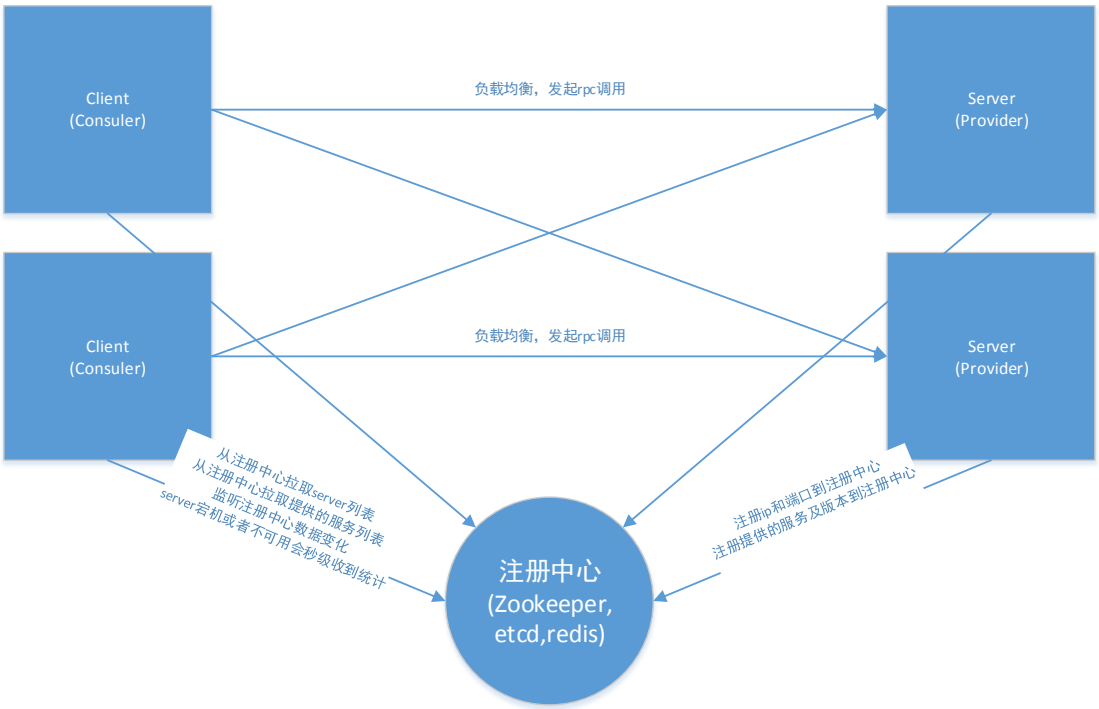
轻量级分布式服务化框架

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

一、负载均衡

提供基于 RoundRobin 和随机方式的负载均衡

二、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。

三、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。

四、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。

五、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。

六、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。

七、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("192.168.139.129:2215,192.168.139.121:222
5,192.168.139.126:2235");
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("192.168.139.129:2215,192.168.139.121:2225,1
92.168.139.126:2235");
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.RpcInvokerAnnotationConfigure
r" 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