(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 101923485 B
(45)授权公告日 2013.02.06

(21)申请号 200910246082.7
(22)申请日 2009.12.01
(66)本国优先权数据
200910086872.3 2009.06.17 CN
(73)专利权人 大唐软件技术股份有限公司
地址 100101 北京市朝阳区北苑路乙 108 号北美国际商务中心
(72)发明人 黄翔
(74)专利代理机构 北京德琦知识产权代理有限公司 11018
代理人 谢安昆 宋志强
(51)Int.Cl.
G06F 9/46(2006.01)
(56)对比文件
CN 1996256 A,2007.07.11, 全文 .
CN 1819588 A,2006.08.16, 全文 .
EP 2015179 A1,2009.01.14, 全文 .

审查员 钟阳雪

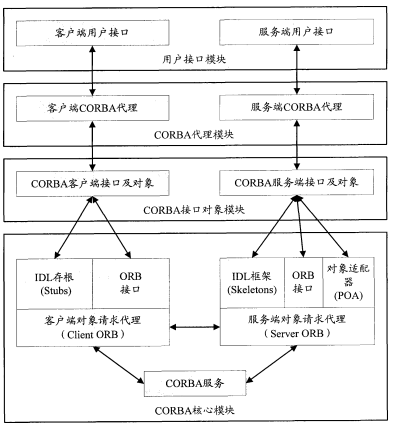
权利要求书 1 页 说明书 11 页 附图 4 页

(54)发明名称

CORBA 系统中的 JAVA 远程调用方法

(57)摘要

本发明公开了一种 CORBA 系统中的 JAVA 远程调用方法，该方法接收用户的调用请求，通过 JAVA 动态代理和拦截对所述调用请求进行重组，将所述调用请求转换为对 CORBA 服务的请求，通过预先封装的 CORBA 接口和实现类调用标准 CORBA 核心实现远程调用。用户进行远程 JAVA 调用时无须关心具体的调用实现过程，也无须实现特定的 CORBA 接口，使 JAVA 远程调用的前期开发和后期维护的工作量大大减少，且实现非常简单。



CN 101923485 B

1. 一种 CORBA 系统中的 JAVA 远程调用方法，其特征在于，该方法包括 ：

接收用户的 JAVA 远程调用请求，通过 JAVA 动态代理对所述调用请求进行拦截并重组，以将所述 JAVA 远程调用请求转换为对标准 CORBA 实现类的调用请求，通过 CORBA 接口调用标准 CORBA 核心实现远程调用 ；

其中，所述通过 JAVA 动态代理对所述调用请求进行重组，包括 ：通过 JAVA 动态代理在所述调用请求中添加用于 CORBA 实现类的控制语句 ；

所述通过 CORBA 接口调用标准 CORBA 核心实现远程调用包括 ：

预先建立统一调用接口，其中包含正向调用的接口对象和回调的接口对象，通过所述控制语句由所述统一调用接口调用标准 CORBA 核心进行远程调用。

2. 如权利要求 1 所述的 JAVA 远程调用方法，其特征在于，所述控制语句采用 XML 格式封装。

3. 如权利要求 1 所述的 JAVA 远程调用方法，其特征在于，所述正向调用接口对象包括 ：用于实现正向调用的方法和用于连通检测的方法 ；所述回调接口对象包括 ：用于实现回调的方法和用于连通检测的方法。

CORBA 系统中的 JAVA 远程调用方法

技术领域

[0001] 本发明涉及 JAVA 技术领域，特别涉及 CORBA 系统中的 JAVA 远程调用方法。

背景技术

[0002] CORBA（通用对象代理体系结构，Common Object Request BrokerArchitecture）是对象管理组织（OMG）为解决分布式处理环境（DCE）中，硬件和软件系统的互连而提出的一种解决方案。具有模型完整、先进，独立于系统平台和开发语言，被支持程度广泛的特点。目前，绝大多数分布计算平台厂商都支持和遵循 CORBA 规范。COBRA 系统，主要分为 3 个模块：对象请求代理模块、公共对象服务模块和公共设施模块。最底层是对象请求代理（ORB）模块，用于实现对象间的通讯和互操作，在 ORB 模块之上为公共服务模块，可以提供诸如并发服务、名字服务、事务服务、交易服务、安全服务等具体应用服务；最上层的公共设施模块则用于提供可直接为业务对象使用的服务，规定业务对象有效协作所需的协议规则。CORBA 允许不同应用之间相互通信，而不管它们存在于哪里以及是谁设计的。其中 ORB 模块是在调用对象之间建立 Client/Server 关系的中间件，是实现不同应用之间互相通信的关键。通过 ORB 模块，客户可以透明地调用一个服务对象上的方法，这个服务对象可以在本地，也可以在通过网络连接的其他机器上。ORB 模块截获这一调用，同时负责查找实现服务的对象并向其传递参数、调用方法返回最终结果。客户并不知道服务对象位于什么地方，它的编程语言和操作系统是什么，也不知道不属于对象接口的其他系统部分。这样，ORB 模块在异构分布环境下为不同机器上的应用提供了互操作性，并无缝 地集成了多种对象系统。

[0003] ORB 模块采用使用接口定义语言（IDL）实现，而 IDL 是独立于其他语言的，这使 ORB 可以适用于采用不同的编程语言实现的模块和组件。更重要的是，ORB 允许通过创建 IDL 对不支持 CORBA 的老旧系统进行建模，将老旧系统进行包装，从而与老旧系统接口之间传递信息。

[0004] CORBA 系统可以实现客户端通过网络从服务器端远程调用服务器端提供的服务实现客户端本地的操作，例如集成在客户端设备中的 CORBA 客户端将客户端的指令和数据等通过客户端 ORB 的包装发送给集成在服务器端设备中的 CORBA 服务器端，CORBA 服务器端解析出具体的指令和数据后交给服务器端设备进行处理，并将处理结果通过服务器端 ORB 包装后再返回 CORBA 客户端，从而实现服务的远程调用，这样的应用可以使客户端将一部分处理任务交给远端的服务器进行处理从而弥补本地客户端处理能力不足的问题。

