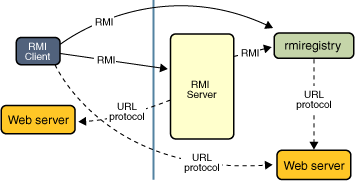
Java RMI介绍

Java RMI(Remote Method Invocation)，是一种基于JAVA的远程方法调用技术，是Java特有的一种RPC实现。能够使部署在不同主机上的Java对象之间进行透明的通信与方法调用，如下图所示：

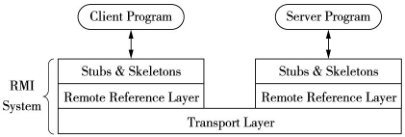


在详解RMI之前，需要了解分布式对象（Distributed Object），指一个对象可以被远程系统调用。对于Java而言，即对象不仅可以被同一个虚拟机中的其他客户端（Client）程序调用，也可以被运行于其他虚拟机中的客户端程序调用，设置可以通过网络被其他远程主机上的客户程序调用。如上图中，分布式对象被调用的过程如下：

* 客户程序调用一个被称为Stub的客户端代理对象，该对象负责对客户端隐藏网络通信细节。Stub知道如何通过Socket发送调用，包括如何将调用参数转换为适当形式以便传输等。
* Stub通过网络将调用传递给服务器端，也就是分布式对象一端的Skeleton的代理对象，同样该代理对象负责对分布式对象隐藏网络通信细节。Skeleton知道如何将Socket中接受调用，包括如何将调用参数从网络传输形式转换为Java形式等
* Skeleton将调用传递给分布式对象，执行相应的操作，然后将返回值传递给Skeleton，进而传递给Stub，最终返回给客户端程序。

基本法则是将行为的定义和行为的具体实现相分离，客户端代理对象Stub和分布式对象都实现相同接口，该接口称为远程接口（Remote Interface）。

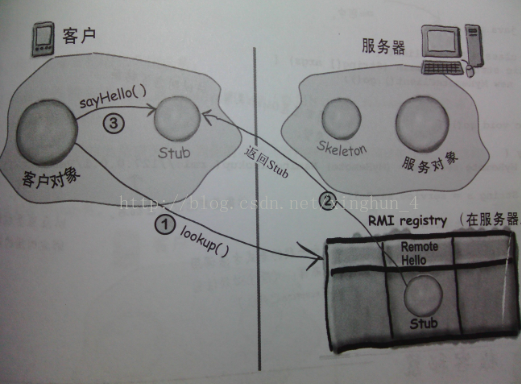
# 1.RMI的原理



RMI的底层由三层构成：

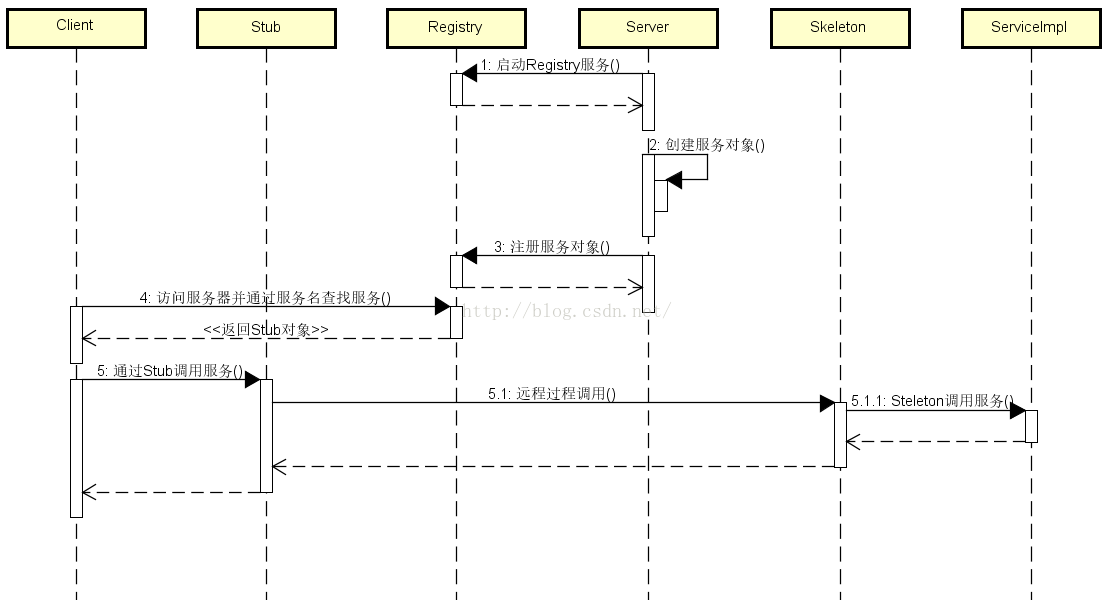
* Stub/Skeleton层，该层提供了客户程序和服务程序交互的接口
* 远程引用层（Remote Reference），相当于在其上的Stub/Skeleton层及其下的Transport层之间的中间件，负责处理远程对象引用的创建和管理
* 传输协议（Transport Protocol）层，该层提供数据协议，通过线路传输客户程序和远程对象间的请求和应答

