

图 4.3 以类为单位的代理存根机制图

如图 4.3 所示, 一个实现了接口 IA, IB，IC 的构件类为实例简单描叙一下远程接口的相对应的数据基本对应关系。对于每个构件实现的接口，存根和代理都会有相对应的接口存根和接口代理相对应，当用户要求获得相应的远程接口的时候,系统就返回相应的接口代理给用户。

通过接口代理传递来的调用请求，会转化到相应的接口存根上，接口存根再将调用请求转发给真正的构件类实现。用户如果获得远程接口 IA，通过其查询 IB 的过程实际上是接口代理 IA 通过代理类查询到接口代理 IB 的过程。整个过程都发生在客户端空间，而无需进行远程的调用。

另一方面在现实的实现中，往往都采用以构件类为最小的生命周期管理单位。所有对于接口的生命周期管理最后都会归结到对接口所在的实现构件类的引用计数的操作和处理。因而采用以类为单位实现远程接口的自动列集\散集也是更加符合现实的构件生命周期管理模型的。类似于 QueryInterface，以类为单位的列集\散集，对于接口的生命周期管理亦可减少跨空间的调用.同时。从实现的角度来看，以类为单位并不会由此付出太多的代价。

存根、代理代表构件服务的存在。用户获得一个远程构件的存在，则是以其进程内创建了一个代理对象为标志，而一个进程提供一个远程服务，则是以其进程空间内存在一个存根对象为标志。代理和存根可以是多对一的关系，存根和一个构件实现类的实例则是一对一的关系，如图 4.4。

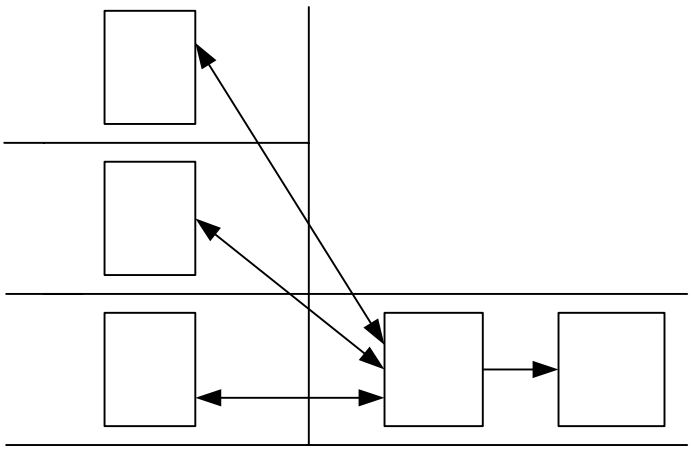

图 4.4 代理对象与存根对象关系图

存根和构件对象实例由于处于同一进程空间，故存根对象可以直接持接构件对象的指针。而代理和存根对象之间的对应关系则是由系统分配系统唯一的资源 oid 所建立的。相应的代理对象和存根对象拥有相同的 oid，这样代理对象就可以找到相应的存根对象。

用户进程1

代理对象

用户进程2

代理对象

用户进程3

代理对象

...

4.4.3 自动列集散集过程

Elanix Server 中的远程 CAR 构件对象和内核对象都是按照 CAR 构件规范编写的，都有完备的元数据。图 4.5 给出了与列集散集过程相关元数据的 C 语言定义。该部分元数据描述各个内核对象类、接口、方法的各自属性，以及它们之间的关系。用户程序调用内核对象服务的列集散集过程描述如下：

```
typedef struct _MethodEntry {
    UINT8          paramNum;
    CIBaseType     *params;
}MethodEntry;
typedef struct _InterfaceEntry {
    IID            iid;
    UINT16         methodNum;
    MethodEntry    *methods;
}InterfaceEntry;
typedef struct   _ClassEntry {
    CLSID          clsid;
    UINT16          interfaceNum;
    InterfaceEntry    *interfaces;

}ClassEntry;
```

图 4.5 与列集散集相关的元数据结构

(1) 当新建一个内核对象的时候，会实例化 CInterfaceStub 类为该内核对象实现的每一个接口创建一个接口存根，同时为该内核对象创建用 oid 唯一标识的 Object 对象，通过 Object 对象可以索引各个接口存根。