[0005] 目前，人们普遍利用 CORBA 系统实现了 JAVA 对象的远程调用，但在 CORBA 系统中要实现 Java 对象远程调用，一般还需要预先规划并实现 JAVA 调用所对应的 CORBA 接口，即针对每一个 JAVA 调用，设计一个接口实现模块，将 JAVA 调用连接到 CORBA 实现类，通过接口实现模块使 JAVA 调用能够使用 CORBA 系统的服务，并通过 ORB 模块实现与远端 CORBA 系统的通信，从远端服务器中调用所需的资源。

[0006] 图 1 为现有在 CORBA 系统中实现 JAVA 对象远程调用的原理图，如图 1 所示，接口实现模块 1 ～ 6 分别对应不同的 JAVA 调用，当客户端接收到 JAVA 对象调用请求时，对应的

接口实现模块将调用请求转换为符合 CORBA 接口规范的请求，并通过 CORBA 接口和对象模块提供的客户端 CORBA 接口和实现类，使用 CORBA 核心模块中的 ORB 和 CORBA 服务将调用请求发送到服务端 ORB，通过服务端对象适配器、ORB 接口等，调用服务端 CORBA 接口和实现类，并通过服务端的接口实现模块将 CORBA 调用和对象转换为服务端相应的 JAVA 调用和对象，服务端处理完成后，将处理结果 经原路径返回客户端。

[0007] 从上述的操作可以看出，JAVA 对象远程调用中，所有的 JAVA 调用均须单独通过特定的接口实现模块来完成与 CORBA 服务的对接，每个 JAVA 调用是单独实现的。如果用户需求的 JAVA 调用的接口变更，则需要重新设计接口实现模块。而接口实现模块，其原理是直接调用 CORBA 接口来实现与 CORBA 对接，该模块中需要进行生成接口 IDL 文件、编译 IDL 文件成 Java 文件，按照编译完成的 Java 文件生成相应实现类，使用 CORBA 的接口连接 CORBA 服务等多个复杂步骤才能完成与 CORBA 的对接，即使很小的修改，也要重新进行一整套复杂的接口实现模块的设计实现过程，而调用接口变更的情况是很常见的，因此对于现有在 CORBA 系统中实现 Java 对象远程调用的方法，其后期系统维护的工作量很大，且整个实现架构比较复杂，需要预先规划并设计特定的接口实现模块，实现起来非常困难。

发明内容

[0008] 本发明实施例提供一种 CORBA 系统中的 JAVA 远程调用方法，无需特定的接口实现模块，实现简单，在 JAVA 调用变更时所需的维护工作量很小。

[0009] 为达到上述目的，本发明的技术方案具体是这样实现的：

[0010] 一种 CORBA 系统中的 JAVA 远程调用方法，该方法包括：

[0011] 接收用户的调用请求，通过 JAVA 动态代理和拦截对所述调用请求进行重组，将所述调用请求转换为对 CORBA 服务的请求，通过预先封装的 CORBA 接口和实现类调用标准 CORBA 核心实现远程调用。

[0012] 由上述的技术方案可见，本发明的这种 CORBA 系统中的 JAVA 远程调用方法，用户进行远程 JAVA 调用时无须关心具体的调用实现过程，也无须实现特定的 CORBA 接口，使 JAVA 远程调用的前期开发和后期维护的工作量大大减少，且实现非常简单。

附图说明

[0013] 图 1 为现有在 CORBA 系统中实现 JAVA 对象远程调用的原理图；

[0014] 图 2 为本发明实施例的 JAVA 远程调用架构示意图；

[0015] 图 3 为本发明实施例的 JAVA 远程调用方法的正向调用流程图；

[0016] 图 4 为本发明实施例的 JAVA 远程调用方法的回调流程图。

具体实施方式

[0017] 为使本发明的目的、技术方案及优点更加清楚明白，以下参照附图并举实施例，对本发明进一步详细说明。

[0018] 本发明主要是在现有 CORBA 接口对象模块之上增加两个模块，即 CORBA 代理模块和用户接口模块，通过用户接口模块接收用户的 JAVA 调用请求，并通过 CORBA 代理模块将用户的 JAVA 调用请求统一进行重组和转换，将 JAVA 调用转换为标准的 CORBA 协议的对象

调用后，再由CORBA接口对象模块和CORBA核心模块完成具体的CORBA调用，使用户进行远程JAVA调用时只需面对用户接口模块提供的简单的用户接口，无须关心具体的调用实现过程，也无须为特定的JAVA调用设计实现针对该JAVA调用的特定的CORBA接口模块，因此该方法只需按照标准CORBA实现统一的CORBA代理模块和用户接口模块，之后若要改变应用和改变接口，只需重新将用户的JAVA调用在CORBA代理模块中相应地重新指向到其它标准CORBA对象或实现类的即可，无须再针对不同的JAVA调用进行特定CORBA接口的实现和更改，使JAVA远程调用的后期维护工作量大大减少，且实现非常简单。

[0019] 图2为本发明实施例的JAVA远程调用架构示意图，如图2所示，该架构从上至下共分4个模块：用户接口模块、CORBA代理模块、CORBA接口对象模块、CORBA核心模块。

[0020] 第1层是用户接口模块，提供了面向用户的远程调用的工具和方法，如绑定到CORBA命名服务、从CORBA命名服务获取对象、数据发送到 CORBA通知服务、从CORBA通知服务接收通知事件等；具体如初始化CORBA代理模块中的CORBA代理对象，并将用户的JAVA调用请求交给CORBA代理模块处理。在该模块中，提供给用户的接口及其实现的功能是可以任意设计的，可以根据具体需要实现，接口的具体实现对于用户来说是透明的，用户并不需要关心，只需要根据提供的接口进行相应调用即可。

