# 使用 RMI + ZooKeeper 实现远程调用框架

目录[-]

 [1 发布 RMI 服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h1_1)

 [1.1 定义一个 RMI 接口](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_2)

 [1.2 编写 RMI 接口的实现类](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_3)

 [1.3 通过 JNDI 发布 RMI 服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_4)

 [2 调用 RMI 服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h1_5)

 [3 RMI 服务的局限性](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h1_6)

 [4 使用 ZooKeeper 提供高可用的 RMI 服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h1_7)

 [4.1 服务提供者](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_8)

 [4.2 服务消费者](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_9)

 [4.3 发布服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_10)

 [4.4 调用服务](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_11)

 [4.5 使用方法](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h2_12)

 [5 总结](http://my.oschina.net/huangyong/blog/345164?fromerr=D9hzs219#OSC_h1_13)

在 Java 世界里，有一种技术可以实现“跨虚拟机”的调用，它就是 RMI（Remote Method Invocation，远程方法调用）。例如，服务A 在 JVM1 中运行，服务B 在 JVM2 中运行，服务A 与 服务B 可相互进行远程调用，就像调用本地方法一样，这就是 RMI。在分布式系统中，我们使用 RMI 技术可轻松将 服务提供者（Service Provider）与 服务消费者（Service Consumer）进行分离，充分体现组件之间的弱耦合，系统架构更易于扩展。

本文先从通过一个最简单的 RMI 服务与调用示例，让读者快速掌握 RMI 的使用方法，然后指出 RMI 的局限性，最后笔者对此问题提供了一种简单的解决方案，即使用 ZooKeeper 轻松解决 RMI 调用过程中所涉及的问题。

下面我们就从一个最简单的 RMI 示例开始吧！

# 1 发布 RMI 服务

发布一个 RMI 服务，我们只需做三件事情：

1. 定义一个 RMI 接口
2. 编写 RMI 接口的实现类
3. 通过 JNDI 发布 RMI 服务

## 1.1 定义一个 RMI 接口

RMI 接口实际上还是一个普通的 Java 接口，只是 RMI 接口必须继承 java.rmi.Remote，此外，每个 RMI 接口的方法必须声明抛出一个 java.rmi.RemoteException 异常，就像下面这样：

**package** demo.zookeeper.rmi.common;

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**public** **interface** HelloService **extends** Remote {

String sayHello(String name) **throws** RemoteException;

}

继承了 Remote 接口，实际上是让 JVM 得知该接口是需要用于远程调用的，抛出了 RemoteException 是为了让调用 RMI 服务的程序捕获这个异常。毕竟远程调用过程中，什么奇怪的事情都会发生（比如：断网）。需要说明的是，RemoteException 是一个“受检异常”，在调用的时候必须使用 try...catch... 自行处理。

## 1.2 编写 RMI 接口的实现类

实现以上的 HelloService 是一件非常简单的事情，但需要注意的是，我们必须让实现类继承 java.rmi.server.UnicastRemoteObject 类，此外，必须提供一个构造器，并且构造器必须抛出 java.rmi.RemoteException 异常。我们既然使用 JVM 提供的这套 RMI 框架，那么就必须按照这个要求来实现，否则是无法成功发布 RMI 服务的，一句话：我们得按规矩出牌！

**package** demo.zookeeper.rmi.server;

**import** demo.zookeeper.rmi.common.**HelloService**;

**import** java.rmi.**RemoteException**;

**import** java.rmi.server.**UnicastRemoteObject**;

public **class** HelloServiceImpl **extends** UnicastRemoteObject implements HelloService {

**protected** **HelloServiceImpl**() **throws** **RemoteException** {

}

@Override

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

**return** **String**.format(**"Hello %s"**, name);

}

}

为了满足 RMI 框架的要求，我们确实做了很多额外的工作（继承了 UnicastRemoteObject 类，抛出了 RemoteException 异常），但这些工作阻止不了我们发布 RMI 服务的决心！我们可以通过 JVM 提供的 JNDI（Java Naming and Directory Interface，Java 命名与目录接口）这个 API 轻松发布 RMI 服务。

