1. 架构

RMI 中有三个重要的角色：注册中心（Registry）、客户端（Client）、服务端（Server）。

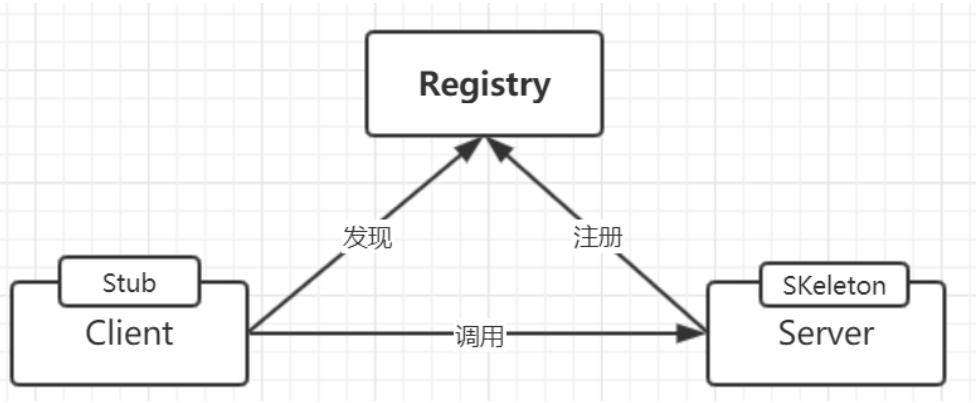

图 1 RMI 架构图

在 RMI 中也要先进行服务注册，客户端从注册中心获取服务。为了屏蔽网络通信的复杂性，RMI 提出了 Stub（客户端存根）和 Skeleton（服务端骨架）两个概念，客户端和服务端的网络信都通过 Stub 和 Skeleton 进行。

