JacORB1.4 编程指南

Gerald Brose, Nicolas Noffke, Sebastian Müller
Institut für Informatik
Freie Universität Berlin, Germany
{brose,noffke,semu}@inf.fu-berlin.de

Revision : 1.19

March 20, 2002

中文翻译：hlstudio(hlstudio@sina.com) ，cocia(cocia@163.com)(第 7 章)
校稿：allen(allen@huihoo.com),fat1(xwcheng@sina.com)
2002.08

目录

1 前言 4
2 安装 JacORB 5
2.1 获取 JacORB 5
2.2 安装 JacORB 5
2.2.1 Ant 和 build.xml 5
2.2.2 配置 5
3 编程起步 9
3.1 JacORB 开发步骤 9
3.2 IDL 接口定义 9
3.3 生成 Java 类 10
3.4 实现接口 10
3.5 编写服务器 11
3.6 编写客户端 12
3.7 桥接方式 13
4 JacORB 命名服务 15
4.1 运行命名服务 15
4.2 访问命名服务 16
4.3 构造层次命名空间 16
4.4 命名管理器 16
5 服务端：POA 及线程 18
5.1 POA 18
5.2 线程 18
6 实现仓库 19
6.1 概述 19
6.2 使用 JacORB 的实现仓库 19
6.3 服务迁移 20
6.4 安全考虑 21
7 Any Value 的动态管理 22
7.1 概述 22
7.2 接口 22
7.3 用法限制 22
7.4 创建一个 DynAny 对象 23
7.5 访问 DynAny 对象的值 24
7.6 传送 DynAny 对象的值 24
7.7 构造类型 25
7.7.1 DynEnum 25
7.7.2 DynStruct 25
7.7.3 DynUnion 25
7.7.4 DynSequence 25
7.7.5 DynArray 25
7.8 Any 和 DnyAny 对象之间的转换 26
7.9 更多例子 26
8 接口仓库 27
8.1 接口仓库中的类型信息 27

8.2 接口仓库设计 27
8.3 使用接口仓库 28
9 JacORB 的小程序代理 30
9.1 使用小程序代理 30
9.1.1 使用步骤 30
9.1.2 程序属性 30
9.1.3 小程序代理和 Netscpe/IE，AppletViewer 31
9.1.4 示例 31
9.2 通过防火墙使用 JacORB 31
9.2.1 小程序代理 31
9.2.2 HTTP 隧道 32
9.2.3 小程序代理和 HTTP 隧道 32
9.2.4 总结 32
10 基于 SSL 的 IIOP 33
10.1 重编译 JacORB 安全库 33
10.2 配置 IAIK 33
10.2.1 设置 IAIK 证书库 33
10.2.2 一步一步生成证书 35
10.3 配置 SSL 的属性 35
10.3.1 客户端配置 36
10.3.2 服务器端配置 36
11 双向 GIOP 37
11.1 设置双向 GIOP 37
11.1.1 设置 ORB 初始属性 37
11.1.2 创建双向策略 37
11.2 验证双向 GIOP 已经使用 37
11.3 TAO 协同性 38
12 可移植拦截器-PI 39
13 JacORB 实用程序 40
13.1 IDL 40
13.2 ns 40
13.3 nmg 41
13.4 lsns 41
13.5 dior 41
13.6 pingo 41
13.7 ir 41
13.8 qir 42
13.9 ks 42
14 配置清单 43
14.1 ORB 的配置 43
14.2 POA 的配置 44
14.3 实现仓库的配置 45
14.4 安全的配置 45
15 缺陷和反馈 47
16 附录 48

1 前言

本文主要介绍 JacORB 的分布式应用开发。JacORB 是免费的 Java 对象请求代理系统，附带全部源代码，包含大部分 CORBA 对象服务的实现，以及大量的代码实例。本文不是 CORBA 的一般性介绍，如果要看 CORBA 方面的介绍，请查阅附录的参考书目[BVD01]及[HV99]。本文所描述的 JacORB 版本为 1.4Beta1，同时介绍一些安装和使用 JacORB 的技巧。

2 安装 JacORB

本章的主要内容是获取和安装 JacORB，以及相关包的主要内容。

2.1 获取 JacORB

JacORB 可以在 JacORB 的主页 http://www.jacorb.org 下载，也可以通过匿名方式访问 FTP 服务器 ftp.inf.fu-berlin.de 目录 pub/jacorb 来下载。
下载的文件是压缩包，unix 为 tar 格式，windows 下为 zip 格式。要安装 JacORB，先解压缩，会生成一个新的目录 JacORB1_4_beta1。然后将 `JacORB1_4_beta1/lib/jacorb.jar`加入到类路径(classpath)中。如果你想重新编译JacORB，也不要忘记将 `JacORB1_4_beta1/class` 加入类路径，并且将 JacORB1_4_beta1/bin 加到 Path，在 JacORB1_4_beta1/bin 下有一些实用程序。

2.2 安装 JacORB

2.2.1 Ant 和 build.xml

JacORB 可以在所有的 Java 虚拟机上运行。重新编译 JacORB(以及 JacORB 附带的例子)需要安装 Ant，Ant 是基于 XML 的制作管理工具。你可以到 http://jakarta.apache.org/ant下载最新版本的Ant。所有的build.xml都是为这个工具写的。要重新编译JacORB，你只需在安装目录(JacORB1_4_beta1)中输入ant命令即可。在执行ant命令之前，可以执行ant clean来清除上次编译产生的中间文件。
最好使用 JDK1.2 以上的版本的虚拟机来运行 JacORB，因为图形工具如使命名管理器(NameManger)、实现仓库管理器(ImRManger)、接口仓库浏览器(IRBrowser)，以及 SSL 的支持都需要 JDK1.2 以下版本的虚拟机。在使用 SSL 时，你还需要第三方的 SSL 协议实现，JacORB 目前支持以下的实现：
1.IAIKs的实现，包含在IAIK–JCE 2.5或更高版本中，以及SSL的库iSaSiLk3.0。使用此实现你可以访问客户端的证书。
2.Sun 的 JSSE 参考实现，包含在 JDK1.4 中；也可单独从 JDC 中下载。

2.2.2 配置

JacORB 有很多可以设为 Java 属性的配置项。在解释这些基本配置项之前，先让我们看看配置方法。其他一些具体的配置项，如和实现仓库及交易服务相关的选项，在相关的章节讲述。
总体说来，有三种方式可以配置 JacORB。
第一种方式是文件方式。JacORB 寻找和加载名为.jacorb_properties 或 jacorb.properties 文件，系统将在下列位置搜索配置文件：
1.类路径中。
2.运行 JacORB 用户的用户主目录，通过运行 System.getProerty("user.home")获得。如果需要知道用户主目录在哪，可以写一小段 Java 程序去测试一下。
3.当前目录中。
4.JDK 安装的 lib 目录。JDK 的主目录通过运行 System.getProperty("java.home")获得。

系统也按上述顺序搜索，一旦找到一个配置文件，系统将停止搜索，并加载该配置文件。

第二种方式是对象方式。对于和应用相关的配置项，可以在应用初始化时给 ORB.init 传递一个 java.util.Property 对象。这种方式会覆盖使用文件方式设置的配置项。下面的代码片段演示了如何向 ORB.init 传递一个 Property 对象(程序中 args 是命令行参数变量)：

```
java.util.Properties props = new java.util.Properties();
props.setProperty("jacorb.implname","StandardNS");
// use put() under Java 1.1
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
```

第三种方式是参数方式。可以通过设置 Java 虚拟机的系统属性来设置配置项，这些属性必须在调用 ORB.init()之前设置，并且此方式会覆盖前两种方式设置的配置项。系统属性可用 System.setProperty，也可以用命令行参数方式 -D<prop_name>=<prop_value>，这是 Java 程序的命令行参数，也可用于 jaco 脚本。需要注意的是这些参数必须放在类名参数之前，放在类名参数之后的都将被 Java 虚拟机解释为该类的参数，并只被传递到该类的 main 方法中。

如果你想通过命令行设置多个参数，你可以用一个特定的属性 custom.props，此属性值为配置文件。在配置文件中你可以加上任意多的配置项。这里的配置项也会覆盖第一种和第二种方式设置的配置项。例如：通常使用标准的 TCP/IP 连接，但可能有时候想使用 SSL。如果只使用一个配置文件，那么要变更连接方式时就得修改配置文件。如果不想每次都修改配置文件，也可以将不同的参数通过命令行参数传递，但这会导致很长的命令。但如果使用特定的属性去指定文件，就可以把所有其他的配置项都放在该文件中，而只使用一个命令行参数，如：

```
$ jaco -Dcustom.props=ssl_props_MyServer
```

我们现在看一看最基本的配置项，下面是一个示例的配置文件：

```
##
## JacORB 配置项
##
#########################################
#                                       #
# 初始化的参考配置                      #
#                                       #
#########################################
#
# IORs存储的URL (在orb.resolve_initial_service()中使用)
#
#
# ORBInitRef在ORB启动时创建. In the
# cases of the services themselves, this may lead to exceptions being
# displayed (because the services aren't up yet). These exceptions
# are handled properly and cause no harm!
#ORBInitRef.NameService=corbaloc::160.45.110.41:38693/StandardNS/NameSer
ver POA/_root
#ORBInitRef.NameService=file:/c:/NS_Ref
ORBInitRef.NameService=http://www.x.y.z/~user/NS_Ref
#ORBInitRef.TradingService=http://www.x.y.z/~user/TraderRef
# JacORB-specific URLs
jacorb.ImplementationRepositoryURL=http://www.x.y.z/~user/ImR_Ref
jacorb.ProxyServerURL=http://www.x.y.z/~user/Appligator_Ref
##################################
#                                #
```

```
# 调试模式                    #
#                             #
##################################
# use (java) jacorb.util.CAD to generate an appropriate
# verbosity level
# 0 = off
# 1 = important messages and exceptions
# 2 = informational messages and exceptions
# >= 3 = debug-level output (may confuse the unaware user :-)
jacorb.verbosity=1
# where does output go? Terminal is default
#jacorb.logfile=LOGFILEPATH
##################################################
#                                               #
# WARNING: The following properties should      #
# only be edited by the expert user. They       #
# can be left untouched for most cases!         #
#                                               #
##################################################
################################
#                              #
# 基本ORB配置                   #
#                              #
################################
# number of retries if connection cannot directly be established
jacorb.retries=5
# how many msecs. do we wait between retries
jacorb.retry_interval=500
# size of network buffers for outgoing messages
jacorb.outbuf_size=2048
# client-side timeout, set no non-zero to stop blocking
# after so many msecs.
#jacorb.connection.client_timeout=0
# max time a server keeps a connection open if nothing happens
#jacorb.connection.server_timeout=10000
#jacorb.reference_caching=off
...
#########################
#                       #
# POA配置                #
#                       #
#########################
# displays a GUI monitoring tool for servers
jacorb.poa.monitoring=off
# thread pool configuration for request processing
jacorb.poa.thread_pool_max=20
jacorb.poa.thread_pool_min=5
# if set, request processing threads in thePOA
# will run at this priority. If not set or invalid,
# MAX_PRIORITY will be used.
#jacorb.poa.thread_priority=
# size of the request queue, clients will receive Corba.TRANSIENT
# exceptions if load exceeds this limit
jacorb.poa.queue_max=100
...
```

配置项包括网络缓冲区的大小、JacORB 在不能建立连接时的重试次数、在重试前等待的时间间隔。ORBInitRef.NameService 的值是 JacORB 命名服务的 URL。这个 URL 被 ORB 用来定位存储服务对象引用的文件(请参见第 4 章)。

`verbosity` 配置项指定 JacORB 在运行时可以忽略多少诊断输出。除非将 `logfile` 配置

项设定为一个文件，否则系统将向终端输出日志。将 verbosity 置为 0 意味着不输出任何诊断信息，2 为详细诊断信息输出。1 则忽略某些信息，例如忽略连接打开、接受和关闭的信息。如果想可选的输出一些诊断信息，可以使用 jacorb.util.CAD 工具生成一个自定义的输出级别。

配置项 jacorb.poa.monitoring 指明 POA 是否打开一个图形界面显示自身的动态信息，例如有多长的请求队列，有多大的线程池。同时，这个工具也可以用来修改 POA 的状态，从活动到保持，详细说明可以参见第 5 章。

现在就可以在 demo 目录下任一子目录来测试安装。在此目录下有很多 JacORB 的例子。

3 编程起步

在开始讲述例子之前，我们先看一下在 JacORB 中开发 CORBA 应用的基本步骤。我们也按照这个基本步骤来讲述例子。例子请参见 demo/grid 目录，在此目录有一个 build.xml，用以使用 ant 来制作，这样不用每回都手工运行每一开发步骤，当然还是应该搞清楚开发的原理。

