Dubbo全链路追踪日志的实现

微服务架构的项目，一次请求可能会调用多个微服务，这样就会产生多个微服务的请求日志，当我们想要查看整个请求链路的日志时，就会变得困难，所幸的是我们有一些集中日志收集工具，比如很热门的ELK，我们需要把这些日志串联起来，这是一个很关键的问题，如果没有串联起来，查询起来很是很困难，我们的做法是在开始请求系统时生成一个全局唯一的id，这个id伴随这整个请求的调用周期，即当一个服务调用另外一个服务的时候，会往下传递，形成一条链路，当我们查看日志时，只需要搜索这个id，整条链路的日志都可以查出来了。

现在以dubbo微服务架构为背景，举个栗子：

```
A -> B -> C
```

我们需要将A/B/C/三个微服务间的日志按照链式打印，我们都知道Dubbo的RpcContext只能做到消费者和提供者共享同一个RpcContext，比如A->B，那么A和B都可以获取相同内容的RpcContext，但是B->C时，A和C就无法共享相同内容的RpcContext了，也就是无法做到链式打印日志了。

那么我们是如何做到呢?

我们可以用左手交换右手的思路来解决，假设左手是线程的ThreadLocal，右手是RpcContext，那么在交换之前，我们首先将必要的日志信息保存到ThreadLocal中。

在我们的项目微服务中大致分为两种容器类型的微服务，一种是Dubbo容器，这种容器的特点是只使用spring容器启动，然后使用dubbo进行服务的暴露，然后将服务注册到zookeeper，提供服务给消费者；另一种是SpringMVC容器，也即是我们常见的WEB容器，它是我们项目唯一可以对外开放接口的容器，也是充当项目的网关功能。

在了解了微服务容器之后，我们现在知道了调用链的第一层一定是在SpringMVC容器层中，那么我们直接在这层写个自定义拦截器就ojbk了，talk is cheap，show you the demo code：

举例一个Demo代码，公共拦截器的前置拦截中代码如下：

```
public class CommonInterceptor implements HandlerInterceptor {  @Override  public boolean p
  @Override
  public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse htt
    throws Exception {

    // ...

    // 初始化全局的Context容器
    Request request = initRequest(httpServletRequest);
    // 新建一个全局唯一的请求traceId，并set进request中
    request.setTraceId(JrnGenerator.genTraceId());
    // 将初始化的请求信息放进ThreadLocal中
    Context.initialLocal(request);

    // ...

    return true;
  }
```

```
    // ...
}
```

系统内部上下文对象：

```
public class Context {  // ...   private static final ThreadLocal<Request> REQUEST_LOCAL = ne

    // ...

    private static final ThreadLocal<Request> REQUEST_LOCAL = new ThreadLocal<>();

    public final static void initialLocal(Request request) {
        if (null == request) {
            return;
        }
        REQUEST_LOCAL.set(request);
    }

    public static Request getCurrentRequest() {
        return REQUEST_LOCAL.get();
    }

    // ...
}
```

拦截器实现了`org.springframework.web.servlet.HandlerInterceptor`接口，它的主要作用是用于拦截处理请求，可以在MVC层做一些日志记录与权限检查等操作，这相当于MVC层的AOP，即符合横切关注点的所有功能都可以放入拦截器实现。

这里的`initRequest(httpServletRequest);`就是将请求信息封装成系统内容的请求对象`Request`，并初始化一个全局唯一的traceId放进Request中，然后再把它放进系统内部上下文ThreadLocal字段中。

接下来讲讲如何将ThreadLocal中的内容放到RpcContext中，在讲之前，我先来说说Dubbo基于spi扩展机制，官方文档对拦截器扩展解释如下：

> 服务提供方和服务消费方调用过程拦截，Dubbo 本身的大多功能均基于此扩展点实现，每次远程方法执行，该拦截都会被执行，请注意对性能的影响。

也就是说我们进行服务远程调用前，拦截器会对此调用进行拦截处理，那么就好办了，在消费者调用远程服务之前，我们可以偷偷把ThreadLocal的内容放进RpcContext容器中，我们可以基于dubbo的spi机制扩展两个拦截器，一个在消费者端生效，另一个在提供者端生效：

在META-INF中加入com.alibaba.dubbo.rpc.Filter文件，内容如下：

```
provider=com.objcoding.dubbo.filter.ProviderFilterconsumer=com.objcoding.dubbo.filter.ConsumerFilter
```

消费者端拦截处理：

```
public class ConsumerFilter implements Filter {  @Override  public Result invoke(Invoker<?>
   @Override
   public Result invoke(Invoker<?> invoker, Invocation invocation)
      throws RpcException {

      //1.从ThreadLocal获取请求信息
      Request request = Context.getCurrentRequest();
      //2.将Context参数放到RpcContext
      RpcContext rpcCTX = RpcContext.getContext();
      // 将初始化的请求信息放进ThreadLocal中
      Context.initialLocal(request);

      // ...

   }
}
```

`Context.getCurrentRequest();`就是从ThreadLocal中拿到Request请求内容，
`contextToDubboContext(request);`将Request内容放进当前线程的RpcContext容器中。

很容易联想到提供者也就是把RpcContext中的内容拿出来放到ThreadLocal中：

```
public class ProviderFilter extends AbstractDubboFilter implements Filter{  @Override  pub
   @Override
   public Result invoke(Invoker<?> invoker, Invocation invocation)
      throws RpcException {
      // 1.获取RPC远程调用上下文
      RpcContext rpcCTX = RpcContext.getContext();
      // 2.初始化请求信息
      Request request = dubboContextToContext(rpcCTX);
      // 3.将初始化的请求信息放进ThreadLocal中
      Context.initialLocal(request);

      // ...

   }
}
```

接下来我们还要配置log4j2，使得我们同一条请求在关联的每一个容器打印的消息，都有一个共同的traceId，那么我们在ELK想要查询某个请求时，只需要搜索traceId，就可以看到整条请求链路的日志了。

我们在Context上下文对象的`initialLocal(Request request)`方法中在log4j2的上下文中添加traceId信息：

```
public class Context {  // ...  final public static String TRACEID = "_traceid";  public fir

   // ...

   final public static String TRACEID = "_traceid";

   public final static void initialLocal(Request request) {
      if (null == request) {
         return;
      }
```

```
        // 在log4j2的上下文中添加traceId
        ThreadContext.put(TRACEID, request.getTraceId());
        REQUEST_LOCAL.set(request);
    }

    // ...
}
```

接下来实现`org.apache.logging.log4j.core.appender.rewrite.RewritePolicy`：

```
@Plugin(name = "Rewrite", category = "Core", elementType = "rewritePolicy", printObject = true)publi
public final class MyRewritePolicy implements RewritePolicy {

    // ...

    @Override
    public LogEvent rewrite(final LogEvent source) {
        HashMap<String, String> contextMap = Maps.newHashMap(source.getContextMap());
        contextMap.put(Context.TRACEID, contextMap.containsKey(Context.TRACEID) ? contextMap
        return new Log4jLogEvent.Builder(source).setContextMap(contextMap).build();
    }

    // ...
}
```