Spring Cloud Config：外部集中化配置管理

Spring Cloud Config 可以为微服务架构中的应用提供集中化的外部配置支持，它分为服务端和客户端两个部分，下面将对其用法进行详细介绍。

Spring Cloud Config 简介

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。市面上开源的配置中心有很多，BAT每家都出过，360的QConf、淘宝的diamond、百度的disconf都是解决这类问题。国外也有很多开源的配置中心Apache的Apache Commons Configuration、owner、cfg4j等等。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是config server，二是config client。

Spring Cloud Config 分为服务端和客户端两个部分。服务端被称为分布式配置中心，它是个独立的应用，可以从配置仓库获取配置信息并提供给客户端使用。客户端可以通过配置中心来获取配置信息，在启动时加载配置。Spring Cloud Config 的配置中心默认采用Git来存储配置信息，所以天然就支持配置信息的版本管理，并且可以使用Git客户端来方便地管理和访问配置信息。

一个配置中心提供的核心功能

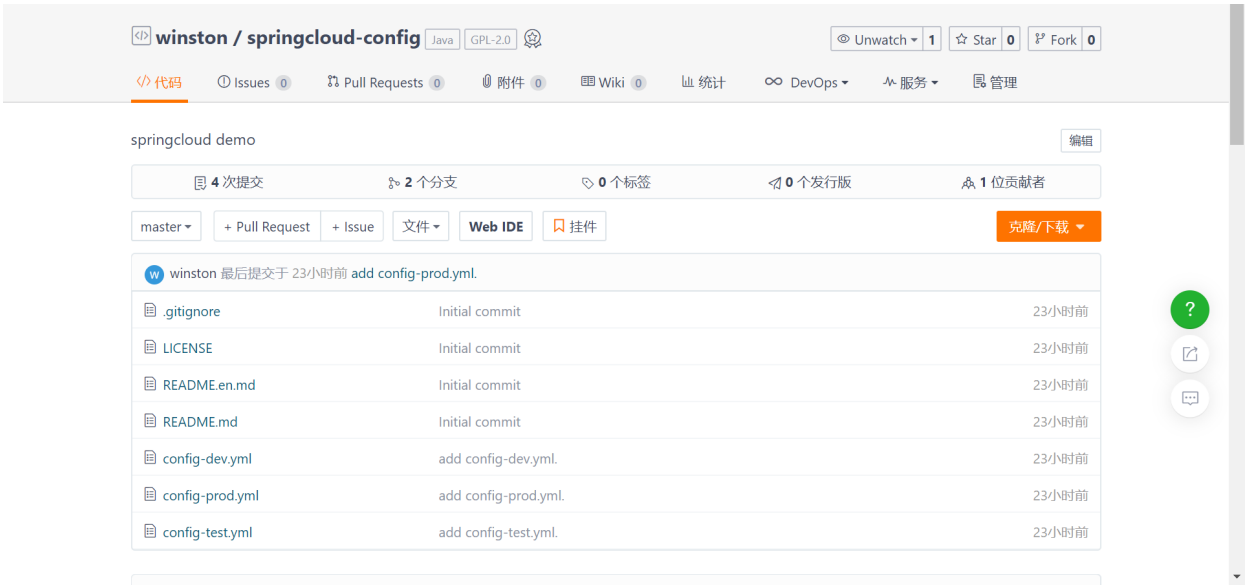
- 提供服务端和客户端支持
- 集中管理各环境的配置文件
- 配置文件修改之后，可以快速的生效
- 可以进行版本管理
- 支持大的并发查询
- 支持各种语言

在Git仓库中准备配置信息

由于Spring Cloud Config 需要一个存储配置信息的Git仓库，这里我们先在Git仓库中添加好配置文件再演示其功能，Git仓库地址为：

https://gitee.com/winstonqq/springcloud-config

配置仓库目录结构



master分支下的配置信息

- config-dev.yml:

```
config:
 info: "config info for dev(master)"
```

- config-test.yml:

```
config:
 info: "config info for test(master)"
```

- config-prod.yml:

```
config:
 info: "config info for prod(master)"
```

dev分支下的配置信息

- config-dev.yml:

```
config:
 info: "config info for dev(dev)"
```

- config-test.yml:

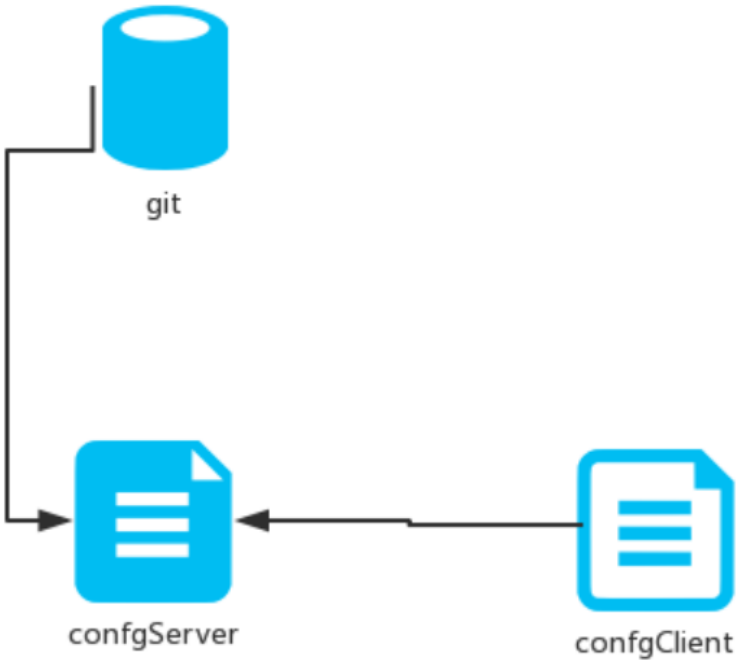
```
config:
 info: "config info for test(dev)"
```

- config-prod.yml:

```
config:
 info: "config info for prod(dev)"
```

创建config-server模块

我们创建一个config-server模块来演示Spring Cloud Config 作为配置中心的功能。



在pom.xml中添加相关依赖

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

在application.yml中进行配置

```
server:
 port: 8901
spring:
 application:
 name: config-server
 cloud:
 config:
```

```
  server:
  git: #配置存储配置信息的Git仓库
   uri: https://gitee.com/macrozheng/springcloud-config.git
   username: macro
   password: 123456
   clone-on-start: true #开启启动时直接从git获取配置
  # search-paths: '{application}'
  eureka:
   client:
   service-url:
   defaultZone: http://localhost:8001/eureka/
```

在启动类上添加@EnableConfigServer注解来启用配置中心功能

```
@EnableConfigServer
@EnableDiscoveryClient
@SpringBootApplication
public class ConfigServerApplication {

 public static void main(String[] args) {
 SpringApplication.run(ConfigServerApplication.class, args);
 }

}
```

通过config-server获取配置信息

这里我们通过config-server来演示下如何获取配置信息。

获取配置文件信息的访问格式

```
# 获取配置信息
/{label}/{application}-{profile}
# 获取配置文件信息
/{label}/{application}-{profile}.yml
```

占位符相关解释

- application：代表应用名称，默认为配置文件中的spring.application.name，如果配置了spring.cloud.config.name，则为该名称；
- label：代表分支名称，对应配置文件中的spring.cloud.config.label；

- profile：代表环境名称，对应配置文件中的spring.cloud.config.profile。

获取配置信息演示

- 启动eureka-server、config-server服务；
- 访问http://localhost:8901/master/config-dev来获取master分支上dev环境的配置信息；

- 访问http://localhost:8901/master/config-dev.yml来获取master分支上dev环境的配置文件信息，对比上面信息，可以看出配置信息和配置文件信息并不是同一个概念；

- 访问http://localhost:8901/master/config-test.yml来获取master分支上test环境的配置文件信息:

创建config-client模块

创建一个config-client模块来从config-server获取配置。

在pom.xml中添加相关依赖

```
1 <dependency>
```

```
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
```

在bootstrap.yml中进行配置

```
server:
 port: 9001
spring:
 application:
 name: config-client
 cloud:
 config: #Config客户端配置
 profile: dev #启用配置后缀名称
 label: dev #分支名称
  uri: http://localhost:8901 #配置中心地址
  name: config #配置文件名称
eureka:
 client:
 service-url:
 defaultZone: http://localhost:8001/eureka/
management:
 endpoints:
 web:
 exposure:
 include: 'refresh'
```

添加ConfigClientController类用于获取配置

```
@RestController
@RefreshScope
public class ConfigClientController {

```

```
5   @Value("${config.info}")
6   private String configInfo;
7
8   @GetMapping("/configInfo")
9   public String getConfigInfo() {
10   return configInfo;
11   }
12  }
```

演示从配置中心获取配置

- 启动config-client服务；
- 访问http://localhost:9001/configInfo，可以获取到dev分支下dev环境的配置；

```
1  config info for dev(dev)
```

获取子目录下的配置

我们不仅可以把每个项目的配置放在不同的Git仓库存储，也可以在一个Git仓库中存储多个项目的配置，此时就会用到在子目录中搜索配置信息的配置。

- 首先我们需要在config-server中添加相关配置，用于搜索子目录中的配置，这里我们用到了application占位符，表示对于不同的应用，我们从对应应用名称的子目录中搜索配置，比如config子目录中的配置对应config应用；

```
1  spring:
2   cloud:
3   config:
4   server:
5   git:
6   search-paths: '{application}'
```

