



咕泡学院 VIP 课：分布式消息通信框架 RMI 原理分析

课程目标

1. 什么是 RPC
2. 实现 JAVA RMI
3. RPC 框架原理
4. 了解 Java RMI
5. 基于 Java RMI 实践
6. RMI 通信原理分析
7. 实现自己的 RPC 框架

什么是 RPC

RPC (Remote Procedure Call, 远程过程调用)，一般用来实现部署在不同机器上的系统之间的方法调用，使得程序能够像访问本地系统资源一样，通过网络传输去访问远端系统资源；对于客户端来说，传输层使用什么协议，序列化、反序列化都是透明的

了解 Java RMI

RMI 全称是 remote method invocation – 远程方法调用，一种用于远程过程调用的应用程序编程接口，是纯 java 的网络分布式应用系统的核心解决方案之一。

RMI 目前使用 Java 远程消息交换协议 JRMP (Java Remote Messaging Protocol) 进行通信，由于 JRMP 是专为 Java 对象制定的，是分布式应用系统的百分之百纯 java 解决方案，用 Java RMI 开发的应用系统可以部署在任何支持 JRE 的平台上，缺点是，由于 JRMP 是专门为 java 对象指定的，因此 RMI 对于非 JAVA 语言开发的应用系统的支持不足，不能与非 JAVA 语言书写的对象进行通信

