

## 23 | 集群扩展：发送请求遇到服务不可用，怎么办？

2023-02-08 何辉 来自北京

《Dubbo源码剖析与实战》

课程介绍 >



讲述：何辉

时长 17:58 大小 16.41M



你好，我是何辉。

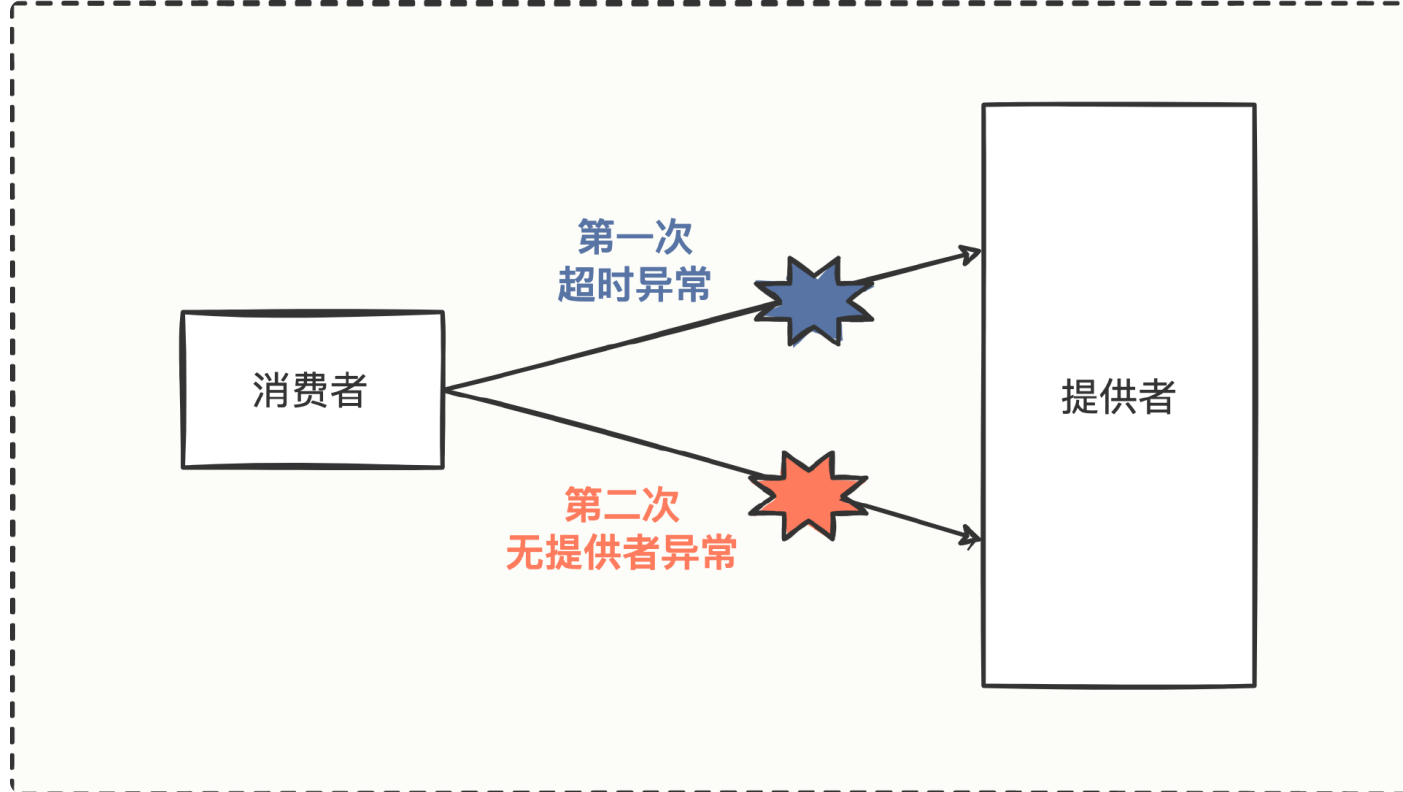
通过特色篇的学习，你可以在日常开发中横着走了，如果你继续深入掌握了源码篇，基本可以把 Dubbo 框架游刃有余地玩弄于鼓掌之中。接下来，我们将进入拓展篇，借助一些产线的真实案例，真枪实弹地教你如何充分挖掘 Dubbo 框架的扩展能力，来实际问题。

今天我们来实操第一个扩展，集群扩展。

你有没有遇到过这样的情况，对于多机房部署的系统应用，线上运行一直比较稳定，可是突然在某一段时间内，部分流量请求先出现一些超时异常，紧接着又出现一些无提供者的异常，最后部分功能就不可用了。

我们结合具体的调用链路图来看。





这是一张消费方调用提供方的简单调用链路图，消费方第一次调用时发生了超时异常，然后第二次调用时却发生了无提供者异常，无提供者异常从字面意思上理解，就是没有服务提供者，结果和服务不可用是一样的。

