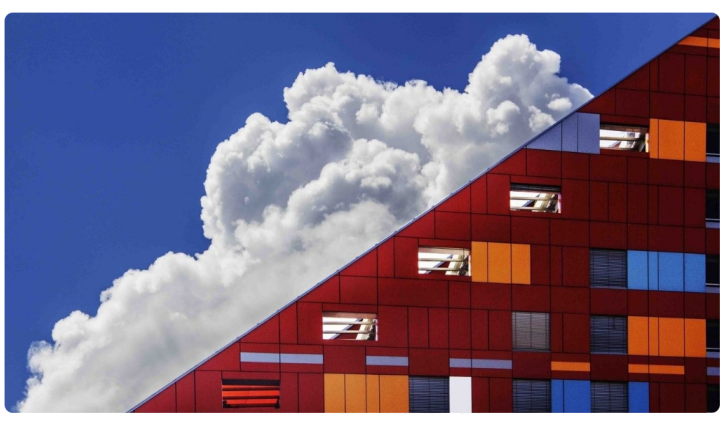
97 | 项目实战三：设计实现一个支持自定义规则的灰度发布组件（设计）

王争 2020-06-15



上一节课，我们介绍了灰度组件的一个需求场景，将公共服务平台的 RPC 接口，灰度替换为新的 RESTful 接口，通过灰度逐步放量，支持快速回滚等手段，来规避代码质量问题带来的不确定性风险。

跟前面两个框架类似，灰度组件的功能性需求也比较简单。上一节课我们做了简单分析，今天我们再介绍一下，这个组件的非功能性需求，以及如何通过合理的设计来满足这些非功能性需求。

话不多说，让我们正式开始今天的学习吧！

非功能性需求

上一节课中，我给你留了一个作业，参照限流框架，分析一下灰度组件的非功能性需求。对于限流框架，我们主要从易用性、扩展性、灵活性、性能、容错性这几个方面，来分析它的非功能性需求。对于灰度组件，我们同样也从这几个方面来分析。

易用性

在前面讲到限流框架和幂等框架的时候，我们都提到了“低侵入松耦合”的设计思想。因为框架需要集成到业务系统中使用，我们希望它尽可能低侵入，与业务代码松耦合，替换、移除起来更容易些。因为接口的限流和幂等跟具体的业务是无关的，我们可以把限流和幂等相关的逻辑，跟业务代码解耦，统一放到公共的地方来处理（比如 Spring AOP 切面中）。

但是，对于灰度来说，我们实现的灰度功能是代码级别的细粒度的灰度，而替代掉原来的 if-else 逻辑，是针对一个业务一个业务来做的，跟业务强相关，要做到跟业务代码完全解耦，是不现实的。所以，在侵入性这一点上，灰度组件只能做妥协，容忍一定程度的侵入。

除此之外，在灰度的过程中，我们要不停地修改灰度规则，在测试没有出现问题的情况下，逐渐放量。从运维的角度来说，如果每次修改灰度规则都要重启系统，显然是比较麻烦的。所以，我们希望支持灰度规则的热更新，也就是说，当我们在配置文件中，修改了灰度规则之后，系统在不重启的情况下会自动加载、更新灰度规则。

扩展性、灵活性

跟限流框架一样，我们希望支持不同格式（JSON、YAML、XML 等）、不同存储方式（本地配置文件、Redis、Zookeeper、或者自研配置中心等）的灰度规则配置方式。这一点在限流框架中已经详细讲过了，在灰度组件中我们就不重复讲解了。

除此之外，对于灰度规则本身，在上一节课的示例中，我们定义了三种灰度规则语法格式：具体值（比如 893）、区间值（比如 1020-1120）、比例值（比如 %30）。不过，这只能处理比较简单的灰度规则。如果我们要支持更加复杂的灰度规则，比如只对 30 天内购买过某某商品并且退货次数少于 10 次的用户进行灰度，现在的灰度规则语法就无法支持了。所以，如何支持更加灵活的、复杂的灰度规则，也是我们设计实现的重点和难点。

性能

在性能方面，灰度组件的处理难度，并不像限流框架那么高。在限流框架中，对于分布式限流模式，接口请求访问计数存储在中心存储器中，比如 Redis。而 Redis 本身的读写性能以及限流框架与 Redis 的通信延迟，都会很大地影响到限流本身的性能，进而影响到接口响应时间。所以，对于分布式限流来说，低延迟高性能是设计实现的难点和重点。

但是，对于灰度组件来说，灰度的判断逻辑非常简单，而且不涉及访问外部存储，所以性能一般不会有太大问题。不过，我们仍然需要把灰度规则组织成快速查找的数据结构，能够支持快速判定某个灰度对象（darkTarget，比如用户 ID）是否落在灰度规则设定的区间内。

容错性

在限流框架中，我们要求高度容错，不能因为框架本身的异常，导致接口响应异常。从业务上来讲，我们一般能容忍限流框架的暂时、小规模的失效，所以，限流框架对于异常的处理原则是，尽可能捕获所有异常，并且内部“消化”掉，不要往上层业务代码中抛出。

