一种高效动态引发接口的研究与设计

A Research and Design on High-Effective Dynamic Invocation Interface

陈 松 孙 明 周明天

(电子科技大学计算机科学与工程学院 成都610054)

Abstract CORBA is the current popular interoperability technology, which provides a distribute computing platform for the communication of heterogeneous platforms and different language platforms. Dynamic invocation can construct the CORBA application flexibly and provide a synchronization and asynchronism method to invocate the object request. Conforming to the CORBA specification and technology of dynamic protocol component, this article discusses the basic theory of the dynamic invocation interface and design of the high effective dynamic invocation interface.

Keywords CORBA, Dynamic invocation interface, Distributed object

1 引言

分布计算技术是指在网络计算平台上开发、部署、管理和维护以资源共享和协同工作为主要应用目标的分布式应用系统。OMG 的 CORBA(Common Object Request Broker Achitecture)标准支持对象异构平台的互操作和可移植,使分布式应用的开发更加方便、快捷。

CORBA 规范[1]提出了静态和动态引发请求两种方式。静态请求引发指根据 IDL 文件生成存根(stub)文件,由存根完成请求的引发,是最常用的请求引发方式,也是效率最高的请求引发方式。但客户必须在引发请求前知道对象的类型,只能使用同步的方式引发请求,应用方式也存在一定的局限性。动态引发方式[2]指客户可以动态创建和引发对对象的请求,提供了一种灵活的请求引发方式,客户可以采用同步、异步的方式来引发对象请求,可以在运行态确定引发对象的类型和操作参数,也可以采用轮询的方式查询请求。

动态引发接口使 CORBA 应用的构建方式更加灵活,但在引发请求前需要额外完成从参数向 Any 的编码和从 Any 向 CDR 流的编码工作,会占用大量的系统时间,效率明显低于静态请求引发接口,这是制约动态引发接口性能的瓶颈。因此,为满足系统对性能的要求,在 DII 的设计时不得不考虑优化 ORB 内部通信设施的性能,提高 DII 请求引发的效率。

2 动态引发的应用模式

2.1 DII 几种典型的应用

客户端可以用静态的方式引发请求,也可以采用动态的方式引发请求。CORBA 规范中定义了动态引发接口——DII(Dynamic Invocation Interface),引发请求、传递上下文、捕获用户定义异常。DII 在实际中有着广泛的应用:

·客户在不知道对象引用的情况下,通过接口池查询通用的对象;

·客户在没有存根文件(stub)的情况下,手动地构造请求;

·成为联结 script(脚本)对象和 CORBA 对象的桥梁;

·ORB 中的拦截,采用 DII 的方式将请求的消息重新封装发送;

·传递上下文对象和传递上下文对象列表;

·除同步和单向的模式之外,提供一种新的应用模式:延迟引发。

2.2 DII 的缺点

动态引发接口在一定程度上提高了 CORBA 应用的灵活性,但需要用户手动地创建请求,完成请求的引发。因此,DII 存在自身的一些缺点:引发速度慢;编程复杂;客户程序容易造成服务器的崩溃。

3 一种高效的动态引发接口的设计

动态引发接口完成和存根相同的功能,请求的引发仍靠 ORB 通信设施来实现。因此,在动态引发的过程中,增强 ORB 通信设施的性能,高效地利用 ORB 内部的通信资源,是提高动态引发请求效率的关键所在。实践证明:ORB 内部使用线程池、连接池合理地利用了系统资源,在一定程度上提高了请求引发的效率;动态协议槽[3,4]提高了通信方式的灵活性,用户可以根据网络情况的不同,选择最合适的通信协议。

动态引发接口主要完成请求对象、NVList 对象、NamedValue 对象、上下文对象、UnknowUserException 对象几个类的设计以及如何利用 ORB 通信资源完成请求的引发。请求根据其它类中提供的参数和数据完成 GIOP 请求的封装和引发以及异常的处理。

3.1 ORB 内部通信设施的改进

(1)线程池 ORB 服务端核心可以维护一定数量的线程,这些线程在需要处理请求时被分配,处理完成后被释放,但并不结束;如果已有线程不够,还可以追加创建;线程的最大数目可以限制在一个峰值;核心还可以自动释放多余的空闲线程。这些机制可使核心总是保持比较优化的性能与开销折衷,而且伸缩性很好。

