RMI 原理及应用

Jack. Wang

2008-05-24

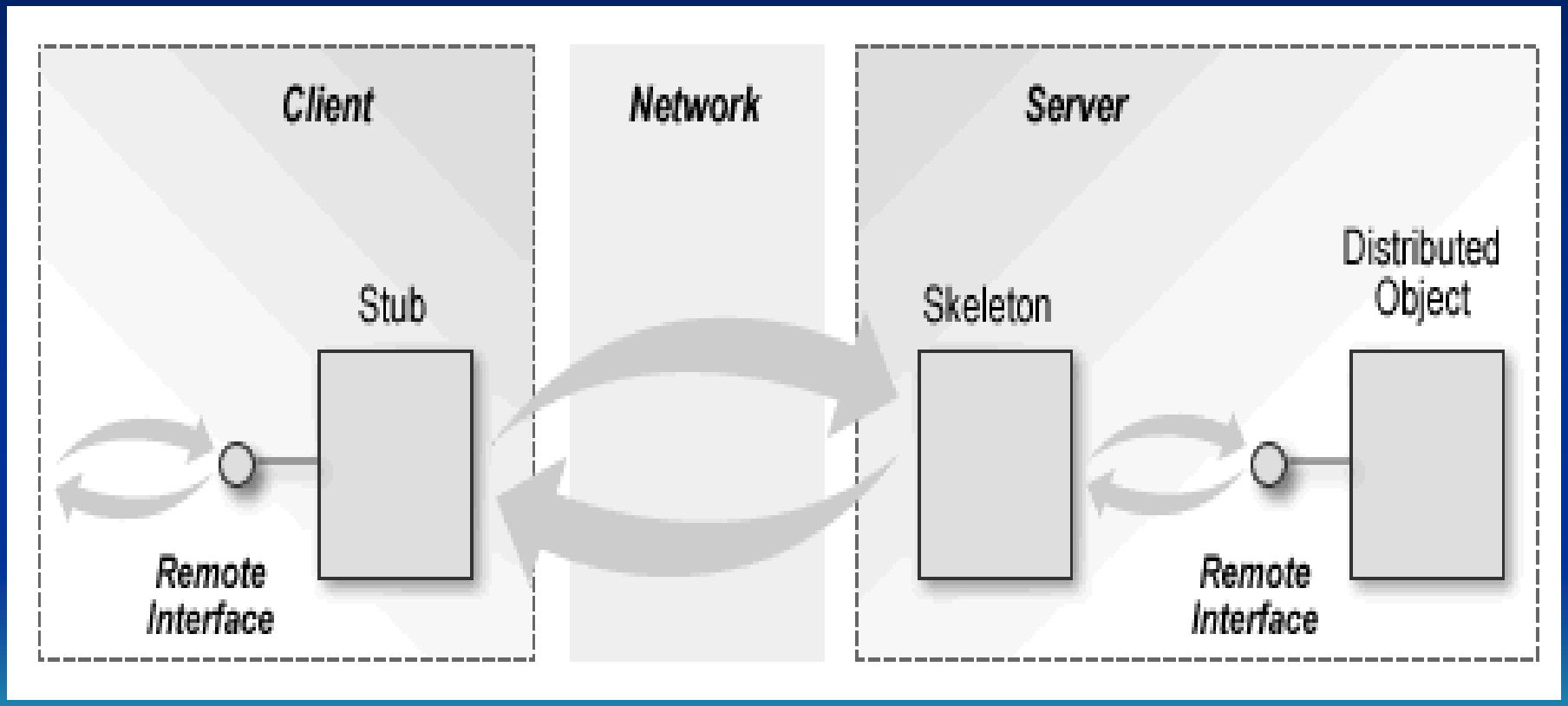
前言

- 在分布式系统中Java 最先支持的是Socket
  优点：Socket的灵活性和扩展性很强
  缺点：Socket需要客户端和服务端, 在应用级定义通信协议，这是非常麻烦的一件 事
- RMI
  1. 在single JVM中，我们可以通过直接调用java object instance来实现通信，那么在远程通信时，如果也能按照这种方式当然是最好了(jdk1.1)
  2. RMI 是 J2EE 的很多分布式技术的基础，比如 RMI-IIOP 乃至 EJB。