对于这样一个看似非常简单的异常现象，我们该怎么解决呢？

## 异常如何解决

要想知道怎么解决，首先就得弄清楚异常发生的原因。

调用链路图中可以看到有两个异常，一个是超时异常，一个是无提供者异常。我们先从超时异常开始分析。

### 1. 超时异常和原因

遇到异常，我们的第一反应就是去认真阅读异常堆栈的详细信息。

你回忆了下超时异常堆栈信息。

领资料

复制代码

```
1 Caused by: org.apache.dubbo.remoting.TimeoutException: Waiting server-side resp
2   at org.apache.dubbo.remoting.exchange.support.DefaultFuture.doReceived(Default
```

```
3 at org.apache.dubbo.remoting.exchange.support.DefaultFuture.received(DefaultF
4 at org.apache.dubbo.remoting.exchange.support.DefaultFuture$TimeoutCheckTask.
5 at org.apache.dubbo.remoting.exchange.support.DefaultFuture$TimeoutCheckTask.
6 at org.apache.dubbo.common.threadpool.ThreadlessExecutor$RunnableWrapper.run(
7 at org.apache.dubbo.common.threadpool.ThreadlessExecutor.waitAndDrain(Threadl
8 at org.apache.dubbo.rpc.AsyncRpcResult.get(AsyncRpcResult.java:193)
9 ... 29 more
```

说明一下，这段报错信息是方便我们具体分析问题模拟出来的，但日常你看到消费方调用的超时异常信息也大同小异，思路都是很类似的。

从异常信息中我们看到有一个 **Caused by** 引发的 **TimeoutException** 异常类，而且异常堆栈都这么明显了，很多人就理所当然地认为是超时异常。

但我们不妨认真想想，**这个异常，单方面从消费方就能找到超时的真正原因么？**

你恍然大悟，好像是哦，单纯从消费方判断超时异常并不能定位到真实原因，得明确下提供方是不是也真的发生了超时异常情况。如果提供方有异常，那好说，顺藤摸瓜找到真正异常的堆栈日志就能继续分析异常原因。

**但如果提供方根本没有收到消费方的任何请求，又该如何继续排查呢？**

消费方有请求，但提供方未收到，难道请求发错了？好端端的线上应用，会出现 Dubbo 负载均衡到错误的目标 IP 上，好像有点匪夷所思。虽然不太相信，但我们本着严谨的态度，还是来求证下目标 IP 是否真的有问题。**有哪些特征可以证明目标 IP 有问题？或者有个什么地方可以看一看目标 IP 的健康状态呢？**

快速联想日常开发中可以监测 IP 健康情况的工具，你一定能胸有成竹地说出 3 个方案：可以从 CAT 监控平台上观察一下目标 IP 是否可以继续接收流量，也可以从 Prometheus 上观察消费方到目标 IP 的 TCP 连接状况，还可以从网络层面通过 tcpdump 抓包检测消费方到目标 IP 的连通性等等。

领资料

好，为求证目标 IP 的健康状况，我们把刚刚提到的检测工具挨个试一下：

- 在 CAT 上发现了目标 IP 在出问题期间几乎没有任何流量进来。

- 在 Prometheus 上发现在最近一段时间内 TCP 的连接耗时特别大，基本上都是有 SYN 请求握手包，但是没有 SYN ACK 响应包。
- 找网络人员帮忙实时 tcpdump 抓包测试，结果仍然发现没有 SYN ACK 响应包。

于是通过监测结论，我们基本确认了目标 IP 处于不可连通的状态，接着只需要去确认目标 IP 服务是宕机了还是流量被拦截了，就大概知道真相了。

经过我们一连串提问和思考，机智的你也许会说，为什么不直接去看看目标 IP 服务是否存活呢？其实也可以，毕竟问题有千万种，只要掌握排查问题的思路，尝试跳跃性地排查也是可以的。

但是到这里还没完，我们只是解决了第一个超时异常，有些节点能正常提供服务，有些节点无法提供服务，更巧的是紧接着还冒出无提供者的异常，这又是怎么回事呢？

## 2. 无提供者异常和原因

无提供者异常，相信你看到这样的异常应该也不陌生，异常堆栈信息长这样。

 复制代码

```
1 org.apache.dubbo.rpc.RpcException: Failed to invoke the method sayHello in the
2   at org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker.checkInvokers(
3   at org.apache.dubbo.rpc.cluster.support.FailoverClusterInvoker.doInvoke(Failo
4   at org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker.invoke(Abstrac
5   at org.apache.dubbo.rpc.cluster.router.RouterSnapshotFilter.invoke(RouterSnap
6   at org.apache.dubbo.rpc.cluster.filter.FilterChainBuilder$CopyOfFilterChainNc
7   at org.apache.dubbo.monitor.support.MonitorFilter.invoke(MonitorFilter.java:9
8   ... 48 more
```

我们精准找出了“No provider available”这样的关键字，说明的确就是找不到服务提供者。可是看着这段再平常不过的异常信息，有点无解了，为什么会报无提供者异常呢？到底是哪个环节走不过去了？又或者是哪段代码导致逻辑走不通了？

 领资料


想到哪段代码，你好像找到了破局之道，我们再三观察堆栈异常信息。

 复制代码

```
1 at org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker.checkInvokers(Ab
2 at org.apache.dubbo.rpc.cluster.support.FailoverClusterInvoker.doInvoke(Failove
```

