分析 JBoss Remoting

----------*BlueDavy http://blog.bluedavy.cn 2008-5-4*

1 分布式应用概述

在 Java 领域中，分布式应用相对集中式应用而言，首先需要解决的一个非常明显的问题就是对象之间如何通讯，在集中式的应用中，当需要调用其他对象时，通常只用 new 相应的对象实例，或通过 IoC Container 注入所需要的对象，抑或通过 Factory 获取所需要的对象等，之后直接调用就行了，转化到分布式应用场景，则不同了，因为调用端或者说客户端只会有远程对象的接口，而没有实现，如何在客户端发起对远程对象的调用，远程对象又是如何接收请求的，这是分布式调用必须解决的问题。

要解决这个问题，首先得解决如何将字节流从一台机器传输到另外一台机器，为了实现这个，两台机器需要采用标准的传输协议将字节流从一台机器传输至另一台机器，目前可采用的传输协议有 tcp 和 udp 两种，其他 http、ftp、telnet 等都是基于 tcp 扩展出来的应用协议，而 DNS、TFTP 等协议则是基于 UDP 扩展出来的。**TCP** 是 Tranfer Control Protocol 的简称，是一种面向连接的保证可靠传输的协议，通过 TCP 协议传输，得到的是一个顺序的无差错的数据流。发送方和接收方的成对的两个 socket 之间必须建立连接，以便在 TCP 协议的基础上进行通信，当一个 socket（通常都是 server socket）等待建立连接时，另一个 socket 可以要求进行连接，一旦这两个 socket 连接起来，它们就可以进行双向数据传输，双方都可以进行发送或接收操作；**UDP** 是 User Datagram Protocol 的简称，是一种无连接的协议，每个数据报都是一个独立的信息，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。

那么在 java 中如何来实现网络通讯呢，java 提供了 java.net 包来实现网络通讯，其中 Socket、ServerSocket 可用于实现基于 tcp 协议的网络通讯，典型的基于 java.net 的 Socket 的网络通讯的程序是这么实现的：Server 端 Listen(监听)某个端口是否有连接请求，Client 端向 Server 端发出 Connect(连接)请求，Server 端向 Client 端发回 Accept（接受）消息，一个连接就建立起来了，Server 端和 Client 端之后就可通过 Send，Write 等方法与对方进行通讯了；采用 java.net 的 DatagramSocket、DatagramPacket 可用于实现基于 udp 协议的网络通讯。

可见基于 java.net 包可比较方便的实现分布式的通讯，但要实现对于远程对象的调用，则还需要进行一定的封装处理，java 本身也提供了像 rmi 这样的远程对象调用的协议，对于小型的分布式应用而言，不一定需要一个完整的分布式支撑框架或平台，可以简单的基于 socket、

rmi、jms、hessian 或 webservice 等方式直接实现远程的通讯或调用即可，但对于较为大型的分布式应用而言，则必须提升到框架级别来更好的解决分布式应用对比集中应用带来的更多的挑战，例如支持多种协议的远程调用、可靠高效的线程池调度、尽量减少应用变成分布式后的性能下降、远程调用带来的异常现象的处理、调用的多种方式的支持等等。

对比集中式应用，来看看分布式应用带来了些什么不同，第一个明显的问题自然是如何调用远程的对象，在 java 领域中最出名的莫过于 RMI、Hessian、Webservice 等等了，基于这些方式可实现从客户端调用远程对象的方法，这里衍生出来的就是调用远程对象的方式，例如同步、异步等；第二个问题则是变成分布式后带来的一个典型问题，就是远程对象是通过监听端口的方式来接收请求，为了避免每个请求阻塞了读，在接收到请求后，就需要通过启动新的线程的方式来执行对象的方法；第三个问题则是当应用变成分布式后，会带来一些在集中式应用中不存在的异常现象，例如网络断了、某方法执行时间过长导致线程被占用等现象，这要求在进行远程调用时能够具备一定的异常处理能力，如超时中断、自动重试等等机制，上述所描述的三个问题是分布式应用中比较明显的问题，而这也是一个远程调用框架应该提供的基础功能，当然，这三个问题其中还包含更多的细节问题，例如第一个问题中可以衍生出能否透明的进行远程调用、能否压缩流进行传输以提高传输速度等。

2 分析 JBoss-Remoting

Java 领域中的分布式框架比较的多，分析一个已有的远程调用框架无论是对于打算采用已有成果还是自己做分布式框架，都是很必要的事情，JBoss Remoting 是其中很好很强大的一个框架，在此来对 JBoss Remoting 进行深入的分析，看看 JBoss Remoting 是如何基于 java.net 提供的包去解决这些问题的，本文所分析的 JBoss Remoting 源码的版本为 2.2.2_SP2。

JBoss Remoting 支持多种协议方式调用远程对象，例如 socket 的方式、RMI 的方式、Http 的方式等等，在这里选择 Socket 方式远程调用进行分析。

2.1 Socket 方式远程调用

2.1.1 使用

首先来看看在 socket 这种最基本的方式下 JBoss Remoting 是如何实现远程对象调用的，例如客户端需要调用服务器端的 DefaultUserService 对象的 getById 方法，此 A 对象实现的接口为 UserService，基于 JBoss Remoting 的 socket 方式下比较典型的实现方式是这样的：

- **服务器端**

  String locatorURI=”socket://127.0.0.1:12345”;

TransporterServer server=TransporterServer.createTransporterServer(locatorURI,new DefaultUserService (),UserService.class.getName());

server.start();

- **客户端**

String locatorURI=”socket://127.0.0.1:12345”;

UserService service=(UserService)TransporterClient.createTransporterClient(locatorURI, UserService.class);

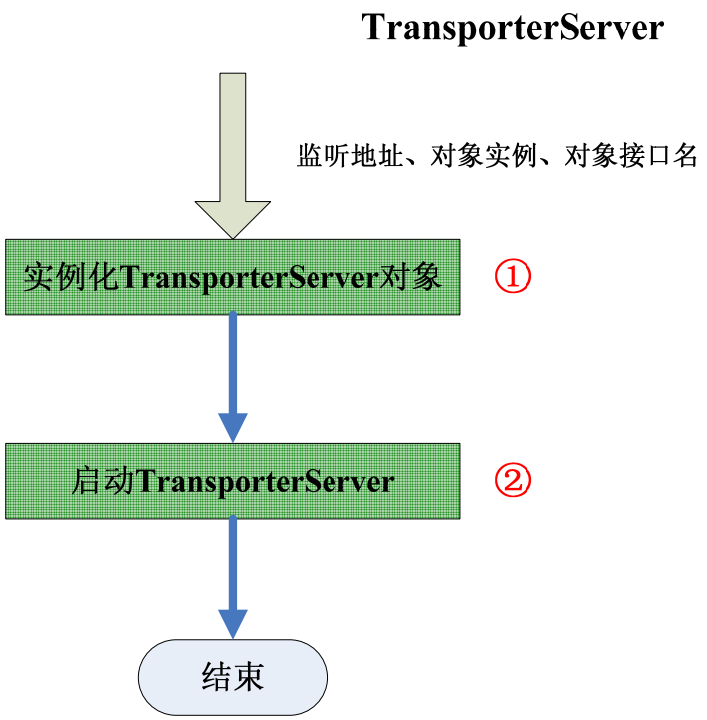
User user=service.getById(1);

可以看到，在 JBoss Remoting 的支持下，客户端可以透明的进行远程调用，就像仍然是在调用本地的对象一样。

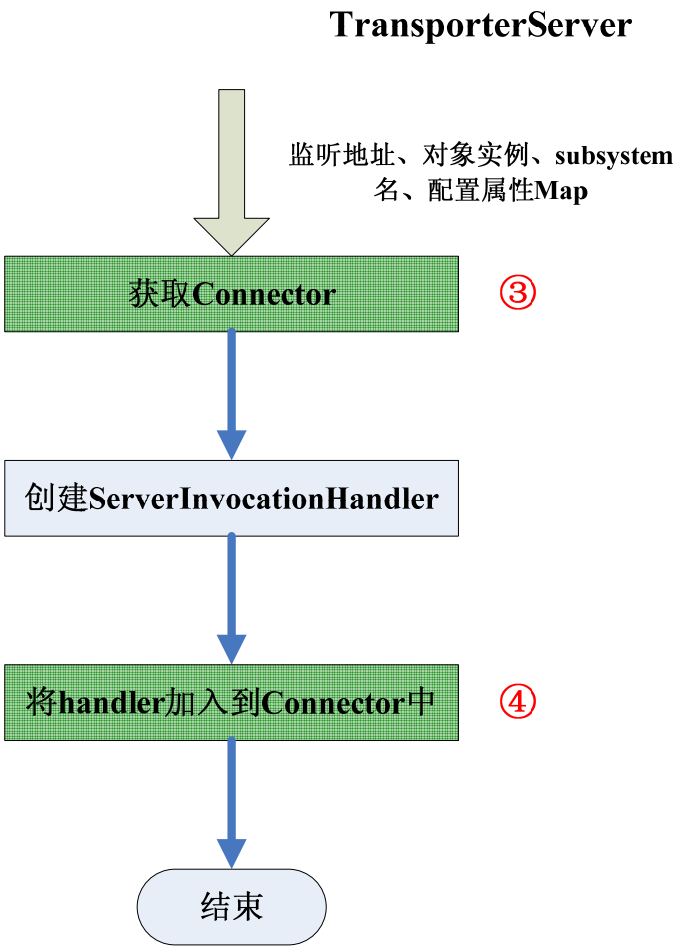
按照 java.net 包提供的 socket 编程方式，可以知道基于 socket 方式要实现远程对象的调用还是有些封装的事情要去做的，但其基于的原理必然也是 Socket 和 ServerSocket 这两个主要的类来实现了，具体来看看 JBoss Remoting 是如何进行封装的，具体做了些什么。