本文只是 JacORB 编程的简短介绍，不会涉及 CORBA IDL 的所有细节，建议看看 demo 下的其他例子，这些例子是按照 CORBA IDL 的方式进行组织的。

3.1 JacORB 开发步骤

JacORB 的应用开发一般分为以下五步：

1.写 IDL 接口定义
2.编译 IDL 接口定义生成 Java 类
3.实现步骤 2 中生成的接口
4.写服务器启动类，并注册到 ORB
5.写客户端去获取服务对象引用

3.2 IDL 接口定义

本例实现了一个简单的服务，其接口在 server.idl 中定义。本例所有的源代码都可以在 jacORB1_4_beta1/demo/grid 目录下找到。

下面是 server.idl 文件的内容：

```
// server.idl
// 二维网格的接口定义:
module demo
{
     module grid
     {
          interface MyServer
          {
               typedef fixed <5,2> fixedT;
               readonly attribute short height; // 网络的高度
               readonly attribute short width; // 网络的宽度
               // 设置n*m的网络
               void set(in short n, in short m, in fixedT value);
               // 返回n*m的网络元素
               fixedT get(in short n, in short m);
               exception MyException
               {
                    string why;
               };
               short opWithException() raises( MyException );
          };
     };
};
```

3.3 生成 Java 类

使用如下命令编译此接口文件:

```
$idl -d ../.. server.idl
```

此命令将生成多个 Java 源文件，Java 源文件的生成依照 OMG 标准的 IDL-Java 语言映射定义。如果你对语言映射定义感兴趣，例如 IDL 的语言结构如何映射为 Java 的语言结构，可以到 www.omg.org 上查找相关文档。JacORB 的 IDL 编译使用 CORBA2.3 定义的语言映射，详细说明可参考附录的参考书目[BVD01]。如何使用 idl 接口定义，可参考 demo 目录下的例子。

IDL 编译器产生 Java 接口 MyServer、MyServerOptions，及桩和骨架文件_MyServerStub、MyServerPOA、MyServerPOATie。我们将在后面详细这几个类。

注意 IDL 编译器将会产生一个和 IDL 接口文件中定义模块相对应的目录结构。如果我们不指定-d ../..，将在当前目录下创建 demo/grid 子目录。

如何存放生成的 Java 源文件各有所好，有人喜欢所所有的文件都放在一个地方(可以使用-d 选项)，有人喜欢一个目录存放生成的的源文件，另外一个目录存放 Java 编译产生的 class 文件。

3.4 实现接口

现在我们来实现接口定义中描述的功能。这个 Java 类命名为 gridImpl，除了要实现接口定义中描述的所有功能外，gridImpl 还应该是在前面生成的类 MyServerPOA 的子类。MyServerPOA 中包含接受远程调用和将结果返回给客户的端的相关代码。

你也许注意到当 gridImpl 要继承另外一个类时，这种方式就有局限性。因为 Java 只允许继承一个父类，在后面将讲到用桥接方式解决这个问题。

这是 gridImpl 的源代码，在代码中使用 java.math.BigDecimal 来保存接口定义中的 fixedT 类型。

```
package demo.grid;
/**
* A very simple implementation of a 2-D grid
*/
import demo.grid.MyServerPackage.MyException;
public class gridImpl extends MyServerPOA
{
	protected short height = 31;
	protected short width = 14;
	protected java.math.BigDecimal[][] mygrid;
	public gridImpl()
	{
		mygrid = new java.math.BigDecimal[height][width];
		for( short h = 0; h < height; h++ )
		{
			for( short w = 0; w < width; w++ )
			{
				mygrid[h][w] = new java.math.BigDecimal("0.21");
			}
		}
	}
	public java.math.BigDecimal get(short n, short m)
	{
		if( ( n <= height ) && ( m <= width ) )
			return mygrid[n][m];
```

```
            else
                  return new java.math.BigDecimal("0.01");
      }
      public short height()
      {
            return height;
      }
      public void set(short n, short m, java.math.BigDecimal value)
      {
            if( ( n <= height ) && ( m <= width ) )
                  mygrid[n][m] = value;
      }
      public short width()
      {
            return width;
      }
      public short opWithException()
            throws  demo.grid.MyServerPackage.MyException
      {
            throw new demo.grid.MyServerPackage.MyException("This is only
a test exception,
      }
}
```

3.5 编写服务器

这一步要写一个类来调用 gridImpl 类，并将其注册到 POA，这样远程对象才能通过 MyServer 接口来访问它。以下是这个类的源代码：

```
package demo.grid;
import java.io.*;
import org.omg.CosNaming.*;
public class Server
{
//////    public static void main( String[] args )
      {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            try
            {
                  org.omg.PortableServer.POA poa =
                        org.omg.PortableServer.POAHelper.narrow(
                        orb.resolve_initial_references("RootPOA"));
                  poa.the_POAManager().activate();
                  org.omg.CORBA.Object o = poa.servant_to_reference(new
gridImpl());
                  if( args.length == 1 )
                  {
                        // write the object reference to args[0]
                        PrintWriter ps = new PrintWriter(
                              new FileOutputStream(
                              new File( args[0] )));
                        ps.println( orb.object_to_string( o ) );
                        ps.close();
                  }
                  else
                  {
                        // register with the naming service
                        NamingContextExt nc =
                              NamingContextExtHelper.narrow(
                        orb.resolve_initial_references("NameService"));
                        nc.bind( nc.to_name("grid.example"), o);
```

```
                }
            }
            catch ( Exception e )
            {
                e.printStackTrace();
            }
            orb.run();
        }
}
```

在初始化ORB之后，要获取一个POA的引用。ORB可以通过"RootPOA"来获取一个初始引用，此引用只是一个CORBA.Object，需要使用POAHelper来实例化为POA的引用。下一步是激活该对象，因为引用刚创建时是“保持”状态，在这种状态下，不能处理任何请求。通过调用POA的POAManager对象的activate()方法将POA激活。现在就可以通过POA将一个Java对象转化成一个CORBA对象。

为了使新创建的CORBA对象能被客户端访问，我们要提供该对象的引用。这一过程通过目录服务--命名服务器--来完成。命名服务 器的引用通过调用 orb.resolve_initial_references("NameService")，然后用 org.omg.CosNaming.NamingContextExtHelper 的narrow()方法实例化为正确的命名服务器对象。最后，调用命名服务器的bind()方法将CORBA对象引用进行发布。对象的名称作为bind()的参数传入，当然，不能只传递一个字符串，而是应传入代表名称的 CosNaming.NameComponents对象。在上例中，我们选用了一个扩展的命名服务，可以方便地将名称转化为名称对象。

3.6 编写客户端

最后，让我们来看一看客户端是如何调用服务程序的：

```
package demo.grid;
import org.omg.CosNaming.*;
public class Client
{
    public static void main(String args[])
    {
        try
        {
            MyServer grid;
           org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            if(args.length==1 )
            {
                // args[0] is an IOR-string
                grid =
MyServerHelper.narrow(orb.string_to_object(args[0]));
            }
            else
            {
                NamingContextExt nc =
                    NamingContextExtHelper.narrow(
                    orb.resolve_initial_references("NameService"));
                grid = MyServerHelper.narrow(
                    nc.resolve(nc.to_name("grid.example")));
            }
        short x = grid.height();
        System.out.println("Height = " + x);
        short y = grid.width();
        System.out.println("Width = " + y);
        x -= 1;
        y -= 1;
```

```
            System.out.println("Old value at (" + x + "," + y +"): " +
            grid.get( x,y));
            System.out.println("Setting (" + x + "," + y +") to 470.11");
            grid.set( x, y, new java.math.BigDecimal("470.11"));
            System.out.println("New value at (" + x + "," + y +"): " +
            grid.get( x,y));
            try
            {
                 grid.opWithException();
            }
            catch (jacorb.demo.grid.MyServerPackage.MyException ex)
            {
                 System.out.println("MyException, reason: " + ex.why);
            }
      }
      catch (Exception e)
      {
            e.printStackTrace();
      }
      }
}
```

在初始化ORB之后，客户端通过命名服务器获取一个"grid"服务的引用。如上所述，通过调用orb.resolve initial references("NameService")获取命名服务，使用resolve()方法在命名服务器上查找"grid"的引用。返回的结果是org.omg.CORBA.Object对象，需要实例化为MyServer。

在将所有的Java类都成功编译以后，现在可以在不同的Java虚拟机中启动服务器和客户端。当然，应该是命名服务最先启动，如果命名服务还没有启动，使用以下命令来启动：

```
$ns /home/me/public_html/NS_Ref
```

/home/me/public_html/NS_Ref是一个本地的可写文件，服务器和客户端都可以通过URL的方式访问。这样，可以不使用端口号，而服务器和客户端都可以获取命名服务的引用，从而实现互相访问。

启动服务器：

```
$jaco demo.grid.Server
```

接着，运行客户端：

```
$jaco demo.grid.Client
```

运行客户端产生以下输出：

```
Height = 31
Width = 14
Old value at (30,13): 0.21
Setting (30,13) to 470.11
New value at (30,13): 470.11
MyException, reason: This is only a test exception, no harm done :-)
done.
```

3.7 桥接方式

如果在实现接口已经继承了另外一个类，而不能继承MyServerPOA时，可以使用桥接方式。简言之，这种方式将继承改为指派，不是继承MyServerPOA，则是实现MyServerOperations接口。

```
package demo.grid;
import demo.grid.MyServerPackage.MyException;
public class gridOperationsImpl implements MyServerOperations
```

```
{
...
}
```

服务器采用以下方式实现：

```
package demo.grid;
import java.io.*;
import org.omg.CosNaming.*;
public class TieServer
{
	public static void main( String[] args )
	{
		org.omg.CORBA.ORB orb =
			org.omg.CORBA.ORB.init(args, null);
		try
		{
			org.omg.PortableServer.POA poa =
				org.omg.PortableServer.POAHelper.narrow(
				orb.resolve_initial_references("RootPOA"));
			// use the operations implementation and wrap it in
			// a tie object
			org.omg.CORBA.Object o =
				poa.servant_to_reference(
				new MyServerPOATie( new gridOperationsImpl()) );
			poa.the_POAManager().activate();
			if( args.length == 1 )
			{
				// write the object reference to args[0]
				PrintWriter ps = new PrintWriter(
					new FileOutputStream(new File( args[0] )));
				ps.println( orb.object_to_string( o ) );
				ps.close();
			}
			else
			{
				NamingContextExt nc =
					NamingContextExtHelper.narrow(
					orb.resolve_initial_references("NameService"));
				NameComponent [] name = new NameComponent[1];
				name[0] = new NameComponent("grid", "whatever");
				nc.bind( name, o );
			}
		}
		catch ( Exception e )
		{
			e.printStackTrace();
		}
		orb.run();
	}
}
```

4 JacORB 命名服务

命名服务使用友好的方式定位对象，使编程者不用去记忆机器和网络地址。如果可以用服务名去查找提供该服务的对象，客户端就可以不用关心提供服务的对象在哪。名称和服务的绑定的改变对客户端来说是透明的。
JacORB实现了OMG的协作命名服务(INS)，提供将名字绑定为对象引用，通过名字查找对象引用的功能。也提供功能让客户端可能很容易地在名字和字符串之间进行转换。JacORB命名服务由两部分组成：一个为服务程序，另一个为一组可以访问服务程序的类和接口。
在使用JDK1.2以上版本时，需要注意JDK有很多和命名服务相关的类都过时了，并且存在bug，这些将导致JacORB不能正常工作。为了不载入有问题的类，最好用`NamingContextExt`来代替`NamingContext`接口，否则，总会碰到空指针或其他异常。在JDK1.1 中没有这些问题。

