第32卷　第22期　　　　计　算　机　工　程　　　　2006年11月
Vol.32　　No.22　　　　Computer Engineering　　　　November 2006
·软件技术与数据库·　　文章编号：1000—3428(2006)22—0099—03　　文献标识码：A　　中图分类号：TP393

一种新的基于序列化的 Java RMI 方法

戴　亮[1]，方晓勤[2]，李　丽[1]
(1. 中国农业大学信息与电气工程学院，北京 100083；2. 九江职业技术学院，九江 332007)

摘　要：在实际应用中，RMI 经常会因为通过网络传递参数需要花费太多时间，影响 RMI 的性能。该文在分析序列化技术的基础上，提出一种新的基于序列化的 RMI 方法，并给出该方法的设计思想和具体实现。实际应用表明，该方法能够有效减少 RMI 通过网络传输的字节流的长度，缩短 RMI 的调用时间，加强 RMI 的性能。
关键词：RMI；序列化；Stub

A New Method for Java RMI Based on Serialization

DAI Liang[1], FANG Xiaoqin[2], LI Li[1]
(1. College of Information and Electrical Engineering, China Agricultural University, Beijing 100083;
2. Jiujiang Vocational & Technical College, Jiujiang 332007)

【Abstract】In practical applications, RMI often spends too much time in sending parameters through the network, which hinder the performance of RMI. This paper designs and implements a new method for RMI based on serialization. The application results show that the method can decrease the length of byte stream, shorten the invoke time, and improve the performance.
【Key words】RMI; Serialization; Stub

Java RMI 作为网络分布式应用系统的核心解决方案之一，其主要功能是使得客户端程序可以调用位于远端服务器上的对象所提供的方法。RMI 从客户端向服务器端传输远程方法的参数及从服务器向客户端返回远程方法的返回值时，会序列化对象，并用于网络间传输对象。但是常规的 RMI 机制会导致相同对象的大量重复传输。而远程调用很可能发生于网络上两个相距甚远的机器，相同对象的重复不仅大大延长了调用花费的时间，而且使得远程调用更加依赖于极其不稳定的网络环境。

1 Java RMI 基本原理

RMI 实现远程调用的过程中，在客户端会有对应于远程对象的代理 Stub，Stub 和远程对象具有相同的接口，客户端只需调用 Stub 的方法就可实现对远程方法的调用，从而屏蔽掉网络通信的所有细节。

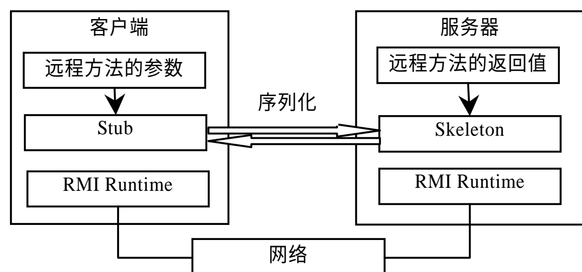

图 1 RMI 基本框架

当Stub被某客户调用时，该Stub使用内部对象引用来转发该调用。Java远程调用(JRMP)引擎会获得一个到服务器的活动连接，如果成功获得连接，就可以开始发送调用了。远程对象的ID、远程方法和远程方法的参数被序列化并通过一个输出流写入连接，发送到服务器。在服务器端，接收到客户端的请求，与客户建立连接之后，就会创建一个线程监听对服务器远程对象的调用。当有调用到来时，线程对传来的字节流进行反序列化，通过这些信息查找到被调用的对象，将参数传给远程对象的方法进行计算。再将计算得出的远程方法的返回值序列化，写入连接，返回给客户端，从而客户端完成了远程方法的调用[1]，如图 1。

2 设计及实现

本文以中国建设银行项目管理系统中的下达任务书的功能为例。该系统包括总行使用的宏观部分和各地分行使用的微观部分。当一个项目立项审批通过后，总行(宏观部分)会下达一份关于这个项目的信息的任务书给分行(微观部分)处理。在系统中宏观部分调用微观部分上的远程方法 submitTaskBook(Taskbook taskBook)实现下达任务书。项目对象中的属性有：任务书列表，项目文档列表，审批历史列表以及其他的项目信息。任务书对象中的属性有：对应的项目，实施进度列表，变更申请列表，概算列表和任务书的其他信息。