发现报错的关键点日志，是在 `AbstractClusterInvoker` 类中的第 366 行报错。

于是打开 `AbstractClusterInvoker` 源码对应的报错地方。

 复制代码

```
1 protected void checkInvokers(List<Invoker<T>> invokers, Invocation invocation)
2     // 检查传入的 invokers 服务提供者列表，若集合为空，则会抛出无提供者异常
3     if (CollectionUtils.isEmpty(invokers)) {
4         // 抛出的 RpcException 异常信息中，会有 No provider available 明显的关键字
5         throw new RpcException(RpcException.NO_INVOKER_AVAILABLE_AFTER_FILTER,
6             + invocation.getMethodName() + " in the service " + getInterface().
7             + ". No provider available for the service " + getDirectory().getCc
8             + " from registry " + getDirectory().getUrl().getAddress()
9             + " on the consumer " + NetUtils.getLocalHost()
10            + " using the dubbo version " + Version.getVersion()
11            + ". Please check if the providers have been started and registered
12    }
13 }
```

通过这段代码中的 `checkInvokers` 方法实现逻辑，我们可以得知，**消费方在内存中找不到对应的提供者，才会提示无提供者异常**。而 `checkInvokers` 方法又被 `FailoverClusterInvoker` 类调用，而这个 `FailoverClusterInvoker` 类，想必你已经联想到了 `FailoverCluster` 类，于是进一步推导是设置了 `cluster = "failover"` 属性才能走到 `FailoverCluster` 类的逻辑中来。

而 `FailoverCluster` 对象的生成是在消费方订阅环节中，另外提供方结点发生变更时，消费方感知到注册中心节点的变化也会重新生成 `FailoverCluster` 对象。

这一通梳理后，`checkInvokers` 方法中 `invokers` 列表对应的源数据，想必你已经明白从哪里来了，消费方启动或注册中心节点变更都会更新这份源数据。

知道这份源数据的来源，我们再结合报错细想，既然消费方已经处于运行状态，只是在调用的时候发生了无提供者异常，说明什么？



你思索了几秒，难道是因为注册中心的节点变更了，导致源数据被更新没了？

没错，就是这么理解。至于为什么注册中心的节点会变更，原因就很多了，比如：

- 服务方动态注销了某个接口服务。
- 服务方节点宕机了。
- 服务方在代码中删掉了这个接口再启动，即永远不会再提供服务了。
- 服务方修改了接口的类名、方法名。
- 服务方未考虑版本兼容性，主动添加了 `group`、`version` 等参数。
- .....

有没有发现，源码层面抛出的一个“No provider available”无提供者异常，居然有这么多的可能场景存在。

通过一系列可能原因的排查，最终我们排查到因为节点宕机，造成提供方节点与注册中心断开心跳连接了，这意味着注册中心会删掉提供方的 IP 节点；然后消费方感知到注册中心的节点发生变更，会更新消费方本地的源数据信息。这也就是为什么消费方在 `checkInvokers` 中发现 `invokers` 为空了。

### 3. 问题总结

分析了超时异常、无提供者异常，我们最终发现是因为某些提供方的 IP 节点宕机，但是还有些提供方的节点 IP 是正常提供服务的，这又是为什么呢？

IP 一些正常一些不正常，难道 IP 还有人品因素么，这明显不现实，你盯着有问题的那一堆 IP 思索了良久，无解。是时候找外援了，在一个公司中谁对 IP 非常熟悉？网络工程师。

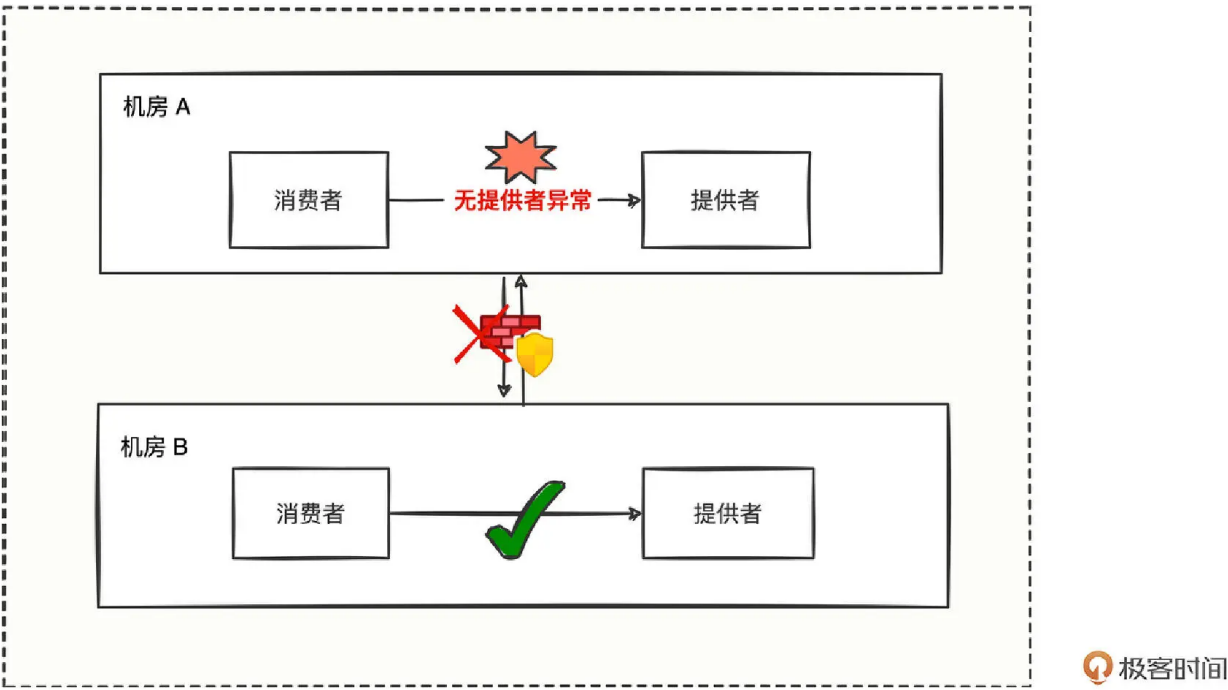
于是我们拿着这一堆有问题的 IP 去找网工，得知它们全都是同一个机房的，再结合刚才的整个调用流程步骤思考：

