Red Hat build of Debezium 2.5.4

Debezium 入门

用于红帽构建的 Debezium 2.5.4

Last Updated: 2024-04-19

Red Hat build of Debezium 2.5.4 Debezium 入门

用于红帽构建的 Debezium 2.5.4

法律通告

摘要

本指南论述了如何开始使用红帽构建的 Debezium。

目录

前言 3
使开源包含更多 3
对红帽文档提供反馈 3

第 1 章 关于本教程 4

第 2 章 DEBEZIUM 简介 5

第 3 章 启动服务 6
3.1. 部署 MYSQL 数据库 6
3.2. 部署 KAFKA CONNECT 7
3.3. 示例：简单的 OPENSHIFT IMAGESTREAM 对象定义 11
3.4. 验证连接器部署 11

第 4 章 查看更改事件 16
4.1. 查看 创建事件 16
4.2. 更新数据库并查看 更新 事件 22
4.3. 删除数据库中的记录并查看 删除 事件 25
4.4. 重启 KAFKA CONNECT 服务 28

第 5 章 后续步骤 32

前言

本教程介绍了如何使用 Debezium 来捕获 MySQL 数据库中的更新。当数据库中的数据发生变化时，您可以看到生成的事件流。

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

对红帽文档提供反馈

我们感谢您对我们文档的反馈。

要改进，创建一个 JIRA 问题并描述您推荐的更改。提供尽可能多的详细信息，以便我们快速解决您的请求。

前提条件

- 您有一个红帽客户门户网站帐户。此帐户可让您登录到 Red Hat Jira Software 实例。
  如果您没有帐户，系统会提示您创建一个帐户。

步骤

1. 单击以下链接： 创建问题。
2. 在 **Summary** 文本框中输入问题的简短描述。
3. 在 **Description** 文本框中提供以下信息：
   - 找到此问题的页面的 URL。
   - 有关此问题的详细描述。
     您可以将信息保留在任何其他字段中的默认值。
4. 点 **Create** 将 JIRA 问题提交到文档团队。

感谢您花时间来提供反馈。

第 1 章 关于本教程

教程包括以下步骤：

- 将带有简单示例数据库的 MySQL 数据库服务器部署到 OpenShift。
- 在 AMQ Streams 中应用自定义资源，以自动构建包含 Debezium MySQL 连接器插件的 Kafka Connect 容器镜像。
- 创建 Debezium MySQL 连接器资源，以捕获数据库中的更改。
- 验证连接器部署。
- 查看连接器从数据库中发送到 Kafka 主题的更改事件。

先决条件

- 熟悉 OpenShift 和 AMQ Streams。
- 您可以访问安装了集群 Operator 的 OpenShift 集群。
- AMQ Streams Operator 正在运行。
- 在 OpenShift 中部署和管理 AMQ Streams 所述，Apache Kafka 集群会被部署 。
- 您有红帽构建的 Debezium 许可证。
- 您知道如何使用 OpenShift 管理工具。已安装 OpenShift **oc** CLI 客户端，或者您可以访问 OpenShift Container Platform Web 控制台。
- 根据您要存储 Kafka Connect 构建镜像的方式，您必须有访问容器 registry 的权限，或者在 OpenShift 上创建 ImageStream 资源：

  将构建镜像存储在镜像 registry 中，如 Red Hat Quay.io 或 Docker Hub

  - 在 registry 中创建和管理镜像的帐户和权限。

  将构建镜像存储为原生 OpenShift ImageStream

  - ImageStream 资源已部署到集群中，以存储新的容器镜像。您必须为集群显式创建 ImageStream。默认无法使用镜像流。

其他资源：

- 在 OpenShift Container Platform 上管理镜像流 。

第 2 章 DEBEZIUM 简介

Debezium 是一个分布式平台，可将现有数据库的信息转换为事件流，使应用程序能够检测并立即响应数据库中的行级更改。

Debezium 基于 Apache Kafka 构建，提供一组 Kafka Connect 兼容连接器。每个连接器都可用于特定的数据库管理系统(DBMS)。连接器通过检测更改并在发生更改时记录数据更改的历史记录，并将每个更改事件的记录流传输到 Kafka 主题。然后，消耗应用程序可以从 Kafka 主题读取生成的事件记录。

通过使用 Kafka 的可靠流平台，Debezium 使应用程序可以正确且完全使用数据库中发生的变化。即使应用程序意外停止，或者丢失了其连接，也不会错过停机期间发生的事件。应用程序重启后，它会从离开的时间点从主题中恢复读取。

以下教程介绍了如何通过简单的配置部署和使用 Debezium MySQL 连接器。有关部署和使用 Debezium 连接器的更多信息，请参阅连接器文档。

其他资源

- Db2 的 Debezium 连接器
- MongoDB 的 Debezium 连接器
- MySQL 的 Debezium 连接器
- Oracle 数据库的 Debezium 连接器
- PostgreSQL 的 Debezium 连接器
- SQL Server 的 Debezium 连接器

