8. 第 8 章 Spring Cloud Stream 的事件驱动架构

本章内容

- 理解事件驱动的架构处理及其与微服务的相关性
- 使用 Spring Cloud Stream 简化微服务中的事件处理
- 配置 Spring Cloud Stream
- 使用 Spring Cloud Stream 和 Kafka 发布消息
- 使用 Spring Cloud Stream 和 Kafka 消费消息
- 使用 Spring Cloud Stream， Kafka 和 Redis 实现分布式缓存

你最后一次和别人坐下来聊天是什么时候？回想一下你是如何和那个人交往的。它是一个完全集中的信息交换，你说了什么，然后没有做任何其他的，而你等待别人作出充分回应？当你说话的时候，你是否完全专注于谈话，不让外界的任何事物干扰你？如果你对这些问题说“是”，那么你已经得到了启发，是一个比我更好的人，应该停止你正在做的事情，因为你现在可以回答一个古老的问题：“一个物体的拍手声是什么？”。还有，我怀疑你没有孩子。

现实是人类不断地处于运动状态，与周围的环境相互作用，同时从周围的事物中发出和接收信息。在我家里，典型的谈话可能是这样的。我正忙着洗盘子，一边和妻子谈话。我告诉她我的一天。她看着她的电话，她听了，处理我所说的，偶尔回应着。我正在洗碗时，听到隔壁房间一阵骚动。我停止我正在做的事情，冲到隔壁房间里找出什么不对，看到了我们比九个月还大的小狗，Vader，刁着我三岁的儿子的鞋，并像战利品一样沿着客厅携带着鞋。我三岁大的儿子不高兴了。我穿过房子，追着狗直到我把鞋子弄回来。然后我回去继续洗盘子，和妻子聊天。

我告诉你们这一点并不是要告诉你们我生命中的一个典型的日子，而是指出我们与世界

的互动不是同步的、线性的、狭义地定义为一个请求响应模型。它是消息驱动的，在那里我们不断地发送和接收消息。当我们收到消息时，我们会对这些信息作出反应，同时常常中断我们正在进行的主要任务。

本章是关于如何设计和实现基于 Spring 的微服务，使用异步消息与其它微服务通信。在应用程序之间使用异步消息进行通讯不是新的。新的概念是使用消息来传递表示状态变化的事件。这个概念称为事件驱动架构（EDA）。它也称为消息驱动架构（MDA）。基于 EDA 的方法允许你构建高度解耦的系统，可以在不紧密耦合到特定库或服务的情况下对更改作出反应。当微服务组合起来，EDA 允许你通过仅仅具有服务监听的一系列由你的应用程序发出的事件（消息），来快速添加新的功能到你的应用程序。

Spring Cloud 项目使通过 Spring Cloud Stream 子项目构建基于消息的解决方案变得微不足道。Spring Cloud Stream 允许你轻松地实现消息发布和消费，同时将你的服务从与底层消息传递平台相关联的实现细节中屏蔽。

8.1. 消息，EDA 和微服务的案例

为什么消息在构建微服务应用程序中很重要？为了回答这个问题，让我们从一个例子开始。我们将使用我们在本书中使用的两个服务：许可服务和组织服务。让我们假设在将这些服务部署到生产之后，您会发现在进行组织服务的组织信息查找时，许可服务调用花费了很长的时间。当你看组织数据的使用模式，你会发现组织的数据很少变化，通过组织记录的主键从组织服务读取大多数的数据。如果可以缓存组织数据的读取信息而不必花费访问数据库的成本，那么可以大大提高许可服务调用的响应时间。

当你在实现一个缓存的解决方案，你意识到你有三个核心要求：

- *缓存的数据需要在许可服务的所有实例之间保持一致。*这意味着你不能在许可服务

中本地缓存数据，因为你希望保证读取相同的组织数据，而不管服务实例是否命中了它。

- *不能将组织数据缓存在承载许可服务的容器的内存中。*承载服务运行时容器通常被限制大小，可以使用不同的访问模式访问数据。本地缓存可能引入复杂性，因为你必须保证你的本地缓存同步到集群中的所有其他服务。
- *当组织记录通过更新或删除更改时，你希望许可服务识别到组织服务中的状态更改。*许可服务将使特定组织的任何缓存数据失效，并从缓存中回收它。

让我们来看一下实现这些需求的两种方法。第一种方法将使用一个同步请求响应模型实现上述需求。当组织状态发生变化时，许可服务和组织服务通过它们的 REST 端点来回通信。第二种方法是组织服务发出一个异步事件（消息），该消息将通知组织服务数据发生了变化。在第二种方法中，组织服务将向组织记录更新或删除的队列发布消息。许可服务将与中介一起监听，查看已发生组织事件，并从它的缓存中清除组织数据。

8.1.1. 使用同步请求响应的方法传递状态改变

为你的组织的数据缓存，你要使用 Redis (http://redis.io/)，一个分布式的 Key-Value 存储数据库。图 8.1 提供了如何使用传统的同步请求响应编程模型建立缓存解决方案的高级概述。

① A licensing service user makes a call to retrieve licensing data.

许可服务用户发起一个调用来检索许可数据。

② The licensing service first checks the Redis cache for the organization data.

许可服务首先检查组织的 Redis 缓存数据。

③ If the organization data isn’t in the Redis cache, the licensing service calls the

organization service to retrieve it.

如果组织的数据不在 Redis 缓存，许可服务调用组织服务来检索。

④ Organization data may be updated via calls to the organization service.

组织数据可以通过对组织服务的调用来更新。

⑤ When organization data is updated, the organization service either calls back into the licensing service endpoint and tells it to invalidate its cache or talks to the licensing service's cache directly.

当组织数据更新时，组织服务要么回调到许可服务端点，并告诉它使缓存失效，要么直接与许可服务的缓存交互。

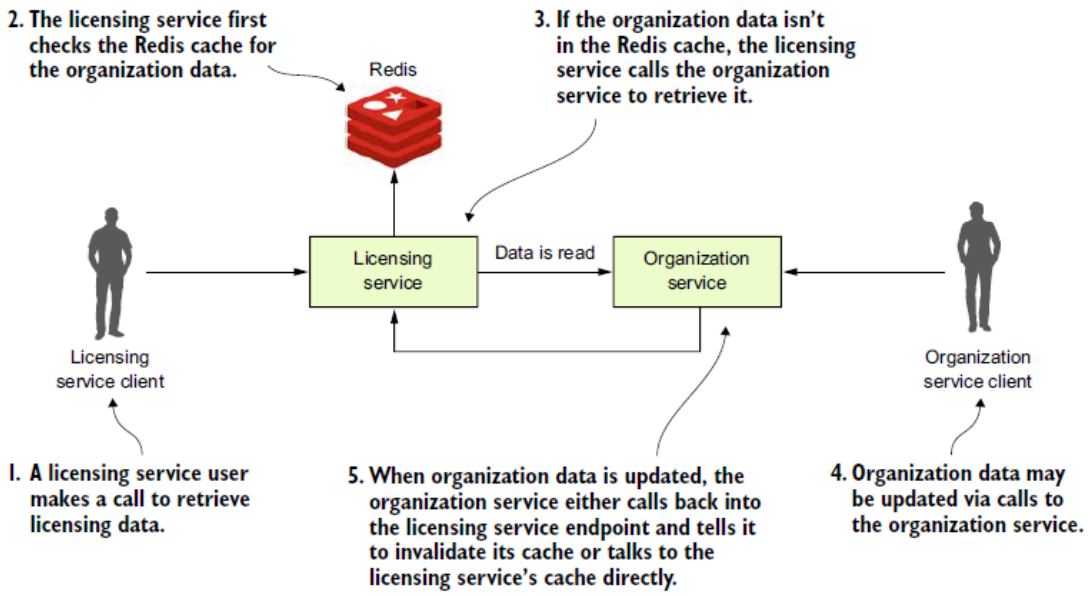

图 8.1 在同步请求响应模型，紧密耦合服务引入了复杂性和脆弱性。

在图 8.1 中，当用户调用许可服务时，许可服务也需要查找组织数据。许可服务将首先检查由其 ID 从 Redis 集群检索所需的组织数据。如果许可服务找不到组织的数据，它将使用基于 REST 的端点调用组织服务并在返回组织数据给用户之前，将返回的数据存储在 Redis。现在，如果有人使用组织服务的 REST 端点更新或删除组织记录，组织服务将需要

调用许可服务上暴露的端点，并告诉它在其缓存中使组织数据失效。在图 8.1 中，如果你看一下组织服务回调许可服务来告诉它使 Redis 缓存失效，你可以看到至少三个问题：

- 组织服务和许可服务紧密耦合。
- 耦合在服务之间已经引入脆弱性。如果缓存变化的许可服务端点失效，但组织服务已经改变。
- 这种方法是不灵活的，因为即使不修改组织服务上的代码以知道它调用了其它服务来让它知道更改，也不能添加组织数据的新消费者。

服务间紧密耦合

在图 8.1 中，你可以看到许可服务和组织服务之间的紧密耦合。许可服务总是依赖于组织服务来检索数据。然而，如果组织服务在组织记录被更新或删除时直接与许可服务通信，那么你就已经将组织服务的耦合引入到许可服务中了。为了使 Redis 缓存数据无效，组织服务需要调用在许可服务暴露的一个端点使它的 Redis 缓存失效，或组织服务必须直接与许可服务所拥有的 Redis 服务器通信，来清除它里面的数据。