(2)连接池 可重用连接指连接创建开销比较大、所需时间比较长的连接,如 IIOP。当连接使用完以后不直接释放,而是由连接池进行管理,当下一个请求到来时,不需要再进行连接的创建,而是在连接池中分配一条空闲的连接,完成消息的发送与应答。连接池中的可重用连接都有一定的连接生命周

陈 松 硕士,主要研究领域为面向对象技术,Web 技术;孙 明 硕士,主要研究领域为面向对象技术,Web 技术等;周明天 教授,博士生导师,主要研究领域为计算机网络,分布对象技术,并行处理等。

期，超过生命周期的连接由连接池进行释放。

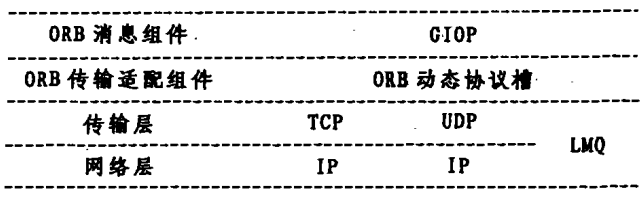

图1 可插卸协议框架

(3)*动态协议槽* 开发标准的可插卸协议框架[3,4]，允许灵活地配置和使用定制消息和传输协议，并且对用户来说是透明的。ORB 中定义了抽象协议层，从各类不同的网络通信协议抽象出了共同的服务接口，因此只需调用这些服务接口而不必关心底层的实现细节就可以直接进行通信。ORB 核心提供了协议槽，能够动态插入底层通信协议模块。同时，核心能够按照最优的算法选择最快的连接进行通信。在传输层和网络层遵循 GIOP 抽象协议映射，可以定制自己的通信协议，如 IIOP(TCP/IP)、UIOP(UDP)、LIOP(LMQ)等，也可以提供安全的加密算法对消息进行加密。

协议槽不属于 GIOP 协议，也不属于底层消息传输层，它是位于 ORB 核心中的一个构件插槽，完成核心对底层通信协议(如 IIOP 协议)的登记。由核心的通信协议选择算法产生默认的通信协议，即最优的通信协议，在对象的请求引发时，由默认的通信协议完成客户与服务对象的通信。协议槽中登记了和具体协议相关的通信实体，如连接工厂、倾听器、映像工厂，协议的实现和 IIOP 相同，在消息发送时调用各自的消息发送函数。

```
TList<TOutgoingConnFactory*,TOutgoingConnFactory*> connFactoryList_;//协议槽所登记的连接工厂
TList<TCommListener*,TCommListener*> listeners_;//协议槽所登记的倾听器
TList<TProfileFactory*,TProfileFactory*> profileFactories_;//协议槽所登记的映像工厂
```

倾听器：服务端 ORB 一开始运行，守护线程开始启动，按照并发服务器模式，监听客户所发起的各种请求，并进行请求分派。

连接工厂：用于客户端创建连接，至于其所创建的细节均被封装在连接工厂中。

端点：标记目标程序的唯一地址(如 IP 地址、端口号)，连接请求通常针对某个端点。

映像工厂：根据标准标记映像，产生协议的标记映像，即将服务端的对象地址信息从网络形态转变为能被客户用以创建连接的主机形态。

连接：即借助下层通信设施实现通信的逻辑实体。

以上所定义的几种逻辑元素中，倾听器在服务端的 ORB 核心协议槽中登记，每个倾听器对应一个端点，连接工厂和映像工厂在客户端的核心协议槽中登记，而连接只是由以上几种元素所产生。

3.2 动态引发接口类的实现

DII 指客户可以动态创建和引发对对象的请求，因此 DII 中需要完成与客户存根相同的功能。DII 请求由对象引用、操作名、参数列表组成。CORBA 规范中规定用 NamedValue 来封装参数，由 NVList 来构建参数列表，用 UnknownUserException 来存放用户返回的异常。通过线程来实现 DII 中的延迟引发和多请求，可以替代客户多线并发的情况。

DII 中最主要的是通过 Request 对象完成对请求的封装和请求的引发，在 Request 中保存了一系列和请求有关的数据结构。因此，DII 的设计，最主要的是实现和请求有关的数据结构和高效使用 ORB 中的资源和通信设施来完成请求的引发，以及请求的延迟引发和请求结果的查取。