2.1.2 服务器端的启动过程

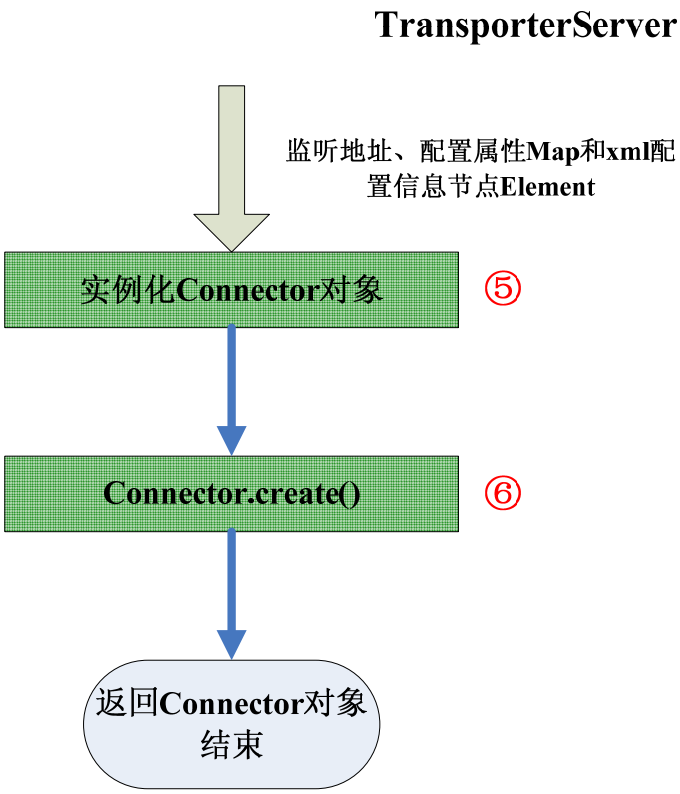
将服务器端的启动过程转化为流程图，如下所示：



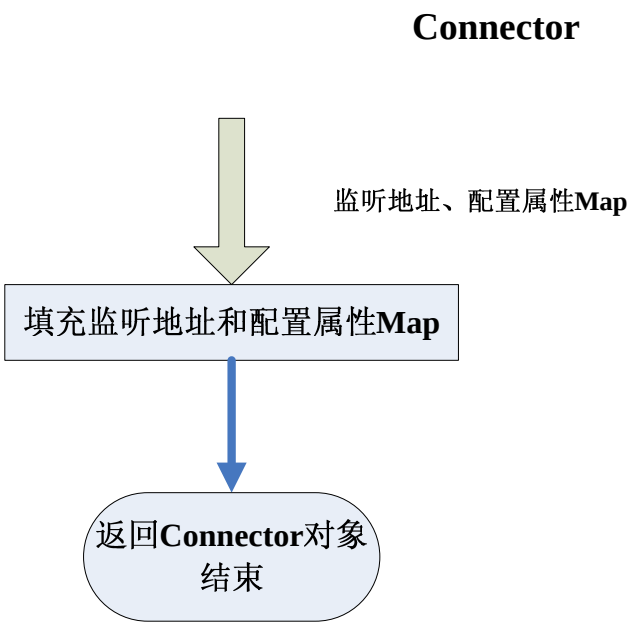
其中第①步的过程是这么实现的：



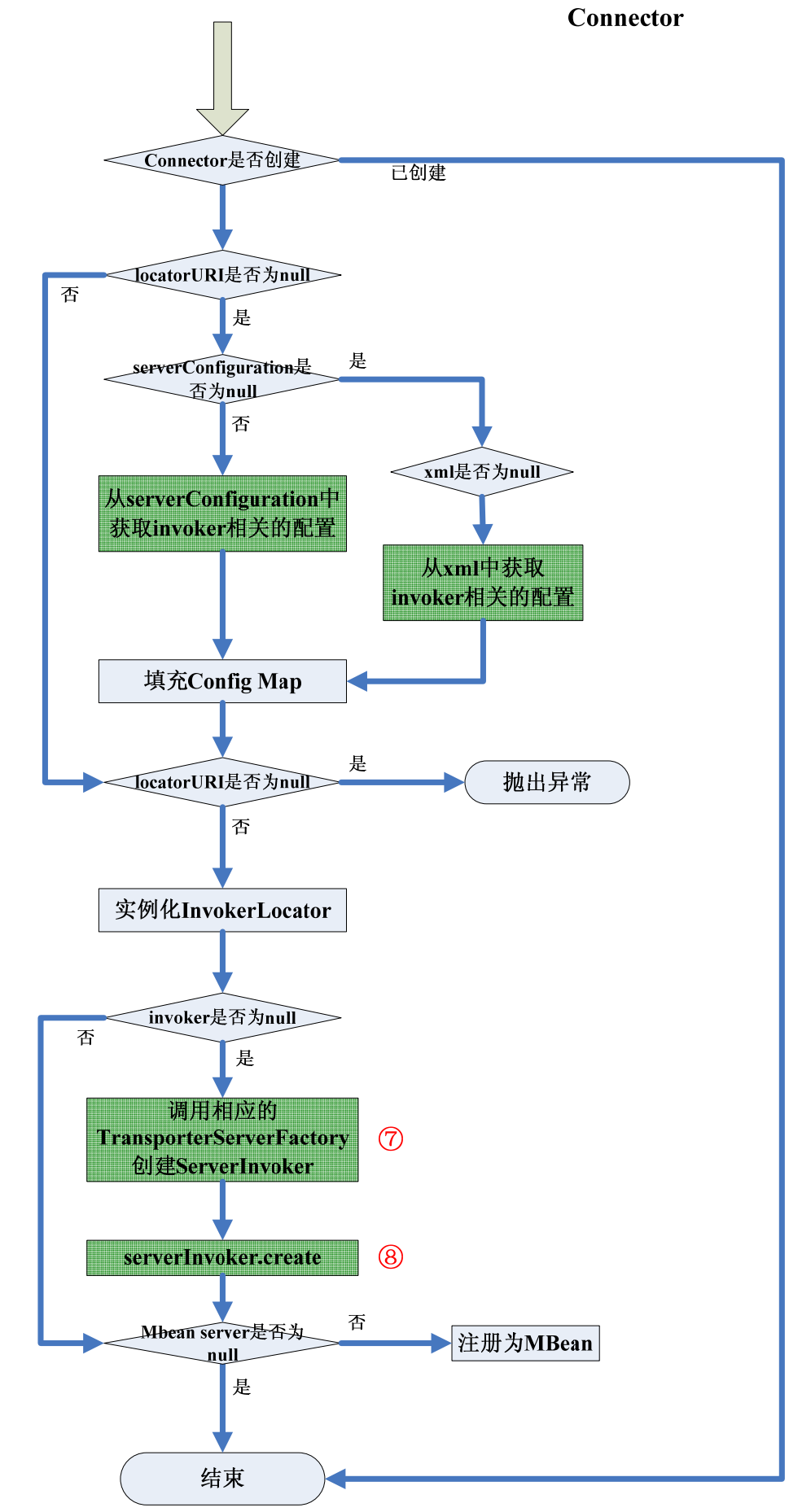
其中第③步的过程是这么实现的：



其中第⑤步的过程是这么实现的：



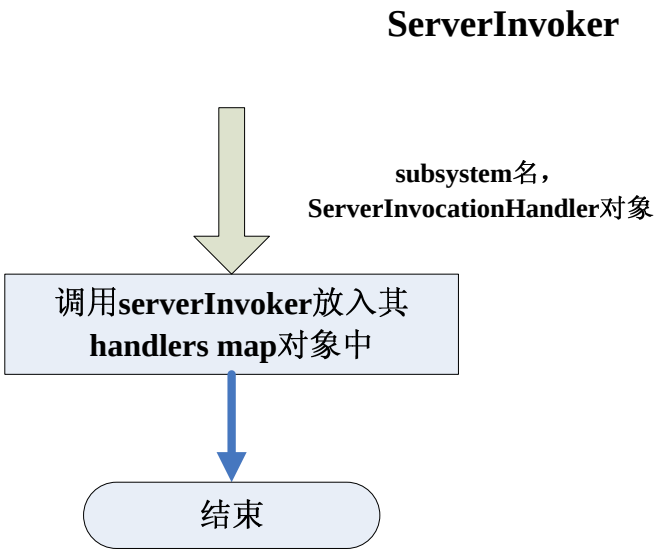
第⑥步的过程是这么实现的：


Connector
Connector是否创建
已创建
locatorURI是否为null
否
是
serverConfiguration是否为null
是
否
从serverConfiguration中获取invoker相关的配置
xml是否为null
从xml中获取invoker相关的配置
填充Config Map
locatorURI是否为null
是
抛出异常
否
实例化InvokerLocator
invoker是否为null
否
是
调用相应的TransporterServerFactory创建ServerInvoker
⑦
serverInvoker.create
⑧
Mbean server是否为null
否
注册为MBean
是
结束

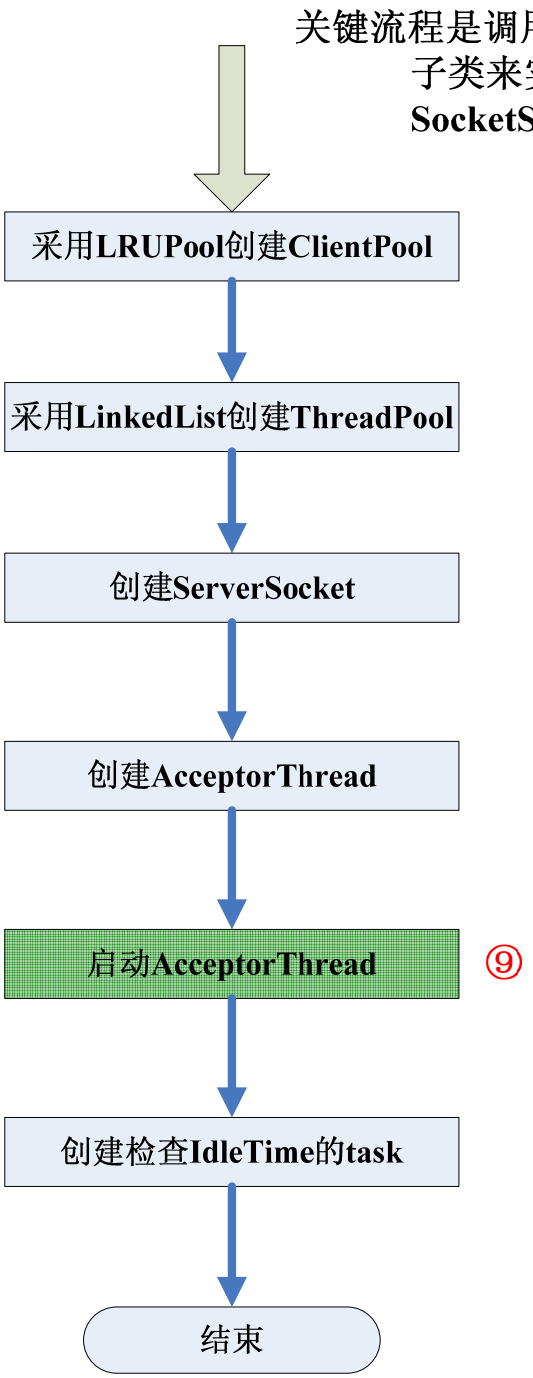
上图中的第⑦步是 JBoss Remoting 中采用的命名约定的一个典型例子，在这里 jboss remoting 会按照 org.jboss.remoting.transport.协议名.TransporterServerFactory 的方式去调用相应的 TransporterServerFactory，所以从代码的 package 上可以很清楚的看出目前 jboss remoting 官方版本所支持的协议有：bisocket、coyote、http、https、local、multiplex、rmi、servlet、socket、sslbisocket、sslmultiplex、sslrmi、sslservlet、sslsocket。

在第⑧步中将调用 JBoss Remoting 提供的 ServerInvoker 的 create 方法，在此方法中所做的事情主要是把配置的一些属性进行了读取和存储，另外就是根据配置创建 ServerSocketFactory。

