RPC入门总结（二）RMI的原理和使用

转载 Zenhobby 最后发布于2017-11-20 14:29:46 阅读数 4295 收藏

转载：架构设计：系统间通信（8）——通信管理与RMI 上篇

转载：JAVA RMI分布式原理和应用

转载：

一、RMI

RMI（Remote Method Invocation，远程方法调用），是JAVA早在JDK 1.1中提供的JVM与JVM之间进行 对象方法调用的技术框架的实现（在JDK的后续版本中，又进行了改进）。通过RMI技术，某一个本地的JVM可以调用存在于另外一个JVM中的对象方法，就好像它仅仅是在调用本地JVM中某个对象方法一样。例如RMI客户端中的如下调用：

```
List< UserInfo > users = remoteServiceInterface.queryAllUserinfo();
```

看似remoteServiceInterface对象和普通的对象没有区别，但实际上remoteServiceInterface对象的具体方法实现却不在本地的JVM中，而是在某个远程的JVM中（这个远程的JVM可以是RMI客户端同属于一台物理机，也可以属于不同的物理机）

二、RMI的适用场景

RMI是基于JAVA语言的，也就是说在RMI技术框架的描述中，只有**Server端使用的是JAVA语言并且Client端也是用的JAVA语言**，才能使用RMI技术（目前在codeproject.com中有一个开源项目名字叫做“RMI for C++”，可以实现JAVA To C++的RMI调用。但是这是一个第三方的实现，并不是java的标准RMI框架定义，所以并不在我们的讨论范围中）。RMI适用于两个系统都**主要使用JAVA语言进行构造**，不需要考虑跨语言支持的情况。并且**对两个JAVA系统的通讯速度有要求**的情况。

RMI 是一个良好的、特殊的RPC实现：使用**JRMP协议**承载数据描述，可以使用**BIO和NIO两种IO通信模型**。RMI框架是可以在**大规模集群系统**中使用的，当然是不是使用RMI技术，还要看您的产品的技术背景、团队的技术背景、公司的业务背景甚至客户的非技术背景等。

三、RMI的基本框架

从设计角度上讲，JAVA采用的是三层结构模式来实现RMI。在整个体系结构中，有如下几个关键角色构成了通信双方：

1.客户端：

1）桩(StubObject)：远程对象在客户端上的代理；

2）远程引用层(RemoteReference Layer)：解析并执行远程引用协议；

3）传输层(Transport)：发送调用、传递远程方法参数、接收远程方法执行结果。

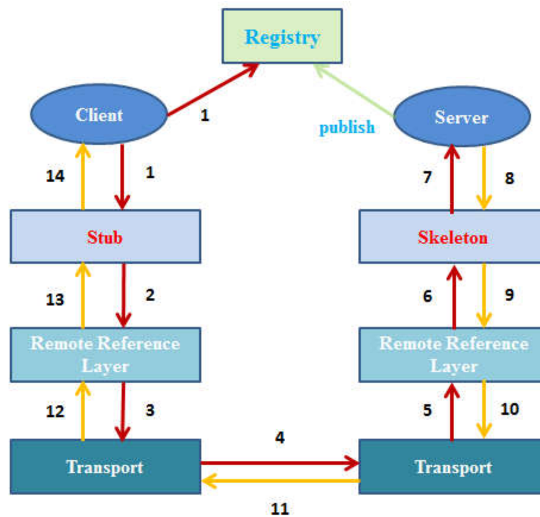
2.服务端：

1）骨架(Skeleton)：读取客户端传递的方法参数，调用服务器方的实际对象方法，并接收方法执行后的返回值；

2）远程引用层(Remote ReferenceLayer)：处理远程引用语法之后向骨架发送远程方法调用；

3）传输层(Transport)：监听客户端的入站连接，接收并转发调用到远程引用层。

3.注册表(Registry)：以URL形式注册远程对象，并向客户端回复对远程对象的引用。



在实际的应用中，客户端并没有真正的和服务端直接对话来进行远程调用，而是通过本地JVM环境下的**桩对象**来进行的。

远程调用过程：

1）客户端从远程服务器的注册表中查询并**获取远程对象引用。**当进行远程调用时，客户端首先会与桩对象(Stub Object)进行对话，而这个桩对象将远程方法所需的参数进行序列化后，传递给它下层的远程引用层(**RRL**)；

2）桩对象与远程对象具有相同的接口和方法列表，当客户端调用远程对象时，实际上是由相应的**桩对象代理**完成的。远程引用层在将桩的本地引用转换为服务器上对象的远程引用后，再将调用传递给传输层(Transport)，由传输层通过**TCP**协议发送调用；

3）在服务器端，传输层监听入站连接，它一旦接收到客户端远程调用后，就将这个引用转发给其上层的远程引用层；

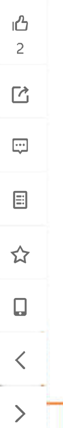
4）服务器端的远程引用层将客户端发送的远程应用转换为本地虚拟机的引用后，再将请求传递给骨架(Skeleton)；

