**Spring Cloud构建微服务架构（七）消息总线**

**原创**

[2016-09-27](http://blog.didispace.com/springcloud7/)

 翟永超

[Spring Cloud](http://blog.didispace.com/categories/Spring-Cloud/)

被围观 **28711** 次

先回顾一下，在之前的Spring Cloud Config的介绍中，我们还留了一个悬念：如何实现对配置信息的实时更新。虽然，我们已经能够通过/refresh接口和Git仓库的Web Hook来实现Git仓库中的内容修改触发应用程序的属性更新。但是，若所有触发操作均需要我们手工去维护Web Hook中的应用位置的话，这随着系统的不断扩张，会变的越来越难以维护，而消息代理中间件是解决该问题最为合适的方案。是否还记得我们在介绍消息代理中的特点时有提到过这样一个功能：消息代理中间件可以将消息路由到一个或多个目的地。利用这个功能，我们就能完美的解决该问题，下面我们来说说Spring Cloud Bus中的具体实现方案。

在[《Spring Boot中使用RabbitMQ》](http://blog.didispace.com/spring-boot-rabbitmq/)一文中，我们已经介绍了关于消息代理、AMQP协议以及RabbitMQ的基础知识和使用方法。下面我们开始具体介绍Spring Cloud Bus的配置，并以一个Spring Cloud Bus与Spring Cloud Config结合的例子来实现配置内容的实时更新。

**RabbitMQ实现**

下面我们来具体动手尝试整个配置过程：

* 准备工作：这里我们不做新的应用，但需要用到上一章中，我们已经实现的关于Spring Cloud Config的几个工程，若读者对其还不了解，建议先阅读第4章的内容。
  + config-repo：定义在Git仓库中的一个目录，其中存储了应用名为didispace的多环境配置文件，配置文件中有一个from参数。
  + config-server-eureka：配置了Git仓库，并注册到了Eureka的服务端。
  + config-client-eureka：通过Eureka发现Config Server的客户端，应用名为didispace，用来访问配置服务器以获取配置信息。该应用中提供了一个/from接口，它会获取config-repo/didispace-dev.properties中的from属性返回。
* 扩展config-client-eureka应用
  + 修改pom.xml增加spring-cloud-starter-bus-amqp模块（注意spring-boot-starter-actuator模块也是必须的）。

|  |  |
| --- | --- |
| 1  2  3  4 | <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency> |

* 在配置文件中增加关于RabbitMQ的连接和用户信息

|  |  |
| --- | --- |
| 1  2  3  4 | spring.rabbitmq.host=localhost  spring.rabbitmq.port=5672  spring.rabbitmq.username=springcloud  spring.rabbitmq.password=123456 |

* 启动config-server-eureka，再启动两个config-client-eureka（分别在不同的端口上，比如7002、7003），我们可以在config-client-eureka中的控制台中看到如下内容，在启动时候，客户端程序多了一个/bus/refresh请求。