第 3 章 启动服务

使用 Debezium 需要 Kafka 和 Kafka Connect、数据库和 Debezium 连接器服务的 AMQ Streams。要为本教程运行服务，您必须：

1. 部署 MySQL 数据库
2. 使用 Debezium MySQL Connector 插件部署 Kafka 连接
3. 镜像流示例
4. 验证连接器部署

3.1. 部署 MYSQL 数据库

部署 MySQL 数据库服务器，其中包含带有数据预先填充的多个表的示例 **inventory** 数据库。Debezium MySQL 连接器将捕获示例表中发生的更改，并将更改事件记录传送到 Apache Kafka 主题。

流程

1. 运行以下命令启动 MySQL 数据库，该命令启动配置了示例 **inventory** 数据库的 MySQL 数据库服务器：

```
$ oc new-app -l app=mysql --name=mysql quay.io/debezium/example-mysql:latest
```

2. 运行以下命令，为 MySQL 数据库配置凭证，该命令更新 MySQL 数据库的部署配置以添加用户名和密码：

```
$ oc set env deployment/mysql MYSQL_ROOT_PASSWORD=debezium
MYSQL_USER=mysqluser MYSQL_PASSWORD=mysqlpw
```

3. 通过调用以下命令来验证 MySQL 数据库是否正在运行，该命令后跟显示 MySQL 数据库正在运行的输出，以及 pod 是否已就绪：

```
$ oc get pods -l app=mysql
NAME          READY   STATUS    RESTARTS   AGE
mysql-1-2gzx5   1/1     Running   1          23s
```

4. 打开一个新终端，再登录示例 **inventory** 数据库。
此命令在运行 MySQL 数据库的 pod 中打开一个 MySQL 命令行客户端。客户端使用您之前配置的用户名和密码：

```
$ oc exec mysql-1-2gzx5 -it -- mysql -u mysqluser -pmysqlpw inventory
mysql: [Warning] Using a password on the command line interface can be insecure.
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.7.29-log MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
```

```
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

5. 列出 **inventory** 数据库中的表：

```
mysql> show tables;
+--------------------+
| Tables_in_inventory |
+--------------------+
| addresses          |
| customers          |
| geom               |
| orders             |
| products           |
| products_on_hand   |
+--------------------+
6 rows in set (0.00 sec)
```

6. 探索数据库并查看其包含的数据，例如，查看 **customers** 表：

```
mysql> select * from customers;
+------+------------+-----------+-----------------------+
| id   | first_name | last_name | email                 |
+------+------------+-----------+-----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne       | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+-----------------------+
4 rows in set (0.00 sec)
```

3.2. 部署 KAFKA CONNECT

部署 MySQL 数据库后，使用 AMQ Streams 构建包含 Debezium MySQL 连接器插件的 Kafka Connect 容器镜像。在部署过程中，您可以创建并使用以下自定义资源(CR)：

- 定义 Kafka Connect 实例的 **KafkaConnect** CR，并包含有关要在镜像中包含的 MySQL 连接器工件的信息。
- **KafkaConnector** CR 提供了包括 MySQL 连接器用来访问源数据库的信息。在 AMQ Streams 启动 Kafka Connect pod 后，您可以通过应用 **KafkaConnector** CR 来启动连接器。

在构建过程中，AMQ Streams Operator 将 **KafkaConnect** 自定义资源（包括 Debezium 连接器定义）中的输入参数转换为 Kafka Connect 容器镜像。构建会从 Red Hat Maven 存储库下载所需的工件，并将它们合并到镜像中。新创建的容器被推送到在 **.spec.build.output** 中指定的容器 registry，用于部署 Kafka Connect pod。

容器镜像可以存储在外部容器 registry 中，如 **quay.io**，或存储在 OpenShift ImageStream 中。因为 ImageStreams 不会被自动创建，因此若要将容器镜像存储在 ImageStream 中，因此您必须在部署 Kafka Connect 前创建 ImageStream。

在 AMQ Streams 构建并存储 Kafka Connect 镜像后，使用 **KafkaConnector** 自定义资源来启动连接器。

先决条件

- AMQ Streams 在 OpenShift 集群上运行。
- AMQ Streams Cluster Operator 已安装到 OpenShift 集群。
- 如果您希望将 KafkaConnect 容器镜像存储在 OpenShift ImageStream 中，则提供了一个 ImageStream。
- Apache Kafka 和 Kafka Connect 在 AMQ Streams 上运行。

步骤

1. 登录 OpenShift 集群，再创建或打开一个项目，如 **debezium**。
2. 为连接器创建 Debezium **KafkaConnect** 自定义资源(CR)，或修改现有的资源。
   以下示例显示了一个 **dbz-connect.yaml** 文件的摘录，该文件描述了 **KafkaConnect** 自定义资源。
   metadata.annotations 和 **spec.build** 属性是必需的。

例 3.1. 定义包含 Debezium 连接器的 KafkaConnect 自定义资源的 dbz-connect.yaml 文件

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  replicas: 1
  version: 3.6.0
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-mysql
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/2.5.4.Final-redhat-00001/debezium-connector-mysql-2.5.4.Final-redhat-00001-plugin.zip (7)
  bootstrapServers: my-cluster-kafka-bootstrap:9093

...
```

