9. 第 9 章 使用 Spring Cloud Sleuth 和 Zipkin 进行分布式跟踪

本章内容

- 使用 Spring Cloud Sleuth 将跟踪信息注入到服务调用
- 使用日志聚合查看分布式交易的日志
- 通过日志聚合工具进行查询
- 使用 OpenZipkin 可视化地了解用户的交易，因为它流经多个微服务调用
- 使用 Spring Cloud Sleuth 和 Zipkin 自定义跟踪信息

微服务架构是一个强大的设计范例，可将复杂的单体软件系统分解为更小，更易于管理的部分。这些可管理的部分可以彼此独立地构建和部署；然而，这种灵活性的代价是复杂。因为微服务本质上是分布式的，所以试图调试问题发生的地方会令人发狂。服务的分布式特性意味着你必须跨多个服务，物理机器和不同的数据存储来跟踪一个或多个交易，并试着把正在发生的事情拼凑起来。

本章介绍了使分布式调试成为可能的几种方法和技术。在本章中，我们看看以下几点：

- 使用关联 ID 将多个服务之间的交易链接在一起
- 将来自多个服务的日志数据汇总到单个可搜索的源中
- 可视化跨多个服务的用户交易流程，并了解交易每个部分的性能特点

要完成这三件事你将使用三种不同的技术：

- Spring Cloud Sleuth (https://cloud.spring.io/spring-cloud-sleuth/)：Spring Cloud Sleuth 是一个 Spring Cloud 项目，它使用关联 ID 处理 HTTP 调用，并提供将生成的跟踪数据提供给 OpenZipkin 的钩子。它通过添加过滤器并与其他 Spring 组件进行交互来完成此操作，以便将生成的关联 ID 传递到所有系统调用。

- Papertrail (https://papertrailapp.com)：Papertrail 是一款基于云的服务（基于免费增值服务）,允许你将来自多个来源的日志数据汇总到单个可搜索的数据库中。你可以选择进行日志聚合，包括内部部署、基于云、开源和商业解决方案。本章后面我们将探讨这些替代方案中的几个。
- Zipkin (http://zipkin.io)：Zipkin 是一个开源的数据可视化工具，可以显示跨多个服务的交易流程。Zipkin 允许你将交易分解成组件，并以可视方式识别可能存在性能热点的地方。

为了开始本章，我们从最简单的跟踪工具开始，即关联 ID。

注意：本章的部分内容依赖于第 6 章中介绍的材料（特别是 Zuul 预过滤器、响应过滤器和后置过滤器）。如果你还没有阅读第 6 章，我建议你在阅读本章之前这样做。

9.1. Spring Cloud Sleuth 和关联 ID

我们在第 5 章和第 6 章首先介绍了关联 ID 的概念。关联 ID 是一个随机生成的唯一编号或字符串，在交易启动时分配给一个交易。由于交易流经多个服务，关联 ID 从一个服务调用传递到另一个。在第 6 章中，你使用 Zuul 过滤器来检查所有传入的 HTTP 请求，并在其不存在的情况下插入一个关联 ID。

一旦关联 ID 出现 ,你就可以在每个服务上使用自定义的 Spring HTTP 过滤器来将传入变量映射到自定义的 UserContext 对象。通过使用 UserContext 对象，你现在可以手动将关联 ID 添加到任何日志语句中，方法是确保将关联 ID 附加到日志语句中，或者只需要一点点工作即可将关联 ID 直接添加到 Spring 的 Mapped Diagnostic 上下文（MDC）。你还编写了一个 Spring 拦截器，它可以确保来自服务的所有 HTTP 调用都会通过将关联 ID 添加到任何出站调用的 HTTP 标头中来传递关联 ID。

哦，你必须执行 Spring 和 Hystrix 魔法，以确保拥有关联 ID 的父线程的线程上下文被正确地传递给 Hystrix。哇，最后，这是很多的基础设施，为了某些你希望只有在问题发生时才被查看的东西（使用关联 ID 来跟踪交易中发生了什么）而设置的。

幸运的是，Spring Cloud Sleuth 为你管理所有这些代码基础架构和复杂性。通过将 Spring Cloud Sleuth 添加到你的 Spring 微服务中，你可以：

- 如果不存在，透明地创建并向服务调用中注入关联 ID。
- 管理关联 ID 传递到出站服务调用，以便交易的关联 ID 自动添加到出站调用。
- 将关联信息添加到 Spring 的 MDC 日志记录中，以便生成的关联 ID 由 Spring Boots 默认的 SL4J 和 Logback 实现自动记录。
- 可选，将服务调用中的跟踪信息发布到 Zipkin 分布式跟踪平台。

注意：如果你使用 Spring Boot 的日志记录实现，Spring Cloud Sleuth 将自动获得添加到你在微服务中放入的日志语句中的关联 ID。

让我们继续，将 Spring Cloud Sleuth 添加到你的许可服务和组织服务中。

9.1.1. 添加 Spring Cloud Sleuth

要开始在你的两个服务（许可和组织）中使用 Spring Cloud Sleuth，你需要在这两个服务中的 pom.xml 文件中添加一个 Maven 依赖：

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

这个依赖将拉取 Spring Cloud Sleuth 所需的所有核心库。是的！一旦这个依赖被拉取，你的服务现在就会：

- 检查每个传入的 HTTP 服务，并确定传入调用中是否存在 Spring Cloud Sleuth

跟踪信息。如果 Spring Cloud Sleuth 跟踪数据确实存在，则传递到你的微服务的跟踪信息将被捕获并提供给你的服务进行日志记录和处理。

- 将 Spring Cloud Sleuth 跟踪信息添加到 Spring MDC 中，以便将由微服务创建的每个日志语句添加到日志中。
- 将 Spring Cloud 跟踪信息注入到你的服务所发出的每个出站 HTTP 调用和 Spring 消息传递通道消息中。

9.1.2. 分析 Spring Cloud Sleuth Trace

如果所有设置都正确，那么写在服务应用程序代码中的任何日志语句都将包含 Spring Cloud Sleuth 跟踪信息。例如，图 9.1 显示了如果你在组织服务上执行 HTTP GET http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a，那么服务的输出将会是什么样子。

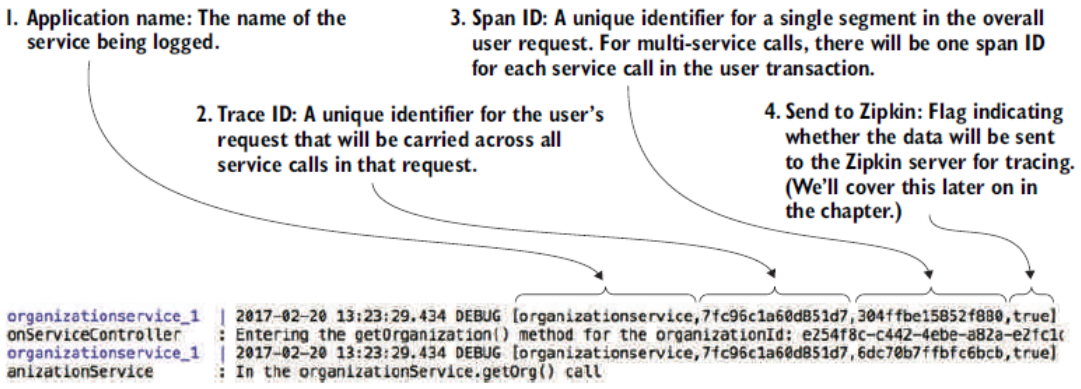

图 9.1 Spring Cloud Sleuth 将四个跟踪信息添加到由你的服务编写的每个日志条目中。这些数据有助于将用户请求的服务调用联系在一起。

① *Application name: The name of the service being logged.*

应用程序名称：正在记录的服务的名称。

② *Trace ID: A unique identifier for the user's request that will be carried across all*

service calls in that request.

Trace ID: 用户请求的唯一标识符，将在该请求中的所有服务调用中携带。

③ Span ID: A unique identifier for a single segment in the overall user request. For multi-service calls, there will be one span ID for each service call in the user transaction.

Span ID: 整个用户请求中的单个段的唯一标识符。对于多业务调用，用户交易中每个服务调用将有一个 span ID。

④ Send to Zipkin: Flag indicating whether the data will be sent to the Zipkin server for tracing. (We'll cover this later on in the chapter.)

发送到 Zipkin：标志表示是否将数据发送到 Zipkin 服务器进行跟踪。（我们将在后面的章节中介绍。）

Spring Cloud Sleuth 将为每个日志条目添加四条信息。这四个部分（编号与图 9.1 中的数字相对应）是：

- 服务的应用程序名称：这将成为日志条目的应用程序的名称。默认情况下，Spring Cloud Sleuth 使用应用程序的名称（spring.application.name）作为写入跟踪的名称。
- Trace ID：Trace ID 是关联 ID 的等效术语。这是一个代表整个交易的唯一编号。
- Span ID：Span ID 是表示整个交易的一部分的唯一 ID。参与交易的每个服务都有自己的 Span ID。当你与 Zipkin 集成以可视化你的交易时，Span ID 尤其相关。
- 跟踪数据是否被发送到 Zipkin：在大量的服务中，生成的跟踪数据量可能会很大，而且不会增加重大的价值。Spring Cloud Sleuth 让你确定何时以及如何向 Zipkin 发送交易。Spring Cloud Sleuth 跟踪块末尾的 true/false 标识符告诉你跟踪信息

