HZ BOOKS

微软公司核心技术书库

Microsoft

Windows

附 赠
CD-ROM

COM+

技术解决方案设计

Designing Solutions with COM+ Technologies

Ray Brown
(美) Wade Baron 著
William D. Chadwick III
梁玉柱 贾 颖 等译

机械工业出版社
China Machine Press

微软公司核心技术书库

COM+技术解决方案设计

Ray Brown
（美） Wade Baron 著
William D. Chadwick Ⅲ
梁玉柱 贾 颖 等译
前导工作室 审校

机械工业出版社
China Machine Press

本书介绍在多种开发环境下使用COM+技术开发各种企业应用的技术。主要内容包括COM+基础、体系结构模式与解决方案、企业环境中的COM+三个部分。本书实例丰富，理论结合实际，可帮助读者深入理解COM+技术。

本书所附光盘包含书中示例程序源代码，方便读者使用。无论对于COM+初学者还是具有一定基础的编程人员，本书都有很高的参考价值。

图书在版编目（CIP）数据

COM+技术解决方案设计 /（美）布朗(Brown R.)等著；梁玉柱等译. - 北京：机械工业出版社，2001.9

（微软公司核心技术书库）

书名原文：Designing Solutions with COM+ Technologies

ISBN 7 - 111 - 09290 - 2

Ⅰ.C… Ⅱ.①布… ②梁… Ⅲ.软件接口，COM - 程序设计 Ⅳ.TP311.52

中国版本图书馆CIP数据核字（2001）第055712号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：谢君英 张鸿斌

北京市密云县印刷厂印刷 · 新华书店北京发行所发行

2001年9月第1版第1次印刷

787mm × 1092mm 1/16 · 41印张

印数：0 001-5 000 册

定价：85.00元（附光盘）

前　言

扎棉机的发明者伊丽·惠特尼，早在1798年就提出了可交换部件这一划时代的设计思想。那时，每个器具，不管是钟表还是家具，都是一个一个采用手工制作完成的。惠特尼打算采用机器生产具有某种规格的武器部件，这些部件能够任意组装并重复使用。这个设计思想首先在与美国政府签定的一万支步枪的生产合同中得以应用。尽管它开始遭受了一些挫折，但最终为美国军方提供了性能优良的武器装备，与当时战场上所用的其他武器装备相比，这些装备更容易维护和使用。惠特尼的产业化设计思想加速了工业革命的进程，并成为目前所有制造业的核心思想。

组件技术便是可交换部件这一设计思想成功地运用于软件行业中的实例。在采用组件技术之前，我们每研制一种新的软件，都要从头开始一个一个地开发，也就是说，我们只能有限地重用部分软件源代码。借助组件技术，我们能采用已创建的具有标准接口的软件来组装新的软件。

微软公司已经成功地为我们提供了COM这一非常具有实用价值的组件技术，然后采用微软事务服务器（MTS）使之更为完善，现在又提出了COM+技术。但是，微软公司仅仅发明了COM，是伊丽·惠特尼发明了可交换部件的原理。正如可交换部件为制造业和现代社会带来的变革一样，采用COM+技术作为软件项目的基础和核心，同样是软件工程的一个巨大发展和进步。这的确是令人兴奋的，难道你不这样认为吗?

1995年，也就是惠特尼提出划时代设计思想的200年之后，Sun公司提出了“网络就是计算机”的销售口号。我对Sun公司的人们提出如此正确的口号感到不可思议。在因特网络和分布企业软件时代，个人计算机的作用将会弱化，网络将成为信息处理和传送的主体。采用COM+技术，将允许你在新的环境中毫不费力地重用软件部件。COM+透明地实现了位于不同进程和计算机上的对象的关联，透明地实现不同软件部件之间线程和可执行模块的协调，显著地提高了软件的可伸缩性，并允许你的软件根据因特网的需求方便地增长，从而更好地满足成千上万个并发用户的需要。

COM+是一种规范，也是一种切实可行的技术，COM+还是一门哲学。这个规范详细说明了COM+互操作性的规则，便于新的语言和代码实现加入到COM+构架中。这种技术使软件重用成为现实，使软件部件具有互操作性，从而实现诸如调用串行化和自动事务处理之类的服务。采用这种技术，可以根据需要选用开发语言和工具。COM+不仅功能强大，而且非常灵活。使用这种技术，无须改变程序结构便可以将不同的部件组合在一起。

读者一定知道，为了编写一段源代码，必须懂得编写软件代码所用的开发工具的某些知识，你必须掌握软件所用的应用编程接口(Application Programming Interface，API)的调用方法，以及API调用所产生的效果。这就是该产品的技术要点。在COM+环境中，这些技术要点意味着要懂得如何获得二进制兼容性，懂得COM+库 API函数调用所完成的工作，懂得集合结构的实质，懂得同步支持和运行时编译(Just-In-Time，JIT)，懂得COM+扩展服务是如何工作的等等。本书

将全面讲述这些内容。

但是，与简单地构造或维护一段独立的源代码不同，设计一个软件系统，除了掌握一些技术要点之外，软件工程师还必须熟悉设计软件所用的开发工具。为了构造健壮的、可维护的、可重用的和可升级的软件，并且能够在规定的时间以及所允许的预算之内完成软件，必须具有结构化设计思想。也就是说，必须懂得结构化设计在物理设计和逻辑设计中的作用。我相信，没有部件技术和中间件等结构化设计技术，就无法凭空设计一个有效的系统。对于软件开发的任何阶段，从概念到分析、设计，从开发到质量保证、交付和支持，都是如此。本书的最终目的就是通过介绍COM+技术以及如何在方案设计时使用COM+技术，从而使你成为一名好的软件设计师。我们在本书中罗列了大量的COM+项目，并对其进行了分析，希望读者能够在自己的方案设计中直接使用这些已经测试过的COM+项目，相信你会从中获益。

