

## 07 | 参数验证：写个参数校验居然也会被训？

2023-01-02 何辉 来自北京



天下无鱼

<https://shikey.com/>

《Dubbo源码剖析与实战》

[课程介绍 >](#)



讲述：何辉

时长 16:17 大小 14.88M



你好，我是何辉。今天我们探索 Dubbo 框架的第六道特色风味，参数验证。

说到参数校验，相信你一定是又爱又恨。在发送或接收请求的时候，必要的参数校验可以拦截非法请求，提升请求质量，这样一个简单的数值比对逻辑看起来很简单，但写的过程却很枯燥和乏味，一不留神就会导致一些必要性校验没考虑到。

现在你的同事小马就因为漏写了参数校验被老大训话了，来看他写的一段消费方调用提供方的代码：

```
1 ///////////////////////////////////////////////////  
2 // 消费方的一段调用下游 validateUser 的代码  
3 ///////////////////////////////////////////////////  
4 @Component  
5 public class InvokeDemoFacade {  
6     @DubboReference
```

复制代码

```

7     private ValidationFacade validationFacade;
8
9     // 一个简单的触发调用下游 ValidationFacade.validateUser 的方法
10    public String invokeValidate(String id, String name, String
11        // 调用下游接口
12        return validationFacade.validateUser(new ValidateUserInfo(id, name, sex
13    }
14 }
15
16 //////////////////////////////////////
17 // 提供方的一段接收 validateUser 请求的代码
18 //////////////////////////////////////
19 @DubboService
20 @Component
21 public class ValidationFacadeImpl implements ValidationFacade {
22     @Override
23     public String validateUser(ValidateUserInfo userInfo) {
24         // 这里就象征性地模拟一下业务逻辑
25         String retMsg = "Ret: "
26             + userInfo.getId()
27             + "," + userInfo.getName()
28             + "," + userInfo.getSex();
29         System.out.println(retMsg);
30         return retMsg;
31     }
32 }

```



老大看完这段代码后，把小马叫了过来。

老大：小马，你过来一下。

小马：好的，老大。

老大：我刚刚在 CR 你的代码，发现你写的这个 `validateUser` 方法，你先看看有没有什么不妥的？

小马：这不就是个校验用户的方法么，接收请求然后直接调用下游，额～我没有看出啥不妥的～

老大：你再想想，如果我随意传入一些空值给下游会怎么样？

小马：额～可能会出现空指针或者其他意外的异常，总之会导致一些不必要的报错。

老大：你都知道会出现不必要的异常发生，是不是该做点什么呢？明明能预见的问题，为什么没有好好处理？一定要等到测试测出问题，或者用户上报系统异常后，再火急火燎地修改线上紧急 bug 么？

小马：老大，我知道错了，我这就回去认真改下。

老大：还有，提供方这边代码也是一样，明显预见可能有参数不合法情况，为什么不前置校验一下参数的合法性呢？你写的另外几个方法也有类似的问题。你回去把这个校验参数的逻辑

辑好好改改，这种低级问题希望别有下次了。

小马：嗯，我这就去改。



看完这段对话，不知道有没有让你回想起当年被训斥的场景，面对这样的一个参数验证合法性情况，你会怎么处理呢？

## 三种验证方案

想要知道如何处理，我们先来梳理下老大指出的几个问题：

- 问题 1：消费方代码，在调用下游的 `validateUser` 方法时，没有预先做一些参数的合法性校验。
- 问题 2：提供方代码，服务方代码在接收请求的时候，没有对一些必要的字段进行合法性校验。
- 问题 3：不仅仅是 `validateUser` 方法，其他几个方法也没有对参数进行合法性校验。

问题都已经罗列出来了，如何修改呢？

### 1. 简单验证

最直接的思路就是，哪里有锅补哪里。这三个问题不就是要对几个方法的参数进行合法性校验嘛，那就简单点，哪里用到了，我们就在哪里提前预判一下，保证不让有问题的参数污染正常的业务逻辑就行。

于是我们写出了这样的代码：

复制代码

```
1  ////////////////////////////////////////////
2  // 简单验证：消费方的一段调用下游 validateUser 的代码
3  ////////////////////////////////////////////
4  @Component
5  public class InvokeDemoFacade {
6      @DubboReference
7      private ValidationFacade validationFacade;
8
9      // 一个简单的触发调用下游 ValidationFacade.validateUser 的方法
10     public String invokeValidate(String id, String name, String sex) {
11         // 校验 id 属性必填
12         if (StringUtils.isBlank(id)) {
```

```

13         throw new RuntimeException("id 不能为空");
14     }
15     // 校验 name 属性的长度必须在 5~10 之间
16     if (StringUtils.isNotBlank(name) && (name.length() < 5 || name.length()
17         throw new RuntimeException("name 必须在 5~10 个长度之间"));
18     }
19     // 然后再调用下游接口
20     return validationFacade.validateUser(new ValidateUserInfo(id, name, sex
21 }
22 }
23
24 ///////////////////////////////////////////////////
25 // 简单验证: 提供方的一段接收 validateUser 请求的代码
26 ///////////////////////////////////////////////////
27 @DubboService
28 @Component
29 public class ValidationFacadeImpl implements ValidationFacade {
30     @Override
31     public String validateUser(ValidateUserInfo userInfo) {
32         // 校验 id 属性必填
33         if (StringUtils.isBlank(userInfo.getId())) {
34             throw new RuntimeException("[provider] id 不能为空");
35         }
36         // 校验 name 属性的长度必须在 5~10 之间
37         String name = userInfo.getName();
38         if (StringUtils.isNotBlank(name) && (name.length() < 5 || name.length()
39             throw new RuntimeException("[provider] name 必须在 5~10 个长度之间");
40         }
41
42         // 这里就象征性的模拟下业务逻辑
43         String retMsg = "Ret: "
44             + userInfo.getId()
45             + "," + userInfo.getName()
46             + "," + userInfo.getSex();
47         System.out.println(retMsg);
48         return retMsg;
49     }
50 }