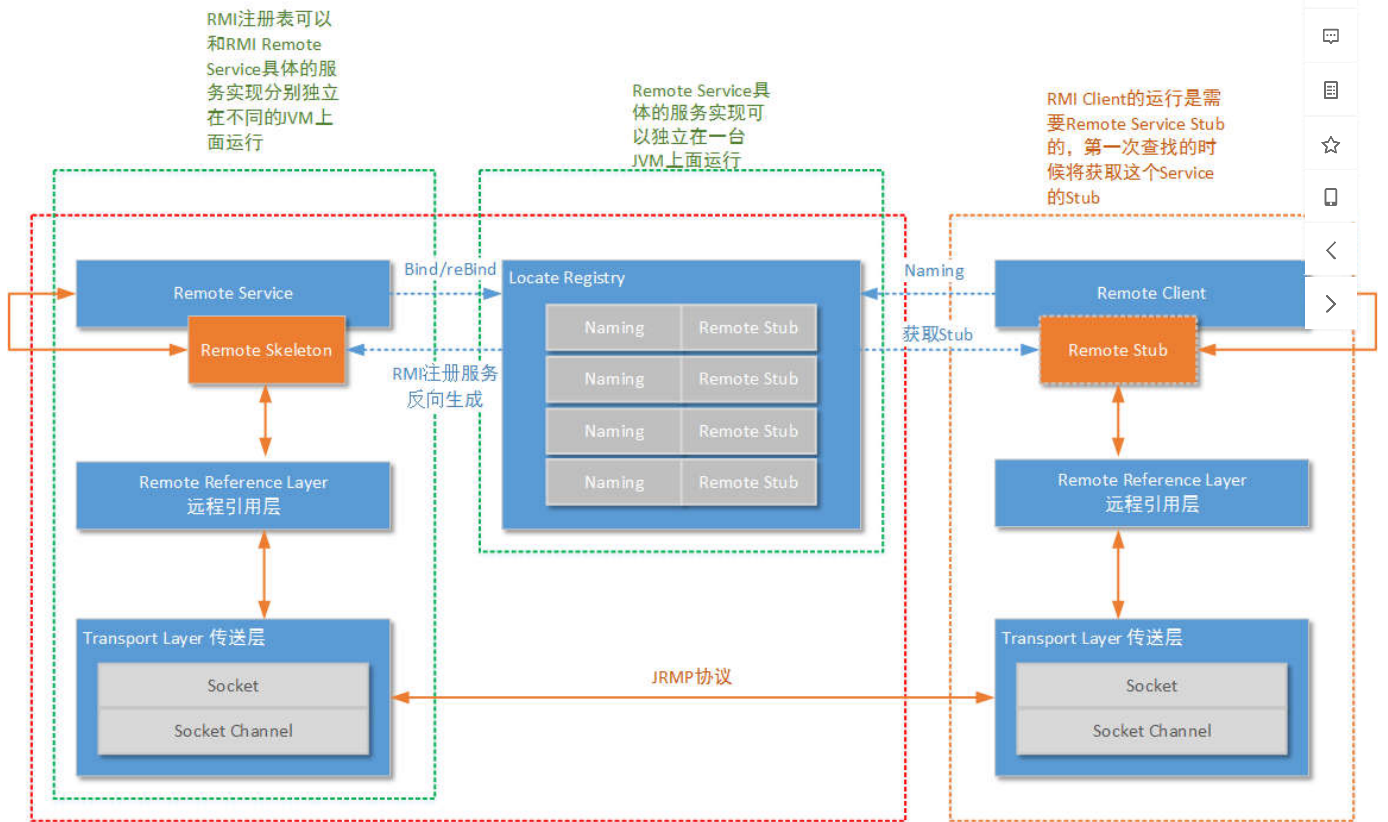
5）骨架读取参数，又将请求传递给服务器，最后由服务器进行实际的方法调用。

结果返回过程：

1）如果远程方法调用后有返回值，则服务器将这些结果又沿着"**骨架->远程引用层->传输层**"向下传递；

2）客户端的传输层接收到返回值后，又沿着"**传输层->远程引用层->桩**"向上传递，然后由桩来反序列化这些返回值，并将最终的结果传递给客户端程序。





从技术的角度上讲，有如下几个主要类或接口扮演着上述三层模型中的关键角色：

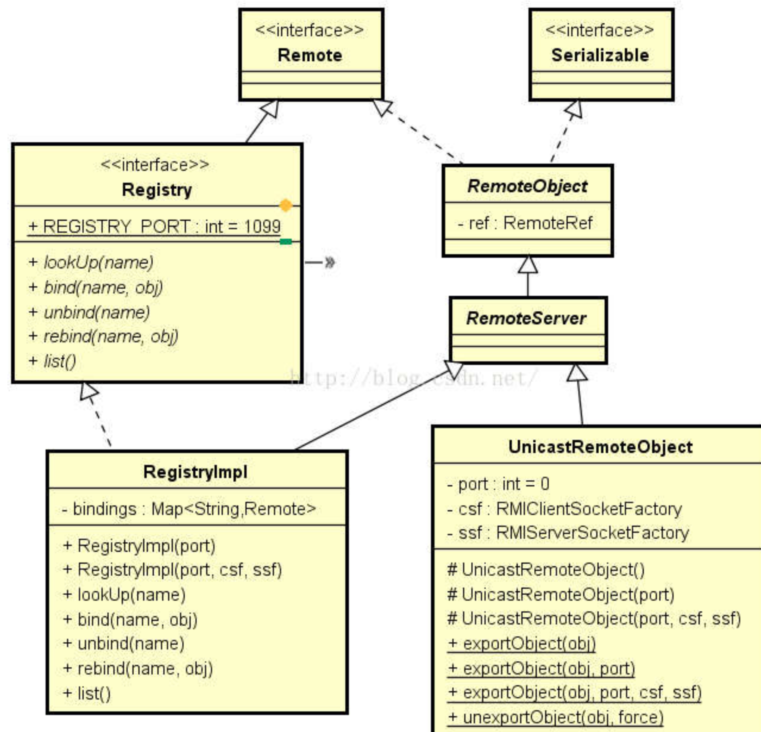
1）注册表：java.rmi.**Naming**和java.rmi.**Registry**；

2）骨架：java.rmi.remote.**Skeleton**

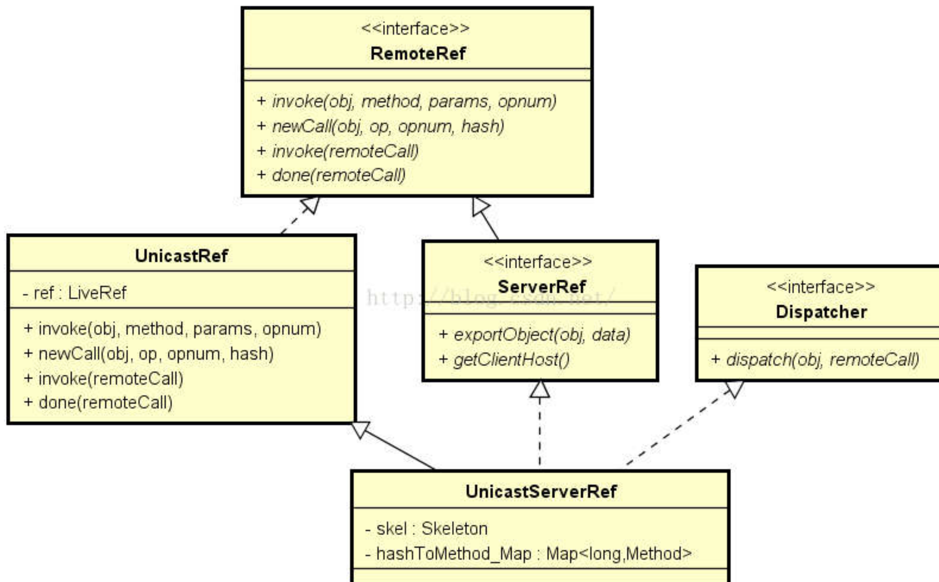
3）桩：java.rmi.server.**RemoteStub**

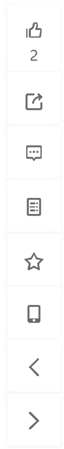
4）远程引用层：java.rmi.server.**RemoteRef**和sun.rmi.transport.**Endpoint**；