(2) 当用户程序请求某个内核对象接口时，将会在用户空间中实例化 CInterfaceProxy 类，为该内核对象接口创建接口代理，并将此接口代理的指针作为内核对象接口指针返回给用户程序。

(3) 当用户程序调用内核对象接口中的方法时，因为此时用户程序得到是接口代理的指针，而且在 C++对象模型中，接口即是纯虚基类，纯虚基类由一张虚函数表和指向该虚函数表的指针组成，所以用户程序中对虚函数表指针的操作，就变成对接口代理成员变量 m_pvVptr 的操作（当且仅当 m_pvVptr 是接口代理的第一个成员变量）；用户程序对虚函数的调用，就变成对 mtable 表中对应序号的 method 函数的调用。接着 method 函数调用该接口代理中的 ProxyEntry 方法，同时把自己在 mtable 中的序号做为参数也传递给 ProxyEntry。

(4) ProxyEntry 根据该方法的元数据将 Oid 标识、接口序号 uIndex、方法序号和以及输入参数打包。然后将打包好的数据通过 SysInvoke 函数发送给对应接口存根，并等待 SysInvoke 函数返回结果。

(5) SysInvoke 能够从用户空间穿越到内核空间，根据 oid 找到对应的

Object 对象，然后，根据接口序号 uIndex 找到对应的接口存根，调用该接口存根中的 Invoke 方法。

(6) 接口存根中的 Invoke 根据元数据信息解包接口代理发来的数据，然后调用真正内核对象接口中对应序号的方法，得到返回结果。再使用元数据将返回结果打包。

(7) SysInvoke 返回后，接口代理中的 ProxyEntry 解包传来的结果，返回给用户程序。

在元数据的支持下，列集层在运行过程中动态自动生成内核对象的代理和存根，摒弃了通过静态编译生成存根、代理的方式，避免了静态编译生成存根、代理常常造成代码量增大，代码逻辑显得极其复杂等问题。

远程对象服务采用与内核对象服务相似的列集散集过程。但底层数据通信方式与内核对象服务有所不同。内核对象服务的传输层使用字符设备文件传输数据，而远程对象服务采用消息队列传输数据。

4.5 CAR构件对象生命周期管理

CAR 远程构件生命周期的管理用于对构件的生存期管理以及对于构件在远程化过程中所需要的资源的生存期管理，其着重于解决远程构件在远程调用过程所必须的资源的创建，计数，以及销毁。

构件本身的 CAR 构件本身的生命周期可以通过其引用计数来控制, 其必须实现的标准构件接口方法 AddRef, Release 可以用来控制引用计数，其方法实现与 COM 相同; 远程 CAR 构件之间的调用则是通过动态生成的存根代理间接完成，所以远程构件生命周期管理亦牵涉到对存根代理的存活期的管理；构件服务进程做为构件服务的载体, 其活动时间亦和构件对象的生命周期紧密相连。

Elanix 采用了“以类为单位”的代理存根机制，其优点之一就是能够在本地实现接口查询和引用计数增减，但同时也带来了一定的复杂性。CAR 的远程构件生命周期管理包含如下五个原则：

(1) 通过引用计数来动态控制构件对象以及存根，代理对象的存活期；
(2) 服务器进程控制它自己的生命周期，可以在任何它意愿的时候终止自己，服务器进程的设计者可以选择将其生命周期和构件对象的生命周期挂钩；
(3) 远程客户端进程在使用完构件前必须增加构件服务的相关引用计数，以防止在使用过程中服务退出，使用完服务后要释放相关引用计数；
(4) 服务器进程异常退出，则所有的构件服务的远程客户端获得的指针将失效，其相关资源依然可以正常释放；
(5) 远程客户端异常退出，其所拥有的远程构件指针以及资源将全部释放；

