

05 | 点点直连：点对点搭建产线“后门”的万能管控

2022-12-28 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

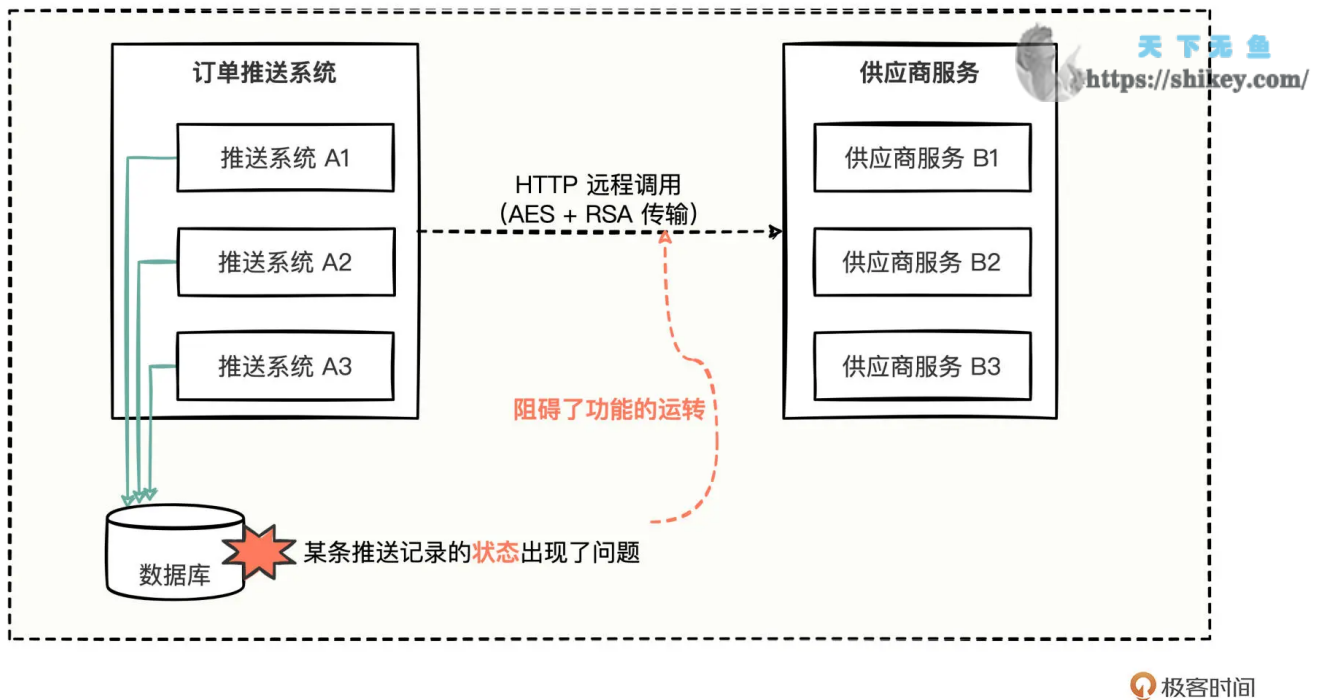
时长 17:25 大小 15.91M



你好，我是何辉。我们继续探索 Dubbo 框架的第四道特色风味，点点直连。

产线问题，一听到这个词，你是不是有一种莫名的紧张和敬畏感，没错，我们今天就来上点强度，聊一聊产线问题如何快速修复的话题。

情况是这样的，一天，运行良好的订单推送系统突然发生了一点异常情况，经过排查后，你发现有一条记录的状态不对，导致订单迟迟不能推送给外部供应商。订单推送系统的相关调用链路是这样的：



供应商系统都是集群部署的，只不过在订单推送系统这边的数据库中，有一条推送记录的状态不对，导致了这笔订单的最新信息无法推送出去，也就阻碍了该笔订单在供应商侧的功能运转。

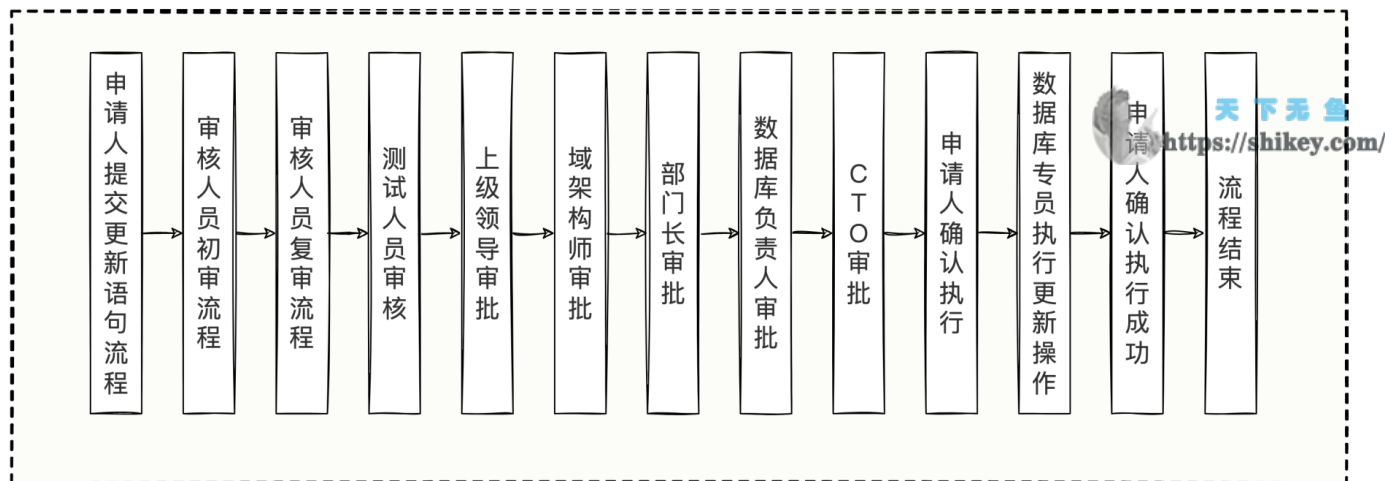
为了争取在最短时间内恢复这笔订单的功能运转，我们需要尽快修改这条推送记录在数据库的状态，修复产线数据。对于这样的紧急情况，你会怎么做？

