Elastos

CAR 构件与编程模型
技术文档

上海科泰世纪科技有限公司

2008 年 11 月

版 本 历 史

版本/状态	作者	参与者	起止日期	备注
1.0 / 草稿	蒋伊婷		2007-7-5	初版
1.1 / 修改稿	蒋伊婷		2007-9-20	1、增加部分数据类型的方法和使用的简单示例程序 2、同步 9.5 对象工厂部分的代码
1.2 / 修改稿	蒋伊婷		2007-11-17	增加 5.3.8GUID 生成算法描述
1.3 / 修改稿	裴喜龙		2008-4-23	增加对 applet 关键字的说明
1.4 / 修改稿	裴喜龙		2008-4-28	把文档中未载明的 CAR 关键字及 CAR 属性补全
1.5 / 修改稿	裴喜龙		2008-4-29	增加 CAR 基础类库编程示例
1.6/ 修改稿	张金焕		2008-4-29	增加各 CAR 数据类型构造函数、方法、操作符及宏定义描述; 五元组成员 m_locks 改成 m_reserve
1.7 / 修改稿	裴喜龙		2008-5-7	把基于 IObject 的说法换为 IInterface 的说法
1.8 / 修改稿	陈卫伍		2009-11-1	增加 Elastos3.0 新增加的关键字和属性字，清理已过时的关键字和属性字。相应地修正所有的例子程序使其在 3.0 上编译通过且正常运行

目 录

第一篇　CAR 的背景和规范

第一章　CAR 技术的起源

为了更深刻的理解CAR技术，我们需要了解一些CAR技术的背景知识，它们对CAR技术的形成与发展有着重要的作用。

1.1 构件的产生

在计算机软件发展的早期，一个应用系统往往是一个单独的应用程序。应用越复杂，程序就越庞大，系统开发的难度也就越大。而且，一旦系统的某个版本完成以后，在下个版本出来之前，应用程序不会再有所改变。而对于庞大的程序来讲，更新版本的周期很长，在两个版本之间，如果由于操作系统发生了变化，或者硬件平台有了变化，则应用系统就很难适应这样的变化。所以这类单体应用程序已经不能满足计算机软硬件的发展需要。

从软件模型角度来考虑，一个很自然的想法就是把一个庞大的应用程序分成多个模块，每一个模块保持一定的功能独立性，在协同工作时，通过相互之间的接口完成实际的任务。我们把每一个这样的模块称为“构件”（component），一个设计良好的应用系统往往被切分成一些构件，这些构件可以单独开发，单独编译，甚至单独调试。当所有的构件开发完成后，把它们组合在一起就得到了完整的应用系统。当系统的外界软硬件环境发生变化或者用户的需求有所更改时，并不需要对所有的构件进行修改，而只需对受影响的构件进行修改，然后重新组合得到新的升级软件。图 1.1 体现了这样的一个升级过程。

图 1.1 中，在应用系统版本一的实际使用过程中，由于软件环境发生了变化，构件 1 和构件 4 受到了影响，于是，在保持原来接口的基础上，对构件 1 和构件 4 进行了修改，分别得到了构件 1’和构件 4’。把修改后的构件和其他的构件合在一起得到了新的应用系统版本二，它可以运行在新的软件环境下。于是在不修改构件 2、3、5、6 的情况下，完成了软件的一次升级。

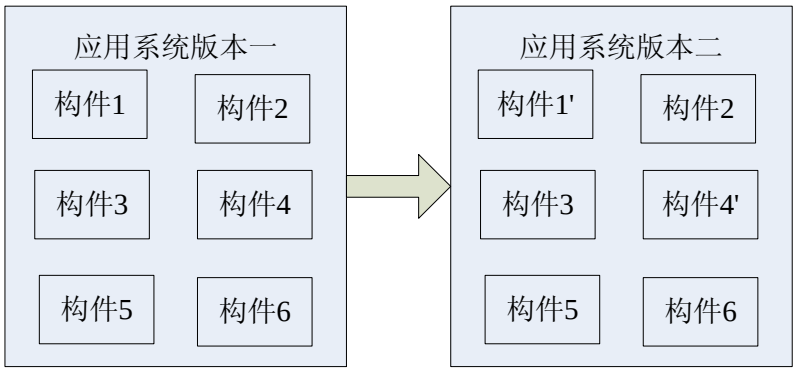


图 1.1　构件化应用程序的一种升级示例(图中没有标出构件的接口关系)

构件化软件结构为我们带来了极大的好处,除了软件升级的灵活性,还有其他一些优势,在后面的论述中我们将可以看到这些优势。有一点需要明确，要实现这种构件化结构模型，并不是很轻松的事情，尤其对于复杂的应用，要把应用分成一些独立的构件，而且这种切分

还要尽可能符合系统的应用逻辑和业务要求，这是一门新的构件化程序设计技术。它不同于传统的结构化程序设计技术，也不同于现在被广泛采用的面向对象程序设计技术。可以说，构件化程序设计位于这二者之上，它更注重于应用系统的全局，要求从应用系统全方位来进行考察；在具体到某个构件或模块的设计时，我们仍然需要结构化程序设计和面向对象程序设计技术作为基础。

我们经常听到“构件软件”和“软件构件”，在这种构件化的应用系统中，所谓构件软件是指按构件模型组合而得到的软件；所谓软件构件是指构成构件软件的每个构件，请读者注意区分这两个概念。

1.2 操作系统的演变

操作系统存在的目的是为了更好地支持应用程序运行。在某种程度上，操作系统所提供的支持决定了应用程序的工作方式。随着因特网时代的到来，应用程序模型已经发生了很大变化。这些变化对操作系统提出了新的要求，也必将引起操作系统的新发展，大致分为以下几代操作系统。

1.2.1 第一代操作系统——DOS 计算模型

因特网时代以前的传统应用软件，大多是静态链接而成，由某一家公司提供，所有功能都集成在同一个软件中，一旦链接之后就不可能替换其中的软件模块。那时的操作系统有两大功能：首先它向用户提供一个分时系统；其次是向用户提供一组函数库。用户程序从主程序起，一步一步驱动软件控制流程，最终完成计算工作。如图 1.2 所示，操作系统始终处于被动地位，为用户程序提供服务。这类系统的典型范例有 DOS 和控制台模式下的 Unix 操作系统，以及目前存在的大多数嵌入式操作系统。

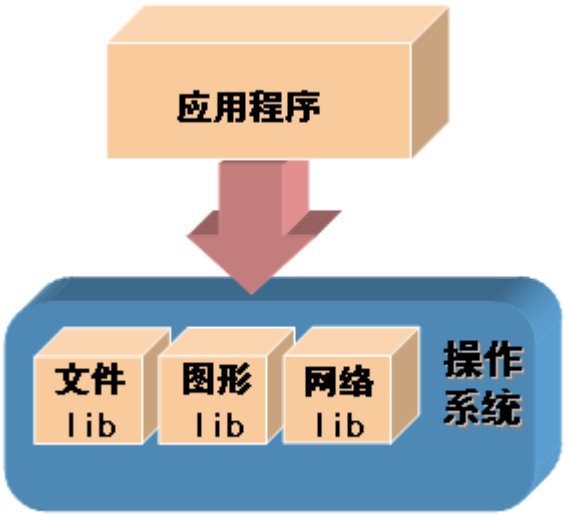


图 1.2　DOS 计算模型

1.2.2 第二代操作系统——Windows 计算模型

基于图形界面的视窗应用软件为了更好地响应键盘，鼠标等外部事件，采用了不同的模式。应用程序从主程序起，一项一项声明所需菜单、选项、窗口等用户界面，注册用户事件（event）的反应函数（callback function），然后应用软件控制流程进入称为消息泵（message pump）的一个死循环，相当于把用户程序控制流交给操作系统。当用户事件发生时操作系统再来调用用户注册的反应函数。如图 1.3 所示，操作系统的地位是先被动后主动，而用户对应用软件，在给定范围内，具有“有限选择”。这类系统叫作基于消息机制的操作系统，比如微软的 Windows 和 Unix 上的 X-Window。

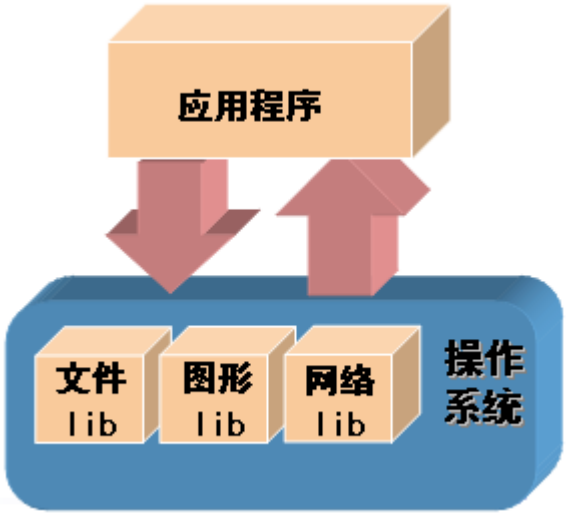


图 1.3　Windows 计算模型

写过视窗程序的人都知道视窗软件的这个消息泵都是千篇一律的，操作系统如果为用户程序“提取公因式”，这段代码理应变成操作系统的一部分。但是这一项编程进化所带来的好处不足以值得改变已有的操作系统模型，因此程序员只好“千古文章一大抄”，将这段死循环抄到所有视窗程序里。其实抄也罢了，只是多写几行程序，浪费一点内存，但是这种消息泵机制还有许多弊病。多数外部事件从硬件驱动程序开始，经过成千上万行代码，被操作系统一直送到那个消息泵死循环里，最后因为用户程序对那些事件根本不感兴趣而不了了之。最典型的是鼠标的移动事件，大多数都是白白浪费 CPU 时间。光浪费时间也罢了，反正现在硬件速度日新月异。但是由于用户可以自定义事件，视窗对象的反应函数就可能要处理无限多种消息，成为一个没有边界的，开放的模型，因此视窗不可能是真正意义上的“面向对象”。

面向对象技术除了需要定义一个数据结构，还需要定义一组作用于这一数据结构的运算，比如 C++语言的虚函数表（v-table），其中函数表的项数只能是有限多个。所以，视窗对象模型的理论基础薄弱，缺乏编程语言的有力支持，其历史局限性日益明显。以视窗对象模型为基础的“面向对象”软件包的典型代表有微软的 MFC（Microsoft Foundation Classes）。MFC 对象永远也不可能“即插即用”（ready to use）。用户程序需要继承并扩展 MFC 对象，其中关键的步骤是要通过“消息图”（message map）来定义向某一视窗对象发送的消息。在给定的应用领域（application domain）里用户程序只向某一视窗对象发送有限多种消息，也就是只做有限多种运算。可是在 MFC 体系中，定义运算个数的工作是

通过特殊的宏定义，在源程序的预处理时才能确定。换句话说，MFC 对象总是半生不熟，必须在程序员加工后才可以使用，因此编程模型非常复杂，不能适应软件工厂化的需求。

1.2.3 第三代操作系统——WEB 服务计算模型

网络操作系统根据不同安全、运营、用户需求，动态控制应用软件，WEB 服务可以在因特网上任何地方运行，随着不同 WEB 服务加载，用户对应用软件，具有“无限选择”，如图 1.4 所示。举一个因特网时代应用程序的例子，它包括了文字、图表、音频、连续图像播放等功能，其中各部分功能的软件模块可以来自不同的软件开发商，它们都是即插即用，在运行时动态组织起来，向用户提供了一个“天衣无缝”的具有复合功能应用软件。这种应用看上去象因特网的浏览器，其实这种构件化（componentization）的软件工程技术早在因特网时代之前就已经初见端倪，比如微软的 OLE 技术在 90 年代初就可以初步支持 Word 和 Excel 应用相互嵌套。不同的是，在因特网时代的今天，构件化技术可以说是无处不在。比如微软的 Windows XP 和 Office XP 的核心都是采用这种构件技术，从视窗桌面到 Word 和 Excel 的窗口全部是“浏览器”，其主要特点是：

(1) 程序作为动态构件自动加载运行，而不需要由用户去逐个启动。

(2) 构件支持脚本语言控制，多个构件可以相互操作，交换信息。

(3) 以浏览器为交互式操作界面，既便于有户掌握，又为程序开发提供了统一标准。

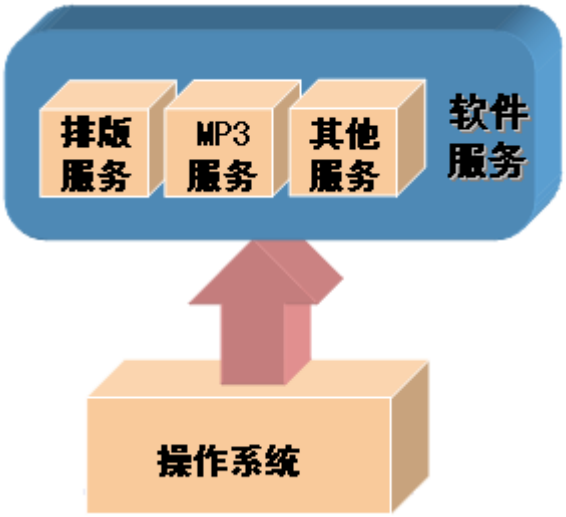


图 1.4 WEB 服务计算模型

在这种新的因特网应用模式下，“浏览器”实际上已经退化为一个可见或不可见的“框”。用操作系统术语讲，这个框是一个可执行文件，或说是一个 EXE 文件；它本身非常简单，不能为用户提供任何应用功能。但是它能为其它构件提供运行环境，而由其它构件提供应用功能。所有构件都被做成一个个动态链接库，或说是 DLL 文件。因为这个小“框”放之四海而皆准，对于任何应用程序都是一样的，操作系统自然可以代劳，在操作系统中做一份就行了。在这种操作系统里启动运行这样一个小小的可执行文件，通常根本不用访问外部存储器（如磁盘），所以效率很高。框里的第一个构件一般需要有一个主函数（main），但是含有主函数的构件并不一定要是第一个加载的构件。这种构件中的主函数与传统操作系统术语中用户程序里的主函数有本质上的区别，这里讲的操作系统模型其实就是 JAVA 虚拟机的编程模型。

既然应用程序员只写构件或动态链接库，而操作系统自始至终控制程序运行的主动权，显而易见，这种操作系统与 DOS 或 Unix 有本质的不同。操作系统可以对应用程序构件进行各种各样的控制，使得封装好的构件能够适应不同的运行环境和用户要求。构件制造商对构件运行环境往往有些特殊的复杂要求，例如构件是否支持多线程或信息加密，这些对于一般用户来讲很难理解和适应；而众多用户的不同好恶，如怀疑构件有病毒，也不是构件制造商料所能及的。这时操作系统控制主动权，动态生成中间件和构造构件运行环境，就能很好地解决这些问题。操作系统利用中间件技术支持和控制应用程序的运行环境，就形成了因特网时代操作系统的关键技术。

1.3 软件编程技术的演变

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

1.3.1 面向对象编程

通过对软件模块的封装，使其相对独立，从而使复杂的问题简单化。面向对象编程强调的是对象的封装，但模块（对象）之间的关系在编译的时候被固定，模块之间的关系是静态的，在程序运行时不可改变模块之间的关系，就是说在运行时不能换用零件。其代表是 C++语言所代表的面向对象编程。

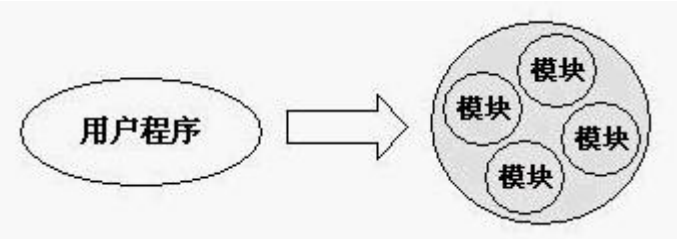


图 1.5　面向对象编程的运行模型，模块之间的关系固定

1.3.2 面向构件编程

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

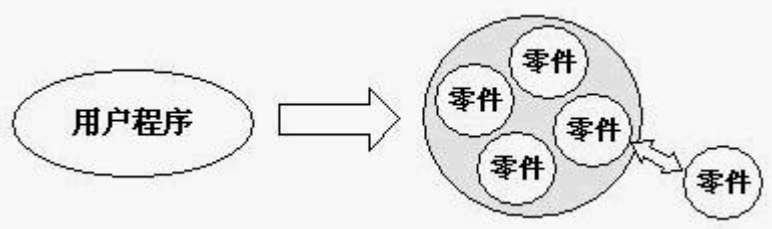


图 1.6　构件化程序的运行模型，运行时零件可替换

1.3.3 面向中间件编程

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

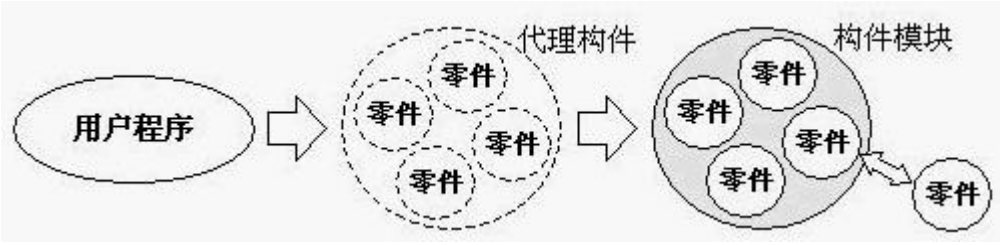


图 1.7　中间件运行环境的模型，动态生成代理构件

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

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

1.4 CAR 技术的产生

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

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

第二章 CAR 技术的发展

2.1 CAR 结构

前面已经提到过，CAR 为构件软件和应用程序之间进行通信提供了统一的标准，它为构件程序提供了一个面向对象的活动环境。

CAR 标准包括规范和实现两大部分，规范部分定义了构件和构件之间（如 CAR 构件、JAVA 构件等之间）通信的机制，这些规范不依赖于任何特定的语言和操作系统，只要按照该规范，任何语言都可使用；CAR 标准的实现部分可能是 Elastos 操作系统，Elastos 操作系统为 CAR 规范的具体实现提供了一些核心服务。

CAR 标准的实现部分也可能是 Elastos 虚拟机 ElaVM。

2.1.1 对象与接口

CAR 是面向对象的软件模型，因而对象是它的基本要素之一，那么 CAR 对象是什么呢？类似于 C++中对象的概念，对象是某个类(class)的一个实例；而类则是一组相关的数据和功能组合在一起的一个定义。使用对象的应用(或另一个对象)称为客户，有时也称为对象的用户。

接口是一组逻辑上相关的函数集合，其函数也被称为接口成员函数。按照习惯，接口名常以“I”为前缀，例如我们下一章专门要讲述的“IObject”。对象通过接口成员函数为客户提供各种形式的服务。

在 CAR 模型中，对象本身对于客户来说是不可见的，客户请求服务时，只能通过接口进行。

每一个接口都由一个 128 位的全局唯一标识符(GUID，Globally Unique Identifier)来标识。在 CAR 的实现中，这个 GUID 已经对用户不可见，也就是用户在写 CAR 文件及写 CAR 构件时，不需要关注 GUID 的存在，它将在 CAR 的编译与运行过程中，由编译工具与运行环境自动生成与管理。

客户通过 Query 机制获得接口的指针，再通过接口指针，客户就可以调用其相应的成员函数。至于具体功能如何实现，则完全由对象的接口内部实现。所以，在 CAR 模型中，对象通过接口及接口中的函数为客户提供服务，对于客户来说，它只与接口打交道。

一般来说，接口是不变的，只要客户期望的接口在构件对象中还存在，它就可以继续使用该接口所提供的服务。对象可以支持多个接口，因此对构件对象的升级可通过增加接口的办法实现，这样得到的新接口可以不影响老接口的使用。新客户可使用新增的接口，老客户可在不更新代码的情况下继续使用老的接口。

客户如何来标识 CAR 对象呢？与接口类似，每个对象也用一个 128 位 GUID 来标识，称为 CLSID(class identifier，类标识符或类 ID)。与 GUID 的情形相似，在 CAR 的实现中，

这个 CLSID 已经对用户不可见，将在 CAR 的编译与运行过程中，由编译工具与运行环境自动生成与管理。用 CLSID 标识对象可以保证(概率意义上)在全球范围内的唯一性。只要系统中含有这类 CAR 对象的信息，并包括 CAR 对象所在的模块文件(DLL 或 EXE 文件)以及 CAR 对象在代码中的入口点(实际的过程要复杂得多)，客户程序就可以由 CISID 来创建 CAR 对象。那么客户怎么使用 CAR 对象提供的服务呢？客户获得的又是什么呢？实际上，客户成功地创建对象后，它得到的是一个指向对象某个接口的指针，因为 CAR 对象至少实现一个接口(没有接口的 CAR 对象是没有意义的)，所以客户就可以调用该接口提供的所有服务。根据 CAR 规范，一个 CAR 对象如果实现了多个接口，则可以从某个接口得到该对象的任意其他接口，所以一旦我们得到了一个接口指针，我们就可以得到其他所有的接口。因此，客户在创建了 CAR 对象得到了某个接口指针后，它就可以调用该对象所有接口提供的服务。从这个过程我们也可以看出，客户与 CAR 对象只通过接口打交道，对象对于客户来说只是一组接口。

但是 CAR 对象可以有其自己的状态，正是这种状态才使客户感觉到 CAR 对象的存在。如果客户同时拥有两个相同（CAR 通过 URL 标识构件，后面会有详细描述）的对象，则两个对象可以有不同的状态，客户完全不必关心 CAR 对象是怎么实现的，以及两个对象的状态数据结构之间有什么关系(数组或者链表)。当然，CAR 对象也可以是无状态的，这种 CAR 对象以提供功能服务为主，可以用来代替传统的 API（Application Programming Interface，应用程序编程接口）函数接口，使得应用程序编程接口更为有序，组织层次性更强。

2.1.2 客户/服务器模型

可以很容易看出，对象和客户之间的相互作用是建立在客户/服务器（Client/Server）模型的基础上的，客户/服务器模型的一个很大的优点是稳定性好，而稳定性正是 CAR 模型的目标，尤其对于跨进程的程序通信，稳定性更会带来高性能和高可靠性。

然而，CAR 不仅仅是一种简单的客户/服务器模型，有时客户也可以反过来提供服务或者服务方本身也需要其他对象的一些功能，在这些情况下，一个对象可能既是服务器也是客户。CAR 能够有效地处理这些情况。

客户/服务器模型是一种发展比较成功的软件模型，因为这种模型有以下一些优势：

(1)稳定性、可靠性好。客户/服务器模型简化了应用，把任务进行分离，客户和服务器各司其职，共同完成任务。

(2)软件的可扩展性更好。一个服务器进程可以为多个客户提供服务，客户也可以连接到不同的服务器上，这种模型的连接非常灵活。

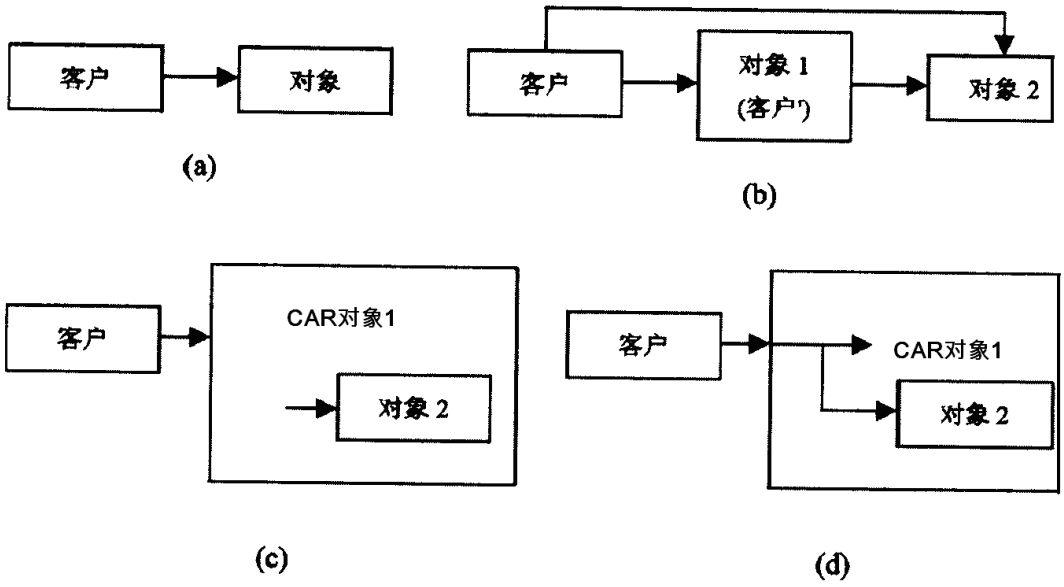
(3)提高性能。根据硬件的配置，可以把繁重的任务放到高配置的一端，用低配置的设备完成一些简单的任务，因此，这种模型使软件运行更加合理。

(4)在网络上实现时，可以降低网络流量。在网络上只传输客户和服务程序所关心的数据。

(5)用于数据库时，可以实现事务(transaction)机制，提供数据备份能力等。

CAR 虽然以客户/服务器模型为基础，但 CAR 可以非常灵活地使用这种模型。图 2.1 中，每一个箭头就代表了一个客户——服务器关系，在图 2.1 (a)中，客户与构件对象只是一个简单的客户 / 服务器模型结构；在图 2.1 (b)中，对象 2 既为客户直接提供服务，也为对象 1 提供服务，这时对象 1 就成了对象 2 的客户，对象 1 为客户提供服务，在这样的模型中，

对象 1 由客户直接创建，而对象 2 既可以由客户创建，也可以由对象 1 创建；图 2.1 (c)和(d)是 CAR 中两种重要的对象重用结构，分别称为包容(containment)和聚合(aggregation)，对于客户来说，只知道对象 1 的存在，并不知道对象 2 的存在，但对象 1 在实现某些服务时，它调用了对象 2 的服务，两者的区别在于，当客户调用由对象 2 提供的服务时，包容模型中，由对象 1 调用对象 2 的服务，再把结果转给客户，所以客户间接地调用对象 2 的服务，而在聚合模型中，虽然客户并不知道对象 2 的存在，但它调用对象 2 的服务是直接进行的。



(a)简单客户/服务器模型；　(b)客户/服务器模型的两重结构
(c)CAR 中包容模型示例；　(d) CAR 中聚合模型示例
图 2.1 CAR 使用客户/服务器模型的几种灵活用法

2.2 CAR 发展简史

CAR 技术在很大程度上借鉴了微软 COM（Component Object Model，构件对象模型）的思想，最初 CAR 是兼容 COM 的，但是和 COM 相比，CAR 删除了 COM 中过时的约定，禁止用户定义 COM 的非自描述接口；完备了构件及其接口的自描述功能，实现了对 COM 的扩展；对 COM 的用户界面进行了简化包装，可以说 CAR 是微软 COM 的一个子集，同时又对微软的 COM 进行了扩展，在 Elastos SDK 工具的支持下，使得高深难懂的构件编程技术很容易被 C/C++程序员理解并掌握，因此最初称之为 ezCOM，其中“ez”源自与英文单词“easy”，恰如其分地反映了这一特点。首先编写一个.cdl 文件，CDL 即构件定义语言，对应于微软的 IDL（接口定义语言），然后将它转换成微软的.idl 文件，最后用 MIDL（微软的 IDL 编译器）进行编译生成相应的代码。

目前的 CAR 技术已经不再保持与 COM 兼容，也不再使用微软的 MIDL 编译器，而是使用自己的工具 carc，lube 和 cppvan。例如，首先编写一个.car 文件，定义构件模块中的构件类、接口以及接口方法等信息，使用 emake 命令，实际上是调用编译器 carc.exe，编译生成代码框架，填写完实现代码后再使用 z 命令编译生成.dll 文件，该文件的资源段中包含了元数据，而 COM 是没有的，这个编译生成代码框架的过程以及如何填写实现代码在后面的

章节中有详细的说明与示例。利用 Elastos IDE 工具将使得用户对这些复杂问题的把握变得容易。

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

2.3 CAR 技术的深远影响

CAR 作为概念完整的构件技术，它提供了一种编程模型，这种编程模型对软件开发有着深远的影响。

2.3.1 CAR 技术对软件工程的作用

CAR的重要特点就是上文所介绍的：构件的相互操作性；软件升级的独立性；编程语言的独立性；进程运行透明度。在实际的编程应用中，CAR技术可以使程序员在以下几个方面得到受益：

1、 易学易用

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

2、 可以动态加载构件

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

3、 采用第三方软件丰富系统功能

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

展系统的功能。

4、 软件复用

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

5、 系统升级

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

6、 实现软件工厂化生产

上述几个特点，都是软件零件工厂化生产的必要条件。构件化软件设计思想规范了工程化、工厂化的软件设计方法，提供了明晰可靠的软件接口标准，使软件构件可以像工业零件一样生产制造，零件可用于各种不同的设备上。

7、 提高系统的可靠性、容错性

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

8、 有效地实现系统安全性

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

2.3.2 CAR 技术的意义

对于软件开发企业而言，采用CAR构件技术具有以下的意义：

- CAR的开发工具自动实现构件的封装，简化了构件编程的复杂性，有利于构件化编程技术的推广普及；
- CAR构件技术是一个实现软件工厂化生产的先进技术，可以大大提升企业的软件开发技术水平，提高软件生产效率和软件产品质量；
- 软件工厂化生产需要有零件的标准，CAR构件技术为建立软件标准提供了参考，有利于建立企业内、行业内的软件标准，有利于建立企业内、行业内的构件库。

第三章 CAR 的基本知识

3.1 CAR 基本定义

CAR，即 Component Assembly Runtime。Component，有“零件”的意思；Assembly，有“部件”、“组装”的意思，所以从字面上理解，CAR 就是在运行时对软件构件进行组装并最终完成预计功能的一种软件技术。

机械行业有“零部件”的说法，零件和部件都是工厂生产线上的安装单元。零件是最基本的安装单元，部件是由零件组成的半成品安装单元。最终产品一般是由某种部件加上外壳而成。

在 Elastos 中，Component Assembly 也包含了这两层含义：（1）软件零件，特指“目标代码单元”。在 CAR 编程规范中就是 DLL，也可以是 JAVA 或 C#中的目标代码文件；（2）软件部件，是软件零件的集合。一般是个“半成品”，通过 XML 或角本语言包装成为“产品”，也可以是个“产品”。软件部件不但包含一组 DLLs（也可以是单个 DLL），还包含了装箱单、数字签名、下载压缩包、元数据信息等打包之后的信息，类似于 JAVA 里面的 JAR 文件、Windows 里面的 CAB 文件等。

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

3.2 CAR 构件技术

CAR构件技术是科泰世纪开发的具有自主知识产权的构件技术，它是一个面向构件的编程模型，也可以说是一种编程思想，它表现为一组编程规范，规定了构件间相互调用的标准，包括构件、类、对象、接口等定义与访问构件对象的规定，使得二进制构件能够自描述，能够在运行时动态链接。

CAR 技术简化了构件的开发过程，编写 CAR 文件后用 CAR 编译器编译便可生成基本的代码框架，开发人员在此基础上开发出自己的构件为客户端提供服务，提高了构件开发的速度及质量。CAR 的编程思想是 Elastos 技术的精髓，它贯穿于整个技术体系的实现中。

“Elastos 构件运行平台”提供了一套符合 CAR 规范的系统服务构件及支持构件相关编程的 API 函数，实现并支持系统构件及用户构件相互调用的机制，为 CAR 构件提供了编程运行环境。Elastos 构件运行平台在不同操作系统上有不同的实现，符合 CAR 编程规范的应用程序通过该平台实现二进制级跨操作系统平台兼容。在 Windows 2000、WinCE、Linux 等其他操作系统上，Elastos 构件运行平台屏蔽了底层传统操作系统的具体特征，实现了一个构件化的虚拟操作系统。在 Elastos 构件运行平台上开发的应用程序，可以不经修改、不损失太多效率、以相同的二进制代码形式，运行于传统操作系统之上。

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

3.3 什么是接口

3.3.1 接口的定义

接口（interface）是用来定义一种程序的协定。实现接口的类或者结构要与接口的定义严格一致。有了这个协定，就可以抛开编程语言的限制（理论上）。接口可以从多个基接口继承，而类或结构可以实现多个接口。接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或接口必须提供的成员。

接口好比一种模版，这种模版定义了对象必须实现的方法，其目的就是让这些方法可以作为接口实例被引用。接口不能被实例化。类可以实现多个接口并且通过这些实现的接口被索引。接口变量只能索引实现该接口的类的实例。例子：

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

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

   //类 CHello
    class CHello {
        interface IHello;
        interface IHey;
    }
}
```

上面例子中的接口包含各自的接口方法。

类和结构可以多重实例化接口，就是一个接口定义可以被多个类定义所引用。

说明：

• CAR 中的接口是独立于类来定义的。这与 C++模型是对立的，在 C++中接口实际上就是抽象基类。

• 类可以继承多个接口。

• 而类可以继承一个基类，接口根本不能继承类。这种模型避免了 C++的多继承问题，C++中不同基类中的实现可能出现冲突。因此也不再需要诸如虚拟继承和显式作用域这类复杂机制。C#的简化接口模型有助于加快应用程序的开发。

• 一个接口定义一个只有抽象成员的引用类型。CAR 中一个接口实际所做的，仅仅只存在着方法标志，但根本就没有执行代码。这就暗示了不能实例化一个接口，只能实例化一个派生自该接口的对象。

3.3.2 接口与构件

接口描述了构件对外提供的服务。在构件和构件之间、构件和客户之间都通过接口进行交互。因此构件一旦发布，它只能通过预先定义的接口来提供合理的、一致的服务。这种接口定义之间的稳定性使客户应用开发者能够构造出坚固的应用。一个构件可以实现多个构件接口，而一个特定的构件接口也可以被多个构件来实现。

构件接口必须是能够自我描述的。这意味着构件接口应该不依赖于具体的实现，将实现和接口分离彻底消除了接口的使用者和接口的实现者之间的耦合关系，增强了信息的封装程度。同时这也要求构件接口必须使用一种与构件实现无关的语言。构件接口的描述标准是 CAR 语言。

由于接口是构件之间的协议，因此构件的接口一旦被发布，构件生产者就应该尽可能地保持接口不变，任何对接口语法或语义上的改变，都有可能造成现有构件与客户之间的联系遭到破坏。

每个构件都是自主的，有其独特的功能，只能通过接口与外界通信。当一个构件需要提供新的服务时，可以通过增加新的接口来实现。不会影响原接口已存在的客户。而新的客户可以重新选择新的接口来获得服务。

3.3.3 构件化程序设计

构件化程序设计方法继承并发展了面向对象的程序设计方法。它把对象技术应用于系统设计，对面向对象的程序设计的实现过程作了进一步的抽象。我们可以把构件化程序设计方法用作构造系统的体系结构层次的方法，并且可以使用面向对象的方法很方便地实现构件。

构件化程序设计强调真正的软件可重用性和高度的互操作性。它侧重于构件的产生和装配，这两方面一起构成了构件化程序设计的核心。构件的产生过程不仅仅是应用系统的需求，构件市场本身也推动了构件的发展，促进了软件厂商的交流与合作。构件的装配使得软件产品可以采用类似于搭积木的方法快速地建立起来，不仅可以缩短软件产品的开发周期，同时也提高了系统的稳定性和可靠性。

构件程序设计的方法有以下几个方面的特点：

- 编程语言和开发环境的独立性.
- 构件位置的透明性.
- 构件的进程透明性.
- 可扩充性.
- 可重用性.
- 具有强有力的基础设施.
- 系统一级的公共服务.

3.4 CAR 构件技术在 Elastos 中的作用

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

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

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

3.5 CAR 的技术内涵

1、CAR 是一种基于构件的软件运行支持技术。构件运行支持能力直接决定所支持的构件的编写方法，结构设计，甚至算法选择。CAR 支持满足“故障独立性”（即某个部件失效不会引起其它部件的失效，是硬件系统可靠性的基本特性）的运行环境，如过程、Domain 等。CAR 通过这种环境所提供的构件动态组装，对外完成预计的计算任务。

2、CAR 是一种构件化的开发语言，它只负责框架部分，具体的实现逻辑由 C/C++等编程语言实现。CAR 所描述的框架部分以元数据的形式存在于构件的发行格式中，元数据通过反射（reflection）机制参与构件组装计算。框架是将具体的应用逻辑通过类似于 COM 的方式（计数管理、接口查询、构件聚合）隐藏起来，并把自己暴露在外的最终运行封装。

3、CAR 支持构件被分布式配置在不同计算容器（进程、Domain、机器）中，从而实现分布式、协同计算。CAR 定义了构件在各种情况下的通信方式，故障处理方式、安全机制等。

4、CAR 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“零件”，从而提高软件生产效率。

5、CAR 定义了一种软件工程化方法，软件发行与配置策略，从而定义了面向服务的软件商业模型。

6、CAR 支持灵活软件架构策略，通过类硬件的构件技术，方便各种软件架构下对构件的使用。

3.6 CAR 的技术特性

3.6.1 构件自描述

构件自描述简单来说是构件能够描述自己的数据信息，它通过元数据的方式来实现。

元数据(metadata)，是描述数据的数据(data about data)，元数据是一种数据,是对数据的抽象，它主要描述了数据的类型信息。CAR 把模块信息（ModuleInfo）,接口信息（InterfaceInfo），类信息（ClassInfo）等作为描述构件的元数据。这些信息由 CAR 文件编译而来，是 CAR 文件的二进制表述。

在 CAR 中，可以使用一个特殊的 CLSID 从构件中取出元数据信息，构件元数据的解释不依赖于其它的 DLL 文件。在 CAR 文件的编译过程中生成的文件××pub.cpp 文件中生成 _CarDllGetClassObject 函数获得相关的接口信息.

在 CAR 的构件封装中，除了构件本身的类信息封装在构件内外，还对构件的依赖关系进行了封装。即把一个构件对其它构件的依赖关系也作为构件的元数据封装在构件中，我们把这种元数据称为构件的导入信息(ImportInfo)。CAR 构件通过对 ClassInfo 和 ImportInfo 的封装，可以实现构件的无注册运行。并可以支持构件的动态升级和自滚动运行。

CAR 构件库定义了一套访问元数据的接口,可以通过映射函数(reflection)获取元数据。

3.6.2 可重用性

与 C++类在源代码级别的重用不同，CAR 构件的重用是建立在对二进制代码重用的基础上的。具体包括包容（containment）、聚合（aggregation）以及二进制继承三种重用模型，其本质也都是在一个构件中对另外一个构件的使用。

3.6.3 面向方面的编程支持

AOP 是一个基于构件技术的面向方面的软件开发模型。在目前的面向对象的编程模式中，仅仅用类的思想来分析和实现软件系统，不能有效地表示软件系统的关注点。AOP 将“关注点”封装在“方面”中，将这些操作与业务逻辑分离，使程序员在编写程序时可以专注于业务逻辑的处理，而利用 AOP 将贯穿于各个模块间的横切关注点自动耦合进来。由此能够改善系统逻辑、减低软件开发难度、提高软件开发质量和提高软件重用性。

基于CAR的AOP机制使用户能够在完全不用修改源代码的情况下简单而方便的动态聚合两个或多个CAR构件类,从而生成一个具有两个或多个CAR构件类所有接口实现的新构件类。比如一个或多个功能构件与一个方面构件聚合,可以使不同的功能构件在某一方面上具有相同的行为和属性。CAR 的 AOP 技术由方面（Aspect）、动态聚合（Dynamic Aggregation）和上下文环境 (context)组成。动态聚合是实现 Aspect 对象的必要条件，Aspect 对象是上下文环境实现的基础。

3.6.4 远程过程调用

当客户端和服务器端所在地址空间不同时，客户端的进程要调用服务器端的构件的服务，属于远程构件调用。CAR 构件技术支持远程接口调用，通过调用数据的列集\散集技术

进行不同地址空间的数据交互。构件服务和构件服务调用者可以处于操作系统的不同空间，而调用者可以如同在同一地址空间里面使用构件一样的透明的进行远程接口调用，也就是说完全向用户屏蔽了底层使用的标准的列集\散集过程。

3.6.5 命名服务机制

命名服务机制的是将一个构件和指定的字符串绑定的过程，构件使用者可以远程通过字符串查询该构件，并获得构件服务。命名服务本身即可以作为一个单独的构件存在，亦可以作为内核功能的一部分。

3.6.6 回调事件机制

当特定事情发生时，如定时消息或用户鼠标操作发生时，构件对象产生一个事件，客户程序可以处理这些事件。构件对象中回调接口并不由构件对象实现，而是由客户端的接收器实现。接收器也是构件对象，它除了实现回调接口外，还负责与可连接对象进行通信。当接收器与可连接对象建立连接后，客户程序可将自己实现的事件处理函数注册，把函数指针告诉构件对象，构件对象在条件成熟时激发事件，回调事件处理函数。

3.6.7 构件缓存机制

为了提高访问网页的速度，IE 浏览器会采用累积式加速的方法，将你曾经访问的网页内容（包括图片以及 cookie 文件等）存放在电脑里。这个存放空间，就是 IE 缓存。以后每次访问网站时，IE 会首先搜索这个目录，如其中有已访问过的内容，那 IE 就不必从网上下载，而直接从缓存中调出来。

CAR 构件缓存机制便是在 IE 缓存机制的思想基础上建立的。CCM（CAR Cache Manager，CAR 高速缓存管理）是一套构件化的缓存管理机制，它主要是为了支持 Elastos 网络操作系统的构件自滚动运行。

3.6.8 构件版本控制

版本可以用来指出在同时发行的软件包或构件库中的某一构件，或是定义构件在一系列版本中的版次。当构件库被多次制作时，各构件库之间可能会有或大或小的改变，识别出构件的版本将可以和其它类似构件进行区别，其它类似构件包括之前或之后以及属于不同版本的特定构件。在 Elastos 系统中，存在一个 CAR 构件的版本管理器，这个版本管理器自身也是一个 CAR 构件，它与通常意义下的 CAR 构件的区别在于：此构件的功能是专门用来管理其它 CAR 构件版本的。

3.6.9 点击运行机制

点击运行是指在网络环境之下，软件无需事先在本地的计算机上安装，而是在需要时通过点击网络中的相关主题或对象（如按钮、图片、滚动条等）来实现动态加载构件，从而为相关应用提供合适的服务。点击运行的核心思想在于系统对应用服务支持的高度自动化。

3.6.10 CAR Web 服务

CAR 网络服务技术是一种构建于 CAR 构件运行平台之上的将运行的构件实例发布为网络服务的方法。任何的 CAR 构件实例，都可以通过 CAR 网络服务运行环境发布到互联网，通过 WWW 标准网络服务协议（SOAP、WSDL、UDDI 等）被远程访问。目前实现的 CAR 网络服务运行环境是一个内建了 CAR 构件网络服务代理功能的简单 HTTP 服务器，采用构件化的体系结构，支持的协议包括 HTTP 1.1、SOAP 1.1 和 WSDL 1.0。其中 CAR 构件网络服务代理的主要功能包括解释、转发客户端的 SOAP 远程调用并返回执行结果，为运行的构件实例自动生成对应的 WSDL 描述并作为 HTTP GET 请求的结果返回等。经过验证，目前实现的 CAR 网络服务运行环境发布的网络服务可以通过 Microsoft .NET、Microsoft SOAP Toolkit、gSOAP 等第三方工具访问。

这些技术特性共同构成了 CAR 技术的精髓，更为详细的介绍见本文第二篇的内容。

3.7 CAR 技术应用范围

1、支持 Web Service 的嵌入式设备。Web Service 的提供一定是基于构件的，而 CAR 构件技术正适应了这种需要。

2、实时性要求不高的控制设备。CAR 通过大量的接口技术实现构件的动态组装，保证了系统尺寸具有较大的弹性。但构件的运行调度会在一定程度上影响实时性，因此可适用于对实时性要求不高的控制设备。

3、软件可靠性要求特别高的场合。CAR 构件具有动态适配能力，通过运行环境中高可靠的调度器进行调度，可以实施诸如冗余计算、3 中取 2 等多种提高系统可靠性

3.8 几个重要的 CAR 关键字

为了使读者对 CAR 技术有一个初步的了解，我们先从几个最能体现 CAR 思想的关键字入手，在后面的章节中会有更为详细的介绍及具体的示例。

在前一节 CAR 的技术特性中，我们提到了回调事件机制，这个特性是由 CAR 的一个重要的关键字 callback 来实现的，我们用 callback 来定义回调接口。回调接口也是接口的一种，该接口中的每个成员函数代表一个回调事件（callback）。当特定事情发生时，如定时消息或用户鼠标操作发生时，构件对象产生一个事件，客户程序可以处理这些事件。构件对象中回调接口并不由构件对象实现，而是由客户端的接收器实现。接收器也是构件对象，它除了

实现回调接口外，还负责与可连接对象进行通信。当接收器与可连接对象建立连接后，客户程序可将自己实现的事件处理函数注册，把函数指针告诉构件对象，构件对象在条件成熟时激发事件，回调事件处理函数。

CAR 的另外一个很重要的思想就是面向方面编程(AOP)，CAR 的 AOP 技术便是由 aspect，aggregate 和 context 这几个关键字所体现的。其中方面(aspect)是一种特殊的构件类实现，关键字 aspect 即用来定义方面构件类，aspect 对象的特征是可以被其它构件对象聚合，该构件类必须实现 IAspect 接口，aspect 对象就是实现了 IAspect 接口的构件对象。CAR 构件的每个接口都是由 IObject 继承而来的，动态聚合是通过 IObject 的 Aggregate 方法来完成的，因此构件编写者定义的每个构件对象都具有聚合其他 aspect 对象的能力。一般实现动态聚合都通过 CAR 构件运行环境中提供的 EzAggregate 方法，而不是显式的去调用 Aggregate。

在实际软件开发中，面向方面编程时，往往需要一个对象聚合多个 aspect 对象，这就是多面聚合。它的实现与单一聚合并没有多大的差别，只要创建多个 aspect 对象，多次调用 EzAggregate 方法就可以使一个对象聚合多个 aspect 对象。语境是对象运行时的环境，一个对象如果进入了语境，那么该对象将具有此语境的特征，一旦对象离开了语境，环境特征就会失去（但该对象很有可能又进入了另外一个语境， 拥有新的环境特征）。在 CAR 中，语境也是一个构件类，它由 context 关键字来定义，它所具有的语境特征由其属性 aspect 来决定。

第四章　CAR 文件结构

4.1 CAR 文件

Elastos 中 CAR（Component Assembly Runtime）的含义是“基于 CPU 指令集的软件零部件运行单元”，简译为“零部件运行单元”。

编写 CAR 文件后运用自动代码生成工具便可生成基本的代码框架，开发人员在此基础上进行开发出自己的构件为客户端提供服务。

4.2 CAR 文件的基本构成

CAR 文件用于定义构件模块中的构件类、接口以及接口方法等信息。CAR 文件主要由构件类、接口、接口方法的定义以及修饰构件类、接口以及接口方法的属性和关键字所构成。

构件类事例是最基本的构件运行实体，一个构件模块可以封装一到多个构件类的实现。构件类的实例是构件对象，构件对象是接口的实现，一个构件对象可以实现多个接口，一个接口可以被多个构件对象实现。

构件类主要有以下三种：普通构件类，方面构件类和上下文构件类。

接口是一组逻辑上相关的函数集合，是构件特征的抽象定义，是最基本的构件使用单位。

在 CAR 文件中，对构件、类、接口和方法等的描述都分为两个部分：属性和定义。CAR 属性位于方括号“[]”中，多个属性之间以逗号“,”作为分隔符；对构件、接口、类等的定义位于花括号“{}”中。

属性总是出现在相应主题（指构件、类、接口或方法）的定义之前，并且只对随后的一个主题有效。

每个 CAR 构件都可以由一个或多个类（包括普通构件类，方面构件类和上下文构件类）组成，每个类可以提供一个或多个接口，每个接口中可以定义一个或多个方法。例如：图 4.1 为构件 HelloDemo 的示意图：

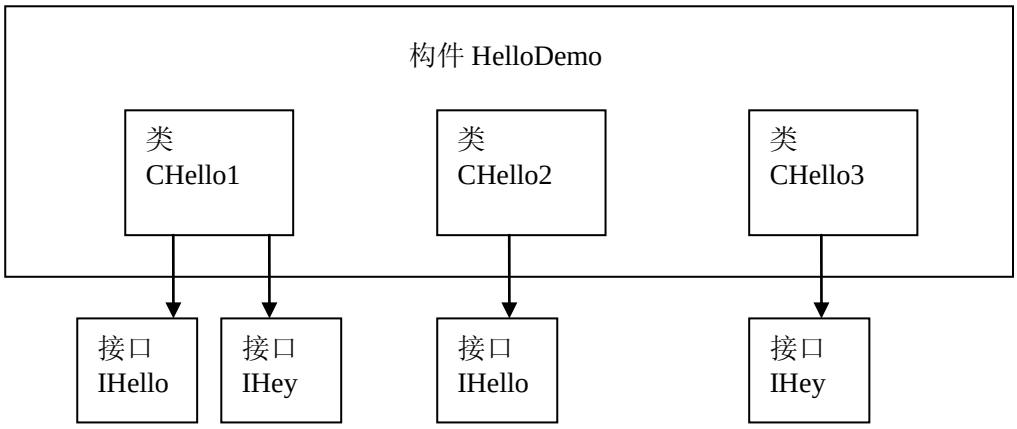


图 4.1 构件 HelloDemo 示意图

从图 4.1 中可以看出：构件 HelloDemo 中定义了三个类：CHello1、CHello2 和 CHello3。其中，类 CHello2 提供了接口 IHello，类 CHello3 提供了接口 IHey，而类 CHello1 同时提供了接口 IHello 和 IHey。这些信息都定义在 HelloDemo.car 文件中。

HelloDemo.car 文件：

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

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

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

    //类 CHello2
    aspect AHello2 {
        interface IHello;
    }

    //类 CHello3
    class CHello3 aggregates AHello2{
        interface IHey;
    }
}
```

CAR 文件中所有的方法其函数返回类型均默认为 ECode， 所以不需要说明方法的返回值类型，否则编译会报错。

第五章　CAR 数据类型

5.1 CAR 支持的数据类型

文法中规定的 CAR 数据类型用于定义接口方法中的参数。下表列出 CAR 支持的数据类型以及在 C++中的映射：

CAR	C++数据类型	数据类型描述
AChar	char	窄字符类型
WChar	wchar_t	宽字符类型
Char32	unsigned int	字符类型
Byte	unsigned char	8 位无符号整数
Flags8	unsigned char	8 位标志类型
Flags16	unsigned short	16 位标志类型
Flags32	unsigned int	32 位标志类型
Flags64	__uint64	64 位标志类型
Int8	signed char	8 位有符号整数
Int16	signed short	16 位有符号整数
Int32	int	32 位有符号整数
Int64	__int64	64 位有符号整数
Float	float	32 位 IEEE 浮点数
Double	double	64 位 IEEE 浮点数
Boolean	unsigned char	8 位整数,值为 0 时表 false,值为 1 时表示 true
struct	struct	结构体
enum	enum	枚举类型
Seconds32	Int32	时间单位秒的 32 位整数
Seconds64	Int64	时间单位秒的 64 位整数
Millisecond32	Int32	时间单位毫秒的 32 位整数
Milliseconds64	Int64	时间单位毫秒的 64 位整数
Microseconds32	Int32	时间单位微秒的 32 位整数
Microseconds64	Int64	时间单位微秒的 64 位整数
Timestamp64		时间标记类
Handle32	unsigned int	32 位无符号整数
Handle64	__uint64	64 位无符号整数
Astring		指向一个常量窄字符串（8 位字符　　串）的指针
WString		指向一个常量宽字符串（16 位字符串）的指针
AStringBuf		存储用户窄字符串的缓冲区数据结构
WStringBuf		存储用户宽字符串的缓冲区

		数据结构
ArrayOf		一种具有自描述功能的 T 类型数组
BufferOf		一种具有自描述功能的 T 类型数组,操作对象是 T 类型的数据块
MemoryBuf		存储用户数据块的缓冲区数据结构
ECode		32 位整数，标准的返回类型
IInterface		接口指针,包括系统接口和自定义接口
EGuid		全球唯一标识符
EIID		接口标示符,用于唯一标示一个特定的 CAR 接口
ClassID		类标示符,用于唯一标示一个 CAR 类
EMuid		Modul 唯一 ID

注释：对应的 C++数据类型是指在 Elastos 开发环境中支持的特定的 C++数据类型

如果 CAR 基本数据类型是有对应的 C++直接映射数据类型，我们采用了 typedef 的方法来自定义了对应的类型，使它在 C++中的名字和在 CAR 中的一致。但是，并不是所有的 CAR 基本数据类型在 C++中都有直接的映射，对于这种情况，我们自定义结构体或类来进行封装。所以，在设计接口，定义参数时，应当选择适当的数据类型。如果构件接口是被 C/C++语言所使用，在定义方法参数时，上述数据类型都可以使用；如果考虑到构件接口是被 Visual Basic 或 Java 等其它脚本语言所使用，在定义方法参数时，应该选择该语言所能处理的数据类型作为参数类型。需要注意的是，如果接口方法参数类型使用非 CAR 支持的基本数据类型，则客户程序与构件程序无法进行远程通信。

5.2 CAR 自描述数据

5.2.1 自描述数据类型的必要性

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

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

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

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

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

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

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

5.2.2 自描述数据类型

通过上一节简单的例子我们可以看到，为传递一个非自描述的数据类型参数，可能需要多个额外的参数加以补充说明。那么什么是自描述的数据类型呢？简单的说，所谓自描述数据类型是指这样的一些数据类型，该数据类型自身所带有的数据信息已经足够描述其自身的特征，比如占用内存的情况、它的基本属性及其它的相关信息等，也就是说不需要其他附加条件也能够实现自我描述的数据类型。

通过该定义，我们基本上可以为传统的数据类型进行分类，如 double、float 等兼容 IEEE 实数标准的数据类型就是属于自描述的数据类型。假设服务端获得一个 double 的参数传递，那就能够确定：①现在得到的是一个占 8 个字节的连续内存区域；②共 64 位；③其中第一位是符号位，11 位是指数位，52 位是尾数位；④它的范围为+/-1.7E308。这些信息是很明确的，也足够描述该数据类型的特征。再比如说如果传递的是 char *指针类型的参数，那么我们可以知道这是一个 32 位的指针，它指向一个以字节为单位的连续的字符缓存空间，该连续空间以‘\0’表示结束。那么我们可以得到该连续字符空间的起始地址及结束地址，也就可以得到该字符串的长度，因此，我们也说 char *是属于自描述的数据类型。如果是 byte *或 void */PVOID 的数据类型呢？可以肯定的是，这些不属于自描述的数据类型，因为他们自身所携带的信息不足以描述他们的自身，这在上例中已有所说明。

非指针型基础数据类型基本上属于自描述类型，除了字符指针外，其他基础数据类型指针型基本上不属于自描述数据类型。

另外，C/C++除了这些基础数据类型外，同时还支持用户自定义数据类型，例如：

```
typedef  class CStudent  CStudent, *pStudent;

class CStudent {
    Byte *pData;
public:
    Int32  age;
    Char  *pClassName;
```

就这个例子而言，CStudent 及 pStudent 都不属于自描述数据类型，其成员 pData 不具有描述自身的特性。如果我们略加修改为：

```
typedef  class CStudent  CStudent, *pStudent;

class CStudent {
    Int32  dataLen;
    Byte *pData;
public:
    Int32  age;
    Char  *pClassName;
}
```

其中新增的成员变量 dataLen 将用于记录 pData 的大小。那么从某种程度上的应用来说这就基本符合了自描述数据类型的要求（但这不能够作为操作系统的自描述基本数据类型，因为这毕竟是用户自定义的，操作系统无从得知用户的约定）。从这里我们可以看出，就应用而言，自描述数据类型是相对于需求的。在实际应用开发中，我们要依据需求通过最简练的设计包容最有效的信息，当然无需刻意追求一种自描述的效果而矫枉过正，因为实现自描述需要额外的系统存储资源。

5.2.3 自描述数据类型在 CAR 构件开发中的重要性

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

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

但在网络技术迅猛发展的今天，“客户/中间件/服务器”三层乃至所谓的多层体系结构、中间件技术、Grid 网络计算等新概念新技术层出不穷，传统的操作系统已不能很好的适应 WEB 服务的要求，而基于构件技术的 Elastos 正是为适应这种新形式而研发出来的新一代操作系统。我们知道，在中间件的应用开发中，构件接口参数的列集（Marshaling）和散集（UnMarshaling）起着关键性的作用，除了整型和布尔型这类的简单类型能被顺利处理外，其他部分的复杂类型则将消耗系统的很大一部分资源用于处理传递参数的列集和散集。而定义出一套基础自描述数据类型将使我们在以下方面获利：

可以通过有限的参数传递，得到理想的数据信息；

能有效地降低服务构件的负载，并能快速响应客户构件的应用请求；

能有效地减少数据的二义性，避免发生人为的不必要的计算错误；

在 CAR 技术中，满足构件兼容性的要求。

5.3 CAR 常用数据结构的详细介绍

5.3.1 CarQuintet 五元组

五元组的结构体定义如下：

```
typedef struct CarQuintet {
        CarQuintetFlags m_flags;  //数组元素的类型标志
        CarQuintetLocks m_reserve;  //暂未使用
        MemorySize      m_used;   //已使用的数据区大小
        MemorySize      m_size;   //数据区大小
        PVoid           m_pBuf;   //指向数据区的指针
    }CarQuintet, *PCarQuintet, *PCARQUINTET;
```

其中，枚举类型 CarQuintetFlag 用来指明数组元素的类型，定义如下：

```
    typedef enum _CarQuintetFlag
    {
        CarQuintetFlag_HeapAlloced      = 0x00010000,
        CarQuintetFlag_Type_Unknown     = 0,
        CarQuintetFlag_Type_Int8        = 1,
        CarQuintetFlag_Type_Int16       = 2,
        CarQuintetFlag_Type_Int32       = 3,
        CarQuintetFlag_Type_Int64       = 4,
        CarQuintetFlag_Type_Byte        = 5,
        CarQuintetFlag_Type_UInt8       = CarQuintetFlag_Type_Byte,
        CarQuintetFlag_Type_UInt16      = 6,
        CarQuintetFlag_Type_UInt32      = 7,
        CarQuintetFlag_Type_UInt64      = 8,
        CarQuintetFlag_Type_Boolean     = CarQuintetFlag_Type_Byte,
        CarQuintetFlag_Type_Float       = 9,
        CarQuintetFlag_Type_Double      = 10,
        CarQuintetFlag_Type_AChar       = 11,
        CarQuintetFlag_Type_WChar       = CarQuintetFlag_Type_UInt16,
        CarQuintetFlag_Type_Char32      = CarQuintetFlag_Type_UInt32,
        CarQuintetFlag_Type_AString     = 12,
        CarQuintetFlag_Type_WString     = 13,
        CarQuintetFlag_Type_EMuid       = 14,
        CarQuintetFlag_Type_EGuid       = 15,
        CarQuintetFlag_Type_ECode       = CarQuintetFlag_Type_Int32,
        CarQuintetFlag_Type_Enum        = CarQuintetFlag_Type_Int32,
```

```
        CarQuintetFlag_Type_Struct      = 16,
        CarQuintetFlag_Type_IObject     = 17,
        CarQuintetFlag_Type_Point       = 18,
        CarQuintetFlag_TypeMask         = 0x0000ffff
    } CarQuintetFlag;
```

5.3.2 BufferOf

BufferOf 类型定义：

```
template <class T>
class BufferOf : public CarQuintet
{
    public:
        // member functions declarations or definitions
     private:
        //constructors or some operator
}
```

构造函数

BufferOf(const T *pBuf, Int32 capacity)

在栈中构造一个 BufferOf 对象，设置数组长度，并用源数组中的数据初始化。

```
 BufferOf (
              const T    *pBuf;      //   T 类型数组
              Int32 capacity        //   待创建的 BufferOf 数组长度
             )
{
            _CarQuintet_Init(this, pBuf, capacity * sizeof(T),
                                    capacity * sizeof(T), Type2Flag<T>::Flag());
}
```

BufferOf(const T *pBuf, Int32 capacity， Int32 used)

在栈中构造一个 BufferOf 对象,并分别设置数组长度和已使用长度,并用源数组中数据初始化。

```
 BufferOf (
                const T    *pBuf;       //   T 类型数组
                Int32 capacity;         //   待创建的 BufferOf 数组长度
                Int32 used              //   待创建的 BufferOf 数组已使用长度
            )
{
             assert(used <= capacity);
             _CarQuintet_Init(this, pBuf, capacity * sizeof(T),
                                        used * sizeof(T), Type2Flag<T>::Flag());
}
```

方法

Alloc()

在堆上动态创建 BufferOf 对象，并指定数组元素个数。

```
Static BufferOf<T> *Alloc(
                Int32 capacity       // 待创建的 BufferOf 数组长度
                    )
{
     return(BufferOf<T>*)_BufferOf_Alloc(
                                        capacity * sizeof(T), Type2Flag<T>::Flag()     );
}
```

Alloc()

在堆上动态创建 BufferOf 对象,并分别设置 BufferOf 数组长度，并用 T 类型数组初始化数组。

```
Static BufferOf<T> *Alloc(
                T *pBuf，          //   T 类型数组
                Int32 capacity     //   待创建的 BufferOf 数组长度
                    )
{
    return(BufferOf<T>*)_BufferOf_Alloc_Box(   pBuf,capacity * sizeof(T),
                                        capacity *sizeof(T),Type2Flag<T>::Flag()     );
}
```

Alloc()

在堆上动态创建 BufferOf 对象，并分别指定 BufferOf 数据区大小为 capacity 及已使用长度为 used，并用 T 类型数组初始化数组。

```
Static BufferOf<T> *Alloc(
                T *pBuf，          //   T 类型数组
                Int32 capacity,   //   待创建的 BufferOf 数组长度
                Int32 used        //  待创建的 BufferOf 数组已使用长度
                    )
{
    Assert(used<=capacity);
    return(BufferOf<T>*)_BufferOf_Alloc_Box(   pBuf,capacity * sizeof(T),
                used*sizeof(T),Type2Flag<T>::Flag()     );
}
```

Append()

在 BufferOf 对象的数据区的已使用空间末尾添加数据 vaule。

```
 BufferOf & Append(
                T   vaule     //   待添加的数据
                    )
```

```
{
        _BufferOf_Append(this, (PByte)&vaule, sizeof(T));
        return *this;
}
```

Append()