```

代码中的改变也比较简单直接，主要有 2 点：

- 消费方代码，在真正发起 `validationFacade.validateUser` 方法调用之前，先判断了 `id` 值不能为空，然后判断了 `name` 值的长度必须在 5~10 之间。
- 提供方代码，同样是在处理核心业务逻辑之前，针对 `id`、`name` 进行合法性判断，若不合法就以异常的方式告诉调用方。

轻轻松松改完一个 `validateUser` 方法的代码，这下应该没啥大问题了吧。

再看老大指出的另外几个方法，怎么办呢？用同样的思路依葫芦画瓢处理下，在消费方调用下游之前检查一下参数的合法性，提供方的代码也再次对入参进行验证一下？



少数几个还能这么做，但你也想到了，以后还有十个、百个方法也要按照这样的方式处理，头就大，好像不能这么干了，应该要找个省事点的、代码少的方式来处理。

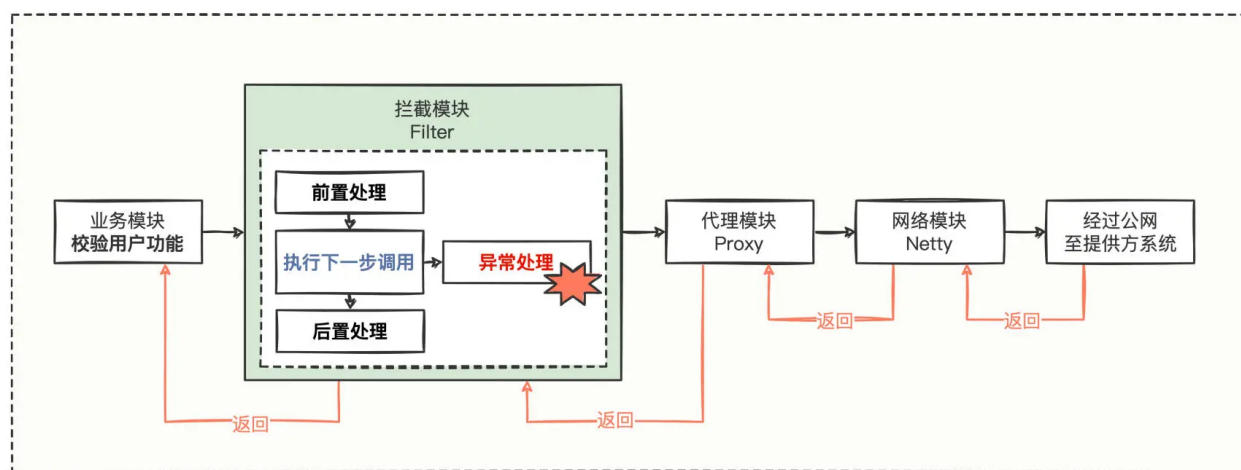
可该如何更省事点呢？

## 2. 通知验证

想省事，我们就不能在代码的各个角落写各种校验的逻辑了，得找个比较集中的地方来写这段校验逻辑，并且这个地方还必须发生在真实调用之前。

回忆上一讲学过的“事件通知”，你有没有灵光一闪，事件通知正好可以在调用之前、之后、异常时触发用户自定义的通知处理类，而我们对参数校验的要求，就是需要在调用之前拦截一下。