前言

- RPC（Remote Procedure Call ）可以用于一个进程调用另一个进程（很可能在另一个远程主机上）中的*过程*，从而提供了过程的分布能力
- RMI 则在 RPC 的基础上向前又迈进了一步充分支持面向对象的特性
- java.rmi 包

分布式对象

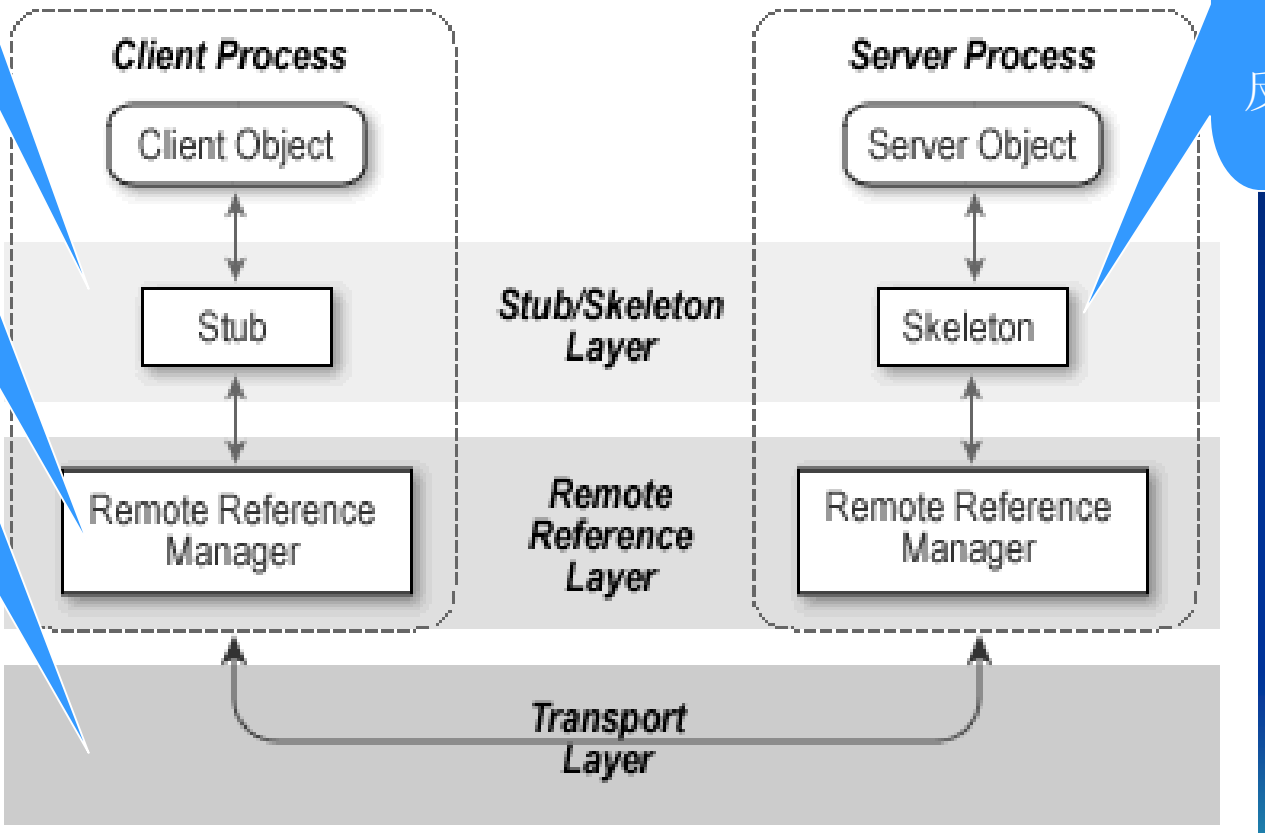


RMI 架构

客户程序和服务程序交互的

负责处理远程对象引

该层负责请求应答消息传输



Java 1.2 之后，RMI 不再需要 Skeleton 而是通过反射机制实现

深入细节

- 编列（Marshal）
- 参数究竟是传值还是传引用
- Skeleton 请求处理
- 异常处理

编列（Marshal）

- 当客户程序调用 Stub 时，Stub 负责将方法的参数转换为序列化（Serialized）形式，我们使用一个特殊的术语，即编列
- 编列的目的是将这些参数转换为可移植的形式，从而可以通过网络传输到远程的服务器一端