表 3.1. Kafka Connect 配置设置的描述

项	描述
1	将 **strimzi.io/use-connector-resources** 注解设置为 **"true"**，使 Cluster Operator 使用 **KafkaConnector** 资源在此 Kafka Connect 集群中配置连接器。

项	描述
2	**spec.build** 配置指定在镜像中存储构建镜像的位置，并列出要在镜像中包含的插件，以及插件工件的位置。
3	**build.output** 指定存储新构建镜像的 registry。
4	指定镜像输出的名称和镜像名称。**output.type** 的有效值是要推送到 Docker Hub 或 Quay 等容器 registry 的有效值，或 **镜像流** 以将镜像推送到内部 OpenShift ImageStream。**要使用 ImageStream，必须将 ImageStream 资源部署到集群中。有关在 KafkaConnect 配置中指定 build.output 的更多信息，请参阅 AMQ Streams Build schema 文档。**
5	**plugins** 配置列出了您要包含在 Kafka Connect 镜像中的所有连接器。对于列表中的每个条目，指定一个插件**名称**，以及有关构建连接器所需的工件的信息。另外，对于每个连接器插件，您还可以包含可用于连接器的其他组件。例如，您可以添加 Service Registry 工件或 Debezium 脚本组件。
6	**artifacts.type** 的值指定在 **artifacts.url** 中指定的工件类型。有效类型为 **zip**、**tgz** 或 **jar**。Debezium 连接器存档以 **.zip** 文件格式提供。JDBC 驱动程序文件采用 **.jar** 格式。**类型** 值必须与 **url** 字段中引用的文件类型匹配。
7	**artifacts.url** 的值指定 HTTP 服务器的地址，如 Maven 存储库，用于存储连接器工件的文件。OpenShift 集群必须有权访问指定的服务器。

3. 输入以下命令将 **KafkaConnect** 构建规格应用到 OpenShift 集群：

```
oc create -f dbz-connect.yaml
```

根据自定义资源中指定的配置，AMQ Streams Operator 准备要部署的 Kafka Connect 镜像。构建完成后，Operator 将镜像推送到指定的 registry 或 ImageStream，并启动 Kafka Connect 集群。集群中提供了您在配置中列出的连接器工件。

4. 创建一个 **KafkaConnector** 资源来定义 MySQL 连接器的实例。
例如，创建以下 **KafkaConnector** CR，并将它保存为 **debezium-inventory-connector.yaml**

例 3.2. 为 Debezium 连接器定义 KafkaConnector 自定义资源的 mysql-inventory-connector.yaml 文件

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: my-connect-cluster
  name: inventory-connector 1
spec:
  class: io.debezium.connector.mysql.MySqlConnector 2
```

```
    tasksMax: 1 (3)
    config: (4)
      database.hostname: mysql (5)
      database.port: 3306 (6)
      database.user: debezium (7)
      database.password: dbz (8)
      database.server.id: 184054
      topic.prefix: dbserver1 (9)
      table.include.list: inventory.* (10)
      schema.history.internal.kafka.bootstrap.servers: 'my-cluster-kafka-bootstrap:9092' (11)
      schema.history.internal.kafka.topic: schema-changes.inventory (12)
```

表 3.2. 连接器配置设置的描述

项	描述
1	使用 Kafka Connect 集群注册的连接器名称。
2	连接器类的名称。
3	任何时候都只能运行一个任务。使用单一连接器任务来确保正确的顺序和事件处理，因为 MySQL 连接器读取 MySQL 服务器的 **binlog**。Kafka Connect 服务使用连接器来启动一个或多个任务来完成工作。它会在 Kafka Connect 服务集群中自动分发正在运行的任务。如果服务停止或崩溃，任务将重新分发到运行的服务。
4	连接器的配置。
5	MySQL 数据库实例的主机名或地址。
6	数据库实例的端口号。
7	Debezium 通过它连接到数据库的用户帐户名称。
8	Debezium 用于连接到数据库用户帐户的密码。
9	MySQL 服务器或集群的主题前缀。此字符串前缀连接器将事件记录发送到的每个 Kafka 主题的名称。
10	连接器捕获更改事件的表列表。连接器仅在 **清单** 表中发生时才会检测更改。
11	连接器用来写入和恢复 DDL 语句到数据库 schema 历史记录主题的 Kafka 代理列表。这与连接器发送更改事件记录的代理相同。重启后，连接器会在连接器恢复读取时恢复 binlog 中存在的数据库模式。
12	数据库架构历史记录主题的名称。本主题仅用于内部使用，不应供消费者使用。

5. 运行以下命令来创建连接器资源：

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

例如，