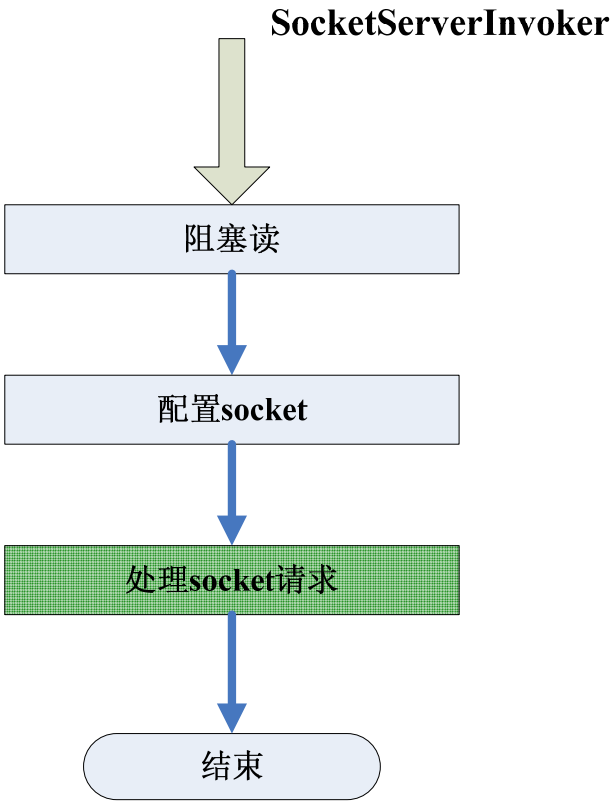
经过上面的过程，第③步的工作完成，来看看第④步的时候做了什么：



经过上面这些过程，第①步工作完成，第②步工作是通过调用各协议自己创建的 ServerInvoker 子类的 start 方法来完成的，因此在这里也就先根据例子来看看 socket 方式下的的 start 方法：



第⑨步执行后所做的具体事情如下：

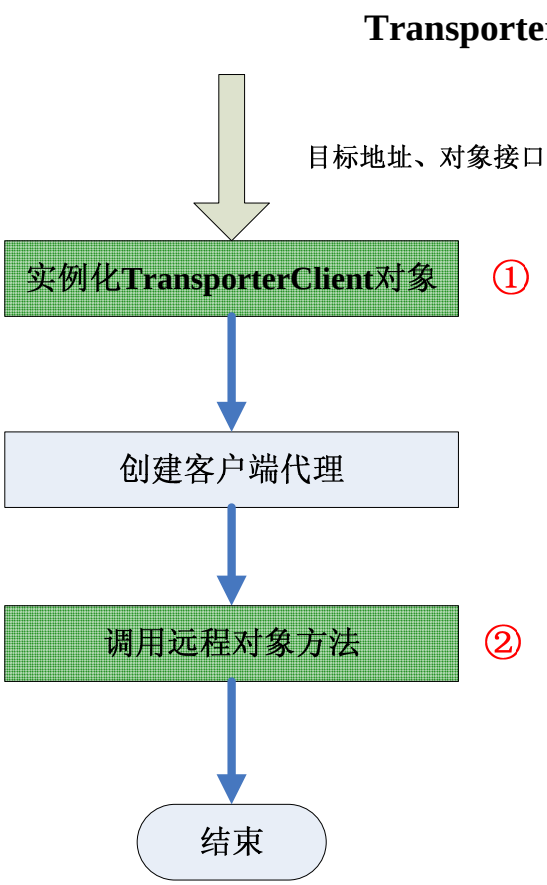


这个过程在之后分析客户端的调用过程时再来详细的进行讲解。

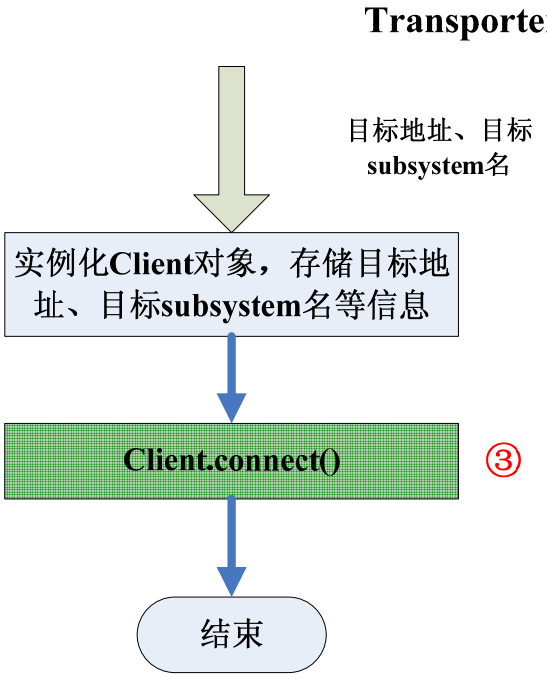
经过上面的过程，服务器端的启动就完成了，从上面的分析中可以看出在服务器端启动时 JBoss Remoting 主要是通过 Connector、TransporterServerFactory 和 ServerInvoker 这几个类来完成的，并通过 TransporterServerFactory 的命名约定来提供了协议实现的扩展方式，从这里我们看到的是一个可支持多种协议的分布式框架的模型（不论是任何协议，都采用同样的方式来进行使用，需要改变的仅仅是传递给它的 URI 头），以及 JBoss Remoting 如何基于 java.net 的 ServerSocket 提供 socket 方式的远程调用的支持，具体的调用的实现过程得根据下面的客户端调用过程的分析来做详细的解释。

2.1.3 客户端的调用过程

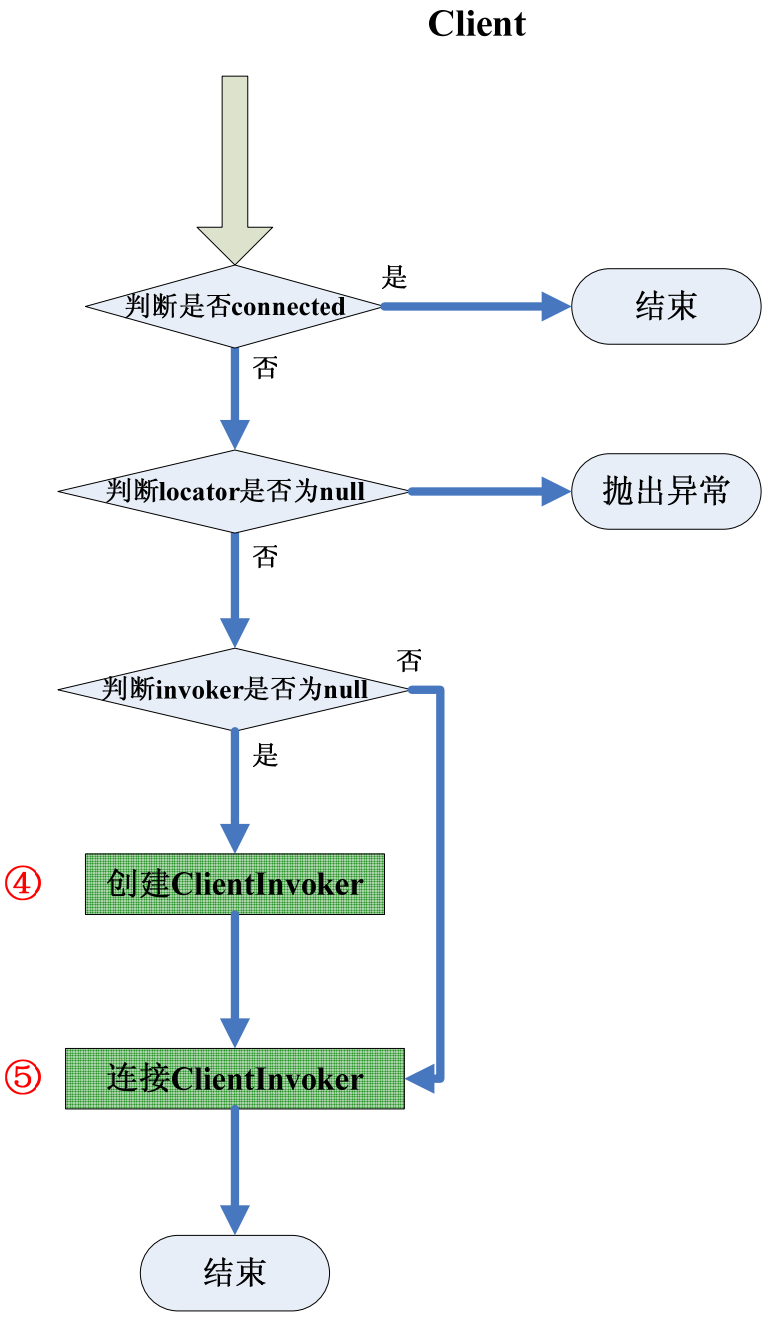
客户端的调用过程转化为流程后图示如下：



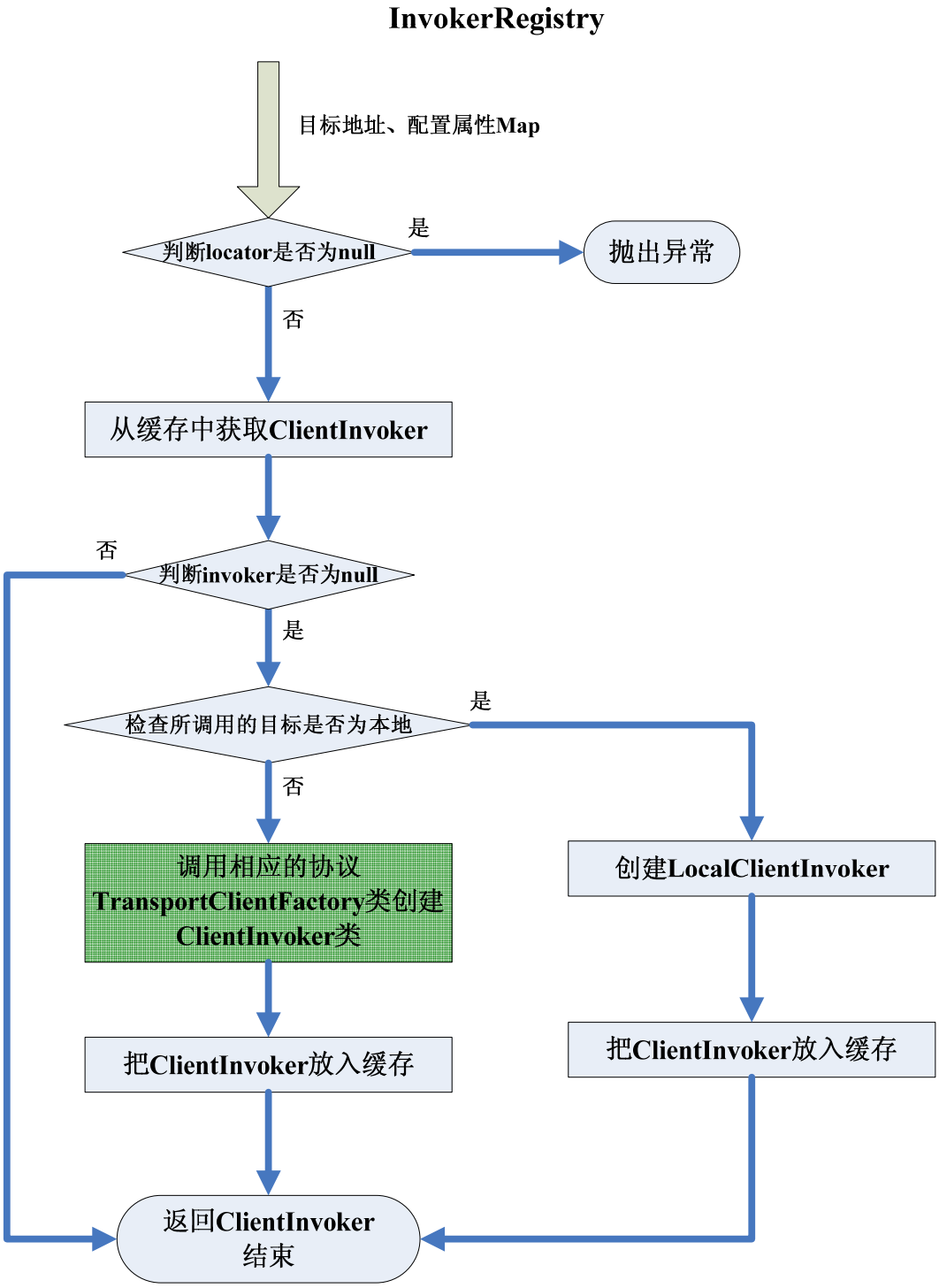
JBoss Remoting 通过创建代理的方式透明实现对远程对象的调用，第①步的具体实现过程如下所示：



