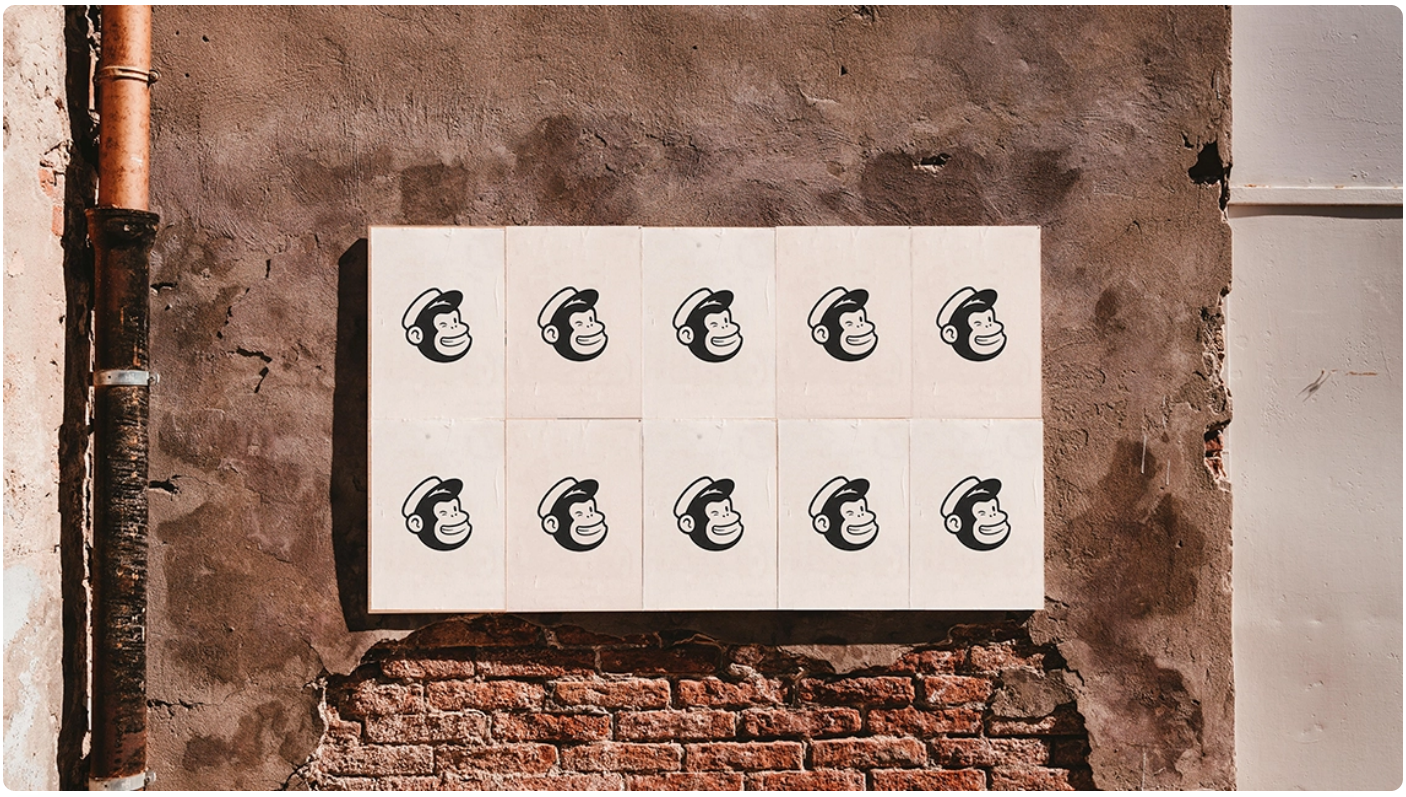
03 | 隐式传递：如何精准找出一次请求的全部日志？

2022-12-23 何辉 来自北京

《Dubbo源码剖析与实战》



讲述：何辉

时长 14:47　大小 13.50M

你好，我是何辉。

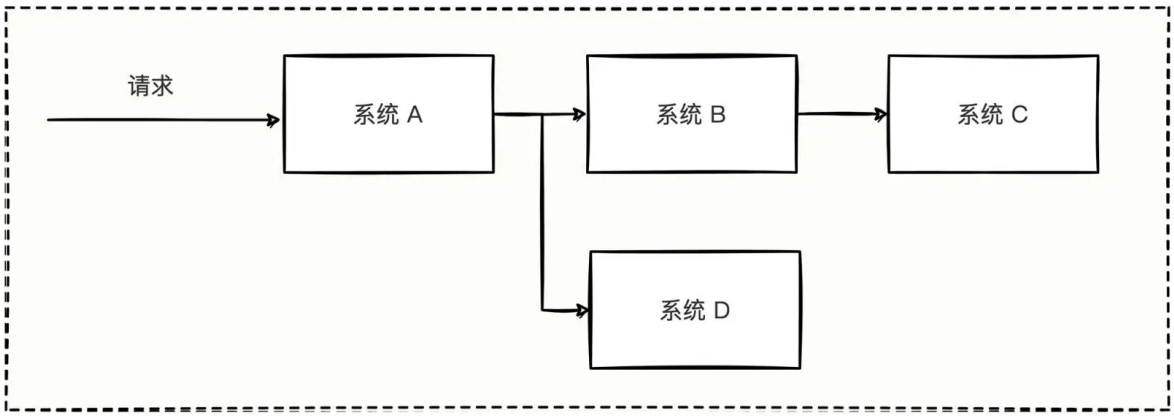
上一讲我们学习了如何把一些耗时的业务进行异步化改造，核心三要素就是开启异步模式、衔接上下文信息、将结果写入到上下文中，这也是 Dubbo 的异步实现原理。

今天我们继续探索 Dubbo 框架的第二道特色风味，隐式传递。

在我们痛并快乐着的日常开发工作中，修 bug 已经是很常见一环了，当编写的功能出了 bug，我们一般会先根据现象分析可能存在的问题，如果没头绪，就会继续根据用户提供的有限关键字信息，去查看相关日志希望能找到蛛丝马迹。

在这个环节，如果关键字信息比较独特且唯一，我们比较容易排查出问题，但如果关键字不那么独特，我们很可能需要从检索出来的一堆日志中继续痛苦地分析。

然而痛苦才刚刚开始，实际开发会涉及很多系统，如果出问题的功能调用流程非常复杂，你可能都不确定找到的日志是不是出问题时的日志，也可能只是找到了出问题时日志体系中的小部分，还可能找到一堆与问题毫无关系的日志。比如下面这个复杂调用关系：



图中描述了一种多系统合作的链路，一个请求调用了系统 A，接着系统 A 分别调用了系统 B 和系统 D，然后系统 B 还调用了系统 C。