组织服务与 Redis 通信有其自身的问题，因为你正在与一个被其他服务直接拥有的数据存储交互。在一个微服务环境中，这是一大禁忌。虽然你可以认为组织数据正确地属于组织服务，但许可服务在特定的上下文中使用它，并可能在围绕数据进行转换或构建业务规则。组织服务直接与 Redis 服务交互，可能不小心打破拥有已经实现了许可服务的团队的规则。

服务之间的脆弱性

许可服务和组织服务之间的紧密耦合也带来了两种服务之间的脆弱性。如果许可服务停止或运行缓慢，组织服务可能受影响因为组织服务现在直接与许可服务通信。再说，如果组织服务直接与许可服务的 Redis 数据存储交互，你现在已经在组织服务和 Redis 之间创建了依赖。在这种情况下，共享 Redis 服务器的任何问题，现在都有可能降低这两个服务。

在为组织服务中的更改添加新的消费者时不灵活

这个架构的最后一个问题是它的灵活性。使用图 8.1 中的模型，如果组织数据更改时有另一个服务感兴趣，则需要往组织服务添加一个向另一个服务的调用。这意味着组织服务的代码更改和重新部署。如果你使用同步、请求-响应模型进行传递状态更改，那么你开始看到应用程序核心服务和其他服务之间几乎存在网状模式的依赖。这些 Web 站点的中心成为你应用程序中的主要故障点。

另一种耦合

然而消息在服务之间增加一个间接层，你仍然可能在两个服务之间通过使用消息而引入紧耦合。在本章后面，你将在组织服务和许可服务之间发送消息。这些信息都会被序列化和反序列化到一个使用 JSON 作为消息传输协议的 Java 对象。如果两个服务不能优雅地处理同一消息类型的不同版本，当在 Java 来回转换时对 JSON 消息结构的变化会导致问题。JSON 并不原生支持版本管理。然而，如果你需要版本管理，你可以使用 Apache Avro (https://avro.apache.org/)。Avro 是一个二进制协议，内置了版本管理。Spring Cloud Stream 支持 Apache Avro 作为消息传递协议。然而，使用 Avro 超出了本书的范围，但是我们真的想让你知道，它确实有帮助，如果你真的担心消息版本管理。

8.1.2. 在服务之间使用消息传递状态改变

用消息的方式，你可以将队列添加在许可服务和组织服务之间。此队列不用于读取组织服务中的数据，但当组织服务管理的组织数据发生任何状态变化时，组织服务将使用该队列来发布数据。图 8.2 演示了这种方法。

① When the organization service communicates state changes, it publishes a message to a queue.

当组织服务传递状态更改时，它将消息发布到队列中。

② The licensing service monitors the queue for any messages published by the organization service and can invalidate the Redis cache data as needed.

许可服务监视由组织服务发布到队列的任何消息，并在需要的时候使 Redis 缓存数据失效。

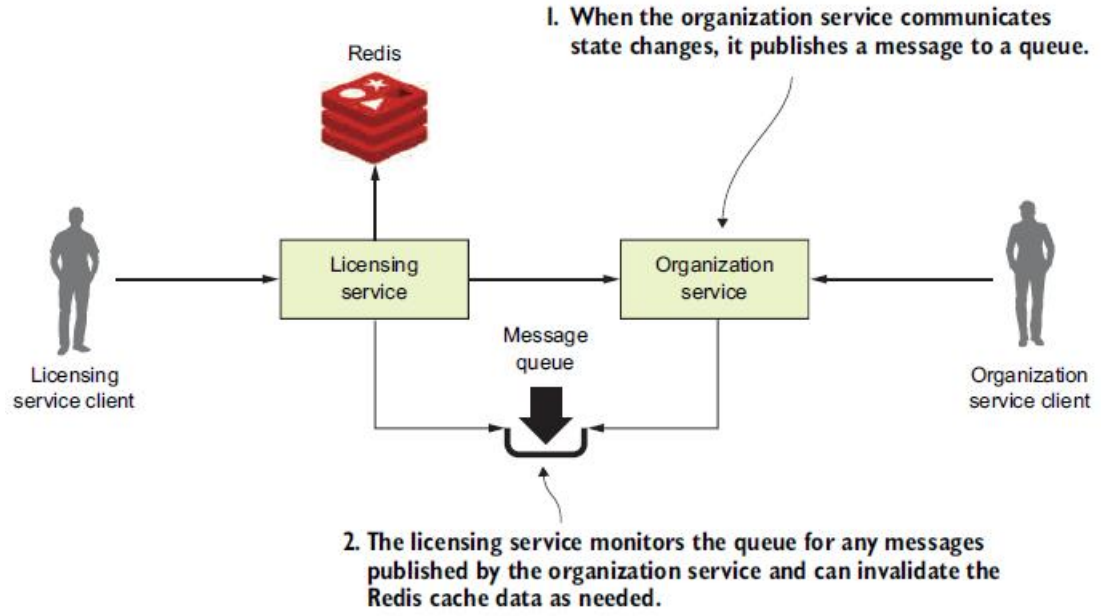

图 8.2 当组织状态改变，消息将被写入位于两个服务之间的消息队列中。

在图 8.2 中的模型中，每当组织数据发生变化时，组织服务都将消息发布到队列中。许可服务正在监视队列中的消息和消息到达时间，并将适当的组织记录清除出 Redis 缓存。当涉及到通信状态时，消息队列充当许可服务和组织服务之间的中介。这种方法有四个好处：

- 松耦合
- 持久性
- 可扩展性
- 灵活性

松耦合

微服务应用程序可以由几十个小型和分布式服务组成，这些服务必须相互交互，并且对

彼此管理的数据感兴趣。正如前面所提出的同步设计所看到的，同步 HTTP 响应在许可服务和组织服务之间创建了一个硬依赖关系。我们不能完全消除这些依赖关系，但是我们只能通过直接管理服务所拥有数据的端点来尽量减少依赖关系。消息传递方式允许你将这两个服务解耦，因为当它涉及到通信状态的变化，也不知道对方的服务。当组织服务需要发布状态更改时，它将消息写入队列。许可服务只知道它收到消息，不知道是谁发布了消息。

持久性

队列的存在允许你保证即使服务消费者已停止，消息也将被传递。即使许可服务不可用，组织服务也可以继续发布消息。消息将存储在队列中，并将一直保留到许可服务可用为止。相反，如果将缓存和队列方法结合在一起，如果组织服务已关闭，则许可服务可以优雅地降级，因为至少部分组织数据将在其缓存中。有时，旧的数据比没有数据更好。

可扩展性

由于消息存储在队列中，消息的发送方不必等待消息消费者返回的响应。他们可以继续使用他们的方式和继续工作。同样，如果读取队列上的消息的消费者没有足够快地处理消息，那么要吸引更多的消费者并让他们处理队列中的消息，这是一项无价值的任务。这种可伸缩性方法非常适合于微服务模型，因为我在本书中强调的一点是，它可以很简单地启动微服务的新实例，并使额外的微服务成为另一个服务，它可以处理包含消息的消息队列。这是水平扩展的示例。从队列中读取消息的传统的扩展机制涉及增加消息消费者一次可以处理的线程数量。不幸的是，使用这种方法，你最终受限于消息消费者可用的 CPU 数量。微服务模型没有这种限制，因为你通过增加承载消费消息的服务的机器数量来进行扩展。

灵活性

消息的发送者不知道谁会使用它。这意味着你可以在不影响原始发送服务的情况下轻松地添加新的消息消费者（和新功能）。这是一个非常强大的概念，因为新功能可以添加到应

用程序，而无需接触到现有服务。相反，新代码可以监听正在发布的事件，并相应地对它们作出响应。

8.1.3. 消息架构的缺点

与任何架构模型一样，基于消息的架构具有权衡性。一种基于消息的架构可以是复杂的并需要开发团队关注的几个关键事情，包括：

- 消息处理的语义
- 消息的可见性
- 消息的编排

消息处理的语义

在微服务应用程序中使用消息不仅仅需要理解如何发布和消费消息。它要求您了解应用程序将如何根据消息的顺序来执行操作，以及如果消息按乱序处理，会发生什么情况。例如，如果你有严格的要求，从单个客户的所有订单必须按照收到的顺序进行处理，那么你将不得不设置不同的消息处理结构，而不是每一条消息都可以彼此独立地消费。

它还意味着，如果你使用消息传递来执行数据的严格状态转换，则需要在设计应用程序时考虑到消息引发异常的场景，或者按错误的顺序处理错误。如果消息失败，你重试处理错误或你让它失败？如果其中一条客户信息失败，你如何处理与该客户相关的未来消息？再说一遍，这些都是需要思考的话题。

消息的可见性

在微服务中使用消息通常意味着异步服务调用和异步处理的混合。消息的异步特性意味着在消息发布或消费时，它们可能不会被接收或处理。另外，有像跨 Web 服务调用跟踪用户的交易关联 ID 和消息对于理解和调试应用程序中的内容至关重要。正如你所记得的，在

第 6 章中，关联 ID 是在用户交易开始时生成并与每个服务调用一起传递的唯一编号。它还应该与发布和消费的每个消息一起传递。

消息的编排

正如在消息可见性部分中提到的，基于消息的应用程序使它更难的原因，通过他们的应用程序的业务逻辑代码，因为他们不再以线性方式进行一个简单的块请求响应模型。相反，调试基于消息的应用程序可能涉及到几个不同服务的日志，这些日志可以在不同的时间和没有顺序执行用户交易。