1. 某机房的某提供方 IP 节点宕机
2. 消费方有缓存，引发消费方调用超时
3. 消费方感知到注册中心节点变更
4. 消费方更新接口对应提供方源数据列表
5. 消费方又有流量发起调用发生无提供者异常





可以梳理出一张不同机房之间消费者调用提供者的链路图。



我们可以看到机房 A 与机房 B 是不能相互访问的，这也正符合机房的隔离性。

那么假设机房 A 的某些提供者宕机了，且机房 A 的消费者状态正常，难道就预示着机房 A 的消费方发起的请求就无法正常调用了么？这样明显不合理，为什么机房 A 的提供者有问题，非得把机房 A 的消费者拉下水导致各种功能无法正常运转。

遇到这种状况，我们该如何改善呢？

## 定制 Cluster 扩展

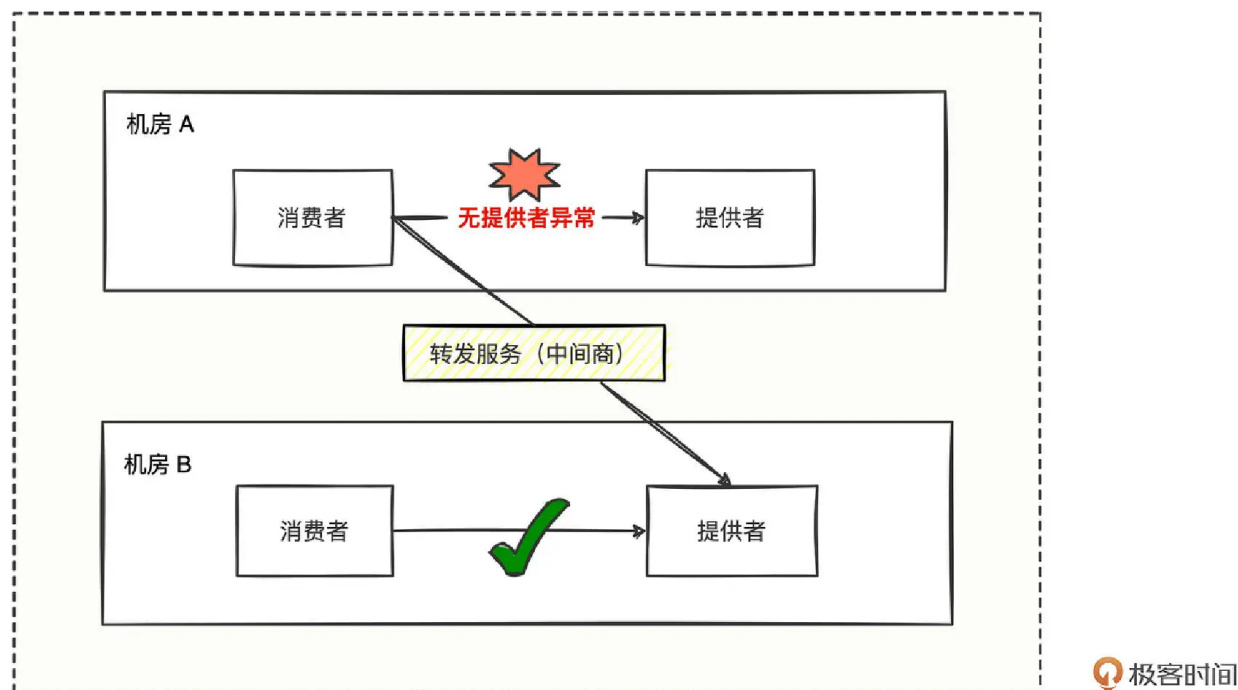
整理现在的问题，在机房 A 的消费者无法调用提供者的时候，我们要想办法让消费者能正常调用提供者拿到结果。

问题乍一看，你是不是感觉自相矛盾。明明同机房已经调用不通，再加上机房之间又是防火墙彻底隔离，还要硬着头皮从宕机的服务拿到响应内容？除非提供方宕机之后又以神一样的毫秒级别恢复起来了，但是我们现在的各种应用，还没有能在毫秒级别启动成功的，基本上都是秒级，大多数都是数十秒才启动成功。

