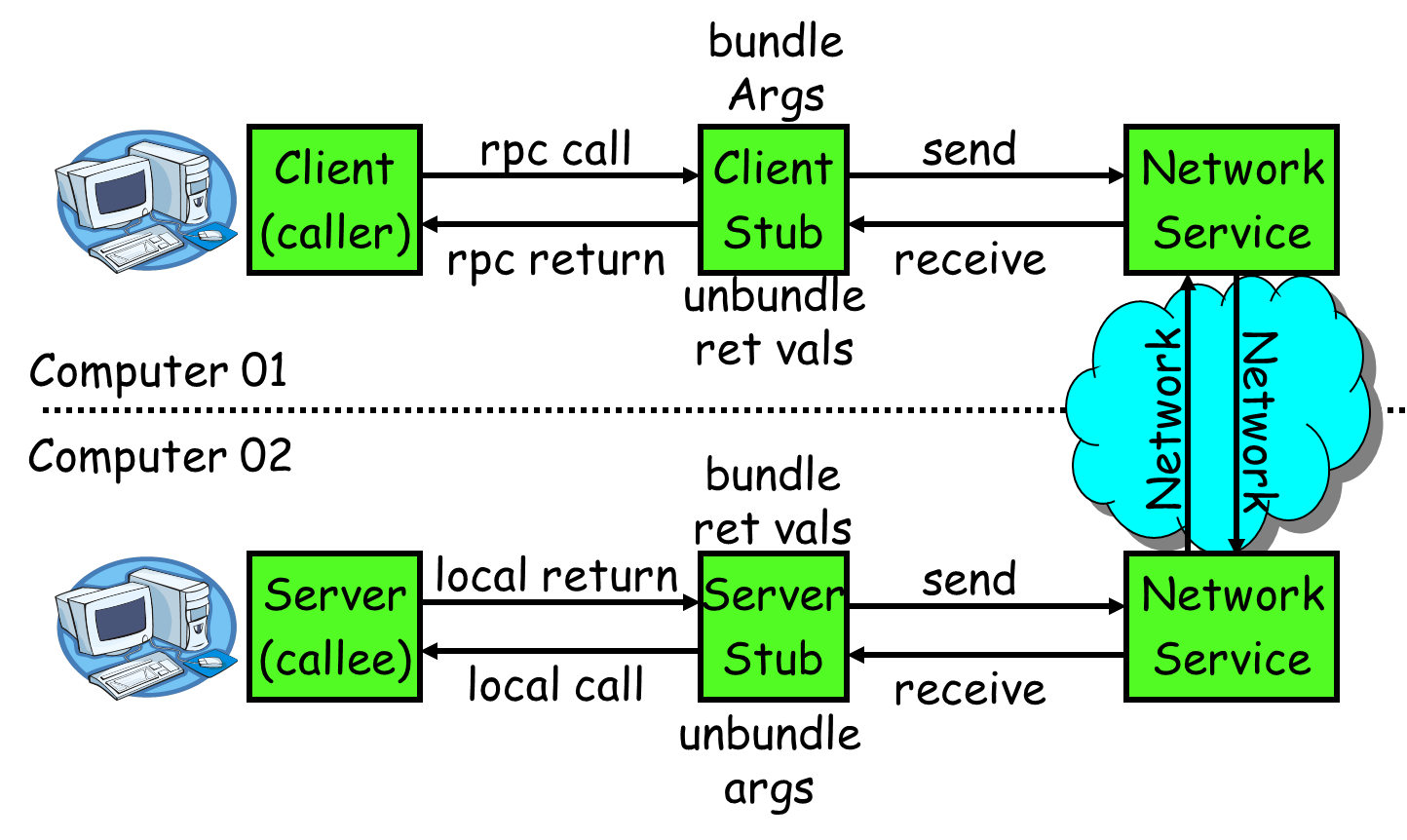
参考：<http://www.cnblogs.com/codingexperience/p/5930752.html>

