|  |
| --- |
|  |
| **EJB的存根和骨架的工作原理** |
|  |
|  |
| **一、RMI工作原理**  RMI的本质就是实现在不同JVM之间的调用,它的实现方法就是在两个JVM中各开一个Stub和Skeleton，二者通过socket通信来实现参数和返回值的传递。  有关RMI的例子代码网上可以找到不少，但绝大部分都是通过extend the interface java.rmi.Remote实现，已经封装的很完善了，不免使人有雾里看花的感觉。下面的例子是我在《Enterprise JavaBeans》里看到的，虽然很粗糙，但很直观，利于很快了解它的工作原理。  1、定义一个Person的接口，其中有两个business method, getAge() 和getName()  代码：   |  | | --- | | public interface Person {      public int getAge() throws Throwable;      public String getName() throws Throwable;  } |   2、Person的实现PersonServer类  代码：   |  | | --- | | public class PersonServer implements Person {      int age;      String name;      public PersonServer(String name, int age) {          this.age = age;          this.name = name;      }      public int getAge() {          return age;      }      public String getName() {          return name;      }  } |   3、好，我们现在要在Client机器上调用getAge()和getName()这两个business method，那么就得编写相应的Stub(Client端)和Skeleton(Server端)程序。这是Stub的实现：  代码：   |  | | --- | | //存根（stub）的实现  import java.io.ObjectOutputStream;  import java.io.ObjectInputStream;  import java.net.Socket;  public class Person\_Stub implements Person {      Socket socket;      public Person\_Stub() throws Throwable {          // connect to skeleton          socket = new Socket("computer\_name", 9000);      }      public int getAge() throws Throwable {          // pass method name to skeleton          ObjectOutputStream outStream =              new ObjectOutputStream(socket.getOutputStream());          outStream.writeObject("age");          outStream.flush();          ObjectInputStream inStream =              new ObjectInputStream(socket.getInputStream());          return inStream.readInt();      }      public String getName() throws Throwable {          // pass method name to skeleton          ObjectOutputStream outStream =              new ObjectOutputStream(socket.getOutputStream());          outStream.writeObject("name");          outStream.flush();          ObjectInputStream inStream =              new ObjectInputStream(socket.getInputStream());          return (String)inStream.readObject();      }  } |   注意，Person\_Stub和PersonServer一样，都implements Person。它们都实现了getAge()和getName()两个business method，不同的是PersonServer是真的实现，Person\_Stub是建立socket连接，并向Skeleton发请求，然后通过Skeleton调用PersonServer的方法，最后接收返回的结果。  4、骨架（Skeleton）的实现  代码：   |  | | --- | | import java.io.ObjectOutputStream;  import java.io.ObjectInputStream;  import java.net.Socket;  import java.net.ServerSocket;  public class Person\_Skeleton extends Thread {      PersonServer myServer;      public Person\_Skeleton(PersonServer server) {          // get reference of object server          this.myServer = server;      }      public void run() {          try {              // new socket at port 9000              ServerSocket serverSocket = new ServerSocket(9000);              // accept stub's request              Socket socket = serverSocket.accept();              while (socket != null) {                  // get stub's request                  ObjectInputStream inStream =                      new ObjectInputStream(socket.getInputStream());                  String method = (String)inStream.readObject();                  // check method name                  if (method.equals("age")) {                      // execute object server's business method                      int age = myServer.getAge();                      ObjectOutputStream outStream =                          new ObjectOutputStream(socket.getOutputStream());                      // return result to stub                      outStream.writeInt(age);                      outStream.flush();                  }                  if(method.equals("name")) {                      // execute object server's business method                      String name = myServer.getName();                      ObjectOutputStream outStream =                          new ObjectOutputStream(socket.getOutputStream());                      // return result to stub                      outStream.writeObject(name);                      outStream.flush();                  }              }          } catch(Throwable t) {              t.printStackTrace();              System.exit(0);          }      }      public static void main(String args []) {          // new object server          PersonServer person = new PersonServer("Richard", 34);          Person\_Skeleton skel = new Person\_Skeleton(person);          skel.start();      }  } |   Skeleton类extends from Thread，它长驻在后台运行，随时接收client发过来的request。并根据发送过来的key去调用相应的business method。  #p#  5、最后一个，Client的实现  代码：   |  | | --- | | public class PersonClient {      public static void main(String [] args) {          try {              Person person = new Person\_Stub();              int age = person.getAge();              String name = person.getName();              System.out.println(name + " is " + age + " years old");          } catch(Throwable t) {              t.printStackTrace();          }      }  } |   Client的本质是，它要知道Person接口的定义，并实例一个Person\_Stub，通过Stub来调用business method，至于Stub怎么去和Server沟通，Client就不用管了。  注意它的写法：   |  | | --- | | Person person = new Person\_Stub(); |   而不是   |  | | --- | | Person\_Stub person = new Person\_Stub(); |   为什么？因为要面向接口编程嘛，呵呵！  //RMI实质上就是生成2个类stub，skeleton来进行参数和返回值的传递，采用值传递方式 //类似于以前写的聊天室程序，被传递的对象应实现java.io.Serializable接口  **二、Websphere实现**  **EJB类一览**  这里结合WebSphere来讲讲各个类的调用关系吧！  假定我们要创建一个读取User信息的SessionBean，需要我们写的有3个文件：  1、UserServiceHome.java  Home接口  2、UserService.java  Remote接口  3、UserServiceBean.java  Bean实现  WSAD最终会生成10个class。其它7个是什么呢？我们一个一个数过来。  4、\_UserServiceHome\_Stub.java  这个当然就是Home接口在Client端(动态加载)的Stub类了，它implements UserServiceHome。  5、\_EJSRemoteStatelessUserServiceHome\_a940aa04\_Tie.java  Home接口在Server端的Skeleton类，"a940aa04"应该是随机生成的，所有其他的相关class名里都会有这个标志串，Tie是Corba对Skeleton的叫法。  6、EJSRemoteStatelessUserServiceHome\_a940aa04.java  Home接口在Server端的实现，当然，它也implements UserServiceHome。  