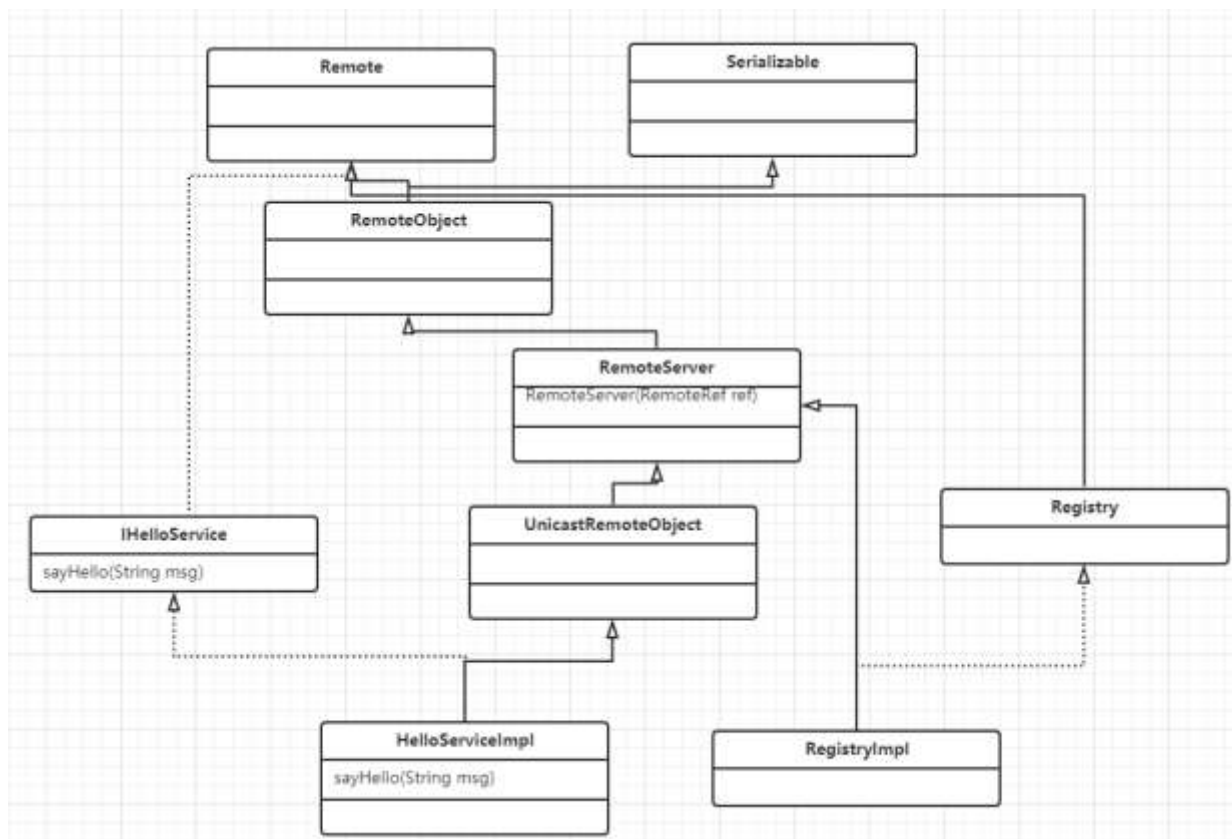
Java RMI 代码实践

详见代码

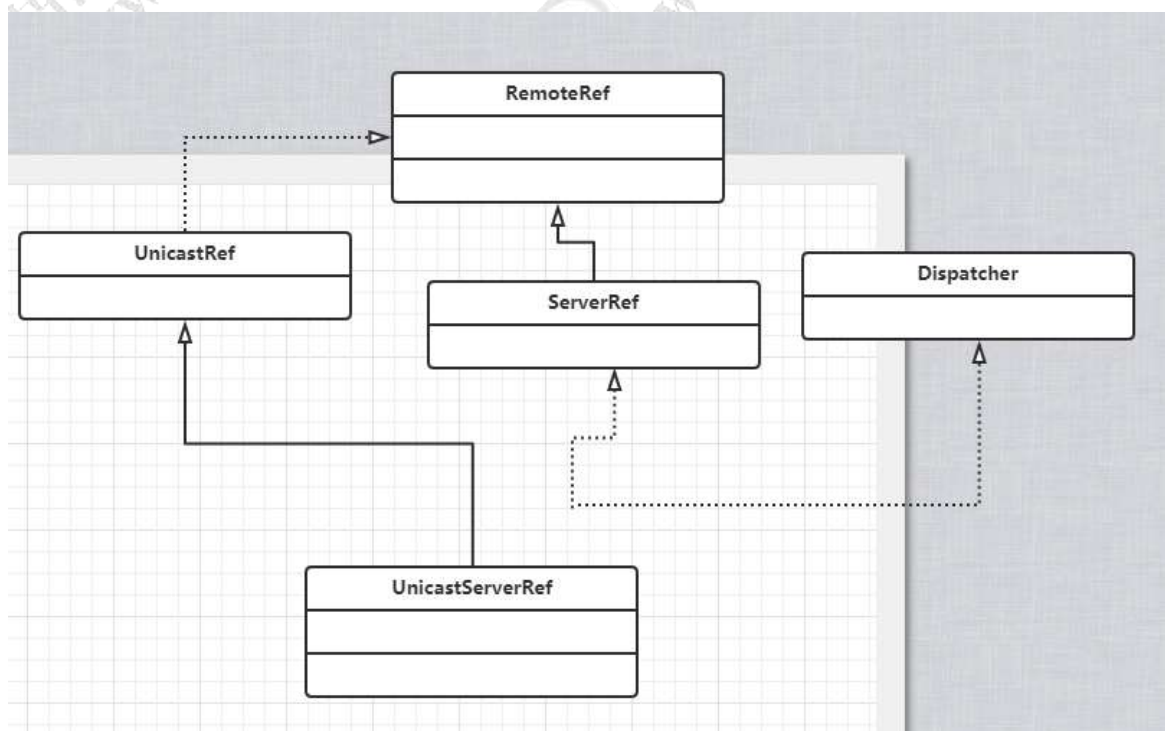
远程对象必须实现 `UnicastRemoteObject`，这样才能保证客户端访问获得远程对象时，该远程对象会把自身的一个拷贝以 `Socket` 形式传输给客户端，客户端获得的拷贝称为“`stub`”，而服务器端本身已经存在的远程对象成为“`skeleton`”，此时客户端的 `stub` 是客户端的一个代理，用于与服务器端进行通信，而 `skeleton` 是服务端的一个代理，用于接收客户端的请求之后调用远程方法来响应客户端的请求

Java RMI 源码分析

远程对象发布



远程引用层



一步步解读源码

发布远程对象

发布远程对象，看到上面的类图可以知道，这个地方会发布两个远程对象，一个是 RegistryImpl、另外一个是我们自己写的 RMI 实现类对象；

从 HelloServiceImpl 的构造函数看起。调用了父类 UnicastRemoteObject 的构造方法，追溯到 UnicastRemoteObject 的私有方法 exportObject()。这里做了一个判断，判断服务的实现是不是 UnicastRemoteObject 的子类，如果是，则直接赋值其 ref (RemoteRef) 对象为传入的 UnicastServerRef 对象。反之则调用 UnicastServerRef 的 exportObject()方法。

```
IHelloService helloService=new HelloServiceImpl();
```

因为 HelloServiceImpl 继承了 UnicastRemoteObject，所以在服务启动的时候，会通过 UnicastRemoteObject 的构造方法把该对象进行发布

```
public class HelloServiceImpl extends UnicastRemoteObject :  
  
    protected HelloServiceImpl() throws RemoteException {  
        super();  
    }
```

```

public static Remote exportObject(Remote obj, int port)
    throws RemoteException
{
    创建UnicastServerRef对象，对象内有引用了LiveRef（tcp通信）
    return exportObject(obj, new UnicastServerRef(port));
}

```

```

public Remote exportObject(Remote var1, Object var2, boolean var3) throws RemoteException {
    Class var4 = var1.getClass();

    Remote var5;
    try {
        创建远程代理类，该代理是对OperationImpl对象的代理，getClientRef提供的InvocationHandler中提供了TCP连接
        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);
    }
    包装实际对象，并将其暴露在TCP端口上，等待客户端调用

    Target var6 = new Target(var1, dispatcher: this, var5, this.ref.getObjID(), var3);
    this.ref.exportObject(var6);
    this.hashToMethod_Map = (Map)hashToMethod_Maps.get(var4);
    return var5;
}

```

服务端启动 Registry 服务

```
LocateRegistry.createRegistry(port: 1099);
```

从上面这段代码入手，开始往下看。可以发现服务端创建了一个 RegistryImpl 对象，这里做了一个判断，如果服务端指定的端口是 1099 并且系统开启了安全管理器，那么就可以在限定的权限集内绕过系统的安全校验。这里纯粹是为了提高效率，真正的逻辑在 this.setup(new UnicastServerRef())这个方法里面


```

public RegistryImpl(final int var1) throws RemoteException {
    if(var1 == 1099 && System.getSecurityManager() != null) {
        try {
            AccessController.doPrivileged(run() -> {
                LiveRef var1x = new LiveRef(RegistryImpl.id, var1);
                RegistryImpl.this.setup(new UnicastServerRef(var1x, (var0) -> {
                    return RegistryImpl.registryFilter(var0);
                }));
            });
        } catch (PrivilegedActionException var3) {
            throw (RemoteException)var3.getException();
        }
    } else {
        LiveRef var2 = new LiveRef(id, var1);
        this.setup(new UnicastServerRef(var2, RegistryImpl::registryFilter));
    }
}

```

[< 1.8 >] java.security
public final class AccessControlContext extends Object

setup 方法将指向正在初始化的 RegistryImpl 对象的远程引用 ref(RemoteRef)赋值为传入的 UnicastServerRef 对象，这里涉及到向上转型，然后继续执行 UnicastServerRef 的 exportObject 方法

```

private void setup(UnicastServerRef var1) throws RemoteException {
    this.ref = var1;
    var1.exportObject(remote: this, (Object)null, b: true);
}