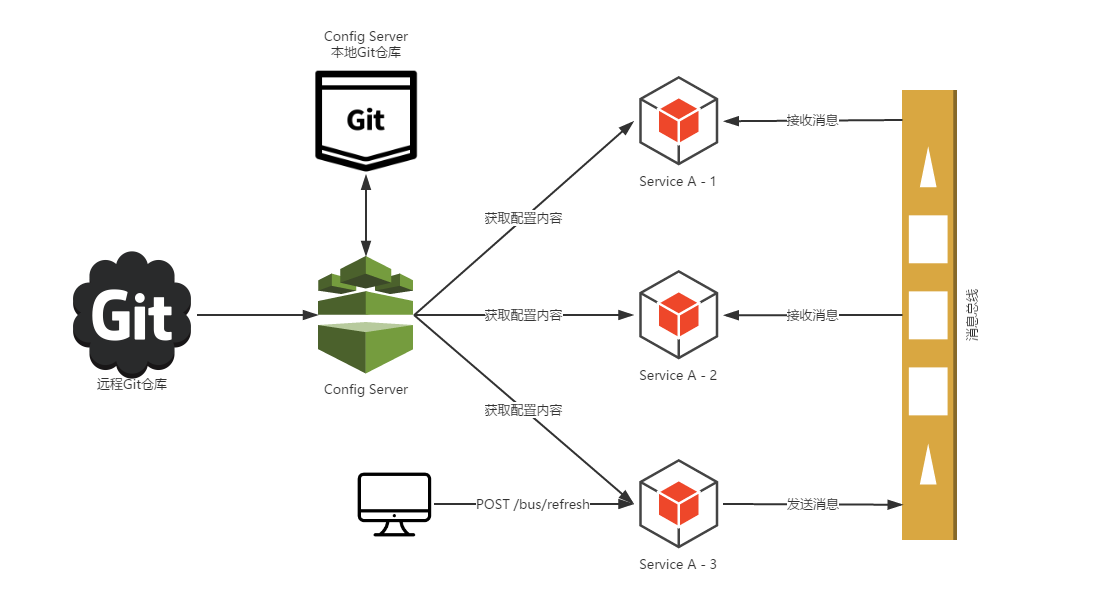
|  |  |
| --- | --- |
| 1 | o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "{[/bus/refresh],methods=[POST]}" onto public void org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String) |

* 先访问两个config-client-eureka的/from请求，会返回当前config-repo/didispace-dev.properties中的from属性。
* 接着，我们修改config-repo/didispace-dev.properties中的from属性值，并发送POST请求到其中的一个/bus/refresh。
* 最后，我们再分别访问启动的两个config-client-eureka的/from请求，此时这两个请求都会返回最新的config-repo/didispace-dev.properties中的from属性。

到这里，我们已经能够通过Spring Cloud Bus来实时更新总线上的属性配置了。

**原理分析**

我们通过使用Spring Cloud Bus与Spring Cloud Config的整合，并以RabbitMQ作为消息代理，实现了应用配置的动态更新。

[](http://blog.didispace.com/assets/5-6.png)

整个方案的架构如上图所示，其中包含了Git仓库、Config Server、以及微服务“Service A”的三个实例，这三个实例中都引入了Spring Cloud Bus，所以他们都连接到了RabbitMQ的消息总线上。

当我们将系统启动起来之后，“Service A”的三个实例会请求Config Server以获取配置信息，Config Server根据应用配置的规则从Git仓库中获取配置信息并返回。

此时，若我们需要修改“Service A”的属性。首先，通过Git管理工具去仓库中修改对应的属性值，但是这个修改并不会触发“Service A”实例的属性更新。我们向“Service A”的实例3发送POST请求，访问/bus/refresh接口。此时，“Service A”的实例3就会将刷新请求发送到消息总线中，该消息事件会被“Service A”的实例1和实例2从总线中获取到，并重新从Config Server中获取他们的配置信息，从而实现配置信息的动态更新。

而从Git仓库中配置的修改到发起/bus/refresh的POST请求这一步可以通过Git仓库的Web Hook来自动触发。由于所有连接到消息总线上的应用都会接受到更新请求，所以在Web Hook中就不需要维护所有节点内容来进行更新，从而解决了通过Web Hook来逐个进行刷新的问题。

**指定刷新范围**

上面的例子中，我们通过向服务实例请求Spring Cloud Bus的/bus/refresh接口，从而触发总线上其他服务实例的/refresh。但是有些特殊场景下（比如：灰度发布），我们希望可以刷新微服务中某个具体实例的配置。