深入细节

- 编列（Marshal）
- 参数究竟是传值还是传引用
- Skeleton 请求处理
- 异常处理

参数究竟是传值还是传引用

1. 对于基本的原始类型（整型，字符型等等），将被自动的序列化，以传值的方式编列。
2. 对于 Java 的对象，如果该对象是可序列化的（实现了 java.io.Serializable 接口），则通过 Java 序列化机制自动地加以序列化，以传值的方式编列。对象之中包含的原始类型以及所有被该对象引用，且没有声明为 transient 的对象也将自动的序列化。当然，这些被引用的对象也必须是可序列化的。

参数究竟是传值还是传引用

3. 绝大多数内建的 Java 对象都是可序列化的。 对于不可序列化的 Java 对象 java.io.File 最典型），或者对象中包含对不可序列化，且没有声明为 transient 的其它对象的引用。则编列过程将向客户程序抛出异常，而宣告失败。
4. 客户程序可以调用远程对象，没有理由禁止调用参数本身也是远程对象（实现了 java.rmi.Remote 接口的类的实例）

深入细节

- 编列（Marshal）
- 参数究竟是传值还是传引用
- Skeleton 请求处理
- 异常处理

Skeleton 请求处理

- 在 RMI 中，有多种的可能的传输机制，比如点对点（Point-to-Point）以及广播（Multicast），不过，**在当前的 RMI 版本中只支持点对点协议**(java.rmi.server.UnicastRemoteObject)
- Skeleton 对象负责将请求转换为对实际的远程对象的方法调用
- 通过反编列（Unmarshal）过程。所有序列化的参数被转换为 Java 形式，其中作为参数的远程对象（实际上发送的是远程引用）被转换为服务器端本地的 Stub 对象。

