crazyhzm 265

Dubbo源码解析（二十一）远程调用——Listener

java dubbo 阅读约 12 分钟

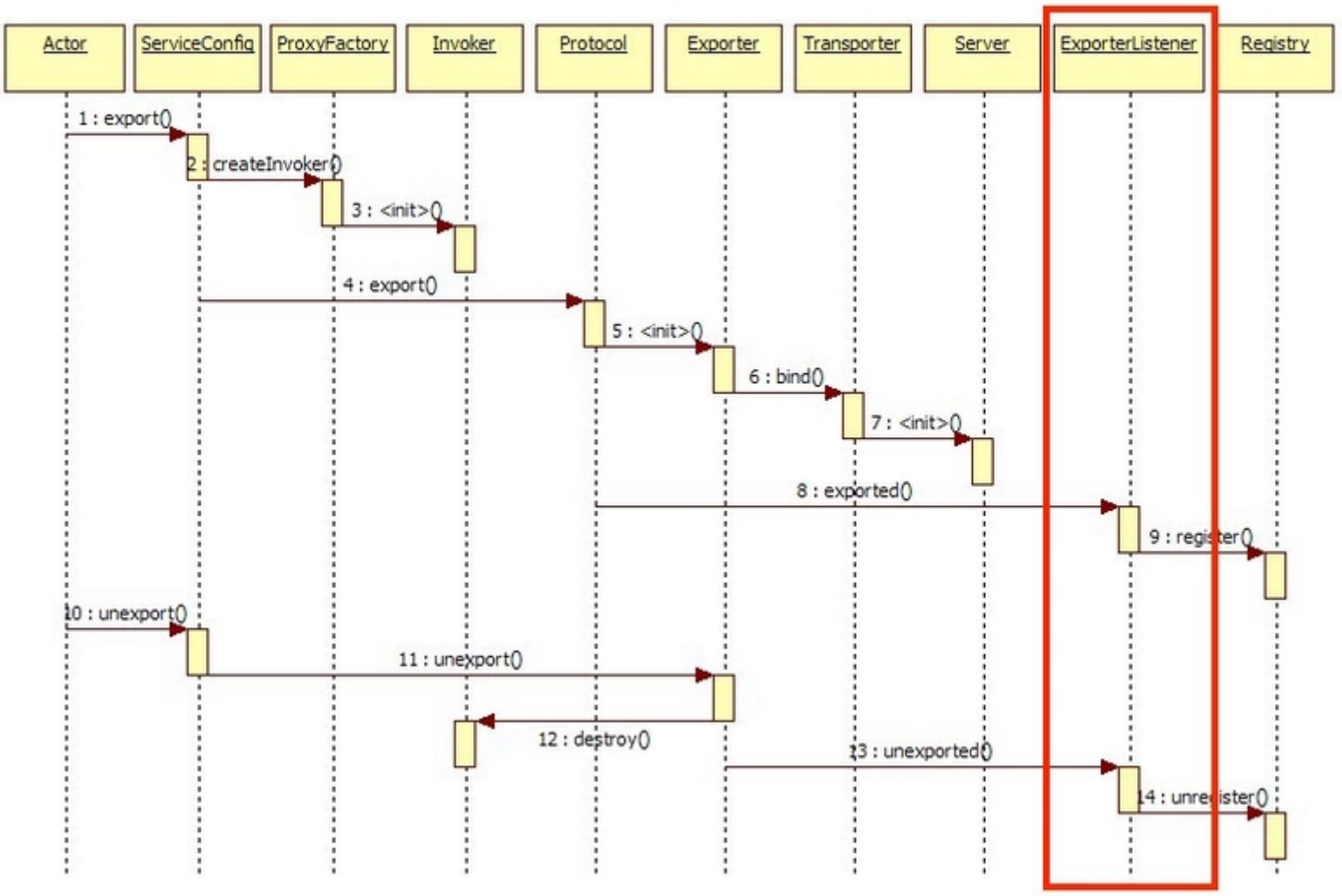
远程调用——Listener

> 目标：介绍dubbo-rpc-api中的各种listener监听器的实现逻辑，内容略少，随便撇两眼，不是重点。

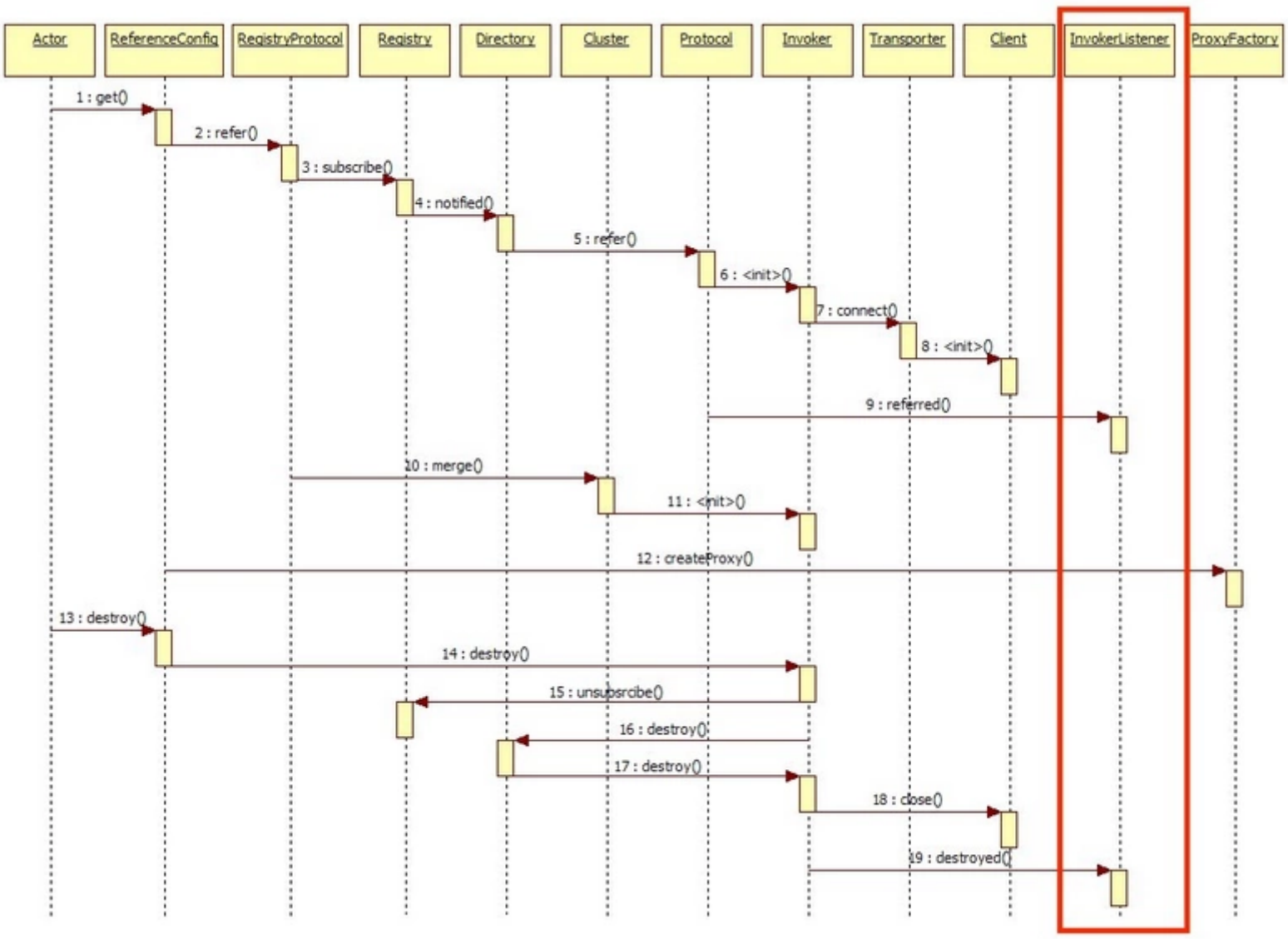
前言

本文介绍监听器的相关逻辑。在服务引用和服务发现中监听器处于的位置请看下面的图：

1. 服务暴露：



1. 服务引用：



这两个监听器所做的工作不是很多，来看看源码理解一下。

源码分析

（一）ListenerInvokerWrapper

该类实现了Invoker，是服务引用监听器的包装类。

1.属性

```
/**
 * invoker对象
 */
private final Invoker<T> invoker;

/**
 * 监听器集合
 */
private final List<InvokerListener> listeners;
```

用到了装饰模式，其中很多实现方法直接调用了invoker的方法。

2.构造方法

```
public ListenerInvokerWrapper(Invoker<T> invoker, List<InvokerListener> listeners) {
    // 如果invoker为空则抛出异常
    if (invoker == null) {
        throw new IllegalArgumentException("invoker == null");
    }
    this.invoker = invoker;
    this.listeners = listeners;
    if (listeners != null && !listeners.isEmpty()) {
        // 遍历监听器
        for (InvokerListener listener : listeners) {
            if (listener != null) {
                try {
                    // 调用在服务引用的时候进行监听
                    listener.referred(invoker);
                } catch (Throwable t) {
                    logger.error(t.getMessage(), t);
                }
            }
        }
    }
}
```