是否发送给 Zipkin。

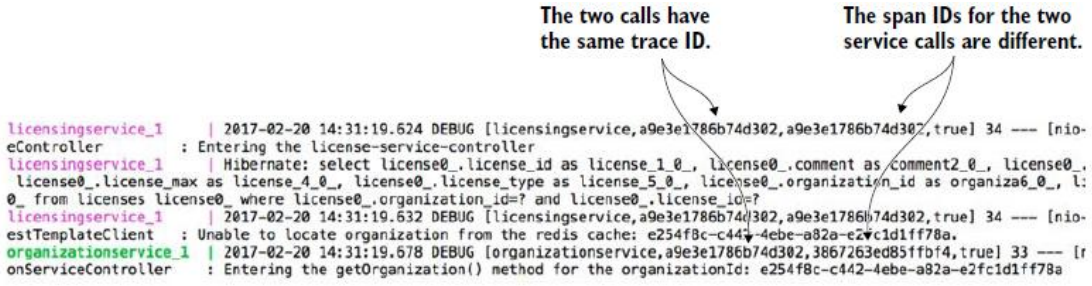

图 9.2 在交易中涉及多个服务的情况下，你可以看到它们共享相同的 Trace ID。

① *The two calls have the same trace ID.*

这两个调用具有相同的 Trace ID。

② *The span IDs for the two service calls are different.*

两个服务调用的 Span ID 是不同的。

到现在为止，我们只查看了单个服务调用产生的日志数据。让我们来看看在 GET http://localhost:5555/api/licensing/v1/organizations/e254f8cc442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a 中调用许可服务时会发生什么。请记住，许可服务还必须向组织服务发出调用。图 9.2 显示了两个服务调用的日志输出。

通过查看图 9.2，可以看到许可服务和组织服务都具有相同的 Trace ID ，值为 a9e3e1786b74d302。但是，许可服务的 Span ID 为 a9e3e1786b74d302（与 Transaction ID 相同的值）。组织服务的 Span ID 为 3867263ed85ffbf4。

通过增加一些 POM 依赖关系，你已经替换了第 5 章和第 6 章中构建的所有关联 ID 基础结构。就个人而言，在这个世界上，没有任何东西让我更开心，然后用别人的代码取代复杂的基础设施风格的代码。

9.2. 日志聚合和 Spring Cloud Sleuth

在大规模的微服务环境中（特别是在云端），日志数据是调试问题的关键工具。但是，由于基于微服务的应用程序的功能被分解为细小的细粒度服务，并且对于单个服务类型可以有多个服务实例，因此试图绑定记录多个服务中的数据来解决用户问题可能非常困难。试图跨多个服务器调试问题的开发人员通常需要尝试以下方法：

- 登录到多个服务器来检查每个服务器上存在的日志。这是一项非常费力的任务，特别是如果有问题的服务具有不同的交易量，导致日志以不同的速率滚动。
- 编写自制的查询脚本，将尝试解析日志并确定相关的日志条目。因为每个查询都可能不同，所以最终会有大量自定义脚本用于查询日志中的数据。
- 延长停机服务进程的恢复时间，因为开发人员需要备份驻留在服务器上的日志。如果托管服务的服务器完全崩溃，则日志通常会丢失。

列出的每个问题都是我遇到的实际问题。在分布式服务器上调试问题是件难事，而且往往会显著增加识别和解决问题所需的时间。

一个更好的方法是将所有服务实例中的所有日志实时流式传输到一个集中式集合点，在该集合点处可以将日志数据编入索引并进行搜索。图 9.3 在概念层面显示了这种“统一”的日志架构是如何工作的。

① Each individual service is producing logging data.

每个服务都在生成日志数据。

② An aggregation mechanism collects all of the data and funnels it to a common data store.

一个聚合机制收集所有的数据并将其引导到一个通用的数据存储。

③ As data comes into a central data store, it is indexed and stored in a searchable

format.

随着数据进入中央数据存储区，它将被索引并以可搜索的格式存储。

④ The development and operations teams can query the log data to find individual transactions. The trace IDs from Spring Cloud Sleuth log entries allow us to tie log entries across services.

开发和运营团队可以查询日志数据以查找单个交易。来自 Spring Cloud Sleuth 日志条目的 Trace ID 使我们能够在各个服务中绑定日志条目。

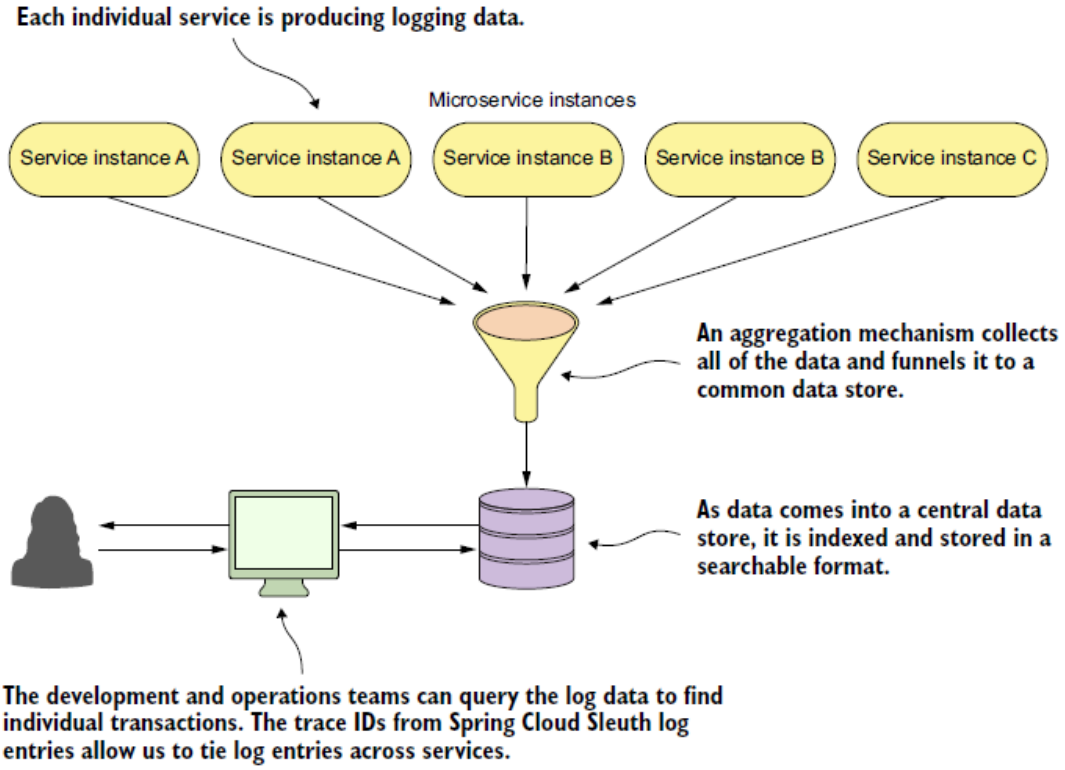

图 9.3 日志聚合和跨服务日志条目的唯一 Transaction ID 的组合使调试分布式事务更易于管理。

幸运的是，有多种开源和商业产品可以帮助你实现前面所述的日志架构。此外，还有多种实施模式可供你在本地管理的本地解决方案或基于云的解决方案之间进行选择。表 9.1 总结了可用于日志记录基础设施的几种选择。

表 9.1 用于 Spring Boot 的日志聚合解决方案的选项

产品名称	实现模式	备注
Elasticsearch, Logstash, Kibana (ELK)	开源 商业 本地典型实现	http://elastic.co 通用搜索引擎 可以通过（ELK-stack）进行日志聚合， 需要最多的手工支持
Graylog	开源 商业 本地	http://graylog.org 开放源代码平台的设计是为了安装在本 地
Splunk	仅商业 本地或基于云	http://splunk.com 最早，最全面的日志管理和聚合的最初 工具是内部部署的解决方案，但之后又 提供了云服务。
Sumo Logic	免费增值模式 商业 基于云	http://sumologic.com 免费/分层定价模式 仅作为云服务运行 需要一个公司工作账户以注册（没有 Gmail 或 Yahoo 账户）
Papertrail	免费增值模式 商业 基于云	http://papertrailapp.com 免费/分层定价模式 仅作为云服务运行

有了这些选择，可能很难选择哪一个最好。 每个组织都会有所不同，有不同的需求。

在本章中，我们将以 Papertrail 为例，介绍如何将 Spring Cloud Sleuth 支持的日志

集成到统一的日志记录平台中。我选择了 Papertrail，因为：

- 它有一个免费增值模式，可以让你注册一个免费的账户。
- 这非常容易设置，特别是像 Docker 这样的容器运行时。
- 这是基于云的。虽然我认为一个良好的日志记录基础设施对微服务应用程序是至关重要的，但我不相信大多数组织都有时间或技术人才来正确设置和管理日志记录平台。

9.2.1. 实战中的 Spring Cloud Sleuth/Papertrail 实现

在图 9.3 中我们看到了一个通用的统一日志架构。现在让我们来看看 Spring Cloud Sleuth 和 Papertrail 如何实现相同的架构。

要设置 Papertrail 与你的环境一起工作，我们必须采取以下措施：

- 创建一个 Papertrail 账户并配置一个 Papertrail 系统日志连接器。
- 定义一个 Logspout Docker 容器(https://github.com/gliderlabs/logspout)，以从所有 Docker 容器中捕获标准。
- 通过基于 Spring Cloud Sleuth 中的关联 ID 发出查询来测试实现。

图 9.4 显示了你的实现的最终状态以及 Spring Cloud Sleuth 和 Papertrail 如何切合你的解决方案。

① The individual containers write their logging data to standard out. Nothing has changed in terms of their configuration.

单个容器将其日志数据写入标准输出。它们的配置没有任何改变。

② In Docker, all containers write their standard out to an internal filesystem called Docker.sock.

在 Docker 中，所有容器都将其标准写入到名为 Docker.sock 的内部文件系统中。

③ A Logspout Docker container listens to Docker.sock and writes whatever goes to standard output to a remote syslog location.