5）传输层：sun.rmi.transport.**Transport**



举报





四、RMI的开发流程

作为一般的RMI应用，JAVA为我们隐藏了其中的处理细节，而让开发者有更多的精力和时间花在实际的应用中。开发RMI的步骤如下所述：

1.**服务端**：

1）定义**Remote子接口**，在其内部定义要发布的远程方法，并且这些方法都要Throws **RemoteException**；

2）定义**远程对象的实现类**，通常有两种方式：

a. 继承**UnicastRemoteObject**或**Activatable**，并同时实现**Remote**子接口；

b. 只实现**Remote**子接口和java.io.Serializable接口。

3）**编译桩**（在JAVA 1.5及以后版本中，如果远程对象实现类继承了**UnicastRemoteObject**或**Activatable**，则无需此步，由JVM自动完成。否则需手工利用RMIC工具编译生成此实现类对应的桩类，并放到和实现类相同的编译目录下）；

4）启动服务器：依次完成**注册表的启动**和**远程对象绑定**。另外，如果远程对象实现类在定义时没有继承UnicastRemoteObject或Activatable，则必须在服务器端**显式的调用UnicastRemoteObject**类中某个**重载的exportObject(Remote remote)静态方法**，将此实现类对象导出成为一个真正的远程对象。

2.**客户端**：

1）定义用于接收远程对象的Remote子接口，只需实现java.rmi.Remote接口即可。但要求**必须与服务器端对等的Remote子接口保持一致**，即有相同的接口名称、包路径和方法列表等。

2）通过符合JRMP规范的**URL字符串**在注册表中获取并**强转成Remote子接口对象**；

3）调用这个Remote子接口对象中的某个方法就是为一次远程方法调用行为。

下面结合一个例子来说明RMI分布式应用的开发步骤。

背景：远程系统管理接口SystemManager允许客户端远程调用其内部的某个方法，来获取服务器环境下某个属性的值。因此，第一步需要在服务端和与之通信的客户端环境下，定义一个完全一样的SystemManager接口，将此接口标记为远程对象。

1.在服务端和客户端定义对等Remote子接口(SystemManager)

```
1  /**
2   * 为代码可读性考虑省略了引用源码中的作者信息如下
3   * @author  <a href="mailto:code727@gmail.com">DanieLe</a>
4   * @version 1.0.0, 2013-5-21
5   * @see
6   * @since   AppDemo1.0.0
7   */
```

```
1  package com.daniele.appdemo.rmi;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  import com.daniele.appdemo.test.domain.User;
7
8  /**
9   * 系统管理远程对象接口
10  */
11 public interface SystemManager extends Remote {
12
```

```
13      /**  14      * 发布的远程对象方法一：获取所有系统环境信息
15       */
16      public String getAllSystemMessage() throws RemoteException;
17
18      /**
19       * 发布的远程对象方法二：获取指定的系统环境信息
20       */
21      public String getSystemMessage(String properties) throws RemoteException;
22  }
```

2.在服务端定义Remote子接口的实现类(SystemManagerImpl)，即远程对象的实际行为逻辑。

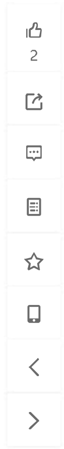
系统管理远程对象实现类。有两种实现方式:

* 1.继承**UnicastRemoteObject**或**Activatable**,并同时实现**Remote**子接口
* 2.只实现**Remote**子接口，这种方式灵活但比较复杂:
* 1)要求此实现类必须实现**java.io.Serializable**接口;
* 2)通过这种方式定义的实现类此时还不能叫做远程对象实现类，因为在服务器端绑定远程对象之前，还需要利用JDK提供的rmic工具将此实现类手工编译生成对应的**桩实现类**，并放到和它相同的编译目录下。

```
1   package com.daniele.appdemo.rmi.server;
2
3   import java.io.Serializable;
4   import java.rmi.RemoteException;
5   import java.rmi.server.RemoteServer;
6   import java.rmi.server.ServerNotActiveException;
7
8   import org.apache.log4j.Logger;
9
10  import com.daniele.appdemo.rmi.SystemManager;
11  import com.daniele.appdemo.test.domain.User;
12  import com.daniele.appdemo.util.SystemUtils;
13
14  public class SystemManagerImpl implements SystemManager, Serializable {
15
16  //public class SystemManagerImpl extends UnicastRemoteObject implements SystemManager {
17
18      private static final long serialVersionUID = 9128780104194876777L;
19      private static final Logger logger = Logger.getLogger(SystemManagerImpl.class);
20      private static SystemManagerImpl systemManager = null;
21
22      /**
23       * 在服务端本地的匿名端口上创建一个用于监听目的的UnicastRemoteObject对象
24       */
25      private SystemManagerImpl() throws RemoteException {
26          super();
27          // 在控制台中显示远程对象被调用，以及返回结果时产生的日志
28          RemoteServer.setLog(System.out);
29      }
30
31      public static SystemManagerImpl getInstance() throws RemoteException {
32          if (systemManager == null) {
33              synchronized (SystemManagerImpl.class) {
34                  if (systemManager == null)
35                      systemManager = new SystemManagerImpl();
36              }
37          }
38          return systemManager;
39      }
40
41      public String getAllSystemMessage() throws RemoteException {
42          try {
43              /*
44               * getClientHost()方法可以获取触发当前远程方法被调用时的客户端的主机名。
45               * 在远程服务端的环境中，如果当前线程实际上没有运行客户端希望调用的远程方法时，
46               * 则会抛出ServerNotActiveException。
47               * 因此，为了尽量避免这个异常的发生，它通常用于远程方法的内部实现逻辑中，
48               * 以便当此方法真正的被调用时，可以记录下哪个客户端在什么时间调用了这个方法。
49               */
50                  logger.info("Client {" + RemoteServer.getClientHost() + "} invoke method [getAllSystemMessage()]" );
51          } catch (ServerNotActiveException e) {
52              e.printStackTrace();
53          }
54          return SystemUtils.formatSystemProperties();
```

举报

```
    }
    public String getSystemMessage(String properties) throws RemoteException {
        try {
            logger.info("Client {"
                    + RemoteServer.getClientHost()
                    + "} invoke method [getAllSystemMessage(String properties)]");
         } catch (ServerNotActiveException e) {
            e.printStackTrace();
        }
        return SystemUtils.formatSystemProperties(properties.split(","));
    }

}
```



3.编译生成打桩类

由于SystemManagerImpl 不是通过继承UnicastRemoteObject 或 Activatable来实现的，因此在服务器端需要利用JDK提供的**rmic**工具编译生成对应的桩类。否则，此步略过。例如，在Windows环境下打开命令控制台后