7、EJSStatelessUserServiceHomeBean\_a940aa04.java  由#6调用，create \_UserService\_Stub。(为什么#6不能直接create \_UserService\_Stub呢？后面再讲。)  8、\_UserService\_Stub.java  Remote接口在Client端(动态加载)的Stub类。它implements UserService。  9、\_EJSRemoteStatelessUserService\_a940aa04\_Tie.java  Remote接口在Server端的Skeleton类。  10、EJSRemoteStatelessUserService\_a940aa04.java  Remote接口在Server端的实现，当然，它也implements UserService。并且，它负责调用UserServiceBean——也就是我们所写的Bean实现类——里面的business method。  那么，各个类之间的调用关系到底是怎么样的呢？简单的说，就是两次RMI循环。       **第一个RMI循环**  先来看看Client端的程序是怎么写的：  代码：   |  | | --- | | try {      InitialContext ctx = new InitialContext();      //第一步      UserServiceHome home =          (UserServiceHome) PortableRemoteObject.narrow(              ctx.lookup(JNDIString),              UserServiceHome.class);      //home: \_UserServiceHome\_Stub      System.out.println(home.toString());      //第二步      UserService object = home.create();      //ojbect: \_UserService\_Stub      System.out.println(object.toString());      //第三步      int userId = 1;      UserInfo ui = object.getUserInfo(userId);  } |   在第一步之后，我们得到了一个UserServiceHome(interface)定义的对象home，那么，home到底是哪个class的instance呢？用debug看一下，知道了home原来就是\_UserServiceHome\_Stub的实例。  #p#  从第二步开始，就是我们的关注所在，虽然只有简单的一行代码，   |  | | --- | | UserService object = home.create(); |   但是他背后的系统是怎么运做的呢？我们进入代码来看吧！  1、调用home.create()  代码：   |  | | --- | | UserServiceHome home;  UserService obj = home.create(); |   2、实际是调用\_UserServiceHome\_Stub.create()，在这个方法里面，Stub向Skeleton发送了一个create的字串：  代码：   |  | | --- | | org.omg.CORBA.portable.OutputStream out = \_request("create", true);  in = (org.omg.CORBA\_2\_3.portable.InputStream)\_invoke(out); |   3、Server端的Skeleton接收Stub发来的request，并调用相应的方法：   代码：   |  | | --- | | \_EJSRemoteStatelessUserServiceHome\_a940aa04\_Tie.\_invoke() {      ......      switch (method.length()) {          case 6:              if (method.equals("create")) {                  return create(in, reply);              }          ......      }  } |   代码：   |  | | --- | | \_EJSRemoteStatelessUserServiceHome\_a940aa04\_Tie.create() {      EJSRemoteStatelessUserServiceHome\_a940aa04 target = null;      result = target.create();      org.omg.CORBA.portable.OutputStream out = reply.createReply();      Util.writeRemoteObject(out,result);      return out;  } |   4、Skeleton调用的是UserServiceHome的Server端实现类的create方法   代码：   |  | | --- | | EJSRemoteStatelessUserServiceHome\_a940aa04.create() {      UserService \_EJS\_result;      \_EJS\_result = EJSStatelessUserServiceHomeBean\_a940aa04.create();  } |   5、#4又调用EJSStatelessUserServiceHomeBean\_a940aa04.create()  代码：   |  | | --- | | UserService result = super.createWrapper(new BeanId(this, null)); |   至此，我们终于结束了第一个RMI循环，并得到了Remote接口UserService的Stub类\_UserService\_Stub，就是#5里面的result。  这里有一个问题，为什么#4不直接create \_UserService\_Stub，而又转了一道#5的手呢？因为#4 extends from EJSWrapper，它没有能力create Stub，因此必须借助#5，which extends from EJSHome，这样才可以生成一个Stub。