Logspout Docker 容器监听 Docker.sock，并将任何标准输出写入远程 syslog 位置。

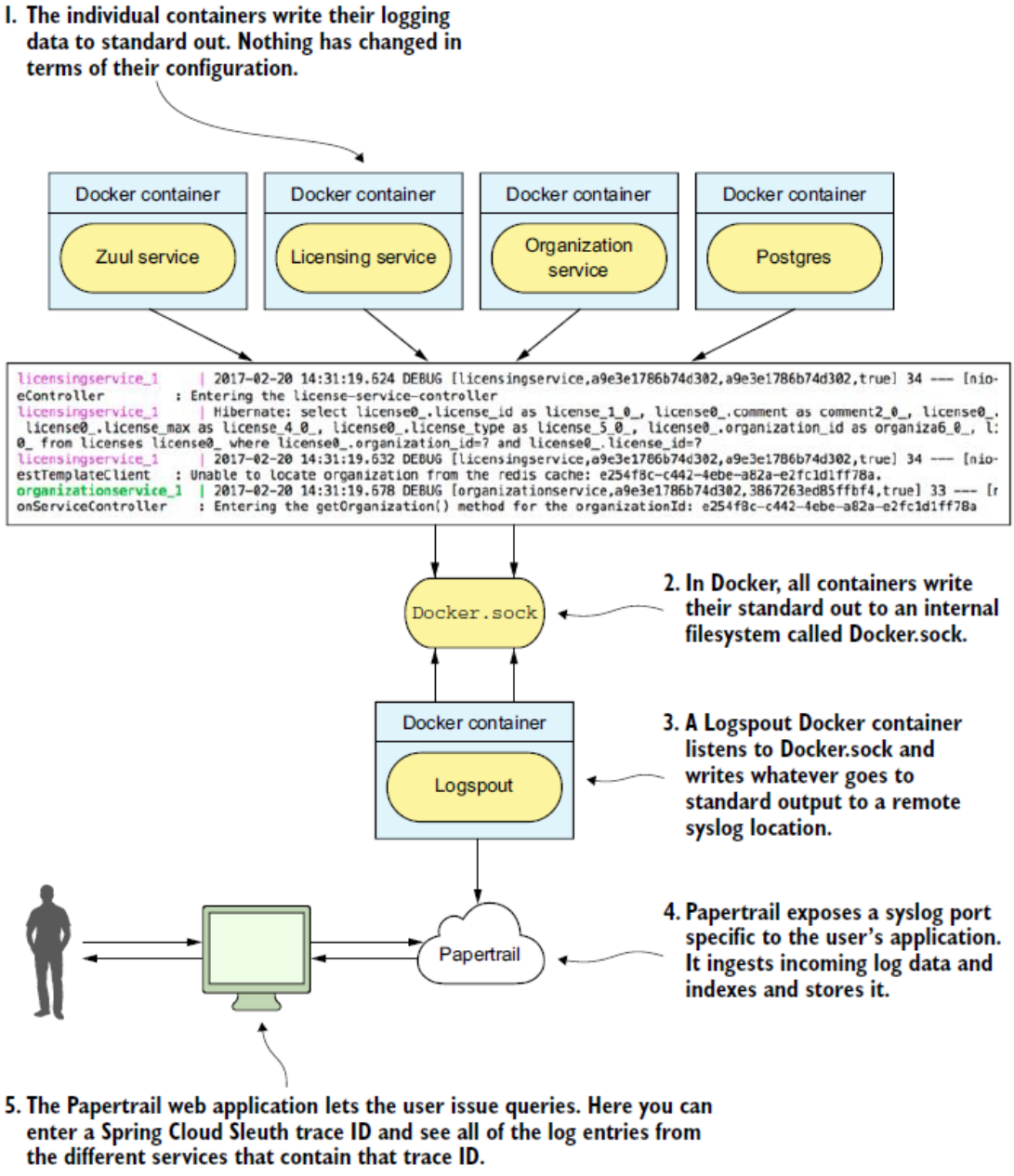

图 9.4 使用原生的 Docker 功能 ,logspot 和 Papertrail 可以让你快速实现统一的日志记录架构。

④ Papertrail exposes a syslog port specific to the user's application. It ingests

incoming log data and indexes and stores it.

Papertrail 暴露了特定于用户应用程序的系统日志端口。 它接收传入的日志数据和索引并存储它。

⑤ The Papertrail web application lets the user issue queries. Here you can enter a Spring Cloud Sleuth trace ID and see all of the log entries from the different services that contain that trace ID.

Papertrail Web 应用程序让用户发出查询。在这里 ,你可以输入 Spring Cloud Sleuth Trace ID，并查看包含该 Trace ID 的不同服务的所有日志条目。

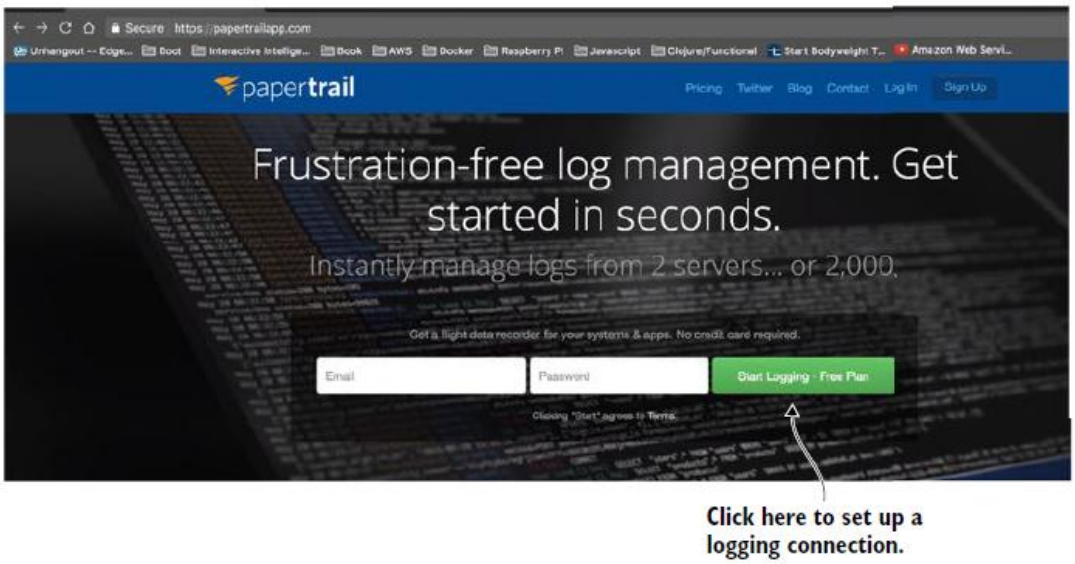

图 9.5 首先，在 Papertrail 上创建一个账户。

① Click here to set up a logging connection.

点击这里设置日志连接。

9.2.2. 创建一个 Papertrail 账户并配置 syslog 连接器

你将开始创建一个 Papertrail。要开始，请访问 https://papertrailapp.com 并点击绿色的“Start Logging - Free Plan”按钮。图 9.5 显示了这一点。

Papertrail 不需要大量的信息就可以开始使用；只需有效的电子邮件地址。填写账户信息后，你将看到一个屏幕，用于设置你的第一个系统以记录数据。图 9.6 显示了这个屏幕。

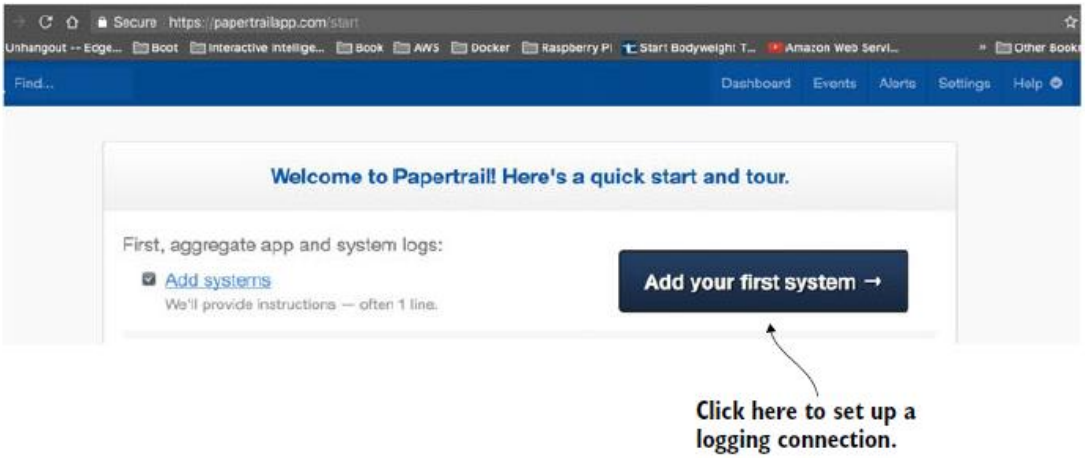

图 9.6 下一步选择如何将日志数据发送到 Papertrail。

① *Click here to set up a logging connection.*

点击这里设置日志连接。

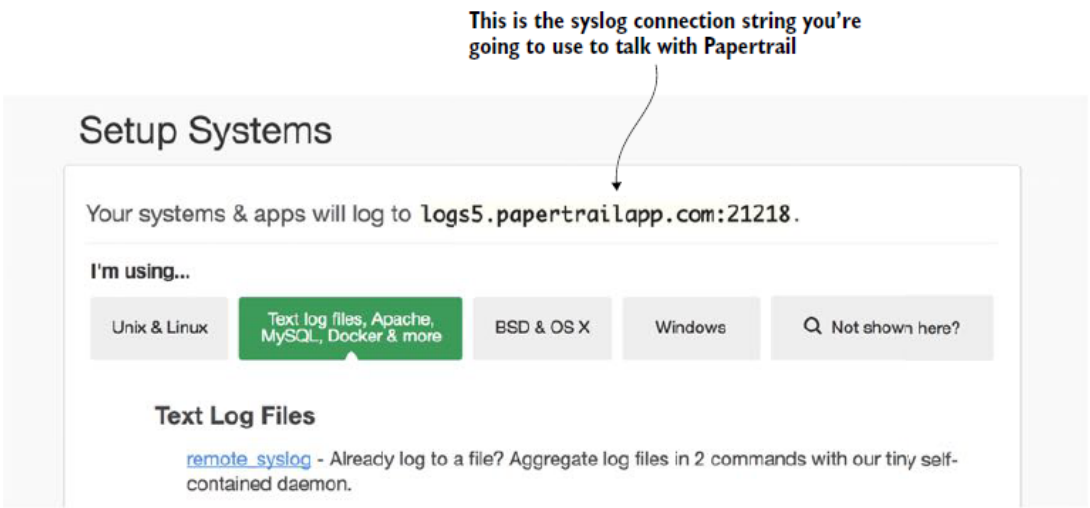

图 9.7 Papertrail 使用 Syslog 作为发送数据的机制之一。

① *This is the syslog connection string you're going to use to talk with Papertrail*