通过请求中的关键字，我们在 A、B、C、D 系统中找到了相关日志：

```
2022-10-28 23:29:01.302 [系统A,DubboServerHandler-1095] INFO com.XxxJob - [JOB]
2022-10-28 23:29:02.523 [系统B,DubboServerHandler-1093] INFO WARN XxxImpl - quer
2022-10-28 23:30:23.257 [系统C,DubboServerHandler-1096] INFO ABCImpl - recv Requ
2022-10-28 23:30:25.679 [系统D,DubboServerHandler-1094] INFO XyzImpl - doQuery S
2022-10-28 23:31:18.310 [系统B,DubboServerHandler-1093] INFO WARN XxxImpl - quer
```

说明一下，这段日志信息是方便我们具体分析问题模拟出来的，但日常你看到的日志格式也是大同小异。看系统 B 的 DubboServerHandler-1093 线程打印的两行日志，第一眼从打印内容的上下文关系上看，我们会误认为这就是要找的错误信息。

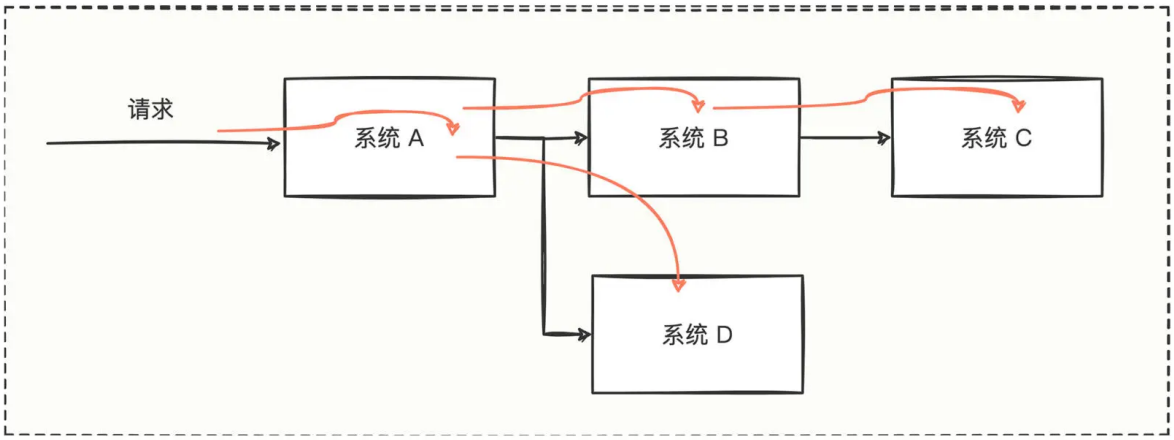
但实际开发中一定要考虑不同请求、不同线程这两个因素，**你能确定这两行日志一定是同一次请求、同一个线程打印出来的么？**