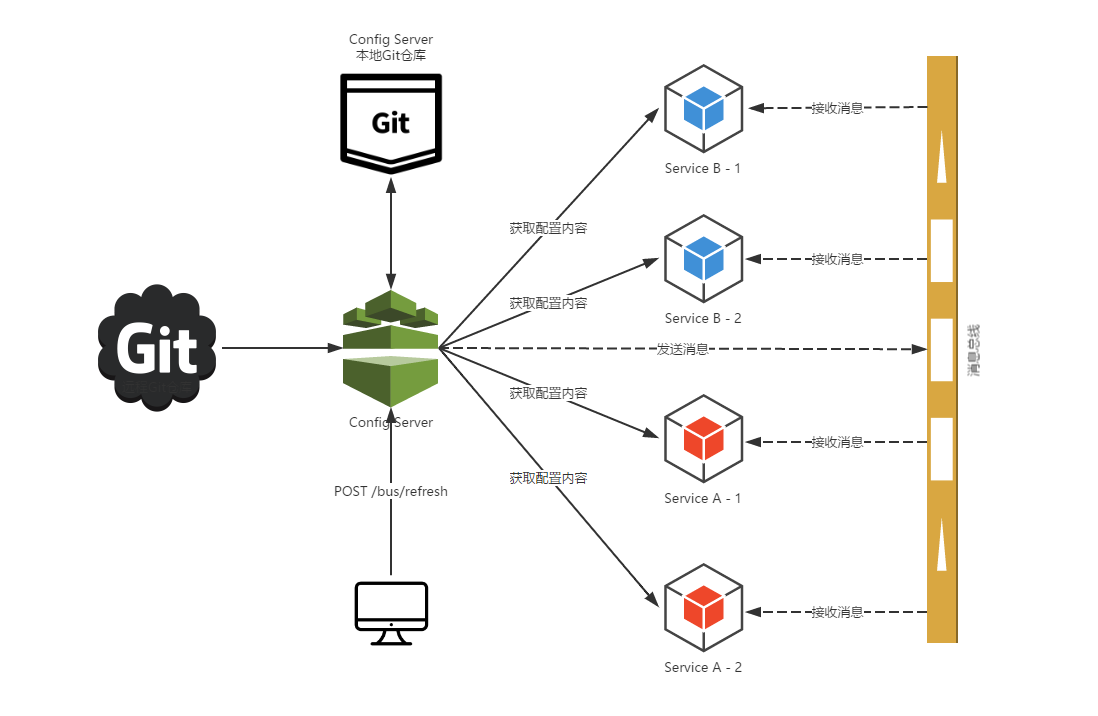
Spring Cloud Bus对这种场景也有很好的支持：/bus/refresh接口还提供了destination参数，用来定位具体要刷新的应用程序。比如，我们可以请求/bus/refresh?destination=customers:9000，此时总线上的各应用实例会根据destination属性的值来判断是否为自己的实例名，若符合才进行配置刷新，若不符合就忽略该消息。

destination参数除了可以定位具体的实例之外，还可以用来定位具体的服务。定位服务的原理是通过使用Spring的PathMatecher（路径匹配）来实现，比如：/bus/refresh?destination=customers:\*\*，该请求会触发customers服务的所有实例进行刷新。

**架构优化**

既然Spring Cloud Bus的/bus/refresh接口提供了针对服务和实例进行配置更新的参数，那么我们的架构也相应的可以做出一些调整。在之前的架构中，服务的配置更新需要通过向具体服务中的某个实例发送请求，再触发对整个服务集群的配置更新。虽然能实现功能，但是这样的结果是，我们指定的应用实例就会不同于集群中的其他应用实例，这样会增加集群内部的复杂度，不利于将来的运维工作，比如：我们需要对服务实例进行迁移，那么我们不得不修改Web Hook中的配置等。所以我们要尽可能的让服务集群中的各个节点是对等的。

因此，我们将之前的架构做了一些调整，如下图所示：

[](http://blog.didispace.com/assets/5-7.png)

我们主要做了这些改动：

1. 在Config Server中也引入Spring Cloud Bus，将配置服务端也加入到消息总线中来。
2. /bus/refresh请求不在发送到具体服务实例上，而是发送给Config Server，并通过destination参数来指定需要更新配置的服务或实例。

通过上面的改动，我们的服务实例就不需要再承担触发配置更新的职责。同时，对于Git的触发等配置都只需要针对Config Server即可，从而简化了集群上的一些维护工作。