Spring Cloud Bus：消息总线

- 通常会使用轻量级的消息代理来构建一个共用的消息主题让系统中所有微服务实例都连接上来，由于该主题中产生的消息会被所有实例监听和消费， 所以称它为消息总线
- 在总线上的各个实例都可以方便地广播一些需要让其他连接在该主题上的实例都知道的消息， 例如配置信息的变更或者其他一些管理操作等
- 消息总线中的常用功能， 比如， 配合Spring Cloud Config 实现微服务应用配置信息的动态更新等

消息代理

- 消息代理 (Message Broker) 是一种消息验证、传输、路由的架构模式。它在应用程序之间起到通信调度并最小化应用之间的依赖的作用，使得应用程序可以高效地解耦通信过程
- 消息代理是一个中间件产品， 它的核心是一个消息的路由程序， 用来实现接收和分发消息， 并根据设定好的消息处理流来转发给正确的应用
    - 它包括独立的通信和消息传递协议， 能够实现组织内部和组织间的网络通信
    - 设计代理的目的就是为了能够从应用程序中传入消息， 并执行一些特别的操作
- 使用消息代理的场景：
    - 将消息路由到一个或多个目的地
    - 消息转化为其他的表现方式
    - 执行消息的聚集、 消息的分解， 并将结果发送到它们的目的地， 然后重新组合响应返回给消息用户
    - 调用Web服务来检索数据
    - 响应事件或错误
    - 使用发布-订阅模式来提供内容或基千主题的消息路由
- Kafka 、RabbitMQ消息中间件与Spring Cloud Bus 配合实现消息总线

RabbitMQ实现消息总线

- RabbitMQ是实现了高级消息队列协议CAMQP)的开源消息代理软件， 也称为面向消 息的中间件
- RabbitMQ服务器是用高性能、 可伸缩而闻名的Erlang语言编写而成的， 其集群和故障转移是构建在开放电信平台框架上的
- AMQP是Advanced Message Queuing Protocol的简称，它是一个面向消息中间件的开放式标准应用层协议。 它定义了以下这些特性
    - 消息方向 、消息队列、消息路由（包括点到点和-发布订阅模式）、可靠性和安全性
- AMQP要求消息的提供者和客户端接收者的行为要实现对不同供应商可以用相同的方式(比如SMTP、 HTTP、FTP等)进行互相操作
- AMQP与JMS不同，JMS定义了一个API和一 组消息收发必须实现的行为，而AMQP是一个线路级协议
    - 线路级协议描述的是通过网络发送的数据传输格式
    - 因此，任何符合该数据格式的消息发送和接收工具都能互相兼容和进行操作，这样就能轻易实现跨技木平台的架构方案

- RabbitMQ以AMQP协议实现， 所以它可以支持多种操作系统、 多种编程语言， 几乎可以覆盖所有主流的企业级技术平台
- 在SpringCloudBus中包含了对Rabbit的自动化默认配置

基本概念

- Broker：可以理解为消息队列服务器的实体， 它是一个中间件应用， 负责接收消息生产者的消息， 然后将消息发送至消息接收者或者其他的Broker
- Exchange: 消息交换机，是消息第一个到达的地方，消息通过它指定的路由规则，分发到不同的消息队列中去
- Queue: 消息队列，消息通过发送和路由之后最终到达的地方，到达Queue的消息即进入逻辑上等待消费的状态。 每个消息都会被发送到 一 个或多个队列 。
- Binding: 绑定，它的作用就是把Exchange和Queue按照路由规则绑定起来， 也就是Exchange和Queue之间的虚拟连接 。
- RoutingKey: 路由关键字，Exchange根据这个关键字进行消息投递。
- Virtual host: 虚拟主机， 它是对Broker的虚拟划分， 将消费者、 生产者 和它们依赖的AMQP相关结构 进行隔离，一般都是为了安全考虑。比如，可以在 一 个Broker中设置多个虚拟主机， 对不同用户进行权限的分离
- Connection: 连接， 代表生产者、 消费者、Broker之间进行通信的物理网络。
- Channel: 消息通道，用于连接生产者和消费者的逻辑结构 。在客户端的每个连接里， 可建立多个Channel, 每个Channel代表 一 个会话任务， 通过Channel可以隔离同一连接中的不同交互内容。
- Producer: 消息生产者， 制造消息并发送消息的程序。
- Consumer: 消息消费者，接收消息并处理消息的程序

