事务性COM+编程
——创建可伸缩应用系统

Transactional COM+ Building Scalable Applications

[美]Tim Ewald 著

覃剑锋 柯晓江 等 译

- 清晰阐述 COM+ 工作原理
- 展示如何开发可伸缩应用程序
- 对实现细节的具体指导

ddison
Wesley

中国电力出版社
www.infopower.com.cn

事务性COM+编程

——创建可伸缩应用系统

Transactional COM+ Building Scalable Applications

[美]Tim Ewald 著

覃剑锋 柯晓江 等 译

中国电力出版社

Transactional COM+: building scalable applications (ISBN 0-201-61594-0)
Tim Ewald
Authorized translation from the English language edition, entitled Transactional COM+: building scalable applications, published by Addison-Wesley, Copyright©2001

CHINESE SIMPLIFIED language edition published by China Electric Power Press
Copyright©2003

本书由美国培生集团授权出版。
北京市版权局著作合同登记号：图字：AWP01-2001-2226

图书在版编目（CIP）数据

事务性 COM+编程：创建可伸缩应用系统 /（美）艾华德著；覃剑锋等译. —北京：中国电力出版社，2003
ISBN 7-5083-1553-7
Ⅰ.事… Ⅱ.①艾… ②覃… Ⅲ.软件接口，COM+一程序设计 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2003）第 057351 号

责任编辑： 朱恩从
丛 书 名： 开发大师系列
书　　名： 事务性COM+编程：创建可伸缩应用系统
编　　著：（美）Tim Ewald
出版发行： 中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88518169
印　　刷： 北京地矿印刷厂
发 行 者： 新华书店总店北京发行所
开　　本： 787×1092 1/16　**印　　张：** 19　**千　　字：** 426
书　　号： ISBN 7-5083-1553-7
版　　次： 2003年9月北京第一版
印　　次： 2003年9月第一次印刷
定　　价： 38.00 元

序　言

美国总统及南北战争时期的将军 Ulysses S. Grant 并不是音乐家。Grant 说："我只懂得两个音调，其中的一个是'扬基歌'（Yankee Doodle），而另外一个则不是。"

Grant 对旋律的二分法和构建一个可伸缩应用的问题有着共通之处：一个应用，要么具有足够的可伸缩性要么不具有，别无选择。

然而，对于应用程序的可伸缩性来说，真正成问题的是，它是一个在设计时期就要考虑的问题。如果应用程序的设计不正确，那它将永远不能伸缩。你需要预先做出正确的决定——以后再去修改它们可能会是一件极其痛苦的事情。而困难的是要明白什么才是正确的决定。在 Microsoft 环境下构建可伸缩的多层应用程序的规则是什么呢？更准确地说，使用 COM+（由微软提供，以帮助创建这些类型的应用程序的核心技术）来构建那些应用程序的规则是什么呢？

本书回答了这些问题。在本书中，Tim Ewald 首先快速地回顾了基础知识，以确保我们都能迅速掌握此问题的基本原理。因为构建一流的企业应用并不是一件简单的事情，所以本书彻底地深入到实质的细节，并提供了使用 COM+来创建可伸缩性应用所需了解的知识。最妙的是，这些信息已被概括成一系列便于记忆并容易理解的规则。

Tim 是惟一有资格编写这本指南的人。他通过大量的学习、教学和实践经历获得了对这个问题的精深理解。他将这些知识很好地融会贯通，而且是一个素质全面的聪明的小伙子。他也不会羞于表达他在实践经历中发展起来的观点。并不是所有人都同意他的观点，不过，能看到如此直率的人总是让人精神振奋的。

微软已经从一个以桌面软件为主的公司成长为现在的企业，它的软件正变得日益复杂。总体上说，这是一件好事情。因为我们尝试构建的应用所使用的软件也随之变得更加复杂了，所以我们需要微软提供的服务。不过，理解这个复杂性仍然颇具挑战性，而阅读本书就是这个过程必不可少的部分。