[0021] 第2层是CORBA代理模块，负责对用户接口模块下发的调用请求进行解析、重组和转发。具体是通过JAVA动态代理和拦截技术拦截用户的JAVA远程调用请求，并在该请求中插入控制说明，并将加入控制说明的调用请求发送给CORBA接口对象模块，进行具体的CORBA调用。由于在请求中插入了控制说明，因此CORBA接口对象模块可以根据控制说明的内容得知该JAVA调用对应哪个标准CORBA对象调用，应执行什么操作，因此实现了将普通Java调用请求转换为CORBA调用。CORBA代理模块在本架构中起到了承上启下的作用，对于用户接口模块而言，该模块屏蔽了直接对CORBA的API调用，而对于CORBA接口对象模块而言，该模块提供了标准的CORBA调用和结果的适配，实际上实现了从Java协议到CORBA协议的转换。其中，调用请求和响应中包含的各种参数和数据可以采用XML语言进行封装以保证兼容性和易用性。

[0022] 第3层是CORBA接口对象模块，该模块包含根据CORBA协议封装的具体接口和实现类，这些接口及其实现类负责通过底层CORBA核心模块实现对象的远程调用。现有CORBA系统中并没有提供对JAVA对象的回调的支持，而在本发明实施例中，通过在本模块中建立一个特殊的接口对象，可以供其他接口对象进行正向调用和回调，从而实现了对JAVA对象的回调，这个特殊的接口对象的具体实现方法如下：首先根据标准的CORBA协议建立IDL文件，该IDL中包含1个正向调用接口对象和1个回调接口对象，正向调用接口对象中实现用于正向调用的方法(invoke)和用于连通检测的方法(ping)，回调对象中实现用于回调的方法(on_data)和同样用于连通检测的方法(ping)；然后将IDL文件编译成Java文件，并分别实现相应 的接口。其中invoke方法是用于正向调用的实现方法，将用户请求调用的对象和参数作为该方法的输入参数，并查找并输出给相应的对象以实现正向调用，而on_data方法是用于回调的实现方法，其原理和invoke方法相同；而ping方法是负责检查待调用的远程对象是不是通的，从而可以实现在不通情况下的自动重连。通过建立支持回调的接口对象，其他接口对象通过调用这个统一的接口对象都可以实现正向调用和回调。另外，由于正向调用对象和回调对象中都包括用于连通检测的方法(ping)，因此，CORBA代理

模块还可以通过 ping 方法对调用对象接口所使用的网络连接进行心跳监测，并对于异常的接口进行自动重连的操作。当然如果不需要自动重连的功能，在正向调用对象和回调对象中也可以不包括 ping 方法。

[0023] 第 4 层是 CORBA 核心模块，是符合 CORBA 规范的标准实现，包含如图中 IDL 存根 (Stubs)、ORB 接口、客户端对象请求代理 (Client ORB)、CORBA 服务、IDL 框架 (Skeletons)、对象适配器 (POA)、服务端对象请求代理 (Server ORB) 等，具体实现可以参考 OMG 组织有关 CORBA 规范的定义，这里不再详细描述。

[0024] 在上述的架构中，实现 JAVA 远程调用将会非常简单，用户只需按照标准 JAVA 发出调用请求，即可实现远程 JAVA 对象的调用，而无须关心具体调用实现，从而使 CORBA 系统中的 JAVA 远程调用的实现变得简单易行。

[0025] 图 3 为本发明实施例的 JAVA 远程调用方法的正向调用流程图，如图 3 所示，该流程包括如下步骤：

[0026] 步骤 301，接收调用请求。用户接口模块接收客户端用户发出的 JAVA 调用请求。

[0027] 步骤 302，拦截用户的调用请求。CORBA 代理模块通过 JAVA 动态代理和拦截技术，如 JDK，将该请求拦截。

[0028] 步骤 303，判断调用的对象是否为空。CORBA 代理模块判断请求中所调用的对象是否为空，若是，则返回步骤 301，否则执行步骤 304。这里判断调用对象是否为空是指判断用户调用的 JAVA 对象是否在 CORBA 接口对 象模块中有对应的 CORBA 实现类。

[0029] JAVA 对象预先需要绑定到 CORBA 的命名服务上，具体绑定方式可以采用如下的函数格式：

[0030] *@param bindName 绑定的名称

[0031] *@param implObj 绑定对象的一个实现类

[0032] *@throws CorbaUtilException 需要捕获的异常

[0033] */

[0034] public void bind(String bindName,Object implObj)throws

[0035] CorbaUtilException 函数。

[0036] 其实现代码如下：

[0037] public class TestBindService{

[0038] public static void main(String[]args){

[0039] CorbaUtil util = CorbaUtil.getInstance();

[0040] Properties props = System.getProperties();

[0041] props.put(" ORBInitRef ", " NotificationService = corbaloc::localhost:14100/

[0042] NotificationService; ;NameService = corbaloc::localhost:5555/NameService");

[0043] try{

[0044] util.bind(" testBindName",new TestAlarmServiceImpl()); //创建一

[0045] 个实现类

[0046] }catch(CorbaUtilException e){

[0047] e.printStackTrace() ;

[0048] }

[0049] }

[0050] }

[0051] 步骤 304,获取调用的方法、参数并重新进行封装。

[0052] 重新封装可以采用往调用请求中插入控制说明的方法,将调用的 JAVA 对象指向到标准的 CORBA 实现类,以实现调用请求中 JAVA 对象到标准的 CORBA 实现类的转换。控制说明的内容可以采用 XML 格式进行封装,以 保证良好的兼容性,当然也可以采用其它私有的格式,只要能够被识别即可。