- 重启config-client，访问http://localhost:9001/configInfo进行测试，可以发现获取的是config子目录下的配置信息。

```
1  config info for config dir dev(dev)
```

刷新配置

当Git仓库中的配置信息更改后，我们可以通过SpringBoot Actuator的refresh端点来刷新客户端配置信息，以下更改都需要在config-client中进行。

- 在pom.xml中添加Actuator的依赖：

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

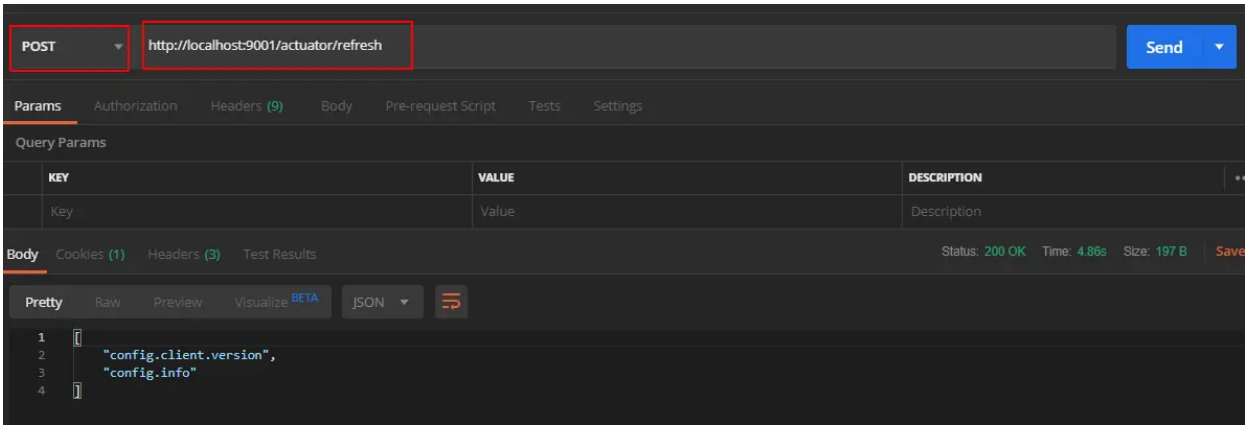
- 在bootstrap.yml中开启refresh端点：

```
management:
 endpoints:
 web:
 exposure:
 include: 'refresh'
```

- 在ConfigClientController类添加@RefreshScope注解用于刷新配置：

```
@RestController
@RefreshScope
public class ConfigClientController {

 @Value("${config.info}")
 private String configInfo;

 @GetMapping("/configInfo")
 public String getConfigInfo() {
  return configInfo;
  }
}
```

- 重新启动config-client后，调用refresh端点进行配置刷新：



- 访问http://localhost:9001/configInfo进行测试，可以发现配置信息已经刷新。

```
update config info for config dir dev(dev)
```

配置中心添加安全认证

通过整合SpringSecurity来为配置中心添加安全认证

创建config-security-server模块

- 在pom.xml中添加相关依赖：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- 在application.yml中进行配置：

```
server:
 port: 8905
spring:
 application:
 name: config-security-server
 cloud:
 config:
 server:
 git:
 uri: https://gitee.com/macrozheng/springcloud-config.git
 username: macro
 password: 123456
 clone-on-start: true #开启启动时直接从git获取配置
 security: #配置用户名和密码
 user:
 name: macro
 password: 123456
```

- 启动config-security-server服务。

修改config-client的配置

- 添加bootstrap-security.yml配置文件，主要是配置了配置中心的用户名和密码：

```
server:
 port: 9002
spring:
 application:
 name: config-client
 cloud:
 config:
 profile: dev #启用配置后缀名称
 label: dev #分支名称
  uri: http://localhost:8905 #配置中心地址
  name: config #配置文件名称
  username: zhuawa
  password: 123456
```

- 使用bootstrap-security.yml启动config-client服务；
- 访问http://localhost:9002/configInfo进行测试，发现可以获取到配置信息。

```
config info for dev(dev)
```

config-sever集群搭建

在微服务架构中，所有服务都从配置中心获取配置，配置中心一旦宕机，会发生很严重的问题，下面我们搭建一个双节点的配置中心集群来解决该问题。

- 启动两个config-server分别运行在8902和8903端口上；
- 添加config-client的配置文件bootstrap-cluster.yml，主要是添加了从注册中心获取配置中心地址的配置并去除了配置中心uri的配置：

```
server:
 port: 9003
```

```yaml
spring:
 application:
 name: config-client
 cloud:
 config:
 profile: dev #启用环境名称
 label: dev #分支名称
  name: config #配置文件名称
  discovery:
  enabled: true
  service-id: config-server
eureka:
  client:
  service-url:
  defaultZone: http://localhost:8001/eureka/
management:
  endpoints:
  web:
  exposure:
  include: 'refresh'
```

- 以bootstrap-cluster.yml启动config-client服务，注册中心显示信息如下：

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-CLIENT	n/a (1)	(1)	UP (1) - DESKTOP-K1F7O7Q:config-client:9003
CONFIG-SERVER	n/a (2)	(2)	UP (2) - DESKTOP-K1F7O7Q:config-server:8902 , DESKTOP-K1F7O7Q:config-server:8903

- 访问http://localhost:9003/configInfo，发现config-client可以获取到配置信息。

```
config info for config dir dev(dev)
```

Spring Cloud Bus：消息总线

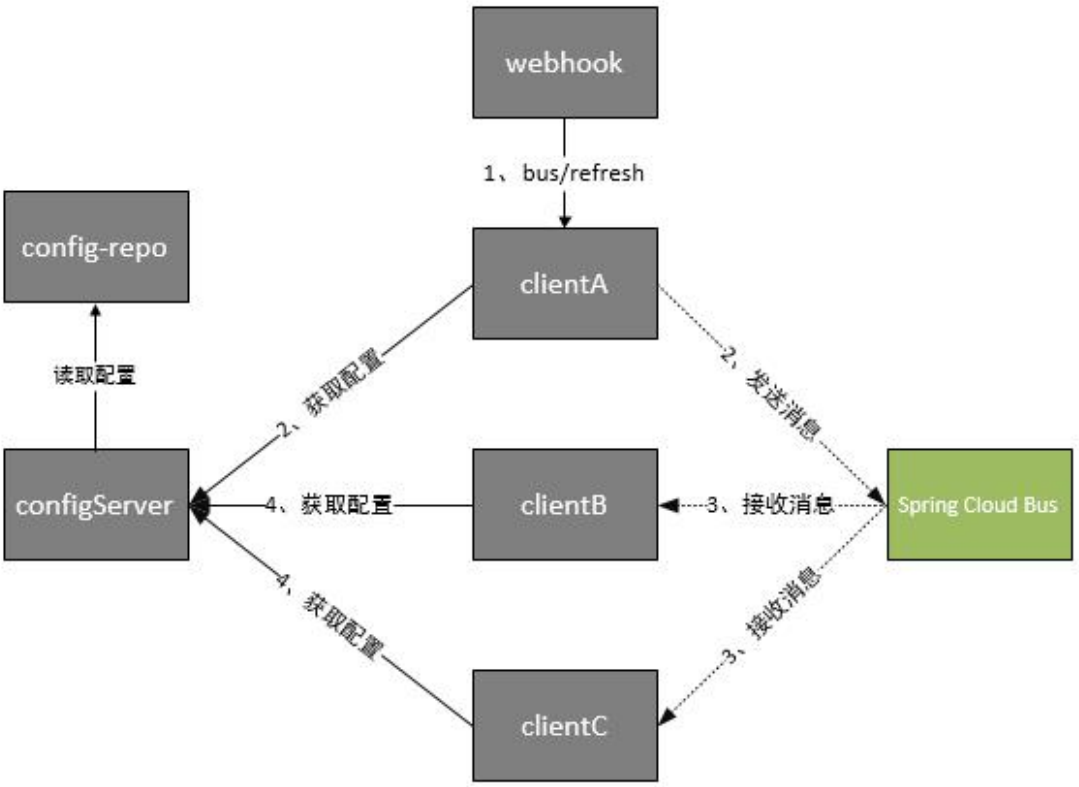
如果需要客户端获取到最新的配置信息需要执行refresh，我们可以利用webhook的机制每次提交代码发送请求来刷新客户端，当客户端越来越多的时候，需要每个客户端都执行一遍，这种方案就不太适合了。使用Spring Cloud Bus可以完美解决这一问题。

Spring Cloud Bus 使用轻量级的消息代理来连接微服务架构中的各个服务，可以将其用于广播状态更改（例如配置中心配置更改）或其他管理指令，下面将对其用法进行详细介绍。

Spring Cloud Bus 简介

Spring cloud bus通过轻量消息代理连接各个分布的节点。这会用在广播状态的变化（例如配置变化）或者其他的消息指令。Spring bus的一个核心思想是通过分布式的启动器对spring boot应用进行扩展，也可以用来建立一个多个应用之间的通信频道。目前唯一实现的方式是用AMQP消息代理作为通道，同样特性的设置（有些取决于通道的设置）在更多通道的文档中。