顺着同机房的思路想必是走进死胡同了，那消费者的请求发给谁才能通呢？



发给谁？突然你眼前一亮，消费者可以将请求发给**中间商**啊，然后中间商想办法找可用提供者，貌似这条路可行。



消费方遇到无提供者异常后，掉头去调用转发服务，由转发服务找可用机房的可用提供者并发起调用并拿到结果，就可以让消费者拿到正常调用的结果。

但问题又来了，**谁写这段调用转发服务的代码呢？**是消费者去写么？还是谁呢？

消费者写也不是不可以，但我们想象一下场景，今天这个应用的消费者遇到无提供者异常需要转发，明天那个应用的消费者遇到无提供者异常需要转发，那么多的系统各种消费方都需要改，肯定是改不过来的。

既然写不过来，那把这段代码封装成插件总可以了，通过提炼公共插件，解决大批量应用的共性问题，这样既做到了代码公用只维护一份代码，又做到了对应用的非侵入特性，岂不是一举两得。

但是问题又来了，**该在调用的哪个环节进行转发服务的处理呢？**

我们再琢磨下，要调用转发服务，就得感知无提供者异常；要想感知无提供者异常，就得在抛异常的外层进行异常捕获处理。

领资料



顺着这个思路，我们去代码翻了翻抛异常的地方，发现在 `FailoverClusterInvoker#doInvoke` 方法中 `checkInvokers` 方法，细心的你还发现了 `checkInvokers` 方法是被 `protected` 修饰的，而 `protected` 修饰就意味着可以被子类重写，说明源码给我们提供了一个检测 `invokers` 是否可用的机制。

所以根据源码，你准备定义一个 **TransferClusterInvoker** 类来继承

**FailoverClusterInvoker**，然后重写 `checkInvokers` 就可以了。但是 `checkInvokers` 只是意在让你检测，不是让你发现 `invokers` 为空就调用转发服务，虽说重写 `checkInvokers` 方法可以达到目的，但和设计者的命名始终有点不符。

既然与设计意图不符，我们又看到 `FailoverClusterInvoker` 的 `doInvoke` 方法是被 `public` 修饰后，是不是可以在子类 **TransferClusterInvoker** 中调用父类 **FailoverClusterInvoker** 的 `doInvoke` 方法，并进行 `try...catch...` 捕获精准异常呢？这样既不会破坏设计者的意图，还能精准处理无提供者异常后转发调用，也是挺完美的。

我们来编写代码。

 复制代码

```
1 public class TransferClusterInvoker<T> extends FailoverClusterInvoker<T> {
2     // 按照父类 FailoverClusterInvoker 要求创建的构造方法
3     public TransferClusterInvoker(Directory<T> directory) {
4         super(directory);
5     }
6     // 重写父类 doInvoke 发起远程调用的接口
7     @Override
8     public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance balancer) {
9         try {
10             // 先完全按照父类的业务逻辑调用处理，无异常则直接将结果返回
11             return super.doInvoke(invocation, invokers, balancer);
12         } catch (RpcException e) {
13             // 这里就进入了 RpcException 处理逻辑
14
15             // 当调用发现无提供者异常描述信息时则向转发服务发起调用
16             if (e.getMessage().toLowerCase().contains("no provider available")) {
17                 // TODO 从 invocation 中拿到所有的参数，然后再处理调用转发服务的逻辑
18                 return doTransferInvoke(invocation);
19             }
20             // 如果不是无提供者异常，则不做任何处理，异常该怎么抛就怎么抛
21             throw e;
22         }
23     }
24 }
```

领资料

在这段代码中 `TransferClusterInvoker` 主要继承了 `FailoverClusterInvoker` 类，然后重写了父类中最重要的 `doInvoke` 方法，重写的实现体逻辑中原封不动地通过 `super.doInvoke` 方法调用父类逻辑，在父类逻辑出现“No provider available”无提供者异常时，才捕捉处理调用转发服务。

最核心的逻辑我们已经实现了，那在代码中该怎么触发这个 `TransferClusterInvoker` 运作呢？

我们还是可以借鉴已有源码编写调用的思路，经过一番查找 `FailoverClusterInvoker` 代码编写调用关系后，我们可以定义一个 `TransferCluster` 类。

 复制代码

```
1 public class TransferCluster implements Cluster {
2     // 返回自定义的 Invoker 调用器
3     @Override
4     public <T> Invoker<T> join(Directory<T> directory, boolean buildFilterChain
5         return new TransferClusterInvoker<T>(directory);
6     }
7 }
```

这段代码中 `TransferCluster` 主要仿照了 `FailoverCluster` 的编码形式，通过在 `TransferCluster` 类中创建 `TransferClusterInvoker` 来处理调用至转发服务器。

当 `TransferClusterInvoker`、`TransferCluster` 都实现好后，按照 Dubbo SPI 的规范将 `TransferCluster` 类路径配置到 `META-INF/dubbo/org.apache.dubbo.rpc.cluster.Cluster` 文件中，看配置。