## 1.3 通过 JNDI 发布 RMI 服务

发布 RMI 服务，我们需要告诉 JNDI 三个基本信息：1. 域名或 IP 地址（host）、2. 端口号（port）、3. 服务名（service），它们构成了 RMI 协议的 URL（或称为“RMI 地址”）：

rmi://<host>:<port>/<service>

如果我们是在本地发布 RMI 服务，那么 host 就是“localhost”。此外，RMI 默认的 port 是“1099”，我们也可以自行设置 port 的值（只要不与其它端口冲突即可）。service 实际上是一个基于同一 host 与 port 下唯一的服务名，我们不妨使用 Java 完全类名来表示吧，这样也比较容易保证 RMI 地址的唯一性。

对于我们的示例而言，RMI 地址为：

**rmi:***//localhost:1099/demo.zookeeper.rmi.server.HelloServiceImpl*

我们只需简单提供一个 main() 方法就能发布 RMI 服务，就像下面这样：

**package** demo.zookeeper.rmi.server;

**import** java.rmi.Naming;

**import** java.rmi.registry.LocateRegistry;

**public** **class** RmiServer {

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

**int** port = 1099;

String url = **"rmi://localhost:1099/demo.zookeeper.rmi.server.HelloServiceImpl"**;

LocateRegistry.createRegistry(port);

Naming.rebind(url, **new** HelloServiceImpl());

}

}

需要注意的是，我们通过 LocateRegistry.createRegistry() 方法在 JNDI 中创建一个注册表，只需提供一个 RMI 端口号即可。此外，通过 Naming.rebind() 方法绑定 RMI 地址与 RMI 服务实现类，这里使用了 rebind() 方法，它相当于先后调用 Naming 的 unbind() 与 bind() 方法，只是使用 rebind() 方法来得更加痛快而已，所以我们选择了它。

运行这个 main() 方法，RMI 服务就会自动发布，剩下要做的就是写一个 RMI 客户端来调用已发布的 RMI 服务。

# 2 调用 RMI 服务

同样我们也使用一个 main() 方法来调用 RMI 服务，相比发布而言，调用会更加简单，我们只需要知道两个东西：1. RMI 请求路径、2. RMI 接口（一定不需要 RMI 实现类，否则就是本地调用了）。数行代码就能调用刚才发布的 RMI 服务，就像下面这样：

package demo.zookeeper.rmi.client;

**import** demo.zookeeper.rmi.common.**HelloService**;

**import** java.rmi.**Naming**;

public class **RmiClient** {

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

**String** url = **"rmi://localhost:1099/demo.zookeeper.rmi.server.HelloServiceImpl"**;

**HelloService** helloService = (**HelloService**) **Naming**.lookup(url);

**String** **result** = helloService.sayHello(**"Jack"**);

**System**.**out**.println(**result**);

}

}

当我们运行以上 main() 方法，在控制台中看到“Hello Jack”输出，就表明 RMI 调用成功。

# 3 RMI 服务的局限性

可见，借助 JNDI 这个所谓的命名与目录服务，我们成功地发布并调用了 RMI 服务。实际上，JNDI 就是一个注册表，服务端将服务对象放入到注册表中，客户端从注册表中获取服务对象。在服务端我们发布了 RMI 服务，并在 JNDI 中进行了注册，此时就在服务端创建了一个 Skeleton（骨架），当客户端第一次成功连接 JNDI 并获取远程服务对象后，立马就在本地创建了一个 Stub（存根），远程通信实际上是通过 Skeleton 与 Stub 来完成的，数据是基于 TCP/IP 协议，在“传输层”上发送的。毋庸置疑，理论上 RMI 一定比 WebService 要快，毕竟 WebService 是基于 HTTP 的，而 HTTP 所携带的数据是通过“应用层”来传输的，传输层较应用层更为底层，越底层越快。