[0053] 其中,本发明在将 JAVA 调用适配成 CORBA 调用过程中,内部通过定义 XML 格式的控制语句,实现了请求的重定义和封装,具体的 XML 格式如下 :

[0054] < ? xml version = " 1.0" encoding = " gb2312" ? >

[0055] <ControlParam>

[0056] <methodName>say</methodName>

[0057] <invokeParam>hello</invokeParam>

[0058] <callbackId>4f6a67ee-4c67-4463-9594-a8cc02ddbfbb</callbackId>

[0059] <callbackName>com.cattsoft.corba.interface.IHelloCallback</callbackName>

[0060] </ControlParam>

[0061] 从这个 XML 可以知道,本次远程调用的方法是 say,方法的参数是 hello,回调对象的名称是 com.cattsoft.corba.interface.IHelloCallback,回调对象的标识是 4f6a67ee-4c67-4463-9594-a8cc02ddbfbb。这样当这个控制请求通过 CORBA 服务真正传递到对端以后,就可通过 JAVA 的反射机制,调用远程对象的 say 方法,并将“hello”作为参数传递过去,并将结果返回,实现远程调用。

[0062] 步骤 305,将重新封装后的调用请求发送到 CORBA 核心模块。

[0063] 步骤 306,CORBA 核心模块中的客户端 ORB 将调用请求发送到服务端 ORB。

[0064] 步骤 307,服务端 ORB 收到调用请求后将请求转发给服务端的 CORBA 接口对象模块。

[0065] 步骤 306 和 307 是标准的 CORBA 远程调用过程,本发明在这里使用现有技术实现,因此不再赘述具体详细实现过程。

[0066] 步骤 308,判断调用请求中请求的服务类是否为空,若是则返回步骤 307,否则执行步骤 309。

[0067] 步骤 309,判断请求参数是否为空,若是,则执行步骤 310,否则执行步骤 311。

[0068] 步骤 310,调用用户请求的实现类,传入空参数,并将结果返回。

[0069] 在调用用户请求的实现类时,需要按照名称,从 CORBA 命名服务获取预先绑定的 JAVA 对象,返回对象是 ServiceClass 类型,具体实现的函数如下 :

[0070] *@param serviceName 服务名称

[0071] *@param serviceClass 服务的类对象

[0072] *@throws CorbaUtilException 需要捕获的异常

[0073] */

[0074] public Object locatService(String serviceName,Class serviceClass) throws CorbaUtilException 函数

[0075] 其实现代码如下：

[0076] public class TestLocator{

[0077] public static void main(String[]args){

[0078] CorbaUtil util = CorbaUtil.getInstance();

[0079] Properties props = System.getProperties();

[0080] props.put(" ORBInitRef ", " NotificationService = corbaloc::hx2009:14100/NotificationService; ;NameService = corbaloc::hx2009:5555/NameService");

[0081] try{

[0082] TestAlmCallbackImpl ac = new TestAlmCallbackImpl();

[0083] ITestAlarmService services = (ITestAlarmService)util.locatService(" testBindName",

[0084] ITestAlarmService.class);

[0085] String[]retuns = services.returnResult(" service2");

[0086] System.out.println(retuns[0]);

[0087] }catch(Exception e){

[0088] e.printStackTrace();

[0089] }

[0090] }

[0091] }

[0092] 步骤 311,判断调用参数中是否包含回调对象,若是,则执行步骤 312,否则执行步骤 313。

[0093] 步骤 312,为回调对象生成动态代理对象,调用用户请求的实现类,将代理对象作为调用参数,将调用结果返回。

[0094] 步骤 313,调用用户请求的实现类,并将结果返回。

[0095] 图 4 为,本发明实施例的 JAVA 远程调用方法的回调流程图,如图 4 所示,该流程包括如下步骤：

[0096] 步骤 401,接收回调请求。

[0097] 步骤 402,拦截回调请求。同样通过 JAVA 动态代理和拦截技术拦截。

[0098] 步骤 403,判断回调的对象是否为空。若是,则返回步骤 401,否则执行步骤 404。

[0099] 步骤 404,获取回调的方法、参数并重新进行封装。封装方法同调用请求

[0100] 步骤 405,将封装后的请求发送到 CORBA 核心模块。

[0101] 步骤 406,服务端 ORB 将请求发送到客户端 ORB。

[0102] 步骤 407,客户端 ORB 收到请求将请求转发到客户端的 CORBA 接口对象模块。

[0103] 步骤 408,判断用户请求调用的服务类是否为空,若是则返回步骤 407,否则执行

步骤 409。

[0104] 步骤 409,判断请求参数是否为空,若是,则执行步骤 410,否则执行步骤 411。

[0105] 步骤 410,调用用户请求的实现类,传入空参数,并将结果返回。

[0106] 步骤 411,调用用户请求的实现类,并将结果返回。

[0107] 回调的流程实际上和正向调用的过程相同,只不过调用的对象是回调的 对象,调用方向是由服务端回调客户端对象。执行回调的前提是必须先有一次正向调用,将回调对象传递给被调用方,然后被调用方才可以根据需要随时反向调用请求方,实现双向调用。

[0108] 另外,在调用和回调过程中,还涉及将数据发送到事件通道和从事件通道接收数据的过程,具体实现的函数如下 :

[0109] 将数据发送到事件通道 :

[0110] *@param xmlMsgs XML 格式的消息对象

[0111] *@param channeled 通道的标识

[0112] *@throws CorbaUtilException 需要捕获的异常

[0113] */

[0114] public void sendToChannel(String xmlMsgs, int channelId) throwsCorbaUtilException 函数

[0115] 将数据发送到事件通道 :

[0116] *@param xmlMsgs 以数组形式存放的 XML 格式的消息对象

[0117] *@param channeled 通道的标识

[0118] *@throws CorbaUtilException 需要捕获的异常

[0119] */

[0120] public void sendToChannel(String[]xmlMsgs, int channelId) throwsCorbaUtilException 函数