```
oc create -n debezium -f mysql-inventory-connector.yaml
```

连接器注册到 Kafka Connect 集群，并开始针对 **KafkaConnector** CR 中的 **spec.config.database.dbname** 指定的数据库运行。连接器 pod 就绪后，Debebe 正在运行。

现在，您已准备好 验证连接器是否已创建， 并已开始捕获 **inventory** 数据库中的更改。

3.3. 示例：简单的 OPENSHIFT IMAGESTREAM 对象定义

以下示例显示了一个简单的 **ImageStream** 对象定义

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: kafka-connect-dbz-mysql
spec:
  lookupPolicy:
    local: true
```

其他资源：

- 在 OpenShift Container Platform 中创建和管理镜像及镜像流。

3.4. 验证连接器部署

如果连接器正确启动且没有错误，它会为每个连接器配置为捕获的表创建一个主题。下游应用程序可以订阅这些主题，以检索源数据库中发生的信息事件。

要验证连接器是否正在运行，您可以从 OpenShift Container Platform Web 控制台或 OpenShift CLI 工具 (oc)执行以下操作：

- 验证连接器状态。
- 验证连接器是否生成主题。
- 验证主题是否填充了读取操作("op":"r")的事件，连接器在每个表的初始快照中生成。

先决条件

- Debezium 连接器部署到 OpenShift 上的 AMQ Streams。
- 已安装 OpenShift **oc** CLI 客户端。
- 访问 OpenShift Container Platform web 控制台。

流程

1. 使用以下方法之一检查 **KafkaConnector** 资源的状态：
   - 在 OpenShift Container Platform Web 控制台中：

a. 导航到 **Home → Search**。

b. 在 **Search** 页面中，点 **Resources** 打开 **Select Resource** 框，然后键入 **KafkaConnector**。

c. 在 **KafkaConnectors** 列表中，点您要检查的连接器的名称，如 **inventory-connector**。

d. 在 **Conditions** 部分，验证 **Type** 和 **Status** 列中的值是否已设置为 **Ready** 和 **True**。

- 在终端窗口中：

a. 使用以下命令：

```
oc describe KafkaConnector <connector-name> -n <project>
```

例如，

```
oc describe KafkaConnector inventory-connector -n debezium
```

该命令返回类似以下示例的状态信息：

例 3.3. KafkaConnector 资源状态

```
Name:         inventory-connector
Namespace:    debezium
Labels:       strimzi.io/cluster=my-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  10.131.1.124:8083
    Name:         inventory-connector
    Tasks:
      Id:               0
      State:            RUNNING
      worker_id:        10.131.1.124:8083
    Type:               source
  Observed Generation:  1
  Tasks Max:            1
  Topics:
    dbserver1
    dbserver1.inventory.addresses
    dbserver1.inventory.customers
    dbserver1.inventory.geom
    dbserver1.inventory.orders
```

```
    dbserver1.inventory.products
    dbserver1.inventory.products_on_hand
Events:  <none>
```

2. 验证连接器是否创建了 Kafka 主题：

- 通过 OpenShift Container Platform Web 控制台。
  a. 导航到 **Home → Search**。
  b. 在 **Search** 页面中，点 **Resources** 打开 **Select Resource** 框，然后键入 **KafkaTopic**。
  c. 在 **KafkaTopics** 列表中，点您要检查的主题名称，例如 **dbserver1.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**。
  d. 在 **Conditions** 部分，验证 **Type** 和 **Status** 列中的值是否已设置为 **Ready** 和 **True**。
- 在终端窗口中：
  a. 使用以下命令：

```
oc get kafkatopics
```

该命令返回类似以下示例的状态信息：

例 3.4. KafkaTopic 资源状态

```
NAME                CLUSTER  PARTITIONS  REPLICATION FACTOR  READY
connect-cluster-configs  my-cluster  1       1           True
connect-cluster-offsets  my-cluster  25      1           True
connect-cluster-status   my-cluster  5       1           True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a my-cluster
50 1 True
dbserver1---a96f69b23d6118ff415f772679da623fbbb99421 my-cluster 1 1 True
dbserver1.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480  my-cluster 1 1 True
dbserver1.inventory.customers---9931e04ec92ecc0924f4406af3fdace7545c483b
my-cluster 1 1   True
dbserver1.inventory.geom---9f7e136091f071bf49ca59bf99e86c713ee58dd5  my-
cluster 1 1   True
dbserver1.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d
my-cluster 1 1   True
dbserver1.inventory.products---df0746db116844cee2297fab611c21b56f82dcef
my-cluster 1 1   True
dbserver1.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5  my-cluster 1  1 True
schema-changes.inventory my-cluster    1          1       True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55                my-
cluster   1    1  True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b my-cluster 1    1  True
```

3. 检查主题内容。

- 在终端窗口中输入以下命令：

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

例如，

```
 oc exec -n debezium  -it my-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=dbserver1.inventory.products_on_hand
```

指定主题名称的格式与 **oc describe** 命令返回的格式与第 1 步中返回，例如 **dbserver1.inventory.addresses**。

对于主题中的每个事件，命令会返回类似以下示例的信息：

例 3.5. Debezium 更改事件的内容


```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"dbserver1.inventory.products_on_hand.Key"},"payload":{"product_id":101}} {"schema":
{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"dbserver1.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"dbserver1.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.mysql.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transaction"}],"optional":false,"name":"dbserver1.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":{"version":"2.5.4.Final-redhat-
```