既然 RMI 比 WebService 快，使用起来也方便，那么为什么我们有时候还要用 WebService 呢？

其实原因很简单，WebService 可以实现跨语言系统之间的调用，而 RMI 只能实现 Java 系统之间的调用。也就是说，RMI 的跨平台性不如 WebService 好，假如我们的系统都是用 Java 开发的，那么当然首选就是 RMI 服务了。

貌似 RMI 确实挺优秀的，除了不能跨平台以外，还有那些问题呢？

笔者认为有两点局限性：

1. RMI 使用了 Java 默认的序列化方式，对于性能要求比较高的系统，可能需要使用其它序列化方案来解决（例如：Protobuf）。
2. RMI 服务在运行时难免会存在出故障，例如，如果 RMI 服务无法连接了，就会导致客户端无法响应的现象。

在一般的情况下，Java 默认的序列化方式确实已经足以满足我们的要求了，如果性能方面如果不是问题的话，我们需要解决的实际上是第二点，也就是说，让使系统具备 HA（High Availability，高可用性）。

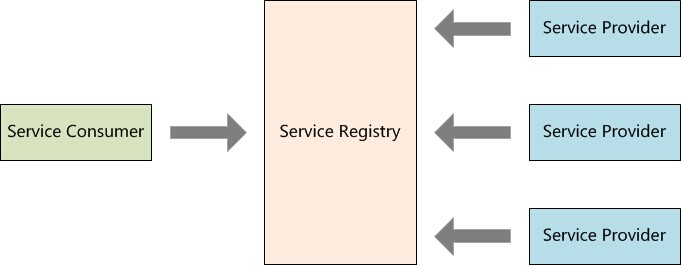
# 4 使用 ZooKeeper 提供高可用的 RMI 服务

ZooKeeper 是 Hadoop 的一个子项目，用于解决分布式系统之间的数据一致性问题。如果读者尚不了解 ZooKeeper 的工作原理与使用方法，可以通过以下链接来了解：

* [ZooKeeper 官网](http://zookeeper.apache.org/)
* [分布式服务框架 ZooKeeper – 管理分布式环境中的数据](https://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/)

本文假设读者已经对 ZooKeeper 有一定了解的前提下，对 RMI 的高可用性问题提供一个简单的解决方案。

要想解决 RMI 服务的高可用性问题，我们需要利用 ZooKeeper 充当一个 服务注册表（Service Registry），让多个 服务提供者（Service Provider）形成一个集群，让 服务消费者（Service Consumer）通过服务注册表获取具体的服务访问地址（也就是 RMI 服务地址）去访问具体的服务提供者。如下图所示：

[](http://static.oschina.net/uploads/space/2014/1115/211519_wVHA_223750.jpg)

需要注意的是，服务注册表并不是 Load Balancer（负载均衡器），提供的不是“反向代理”服务，而是“服务注册”与“心跳检测”功能。

利用服务注册表来注册 RMI 地址，这个很好理解，那么“心跳检测”又如何理解呢？说白了就是通过服务中心定时向各个服务提供者发送一个请求（实际上建立的是一个 Socket 长连接），如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，只会从还“活着”的服务提供者中选出一个做为当前的服务提供者。

也许读者会考虑到，服务中心可能会出现单点故障，如果服务注册表都坏掉了，整个系统也就瘫痪了。看来要想实现这个架构，必须保证服务中心也具备高可用性。

ZooKeeper 正好能够满足我们上面提到的所有需求。

1. 使用 ZooKeeper 的临时性 ZNode 来存放服务提供者的 RMI 地址，一旦与服务提供者的 Session 中断，会自动清除相应的 ZNode。
2. 让服务消费者去监听这些 ZNode，一旦发现 ZNode 的数据（RMI 地址）有变化，就会重新获取一份有效数据的拷贝。
3. ZooKeeper 与生俱来的集群能力（例如：数据同步与领导选举特性），可以确保服务注册表的高可用性。