1）进入工程根目录：

cd d:\ Development\AppDemo

2）桩编译:

rmic -classpath WebContent\WEB-INF\classes com.daniele.appdemo.rmi.server.SystemManagerImpl

语法格式为:

rmic -classpath <**远程对象实现类bin目录的相对路径**> <**远程对象实现类所在的包路径**>.<**远程对象实现类的名称**>

完成后，rmic将在相对于根目录的com\daniele\appdemo\rmi\server子目录中自动生成**SystemManagerImpl_Stub**桩对象类 (即"远程对象实现类名称**_Stub**"）的编译文件，此时需要再将**此编译文件**拷贝到**与远程对象实现类SystemManagerImpl相同的编译目录**(\WebContent\WEB-INF\classes\com\daniele\appdemo\rmi\server)中，否则在服务器端发布远程对象时将会抛出java.rmi.StubNotFoundException。如下图所示。

需要特别注意的是：如果服务端中的远程对象实现类存在有对应的桩对象类编译文件，则要求在**RMI客户端的环境**中，也**必须有这个对等的桩对象类编译文件**，即意味着这个文件在两端有着**相同的包路径、文件名和内部实现细节**。因此，最简单的做法就是连同整个包(com\daniele\appdemo\rmi\server)在内，将图2中的SystemManagerImpl_Stub.class文件拷贝到RMI客户端工程的bin目录下即可。如下图。否则，当RMI客户端调用远程服务时将会抛出java.rmi.StubNotFoundException。

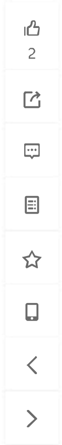
D:\Development\RMIClient\bin\com\daniele\appdemo\rmi\server

名称	修改日期	类型	大小
SystemManagerImpl_Stub.class	2013/5/23 18:33	CLASS 文件	

4.创建用于发布远程对象目的用的服务器(SystemManagerServer)

```
package com.daniele.appdemo.rmi.server;

import java.io.IOException;
import java.util.Arrays;
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

import org.apache.log4j.Logger;

import com.daniele.appdemo.rmi.SystemManager;

/**
 *   系统管理远程服务器，它主要完成如下任务：
 *     1.在绑定之前先启动注册表服务。
 *     2.将远程对象SystemManager绑定到注册表中，以便让客户端能远程调用这个对象所发布的方法；
 */
public class SystemManagerServer {

    private static final Logger logger = Logger.getLogger(SystemManagerServer.class);

    public static void main(String[] args) {
        try {

                SystemManager systemManager = SystemManagerImpl.getInstance();

                /*
```

```
28                 *  如果远程对象实现类不是通过继承UnicastRemoteObject或Activatable来定义的, 29
                 *  则必须在服务器端显示的调用UnicastRemoteObject类中某个重载的exportObject(Remote remote)静态方法, 30
                 *  将此实现类的某个对象导出成为一个真正的远程对象。否则，此步省略。 31                  */
32                UnicastRemoteObject.exportObject(systemManager);
33
34                int port = 9527;
35                String url = "rmi://localhost:" + port + "/";
36                String remoteObjectName = "systemManager";
37
38                /*
39                 *  在服务器的指定端口(默认为1099)上启动RMI注册表。
40                 *  必不可缺的一步，缺少注册表创建，则无法绑定对象到远程注册表上。
41                 */
42                LocateRegistry.createRegistry(port);
43
44                /*
45                 *  将指定的远程对象绑定到注册表中:
46                 *  1.如果端口号不为默认的1099，则绑定时远程对象名称中必须包含Schema,
47                 *     即"rmi://<host或ip><:port>/"部分，并且Schema里指定的端口号应与createRegistry()方法参数中的保持一致
48                 *  2.如果端口号为RMI默认的1099，则远程对象名称中不用包含Schema部分，直接定义名称即可
49                 */
50                if (port == 1099)
51                    Naming.rebind(remoteObjectName, systemManager);
52                else
53                    Naming.rebind(url + remoteObjectName, systemManager);
54               logger.info("Success bind remote object " + Arrays.toString(Naming.list(url)));
55            } catch (IOException e) {
56                e.printStackTrace();
57            }
58        }
59
60    }
```



此处需要注意：Registry所在的JVM可以和Service提供者的JVM不相同，即可以专门使用一个JVM作为服务注册中心使用，此时的代码演变如下：

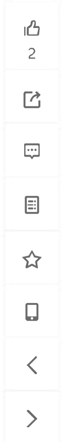
```
 1  package testRMI;
 2
 3
 4  import java.rmi.registry.LocateRegistry;
 5  import java.rmi.registry.Registry;
 6
 7  public class RemoteRegistryUnicastMain {
 8      public static void main(String[] args) throws Exception {
 9          /*
10           * 我们通过LocateRegistry的get方法，寻找一个存在于远程JVM上的RMI注册表
11           * */
12          Registry registry = LocateRegistry.getRegistry("192.168.61.1", 1099);
13
14
15          // 以下是向远程RMI注册表（绑定/重绑定）RMI Server的Stub。
16          // 同样的远程RMI注册表的JVM-classpath下，一定要有这个RMI Server的Stub
17          RemoteUnicastServiceImpl remoteService = new RemoteUnicastServiceImpl();
18
19          /*
20           *
21           * 注册的服务仓库存在于192.168.61.1这个IP上
22           * 使用注册表registry进行绑定或者重绑定时，不需要写完整的RMI URL
23           * */
24          registry.bind("queryAllUserinfo" , remoteService);
25      }
26  }
```

使用getRegistry函数可以获取远程RMI注册表，之后的工作与本地注册表没啥两样。

5.创建发出远程调用请求的客户端(SystemManagerClient)

```
1  package com.daniele.appdemo.rmi.client;
2
3  import java.io.IOException;
4  import java.rmi.Naming;
5  import java.rmi.NotBoundException;
6
7  import com.daniele.appdemo.rmi.SystemManager;
```

```java
import com.daniele.appdemo.test.domain.User;

/**
 * 系统管理进行远程调用的客户端
 */

public class SystemManagerClient {

    public static void main(String[] args) {
        /*
         * RMI URL格式：Schame/<远程对象名>
         * 1.Schame部分由"rmi://<server host或IP>[:port]"组成，
         *    如果远程对象绑定在服务端的1099端口(默认)上，则port部分可以省略，否则必须指定。
         * 2.URL最后一个"/"后面的值为远程对象绑定在服务端注册表上时定义的远程对象名，
         *    即对应Naming.rebind()方法第一个参数值中最后一个"/"后面的值
         */
        String rmi = "rmi://192.168.1.101:9527/systemManager";
        try {
                /*
                 * 根据URL字符串查询并获取远程服务端注册表中注册的远程对象，
                 * 这里返回的是本地实现了Remote接口的子接口对象(Stub)，
                 * 它与服务端中的远程对象具有相同的接口和方法列表，因而作为在客户端中远程对象的一个代理。
                 */
                SystemManager systemManager = (SystemManager) Naming.lookup(rmi);

//              System.out.println(systemManager.getAllSystemMessage());
                System.out.println(systemManager.getSystemMessage("java.version,os.name"));
            } catch (IOException e) {
                e.printStackTrace();
            } catch (NotBoundException e) {
                e.printStackTrace();
            }
    }

}
```