其实并不能，那有什么更好的办法来区分出来呢。

日志检索问题

现在的主要问题就变成了一次请求调用系统 A、B、C、D，如何精准找出一次请求所打印的全部日志？

其实思路也很简单，类似田径 400 米接力赛跑，一个请求，如果有一个序列号，而且序列号还能被“接棒”传到系统 A、B、C、D，那么我们就可以利用这个序列号，将这次请求的日志全部检索出来。



极客时间

结合刚才的例子，系统 B 的 DubboServerHandler-1093 线程打印的两行日志，如果有一个序列号的体现，我们就能确定是不是同一次请求所打印的。

可是该怎么把一次请求动态生成的序列号，传递到此次请求所涉及的所有系统中去呢？

1. 显式传递

最显而易见的方案就是显式传参。我们可以在所有系统接收请求的对象中，统一添加一个序列号字段，让大家都按照一定的规约进行编码传递。来草拟一个抽象请求基类：

复制代码

```
@Setter
@Getter
@ToString
public class AbstractRequest implements Serializable {

    /** <h2>请求流水号，打印日志使用，而且还是必填字段</h2> **/
    @NotBlank
```

```
    private String reqNo;
}
```


在代码中定义一个 AbstractRequest 抽象的请求类，再定义一个请求流水号字段，并且标明是必填字段，让系统所有接收请求的对象都继承这个 AbstractRequest 抽象类。

这样一来，因为继承关系，每个接收请求的对象都相当于有了 reqNo 字段，最后告诉调用方按照要求传参就可以了。

思路很美好，但这个时候相信你一定发现了第一个盲点：**考虑方案的落地，已有对象如何修改呢？**

我们顺着这个思路先比划比划流程：

把系统中所有接收请求的对象找出来，然后挨个修改这些对象的继承属性，改为继承 AbstractRequest 类，最后在打印日志的时候顺便把 reqNo 字段值打印出来。

但是需要修改多少个接收请求对象的类呢？系统中有 10 个对象就要改 10 个，有 100 个对象那就改 100 个，但系统中有 10000 个对象，要改 10000 个？不太实际，而且在开发环境中，系统中接收请求对象的个数是不可估量的，我们光改个继承关系，就能改上几天几夜，还无法保证不出问题。

哪怕假设修改量不大，我们在打印日志的时候还会遇到第二个更难处理的问题。

打印的一般思路是在处理业务逻辑之前，将入参 req 对象中的 reqNo 字段值打印出来：

```java
@DubboService
@Component
public class UserQueryFacadeImpl implements UserQueryFacade {
    private Logger logger = LoggerFactory.getLogger(UserQueryFacadeImpl.class);
    @Override
    public QueryUserInfoUserResp queryUserInfo(QueryUserInfoReq req) {
        logger.info("reqNo: {}, queryUser 入参参数为: {}", req.getReqNo(), req);

        // 省略其他逻辑...
    }
}
```


看起来不错，至少可以通过流水号来检索出日志了。但问题是我们只有在 queryUserInfo 当前的方法体中才能拿到 req 对象。

万一其他接口的请求对象不是 QueryUserInfoReq 这种类型，而是 String、Integer 这种类型的话，该怎么继承呢？String、Integer 都是 JDK 中非常基础的类，根本无法修改，即便能修改，我们也不敢改，毕竟这些类在代码中遍地都是，一旦修改影响面根本无法预估。

而且为了确保在代码的任何一个角落都能拿到 reqNo 字段值，如果想在子方法、子子方法中继续打印 reqNo 字段，还得把 req 对象传下去，我们都能想像到将来代码中遍地都是各种 req 对象传来传去惨不忍睹的画面，代码的可读性非常糟糕。

如果 queryUserInfo 还需要调用下游系统，还得想办法把 QueryUserInfoReq 对象中的 reqNo 字段值取出来，然后赋值到下游的请求对象中，这又得把所有涉及下游系统请求对象的代码又全部改一遍。