要从别人的教训中吸取成功的经验，不要重蹈覆辙。如果一个建筑工程师没有研究过同事的成功和失败，他就无法建造一座桥梁。同样，对于一个软件工程师来说也是如此。我们开发的软件项目中的问题，与其他工程领域中的问题将会一样多。曾几何时，一些软件项目因为花费太长的时间、开销太大但提交的成果太少而遭受失败，我认为我们应该停下来，思考一下如何改进我们的项目。让我们通过迎接挑战来提高成功的机会吧。让我们把我们的职业作为工程训练吧。让我们不要犯别人曾经犯过的类似错误。让我们学习这些方法吧。为了这些目的，本书将从三个专题讨论COM+技术。

1) COM + 基础。在这个部分中，我将讨论基本的COM+工作原理以及如何正确地使用COM+。由于本书的主要目的是精通体系结构，而不仅是一些技术细节，因而本书主要讲述如何利用COM+技术。这一部分将讨论错误处理、灵巧指针、字符串和并发操作，以及如何有效地利用COM+以使不同语言编写的部件能够进行互操作。之所以选择这些专题，是因为我在访问许多项目小组时发现许多非常有经验的开发者经常在这些领域内犯同样的错误。其中一些错误是由于对COM+服务一知半解所致，但是大多数的错误来自忽视对COM+的充分利用。例如，无法从CoCreateInstance的文档中得出这样的结论：在采用C++编程时返回的结果接口指针应该为灵巧指针，并且某种灵巧指针配合操作将导致内存泄露。但是，对于COM+编程来说，通过使用基本的COM+服务，可构造健壮的、可重用的和可升级的部件。除了分析新的COM服务和一些原始的服务之外，这一部分的目的是将上述知识传授给读者。

在阅读本书的其他部分之前，应该首先阅读第1、2、3章。后面章节中的示例程序代码都假定读者已经熟悉了基本的错误处理、字符串和接口指针管理等内容。在阅读第13章之前，需要首先阅读第4章。在讨论可伸缩性等内容之前，需要明白基本的COM+并发操作知识。图Ⅰ-1显示了阅读本书内容的最好方法和途径。

2) 体系结构模式与解决方案。这一部分的内容论及了在进行COM+设计时经常遇到的问题的解决方法。在这里，你会找到使软件能正确执行并且容易升级和维护的方法。由于我们已经知道一个成功的项目必须具有可升级性、可维护性和代码可重用性，因此在全书内容中将贯彻这些目标。这一部分的内容尤其如此。这里我们讨论了源代码和组件重用的原理，跨平台数据流和对象持久的方法，与语言无关按值排列和引用循环管理等。这一部分的内容还讨论了如何在COM+项目中进行通用程序设计，这主要关于C++的话题。这一部分的每章内容都包含详细的

解决方案框架。

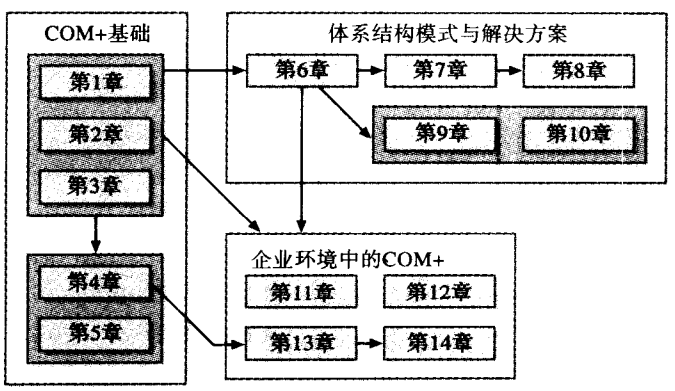
建议读者按照第6、7、8章的顺序进行阅读。第6章阐明了贯彻于本书其他章节的重用策略。

3) 企业环境中的COM+。这一部分详细阐述了用于可伸缩的企业环境的分布式COM+项目所使用的有价值的技术。第11章提出了一个四层应用体系结构，它将中间层分为客户和服务器。这样的体系结构通过健全的商务层接口，确保了客户端的可维护性；并且对于跨越不同机器而激活的商务对象，能够通过无状态接口确保性能。第12章讨论了简单对象访问协议（Simple Object Access Protocol，SOAP）。解释了在COM+项目中应用SOAP的环境，如何学习SOAP以及应该注意的问题。第13章详细讨论了在不考虑健壮性与可维护性之间的折衷时使用MTS在软件并发操作方面带来的巨大进步。你会进一步熟悉COM+的技术方法，从而适应因特网时代软件不断升级的需要。第14章详细阐述了在数据库中访问共享对象状态的方法，简要叙述了通过Microsoft统一数据访问 (Uniform Data Access，UDA)策略来获得灵活性的方法。除此之外，你还会看到一系列访问数据仓库的方法以及这些方法之间的比较。