00001","connector":"mysql","name":"dbserver1","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"mysql-bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"transaction":null}}

在前面的示例中，**有效**负载 值显示连接器快照从 **dbserver1.products_on_hand** 表生成 read (**op" ="r"**)事件。**product_id** 记录的 **"before"** 状态为 **null**，表示该记录不存在之前的值。**"after"** 状态对于 **product_id** 为 **101** 的项目的 **quantity** 显示为 **3**。

现在，您可以查看 Debezium 连接器从 **inventory** 数据库捕获的更改事件。

第 4 章 查看更改事件

部署 Debezium MySQL 连接器后，它开始捕获 **inventory** 数据库的更改。

当连接器启动时，它会将事件写入一组 Apache Kafka 主题，每个主题代表 MySQL 数据库中的一个表。每个主题的名称都以数据库服务器的名称 **dbserver1** 开头。

连接器写入以下 Kafka 主题：

dbserver1

应用于捕获更改的表的 DDL 语句的 schema 更改主题会被写入。

dbserver1.inventory.products

接收 **inventory** 数据库中 **products** 表的更改事件记录。

dbserver1.inventory.products_on_hand

接收 **inventory** 数据库中 **products_on_hand** 表的更改事件记录。

dbserver1.inventory.customers

接收 **inventory** 数据库中 **customer** 表的更改事件记录。

dbserver1.inventory.orders

接收 **inventory** 数据库中 **orders** 表的更改事件记录。

本教程的其余部分检查 **dbserver1.inventory.customers** Kafka 主题。当您查看这个主题的更多信息，您会看到它如何代表不同类型的更改事件，并查找有关连接器捕获的每个事件的信息。

教程包含以下部分：

- 查看 *创建事件*
- 更新数据库并查看 *更新* 事件
- 删除数据库中的记录并查看 *删除* 事件
- 重启 Kafka Connect 并更改数据库

4.1. 查看 *创建事件*

通过查看 **dbserver1.inventory.customers** 主题，您可以看到 MySQL 连接器如何在 **inventory** 数据库中捕获 *create* 事件。在这种情况下，*创建事件* 捕获正在添加到数据库中的新客户。

流程

1. 打开一个新终端，并使用 **kafka-console-consumer** 来使用主题开头的 **dbserver1.inventory.customers** 主题。
   此命令在运行 Kafka 的 Pod 中运行一个简单的消费者(**kafka-console-consumer.sh**) (**my-cluster-kafka-0**)：

```
$ oc exec -it my-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --from-beginning \
  --property print.key=true \
  --topic dbserver1.inventory.customers
```

消费者返回四个消息（采用 JSON 格式），每行一个在 customer 表中。**每个消息都包含对应表行的事件记录。**

每个事件有两个 JSON 文档：一个 *key* 和一个 *value*。键对应于行的主键，值显示行的详细信息（行包含的字段、每个字段的值以及行上执行的操作类型）。

2. **对于最后一个事件，请查看 *密钥* 的详细信息。**

以下是最后一次事件 *的密钥* 的详细信息（用于可读性的格式）：

```
{
 "schema":{
  "type":"struct",
   "fields":[
    {
     "type":"int32",
     "optional":false,
     "field":"id"
    }
   ],
  "optional":false,
  "name":"dbserver1.inventory.customers.Key"
 },
 "payload":{
  "id":1004
 }
}
```

事件有两个部分： schema 和一个 payload。模式 包含一个 Kafka Connect 模式，描述有效负载中的内容。在这种情况下，有效负载是名为 dbserver1.inventory.customers.Key 的 struct，它是可选的，且有一个必填字段（类型为 int32 的 id ）。

