# 轻量级分布式服务化框架

## 基本原理



轻量级分部署服务调度框架的基本原理是服务提供方Provider提供rpc服务，同时把ip和端口以及发布的rpc服务注册到注册中心，客户端或者rpc消费者从注册中心获取服务Provider列表，同时获取Provider提供的服务列表。另外客户端还会监听注册中心的数据变化，获知server宕机或者服务不可用，将该Provider从客户端Provider缓存列表中剔除，方便做容错和负载均衡。

### 特性

1. 负载均衡

提供基于RoundRobin和随机方式的负载均衡

1. 高可用

Consumer会从注册中心获取到服务列表及该服务的提供者列表，如果某个提供者Provider网络异常或者宕机，Consumer能马上感知到，加入不可用列表，如果从注册中心收到服务不可用会剔除缓存，不可用列表会重新尝试发起连接，如果网络正常了会立即恢复。

1. 泛型

一般的rpc调用需要拿到服务提供方的业务api(interface class，入参class，返回值class打包到一个jar中，依赖该jar)，如果使用泛型，只需要填写interface的name,版本，方法名称，参数名称，参数值，如果是对象，将对象字段封装到一个Map中即可，无需依赖任何业务jar即可完成rpc调用。

1. Rpc上下文附件

Rpc调用方可以将需要传递的上下文信息填写到上下文中，而不是作为rpc的入参，这样Provider可以从上下文中获取到掉用方的上下文信息。

1. 高可用注册中心

Rpc 框架提供了zookeeper，etcd，redis pubsub（支持单个，或者sentinel集群模式）的注册中心，具备高可用功能。

1. 实时动态监控

Provider提供了监控的api，监控可以使用该api加入到项目或者公司的监控平台。

1. 内存使用少

Rpc使用的内存模型是tcp连接建立后自动分配一块内存，读和写都在该内存中，不需要为每次请求分配内存，同一个tcp内存复用，数据使用了压缩的方式保存和发送，不支持返回数据量超大的调用（压缩后超过1m）。

## 整体架构



泛型：GenericService，Consumer不依赖Provider的api jar包即可完成remote api调用

监控：StatMonitor,consumer注册需要的远程服务StatMonitor，调用rpc获取监控数据

Webui:一个可视化rpc管理界面，<https://github.com/lindzh/rpc-webui>

RPC调用：使用jdk proxy封装发送tcp数据，并等待数据返回，完成RPC调用。

负载均衡：在集群模式下，同一个版本的Rpc服务在多台服务器上部署，Consumer发起rpc调用使用负载均衡。

自动容错：发现rpc provider不可用及时剔除，当可用时加入。

代理：rpcClient注册一个remote intface时会返回一个代理。

多注册中心：提供Zookeeper，etcd，redis等注册中心，并可以实现高可用。

## 部署

说明：直连模式不需要注册中心，无需部署注册中心

### 注册中心

Zookeeper: <http://zookeeper.apache.org/doc/r3.4.5/zookeeperStarted.html>

Etcd：<https://github.com/coreos/etcd/releases>

Redis: <http://redis.cn/>

### 代码编译与上传maven仓库

Rpc框架编译：

RPC:

git clone https://github.com/lindzh/rpc.git

cd rpc

//没有maven仓库上传到本地maven

maven clean install –Dmaven.test.skip

//有maven仓库 增加pom distributionManagement配置

maven clean install deploy –Dmaven.test.skip

RPC cluster组件

git clone <https://github.com/lindzh/rpc-cluster.git>

cd rpc-cluster

//没有maven仓库上传到本地maven

maven clean install –Dmaven.test.skip

//有maven仓库 增加pom distributionManagement配置

maven clean install deploy –Dmaven.test.skip

RPC webui

Web ui需要tomcat部署war包

git clone <https://github.com/lindzh/rpc-webui.git>

cd rpc-webui

maven clean install war:inplace –Dmaven.test.skip

在target 下面会生成war包

使用war包部署，解压war并配置

配置

webui.json

[

{

"namespace": "default",

"protocol": "redis",

"etcdUrl": null,

"redisHost": "192.168.139.129",

"redisPort": 7770,

"sentinelMaster": null,

"sentinels": null,

"providerHost": null,

"providerPort": 0,

"zkConnectionString": null

}

]

namespace:for different cluster

protocol:redis,etcd,zookeeper,simple,for cluster message notify and store

etcdUrl:when protocol is etcd use

redisHost:when protocol is redis use and use a single redis node

redisPort:when protocol is redis use and use a single redis node

sentinelMaster:when protocol is redis use and use redis sentinel for fail over

sentinels:when protocol is redis use and use redis sentinel for fail over

providerHost:when protocol is simple single node provider use

providerPort:when protocol is simple single node provider use

zkConnectionString:when protocol is zookeeper use

## 使用

### 直连模式

直连模式无需部署注册中心，Provider和Consumer直接通过TCP连接通信，发起Rpc调用。此种模式不具备高可用和负载均衡功能。

#### Privider

SimpleRpcServer rpcServer = **new** SimpleRpcServer();

rpcServer.setHost("192.168.132.87");

rpcServer.setPort(4321);