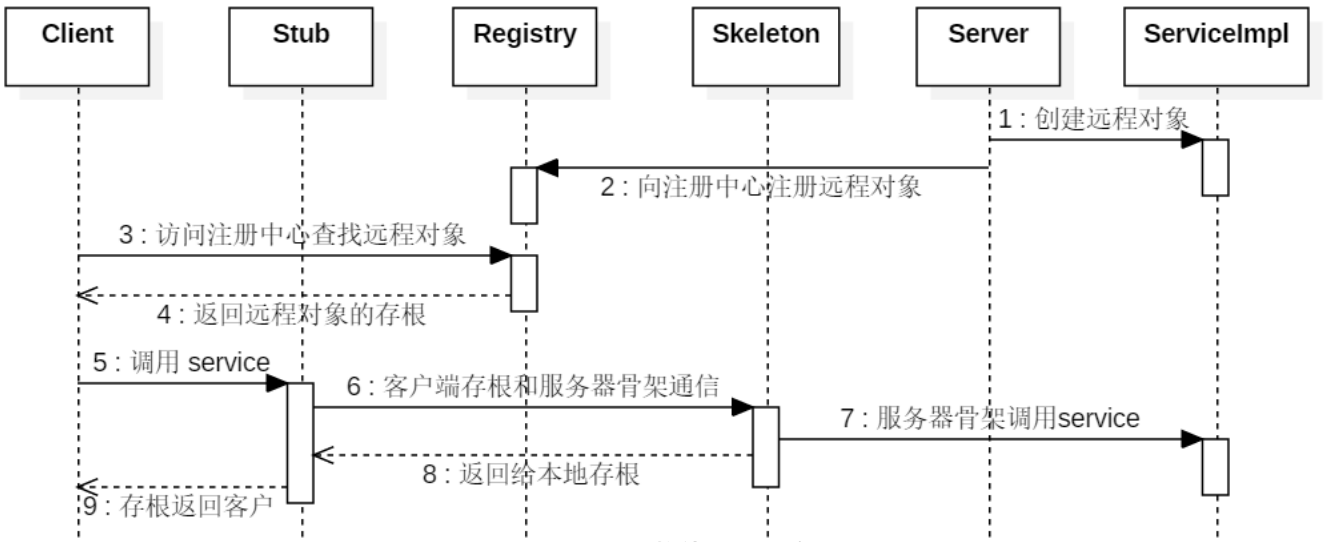

图 2 RMI 整体调用时序图

总结： 整体还是可以分为三部分，服务注册、服务发现、服务调用。

1. 服务注册（1 ~ 2）
   - 第一步：创建远程对象包括两部分。一是创建 ServiceImpl 远程对象；二是发布 ServiceImpl 服务。ServiceImpl 继承自 UnicastRemoteObject，在创建时默认会随机绑定一个端口，监听客户端的请求。所以即使可以不注册，直接请求这个端口也可以进行通信。
   - 第二步：向注册中心注册该服务。注意：和其它的注册中心不同，Registry 只能注册本地的服务。
2. 服务发现（3 ~ 4）
   - 向注册中心查找本地存根，返回给客户端。需要注意的是，Dubbo 先从注册中心获服务的 ip、port 等配置信息，然后在客户端生成 Stub 代理，而 RMI 不一样，已经在服务端保存了 Stub 代理对象，直接通过网络传输直接将 Stub 对象进行序列化与反序列化。
3. 服务调用（5 ~ 9）
   - 客户端存根和服务器骨架通信，返回结果。

2. 服务注册

首先回顾一下 RMI 服务发布的使用方法：

```java
@Test
public void server() {
        // 1. 服务创建及发布。注意：HelloServiceImpl extends UnicastRemoteObject
    HelloService service = new HelloServiceImpl();
    // 2. 创建注册中心：创建本机 1099 端口上的 RMI 注册表
    Registry registry = LocateRegistry.createRegistry(1099);
    // 3. 服务注册：将服务绑定到注册表中
    registry.bind(name, service);
}
```

总结： RMI 服务发布有三个流程：

1. 服务创建及发布：HelloServiceImpl 需要继承自 UnicastRemoteObject，当初始化时会自动将 HelloServiceImpl 任务一个服务发布，绑定一个随机端口。
2. 创建注册中心：注册中心实际和普通的服务一样，也会将自己作为一个服务发布。
3. 服务注册：将 service 注册到注册中心。