4.1 运行命名服务

JacORB命名服务是一个程序，需要在访问前启动。使用如下命令启动：
```
$ns <ior filename> [<timeout>]
```
也可以通过java程序直接启动：
```
$jaco jacorb.naming.NameServer <filename> [<timeout>]
```
在上例中
```
$ns /home/me/public_html/NS_Ref
```
命名服务将位置信息和日志信息写入`/home/me/public_html/NS_Ref`文件中。客户端使用这个文件来定位命名服务。但是，在默认情况下，客户端并不是通过本地或共享文件的方式来读取该文件，而是使用URL通过http协议来访问。这也表明命名服务日志文件可能通过HTTP方式进行访问，如果知道位于域中哪一个Web服务器中。
这种实现方式的优点在于客户端不必知道命名服务的端口号，而是通过URL的方式来访问。如果想限制命名服务在域中的可见性，或者无法访问一个Web服务器，那么可以使用文件URL的方式来代替HTTP的URL方式，例如，可以使用如下方式
```
file:/home/brose/public_html/NS_Ref
```
来替代
```
http://www.inf.fu-berlin.de/~brose/NS_Ref
```
使用URL的方式也方便没有网络连接的机器，请注意HTTP协议并不总能被使用，当客户端已经定位命名服务以后，后续的请求将使用标准的CORBA方法调用，也就是使用IIOP协议(基于TCP/IP)。
命名服务将自身的内部状态，例如在命名空间中绑定的名称，存储在当前目录的文件中。目录也可以用参数`jacorb.naming.db_dir`来指定。当命名服务正常终止时，内部状态都将被保存，但使用Kill或Control-C时会造成数据丢失。如果状态文件存在且不为空，命名服务就能从该文件中恢复状态。
第二个参数是以毫秒计算的超时值。如果这个参数被设置，则命名服务在到运行指定时间后，将保存状态并正常关闭。这种方式在命名服务注册到实现仓库时很有用，这样，命名服务可以按需启动。

配置根命名空间

配置一个命名空间(例如命名服务)作为ORB根命名空间，只需要将启动文件指定到配置文件`.jacorbproperties`即可。也可以将启动文件设为配置项`ORBInitRef.NameService`的值。
在根命名空间配置好以后，对象NamingContextExt所有的调用都将访问`orb.resolve_initial_references("NameService")`，当然，要求命名服务已经启动或被实现仓库启动。

4.2 访问命名服务

在JacORB中通过标准的CORBA定义方式访问命名服务：

```
//获取命名服务的引用
ORB orb = ORB.init(args, null);
org.omg.CORBA.Object o = orb.resolve_initial_references("NameService")
NamingContextExt nc = NamingContextExtHelper.narrow( o );
//查找对象
server s =
serverHelper.narrow( nc.resolve(nc.to_name("server.service")) );
```

在查找对象之前，需要获取命名服务的引用。获取引用的标准方式是调用`orb.resolve_initial_references("NameService")`，使用这种方式调用，扩展的命名服务，对象名作为一个`NameComponents`数组进行传递，这样能传递结构名称。通过这种方式，可以构造这样一个数组，指定服务的名为"server"，类型为"service"，而不用指定一个命名空间。另外，也可以调用接口`NamingContextExt`的方法to_name()将字符串"server.service"转化结构名称。
然后，可以在名称命名空间中调用`resolve()`方法来查找对象，并传递一个结构名称作为参数。

4.3 构造层次命名空间

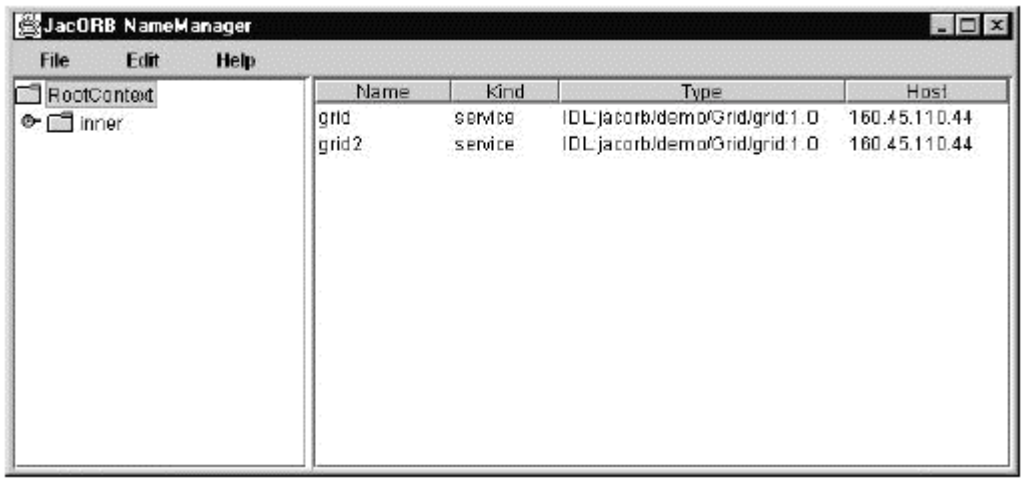
就象文件系统的目录结构一样，命名空间也可以包含其他的命名空间，从而构成层次结构，而不是平面结构。对象的结构名称的各部分就构成了名称的路径，在最内层的命名空间中绑定对象的名称。这可以使用命名管理器实现，或者通过编码实现。
在已有的命名空间中创建新的子命名空间可以通过`new_context()`或bind_new context()方法。下面的代码片段演示了如何创建一个子命名空间，并返回它的引用：

```
// get a reference to the naming service
ORB orb = ORB.init();
org.omg.CORBA.Object o =
orb.resolve_initial_references("NameService");
NamingContextExt rootContext =
NamingContextExtHelper.narrow( o );
// look up an object
NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("sub","context");
NamingContextExt subContext =
NamingContextExtHelper.narrow( rootContext.bind_new_context( name ));
```

请注意在JacORB中总是在内部使用`NamingContextExt`对象，即使代码中写的是`NamingContext`对象，原因在前面已经描述过。

4.4 命名管理器

如果使用JDK1.2 以上的Java虚拟机，或带有JFC相关类。图形界面的命名管理器就可以通过以下命令启动：
$nmg
命名管理器启动以后查找根命名空间并显示其内容，请看屏幕截图：



命名管理器有绑定名称，创建子命名空间的菜单。如图，右击RootContext，在弹出的菜单中选择new context，输入名称，可创建子命名空间。

5 服务端：POA 及线程

本章介绍JacORB提供控制服务启动和运行的应用及接口，包括激活服务、可移植对象适配器(POA)、线程。
本章只给出一个POA的简单介绍，并不涉及如何使用POA不同的设置和不同的策略，详细信息可以参考附录的参考书目[BVD01]，也可参考另一书目[HV99, Vin98]。如何在C++中处理可参考网站`http://www.cs.wustl.edu/~schmidt/report-doc.html`上的相关文章，最终的参考是CORBA规范。

5.1 POA

POA提供复杂的接口来管理对象引用及伺服程序。使用POA接口写的代码现在可以在兼容CORBA2.2或以上版本的各个ORB中移植。
POA定义了以下几个方面的标准接口：

- ➉映射对象引用到实现对象的伺服程序
- ➉允许透明地激活对象
- ➉将策略信息绑定到对象
- ➉可以在几个服务进程中持续化CORBA对象

在POA规范中，已经废弃了伪IDL的使用，而直接使用标准IDL，使用标准的语言映射定义来将IDL映射为编程语言，但这有本地限制。因为这意味着对象引用不能传递到服务器地址空间以外，POA接口本身也是有本地限制的。
POA是CORBA负责创建CORBA对象及对象引用的部分，并且，也有助于骨架(skeleton)将请求调度到真正的对象。为了和实现仓库一致，也可以激活对象，例如，启动提供CORBA对象程序的进程。

5.2 线程

JacORB提供服务器端线程模式。POA负责调度中央线程池从中获得一个线程来处理请求。线程池的大小由参数`jacorb.poa.thread_pool_max`和`jacorb.poa.thread_pool_min`决定。
当一个请求到达时，如果线程池中的所有线程都忙，并且线程总数小于`jacorb.poa.thread_pool_max`时，会启动一个新线程来处理请求。否则，请求将会阻塞直至有可用的线程。当线程处理完一个请求之后，必须决定是返回线程池还是被销毁。当线程池已经超过最小值时，处理中的线程不会返回线程池，从而保证线程池总在最大值和最小值之间。
将最小值设置为大于 1，意味着在线程池中总有一些线程处于激活状态等待处理请求。这对于请求以脉冲方式到达的情况最为有效。限制线程池的最大值可以防止服务器占用过多的资源。
请求处理线程通常以较高的线程优先级运行。线程优先级通过参数`jacorb.poa.thread_priority`来设置，参数值为Thread.MIN_PRIORITY、Thread.MAX_PRIORITY、Thread.NORMAL_PRIORITY之一，如果设置错误，JacORB将其值设为Thread.MAX_PRIORITY。

6 实现仓库

实现仓库(ImR)并不象其名所表述的，是一个保存实现的数据库。相反，它只保存如何将请求调度到真正的CORBA对象，如何实例化一个实现的信息。"实例化一个实现"是指启动一个包含所需CORBA对象的服务程序。在本章我们只是简单介绍如何使用实现仓库，详细信息请参考[HV99]。

6.1 概述

一般地，实现仓库中保存请求如何使用持久对象引用的信息。一个持久对象引用是POA通过持久生命策略创建的。这意味该对象的生命周期比创建它的POA还长久。在实现仓库中使用持久对象引用，实现仓库就可以不用关心当请求到来时，对象是否存在，如果不存在，就创建一个。
为了达到这个目的，实现仓库必须知道每个请求和持久对象的关系。这一点可以通过重写持久对象引用，使其包含实现仓库的地址信息，而不是包含服务程序地址信息来达到。因此，请求首先到达实现仓库，而不是实际的服务程序，因为服务程序可能在请求到达时并不存在。如果这样的请求到达实现仓库，实现仓库首先在内部表中查找服务的信息，判断这个服务对象存不存在，如果不存在，实现仓库有足够的信息来启动一个相关的服务对象，在服务对象起来以后，实现仓库发给客户端一个LOCATION_FORWARD的异常，在此异常中，包含新服务对象引用，这样，客户端能透明地使用新服务对象引用来处理请求。

6.2 使用 JacORB 的实现仓库

JacORB实现仓库由两部分组成：仓库进程，在域中只存在一个；服务启动进程，在每一台需要启动的服务的主机上都要有一个。注意，不需要一个处理具有易失生命周期特性对象的进程。
首先，必须先启动中央仓库进程(即实现仓库)：

```
$imr [-n] [-p <port>] [-i <ior file>][-f <file>][-b <file>] [-a]
```

实现仓库使用配置项`ORBInitRef.ImplementationRepository`来定位。这个属性文件必须设置到一个可用的WEB服务器上，并且实现仓库的IOR能被读取。下一步，启动进程在需要的主机上启动，使用如下命令：

```
$imr_ssd
```

当启动进程启动时，它会和实现仓库通讯。
使用以下命令将可以将启动的服务注册到实现仓库，在能访问实现仓库的任何主机都可以使用此命令。

```
$imr_mg add "AServerName" -c "jaco MyServer"
```

imr_mg命令是操作实现仓库的通用命令。在上述命令中，参数add指明将服务注册到实现仓库，紧接着的参数是实现名，如果服务所在主机不是本机，还需要用参数-h
hostname来指定目的主机。最后，实现仓库需要知道如何启动服务，由参数-c 来指定，jaco MyServer说明了如何启动本服务。-
c参数的格式和启动服务的命令一样，这个命令必须服务所在主机能执行。如在Windows下可以使用start jaco MyServer将服务启动到独立的控制台中，在unix下则可以使用`xterm -e jaco MyServer`来实现同一目的。
启动命令作为一个字符串将完整直接地传递到`Runtime.exec()`方法中，不会被解释或修改。因为`Runtime.exec()`是和平台相关的，所以启动参数也和平台相关。大多数unix系统中可以使用*或~来防止shell解释，windows系统不将字符串传递给命令行解释器，因此，即使在dos提示符下输入jaco MyServer能执行，也不能保证将此字符串作为imr_mg能成功。最好将启动命令封装成直接调用命令行解释器，在NT下，使用如下命令：`cmd /c "jaco MyServer"`。请记住，在使用imr_mg设置启动命令时，要将命令中的双引号进行转码。
如果不想让服务由实现仓库自动启动，可以设置配置项`jacorb.imr.allow_auto_register`或

使用实现仓库命令的-a参数。如果此配置项被设置，在POA激活时，实现仓库交自动为服务创建一个新配置，如果服务以前没有注册到实现仓库，那么，不能通过实现仓库管理器来注册该服务。
客户端需要一个服务对象引用来提交请求，直接现在，我们还没有提到如何将持久对象引用激活的问题。引用创建和通常一样，当一个引用被创建成可持久化，POA必须使用持久生命周期策略来创建它，如以下代码所示：

```
/* 初始化ORB和根POA */
orb = org.omg.CORBA.ORB.init(args, props);
org.omg.PortableServer.POA rootPOA
=org.omg.PortableServer.POAHelper.narrow(
orb.resolve_initial_references("RootPOA"));
/* 生成策略 */
org.omg.CORBA.Policy [] policies = new org.omg.CORBA.Policy[2];
policies[0] =
rootPOA.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID);
policies[1] =
rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
/* 创建POA */
POA myPOA = rootPOA.create_POA("XYZPOA",rootPOA.the_POAManager(),
policies);
/* 激活POA */
poa.the_POAManager().activate();
```