要想从 COM+中获得尽可能多的知识意味着要深入掌握它的工作原理。在你目前所了解的和你需要为构建可伸缩性 COM+应用而了解的知识当中，事务 COM+是我所见到过的最为直接的途径。

David Chappell

前　言

在我刚开始接触到 MTS（Microsoft Transaction Server，微软事务服务器），也就是 COM+的前身时，我就知道作为一个 COM 开发者，我的生活已经由此而永远改变了。很快我便发现，过去习惯使用的传统 COM 对象模型在 MTS 环境下并不能有效地工作。这个问题既让我苦恼又让我着迷，因此我开始着手了解个中的原因。一开始我很盲目地认为是 MTS 小组出现了某些问题——它的创建者根本不理解分布式对象。然而，随着时间的推移，我逐渐明白了 MTS 为什么会以这样的方式工作。很简单，答案就是可伸缩性。

MTS 被设计成用于简化可伸缩的分布式应用程序的开发过程，它所做的每一件事都是为了达到这个目的。根据这个观点，MTS 对对象所做的处理（例如，不在客户之间共享对象，以及在对象的事务终止时释放它们）对我来说就有意义了。正因为如此，我才花费了很多时间来编写以及教授关于 MTS 的课程。我遇到过很多开发人员，他们正朝着和我一样的目标而努力奋斗，而且他们需要帮助。像我一样，他们希望了解 MTS 是如何工作的，以及它为什么以这样的方式工作。最为重要的是，他们想了解如何设计使用 MTS 的系统。因此我编写了这本书。

编写这本书花费了很长的时间，以致于这本书根本不是关于 MTS 的，而是关于它的后代 COM+的。

COM+是什么？

COM+是一个运行时的环境，当类声明了它们所需要使用的服务后，COM+就会为类的实例提供这些服务。例如，如果一个类声明了它需要基于因果关系的调用同步，那么 COM+运行时就会确保每次只有一个逻辑的线程动作调用一个实例的方法。或者，当一个类声明了它需要一个分布式事务时，COM+运行时就会确保有一个这样的事务可用。现在，基于 COM+的系统能用 C++、Visual Basic 6 或者用其他 COM 友好的语言来编写；而在将来，这些系统能用 C#、Visual Basic 7 或者其他通用语言运行时（Common Language Runtime）友好的语言编写。（详情请参阅附录 A，“关于.NET”。）COM+运行时是许多高级技术的基础，其中包括 Internet Information Server（IIS）、Active Server Pages（ASP）、Site Server、Application Center 2000 以及 Biztalk Server 2000。

对于一种被设计用于在 Windows 平台上开发大规模分布式应用的技术而言，COM+是这种技术框架的基础。这个框架的当前版本被称为 Windows DNA，下一个版本则被称

为.NET。这两个版本有着类似的总体结构，而且都是以如下与可伸缩系统需求有关的三个假设为基础的：

1. 它们必须能被内部网络和因特网上的多个用户访问，运行基于浏览器以及自定义的客户应用程序。
2. 它们必须使用多台服务器机器来并行处理大量的客户请求。
3. 在发生故障的时候，它们必须是健壮的。

根据这些假设，从这两种框架引申出三条基本原则：

1. 系统逻辑是统一在服务器而不是在客户或后端数据库上的。服务器能共享资源（例如数据库连接），封装数据库模式和数据访问技术，并提供一个受到严密控制的安全环境。
2. 事务位于程序设计模型的中心地位。在遇到并行访问或系统故障的时候，它们提供一个标准模型以保护分布式的系统状态。多数系统状态都必须处于事务的控制之下（例如在一个数据库中）。
3. 一个系统中的组件之间使用一系列协议来进行通信。一般情况下，客户使用 HTTP 并越来越多地使用简单对象访问协议（SOAP）来和服务器进行通信；有时也会使用 DCOM 和微软信息队列（MSMQ）进行通信。服务器则通常使用 DCOM、MSMQ 和特定的数据库访问协议来互相通信，但有时也使用 HTTP 和 SOAP。