服务创建及发布和创建注册中心流程完全相同，至于服务注册则是将 service 注册到一个 map 中，非常简单。所以服务的注册主要围绕服务创建及发布展开。

2.1 服务发布整体流程

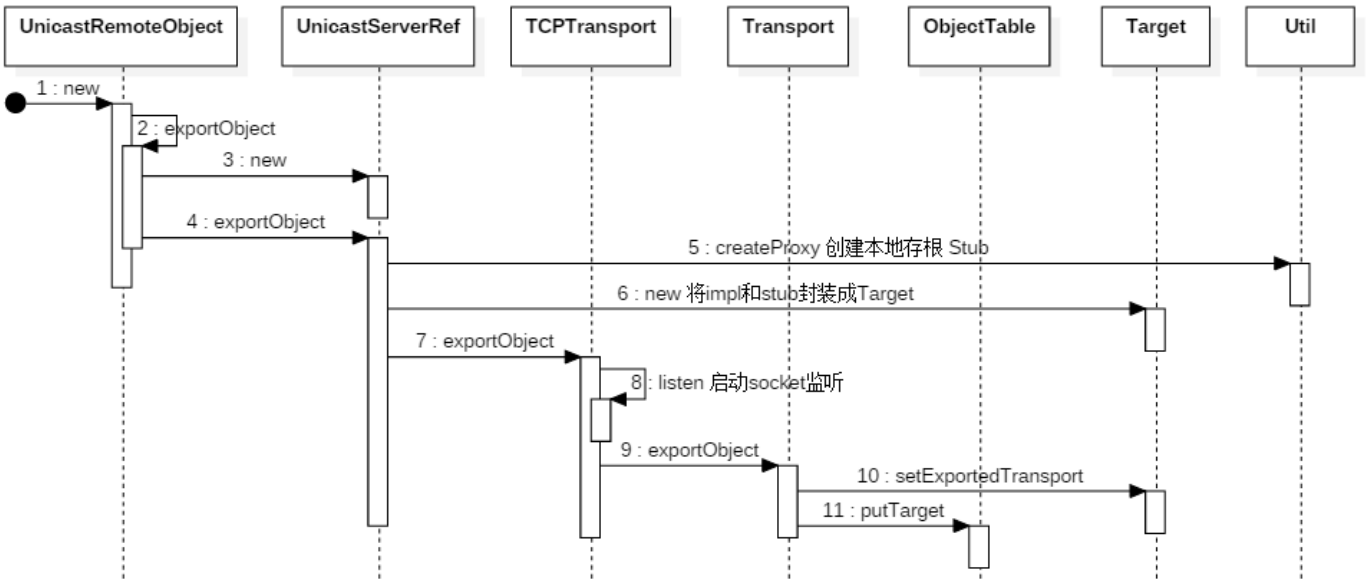

图 3 RMI 服务发布时序图

服务的发布的关键点有以下几个：

1. 创建本地存根，用于客户端访问。
2. 启动 socket。
3. 服务注册与查找。

无论是 HelloServiceImpl 还是 Registry 都是 Remote 的子类，准确的说是 RemoteObject 的子类。RemoteObject 最重要的属性是 `RemoteRef ref`，RemoteRef 的实现类 UnicastRef，UnicastRef 包含属性 `LiveRef ref`。LiveRef 类中的 Endpoint、Channel 封装了与网络通信相关的方法。类结构如下：

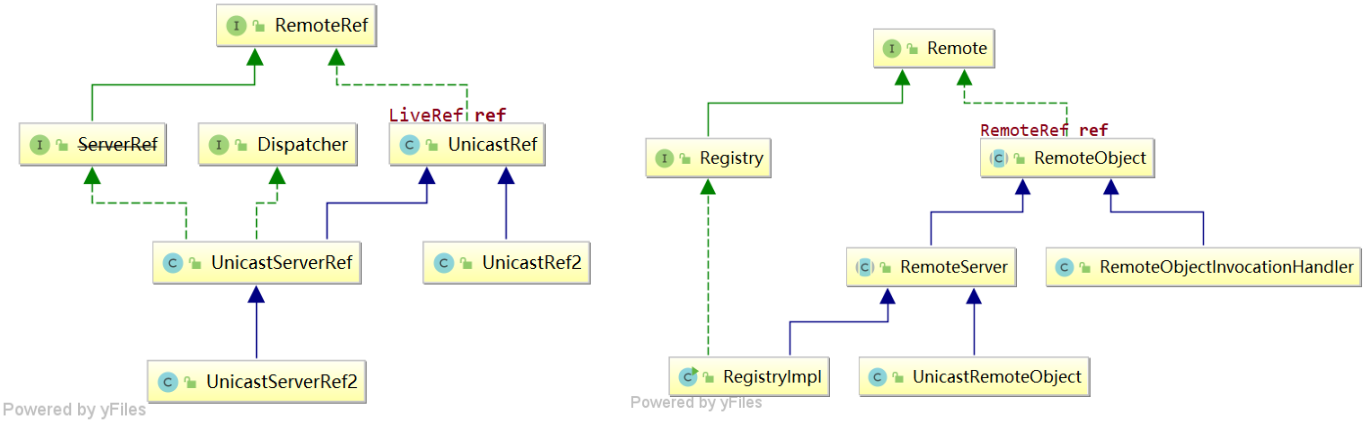

图 4 Remote 和 RemoteRef 类结构

2.2 服务暴露入口 exportObject

HelloServiceImpl 的构造器中调用父类 UnicastRemoteObject，最终调用 `exportObject((Remote) this, port)`

```
protected UnicastRemoteObject(int port) throws RemoteException {
    this.port = port;
    exportObject((Remote) this, port);
}
private static Remote exportObject(Remote obj, UnicastServerRef sref)
    throws RemoteException {
    if (obj instanceof UnicastRemoteObject) {
        ((UnicastRemoteObject) obj).ref = sref;
    }
    return sref.exportObject(obj, null, false);
}
```