## 4.1 服务提供者

需要编写一个 ServiceProvider 类，来发布 RMI 服务，并将 RMI 地址注册到 ZooKeeper 中（实际存放在 ZNode 上）。

package demo.zookeeper.rmi.server;

**import** demo.zookeeper.rmi.common.Constant;

**import** java.io.IOException;

**import** java.net.MalformedURLException;

**import** java.rmi.Naming;

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.util.concurrent.CountDownLatch;

**import** org.apache.zookeeper.CreateMode;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooDefs;

**import** org.apache.zookeeper.ZooKeeper;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

public **class** ServiceProvider {

private static final Logger LOGGER = LoggerFactory.getLogger(ServiceProvider.class);

*// 用于等待 SyncConnected 事件触发后继续执行当前线程*

private CountDownLatch latch = **new** CountDownLatch(1);

*// 发布 RMI 服务并注册 RMI 地址到 ZooKeeper 中*

public **void** publish(Remote remote, String host, int port) {

String url = publishService(remote, host, port); *// 发布 RMI 服务并返回 RMI 地址*

**if** (url != **null**) {

ZooKeeper zk = connectServer(); *// 连接 ZooKeeper 服务器并获取 ZooKeeper 对象*

**if** (zk != **null**) {

createNode(zk, url); *// 创建 ZNode 并将 RMI 地址放入 ZNode 上*

}

}

}

*// 发布 RMI 服务*

private String publishService(Remote remote, String host, int port) {

String url = **null**;

**try** {

url = String.format(**"rmi://%s:%d/%s"**, host, port, remote.getClass().getName());

LocateRegistry.createRegistry(port);

Naming.rebind(url, remote);

LOGGER.debug(**"publish rmi service (url: {})"**, url);

} **catch** (RemoteException | MalformedURLException e) {

LOGGER.error(**""**, e);

}

**return** url;

}

*// 连接 ZooKeeper 服务器*

private ZooKeeper connectServer() {

ZooKeeper zk = **null**;

**try** {

zk = **new** ZooKeeper(Constant.ZK\_CONNECTION\_STRING, Constant.ZK\_SESSION\_TIMEOUT, **new** Watcher() {

@Override

public **void** process(WatchedEvent event) {

**if** (event.getState() == Event.KeeperState.SyncConnected) {

latch.countDown(); *// 唤醒当前正在执行的线程*

}

}

});

latch.await(); *// 使当前线程处于等待状态*

} **catch** (IOException | InterruptedException e) {

LOGGER.error(**""**, e);

}

**return** zk;

}

*// 创建 ZNode*

private **void** createNode(ZooKeeper zk, String url) {

**try** {

byte[] data = url.getBytes();

String path = zk.create(Constant.ZK\_PROVIDER\_PATH, data, ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL); *// 创建一个临时性且有序的 ZNode*

LOGGER.debug(**"create zookeeper node ({} => {})"**, path, url);

} **catch** (KeeperException | InterruptedException e) {

LOGGER.error(**""**, e);

}

}

}

涉及到的 Constant 常量，见如下代码：

package demo.zookeeper.rmi.common;

public **interface** Constant {

String ZK\_CONNECTION\_STRING = **"localhost:2181"**;

int ZK\_SESSION\_TIMEOUT = 5000;

String ZK\_REGISTRY\_PATH = **"/registry"**;

String ZK\_PROVIDER\_PATH = ZK\_REGISTRY\_PATH + **"/provider"**;

}

注意：我们首先需要使用 ZooKeeper 的客户端工具创建一个持久性 ZNode，名为“/registry”，该节点是不存放任何数据的，可使用如下命令：

create /registry null

## 4.2 服务消费者

服务消费者需要在创建的时候连接 ZooKeeper，同时监听 /registry 节点的 NodeChildrenChanged 事件，也就是说，一旦该节点的子节点有变化，就需要重新获取最新的子节点。这里提到的子节点，就是存放服务提供者发布的 RMI 地址。需要强调的是，这些子节点都是临时性的，当服务提供者与 ZooKeeper 服务注册表的 Session 中断后，该临时性节会被自动删除。