COM+被开发来简化遵循这些原则的系统的开发工作。其目标首先在 Windows DNA 框架中得到了确定，并将在.NET 中继续保持下去。

本书的内容

本书系统所要研究的是 COM+如何工作、为什么能这样工作以及如何构建基于 COM+的应用。除非理解了一个问题的基本特性、一般解决方案和你所要使用的技术的完整细节，否则你就不能编写出能够解决该问题的软件。概括地说，本书是关于可伸缩系统设计的。具体来说，它讨论了 COM+运行时的机制，其中包括了对进程、描述表、因果关系、线程、对象、事务和通信协议的使用。

值得期待的内容

下面是对本书每一章的简单描述：

第 1 章“可伸缩性”描述了可伸缩性的基础问题，解释了为什么可伸缩系统要使用事务以及它们为什么不使用传统的对象模型。它发展了在基于 COM+的系统中使用的基本对象模型。

第 2 章“原子”描述了描述表和因果关系，所有的 COM+运行时服务都建立在这两个基础的构成部分之上。它解释了这两个构成部分如何与对象相关联以及对象是如何跟它们互相作用的。

第 3 章“机制”详细介绍了描述表和对象之间的关系，其中包括了接口指针的描述表相关性和描述表表示在时间和空间上的系统开销。

第 4 章“线程”介绍了套间以及活动集，COM+分别用这两个高级的构成部分来控制管理一个对象的线程相似性和同步的程度。

第 5 章“对象”把注意力集中于对象库和即时激活（JITA）以及这些服务是怎样改变一个对象的生存周期以更有效地使用资源的。这一章特别说明了 JITA 所提供的可伸缩性优点。

第 6 章“事务”探讨了本地和分布式事务的结构并介绍了事务流，事务流是 COM+用来把对象和分布式事务关联起来的高级构成部分。

第 7 章“隔离性”讨论了数据库用来终止互相干涉的事务并同时把并发性和吞吐量最大化的基础技术。本章也讨论了跨事务应用程序级加锁方案。

第 8 章“协议”阐述了 Internet Information Server 和 COM+之间的集成，其中包括 ISAPI DLLs 和 ASP 页是如何与描述表、套间、活动集和事务流相关的。此外，本章也讨论了 SOAP 和 MSMQ。

第 9 章“设计”提供了对基于 COM+的系统设计的一般建议。其中的主题包括：使用一个或多个事务来实现客户任务；高效率的数据访问；中间层；共享状态、每客户会话状态管理；以及在伸缩性和重用性之间的固有矛盾。

此外，本书还有四个附录。附录 A“关于.NET”讨论了向.NET 的转变并解释了 CLR 类是怎样利用 COM+的。附录 B“构建一个理想的连接池”演示了如何使用对象池来构建一个数据库连接池，这个连接池比 OLE DB 提供（并由 ADO 使用）的连接池具有更好的适应性。附录 C“调试”提供了有用的信息，这些信息令调试 COM+代码变得更加容易。附录 D“目录管理器属性和组件服务浏览器属性页”包含了一些图表，这些图表在组件服务资源管理器属性页上把目录管理器属性映射成用户接口元素。

不应期待的内容

本书是为那些正在设计和实现基于 COM+的系统开发人员服务的。它假定你知道如何实现 COM 类并知道如何编写简单的 ASP 页。同时，假定你知道如何使用 COM+和 IIS 管理工具、组件服务资源管理器（CSE）以及因特网服务管理器。

有三个关于 COM+的主题我在本书中没有介绍。首先，当这本书在几个相关的地方叙述性地提到基于 COM+角色的安全性时，并不会包含关于这个主题的完整的处理方法，这是因为在 Keith Brown 的优秀著作《Programming Windows Security》中对此已经有详细的讨论了。