 复制代码

```
1 transfer=com.hmilyylimh.cloud.TransferCluster
```

这段配置为 `TransferCluster` 取了个 `transfer` 名字，将来程序调用的时候能够被用上。



最后我们只需要在 `dubbo.properties` 指定全局使用，看配置。

 复制代码

```
1 dubbo.consumer.cluster=transfer
```

全局指定名字叫 **transfer** 的集群扩展器，然后在调用时会触发 **TransferCluster** 来执行遇到无提供者异常时，掉头转向调用转发服务器。

## 集群拓展的应用

Dubbo 的集群扩展，相信你现在已经非常清楚了。**Cluster** 作为路由层，封装多个提供方的路由及负载均衡，并桥接注册中心以 **Invoker** 为中心发起调用，那哪些应用场景可以考虑集群扩展呢？

第一，同机房请求无法连通时，可以考虑转发 **HTTP** 请求至可用提供者。

第二，内网本机访问测试环境无法连通时，可以转发请求至 **HTTP** 协议的接口，然后在接口中泛化调用各种 **Dubbo** 服务。

第三，如果针对接口的多个提供者需要做适应当前公司业务的筛选、剔除、负载均衡之类的诉求时，也是可以考虑集群扩展的。

总之，不管是无提供者问题，还是公司特殊定制化筛选负载问题，核心都是在针对接口的所有提供者做逻辑处理，提供者为空做转发兼容处理，提供者不为空做特殊筛选负载处理，目的都是在调用时做一种容错兼容处理，让应用程序更加健壮稳定。

## 总结

今天，我们从一个消费方在同机房调用连续发生两种异常开始，分析了超时异常和无提供者异常大致方向的定位排查。

通过 **CAT**、**Prometheus**、**tcpdump** 可以发现引发超时的深层次原因。通过“**No provider available**”关键字，从代码层面反推引起注册节点变更的可能因素，来进一步找到引发无提供者异常的底层原因；然后深入挖掘错误关键字在代码中的报错点，为问题的解决撕开了一道口子，最终通过自定义集群扩展从代码层面落实方案。



这里也总结一下自定义集群扩展四部曲。

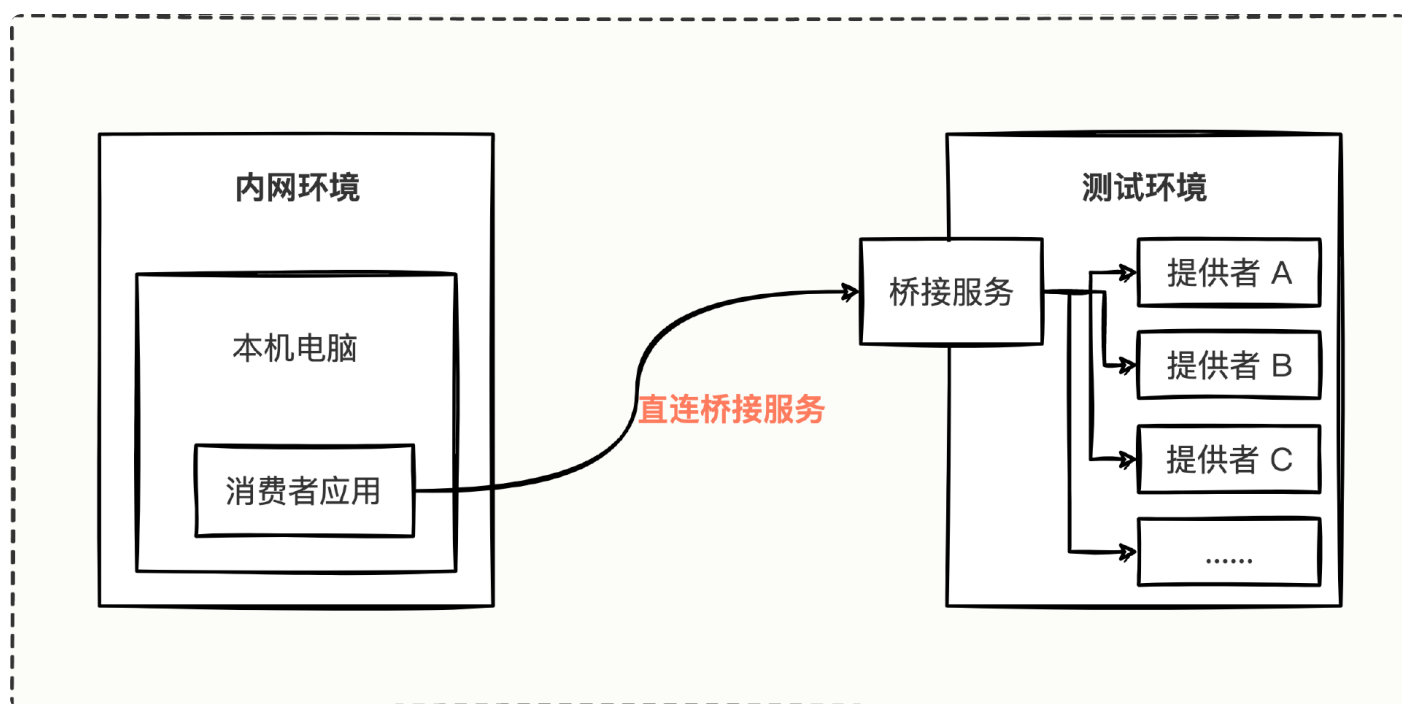
- 首先定义一个 **TransferClusterInvoker** 集群扩展处理器来处理核心的调用转发逻辑。
- 其次定义一个 **TransferCluster** 集群扩展器来封装集群扩展处理器。