如果不是为了生成这个Stub，应该可以不走#5这一步。  **第二个RMI循环**  OK, now we got the object which is instanceOf \_UserService\_Stub, and implements UserService  现在我们的Client端走到第三步了：   |  | | --- | | UserInfo ui = object.getUserInfo(userId); |   继续看代码，开始第二个RMI循环：  1、调用object.getUserInfo()  代码：   |  | | --- | | UserService object;  object.getUserInfo(userId); |   2、实际是调用\_UserService\_Stub.getUserInfo(int arg0)，在这个方法里面，Stub向Skeleton发送了一个getUserInfo的字串和arg0这个参数：  代码：   |  | | --- | | org.omg.CORBA.portable.OutputStream out = \_request("getUserInfo", true);  out.write\_long(arg0);  in = (org.omg.CORBA\_2\_3.portable.InputStream)\_invoke(out); |   3、Server端的Skeleton接收Stub发来的request，并调用相应的方法：  代码：   |  | | --- | | \_EJSRemoteStatelessUserService\_a940aa04\_Tie.\_invoke() {      switch (method.charAt(5))      {          case 83:              if (method.equals("getUserInfo")) {                  return getUserInfo(in, reply);              }          ......      }  }  \_EJSRemoteStatelessUserService\_a940aa04\_Tie.getUserInfo() {      EJSRemoteStatelessUserService\_a940aa04 target = null;      int arg0 = in.read\_long();      UserDTO result = target.getUserInfo(arg0);      org.omg.CORBA\_2\_3.portable.OutputStream out = reply.createReply();      out.write\_value(result,UserDTO.class);      return out;  } |   #p#  4、Skeleton调用的是UserService的Server端实现类的getUserInfo方法  代码：   |  | | --- | | EJSRemoteStatelessUserService\_a940aa04.getUserInfo() {      UserServiceBean \_EJS\_beanRef = container.preInvoke(this, 0, \_EJS\_s);      \_EJS\_result = \_EJS\_beanRef.getUserInfo(id);  } |   最后的最后，#4终于调用了我们写的UserServiceBean里的getUserInfo方法，这才是我们真正想要去做的事情。  至此，第二个RMI循环也终于结束了。        **调用流程图**  回顾一下上面的分析，可以很清晰的看到两次RMI循环的过程，下图(见链接)描述了整个流程：  <http://www.51cto.com/files/uploadimg/20071016/1708590.gif>     |  | | --- | | http://www.51cto.com/files/uploadimg/20071016/1708591.gif | | 图1 |   黄色的1，6，10是程序员要写的，其余是系统生成的。  #1是Home interface, #2和#4都implements了它。  #6是Remote interface, #7和#9都implements了它。  #10是Bean实现。  **三、weblogic实现**  一个远程对象至少要包括4个class文件：远程对象；远程对象的接口；实现远程接口的对象的stub；对象的skeleton这4个class文件。  在EJB中则至少要包括10个class：  **Bean类，特定App Server的Bean实现类**  Bean的remote接口，特定App Server的remote接口实现类，特定App  Server的remote接口的实现类的stub类和skeleton类  Bean的home接口，特定App Server的home接口实现类，特定App  Server的home接口的实现类的stub类和skeleton类  和RMI不同的是，EJB中这10个class真正需要用户编写的只有3个，分别是Bean类和它的remote接口，home接口，至于其它的7个class到底是怎么生成，被打包在什么地方，或者是否需要更多的类文件，会根据不同的App Server表现出比较大的差异，不能一概而论。  拿Weblogic的来说吧！Weblogic的Bean实现类，以及两个接口的Weblogic的实现类是在ejbc的时候被打包到EJB的jar包里面的，这3个class文件可以看到。而home接口和remote接口的Weblogic的实现类的stub类和skeleton类是在EJB被部署到Weblogic的时候，由Weblogic动态生成stub类和Skeleton类的字节码，因此看不到这4个类文件。  对于一次客户端远程调用EJB，要经过两个远程对象的多次RMI循环。首先是通过JNDI查找Home接口，获得Home接口的实现类，这个过程其实相当复杂，首先是找到Home接口的Weblogic实现类，然后创建一个Home接口的Weblogic实现类的stub类的对象实例，将它序列化传送给客户端（注意stub类的实例是在第1次RMI循环中，由服务器动态发送给客户端的，因此不需要客户端保存Home接口的Weblogic实现类的stub类），最后客户端获得该stub类的对象实例（普通的RMI需要在客户端保存stub类，而EJB不需要，因为服务器会把stub类的对象实例发送给客户端）。  