消息投递到队列过程

1. 客户端连接到消息队列服务器， 打开一 个Channel
2. 客户端声明一个Exchange, 并设置相关属性
3. 客户端声明一个Queue, 并设置相关属性
4. 客户端使用Routing Key, 在Exchange和Queue之间建立好绑定关系
5. 客户端投递消息到Exchange
6. Exchange接收到消息后，根据消息的Key和已经设置的Binding,进行消息路由，将消息投递到一个或多个Queue里

- Exchange类型
    - Direct交换机：完全根据Key进行投递。 比如，绑定时设置了 Routing Key为abc, 那么客户端提交的消息，只有设置了Key为abc 的才会被投递到队列
    - Topic交换机：对Key进行模式匹配后进行投递，可以使用符号#匹配一 个或多个词， 符号*匹配正好一个词。比如，abc.#匹配abc.def.ghi,abc.*只匹配abc.def
    - Fanout交换机：不需要任何Key,它采取广播的模式，一个消息进来时，投递到与该交换机绑定的所有队列
- 消息的待久化
    - 也就 是将数据写在磁盘上
    - Exchange 持久化，在声明时指定durable => 1
    - Queue 待久化，在声明时指定durable => 1
    - 消息持久化，在投递时指定delivery_mode => 2 (1是非持久化)

    - 如果Exchange和Queue都是持久化的，那么它们之间的Binding也是持久化的。 如果Exchange和Queue两者之间有一 个是待久化的，一 个是非持久化的， 就不允许建立绑定

使用

- 安装web页面管理插件，`web页面`：`http://localhost:15672`，默认guest/guest
    - Tags 标签是 RabbitMQ 中的角色分类
        - none: 不能访问 management plugin
        - management: 用户可以通过 AMQP 做的任何事外加如下内容
            - 列出自己可以通过 AMQP 登入的 virtual hosts。
            - 查看自己的 virtual hosts 中的 queues、 exchanges 和 bindings。
            - 查看和关闭自己的 channels 和 connections。
            - 查看有关自己的 virtual hosts 的 “ 全局 ” 统计信息， 包含其他用户在这些 virtual hosts 中的活动
        - policymaker: management 可以做的任何事外加如下内容
            - 查看、 创建和删除自己的 virtual hosts 所属的 policies 和 parameters
        - monitoring: management 可以做的任何事外加如下内容
            - 列出所有 virtual hosts, 包括它们不能登录的 virtual hosts。
            - 查看其他用户的 connections 和 channels。
            - 查看节点级别的数据， 如 clustering 和 memory 的使用情况
            - 查看真正的关于所有virtual hosts的全局的统计信息
        - administrator: policymaker和monitoring可以做的任何事外加如下内容
            - 创建和删除virtual hosts。
            - 查看、 创建和删除users。
            - 查看、 创建和删除permissions。 . 关闭其他用户的connections
- `api端口`：`5672`
- `rabbitmq-hello`：`amqp`依赖
- 主类不需要加额外注解

```
# application.properties
spring.application.name=rabbitmq-hello

spring.rabbitmq.host=192.168.55.121
spring.rabbitmq.port=5672
spring.rabbitmq.username=vagrant
spring.rabbitmq.password=vagrant
```

```
// 消息生产者
@Component
public class Sender {
    //AmqpTemplate接口定义了一套针对AMQP协议的基础操作。 在SpringBoot中会根据配置来注入其具体实现。
```

```
    @Autowired
    private AmqpTemplate amqpTemplate;

    public void send() {
        String str = "Hello " + new Date();
        System.out.println("Sender:" + str);
        //产生一个字符串，并发送到名为hello的队列中
        this.amqpTemplate.convertAndSend("hello", str);
    }
}
```

```
//消息接收者
@Component
//通过@RabbitListener注解定义该类对hello队列的监听
@RabbitListener(queues = "hello")
public class Receiver {
    //用@RabbitHandler注解来指定对消息的处理方法
    @RabbitHandler
    public void process(String hello) {
        System.out.println("Receiver: " + hello);
    }
}
```

```
//RabbitMQ 的配置类 RabbitConfig，用来配置队列、交换器、路由等高级信息
@Configuration
public class RabbitConfig {
    @Bean
    public Queue helloQueue() {
        return new Queue("hello");
    }
}
```

```
//单元测试类， 用来调用消息生产
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitHelloApplicationTests {

  @Autowired
  private Sender sender;

  @Test
  public void contextLoads() {
    sender.send();
  }
}
```

