反省的对象中间件*)
——高可伸缩的反省ORB

Reflective Object-Oriented Middleware——High Scalable Reflective ORB

王敏毅　赵　东　姚绍文　周明天
（电子科技大学计算机学院微机所　成都610054）

Abstract　Today's object-oriented middlewares, especially CORBA, are confronted with challenges from progressive applications. Many black-box based ORBs are not flexible enough to adapt themselves quickly to satisfy lots of new requirements. This paper applies reflective computation technology to the architecture of ORBs, proposing a kind of reflection on message transport protocols, extending the meta-level functions for object activation, and implementing the structural and behavioral reflection of IDL interfaces with help of the IDL compiler. By introducing "slots", meta-components can be dynamically selected and loaded into run-time kernel, that improves system's scalability effectively and partly solves problem of inconsistency existing in distributed reflective system. Based on reflection, we implement some critical functions aiming to enterprise applications, including pluggable protocols framework, integration of message-oriented middleware, and component-based applications. This paper ends with the conclusion that reflection is very useful in building flexible and scalable object-oriented middleware.

Keywords　Reflection, CORBA, Middleware, Meta-computing, Scalability

1　引言

以CORBA为代表的对象中间件在企业应用中越来越发挥着重要的作用，但日趋增多的应用领域如：电子商务、多媒体与实时系统、移动计算等，及新CORBA标准都对中间件开发者提出了新的挑战。只有协调好统一的体系结构、广泛而灵活的适应性及优化的性能这三方面问题，产品才能快速适应各种应用领域的需求，同时保持良好的性能。

但当前多数CORBA产品或系统并不符合上述要求，究其本质，是因为设计上受传统软件工程思想的影响：一般的OO建模在处理复杂问题时通过抽象将实现细节对用户隐藏，这就是黑盒方法。基于黑盒的ORB系统面临不断增加的新需求时调整比较困难，例如：改变核心协议栈、增加传输QoS管理、数据包压缩、加密或过滤等等。如果需要一个移动的系统能检测到环境的剧烈变化，并及时自动调整和适应，现有的体系结构根本不能胜任。

反省计算以白盒模型和开放实现(Open Implementation)为基础[3]，是解决系统灵活性和可重构性的经典方法之一，过去一直限于OO语言和操作系统领域[4,5]。最近几年，国际上许多学者开始重视它在分布系统，尤其是中间件领域中的研究[1,8,9]，并用于体系结构的设计，但由于起步不久，多处于模型阶段。

值得注意的是，OMG已经开始意识到前面问题的严重性，因此CORBA标准中也提供了部分解决手段，如：DII/DSI、构件模型、XMI、MOF、拦截等等。这些技术本质上是对传统黑盒模型的一些扩展，或仅仅是比较粗糙的反省，距离系统动态可重构性、灵活性和自适应性等要求还有欠缺。我们认为：简单的扩展并不适合成为新一代CORBA产品体系结构的主流。

本文将反省技术用于ORB系统设计中，提出了反省的ORB体系结构，并很好地解决了以下关键问题：传输协议与对象激活的行为反省定义或扩展、IDL接口反省、元构件的动态加载和选择、可插卸的协议构件、集成消息中间件、构件化应用等。可以预见，这种具有高度可伸缩性和灵活性的系统可以广泛地适应各种应用领域的需求。

2　相关研究

基于可插卸协议框架的TAO[6]：通过该框架解决支持通信层QoS的协议问题。与本文类似的是：该系统也支持多协议的动态插卸，但最大不同之处在于：TAO使用若干设计模式解决框架中的关键问题，如：静态和动态协议增加、消息和传输分层、连接建立与接受等；而本文采用统一的反省模型，通过定义元对象与基础级槽的开放绑定，以简单的方式使系统在更多方