其次，我忽略了以下两个辅助的 COM+服务[QC（排队组件）和 LCE（松散耦合事件）]因为这两种机制有着明显的局限性，而这些局限性又会在大多数情况下使它们失去作用。特别地，尽管 QC 能被用来给一个 COM+服务器进程异步地发送信息，但是它们不能被用来给其他进程发送信息，也就是说，回到了一个客户进程。虽然 LCE 可以使出版商将一个事件发送给多个订户而无须知道他们是谁，但这种默认的发送机制是一种同步的、分块的方法调用。通过和 QC 一起使用，可以使 LCE 变成异步的。但这样的话，

事件就只能被发送到 COM+服务器进程中了。这些问题使得这两种服务从根本上对于双向的客户-服务器通信是无用的（在某些情况下，它们可能会对服务器-服务器通信有用）。一般而言，我还是推荐使用微软信息队列（Microsoft Message Queue）而摒弃 QC 和 LCE，这能使我们不受这些局限性的限制，从而容易地建立起等价的功能集。

最后，我并没有对补偿资源管理器（CRM）进行介绍，因为缺乏必要的时间和篇幅。

规则

这不是一本 COM+大全。不过，我尽了自己最大的努力，提供尽量具体的建议来帮助读者理解如何设计和实现基于 COM+的系统。有些特殊的建议在书中标识为“规则”，并采用如下突出的形式：

规则 **P.1**

注意这些规则。

在后面的章节中只有少量的规则，这反映了一个简单的事实，即讨论的主题越复杂，关于如何做的具体的指导方针就会越少。最后一章为此做了弥补，这一章提供了关于系统设计的一般性建议，这些建议遵从早先定义的那些规则。

关于源代码的一点提醒

像 COM+一样，本书具有通用的程序设计语言风格。在任何情况下，它都尽可能地使用普通术语，不过，在必要的情况下也讨论了特定语言的问题。本书中的大多数实例源代码都是用 C++编写的，不过也有些是用 Visual Basic 和 JavaScript 编写的。我之所以选择 C++，部分原因是因为它是惟一一种允许你探究 COM+的所有奥秘面的语言，另一部分原因是由于它是我个人的语言选择（或者说，直到 C#问世之前它都是的）。另外，本书大多数实例代码的编写风格都是从我的朋友 Chris Sells 那里学来的。这种风格在很大程度上利用了 ATL 的精确类型（例如 CComPtr、CComBSTR 等等）和蹩脚的异常处理机制，也就是说，从函数的中间返回。不管怎样，我的语言或我的风格都不应该被看作是代表性的，你应该用你喜欢的语言和风格来编写自己的 COM+类。

附加信息和勘误

本书包括一个网站，该网站拥有实例代码和其他的资源。它的 URL 是：http://www.develop.com/books/txcom。

我已经尽了最大的努力来确保本书没有错误。然而，考虑到这项工作所涉及的范围，

尤其是考虑到对 COM+运行时奥秘的探究，注定会出现一些问题。如果你找到了错误，请把它转寄到本书的网站，我将在那里保留一个最新的修改列表。

致　谢

首先，我要感谢 Sarah Shor，她陪我共同度过了很多时间。毫无疑问，你比所有的技术都重要。同时还要感谢我家庭中的其余成员，这么长时间以来，我一直忽略了你们，而你们却毫无怨言。另外，要特别感谢 Alan Ewald，他一直和我分享他的关于分布式系统的经验和观点。