Spring cloud bus被国内很多都翻译为消息总线，也挺形象的。大家可以将它理解为管理和传播所有分布式项目中的消息既可，其实本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。利用bus的机制可以做很多的事情，其中配置中心客户端刷新就是典型的应用场景之一，我们用一张图来描述bus在配置中心使用的机制。



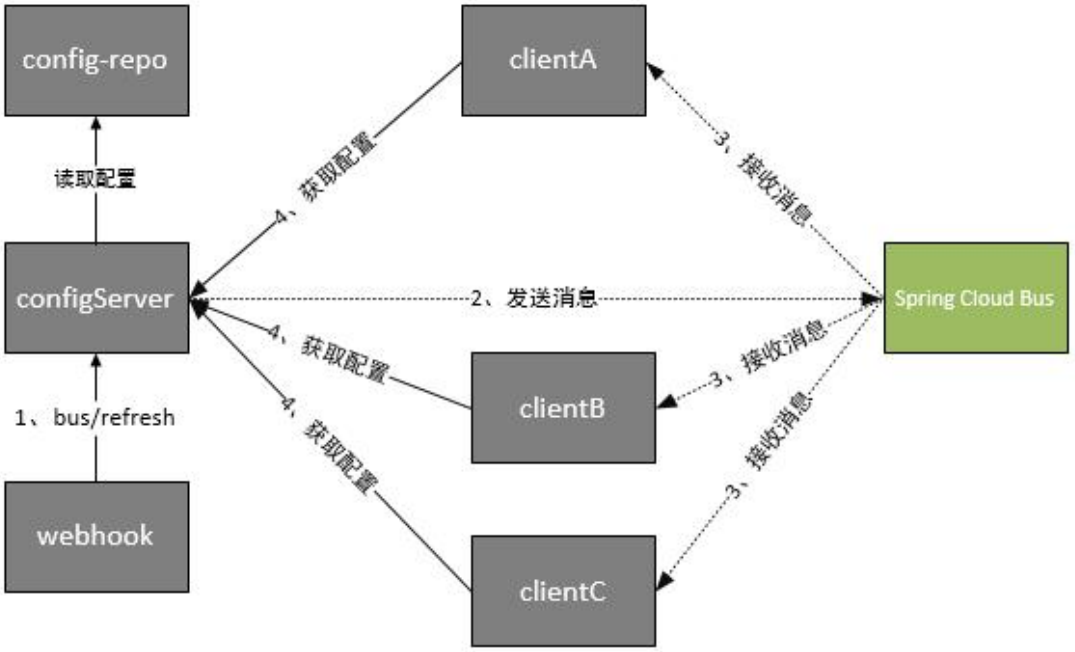
根据此图我们可以看出利用Spring Cloud Bus做配置更新的步骤:

- 1、提交代码触发post给客户端A发送bus/refresh
- 2、客户端A接收到请求从Server端更新配置并且发送给Spring Cloud Bus
- 3、Spring Cloud bus接到消息并通知给其它客户端
- 4、其它客户端接收到通知，请求Server端获取最新配置
- 5、全部客户端均获取到最新的配置

在上面的流程中，我们已经到达了利用消息总线触发一个客户端`bus/refresh`,而刷新所有客户端的配置的目的。但这种方式并不优雅。原因如下：

- 打破了微服务的职责单一性。微服务本身是业务模块，它本不应该承担配置刷新的职责。
- 破坏了微服务各节点的对等性。
- 有一定的局限性。例如，微服务在迁移时，它的网络地址常常会发生变化，此时如果想要做到自动刷新，那就不得不修改WebHook的配置。

因此我们将上面的架构模式稍微改变一下

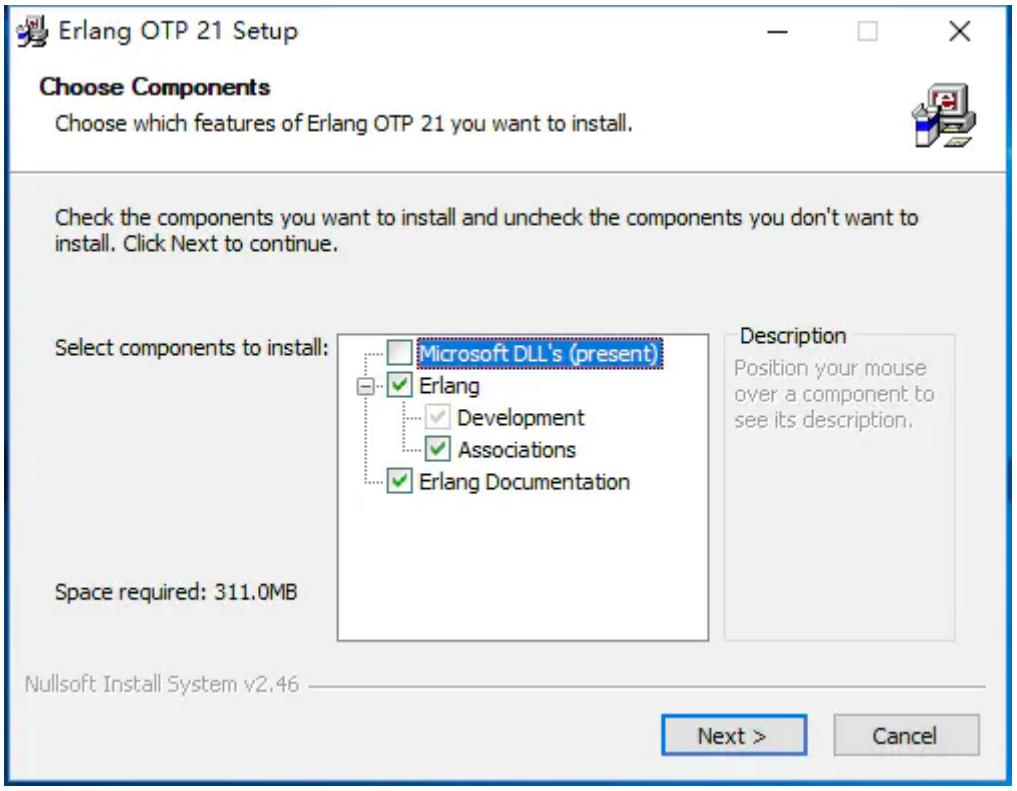


这时Spring Cloud Bus做配置更新步骤如下:

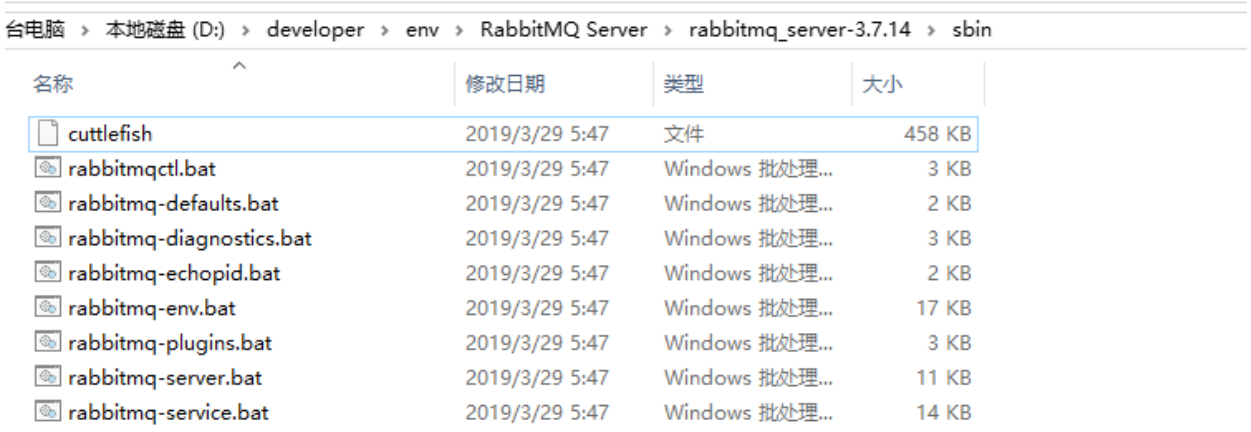
- 1、提交代码触发post请求给bus/refresh
- 2、server端接收到请求并发送给Spring Cloud Bus
- 3、Spring Cloud bus接到消息并通知给其它客户端
- 4、其它客户端接收到通知，请求Server端获取最新配置
- 5、全部客户端均获取到最新的配置