客户端拿到服务器给它的Home接口的Weblogic实现类的stub类对象实例以后，调用stub类的create方法，(在代码上就是home.create()，但是后台要做很多事情),于是经过第2次RMI循环，在服务器端，Home接口的Weblogic实现类的skeleton类收到stub类的调用信息后，由它再去调用Home接口的Weblogic实现类的create方法。  在服务端，Home接口的Weblogic实现类的create方法再去调用Bean类的Weblogic实现类的ejbCreate方法，在服务端创建或者分配一个EJB实例，然后将这个EJB实例的远程接口的Weblogic实现类的stub类对象实例序列化发送给客户端。  客户端收到remote接口的Weblogic实现类的stub类的对象实例，对该对象实例的方法调用（在客户端代码中实际上就是对remote接口的调用），将传送给服务器端remote接口的Weblogic实现类的skeleton类对象，而skeleton类对象再调用相应的remote接口的Weblogic实现类，然后remote接口的Weblogic实现类再去调用Bean类的Weblogic实现类，如此就完成一次EJB对象的远程调用。  看了一遍文章，感觉还是没有说太清楚，既然写了帖子，就想彻底把它说清楚。  先拿普通RMI来说，有4个class，分别是远程对象，对象的接口，对象的stub类和skeleton类。而对象本身和对象的stub类同时都实现了接口类。而我们在客户端代码调用远程对象的时候，虽然在代码中操纵接口，实质上是在操纵stub类，例如：  **接口类：Hello**  远程对象：Hello\_Server  **stub类：Hello\_Stub**  **skeleton类：Hello\_Skeleton**  客户端代码要这样写：   |  | | --- | | Hello h = new Hello\_Stub(); h.getString(); |   我们不会这样写：   |  | | --- | | Hello\_Stub h = new Hello\_Stub(); h.getString(); |   因为使用接口适用性更广，就算更换了接口实现类，也不需要更改代码。因此客户端需要Hello.class和Hello\_Stub.class这两个文件。但是对于EJB来说，就不需要Hello\_Stub.class，因为服务器会发送给它，但是Hello.class文件客户端是省不了的，必须有。表面上我们的客户端代码在操纵Hello，但别忘记了Hello只是一个接口，抽象的，实质上是在操纵Hello\_Stub。  #p#  拿Weblogic上的EJB举例子，10个class分别是：  Bean类：HelloBean （用户编写） Bean类的Weblogic实现类：HelloBean\_Impl （EJBC生成） Home接口：HelloHome （用户编写） Home接口的Weblogic实现类 ((Hello Bean))\_HomeImpl（EJBC生成） Home接口的Weblogic实现类的stub类 ((Hello Bean))\_HomeImpl\_WLStub（部署的时候动态生成字节码） Home接口的Weblogic实现类的skeleton类 ((Hello Bean))\_HomeImpl\_WLSkeleton（部署的时候动态生成字节码） Remote接口： Hello （用户编写） Remote接口的Weblogic实现类 ((Hello Bean))\_EOImpl（EJBC生成） Remote接口的Weblogic实现类的stub类 ((Hello Bean))\_EOImpl\_WLStub（部署的时候动态生成字节码） Remote接口的Weblogic实现类的skeleton类 ((Hello Bean))\_EOImpl\_WLSkeleton（部署的时候动态生成字节码）  客户端只需要Hello.class和HelloHome.class这两个文件。  ((Hello Home)) home = (Home) ((Portable Remote Object)).narrow(ctx.lookup("Hello"), ((Hello Home)).class);  这一行代码是从JNDI获得Home接口，但是请记住！接口是抽象的，那么home这个对象到底是什么类的对象实例呢？很简单，用toString()输出看一下就明白了，下面一行是输出结果：   |  | | --- | | ((Hello Bean))\_HomeImpl\_WLStub@18c458 |   这表明home这个通过从服务器的JNDI树上查找获得的对象实际上是HelloBean\_HomeImpl\_WLStub类的一个实例。  接下来，客户端代码：   |  | | --- | | Hello h = home.create() |   同样Hello只是一个抽象的接口，那么h对象是什么东西呢？打印一下：   |  | | --- | | ((Hello Bean))\_EOImpl\_WLStub@8fa0d1 |   原来是HelloBean\_EOImpl\_WLStub的一个对象实例。  用这个例子来简述一遍EJB调用过程：  首先客户端JNDI查询，服务端JNDI树上Hello这个名字实际上绑定的对象是HelloBean\_HomeImpl\_WLStub，所以服务端将创建HelloBean\_HomeImpl\_WLStub的一个对象实例，序列化返回给客户端。  于是客户端得到home对象，表面上是得到HelloHome接口的实例，实际上是进行了一次远程调用得到了HelloBean\_HomeImpl\_WLStub类的对象实例，别忘记了HelloBean\_HomeImpl\_WLStub也实现了HelloHome接口。  然后，home.create()实质上就是HelloBean\_HomeImpl\_WLStub.create()，该方法将发送信息给HelloBean\_HomeImpl\_WLSkeleton，而HelloBean\_HomeImpl\_WLSkeleton接受到信息后，再去调用HelloBean\_HomeImpl的create方法，至此完成第1次完整的RMI循环。  注意在这次RMI循环过程中，远程对象是HelloBean\_HomeImpl，远程对象的接口是HelloHome，对象的stub是HelloBean\_HomeImpl\_WLStub，对象的skeleton是HelloBean\_HomeImpl\_WLSkeleton。  然后HelloBean\_HomeImpl再去调用HelloBean\_Impl的ejbCreate方法，而HelloBean\_Impl的ejbCreate方法将负责创建或者分配一个Bean实例，并且创建一个HelloBean\_EOImpl\_WLStub的对象实例。  