因为 Elanix 对构件的计数是以对象为单位，远程对象接口、接口代理、接口存根都与生命周期管理无关。可以将与远程对象生命周期管理相关的对象分为五个部分：

表 4.3 与生命周期管理相关分类

部 分	位 置	说 明
第一部分	客户进程空间	调用远程 CAR 对象的客户程序
第二部分	客户进程空间	为远程 CAR 对象生成的对象代理 CObjectProxy
第三部分	内核空间	远程 CAR 对象的命名服务 NameHook 和对应 Object 对象
第四部分	内核空间或服务进程空间	为 CAR 对象生成的对象存根 CObjectStub
第五部分	内核空间或服务进程空间	远程 CAR 对象

不同类别部分的活动将会对相连部分的引用技术产生影响。活动的发起者一般是客户线程请求调用远程对象服务或者内核对象服务，但也可能是注册命名服务。客户线程的第一次调用将会使每一部分初始化引用计数。此后客户线程的活动将产生连锁反应，前一部分的引用计数增减将影响到后一部分的增减，

最终客户线程的计数将反应到服务端的 CAR 对象创建和销毁上。

由此可以将与远程对象生命周期管理相关的活动分为四类：

表 4.4 生命周期管理活动

活　动	说　明
对象存根 CObjectStub 的创建与销毁	对象存根的创建将会使远程 CAR 对象的引用计数加一； 对象存根的销毁将会使远程 CAR 对象的引用计数减一； 对象存根的引用计数增减不会影响 CAR 对象的引用计数； 创建对象存根时也会创建 Object 对象，但这不会增加对象存根的引用计数；
命名服务的注册与注销	命名服务可以看成是和 Proxy 相同的 CAR 对象客户； 注册命名服务时对象存根引用计数加一； 注销命名服务时对象存根引用计数减一；
对象代理 CObjectProxy 的创建与销毁	对象代理与对象存根是多对一的关系； 对象代理创建时对象存根引用计数加一； 对象代理销毁时对象存根引用计数减一； 对象代理的引用计数增减不会影响对象存根的引用计数。
用户程序调用远程对象	用户程序使用远程对象接口将会影响对象代理的引用计数； 一个用户程序中只有一个该远程对象的对象代理；

由上可知，各部分分离使得生命周期管理具有以下特点：

(1) 对象代理的生命周期管理大部分在本地执行，只有创建和销毁的时候才会影响远程的对象存根引用计数。

(2) 命名服务和对象代理的地位同等，只有在注册和注销的时候才会影响远程对象存根的引用计数。命名服务和对象代理分开，两者同时存在或只存在其一的情况下，远程对象存根都会存在，只有命名服务和对象代理都不存在了，远程对象存根才会销毁。

(3) 对象存根的生命周期管理大部分自己完成，只有创建和销毁的时候才会影响 CAR 对象的引用计数。从这个角度说对象存根和同一进程内的 CAR 对象本地调用的地位相同，两者同时存在或只存在其一的情况下，CAR 对象都会存在，只有对象存根和本地调用都不存在了，CAR 对象才会销毁。

第5章 Elanix开发与测试

5.1 Elanix开发与测试环境

5.1.1 Elanix开发环境

Elanix 环境是为 Elanix 虚拟操作系统项目在 Linux 操作系统上搭建的环境。Elanix 环境目前设计为三种用途。这三种用途按重要性和优先级排列如下：

- 首先 Elanix 环境是 Elastos Linux 虚拟机的开发环境
- 其次 Elanix 环境是 Elastos 应用程序的运行环境
- 再次 Elanix 环境是 Elastos 应用程序的开发环境

在 Elanix 环境中编译生成一个程序的流程比较复杂，涉及到 setenv，emake，makfile.gnu 等多个脚本。其中 makefile.gnu 是最为重要的，它是一张大的通用的 makefile，用户在自己目录中写的 sources、dirs、makefile.inc 等文件都将嵌入到该 makefile 中。makfile.gnu 中已经设置了多种操作，用户在 source 等文件中设置的不同宏定义，将使得 makefile.gnu 做不同的操作。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_USECOMPTR
LINK_FLAGS=-ldl
SPECIAL_TARGET= $(SYSTEM_BIN_PATH)/$(TARGET)

SOURCES= \
        elanix.cpp \
        loader.cpp \

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

图 5.1 sources 文件示例

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。该 sources 文件将嵌入到一张通用 makefile.gnu 文件中，现将通用 makefile 文件结构用表 5.1 表示：