```

进入 UnicastServerRef 的 exportObject() 方法。可以看到，这里首先为传入的 RegistryImpl 创建一个代理，这个代理我们可以推断出就是后面服务于客户端的 RegistryImpl 的 Stub (RegistryImpl_Stub) 对象。然后将 UnicastServerRef 的 skel (skeleton) 对象设置为当前 RegistryImpl 对象。最后用 skeleton、stub、UnicastServerRef 对象、id 和一个

boolean 值构造了一个 Target 对象，也就是这个 Target 对象基本上包含了全部的信息，等待 TCP 调用。调用 UnicastServerRef 的 ref (LiveRef) 变量的 exportObject() 方法。

【var1=RegistryImpl ; var 2 = null ; var3=true】

LiveRef 与 TCP 通信的类

```
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, dispatcher: this, var5, this.ref.getObjID(), var3);
    this.ref.exportObject(var6);
    this.hashToMethod_Map = (Map)hashToMethod_Maps.get(var4);
    return var5;
}
```

到上面为止，我们看到的都是一些变量的赋值和创建工作，还没有到连接层，这些引用对象将会被 Stub 和 Skeleton 对象使用。接下来就是连接层上的了。追溯 LiveRef 的 exportObject() 方法，很容易找到了 TCPTransport 的 exportObject() 方法。这个方法做的事情就是将上面构造的 Target 对象暴露出去。调用 TCPTransport 的 listen() 方法，listen() 方法创建了一个 ServerSocket，并且启动了一条线程等待客户端的请求。接着调用父类 Transport 的

exportObject()将 Target 对象存放在 ObjectTable 中。

```
public void exportObject(Target var1) throws RemoteException {  
    synchronized(this) {  
        this.listen();  
        ++this.exportCount;  
    }  
  
    boolean var2 = false;  
    boolean var12 = false;  
  
    try {  
        var12 = true;  
        super.exportObject(var1);  
        var2 = true;  
        var12 = false;  
    } finally {  
        if(var12) {  
            if(!var2) {  
                synchronized(this) {  
                    this.decrementExportCount();  
                }  
            }  
        }  
    }  
}
```

到这里，我们已经将 RegistryImpl 对象创建并且起了服务等待客户端的请求。

客户端获取服务端 Registry 代理

```
IService helloService=  
    (IService)Naming.lookup("rmi://127.0.0.1/Hello");
```

从上面的代码看起，容易追溯到 LocateRegistry 的

getRegistry()方法。这个方法做的事情是通过传入的 host 和 port 构造 RemoteRef 对象，并创建了一个本地代理。这个代理对象其实是 RegistryImpl_Stub 对象。这样客户端便有了服务端的 RegistryImpl 的代理（取决于 ignoreStubClasses 变量）。但注意此时这个代理其实还没有和服务端的 RegistryImpl 对象关联，毕竟是两个 VM 上面的对象，这里我们也可以猜测，代理和远程的 Registry 对象之间是通过 socket 消息来完成的。

```
public static Registry getRegistry(String host, int port,
                                   RMIClientSocketFactory csf)
    throws RemoteException
{
    Registry registry = null;

    if (port <= 0)
        port = Registry.REGISTRY_PORT; // 获取仓库地址

    if (host == null || host.length() == 0) {
        // If host is blank (as returned by "file:" URL in 1.0.2 used in
        // java.rmi.Naming), try to convert to real local host name so
        // that the RegistryImpl's checkAccess will not fail.
        try {
            host = java.net.InetAddress.getLocalHost().getHostAddress();
        } catch (Exception e) {
            // If that failed, at least try "" (localhost) anyway...
            host = "";
        }
    }

    LiveRef liveRef =
        new LiveRef(new ObjID(ObjID.REGISTRY_ID), // 与TCP通信的类
                   new TCPEndpoint(host, port, csf, rmiServerSocketFactory: null),
                   b: false);

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

    // 创建远程代理类，引用liveRef，好让动态代理时，能进行TCP通信
    return (Registry) Util.createProxy(RegistryImpl.class, ref, b: false);
}
```

调用 RegistryImpl_Stub 的 ref (RemoteRef) 对象的 newCall()方法, 将 RegistryImpl_Stub 对象传了进去, 不要忘了构造它的时候我们将服务器的主机端口等信息传了进去, 也就是我们把服务器相关的信息也传进了 newCall()方法。newCall()方法做的事情简单来看就是建立了跟远程 RegistryImpl 的 Skeleton 对象的连接。(不要忘了上面我们说到过服务端通过 TCPTransport 的 exportObject()方法等待着客户端的请求)

```
public RemoteCall newCall(RemoteObject var1, Operation[] var2, int var3, long var4) throws RemoteException {
    clientRefLog.log(Log.BRIEF, s: "get connection");
    Connection var6 = this.ref.getChannel().newConnection();

    try {
        clientRefLog.log(Log.VERBOSE, s: "create call context");
        if(clientCallLog.isLoggable(Log.VERBOSE)) {
            this.logClientCall(var1, var2[var3]);
        }

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

        try {
            this.marshalCustomCallData(var7.getOutputStream());
        } catch (IOException var9) {
            throw new MarshalException("error marshaling custom call data");
        }

        return var7;
    } catch (RemoteException var10) {
        this.ref.getChannel().free(var6, b: false);
        throw var10;
    }
}
```