*)本文工作得到国防预研基金DJ8.1.3的支持。

面可以重构.因此更加灵活、伸缩性也更好。

dynamicTAO[5]是一个反省的ORB，在TAO的基础上修改而成，其中通过一组构件配置器实体实现内部结构的具体化(Reifying)，可用来进行动态的定制。一个构件配置器维持了某个构件和其他系统构件的依赖，其中包括策略附接的钩子，即安装点(mount point)，在这里指定的策略实现可以挂在ORB上。构件的实现是动态可装载的库，可以在运行时连接入ORB的进程空间。dynamicTAO的主要不足之处是：系统反省范围仅涉及诸如策略、打包/拆包和并发等常规内容，对分布特征反省不够；基于配置器模式的反省还不能将基础级和元级清晰地分开；对一致性的问题的解决方法比较复杂；远程操纵配置器的方法并不标准，缺乏通用性。

Quarterware[2]是一个可定制的中间件体系，它抽象了基本中间件功能，容许针对特定的应用进行扩展。利用Quarterware可以演化出CORBA、RMI和MPI的核心设施实现。Quarterware对中间件功能的抽象相当于反省的雏形，没有模型化，主要是引发、打包/拆包和分派。因此，在运行中对系统调整仍然比较困难。

从以上情况不难看出，现有的研究还缺乏对ORB核心足够细致而针对性的反省，基础级ORB程序如何使用元级对象/构件也没有统一的模型，本文将对这些问题提出一些模型或解决方案。

3 体系结构

反省ORB的体系结构如图1所示，总体上与一般的ORB类似，但为了反省，设计上做了以下特殊的处理：

消息层与传输层的分离：GIOP的语义由GIOP驱动器(消息层)处理，其中使用流和连接器两种元对象，使用前需先进行动态绑定。目前仅在元级实现了符合CDR规范的流，而连接器由传输层的其它元对象产生。

协议槽：针对某种协议有若干互相依赖的元对象，这些元对象成组插在核心协议槽中，可以静态配置或动态插卸。由于槽的存在，同一类的元对象可以并存在核心中，只在用的时候才动态绑定。

存根与骨架：不同于一般ORB，本系统IDL编译器可以根据IDL定义的接口自动生成相关的元信息及元对象，典型的有：服务对象中维护的接口元信息；方法分派时可动态产生引发的元信息；用于存根方法参数打包和析取结果的元对象；用于持久对象状态的保存和恢复的元对象等等。

元构件库：即核心协议构件库和应用构件库。前者的元构件被核心加载后可登记到槽中；而后者提供了服务对象激活的元构件及服务对象本身的实现，新的应用构件可以由用户单独生成，其中可能借助IDL编译器生成代码框架。应用构件根据需要可以被运行核心动态加载，因此大大提高了系统的可伸缩性。

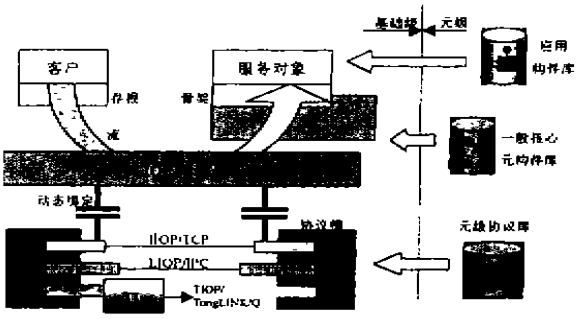

图1 反省ORB的体系结构

4 关键问题

4.1 ORB的行为与结构反省

本文首先解决的是反省ORB中需要将什么地方做白盒实现，或者说定义哪些元对象协议MOP(Meta-Object Protocol)。定义MOP的原则是：一切元级的修改对基础级程序都应当透明。对基础级系统的反省主要有两种类型[1,3]：结构反省和行为反省，前者对基础程序的结构化信息、状态空间等进行反省，元级信息包括：对象所属的类、派生结构、当前运行方法、消息传输协议等；后者指元对象可以检测、访问基础应用中的一些涉及行为关键语义的事件，或对行为进行修改，而这一切对基础级应用透明。典型的行为有：方法执行、消息分派、发送和接收消息、对象激活等等。本文定义或扩展以下MOPs：

