


rpc系列5-添加拦截器链，实现rpc层面的AOP



作者

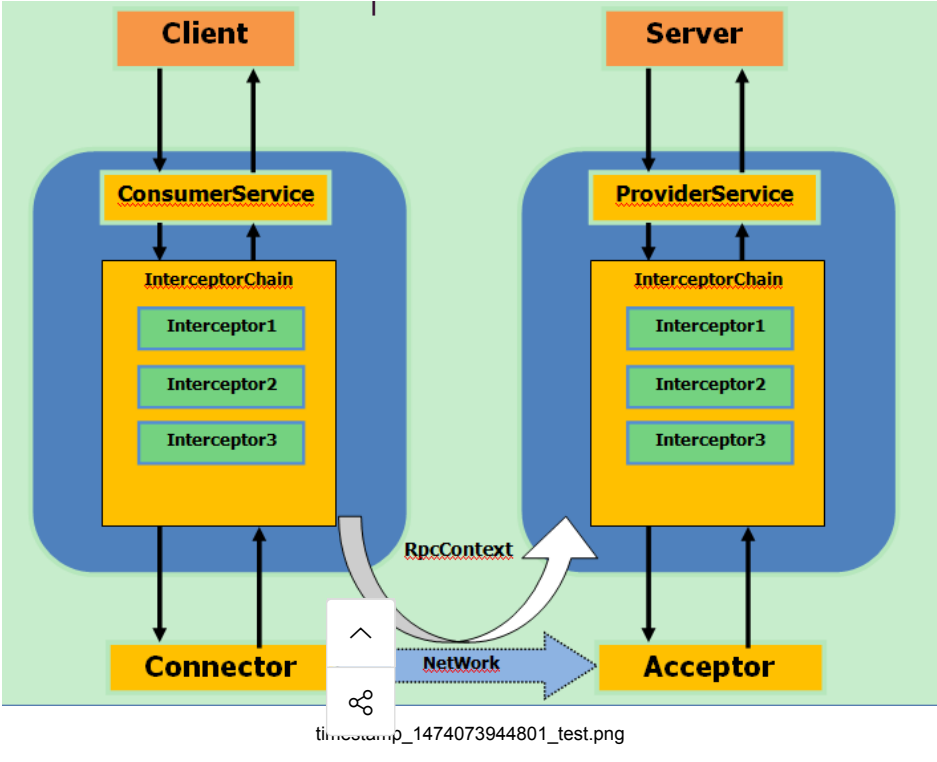
TopGun_Viper (/u/aee2b91e3398)

+ 关注

2016.11.05 15:13 字数 2149 阅读 34 评论 0 喜欢 0

(/u/aee2b91e3398)

首先再次贴出我们最初的预期各个模块的交互图：



目前rpc的基本功能都已经能够实现，下面我们加点新的需求：

- 用户可以添加自定义拦截器，实现rpc层面的AOP功能。

1.概述

首先，拦截器的作用很强大，它提供RPC层面的AOP功能，比如进行日志记录，rpc调用次数统计，service提供方添加白名单等等。在平时的学习过程中，很多框架都会实现这样的功能，比如：Servlet中的Filter，Struts中的Interceptor，Netty中的PipelineChannel和ChannelHandler，Tomcat中的Realm等等。这其实是软件可扩展性的一种体现，那么我们的rpc框架必须也要紧跟时代潮流！

2.实现方案

虽然不同框架对于这一功能的实现略有不同，但是思路基本一致：责任链+命令模式。

责任链模式和命令模式就不介绍了，网上很多，大家可以自行baidu or google

基本思路

首先，责任链模式的最大作用就是可以让**请求**在链上传播，那么链上每个节点都可以对请求进行处理。这里的请求是什么？当然就是一次rpc调用了，它肯定包含调用的方法名，调用方法参数列表、参数值、调用方法所属的类等等信息。可以看到，这里的方法的调用者invoker，和方法调用的接受者receiver并不是直接交互的，这就很容易想到了命令模式，命令模式不正是将调用者和接受者进行解耦吗？哈哈。。。大体思路有了，那么就开搞吧！

下面是设计到的主要类：

- MethodInvocation：代表每次方法调用
- RpcMethodInvocation：MethodInvocation子接口，代表一次rpc调用
- DefaultMethodInvocation：MethodInvocation的默认实现
- Interceptor：拦截器接口
- InterceptorChain：Interceptor容器，其实叫InterceptorManager貌似更合适。