[0121] 其具体实现代码如下 :

```
[0122] public class TestCorbaSender{
[0123]  public static void main(String[] args){
[0124]    CorbaUtil util = CorbaUtil.getInstance();
[0125]    Properties props = System.getProperties();
[0126] props.put( " ORBInitRef ", " NotificationService = corbaloc::hx2009:14100/NotificationService; ;NameService = corbaloc::hx2009:5555/NameService" );
[0127]      try{
[0128]        util.sendToChannel(" hello",0);
[0129]        Thread.sleep(5000);
[0130]      }catch(Exception e){
[0131]        e.printStackTrace();
[0132]      }
[0133]  }
[0134] }
```

[0135] 从事件通道接收数据：

[0136] 从通道获取数据，该方法供没有回调对象的使用，是阻塞方法：

[0137] *@param channelId 通道的标识

[0138] *@throws CorbaUtilException 需要捕获的异常

[0139] */

[0140] public Object receiveFromChannel(int channelId) throwsCorbaUtilException 函数

[0141] 从通道获取数据，并调用 pusher 的回调方法将数据发出：

[0142] *@param pusher 该接口的实现类

[0143] *@param channeled 通道的标识

[0144] *@throws CorbaUtilException 需要捕获的异常

[0145] */

[0146] public void receiveFromChannel(ICorbaPusher pusher, int channelId) throws CorbaUtilException 函数。

[0147] 其实现代码如下：

[0148]
```
public class TestCorbaReceiver{
```

[0149]
```
 public static void main(String[]args){
```

[0150]
```
    CorbaUtil util = CorbaUtil.getInstance();
```

[0151]
```
    Properties props = System.getProperties();
```

[0152]
```
props.put( " ORBInitRef ", " NotificationService = corbaloc::hx2009:14100/Notif icationService; ;NameService = corbaloc::hx2009:5555/NameService" );
```

[0153]
```
      try{
```

[0154]
```
          util.receiveFromChannel(new CorbaPusherImpl(),0);
```

[0155]
```
      }catch(Exception e){
```

[0156]
```
        e.printStackTrace();
```

[0157]
```
      }
```

[0158]
```
    }
```

[0159]
```
}
```

[0160] 由上述的实施例可见，本发明的 JAVA 远程调用方法的关键在于将用户的 JAVA 调用请求通过重组转换为 CORBA 系统可识别并操作的标准 CORBA 调用，再通过标准的 CORBA 远程调用过程，将用户的 JAVA 调用从本地客户端传送到远端的服务端，再通过 CORBA 实现服务端实现类的调用并将调用的结果返回客户端，从而使 JAVA 远程调用对于用户完全透明，用户不需要为每一种 JAVA 远程调用而单独设计特定的接口实现模块以与 CORBA 核心模块对接，使 JAVA 远程调用的后期维护的工作量大大减少，且实现非常简单。

[0161] 所应理解的是，以上所述仅为本发明的较佳实施方式而已，并不用于限定本发明的保护范围，凡在本发明的精神和原则之内，所做的任何修改、等同替换、改进等，均应包含在本发明的保护范围之内。

[0162] 随附 XML 格式以及示例代码如下。

[0163] XML 格式

[0164] 本发明在将 JAVA 调用适配成 CORBA 调用过程中，内部通过定义 XML 格式的控制语句，实现了请求的重定义和封装，具体的 XML 格式如下：

[0165] <? xml version =" 1.0" encoding =" gb2312" ? >

[0166] <ControlParam>

[0167] <methodName>say</methodName>

[0168] <invokeParam>hello</invokeParam>

[0169] <callbackId>4f6a67ee-4c67-4463-9594-a8cc02ddbfbb</callbackId>

[0170] <callbackName>com.cattsoft.corba.interface.IHelloCallback</callbackName>

[0171] </ControlParam>

[0172] 从这个 XML 可以知道，本次远程调用的方法是 say，方法的参数是 hello，回调对象的名称是 com.cattsoft.corba.interface.IHelloCallback，回调对象的标识是 4f6a67ee-4c67-4463-9594-a8cc02ddbfbb。这样当这个控制请求通过 CORBA 服务真正传递到对端以后，就通过 JAVA 的反射机制，调用远程对象的 say 方法，并将“hello”作为参数传递过去，并将结果返回。

[0173] 示例代码

[0174] 绑定对象

[0175] public class TestBindService{

[0176] public static void main(String[]args){

[0177] CorbaUtil util = CorbaUtil.getInstance() ;

[0178] Properties props = System.getProperties() ;

[0179] props.put(" ORBInitRef " , " NotificationService = corbaloc::localhost:14100/NotificationService ; ;NameService

[0180] = corbaloc::localhost:5555/NameService") ;

[0181] try{

[0182] util.bind(" testBindName" ,newTestAlarmServiceImpl()) ;// 创建一个实现类

[0183] }catch(CorbaUtilException e){

[0184] e.printStackTrace() ;

[0185] }

[0186] }

[0187] }

[0188] 获取对象

[0189] public class TestLocator{

[0190] public static void main(String[]args){

[0191] CorbaUtil util = CorbaUtil.getInstance() ;

[0192] Properties props = System.getProperties() ;

[0193] props.put(" ORBInitRef " , " NotificationService =

corbaloc::hx2009:14100/NotificationService；;NameService＝

[0194] corbaloc::hx2009:5555/NameService″）；

[0195] try{

[0196] TestAlmCallbackImpl ac＝newTestAlmCallbackImpl()；

[0197] ITestAlarmService services＝(ITestAlarmService)util.locatService(″testBindName″，