RabbitMQ的安装

- 安装Erlang，下载地址：
  http://erlang.org/download/otp_win64_21.3.exe

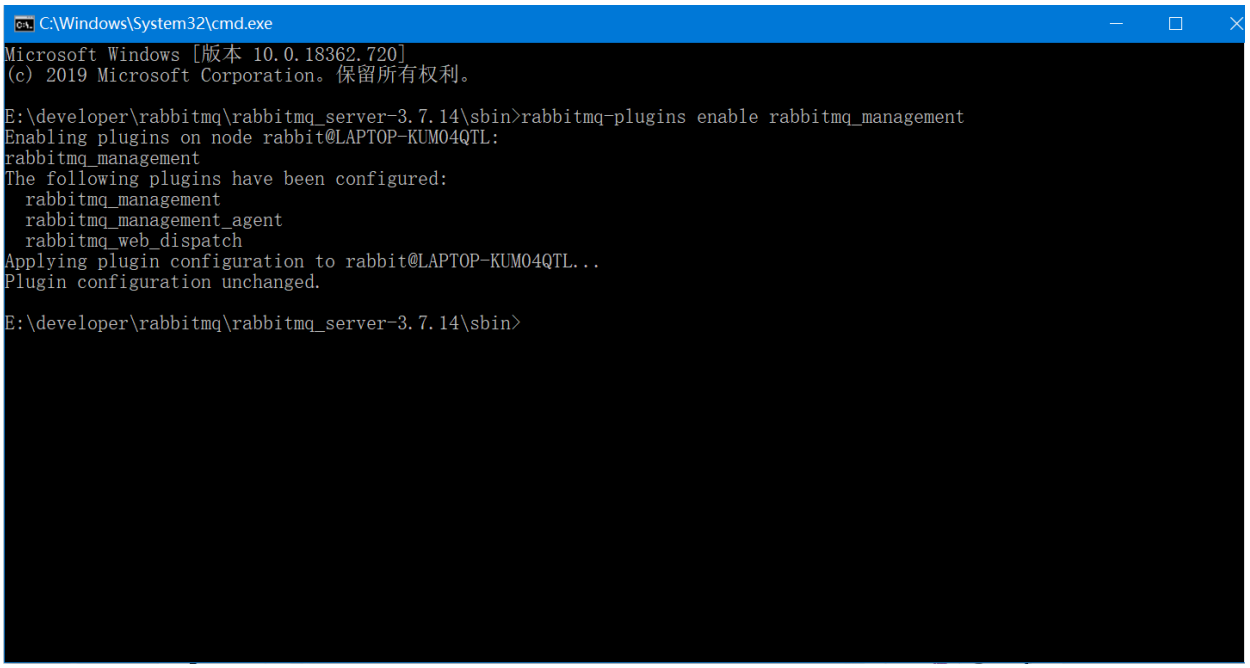


- 安装RabbitMQ，下载地址：https://dl.bintray.com/rabbitmq/all/rabbitmq-server/3.7.14/rabbitmq-server-3.7.14.exe
- 安装完成后，进入RabbitMQ安装目录下的sbin目录：

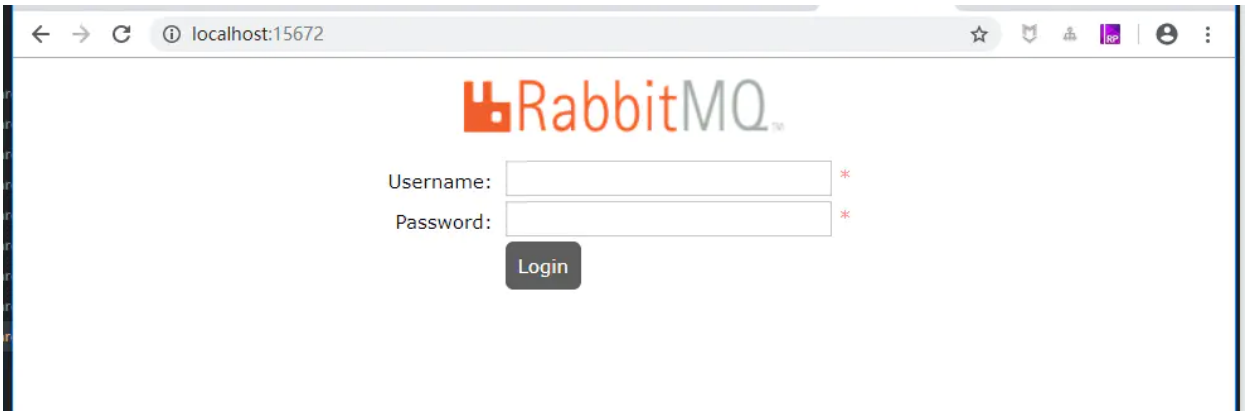


- 在地址栏输入cmd并回车启动命令行，然后输入以下命令启动管理功能：

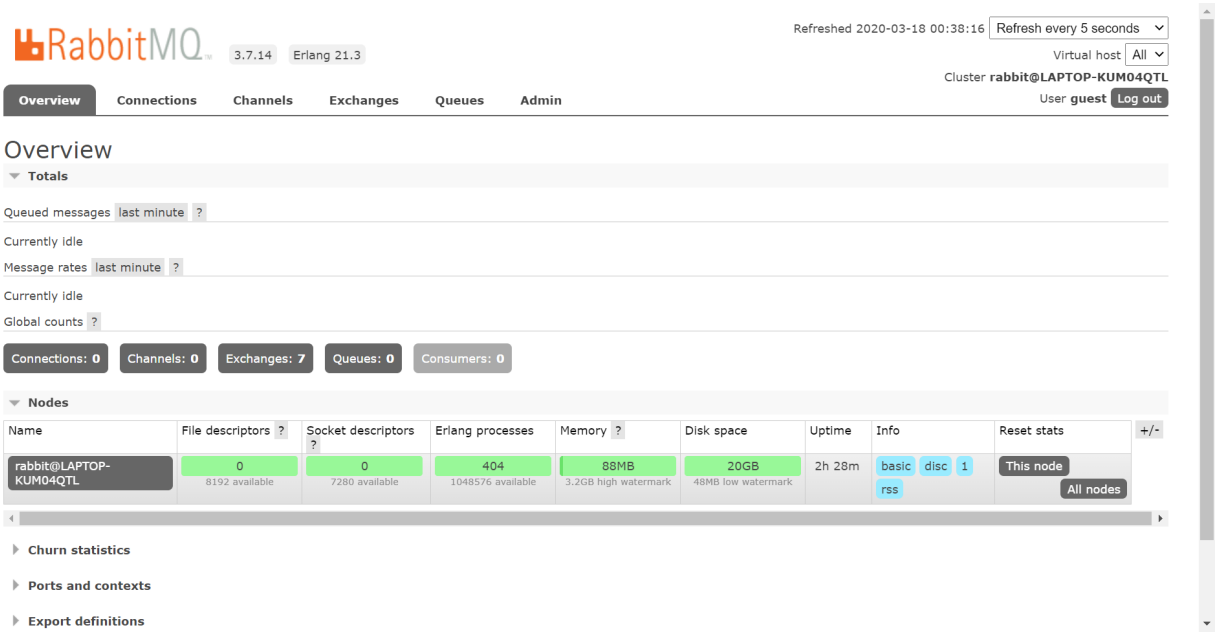
```
rabbitmq-plugins enable rabbitmq_management
```



- 访问地址查看是否安装成功：http://localhost:15672/



- 输入账号密码并登录：guest guest



动态刷新配置

使用 Spring Cloud Bus 动态刷新配置需要配合 Spring Cloud Config 一起使用，我们使用上一节中的config-server、config-client模块来演示下该功能。

给config-server添加消息总线支持

- 在pom.xml中添加相关依赖：

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--使用消息总线刷新配置时添加-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- 添加配置文件application-amqp.yml，主要是添加了RabbitMQ的配置及暴露了刷新配置的Actuator端点；

```
server:
 port: 8904
spring:
 application:
 name: config-server
 cloud:
 config:
 server:
 git:
 uri: https://gitee.com/macrozheng/springcloud-config.git
 username: macro
 password: 123456
 clone-on-start: true # 开启启动时直接从git获取配置
 rabbitmq: #rabbitmq相关配置
 host: localhost
 port: 5672
 username: guest
 password: guest
eureka:
 client:
 service-url:
 defaultZone: http://localhost:8001/eureka/
```

```yaml
management:
 endpoints: #暴露bus刷新配置的端点
 web:
 exposure:
 include: 'bus-refresh'
```

给config-client添加消息总线支持

- 在pom.xml中添加相关依赖：

```xml
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- 添加配置文件bootstrap-amqp1.yml及bootstrap-amqp2.yml用于启动两个不同的config-client，两个配置文件只有端口号不同；

```yaml
server:
 port: 9004
spring:
 application:
 name: config-client
 cloud:
 config:
 profile: dev #启用环境名称
 label: dev #分支名称
 name: config #配置文件名称
 discovery:
 enabled: true
 service-id: config-server
 rabbitmq: #rabbitmq相关配置
 host: localhost
 port: 5672
 username: guest
 password: guest
eureka:
 client:
 service-url:
 defaultZone: http://localhost:8001/eureka/
management:
 endpoints:
 web:
```