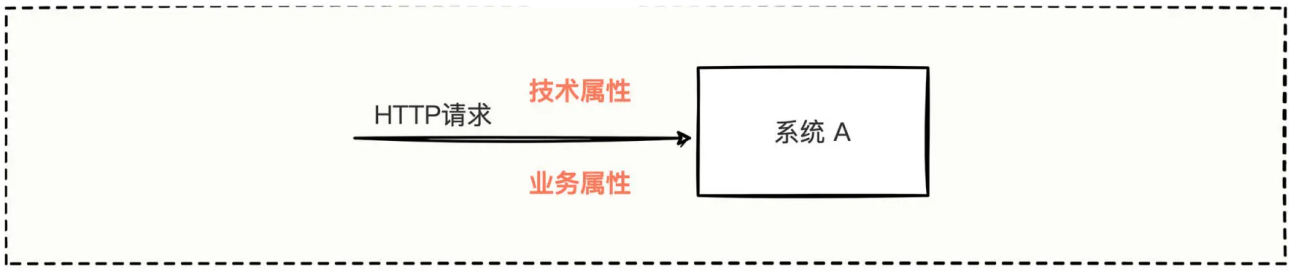
所以，按显式方式处理序列号越想问题越多，我们不但得一顿大改，还会把整个工程代码整得鸡犬不宁，乌烟瘴气。既然显式传参不行，有没有其他方案呢？

2. 隐式传递

现在的问题就转变为，既不能改动方法的请求入参对象，也不能在工程中遍地开花地修改代码，还要能在日志中看到序列号，似乎有点困难。

但是仔细分析这个需求，你就能找到突破点。请求入参对象，都是一些处理业务功能需要的对象，但是这个序列号参数，业务其实并不需要，因为业务不关心日志怎么打印，也不关心开发人员怎么排查问题。

所以序列号和业务对象在一定程度上划分了界限，我们可以**把业务对象划分到业务属性，把请求序列号划分到技术属性**。以一次简单的请求发送为例：



HTTP 请求在向系统 A 发起请求时，需要想办法将技术属性和业务属性都发送给系统 A，怎么编写代码发送 HTTP 请求的呢？

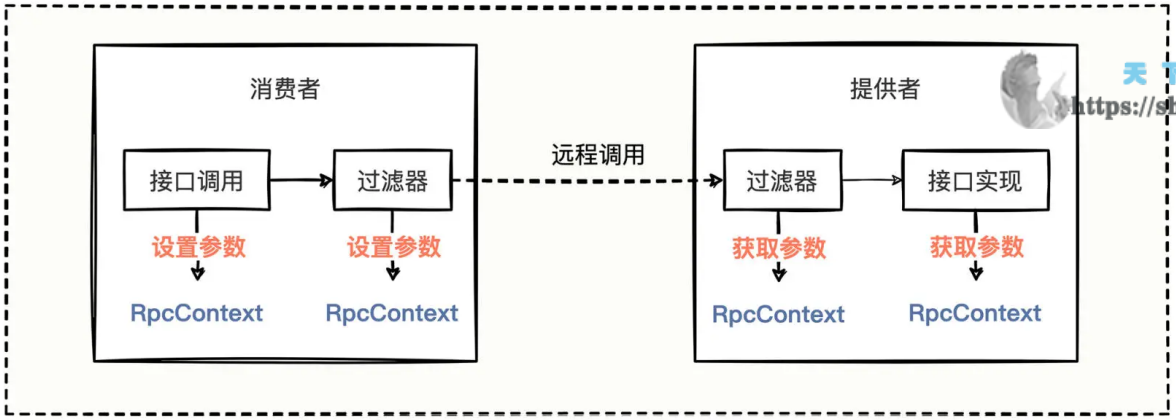
回忆平时发送 HTTP 请求的逻辑步骤：

1. 将业务对象设置到 HTTP 的请求体中；
2. 将一些 Content-Type、Content-Length 属性等设置到请求头中；
3. 设置 URL，发起 HTTP 请求。

所以我们可以将业务属性放到请求体中，将技术属性放到请求头中，这样不就把技术属性和业务属性一起发送出去了么。

可是系统 A 与系统 B、C、D 之间是 Dubbo 调用的，又该怎么解决技术属性的传递问题呢？

同样地，我们回忆平时编写代码进行 Dubbo 远程调用时的流程链路：



这是一个比较简单的消费者调用提供者的链路图，在消费者调用的过程中，一些附加的信息可以设置到 RpcContext 上下文中去，然后 RpcContext 中的信息就会随着远程调用去往提供者那边。

