Lnk RPC 使用说明文档

第一章　概述

1.1 项目背景

使用 MQ 实现的 RPC 特性：
1.可以把请求的压力保存到 MQ，逐渐释放出来，让处理者按照自己的节奏来处理。
2.由于 MQ 本身具有的缓冲性质，可以在应对高并发的时候保证请求不丢失。
3.异步单向的消息，不需要等待消息处理的完成，故原生支持异步调用模型。
4.无需服务注册中心和负载均衡的概念，完全依赖 MQ 来完成。
5.引入一下新的 MQ 结点，系统的可靠性会受 MQ 结点的影响。
6.对于有同步返回需求，用 MQ 则变得麻烦。
7.因为 AMQP 协议面向的场景主要是高可用性的场景，所以其服务端的实现略复杂，并且协议数据比直接 tcp 的裸数据大不少，另外无论是服务端还是客户端都存在着成对的拆包和封包的过程。

基于 Netty 和 Mina 等通讯框架实现的 RPC 的特性：
1.请求压力会直接传递到服务端，需要熔断，限流，降级，超时等服务治理策略。
2.需要有注册中心和负载均衡策略。
3.支持同步和异步调用。
4.性能比较高。

综上所述：由于 MQ 的稳定性依然成为我们目前系统稳定性和可靠性的最大障碍，而 RabbitMQ 的优化定制难度较高，所以寻找一款可替代的 MQ 服务或者去除 MQ 结点成为本项目的主题，在对市面上 MQ 的调研之后，我们决定选择后者方案，因为基于点对点模型的 RPC 解决方案目前来说比较成熟，可以借鉴学习的产品和思路比较多，很容易打造一款适合我们自身业务特性的 RPC 产品。

Lnk RPC 是一款基于 **Netty 和 Mina** 实现 RPC 通讯协议，支持同步，异步和异步回调三种 RPC 调用方式，支持参数和返回值多态。支持多种负载均衡方式，支持调用流量控制，支持 zookeeper 服务注册发现方式，服务端口支持开发人员，运维人员配置以及动态分配，支持服务依赖关系梳理以及调用链路跟踪。支持 spring 配置。在服务端通过分组策略将来自不同组别的请求处理资源隔离，该思路借鉴与 RocketMQ 的实现思想。

1.2 面向的读者

运维人员，开发人员。

1.3 RPC 产品所支持的功能

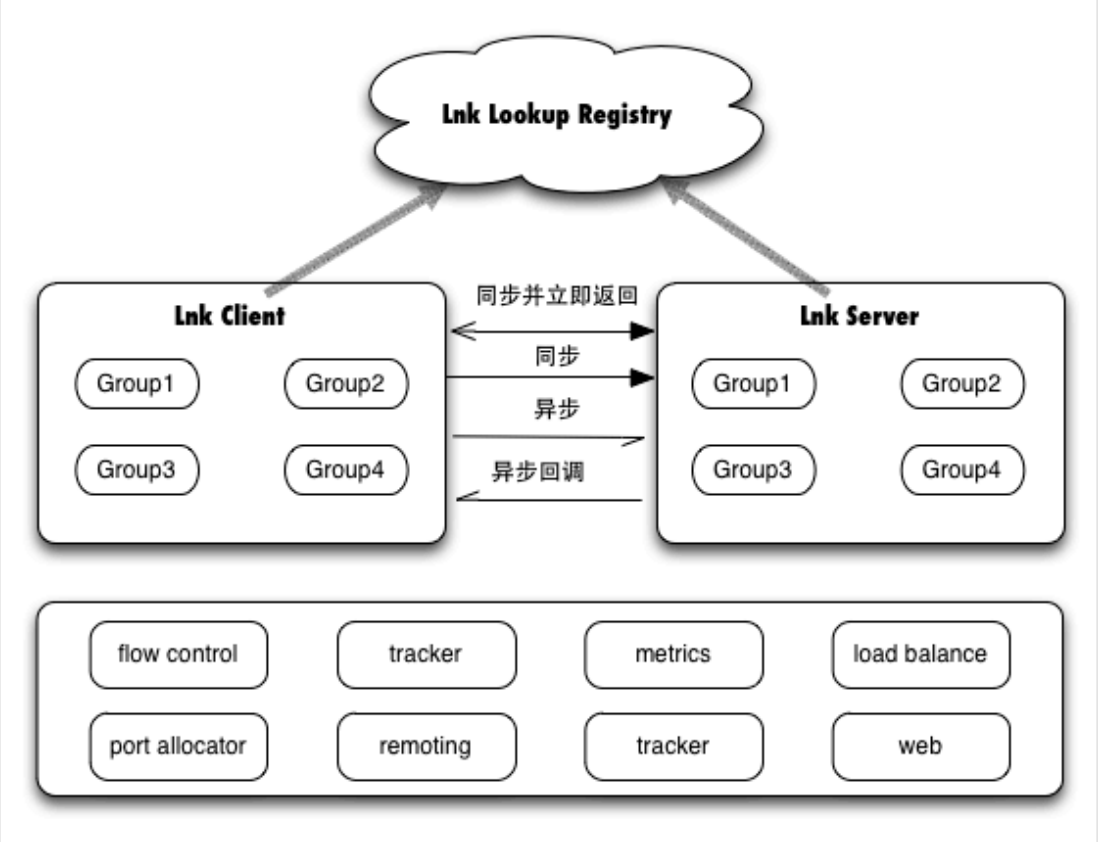
1. 同步调用
2. 异步调用
3. 异步组播调用
4. 异步结果回调
5. 支持服务多版本
6. 支持入参和返回值多态
7. 支持异常
8. 支持 RPC 服务序列化（实现异步回调的基础）