修复数据，每个公司的流程规范都不一样，有时候得按照正规流程来，有时候得简单粗暴，在稳的情况下怎么快怎么来，时刻以解决用户的紧急诉求为准则。

1. 正规流程

参考公司平时遇到需要修复数据的情景，你也许会说，这有何难，找到那行记录，编写一个 Update 语句，然后提交一个数据订正的流程。

可是，你有想过一个公司有点规范性的数据订正流程有多长么？我们看一看公司的数据订正流程环节：



不看不知道，一看吓一跳，数据订正流程长达 12 个环节，需要集齐各路人马一一审批，光走流程，怎么着也得半个小时吧。

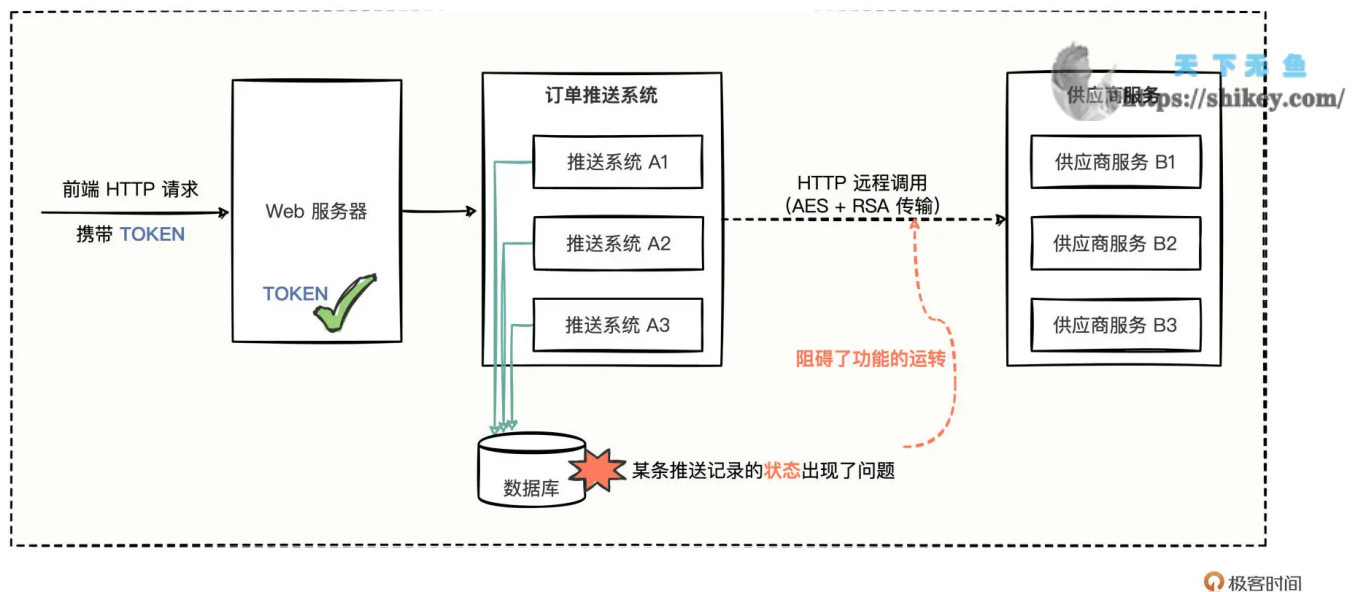
平常如果不是很紧急，可以慢慢走流程。可是现在偏偏赶上紧急关头，走正规的数据订正流程的确太慢，这个时候有人就想点子了，正规流程不行，那走邮件审批呢，让各个相关负责人邮件审批一下，后补流程，这总行了吧？

但如果真这么干，我们这个小小的产线事件可能就变性质了。因为在公司中，走邮件审批加速流程推进，某种程度上其实已经上升到事故性质了，后面各种检讨与善后措施肯定少不了，再说我们这功能好像也没有那么大的迫切性。

邮件审批不太合适，但按照正规流程修复数据也不可行，可能花儿都谢了，流程还没走完。那怎么办，有什么简单粗暴的方式呢？

2. 粗暴流程

粗暴流程嘛，有是有，只不过有点麻烦，我们看流程图，从前端切入，重点标出了 Web 服务器的 TOKEN 概念：



可以从 Web 服务器的后台日志中，弄出用户的 TOKEN，然后找到可以更新这条推送记录的 URL 地址，最后模拟用户的请求，把这条推送记录更新掉就行了。

这倒确实是一条路，但是细想一下，如果后台没有暴露这样功能的 URL 地址呢？或者即使暴露了这样的 URL 地址，它背后的功能实现逻辑根本不是更新推送记录的，怎么办呢？唉，模拟用户操作不但麻烦，还不一定能搞定，不可行。