这是你将用于与 Papertrail 交互的系统日志连接字符串

默认情况下，Papertrail 允许你通过 Syslog 调用(https://en.wikipedia.org/wiki/Syslog)发送日志数据给它。Syslog 是源自 UNIX 的日志消息传递格式。Syslog 允许通过

TCP 和 UDP 发送日志消息。Papertrail 将自动定义一个 Syslog 端口，你可以使用它来写入日志消息。为了讨论的目的，你将使用此默认端口。图 9.7 显示了当你点击图 9.6 所示的“Add your first system”按钮时自动生成的 Syslog 连接字符串。

此刻，你们用 Papertrail 进行了所有的设置。你现在必须配置你的 Docker 环境，以将每个运行你的服务的容器的输出捕获到图 9.7 中定义的远程 syslog 端点。

注意：图 9.7 中的连接字符串对我的账户是唯一的。你需要确保使用 Papertrail 为你生成的连接字符串，或者通过 Papertrail Settings > Log destinations 菜单选项定义一个连接字符串。

9.2.3. 将 Docker 输出重定向到 Papertrail

通常情况下，如果你在自己的虚拟机上运行每个服务，则必须配置每个服务的日志记录配置，以将其日志记录信息发送到远程 syslog 端点（如通过 Papertrail 暴露的那个端点）。

幸运的是，Docker 使得捕获物理或虚拟机上运行的任何 Docker 容器的所有输出变得非常简单。Docker 守护进程通过一个名为 docker.sock 的 Unix 套接字与它管理的所有 Docker 容器进行通信。运行 Docker 的服务器上运行的任何容器都可以连接到 docker.sock，并接收由该服务器上运行的所有其他容器生成的所有消息。简而言之，docker.sock 就像一个管道，你的容器可以插入并捕获在运行 Docker 守护进程的虚拟服务器上的 Docker 运行时环境中进行的全部活动。

你将使用一个名为 Logspout(https://github.com/gliderlabs/logspout) 的“Dockerized”软件，它将监听 docker.sock 套接字，然后捕获在 Docker 运行时生成的任何标准输出消息，并将输出重定向到远程系统日志（Papertrail）。要设置你的 Logspout 容器，你必须将一个条目添加到你用于启动本章中代码示例所用的所有 Docker 容器的

docker-compose.yml 文件中。你需要修改 docker/common/dockercompose.yml 文件，应该在其中添加以下条目：

```
logspout:
    image: gliderlabs/logspout
    command: syslog://logs5.papertrailapp.com:21218
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

注意：在前面的代码片段中，你需要用“Paratrail”中提供给您的值替换“command”属性中的值。如果你使用以前的 Logspout 代码片段，你的 Logspout 容器将会愉快地写入你的日志条目到我的 Papertrail 账户。

现在在本章中启动 Docker 环境时，发送到容器标准输出的所有数据都将发送到 Papertrail。你可以在开始第 9 章的 Docker 示例并单击屏幕右上角的事件按钮后，通过登录到你的 Papertrail 账户来看到这一点。

图 9.8 显示了发送给 Papertrail 的数据的一个例子。

图 9.8 在定义了 Logspout Docker 容器的情况下，写入每个容器标准输出的数据将被发送到 Papertrail。

① Individual service log events are written to the container’s stdout. The stdout from the container is captured by Logspout and then sent to Papertrail.

单独的服务日志事件被写入容器的标准输出。容器的标准输出被 Logspout 捕获，然后发送

到 Papertrail。

② *Click here to see the logging events being sent to Papertrail.*

点击这里查看发送给 Papertrail 的记录事件。

为什么不使用 Docker 日志记录驱动程序？

Docker 1.6 及更高版本允许你定义替代日志记录驱动程序来编写从每个容器写入的 stdout/stderr 消息。其中一个日志记录驱动程序是一个 syslog 驱动程序，可用于将消息写入远程 syslog 监听程序。

为什么选择 Logspout 而不是使用标准的 Docker 日志驱动程序？主要原因是灵活性。Logspout 提供了用于自定义将日志数据发送到日志聚合平台的功能。Logspout 提供的功能包括：

- 一次将日志数据发送到多个端点的功能。许多公司都希望将自己的日志数据发送到日志聚合平台，并希望安全监控工具能够监控生成的日志中的敏感数据。
- 用于过滤哪些容器将发送其日志数据的集中位置。使用 Docker 驱动程序，你需要在 docker-compose.yml 文件中手动设置每个容器的日志驱动程序。使用 Logspout，你可以在特定的容器中定义过滤器，甚至在集中配置中定义特定的字符串模式。
- 允许应用程序通过特定的 HTTP 端点写入日志信息的自定义 HTTP 路由。此功能允许你执行诸如将特定日志消息写入特定的下游日志聚合平台的操作。例如，你可能会将标准日志消息从 stdout/stderr 转到 Papertrail，你可能希望将特定的应用程序审计信息发送到内部的 Elasticsearch 服务器。
- 与 syslog 以外的协议集成。Logspout 允许你通过 UDP 和 TCP 协议发送消息。Logspout 也有第三方模块，可以将 Docker 的 stdout/stderr 整合到 Elasticsearch

中。

9.2.4. 在 Papertrail 里搜索 Spring Cloud Sleuth Trace ID

现在你的日志正在流向 Papertrail ,你可以真正开始欣赏 Spring Cloud Sleuth 将 Trace ID 添加到所有日志条目。要查询与单个交易相关的所有日志条目，只需在 Papertrail 事件屏幕的查询框中输入一个 Trace ID 并进行查询。图 9.9 显示了如何通过我们前面在 9.1.2 节使用的 Spring Cloud sleuth Trace ID：a9e3e1786b74d302 来执行一个查询。

合并日志和赞美平凡

不要低估整合日志架构和服务关联策略的重要性。这似乎是一个平凡的任务，但在编写本章时，我使用了类似于 Papertrail 的日志聚合工具来为我正在处理的项目追踪三种不同服务之间的竞争情况。事实证明，竞争的状况已经存在了一年多了，但是处于竞争状况的服务一直很好，直到我们在混合状况下增加了一些负荷和另外一个角色导致了问题发生。

我们在花了 1.5 周完成日志查询并通过分析几十个独特场景的跟踪输出之后才发现了这个问题。如果没有已安装的聚合日志平台，我们就不会发现问题所在。

这个经历重申了几件事情：

- *确保在服务开发的早期定义和实现日志记录策略*：一旦项目正在进行，实施日志记录基础设施是乏味的，有时是困难的，并且是耗时的。
- *日志记录是微服务基础设施的关键部分*：在实施自己的日志记录解决方案之前，花点时间思考，甚至尝试实施本地日志记录解决方案。花在基于云的日志记录平台的钱是值得的。
- *学习你的日志工具*：几乎每个日志记录平台都有一个用于查询合并日志的查询语言。日志是一个令人难以置信的信息来源和指标。它们本质上是另一种类型的数据

库，你花时间学习查询将带来极大的利益。

① *The logs show that the licensing service and then the organization service were called as part of this single transaction.*

日志显示，许可服务，然后是组织服务作为这个单一交易的一部分被调用。

② *Here's the Spring Cloud Sleuth trace ID you're going to query for.*

这是你要查询的 Spring Cloud Sleuth Trace ID。

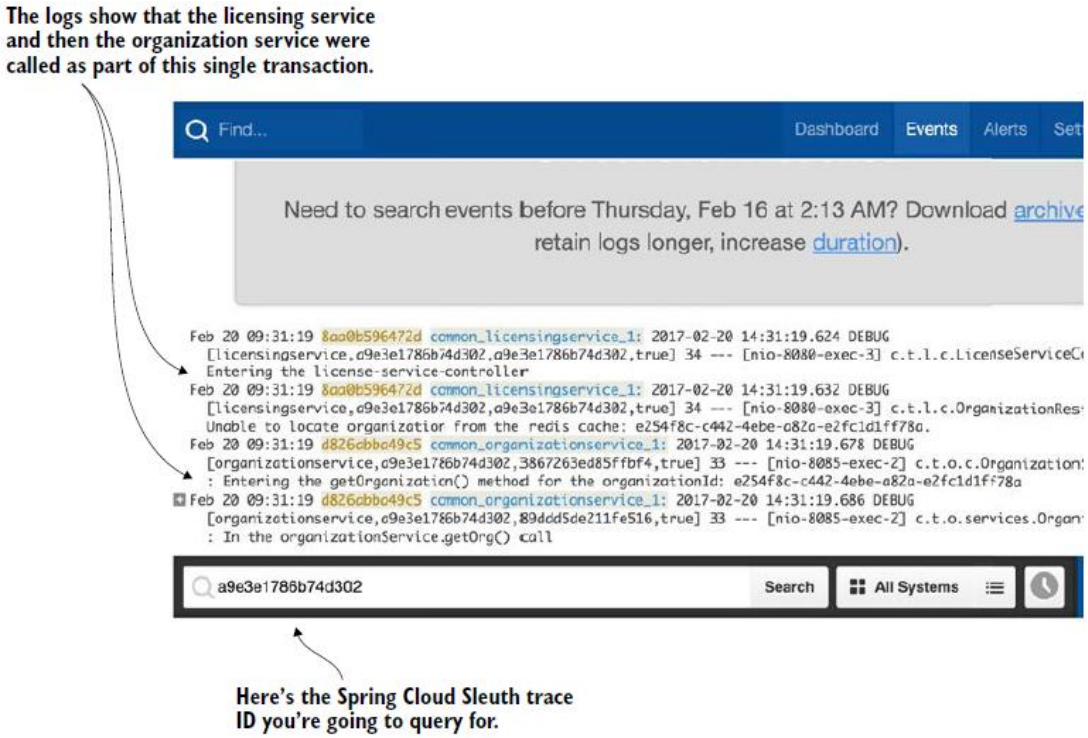

图 9.9 Trace ID 允许你过滤与单个交易相关的所有日志条目。

9.2.5. 使用 Zuul 将关联 ID 添加到 HTTP 响应中

如果你检查了使用 Spring Cloud Sleuth 进行的任何服务调用返回的 HTTP 响应，你将看到调用中使用的 Trace ID 从不会在 HTTP 响应头中返回。如果你检查 Spring Cloud Sleuth 的文档，你会发现 Spring Cloud Sleuth 团队认为，返回任何跟踪数据都可能是潜在的安全问题（尽管他们没有明确列出他们为什么相信这一点）。

但是，我发现在调试问题时，在 HTTP 响应中返回关联 ID 或 Trace ID 是非常有用的。Spring Cloud Sleuth 允许你用 Trace ID 和 Span ID“装饰”HTTP 响应信息。但是，这样做的过程涉及编写三个类并注入两个自定义 Spring bean。如果你想采取这种方法，你可以在 Spring Cloud Sleuth 文档(http://cloud.spring.io/spring-cloud-static/spring-cloudsleuth/1.0.12.RELEASE/)中看到它。一个更简单的解决方案是编写一个 Zuul“POST”过滤器，将 Trace ID 注入到 HTTP 响应中。

在第 6 章中，当我们介绍 Zuul API 网关时，我们看到了如何构建一个 Zuul“POST”响应过滤器，以将你在服务中使用的关联 ID 添加到返回给调用者的 HTTP 响应中。你现在要修改该过滤器来添加 Spring Cloud Sleuth 头。

要创建你的 Zuul 响应过滤器，你需要将单个 JAR 依赖添加到你的 Zuul 服务器的 pom.xml 文件中：spring-cloud-starter-sleuth。spring-cloud-starter-sleuth 依赖将被用来告诉 Spring Cloud Sleuth 你想让 Zuul 参与 Spring Cloud 的跟踪。在本章的后面，当我们介绍 Zipkin 的时候，你会发现 Zuul 服务将成为任何服务调用的第一个调用。

对于第 9 章 这个文件可以在 zuulsvr/pom.xml 中找到。下面的清单显示了这些依赖。

清单 9.1 将 Spring Cloud Sleuth 加入到 Zuul

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

①向 Zuul 添加 spring-cloud-starter-sleuth 将为在 Zuul 中被调用的每个服务生成一个 Trace ID

一旦这个新的依赖到位，实际的 Zuul“post”过滤器实现起来是微不足道的。下面的清单显示了用于构建 Zuul 过滤器的源代码。该文件位于 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/ResponseFilter.java。

清单 9.2 通过 Zuul POST 过滤器添加 Spring Cloud Sleuth Trace ID