//将一个service暴露为rpc服务

rpcServer.register(LoginRpcService.**class**, **new** LoginRpcServiceImpl());

rpcServer.startService();

// Thread.*currentThread*().*sleep*(100000);//wait for call

rpcServer.stopService();

说明：服务提供方将一个rpc服务暴露出去后必须调用startService，这个方法会初始化tcp socket，等待Consumer发起Rpc。

#### Consumer

SimpleRpcClient rpcClient = **new** SimpleRpcClient();

rpcClient.setHost("192.168.132.87");

rpcClient.setPort(4321);

LoginRpcService loginRpcService = rpcClient.register(LoginRpcService.**class**);

rpcClient.startService();

**boolean** loginResult = loginRpcService.login("admin", "admin");

rpcClient.stopService();

register实例化一个rpc远程服务，返回一个代理对象，调用该代理对象发起rpc请求。

### 集群模式

#### Provider

EtcdRpcServer rpcServer = **new** EtcdRpcServer();

rpcServer.setEtcdUrl("http://192.168.139.129:2911");

rpcServer.setNamespace("myapp-mymodule");

rpcServer.setPort(3351);

rpcServer.register(LoginRpcService.**class**, **new** LoginRpcServiceImpl());

rpcServer.startService();

zookpeeper注册中心

ZkRpcServer rpcServer = **new** ZkRpcServer();

rpcServer.setConnectString("192.168.139.129:2215,192.168.139.121:2225,192.168.139.126:2235");

rpcServer.setNamespace("myapp-mymodule");

说明：添加namespace的原因是为了区分不同的产品和模块,多个产品和模块可以使用同一个注册中心和webui

#### Consumer

//etcd注册中心

EtcdRpcClient client = **new** EtcdRpcClient();

client.setEtcdUrl("http://192.168.139.129:2911");

client.setNamespace("myapp-mymodule");

HelloRpcService rpcService = client.register(HelloRpcService.**class**);

client.startService();

//zookeeper注册中心

ZkRpcClient client = **new** ZkRpcClient();

client.setConnectString("192.168.139.129:2215,192.168.139.121:2225,192.168.139.126:2235");

client.setNamespace("myapp-mymodule");

HelloRpcService rpcService = client.register(HelloRpcService.**class**);

client.startService();

### Filter使用

过滤器和tomcat的过滤器比较像，可以使用filter做过滤或者日志记录等。

实现一个Filter，只需实现RpcFilter接口即可

**public** **class** MyTestRpcFilter **implements** RpcFilter{

**private** Logger logger = Logger.*getLogger*(MyTestRpcFilter.**class**);

@Override

**public** **void** doFilter(RpcObject rpc, RemoteCall call, RpcSender sender,

RpcFilterChain chain) {

logger.info("request ip:"+rpc.getHost()+" port:"+rpc.getPort());

chain.nextFilter(rpc, call, sender);

}

}

有了RpcFilter之后才能加入到Provider中去

rpcServer.addRpcFilter(new MyTestRpcFilter());

rpcServer.startService();

### 附件使用

附件是Consumer上传，从Provider端获取的属性

#### Provider

@Override

**public** **boolean** login(String username, String password) {

//获取上下文附件

String haha = (String)RpcContext.*getContext*().getAttachment("haha");

System.*out*.println("login:user:"+username+" pass:"+password+" attach haha:"+haha);

String pass = cache.get(username);

//清除上下文附件

RpcContext.*getContext*().clear();

**return** pass!=**null**&&pass.equals(password);

}

为什么要清除？

RpcContext使用的是ThreadLocal，执行完毕后必须清除

#### Consumer

LoginRpcService loginRpcService = client.register(LoginRpcService.**class**);

RpcContext.*getContext*().setAttachment("haha", "this is haha");

**boolean** login = loginRpcService.login("admin", "admin");

RpcContext.*getContext*().clear();

### 泛型使用

泛型是客户端不需要依赖provider的jar以及class文件，只需要知道class名称，版本，方法名称，方法参数类型的String值即可发起Rpc调用

Provider提供方的Rpc如下：

//remote api定义如下

**public** **interface** HelloRpcService {

**public** **void** sayHello(String message,**int** tt);

**public** TestRemoteBean getBean(TestBean bean,**int** id);

}

//实现类如下

**public** **class** HelloRpcServiceImpl **implements** HelloRpcService{

**private** Logger logger = Logger.*getLogger*(HelloRpcServiceImpl.**class**);

@Override

**public** **void** sayHello(String message,**int** tt) {

Object attachment = RpcContext.*getContext*().getAttachment("myattachment");

System.*out*.println("my attachment:"+attachment);

System.*out*.println("sayHello:"+message+" intValue:"+tt);

}

@Override

**public** TestRemoteBean getBean(TestBean bean, **int** id) {

Object attachment = RpcContext.*getContext*().getAttachment("myhaha");