我们继续研究流程图，看看还可以从哪里撕开一道口子。前端刚分析了比较难操作，如果从后端操作呢？

后端有我们的订单推送系统，还有供应商的系统，挨个来看看。

订单推送系统

如果从订单推送系统下手，目的很明确，就是要更新数据库。平常更新数据库基本上有三种方案：

1. 调用系统对外暴露的接口，间接通过接口的业务逻辑来更新数据库。
2. 想办法拿到 Dao 层实例对象，通过 Dao 层来操作 XML 更新数据库。
3. 想办法拿到 DataSource 数据源，通过最原始的 execSQL 形式更新数据库。

前两种方案，有则用，无则放弃，因为不可能每次出现产线事件，代码中都有预先准备好的修复方法。而最后一种，我们需要利用最原始的 API 来更新数据库，有一定难度，毕竟修复产线数据，安全、稳定、可靠性因素不容忽视。这三种方案都不是那么乐观，我们再想别的招。

考虑到订单推送系统是 Dubbo 服务提供的系统，我们是否可以像操作 CURL 命令发起 HTTP 请求那样，来调用 Dubbo 接口呢？什么命令可以向 Dubbo 发起请求呢？

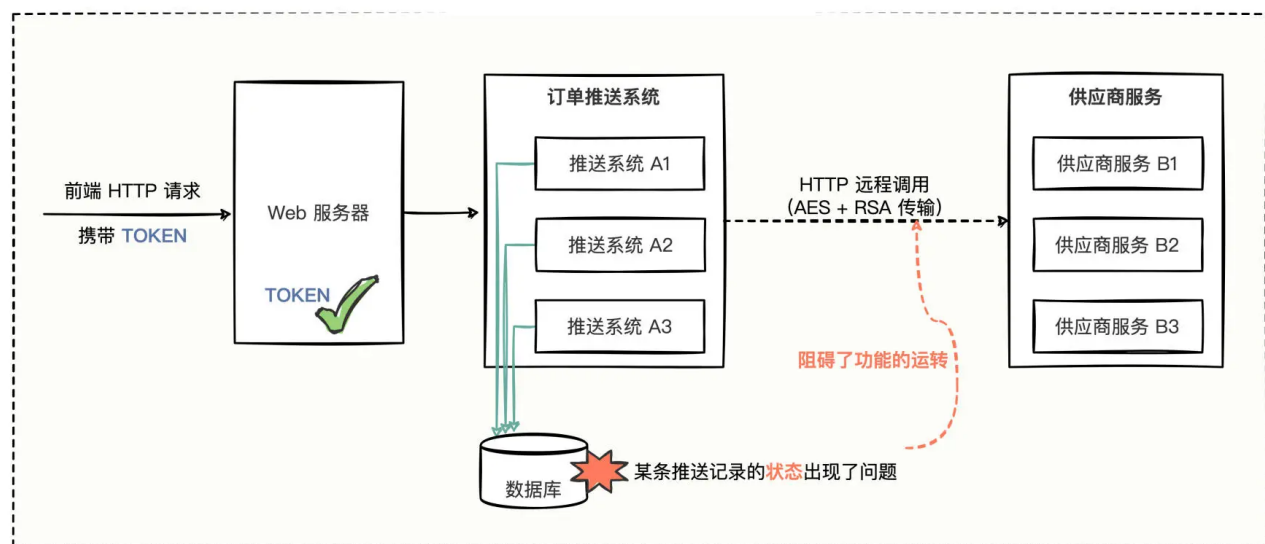
试着翻看 Dubbo 的参考手册，在 Dubbo2.7 的文档的 [文档 2.x -> 用户文档 -> 参考手册 -> Telnet 手册](#) 路径下，你会找到一个特别显眼的 telnet 关键字，发现里面居然可以执行 invoke 命令来调用 Dubbo 接口，这是重大发现。

但是这个 invoke 命令也只能调用 Dubbo 接口，假如我们恰好没有更新推送记录的 Dubbo 接口，岂不是白搭，而且这个命令不但权限大且操作麻烦，一般在产线上也会被运维禁用。

在订单推送系统中，我们尝试了所有的可能，都是徒劳而终。那从供应商服务考虑呢？

供应商服务

看通往供应商服务的 HTTP 调用接口：



想要修复数据，需要将报文 AES 加密，然后在 RSA 加签，最后将组装好的报文发往供应商，这一段复杂的操作之外，我们还得确认内网环境与供应商之间没有防火墙的阻挠，要能很轻松地拿到 AES 密钥、RSA 私钥，而且还得确认参数字段组装万无一失。