消息可以是复杂而强大的

前面的部分并不是为了让你对在应用程序中使用消息传递感到害怕。相反，我的目标是强调在你的服务总使用消息需要深谋远虑。我最近完成了一个主要项目，我们需要为每一个客户带来一组有状态的 AWS 服务器实例。我们需要通过使用 AWS 简单队列服务（SQS）和 Kafka 整合微服务调用的组合和消息。虽然这个项目很复杂，但我亲眼看到了消息传递的力量，当项目结束时，我们意识到我们需要处理在服务器终止之前必须确保从服务器取回某些文件的问题。这一步必须通过用户工作流进行大约 75%的工作，整个过程在流程完成之前不能继续。幸运的是，我们有一个微服务（称为我们的文件恢复服务）可以做很多工作来检查和查看文件从服务器是否已被弃用。因为服务器通过事件传递它们所有的状态变化（包括已被弃用的），我们只需要将文件恢复服务器插入到一个来自已弃用的服务器的事件流，并让他们监听“decommissioning”事件。

如果整个过程是同步的，添加这个文件耗尽步骤将是非常痛苦的。但最终，我们需要一个现有的生产服务，来监听来自现有消息队列的事件并作出反应。这项工作是在几天内完成的，我们在项目交付过程中从未遗漏过一次。消息允许你将服务连接在一起，而无需在基于代码的工作流中硬编码服务。

8.2. 介绍 Spring Cloud Stream

Spring Cloud 可以轻松地将消息集成到基于 Spring 的微服务中。它通过 Spring Cloud Stream 项目实现这一目标(https://cloud.spring.io/spring-cloud-stream/)。Spring Cloud Stream 项目是一个注解驱动的框架，它允许你在 Spring 应用程序中轻松创建消息发布者和消费者。

Spring Cloud Stream 还允许你抽象你正在使用的消息传递平台的实现细节。多个消息平台可以与 Spring Cloud Stream(包括 Apache Kafka 项目和 RabbitMQ)一起使用，并且平台的具体实现细节被保留在应用程序代码之外。应用程序中的消息发布和消费的实现是通过平台无关的 Spring 接口实现的。

> ***注意***:在本章中，你将使用一个称为 Kafka(https://kafka.apache.org/)的轻量级消息总线。Kafka 是一个轻量级的，高效的消息总线，它允许你从一个应用到一个或多个应用程序异步发送消息流。使用 Java 编写，Kafka 已经成为许多基于云的应用程序的事实消息总线，因为它具有高度的可靠性和可伸缩性。Spring Cloud Stream 也支持使用 RabbitMQ 作为消息总线。Kafka 和 RabbitMQ 是强大的消息平台，我选择了 Kafka，因为那是我最熟悉的。

为了理解 Spring Cloud Stream，让我们首先讨论 Spring Cloud Stream 体系结构，熟悉 Spring Cloud Stream 的术语。如果你以前从未使用过基于消息的平台，那么所涉及到的新术语可能会有些令人难以理解。

8.2.1. Spring Cloud Stream 体系结构

让我们通过消息传递的两个服务的镜头来查看 Spring Cloud Stream 体系结构，从而开始我们的讨论。一个服务将是消息发布者，一个服务将是消息消费者。图 8.3 显示了 Spring Cloud Stream 是如何被用于帮助传递消息的。

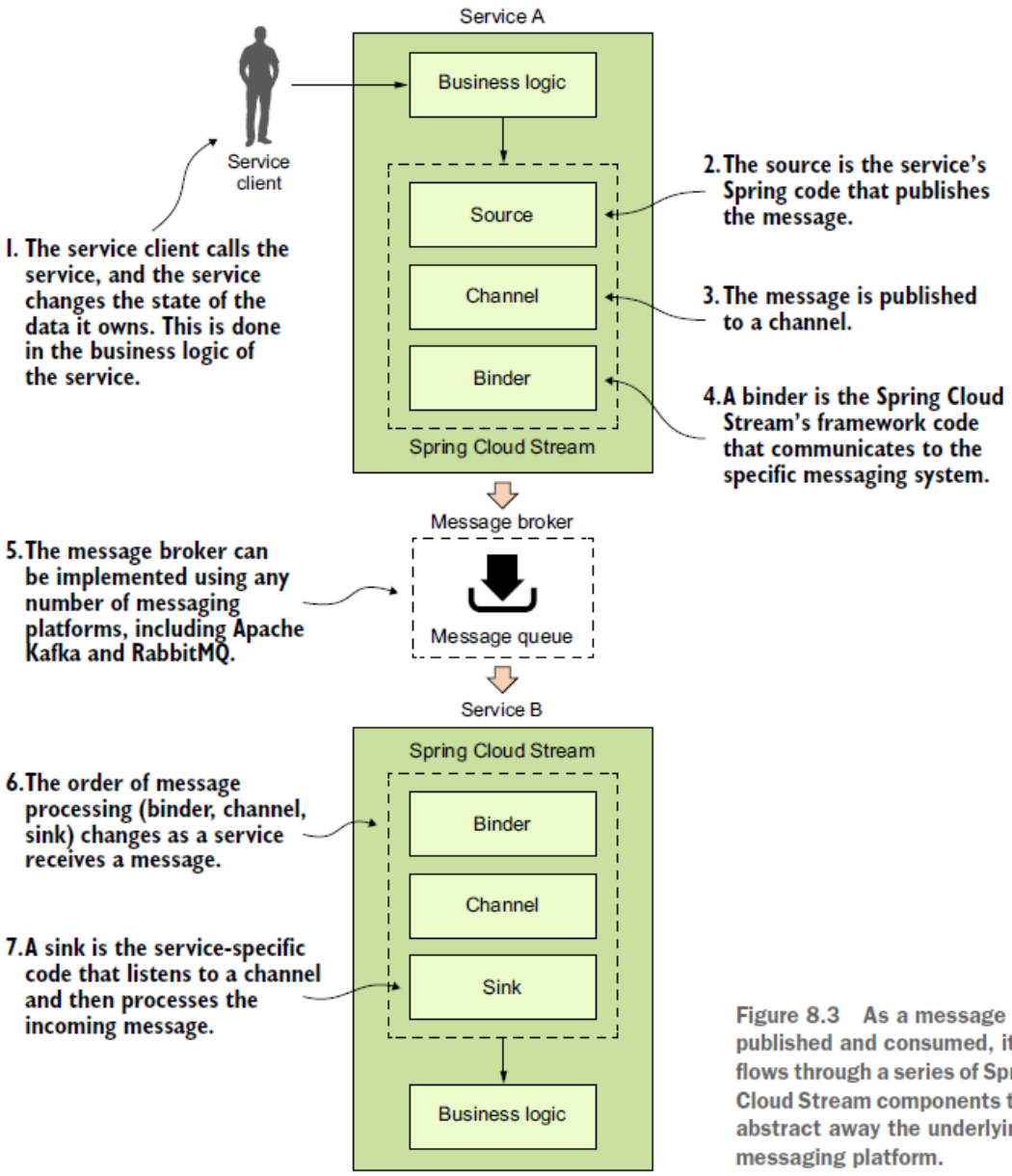

Figure 8.3 As a message is published and consumed, it flows through a series of Spring Cloud Stream components that abstract away the underlying messaging platform.

图 8.3 当消息被发布和消费时，它流经一系列抽象底层消息平台的 Spring Cloud Stream 组件。

① The service client calls the service, and the service changes the state of the data it owns. This is done in the business logic of the service.

服务客户端调用服务，服务更改它拥有的数据的状态。这是在服务的业务逻辑中完成的。

② The source is the service's Spring code that publishes the message.

消息发布的源头是服务的 Spring 代码。

③ The message is published to a channel.

消息被发布到通道中。

④ A binder is the Spring Cloud Stream' s framework code that communicates to the specific messaging system.

绑定器是 Spring Cloud Stream 的框架代码，它与特定的消息传递系统通信。

⑤ The message broker can be implemented using any number of messaging platforms, including Apache Kafka and RabbitMQ.

消息代理可以使用任意数量的消息平台实现，包括 Apache Kafka 和 RabbitMQ。

⑥ The order of message processing (binder, channel, sink) changes as a service receives a message.

消息处理（绑定器、通道、接收器）的顺序随着服务接收消息而发生变化。

⑦ A sink is the service-specific code that listens to a channel and then processes the incoming message.

接收器是特定于服务的代码，它监听通道并处理传入消息。

随着 Spring Cloud 中消息的发布和消费，在消息的发布和消费中涉及到四个组件：

- 消息源
- 通道
- 绑定器
- 接收器

消息源

当服务准备发布消息时，它将使用消息源来发布消息。消息源是一个 Spring 注解的接口，用普通的 Java 对象（POJO）代表被发布的消息。消息源将消息序列化（默认的序列

化格式是 JSON），并发布消息到通道。

通道

通道是队列的一种抽象，它在消息生产者发布或由消息消费者消费后将保留消息。通道名称总是与目标队列名称相关联。但是，该队列名从不直接暴露在代码中。相反，在代码中使用通道名称，这意味着你可以通过更改应用程序的配置而不是应用程序的代码来更改通道读或写的队列。

绑定器

绑定器是 Spring Cloud Stream 框架的一部分。它是与特定的消息平台进行通信的 Spring 代码。Spring Cloud Stream 框架的绑定器部分允许你处理消息，而不必向用于发布和消费消息的特定平台库和 API 暴露。

接收器

在 Spring Cloud Stream 中，当服务从队列接收消息时，它通过接收器执行。接收器监听传入的消息通道和将消息反序列化为一个普通的 Java 对象。从那里，消息可以由 Spring 服务的业务逻辑处理。

8.3. 写一个简单的消息生产者和消费者

现在我们已经了解了 Spring Cloud Stream 中的基本组件，让我们来看一个简单的 Spring Cloud Stream 示例。对于第一个示例，你将组织服务的消息传递给许可服务。你在许可服务处理消息的唯一事情就是打印日志消息到控制台。