**一、RPC简介**

RPC，全称为Remote Procedure Call，即远程过程调用，它是一个计算机通信协议。它允许像调用本地服务一样调用远程服务。它可以有不同的实现方式。如RMI(远程方法调用，me：是java的rpc，这个只能在java下使用)、Hessian、Http invoker等。另外，RPC是与语言无关的。

**RPC示意图**



如上图所示，假设Computer1在调用sayHi()方法，对于Computer1而言调用sayHi()方法就像调用本地方法一样，调用 –>返回。但从后续调用可以看出Computer1调用的是Computer2中的sayHi()方法，RPC屏蔽了底层的实现细节，让调用者无需关注网络通信，数据传输等细节。

**二、RPC框架的实现**

    上面介绍了RPC的核心原理：RPC能够让本地应用简单、高效地调用服务器中的过程（服务）。它主要应用在分布式系统。如Hadoop中的IPC组件。但怎样实现一个RPC框架呢？

从下面几个方面思考，仅供参考：

1.通信模型：假设通信的为A机器与B机器，A与B之间有通信模型，在Java中一般基于BIO或NIO（me：异步通信）；。

2.过程（服务）定位：使用给定的通信方式，与确定IP与端口及方法名称确定具体的过程或方法；

3.远程代理对象：本地调用的方法(服务)其实是远程方法的本地代理，因此可能需要一个远程代理对象，对于Java而言，远程代理对象可以使用Java的动态对象（me：即多态，接口的实现类，需要继承自同一个接口，也可以在不同项目同一个包名下建立完全相同的接口）实现，封装了调用远程方法调用；

4.序列化，将对象名称、方法名称、参数等对象信息进行网络传输需要转换成二进制传输，这里可能需要不同的序列化技术方案。如:protobuf，Arvo等。

**三、Java实现RPC框架**

**1、实现技术方案**

     下面使用比较原始的方案实现RPC框架，采用Socket通信、动态代理与反射与Java原生的序列化。

**2、RPC框架架构**

RPC架构分为三部分：

1）服务提供者，运行在服务器端，提供服务接口定义与服务实现类。

2）服务中心，运行在服务器端，负责将本地服务发布成远程服务（register），管理远程服务，提供给服务消费者使用。

3）服务消费者，运行在客户端，通过远程代理对象调用远程服务。

**3、 具体实现**

服务提供者接口定义与实现，代码如下：

|  |
| --- |
| public interface HelloService {        String sayHi(String name);    } |

HelloServices接口实现类（me:定义在remote端）：

|  |
| --- |
| public class HelloServiceImpl implements HelloService {        public String sayHi(String name) {          return "Hi, " + name;      }  } |

服务中心代码实现，代码如下：

|  |
| --- |
| public interface Server {      public void stop();        public void start() throws IOException;        public void register(Class serviceInterface, Class impl);        public boolean isRunning();        public int getPort();  } |

