

10 | 服务认证：被异构系统侵入调用了，怎么办？

2023-01-09 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

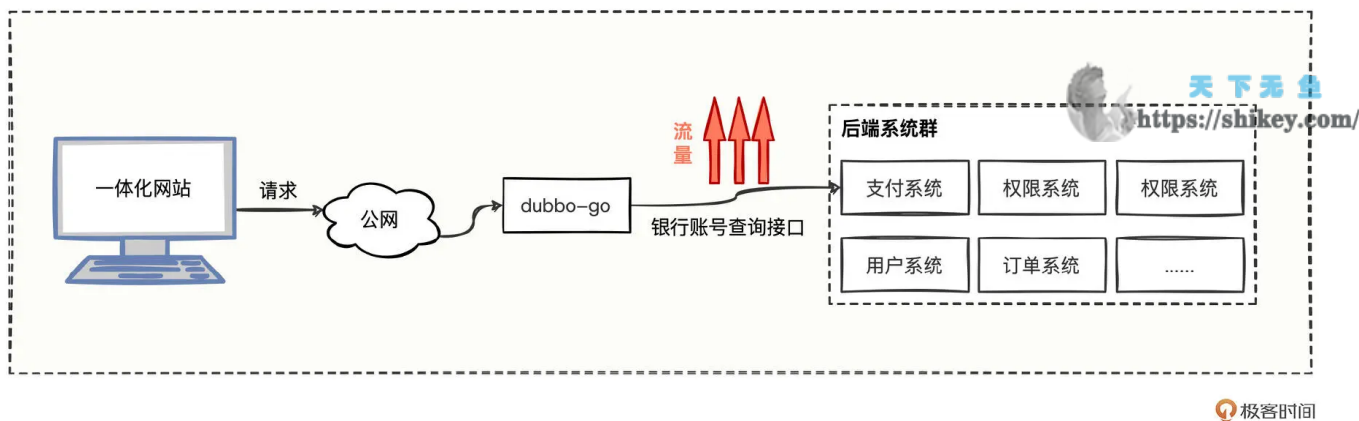
时长 15:02 大小 13.72M



你好，我是何辉。今天我们探索 Dubbo 框架的第九道特色风味，服务认证。

通过集成 Java 语言编写的 Dubbo 框架来提供服务，你已经非常熟悉了，作为 Dubbo 多语言生态最火热的项目，用 Go 语言开发的 dubbo-go 框架，想必你也有所耳闻，然而，就是这样一款非常实用且轻量级的优秀框架，却引发了一些产线事件。

事情是这样的，公司最近要做一个关于提升效能的一体化网站，我们的后端服务全是 Dubbo 提供者，但是负责效能开发的同事只会使用 Go 或 Python 来编写代码，于是经过再三考虑，效能同事最后使用 dubbo-go 来过渡对接后端的 Dubbo 服务。就像这样：



然而，dubbo-go 服务上线后不久，某个时刻，支付系统的银行账号查询接口的 QPS 异常突增，引起了相关领导的关注。

一番排查后，我们发现银行账号查询接口的来源 IP 格式比较怪异，找网工帮忙分析了一下，怪异的 IP 是一个异构系统 dubbo-go 服务发出来的请求（至于一体化网站为什么需要查询该接口就是后话了）。

目前暴露了一个比较严重的问题，被异构系统访问的接口缺乏一种认证机制，尤其是安全性比较敏感的业务接口，随随便便就被异构系统通过非正常途径调通了，有不少安全隐患。因此很有必要添加一种服务与服务之间的认证机制。

对于这个添加服务认证的需求，你会如何处理呢？

认证什么？

服务认证是一个很空泛的概念，很多人会疑惑服务认证，到底是在认证什么呢？

我们联想生活中认证含义，你可以从网络搜索到各式各样的解释，有人说，认证是由认证机构进行的一种**合格评定**活动，也有人说，认证是一种**信用保证**形式，还有人说，认证是指由认证机构证明产品、服务、管理体系**符合**相关技术**规范**的强制性要求或者**标准**的合格评定活动。

众说纷纭，但都没有错，不同的领域对于“认证”概念自然有着不同的内涵。我们简单分析一下这三种说法，关键在于“合格”、“信用保证”、“标准”，本质上有一种谁和谁作比较的概念。