注意，使用持久对象引用的POA其ID应设置为USER_ID，因为这个值会在保存对象状态时保存到状态库中。如果POA使用持久生命周期策略创建，并且ORB的"use_imr"配置项已经设置，ORB会通知实现仓库，因此，实现仓库知道不用创建一个新的对象来处理请求。只需将ORB的配置项设置为：`jacorb.use imr=on`即可。ORB使用另一个配置项`jacorb.implname`作为通知的参数。如果服务已经注册到实现仓库，配置项的值应该和注册时使用的实现名匹配。
在程序中可以通过命令行参数`-Djacorb.implname=MyName`设置这个值，也可以使用以下代码来设置：

```
/* create and set properties */
java.util.Properties props = new java.util.Properties();
props.setProperty("jacorb.use_imr","on");
props.setProperty("jacorb.implname","MyName");
/* init ORB */
orb = org.omg.CORBA.ORB.init(args, props);
```

当在启动时恢复对象状态或在关闭时保存对象状态，有几件事情需要注意。在启动时，对象激活之后初始化才算完成，因为只有这时候对象才能接受请求。仓库在第一个具有持久生命周期的POA注册之后，才能知道服务，但此时还不能将服务引用传递给客户端，因为服务并不真正可用。可以解决此问题的一个不可靠的方法是增加配置项`jacorb.imr.object_activation_sleep`的值，这样，实现仓库需要更长的时间来完成启动，从而可以等待服务就绪。
在服务关闭时，也即时服务的状态被最后的POA保存时，因为实现仓库认为服务已经关闭，如果此时有新的请求到来，实现仓库就会启动一个新的服务进程来处理。此时，服务的实现者应该注意服务保存状态和读取状态之间的同步问题。一个解决方法是使用POA管理器将POA设置为在保存信息或不活动时，处于保持状态，这样请求会被阻塞至POA激活。
请记住即使在服务关闭时你不用去保存服务的状态，在退出时也一定要将POA失活，可以通过调用orb.shutdown(...)，否则实现仓库认为服务依然活跃，从而给客户端返回错误的IOR。在服务崩溃时可以使用命令imr_mg setdown AserverName来通知服务仓库该服务已经中止。

6.3 服务迁移

实现仓库提供了另一项有用的功能：服务迁移。设想以下场景：当使用持久POA来实现服务时，在一段时间以后，主机变得很慢，以至于很难处理所有的请求。这时候，将服务迁移到一台更强大的

主机上是理所当然的事。使用实现仓库，客户引用不包含服务的地址信息，因此，可以透明地完成服务迁移。

例如，你在一台慢的主机上启动服务：

```
$imr_mg add AServerName -h a_slow_machine -c "jaco MyServer"
```

第一步是保持服务，这意味实现仓库将阻塞所有的请求直至服务再次发布。

```
$imr_mg hold AServerName
```

现在，服务不能接受已注册POA，如果不能在服务停止时使用POA失活，你可以使用以下命令：

```
$imr_mg setdown AServerName
```

否则，POA将不能再次被激活，因为实现仓库认为POA一直处于活动状态。

如果希望服务被自动启动，你需要通知实现仓库新的主机地址，也可能还有新的启动命令：

```
$imr_mg edit AServerName -h the_fastest_available_machine -c "jaco
MyServer"
```

如果服务能自动启动，你就不必手工启动，它将会在第一请求到达时被实现仓库启动。当然，你也可以在新主机上手工启动。

最后一步是发布服务，让所有阻塞的请求可以被处理：

```
$imr_mg release AServerName
```

现在，服务就运行在新的主机上，而不用通知客户端。

6.4 安全考虑

采用实现仓库可能会对系统构成安全威胁。考虑以下情况：实现仓库运行在一台主机上，它的IOR放置在一个可以被WEB服务器访问的目录下，在另外的几台主机上运行着几个服务启动进程。一个攻击者可以在一台运行服务启动进程的主机上执行以下操作：

将配置项`ORBInitRef.ImplementationRepository`设为实现仓库所在主要的IOR文件。

1. 创建一个入侵服务，在想入侵的主机(已有ssd在运行)执行一个非法的启动命令。这是最关键的一点，因为服务启动进程只是将启动命令作为参数传递给Runtime.exec()，而没有方法去检查命令是否合法，例如是否是启动一个服务。
2. 使用imr_mg启动该入侵服务，指定的命令将在被入侵的主机上执行。

虽然有风险，但并不是不鼓励使用实现仓库，这个风险可以使用以下方法显著降低：

1. 控制IOR文件的分发。但不要使用隐藏文件的方式，这不是一个好方法。最好使用文件系统的组策略或访问控制策略(ACL)。
2. 使用防火墙来过虑请求。但要注意，当攻击来自内部时，防火墙很难防范。也要注意，写一个穿过防火墙的木马程序并不是很困难的事。
3. 使用SSL连接来访问实现仓库。这可以阻止没有合法客户端证书的访问，详见第 9 章。

7 Any Value 的动态管理

本章的目的是描述DynAny规范，这个规范是关于Any Values的动态管理。本章仅仅描述DynAny规范的主要特性。
如果想得到本章内容全部相关的参考，请取得OMG的关于CORBA规范，自行参考。

7.1 概述

DynAny 对象被用来动态构造和传输访问任意类型的值。一个 DynAny 可以描述一个基本类型，例如 boolean 或者 long,或者一个构造类型，例如 enum 或 struct。

7.2 接口

下面的UML框图说明了org.omg包中的接口之间的关系。

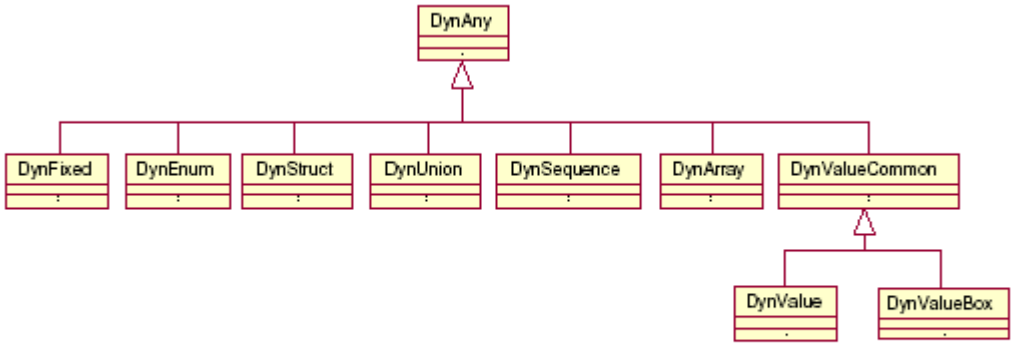


Figure 7.1: DynAny Relationships

DynAny 接口是描述基本类型的基类接口。针对每一种结构类型，有一个对应的从 DynAny 接口继承的接口和定义的一些专用于这些结构类型的操作。下面的表格列出了动态任意（DynamicAny）模型和他们所描述的值。

Interface	Type
DynAny	basic types (boolean, long, etc.)
DynFixed*	fixed
DynEnum	enum
DynStruct	struct
DynUnion	union
DynSequence	sequence
DynArray	array
DynValue*	non-boxed valuetype
DynValueBox*	boxed valuetype

* Not currently implemented in JacORB.

7.3 用法限制

从 DynamicAny 模块中的接口实现的对象，被有意的本地化来处理结构类型。结果是，这些对象的引用不能导出到别的程序中或使用 ORB::object 到 string 的下塑造型；一个试图这样做的操作将抛出 MARSHAL 系统异常。

7.4 创建一个 DynAny 对象

DynAnyFactory 接口被用来创建 DynAny 对象。有两个操作用来创建 DynAny 对象；在下表列出。

Operation	Description
create_dyn_any	Constructs a DynAny object from an Any value
create_dyn_any_from_type_code	Constructs a DynAny object from a TypeCode

下面的例子演示了怎样获得一个DynAnyFacory对象的引用，然后使用他的每个构建操作来构建一个DnyAny对象。

下面的代码导入DynamicAny包中的类。

```
import org.omg.DynamicAny.*;
```

下面的代码片断获得一个 DynAnyFacory 对象的应用。

```
DynAnyFactory factory = null;
DynAny DynAny = null;
DynAny DynAny2 = null;
org.omg.CORBA.Any any = null;
org.omg.CORBA.TypeCode tc = null;
org.omg.CORBA.Object obj = null;
//获得DynAnyFactory的对象引用
obj = orb.resolve_initial_references ("DynAnyFactory");
//转换应用到正确类型
factory = DynAnyFactoryHelper.narrow (obj);
```

下面的代码片断使用每种创建方法创建DynAny对象。

```
//从Any创建一个DnyAny对象
any = orb.create_any ();
any.insert_long (1);
DynAny = factory.create_dyn_any (any);
//从TypeCode创建一个DynAny对象
tc = orb.get_primitive_tc (org.omg.CORBA.TCKind.tk_long);
DynAny2 = factory.create_dyn_any_from_type_code (tc);
```

如果Any Value或者TypeCode描述一个结构类型，那么DynAny可以转换到适当的类型，例示如下。

下面的IDL定义一个结构类型

```
// example struct type
struct StructType
{
long field1;
string field2;
};
```

下面的代码例示了一个代表StructType类型值的DynStruct对象的创建

```
StructType type = null;
DynStruct dynStruct = null;
//创建一个包含StructType对象类型的Any
type = new StructType (999, "Hello");
any = orb.create_any ();
StructTypeHelper.insert (any, type);
```

```
//从一个Any和将他造型到一个DynStruct构建一个DynAny
dynStruct = (DynStruct) factory.create_dyn_any (any);
```

7.5 访问 DynAny 对象的值

DynAny接口定义了一套用于访问代表基本类型<type>的DynAny对象的操作。这个用来从DynAny对象获得基本类型的值的操作用这样的方式get_<type>。用来插入基本类型<type>值的到DynAny对象使用这样的形式insert_<type>。如果类型操作使用得get/insert一个值到一个DynAny对象类型匹配错误，将抛出TypeMismatch异常。`

用来访问代表构造类型的的 DynAny 的操作是专门针对构造类型设计的。例如，DynStruct 接口定义一个访问成员的操作，他返回一个代表这个 DynStruct 对象的结构或者异常的成员的名称/值对序列。

7.6 传送 DynAny 对象的值

DynAny对象可以被看作是一个有序的DynAny的集合。例如，在一个DynStruct对象中，他代表的是许多结构和异常的有序集合。因为DynAny对象表示的是基本类型和没有其他成分的构造类型，DynAny的成分集合是空的。

所有的DynAny对象拥有当前位置。例如在一个代表构造类型的DynAny中，当前的位置就是这个DynAny中，这个将要被调用的成分的位置（在下表说明）。DynAny对象中的DynAny成分的索引从0到n-1；n是成分的数量。例如DynAny表示基本类型或没有成分的构造类型，当前的位置缺省值就是-1。

用来访问DynAny对象的DynAny成分的操作对所有的DynAyn的子类型通用，所以这些操作定义在基类接口中。下表列出了用来访问DynAny对象的可用操作。

Operation	Description
seek	Sets the current position to the specified index

Operation	Description
rewind	Sets the current position to the first component (index 0)
next	Advances the current position to the next component
component_count	Returns the number of components
current_component	Returns the component at the current position

下面的代码例示了一种访问一个DynStruct对象的DynAny成分的方法。当DynStruct被访问，每个成分的值被取到和打印。这里忽略了异常。

```
DynAny curComp = null;
//打印第一个成分的值
curComp = dynStruct.current_component ();
System.out.println ("field1 = " + curComp.get_long ());
//前进到下一个
dynStruct.next ();
//打印第二个成分的值
curComp = dynStruct.current_component ();
System.out.println ("field2 = " + curComp.get_string ());
```

下面的代码演示了完成同样任务的另一种方法。

```
// go back to the first component
dynStruct.rewind (); // same as calling seek (0)
```

```
// print the value of the first component
//打印第一个成分的值
System.out.println ("field1 = " + dynStruct.get_long ());
//前进到下一个
dynStruct.seek (1);
//打印第二个成分的值
System.out.println ("field2 = " + dynStruct.get_string ());
```

在第二段演示代码中，如果成分 DynAny 代表一个基本类型，可用通过调用他的父类的 DynAny 的访问操作直接访问，比调用当前的成分的访问操作方式要好。

7.7 构造类型

这部分描述JacORB中支持的DynamicAny模型中的接口。这些接口都是从DynAny接口继承而来。

7.7.1 DynEnum

一个DynEnum对象代表一个枚举值。枚举的整型值能够通过作为ulong类型的get和set操作被访问到。枚举的字符串值能够使用字符串的get和set操作访问。
一个DynEnum对象没有其他的成分。

