06 | 事件通知：一招打败各种神乎其神的回调事件

2022-12-30 何辉 来自北京

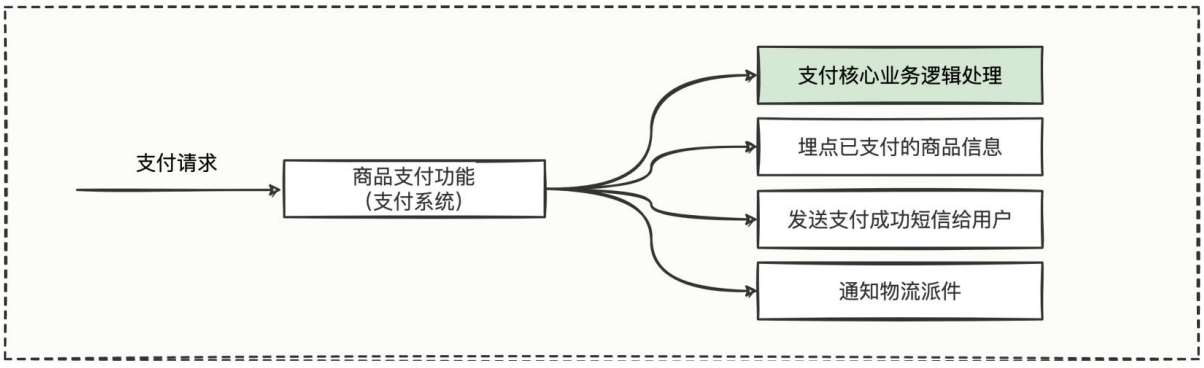
《Dubbo源码剖析与实战》

讲述：何辉

时长 17:57　大小 16.41M

你好，我是何辉。今天我们探索 Dubbo 框架的第五道特色风味，事件通知。

如果你用过 Spring 的 Event 事件，想必对事件通知不陌生，我们看个项目例子回顾一下，比如有个支付系统提供了一个商品支付功能：

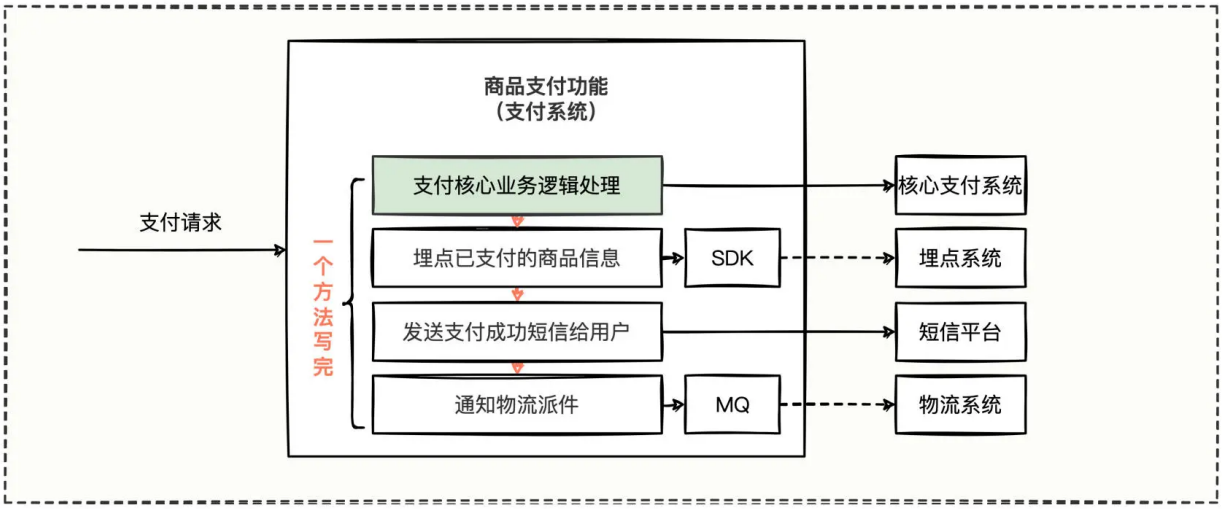


图中的支付系统暴露了一个支付请求的 Dubbo 接口，支付核心业务逻辑是调用核心支付系统完成，当支付状态翻转为支付成功后，还需要额外做些事情，比如埋点商品信息、短信告知用户和通知物流派件。

面对这样一个完成核心功能后还需要额外处理多个事件的需求，你会怎么优雅地快速处理呢？

面向过程编程

商品支付成功后需要处理三件事（埋点、发短信、通知物流），这样的需求，你一定觉得简直太小儿科了，从上到下写就完事了。实现体逻辑就是图中线性的形式：



极客时间

复制代码

```
@DubboService
@Component
public class PayFacadeImpl implements PayFacade {
    // 商品支付功能：一个大方法
    @Override
    public PayResp recvPay(PayReq req){
        // 支付核心业务逻辑处理
        此处省略若干行代码

        // 埋点已支付的商品信息
        此处省略若干行代码

        // 发送支付成功短信给用户
        此处省略若干行代码

        // 通知物流派件
        此处省略若干行代码
```