所以总的来看，粗暴流程，困难和解法层出不穷，但是行之有效且简单的却少之又少。

3. 万能管控



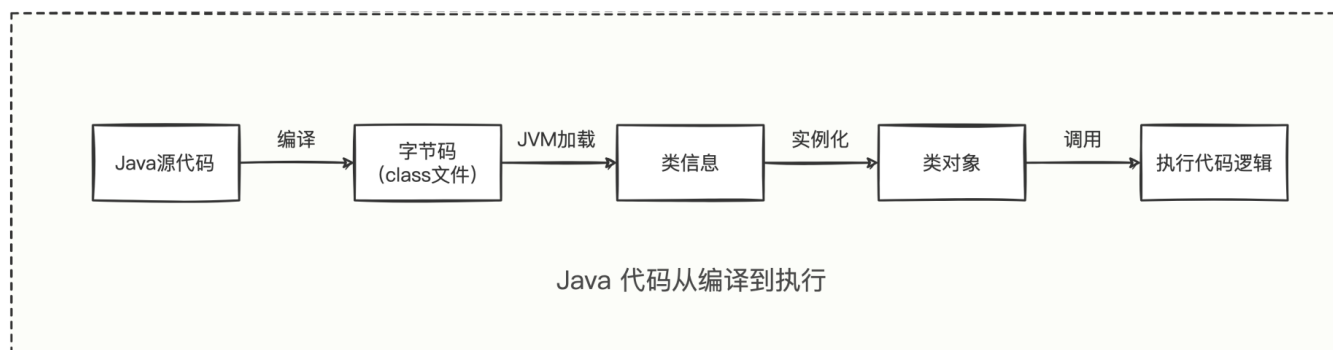
不过经过这些尝试，我们虽然没有解决数据修复的问题，却已经有了很多修复数据的点子，汇总一下：有没有一种万能管控措施，既能调用已有的 Dubbo 接口、Dao 层方法，又能操作 DataSource 数据源进行原始 SQL 操作，还能聚合调用各种方法，来达到期望的修复目的呢？

对于单一接口，我们可以用一个接口去想办法调用 Dubbo 接口，一个接口去调用系统各种 Dao 接口，一个接口去调用 DataSource 数据源，一共三个接口搞定。

但是，聚合调用各种方法，相当于三个接口的混合体，有无数种组合，我们不可能提前把各种组合的代码写好，然后等着出产线问题吧，这有点难。

但仔细想想，**既然提前写好的代码能被调用，但不可能把所有组合都写好，是不是可以考虑动态调用代码呢？**

好像可行。那如何动态编译呢？我们回忆 Java 代码从编译到执行的流程：




开发者编写的“Java 源代码”被编译后变成 class 字节码文件，然后字节码文件被 JVM 加载，直到变成可使用的类。在这样的开发过程中，动态编译一般有两种方式：

- 自主编码实现，比如通过 Runtime 调用 javac，或者通过 JavaCompile 调用 run。
- 调用插件实现，比如使用市面上常用的 groovy-all.jar 插件。

如果有时间有精力研究，可以考虑自主编码实现，但如果想在短时间比较有质量地完成任务，建议调用插件实现，可以少走很多弯路。

那接下来该如何发起调用呢？

参考前面的思路，可以直接在订单推送系统里定义一个 Dubbo 接口，采用 `invoke` 命令调用。 <https://shike.com/>
话音刚落，相信你也想到了第一个问题——**没有 `invoke` 命令执行权限**。

既然自己登录到服务器不能执行 `invoke` 命令，是不是可以绕一绕想办法让别人调用 Dubbo 接口？由别人调用，也得组装请求参数再调用，如果能把变化的参数做到页面上，别人从页面读取控件值然后发起调用，就更完美了。

设想还行，不失为一种非常灵活的方式。那第二个问题就来了——**产线的环境问题**，假如产线有 UAT1、UAT2、PRD1、PRD2、PRD3 多套环境，能随便把请求负载到产线任意环境中的任意一台机器么？

当然不行。因为产线分了很多环境，验证功能时，我们一般会拿某台机器节点做个小范围验证，否则调用到一台不该调用的节点，引发不必要的产线事件就太划不来了。

所以我们需要指定机器节点进行接口调用，那怎么指定具体的机器节点调用呢？

这不就像我们当初学习 `Socket` 和 `HTTP` 客户端编程一样，知道 `IP` 和 `PORT` 就可以与服务建立连接，通过 `IP` 可以定位到一台唯一的服务器，通过 `PORT` 可以在这台服务器定位到一个唯一的进程服务。

知道了 `IP` 和 `PORT`，可是进程服务有那么多 `Dubbo` 接口，要调用哪个接口呢？

同样地，我们类比 `Socket` 和 `HTTP`，`Socket` 编程中通过约定报文中的服务码来表示特定的功能，`HTTP` 编程中通过 `URL` 路径来表示特定的功能。`Dubbo`，也可以考虑采用 `Dubbo` 接口的“接口类名 + 接口方法名 + 接口方法入参类名”`Dubbo` 接口路径来表示特定的功能。

前 2 个难题都解决了，最后一个问题——**我们怎么通过 `IP`、`PORT`、`Dubbo` 接口路径，调用 `Dubbo` 接口呢？**

这个还真没谱，不过机智的你一定想到上一讲我们学过泛化调用，可以尝试从 `ReferenceConfig` 里找找，相信优秀的框架设计者应该能考虑到这种调用方式吧。

接下来就去 `ReferenceConfig` 类中看一看有没有类似设置 `IP`、`PORT`、`URL` 之类的字段：

```

1 public class ReferenceConfig<T> extends ReferenceConfigBase<T> {
2     // 省略了其他内容
3 }
4
5 public abstract class ReferenceConfigBase<T> extends AbstractReferenceConfig {
6     /**
7      * The url for peer-to-peer invocation
8      */
9     protected String url;
10
11     // 省略了其他内容
12 }