在 BufferOf 数组中追加另一 BufferOf 数组中数据。

```
 BufferOf & Append(
                Const BufferOf<T> *pSrc     //   待添加的 BufferOf 对象
                    )
{
        _BufferOf_Append(this, (PByte)pSrc->m_pBuf, pSrc->GetUsed()sizeof(T));
        return *this;
}
```

Append()

在 BufferOf 数组中追加另一数组中的 n 个数据。

```
 BufferOf & Append(
                Const T *pBuf ,     //   T 类型数组
                Int32       n          // 待添加的数组元素个数
                    )
{
        _BufferOf_Append(this, (PByte)pBuf, n*sizeof(T));
        return *this;
}
```

Clone ()

获取一个 BufferOf 对象的深拷贝，即在堆上复制 BufferOf 数组。

```
BufferOf<T> *Clone() const
{
```

```
    return (BufferOf<T> *)_CarQuintet_Clone((const PCarQuintet)this);
}
```

Copy()

将源 BufferOf 数组中数据复制到当前 BufferOf 数组中。

```
BufferOf & Copy(
        const BufferOf<T> *pSrc     //源 BufferOf 对象
        )
{
    _BufferOf_Copy(this,pSrc);
    return *this;
}
```

Copy()

将 T 类型数组数组复制到当前 BufferOf 数组中，并指定拷贝元素个数。

```
BufferOf & Copy(
        const T *pBuf,     //     T 类型数组
          Int32 n          //     待复制的数组元素个数
       )
{
    _BufferOf_CopyEx(this,(Byte*)pBuf,n*sizeof(T));
    return *this;
}
```

Free()

删除由 Alloc 或 Clone 在堆上创建的 BufferOf 对象，并释放其所占内存空间。

```
static void Free (
            Const BufferOf<T> *pArray      //   源 BufferOf 对象
```

```
                    )
{
        _CarQuintet_Free(pArray);
}
```

GetPayload()

获得指向 BufferOf 数组的指针。

```
T * GetPayload() const
{
    return (T*)m_pBuf;
}
```

GetCapacity()

获取当前 BufferOf 数组总长度。

```
Int32 GetCapacity() const
{
    return m_size/sizeof(T);
}
```

GetAvailableSpace()

获取当前 BufferOf 数组可用长度。

```
Int32 GetAvailableSpace() const
{
return  ( m_size-m_used ) /sizeof(T);
}
```

GetUsed ()

取得 BufferOf 数组的已使用的数组元素数。

```
Int32 SetUsed() const
```

```
{
    return m_used/sizeof(T);
}
```

IsNull()

判断指向 BufferOf 数组的数据区指针是否为空。

```
Boolean IsNull ( ) const
{
    return m_pBuf==NULL;
}
```

IsEmpty()

判断 BufferOf 数组的数据区内容是否为空。

```
Boolean IsEmpty ( ) const
{
    return m_used==0;
}
```

IsNullorEmpty()

判断 BufferOf 数组的数据区内容或指向 BufferOf 数组的数据区指针否为空。

```
Boolean IsNullorEmpty ( ) const
{
    return ( m_pBuf==NULL || m_used==0);
}
```

Insert()

在当前 BufferOf 数组指定位置插入源数组中指定个数的数据。

```
BufferOf & Insert(
```

```
                const T *pBuf, //   pBuf：源 T 类型数组
                Int32 offset,    //   offset:  当前 BufferOf 对象待要插入数据的起始位置
                Int32 n          //   n:       当前 BufferOf 对象将要插入的数据个数
)
{
    _BufferOf_Insert(this, (PByte)pBuf, offset *sizeof(T), n*sizeof(T));
    return *this;
}
```

SetUsed()

设置 BufferOf 数组的已使用的数组元素数。

```
Int32 SetUsed(
            Int32 used      //已使用长度设置值
        )
{
    If (used<0)
        { return -1;}
    used=MIN(used,GetCapacity());
    m_used=used*sizeof(T);
    return used;
}
```

Replace()

用源数据替换 BufferOf 数组从指定位置的内容。

```
BufferOf & Replace(
                Int32 offset,   // offset:  当前 BufferOf 对象待开始替换数据的起始位置
                const T *pBuf, //   pBuf：源 T 类型数组
                Int32 n          //   n:     要替换的数据个数
                    )
```

```
{
    _BufferOf_Replace(this, offset *sizeof(T), (PByte)pBuf, n*sizeof(T));
    return *this;
}
```

操作符

[]

BufferOf 对象的存取数组元素操作符，用来对数组元素进行存取操作。

```
 T& operator [] (
                Int32 idx           //   数组下标
              )
{
          assert( m_pBuf   &&   index >= 0   &&   index < GetUsed() );
          return ( (T*) (m_pBuf) )[index];
}
```

[]

BufferOf 对象的存取数组元素操作符，用来对数组元素进行存取操作。

```
 const T & operator [] (
                Int32 idx           //   数组下标
              )
{
          assert( m_pBuf   &&   index >= 0   &&   index < GetUsed() );
          return ( (T*) (m_pBuf) )[index];
}
```

PCarQuintet

取五元组的指针。

```
operator PCarQuintet (   )
```

```
{
        return this;
}
```

ArrayOf<T> &

类型转换符，将 BufferOf 对象转化为 ArrayOf 对象。

```
operator PCarQuintet (   )
{
        return this;
}
```

5.3.3 ArrayOf

ArrayOf 类型定义：

```
template <class T>
class ArrayOf : public CarQuintet
{
    public:
        // public section
     private:
        //private section
}
```

构造函数

ArrayOf(const T *pBuf, Int32 capacity)

在栈中构造一个对象，使指针指向用户指定的内存块，同时设置其长度 capacity。

```
ArrayOf (
            const T   *pBuf;      //   T 类型数组
            Int32 capacity        //   待构造的 ArrayOf 数组区最大长度
```

```
                )
{
                _CarQuintet_Init(this, pBuf, capacity * sizeof(T),
                                        capacity * sizeof(T), Type2Flag<T>::Flag());
}
```

方法

Alloc()

在堆上动态创建 ArrayOf 对象，并指定数据区大小为 capacity。

```
Static ArrayOf<T> *Alloc(
                Int32 capacity      // 待创建的 ArrayOf 对象最大数据个数
                    )
{
    return(ArrayOf<T>*)_ArrayOf_Alloc(
                                        capacity * sizeof(T),
                                        Type2Flag<T>::Flag()
                                         );
}
```

Alloc()

在堆上动态创建 ArrayOf 对象，使指针指向用户指定的内存块，并指定 ArrayOf 数据区大小为 capacity。

```
Static ArrayOf<T> *Alloc(
                T *pBuf ,           //   T 类型数组
                Int32 capacity          // 待创建的 ArrayOf 对象数据区已使用数组元素个
数
                    )
{
    return(ArrayOf<T>*)_ArrayOf_Alloc_Box(
```

```
                                        pBuf,capacity * sizeof(T),
                                        capacity *sizeof(T),Type2Flag<T>::Flag()
                                        );
}
```

Clone ()

获取一个 ArrayOf 对象的深拷贝，即在堆上复制 ArrayOf 数组。

```
ArrayOf<T> *Clone() const
{
    return (ArrayOf<T> *)_CarQuintet_Clone((const PCarQuintet)this);
}
```

Copy()

将源 ArrayOf 对象数据区的内容复制到当前 ArrayOf 对象数据区内。

```
ArrayOf& Copy(
            const ArrayOf<T> *pSrc    //源 ArroyOf 对象
            )
{
        _ArrayOf_Copy(this, pSrc);
        return *this;
} //返回值说明：返回当前 ArrayOf 对象引用。如果源 ArrayOf 数组为空，则返回-1；
拷贝大小为源 ArrayOf 对象已用数据区大小和当前对象数据区总长度两者中取其中较
小的那个。
```

Copy()

将 T 类型指针 pBuf 所指内容复制到当前 ArrayOf 对象数据区内，并制定将要复制的数据个数为 n。

```
ArrayOf & Copy(
        const T *pBuf,     //     T 类型数组
```

```
            Int32 n             //      待复制的数组元素个数
        )
{
    _ArrayOf_CopyEx(this,(Byte*)pBuf,n*sizeof(T));
    return *this;
}
```

Free()

释放 ArrayOf 对象。

```
 static void Free (
                Const ArrayOf<T> *pArray       //    源 ArrayOf 对象
                    )
{
        _CarQuintet_Free(pArray);
}
```

GetPayload()

得到指向 ArrayOf 对象的数据区的指针，即返回指向类型 T 的指针。

```
T * GetPayload() const
{
    return (T*)m_pBuf;
}
```

GetLength()

获取当前 ArrayOf 对象数据区大小。

```
Int32 GetLength(
                    ) const
{
            return m_pBuf ? m_size / sizeof(T) : 0;
```

```
}
```

Replace()

替换当前 ArrayOf 对象数据区指定位置开始的 n 个数据。

```
ArrayOf & Replace(
                Int32 offset,    // offset: 当前 ArrayOf 对象待开始替换数据的起始位置
                const T *pBuf, //    pBuf：源 T 类型数组
                Int32 n          //    n:     要替换的数据个数
                    )
{
    _ArrayOf_Replace(this, offset *sizeof(T), (PByte)pBuf, n*sizeof(T));
    return *this;
}
```

操作符

[]

ArrayOf 对象的存取数组元素操作符，用来对数组元素进行存取操作。

```
 T& operator [] (
                Int32 idx            //    数组下标
               )
{
            assert( m_pBuf    &&    index >= 0    &&    index < GetUsed() );
            return ( (T*) (m_pBuf) )[index];
}
```

[]

ArrayOf 对象的存取数组元素操作符，用来对数组元素进行存取操作。

```
 const T & operator [] (
                Int32 idx            //    数组下标
```

```
            )
{
            assert( m_pBuf   &&   index >= 0   &&   index < GetUsed() );
            return ( (T*) (m_pBuf) )[index];
}
```

PCarQuintet

取五元组的指针。

```
operator PCarQuintet (   )
{
          return this;
}
```

BufferOf<T> &

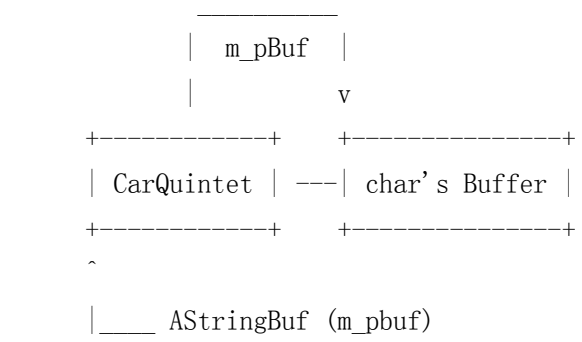
类型转换符，将 ArrayOf 对象转化为 ArrayOf 对象。

```
operator PCarQuintet (   )
{
          return this;
}
```

5.3.4 AStringBuf/WStringBuf

AStringBuf 是为了支持构件化编程而设计的数据结构。AStringBuf 可以转换成 AChar 类型的字符数组，也进行各种操作。

AStringBuf 是由一个五元组 CarQuintet 和一个存放 char 类型字符串的数据区组成，其内存结构如下：



在 CAR 规范中，AString 是所有字符串类型的入口参数，AStringBuf 是所有字符串类型的出口参数。AStringBuf 的对象由客户端负责创建及释放。

AStringBuf 是 C++实现的类，若在栈中声明该类的一个对象的方法为 AStringBuf_<size> buf，其中 buf 为该对象变量名称，size 是被封装的 AStringBuf 的有效容量。

WStringBuf 可以转换成 WChar 类型的字符数组，也可以进行各种操作。WStringBuf 的所有构造函数、方法和操作符都与 AStringBuf 的一致，AStringBuf 中的 char 和 AChar 分别对应 WStringBuf 中的 wchar_t 和 WChar，在此不累述。

定义枚举类型

```
typedef enum
{
    NumberFormat_RightAlign             = 0x00000000,
    NumberFormat_LeftAlign              = 0x00010000,
    NumberFormat_Signed                 = 0x00020000,
    NumberFormat_ZeroPrefixed           = 0x00040000,
    NumberFormat_BlankPrefixed          = 0x00080000,
    NumberFormat_PoundSign              = 0x00100000,
    NumberFormat_PrefixMask             = 0x00FF0000,

    NumberFormat_Decimal                = 0x00000000,
    NumberFormat_UsignedDecimal         = 0x01000000,
    NumberFormat_Octal                  = 0x02000000,
    NumberFormat_Hex                    = 0x04000000,
    NumberFormat_BigHex                 = 0x08000000,
    NumberFormat_IntegerMask            = 0x0F000000,

    NumberFormat_Double                 = 0x00000000,
    NumberFormat_ScientificDouble       = 0x10000000,
    NumberFormat_BigScientificDouble    = 0x20000000,
    NumberFormat_FlexScientificDouble   = 0x40000000,
    NumberFormat_BigFlexScientificDouble = 0x80000000,
    NumberFormat_DoubleMask             = 0xF0000000,

    NumberFormat_TypeMask               = 0xFF000000,

    NumberFormat_Mask                   = 0xFFFF0000,
} _NumberFormat;
```

AStringBuf 类型定义：

```
#include <ezstring.h>     //包含 AString/WString 定义头文件

template <class T>
class AStringBuf : public CarQuintet
{
      public:
          // public section
       private:
          //private section
}
```

构造函数

AStringBuf(char *pstr,Int32 size)

AStringBuf 的构造函数，用于构造一个 AStringBuf 对象，并指定其字符串长度。

```
AStringBuf(
              char *pstr,
              Int32 size)
 {
          assert(pstr && size >= 0);
          _AStringBuf_Box_Init(this, pstr, size, FALSE);
}
```

AStringBuf(const AStringBuf& buf)

构造一个 AStringBuf 对象

```
AStringBuf(
             const AStringBuf& buf
             )
{
}
```

方法

Alloc()

动态创建(在堆上) AStringBuf 对象，数据区大小为 n。

```
static AStringBuf *Alloc(
                    Int32 n       //数据区大小
                        )
{
            return (AStringBuf *)_AStringBuf_Alloc(n);
}
```

Alloc()

动态创建(在堆上) AStringBuf 对象。

```
static AStringBuf *Alloc(
                        char *pstr,
                        Int32 size
                        )
{
            assert(pstr);
            return (AStringBuf *)_AStringBuf_Alloc_Box(pstr, size);
}
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符串。

```
AStringBuf& Append(
                        AString as          //待插入的字符串
                        )
```

```
    {
        return Insert(GetLength(), as);
    }
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符串从指定偏移位置开始的指定个数的字符。

```
AStringBuf& Append(
                        AString as,     //待插入的字符串
                        Int32 offset,     //开始插入的位置
                        Int32 count)     //待插入的字符个数
    {
        _AStringBuf_Append_AString(this, as, offset, count);
        return *this;
    }
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符串，并指定编码格式。

```
AStringBuf& Append(
                    WString ws,     //待插入的 WString 对象
                    AString encoding = NULL    //特定的编码格式
                    )
    {
        _AStringBuf_Append_WString(this, ws, encoding);
        return *this;
    }
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符。

```
AStringBuf& Append(
                AChar ac    //追加的字符
                )
    {
        _AStringBuf_Append_AChar(this, ac);
        return *this;
    }
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符。

```
AStringBuf& Append(
                WChar ac    //追加的字符
                )
    {
        _AStringBuf_Append_WChar(this, ac);
        return *this;
    }
```

Append()

在 AStringBuf 对象中原字符串末尾追加另一指定字符串。

```
AStringBuf& Append(
                    Boolean Boolean    //追加的 boolean 值
                    )
    {
        if (boolean) {
            return Append("True");
        }
        else {
            return Append("False");
```

```
        }
    }
```

Append()

按一定格式在 AStringBuf 对象中原字符串末尾追加 Int8 数值。

```
AStringBuf& Append(
                    Int8 value,
                    UInt32 fmt = 0)
    {
        _AStringBuf_Append_Int8(this, value, fmt);
        return *this;
    }
```

Append()

按一定格式在 AStringBuf 对象中原字符串末尾追加 Int16 数值。

```
AStringBuf& Append(
                    Int16 value,
                    UInt32 fmt = 0)
    {
        _AStringBuf_Append_Int16(this, value, fmt);
        return *this;
    }
```

Append()

按一定格式在 AStringBuf 对象中原字符串末尾追加 Int32 数值。

```
AStringBuf& Append(
                        Int32 value,
                        UInt32 fmt = 0)
{
```

```
        _AStringBuf_Append_Int32(this, value, fmt);
        return *this;
}
```

Append()

按一定格式在 AStringBuf 对象中原字符串末尾追加 Int64 数值。

```
AStringBuf& Append(
                    Int64 value,
                    UInt32 fmt = 0)
{
        _AStringBuf_Append_Int64(this, value, fmt);
        return *this;
 }
```

Append()

按一定格式在 AStringBuf 对象中原字符串末尾追加 double 数值。

```
AStringBuf& Append(
                Double value,
                UInt32 fmt = 0)
{
        _AStringBuf_Append_Double(this, value, fmt);
        return *this;
}
```

Clone ()

获取一个 AStringBuf 对象的克隆，即在堆上复制 AStringBuf 对象和指向的字符串。

```
AStringBuf *Clone() const

{
```

```
        return (AStringBuf *)_CarQuintet_Clone((const PCarQuintet)this);

}
```

Copy()

将源 AString 对象内容拷贝到当前 AStringBuf 对象的数据区中。

```
AStringBuf& Copy(

                AString as //待拷贝的源AString对象

                )

{

        _AStringBuf_Copy(this, as);

        return *this;

}
```

Copy()

以特定的编码格式将 WString 对象的内容拷贝到当前 AStringBuf 对象数据区中

```
AStringBuf& Copy(

            WString ws,       //特定的WString对象

            AString encoding = NULL    //指定的编码格式

            )

{

        SetEmpty();

        _AStringBuf_Append_WString(this, ws, encoding);

        return *this;

}
```

Contains()

判断 AStringBuf 数据区中否包含指定字符串

```
Boolean Contains(
                        const char *substr,     //待查找的宽字符子串
                        StringCase stringCase = StringCase_Sensitive,//大小写标示符
                         Encoding encoding = Encoding_Default//  指定字符编码方式
                        ) const
{
          return _WStringBuf_Contains((const PCarQuintet)this, substr, stringCase);
}
```

Compare()

把 AStringBuf 对象中数据内容与另一个给定的字符串作比较。

```
Int32 Compare(
                    AString as,       //待比较的字符串
                    StringCase stringCase = StringCase_Sensitive    //大小写标示符
                    ) const
{
            return _AString_Compare((char *)m_pBuf, as, stringCase);
}
```

Compare ()

把 AStringBuf 对象中原字符串与另一个给定的 AStringBuf 对象中的字符串作比较。

```
Int32 Compare(

        const AStringBuf& asb, //待比较的另一AStringBuf对象

        StringCase stringCase = StringCase_Sensitive    //大小写标示符

        ) const

{

        return _AString_Compare((char *)m_pBuf, asb, stringCase);

}
```

EndWith()

AStringBuf 对象数据区内容是否以指定字符串结尾

```
Boolean EndWith(

            const char *substr,     // 待比较的宽字符子串

            StringCase stringCase = StringCase_Sensitive,//大小写标示符

            Encoding encoding = Encoding_Default    //编码方式

                ) const

{

        return _WStringBuf_StartWith((const PCarQuintet)this, substr,
stringCase);

}
```

Free()

释放由 Alloc 或 Clone 方法为 AStringBuf 对象分配的内存空间。

```
static void Free(
                AStringBuf *pBuf
                )
{
        _CarQuintet_Free(pBuf);

}
```

GetChar ()

按指定编码取 AStringBuf 对象数据区中 Index 位置的字符。

```
Char32 GetChar(

                Int32 index,     //字符索引值

                Encoding encoding = Encoding_Default     //编码方式

                ) const

 {

        if (m_used < (Int32)sizeof(AChar)) return 0xFFFFFFFF;

        return _AStringBuf_CharAt((PCarQuintet)this, index, encoding);

  } //返回值说明：如果AStringBuf对象内容为为空，则程序被终止。否则返回相应index
                位置上的字符
```

GetCharCount ()

获得在指定编码格式下的 AStringBuf 对象数据区内容字符串长度。

```
Int32 GetCharCount(

                Encoding encoding = Encoding_Default    //指定编码值

                ) const {
```

```
    if (m_used < (Int32)sizeof(AChar)) return 0;

    return _AStringBuf_GetCharCount((PCarQuintet)this, encoding);

}  //返回值说明：如果数据区内容为空，则返回0；否则返回数据区内容字符数
```

GetPayload()

返回指向 AStringBuf 对象的数据区的指针，即返回指向窄字符串的指针。

```
char *GetPayload() const

{

    return (char *)m_pBuf;

 }
```

GetCapacity()

获取数据区的总容量，以字符数为单位。

```
Int32 GetCapacity() const

{

    if (!m_pBuf || m_size < (Int32)sizeof(AChar)) return 0;

    return m_size / sizeof(AChar) - 1;

  }
```

GetLength()

获取当前已使用的数据区长度，以字符数为单位。

```
Int32 GetLength() const
```

```
{

        if (m_used < (Int32)sizeof(AChar)) return 0;

        return m_used / sizeof(AChar) - 1;

}
```

GetAvailableSpace()

获取当前 AStringBuf 对象数据区还可用长度，以字符为单位。

```
Int32 GetAvailableSpace() const

{

        return (m_size - m_used) / sizeof(AChar);

}
```

Insert()

向当前数据区制定位置插入特定字符串。

```
AStringBuf& Insert(

                Int32 offset,     //开始插入的位置

                AString as)       //插入的字符串

    {

        _AStringBuf_Insert(this, offset, as);

        return *this;

    }
```

IsNull()

判断指向当前 AStringBuf 对象数据区的指针是否为空。

```
Boolean IsNUll () const

{

    return m_pBuf=NULL;

 }
```

IsEmpty()

判断当前 AStringBuf 对象的数据区内容是否为空。

```
Boolean IsEmpty() const {

        assert(m_pBuf);

        return (m_used == sizeof(AChar)) && (*(AChar *)m_pBuf == '\0');

    }
```

IsNullorEmpty()

判断指向当前 AStringBuf 对象的数据区指针或者其内容是否为空。

```
Boolean IsNullOrEmpty() const {

        return m_pBuf == NULL || *(AChar *)m_pBuf == '\0';

    }
```

IndexOf()

获得指定字符在 AStringBuf 对象数据区内容中的索引值

```
Int32 IndexOf(

                AChar ch,          //指定字符
```

```
                StringCase stringCase = StringCase_Sensitive     //大小写标示符

                ) const

{

        return _AStringBuf_IndexOf_AChar((const PCarQuintet)this, ch,
stringCase);

}
```

IndexOf()

获得指定字符串在 AStringBuf 数据区内容出现的索引值

```
Int32 IndexOf(

                AString str,            //指定字符串

                StringCase stringCase    = StringCase_Sensitive, //大小写标示符

                Encoding encoding = Encoding_Default     //指定编码方式

                 ) const

{

          return _AStringBuf_IndexOf_SubString((const PCarQuintet)this, str,

                stringCase, encoding);

 }
```

IndexOfAny()

返回特定字符串中任意一个字符在当前对象数据区内容中的索引值。

```
Int32 IndexOfAny(

                AString strCharSet,        //待查找的字符串
```

```
                StringCase stringCase = StringCase_Sensitive       //大小写标示符

                    ) const

{

        return _AStringBuf_IndexOf_AnyAChar((const PCarQuintet)this,
strCharSet, stringCase);

 }

//返回值说明：返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的索引值，
                否则返回-1
```

IndexOfChar()

返回指定字符在当前对象数据区内容的位置

```
Int32 IndexOfChar(

                    Char32 ch,      //指定字符

                   StringCase stringCase   = StringCase_Sensitive,   //大小写标
                                                                       示符

                    Encoding encoding = Encoding_Default   //指定编码方式

                  ) const

 {

        return _AStringBuf_IndexOf_Char((const PCarQuintet)this, ch, stringCase,

                encoding);

    }
```

IndexOfAnyChar()

返回指定字符串中任意一个字符在当前对象数据区内容的最后的索引值。

```
Int32 IndexOfAnyChar(
                Char32 *strCharSet,       //字符指针
                StringCase stringCase = StringCase_Sensitive,    //大小写标示符
                Encoding encoding = Encoding_Default       //指定编码格式
                ) const
{
    return _AStringBuf_IndexOf_AnyChar((const PCarQuintet)this, strCharSet,
                stringCase, encoding);
 }
```

LastIndexOf ()

获得字符最后一次出现的索引值

```
Int32 LastIndexOf(
                AChar ch,     //指定字符
                StringCase stringCase = StringCase_Sensitive      //大小写标示
                ) const
{
          return _AStringBuf_LastIndexOf_AChar((const PCarQuintet)this, ch,
stringCase);
}    //返回值说明：如果此索引值不存在则返回-1
```

LastIndexOf ()

获知特定的字符串是否在当前对象的 m_string 字符串中出现过，返回值为 0 或 1

```
Int32 LastIndexOf(
                    AString str,       //指定字符串
                    StringCase stringCase    = StringCase_Sensitive,    //大小写标示符
                    Encoding encoding = Encoding_Default    //指定编码方式
                  ) const
{
        return _AString_LastIndexOf_SubString(m_string, str, stringCase, encoding);
 }
```

LastIndexOfAny ()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的最后的索引值。

```
Int32 LastIndexOfAny(
                    AString strCharSet,          //指定字符串
                    StringCase stringCase = StringCase_Sensitive    //大小写标示符
                       ) const
{
        return _AStringBuf_LastIndexOf_Char((const PCarQuintet)this, ch,
stringCase, encoding);
 }
```

LastIndexOfChar ()

获得指定字符串在 AString 对象中的最后的索引值

```
Int32 LastIndexOfChar(

                Char32 ch,                  //指定的待索引的字符

                StringCase stringCase    = StringCase_Sensitive,    //大小写标示符

                Encoding encoding = Encoding_Default

                ) const

{

        return _AStringBuf_LastIndexOf_Char((const PCarQuintet)this, ch,
stringCase, encoding);

    }
```

LastIndexOfAnyChar ()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的最后的索引值。

```
Int32 LastIndexOfAnyChar(

                Char32 *strCharSet,     /字符型指针

                StringCase stringCase = StringCase_Sensitive,    //大小写标示符

                Encoding encoding = Encoding_Default          //指定的编码方式

                  ) const

{

        return _AStringBuf_LastIndexOf_AnyChar((const PCarQuintet)this,
strCharSet, stringCase, encoding);       }
```

PadLeft()

在原 AStringBuf 对象数据区内容中添加特定长度的空格字符。

```
AStringBuf& PadLeft(

            Int32 width  //待增加的字符个数

             )

{

        _AStringBuf_PadLeft((PCarQuintet)this, width);

        return *this;

    }
```

PadLeft( )

在原 AStringBuf 对象数据区内容中添加特定长度的特定字符。

```
AStringBuf& PadLeft(

                    Int32 width,//代替换的字符个数

                    AChar ch    //待替换的字符

)

{

        _AStringBuf_PadLeft_AChar((PCarQuintet)this, width, ch);

        return *this;

    }
```

PadRight()

用制定长度设置 AStringBuf 对象数据区长度。

```
AStringBuf& PadRight(

                Int32 width //代替换的字符个数

                )

{

        _AStringBuf_PadRight((PCarQuintet)this, width);

        return *this;

    }
```

PadRight()

用制定长度设置 AStringBuf 对象数据区长度。

```
AStringBuf& PadRight(

                Int32 width,  //代替换的字符个数

                AChar ch       //待替换的字符

                )

{

        _AStringBuf_PadRight_AChar((PCarQuintet)this, width, ch);

        return *this;

    }
```

Replace()

将缓冲区中从指定位置开始替换为源数据前指定个字符。

```
AStringBuf& Replace(

                Int32 offset,    //开始位置

                Int32 count,    //字符个数

                AString as      //待替换的字符串

                )

{

        _AStringBuf_Replace(this, offset, count, as);

        return *this;

}
```

SetLength()

用制定长度设置 AStringBuf 对象数据区长度。

```
Int32 SetLength(

                Int32 len     //待设置的数据区长度

                ) const

{

        return _AStringBuf_SetLength((PCarQuintet)this, len);

}//返回值说明：如果 len<0,则返回原数据区长度，否则将此对象的数据区长度设置
                为 len
```

SetEmpty()

设置 AStringBuf 对象数据区内容为空。

```
void SetEmpty()

{

        m_used = sizeof(AChar);

        *(AChar *)m_pBuf = '\0';

    }
```

StartWith()

判断 AStringBuf 对象数据区内容是否是以指定字符串。

```
Boolean StartWith(

                const char *substr,     //待比较的字符串

                StringCase stringCase = StringCase_Sensitive,     //大小写标示符

                Encoding encoding = Encoding_Default       //指定的编码格式

                ) const

{

       return _AStringBuf_StartWith((const PCarQuintet)this, substr, stringCase);

   }
```

SubString ()

获得当前 AStringBuf 数据区内容开始位置的字符串

```
AString SubString(

                    Int32 start,    //开始位置
```

```
                    Encoding encoding = Encoding_Default    //指定编码格式

                    )

{

        return _AStringBuf_SubString((const PCarQuintet)this, start, encoding);

}
```

SubString ()

返回 AStringBuf 对象数据区内容字符串子串。

```
AString SubString(

        Int32 start,     //起始位置

        Int32 len;       //子串长度

        Encoding encoding = Encoding_Default       //指定编码格式

        )

{

        return _AStringBuf_SubString_Length((const PCarQuintet)this, start,

            len, encoding);

}
```

ToInt32()

将 AStingBuf 对象数据区内容转化为 32 位整型数。

```
Int32 ToInt32() const

{

        return _AStringBuf_ToInt32((const PCarQuintet)this);
```

```
}//返回值说明：如果AStringBuf对象数据区内容为空，则返回0；否则返回对应的
                32位整型数值
```

ToInt64()

将 AStingBuf 对象数据区内容转化为 64 位整型数值。

```
Int64 ToInt64() const

{

        return _AStringBuf_ToInt64((const PCarQuintet)this);

    }//返回值说明：如果AStringBuf对象数据区内容为空，则返回0；否则返回对应的
                64位整型数值
```

ToDouble()

将 AStingBuf 对象数据区内容转化为 Double 类型数值。

```
Double ToDouble() const

{

        return _AStringBuf_ToDouble((const PCarQuintet)this);

 }//返回值说明：如果AStringBuf对象数据区内容为空，则返回0；否则返回对应的
                double型数值
```

ToBoolean()

将 AStingBuf 对象数据区内容转化为 boolean 类型数值。

```
Boolean ToBoolean() const {
```

```
        return _AStringBuf_ToDouble((const PCarQuintet)this);

    } //返回值说明：如果AStringBuf对象数据区内容为空，则返回FALSE；AStringBuf
                对象数据区内容为”1”或者”true”时返回TRUE，否则返回FALSE
```

ToLowerCase ()

按某种编码格式将 AStringBuf 对象字符串中字符大写变小写。

```
AStringBuf& ToLowerCase(

        Encoding encoding = Encoding_Default

        )

{

    _AStringBuf_ToUpperCase((PCarQuintet)this, encoding);

    return *this;

}
```

ToUpperCase ()

按某种编码格式将 AString 对象字符串中字符小写变大写

```
AStringBuf& ToUpperCase(

        Encoding encoding = Encoding_Default

        )

{

    _AStringBuf_ToUpperCase((PCarQuintet)this, encoding);

    return *this;

}
```

TrimStart ()

返回一个由非空格、水平制表、垂直制表、换页、回车和换行符开始的字符串作为数据区内容的 AStringBuf 对象。

```
AStringBuf& TrimStart() const

{

        AStringBuf_TrimStart((PCarQuintet)this);

         return *this;

}
```

TrimEnd ()

返回以源 AStringBuf 对象尾部由非空格、水平制表、垂直制表、换页、回车和换行符组成的字符串作为数据区内容的 AStringBuf 对象

```
AStringBuf& TrimEnd() const

{

        _AStringBuf_TrimEnd((PCarQuintet)this);

        return *this; }
```

Trim ()

将源 AString 对象对应的字符串前后的空格、水平制表、垂直制表、换页、回车和换行符去掉。

```
AStringBuf& Trim() const

{

        _AStringBuf_Trim((PCarQuintet)this);

         return *this;
```

```
}
```

操作符

Char *

类型转换操作符，将 AStringBuf 中的用户字符串转换成 char 的指针。

AString

类型转换操作符，把 AStringBuf 对象转换成 AString 型对象。

[]

存取数组元素操作符，用来对 AStringBuf 对象的数组元素进行存取操作。

>>

相当于 APPLEND。

宏

NULL-ASTRINGBUF

构造一个空的 AStringBuf 对象

NULL-ASTRINGBUF(n)

构造一个 AStringBuf 对象

5.3.5 AString/WString

WString 是为了支持构件化编程而设计的数据结构。WString 可以转换成(WChar)类型的字符数组，也可以进行各种操作。

WString 是由一个指针型成员变量 m_wstring 和一个存放 wchar_t 类型字符串的数据区组成，其内存结构如下：

```
+---------------+
| wchar's Buffer |
```

```
+---------------+
^

|____ WString (m_wstring)
```

AString 是为了支持构件化编程而设计的数据结构。AString 可以转换成 AChar 类型的字符数组，也可以进行各种操作。

AString 的所有构造函数、方法和操作符都与 WString 的一致，WString 中的 wchar_t 和 WChar 分别对应 AString 中的 char 和 AChar，在此不累述。

设置是否区分大小写标示符枚举项：

```
typedef enum _StringCase
{
    StringCase_Sensitive      = 0x0000,//区分大小写
    StringCase_Insensitive    = 0x0001,//不区分大小写
} StringCase;
```

设置编码格式枚举项：

```
typedef enum _Encoding
{
    Encoding_ASCII            = 0x0001,
    Encoding_UTF7             = 0x0002,
    Encoding_UTF8             = 0x0003,
    Encoding_UTF16            = 0x0004,
    Encoding_GB18030          = 0x0005,
    Encoding_Default          = Encoding_GB18030
} Encoding;
```

AString 类型定义：

```
class AString
{
    public:
        // member functions declarations or definitions
     private:
        void operator==(const char *) {}
        void operator!=(const char *) {}
        void operator!() {}
        void operator*() {}
```

```
        void operator+=(const char *) {}
        void operator+(const char *) {}
        void operator+=(const int) {}
        void operator-=(const int) {}
        void operator+(const int) {}
        void operator-(const int) {}
        const char* m_string;}
```

构造函数

AString

构造一个 AString 对象，并初始化字符串指针为 NULL。

```
    AString ():m_string(NULL)
{


}// e.g.AString as;
```

AString

构造一个 AString 对象，并初始化字符串指针为特定字符串。

```
AString (const char *pChar):m_string(pChar)
{


} // e.g., AString as("Hello");
```

方法

Compare()

把 AString 对象中原字符串与另一个给定的 AString 对象中的字符串作比较。

```
Int32 Compare(
        AString str,     //指定的AString对象
        StringCase stringCase = StringCase_Sensitive //指定大小写标示符，默认为
                                                     区分大小写
        )const
{
        return _AString_Compare(m_string, str.m_string, stringCase);
}//e.g., as.Compare(str);
```

GetLength()

获取 AString 对象中字符串的长度，以字符数为单位。

```
Int32 GetLength(
                Int32 maxLen     //设置的最大长度，如果假如AString对象中字符串的
                                 字符个数超过maxLen，则返回-1，否则返回字符串
                                 的字符数，不包含"\0"结束符。
                ) const
{
        return _AString_Length(m_string, maxLen);//如果AString对象的m_string串为
                                                  空或者maxlen<0.，则返回-1
 }    // e.g., as.Length(64);
```

Contains()

判断是否包含指定字符串

```
Boolean Contains(
                        const char *substr, //待查找的子串
                        StringCase stringCase = StringCase_Sensitive,//大小写标示符
                         Encoding encoding = Encoding_Default//  指定字符编码方式
                        ) const
{
            return _AString_Contains(m_string, substr, stringCase, encoding);
}
```

EndWith()

是否以指定字符串结尾

```
Boolean EndWith(
                    const char *substr,     //待比较的子串
                    StringCase stringCase = StringCase_Sensitive,//大小写标示符
                    Encoding encoding = Encoding_Default    //编码方式
                        ) const
{
            return _AString_EndWith(m_string, substr, stringCase, encoding);
}
```

GetChar ()

按指定编码取 Index 位置的字符。

```
Char32 GetChar(
                    Int32 index,     //字符索引值
                    Encoding encoding = Encoding_Default     //编码方式
                    ) const
 {
          return _AString_CharAt(m_string, index, encoding);
  } //如果AString对象的m_string字符串为空，则程序被终止
```

GetCharCount ()

获得指定编码的字符串长度。

```
Int32 GetCharCount(
                        Encoding encoding = Encoding_Default    //指定编码值
                        ) const {
          return _AString_GetCharCount(m_string, encoding);
     }
```

IsNull()

判断当前 AString 对象的字符串指针是否为空。

```
Boolean IsNUll () const
{
      return m_string=NULL;
```

```
}    // e.g., if (str.IsNull()) {...} or Boolean b = str.IsNull();
```

IsEmpty()

判断当前 AString 对象的字符串内容是否为空。

```
Boolean IsEmpty() const {

        assert(m_string);

        return m_string[0] == '\0';

    }
```

IsNullorEmpty()

判断当前 AString 对象的字符串指针或者内容是否为空。

```
Boolean IsNullOrEmpty() const {

        return (m_string == NULL || m_string[0] == '\0');

    }
```

IndexOf()

获得指定字符在 AString 对象中的索引值

```
Int32 IndexOf(

            AChar ch,           //指定字符

            StringCase stringCase = StringCase_Sensitive     //大小写标示符

            ) const

{

        return _AString_IndexOf_AChar(m_string, ch, stringCase);

}
```

IndexOf()

获得指定字符串在 AString 对象是否出现过

```
Int32 IndexOf(
                AString str,            //指定字符串
                StringCase stringCase    = StringCase_Sensitive, //大小写标示符
                Encoding encoding = Encoding_Default     //指定编码方式
                 ) const
{
          return _AString_IndexOf_SubString(m_string, str, stringCase, encoding);
 }
```

IndexOfAny()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的索引值。

```
Int32 IndexOfAny(
                AString strCharSet,        //待查找的字符串
                StringCase stringCase = StringCase_Sensitive       //大小写标示符
                    ) const
{
          return _AString_IndexOf_AnyAChar(m_string, strCharSet, stringCase);
   }
//返回值说明：返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的索引值，
                否则返回-1
```

IndexOfChar()

返回指定字符在当前对象字符串中的位置

```
Int32 IndexOfChar(
                        Char32 ch,      //指定字符
                       StringCase stringCase   = StringCase_Sensitive,   //大小写标
                                                                          示符
                        Encoding encoding = Encoding_Default   //指定编码方式
                      ) const
 {
          return _AString_IndexOf_Char(m_string, ch, stringCase, encoding);
     }
```

IndexOfAnyChar()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的最后的索引值。

```
Int32 IndexOfAnyChar(
                Char32 *strCharSet,      //字符指针
                StringCase stringCase = StringCase_Sensitive,   //大小写标示符
                Encoding encoding = Encoding_Default      //指定编码格式
                ) const
{
    return _AString_IndexOf_AnyChar(m_string, strCharSet, stringCase, encoding);
 }
```

LastIndexOf ()

获得字符最后一次出现的索引值

```
Int32 LastIndexOf(
                AChar ch,     //指定字符
                StringCase stringCase = StringCase_Sensitive     //大小写标示
                ) const
{
          return _AString_LastIndexOf_AChar(m_string, ch, stringCase);
}    //返回值说明：如果此索引值不存在则返回-1
```

LastIndexOf ()

获知特定的字符串是否在当前对象的 m_string 字符串中出现过，返回值为 0 或 1

```
Int32 LastIndexOf(
                AString str,      //指定字符串
                StringCase stringCase    = StringCase_Sensitive,   //大小写标示符
                Encoding encoding = Encoding_Default    //指定编码方式
               ) const
{
      return _AString_LastIndexOf_SubString(m_string, str, stringCase, encoding);
 }
```

LastIndexOfAny ()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的最后的索引值。

```
Int32 LastIndexOfAny(
                AString strCharSet,          //指定字符串
                StringCase stringCase = StringCase_Sensitive    //大小写标示符
                    ) const
{
        return _AString_LastIndexOf_AnyAChar(m_string, strCharSet, stringCase);
 }
```

LastIndexOfChar ()

获得指定字符串在 AString 对象中的最后的索引值

```
Int32 LastIndexOfChar(
                Char32 ch,                    //指定的待索引的字符
                StringCase stringCase    = StringCase_Sensitive,    //大小写标示符
                Encoding encoding = Encoding_Default
                ) const
{
            return _AString_LastIndexOf_Char(m_string, ch, stringCase, encoding);
     }
```

LastIndexOfAnyChar ()

返回 strCharSet 中任意一个字符在当前对象 m_string 字符串中的最后的索引值。

```
Int32 LastIndexOfAnyChar(
```

```
                Char32 *strCharSet,     /字符型指针

                StringCase stringCase = StringCase_Sensitive,    //大小写标示符

                Encoding encoding = Encoding_Default          //指定的编码方式

                  ) const

{

        return _AString_LastIndexOf_AnyChar(m_string, strCharSet,

                stringCase, encoding);

    }
```

StartWith()

判断 AString 对象字符串头是否是以指定字符串。

```
Boolean StartWith(

                const char *substr,     //待比较的字符串

                StringCase stringCase = StringCase_Sensitive,     //大小写标示符

                Encoding encoding = Encoding_Default       //指定的编码格式

                ) const

{

          return _AString_StartWith(m_string, substr, stringCase, encoding);

   }
```

SubString ()

获得当前 AString 对象开始位置的字符串

```
AString SubString(
                    Int32 start,    //开始位置
                    AStringBuf& sub,    //AStringBuf对象引用
                     Encoding encoding = Encoding_Default    //指定编码格式
                    )
{
        return _AString_SubString_Buffer(m_string, start,
            (PCarQuintet)&sub, encoding);
    }
```

SubString ()

返回 AString 对象字符串子串。

```
AString SubString(
            Int32 start,     //起始位置
            Encoding encoding = Encoding_Default       //指定编码格式
            )
{
        return _AString_SubString(m_string, start, encoding);
}
```

SubString ()

返回 AString 对象字符串子串。

```
AString SubString(
```

```
                Int32 start,        //开始位置

                Int32 len,          //子串长度

                AStringBuf& sub,    //AStringBuf对象引用

                Encoding encoding = Encoding_Default     //编码格式

                )

{

        return _AString_SubString_Length_Buffer(m_string, start, len,

                (PCarQuintet)&sub, encoding);

  }
```

SubString ()

返回 AString 对象字符串子串。

```
AString SubString(

                    Int32 start,     //开始位置

                    AStringBuf& sub,    //AStringBuf对象引用

                     Encoding encoding = Encoding_Default    //编码格式

                     )

{

        return _AString_SubString_Buffer(m_string, start,

                (PCarQuintet)&sub, encoding);

    }
```

ToInt32()

将 ASting 对象转化为 32 位整型数。

```
Int32 ToInt32() const

{

        return _AString_ToInt32(m_string);

    }//返回值说明：如果AString对象字符串为空，则返回0；否则返回字符串对应的
                32位整型数值
```

ToInt64()

将 ASting 对象转化为 64 位整型数值。

```
Int64 ToInt64() const

{

        return _AString_ToInt64(m_string);

    }//返回值说明：如果AString对象字符串为空，则返回0；否则返回字符串对应的
                64位整型数值
```

ToDouble()

将 ASting 对象转化为 Double 类型数值。

```
Double ToDouble() const

{

        return _AString_ToDouble(m_string);
```

```
}//返回值说明：如果AString对象字符串为空，则返回0；否则返回字符串对应的double
                型数值
```

ToBoolean()

将 ASting 对象转化为 boolean 类型数值。

```
Boolean ToBoolean() const {

        return _AString_ToBoolean(m_string);

} //返回值说明：如果AString对象字符串为空，则返回FALSE；AString对象字符
                串为”1”或者”true”时返回TRUE，否则返回FALSE
```

ToLowerCase ()

按某种编码格式将 AString 对象字符串中字符大写变小写。

```
AString ToLowerCase(

        AStringBuf& lowser,

        Encoding encoding = Encoding_Default

        )

{

        return _AString_ToUpperCase(m_string, (PCarQuintet)&lowser,
encoding);

}
```

ToUpperCase ()

按某种编码格式将 AString 对象字符串中字符小写变大写

```
AString ToUpperCase(
```

```
                AStringBuf& upper,

                Encoding encoding = Encoding_Default

                )

{

        return _AString_ToUpperCase(m_string, (PCarQuintet)&upper, encoding);

}
```

TrimStart ()

返回一个由非空格、水平制表、垂直制表、换页、回车和换行符开始的字符串

```
AString TrimStart(

          AStringBuf& str     //AStringBuf对象引用

       ) const

{

        return _AString_TrimStart(m_string, (PCarQuintet)&str);

}
```

TrimEnd ()

返回源 AString 对象尾部由非空格、水平制表、垂直制表、换页、回车和换行符组成的字符串

```
AString TrimEnd(

                AStringBuf& str

                ) const

{
```

```
        return _AString_TrimEnd(m_string, (PCarQuintet)&str);

 }
```

Trim ()

将源 AString 对象对应的字符串前后的空格、水平制表、垂直制表、换页、回车和换行符去掉

```
AString Trim(

        AStringBuf& str     //AStingBuf对象引用

        ) const

{

        return _AString_Trim(m_string, (PCarQuintet)&str);

 }
```

操作符

=

给 AString 对象赋值

=

给 AString 对象赋值

[]

存取数组元素操作符，用来对 AString 对象中的字符串数组元素进行存取操作。

Const char *

类型转换操作符，把 AString 对象转换成字符串的

```
举例

AString as = "hello";
char *str = (char*)(const char*)as;
 const char *str = as; //尽量使用这种方式。
 strcmp(as, "hi); //就可以了，很多代码使用了 strcmp((char *)as, "hi); 现在是错误的。
                  //如果参数是 const char* 直接把 as 放进去就可以了，编译器会自动转换，
                  //没必要强转如 strcmp((const char*)as, "hi)
```

5.3.6 MemoryBuf

MemoryBuf 继承自 BufferOf。操作对象是字节的内存块。

MemoryBuf 构造函数和各种方法使用介绍：

以下只列出比 BufferOf 多的方法，相同或相似的方法请参考 BufferOf

操作符与 BufferOf 相同

MemoryBuf 类型定义：

```
template <class T>
class MemoryBuf : public BufferOf<Byte>
{
      public:
            // member functions declarations or definitions
       private:
            //constructors or some operator
}
```

构造函数

MemoryBuf

在栈中构造一个 MemoryBuf 对象，包括 capacity 个字节的 Buffer 内存

```

```

```
    MemoryBuf (
                Byte *pBuf,
                Int32 capacity) : BufferOf<Byte>(pBuf, capacity)
{


}
```

MemoryBuf

在栈中构造一个 MemoryBuf 对象,将 m_pBuf 指向用户指定的内存块，并分别其长度和已使用长度。

```
MemoryBuf(
            Byte *pBuf,
            Int32 capacity,
            Int32 used) :BufferOf<Byte>(pBuf, capacity, used)
{


}
```

方法

Alloc()

在堆上动态创建 MemoryBuf 对象，并指定最大数据数为 capacity。

```
static MemoryBuf *Alloc(
                        Int32 capacity)
{
        return (MemoryBuf *)BufferOf<Byte>::Alloc(capacity);
}
```

Alloc()

在堆上动态创建 BufferOf 对象，并分别指定 BufferOf 数据区大小为 capacity。

```
static MemoryBuf *Alloc(
                        Byte *pBuf,
                        Int32 capacity)
{
        return (MemoryBuf *)BufferOf<Byte>::Alloc(pBuf, capacity);
}
```

Alloc()

在堆上动态创建 MemoryBuf 对象，并分别指定 MemoryBuf 数据区大小为 capacity 及已使用长度为 used。

```
static MemoryBuf *Alloc(
                        Byte *pBuf,
                        Int32 capacity,
                        Int32 used)
{
        return (MemoryBuf *)BufferOf<Byte>::Alloc(pBuf, capacity, used);
}
```

Append()

在 MemoryBuf 对象的数据区的已使用空间末尾添加数据 vaule。

```
MemoryBuf& Append(
                        Byte value
                        )
{
        BufferOf<Byte>::Append(value);
        return *this;
```

```
}
```

Append()

在 MemoryBuf 对象的数据区的已使用空间末尾添加数据。

```
MemoryBuf& Append(
                    const PCarQuintet pSrc    //五元组对象
                    )
{
        BufferOf<Byte>::Append((BufferOf<Byte> *)pSrc);
        return *this;
}
```

Append()

在 MemoryBuf 对象的数据区的已使用空间末尾添加 n 个数据。

```
MemoryBuf& Append(
                    const Byte* pBuf,     //待添加的数据串
                    Int32 n)                  //待添加的个数
{
        BufferOf<Byte>::Append(pBuf, n);
        return *this;
}
```

Clone ()

获取一个 MemoryBuf 对象的深拷贝，即在堆上复制数组。

```
MemoryBuf *Clone() const
{
        return (MemoryBuf *)BufferOf<Byte>::Clone();
```

```
}
```

Copy()

将源五元组对象数据区的内容复制到当前 MemoryBuf 对象数据区内。

```
MemoryBuf& Copy(
                const PCarQuintet pSrc)
{
        BufferOf<Byte>::Copy((BufferOf<Byte> *)pSrc);
        return *this;
}
```

Copy()

将 Byte 类型指针 pBuf 所指内容复制到当前 MemoryBuf 对象数据区内，并制定将要复制的数据个数为 n。

```
MemoryBuf& Copy(
                const Byte* pBuf,
                Int32 n
                )
{
        BufferOf<Byte>::Copy(pBuf, n);
        return *this;
}
```

Compare ()

与指定内存块比较 n 个字节，相当于 memcmp。

```
Int32 Compare(
                const Byte* pBuf,     //待比较的字符串
                Int32 n           //待比较的字符数
                ) const
```

```
{
        return _MemoryBuf_Compare((const PCarQuintet)this, pBuf, n);
}
```

Compare ()

和指定的五元组对象相比较。

```
Int32 Compare(
                const PCarQuintet pCq
                ) const
{
        assert(pCq && pCq->m_pBuf);
        return _MemoryBuf_Compare((const PCarQuintet)this,
                                        (const Byte*)pCq->m_pBuf, pCq->m_used
                                        );
}
```

Insert()

在当前 MemoryBuf 对象数据区指定位置插入 n 个数据

```
MemoryBuf& Insert(
                const Byte* p,      //待插入数据
                Int32 offset,       //指定位置
                Int32 n)
{
        BufferOf<Byte>::Insert(p, offset, n);
        return *this;
}
```

SetByteVaule ()

将 MemoryBuf 对象数据区内容设置为特定值。

```
MemoryBuf& SetByteValue(
                        Byte value     //待设置的值
                        )
{
            _MemoryBuf_SetByteValue(this, value);
            return *this;
}
```

Replace()

替换当前 MemoryBuf 对象数据区指定位置开始的 n 个数据。

```
MemoryBuf& Replace(
                    Int32 offset,                   //开始替换位置
                    const Byte* pBuf,           //待替换的数据
                    Int32 n)        //替换的个数
{
            BufferOf<Byte>::Replace(offset, pBuf, n);
            return *this;
}
```

宏

NULL_MEMORYBUF

构造一个内容为空的 MEMORYBUF<T>对象

AUTO_MEMORYBUF(n)

构造一个 MemoryBuf 对象。

5.3.7 ECode 返回值

在 CAR 文件中定义的接口方法，返回值类型全部都是 ECode。

ECode 是一个 32 位二进制整数。ECode 高 16 位中，最高位表示方法调用错误(1)或成功(0)；其余 15 位表示具体的错误信息，例如：接口错误、驱动错误、CRT 错误、文件系统错误、图形系统错误等。对于许多 CAR 兼容的实现语言（例如：Visual Basic、Java）而言，这些 ECode 被运行时库或者虚拟机截取，然后被映射为语言中特定的异常（exception）。ECode 低 16 位用作调试和测试用。

常见的 ECode 返回值及其解释，列表如下：

返回值	十六进制值	描述
NOERROR	0x00000000	方法调用成功
E_PROCESS_NOT_ACTIVE	0x81010000	进程不是活动状态
E_PROCESS_STILL_ACTIVE	0x81020000	进程仍是活动状态
E_PROCESS_NOT_STARTED	0x81030000	进程未开始执行
E_PROCESS_ALREADY_STARTED	0x81040000	进程已经开始执行
E_PROCESS_ALREADY_EXITED	0x81050000	进程已经结束
E_PROCESS_NOT_EXITED	0x81060000	进程未结束
E_THREAD_NOT_ACTIVE	0x81070000	线程未处于活动状态
E_THREAD_STILL_ACTIVE	0x81080000	线程仍处于活动状态
E_THREAD_UNSTARTED	0x81090000	线程未开始执行
E_THREAD_ALREADY_FINISHED	0x810A0000	线程已经完成
E_THREAD_NOT_STOPPED	0x810B0000	线程未停止
E_DOES_NOT_EXIST	0x810C0000	某资源不存在
E_ALREADY_EXIST	0x810D0000	某资源已经存在
E_INVALID_OPTIONS	0x810E0000	选项无效
E_INVALID_OPERATION	0x810F0000	进行无效操作
E_TIMED_OUT	0x81100000	方法超时
E_INTERRUPTED	0x81110000	方法被中断
E_NOT_OWNER	0x81120000	线程并未占有互斥体对象
E_ALREADY_LOCKED	0x81130000	调用线程已经以读者身份占有了读写锁对象
E_INVALID_LOCK	0x81140000	调用线程已经以写者身份占有了读写锁对象
E_NOT_READER	0x81150000	调用线程并未以读者身份占有读写锁对象
E_NOT_WRITER	0x81160000	调用线程并未以写者身份占有读写锁对象
E_NOT_ENOUGH_ADDRESS_SPACE	0x81170000	没有足够的地址空间

E_BAD_FILE_FORMAT	0x81180000	错误的文件格式
E_BAD_EXE_FORMAT	0x81190000	错误的 exe 格式
E_BAD_DLL_FORMAT	0x811A0000	错误的 dll 格式
E_PATH_TOO_LONG	0x811B0000	路径名太长
E_PATH_NOT_FOUND	0x811C0000	路径未找到
E_FILE_NOT_FOUND	0x811D0000	文件未找到
E_NOT_SUPPORTED	0x811E0000	方法不支持该操作
E_IO	0x811F0000	I/O 错误
E_BUFFER_TOO_SMALL	0x81200000	缓存太小
E_THREAD_ABORTED	0x81210000	线程中止
E_SERVICE_NAME_TOO_LONG	0x81220000	服务名太长
E_READER_LOCKS_TOO_MANY	0x81230000	调用线程所获取的读锁个数已达到了 MAXIMUM_OWNED_READER_LOCKS 个
E_ACCESS_DENIED	0x81240000	拒绝访问
E_OUT_OF_MEMORY	0x81250000	内存分配不成功
E_INVALID_ARGUMENT	0x81260000	一个或多个无效参数

注意：NOERROR 的值的定义与 C/C++中 TRUE 的定义相反。

CAR 中提供以下两个宏，使用这两个宏可以通过判断 ECode 返回值得知方法调用是否成功：

宏定义	含义
SUCCEEDED(ec)	判断方法调用是否成功
FAILED(ec)	判断方法调用是否失败

5.3.8 ClassId

CAR 类标识符，用于唯一标识一个特定 CAR 类。

备 注

ClassId 是 CLSID 的扩展，ClassId 结构体中包含 uunm 信息。

在 CAR 中，使用 ClassId 结构体做接口参数时，通常用于创建对象。如果使用 CLSID 做参数，由于 CLSID 结构体中不包含 uunm 信息，在 Elastos 上运行构件时，将无法找到构件所在的位置，无法成功创建对象。

全球唯一标识符

为了确保编写的组件或构件在时间和空间上的唯一性，通常使用标识符来标识接口、类或类别。

GUID（Globally Unique Identifier）：全球唯一标识符，一个 16 字节，128 位的数，它唯一标识一些实体，例如：一个类或一个接口。GUID 的实例有 ClassID、InterfaceId 等。GUID 也叫做 UUID（Universally Unique Identifier）通用唯一标识符。

下表列出的是常用的 GUID：

标识符	说明
ClassID(Class Identifier)	类标识符。CLSID 是一个唯一标识一个特定 CAR 类的 GUID。
InterfaceId(Interface Identifier)	接口标识符。IID 是一个唯一标识一个特定 CAR 接口的 GUID。

在 CAR 中，定义了一个标识符：ClassId，它是 CAR 类标识符，ClassId 是一个唯一标识一个特定 CAR 类的 GUID。ClassId 里面包含 uunm 信息。详细情况请参见 ClassId 文档。

我们系统生成 GUID 的具体算法由函数 GuidFromSeedString 实现，其中 pszSeed 为 in 参数，pGuid 为 out 参数。

要求 module 的 UUID，则用 module 的 uunm 字符串作为参数 pszSeed 传给该函数即可求得。

要求类的 ClassId，则要分两步走，先用"/"把类名和 module 名连接成一个字符串，再把该字符产作为参数 pszSeed 传给 GuidFromSeedString 函数，即可得到 ClassId。

要求接口的 InterfaceId，则要分两步走，先调用 GenIIDSeedString(InterfaceDirEntry *pInterface, char *pszBuf)函数得到一个表示 pInterface 的字符串 pszBuf，再用 pszBuf 作为参数 pszSeed 传给 GuidFromSeedString 函数，即可得到 InterfaceId。

```
int GuidFromSeedString(const char *pszSeed, GUID *pGuid)
{
    char szLongSeed[c_nMaxSeedSize + 1];
    BYTE *p;
    DWORD result[3];
    int n, nEncoder, cEncoded, cPrevEncoded;

    n = strlen(pszSeed);
    if (n > c_nMaxSeedSize) {
        memcpy(szLongSeed, pszSeed, c_nMaxSeedSize);
        szLongSeed[c_nMaxSeedSize] = 0;
        pszSeed = szLongSeed;
        n = c_nMaxSeedSize;
    }
    p = (BYTE *)pGuid;
    p[1] = n;
    nEncoder = 0;
    cPrevEncoded = 0;
```

```
        for (n = 0; n < c_cEncoders; n++) {
            memset(result, 0, 12);
            cEncoded = (*c_encoders[n].fnEncode)(
                            c_encoders[n].pvArg, pszSeed, result);

            assert(cEncoded <= p[1] && "Guid Encoding Size Error");
            if (cEncoded > cPrevEncoded) {
                memcpy((BYTE *)pGuid + 4, result, 12);
                nEncoder = n;
                cPrevEncoded = cEncoded;
            }
            if (cEncoded == p[1]) break;
        }
        *(WORD *)&p[2] = GenerateCheckSum((WORD *)pszSeed, p[1]);

        p[0] = (BYTE)(nEncoder << 5);
        p[0] |= (BYTE)GenerateGuidCheckSum(*pGuid);

        return cPrevEncoded;
    }
```

第六章　CAR 关键字

关键字也称保留字。它是预先定义好的表示符，这些表示符对 CAR 编译程序有着特殊的含义。下面对 CAR 中几个主要关键字的含义和表示方法一一进行介绍。

- 用于定义构件的关键字:

基本关键字	描述
module	用于指定.dll 形式的构件
library	用于指定.lib 形式的构件

- 用于定义接口的关键字：

基本关键字	描述
interface	用于定义或声明一个接口
callbacks	用于定义或声明一个异步回调接口
delegates	用于定义或声明一个同步回调接口

- 用于定义类的关键字：

基本关键字	描述
class	用于定义一个普通构件类
generic	用于定义一个泛类
applet	用于定义一个 applet 类
aspect	用于定义一个方面构件类
context	用于定义一个语境类

- 用于继承的关键字：

基本关键字	描述
inherits	用于表示普通类 class 继承普通类 class
extends	表示接口 interface 继承接口 interface
final	用于修饰普通构件类 class，表示该类不允许被继承
substitutes	用于表示普通类 class 继承泛类 generic

- 用于修饰接口的关键字：

基本关键字	描述
virtual	用于修饰 interface,表示声明虚接口

asynchronous	用于修饰 interface,表明被修饰的 interface 的接口的方法将被异步方式调用
privileged	用于修饰 interface,并只能用于 aspect 中，指定其接口只能在聚合和解聚合的时候可见
filtering	用于修饰 callbacks 接口，表示子类过滤基类所触发的回调事件

- 用于修饰类的关键字：

基本关键字	描述
singleton	用于表示该类只能创建出唯一的对象
aggregates	表示被修饰的构件类创建对象时将自动聚合该属性中列举的方面构件对象
pertainsto	用于修饰语境构件类 context，表示该语境构件类拥有该属性中列举的方面构件类
affiliates	用于修饰方面构件类 aspect，表示被修饰的方面构件类只能被该属性中列举的非方面构件类聚合

- 用于定义数据结构的关键字：

基本关键字	描述
const	用于定义 int 类型的常量
enum	用于定义枚举类型，使用方法与其在 C 语言中用法基本相同
struct	用于定义结构体，其使用方法与在 C 语言中用法基本相同
typedef	用于定义类型，其使用方法与在 C 语言中用法基本相同

- 用于合并构件的关键字：

基本关键字	描述
merge	用于在本构件中合并用到的其它构件(.car 文件)
mergelib	用于在本构件中合并用到的其它构件（.dll 或者.cls 文件）中被当前构件的定义引用到的部分
import	指定本构件中用到的其它构件（.car 文件）
importlib	指定本构件中用到的其它构件（.dll 或者.cls）

- 其它关键字：

基本关键字	描述
constructor	用于在类中定义构造函数
pragma	禁止或允许显示 CAR 编译器给出的警告信息
coalesce	用于在类中修饰 callbacks 接口方法，表示合并同类回调事件

6.1 module

此关键字用于定义构件,例如 ModuleDemo.car 文件中的内容:

```
module www.elastos.com/car/HelloModuleDemo.dll
{
    interface IHello {
        Func([in] Int32 num);
    }

    class CHello {
        interface IHello;
    }
}
```

示例中，module 为关键字，其后是该构件的全球唯一名字(uunm)，包括 URL 路径和 dll 名字，其后的{}内定义了一个构件 ModuleDemo。该构件中定义了一个接口 IInterface 和一个类 CClass。

在一个 CAR 文件中有且只能有一个 module 即一个构件的定义。

构件 dll 名字和最后编译出来的文件名必须一致，与该 CAR 文件的文件名也必须一致，如不指定 uunm，则默认的 dll 名是 car 文件名加.dll 后缀，并且，构件名不能和该构件的类名重复。

不允许一个空的 module 定义，即不含有任何的类，接口和方法。编译器会报错。

module 指定生成 dll 形式的 car 构件，且 module 中必须定义具体的类。

6.2 library

此关键字用于定义构件,例如 LibraryDemo.car 文件中的内容:

```
library www.elastos.com/car/LibraryDemo.lib
{
    interface IFoo {
        Foo([in] Int32 num);
    }

    interface IBar {
        Bar();
    }
}
```

示例中，library 为关键字，其后是该构件的全球唯一名字(uunm)，包括 URL 路径和 dll 名字,其后的{}内定义了一个构件库 ModuleDemo。该构件中定义了一个接口 IInterface 和一个类 CClass。

library 指定生成 lib 形式的 car 构件，可以只定义接口，然后在其它 module 中实现。

6.3 interface

此关键字用于定义接口。例如 InterfaceDemo.car 中：

```
module
{
    interface IInterface1 {
        Foo();
    }
    interface IInterface2 {
        Bar();
    }

    class CClass {
        interface IInterface1;
        interface IInterface2;
    }
}
```

示例构件 InterfaceDemo 中，interface 关键字后面是接口的名字，接口名字后面“{}”中，是该接口的定义。定义了两个接口 IInterface1，IInterface2，接口中分别定义了方法 Foo，Bar。

一个构件中可以定义一个或多个接口，每个接口中可以有一个或多个方法。接口定义主要包括接口方法的声明。一个没有接口方法的接口在现实中没有意义。一个构件中不支持同名的方法。

接口是客户程序和类对象之间的桥梁，接口的更改要求构件和客户程序也必须做出相应的改动，这样做不符合构件化程序的思想。因此，一个接口定义下来并公布之后，该接口中方法的位置、方法中参数的类型及个数都不能再做更改。

接口还可以继承其它接口。例如：

```
module
{
    interface IInterface1 {
        ……
    }
    interface IInterface2 : IInterface1 {
        ……
    }
    ……
}
```

示例构件 InterfaceDemo2 中，接口 IInterface2 继承接口 IInterface1，接口 IInterface2 中就拥有接口 IInterface1 中定义的所有方法。

另外，CAR 中还定义有一个关键字 callback，和 interface 关键字一起用于定义回调接口。相对于回调接口来说，单独使用 interface 关键字定义的接口也称为普通接口。

6.4 callback

此关键字用于定义回调接口。回调接口也是接口的一种，该接口中的每个成员函数代表一个回调事件（Callback）。

6.4.1 回调机制

回调接口中每个成员函数代表一个事件（event）。当特定事情发生时，如定时消息或用户鼠标操作发生时，构件对象产生一个事件，客户程序可以处理这些事件。构件对象中回调接口并不由构件对象实现，而是由客户端的接收器实现。接收器也是构件对象，它除了实现回调接口外，还负责与可连接对象进行通信。当接收器与可连接对象建立连接后，客户程序可将自己实现的事件处理函数注册，把函数指针告诉构件对象，构件对象在条件成熟时激发事件，回调事件处理函数。

6.4.2 callback 的语义及实例讲解

编写 CallbackDemo.car 文件如下：

```
module
{
    interface IFoo {
        Foo();
     }

    callbacks JFooEvent {
       FooEvent ();
     }

    class CFoo {
       interface IFoo;
       callbacks JFooEvent;
    }
}
```

注意：

1.回调接口命名必须以字母“J”开头；回调接口不能继承其它接口，也不能被其它接口继承。

2.回调接口中声明的每个方法代表一个回调事件。

3.Callbacks 接口声明的回调事件的参数属性只能具有 in 属性，delegates 接口方法可以具有 out 属性，但不支持 local 属性。

4.一个构件类不能只含有一个回调接口，至少要有一个普通接口。

此关键字用于定义或声明异步回调接口。回调包括两种：同步回调和异步回调。定义同步回调接口请参见关键字 delegates。上述示例中，定义了普通接口 IFoo 和异步回调接口 JFooEvent，回调接口中方法 FooEvent()就是一个回调事件。编译该 car 文件，并在自动生成的代码框架中添加如下代码：

CFoo.cpp 文件：

```
#include "CFoo.h"
#include "_CFoo.cpp"

ECode CFoo::Foo()
{
    // TODO: Add your code here
    CConsole::WriteLine(L"Common Function:Foo");
    Callback::FooEvent();
    return NOERROR;
}
```

客户端代码如下：

```
#include "callbackDemo.h"
using namespace Elastos;

ECode OnFooEvent(PVoid userData, PInterface pSender)
{
    CConsole::WriteLine(L"callback event:FooEvent");
    CFoo::RemoveAllCallbacks(pSender);
    CApplet::Finish(AppletFinish_ASAP);
    return NOERROR;
}

ECode ElastosMain(const BufferOf<WString>& args)
{
    ECode ec;
    IFoo* pIFoo;

    ec = CFoo::New(&pIFoo);
    if (FAILED(ec)) {
        return ec;
    }
```

```
        pIFoo->Foo();

        ec = CFoo::AddFooEventCallback(pIFoo, OnFooEvent, NULL);
        if (FAILED(ec))
        {
            pIFoo->Release();
            return ec;
        }

        pIFoo->Foo();

        CObject::ReleaseAtThreadQuit(pIFoo);
        return NOERROR;
    }
```

运行结果如下：

```
Common Function:Foo
Common Function:Foo
callback event:FooEvent
```

在 server 端所有 callback 接口的方法已经由工具生成的代码实现了，只需在需要回调的地方调用 Server 类对应的成员方法，Callback 机制会自动调用所有已注册在该回调事件的回调处理函数。CallbackDemo.car 文件经过 CAR 工具编译后，自动生成 AddXxxCallback 和 RemoveXxxCallback（Xxx 代表回调事件名）函数提供给客户端使用来注册和注销事件处理函数。

回调处理函数的第一个参数是 Pvoid 类型的用户参数，第二个参数是 PInterface 类型的一个接口指针类型，如：

```
ECode OnFooEvent(PVoid userData, PInterface pSender)
```

以下结合上述例子具体介绍客户与可连接对象从建立连接到断开连接的整个过程。

1、建立连接并注册回调处理函数

在客户端第一次注册回调处理函数时，注册函数 AddXxxCallback 会创建一个和可连接对象有关联的接收器，并把回调函数注册到该接收器中，之后每一次注册其它回调函数时，只要找到这个接收器，然后直接把回调函数注册到该接收器中就行了。

客户注册了事件函数后，接收器对象保存了该函数指针。例如：

```
ec = CFoo::AddFooEventCallback(pIFoo, OnFooEvent, NULL);
```

2、事件激发

在上述例子里，客户端通过 pIFoo 指针调用入接口函数 Foo，而在可连接对象里，Foo 方法触发 FooEvent 事件，通过其所保存的回调接口 JFooEvent 的接口指针调用 FooEvent 方法，这时就进入了接收器对象，客户对事件 FooEvent 注册了自己的实现方法 OnFooEvent，接收器对象会检测到客户的函数指针并调用之，至此，事件激发过程结束。

3、事件注销

客户可以通过 RemoveXxxCallback（Xxx 代表回调事件名）函数来注销自己曾注册的事件。这个函数的参数和 AddXxxCallback 函数参数一样；一旦客户调用这个函数成功，那么客户注册的事件函数指针就会从接收器对象里去掉，事件激发时就只能调用到接收器对该事件的默认实现（只是简单地返回 NOERROR）。

4、断开连接

当 proxy(或本地 server)release 到 0 时接收器会自动中止客户和可连接对象的双向通信，这时接收器对象和可连接对象不再有任何关系，可连接对象不再保存接收器对象实现的回调接口指针，也就不能触发回调接口的事件了。最后接收器自己也会自动释放掉。

6.5 delegates

此关键字用于定义或声明同步回调接口。例如 DelegatesDemo.car 文件中：

```
module
{
    interface IFoo {
        Foo();
    }
    delegates JFooErrands
    {
        FooErrand();
    }

    class CFoo {
        interface IFoo;
        delegates JFooErrands;
    }
}
```

使用 delegates 时要注意如下几点说明：

1. 回调接口命名必须以字母“J”开头；回调接口不能继承其它接口，也不能被其它接口继承。

2. Delegates 接口方法可以具有 in 属性和 out 属性,但不支持 local 属性。

3. Delegates 接口方法只能被注册一次，再次注册会返回失败。

4. delegates 表示同步调用，直到 client 端回调函数结束才返回。

5. 如果没有注册 Delegate，调用 Delegate::FooErrand()会返回 E_NO_DELEGATE_REGISTERED，如果已调用过 CFoo::AssignFooErrandDelegate 注册 FooErrand，再次注册时会返回 E_DELEGATE_ALREADY_REGISTERED。

6. 调用 delegate 回调函数的几种方式：

```
Delegate::FooErrand();
Delegate::FooErrandWithTimeout(1000);
Delegate::FooErrandWithPriorityAndTimeout(CallbackPriority_Highest,
1000);
```

7. 激发 delegate 时可以设置 timeout，这个 timeout 目前只支持 16bits，以毫秒为单位，即最多支持 65 秒左右。

编译上述 delegateDemo.car 文件，在生成的文件中实现 Foo 方法如下：

```
ECode CFoo::Foo()
{
    CConsole::WriteLine("CFoo::Foo()");
    Delegate::FooErrands();
    CConsole::WriteLine("delegates call finish...");
    return NOERROR;
}
```

这里我们触发了事件 FooErrands，其在客户端实现回调函数 FooErrands 并注册，当我们在客户端调用 **Foo** 方法的时候，会触发客户端注册的回调事件，而且会等回调事件结束才返回。

例如而在客户端做如下实现：

```
#include "delegateDemo.h"
using namespace Elastos;

ECode AtFooErrand(PVoid pUserdata, PInterface pSender)
{
    CConsole::WriteLine("AtFooErrand");
    return NOERROR;

}

ECode ElastosMain(const BufferOf<WString> & args)
{
    IFoo *pFoo;
    ECode ec;
    ec = CFoo::New(&pFoo);
    if (FAILED(ec)) return ec;

    CFoo::AssignFooErrandDelegate(pFoo, &AtFooErrand, NULL);
    pFoo->Foo();
    CFoo::RevokeFooErrandDelegate(pFoo, &AtFooErrand, NULL);

    pFoo->Release();
    CApplet::Finish(AppletFinish_ASAP);
    return NOERROR;
}
```

运行结果为

```
CFoo::Foo()
AtFooErrand
delegates call finish...
```

6.6 class

此关键字用于定义普通构件类，例如 ClassDemo.car 文件中：

```
module
{
    interface IFoo {
        Foo();
    }
    class CClass {
        interface IFoo;
    }
}
```

示例构件 ClasstDemo 中，定义了一个类 CClass，class 关键字后面是类的名字。类名字后面“{}”中，是该类的定义。类定义中主要包括该类提供的接口。因为类对象中能够和外界打交道的只有接口，所以一个没有接口的类对象的存在，是没有意义的。

在一个 CAR 文件中，可以定义一个或多个类。例如 ClassDemo2.car 文件中：

```
module
{
    interface IFoo {
        Foo();
    }
    interface IBar {
        Bar();
    }

    class CClass1 {
        interface IFoo;
        interface IBar;
    }
    class CClass2 {
        interface IFoo;
    }
}
```

示例构件 ClassDemo2 中，定义了两个接口：接口 IFoo 和接口 IBar；定义了两个类：类 CClass1 和类 CClass2。类 CClass1 中提供接口 IFoo 和接口 IBar，类 CClass2 中提供接口 IFoo。类 CClass1 中和 CClass2 中都提供了接口 IFoo，该接口在两个类中可以有各自不同的实现。

car 文件中，一个类中最多包含 32 个接口，如果有 callback 接口，由于 sink 的原因，一个 callback 接口充当两个接口。接口太多时，carc 编译器会报错

一个类中没有接口的定义编译器会报错。

car 中类的定义包括五种，除了普通类外，另外四种分别是：aspect 类、context 类、generic 类和 applet 类。 普通类可以使用关键字 inherits 继承其他普通类，但被继承的基类必须包含虚接口 virtual。如果要继承由 generic 关键字定义的泛类，需使用关键字 substitutes。 例如在 ClassDemo3.car 文件中：

```
module
{
    interface IFoo{
            Foo();
    }
    interface IBar{
            Bar();
    }

    class CFoo{
        virtual interface IFoo;
    }
    class CBar inherits CFoo{
        interface IFoo;
        interface IBar;
    }
}
```

上述示例定义了两个接口 IFoo 和 IBar，定义了两个类 CFoo 和 CBar。类 CFoo 声明了虚接口 IFoo，类 CBar 继承自 CFoo，CFoo 和 CBar 都声明了接口 IFoo，该接口在两个类中可以具有各自不同的实现。

6.7 generic

此关键字的基本意图是给类定一些类别，同时提供开发者一个该类别下所有类的默认构件接口 New。

6.7.1 generic 机制

我们一般由 CAR 里面定义的类别，用 CAR 编译器生成 C++代码的实现框架，同时生成“类厂”的实现。用户一般没有办法更改“类厂”的实现，除非他用“generic”来定义一个“泛型”的类。

泛型技术源于需要用一种一般的方法处理对象各种可能的类型，而不需要关心他们具体的类型。泛型编程和面向对象编程不同，它并不要求你通过额外的间接层来调用函数，它让你编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。

在 CAR 中，generic 被称为“泛类”，定义这样一种类的目的是为了抽象出各个一般类所具有的共同的类或接口特性，并且在定义这个泛类的 New 函数时，可以根据硬件条件的变化自动选择创建不同的具体的类。只要这些具体的类都是从这个泛类继承而来，创建这些具体的类时直接使用泛类的 New 函数就能完成，客户端不需要关心到底是创建哪个具体的类。

6.7.2 generic 语义及实例讲解

编写 GenericDemo.car 文件如下：

```
module
{
    interface IMouse{
        Connect();
        Disconnect();
    }
    interface IUsbmouse{
        Ummethod();
    }
    interface IPs2mouse{
        Pmmethod();
    }

    generic GMouse{
        interface IMouse;
    }

    class CUsbmouse substitutes  GMouse{
        interface IMouse;
        interface IUsbmouse;
    }

    class CPs2mouse substitutes GMouse{
        interface IMouse;
        interface IPs2mouse;
    }
}
```

使用说明：

1.泛类的类名必须以字母“G”开头。

2.泛类可以声明普通接口和回调接口，并支持回调函数的注册与注销。注意：aggregates 不能修饰 generic。

3.继承泛类的子类必须实现泛类声明的所有接口。

4.泛类创建对象时,会调用其 New 函数,至于如何根据应用逻辑来创建不同的子类对象,需要用户在 New 函数里自己手动添加。

此例中定义一个 generic 类别 GMouse，提供接口 IMouse。

类 CUsbmouse 和 CPs2mouse 都继承类别 GMouse，所以这两个类也都提供接口 IMouse，同时，它们分别提供各自的接口 IUsbmouse 和 IPs2mouse。

编译 genericDemo.car 文件以后会生成文件框架 GMouse.cpp，CUsbmouse.h，CUsbmouse.cpp，CPs2mouse.h，CPs2mouse.cpp。

CUsbmouse.h 文件如下：

```
#ifndef __CUSBMOUSE_H__
#define __CUSBMOUSE_H__

#include "_CUsbmouse.h"

CarClass(CUsbmouse)
{
public:
    CARAPI Connect();
    CARAPI Disconnect();
    CARAPI Ummethod();

private:
    // TODO: Add your private member variables here.
};

#endif // __CUSBMOUSE_H__
```

CUsbmouse.cpp 文件如下：

```
#include "CUsbmouse.h"
#include "_CUsbmouse.cpp"

ECode CUsbmouse::Connect()
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}

ECode CUsbmouse::Disconnect()
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}
```

```
ECode CUsbmouse::Ummethod()
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}
```

添加代码后的 CUsbmouse.cpp 文件如下：

```
#include "CUsbmouse.h"
#include "_CUsbmouse.cpp"

ECode CUsbmouse::Connect()
{
    printf("Connected\n");
    return NOERROR;}

ECode CUsbmouse::Disconnect()
{
    printf("Disconnected\n");
    return NOERROR;
}
ECode CUsbmouse::Ummethod()
{
    printf("Use usbmouse\n");
    return NOERROR;
}
```

CPs2mouse.h 和 CPs2mouse.cpp 类似，在此不列出程序框架。

GMouse.cpp 文件如下：

```
#include "GMouse.h"
#include "_GMouse.cpp"

ECode GMouse::New(
    /*[out]*/ IMouse **pIMouse)
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}
```

添加代码后的 GMouse.cpp 文件如下：

```
#include "GMouse.h"
#include "_GMouse.cpp"
#include "CPs2mouse.h"
```

```
#include "CUsbmouse.h"

#define MOUSE_TYPE_PS2      0x01
#define MOUSE_TYPE_USB      0x02

Int32 CheckMouseType()
{
    return MOUSE_TYPE_PS2;//此处决定创建的是 Ps2mouse,也可以改成 Usbmouse。
}

ECode GMouse::New(
    /*[out]*/ IMouse **ppIMouse)
{
    // TODO: Add your code here
    ECode ec;
    IMouse * pIMouse;

    if (CheckMouseType() == MOUSE_TYPE_PS2) {
        ec = CPs2mouse::New(&pIMouse);
        if (FAILED(ec)) return ec;
    }
    else if (CheckMouseType() == MOUSE_TYPE_USB) {
        ec = CUsbmouse::New(&pIMouse);
        if (FAILED(ec)) return ec;
    }

    *ppIMouse = pIMouse;
    return NOERROR;
}
```

Client 端的代码如下：

```
#include "genericDemo.h"
using namespace Elastos;

int main(int argc, char* argv[])
{
    // I need to get a mouse handle...
    IMouse* pMouse = NULL;

    // but I don't know MouseType on my pc, so...
    ECode ec = GMouse::New(&pMouse);
    if (FAILED(ec)) {
```

```
            CConsole::WriteLine("Sorry, No mouse on your pc!");
            return -1;
        }

        // OK, connect my mouse
        pMouse->Connect();

        // do something...


        // well, disconnect my mouse
        pMouse->Disconnect();

        // ready to exit.
        if (pMouse) pMouse->Release();

        return 0;
    }
```

运行 Client.exe 结果如下：

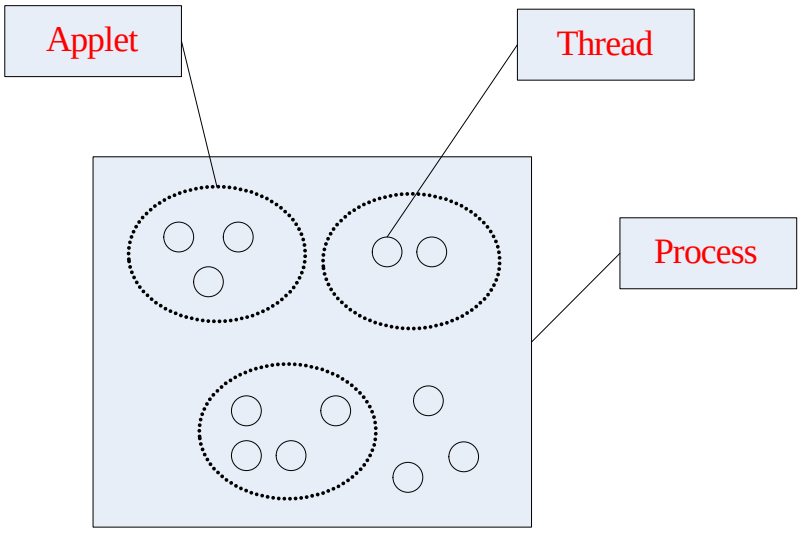
```
Done Ps2 mouse connected!
Ps2 mouse Disconnected!
```

6.8 applet

Applet，顾名思义就是“应用”，比如一个计算器，一个记事本等等。它是 Elastos2.1 操作系统所支持程序模型的一个重要概念和 CAR 技术的重要组成部分，之前我们基于 CAR 编程模型的应用都在进程内不同的线程中独立运行，一个应用可能有一个线程，也可能有若干个线程，进程内这些线程没有什么差别，他们共享相同的地址空间和数据，应用之间没有明确的界限。有了 applet 的概念以后，每次启动一个 applet 它会创建一个独立的属于它的 applet 主线程，在这个线程的局数存储（TLS，Thread Local Stogage）上会有这个 applet 的属性数据，有别于其他 applet，当这个主线程需要创建子线程时，这些属性数据也会被子线程继承，这样，凭借 applet 属性就可以明确地划分出进程内不同 applet 的边界，我们可以判断出一个调用是在 applet 内部，还是跨 applet 边界的。“应用”这个逻辑的、业务上的概念在编程时有了对应的实现手段。

Applet 是一种新的 class 类型，使用 T 字头命名。它是在进程内一个相对独立的小应用，具有自己的消息处理线程，它所注册的回调和它的子线程所注册的回调都由 applet 的回调线程处理。

进程，线程，Applet 和 ElastosMain 的关系如下图所示：



这里我们需要注意的几点是：

1. Applet 有自己的“主线程”
2. Applet 有自己的 Main 方法，并是在自己的主线程里执行 Main 方法
3. 用户的 Main 方法类似于 ElastosMain，如果 Applet::Main 返回 NOERROR，Applet 的主线程也进入消息循环
4. Applet 的主线程就是自己的消息处理线程
5. 所有 Applet 的子线程都属于该 Applet
6. 所有 Applet 的子线程都继承 Applet 的回调线程
7. 进程可以被看作一个大的 Applet，它的子线程都使用进程的消息处理线程（即主线程）
8. 无论谁创建的 Applet，Applet 都使用它自己的消息处理线程
9. Applet 内创建的子线程也都使用该 Applet 的消息处理线程
10. Singleton 是个例外，它和它内部使用的 callback 是由进程的主线程来处理，因为它的生命周期可能比它所在的 Applet 还要长
11. 在创建应用时，将 Applet 指针放到 TLS 上的 Domain 槽里，以便区分应用。
12. 图形内部还是只能通过伪 Domain 区分应用，因此创建新线程时复制 TLS 上的 Domain 槽。
13. 图形应用退出时退掉当前应用消息循环。
14. 当 Release 一个 Applet 时退掉被释放 Applet 消息循环。

有关 Applet 生成代码的操作中将堆内存委托给 ThreadQuitRoutine，在线程退出时释放。

Applet 示例：

```
module
{
    interface ITimer {
        SetInterval([in] Int32 interval);
    }
```

```
    interface ITimerEvent {
        Alarm();
    }
    class CTimer {
        interface ITimer;
        callbacks JTimerEvent;
    }

    applet TTita {

    }
}
```

使用说明：

1.applet 类名必须以字母“T”开头。

2.applet 类不能声明构造函数 constructor。

编译上述 titaDemo.car 文件，修改生成的 CTimer.cpp 文件如下：

```
#include "CTimer.h"
#include "_CTimer.cpp"

Int32 g_nInterval = 0;

ECode CTimer::SetInterval(
    /* [in] */ Int32 interval)
{
    g_nInterval = interval;
    return NOERROR;
}
```

修改 TTita.cpp 文件如下，其中 TimerRoutine 是 timer 的一个后台线程，每秒钟激发一次 timer 的 Alarm 事件。激发十次。：

```
#include "TTita.h"
#include "_TTita.cpp"
#include "CTimer.h"

ECode OnAlarm(PVoid userData, PInterface pSender)
{
    static int nCount = 0;
    CConsole::Write("Tita.");
    CConsole::WriteLine(++nCount);
```

```
    if (nCount >= 10) {
        CTimer::RemoveAllCallbacks(pSender);
        CApplet::Finish(AppletFinish_ASAP);
    }
    return NOERROR;
}

ECode TimerRoutine(PVoid pParam)
{
    CTimer *pCTimer = (CTimer *)pParam;
    int i;

    for(i = 0; i < 10; i++) {
        pCTimer->Callback::Alarm();
    }

    return NOERROR;
}

ECode TTita::Main(
    /* [in] */ const BufferOf<WString> & wargs)
{
    // TODO: Add your code here
    ITimer *pTimer;
    ECode ec = CTimer::New(&pTimer);
    if (FAILED(ec))   return ec;
    ec = CTimer::AddAlarmCallback(pTimer, &OnAlarm, NULL);

    pTimer->SetInterval(1000);

    CObject::ReleaseAtThreadQuit(pTimer);

    IThread *pThread;
    ec = CThread::New(TimerRoutine, pTimer, 0, &pThread);
    if (FAILED(ec))  return ec;

    pThread->Start();
    pThread->Join(INFINITE, NULL);
    pThread->Release();

    return NOERROR;
```

```
}
```

Applet 的 Main 方法作为 Applet 的入口点，相当于 crt 的 main 函数，用户可以在 Main 方法里初始化程序，调用其它构件，或者创建新线程等等。修改使用 titaDemo.dll 的客户端代码如下：

```
#include "titaDemo.h"
using namespace Elastos;


ECode ElastosMain(const BufferOf<WString> & args)
{
    IApplet* pApplet;
    ECode ec = TTita::New(&pApplet); //创建对象
    if (FAILED(ec)) return ec;

    ec = pApplet->Start(args);        //调用 Start 方法启动 applet
    if (FAILED(ec)) {
        pApplet->Release();
        return ec;
    }

    //等待 applet 结束，20 秒超时，若超时则主动终止 applet

    WaitResult wr;
    pApplet->WaitUntilFinished(20000, &wr);
    if (wr == WaitResult_TimedOut) {
        pApplet->Finish(AppletFinish_ASAP);
        CApplet::Finish(AppletFinish_ASAP);
    }

    CObject::ReleaseAtThreadQuit(pApplet);
    CApplet::Finish(AppletFinish_ASAP);

    return NOERROR;
}
```

运行结果如下:

```
Tita.1
Tita.2
Tita.3
Tita.4
Tita.5
Tita.6
```

```
Tita.7
Tita.8
Tita.9
Tita.10
```

6.9 aspect

此关键字用来定义方面构件类。

6.9.1 AOP(面向方面编程)概述

在 COM 里，聚合体现了构件软件的重用性，聚合的目的就是使两个构件对象特征成为一个构件对象(外部对象)的特征，而另外一个构件对象对客户透明. CAR 的 AOP 机制使用户能够在完全不用修改源代码的情况下简单而方便的动态聚合两个 CAR 构件类，从而生成一个具有两个 CAR 构件类所有接口实现的新构件类。CAR 的 AOP 技术是由 aspect, 动态聚合，语境(context)组成。

6.9.2 aspect(方面)

aspect 是一种特殊的构件类实现，aspect 对象的特征是可以被其它构件对象聚合，该构件类必须实现 IAspect 接口，aspect 对象就是实现了 IAspect 接口的构件对象。

例如 AspectDemo.car 文件中：

```
module
{
    interface IFoo{
        FooHello();
    }

    interface IBar {
        BarHello();
    }
    aspect AFoo {
        interface IFoo;
    }
    class CBar {
        interface IBar;
    }
}
```

示例 AspectDemo 构件中，aspect 关键字后面是 aspect 构件类类名，类名后面“{}”中，是该类的定义。AFoo 被定义为 apsect 的构件类，类中又定义了 IFoo 接口。

使用说明：

1. CAR 构件技术里只有 aspect 构件对象可以被聚合。
2. 不允许一个 aspect 构件类包含回调接口。
3. aspect 对象可以被其他构件对象聚合，但是它不可以聚合其它 aspect 对象。
4. aspect 类中不允许定义带参数的 constructor,也就是 aspect 对象不可以被单独的 New 出来。

aspect 构件类不但会实现自身定义的接口，还会实现 IAspect 接口。CAR 自动代码工具会自动生成这部分代码。跟普通的 CAR 接口不一样，普通的 CAR 接口都继承于 IObject， 而 IAspect 接口不从 IObject 继承，除了方法名与 IObject 不一样外，IAspect 的接口其它定义与 IObject 完全相同。（IObject 定义在动态聚合中介绍）

在本示例中，AFoo 构件类会实现 IFoo 接口。

编译 car 文件之后，自动代码生成框架会生成 AFoo.cpp, CBar.cpp，分别改写如下：

AFoo.cpp：

```
#include "AFoo.h"
#include "_AFoo.cpp"

ECode AFoo::FooHello()
{
     CConsole::WriteLine("Hello, I am from AFoo!");
     return NOERROR;
}
```

CBar.cpp:

```
#include "CBar.h"
#include "_CBar.cpp"

ECode CBar::BarHello()
{
    CConsole::WriteLine("Hello, I am from CBar!\n");
    return NOERROR;
}
```

6.9.3 动态聚合

CAR 构件的每个接口都是由 IObject 继承而来的，CAR 中 IObject 接口的定义如下；

```
IObject {
    virtual CARAPI QueryInterface(              //
            /* [in] */ RIID riid,
            /* [out] */ PObject *ppObject) = 0;

    virtual CARAPI_(ULONG) AddRef() = 0;              //增加引用计数

    virtual CARAPI_(ULONG) Release() = 0;              //减少引用计数
```

```
    virtual CARAPI Aggregate(          //动态聚合，一般用户不会直接调用该方法
            /* [in] */ AggregateType type,
            /* [in] */ PObject pObject) = 0;
};
```

以上每个方法的实现都在定义接口的类里，由自动代码生成框架生成。

动态聚合是通过 IObject 的 Aggregate 方法来完成的，因此构件编写者定义的每个构件对象都具有聚合其他 aspect 对象的能力。一般实现动态聚合都通过方面构件类提供的静态方法 Attach(IObject* pObj)和 Detach(IObject* pObj)方法来完成。

如有一个构件对象 A(构件类为 CA)和一个 aspect 对象 B(构件类为 AB)，对于构件 A 的编写者来说，在构件对象 A 里如果要聚合 aspect 对象 B，那么只要通过如下方法就可完成聚合任务：

```
ec = AAspect::Attach(pIAB);
```

Attach 函数调用成功后，那么对象 B 里保存了外部对象 A 的指针，对象 A 则保存了内部对象 B 的 IAspect 指针，同时内部对象 B 的引用计数也转嫁到了外部对象 A 上了。

在方面构件类 AAspect 中，静态函数 Attach 声明如下：

```
static ECode Attach(
        /*[in]*/ IObject* pObj)
```

其中：pAggregator 为外部对象指针，pAspect 为内部对象(aspect 对象指针)。

1.实例讲解

以上面的 AspectDemo 为例，编写客户端程序 client.cpp 如下：

```
#include "AspectDemo.h"
using namespace Elastos;

ECode ElastosMain(const BufferOf<WString> & args)
{
    IFoo *pFoo;
    ECode ec;
    IBar *pBar;

    ec = CBar::New(&pBar);
    if (FAILED(ec)) return ec;

    pBar->BarHello();

    ec = AFoo::Attach(pBar);
    if (FAILED(ec)) return ec;

    pFoo = IFoo::Probe(pBar);
    if (NULL == pFoo) return E_NO_INTERFACE;
```

```
        pFoo->FooHello();
        CObject::ReleaseAtThreadQuit(pBar);

        CProcess::Exit(0);
        return NOERROR;
    }
```

编译运行程序之后的结果为：

```
Hello, I am from CBar!
Hello, I am from AFoo!
```

2.多面聚合

上面我们介绍的只是两个对象的聚合，普通构件对象只聚合了一个 aspect 对象，但实际上，在面向方面编程时，往往需要一个对象聚合多个 aspect 对象，这就是多面聚合。在完成多面聚合时，实现上并没有多大的变化，就是创建多个 aspect 对象，多次调用 Attach 方法使一个对象聚合多个 aspect 对象。在此就不累述。

6.10 context

6.10.1 什么是语境

语境是对象运行时的环境，一个对象如果进入了语境，那么该对象将具有此语境的特征，一旦对象离开了语境，环境特征就会失去（但该对象很有可能又进入了另外一个语境，拥有新的环境特征）。这里环境特征就是方面 aspect.

语境是基于动态聚合实现的一种技术，所谓的具有环境特征或失去环境特征，就是语境会为对象动态的聚合或拆卸聚合一个或多个 aspect 对象。这些 aspect 对象是语境在 car 文件中定义时指定的。

6.10.2 context 语义及实例讲解

在 CAR 中，实现了语境技术，通过系统语境构件类 CContext 来实现普通类对象 CClass 进出普通语境类对象 KContext，以及对已经进入 KContext 的多个普通类对象进行管理，并实现了一些通用的接口。

在 CAR 中，用户定义的 KContext 也是一个构件类，它所具有的语境特征由其属性 aspect 来决定，并且 context 一般情况下要带属性 aspect。在编写 CAR 文件时，context 作为关键字用来定义一个普通语境构件类。普通语境构件类的名字必须以 K 打头。例如：

```
module
{
    interface IChild {
        Play();
```

```
            GetName([out] WStringBuf<10> nameBuf);
        }

        class CChild {
            constructor([in] WString name);
            interface IChild;
        }

        interface IStudent {
            Study();
            GetID([out] Int32* pId);
        }

        interface IStudentAdmin {
            SetID([in] Int32 id);
        }

        aspect AStudent {
            interface IStudent;
            privileged interface IStudentAdmin;
        }

        interface ISchool {
            Open();
        }

        context KSchool pertainsto AStudent {
            interface ISchool;
        }
    }
```

在该示例中定义了一个方面构件类（AStudent）和一个普通语境构件类（KSchool）。KSchool 语境拥有特征 AStudent。当一个 CChild 对象被创建时，他是一个“小孩”，具有小孩的行为“玩”（Play），而把“小孩”送到学校，他就变成了学生，具有了学生的行为“学习”（Study），而当他从学校毕业，在进入新的环境（语境 context）前他又变成了“小孩”。

换成代码描述就是：一个构件对象（CChild）被创建，它只具有 Play()方法，在它进入了一个语境（KSchool），那么该对象会聚合语境的特征（AStudent），也就是聚合了语境属性里的 aspect 对象（AStudent），从而具有了 Study()方法；如果该对象离开此语境，那么会拆卸聚合该语境属性里的 aspect 对象。

服务端

对于每个 context 构件类，可重载如下几个函数来满足它的对象进行控制:

virtual CARAPI OnObjectEntering(PObject pObj) // 对象进入前调用，可以在这里检查对象是否满足要求。比如是否达到入学成绩或年龄。

virtual CARAPI OnObjectEntered(PObject pObj) // 对象进入后（已经聚合上 context 定义的 aspects）调用。比如为学生分班级和学号。

virtual CARAPI OnObjectLeaving(PObject pObj) // 对象离开前调用，context 可以在这里控制是否允许对象离开。比如判断其是否毕业成绩达标。

virtual CARAPI OnObjectLeft(PObject pObj) // 对象离开后调用，在这里做善后工作。比如要张榜公告一下毕业名单之类的。

继续上面的“小孩”“学生”和“学校”的例子，

KSchool.cpp 对应语境（context）KSchool 的实现代码

```
#include "KSchool.h"
#include "_KSchool.cpp"
#include <stdio.h>

ECode KSchool::Teach()
{ // 学校的行为“教学”
    printf("Teaching ...\n");
    return NOERROR;
}
ECode KSchool::OnObjectEntering(IObject * pObj)
{ // 对象进入学校之前
    printf("OnObjectEntering()... \n");
    return NOERROR;
}
ECode KSchool::OnObjectEntered(IObject * pObj)
{ // 进入以后为学生分配学号，以及输出提示信息
    static int NO = 10;
    IStudent * pStudent = NULL;
    IChild * pChild = NULL;
    AStringBuf_<12> aName;
    ECode ec;
    NO++;
    ec = IStudent::Query(pObj, &pStudent);
    if(FAILED(ec)) {
        printf("The IStudent interface couldn't be queried..\n");
        goto EXIT;
    }
    ec = IChild::Query(pObj, &pChild); // 已经聚合成功了，
```

```
                                                    // 两个接口都可以被 QI
        if(FAILED(ec)) {
            printf("The IChild interface couldn't be queried.\n");
            goto EXIT;
        }
        pStudent->SetNO(NO);
        pChild->GetName(aName);
        printf("%s entered school and student number is NO.%d ...\n", (char*)aName,
NO);
    EXIT:
        if(pStudent)
            pStudent->Release();
        if(pChild)
            pChild->Release();
        return NOERROR;
    }
    ECode KSchool::OnObjectLeaving(IObject * pObj)
    { // 离开学校之前
        printf("OnObjectLeaving()... \n");
        return NOERROR;
    }

    ECode KSchool::OnObjectLeft(IObject * pObj)
    { // 已经从学校毕业
        IChild * pChild = NULL;
        AStringBuf_<12> aName;
        ECode ec;

        ec = IChild::Query(pObj, &pChild); // 只能 QI 到 IChild 接口
        if(FAILED(ec)) {
            printf("The Child interface couldn't be queried.\n");
            goto EXIT;
        }
        pChild->GetName(aName);
        printf("%s graduated from school...\n", (char*)aName);

    EXIT:
        if(pChild)
            pChild->Release();
        return NOERROR;
    }
```

对于 aspect 方面对象 AStudent，我们希望在它被自动聚合时也能够做一些相应的工作，编写 AStudent.cpp 如下：

```
#include "AStudent.h"
#include "_AStudent.cpp"
#include <stdio.h>

ECode AStudent::Study()
{ // 学习的行为
    IChild * pChild = NULL;
    AStringBuf_<12> aName;
    ECode ec;

    ec = IChild::Query(this, &pChild); // 获取 IChild 接口
    if(FAILED(ec)) {
        printf("The Child interface couldn't be queried.\n");
        goto EXIT;
    }
    pChild->GetName(aName); // 获取学生的姓名
    printf("NO.%d, %s is studying ... \n", m_nStudentNO, (char*)aName);
EXIT:
    if(pChild)
        pChild->Release();
    return NOERROR;
}
ECode AStudent:: OnAspectAttaching(IObject * pOuter)
{ // 被聚合时
    printf("AStudent::OnAspectAttaching()...\n");
    return NOERROR;
}
ECode AStudent:: OnAspectDetaching(IObject * pOuter)
{ // 被拆卸聚合时
    printf("AStudent::OnAspectDetaching()...\n");
    return NOERROR;
}
ECode AStudent::SetNO(Int32 no)
{
    m_nStudentNO = no;
    return NOERROR;
}
ECode AStudent::GetNO(Int32* pNo)
```

```
{
    *pNo = m_nStudentNO;
    return NOERROR;
}
```

最后，实现一个普通的构件对象 CChild，希望这个构件对象进入 KSchool 这个 context 的时候能够自动的聚合 AStudent 这个方面对象，并在聚合了这个对象之后打印出一些信息：

```
#include "CChild.h"
#include "_CChild.cpp"
#include <stdio.h>

ECode CChild::constructor(AString name)
{
    strcpy(m_szName, name);
    return NOERROR;
}
ECode CChild::Play()
{
    printf("%s is playing ... \n", m_szName);
    return NOERROR;
}
ECode CChild::GetName(AStringBuf name)
{
    strcpy(name, m_szName);
    return NOERROR;
}
ECode CChild:: OnAspectAttaching(IObject* pAspect)
{// 聚合成功后自动调用
    printf("CChild::OnAspectAttaching()...\n");
    return NOERROR;
}
ECode CChild:: OnAspectDetaching(IObject* pAspect)
{// 拆卸聚合成功后自动调用
    printf("CChild::OnAspectDetaching()...\n");
    return NOERROR;
}
```

客户端

CAR 构件库提供系统语境构件 context.lib 来实现普通类对象 CClass 进出普通语境类对象 KContext，以及对已经进入 KContext 的多个普通类对象进行管理，并实现了一些通用的接口。其中定义了一个接口，包含了以下一组方法：

```
AddItem(PObject pObject);  //向 KContext 中添加普通类对象
RemoveItem(PObject pObject); //从 KContext 中移除普通类对象
GetAllItems(IObjectEnumerator **enumerator); //获得 KContext 中普通类对象的枚举
RemoveAllItems(); //获得 KContext 中普通类对象的个数
```

其中 pObject 为进入或离开语境的普通类构件对象指针。

AddItem 函数完成的功能是：由语境创建其特征实例(aspect 对象)，而语境进入者 pObject 动态聚合这些 aspect 对象。

RemoveItem 实现的功能是对象进入者 pObject 拆卸聚合语境的特征。

当客户端定义的构件对象调用这两个方法，进入一个语境，就会自动聚合 server 端 KContext 构件类属性 aspect 中所指定要聚合的 AAspect 对象。离开这个语境时会自动拆卸聚合。

具体客户端的实现如下：

```
#include <stdio.h>
#import <ContextDemo.dll>

int main(int argc, char** argv)
{
    IStudent * pStudent = NULL;
    IChild * pChild = NULL;
    IContext* pMyCxt = NULL;

    ECode ec = NOERROR;

    ec = CChild::New("LanLan", &pChild);
    if(FAILED(ec)) {
        printf("The CChild object couldn't be constructed.\n");
        goto Exit;
    }
    pChild->Play();

    ec = KSchool::New(&pMyCxt);
    if(FAILED(ec)) {
        printf("The KSchool object couldn't be constructed.\n");
        goto Exit;
    }

    ec = pMyCxt->AddItem(pChild);
```

```
        if (FAILED(ec)) {
            printf("The child failed to enter the school. Error, ec = %x\n", ec);
            goto Exit;
        }

        ec = IStudent::Query(pChild, &pStudent);
        if (FAILED(ec)) {
            printf("The student interface couldn't be queried.Error, ec = %x\n",
ec);
            goto Exit;
        }

        pStudent->Study();

        ec = pMyCxt->RemoveItem(pChild);
        if (FAILED(ec)) {
            printf("The child failed to leave the school. Error, ec = %x\n", ec);
            goto Exit;
        }

    Exit:
        if(pStudent)
            pStudent->Release();
        if(pChild)
            pChild->Release();
        if (pMyCxt) {
            pMyCxt->Release();
        }
        return 0;
    }
```

编译后运行的结果为：

```
LanLan is playing ...
OnObjectEntering()...
CChild::OnAspectAttaching()...
AStudent::OnAspectAttaching()...
LanLan entered school and student number is NO.11 ...
NO.11, LanLan is studying ...
OnObjectLeaving()...
CChild::OnAspectDetaching()...
AStudent::OnAspectDetaching()...
LanLan graduated from school...
```

6.11 inherits

此关键字用于表示普通类class继承普通类class。例如在inheritsDemo.car文件中：

```
    module
     {
        interface IFoo {
            Foo();
        }

        interface IBar  {
                Bar();
        }
        class CFoo {
                virtual interface IFoo;
            }

        class CBar inherits CFoo {
                interface IBar;
        }
}
```

使用 inherits 关键字是必须注意：

1. 被继承的基类必须包含虚接口。

2. 此关键字只用于表示普通类之间的继承，至于用户的 applet 继承于系统实现的 TApplet，是一种特殊情况。

3. 此关键字只能用于单继承。

4. inherits 结构、aggregates 结构以及 substitutes 结构都可以修饰普通类 class，这三者的顺序是先 inherits 结构，再 aggregates 结构，再 substitutes 结构。

6.12 extends

此关键字表示接口 interface 继承接口 interface。而且只能用于接口间单继承。例如在 extendsDemo.car 文件中：

```
module
     {
        interface IFoo {
                Foo();
        }

    interface IBar extends IFoo  {
```

```
                Bar();
            }
            class CBar {
                  interface IBar;
        }
    }
```

6.13 final

此关键字用于修饰普通构件类 class，表示该类不允许被继承。例如在 finalDemo.car 文件中：

```
    module
    {
        interface IFoo {
                Foo();
        }

             interface IBar  {
                Bar();
        }
        final class CFoo {
            virtual interface IFoo;
        }

        class CBar inherits CFoo {   //Error：can’t inherits final class
                interface IBar;
        }
     }
```

使用说明：

1. final 只能修饰 class，不能修饰其它构件类。
2. final 可以和关键字 singleton 同时修饰构建类，使用顺序是 singleton 在前，final 在后。

6.14 substitutes

此关键字用于表示普通类 class 继承泛类 generic。例如在 substitutesDemo.car 文件中：

```
    module
    {
             interface IFoo {
```

```
            Foo();
               }

             interface IBar {
            Bar();
             }


             generic GFoo {
             interface IBar;
             }

             class CFoo substitutes GFoo {
              interface IBar; //必须声明 GFoo 实现的接口
              interface IFoo;
             }
    }
```

使用说明：

1. 子类中必须声明继承的泛类中声明的接口。

2. 泛类中不允许声明虚接口 virtual。

3. 泛类只能用于被普通类 class 继承，不能被 aspect、context、applet 继承。

4. inherits 结构、aggregates 结构以及 substitutes 结构都可以修饰普通类 class，这三者的顺序是先 inherits 结构，再 aggregates 结构，再 substitutes 结构。

6.15 virtual

此关键字用于在类中声明虚接口，虚接口可以被派生类重载。例如在 virtualDemo.car 文件中：

```
    module
    {
        interface IAnimal {
            Says([in] WString sound);
        }

        class CAnimal {
            virtual interface IAnimal;
        }

        class CPig inherits CAnimal {
            interface IAnimal;
        }
```

```
        class CCat inherits CAnimal {
            interface IAnimal;
        }
    }
```

使用说明：

1.被继承的基类 class 中必须声明虚接口；

2.派生类可以调用基类中非 virtual 接口中的方法。

3.当基类声明了虚接口后，派生类可以有选择的重载基类的虚接口。在派生类中重新声明该接口则表示重载基类的虚接口；如果在派生类中不重新声明，则表示不重载。

4.派生类重载基类虚接口时，需重载该接口中的所有方法。但不论重载与否，在派生类的所有方法中均可以通过 Super::前缀，调用基类的虚接口中的方法。

5.虚接口有着类似 c++函数的特性，即运行时的多态性。在程序运行时，若在基构件中调用虚接口方法，程序会跳到派生构件里相应虚接口的方法中执行，而派生构件可以重载和直接调用基构件中相应虚接口的方法的实现。

编译上述 virtualDemo.car 文件，在生成的代码框架中添加如下:

```
    //CCat.cpp 文件

    #include "CCat.h"
    #include "_CCat.cpp"
    ECode CCat::Says(
      /* [in] */ WString sound)
    {
          // TODO: Add your code here
        CConsole::Write("There is a cat saying: ");
        CConsole::WriteLine(sound);
        return NOERROR;
    }
    //CPig.cpp 文件

    #include "CPig.h"
    #include "_CPig.cpp"
    ECode CPig::Says(
       /* [in] */ WString sound)
    {
        // TODO: Add your code here
        CConsole::Write("There is a pig saying: ");
        CConsole::WriteLine(sound);
        return NOERROR;
```

```
}

// CAnimal.cpp 文件

 #include "CAnimal.h"
 #include "_CAnimal.cpp"
ECode CAnimal::Says(
    /* [in] */ WString sound)
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}
```

客户端实现如下：

```
#include "virtualDemo.h"
using namespace Elastos;


ECode ElastosMain(const BufferOf<WString>& args)
{
    ECode ec;
     IAnimal* pAnimal = NULL;

         ec = CPig::New(&pAnimal);
           if (FAILED(ec)) {
          CConsole::WriteLine("The pig isn't here...");
          return ec;
     }

         pAnimal->Says(L"meow!");
         pAnimal->Release();

     ec = CCat::New(&pAnimal);
     if (FAILED(ec)) {
           CConsole::WriteLine("The cat isn't here...");
           return ec;
      }

         pAnimal->Says(L"oink!");
      pAnimal->Release();

      CProcess::Exit(0);
```

```
            return NOERROR;
        }
```

运行结果如下：

```
        There is a pig saying: meow!
        There is a cat saying: oink!
```

6.16 asynchronous

此关键字用于修饰接口，表示被修饰的接口包含的方法将以异步方式被调用。 如果某个接口包含比较耗时的操作，则可以将该接口声明为异步接口。当该接口的方法被调用时，调用者不必等待被调方法执行完成就可以继续后面的操作。被调用的接口方法则由后台线程负责执行。例如在 asynchronousDemo.car 文件中：

```
    module
    {
        interface IFooFile{
            FooWrite([in] WString name, [in] MemoryBuf<> buffer);
        }

        callbacks JFooFileEvents {
            FooCompleted([in] Int32 writtenBytes);
        }

        class CFooFile {
            asynchronous interface IFooFile;
            callbacks JFooFileEvents;
        }
    }
```

使用说明：

1. asynchronous 的接口方法参数不能包含 out 参数。

2. 实现 asynchronous 接口与普通接口一样，但要注意，用户的实现代码是被其所属的回调线程调用执行。

编译上述 asynchronousDemo.car 文件将生成 CFooFile.h 和 CFooFile.cpp 文件。填充 CFooFile.cpp 文件如下：

```
    #include "CFooFile.h"
    #include "_CFooFile.cpp"
    #include <String.h>
    #include <stdio.h>
```

```
#include <time.h>

ECode CFooFile::FooWrite(
    /* [in] */ WString name,
    /* [in] */ const MemoryBuf & buffer)
{
    // TODO: Add your code here

    FILE *fp = _wfopen(name, L"w");
    int i;

    if (NULL == fp) {
        CConsole::WriteLine("open error");
        return -1;
    }

    for (i = 0; i < 50; i ++) {
        fwrite(buffer.GetPayload(), 1, buffer.GetCapacity(), fp);
    }

    Callback::FooCompleted(1);


    time_t Cuntime;
    time(&Cuntime);

    CConsole::Write(L"Current time is ");
    CConsole::WriteLine((Int32)Cuntime);

    return NOERROR;
}
```

客户端代码如下：

```
#include "asynchronousDemo.h"
#include <time.h>
using namespace Elastos;

ECode OnFooCompleted(PVoid userData, PInterface pSender, Int32 nWrittenBytes)
{
    CConsole::Write(L"callback time: ");
    time_t begTime;
```

```
    time(&begTime);

    CConsole::WriteLine((Int32)begTime);

    CFooFile::RemoveAllCallbacks(pSender);
    CApplet::Finish(AppletFinish_ASAP);

    return NOERROR;
}

ECode ElastosMain(const BufferOf<WString>& args)
{
    IFooFile * pIFooFile;
    ECode ec;
    MemoryBuf_<994851> buffer;

    time_t begTime;
    time(&begTime);

    CConsole::Write (L"Begin time is   ");
    CConsole::WriteLine((Int32)begTime);

    ec = CFooFile::New(&pIFooFile);
    CFooFile::AddFooCompletedCallback(pIFooFile, &OnFooCompleted);

    pIFooFile->FooWrite(L"user.dat",  buffer);

    CConsole::Write(L"End time is: ");
    time_t endTime;
    time(&endTime);

    CConsole::WriteLine((Int32)endTime);

    CObject::ReleaseAtThreadQuit(pIFooFile);

    return NOERROR;
}
```

运行结果如下：

```
Begin time is   1234777792
End time is 1234777792
```

```
Current time is 1234777794
callback time: 1234777794
```

示例 asynchronousDemo.car 文件中，声明了 asynchronous 接口，当 IFile 的 Write 方法被调用时，系统自动通过后台线程完成，调用者线程直接继续后面的操作，所以我们看到在 Write 调用前后的 time(0)返回的时间相同。

6.17 privileged

此关键字用于设置方面类 aspect 中接口的可见性，表示其只能在聚合或解聚合过程中被 Context 相关构件可见。例如在文件 privilegedDemo.car 文件中：

```
module
{
    interface IStudent {
        Study();
        GetID([out] Int32* id);
    }
    interface IChild {
        Play();
    }
    interface ISchool {
        Open();
    }
    interface IStudentAdmin {
        SetID([in] Int32 id);
    }
    class CChild {
        interface IChild;
    }
    aspect AStudent {
        interface IStudent;
        privileged interface IStudentAdmin;
    }

    context KSchool pertainsto AStudent {
        interface ISchool;
    }
}
```

使用说明：

1.此关键字不能用于修饰其它构件类的接口，只能用于修饰 aspect 中的接口。

2.此关键字只能在 context 相关的聚合和解聚合的情况下使用,具体指在以下方法中可以被 probe 出:

context 涉及聚合的所有对象的 OnAspectAttaching 和 OnAspectDetaching 方法。

Context 构件的 OnObjectEntered 方法、OnObjectLeaving 方法。注意，在 OnObjectEntering()方法和 OnObjectLeft()方法中 Probe 不到。

编译上述 privilegedDemo.car，自动生成代码框架如下：

KSchool.cpp KSchoo.h AStudent.cpp AStudent.h CChild.cpp CChild.h

修改 KSchool.cpp 文件如下：

```
#include "KSchool.h"
#include "_KSchool.cpp"
ECode KSchool::Open()
{
    // TODO: Add your code here
    CConsole::WriteLine("School is open ...");
    return NOERROR;
}

ECode KSchool::AcquireClassFactory(
    /* [in] */ RClassID pRclsid,
    /* [out] */ PInterface * ppPpObject)
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}

ECode KSchool::OnObjectEntering(PInterface pObject)
{
    CConsole::WriteLine(L"-> KSchool::OnObjectEntering...");
    return NOERROR;
}

ECode KSchool::OnObjectEntered(PInterface pObject)
{
    CConsole::WriteLine(L"-> KSchool::OnObjectEntered()...");
    IStudentAdmin* pStudentAdmin;
    pStudentAdmin = IStudentAdmin::Probe(pObject);
    if (!pStudentAdmin) {
        return E_NO_INTERFACE;
    }
    else CConsole::WriteLine("Probe sucess!");
```

```
    return NOERROR;
}
ECode KSchool::OnObjectLeaving(PInterface pObject)
{
    CConsole::WriteLine(L"-> KSchool::OnObjectLeaving...");
    IStudentAdmin* pStudentAdmin;
    pStudentAdmin = IStudentAdmin::Probe(pObject);
    if (!pStudentAdmin) {
        return E_NO_INTERFACE;
    }
    else CConsole::WriteLine("Probe sucess!");
    return NOERROR;
}
ECode KSchool::OnObjectLeft(PInterface pObject)
{
    CConsole::WriteLine(L"-> KSchool::OnObjectLeft...");

    return NOERROR;
}
```

修改 AStudent.cpp 文件如下：

```
#include "AStudent.h"
#include "_AStudent.cpp"
ECode AStudent::Study()
{
    // TODO: Add your code here
    CConsole::WriteLine("I'm a student studying...");
    return NOERROR;
}

ECode AStudent::GetID(
    /* [out] */ Int32 * pId)
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}

ECode AStudent::SetID(
    /* [in] */ Int32 id)
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
```

```
}

ECode AStudent::OnAspectAttaching(
    /* [in] */ PInterface pObject)
{
    CConsole::WriteLine(L"-> AStudent::OnAspectAttaching()...");
    return NOERROR;
}

ECode AStudent::OnAspectDetaching(
    /* [in] */ PInterface pObject)
{
    CConsole::WriteLine(L"-> AStudent::OnAspectDetaching()...");
    IStudentAdmin* pStudentAdmin;
    pStudentAdmin = IStudentAdmin::Probe(pObject);
    if (!pStudentAdmin) {
        return E_NO_INTERFACE;
    }
    else CConsole::WriteLine("Probe sucess!");
    return NOERROR;
}
```

修改 CChild.cpp 文件如下：

```
#include "CChild.h"
#include "_CChild.cpp"

ECode CChild::Play()
{
    // TODO: Add your code here
    CConsole::WriteLine("I'm a child playing...");
    return NOERROR;
}

ECode CChild::OnAspectAttaching(
    /* [in] */ PInterface pAspect)
{
    CConsole::WriteLine(L"-> CChild::OnAspectAttaching()...");
    IStudentAdmin* pStudentAdmin;
    pStudentAdmin = IStudentAdmin::Probe(pAspect);
    if (!pStudentAdmin) {
        return E_NO_INTERFACE;
    }
```

```
        else CConsole::WriteLine("Probe sucess!");
        return NOERROR;
    }

    ECode CChild::OnAspectDetaching(
        /* [in] */ PInterface pAspect)
    {
        CConsole::WriteLine(L"-> CChild::OnAspectDetaching()...");
        IStudentAdmin* pStudentAdmin;
        pStudentAdmin = IStudentAdmin::Probe(pAspect);
        if (!pStudentAdmin) {
            return E_NO_INTERFACE;
        }
        else CConsole::WriteLine("Probe sucess!");
        return NOERROR;
    }
```

客户端实现如下：

```
    #include "priviledgedDemo.h"
    using namespace Elastos;

    ECode ElastosMain(const BufferOf<WString>& args)
    {
        IChild *pChild;
        ISchool *pSchool;

        ECode ec = CChild::New(&pChild);
        if (FAILED(ec)) return ec;
        pChild->Play();

        ec = KSchool::New(&pSchool);
        if (FAILED(ec)) return ec;
        pSchool->Open();

        CConsole::WriteLine("child not entered school now!");
        ec = CObject::EnterContext(pChild, pSchool);
        CConsole::WriteLine("child entered school already!");

        IStudent *pStudent = NULL;
        pStudent = IStudent::Probe(pChild);
        if (NULL == pStudent) return ec;
        pStudent->Study();
```

```
        CObject::LeaveContext(pChild, pSchool);
        CConsole::WriteLine("child left school already!");
        pChild->Release();
        pSchool->Release();

        CProcess::Exit(0);
        return NOERROR_EXIT;
    }
```

运行结果如下：

```
    I'm a child playing...
    School is open ...
    child not entered school now!
    -> KSchool::OnObjectEntering...
    -> AStudent::OnAspectAttaching()...
    -> CChild::OnAspectAttaching()...
    Probe sucess!
    -> KSchool::OnObjectEntered()...
    Probe sucess!
    child entered school already!
    I'm a student studying...
    -> KSchool::OnObjectLeaving...
    Probe sucess!
    -> CChild::OnAspectDetaching()...
    Probe sucess!
    -> AStudent::OnAspectDetaching()...
    -> KSchool::OnObjectLeft...
    child left school already!
```

可以看出：在 CChild 对象进入 KSchool 后，可以查询到 privileged 接口 IStudentAdmin。进入 KSchool 前和离开 KShool 后均不能再查询到此接口。

6.18 filtering

此关键字用于派生类拦截基类抛出的回调事件，从而可以过滤回调事件，有选择的重新抛出、不抛或者抛出修改过的事件。比如有一个 CForm 的基类，它会抛出 Click 的事件，又有一个 CDialog 继承于 CForm，当用户创建一个 CDialog 对象时，可以注册 Click 事件，对于 CForm 来说，用户鼠标在 Form 范围内任何地方的点击事件都会抛出，而使用 CDialog 的用户则可能只关心点击到 button 的 Click 事件，这时就可以利用回调事件的过滤机制拦截掉多余的 Click 事件。 例如在 filteringDemo.car 文件中

```
    module
    {
```

```
        interface IFoo {
            Foo();
        }

        callbacks JFooEvents {
            FooEvent();
        }

        class CFooBase {
            virtual interface IFoo;
            callbacks JFooEvents;
        }

        class CFoo inherits CFooBase {
            interface IFoo;
            filtering callbacks JFooEvents;
        }
    }
```

使用说明：

1. car 编译工具并不会根据 filter 关键字在前台自动生成下述函数框架，需要手动添加。 注意此方法的参数，除第一个参数外其余参数必须与回调方法中声明的参数一致。

```
        virtual CARAPI FooEventFilter(Int32 cFlags, Int32 id);
```

2. 当要过滤某个事件时，要将对应事件的 filter 函数分别在代码框架（.h 和.cpp）中声明和实现。Filter 函数的声明格式为“回调事件名”+“Filter” 如：回调事件为 FooEvent，则过滤函数名为 FooEventFilter。

上述 car 文件定义了类 CFooBase 及其派生类 CFoo。类 CFoo 可以过滤其从 CFooBase 继承的回调事件。编译 car 文件，自动生成代码框架，主要源文件如下：CFooBase.h 文件、CFooBase.cpp 文件、CFoo.h 文件、CFoo.cpp 文件。需要在过滤事件所在的类文件 CFoo.h 中声明过滤函数：

```
        virtual CARAPI FooEventFilter(Int32 cFlags);
```

添加后 CFoo.h 文件如下：

```
    #ifndef __CFOO_H__
    #define __CFOO_H__

    #include "_CFoo.h"

    CarClass(CFoo)
    {
    public:
        CARAPI Foo();
```

```
    virtual CARAPI FooEventFilter(Int32 cFlags);

private:
    // TODO: Add your private member variables here.
};

#endif // __CFOO_H__
```

修改 CFoo.cpp 文件如下：

```
#include "CFoo.h"
#include "_CFoo.cpp"

ECode CFoo::Foo()
{
    // TODO: Add your code here
    CConsole::WriteLine("In CFoo::Foo()");
    Callback::FooEvent();
    return NOERROR;
 }

ECode CFoo::FooEventFilter(Int32 cFlags)
{
    CConsole::WriteLine(L"CFoo::FooEventFilter");

    Callback::FooEventWithPriority(cFlags);  //激发（具有优先级的）回调
    CApplet::Finish(AppletFinish_Nice);

    return NOERROR;
}
```

修改 CFooBase.cpp 文件如下：

```
#include "CFooBase.h"
#include "_CFooBase.cpp"

ECode CFooBase::Foo()
{
    // TODO: Add your code here
    CConsole::WriteLine("In CFooBase::Foo()\n");
    Callback::FooEvent();

    return NOERROR;
}
```

客户端实现如下：

```
#include "filteringDemo.h"
using namespace Elastos;

ECode OnFooEvent(PVoid pUserData, PInterface pSender)
{
    CConsole::Write("In OnFooEvent ");
    CFooBase::RemoveAllCallbacks(pSender);

    return NOERROR;
}

ECode ElastosMain(const BufferOf<WString>& args)
{
    IFoo *pFoo = NULL;

    ECode ec = CFoo::New(&pFoo);
    if (FAILED(ec)) return ec;

    ec = CFoo::AddFooEventCallback(pFoo, OnFooEvent, NULL);

    if (FAILED(ec)) {
        pFoo->Release();
        return ec;
    }

    pFoo->Foo();

    CObject::ReleaseAtThreadQuit(pFoo);

    return NOERROR;
}
```

运行结果如下：

```
In CFoo::Foo()
CFoo::FooEventFilter  //过滤了回调事件
In OnFooEvent         //filter 函数中激发的回调事件
```

6.19 singleton

此关键字用于修饰构件类（aspect 构件类除外），表示该类只能创建出唯一的对象。例如在 singletonDemo.car 文件中：