貌似可以实现，根据这个思路，我们稍微调整一下事件通知的链路：



极客时间

校验用户功能，在调用下游的 `validateUser` 用户之前拦截一下，然后按照事件通知的约束规范，写一个校验用户的事件通知类，就可以比较轻松地解决问题了。兴奋的你立马写出了消费方的代码改造：



```
1 ///////////////////////////////////////////////////
2 // 通知验证：消费方的一段调用下游 validateUser 的代码
3 ///////////////////////////////////////////////////
4 @Component
5 public class InvokeDemoFacade {
6     @DubboReference(
7         /** 接口调研超时时间，1毫秒 **/
8         timeout = 5000,
9         /** 启动时不检查 DemoFacade 是否能正常提供服务 **/
10        check = false,
11        /** 为 ValidationFacade 的 validateUser 方法设置事件通知机制 **/
12        methods = {@Method(
13            name = "validateUser",
14            oninvoke = "eventValidateUserService.onInvoke",
15            onreturn = "eventValidateUserService.onReturn",
16            onthrow = "eventValidateUserService.onThrow"})}
17    )
18    private ValidationFacade validationFacade;
19    // 一个简单的触发调用下游 ValidationFacade.validateUser 的方法
20    public String invokeValidate(String id, String name, String sex) {
21        return validationFacade.validateUser(new ValidateUserInfo(id, name, sex)
22    }
23 }
24
25 ///////////////////////////////////////////////////
26 // 专门为调用下游 validateUser 方法时定制的一套事件通知类
27 ///////////////////////////////////////////////////
28 @Component("eventValidateUserService")
29 public class EventValidateUserServiceImpl implements EventValidateUserService {
30     // 调用之前
31     @Override
32     public void onInvoke(ValidateUserInfo userInfo){
33         System.out.println("[事件通知][调用之前] onInvoke 执行.");
34         // 校验 id 属性必填
35         if (StringUtils.isBlank(userInfo.getId())) {
36             throw new RuntimeException("[onInvoke] id 不能为空");
37         }
38         // 校验 name 属性的长度必须在 5~10 之间
39         String name = userInfo.getName();
40         if (StringUtils.isNotBlank(name) && (name.length() < 5 || name.length()
41             throw new RuntimeException("[onInvoke] name 必须在 5~10 个长度之间");
42         }
43     }
44     // 调用之后
45     @Override
46     public void onReturn(String result, ValidateUserInfo userInfo){
47         System.out.println("[事件通知][调用之后] onReturn 执行.");
48     }
49     // 调用异常
50     @Override
51     public void onThrow(Throwable ex, ValidateUserInfo userInfo){
52         System.out.println("[事件通知][调用异常] onThrow 执行.");
53     }
```

代码做出的改变主要有 3 点：

- 新增了一个专门拦截调用下游 `validateUser` 方法的事件通知实现类 `EventValidateUserServiceImpl`。
- 在 `InvokeDemoFacade` 中，`validationFacade` 字段的 `@DubboReference` 注解，定义了 `methods` 属性，专门指向 `validateUser` 方法，并为该方法配上了事件通知处理类。
- `ValidateUserInfo` 对象中的 `id`、`name` 字段的校验逻辑，完全搬到了事件通知类的 `onInvoke` 方法中。

接下来，我们只需要把调用其他下游的方法，都配上事件通知类即可，能复用就复用，不能复用就新写，这样应该挺完美了。

正常逻辑很流畅，但思维缜密的你考虑到异常逻辑，发现了一个细节问题：在事件通知处理类的 `onInvoke` 方法中，参数不合法会抛异常，那抛出的这个异常会中断调用流程么？或者说 **`onInvoke` 抛出异常后，消费方还会把请求发到提供方么？**

一语中的，我们在消费方写段代码调用一下看看效果：

 复制代码

```
1 // 从 Spring 的上下文中拿到 InvokeDemoFacade 的实例对象
2 InvokeDemoFacade invokeDemoFacade = SpringCtxUtils.getBean(InvokeDemoFacade.class);
3 // 然后想办法触发调用一下 invokeValidate 方法，目的是需要调用下游的 validateUser 方法
4 String retMsg = invokeDemoFacade.invokeValidate("1", "Geek", "F");
5 // 然后将返回的结果打印出来
6 System.out.println(retMsg);
```

运行之后，得到如下结果：

 复制代码

```
1 [事件通知][调用异常] onThrow 执行。
2 [事件通知][调用之后] onReturn 执行。
3 Ret: 1,Geek,F
```

现实的结果和我们预想的不太一样，明明拦截到了，也抛了异常，为什么还能拿到返回结果呢？



看来我们得到提供事件通知机制的过滤器 `FutureFilter` 里面探个究竟：

 复制代码

```
1 // FutureFilter#fireInvokeCallback, 通过反射调用通知处理类的 onInvoke 方法
2 private void fireInvokeCallback(final Invoker<?> invoker, final Invocation invc
3     // 省略了部分其他代码
4     Object[] params = invocation.getArguments();
5     try {
6         // 这里会反射调用 EventValidateUserServiceImpl 的 onInvoke 方法
7         onInvokeMethod.invoke(onInvokeInst, params);
8     } catch (InvocationTargetException e) {
9         // 如果 onInvoke 进入了这里的异常，会继续回调 onThrow 方法
10        fireThrowCallback(invoker, invocation, e.getTargetException());
11    } catch (Throwable e) {
12        // 如果 onInvoke 进入了这里的异常，会继续回调 onThrow 方法
13        fireThrowCallback(invoker, invocation, e);
14    }
15 }
16
17 // FutureFilter#fireThrowCallback, 通过反射调用通知处理类的 onThrow 方法
18 private void fireThrowCallback(final Invoker<?> invoker, final Invocation invoc
19     // 省略了部分其他代码
20     Class<?>[] rParaTypes = onthrowMethod.getParameterTypes();
21     if (rParaTypes[0].isAssignableFrom(exception.getClass())) {
22         try {
23             // 省略了部分其他代码
24             // 这里会反射调用 EventValidateUserServiceImpl 的 onThrow 方法
25             onthrowMethod.invoke(onthrowInst, params);
26         } catch (Throwable e) {
27             // 如果 onThrow 进入了这里的异常，则只是打个错误日志而已，会吞掉异常
28             logger.error(invocation.getMethodName() + ".call back method invoke
29         }
30     } else {
31         logger.error(invocation.getMethodName() + ".call back method invoke err
32     }
33 }
```