```



从源码 `ReferenceConfig` 中没有找到任何有用字段，接着向上找到父类 `ReferenceConfigBase`，发现了一个与地址有关的巧妙的 `url` 字段，英文注释还写到“该 `url` 是为点对点设计的”。

然而，现在问题又来了，从源码中找到了 `url` 字段，怎么才能知道 `url` 的构成规则呢？

这里我教你一个小技巧，当我们进入源码后，首先一定要认真看该字段的描述信息，接着要尝试检索该字段是如何被使用或被拆解使用的，然后在被使用的地方，时刻关注周围有没有一些描述信息。

按照小技巧的思路，果不其然，我们一路找到了答案，过程代码如下：

```

1 /**
2  * Parse the directly configured url.
3  * 解析直连的配置地址。
4  */
5 // org.apache.dubbo.config.ReferenceConfig#parseUrl
6 private void parseUrl(Map<String, String> referenceParameters) {
7     // 将配置的url地址按照分号进行切割，得到一个字符串数组
8     String[] us = SEMICOLON_SPLIT_PATTERN.split(url);
9     if (ArrayUtils.isEmpty(us)) {
10         // 然后循环url切出来的数组
11         for (String u : us) {
12             // 将切出来的每个元素传入URL的valueOf方法中，得到一个可被识别的对象
13             URL url = URL.valueOf(u);
14
15             // 省略了其他内容
16         }
17     }
18 }

```



```

17     }
18 }
19
20     ↓
21 /**
22  * parse decoded url string, formatted dubbo://host:port/path?param=value, into
23  * 解析解码后的url字符串, url的格式后的内容为: dubbo://host:port/path?param=value, int
24  *
25  * @param url, decoded url string
26  * @return
27  */
28 // org.apache.dubbo.common.URL#valueOf(java.lang.String)
29 public static URL valueOf(String url) {
30     // 紧接着继续调用URL中的另外一个valueOf方法
31     return valueOf(url, false);
32 }
33
34     ↓
35 /**
36  * parse normal or encoded url string into struttred URL:
37  * 将普通或编码过的url字符串变成可被支持识别的URL
38  * - dubbo://host:port/path?param=value
39  * - URL.encode("dubbo://host:port/path?param=value")
40  *
41  * @param url, url string
42  * @param encoded, encoded or decoded
43  * @return
44  */
45 // org.apache.dubbo.common.URL#valueOf(java.lang.String, boolean)
46 public static URL valueOf(String url, boolean encoded) {
47     if (encoded) {
48         return URLStrParser.parseEncodedStr(url);
49     }
50     // 接着又继续调用一个专门解析url规则的URLStrParser类
51     return URLStrParser.parseDecodedStr(url);
52 }
53
54     ↓
55 /**
56  * @param decodedURLStr : after {@link URL#decode} string
57  * decodedURLStr format: protocol://username:password@host
58  * [protocol://][username:password@][host:port]/[path][?k1
59  */
60 // org.apache.dubbo.common.URLStrParser#parseDecodedStr
61 public static URL parseDecodedStr(String decodedURLStr) {
62     // 省略了其他内容
63 }

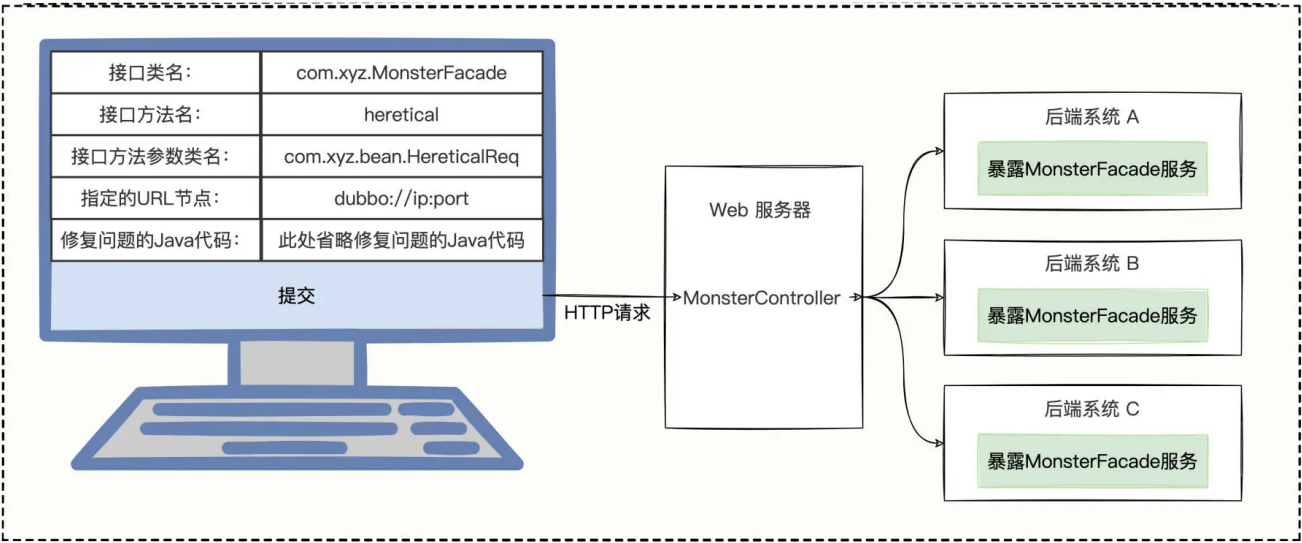
```