既然有比较，就得弄清楚具体要比较的内容，而内容真假好坏的鉴定也需要一套规则。那我们就可以得出所谓的认证，就是用规则来鉴定内容。

对于服务之间的认证而言，**比较的内容是什么？规则又是什么呢？**

比较的内容其实好说，无非就是客户端发送给服务端的数据。那我们要利用规则对数据进行怎样的鉴定呢？鉴定数据的真假？还是数据是否被篡改过？还是其他的什么呢？

参考一般的业务需求，目前我们也没有想到更多的鉴别规则了，那就先按照鉴定真假和鉴定篡改两个规则分析。

1. 鉴定真假

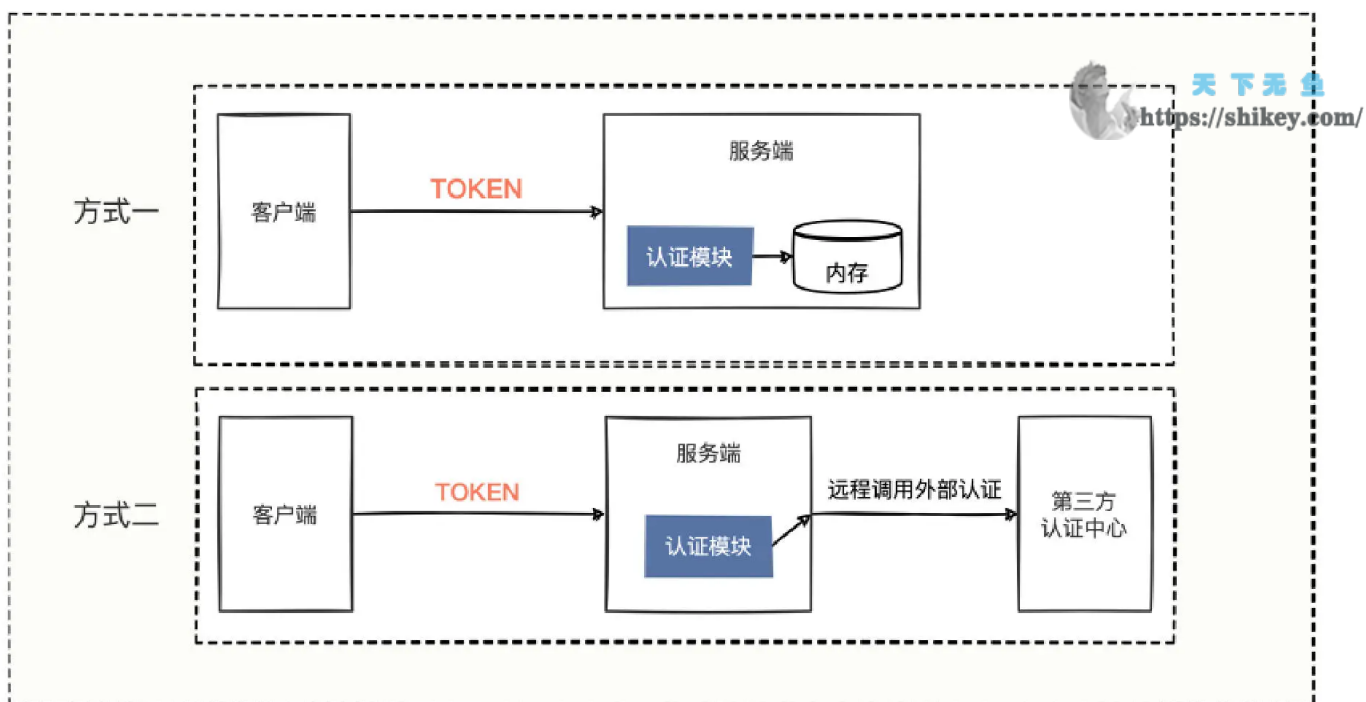
具体如何鉴别真假呢？我们联想业界已有的方案，比如对接微信支付，在向微信发送请求数据的时候，需要填写了 `appId` 和 `secret` 两个重要的字段，然后微信支付后台会比对这两个字段的值是否存在。

那我们是不是也可以仿照微信支付请求，也在数据里安插几个重要字段来鉴别呢？

应该可行，**在客户端发送数据的时候我们添加一个 TOKEN 字段**，然后，服务端收到数据先验证 `TOKEN` 字段值是否存在，若存在则认为是合法可信任的请求，否则就可以抛出异常中断请求了。

这个 `TOKEN` 在服务端怎么验证是否存在呢？

肯定难不倒你，一般两种处理方式，要么服务端内部就有这个 `TOKEN` 直接验证，要么服务端去第三方媒介间接验证，总之处理请求的是服务端，至于服务端是内部验证还是依赖第三方媒介验证，那都是服务端的事情。顺着思路画出了这样的调用链路图：



极客时间

处理方式一是服务端收到 **TOKEN** 凭证后与自身内存的值做比对验证，方式二是收到 **TOKEN** 后调用第三方认证中心进行比对验证。区别就在于，前者无需调用远程服务可以直接验证，而后者需要调用远程才能进行验证。

不过，我们要面临分支选择了，该使用哪种方式呢？

- 方式一无需调用远程，虽然节省了远程调用的开销，加快了处理验证的时效，但是这个 **TOKEN** 是位于服务端内部的，那就意味着 **TOKEN** 和服务端的方法有着一定的强绑定关系，不够灵活。
- 方式二远程调用，第三方认证中心可以根据不同的客户端分配不同的 **TOKEN** 值，并为 **TOKEN** 设置过期时间，灵活性和可控性变强了，但同时也牺牲了一定的远程调用时间开销。