此外，因为在这个示例中，你只需要一个 Spring Cloud Stream 源（消息生产者）和接收器（消息消费者），你将以几个简单的 Spring Cloud 快捷方法开始示例，该快捷方法将在组织服务中设置消息源，并在许可服务中设置一个接收器。

8.3.1. 在组织服务中编写消息生产者

首先要修改组织服务，以便每次添加、更新或删除组织数据，组织服务将向 Kafka 主题发布消息，表明组织更改事件已经发生。图 8.4 突出显示了消息生产者，并从图 8.3 中构建了 Spring Cloud Stream 的总体结构。

发布的消息将包含与更改事件相关的组织 ID，还将包括发生的操作（添加、更新或删除）。

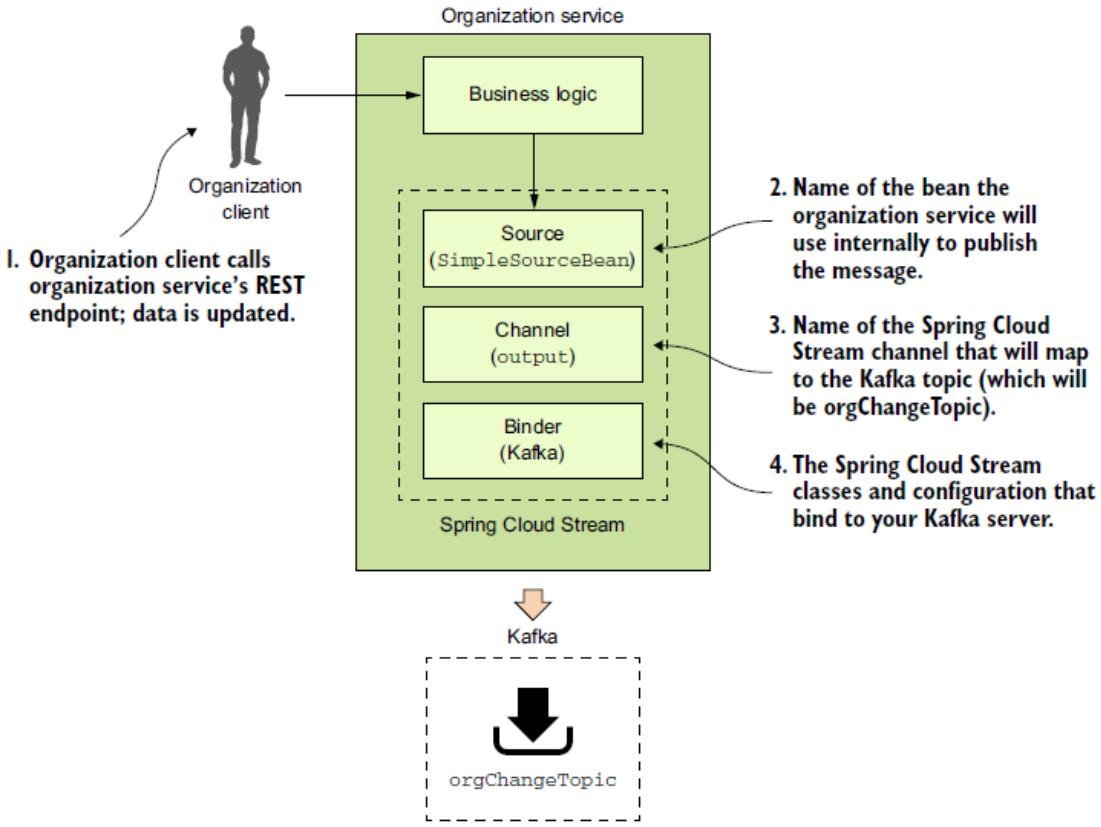

图 8.4 当组织服务数据更改时，它将向 Kafka 发布消息。

① Organization client calls organization service's REST endpoint; data is updated.

组织客户端调用组织服务的 REST 端点；数据被更新。

② Name of the bean the organization service will use internally to publish the message.

组织服务将在内部使用的 bean 的名称来发布消息。

③ Name of the Spring Cloud Stream channel that will map to the Kafka topic (which will be orgChangeTopic).

Spring Cloud Stream 通道的名称将映射到 Kafka 的主题名称(这将是 orgChangeTopic)。

④ The Spring Cloud Stream classes and configuration that bind to your Kafka server.

Spring Cloud Stream 类和绑定到 Kafka 服务器的配置。

你需要做的第一件事是在组织服务的 Maven pom.xml 文件设置你的 Maven 的依赖。pom.xml 文件可以在组织服务目录中找到。在 pom.xml 文件，你需要添加两个依赖：一个是 Spring Cloud Stream 的核心库，另一个是包括 Spring Cloud Stream Kafka 库：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

一旦 Maven 的依赖已被定义，你需要告诉你的应用程序，它将绑定到 Spring Cloud Stream 消息代理。你通过使用 @EnableBinding 注解注释组织服务的引导类 organization-service/src/main/java/com/thoughtmechanix/organization/Application.java 来实现这一点。下面的清单显示了组织服务的 Application.java 类源代码。

清单 8.1 带注释的 Application.java 类

```
package com.thoughtmechanix.organization;

import com.thoughtmechanix.organization.utils.UserContextFilter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
```

```
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Source.class)
public class Application {
    @Bean
    public Filter userContextFilter() {
        UserContextFilter userContextFilter = new UserContextFilter();
        return userContextFilter;
    }

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

①@EnableBinding 注解告诉 Spring Cloud Stream 将应用程序绑定到消息代理。

在清单 8.1 中，@EnableBinding 注解告诉 Spring Cloud Stream 你希望将服务绑定到消息代理。在@EnableBinding 注解中 Source.class 类的使用告诉 Spring Cloud Stream，这个服务将通过定义在 Source 中的一组通道与消息代理通信。记住，通道位于消息队列的上方。Spring Cloud Stream 有一组默认的通道，可以配置为对消息代理进行会话。

此时，你还没有告诉 Spring Cloud Stream 你希望组织服务绑定到什么消息代理。我们很快就会明白的。现在，你可以继续实现发布消息的代码。

消息发布代码可以在 organization-service/src/com/thoughtmechanix/organization/events/source/SimpleSourceBean.java 类中找到。下面的清单显示了该类的代码。

清单 8.2 向消息代理发布消息

```
package com.thoughtmechanix.organization.events.source;

@Component
public class SimpleSourceBean {
```

```
    private Source source;
    private static final Logger logger = LoggerFactory.getLogger(SimpleSourceBean.class);

    @Autowired
    public SimpleSourceBean(Source source){
        this.source = source;
    }

    public void publishOrgChange(String action,String orgId){
        logger.debug("Sending Kafka message {}
            ➥ for Organization Id: {}",
            ➥action, orgId);

        OrganizationChangeModel change = new OrganizationChangeModel(
            OrganizationChangeModel.class.getTypeName(),
            action,
            orgId,
            UserContext.getCorrelationId());

        source.output()
                .send(MessageBuilder.withPayload(change).build());
    }
}
```

①Spring Cloud Stream 将注入一个 Source 接口实现，供服务使用。

②被发布的消息是一个 Java 的 POJO。

③当你准备好要发送的消息，使用在 Source 类定义的通道的 send()方法。

在清单 8.2 中，你将 Spring Cloud Source 类注入到代码中。记住，特定消息主题的所有通信都是通过一个称为通道的 Spring Cloud Stream 结构来实现的。通道是由 Java 接口类来表示。在这个清单中，你正在使用 Source 接口。Source 接口是一个 Spring Cloud 定义的接口，它暴露一个称为 output()的单一方法。当你的服务只需要发布到一个单独的通道时，Source 接口是一个方便的接口。output()方法返回一个类型为 MessageChannel 的类。MessageChannel 是如何将消息发送到消息代理。在本章后面，我将向你展示如何使用自定义接口暴露多个消息传递通道。

消息实际的发布发生在 publishOrgChange()方法。这种方法创建了一个称为 OrganizationChangeModel 的 Java POJO。我不打算把 OrganizationChangeModel 的代码放在本章里，因为这个类仅仅是围绕三个数据元素的 POJO：

- 动作：这是触发事件的动作。我已经在消息中包含了动作，以便给消息消费者更多

的上下文，说明它应该如何处理事件。

- 组织 ID：这是与事件相关联的组织 ID。
- 关联 ID：这是触发事件的服务调用的关联 ID。你应该总是在事件中包含相关 ID，因为它有助于跟踪和调试通过服务的消息流。

当你准备好发布消息，使用从 source.output()方法返回的 MessageChannel 类的 send()方法。

```
source.output().send(MessageBuilder.withPayload(change).build());
```

send()方法获得一个 Spring 消息类。你使用一个称为 MessageBuilder 的 Spring 辅助类来获取 OrganizationChangeModel 类的内容，并将其转换为 Spring 消息类。

这是你发送消息所需的所有代码。然而，在这一点上，每件事情都应该感觉有点像魔术，因为你还没有看到如何将组织服务绑定到特定的消息队列，更不用说实际的消息代理了。这一切都是通过配置。清单 8.3 显示了映射服务的 Spring Cloud Stream 源到 Kafka 消息代理和 Kafka 中的消息主题的配置。该配置信息可以定位在服务的 application.yml 文件或在服务的 Spring Cloud 配置项中。

清单 8.3 用于发布消息的 Spring Cloud Stream 配置