可以按任何顺序来阅读这些章节。然而，我们建议在阅读第14章之前首先阅读第13章的内容，因为第13章详细叙述了COM+项目进行数据访问的意义，而第14章则详细叙述了有效的数据访问方法。

将配套光盘插入光驱中，运行启动应用程序，如果启动程序不能自动运行，请运行光盘根目录中的StartCD.exe程序。

关于本书，读者也可以访问作者的网站http://www.magenic.com/complusbook/。



图I-1　章节图

由于我们所面临的挑战，成为一个软件工程师是一件激动人心的事情。现在，我们每天都在做着几年前连想都不敢想的事情。COM+可以帮助你迎接这些挑战，我们衷心希望读者通过阅读本书而深入理解COM+技术。

Ray Brown

2000年10月于旧金山

第一部分 COM+基础

第1章 错误处理

软件开发者曾一度把错误处理视为令人分心的事情和事后完成的工作。这种态度是不对的。我曾经目睹了许多开发者花费数星期以至数月的时间来查找故障的根源。如果开发者最初使用错误检测和传播机制来预测某种特殊的失败模式，事后只需花几分钟的时间就可诊断出故障位置。或者这样考虑：如果你最后得到一个类似“0x8004024C”但没有任何说明的错误代码(只有一个含糊不清的意思)时，从哪里查找它的含义呢？对于预先包装过的组件来说，正确和完备的错误处理是至关重要的；从长远来看，它也可以极大地提高软件项目的可维护性。请永远记住，那个六个月后遇到组件出错的开发者可能就是你自己。

生成清晰的错误代码和上下文信息，从以下方面可看到有利于COM+接口属性和方法的调用者：

- 它可以避免调用者选择不适当的代码路径，从而防止更严重故障的发生。
- 它能够告知调用者正确的调用方式(这时使用成功代码)。
- 它允许调用者基于错误代码值选择代码路径，从而能在故障情况下采取补救措施。
- 它为调用代码的开发者或最终用户提供了界面友好的故障描述，甚至可能是帮助文件参考。所生成的接口的GUID以及实现对象的 ProgID也可以被返回。这些信息在代码调试时非常有用，但也可以被最终用户或信息服务（Information Services，IS）人员用来报告软件故障，从而满足软件开发者的需要。

从某种意义上讲，上述所有的功能和灵活性会使COM+错误处理模型变得比较复杂。开发者之所以对错误处理程序具有畏难情绪，并不是因为他们想偷懒，而是因为他们容易混淆COM+错误处理的原理和规则，并且缺乏能够应用于整个项目的一致方法。本章的主要目的就是有效地清除上述两个障碍。

1.1 COM+错误和结构化异常处理

那些懂得结构化异常处理（structured exception handling，SEH）并在最新版本的Windows操作系统担负着重要角色的开发者，有时会提出这样的问题：为什么Microsoft 公司不选择SEH作为COM错误处理框架的基础呢？那些知道Microsoft Visual C++编译器的当前版本依据SEH来实现C++异常的开发者也常常有这样的疑问。他们推测，集成COM和SEH将允许他们跨接口边界抛出C++异常并且捕获它们，就像捕获跨C++函数和类方法抛出的异常一样。

至少有两个理由来说明为什么COM+的错误处理不是基于SEH，即使只考虑Windows 操作

系统平台（区别于非Windows平台，例如UNIX或MVS)。因此，我们必须首先回顾COM产生的起源以及它的发展历程。

首先，COM+的核心是一个二进制兼容标准，用来实现分布组件之间的互操作性。即使在今天，COM+运行时并没有把它本身插入到位于同一环境的调用者和对象之间。在调用者和被调用者之间的二进制接口之间采用vptrs 和 vtbls （虚函数指针和虚函数表）定义，这些虚函数表的定义正好与最早版本的Visual C++编译器实现的结构相一致。为了提供COM+联编，一个编程环境所要完成的就是采用C++编译器执行虚函数调用所用的方法来执行COM+接口方法调用，并且在该环境中为接口实现提供一个vtbl布局。为了与COM+集成，编程环境不需要知道SEH。这样的需求会致使一些编译器和解释程序更难于与COM+集成，从而妨碍COM+的应用。并且，不要无视这样的事实，许多与COM+联编的环境（例如C和Perl)并不提供带外的异常处理模型。

其次，COM+具有分布的特性。SEH异常处理尤其适合于展开特定线程的堆栈段并适合于在该线程中捕获错误信息，但并不适合于不同线程、进程和计算机之间。编程环境控制着EXCEPTION_RECORD结构的 ExceptionInformation 字段，从而为用户定义的错误来充分利用SEH；但是把它们作为一个整体，包装起来交付给另外的软件或另外一台主机使用，将是非常困难的，甚至是不可能的。

对于C和 C++程序员来说，尤其需要注意的是：一定不要让一个结构化或C++异常错误跨COM+边界。在某些情况下，你可以采用如下的方法来实现：占位程序可为某个异常处理器所知晓，并把结构化异常转换为COM+错误代码RPC_E_SERVERFAULT（服务器抛出一个异常）中。任何结构化异常都会导致这个错误的发生，尽管这常常是由于对服务器的非法访问所致。当没有代理和占位程序时，也就是说，调用者和被调用者在同一个上下文中，违反该规则会导致调用者遭到严重破坏，致使出现崩溃甚至立即中断执行。对于采用C++实现的调用程序尤其如此，因为它们一般希望根据它们的需要公告COM+错误语义，而不是使用try/catch模块包含接口方法调用。