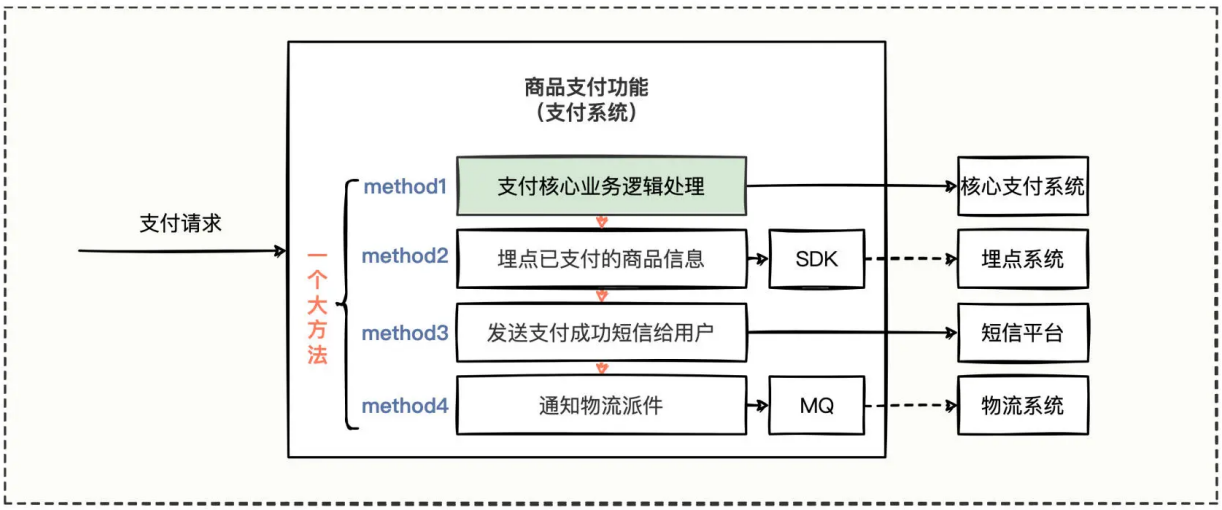
```
        // 返回支付结果
        return buildSuccResp();
    }
}
```


支付系统完成核心业务逻辑后，先调用埋点 SDK 进行支付商品埋点，然后调用短信平台发送短信，最后向 MQ 发送了一条物流派送消息。整个编码流程一气呵成。

但是细想一下，一个大方法里写完所有功能逻辑，搞不好会有几千行代码。这种面向过程的写法，虽然写代码的人很爽，但是看代码的人可能会迷失方向，更重要的是可读性变差，增加了维护成本，也延长了维护周期。

机智的你要补充了，那就**用面向对象编程的思路，把一些小功能用小方法封装一下**，让那一大坨代码整体看起来整洁点。

好，我们整顿一下，把原来的大方法做细粒度的拆分：



复制代码

```
@DubboService
@Component
public class PayFacadeImpl implements PayFacade {
    // 商品支付功能：一个大方法
    @Override
    public PayResp recvPay(PayReq req){
        // 支付核心业务逻辑处理
        method1();
```

```
9
10          // 埋点已支付的商品信息
11          method2();
12
13          // 发送支付成功短信给用户
14          method3();
15
16          // 通知物流派件
17          method4();
18
19          // 返回支付结果
20          return buildSuccResp();
21      }
22  }
```


商品支付功能的核心业务逻辑和另外 3 件事，是相对独立的功能，分别封装成了小方法。

改善后，支付功能的实现体逻辑整洁多了，大方法里面的 4 个小方法已经体现了该商品支付的大致业务流程，大体上有点封装的样子，代码主流程变清晰，可读性变强，各个模块之间有了一定的边界，维护起来也更容易。

但停在这里，可能会为我们之后的繁重工作量埋下种子，比如一周后需求又来了，要发送邮件、通知结算，怎么办呢？是不是还得继续添加小方法 5 和小方法 6？

其实吧，**继续追加小方法是我们大多数开发人员惯用的伎俩**，直接在已有的代码后面接着写就完事，这样也没有什么大毛病，无非就是逻辑多了，方法多了，阅读起来比较费劲，只要功能能跑就阿弥陀佛了。

可是慢慢地，你就会发现自己变成只会垒业务代码的机器人了，毫无思考、毫无设计，很难掌控大型功能、系统或系统群的设计，将来机会来临时处处捉襟见肘，实施起来困难重重。

为了提升代码水平，我们继续思考商品支付的功能设计。

改造小技巧

分析现在的代码逻辑，虽然大方法中用一堆小方法串起来了，流程好像挺完美的，但核心逻辑和三个不那么关键的事件合在一起写，似乎没有重点可言，难以看出主要解决什么问题，容易让后续维护者随波逐流修改起来毫无章法，导致难以维护、复用和灵活扩展。

