**RMI与RMI-IIOP区别,以及JNDI相关**

参考：<http://blog.csdn.net/lcj8/article/details/3827679>

1.         RMI-IIOP(Java Remote Method Invocation over the Internet Inter-ORB Protocol)

         它是一种简单的，有效的网络通讯机制；

         能够让对象在内存中，java虚拟机中，不同的物理机器间通讯；

         RMI-IIOP并不是做远程调用的唯一选择，也可以使用java的RMI机制。RMI是java中做远程调用的最原始的机制，放在java.rmi包中。RMI-IIOP是RMI的一个特殊的版本，主要是为了兼容CORBA，它既使用java.rmi包，又使用javax.rmi包。RMI有一些很有用的功能在RMI-IIOP中并没有，比如：分布式垃圾收集，对象激活，可下载的class文件等。但是在EJB和j2ee中必须使用RMI-IIOP而不是RMI。

         基本概念：

         Remote Producure Call(RPC): 可以让一个过程（Producure）调用其它机器上的过程的机制；

         Remote Method Invocation(RMI): 可以让分布式的对象互相通信的机制。它可以让对象进行远程调用，而不仅仅是过程。这样在程序中就可以使用面向对象的特性，如：多态，继承，封装等；

         为什么要使用RMI-IIOP而不是RMI呢？因为在RMI中还有以下一些问题没有解决：

         Marshalling and Unmarshalling(组装和分解)：因为不同机器的物理结构可能不同，所以相互传输的数据要组装或分解以适应不同类型的机器。

         参数传递：java调用函数时传递参数主要有两种方法：值传递和引用传递。因为在远程调用中很难对引用进行操作，所以主要使用值传递，但是值传递要对参数进行拷贝。

         网络或机器的可靠性：在一个虚拟机中，如果虚拟机瘫痪了，那么整个程序都会瘫痪，所以不用考虑太多。但是在远程调用中，如果网络瘫痪或有一个参与调用的虚拟机瘫痪，则会引起程序出错，这是程序员必须考虑这些问题。

         RMI-IIOP做了很多事情来解决以上的问题，这样会减少程序员处理这些问题的工作。

         远程接口。为什么必须使用接口而不是具体的类？首先要说明一下接口的概念：

         Interface(接口)：它的目的是将类的某些信息暴露出来，如：函数的名称和参数等；而将具体实现隐藏起来。客户端只要面对最终的结果：对象所要暴露出来的函数。

         Implementation(实现)：核心的逻辑实现。

         将接口与实现分离开来，就可以更改具体的实现而不修改客户端的代码。

         RMI-IIOP使用了这种思想。在网络中传输的都必须是接口，而不是具体的实现，直接调用实际的类是不可能的。也就是说，必须要建立一个暴露远程对象的接口称之为远程接口（remote interface），这个接口必须继承于java.rmi.Remote。

         RMI-IIOP通过接口技术实现了一种“本地/远程透明”的机制，要实现这种机制，它用到了两个接口：

       Stub: 他接受客户端对远程对象的调用，然后代理(delegate)调用到服务器端，这就使得远程调用就像本地调用一样。你可以想象stub就像一个对象的指路人（placeholder），她知道怎样在网络上找到真正的对象，因为调用的是本地的stub，所以很多网络上很繁琐的问题就被隐藏起来了。

       Skeleton: 接收网络上的或者来自stub的调用并且代理这些调用到实际实现的对象，她就像stub一样为远程对象提供一个方便的途径将网络问题隐藏起来。

       J2ee服务器都带有能生成stub和skeleton的自动化工具，如：sun的j2ee server带有的工具：rmic。

       注：我个人认为stub和skeleton这两个接口肯定对应着两个具体实现，这两个具体的实现作的工作应该就是在客户端和服务器端之间建立一个socket连接，通过底层的数据通讯达到对象方法调用的目的。

         对象序列化和参数传递

         Stub和skeleton的大部分工作都是处理参数。

         当使用RMI-IIOP调用远程对象时所有的参数都是值传递的，如果进行值传递的话就会引起很多问题，如：一个对象内部还引用另外一个对象，对方机器的内存编址方式与本机不同，甚至一些引用的对象根本不能传输到在远程机器上等等，要解决这些问题就要使用java的对象序列化机制。