完成后，再依次到服务器和客户端上启动SystemManagerServer和SystemManagerClient即可。

五、RMI的源码解析

1. 服务端输出远程对象

Server端调用UnicastRemoteObject的export方法输出远程对象，export方法会在一个线程里监听某个TCP端口上的方法调用请求：

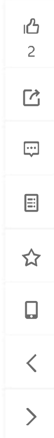
```java
public void exportObject(Target target) throws RemoteException {
    // other code
    while (true) {
        ServerSocket myServer = server;
        if (myServer == null)
          return;
        Throwable acceptFailure = null;
        final Socket socket;

        try {
            socket = myServer.accept();
            /*
             * Find client host name (or "0.0.0.0" if unknown)
             */
            InetAddress clientAddr = socket.getInetAddress();
            String clientHost = (clientAddr != null
                        ? clientAddr.getHostAddress()
                        : "0.0.0.0");
            /*
             * Spawn non-system thread to handle the connection
             */
            Thread t = (Thread) java.security.AccessController.doPrivileged (
                    new NewThreadAction(new ConnectionHandler(socket,clientHost),
                            "TCP Connection(" + ++ threadNum + ")-" + clientHost,
                            true, true));
            t.start();
        } catch (IOException e) {
            acceptFailure = e;
        } catch (RuntimeException e) {
            acceptFailure = e;
        } catch (Error e) {
```

```
32              acceptFailure = e;   33 |          }
34      }
35      // other code
36  }
```

Server端就是在ServerSocket的accept方法上面监听到来的请求，如果有新的方法调用请求到来，Server产生一个单独的线程来处理新接收的请求：

```
1   public void dispatch(Remote obj, RemoteCall call) throws IOException {
2       // positive operation number in 1.1 stubs;
3       // negative version number in 1.2 stubs and beyond...
4       int num;
5       long op;
6       try {
7           // read remote call header
8           ObjectInput in;
9           try {
10              in = call.getInputStream();
11              num = in.readInt();
12              if (num >= 0) {
13                  if (skel != null) {
14                  oldDispatch(obj, call, num);
15                  return;
16                  } else {
17                      throw new UnmarshalException(
18                          "skeleton class not found but required " +
19                          "for client version");
20                  }
21              }
22              op = in.readLong();
23          } catch (Exception readEx) {
24              throw new UnmarshalException("error unmarshalling call header",
25                                  readEx);
26          }
27          /*
28           * Since only system classes (with null class loaders) will be on
29           * the execution stack during parameter unmarshalling for the 1.2
30           * stub protocol, tell the MarshalInputStream not to bother trying
31           * to resolve classes using its superclasses's default method of
32           * consulting the first non-null class loader on the stack.
33           */
34          MarshalInputStream marshalStream = (MarshalInputStream) in;
35          marshalStream.skipDefaultResolveClass();
36          Method method = (Method) hashToMethod_Map.get(new Long(op));
37          if (method == null) {
38              throw new UnmarshalException("invalid method hash");
39          }
40          // if calls are being logged, write out object id and operation
41          logCall(obj, method);
42          // unmarshal parameters
43          Class[] types = method.getParameterTypes();
44          Object[] params = new Object[types.length];
45          try {
46              unmarshalCustomCallData(in);
47              for (int i = 0; i < types.length; i++) {
48                  params[i] = unmarshalValue(types[i], in);
49              }
50          } catch (java.io.IOException e) {
51              throw new UnmarshalException(
52                  "error unmarshalling arguments", e);
53          } catch (ClassNotFoundException e) {
54              throw new UnmarshalException(
55                  "error unmarshalling arguments", e);
56          } finally {
57              call.releaseInputStream();
58          }
59          // make upcall on remote object
60          Object result;
61          try {
62              result = method.invoke(obj, params);
63          } catch (InvocationTargetException e) {
64              throw e.getTargetException();
65          }
66          // marshal return value
67          try {
68              ObjectOutput out = call.getResultStream(true);
69              Class rtype = method.getReturnType();
70              if (rtype != void.class) {
```

```
71                  marshalValue(rtype, result, out);   72                  }
73          } catch (IOException ex) {
74          throw new MarshalException("error marshalling return", ex);
75          /*
76           * This throw is problematic because when it is caught below,
77           * we attempt to marshal it back to the client, but at this
78           * point, a "normal return" has already been indicated,
79           * so marshalling an exception will corrupt the stream.
80           * This was the case with skeletons as well; there is no
81           * immediately obvious solution without a protocol change.
82           */
83          }
84      } catch (Throwable e) {
85          logCallException(e);
86
87          ObjectOutput out = call.getResultStream(false);
88          if (e instanceof Error) {
89              e = new ServerError(
90                  "Error occurred in server thread", (Error) e);
91          } else if (e instanceof RemoteException) {
92              e = new ServerException(
93                  "RemoteException occurred in server thread",
94                  (Exception) e);
95          }
96          if (suppressStackTraces) {
97              clearStackTraces(e);
98          }
99          out.writeObject(e);
100     } finally {
101         call.releaseInputStream(); // in case skeleton doesn't
102         call.releaseOutputStream();
103     }
104 }
105
```



dispatch方法处理接收的请求，先从输入流中读取方法的编号来获得方法名称：op = in.readLong()，然后读取方法的所有参数：params[i] =**unmarshalValue**(types[i], in)，接着就可以执行方法调用了：result = method.**invoke**(obj, params)，最后把方法执行结果写入到输出流中：**marshalValue**(rtype, result, out)。一个方法调用就执行完成了。

2. 客户端发起调用请求

Client端请求一个远程方法调用时，先建立连接：**Connection conn** = **ref.getChannel().newConnection()**，然后发送方法参数： marshalValue(types[i], params[i], out)，再发送执行方法请求：call.**executeCall**()，最后得到方法的执行结果：Object returnValue = unmarshalValue(rtype, in)，并关闭接：ref.getChannel().free(conn, true)。

