crazyhzm 265

Dubbo源码解析（二十）远程调用——Filter

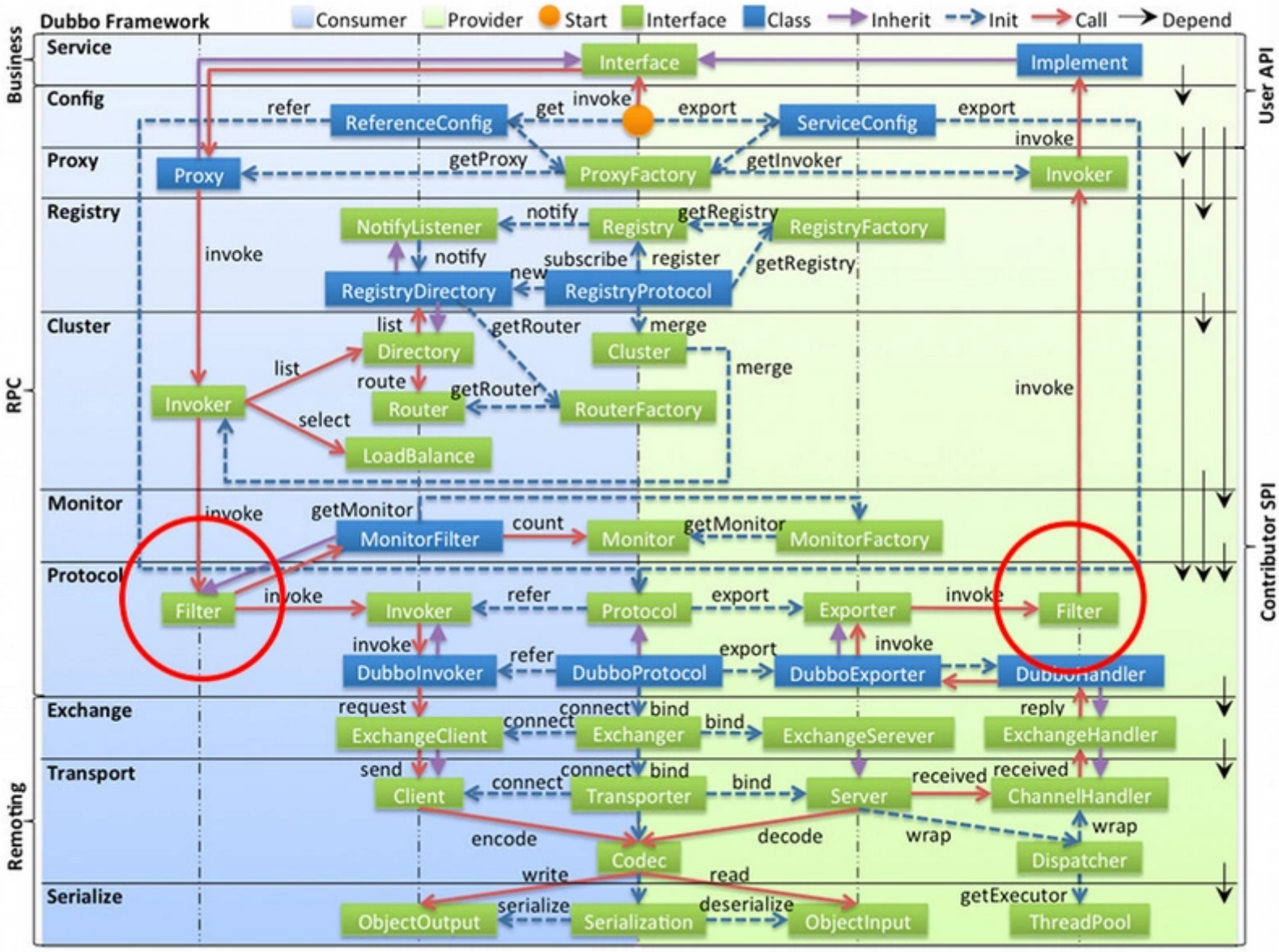
java dubbo filter 阅读约 88 分钟

远程调用——Filter

> 目标：介绍dubbo-rpc-api中的各种filter过滤器的实现逻辑。

前言

本文会介绍在dubbo中的过滤器，先来看看下面的图：



可以看到红色圈圈不服，在服务发现和服务引用中都会进行一些过滤器过滤。具体有哪些过滤器，就看下面的介绍。

源码分析

（一）AccessLogFilter

该过滤器是对记录日志的过滤器，它所做的工作就是把引用服务或者暴露服务的调用链信息写入到文件中。日志消息先被放入日志集合，然后加入到日志队列，然后被放入到写入文件到任务中，最后进入文件。

1.属性

```
private static final Logger logger = LoggerFactory.getLogger(AccessLogFilter.class);

/**
 * 日志访问名称，默认的日志访问名称
 */
private static final String ACCESS_LOG_KEY = "dubbo.accesslog";

/**
 * 日期格式
 */
private static final String FILE_DATE_FORMAT = "yyyyMMdd";

private static final String MESSAGE_DATE_FORMAT = "yyyy-MM-dd HH:mm:ss";

/**
 * 日志队列大小
 */
private static final int LOG_MAX_BUFFER = 5000;

/**
 * 日志输出的频率
 */
private static final long LOG_OUTPUT_INTERVAL = 5000;

/**
 * 日志队列 key为访问日志的名称，value为该日志名称对应的日志集合
```

按照我上面讲到日志流向，日志先进入到是日志队列中的日志集合，再进入logQueue，在进入logFuture，最后落地到文件。

2.init

```
private void init() {
    // synchronized是一个重操作消耗性能，所有加上判空
    if (logFuture == null) {
        synchronized (logScheduled) {
            // 为了不重复初始化
            if (logFuture == null) {
                // 创建日志记录任务
                logFuture = logScheduled.scheduleWithFixedDelay(new LogTask(), LOG_OUTPUT_INTERVAL,
LOG_OUTPUT_INTERVAL, TimeUnit.MILLISECONDS);
            }
        }
    }
}
```

该方法是初始化方法，就创建了日志记录任务。

3.log

```
private void log(String accesslog, String logmessage) {
    init();
    Set<String> logSet = logQueue.get(accesslog);