表 5.1 通用 makefile 结构

部　　分	说　　明
Build environment	依据环境变量，设定了外部工具等路径
Set target	设定编译针对的开发板的类型
Set TARGET paths	设定目标代码路径，
Find sub dirs/files to build	将源代码目录中的 sources 和 dirs 文件包含入本文件
Select C/C++ compiler and assembler	设置编译工具
Set Linker and its flags	设置了编译和链接选项
Other build tool macros	一些辅助基本命令如复制、创建目录、删除
Tidy up custom specified macros	清理一些辅助的宏，包括 TARGET、LIBRARY 等
Create OBJECTS from SOURCES	规定了源文件编译生成的目标文件，默认 Include 路径
First all	第一次进入 makefile 时，执行的工作，主要是创建或者删除特定目录
Building rules	各种源文件编译的实际命令
Second all	第二次进入 makefile 时，执行的工作，主要是实际编译和链接
depend.mk	生成目标文件对头文件的依赖关系

Elanix 环境实际上是由添加了环境变量和 alias 的新 Bash，以及一些工具软件组成。Elanix 环境目前可以将 C/C++程序和.S 汇编程序编译为 Linux 下的可执行文件、静态库.a 文件、动态链接库.so 文件和 Linux 内核模块.mo 文件，并区分了 dbg 和 rls 版本。Elanix 环境目前可以通过使用编译生成的 elanix.exe 命令，运行 Elastos 应用程序。

5.1.2 Elanix自动测试工具

Elanix 这样的项目需要划分成不同的功能模块，由不同的人去完成。必须保证各人负责的不同模块能够完美地结合在一起而不会相互“排斥”。按照以往的办法，是在开发过程中，各模块自己保证自己的正确性，到了开发末期，可以进行联调。将各模块整合在一起，然后进行各项测试工作。这样做相当于是把大部分测试和修 BUG 的工作都堆在了工程的后期进行，并且都是人手操作的。必须想办法把这些测试工作比较均匀地整合在整个工程的全过程中进行，使其自动化。在整个开发过程中，使用比较直观的办法把最近一段时间的各项性能指标的变化显示出来，这样可以更加方便地知道每天的提交的代码产生了一些什么样的影响，并决定是否应该把某些提交代码给返回。所以 Elanix 项目中采用一个自动反应每天代码与测试情况工具 Daily Build 来解决以上的问题。

Daily Build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。DailyBuild包括了回归测试[37][38]。目前实现的功能是的回归测试和代码行统计，以及使用gnuplot生成图表，如图 5.2。