第③步的具体实现过程如下所示：

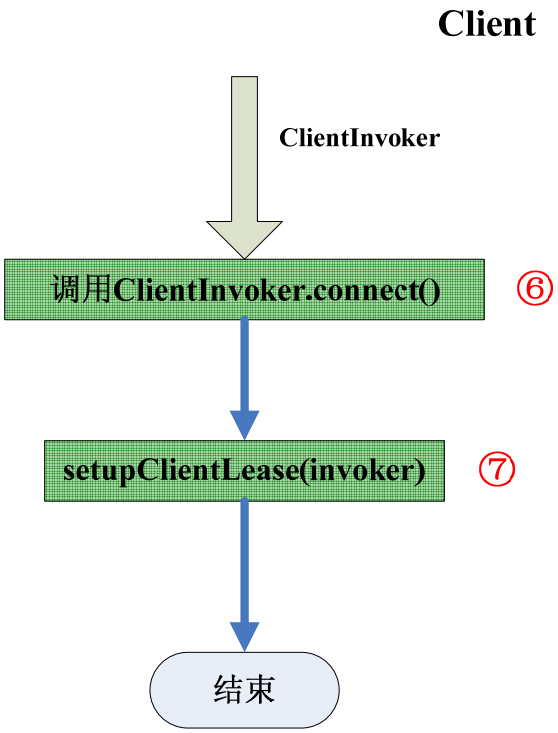


第④步的具体实现过程如下所示：

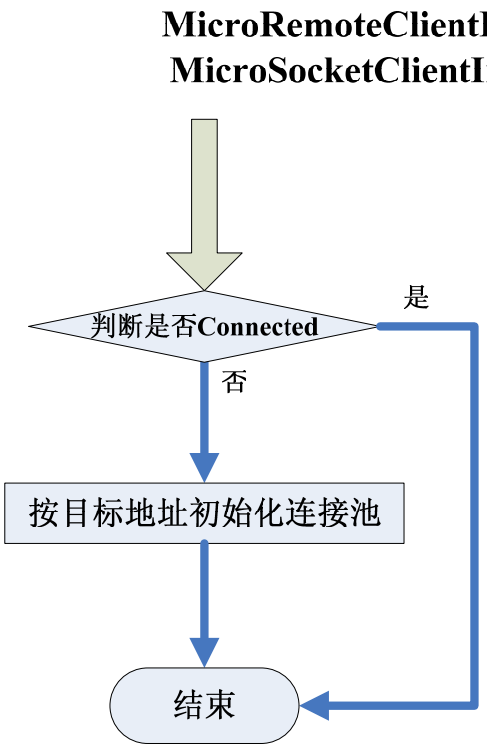


上图的过程和 Server 端创建 ServerInvoker 时如出一辙，JBoss 在此处体现出了它的设计功力。

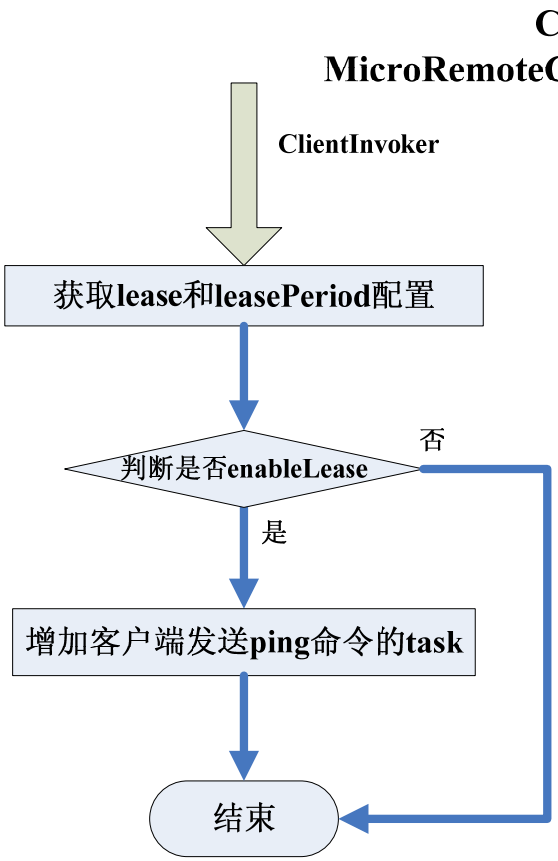
在创建完 ClientInvoker 后，回过头看第⑤步是如何实现的：



其中第⑥步是调用具体的协议的 ClientInvoker 实现，Socket 方式下是这么实现的：

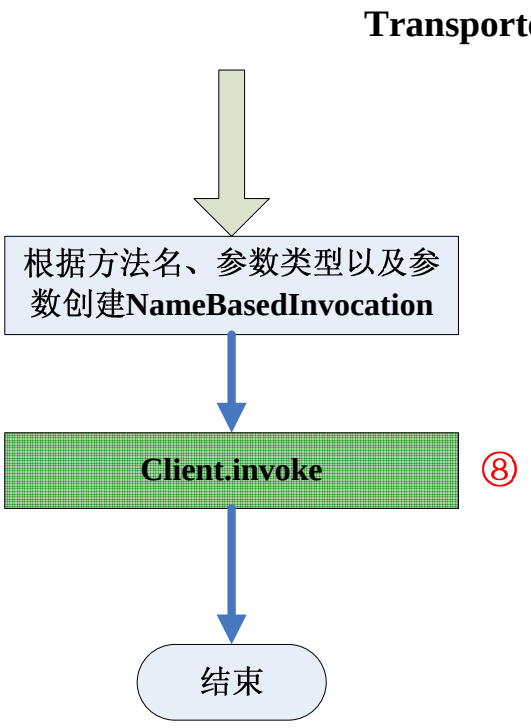


第⑦步的实现过程如下所示：



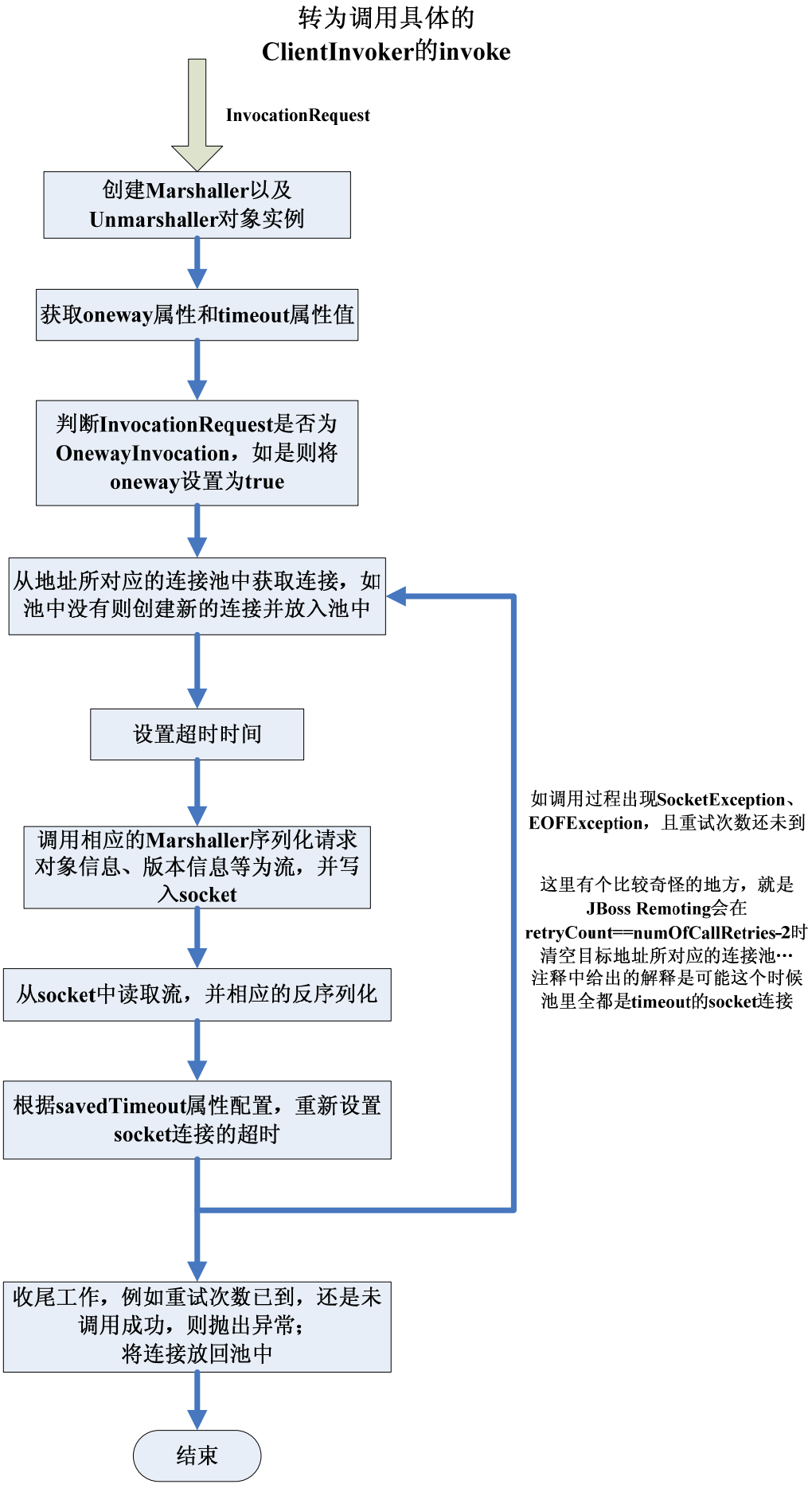
这一步是用于提供给服务器端校验客户端是否还 live 的方法，在 jboss remoting 的使用配置中有对应的 lease 项。

