QiuRPC 参考手册

目录

RPC 常见功能

一个通用的网络 RPC 框架，它应该包括如下元素：

1.具有服务的分层设计，借鉴 Future/Service/Filter 概念
2.具有网络的分层设计，区分协议层、数据层、传输层、连接层
3.独立的可适配的 codec 层，可以灵活增加 HTTP，Memcache，Redis，MySQL/JDBC，Thrift 等协议的支持。
4.将多年各种远程调用 High availability 的经验融入在实现中，如负载均衡，failover，多副本策略，开关降级等。
5.通用的远程调用实现，采用 async 方式来减少业务服务的开销，并通过 future 分离远程调用与数据流程的关注。
6.具有状态查看及统计功能
7.当然，最终要的是，具备以下通用的远程容错处理能力，超时、重试、负载均衡、failover……

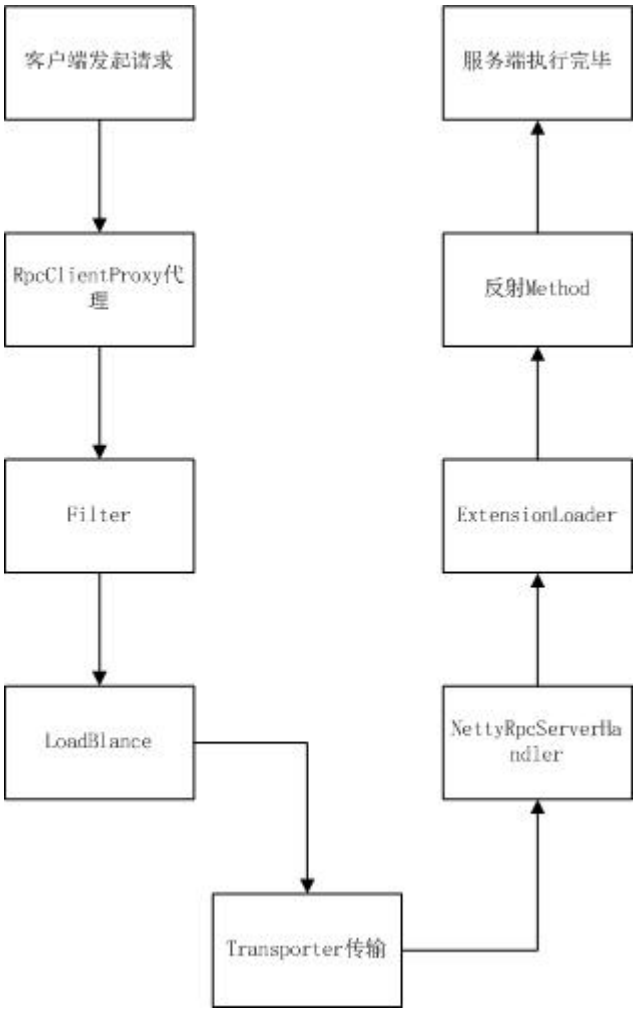
QiuRPC 特点

QiuRPC 是一个采用 JAVA 实现的小巧的 RPC 框架，一共 3K 多行代码，实现了 RPC 的基本功能，开发者也可以自定义扩展，可以供大家学习探讨或者在小项目中使用，目前 QiuRPC 具

有如下特点：

1. 服务端基于注解，启动时自动扫描所有 RPC 实现，基本零配置
2. 客户端实现 Filter 机制，可以自定义 Filter
3. 基于 netty 的 Reactor IO 多路复用网络模型
4. 数据层提供 protobuff 和 hessian 的实现，可以扩展 ISerializer 接口自定义实现其他
5. 负载均衡算法采用最少活跃调用数算法，可以扩展 ILoadBlance 接口自定义实现其他
6. 客户端支持服务的同步或异步调用

QiuRPC 流程



系统改进点

1. 增加注册中心功能，在大项目中，一个项目可能依赖成百上千个服务，如果基于配置文件直接指定服务地址会增加维护成本，需要引入注册中心。
2. 目前用的是反射和 java 代理实现的服务端存根和客户端代理，为了提高性能，可以把

这些用 javassit，asm 等 java 字节码工具实现

3. 增加一些监控功能，为了增强服务的稳定性和服务的可控性，监控功能是不可或缺的
4. 目前应用协议采用的是最简单的协议，仅仅一个魔数+序列化的实体，这些需要增强，比如增加版本号以解决向前兼容性
5. 增加 High availability 的一些手段，目前只有负载均衡，其他的比如 failover，多副本策略，开关降级等，过载保护等需要自己实现。

参考例子

1. 编写服务端接口

```
public interface IServer1 {
   public String getMsg();

   public Message echoMsg(String msg);

   public Message echoMsg(int msg);
}
```

2. 编写服务端实现类

```
@ServiceAnnotation(name="myserver1")
public class MyServer1 implements
IServer1{
   private static final Log
log=LogFactory.getLog(MyServer1.class);
```

```
public String getMsg()
{
  log.info("getMsg echo");
  return "Hello";
}

@Override
public Message echoMsg(String msg) {
  Message result=new Message();
  result.setMsg(msg);
  result.setData(new Date());
  return result;
}

@Override
public Message echoMsg(int msg) {
  Message result=new Message();
  result.setMsg("int:"+msg);
  result.setData(new Date());
  return result;
}
```

```
}
```

3. 启动服务

```
public static void main(String[] args) {
    RpcServerBootstrap bootstrap=new
RpcServerBootstrap();
    bootstrap.start(8080);
  }
```

4. 编写客户端调用代码

```
public class Client1 {

  public static void main(String[] args)
{
    try {
      final IServer1
server1=RpcClientProxy.proxy(IServer1.cla
ss,"server1" , "myserver1");
      long
startMillis=System.currentTimeMillis();
```

```
        for(int i=0;i<10000;i++)
        {
          final int f_i=i;
          send(server1,f_i);
        }
        long
endMillis=System.currentTimeMillis();
        System.out.println("spend
time:"+(endMillis-startMillis));
      } catch (Throwable e) {
        e.printStackTrace();
      }
    }

    public static void send(IServer1
server1,int f_i)
    {
      Message msg = null;
      try
      {
        //由于客户端配置的async="true"，我们
用异步方式来获取结果，如果是同步方式，直接
```

```
msg=server1.echoMsg(f_i)即可
        server1.echoMsg(f_i);
        Future<Message> future =
RpcContext.getContext().getFuture();
        msg=future.get();

   System.out.println("msg:"+msg.getMsg()+
","+msg.getData());
      }
      catch(Throwable e)
      {
        e.printStackTrace();
      }
   }
}
```

5. 编写客户端配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 客户端配置文件 -->
<application maxThreadCount="100">
  <service name="server1"
```

```
connectStr="127.0.0.1:9090;127.0.0.1:8080
" maxConnection="100"
async="true"></service>
<!--  <service name="server1"
connectStr="127.0.0.1:8080"
maxConnection="100"
async="false"></service> -->
</application>
```