这一步比较有趣的是，在前一步RMI循环中，远程对象HelloBean\_HomeImpl在客户端有一个代理类HelloBean\_HomeImpl\_WLStub，但在这一步，HelloBean\_HomeImpl自己却充当了HelloBean\_Impl的代理类，只不过HelloBean\_HomeImpl不在客户端，而是在服务端，因此不进行RMI。  然后HelloBean\_EOImpl\_WLStub的对象实例序列化返回给客户端，这一步也很有趣，上次RMI过程，主角是HelloBean\_HomeImpl和它的代理类HelloBean\_HomeImpl\_WLStub，但这这一次换成了HelloBean\_EOImpl和它的代理类HelloBean\_EOImpl\_WLStub来玩了。   |  | | --- | | Hello h = home.create();h.helloWorld(); |   假设Hello接口有一个helloWorld远程方法，那么表面上是在调用Hello接口的helloWorld方法，实际上是在调用HelloBean\_EOImpl\_WLStub的helloWorld方法。  然后HelloBean\_EOImpl\_WLStub的helloWorld方法将发送信息给服务器上的HelloBean\_EOImpl\_WLSkeleton，而HelloBean\_EOImpl\_WLSkeleton收到信息以后，再去调用HelloBean\_EOImpl的helloWorld方法。至此，完成第2次完整的RMI循环过程。  在刚才HelloBean\_EOImpl是作为远程对象被调用的，它的代理类是HelloBean\_EOImpl\_WLStub，但现在HelloBean\_EOImpl要作为HelloBean\_Impl的代理类了。现在HelloBean\_EOImpl去调用HelloBean\_Impl的helloWorld方法。注意！HelloBean\_Impl继承了HelloBean，而HelloBean中的helloWorld方法是我们亲自编写的代码，现在终于调用到了我们编写的代码了！  至此，一次EJB调用过程终于完成。在整个过程中，服务端主要要调用的类是HelloBean\_Impl， HelloBean?\_HomeImpl，HelloBean\_HomeImpl\_WLSkeleton，HelloBean\_EOImpl，HelloBean\_EOImpl\_WLSkeleton。客户端主要调用的类是HelloBean\_HomeImpl\_WLStub，HelloBean\_EOImpl\_WLStub，这两个类在客户端代码中并不会直接出现，出现在代码中的类是他们的接口HelloHome和Hello，因此客户端需要这两个接口文件，而Stub是服务器传送给他们的。  **四、理解体会**  简单讲，就是为了适应分布式开发的需要。  首先，回到我最后给出的流程图。  <http://www.51cto.com/files/uploadimg/20071016/1708592.gif>     |  | | --- | | http://www.51cto.com/files/uploadimg/20071016/1708593.gif | | 图2 |   Client端最原始的冲动，肯定是能直接调用#10.UserServiceBean就爽了。那么第一个问题来了，Client和Server不在一个JVM里。  这好办，我们不是有RMI吗？好，这个问题就这么解决了：  1. UserServiceBeanInterface.getUserInfo()  2. UserServiceBeanStub  3. UserServiceBeanSkeleton  4. UserServiceBean  用着用着，第二个问题来了，UserServiceBean只有人用，没人管理，transaction logic, security logic, bean instance pooling logic这些不得不考虑的问题浮出水面了。  OK，我们想到用一个delegate，EJBObject，来进行所有这些logic的管理。client和EJBObject打交道，EJBObject调用UserServiceBean。  注意，这个EJBObject也是一个Interface，#6.UserService这个interface正是从它extends而来。并且EJBObject所管理的这些logic，正是AppServer的一部分。  现在的流程变为了：  EJBObject  1. UserService.getUserInfo()  2. UserServiceStub  3. UserServiceSkeleton  4. UserServiceImp  5. UserServiceBean  这已经和整幅图里的#6, #7, #8, #9, #10一一对应了。  现在能满足我们的需求了吗？不，第三个问题又来了：  既然是分布式开发，那么我当然没理由只用一个Specified Server，我可能需要用到好几个不同的Server，而且EJBObject也需要管理呀！  OK，为了适应你的需要，我们还得加再一个HomeObject，首先它来决定用哪个Server(当然，是由你用JNDI String设定的)，其次，它来管理EJBObject。  注意，这个EJBHome也是一个Interface，#1.UserServiceHome这个interface正是从它extends而来。并且EJBHome管理EJBObject的logic，也是AppServer的一部分。  现在的调用次序是：  1. EJBHome.create()  2. EJBHomeStub  3. EJBHomeSkeleton  4. EJBHomeImp(EJSWrapper)  5. EJSHome  得到EJBObject。  6. UserService.getUserInfo()  7. UserServiceStub  8. UserServiceSkeleton  9. UserServiceImp  10. UserServiceBean  现在已经完全和流程图的调用顺序一致了。  //EJB的基础是RMI IIOP，原理并不是很难，关键是实现起来比较绕，一个简单的功能要用10个（或更多）类来实现，但每一个都不是多余的。  //EJB的这种模式（或说RMI）完全屏蔽了底层的网络，并很好的实现了对业务代码的保护。  【责任编辑：[火凤凰](mailto:sunsj@51cto.com) TEL：（010）68476606】 |
|  |