这段代码，先是通过 url 找到了 parseUrl 方法，并在 parseUrl 方法上看到了“解析直连的配置地址”的描述信息，至少可以确认一点，确实是为直连点对点调用准备的。

parseUrl 方法被分号切割后传入到了 URL.valueOf 方法中，继续深入 valueOf 的代码逻辑，你会发现最后专门调用了一个 URLStrParser 类来解析为可以被识别的 URL 对象。别忘了看这些方法的描述信息，多处地方都描述着 url 的构成规则为：[protocol://][username:password@][host:port]/[path][?k1=v1&k2=v2]。

相信你现在一定恍然大悟了，url 的构成规则，居然和 http 的构成规则如出一辙，那我们试着通过赋值 url 为dubbo://[机器IP结点]:[机器IP提供Dubbo服务的端口]，应该就大功告成了。

好，我们整理思绪，设计了一下改造的大致思路：



极客时间

首先需要准备一个页面，填入 5 个字段信息，接口类名、接口方法名、接口方法参数类名、指定的 URL 节点、修复问题的 Java 代码，然后将这 5 个字段通过 HTTP 请求发往 Web 服务器，Web 服务器接收到请求后组装泛化所需对象，最后通过泛化调用的形式完成功能修复。

最终 Web 服务器代码和 MonsterFacade 代码设计如下：

复制代码

```
1 @RestController
2 public class MonsterController {
3     // 响应码为成功时的值
4     public static final String SUCC = "000000";
5
6     // 定义URL地址
```

```
@PostMapping("/gateway/repair/request")
```

```
public String repairRequest(@RequestBody RepairRequest repairRequest){
```

```
// 将入参的req转为下游方法的入参对象，并发起远程调用
```

```
return commonInvoke(repairRequest);
```

```
}
```

```
private String commonInvoke(RepairRequest repairRequest) {
```

```
// 然后试图通过类信息对象想办法获取到该类对应的实例对象
```

```
ReferenceConfig<GenericService> referenceConfig =
```

```
createReferenceConfig(repairRequest.getClassName(), repairReque
```

```
// 远程调用
```

```
GenericService genericService = referenceConfig.get();
```

```
Object resp = genericService.$invoke(
```

```
repairRequest.getMtdName(),
```

```
new String[]{repairRequest.getParameterTypeName()},
```

```
new Object[]{JSON.parseObject(repairRequest.getParamsMap(), Map
```

```
// 判断响应对象的响应码，不是成功的话，则组装失败响应
```

```
if(!SUCC.equals(OgnlUtils.getValue(resp, "respCode"))){
```

```
return RespUtils.fail(resp);
```

```
}
```

```
// 如果响应码为成功的话，则组装成功响应
```

```
return RespUtils.ok(resp);
```

```
}
```

```
private static ReferenceConfig<GenericService> createReferenceConfig(String
```

```
DubboBootstrap dubboBootstrap = DubboBootstrap.getInstance();
```

```
// 设置应用服务名称
```

```
ApplicationConfig applicationConfig = new ApplicationConfig();
```

```
applicationConfig.setName(dubboBootstrap.getApplicationModel().getAppli
```

```
// 设置注册中心的地址
```

```
String address = dubboBootstrap.getConfigManager().getRegistries().iter
```

```
RegistryConfig registryConfig = new RegistryConfig(address);
```

```
ReferenceConfig<GenericService> referenceConfig = new ReferenceConfig<>
```

```
referenceConfig.setApplication(applicationConfig);
```

```
referenceConfig.setRegistry(registryConfig);
```

```
referenceConfig.setInterface(className);
```

```
// 设置泛化调用形式
```

```
referenceConfig.setGeneric("true");
```

```
// 设置默认超时时间5秒
```

```
referenceConfig.setTimeout(5 * 1000);
```

```
// 设置点对点连接的地址
```

```
referenceConfig.setUrl(url);
```

```
return referenceConfig;
```

```
}
```

```
}
```

```
}
```

```
@Setter
```

```
@Getter
```

```
public class RepairRequest {
```



天下无鱼

<https://shikey.com/>

```

59  /** <h2>接口类名，例：com.xyz.MonsterFacade</h2> */
60  private String className;
61  /** <h2>接口方法名，例：heretical</h2> */
62  private String mtdName;
63  /** <h2>接口方法参数类名，例：com.xyz.bean.HereticalReq</h2> */
64  private String parameterTypeName;
65  /** <h2>指定的URL节点，例：dubbo://ip:port</h2> */
66  private String url;
67  /** <h2>可以是调用具体接口的请求参数，也可以是修复问题的Java代码</h2> */
68  private String paramsMap;
69

```




天下无鱼

<https://shikey.com/>

这段代码在 Web 服务器中完成了页面数据的转发，主要步骤是 3 点：

- 首先，定义一个 `MonsterController` 控制器专门来接收页面的请求。
- 其次，创建泛化调用所需的 `referenceConfig` 对象，并将 `url` 设置到 `referenceConfig` 对象中。
- 最后，套用之前学过的泛化调用代码，完成页面数据的转发。