    if (logSet == null) {
        logQueue.putIfAbsent(accesslog, new ConcurrentHashSet<String>());
        logSet = logQueue.get(accesslog);
    }
    if (logSet.size() < LOG_MAX_BUFFER) {
        logSet.add(logmessage);
    }
}
```

该方法是增加日志信息到日志集合中。

4.invoke

```
@Override
public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {
    try {
        // 获得日志名称
        String accesslog = invoker.getUrl().getParameter(Constants.ACCESS_LOG_KEY);
        if (ConfigUtils.isNotEmpty(accesslog)) {
            // 获得rpc上下文
            RpcContext context = RpcContext.getContext();
            // 获得调用的接口名称
            String serviceName = invoker.getInterface().getName();
            // 获得版本号
            String version = invoker.getUrl().getParameter(Constants.VERSION_KEY);
            // 获得组，是消费者侧还是生产者侧
            String group = invoker.getUrl().getParameter(Constants.GROUP_KEY);
            StringBuilder sn = new StringBuilder();
            sn.append("[").append(new SimpleDateFormat(MESSAGE_DATE_FORMAT).format(new Date())).append("]
").append(context.getRemoteHost()).append(":").append(context.getRemotePort())
                    .append(" -> ").append(context.getLocalHost()).append(":").append(context.getLocalPort())
                    .append(" - ");
            // 拼接组
            if (null != group && group.length() > 0) {
                sn.append(group).append("/");
            }
            // 拼接服务名称
            sn.append(serviceName);
            // 拼接版本号
```