而 `Registry createRegistry(int port)` 创建注册中心时也会调用 exportObject 方法。

```
public RegistryImpl(int port) throws RemoteException
    LiveRef lref = new LiveRef(id, port);
        setup(new UnicastServerRef(lref, RegistryImpl::registryFilter));
}
private void setup(UnicastServerRef uref) throws RemoteException {
    ref = uref;
    uref.exportObject(this, null, true);
}
```

总结： Registry 和 HelloServiceImpl 最终都调用 exportObject 方法，那 exportObject 到底是干什么的呢？从字面上看 exportObject 暴露对象，事实上正如其名，exportObject 打开了一个 ServerSocket，监听客户端的请求。

```
public Remote exportObject(Remote impl, Object data, boolean permanent)
        throws RemoteException {
    // 1. 创建本地存根，封装网络通信
    Class<?> implClass = impl.getClass();
    Remote stub = Util.createProxy(implClass, getClientRef(), forceStubUse);
    ...
    // 2. 服务暴露，this 是指 RemoteObject 对象
    Target target = new Target(impl, this, stub, ref.getObjID(), permanent);
    ref.exportObject(target);
    return stub;
}
```

总结： exportObject 核心的方法有两个：一是生成本地存根的代理对象；二是调用 `ref.exportObject(target)` 启动 socket 服务。

注意：exportObject 时会先将 impl 和 stub 等信息封装到 Target 对象中，最终注册到 ObjectTable。

2.3 生成本地存根

在 Util.createProxy() 方法中创建代理对象。

```
public static Remote createProxy(Class<?> implClass, RemoteRef clientRef,
                 boolean forceStubUse) throws StubNotFoundException {
    Class<?> remoteClass = getRemoteClass(implClass);

    // 1. 是否存在以 Stub 结尾的类。remoteClass + " Stub"
    //    forceStubUse 表示当不存在时是否抛出异常
    if (forceStubUse ||
        !(ignoreStubClasses || !stubClassExists(remoteClass))) {
        return createStub(remoteClass, clientRef);
    }

    // 2. jdk 动态代理
    final ClassLoader loader = implClass.getClassLoader();
    final Class<?>[] interfaces = getRemoteInterfaces(implClass);
    final InvocationHandler handler = new RemoteObjectInvocationHandler(clientRef);
    return (Remote) Proxy.newProxyInstance(loader, interfaces, handler);
}
```

总结： 创建代理对象有两种情况：

1. 存在以 _Stub 结尾的类（eg: RegistryImpl_Stub）则直接返回，当 forceStubUse=true 时不存在则抛出异常。
2. JDK 动态代理。RemoteObjectInvocationHandler#invoke 方法实际上直接委托给了 RemoteRef#invoke 方法进行网络通信，具体代码见 `UnicastRef#invoke(Remote, Method, Object[], long)`。

2.4 服务监听

跟踪 LiveRef#exportObject 方法，最终调用 TCPTransport#exportObject 方法。

```
public void exportObject(Target target) throws RemoteException {
    // 1. 启动网络监听，默认 port=0，即随机启动一个端口
    synchronized (this) {
        listen();
        exportCount++;
    }
    // 2. 将 Target 注册到 ObjectTable
    super.exportObject(target);
}
```

总结： 最终服务暴露时做了两件事，一是如果 socket 没有启动，启动 socket 监听；二是将 Target 实例注册到 ObjectTable 对象中。

```
private void listen() throws RemoteException {
     TCPEndpoint ep = getEndpoint();
     int port = ep.getPort();

     if (server == null) {
            server = ep.newServerSocket();
            Thread t = new NewThreadAction(new AcceptLoop(server),
                        "TCP Accept-" + port, true));
            t.start();
        } catch (IOException e) {
            throw new ExportException("Listen failed on port: " + port, e);
        }
     }
 }
```