7.7.2 DynStruct

一个DynStruct对象代表一个结构值或者一个异常值。当前的成员名称和当前的成员类操作返回名称和DynStruct的当前位置的成员的TypeCode的TCKind值。DynStruct的成员可以通过get和set方法访问。
一个DynStruct对象的成分DynAny是结构或异常的成员。一个DynStruct代表一个没有成分的空异常。

7.7.3 DynUnion

一个DynUnion对象代表一个联合体(union)值。辨别器（discriminator）的值可以通过使用get和set方法访问。
如果辨别器被设置为一个联合体（union）的成员名称，那么这个成员就变成活动的。否则，就不是活动成员。
如果存在活动成员，成员操作返回的值作为一个 DynAny 对象，并且这个成员和成员性质的操作返回他的名字和它的 TypeCode 的 TCKind 值。如果这个联合体没有活动成员，这些操作将抛出 InvalidValue 异常。一个 DynUnion 对象可以拥有一个或两个成分。第一成分总是辨别器的值。如果活动成员存在，第二个成分就是活动成员的值。

7.7.4 DynSequence

一个DynSequence对象代表一个序列。序列的长度可以通过使用长度的get和set操作得到。序列的元素可以通过元素set和get操作访问。
一个DynSequence对象的DynAny成分是序列的元素。

7.7.5 DynArray

一个DynArray对象代表一个数组。数组元素可以通过使用get和set操作访问。
一个DynArray对象的DynAny成分是数组的元素。

7.8 Any 和 DnyAny 对象之间的转换

DynAny接口定义了一些用来在Any对象和DynAny对象之间的转换方法。任何操作都要初始化拥有指定的Any值的DynAny。如果Any的类型和DynAny的不匹配，将抛出TypeMismatch异常。通过任何操作从一个DnyAny创建一个Any。

作为一个演示怎样使用这些操作的例子，可能有人想要动态改变一个构造类型的内容，例如struct,他用一个Any来代表。

通过下面的步骤完成这个任务：

1. 使用DynAnyFactory::create dyn any的类型操作从结构的TypeCode构造一个DynStruct对象。
2. DynAnyu::from 任意操作被用来初始化带有Any值的DynStruct。
3. 现在，DynStruct的内容可以被访问和修改了。
4. 使用DynAny::to 任意操作可以创建一个代表修改过的结构的新Any。

7.9 更多例子

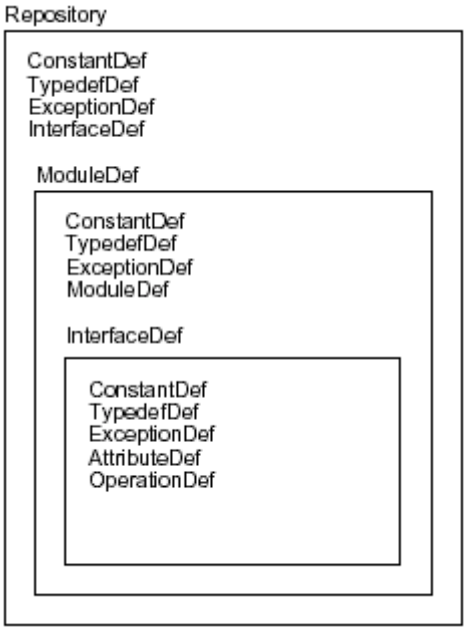
在JacORB的demo/dynany的目录中包含一些使用DynAny对象的例子代码。在JacORB-Test 的org.jacorb.test.orb.dynany包中还有更多的例子代码。

8 接口仓库

在CORBA中使用ORB的接口仓库(IR)组件来管理运行时的类型信息。它允许动态的调用、查看、修改IDL中的类型信息，例如，查找一个对象支持何种操作。有些ORB也需要接口仓库来确认一个对象是否是另一个对象的子类，但大多数的ORB都能不通过接口仓库，而使用IDL编译时产生的伺服程序来获取这种类型信息。
本质上，接口仓库是一个提供获取类型信息的可以远程访问的CORBA对象。注意JacORB的接口仓库只支持JDK1.2 或以上版本。

8.1 接口仓库中的类型信息

接口仓库使用与IDL接口文件相对应的层次包含结构来管理类型信息：模块包含接口、结构、常数等，接口包含异常、操作、常数等。下图显示了这种层次关系：



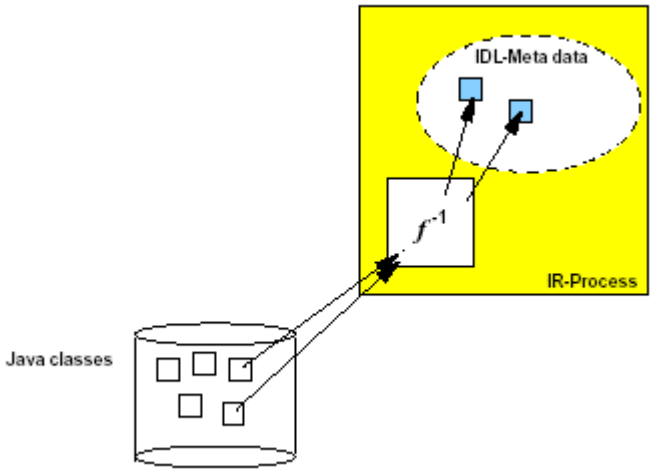
接口仓库的内部描述通过几种方式来标识。接口仓库中的每一个元素都有一个唯一的全名，这和IDL接口文件中的名空间相对应。在模块M1 的子模块M2 中定义的接口I1 有一个命名为M1::M2::I1。接口仓库也提供另一种更灵活的方式--接口仓库标识(RID)--来命名IDL接口文件的结构。有很多种不同的RID格式，但每个接口仓库必须能够处理下面的这种格式，以"IDL:"为前缀，以一个版本号为后后缀，例如："`IDL:jacorb/demo/grid:1.0`"，在分号之间的名称可以使用IDL编译器预处理指令`#pragma prefix`和`#pragma ID`来设置，如果没有设置，则为上述的全名。

8.2 接口仓库设计

在设计接口仓库时，我们的目标是充分利用Java反射API的功能，从而避免再自行定义一套IDL类型描述的基础数据。另一种设计是将接口仓库作为IDL编译器的后台，但是我们不想引入这样的依赖性，而想让接口仓库成为轻量级的仓库服务器。前一种设计是可以实现的，因为Java和CORBA对象的类似性，从而可以在运行时取得IDL类型信息。由此看来，我们可以不依赖编译时的IDL接口文件

来获取类型信息。另外，这种实现最大的好处是避免了数据的冗余以及IDL接口文件与Java实现时导致的数据不一致，因为Java类已经生成。

因此，接口仓库必须载入Java类，使用反射进行分析，生成相应的IDL类型信息。最后，接口仓库实



现了Java语言到IDL语言的反向对应，如下图所示，其中f^{-1}表示了语言的反向对应。

8.3 使用接口仓库

ORB要能访问接口仓库，接口仓库服务进程必须已经启动。启动接口仓库命令如下：

```
$ir /home/brose/classes /home/brose/public_html/IR Ref
```

第一个参数是包含class文件和包的目录，接口仓库从这里载入这些类，并认为是IDL编译器产生的Java类，如果上一步成功，则生成相关IDL的类型信息。第二个参数指向IOR文件。

要查询接口仓库的内容，可以使用图形化的接口仓库浏览器或查询命令。首先，让我们查询一个指定的ID，JacORB提供命令qir(query IR)来完成此功能：

```
$qir IDL:raccoon/test/cyberchair/Paper:1.0
```

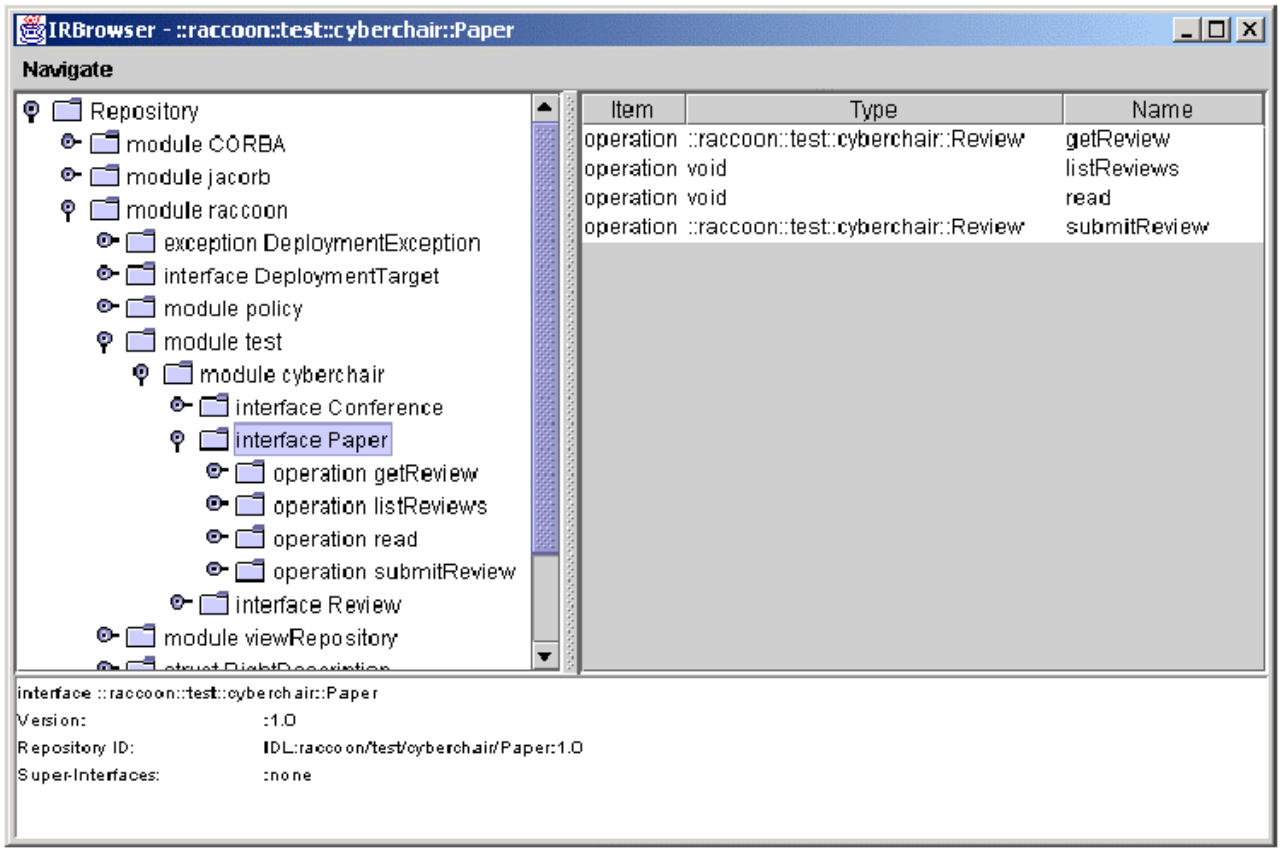
接口仓库返回InterfaceDef对象，qir分析此对象并输出如下结果：

```
interface Paper
{
      void read(out string arg_0);
      raccoon::test::cyberchair::Review getReview(in long arg_0);
      raccoon::test::cyberchair::Review submitReview(
            in string arg_0, in long a rg_1);
      void listReviews(out string arg_0);
};
```

启动接口仓库浏览器，使用以下命令：

```
$irbrowser
```

下图为屏幕截图：



IDL编译器使用OMG标准的IDL/JAVA语言映射来生成Java类，但这些Java类并没有足够的信息来重建原始的IDL接口描述。例如，无法判断一个接口是否是只读的，或者一个参数是传入类型还传入/传出类型的，而且，IDL的模块在Java中并没有明确的表述，因此要判断Java中一个目录是不是模块并不很容易。由于以下原因，JacORB的IDL编译器使用-ir选项产生另外一些Java类来保存这些信息。

```
$idl -ir myIdlFile.idl
```

编译器产生的其他类包括：

- ➊_Xmodule.java对应IDL定义的模块X
- ➊YIRHelper.java对应IDL定义的接口Y

如果接口仓库服务进程没有找到这些额外产生的类，它不会产生任何描述。注意，IDL编译器不会更改标准定义生成的Java文件，它只是生成一些额外的类。

在这些额外的类中，_Xmodule.java只包含在接口文件中定义的模块，而不会包含使命-d选项生成的Java包，因此，这些目录中的类也不会被接口仓库服务分析。

当客户端对象调用get_interface()操作时，ORB查询接口仓库，返回一个描述接口的InterfaceDef元对象，使用此对象可以查询更多的信息，如接口中的操作定义和属性定义等。