其中Stub和Skeleton代理在程序中由RMI系统动态生成，如下图所示：



# 2.RMI源码分析

RMI封装了远程调用的细节部分，比如序列化和反序列化、连接的建立和释放等，下面是RMI的具体执行过程：



RMI的核心组件是Registry，提供服务名到服务映射，相关类图如下：



## 2.1 启动Registry

服务端启动Registry服务，如下所示：

*LocateRegistry.createRegistry(8081);*

Server端创建RegistryImpl对象，指定端口为8081

*public RegistryImpl(final int var1) throws RemoteException {*

*if(var1 == 1099 && System.getSecurityManager() != null) {*

*……*

*LiveRef var1x = new LiveRef(RegistryImpl.id, var1);*

*RegistryImpl.this.setup(new UnicastServerRef(var1x, (var0) -> {*

*return RegistryImpl.registryFilter(var0);*

*}));*

*….*

*} else {*

*LiveRef var2 = new LiveRef(id, var1);*

*this.setup(new UnicastServerRef(var2, RegistryImpl::registryFilter));*

*}}*

在setup方法中将初始化的RegistryImpl对象的远程引用ref赋值给传入的UnicastServerRef对象，然后移交给UnicastServerRef的exportObject方法：

*private void setup(UnicastServerRef var1) throws RemoteException {*

*this.ref = var1;*

*var1.exportObject(this, (Object)null, true);*

*}*

在exportObject中为RegistryImpl创建一个代理，代码如下：

*public Remote exportObject(Remote var1, Object var2, boolean var3) throws RemoteException {*

*Class var4 = var1.getClass();*

*Remote var5;*

*try {*

*var5 = Util.createProxy(var4, this.getClientRef(), this.forceStubUse);*

*} catch (IllegalArgumentException var7) {*

*throw new ExportException("remote object implements illegal remote interface", var7);*

*}*

*if(var5 instanceof RemoteStub) {*

*this.setSkeleton(var1);*

*}*

*Target var6 = new Target(var1, this, var5, this.ref.getObjID(), var3);*

*this.ref.exportObject(var6);*

*this.hashToMethod\_Map = (Map)hashToMethod\_Maps.get(var4);*

*return var5;*

*}*

将UnicastServerRef的skel对象设置为当前的RegistryImpl对象，然后用skeleton,stub,UnicastServerRef,id等构造一个Target对象。在exportObject中，调用TCPTransport的exportObject方法，并调用listen方法，创建一个ServerSocket，启动线程并等待客户端请求：

*public void exportObject(Target var1) throws RemoteException {*

*synchronized(this) {*

*this.listen();*

*++this.exportCount;*

*}*

*super.exportObject(var1);*

*}*

接着调用父类Tranport的exportObject将Target对象放入到ObjectTable中。

## 2.2 客户端获取服务端Registry代理

客户端通过下面代码取服务端Registry代理:

*IOperation iOperation= (IOperation) Naming.lookup("rmi://127.0.0.1:1099/Operation");*