第二章　总体设计

2.1 项目模块划分

Lnk RPC 分为 api，cluster，core，demo，flow，lookup，port，protocol，remoting，spring，track，web 等模块。
API 模块主要定义各个模块所需的接口规范以及各个模块公用的工具和服务暴露所需的注解等配置
Cluster 模块主要是实现各种负载均衡策略，如：随机策略，一致性 hash 策略，轮询策略和本地优先策略。
Core 模块主要是通过聚合各个模块实现 RPC 服务端和客户端核心功能。
Demo 模块是 Lnk RPC 的各种使用方式实例。
Flow 模块是用于 Lnk 服务流量控制的模块。
Lookup 模块是实现服务注册发现中心的。
Port 模块是完成 RPC 服务端端口的配置和自动分配的模块。
Protocol 模块是对服务调用传输对象的序列化协议抽象模块。
Remoting 模块是对 **Netty 和 Mina** 的语意进行封装，实现同步和异步的调用方式，以及对不能的请求进行分组和资源隔离。
Spring 模块是扩展 spring 完成对 Lnk RPC 的配置和启动，实现服务的发布和依赖注入。
Track 模块是对 Lnk RPC 的调用实现跟踪的模块。
Web 模块主要是用于展示 Lnk 服务调用链路以及相应的应用依赖关系和进行服务治理的模块项目。

2.2 结构流程图



第三章　配置使用

3.1 使用配置

3.1.1 Spring XML 配置

```
<lnk:lnk id="payment-server" worker-threads="20"
selector-threads="15" connect-timeout-millis="3000"
      channel-maxidletime-seconds="120"
socket-sndbuf-size="65535" socket-rcvbuf-size="65535"
      pooled-bytebuf-allocator-enable="true"
default-worker-processor-threads="10"
default-executor-threads="8"
      use-epoll-native-selector="false">
      <lnk:application app="lnk-demo" ns-home="${nsHome}"
type="jar"/>
      <lnk:registry address="${registry}"/>
      <lnk:load-balance type="hash"/>
      <lnk:flow-control permits="10000"/>
```

```
      <lnk:bind>
         <lnk:service-group
service-group="biz-pay-bgw-payment.srv"
worker-threads="50"/>
         <lnk:service-group
service-group="biz-pay-bgw-payment.router.srv"
worker-threads="50"/>
      </lnk:bind>
   </lnk:lnk>
```

配置项说明：

listen-port：配置服务监听端口，可选配置，该配置项优先使用开发人员配置，如果没有配置或者端口被占用，框架会通过系统参数获取当前应用的分配端口，如果系统参数没有配置或者系统参数配置分配的端口被占用的话，框架会自动分配一个当前机器未使用的端口。

worker-threads：**Netty 和 Mina** 服务端事件处理线程池大小，默认为 10.