感谢 DevelopMentor 的所有技术同事，在这个令人惊讶的特殊团体中，我被赋予了足够的工作特权。通过参与关于 COM+原理的、看起来像是永无止境的讨论，你们已经对我的这项工作提供了你们想象不到的帮助。特别地，要对 Craig Andera、Dan Sullivan、Martin Gudgin、Jon Flanders、Bob Beauchamin、Stu Halloway、Simon Horrell、Keith Brown 和 Chris Sells 致以我真诚的谢意，是你们一直相信我将会讲述一个有趣的故事。还有，我必须特别感谢 Don Box，你了解在我的心中一直有一首歌需要唱出来，而且你帮助我度过了一段丰富多彩且充实的生活。还要感谢在 DevelopMentor 的所有其他人。在我完成这项工程的过程中，你们对我非常耐心。尤其要感谢 Mike Abercrombie，你一直非常理解我，认为我的技术工作必须首先问世。

感谢 Simon Horrell、Dan Sullivan、Bob Beauchamin、Stu Halloway、Martin Gudgin、Alan Ewald 和微软的 Mary Kirtland，你们复审了所有的章节并提供了反馈意见。感谢所有的学生、会议的出席者和邮件列表的参与者，你们也提供了本书的不同部分的反馈意见。感谢微软的 Joe Long 和 Jonathan Hawkins，Joe Long 回答了我关于各种 COM+运行时服务机制的问题，而 Jonathan Hawkins 则解释了.NET 和 COM+两者之间的关系。

感谢 David Chappell 为我写的序言，而且为我包装得这样好。是你让所有人都有所期待。

感谢在 Addison-Wesley 的所有人，你们帮助出版了这本书。这其中包括 Marilyn Rash，尽管我很拖沓，但你还是加快了本书的出版工作；John Wait 为我解释了某些技术出版行业的内部运转方式；而 Carter Shanklin 则为我签定了合同（在让 Addison-Wesley 处理其他事务之前就为我完成了这个工作）。另外，最重要的是，必须感谢我的编辑 Kristin Erickson，他一直陪着我完成这项工作并忍受我的打扰，而且到现在他仍然很乐意接我的电话。

最后，感谢微软，是你们创造的这些技术让我度过了过去的十年时间。COM+是一个谜；让我们继续探索这个谜的谜底吧。

Tim Ewald
Nashua,NH

目　录

第 1 章

可伸缩性

假定你正在努力构建一个三层的分布式系统，且这个系统能并行地为许多客户请求提供服务。你的系统可能是作为内部使用，或者由你公司的客户和商业伙伴外部使用，他们希望通过因特网来访问它。你希望你的系统具有一个可伸缩的体系结构，不过，这到底意味着什么呢？分布式系统的体系结构经常是根据它们的吞吐量来评价的，也就是说，根据它们在一个特定的时间内完成特定数量的逻辑操作的能力。其目标是，即使在高峰负荷时间，该体系结构也能确保有足够高的吞吐量以满足客户需要，而无须为个别的请求牺牲合理的响应时间。可伸缩的体系结构能随着客户需求的增加而增大吞吐量，为此只需利用附加的硬件设计而无须重新。我编写这本书的目的是帮助你理解如何设计基于 COM+的、可伸缩的体系结构，这样的体系结构使用对象来为一个问题域的特定部分建立模型。很明显，COM+设计风格要求从传统的、在单机上使用的面向对象模型以及基于 COM+的、小型的分布式系统开始。在我开始解释为什么之前，让我们回顾一下关于可伸缩性的基础概念。

1.1 可伸缩性基础

一个体系结构的伸缩能力通常是根据它所提供的吞吐量增长的最终高度来评定的。考虑一下图 1-1 所示的曲线图，它以硬件资源为对比，为一个假设的设计计算吞吐量。注意，这条曲线是呈非线性增长的。换句话说，吞吐量的增长率逐渐降低而且最终趋于平稳。这个平稳值代表了体系结构的最大吞吐量而不管有多少新的硬件资源可用。