         对象序列化：是将对象转化为字节流的形式。想要将一个对象序列化，很简单，只要让这个对象实现java.lang.Serializable接口即可，这个接口没有定义任何方法，它只是一个标记接口(marker interface)，他标记了你的对象可以被序列化和反序列化。可以重载writeObject()和readObject()来自定义序列化方式。

         序列化的规则：

       基本类型（int，long，float等）可以自动被序列化；

       如果不想让某个对象被序列化，可以在对象前面加transient关键字；否则，如果这个对象既没有加入transient关键字也没有实现Serializable接口，在序列化的时候会报错。

         对象所引用的子对象，只要满足以上规则，则会被java自动序列化。

         哪些情况下应该将一个对象标记为transient呢？

       对象太大；

       对象含有一些不能传输到远端机器的资源，如数据库连接或socket连接等；

       对象中包含一些诸如密码等敏感信息。

         对象序列化是一个重量级(heavyweight)的操作。

         RMI-IIOP和对象序列化

       如果必须进行引用传递，那么这个参数必须是一个远程对象，而且在实际调用中，参数实际上会被替换为这个远程对象的stub，所以本质上还是值传递。

         如果想要client能够找到server，那么server必须将自己发布出来，这就要使用到另外一套机制：JNDI。

2.         JNDI（Java Naming and Directory Interface）

         JNDI是j2ee的一套API，提供一个标准的接口用来定位系统中的资源。

         基本概念：

         Naming service: 非常像一个电话接线员，你不必记得电话号码而只要提供对方的名字即可。它主要起两个作用：

       将名字与对象相关联起来，这称之为绑定binding。这就像电话公司将用户名和电话号码对应起来一样。

       提供一种使用名称就可以定位（look up）到对象的机制。

         Directory service: 是一种被扩展和加强了的naming service，可以像目录那样来操作属性。一个目录是一组连接在一起的目录系统，他有树状的层次继承关系，他在某些方面很像数据库系统。

         Aotomic name: 是一个简单的，基本的，独立的名称；

         Compound name: 是零个或多个原子名称聚合在一起的名称；

         Binding: 名称与对象之间的关联；

         Context: 是包含零个或多个binding的对象；

         Naming system: 是一组连通了的context；

         Namingspace: 是naming system中包含的所有名称 (names)；

         Composite name: 是使用到多个naming system的复合名称；

         现在有很多提供目录服务的产品，但是他们之间并不兼容，所以sun提供了类似JDBC的JNDI接口来统一管理这些产品，它的优点有：

         只要学习一种API就可以操作所有的目录系统；

         将目录系统的具体实现细节隐藏起来；

         可以使用JNDI从目录系统中读取java对象；

         可以将不同类型的目录系统连接在一起。

         在j2EE中JNDI有以下作用：

         得到JTA中的user transaction接口；

         连接到资源工厂，如：JDBC或jms驱动等；

         让一个bean能够找到其他的bean。

         JNDI的结构两大块：

         Client API: 用来操作目录的函数，这些函数对于所有目录系统都是一样的，我们主要在学习JNDI时其实主要是学习这些函数。就像学习JDBC中的Connection，Statement，Resultset一样；

         Service provider interface(SPI): 这是提供给目录系统厂商开发的接口。它独立于各个厂家但是也可以像JDBC那样提供不同产品的特殊功能。

         进入一个目录系统的起始点叫做Initial Context，可以用Initial Context Facotry用来创建Initial Context，这个工厂类大部分情况下就是JNDI的驱动。

         在j2ee环境下，如果想得到一个initial context，必须提供一些信息：

         j2ee server 的IP 地址和端口号；

         JNDI树的起始点；

         能够操作j2ee server 的用户信息。

         注意：一些j2ee server在创建一个Initial context时需要比较长的时间，所以应该在创建一个Initial context之后将其缓存起来。

         可以通过以下方式得到Initial context：

javax.naming.Context cxt = new javax.naming.InitialContext(System.getProperties());

java  -Djava.naming.factory.initial=com.sun.JNDI.fscontext.RefFSContextFactory -Djavax.naming.provider.url=file:c:/

第一个-D之后的参数指定创建Initial context的工厂类；第二个参数指定目录树的入口点。

         得到Initial context之后，就可以进行JNDI操作，常用的操作有：

         List(): 从当前的context中得到数据；

         Lookup(): 从当前context移动到另外一个context；也可以查找绑定在树上的对象；

         Bind(): 绑定对象到JNDI树，其实就是在树中写入数据。