该方法是最重要的方法，从拼接了日志信息，把日志加入到集合，并且调用下一个调用链。

4.LogTask

```
private class LogTask implements Runnable {
    @Override
    public void run() {
        try {
            if (logQueue != null && logQueue.size() > 0) {
                // 遍历日志队列
                for (Map.Entry<String, Set<String>> entry : logQueue.entrySet()) {
                    try {
                        // 获得日志名称
                        String accesslog = entry.getKey();
                        // 获得日志集合
                        Set<String> logSet = entry.getValue();
                        // 如果文件不存在则创建文件
                        File file = new File(accesslog);
                        File dir = file.getParentFile();
                        if (null != dir && !dir.exists()) {
                            dir.mkdirs();
                        }
                        if (logger.isDebugEnabled()) {
                            logger.debug("Append log to " + accesslog);
                        }
                        if (file.exists()) {
                            // 获得现在的时间
                            String now = new SimpleDateFormat(FILE_DATE_FORMAT).format(new Date());
                            // 获得文件最后一次修改的时间
                            String last = new SimpleDateFormat(FILE_DATE_FORMAT).format(new
```

该内部类实现了Runnable，是把日志消息落地到文件到线程。

（二）ActiveLimitFilter

该类时对于每个服务的每个方法的最大可并行调用数量限制的过滤器，它是在服务消费者侧的过滤。

```java
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得url对象
    URL url = invoker.getUrl();
    // 获得方法名称
    String methodName = invocation.getMethodName();
    // 获得并发调用数（单个服务的单个方法），默认为0
    int max = invoker.getUrl().getMethodParameter(methodName, Constants.ACTIVES_KEY, 0);
    // 通过方法名来获得对应的状态
    RpcStatus count = RpcStatus.getStatus(invoker.getUrl(), invocation.getMethodName());
    if (max > 0) {
        // 获得该方法调用的超时次数
        long timeout = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.TIMEOUT_KEY, 0);
        // 获得系统时间
        long start = System.currentTimeMillis();
        long remain = timeout;
        // 获得该方法的调用数量
        int active = count.getActive();
        // 如果活跃数量大于等于最大的并发调用数量
        if (active >= max) {
            synchronized (count) {
                // 当活跃数量大于等于最大的并发调用数量时一直循环
                while ((active = count.getActive()) >= max) {
                    try {
                        // 等待超时时间
                        count.wait(remain);
```

该类只有这一个方法。该过滤器是用来限制调用数量，先进行调用数量的检测，如果没有到达最大的调用数量，则先调用后面的调用链，如果在后面的调用链失败，则记录相关时间，如果成功也记录相关时间和调用次数。

（三）ClassLoaderFilter

该过滤器是做类加载器切换的。

```java
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得当前的类加载器
    ClassLoader ocl = Thread.currentThread().getContextClassLoader();
    // 设置invoker携带的服务的类加载器
    Thread.currentThread().setContextClassLoader(invoker.getInterface().getClassLoader());
    try {
        // 调用下面的调用链
        return invoker.invoke(invocation);
    } finally {
        // 最后切换回原来的类加载器
        Thread.currentThread().setContextClassLoader(ocl);
    }
}
```