既然一起写不那么友好的话，分开不就行了？怎么分呢？

如何解耦

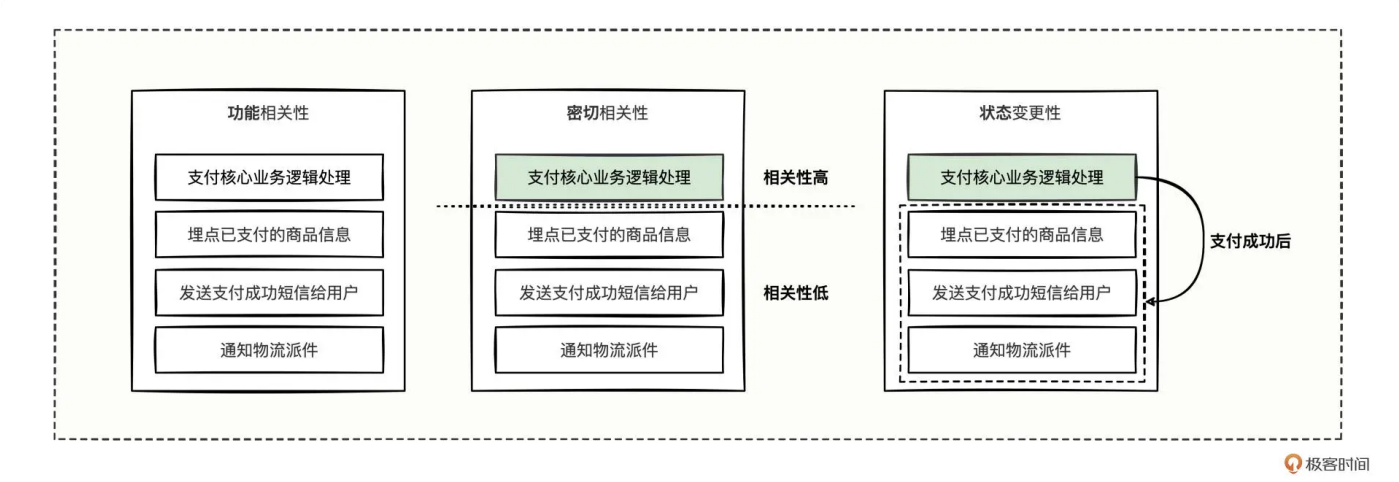
“分开”其实就是要做解耦，这里我教你一个解耦小技巧，从 3 方面分析：

1. **功能相关性**。将一些功能非常相近的汇聚成一块，既是对资源的聚焦整合，也降低了他人的学习成本，尊重了人类物以类聚的认知习惯。

具体怎么判断功能相近受很多因素影响，比如你的团队构成、组织划分、认知高低对业务的理解等等，倾向各不相同，并没有绝对的标准，合适的、团队认可的才是最好的。

2. **密切相关性**。按照与主流程的密切相关性，将一个个小功能分为密切与非密切。
3. **状态变更性**。按照是否有明显业务状态的先后变更，将一个个小功能再归类。

按照小技巧我们再梳理一下这 4 个功能：



先看功能相关性，四个小功能的拆分没问题；再看密切相关性，支付核心业务逻辑是最重要的，其他三个事件的重要程度和迫切性并不高；最后看状态变更性，核心业务逻辑有了明显的状态变更后，在支付成功的关键节点后，驱动着继续做另外三件事。

所以如何解耦呢？**我们可以将核心逻辑与三个事件剥离开，只有核心逻辑翻转为支付成功时才处理三个事件。**

这个例子的需求比较简单，你可以重点体会这个解耦技巧的思想，实际开发过程中的需求会复杂的多，用这个方法能很好地辅助你解耦需求问题。

如何串联

解耦完成，如何形成有血有肉、主次分明的结构呢？

相信你也想到了——以事件驱动的方式串联起来。事件驱动，简单理解就是一个事件会触发另一个或多个事件做出对应的响应行为。

在这个例子里，支付核心业务逻辑翻转为支付成功时，以这个状态变更节点为基准点，会触发另外三个事件做出对应的变化，**该怎么从代码层面把这个串联的思路落地呢？**

难道要核心逻辑执行后，再执行三个事件逻辑？如果是这样，岂不是又回到了面向过程的写法了，这明显不行。难道要把三个事件的代码写到核心逻辑以外么？

复制代码

```
@DubboService
@Component
public class PayFacadeImpl implements PayFacade {
    // 商品支付功能：一个大方法
    @Override
    public PayResp recvPay(PayReq req){
        // 支付核心业务逻辑处理
        method1();

        // 返回支付结果
        return buildSuccResp();
    }
}
```

商品支付功能的实现体逻辑中只有 method1 方法，其他三个事件没有写在 recvPay 方法中了。考虑到我们要体现 recvPay 的核心逻辑，然后在核心逻辑翻转为支付成功时调用另外 3 个事件，**现在的问题就是这个调用时机该怎么切入，或者说，怎么找到三个事件被调用的入口？**

为了快速落地编码，我们需要一个大的指导方向，拎清代码逻辑的每个环节需要什么，可以从哪些环节下手，来更好地把事件放到代码合适的位置进行解耦。

这里我再给你介绍一个小技巧——6W 分析模型：

- Who 谁产生的事件？是功能本身业务实现体产生，还是功能归属源码框架来产生？
- What 产生什么功能的事件？事件数据对象包含业务信息和事件凭证信息。
- When 什么时候发生的事件？在业务发生之前、之后，还是业务发生异常时发布。
- Where 在哪个系统、哪个功能、哪段代码位置发生的事件？
- Why 为什么要有这个事件？解决某类或某系列的问题，提供一种扩展机制来丰富事件价值。
- How 怎么把事件发布给需要关注的消费者？是自研框架，还是扩展已有框架中具有拦截或过滤特性的机制。