1.2 COM+ 错误处理模型

COM+错误处理模型由两部分组成：返回代码和上下文信息。在本节内容中，我们将仔细研究这两个内容以及它们之间的相互关系。

错误类型

项目体系结构通常将错误出现的条件划分为以下几种情况。

1. **执行假定**。这是一个假定，例如，假定一个局部索引变量对于循环操作都在一定的范围中，这反映了开发者的意图，也就是他所编写代码总是具有某种行为，并且必须只间接地由外部因素来确定。这样的假定最好在调试时来验证，并且所有的验证代码必须在发布联编之前删除，从而提高代码执行的效率。这种验证代码一般称为断言（assertion)。

Microsoft Visual C++使用Assert.h头文件、活动模板库(ATL)宏以及Microsoft基本类库(MFC)框架结构，它们提供了一系列方便的方法来执行断言。违反执行假定将会导致断言

激活，但是随后的程序行为将没有定义。在运行时刻改变外部条件是非常重要的，例如不同的输入参数集、文件系统或注册表的改变，或对API或其他组件的不同响应，在调试期间将不会出现类型错误；否则，在发布联编时将出现意外的灾难性故障。在检查出错条件时，如果你不能确定，应该选择下面两种条件分类中的一个。

2. 预期条件。正常情况下，这种错误预期出现的概率非常大。通常，这种错误是由于不正确或不一致的用户输入所致。

在这些情况下，尽管使用COM+错误处理机制是可行的，甚至是合法的，但是通常最好设计采用一个接口来与外部参数进行通信。（在编程语言领域，这个方法类似于返回状态码或一个布尔值，而不是引起一个异常错误。）然而，采用COM+接口方法的返回值来完成上述功能一般是不可行的，因为返回值需要传递的是真正异常故障的情况。同时，如果你决定使用COM+错误与这种情况进行交流，必须准备错误上下文信息，这正如一个对象实现是基于接口而不是基于错误来提供这类信息一样。这是不选择COM+错误作为这种类型的错误条件的一个原因：因为这种条件是可预计的，调用者一般丢弃上下文信息而只查找故障代码。简单的输出参数一般更容易实现并且更为有效。

3. 异常。正常情况下这种错误一般不会发生，它一般发生在调用程序的组件编程模型中或组件的运行时刻环境中。内存溢出是一种典型的异常错误。

编程环境通常通过自动发出环境设置异常来辅助检测这种情形的发生。如果合适地配置，在内存分配故障时，Visual C++ 会报告std::bad_alloc[⊖]，同时MFC也会报告CMemoryException。来自次要对象的COM+错误通常属于这种类型。然而，构造和返回COM+错误和错误上下文信息能够很好地处理这种情况，本章将讲述这方面的内容。

既然你不得不参与编程环境所提供的带外异常处理机制，并且控制接口之间的异常行为，从而满足COM+错误编程模型的需要。使用任何一种错误处理的方法将会提高代码的可维护性。实现假定相对独立，而将预期条件与异常条件进行分类却是一件困难的事情。如果不能确定的话，一般情况下，创建异常条件比较安全，尽管COM+上下文信息不能立即提供使用，但它却能终止调用者的执行。

1.2.1 结果代码

COM+接口中的每个非局部方法必须返回一个HRESULT类型的返回代码。其中一个原因就是提供COM+中间件一种自身状态报告机制。例如，当远程主机是不可访问时，或对象被一个不正确的线程所激活，或对象本身与它的客户失去连接，一个对象能够返回一个错误代码。

对于中间件改变对象返回的返回代码值也是可能的，但是只有返回成功代码时才能改变返回代码值。在这种情况下，中间件执行正确的清除动作，清除输出参数和释放返回的对象引用。在装入proxy/stub DLL错误，或者参数排列错误时，将会出现这样的情况；当事务组件调用SetComplete，由于某种原因事务提交失败时，也会导致CONTEXT_E_ABORTED错误。

允许COM+接口方法调用涉及的代理各自设置返回代码，将会导致返回代码值的不一致性。

⊖本书作者在http://www.codeguru.com/cpp_mfc/FDIS_new.shtml有一篇文章“An FDIS compliant Operator'new'(与FDIS兼容的操作符new)”，它说明了如何实现这一配置。