可以看到先切换成当前的线程锁携带的类加载器，然后调用结束后，再切换回原先的类加载器。

（四）CompatibleFilter

该过滤器是做兼容性的过滤器。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 调用下一个调用链
    Result result = invoker.invoke(invocation);
    // 如果方法前面没有$或者结果没有异常
    if (!invocation.getMethodName().startsWith("$") && !result.hasException()) {
        Object value = result.getValue();
        if (value != null) {
            try {
                // 获得方法
                Method method = invoker.getInterface().getMethod(invocation.getMethodName(),
invocation.getParameterTypes());
                // 获得返回的数据类型
                Class<?> type = method.getReturnType();
                Object newValue;
                // 序列化方法
                String serialization = invoker.getUrl().getParameter(Constants.SERIALIZATION_KEY);
                // 如果是json或者fastjson形式
                if ("json".equals(serialization)
                        || "fastjson".equals(serialization)) {
                    // 获得方法的泛型返回值类型
                    Type gtype = method.getGenericReturnType();
                    // 把数据结果进行类型转化
                    newValue = PojoUtils.realize(value, type, gtype);
                    // 如果value不是type类型
                } else if (!type.isInstance(value)) {
```

可以看到对于调用链的返回结果，如果返回值类型和返回值不一样的时候，就需要做兼容类型的转化。重新把结果放入RpcResult，返回。

（五）ConsumerContextFilter

该过滤器做的是在当前的RpcContext中记录本地调用的一次状态信息。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 设置rpc上下文
    RpcContext.getContext()
            .setInvoker(invoker)
            .setInvocation(invocation)
            .setLocalAddress(NetUtils.getLocalHost(), 0)
            .setRemoteAddress(invoker.getUrl().getHost(),
                    invoker.getUrl().getPort());
    // 如果该会话域是rpc会话域
    if (invocation instanceof RpcInvocation) {
        // 设置实体域
        ((RpcInvocation) invocation).setInvoker(invoker);
    }
    try {
        // 调用下个调用链
        RpcResult result = (RpcResult) invoker.invoke(invocation);
        // 设置附加值
        RpcContext.getServerContext().setAttachments(result.getAttachments());
        return result;
    } finally {
        // 情况附加值
        RpcContext.getContext().clearAttachments();
    }
}
```

可以看到RpcContext记录了一次调用状态信息，然后先调用后面的调用链，再回来把附加值设置到RpcContext中。然后返回RpcContext，再清空，这样是因为后面的调用链中的附加值对前面的调用链是不可见的。

（六）ContextFilter

该过滤器做的是初始化rpc上下文。

```
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        // 获得会话域的附加值
        Map<String, String> attachments = invocation.getAttachments();
        // 删除异步属性以避免传递给以下调用链
        if (attachments != null) {
            attachments = new HashMap<String, String>(attachments);
            attachments.remove(Constants.PATH_KEY);
            attachments.remove(Constants.GROUP_KEY);
            attachments.remove(Constants.VERSION_KEY);
            attachments.remove(Constants.DUBBO_VERSION_KEY);
            attachments.remove(Constants.TOKEN_KEY);
            attachments.remove(Constants.TIMEOUT_KEY);
            attachments.remove(Constants.ASYNC_KEY);// Remove async property to avoid being passed to the
following invoke chain.
        }
        // 在rpc上下文添加上一个调用链的信息
        RpcContext.getContext()
                .setInvoker(invoker)
                .setInvocation(invocation)
//                .setAttachments(attachments)  // merged from dubbox
                .setLocalAddress(invoker.getUrl().getHost(),
                        invoker.getUrl().getPort());

        // mreged from dubbox
        // we may already added some attachments into RpcContext before this filter (e.g. in rest protocol)
```

在[《 dubbo源码解析（十九）远程调用——开篇》]中我已经介绍了RpcContext的作用，角色。该过滤器就是做了初始化RpcContext的作用。

（七）DeprecatedFilter

该过滤器的作用是调用了废弃的方法时打印错误日志。

```
private static final Logger LOGGER = LoggerFactory.getLogger(DeprecatedFilter.class);

/**
 * 日志集合
 */
private static final Set<String> logged = new ConcurrentHashSet<String>();

@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得key 服务+方法
    String key = invoker.getInterface().getName() + "." + invocation.getMethodName();
    // 如果集合中没有该key
    if (!logged.contains(key)) {
        // 则加入集合
        logged.add(key);
        // 如果该服务方法是废弃的，则打印错误日志
        if (invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.DEPRECATED_KEY, false)) {
            LOGGER.error("The service method " + invoker.getInterface().getName() + "." +
getMethodSignature(invocation) + " is DEPRECATED! Declare from " + invoker.getUrl());
        }
    }
    // 调用下一个调用链
    return invoker.invoke(invocation);
}

/**
```