```
module
{
    interface IFoo {
        Hello();
    }

    singleton class CFoo {
        constructor();
        interface IFoo;
    }
}
```

使用说明：

1.singleton 修饰的构件类只能拥有不带参数的构造函数。

2.Singleton 不能修饰 aspect 构件类。

3.singleton 可以和关键字 final 同时修饰构件类,使用顺序是 singleton 在前,final 在后。

上述示例定义了一个具有 Singleton 属性的类 CFoo。编译 singleton.car，扩充生成的 CFoo.cpp 文件中 constructor()方法如下：

```
ECode CFoo::constructor()
{
    CConsole::WriteLine ("Constructor.");
    return NOERROR;
}
```

客户端调用 CFoo:AcquireSingleton()或 CFoo::AcquireSingletonInContext()来获取对象。第一次调用创建出实际的对象，随后的调用仅仅返回已创建的对象。

```
#include <stdio.h>
#include "singletonDemo.h"
using namespace Elastos;

int main()
{
    IFoo* pIFoo1, *pIFoo2;
    ECode ec;

    //创建 Singleton 对象 pIFoo1
    ec = CFoo::AcquireSingleton(&pIFoo1);
    if (FAILED(ec)) return ec;

    //创建 Singleton 对象 pIFoo2
    ec = CFoo::AcquireSingleton(&pIFoo2);
```

```
        if (FAILED(ec)) {
            pIFoo1->Release();
            return ec;
        }

       //打印出两个对象指针的地址，验证是否指向同一个对象
        printf("pIFoo1 :%p\n", pIFoo1);
        printf("pIFoo2 :%p\n", pIFoo2);

        pIFoo1->Release();
        pIFoo2->Release();
        return NOERROR;
    }
```

程序运行结果为:

```
Constructor.
pIFoo1 : 00963A08
pIFoo2 : 00963A08
```

由此可见，CFoo 的构造函数只是在第一次创建对象时被调用。

备注：

1. AcquireSingletonInContext()方法的 signature 为服务器端自动代码生成框架会生成三个方法供客户端创建一个 singleton 对象:

```
static ECode AcquireSingleton(/*[out]>*/ IFoo **ppIFoo)
// 创建一个 singleton 对象(同一 Domain)

static ECode AcquireSingletonInContext(
            /*[in] */ IContext* pContext,
            /*[out] */ IFoo** ppIFoo)
//根据 pContext 创建一个 singleton 对象

static ECode AcquireSingletonInContext(
            /*[in]*/ IInterface* pContextObject,
            /*[out]*/ IFoo** ppIFoo)
//根据 pContextObject 创建一个 singleton 对象
```

当参数 PContext 取值为 CTX_SAME_DOMAIN 或者 CTX_SAME_PROCESS 时，构件对象创建在与该 Client 端同一个进程空间内；当参数 PContext 取值为 CTX_DIFF_DOMAIN 或者 CTX_DIFF_PROCESS 或者 CTX_DIFF_MACHINE 时，NewInContext 把构件对象创建在与该 Client 端不同的进程空间内。

6.20 aggregates

此关键字用于修饰构件类（aspect 类除外），表示被修饰的构件类创建对象时将自动聚合该属性中列举的方面构件对象。例如在 aggregatesDemo.car 文件中：

```
module
{
    interface IFoo {
        Foo();
    }
    aspect AFoo {
        interface IFoo;
    }
    interface IBar {
        Bar();
    }

    class CBar aggregates AFoo {
        interface IBar;
    }
}
```

使用说明：

1. inherits 结构、aggregates 结构以及 substitutes 结构都可以修饰普通类 class，这三者的顺序是先 inherits 结构，再 aggregates 结构，再 substitutes 结构。

2. aggregates 不能修饰 generic。

3. aggregates 结构和 pertainsto 结构都可以修饰 applet 和 context，它们的顺序是先 aggregates，再 pertainsto 结构。

编译上述 aggregatesDemo.car 文件，修改生成的代码框架如下：

```
//CBar.cpp 文件：

#include "CBar.h"
#include "_CBar.cpp"
ECode CBar::Bar()
{
    // TODO: Add your code here
    CConsole::WriteLine("call CBar::Bar() success...");
    return NOERROR;
}

//AFoo.cpp 文件：
```

```
#include "AFoo.h"
#include "_AFoo.cpp"
ECode AFoo::Foo()
{
    // TODO: Add your code here
    CConsole::WriteLine("call AFoo::Foo() success...");
    return NOERROR;
}
```

客户端实现如下：

```
#include "aggregateDemo.h"
using namespace Elastos;


ECode ElastosMain(const BufferOf<WString>& args)
{
    IFoo *pFoo = NULL;
    IBar *pBar = NULL;
    ECode ec;

    ec = CBar::New(&pBar);
    if (FAILED(ec)) return ec;

    pBar->Bar();

    pFoo = IFoo::Probe(pBar);
    if (NULL == pFoo) return E_NO_INTERFACE;

    pFoo->Foo();

    pBar->Release();

    CProcess::Exit(0);
    return NOERROR;
}
```

运行结果如下：

```
call CBar::Bar() success...
call AFoo::Foo() success...
```

6.21 pertainsto

此关键字用于修饰语境构件类 context，表示该语境构件类拥有该属性中列举的方面构件类，当其它构件对象进入该语境构件类时，其它构件对象会自动聚合上这些方面构件类；当对象离开该语境构件类时，将自动解聚合掉这些方面构件类。例如在 pertainstoDemo.car 文件中：

```
module
{
    interface IHello {
        Hello();
    }

    context KFoo pertainsto AObjectEx {
        interface IHello;
    }
}
```

使用说明：

pertainsto 结构和 aggregates 结构都可以修饰 applet 和 context，它们的顺序是先 aggregates，再 pertainsto 结构。

上述 car 构件定义了语境构件类 KFoo，该构件类由 pertainsto 修饰，表示 KFoo 具有 AObjectEx 特征。当其他构件对象进入 KFoo 对象时就会自动聚合 AObjectEx 方面（该方面是 Elastos 系统预定义的 aspect 类，也可使用自定义的 aspect 类）。聚合上这些这些方面以后，就可以访问这些方面包含的接口。当这些构件对象离开 KFoo 环境时，就会自动拆卸与 AobjectEx 的聚合。

详细的编程例子可以参考语境类编程。

6.22 affiliates

此关键字用于修饰方面构件类 aspect，表示被修饰的方面构件类只能被该属性中列举的非方面构件类聚合，且 aspect 将使用目标 class 的线程模型。例如在 affiliatesDemo.car 文件中：

```
module
{
    interface IChildZhang {
        Play();
    }
    interface IChildLi {
        Play();
```

```
        }
        interface IStudent {
            Study();
        }

        class CChildZhang {
            interface IChildZhang;
        }
        class CChildLi {
            interface IChildLi;
        }

        aspect AStudent affiliates CChildZhang {
            interface IStudent;
        }
    }
```

示例中，方面类 AStudent 被“affiliates CChildZhang”修饰，这表示 AStudent 方面类只能与 CChildZhang 类聚合，而不能和其他普通类 CChildLi 聚合，并且 AStudent 方面类将使用 CChild 的锁和相同的线程模型。

客户端实现如下:

```
#include "classDemo.h"
using namespace Elastos;

ECode ElastosMain(const BufferOf<WString>& args)
{
    IChildZhang * pChildZhang;
    IChildLi *pChildLi = NULL;
    IStudent *pStudent = NULL;
    ECode ec;

    ec = CChildZhang::New(&pChildZhang);
    if (FAILED(ec)) {
        return ec;
    }

    ec = AStudent::Attach(pChildZhang);
    if (FAILED(ec)) {
        return ec;
    }
    CConsole::WriteLine("ChildZhang attach AStduent sucess!");
```

```
        pStudent = IStudent::Probe(pChildZhang);
        if (NULL != pStudent) {
            CConsole::WriteLine("IChildZhang Probe IStduent sucess!");
        }
        pChildZhang->Release();

        ec = CChildLi::New(&pChildLi);
        if (FAILED(ec)) {
            return ec;
        }

        ec = AStudent::Attach(pChildLi);
        if (FAILED(ec)) {
            CConsole::WriteLine("ChildLi attach AStudent error");
            pChildLi->Release();
            return ec;
        }

        CProcess::Exit(0);
        return NOERROR;
    }
```

运行结果如下：

```
ChildZhang attach AStduent sucess!
IChildZhang Probe IStduent sucess!
ChildLi attach AStudent error
```

6.23 const

此关键字用于定义 int 类型的常量，语法为：

```
const ident = < integer | hinteger >;
```

例如：const SIZE = 16;

定义的常量可以在CAR的数组或者缓冲区大小定义中使用,也可以作为常量用在struct普通数组变量的[]内。如WStringBuf_<SIZE>或者 struct Build { Int32 days[SIZE]; }。也可以在CPP源文件中使用。例如在constDemo.car文件中：

```
module
{
    ...
    const SIZE = 16;
    struct MyStruct{
        WChar wchName[SIZE];
```

```
        ...
    }

    interface IMyStruct {
        ECode Func([in] MyStruct eMyStruct);
    }
    ...
}
```

上述示例定义了常量 SIZE，用于指定结构体成员 wchName 数组的长度。

6.24 enum

此关键字用于定义枚举类型，使用方法与在 C/C++程序中基本相同。例如：

```
module
{
    ……
    enum MyEnum {
        MyEnum_red,
        MyEnum_blue,
        MyEnum_white = 1976,
        MyEnum_black, （该逗号可有可无）
    }
    interface IMyEnum {
        ECode Func([in] MyEnum eMyEnum);
    }
    ……
}
```

示例中，定义了一个枚举类型 MyEnum 以及使用 MyEnum 类型的参数 eMyEnum。

6.25 struct

此关键字用于定义结构体，使用方法与在 C/C++程序中基本相同。例如：

```
module
{
    ……
    struct  MyStruct  {
        WChar  wcElem;
        Float  fElem;
        Int64 *plElem;
        Int32Array naElem;
```

```
        }
        ……
    }
```

示例中，定义了一个结构体 MyStruct。由 struct 关键字定义的结构体中的数据类型必须是 CAR 支持的数据类型。

6.26 typedef

此关键字用于定义类型，使用方法与在 C/C++程序中基本相同。例如：

```
module
{
    ……
    typedef char CHAR, CHARR;
    typedef CHAR* PCHAR;
    typedef struct {
        Int64 a;
        AString asName;
    }*A, B;
    typedef enum {
        red,
        white,
    }Aenum;（该分号必须有）
    ……
}
```

6.27 merge

此关键字用于在本构件中合并用到的其它构件（.car 文件），就象其它构件的 CAR 文件直接写在本 CAR 文件的当前位置一样。这时的 module 关键字及最外层的大括号“｛”的定义将被隐掉。

语法为：merge ("构件名.car")，参照关键字 import 在 car 文件中的示例，把 import("Demo.car")改成 merge ("Demo.car")即可。

假设在本示例 mergeDemo.car 文件的上级目录中存在一个 foo.car 文件，内容如下：

```
module
{
    interface IFoo {
        Foo();
    }
    class CFoo {
        interface IFoo;
    }
```

```
}
```

编写如下 mergeDemo.car 示例内容如下：

```
module
{
    merge ("..\foo.car");

    interface IBar {
        Bar();
    }
    class CBar {
        interface IBar;
    }
}
```

示例 car 中，由于 mergeDemo.car 中 merge 了 foo.car，其自动生成代码框架中，包含了 foo.car 的自动代码框架生成的 CFoo.h 和 CFoo.cpp 文件。

6.28 mergelib

此关键字用于在本构件中合并用到的其它构件（.dll 或者.cls 文件）中被当前构件的定义引用到的部分。

语法为：mergelib("构件名.dll")或者 mergelib("构件名.cls")。

使用说明：

1.所引用的.dll 或.cls 须存在，否则编译报错。

2.可使用 carc -c xxx.car 命令根据 xxx.car 文件生成其.cls 文件。

3.不能指定路径名。

假设在本示例 mergelibDemo.car 文件的上级目录中存在一个 foo.car 文件，内容如下：

```
module
{
    interface IFoo {
        Foo();
    }
    class CFoo {
        interface IFoo;
    }
}
```

编写如下 mergelibDemo.car 示例内容如下：

```
module
{
```

```
        mergelib ("foo.dll");

        interface IBar {
            Bar([in] Int32 i);
        }
        class CBar {
            interface IBar;
        }
    }
```

示例 car 中，由于 mergelibDemo.car 中 mergelib 了 foo.car，其自动生成代码框架中，包含了 foo.car 的自动代码框架生成的 CFoo.h 和 CFoo.cpp 文件。

6.29 import

在 car 文件里，此关键字用于引入别的 car 文件。

语法为：import("构件名.car")。

例如，构件 ImportDemo1 的 car 文件：

```
    module
    {
        interface IInterface3 {
            ……
        }
        import("Demo.car");
        class CClass1 {
            interface IInterface1;
            interface IInterface3;
        }
        class CClass2{
            interface IInterface2;
        }
    }
```

示例构件 ImportDemo1 中，定义了一个接口 IInterface3，两个类：CClass1 和 Cclass2，其中 IInterface1, IInterface2 分别来自引入的构件 Demo.dll。 在代码生成框架中需要重新实现这两个接口中的方法。

下面是引入的构件 Demo 的定义信息：

引入构件 Demo.car：

```
    module
    {
        interface IInterface1 {
            Hello();
```

```
        }
        interface IInterface2 {
            ……
        }
       Class CDemo{
            interface IInterface1
        }
    }
```

示例构件 Demo 中，定义了两个接口 IInterface1 和 IInterface2。

使用说明：

1.在引入构件时,如果用户不指定引入构件所在路径,将在系统默认路径中搜索该构件。

2.在使用该关键字引入构件前，首先要确定该构件的存在，否则将在编译时给出警告。

在 cpp 文件里，此关键字用于引入其它构件（dll 文件）。这样，本构件就可以直接实例化引入构件并使用引入构件中定义的接口，类等信息。

语法为：

```
#import<构件名.dll>
```

或者：

```
#import "构件名.dll"
```

引入一个 dll 的语句必须在一行中完成,中间不能有回车换行符,并且一次只能 import 一个 dll。

6.30 importlib

此关键字用于在本构件中引入用到的其它构件（.dll 或者.cls 文件），这样，本构件就可以直接使用在引入构件中定义的接口，类等信息。

语法为：importlib("构件名.dll")或者 importlib("构件名.cls")。

使用说明：

1.所引用的.dll 或.cls 须存在，否则编译报错。

2.可使用 carc -c xxx.car 命令根据 xxx.car 文件生成其.cls 文件。

3.不能指定路径名。

假设在本示例 importlibDemo.car 文件的上级目录中存在一个 foo.car 文件，内容如下：

```
module
{
    interface IFoo {
        Foo();
    }
    class CFoo {
        interface IFoo;
```

```
    }
}
```

编写如下 importlibDemo.car 示例内容如下：

```
module
{
    importlib("foo.dll");
    interface IBar {
        Bar();
    }
    class CBar {
        interface IFoo;
        interface IBar;
    }
}
```

示例 car 中，由于 importlibDemo.car 中引入了 foo.car，便可使用 foo.car 的成员 IFoo 接口。

6.31 constructor

此关键字用于定义构造函数，使 CAR 构件类在实例化时传入初始化参数。例如：

```
module
{
    interface IInterface {
        Hello(int nTimes);
    }
    class CClass {
        constructor ([in] WString wstr); //参数只能是 in 属性
        interface  IInterface;
    }
}
```

示例构件 ConstructorDemo 中，定义了本构件的构造函数，构造函数不需要名字，并且构造函数的参数只能是[in]参数。

注意：

- 两个构造函数参数个数和类型一样，其实是一个构造函数。
- 可以定义多个构造函数，构造函数和构造函数的参数不能完全一样，也就是“参数 1 字符串＋参数 2 字符串＋参数 3 字符串＋……最后一个参数字符串”在一个构件类中不可以重复。

  象下面这类构造函数同时定义在一个构件类中是错的：

```
constructor ([in] WString wstr, [in] WString wstr);
constructor ([in] AString wst, [in] AString rwstr);
```

参数只能是[in]属性的，不能有[out]属性的参数。

构造函数有以下规则：

1.在 Server 端生成的代码框架中，会为 C++类(class CClass)生成 New 方法，如 CClass 就有如下 New 方法：

```
Static ECode CClass::New(WString wstr, IInterface **pIInterface);
```

如果没有构造函数就只有：

```
Static ECode CClass::New(IInterface **pIInterface);
```

如果一个构件类有 m 个构造方法和 n 个接口（目前不包括事件接口），那么就有 m*n 个 New 方法，用户可用任意的构造函数来创建出改类任意的接口，如果没有构造函数，那么 New 方法中就只有接口参数。

在 New 方法里实际调用顺序是先创建对象，然后调用服务端相应的 constructor。

2. 客户端可以直接调用 New 方法

有构造函数的构件类：

```
ec = CClass::New((L"Hello!", &pIInterface);
```

没有构造函数的构件类：

```
ec = CClass::New((&pIInterface );
```

6.32 pragma

此关键字用于禁止或允许显示 CAR 编译器给出的警告信息。

语法为：

禁止警告 #pragma (disable:nnn)

允许警告 #pragma (enable:nnn)，

其中 disable 表示禁止，enable 表示允许，nnn 为警告号。例如：

```
……
interface IInterface1 {
    Func([in] int i);
}
interface IInterface2 {
    Func([in] int i);
}
……
```

编译有示例中代码的 CAR 文件时，编译器会显示一条警告信息：

```
CARC(0X100B): Warning - Method name conflict: "Func()" in "IInterface1, IInterface2".
```

其中 0X100B 是警告号。

在 CAR 文件中导致此警告出现的代码模块前加入#pragma (disable:0X100B)语句，就可以禁止该警告信息的显示。

```
……
interface IInterface1 {
    Func([in] int i);
```

```
}
#pragma (disable:0X100B)
interface IInterface2 {
    Func([in] int i);
}
……
```

当根据需要使用#pragma (enable:0X100B)语句时，则可以再次显示该警告信息。例如：

```
……
interface IInterface1 {
    Func([in] int i);
}
#pragma (enable:0X100B)
interface IInterface2 {
    Func([in] int i);
}
……
```

6.33 coalesce

此关键字用于合并同类回调事件。比如下载文件的进度条，每下载一定量的文件，服务器端可以抛出消息，客户端可以合并和处理这些消息（截获消息并计算总共下载量），以更新下载进度条。

语法为：

```
coalesce callback methodName;
```

例如在 coalesceDemo.car 文件中：

```
module
{
    interface IFoo {
        Foo();
    }
    callbacks JFooEvent {
        FooEvent([in] Int32 nId);
    }

    class CFoo {
        interface IFoo;
        callbacks JFooEvent;
        coalesce callback FooEvent;
    }
}
```

使用说明：

1.coalesce 方法必须包含参数。

2.Coalescer 函数必须配合 UpdatingXXX()函数使用。如果不调用 UpdatingXXX()函数，则 Coalescer 函数不会被执行。函数 UpdatingXXX（参数列表）可单独使用。如果此处是单独使用 UpdatingXXX(参数列表)，即在 car 文件中未声明相应的 coalesce 关键字，执行此处语句，查找消息后若有同类消息将直接丢弃原来的消息，最后消息队列里保留的是 FooEvent(4)。

3.当用户实现的 coalesce 函数返回 E_CANCLE_BOTH_EVENTS 这个返回值时,将要合并的新旧两个事件消息都抛弃。

编译此 car 文件，生成的代码框架 CFoo.cpp 和 CFoo.h。其中 CFoo.h 文件中包含一个用来处理合并回调事件的函数如下：

```
ECode FooEventCoalescer(const IFooEvent_FooEvent_Params* pNew)
{
    return NOERROR;
}
```

用户可补充该函数，用以合并消息的同时完成一些处理工作，例如下述表示：

```
ECode FooEventCoalescer(const IFooEvent_FooEvent_Params* pNew)
{
    CConsole::Write("calling coalescer!");
    CConsole::WriteLine(pNew->m_nId);

    return NOERROR;
}
```

另外，修改 CFoo.cpp 文件，多次激发 FooEvent 回调。

```
#include "CFoo.h"
#include "_CFoo.cpp"

ECode CFoo::Foo()
{
    // TODO: Add your code here
     CConsole::WriteLine(L"Foo");

    Callback::FooEvent(1);
    Callback::FooEvent(2);
    Callback::FooEvent(3);

    Callback::UpdatingFooEvent(4);

    return NOERROR;
}
```

上述代码中，执行函数 Callback::UpdatingFooEvent()时，服务端对回调事件队列的处理流程是这样的：首先查找事件队列的尾部事件(是 FooEvent(3))，与当前的新事件 FooEvent(4)进行比较，由于是同类事件，所以调用事件合并处理函数 FooEventCoalescer(4)，将这两条事件合并为事件 FooEvent(3)。然后继续依次比较队列中的事件，有同类的 FooEvent 事件就调用相应的事件合并处理函数，直至最后合并为事件 FooEvent(1)，并根据该事件的优先级塞入事件队列的适当位置上。

客户端代码如下：

```
#include "coalesceDemo.h"
using namespace Elastos;

ECode OnFooEvent(PVoid userData, PInterface pSender,Int32 nId)
{
    CConsole::Write(L"callback event:FooEvent");
    CConsole::WriteLine(nId);

    if (1 == nId) {
        CFoo::RemoveAllCallbacks(pSender);
        CApplet::Finish(AppletFinish_ASAP);
    }
    return NOERROR;
}

ECode ElastosMain(const BufferOf<WString>& args)
{
    IFoo *pIFoo;
    ECode ec;

    ec = CFoo::New(&pIFoo);
    if (FAILED(ec)) {
        CConsole::WriteLine("error");
        return ec;
    }

    pIFoo->Foo();

    ec = CFoo::AddFooEventCallback(pIFoo, OnFooEvent, NULL);
    if (FAILED(ec)) {
        CConsole::WriteLine("error");
        return ec;
    }
```

```
        pIFoo->Foo();

        CObject::ReleaseAtThreadQuit(pIFoo);

        return NOERROR;

    }
```

运行结果如下：

```
Foo
Foo
calling coalescer!4
calling coalescer!3
calling coalescer!2
callback event:FooEvent1
```

第七章　CAR 属性

属性用于修饰 CAR 中构件类、接口等各个部分的内容，包括：uunm, aspect, in, out 等。用方括号表示一个属性。

下面对 CAR 中支持的属性的含义和表示方法一一进行介绍。

按照所修饰的内容不同，我们将 CAR 属性分为以下几类：

- 用于修饰构件的属性：

属性	描述
version	用于标识构件的版本号

- 用于修饰普通类的属性：

属性	描述
synchronized	用于标识构件对象同时只允许一个线程进入到构件的实现代码中，其它访问该构件对象的线程将被阻塞等待（一旦线程调用到对象方法内，在从对象方法返回之前，其它线程对该对象的访问均会被阻塞）
sequenced	用于标识构件对象同时只允许一个线程进入到构件的实现代码中，其它访问该构件对象的线程将被阻塞等待（对象方法访问其它对象时临时释放锁，允许其它线程进入对象）
private	用于标识所修饰的类只能在服务端被实例化
local	用于修饰带有 constructor（内有非自描述参数）的类
deprecated	用于兼容移植来的第三方软件中不符合 CAR 规范的数据类型使用
applet	此属性用于修饰类，表示该类接口方法的调用具有类似于 applet 的接口方法行为。

- 用于修饰接口的属性：

属性	描述
local	表示提供该接口的类的类对象与客户程序只能在相同的域内使用，不能用于远程调用
deprecated	用于兼容移植来的第三方软件中不符合 CAR 规范的数据类型使用

- 用于修饰接口方法参数的属性：

属性	描述

in	用于标识该参数为输入参数
out	用于标识该参数为输出参数
callee	用于修饰五元组，表示该参数由客户端程序分配

7.1 version

此属性用于标识构件的版本号。语法为：version(M.N)，其中 M（主版本号）和 N（副版本号）为 0 至 255 的整数。例如：

```
[version (1.0)]
module
{
…….
}
```

每次 CAR 文件对外发布，如果 CAR 文件有实质性变化，version 数应该增加（按 16 位计算）。

此属性可以不使用。

7.2 synchronized & sequenced

7..1 synchronized & sequenced 的含义与区别

synchronized 和 sequenced 属性标识构件对象同时只允许一个线程进入到构件的实现代码中，其它访问该构件对象的线程将被阻塞等待，例如 synchronizedDemo.car：

```
module
{
    interface IFoo{
        Foo1();
        Foo2();
    }

    [synchronized]
        class CFoo {
        interface IFoo;
    }
}
```

在 CAR 构件类前应用了 synchronized/sequenced 属性后，该 CAR 构件类的实例将与一个锁对象自动关联（实现上即为 CriticalSection）。当线程调用该构件实例的任一方法时，将首先自动获取该实例相关联的锁对象；当调用方法返回之后，最后还会自动释放相关联的

锁对象。从而保证了在任一时刻，最多只有一个线程能够通过调用该构件实例的某个方法来访问该构件实例；而此刻并发访问该构件实例的任一方法的其他线程将被阻塞，直至当前正在调用该构件实例方法的线程结束对其方法的调用而返回，阻塞在该构件实例任一方法调用上线程中的一个才会被唤醒并被允许进入其所调用的方法中。

另外，锁是可重入的。亦即某线程调用 A 构件的 X 方法获得了与 A 构件相关联的锁，而后经过 n 层嵌套调用到 A 构件的 Y 方法需要再次获得该锁，这个线程一定能够再次获取到该锁并顺利调用 Y 方法。

运用 synchronized 和 sequenced 的区别是：

应用 synchronized 时，一旦线程调用到对象方法内，在从对象方法返回之前，其它线程对该对象的访问均会被阻塞。

而 sequenced 则会在对象方法访问其它对象时临时释放锁，允许其它线程进入对象。比如：当线程一从 A 构件的的 X 方法中调用 B 构件的 Y 方法时，会自动释放掉与与 A 构件相关联的锁并且自动获取与 B 构件相关联的锁。此时，当线程一释放掉与 A 构件相关联的锁并获取了 B 构件相关联的锁后再等待线程二调用 A 构件的 Z 方法时，线程二可以获取与 A 构件相关联的锁而并不会引起死锁。故 sequenced 属性可以避免掉一些原本用 synchronized 属性所引发的死锁问题

对已实现的对象来讲，泛用 synchronized 容易造成死锁，但利于严谨的同步查错。使用 sequenced 则正好相反。

因此建议新写的构件类使用 synchronized 关键字，而已实现的构件则使用 sequenced。

注意：这两个属性不能同时使用

7.2.2 synchronized & sequenced 与锁

Sychronized 对 CAR 构件类的每个方法用同一个 Critical Section 锁包装起来

Sequenced 同样为每个方法加了锁。区别在于在方法内部如果调用了另一个带 Sequenced 的构件类的方法或跨域调用前会先解锁；调用完成，返回后又重新加上锁。

举下述函数为例：

```
CFoo::Foo()
{
    m_count++;
    m_pIBar->Bar();  //*
    m_list->Add(…);
    …
    EzAPI(…);        //*
}
```

Sychronized 会对整个 Foo 函数加锁。

Sequenced 会在调用加*号的语句前解锁，调用加*号的语句后又加上锁。

生成伪代码情况如下：

Sychronized 生成伪代码：

```
CFoo::Foo()
{
```

```
    EnterCriticalSection(&_m_CFooLock);
    m_count++;
    m_pIBar->Bar();
    m_list->Add(…);
    …
    EzAPI(…);
    LeaveCriticalSection(&_m_CFooLock);
}
```

Sequenced 生成伪代码：

```
CFoo::Foo()
{
    EnterCriticalSection(&_m_CFooLock);
    m_count++;
    LeaveCriticalSection(&_m_CFooLock);
    m_pIBar->Bar();
    EnterCriticalSection(&_m_CFooLock);
    m_list->Add(…);
    …
    LeaveCriticalSection(&_m_CFooLock);
    EzAPI(…);
    EnterCriticalSection(&_m_CFooLock);
    LeaveCriticalSection(&_m_CFooLock);
}
```

Sychronized 锁的粒度较大，存在效率问题，也容易死锁。

Sequenced 与函数已有的 Mutex 混用，会导致加锁顺序混乱，有潜在死锁的危险。

7.2.3 synchronized & sequenced 实例讲解

synchronized：

上面的 synchronizedDemo.car 构件编译生成后的 cpp 文件编写如下：

```
#include "CFoo.h"
#include "_CFoo.cpp"

#define Sleep(ms) CThread::Sleep(ms, NULL)

ECode CFoo::Foo1()
{
    // TODO: Add your code here
    CConsole::WriteLine("Enter CFoo::Foo1()");
```

```
    Sleep(20);
    CConsole::WriteLine("Leave CFoo::Foo1()");
    return NOERROR;

}

ECode CFoo::Foo2()
{
    // TODO: Add your code here
    CConsole::WriteLine("Enter CFoo::Foo2()");
    Sleep(20);
    CConsole::WriteLine("Leave CFoo::Foo2()");
    return NOERROR;

}
```

编写客户端代码如下：

```
#include "synchronizedDemo.h"
using namespace Elastos;

//线程函数
ELFUNC Thread1(void *pArg)
{
    IFoo *pIFoo = (IFoo *)pArg;
    for (int n = 0; n < 2; n++) {
        pIFoo->Foo1();
    }
    return NOERROR;
}

int main()
{
    ECode ec;
    IFoo* pIFoo;
    IThread *pThread;
    int n;

    //创建一个构件对象
    ec = CFoo::New(&pIFoo);
    if (FAILED(ec)) return ec;

    //创建一个线程
```

```
        ec = CThread::New(Thread1, pIFoo, 0, &pThread);
        if (FAILED(ec)) {
            pIFoo->Release();
            return ec;
        }

        for (n = 0; n < 2; n++) {
            pIFoo->Foo2();
        }
        pThread->Join(INFINITE, NULL);
        pThread->Release();
        pIFoo->Release();
        return NOERROR;
    }
```

运行后的结果为：

```
Enter CFoo::Foo2()
Leave CFoo::Foo2()
Enter CFoo::Foo1()
Leave CFoo::Foo1()
Enter CFoo::Foo2()
Leave CFoo::Foo2()
Enter CFoo::Foo1()
Leave CFoo::Foo1()
```

由上可以看出，当调入一个方法比如 Foo2 之后，直到该方法执行完才可能会执行 Foo1。

sequenced：

如果将 foo.car 构件中 CFoo 的属性改为[sequenced]，不改变其他的文件，编译执行客户端代码如下：

```
Enter CFoo::Foo2()
Enter CFoo::Foo1()
Leave CFoo::Foo2()
Leave CFoo::Foo1()
Enter CFoo::Foo2()
Enter CFoo::Foo1()
Leave CFoo::Foo2()
Leave CFoo::Foo1()
```

由上可以看出，当调入一个接口方法比如 Foo2 之后，该方法未执行完就可能会执行 Foo1。

7.3 private

此属性只能用于修饰 class，指定该类只能在构件内被实例化，换句话说，客户端程序不能实例化该构件类。构件编写者要明确知道哪些构件是给外部用的，哪些是外部用不到的，外部用不到的构件，要将其定义为 private。例如：

```
module
{
    interface IFoo {
        Foo();
    }

    class CFoo {
        interface IFoo;
    }

    [private]
    class CBar {
    interface IFoo;
    }
}
```

上述 car 构件定义了两个类 CFoo 和 CBar，由于 CBar 具有 private 属性，因此不能在客户端调用 CBar::New（&pIFoo）创建对象，只能在服务器端创建。而 CFoo 对象在客户端和服务端均可创建。

7.4 local

此属性有两种用法，一种用于修饰接口。

如果要取得的接口的属性为 local，标识提供该接口的类的类对象与客户程序只能在相同的域内使用，不能用于远程调用。例如：

```
module
{
    [local]
    interface IFoo {
        Foo ();
    }

    interface IBar {
        Bar ();
    }
```

```
        [local]
        class CFoo {
            constructor();
            constructor([in] Int32 id);
            constructor([in] PVoid param); // 非自描述类型
            interface IFoo;
        }

        class CFooBar {
            interface IFoo;
            interface IBar;
        }
    }
```

示例构件 LocalDemo 中，定义的接口 IFoo 使用了 local 属性，所以不能使用 NewInContext（CTX_DIFF_DOMAIN，&pFoo）来创建对象。

但是当客户程序要取得的接口为非 local 的接口 IBar 时，可以在不同的域内实例化 CClass 类的对象，也就是可以使用 NewInContext（CTX_DIFF_DOMAIN，&pBar）来创建对象，只不过这时具有 local 接口的 IFoo 就不能被查询出来。

此属性的另外一种用法是修饰 class 类。

用于修饰带有 constructor（内有非自描述参数）的类，如果这样的类不用 local 修饰，编译器会报 warning，并自动把生成的类厂接口转为 local，如给这样的类加上 local，则可去除该 warning。用法如下：

```
    [local]
    class CFoo {
        constructor();
        constructor([in] Int32 id);
        constructor([in] PVoid param); // 非自描述类型
        interface IFoo;
    }
```

加了 local 属性的类，其(全部)构造函数就不能远程。也就是 NewInContext 的第一个参数不可以是 CTX_DIFF_DOMAIN,否则的话就返回 E_CONFLICT_WITH_LOCAL_KEYWORD。

在构件定义中,某接口参数或者 constructor 的参数使用非 CAR 支持的数据类型或者不可以用于列集/散集的数据类型，编译时，编译器将自动强制给该接口或者类厂接口加上 local 属性。

7.5 deprecated

此属性关键字用于兼容移植来的第三方软件中不符合 CAR 规范的数据类型使用。

“deprecated”修饰的其实是方法中的参数的数据类型，表示该数据类型的此种使用方法非常规用法，它与方法、接口能否被 marshal 无关，也与 class 是否是 local 或 private 的也无关。同样的，此关键字也可以修饰构件类，用于兼容用户 constructor 方法中参数的不规范的用法。

语法为：

```
[deprecated]
interface InterfaceName InterfaceBody
| class ClassName ClassBody
```

使用说明：

1. deprecated 属性不能修饰 aspect 类。

2. 详细使用规则请参见数据类型修饰方法参数时的使用规则一览表。

使用范例：

```
[local, deprecated]
interface IFoo {
    Foo([in] UInt32 * ID);
}
```

其中 UInt32 修饰输入参数，推荐的用法是不加“*”，但是它加了，为了让程序员意识到自己非常规的用法，我们要求加上“deprecated”。

7.6 applet

此属性用于修饰类，表示该类接口方法的调用具有类似于 applet 的接口方法行为。

语法：

```
[applet]
class className classBody
```

使用范例：

```
interface IBar {
    Bar();
}

[applet]
class CBar {
    interface IBar;
}
 ...
IBar *pBar;
ECode ec = CBar::New(&pBar);
pBar->Bar();  //此调用类似于 applet 接口方法行为
...
```

如果 CBar 没有 applet 属性，那么这是一个普通的接口方法的调用。

上述代码片断中，对 pBar->Bar()的调用，会转变为一个事件被抛到消息队列中，最终由 pBar 创建时所在的 applet 来完成实际的调用。

7.7 in & out

in 属性和 out 属性只能用于修饰接口方法参数，in 标识该参数为输入参数， out 标识该参数为输出参数。参数由客户程序（Client）分配和释放。

语法为：[in] 参数类型 参数名。

[out] 参数类型 参数名

例如：

```
module
{
    interface IHello {
        Hello([in] Int32 n , [out] AStringBuf<> asbName);
    }

    class CHello {
        interface IHello;
    }
}
```

示例构件 InOutDemo 中，接口 IHello 中 Hello 方法的参数 n 为输入参数,asbName 为输出参数。

注意：1. 不支持参数同时具有[in]属性和[out]属性。

2. 参数的 in、out、callee 等属性必须显式指明，否则编译器报错。具体使用规则请参见数据类型修饰方法参数时的使用规则一览表。

7.8 callee

此属性用于修饰 car 文件中的接口方法参数，表示该参数由被调方程序分配。

语法：

```
[out,callee] type parameterName
```

使用说明：

1. 该属性与 in 属性,out 属性一样，均用于修饰 CAR 文件中的方法参数。

2. 该属性只能用于 local 情况下，且只能修饰五元组数据类型，使用时必须与[out]属性同时使用。

3. 具体使用规则请参见数据类型修饰方法参数时的使用规则一览表。

使用范例：

```
module
{
    interface IHello {
```

```
            Hello([in] Int32 n, [out, callee] AStringBuf<> * asbName);
        }

        class CHello {
            interface IHello;
        }
    }
```

第八章　CAR 构件编程基础

8.1　构件实例化

现有一构件的 CAR 文件如下：

```
module
{
    interface IChild{
        HelloChild();
    }
    class CChild{
        constructor([in]Int32 age, [in]AString asName);
        interface IChild;
    }
}
```

当在构件外部（也就是在客户端而不在该构件 dll 内）创建并获取的构件对象实例时，同时客户端与该构件所在的服务器端属于同一个进程空间，那么就采用函数 New 来实例化构件。

在 Server 端生成的代码框架中，会为 C++类（class CChild）生成 New 方法，供 Client 端调用：

```
class CChild
{
public:
    static ECode New(
        /*[in]*/  Int32 num,
        /*[in]*/  AString asName,
        /*[out]*/ IChild **pIChild)
    {
        ……
    }
    ……
}
```

在 Client 端引用时的写法：

```
#include "NewDemo.h"
using namespace Elastos;
```

```
int main()
{
    IChild *pIChild;

    //创建一个构件对象
    ECode ec = CChild::New(5, "lanlan", &pIChild);
    if (FAILED(ec)) return ec;

    pIChild->HelloChild();
    pIChild->Release();
    return NOERROR;
}
```

New 的参数集是由构造函数（constructor）所有参数加上一个接口指针的引用构成的。如果一个构件类有 m 个构造方法和 n 个接口（目前不包括事件接口），那么就有 m*n 个 New 方法，用户可用任意的构造函数来创建出该类任意的接口。

如果没有构造函数，生成的 New 方法如下：

```
class CChild
{
public:
    static ECode New(
        /*[out]*/ IChild **pIChild)
    {
        ……
    }
    ……
}
```

在 Client 端引用时的写法：

```
IChild *pIChild;
//一个接口参数
ECode ec = CChild::New(&pIChild);
if (FAILED(ec)) return ec;
pIChild->HelloChild();
pIChild->Release();
```

在 New 方法里实际调用顺序是先创建对象，然后调用服务端相应的 constructor。

当被创建的构件类被 singleton 属性修饰时，用 AcquireSingleton 代替 New 来实例化构件对象，除构造函数不能有参数外，其它用法同 New 一样。

8.2 查询接口

8.2.1 Probe

要想调用 CAR 构件的方法，必须得到定义该方法的接口指针。用户只要得到构件的一个接口指针，那么就可以通过 Probe 方法得到任一个该构件的其它接口指针。

现有一构件的 CAR 文件如下：

```
module
{
    interface IChild1{
        HelloChild1();
    }
    interface IChild2{
        HelloChild2();
    }
    class CChild{
        constructor([in]Int32 age, [in]AString asName);
        interface IChild1;
        interface IChild2;
    }
}
```

在 Client 端使用 Probe 方法的写法：

```
IChild1 *pIChild1;
IChild1 *pIChild2;
//得到类对象的指针
ECode ec = CChild::New(&pIChild1);
if(FAILED(ec)) {
        ……
}
//通过 IChild1 接口指针查询出 pIChild2 接口指针
pIChild2= IChild2::Probe(pIChild1);
pIChild2->HelloChild2();  // 调用 pIChild2 接口中定义的 HelloChild2 方法

pIChild1->Release();
```

Probe 返回的接口指针，接口引用计数没有 AddRef 过，这可以提高接口查询速度，因为没有 AddRef 过，自然就不用 Release 了。

8.3 析构 CAR 构件

8.3.1 引用计数

多个客户端使用同一 CAR 构件时，只要有客户要使用构件，需要保证构件对象不被销毁，而当没有客户使用构件时，要及时销毁构件对象，以回收资源。那么如何才能知道还有没有客户使用构件呢，在 CAR 构件中使用引用计数可以解决这个问题。

引用计数基本规则：

客户创建构件对象并获得了第一个接口指针后，引用计数应该为 1；

在客户程序中，当把接口指针赋给其他变量时，应该调用 AddRef，使用引用计数增 1；

在客户程序中，当一个接口指针被使用完后，应该调用 Release，使引用计数减 1。

8.3.2 Release

当一个接口指针被使用完后，应该调用 Release，使引用计数减 1。当引用计数的值为 0 时，CAR 构件自动析构。

CAR 构件所有接口的引用技术最终都关联到构件类的引用计数上，以 8.2.1 的例子来说，如下两行代码：

```
pIChild1->Release();
pIChild2->Release();
```

其实它们都调用了下述函数，每次都会检查 CChild 构件类的引用计数是否为 0，当为 0 时，析构该 CAR 构件。

```
UInt32 _CChild::Release()
{
    Int32 nRef = m_cRef.Decrement();
    if (0 == nRef) {
        ((CChild*)this)->_Uninitialize_();
#ifndef _NO_CCHILD_CLASSOBJECT_
        ((CChild*)this)->~CChild();
        EzTaskMemFree(this);
#else
        delete this;
#endif // _NO_ CCHILD_CLASSOBJECT_
    }
    return nRef;
}
```

8.4 与 Aspect 有关

aspect 关键字用来定义方面构件类，它是一种特殊的构件类实现，aspect 对象的特征可以被其它构件类对象聚合和拆卸，现有一构件的 CAR 文件如下：

```
module
{
    interface IHello1 {
        Hello1();
    }
    interface IHello2 {
        Hello2();
    }
    class CClass {
        interface IHello1;
    }
    aspect AAspect {
        interface IHello2;
    }
}
```

使用说明：

1. CAR 构件技术里只有 aspect 构件对象可以被聚合。
2. 不允许一个 aspect 构件类包含回调接口。
3. aspect 对象可以被其他构件对象聚合，但是它不可以聚合其它 aspect 对象。
4. aspect 类中不允许定义带参数的 constructor,也就是 aspect 对象不可以被单独的 New 出来。

8.4.1 Attach

使用 aspect 对象的 Attach 函数可以使 aspect 对象被其它构件类对象聚合，其函数声明如下：

```
ECode AAspect::Attach(IObject* pObj);
```

在调用 Attach 方法期间，会自动调用如下两个函数，用户可重载它们来满足对对象进行控制的需要：

```
virtual CARAPI OnAspectAttaching(PObject pAspect) //对于每个 class 类，聚合时调用。
virtual CARAPI OnAggregated(PObject pOuter) //对于每个 aspect 类，聚合后调用。
```

Client 端编写代码如下：

```
$using server.dll;
#include <stdio.h>
```

```
Int32 main()
{
    ECode ec;
    IHello1* pIClass = NULL;
    IHello2* pIAspect = NULL;

    ec = CClass::New(&pIClass);
    if (FAILED(ec)) {
        printf("CClass::New Error, ec = %x\n", ec);
        goto Exit;
    }
    //AAspect 方面构件类被 CClass 构件类聚合
    ec = AAspect::Attach(pIClass);
    if (FAILED(ec)) {
        printf("Aggregate failed!\n");
        goto Exit;
    }

    ec = IHello2::Query(pIClass, &pIAspect);
    if (FAILED(ec)) {
        printf("Query failed!\n");
        goto Exit;
    }

    printf("\nAspect:\n");
    pIAspect->Hello2();
    pIAspect->Release();

Exit:
    if (pIClass) {
        pIClass->Release();
    }
    if (pIAspect) {
        pIAspect->Release();
    }
    return 0;
}
```

从程序运行结果可以看出，AAspect 方面构件类被 CClass 构件类聚合，用户可以通过 CClass 的接口指针 IHello1 查询出 AAspect 的接口指针 IHello2，从而调用 IHello2 里的方法 Hello2()。

8.4.2 Detach

使用 aspect 对象的 Detach 函数可以使 aspect 对象被已聚合它的构件类对象拆卸，其函数声明如下：

```
Ecode AAspect::Detach(IObject* pObj);
```

在调用 Detach 方法期间，会自动调用如下两个函数，用户可重载它们来满足对对象进行控制的需要：

```
virtual CARAPI OnAspectDetached(PObject pAspect) //对于每个 class 类，拆卸后调用。
virtual CARAPI OnUnaggregated(PObject pOuter) //对于每个 aspect 类，拆卸后调用。
```

Client 端编写代码如下：

```
$using server.dll;
#include <stdio.h>

Int32 main()
{
    ECode ec;
    IHello1* pIClass = NULL;
    IHello2* pIAspect = NULL;

    ec = CClass::New(&pIClass);
    if (FAILED(ec)) {
        printf("CClass::New Error, ec = %x\n", ec);
        goto Exit;
    }
    //AAspect 方面构件类被 CClass 构件类聚合
    ec = AAspect::Attach(pIClass);
    if (FAILED(ec)) {
        printf("Aggregate failed!\n");
        goto Exit;
    }

    ec = IHello2::Query(pIClass, &pIAspect);
    if (FAILED(ec)) {
        printf("Query failed!\n");
        goto Exit;
    }

    printf("\nAspect:\n");
    pIAspect->Hello2();
```

```
        pIAspect->Release();
        //AAspect 方面构件类被 CClass 构件类拆卸
        ec = AAspect::Detach(pIClass);

        ec = IHello2::Query(pIClass, &pIAspect);
        if (FAILED(ec)) {
            printf("Query failed!\n");
            goto Exit;
        }

        printf("\nDetach Aspect:\n");
        pIAspect->Hello2();

    Exit:
        if (pIClass) {
            pIClass->Release();
        }
        if (pIAspect) {
            pIAspect->Release();
        }
        return 0;
    }
```

从程序运行结果可以看出，AAspect 方面构件类被 CClass 构件类拆卸后，用户试图通过 CClass 的接口指针 IHello1 查询 AAspect 的接口指针 IHello2 时失败。

第九章　CAR 构件运行基础

9.1　构件模块、构件类、接口

9.1.1 构件模块

CAR 构件由其所定义的接口、构件类及相关元数据组成，并以构件模块为封装和存储单位,并通过 dll 实现(每个 CAR 构件模块都是一个 dll)。构件模块由 uunm(Universal Unique Name）唯一标识，在本机则以存储的文件名为标识。uunm 是一串 unicode 字符串，其格式如下：

[length][\000][URL][CAR module name]

length 是 uunm 的长度, URL 描述了 CAR module 的 WEB 路径，CAR module name 是 CAR 构件模块的名字，在本机上， URL 可以不存在。如对于构件模块 hello.dll 其 car module 的 uunm 为：L"\x29\000www.elastos.com.cn/car/hello.dll"在本机上为 L"\x12\000hello.dll"。

uunm 是 CAR 编译器自动生成的，它使得 CAR 构件的使用者不必关心 CAR 构件的所在位置。

9.1.2 接口、构件类、构件对象

接口是一组逻辑上相关的函数集合,是构件特征的抽象定义,是最基本的构件使用单位。

构件类是最基本的构件运行实体（指构件对象），一个构件模块可以封装一到多个构件类的实现。构件类的实例是构件对象， 构件对象是接口的实现，一个构件对象可以实现多个接口，一个接口可以被多个构件对象实现。

IID 是接口的标识，它由接口名、接口的方法名以及父接口名等等诸多因素决定，CLASSID 是构件类的标识(CLASSID 由构件类名字决定),它在构件模块内唯一。IID 和 CLASSID 都是由 CAR 编译器自动生成，构件编写者只需定义接口名和构件类名就可以了。

IID 是一个 128 位的全局唯一标志符，CAR 构件客户通过 IID 获得接口指针，再通过接口指针调用接口服务。如接口 IHello 的 IID：

```
static const InterfaceId InterfaceId_IHello = \
{0xB7E30C02,0x8452,0xAFBE,{0xFC,0x47,0xE8,0xFB,0xCA,0x52,0x84,0xBE}};
```

一般地，接口的 IID 名称通常由 IID_和接口名组成。

CLASSID 标志了构件类，不同于接口的标志， CLASSID 不但包含了 128 位的全局唯一标志符(CLSID)，还包含了构件模块的 uunm：

[CLSID][uunm]

如构件类 CHello 的 CLASSID：

```
static const CLASSID CLSID_CHello = {
```

{0x97E90C1D,0x844C,0xAFBE,{0xFC,0xAF,0xE8,0xFB,0xCA,0x4C,0x84,0xBE}},
L"\x29\000www.elastos.com.cn/car/hello.dll};

一般地，构件类的 CLASSID 名由 CLSID_和构件类名组成。

客户端在实例化某个构件类时，如通过 New 函数，客户传入构件类的 CLASSID,CAR 构件加载程序会根据其中的 uunm 自动去加载相应的 CAR 构件，然后根据 CAR 构件的元数据创建出构件对象。不象 MSCOM,所有的 COM 对象的 CLSID 都是通过注册表来管理，客户必须先安装构件，注册 CLSID，而 CAR 构件对本身有足够的自描述能力，客户通过 CAR 构件运行平台使用 CAR 构件服务，直接使用即可，根本无需知道软件（CAR 构件）的存在。

9.1.3 接口的二进制结构

接口对客户来说只是功能上的描述，其实现是被构件类实现的。一个构件类可以实现多个接口， 一个接口可被多个构件类实现， 二者是多对多的关系。下面我们就这种关系讨论接口的内存结构。

1.构件类实现一个接口的内存结构

考虑接口 IHello, 它有一个函数 Hello，采用类似于 COM 的 IDL 描述如下：

```
interface IHello
{
    Hello();
}
```

若此接口被构件类 CHello 实现，C++语言描述如下：

```
CarClass(CHello)
{
    public:
        CHello();
        ~CHello();
    public:
        CARAPI Hello();
    private:
        char * m_helloChars;

    //other members
……
}
```

则 CHello 的构件类对象的内存结构如图 9.1 所示。

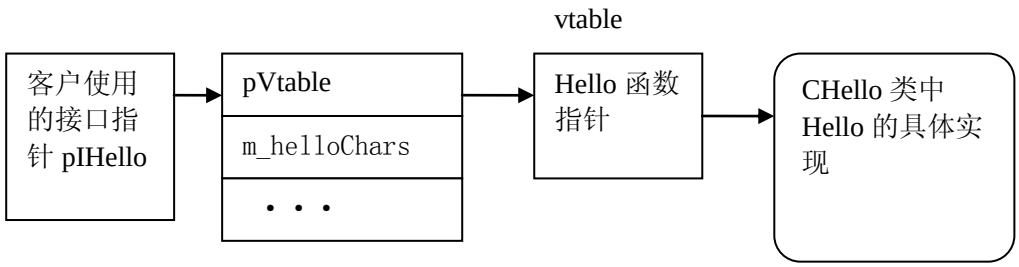


图 9.1 构件类对象的内存结构 1

2. 构件类实现多个接口的内存结构

考虑接口 IWorld：

```
interface IWorld
{
    World();
}
```

若 CHello 实现了 IHello 和 IWorld:

```
CarClass(CHello)
{
    public:
        CHello();
        ~CHello();
    public:
        CARAPI Hello();
        CARAPI World();
    private:
        char * m_helloChars;
        char * m_worldChars;

    //other members
    ……
}
```

则 CHello 构件类对象的内存结构如图 9.2 所示。

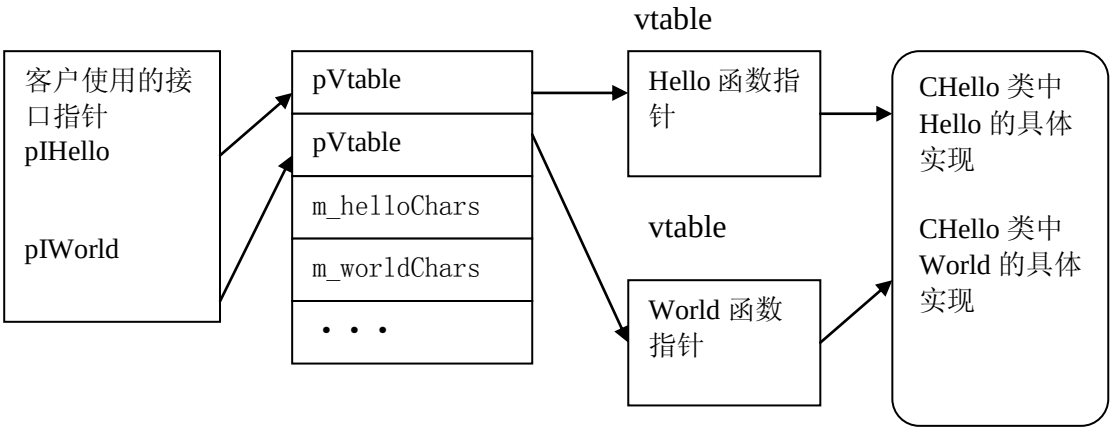


图 9.2 构件类对象的内存结构 2

3.一个接口被多个构件类实现的内存结构

如果接口 IHello 不但被构件类 CHello 实现了，而起还被另外一个构件类 COtherHello 实现(COtherHello 和 CHello 可以在一个构件模块中，也可以存在于不同的构件模块)，则这两个构件类对象的内存结构如图 9.3 所示。

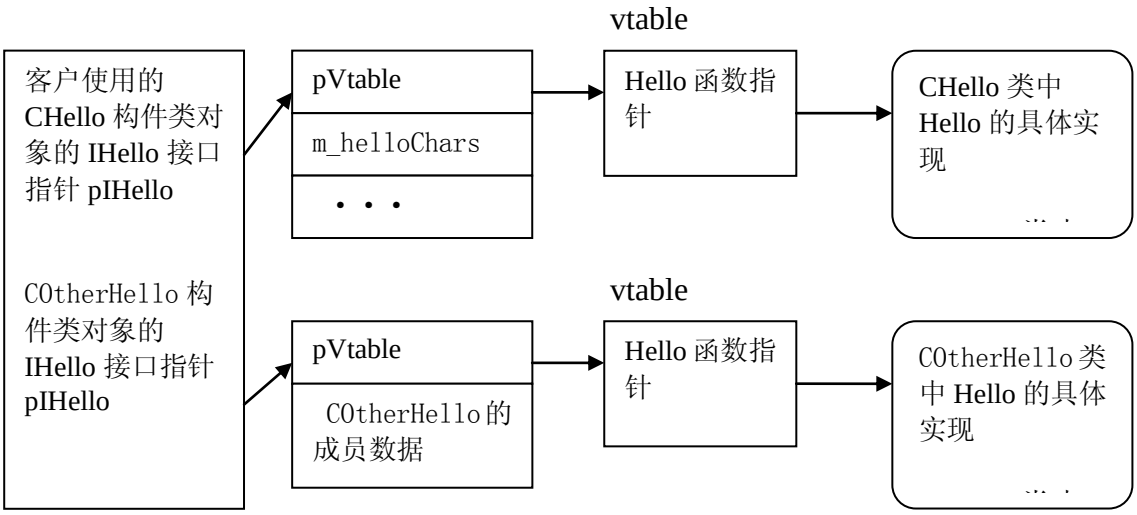


图 9.3 构件类对象的内存结构 3

9.2 CAR 对象与 C++对象的比较

尽管 CAR 对象建立在二进制一级的基础上，而 C++对象建立在源代码一级的基础上，但从特性上，我们仍可以作一比较。

1. 封装特性

数据封装是两者都具有的特性，但其形式有所不同。在 CAR 对象中，数据是完全封装在对象内部的，外部不可能直接访问对象的数据属性，因为 CAR 对象和客户程序可能在不同的模块中甚至在不同的进程中或不同的机器上，因此，客户直接访问 CAR 属性不仅不合理，有

时也不太可能。而且，通过 CAR 对象提供的接口成员函数访问对象的属性，为 CAR 对象对属性的控制提供了机会，对象可以在成员函数中对新的属性值进行有效性判断，若新值合理则接受，否则便拒绝，还可以引发一些相应的事件。下面的代码逻辑可以说明这种用法：

```
HRESULT CXXX: : SetProperty(int newValue) {
if(newValue is valid){
    MyProperty=newValue;
}else{
FireEvent(“The newValue is invalid!”)
}
}
```

C++对象的封装特性与 CAR 对象的封装有所不同，因为 C++对象的使用者与对象往往在同一个程序模块中(至少在同一个进程中)，所以使用者有可能直接访问对象中的数据成员，因此 C++语言对类(class)的成员访问进行控制，类的成员分别为公共数据成员(public)、私有数据成员(private)、保护数据成员(protected)。私有数据成员只能在对象内部直接访问，在对象外部是访问不到的，而公共数据成员可以在对象外部直接访问，保护数据成员可以在其派生类的成员函数中访问。

对于这两种情况的封装特性，我们可以这样来理解，CAR 对象的数据成员的封装以构件模块为最终边界，对于对象用户是完全透明的、不可见的；而 C++对象的封装特性只是语义上的封装，对于对象用户是可见的。

2. 可重用性

可重用性是面向对象系统的重要特性，因此也是 CAR 对象和 C++对象的共同特性，但两者的表现形式不同，CAR 对象的可重用性表现在 CAR 对象的包容和聚合，一个对象可以完全使用另一个对象的所有功能；而 C++对象的可重用表现在 C++类的继承性，派生类可以调用其父类的非私有成员函数。

一个 CAR 对象 A 如果要使用另一个 CAR 对象 B 的功能，则可以通过两种方式实现：包容或聚合。不管哪种形式，CAR 对象 A 都可以完全重用对象 B 的功能，就如同对象 A 自己实现了对象 B 的功能，而且，当对象 B 更新了版本或者增强了功能时，对象 A 自动使用新版本的对象 B，而根本不需要重新编译或者重新设置，因此，CAR 对象的重用是动态的，可以在对象 A 和对象 B 完全独立的情况下，对象 A 重用对象 B 的功能。

C++对象的重用性表现在源代码一级的重用性上，通过 C++语言的类继承来实现。从类 A 派生得到的类 B 可以继承类 A 的所有非私有成员，包括数据成员和函数成员。但类 B 与类 A 有紧密的派生和继承关系，当类 A 的实现作了修改时，则类 B 必须重新编译或者需要修改相应的代码，然后才能适应新的类 A。而且，在最终得到的可执行代码中，类 A 和类 B 在同一模块中，重用性只体现在程序模块的内部，对模块外部而言。重用性只体现在对代码的有效管理上。

C++类和对象重用性应用最广泛的是类库，例如 Microsoft Visual C++提供的 MFC 库以及 Borland C++提供的 OWL(Object Windows Library，面向对象的窗口库)类库是典型的例子，它们为 Windows 应用的开发提供了最基本的代码，开发人员直接使用这些类库就可以做出一些基本的 Windows 应用来。一个有良好习惯的 C++程序员可以在长期工作中形成自己的类库，新应用的开发可以使用以前积累下来的类代码，这也是 C++类重用的一个有意义的用途。

虽然 CAR 对象和 C++对象的重用性层次不同、机制不同，但由于在 C++语言中，通常用类来实现 CAR 构件对象，因此，这两种重用机制对我们都会有用。在源代码一级可以使用 C++类的重用性，在构件一级使用 CAR 对象的重用性。

除了封装特性和重用特性，C++对象还有一个重要特性是多态性。正是 C++对象的多态性，才体现了 C++语言用类描述事物的高度抽象的特征；CAR 对象也具有多态性，但这种多态性需要通过 CAR 对象所具有的接口才能体现出来，就像 C++对象的多态性需要通过其虚(virtual)函数才能体现一样。

9.3 CAR 接口

在前面的论述中，我们一再提到 CAR 接口，因为 CAR 对象的客户与对象之间通过接口进行交互，所以构件之间接口的定义至关重要。CAR 规范的核心内容是关于接口的定义，虽然 CAR 接口本身并不复杂，但围绕 CAR 接口有很多内容值得仔细探讨，包括接口的标识、接口函数的调用习惯、参数处理、接口与对象的关系以及接口与 C/C++的关系、CAR 接口所具有的特性等。

9.3.1 从 API 到 CAR 接口

假如我们要实现一个字处理应用系统，它需要一个查字典的功能，按照构件化程序设计的方法，自然应该把查字典的功能放到一个构件程序(.dll)中实现。如果以后字典程序的查找算法或者字典库改变了，只要应用程序与构件程序之间的接口不变，则新的构件程序仍然可以被应用系统使用。这就是采用构件程序带来的灵活性。

为了把应用系统和构件程序连接起来，又能使它们协同工作，最简单的做法就是先定义一组查字典的函数，而且这组函数尽可能一般化，不要加入特定的与字典库相关的知识。我们可以按表 2.1 列出的函数来定义字典库的功能。

表 9.1 字典库 API 函数

函 数	功能说明
Initialize	初始化
LoadLibrary	装入字典库
InsertWord	插入一个单词
DeleteWord	删除一个单词
LookupWord	查找单词
RestoreLibrary	把内存中的字典库存人指定的文件中
FreeLibrary	释放字典库

在字典构件程序中，我们可以这样定义 API 接口函数：

```
BOOL EXPORT Initialize();
BOOL EXPORT LoadLibrary(char*);
BOOL EXPORT InsertWord(char*, char*);
void EXPORT DeleteWord(char*);
```