在如此一个复杂的对象关系中，当宏观调用微观的远程方法 submitTaskBook(Taskbook taskBook)下达任务书时，taskBook作为远程方法的参数会序列化写进连接，传到微观，按照序列化机制的深度拷贝原则，taskBook引用的项目、实施进度列表、变更申请列表等也会拷贝传到微观。如果再次下达和taskBook引用同一个项目的其他任务书到微观，或者甚至是再次下达taskBook时，RMI会同样再发送一份该对象的深度拷贝给微观[2]。也就是说会再次将taskBook引用的项

作者简介：戴　亮(1980－)，男，硕士生，主研方向：分布式系统，计算机图形图像技术及其应用等；方晓勤，讲师；李　丽，硕士、副教授
收稿日期：2006-02-28　**E-mail：**isliangd@gmail.com

目对象发送给微观，即使是项目的项目文档列表、审批历史列表、基本信息这些内容没有发生改变，也要重复发送这些数据。这明显传输了大量的重复信息，而且是通过具有众多不确定性的网络。

2.1 原型设计

如果系统能够识别出已经传输过的，并且没有变化的对象，而不重复传递这些对象，就会大大节省需要传递的字节流的长度。

考虑在客户端和服务器各保留一份已传输的对象的深度拷贝的列表，因为是深度拷贝，所以当添加了一个对象的拷贝后，认为该对象引用的子对象的拷贝也是在这个列表中的。在判断一个对象的拷贝是否在这个列表中，也就是是否已经传输过该对象时，不仅与列表的各元素比较，而且还与各列表元素直接或间接引用的子对象比较。如果该列表的元素以及列表元素的直接或间接引用的子对象，均不是要传输对象的一个拷贝，则认为没有传输过该对象，将该对象的拷贝作为一个新的元素添加到列表中。当传输一个已经传输过的对象时，该对象中已经传输过且值没有发生改变的子对象将不再传输，从而减少重复传输的数据。

在上面例子中，当宏观系统调用微观的 submitTaskBook (Taskbook taskBook)时，宏观系统实际上是调用 Stub 的方法，通过 Stub 来转发该调用。在 Stub 中试图建立一个关于远程对象的访问，如果访问成功，就获得一个 OutputStream，再调用 writeObject()方法将远程对象的参数 taskBook 序列化写入该 OutputStream。

Stub 是按照远程对象生成的，为了保证 Stub 不改变，希望 Stub 处理的是实际需要传递的远程方法的参数，而通过序列化之后，写入字节流传输的却仅仅是经过与已传输的对象列表进行比较，修改简化了的对象。在服务器端收到简化的对象，能反序列出实际要传输的对象。因此将与已经传输的对象列表比较，以及简化对象的过程封装到序列化对象的 writeObject()方法中，将与列表比较和恢复对象的过程封装到 readObject()方法中。如图 2，在 Stub 与套接字中间插入一个自定义的序列化过程，在 Skeleton 与套接字中间插入一个自定义的解序列化过程。

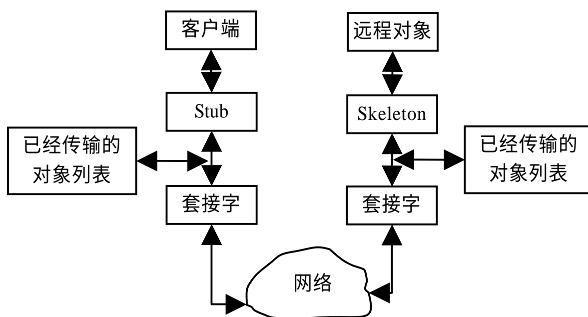

图 2 改进的 RMI 基本结构

具体的实现步骤是，为要传输的对象添加 writeObject()方法和 readObject()方法。当 Stub 序列化 taskBook 时，会调用添加的 writeObject()方法来序列化对象。在 writeObject()方法中，在已传输对象列表 objectList 的元素以及元素引用的对象中查找该对象 taskBook，得出 taskBook 先前是否已经传输过了。