```
26  exposure:
27  include: 'refresh'
```

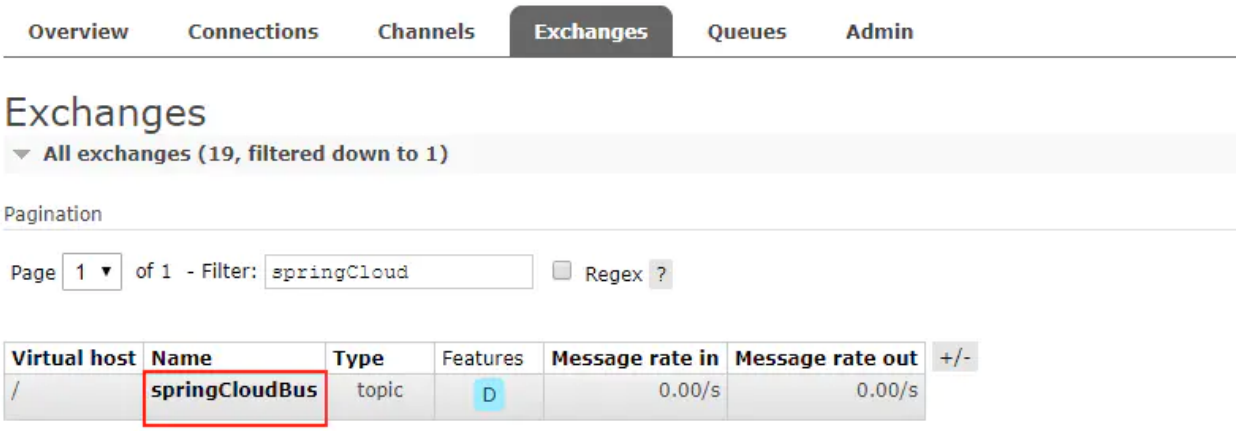
动态刷新配置演示

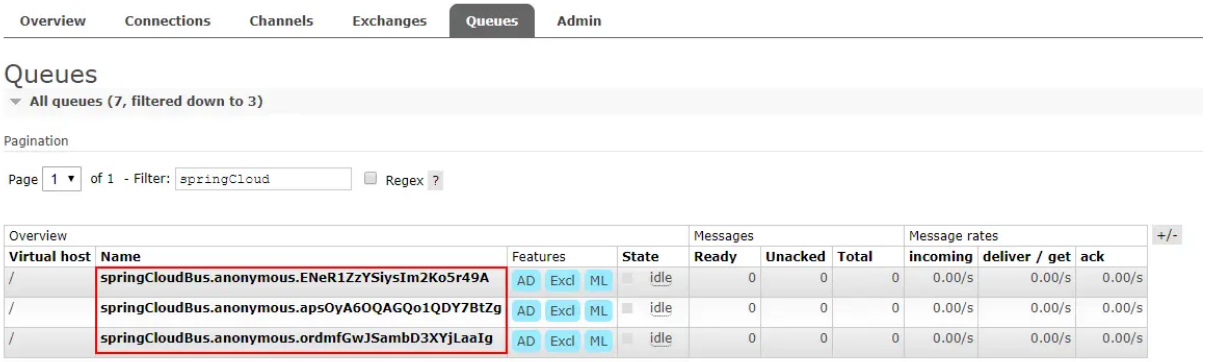
我们先启动相关服务，启动eureka-server，以application-amqp.yml为配置启动config-server，以bootstrap-amqp1.yml为配置启动config-client，以bootstrap-amqp2.yml为配置再启动一个config-client，启动后注册中心显示如下：

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-CLIENT	n/a (2)	(2)	UP (2) - DESKTOP-K1F7O7Q:config-client:9004 , DESKTOP-K1F7O7Q:config-client:9005
CONFIG-SERVER	n/a (1)	(1)	UP (1) - DESKTOP-K1F7O7Q:config-server:8904

启动所有服务后，我们登录RabbitMQ的控制台可以发现Spring Cloud Bus 创建了一个叫springCloudBus的交换机及三个以 springCloudBus.anonymous开头的队列：

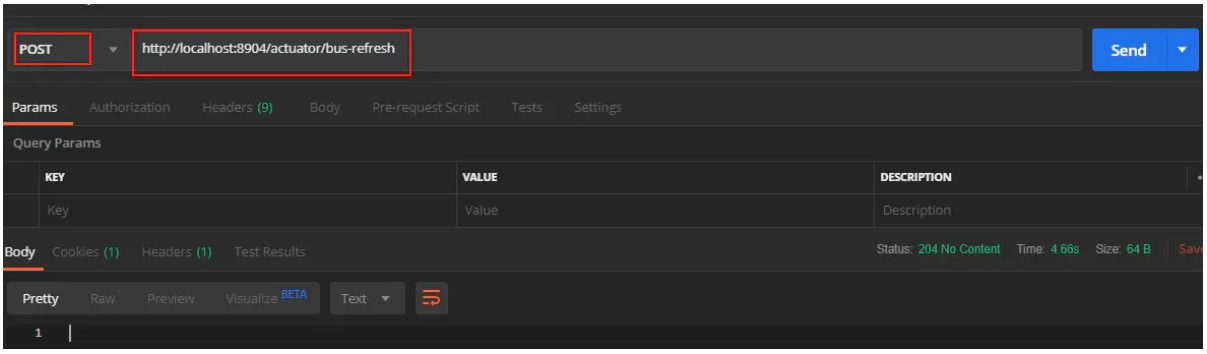




- 我们先修改Git仓库中dev分支下的config-dev.yml配置文件：

```
# 修改前信息
config:
 info: "config info for dev(dev)"
# 修改后信息
config:
 info: "update config info for dev(dev)"
```

- 调用注册中心的接口刷新所有配置：http://localhost:8904/actuator/bus-refresh



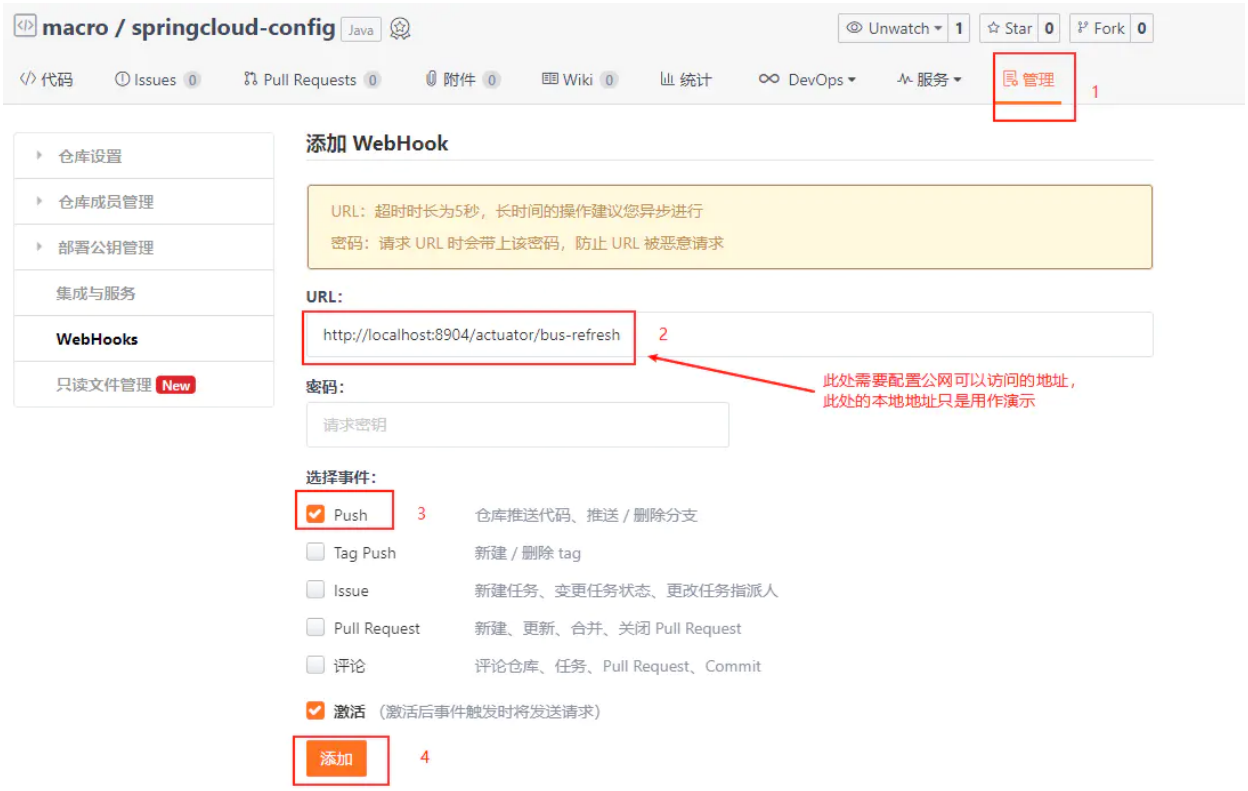
- 刷新后再分别调用 http://localhost:9004/configInfo 和 http://localhost:9005/configInfo 获取配置信息，发现都已经刷新了；

```
update config info for dev(dev)
```

- 如果只需要刷新指定实例的配置可以使用以下格式进行刷新：http://localhost:8904/actuator/bus-refresh/{destination}，我们这里以刷新运行在9004端口上的config-client为例 http://localhost:8904/actuator/bus-refresh/config-client:9004。

配合WebHooks使用

WebHooks相当于是一个钩子函数，我们可以配置当向Git仓库push代码时触发这个钩子函数，这里以Gitee为例来介绍下其使用方式，这里当我们向配置仓库push代码时就会自动刷新服务配置了。



Spring Cloud Stream 构建消息驱动微服务

快速实战

下面展示如何创建一个Spring Cloud Stream的应用程序，它是如何从消息中间件中接收并输出接收的信息到console，这里的消息中间件有两种选择：RabbitMQ和Kafka，我们以RabbitMQ为准。

这节主要简化官方文档为两步:

1. 使用idea新建项目
2. 添加 Message Handler , Building 并运行

一、使用idea新建项目

打开项目目录，新建一个moudle，名为first-stream，pom文件如下

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apac
he.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5    <parent>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
 </parent>
 <groupId>com.zhuawa</groupId>
 <artifactId>first-stream</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <name>first-stream</name>
 <description>Demo project for Spring Boot</description>

 <properties>
 <java.version>1.8</java.version>
 <spring-cloud.version>Hoxton.SR3</spring-cloud.version>
 </properties>

 <dependencies>
 <dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-stream</artifactId>
 </dependency>
 <!-- Stream rabbit 依赖中包含 binder-rabbit,所以只需导入此依赖即可 -->
 <dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
 </dependency>

 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
 <exclusions>
 <exclusion>
 <groupId>org.junit.vintage</groupId>
 <artifactId>junit-vintage-engine</artifactId>
 </exclusion>
 </exclusions>
 </dependency>
 <dependency>
 <groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-stream-test-support</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>

</project>
```