```
spring:
  application:
    name: organizationservice
  #Remove for conciseness
    stream:
      bindings:
        output:
          destination: orgChangeTopic
          content-type: application/json
      kafka:
        binder:
          zkNodes: localhost
          brokers: localhost
```

①stream.bindings 是服务向 Spring Cloud Stream 消息代理发布消息所需的配置的开始。

②output 是你的通道和映射到清单 8.2 中所看到的 Source.output()通道的名称。

③orgChangeTopic 是要写入消息的消息队列（或主题）的名称。

④content-type 向将要发送和接收的消息类型的 Spring Cloud Stream 提供了提示（在本例中是 JSON）。

⑤stream.bindings.kafka 属性告诉 Spring 你打算用 Kafka 作为服务消息总线（你可以使用 RabbitMQ 作为替代）。

⑥zknodes 和 brokers 属性告诉 Spring Cloud Stream 你的 Kafka 和 ZooKeeper 的网络位置

清单 8.3 中的配置看起来很密集，但很简单。清单中的配置属性 spring.stream.bindings.output 将清单 8.2 中的 source.output()通道映射到要与之通信的消息代理 orgChangeTopic。它还告诉 Spring Cloud Stream 发送到这个主题的消息应该序列化为 JSON。Spring Cloud Stream 可以序列化多种格式的消息，包括 JSON，XML 和 Apache 基金会的 Avro 格式(https://avro.apache.org/)。

清单 8.3 中的配置属性 spring.stream.bindings.kafka ,也告诉 Spring Cloud Stream 将服务绑定到 Kafka。子属性告诉 Spring Cloud Stream Kafka 消息代理的网络地址和 Apache ZooKeeper 服务器与 Kafka 一起运行。

现在你已经编写了将通过 Spring Cloud Stream 发布消息的代码和配置，告诉 Spring Cloud Stream 它将使用 Kafka 作为消息代理 ,让我们看看组织服务中消息发布的实际位置。这项工作将在 organization-service/src/main/java/com/thoughtmechanix/organization/services/OrganizationService.java 类中完成。下面的清单显示了该类的代码。

清单 8.4 在组织服务中发布消息

```
package com.thoughtmechanix.organization.services;

@Service
public class OrganizationService {
    @Autowired
    private OrganizationRepository orgRepository;

    @Autowired
    SimpleSourceBean simpleSourceBean;

    public void saveOrg(Organization org){
        org.setId( UUID.randomUUID().toString());

        orgRepository.save(org);
        simpleSourceBean.publishOrgChange("SAVE", org.getId());
    }
}
```

①Spring 自动装配是用来为你的组织服务注入 SimpleSourceBean。

②用于更改组织数据的服务中的每个方法，调用 simpleSourceBean.publishOrgChange()。

我应该在消息中放什么数据？

当团队第一次开始他们的消息之旅时，我得到的最常见的问题之一就是他们的消息到底应该包含多少数据。我的答案是，这取决于你的应用程序。你可能注意到，在我所有的示例中，我只返回已更改的组织记录的组织 ID。我从来没有把对数据更改的一份副本放到消息中。在我的示例中（以及在电话空间中处理的许多问题中），执行的业务逻辑对数据的变化非常敏感。我使用基于系统事件的消息告诉其他服务，数据状态已经改变，但我总是强迫其他服务返回给主服务（拥有数据的服务）来检索数据的新副本。这种方法在执行时间上更昂贵，但它也保证我总是有最新的数据副本来工作。机会仍然存在，在你从源系统读取数据之后，你所处理的数据可能会立即改变，但这比盲目地从队列中直接消费消息的可能性小得多。

仔细考虑你传递了多少数据。迟早，你会遇到一种情况，你所传递的数据是陈旧的。它可能是过时的，因为一个问题导致它在消息队列中待得太久，或者前面的消息包含数据失败，而现在传递的数据表示不一致状态的数据（因为应用程序依赖于消息的状态而不是底层数据存储中的实际状态。）。如果你要在消息中传递状态，也要确保在消息中包含日期、时间戳或版本号，以便消费这些数据的服务可以检查所传递的数据，并确保它不比已经拥有的数据的副本更旧。（记住，数据可以不按顺序检索。）

8.3.2. 在许可服务中编写消息消费者

此时，你已经修改了组织服务，以便每当组织服务更改组织数据时，都会向 Kafka 发布消息。任何有兴趣的人都可以做出反应，而不必由组织服务明确地调用。这也意味着您可以通过让组织服务监听消息队列中的消息来轻松添加能够对组织服务中的更改作出响应的新功能。现在让我们换方向，看看服务如何使用 Spring Cloud Stream 消费消息。

在这个例子中，你将要让许可服务使用组织服务发布的消息。图 8.5 显示了许可服务将

适用于图 8.3 中所示的 Spring Cloud 架构的位置。

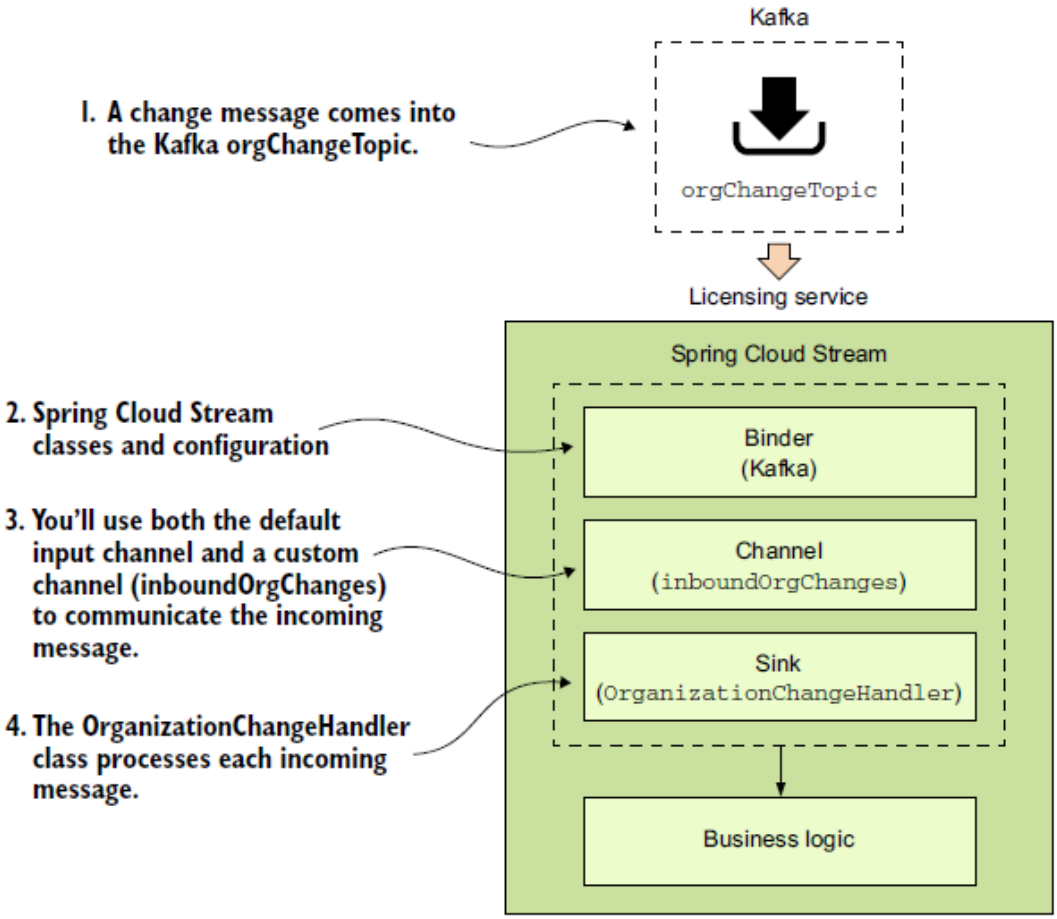

图 8.5 当一条消息进入 Kafka orgChangeTopic 时，许可服务会响应。

① *A change message comes into the Kafka orgChangeTopic.*

改变信息进入 Kafka orgChangeTopic。

② *Spring Cloud Stream classes and configuration*

Spring Cloud Stream 类和配置

③ *You' ll use both the default input channel and a custom channel (inboundOrgChanges) to communicate the incoming message.*

你将同时使用默认输入通道和自定义通道（inboundOrgChanges）来传递传入消息。

④ *The OrganizationChangeHandler class processes each incoming message.*

OrganizationChangeHandler 类处理每个传入消息。

首先，你需要再次将 Spring Cloud Stream 依赖添加到许可服务 pom.xml 文件中。这个 pom.xml 文件可以在本书源代码的许可服务目录中找到。与前面所看到的组织服务 pom.xml 文件类似，你将添加以下两个依赖项：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

然后，你需要告知许可服务，它需要使用 Spring Cloud Stream 绑定到消息代理。与组织服务一样，我们将使用 @EnableBinding 注解来注释许可服务引导类 (licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java)。许可服务和组织服务之间的区别是你要传递给@EnableBinding 注解的值，如下面的清单所示。

清单 8.5 使用 Spring Cloud Stream 消费消息

```
package com.thoughtmechanix.licenses;