对于幂等框架来说，我们不能容忍框架暂时、小规模的失效，因为这种失效会导致业务有可能多次被执行，发生业务数据的错误。所以，幂等框架对于异常的处理原则是，按照 fail-fast 原则，如果异常导致幂等逻辑无法正常执行，让业务代码也中止。因为业务执行失败，比业务执行出错，修复的成本更低。

对于灰度组件来说，上面的两种对异常的处理思路都是可以接受的。在灰度组件出现异常时，我们既可以选择中止业务，也可以选择让业务继续执行。如果让业务继续执行，本不应该被灰度到的业务对象，就有可能被执行。这是否能接受，还是要看具体的业务。不过，我个人倾向于采用类似幂等框架的处理思路，在出现异常时中止业务。

框架设计思路

根据刚刚对灰度组件的非功能性需求分析，以及跟限流框架、幂等框架非功能性需求的对比，我们可以看出，在性能和容错性方面，灰度组件并没有需要特别要处理的地方，重点需要关注的是易用性、扩展性、灵活性。详细来说，主要包括这样两点：支持更灵活、更复杂的灰度规则和支持灰度规则热更新。接下来，我们就重点讲一下，针对这两个重点问题的设计思路。

首先，我们来看，如何支持更灵活、更复杂的灰度规则。

灰度规则的配置也是跟业务强相关的。业务方需要根据要灰度的业务特点，找到灰度对象（上节课中的 darkTarget，比如用户 ID），然后按照给出的灰度规则语法格式，配置相应的灰度规则。

对于像刚刚提到的那种复杂的灰度规则（只对 30 天内购买过某某商品并且退货次数少于 10 次的用户进行灰度），通过定义语法规则来支持，是很难实现的。所以，针对复杂灰度规则，我们换个思路来实现。

我暂时想到了两种解决方法。其中一种是使用规则引擎，比如 Drools，可以在配置文件中调用 Java 代码。另一种是支持编程实现灰度规则，这样做灵活性更高。不过，缺点是更新灰度规则需要更新代码，重新部署。

对于大部分业务的灰度，我们使用前面定义的最基本的语法规则（具体值、区间值、比例值）就能满足了。对于极个别复杂的灰度规则，我们借鉴 Spring 的编程式配置，由业务方编程实现，具体如何来做，我们放到下一节课的代码实现中讲解。这样既兼顾了易用性，又兼顾了灵活性。

之所以选择第二种实现方式，而不是使用 Drools 规则引擎，主要是出于不想为了不常用的功能，引入复杂的第三方框架，提高开发成本和灰度框架本身的学习成本。

其次，我们来看，如何实现灰度规则热更新。

规则热更新这样一个功能，并非灰度组件特有的，很多场景下都有类似的需求。在第 25、26 讲中，讲到性能计数器项目的时候，我们也提到过这个需求。

灰度规则的热更新实现起来并不难。我们创建一个定时器，每隔固定时间（比如 1 分钟），从配置文件中，读取灰度规则配置信息，并且解析加载到内存中，替换掉老的灰度规则。需要特别强调的是，更新灰度规则，涉及读取配置、解析、构建等一系列操作，会花费比较长的时间，我们不能因为更新规则，就暂停了灰度服务。所以，在设计和实现灰度规则更新的时候，我们要支持更新和查询并发执行。具体如何来做，我们留在下一节课的实现中详细讲解。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天，我们对比限流框架、幂等框架，讲解了灰度组件的非功能性需求，主要包含易用性、扩展性、灵活性、性能、容错性这样几个方面，并且针对性地解释了对应的设计思路。

在易用性方面，我们重点讲解了“低侵入、松耦合”的设计思想。限流、幂等因为其跟业务无关，可以做到最大程度跟业务解耦，做到低侵入。但是，我们这里实现的代码层面的灰度，因为跟业务强相关，所以，跟业务代码耦合的比较紧密，比较难做到低侵入。

在扩展性、灵活性方面，除了像限流框架那样，支持各种格式、存储方式的配置方式之外，灰度组件还希望能支持复杂的灰度规则。对于大部分业务的灰度，我们使用最基本的语法规则（具体值、区间值、比例值）就能满足了。对于极个别复杂的灰度规则，我们借鉴 Spring 的编程式配置，由业务方编程实现。

在性能方面，灰度组件没有需要特殊处理的地方。我们只需要把灰度规则组织成快速查找的数据结构，能够支持快速判定某个灰度对象（darkTarget，比如用户 ID），是否落在灰度规则设定的区间内。

在容错性方面，限流框架要高度容错，容忍短暂、小规模的限流失效，但不容忍框架异常导致的接口响应异常。幂等框架正好相反，不容忍幂等功能的失效，一旦出现异常，幂等功能失效，我们的处理原则是让业务也失败。这两种处理思路都可以用在灰度组件对异常的处理中。我个人倾向于采用幂等框架的处理思路。

课堂讨论

在项目实战这部分中，我们多次讲到“低侵入、松耦合”的设计思路，我们平时使用 Logger 框架，在业务代码中打印日志，算不算是对业务代码的侵入、耦合？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。