- 在整个生产消费过程中， 生产和消费是一个异步操作， 这也是在分布式系统中要使用消息代理的重要原因， 以此可以使用通信来解耦业务逻辑
- 先生产消息，消费者启动后会去消费消息，通过生产消费模式的异步操作，系统间调用就没有同步调用需要那么高的实时性要求，同时也更容易控制处理的吞吐量以保证系统的正常运行等

整合Spring Cloud Bus

- 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
- 微服务应用的实例中都引入了Spring Cloud Bus, 所以它们都连接到了RabbitMQ的消息总线上
- 扩展config-client应用
  - 增加`bus-amqp`、`actuator`模块
  - application.properties中增加RabbitMQ的连接和用户信息

```
spring.rabbitmq.host=192.168.55.121
spring.rabbitmq.port=5672
spring.rabbitmq.username=vagrant
spring.rabbitmq.password=vagrant
```

- 启动`config-server`
- 再启动两个config-client（分别在不同的端口上，比如7002、7003）
- 先访问两个 config-client-eureka 的/from 请求
- 修改git上配置文件中的from属性值
- 发送PSOT请求到其中的一个`config-client`的/bus/refresh
- 再访问两个 config-client-eureka 的/from 请求
- 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l和实例2从总线中获取到，并重新从ConfigServer中获取它们的配置信息，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收到更新请求，所以在Web Hook中就不需要维护所有节点内容来进行更新，从而解决仅通过Web Hook来逐个进行刷新的问题
- 指定刷新范围
  - 刷新微服务中某个具体实例的配置
    - /bus/refresh 接口提供了 一个destination参数， 用来定位具体要刷新的应用程序
    - 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
  - 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)来实现的，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
  - 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
  - 结果是， 指定的应用实例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， 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。 所以要尽可能地让服务集群中的各个节点是对等的
  - 改动
    - ConfigServer中也引入SpringCloudBus, 将配置服务端也加入到消息总线中来
    - /bus/refresh请求不再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
    - 服务实例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ConfigServer即可， 从而简化了集群上的一些维护工作
- RabbitMQ配置
  - SpringCloud Bus中的RabbitMQ整合使用了Spring Boot的ConnectionFactory, 所以在SpringCloud Bus中支持使用以spring.rabbitmq为前缀的Spring Boot配置属性

Kafka实现消息总线

- Kafka是一个由Linkedln开发的分布式消息系统
- Kafka使用Scala实现， 被用作Linkedln的活动流和运营数据处理的管道， 现在也被诸多互联网企业广泛地用作数据流管道和消息系统
- Kafka是基于消息发布-订阅模式实现的消息系统， 其主要设计目标如下：
  - 消息持久化: 以时间复杂度为 0(1)的方式提供消息持久化能力， 即使对 TB 级以上的数据也能保证常数时间复杂度的访问性能
  - 高吞吐:在廉价的商用机器上也能支持单机每秒10万条以上的吞吐量。
  - 分布式 : 支待消息分区以及分布式消费， 并保证分区 内的消息顺序。
  - 跨平台 : 支待不同技术平台的客户端(如Java、 PHP、 Python 等)。
  - 实时性: 支待实时数据处理和离线数据处理 。
  - 伸缩性: 支持水平扩展。
- 基本概念
  - Broker: Kafka集群包含一 个或多个服务器 ， 这些服务器被称为Broker。
  - Topic: 逻辑上同RabbitMQ的Queue队列相似，每条发布到Kafka集群的消息都必 须有一 个Topico (物理上不同Topic的消息分开存储 ，逻辑上一 个Topic的消息虽然 保存于 一 个或多个Broker上 ， 但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处。)
  - Part山on:Partition是物理概念上的分区，为了提供系统吞吐率，在物理上每个Topic 会分成一个或多个Part巾on, 每个Partition对应一个文件夹(存储对应分区的消息内容和索引文件)。
  - Producer: 消息生产者，负责生产消息并发送到KafkaBroker。
  - Consumer: 消息消费者， 向KafkaBroker读取消息并处理的客户端。
  - ConsumerGroup: 每个Consumer属千一 个特定的组(可为每个Consumer指定属于 一 个组，若不指定则属千默认组)，组可以用来实现一 条消息被组内多个成员消费等功能。