在创建出提供给客户端使用的 proxy 后，当客户端发起调用时，将执行如下过程：

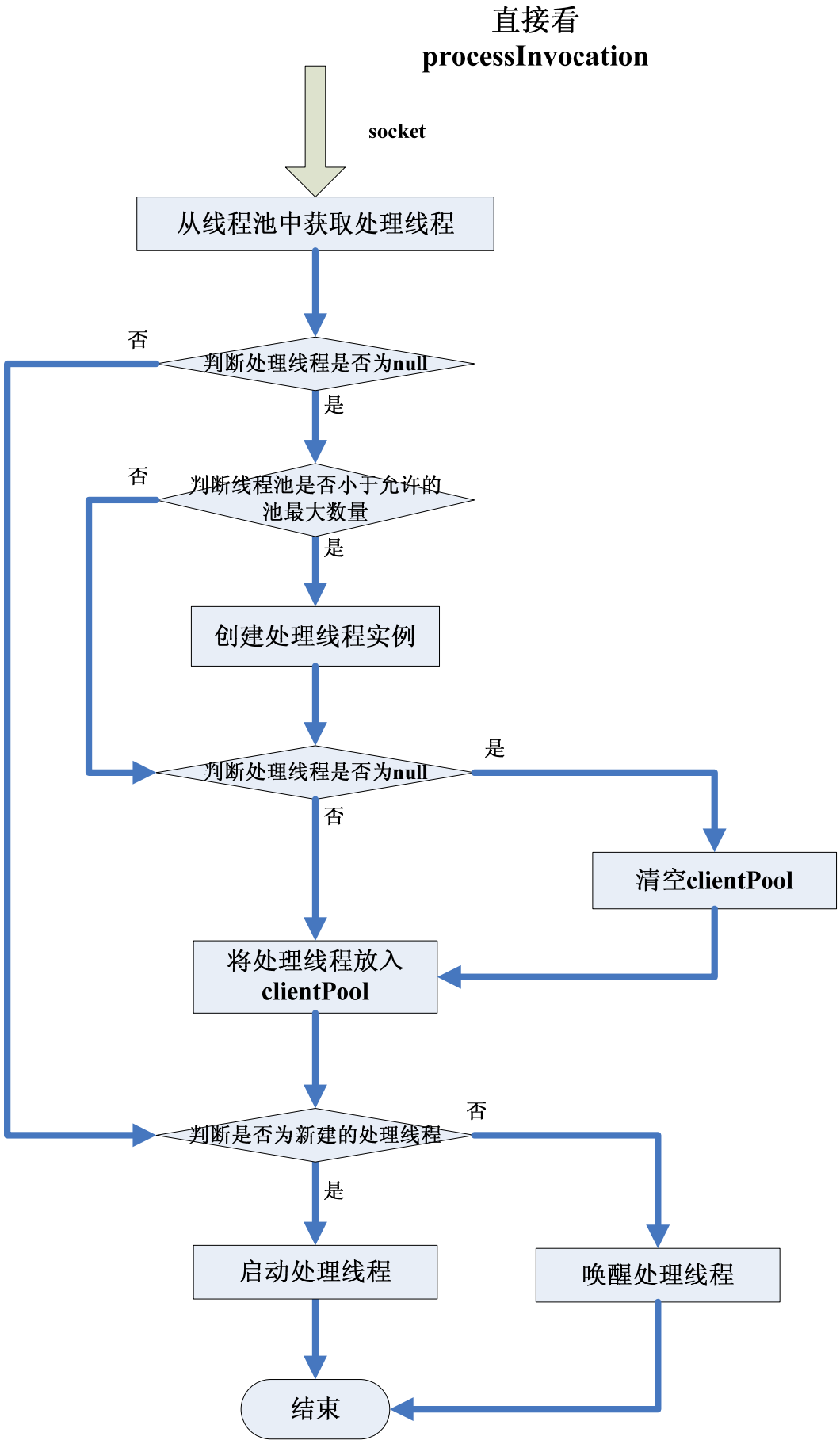


第⑧步调用具体的 clientInvoker 的 invoke 方法实现，例如 socket 协议方式下就调用

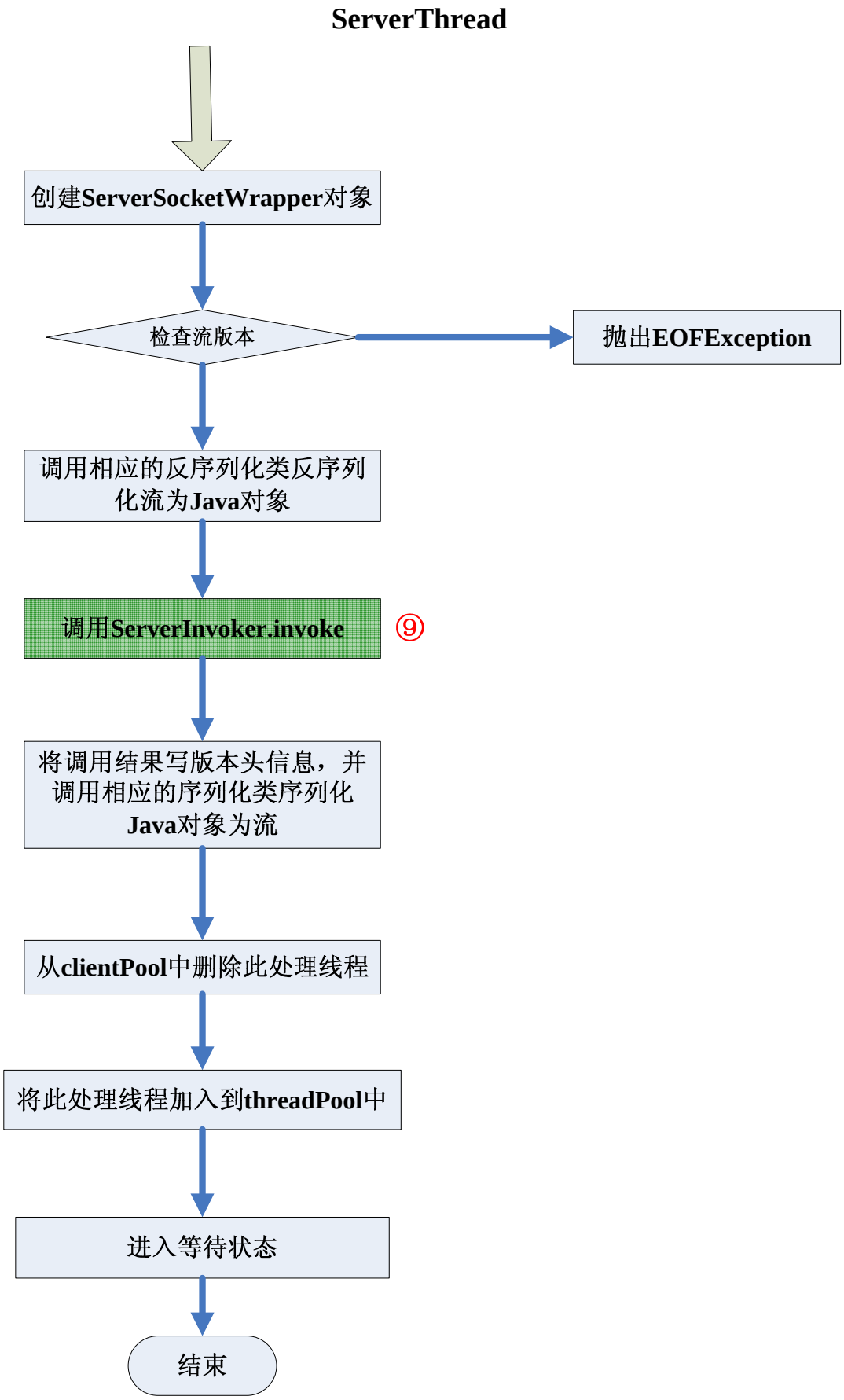
SocketClientInvoker：


转为调用具体的
ClientInvoker的invoke
InvocationRequest
创建Marshaller以及Unmarshaller对象实例
获取oneway属性和timeout属性值
判断InvocationRequest是否为OnewayInvocation，如是则将oneway设置为true
从地址所对应的连接池中获取连接，如池中没有则创建新的连接并放入池中
设置超时时间
调用相应的Marshaller序列化请求对象信息、版本信息等为流，并写入socket
从socket中读取流，并相应的反序列化
根据savedTimeout属性配置，重新设置socket连接的超时
收尾工作，例如重试次数已到，还是未调用成功，则抛出异常；
将连接放回池中
结束
如调用过程出现SocketException、EOFException，且重试次数还未到
这里有个比较奇怪的地方，就是JBoss Remoting会在retryCount==numOfCallRetries-2时清空目标地址所对应的连接池…注释中给出的解释是可能这个时候池里全都是timeout的socket连接