```
package com.thoughtmechanix.zuulsvr.filters;

import org.springframework.cloud.sleuth.Tracer;
```

```
@Component
public class ResponseFilter extends ZuulFilter{

   private static final int FILTER_ORDER=1;
   private static final boolean SHOULD_FILTER=true;
   private static final Logger logger = LoggerFactory.getLogger(ResponseFilter.class);

   @Autowired
   Tracer tracer;

   @Override
   public String filterType() {return "post";}

   @Override
   public int filterOrder() {return FILTER_ORDER;}

   @Override
   public boolean shouldFilter() {return SHOULD_FILTER;}

   @Override
   public Object run() {
      RequestContext ctx = RequestContext.getCurrentContext();
      ctx.getResponse().addHeader("tmx-correlation-id",
         ➥ tracer.getCurrentSpan().traceIdString());
      return null;
   }
}
```

①Tracer 类是访问 Trace ID 和 Span ID 信息的入口点。

②你将添加一个名为 tmxcorrelation-ID 的新 HTTP 响应头，以保存 Spring Cloud Sleuth Trace ID。

由于 Zuul 现在支持 Spring Cloud Sleuth，因此可以通过在 Tracer 类中自动装入 ResponseFilter 来访问 ResponseFilter 中的跟踪信息。Tracer 类允许你访问正在执行的当前 Spring Cloud Sleuth 跟踪信息。tracer.getCurrentSpan().traceIdString()方法允许你检索正在进行的交易的当前 Trace ID（作为字符串）。

将 Trace ID 添加到通过 Zuul 传回的传出 HTTP 响应是微不足道的。这是通过以下调用完成的：

```
RequestContext ctx = RequestContext.getCurrentContext();
      ctx.getResponse().addHeader("tmx-correlation-id",
         ➥ tracer.getCurrentSpan().traceIdString());
```

有了这个代码，如果你通过你的 Zuul 网关调用一个 EagleEye 微服务，你应该得到一

个称为 tmx-correlation-id 的 HTTP 响应。

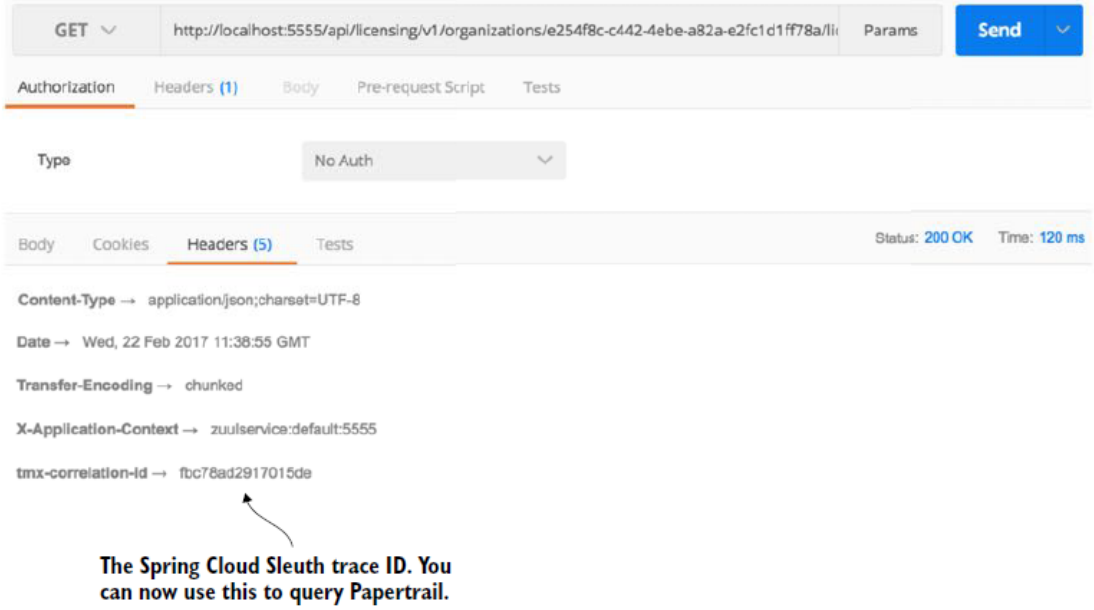

图 9.10 使用返回的 Spring Cloud Sleuth Trace ID，你可以轻松查询 Papertrail 的日志。

① *The Spring Cloud Sleuth trace ID. You can now use this to query Papertrail.*

Spring Cloud Sleuth Trace ID。你现在可以用它来查询 Papertrail。

图 9.10 显示了 GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a 调用的结果。

9.3. 使用 Zipkin 实现分布式跟踪

拥有统一的具有关联 ID 的日志记录平台是一个强大的调试工具。然而，在本章的其余部分，我们将从追踪日志条目转移出来，而是看看如何可视化在不同的微服务中移动的交易流。一张干净、简洁的，可以工作的图片超过一百万条日志条目。

分布式跟踪涉及提供一个交易如何跨不同的微服务流动的可视化图片。分布式跟踪工具也将对单个微服务的响应时间给出一个粗略的近似值。但是，分布式跟踪工具不应与完整的

应用程序性能管理（APM）软件包混淆。这些软件包可以为服务中的实际代码提供开箱即用的低层性能数据，还可以提供响应时间（例如内存，CPU 利用率和 I/O 利用率）以外的性能数据。

这就是 Spring Cloud Sleuth 和 OpenZipkin（也称为 Zipkin）项目的亮点。Zipkin(http://zipkin.io/)是一个分布式的跟踪平台，允许你跟踪跨多个服务调用的交易。Zipkin 允许你以图形方式查看交易所花费的时间，并细化调用中涉及的每个微服务所花费的时间。Zipkin 是一个在微服务架构中识别性能问题的宝贵工具。

设置 Spring Cloud Sleuth 和 Zipkin 涉及四项操作：

- 将 Spring Cloud Sleuth 和 Zipkin JAR 文件添加到捕获跟踪数据的服务中
- 在每个服务中配置一个 Spring 属性以指向将收集跟踪数据的 Zipkin 服务器
- 安装和配置 Zipkin 服务器来收集数据
- 定义每个客户端将用于向 Zipkin 发送跟踪信息的采样策略

9.3.1. 配置 Spring Cloud Sleuth 和 Zipkin 依赖

到目前为止，你已经为你的 Zuul、许可服务和组织服务添加了两套 Maven 依赖。这些 JAR 文件是 spring-cloud-starter-sleuth 和 spring-cloud-sleuth-core 的依赖。spring-clouds-tarter-sleuth 依赖被用来包含在一个服务中启用 Spring Cloud Sleuth 所需的基本的 Spring Cloud Sleuth 库。Spring-cloudsleuth-core 依赖关在你必须以编程方式与 Spring Cloud Sleuth 进行交互时使用（你将在本章稍后再做）。

要与 Zipkin 集成，您需要添加第二个名为 spring-cloud-sleuth-zipkin 的 Maven 依赖。下面的清单显示了一旦添加了 spring-cloud-sleuth-zipkin 依赖，应该在 Zuul、许可服务和组织服务中出现的 Maven 条目。

清单 9.3 客户端 Spring Cloud Sleuth 和 Zipkin 依赖

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

9.3.2. 配置服务指向 Zipkin

使用 JAR 文件，你需要配置每个想要与 Zipkin 进行通信的服务。你可以通过设置一个 Spring 属性来定义用于与 Zipkin 进行通信的 URL。需要设置的属性是 spring.zipkin.baseUrl。该属性在每个服务的 application.yml 属性文件中设置。

> ***注意***：spring.zipkin.baseUrl 也可以作为 Spring Cloud Config 中的一个属性进行外部配置。

在每个服务的 application.yml 文件中，该值设置为 http://localhost:9411。但是，在运行时，我使用在每个服务 Docker 配置文件(docker/common/docker-compose.yml)上传递的 ZIPKIN_URI(http://zipkin:9411) 变量来覆盖此值。

> **Zipkin, RabbitMQ 和 Kafka**
>
> Zipkin 确实能够通过 RabbitMQ 或 Kafka 将其跟踪数据发送到 Zipkin 服务器。从功能的角度来看，如果你使用 HTTP，RabbitMQ 或 Kafka，Zipkin 的行为没有任何区别。通过 HTTP 跟踪，Zipkin 使用异步线程发送性能数据。使用 RabbitMQ 或 Kafka 收集您的追踪数据的主要优点是，如果你的 Zipkin 服务器关闭，发送给 Zipkin 的任何追踪消息将被“入队”，直到 Zipkin 可以获取数据。
>
> Spring Cloud Sleuth 通过 RabbitMQ 和 Kafka 将数据发送到 Zipkin 的配置在 Spring Cloud Sleuth 文档中有介绍，所以我们在这里不再详细介绍。

9.3.3. 安装和配置 Zipkin Server

要使用 Zipkin，首先需要按照本书多次完成的方式设置 Spring Boot 项目。（在本章的代码中，这是调用 zipkinsvr。）然后你需要将两个 JAR 依赖添加到 zipkinsvr/pom.xml 文件中。下面列出了这两个 JAR 依赖。

清单 9.4 Zipkin 服务需要的 JAR 依赖