连接建立之后自然就是发送请求了。我们知道客户端终究只是拥有 Registry 对象的代理, 而不是真正地位于服务端

的 Registry 对象本身，他们位于不同的虚拟机实例之中，无法直接调用。必然是通过消息进行交互的。看看 `super.ref.invoke()` 这里做了什么？容易追溯到 `StreamRemoteCall` 的 `executeCall()` 方法。看似本地调用，但其实很容易从代码中看出来是通过 tcp 连接发送消息到服务端。由服务端解析并且处理调用。

```
public Remote lookup(String var1) throws AccessException, NotBoundException, RemoteException {
    try {
        RemoteCall var2 = super.ref.newCall(obj: this, operations, opnum: 2, hash: 4905912898345647071L);

        try {
            ObjectOutputStream var3 = var2.getOutputStream();
            var3.writeObject(var1);
        } catch (IOException var18) {
            throw new MarshalException("error marshalling arguments", var18);
        }

        super.ref.invoke(var2);

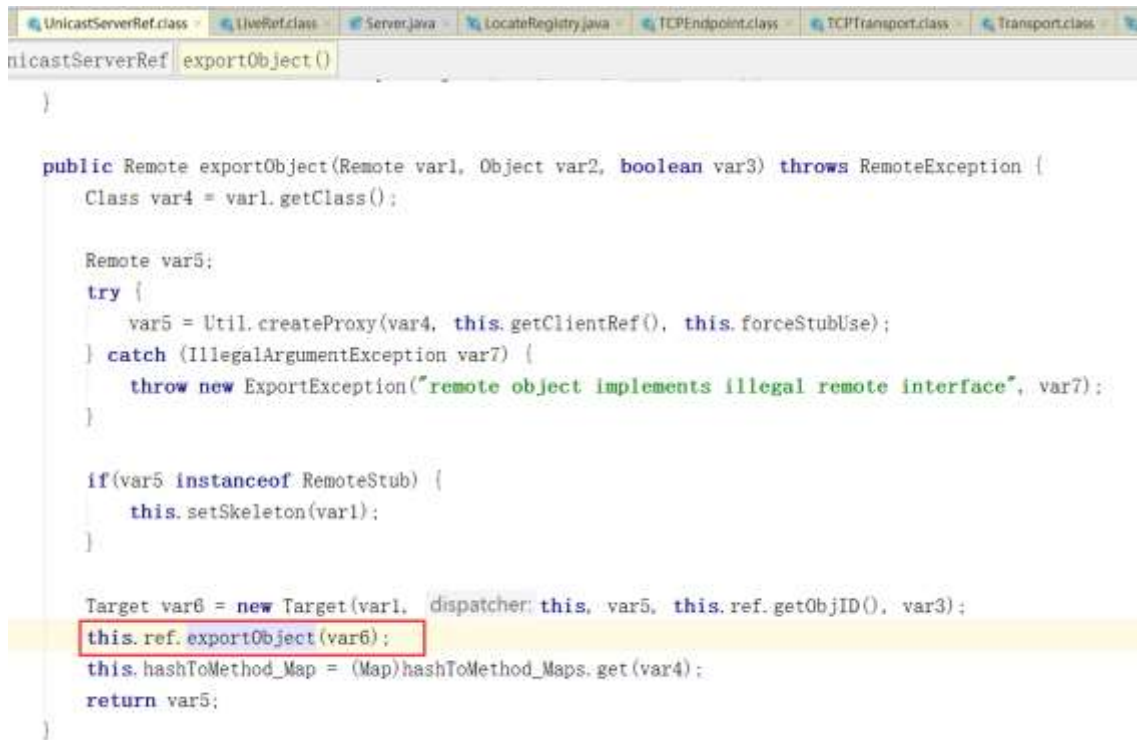
        Remote var23;
        try {
            ObjectInput var6 = var2.getInputStream();
            var23 = (Remote)var6.readObject();
        }
    }
}
```

```
public void invoke(RemoteCall var1) throws Exception {  
    try {  
        clientRefLog.log(Log.VERBOSE, s: "execute call");  
        var1.executeCall();  
    } catch (RemoteException var3) {  
        clientRefLog.log(Log.BRIEF, s: "exception: ", var3);  
        this.free(var1, var2: false);  
        throw var3;  
    } catch (Error var4) {  
        clientRefLog.log(Log.BRIEF, s: "error: ", var4);  
        this.free(var1, var2: false);  
        throw var4;  
    } catch (RuntimeException var5) {  
        clientRefLog.log(Log.BRIEF, s: "exception: ", var5);  
        this.free(var1, var2: false);  
        throw var5;  
    } catch (Exception var6) {  
        clientRefLog.log(Log.BRIEF, s: "exception: ", var6);  
        this.free(var1, var2: true);  
        throw var6;  
    }  
}
```

