96-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（分析）

到现在为止，我已经带你学习了接口限流框架和接口幂等框架两个实战项目。接下来，我再带你实战一个新的项目：灰度发布组件。这也是我们专栏的最后一个实战项目。还是老套路，我们把它分为分析、设计、实现三个部分、对应三节课来讲解。今天，我们对灰度发布组件进行需求分析，搞清楚这个组件应该具有哪些功能性和非功能性需求。

话不多说，让我们正式开始今天的学习吧！

需求场景

还记得我们之前接口限流和幂等框架的项目背景吗？我们开发了一个公共服务平台，提供公共业务功能，给其他产品的后端系统调用，避免重复开发相同的业务代码。

最初，公共服务平台提供的是，基于某个开源RPC框架的RPC格式的接口。在上线一段时间后，我们发现这个开源RPC框架的Bug很多，多次因为框架本身的Bug，导致整个公共服务平台的接口不可用，但又因为团队成员对框架源码不熟悉，并且框架的代码质量本身也不高，排查、修复起来花费了很长时间，影响面非常大。所以，我们评估下来，觉着这个框架的可靠性不够，维护成本、二次开发成本都太高，最终决定替换掉它。

对于引入新的框架，我们的要求是成熟、简单，并且与我们现有的技术栈（Spring）相吻合。这样，即便出了问题，我们也能利用之前积累的知识、经验来快速解决。所以，我们决定直接使用Spring框架来提供RESTful格式的远程接口。

把RPC接口替换成RESTful接口，除了需要修改公共服务平台的代码之外，调用方的接口调用代码也要做相应的修改。除此之外，对于公共服务平台的代码，尽管我们只是改动接口暴露方式，队业务代码基本上没有改动，但是，我们也并不能保证就完全不出问题。所以，为了保险起见，我们希望灰度替换掉老的RPC服务，而不是一刀切，在某个时间点上，让所有的调用方一下子都变成调用新的Resful接口。

我们来看下具体如何来做。

因为替换的过程是灰度的，所以老的RPC服务不能下线，同时还要部署另外一套新的RESTful服务。我们先让业务不是很重要、流量不大的某个调用方，替换成调用新的RESTful接口。经过这个调用方一段时间的验证之后，如果新的RESTful接口没有问题，我们再逐步让其他调用方，替换成调用新的RESTful接口。

但是，如果万一中途出现问题，我们就需要将调用方的代码回滚，再重新部署，这就会导致调用方一段时间内服务不可用。而且，如果新的代码还包含调用方自身新的业务代码，简单通过Git回滚代码重新部署，会导致新的业务代码也被回滚。所以，为了避免这种情况的发生，我们就得手动将调用新的RESTful接口的代码删除，再改回为调用老的RPC接口。

除此之外，为了不影响调用方本身业务的开发进度，调用方基于回滚之后的老代码，来做新功能开发，那替换成新的RPC接口的那部分代码，要想再重新merge回去就比较难了，有可能会出现代码冲突，需要再重新开发。

怎么解决代码回滚成本比较高的问题呢？你可以先思考一下，再看我的讲解。

在替换新的接口调用方式的时候，调用方并不直接将调用RPC接口的代码逻辑删除，而是新增调用RESTful

接口的代码，通过一个功能开关，灵活切换走老的代码逻辑还是新的代码逻辑。代码示例如下所示。如果callRestfulApi为true，就会走新的代码逻辑，调用RESTful接口，相反，就会走老的代码逻辑，继续调用RPC接口。

```
boolean callRestfulApi = true;

if (!callRestfulApi) {
  // 老的调用RPC接口的代码逻辑
} else {
  // 新的调用Resful接口的代码逻辑
}
```

不过，更改callRestfulApi的值需要修改代码，而修改代码就要重新部署，这样的设计还是不够灵活。优化的方法，我想你应该已经想到了，把这个值放到配置文件或者配置中心就可以了。