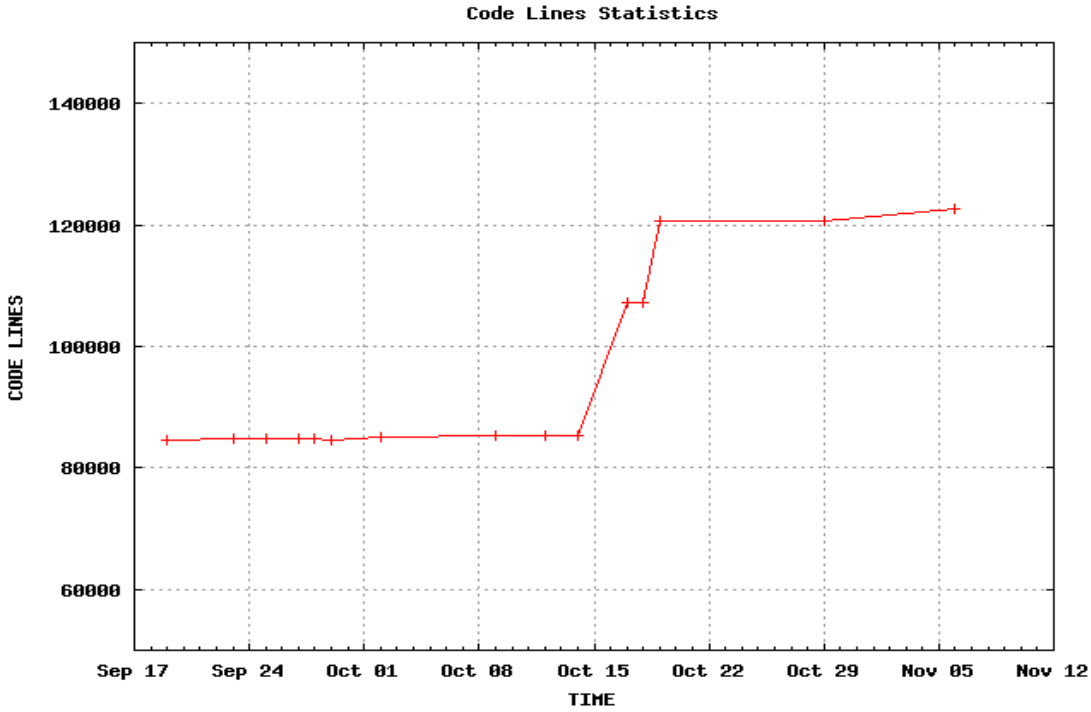

图 5.2 代码行数统计图示例

表 5.2 Daily Build 的基本流程

测试步骤	步骤说明
代码的更新	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
	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
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
	设置编译环境为 debug 环境，并编译所有为 debug 版本
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。
代码行统计并生成图表	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一个名为 statistics.txt 的文本文件里。
	使用 gnuplot 根据 statistics.txt 中的统计数据生成图表

Html 页面是通过把全部测试过程中的结果直接和 html 标签结合起来输出，然后可以把所有这些输出保存到一个文本文件中（重定向到指定的文件），这个文本文件就含有最后测试报告的 Html 页面，用浏览器直接打开该文件就可以看到测试报告。

5.2 Elanix与Wine比较

Wine 是 Linux 上 Windows 虚拟操作系统。与 Elanix 一样，Wine 也必须模拟部分 Windows 内核的功能，向运行在 Wine 上的 Windows 应用程序提供共享数据、进程线程管理、同步、互斥、信号量、消息队列等等系统服务。在 Wine 中，由 Wine Server 负责这部分工作。可以说 Wine Server 是 Wine 的核心部件，它通过一整套机制完成了上文提到的功能，为 Wine 应用程序提供服务。

由于 Wine Server 和 Elanix Server 实现机制不同，通过测试进行性能比较分析可以说明两者的各自优势。

5.2.1 Wine Server特点

Wine Server 的机制有三大特点：Socket 连接、单线程轮询、共享内存传参数。

Wine Server 是一个独立的 Linux 用户进程，它建立了一个 Socket 服务，Wine 其他用户进程中的线程都做为 Socket 的一个客户端与 Wine Server 相连。虽然 Wine Server 和 Wine 应用程序之间是一对多的关系，但是并不是一个时间内系统中就只有一个 Wine Server 在运行。准确的说应该是一个时间内系统中的一个用户只能运行一个 Wine Server，如果有多个用户在使用 Wine，那么 Linux 系统中就可能存在多个 Wine Server。

Wine Server 与各个 Wine 用户进程线程相连后，这些线程就可以通过 Socket 发送请求，访问 Wine Server 内存放的共享数据。为了保证这些共享数据的完整

性和一致性，Wine Server 采取单线程轮询机制来处理每个客户线程发来的请求。也就是说，Wine Server 中只有一个线程，对客户端发送的请求，是一个接一个处理的，一个请求未处理完毕，是不会处理下一个请求的。所以 Wine Server 总是处在一个大的 poll()循环中。每次循环 Wine Server 会知道发生了什么，比如有客户线程发来了请求或是一个等待条件已经满足。Wine Server 这样做的好处是它的所有步骤都可以看成是“原子操作”，内部不存在竞争造成的风险。这种方法简单又可靠，只是速度很慢。