至此，我们已经将客户端的服务查询请求发出了。

服务端接收客户端的服务查询请求并返回给客户端结果

这里我们继续跟踪 server 端代码的服务发布代码，一步步往上面翻。按照下图顺序



```
UnicastServerRef.class LiveRef.class Server.java LocateRegistry.java TCPEndpoint.class TCPTransport.class Transport.class
UnicastServerRef exportObject()

}

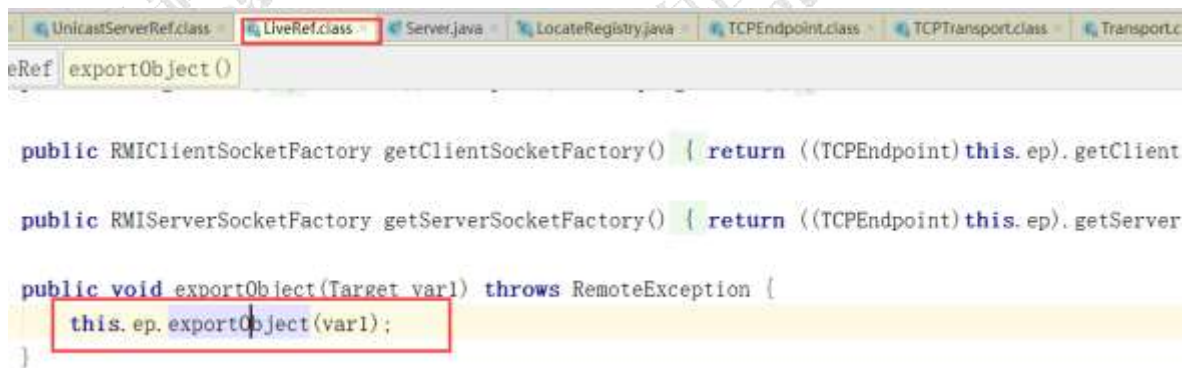
public Remote exportObject(Remote var1, Object var2, boolean var3) throws RemoteException {
    Class var4 = var1.getClass();

    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, dispatcher: this, var5, this.ref.getObjID(), var3);
    this.ref.exportObject(var6);
    this.hashToMethod_Map = (Map)hashToMethod_Maps.get(var4);
    return var5;
}
```

LiveRef.class



```
UnicastServerRef.class LiveRef.class Server.java LocateRegistry.java TCPEndpoint.class TCPTransport.class Transport.class
LiveRef exportObject()

public RMIClientSocketFactory getClientSocketFactory() { return ((TCPEndpoint) this.ep).getClientSocketFactory(); }

public RMIServerSocketFactory getServerSocketFactory() { return ((TCPEndpoint) this.ep).getServerSocketFactory(); }

public void exportObject(Target var1) throws RemoteException {
    this.ep.exportObject(var1);
}
```

TCPEndpoint.class

```

TCPEndpoint.exportObject()
    Iterator var0 = knownTransports().iterator();

    while(var0.hasNext()) {
        TCPTransport var1 = (TCPTransport) var0.next();
        var1.shedConnectionCaches();
    }

}

public void exportObject(Target var1) throws RemoteException {
    this.transport.exportObject(var1);
}

```

TCPTransport.class

```

TCPTransport.exportObject()
    }

}

}

public void exportObject(Target var1) throws RemoteException {
    synchronized(this) {
        this.listen();
        ++this.exportCount;
    }

    boolean var2 = false;
    boolean var12 = false;

    try {
        var12 = true;
        super.exportObject(var1);
    }
}