为了更加保险，不只是使用功能开关做新老接口调用方式的切换，我们还希望调用方在替换某个接口的时候，先让小部分接口请求，调用新的RESTful接口，剩下的大部分接口请求，还是调用老的RPC接口，验证没有问题之后，再逐步加大调用新接口的请求比例，最终，将所有的接口请求，都替换成调用新的接口。这就是所谓的“灰度”。

那这个灰度功能又该如何实现呢？同样，你还是先思考一下，再来看我的讲解。

首先，我们要决定使用什么来做灰度，也就是灰度的对象。我们可以针对请求携带的时间戳信息、业务ID等信息，按照区间、比例或者具体的值来做灰度。我举个例子来解释一下。

假设，我们要灰度的是根据用户ID查询用户信息接口。接口请求会携带用户ID信息，所以，我们就可以把用户ID作为灰度的对象。为了实现逐渐放量，我们先配置用户ID是918、879、123（具体的值）的查询请求调用新接口，验证没有问题之后，我们再扩大范围，让用户ID在1020~1120（区间值）之间的查询请求调用新接口。

如果验证之后还是没有问题，我们再继续扩大范围，让10%比例（比例值）的查询请求调用新接口（对应用户ID跟10取模求余小于1的请求）。以此类推，灰度范围逐步扩大到20%、30%、50%直到100%。当灰度比例达到100%，并且运行一段时间没有问题之后，调用方就可以把老的代码逻辑删除掉了。

实际上，类似的灰度需求场景还有很多。比如，在金融产品的清结算系统中，我们修改了清结算的算法。为了安全起见，我们可以灰度替换新的算法，把贷款ID作为灰度对象，先对某几个贷款应用新的算法，如果没有问题，再继续按照区间或者比例，扩大灰度范围。

除此之外，为了保证代码万无一失，提前做好预案，添加或者修改一些复杂功能、核心功能，即便不做灰度，我们也建议通过功能开关，灵活控制这些功能的上下线。在不需要重新部署和重启系统的情况，做到快速回滚或新老代码逻辑的切换。

需求分析

从实现的角度来讲，调用方只需要把灰度规则和功能开关，按照某种事先约定好的格式，存储到配置文件或者配置中心，在系统启动的时候，从中读取配置到内存中，之后，看灰度对象是否落在灰度范围内，以此来

判定是否执行新的代码逻辑。但为了避免每个调用方都重复开发，我们把功能开关和灰度相关的代码，抽象封装为一个灰度组件，提供给各个调用方来复用。

这里需要强调一点，我们这里的灰度，是代码级别的灰度，目的是保证项目质量，规避重大代码修改带来的不确定性风险。实际上，我们平时经常讲的灰度，一般都是产品层面或者系统层面的灰度。

所谓产品层面，有点类似A/B Testing，让不同的用户看到不同的功能，对比两组用户的使用体验，收集数据，改进产品。所谓系统层面的灰度，往往不在代码层面上实现，一般是通过配置负载均衡或者API-Gateway，来实现分配流量到不同版本的系统上。系统层面的灰度也是为了平滑上线功能，但比起我们讲到的代码层面的灰度，就没有那么细粒度了，开发和运维成本也相对要高些。

现在，我们就来具体看下，灰度组件都有哪些功能性需求。

我们还是从使用的角度来分析。组件使用者需要设置一个key值，来唯一标识要灰度的功能，然后根据自己业务数据的特点，选择一个灰度对象（比如用户ID），在配置文件或者配置中心中，配置这个key对应的灰度规则和功能开关。配置的格式类似下面这个样子：

```
dark:
  --key: call_newapi_getUserById
   enabled: true // enabled为true时，rule才生效
   rule: {893,342,1020-1120,%30} // 按照用户ID来做灰度
  --key: call_newapi_registerUser
   enabled: true
   rule: {1391198723, %10}  //按照手机号来做灰度
  --key: newalgo_loan
   enabled: true
   rule: {0-1000} //按照贷款(loan)的金额来做灰度
```