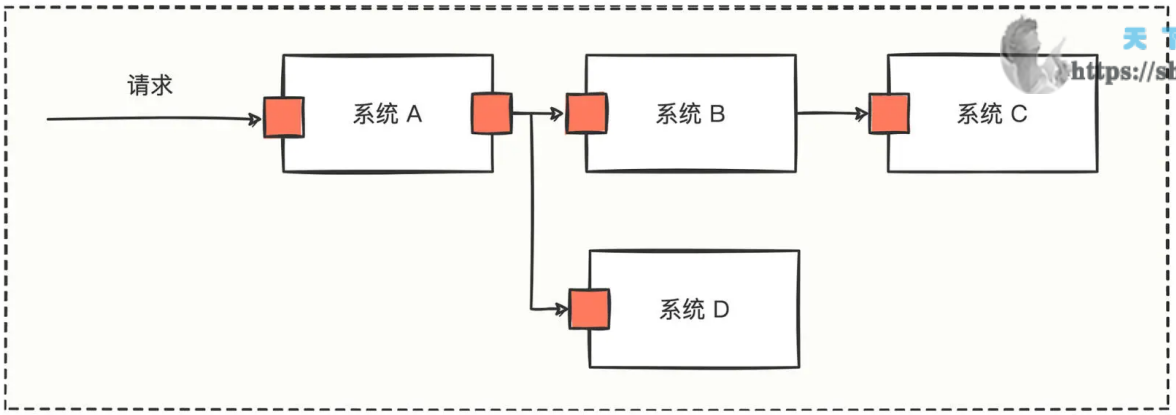
可以看到，调用链路图中的附带参数设置形式和 HTTP 的请求头设置形式，几乎是如出一辙。所以我们可以考虑将序列号设置到 RpcContext 中，这样就能既不改动方法的请求入参对象，又不用大范围进行修改了。

那如何把参数设置到 RpcContext 中呢？我们进入今天的高潮环节——隐式传参的实现。

编码实现

对于技术属性的设置，现在的你肯定知道是不能大肆在代码各个角落进行操作的，也不现实，我们需要有一个集中的环节可以进行操作，那么这个集中环节在哪里呢？

再来看调用链路图：



为了尽可能降低侵入性，我们最好能在系统的入口和出口，把接收数据的操作以及发送数据的操作进行完美衔接。这就意味着需要在接收请求的内部、发送请求的内部做好数据的交换。

再结合前面的消费者调用提供者的链路图，编写业务代码无感知的“过滤器”，似乎能完美满足需求：

- 系统 A 接收请求，站在提供者的角度，在处理接口实现逻辑之前会经过滤器处理。
- 系统 A 发送请求，站在消费者的角度，一样会经过滤器处理。

所以我们只需要定义一个消费者维度的序列号过滤器，然后再定义一个提供者维度的序列号过滤器，就可以把序列号在调用链路中完美衔接起来了。

来看代码实现：

复制代码

```
@Activate(group = PROVIDER, order = -9000)
public class ReqNoProviderFilter implements Filter {
    public static final String TRACE_ID = "TRACE-ID";
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
        // 获取入参的跟踪序列号值
        Map<String, Object> attachments = invocation.getObjectAttachments();
        String reqTraceId = attachments != null ? (String) attachments.get(TRAC

        // 若 reqTraceId 为空则重新生成一个序列号值，序列号在一段相对长的时间内唯一足够了
        reqTraceId = reqTraceId == null ? generateTraceId() : reqTraceId;

        // 将序列号值设置到上下文对象中
```

```
        RpcContext.getServerAttachment().setObjectAttachment(TRACE_ID, reqTrace
        // 并且将序列号设置到日志打印器中，方便在日志中体现出来
        MDC.put(TRACE_ID, reqTraceId);
        // 继续后面过滤器的调用
        return invoker.invoke(invocation);
    }
}

@Activate(group = CONSUMER, order = Integer.MIN_VALUE + 1000)
public class ReqNoConsumerFilter implements Filter, Filter.Listener {
    public static final String TRACE_ID = "TRACE-ID";
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
        // 从上下文对象中取出跟踪序列号值
        String existsTraceId = RpcContext.getServerAttachment().getAttachment(T
        // 然后将序列号值设置到请求对象中
        invocation.getObjectAttachments().put(TRACE_ID, existsTraceId);
        RpcContext.getClientAttachment().setObjectAttachment(TRACE_ID, existsTr
        // 继续后面过滤器的调用
        return invoker.invoke(invocation);
    }
}
```