```

在 TCP 协议层发起 socket 监听, 并采用多线程循环接收请求: `TCPTransport.AcceptLoop(this.server)`



继续通过线程池来处理 socket 接收到的请求

```

private void executeAcceptLoop() {
    if (TCPTransport.tcpLog.isLoggable(Log.BRIEF)) {
        TCPTransport.tcpLog.log(Log.BRIEF, "listening on port " + TCPTransport.this.getEndpoint().getPort());
    }

    while (true) {
        Socket var1 = null;

        try {
            var1 = this.serverSocket.accept();
            InetAddress var16 = var1.getInetAddress();
            String var3 = var16 != null ? var16.getHostAddress() : "0.0.0.0";

            try {
                TCPTransport.connectionThreadPool.execute(TCPTransport.this.new ConnectionHandler(var1, var3));
            } catch (RejectedExecutionException var11) {
                TCPTransport.closeSocket(var1);
                TCPTransport.tcpLog.log(Log.BRIEF, "rejected connection from " + var3);
            }
        } catch (Throwable var15) {
        }
    }
}

UnicastServerRef.class LiveRef.class TCPTransport.class TCPEndpoint.class Server.java LocateRegistry.java
transport ConnectionHandler ConnectionHandler()
    this.authCache.put(var2, new SoftReference(var2));
    this.okContext = var2;
}

}

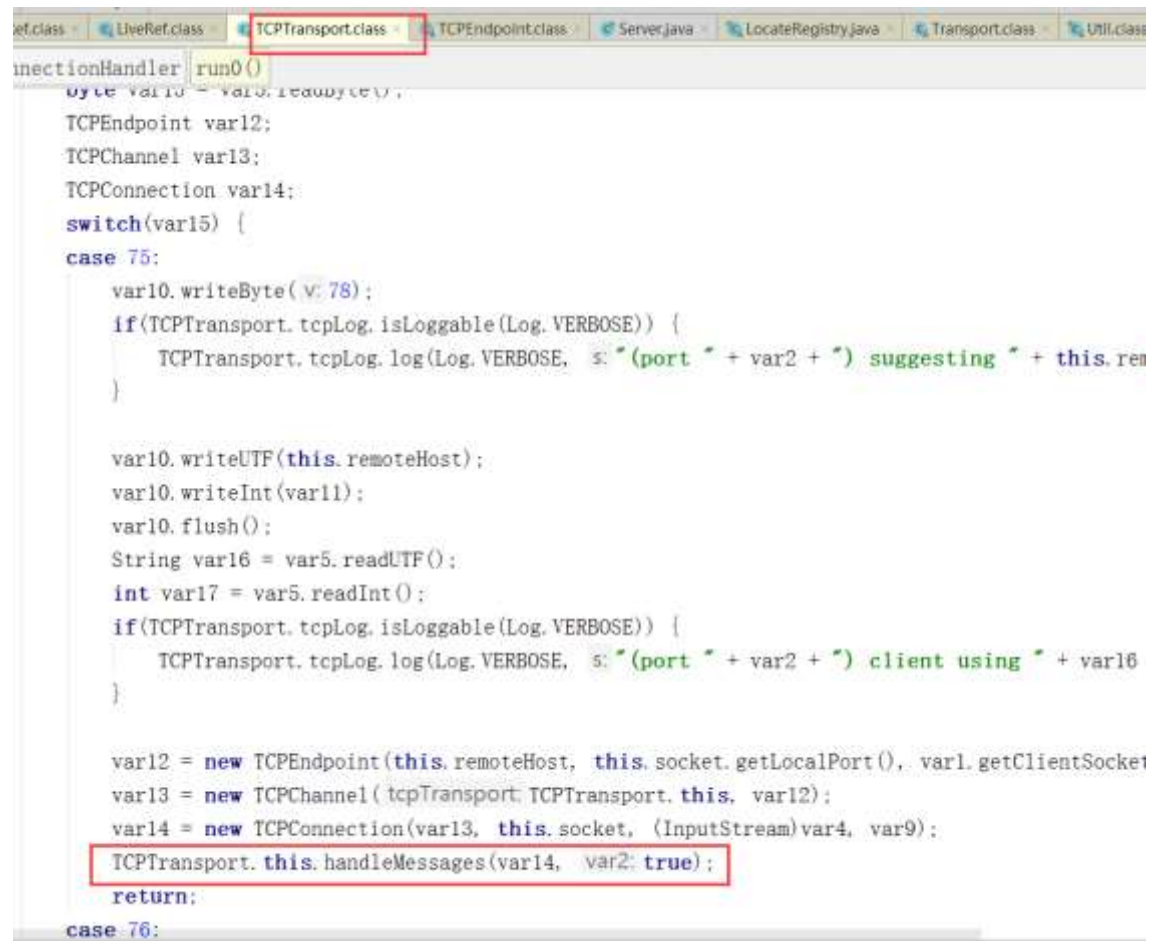
public void run() {
    Thread var1 = Thread.currentThread();
    String var2 = var1.getName();

    try {
        var1.setName("RMI TCP Connection(" + TCPTransport.connectionCount.increase
        AccessController.doPrivileged(() -> {
            this.run0();
            return null;
        }); TCPTransport.NOPERMS_ACC);
    } finally {
        var1.setName(var2);
    }
}
}

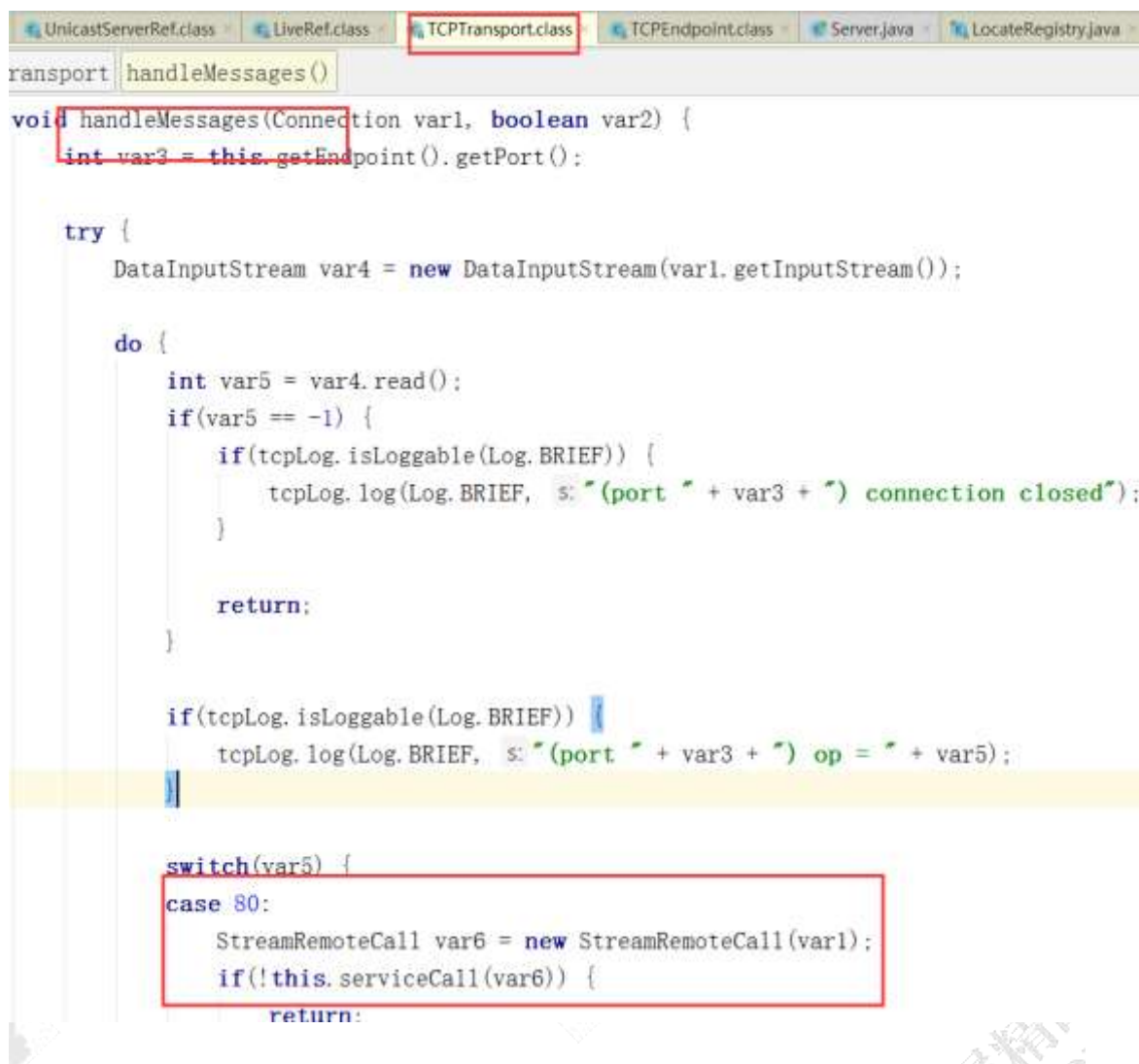
```