```
<dependency>
   <groupId>io.zipkin.java</groupId>
   <artifactId>zipkin-server</artifactId>
</dependency>

<dependency>
   <groupId>io.zipkin.java</groupId>
   <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

①这个依赖包含了设置 Zipkin 服务器的核心类。

②这个依赖包含了用于运行 Zipkin 服务器的 UI 部分的核心类。

@EnableZipkinServer 与@EnableZipkinStreamServer：哪个注解？

有一件事，要注意上面的 JAR 依赖，它们不是基于 Spring Cloud 的依赖。虽然 Zipkin 是基于 Spring Boot 的项目，但@EnableZipkinServer 不是 Spring Cloud 注解。这个注解是 Zipkin 项目的一部分。由于 Spring Cloud 团队的确把编写的@EnableZipkinStreamServer 注解作为 Spring Cloud Sleuth 的一部分，所以这往往会让 Spring Cloud Sleuth 和 Zipkin 的新手感到困惑。@EnableZipkinStreamServer 注解简化了 Zipkin 与 RabbitMQ 和 Kafka 的使用。

我选择使用@EnableZipkinServer，因为它在本章的设置很简单。使用@EnableZipkinStream 服务器，你需要设置和配置正在跟踪的服务，并使用 Zipkin 服务器发布/监听 RabbitMQ 或 Kafka 以跟踪数据。@EnableZipkinStreamServer 注解的优点是，即使 Zipkin 服务器不可用，你也可以继续收集跟踪数据。这是因为跟踪消息将在消息队列上累积跟踪数据，直到 Zipkin 服务器可用于处理记录。如果你使用

@EnableZipkinServer 注解并且 Zipkin 服务器不可用，那么服务已发送到 Zipkin 的跟踪数据将会丢失。

在定义 Jar 依赖之后，你现在需要将@EnableZipkinServer 注解添加到你的 Zipkin 服务引导类。这个类位于 zipkinsvr/src/main/java/com/thoughtmechanix/zipkinsvr/ZipkinServerApplication.java。以下清单显示了该引导类的代码。

清单 9.5 创建你的 Zipkin 服务器引导类

Building your Zipkin servers bootstrap class

```
package com.thoughtmechanix.zipkinsvr;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import zipkin.server.EnableZipkinServer;

@SpringBootApplication
@EnableZipkinServer
public class ZipkinServerApplication {
   public static void main(String[] args) {
      SpringApplication.run(ZipkinServerApplication.class, args);
   }
}
```

①@EnableZipkinServer 允许你作为 Spring Boot 项目快速启动 Zipkin。

在这个清单中要注意的关键事情是使用@EnableZipkinServer 注解。这个注释使你能够将这个 Spring Boot 服务作为 Zipkin 服务器启动。此时，你可以构建、编译并启动 Zipkin 服务器，它作为本章的 Docker 容器之一。

运行 Zipkin 服务器需要少量的配置。Zipkin 运行时唯一需要配置的一件事就是 Zipkin 将用来存储服务跟踪数据的后端数据存储。Zipkin 支持四种不同的后端数据存储。这些数据存储是：

- In-memory data
- MySQL: http://mysql.com
- Cassandra: http://cassandra.apache.org
- Elasticsearch: http://elastic.co

默认情况下，Zipkin 使用内存数据存储来存储跟踪数据。Zipkin 团队建议不要在生产系统中使用内存数据库。内存数据库可以保存有限的数据，当 Zipkin 服务器关闭或宕机时，数据将丢失。

> ***注意***：为了本书的目的，你将使用带有内存数据存储的 Zipkin。配置 Zipkin 中使用的单个数据存储超出了本书的范围，但是如果你对该主题感兴趣，可以在 Zipkin GitHub 存储库(https://github.com/openzipkin/zipkin/tree/master/zipkin-server)中找到更多信息。

9.3.4. 设置跟踪级别

此时，你将客户端配置为与 Zipkin 服务器通信，并且已经配置服务器并准备好运行。在开始使用 Zipkin 之前，你还需要做更多的事情。你需要定义每个服务将数据写入 Zipkin 的频率。

默认情况下，Zipkin 只会将所有交易的 10%写入 Zipkin 服务器。通过在向 Zipkin 发送数据的每个服务上设置一个 Spring 属性，可以控制交易抽样。这个属性被称为 spring.sleuth.sampler.percentage。该属性值介于 0 和 1 之间：

- 值为 0 意味着 Spring Cloud Sleuth 不会向 Zipkin 发送任何事务。
- 0.5 意味着 Spring Cloud Sleuth 将发送所有交易的 50%。

出于我们的目的，你将发送所有服务的跟踪信息。为此，可以设置 spring.sleuth.sampler.percentage 的值，也可以将 Spring Cloud Sleuth 中使用的默认 Sampler 类替换为 AlwaysSampler。AlwaysSampler 可以作为 Spring Bean 注入到应用程序中。例如，许可服务将 AlwaysSampler 定义为它的 licensingservice/src/main/java/com/thoughtmechanix/licenses/Application.java 类中的 Spring Bean：

```
@Bean
public Sampler defaultSampler() { return new AlwaysSampler();}
```

Zuul、许可服务和组织服务都具有定义在其中的 AlwaysSampler，因此在本章中，所有交易都将使用 Zipkin 进行追踪。

9.3.5. 使用 Zipkin 跟踪交易

我们从一个场景开始这一节。想象一下，你是 EagleEye 应用程序的开发人员之一，并且你本周随叫随到。你从抱怨 EagleEye 应用程序中的某个屏幕运行缓慢的客户那里获得支持工单。你怀疑屏幕使用的许可服务运行缓慢。但是为什么呢？在哪里？许可服务依赖于组织服务，并且这两个服务都会调用不同的数据库。哪个服务是性能比较差呢？另外，你知道这些服务正在不断地被修改，所以有人可能会添加一个新的服务调用。了解参与用户交易的所有服务及其单独的性能时间对于支持分布式架构（如微服务架构）至关重要。

你将开始使用 Zipkin 从你的组织服务中查看两笔交易，因为它们是由 Zipkin 服务跟踪的。组织服务是一个简单的服务，只调用一个数据库。你要做的是使用 POSTMAN 发送两个到组织服务的调用(GET http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a)。组织服务调用将通过 Zuul API 网关进行传输，然后将调用定向到组织服务实例的下游。

在对组织服务进行了两个调用之后，请转至 http://localhost:9411 并查看 Zipkin 捕获的跟踪结果。从屏幕最左上角的下拉框中选择“organization service”，然后按“Find traces”按钮。图 9.11 显示了采取这些操作之后的 Zipkin 查询屏幕。

现在，如果你看图 9.11 中的屏幕，你会看到 Zipkin 捕获了两个交易。每个交易都被分解成一个或多个 span。在 Zipkin 中，span 代表一个特定的服务或调用，其中时间信息被捕获。图 9.11 中的每个交易都有三个 span：Zuul 网关中的两个 span，然后是组织服务的 span。请记住，Zuul 网关不会盲目转发 HTTP 调用。它接收传入的 HTTP 调用，终止传入

的调用，然后建立一个到目标服务的新的调用（在这种情况下，组织服务）。原始调用的终止是 Zuul 如何为进入网关的每个调用添加前置、响应和后置过滤器。这也是为什么我们在 Zuul 服务中看到两个 span。