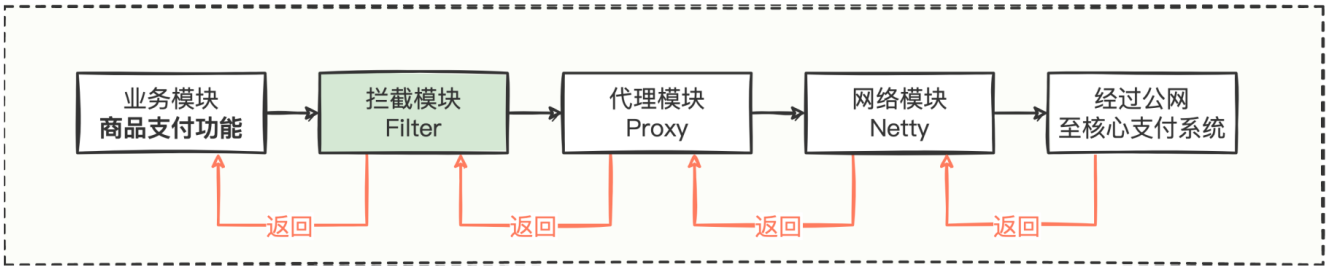
我们用这个模型先简单分析这三个事件，Who，都是商品支付功能中产生的支付完成事件；What，事件里面应该都具有商品和支付的关键信息，以及事件的唯一凭证信息；When，在商品支付功能核心逻辑处理完成后，支付状态翻转为支付成功时触发。

接下来的 Where，支付成功后，这三个额外的事件放到哪个位置合适呢？

按照 6W 中 Who，这三个事件是商品支付功能支付完成后需要去完成的事件，即我们需要在商品支付功能中提供这样的机制。但 recvPay 方法主体是用来体现核心支付逻辑的，这个模块中已经没法再追加代码了。那怎么办呢？

考虑到商品支付这个功能也只是 Dubbo 众多接口中的一个，我们不妨升维思考，站在 Dubbo 接口的框架调用流程中，看看是否可以在商品支付功能 method1 远程调用的前后做点事情来提供事件通知的入口。

回忆 Dubbo 调用远程的整个流程：



从商品支付功能的处理，到拦截模块，到代理模块，到网络模块，最后经过公网发送至核心支付系统。

从这张图中，你能看到业务模块是触发事件的源头，调用出去时经过拦截模块，返回也经过拦截模块，所以似乎可以借助拦截模块，实现一种新机制拦截 Dubbo 接口，类似面向切面编程的思想。

一想到 AOP，我们就有想法了：

1. 通过定义标准接口，让需要额外事件特性的功能去实现标准接口。
2. 通过反射调用，去调用任意指定的回调方法，按照约定好的规则去接口级别配置回调方法。

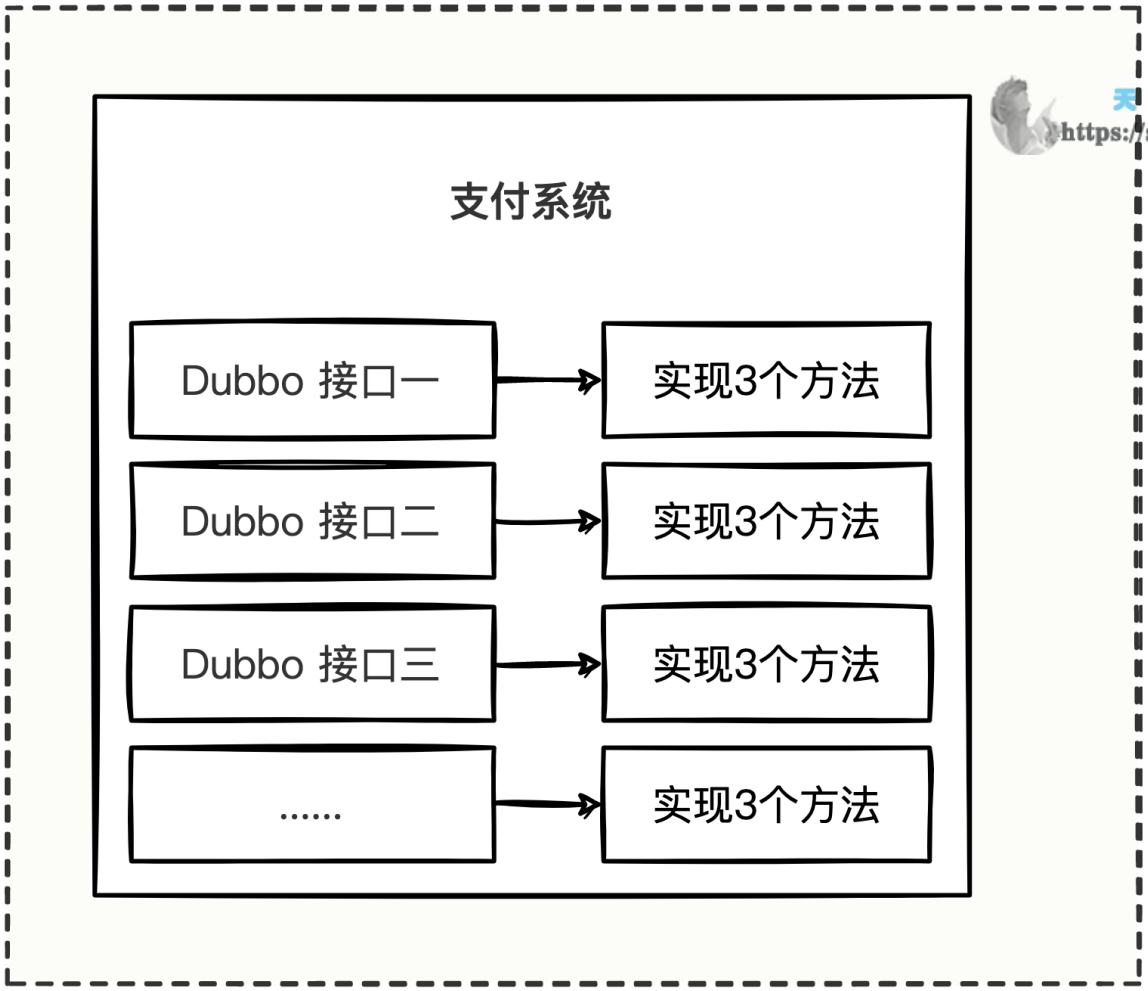
想法一标准接口

先看想法一，在底层定制标准接口：

复制代码

```
// 假设定制底层标准接口
public interface FrameworkNotifyService {
    // 调用之前
    void onInvoke(Request req);
    // 调用之后
    void onReturn(Response resp, Request req);
    // 调用异常
    void onThrow(Throwable ex, Request req);
}
```