下面这个 run0 方法里面做了一些判断，具体的功能是干嘛不太清楚，我猜想是对不同的协议来做处理。我们的这个案例中，会走到如下的代码中来。最终调用

TCPTransport.this.handleMessages(var14, true);



这个地方也做了判断，你们如果不知道怎么走的话，直接在这里加断点就知道。这里会走到 case 80 的段落，执行 serviceCall()这个方法



```
UnicastServerRef.class LiveRef.class TCPTransport.class TCPEndpoint.class Server.java LocateRegistry.java
transport handleMessages()

void handleMessages(Connection var1, boolean var2) {
    int var3 = this.getEndpoint().getPort();

    try {
        DataInputStream var4 = new DataInputStream(var1.getInputStream());

        do {
            int var5 = var4.read();
            if(var5 == -1) {
                if(tcpLog.isLoggable(Log.BRIEF)) {
                    tcpLog.log(Log.BRIEF, s: "(port " + var3 + ") connection closed");
                }

                return;
            }

            if(tcpLog.isLoggable(Log.BRIEF)) {
                tcpLog.log(Log.BRIEF, s: "(port " + var3 + ") op = " + var5);
            }

            switch(var5) {
                case 80:
                    StreamRemoteCall var6 = new StreamRemoteCall(var1);
                    if(!this.serviceCall(var6)) {
                        return;
                    }
            }
        } while(true);
    } catch (IOException var7) {
        tcpLog.log(Log.ERROR, s: "IOException: " + var7.getMessage());
    }
}
```

一步一步我们找到了 Transport 的 serviceCall()方法, 这个方法是关键。瞻仰一下主要的代码, 到 ObjectTable.getTarget()为止做的事情是从 socket 流中获取 ObjId, 并通过 ObjId 和 Transport 对象获取 Target 对象, 这里的 Target 对象已经是服务端的对象。再借由 Target 的派发器 Dispatcher, 传入参数服务实现和请求对象 RemoteCall, 将请求派发给服务端那个真正提供服务的 RegistryImpl 的 lookUp()方法, 这就是 Skeleton 移交给具

体实现的过程了，Skeleton 负责底层的操作。

```
UnicastServerRef.class Transport.class LiveRef.class TCPTransport.class TCPEndpoint.class Server.java
nsport serviceCall()

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; 此处的Target为服务端对象
        Target var5 = ObjectTable.getTarget(new ObjectEndpoint(var39, var40));
        final Remote var37;
        if(var5 != null && (var37 = var5.getImpl()) != null) {
            final Dispatcher var6 = var5.getDispatcher();
            var5.incrementCallCount();

            boolean var8;
            try {
                transportLog.log(Log.VERBOSE, s: "call dispatcher");
                final AccessControlContext var7 = var5.getAccessControlContext();
                ClassLoader var41 = var5.getContextClassLoader();
                ClassLoader var9 = Thread.currentThread().getContextClassLoader();
```

```

nicastServerRef.class x Transport.class x LiveRef.class x TCPTransport.class x TCPEndpoint.class x Server.java x
t serviceCall()

ClassLoader var41 = var5.getContextClassLoader();
ClassLoader var9 = Thread.currentThread().getContextClassLoader();

try {
    setContextClassLoader(var41);
    currentTransport.set(this);

    try {
        AccessController.doPrivileged(run() -> {
            Transport.this.checkAcceptPermission(var7);
            var6.dispatch(var37, var1);
            return null;
        }, var7);
        return true;
    } catch (PrivilegedActionException var31) {
        throw (IOException) var31.getException();
    }
} finally {
    setContextClassLoader(var9);
    currentTransport.set((Object) null);
}

```

所以客户端通过

```

IHelloService helloService=
    (IHelloService) Naming.lookup( name: "rmi://127.0.0.1/Hello");

```

先会创建一个 RegistryImpl_Stub 的代理类，通过这个代理类进行 socket 网络请求，将 lookup 发送到服务端，服务端通过接收到请求以后，通过服务端的 RegistryImpl_Stub (Skeleton)，执行 RegistryImpl 的 lookUp。而服务端的 RegistryImpl 返回的就是服务端的 HelloServiceImpl 的实现类

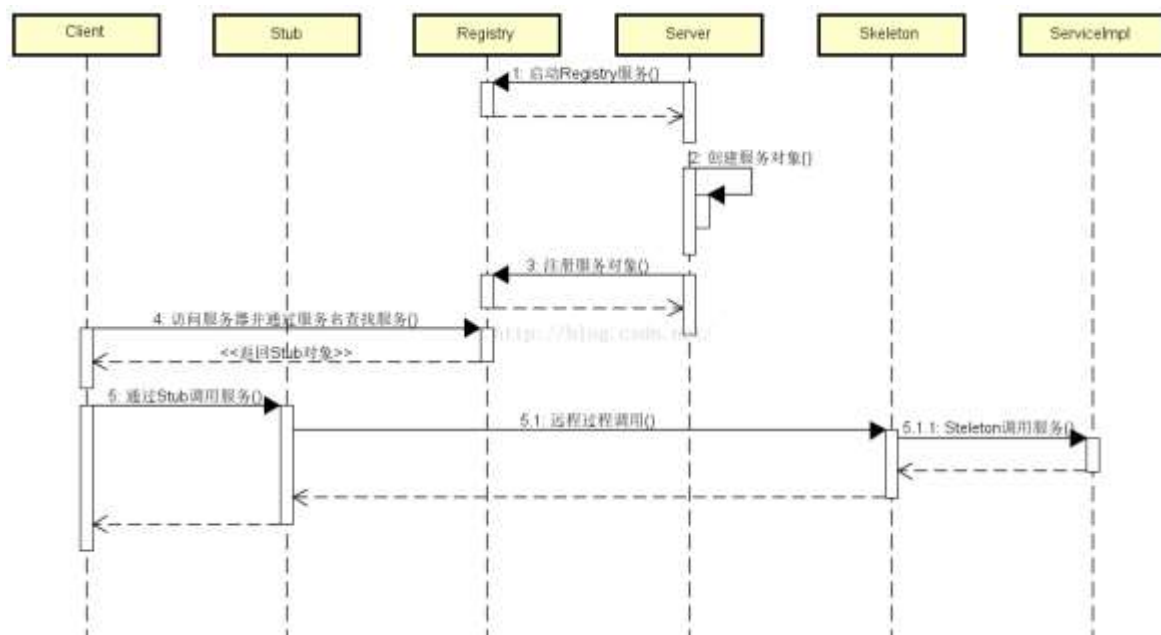
```
public Remote lookup(String var1) throws RemoteException, NotBoundException {
    Hashtable var2 = this.bindings;
    synchronized(this.bindings) {
        Remote var3 = (Remote) this.bindings.get(var1);
        if(var3 == null) {
            throw new NotBoundException(var1);
        } else {
            return var3;
        }
    }
}
```