Wine Server 采用 Socket 连接和单线程轮询的方式，确实可以模拟部分 Windows NT 内核功能。但这样做的代价是对客户请求响应缓慢。所以 Wine Server 和每个客户端的线程共享一小块内存。这样可以加快 Wine Server 和 Wine 用户进程间的数据传递。如果每个线程的系统调用的数据都通过 Socket 与 Wine Server 相互传递，那么 Wine 的性能就会大大降低。现在采用的方法是，当 Wine 用户进程内的一个线程进行系统调用时，就会把系统调用需要的参数放在共享内存中，然后通过 Socket 向 Wine Server 发送一个命令码，接着就是堵塞自己直到 Wine Server 返回处理完毕的信号。Wine Server 接到命令码后会在合适的时间加以处理，处理的时候从对应线程的共享内存中取出参数，根据命令码做相应操作，最后再将结果放入共享内存，通过 Socket 通知对应客户线程，处理完毕。

5.2.2 性能测试结果与分析

此次测试专门 Wine Server 和 Elanix Server 对客户程序请求的响应时间。考虑到 Wine 与 Elanix 模拟的操作系统不同，前者是 Windows，后者是 Elastos。Wine 和 Elanix 上在客户程序的 API 函数转换上处理不同，Wine Server 和 Elanix Server 对请求的处理也不相同。所以此次测试修改了 Wine Server 和 Elanix Server 定义了一个新的空系统服务 NoOperation，Wine Server 和 Elanix Server 对该系统服务请求不做任何处理，直接返回空值。另外在客户端直接在 Wine 的 Windows API 转换层和 Wine Server 的相连处 ntdll.dll.so 中发送对 NoOperation 的系统服务请求，同样在 Elanix 上是在 Elastos API 转换层和 Elastos Sever 的相连处 elastos.so 中直接发送对 NoOperation 的系统服务请求。

测试使用循环不断发送请求，记录下循环 100 万次，200 万次，300 万次的

时间。用总时间除以循环次数，可以得到单次响应的时间。再求出三者平均值，就可以得到平均单次响应时间。

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

软件环境：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。

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。得到测试结果如表 5.3 所示。

表 5.3 Elanix Server 和 Wine Server 的性能测试结果

VOS	1 百万次	2 百万次	3 百万次	平均单次
Elanix Server	0.579352 (ms)	1.158849 (ms)	1.737585 (ms)	0.579324 (μs)
Wine Server	10.28146 (ms)	21.01983 (ms)	31.63047 (ms)	10.44497 (μs)

可以看到 Wine Server 的响应时间明显要大于 Elanix Server 的响应时间。前者平均单次响应时间是 10.44497 微秒，后者平均单次响应时间是 0.579324 微秒，仅为前者的 1/18。原因在于，Elanix Server 采用了完全不同于 Wine Server 的通信机制。

最重要的原因是，Elanix Server 是一个处于 Linux 内核空间的内核模块，响应客户线程请求不需要切换进程上下文。而 Wine Server 作为一个独立的用户进程，每次响应客户进程中线程的请求，都必须在两个进程上下文间来回切换。每次进程上下文切换包括保存当前进程环境、刷新 CPU TLB、运行调度算法、设置 MMU、恢复下一个将运行的进程环境。这大大增加了 Wine Server 的响应时间。

另外 Elanix Server 作为一个 Linux 内核模块为客户线程提供服务，客户线程与 Elanix Server 之间是通过字符型设备文件互连。在 Elanix Server 中通过 Linux 内核提供的 get_user()和 put_user()函数，读写客户线程传递的数据。这种连接方式要快于 Wine Server 采用的 Socket 连接方式。

虽然和Elanix Server相比，Wine Server对客户线程的请求响应缓慢，但它有可移植性方面的优势。Wine是为符合POSIX[39][40]标准的操作系统开发的。Wine的所有模块都是用户程序，非常容易在类Unix的平台上移植。事实上Wine可以

在多种类Unix操作系统上运行，包括Linux、Solaris、FreeBSD等等。

Elanix Server 是 Linux 内核模块，与 Linux 内核耦合性较强。所以 Elanix 可移植性不如 Wine。目前 Elanix 还没有要在其他类 Unix 操作系统上移植的需求。但 Elastos 应用程序的运行对 Elanix Server 响应时间要求很高。Elanix Server 的实现方式满足了 Elastos 应用程序运行以构件为单位的要求。