底层标准接口里面有调用前、调用完成、调用异常的三个方法，然后让各自接口去实现标准接口，呈现出来的代码调用形式如图：



复制代码

```java
@DubboService
@Component
public class PayFacadeImpl implements PayFacade, FrameworkNotifyService {
    // 商品支付功能：一个大方法
    @Override
    public PayResp recvPay(PayReq req){
        // 支付核心业务逻辑处理
        method1();
        // 返回支付结果
        return buildSuccResp();
    }
    // 调用之前
    @Override
    void onInvoke(Request req){
        System.out.println("[事件通知][调用之前] onInvoke 执行.");
    }
    // 调用之后
    @Override
    void onReturn(Response resp, Request req){
        System.out.println("[事件通知][调用之后] onReturn 执行.");
        // 埋点已支付的商品信息
        method2();
        // 发送支付成功短信给用户
```

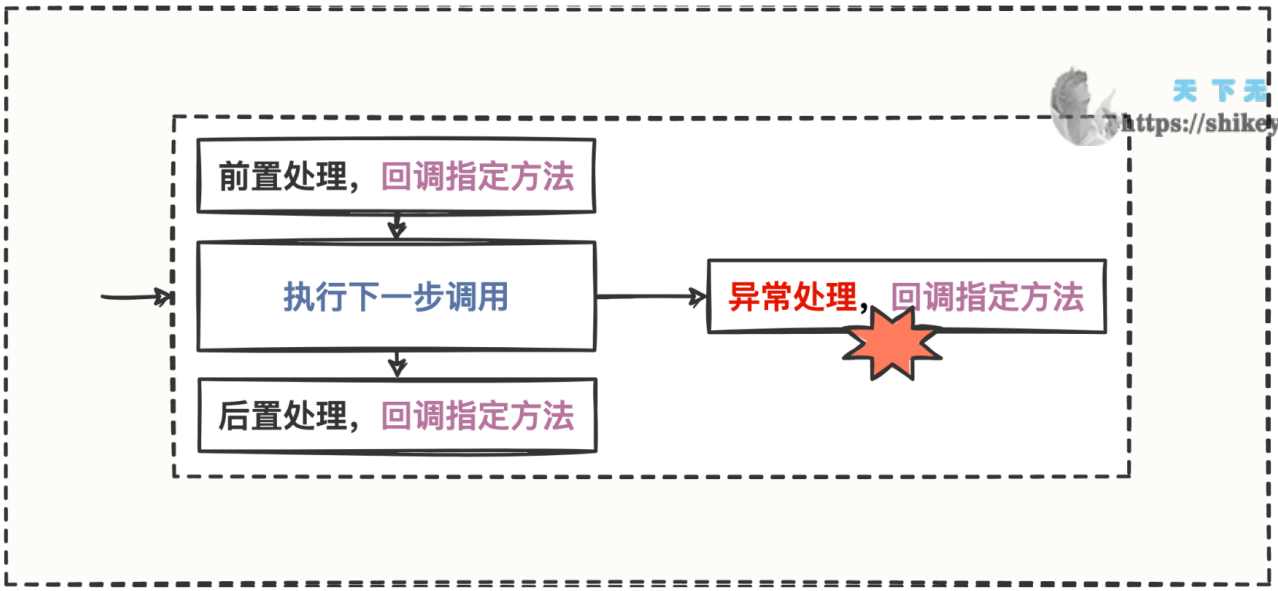
```
24          method3();
25          // 通知物流派件
26          method4();
27      }
28      // 调用异常
29      @Override
30      void onThrow(Throwable ex, Request req){
31          System.out.println("[事件通知][调用异常] onThrow 执行.");
32      }
33      @Override
34      public AliPayResp aliPay(AliPayReq req){
35          // 此处省略若干行代码
36          // 返回支付结果
37          return buildAliPayResp();
38      }
39      @Override
40      public WeixinPayResp weixinPay(WeixinerPayReq req){
41          // 此处省略若干行代码
42          // 返回支付结果
43          return buildWeixinPayResp();
44      }
45  }
```


我们已为 recvPay 方法在类上继承了 FrameworkNotifyService 底层标准接口，但是还想为 aliPay、weixinPay 方法再继续配置 FrameworkNotifyService 底层标准接口时，发现行不通了，因为 FrameworkNotifyService 在 PayFacadeImpl 中只能被 implements 声明一次，声明多次是不符合语法的。

即使假设可以在 PayFacadeImpl 中声明 3 个 FrameworkNotifyService 标准接口，那么会出现 3 个一模一样的 onInvoke 方法需要被重写，到时候我们怎么分清哪个 onInvoke 是属于 recvPay 方法的接口回调呢？明显更加不合理了，想法一 PASS。