客户端获取通过 lookup() 查询获得的客户端 HelloServiceImpl 的 Stub 对象

客户端通过 Lookup 查询获得的是客户端 HelloServiceImpl 的 Stub 对象（这一块我们看不到，因为这块由 Skeleton 为我们屏蔽了），然后后续的处理仍然是通过 HelloServiceImpl_Stub 代理对象通过 socket 网络请求到服务端，通过服务端的 HelloServiceImpl_Stub(Skeleton) 进行代理，将请求通过 Dispatcher 转发到对应的服务端方法获得结果以后再次通过 socket 把结果返回到客户端；

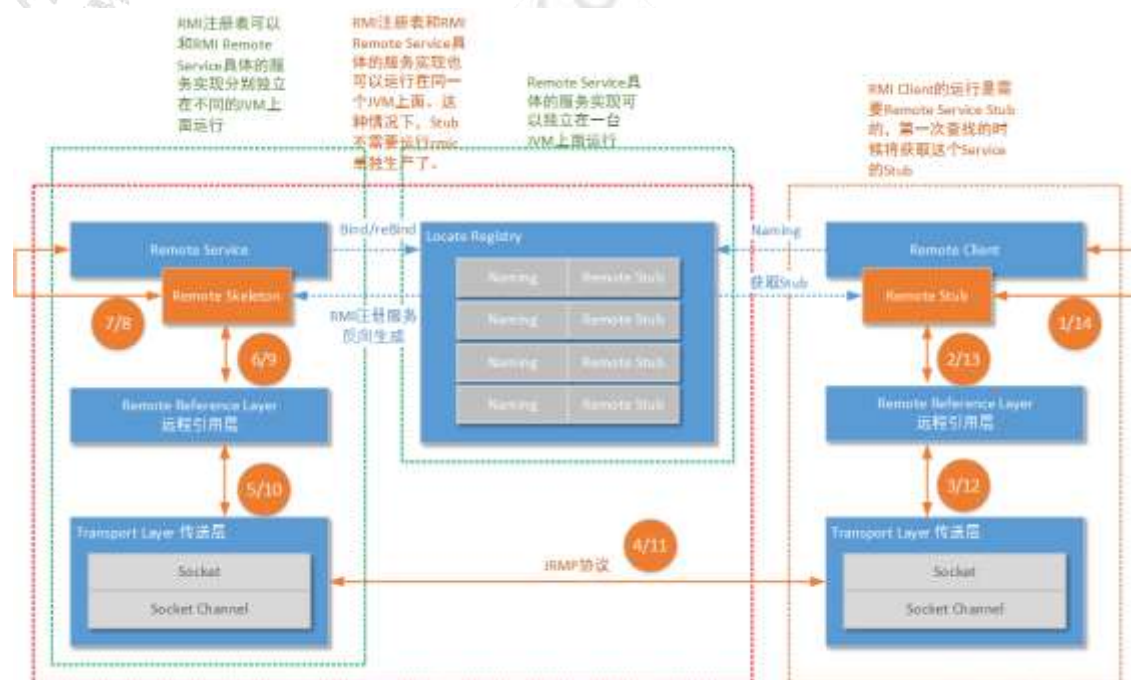
RMI 做了什么

根据上面的源码阅读，实际上我们看到的应该是有两个代理类，一个是 RegistryImpl 的代理类和我们 HelloServiceImpl 的代理类。



通信原理

我们大概知道了 RMI 框架是如何使用的。下面我们来讲解一下 RMI 的基本原理



(上图摘自于网络)

一定要说明，在 RMI Client 实施正式的 RMI 调用前，它必须通过 LocateRegistry 或者 Naming 方式到 RMI 注册表寻找要调用的 RMI 注册信息。找到 RMI 事务注册信息后，Client 会从 RMI 注册表获取这个 RMI Remote Service 的 Stub 信息。这个过程成功后，RMI Client 才能开始正式的调用过程。

另外要说明的是 RMI Client 正式调用过程，也不是由 RMI Client 直接访问 Remote Service，而是由客户端获取的 Stub 作为 RMI Client 的代理访问 Remote Service 的代理 Skeleton，如上图所示的顺序。也就是说真实的请求调用是在 Stub-Skeleton 之间进行的。

Registry 并不参与具体的 Stub-Skeleton 的调用过程，只负责记录“哪个服务名”使用哪一个 Stub，并在 Remote Client 询问它时将这个 Stub 拿给 Client（如果没有就会报错）。

自己实现 RPC 框架

参考 gitlab 上的代码