selector-threads：**Netty 和 Mina** 服务 NIO selector 事件处理线程池大小，默认为 5

connect-timeout-millis：客户端连接超时事件，单位毫秒。默认 3000 毫米即 3 秒。

channel-maxidletime-seconds：**Netty 和 Mina** 服务端连接通道最大空闲时间，单位为秒，默认为 120 秒

socket-sndbuf-size：服务端 socket 发送数据大小，默认为 65535

socket-rcvbuf-size：服务端 socket 接收数据大小，默认为 65535

pooled-bytebuf-allocator-enable：是否开启 Direct Buffer 选项，开启之后内核缓冲区的内容直接写入了 Channel，这样减少了数据拷贝。默认为 true

default- worker-processor-threads：服务端默认消息处理线程池的大小，用于处理一些未分组的请求。默认为 10

default- executor-threads：服务端默认线程池大小，主要用于处理异步回调事件。默认为 8

use-epoll-native-selector：是否使用操作系统 EPoll selector，默认 false。

application 子节点主要是标示应用名称和应用类型，主要用于做服务调用链路跟

踪和应用以来关系梳理。app 属性标示应用名称，type 标示应用类型分为 jar 和 war 类型。

registry 子节点主要标示服务注册中心类型和地址，用于 server 端注册自己的服务调用地址和端口，目前支持 zookeeper 注册中心，address 标示注册中心地址

load-balance 子节点表示选择一组服务调用地址的算法，type 指定算法类型，目前支持 hash 一致性 hash 算法，random 随机算法，roundrobin 轮询算法和本地优先算法。

flow-control 子节点主要标示流量控制单元，目前只支持使用 semaphore 信号量实现的流量控制。

bind 子节点主要是用于将服务端的服务划分为不同的组别，不同的组别使用自身组别的线程池，是的各个组别对外提供服务的线程等资源相互隔离。service-group 标示组别名称，worker-threads 标示改组请求处理线程池大小。默认为 10.

3.1.2 Server 端注解的使用

1. 接口级别的注解

```
@LnkService(group = "biz-pay-bgw-payment.srv")
public interface AuthService {
    @LnkMethod(type = InvokeType.SYNC, timeoutMillis = 3000L)
    AuthResponse auth(AuthRequest request) throws
AppBizException;

    @LnkMethod(type = InvokeType.SYNC, timeoutMillis = 3000L)
    AuthResponse auth_poly(AuthRequest request) throws
AppBizException;
}
```

@LnkService 注解有 2 个元素，group 元素标记了这个服务的资源组，protocol 元素标记了这个服务使用的序列化传输协议编码

@LnkMethod 注解有 2 个元素，type 元素标示了这个方法的调用类型，同步，异步和异步组播，timeoutMillis 元素标记了这个方法调用超时时间，单位为毫秒。

2. 接口实现级别的注解

```
@LnkServiceVersion(version = "2.0.0")
public class V2AuthService implements AuthService {
    ……
}
```

@LnkServiceVersion 注解标记了这个实现的版本，不配置这个注解的化，默认是 1.0.0 版本。

3.1.3 Client 端注解的使用

```
@Lnkwired(localWiredPriority = false)
AuthService defaultAuthService;

@Lnkwired(version = "2.0.0", localWiredPriority = false)
AuthService v2AuthService;
```

@Lnkwired 注解标记了要使用的这个 RPC 服务使用情况，localWiredPriority 标示是否优先使用本地的 bean，version 标示了使用的这个服务的版本编号。

第四章　开发注意事项

4.1 入参和返回值多态的使用

如定义接口如下：

```
@LnkService(group = "biz-pay-bgw-payment.srv")
public interface AuthService {
    @LnkMethod(type = InvokeType.SYNC, timeoutMillis = 3000L)
    AuthResponse auth_poly(AuthRequest request) throws
AppBizException;
}
```

服务端的实现如下：

```
@Override
    public AuthResponse auth_poly(AuthRequest request)
throws AppBizException {
        PolyAuthRequest polyAuthRequest = (PolyAuthRequest)
request;
        PolyAuthResponse response = new PolyAuthResponse();
        response.setGateId("0106");
        response.setGateRespCode("TXN.000");
        response.setGateRespMsg("Default V1 : " +
polyAuthRequest);
        response.setTxnId(polyAuthRequest.getTxnId());
```

```
        response.setPrincipalId("101");
        response.setTxnType("00001");
        response.setProductType("QP");
        response.setIdType("01");
        return response;
    }
```

可以看到 Lnk RPC 对多态的支持无需任何代码级别的配置和标记