想法二反射调用

那就看想法二，在接口级别配置回调方法，配置是开发人员配的：

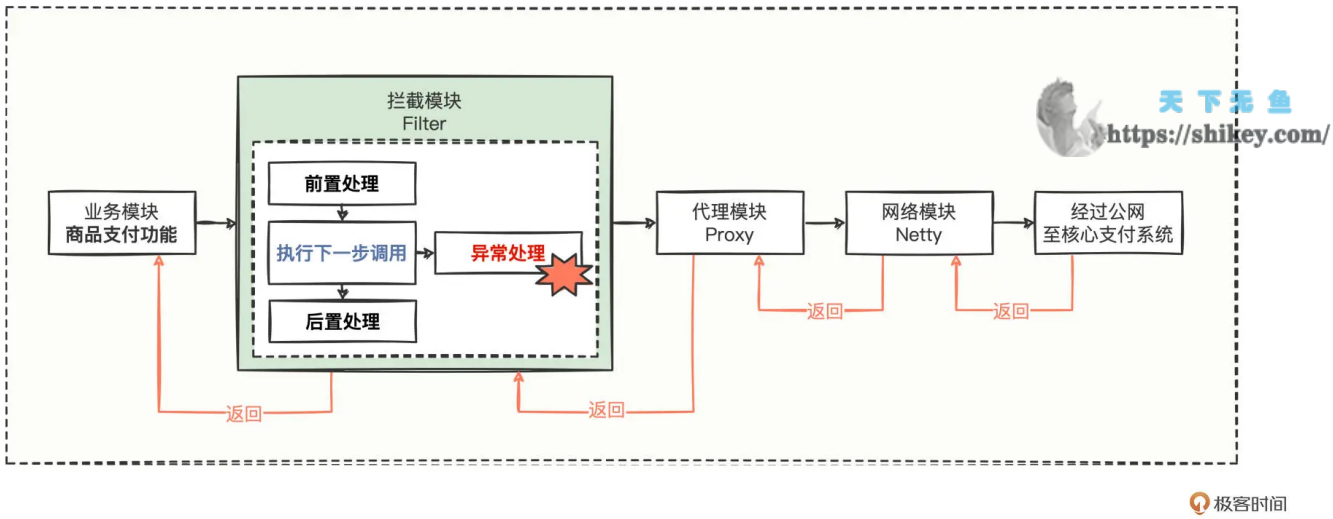


在前置、后置、异常时都去读取配置回调指定的方法，即使将来配错了出现 bug 也是开发人员的问题，并不是我们框架的问题，框架只需要把这套反射回调的逻辑做稳定。这样一来 **6W 中 Why** 落地了。

这套反射回调方法的机制是在实现体调用的前、后和异常时被触发的，那 **6W 中 When 和 How** 也落地了。至于 **Where，**要想拥有这样的环绕切面效果，自然得在过滤器里面的 invoker.invoke(invocation) 方法前后执行。

就剩下 **6W 中 What**，事件信息该怎么传递到回调方法里。如果底层框架定制一套事件对象，不可避免将来要为维护事件对象买单，所以简单起见，不如跟随 Dubbo 接口的入参和反参来定义事件信息，免去后顾之忧。

好 6W 都想清楚了，按照想法二，我们画出逻辑图：



把 AOP 的思想充分应用到了拦截模块中，在执行下一步调用之前、之后、异常时包裹了一层。

那过滤器的代码该怎么写呢？其实 Dubbo 底层的源码已经按照我们的预想实现了一番：

复制代码

```
@Activate(group = CommonConstants.CONSUMER)
public class FutureFilter implements ClusterFilter, ClusterFilter.Listener {
    protected static final Logger logger = LoggerFactory.getLogger(FutureFilter
    @Override
    public Result invoke(final Invoker<?> invoker, final Invocation invocation)
        // 调用服务之前：执行Dubbo接口配置中指定服务中的onInvoke方法
        fireInvokeCallback(invoker, invocation);
        // need to configure if there's return value before the invocation in c
        // necessary to return future.
        // 调用服务并返回调用结果
        return invoker.invoke(invocation);
    }

    // 调用服务之后：
    // 正常返回执行Dubbo接口配置中指定服务中的onReturn方法
    // 异常返回执行Dubbo接口配置中指定服务中的onThrow方法
    @Override
    public void onResponse(Result result, Invoker<?> invoker, Invocation invoca
        if (result.hasException()) {
            // 调用出现了异常之后的应对处理
            fireThrowCallback(invoker, invocation, result.getException());
        } else {
            // 正常调用返回结果的应对处理
            fireReturnCallback(invoker, invocation, result.getValue());
        }
    }

```

```
    // 调用框架异常后：
    // 异常返回执行Dubbo接口配置中指定服务中的onThrow方法
    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation)
        fireThrowCallback(invoker, invocation, t);
    }
}
```


从源码中可以看出，利用 FutureFilter 过滤器，主要做了 3 件事情：

- 在 invoker.invoke(invocation) 方法之前，利用 fireInvokeCallback 方法反射调用了接口配置中指定服务中的 onInvoke 方法。
- 然后在 onResponse 响应时，处理了正常返回和异常返回的逻辑，分别调用了接口配置中指定服务中的 onReturn、onThrow 方法。
- 最后在 onError 框架异常后，调用了接口配置中指定服务中的 onThrow 方法。