如果 taskBook 没有传输过，甚至是没有作为其他对象引用的对象传输过，则将 taskBook 的深度拷贝添加到 objectList 中，然后调用默认的 defaultWriteObject()序列化该对象。在添加过程中，如果 taskBook 的引用的子对象已经存在于列表中，则 taskBook 的引用指向列表中已存在的子对象，并更新子对象的值，从而使得列表中只有子对象的一份最新的拷贝。

如果在 objectList 中找到 taskBook 的拷贝 objectCopy，意味着先前已经传输过 taskBook 了。则使用方法 compareObject (taskBook,objectCopy)比较 taskBook 和 object Copy。

compareObject()方法会返回值到数组 tagArray 中，来标识 objectCopy 中与 taskBook 中子对象相等的所有子对象在 objectCopy 中的位置。同时 compareObject()方法将 taskBook 中与 objectCopy 中相等的子对象设为空，从而得到简化了的对象；将 taskBook 中与 objectCopy 不相等的子对象复制到 objectCopy，更新 objectList。这样要传输的对象就改装成了不含重复传输对象的简化对象。

在服务器同样也会为每个客户保留一份已传输的对象深度拷贝的列表 objectList，当 Skeleton 接收到来自客户端的字节流后，readObject()方法从字节流中取出对象，并且试图取出标识 taskBook 与 objectCopy 中相等子对象位置的数组 tagArray。在 objectList 中查找 taskBook 的拷贝 objectCopy，如果未找到 objectCopy，则认为该对象为一未传输过的对象，将该对象的拷贝添加到 objectList ;如果找到了 objectCopy 并且从字节流中获得了标识数组 tagArray，则使用 assembleObject()方法拼装出远程方法的实际参数，也就是将 objectCopy 中 tagArray 所标识位置的子对象添加到 taskBook 中 ;同时将 taskBook 中与 objectCopy 不相等的子对象覆盖到 objectCopy，更新 objectList。最终服务器端的 Skeleton 得到 submitTaskBook()方法的实际参数 taskBook，进行运算并返回值给客户端，从而完成远程方法调用。

2.2 功能实现

下面列出实现上面功能的部分代码：

(1)writeObject()方法中，将当前对象与已传输对象深度拷贝的列表进行比较，得到新的简化的对象。并序列化该简化的对象。

```
writeObject(ObjectOutputStream stream) throws IOException {
List objectList = getTransObjectList()；
Object objectCopy = objectList.findObject(this);
if(objectCopy == null){
    objectList.addObject(this);
stream.defaultWriteObject();
    }
else{ int[] tagArray = compareObject(taskBook，objectCopy)；
stream.defaultWriteObject();
stream. writeObject(tagArray);
} }
```

(2)readObject()方法中，从传递来到字节流中读出简化的对象，并将其还原成原始的对象。

```
readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException {
        …
List objectList = getTransObjectList()；
stream.defaultReadObject();
  try{tagArray = (int []) stream.readObject();
}catch(EOFException e){
```

```
        //没有传输标识数组 }
    Object objectCopy = objectList.findObject(this);
    if(objectCopy ==null ){
       objectCopy.addObject(this); }
    else{
    assembleObject(this, tagArray);
    } }
```

(3)objectList.findObject(object)时，遍历 objectList 的每一项，使用方法 compareObject (Object object,Object objectCopy)判断 objectList 的每个元素是否是 object 的拷贝，或者是否是直接或间接引用到元素 object 的拷贝。如果是则返回该拷贝。方法 compareObject (Object object,Object objectCopy)是在对象 objectCopy 中查找是否有对象 object 的方法。

```
    Object compareObject (Object object,Object objectCopy){
    private ArrayList visited = new ArrayList();
    if (visited.contains(objectCopy)) return null;
    visited.add(objectCopy);
    Class objClass= objectCopy.getClass();
       if (objClass.isArray()){
    for (int i = 0; i < Array.getLength(objClass); i++)
    {   Object valObject = Array.get(objectCopy, i);
if(valObject.equal(object)){
      return    valObject}//需重载 equal 方法
       else{ compareObject (object, valObject);}
    }    }
    return null;
    Field[] fields = objClass.getDeclaredFields();
    AccessibleObject.setAccessible(fields, true);
    for (int i = 0; i < fields.length; i++){
      Field f = fields[i];
          if (!Modifier.isStatic(f.getModifiers())){
    try {
             Class t = f.getType();
             Object valObject = f.get(objectCopy);
                if(valObject.equal(object))
    {return    valObject}//需重载 equal 方法
             else{compareObject (object, valObject);}
           } catch (Exception e) { e.printStackTrace(); }
    }
```