接口仓库也可以象其他的CORBA对象一样调用，使用lookup()或lookup_name()操作，单个容器的内容也可以被显示出来。

接口仓库元对象提供更多的描述操作。对于一个InterfaceDef对象，可以查看它包含的其他元对象(如OperationDef对象)。也可以直接获取结构描述对象InterfaceDescription或FullInterfaceDescription，因为结构通过值进行传递，因此，FullInterfaceDescription提供了所有包含的信息，而不用再通过远程调用来搜索结构。

9 JacORB 的小程序代理

从JacORB1.0beta13版本开始，JacORB包含一个IIOP的代理，称之为小程序代理(Appligator)。使用小程序代理，你可以在JacORB上运行Java小程序(Applets)。通常，Java应用程序可以和Internet上的任何一个主机进行连接，而小程序则只能和所在主机(从该主机下载的小程序)进行连接。如果没有使用代理，则小程序只能连接所在主机的CORBA服务器。使用JacORB的小程序代理，小程序的访问将不再受到限制。小程序代理运行在小程序所在主机，透明的处理小程序的所有请求。

9.1 使用小程序代理

因为JacORB小程序代理的透明性，在写小程序时你可以象写应用程序一样。你要做的唯一的一件事是使用如下方法初始化ORB：`jacorb.orb.ORB(java.applet.Applet, java.util.Properties)`。
JacORB的应用程序读取本地的配置文件(如`jacorb.properties`)来获取命名服务的URL及其他一些重要参数，而小程序则没有本地文件可言，但是有一个远程的：和小程序放在同一目录的配置文件，注意文件名必须是`jacorb.properties`。
和命名服务一样，小程序代理将自己的IOR写入到文件，小程序获取小程序代理IOR的文件位置，可以通过配置项
jacorb.ProxyServerURL来设置，或者使用<applet>中的参数`JacorbProxyServerURL`来设置。如果该参数未设置，或设置错误，JacORB将在小程序指定的basecode目录和WEB的根目录中寻找名为"proxy.ior"的IOR文件。
应该保证配置项`jacorb.NameServerURL`指向小程序所在主机，否则，在小程序使用命名服务时会产生一个安全异常。
使用如下命令启动小程序代理：
```
$appligator <port> <filename>
```
第一个参数为端口号，第二个指定小程序代理写入IOR的文件。这个文件必须和上述配置项
jacorb.ProxyServerURL 的值一致。

9.1.1 使用步骤

- ➊使用 `jacorb.orb.init(applet,properties)`初始化ORB，其中applet为小程序自身，properties可以为空
- ➊将`jacorb.properties`文件放置在小程序所在目录
- ➊使用配置项jacorb.ProxyServerURL或小程序参数指定小程序代理的IOR文件
- ➊确保命名服务的IOR文件可以被小程序访问(在同一主机上)
- ➊在同一主机上启动小程序代理`appligator<filename>`，其中文件为小程序代理的IOR文件

9.1.2 程序属性

如上所述，小程序有几种方式获取属性。最重要的属性是小程序代理的IOR文件URL，没有这个属性，小程序将无法工作。如果使用命名服务，命名服务的IOR文件URL也必须设置。
属性可以通过以下三种方式设置：

⑩在ORB.init()时传递属性
⑩在配置文件jacorb.properties中定义
⑩小程序代理的IOR文件URL也可以在HTML文件中设置小程序的参数

9.1.3 小程序代理和 Netscpe/IE，AppletViewer

Netscape自身对CORBA(但好久没有更新)支持，在使用JacORB时应该删除Netscape的CORBA类，即位于目录<NS_ROOT>/java/classes的iiop10.jar(在删除前最好保留备份)。仅仅将这个文件的后缀改变是不够的，因为Netscape将此目录下的所有文件载入。然后将jacorb.jar文件放入此目录中。
如果Netscape载入了错误的类或抛出了安全异常(这可以通过Netscape的Java终端看到)，就需要检查类路径，查找老的jar文件，从类路径中删除所有JacORB和VisiBroker的类文件。我们在Netscape4.72和JDK1.3的插件上成功运行了JacORB的小程序。
微软的IE比Netscape更严格：即使是下载的类也不允许监听一个端口。我们强烈建议你使用JDK1.3的IE插件。为了欺骗IE来使用JacORB，需要将JacORB的类放在$WINNT\Java\TrustLib目录下。你可以将jacorb.jar拷贝到此目录然后解包，或者只将jacorb, org及HTTPClient目录拷入此目录。
小程序代理在AppletViewer上能正常工作。只需要将JacORB的CORBA类代替Sun的CORBA类。通常使用以下命令来显示小程序：

```
$appletviewer http://www.example.com/CORBA/dii_example.html
```

在JacORB的bin目录中有一个名为jacapplet的脚本程序，已经设置好调用appletviewer的参数(需要修改其中JDK的参数)。
如果在其他的浏览器上使用小程序代理，或者知道更好的方式去使用小程序代理，请告诉我们。

9.1.4 示例

在目录jacorb/demo/applet下有几个小程序的例子，它们都基于应用的例子，都附带有调用小程序的html文件。要运行这些例子，先启动命名服务，再启动小程序代理，启动应用示例的服务，然后就可以使用appletviewer或Netscape/IE来访问小程序了。
确保jacorb.properties和jacorb.jar在正确的目录下。

9.2 通过防火墙使用 JacORB

防火墙通常做两件事：按端口过滤信息和按协议过滤信息。JacORB附带两个实用程序来解决这些两个问题。JacORB小程序代理可以处理端口限制，而HTTP隧道将GIOP包封装成HTTP包来通过防火墙。

9.2.1 小程序代理

小程序代理解决了小程序的沙箱限制。未签名的小程序只能连接到所在主机，这使它们在大多数的分布式CORBA应用中无用武之地。小程序代理是GIOP代理，它保证小程序能连到任何一个CORBA服务器上，因为，所有到CORBA服务的请求都经过代理，而所有返回的结果也经过代理，故而CORBA服务对小程序而言是透明的。
小程序代理不仅可用于小程序，还可以作为GIOP的代理，小程序代理本身是一个CORBA对象，并在指定的端口号在启动，所有访问小程序代理的请求将被发送到该端口，你可以配置在防火墙后面的CORBA对象都通过小程序代理来处理请求。一个好的实现是区分通过防火墙的请求和在防火墙之内的请求，这会在小程序代理的下一版本实现。
为了使端口过滤防火墙能处理CORBA和GIOP包，必须让系统管理员在防火墙上保留一个可用的端口，使用此端口启动小程序代理。

现在所有在防火墙后面的服务器都可以通过小程序代理来联系。如果CORBA客户端要访问位于Internet上的CORBA服务器，那么可以通过小程序代理。注意，Netscape不支持从Internet上回调客户。
为了使CORBA应用程序及服务能识别小程序代理，需要在配置文件中增加一个配置项。小程序会自动地使用小程序代理，如果想让应用程序也使用小程序代理，将配置项jacorb.use_appligator_for_applications设为
on，如果希望关掉小程序代理，则将上述配置项设为off，对小程序将jacorb.use_appligator_for_applets设为off。
最后，应该指明小程序代理的位置，这和指定命名服务的方式一样：当小程序代理启动时，IOR写入命令行参数指定的文件，这个文件必须可以被想使用小程序代理的客户端访问，可以使用共享文件的方式或使用WEB服务器，小程序代理的IOR文件URL必须和配置项jacorb.ProxyServerURL一致。

9.2.2 HTTP 隧道

如果防火墙使用协议过滤策略并且没有配置成允许GIOP包通过，可以通过HTTP隧道将GIOP包封装成HTTP包从而通过防火墙。JacORB内置HTTP隧道支持，生成HTTP1.1格式的数据包，在进行远程调用时，防火墙看到的是标准的HTTP请求(从CORBA客户端到CORBA服务器)和HTTP回复(从CORBA服务器到CORBA客户端)。
JacORB的HTTP隧道不和任何OMG标准兼容，因此只能用于JacORB的客户端和服务器。每一个JacORB服务器都不需特殊配置而能正确处理传入的HTTP通信，配置在客户端完成。可以依据IP地址来配置HTTP隧道，如果指定某地址需要HTTP隧道，则所有请求该主机的通信都将通过HTTP隧道完成。多个IP地址用逗号分隔，例如，设置配置项jacorb.use_httptunneling_for=192.168.0.1,192.168.0.2
，则所有到这两个主机的通信都使用HTTP隧道。

9.2.3 小程序代理和 HTTP 隧道

因为小程序代理也是一个CORBA服务，所以它也支持HTTP隧道。不需要额外的配置小程序代理就可以使用HTTP隧道来通过防火墙。

9.2.4 总结

- ➉当防火墙只允许部分端口通过时，可以使用小程序代理作为GIOP代理
- ➉让系统管理员配置一个防火墙允许通过的端口，如 7777，然后在该端口上启动小程序代理 $appligator 7777 Appligator_Ref
- ➉所有的CORBA对象都需要使用小程序代理来被防火墙外的客户访问或去访问防火墙外的其他服务。在这些应用的配置文件中加入配置项 jacorb.use_appligator_for_applications=on。
- ➉在客户端使用配置项jacorb.ProxyServerURL来设置小程序代理的IOR文件URL。
- ➉使用配置项jacorb.use_httptunneling_for来设置需要HTTP隧道访问主机。

10 基于 SSL 的 IIOP

不需要修改任何代码，就可以使用SSL来认证客户端并保护客户端和服务器之间的通信。当然在连接层使用SSL代替了TCP后，建立连接会慢一些。
在使用SSL之前，需要重新编译JacORB，使它支持加密。也需要配置一个证书库来保存加密时用到的信息，并配置SSL要用到的其他一些属性，这些都会在本章讲到。

10.1 重编译 JacORB 安全库

在标准版本中，JacORB没有激活安全库的支持。因此，你需要将使用的安全库加入类路径中，然后重新编译JacORB。如果安全库未找到，则编译以后的JacORB依然不激活安全库的支持。
在编译之前，以下条件必须满足：

- JDK1.2或以上版本
- 当使用IAIKs库时：
  - IAIK-JCE2.591或以上版本，参考站点`http://jcewww.iaik.tu-graz.ac.at/`
  - IsaSiLK 3.0或以上版本
- 当使用Sun的库时：
  - JDK1.4或JSSE1.02，可以从JDC中下载(对于JSSE1.02，参考目录`src/org/jacorb/security/ssl/sun_jsse`的文件`README.jsse_1_0_2`来编译
  - 对于证书钥管理，需要其他的包，如OpenSSL等，但不影响JacORB的运行

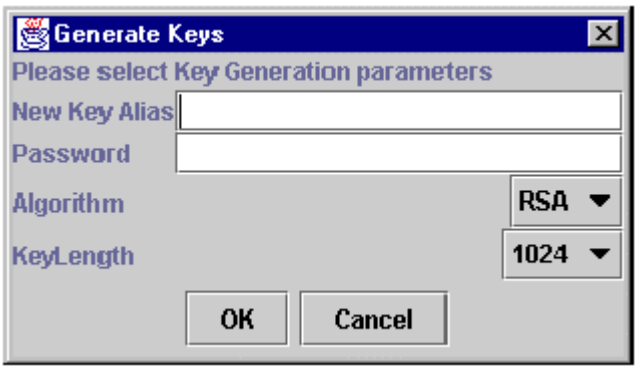
安装所需的包并仔细阅读相关文档，然后在JacORB的安装目录中执行ant命令来重新编译。