[0198] ITestAlarmService.class)；

[0199] String[]retuns＝services.returnResult(″service2″)；

[0200] System.out.println(retuns[0])；

[0201] }catch(Exception e){

[0202] e.printStackTrace()；

[0203] }

[0204] }

[0205] }

[0206] 发送数据到事件通道

[0207] public class TestCorbaSender{

[0208] public static void main(String[]args){

[0209] CorbaUtil util＝CorbaUtil.getInstance()；

[0210] Properties props＝System.getProperties()；

[0211] props.put(″ORBInitRef″，″NotificationService＝corbaloc::hx2009:14100/NotificationService；;NameService＝

[0212] corbaloc::hx2009:5555/NameService″)；

[0213] try{

[0214] util.sendToChannel(″hello″,0)；

[0215] Thread.sleep(5000)；

[0216] }catch(Exception e){

[0217] e.printStackTrace()；

[0218] }

[0219] }

[0220] }

[0221] 从事件通道接收数据

[0222] public class TestCorbaReceiver{

[0223] public static void main(String[]args){

[0224] Co由aUtil util＝CorbaUtil.getInstance()；

[0225] Properties props＝System.getProperties()；

[0226] props.put(″ORBInitRef″，″NotificationService＝corbaloc::hx2009:14100/NotificationService；;NameService＝

[0227] corbaloc::hx2009:5555/NameService″)；

```
[0228]          try{
[0229]                util.receiveFromChannel(new CorbaPusherImpl(),0);
[0230]          }catch(Exception e){
[0231]              e.printStackTrace();
[0232]          }
[0233]     }
[0234]  }
```

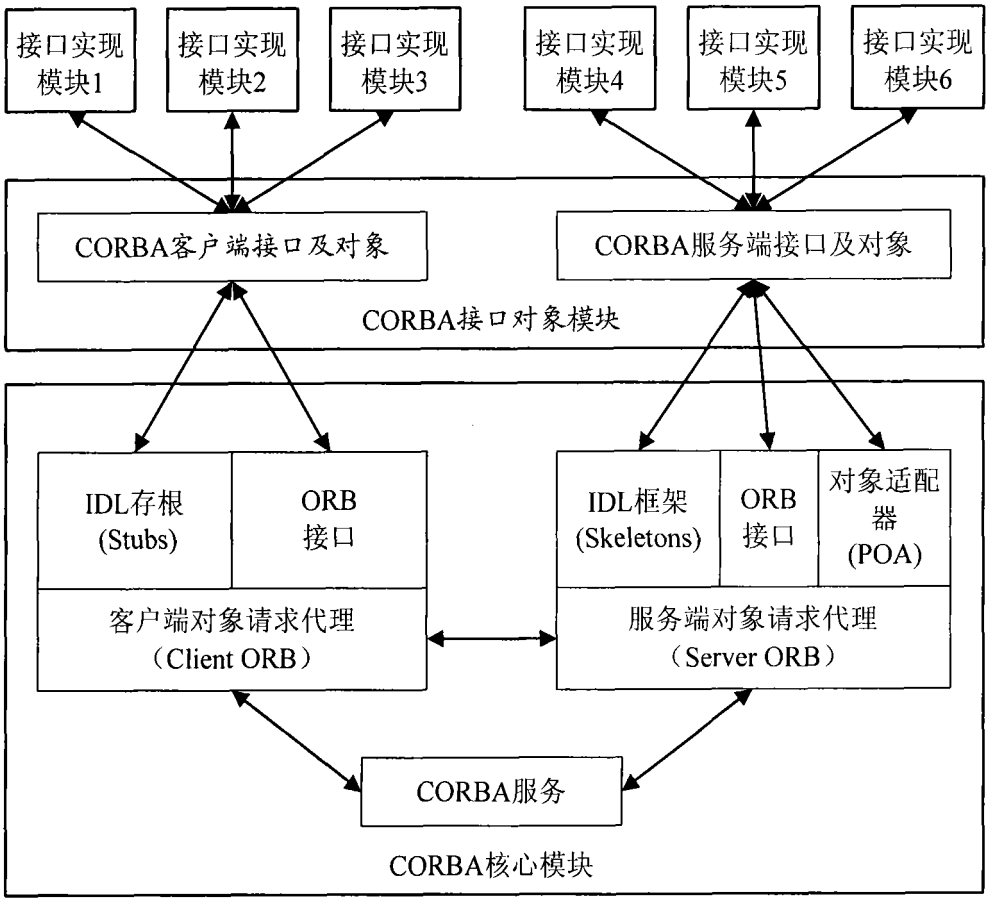

接口实现模块1
接口实现模块2
接口实现模块3
接口实现模块4
接口实现模块5
接口实现模块6
CORBA客户端接口及对象
CORBA服务端接口及对象
CORBA接口对象模块
IDL存根(Stubs)
ORB接口
IDL框架(Skeletons)
ORB接口
对象适配器(POA)
客户端对象请求代理（Client ORB）
服务端对象请求代理（Server ORB）
CORBA服务
CORBA核心模块