4.1.1 消息传输协议 对协议行为的反省就是从基础级系统中抽象出以下元级行为：

倾听(Listen)：在某个端点(EndPoint)上倾听请求。

创建连接(CreateConnection)：在客户端是主动创建连接；而服务器端是接受(Accept)请求后被动创建连接。

创建映像(CreateProfile)：服务器对象根据通信设施产生相应的IOP映像；或客户端借助工厂创建对应标记映像的协议映像。

发送(Send)和接收(Receive)：使用连接发送和接收消息。

基于上述行为反省的MOPs在分布式系统的反省研究[1-9]中虽曾部分被提出，但本文首次应用于ORB中。实现这些MOPs的元对象存在于元构件中，结合下文将讨论到的元构件加载和选择技术，就构成了可插卸协议框架的基础。

4.1.2 对象激活 对象适配器根据对象键或接

口类型ID激活一个服务对象可以相当简单，即定位对象实例；也可以比较复杂，手段包括：动态创建对象实例并激活、从构件中加载服务对象并激活、从存储器中恢复对象的状态并激活、甚至从其它服务器上将服务对象迁移到本地激活等。

本系统在对象适配器中，对激活行为进行反省，即在元级做灵活的实现。适配器根据不同的策略绑定不同的元对象。对同样的服务对象，只要登记在不同策略的适配器中，就被赋予不同的元级行为。

4.1.3 *远程方法引发* 对客户端对象引用的引发(Invoke)行为进行反省，借助IDL编译器生成与接口方法相关的元对象Invoker，其元级实现了远程方法引发前参数的打包(marshalling)和引发后的结果拆包(unmarshalling)。对象引用(基础级)和Invoker对象(元级)在运行时动态绑定，就可以做到同样的基础级方法满足不同的应用方法需求，基础级程序根本不需要改动。图2进一步示意了以上对行为的反省。

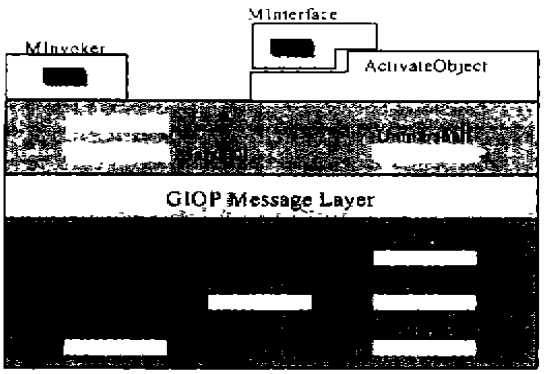

图2 ORB中对行为的反省

4.1.4 *IDL接口* IDL语言本身用于描述服务对象接口的元信息，因此很自然可以反省。本文借助编译器实现了涉及接口结构、方法引发等信息或行为的反省。

在骨架对象或代理对象(存根)中可以很容易地增加一些涉及接口元信息的属性和方法，对象方法中也可增加结构反省的代码，这样处理后，用户可以很容易地访问到接口的元信息，如：方法(包括名字、参数等)、接口名、派生层次等。

另外，IDL编译器还可以生成一些元对象或代码框架，如：服务对象工厂、持久对象的流化方法等。核心激活对象时，可以使用这些元级对象或方法。

IDL接口描述中涉及：接口、属性、操作，映射到OO语言中就是：类、成员(实际是属性访问方法)和方法。因此，本文定义了类似Iguana语言[8]的选择反省机制，即在IDL描述中扩展对上述元素的反省。相应地，修改编译器支持MOP选择声明的语法，这样就可以做到接口的行为反省。

对接口的行为反省可以不改变接口定义而修改操作或状态访问的语义，如：加解密、向交易服务OTS登记资源、状态并发访问控制等。其机制比在操作中定义上下文(context)更灵活。