payload 是一个单一的 id 字段，它的值为 1004。

通过查看事件的 *key*，您可以看到此事件应用到 inventory.customers 表中的 id 主键栏的值为 1004 的行。

3. **检查同一事件 *值*。**

事件 *的值* 显示创建了行，并描述了它所包含的内容（本例中为 id、first_name、last_name，以及插入行 的电子邮件 ）。

以下是最后一次事件 *的值* 详情（用于可读性）：


```
{
 "schema": {
  "type": "struct",
  "fields": [
   {
    "type": "struct",
    "fields": [
     {
      "type": "int32",
      "optional": false,
      "field": "id"
     },
     {
      "type": "string",
      "optional": false,
      "field": "first_name"
     },
     {
      "type": "string",
      "optional": false,
      "field": "last_name"
     },
     {
      "type": "string",
      "optional": false,
      "field": "email"
     }
    ],
    "optional": true,
    "name": "dbserver1.inventory.customers.Value",
    "field": "before"
   },
   {
    "type": "struct",
    "fields": [
     {
      "type": "int32",
      "optional": false,
      "field": "id"
     },
     {
      "type": "string",
      "optional": false,
      "field": "first_name"
     },
     {
      "type": "string",
      "optional": false,
      "field": "last_name"
     },
     {
      "type": "string",
```

```
        "optional": false,
        "field": "email"
      }
    ],
    "optional": true,
    "name": "dbserver1.inventory.customers.Value",
    "field": "after"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "field": "version"
      },
      {
        "type": "string",
        "optional": false,
        "field": "name"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "server_id"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "ts_sec"
      },
      {
        "type": "string",
        "optional": true,
        "field": "gtid"
      },
      {
        "type": "string",
        "optional": false,
        "field": "file"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "pos"
      },
      {
        "type": "int32",
        "optional": false,
        "field": "row"
      },
      {
        "type": "boolean",
        "optional": true,
        "field": "snapshot"
      },
```

```
        {
          "type": "int64",
          "optional": true,
          "field": "thread"
        },
        {
          "type": "string",
          "optional": true,
          "field": "db"
        },
        {
          "type": "string",
          "optional": true,
          "field": "table"
        }
      ],
      "optional": false,
      "name": "io.debezium.connector.mysql.Source",
      "field": "source"
    },
    {
      "type": "string",
      "optional": false,
      "field": "op"
    },
    {
      "type": "int64",
      "optional": true,
      "field": "ts_ms"
    }
  ],
  "optional": false,
  "name": "dbserver1.inventory.customers.Envelope",
  "version": 1
},
"payload": {
  "before": null,
  "after": {
    "id": 1004,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": {
    "version": "2.5.4.Final",
    "name": "dbserver1",
    "server_id": 0,
    "ts_sec": 0,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 154,
    "row": 0,
    "snapshot": true,
    "thread": null,
    "db": "inventory",
    "table": "customers"
```

```
    },
    "op": "r",
    "ts_ms": 1486500577691
  }
}
```

此部分事件的时间较长，但与事件 *的密钥* 一样，它也具有 schema 和 payload。schema 包括了一个 Kafka Connect 方案，称为 dbserver1.inventory.customers.Envelope (version 1)，它可以包括五个字段：

op

包含描述操作类型的字符串值的必填字段。MySQL 连接器的值是 c 用于创建（或插入）、u 表示更新，d 表示删除，r 表示读取（快照时为 r）。

before

可选字段（如果存在）包含事件 *发生前* 行的状态。该结构由 dbserver1.inventory.customers.Value Kafka Connect 模式描述，其 dbserver1 连接器用于 inventory.customers 表中的所有行。

after

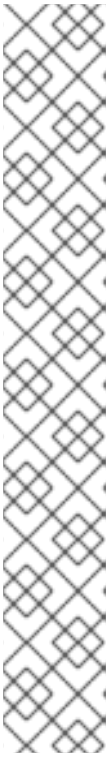
可选字段（如果存在）包含事件发生 *后* 行的状态。该结构与之前使用的 dbserver1.inventory.customers.Value Kafka Connect schema 相同。

source

包含用于描述事件源元数据的结构的必填字段（如果是 MySQL ），其中包含几个字段：连接器名称、记录事件和表的 binlog 文件的名称，该 binlog 文件中出现事件的位置，事件中的行（如果有多个是一）、受影响的数据库和表的名称， 进行更改的 MySQL 线程 ID（此事件是快照的一部分），以及 MySQL 服务器 ID 以及时间戳（以秒为单位）。

ts_ms

可选的字段（如果存在），其中包含运行 Kafka Connect 任务的 JVM 中的系统时钟（使用连接器处理事件）。



注意

事件的 JSON 表示比它们描述的行要长。这是因为，对于每个事件键和值，Kafka Connect 都会包括描述 *有效负载的 schema*。随着时间的推移，此结构可能会改变。但是，在事件本身中有键和值的架构可以更轻松地使用应用程序了解消息，特别是随着时间推移而变化。

Debezium MySQL 连接器基于数据库表的结构构建这些模式。如果您使用 DDL 语句更改 MySQL 数据库中的表定义，连接器会读取这些 DDL 语句并更新其 Kafka Connect 模式。这是每个事件的结构与事件源自于该表的唯一方式。但是，包含单个表的所有事件的 Kafka 主题可能具有与表定义的每个状态对应的事件。

JSON 转换程序在每个消息中包含 key 和 value 模式，因此它会生成非常详细的事件。

4. 将事件的 *key* 和 *value* 方案与 inventory 数据库的状态进行比较。在运行 MySQL 命令行客户端的终端中，运行以下语句：

```
mysql> SELECT * FROM customers;
+------+------------+-----------+----------------------+
| id   | first_name | last_name | email                |
+------+------------+-----------+----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne       | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+----------------------+
4 rows in set (0.00 sec)
```

这表明您检查的事件记录与数据库中的记录匹配。

4.2. 更新数据库并查看 *更新* 事件

现在，您已看到 Debezium MySQL 连接器如何捕获 inventory 数据库中的 *create* 事件，现在您将更改其中一个记录并查看连接器如何捕获它。