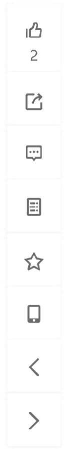
```
1   public Object invoke(Remote obj, java.lang.reflect.Method method, Object[] params, long opnum) throws Exception {
2       if (clientRefLog.isLoggable(Log.VERBOSE)) {
3           clientRefLog.log(Log.VERBOSE, "method: " + method);
4       }
5       if (clientCallLog.isLoggable(Log.VERBOSE)) {
6           logClientCall(obj, method);
7       }
8       Connection conn = ref.getChannel().newConnection();
9       RemoteCall call = null;
10      boolean reuse = true;
11      /* If the call connection is "reused" early, remember not to
12       * reuse again.
13       */
14      boolean alreadyFreed = false;
15      try {
16          if (clientRefLog.isLoggable(Log.VERBOSE)) {
17          clientRefLog.log(Log.VERBOSE, "opnum = " + opnum);
18          }
19          // create call context
20          call = new StreamRemoteCall(conn, ref.getObjID(), -1, opnum);
21          // marshal parameters
22          try {
23              ObjectOutput out = call.getOutputStream();
24              marshalCustomCallData(out);
25              Class[] types = method.getParameterTypes();
26              for (int i = 0; i < types.length; i++) {
27                  marshalValue(types[i], params[i], out);
28              }
29          } catch (IOException e) {
30              clientRefLog.log(Log.BRIEF,
31                  "IOException marshalling arguments: ", e);
32              throw new MarshalException("error marshalling arguments", e);
33          }
34          // unmarshal return
```

```
35          call.executeCall();
36          try {
37              Class rtype = method.getReturnType();
38              if (rtype == void.class)
39                  return null;
40              ObjectInput in = call.getInputStream();
41
42              /* StreamRemoteCall.done() does not actually make use
43               * of conn, therefore it is safe to reuse this
44               * connection before the dirty call is sent for
45               * registered refs.
46               */
47              Object returnValue = unmarshalValue(rtype, in);
48              /* we are freeing the connection now, do not free
49               * again or reuse.
50               */
51              alreadyFreed = true;
52              /* if we got to this point, reuse must have been true. */
53              clientRefLog.log(Log.BRIEF, "free connection (reuse = true)");
54              /* Free the call's connection early. */
55              ref.getChannel().free(conn, true);
56              return returnValue;
57
58          } catch (IOException e) {
59              clientRefLog.log(Log.BRIEF,
60                       "IOException unmarshalling return: ", e);
61              throw new UnmarshalException("error unmarshalling return", e);
62          } catch (ClassNotFoundException e) {
63              clientRefLog.log(Log.BRIEF,
64                  "ClassNotFoundException unmarshalling return: ", e);
65              throw new UnmarshalException("error unmarshalling return", e);
66          } finally {
67              try {
68                  call.done();
69              } catch (IOException e) {
70                  /* WARNING: If the conn has been reused early,
71                   * then it is too late to recover from thrown
72                   * IOExceptions caught here. This code is relying
73                   * on StreamRemoteCall.done() not actually
74                   * throwing IOExceptions.
75                   */
76                  reuse = false;
77              }
78          }
79      } catch (RuntimeException e) {
80          /*
81           * Need to distinguish between client (generated by the
82           * invoke method itself) and server RuntimeExceptions.
83           * Client side RuntimeExceptions are likely to have
84           * corrupted the call connection and those from the server
85           * are not likely to have done so.  If the exception came
86           * from the server the call connection should be reused.
87           */
88          if ((call == null) ||
89          (((StreamRemoteCall) call).getServerException() != e)) {
90              reuse = false;
91          }
92          throw e;
93      } catch (RemoteException e) {
94          /*
95           * Some failure during call; assume connection cannot
96           * be reused.  Must assume failure even if ServerException
97           * or ServerError occurs since these failures can happen
98           * during parameter deserialization which would leave
99           * the connection in a corrupted state.
100          */
101         reuse = false;
102         throw e;
103     } catch (Error e) {
104         /* If errors occurred, the connection is most likely not
105              *  reusable.
106          */
107         reuse = false;
108         throw e;
109     } finally {
110         /* alreadyFreed ensures that we do not log a reuse that
111          * may have already happened.
112          */
113         if (!alreadyFreed) {
114         if (clientRefLog.isLoggable(Log.BRIEF)) {
```

2

举报

```
115            clientRefLog.log(Log.BRIEF, "free connection (reuse = " +
116                reuse + ")");
117        }
118        ref.getChannel().free(conn, reuse);
119        }
120    }
121 }
```



3. I/O模型

现在的rmi实现采用的io是bio，并没有采用jdk1.4提供的nio功能。JDK8版本及以前版本，RMI采用的IO模型是**BIO**

4. 线程模型

在JDK1.5及以前版本中，RMI每接收一个远程方法调用就生成一个单独的线程来处理这个请求，请求处理完成后，这个线程就会释放：

```
1 Thread t = (Thread)java.security.AccessController.doPrivileged (
2                 new NewThreadAction(new ConnectionHandler(socket,clientHost),
3                 "TCP Connection(" + ++ threadNum +  ")-" + clientHost, true, true));
```

在JDK1.6之后，RMI使用**线程池**来处理新接收的远程方法调用请求-**ThreadPoolExecutor**。

在JDK1.6中，RMI提供了可配置的线程池参数属性：
sun.rmi.transport.tcp.maxConnectionThread - 线程池中的最大线程数量
sun.rmi.transport.tcp.threadKeepAliveTime - 线程池中空闲的线程存活时间