但为了不影响兼容性，本文对IDL的扩展只在特殊注释中实现。格式为：

```
//{Reify==>}(MOP name[,MOP name]*)
```

以下是几种IDL接口反省声明的例子。

```
interface IHello World //{Reify==>}Interface
{
    attribute long l; //without reflection
    //this attribute is reflective
    attribute float f; //{Reify==>}StateAccess
    void op1(); //without reflection
    long op2(in string s,inout double d); //{Reify==>}
    MethodAccess
}
```

其中，MOP声明不再被编译器解释为注释。Interface表示接口的元信息需要被反省；StateAccess和MethodAccess和Iguana语言中定义的语义基本一致，不同之处在于作用的目标不同：本文中是接口，而Iguana中是对象。

此外，还有一些常规的MOPs：方法分派和引发、GIOP消息打包与拆包等，也在本文系统中得到了实现，这里不再介绍。

4.2 元对象的选择

可重构的反省系统必须能在运行中改变元级行为，因此本文引入抽象工厂(Abstract Factory)模式[6]和槽，对元级对象不是做简单的替换，而是允许若干元级对象在槽中并存，通过某种选择匹配依据找到合适的元对象，与基础级对象进行动态绑定；或借助合适的元级工厂设施(类似于文[9]中的Composer)创建相应的元对象。第二种方式适合那些必须临时创建的元对象，如连接器。

某种匹配规则对元对象的选择是至关重要的，但规则对不同MOPs是不一样的。例如对所有协议元对象而言，都可以采用CORBA规范中定义的标记(Tag)来匹配；激活服务对象的元对象则可以依据适配器的策略选择。至于消息对流的打包和拆包可以根据需要选取不同的元对象，如：CDR、压缩、加密(可以选用不同的加密算法)，并与基础级对象(如：GIOP驱动器)静态绑定，这样就可以实现不同的流。

本系统中基础级对象和元对象均采用动态绑定的方法进行关联，在执行完功能后，这种关联可以解除。比较典型的如：GIOP驱动器(基础级)和连接器(元级)之间在发送和接收GIOP请求/应答时的绑定。

与一般反省系统中简单替换元构件的做法[5,7]相比，上述元对象并存、选择匹配、动态创建与绑定等机制可以比较好地解决分布系统反省产生的一致性问

题。

4.3 元构件的配置与动态加载

为了便系统具有高可伸缩性、高可用性，本文同时采用了元构件静态配置与动态加载的技术，其基础就是上述槽和元对象选择的机制。

静态配置适用于默认使用的元构件，如：IIOP 协议、CDR 流等，适合于最小化的反省ORB，配置这些元构件是互操作的基础。

而构件的动态加载可以使系统在不停止运行的前提下动态伸缩，因此可以获得7x24的高可用性。这种模式下，元构件和基础系统彻底分离，甚至可由另外的用户开发，只是在运行中元构件被动态插入到基础系统中。

5 实现

本文反省 ORB 用 C++ 实现，为了保证互操作性，系统将 IIOP 协议实现为基本的协议元构件，而 GIOP 消息的打包和拆包采用符合 CDR 格式的流，它们被静态配置在核心中；另外，应用 API 和常规 ORB 基本一样，从而使应用在不同的 ORB 之间移植性更好。

但本系统的设计策略更强调高可伸缩性，因此使用反省确保程序基础级与元级逻辑的分离，使用构件动态加载技术达到物理的分离。实现上与常规 ORB 最大的不同包括以下几点：

1）可插卸协议框架：核心可以静态或动态地增加对新协议的支持。新的协议以构件形式存在，其中包括一组元对象：映像工厂、标记映像、端点、倾听器、连接器等，实现了对协议反省后的元级功能。元对象被加载后插入到核心协议槽中，就可以起作用了。这种体系，以比较单一的方式大大提高了系统的可扩展性和可伸缩性。而且，进一步实现针对实时系统的特殊协议或 QoS 管理功能也会相对比较容易。