2.5 ObjectTable 注册与查找

ObjectTable 用来管理所有发布的服务实例 Target，ObjectTable 提供了根据 ObjectEndpoint 和 Remote 实例两种方式查找 Target 的方法。先看注册：

```
private static final Map<ObjectEndpoint,Target> objTable = new HashMap<>();
private static final Map<WeakRef,Target> implTable = new HashMap<>();

// Target 注册
static void putTarget(Target target) throws ExportException {
    ObjectEndpoint oe = target.getObjectEndpoint();
    WeakRef weakImpl = target.getWeakImpl();

    synchronized (tableLock) {
        if (target.getImpl() != null) {
            ...
            objTable.put(oe, target);
            implTable.put(weakImpl, target);
        }
    }
}
```

那实例查找也就很简单了，之后就可以根据 Target 对象获取本地存根 stub。

```
static Target getTarget(ObjectEndpoint oe) {
    synchronized (tableLock) {
        return objTable.get(oe);
    }
}
public static Target getTarget(Remote impl) {
    synchronized (tableLock) {
        return implTable.get(new WeakRef(impl));
    }
}
```

2.6 服务绑定

当服务 HelloService 和 Registry 均已创建并发布后，之后需要将服务绑定到注册中心。这一步就很简单了，代码 `registry.bind(name, service)`。

```
// 服务名称 -> 实例 impl
private Hashtable<String, Remote> bindings = new Hashtable<>(101);

public void bind(String name, Remote obj)
    throws RemoteException, AlreadyBoundException, AccessException {
    checkAccess("Registry.bind");
    synchronized (bindings) {
        Remote curr = bindings.get(name);
        if (curr != null)
            throw new AlreadyBoundException(name);
        bindings.put(name, obj);
    }
}
```

总结： service 绑定到注册中心实际就很简单了，将服务名称和实例保存到 map 中即可。查找时可以通过 name 查找到 impl，再通过 impl 在 ObjectTable 中查找到对应的 Target。

2.7 总结

服务暴露主要完成两件事：一是服务端生成本地存根 stub，并包装成 Target 对象，最终注册到 ObjectTable 中；二是启动 ServerSocket 绑定端口，监听客户端的请求。 又可以分为普通服务暴露和注册中心暴露，两种服务暴露过程完全相同。

1. 普通服务暴露（HelloService）：默认绑定随机端口。使用 HelloServicempl 实例，根据动态代理生成本地存储 stub，RemoteObjectInvocationHandler#invoke 最终调用 `UnicastRef#invoke(Remote, Method, Object[], long)` 方法。

2. 注册中心暴露（Registry）：LocateRegistry.createRegistry(port) 需要指定绑定端口，默认 1099。使用 RegistryImpl 实例，本地存根使用 RegistryImpl_Stub。

3. 服务发现

```
@Test
public void client() {
    String name = HelloService.class.getName();
    // 获取注册表
    Registry registry = LocateRegistry.getRegistry("localhost", 1099);
    // 查找对应的服务
    HelloService service = (HelloService) registry.lookup(name);
}
```

总结： RMI 服务发现核心步骤两步：一是获取注册中心 registry；二是根据注册中心获取服务的代理类 service。registry 和 service 都是通过 Util.createProxy() 方法生成的代理类，不过这两个代理类的生成时机完全不同，registry 是在客户端生成的代理类，service 是在服务端生成的代理类。

3.1 注册中心 Stub

```
public static Registry getRegistry(String host, int port, RMIClientSocketFactory csf)
{
    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);
}
```

由于默认存在 RegistryImpl_Stub，所以直接返回 RegistryImpl_Stub 的实例。

```
public Remote lookup(String var1) throws AccessException, NotBoundException,
RemoteException {
    RemoteCall var2 = super.ref.newCall(this, operations, 2, 4905912898345647071L);
    ObjectOutput var3 = var2.getOutputStream();
    var3.writeObject(var1);

    super.ref.invoke(var2);
    ObjectInput var6 = var2.getInputStream();
    Remote var23 = (Remote)var6.readObject();
    super.ref.done(var2);
    return var23;
}
```