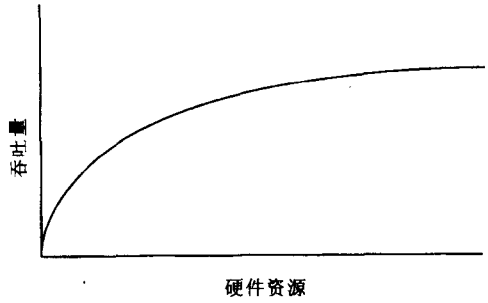

图 1-1　与硬件资源对比的吞吐量

决定吞吐量最终爬升高度的因素是运行时对资源的争用程度。当多个并发操作试图对一个共享资源进行排他性的访问时，争用就出现了。因为每次只有一个操作能访问此资源，所以所有其他的操作都会被阻塞并等待，直到此资源可用或者直到它们决定（或被迫）放弃为止。当对一个资源的争用增加时就形成了一个瓶颈。在这种情况下，增加并发性并不能提供额外的好处。实际上，它很可能还会让事情变得更糟糕。一个正在等待一个资源的操作，它需要用这个资源来完成工作，可能已经获得了其他的资源，而这些资源正是其他操作所需要用来完成其工作的。这样的话，那些其他的操作也不得不被阻塞了。由此而产生的连锁反应会急剧减慢操作速度，而且有可能导致死锁。一旦这个过程开始了，那么启动更多的操作只会把事情弄得越来越糟。

可以采取两个步骤来降低对一个特定资源的争用。首先，确保资源只是在绝对必要的时候才被使用，而且确保使用过程尽可能的短。换句话说，就是要让代码以尽可能高的效率使用宝贵的资源。不幸的是，提高效率仅仅能推迟对一个资源的访问变成瓶颈所需的时间。如果并发性持续增加的话，那么就算有世界上最有效率的代码也没用。到最后，当过多的操作需要访问这个资源时就有必要采取第二个步骤了。这个资源将不得不被复制。

硬件是一种争用资源。服务器需要访问硬件资源（例如 CPU 周期、网络带宽和磁盘）以处理客户请求。如果一个服务器接受了过多的并发的客户请求，那么对其硬件资源的争用就将导致瓶颈的出现。一旦发生了这种情况，那么接受更多的请求便是错误的，并且会导致返回速度的快速下降。升级资源和优化使用这些资源来为请求提供服务的代码能推迟瓶颈形成的时刻。一旦耗尽了所有的升级选择或者升级的代价太高昂，又或者代码也已经是最有效率的话，那么最后的步骤就是复制资源——增加一台服务器。

信息也是一种争用资源。大多数客户请求需要访问某种类型的数据。在许多分布式系统中，每一块数据都是放置在一个地方，也就是在一个特定的服务器机器上特定的进程内，通过使用某种形式的进程间通信（IPC），数据块能在此进程内被处理。因为一块数据存在于一台机器中，所以所有对其硬件资源的有效使用的关注都是有关联的。不过，还有另外一个问题。如果多个并发客户请求需要对同一块信息单独进行访问，那么它们的工作必须通过使用一个排它锁来同步。如果一个服务器接受了足够多的并发客户请求，而这些请求需要访问同样的数据块，那么对锁的争用将会导致一个瓶颈出现。在对硬件资源的争用成为一个问题之前，这种情况可能会出现也可能不会出现，具体取决于正在

执行的操作和加锁策略的完善程度。优化那些使用某些数据来为请求提供服务的代码能推迟锁变成瓶颈的那个时刻的到来。一旦代码的效率已经是尽可能高了，那么最后的步骤就是复制资源——建立这个数据的一个副本。

不幸的是，复制数据可能并不是件容易的事情。对于那些不会改变的数据，例如只读的引用数据，这不会有问题。你可以根据需要对它进行任意次数的复制，也可以将其复制到任意多的地方。不过，对于那些会改变的数据就有问题了。如果数据被复制，你就需要某种方案来保持所有副本的同步。这会显著地增加设计的复杂度。对效率的关注和分布式的故障模式使我们不可能始终保持在多台机器上的数据副本的同步。相反，副本的同步只是周期性的，这就意味着所做的改变从数据的一个副本到另一个副本的传送会有等待时延。如果被复制的数据是用来响应客户请求的，那么一个客户得到的应答可能会取决于用来响应它的查询的是数据的哪一个副本。数据的复制是否是必须的或者是否是可接受的则取决于你所构建系统的类型——尽管对于很大型的系统来说，它将不可避免地成为必要的（或许，你能够在复制数据之前先对它进行分割从而解决某些瓶颈问题。然而，在每一块数据分割中，对数据的争用最终还是会导致复制）。