第6章 工作总结与展望

6.1 工作总结

本文对虚拟操作系统技术和构件技术进行了详细的分析和讨论。虚拟操作系统是虚拟机技术的一部分。虚拟操作系统（Virtual Operation System）只是对操作系统进行抽象，在新的操作系统环境中虚拟出原来操作系统的环境，使得应用程序可以在不用改动代码的情况下，在新的操作系统上运行。构件技术的一个目标是构件具有平台无关性。一个开发好的构件，理想情况下，应该可以在任意操作系统，软件平台上独立部署，正常运行。虚拟操作系统的作用是为构件提供一个虚拟的运行环境，让构件具有跨操作系统平台的性能。

本文研究了虚拟操作系统与构件技术相结合的相关机制，实现了一个具体应用——Elanix 虚拟操作系统。Elanix 是二进制级别的虚拟操作系统，可以让编译生成后的和欣操作系统（Elastos）的应用程序直接运行在 Linux 操作系统上。本文分别设计并实现了 Elanix 虚拟操作系统的三个部分：PE 加载器（Loader），API 转换层（Elastos.so），内核对象服务模块（Server），并着重对 Elanix Server 模块进行了详细地说明。Elanix Server 采用 Linux 模块技术，可以在操作系统运行时动态地加入 Linux 内核空间中，Elanix Server 中的数据可以被所有 Elanix 进程共享，且能得到有效防护，避免用户程序直接对数据操作；而且 Elanix Server 易于发布，便于安装。Elanix Server 通过自己独立实现和引用 Linux 内核相关数据及函数的方法，将 Linux 内核功能模块封装为与 Elastos 相同的内核对象，向用户程序提供与 Elastos 操作系统无差别的内核对象服务。

本文还对 Elanix 虚拟操作系统中的 CAR 构件支撑技术进行了研究。CAR 构件技术是 Elastos 操作系统的最大特色，Elanix 作为 Elastos 在 Linux 上的虚拟操作系统，必须实现对 CAR 构件的全面支持。Elanix 使用 PE 加载器可以将 CAR 构件装载到进程空间中。Elanix 定义了四层通信层次，确保了 CAR 构件远程对象调用和内核对象调用的顺利进行。Elanix 中的命名服务机制提供一种发布，获取，使用 CAR 构件的方法。远程对象服务和内核对象服务都是通过命名服务机

制向客户程序发布服务的。命名服务是一种以字符串为标识的服务。而且 Elanix 基于元数据实现了自动列集散集机制，避免了代码量过大，代码逻辑显得极其复杂等问题。

综上所述，本文对虚拟操作系统技术与构件技术的结合作了深入的研究，进一步丰富了 CAR 构件运行机制，使得 CAR 构件可以二进制直接运行在 Linux 操作系统之上。

6.2 工作展望

Elanix 虚拟操作系统是 Elastos 操作系统和 Linux 操作系统的结合，从 Linux 操作系统的角度看，Elanix 虚拟操作系统为 Linux 提供了支持 CAR 构件加载与运行的相关支撑技术，从 Elastos 应用程序的角度看，Elanix 虚拟操作系统依靠 Linux 为 Elastos 的跨平台应用提供了强大的功能和高效的性能。处于 Linux 内核中的 Elastos Server 正是两者的结合点，如何为 Linux 引入 Elastos 的新技术和如何挖掘 Linux 潜力为 Elastos 应用提供更为强大的服务，是 Elanix 进一步要研究和讨论的问题。

参考文献

[1] G. Bracha, J. Gosling, B. Joy, G.Steele. “The Java Language Specification”. Addison-Wesley, 2000.

[2] 李霞，杨传斌. 基于 Java 的软件重用技术及其应用. 微机发展. 2001 年第 2 期

[3] Ruben Pinilla, Marisa Gil. JVM : platform independent vs. performance dependent ACM SIGOPS Operating Systems Review. Volume 37, Issue 2. April. 2003. P44~56

[4] Jeffrey Richter. Applied Microsoft .NET FrameWork Programming. RedMond: Microsoft Press. 2002

[5] Microsoft .NET Home. Available online at http://wwww.microsoft.com/net/

[6] 王宇新，陈锋，徐亮等. Windows 平台上多接口虚拟实时操作系统设计与实现. 大连理工大学学报. 第 43 卷增刊 1. 2003 年 10 月

[7] 徐光祐，史元春，谢伟凯. 普适计算. 计算机学报 2003 年 9 月，第 26 卷 第 9 期.