所以，想通过单独认证中心进行统一约束和管理，且可以容忍远程调用的少许耗时，可以考虑方式二。如果只是想简单处理，或不能容忍性能的少许耗时，可以考虑方式一。

这里为了方便演示，我们就把方式一落实到代码，你知道该怎么做了吧？

还是先梳理改造的思路：

1. 客户端在发送请求数据时，需要额外添加一个 TOKEN 字段，参考“[隐式传递](https://shikey.com/)”，我们可以把 TOKEN 放在 Invocation 的 attachments 里面。
2. 服务端在处理请求的认证逻辑时，为了不侵入业务逻辑，可以在过滤器里面处理，
3. 过滤器需要优先处理 TOKEN 字段值是否存在，如果存在则继续后面的业务逻辑处理，否则就直接抛出异常。

接下来，编写代码，每一行我都详细写了注释，你可以对照着看：

复制代码

```
1 ///////////////////////////////////////////////////
2 // 提供方：自定义TOKEN校验过滤器，主要对 TOKEN 进行验证比对
3 ///////////////////////////////////////////////////
4 @Activate(group = PROVIDER)
5 public class ProviderTokenFilter implements Filter {
6     /** <h2>TOKEN 字段名</h2> */
7     public static final String TOKEN = "TOKEN";
8     /** <h2>方法级别层面获取配置的 auth.enable 参数名</h2> */
9     public static final String KEY_AUTH_ENABLE = "auth.enable";
10    /** <h2>方法级别层面获取配置的 auth.token 参数名</h2> */
11    public static final String KEY_AUTH_TOKEN = "auth.token";
12    @Override
13    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
14        // 从方法层面获取 auth.enable 参数值
15        String authEnable = invoker.getUrl().getMethodParameter
16            (invocation.getMethodName(), KEY_AUTH_ENABLE);
17        // 如果不需要开启 TOKEN 认证的话，则继续后面过滤器的调用
18        if (!Boolean.TRUE.toString().equals(authEnable)) {
19            return invoker.invoke(invocation);
20        }
21        // 能来到这里，说明需要进行 TOKEN 认证
22        Map<String, Object> attachments = invocation.getObjectAttachments();
23        String recvToken = attachments != null ? (String) attachments.get(TOKEN)
24        // 既然需要认证，如果收到的 TOKEN 为空，则直接抛异常
25        if (StringUtils.isBlank(recvToken)) {
26            throw new RuntimeException(
27                "Recv token is null or empty, path: " +
28                String.join(".", invoker.getInterface().getName(), invocat
29        }
30        // 从方法层面获取 auth.token 参数值
31        String authToken = invoker.getUrl().getMethodParameter
32            (invocation.getMethodName(), KEY_AUTH_TOKEN);
33        // 既然需要认证，如果收到的 TOKEN 值和提供方配置的 TOKEN 值不一致的话，也直接抛异常
34        if (!recvToken.equals(authToken)) {
35            throw new RuntimeException(
36                "Recv token is invalid, path: " +
37                String.join(".", invoker.getInterface().getName(), invocat
38        }
```

```

39 // 还能来到这，说明认证通过，继续后面过滤器的调用
40 return invoker.invoke(invocation);
41 }
42 }
43
44 ///////////////////////////////////////////////////
45 // 提供方：支付账号查询方法的实现逻辑
46 // 关注 auth.token、auth.enable 两个新增的参数
47 ///////////////////////////////////////////////////
48 @DubboService(methods = {@Method(
49     name = "queryPayAccount",
50     parameters = {
51         "auth.token", "123456789",
52         "auth.enable", "true"
53     })})
54 )
55 @Component
56 public class PayAccountFacadeImpl implements PayAccountFacade {
57     @Override
58     public String queryPayAccount(String userId) {
59         String result = String.format(now() + ": Hello %s, 已查询该用户的【银行账号】",
60             System.out.println(result);
61         return result;
62     }
63     private static String now() {
64         return new SimpleDateFormat("yyyy-MM-dd_HH:mm:ss.SSS").format(new Date(
65         )
66     }
67
68 ///////////////////////////////////////////////////
69 // 消费方：自定义TOKEN校验过滤器，主要将 TOKEN 传给提供方
70 ///////////////////////////////////////////////////
71 @Activate(group = CONSUMER)
72 public class ConsumerTokenFilter implements Filter {
73     /** <h2>方法级别层面获取配置的 auth.token 参数名</h2> **/
74     public static final String KEY_AUTH_TOKEN = "auth.token";
75     /** <h2>TOKEN 字段名</h2> **/
76     public static final String TOKEN = "TOKEN";
77     @Override
78     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
79         // 从方法层面获取 auth.token 参数值
80         String authToken = invoker.getUrl().getMethodParameter
81             (invocation.getMethodName(), KEY_AUTH_TOKEN);
82         // authToken 不为空的话则设置到请求对象中
83         if (StringUtils.isNotBlank(authToken)) {
84             invocation.getObjectAttachments().put(TOKEN, authToken);
85         }
86         // 继续后面过滤器的调用
87         return invoker.invoke(invocation);
88     }
89 }
90