为了减少这个问题的发生，需要为代码的结构和使用建立约束条件。

对于32位系统，HRESULT的结构如图1-1所示。

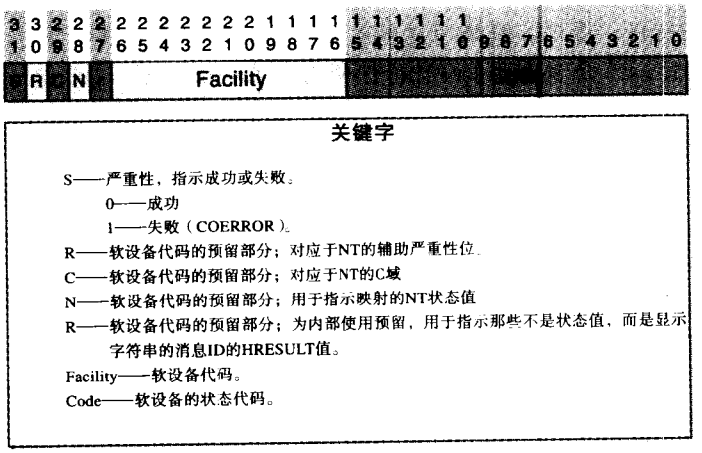


图1-1　HRESULT的一种结构

为接口定义的HRESULT必须在软设备FACILITY_ITF范围之内。中间件从不会使用这个软设备的代码，从而避免代码值冲突。

返回代码必须相对能够返回的接口来定义，而不是相对于某个接口实现。这就意味着，当定义和发布一个接口时，必须提供所有方法所返回代码的详细列表。系统错误代码则不包含在内，不管它们是否在接口中存在，它们都会返回。这有助于中间件完成其替换功能。当然，你可以为某个接口定义某种错误代码，而对另一个接口定义另外的错误代码。例如，你可以声明Ifoo能返回代码x、y和z，以及从IBar返回的任何代码。这仍需要考虑返回代码值的冲突问题。毕竟，你必须考虑x、y和z,以及从IBar返回代码的冲突问题，否则，你必须为调用者创建多义返回代码，避免他来确定到底发生了哪种情况。

为某个接口定义某种错误代码，而对另一个接口定义另外的错误代码是非常有效的，使得代码实现变得相对简单。在上面示例的上下文中，Ifoo的实现代码只要简单地把从Ibar调用收到的所有错误传播出去，而没有必要进行任何检查。现在你所看到的是使用接口返回代码，而不是与代码实现相关的返回代码，尽管Ibar的代码实现可能会发生改变，Ifoo的简单错误传播代码实现策略仍是可用的，并能确保不会发生冲突现象。使返回代码成为接口定义的一部分，可以使对象的代码实现独立于它们所依赖接口实现的改变，正如接口本身的不变性可以防止客户实现代码的改变一样。

在别的情况下，例如随着时间的推移，在代码实现中开始采用新的接口，或者并非所有的Ibar定义的返回代码，能够允许从Ifoo返回。对象实现在返回结果代码之前，必须仔细检查从新接口或IBar返回的每个代码。如果结果代码最初没有被Ifoo所知，并且这个代码不是一个系统结果代码，在程序实现中必须把系统结果代码E_UNEXPECTED返回它的调用者。尽管这个结果代码不是非常有价值的信息，至少它会防止由于结果代码冲突而导致多义性问题的发生。如果需要返回新的结果代码，必须定义一个新的接口，这正如一个已经发布的COM+接口的改变一样。

最后，注意到返回的结果代码只要不是错误代码（SUCCEEDED宏一般等价于“真”），就是合法的，并且是有利的事情。这样做对于某些调用程序可能会产生问题。例如，Microsoft Visual Basic编程环境和许多脚本环境，并不提供对结果代码的访问方法，除非这些结果代码是一个错误代码。你可以根据需要，返回传递重要信息的成功代码。

1.2.2 错误上下文

当返回的结果代码是一个错误代码时，也就是说，通过设置最高位为1，将结果代码的severity(严重性)字段设置为Fail(失败)，对象的代码实现就可以传递更多关于上述错误的上下文信息。这些信息包含如下几方面的内容：

- 关于该错误的文字描述信息。
- 用于定义该错误的接口GUID。
- 用来描述该错误的帮助文件的路径。这些信息对于提示组件编程模型被错误使用尤其有用。帮助文件可以帮助说明编程模型。
- 在上述帮助文件中的帮助上下文标识符。
- 产生该错误的类ProgID。

对于提供上下文信息用于错误发布的对象的代码实现Ifoo来说，它必须完成如下工作：

1) 实现接口IsupportErrorInfo。使用Ifoo的接口ID作为该方法的唯一参数，调用Interface SupportsErrorInfo 方法，返回S_OK。

2) 在从Ifoo的任何方法返回之前，调用API函数CreateErrorInfo,通过IcreateErrorInfo接口为返回错误上下文对象设置合适的值。

3) 在完成上述步骤之后，但在从Ifoo的接口方法中返回之前，调用SetErrorInfo并把一个IerrorInfo接口指针传递给错误上下文对象。这个对象现在可以被系统引用，并且程序实现在返回之前能够释放错误上下文对象。

为了能够检索错误的上下文信息，调用程序必须在从IFoo接收到一个错误代码之后，执行如下的步骤：

1) 检索对象的ISupportErrorInfo，如果对象支持该接口则进行处理。使用IFoo的接口ID作为该方法的唯一参数调用InterfaceSupportsErrorInfo。只有在对象返回S_OK时才进行处理。

2) 调用API函数GetErrorInfo，通过它的IerrorInfo接口，从返回错误上下文对象中读取上下文信息。注意，此时系统将释放错误上下文对象，并且该线程的扩展错误信息随后被清除。

3) 在完成之后，释放错误上下文对象。

此处你所看到的是对错误上下文非常底层通信的描述。你的编程环境可能把这些具体的细节隐藏了起来。在以后的章节中我们将讨论一些较为高层的机制。

错误上下文对象传输

当对GetErrorInfo和SetErrorInfo机制进行深入思考时，开发者有时会提出这样的疑问：COM+库的底层代码是如何实现的呢？如同C库非常古老的errno状态变量一样，错误上下文对象由每个线程进行维护。对此，人们会猜测使用线程局部存储（TLS）来实现，然而，

由于COM+接口调用能够跨越线程、进程边界和主机边界，因此一定还有比简单的TLS更为复杂的机制。

事实上，为了在套间和主机之间传递错误上下文对象的引用信息，COM+库求助于通道钩子机制⊖，它允许把带外有效载荷与对象的RPC（ORPC）调用一同传输。传送给SetErrorInfo的接口指针，便透明地排列到调用程序的上下文中。