很明显，对共享硬件和信息资源的争用程度抑制了体系结构的可伸缩性。可伸缩设计的最终目标是消除争用，但这是不可能的。在每一个系统中都有一个瓶颈，当形成了这个瓶颈时，它就会导致吞吐量达到如图 1-1 所示的稳定水平。在所有的瓶颈都被消除了之后，吞吐量就会提高，不过，因为总会存在着某些或明或暗的障碍，所以你不能得到持续的、线性的吞吐量增长，吞吐量始终会有一个稳定值。必须从系统中消除的瓶颈个数取决于你需要多大的吞吐量。刚好消除足够的瓶颈以满足你的需要是可伸缩系统设计的本质。

设计一个可伸缩系统（就如何管理资源作出正确的决定）是比较困难的。设计一个使用对象的可伸缩系统尤其是一个挑战，更不用说你使用的是 COM+了。从单机单进程应用程序发展而来的许多传统的面向对象设计都已经被应用到分布式的对象应用程序中。不幸的是，这些设计的趋势是把问题域中的每一个抽象实体都映射为位于内存某处的单独的一个对象，而这个趋势会抑制可伸缩性。这个问题的症结在于对象本身，在被用于共享资源的时候，这些对象代表着一个争用源。应该理解为何要切实地掌握对象一致性的概念，这是实现支持可伸缩性的对象模型的必不可少的第一步，而可伸缩性是基于 COM+系统的目标。

1.2 一 致 性

大多数开发人员都同意对象是状态和行为的结合，但还有一个关键因素，即一致性（identity）。一致性是对象的一个独特的方面，它控制着对象的类的行为向状态映射的方式。在现代程序设计语言中，例如 C++、Visual Basic（VB）和 Java，这个方面就是对象在内存中的位置。换句话说，一致性是根据地址来定义的。

考虑下面这个简单的类，它是用 C++定义的，对一个人进行了描述：

```
class Person
{
    long m_nAge;
public:
   void AgeGracefully (long nYears) { m_nAge += nYears; }
};
```

这个 Person 类用数据成员 m_nAge 来跟踪年龄，而方法 AgeGracefully 用 nYears 来调整 m_nAge。

任何使用 Person 类的进程，在运行时都会在内存中拥有其代码的一个单独的副本。随着副本的出现，任意数量的 Person 对象就能被实例化并得到处理。例如，你可以创建三个人并分别用 17、29 和 50 年来设定他们的年龄：

```
Person p1, p2, p3;
p1.AgeGracefully(17);
p2.AgeGracefully(29);
p3.AgeGracefully(50);
```

这个例子引出了一个有趣的问题：如果在内存中只有 AgeGracefully 方法代码的一个副本，那么它是如何知道自己应该影响哪个对象的状态呢？

答案是一致性。每一个类的每一个方法都有一个附加的、隐含的第一个参数，这个参数是对对象状态的一个引用，而行为则会被应用到这个状态。对调用程序来说这是隐藏起来的。它会和其他参数以及返回地址一起被推入到堆栈中。然而，从被调用程序的观点来看，这个引用可以被用作一个成员变量，也就是 C++中的 this 指针、Visual Basic 的 Me 引用或 Java 的 this 引用。AgeGracefully 方法能被重新实现并显式地使用这个引用。

```
void AgeGracefully(long nYears)
{
   this->m_nAge += nYears;
}
```