在 `FutureFilter` 的源码中能发现，无论如何，不管是 `onInvoke` 抛出了异常，还是 `onInvoke` 抛出的异常进入了 `onThrow` 逻辑，甚至是 `onThrow` 逻辑抛出的异常，都会被吞掉，也就是说，事件通知给我们提供了一种安全的调用机制，无论在通知机制的实现类中发生什么样的异常，最终都不会影响程序继续往后执行。



所以，到这里我们发现事件通知机制会吃掉参数校验抛出的异常，无法阻断流程，这个方法行不通了。这也是源码有时候不尽如人意的地方，但是反过来想这也是一种约束，毕竟事件通知不是专门为参数校验而设计的，它不是一个万能的 USB 设备，谁插上了就能用。

天下无鱼  
<https://shikey.com/>

既然事件通知机制不能满足我们的诉求，那我们是不是可以仿照事件通知机制的做法，自己实现一套校验机制呢？

### 3. 统一验证

顺着这个想法我们继续思考。

先梳理一下大思路，类似事件机制，我们可以做到拦截所有请求，请求会触发过滤器的 `invoke` 方法，然后又因为 `invoke` 方法中的 `invocation` 对象封装了任意接口的请求参数，我们就只需要拿着 `invocation` 里面的入参值进行挨个校验。

这个思路好像挺可行的，我们继续思考细节。

假使从 `invocation` 里面拿到了一个个参数值，虽然知道每个字段值是什么，但是对每个字段按照怎么样的规则进行校验，这个标准是缺乏的，**所以需要一个可衡量的、针对字段按照怎样的规则进行校验的标准产物，而且这个标准产物还得必须在 `invoke` 被触发调用的运行时态能获取到。**

回忆我们所学的 Java 知识点，哪个是可以在运行时被获取到，能跟着任何字段走，并且字段的校验规则不同还能提供不同的内容呢？


你一定想到了——注解。

没错，注解就是我们的标准产物。只要自定义一个注解，在注解中提供我们的校验规则，然后在 `invocation` 中找字段时，顺便看看这个字段注解中的校验规则是怎么样，一五一十地按照校验规则校验字段值不就万事大吉了么。

每个环节都想清楚，接下来我们就可以准备动手定义一个过滤器了。

别急，我这里教你一个小技巧，**像过滤器这种具有拦截所有请求机制功能的类，一定要先看看你所在系统的相关底层能力支撑，说不定类似的功能已经存在，我们就能物尽其用了。**具体步

步骤就是：

- 首先找到具有拦截机制的类，这里就是 `org.apache.dubbo.rpc.Filter` 过滤器。 <https://shikey.com/>
- 其次找到该 `org.apache.dubbo.rpc.Filter` 过滤器的所有实现类。
- 最后认真阅读每个过滤器的类名，翻阅一下每个过滤器的类注释，看看有什么用。

我们按照小技巧操作一下，`org.apache.dubbo.rpc.Filter` 接口下有好多个实现类：

 复制代码

```
1 Filter (org.apache.dubbo.rpc)
2 - TokenFilter (org.apache.dubbo.rpc.filter)
3 - MetricsFilter (org.apache.dubbo.monitor.dubbo)
4 - DeprecatedFilter (org.apache.dubbo.rpc.filter)
5 - ClassLoaderCallbackFilter (org.apache.dubbo.rpc.filter)
6 - CacheFilter (org.apache.dubbo.cache.filter)
7 - ActiveLimitFilter (org.apache.dubbo.rpc.filter)
8 - ConsumerContextFilter (org.apache.dubbo.rpc.filter)
9 - ValidationFilter (org.apache.dubbo.validation.filter)
10 - GenericImplFilter (org.apache.dubbo.rpc.filter)
11 - MetricsFilter (org.apache.dubbo.monitor.support)
12 - CompatibleFilter (org.apache.dubbo.rpc.filter)
13 - ProfilerServerFilter (org.apache.dubbo.rpc.filter)
14 - ClassLoaderFilter (org.apache.dubbo.rpc.filter)
15 - ExceptionFilter (org.apache.dubbo.rpc.filter)
16 - ProviderAuthFilter (org.apache.dubbo.auth.filter)
17 - ListenableFilter (org.apache.dubbo.rpc)
18 - FutureFilter (org.apache.dubbo.rpc.protocol.dubbo.filter)
19 - AccessLogFilter (org.apache.dubbo.rpc.filter)
20 - TimeoutFilter (org.apache.dubbo.rpc.filter)
21 - TraceFilter (org.apache.dubbo.rpc.protocol.dubbo.filter)
22 - ConsumerSignFilter (org.apache.dubbo.auth.filter)
23 - TpsLimitFilter (org.apache.dubbo.rpc.filter)
24 - GenericFilter (org.apache.dubbo.rpc.filter)
25 - MonitorFilter (org.apache.dubbo.monitor.support)
26 - ExecuteLimitFilter (org.apache.dubbo.rpc.filter)
27 - ContextFilter (org.apache.dubbo.rpc.filter)
28 - Filter (com.alibaba.dubbo.rpc)
29 - EchoFilter (org.apache.dubbo.rpc.filter)
```