服务中心实现类：

|  |
| --- |
| public class ServiceCenter implements Server {      private static ExecutorService executor = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());        private static final HashMap<String, Class> serviceRegistry = new HashMap<String, Class>();        private static boolean isRunning = false;        private static int port;        public ServiceCenter(int port) {          this.port = port;      }        public void stop() {          isRunning = false;          executor.shutdown();      }        public void start() throws IOException {          ServerSocket server = new ServerSocket();          server.bind(new InetSocketAddress(port));          System.out.println("start server");          try {              while (true) {              // 1.监听客户端的TCP连接，接到TCP连接后将其封装成task，由线程池执行                  executor.execute(new ServiceTask(server.accept()));              }          } finally {              server.close();          }      }        public void register(Class serviceInterface, Class impl) {          serviceRegistry.put(serviceInterface.getName(), impl);      }        public boolean isRunning() {          return isRunning;      }        public int getPort() {          return port;      }        private static class ServiceTask implements Runnable {          Socket clent = null;            public ServiceTask(Socket client) {              this.clent = client;          }            public void run() {              ObjectInputStream input = null;              ObjectOutputStream output = null;              try {                  // 2.将客户端发送的码流反序列化成对象，反射调用服务实现者，获取执行结果                  input = new ObjectInputStream(clent.getInputStream());                  String serviceName = input.readUTF();                  String methodName = input.readUTF();                  Class<?>[] parameterTypes = (Class<?>[]) input.readObject();                  Object[] arguments = (Object[]) input.readObject();                  Class serviceClass = serviceRegistry.get(serviceName);                  if (serviceClass == null) {                      throw new ClassNotFoundException(serviceName + " not found");                  }                  Method method = serviceClass.getMethod(methodName, parameterTypes);                  Object result = method.invoke(serviceClass.newInstance(), arguments);                    // 3.将执行结果反序列化，通过socket发送给客户端                  output = new ObjectOutputStream(clent.getOutputStream());                  output.writeObject(result);              } catch (Exception e) {                  e.printStackTrace();              } finally {                  if (output != null) {                      try {                          output.close();                      } catch (IOException e) {                          e.printStackTrace();                      }                  }                  if (input != null) {                      try {                          input.close();                      } catch (IOException e) {                          e.printStackTrace();                      }                  }                  if (clent != null) {                      try {                          clent.close();                      } catch (IOException e) {                          e.printStackTrace();                      }                  }              }            }      }  } |

　客户端的远程代理对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public class RPCClient<T> {      public static <T> T getRemoteProxyObj(final Class<?> serviceInterface, final InetSocketAddress addr) {          // 1.将本地的接口调用转换成JDK的动态代理，在动态代理中实现接口的远程调用          return (T) Proxy.newProxyInstance(serviceInterface.getClassLoader(), new Class<?>[]{serviceInterface},                  new InvocationHandler() {                      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {                          Socket socket = null;                          ObjectOutputStream output = null;                          ObjectInputStream input = null;                          try {                              // 2.创建Socket客户端，根据指定地址连接远程服务提供者                              socket = new Socket();                              socket.connect(addr);                                // 3.将远程服务调用所需的接口类、方法名、参数列表等编码后发送给服务提供者                              output = new ObjectOutputStream(socket.getOutputStream());                              output.writeUTF(serviceInterface.getName());                              output.writeUTF(method.getName());                              output.writeObject(method.getParameterTypes());                              output.writeObject(args);                                // 4.同步阻塞等待服务器返回应答，获取应答后返回                              input = new ObjectInputStream(socket.getInputStream());                              return input.readObject();                          } finally {                              if (socket != null) socket.close();                              if (output != null) output.close();                              if (input != null) input.close();                          }                      }                  });      }  } |

最后为测试类：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class RPCTest {        public static void main(String[] args) throws IOException {          new Thread(new Runnable() {              public void run() {                  try {                      Server serviceServer = new ServiceCenter(8088);                      serviceServer.register(HelloService.class, HelloServiceImpl.class);                      serviceServer.start();                  } catch (IOException e) {                      e.printStackTrace();                  }              }          }).start();          HelloService service = RPCClient.getRemoteProxyObj(HelloService.class, new InetSocketAddress("localhost", 8088));          System.out.println(service.sayHi("test"));      }  } |

运行结果：

|  |  |
| --- | --- |
| 1  2  3 | regeist service HelloService  start server  Hi, test |

**四、总结**

      RPC本质为消息处理模型，RPC屏蔽了底层不同主机间的通信细节，让进程调用远程的服务就像是本地的服务一样。

**五、可以改进的地方**

     这里实现的简单RPC框架是使用Java语言开发，与Java语言高度耦合（me：为rmi），并且通信方式采用的Socket是基于BIO（me：block io即阻塞io）实现的，IO效率不高，还有Java原生的序列化机制占内存太多，运行效率也不高。可以考虑从下面几种方法改进。

1. 可以采用基于JSON数据传输的RPC框架；
2. 可以使用NIO或直接使用Netty替代BIO实现；
3. 使用开源的序列化机制，如Hadoop Avro与Google protobuf等；
4. 服务注册可以使用Zookeeper进行管理，能够让应用更加稳定。