```



天下无鱼

<https://shikey.com/>

```
91 ///////////////////////////////////////////////////  
92 // 消费方：触发调用支付账号查询接口的类  
93 // 关注 auth.token 这个新增的参数  
94 ///////////////////////////////////////////////////  
95 @Component  
96 public class InvokeAuthFacade {  
97     // 引用下游支付账号查询的接口  
98     @DubboReference(timeout = 10000, methods = {@Method(  
99         name = "queryPayAccount",  
100         parameters = {  
101             "auth.token", "123456789"  
102         })})  
103     private PayAccountFacade payAccountFacade;  
104  
105     // 该方法主要用来触发调用下游支付账号查询方法  
106     public void invokeAuth(){  
107         String respMsg = payAccountFacade.queryPayAccount("Geek");  
108         System.out.println(respMsg);  
109     }  
110 }
```

代码写后，我们验证一下，想办法触发调用消费方的 `invokeAuth` 方法，打印结果长这样：

[复制代码](#)

```
1 2022-11-22_23:51:07.899: Hello Geek, 已查询该用户的【银行账号信息】
```

调用没有抛出异常，说明认证通过，而且正常拿到提供方的结果了。

接下来把消费方 `queryPayAccount` 方法的 `auth.token` 的值修改一下看是否报错，这里我就随便改成 `123` 了，再尝试调用一下 `invokeAuth` 方法，打印如下：

[复制代码](#)

```
1 Caused by: org.apache.dubbo.remoting.RemotingException: java.lang.RuntimeExcept  
2 java.lang.RuntimeException: Recv token is invalid, path: com.hmilyylimh.cloud.f  
3 at com.hmilyylimh.cloud.auth.config.ProviderTokenFilter.invoke(ProviderTokenF  
4 at org.apache.dubbo.rpc.cluster.filter.FilterChainBuilder$CopyOfFilterChainNc  
5 at org.apache.dubbo.monitor.support.MonitorFilter.invoke(MonitorFilter.java:9
```

把 `auth.token` 的值乱改一通后，发现接口调用不通了，因为打印的异常日志中已经明确提示 `TOKEN` 是无效的，说明 `TOKEN` 的正常流程和异常流程都是符合预期的。

改造完成！我们简单回看一下代码的主要改动点。

代码的实现主要有消费方和提供方两部分，提供方的代码主要有 4 点改动：



1. 在提供方定义一个 `ProviderTokenFilter` 过滤器类，然后实现 `invoke` 方法。
2. 在 `invoke` 方法中，从 `invocation` 的 `attachments` 中获取 `TOKEN` 值，再从上下文中获取方法层面配置的 `TOKEN` 值，最后直接比较两个 `TOKEN` 值是否一样，一样就正常往下执行，不一样则抛出异常直接中断调用流程。
3. 在银行账号查询接口所在服务的 `@DubboService` 注解中，为该方法添加一个 `TOKEN` 参数（`auth.token`）以及对应的值（`123456789`）。
4. 将 `ProviderTokenFilter` 的类路径添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.rpc.Filter` 文件中。

消费方的代码也有 4 点改动：

1. 在消费方定义一个 `ConsumerTokenFilter` 过滤器类，然后也实现 `invoke` 方法。
2. 在调用银行账号查询接口所在服务的 `@DubboReference` 注解中，为该方法添加一个 `TOKEN` 参数（`auth.token = 123456789`）以及声明该方法需要开始 `TOKEN` 认证（`auth.enable = true`）。
3. 在 `invoke` 方法中，从上下文中获取方法层面配置的 `TOKEN` 值，然后放进 `invocation` 的 `attachments` 中，跟随远程调用去往提供方。
4. 将 `ConsumerTokenFilter` 的类路径添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.rpc.Filter` 文件中。

按照不发起远程调用的形式把代码实现后，是不是也并没有你想象中的那么难，关键就是找到突破口，写代码只是顺带的事情。在这个基础之上，再来扩展改造为调用远程的方式进行认证，想必难不倒你了，记得课后挑战一下。

2. 鉴定篡改

我们接着看第二个认证工作，鉴定篡改，这个比较好理解，就是证明别人发送过来的内容没有在传输过程中被偷偷改动。

对加密算法有一定了解的想必也马上想到了，证明数据是否被改动，可以考虑加密或加签。

加签是为了校验数据在传输过程中是否被修改，而**加密**其实就是把明文变成密文，保护数据在传输过程中信息不泄密。



那为了提升安全敏感度，我既想保护数据的隐私，又想保护数据不被篡改，那是不是可以将加密和加签都派上用场呢？

这个当然可以，为了安全因素的考量，有时候必须得在性能损耗上做出一定让步。但是也不能让步的太出格。像公司内部系统就会舍弃加密的处理，毕竟安全等级越高，加密出来的密文体积就会越大，在传输过程中会大大增加带宽的消耗。