如果这种访问涉及到跨越套间，甚至通过代理跨越主机，对错误上下文对象属性访问的开销将非常大。但是把对这个对象的访问拆分为GetErrorInfo，将不会导致性能的下降，并不会引起进一步的ORPC调用。上下文对象传输难题的最后一个事项，在于错误上下文对象的IMarshal代码实现中：这个对象使用一个集合从CreateErrorInfo函数返回，并且调用程序的所有上下文属性访问都是暂时的。

1.3 Visual Basic环境需要考虑的问题

Visual Basic COM+错误集成的最大特点就是它的不可见性：当执行Visual Basic COM+组件时，你甚至不知道存在COM+接口边界。ISupportErrorInfo、IErrorInfo和SetErrorInfo等，都由Visual Basic 运行时间所执行，它们对于程序员是不可见的。Visual Basic的异常处理编程模型基于全局作用域的Err对象。这个模型是带外的，开发者可以通过调用Err对象的Raise方法，激活一个堆栈展开异常错误，并且提供一个与IerrorInfo接口属性兼容的参数。一旦它们达到接口边界，Visual Basic运行时间就透明地把这个异常转化为COM+错误上下文对象。

捕获COM+错误同样是容易的，它与Visual Basic Sub程序、函数或属性的错误处理方法相同：on error指示用来在当前的程序中委派一段处理程序代码,完全删除这个错误，或导致它向调用程序传送。处理程序代码，不管是否内嵌的，能够检查Err对象来确定错误的属性。在缺省情况下，Visual Basic对象的方法或属性实现所产生的错误，作为实现COM+错误将继续产生，这些对程序员不会产生任何影响。

在所有情况下，这种无缝集成的确具有一个缺点：Visual Basic开发者很难理解发布接口中结果代码保持不变的重要性。尽管文档确实鼓励开发者编写排列良好的代码（Err.Raise中的number变元、由vbObjectError常数和代码相加得出，只要代码是一个16位的值，它们的累加和就在FACILITY_ITF之内），同样的文档中对结果代码规则却没有任何要求。当然，Visual Basic的代码自动传播功能也不鼓励开发者把调用程序暴露给返回代码。最低要求Visual Basic错误集成模型是最方便使用的，并且是最有效的。

对于Visual Basic项目，我建议对能够产生错误数字的全局函数与只有一个数字参数的描述字符串相加。这个参数便会像资源索引字符串一样，具有代码的功能。这样做带来的好处是：

- 在代码中把结果代码和描述字符串进行联编是隐式的，并限制在一个地方。这样，就会减少潜在的描述符与代码之间的不匹配现象，对于结果代码在多处使用的情况尤其如此。
- 因为描述字符串位于资源中，它们可以非常容易地实现本地化。

⊖ 请参阅Microsoft Systems Journal(MSJ)1998年1月刊(http://msdn.microsoft.com/library/periodic/period98/activex0198.htm)中的Don Box的“ActiveX/COM Q&A”专栏，它研究了少有论述的通道钩子特性。

这样的函数一般具有如下的形式：

```
' This sub uses a resource file to retrieve strings indexed by the
' error number you are raising..
Public Sub RaiseError(lngErrorNumber As Long,strSource As String)
   ' Raise an error back to the client
   Err.Raise vbObjectError +lngErrorNumber,strSource,_
      LoadResString(lngErrorNum)
End Sub
```

如果需要的话，可以将该函数进一步升级，允许把出错代码与帮助文件引用进行联编。最后每个类模块能包含一个私有的子程序，它的任务是在将出错代码传递给RaiseError之前，代替该模块的源标识符。这样，类模块内所有的出错代码，就可以使用单个代码参数调用这个本地私有子程序。

1.4 Visual C++ 环境需要考虑的问题

大体上讲，如果需要产生和拦截COM+错误，C++程序员便不得不把手头的工作停下来。ATL对象向导提供了一个选项，用于在新对象中为IsupportErrorInfo包含最初的支持。ATL还提供了一些帮助函数，它们具有构造出错上下文对象的能力。在调用端，由#import产生的灵巧指针具有将COM+出错转化为C++异常的能力（关于此功能的有关信息，可参见第2章）。但是，这些功能需要环境支持。并且C++程序员需要小心防止将环境的本地异常（这是SEH类型）传播到其他接口。具有讽刺意味的是，甚至对于COM+友好的#import机制产生的异常也必须避免，也就是说，当在一个方法实现中完成的所有工作就是通过灵巧指针调用其他接口时，一般需要获取句柄。但是，所有的并未丢失，程序设计者可以通过COM+错误集成策略的一致应用，重新获得较高的生产率。揭示这个策略是以下章节的主要内容。

许多开发者认为在如下情况下可以将C++对象的本地异常跨越接口边界：

- 该接口被声明为局部的，或者该对象需要驻留在调用程序的上下文中。
- 调用程序采用C++实现，或由于项目约定，或者由于接口的IDL（接口描述语言）中包含的功能声明，使C++是惟一可能的客户实现语言。
- 调用程序规定C++异常通过接口实现来发布。