图 1

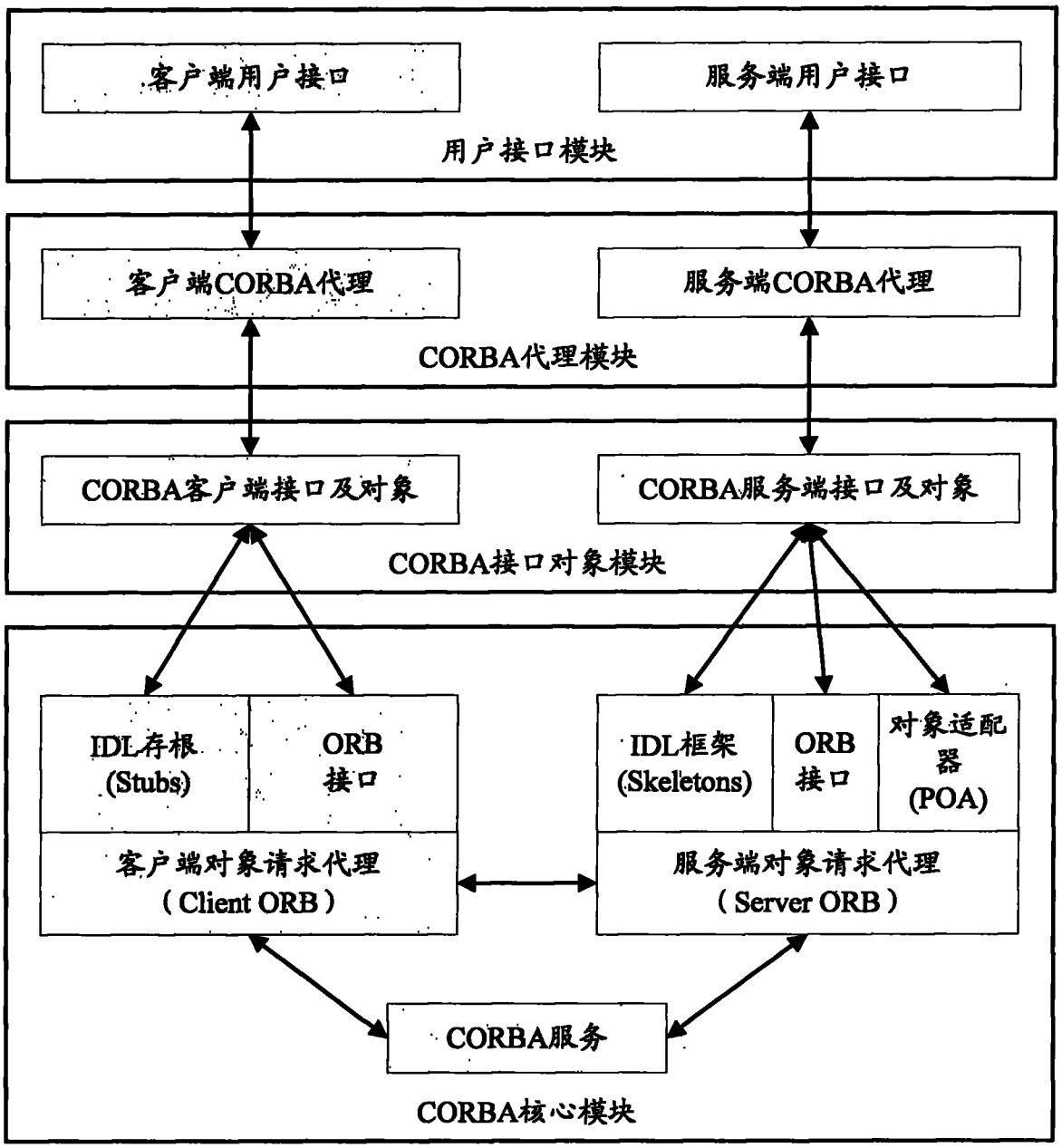

客户端用户接口
服务端用户接口
用户接口模块
客户端CORBA代理
服务端CORBA代理
CORBA代理模块
CORBA客户端接口及对象
CORBA服务端接口及对象
CORBA接口对象模块
IDL存根
(Stubs)
ORB
接口
IDL框架
(Skeletons)
ORB
接口
对象适配
器
(POA)
客户端对象请求代理
（Client ORB）
服务端对象请求代理
（Server ORB）
CORBA服务
CORBA核心模块