```
BOOL EXPORT LookupWord(char*, char**);
BOOL EXPORT RestoreLibrary(char*);
void EXPORT FreeLibrary();
```

通过这些函数，建立了主应用程序和字典构件程序之间的连接，当应用需要查字典功能时，它可以直接调用这些函数以完成必要的操作；而且对于构件程序来说，由于它提供了常用的查字典和动态维护字典库的基本功能，所以该库也可以用于其他的应用，这也充分体现了构件程序的重用性，但这种重用性要建立在良好的接口内涵基础上。

图 9.4 显示了这种 API 接口的示意图，应用程序或者构件程序通过一个平面结构的 API 层与字典构件进行交互。我们经常可以见到这种接口形式，尤其在一些软件开发包(SDK)中最为常见，比如早期的 Windows API、一些硬件厂商提供的软件开发包等。

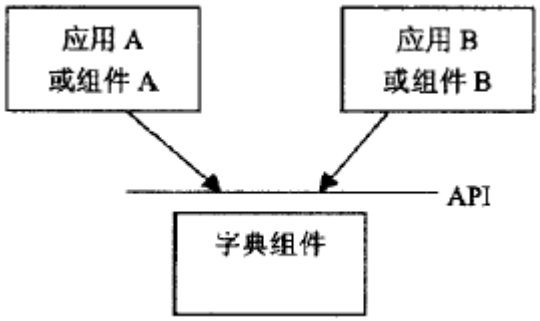


图 9.4 应用程序 A 和 B 通过 API 接口共用一个字典构件

平面型的 API 接口层可以很好地把两个程序连接起来，但存在以下一些问题：

(1)当 API 函数非常多时，使用会非常不方便，需要对函数进行组织。例如 Windows API 有 300 多个函数，一般的 SDK 也有几十个之多，编程“接口面”太宽不利于接口层的管理。

(2)API 函数需要标准化，按照统一的调用方式进行处理，以适应不同的语言编程实现。参数的传递顺序、参数类型、函数返回处理（如是调用者，还是被调用者负责维护栈）都需要标准化。 CAR 定义了一套完整的接口规范，不仅可以弥补以上 API 作为构件接口的不足，还充分发挥了构件对象的优势，并实现了构件对象的多态性。

9.3.2 接口定义和标识

从技术上讲，接口是包含了一组函数的数据结构，通过这组数据结构，客户代码可以调用构件对象的功能。接口定义了一组成员函数，这组成员函数是构件对象暴露出来的所有信息，客户程序利用这些函数获得构件对象的服务。

客户程序用一个指向接口数据结构的指针来调用接口成员函数。如图 9.5 所示，接口指针实际上又指向另一个指针，这第二个指针指向一组函数，称为接口函数表，接口函数表中每一项为 4 个字节长的函数指针,每个函数指针与对象的具体实现连接起来。通过这种方式，客户只要获得了接口指针，就可以调用到对象的实际功能。

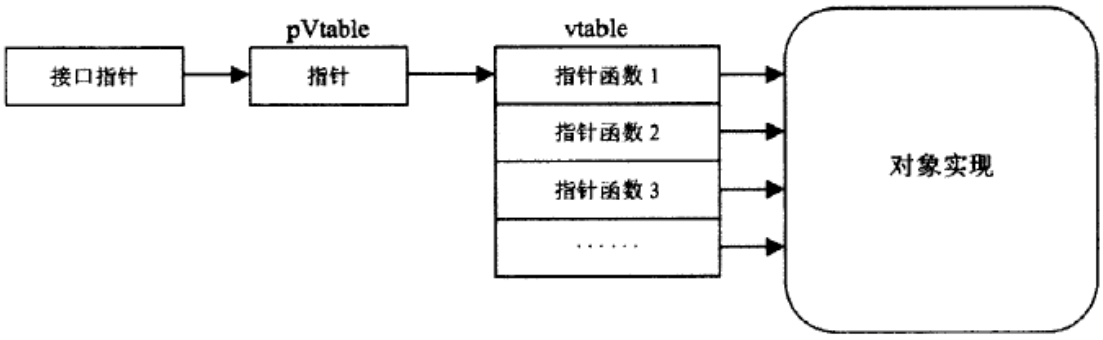


图 9.5 接口结构

通常，我们把接口函数表称为虚函数表(virtual. function table，简称 vtable)，指向 vtable 的指针为 pVtable。 对于一个接口来说，它的虚函数表 vtable 是确定的，因此接口的成员函数个数是不变的，而且成员函数的先后顺序也是不变的；对于每个成员函数来说，其参数和返回值也是确定的。在一个接口的定义中，所有这些信息都必须在二进制一级确定，不管什么语言，只要能支持这样的内存结构描述，就可以定义接口。

例如，我们可以用 C 语言来描述前面提到过的字典接口：

```
struct IDictionaryVtbl;
struct IDictionary
{
    IDictionaryVtbl *pVtbl;
};
struct IDictionaryVtbl
{
    BOOL(*Initialize)(IDictionary *this);
    BOOL(*LoadLibrary)(IDictionary *this, String);
    BOOL(*InsertWord)(IDictionary *this, String, String);
    void(*DeleteWord)(IDictionary *this, String);
    BOOL(*LookupWord)(IDictionary *this, String, String*);
    BOOL(*RestoreLibrary)(IDictionary *this, String);
    void(*FreeLibrary)(IDictionary *this);
};
```

以上定义有几点需要说明：

(1)每一个接口成员函数的第一个参数为指向 IDictionary 的指针，这是因为接口本身并不独立使用，它必定存在于某个 CAR 对象上，因此，该指针可以提供对象实例的属性信息，在被调用时，接口可以知道是对哪个 CAR 对象在进行操作。所以，该 this 指针非常类似于 C++类成员函数定义中被隐藏的 this 指针。如果我们在一个应用系统中，同时用到了两本字典，即存在两个字典对象，则不同的字典对象，其 this 指针不同。在本节后面我们讲到接口内存模型时可以看得更清楚。

(2)在接口成员函数中，字符串变量必须用 Unicode 字符指针，CAR 规范要求使用 Unicode 字符，而且 CAR 库中提供的 CAR API 函数也使用 Unicode 字符。所以，如果在构件程序内部用到了 ANSI 字符的话，则应该进行两种字符表达的转换，操作系统或者 C/C++编译库会提

供这样的转换函数。当然，在读者既建立构件程序又建立客户程序的情况下，可以使用自己定义的参数类型，只要它们与 CAR 所能识别的参数类型兼容。

(3)不仅成员函数的参数类型是确定的，而且应该使用同样的调用习惯。客户程序在调用成员函数之前，必须先把参数压到栈中，然后再进入成员函数中，成员函数依次把参数从栈中取出来，在函数返回之前或返回之后，必须恢复栈的当前位置，才能保证程序正常运行。我们知道，在 Windows 平台上有两种调用习惯，分别为_cdecl 和_stdcall(在有的编译器中称为 pascal)，采用_cdecl 可以实现 C 语言中用到的函数可变参数的特性(例如 printf 函数)，因为在这种方式下，由调用程序处理栈的恢复。由于大多数语言(除 C/C++之外)都使用了_stdcall 或 pascal 调用习惯，而且大多数的系统 API(支持可变参数的函数例外)也都使用这种调用习惯，所以，CAR 规范也采用_stdcall 或 pascal 调用习惯，并且，所有的 CAR API 函数也使用了_stdcall 调用习惯。当然，调用习惯不是绝对的，但必须保证调用方和被调用方使用一致的调用习惯，如果接口成员函数使用了_cdecl 调用习惯，则 C/C++之外的大多数语言就不能使用这样的接口，所以，除非要使用可变参数特性，否则就使用_stdcall 调用习惯。

(4)在 C 语言中，用这种结构只是描述了接口，并没有提供具体的实现，对于客户程序，它只需要这样的描述，就可以调用 CAR 对象的接口；而对于构件程序，基于这样的描述，必须提供具体的实现过程，也就是说，如果一个 CAR 对象实现了这个接口，则它所提供的接口指针 IDictionary 所指向的 IDictionaryVtbl 结构中，每个成员必须是有效的函数指针。

(5)从 C 语言的描述我们也可以看出，由于 CAR 接口的这种二进制结构，只要一种编程语言能够支持“structure”或“record”类型，并且这种类型能够包含双重的指向函数指针表的成员，则它就可以支持接口的描述，从而可以用于编写 CAR 构件或者使用 CAR 构件。

因为接口被用于构件程序和客户程序之间的通信桥梁，所以接口应该具有不变性，一个 CAR 对象可以支持很多个接口。例如，字典对象除了支持前面提到的 IDictionary 接口，还可以支持拼写检查接口。ISpellCheck，这样实际上就把平面型的 API 接口按照实际用途分成了几组。但客户程序如何来标识一个接口呢?类似于 CAR 对象的标识方法，CAR 接口也采用了上节提到的全局唯一标识符，它被称为接口标识符(InterfaceId，interface identifier)。例如：

```
extern“C”const InterfaceId InterfaceId_IUnknown =
{0X00000000，0x0000，0x0000，
{0xc0，0x00，0x00，0x00，0x00，0x00，0x00，0x46}};
```

如果客户程序要使用一个 CAR 对象的某个接口，则它必须知道该接口的 IID 和接口所能够提供的方法(即接口成员函数)。最新的 CAR 技术已经不需要用户了解这些 IID 信息了，编译器会根据元数据自动生成与管理这类信息。

9.3.3 用 C++语言定义接口

如果读者熟悉 C++语言 class 的实现机理，则不难发现，CAR 接口结构中的 vtable 与 class 的 vtable(类的虚函数表)完全一致，因此，用 class 描述 CAR 接口是最方便的手段。

我们可以用 C++类来重新定义 IDictionary：

```
class IDictionary
{
```

```
        virtual BOOL Initialize()=0;
        virtual BOOL LoadLibrary(String)=0;
        virtual BOOL InsertWord(String, String)=0;
        virtual void DeleteWord(String)=0;
        virtual BOOL LookupWord(String, String*)=0;
        virtual BOOL RestoreLibrary(String)=0;
        virtual void FreeLibrary()=0;
        };
```

因为 class 定义中隐藏了虚函数表 vtable，并且，每个成员函数隐藏了第一个参数 this 指针，this 指针指向类的实例。图 9.6 显示了类 IDictionary 的内存结构：

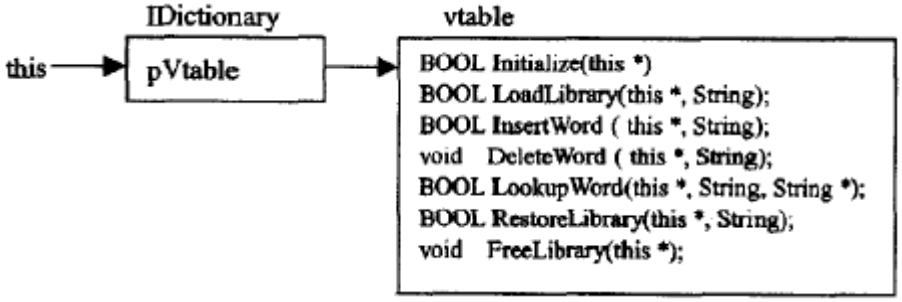


图 9.6 类 IDictionary 的内存结构

从图 2.3 我们可以看出，类 IDictionary 的内存结构与 CAR 接口规范所要求的完全一致。而 class IDictionary 的说明则比 struct IDictionary 的说明显然要简捷得多。

class IDictionary 的说明使用了纯虚函数，因为接口只是一种描述，并不提供具体的实现过程。如果 CAR 对象要实现接口 IDictionary，则 CAR 对象必须以某种方式把它自身与类 IDictionary 联系起来，然后把 IDictionary 的指针暴露给客户程序，于是客户程序就可以调用该对象的字典功能了。

我们再来看客户端调用的情形，当客户程序获得了某个字典对象的接口指针 pIDictionary 之后，它就可以调用接口的成员函数，例如：

pIDictionary —>LoadLibrary(“Eng_Ch.dict”)；

如果使用 C 语言的 struct IDictionary，则对接口成员函数的调用应该这样：

pIDictionary —>pVtbl —>LoadLibrary(pIDictionary, “Eng_Ch.dict”)；

由 C++语言 class 的特性可知上述两种调用完全等价。

9.3.4 接口的内存模型

实际上从图 2.2 我们已经看到了接口的二进制结构，但从图中我们看不到 CAR 接口与 CAR 对象的关系，虽然我们提到了每个成员函数的第一个参数是一个指向接口自身的 this 指针，this 指针可以为我们提供 CAR 对象的信息，但我们并不知道 this 指针如何与对象的状态信息联系起来。

CAR 对象往往有自己的属性数据，这些属性数据反映了对象的状态，也正是通过这些属性数据，才反映了此对象与彼对象的不同。例如，字典对象有一个字典数据表 m_pData 成员和字典文件名 m_DictFilename 作为其基本的属性数据。如果我们用 C++语言来实现字典对象，则可以用以下的类定义字典对象：

```
class CDictionary: public IDictionary
{
    public:
    CDictionary();
    ~CDictionary();
    public:
    virtual BOOL Initialize();
    virtual BOOL LoadLibrary(String);
    virtual BOOL InsertWord(String, String);
    virtual void DeleteWord(String);
    virtual BOOL LookupWord(String, String*);
    virtual BOOL RestoreLibrary(String);
    virtual void FreeLibrary();
    private:
    struct DictWord *m_pData;
    char *m_DictFilename[128];
    private:
//other private helper function
……
};
```

按照类 CDictionary 的定义，则接口 IDictionary 和字典对象的内存结构变成图 9.7。

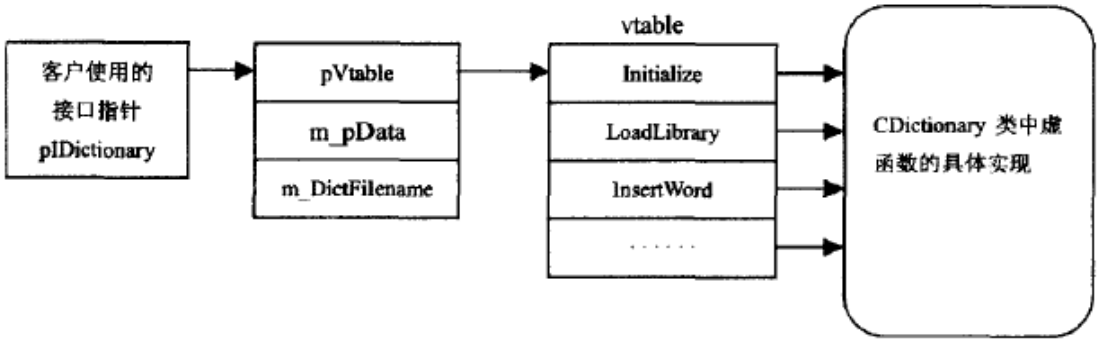


图 9.7 接口 IDictionary 与字典对象属性之间的结构关系

如果一个客户使用了两个字典对象，则显然两个字典对象公用了成员函数，但数据属性是不能公用的，根据 C++中 class 的基本编译原理，这时内存结构如图 9.8 所示。

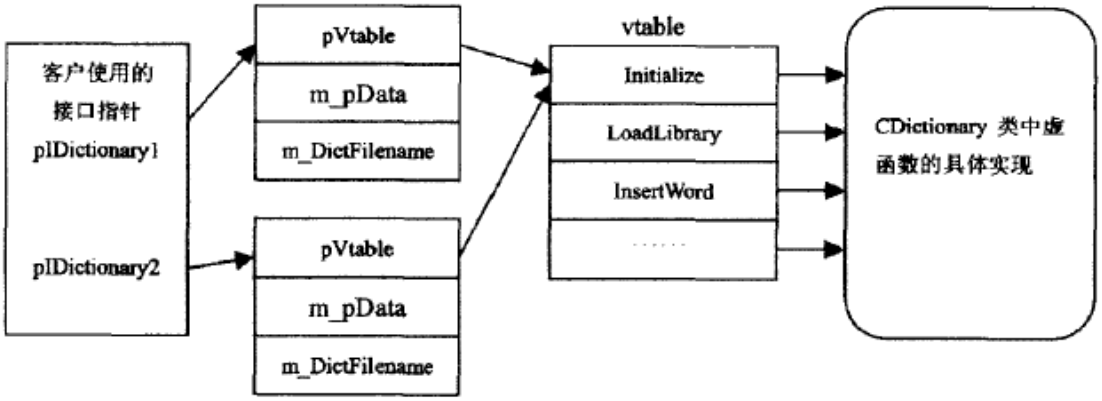


图 9.8 多个字典对象与接口 IDictionary 之间的结构关系

如果第二个字典构件对象没有采用 CDictionary 类的结构来实现其字典功能，但也实现了 IDictionary 接口，则此时内存结构与图 9.8 又有所不同，如图 9.9 所示。

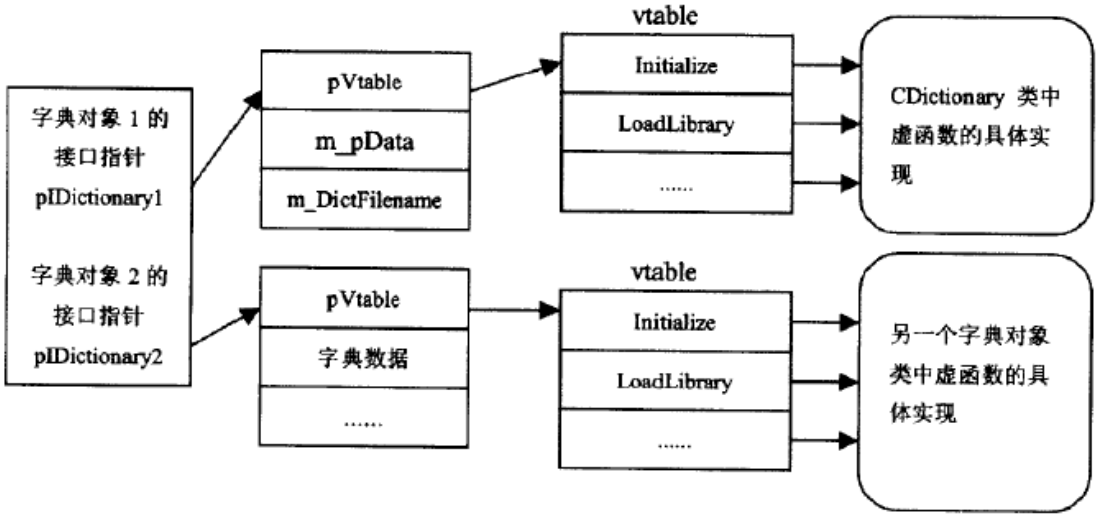


图 9.9 不同方法实现的两个字典对象与接口

在以上给出的三个模型图中，每个接口成员函数都包含一个 this 指针，通过该 this 指针，接口成员函数可以访问到字典对象的属性数据。按照 CDictionary 的定义方法，该 this 指 IDictionary 之间的结构关系针就是指向 CDictionary 类的对象，因此，在虚函数中可以直接访问 CDictionary 的数据成员。

我们并不一定非要用这种方式使字典对象支持 IDictionary 接口，但这种实现方法比较直观、简捷，而且把接口指针与对象数据绑在一起也易于理解。但实际上，我们也可以采用其他的方法来实现接口，只要接口成员函数中 this 指针(即接口指针)与对象数据能建立确定的连接，在接口成员函数中可以访问到对象数据即可。

9.3.5 接口的一些特点

本节前面的内容讲述了 CAR 接口的定义以及用 C++实现接口的方法，还介绍了接口的内存模型，最后我们再讨论一下接口所具有的一些特点。

1. 二进制特性

接口规范并不建立在任何编程语言的基础上，而是规定了二进制一级的标准。任何语言只要有足够的数据表达能力，它就可以对接口进行描述，从而可以用于与构件程序有关的应用开发。从前面的论述我们也可以看出，C++的 class 可以以最简捷的方式描述 CAR 接口，

而且，用 class 描述接口隐藏了接口的虚函数表以及接口成员函数的 this 指针参数，使得接口的定义更易于理解。

2．接口不变性

接口是构件客户程序和构件对象之间的桥梁，接口如果经常发生变化，则客户程序和构件程序也要跟着变化，这对于应用系统的开发非常不利，也不符合构件化程序设计的思想。因此，接口应该保持不变，只要客户程序和构件程序都按照既定的接口设计进行开发，则可以保证在两者独立开发结束后，它们的协作运行能力能达到预期的效果。当然，接口不变性就要求我们在定义构件对象的接口时，要充分考虑构件对象所提供功能的一般性特征，以使接口描述更为通用。很难给出接口设计的一般规则，但我认为一些设计良好的接口可以为我们提供很多启示，例如在小精灵应用（widgets）系统中，有很多接口用于客户程序和 CAR 对象的界面交互，而另一些接口则用于数据交互，我们既可以直接使用这些标准接口，也可以从中学习接口的设计方法。

3．继承性(扩展性)

CAR 接口具有不变性，但不变性并不意味着接口不再发展，随着应用系统和构件程序的发展，接口也需要发展。类似于 C++中类的继承性，接口也可以继承发展，但接口继承与类继承不同。首先，类继承不仅是说明继承，也是实现继承，即派生类可以继承基类的函数实现，而接口继承只是说明继承，即派生的接口只继承了基接口的成员函数说明，并没有继承基接口的实现，因为接口定义不包括函数实现部分。其次，类继承允许多重继承，一个派生类可以有多个基类，但接口继承只允许单继承，不允许多重继承。

根据 CAR 规范，所有的接口都必须从 IObject(将在下节讲述)派生，可以是直接派生，也可以是间接派生，但事实上，大多数接口直接派生于 IObject 接口，而并没有使用自定义接口之间的继承特性，CAR 并不提倡接口继承，除非在发展接口时，无论从功能上还是从语义上都应该采用接口继承，才会使用接口的继承特性。

4．多态性——运行过程中的多态性

多态性是面向对象系统的重要特性，CAR 对象也具有多态性，其多态性通过 CAR 接口体现。多态性使得客户程序可以用统一的方法处理不同的对象，甚至是不同类型的对象，只要它们实现了同样的接口。如果几个不同的 CAR 对象实现了同一个接口，则客户程序可以用同样的代码调用这些 CAR 对象。

因为 CAR 规范允许一个对象实现多个接口，因此，CAR 对象的多态性可以在每个接口上得到体现。

本节我们讨论了 CAR 规范中接口的一些细节，虽然接口把构件对象的功能暴露给客户程序了，但客户程序如何通过接口控制对象的生存期，又如何从对象的一个接口跳转到另一个接口呢？这就是 IObject 接口所提供的功能，在下节中我们详细讨论这些内容。

9.4 基接口 IInterface

9.4.1 IInterface 接口方法

所有的 CAR 接口都直接或间接地继承于 IInterface。RIID 为接口唯一标识，它存在于每个接口类的内存区的最前面，所以指向接口类的指针，可以强制性地被认为是一个指向 RIID 的指针。

其方法成员有：

```
IInterface
{
    virtual CARAPI_(PInterface) Probe(
            /* [in] */ RIID riid) = 0;

    virtual CARAPI_(_ELASTOS UInt32) AddRef() = 0;

    virtual CARAPI_(_ELASTOS UInt32) Release() = 0;

    virtual CARAPI Aggregate(
            /* [in] */ AggregateType type,
            /* [in] */ PInterface pObject) = 0;

    virtual CARAPI GetDomain(
            /* [out] */ PInterface *ppObject) = 0;

    static CARAPI_(PInterface) ProbeDefault(IInterface* pObj)
    {
        return pObj->Probe(EIID_IInterface);
    }
};
```

注：

斜体字的部分现在尚未实现，列在这里，只供参考。

有关 RIID 的定义：

```
typedef struct  _tagGUID
{
    UINT32 Data1;
    UINT16 Data2;
    UINT16 Data3;
    UINT8 Data4[ 8 ];
} GUID;
```

```
typedef GUID *PIID;
typedef GUID IID;
#define RIID   const IID &
```

接口中的前三个方法与 COM 的 IUnknown 接口的三个同名方法基本等价(除了 Probe 在参数处理方式上有些不同)，但多了个 Aggregate()方法，这就与 COM 不兼容了。

AggregateType 各成员简单描述如下：

成员	描述
AggrType_Virtual	用于二进制继承
AggrType_AspectAttach	用于动态聚合，将被聚合对象加入聚合链
AggrType_AspectDetach	用于动态聚合，从聚合链里去掉聚合对象
AggrType_Aggregate	用于动态聚合，聚合 aspect 对象
AggrType_Reaggregate	用于动态聚合
AggrType_Unaggregate	用于动态卸载聚合
AggrType_EnterContext	用于语境的进入
AggrType_LeaveContext	用于语境的离开
AggrType_Connect	用于接收器与可连接对象的连接
AggrType_Disconnect	用于接收器与可连接对象的断开
AggrType_AddConnection	用于回调接口方法的注册

CAR 构件技术许多重要的特征如二进制继承，回调机制，面向方面编程，Aggregate 方法扮演了非常重要的角色。

9.4.2 IInterface 方法透明实现

CAR 自动代码工具可以根据构件定义自动生成 IInterface 中四个方法的实现代码，普通的构件编写者不必关心这四个方法的实现。考虑 CHello 实现了两个接口 IHello 和 IWorld 的情况，CHello 构件类对象内存模型如图 9.10 所示。

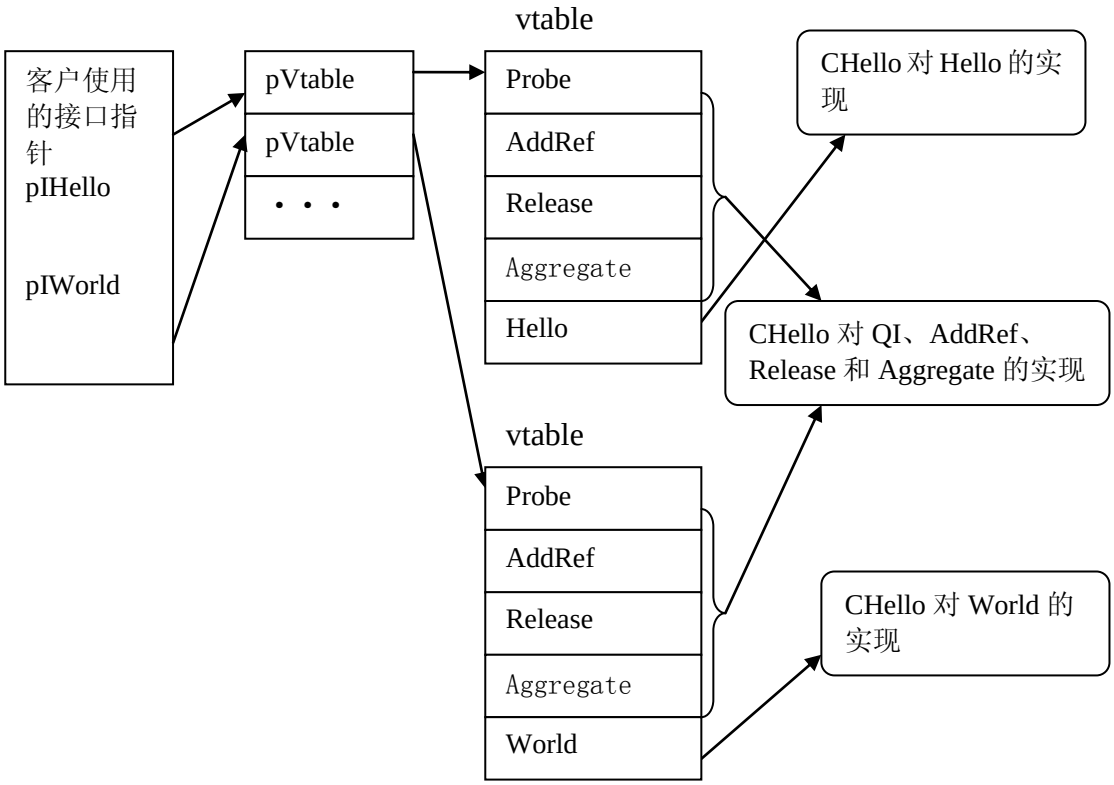


图 9.10 CHello 构件类对象内存模型

IHello 和 IWorld 两个接口都直接继承了 IInterface 的四个方法，CAR 自动代码工具自动生成了这四个方法的实现，构件编写者只需实现自己定义的接口方法即可，接口查询代码和构件的生存周期管理对构件编写者来说都是透明的。

9.4.3 接口查询规范

CAR 构件的接口查询规范完全遵守 COM 的接口查询规范，只不过二者的基接口不同而已。

对于同一个 CAR 构件对象的不同接口指针，查询得到的 IInterface 的接口必须完全相同。

- 接口对称性。对一个接口指针查询其自身应该成功。
- 接口自反性。若从一个接口指针查询另一个接口指针，那么从第二个接口指针查询第一个接口指针也应该是成功的。
- 接口传递性。若从接口指针 A 查询出接口指针 B，从 B 查询出接口指针 C，那么从 C 一定可以查询出 A。
- 查询时间无关性。若在某一时刻能够查询出接口指针 A，那么在以后的任何时刻再查询 A，也一定可以查询成功。

9.4.4 引用计数

CAR 构件的引用计数的实现对构件编写者来说是透明的，这里只说明一下 CAR 构件引用计数的实现模型。

根据 COM 规范，COM 的引用计数可以在构件、构件对象及对象接口这三个级别上实现的。(具体内容可参看 COM 的有关文档)。CAR 构件是在构件对象，构件模块这两个级别上实现引用计数的，其实现模型如图 9.11 所示。

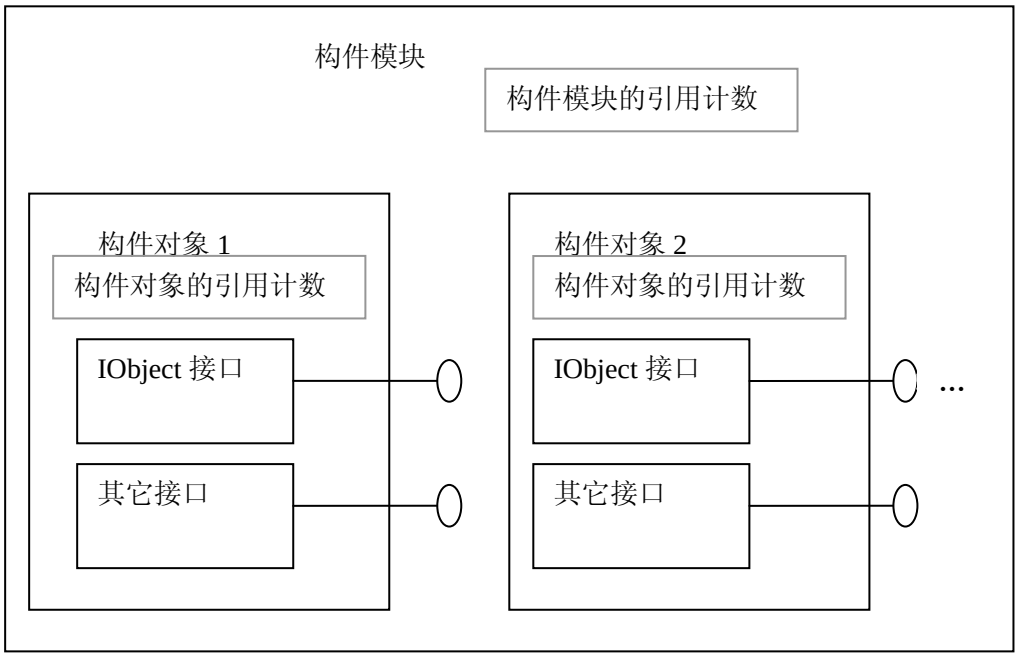


图 9.11 CAR 构件引用计数模型

CAR 构件接口在调用 AddRef，Release 的时候，分别是实现此接口的构件对象的引用计数加 1 和减 1，当客户程序创建一个构件对象的时候，封装此构件类的构件模块的引用计数加 1， 当一个构件对象的引用计数为 0 的时候，此构件对象被 delete，相应的构件模块的引用计数减 1，当构件模块的引用计数为 0 的时候，此构件模块被卸载。

CAR 的使用引用计数的规则完全遵循 COM 的规则。

9.5 对象工厂

9.5.1 IClassObject

对象工厂是创建构件类对象实例的一种特殊构件类，所有的 CAR 对象工厂都必须实现对象工厂接口 IClassObject：

```
IClassObject {
    virtual CARAPI QueryInterface(
            /* [in] */ RIID riid,
            /* [out] */ PObject *ppObj) = 0;

    virtual CARAPI_(UInt32) AddRef() = 0;

    virtual CARAPI_(UInt32) Release() = 0;
```

```
    virtual CARAPI CreateObject(
            /* [in] */ PIInterface pOuter,
            /* [in] */ RIID riid,
            /* [out] */ PObject *ppObject) = 0;

    virtual CARAPI StayResident(
            /* [in] */ BOOL bIsStayResident) = 0;
};
```

IClassObject 中有 QueryInterface、AddRef、Release 等 IObject 的所有接口方法，这使得对象工厂拥有普通 CAR 构件对象的所有特征。该接口与 COM 类厂接口 IClassFactory 大致等价，对象工厂的实现由构件库及自动代码工具封装，普通构件编写者同样不必关心它的实现。

CreateObject 方法中，pOuter 参数用来 CAR 构件对象被聚合的情况(这个参数用在对象创建时聚合里，目前我们没有用到)；riid 为对象创建完成后客户应该得到的初始接口 IID；ppObject 存放返回的接口指针。StayResident 和 COM 对象工厂的 LockServer 等价，是用来控制 CAR 构件的生存期的。

9.5.2 CAR 构件对象工厂的实现

每一个 CAR 构件类都有一个对象工厂实例(对象工厂对象)来负责构件类对象的创建，如果一个 CAR 构件模块里有多个构件类，那么就有多个对象工厂实例，这里要注意的是，负责某个构件类对象创建的对象工厂实例并非是这个构件对象的私有成员，而是它的一个全局对象，所有的对象工厂实例是由 CAR 构件模块来管理的，如图 9.12 示。

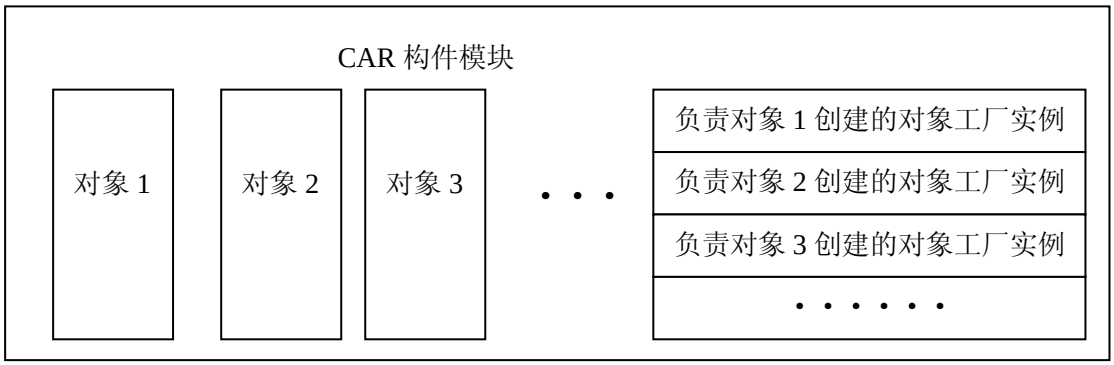


图 9.12 CAR 构件对象工厂

1.对象工厂类_CBaseClassObject

对象工厂类_CBaseClassObject 是一个比较通用的对象工厂代码，是由 CAR 的程序库实现的。其定义如下：

```
class _CBaseClassObject : public IClassObject
{
```

```
    public:
        CARAPI QueryInterface(
                    /* [in] */ RIID riid,
                    /* [out] */ PObject *ppObject);

        CARAPI_(UInt32) AddRef();

        CARAPI_(UInt32) Release();

        CARAPI CreateObject(
                    /* [in] */ PObject pOuter,
                    /* [in] */ RIID riid,
                    /* [out] */ PObject *ppObject);

        CARAPI StayResident(
                    /* [in] */ BOOL bIsStayResident);

        _CBaseClassObject(_CreateObjectFn fn) : m_fnCreateObject(fn) {};

    private:
        _CreateObjectFn m_fnCreateObject;
    };
```

对象工厂有一个私有成员 m_fnCreateObject，类型为_CreateObjectFn，这是一个函数指针，其定义为：

```
    typedef ECode (CARAPICALLTYPE *_CreateObjectFn)(
            PObject *ppObject);
```

它的参数对应于 CreateObject 的第三个参数。m_fnCreateObject 是用来创建构件类对象的函数代码，这个函数是由构件模块实现的，如果构件模块有两个构件类 CA、CB,那么构件模块就会实现函数_CACreateObject 和_CBCreateObject,分别用于实例化 CA 和 CB。

构件模块在为构件类 CA 定义类工厂对象的时候，将_CACreateObject 传给对象工厂的构造函数，如：

```
    _CBaseClassObject _g_CA_ClsObj(_CACreateObject);
```

这样，负责构件类 CA 的对象工厂对象_g_CA_ClsObj 就有了专门实例化构件类 CA 的特定代码。如图 9.13 所示。

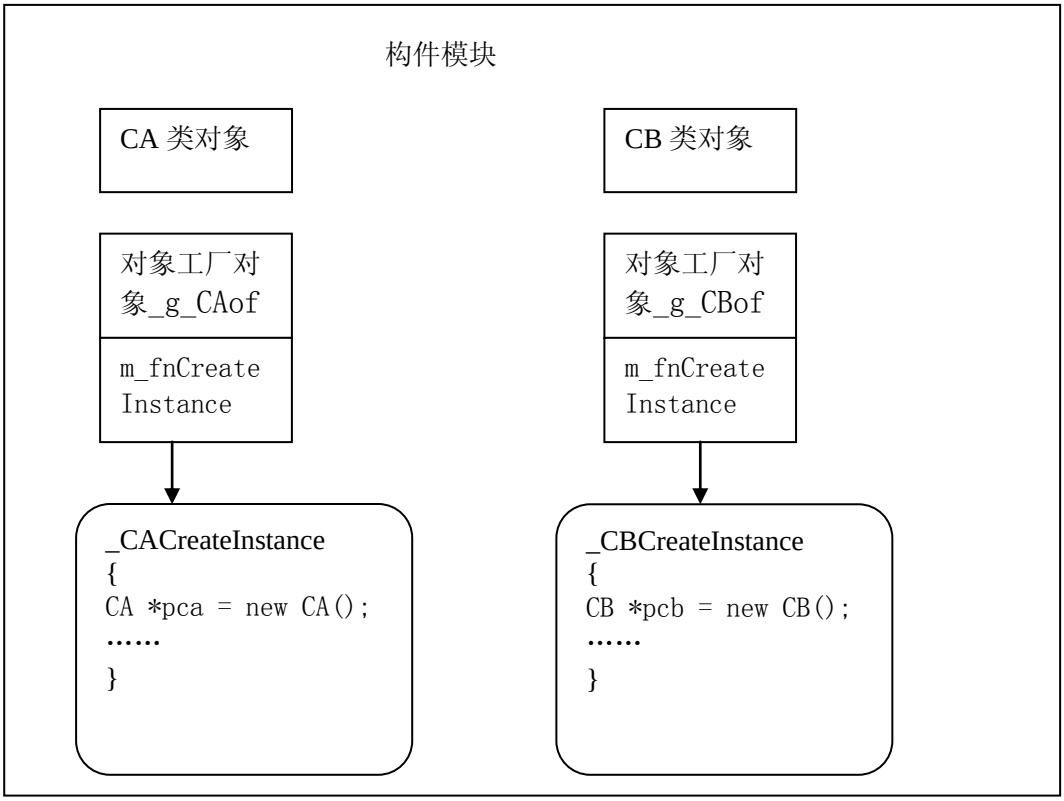


图 9.13 CAR 构件对象工厂对象实例化构件类

2.对象工厂的静态定义

不同于普通的构件对象，对象工厂不是在堆上创建的，而是在编译时静态创建的(即对象工厂是构件模块的一个全局变量，是数据段上的变量)，CAR 构件自动代码生成工具在每个构件类实现的_XXXX.cpp 文件里生成了相应的对象工厂定义。如 CClass 实现了接口 IHello，那么_CClass.cpp 有如下代码段：

```
CARAPI _CClassCreateObject(IObject **ppObj)
{
    ECode ec;

    void* pLocation = _CMemoryHeap_Alloc(sizeof(_CSandwichCClass), TRUE);
    if (!pLocation) return E_OUT_OF_MEMORY;
    CClass *pObj = (CClass *)new(pLocation) _CSandwichCClass;

    pObj->AddRef();
    ec = pObj->_Initialize_();
    if (FAILED(ec)) goto ErrorExit;
    *ppObj = (_IObject*)pObj;
    return NOERROR;
ErrorExit:
    ((_CSandwichCClass*)pObj)->~_CSandwichCClass();
```

```
        _CMemoryHeap_Free(pObj);
        return ec;
    }
    extern "C" _CBaseClassObject _g_CClass_ClsObj;
    _CBaseClassObject _g_CClass_ClsObj(_CClassCreateObject);
```

上述代码实现了以下功能：

(1)定义了_CClassCreateObject 函数；

(2)将_CClassCreateObject 函数地址传给对象工厂的构造函数，声明对象工厂变量_g_CClassCLsObj；

这样，构件模块就有了创建 CClass 对象的对象工厂了。

3.创建构件类对象

CreateObject 函数的实现就是调用 m_fnCreateObject 成员实现构件类对象创建的。

```
    ECode _CBaseClassObject::CreateObject(
        IObject *pOuter, RIID riid, IObject **ppObj)
    {
        IObject *pObj;
        if (NULL == m_fnCreateObject) return E_CLASS_NOT_AVAILABLE;
        ECode ec = (*m_fnCreateObject)(&pObj);
        if (FAILED(ec)) return ec;
        ec = pObj->QueryInterface(riid, ppObj);
        pObj->Release();
        return ec;
    }
```

4.构件模块的生命周期

(目前的实现还未完备)略

5.客户端怎样得到对象工厂

CAR 构件程序库提供了_EzAcquireClassObject 函数让客户得到相应构件类的对象工厂，这个函数的声明如下：

```
    EZAPI _EzAcquireClassObject(
        /* [in] */ REZCLSID rclsid,
        /* [in] */ PContext pContext,
        /* [out] */ PObject *ppObject);
```

rclsid 指定构件类的 ClassID, pContext 是有关对象的运行环境信息，在此先不介绍，读者可以先不管，但我们假定客户程序和构件模块是在同一进程的。ppObject 返回得到的对象工厂指针。

这个函数的实现可以简单地说明如图 9.14 所示。

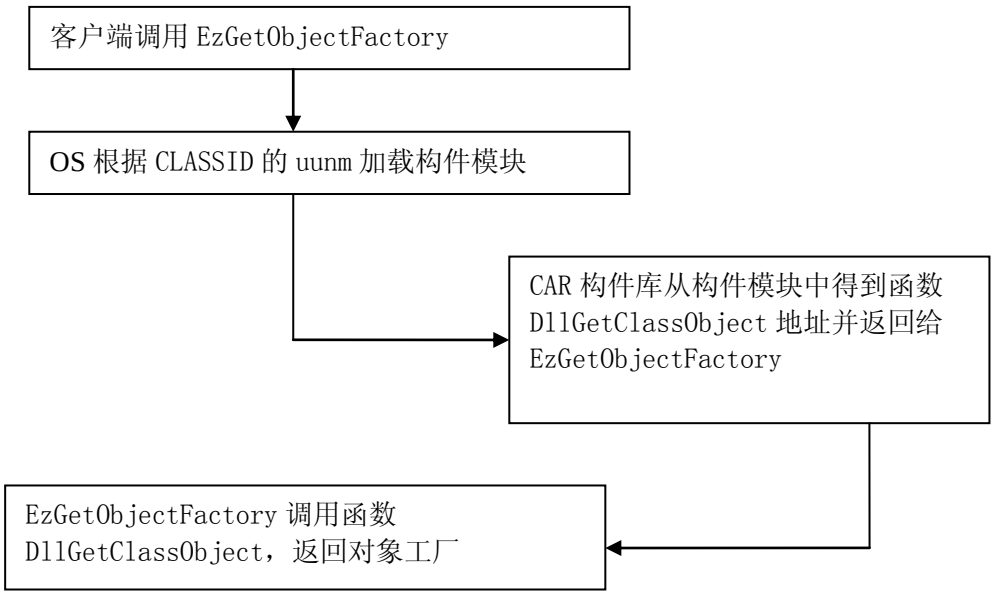


图 9.14 EzGetObjectFactory 函数实现流程图

对于 OS 加载构件模块过程读者不必关心，我们要了解的是 DllGetClassObject 函数。对于每个构件模块，都会实现这个函数并导出，它是用来得到构件模块定义的所有的对象工厂。

DllGetClassObject 的声明：

```
CARAPI _CarDllGetClassObject(
    REFCLSID clsid, RIID riid, IObject **ppObj)
```

构件模块实现的_CarDllGetClassObject，是以函数名 DllGetClassObject 导出的，clsid 为构件类的 ClassID, riid 为对象工厂接口的 IID, _CarDllGetClassObject 被调用时，riid 指定为 EIID_IClassObject，ppObj 返回对象工厂指针。

这个函数是 CAR 构件自动代码生成工具生成的，普通的构件编写者也不必自己实现这个函数。

考虑一构件模块封装了构件类 CA 和 CB，那么我们可以简单地实现 DllGetClassObject 函数如下：

```
//CA 和 CB 的对象工厂的声明
extern _CBaseObjectObject _g_CAOF;
extern _CBaseObjectFactory _g_CBOF;

EXTERN_C CARAPI _CarDllGetClassObject(
    REFCLSID clsid, RIID riid, IObject **ppObj)
{
    if (ECLSID_CA == clsid) {
        return _g_CA_ClsObj.QueryInterface(riid, ppObj);
    }
    if (ECLSID_CB == clsid) {
        return _g_CB_ClsObj.QueryInterface(riid, ppObj);
```

```
        }
        //其它实现
        ……
        return E_CLASS_NOT_AVAILABLE;
    }
```

每个对象工厂都会调用 QueryInterface 查询并返回各自的 IObjectFactory 接口指针。

第十章　CAR 构件自动生成代码框架

10.1 编写 car 文件

开发 CAR 构件的第一步是编写 car 文件。Car 文件用于定义构件中的类、接口、方法及其参数等信息。

下面请看本章范例 hello 构件的 CAR 文件（hello.car）内容：

```
module
{
    interface IHello
    {
        Hello();
    }
    class CHello
    {
        interface IHello;
    }
}
```

示例 hello.car 文件中，定义了构件 hello。该构件定义了接口 IHello 和类 CHello。IHello 接口提供 Hello 方法；类 CHello 实现接口 IHello。

10.2 生成源程序框架

在编写完 car 文件后，用户在 ElastosSDK 开发环境下，使用 emake 工具可以生成构件源程序框架。这将减少程序员的输入量，并且可以有效避免此环节的拼写错误。具体用法是执行下面的语句：

```
emake <carfile>
```

其中<carfile>为 car 文件路径名。执行该命令后，将在当前目录生成相应的头文件、cpp 文件和 sources 文件，其中头文件和 cpp 文件为程序框架文件，sources 文件用于指定如何编译源代码，生成什么类型的目标文件等信息。头文件和 cpp 文件的文件名由 CAR 文件中定义的类名指定。当 car 文件中定义了多个类时，将生成对应的多个头文件和.cpp 文件。

在本示例中，执行下面语句：

```
emake hello.car
```

将生成 CHello.h 文件、CHello.cpp 文件和 sources 文件。sources 文件生成后一般不需要修改，下面是生成的 sources 文件的内容：

```
TARGET_NAME= hello
TARGET_TYPE= dll
```

```
SOURCES= \
    hello.car \
    CHello.cpp \

ELASTOS_LIBS = \
    elastos.lib \
    elacrt.lib \
```

TARGET_NAME 指定生成构件的名字，此名字必须与 car 文件名相同，否则不能通过编译。生成构件的类型默认为 dll，由 TARGET_TYPE 指定。

10.3 填写实现代码

本示例中无须修改生成的头文件。首先来看一下生成的 cpp 源程序框架的内容：

```
#include "CHello.h"
#include "_CHello.cpp"

ECode CHello::Hello()
{
    // TODO: Add your code here
    return E_NOT_IMPLEMENTED;
}
```

在生成的文件框架中，方法的返回的 ECode 值为 E_NOT_IMPLEMENTED，表示此方法未实现。在填写完实现代码后，需要修改为适当的 ECode 值。

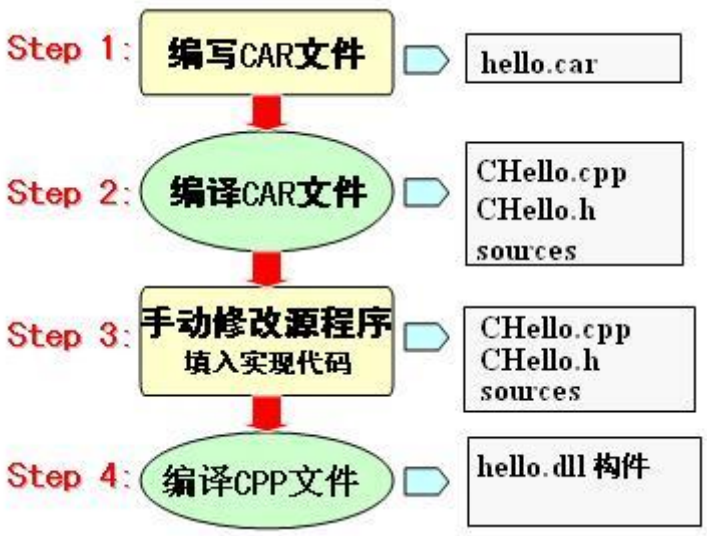
下面请看构件 hello 的实现，使用实现代码替换 TODO 注释（文中黑体部分为填充代码）：

CHello.cpp 文件：

```
#include "CHello.h"
#include "_CHello.cpp"
#include <stdio.h>

ECode CHello::Hello()
{
    puts("Hello, world! -----\n");
    return NOERROR;
}
```

执行完以上步骤，基本上完成了构件 hello 的编写。

10.4 CAR 文件的编译过程



10.4.1 emake.bat 对 car 文件的编译

系统中提供了 emake 工具对当前环境下的所有可编译的文件进行编译（命令行方式命令）：

```
emake hello.car
```

这里说明一下 emake 对 car 文件的编译。归根结底，emake 对 car 文件的编译分两个步骤：即 carc.exe 和 lube.exe 的编译过程。命令执行后系统会在当前目录下生成 cls 文件，然后通过对 cls 文件的编译生成对应的代码框架、相应构件的文档框架（xml 描述的该构件的框架结构）以及相应的 sources 文件。

10.4.2 carc.exe 对 car 文件的编译

carc.exe 对 car 文件的编译是在指定目录（默认是当前目录下）生成对应的 cls 文件：

```
carc.exe  -c hello.car
```

也即 class information，或者叫元数据信息（metadata，描述数据的数据，data about data），这是 CAR 的一个特性之一。元数据中包含的类对象的组织信息（如类的排列顺序、包含的接口）、各个接口的信息（如接口的种类、包含的接口方法）、各个方法的信息（如接口方法参数的种类、排列顺序等）以及 CLSID 等信息。这些信息是经过压缩保存起来的，编译的时候把元数据文件打包到 dll 中，就可以通过 DLL 的导出函数找到。

构件以接口方式向外提供服务，接口也需要元数据来描述才能让其他使用服务的用户使用。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。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不同构件之间的调用才成为可能，构件的远程化，进程间通讯，自动生成 Proxy 和 Stub 及自动 Marshalling、Unmarshalling 才能正确地进行。在微软由 idl（Interface Description Language）文件来描述接口，具体可查看微软 msdn 中有关 idl 的章节。.idl 信息在经 midl 编译后，会被编译为单独的 tlb

文件，打包在可执行文件.exe 或者.dll 中成为文件资源的一部分。微软提供了一组接口 ITypelib 对 tlb 信息进行访问，具体情况可参见微软 msdn 有关 ITypelib 的接口文档。

总之，carc.exe 的编译就是生成元数据(.cls)的过程，接下来 lube.exe 处理的正是由 carc 生成的元数据（.cls 文件）。

10.4.3 lube.exe 对文件的编译

根据 carc.exe 生成的元数据（即.cls 文件），lube 工具将生成构件类的代码框架，包括对应构件模块中所包括的构件类的所有.cpp 和.h 文件以及编译所需要的 sources 文件，另外还有两个 xml 文件。

Lube 工具包括 lubc.exe 和 lube.exe。

Lubc：对 templates 中所有的模板（.lub 文件）进行分析，将分析后的结果放入 LubeHeader 这个数据结构（lubedef.h 中定义），最后生成.lbo 文件，以资源文件的形式保存到 lube.exe 中。

Lube:分析 cls 文件，获得必要的元数据信息，根据命令行上指定的 templates 名称从 lbo 资源中获取相应的 template 来生成代码框架。

10.5 编写使用 CAR 构件的客户程序

一旦在操作系统上得到了完整的 CAR 构件，那么我们就可以在设计软件应用程序时使用它了。以上面的 hello 构件为例，我们来编写一个使用它的客户程序。我们既可以使用 C++，也可以使用 VB 来开发客户程序，将来还可以使用高级的脚本语言进行开发。

下面请看构件 hello 的 C++客户程序代码（client.cpp 文件）：

```
#include <stdio.h>
#define _SMARTCLASS
#include “hello.h”

int main()
{
    ECode ec;
    IHello *pHello;

    ec = CHello::New(&pHello);
    if (FAILED(ec)) goto ErrorExit;
    pHello-> Hello();

    return 0;

ErrorExit:
    printf("Error, ec = %x\n", ec);
    return 1;
```

```
}
```

编写 sources 文件如下：

```
TARGET_NAME= hello_c
TARGET_TYPE= exe

SOURCES= \
    client.cpp \

ELASTOS_LIBS= \
    elacrt.lib \
    elastos.lib \
```

编译后运行客户端程序结果为：Hello, world! -----

10.6 自动生成代码框架的优点

编译环境自动生成代码的主要作用是封装和抽象构件的编写和使用，让用户更方便的开发和使用 CAR 构件。

以下几个方面体现了 CAR 构件技术的这种易用性：

用 car 书写的 CAR 构件接口描述文件。Hello 构件虽然接口简单，但如果用户用 C++编写，仍然要考虑到许多技术细节。采用 car 语言编写 hello.car，不仅简单而且让用户更加习惯构件编程的思维方法，不纠缠于内部的技术细节。

自动生成元数据。CLS 文件是 CAR 文件压缩后的样式库，里面描述了整个构件的接口、类、方法、结构、枚举等的定义。有了 cls 文件，就可以得到该构件的接口，函数的调用方法，甚至生成源码框架。在链接生成 hello.dll 的时候，将 hello.cls 也放入 hello.dll，这样就等于将一本说明书和构件捆绑在一起。无论构件被复制移动到哪里，都可以知道该构件该如何使用。

许多构件的重要组成部分，象类厂，DLL 中的注册函数等等，这些都由编译环境自动生成，不必用户再手工实现。

当然对于大多数用户来说，完全可以不必了解这些编译过程中的细节。只需在 CAR 编译环境生成的源代码框架中填写自己的实现代码，就可以做到构件编程。但对一些喜欢研究构件技术的用户，这些知识可能会对编程有所帮助。

值得注意的是：中间件 marshalling 主要分为 build in 和动态代理两种方法，Elastos2.0 的 CAR 构件编译过程中，虽然由编译环境生成了一些代码，包括源码框架和构件接口的抽象，但这些代码都不用于 marshalling，不会生成 proxy 和 stub。

第二篇　CAR 的技术特性

第十一章　构件自描述

11.1 构件自描述概念

构件自描述是构件能够描述自己的数据信息。在 COM 技术中，强调构件的自描述，强调接口数据类型的自描述，以便于从二进制级上把接口与实现分离，并达到接口可以跨地址空间的目的。

构件及接口数据的自描述是 COM 的理论基础及立足点之一,但在一些广泛采用的 COM 的具体设计和实现上，并未完全贯彻这种思想。比如微软的 MS COM 就是一个例子，其不足之处主要体现在以下几点：

在 MS COM 中，构件的一些相关运行信息都存放在系统的全局数据库——注册表中，构件在能够正确运行之前，必须进行注册。而构件的相关运行信息本身就应该是构件自描述的内容之一。

MS COM 对构件导出接口的描述方法之一是使用类型库（Type Library）元数据（Meta Data，用于描述构件信息的数据），类型库本身是跟构件的 DLL（Dynamic Link Library）文件打包在一起的。但类型库信息却不是由构件自身来来解释，而是靠系统程序 OLE32.DLL 来提取和解释，这也不符合构件的自描述思想。

大多数情况下，一个构件会使用到另一些构件的某种功能，即构件之间存在相互的依存关系。MSCOM 中，构件只有关于自身接口（或者说功能）的自描述，而缺少对构件依赖关系的自描述。在网络计算时代的今天，正确的构件依赖关系是构件滚动运行、动态升级的基础。

正是意识到 MS COM 中存在的种种问题，CAR 在继承了 COM 自描述思想的同时，针对上述问题，对 COM 的具体设计和实现进行了如下扩展和改进：

CAR 把类信息（ClassInfo）作为描述构件的元数据，类信息所起的作用与 MSCOM 的类型库相似，类信息由 CAR 文件编译而来，是 CAR 文件的二进制表述。与 MS COM 不同的是，MS COM 使用系统程序 OLE32.DLL 来取出并解释类型库信息；在 CAR 中，可以使用一个特殊的 CLSID 从构件中取出元数据信息，构件元数据的解释不依赖于其它的 DLL 文件。

在 CAR 的构件封装中，除了构件本身的类信息封装在构件内外，还对构件的依赖关系进行了封装。即把一个构件对其它构件的依赖关系也作为构件的元数据封装在构件中，我们把这种元数据称为构件的导入信息（ImportInfo）。

CAR 构件通过对 ClassInfo 和 ImportInfo 的封装，可以实现构件的无注册运行。并可以支持构件的动态升级和自滚动运行。

11.2 CAR 语言与构件元数据（metadata）

11.2.1 CAR 语言

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

下面请看 hello 构件的 CAR 文件（hello.car）的内容：

程序清单 11.1　hello.car 文件

```
module
{
    interface IHello {
        Hello();
    }

    class CHello {
        interface IHello;
    }
}
```

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

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

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

11.2.2 CAR 构件元数据(metadata)

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

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

CAR 构件以接口方式向外提供服务，构件接口需要元数据来描述才能让其他使用构件服务的用户使用。构件为了让接口与实现无关，从而保持了接口的不变性，使得动态升级成为可能；并且使用 vptr 结构将接口的内部实现隐藏起来，由接口的元数据来描述接口的函数布局和函数参数属性。接口的元数据描述的就是服务和调用之间的关系。有了这种描述，不

同构件之间的调用才成为可能，构件的远程化，进程间通讯，自动生成 Proxy 和 Stub 及自动 Marshalling、Unmarshalling 才能正确地进行。

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

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

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

CAR 构件的自描述信息主要包含类信息(ClassInfo)和导入信息(ImportInfo)两种。ClassInfo 被作为构件程序的元数据信息，用于描述构件导出的接口及方法列表；与此相反，ImportInfo 则描述该构件运行时需要用到的别的服务性构件的信息。

11.2.3 ClassInfo 构成

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

我们可通过如下示意图说明构件模块信息的主要构成：

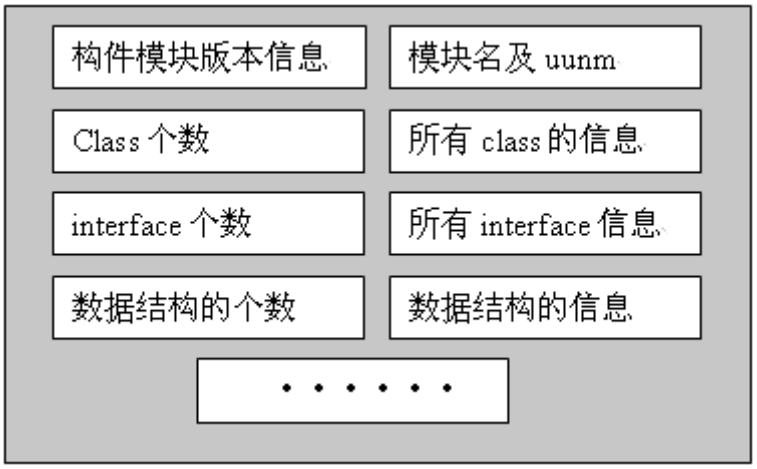


图 11.1　模块信息主要构成

对于每个构件类信息构成，可简单地如下图实例：

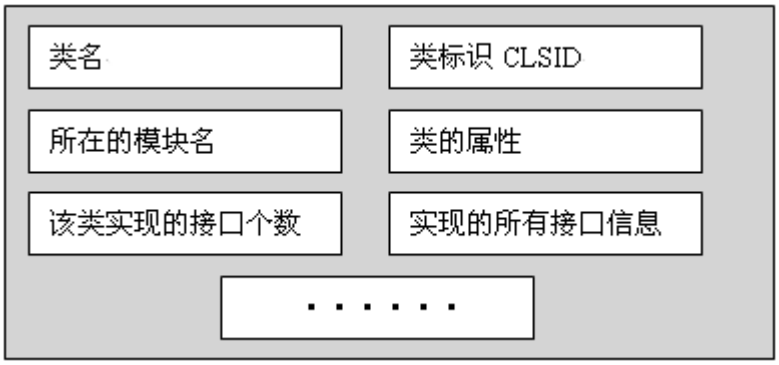


图 11.2　类信息主要构成

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

对于每个接口信息，其主要构成如下：

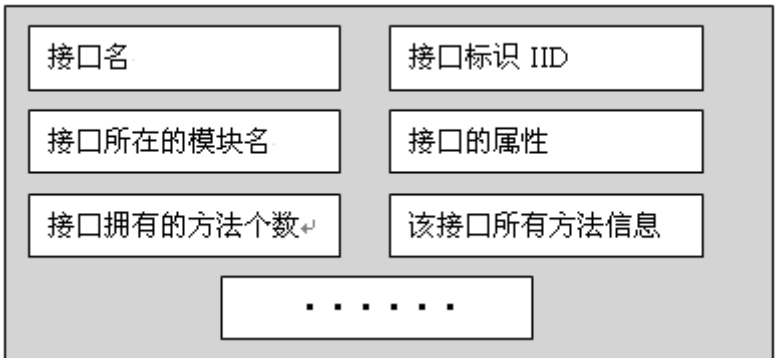


图 11.3　接口信息主要构成

接口也可以是另外模块定义，所以接口信息也记录了接口所在的模块名。

其中每个方法的信息结构如下：

图 11.4　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。

11.2.4 ImportInfo 构成

在 CAR 构件中，使用 ImportInfo 来描述构件运行所需要的其它构件的信息。ImportInfo 最重要的信息是构件模块的 uunm，uunm 是关于构件 DLL 文件的网络定位信息，用以唯一标识构件模块。

为了对构件文件进行快速定位，CAR 对 COM 标准的 CLSID 进行了扩展，引入了 ClassId，ClassId 除了包含构件类的 CLSID 外，还包括构件模块的 uunm。其 C/C++定义如下：