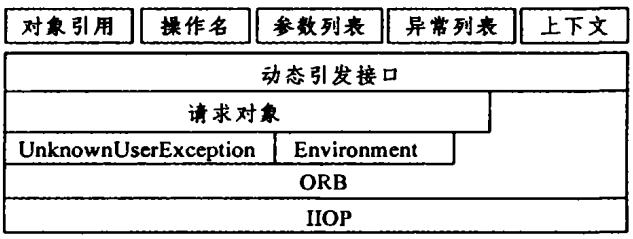

图2 动态引发接口类的关系层次图

(1)Request *对象* 客户程序引发一个服务对象上的方法，一个请求对象被创建出来代表方法引发。请求对象被写到一个缓冲区，并发送到服务方的对象实现。当客户程序使用客户 stub，此进程透明地发生，DII 的客户程序必须自己创建和发送请求对象。一个请求对象代表在 CORBA 对象上的方法的一个引发。如果想引发同一个 CORBA 对象上的两个方法，或不同对象上的相同方法，就需要两个请求对象。

创建一个请求对象的更复杂的方法是 Object::_create_request()方法，这也是所有 CORBA 对象继承的。此方法有多个参数，它刻画了结果的类型和可能产生的用户异常。为使用_create_request()方法，你必须创建它作为参数的构件。使用_create_request()方法的潜在的优点是性能。可以在多个_create_request()调用里重用参数构件，如果你在多个目标对象上引发相同的方法的话。

```
class Request{
    ...
    Any& add_value_arg(char * name,Any * any,Flags flag);
    Void set_return_type(TypeCode_ptr tc);
    Any& return_value();
    Status invoke() //同步引发请求
    Status send_oneway() //单向引发请求
    Status send_deferred() //延迟引发请求
    Status get_response() //取回请求引发结果
    Boolean poll_response() //查询请求
}
```

图3 Request 对象的类结构

(2)*用 Any 类型来封装参数* Any 是一个通用 CORBA 对象，封装了任何类型的参数。Any 可以放被定义在 IDL 中的任何类型。确定一个到请求的 Any 参数允许请求放任易参数类型和值，而不用编译器来解释类型匹配(这对结果和异常同样适用)。因此，用 Any 对象来确定目标对象方法的参数、结果和异常。

Any 由一个 TypeCode 和值组成。TypeCode 是一个对象，描述了对应值的类型。一个 Typecode 可以是简单的或递归的。简单的 TypeCode 由简单 IDL 类型(如 long)组成，由 IDL 编译器产生。递归的 TypeCode(如 struct)指 IDL 构造类型，因为它们描述的类型可以是递归的。IDL 编译器将为 IDL 文件中的构造类型产生 TypeCode。TypeCode 可以通过调用 ORB::create_struct_tc()或 ORB::create_exception_tc()实时获得。

(3)NamedValue *对象和* NVList *对象* NamedValue 对象是一个 OMG IDL 中一个常用的数据类型。它既可以直接作为一个参数的类型，也可以作为一种描述请求参数的机制。

NamedValue 表示引发对象方法的输入参数(in，inout)和输出参数(inout，out)，也表示请求的结果。名字特性是一个简单的字符串，其值的类型用 Any 表示。每个参数都以它的本地数据形式传递。

NVList 对象存放一系列名值对的列表，创建 NVList 列表操作在 ORB 接口中定义。NVList 由一个计数器和一个 NamedValue 的序列组成，每一个都有名字、值和标志，名字是参数名，值是封装了参数的 Any，标志表示参数的 IDL 模式。由于 NamedValue 和 NVList 对象的实现比较简单，在这里不做过多的论述，下面是参数标志的具体含义：