耐心些一路看下去，在第 9 行，你会发现一个 `ValidationFilter` 类，通过类名的英文单词能大致看出是一个验证的过滤器。是不是底层框架已经做了我们想做的事情了呢？我们进入源码一探究竟。

ValidationFilter 的类注释信息是这样的：



```
1 ValidationFilter invoke the validation by finding the right Validator instance
```

简单理解就是，ValidationFilter 会根据 url 配置的 validation 属性值找到正确的校验器，在方法真正执行之前触发调用校验器执行参数验证逻辑。

类注释里还举了个例子：

 复制代码

```
1 e.g. <dubbo:method name="save" validation="jvalidation" />
```

可以在方法层面添加 validation 属性，并设置属性值为 jvalidation，这样就可以正常使用底层提供的参数校验机制了。

还提到特殊设置方式：

 复制代码

```
1 e.g. <dubbo:method name="save" validation="special" />
2 where "special" is representing a validator for special character.
3 special=xxx.yyy.zzz.SpecialValidation under META-INF folders org.apache.dubbo.v
```

可以在 validation 属性的值上，填充一个自定义的校验类名，并且嘱咐我们记得将自定义的校验类名添加到 META-INF 文件夹下的 org.apache.dubbo.validation.Validation 文件中。

从源码层面了解一番后，我们找到了现成的工具，可以尽情改造了。

## 代码改造

改造前，我们还是先梳理下要改造的地方：

1. 为调用下游的接口添加 validation 属性。
2. 从源码中寻找能提供校验规则的标准产物，也就是注解。

3. 在下游的方法入参对象中，为需要校验的字段添加注解。

这里有个小难题，标准产物怎么寻找呢？



同样地，先别着急自己写，Dubbo 作为一个无数开发者使用过的优秀框架，说不定已经想我们所想，做我们所做了。

顺着 `ValidationFilter` 的源码逻辑，不用精细研读每行代码，在大致浏览代码调用链路的过程中，注意看不同的方法，有没有关于 `check`、`verify`、`validator` 之类校验单词的 `jar` 包或包路径，你会有意想不到的收获。

我们再次进入 `ValidationFilter` 源码类看看，找到 `invoke` 方法，顺着逻辑逐行点进每个方法：

复制代码

```
1 // org.apache.dubbo.validation.filter.ValidationFilter.invoke
2 public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcExcep
3     // Validation 接口的代理类被注入成功，且该调用的方法有 validation 属性
4     if (validation != null && !invocation.getMethodName().startsWith("$")
5         && ConfigUtils.isNotEmpty(invoker.getUrl().getMethodParameter(invoc
6     try {
7         // 接着通过 url 中 validation 属性值，并且为该方法创建对应的校验实现类
8         Validator validator = validation.getValidator(invoker.getUrl());
9         if (validator != null) {
10             // 若找到校验实现类的话，则真正开启对方法的参数进行校验
11             validator.validate(invocation.getMethodName(), invocation.getPa
12         }
13     } catch (RpcException e) {
14         // RpcException 异常直接抛出去
15         throw e;
16     } catch (Throwable t) {
17         // 非 RpcException 异常的话，则直接封装结果返回
18         return AsyncRpcResult.newDefaultAsyncResult(t, invocation);
19     }
20 }
21 // 能来到这里，说明要么没有配置校验过滤器，要么就是校验了但参数都合法
22 // 既然没有抛异常的话，那么就直接调用下一个过滤器的逻辑
23 return invoker.invoke(invocation);
24 }
25
26 // org.apache.dubbo.validation.support.AbstractValidation.getValidator
27 public Validator getValidator(URL url) {
28     // 将 url 转成字符串形式
29     String key = url.toFullString();
30     // validators 是一个 Map 结构，即底层可以说明每个方法都可以有不同的校验器
31     Validator validator = validators.get(key);
```

```
32     if (validator == null) {
33         // 若通过 url 从 Map 结构中找不到 value 的话, 则直接根据 url 创建一个校验器实现类
34         // 而且 createValidator 是一个 protected abstract 修饰的
35         // 说明是一种模板方式, 创建校验器实现类, 是可被重写重新创建自定义的校验器
36         validators.put(key, createValidator(url));
37         validator = validators.get(key);
38     }
39     return validator;
40 }
41
42 // org.apache.dubbo.validation.support.jvalidation.JValidation
43 public class JValidation extends AbstractValidation {
44     @Override
45     protected Validator createValidator(URL url) {
46         // 创建一个 Dubbo 框架默认的校验器
47         return new JValidator(url);
48     }
49 }
50
51 // org.apache.dubbo.validation.support.jvalidation.JValidator
52 public class JValidator implements Validator {
53     // 省略其他部分代码
54     // 进入到 Dubbo 框架默认的校验器中, 发现真实采用的是 javax 第三方的 validation 插件
55     // 由此, 我们应该找到了标准产物的关键突破口了
56     private final javax.validation.Validator validator;
57 }
```

跟踪源码的过程:

- 先找到 ValidationFilter 过滤器的 invoke 入口。
- 紧接着找到根据 validation 属性值创建校验器的 createValidator 方法。
- 然后发现创建了一个 JValidator 对象。
- 在该对象中发现了关于 javax 包名的第三方 validation 插件。