```
typedef struct  ClassId {
    CLSID clsid;
    WString urn;
} ClassId;
```

如图 11.5，除了 uunm 和 ClassId 外，构件导入信息还包括构件的版本号、最后修改日期、更新周期等，这些信息在构件升级及错误恢复时发挥重要作用。

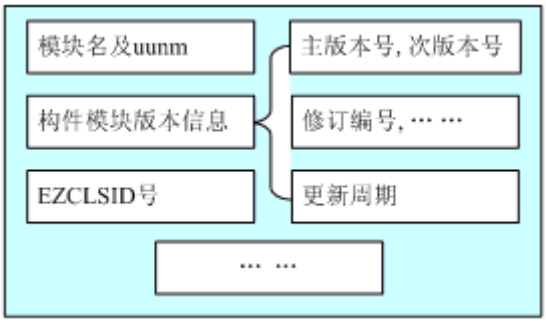


图 11.5 CAR 构件的 ImportInfo 元数据信息

利用 CAR 构件的导入信息，使得只要具备基本的构件运行环境，CAR 构件或构件客户程序就可以自滚动地运行。如：构件 A 依赖于构件 B，构件 B 依赖于 C。在某系统中最初只安装了构件 A，在构件 A 运行时，构件 A 在创建构件 B 的构件对象时，通过 ClassId 指定了构件 B 的 URN，系统就可以自动到网络上下载构件 B 的程序。同理，在没有事先安装构件 C 的情况下，构件 B 也能够得到正确运行。

这种自滚动运行机制给软件使用者带来了极大的方便，使用者根本不需要了解除了他直接使用的软件之外的其它任何信息。软件的开发者也不再需要费心尽力的去为一个庞大而关系复杂的软件制作安装程序。

11.2.5 CTL metadata

Elastos 里的接口元数据来源于 CAR（component assembly runtime）文件，该文件等同于微软的 idl 文件，经过 Car 编译器 carc.exe 产生元数据文件 cls 文件。编译的时候把元数据文件打包到 dll 中，就可以通过 DLL 的导出函数 DllGetClassObject()找到。

CAR 文件描述了一个构件里所包含的类对象的组织信息（如类的排列顺序、包含的接口）、各个接口的信息（如接口的种类、包含的接口方法）、各个方法的信息（如接口方法参数的种类、排列顺序等）以及 CLSID 等信息。

这里编译前面的 hello.car，编译器在目标文件夹生成.cls 文件，这是一个经过 zip 压缩过的文件。

然后根据生成的.cls 文件，在目标目录产生包含客户所需接口信息的头文件 hello.h 供客户端使用。还有一种自描述的方式，就是在客户文件中用#import <hello.dll>代替#include<hello.h>。这是因为系统工具会在编译前使用工具 mkimport 对原文件作预处理，遇到 import 语句时，会到该 dll 的资源段寻找元数据生成相应的信息，从而实现了 dll 自带元数据的自描述。

根据生成的.cls 文件，通过系统工具 lube 的 5 大模板：header, background, foreground, public, serverh 来生成代码框架。

(1) 用 lube -C aspect.cls -r header 命令用 header 模板生成_hello.h、hello.h、_hello_c.h 和_hello_ref.h 文件。

hello.h： 用于实现 New，NewInContext 创建构件对象的方法。

_hello.h： 所有接口都继承于 IObject

_hello_c.h： 把 car 构件的接口以 C 的方式定义。以便用 C 语言实现客户端程序使用 car 构件时调用。

_hello_ref.h：实现类智能指针

(2) 用 lube -C aspect.cls -r foreground 命令用 foreground 模板生成 CHello.h、CHello.cpp、hello_cn.xml、hello_en.xml、sources 文件。

CHello.h： 用于实现继承关系：类 CFoo 继承于_CFoo(基类)

CHello.cpp： 生成实现类方法的代码框架。

hello_cn.xml： 帮助文档框架。

hello_en.xml： 帮助文档框架。

sources： 编译配置文件。

(3) 用 lube -C aspect.cls -r background 命令用 background 模板生成_CHello.cpp、_CHello.h、_hellopub.cpp 文件。

_CHello.h： 基类的定义。

_CHello.cpp： 基类中方法的实现。

_hellopub.cpp：生成了_CarDllGetClassObject 函数的实现。

(4) 用 lube -C aspect.cls -r cls2abrg 命令用 public 模板子集生成_helloabrg.cpp 文件，它是一个精简的 cls，用于 Marshal。

这里的运行 lube 传入 public 参数，可以文件转换成 car,tlb 文件，并生成如下文件：_test3abrg.cpp 、 _test3uuid.cpp、_test3uuid.h。

(5) 用 lube -C aspect.cls -r serverh 命令用 serverh 模板生成 hello_server.h 文件。

Hello_server.h: 服务器端编译 car 文件的时候生成的 _CHello.h 需要 include 该模板生成的头文件。

这些自动生成的文件会和用户自己编写的源文件一起生成服务器构件。具体流程请参看下图。其中灰色部分为用户所需完成的原始文件。绿色部分文件为用户所需填写代码的构件类源文件。这里的 lube 命令实际为：

lube -Chello.cls -r header;serverh;cls2abrg;background;

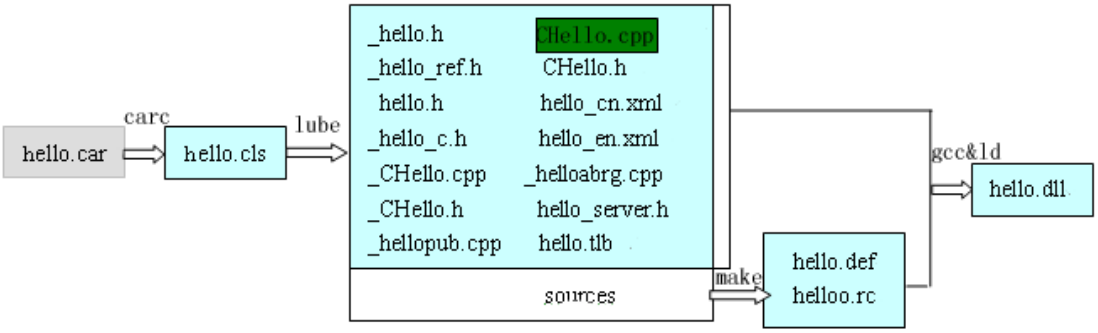


图 11.6　构件生成流程

11.2.6 Clsinfo metadata

同样以 hello 构件为例，hello.car 文件被 car 编译器编译后产生_helloabrg.cpp 如下：

程序清单 11.2　_helloabrg.cpp 文件

```
#if !defined(__CAR_HELLOABRG_CPP__)
#define __CAR_HELLOABRG_CPP__
#include <clsinfo.h>

static unsigned char s_hello_classInfo[96] = {
    0x60, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00,
    0x2c, 0x00, 0x00, 0x00, 0x1d, 0x0c, 0xe9, 0x97,
    0x4c, 0x84, 0xbe, 0xaf, 0xfc, 0xaf, 0xe8, 0xfb,
    0xca, 0x4c, 0x84, 0xbe, 0x01, 0x00, 0xe8, 0xfb,
    0x5c, 0x00, 0x00, 0x00, 0x02, 0x00, 0x22, 0x00,
    0x44, 0x00, 0x00, 0x00, 0x02, 0x0c, 0xe3, 0xb7,
    0x52, 0x84, 0xbe, 0xaf, 0xfc, 0x47, 0xe8, 0xfb,
    0xca, 0x52, 0x84, 0xbe, 0x02, 0x00, 0x82, 0x00,
    0x54, 0x00, 0x00, 0x00, 0x00, 0x14, 0x00, 0x78,
    0x5c, 0x00, 0x00, 0x00, 0x03, 0x04, 0x00, 0x00,
    0x0b, 0x04, 0x00, 0x00, 0x2c, 0x00, 0x00, 0x00,
};

CIClassInfo *g_hello_classInfo = (CIClassInfo *)s_hello_classInfo;

#endif // __CAR_HELLOABRG_CPP__
//
//    [ size(96), class(1), interface(1) ]
//    Text reverted from hello's Abridged-CLS
//
//
```

```
//    [ b7e30c02-8452-afbe-fc47-e8fbca5284be ]
//    interface IHello {
//        [0x82] Method_00(
//                    [in] uint32,
//                    [in] interface)
//        [0x00] Method_01()
//    }
//
//    [ 97e90c1d-844c-afbe-fcaf-e8fbca4c84be ]
//    class CHello {
//        interface IHello;
//    }
```

它的文件结构描述在 clsinfo.h 文件中（仅截取主要部分）：

程序清单 11.3　clsinfo.h 文件（仅截取主要部分）

```
#ifndef _CLSINFO_H_
#define _CLSINFO_H_
#include <elatypes.h>

EXTERN_C const CLSID CLSID_ClassInfo;
// { 0x4CDBF5FC,0xD030,0x4583,{ 0xAE,0xCD,0xA2,0x6E,0x95,0xB3,0x02,0x6F }}
typedef UINT32  CIBaseType;
//+---------+----------+------------+-------+--------------+
//| 31 ~ 16 |15|14|13|12|  11  |10| 9 |   8   |7|6|5|4|3|2|1|0|
//+---------+----------+------------+-------+--------------+
//|type size| reserved  | attributes  |pointer|     type      |
//+---------+----------+------------+-------+--------------+
//                     |retval|in|out|
//                     +------+--+---+
typedef struct _CIMethodEntry {
    UINT8          paramNum;
    UINT8          result;    // OBSOLETE. to be removed
    UINT8          reserved1; // the  highest digit for mark [in] interface
                              // the next for [out] interface
                              // the others is the stack length
    CIBaseType     *params;   // size_is(paramNum)
}   CIMethodEntry;

typedef struct _CIInterfaceEntry {
    UINT16         methodNumMinus3;// exclude IUnknown's 3 methods
    CIMethodEntry  *methods;       // size_is(methodNumMinus3)
```

```
    InterfaceId                     iid;
}   CIInterfaceEntry;

typedef struct _CIClassEntry {
    CLSID                   clsid;
    UINT16                  interfaceNum;
    CIInterfaceEntry        **interfaces;   // size_is(interfaceNum)
}   CIClassEntry;

typedef struct _CIClassInfo {
    int                     totalSize;
    int                     classNum;
    int                     interfaceNum;
    CIClassEntry            *classDir;      // size_is(classNum)
    CIInterfaceEntry        *interfaceDir;  // size_is(interfaceNum)
}   CIClassInfo;

typedef struct _CIClassInfoNode _CIClassInfoNode;
struct _CIClassInfoNode {
    CIClassInfo *m_pClsInfo;
    _CIClassInfoNode *m_pNext;
};

typedef struct _CIClassInfoNode CIClassInfoNode;

extern ECode InitServerClassInfo();

extern ECode RegisterServerClassInfo(
        /* [in] */ const CIClassInfo *pClassInfo);

extern ECode LookupServerClassEntry(
        /* [in] */ REFCLSID rclsid,
        /* [out] */ const CIClassEntry **pClassInfo);

extern ECode UnregisterServerClassInfos();
#endif // _CLSINFO_H_
```

clsinfo.h 中严格定义了上面数组的格式，有了这部分描述的信息上面的数组就是可读的了。

在生成的_hellopub.cpp 文件中可通过调用_CarDllGetClassObject 获得相关的接口信息。_CarDllGetClassObject 函数的定义如下：

程序清单 11.4 _CarDllGetClassObject 函数

```
EXTERN_C CARAPI _CarDllGetClassObject(
    REFCLSID clsid, RIID riid, IObject **ppObj)
{
    if (CLSID_CHello == clsid) {
        return _g_CHello_ClsObj.QueryInterface(riid, ppObj);
    }
    if (CLSID_ClassInfo == clsid) {
        *ppObj = (IObject *)g_hello_classInfo;
        return NOERROR;
    }
    return E_CLASS_NOT_AVAILABLE;
}
```

该函数增加了 CLSID_ClassInfo 选项，客户可通过调用_CarDllGetClassObject 取得该 dll 所包含的接口元数据。

11.3 元数据的使用

11.3.1 元数据在 elastos 中的使用

目前系统中存在两种接口元数据：一种是通过数组打包到 exe 或 dll 程序中；另一种是以单独的文件形式存在，存放在目标目录中。文件的后缀名为 cls，以 zip 压缩的方式存储。当生成一个 CAR 构件的时候，编译环境会将其放置到可执行文件的资源段中。

系统在加载 CAR 构件的过程中，会调用 SysRegisterClassInfo 将接口信息注册到内核。根据元数据在 Marshalling 和 unMarshalling 的过程中，可以将数据打包构建调用栈以及数据解包。

此外，在 java 和.net 中也存在元数据。Java 元数据的组织可参考 JMI(Java Metadata Interface)文档（可以从 sun 的官方网站 http://www.sun.com 下载）;.Net 的元数据组织可参考微软的相关网站或《Inside Microsoft .Net Il Assembler》一书，其中文名《Microsoft .Net Il 汇编语言程序设计》。

11.3.2 访问元数据接口

CAR 构件库定义了一套访问元数据的接口：

```
interface IModuleInfo;              //对构件模块信息的访问
interface IInterfaceInfo;            //对构件接口信息的访问
interface IClassInfo;               //对构件类信息的访问
interface IMethodInfo;              //对方法信息的访问
interface IParameterInfo;           //对参数信息的访问
interface IEnumerationInfo;         //对枚举器信息的访问
interface IEnumElementInfo;         //对枚举类类型元素信息的访问
```

```
    interface IDataTypeInfo;            //对数据类型信息的访问
    interface IStructInfo;               //对结构体类型信息的访问
    interface IAliasInfo;                //对别名信息的访问
    interface IFieldInfo;                //对字段信息的访问
    interface IMethodDispatcher;
```

上面的各个接口方法分别定义如下：

程序清单 11.5　接口 IModuleInfo 定义

```
    interface IModuleInfo
    {
            // Global operations
            GetName (  [out] AStringBuf Name);
            GetEZMODID([out] EZMODID *pezModid);
            //得到版本信息
            GetVersion([out] AStringBuf Ver);
            //得到属性
            GetAttributes( [out] CARAttrib *pAttibute );
            //得到构件模块里所定义的构件类个数
            // Class reflect information operations
            GetClassCount( [out] UINT *pCount );
           //得到构件类信息的枚举器，可用来访问所有的构件类信息
            GetClasses( [out] IObjectEnumerator **ppClassInfos);
            GetClassByName( [in]  AString Name, [out] IClassInfo **piClassInfo);
            GetClassByIndex( [in]  UINT uIndex, [out] IClassInfo **piClassInfo );
           //得到构件模块里所定义的接口总数
            // Interface reflect information operations
            GetInterfaceCount([out] UINT *pCount);
           //得到构件接口信息的枚举器，可用来访问所有的构件接口信息
            GetInterfaces( [out] IObjectEnumerator ** ppClassInfos);
           //通过接口名得到接口信息
            GetInterfaceByName([in]  AString Name, [out] IInterfaceInfo
**piInterfaceInfo);
           //通过接口 index 得到接口信息
            GetInterfaceByIndex( [in] UINT uIndex, [out] IInterfaceInfo
**piInterfaceInfo);
            // Struct reflect information operations
            GetStructCount( [out] UINT *pCount);
            GetStructs( [out] IObjectEnumerator **ppStructInfos );
            GetStructByName( [in]  AString Name, [out] IStructInfo
**piStructInfo);
            GetStructByIndex( [in] UINT uIndex, [out] IStructInfo **piStructInfo);
```

```
            // Enumeration reflect information operations
            GetEnumerations([out] IObjectEnumerator **ppEnumerationInfos );
            GetEnumerationCount([out] UINT *pCount);
            GetEnumerationByName( [in]  AString Name,
                [out] IEnumerationInfo **piEnumerationInfo
            );
            GetEnumerationByIndex([in] UINT uIndex,
                [out] IEnumerationInfo **piEnumerationInfo
            );
            GetEnumerationValueByName([in] AString Name, [out] INT *pValue );
            // Type Alias reflect information operations
            GetTypeAliasCount( [out] UINT *pCount );
            GetTypeAliases([out] IObjectEnumerator **ppAliasInfos);
            GetTypeAliasByName( [in]  AString Name, [out] IAliasInfo
**piAliasInfo
            ) ;
            GetTypeAliasByIndex( [in] UINT uIndex, [out] IAliasInfo
**piAliasInfo ) ;
    }
```

程序清单 11.6　接口 IClassInfo 定义

```
    interface IClassInfo
    {
            GetName ( [out] AStringBuf Name );
            GetCLASSID( [out] CLASSID *pClassId);
            GetModule( [out] IModuleInfo **piModuleInfo );
            GetAttributes( [out] ClassAttrib *pAttibute );
            GetConstructors( [out] IObjectEnumerator **ppMethodInfos);
            GetInterfaces( [out] IObjectEnumerator **ppInterfaceInfos);
            GetInterfaceCount([out] UINT *pCount);
            GetInterfaceByName( [in]  AString Name, [out] IInterfaceInfo
**piInterfaceInfo);
            GetInterfaceArttributeByName( [in]  AString Name,[out]
ClassInterfaceAttrib *pAttribute);
            GetCallBackInterfaces( [out] IObjectEnumerator **ppInterfaceInfos);
            GetCallBackInterfaceCount( [out] UINT *pCount );
            GetCallBackInterfaceByName([in]  AString Name,[out] IInterfaceInfo
**piInterfaceInfo);
            GetCallBackMethodByName(
                [in]  AString Name,
                [out] IMethodInfo **piMethodInfo,
                [out] int *pEnumValue
```

```
            );
            // Aggregate Class reflect information operations
            GetAggregateCount([out] UINT *pCount);
            GetAggregates( [out] IObjectEnumerator **ppClassInfos);
            GetParentClass( [out] IClassInfo **piClassInfo );
            //TODO: Add Aspects Info functions
            GetMethods([out] IObjectEnumerator **ppMethodInfos );
            GetMethodCount([out] UINT *pCount );
            GetMethodByName([in]  AString Name, [out] IMethodInfo
**piMethodInfo);
            GetMethodByIndex([in] UINT uIndex, [out] IMethodInfo **piMethodInfo );
            CreateObject([in] PDOMAININFO pDomainInfo, [out] PObject * ppObject );
    }
```

程序清单 11.7　接口 IInterfaceInfo 定义

```
    interface IInterfaceInfo
    {
            GetName ( [out] AStringBuf Name);
            GetIndex ( [out] UINT *pIndex);
            GetIID( [out] InterfaceId *piid);
            GetModule( [out] IModuleInfo **piModuleInfo);
            GetAttributes([out] InterfaceAttrib *pAttibute);
            GetMethods( [out] IObjectEnumerator **ppMethodInfos );
            GetMethodCount( [out] UINT *pCount );
            GetMethodByName( [in]  AString Name, [out] IMethodInfo
**piMethodInfo);
            GetMethodByIndex([in] UINT uIndex, [out] IMethodInfo **piMethodInfo);
            GetParentInterface( [out] IInterfaceInfo **piInterfaceInfo);
    }
```

程序清单 11.8　接口 IMethodInfo 定义

```
    interface IMethodInfo
    {
            GetName ([out] AStringBuf Name );
            GetIndex ([out] UINT *pIndex );
            GetParameterCount([out] UINT *pCount);
            GetInParameterCount([out] UINT *pCount);
            GetParameters([out] IObjectEnumerator **ppParamInfos);
            GetParameterByName(
                [in]  AString Name,
                [out] IParameterInfo **piParameterInfo
            );
```

```
            GetParameterByIndex([in] UINT uIndex, [out] IParameterInfo
**piParameterInfo);
            /*
            GetAttributes(
                [out] MethodAttribute *pAttibute
            );
            */
            CreateMethodDispatcher(
                [in]  PObject pObj,
                [out] IMethodDispatcher **piMethodDispatcher
            );
    }
```

程序清单 11.9　接口 IMethodInfo 定义

```
    interface IParameterInfo
    {
        GetName ( [out] AStringBuf Name );
        GetDataTypeInfo([out] IDataTypeInfo **piDataTypeInfo );
        GetAttributes([out] ParamAttrib *pAttibute);
    }
```

11.3.3 相关 API 函数

可以通过映射函数(reflection)获取元数据：

```
    //从模块的文件名中获得 IModuleInfo
    EZAPI EzAcquireModuleInfo(WString wsName,
        IModuleInfo **piModuleInfo)
    //从构件对象指针中获得 IModuleInfo
    EZAPI EzReflectModuleInfo(PObject pObj,
        IModuleInfo **piModuleInfo)
    //从构件对象指针中获得 IClassInfo
    EZAPI EzReflectClassInfo(PObject pObj,
        IClassInfo **piClassInfo)
    //从构件对象指针中获得 IInterfaceInfo
    EZAPI EzReflectInterfaceInfo(PObject pObj,
        IInterfaceInfo **piInterfaceInfo)
```

第十二章　面向方面的 AOP 编程模式

12.1 基于 CAR 的 AOP 技术

12.1.1 什么是 AOP

面向方面的编程（AOP，Aspect Oriented Programming）是在面向对象编程的基础上引入方面(Aspect)的概念的一种编程方法。在目前的面向对象的编程模式中，仅仅用类的思想来分析和实现软件系统，不能有效地表示软件系统的关注点。关注点是指系统中的一些横向逻辑代码。如调试，权限管理，缓存，内容传递，错误处理，记录跟踪，同步，持久化，负载均衡等，这些代码一般不代表系统的主要逻辑，但会分散在许多模块当中与其他代码缠结在一起。在 AOP 中称这些横向代码为横切关注点，也就是方面（Aspect）。AOP 将“关注”封装在“方面”中，将这些操作与业务逻辑分离，使程序员在编写程序时可以专注于业务逻辑的处理，而利用 AOP 将贯穿于各个模块间的横切关注点自动耦合进来。因此 AOP 技术是实现关注点的分离、改善系统逻辑、减低软件开发难度、提高软件开发质量和软件重用性的良好方法。

12.1.2 CAR 构件技术与 AOP 的结合

Elastos 通过其提供了一种基于二进制的 AOP 的实现，其能够灵活的实现基于构件级别的代码的动态插入，拦截，从而能提供构件的动态组合，扩展，以及实现各种功能。CAR 的 AOP 机制使用户能够在完全不用修改源代码的情况下简单,方便的动态聚合两个 CAR 构件类，从而生成一个具有两个 CAR 构件类所有接口实现的新构件类;亦可支持用户动态的在一个构件接口的实现方法前面或者后面动态的插入另一个构件的接口方法,或者用一个构件的接口方法的实现取代另一个构件接口的某个指定方法的实现等等。关于 CAR 的 AOP 机制的各种运用方法将在第 12.4 节中介绍。

12.2 CAR 的 AOP 技术组成

CAR 的 AOP 技术是由 aspect，动态聚合，语境(context)组成，aspect 对象是实现动态聚合必要条件，动态聚合是语境实现的基础。下面分别介绍这三个 CAR 的 AOP 技术的组成要素。

12.2.1 aspect 对象

aspect 是一种特殊的构件类实现，aspect 对象的特征是可以被其它构件对象聚合，该构件类必须实现 IAspect 接口，aspect 对象就是实现了 IAspect 接口的构件对象。

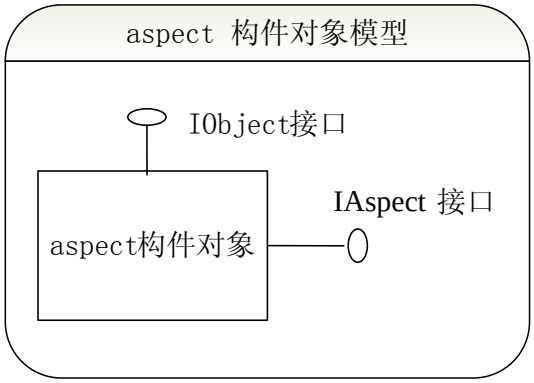


图 12.1 aspect 构件对象模型

在 CAR 里，只有 aspect 对象才可被聚合， 跟普通的 CAR 接口不一样，IAspect 接口不从 IObject 继承，但是它除了方法名与 IObject 不一样外，IAspect 的接口其它定义与 IObject 是完全相同的，下面是 IObject 接口的定义：

```
typedef struct IObjectVtbl {
    ECode (CARAPICALLTYPE *QueryInterface)(
            PObject pThis,
            /* [in] */ RIID riid,
            /* [out] */ PObject *ppObject);

    UInt32 (CARAPICALLTYPE *AddRef)(
            PObject pThis);                          //增加引用计数
    UInt32 (CARAPICALLTYPE *Release)(
            PObject pThis);                          //减少引用计数
    ECode (CARAPICALLTYPE *Aggregate)(
            PObject pThis,
            /* [in] */ AggregateType type,
        /* [in] */ PObject pObject);                 //动态聚合，一般用户不会直
                                                      接调用该方法
}   IObjectVtbl;

interface IObject {
    CONST_VTBL struct IObjectVtbl *v;
};
```

IAspect 另一个比较特别的地方是：没有 IID 与其对应。实际上，除了聚合的外部对象外，IAspect 接口对上层是不可见的，即 IAspect 接口不出现在 QI(QueryInterface)列表中。

注意：

1. CAR 构件技术里只有 aspect 构件对象可以被聚合。
2. 不允许一个 aspect 构件类包含回调接口。
3. aspect 对象不但可以被其他构件对象聚合，而且它也可以聚合其它 aspect 对象。

在 CAR 里， aspect 构件类在 CAR 文件里是用 aspect 关键字来标识的，用 CAR 语言定义一个 aspect 对象非常简单，可描述如下：

```
module
{
    interface Ihello {
        Hello();
    }
    aspect AHello {
        interface IHello;
    }
 }
```

AHello 构件类就是一个可被聚合的 aspect 构件类，被实例化后就是 aspect 对象。AHello 构件类不但会实现 IHello 接口，还会实现 IAspect 接口，CAR 自动代码工具会自动生成这部分代码。

aspect 对 IAspect 的实现是真正意义上的 IObject 实现，而对 IObject 接口的实现只是进行简单的转接。当 aspect 构件对象作为一个独立的构件对象存在时，对 IObject 的方法调用将会完全转接到 IAspect 接口的对应方法上。如果 aspect 对象被其它构件对象聚合，对 IObject 的方法调用则会被委托给外面的聚合对象上，外部对象保存 aspect 对象的 IAspect 的接口指针，用于 aspect 对象的真正 QI。

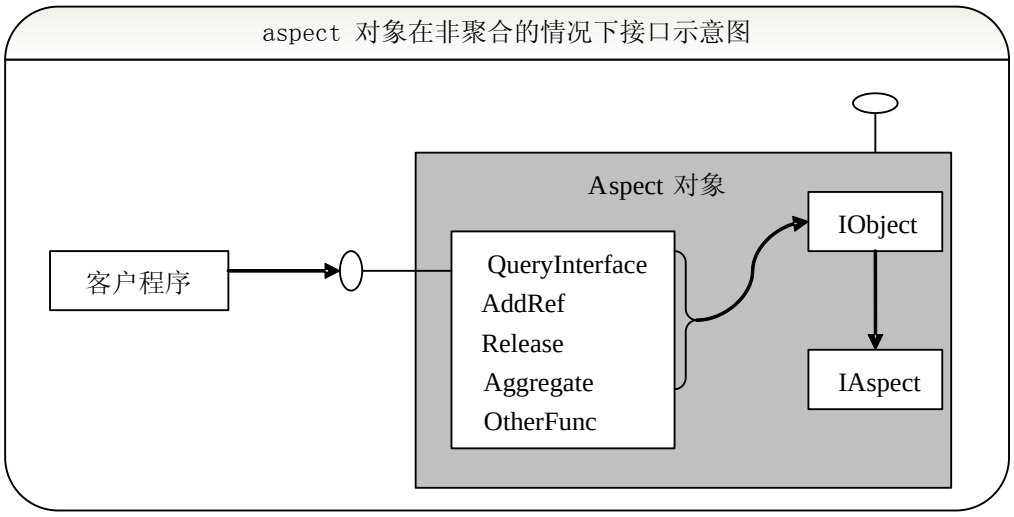


图 12.2 aspect 对象在非聚合情况下的接口示意图

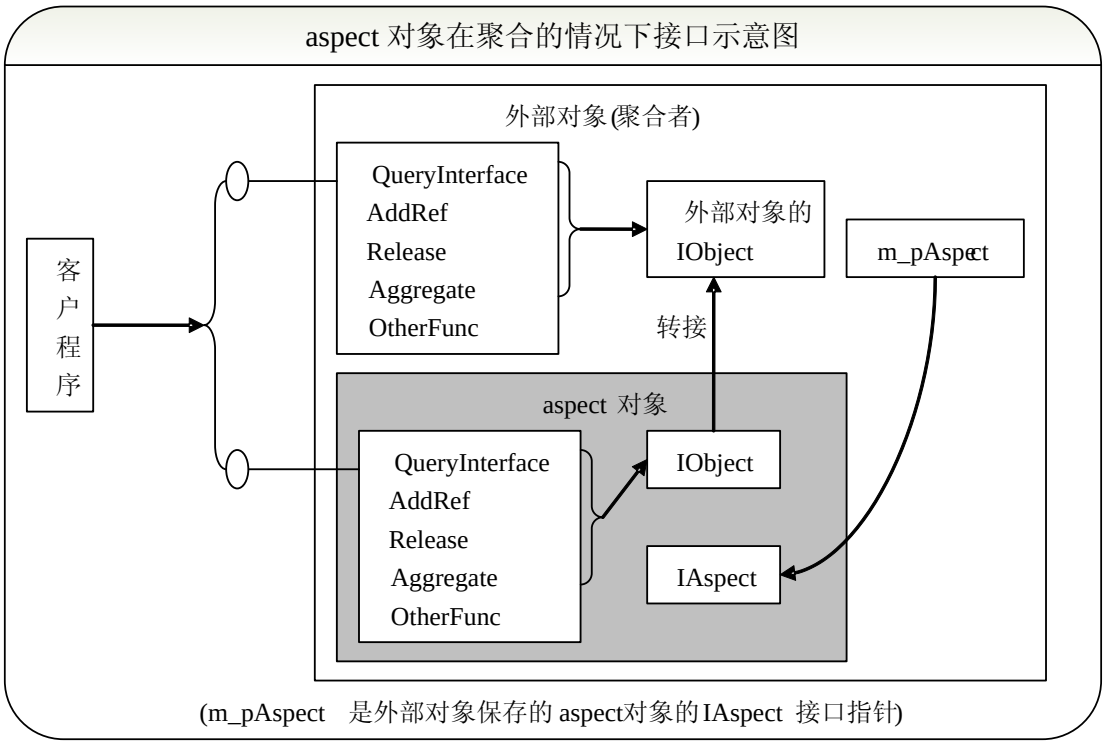


图 12.3 aspect 对象在聚合情况下的接口示意图

另外，CAR 构件技术里不允许一个 aspect 构件类包含回调接口。aspect 对象不但可以被其他构件对象聚合，而且它也可以聚合其它 aspect 对象。

12.2.2 动态聚合

一、动态聚合

动态聚合是构件对象在运行时随着运行环境的改变需要聚合其它的构件对象的聚合，虽然 COM 里的静态聚合能够满足一定的应用需求，但在现实模型中，更多的情况却是动态聚合。

比如宠物店的猫（构件对象），具有宠物和商品两方面的特征（Aspects）。但猫刚生下来时却未必是宠物，更不会是商品；而一旦被售出，就不再是商品，但宠物特征却保留了下来；护士在医院里是护士，回家后可能就是贤惠的妻子等等。

可以看出，随着构件对象的环境（context）的改变，构件对象所具有的特征（Aspects）也可能是不一样的。

因此，随时聚合、随时拆卸才是真正的面向方面的聚合模型。

下面介绍动态聚合的实现，语境将在 12.2.3 介绍。

动态聚合是通过 IObject 的 Aggregate 方法来完成的，因此构件编写者定义的每个构件对象都具有聚合其他 aspect 对象的能力。如有一个构件对象 A(构件类为 CA)和一个 aspect 对象 B(构件类为 AB)，对于构件 A 的编写者来说，在构件对象 A 里如果要聚合 aspect 对象 B，那么只要通过如下过程就可完成聚合任务：

1、创建 aspect 对象 B，如：

```
ec = AB::New(&pIB);
```

2、 调用 CAR 构件库提供的聚合函数 EzAggregate：

```
ec = EzAggregate((IObject *)this, pIB);
pIB->Release();
```

EzAggregate 函数调用成功后，那么对象 B 里保存了外部对象 A 的指针，对象 A 则保存了内部对象 B 的 IAspect 指针，同时内部对象 B 的引用计数也转嫁到了外部对象 A 上了。

EzAggregate 函数声明如下：

```
EZAPI EzAggregate(
                    /* [in] */ PObject pAggregator,
                    /* [in] */ PObject pAspect);
```

pAggregator 为外部对象指针，pAspect 为内部对象(aspect 对象指针),而实际上 EzAggregate 的实现就是调用 Aggregate 方法：

```
EZAPI EzAggregate(
                   /* [in] */ PObject pAggregator,
                   /* [in] */ PObject pAspect){
    return pAspect->Aggregate(AggrType_Aggregate, pAggregator);
}
```

聚合时我们也可直接象上面那样调用 Aggregate 方面来完成聚合过程，如：

```
ec = pObj->Aggregate(AggrType_Aggregate, (IObject *)this);
// AggrType_Aggregate 指定 Aggregate 方法是聚合行为。
```

Aggregate 方法在聚合时是引发了一系列的调用来完成聚合过程的，这个过程可以简单地如下表示(为了简便，我们将外部对象指针用 pOuter 表示，pAspect 表示 IAspect 指针)：

1）外部对象 A 调用：

```
pAspect->Aggregate(AggrType_Aggregate, pAggregator);
```

2）aspect 对象 B 及时反调外部对象的 Aggregate 方法：

```
pOuter->Aggregate(AggrType_AspectAttach, pAspect);
```

3）外部对象保存 aspect 对象指针

4）aspect 对象保存外部对象指针并将所有的引用计数全部转嫁到外部对象，聚合过程完成。

aggregate 过程是在 CA 里调用了 CB 的 Aggregate 方法，其 AggregateType 为 AggrType_Aggregate，然后在内部对象 CB 里又及时反调了外部对象的 Aggregate 方法，这时的 AggregateType 为 AggrType_AspectAttach，这个过程可称之为 attach 过程,attach 过程保存了 aspect 对象的 IAspect 接口指针，在多面聚合的情况下，attach 过程就是将一个 aspect 对象添加到聚合链里。

通过上面的聚合过程，外部对象就有了指向 aspect 对象的指针(m_pAspect)，被聚合的 aspect 对象就有了指向外部对象的指针(m_pOuter)，聚合后的两个对象的状态可简单地标识为：

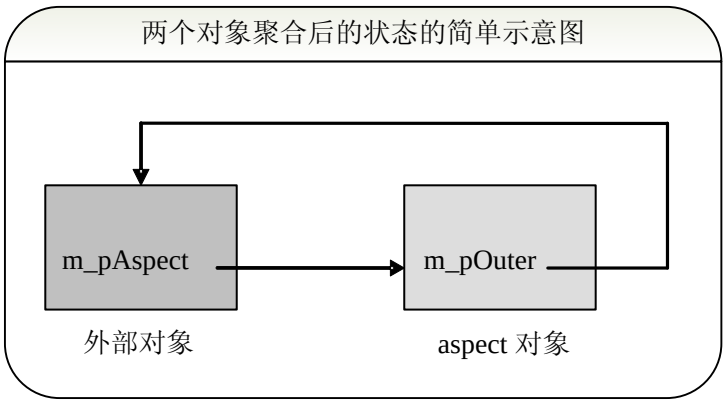


图 12.4 两个对象聚合后的状态的简单示意图

二、实例讲解

编写一个 aspect 客户端程序 client.cpp 如下：

```
#include <stdio.h>
#import <aspect.dll>

int main()
{
    ECode ec;
    IFoo* pIFoo;
    IBar* pIBar;
    //创建构件对象 pIFoo
    ec = CFoo::New(&pIFoo);
    if (FAILED(ec))
        return ec;
    //创建构件对象 pIBar
    ec = CBar::New(&pIBar);
    if (FAILED(ec)) {
        pIFoo->Release();
        return ec;
    }

    pIBar->BarHello();
    // pIBar 聚合构件对象 pIFoo
    ec = EzAggregate(pIBar, pIFoo);
    if (FAILED(ec)) {
        printf("Aggregate failed!\n");
        goto _exit1;
    }
    pIFoo->Release();

    //pIBar 查询出被聚合的构件对象
```

```
        ec = IFoo::Query(pIBar, &pIFoo);
        if (FAILED(ec)) {
            printf("Query failed!\n");
            goto _exit2;
        }

        pIFoo->FooHello();
        pIFoo->Release();
        //拆卸聚合
        ec = EzUnaggregate(pIBar, CLSID_CFoo);
        if (FAILED(ec)) {
            goto _exit2;
        }
        //拆卸聚合再次查询聚合的对象将失败
        ec = IFoo::Query(pIBar, &pIFoo);
        if (FAILED(ec)) {
            printf("Unaggregated!\n");
        }
        pIBar->Release();

        return 0;

    _exit1:
        pIFoo->Release();
    _exit2:
        pIBar->Release();
        return 1;
    }
```

编译运行程序之后的结果为：

```
Hello, I am from CBar!
Hello, I am from AFoo!
Unaggregated!
```

通过上例，可以看出，在完成聚合之后，可以调用 Query 方法来查询出所聚合的构件对象。Query 方法也是由自动代码生成框架生成，是相应构件接口的静态方法，上例中 IFoo 接口生成的 Query 方法的定义如下：

```
static CARAPI Query(IObject* pObj, IFoo** ppIFoo)
```

第一个参数 pObj 是构件对象，第二个参数是要 query 出的对象。实际上 Query 方法是对 QueryInterface 方法的一层包装，内部调用了 QueryInterface 方法。在编写代码时尽量使用 Query 方法，而不是 QueryInterface。

三、 多面聚合

上面我们介绍的只是两个对象的聚合，外部对象只聚合了一个 aspect 对象，但实际上，在面向方面编程时，往往需要一个对象聚合多个 aspect 对象，这就是多面聚合。我们在完成多面聚合时,实现上并没有多大的变化,就是创建多个 aspect 对象,多次调用 EzAggregate 方法使一个对象聚合多个 aspect 对象。

在上面一节中我们用了一个简单的示意图描述了两个对象聚合后的状态,类似地我们将多面聚合的结果示意如下：

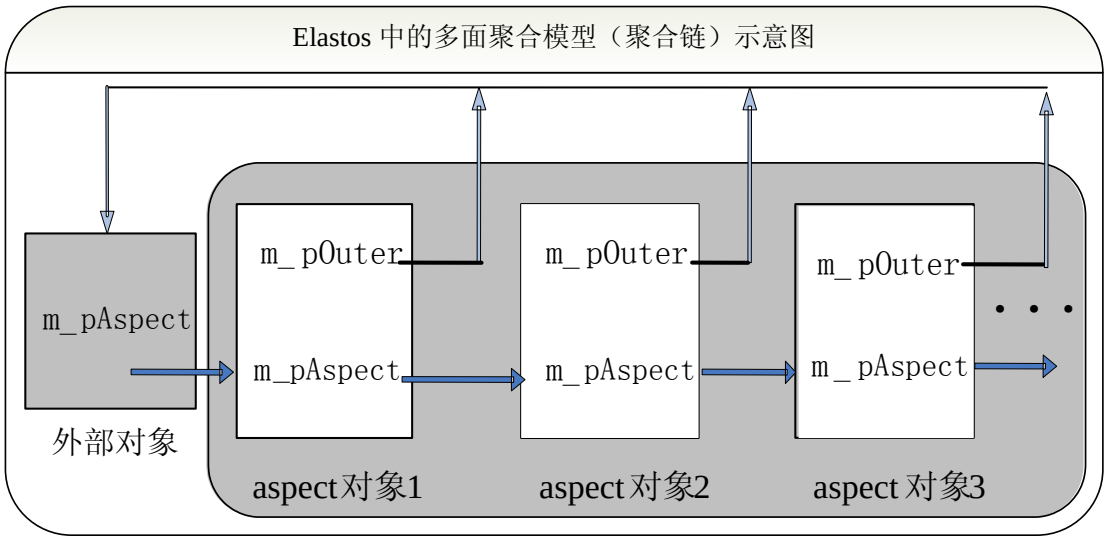


图 12.5 Elastos 中的多面聚合模型（聚合链）示意图

可以看出,构件对象指向下层被聚合的 aspect 对象的指针(m_pAspect)构成了一个单向链表，而链表中的每一个 aspect 对象的 m_pOuter 指针都指向最外层的外部对象(聚合者)。通过这种方式，QI 调用可以在链表中从头向尾传递，而所有的 aspect 对象的引用计数都委托给了最外层的外部对象。这与前面所介绍的两个对象的聚合情况并没有多大区别。

当然,上图是一种比较理想的聚合情况,在某些情况下,下层的 aspect 对象的 m_pOuter 指针可能会指向中间的 aspect 对象，最坏的情况是每一个 aspect 对象的 m_pOuter 指针都指向它相邻的上层对象。虽然这样也能正确运行，不影响执行的结果，但运行效率却十分低下。

多面聚合在实现上与两个对象聚合的情形最大区别就是 attach 过程的实现。在某时刻，外部对象聚合一个 aspect 对象时，在 attach 的时候，外部对象首先检查自己是否已经聚合其它 aspect 对象，如果有(m_pAspect 不为空)，就继续向 aspect 对象 1 attach，aspect 对象 1 又会检查自己的 m_pAspect，如果为空就保存了将要被聚合的 aspect 对象指针，否则继续向聚合链里的下一个 aspect 对象 attach，直到聚合链的最后一个 aspect 对象，聚合链里的最后一个 aspect 对象的 m_pAspect 指针一定是为空的。

四、动态拆卸聚合

CAR 构件库提供了 EzUnaggregate 函数来实现动态拆卸聚合：

```
EZAPI EzUnaggregate(
                        /* [in] */ PObject pAggregator,
                        /* [in] */ RCLASSID rAspectClsid)
```

pAggregator 为外部对象，rAspectClsid 为 aspect 对象构件类的 CLASSID 标识，这个函数可以使外部对象动态地拆卸某个已被聚合了的 aspect 对象，其大致的实现过程如下：

1）外部对象通过 aspect 对象标识 QI 出相应的 aspect 对象 IAspect 接口指针：

```
pAggregator->QueryInterface((REFIID)rAspectClsid, &pAspect);
```

由于 IAspect 接口没有 IID 与之对应，IAspect 接口对上层也是透明的，所以通过 aspect 对象的构件类标识来 QI。虽然 CLSID 只是用于标识构件类，一般情况下不能标识对象，但对被聚合的 aspect 对象却可以，因为聚合同一个 Aspect 构件类的多个对象实例是没有意义的。

如果当前对象找不到与之匹配的类标识，那么会传替给聚合链里的下一个对象，直到匹配为止，并返回对应的 IAspect 指针。

2） 与 rAspectClsid 匹配 aspect 对象断开自己并维持剩余的聚合链表

3） 还原引用计数。由于在聚合时 aspect 对象将引用计数完全转嫁到了外部对象，所以在拆卸的时候必须还原，还原的过程就是聚合时转嫁的相反过程。

注：目前动态拆卸聚合实现上还不完善，在拆卸时很容易破坏聚合链，关于动态拆卸的介绍以后再补充。

12.2.3 语境（context）

一、语境

语境是对象运行时的环境，一个对象如果进入了语境，那么该对象将具有此语境的特征，一旦对象离开了一个语境，则这个环境的特征就会失去，但该对象很有可能又进入了另外一个语境， 拥有新的环境特征。

在 CAR 里，语境也是一种构件对象，它具有普通构件对象的所有功能；aspect 对象作为语境的属性来表示语境的特征，在 CAR 文件里我们可以如下定义语境构件类：

```
module
{
    importlib("hello.dll");

    interface IObjectDump {
        DumpObjectInfo();
    }

    aspect AObjectDump {
        interface IObjectDump;
    }

    [aspect(AObjectEx, AObjectDump)]
    interface IHello{
        Hello();
    }

    context KContextTest {
        interface IHello;
```

```
        }
    }
```

CAR 语言提供 context 关键字来定义一个语境构件类，在语境构件类的属性里采用 aspect（aspect 对象 1， aspect 对象 2, …）来描述一个语境的特征。KContextText 语境拥有两个特征： AObjectEx 和 AObjectDump，这两个是 Elastos 系统定义的两个 aspect 对象，同时 KContextText 还实现了接口 IHello。

二、语境的进入和离开

如果一个构件对象进入了一语境，那么该对象会聚合语境的特征，也就是聚合了语境属性里的 aspect 对象，从而使该对象拥有了语境特征；如果该对象离开此语境，那么会拆卸聚合该语境属性里的 aspect 对象。

CAR 构件库提供如下两个函数分别完成语境的进入和离开：

```
EZAPI EzEnterContext(
        /* [in] */ PObject pContext,      //语境指针
        /* [in] */ PObject pObj)           //进入或离开语境的构件对象指针

EZAPI EzLeaveContext(
        /* [in] */ PObject pContext,
        /* [in] */ PObject pObj)
```

EzEnterContext 函数完成的功能是：由语境创建其特征实例(aspect 对象)，而语境进入者 pObj 动态聚合这些 aspect 对象。EzLeaveContext 实现的功能是对象进入者 pObj 拆卸聚合语境 pContext 的特征。

关于语境进入者、语境以及语境特征三者的关系要注意如下几点：

1）对象进入语境后，对象并不聚合语境本身所实现的接口功能，对象只是聚合语境的特征(aspect 对象)。

2）语境的特征只是语境的属性，在实现上是 aspect 对象，但语境本身并不具有这些 aspect 对象的功能。如宠物店语境并不具有宠物和商品对象实例的功能。

3）语境的特征只是语境的静态属性

4）一个对象可以进入多个语境，每个语境可被多个对象进入

5）在实际上不是所有的对象都可进入某个语境的，语境可对进入者设置条件，只有条件满足者才可进入。

6）也不是所有的对象都可随便离开某个语境，语境可对对象设置条件，只有条件满足者才可离开。

在一个语境构件类的定义之前可以指定其属性：aspect, aggregate,可以指定其中任意一个，也可以同时指定。格式如下：

aspect（aspect 对象 1， aspect 对象 2, …）来描述一个语境的特征。

aggregate（aspect 对象 1， aspect 对象 2, …）描述一个语境自身聚合的方面。

例如：

```
module
{
```

```
    interface IFoo {
        FooHello();
    }
    interface IBar{
        BarHello();
    }
    apect AFoo {
        interface IFoo;
    }
    [ aspect(AFoo) ]
    context KBar{
        interface IBar;
    }
}
```

在该示例中定义了一个方面构件类和一个语境构件类。CBar 语境拥有特征 CFoo.

如果一个构件对象进入了一语境，那么该对象会聚合语境的特征，也就是聚合了语境属性里的 aspect 对象，从而使该对象拥有了语境特征；如果该对象离开此语境，那么会拆卸聚合该语境属性里的 aspect 对象。

对于每个 context 构件类，可重载如下几个函数来满足它的对象进行控制:

```
virtual CARAPI OnObjectEntering(PObject pObj)
```

对象进入前调用，可以在这里检查对象是否满足要求。

```
virtual CARAPI OnObjectEntered(PObject pObj)
```

对象进入后（已经聚合上 context 定义的 aspects）调用。

```
virtual CARAPI OnObjectLeaving(PObject pObj)
```

对象离开前调用，context 可以在这里控制是否允许对象离开。

```
virtual CARAPI OnObjectLeft(PObject pObj)
```

对象离开后调用，在这里做善后工作。

下面我们假设有一个普通构件 hello.dll 希望能够进入到 KBar 这个语境。

（1） 服务端：

对 context.dll 这个构件对象，编写自动代码生成框架生成的 KBar.cpp 如下：

```
#include "CBar.h"
#include "_CBar.cpp"
#include <stdio.h>

ECode KBar::BarHello()
{
    printf("Hi, I am from KBar.\n");
    return NOERROR;
}
//对象进入后（已经聚合上 AFoo）调用
```

```
ECode KBar::OnObjectEntered(PObject pObj)
{
    printf("Object entered the context.\n");
    return NOERROR;
}
//对象离开语境后调用
ECode KBar::OnObjectLeft(PObject pObj)
{
    printf("Object left the context.\n");
    return NOERROR;
}
```

对于 aspect 方面对象 AFoo，我们希望在普通构件对象进入语境而自动聚合该方面时也能够做一些相应的工作，编写 AFoo.cpp 如下：

```
#include "AFoo.h"
#include "_AFoo.cpp"
#include <stdio.h>
#import <hello.dll>

ECode AFoo::FooHello()
{
    printf("Hi, I am form AFoo.\n");
    return NOERROR;
}
//被聚合成功后自动调用
ECode AFoo::OnAggregated(PObject pOuter)
{
    IHello* pIHello;
    ECode ec;
    //QI 出聚合者 pIHello
    ec = pOuter->QueryInterface(InterfaceId_IHello, (PObject *)&pIHello);
    if (FAILED(ec)) return ec;

    printf("Aggregated say hello: ");
    pIHello->Hello();
    pIHello->Release();
    return NOERROR;
}
//拆卸聚合成功后自动调用
ECode AFoo::OnUnaggregated(PObject pOuter)
{
    printf("Unaggregated ,bye-bye!\n");
```

```
        return NOERROR;
    }
```

现在我们定义一个普通的构件对象 hello.car，希望这个构件对象进入 KBar 这个 context 的时候能够自动的聚合 AFoo 这个方面对象，并在聚合了这个对象之后打印出一些信息：

```
    module
    {
        interface IHello {
            Hello();
        }
        class CHello {
            interface IHello;
        }
    }
```

编译之后编写自动代码框架所生成的 hello.cpp 文件：

```
    #include "CHello.h"
    #include "_CHello.cpp"
    #include <stdio.h>
    #import <context.dll>

    ECode CHello::Hello()
    {
        printf("Hi, I am from CHello.\n");
        return NOERROR;
    }
    //聚合成功后自动调用
    ECode CHello::OnAspectAttached(PObject pAspect)
    {
        ECode ec;
        IFoo  * pIFoo;
        //QI 出被聚合的对象 pIFoo
        ec = pAspect->QueryInterface(InterfaceId_IFoo, (PObject *)&pIFoo);
        if (FAILED(ec)) return ec;

        printf("Attached aspect and say hello: ");
        pIFoo->FooHello();
        pIFoo->Release();

        return NOERROR;
    }
```

```
//拆卸聚合成功后自动调用
ECode CHello::OnAspectDetached(PObject pAspect)
{
    printf("Dettached aspect, bye-bye!\n");
    return NOERROR;
}
```

（2） 客户端

CAR 构件库提供如下两个函数分别完成语境的进入和离开：

```
EZAPI EzEnterContext(
        /* [in] */ PObject pContext,
        /* [in] */ PObject pObj)
EZAPI EzLeaveContext(
        /* [in] */ PObject pContext,
        /* [in] */ PObject pObj)
```

其中 pContext 为语境指针，pObj 为进入或离开语境的构件对象指针。

EzEnterContext 函数完成的功能是：由语境创建其特征实例(aspect 对象)，而语境进入者 pObj 动态聚合这些 aspect 对象。

EzLeaveContext 实现的功能是对象进入者 pObj 拆卸聚合语境 pContext 的特征。

当客户端定义的构件对象调用这两个方法进入一个语境，就会自动聚合 server 端 context 构件类属性 aspect 中所指定要聚合的 IAspect 对象。离开这个语境时会自动拆卸聚合。

具体客户端的实现如下：

```
#include <stdio.h>
#import <hello.dll>
#import <context.dll>

int main()
{
    IHello* pIHello;
    IBar* pIKtx;
    ECode ec = NOERROR;
    //创建构件对象 pIHello
    ec = CHello::New(&pIHello);
    if (FAILED(ec))
        return ec;
     pIHello->Hello();

    //创建构件对象 pIKtx
    ec =  KBar::New(&pIKtx);
    if (FAILED(ec)){
```

```
            pIHello->Release();
            return ec;
        }
        //pIHello 构件对象进入语境 pIKtx
        ec = EzEnterContext(pIKtx, pIHello);
        if (FAILED(ec)) goto _error_exit;
        //pIHello 构件对象离开语境 pIKtx
        ec = EzLeaveContext(pIKtx, pIHello);
        if (FAILED(ec)) goto _error_exit;

    _error_exit:
        pIHello->Release();
        pIKtx->Release();

        return ec;
    }
```

编译后运行的结果为：

```
Hi, I am from CHello.
Attached aspect and say hello: Hi, I am form AFoo.
Aggregated say hello: Hi, I am from CHello.
Object entered the context.
Dettached aspect, bye-bye!
Unaggregated ,bye-bye!
Object left the context.
```

12.3 CAR 的 AOP 技术实现过程

12.3.1 AOP 的实现原理

AOP 用到的最主要的实现思想是通过建立一个对象代理，该对象代理提供对用户调用的接口，对象代理然后根据用户调用的接口方法以及用户指定的绑定形式，按照用户设定的次序调用绑定，替代方法或者真正的对象接口方法实现。实现逻辑如下图：

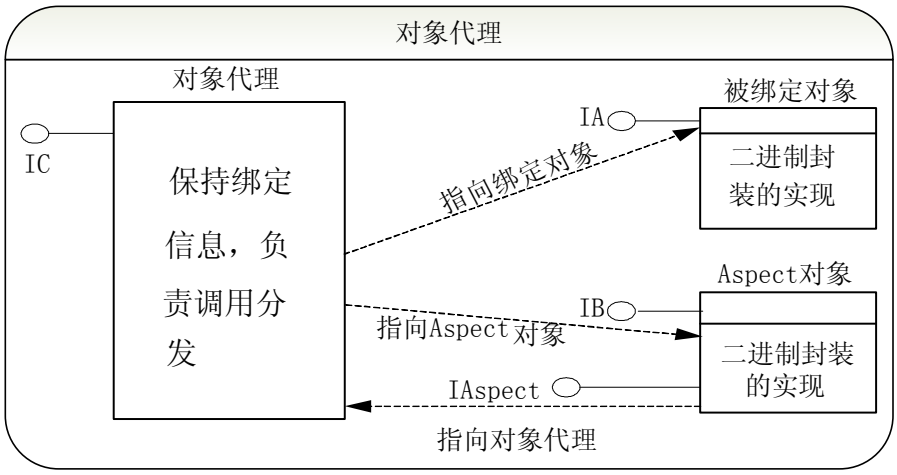


图 12.6 对象代理示意图

用户通过上图中对象代理暴露的 IC 接口（当然可以是多个接口）， 进行调用，对于实例中 A,C 两种情况， IC 接口的定义则完全同于被绑定对象暴露的接口 IA（当然可以是多个接口）， 对于实例中的 B，则 IC 接口聚合了 IA 接口以及 Aspect 对象所暴露的接口 IB（当然可以是多个接口）。对象代理将根据用户事先设定的绑定策略依次调用相关方法或者函数。

用户通过实现了 IAspect 的对象隐含的 IAspect 接口对对象代理进行操作，IAspect 接口的定义如下:

```
interface IAspect
{
   ECode SetObject(IObject  * pIObj,
                   Boolean aggregate,
                   void ** ppIObj);

   ECode SetPointCutByName(char * szObjectMethod,
                           char * szAspectMethond,
                           Advise uAdvise);

   ECode SetPointCutByAddr(char * szObjectMethod,
                           UINT uFuncAddr,
                           Advise uAdvise);
}
```

其中参数 Advise 类型的定义如下：

```
typedef enum Advise {
       Advise_Before  = 0x01;
       Advise_Around  = 0x02;
     Advise_After     = 0x03;
    } Advise;
```

下面就 IAspect 的方法做一些简单的介绍：

```
ECode SetObject(  /*in*/ IObject  * pIObj,
                  /*in*/ Boolean aggregate,
```

```
                    /*out */ void ** ppIObj)
```

SetObject 将接口 pIObj 绑定到某个 Aspect 上，如果参数 aggregate 为真，则表示该操作行为为动态聚合，即带有 Aspect 构件将与 pIObj 构件聚合在一块。返回的 ppIObj 指针即为聚合或者绑定后的指针。

```
ECode SetPointCutByName(
                    /* in */ char * szObjectMethod,
                    /* in */ Char * szAspectMethond,
                    /* in */ Advise uAdvise)
```

SetPointCutByName 方法将以方法名字字符串的形式选择将某个实现了 IAspect 接口的构件方法插入(或者替代)以字符串 szObjectMethod 为方法名字的相应位置， uAdvise 参数决定了插入位置， Advise_Before 表示插到方法前面，在调用构件方法之前调用实现了 IAspect 构件的指定方法；Advise_After 表示插到方法后面； Advise_Around 则表示取代该方法。

SzObjectMethod 为 0 的时候表示将实现了 IAspect 接口的构件的指定方法插入(或者替代)到另一构件的所有方法的指定位置。