- 然后在 META-INF/dubbo/org.apache.dubbo.rpc.cluster.Cluster 文件中定义 TransferCluster 的类路径并取个别名。
- 最后在 dubbo.properties 配置文件通过一个别名来指定消费者需要使用的集群扩展器。

集群扩展的应用场景主要有 3 类，同机房请求无法连通，内网本机请求测试环境无法连通，特殊业务需要对接口的多个提供者进行筛选、剔除、负载均衡等诉求。

## 思考题

你已经学会了如何使用集群扩展特性来处理无法连通的问题了，那你能否尝试研究并编写代码来模拟实现应用场景中内网电脑请求测试环境的连通问题呢？



图中描绘了内网环境调用测试环境的调用链路图，虚拟出一个桥接服务节点，接下来需要想办法让消费者应用的所有接口，都能通过桥接服务，调用到对应的提供者。

这个问题涉及的知识体系比较庞大，所以这里，我们只实现“**集群扩展**”“**桥接服务通用 HTTP 接口的定义**”这两部分功能就可以了。小提示，集群扩展的实现可以参考今天总结的自定义集群扩展四部曲，而 HTTP 接口要关注入参的通用性。

领资料

期待在留言区看到你的思考，参与讨论。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，可能就帮 TA 解决了一个困惑。我们下一讲见。

上一期留了个作业，研究下 `encodeResponse` 方法的逻辑，总结与 `encodeRequest` 的异同点。

这个问题其实不难，我们只要认真分析一遍 `encodeResponse` 方法，答案也就呼之欲出了。直接进入 `encodeResponse` 的方法仔细看看。

 复制代码

```
1 ///////////////////////////////////////////////////
2 // 1、 org.apache.dubbo.remoting.exchange.codec.ExchangeCodec#encodeResponse
3 // 2、将 Response 对象按照 Dubbo 协议格式编码为字节流
4 // 3、重点关注我在代码中描述的 ①②③④⑤⑥⑦ 几个关键位置
5 ///////////////////////////////////////////////////
6 // header length.
7 protected static final int HEADER_LENGTH = 16;
8 // magic header.
9 protected static final short MAGIC = (short) 0xdabb;
10 protected static final byte MAGIC_HIGH = Bytes.short2bytes(MAGIC)[0];
11 protected static final byte MAGIC_LOW = Bytes.short2bytes(MAGIC)[1];
12 // message flag.
13 protected static final byte FLAG_REQUEST = (byte) 0x80;
14 protected static final byte FLAG_TWOWAY = (byte) 0x40;
15 protected static final byte FLAG_EVENT = (byte) 0x20;
16 protected static final int SERIALIZATION_MASK = 0x1f;
17
18 // 将 Response 对象按照 Dubbo 协议格式编码为字节流
19 protected void encodeResponse(Channel channel, ChannelBuffer buffer, Response r
20     int savedWriteIndex = buffer.writerIndex();
21     try {
22         Serialization serialization = getSerialization(channel, res);
23         // header.
24         // ① 针对 header 字节数组赋值一个 0xdabb 魔术值
25         byte[] header = new byte[HEADER_LENGTH];
26         // set magic number.
27         Bytes.short2bytes(MAGIC, header);
28
29         // set request and serialization flag.
30         // ② 设置序列化方式，序列化方式从 channel 的 url 取出 serialization 对应的参数
31         header[2] = serialization.getContentTypeId();
32         if (res.isHeartbeat()) {
33             // ③ 设置请求类型，根据 mTwoWay、mEvent 来决定是怎样的请求类型
34             header[2] |= FLAG_EVENT;
35         }
36
37         // set response status.
38         // ④ 设置响应码，这里因为是响应成功，因此 status = 20
39         //
```

领资料

```

40 // status 的值，在源码中有如下这些值：
41 // /** ok. 响应成功 */
42 // public static final byte OK = 20;
43 // /** client side timeout. 消费方超时 */
44 // public static final byte CLIENT_TIMEOUT = 30;
45 // /** server side timeout. 提供方超时 */
46 // public static final byte SERVER_TIMEOUT = 31;
47 // /** channel inactive, directly return the unfinished requests. 通道cl
48 // public static final byte CHANNEL_INACTIVE = 35;
49 // /** request format error. 请求数据格式错误 */
50 // public static final byte BAD_REQUEST = 40;
51 // /** response format error. 响应数据格式错误 */
52 // public static final byte BAD_RESPONSE = 50;
53 // /** service not found. 提供方服务找不到 */
54 // public static final byte SERVICE_NOT_FOUND = 60;
55 // /** service error. 提供方错误 */
56 // public static final byte SERVICE_ERROR = 70;
57 // /** internal server error. 提供方内部错误 */
58 // public static final byte SERVER_ERROR = 80;
59 // /** internal server error. 消费方内部错误 */
60 // public static final byte CLIENT_ERROR = 90;
61 // /** server side threadpool exhausted and quick return. 线程池满拒绝请求
62 // public static final byte SERVER_THREADPOOL_EXHAUSTED_ERROR = 100;
63 byte status = res.getStatus();
64 header[3] = status;
65
66 // ⑤ 设置响应ID，该ID是请求发送过来的ID
67 // set request id.
68 Bytes.long2bytes(res.getId(), header, 4);
69 buffer.writerIndex(savedWriteIndex + HEADER_LENGTH);
70 ChannelBufferOutputStream bos = new ChannelBufferOutputStream(buffer);
71
72 // ⑥ 构建一个输出流，根据 mEvent 的值来将 mData 进行序列化转为字节数组
73 // encode response data or error message.
74 if (status == Response.OK) {
75     if(res.isHeartbeat()){
76         // heartbeat response data is always null
77         bos.write(CodecSupport.getNullBytesOf(serialization));
78     }else {
79         ObjectOutput out = serialization.serialize(channel.getUrl(), bc
80         if (res.isEvent()) {
81             encodeEventData(channel, out, res.getResult());
82         } else {
83             encodeResponseData(channel, out, res.getResult(), res.getVe
84         }
85         out.flushBuffer();
86         if (out instanceof Cleanable) {
87             ((Cleanable) out).cleanup();
88         }
89     }
90 } else {
91     ObjectOutput out = serialization.serialize(channel.getUrl(), bos);