1.EJB事务分类

* 容器管理事务的回滚

        你能够在任何企业Bean中使用容器管理事务：会话Bean、实体Bean或者 Message-driven Bean。容器管理事务可以分两种：

       第一，如果产生一个系统异常，容器将自动回滚该事务。

       第二，通过调用EJBContext接口SetRollbackOnly方法，Bean方法通知容器回滚该事务。如果Bean抛出一个应用异常，事务将不会自动回滚，但可以调用SetRollbackOnly回滚。

* bean管理事务

     在一个Bean管理事务中，会话Bean或者Message-driven Bean是用代码显式设置事务界线的。实体Bean不能使用Bean管理事务，只能使用容器管理的事务。虽然容器管理事务Bean需要较少的代码，但它也有一个局限：方法执行时，它只能关联一个事务或不关联任何事务。如果这个局限使你Bean编码困难，你应该考虑使用Bean管理事务

     其中bean管理事务可以分为JDBC事务

**JDBC 事务**

　　JDBC事务通过DBMS事务管理器来控制。你可能会为了使用会话Bean中的原有代码而采用JDBC事务将这些代码封装到一个事务中。使用JDBC事务，要调用java.sql.Connection接口的commit和rollback方法。事务启动是隐式的。一个事务的从最近的提交、回滚或连接操作后的第一个SQL的语句开始。（这个规则通常是正确的，但可能DBMS厂商的不同而不同）