尽管这些意见来自实际的要求，并且可能有助于简化C++代码实现，我基本上对具有这种行为的对象能被COM+对象所调用持否认态度。事实上，COM+是一个二进制兼容标准，SEH异常的实现违背二进制兼容的某个规则。并且记住：你不仅需要实现C++客户程序，而且有可能实现一个C++编译器版本。正是由于C++标准是一个源代码标准，而不是二进制标准；C++编译器可以自由地以任何方式实现C++异常处理，SEH仅是其中的一个选择，并且是非常流行的。但是，当两个编译器都选择SEH作为C++异常处理方法时，这两种代码实现可能不会兼容，事实上，也不会兼容。

我建议当C++异常跨越接口边界时，宁可使用C++实现类，也不要产生COM+对象，这个类可以被调用程序通过协议类所访问（这是一个非常有用的设计技术，有时被称为Cheshire Cat模

式）。同时认为，此时使用IDL和Microsoft IDL (MIDL)编译器产生协议类定义没有任何好处，因为IDL主要用于产生与语言无关的接口。简单地把接口直接定义为抽象C++类会更有意义。这样同时可以访问C++所特有的功能，这是采用IDL所不具备的，例如，在方法声明中使用的异常处理说明。异常处理说明一般非常有用（如果编译器实现的话），因为C++异常处理模型非常丰富，并且在方法声明中使用时，它提供了更多的功能用于提高代码的可维护性。

C++异常处理说明

有趣的是，C++异常处理说明功能可获得COM+结果代码规则的一个目标：它把所允许的异常处理类型与类定义进行联编，这样类的接口正好与它的代码实现相对应。因为可允许的类型可以在接口约定中显式地说明，相容⊖的编译器在违背这个约定时，会通过调用unexpected _handler函数，强迫这个约定的执行。也就是说，对于具有异常处理说明的函数产生异常错误时，它并不包含这个异常错误类型。这样的处理程序一般会中断处理过程，但是如果该异常类型并不在当前函数的规格说明中，并且这个函数确实包含std::bad_exception类型，那么所产生的异常错误将被std::bad_exception类型的对象所代替，并且从外部函数开始搜索这个类型的异常处理函数。你可以通过调用set_unexpected函数来代替自己的unexpected_handler函数。

要想更深入地研究C++异常处理模型，可参阅Microsoft Developer Network (MSDN)中Robert Schmidt的系列丛书“C和C++异常处理”，其网页地址为：http://msdn.microsoft.com/library/welcome/dsmsdn /deep051099.htm。

1.5 C++错误模型综合处理方法

在本章的最后，将讨论如同在Visual Basic中那样，在C++中产生COM+错误的相关技术。为了使这些技术最终能够成功应用，建议读者能够在自己的C++代码实现中坚持使用这些技术，一旦读者熟悉它的机理，相关的处理就变得较为容易了。

这个解决方案包含三部分：错误报告函数、异常类和异常处理宏。读者所看到的源代码是从COM STL Bridge的代码实现中截取的（参见第10章）。本书的配套光盘中包含CSB的源代码文件。

1.5.1 结果代码的框架

正如把来自不同技术例如远程过程调用（remote procedure call，RPC）和消息队列的错误代码分为各自的软设备，系统的总体结构就会更为清晰一样，单个项目或多个项目也会从错误代码的分割中受益。这样的策略允许编程者把错误代码头拆分为独立的文件，从而避免开发者在完成某个项目的不同部分时需要协商错误代码的分配范围。我把这些项目层的软设备称为子软设备。把这样的结构叠加到正常的结果代码结构中，产生了如图1-2所示的定义。

⊖ Microsoft Visual C++ 6.0是不完全兼容的：它允许异常说明语法，但是不实施它。这意味着，你可以在代码中包含说明，但是必须期待编译器未来版本的实施。

未声明位的意图是推迟对子软设备和代码范围的决定，直到项目能够决定哪个实体更有可能需要该值。将低字的高位置为0，允许子软设备和代码部分能够直接映射为描述字符串的资源ID。

除了为一个给定的错误代码提供可读的描述符和其他上下文信息外，还需要能够让用户通过符号的形式来访问错误代码。这对于客户需要测试所定义的一个或多个特定错误时尤为有用。下面的宏正是为了这个目的：

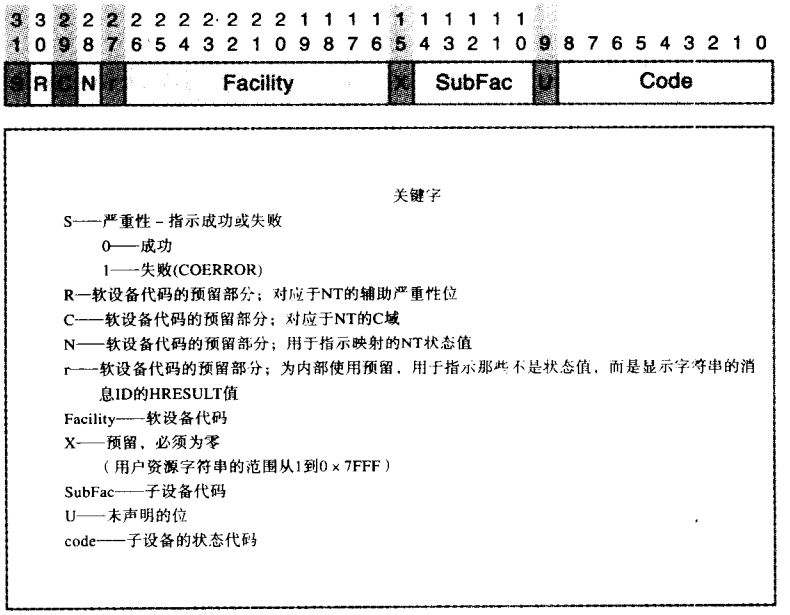


图1-2 结果代码的结构

```
//Shift-expanded HRESULT #defines
#define CSB_SEVERITY_SUCCESS          0x00000000
#define CSB_SEVERITY_ERROR            0x80000000
#define CSB_FACILITY_ITF              0x00040000

//CSB substitute for MAKE_HRESULT that works with midl
#define CSB_HRESULT(sev,fac,code)       (sev |fac |code)
```