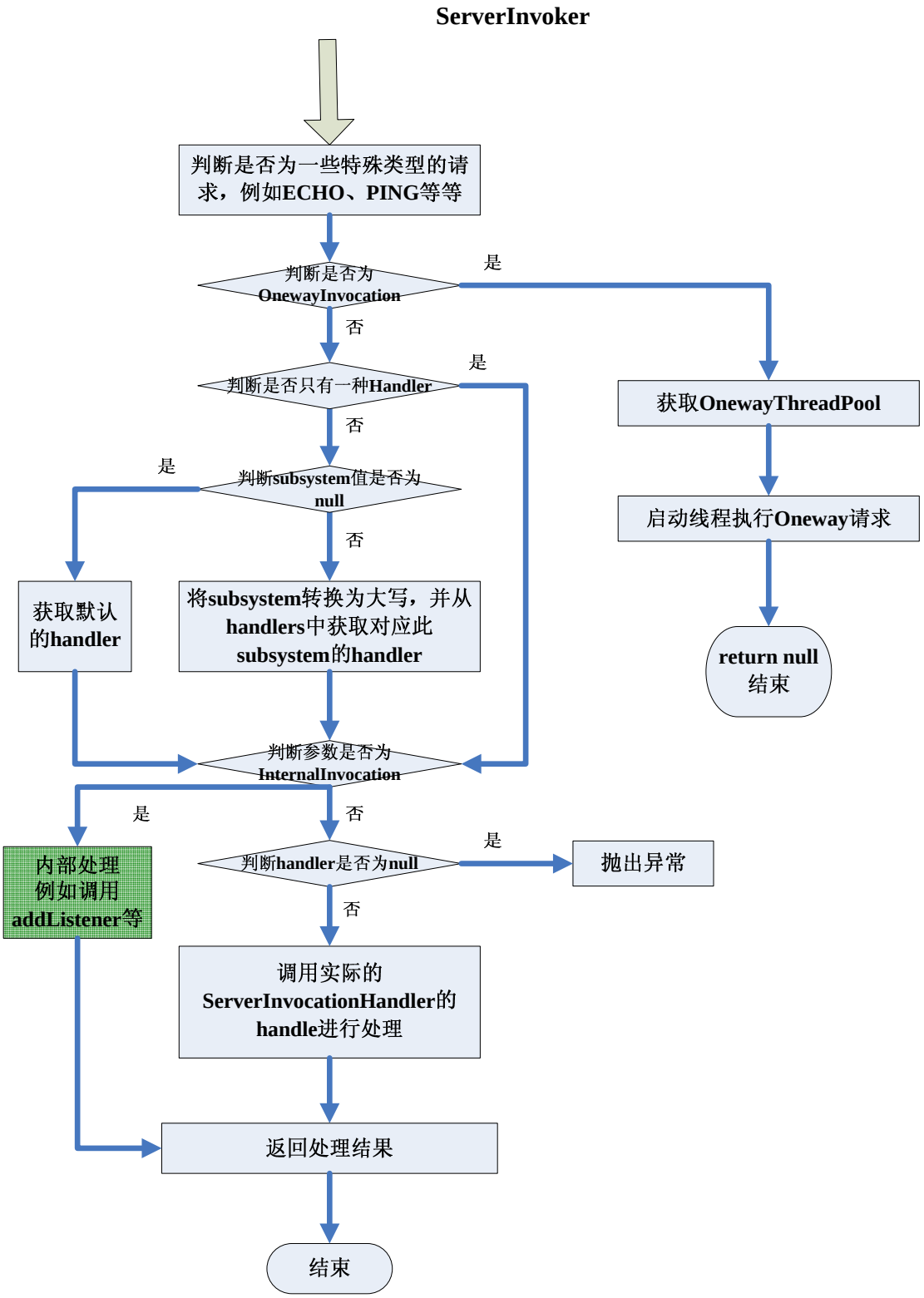
客户器端的全过程就是这样的，服务器端接收到请求后是怎么个处理过程呢，这就可以接着之前分析服务器端的启动过程来讲了，当时启动过程分析到服务器端启动了 AcceptThread，但没有更加深入的分析下去，现在再来分析 AcceptThread 会做什么：


直接看
processInvocation
socket
从线程池中获取处理线程
判断处理线程是否为null
否
是
判断线程池是否小于允许的
池最大数量
否
是
创建处理线程实例
判断处理线程是否为null
是
否
清空clientPool
将处理线程放入
clientPool
判断是否为新建的处理线程
否
是
启动处理线程
唤醒处理线程
结束

在上过程中启动或唤醒了处理线程后，AcceptThread 就可以继续接收其他的请求了，处理线程启动后会做如下事情：


ServerThread
创建ServerSocketWrapper对象
检查流版本
抛出EOFException
调用相应的反序列化类反序列化流为Java对象
调用ServerInvoker.invoke
⑨
将调用结果写版本头信息，并调用相应的序列化类序列化Java对象为流
从clientPool中删除此处理线程
将此处理线程加入到threadPool中
进入等待状态
结束

其中第⑨步的具体实现过程如下所示：



经过了上面的分析后，就看到了 Socket 方式下 JBoss Remoting 实现远程对象调用的方式，至于其他协议的方式的实现就不进行详细的分析了，大家可以按照这个过程直接去看相应的 transport 中扩展的 TransportServerFactory 和 TransportClientFactory，这其实呢只是个基本的

过程，应该说一个基本的远程对象调用的框架基本都是这么实现的，其实呢之上分析的过程中有不少是具备特色的，只是没有深入去讲解，接下来将进一步的来分析 JBoss Remoting 作为一个较完整且强大的分布式框架的一些特性是如何实现的。

2.2 多种调用方式的支持

同步的调用方式在之前已经以 Socket 方式为例做了详细的分析了，接下来来看看 JBoss Remoting 对于 Oneway 方式和异步方式支持的实现方式。

2.2.1 Oneway

Oneway 方式的调用呢，其实也可以列入异步调用的范畴，不过 JBoss Remoting 对于这两种调用的实现方法是不同的，因此还是分开来进行分析。

来看下 oneway 方式调用的发起方法，服务器端没有任何不同，只是客户端在发起调用时稍有不同，调用服务器端的 UserService 接口的实现对象的 insertUser 方法：

```
String serverUrl=”socket://127.0.0.1:12345”;
InvokerLocator locator=new InvokerLocator(serverUrl);
Client remoteClient=new Client(locator,UserService.class.getName());
remoteClient.connect();
User user=new User();
remoteClient.invokeOneway(user);
```

从代码上可以看出，发起 oneway 调用并不能做到和之前同步调用那样透明化的进行，这个就需要在使用 JBoss Remoting 时进行一定的封装，以达到 oneway 方式也能透明化的调用。

对于 Oneway 方式的调用，JBoss Remoting 支持两个层面的，一个是服务器端的 oneway，一个则是客户端方式的 oneway，这分别是什么意思呢？

服务器端的 oneway 指的是客户端等待请求发送至服务器端，服务器端处理时启动线程来进行处理，而不需要客户端等待服务器端的处理结果；

客户端的 oneway 调用则是指在客户端就启动线程来发起请求，因此当客户端发起调用时就已经直接返回了，而无需等待请求传递至服务器端。

JBoss Remoting 对于 oneway 方式调用的支持是基于线程方式来实现的，这些线程放入了 JBoss Remoting 的 Oneway ThreadPool 进行管理。

2.2.2 异步

异步调用一直以来都是个复杂的问题，异步调用带来了太多的不同，通常来讲异步调用的需求有这么几种：

- 之前所说的 Oneway 方式；
- 调用的发起是异步的，当服务器端处理完毕后，需要将结果返回给调用端；
- 调用的发起是异步的，当服务器端处理完毕后，需要将结果返回给调用端，同时还要确认调用端已收到结果。

通过来讲，要实现这些需求会采用消息中间件来进行实现，JBoss Remoting 对以上三种需求也都提供了支持，oneway 方式在之前已经分析过了，在此就针对后面两种需求进行分析，来看看 JBoss Remoting 如何支持以及它是如何实现的：

- 调用的发起是异步的，当服务器端处理完毕后，需要将结果返回给调用端；

  JBoss Remoting 对于此类需求提供的为通过 callback 的方式来支持，按照需求所描述的方式调用服务器端 UserService 的 insertUser 方法：

  - **服务器端**

```
public class UserServiceInvocationHandler implements ServerInvocationHandler{

    private List<InvokerCallbackHandler> callbackHandlers=new ArrayList<
InvokerCallbackHandler >();

    private Object target;

    public UserServiceInvocationHandler(Object target){
        this.target=target;
    }

    public void addListener(InvokerCallbackHandler callbackHandler){
        callbackHandlers.add(callbackHandler);
    }

    public Object invoke(InvocationRequest request) throws Throwable{
        // 处理完毕后调用 callbackHandler 进行回调
    }
```

```
        …
    }

    String locatorURI=”socket://127.0.0.1:12345”;
    Connector connector=new Connector(locatorURI);
    connector.addHandler(UserService.class.getName(),new  UserServiceInvocationHandler(new
UserServiceImpl()));
    server.start();
```

■ **客户端**

由于 JBoss Remoting 未提供像 TransporterClient 那样的方式来实现透明的异步调用的方式，因此在远程对象的调用时需要稍做处理。

```
    public class DefaultInvokerCallbackHandler implements InvokerCallbackHandler{
        public void handleCallback(Callback callback) throws HandleCallbackException{
            // 执行回调
            …
        }
    }

    String locatorURI=”socket://127.0.0.1:12345”;
    Client client=new Client(locatorURI,UserService.class.getName());
    client.addListener(new DefaultInvokerCallbackHandler(),new HashMap());
    // 封装一个包含了调用方法、参数和参数类型的对象，类似 NameBasedInvocation
    // 采用 oneway 方式以实现异步的调用，调用的结果则等待 callback
    client.invokeOneway(上面封装的对象,new HashMap(),true);
```

经过上面的步骤，就实现了对于服务器端 UserService 的异步调用，不过上面是一种较为简单的应用方法，其实还有更多的使用方法，具体可以参见 JBoss Remoting 的 User Guide。

从使用代码来看，可看出要实现异步调用，最主要的就是客户端需要 addListener，服务器端采用线程方式进行异步处理，之后通过客户端注册的 listener 进行回调，那么里面的具体实现过程到底是怎么样的呢，跟随 JBoss Remoting 的代码来看看。

在跟踪代码进行分析时，可以看到对于 callback 方式，JBoss Remoting 不断的提及 pull callback 和 push callback，来分别看下这两种 callback 的定义、实现以及使用的方法。

- **Pull callback**

Pull Callback，是指客户端需要主动通过调用 Client 的 getCallbacks 方法来获取回调的信息，服务器端处理完毕后并不会主动的将结果推向客户端。

addListener 时客户端发起一个请求给服务器端，Client 会根据注册的 InvokerCallbackHandler 对象实例生成一个 GUID 串，并将此串作为 Listener_id 注册到服务器端，服务器端在接收到请求后将此 Listener_id 进行保存，并生成一个服务器端的 ServerInvokerCallbackHandler 对象，调用 handler 的 addListener 方法并传入此 ServerInvokerCallbackHandler 对象。

当服务器端处理完毕，调用 callbackHandler 对象的 handleCallback 方法时，服务器端会将此结果放入 callback Store 中进行存储。

客户端通过 getCallbacks 进行调用时，会发起一个同步的请求给服务器端，服务器端从 callback store 中获取相应的 callback 结果。

使用时，调用 Client 的如下接口将采用 pull callback：

addListener(InvokerCallbackHandler)

- **Push callback**

Push callback，是指服务器端会将处理的结果推送给客户端，而无需客户端主动去调用 getCallbacks 来获取处理的结果。

Push Callback，在 JBoss Remoting 中又有两种，一种称为 true push callback，另一种为 simulate push callback。

true push callback 是指客户端创建一个监听 connector，相当于也是 server，当服务器端处理完毕调用 handleCallback 时，会发起请求给此 connector；

simulate push callback 是指客户端并不创建 connector，而是定时调用 client 的 getCallbacks 方法来获取服务器端的处理结果。

使用 True push callback 的前提是服务器端和客户端是可以双向连通的，或者采用了 jboss remoting 提供的 multiplex 类型的 transport，例如 bisocket 等，如采用的 transport 本身已经是支持 multiplex 的，那么可通过调用如下接口来实现 true push callback：

addListener(InvokerCallbackHandler,Map)

addListener(InvokerCallbackHandler,Map,Object)