[8] J.E. Smith and Ravi Nair. An overview of virtual machine architectures, November 1,2003

[9] MingW Online Document Availuable online at http://www.mingw.org/

[10] Uwe Bonnes, Jonathan Buzzard, Zoran Dzelajlija, et al. Wine Developer’s Guide http://www.winehq.com/site/documentation

[11] Wine User Guide. http://www.winehq.com/site/documentation

[12] Biggerstaff, Perlis, "Software Reusability: Concepts and Models", Vol. I & II, ACM Press, Addison-Wesley. 1989

[13] Elastos, Inc. Elastos 1.1 Information Repository. Available online at http://wwww.elastos.com/download.php

[14] 陈榕，苏翼鹏，杜永文等. 构件自描述封装方法及运行方法. 中国专利，公开号 No.1512332, July 21,2004

[15] 潘爱民. COM 原理与应用. 北京：清华大学出版社, 1999

[16] Microsoft Portable Executable and Common Object File Format Specification[S], 1999.

[17] Executable and Linking Format Spec v1.2[S], TIS Committee, 1995.

[18] Randal E. Bryant, David O’Hallaron. Computer Systems A programmer’s Perspective[M], Prentice Hall, Inc, 2003.

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

[20] Alessandro Rubini, Jonathan Corbet. Linux Device Drivers, O'REILLY, 2002,10

[21] 孙天泽，袁文菊. 虚拟设备在嵌入式 LINUX 系统中的使用. 微电子技术. 2003 年 06 期

[22] 李海刚，崔杜武，姚全珠等. Linux 模块技术分析与应用. 计算机工程. 2003 年 1 月.

[23] De Goyeneche, J.-M. De Sousa. Loadable Kernel Modules, E.A.F. Software[J], IEEE.Volume 16, Issue 1, Jan.-Feb, 1999, P65 -71.

[24] Peter Jay Salzman, Ori Pomerantz. The Linux Kernel Module Programming Guide[EB/OL]. http://tldp.org/LDP/lkmpg/2.6/html/, 2005,1.

[25] "C++ ABI: Exception Handling". Availuable online at http://www.codesourcery.com/cxx-abi/abi-eh.html

[26] Halldor Isak Gylfason, Gisli Hjolmtysson, Exceptional Kernel Using C++ exceptions in the Linux kernel. http://netlab.ru.is/exception/KernelExceptions.pdf

[27] 李文军，周晓聪，李师贤. 分布式对象技术. 机械工业出版社.

[28] Katharine Whitehead. 基于组件开发（Component-based Development），王海鹏，沈华峰译。北京：人民邮电出版社，2003

[29] Don Box. COM 本质论(Essential COM) ，潘爱民译. 北京：中国电力出版社，2001

[30] David Chappell. .Net 大局观(Understanding .NET – A Tutorial and Analysis)，侯捷，荣耀译. 武汉：华中科技大学出版社，2003

[31] CORBA. Availuable online at http://www.omg.org/

[32] Catalog of OMG IDL/Language Mappings Specifications. http://www.omg.org/technology/documents/idl2x_spec_catalog.htm

[33] EJB. Availuable online at http://java.sun.com/products/ejb/

[34] Werner Staringer. Constructing Applications from Reusable Components. Software, IEEE, 1994, 11(5):61~68

[35] MIDL and ODL. Availuable online at http://msdn.microsoft.com/library/en-us/midl/midl/midl_and_odl.asp

[36] Extensible Markup Language(XML). http://www.w3c.org/XML/

[37] 郑亚玲，胡和平. 回归测试策略的新领域. 计算机应用与研究. 2000 年第 6 期.

[38] Rothermel, G and M.J. Harrold. A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology, Vol.6, No.2 April 1997, pp173~210

[39] Lewine D. POSIX Programmer’s Guide. NewYork: O’Reilly Book stores, 1991.

[40] IEEE Std 1003. 122001, The Open Group Base Specificat ions Issue 6[S]. New York: The IEEE and The Open Group , 2001.