思路也很清晰，主要新增了两个过滤器：

- ReqNoProviderFilter 为提供者维度的过滤器，主要接收请求参数中的 traceId，并将 traceId 的值放置到 RpcContext 上下文对象中。
- ReqNoConsumerFilter 为消费者维度的过滤器，主要从 RpcContext 上下文对象中取出 traceId 的值，并放置到 invocation 请求对象中。

然后遵循 Dubbo 的 SPI 特性将两个过滤器添加到 META-INF/dubbo/org.apache.dubbo.rpc.Filter 配置文件中：

复制代码

```
reqNoConsumerFilter=com.hmilyylimh.cloud.ReqNoConsumerFilter
reqNoProviderFilter=com.hmilyylimh.cloud.ReqNoProviderFilter
```

最后修改一下日志器的打印日志模式：

```
%d{yyyy-MM-dd HH:mm:ss.SSS} [${APP_NAME}, %thread, %X{X-TraceId}] %-5level %c{1
```


经过改造后，我们看到的日志就会是这样的：

```
2022-10-29 14:29:01.302 [系统A,DubboServerHandler-1095,5a2e67913efee084] INFO cc
2022-10-29 14:29:02.523 [系统B,DubboServerHandler-1093,9b42e2bf4bc2808e] INFO WA
2022-10-29 14:30:23.257 [系统C,DubboServerHandler-1096,6j40e2mn4bc4508e] INFO AE
2022-10-29 14:30:25.679 [系统D,DubboServerHandler-1094,wx92bn9f4bc2m8z4] INFO Xy
2022-10-29 14:31:18.310 [系统B,DubboServerHandler-1093,9b42e2bf4bc2808e] INFO WA
```

看系统 B 的 DubboServerHandler-1093 线程打印的两行日志，1093 后面都是 9b42e2bf4bc2808e 这个序列号值，说明这一定是同一次请求、同一个线程打印出来的日志。

有了这样的日志检索能力支撑，之前比较难排查的问题，基本上我们只需要两步就能很快搞定：

- 第一步，通过用户提供的少许关键字，检索出符合条件的日志，从这些日志中大致筛选出符合用户出问题的日志，然后从中随机拷贝一行日志的跟踪号。
- 第二步，用拷贝的跟踪号继续检索所有日志，就可以将跟踪号这一次请求的所有日志全部检索出来，再来着重分析问题。

隐式传递的应用

显式传递，我们都知道，一般是在明确业务意义的一种接口契约形式，大家都按照接口契约各自行事。那今天学习的隐式传递，在我们的日常开发中，又有哪些应用场景呢？

第一，传递请求流水号，分布式应用中通过链路追踪号来全局检索日志。

第二，传递用户信息，以便不同系统在处理业务逻辑时可以获取用户层面的一些信息。

第三，传递凭证信息，以便不同系统可以有选择性地取出一些数据做业务逻辑，比如 Cookie、Token 等。

总体来说传递的都是一些技术属性数据，和业务属性没有太大关联，为了方便开发人员更为灵活地扩展系统能力，来更好地支撑业务的发展。

总结

今天，我们从一个检索日志困难的问题开始，分析了显式传递和隐式传递两种不同方案带来的问题和影响。

显式传递，会导致所有请求对象大改造、请求对象为 String 类型的局限性、调用下游系统设置入参逻辑大改造等，所以我们考虑通过隐式传递来处理。

类比 HTTP 协议的设计理念，我们发现技术属性与业务属性的区别，再结合 Dubbo 调用的流程图，在调用过程中，可以将技术属性设置到 RpcContext 中进行系统间传递，这样既不需要大改代码，也不需要和业务属性混在一起，最终通过自定义两个过滤器从代码层面落实方案。

这里也总结一下自定义过滤器的四个步骤：

- 首先，创建一个自定义的类，并实现 org.apache.dubbo.rpc.Filter 接口；
- 其次，在自定义类上通过 @Activate 注解标识是提供方维度的过滤器，还是消费方维度的过滤器；
- 然后，在自定义类中的 invoke 方法中实现传递逻辑，提供方过滤器从 invocation 取出 traceId 并设置到 ClientAttachment、MDC 中，消费方过滤器从 ClientAttachment 取出 traceId 并设置到 invocation 中；
- 最后，将自定义的类路径添加到 META-INF/dubbo/org.apache.dubbo.rpc.Filter 文件中，并取个别名。