如确认服务器端是可以连上客户端的，那么可以采用如下接口来实现 true push callback：

addListener(InvokerCallbackHandler,Map,Object,boolean)

并将最后一个参数设置为 true。

如已有可用的 connector，那么可通过如下接口来实现 true push callback：

addListener(InvokerCallbackHandler,InvokerLocator)

addListener(InvokerCallbackHandler,InvokerLocator,Object)

如要采用 simulate push callback，当服务器端连接不到客户端时，或采用的 transport 是非 multiplex 类型的时，那么调用如下接口时 JBoss Remoting 会自动的采用 simulate push callback：

addListener(InvokerCallbackHandler,Map)

addListener(InvokerCallbackHandler,Map,Object)

以上就是 JBoss Remoting 对于异步调用的支持以及具体的实现方法，从上面可以看出，目前 JBoss Remoting 在异步调用这块封装的还是没有同步调用的接口好用，如果在实际中需要使用 JBoss Remoting 来实现异步调用的话，需要在此层面上做一定的封装。

- 在上面的步骤中增加确认调用端已收到结果

  JBoss Remoting 通过 CallbackListener 来提供对于此需求的支持，首先服务器端的 ServerInvocationHandler 需要增加对于 CallbackListener 接口的实现，以接收客户端的反馈。

  另外就是客户端在被回调后，需要调用 client 的 acknowledgeCallback 方法来通知服务器端，具体实现时即为在此处封装相关的参数，然后发起 JBoss Remoting 的内部同步调用（参数类型为：InternalInvocation），由于发起同步调用的过程和之前分析的基本一致，在此就不再详细描述了。

2.3 远程加载 Class 的支持

当在客户端进行反序列化时，如相应的 class 未在客户端找到时，JBoss Remoting 支持从远程加载相应的 class，这个部分不需要进行特殊的配置。

这部分的关键代码在 AbstractInvoker 构造器进行构造时以及 ClassByteClassLoader。

在 AbstractInvoker 进行构造时，有这么几行代码：

```
InvokerLocator loaderLocator = MarshallLoaderFactory.convertLocator(locator);
if(loaderLocator != null)
{
```

```
    classbyteloader.setClientInvoker(new Client(loaderLocator));
}
```

接下去具体看下 ClassByteClassLoader 的代码，在 ClassByteClassLoader 中关于从远程加载 class 的最重要的代码就是 loadFromNetwork 部分的代码了，在这部分代码中可以看到它会发起一个对远程服务器的调用（端口和普通的远程调用是不同的，可通过配置中的 loader_port 进行配置），传递的参数为 load_class 以及类名。

服务器端处理此请求的类为 MarshallerLoaderHandler 类，此类在处理 load_class 调用时将会从服务器端获取需要加载的类的字节（读取相应的.class 文件来获得），并将字节和类名一起传入组成 ClassBytes 对象返回给客户端。

客户端在接收到此 ClassBytes 对象后，这样将其动态的加入到 classloader 中：

```
Class cl = null;
String name = classBytes.getClassName();
if(loadedClasses.containsKey(name) == false)
{
  byte buf[] = classBytes.getClassBytes();
  boolean array = ClassUtil.isArrayClass(name);
  String cn = (array) ? ClassUtil.getArrayClassPart(name) : name;
  cl = defineClass(cn, buf, 0, buf.length);
  resolveClass(cl);
  addClassResource(cn, buf);
  loadedClasses.put(cn, new MyRef(cn, cl));
}
```

经过如上步骤，JBoss Remoting 就实现了动态的加载远程 class。

2.4 高并发下的稳定性

在高并发场景下，系统并发的运行很有可能会带来很多低并发场景下无法想象和估计的问题，在这种时候，最重要的是保证系统的稳定性，避免出现雪崩效应是分布式框架中一个非常值得注意的问题。

对于框架而言，常见的现象会有程序的异常或程序对于框架的误用造成整个框架运行崩溃的可能，而对于分布式框架而言，在高并发的情况下以下两个比较常见的问题也有可能会导致整个框架甚至应用崩溃：

- **连接泄露或无限增长**

  这种问题对于低并发的情况下基本是很难发现的，而在高并发的情况下有可能会非常明显，造成连接泄露或无限增长的原因会有这么几种：

  - 连接不是放入池内，每次都重新创建，但没有释放；

    忘记释放的可能性比较低，但在出现异常的情况下有可能会发生，例如处理失败或释放失败等极端异常现象。

  - 连接放入池内，池的大小没有做合理的限制或连接一直放在池中而不销毁；

    这种现象则完全有可能出现，当池的大小未做合理的限制时，有可能会因为瞬间高峰造成系统资源耗尽，整个应用崩溃；而连接一直放在池中不销毁，则很容易造成服务器端的吞吐量降低甚至服务器端不再响应。

- **线程泄露或无限增长**

  这种问题对于低并发的情况下也很难发现，而在高并发的情况下也同样会非常明显，造成线程泄露或无限增长的原因会有这么几种：

  - 未采用线程池的情况下，创建的线程很久或一直不退出；

    这个现象出现的可能性较低，但仍然存在，典型的场景当线程中执行的方法出现逻辑错误或极端异常、又或所依赖的其他资源响应慢（例如数据库操作），很久或一直都不退出，而其他新的线程又不断创建的话，很有可能会造成整个系统资源耗尽而导致应用退出。

    另外一个现象仍然是瞬间高峰的问题。

  - 采用线程池的情况下，未做合理限制或线程永不超时销毁；

    未做合理限制仍然是无法应对瞬间高峰的问题，同时线程池限制的过小的话会导致系统的吞吐量降低；

    如果池中的线程永不超时销毁的话，容易出现的一个问题就是耗时长的线程占据了整个线程池，导致系统无法响应。

来看看 JBoss Remoting 是如何应对这些典型问题的。

2.4.1 程序的异常或误用导致框架或整个应用崩溃

这个问题对于框架来说向来就比较麻烦，因为 java 毕竟是单进程的程序，容错能力相对来讲会弱于进程隔离的程序。

JBoss Remoting 对于程序执行异常做了很多的控制，例如对于程序执行时的 Throwable 的捕捉，避免运行时的异常造成框架运行的崩溃。

在误用方面 JBoss Remoting 也做了很多的控制，例如判断传入参数的 null 以及不合理的等等现象。

JBoss Remoting 代码传达了 fail fast 的实现，这对于框架而言是非常重要的，毕竟很难控制框架的使用者如何去使用框架。

2.4.2 连接泄露或无限增长

在连接泄露或无限增长方面，JBoss Remoting 通过提供连接池，可限制大小以及超时时间的方式来避免连接泄露和无限增长，当然，这些还需要使用者合理的使用才能起到作用。

2.4.3 线程泄露或无限增长

在线程泄露或无限增长方面，JBoss Remoting 通过提供线程池、可限制大小以及超时时间的方式来避免线程泄露或无限增长。

其实 JBoss Remoting 除了提供了上述的一些保证高并发的稳定性的方法外，从代码上也做到了很多其他的工作，这都是值得学习的，例如在 JBoss Remoting 的代码（ServerThread）中有这样的注释：

```
// The following code has been changed to eliminate a race condition with
// SocketServerInvoker.cleanup().
//
// A ServerThread can shutdown for two reasons:
// 1. the client shuts down, and
// 2. the server shuts down.
//
// If both occur around the same time, a problem arises.  If a ServerThread
starts to
// shut down because the client shut down, it will test shutdown, and if
it gets to the
// test before SocketServerInvoker.cleanup() calls ServerThread.stop()
to set shutdown
// to true, it will return itself to threadpool.  If it moves from
clientpool to
// threadpool at just the right time, SocketServerInvoker could miss it
in both places
// and never call stop(), leaving it alive, resulting in a memory leak.
The solution is
// to synchronize parts of ServerThread.run() and
SocketServerInvoker.cleanup() so that
// they interact atomically.
```

虽然这样的方法是部分的损耗性能了，但对于应用而言，最重要的仍然是稳定性，而性能则

可以从很多其他的方面来做提升，绝对不能以降低稳定性以及系统运行的错误性来获取性能，这也是并发编程中需要注意的典型问题，因为集中式应用中很多的提高性能的方法到了分布式应用中可能会变成带来灾难的方法。

2.5 异常处理

在分布式应用中，异常的现象会对集中式应用多出很多，例如网络异常、序列化异常、并发异常等等；同时还有另外的一个现象就是从客户端看不出具体的服务器端异常，这给程序的编写带来了很多的难点。

来看下 JBoss Remoting 怎么应对这些问题。

2.5.1 服务器端异常的准确反馈

JBoss Remoting 支持将服务器端的异常信息传递回客户端，并能和客户端的堆栈汇集到一起最后形成完整的异常堆栈，这对于客户端开发而言是非常有帮助的。

JBoss Remoting 在这方面的做法是将服务器端的异常信息序列化返回至客户端，由客户端进行反序列化，这种情况下可能会出现的现象就是客户端并没有对应的异常类，造成反序列化失败，不过此时在日志中的异常信息还是很准确的。

2.5.2 自动重试机制

自动重试机制对于分布式框架而言，非常重要，自动重试机制能够在网络瞬间异常、服务器端方法执行的偶尔异常等情况下起到作用，避免无谓的错误。

JBoss Remoting 提供了 CallOfRetries 的方式来支持自动的重试。

2.5.3 服务器端连接不上时的通知（ConnectionListener）

当服务器端连接不上时，JBoss Remoting 提供了 ConnectionListener 来通知客户端，这一点非常的重要，ConnectionListener 会以一定的频率来检查服务器端的连接是否有效。

同时也可以主动的调用 ConnectionValidator 来检查是否能连接到服务器端。

2.5.4 客户端连接不上时的通知（ConnectionListener）

有些时候能否连上对应的客户端也是服务器端很关心的问题，JBoss Remoting 提供了 leasePeriod 这样的方式以及服务器端的 connectionListener 来确认客户端是否能够连接。

2.6 提升性能

性能无疑是分布式框架中重要的一点，JBoss Remoting 采取了些什么方法来提升其性能呢，主要采用了以下 5 种，来一一对其实现方法进行分析。

2.6.1 连接池

连接池是分布式框架中通常都会采用的一种方法，在连接池中会根据目标地址缓存已连接上

的连接，降低在高并发情况下重复建立连接的消耗。

在连接池上 JBoss Remoting 采用 LinkedList 来实现，对于连接池的管理性质的操作均采用同步此 LinkedList 锁来避免并发问题，池中连接的超时则通过 socket 本身的超时来控制。

2.6.2 智能本地调用

在之前分析 socket 方式的远程调用时，可以看到 JBoss Remoting 在客户端时就会判断此调用是否为本地的调用，如果是的话则不会跳转至远程进行调用，这对于性能的提升是很有作用的，避免了无谓的消耗，这算是 JBoss Remoting 的特点之一了，不过 JBoss Remoting 目前的这种判断是否是本地调用的方式在集群的场景下则需要做下另外的动作才能继续保持有效，至于为什么需要做调整，大家可以想下集群场景下是怎么个情况。