事件驱动模型

- Spring 的事件驱动模型中包含了三个基本概念: 事件、 事件监听者和 事件发布者

- 事件：
  - Spring中定义了事件的抽象类ApplicationEvent， 它继承自JDK的Event Object类
  - 包含了两个成员变量：timestamp，该字段用于存储事件发生的时间戳， 以及父类中的source , 该字段表示源事件对象
  - 当需要自定义事件的时候，只需要继承ApplicationEvent，比如RernoteApplicationEvnet、RefreshRemoteApplicationEvent等， 可以在自定义的Event中增加一些事件的属性来给事件监听者处理
- 事件监听者：
  - Spring中定义了事件监听者的接口ApplicationListener, 它继承自 JDK 的EventListener接口， 同时ApplicationListener接口限定了ApplicationEvent子类作为该接口中onApplicationEvent(E event); 函数 的参数
  - 所以，每一个ApplicationListener 都是针对某个ApplicationEvent子类的监听和处理者
- 事件发布者：
  - Spring中定义了ApplicationEventPublisher和ApplicationEventMulticaster两个接口用来发布事件
  - ApplicationEventPublisher 接口定义了发布事件的函数publishEvent(ApplicationEvent event)和 publishEvent(Object event);
    - ApplicationEvent Publisher 的publishEvent实现在AbstractApp让cationContext 中
    - 它最终会调用 ApplicationEventMulticaster的multicastEvent来具体实现发布事件给监听者的操作
  - 而ApplicationEventMulticaster接口中定义了对ApplicationListener 的维护操作(比如新增、移除等)以及将 ApplicationEvent多播给可用ApplicationListener的 操作
    - App让calionEventMulticaster在Spring的默认实现位于SimpleApplicationEventMulticaster中，SimpleApp让cationEventMulticaster 通过遍历维护的 ApplicationListener集合来找到对应 ApplicationEvent 的监听器，然后调用监听器的 onApplicationEvent函数来对具体事件做出处理操作

总结

- Spring Cloud Bus 在绑定具体消息代理的输入与输出通道时均使用了抽象接口的方式， 所以真正的实现来自于spring-cloud-starter-bus-amqp 和 spring-cloud-starter-bus-kafka 的依赖
- spring-cloud-starter-bus-amqp依赖了spring-cloud-starter-stream-rabbit
- spring-cloud-starter-bus-kafka依赖了spring-cloud-starter-stream-kafka
- 真正实现与这些消息代理进行交互操作的是Spring Cloud Stream。
- 使用的所有Spring Cloud Bus的消息通信基础实际上都是由Spring Cloud Stream所提供的
- 一定程度上，可以将Spring Cloud Bus理解为是一个使用了Spring Cloud Stream构建的上层应用
- 由于Spring Cloud Stream为了让开发者屏蔽各个消息代理之间的差异，将来能够方便地切换不同的消息代理而不影响业务程序，所以在业务程序与消息代理之间定义了一层抽象，称为绑定器（Binder)
- 在整合RabbitMQ和Kafka的时候就是分别引入了它们各自的绑定器实现
- 不论使用RabbitMQ还是Kafka实现，在程序上其实没有任何变化，变化 的只是对绑定器的配置，所以，当要在其他消息代理上使用Spring Cloud Bus消息总线时， 只需要去实现一套指定消息代理的绑定器即可