5. Marshell和unMarshell

从客户端发送数据，到拿到service实例的过程，其实就是 **对象的序列号–>网络传输–>反序列化**的过程。

marshal 和 **unmarshal**的意思，其实就是 **serialize**和 **unserialize**的意思

```
1 protected static void marshalValue(Class type, Object value, ObjectOutput out)  throws IOException {
2     if (type.isPrimitive()) {
3         if (type == int.class) {
4         out.writeInt(((Integer) value).intValue());
5         } else if (type == boolean.class) {
6         out.writeBoolean(((Boolean) value).booleanValue());
7         } else if (type == byte.class) {
8         out.writeByte(((Byte) value).byteValue());
9         } else if (type == char.class) {
10        out.writeChar(((Character) value).charValue());
11        } else if (type == short.class) {
12        out.writeShort(((Short) value).shortValue());
13        } else if (type == long.class) {
14        out.writeLong(((Long) value).longValue());
15        } else if (type == float.class) {
16        out.writeFloat(((Float) value).floatValue());
17        } else if (type == double.class) {
18        out.writeDouble(((Double) value).doubleValue());
19        } else {
20        throw new Error("Unrecognized primitive type: " + type);
21        }
22    } else {
23        out.writeObject(value);
24    }
25 }
26 rotected static Object unmarshalValue(Class type, ObjectInput in) throws IOException, ClassNotFoundException {
27    if (type.isPrimitive()) {
28        if (type == int.class) {
29        return new Integer(in.readInt());
30        } else if (type == boolean.class) {
31        return new Boolean(in.readBoolean());
32        } else if (type == byte.class) {
33        return new Byte(in.readByte());
34        } else if (type == char.class) {
35        return new Character(in.readChar());
36        } else if (type == short.class) {
37        return new Short(in.readShort());
38        } else if (type == long.class) {
```

```
39            return new Long(in.readLong());
40            } else if (type == float.class) {
41            return new Float(in.readFloat());
42            } else if (type == double.class) {
43            return new Double(in.readDouble());
44            } else {
45            throw new Error("Unrecognized primitive type: " + type);
46            }
47        } else {
48            return in.readObject();
49        }
50    }
```

6. 远程对象

实现类继承**UnicastRemoteObject**时，lookup出来的对象类型是**$Proxy0**，而不继承UnicastRemoteObject的类（对象类型是com.guojje.Hello）。
我们把继承**UnicastRemoteObject**类的对象叫做**远程对象**，我们lookup出来的对象，只是**该远程对象的存根(Stub)对象**,而远程对象永远在服务端。客户端每一次的方法调用，**最后都是仅有的那一个远程对象的方法调用。**
没有继承UnicastRemoteObject类的对象，同样可以bind到Registry，但lookup出来了对象却是**远程对象经过序列化**，然后到客户端**反序列化出来的新的对象**，后续的方法调用**与远程对象再无关系。**

点赞 2 收藏 分享 ...

Zenhobby
发布了17 篇原创文章 · 获赞 174 · 访问量 56万+
他的留言板 关注

西柏思别墅电梯
零底坑 | 小尺寸 了解更多

想对作者说点什么

浅谈RMI 阅读数 3404
目录1.RMI简介1.1实现原理2.实现流程3.程序结构 1.RMI简介RMI用于不同虚拟机之间的通信，这些虚... 博文 来自： wubinghai的...

举报

【java基础】RMI 阅读数 1434
提取总结自从懵逼到恍然大悟之Java中RMI的使用及Java RMI详解RMI （Remote Method Invocatio... 博文 来自： 星辰光年

RMI和RPC比较 阅读数 885

大家应该都是知道，节点通信是分布式架构的核心之一，那我们来理解一下远程方法调用的相关的RMI... 博文 来自： 段恩刚--Baron

2

轻松理解RPC及RMI 阅读数 1037

温馨提示：本篇完整代码已上传至https://github.com/Monkey-yc/RMI，欢迎访问！RPC（Remote ... 博文 来自： Monkey-yc

RMI一步一步来 阅读数 816

RMI，远程方法调用（Remote Method Invocation）是Enterprise JavaBeans的支柱，是建立分布... 博文 来自： 薛正华的专栏

RMI 使用简单教程 阅读数 353

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 Invocatio... 博文 来自： W.J.H.

java RMI原理详解 阅读数 1万+

定义RMI（Remote Method Invocation）为远程方法调用，是允许运行在一个Java虚拟机的对象调用... 博文 来自： xinghun_4的...

Java RMI详解 阅读数 241

RMI:远程方法调用(Remote Method Invocation)。能够让在某个java虚拟机上的对象像调用本地对象... 博文 来自： weixin_30687...

（二）RMI详解 阅读数 865

1.RMI的概述: RMI(remote method invocation)，可以认为是RPC的java版本 RMI使用的是JRMP（J... 博文 来自： 吾生也有涯，...

深究Java中的RMI底层原理 阅读数 1万+

前言：随着一个系统被用户认可，业务量、请求量不断上升，那么单机系统必然就无法满足了，于是系... 博文 来自： Coder Wu的博...

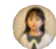
RPC 通信和 RMI 区别 阅读数 745

RPC（Remote Procedure Call Protocol）远程过程调用协议，通过网络从远程计算机上请求调用某种... 博文 来自： 张花生的博客

行走的技术员 6篇文章 关注 排名:千里之外

没有大海的星辰没有灵魂 526篇文章 关注 排名:901

段恩刚--Baron 124篇文章 关注 排名:千里之外

 不蛋定 60篇文章 关注 排名:千里之外

（转）用c++实现java的rmi服务 04-01

NULL 博文链接：https://eric-gcm.iteye.com/blog/1420827 下载

RPC和RMI 阅读数 248

1、RPC（远程过程调用）简介 RPC（Remote Procedure Call Protocol）——远程过程调用协议，它... 博文 来自： JinXYan的博客

RMI是什么 阅读数 531

RMI，远程方法调用（Remote Method Invocation）是Enterprise JavaBeans的支柱，是建立分布式J... 博文 来自： jiangxindu1的...

从懵逼到恍然大悟之Java中RMI的使用 阅读数 5万+

此处讲的是Java中的RMI，而不是通用意义上的RMI，关于通用的RMI可以参考分布式之RPC的协议以... 博文 来自： lmy86263的博客

【Consul】Consul实践指导-RPC机制 阅读数 3511

Consul agent提供了完整的RPC机制——用于agent编程。RPC机制同CLI一样，但是可以被其他应用... 博文 来自： 土著部落

开源项目之C++远程方法调用框架 RMI for C++ 阅读数 6782

RMI for C++ 是一个专为 C++ 语言提供的远程方法调用框架，与 CORBA 不同的是，CORBA 适合不... 博文 来自： banketree

RMI介绍与使用 阅读数 6172

今天在这边介绍一下Java基础中的rmi使用。其实rmi有什么样的使用场景呢？它跟webservice有什么... 博文 来自： 飞血泪无痕的...

RMI：利用JDK中的Remote实现远程方法调用 阅读数 761

Java RMI：即Java远程方法调用，是针对Java语言的一种特殊RPC调用，一种用于实现远程过程调用... 博文 来自： Apeopl的博客

获取真实IP地址 阅读数 59

转载：http://blog.sina.com.cn/s/blog_407a68fc01000ai7.html在JSP里，获取客户端的IP地址的方... 博文 来自： dingjun1

浅析 Stubs/Skeletons 机制 与 RMI 调用 阅读数 2123

存根类是一个类，它实现了一个接口，但是实现后的每个方法都是空的。它的作用是：如果一个接口有... 博文 来自： 稚枭天卓

举报

Java RMI 阅读数 389

一. Java RMI概述 RMI远程方法调用是计算机之间通过网络实现对象调用的一种通讯机制。使用这种机... 博文 来自： 扶我起来。。。

RMI 阅读数 164

RMIRMI(Remote Method Invocation，远程方法调用)是用Java在JDK1.2中实现的，它大大增强了Jav... 博文 来自： weixin_41924...