二、 添加 Message Handler , Building 并运行

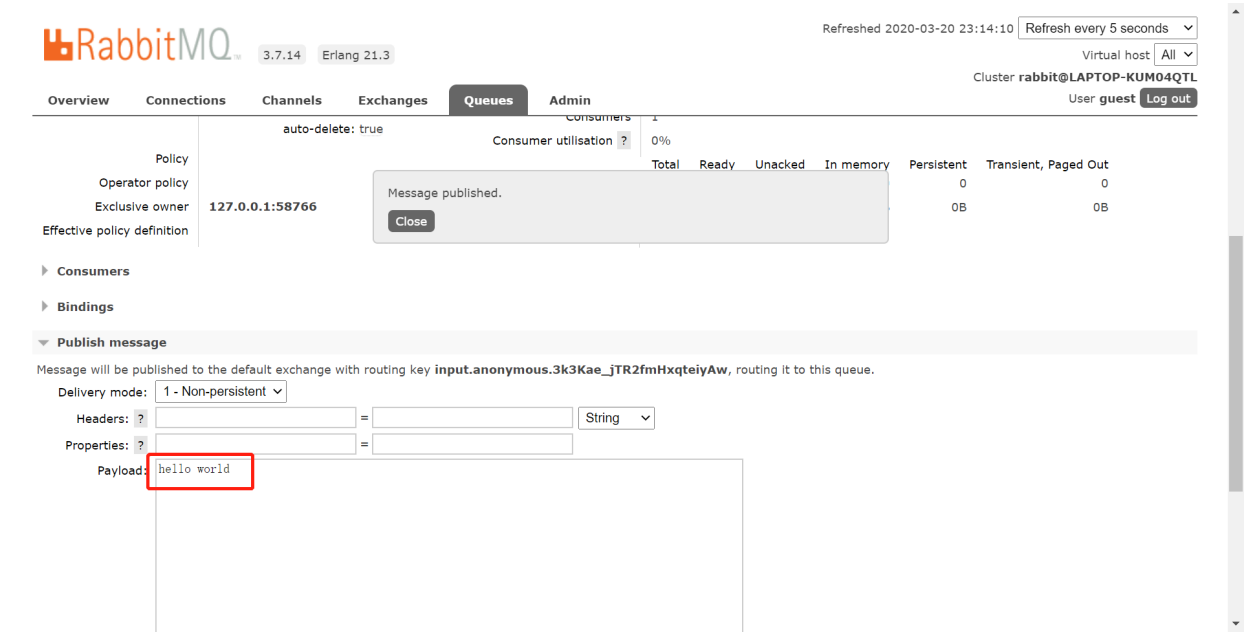
添加启动类，并添加如下代码：

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class FirstStreamApplication {

 private static final Logger logger = LoggerFactory.getLogger(FirstStream
Application.class);

 public static void main(String[] args) {
 SpringApplication.run(FirstStreamApplication.class, args);
 }
```

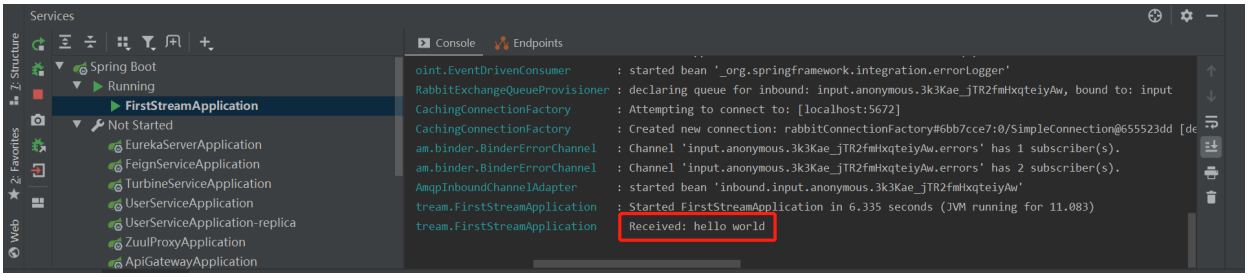
```
10
11  @StreamListener(Sink.INPUT)
12  public void receive(Object payload) {
13  logger.info("Received: " + payload);
14  }
15 }
```

- 我们通过使用`@EnableBinding(Sink.class)`开启了`Sink`的binding(绑定)，这样做会向框架发出信号，以启动与消息传递中间件的绑定，并自动创建绑定到Sink.INPUT通道的目标（即queue，topic和其他）。
- 我们添加了一个处理方法，去监听消息类型为String的消息，这么做是为了向你展示框架的核心特性之一——自动转换入参消息体为指定类型

启动项目，我们去查看RabbitMQ的网页http://localhost:15672/ 点击Connections，发现现在已经有一个连接进来了，我们刚才的项目，在Queues中也有一个队列被创建，我的是`input.anonymous.3k3Kae_jTR2fmHxqteiyAw`，我们点开那个唯一的队列，往下拉点开publish message，payload处输入一个`hello world`，点Publlish message发送一个消息



查看控制台，你会看到`Received: hello world`



对于连接非本地RabbitMQ的配置：

```
1 spring.rabbitmq.host=<rabbitMQ所在的ip>
2 spring.rabbitmq.port=<端口号>
```

```
spring.rabbitmq.username=<登录用户名>
spring.rabbitmq.password=<密码>
```

Spring Cloud Stream介绍

Spring Cloud Stream是一个用于构建消息驱动的微服务应用程序的框架，是一个基于Spring Boot 创建的独立生产级的，使用Spring Integration提供连接到消息代理的Spring应用。接下来介绍`持久发布 - 订阅(persistent publish-subscribe)`的语义，`消费组(consumer groups)`和`分区(partitions)`的概念。

我们可以添加`@EnableBinding`注解在你的应用上，从而立即连接到消息代理，在方法上添加`@StreamListener`以使其接收流处理事件，下面的例子展示了一个Sink应用接收外部信息

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

 public static void main(String[] args) {
 SpringApplication.run(VoteRecordingSinkApplication.class, args);
 }

 @StreamListener(Sink.INPUT)
  public void processVote(Vote vote) {
  votingService.recordVote(vote);
  }
}
```

`@EnableBinding`注解会带着一个或多个接口作为参数(举例中使用的是Sink的接口)，一个接口往往声名了输入和输出的渠道，Spring Stream提供了`Source`、`Sink`、`Processor`这三个接口，我们也可以自己定义接口。

下面是Sink的接口内容

```
public interface Sink {
 String INPUT = "input";

 @Input(Sink.INPUT)
 SubscribableChannel input();
}
```

`@Input`注解区分了一个输入channel，通过它接收消息到应用中，使用`@Output`注解 区分输出channel，消息通过它离开应用，使用这两个注解可以带一个channel的名字作为参数，**如果未提供channel名称，则使用带注释的方法的名称。**

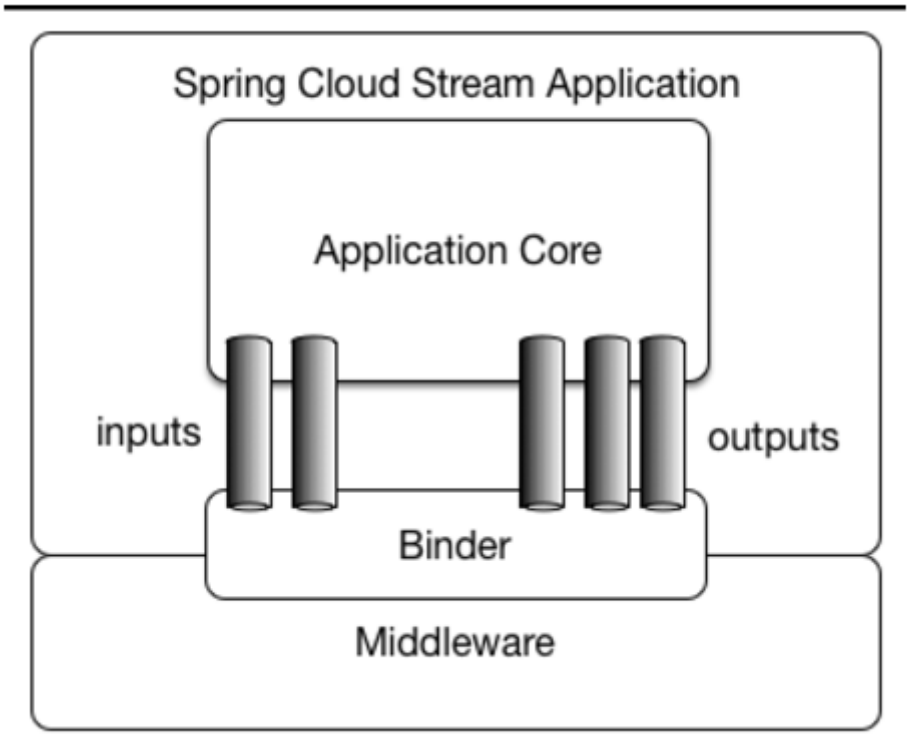
可以使用Spring Cloud Stream 现成的接口，也可以使用`@Autowired`注入这个接口，下面在测试类中举例