隐式传递的应用场景主要有 3 类，传递链路追踪号、传递用户信息、传递凭证信息。

思考题

你已经学会了使用 RpcContext 设置技术属性并传递到不同的系统中去，而 RpcContext 这个类也确实在实际开发中很常用。源码中是这么描述它的：

复制代码

```
1 RpcContext is a temporary state holder. States in RpcContext changes every time
```

由此可见，RpcContext 是临时状态的保持器，每次发送或接收请求时，RpcContext 中的状态都会发生变化。

RpcContext 有 4 个重要属性和你设置隐式的技术属性有关，分别是 SERVER_LOCAL、CLIENT_ATTACHMENT、SERVER_ATTACHMENT、SERVICE_CONTEXT，你知道它们的生命周期吗？

期待看到你的思考，如果你对隐式传递还有什么困惑，欢迎在留言区提问，我会第一时间回复。

如果觉得今天的内容对你有帮助，也欢迎分享给身边的朋友一起讨论。我们下一讲见。

02 思考题参考

上一期的问题是利用 CompletableFuture 中的一些 API ，将复杂的多任务场景通过编写代码计算出累加和。解答此题我们关注 3 点：

1. 任务执行完成后再并发执行其他任务，可以使用 thenXxxAsync 这样的方法，并且执行完成之后还得返回结果值，那么我们就可以采用 thenApplyAsync 方法。
2. 合并两个线程任务的结果，并做进一步累加和处理，这里我们可以采用 thenCombine 方法。
3. 两个线程任务并发执行，谁先执行完成就以谁为准，这里我们可以采用 applyToEither 方法。

最终实现的代码如下：

复制代码

```
public static void main(String[] args) throws Throwable {
    ////////////////////////////////////////////////////
    // 任务一执行流程
    ////////////////////////////////////////////////////
    // 执行 taskA1
    CompletableFuture<Integer> taskA1 = CompletableFuture.supplyAsync(() -> 1);
    // taskA1 执行完后，再并发执行 taskB1、taskC1
    CompletableFuture<Integer> taskB1 = taskA1.thenApplyAsync(integer -> 2);
    CompletableFuture<Integer> taskC1 = taskA1.thenApplyAsync(integer -> 3);
```

```
    // 任务一的结果
    CompletableFuture<Integer> result1 =
            // 等到 taskB2、taskC2 都执行完并合并结果后
            taskB1.thenCombine(taskC1, Integer::sum)
            // 再合并 taskA1 的结果后
            .thenCombine(taskA1, Integer::sum)
            // 再异步执行 taskD1
            .thenApplyAsync(integer -> integer + 4);

    ////////////////////////////////////////////////
    // 任务二执行流程
    ////////////////////////////////////////////////
    // 执行 taskA2
    CompletableFuture<Integer> taskA2 = CompletableFuture.supplyAsync(() -> 1);
    // taskA2 执行完后，再并发执行 taskB2、taskC2
    CompletableFuture<Integer> taskB2 = taskA2.thenApplyAsync(integer -> 2);
    CompletableFuture<Integer> taskC2 = taskA2.thenApplyAsync(integer -> 3);
    // 任务二的结果
    CompletableFuture<Integer> result2 =
            // 等到 taskB2、taskC2 任意其中一个有结果后
            taskB2.applyToEither(taskC2, Function.identity())
            // 再合并 taskA2 的结果后
            .thenCombine(taskA2, Integer::sum)
            // 再异步执行 taskD2
            .thenApplyAsync(integer -> integer + 4);

    ////////////////////////////////////////////////
    // 任务一 + 任务二，合并结果
    ////////////////////////////////////////////////
    CompletableFuture<Integer> result = result1.thenCombine(result2, Integer::s
    try {
        // 任务总超时时间设置为5s
        System.out.println(result.get(5, TimeUnit.SECONDS));
    } catch (InterruptedException | ExecutionException | TimeoutException e) {
        // 超时则打印0
        System.out.println(0);
        e.printStackTrace();
    }
}
```


到这里，我们就用代码将图中的多任务场景实现完了，最终打印的结果为 17 或者 18，结果为 0 的概率非常非常低。

上一篇 02丨异步化实践：莫名其妙出现线程池耗尽怎么办？

下一篇 04丨泛化调用：三步教你搭建通用的泛化调用框架

精选留言 (5)