你有没有发现，根据现状诉求来推导，逐步分析问题并实现一套简单的事件通知机制，也能达到和源码如出一辙的效果，可谓是条条大路通罗马。

如何改造

接下来就是轻松环节了，核心逻辑和三个事件逻辑该怎么写呢？我们可以直接根据源码所提供的支撑能力，重新修改 recvPay 方法：

复制代码

```
@DubboService
@Component
public class PayFacadeImpl implements PayFacade {
    @Autowired
    @DubboReference(
            /** 为 DemoRemoteFacade 的 sayHello 方法设置事件通知机制 **/
            methods = {@Method(
                    name = "sayHello",
                    oninvoke = "eventNotifyService.onInvoke",
                    onreturn = "eventNotifyService.onReturn",
                    onthrow = "eventNotifyService.onThrow")}
    )
    private DemoRemoteFacade demoRemoteFacade;

    // 商品支付功能：一个大方法
    @Override
    public PayResp recvPay(PayReq req){
```

```
        // 支付核心业务逻辑处理
        method1();
        // 返回支付结果
        return buildSuccResp();
    }
    private void method1() {
        // 省略其他一些支付核心业务逻辑处理代码
        demoRemoteFacade.sayHello(buildSayHelloReq());
    }
}

// 专门为 demoRemoteFacade.sayHello 该Dubbo接口准备的事件通知处理类
@Component("eventNotifyService")
public class EventNotifyServiceImpl implements EventNotifyService {
    // 调用之前
    @Override
    public void onInvoke(String name) {
        System.out.println("[事件通知][调用之前] onInvoke 执行.");
    }
    // 调用之后
    @Override
    public void onReturn(String result, String name) {
        System.out.println("[事件通知][调用之后] onReturn 执行.");
        // 埋点已支付的商品信息
        method2();
        // 发送支付成功短信给用户
        method3();
        // 通知物流派件
        method4();
    }
    // 调用异常
    @Override
    public void onThrow(Throwable ex, String name) {
        System.out.println("[事件通知][调用异常] onThrow 执行.");
    }
}
```


代码中，我们主要做了 4 点调整：

- 创建了一个服务类 EventNotifyServiceImpl，里面定义了 onInvoke、onReturn、onThrow 三个事件通知机制的方法。
- 按照 FutureFilter 的规定定义好 EventNotifyServiceImpl 中三个方法的入参。
- 在 demoRemoteFacade 字段修饰的 @DubboReference 注解上添加事件通知的 @Method 相关属性配置。

- 将支付核心业务逻辑处理继续留在 recvPay 方法中，然后将埋点、发短信、通知物流三个事件转移到了 EventNotifyServiceImpl.onReturn 方法中了。

通过这样的整理，我们彻底在 recvPay 方法中凸显了支付核心业务逻辑的重要性，剥离解耦了其他三件事与主体核心逻辑的边界。

当然，不光是 Dubbo 框架有这样优秀的回调机制，我们来回顾一下常用的一些事件回调机制：

- Spring 框架，使用 publishEvent 方法发布 ApplicationEvent 事件。
- Tomcat 框架，在 Javax 规范的 ServletContainerInitializer.onStartup 方法中继续循环回调 ServletContextInitializer 接口的所有实现类。
- JVM 的关闭钩子事件，当程序正常退出或调用 System.exit 或虚拟机被关闭时，会调用 Runtime.addShutdownHook 注册的线程。

类比我们今天分析的 Dubbo 源码思路，现在你可以设身处地的站在设计者兼使用者的角度再看这些源码了，相信你一定会有新体会。

我们常说源码中有很多优秀的设计值得学习，有时候并不是因为源码有多高深，而是在特定的框架中解决了某一类或某一系列的问题，而恰巧框架的这个解决方案又满足了我们在特定场景的诉求，使得我们可以在非常短的时间内实现业务需求。

事件通知的应用

事件通知的应用我们已经掌握了，不过，事件通知也只是一种机制流程，那在我们日常开发中，哪些应用场景可以考虑事件通知呢？

第一，职责分离，可以按照功能相关性剥离开，让各自的逻辑是内聚的、职责分明的。

第二，解耦，把复杂的面向过程风格的一坨代码分离，可以按照功能是技术属性还是业务属性剥离。

第三，事件溯源，针对一些事件的实现逻辑，如果遇到未知异常后还想再继续尝试重新执行的话，可以考虑事件持久化并支持在一定时间内重新回放执行。

总结

今天，我们从一个普通的商品支付功能开始，抛出如何优雅快速处理多个事件的问题，从面向过程，到简单封装，再到面向对象分析，我们采用巧妙的 6W 分析模型，推导出了与源码不谋而合的解决方案。

回顾一下 6W 分析模型要素：

- Who：谁产生的事件，是功能本身业务实现体产生，还是功能归属源码框架来产生。
- What：产生什么功能的事件，事件数据对象包含业务信息和事件凭证信息。
- When：什么时候发生的事件，在业务发生之前、业务发生之后，还是业务发生异常时发布事件。
- Where：是在哪个系统、哪个功能、哪段代码位置发生的。
- Why：解决某一类或某一系列的问题，提供一种扩展机制来丰富事件价值。
- How：怎么发布事件，是自研框架，还是扩展已有框架中具有拦截或过滤特性的机制。

Dubbo 框架中设置事件通知的简单三部曲：

- 首先，创建一个服务类，在该类中添加 onInvoke、onReturn、onThrow 三个方法。
- 其次，在三个方法中按照源码 FutureFilter 的规则定义好方法入参。
- 最后，@DubboReference 注解中或者 dubbo:reference/ 标签中给需要关注事件的 Dubbo 接口添加配置即可。