Java RMI和Spring RMI的实现 阅读数 760

初学RMI时，在网上搜java RMI的使用方法，很容易就搜到了，也很简单，整个过程也就是以下几个步... 博文 来自： ibigboy

什么是RMI 阅读数 694

大家好，我是IT修真院武汉分院第15期的学员，一枚正直纯洁善良的JAVA程序员。今天给大家分享一... 博文 来自： qq_42218123...

什么是RMI，为什么要使用RMI框架? 阅读数 228

这里是修真院后端小课堂，每篇分享文从八个方面深度解析后端知识/技能，本篇分享的是：【什么是R... 博文 来自： learning_java...

Java RMI （Remote Method Invoke 远程方法调用） 阅读数 2141

import java.rmi.Remote;import java.rmi.RemoteException;public interface IService extends Re... 博文 来自： liaoguanghui0...

RMI 浅析 阅读数 1993

本文目录1 RMI简介2 实现步骤：Server端3 实现步骤：Client端4 执行步骤1 RMI简介RMI是Remote ... 博文 来自： 崔显龙的博客

RMI的简单使用? 阅读数 26

这里是修真院后端小课堂，每篇分享文从八个方面深度解析后端知识/技能，本篇分享的是：【RMI的简... 博文 来自： learning_java...

RPC和RMI的区别 07-13

最近学习了一些系统交互的名称RPC和RMI，也看了好多书籍。 分布式系统一书中对RPC和RMI的异同描述... 论坛

警告 [RMI TCP Connection(2)-127.0.0.1] 10-05

在idea中运行web项目的时候出现这个是什么情况? 警告 [RMI TCP Connection(2)-127.0.0.1] 问答

RMI异常，java.rmi.ConnectException: Connection refused to host 08-08

[code="java"] String name = "Compute"; HelloWorldCompute engine = new HelloWorldEngine(); H... 问答

让C++也支持RMI 阅读数 2385

http://www.vckbase.com/document/viewdoc/?id=1846 作者：王树栋下载源代码摘要RMI(Remot... 博文 来自： 杨德龙的专栏

C++远程方法调用框架 RMI for C++ 附加boost库 09-12

C++远程方法调用框架 RMI for C++ 附加boost库 下载

rmi使用 阅读数 44

RmiClientpackage com.lv.rmi;import java.rmi.Naming;import java.util.List;/** * Created by lvyan... 博文 来自： fengyingsuixu...

RMI如何停止服务器。 12-29

不知道RMI服务器端程序开启之后怎么停止，我想监听服务器停止事件该如何写啊。比如在服务器停止之前... 论坛

Dubbo采用RMI协议时怎样设置servicePort(服务端口) 11-12

如果不用dubbo，直接用spring暴露RMI的时候需要配置2个端口。其中一个是registerPort，即注册端口； ... 问答

爬虫福利二 之 妹子图网MM批量下载 阅读数 21万+

爬虫福利一：27报网MM批量下载 点击看了本文，相信大家对爬虫一定会产生强烈的兴趣，激励自己去... 博文 来自： Nick.Peng 的...

Java学习的正确打开方式 阅读数 29万+

在博主认为，对于入门级学习java的最佳学习方法莫过于视频+博客+书籍+总结，前三者博主将淋漓尽... 博文 来自： 程序员宜春的...

程序员必须掌握的核心算法有哪些? 阅读数 41万+

由于我之前一直强调数据结构以及算法学习的重要性，所以就有一些读者经常问我，数据结构与算法应... 博文 来自： 帅地

python json java mysql pycharm android linux json格式 c#影院售票系统有哪些 c#鼠标相对窗体的坐标 c#如何快速的求和 c# 界面设计 c#窗口隐藏 c# 动态注入il 测试c#程序的软件 加入队列c# c# 模型验证取消 c# 小数点后保留4位

©2019 CSDN 皮肤主题: 编程工作室 设计师: CSDN官方博客

Zenhobby

TA的个人主页 >

原创	粉丝	获赞	评论	访问
17	372	174	50	56万+

举报

等级: 博客 6　　周排名: 9万+
积分: 5300　　总排名: 8779
勋章:

关注　私信

2

最新文章

我在ThoughtWorks学软开（一）敏捷之于开发如同蜜糖，甜到发腻齁到忧伤
Node.js从入门到实战（八）Solr的层级
Node.js从入门到实战（七）Solr查询规则总结
DevOps入门（三）自动化构建工具Gradle
DevOps入门（二）包管理工具yarn与npm对比

分类专栏

我在ThoughtWorks...	1篇
MFC	40篇
STL	6篇
Android	2篇
C++	64篇

展开

归档

2018年10月	1篇
2018年2月	2篇
2018年1月	22篇
2017年12月	2篇
2017年11月	23篇
2017年10月	24篇
2017年9月	22篇
2017年8月	12篇

展开

热门文章

MQ入门总结（一）消息队列概念和使用场景
阅读数 48932
RPC入门总结（一）RPC定义和原理
阅读数 47104
MySql从一窍不通到入门（五）Sharding：分表、分库、分片和分区
阅读数 33656
算法概念：大O表示法/小o表示法/Ω/Θ
阅读数 17704
MFC中的文件读写方法总结
阅读数 16366

举报

最新评论

C++中引用（&）的用法和...
small21：mark,经常看经常忘

五大算法思想：分治、动态规划、贪心...
Chloe_：这篇总结简直是宝藏，感谢~

Java 数据库连接池的实现
qq_22038259：[reply]huahangwanghao[/reply] hia,hia

RPC入门总结（一）RPC定义和原理
qq_41643060：请问一下,客户端编码后把请求发送给服务端，服务端解码后处理完返回数据给 ...

MySql从一窍不通到入门（五）S...
Vibugs：虽然是转载的，但是文章质量很高。谢谢作者

2

SONY
WHAT HI·FI

WF-1000XM3
July 2019
"THE BEST TRUE WIRELE
HEADPHONES YOU CAN E
LEARN MORE

QQ客服 kefu@csdn.net
客服论坛 400-660-0108
工作时间 8:30-22:00

关于我们 招聘 广告服务 网站地图

京ICP备19004658号 经营性网站备案信息
公安备案号 11010502030143
©1999-2020 北京创新乐知网络技术有限公司 网络110报警服务
北京互联网违法和不良信息举报中心
中国互联网举报中心 家长监护 版权申诉

举报