写留言

乌凌先森

2023-01-19 来自广西

老师你好，@DubboService + @Component 这种使用方式有啥好处？

作者回复: 你好，乌凌先森：这俩注解各自解决的问题不一样：

1.@DubboService 解决的是在编码层面时接口实现类可以处理Dubbo的接收请求。

2.@Component 解决的是在 Spring 框架中该接口实现类变成单实例对象以便后续可以被 @Autowired、@Resource 进行注入使用。

张三丰

2023-01-03 来自广东

感觉文中获取traceid的地方有问题，不应该在提供者处生成，应该在消费端生成是traceid，再把traceid传给提供者，如果提供者拿到了traceid就打印出来，如果拿不到再生成traceid返回给消费者

作者回复: 你好，张三丰：你站在的角度是消费者应用一定是请求的源头，所以你会这么理解。若消费方是前端呢？难道需要前端来生成traceId么？

只是站在看问题的角度不同罢了，不过也挺好的，说明至少是认真在思考这个traceId传递衔接的问题，挺棒的。

我这里引用我之前回答过的内容，如下：

说消费方还有点不太准确，精准点说，应该是从接收请求的那个源头就可以考虑生成traceId。

比如接收前端请求，Web容器比如Tomcat的Filter是最能第一时间感知请求的存在，可以在这里进行拦截直接生成TraceId，至于Tomcat在Filter之后的一些处理环节，就可以直接拿到TraceId了。再进入Controller调用下游Dubbo接口的话，消费方发现上下文没有 traceId 的也是可以考虑生成，也是一种兼容考虑方法，挺好的。

比如 A ->B -> C，抛开Web容器来看待的话，A 的消费方过滤器其实拿到的 traceId 是 null 值，但是 B 所能很好衔接 traceId 的话，那么 B 在发起调用 C 的时候，B 的消费方过滤器是能正常拿到 traceId 的。

共 4 条评论 >

1

小白

2022-12-27 来自广东

老师，还有一个问题，为什么不直接用RpcContext进行get 和 set 传递呢？为什么还要涉及到invocation，这块不是很理解。

作者回复: 你好，小白：org.apache.dubbo.rpc.RpcContext#get()、org.apache.dubbo.rpc.RpcContext#set、org.apache.dubbo.rpc.RpcContext#remove、org.apache.dubbo.rpc.RpcContext#get(java.lang.String) 等等 API 已经在 RpcContext 中被标注 @Deprecated 注解，说明在新版本是不再建议使用了。

而是使用更加明确的获取方式（invocation.getObjectAttachments()、RpcContext.getClientAttachment()），从 invocation 中获取数据是明确表示该数据一定是从接收的参数中获取的，这是一种见名知意的写代码表述方式而已。

但是这里你还要结合 ContextFilter、ConsumerContextFilter 来看，你要把数据放对就行了。

Geek_10086

2022-12-24 来自广东

老师您好，traceId是不是应该在消费者端（ReqNoConsumerFilter）生成，通过隐式传递到服务提供端（ReqNoProviderFilter），文中代码在ReqNoConsumerFilter中从上下文获取应该是null吧

作者回复: 你好，Geek_10086：说消费方还有点不太准确，精准点说，应该是从接收请求的那个源头就可以考虑生成traceId。

比如接收前端请求，Web容器比如Tomcat的Filter是最能第一时间感知请求的存在，可以在这里进行拦截直接生成TraceId，至于Tomcat在Filter之后的一些处理环节，就可以直接拿到TraceId了。再进入Controller调用下游Dubbo接口的话，消费方发现上下文没有 traceId 的也是可以考虑生成，也是一种

兼容考虑方法，挺好的。

至于你说的消费方过滤器默认为 null 的情况，一半对一半不对，比如 A ->B -> C，抛开Web容器来看待的话，A 的消费方过滤器其实拿到的 traceId 是 null 值，但是 B 所能很好衔接 traceId 的话，那么 B 在发起调用 C 的时候，B 的消费方过滤器是能正常拿到 traceId 的。

王巍

2022-12-23 来自广东

多线程的情况下，线程也能获取到正确的 traceId 吗？

作者回复: 你好，王巍：你问得这个细节非常 nice，进程中多线程的 traceId 传递，是另外一个话题。

这里我给个大概思路，你可以单独将这些多线程之间如何传递 traceId 做成一个插件，比如可以横切Spring的AsyncTaskExecutor的方法，比如统一指定公司规范使用某几种 Runnable/Callable 来操作线程，比如 MQ/Job 在触发时刻的源头直接自动横切一刀赋上traceId，等等等等，总之旨在将方法执行前与方法执行后的traceId衔接起来。