灰度组件在业务系统启动的时候，会将这个灰度配置，按照事先定义的语法，解析并加载到内存对象中，业务系统直接使用组件提供的灰度判定接口，给业务系统使用，来判定某个灰度对象是否灰度执行新的代码逻辑。配置的加载解析、灰度判定逻辑这部分代码，都不需要业务系统来从零开发。

所以，总结一下的话，灰度组件跟限流框架很类似，它也主要包含两部分功能：灰度规则配置解析和提供编程接口（DarkFeature）判定是否灰度。

跟限流框架类似，除了功能性需求，我们还要分析非功能性需求。不过，因为前面已经有了限流框架的非功能性需求的讲解，对于灰度组件的非功能性需求，我就留给你自己来分析。在下一节课中，我会再给出我的分析思路，到时候，你可以对比一下。

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。

灰度发布可以分为三个不同层面的灰度：产品层面的灰度、系统层面的灰度和代码层面的灰度。我们今天重点讲解代码层面的灰度，通过编程来控制是否执行新的代码逻辑，以及灰度执行新的代码逻辑。

代码层面的灰度，主要解决代码质量问题，通过逐渐放量灰度执行，来降低重大代码改动带来的风险。在出

现问题之后，在不需要修改代码、重新部署、重启系统的情况下，实现快速地回滚。相对于系统层面的灰度，它可以做得更加细粒度，更加灵活、简单、好维护，但也存在着代码侵入的问题，灰度代码跟业务代码耦合在一起。

灰度组件跟之前讲过的限流框架很相似，主要包含配置的解析加载和灰度判定逻辑。除此之外，对于非功能性需求，我们留在下一节课中讲解。

课堂讨论

参照限流框架的非功能性需求，分析一下灰度组件的非功能性需求。

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。

精选留言：

- Jxin 2020-06-12 01:46:50
  1.灰度代码不该和业务代码耦合在一起。
  2.以替换rpc实现为例。为每个rpc接口都创建防腐层，业务代码依赖防腐层的接口来写逻辑。这样rpc实例的选择和使用就解耦开了。
  a.在灰度发布的场景，可以在防腐层加这个功能开关，既与业务代码分离，且一处变更全局生效。
  b.接口隔离，服务方提供的api一般功能都比较多，出入参也会比较臃肿。有防腐层，调用方就可以按需设计防腐层接口，用适配的方式隔离掉服务提供方接口的复杂性。
  c.开发隔离，不用再等到服务提供方提供接口才能开发了。整个开发过程基于自己的防腐层写代码，写完用mock方式自测。项目经理再也不用担心我的进度被阻塞。

  3.单个函数内部的代码逻辑做灰度。这个看情况，原业务逻辑很简单没几行代码，就耦合呗，问题不大。如果原逻辑比较复杂。那么就可能得抽局部功能的中间函数咯。使用侧依赖中间函数，中间函数做灰度切换，新老具体功能各自封装成函数。如此一来，与依赖接口编程异曲同工。隔离复杂性，一处改动全局生效。
  [2赞]

- 小晏子 2020-06-12 09:21:36
  在易用性方面，框架接入要简单方便，学习成本低，尽量减少与业务代码的耦合，最
  比如能以自动注入的方式提供开关配置。
  在性能方面，因为每次请求要获取开关配置信息，所以要让灰度框架尽可能低延迟，尽可能减少对请求本身响应时间的影响。
  在容错性方面，要保证不会因为灰度框架本身的异常引起整个请求异常，影响业务可用性，当灰度框架有异常的时候，请求要能回滚到原来的请求方式。

- liu_liu 2020-06-12 08:42:33
  简单易用，对业务代码侵入尽可能小。框架异常时不影响业务代码。