System.*out*.println("my attachment:"+attachment);

//logger.info("id:"+id+" bean:"+bean.toString());

TestRemoteBean remoteBean = **new** TestRemoteBean();

remoteBean.setAction("fff-"+id);

remoteBean.setAge(id\*2);

remoteBean.setName("serviceBean");

**return** remoteBean;

}

}

//TestBean定义如下：

**public** **class** TestBean **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = -6778119358481557931L;

**private** **int** limit;

**private** **int** offset;

**private** String order;

**private** String message;

//getter and setter

}

//暴露rpc

//Provider暴露RPC

rpcServer.register(HelloRpcService.class,new HelloRpcServiceImpl());

此时Consumer可以发起RPC调用

GenericService service = rpcClinet.register(GenericService.**class**);

String[] getBeanTypes = **new** String[]{"com.linda.framework.rpc.TestBean","int"};

HashMap<String,Object> map = **new** HashMap<String,Object>();

map.put("limit", 111);

map.put("offset", 322);

map.put("order", "trtr");

map.put("message", "this is a test");

Object[] getBeanArgs = **new** Object[]{map,543543};

RpcContext.*getContext*().setAttachment("myhaha", "myattachment value");

Object hh = service.invoke("com.linda.framework.rpc.HelloRpcService", RpcUtils.*DEFAULT\_VERSION*, "getBean", getBeanTypes, getBeanArgs);

RpcContext.*getContext*().clear();

System.*out*.println("getBean result:"+hh);

String[] argTypes = **new** String[]{"java.lang.String","int"};

Object[] args = **new** Object[]{"hello,this is linda",543543};

RpcContext.*getContext*().setAttachment("myattachment","myhaha value");

Object invoke = service.invoke("com.linda.framework.rpc.HelloRpcService", RpcUtils.*DEFAULT\_VERSION*, "sayHello", argTypes, args);

RpcContext.*getContext*().clear();

System.*out*.println("sayHello result:"+invoke);

### 监控

监控的原理是内置了一个RPC的Filter和一个RPCmonitor的组件，发起RPC调用获取监控数据

**public** **class** RpcStatFilter **implements** RpcFilter,Service,StatMonitor

//获取监控对象

StatMonitor monitor = client.register(StatMonitor.class);

//定时获取数据

### Spring配置

#### RpcServer的实例化注入

<bean id="simpleRpcServer" class="com.linda.framework.rpc.server.SimpleRpcServer">

<property name="host" value="127.0.0.1"></property>

<property name="port" value="5432"></property>

</bean>

<bean class="com.linda.framework.rpc.spring.provider.RpcProviderProcessor" destroy-method="stopRpcService"/>

<context:component-scan base-package="com.linda.framework.rpc.spring.impl"></context:component-scan>

说明：simpleRpcServer无需指定初始化方法，RpcProviderProcessor会自动扫描并启动

Java注解暴露一个rpc服务

@Service

@RpcProviderService(rpcServer="simpleRpcServer")

public class HelloRpcServiceImpl implements HelloRpcService{

说明：RpcProviderProcessor会找到使用了RpcProviderService注解的bean，注册到RpcServer，如果是etcd或者zookeeper的会注册到注册中心。

自定义Filter加入到server中

@Component

@RpcProviderFilter(rpcServer="simpleRpcServer")

public class RpcTestFilter implements RpcFilter

使用RpcProviderFilter注解，RpcProviderProcessor会从spring容器中查找实现了该注解的bean，注册到rpcserver上

#### RpcConsumer配置

<bean id="simpleRpcClient" class="com.linda.framework.rpc.client.SimpleRpcClient">

<property name="host" value="127.0.0.1"></property>

<property name="port" value="5432"></property>

</bean>

<bean id="rpcPackages" class="java.util.ArrayList">

<constructor-arg>

<list>

<value>com.linda.framework.rpc.spring.test</value>

</list>

</constructor-arg>

</bean>

<bean id="rpcInvokerAnnotationConfigurer" class="com.linda.framework.rpc.spring.invoker.RpcInvokerAnnotationConfigurer" destroy-method="stopRpcService">

<property name="packages" ref="rpcPackages"/>

</bean>

申明client和interface的包，会自动扫描并实例化，interface需要注解

@RpcInvokerService(rpcServer="simpleRpcClient")

public interface HelloRpcService {

使用rpc服务HelloRpcService，和本地没任何区别，使用@Resource等注解直接申明即可使用

## 加入项目与提交patch

项目分为三大部分：

Rpc基本包: https://github.com/lindzh/rpc

分布式集群支持包：<https://github.com/lindzh/rpc-cluster>

Webui工程：<https://github.com/lindzh/rpc-webui>

有bug直接提issues，或者qq发送给我,加入项目直接发qq

QQ:839861706

另外：rpc-netty只是server和client端通信的实现，不需要依赖。如果要使用请看testcase，每个工程都有很多testcase