@EnableBinding(Sink.class)
public class Application {

    @StreamListener(Sink.INPUT)
    public void loggerSink(OrganizationChangeModel orgChange)
        logger.debug("Received an event for organization id {}",
            orgChange.getOrganizationId());
    }
}
```

①@EnableBinding 注解告诉服务使用 Sink 接口中定义的通道来监听传入的消息。

②每当收到来自输入通道的消息时，Spring Cloud Stream 将执行此方法。

由于许可服务是消息的消费者，因此你将传递给@EnableBinding 注解的值是 Sink.class。这告诉 Spring Cloud Stream 使用默认的 Spring Sink 接口绑定到消息代理。与 8.3.1 节中介绍的 Spring Cloud Steam Source 接口类似，Spring Cloud Stream 在 Sink 接口上暴露一个默认通道。Sink 接口上的通道称为 input，用于监听通道上的传入消息。

一旦你已经定义你想通过@EnableBinding 注解来收听消息，你可以编写代码来处理来自 Sink 输入通道的消息。为此，请使用 Spring Cloud Stream @StreamListener 注解。

@StreamListener 注解告诉 Spring Cloud Stream 在每次从输入通道接收到消息时执行 loggerSink()方法。Spring Cloud Stream 会自动将来自通道的消息反序列化为名为 OrganizationChangeModel 的 Java POJO。

再次，消息代理主题到输入通道的实际映射在许可服务的配置中完成。对于许可服务，其配置如下所示，可以在许可服务的 licensing-service/src/main/resources/application.yml 文件中找到。

清单 8.6 将许可服务映射到 Kafka 中的消息主题

```
spring:
  application:
    name: licensingservice
  cloud:
    stream:
      bindings:
        input:
          destination: orgChangeTopic
          content-type: application/json
          group: licensingGroup
        binder:
          zkNodes: localhost
          brokers: localhost
```

①spring.cloud.stream.bindings.input 属性将输入通道映射到 orgChangeTopic 队列。

②group 属性用于保证服务的过程一次语义。

此清单中的配置与组织服务的配置类似。但是，它有两个关键的区别。首先，您现在在 spring.cloud.stream.bindings 属性下定义了一个称为 input 的通道。该值映射到清单 8.5 代码中定义的 Sink.INPUT 通道。该属性将输入通道映射到 orgChangeTopic。其次，你将看到称为 spring.cloud.stream.bindings.input.group 的新属性的引入。group 属性定义将消费该消息的消费者组的名称。

消费者组的概念是这样的：你可能有多个服务，每个服务有多个实例监听同一个消息队列。你希望每个独特的服务处理消息的副本，但只希望一组服务实例中的一个服务实例消费

和处理消息。group 属性标识该服务所属的使用者组。只要所有服务实例具有相同的组名，Spring Cloud Stream 和底层消息代理将保证只有一个消息副本将被属于该组的服务实例使用。对于你的许可服务，组属性值将被称为 licensingGroup。

图 8.6 说明了如何使用消费者组来帮助实施跨多个服务消费消息的一次消费语义。

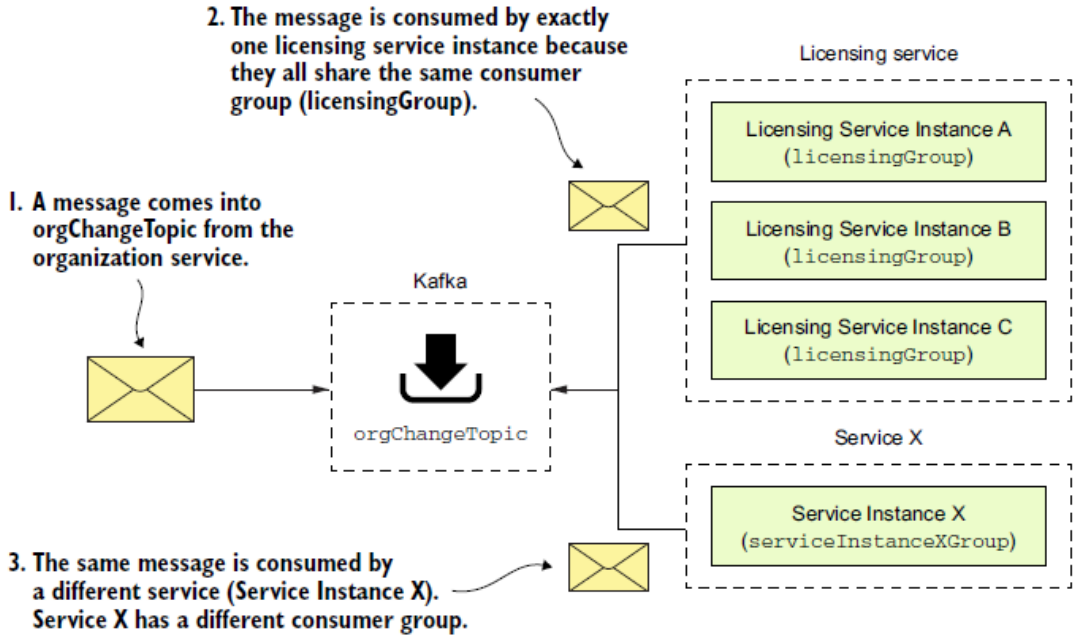

图 8.6 消费者组保证消息只能被一组服务实例处理一次。

① A message comes into orgChangeTopic from the organization service.

消息从组织服务进入 orgChangeTopic。

② The message is consumed by exactly one licensing service instance because they all share the same consumer group (licensingGroup).

该消息正好由一个许可服务实例使用，因为它们都共享相同的消费者组(licensingGroup)。

③ The same message is consumed by a different service (Service Instance X). Service X has a different consumer group.

相同的消息被不同的服务（服务实例 X ）消费。 X 服务有不同的消费群体。

8.3.3. 查看运行中的消息服务

此时，组织服务每次添加，更新或删除记录时都会向 orgChangeTopic 发布消息，并且许可服务会接收同一主题的消息。现在，你将通过更新组织服务记录并观察控制台，以查看许可服务中显示的相应日志消息来看到此代码。

要更新组织服务记录，你将在组织服务上发出 PUT 以更新组织的联系电话号码。你将使用 http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a 端点进行更新。你要发送的 PUT 调用到端点的报文体是：

```
{
    "contactEmail": "mark.balster@custcrmco.com",
    "contactName": "Mark Balster",
    "contactPhone": "823-555-2222",
    "id": "e254f8c-c442-4ebe-a82a-e2fc1d1ff78a",
    "name": "customer-crm-co"
}
```

图 8.7 显示了这个 PUT 调用的返回输出。

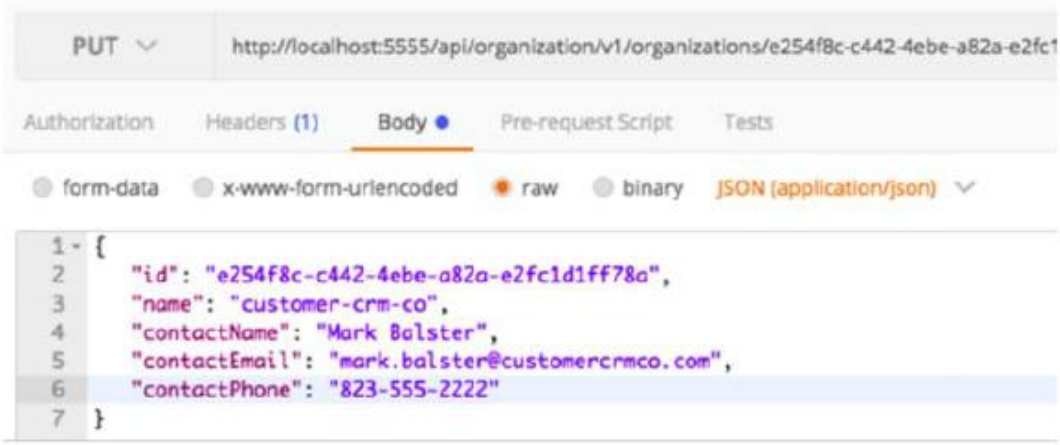

图 8.7 使用组织服务更新联系人电话号码

组织服务调用完成后，你应该在运行服务的控制台窗口中看到以下输出。图 8.8 显示了这个输出。

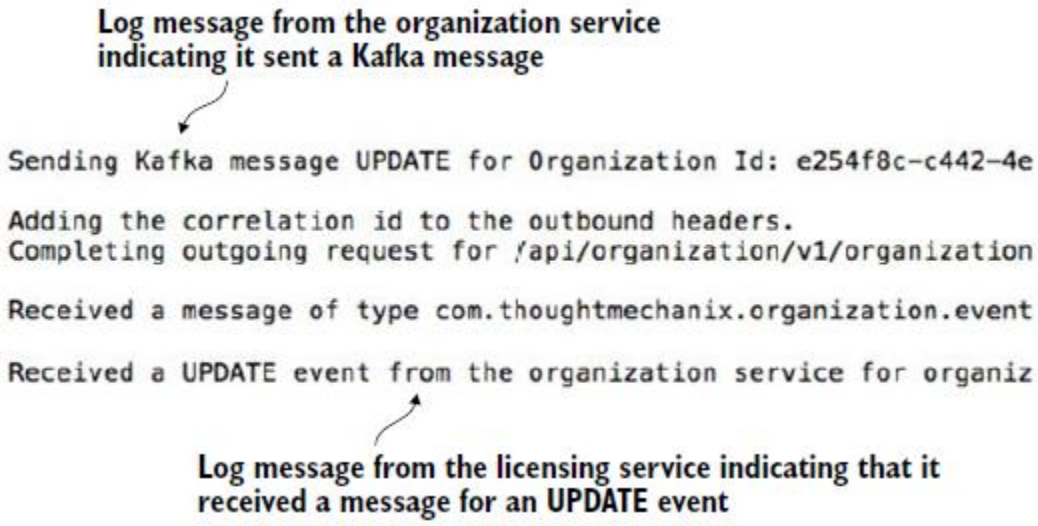

图 8.8 控制台显示组织服务发送并接收到的消息。

① *Log message from the organization service indicating it sent a Kafka message*

来自组织服务的日志消息表明它发送了 Kafka 消息

② *Log message from the licensing service indicating that it received a message for an UPDATE event*

记录来自许可服务的消息，表明它收到 UPDATE 事件的消息

现在你有两个使用消息传递的服务。Spring Cloud Stream 充当这些服务的中间人。从消息传递的角度来看，服务对彼此一无所知。它们使用消息代理作为中介和 Spring Cloud Stream 作为消息传递代理的抽象层进行通信。

8.4. Spring Cloud Stream 使用案例：分布式缓存

此刻，你有两个服务与消息传递，但你真的没有使用消息做任何事情。现在你将构建本章前面讨论的分布式缓存示例。你将有许可服务始终检查与特定许可证关联的组织数据的分布式 Redis 缓存。如果组织数据存在于缓存中，则会从缓存中返回数据。如果没有，你将调用组织服务并将调用的结果缓存在 Redis 哈希中。

在组织服务中更新数据时，组织服务将向 Kafka 发送消息。许可服务将收到消息并针对 Redis 发出删除以清除缓存。

云缓存和消息

使用 Redis 作为分布式缓存与云中的微服务开发非常相关。与我现在的雇主一起，我们使用亚马逊的 Web 服务构建解决方案（AWS），是亚马逊 DynamoDB 的重要用户。我们也使用亚马逊的 ElastiCache（Redis）来：

- *提高查找常用数据的性能*：通过使用缓存，我们已经显著改善了我们几个关键服务的性能。我们销售的产品中的所有表格都是多租户（将多个客户记录保存在一个表格中），这意味着它们可能相当大。因为高速缓存趋向于“严重”使用数据，所以通过使用 Redis 和高速缓存以避免读取到 Dynamo，我们看到了显着的性能提升。
- *减少持有我们数据的 Dynamo 表上的负载（和成本）*：在 Dynamo 中访问数据可能是一个昂贵的建议。你所做的每一个读取都是一个花费时间成本的事件。使用 Redis 服务器读取主键然后读取 Dynamo 要快得多。
- *如果我们的主数据存储（Dynamo）出现性能问题，请提高弹性以便我们的服务可以优雅地降级*：如果 AWS Dynamo 出现问题（偶尔会发生问题），则使用 Redis 等缓存可以帮助您的服务优雅地降级。根据你在缓存中保留的数据量，缓存解决方案可以帮助减少你从数据存储中命中的错误数量。

Redis 不仅仅是一个缓存解决方案，但是如果你需要一个分布式缓存，它可以满足这个角色。

8.4.1. 使用 Redis 缓存查找

现在你将开始设置许可服务以使用 Redis。幸运的是，Spring Data 已经可以将 Redis 引入许可服务。要在许可服务中使用 Redis，你需要执行以下四项操作：

- 配置许可服务以包含 Spring Data Redis 依赖
- 构建到 Redis 的数据库连接
- 定义 Spring Data Redis 存储库，你的代码将用于与 Redis 哈希交互
- 使用 Redis 和许可服务来存储和读取组织数据

配置许可服务使用 Spring Data Redis 依赖

你需要做的第一件事是将 spring-data-redis 依赖项以及 jedis 和 common-pools2 依赖项包含到许可服务的 pom.xml 文件中。下面的清单中显示了包含的依赖。

清单 8.7 添加 Spring Redis 依赖

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-redis</artifactId>
   <version>1.7.4.RELEASE</version>
</dependency>

<dependency>
   <groupId>redis.clients</groupId>
   <artifactId>jedis</artifactId>
   <version>2.9.0</version>
</dependency>

<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-pool2</artifactId>
   <version>2.0</version>
</dependency>
```