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class LoggingConsumerApplicationTests {

 @Autowired
 private Sink sink;

 @Test
 public void contextLoads() {
  assertNotNull(this.sink.input());
  }
}
```

主要概念(Main Concepts)

应用模型

应用程序通过 inputs 或者 outputs 来与 Spring Cloud Stream 中Binder 交互，通过配置来绑定，而 Spring Cloud Stream 的 Binder 负责与中间件交互。所以，我们只需要搞清楚如何与 Spring Cloud Stream 交互就可以方便使用消息驱动的方式。



抽象绑定器（The Binder Abstraction)

Spring Cloud Stream实现Kafkat和RabbitMQ的Binder实现，也包括了一个TestSupportBinder，用于测试。你也可以写根据API去写自己的Binder.

Spring Cloud Stream 同样使用了Spring boot的自动配置，并且抽象的Binder使Spring Cloud Stream的应用获得更好的灵活性，比如：我们可以在application.yml或application.properties中指定参数进行配置使用Kafka或者RabbitMQ，而无需修改我们的代码。

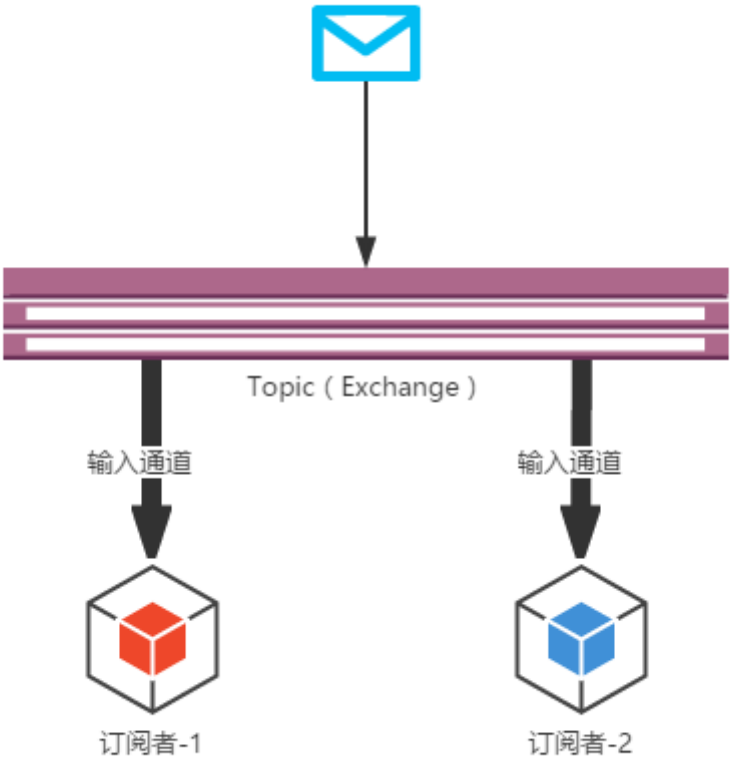
在前面我们测试的项目中并没有修改application.properties，自动配置得益于Spring Boot。

通过 Binder ，可以方便地连接中间件，可以通过修改application.yml中的`spring.cloud.stream.bindings.input.destination` 来进行改变消息中间件（对应于Kafka的topic，RabbitMQ的exchanges）

在这两者间的切换甚至不需要修改一行代码。

发布-订阅(Persistent Publish-Subscribe Support)

如下图是经典的Spring Cloud Stream的 发布-订阅 模型，生产者 生产消息发布在shared topic（共享主题）上，然后 消费者 通过订阅这个topic来获取消息

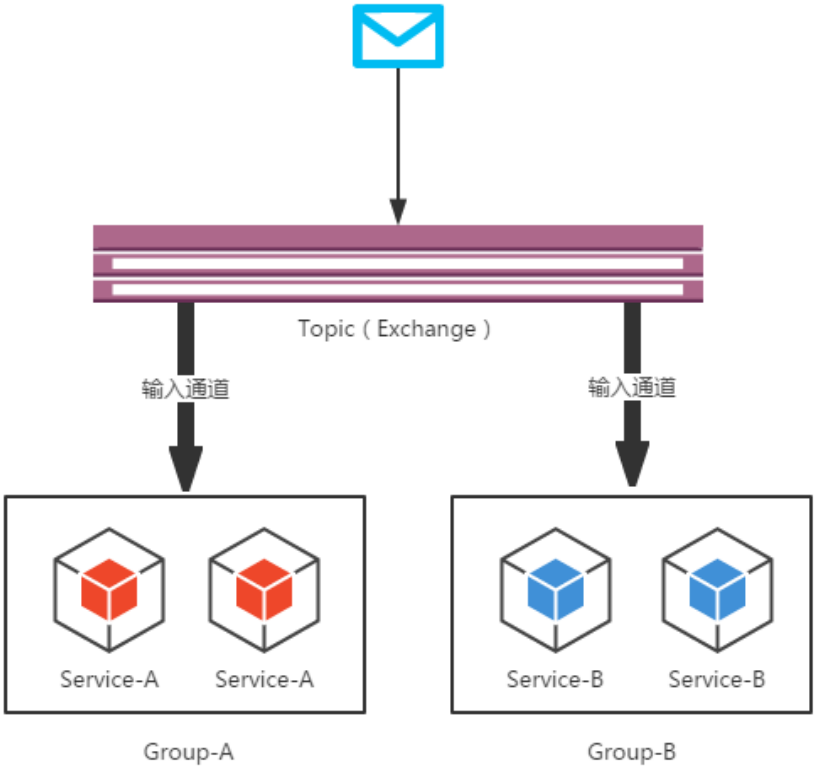


其中topic对应于Spring Cloud Stream中的destinations（Kafka 的topic，RabbitMQ的exchanges）

消费组（Consumer Groups）

尽管发布-订阅 模型通过共享的topic连接应用变得很容易，但是通过创建特定应用的多个实例的来扩展服务的能力同样重要，但是如果这些实例都去消费这条数据，那么很可能会出现重复消费的问题，我们只需要同一应用中只有一个实例消费该消息，这时我们可以通过消费组来解决这种应用场景， **当一个应用程序不同实例放置在一个具有竞争关系的消费组中，组里面的实例中只有一个能够消费消息**

设置消费组的配置为`spring.cloud.stream.bindings.<channelName>.group`，
举个例子：
下图中，通过网络传递过来的消息通过主题，按照分组名进行传递到消费者组中
此时可以通过`spring.cloud.stream.bindings.input.group=Group-A`或`spring.cloud.stream.bindings.input.group=Group-B`进行指定消费组



所有订阅指定主题的组都会收到发布消息的一个备份，每个组中只有一个成员会收到该消息；如果没有指定组，那么默认会为该应用分配一个匿名消费者组，与所有其它组处于 订阅-发布 关系中。ps:也就是说如果管道没有指定消费组，那么这个匿名消费组会与其它组一起消费消息，出现了重复消费的问题。

消费者类型（Consumer Types）

1）支持有两种消费者类型：

- Message-driven (消息驱动型，有时简称为异步)
- Polled (轮询型，有时简称为 同步)

在Spring Cloud 2.0版本前只支持 Message-driven这种异步类型的消费者，消息一旦可用就会传递，并且有一个线程可以处理它；当你想控制消息的处理速度时，可能需要用到同步消费者类型。

2）持久化

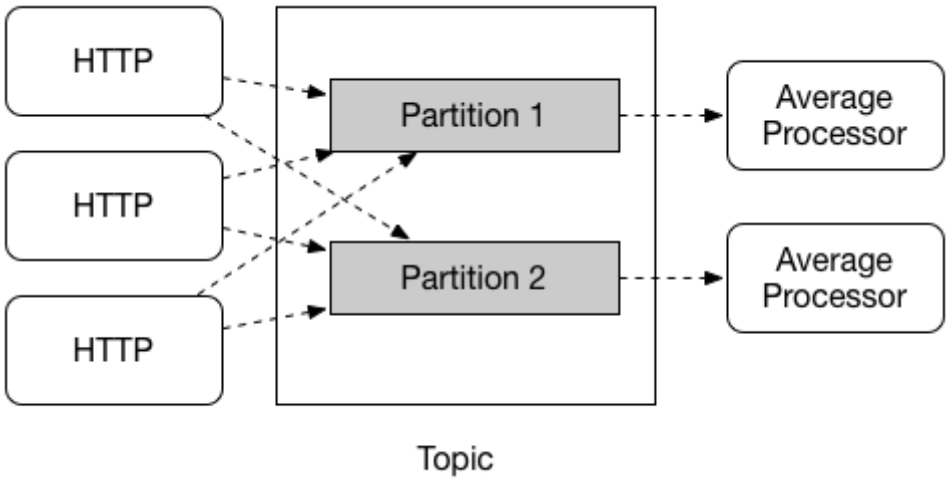
一般来说所有拥有订阅主题的消费组都是持久化的，除了匿名消费组。 Binder的实现确保了所有订阅关系的消费订阅是持久的，一个消费组中至少有一个订阅了主题，那么被订阅主题的消息就会进入这个组中，无论组内是否停止。

注意： 匿名订阅本身是非持久化的，但是有一些Binder的实现（比如RabbitMQ）则可以创建非持久化的组订阅

通常情况下，当有一个应用绑定到目的地的时候，最好指定消费消费组。扩展Spring Cloud Stream应用程序时，必须为每个输入绑定指定一个使用者组。这样做可以防止应用程序的实例接收重复的消息（除非需要这种行为，这是不寻常的）。

分区支持（Partitioning Support）

在消费组中我们可以保证消息不会被重复消费，但是在同组下有多个实例的时候，我们无法确定每次处理消息的是不是被同一消费者消费，分区的作用就是为了**确保具有共同特征标识的数据由同一个消费者实例进行处理**，当然前边的例子是狭义的，通信代理（broken topic）也可以被理解为进行了同样的分区划分。Spring Cloud Stream 的分区概念是抽象的，可以为不支持分区Binder实现（例如RabbitMQ）也可以使用分区。



注意：要使用分区处理，你必须同时对生产者和消费者进行配置。

编程模型（Programming Model）

为了理解编程模型，需要熟悉下列核心概念：

- **Destination Binders（目的地绑定器）**： 负责与外部消息系统集成交互的组件
- **Destination Bindings（目的地绑定）**： 在外部消息系统和应用的生产者和消费者之间的桥梁（由Destination Binders创建）
- **Message （消息）**： 用于生产者、消费者通过Destination Binders沟通的规范数据。

1. Destination Binders（目的地绑定器）：

Destination Binders是Spring Cloud Stream与外部消息中间件提供了必要的配置和实现促进集成的扩展组件。集成了生产者和消费者的消息的路由、连接和委托、数据类型转换、

用户代码调用等。

尽管Binders帮我们处理了许多事情，我们仍需要对他进行配置。

2. Destination Bindings （目的地绑定）：

如前所述，Destination Bindings 提供连接外部消息中间件和应用提供的生产者和消费者中间的桥梁。

使用@EnableBinding 注解打在一个配置类上来定义一个Destination Binding，这个注解本身包含有@Configuration，会触发Spring Cloud Stream的基本配置。

接下来的例子展示完全配置且正常运行的Spring Cloud Stream应用，由INPUT接收消息转换成String 类型并打印在控制台上，然后转换出一个大写的信息返回到OUTPUT中。