通过完成此步骤，您将了解如何查找数据库提交中更改的详细信息，以及如何比较更改事件，以确定更改事件何时与其他更改相关。

流程

1. 在运行 MySQL 命令行客户端的终端中，运行以下语句：

```
mysql> UPDATE customers SET first_name='Anne Marie' WHERE id=1004;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

2. 查看更新的 customers 表：

```
mysql> SELECT * FROM customers;
+------+------------+-----------+--------------------+
| id   | first_name | last_name | email              |
+------+------------+-----------+--------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com |
| 1003 | Edward     | Walker    | ed@walker.com      |
| 1004 | Anne Marie | Kretchmar | annek@noanswer.org |
+------+------------+-----------+--------------------+
4 rows in set (0.00 sec)
```

3. 切换到运行 kafka-console-consumer 的终端，以查看 *一个新的* 第五个事件。

通过更改 customers 表中的记录，Debezium MySQL 连接器会生成新事件。您应该会看到两个新 JSON 文档：一个用于事件 *的密钥*，另一个用于新事件 *的值*。

以下是 *更新* 事件 *的密钥* 的详细信息（用于可读性的格式）：

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

这个密钥 与之前事件 *的密钥* 相同。

以下是新事件 *的值*。schema 部分没有更改，因此仅显示 payload 部分（用于可读性的格式）：

```
{
 "schema": {...},
 "payload": {
  "before": { 1
    "id": 1004,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "after": { 2
    "id": 1004,
    "first_name": "Anne Marie",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": { 3
    "name": "2.5.4.Final",
    "name": "dbserver1",
    "server_id": 223344,
    "ts_sec": 1486501486,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 364,
    "row": 0,
    "snapshot": null,
    "thread": 3,
    "db": "inventory",
    "table": "customers"
  },
  "op": "u", 4
  "ts_ms": 1486501486308 5
 }
}
```

表 4.1. 更新 事件值有效负载中的字段描述

项	描述
1	**before** 字段显示数据库提交前行中存在的值。原始 **first_name** 值是 **Anne**。
2	**after** 字段显示更改事件后行的状态。**first_name** 值现在是 **Anne Marie**。

项	描述
3	**source** 字段结构有许多与之前相同的值，但 **ts_sec** 和 **pos** 字段已更改（该文件在其他情况下可能已更改）。
4	**op** 字段值现在是 **u**，表示此行因为更新而有所变化。
5	**ts_ms** 字段显示指示 Debezium 处理此事件的时间戳。

通过查看 payload 部分，您可以了解 *update* 事件的几个重要事项：

- **通过比较 before 和 after 结构，您可以确定由于提交，在受影响行中实际更改的内容。**
- **通过查看 源 结构，您可以找到有关 MySQL 的更改记录的信息（提供可追溯性）。**
- **通过将事件的 payload 部分与同一主题（或不同的主题）中的其他事件进行比较，您可以确定事件是之前、之后还是作为与另一个事件相同的 MySQL 提交的一部分。**

4.3. 删除数据库中的记录并查看 *删除* 事件

现在，您已了解 Debezium MySQL 连接器如何捕获 inventory 数据库中的 *create* 和 *update* 事件，现在您可以删除其中一个记录以查看连接器如何捕获它。

通过完成这个过程，您将了解如何查找有关 *删除* 事件的详情，以及 Kafka 如何使用 *日志压缩* 来减少 *删除* 事件的数量，同时仍然使消费者获取所有事件。

流程

1. **在运行 MySQL 命令行客户端的终端中，运行以下语句：**

```
mysql> DELETE FROM customers WHERE id=1004;
Query OK, 1 row affected (0.00 sec)
```

注意

如果上述命令因为外键约束违反情况失败，则必须使用以下声明从地址表中删除客户 *地址* 的引用：

```
mysql> DELETE FROM addresses WHERE customer_id=1004;
```

2. 切换到运行 kafka-console-consumer 的终端，以查看 *两个新* 事件。

   通过删除 customers 表中的行，Debezium MySQL 连接器会生成两个新事件。

3. 对第一个新时间检查 ***key*** 和 ***value***。

   以下是第一个新事件 *的密钥* 的详细信息（用于可读性的格式）：

```
{
 "schema": {
  "type": "struct",
  "name": "dbserver1.inventory.customers.Key"
  "optional": false,
  "fields": [
   {
    "field": "id",
    "type": "int32",
    "optional": false
   }
  ]
 },
 "payload": {
  "id": 1004
 }
}
```

   这个 ***key*** 与在前两个事件中的 ***key*** 相同。

   以下是第一个新事件 *的值* （用于可读性的格式）：