将数据库连接构建到 Redis 服务器上

现在你已经有 Maven 的依赖关系，你需要创建一个到你的 Redis 服务器的连接。Spring 使用开源项目 Jedis(https://github.com/xetorthio/jedis)与 Redis 服务器进行通信。为了与特定的 Redis 实例进行通信，你将在 licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java 类中暴露 JedisConnectionFactory 作为 Spring Bean。一旦你连接到 Redis，你将使用该连接来创建一个 Spring RedisTemplate

对象。RedisTemplate 对象将被 Spring Data 存储库类使用，你将在稍后实现查询并保存组织服务数据到你的 Redis 服务。下面的清单显示了这个代码。

清单 8.8 确定你的许可服务将如何与 Redis 进行通信

```
package com.thoughtmechanix.licenses;

import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableBinding(Sink.class)
public class Application {
    @Autowired
    private ServiceConfig serviceConfig;

    @Bean
    public JedisConnectionFactory jedisConnectionFactory() {
        JedisConnectionFactory jedisConnFactory = new JedisConnectionFactory();
        jedisConnFactory.setHostName( serviceConfig.getRedisServer() );
        jedisConnFactory.setPort( serviceConfig.getRedisPort() );
        return jedisConnFactory;
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();
        template.setConnectionFactory(jedisConnectionFactory());
        return template;
    }
}
```

① jedisConnectionFactory()方法设置到 Redis 服务器的实际数据库连接。

②redisTemplate()方法将创建一个 RedisTemplate，用于对你的 Redis 服务器执行操作。

设置许可服务与 Redis 通信的基础工作已经完成。现在让我们转到编写将从 Redis 获取，添加，更新和删除数据的逻辑。

定义 Spring Data Redis 存储库

Redis 是一个键值存储数据存储，其功能类似于一个大的分布式内存哈希映射。在最简单的情况下，它存储数据并通过键查找数据。它没有任何一种复杂的查询语言来检索数据。它的简单性就是它的优势，也是为什么这么多项目将其用于项目中的原因之一。

因为你使用 Spring Data 来访问你的 Redis 存储库，所以你需要定义一个存储库类。你可能还记得，从第二章开始，Spring Data 就使用用户定义的存储库类来为 Java 类提供一个简单的机制来访问 Postgres 数据库，而不必编写低级别的 SQL 查询。

对于许可服务，你将为你的 Redis 存储库定义两个文件。你要写的第一个文件将是一个 Java 接口，将被注入任何需要访问 Redis 的许可服务类中。下面的清单显示了这个接口(licensing-service/src/main/java/com/thoughtmechanix/licenses/repository/OrganizationRedisRepository.java)。

清单 8.9 OrganizationRedisRepository 定义用于调用 Redis 的方法

```
package com.thoughtmechanix.licenses.repository;

import com.thoughtmechanix.licenses.model.Organization;

public interface OrganizationRedisRepository {
    void saveOrganization(Organization org);
    void updateOrganization(Organization org);
    void deleteOrganization(String organizationId);
    Organization findOrganization(String organizationId);
}
```

第二个文件是 OrganizationRedisRepository 接口的实现。licensing-service/src/main/java/com/thoughtmechanix/licenses/repository/OrganizationRedisRepositoryImpl.java 类是该接口的实现，使用你在清单 8.8 列出的 RedisTemplate Spring bean 来与 Redis 服务器交互，并对 Redis 服务器执行操作。下一个清单显示正在使用的这个代码。

清单 8.10 OrganizationRedisRepositoryImpl 实现

```
package com.thoughtmechanix.licenses.repository;

import org.springframework.data.redis.core.HashOperations;
import org.springframework.data.redis.core.RedisTemplate;

@Repository
public class OrganizationRedisRepositoryImpl implements
    OrganizationRedisRepository {
```

①@Repository 注解告诉 Spring，这个类是一个与 Spring Data 一起使用的 Repository 类。

```
    private static final String HASH_NAME="organization";

    private RedisTemplate<String, Organization> redisTemplate;
    private HashOperations hashOperations;

    public OrganizationRedisRepositoryImpl(){
        super();
    }

    @Autowired
    private OrganizationRedisRepositoryImpl(RedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    @PostConstruct
    private void init() {
        hashOperations = redisTemplate.opsForHash();
    }

    @Override
    public void saveOrganization(Organization org) {
        hashOperations.put(HASH_NAME, org.getId(), org);
    }

    @Override
    public void updateOrganization(Organization org) {
        hashOperations.put(HASH_NAME, org.getId(), org);
    }

    @Override
    public void deleteOrganization(String organizationId) {
        hashOperations.delete(HASH_NAME, organizationId);
    }

    @Override
    public Organization findOrganization(String organizationId) {
        return (Organization) hashOperations.get(HASH_NAME, organizationId);
    }
}
```

②组织数据存储在您的 Redis 服务器中的散列名称

③HashOperations 类是一组用于在 Redis 服务器上执行数据操作的 Spring 辅助方法

④与 Redis 的所有交互都将使用按键存储的单个 Organization 对象。

OrganizationRedisRepositoryImpl 包含用于存储和检索来自 Redis 的数据的所有 CRUD（创建，读取，更新，删除）逻辑。清单 8.10 中的代码有两点需要注意：

- Redis 中的所有数据都通过键进行存储和检索。由于你存储的是从组织服务中检索

的数据，因此组织 ID 是用于存储组织记录的键的自然选择。

- 第二件要注意的事情是 Redis 服务器可以在其中包含多个哈希和数据结构。在针对 Redis 服务器的每个操作中，你需要告诉 Redis 你正在执行操作的数据结构的名称。在清单 8.10 中，您使用的数据结构名称存储在 HASH_NAME 常量中，称为“organization”。

使用 Redis 和许可服务存储和读取组织数据

现在你已经有了对 Redis 执行操作的代码，你可以修改许可服务，以便每次许可服务需要组织数据时，都会在调用组织服务之前检查 Redis 缓存。检查 Redis 的逻辑将在 licensing-service/src/main/java/com/thoughtmechanix/licenses/clients/OrganizationRestTemplateClient.java 类进行。这个类的代码如下所示。