10.2 配置 IAIK

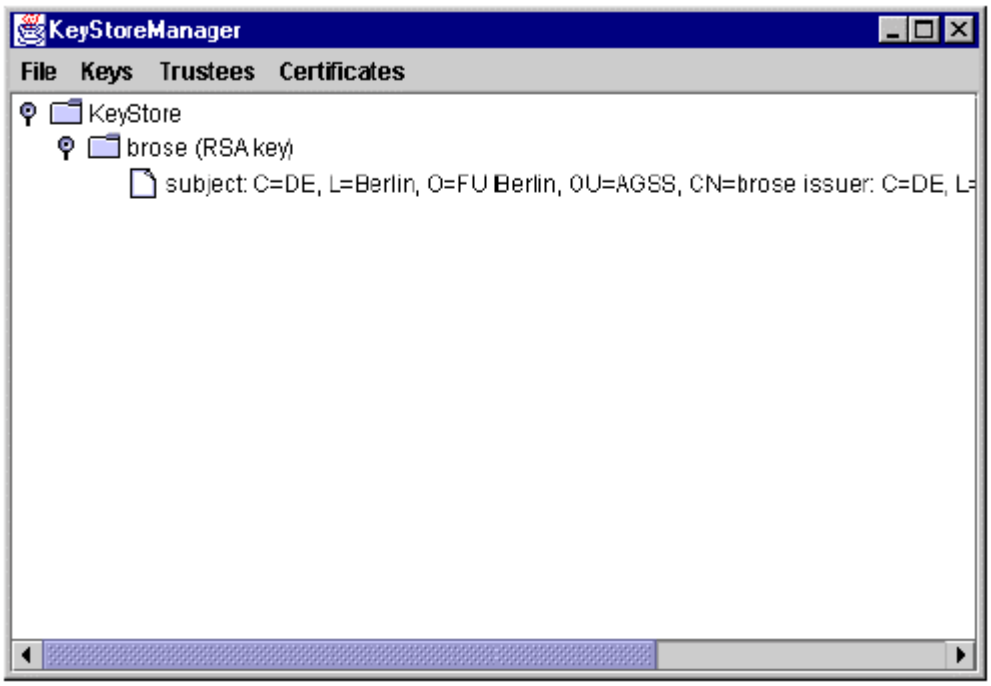
本章说明IAIK的配置

10.2.1 设置 IAIK 证书库

SSL依赖于X.509格式的公钥证书。这个证书在建立连接时使用，并用来生成会话时使用的密钥。
这一节描述如何创建和保存这些证书。
Java2提供访问称为证书库的持久数据结构的接口。一个证书库就是保存公钥公钥证书及相关私钥的文件，也包含能够验证公钥的其他证书。所有的加密数据使用密码保护，通过别名进行访问。
JacORB提供一个图形化的工具来创建和操作证书库，称之为证书管理器。使用它可以产生公钥对、对公钥进行签名、导入/导出证书、定义信任的认证中心。要启动证书管理器，只需输入ks命令即可，然后可以使用图形化的方式选择或打开已有的证书库，或创建一个新的证书库。
启动证书管理器，创建一个新的证书库，并创建相关的公钥对及证书。从文件菜单中选择新建即可新建一个证书库，从钥匙菜单中选择新建，创建一个新的公钥对，这时，要求提供一个别名，并需要输入一个密码，对加密算法和公钥长度可以使用缺省值。如下图所示：

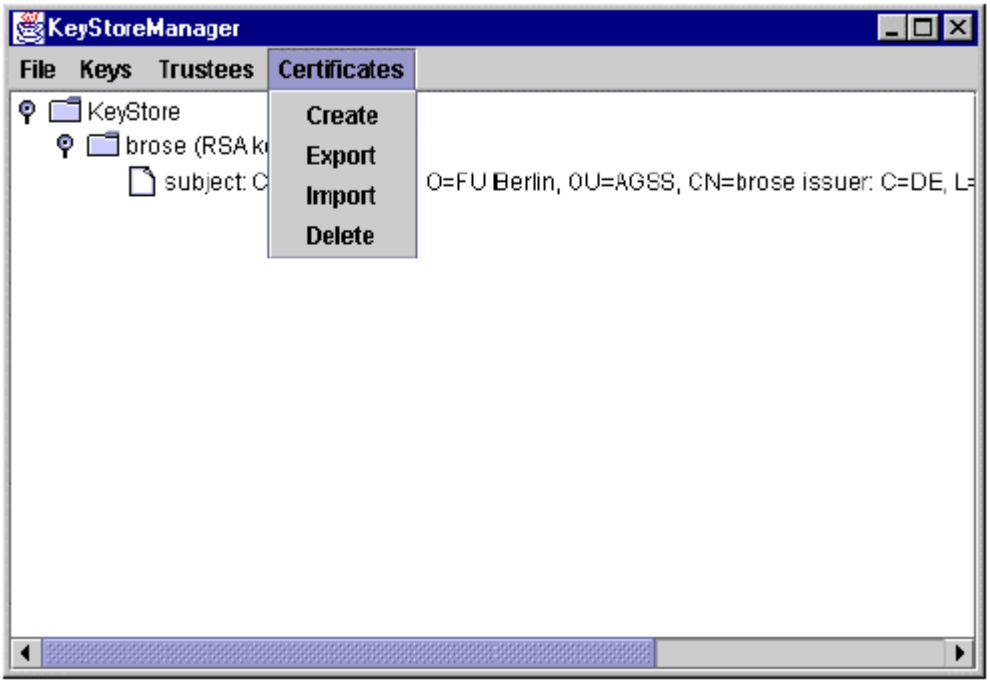


现在就有了一个可以用来认证的公钥证书，标识证书的别名也已保存在证书中。因为任何人都能生成这样的一个公钥证书，因此，收到证书的人要求此证书有自己信任的认证中心的数字签名。要签名此证书，认证中心提供证书主题的标识声明。这个签名被信任，通常认证中心会审查需要签名的



公钥证书，甚至只签名自己发放的公钥证书。

为了使用方便，你可以自己充当认证中心，使用证书管理器导入证书，签名然后导出。原始的证书库能重新导入这个经过某认证中心签名的证书。证书库有使用一个标准的链格式来存储公钥证书，链的根证书就是在生成证书库时产生的证书。它自动地被私钥签名。链上的第二个证书就是被认证中心签名的证书。链上的最后一个证书必须是认证中心的公钥证书，并被认证中心自己签名。



可以选择一个证书的别名，然后从钥匙菜单选择验证来验证此钥匙链的有效性。必须保证钥匙链的格式正确，并且已经通过信任者菜单中加入认证中心为信任者，否则验证会失败。只有验证通过，该公钥证书才被用于建立SSL连接。更多关于证书库的文档可以参看Sun工具keytool的相关文档。如果你关心“真正”的安全性，在整个系统中使用一个管理认证中心是必要的。

10.2.2 一步一步生成证书

为了创建一个简单的证书体系，应按照如下步骤进行(括号中是管理器的菜单命令)：
1.创建一个用户的证书库(File|New)，生成认证中心和用户的公钥对(Keys|New)
2.打开该证书库(File|Open)，选择并导出自签名用户证书(Certificates|Export)
3.打开认证中心的证书库，并将用户证书加入到信任者(Trustees|Add)
4.选定信任的用户证书，并生成一个签名的公钥证书(Certificates|Create)。保留角色名为空，并输入私钥密码，点击OK生成证书。
5.导出认证中心自签名的证书。从认证中心的证书库中删除信任的证书(Trustees|Delete)
6.再次用户的证书库，导入认证中心已经签名的用户证书(Certificate|Import)，并导入认证中心的公钥证书。
7.将认证中心的公钥证书设为信任者。这只是为了验证链用，发布证书库并不需要。注意验证失败会导致一个签名异常。

10.3 配置SSL 的属性

当ORB被应用初始化时，它从配置文件或命名行读入配置项。为了JacORB支持SSL，应设置以下配置项：`jacorb.security.support_ssl=on`。这样，在启动时会载入SSL的相关类，通过不同的配置项来配置SSL的其他信息。
正如在前面章节讲到的，加密数据(公钥对和证书)保存在证书库中，要配置证书库的文件名，需要设置以下配置项：`jacorb.security.keystore=AKeystoreFileName`。
证书库文件可以是绝对路径，也可以相对于主目录的相关路径。ORB按上述方式搜索证书库，在找到第一个证书库时停止搜索。如果该配置项没以设置，在ORB启动时会提示用户输入证书库的路

径。
为了避免输入很多别名和密码(一组用于证书库，每个证书也需要一组)，可以使用以下配置项定义缺省的别名和密码：

```
# the name of the default key alias to look up in the keystore
jacorb.security.default_user=brose
jacorb.security.default_password=jacorb
```

可以进一步使用以下配置项来设置SSL：

```
jacorb.security.ssl.client.supported_options=0
jacorb.security.ssl.client.required_options=0
jacorb.security.ssl.server.supported_options=0
jacorb.security.ssl.server.required_options=0
```

这些配置项的值是十六进制的掩码，每一位的含义在CORBA安全服务规范中定义，以下取自JacORB的security.idl文件：

```
typedef unsigned short AssociationOptions;
const AssociationOptions NoProtection = 1;
const AssociationOptions Integrity = 2;
const AssociationOptions Confidentiality = 4;
const AssociationOptions DetectReplay = 8;
const AssociationOptions DetectMisordering = 16;
const AssociationOptions EstablishTrustInTarget = 32;
const AssociationOptions EstablishTrustInClient = 64;
const AssociationOptions NoDelegation = 128;
const AssociationOptions SimpleDelegation = 256;
const AssociationOptions CompositeDelegation = 512;
```

10.3.1 客户端配置

配置项`jacorb.security.ssl.client.supported_options=20`表示客户端使用SSL。实际上这是SSL的缺省配置，并且总是应该被客户端支持。
配置项`jacorb.security.ssl.client.supported_options=40`表明客户端从自己的证书库中载入公钥对和证书，因为客户端也要认证服务器。
配置项`jacorb.security.ssl.client.required_options=20`表示强制使用SSL。
配置项`jacorb.security.ssl.client.required_options=40`表示强制使用SSL。实际上此项无意义，因为客户端无法强制服务器用SSL来认证它(也许服务器不支持SSL呢)。

10.3.2 服务器端配置

配置项`jacorb.security.ssl.server.supported_options=1`向客户端表明服务器也支持不保护的连接。如果设为此项，则`required`的配置项无须设置，因为此项设置将覆盖它们的设置。
配置项`jacorb.security.ssl.server.supported_options=20`表明服务器支持SSL。实际上这是缺省的SSL行为，并且总应该被服务器支持。这也使得服务器从证书库中载入公钥对及证书。
配置项`jacorb.security.ssl.server.supported_options=40`被忽略，因为认证客户端要么已经要求，要么不支持(不能由客户端来要求被认证)。
配置项`jacorb.security.ssl.server.required_options=20`表示强制使用SSL。
配置项`jacorb.security.ssl.server.required_options=40`表示强制使用SSL，并要认证客户端。将从客户端获取证书进行认证。

11 双向 GIOP

双向GIOP主要用于小程序回调，或者有防火墙而不能直接访问目的主机时。例如，设想你想通过一个小程序来监视服务器的运行，这通常是通过在服务器上注册的小程序回调对象来完成的，而不用小程序时刻去查询服务器的事件。如果没有双向GIOP，要完成此功能，服务器必须新建一个到客户端的连接，而这对小程序无效，因为小程序不能作为服务器(例如，新建ServerSocket)。双向GIOP能起作用是因为它能重用小程序新建到服务器的连接来传递服务器到小程序的GIOP包(而这在标准GIOP中是不可能的)。

11.1 设置双向 GIOP

设置双向GIOP包含以下两个步骤：
1.设置ORB的初始属性及双向策略
2.将双向策略加入到伺服程序的POA中

11.1.1 设置 ORB 初始属性

第一件事是要设置以下配置项：
```
org.omg.PortableInterceptor.ORBInitializerClass.bidir_init=org.jacorb.orb.connection.BiDirConnectionInitializer
```
这样，在ORB启动时，相关策略工厂及解释器将被载入。

11.1.2 创建双向策略

通过策略工厂来创建双向策略：
```
import org.omg.BiDirPolicy.*;
import org.omg.CORBA.*;
[...]
Any any = orb.create_any();
BidirectionalPolicyValueHelper.insert( any, BOTH.value );
Policy p = orb.create_policy( BIDIRECTIONAL_POLICY_TYPE.value,any );
```

新策略的值传递给工厂any，ORB使用指定类型和值来创建一个策略，此策略用于生成POA。注意如果所有的POA都设置了此策略，则所有的连接都激活双向GIOP，甚至对于没有使用此策略的POA。完整的源代码可以在demo目录下找到。

11.2 验证双向 GIOP 已经使用

在应用程序内部，可以判断连接是标准的还是双向的。要么通过网络监视工具，要么察看JacORB的输出。
如果调试模式设为 2 或更高，服务器端以下输出表时连接被重用：
```
[ ConnectionManager: found conn to target <my IP>:<my port> ]
```
如果连接未被重用，则在客户端有如下输出：
```
[ Opened new server-side TCP/IP transport to <my host>:<my port> ]
```

11.3 TAO 协同性

在TAO和JacORB使用双向GIOP互操作时存在一个问题：如果JacORB使用IP地址作为主机名(JacORB缺省情况)，而TAO使用DNS名作为主机名(TAO缺省情况)，从JacORB客户端到TAO服务器的连接不能被重用。当然，如果二者使用同一格式的主机名，则能解决此问题。因此，有两种方法可以解决此问题：