总结： LocateRegistry.getRegistry 获取注册中心时，在客户端直接生成代理对象 RegistryImpl_Stub，RegistryImpl_Stub 实际调用 RemoteRef 的 invoke 方法进行网络通信。

3.2 普通服务 Stub

和 RegistryImpl_Stub 不同，普通服务是在服务端生成本地存根 Stub。在服务注册的阶段，我们提到服务暴露时会将服务实例及其生成的 Stub 包装成 Target，并最终注册到 ObjectTable 上。那客户端 registry.lookup(name) 是如何最终查找到对应服务的 Stub 中的呢?

首先客户端调用 registry.lookup(name) 时，会通过网络通信最终调用到 RegistryImpl#lookup 方法，查找到对应的 Remote 实例，之后将这个实例返回给客户端。

但是这个 Socket 输出流是被 MarshalOutputStream 包装过的，在输出对应时会将 Remote 替换为 Stub 对象。也就是说客户端直接可以拿到代理后的对象，反序列后进行网络通信，不需要在客户端生成代理对象。代码如下：

```
protected final Object replaceObject(Object obj) throws IOException {
    if ((obj instanceof Remote) && !(obj instanceof RemoteStub)) {
        Target target = ObjectTable.getTarget((Remote) obj);
        if (target != null) {
            return target.getStub();
        }
    }
    return obj;
}
```

总结： registry.lookup(name) 获取服务端生成的代理对象 stub。这个 stub 代理对象调用 `UnicastRef#invoke(Remote, Method, Object[], long)` 方法进行网络通信。

注意： 如果该服务没有暴露，则 target=null，也就是直接将服务端注册的实例而不是存根 Stub 返回，所以在客户端必须有该类的实现，否则反序列反时会抛出异常。不过，不暴露服务这种情况好像并没有什么意义。