CORBA::ARG_IN	相关值是输入参数
CORBA::ARG_OUT	相关值是输出参数
CORBA::ARG_INOUT	相关值是输入输出参数

(4)Context 对象　一个上下文对象[1]包含一个特性列表，每个特性代表一个名字和与那个名字相关联的一个字符串。上下文特性代表了客户、环境或者不便于作为参数传递请求情况的信息。上下文特性可以表示一个客户或应用环境的一部分，可以传递给一个服务环境，服务器可以向上下文对象查询这些特性。以下是 Context 对象提供的部分接口：

```
class Context {
public:
    ……
    virtual CORBA::Context_ptr parent() const;
        //返回父 Context 对象的指针
    virtual CORBA::Status create_child(const char *, Context_
      out);//创建一个子 Context 对象
    virtual CORBA::Status set_one_value(const char *, const
      CORBA::Any&);//设置对象的名和值
    virtual CORBA::Status set_values(CORBA::NVList_ptr);
        //设置 Context 对象 NVList 指针
    virtual CORBA::Status delete_values(const char *);
        //删除 Context 对象的一个值
    virtual CORBA::Status get_values(const char * start_scope,
      CORBA::Flags op_flags,
      const char * pattern,
      CORBA::NVList_out values);
        //获得 Context 对象中的特性值
    static Context_ptr unmarshalContext(GIOPStream& s);
        //Context 对象向 GIOP 流中编码
    static Context_ptr unmarshalContext(GIOPStream& s);
        //Context 对象从 GIOP 流中解码
}
```

3.3　动态引发接口请求的引发过程

动态引发接口请求的构建由用户完成，请求引发依靠 ORB 核心通信设施来完成。在请求的引发过程中，可以选择和当前网络情况相适应的通信协议，保持请求的高效。连接池的可重用连接可以减少客户创建连接的开销。

```
//动态引发请求的主要流程
1)取服务对象的对象引用，创建请求对象，主要由用户完成
2)根据通信协议优先级，从动态协议槽选择最优的通信协议
3)获取服务程序的地址，即端点信息
4)查询连接池中有无和给定端点信息匹配的连接，否则创建新连接
5)创建请求 ID
6)由请求 ID、对象键、操作名、oneway 标记创建 GIOP 流
7)对 GIOP 流进行参数编码
8)引发请求
……
```

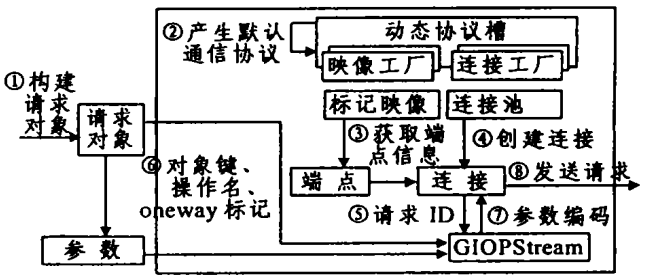

图4　动态引发接口的引发流程

3.4　动态引发接口的几种引发模式

当一个请求对象被创建和按正确的顺序加入参数后，可以按同步引发、同步延迟引发、单向三种方式[2]来引发请求。图3中给出了这三种模式的接口。

(1)同步引发　客户引发请求，阻塞等待响应。从客户的角度看，同步引发在行为意义上和 RPC 相同。这是大多 CORBA 应用程序所采用的公共通用模型，因为静态存根也支持这种方式。

(2)同步延迟引发　客户引发请求，不需要等待操作结束就把控制返回给调用者，直到请求的方法分派，才接收响应。客户可以引发一定数量的、独立的、长时间运行的服务。这种方式优于串行的阻塞等待请求方式，所有请求都可以并行发送，和在应答到达时接收响应。同步延迟是代替使用并发线程的一个方法。对于很多操作系统，send_deferred( )方法比创造一个线程更加经济。

(3)单向引发　当且仅当目标方法在 IDL 中用 oneway 定义。客户引发请求，不必等待服务器的应答。

结束语　为灵活地构建 CORBA 应用和提供同步、异步的方式来引发 CORBA 的对象请求，CORBA 规范提出了动态引发接口。本文从 ORB 通信核心改进和动态引发接口设计两个方面论述了一种高效动态引发接口的设计。经实际应用证明，这种方式设计的动态引发接口具有良好的可伸缩性、高效性。但由于有些通信协议本身对平台的支持程度和对消息包大小的支持程度，这种应用方式在某些平台上和消息传递上也必然存在一些局限性。

参 考 文 献

1　Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.3. June 1999

2　Vinoski S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. the IEEE Communications Magazine, 1997, 35(2)

3　Kuhns F, et al. The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware. the IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN '99), Salem, MA, August, 1999. 25～27

4　O'Ryan C, et al. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. the Middleware 2000 Conf. ACM/IFIP, Apr. 2000