1.使用`‘‘-ORBdotteddecimaladdresses 1’’`作为TAO服务器的命令行参数

2.重新编译JacORB使其支持DNS(更多信息请察看安装文件)

12 可移植拦截器-PI

从 1.1 版本开始JacORB支持可移植拦截器，其规范可参考http://cgi.omg.org/cgi-bin/doc?ptc/00-03-03
因此我们不提供PI的编程文档，而只是说明JacORB解决方案。
将拦截器集成进JacORB的第一步是通过以下格式的配置项注册拦截器：

```
org.omg.PortableInterceptor.ORBInitializerClass.<any_suffix>=
<orb initializer classname>
```

后缀仅用来区别不同的拦截器，而没有任何实际意义。配置项的值必须是拦截器的全类名(包含包)。
如果调试方式设置 2 或以上，当此类不在类路径中时JacORB会抛出一个类找不到的异常。
一个示例的配置如下：

```
org.omg.PortableInterceptor.ORBInitializerClass.my_init=
test.MyInterceptorInitializer
```

不过，规范不提供任何访问ORB的接口。如果你需要从拦截器外访问ORB，你需要将`ORBInitInfo`对象强制转换为`orb.portableInterceptor.ORBInitInfoImpl`并调用getORB()方法来获取ORB的引用。
当使用服务命名空间时，注意不要用`java.lang.Integer.MAX_VALUE`作为ID，应为此值已被内部使用。另外，也要注意使用完或出现异常时关闭ORB。

13 JacORB 实用程序

本章介绍JacORB提供的实用程序，包括idl及命名服务。

13.1 IDL

IDL编译器分析IDL接口文件并将期映射为Java类，IDL的接口翻译为Java的接口，其他各项也翻译为对应的Java类。使用IDL编译器自动生成所有接口的桩和骨架。

用法
```
idl [-h|-help] [-v|-version] [-syntax] [-all] [-Idir] [-
Dsymbol[=value]] [-d <Output Dir>] [-p <package prefix>] [-i2jpackage
<mapping>][-W debug level] <filelist>
```

-h | -help　显示用法帮助
-v | -version 显示版本信息
-noskel　禁止生成骨架文件
-ir 生成接口仓库需要的额外的文件，通常每个模块和接口各生成一个文件
-syntax 只进行语法检查，不生成文件
-all 也生成包含文件的代码，缺省条件下不生成
-I 指定搜索IDL文件的目录，缺省情况下只在当前目录搜索。
-D 定义编译器使用的符号，值缺省为 1。
-U 去掉符号定义。
-d 生成Java源文件的起始目录，缺省为当前目录。
-p 指定生成Java文件的包前缀，编译器会生成相应的目录结构。
-i2jpackage 将模块转定义为包。例如`-i2jpackage X:a.b.c`将模块X转化为包a.b.c，则模块X下接口ident，编译后为a.b.c.ident

IDL分析器使用Scott Hudson’s
CUP分析生成器生成，CORBA接口文件的LALR语法在`jacorb/Idl/parser.cup`文件中定义。

13.2 ns

JacORB提供命名服务，命名服务本身也是标准的CORBA服务，是通用对象服务规范中命名服务的直接实现。

用法
```
$ ns <filename> [<timeout>]
```
或
```
$ jaco jacorb.Naming.NameServer <filename> [<timeout>]
```

例如
```
$ ns ~/public html/NS Ref
```

命名服务没有知名的端口号。

13.3 nmg

JacORB命名服务管理器，图形化的管理工具。

用法
$nmg

13.4 lsns

列出根命名空间的的内容。只有已经注册并处于活动状态的服务才显示。-r选择表示递归显示。

用法
$lsns [-r]

示例
$lsns
当只有grid示例服务运行，并已注册到命名服务时显示：
/grid.service

13.5 dior

解码IRO字符串，使其可读性更好一些。

用法
```
$ dior <IOR-string> | -f <filename>
```

示例
```
$ dior -f ~/public html/NS Ref
------IOR components-----
TypeId : IDL:omg.org/CosNaming/NamingContextExt:1.0
Profile Id : TAG_INTERNET_IOP
IIOP Version : 1.0
Host : 160.45.110.41
Port : 49435
Object key : 0x52 6F 6F 74 50 4F 41 3A 3A 30 D7 D1 91 E1 70 95 04
```

13.6 pingo

ping一个对象查看是否存在，此命令会调用目的对象的non existent()方法。

用法
```
$ pingo <IOR-string> | -f <filename>
```

13.7 ir

启动接口仓库

用法

```
$ ir <reppository_calss_path> <IOR_filename>
```

13.8 qir

查询接口库，输出重新生成的IDL

用法

```
$ qir <reppository Id>
```

13.9 ks

启动证书库管理器

用法

```
$ ks
```

14 配置清单

完整的JacORB配置项及其含义见下表：

14.1 ORB 的配置

配置项	描述	类型
ORBInitRef.<Service>	配置可以通过 ORB 的 resolve_intial_references 方法获取的初始服务对象引用，支持文件和 HTTP 方式的 URL	URL
org.omg.PortableInterceptor.ORBInitializer.Class.<Name>	拦截器的 Java 类名	类
jacorb.ProxyServerURL	小程序代理的 URL	URL
jacorb.orb.print_version	JacORB 启动时是否显示版本号，缺省为是	On/off
jacorb.verbosity	调试模式	整型
jacorb.logfile	日志文件，缺省为标准输出	文件
jacorb.debug.dump_outgoing_messages	输出信息为十六进制格式，缺省为关闭	On/off
jacorb.debug.dump_incoming_messages	输入信息为十六进制格式，缺省为关闭	On/off
jacorb.giop_minor_version	最新的 GIOP 版本号，用于生成 IOR，缺省为 2	整型
jacorb.retries	连接不能正常建立时的重试次数，缺省为 5	整型
jacorb.retry_interval	重试之间的间隔，缺省为 500	毫秒
jacorb.outbuf_size	输出缓冲区的大小，缺省为 4096	字节
jacorb.maxManagedBufSize	这不是最大可用缓冲区的大小，而是最大保留和管理的缓冲区的大小。这个值会加上常数 5，因此真正原大小为 2*(5+maxSize-1)byte。当处理大的数据结构时可以加大此值，减少此值保证大的缓冲马上被释放。缺省为 18	整型
jacorb.connection.client_timeout	客户端超时，在请求阻塞多少时间后超时。缺省不设置超时	毫秒
jacorb.connection.server_timeout	服务器保持空闲连接的最大时间。缺省不设置超时	毫秒
jacorb.reference_caching	是否缓存对象引用。缺省不设置	On/off

配置项	描述	类型
jacorb.hashtable_class	指定用于引用缓存的哈希类，WeakHashtable 使用 WeakReferences，因此当只有哈希表中引用时，可以被垃圾收集器，这对于大量短生命周期的持久对象很有用。要求使用 JDK1.2 以上版本。而标准的 Hashtable 则会一直保留对象引用直至调用 _release()才显式删除	类
jacorb.use_bom	使用 GIOP1.2 字节顺序标记，从 CORBA2.4/2.5 起。缺省为关闭	On/off
jacorb.giop.add_1_0_profiles	加入额外的 IIOP1.0 说明，即使是使用 IIOP1.2。缺省为关闭	On/off
org.omg.PortableInterceptor.ORBInitializerClass.bidir_init	要支持双向 GIOP，此拦截器的设置。缺省不设置	类
jacorb.ior_proxy_host	在 ServerSocket 不能访问时(如有防火墙时)，使用配置项 jacorb.ior_proxy_host 及 jacorb.ior_proxy_port 来告诉 ORB 应该包含什么 host/port 的 IOR。慎用此选项，缺省无设置	主机
jacorb.ior_proxy_port	见上述	端口号
OAIAddr	对象适配器的 IP 地址，在有多个 IP 的主机中使用此 IP 地址来生成对象引用。注意 127.0.0.x 的只能被本机访问。缺省无设置	主机
OAPort	见上述	端口号
jacorb.use_appligator	设置小程序是否使用小程序代理。缺省为关闭	On/off
org.omg.PortableInterceptor.ORBInitializerClass.standard_init	标准拦截器，不要删除此配置项	类

14.2 POA 的配置

配置项	描述	类型
jacorb.poa.monitoring	启动服务器的图形化监视器。缺省为关闭	On/off
jacorb.poa.thread_pool_max	POA 线程池的最大值	整型
jacorb.poa.thread_pool_min	POA 线程池的最小值	整型

配置项	描述	类型
jacorb.poa.thread_priority	线程的优先级，如果不设置或设置错误，使用最高优先级。缺省不设置	整型
jacorb.poa.queue_max	请求队例的长度。如果已经达到此值，后续的请求将收到一个 Corba.TRANSIENT 异常。缺省为 100	整型

14.3 实现仓库的配置

配置项	描述	类型
jacorb.use_imr	是否服务启动时都和实现仓库通信。缺省为关闭	On/off
jacorb.imr.allow_auto_register	是否服务启动时自动注册到实现仓库。缺省为关闭	On/off
ORBInitRef.ImplementationRepository	实现仓库的初始引用	URL
jacorb.imr.table_file	实现仓库存储数据的文件	FILE
jacorb.imr.backup_file	实现仓库备份数据的文件	FILE
jacorb.imr.ior_file	实现仓库写入 IOR 的文件	FILE
jacorb.imr.timeout	实现仓库等待服务注册的时间，超时的服务被实现仓库认为启动失败。缺省为 2000	毫秒
jacorb.imr.no_of_poas	随实现仓库一起启动的 POA 数量，此值用于优化内部数据存储，可以被超过。缺省值为 100	整型
jacorb.imr.no_of_servers	随实现仓库一起启动的服务数量，此值用于优化内部数据存储，可以被超过。缺省值为 5	整型
jacorb.imr.connection_timeout	实现仓库等待客户端连接的超时设置。缺省为 2000	毫秒
jacorb.implname	持久服务的实现名，这是在实现仓库中注册的服务名。	名
jacorb.java_exec	实现仓库启动的命令	命令字符串

14.4 安全的配置

配置项	描述	类型
OASSLPort	SSL 使用的端口号。缺省为动态分配	端口号
org.omg.PortableInterceptor.ORBInitializerClass.	被要求使用 SSL 的可移植拦截器。缺省无设置	类

配置项	描述	类型
ForwardInit	无设置	
jacorb.security.principal_authenticator	认证类的全名列表，以逗号分隔	类列表
jacorb.ssl.socket_factory	Socket 工厂类全名	类
jacorb.security.change_ssl_roles	交换客户端和服务器的角色以强制进行客户端验证。注意，如果一方未实现此功能，会带来一些问题。缺省为关闭	On/off
jacorb.security.support_ssl	是否启动 SSL。缺省为关闭	On/off
jacorb.security.ssl.client.supported_options	SSL 客户端支持选项	整型
jacorb.security.ssl.client.required_options	SSL 客户端要求选项	整型
jacorb.security.ssl.server.supported_options	SSL 服务器支持选项	整型
jacorb.security.ssl.server.required_options	SSL 服务器要求选择	整型
jacorb.security.keystore	证书库的位置	FILE
jacorb.security.keystore_password	证书库的口令	字符串
jacorb.security.trustees	受信任的认证中心公钥证书列表文件，注意不设置，IAIK 认为所有认证中心签名的证书都是正确的	文件
jacorb.security.default_user	查找证书库时使用缺省别名	名字符串
jacorb.security.default_password	上述别名的密码	字符串
jacorb.security.iaik_debug	是否输出 IAIK 的调试信息。缺省为关闭	On/off
jacorb.security.jsse.trustees_from_ks	Sun JSSE 设置：从证书库中取信任的认证中心列表。缺省为关闭	On/off

15 缺陷和反馈

缺陷列表(并不完整)
1.IDL编译器不支持context结构
2.API文档及本文都不完整

有任何反馈及bug报告请发邮件至brose@inf.fu-berlin.de

译者注：对本译文有任何反馈及bug报告请发邮件至hlstudio@sina.com，或进入www.huihoo.com的论坛，关于JacORB的技术讨论也请访问该网站。

16 附录

[BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA.* John Wiley& Sons, 3rd edition, 2001.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison–Wesley, 1999.

[OMG97] OMG. *CORBAservices: Common Object Services Specification*, November 1997.

[Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming*. Wiley, 2nd edition, 2000.

[Vin97] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments.*IEEE Communications Magazine*, 14(2), February 1997.

[Vin98] Steve Vinoski. New features for corba 3.0. *CACM*, 41(10):44–52, October 1998.