MethodInvocation和RpcMethodInvocation的关系：在MethodInvocation接口的基础上又实现了一个子接口RpcMethodInvocation，这是考虑到可能存在injvm调用的情况，这样设计更加合理。

下面是具体实现：

```
/**
 * 抽象出方法的执行
 *
 * @author wqx
 */
public interface MethodInvocation {

    /**
     * 获取方法
     *
     * @return
     */
    public Method getMethod();

    /**
     *
     * @return
     */
    public Object[] getParameters();

    /**
     * invoke next Interceptor
     *
     * @return
     * @throws Exception
     */
    Object executeNext() throws Exception;
}

/**
 * RpcMethodInvocation
 *
 * @author wqx
 */
public interface RpcMethodInvocation extends MethodInvocation {

    /**
     * 是否是rpc调用
     *
     * @return
     */
    public boolean isRpcInvocation();
}
```

下面看下MethodInvocation的默认实现类：DefaultMethodInvocation

```
public class DefaultMethodInvocation implements RpcMethodInvocation {

    private Object target;
    private Object proxy;
    private Method method;
    private Object[] parameters;

    private boolean isRpcInvocation;

    //拦截器
    private List<Interceptor> interceptors;

    //当前Interceptor索引值，初始值：-1，范围：0-interceptors.size()-1
    private int currentIndex = -1;

    public DefaultMethodInvocation(Object target, Object proxy, Method method, Object[] parameters, List<Interceptor> interceptors){
        this.target = target;
        this.proxy = proxy;
        this.method = method;
        this.parameters = parameters;
        this.interceptors = interceptors;
    }

    @Override
    public Object executeNext() throws Exception {
        if(this.currentIndex == this.interceptors.size() - 1){
            if(this.isRpcInvocation){
                RpcRequest request = new RpcRequest(target.getClass().getName(), method.getName(), method.getParameterTypes(), parameters, RpcContext.getAttributes());
                return ((RpcConsumer)target).sendRequest(request);
            }else{
                method.setAccessible(true);
                return method.invoke(target, parameters);
            }
        }
        Object interceptor = this.interceptors.get(++this.currentIndex);
        return ((Interceptor)interceptor).intercept(this);
    }

    //getter setter...

    @Override
    public boolean isRpcInvocation() {
        return isRpcInvocation;
    }

}
```

InterceptorChain中维护了一个List类型的拦截器列表，还有一个指向当前拦截器的索引变量currentIndex，下面重点看下executeNext方法。

首先，判断拦截器链是否执行完，如下：

```
1. if(this.currentIndex == this.interceptors.size() - 1){}
```

如果拦截器链没有执行完，那么进行2,3 两句，思路很简单，获取下一个拦截器对象，并将索引值加1，然后调用下一个拦截器的intercept方法。

```
2.Object interceptor = this.interceptors.get(++this.currentIndex);
```

```
3.return ((Interceptor)interceptor).intercept(this);
```

如果拦截器链已经执行完了，那么需要判断isRpcInvocation的值：

- isRpcInvocation=true：说明这是客户端的调用，将方法调用信息封装到RpcRequest

中，通过sendRequest发送到server端。

- isRpcInvocation=false：这是server端本地调用，通过反射执行方法即可。

```
if(this.isRpcInvocation){
    RpcRequest request = new RpcRequest(target.getClass().getName(), method.getName(),method.getParameterTypes(),parameters,RpcContext.getAttributes());
    return ((RpcConsumer)target).sendRequest(request);
}else{
    method.setAccessible(true);
    return method.invoke(target, parameters);
}
```