所以**有时候舍弃加密处理是一种折中考量，采用加签的方式基本上就足够了。**

这里我们就以一套常用的 **RSA** 加签方式进行演示。第一步当然还是梳理代码修改思路：

1. 客户端新增一个 **ConsumerAddSignFilter** 加签过滤器，同样服务端也得增加一个 **ProviderVerifySignFilter** 验签过滤器。
2. 客户端将加签的结果放在 **Invocation** 的 **attachments** 里面。
3. 服务端获取加签数据时，进行验签处理，若验签通过则放行，否则直接抛出异常。

大致思路梳理完后，我们开始改造，同样地，可以参考我写的详细注释看：

复制代码

```
1  //////////////////////////////////////
2  // 提供方：自定义验签过滤器，主要对 SIGN 进行验签
3  //////////////////////////////////////
4  @Activate(group = PROVIDER)
5  public class ProviderVerifySignFilter implements Filter {
6      /** <h2>SING 字段名</h2> */
7      public static final String SING = "SING";
8      /** <h2>方法级别层面获取配置的 auth.ras.enable 参数名</h2> */
9      public static final String KEY_AUTH_RSA_ENABLE = "auth.rsa.enable";
10     /** <h2>方法级别层面获取配置的 auth.rsa.public.secret 参数名</h2> */
11     public static final String KEY_AUTH_RSA_PUBLIC_SECRET = "auth.rsa.public.se
12     @Override
13     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
14         // 从方法层面获取 auth.ras.enable 参数值
15         String authRsaEnable = invoker.getUrl().getMethodParameter
16             (invocation.getMethodName(), KEY_AUTH_RSA_ENABLE);
17         // 如果不需要验签的话，则继续后面过滤器的调用
```



```
18     if (!Boolean.TRUE.toString().equals(authRsaEnable)) {
19         return invoker.invoke(invocation);
20     }
21
22     // 能来到这里，说明需要进行验签
23     Map<String, Object> attachments = invocation.getObjectAttachments();
24     String recvSign = attachments != null ? (String) attachments.get(SING)
25     // 既然需要认证，如果收到的加签值为空的话，则直接抛异常
26     if (StringUtils.isBlank(recvSign)) {
27         throw new RuntimeException(
28             "Recv sign is null or empty, path: " +
29             String.join(".", invoker.getInterface().getName(), invocat
30     }
31
32     // 从方法层面获取 auth.rsa.public.secret 参数值
33     String rsaPublicSecretOpsKey = invoker.getUrl().getMethodParameter
34         (invocation.getMethodName(), KEY_AUTH_RSA_PUBLIC_SECRET);
35     // 从 OPS 配置中心里面获取到 rsaPublicSecretOpsKey 对应的密钥值
36     String publicKey = OpsUtils.get(rsaPublicSecretOpsKey);
37     // 加签处理
38     boolean passed = SignUtils.verifySign(invocation.getArguments(), public
39     // sign 不为空的话则设置到请求对象中
40     if (!passed) {
41         throw new RuntimeException(
42             "Recv sign is invalid, path: " +
43             String.join(".", invoker.getInterface().getName(), invocat
44     }
45
46     // 继续后面过滤器的调用
47     return invoker.invoke(invocation);
48 }
49 }
50
51 ///////////////////////////////////////////////////
52 // 提供方：支付账号查询方法的实现逻辑
53 // 关注 auth.rsa.public.secret、auth.rsa.enable 两个新增的参数
54 ///////////////////////////////////////////////////
55 @DubboService(methods = {@Method(
56     name = "queryPayAccount",
57     parameters = {
58         "auth.rsa.public.secret", "queryPayAccoun_publicSecret",
59         "auth.rsa.enable", "true"
60     })})
61 )
62 @Component
63 public class PayAccountFacadeImpl implements PayAccountFacade {
64     @Override
65     public String queryPayAccount(String userId) {
66         String result = String.format(now() + ": Hello %s, 已查询该用户的【银行账号】
67         System.out.println(result);
68         return result;
69     }
}
```

```
70     private static String now() {
71         return new SimpleDateFormat("yyyy-MM-dd_HH:mm:ss.SSS").format(new Date(
72     })
73 }
74
75 ///////////////////////////////////////////////////
76 // 消费方：自定义加签过滤器，主要将 SIGN 传给提供方
77 ///////////////////////////////////////////////////
78 @Activate(group = CONSUMER)
79 public class ConsumerAddSignFilter implements Filter {
80     /** <h2>SIGN 字段名</h2> */
81     public static final String SING = "SING";
82     /** <h2>方法级别层面获取配置的 auth.rsa.private.secret 参数名</h2> */
83     public static final String KEY_AUTH_RSA_PRIVATE_SECRET = "auth.rsa.private.
84 @Override
85     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE
86         // 从方法层面获取 auth.token 参数值
87         String aesSecretOpsKey = invoker.getUrl().getMethodParameter
88             (invocation.getMethodName(), KEY_AUTH_RSA_PRIVATE_SECRET);
89         // 从 OPS 配置中心里面获取到 aesSecretOpsKey 对应的密钥值
90         String privateKey = OpsUtils.get(aesSecretOpsKey);
91         // 加签处理
92         String sign = SignUtils.addSign(invocation.getArguments(), privateKey);
93         // sign 不为空的话则设置到请求对象中
94         if (StringUtils.isNotBlank(sign)) {
95             invocation.getObjectAttachments().put(SING, sign);
96         }
97         // 继续后面过滤器的调用
98         return invoker.invoke(invocation);
99     }
100 }
101
102 ///////////////////////////////////////////////////
103 // 消费方：触发调用支付账号查询接口的类
104 // 关注 auth.rsa.private.secret 这个新增的参数
105 ///////////////////////////////////////////////////
106 @Component
107 public class InvokeAuthFacade {
108     // 引用下游支付账号查询的接口
109     @DubboReference(timeout = 10000, methods = {@Method(
110         name = "queryPayAccount",
111         parameters = {
112             "auth.rsa.private.secret", "queryPayAccoun_privateSecret"
113         })})
114     private PayAccountFacade payAccountFacade;
115
116     // 该方法主要用来触发调用下游支付账号查询方法
117     public void invokeAuth(){
118         String respMsg = payAccountFacade.queryPayAccount("Geek");
119         System.out.println(respMsg);
120     }
121 }
```