**package** demo.zookeeper.rmi.client;

**import** demo.zookeeper.rmi.common.Constant;

**import** java.io.IOException;

**import** java.net.MalformedURLException;

**import** java.rmi.ConnectException;

**import** java.rmi.Naming;

**import** java.rmi.NotBoundException;

**import** java.rmi.Remote;

**import** java.rmi.RemoteException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.ThreadLocalRandom;

**import** org.apache.zookeeper.KeeperException;

**import** org.apache.zookeeper.WatchedEvent;

**import** org.apache.zookeeper.Watcher;

**import** org.apache.zookeeper.ZooKeeper;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**public** **class** ServiceConsumer {

**private** **static** **final** Logger LOGGER = LoggerFactory.getLogger(ServiceConsumer.class);

*// 用于等待 SyncConnected 事件触发后继续执行当前线程*

**private** CountDownLatch latch = **new** CountDownLatch(1);

*// 定义一个 volatile 成员变量，用于保存最新的 RMI 地址（考虑到该变量或许会被其它线程所修改，一旦修改后，该变量的值会影响到所有线程）*

**private** **volatile** List<String> urlList = **new** ArrayList<>();

*// 构造器*

**public** ServiceConsumer() {

ZooKeeper zk = connectServer(); *// 连接 ZooKeeper 服务器并获取 ZooKeeper 对象*

**if** (zk != **null**) {

watchNode(zk); *// 观察 /registry 节点的所有子节点并更新 urlList 成员变量*

}

}

*// 查找 RMI 服务*

**public** <T **extends** Remote> T lookup() {

T service = **null**;

**int** size = urlList.size();

**if** (size > 0) {

String url;

**if** (size == 1) {

url = urlList.get(0); *// 若 urlList 中只有一个元素，则直接获取该元素*

LOGGER.debug(**"using only url: {}"**, url);

} **else** {

url = urlList.get(ThreadLocalRandom.current().nextInt(size)); *// 若 urlList 中存在多个元素，则随机获取一个元素*

LOGGER.debug(**"using random url: {}"**, url);

}

service = lookupService(url); *// 从 JNDI 中查找 RMI 服务*

}

**return** service;

}

*// 连接 ZooKeeper 服务器*

**private** ZooKeeper connectServer() {

ZooKeeper zk = **null**;

**try** {

zk = **new** ZooKeeper(Constant.ZK\_CONNECTION\_STRING, Constant.ZK\_SESSION\_TIMEOUT, **new** Watcher() {

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getState() == Event.KeeperState.SyncConnected) {

latch.countDown(); *// 唤醒当前正在执行的线程*

}

}

});

latch.await(); *// 使当前线程处于等待状态*

} **catch** (IOException | InterruptedException e) {

LOGGER.**error**(**""**, e);

}

**return** zk;

}

*// 观察 /registry 节点下所有子节点是否有变化*

**private** **void** watchNode(**final** ZooKeeper zk) {

**try** {

List<String> nodeList = zk.getChildren(Constant.ZK\_REGISTRY\_PATH, **new** Watcher() {

@Override

**public** **void** process(WatchedEvent event) {

**if** (event.getType() == Event.EventType.NodeChildrenChanged) {

watchNode(zk); *// 若子节点有变化，则重新调用该方法（为了获取最新子节点中的数据）*

}

}

});

List<String> dataList = **new** ArrayList<>(); *// 用于存放 /registry 所有子节点中的数据*

**for** (String node : nodeList) {

**byte**[] data = zk.getData(Constant.ZK\_REGISTRY\_PATH + **"/"** + node, **false**, **null**); *// 获取 /registry 的子节点中的数据*

dataList.add(**new** String(data));