感觉很不好的实现方式，各种角色耦合度太高了！！！以后肯定需要改进！！！！

MethodInvocation设计完后，再看看拦截器Interceptor和拦截器链InterceptorChain的设计思路，源码如下：

```
public interface Interceptor {

    Object intercept(MethodInvocation invocation) throws Exception;

}

public class InterceptorChain {

    private List<Entry> interceptors;

    private Set<String> registeredNames;

    public InterceptorChain(){
        interceptors = new ArrayList<Entry>();
        registeredNames = new HashSet<String>();
    }

    public void addLast(String name, Interceptor interceptor){
        synchronized(this){
            checkDuplicateName(name);
            Entry entry = new Entry(name,interceptor);
            register(interceptors.size(), entry);
        }
    }

    public void addFirst(String name, Interceptor interceptor){
        synchronized(this){
            checkDuplicateName(name);
            Entry entry = new Entry(name,interceptor);
            register(0, entry);
        }
    }

    public void addBefore(String baseName, String name, Interceptor interceptor){
        synchronized(this){
            checkDuplicateName(name);
            int index = getInterceptorIndex(baseName);
            if(index == -1)
                throw new NoSuchElementException(baseName);
            Entry entry = new Entry(name,interceptor);
            register(index, entry);
        }
    }

    public void addAfter(String baseName, String name, Interceptor interceptor){
        synchronized(this){
            checkDuplicateName(name);
            int index = getInterceptorIndex(baseName);
            if(index == -1)
                throw new NoSuchElementException(baseName);
            Entry entry = new Entry(name,interceptor);
            register(index+1, entry);
        }
    }

    private int getInterceptorIndex(String name) {
        List<Entry> interceptors = this.interceptors;
        for(int i = 0; i < interceptors.size(); i++){
            if(interceptors.get(i).name.equals(name)){
                return i;
            }
        }
        return -1;
    }

}
```

```
private void register(int index, Entry entry){
    interceptors.add(index, entry);
    registeredNames.add(entry.name);
}
private void checkDuplicateName(String name) {
    if (registeredNames.contains(name)) {
        throw new IllegalArgumentException("Duplicate interceptor name: " + name);
    }
}
@SuppressWarnings("unchecked")
public List<Interceptor> getInterceptors() {
    if(!CollectionUtils.isEmpty(this.interceptors)){
        List<Interceptor> list = new ArrayList<>(this.interceptors.size());
        for(Entry entry: this.interceptors){
            list.add(entry.interceptor);
        }
        return Collections.unmodifiableList(list);
    }else{
        return (List<Interceptor>) CollectionUtils.EMPTY_COLLECTION;
    }
}
static class Entry{
    String name;
    Interceptor interceptor;

    public Entry(String name, Interceptor interceptor) {
        super();
        this.name = name;
        this.interceptor = interceptor;
    }
}
}
```

首先看InterceptorChain的成员变量：

```
private List<Entry> interceptors;
private Set<String> registeredNames;
```

- 第一个变量是List<Entry>类型，代表整个拦截器链，不过并没有直接存储Interceptor对象，而是将interceptor和其对应的name封装成Entry对象。
- 第二个变量是Set<String>类型，用处很简单，保存已经注册过的拦截器，防止重复注册。

下面在分析下InterceptorChain的构造函数和成员方法：

```
public InterceptorChain(){
    interceptors = new ArrayList<Entry>();
    registeredNames = new HashSet<String>();
}
```

从构造函数可以确定上述两个成员变量的类型分别是：ArrayList和HashSet。

主要成员函数：

- addLast
- addFirst
- addBefore
- addAfter

上面四个函数是InterceptorChain提供给用户使用的主要API，实现很简单，无非就是向List中的指定位置增加元素罢了。下面重点分析下InterceptorChain的线程安全性。

InterceptorChain是线程安全的吗？

要回答某个类是否是线程安全的问题，首先看实例化对象是否有状态的？**无状态的对象一定是线程安全的**，InterceptorChain显然是有状态的：有interceptors和registeredNames两个成员变量。若有状态，再分析下该类是否会存在与并发环境中，如

果只可能被单线程操作，那么我们就无需考虑线程安全问题了。InterceptorChain明显可能被多个线程同时操作，所以可能存在于并发环境中。好了，既然这样就想想如何解决？在Java中遇到并发问题是很常见的，解决思路其实无非就这么几种：

- 加锁（典型解决方案：synchronized、ReentrantLock等）
- volatile
- 使用现成的并发容器（ConcurrentHashMap，CopyOnWriteArrayList等）

基本思路都是**并发操作串行化**，那么这几种方案该如何选择呢？首先，当我们分析一个类的线程安全性的时候，需要找出可能改变类自身状态的操作，然后分析每个操作在多线程环境中是否会造成状态的不一致性。对于InterceptorChain来说，前面提到的

- addLast
- addFirst
- addBefore
- addAfter

这四个方法显然会改变其自身的状态，那么拿addAfter来具体分析下：

```
public void addAfter(String baseName, String name, Interceptor interceptor){
    synchronized(this){
        checkDuplicateName(name);
        int index = getInterceptorIndex(baseName);
        if(index == -1)
            throw new NoSuchElementException(baseName);
        Entry entry = new Entry(name,interceptor);
        register(index+1, entry);
    }
}
```

addAfter的逻辑很简单，首先检查待注册的Interceptor的name是否已经存在，如果不存在，那么获取拦截器baseName的索引，如果不存在抛出NoSuchElementException异常，如果存在，将待注册拦截器插入到baseName之后，下面看下注册操作register：

```
private void register(int index, Entry entry){
    interceptors.add(index, entry);
    registeredNames.add(entry.name);
}
```