代码的实现还是有消费方和提供方两部分。提供方的代码主要有 4 点改动：

1. 在提供方定义一个 `ProviderVerifySignFilter` 验签过滤器类，然后实现 `invoke` 方法。
2. 在 `invoke` 方法中，从 `invocation` 的 `attachments` 中获取 `SIGN` 加签值，再从上下文中获取方法层面配置的 `RSA` 公钥 `OPS` 配置值（`auth.rsa.public.secret`），最后进行验签，不一样则抛出异常直接中断调用流程。
3. 在银行账号查询接口所在服务的 `@DubboService` 注解中，为该方法添加一个 `RSA` 公钥的 `OPS` 配置（`auth.rsa.public.secret`）以及声明该方法需要开启 `RSA` 验签（`auth.rsa.enable = true`）。
4. 将 `ProviderVerifySignFilter` 的类路径添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.rpc.Filter` 文件中。

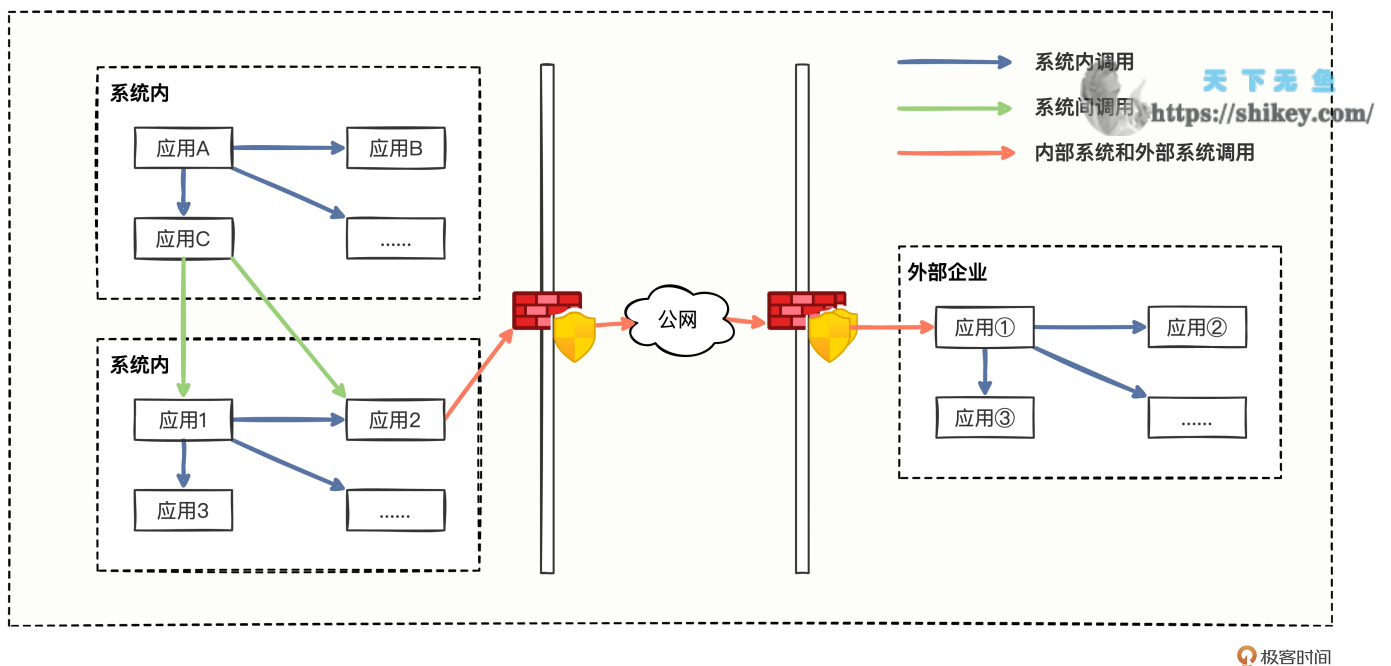
消费方的代码主要有 4 点改动：

1. 在消费方定义一个 `ConsumerAddSignFilter` 加签过滤器类，然后也实现 `invoke` 方法。
2. 在调用银行账号查询接口所在服务的 `@DubboReference` 注解中，为该方法添加一个 `RSA` 私钥的 `OPS` 配置（`auth.rsa.private.secret`）。
3. 在 `invoke` 方法中，将 `invocation` 中 `arguments` 进行加签操作，然后放进 `invocation` 的 `attachments` 中，跟随远程调用去往提供方。
4. 将 `ConsumerAddSignFilter` 的类路径添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.rpc.Filter` 文件中。

到这里，鉴定真假、鉴定篡改这两种方式如何实现我们就学完了，细心的你想必应该也发现了，其实鉴定真假和鉴定篡改也是可以一起使用的，无非就是看安全度需要提升到什么粒度而已。

服务认证的应用

以后看到异构系统，相信你应该知道怎么轻量级地进行服务认证了。一般来说，凡是存在交互的地方，就会存在信任与不信任，那在实际开发的过程中，还有哪些应用场景需要进行服务认证呢？从一张系统拓扑图中，我们来详细分析一下：



从图中可以总结有三种调用场景。

第一，系统内应用之间的调用，这种基本上是一个系统群或者系统域，里面有很多应用，应用之间的调用可以考虑服务认证。