通过在项目IDL文件的库中插入typedef，错误代码MYSUBFACILITY_E_FOO（定义为15位）能够用来产生与语言无关的符号常量。

```
//Errors
typedef enum t_MYPROJECT_ERRORS
{
   ERC_MYSUBFACILITY_E_FOO =CSB_HRESULT(CSB_SEVERITY_ERROR,
                            CSB_FACILITY_ITF,MYSUBFACILITY_E_FOO),

   //append more codes here
   ⋮

}t_MYPROJECT_ERRORS;
```

1.5.2 报告函数

错误报告函数是一个函数，它能够从给定的参数中构造出错误代码和错误上下文信息。通常需要下列三种错误报告函数，每种适应于不同的情况。

1) 报告项目定义的一般错误：该函数在缺省情况下从调用函数的资源句柄开始，从多个位置搜索描述字符串资源。接着，在函数的实现中处理库的资源句柄，最后处理主进程的资源句柄。这使项目能够把多个组件使用的资源字符串进行集中。

2) 报告系统定义的错误，例如E_NOTIMPL ：函数使用FormatMessage API 来产生错误描述符。

3) 报告自己的对象、第三方软件对象或系统对象中的接口方法出现的错误。该函数调查第二个对象是否提供了错误上下文；如果已经提供了错误上下文，函数将广播这个上下文；否则该函数采用它自己的上下文来代替。这个函数释放第二个组件，并报告是否需要代替它自己的错误上下文。该函数假定接口已经定义自己的错误代码集，并且确保你的对象能够完成它通过ISupportErrorInfo 所做的承诺，而不必考虑第二个组件的行为。

以上调用各自具有如下的原型：

```
//Reporting your own errors
HRESULT CSBReportError(REFCLSID rtClsid,UINT nID,REFIID rtIid,
                       HINSTANCE hInst =_Module.GetResourceInstance());
                          //by default,caller's resource handle
                          //is tried first

//Reporting system errors
HRESULT CSBReportError(REFCLSID rtClsid,HRESULT hRes);

//Reporting errors from other (possibly your own)components
HRESULT CSBReportError(REFCLSID rtLocalClsid,HRESULT hRes,
                       REFIID rtSecondaryIid,
                       IUnknown&riSecondaryComponent,
                       bool bReleaseSecondary =false,
                       bool*pbInfoProvidedBySecondary =NULL);
```

这些代码能够采用如下的代码来实现。注意这些代码实现中涉及到异常类中的重载静态函数GetMessage，下面很快就会提到这一点。

```
HRESULT CSBReportError(REFCLSID rtClsid,UINT nID,REFIID rtIid,
                  HINSTANCE hInst /*=_Module.GetResourceInstance()*/)
{
   //Verify validity of CSB code (See Subfacilities.h)
   _ASSERTE (
      !(nID &0x8200)
      //bits 15 and 9 must not be used (9 can be used later)
&&nID &0x7C00
      //the subfacility code must not be zero
&&nID &0x01FF);
```

```
      //the actual code must not be zero

//Now let ATL set up the error context
return AtlReportError(rtClsid,
                      CCSBStdException::GetMessage(nID,hInst),rtIid,
                     MAKE_HRESULT(3,FACILITY_ITF,nID));
}

HRESULT CSBReportError(REFCLSID rtClsid,HRESULT hRes)
{
   //Must be a real error,not an advisory
   _ASSERTE (HRESULT_SEVERITY(hRes)==SEVERITY_ERROR);

   //ATL will set up the error context
   AtlReportError(rtClsid,CCSBStdException::GetMessage(hRes),
                  GUID_NULL,hRes);

   return hRes;
}

HRESULT CSBReportError(REFCLSID rtLocalClsid,HRESULT hRes,
                       REFIID rtSecondaryIid,
                       IUnknown&riSecondaryComponent,
                       bool bReleaseSecondary /*=false*/,
                       bool*pbInfoProvidedBySecondary /*=NULL*/)
{
   //Must be a real error,not an advisory
   _ASSERTE (HRESULT_SEVERITY(hRes)==SEVERITY_ERROR);

   //Release given interface pointer upon return,if requested
   CComPtr<IUnknown>ciAutoRelease;

   if (bReleaseSecondary)
      ciAutoRelease.Attach(&riSecondaryComponent);

   //Determine whether the secondary component created an error object
   CComQIPtr<ISupportErrorInfo,&IID_ISupportErrorInfo>
      ciSupportErrorInfo(&riSecondaryComponent);
   if (ciSupportErrorInfo.p
   &&ciSupportErrorInfo->InterfaceSupportsErrorInfo(rtSecondaryIid)
     ==S_OK)
   {
     if (pbInfoProvidedBySecondary)
     *pbInfoProvidedBySecondary =true;

      return hRes;
   }
```