可见，从checkDuplicateName到最后注册操作register，这是一系列复合操作集合。如果不进行任何额外的同步操作，就会出现错误（两个相同name的拦截器可能被注册了），加了synchronized保证了对interceptors和registeredNames操作的串行化，保证了InterceptorChain的线程安全性。

InterceptorChain是线程安全的，那么用户自定义的Interceptor是不是线程安全的呢？答案是：不知道！！一个拦截器可能被加入多个拦截器链，所以拦截器的线程安全性只能通过用户来保证！当然了，用户自定义的类理应由用户负责。

下面进行测试

用户自定义拦截器TimeInterceptor，client端用来记录rpc调用时间

```
/**
 * TimeInterceptor: 记录方法执行时间
 *
 * @author wqx
 */
public class TimeInterceptor implements Interceptor {

    @Override
    public Object intercept(MethodInvocation invocation) throws Exception {
        long start = System.currentTimeMillis();
        Object retVal = null;
        try{
            retVal = invocation.executeNext();
        }finally{
            long end = System.currentTimeMillis();
            System.out.println("execute time :" + (end-start) + "ms.");
            RpcContext.addAttribute("time", (end-start));
        }
        return retVal;
    }
}
```

服务端自定义拦截器MethodFilterInterceptor：

```
public class MethodFilterInterceptor implements Interceptor {

    private Set<String> exclusions = new HashSet<>();

    public MethodFilterInterceptor(Set<String> exclusions){
        this.exclusions = exclusions;
    }
    @Override
    public Object intercept(MethodInvocation invocation) throws Exception {
        String methodName = invocation.getMethod().getName();
        if(exclusions.contains(methodName)){
            throw new RpcException("method " + methodName + " is not allowed!");
        }
        return invocation.executeNext();
    }
}
```

MethodFilterInterceptor主要进行黑名单过滤，如果client端调用的方法在exclusions集合中，那么直接抛出RpcException。下面看下客户端和服务端测试代码：

```
server端:
public class ServerTest {

    private static int port = 8888;

    public static void main(String[] args) throws IOException {
        UserService userService = new UserServiceImpl();

        Set<String> exclusions = new HashSet<>();
        exclusions.add("getMap");

        //新增拦截器MethodFilterInterceptor
        RpcProvider.interceptorChain.addLast("methodFilter",
            new MethodFilterInterceptor(exclusions));

        RpcProvider.publish(userService, port);
    }
}

client端:
public class ClientTest {

    private static RpcConsumer consumer;
    static {
        consumer = new RpcConsumer();

        //客户端添加记录rpc调用时间的拦截器
        consumer.getInterceptorChain().addLast("time", new TimeInterceptor());

        userService = (UserService)consumer.targetHostPort(host, port)
            .interfaceClass(UserService.class)
            .timeout(TIMEOUT)
            .newProxy();
    }
    @Test
    public void testInterceptorChain(){
        try{
            Map<String, Object> resultMap = userService.getMap();
        }catch(Exception e){
            System.out.println(e);
            Assert.assertEquals("method getMap is not allowed!", e.getMessage());
        }
    }
}
```

执行结果：

```
execute time :103ms.
```

```
edu.ouc.rpc.model.RpcException: method getMap is not allowed!
```

符合预期，bingo！！！！

总结

想想以前的学习方式，真的是误入歧途！

1. 所谓设计模式不过是解决某一领域问题的最佳实践，问题的多样化带来的肯定是解决方案的多样化，不要为了模式而模式，需要根据具体问题具体对待。
2. 实践出真知！！！！对于一个IT工程师更是如此！
3. 重构！用心重构！这是自我提升的必然道路。
4. 不要过于在乎写了多少行代码，做了多少个项目。一个自称有十年开发经验的人，谁能保证他不是一年经验用了十年呢？

完整源码 (https://github.com/TopGunViper/rpc-race/tree/v1.0.1_aop)



TopGun_Viper (/u/aee2b91e3398)

写了 23557 字，被 15 人关注，获得了 29 个喜欢 (/u/aee2b91e3398)


+ 关注


吾日三省吾code，可以为师矣。。。


如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持

♡ 喜欢 (/sign_in) | 0







更多分享

(http://cwb.assets.jianshu.io/notes/images/6769511



登录 (/sign_in) 后发表评论

评论

智慧如你，不想发表一点想法 (/sign_in)咩~

被以下专题收入，发现更多相似内容

- Java设计模式
- Java基础
- JAVA开发
- 程序员