第二，系统间不同应用的调用，这种有点像不同部门或不同系统域之间的应用调用，有点跨组织的概念，往往会考虑服务认证。

第三，系统内外不同系统的调用，这种有点像内部系统与外部公网之外的系统进行调用，一旦谈到公网，各种安全因素一定少不了，因此就更有必要考虑认证了。

总结

今天，从一个异构系统调用支付系统引发了 QPS 暴增的问题开始，我们意识到内部系统与异构系统之间缺乏一种安全的认证机制，分析了服务如何进行认证：

- 鉴定真假，通过安插一些特殊字段（比如 **TOKEN** 字段）放在请求数据中，然后服务端识别这些特殊字段值进行一定的比对验证。
- 鉴定篡改，通过对数据进行加签、加密，将加签结果、加密结果放在请求数据中，然后服务端展开验签、解密操作，以确保数据的原始完整性。

这里也总结下自定义认证过滤器的通用三部曲：

- 首先，自定义过滤器继承 `org.apache.dubbo.rpc.Filter` 接口，并实现 `invoke` 方法，同时在过滤器的 `@Activate` 注解中声明是提供方或消费方，还是两者都是。
- 其次，可以在 `@Method` 的 `parameters` 字段中为方法自定义一些属性，以辅助过滤器的通用逻辑处理。
- 最后，将自定义过滤器的类路径一定要记得添加到 `META-INF` 文件夹下面的 `org.apache.dubbo.rpc.Filter` 文件中。

服务认证的应用场景有三类，系统内应用之间的调用、系统间不同应用的调用、系统内外不同系统的调用。

思考题

留个作业给你，Dubbo 框架也有类似服务认证这样的支撑能力，你可以研究下 `TokenFilter`、`ConsumerSignFilter`、`ProviderAuthFilter`，看看如何应用。

欢迎在留言区分享你的思考和学习心得。我们下节课再见。

09 思考题参考

上一期留了两个作业：

- 如何在 `CustomCacheFilter` 基础之上继续实现消费方的限流解决方案呢？
- 如何简单地应用 `ActiveLimitFilter`、`ExecuteLimitFilter` 分两个过滤器来控制流量？

问题一，其实在 `CustomCacheFilter` 底层，根本不区分到底是消费方还是提供方，底层只是识别了服务名和方法名就完成了一切了。因此只需要做两步：

将 `CustomCacheFilter` 的 `@Activate` 类注解修改下，既支持提供方，也支持消费方即可：

复制代码

```
1 @Activate(group = {PROVIDER, CONSUMER})
2 public class CustomLimitFilter implements Filter {
3     // 省略其他代码
4 }
```