虽然这样做可能会被看成是更加准确的，但是方法通常不是以这种方式实现的。相反，每一个方法都依靠编译器来隐式地使用这个引用——当它指向数据成员或它的类的其他方法的时候。

1.3 COM 中的一致性

COM 中的一致性也是基于地址的，不过其机制稍有不同。COM 把指针当作接口，而不是从总体上当作对象来处理，而且它根据所有 COM 类都要确保实现的接口 IUnknown 来定义一致性。COM 的规格说明规定，在一个对象被请求通过一个对 QueryInterface 的调用从而调用 IUnknown 的时候，此对象必须总是返回同样的物理指针数值。如何将 COM 的一致性概念映射到一个基础的实现语言的一致性概念中，是一个

实现上的细节。在 C++中，它通常是多重继承的一个副作用。而在 Visual Basic 和 Java 中，它是由虚拟机来管理的。虽然 COM 中的一致性与 C++、Visual Basic 和 Java 中的一致性非常相似并且彼此对应，不过它提供了一个额外的特性，这个特性是语言层次上的一致性所不能提供的。在面对分布式的时候，COM 的一致性能继续维持，甚至当代码调用对象的一个方法而且这个对象本身存在于一个网络的不同机器上分离的进程中的时候，这一点也不会改变。

远程的 COM 体系结构使用代理对象来支持这个行为。无论何时，只要在一个描述表中执行的代码需要调用存在于另一个描述表（或许是在另一台机器上的另一个进程）中的 COM 对象的一个方法，那么这个调用就要通过一个代理进行。[1]代理（proxy）是真实对象的一个替身，这个替身知道如何组装方法调用并把它们发送给真实的对象，而不管它在哪里。存根（stub）——运行在真实对象的描述表中的一个对象，会截取被发送的调用并把它们转换回标准调用。

当接口指针被传递给接口方法和系统 API 并从两者中返回的时候，代理和存根会根据需要被自动创建。COM 底层设施确保了对象在它自己的描述表中至多与一个存根相关联，而且在任何其他的描述表中至多与一个代理相关联，如图 1-2 所示。在此图中，由一个惟一的 IUnknown 指针值所标识的对象存在于描述表 C 中。此外，代理也是由一个惟一的 IUnknown 指针值所标识的，它代表了在两个外部环境 A 和 B 中的任意一个原始对象。这些保证使得 COM 的一致性概念可以跨描述表保持。每一个存根都被连接到一个特定的 IUnknown 指针值，而每一个代理都被连接到一个特定的存根。每一个代理都是一个在它自己的权限内的 COM 对象，而这个代理是由惟一的一个 IUnknown 指针值所识别的。换句话说，在它自己的描述表中，COM 对象是由它的 IUnknown 指针的值所惟一标识的。在其他的描述表中，COM 对象是由它的代理的 IUnknown 指针的值所惟一标识的。这个体系结构使任何数量的客户都有可能访问相同的 COM 对象。

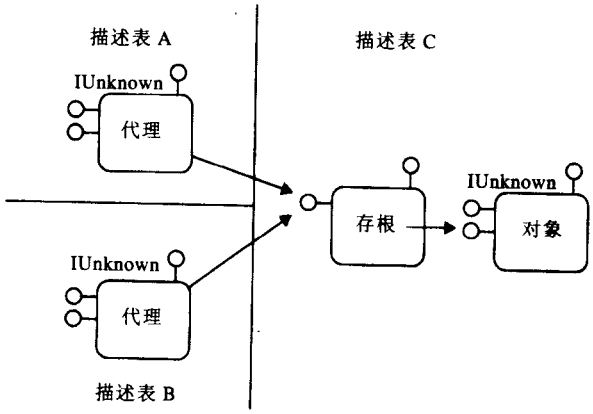

图 1-2　跨描述表一致性

1　描述表就是一个环境，在该环境中一个进程为对象提供了运行时服务。第 2 章有对描述表的详细讨论。