清单 8.11 OrganizationRestTemplateClient 类将实现缓存逻辑

```
package com.thoughtmechanix.licenses.clients;

@Component
public class OrganizationRestTemplateClient {
    @Autowired
    RestTemplate restTemplate;

    @Autowired
    OrganizationRedisRepository orgRedisRepo;

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

    private Organization checkRedisCache(String organizationId) {
      try {
          return orgRedisRepo.findOrganization(organizationId);
      }
      catch (Exception ex){
        logger.error("Error encountered while trying to
          ➥ retrieve organization {} check Redis Cache.
          ➥ Exception {}", organizationId, ex);
        return null;
      }
```

①OrganizationRedisRepository 类在 OrganizationRestTemplateClient 中是自动装配的。

②尝试从 Redis 检索具有组织 ID 的 Organization 类

```
}

private void cacheOrganizationObject(Organization org) {
   try {
        orgRedisRepo.saveOrganization(org);
   }catch (Exception ex){
        logger.error("Unable to cache organization {} in Redis.
          ➥Exception {}" org.getId(), ex);
   }
}

public Organization getOrganization(String organizationId){
   logger.debug("In Licensing Service
     ➥.getOrganization: {}",
     ➥UserContext.getCorrelationId());

   Organization org = checkRedisCache(organizationId);

   if (org!=null){ ◄──────────────── ③如果你无法从 Redis 检索数据，则会调用组织服务以从源数据库检索数据。
      logger.debug("I have successfully
        ➥  retrieved an organization {}
        ➥from the redis cache: {}", organizationId, org);
      return org;
    }

    logger.debug("Unable to locate
      ➥  organization from the
      ➥  redis cache: {}.", organizationId);

    ResponseEntity<Organization> restExchange =
        restTemplate.exchange(
           "http://zuulservice/api/organization/v1/organizations/{organizationId}",
           HttpMethod.GET,
           null,
           Organization.class,
           organizationId);

    org = restExchange.getBody();

    if (org!=null) {
       cacheOrganizationObject(org); ◄──────────── ④将检索到的对象保存到缓存中
    }

    return org;
```

```
    }
}
```

getOrganization()方法是对组织服务的调用发生的地方。在进行实际 REST 调用之前，你尝试使用 checkRedisCache()方法检索与来自 Redis 的调用相关联的 Organization 对象。如果有问题的组织对象不在 Redis 中，则代码将返回空值。如果从 checkRedisCache()方法返回空值，则代码将调用组织服务的 REST 端点来检索所需的组织记录。如果组织服务返回一个组织，则使用 cacheOrganizationObject()方法缓存返回的组织对象。

> ***注意***：在与缓存进行交互时，要密切注意异常处理。为了增强弹性，如果我们无法与 Redis 服务器进行通信，我们决不会让整个调用失败。相反，我们记录异常并让调用转到组织服务。在这个特定的用例中，缓存意味着帮助提高性能，而缓存服务器不存在不应该影响调用的成功。

使用 Redis 缓存代码后，你应该点击许可服务（是的，你只有两个服务，但是你有很多的基础设施）并且看清单 8.10 中的日志消息。如果要在以下许可服务端点上执行两次背靠背的 GET 请求， http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8cc338-4ebe-a82a-e2fc1d1ff78a，则应该在日志中看到以下两条输出语句：

```
licensingservice_1 | 2016-10-26 09:10:18.455 DEBUG 28 --- [nio-8080-exec-
    1] c.t.l.c.OrganizationRestTemplateClient : Unable to locate
    organization from the redis cache: e254f8c-c442-4ebe-a82a-e2fc1d1ff78a.
licensingservice_1 | 2016-10-26 09:10:31.602 DEBUG 28 --- [nio-8080-exec-
    2] c.t.l.c.OrganizationRestTemplateClient : I have successfully
    retrieved an organization e254f8c-c442-4ebe-a82a-e2fc1d1ff78a from the
    redis cache: com.thoughtmechanix.licenses.model.Organization@6d20d301
```

控制台的第一行显示你第一次尝试访问组织的许可服务端点 e254f8c-c442-4ebe-a82a-e2fc1d1ff78a。许可服务首先检查了 Redis 缓存，并找不到要查找的组织记录。代码然后调用组织服务来检索数据。从控制台打印的第二行显示，当你再次点击许可服务端点时，组织记录现在被缓存。

8.4.2. 定义自定义 channels

以前，你在许可服务和组织服务之间建立了消息集成，以使用与 Spring Cloud Stream 项目中 Source 和 Sink 接口封装的默认输出和输入通道。但是，如果要为应用程序定义多个通道，或者要定制通道名称，则可以定义自己的接口，并根据应用程序的需要显示尽可能多的输入和输出通道。

要创建自定义通道，请在许可服务中调用 inboundOrgChanges。你可以在 licensing-service/src/main/java/com/thoughtmechanix/licenses/events/ CustomChannels.java 接口中定义通道，如下面的清单所示。

清单 8.12 为许可服务定义自定义输入通道

```
package com.thoughtmechanix.licenses.events;

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

public interface CustomChannels {
    @Input("inboundOrgChanges")
    SubscribableChannel orgs();
}
```

①@Input 注解是定义通道名称的方法级注解。

②通过@Input 注解暴露的每个通道必须返回一个 SubscribableChannel 类。

清单 8.12 的关键之处在于，对于你要暴露的每个自定义输入通道，你标记一个带有@Input 注解并返回 SubscribableChannel 类的方法。如果你想定义发布消息的输出通道，你可以在被调用的方法上面使用@OutputChannel。在输出通道的情况下，定义的方法将返回一个 MessageChannel 类，而不是与输入通道一起使用的 SubscribableChannel 类：

```
@OutputChannel(“outboundOrg”)
MessageChannel outboundOrg();
```

现在你已经定义了一个自定义输入通道，你需要在许可服务中修改两个地方来使用它。首先，你需要修改许可服务以将你的自定义输入通道名称映射到你的 Kafka 主题。下面的清单显示了这一点。

清单 8.13 修改许可服务以使用你的自定义输入通道

```
spring:
```

```
...
  cloud:
    ...
      stream:
        bindings:
          inboundOrgChanges:
            destination: orgChangeTopic
            content-type: application/json
            group: licensingGroup
```

①将通道的名称从 input 更改为 inboundOrgChanges。

要使用你的自定义输入通道，你需要将你定义的 CustomChannels 接口注入到要使用它来处理消息的类中。对于分布式缓存示例，我已将用于处理传入消息的代码移到以下许可服务类中：licensing-service/src/main/java/com/thoughtmechanix/licenses/events/handlers/OrganizationChangeHandler.java 。以下清单显示了你将与你定义的 inboundOrgChanges 通道一起使用的消息处理代码。

清单 8.14 修改许可服务以使用你的自定义输入通道

```
@EnableBinding(CustomChannels.class)
public class OrganizationChangeHandler {

    @StreamListener("inboundOrgChanges")
    public void loggerSink(
      OrganizationChangeModel orgChange) {
      .... //我们将很快进入代码的其余部分
    }
}
```

①将@EnableBindings 移出 Application.java 类并移入 OrganizationChangeHandler 类。这一次，而不是使用 Sink.class，使用你的 CustomChannels 类作为参数传递。

②使用@StreamListener 注解，你传递了通道的名称“inboundOrgChanges”，而不是使用 Sink.INPUT。

8.4.3. 将它们放在一起：收到消息时清除缓存

此刻，你不需要对组织服务做任何事情。该服务全部设置为当组织被添加，更新或删除时发布消息。你所要做的就是从清单 8.14 中构建出 OrganizationChangeHandler 类。下面的清单显示了这个类的完整实现。

清单 8.15 处理许可服务中的组织更改

```
@EnableBinding(CustomChannels.class)
public class OrganizationChangeHandler {
    @Autowired
    private OrganizationRedisRepository organizationRedisRepository;
    private static final Logger logger = LoggerFactory.getLogger(OrganizationChangeHandler.class);
```

①你用来与 Redis 交互的 OrganizationRedisRepository 类被注入到 OrganizationChangeHandler 中。

```
@StreamListener("inboundOrgChanges")
public void loggerSink(OrganizationChangeModel    orgChange) {
    switch(orgChange.getAction()){
        //Removed for conciseness
    case "UPDATE":
      logger.debug("Received a UPDATE event
        ➥from the organization service for
        ➥ organization id {}",
        ➥ orgChange.getOrganizationId());
    organizationRedisRepository
      .deleteOrganization(orgChange.getOrganizationId());
    break;
    case "DELETE":
      logger.debug("Received a DELETE event
        ➥ from the organization service for organization id {}",
        ➥orgChange.getOrganizationId());
      organizationRedisRepository
        .deleteOrganization(orgChange.getOrganizationId());
      break;
    default:
      logger.error("Received an UNKNOWN event
        ➥ from the organization service of type {}",
        ➥ orgChange.getType());
      break;
  }
}
```

②当你收到消息时，请检查数据所采取的行动，然后做出相应的反应。

③如果更新或删除组织数据，则通过 OrganizationRedisRepository 类从 Redis 中清除组织数据。

8.5. 小结

- 使用消息传递的异步通信是微服务架构的关键部分。
- 在应用程序中使用消息传递可以使你的服务扩展并变得更容错。
- Spring Cloud Stream 通过使用简单的注解和抽象出底层消息平台的平台特定细节来简化消息的生产和消费。
- Spring Cloud Stream 消息源是一个带注解的 Java 方法，用于将消息发布到消息代理的队列中。
- Spring Cloud Stream 消息接收器是一个带注解的 Java 方法，它从消息代理的队列中

接收消息。

- Redis 是一个可用作数据库和缓存的键值存储。