**JTA 事务**

　　JTA是Java Transaction API 的缩写。这些API 允许你用独立于具体的事务管理器实现的方法确定事务界限。J2EE SDK 事务管理器通过Java事务服务（Java Transaction Service, JTS）实现。但是你的代码并不直接调用JTS中的方法，而是调用JTA方法来替代，JTA方法会调用底层的JTS实现。

　　JTA事务被J2EE 事务管理器管理。你可能需要使用一个JTA事务，因为它能够统一操作不同厂商的数据库。一个特定DBMS的事务管理器不能工作在不同种类的数据库上。然而J2EE事务管理器仍然有一个限制——它不支持嵌套事务。就是说，它不能在前一个事务结束前启动另一个事务。

　   要自己确定事务界限，可以调用javax.transaction.UserTransaction接口的begin、commit和rollback方法来确定事务界限（该接口只能在SessionBean中使用，实体Bean不允许使用用户自定义的）。

2.事务属性

Required   
如果客户端正在一个运行的事务中调用一个企业Bean的方法，这个方法就在这个客户端的事务中执行。如果客户端不关联一个事务，这个容器在运行该方法前开始一个新的事务。

Required属性在许多事务环境中可以很好的工作，因此你可以把它作为一个默认值，至少可以在早期开发中使用。因为事务的属性是在部署描述符中声明的，在以后的任何时候修改它们都很容易。

RequiresNew   
如果客户端在一个运行的事务中调用企业Bean的方法，容器的步骤是：

1．挂起客户端的事务

2．开始一个新的事务

3．代理方法的调用

4．方法完成后重新开始客户端的事务

如果客户端不关联一个事务，容器运行这个方法以前同样开始一个新的事务。如果你想保证该方法在任何时候都在一个新事物中运行，使用RequiresNew属性。

Mandatory  
如果客户端在一个运行的事务中调用企业Bean的方法，这个方法就在客户端的事务中执行。如果客户端不关联事务，容器就抛出TransactionRequiredException 异常。

如果企业Bean的方法必须使用客户端的事务，那么就使用Mandatory属性。

NotSupported   
如果客户端在一个运行的事务中调用企业Bean的方法，这个容器在调用该方法以前挂起客户端事务。方法执行完后，容器重新开始客户端的事务。

如果客户端不关联事务，容器在方法运行以前不会开始一个新的事务。为不需要事务的方法使用NotSupported属性。因为事务包括整个过程，这个属性可以提高性能。

Supports

如果客户端在一个运行的事务中调用企业Bean的方法，这个方法在客户端的事务中执行，如果这个客户端不关联一个事务，容器运行该方法前也不会开始一个新的事务。因为该属性使方法的事务行为不确定，你应该谨慎使用Supports属性。

Never

如果客户端在一个运行的事务中调用企业Bean的方法，容器将抛出RemoteException异常。如果这个客户端不关联一个事务，容器运行该方法以前不会开始一个新的事务。

例子:

假设有两个EJB,EJB1和EJB2。EJB1的method1中n次调用EJB2的method2.典型代码如下:

method1{

  for(int i=0;i<n;i++){

    EJB2的method2..

}

}

如果在for循环到第j次异常.那么如果method1为Required,method2为Required,则会回滚所有的操作.结果就是

method2 对于i从0－－j的所有数据库操作都回滚.method1也回滚.因为都是同一个事务

如果在for循环到第j次一场,那么如果method1为Required,method2为RequirsNew,则不会回滚所有的操作.结果就是method2对于i从0－－j的所有数据库操作都不回滚.但method1回滚.因为对于不同的i值.都为新的事务。

如果想让程序执行i从0－－n次.那么需要记录执行正常和不正常的情况.此时只要try,catch 其中EJB的method2就可.在catch代码中记录异常情况.这样因为没有再throw异常给method1。代码可以一直执行n次。达到记录正常和异常的记录。当然对method1不使用事务,method2使用required也可以达到目的.对method1不使用事务的简单方法就是 设置其为Bean管理EJB,并在代码中不手动开启事务