```
{
 "schema": {...},
 "payload": {
  "before": { (1)
   "id": 1004,
```

```
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,  2
    "source": {  3
      "name": "2.5.4.Final",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501558,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 725,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "d",  4
    "ts_ms": 1486501558315  5
  }
}
```

1 1 1 1

before 字段现在具有数据库提交中删除的行的状态。

2 2 2 2

after 字段为 null，因为行不再存在。

3 3 3 3

source 字段结构有许多与之前相同的值，但 ts_sec 和 pos 字段已更改(文件 在其他情况下可能已更改)。

4 4 4 4

op 字段值现在是 d，表示此行已被删除。

5 5 5 5

ts_ms 字段显示 Debezium 处理此事件的时间戳。

因此，此事件提供了一个消费者，其中包含处理删除行所需的信息。也提供了旧值，因为有些用户可能需要它们才能正确处理移除。

4. 为第二个新时间检查 *key* 和 *value*。

以下是第二个新事件 *的密钥*（用于可读性的格式）：

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

同样，这个 *key* 与前三个事件的键完全相同。

以下是同一事件 *的值*（用于可读性的格式）：

```
{
  "schema": null,
  "payload": null
}
```

如果将 Kafka 设置为 *压缩日志*，它将从主题中删除旧的消息（如果稍后位于具有相同键的主题中至少有一个消息）。最后的事件称为 *tombstone* 事件，因为它有一个 key 和一个空值。这意味着 Kafka 将删除具有相同键的所有之前消息。虽然前面的消息将被删除，但 tombstone 事件意味着，消费者仍然可以从开始中读取主题，而不是错过任何事件。

4.4. 重启 KAFKA CONNECT 服务

现在，您已看到 Debezium MySQL 连接器如何捕获 create、update 和 delete 事件，现在您会看到它如何捕获更改事件，即使它没有运行。

Kafka Connect 服务自动管理其注册连接器的任务。因此，如果它离线，当它重启时，它将启动任何非运行的任务。这意味着，即使 Debezium 没有运行，它仍然可以报告数据库中的更改。

在此过程中，您将停止 Kafka Connect，更改数据库中的一些数据，然后重启 Kafka Connect 以查看更改事件。

流程

1. 停止 Kafka Connect 服务。

   a. 打开 Kafka Connect 部署的配置：

   ```
   $ oc edit deployment/my-connect-cluster-connect
   ```

   部署配置将打开：

   ```
   apiVersion: apps.openshift.io/v1
   kind: Deployment
   metadata:
     ...
   spec:
     replicas: 1
   ...
   ```

   b. 将 spec.replicas 值更改为 0。

   c. 保存配置。

   d. 验证 Kafka Connect 服务是否已停止。

   此命令显示 Kafka Connect 服务已完成，且没有运行 pod：

   ```
   $ oc get pods -l strimzi.io/name=my-connect-cluster-connect
   NAME                                     READY   STATUS      RESTARTS   AGE
   my-connect-cluster-connect-1-dxcs9       0/1     Completed   0          7h
   ```

2. 当 Kafka Connect 服务停机时，切换到运行 MySQL 客户端的终端，并在数据库中添加新记录。

```
mysql> INSERT INTO customers VALUES (default, "Sarah", "Thompson",
"kitt@acme.com");
```

3. 重启 Kafka Connect 服务。

   a. 打开 Kafka Connect 服务的部署配置。

   ```
   $ oc edit deployment/my-connect-cluster-connect
   ```

   部署配置将打开：

   ```
   apiVersion: apps.openshift.io/v1
   kind: Deployment
   metadata:
     ...
   spec:
     replicas: 0
   ...
   ```

   b. 将 spec.replicas 值更改为 1。

   c. 保存部署配置。

   d. 验证 Kafka Connect 服务是否已重启。

   此命令显示 Kafka Connect 服务正在运行，并且 pod 已就绪：

   ```
   $ oc get pods -l strimzi.io/name=my-connect-cluster-connect
   NAME                                     READY   STATUS    RESTARTS   AGE
   my-connect-cluster-connect-2-q9kkl       1/1     Running   0          74s
   ```

4. 切换到正在运行 kafka-console-consumer.sh 的终端。新事件到达时弹出。

5. 检查您在 Kafka Connect 离线时创建的记录（用于可读性）：

   ```
   {
     ...
   ```

```
  "payload":{
    "id":1005
  }
}
{
  ...
  "payload":{
    "before":null,
    "after":{
      "id":1005,
      "first_name":"Sarah",
      "last_name":"Thompson",
      "email":"kitt@acme.com"
    },
    "source":{
      "version":"2.5.4.Final",
      "connector":"mysql",
      "name":"dbserver1",
      "ts_ms":1582581502000,
      "snapshot":"false",
      "db":"inventory",
      "table":"customers",
      "server_id":223344,
      "gtid":null,
      "file":"mysql-bin.000004",
      "pos":364,
      "row":0,
      "thread":5,
      "query":null
    },
    "op":"c",
    "ts_ms":1582581502317
  }
}
```

第 5 章 后续步骤

完成教程后，请考虑以下步骤：

- 进一步研究教程。

  使用 MySQL 命令行客户端在数据库表中添加、修改和删除行，并查看对主题的影响。请记住，您无法删除外部密钥引用的行。

- 规划 Debezium 部署。

  您可以在 OpenShift 或 Red Hat Enterprise Linux 中安装 Debezium。如需更多信息，请参阅以下：

  - 在 OpenShift 上安装 Debezium
  - 在 RHEL 上安装 Debezium

更新于 2024-04-19