最终我们确实发现了一个第三方 validation 插件, 顺藤摸瓜你可以找到对应的 maven 坐标:

 复制代码

```
1 <dependency>
2     <groupId>org.hibernate</groupId>
3     <artifactId>hibernate-validator</artifactId>
4 </dependency>
```

如愿发现了 **hibernate-validator** 这个第三方插件。但你可能会问了，我们还是没能找到这个标准产物啊，问题还是没有解决？



不急，既然找到了第三方插件，不妨将这个 **maven** 坐标引入到消费方工程中，再进入到该插件的 **pom** 文件中，看看有没有意外的收获。我们进入 **hibernate-validator** 插件的 **pom** 中，能看到里面引用了一个 **validation-api** 插件坐标：

复制代码

```
1 <dependency>
2     <groupId>javax.validation</groupId>
3     <artifactId>validation-api</artifactId>
4 </dependency>
```

现在你再进入这个坐标对应的 **java** 代码中，会看到了一堆注解，比如 **NotNull**、**Max**、**Size** 等等，说明**这些注解天生就可以设置不同的校验规则**。现在标准产物找到了，我们可以放心改造了。

万事俱备，只欠代码，改造如下：

复制代码

```
1 ///////////////////////////////////////////////////
2 // 统一验证：下游 validateUser 的方法入参对象
3 ///////////////////////////////////////////////////
4 @Setter
5 @Getter
6 public class ValidateUserInfo implements Serializable {
7     private static final long serialVersionUID = 1558193327511325424L;
8     // 添加了 @NotBlank 注解
9     @NotBlank(message = "id 不能为空")
10    private String id;
11    // 添加了 @Length 注解
12    @Length(min = 5, max = 10, message = "name 必须在 5~10 个长度之间")
13    private String name;
14    // 无注解修饰
15    private String sex;
16 }
17
18 ///////////////////////////////////////////////////
19 // 统一验证：消费方的一段调用下游 validateUser 的代码
20 ///////////////////////////////////////////////////
21 @Component
22 public class InvokeDemoFacade {
23
```



```

24 // 注意, @DubboReference 这里添加了 validation 属性
25 @DubboReference(validation = "jvalidation")
26 private ValidationFacade validationFacade;
27
28 // 一个简单的触发调用下游 ValidationFacade.validateUser 的方法
29 public String invokeValidate(String id, String name, String sex) {
30     return validationFacade.validateUser(new ValidateUserInfo(id, name, sex
31 }
32 }
33
34 //////////////////////////////////////
35 // 统一验证: 提供方的一段接收 validateUser 请求的代码
36 //////////////////////////////////////
37 // 注意, @DubboService 这里添加了 validation 属性
38 @DubboService(validation = "jvalidation")
39 @Component
40 public class ValidationFacadeImpl implements ValidationFacade {
41     @Override
42     public String validateUser(ValidateUserInfo userInfo) {
43         // 这里就象征性的模拟下业务逻辑
44         String retMsg = "Ret: "
45             + userInfo.getId()
46             + "," + userInfo.getName()
47             + "," + userInfo.getSex();
48         System.out.println(retMsg);
49         return retMsg;
50     }
51 }

```



天下无鱼

<https://shikey.com/>

其他代码没有多大变化, 主要改动是 3 点, 也就是我们前面梳理的需要改造的 3 点:

1. 提供方将方法入参的 id、name 字段添加了注解。
2. 提供方在 ValidationFacadeImpl 类的 @DubboService 注解中添加了 validation 属性, 属性对应的值为 jvalidation。
3. 消费方在调用提供方时, 在 InvokeDemoFacade 中给 validationFacade 字段的 @DubboReference 注解中也添加了一个 validation 属性, 属性对应的值也为 jvalidation。

代码写完, 再回过头来看看老大训斥的 3 个问题有没有解决:

- 问题 1: 消费方代码, 在调用下游的 validateUser 方法时, 没有预先做一些参数的合法性校验。

- 问题 2: 提供方代码, 服务方代码在接收请求的时候, 没有对一些必要的字段进行合法性校验。
- 问题 3: 不仅仅是 `validateUser` 方法, 其他几个方法也没有对参数进行合法性校验。



天下无鱼

<http://shikey.com/>

问题 1 我们可以通过在 `@DubboReference` 注解中添加 `validation` 属性解决, 问题 2 我们可以在 `@DubboService` 注解中添加 `validation` 属性解决, 问题 3 可以参考问题 1 和 2。

这样, 我们轻松做到了既能在消费方提前预判参数的合法性, 也能在提供方进行参数的兜底校验, 还能让代码更加精简提升编码效率, 减少大量枯燥无味的雷同代码。

## 参数验证的应用

但是这种简单的参数校验也不是万能的, 在我们实际应用开发过程中, 哪些应用场景可以考虑这种参数校验呢?