```



```

92         out.writeUTF(res.getMessage());
93         out.flushBuffer();
94         if (out instanceof Cleanable) {
95             ((Cleanable) out).cleanup();
96         }
97     }
98     bos.flush();
99     bos.close();
100     int len = bos.writtenBytes();
101     checkPayload(channel, len);
102     Bytes.int2bytes(len, header, 12);
103     // write
104     buffer.writerIndex(savedWriteIndex);
105     buffer.writeBytes(header); // write header.
106
107     // ⑦ 最终将序列化出来的字节数组的长度填充至报文体长度位置
108     buffer.writerIndex(savedWriteIndex + HEADER_LENGTH + len);
109 } catch (Throwable t) {
110     // 下面的逻辑都是异常处理，主要想办法赋值 status、errorMessage 字段
111     // 将错误信息体现出来，告诉消费方
112     // clear buffer
113     buffer.writerIndex(savedWriteIndex);
114     // send error message to Consumer, otherwise, Consumer will wait till t
115     if (!res.isEvent() && res.getStatus() != Response.BAD_RESPONSE) {
116         Response r = new Response(res.getId(), res.getVersion());
117         r.setStatus(Response.BAD_RESPONSE);
118         if (t instanceof ExceedPayloadLimitException) {
119             logger.warn(t.getMessage(), t);
120             try {
121                 r.setErrorMessage(t.getMessage());
122                 channel.send(r);
123                 return;
124             } catch (RemotingException e) {
125                 logger.warn("Failed to send bad_response info back: " + t.g
126             }
127         } else {
128             // FIXME log error message in Codec and handle in caught() of I
129             logger.warn("Fail to encode response: " + res + ", send bad_res
130             try {
131                 r.setErrorMessage("Failed to send response: " + res + ", ca
132                 channel.send(r);
133                 return;
134             } catch (RemotingException e) {
135                 logger.warn("Failed to send bad_response info back: " + res
136             }
137         }
138     }
139
140     // Rethrow exception
141     if (t instanceof IOException) {
142         throw (IOException) t;
143     } else if (t instanceof RuntimeException) {


```

```
144         throw (RuntimeException) t;
145     } else if (t instanceof Error) {
146         throw (Error) t;
147     } else {
148         throw new RuntimeException(t.getMessage(), t);
149     }
150
151
```

分析后，我们发现也没什么特别的，总结下来有 4 个区别。

- 多了一个响应码的设置，即帧格式中的第 4 个字节内容的设置。
- 请求唯一 ID 的设置，用的是接收请求时 ID 值。
- 若响应码不是 OK，就直接将 `errorMessage` 刷到了响应体数据中。
- 多了比较丰富的异常处理，并且针对一些特殊的异常，会将异常信息发送回消费方，以便消费方感知到提供方到底是出现了什么异常。

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 3  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 22 | 协议编解码：接口调用的数据是如何发到网络中的？

下一篇 24 | 拦截扩展：如何利用 Filter 进行扩展？

领资料

# 李三红·搞定 Java 开发基础

极客时间 × 阿里云开发者社区联合出品

李三红  
阿里云程序语言  
与编译器技术总监  
Java Champion



免费订阅 

## 精选留言 (1)

 写留言



杨老师

2023-03-09 来自北京

doTransferInvoke(invocation);

老师，这块调用转发服务的逻辑，大致思路是啥样的？

可以运用之前学到的点点直连吗？

作者回复: 你好，杨老师：可以运用之前的点点直连。大致思路就是：

- 1、从 invocation 将原始的请求数据提取出来，attachment（技术属性）+ 业务属性；
- 2、通过 http、tcp 将提取出来的数据发送至可用服务器（比如：隔壁机房）；
- 3、将 http、tcp 的返回数据又继续想办法封为 Result 对象返回即可；
- 4、若 1~3步还是报错拿不到结果，那作为备选方案都无法在通信层面建立连接的话，那就走到了最坏的情况，该抛异常的还是得抛异常。

领资料