当消费方发起调用的时候，在 `@DubboReference` 注解中同样添加 `qps.enable`、`qps.value`、`qps.type` 这三个参数即可：



复制代码

```
1 @DubboReference(timeout = 10000, methods = {@Method(  
2     name = "queryRoleList",  
3     parameters = {  
4         "qps.enable", "true",  
5         "qps.value", "3"  
6     })))  
7 private RoleQueryFacade roleQueryFacade;
```

问题二就更简单了，有了编写 `CustomCacheFilter` 的经验，再来看 `ActiveLimitFilter`、`ExecuteLimitFilter` 这两个类的实现逻辑，简直是秒懂。

打开 `ActiveLimitFilter` 看看：

复制代码

```
1 @Activate(group = CONSUMER, value = ACTIVES_KEY)  
2 public class ActiveLimitFilter implements Filter, Filter.Listener {  
3     private static final String ACTIVE_LIMIT_FILTER_START_TIME = "active_limit_  
4     @Override  
5     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcE  
6         URL url = invoker.getUrl();  
7         String methodName = invocation.getMethodName();  
8         // 参数一：设置 actives 属性  
9         int max = invoker.getUrl().getMethodParameter  
10             (methodName, "actives", 0);  
11         final RpcStatus rpcStatus = RpcStatus.getStatus(invoker.getUrl(), invoc  
12         if (!RpcStatus.beginCount(url, methodName, max)) {  
13             // 参数二：设置 timeout 属性  
14             long timeout = invoker.getUrl().getMethodParameter  
15                 (invocation.getMethodName(), "timeout", 0);  
16             // 省略了其他代码逻辑  
17             // 大致含义是，在有限的超时时间范围内会继续等待  
18             // 直到超时时间到了，还未拿到计数资源的话，那么就抛出异常  
19         }  
20         invocation.put(ACTIVE_LIMIT_FILTER_START_TIME, System.currentTimeMillis  
21         return invoker.invoke(invocation);  
22     }  
23     // 省略了其他代码逻辑  
24 }
```

阅读 `ActiveLimitFilter` 代码后，我们可以得出 3 个规则：



- `@Activate` 中指定的是 `CONSUMER`，说明只在消费方生效。
- 在方法级别层面，可以设置 `actives`、`timeout` 两个参数来进行消费方限流。
- 特别之处在于 `timeout` 参数，如果拿不到计数限流资源，仍未超时，会继续等待片刻，直到有计数资源释放后再次进行尝试获取计数资源，只要超时时间一到，还没有获取到计数资源的话，则抛出异常。

接下来再打开 `ExecuteLimitFilter` 看看：

复制代码

```
1 @Activate(group = CommonConstants.PROVIDER, value = EXECUTES_KEY)
2 public class ExecuteLimitFilter implements Filter, Filter.Listener {
3     private static final String EXECUTE_LIMIT_FILTER_START_TIME = "execute_limit_filter_start_time";
4     @Override
5     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
6         URL url = invoker.getUrl();
7         String methodName = invocation.getMethodName();
8         // 参数一：设置 executes 属性
9         int max = url.getMethodParameter(methodName, "executes", 0);
10        // 若获取不到计数资源的话，则抛出异常
11        if (!RpcStatus.beginCount(url, methodName, max)) {
12            throw new RpcException(RpcException.LIMIT_EXCEEDED_EXCEPTION,
13                "Failed to invoke method " + invocation.getMethodName() + " on " +
14                url + ", cause: The service using threads greater than " +
15                max + " is limited.");
16        }
17        invocation.put(EXECUTE_LIMIT_FILTER_START_TIME, System.currentTimeMillis());
18        try {
19            // 能来到这里，说明已获取到计数资源
20            return invoker.invoke(invocation);
21        } catch (Throwable t) {
22            if (t instanceof RuntimeException) {
23                throw (RuntimeException) t;
24            } else {
25                throw new RpcException("unexpected exception when ExecuteLimitFilter invoke");
26            }
27        }
28    }
29    // 省略了其他代码逻辑
30 }
```


阅读 `ExecuteLimitFilter` 代码后，同样可以以下 2 个规则：

- 规则一，`@Activate` 中指定的是 `PROVIDER`，说明只在提供方生效。
- 规则二，在方法级别层面，可以设置 `executes` 两个参数来进行提供方限流。



阅读 `ActiveLimitFilter`、`ExecuteLimitFilter` 两个源码类，细心的你可能会发现，消费方比提供方多了一个时间等待的操作，采用了等待的操作进行最大力度的获取计数资源，来充分利用 `timeout` 超时属性提高接口调用的成功概率。

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 4  提建议

© 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 09 | 流量控制：控制接口调用请求流量的三个秘诀

下一篇 11 | 配置加载顺序：为什么你设置的超时时间不生效？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。