该过滤器比较简单。

（八）EchoFilter

该过滤器是处理回声测试的方法。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {
    // 如果调用的方法是回声测试的方法 则直接返回结果，否则 调用下一个调用链
    if (inv.getMethodName().equals(Constants.$ECHO) && inv.getArguments() != null && inv.getArguments().length == 1)
        return new RpcResult(inv.getArguments()[0]);
    return invoker.invoke(inv);
}
```

如果调用的方法是回声测试的方法 则直接返回结果，否则 调用下一个调用链。

（九）ExceptionFilter

该过滤器是作用是对异常的处理。

```
private final Logger logger;

public ExceptionFilter() {
    this(LoggerFactory.getLogger(ExceptionFilter.class));
}

public ExceptionFilter(Logger logger) {
    this.logger = logger;
}

@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    try {
        // 调用下一个调用链，返回结果
        Result result = invoker.invoke(invocation);
        // 如果结果有异常，并且该服务不是一个泛化调用
        if (result.hasException() && GenericService.class != invoker.getInterface()) {
            try {
                // 获得异常
                Throwable exception = result.getException();

                // directly throw if it's checked exception
                // 如果这是一个checked的异常，则直接返回异常，也就是接口上声明的Unchecked的异常
                if (!(exception instanceof RuntimeException) && (exception instanceof Exception)) {
                    return result;
                }
```

可以看到除了接口上声明的Unchecked的异常和有定义的异常外，都会包装成RuntimeException来返回，为了防止客户端反序列化失败。

（十）ExecuteLimitFilter

该过滤器是限制最大可并行执行请求数，该过滤器是服务提供者侧，而上述讲到的ActiveLimitFilter是在消费者侧的限制。

```
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        // 获得url对象
        URL url = invoker.getUrl();
        // 方法名称
        String methodName = invocation.getMethodName();
        Semaphore executesLimit = null;
        boolean acquireResult = false;
        int max = url.getMethodParameter(methodName, Constants.EXECUTES_KEY, 0);
        // 如果该方法设置了executes并且值大于0
        if (max > 0) {
            // 获得该方法对应的RpcStatus
            RpcStatus count = RpcStatus.getStatus(url, invocation.getMethodName());
//            if (count.getActive() >= max) {
            /**
             * http://manzhizhen.iteye.com/blog/2386408
             * use semaphore for concurrency control (to limit thread number)
             */
            // 获得信号量
            executesLimit = count.getSemaphore(max);
            // 如果不能获得许可，则抛出异常
            if(executesLimit != null && !(acquireResult = executesLimit.tryAcquire())) {
                throw new RpcException("Failed to invoke method " + invocation.getMethodName() + " in provider "
+ url + ", cause: The service using threads greater than <dubbo:service executes=\"" + max + "\" /> limited.");
            }
        }
```

为什么这里需要用到信号量来控制，可以看一下以下链接的介绍：http://manzhizhen.iteye.com/b...

（十一）GenericFilter

该过滤器就是对于泛化调用的请求和结果进行反序列化和序列化的操作，它是服务提供者侧的。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {
    // 如果是泛化调用
    if (inv.getMethodName().equals(Constants.$INVOKE)
            && inv.getArguments() != null
            && inv.getArguments().length == 3
            && !invoker.getInterface().equals(GenericService.class)) {
        // 获得请求名字
        String name = ((String) inv.getArguments()[0]).trim();
        // 获得请求参数类型
        String[] types = (String[]) inv.getArguments()[1];
        // 获得请求参数
        Object[] args = (Object[]) inv.getArguments()[2];
        try {
            // 获得方法
            Method method = ReflectUtils.findMethodByMethodSignature(invoker.getInterface(), name, types);
            // 获得该方法的参数类型
            Class<?>[] params = method.getParameterTypes();
            if (args == null) {
                args = new Object[params.length];
            }
            // 获得附加值
            String generic = inv.getAttachment(Constants.GENERIC_KEY);

            // 如果附加值为空，在用上下文携带的附加值
            if (StringUtils.isBlank(generic)) {
```