深入细节

- 编列（Marshal）
- 参数究竟是传值还是传引用
- Skeleton 请求处理
- 异常处理

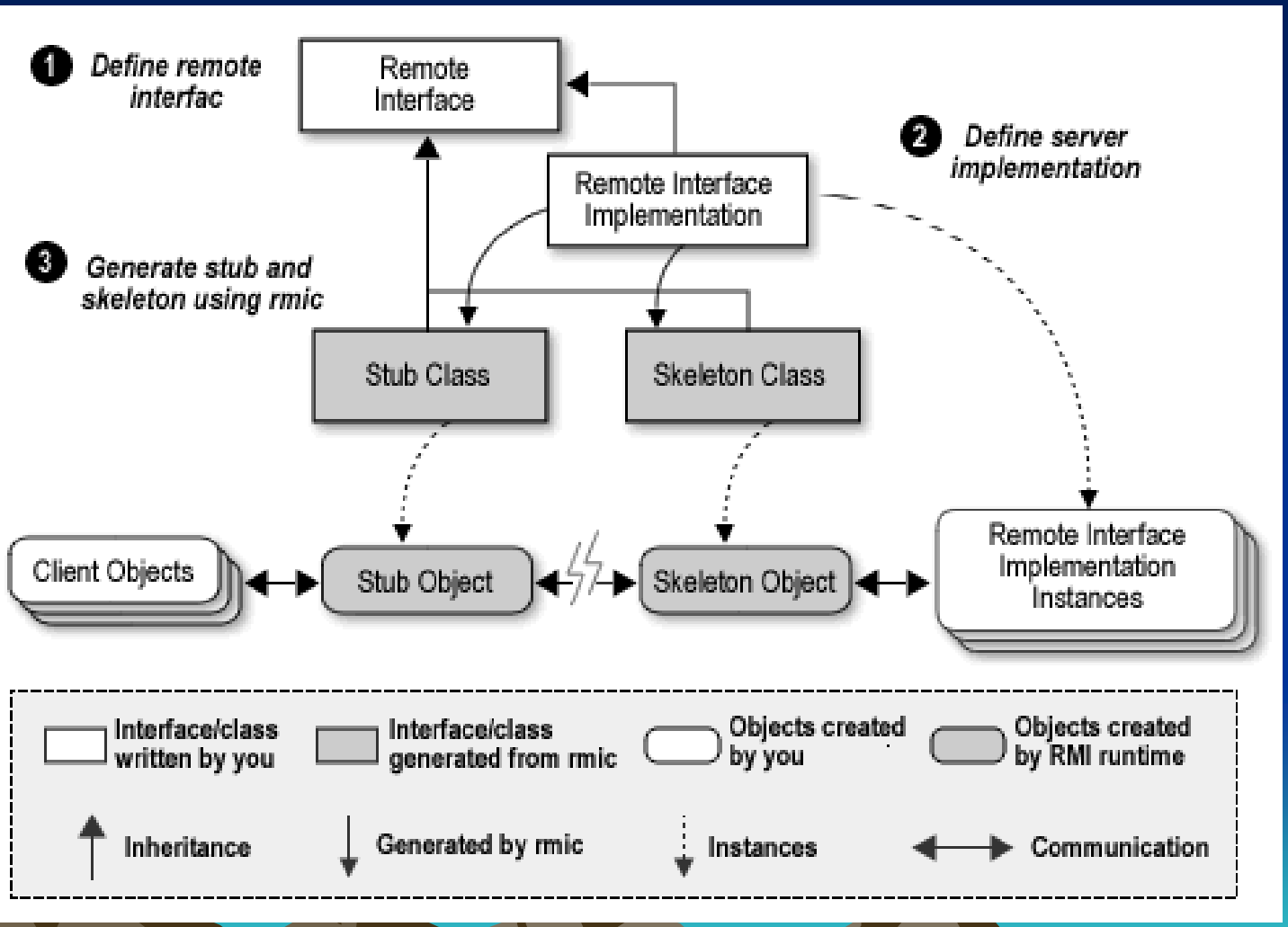
异常处理

- 如果方法调用有返回值或者抛出异常，则 Skeleton 负责编列返回值或者异常，通过服务器端的远程引用层，经传输层传递给客户端；相应地，客户端的远程引用层和 Stub 负责反编列并最终将结果返回给客户程序。

RMI 对象服务

- 当服务器端想向客户端提供基于 RMI 的服务时，它需要将一个或多个远程对象注册到本地的 RMI 注册表中（参见java.rmi.registry.Registry API）。
- 客户程序通过命名服务（参见 java.rmi.Naming API），获得指向远程对象的远程引用
- 在 Naming 中的 lookup() 方法找到远程对象所在的主机后，它将检索该主机上的 RMI 注册表，并请求所需的远程对象。如果注册表发现被请求的远程对象，它将生成一个对该远程对象的远程引用，并将其返回给客户端，客户端则基于远程引用生成相应的 Stub 对象，并将引用传递给调用者

实战 RMI



1. 远程接口

- ```
  import java.rmi.Remote;
  import java.rmi.RemoteException;
  public interface IService extends Remote {
      public Object executeTask(ITask task)
      throws  RemoteException;
  }
  ```


  ```

远程接口实现

- import java.rmi.RemoteException;
  import java.rmi.server.UnicastRemoteObject;
  public class IServiceImpl extends UnicastRemoteObject implements IService {
  .....
  }

  UnicastRemoteObject的构造函数会调用他的exportObject() 方法。导出（Export）对象是指使远程对象准备就绪，可以接受进来的调用的过程。而这个过程的最重要内容就是建立服务器套接字，监听特定的端口，等待客户端的调用请求。

任务接口

- import java.io.Serializable;
  public interface ITask extends Serializable
  {
  .....
  }

任务实现类

```
public class TaskImpl implements ITask {
   ....
}
```

Server 端代码

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Server {
   private static Registry createRegistry() {
     ….
     }
 public static void bind() {
  Registry registry = null;
  registry = createRegistry();
  try {
   IServiceImpl impl = new IServiceImpl();
   registry.rebind("mytask", impl);
   } catch (Exception e) {
      e.printStackTrace();
   }
 }
  public static void main(String[] args) {
    try {
      bind();
    } catch (Exception e) {
      e.printStackTrace();
  }
 }
}
```

Client 端代码

```
public class RMIClient {
 public static void getRemoteObject() throws Exception {
   IService obj = (IService)
   Naming.lookup("rmi://210.43.109.28:1111/mytask");
   TaskImpl task = new TaskImpl();
   Object result = obj.executeTask(task);
  }
public static void main(String[] args) {
 try {
  getRemoteObject();
 } catch (Exception e) {
  e.printStackTrace();
 }
}
}
```

The end

谢谢