3 结束语

RMI 需要在运行时通过网络传输远程方法的参数和返回值，由于网络存在延误时间和局部故障，传输长的字节流，必然使得调用时间增长、性能下降。文章通过自定义的序列化方法取代默认的深度拷贝的序列化机制，使得通过网络只发送实体修改过的部分，在应用实践中大大缩短了传输的字节流长度。

本文只讨论了自定义序列化在从客户端往服务器端传送远程方法参数过程中的应用，当从服务器端往客户端传送远程方法的返回值时，同样可以采用类似的方法来缩短传输的字节流长度。

参考文献

1 Reilly D, Reilly M. Java™ Network Programming and Distributed Computing[M]. Addison-Wesley, 2002.

2 Grosso W. Java RMI[M]. O'Reilly, 2001.

3 Oberg R. Mastering RMI: Developing Enterprise Applications in Java and EJB[M]. John Wiley & Sons, 2001.

4 Matiaz J B, Ivan R. Java 2 RMI and IDL Comparison[M]. SIGS Publications, 2000-02.

5 Matjaz J B, Ales Z, Ivan R. Are Distributed Objects Fast Enough[M]. SIGS Publications, 1998-05.

~~~~~~

(上接第 88 页)

(2)从图 2 还可以知道，除了可在控制器实现数据保护外，也可通过在模型端新建设备结合 VFS 来实现数据保护。建立一新的虚拟块设备，所有需保护的数据均置于此块设备中，对虚拟块设备创建具有数据保护机制的设备驱动程序，从而达到数据保护的目的。

在基于虚拟块设备对数据保护中，要在磁盘中划出一部分磁盘空间，并且把此部分空间映射到一个虚拟块设备中，撰写虚拟磁盘驱动代码，把要保护的数据放入此虚拟设备中去[4]。

如不撰写驱动程序，也可以利用Linux中实现的回接设备loopX(X表示 0~7 的数字)，通过将回接设备回接到一定容量的文件中，从而将这个文件当作一个可加解密的块设备来使用[3]。

3 结束语

近几年来，软件从基于构件(Component) 的开发迈向了基于软件框架(Framework) 和体系结构的开发道路[2]，特别是对软件体系结构的研究已成为软件工程中实现软件复用的又一轮新的技术研究热潮，对提高软件生产效率和软件产品质量有着重要的意义。目前虽然在软件体系结构的概念和理论方面还没有达成一定的共识和规范，但这并没有阻止基于软件体系结构的开发和应用的研究，基于不同领域(领域工程)和不同侧面的软件体系结构的研究如火如荼。而人们对软件体系结构研究的最终目的是为了获得所需的软件产品(软件的实现）。

本文通过分析 MVC 体系结构模式，阐述了基于 MVC 模式的文件加解密体系结构和这种体系结构的工作机理，并结合 MFC 给出文件保密柜的实现。可以看出，基于文件加解密体系结构模式的文件保护，由于它的灵活性、可复用性和易扩展性等特点，对数据保密有着重要的指导作用。

参考文献

1 冯　冲, 江　贺, 冯静芳. 软件体系结构理论与实践[M]. 北京: 人民邮电出版社, 2004.

2 Guttorm S. The REBOOT Approach to Software Reuse [J]. System Software, 1995, 30(3): 201-212.

3 毛德操, 胡希明. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2001.

4 Rubini A, Corbet J. Linux Device Drivers[M]. O’Reilly & Associates, 2001.

5 陈莉君. 深入分析 Linux 内核源代码[M]. 北京: 邮电出版社, 2002.
~~~~~~