```
ECode  SetPointCutByAddr(
                    /*in*/char * szObjectMethod,
                    /*in*/ UINT uFuncAddr,
                    /*in*/ Advise uAdvise)
```

SetPointCutByAddr 的作用同于 SetPointCutByName，不过它传入的第二个参数事一个普通函数地址而已。

IAspect 接口方法 SetObject(IObject * pIObj， Boolean aggregate, void ** ppIObj)将创建一个对象代理， 并根据传入的被绑定对象的接口指针 pIObj 以及参数 aggregate 创建对象代理的内部数据项接口代理等， 最后的返回指针 ppIObj 实际上是接口代理的指针。当用户调用 SetPointCutByName 或者 SetPointCutByAddr 方法的时候,则根据用户传递的绑定策略设置对象代理的相关数据信息。用户通过 IC 接口调用的时候将根据这些绑定信息将调用分发到相关函数或者方法。

下面再根据创建并初始化对象代理以及用户通过 IC 接口来调用的过程两个方面来进一步说明 AOP 的实现机理。

12.3.2 创建并初始化对象代理

第 1 步：获得 IAspect 接口

通过一个 QI(QuerryInterface)一个实现了 IAspect 接口的对象指针,或 者通过 EzGetNULLAspect api 函数获得一个空的 IAspect 指针(空意味着除了 IAspect 接口的实现外,没有其它接口实现)，在 CAR 文件中如果将一个实现类声明为 Ascpect 属性， 该类将自动实现 IAspect 指针(由 CAR 编译器生成相关自动代码)

第 2 步：调用 IAspect 的方法 SetObject 绑定要绑定的对象

SetObject(IObject * pIObj， Boolean aggregate, void ** ppIObj)首先将创建一个对象代理。对象代理的数据结构以及它与被绑定构件的关系(不考虑聚合)大致如下：

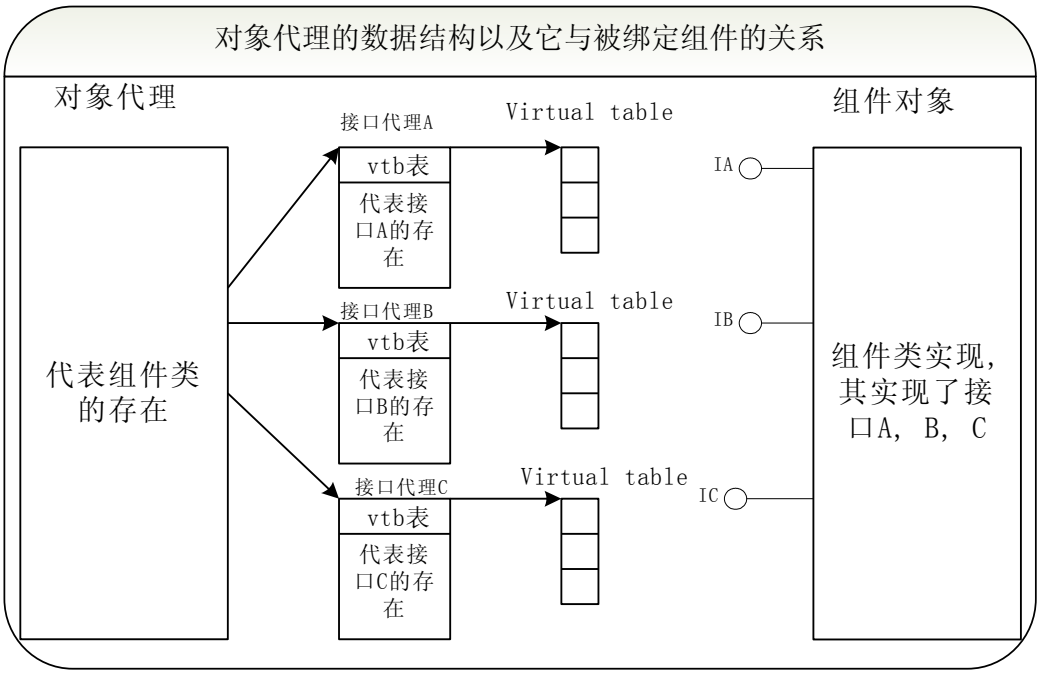


图 12.7 对象代理的数据结构以及它与被绑定构件的关系

如果 aggregate 为 TRUE,则表示为动态聚合,则表示代理要向用户暴露绑定构件以及被绑定构件的所有接口(不包括 IAspect 接口),如果是通过 EzGetNULLAspect 获得的接口,则 aggregate 参数只能为 False,即不允许动态聚合。 通过 IAspect 指针以及传入的 pIObj 能够获得所有接口的信息.(在 Elastos 的 CAR 机制中,接口是一个自描叙性数据,通过接口的 QuerryInterface 方法获得 ClSID 以及接口 IID, 通过 ClSID 可以在系统中查询构件,类,接口的元数据, 构件的元数据是构件所在的实现模块被加载到系统中的时候自动注册的),根据接口信息,将建立相应数目的接口代理(如果是动态聚合,则建立两个构件所实现的接口总和数减去 1 个接口代理,否则则建立被绑定构件所实现的接口数目个接口代理),建立接口代理后,将初始化接口代理的一些数据。

接口代理初始化数据包含如下过程：

首先，先将跳转表函数入口地址设置为接口代理的某个静态方法。其次，分配相应接口方法个数的 vtbl 表，将 vtbl 的各项如上图所示那么填写， 跳转表是系统内每个进程空间一个固定地址开始的表项，这段表的每项都具有相同的汇编代码,它的每一表项的代码执行最后都会 call 接口代理的第二项，跳转表函数入口， 基于接口的 Before，After，Around 各项都被赋空。然后，分配一个绑定函数数组，其包含相应接口方法个数的表项，每个表项包含三个指针，即一个前插函数，取代函数，后插函数指针，并赋 0， 接口代理有一个成员变量指向该数组。ppIObj 返回的实际上是相应的接口代理指针。

第三步：调用 SetPointCutByName 或者 SetPointCutByAddr 设置绑定方式

实际上它将设置接口代理的三个基于接口的 Before/After/Around 参数或者改变绑定函数数组的相应项。

12.3.3 用户通过返回的接口代理指针进行调用的过程

一、用户接口方法调用的实际过程

上面分析已经指出,用户获得的接口指针实际上是一个接口代理指针， 通过对比 C++对象的内存布局以及接口代理结构的数据项，我们就可以发现,用户用接口的形式调用其接口方法的时候，实际上就是调用接口代理的 vbtl 表的相应项,如果是 QI(QuerryInterface) ,AddRef 或 Release 函数,则实际将调用转到对象代理的 QI、AddRef、Release 上， 这种实现是基于接口的生命周期由对象的生命具体体现， 而对象代理,接口代理分别与对象， 对象的接口一一对应的。而对于其它方法的调用则实际上是跳到 vtbl 表的其它相应表项，即跳转表的相应表项执行，跳转表是由 Elastos 的每个用户进程运行用户程序前由系统在进程空间内分配并初始化的， 它被分配在每个进程空间相同的地址上，其每一个表项内容都是一段完全相同的汇编代码， 这段汇编代码将根据传入的接口 this 指针(其实是接口代理指针)，找到接口代理结构的第二项(即跳转表函数入口)，最后会 call 这个函数地址。

二、跳转表函数入口完成的功能

在初始化对象代理的说明中我们已经指出,跳转表函数入口是每个跳转表项最后所调用的函数的地址，它是在初始化对象代理过程中被赋值为接口代理的某个函数,所以这个时候调用就会跑到这个函数中执行，在这个函数中：

它首先将查询接口代理的基于接口的Around项,如果它被赋值,则将执行Around项的函数,并把方法调用传入的参数传给它,调用完直接返回， 如果：

1、Around 项为空,往下执行 2

2、它将根据跳转表表项中的 Call 指令的返回地址以及跳转表的基址计算出当前调用是接口的第几个方法.

3、检查绑定函数数组中的相应项的 Around,如果不为空,则执行该函数,调用完直接返回,为空则往下执行 4

4、检查基于接口的 Before 项是否赋值,如果被赋值,则先执行该项的函数,继续往下执行 5

5、检查绑定函数数组中的相应项的 Before,如果不为空,则执行该函数， 往下执行 6

6、获取对象代理保持的被绑定构件指针， 执行被绑定构件的相应方法,往下执行 7

7、检查绑定函数数组中的相应项的 After,如果不为空,则执行该函数， 往下执行 8

8、检查基于接口的 After 项是否赋值,如果被赋值,则执行该项的函数， 执行完整个调用过程就完成了。

第十三章　远程过程调用

13.1 CAR 远程构件调用的基本原理

13.1.1 CAR 远程接口自动列集\散集技术简介

当客户端和服务器端所在地址空间不同时，客户端进程对服务器端构件服务的调用，属于远程构件调用。由于两个不同空间之间不允许彼此直接访问或者具有不同的访问权限，所以需要某种通讯机制来实现不同地址空间之间的数据交互。

CAR 构件技术支持远程接口调用，通过数据的自动列集\散集技术进行不同地址空间的数据交互。构件服务和构件服务调用者可以处于操作系统的不同空间，而调用者可以如同在同一地址空间里面使用构件一样透明的进行远程接口调用，也就是说完全向用户屏蔽了底层使用的标准的列集\散集过程。

13.1.2 CAR 远程接口自动列集\散集技术的基本对象及其关系

CAR 的自动列集\散集主要用于 Elastos，它在构件的调用过程中地位类似于 COM 的自动列集\散集。用户如果采用默认的列集\散集过程，则使用一个远程接口如同使用一个本地接口一样，完全屏蔽了数据的交换，传递过程。

Elastos2.0 以存根\代理机制来实现远程接口自动列集\散集，主要涉及到三个对象，处于客户端的代理(Proxy)对象，处于服务端的存根(stub)对象，以及处于内核的(Object)对象。

一个客户端进程不一定只调用一个远程构件的服务，为了更方便有效的和各个远程构件交互数据，Elastos 在客户端为每一个对应的远程服务建立一个代理对象，记录一些客户进程的信息、远程服务的构件对象的信息以及一些调用的状态等，负责为客户进程与对应的远程服务联系。

Elastos 会为每个提供远程服务的构件对象建立一个存根对象，客户端代理不是直接与远程提供服务的构件对象联系的，而是与存根对象进行联系，通过存根对象来调用构件对象。

内核 Object 对象是联系客户端和服务器端的枢纽，保持了相关服务的信息以及创建对象代理所需要的一些信息，它的建立标志着用户可以通过某种方式远程获得相关服务(服务的发布)。

CAR 的自动列集\散集是通过在程序运行过程中，动态生成存根\代理来实现的。一个用户的远程构件调用首先通过 proxy 对象转发到内核相应的 Object 对象，Object 对象则将找到相应的服务进程以及 stub 对象，启动某一个服务线程并将调用转发给 stub 对象，然后再由 stub 对象去完成调用构件方法的过程. 而调用返回的过程正好与这个流程相反，图 13.1 是对象流程调用的一个简单示意图。

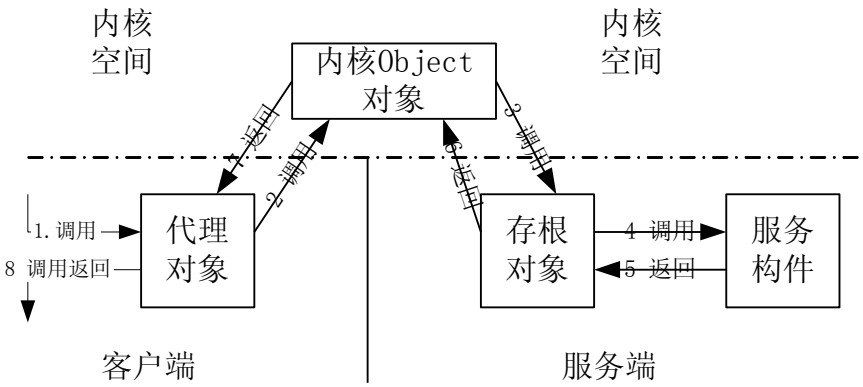


图 13.1 对象流程调用示意图

13.2 从数据项的建立来看 CAR 远程接口列集\散集的整个实现过程

从建立整个列集\散集过程来看,可以分为三部分，一部分是服务创建的过程，主体就是存根对象建立过程以及内核 object 对象的建立过程等；一部分是服务获取的过程，主体就是 proxy 对象被建立的过程；还有一部分是用户进行远程服务调用过程，将在下一节中重点介绍

13.2.1 服务创建的过程

服务器端建立的基本步骤：

1、通过 CAR 构件平台或者说 ElastosSDK 创建一个 CAR 构件对象；

2、创建一个该构件对象相关的存根对象，通过该存根对象可以调用相关的 CAR 构件对象；

3、向内核注册相关存根对象以及构件对象的信息。

服务创建的过程包括元数据的提取以及注册,类实例的创建, 对象存根的创建，内核 object 对象的建立, ExportObject 对象的建立等部分,具体见下面步骤：

1、根据 CLSID 查找提供服务的实现构件类所在的 dll, 如果在则 load 到本进程空间；

2、将相关构件类的元数据(metadata) 拷贝到共享内存区域,并向内核注册相关信息,当前进程保持构件类的元数据信息索引；

3、通过类厂创建类实例；

4、创建一个 stub 对象以及相关信息,并根据元数据初始化相关的数据；

5、向内核注册该服务,并创建相应的 object 对象, 内核分配 oid 资源并返回给 stub 对象,内核 object 对象的建立标志着用户可以通过某种方式远程获得该服务(服务的发布),object 对象保持了相关服务的信息以及创建对象代理所需要的一些信息. 通过 object 对象,可以找到相应的服务进程以及存根对象, 另一方面, 用户通过查询 object 对象,亦可获得创建对象代理所需要的全部信息。系统所有的 object 对象由内核全局管理,并被组织成一个 hash 链表，其主要数据项如图 13.2：

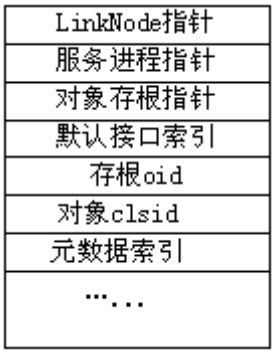


图 13.2 内核 Object 对象结构

6、创建一个 ExportObject 对象，并加入本地的 export(导出)表链中 ，export 表为一 hash 表，每个进程都拥有一个 export 表，一个 ExportObject 对象代表了进程的一个导出 CAR 服务(导出服务意味着其它进程可以通过某种方式获得该服务)， ExportObject 对象记录了 stub 对象指针,以及服务对象指针，stub 的 oid 等信息；

13.2.2 服务获取的过程

客户端建立的基本步骤：

1、查询此远程服务的构件对象和存根对象是否存在，如存在，执行步骤 3；

2、等待服务器端构件和存根对象的创建和注册；

3、查询本进程内的信息，是否已经存在相应的代理，如果存在，则直接返回，否则继续步骤 4；

4 、通过系统调用从内核获得相关构件服务信息；

5 、通过这些信息建立代理对象，并返回。

服务获取的过程，其数据建立主要包括对象代理 和 ImportObject 对象的建立,具体实现见如下流程：

1、用户通过查询 object 对象获得相关的建立对象代理所必须的数据信息，包括:对象存根的 oid，服务进程的 pid，构件对象的 clsid，构件对象的默认接口索引，构件对象的元数据的信息.；

2、首先寻找 export(导出)表中是否存在和对象存根 oid 对应的 ExportObject 对象，如果有，表明该服务是个本地服务，则直接通过 ExportObject 中保存的服务对象指针以及接口索引，找到相关的接口指针并返回,如果没有则走第三步；

3、寻找 import(导入)表中是否存在和对象存根 oid 对应的 ImportObject 对象,导入表是一个进程全局的 hash 表，每个进程都维护一个自己的导入表，当一个进程引入一个远程对象服务的时候，它会将相关的远程对象服务的代理信息存放到该 hash 表中；

4 、如果有，表明该进程已经引入该远程对象服务，则可以通过 ImportObject 对象，找到相应的对象代理，然后根据接口索引找到接口代理并返回给用户；如果没有，则继续第 5 步；

5、创建一个对象代理,并根据传入的对象存根的 oid、 构件对象的 clsid、 构件对象的元数据的信息初始化其数据项,并创建相应的接口代理；

6、将远程对象存根的计数加 1，向构件服务表明存在一个新的使用服务的用户；

7、创建一个相应的 ImportObject 对象，往 import(导入)表中插入该对象；

8、通过接口索引找到新建的对象代理中的相应的接口代理，并返回给用户，整个流程完毕。

13.3 以数据流程的形式分析远程调用的过程

13.3.1 客户端调用远程服务步骤

当确定代理、存根都存在并且从客户端到服务器端建立好一条可以相互通信的通路以后，客户端就可以开始远程调用了，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，代理对象将调用栈里面的数据根据元数据信息打包， 并传给内核

2、内核根据注册信息找到服务以及存根对象，并将打包的信息传给存根对象

3、存根对象根据元数据将数据解包， 并构建和客户端调用栈相应的栈内数据，并调用真正的构件服务接口函数

4、服务构件接口函数完成调用，并返回

5、存根对象获取接口函数调用的参数信息以及返回信息， 并将返回信息以及参数信息打包，并通过系统调用返回到内核

6、内核将服务器端的返回信息传递给客户端， 客户端从系统调用返回

7、代理对象获得返回信息，并根据元数据解包，并回填到用户调用栈中，整个远程构件方法调用过程完成。

13.3.2 客户端调用远程服务图解

以存根\代理机制实现远程接口自动列集\散集， 在操作系统内核构建一层对该机制的支持，将同一进程内的用户空间和内核空间定义为接口不可相互直接访问，进一步根据服务以及服务调用者所在的空间位置不同（如图 13.3 所示），以及空间之间传递数据的策略的不同，采取不同的列集\散集机制，下面一一展开分析。

	用户端	服务器端
第一种情况	进程 A 空间	进程 B 空间
第二种情况	同进程用户空间	同进程内核空间
第三种情况	同进程内核空间	同进程用户空间

图 13.3 服务和服务调用者所在空间

1、对于整个流程来言，很重要的一点是整个数据的传递过程与对象流程调用过程是基本吻合的，图 13.4 按照整个数据流程的形式给出一个进程间远程调用的整个过程，也就是列集\散集的第一种情况：

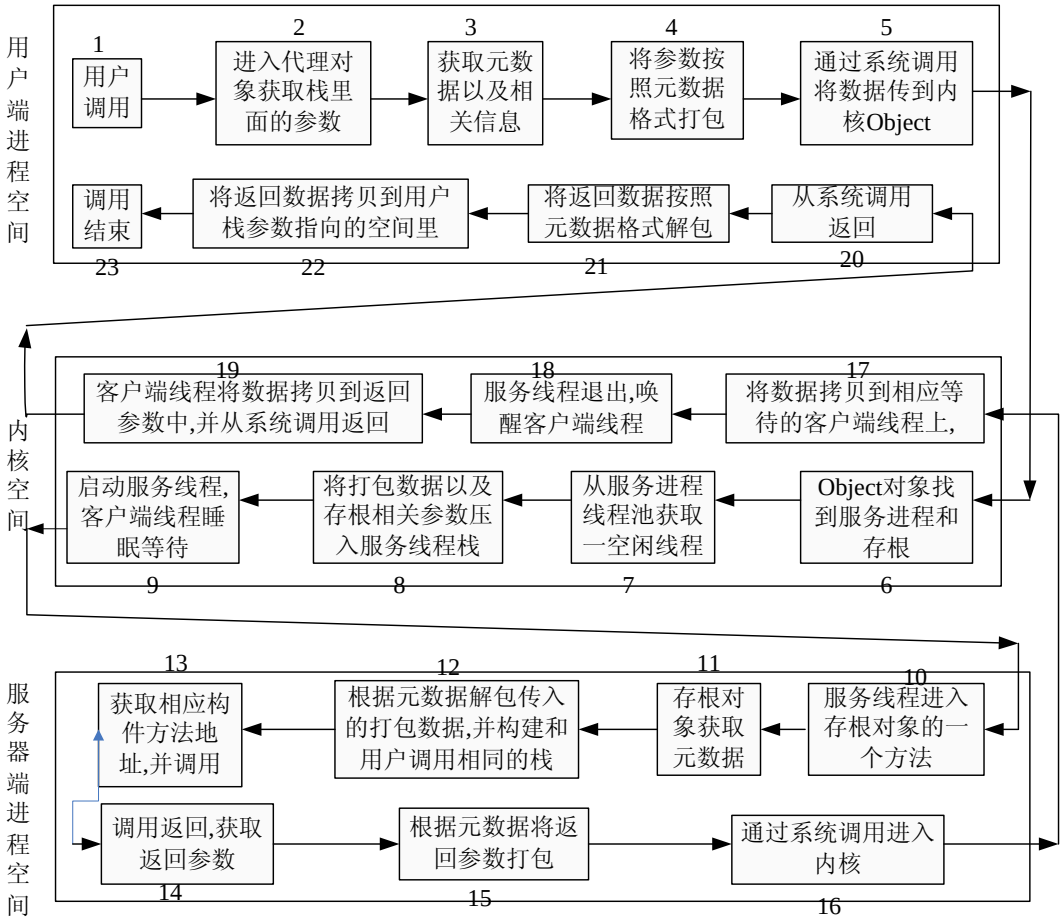


图 13.4 进程间远程调用列集\散集图 1

用户端空间的操作主要是在 proxy 对象中完成的，而服务器端空间的操作则主要是在 stub 对象中完成的，内核 ojbect 对象成了联系客户端和服务器端的枢纽,但服务器端退出，用户端获取数据并返回则是通过线程间同步机制实现的。

实际上第 4 步和第 15 的打包过程就是数据的列集过程，而第 12 步和第 21 步则是数据的散集过程,散集和列集过程都依赖于接口方法的元数据(metadata)。

2、列集\散集的第二种情况，其数据流程处理如图 13.5：

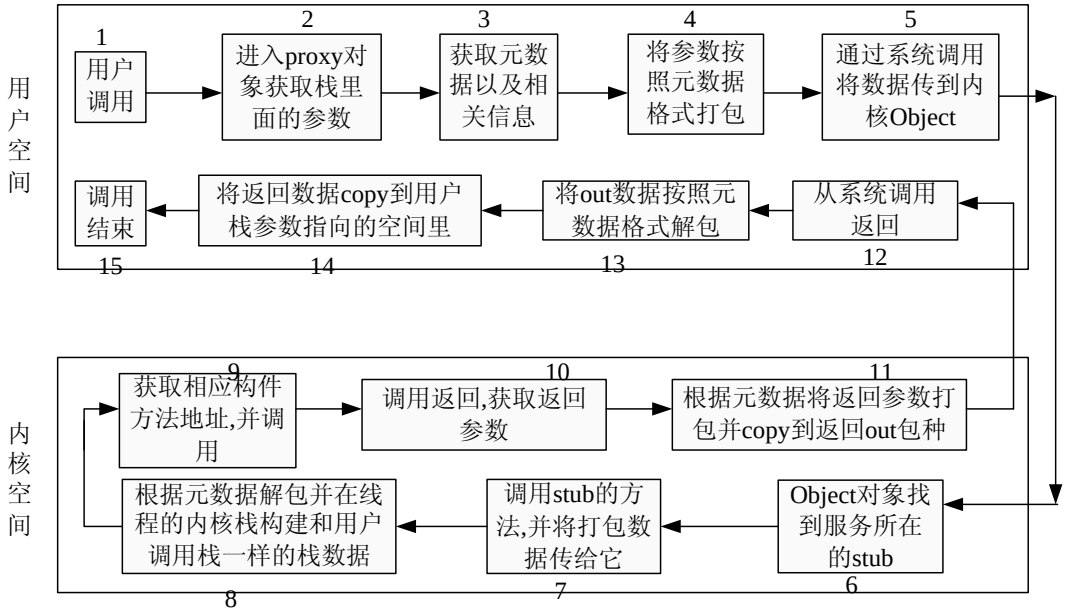


图 13.5 进程间远程调用列集\散集图 2

3、列集\散集的第三种情况，其数据流程处理如图 13.6：

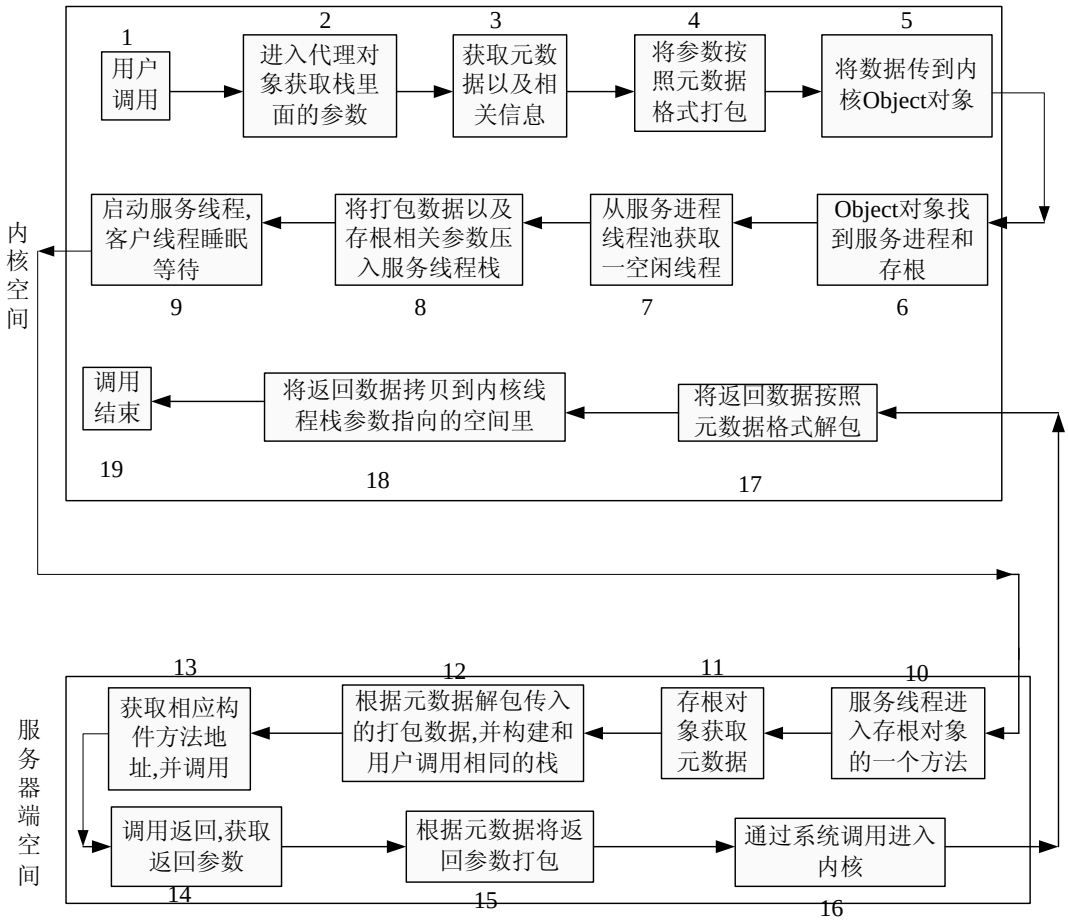


图 13.6 进程间远程调用列集\散集图 3

针对这三种不同的地址空间情况，其处理流程有所不同外，同时根据相应地址空间不同的访问权限，对数据的打包和解包都会有不同的方式。比如第二种情况主要用于内核功能的导出，故效率是最为重要的，其打包和解包过程相对简单，但高效。而第一种情况主要用于进程间通讯，它对支持的接口参数的数据格式更为广泛，且处理也更为完整。

13.4 Elastos 基于引用计数的远程构件生命周期管理的实现机制

为了合理的利用资源，存根和代理不能一直存在于内存中，当客户端进程调用远程服务结束时，代理应该被释放；当没有进程调用对象构件的服务时，此对象构件对应的存根应该被释放（在必要的时候，对象构件也要被释放）。对象存在于内存中的这段时间就是此对象的生命周期，我们应该有一种合理又高效的方法对远程构件调用过程中各对象的生命周期进行管理。

Elastos 平台以类为单位的远程构件生命周期管理的方法是，给代理对象、存根对象和构件对象都设置引用记数，即设置各对象对应的引用计数器、AddRef 方法和 Release 方法。所谓引用计数，就是一个对象被其他对象所引用的次数，如果没有其他对象对其引用了，那么这个对象就没有存在的必要了。调用 AddRef 方法一次，引用计数器值加 1；调用 Release 方法一次，引用计数器值减 1。

当创建构件对象时，设置构件对象的引用计数器值为 0；

当创建存根对象时，设置存根对象的引用计数器值为 0，调用其对应的构件对象的 AddRef 方法一次（被存根对象引用）；

当创建代理对象时，设置代理对象的引用计数器值为 0，调用此代理对象的 AddRef 方法一次（被客户进程引用），并远程调用其对应的存根对象的 AddRef 方法一次（被代理对象引用）；

当客户进程异常退出时，调用对应代理对象的 Release 方法一次（被客户进程的引用断开），在所述客户端的进程对象中删除所述存根对象的相关信息，调用对应存根对象的 Release 方法一次（被对应代理对象的引用断开）；

当服务器端异常退出时，释放所对应的存根对象及其相关的资源，在内核中删除存根对象和构件对象的注册信息；所对应的代理对象会定时的检查存根对象是否存在，当检查到存根对象不存在时，则释放掉代理对象以及与其相关的资源。

当代理对象的引用计数器值为 0，则释放掉代理对象以及与其相关的资源；

当存根对象的引用计数器值为 0，则释放掉存根对象以及与其相关的资源，在内核中删除存根对象的注册信息，并调用对应构件对象的 Release 方法一次（被对应存根对象的引用断开）。

13.5 Elastos 以类为单位实现远程接口的自动列集\散集

在通常的构件技术中，如微软公司的 COM，针对跨域的远程构件调用，是以接口为单位实现列集\散集，而构件本身的实现则以类为单位，即一个构件类可以实现多个接口。从远

程接口的自动列集\散集的实现来看，既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。

对于同一实现类的构件接口，用户通常通过 QI(QuerryInterface) 来获得构件接口,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。假如客户端的一个进程的几个线程分别调用同一远程构件服务的几个不同的接口，就要分别进行远程调用，建立不同的接口代理对象，对各个接口代理对象的生命周期都要进行单独管理。而远程的数据获取,尤其是分布式环境的情况下，是非常耗时的，我们应尽量减少远程获取数据的次数，每次获取,又应尽量带足够多的信息。

针对上述现状，Elastos 选择以类为单位进行远程接口的自动列集\散集的实现。当用户通过 QI 获取接口的时候,所以的处理都在客户端进行，另一方面，该方法把所有对于接口代理或接口存根的生命周期管理最后都归结到对应的类代理或类存根的引用计数的操作上来，这更加符合现实的构件生命周期管理模型，能够减少远程调用次数，降低调用耗时，提高远程构件服务效率。

所谓类代理，就是能用来索引接口代理以及完全代理客户进程的一个代理对象，一个代理对象可以索引多个接口代理。如图 13.7 所示。

用户如果获得远程接口 IA, 通过其 QI IB 的过程实际上是接口代理 IA 通过类代理查询到接口代理 IB 的过程，整个过程都发生在 Client 端空间,而无需进行远程的调用；而对于用户获得的远程接口 IA, IB, IC 的生命周期管理，则体现在类代理上，引用计数器只存在于类代理上，AddRef 方法和 Release 方法也只作用于类代理上，大大减少了相对于以接口为单位的列集方式的远程调用次数。

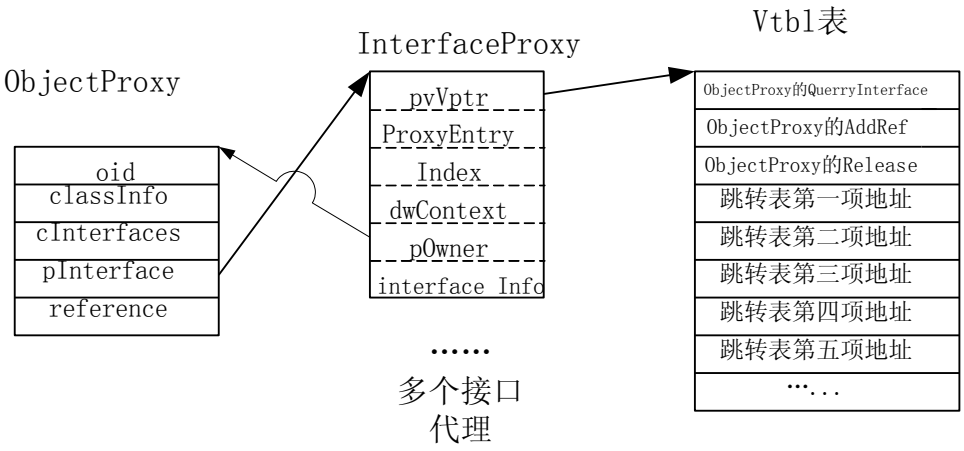


图 13.7　类代理的实现

所谓类存根,就是能用来索引接口存根以及负责代表构件对象和其接口与远程用户进行联系的一个存根对象，一个存根对象可以索引多个接口存根。如图 13.8 所示。

对各接口调用的生命周期的管理，体现在类存根上，引用计数器只存在于类存根上，AddRef 方法和 Release 方法也只作用于类存根上。

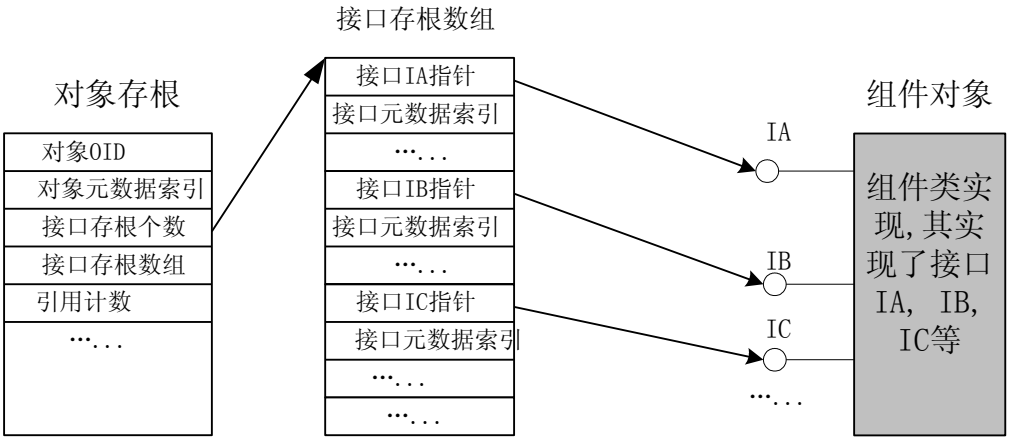


图 13.8　类存根的实现

如图 13.9 所示，当用户要求获得相应的远程接口的时候，系统就根据对象代理的接口索引在客户端返回相应的接口代理给用户。通过接口代理传递过来的调用请求，会通过对象存根的接口索引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。

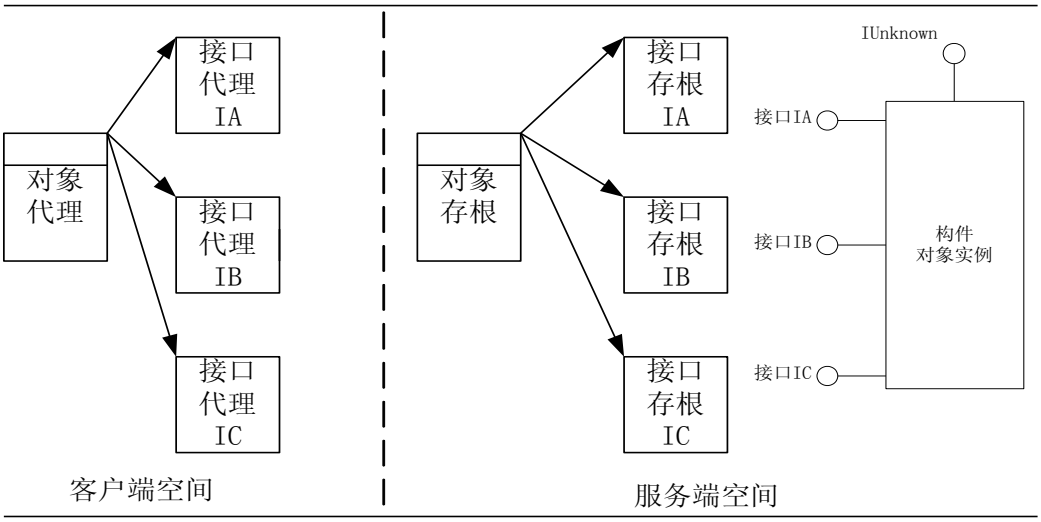


图 13.9 各对象之间的关系

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，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，存根和构件实现类的实例则是一对一的关系，如下图：

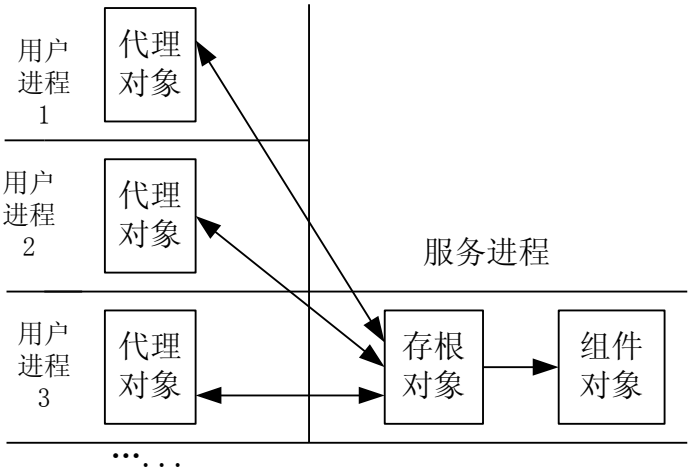


图 13.10 代理、存根和构件对象的对应关系

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 hold 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同样的 oid，这样代理对象就可以找到相应的存根对象，从上图也可以看出，一个存根对象可以对于多个代理对象，所以存根对象是无法直接找到某个特定的代理对象的，实际上我们是通过线程的同步机制来实现存根对象找到相应的代理对象的。

13.6 CAR 构件自定义列集/散集机制

CAR 构件自定义列集/散集机制（custom marshaling）是 CAR 构件运行平台提供给开发者的一种针对构件实例接口指针的自定义列集/散集方法，使开发者可以在远程传递接口指针类型参数时控制传递过程并影响返回结果。

从设计思想上来看，自定义列集/散集机制一方面是一种基于 CAR 构件运行平台本身所提供的远程过程调用技术之上的扩展机制;另一方面它的运行又依赖于 CAR 构件运行平台本身所提供的远程过程调用技术。与其它同类技术（例如 Microsoft DCOM 中的自定义机制）相比，它最显著的优点是逻辑结构更为清晰，而且在设计 CAR RPC 时无需为 CAR Custom marshaling 作额外考虑，减轻设计负担。

要使用CAR构件自定义列集/散集,只需在构件中声明并实现ICustomMarshal接口即可。

```
ICustomMarshal {
    ECode GetClsid([out] ClassId* pClassId);
    ECode CreateObject([in] IObject *pOriginProxy, [out] IObject
        **ppNewProxy);
}
```

CAR 构件自定义列集/散集机制分为如下四个运行阶段：

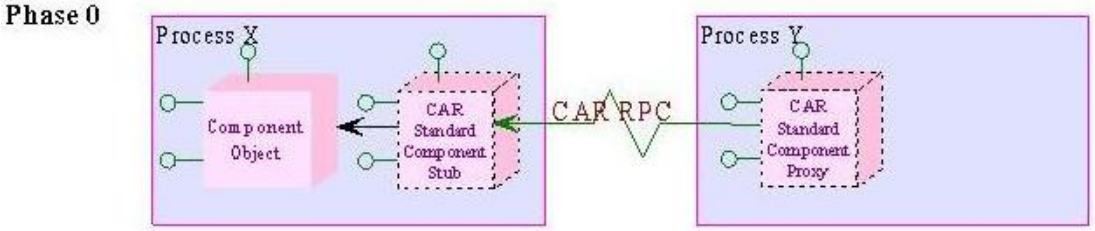


图 13.11 自定义列集/散集运行阶段一

第一阶段，CAR 构件运行平台使用 CAR RPC 的标准列集/散集技术对构件实例进行列集/散集。该过程结束后，在构件实例所运行的服务进程（X）会创建一个 CAR 标准构件存根，同时在客户进程（Y）内会创建出一个 CAR 标准构件代理，这一过程实质上就是 CAR RPC 标准列集/散集机制的实现过程。

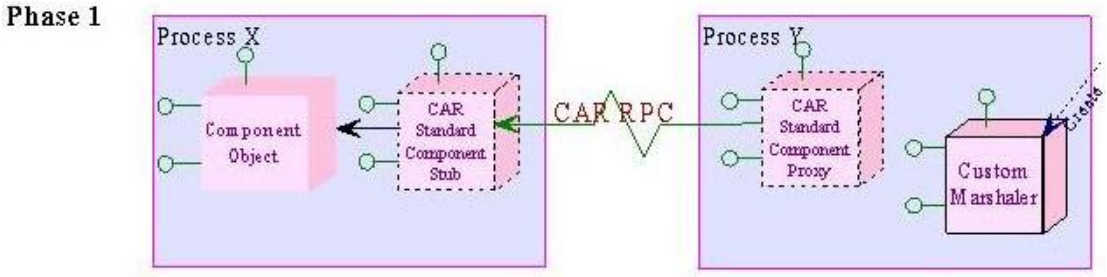


图 13.12 自定义列集/散集运行阶段二

第二阶段，CAR 构件运行平台通过查找元数据来判断该构件是否实现了 ICustomMarshal 接口。如果实现了 ICustomMarshal 接口，则返回 CAR 标准构件代理（即 CAR RPC 标准列集/散集机制结束）。否则将通过调用 CAR 标准构件代理的 QueryInterface 方法获取该构件的 ICustomMarshal 接口指针，然后调用该接口指针的 GetClsid 方法获得其 Custom Marshaler 构件对象的 ClassId，最后利用 EzCreateObject 在进程 Y 创建出一个新的 Custom Marshaler 构件对象实例（注意：Custom Marshaler 构件也必须实现了 ICustomMarshal 接口）。

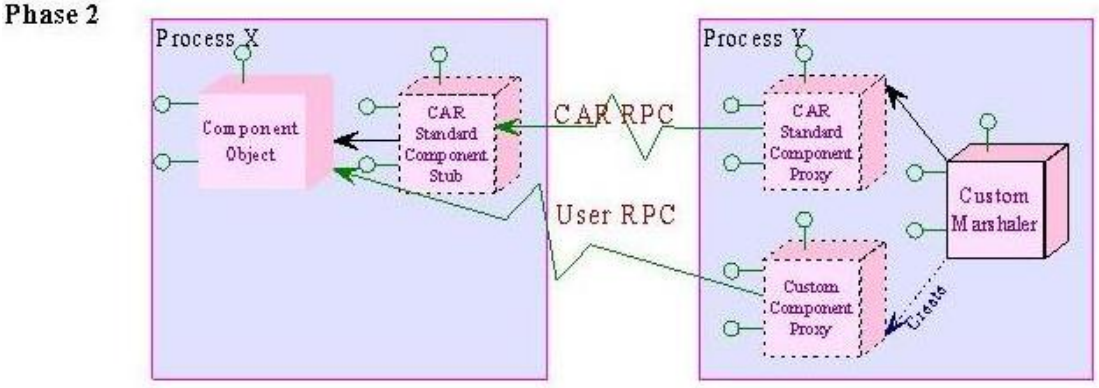


图 13.12 自定义列集/散集运行阶段三

第三阶段，CAR 构件运行平台以 CAR 标准构件代理的接口指针作为输入参数 pOriginProxy 调用 Custom Marshaler 构件实例 ICustomMarshal 接口的 CreateObject 方法，然后将该方法的输出参数 ppNewProxy 作为自定义构件代理返回（ppNewProxy 即上图中 Custom component proxy 的对象指针）。

从对该阶段的描述可知，在 CreateObject 方法的实现中如何利用接收的 pOriginProxy 指针创建出 Custom component proxy 完全由 Custom Marshaler 构件的实现来控制。同时，为了保证下次 Custom component proxy 构件对象被列集/散集时仍然使用自定义列集/散集机制，该构件也应该实现 ICustomMarshal 接口。

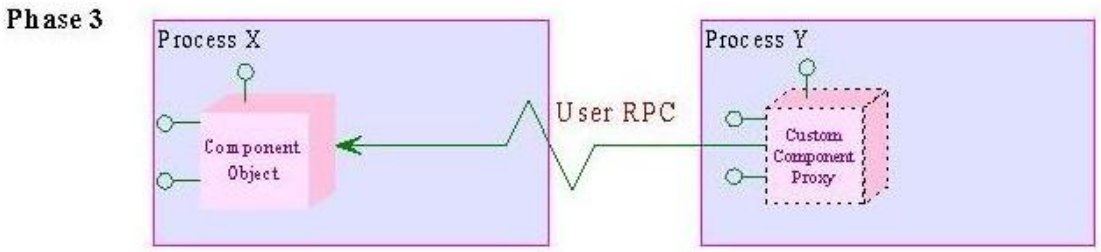


图 13.14 自定义列集/散集运行阶段四

第四阶段，CAR 构件运行平台调用其接口 Release 方法删除临时使用的 CAR 标准构件代理对象和 Custom Marshaler 对象。

该阶段完成后，Custom component proxy 与进程 X 中的构件对象实例的 CAR RPC 通信链路被断开，Custom component proxy 对象需要自己负责与进程 X 中的构件对象进行通信（注意：Custom component proxy 也可以通过保存 CAR 标准构件代理对象的指针并调用 AddRef 来保留这条通信链路，具体情况由 Custom component proxy 构件的实现决定）。

第十四章　命名服务机制

14.1 命名服务的简介

现有的客户端/服务器模式，客户端如果要使用服务器端的某种服务，以往我们采用的方式是通过列集/散集来完成对远程接口的调用，而 Elastos 的命名服务机制为客户端程序提供了一种查询机制，可以通过字符串形式的名称查询在服务器端有没有某种服务，如果有就可以使用，如果没有服务器端会返回错误通知，告知客户端没有相应的服务。服务的使用过程屏蔽了列集/散集等具体调用细节，向用户提供的是完全透明的服务。

14.2 命名服务的原理

命名服务机制的实质是将一个构件和指定的字符串绑定的过程，构件使用者可以远程通过字符串查询该构件，并获得构件服务。命名服务本身即可以作为一个单独的构件存在，亦可以作为内核功能的一部分。

远程服务构件在创建的时候会向内核注册相关信息，并建立存根，称为 Stub，远程用户获得构件指针的时候在自己进程空间建立代理，称为 proxy。

远程服务构件在内核注册的信息代表了该构件对象的存在，我们通过一个内核对象 Object 代表某个构件的注册信息，通过此信息，可以找到相关建立代理所必须的信息，另一方面，也可通过这些信息找到该远程构件以及构件服务相关信息。

如图 14.1 所示，命名服务的实现主要是通过在内核创建代表一个命名服务的对象 NameHook， 然后将该对象与服务对象在内核的注册信息 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 则可以发现相应构件对象的 Object，通过 Object 又可获得足够建立 Proxy 的信息。

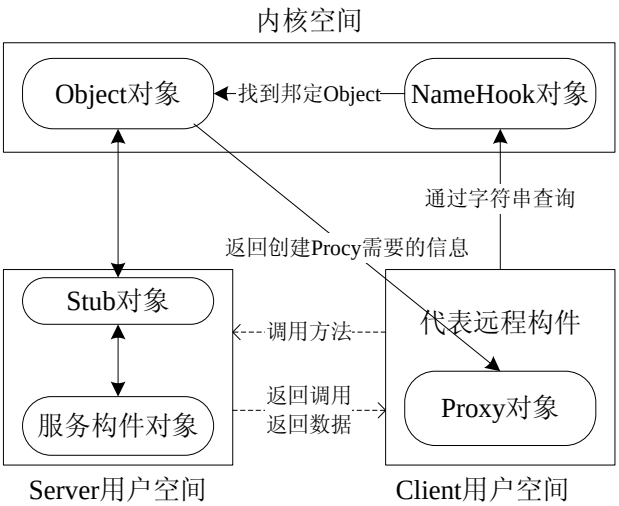


图 14.1 命名服务原理图

14.3 命名服务的步骤

命名服务的整体过程包括了四大部分：等待注册、注册、使用和注销。以下就每个部分所对应的函数代码作具体的分析。

14.3.1 等待注册命名服务

用户进程用函数 EzWaitForService 等待指定的命名服务被注册。输入参数为指定的服务名字和最长等待时间。

具体步骤如下：

1、在 NameHook 对象所组成的链表中查找是否有指定名字的服务，如有，则表示等待的服务已经被注册，可以立刻返回并使用该服务；

2、如果没有找到指定名字的服务，则把记录该服务名字的线程放入等待队列；

3、在设定的最长等待时间范围内，如果该线程被服务器端注册这个命名服务的线程唤醒，则可以使该线程退出等待队列，然后返回并使用该服务。

4、如果等待时间超过了设定的最长等待时间，则该线程退出等待队列并返回超时出错信息。

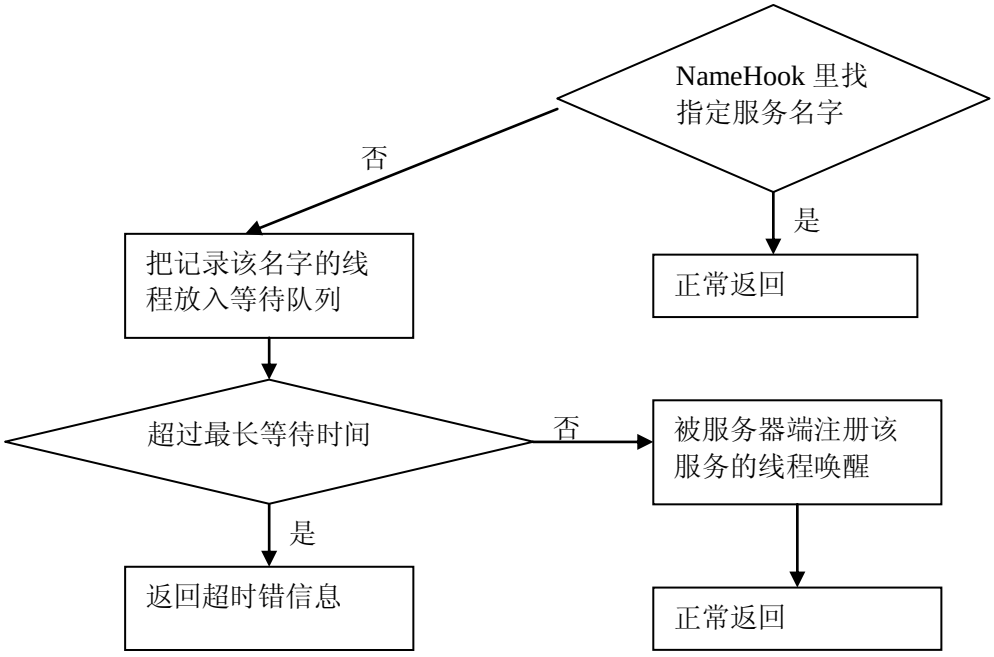


图 14.2 EzWaitForService 流程图

14.3.2 注册命名服务

服务器端通过函数 EzRegisterService 注册命名服务。该函数将一个字符串与一个构件接口相绑定，实际上这里的构件接口可能是一个远程构件接口代理，也可能是一个本地对象接口指针，这样就分成两种情况做不同的处理，如图 14.3 所示。

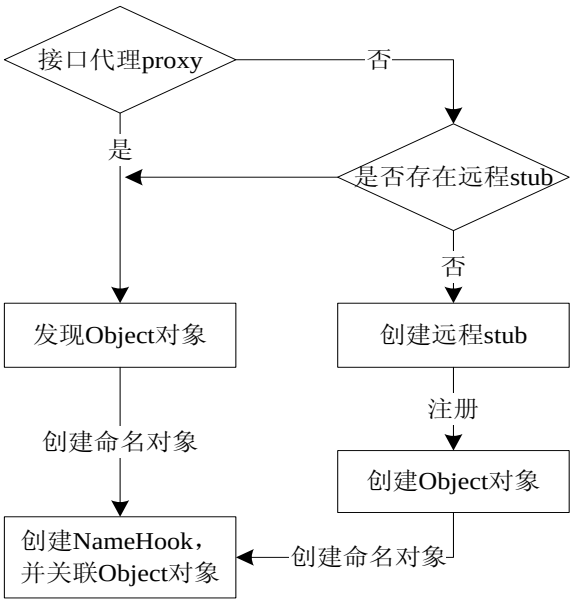


图 14.3 EzRegisterService 流程图

1、接口是一个远程构件接口代理，具体处理步骤如下：

1）通过接口代理，找到相应的 Object；

2）如发现要注册的名字已存在或者该 Object 对象已被注册为其它名字，则返回出错信息；

3）创建 NameHook 对象，并和 Object 对象建立关联；

4）服务提供者的进程的引用计数加一，使其被引用而继续存活在内存空间中；

5）唤醒客户端所有等待该服务的进程，告知它们注册工作已经完成，可以通过 EzFindService 函数来查找并使用该服务；

6）释放之前的接口代理，因为以后可以通过命名服务直接找到该服务的接口。

2、接口是一个本地对象接口指针，具体处理步骤如下：

1）创建服务对象对应的 stub 对象，并使其能映射服务对象的各个接口；

2）使 stub 对象的引用计数加一；

3）使服务对象的引用计数加一；

4）如发现要注册的名字已存在或者该 Object 对象已被注册为其它名字，则返回出错信息；

5）在内核中创建服务对应的 Object 对象，并把服务对象的相关信息写入 Object 对象的数据结构中，也就是通常所说的向内核注册相关信息；

6）创建 NameHook 对象，并和 Object 对象建立关联；

7）唤醒客户端所有等待该服务的进程，告知它们注册工作已经完成，可以通过 EzFindService 函数来查找并使用该服务。

14.3.3 使用命名服务

客户端的程序可以利用函数 EzFindService 来查询系统中的某个服务。如果服务存在，则客户端程序获得了服务构件的接口指针，从而可以调用服务构件的各种方法。如果服务不存在，客户端会被告知其查询的服务不存在。

具体步骤如下：

1、通过系统调用陷入内核，在全局的 NameHook 对象 hash 表中查找是否存在已知名字的服务，不存在则返回出错信息；

2、找到该名字的 NameHook 对象，然后就可以找到对应的 Object 对象，根据 Object 对象的服务进程指针分三种情况讨论：

1）服务进程指针等于当前进程指针，则该服务构件在当前进程的用户空间,则通过 NameHook 对象保持的 stub 对象获得构件服务指针(存根对象保持了构件对象的指针)，设置 pDomainInfo 的值为 CTX_SAME_PROCESS (pDomainInfo 代表了服务和服务调用者的位置关系)；

2）服务进程指针为 NULL，表明构件服务是一个内核对象，则将返回参数 pDomainInfo 设置为 CTX_USER_2_KERN；

3）服务进程指针不等于 NULL，也不等于当前进程指针，证明其是另一个进程所提供的构件服务，则将 pDomainInfo 设置为 CTX_DIFF_PROCESS。

3、将 object 对象保存的四个数据:：oid、默认接口索引、元数据指针、构件对象的 clsid 和服务进程的 pid 复制到一个 iPack 接口包中，通过系统调用返回给用户空间。

4、根据 oid 查找 export(导出)表中是否存在对应的 ExportObject 对象，如果有,表明该服务是个本地服务，则通过 ExportObject 对象找到 stub 对象,再根据 iPack 中的默认接口索引，通过接口存根找到服务对象的相关接口指针并返回给用户,同时使该接口的引用计数加一，使 stub 对象的引用计数减一，也就是用户可以直接调用服务对象而不用通过 stub 对象来调用，函数 EzFindService 结束；如没有找到 ExportObject 对象则执行 5；

5、根据 oid 查找 import（导入）表中是否存在对应的 ImportObject 对象（导入表是一个进程全局的 hash 表，每个进程都维护一个自己的导入表，当一个进程引入一个远程对象服务的时候，它会将相关的远程对象服务的代理信息存放到该 hash 表中），如果有，表明该进程已经引入该远程对象服务，则可以通过 ImportObject 对象，找到相应的对象代理，然后根据默认接口索引找到接口代理并返回给用户，函数 EzFindService 结束；如没有找到 ImportObject 对象则执行 6；

6、根据 iPack 接口包中的信息找到服务提供者的进程，使其引用计数加一，从而被引用而继续存活在内存空间中；

7、创建一个类代理,并根据传入的 oid，服务构件的 clsid，服务构件的元数据的信息初始化其数据项,并创建相应的接口代理数组，使各接口代理的 m_pvVptr 指针指向各张虚表。

14.3.4 注销命名服务

一旦某个服务构件不再被任何客户端程序所调用，系统可以用函数 EzUnregisterService 将其注销，并释放命名服务所占有的所有构件接口指针。

只有注册命名服务的那个进程才有权取消自己注册的服务，其它进程调用本函数则会返回失败。

一旦调用本函数取消指定的命名服务后，使用 EzFindService 函数将不再能获取到该服务的接口，但这并不影响之前已经通过 EzFindService 获取的服务接口，除非该服务进程已经退出。

具体步骤如下：

1、找到待注销的服务的 NameHook 对象；

2、取消 NameHook 与其对应的 Object 对象的关联；

3、调用 Release 方法使 stub 对象和服务对象的引用计数都减少而被释放；

4、释放 NameHook 对象。

14.4 命名服务机制示例

在这个例子中我们使用 dll 形式的 CAR 构件，通过这个服务可以实现在控制台下打印一个字符串。我们用客户端程序来调用这个构件。

在 hello.car 文件中构件，实现类，接口以及方法的声明表示如下：

```
module  // 构件 Hello
{
    interface IHello { // 接口 IHello
        Hello([in] WString inStr); // 方法 Hello
    }
    class CHello {   //实现类 CHello
        interface IHello;
    }
}
```

在 CHello.h 以及 CHello.cpp 文件中声明了 CHello 对象，并实现接口方法 Hello。

CHello.h 文件中主要代码表示如下：

```
CarClass(CHello) // CHello 定义
{
    public:
        CARAPI Hello(/* [in] */ WString inStr);
};
```

CHello.cpp 文件中主要代码表示如下：

```
// Hello 方法的实现代码
ECode CHello::Hello(/* [in] */ WString inStr)
{
    printf("%S\n", (wchar_t*)inStr);
    return NOERROR;
}
```

在 server.cpp 中，我们将创建一个 Hello 的构件，并将其注册为以“hello” 为标志的命名服务，再通过系统 API 函数 EzCreateEvent 获取一个内核 Event 对象服务，并将其注册为“event”为标志的命名服务，然后唤醒所有等待该服务注册的线程，并使自己进入等待状态，当被通知服务使用完毕时，注销服务，并释放相关资源。程序被编译成 server.exe，主要代码如下：

```
#include <stdio.h>
#include <eladef.h>
#include <elastos.h>
#import <Hello.dll>

int main()
{
    ECode ec;
    WaitResult pResult;
    IHello *pIHello;    // 声明 IHello 接口指针
    IEvent *pIEvent;      // 声明 Event 接口指针

    // 在本进程空间内创建一个 Hello 构件
    ec = CHello::New(&pIHello);
    if (FAILED(ec)) {
        printf("NewHelloerror\n");
        return 1;
    }
    else
        printf("NewHello\n");
    // 注册命名服务，将 CHello 构件与字符串“hello”绑定
    ec = EzRegisterService(L"hello", pIHello);
    if (FAILED(ec)) {
        printf("Registerhelloerror\n");
        pIHello->Release();
        return 1;
    }
    else
        printf("Registerhello\n");
    // 通过 API 函数获得一个内核 Event 对象服务
    ec = EzCreateEvent(true, false, &pIEvent);
    if (FAILED(ec)) {
        printf("CreateEventerror\n");
        pIHello->Release();
        return 1;
    }
```

```
        else
            printf("CreateEvent\n");
        // 注册命名服务，将 Event 构件与字符串“event”绑定
        ec = EzRegisterService(L"event", pIEvent);
        if (FAILED(ec)) {
            printf("Registereventerror\n");
            pIHello->Release();
            pIEvent->Release();
            return 1;
        }
        else
            printf("Registerevent\n");

        // 让该进程等待
        pIEvent->Wait(&pResult,NULL);
        // 被其它进程唤醒，等待结束，注销服务，释放资源
        ec = EzUnregisterService(L"event"); // 注销 Event 服务
        if (FAILED(ec)) {
            printf("Unregistereventerror\n");
            pIHello->Release();
            pIEvent->Release();
            return 1;
        }
        else {
            printf("Unregisterevent\n");
            pIEvent->Release(); // 释放 pIEevent 指针
        }
        ec = EzUnregisterService(L"hello"); // 注销 hello 服务
        if (FAILED(ec)) {
            printf("Unregisterhelloerror\n");
            pIHello->Release();
            return 1;
        }
        else {
            printf("Unregisterhello\n");
            pIHello->Release();  // 释放 pIHello 指针
        }
        return 0;
    }
```

在 client.cpp 中，我们首先创建一个和它并发地服务器端进程 server.exe,之后通过 EzWaitForService 函数来等待以“hello” 和 “event”为标志的命名服务被注册，当该线程被服务器端注册“hello”和 “event”的线程唤醒后，使用 EzFindService 函数来查找以“hello”以及“event”为标志的命名服务，通过命名服务，我们找到与名字绑定的相关构件服务，并进行调用，调用完释放掉相关的构件服务指针，并通知服务器端。主要代码如下：

```
#include <stdio.h>
#include <eladef.h>
#include <elastos.h>
#import <Hello.dll>

int main()
{
    ECode ec;
    IHello * pIHello; // 声明一个 IHello 接口指针
    IEvent * pIEvent; // 声明一个 IEvent 接口指针
    WaitResult pResult;
    // 创建服务器端进程
    ec = EzCreateProcess(
                L"server.exe", NULL, NULL);
    if (FAILED(ec)) {
        printf("Create server process failed. ec = 0x%08x\n", ec);
        return 1;
    }
    else
        printf("CreateProcess\n");

    // 等待以“hello”为名字的服务构件的注册
    ec = EzWaitForService(L"hello", INFINITE,&pResult);
    if (FAILED(ec)) {
        printf("WaitForhelloerror\n");
        return 1;
    }
    else
        printf("WaitForhello\n");
    //等待以“event”为名字的服务构件的注册
    ec = EzWaitForService(L"event", INFINITE,&pResult);
    if (FAILED(ec)) {
        printf("WaitForeventerror\n");
        return 1;
    }
```

```
        else
            printf("WaitForevent\n");
        // 通过字符串“hello”找到相关构件服务
        ec = EzFindService(L"hello", (IObject**)&pIHello);
        if (FAILED(ec)) {
            printf("Findhelloerror\n");
            return 1;
        }
        else {
            printf("Findhello\n");
            // 远程调用 Hello 方法，打印”hello, world”
            pIHello->Hello(L"helloworld\n");
            pIHello->Release(); // 释放获取的 IHello 接口指针
        }
        // 通过字符串“event”找到相关构件服务
        ec = EzFindService(L"event", (IObject**)&pIEvent);
        if (FAILED(ec)) {
            printf("Findeventerror\n");
            return 1;
        }
        else {
            printf("Findevent\n");
            // 通过 notify 方法唤醒 server 进程
            pIEvent->Notify(1);
            pIEvent->Release(); // 释放获取的 IEvent 接口指针
        }
        return 0;
    }
```

编译链接成 nameservice.exe，由于两个进程交替执行，所以注册和等待注册的打印顺序不定，以下为其中一种运行结果：

```
    CreateProcess
    NewHelloright
    WaitForhello
    Registerhello
    CreateEvent
    WaitForevent
    Registerevent
    Findhello
    helloworld
    Findevent
    Unregisterevent
```

```
Unregisterhello
```

14.5 Elastos 命名服务机制的优点

通过上述示例，展示了 Elastos 命名服务机制的优点：

1、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。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。

2、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。

3、从简单性来看，命名服务机制以简单的四个 API(EzWaitForService，EzRegisterService, EzFindService，EzUnregisterService,)函数，提供等待服务，到服务构件和字符串的绑定，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。并且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。

第十五章　构件回调机制

15.1 回调

15.1.1 什么是回调（CALLBACK）

回调是一种双向调用模式，也就是说，被调用方在接口被调用时也会调用对方的接口。讲的更具体一点，就是模块 A 调用模块 B，而模块 B 中又存在调用模块 A 中的一个函数 c 的代码，图 15.1 所示的形式就叫回调，其中的函数 c 就是回调函数：

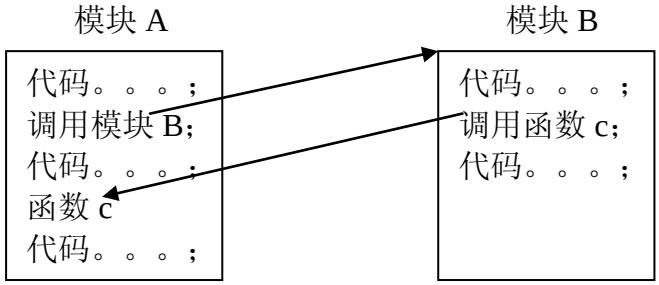


图 15.1 回调示意图

15.1.2 回调的用处

如以上模型，A 要调用 B，可是又希望 B 在处理过程中用到 A 中的某些信息，具体是什么信息，怎么处理这些信息，都由 A 决定。而 B 只要知道调用 A 中的函数 c 能得到这方面的信息就行，函数 c 怎么实现就不用管了。

其实 B 不一定由 A 来调用，当 B 所在的系统发生某件事件时，系统会自动调用 B 并通过 B 模块来调用由 A 定义的函数 c，起到了 B 对 A 的通知作用。

15.1.3 回调在客户/服务器模型或构件调用模型中的应用

客户端（模块 A 地址空间）首先通过同步方式调用服务端（模块 B 地址空间）的注册接口来注册回调接口，服务端收到该请求以后，就会保留该接口引用，如果发生某种事件需要向客户端通知的时候就通过该引用调用客户方的回调函数，以便对方及时处理。

构件的使用方式类似于“客户/服务器”模型，构件充当服务器的角色。对一般的构件来说，客户与构件之间的通信过程是单向的，客户创建构件对象，然后客户调用对象所提供的接口函数，我们称这种通信接口为入接口。而对于一个全面的交互过程来说，这样的单向通信往往不能满足实际的需要，有时候构件对象也要主动与客户进行通信，因此，与入接口相对应，构件也可以提供出接口（也就是回调接口）与客户进行通信。

15.2 CAR 构件的回调机制

在 CAR 构件技术里，支持一个或多个回调接口的 CAR 构件对象叫做可连接对象。可连接对象支持两种接口，一种是普通接口，它里面的成员函数由服务器端自己实现；另一种就是回调接口，其中每个成员函数代表一个事件（event），处理每个事件的函数由客户端实现。当特定事情发生时，如定时消息或用户鼠标操作发生时，构件对象产生一个事件，客户程序可以处理这些事件。

Microsoft 提供的可连接对象技术需要用户去实现客户程序与构件对象的连接、事件的激发、接收器的编写等，而且只能以接口为单位注册，即不能为接口中每个成员方法分别注册。在 CAR 的回调事件机制中，客户程序可以单独注册事件的某一个事件处理函数，大大简化了编程。

在 CAR 的回调机制中，也有一种接收器(sink)对象，它的方式和作用都与前者有区别。CAR 里的接收器对象相当于一个客户端回调函数的容器，在客户端的地址空间里，负责与可连接对象进行通信。只要有可连接对象存在，那么在客户端肯定要有接收器的存在。多个回调接口可对应一个接收器对象，这样可以减少通信花费的开销。当接收器与可连接对象建立连接后，客户程序可将自己实现的事件处理函数（回调函数）向接收器进行注册，而不是向可连接对象进行注册，又一次减少了通信开销。接收器会自动把在它里面注册的函数的回调接口指针告诉构件对象，构件对象在条件成熟时激发事件，如果客户注册了自己的回调接口方法，那么就会被调用，否则就调用接收器默认实现的回调接口方法。事实上，接收器是向客户程序屏蔽的，客户只需通过 AddXxxCallback 和 RemoveXxxCallback（Xxx 代表回调事件名）函数来注册和注销事件处理函数，完全不用关心接收器的存在。

在编写构件程序时，用户只需关心何时激发事件，而在编写客户端程序时，用户只需在适当的时候注册事件处理函数。其它的工作，如接收器对象的创建与实现、接收器与可连接对象建立通信的具体过程、事件的分发回调过程等都由 CAR 实现。这样，用户在编写具有回调接口的构件和编写使用该构件的客户端程序都会变得相当简单。

15.3 CAR 构件回调机制实现的一个示例程序

现以大楼火警为例说明回调机制的实现过程，这个例子里有一个 server(Building.dll)和一个 client(People.exe)。

15.3.1 服务器端的实现

Building.car 文件如下：

```
module
{
    interface IBuilding {
        OnFire();
    }
    callbacks JFireAlarmRing {
        Employee();
```