事件通知的应用场景主要有 3 类，职责分离、解耦、事件溯源。

思考题

你已经学会了事件通知机制的推导解决方案，可以说是掌握了使用和设计事件机制的精髓，今天的巩固练习是尝试研究源码，思考为什么 Dubbo 框架的事件通知机制不会因为重试机制的存在而触发多次呢？

欢迎留言参与讨论，如果有收获也欢迎分享给身边的朋友，说不定就帮他解决了一个问题，我们下一讲见。

05 思考题参考

上一期的问题是源码中，点点直连在 ReferenceConfig 中设置的 url 属性是怎么和提供方建立通信连接的。

想要解答这个问题，与其顺着源码找答案，不如逆向通过异常结果找答案。其实挺简单的，比如 url = dubbo://192.168.43.200:20883，IP 和 PORT 都是随便填的，我们就是要看报错，只有报错了，我们才知道源码在哪里发起的连接。

先想办法运行程序来个报错先：

```
Exception in thread "main" org.apache.dubbo.rpc.RpcException: Fail to create re
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.initClient(DubboProtocol
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.buildReferenceCountExcha
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.buildReferenceCountExcha
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.getSharedClient(DubboPro
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.getClients(DubboProtocol
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.protocolBindingRefer(Dub
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.refer(DubboProtocol.java
  at org.apache.dubbo.rpc.protocol.ProtocolListenerWrapper.refer(ProtocolListen
  at org.apache.dubbo.qos.protocol.QosProtocolWrapper.refer(QosProtocolWrapper.
  at org.apache.dubbo.rpc.cluster.filter.ProtocolFilterWrapper.refer(ProtocolFi
  at org.apache.dubbo.rpc.protocol.ProtocolSerializationWrapper.refer(ProtocolS
  at org.apache.dubbo.rpc.Protocol$Adaptive.refer(Protocol$Adaptive.java)
  at org.apache.dubbo.config.ReferenceConfig.createInvokerForRemote(ReferenceCo
  at org.apache.dubbo.config.ReferenceConfig.createProxy(ReferenceConfig.java:3
  at org.apache.dubbo.config.ReferenceConfig.init(ReferenceConfig.java:285)
  at org.apache.dubbo.config.ReferenceConfig.get(ReferenceConfig.java:219)
  at com.hmilyylimh.cloud.generic.string.Dubbo15GenericStringConsumerApplicatio
  at com.hmilyylimh.cloud.generic.string.Dubbo15GenericStringConsumerApplicatio
Caused by: org.apache.dubbo.remoting.RemotingException: client(url: dubbo://192
  at org.apache.dubbo.remoting.transport.netty4.NettyClient.doConnect(NettyClie
  at org.apache.dubbo.remoting.transport.AbstractClient.connect(AbstractClient.
  at org.apache.dubbo.remoting.transport.AbstractClient.<init>(AbstractClient.j
  at org.apache.dubbo.remoting.transport.netty4.NettyClient.<init>(NettyClient.
  at org.apache.dubbo.remoting.transport.netty4.NettyTransporter.connect(NettyT
  at org.apache.dubbo.remoting.Transporter$Adaptive.connect(Transporter$Adaptiv
  at org.apache.dubbo.remoting.Transporters.connect(Transporters.java:74)
  at org.apache.dubbo.remoting.exchange.support.header.HeaderExchanger.connect(
  at org.apache.dubbo.remoting.exchange.Exchangers.connect(Exchangers.java:107)
  at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol.initClient(DubboProtocol
  ... 17 more
```

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，但你按照泛化调用的方式指定 url 属性后发起调用发生异常后，看到的日志格式也是大同小异的。

从异常信息中，我们可以看到 RpcException 和 RemotingException 异常类，在 RemotingException 的下面，可以看到使用了 netty4 版本，调用 NettyClient.doConnect 发生了连接异常。

紧接着进入到 NettyClient.doConnect 代码中：

复制代码

```
protected void doConnect() throws Throwable {
    long start = System.currentTimeMillis();
    // 调用 Netty 框架中的 Bootstrap 类与提供方发起连接请求
    ChannelFuture future = bootstrap.connect(getConnectAddress());
    // 省略了其他的代码
}
```

由此可知，消费方与提供方想建立连接请求，默认是通过 netty 通信框架，源码中使用 Bootstrap.connect 来发起连接请求。

到这里，你就知道在哪里发起了连接请求了。想继续深入，你可以顺着 Bootstrap.connect 这个方法一直探究，直至找到 io.netty.channel.nio.AbstractNioChannel 这个抽象类，绑定、连接、断开等操作都在这个抽象类中，这是客户端与服务端建立通信连接最重要的类。

上一篇　05 | 点点直连：点对点搭建产线“后门”的万能管控

下一篇　07 | 参数验证：写个参数校验居然也会被训？

精选留言 (2)

hello

2023-01-29 来自广东

虽然埋点、发短信、通知物流三个事件转移到了 EventNotifyServiceImpl.onReturn 方法中了，如果后续继续增加 发邮件 啥的功能还是会出现同样的问题。当然，这篇文章站在分析问题的角度解读源码就另当别论了。

作者回复: 你好，hello：是滴，能看出你已经有了自己独有的理解想法了，这里是通过引出案例分析解耦理解源码思想而已。

熊猫

2022-12-30 来自广东

老师，你好，怎么解决当dubbo服务调用返回的对象中有枚举类型，当枚举类型变更时，报错的问题？

作者回复: 你好，熊猫：我有点没太理解，当枚举值变更时，是消费方判断报文的值不在自己的范围内么？还是哪种情况？

共 2 条评论 >