4. 服务调用

RMI 中网络通信相关的逻辑都是由 RemoteRef 完成的，客户端的实现是 UnicastRef，而服务端则是 UnicastServerRef。

4.1 客户端调用过程

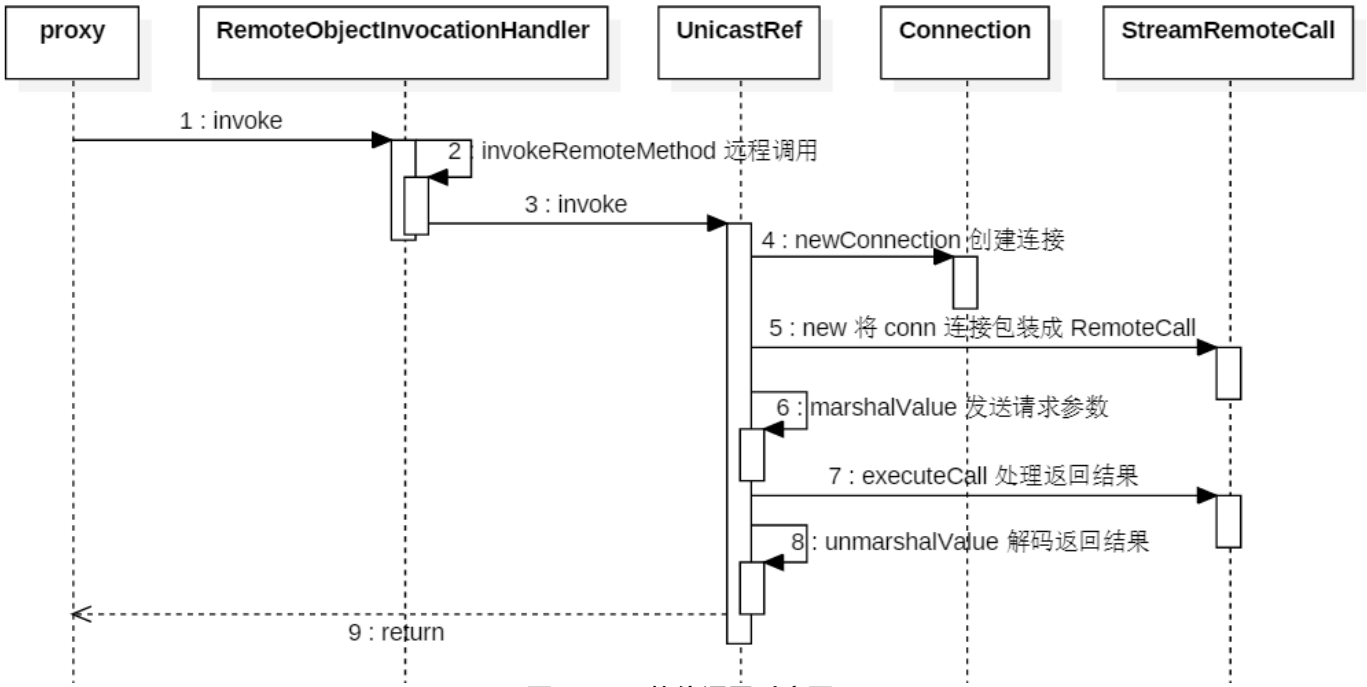

图 5 RMI 整体调用时序图

客户端生成的代理对象调用 UnicastRef.invoke 进行网络传输，至少要告诉服务端以下信息：

1. 接口名
2. 方法名称
3. 参数类型
4. 参数

通过 接口 + 方法名称 + 参数类型 三个坐标就可以确定调用的具体方法。RMI 中通过传递 opnum 参数标记是这个类的第几个方法来确定。

```
public Object invoke(Remote obj, Method method, Object[] params, long opnum)
    throws Exception {

    // 1. create call context
    Connection conn = ref.getChannel().newConnection();
    RemoteCall call = new StreamRemoteCall(conn, ref.getObjID(), -1, opnum);

    // 2. marshal parameters
    try {
        ObjectOutput out = call.getOutputStream();
        marshalCustomCallData(out);
        Class<?>[] types = method.getParameterTypes();
        for (int i = 0; i < types.length; i++) {
            marshalValue(types[i], params[i], out);
        }
    } catch (IOException e) {
        throw new MarshalException("error marshalling arguments", e);
    }

    // 3. unmarshal return
    call.executeCall();

    try {
        Class<?> rtype = method.getReturnType();
        if (rtype == void.class)
            return null;
        ObjectInput in = call.getInputStream();

        Object returnValue = unmarshalValue(rtype, in);

        ref.getChannel().free(conn, true);
        return returnValue;
    } finally {
         // 关闭连接等
        call.done();
    }
}
```