```
            Fireman();
        }
        class CBuilding {
            interface IBuilding;
            callbacks JFireAlarmRing;
        }
    }
```

类 CBuilding 实现了两个接口，一个是普通接口 IBuilding，它有一个方法 OnFire()，一个是回调接口 IFireAlarmRing，它有两个事件方法 Employee()和 Fireman()。

经过 CAR 工具编译后，我们实现 OnFire()方法如下：

```
#include "CBuilding.h"
#include "_CBuilding.cpp"
ECode CBuilding::OnFire()
{
    // TODO: Add your code here
    CConsole::WriteLine("The building is on fire!\n");
    //触发事件，以下两个事件方法让客户端实现
    Callback::Employee();

    Callback::Fireman();
    return NOERROR;

}
```

在 CBuilding 的 OnFire 方法里我们触发了事件 Employee 和 Fireman，这两个方法我们会在客户端实现并注册。

编译服务器端程序后会得到构件 Building.dll。

15.3.2 客户端的实现

在客户端的 People.cpp 文件如下：

```
#include "Building.h"
using namespace Elastos;

//实现回调事件方法 EmployeeRunAway
ECode EmployeeRunAway(PVoid userData, PInterface pSender)
{
    CConsole::WriteLine("Employees are running awary...\n");

    return NOERROR;
}
//实现回调事件方法 FiremanRushIntoFire
```

```
    ECode FiremanRushIntoFire(PVoid userData, PInterface pSender)
    {
        CConsole::WriteLine("Firemen are fighting with fire...\n");
        return NOERROR;
    }

    ECode ElastosMain(const BufferOf<WString>& args)
    {
        ECode ec;
        IBuilding* pIBuilding;
        CResult fireResult;
        ec = CBuilding::New(&pIBuilding);
        if (FAILED(ec)) {
            return ec;
        }

        pIBuilding->OnFire();

        ec = CBuilding::AddEmployeeCallback(pIBuilding, EmployeeRunAway, NULL);
        if (FAILED(ec))
        {
            pIBuilding->Release();
            return ec;
        }
        ec = CBuilding::AddFiremanCallback(pIBuilding, FiremanRushIntoFire,
NULL);
        if (FAILED(ec))
        {
            pIBuilding->Release();
            return ec;
        }
        pIBuilding->OnFire();

        CObject::ReleaseAtThreadQuit(pIBuilding);
        return NOERROR;
    }
```

编译客户端程序并运行 People.exe，会得到如下结果：

```
    The building is on fire!
    Employees are running awary...
    Firemen are fighting with fire...
```

15.3.3 分析客户与可连接对象通信的各个过程

Building.car 文件经过 CAR 工具编译后，自动生成 AddXxxCallback 和 RemoveXxxCallback（Xxx 代表回调事件名）函数提供给客户端使用来注册和注销事件处理函数。

回调处理函数的第一个参数必须是第一个参数为 Pvoid 类型传递用户参数，另外一个 PInterface 类型传递接口指针类型，如：

```
EmployeeRunAway(PVoid userData, PInterface pSender)
```

以下结合上述例子具体介绍客户与可连接对象从建立连接到断开连接的整个过程。

1、建立连接并注册回调处理函数

在客户端第一次注册回调处理函数时，注册函数 AddXxxCallback 会创建一个和可连接对象有关联的接收器，并把回调函数注册到该接收器中，之后每一次注册其它回调函数时，只要找到这个接收器，然后直接把回调函数注册到该接收器中就行了。

客户注册了事件函数后，接收器对象保存了该函数指针。例如：

```
CBuilding::AddEmployeeCallback(pIBuilding, EmployeeRunAway, NULL);
```

语法形式一：

第一个参数是 server 的接口指针；

如果 callback 处理函数是普通方法，第二个参数就是函数指针；如：

```
CBuilding::AddEmployeeCallback(pBuildingObj, &EmployeeRunAway);
```

语法形式二：

如果回调函数是类成员函数，而这类函数引用的第一个参数在 C++里是 this 指针（就是当前类对象指针），CAR 需要了解类指针，才可以在操作具体的回调函数时把 this 指针传进去。所以有下面的语法形式。

如果 callback 处理函数是类成员方法，第二个参数是类对象指针，第三个参数是类成员函数指针。如：

```
CBuilding::AddFiremanCallback(pBuildingObj, &fireResult,
&CResult::FiremanPutFireOut);
```

2、事件激发

在上述例子里，客户端通过 pBuilding 指针调用入接口函数 OnFire，而在可连接对象里，OnFire 方法触发 Employee 事件，通过其所保存的回调接口 IFireAlarmRing 的接口指针调用 Employee 方法，这时就进入了接收器对象，客户对事件 Employee 注册了自己的实现方法，接收器对象会检测到客户的函数指针并调用之，至此，事件激发过程结束。

3、事件注销

客户可以通过 RemoveXxxCallback（Xxx 代表回调事件名）函数来注销自己曾注册的事件。这个函数的参数和 AddXxxCallback 函数参数一样；一旦客户调用这个函数成功，那么客户注册的事件函数指针就会从接收器对象里去掉，事件激发时就只能调用到接收器对该事件的默认实现（只是简单地返回 NOERROR）。在上面的例子里，客户向接收器对象注销自己

实现的事件函数 EmployeeRunAway 后，再通过 pBuilding 指针调用入接口函数 OnFire 时，字符串“Employees are running awary...”就不会打印出来了。

4、断开连接

当 proxy(或本地 server)release 到 0 时接收器会自动中止客户和可连接对象的双向通信，这时接收器对象和可连接对象不再有任何关系，可连接对象不再保存接收器对象实现的回调接口指针，也就不能触发回调接口的事件了。最后接收器自己也会自动释放掉。

第十六章　构件缓存机制

16.1 CAR 构件缓存机制介绍

16.1.1 IE 缓存机制简介

为了提高访问网页的速度，Internet Explorer 浏览器会采用累积式加速的方法，将曾经访问的网页内容（包括图片以及 cookie 文件等）存放在电脑里。这个存放空间，就称为 IE 缓存。以后每次访问网站时，IE 会首先搜索这个目录，如果其中已经有访问过的内容，那 IE 就不必从网上下载，而直接从缓存中调出来，从而提高了访问网站的速度。

具体的工作原理是，在浏览网页时，浏览器会把从网上读出的网页、图像以及其他数据存放在磁盘缓存之中，并建立相应的文档索引。按照信息存放的位置可分成内存缓存和硬盘缓存两种。内存缓存是用于暂时存储本次上网所调用的数据资料的，从 Internet 上传来的每一个网页信息，在内存缓存中都相应地给予保存一个备份，“前进”和“后退”是从内存缓存中读取数据。硬盘缓存则保存用户前几次上网时所调用的信息资料。

16.1.2 CAR 构件缓存机制原理

CAR 构件缓存机制是在 IE 缓存机制的思想基础上建立的。

CCM（CAR Cache Manager，CAR 高速缓存管理）是一套构件化的缓存管理机制，它主要是为了支持Elastos网络操作系统的构件自滚动运行。CMM划分成三个部分，在cachemgr.dll中实现为三个构件，分别为：Cache，Download，CCM。CCM 结构如图 16.1 所示。

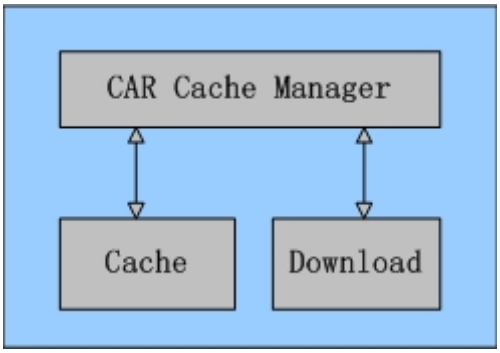


图 16.1 CCM 结构示意图

Cache 构件负责管理 Cache 目录及索引文件，是一个纯粹的 Cache 目录或者说是索引管理；Download 构件负责从网络下载文件，可以支持多种协议，但目前只实现了 HTTP 协议；CCM 作为策略层，通过调用 Cache 接口和 Download 接口，加上自身的策略来完成各种功能。在 Elastos2.0 上将 CCM 注册为“cachemgr”服务，主要向构件层（elastos@DDK）提供服务。当构件的 Loader 访问 CCM，询问指定的 uunm 在本地对应文件的路径时，CCM 会先从 Cache 中查找，如果本地已经存在，则将 Cache 返回的路径返回给 Loader；如果本地不存在，则

调用 Download 的接口到网上去下载文件，并保存到 Cache 目录当中。下载成功后，将 uunm 及文件全路径名添加到 Cache 的索引当中，下次访问时就不必重新下载了。

CCM 提供了参数。用户可以指定该参数，通过三种途径获取文件路径。这三种途径分别为：只从 Cache 获取，只从网络获取及先从 Cache 获取，没有命中再到网络去下载。一般情况下，我们认为 Cache 索引文件只包含 Cache 目录下的文件记录。同时，Cache 也提供了接口，可以将 Cache 目录之外的文件添加到 Cache 的索引当中，交由 Cache 来访问。

16.2 CAR 构件缓存机制的用法

CCM 通过在本地使用一个索引文件来管理文件。每个文件对应于索引文件中的一条记录。记录存储了文件的相关信息，包括网络地址 URL、本地地址（全路径名）。从网络上下载的文件保存在 Cache 目录。下面介绍 CCM 的使用方法:

1、打开 DDK 中的编译选项，在 D:\Elastos\src\libpub\inc\elasys\eladef.h 中有一行：

```
//#define CACHEMANAGER
```

将前面的注释符去掉，改为:

```
#define CACHEMANAGER
```

编译 DDK，再执行 dropsdk.bat。

2、设置配置文件（config）。

（1）在 mkpkg.cfg 中添加：

```
%ELASTOS_ROOT%\obj\%ELASTOS_BUILD_KIND%\bin\elasock.dll \
%ELASTOS_ROOT%\obj\%ELASTOS_BUILD_KIND%\bin\cachemgr.dll \
```

（2）在 service.cfg 中添加注册 Cache Manager 服务的设置（必须作为第一个服务启动）。

```
RegisterService("cachemgr.dll", "CCARCacheMgr", "ICARCacheMgr", "cachemgr");
```

（3）在 startup.cfg 中注册服务（默认已设置好）。

```
waitsvr.exe cachemgr bootmods filesys memfs diskmgr
```

（4）将网络和文件系统配置好（使用硬盘的情况下），例如：

```
[Disk3]
Name =E:
Description =FileSys
ServiceName =ktfs
DeviceName =device:idepartition0
……
[Usnet]
NetID =none
LocalIP =192.158.2.236
SubnetMask =255.255.248.0
DefaultGateway =192.158.0.1
NeedFtpServer =1
……
```

3、在 SDK 环境下编译 cachemgr 代码。目前 CCM 的 Cache 目录和索引文件都在程序中直接指定，而不采用配置文件的方式。这两个参数在 src/cachemgr/CCARCache.cpp 中指定，如下：

```
wchar_t * g_pIndexFile = L"/index.dat";
wchar_t * g_pCacheDir  = L"/";
```

其中 g_pIndexFile 指定了索引文件，g_pCacheDir 变量指定了 Cache 目录。虽然索引文件不要求一定要放在 Cache 目录中，但建议还是放在 Cache 目录为好。

4、执行 mkpkg.exe。

按照以前的习惯，所有目标程序都被拷贝到镜像目录下。现在，如果用户不打开 CCM 的开关（eladef.h），CCM 并没有改变这种习惯。但如果开关被打开，就必须在 SDK 环境下按照上文“CCM 用法”中介绍的流程进行配置，在系统启动时注册 CCM 服务，否则无法正确加载构件程序。默认情况下该开关是关闭状态。

因为一般 uunm 指定的网络地址不是开发机，而目标文件却一般都保存在开发机上，如果每次将文件拷贝到网络，会很不方便，因此建议在开发过程中不要打开 CCM 开关。当然，也可以调用 ICARCache 接口中的 Add 方法将自己的 dll 程序添加到索引中。

使用 CCM 带来了很多好处，不仅可以实现 CAR 构件的自滚动下载运行、点击下载运行等，还可以使用同名目标文件。在现有的开发环境中，所有目标程序文件被拷贝到镜像目录下，如 exe、dll 都放在镜像目录的 bin 目录中，这种情况下我们不能生成同名文件。但如果使用 CCM，我们就可以生成同名目标文件，只不过它们的路径不同，对于 CAR 构件程序来说，它们的 uunm 不同即可。

16.3 CAR 构件缓存机制的示例程序

下面通过一个范例来演示使用 CAR 构件缓存机制来实现 CAR 构件自滚动下载运行的方法。该范例程序位于 D:\Elastos20.SDK\src\internal_samples\ezcom\cachemgr\autoroll 目录中。

在这个例子中，实现了四个构件：CA、CB、CC、CD，分别在四个构件程序 cma.dll、cmb.dll、cmc.dll、cmd.dll 中实现，各构件分别实现了 IA、IB、IC、ID 接口，每个接口有个对应的方法，分别为 FA、FB、FC、FD。

客户程序调用了构件 CA 的 FA 方法；在构件 CA 中，FA 方法调用了构件 CB 的 FB 方法、构件 CC 的 FC 方法；在构件 CB 中，FB 方法调用了构件 CD 的 FD 方法。如图 16.2 所示。

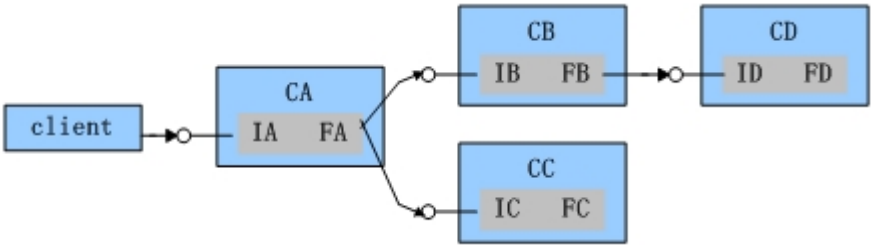


图 16.2 构件间滚动调用示意图

为方便观察演示结果，CCM 中添加了一些打印信息，用于输出构件的 uunm 信息(该构件在互联网上的地址，可以是 IP 地址也可以是域名，用户需要根据实际情况来重新指定)或对应的本地文件路径名。程序运行两次，显示结果如下：

```
[/ $]autoroll.exe
Download the file: 192.158.1.82/car/cma.dll
        CA::FA() is called.
CA: I'll call CB::FB().
Download the file: 192.158.1.82/car/cmb.dll
        CB::FB() is called.
CB: I'll call CD::FD().
Download the file: 192.158.1.82/car/cmd.dll
        CD::FD() is called.
CA: I'll call CC::FC().
Download the file: 192.158.1.82/car/cmc.dll
        CC::FC() is called.

[/ $]autoroll.exe
File is in the Cache: 192.158.1.82/car/cma.dll->/msdos/cache/1727.dll
        CA::FA() is called.
CA: I'll call CB::FB().
File is in the Cache: 192.158.1.82/car/cmb.dll->/msdos/cache/6a40.dll
        CB::FB() is called.
CB: I'll call CD::FD().
File is in the Cache: 192.158.1.82/car/cmd.dll->/msdos/cache/4654.dll
        CD::FD() is called.
CA: I'll call CC::FC().
File is in the Cache: 192.158.1.82/car/cmc.dll->/msdos/cache/54ef.dll
        CC::FC() is called.
```

第一次运行时，因为本地还没有对应的文件，需要从网络上下载，只输出了构件的 uunm 信息；第二次运行时因为本地 Cache 已经有对应文件，所以直接使用，不需从网络再次下载，输出了构件的 uunm 信息和对应的本地文件路径名。

第三篇　编程示例

第十七章　编程示例

17.1 嵌有 Lua 程序的 XML-Glue 程序示例

```
<?xml version="1.0" encoding="utf-8"?>
<x:xmlglue xmlns:x="http://www.elastos.com/xml-glue" xmlns:w="elactrl.dll">
    <w:form x:id="MainForm" nControlStyle="FormStyle_DoubleBuffer,
FormStyle_PixelAlphaChannel,
                ControlStyle_NoBackground" esCaption="sub" nLeft="20"
nTop="120" nWidth="100" nHeight="120">
        <w:pictureBox  x:id="picBox" nLeft="0" nTop="0" nWidth="100"
nHeight="120" nControlStyle="ControlStyle_NoBackground"/>
    </w:form>
    <script language="lua">

        function OnMouseDown(st, x, y, b)
            MainForm:SetWindowLevel(1)
        end

        function OnKeyDown(src, id, d)
            if id == 141 or id == 144 or id == 137 then
                MainForm:KillTimer(1)
                MainForm:Close()
            end
        end

        j = 0;
        images = {}
        elagdi = Elastos.Using("elagdi.dll")
        for j = 1, 17 do
            filename = "dancer" .. j .. ".png"
            image = elagdi.CImage()
            image:InitFromFile(resource(filename))
            images[j] = image
        end

        function OnTimer()
```

```
                local i = 1
                return function(src, id)
                    i = i % 17 + 1
                    picBox:SetImage(images[i])
                    src:Update()
                end
            end

            picBox:SetImage(images[1]);
            MainForm:SetStackingClass(2)
            MainForm:SetTimer(1, 80)

            MainForm.Timer = OnTimer()
            picBox.MouseDown = OnMouseDown
            MainForm.KeyDown = OnKeyDown
            MainForm.InActive = OnLostFocus;
            MainForm.Active = OnGotFocus;
            MainForm:Show()

        </script>
    </x:xglue>
```

17.2 JavaScript 程序示例

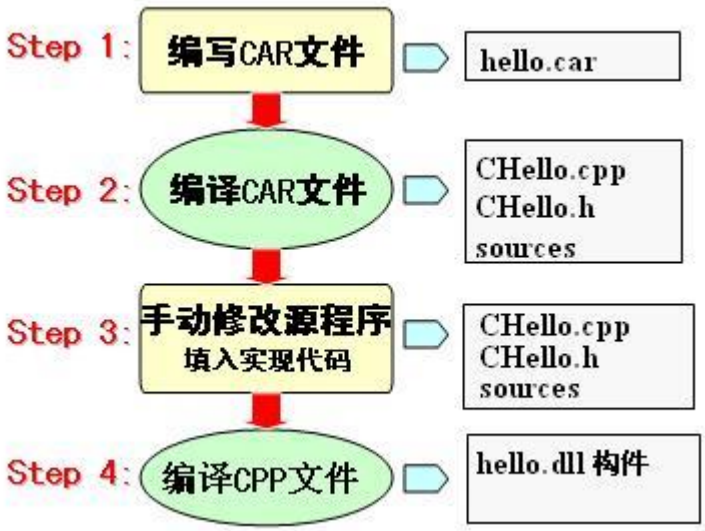
```
    function onButtonClick(src) {
        print(src.text + " clicked!");
    }

    elactrl = Elastos.Using("elactrl.dll");
    form = elactrl.Form.New(
               "JavaScriptDemo", 0, 0, 240, 320, 0); button = elactrl.Button.New(
                  "Hello", 90, 60, 60, 25, 0, form); button.Click = onButtonClick;
form.Show();
```

17.3 C/C++程序示例

```
    #include <stdio.h>       // support mostly standard .h files
    #import <foobar.dll>     // DLLs are used in design-time and run-time
```

```
    Boolean exitFlag = FALSE;

    ECode OnClick(IFoo sender) {
        exitFlag = TRUE;
        return NOERROR;
    }

    int main()
    {
        IBar* pBar;
        ECode ec = CFooBar::New("Hello", &pBar); // allow constructors.  btw,
CProcess::New(..), CThread::New(..) works as well.
        // ECode ec = CButton::NewInContext(pOuterSpace, "Hello", &pBar); // trying
to contact aliens on a machine far, far away...
        if (FAILED(ec)) { … }
        pBar->Bar();

        IFoo* pFoo;
        ec = IFoo::Query(pBar, &pFoo);  // does the object support IFoo?
        if (FAILED(ec)) { … }
        pFoo->Foo();

        CFooBar::AddClickCallback(pFoo, &OnClick);  // bind a callback handler
        while (exitFlag == FALSE) sleep(1);  // waiting for the callback handler
to be called...

        pBar->Release();  // still use the good ol' reference counting,
unfortunately.
        pFoo->Release();
        return 0;
    }
```

编译与运行一个 CAR 构件过程：



17.4 CAR 基础类库操作

17.4.1 StringTokenizer

```
//文件 StringTokenizerDemo.cpp

#include <elastos.h>
using namespace Elastos;

ECode ElastosMain(const ArrayOf<WString>& args)
{
    AStringBuf_<50> asb;
    asb << "AsciiString" << ' ' << L"WideString" << ' '
        << Boolean(TRUE) << ' ' << 150 << ' ' << 3.14;
    CConsole::WriteLine(asb);

    AStringTokenizer ast(asb);          // use default separators

    CConsole::WriteLine(ast.NextToken());
    CConsole::WriteLine(ast.NextToken());
```

```
    CConsole::WriteLine(ast.NextToken().ToBoolean());
    CConsole::WriteLine(ast.NextToken().ToInt32());
    CConsole::WriteLine(ast.NextToken().ToDouble());

    CConsole::WriteLine();

    WStringBuf_<50> wsb;
    wsb << "AsciiString" << L',' << L"WideString" << L','
        << Boolean(FALSE) << L',' << 155 << L',' << 3.1415;
    CConsole::WriteLine(wsb);

    WStringTokenizer wst(wsb, L",");    // use "," as the separator

#if 1
    // it's not the most beautiful code in the world because
    // it's error prone.  nevertheless, it proves a concept.
    //
    WStringBuf_<50> wsb1;
    WStringBuf_<50> wsb2;
    Boolean b;
    Int32 n;
    Double d;

    wst >> wsb1 >> wsb2 >> b >> n >> d;

    CConsole::WriteLine(wsb1);
    CConsole::WriteLine(wsb2);
    CConsole::WriteLine(b);
    CConsole::WriteLine(n);
    CConsole::WriteLine(d);
#else

    CConsole::WriteLine(wst.NextToken());
    CConsole::WriteLine(wst.NextToken());
    CConsole::WriteLine(wst.NextToken().ToBoolean());
    CConsole::WriteLine(wst.NextToken().ToInt32());
    CConsole::WriteLine(wst.NextToken().ToDouble());
#endif

    CProcess::Exit(0);
    return NOERROR;
```

```
}

//文件 sources
TARGET_NAME= StringTokenizerDemo
TARGET_TYPE= exe
SOURCES= \
    StringTokenizerDemo.cpp \
LIBRARIES = $(XDK_USER_LIB)\elacrt.lib     $(XDK_LIB_PATH)\elastos.lib \


//运行结果
AsciiString WideString True 150 +3.140000
AsciiString
WideString
True
150
+3.140000

AsciiString,WideString,False,155,+3.141500
AsciiString
WideString
False
155
+3.141500
```

17.4.2 Int32ArrayDemo

```
//文件 Int32ArrayDemo.cpp
//
// There are four ways to create a CarArray.
// It is also easy to hack a CarArray to achieve C/C++ like performance.
//
// A CarArray is, in fact, defined as following:
// struct CarQuintet {
//     CarQuintetFlags m_flags;
//     CarQuintetLocks m_locks;
```

```
//     MemorySize      m_used;
//     MemorySize      m_size;
//     PVoid           m_pBuf;
// };
// where m_pBuf is really just a pointer of C/C++ buffer.
//

#include <elastos.h>
using namespace Elastos;

void PrintThreeIntegers(const ArrayOf<Int32>& ar)
{
    WStringBuf_<20> wstrBuf;

    wstrBuf.Append(ar[0]); wstrBuf.Append(L", ");
    wstrBuf.Append(ar[1]); wstrBuf.Append(L", ");
    wstrBuf.Append(ar[2]);

    CConsole::WriteLine(wstrBuf);
}

ECode ElastosMain(const ArrayOf<WString>& args)
{
    // ==========
    // Create a CAR array on the stack.
    //
    ArrayOf_<Int32, 20> myArray;
    myArray[0] = 100;
    myArray[1] = 101;
    myArray[2] = 102;
    PrintThreeIntegers(myArray);

    // ==========
    // Create a CAR array on the heap.
    // User has to remember to free the array.
    //
    ArrayOf<Int32>* pMyArray = ArrayOf<Int32>::Alloc(20);
    if (pMyArray == NULL) {
        return E_OUT_OF_MEMORY;
    }
    (*pMyArray)[0] = 200;
```

```
    (*pMyArray)[1] = 201;
    (*pMyArray)[2] = 202;
    PrintThreeIntegers(*pMyArray);
    ArrayOf<Int32>::Free(pMyArray);

    // ==========
    // Create a CAR array on the stack if it's small enough; Otherwise,
    // create it on the heap.  User has to remember to free the array.
    //
    ArrayOf<Int32>* pAutoArray = AUTO_ARRAYOF(Int32, 20);
    if (pAutoArray == NULL) {
        return E_OUT_OF_MEMORY;
    }
    (*pAutoArray)[0] = 300;
    (*pAutoArray)[1] = 301;
    (*pAutoArray)[2] = 302;
    PrintThreeIntegers(*pAutoArray);
    ArrayOf<Int32>::Free(pAutoArray);

    // ==========
    // Create a traditional C array on the stack to achieve better
    // performance, then  package it into a CAR array.
    //
    Int32 cArray[20];
    cArray[0] = 400;
    cArray[1] = 401;
    cArray[2] = 402;

    ArrayOf<Int32> myBoxArray(cArray, 20);
    PrintThreeIntegers(myBoxArray);

    // ==========
    // Create a CAR array on the stack.
    // Then hack its gut out to achieve better performance.
    //
    ArrayOf_<Int32, 20> myWrapArray;
    Int32* p = myWrapArray.GetPayload();
    p[0] = 500;
    p[1] = 501;
    p[2] = 502;
    PrintThreeIntegers(myWrapArray);
```

```
    CProcess::Exit(0);
    return NOERROR;
}

//文件 sources
TARGET_NAME= Int32ArrayDemo
TARGET_TYPE= exe
SOURCES= \
    Int32ArrayDemo.cpp \
LIBRARIES = $(XDK_LIB_PATH)\elastos.lib \

//运行结果
100, 101, 102
200, 201, 202
300, 301, 302
400, 401, 402
500, 501, 502
```

17.4.3 WStringBufDemo

```
//文件 WStringBufDemo.cpp
//
// The WStringBuf methods demonstrated in this file are safer and faster than
// equivalent lib-C functions, which are listed in comments for reference.
//

// #include <stdio.h>
// #include <stdlib.h>
// #include <string.h>
//
#include <elastos.h>
using namespace Elastos;

// int wmain(int argc, wchar_t** argv)
//
ECode ElastosMain(const ArrayOf<WString>& args)
```

```
{
    // wchar_t wstrBuf[20];
    //
    WStringBuf_<20> wstrBuf;

    // wcscpy(wstrBuf, L"Great");
    // _putws(wstrBuf);
    //
    wstrBuf.Copy(L"Great");
    CConsole::WriteLine(wstrBuf);

    // wcscat(wstrBuf, L" Wall");
    // _putws(wstrBuf);
    //
    WString wstr = L" Wall";
    wstrBuf.Append(wstr);
    CConsole::WriteLine(wstrBuf);

    // wcscpy(wstrBuf, L"Great");
    // wcscat(wstrBuf, L" Wall");
    // wcscat(wstrBuf, L" of");
    // wcscat(wstrBuf, L" China");
    // _putws(wstrBuf);
    //
    wstrBuf.SetEmpty();
    wstrBuf.Concatenate(L"Great", L" Wall", L" of", L" China", NULL);
    CConsole::WriteLine(wstrBuf);

    // Int32 len = wcslen(wstrBuf);
    // _itow(len, wstrBuf, 10);
    // _putws(wstrBuf);
    //
    Int32 len = wstrBuf.GetLength();
    CConsole::WriteLine(len);

    // swprintf(wstrBuf, L"%d", 500);
    // _putws(wstrBuf);
    //
    wstrBuf.SetEmpty();
    wstrBuf.Append(500);
    CConsole::WriteLine(wstrBuf);
```

```
    // swprintf(wstrBuf, L"%x", 500);
    // _putws(wstrBuf);
    //
    wstrBuf.SetEmpty();
    wstrBuf.Append(500, NumberFormat_Hex);
    CConsole::WriteLine(wstrBuf);

    // swprintf(wstrBuf, L"%10X", 500);
    // _putws(wstrBuf);
    //
    wstrBuf.SetEmpty();
    wstrBuf.Append(500, MakeNumberFormat(NumberFormat_BigHex, 10));
    CConsole::WriteLine(wstrBuf);

    // swprintf(wstrBuf, L"%10.5d", 500);
    // _putws(wstrBuf);
    //
    wstrBuf.SetEmpty();
    wstrBuf.Append(500, MakeNumberFormat(NumberFormat_Decimal, 10, 5));
    CConsole::WriteLine(wstrBuf);

    CProcess::Exit(0);
    return NOERROR;
}

//文件 sources
TARGET_NAME= WStringBufDemo
TARGET_TYPE= exe
SOURCES= \
    WStringBufDemo.cpp \
LIBRARIES = $(XDK_LIB_PATH)\elastos.lib

//运行结果
Great
Great Wall
Great Wall of China
19
500
1f4
       1F4
```

```
00500
```

17.4.4 ByteArrayDemo

```
//文件 ByteArrayDemo.cpp
//
#include <elastos.h>
using namespace Elastos;

void PrintByteArray(const ArrayOf<Byte>& buf)
{
    AStringBuf_<200> astrBuf;

    int used = buf.GetUsed();
    for (int i = 0; i < used; i++) {
        astrBuf.Append((AChar)buf[i]);
    }

    CConsole::WriteLine(astrBuf);
}

ECode ElastosMain(const ArrayOf<WString>& args)
{
    ArrayOf_<Byte, 200> myArray;
    Byte a[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };

    myArray.Copy(a, 7);
    PrintByteArray(myArray);

    myArray.Append((Byte *)"HIJK", 4);
    PrintByteArray(myArray);

    myArray.Replace(3, (Byte *)"XYZ", 3);
    PrintByteArray(myArray);

    CProcess::Exit(0);
```

```
    return NOERROR;
}

//文件 sources
TARGET_NAME= ByteArrayDemo
TARGET_TYPE= exe
SOURCES= \
    ByteArrayDemo.cpp \
LIBRARIES = $(XDK_LIB_PATH)\elastos.lib

//运行结果
ABCDEFG
ABCDEFGHIJK
ABCXYZGHIJK
```

第四篇　编程参考

第十八章　编程参考

18.1 CAR 语法规范

18.1.1 CAR 文件与调用 CAR 构件的 C++文件的对应关系

CAR 文件的源代码：

```
module car.elastos.com/native/foobar.dll
{
    interface IFoo[2] {
        Foo(Int32 x);
    }

    interface IBar[3] {
        Bar(Float y, WString ws);
    }

    interface IFooEvent {
        FooEvent[4]();
    }

    class CFooBar[1] {
        interface IFoo[2];
        interface IBar[3];
        callback interface IFooEvent;
    }
}
```

使用 CAR 构件的 C++文件源代码：

```
#include <stdio.h>
#import "foobar.dll"

ECode MyFooCallback(IFoo *pSender) {
    puts("MyFooCallback");
```

```
        return NOERROR;
    }

    int main() {
    ECode ec;
    Int32 x;
    Float y;
    WString ws;
    IBar[3] * pBar;
    IFoo[2] * pFoo;

        ec = CFooBar[1]::New(&pBar);
        pBar->Bar(y, ws);

        ec = CFooBar::AddFooEventCallback[4](pBar, &MyFooCallback);

        ec = IFoo::Query(pBar[5], &pFoo[6]);
        pFoo->Foo(Int32 x);

        pBar->Release(); pFoo->Release();
        return 0;
    }
```

1. C++文件中的 CFooBar 来自于 CAR 文件中定义的构件类 CFooBar；
2. C++文件中的接口指针 pFoo 来自于 CAR 文件中定义的构件接口 IFoo；
3. C++文件中的接口指针 pBar 来自于 CAR 文件中定义的构件接口 IBar；
4. C++文件中的 AddFooEventCallback 来自于 CAR 文件中定义的构件回调接口 IFooEvent 的回调方法 FooEvent，在此也可以看出添加回调事件的函数名构成原则，即在回调方法的前后分别加入前缀和后缀，形成 AddXXXCallback 的名称。
5. Query 方法中的 pBar 是指需要查询的构件的任一一个指针；
6. Query 方法中的&pFoo 是指存放获得的接口指针的地址。

注意：CAR 文件中的 IFoo 和 IBar 接口中的方法不能同名。

18.1.2 CAR 文件与编译 CAR 文件自动生成 C++文件的对应关系

CAR 文件的源代码：

```
module car.elastos.com/native/foobar.dll
{
    interface IFoo {
```

```
        Foo[2](Int32 x);
    }

    interface IBar {
        Bar[3](Float y, WString ws);
    }

    interface IFooEvent {
        FooEvent();
    }

    class CFooBar[1] {
        interface IFoo;
        interface IBar;
        callback interface IFooEvent;
    }
}
```

生成的名为 CFooBar.cpp 的 C++文件：

```
#include "CFooBar.h"
#include "_CFooBar.cpp"

ECode CFooBar[1]::Foo[2](/* [in] */ Int32 x)
{
    // TODO: Add your code here[4]
    return E_NOT_IMPLEMENTED;[5]
}

ECode CFooBar[1]::Bar[3](/* [in] */ Float y,
                         /* [in] */ WString ws)
{
    // TODO: Add your code here[4]
    return E_NOT_IMPLEMENTED;[5]
}
```

生成的名为 CFooBar.h 的头文件：

```
#ifndef __CFOOBAR_H__
```

```
#define __CFOOBAR_H__
#include "_CFooBar.h"

CarClass(CFooBar)[1]
{
public:
    CARAPI Foo[2](/* [in] */ Int32 x);

    CARAPI Bar[3](/* [in] */ Float y,
                  /* [in] */ WString ws);

private:
    //TODO:Add your private member variables here.[6]
};

#endif // __CFOOBAR_H__
```

1. CAR 文件中的 CFooBar 在头文件中声明为构件类 CFooBar；
2. CAR 文件中的接口方法 Foo 在 C++文件中被定义为构件类 CFooBar 的方法 Foo；
3. CAR 文件中的接口方法 Bar 在 C++文件中被定义为构件类 CFooBar 的方法 Bar；
4. 在 C++文件中添加 Foo 和 Bar 方法的具体实现；
5. 在默认情况下，方法返回 E_NOT_IMPLEMENTED，方法未实现；
6. 在头文件中可以添加构件类的私有方法，并在 C++文件中添加其的具体实现。

18.1.3 XML-Glue 文件和 CAR 文件的对应关系

XML-Glue 文件（文件名：hello.xml）的源代码：

```
<?xml version="1.0" encoding="utf-8"?>
<x:xmlglue xmlns:x="http://car.elastos.com/xml-glue"
           xmlns:w="http://car.elastos.com/native/elactrl.dll"[1]">
    <script>
        function onButtonClick() {
            print("Button Clicked!");
        }
    </script>

    <w:form[2] caption[4]="" left[4]="0" top[4]="0" width[4]="240 " height[4]="320">
        <w:button[3] caption[4]="Hello World!"
```

```
                        left[4]="70" top[4]="60" width[4]="100" height[4]="25"
click[5]=onButtonClick() />
    </w:form>
</x:xmlglue>
```

elactrl.car 文件的部分源代码：

```
module car.elastos.com/native/elactrl.dll[1]
{
    interface IControl;
    interface IForm[2];
    interface IForm[2] : IControl
    interface IButton[3] : IControl {}
class CButton {
        interface IButton;
        callback interface IButtonEvent[5];
        callback interface IControlEvent;
    }
interface IControl
{
    Init(
            [in] Int32   controlStyle,
            [in] WString caption[4],
            [in] Int32 left[4],
            [in] Int32 top[4],
            [in] Int32 width[4],
            [in] Int32 height[4],
            [in] IControl *parent);
}
interface IButtonEvent {
        Click()[5];
    }
```

1. XML-Glue 文件需要首先定义 xml 命名空间，将 elactrl 图形构件定义名为 w 的命名空间，与 CAR 构件自身声明基本一致；

2. 在 elactrl 这个控件中可以调用的接口方法，在 XML-Glue 中利用标签的形式表现出来，<w:form>就是 IForm 接口，而其后的属性就是对应的初始化方法的各个参数；
3. <w:button>标签与<w:form>标签类似，表示调用了 IButton 接口；
4. 这些标签的属性实际上就是接口的初始化方法的各个参数；
5. click 表示注册回调函数，其在 CAR 里就是回调接口 IButtonEvent 的 Click 方法。

18.1.4 JavaScript 文件与 CAR 文件的对应关系

JavaScript 文件的源代码：

```
function onButtonClick(src) {
    print(src.text + " clicked!");
}

elactrl = Elastos.Using("elactrl.dll")[1];
form[2] = elactrl.Form[2].CreateObject(
            "JavaScriptDemo", 0, 0, 240, 320, 0);
button[3] = elactrl.Button[3].CreateObject(
                "Hello", 90, 60, 60, 25, 0, form);
button.Click[4] = onButtonClick;
form.Show()[5];
```

调用的 CAR 构件还是 elactrl.dll，具体可参照上文。

1. 在 JavaScript 中使用 CAR 构件的方法是将 elactrl.dll 构件定义为一个 JavaScript 对象；
2. 使用 IForm 接口创建 form 对象；
3. 使用 IButton 接口创建 button 对象；
4. 注册 button 的事件回调函数；
5. 显示 form 控件。

18.1.5 Lua 文件与 CAR 文件的对应关系

Lua 文件的源代码：

```
function OnButtonClick(src)
    form:Close();
end
```

```
elactrl = Elastos.Using("elactrl.dll")[1]
Form = elactrl.CForm[2]
form[2] = Form()
form:Init(0, "LUA", 10, 10, 180, 240, 0)
Button = elactrl.CButton[3]
button[3] = Button()
button:Init(0, "hello", 10, 10, 60, 25, form)
button.Click[4] = OnButtonClick;

form:Show()[5]
elagdi = Elastos.Using("elagdi.dll")
app = elagdi.AGrafixAppletAspect()
app:Run()
```

调用的 CAR 构件还是 elactrl.dll，具体可参照上文。

1. 在 Lua 中使用 CAR 构件的方法是将 elactrl.dll 定义为一个 Lua 对象；
2. 使用 IForm 接口创建 form 对象；
3. 使用 IButton 接口创建 button 对象；
4. 注册 button 的事件回调函数；
5. 显示 form 控件。

18.2 CAR 编译提示信息

```
    static ErrorMessage s_errorMessages[] = {
        { CAR_W_LocalResult, "Return type of method \"%s\" is not ECode." },
        { CAR_W_LocalArg, "Parameter \"%s\" is local type." },
        { CAR_W_NoMethods, "No methods defined in interface." },
        { CAR_W_LocalParent, "Inherited from local interface \"%s\"." },
        { CAR_W_NoClassInterfaces, "Class has no interface included." },
        { CAR_W_LoadLibrary, "Library \"%s\" count not be loaded." },
        { CAR_W_TooManyLibraries, "Too many libraries imported." },
        { CAR_W_LocalClass, "All interfaces are local in class, set as local." },
        { CAR_W_DupMethodName, "Interface method name \"%s\" is duplicated." },
        { CAR_W_UnexpectFileType, "File \"%s\" has an unexpected type, ignored." },
```

```
        { CAR_W_NoAutoParamAttrib, "Attributes of parameter \"%s\" can "
                "not be resolved automatically." },
        { CAR_W_IllegalMemberName, "Illegal member Name \"%s\"." },
        { CAR_W_IllegalCharacterInURL, "Illegal character \'%c\' in URL \"%s\"." },

        { CAR_E_UnexpectEOF, "Unexpected end of file." },
        { CAR_E_UnexpectSymbol, "Unexpected symbol \"%s\"." },
        { CAR_E_UnexpectChar, "Unexpected character '%c'." },
        { CAR_E_SymbolTooLong, "Symbol is too Int64." },
        { CAR_E_IllegalChar, "Illegal character '%c'." },
        { CAR_E_LibraryProject, "The attribute \"project\" " \
            "can't use with \"library\"" },
        { CAR_E_UunmUndef, "No UUNM string specified." },
        { CAR_E_CARUuidUndef, "No UUNM specified for component." },
        { CAR_E_LoadLibrary, "Library \"%s\" could not be loaded." },
        { CAR_E_ExpectSymbol, "Symbol \"%s\" may be missing." },
        { CAR_E_UuidFormat, "Illegal uuid format." },
        { CAR_E_OutOfMemory, "Compilation out of memory." },
        { CAR_E_AttribConflict, "Attributes conflict: %s with %s." },
        { CAR_E_MscomNoUuid, "The uuid is needed when \"mscom\" specified." },
        { CAR_E_IllegalValue, "Illegal number value." },
        { CAR_E_DupEntry, "%s \"%s\" redefined." },
        { CAR_E_FullEntry, "Too many %s defined." },
        { CAR_E_NameConflict, "Symbol(%s \"%s\") has been defined." },
        { CAR_E_NotFound, "Undefined %s \"%s\"." },
        { CAR_E_UndefinedSymbol, "Undefined symbol \"%s\"." },
        { CAR_E_UuidNoMscom, "The uuid specified but \"mscom\" is not declared." },
        { CAR_E_RedefUuid, "The uuid redefined." },
        { CAR_E_InterfaceAttrib, \
            "Above attributes can be used before interface body only." },
        { CAR_E_ClassAttrib, \
            "Above attributes can be used before class body only." },
        { CAR_E_ExpectInterfaceName, "Interface name expected." },
        { CAR_E_ExpectClassName, "Class name expected." },
        { CAR_E_ExpectStructName, "Struct name expected." },
        { CAR_E_ExpectEnumName, "Enum name expected." },
        { CAR_E_UndefType, "Undefined type \"%s\"." },
        { CAR_E_ExpectMethodName, "Method name expected." },
        { CAR_E_ExpectParamName, "Parameter name expected." },
        { CAR_E_TypeConflict, "Type conflicted." },
        { CAR_E_AspectUse, "aspect can only used with context." },
```

```
        { CAR_E_ContextNoAspect, "No aspects declared for context." },
        { CAR_E_NotAspect, "\"%s\" is not an aspect." },
        { CAR_E_TooManyParents, "Too many parents defined." },
        { CAR_E_NestedType, "Type nested with EzArray or EzEnum." },
        { CAR_E_VoidArg, "Arg \"%s\" has an illegal type \"void\"." },
        { CAR_E_VoidStructElem, "\"%s\" has an illegal type \"void\"." },
        { CAR_E_AsyncOut, "Attribute \"out\" not "
            "permitted in async interface." },
        { CAR_E_NoMethods, "No methods defined in interface." },
        { CAR_E_DupUuid, "The uuid is a duplicate of previous definition." },
        { CAR_E_RedefMain, "Attribute 'main' appear more than once." },
        { CAR_E_NoMainClass, "No main class specified for component." },
        { CAR_E_NoClassInterfaces, "Class has no interface included." },
        { CAR_E_LoadSystemLib, "Can't load system types library." },
        { CAR_E_NestedStruct, "Struct has a nested member \"%s\"." },
        { CAR_E_MergeCLS, "Error when merge library \"%s\"." },
        { CAR_E_GenDisp, "Error on generate dispatch interface of class." },
        { CAR_E_DupMethodName, "Method name %s is duplicated in class." },
        { CAR_E_InheritNoVirtual, "Inherit from none-virtual interface class." },
        { CAR_E_NoClasses, "No class defined in component." },
        { CAR_E_OpenFile, "Can't open file \"%s\"." },
        { CAR_E_IllegalOut, "\"%s\" is not a valid [out] parameter." },
        { CAR_E_NestedInherit, "%s \"%s\" is nested inheriting." },
        { CAR_E_OutParameterInCtor, "Parameter can't be out in constructor!" },
        { CAR_E_ParameterInSingletonCtor, "Constructor of singleton class can't
have any parameter!" },
        { CAR_E_InvalidMemberName, "Invalid member name \"%s\"." },
        { CAR_E_IllegalClassName, "Illegal class name. First charactor should be
'C'."},
        { CAR_E_IllegalAspectName, "Illegal aspect name. First charactor should be
'A'."},
        { CAR_E_IllegalContextName, "Illegal context name. First charactor should
be 'K'."},
        { CAR_E_IllegalDomainName, "Illegal domain name. First charactor should be
'D'."},
        { CAR_E_IllegalGenericName, "Illegal generic name. First charactor should
be 'G'."},
        { CAR_E_IllegalInterfaceName, "Illegal interface name. First charactor
should be 'I'."},
        { CAR_E_IlleagalSizeType, "Illegal n type of xxxArray_<n> or xxxBuf_<n>."},
        { CAR_E_ExpectConstName, "Const name expected." },
```

```
};
```

18.3 常用宏定义使用规范

基本数据类型：

```
    typedef signed char Int8;
    typedef unsigned char UInt8;
    typedef UInt8 Byte;
    typedef char AChar;
    typedef unsigned short WChar;
    typedef signed short Int16;
    typedef unsigned short UInt16;
    typedef int Int32;
    typedef unsigned int UInt32;
    typedef __int64 Int64;
    typedef __uint64 UInt64;
    typedef float Float;
    typedef double Double;
    typedef unsigned char Boolean;
    typedef Int32 ECode;

    typedef AChar *PAChar;
    typedef WChar *PWChar;
    typedef Int8 *PInt8;
    typedef Byte *PByte;
    typedef UInt8 *PUInt8;
    typedef Int16 *PInt16;
    typedef UInt16 *PUInt16;
    typedef Int32 *PInt32;
    typedef UInt32 *PUInt32;
    typedef Int64 *PInt64;
    typedef UInt64 *PUInt64;
    typedef Float *PFloat;
    typedef Double *PDouble;
    typedef Boolean *PBoolean;
```

基础数据结构：

```
    typedef struct DECL_PACKED _tagGUID {
        UINT32 Data1;
        UINT16 Data2;
        UINT16 Data3;
```

```
    UINT8 Data4[8];
}   GUID;

typedef GUID *PGUID;
typedef GUID InterfaceId;
typedef GUID *PIID;
typedef GUID CLSID;
typedef GUID *PCLSID;
typedef GUID CATID;
typedef GUID *PCATID;

typedef struct CLASSID {
    CLSID clsid;
    WCHAR *pUunm;
}   CLASSID, *PCLASSID;

extern const GUID GUID_NULL;
#define InterfaceId_NULL   GUID_NULL
#define CLSID_NULL GUID_NULL

#if defined(__cplusplus)
#define REFGUID  const GUID &
#define RIID   const InterfaceId &
#define REFCLSID const CLSID &
#define REFCATID const CATID &
#define RCLASSID const CLASSID &
#else // !__cplusplus
#define REFGUID  const GUID * const
#define RIID   const InterfaceId * const
#define REFCLSID const CLSID * const
#define REFCATID const CATID * const
#define RCLASSID const CLASSID * const
#endif // !__cplusplus
```

调用类型定义：

```
#define EZAPICALLTYPE       CDECL
#define CARAPICALLTYPE      STDCALL

#define EZAPI               EXTERN_C ECode EZAPICALLTYPE
#define EZAPI_(type)        EXTERN_C type EZAPICALLTYPE
```

```
    #define CARAPI              ECode CARAPICALLTYPE
    #define CARAPI_(type)       type CARAPICALLTYPE
```

返回值定义：

```
    ECode layout:
    3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
    1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
    +-+-+-+-+-+-+-+-+--------------+------------------------------+
    |S| family code |   info code   |         debug code           |
    +-+-+-+-+-+-+-+-+--------------+------------------------------+

    #define KERNEL_ERROR(c)          MAKE_ECode(SEVERITY_ERROR, FAMILY_KERNEL, c)
    #define DRIVER_ERROR(c)          MAKE_ECode(SEVERITY_ERROR, FAMILY_DRIVER, c)
    #define FILESYS_ERROR(c)         MAKE_ECode(SEVERITY_ERROR, FAMILY_FILESYS,
c)
    #define TCPIP_ERROR(c)           MAKE_ECode(SEVERITY_ERROR, FAMILY_TCPIP, c)
    #define DRM_ERROR(c)             MAKE_ECode(SEVERITY_ERROR, FAMILY_DRM, c)
    #define DATABASE_ERROR(c)        MAKE_ECode(SEVERITY_ERROR, FAMILY_DATABASE,
c)
    #define XML_ERROR(c)             MAKE_ECode(SEVERITY_ERROR, FAMILY_XML, c)
    #define XMLGLUE_ERROR(c)         MAKE_ECode(SEVERITY_ERROR, FAMILY_XMLGLUE,
c)

    #define FAMILY_MOBILE_ERROR(c)       MAKE_ECode(SEVERITY_ERROR, FAMILY_MOBILE,
c)
    #define FAMILY_GRAPHIC_ERROR(c)     MAKE_ECode(SEVERITY_ERROR, FAMILY_GRAPHIC,
c)
    #define CAR_ERROR(c)                 MAKE_ECode(SEVERITY_ERROR, FAMILY_CAR,
c)

    #define KERNEL_SUCCESS(c)        MAKE_SUCCESS(FAMILY_KERNEL, c)
    #define DRIVER_SUCCESS(c)        MAKE_SUCCESS(FAMILY_DRIVER, c)
    #define FILESYS_SUCCESS(c)       MAKE_SUCCESS(FAMILY_FILESYS, c)
    #define TCPIP_SUCCESS(c)         MAKE_SUCCESS(FAMILY_TCPIP, c)

    Elastos kernel error codes (Family: 0x0)
    #define E_PROCESS_NOT_ACTIVE         KERNEL_ERROR(0x01) // 0x81010000
    #define E_PROCESS_STILL_ACTIVE       KERNEL_ERROR(0x02) // 0x81020000
    #define E_PROCESS_NOT_STARTED        KERNEL_ERROR(0x03) // 0x81030000
    #define E_PROCESS_ALREADY_STARTED    KERNEL_ERROR(0x04) // 0x81040000
```

```
#define E_PROCESS_ALREADY_EXITED    KERNEL_ERROR(0x05) // 0x81050000
#define E_PROCESS_NOT_EXITED        KERNEL_ERROR(0x06) // 0x81060000
#define E_THREAD_NOT_ACTIVE         KERNEL_ERROR(0x07) // 0x81070000
#define E_THREAD_STILL_ACTIVE       KERNEL_ERROR(0x08) // 0x81080000
#define E_THREAD_UNSTARTED          KERNEL_ERROR(0X09) // 0x81090000
#define E_THREAD_ALREADY_FINISHED   KERNEL_ERROR(0x0A) // 0x810A0000
#define E_THREAD_NOT_STOPPED        KERNEL_ERROR(0x0B) // 0x810B0000
#define E_DOES_NOT_EXIST            KERNEL_ERROR(0x0C) // 0x810C0000
#define E_ALREADY_EXIST             KERNEL_ERROR(0x0D) // 0x810D0000
#define E_INVALID_OPTIONS           KERNEL_ERROR(0x0E) // 0x810E0000
#define E_INVALID_OPERATION         KERNEL_ERROR(0x0F) // 0x810F0000
#define E_TIMED_OUT                 KERNEL_ERROR(0x10) // 0x81100000
#define E_INTERRUPTED               KERNEL_ERROR(0x11) // 0x81110000
#define E_NOT_OWNER                 KERNEL_ERROR(0x12) // 0x81120000
#define E_ALREADY_LOCKED            KERNEL_ERROR(0x13) // 0x81130000
#define E_INVALID_LOCK              KERNEL_ERROR(0x14) // 0x81140000
#define E_NOT_READER                KERNEL_ERROR(0x15) // 0x81150000
#define E_NOT_WRITER                KERNEL_ERROR(0x16) // 0x81160000
#define E_NOT_ENOUGH_ADDRESS_SPACE  KERNEL_ERROR(0x17) // 0x81170000
#define E_BAD_FILE_FORMAT           KERNEL_ERROR(0x18) // 0x81180000
#define E_BAD_EXE_FORMAT            KERNEL_ERROR(0x19) // 0x81190000
#define E_BAD_DLL_FORMAT            KERNEL_ERROR(0x1A) // 0x811A0000
#define E_PATH_TOO_LONG             KERNEL_ERROR(0x1B) // 0x811B0000
#define E_PATH_NOT_FOUND            KERNEL_ERROR(0x1C) // 0x811C0000
#define E_FILE_NOT_FOUND            KERNEL_ERROR(0x1D) // 0x811D0000
#define E_NOT_SUPPORTED             KERNEL_ERROR(0x1E) // 0x811E0000
#define E_IO                        KERNEL_ERROR(0x1F) // 0x811F0000
#define E_BUFFER_TOO_SMALL          KERNEL_ERROR(0x20) // 0x81200000
#define E_THREAD_ABORTED            KERNEL_ERROR(0x21) // 0x81210000
#define E_SERVICE_NAME_TOO_LONG     KERNEL_ERROR(0x22) // 0x81220000
#define E_READER_LOCKS_TOO_MANY     KERNEL_ERROR(0x23) // 0x81230000
#define E_ACCESS_DENIED             KERNEL_ERROR(0x24) // 0x81240000
#define E_OUT_OF_MEMORY             KERNEL_ERROR(0x25) // 0x81250000
#define E_INVALID_ARGUMENT          KERNEL_ERROR(0x26) // 0x81260000

#define S_TIMED_OUT                 KERNEL_SUCCESS(0x01)    // 0x00010000
#define S_INTERRUPTED               KERNEL_SUCCESS(0x02)    // 0x00020000
#define S_NOT_EXIST                 KERNEL_SUCCESS(0x03)    // 0x00030000
#define S_ALREADY_EXISTS            KERNEL_SUCCESS(0x04)    // 0x00040000
#define S_BUFFER_TOO_SMALL          KERNEL_SUCCESS(0x05)    // 0x00050000
```

```
Driver error codes (Family: 0x1)
Error codes 0x001 - 0x04f are reserved for system
#define E_DEVICE_NAME_TOO_LONG              DRIVER_ERROR(0x01)
#define E_DEVICE_EXISTS                     DRIVER_ERROR(0x02)
#define E_DEVICE_NOT_FOUND                  DRIVER_ERROR(0x03)
#define E_DRIVER_BUSY                       DRIVER_ERROR(0x04)
#define E_DMAC_NOT_FOUND                    DRIVER_ERROR(0x05)
#define ELADRV_S_DRIVER_NOT_FOUND           DRIVER_SUCCESS(0x01)
#define ELADRV_S_CREATE_DRIVER_FAILED       DRIVER_SUCCESS(0x02)
#define ELADRV_S_REGISTER_DRIVER_FAILED     DRIVER_SUCCESS(0x03)
#define ELADRV_S_UNREGISTER_DRIVER_FAILED   DRIVER_SUCCESS(0x04)

Macros and constants for FACILITY_CAR error codes
#define E_NO_CLASS_INFO                     CAR_ERROR(0X01)
#define E_NO_EXPORT_SERVER                  CAR_ERROR(0X02)
#define E_NO_IMPORT_SERVER                  CAR_ERROR(0X03)
#define E_MARSHAL_DATA_TRANSPORT_ERROR      CAR_ERROR(0X04)
#define E_ERROR_STRING                      CAR_ERROR(0x10)
#define E_ERROR_STRING_A                    CAR_ERROR(0x11)
#define E_ERROR_URL                         CAR_ERROR(0x12)
#define CONNECT_E_NOCONNECTION              CAR_ERROR(0x13)
#define CONNECT_E_ADVISELIMIT               CAR_ERROR(0x14)
#define CONNECT_E_CANNOTCONNECT             CAR_ERROR(0x15)
#define CONNECT_E_OVERRIDDEN                CAR_ERROR(0x16)

DRM error codes (Family: 0x5)
#define E_DRM_FILE_PARSE                    DRM_ERROR(0X01)
#define E_DRM_NOT_OPEN_RODB                 DRM_ERROR(0X02)
#define E_DRM_NORO                          DRM_ERROR(0X03)
#define E_DRM_FILE_OP                       DRM_ERROR(0X04)
#define E_DRM_NO_COUNT                      DRM_ERROR(0X05)
#define E_DRM_OUT_OF_DATE                   DRM_ERROR(0X06)
#define E_DRM_MEDIA_OBJECT_EXIST            DRM_ERROR(0X07)
#define E_DRM_RO_EXIST                      DRM_ERROR(0X08)
#define E_DRM_PERMISSION_DISABLE            DRM_ERROR(0X09)
#define E_DRM_INVALID_RO                    DRM_ERROR(0X0A)
#define E_DRM_DB_KEY_NOT_EXIST              DRM_ERROR(0X0B)
#define E_DRM_INVALID_XML_FORMAT            DRM_ERROR(0X0C)
#define E_DRM_TIME_UNAVOIDABLE              DRM_ERROR(0X0D)

Database error codes (Family: 0xD)
```

```
#define E_DB_SQL_ERROR                  DATABASE_ERROR(0x01)
#define E_DB_INTERNAL_ERROR             DATABASE_ERROR(0x02)
#define E_DB_PERMISSION_DENIED          DATABASE_ERROR(0x03)
#define E_DB_REQUESTED_ABORT            DATABASE_ERROR(0x04)
#define E_DB_FILE_BUSY                  DATABASE_ERROR(0x05)
#define E_DB_TABLE_LOCKED               DATABASE_ERROR(0x06)
#define E_DB_OUT_OF_MEMORY              DATABASE_ERROR(0x07)
#define E_DB_READ_ONLY_DATABASE         DATABASE_ERROR(0x08)
#define E_DB_OPERATION_INTERRUPTED      DATABASE_ERROR(0x09)
#define E_DB_DISK_IO_ERROR              DATABASE_ERROR(0x0A)
#define E_DB_FILE_CORRUPTED             DATABASE_ERROR(0x0B)
#define E_DB_TABLE_NOT_FOUND            DATABASE_ERROR(0x0C)
#define E_DB_DATABASE_FULL              DATABASE_ERROR(0x0D)
#define E_DB_CANT_OPEN_DATABASE         DATABASE_ERROR(0x0E)
#define E_DB_LOCK_PROTOCOL_ERROR        DATABASE_ERROR(0x0F)
#define E_DB_DATABSE_EMPTY              DATABASE_ERROR(0x10)
#define E_DB_SCHEMA_CHANGED             DATABASE_ERROR(0x11)
#define E_DB_DATA_TOO_BIG               DATABASE_ERROR(0x12)
#define E_DB_CONSTRAINT_VIOLATION       DATABASE_ERROR(0x13)
#define E_DB_DATA_TYPE_MISMATCH         DATABASE_ERROR(0x14)
#define E_DB_LIBRARY_MISUSE             DATABASE_ERROR(0x15)
#define E_DB_NO_OS_SUPPORT              DATABASE_ERROR(0x16)
#define E_DB_AUTHORIZATION_DENIED       DATABASE_ERROR(0x17)
#define E_DB_AUXILIARY_DB_FORMAT_ERROR  DATABASE_ERROR(0x18)
#define E_DB_PARAMETER_OUT_OF_RANGE     DATABASE_ERROR(0x19)
#define E_DB_NOT_A_DATABASE_FILE        DATABASE_ERROR(0x1A)
#define E_DB_OPERATE_ON_CLOSED_OBJECT   DATABASE_ERROR(0x30)

FS error:
#define E_FS_NO_PERMISSION      FILESYS_ERROR(0x01)
#define E_FS_NO_SUCH_FILE       FILESYS_ERROR(0x03)
#define E_FS_IO_ERROR           FILESYS_ERROR(0x05)
#define E_FS_TRY_AGAIN          FILESYS_ERROR(0x0b)
#define E_FS_FILE_BUSY          FILESYS_ERROR(0x10)
#define E_FS_FILE_EXISTS        FILESYS_ERROR(0x11)
#define E_FS_NOT_DIRECTORY      FILESYS_ERROR(0x14)
#define E_FS_IS_DIRECTORY       FILESYS_ERROR(0x15)
#define E_FS_NO_SPACE           FILESYS_ERROR(0x1c)
#define E_FS_READ_ONLY          FILESYS_ERROR(0x1e)
#define E_FS_NO_ROOT            FILESYS_ERROR(0x21)
```

```
    Success codes:
    #ifndef NOERROR
    #define NOERROR         ((ECode)0x00000000L)
    #endif
    #define S_FALSE         ((ECode)0x00000001L)

    Severity values:
    #define SEVERITY_SUCCESS    0
    #define SEVERITY_ERROR      1

    Define the family codes:
    #define FAMILY_NULL         0x00
    #define FAMILY_KERNEL       0x01
    #define FAMILY_DRIVER       0x02
    #define FAMILY_CRT          0x03
    #define FAMILY_FILESYS      0x04
    #define FAMILY_TCPIP        0x05
    #define FAMILY_DRM          0x06
    #define FAMILY_CAR          0x07
    #define FAMILY_MOBILE       0x08
    #define FAMILY_GRAPHIC      0x09
    #define FAMILY_DATABASE     0x0B
    #define FAMILY_XML          0x0E
    #define FAMILY_XMLGLUE      0x0F

    Macros and constants for FAMILY_NULL error codes:
    #define E_UNEXPECTED            MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x00)
    #define E_NOT_IMPLEMENTED       MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x01)
    #define E_NO_INTERFACE          MAKE_ECode(SEVERITY_ERROR,FAMILY_NULL, 0x02)
    #define E_INVALID_POINTER       MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x03)
    #define E_ABORT                 MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x04)
    #define E_FAIL                  MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x05)
    #define E_NO_DEFAULT_CTOR       MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x06)
    #define E_CLASS_NO_AGGREGATION  MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x07)
```

```
    #define E_CLASS_NOT_AVAILABLE   MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x08)
    #define E_CATID_NOT_EXIST       MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x09)
    #define CAT_E_NODESCRIPTION     MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x0A)
    #define E_AGGREGATE_FAILED      MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x0B)
    #define E_UNAGGREGATE_FAILED    MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x0C)
    #define CO_NOTALLINTERFACES     MAKE_ECode(SEVERITY_ERROR, FAMILY_NULL,
0x14)
```

有关 ECode 操作的宏：

```
    //
    // Create an ECode value from component pieces
    //

    #define MAKE_ECODE(sev, family, code) \
            (ECode)(((family << 24) | (code << 16 )) | (sev << 31) )


    #define MAKE_SUCCESS(family, code) \
            (ECode)(((family << 24) | (code << 16 )) & 0x7FFFFFFF)


    #define ERROR_DETAIL(c) \
            (ECode) (c & 0x0000FFFF)

    #define ERROR(c) \
            (ECode) (c & 0xFFFF0000)


    #define SUCCEEDED(x)    ((ECode) (x) >= 0)
    #define FAILED(x)       ((ECode) (x) < 0)
```