Nameing.lookup中调用LocateRegistry的getRegistry：

*return LocateRegistry.getRegistry(parsed.host, parsed.port);*

通过传入host和port构造RemoteRef对象，并创建一个本地代理，如下所示：

*getRegistry(String host, int port,RMIClientSocketFactory csf)*

*throws RemoteException{*

*Registry registry = null;*

*if (port <= 0)*

*port = Registry.REGISTRY\_PORT;*

*LiveRef liveRef =*

*new LiveRef(new ObjID(ObjID.REGISTRY\_ID),*

*new TCPEndpoint(host, port, csf, null),*

*false);*

*RemoteRef ref =*

*(csf == null) ? new UnicastRef(liveRef) : new UnicastRef2(liveRef);*

*return (Registry) Util.createProxy(RegistryImpl.class, ref, false);*

*}*

在createProxy中创建RemoteStub，如下所示：

*public static Remote createProxy(Class<?> var0, RemoteRef var1, boolean var2) throws StubNotFoundException {*

*Class var3;*

*var3 = getRemoteClass(var0);*

*if(var2 || !ignoreStubClasses && stubClassExists(var3)) {*

*return createStub(var3, var1);*

*}*

*}*

代理和远程的Registry之间通过Socket来建立连接。

## 2.3服务端创建对象

从OpertionImpl的构造函数看起，调用父类UnicastRemoteObject的构造方法，其中调用exportObject方法，将对象传入到UnicastServerRef对象，反之调用UnicastServerRef的exportObject方法，如下所示：

*private static Remote exportObject(Remote obj, UnicastServerRef sref)*

*throws RemoteException*

*{*

*// if obj extends UnicastRemoteObject, set its ref.*

*if (obj instanceof UnicastRemoteObject) {*

*((UnicastRemoteObject) obj).ref = sref;*

*}*

*return sref.exportObject(obj, null, false);*

*}*

## 2.4将服务绑定到服务端的Registry上，使得客户端只与Registry交互

*Naming.bind("rmi://127.0.0.1:1099/Operation",iOperation);*

从代码上看，Naming中的方法全部是调用Registry，通过host和port找到第一步启动的服务端的Registry对象，把名字和服务存入到一个Map中：

*public static void bind(String name, Remote obj)*

*throws AlreadyBoundException,java.net.MalformedURLException,RemoteException{*

*ParsedNamingURL parsed = parseURL(name);*

*Registry registry = getRegistry(parsed);*

*if (obj == null)*

*throw new NullPointerException("cannot bind to null");*

*registry.bind(parsed.name, obj);*

*}*

## 2.5客户端查找远程服务

查找服务，如下：

*IOperation iOperation= (IOperation) Naming.lookup("rmi://127.0.0.1:1099/Operation");*

客户端获取远程代理对象后，调用RegistryImpl\_Stub的lookUp方法，通过服务端host和port等信息创建的Sub对象构建RemoteCall调用对象，operations参数中是各个Registry声明的操作：

*public Remote lookup(String var1) throws AccessException, NotBoundException, RemoteException {*

*try {*

*RemoteCall var2 = super.ref.newCall(this, operations, 2, 4905912898345647071L);*

*try {*

*ObjectOutput var3 = var2.getOutputStream();*

*var3.writeObject(var1);*

*} catch (IOException var18) {*

*throw new MarshalException("error marshalling arguments", var18);*

*}*

*super.ref.invoke(var2);*

*Remote var23;*

*try {*

*ObjectInput var6 = var2.getInputStream();*

*var23 = (Remote)var6.readObject();*

*}*

*}*

调用RegistryImpl\_Stub的ref对象newCall方法，将Stub对象传进入，建立与远程RegistryImpl的Skeleton对象的连接。

*public RemoteCall newCall(RemoteObject var1, Operation[] var2, int var3, long var4) throws RemoteException {*

*Connection var6 = this.ref.getChannel().newConnection();*

*StreamRemoteCall var7 = new StreamRemoteCall(var6, this.ref.getObjID(), var3, var4);*