图 2

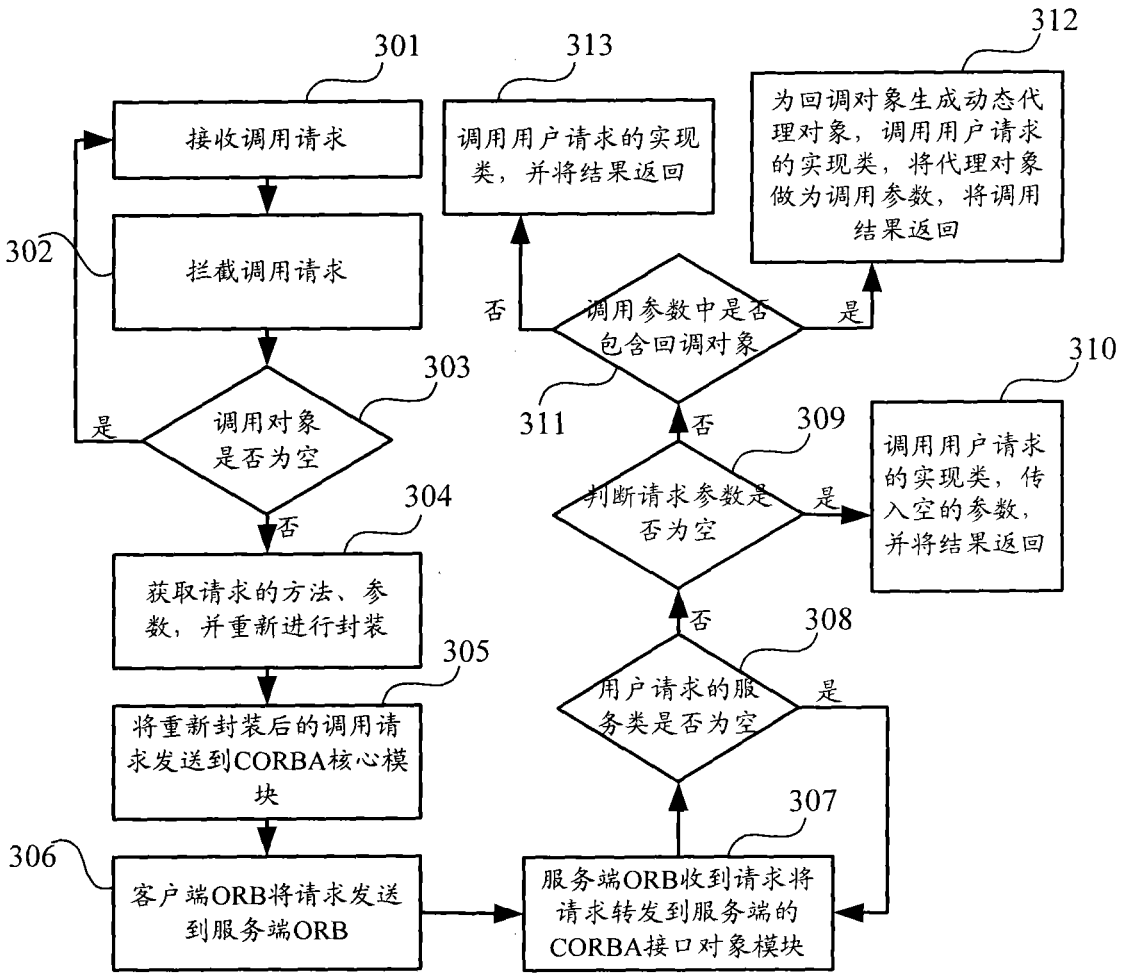

301
接收调用请求
302
拦截调用请求
303
调用对象是否为空
是
否
304
获取请求的方法、参数，并重新进行封装
305
将重新封装后的调用请求发送到CORBA核心模块
306
客户端ORB将请求发送到服务端ORB
307
服务端ORB收到请求将请求转发到服务端的CORBA接口对象模块
308
用户请求的服务类是否为空
是
否
309
判断请求参数是否为空
是
否
310
调用用户请求的实现类，传入空的参数，并将结果返回
311
调用参数中是否包含回调对象
是
否
312
为回调对象生成动态代理对象，调用用户请求的实现类，将代理对象做为调用参数，将调用结果返回
313
调用用户请求的实现类，并将结果返回

图 3

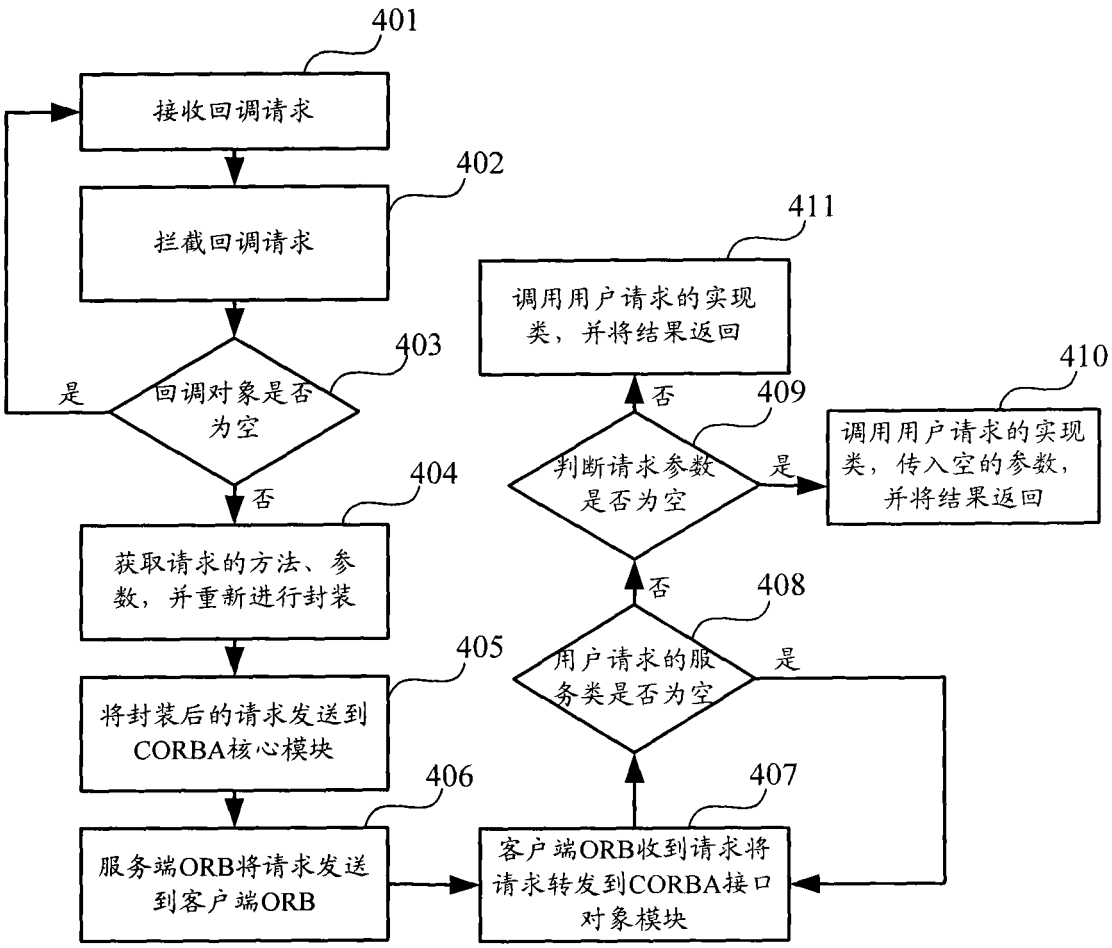

401
接收回调请求
402
拦截回调请求
403
回调对象是否为空
是
否
404
获取请求的方法、参数，并重新进行封装
405
将封装后的请求发送到CORBA核心模块
406
服务端ORB将请求发送到客户端ORB
407
客户端ORB收到请求将请求转发到CORBA接口对象模块
408
用户请求的服务类是否为空
是
否
409
判断请求参数是否为空
是
否
410
调用用户请求的实现类，传入空的参数，并将结果返回
411
调用用户请求的实现类，并将结果返回

图 4