构造方法中直接调用了监听器的服务引用。

3.destroy

```
@Override
public void destroy() {
    try {
        // 销毁invoker
        invoker.destroy();
    } finally {
        // 销毁所有监听的实体域
        if (listeners != null && !listeners.isEmpty()) {
            for (InvokerListener listener : listeners) {
                if (listener != null) {
                    try {
                        listener.destroyed(invoker);
                    } catch (Throwable t) {
                        logger.error(t.getMessage(), t);
                    }
                }
            }
        }
    }
}
```

该方法是把服务引用的监听器销毁。

（二）InvokerListenerAdapter

```
public abstract class InvokerListenerAdapter implements InvokerListener {

    /**
     * 引用服务
     * @param invoker
     * @throws RpcException
     */
    @Override
    public void referred(Invoker<?> invoker) throws RpcException {
    }

    /**
     * 销毁
     * @param invoker
     */
    @Override
    public void destroyed(Invoker<?> invoker) {
    }

}
```

该类是服务引用监听器的适配类，没有做实际的操作。

(三) DeprecatedInvokerListener

```
@Activate(Constants.DEPRECATED_KEY)
public class DeprecatedInvokerListener extends InvokerListenerAdapter {

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

    @Override
    public void referred(Invoker<?> invoker) throws RpcException {
        // 当该引用的服务被废弃时，打印错误日志
        if (invoker.getUrl().getParameter(Constants.DEPRECATED_KEY, false)) {
            LOGGER.error("The service " + invoker.getInterface().getName() + " is DEPRECATED! Declare from " +
invoker.getUrl());
        }
    }

}
```

该类是当调用废弃的服务时候打印错误日志。

(四) ListenerExporterWrapper

该类是服务暴露监听器包装类。

1.属性

```
/**
 * 服务暴露者
 */
private final Exporter<T> exporter;

/**
 * 服务暴露监听者集合
 */
private final List<ExporterListener> listeners;
```

用到了装饰模式，其中很多实现方法直接调用了exporter的方法。

2.构造方法

```
public ListenerExporterWrapper(Exporter<T> exporter, List<ExporterListener> listeners) {
    if (exporter == null) {
        throw new IllegalArgumentException("exporter == null");
    }
    this.exporter = exporter;
    this.listeners = listeners;
    if (listeners != null && !listeners.isEmpty()) {
        RuntimeException exception = null;
        // 遍历服务暴露监听集合
        for (ExporterListener listener : listeners) {
            if (listener != null) {
                try {
                    // 暴露服务监听
                    listener.exported(this);
                } catch (RuntimeException t) {
                    logger.error(t.getMessage(), t);
                    exception = t;
                }
            }
        }
        if (exception != null) {
            throw exception;
        }
    }
}
```

该方法中对于每个服务暴露进行监听。

3.unexport

```
@Override
public void unexport() {
    try {
        // 取消暴露
        exporter.unexport();
    } finally {
        if (listeners != null && !listeners.isEmpty()) {
            RuntimeException exception = null;
            // 遍历监听集合
            for (ExporterListener listener : listeners) {
                if (listener != null) {
                    try {
                        // 监听取消暴露
                        listener.unexported(this);
                    } catch (RuntimeException t) {
                        logger.error(t.getMessage(), t);
                        exception = t;
                    }
                }
            }
            if (exception != null) {
                throw exception;
            }
        }
    }
}
```

该方法是对每个取消服务暴露的监听。

（五）ExporterListenerAdapter

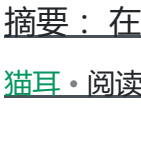
```java
public abstract class ExporterListenerAdapter implements ExporterListener {

    /**
     * 暴露服务
     * @param exporter
     * @throws RpcException
     */
    @Override
    public void exported(Exporter<?> exporter) throws RpcException {
    }

    /**
     * 取消暴露服务
     * @param exporter
     * @throws RpcException
     */
    @Override
    public void unexported(Exporter<?> exporter) throws RpcException {
    }

}
```

该类是服务暴露监听器的适配类，没有做实际的操作。

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listener监听器，其中内容很少。接下来我将开始对rpc模块的协议protocol进行讲解。



crazyhzm

265