2.6.3 合理的缓存使用

JBoss Remoting 中大量的使用了本地缓存来提升系统的响应速度，以避免耗时的工作重复的进行，例如 ClientInvoker：

```
     ClientInvoker invoker = null;

     List holderList = (List) clientLocators.get(locator);
     if (holderList != null)
     {
        for (int x = 0; x < holderList.size(); x++)
        {
           ClientInvokerHolder holder = (ClientInvokerHolder)
holderList.get(x);
           if (sameInvoker(holder, configuration))
           {
              incrementClientInvokerCounter(holder);
              invoker = holder.getClientInvoker();
           }
        }
     }

     return invoker;
```

2.6.4 处理线程池

处理线程池是分布式框架中必备的，JBoss Remoting 在此部分的特色是它是采用的两个池来维护线程池的管理的，通过 LRUPool 来保持一个具备算法、时间性质的线程池的管理，同时使用简单的 LinkedList 来实现真实的线程的缓存，这种分离方式使得线程池的超时管理等得以很容易实现。

2.6.5 JBoss 序列化

JBoss 序列化并不属于 JBoss Remoting 范畴，但它是 JBoss Remoting 性能提升的关键点之一，这个从 JBoss Remoting 的 Performance BenchMark 中可以看出很多，但由于 JBoss 序列化不是 JBoss Remoting 范畴的，就不在这里对其进行分析了。

不过 JBoss Remoting 在高并发的情况好像有问题，不知道有没有同学在高并发的情况下使用过。

2.7 集群的支持

JBoss Remoting 直接提供了对于集群的支持，也就是说当客户端在调用服务器端时，如一台目标服务器出现异常时，JBoss Remoting 可支持自动 fail-over 到另外一台机器上去，不过 JBoss Remoting 并不提供对于 load balance 的支持，在寻找调用的目标机器时，它采取的是随机选择的方式，来看看其对于集群支持的具体实现方法。

要实现集群的支持，也就是说 JBoss Remoting Client 在访问目标 subsystem 时必须知道这些 subsystem 目前在哪些机器上注册了，JBoss Remoting Client 提供了 Multicast 和 JNDI Detector 两种方式来实现获取 subsystem 的目标地址信息。

2.7.1 Multicast Detector

Multicast 方式基于 UDP 协议来实现，只要对 UDP 协议稍有理解，就可以很容易的看懂 MulticastDetector 部分的代码了，关键的代码片段主要是以下几段：

```
      if(addr == null)
     {
        this.addr = InetAddress.getByName(defaultIP);
     }
     // check to see if we're running on a machine with loopback and no
NIC
     InetAddress localHost = InetAddress.getLocalHost();
     if(bindAddr == null &&
localHost.getHostAddress().equals("127.0.0.1"))
     {
        // use this to bind so multicast will work w/o network
        this.bindAddr = localHost;
     }
     SocketAddress saddr = new InetSocketAddress(bindAddr, port);
     socket = new MulticastSocket(saddr);
     socket.joinGroup(addr);
```

以上代码的作用为启动 UDP 的 socket 监听，接收通过 MulticastDetector 中启动的 listener 线程来实现，在这个线程中，启动了 socket.receive(p);来接收广播的数据：

```
// take the multicast, and deserialize into the detection event
          ByteArrayInputStream byteInput = new
ByteArrayInputStream(buf);
          ObjectInputStream objectInput = new
ObjectInputStream(byteInput);
          Object obj = objectInput.readObject();
          if(obj instanceof Detection)
          {
             Detection msg = (Detection)obj;
             if(log.isTraceEnabled())
             {
                log.trace("received detection: " + msg);
             }

             // let the subclass do the hard work off handling detection
             detect(msg);
          }
```

MulticastDetector 自身通过 hearbeart 来对外发送自己机器所注册的 subsystem 的信息：

```
        Detection msg = createDetection();
        try
        {
           if(log.isTraceEnabled())
           {
              log.trace("sending heartbeat: " + msg);
           }
           ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
           ObjectOutputStream objectOut = new
ObjectOutputStream(byteOut);
           objectOut.writeObject(msg);
           objectOut.flush();
           byteOut.flush();
           byte buf[] = byteOut.toByteArray();
           DatagramPacket p = new DatagramPacket(buf, buf.length, addr,
port);
           socket.send(p);
        }
        catch(Throwable ex)
        {
           // its failed
           log.debug("heartbeat failed", ex);
        }
```

通过以上代码就完成了活动的服务器的 subsystem 信息的注册和接收，那么对于不活动了的

服务器怎么办呢？MutlicastDetector 的父类在 start 时同时启动了一个 pinger 线程，这个就会负责完成对服务器健康性的检查，具体代码参见 AbstractDetector 中的内部类：FailureDetector。

JBoss Remoting 可通过配置 domain 来仅获取感兴趣的组的 subsystem 信息。

2.7.2 JNDI Detector

JNDI 方式的情况下，就是通过 JNDI 来获取具体的信息了，其他实现思路和 MulticastDetector 差不多，最关键的代码仍然是 JNDIDetector 中的 start、hearbeart 这些方法。

2.7.3 Fail Over

在具备了这些目标地址信息后，JBoss Remoting 需要做的就是在调用的时候支持 fail over 了：

```
do
{
    try
    {
        failOver = false;
        response = remotingClient.invoke(request);
    }
    catch (CannotConnectException cnc)
    {
        failOver = findAlternativeTarget();
        if (!failOver)
        {
            throw cnc;
        }
    }
    catch (InvocationTargetException itex)
    {
        Throwable rootEx = itex.getCause();
        throw rootEx;
    }
}
```

```
        while (failOver);
```

这个 JBoss Remoting 默认情况下采用的是随机的方式，具体代码参见 DefaultLoadBalancer：

```
if (servers != null){
    int size = servers.size();
    if (size > 1) {
        index = new Random().nextInt(size - 1);
    }
}
```

在具体的使用时，可以根据情况自行实现 load balance。

3 学到了什么

3.1 协议的灵活扩展和统一使用模型

JBoss Remoting 在统一 API 方面做的非常的好，并且同时构造了一个不错的扩展模型，这个是我们在做框架设计时值得参考和学习的。

3.2 多线程编程的最佳实践

JBoss Remoting 可以说是多线程编程领域的一个典型的产品，大量的线程的合理使用是 JBoss Remoting 的特色，观看 JBoss Remoting 的代码可以看到很多关于多线程编程的实践的最佳方式的指导，例如锁的使用、线程通知等，当然，随着 5.0 的改进，JBoss Remoting 在此方面也可以做更多的改进。

- **多线程下的资源的保护**

  在多线程的情况下，最明显的一点就是资源的保护了，并发情况下资源的读写尤其需要注意，这个从 JBoss Remoting 中很多地方采用同步机制可以看出，也是值得学习的，很多地方可以试着去想想，如果不用同步机制会发生什么，相信这对于提升多线程情况下的编程能力能够起到很大的帮助。

- **给线程取名字**

  给线程取名字，这个看起来很普通也很容易，但这点对于分布式应用的系统而言非常重要，尤其是在调试时。

  在 JBoss Remoting 的代码中所有新建线程的地方都会给线程命名，因此在调试时非常容易识别哪些线程是 JBoss Remoting 创建的。

3.3 良好的框架编程习惯

- **避免输入参数的错误造成崩溃**

  在 JBoss Remoting 中，可以看到很多对于输入参数合法性和运行时变量的校验，这是个非常好的习惯，也是保持框架稳定性的重要实践。

- **详细的 trace 日志**

  对于框架而言，详细的日志是非常重要，JBoss Remoting 基本上完全做到了，可以说看它的 trace 日志基本上就已经可以分析出 JBoss Remoting 在实现各个功能的关键代码流程了，这点上 JBoss Remoting 绝对是榜样。

- **详细单元测试和示例的编写**

  单元测试的重要性不在此啰嗦，JBoss Remoting 提供了非常大量的单元测试和示例，对于分布式这类的框架，很多人都会认为其单元测试是很难做的，而 JBoss Remoting 则向大家展示了 nothing is impossible，这也是非常值得学习的地方。

  示例则是快速上手的指南了，JBoss Remoting 中几乎所有的使用都可以从它提供的 examples 进行学习，很快就知道改怎么去使用了。

4 总结

从对 JBoss Remoting 的分析中，分布式应用带来的需要深入学习的知识体系较之集中式的应用多了很多，最基础也最明显的涉及到的知识体系有：网络通讯（涉及到的有协议、网络 I/O 等）、java 网络编程（java.net 包、NIO 等）、序列化机制、并发编程、池技术等，就这些知识点每个拓展开来讲都可以讲成大篇甚至一本书的范畴，对于大型分布式应用而言，涉及的知识体系就更多了，例如还需要掌握 cluster 环境下的处理（很多东西到了集群环境下复杂程度绝对是需要以翻倍来计算的，而且很多现在的处理方式都会变得不可用）、load balance 策略等，本文也只是对 JBoss Remoting 的一些基本以及关键的特点进行了分析（最开始的时候打算叫深入分析 JBoss Remoting 的，不过写到最后发现其实还有很多细节和深入的部分并没有写，因此还是改名叫分析 JBoss Remoting 了），其中其实还有很多细节是值得研究和学习的，希望有研究的同仁们贡献出其他方面的研究或指出本文错误的地方，非常感谢。

大型的分布式应用中不可能要求每个开发人员都去掌握这些知识体系，因此对于大型分布式应用而言，提供一个分布式的框架是非常有必要的，做到将分布式应用涉及的相关知识点尽量的剥离，就像 Erlang，做到将并发的基础知识分离。

JBoss Remoting 解决了很多分布式应用所需面对的问题，是目前可选的开源分布式框架中一

个很不错的选择，并且其在保证高并发场景下的稳定性和性能提升上也做了很多的工作，但还是有很多可提升的空间，例如统一的远程调用的 API 上（还可以进一步加强透明化的 oneway、异步的调用）、NIO 的支持、异步调用的提升（或者可以考虑提供结合 MQ 实现的异步调用）等等，而如果要成为大型分布式应用的支撑平台，无论是性能上还是功能上，JBoss Remoting 还有不少需要改进和提升的地方。

分布式应用较之集中式应用在对象的接口的设计和使用上也有了更高的要求，例如不要出现依靠参数引用传递来隐性的填充一些值、远程对象应是线程安全的、尽量不要出现频繁调用远程对象的现象、尽量减少往返传输大对象的现象等等细节。

分布式应用对比集中式应用而言，无论是开发还是支撑框架上都复杂了很多，因此尽管分布式应用相对集中式应用而言，能够带来机器配置要求降低、系统结构更加清晰和松耦合、降低维护的复杂度等等优点，但还是应该做到能不分布式就尽量不要分布式。

5 参考资料

Java网络编程：http://java.ccidnet.com/art/3737/20060309/457221_1.html

《JBoss Remoting User Guide》