值得一提的是：本文利用上述框架首次将第一代面向消息的中间件产品（注：TongLINK/Q，东方通科技公司产品，在国内金融行业已得到广泛应用）与反省 ORB 集成。消息中间件的特长是借助消息队列高效而可靠地传递大消息或文件，对消息中间件的集成可以大大提高 ORB 在大型企业应用的性能。

另外，本地 IPC 通信的优化也作为一种特殊的协议元构件实现，与 TAO[6] 做法不同的是：优化方法仍然是借助传输协议反省的机制，并没有另外引入新的方法。

2）构件化应用：ORB 核心不仅可作为应用开发库提供，也可表现为运行系统，应用以构件模式开发，其中除包括服务对象外，还有一些供对象激活的元对象，应用构件能被运行核心动态加载，显然这样的模式是伸缩性及可用性非常好，系统可以运行在比较经济的规模，而且新服务对象的开发可以完全与核心脱离，不必再重新编译核心。

3）有状态对象：由于对象激活的反省，核心可以方便地支持多种激活模式，而且与服务对象关系很小，以此为基础本系统实现了有状态的会话对象、持久对象等功能，对适配器而言，设计上是统一的，有状态会话对象对构造面向关键任务的企业系统是很重要的，但现有的许多其它 ORB 还不能从核心级支持。

结论与展望 本文基于反省技术实现了高可伸缩性的 ORB，其中对面向消息的传输协议做了细致的反省，并扩展了对象激活的反省实现；文中提出基于槽的元构件配置和动态加载模型，结合元对象工厂、选择和动态绑定的机制，可以有效提高系统的伸缩性，并部分解决分布系统反省的一致性问题；另外，借助 IDL 编译器可以对 IDL 描述的接口做比较细致的结构和行为反省。基于上述技术，本文的反省 ORB 支持可插卸协议、构件化应用、有状态会话对象等涉及企业级中间件应用的关键功能；同时，集成消息中间件也证明了使用反省技术对构造企业级分布系统是比较方便而灵活的。

目前，我们正基于反省机制实现 CORBA 安全和事务服务。今后可以深入的工作包括：用现有体系和 MOPs 实现对应 CORBA3功能的元构件；进一步优化反省的性能；解决系统功能或规模自适应收缩时的不一致问题；提出元对象合成的统一模型；处理元构件加载对运行核心的安全性问题等等。

参 考 文 献

1 Blair G S, Coulson G. The Case for Reflective Middleware. In: Proc. 3rd Cabernet Plenary Workshop. Rennes, France, April 1997

2 Singhai A, Sane A, Campbell R H. Quarterware for Middleware. In: the Proc. of ICDCS'98, IEEE, May 1998

3 Kiczales G, Lamping J, et al. Open Implementation Design Guidelines. In: Proc. of the 19th Intl Conf. on Software *Engineering* (ICSE), *Boston*, USA. ACM *Press*, May 1997

4 BRIOT Jean-Pierre. Concurrency and Distribution in Object-Oriented Programming, ACM Computing Surveys, Sep. 1998

5 Gowing B, Cahill V. Meta-object Protocols for C++: The Iguana Approach. Proceedings of Reflection'96, 1996

6 O'Ryan C, Kuhns F, Schmidt D C, et al. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. the I-FIP/ACM Middleware 2000 Conference, Pallisades, New York, April2000. 3～7

7 Roman M, Kon F, Campbell R H. Supporting Dynamic Reconfiguration in the Dynamic TAO Reflective ORB. ICDCS'99 Workshop on Middleware, Jan. 1999

8 Fabre J-C, Tanguy P. A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach. IEEE Transactions on Computers. Special Issue on Dependability of Computing Systems, 1998, 47(1): 78～95

9 Oliva A, Buzato L E. An Overview of MOLDS: A Meta-Object Library for Distributed Systems. In: Proc. of the Second Workshop on Distributed Systems (WoSiD'98), Sep. 1998