（十二）GenericImplFilter

该过滤器也是对于泛化调用的序列化检查和处理，它是消费者侧的过滤器。

```
private static final Logger logger = LoggerFactory.getLogger(GenericImplFilter.class);

/**
 * 参数集合
 */
private static final Class<?>[] GENERIC_PARAMETER_TYPES = new Class<?>[]{String.class, String[].class,
Object[].class};

@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得泛化的值
    String generic = invoker.getUrl().getParameter(Constants.GENERIC_KEY);
    // 如果该值是nativejava或者bean或者true，并且不是一个返回调用
    if (ProtocolUtils.isGeneric(generic)
            && !Constants.$INVOKE.equals(invocation.getMethodName())
            && invocation instanceof RpcInvocation) {
        RpcInvocation invocation2 = (RpcInvocation) invocation;
        // 获得方法名称
        String methodName = invocation2.getMethodName();
        // 获得参数类型集合
        Class<?>[] parameterTypes = invocation2.getParameterTypes();
        // 获得参数集合
        Object[] arguments = invocation2.getArguments();

        // 把参数类型的名称放入集合
        String[] types = new String[parameterTypes.length];
```

（十三）TimeoutFilter

该过滤器是当服务调用超时的时候，记录告警日志。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得开始时间
    long start = System.currentTimeMillis();
    // 调用下一个调用链
    Result result = invoker.invoke(invocation);
    // 获得调用使用的时间
    long elapsed = System.currentTimeMillis() - start;
    // 如果服务调用超时，则打印告警日志
    if (invoker.getUrl() != null
            && elapsed > invoker.getUrl().getMethodParameter(invocation.getMethodName(),
            "timeout", Integer.MAX_VALUE)) {
        if (logger.isWarnEnabled()) {
            logger.warn("invoke time out. method: " + invocation.getMethodName()
                    + " arguments: " + Arrays.toString(invocation.getArguments()) + " , url is "
                    + invoker.getUrl() + ", invoke elapsed " + elapsed + " ms.");
        }
    }
    return result;
}
```

（十四）TokenFilter

该过滤器提供了token的验证功能，关于token的介绍可以查看官方文档。

```
@Override
public Result invoke(Invoker<?> invoker, Invocation inv)
        throws RpcException {
    // 获得token值
    String token = invoker.getUrl().getParameter(Constants.TOKEN_KEY);
    if (ConfigUtils.isNotEmpty(token)) {
        // 获得服务类型
        Class<?> serviceType = invoker.getInterface();
        // 获得附加值
        Map<String, String> attachments = inv.getAttachments();
        String remoteToken = attachments == null ? null : attachments.get(Constants.TOKEN_KEY);
        // 如果令牌不一样，则抛出异常
        if (!token.equals(remoteToken)) {
            throw new RpcException("Invalid token! Forbid invoke remote service " + serviceType + " method " +
inv.getMethodName() + "() from consumer " + RpcContext.getContext().getRemoteHost() + " to provider " +
RpcContext.getContext().getLocalHost());
        }
    }
    // 调用下一个调用链
    return invoker.invoke(inv);
}
```

（十五）TpsLimitFilter

该过滤器的作用是对TPS限流。

```
/**
 * TPS 限制器对象
 */
private final TPSLimiter tpsLimiter = new DefaultTPSLimiter();

@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {

    // 如果限流器不允许，则抛出异常
    if (!tpsLimiter.isAllowable(invoker.getUrl(), invocation)) {
        throw new RpcException(
                "Failed to invoke service " +
                        invoker.getInterface().getName() +
                        "." +
                        invocation.getMethodName() +
                        " because exceed max service tps.");
    }