有了 Web 服务器的代码还不够，缺少了最重要的万能管控 `MonsterFacade` 的核心逻辑：

 复制代码

```

1  public interface MonsterFacade {
2      // 定义了一个专门处理万能修复逻辑的Dubbo接口
3      AbstractResponse heretical(HereticalReq req);
4  }
5
6  public class MonsterFacadeImpl implements MonsterFacade {
7      @Override
8      AbstractResponse heretical(HereticalReq req){
9          // 编译 Java 代码，然后变成 JVM 可识别的 Class 对象信息
10         Class<?> javaClass = compile(req.getJavaCode());
11
12         // 为 Class 对象信息，自定义一个名称，将来创建 Spring 单例对象要用到
13         String beanName = "Custom" + javaClass.getSimpleName();
14
15         // 通过 Spring 来创建单例对象
16         generateSpringBean(beanName, javaClass);
17
18         // 获取 beanName 对应的单例对象
19         MonsterInvokeRunnable runnable = (MonsterAction)SpringContextUtils.getB
20
21         // 执行单例对象的方法即可
22         Object resp = runnable.run(req.getReqParamsMap());
23
24         // 返回结果

```

```
25     return new AbstractResponse(resp);
26 }
27
28 // 利用 groovy-all.jar 中的 groovyClassLoader 来编译 Java 代码
29 private Class<?> compile(String javaCode){
30     return groovyClassLoader.parseClass(javaCode);
31 }
32
33 // 生成Spring容器Bean对象
34 private void generateSpringBean(String beanName, Class<?> javaClass){
35     // 构建 Bean 定义对象
36     BeanDefinitionBuilder beanDefinitionBuilder =
37         BeanDefinitionBuilder.genericBeanDefinition(javaClass);
38     AbstractBeanDefinition rawBeanDefinition = beanDefinitionBuilder.getRaw
39
40     // 将 bean 移交给 Spring 去管理
41     ConfigurableApplicationContext appCtx =
42         (ConfigurableApplicationContext)SpringContextUtils.getContext()
43     appCtx.getAutowireCapableBeanFactory()
44         .applyBeanPostProcessorsAfterInitialization(rawBeanDefinition,
45         ((BeanDefinitionRegistry)appCtx.getBeanFactory()).registerBeanDefinitio
46 }
47 }
48
```



天下无鱼

<https://shikey.com/>

这段代码使用 Groovy 和 Spring，完成了万能管控代码的最核心逻辑：

- 首先，将接收的 Java 代码利用 Groovy 插件编译为 Class 对象。
- 其次，将得到的 Class 对象移交给 Spring 容器去创建单例 Bean 对象。
- 最后，调用单例 Bean 对象的 run 方法，完成最终动态 Java 代码的逻辑执行，并达到修复功能的目的。

点点直连的应用

好，点点直连的代码逻辑我们就掌握了，之后如果能应用到自己的项目中，相信你再也不用担心紧急的数据订正事件了。在日常开发中，哪些应用场景可以考虑点点直连呢？

第一，修复产线事件，通过直连 + 泛化 + 动态代码编译执行，可以轻松临时解决产线棘手的问题。

第二，绕过注册中心直接联调测试，有些公司由于测试环境的复杂性，有时候不得不采用简单的直连方式，来快速联调测试验证功能。

第三，检查服务存活状态，如果需要针对多台机器进行存活检查，那就需要循环调用所有服务的存活检查接口。



总结

今天，我们从一个修复产线数据的事件开始，通过正规流程、粗暴流程、万能管控三种方式来尝试快速解决产线问题。

正规流程需要提交申请或者邮件审批，在时间紧、任务急的情况下，因公司而异，可能并非最快的方式。粗暴流程，通过模拟用户请求、供应商请求、`invoke` 命令调用接口等非主流方式，虽然可以碰巧解决，但是并非每次都有好运气，而且还有各种 `Linux` 命令的权限限制，实属不易。

在万能管控中，通过页面发送 `HTTP` 请求、泛化调用、点对点直连、`Groovy` 插件编译、`Spring` 实例化对象等一系列组合方式，我们完成了最简单、最实用修复数据的平台搭建。

这里总结一下通过直连进行泛化调用的三部曲：

- 接口类名、接口方法名、接口方法参数类名、业务请求参数，四个维度的数据不能少。
- 根据接口类名创建 `ReferenceConfig` 对象，设置 `generic = true`、`url = 协议 + IP + PORT` 两个重要属性，调用 `referenceConfig.get` 拿到 `genericService` 泛化对象。
- 传入接口方法名、接口方法参数类名、业务请求参数，调用 `genericService.$invoke` 方法拿到响应对象，并通过 `Ognl` 表达式语言判断响应成功或失败，然后完成数据最终返回。

最后总结一下 `Groovy+Spring` 完成动态编译调用的三部曲：

- 首先，将 `Java` 代码利用 `Groovy` 插件的 `groovyClassLoader` 加载器编译为 `Class` 对象。
- 其次，将 `Class` 信息创建 `Bean` 定义对象后，移交给 `Spring` 容器去创建单例 `Bean` 对象。
- 最后，调用单例 `Bean` 对象的 `run` 方法，完成动态代码调用。

点点直连的应用场景主要有 3 类，修复产线事件，绕过注册中心直接联调测试，检查服务存活状态。

思考题

你已经学会了点点直连的精髓用处，运用泛化调用方式搭建一套数据订正的平台能大大提升解决产线问题的效率。那你能否尝试研究一下源码中，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呢？