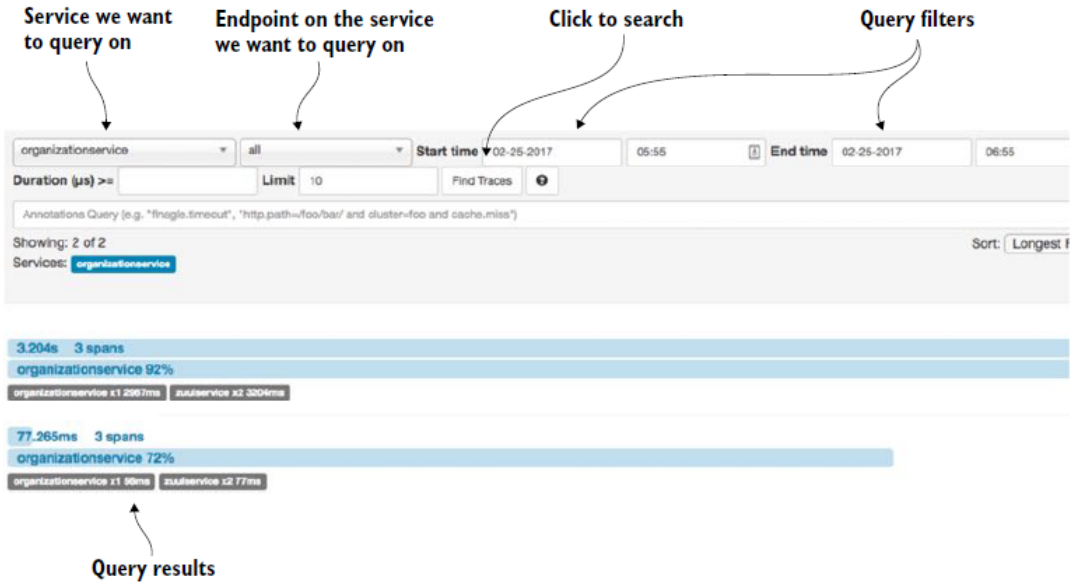

图 9.11 Zipkin 查询屏幕允许你选择要跟踪的服务以及一些基本的查询过滤器。

① *Service we want to query on*

我们要查询的服务

② *Endpoint on the service we want to query on*

我们要查询的服务端点

③ *Click to search*

点击搜索

④ *Query filters*

查询过滤器

⑤ *Query results*

查询结果

通过 Zuul 的两次对组织服务的调用分别花费了 3.204 秒和 77.2365 毫秒。因为你查询

了组织服务调用（而不是 Zuul 网关调用），你可以看到组织服务占用了交易时间总时间的 92%和 72%。

让我们深入了解最长时间的调用（3.204 秒）。你可以通过点击交易并深入细节来查看更多细节。图 9.12 显示了你单击深入了解更多细节的详细信息。

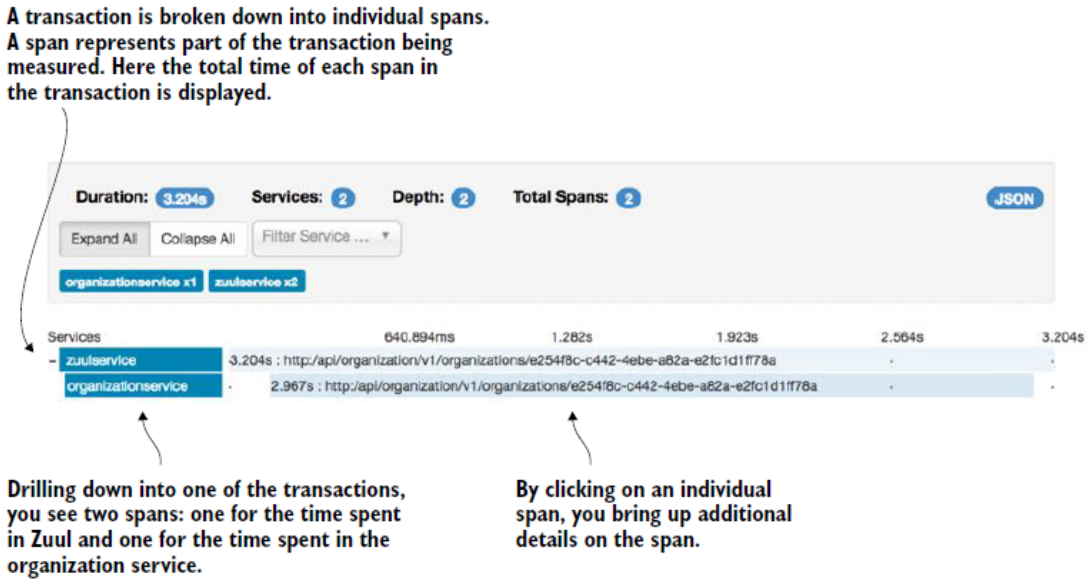

图 9.12 Zipkin 允许你深入查看交易中每个 span 所花费的时间。

① A transaction is broken down into individual spans. A span represents part of the transaction being measured. Here the total time of each span in the transaction is displayed.

一个交易被分解为单个 span。一个 span 代表被测量的交易的一部分。这里显示交易中每个 span 的总时间。

② Drilling down into one of the transactions, you see two spans: one for the time spent in Zuul and one for the time spent in the organization service.

深入到其中一个交易中，你会看到两个 span：一个是在 Zuul 花费的时间，另一个是花在组织服务上的时间。

③ By clicking on an individual span, you bring up additional details on the span.

通过单击一个单独的 span，可以提供 span 上的更多细节。

在图 9.12 中，你可以看到从 Zuul 角度来看，整个交易大约需要 3.204 秒。然而，Zuul 所做的组织服务调用在整个调用中占用了 3.204 秒的 2.967 秒。展示的每个 span 可以深入到更多的细节。点击在 organizationservice 上的 span，看看调用中可以看到哪些额外的细节。图 9.13 显示了这个调用的细节。

图 9.13 中最有价值的信息之一是对客户端（Zuul）调用组织服务的时间，组织服务接到调用的时间以及组织服务响应返回的时间进行分类。这种类型的时间信息在检测和识别网络延迟问题方面是无价的。

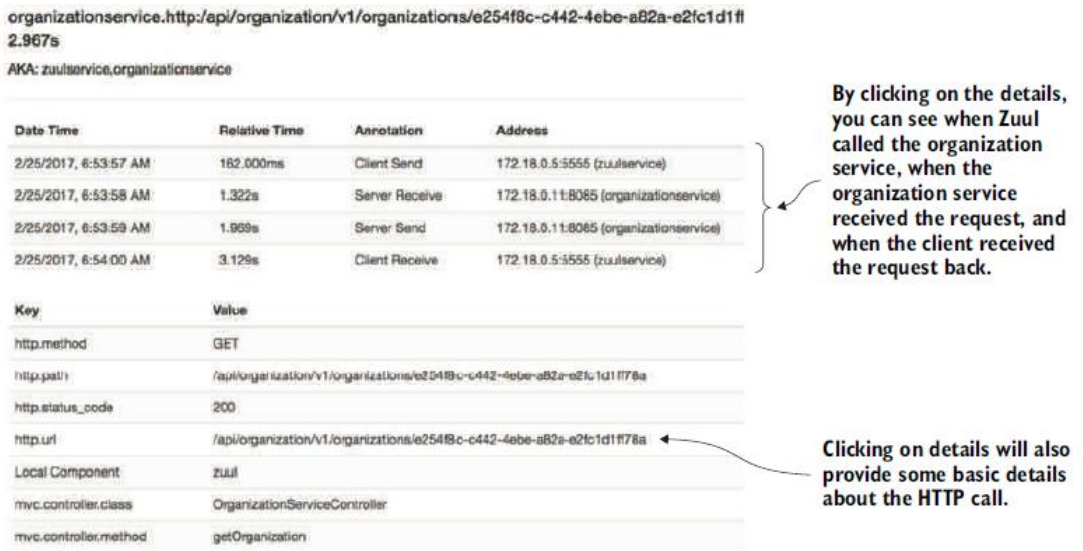

图 9.13 点击一个单独的 span 可以获得关于调用时间和 HTTP 调用细节的更多详细信息。

① By clicking on the details, you can see when Zuul called the organization service, when the organization service received the request, and when the client received the request back.

通过点击详细信息，你可以看到 Zuul 何时调用组织服务，何时组织服务收到请求，以及客户端何时收到请求返回。

② Clicking on details will also provide some basic details about the HTTP call.

点击详细信息也将提供一些关于 HTTP 调用的基本细节。

9.3.6. 可视化更复杂的交易

如果你想明确哪些服务调用之间存在服务依赖关系，该怎么办？你可以通过 Zuul 调用许可服务，然后查询 Zipkin 以获得许可服务跟踪信息。你可以通过对许可服务端点 http://localhost:5555/api/licensing/v1/organizations/e254f8cc442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82ae2fc1d1ff78a 进行 GET 调用来完成此操作。

图 9.14 显示了对许可服务的调用的详细跟踪。

图 9.14 查看许可服务调用如何从 Zuul 流向许可服务，然后流向组织服务的跟踪细节

在图 9.14 中，你可以看到对许可服务的调用涉及 4 个分离的 HTTP 调用。你看到调用到 Zuul 网关，然后从 Zuul 网关到许可服务。然后，许可服务通过 Zuul 调用组织服务。

9.3.7. 捕获消息跟踪

Spring Cloud Sleuth 和 Zipkin 不会跟踪 HTTP 调用。Spring Cloud Sleuth 还会在服务中注册的任何入站或出站消息通道上发送 Zipkin 跟踪数据。

消息传递可以在应用程序中引入自己的性能和延迟问题。服务可能不会足够快地处理来自队列的消息。或者可能有网络延迟问题。我在构建基于微服务的应用程序时遇到了所有这

些情况。

通过使用 Spring Cloud Sleuth 和 Zipkin，你可以确定消息何时从队列发布以及何时收到。你还可以看到在队列中接收到消息并进行处理时发生了什么行为。

正如你在第 8 章中的那样，无论何时添加、更新或删除组织记录，都会通过 Spring Cloud Stream 生成并发布 Kafka 消息。许可服务接收消息并更新它用于缓存数据的 Redis 键值存储。

现在，你将继续删除组织记录，并观察 Spring Cloud Sleuth 和 Zipkin 追踪的交易。你可以通过 POSTMAN 向组织服务发出 DELETE http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a。

请记住，在本章前面，我们看到了如何将 Trace ID 添加为 HTTP 响应头。你添加了一个名为 tmx-correlationid 的新 HTTP 响应头。在我的调用中，我的 tmx-correlation-id 返回值是 5e14cae0d90dc8d4。你可以通过在 Zipkin 查询屏幕右上角的搜索框中输入你的调用所返回的 Trace ID 来作为搜索 Zipkin 的特定 Trace ID。图 9.15 显示了你可以在哪里输入 Trace ID。

图 9.15 使用 HTTP 响应 tmx-correlation-id 字段中返回的 Trace ID，你可以轻松找到要查找的交易。

① Enter the trace ID here and hit Enter. This will bring up the specific trace you're looking for.