}

LOGGER.debug(**"node data: {}"**, dataList);

urlList = dataList; *// 更新最新的 RMI 地址*

} **catch** (KeeperException | InterruptedException e) {

LOGGER.**error**(**""**, e);

}

}

*// 在 JNDI 中查找 RMI 远程服务对象*

@SuppressWarnings(**"unchecked"**)

**private** <T> T lookupService(String url) {

T remote = **null**;

**try** {

remote = (T) Naming.lookup(url);

} **catch** (NotBoundException | MalformedURLException | RemoteException e) {

**if** (e **instanceof** ConnectException) {

*// 若连接中断，则使用 urlList 中第一个 RMI 地址来查找（这是一种简单的重试方式，确保不会抛出异常）*

LOGGER.**error**(**"ConnectException -> url: {}"**, url);

**if** (urlList.size() != 0) {

url = urlList.get(0);

**return** lookupService(url);

}

}

LOGGER.**error**(**""**, e);

}

**return** remote;

}

}

## 4.3 发布服务

我们需要调用 ServiceProvider 的 publish() 方法来发布 RMI 服务，发布成功后也会自动在 ZooKeeper 中注册 RMI 地址。

**package** demo.zookeeper.rmi.server;

**import** demo.zookeeper.rmi.common.HelloService;

**public** class Server {

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

**if** (args.length != 2) {

System.err.println(**"please using command: java Server <rmi\_host> <rmi\_port>"**);

System.exit(-1);

}

**String** host = args[0];

int port = Integer.parseInt(args[1]);

ServiceProvider provider = **new** ServiceProvider();

HelloService helloService = **new** HelloServiceImpl();

provider.publish(helloService, host, port);

Thread.sleep(Long.MAX\_VALUE);

}

}

注意：在运行 Server 类的 main() 方法时，一定要使用命令行参数来指定 host 与 port，例如：

java Server localhost 1099

java Server localhost 2099

以上两条 Java 命令可在本地运行两个 Server 程序，当然也可以同时运行更多的 Server 程序，只要 port 不同就行。

## 4.4 调用服务

通过调用 ServiceConsumer 的 lookup() 方法来查找 RMI 远程服务对象。我们使用一个“死循环”来模拟每隔 3 秒钟调用一次远程方法。

package demo.zookeeper.rmi.client;

**import** demo.zookeeper.rmi.common.**HelloService**;

public class **Client** {

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

**ServiceConsumer** consumer = new **ServiceConsumer**();

**while** (**true**) {

**HelloService** helloService = consumer.lookup();

**String** **result** = helloService.sayHello(**"Jack"**);

**System**.**out**.println(**result**);

**Thread**.sleep(3000);

}

}

}

## 4.5 使用方法

根据以下步骤验证 RMI 服务的高可用性：

1. 运行两个 Server 程序，一定要确保 port 是不同的。
2. 运行一个 Client 程序。
3. 停止其中一个 Server 程序，并观察 Client 控制台的变化（停止一个 Server 不会导致 Client 端调用失败）。
4. 重新启动刚才关闭的 Server 程序，继续观察 Client 控制台变化（新启动的 Server 会加入候选）。
5. 先后停止所有的 Server 程序，还是观察 Client 控制台变化（Client 会重试连接，多次连接失败后，自动关闭）。

# 5 总结

通过本文，我们尝试使用 ZooKeeper 实现了一个简单的 RMI 服务高可用性解决方案，通过 ZooKeeper 注册所有服务提供者发布的 RMI 服务，让服务消费者监听 ZooKeeper 的 Znode，从而获取当前可用的 RMI 服务。此方案局限于 RMI 服务，对于任何形式的服务（比如：WebService），也提供了一定参考。

如果再配合 ZooKeeper 自身的集群，那才是一个相对完美的解决方案，对于 ZooKeeper 的集群，请读者自行实践。

由于笔者水平有限，对于描述有误之处，还请各位读者提出建议，并期待更加优秀的解决方案。