期待看到你的思考，如果你对今天的内容还有什么困惑，欢迎在留言区提问，我会第一时间回复。我们下一讲见。

参考资料

如果你对 Groovy 的使用还有疑问，我之前录制了 [如何使用 Groovy 动态加载 Java 代码为 class 并注册 Spring](#) 可以学习。

04 思考题参考

上一期的问题是 `CommonController` 这套泛化调用流程的代码，有哪些可以改善。

1. `createReferenceConfig` 方法中，`address` 注册中心地址的获取，如果是多注册中心，且对注册中心的选择比较敏感，这里可能需要根据一些变量标识来做出相应选择。
2. `createReferenceConfig` 方法中的 `referenceConfig` 对象，有许多设置接口的类、方法维度的属性，这里我为了演示效果仅设置了 `timeout` 属性，若有更多参数设置的诉求，可以入参传进来，或者通过配置中心来支持不同接口的个性化参数设置。
3. `commonInvoke` 方法中的 `genericService.$invoke` 调用参数，目前 `parameterTypeName` 不支持集合类型的类名。如果需要在支持集合类型，还得针对 `parameterTypeName`、`reqParamsStr` 两个参数做一定的约定设计。
4. `commonInvoke` 方法中 `respCode` 响应码的判断，并不是所有下游系统的响应码字段都叫 `respCode` 这个名字，也可能有 `resCode`、`errCode`、`errorCode` 等等，而且下游系统也并不都是用 `000000` 表示成功码，也可能是其他形式，总之具有不确定性，这一块可以考虑通过入参形式或者配置中心方式，利用 `Ognl` 表达式来做简单的动态化处理。
5. `commonRequest` 方法只支持 `POST` 方式，若需要在支持 `GET` 方式，还得设计一套适合 `GET` 方式调用的方法逻辑。
6. `@PostMapping` 里面的地址，`/gateway/{className}/{mtdName}/request` 是占位符形式的，将来用户请求时，输入的是这样
`/gateway/com.hmily.cloud.QueryUserFacade/queryUserInfo/request` 比较低 `low` 的 URL 地址，会暴露一些“类名”“方法名”“方法参数类名”，相当于把后端的一些本不该

让用户感知的代码暴露给了用户。这样的设计在内网比较常见。如果觉得碍眼不规范，可以考虑建立一套“标准 URL”与“占位符格式 URL”的映射关系，可以存储在配置中心，然后在 Web 应用启动的时候，利用 `RequestMappingHandlerMapping` 中的注册方法动态将标准 URL 注册到 Web 容器中。

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 1 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 04 | 泛化调用：三步教你搭建通用的泛化调用框架

下一篇 06 | 事件通知：一招打败各种神乎其神的回调事件

精选留言 (5)

写留言



熊悟空的凶

2022-12-28 来自广东

1. 这种点点直连的应用是不是也不是安全的，万一参数填错了等可能会有风险；如果线上问题晋级，写代码可能来不及。
2. 这种是不是在测试环境代码改后 可以用热更新class的方式处理（如arthas等）

作者回复: 你好，熊悟空的凶：任何技术带来了超级的便利性，也同时存在着一定的不友好性，正所谓有利有弊，就看我们怎么看待怎么来规范约束大家来使用罢了。

- 1、修改产线数据，就不应该存在填错的因素，一般填错了就是连带责任，所以要对自己修改的任何数据要抱着严谨认真的态度，经过反复验证没问题后才执行到产线的。
- 2、肯定要在测试环境自己充分验证没问题才执行到产线的，至于想使用arthas的话，只要是你们公司认可，适合你们公司的，适合绝大多数开发人员操作习惯的，都是可以的。



1



张申傲

2023-02-01 来自北京

点点直连可以理解为泛化调用的一种应用场景，只要掌握了泛化调用，再了解 Dubbo 通用的协议格式，实现点点直连就不在话下了~



天下无鱼

<https://shikey.com/>

作者回复: 你好，张申傲：是的，你理解的是对的，对点点直连和泛化调用理解的很透彻，点赞~



星期八

2023-01-14 来自浙江

有点疑问：泛化调用，MonsterFacade接口在consumer侧有实现MonsterFacadeImpl类，所以泛化调用是调用本地的吗？如果是调用本地的MonsterFacadeImpl类，那么referenceConfig.setUrl(url);这段配置是给谁的用的？

作者回复: 你好，星期八：泛化调用是调用远程的，即实现消费方调用提供方一种高级代码编写形式。

setUrl 的目的是想办法在消费方调用到指定IP地址的提供方。



Six Days

2023-01-13 来自广东

思考题：dubbo 框架提供了不同的协议类型，通过dubbo协议直连的话，dubbo 传输模块则通过提供的ip和端口定位到具体服务实例，进行服务调用

作者回复: 你好，Six Days：特别好，抓住了传输模块，抓住了关键的IP和PORT，列出了这几个重要的信息，大方向是对的，剩下的就是一些细节用什么工具实现的问题了。



天天有吃的

2022-12-29 来自北京

完整的git，什么时候会发出来呀

编辑回复: 已经发出来啦，课程的代码仓库 <https://gitee.com/yimhhmily/GeekDubbo3Tutorial>