- 第一, 单值简单规则判断, 各个字段的校验逻辑毫无关联、相互独立。
- 第二, 提前拦截掉脏请求, 尽可能把一些参数值不合法的情况提前过滤掉, 对于消费方来说尽量把高质量的请求发往提供方, 对于提供方来说, 尽量把非法的字段值提前拦截, 以此保证核心逻辑不被脏请求污染。
- 第三, 通用网关校验领域, 在网关领域部分很少有业务逻辑, 但又得承接请求, 对于不合法的参数请求就需要尽量拦截掉, 避免不必要的请求打到下游系统, 造成资源浪费。

## 总结

对于小马写的参数校验, 老大 CR 后发现好几个方法并不完善, 我们分别从简单验证、通知验证、统一验证三种方式分析并尝试解决。

- 简单验证, 好处是简单直接, 不用过多抽象封装代码, 但会导致代码充斥着大量价值不高的雷同代码。
- 通知验证, 借鉴事件通知采用拦截的思路, 但是忽略了事件通知机制的特定应用场景, 事件通知机制会吞掉各种实现类抛出的异常。
- 统一验证, 继续思考拦截的思路, 自定义过滤器, 并在源码中找到了破解之道。

从源码中我们发现 `Dubbo` 参数校验有两种方式:

- 一般方式，设置 `validation` 为 `jvalidation`、`jvalidationNew` 两种框架提供的值。
- 特殊方式，设置 `validation` 为自定义校验器的类路径，并将自定义的类路径添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.validation.Validation` 文件中。



如何进行参数校验改造，也有通用三部曲：

- 首先，寻找具有拦截机制的接口，且该接口具有读取请求对象数据的能力。
- 其次，寻找一套注解来定义校验的标准规则，并将该注解修饰到请求对象的字段上。
- 最后，寻找一套校验逻辑来根据注解的标准规则来校验字段值，提供通用的校验能力。

参数校验的应用场景主要有 3 类，单值简单规则判断、降低无谓的脏请求、通用网关校验领域。

## 思考题

你已经掌握了在 `Dubbo` 中如何更加优雅地进行简单的参数验证，这里留一个小实验让你活学活用，参考参数校验的通用三部曲，尝试在 `Spring` 的切面中完成对参数的统一验证。

欢迎写下你的思路和代码，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。

## 06 思考题参考

`Dubbo` 框架的事件通知机制，即使有重试机制的存在，但也只会触发一次。这个问题简单，我比较推荐在异常中寻找答案，比如通过让调用出现超时异常。

先写上这样一段简单的模拟调用代码：

复制代码

```
1 @Component
2 public class InvokeDemoFacade {
3     @Autowired
4     @DubboReference(
5         /** 接口调研超时时间，1毫秒 **/
6         timeout = 1,
7         /** 启动时不检查 DemoFacade 是否能正常提供服务 **/
8         check = false,
9     )
```

```

10      /** 为 DemoFacade 的 sayHello 方法设置事件通知机制 **/
11      methods = {@Method(
12          name = "sayHello",
13          oninvoke = "eventNotifyService.onInvoke",
14          onreturn = "eventNotifyService.onReturn",
15          onthrow = "eventNotifyService.onThrow")}]
16  )
17  private DemoFacade demoFacade;
18
19  // 简单的一个 hello 方法，然后内部调用下游Dubbo接口 DemoFacade 的 sayHello 方法
20  public String hello(String name){
21      return demoFacade.sayHello(name);
22  }
23  }

```

代码比较简单，定义了一个 `InvokeDemoFacade` 类，接着在里面定义一个 `hello` 方法，然后在 `hello` 方法里面调用下游的 Dubbo 接口 `DemoFacade` 的 `sayHello` 方法。同时为 `demoFacade` 字段添加了 `@DubboReference` 注解，并在 `@DubboReference` 注解中为下游的 `sayHello` 方法添加了事件通知配置。

这里需要特别关注的是 `timeout = 1` 这个 1 毫秒的设置，目的是为了重试多次最后抛异常，方便查看调用栈关系。

然后在 `org.apache.dubbo.rpc.cluster.support.FailoverClusterInvoker#doInvoke` 方法开头打个断点，想办法以 `Debug` 方式触发调用一下 `InvokeDemoFacade.hello` 方法，接着你会发现这样的调用栈关系：