    // 调用下一个调用链
    return invoker.invoke(invocation);
}
```

其中关键是TPS 限制器对象，请看下面的分析。

（十六）TPSLimiter

```
public interface TPSLimiter {

    /**
     * judge if the current invocation is allowed by TPS rule
     * 是否允许通过
     * @param url        url
     * @param invocation invocation
     * @return true allow the current invocation, otherwise, return false
     */
    boolean isAllowable(URL url, Invocation invocation);

}
```

该接口是tps限流器的接口，只定义了一个是否允许通过的方法。

（十七）StatItem

该类是统计的数据结构。

```
class StatItem {

    /**
     * 服务名
     */
    private String name;

    /**
     * 最后一次重置的时间
     */
    private long lastResetTime;

    /**
     * 周期
     */
    private long interval;

    /**
     * 剩余多少流量
     */
    private AtomicInteger token;

    /**
     * 限制大小
     */
    private int rate;
```

可以看到该类中记录了一些访问的流量，并且设置了周期重置机制。

（十八）DefaultTPSLimiter

该类实现了TPSLimiter，是默认的tps限流器实现。

```
public class DefaultTPSLimiter implements TPSLimiter {

    /**
     * 统计项集合
     */
    private final ConcurrentMap<String, StatItem> stats
            = new ConcurrentHashMap<String, StatItem>();

    @Override
    public boolean isAllowable(URL url, Invocation invocation) {
        // 获得tps限制大小，默认-1，不限制
        int rate = url.getParameter(Constants.TPS_LIMIT_RATE_KEY, -1);
        // 获得限流周期
        long interval = url.getParameter(Constants.TPS_LIMIT_INTERVAL_KEY,
                Constants.DEFAULT_TPS_LIMIT_INTERVAL);
        String serviceKey = url.getServiceKey();
        // 如果限制
        if (rate > 0) {
            // 从集合中获得统计项
            StatItem statItem = stats.get(serviceKey);
            // 如果为空，则新建
            if (statItem == null) {
                stats.putIfAbsent(serviceKey,
                        new StatItem(serviceKey, rate, interval));
                statItem = stats.get(serviceKey);
            }
```

是否允许的逻辑还是调用了统计项中的isAllowable方法。

本文介绍了很多的过滤器，哪些过滤器是在服务引用的，哪些服务器是服务暴露的，可以查看相应源码过滤器的实现上的注解，

例如ActiveLimitFilter上：

```
@Activate(group = Constants.CONSUMER, value = Constants.ACTIVES_KEY)
```

可以看到group为consumer组的，也就是服务消费者侧的，则是服务引用过程中的的过滤器。

例如ExecuteLimitFilter上：

```
@Activate(group = Constants.PROVIDER, value = Constants.EXECUTES_KEY)
```

可以看到group为provider组的，也就是服务提供者侧的，则是服务暴露过程中的的过滤器。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了在服务引用和服务暴露中的各种filter过滤器。接下来我将开始对rpc模块的监听器进行讲解。

crazyhzm
265

关注作者

2 条评论

得票 • 时间

撰写评论 ...

提交评论

一念花开 ： 可以看到group为provider组的，也就是服务消费者侧的，则是服务暴露过程中的的过滤器。
应该是服务提供者侧的吧

_ • 回复 • 9月24日

crazyhzm： 是的，感谢指正，我会修改它

_ • 回复 • 9月24日

推荐阅读

聊聊Dubbo - Dubbo可扩展机制源码解析

摘要： 在Dubbo可扩展机制实战中，我们了解了Dubbo扩展机制的一些概念，初探了Dubbo中LoadBalance的实现，并自己实现了...

猫耳 • 阅读 21

dubbo注册服务IP解析异常及IP解析源码分析

在使用dubbo注册服务时会遇到IP解析错误导致无法正常访问.比如:本机设置的IP为172.16.11.111,但实际解析出来的是180.20.174.1...

平常 • 阅读 37