*......*

*return var7;*

*}*

*}*

在RegistryImpl\_Stub中支持的OPERATION包括如下：

*private static final Operation[] operations = new Operation[]{*

*new Operation("void bind(java.lang.String, java.rmi.Remote)"),*

*new Operation("java.lang.String list()[]"),*

*new Operation("java.rmi.Remote lookup(java.lang.String)"),*

*new Operation("void rebind(java.lang.String, java.rmi.Remote)"),*

*new Operation("void unbind(java.lang.String)")*

*};*

至此，已经将客户端的服务查询请求发出去了。

## 2.6服务端接收客户端的服务查询请求并返回给客户端结果

服务端接收RegistryImpl\_Stub发送的lookup请求后，调用Transport.servicecCall执行客户端请求：

*public boolean serviceCall(final RemoteCall var1) {*

*try {*

*ObjID var39;*

*try {*

*var39 = ObjID.read(var1.getInputStream());*

*} catch (IOException var33) {*

*throw new MarshalException("unable to read objID", var33);*

*}*

*Transport var40 = var39.equals(dgcID)?null:this;*

*Target var5 = ObjectTable.getTarget(new ObjectEndpoint(var39, var40));*

*final Remote var37;*

*final Dispatcher var6 = var5.getDispatcher();*

*var5.incrementCallCount();*

*boolean var8;*

*try {*

*final AccessControlContext var7 = var5.getAccessControlContext();*

*ClassLoader var41 = var5.getContextClassLoader();*

*ClassLoader var9 = Thread.currentThread().getContextClassLoader();*

*try {*

*setContextClassLoader(var41);*

*currentTransport.set(this);*

*try {*

*AccessController.doPrivileged(new PrivilegedExceptionAction() {*

*public Void run() throws IOException {*

*Transport.this.checkAcceptPermission(var7);*

*var6.dispatch(var37, var1);*

*return null;*

*}*

*}, var7);*

*return true;*

*}*

*}……*

再由Target派发到Dispatcher，最终调用RegistryImpl的lookUp方法，也就是Skeleton，来负责底层的操作：

*public Remote lookup(String var1) throws RemoteException, NotBoundException {*

*Hashtable var2 = this.bindings;*

*synchronized(this.bindings) {*

*Remote var3 = (Remote)this.bindings.get(var1);*

*if(var3 == null) {*

*throw new NotBoundException(var1);*

*} else {*

*return var3;*

*}*

*}*

*}*

## 2.7 客户端通过lookup查询后去客户端OperationImpl的Stub对象

*public Remote lookup(String var1) throws AccessException, NotBoundException, RemoteException {*

*try {*

*RemoteCall var2 = super.ref.newCall(this, operations, 2, 4905912898345647071L);*

*try {*

*ObjectOutput var3 = var2.getOutputStream();*

*var3.writeObject(var1);*

*} catch (IOException var18) {*

*throw new MarshalException("error marshalling arguments", var18);*

*}*

*super.ref.invoke(var2);*

*Remote var23;*

*try {*

*ObjectInput var6 = var2.getInputStream();*

*var23 = (Remote)var6.readObject();*

*} catch (IOException var15) {*

*throw new UnmarshalException("error unmarshalling return", var15);*

*} catch (ClassNotFoundException var16) {*

*throw new UnmarshalException("error unmarshalling return", var16);*

*} finally {*

*super.ref.done(var2);*

*}*

*return var23*

*}*

## 2.8 客户端进行远程服务调用

客户端已经构建了Stub对象，可以与服务端进行调用。RMI中通过RegistryImpl\_Skel和RegistryImpl\_Stub屏蔽远程调用的细节。

## 3.代码示例

1）定义服务接口

*public interface HelloService extends Remote {*

*String sayHello(String someOne) throws RemoteException;*

*}*

2）定义实现类

*public class HelloServiceImpl extends UnicastRemoteObject implements HelloService {*

*private static final long serialVersionUID = -1;*

*public HelloServiceImpl() throws RemoteException {*

*super();*

*}*

*public String sayHello(String someOne) throws RemoteException {*

*return "Message from Server : Hello, "+ someOne;*

*}*

*}*

继承UnicatRemoteObject抽象类

3）定义服务端程序

*public class RMIServer {*

*public static void main(String[] args) throws Exception {*

*HelloService service = new HelloServiceImpl();*

*LocateRegistry.createRegistry(8081);*

*Naming.bind("rmi://192.168.246.153:8081/helloService", service);*

*System.out.println("rmi server provide RPC service now");*

*}*

*}*

4）定义客户端程序

*public class RMIClient {*

*public static void main(String[] args) throws Exception {*

*HelloService clientService = (HelloService) Naming.lookup("rmi://192.168.246.153:8081/helloService");*

*System.out.println(clientService.sayHello("fys")); }}*

5）客户端执行结果

*Message from Server : Hello, fys*

*Process finished with exit code 0*

RMI底层分析：

http://blog.csdn.net/sinat\_34596644/article/details/52599688

Jaa RMI入门指南：

http://www.blogjava.net/boddi/archive/2006/10/11/74430.html