```
@SpringBootApplication
@EnableBinding(Processor.class)
public class MyApplication {

 public static void main(String[] args) {
 SpringApplication.run(MyApplication.class, args);
 }

 @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
  public String handle(String value) {
  System.out.println("Received: " + value);
  return value.toUpperCase();
  }
}
```

通过SendTo注解将方法内返回值转发到其他消息通道中，这里因为没有定义接收通道，提示消息已丢失，解决方法是新建一个接口，如下

```
public interface MyPipe{
  //方法1
  @Input(Processor.OUTPUT) //这里使用Processor.OUTPUT是因为要同一个管道，或者名称相同
  SubscribableChannel input();
  //还可以如下这样=====二选一即可==========
  //方法2
  String INPUT = "output";
  @Input(MyPipe.INPUT)
  SubscribableChannel input();
```

```
}
```

然后在在上边的方法下边加一个方法，并在@EnableBinding注解中改成@EnableBinding({Processor.class, MyPipe.class})

```
@StreamListener(MyPipe.INPUT)
public void handleMyPipe(String value) {
System.out.println("Received: " + value);
}
```

Spring Cloud Stream已经为我们提供了三个绑定消息通道的默认实现

- Sink：通过指定消费消息的目标来标识消息使用者的约定。
- Source：与Sink相反，用于标识消息生产者的约定。
- Processor：集成了Sink和Source的作用，标识消息生产者和使用者

他们的源码分别为：

```
public interface Sink {
 String INPUT = "input";

 @Input("input")
 SubscribableChannel input();
}

public interface Source {
 String OUTPUT = "output";

  @Output("output")
  MessageChannel output();
 }

public interface Processor extends Source, Sink {
}
```

Sink和Source中分别通过@Input和@Output注解定义了输入通道和输出通道，通过使用这两个接口中的成员变量来定义输入和输出通道的名称，Processor由于继承自这两个接口，所以同时拥有这两个通道。

注意：拥有多条管道的时候不能有输入输出管道名相同的，否则会出现发送消息被自己接收或报错的情况

我们可以根据上述源码的方式来定义我们自己的输入输出通道，定义输入通道需要返回SubscribaleChannel接口对象，这个接口继承自MessageChannel接口，它定义了维护消

息通道订阅者的方法；定义输出通道则需要返回MessageChannel接口对象，它定义了向消息通道发送消息的方法。

自定义消息通道 发送与接收

依照上面的内容，我们也可以创建自己的绑定通道 *如果你实现了上边的MyPipe接口，那么直接使用这个接口就好*

1. 和主类同包下建一个MyPipe接口，实现如下

```
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.SubscribableChannel;

public interface MyPipe {

 //方法1
// @Input(Source.OUTPUT) //Source.OUTPUT的值是output，我们自定义也是一样的
 // SubscribableChannel input(); //使用@Input注解标注的输入管道需要使用SubscribableChannel来订阅通道

  //========二选一使用===========

  //方法2
  String INPUT = "output";

  @Input(MyPipe.INPUT)
  SubscribableChannel input();
}
```

这里用Source.OUTPUT和第二种方法 是一样的，我们只要将消息发送到名为output的管道中，那么监听output管道的输入流一端就能获得数据

扩展主类，添加监听output管道方法

```
  @StreamListener(MyPipe.INPUT)
  public void receiveFromMyPipe(Object payload){
  logger.info("Received: "+payload);
  }
```

- 在主类的头上的@EnableBinding改为`@EnableBinding({Sink.class, MyPipe.class})`，加入了Mypipe接口的绑定
- 在test/java下创建`com.cnblogs.hellxz`，并在包下新建一个测试类，如下

```java
@RunWith(SpringRunner.class)
@EnableBinding(value = {Source.class})
@SpringBootTest
public class TestSendMessage {

 @Autowired
 private Source source; //注入接口和注入MessageChannel的区别在于发送时需不需要调用接口内的方法

 @Test
  public void testSender() {
  source.output().send(MessageBuilder.withPayload("Message from MyPipe").build());
  //假设注入了MessageChannel messageChannel; 因为绑定的是Source这个接口，
  //所以会使用其中的唯一产生MessageChannel的方法，那么下边的代码会是
  //messageChannel.send(MessageBuilder.withPayload("Message from MyPipe").build());
  }
}
```

- 启动主类，清空输出，运行测试类，然后你就会得到在主类的控制台的消息以log形式输出`Message from MyPipe`

我们是通过注入消息通道，并调用他的output方法声明的管道获得的MessageChannel实例，发送的消息

管道注入过程中可能会出现的问题

通过注入消息通道的方式虽然很直接，但是也容易犯错，当一个接口中有多个通道的时候，他们返回的实例都是MessageChannel，这样通过@Autowired注入的时候往往会出现有多个实例找到无法确定需要注入实例的错误，我们可以通过@Qualifier指定消息通道的名称，下面举例：

- 在主类包内创建一个拥有多个输出流的管道

```java
/**
 * 多个输出管道
 */
public interface MutiplePipe {

 @Output("output1")
 MessageChannel output1();

```

```
9   @Output("output2")
10   MessageChannel output2();
11  }
```

- 创建一个测试类

```
1  @RunWith(SpringRunner.class)
2  @EnableBinding(value = {MutiplePipe.class}) //开启绑定功能
3  @SpringBootTest //测试
4  public class TestMultipleOutput {
5
6   @Autowired
7   private MessageChannel messageChannel;
8
9   @Test
10   public void testSender() {
11   //向管道发送消息
12   messageChannel.send(MessageBuilder.withPayload("produce by multiple pip
e").build());
13   }
14  }
```

启动测试类，会出现刚才说的不唯一的bean，无法注入

```
1  Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionExcept
ion: No qualifying bean of type 'org.springframework.messaging.MessageChann
el' available: expected single matching bean but found 6: output1,output2,i
nput,output,nullChannel,errorChannel
```

我们在`@Autowired`旁边加上`@Qualifier("output1")`，然后测试就可以正常启动了

通过上边的错误，我们可以清楚的看到，每个MessageChannel都是使用消息通道的名字做为bean的名称。
这里我们没有使用监听这个管道，仅为了测试并发现问题

常用配置

消费组和分区的设置

给消费者设置消费组和主题

1. 设置消费组： `spring.cloud.stream.bindings.<通道名>.group=<消费组名>`

2. 设置主题： `spring.cloud.stream.bindings.<通道名>.destination=<主题名>`

给生产者指定通道的主题：`spring.cloud.stream.bindings.<通道名>.destination=<主题名>`

消费者开启分区，指定实例数量与实例索引

1. 开启消费分区： `spring.cloud.stream.bindings.<通道名>.consumer.partitioned=true`
2. 消费实例数量： `spring.cloud.stream.instanceCount=1` (具体指定)
3. 实例索引： `spring.cloud.stream.instanceIndex=1` #设置当前实例的索引值

生产者指定分区键

1. 分区键： `spring.cloud.stream.bindings.<通道名>.producer.partitionKeyExpress=<分区键>`
2. 分区数量： `spring.cloud.stream.bindings.<通道名>.producer.partitionCount=<分区数量>`