 复制代码

```

1  doInvoke:58, FailoverClusterInvoker (org.apache.dubbo.rpc.cluster.support)
2  invoke:340, AbstractClusterInvoker (org.apache.dubbo.rpc.cluster.support)
3  invoke:46, RouterSnapshotFilter (org.apache.dubbo.rpc.cluster.router)
4  invoke:321, FilterChainBuilder$CopyOfFilterChainNode (org.apache.dubbo.rpc.clus
5  invoke:99, MonitorFilter (org.apache.dubbo.monitor.support)
6  invoke:321, FilterChainBuilder$CopyOfFilterChainNode (org.apache.dubbo.rpc.clus
7  invoke:51, FutureFilter (org.apache.dubbo.rpc.protocol.dubbo.filter)
8  invoke:321, FilterChainBuilder$CopyOfFilterChainNode (org.apache.dubbo.rpc.clus
9  invoke:109, ConsumerContextFilter (org.apache.dubbo.rpc.cluster.filter.support)
10 invoke:321, FilterChainBuilder$CopyOfFilterChainNode (org.apache.dubbo.rpc.clus
11 invoke:193, FilterChainBuilder$CallbackRegistrationInvoker (org.apache.dubbo.rp
12 invoke:92, AbstractCluster$ClusterFilterInvoker (org.apache.dubbo.rpc.cluster.s
13 invoke:97, MockClusterInvoker (org.apache.dubbo.rpc.cluster.support.wrapper)
14 invoke:280, MigrationInvoker (org.apache.dubbo.registry.client.migration)
15 invoke:57, InvocationUtil (org.apache.dubbo.rpc.proxy)
16 invoke:73, InvokerInvocationHandler (org.apache.dubbo.rpc.proxy)

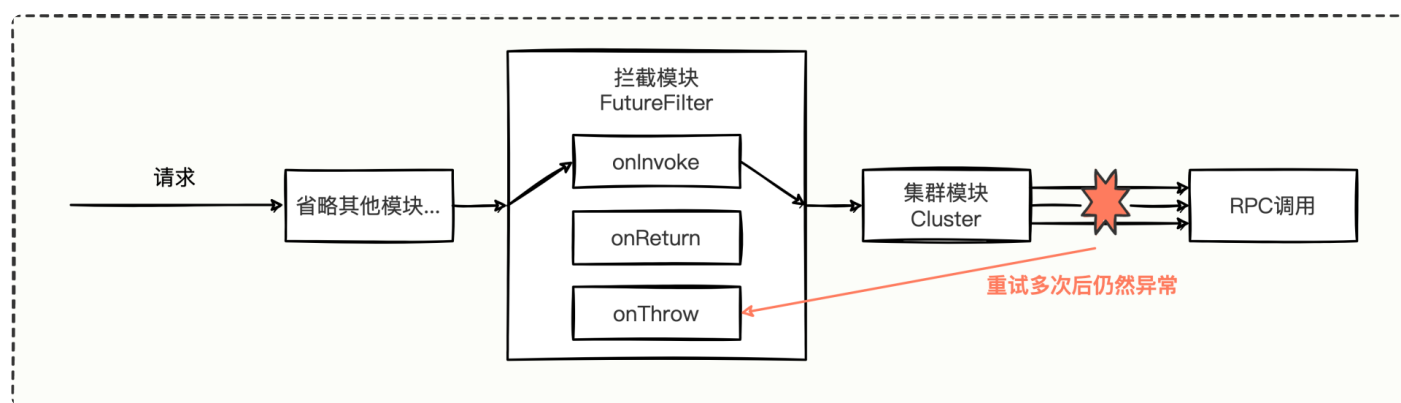
```

```
17 sayHello:-1, DemoFacadeDubboProxy1 (com.hmilyylimh.cloud.facade.demo)
18 invoke0:-1, NativeMethodAccessorImpl (jdk.internal.reflect)
19 invoke:62, NativeMethodAccessorImpl (jdk.internal.reflect)
20 invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)
21 invoke:567, Method (java.lang.reflect)
22 invokeJoinpointUsingReflection:344, AopUtils (org.springframework.aop.support)
23 invoke:208, JdkDynamicAopProxy (org.springframework.aop.framework)
24 sayHello:-1, $Proxy56 (com.sun.proxy)
25 hello:39, InvokeDemoFacade (com.hmilyylimh.cloud.anno.demo)
```



图中代码就是刚刚 Debug 时到达了 `FailoverClusterInvoker.doInvoke` 方法的断点这里，我们顺着调用栈往下看，你会发现 `FutureFilter` 是在 `FailoverClusterInvoker` 之前调用的，不难得出 `FailoverClusterInvoker` 即使内部尝试了多次失败后抛出的异常，最后会被 `FutureFilter` 捕获到，这就能解释为什么重试机制执行多次后，事件通知机制只执行一次。

用一张图来描述上述 Debug 的调用流程：



请求会经历一些其他的模块，但始终会经过 `FutureFilter` 过滤器，过滤器内部先触发调用 `onInvoke` 方法，然后紧接着触发集群模块进行 RPC 调用。

因为有着 `timeout = 1` 这个 1 毫秒的超时设置，所以几乎是请求还没发出去就超时了，因此集群模块重试了 3 次后，会继续调用 `FutureFilter` 的 `onThrow` 方法，而 `onThrow` 在 `FutureFilter` 源码中的流程就会进行反射调用 Dubbo 接口 `sayHello` 设置的事件回调配置。

到解释了为什么 Dubbo 框架的事件通知机制不会因为有多重试机制而触发多次。若对 `FutureFilter` 是怎么反射调用的操作感兴趣，你可以继续深入研究一番。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 事件通知：一招打败各种神乎其神的回调事件

下一篇 08 | 缓存操作：如何为接口优雅地提供缓存功能？

## 精选留言 (1)

写留言



Six Days

2023-01-29 来自广东

思考题：Spring 的切面中完成对参数的统一验证？

根据本文中作者着重梳理通用验证思路，可以在项目中自定义注解定义校验参数的标准，可通过自定义注解标记AOP拦截的切入点，识别拦截点之后，可在方法调用前进行自定义的参数校验逻辑；在项目内可通过自定义注解实现参数的统一验证，有利于代码的维护

作者回复：你好，Six Days：你理解的非常透彻，有了思路后只不过就是换了一个地方(Spring)进行参数验证而已，点赞～