总结： UnicastRef.invoke 方法过程很清晰，先发送 opnum，再发送参数，最后处理返回结果。

4.2 服务端处理过程

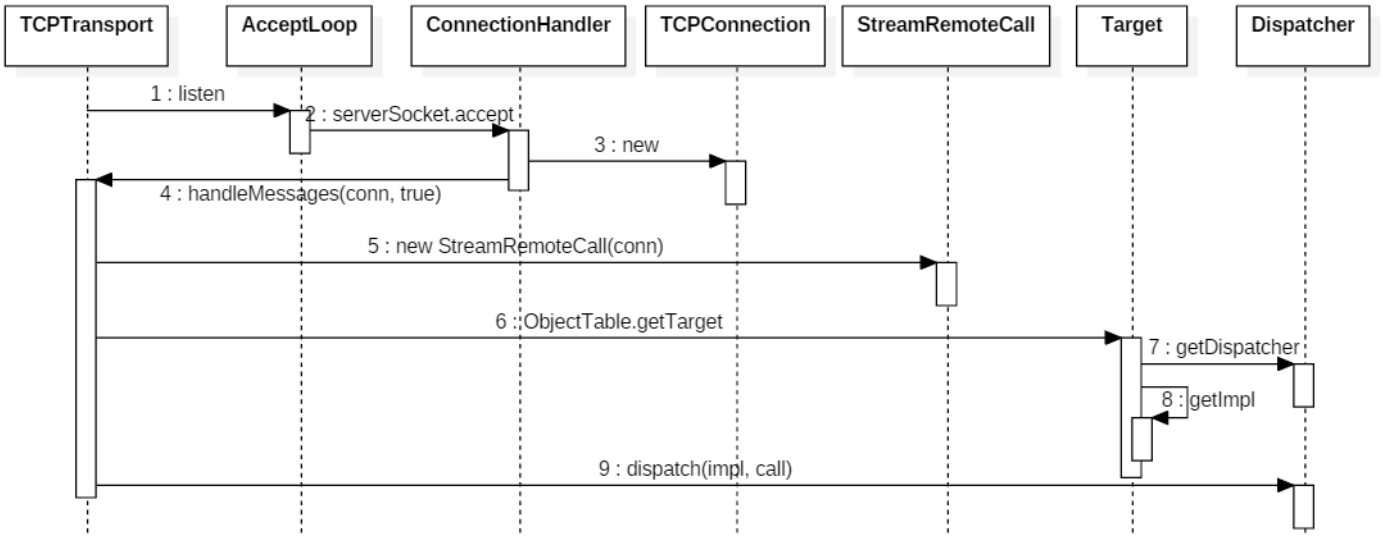

图 6 RMI 服务端调用时序图

总结：

1. 服务暴露时，通过 listen 方法启动 socket，创建 AcceptLoop 线程接收客户端的连接，每一个 socket 连接创建一个 ConnectionHandler 处理，这也是 BIO 处理客户端连接的基本套路。

2. 然后从 ObjectTarget.getTarget 中获取服务端保存的 Target 对象，可以获取 impl 和 dispatcher 对象。dispatcher 的实现类是 UnicastServerRef。

3. 服务端获取 opnum 确定具体的方法 method，再接收方法的参数，调用 method.invoke(obj, params) 后将结果返回给客户端。

下面就看一下 UnicastServerRef#dispatch 具体做了些什么。

```
public void dispatch(Remote obj, RemoteCall call) throws IOException {
    // positive operation number in 1.1 stubs;
    // negative version number in 1.2 stubs and beyond...
    int num;
    long op;

    // 1. 处理 num 和 op 参数，jdk1.1 oldDispatch
     try {
        ObjectInput in;
        try {
            in = call.getInputStream();
            num = in.readInt();
            if (num >= 0) {
                if (skel != null) {
                    oldDispatch(obj, call, num);
                    return;
                } else {
                    throw new UnmarshalException(
                        "skeleton class not found but required for client version");
                }
            }
            op = in.readLong();
        } catch (Exception readEx) {
            throw new UnmarshalException("error unmarshalling call header", readEx);
        }

        MarshalInputStream marshalStream = (MarshalInputStream) in;
        marshalStream.skipDefaultResolveClass();

        // 2. 根据 op 获取 method
        Method method = hashToMethod_Map.get(op);
```

```
        // 3. unmarshal parameters
        Object[] params = null;
        try {
            unmarshalCustomCallData(in);
            params = unmarshalParameters(obj, method, marshalStream);
        } finally {
            call.releaseInputStream();
        }

        // 4. make upcall on remote object
        Object result;
        try {
            result = method.invoke(obj, params);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }

        // 5. marshal return value
        try {
            ObjectOutput out = call.getResultStream(true);
            Class<?> rtype = method.getReturnType();
            if (rtype != void.class) {
                marshalValue(rtype, result, out);
            }
        } catch (IOException ex) {
            throw new MarshalException("error marshalling return", ex);
        }
    } catch (Throwable e) {
        ObjectOutput out = call.getResultStream(false);
        out.writeObject(e);
    } finally {
        call.releaseInputStream(); // in case skeleton doesn't
        call.releaseOutputStream();
    }
}
```

总结： UnicastServerRef#dispatch 方法也很清晰，无非是根据 op 确定具体的方法 method，获取参数，将反射的结果通过网络返回给客户端。