在这里输入 Trace ID，然后按 Enter 键。这将为你带来你正在寻找的具体跟踪信息。

通过 Trace ID，你可以查询 Zipkin 的具体交易，并可以看到发布删除消息到你的输出消息更改。此消息通道 ,output ,被用于发布到一个称为 orgChangeTopic 的 Kafka 主题。

图 9.16 显示了输出消息通道及其在 Zipkin 跟踪中的显示方式。

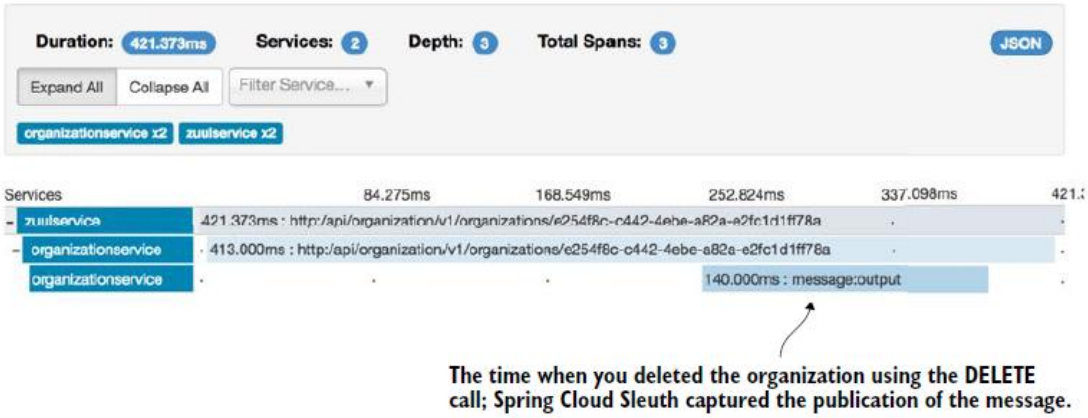

图 9.16 Spring Cloud Sleuth 将自动跟踪 Spring 消息通道上的消息的发布和接收。

① *The time when you deleted the organization using the DELETE call; Spring Cloud Sleuth captured the publication of the message.*

你使用 DELETE 调用删除组织的时间; Spring Cloud Sleuth 捕获了该消息的发布。

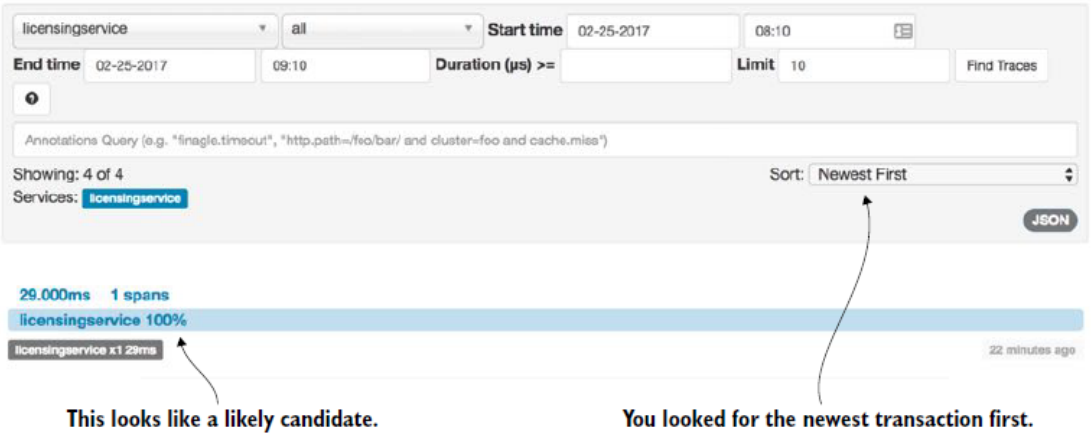

图 9.17 你正在查找接收到 Kafka 消息的许可服务调用。

① *This looks like a likely candidate.*

这看起来像一个可能的候选人。

② *You looked for the newest transaction first.*

你首先查找最新的交易。

你可以通过查询 Zipkin 并查找收到的消息来看到许可服务收到的消息。不幸的是，Spring Cloud Sleuth 不会将已发布消息的 Trace ID 传递给消息的消费者。相反，它生成一个新的 Trace ID。但是，你可以查询 Zipkin 服务器的任何许可服务交易，并通过最新的消息来订购。图 9.17 显示了这个查询的结果。

图 9.18 使用 Zipkin 可以看到组织服务发布的 Kafka 消息。

① *You can see a message received on your inboundOrgChanges channel.*

你可以看到你的 inboundOrgChanges 通道收到的消息。

现在你已经找到目标许可服务交易 ,你可以深入了解交易。图 9.18 显示了这个明细的结果。

到目前为止，你已经使用 Zipkin 来跟踪你的服务中的 HTTP 和消息传递调用。但是，如果你想要追溯到 Zipkin 没有检测到的第三方服务呢？例如，如果你想要获取特定 Redis 或 Postgres SQL 调用的跟踪和时间信息，该怎么办？幸运的是，Spring Cloud Sleuth 和 Zipkin 允许你为交易添加自定义 span，以便跟踪与这些第三方调用相关的执行时间。

9.3.8. 添加自定义 span

在 Zipkin 中添加自定义 span 非常容易。你可以开始为你的许可服务添加一个自定义 span，以便你可以追踪将数据从 Redis 中提取出来需要多长时间。然后，你将向组织服务添加自定义 span，以查看从组织数据库检索数据需要多长时间。

为许可服务到 Redis 的调用添加一个自定义 span，你将使用 licensing-service/src/main/java/com/thoughtmechanix/licenses/clients/OrganizationRestTemplateClient.java 类。在这个类中，你将要使用 checkRedisCache()方法。下面的清单显示了这个代码。

清单 9.6 通过调用来从 Redis 读取许可数据

```
import org.springframework.cloud.sleuth.Tracer;

@Component
public class OrganizationRestTemplateClient {
    @Autowired
    RestTemplate restTemplate;

    @Autowired
    Tracer tracer;

    @Autowired
    OrganizationRedisRepository orgRedisRepo;

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

    private Organization checkRedisCache(String organizationId) {
      Span newSpan = tracer.createSpan("readLicensingDataFromRedis");
      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;
      }
      finally {
        newSpan.tag("peer.service", "redis");
        newSpan.logEvent(
            org.springframework.cloud.sleuth.Span.CLIENT_RECV);
        tracer.close(newSpan);
      }
    }
}
```

①Tracer 类用于以编程方式访问 Spring Cloud Sleuth 跟踪信息。

②对于你的自定义 span，请创建一个名为“readLicensingDataFromRedis”的新 span。

③用一个 finally 块关闭 span。

④你可以将 tag 信息添加到 span。在这个类中，你提供了将由 Zipkin 捕获的服务的名称。

⑤记录一个事件，告诉 Spring Cloud Sleuth 它应该捕获调用完成的时间。

⑥关闭跟踪。如果你不调用 close()方法，则会在日志中显示错误消息，表明 span 已打开。

清单 9.6 中的代码创建了一个名为 readLicensingDataFromRedis 的自定义 span。现在，你还将向组织服务添加一个名为 getOrgDbCall 的自定义 span，以监控从 Postgres 数据库中检索组织数据所需的时间。组织服务数据库调用的跟踪可以在 organizationservice/src/main/java/com/thoughtmechanix/organization/services/OrganizationService.java 类中看到。包含自定义跟踪的方法是称为 getOrg()的方法。

以下清单显示组织服务的 getOrg()方法的源代码。

清单 9.7 使用 getOrg()方法

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

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

    @Autowired
    private Tracer tracer;

    @Autowired
    SimpleSourceBean simpleSourceBean;

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

    public Organization getOrg (String organizationId) {
        Span newSpan = tracer.createSpan("getOrgDBCall");
        logger.debug("In the organizationService.getOrg() call");
        try {
            return orgRepository.findById(organizationId);
        }finally{
            newSpan.tag("peer.service", "postgres");
            newSpan.logEvent(
            org.springframework.cloud.sleuth.Span.CLIENT_RECV);
            tracer.close(newSpan);
        }
    }
}
```

有了这两个自定义 span，重新启动服务，然后点击 GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f

8c-c338-4ebe-a82a-e2fc1d1ff78a 端点。如果我们看看 Zipkin 中的交易，你应该看到增加的两个额外 span。图 9.19 显示了在调用许可服务端点来检索许可信息时添加的其他自定义 span。

图 9.19 使用已定义的自定义 span，它们现在将显示在交易跟踪中。

① *Your custom spans now show up in the transaction trace.*

你的自定义 span 现在显示在交易跟踪中。

现在你可以从图 9.19 中看到与你的 Redis 和数据库查询相关的其他跟踪和时间信息。你可以发现，向 Redis 的读调用花费了 1.099 毫秒。由于调用没有在 Redis 缓存中找到某个数据项，因此对 Postgres 数据库的 SQL 调用花费了 4.784 毫秒。

9.4. 小结

- Spring Cloud Sleuth 允许你无缝添加跟踪信息（关联 ID）到你的微服务调用。
- 关联 ID 可以用来链接多个服务的日志条目。它们允许你查看单个交易中涉及的所有服务的交易行为。
- 虽然关联 ID 功能强大，但你需要将此概念与日志聚合平台结合使用，以便从多个来源获取日志，然后搜索和查询其内容。
- 虽然存在多个内部部署日志聚合平台，但基于云的服务允许你管理日志，而无需拥有大

规模的基础架构。它们还允许你随着应用程序日志卷的增长轻松扩展。

- 你可以将 Docker 容器与日志聚合平台集成，以捕获正在写入容器 stdout/stderr 的所有日志记录数据。在本章中，你将你的 Docker 容器、Logspout 和在线云日志记录提供程序 Papertrail 集成，以捕获和查询你的日志。
- 虽然统一的日志记录平台很重要，但通过微服务来直观地跟踪交易的能力也是一个有价值的工具。
- Zipkin 允许你在调用服务时查看服务之间存在的依赖关系。
- Spring Cloud Sleuth 与 Zipkin 集成。Zipkin 允许你以图形方式查看交易流程，并了解用户交易中涉及的每个微服务的性能特征。
- Spring Cloud Sleuth 将自动捕获启用了 Spring Cloud Sleuth 的服务中使用的 HTTP 调用和入站/出站消息通道的跟踪数据。
- Spring Cloud Sleuth 将每个服务调用映射到一个 span 的概念。Zipkin 让你看到一个 span 的性能。
- Spring Cloud Sleuth 和 Zipkin 还允许你定义自己的自定义 span，以便了解基于非 Spring 的资源（如 Postgres 或 Redis 等数据库服务器）的性能。