



Red Hat Integration 2022.Q3

Debezium スタートガイド

Debezium 1.9.5 での使用向け

Red Hat Integration 2022.Q3 Debezium スタートガイド

Debezium 1.9.5 での使用向け

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Getting_Started_with_Debezium.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本ガイドでは、Debezium を使用する方法を説明します。

目次

はじめに	3
多様性を受け入れるオープンソースの強化	3
第1章 DEBEZIUM の紹介	5
第2章 サービスの起動	6
2.1. MYSQL データベースのデプロイ	6
2.2. KAFKA CONNECT のデプロイ	7
2.3. コネクタのデプロイメントの確認	11
第3章 変更イベントの表示	16
3.1. 作成 イベントの表示	16
3.2. データベースの更新および 更新 イベントの表示	22
3.3. データベースのレコードの削除および 削除 イベントの表示	24
3.4. KAFKA CONNECT サービスの再起動	26
第4章 次のステップ	30

はじめに

このチュートリアルでは、Debezium を使用して MySQL データベースの更新をキャプチャーする方法を紹介します。データベースのデータが変更されると、結果となるイベントストリームを確認できます。

チュートリアルには、次の手順が含まれています。

- 簡単なサンプルデータベースを含む MySQL データベースサーバーを OpenShift にデプロイします。
- AMQ Streams にカスタムリソースを適用して、Debezium MySQL コネクタプラグインを含む Kafka Connect コンテナイメージを自動的に構築します。
- Debezium MySQL コネクタリソースを作成して、データベースの変更をキャプチャーします。
- コネクタのデプロイメントを確認します。
- コネクタがデータベースから Kafka トピックに発行する変更イベントを表示します。

前提条件

- OpenShift および AMQ ストリームに精通している。
- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、[Apache Open Shift での AMQ ストリームのデプロイとアップグレード](#)に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。
- OpenShift 管理ツールの使用方法を知っている。[OpenShift oc CLI クライアントがインストールされている](#)、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、コンテナレジストリーにアクセスするためのパーミッションを持っているか、OpenShift 上に ImageStream リソースを作成する必要があります。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存するには、以下を実行します。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション。

ビルドイメージをネイティブ OpenShift ImageStream として保存します。

- [ImageStream](#) リソースがクラスターにデプロイされている。クラスターの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリ

スト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#)をご覧ください。

第1章 DEBEZIUM の紹介

Debezium は、既存のデータベースからの情報をイベントストリームに変換する分散プラットフォームであり、アプリケーションがデータベース内の行レベルの変更を検出して即座に対応できるようにします。

Debezium は [Apache Kafka](#) の上に構築され、一連の [Kafka Connect](#) 互換コネクタを提供します。各コネクタは、特定のデータベース管理システム (DBMS) で動作します。コネクタは、変更が発生したときにそれを検出し、各変更イベントのレコードを Kafka トピックにストリーミングすることで、DBMS のデータ変更の履歴を記録します。その後、消費アプリケーションは、結果のイベントレコードを Kafka トピックから読み取ることができます。

Debezium は、Kafka の信頼性の高いストリーミングプラットフォームを利用することで、アプリケーションがデータベースで発生した変更を正確かつ完全に利用できるようにします。アプリケーションが予期せず停止したり、接続が失われたりしても、停止中に発生したイベントを見逃すことはありません。アプリケーションの再起動後、中断した時点からトピックからの読み取りを再開します。

次のチュートリアルでは、簡単な設定で [Debezium MySQL コネクタ](#) をデプロイして使用方法を示します。Debezium コネクタのデプロイおよび使用の詳細については、コネクタのドキュメントを参照してください。

関連情報

- [Db2 の Debezium コネクタ](#)
- [MongoDB の Debezium コネクタ](#)
- [MySQL の Debezium コネクタ](#)
- [Oracle データベース用の Debezium コネクタ](#)
- [PostgreSQL の Debezium コネクタ](#)
- [SQL Server の Debezium コネクタ](#)

第2章 サービスの起動

Debezium を使用するには、Kafka および Kafka Connect を使用した AMQ Streams、データベース、および Debezium コネクターサービスが必要です。このチュートリアルサービスを実行するには、以下を行う必要があります。

1. [MySQL データベースのデプロイ](#)
2. [Debezium MySQL Connector プラグインでの Kafka Connect のデプロイ](#)
3. [Verify the connector deployment](#)

2.1. MYSQL データベースのデプロイ

データが事前入力されたいくつかのテーブルを含むサンプル **inventory** データベースを含む MySQL データベースサーバーをデプロイします。Debezium MySQL コネクターは、サンプルテーブルで発生する変更をキャプチャーし、変更イベントレコードを Apache Kafka トピックに送信します。

手順

1. 以下のコマンドを実行して MySQL データベースを起動します。このコマンドは、**inventory** データベースのサンプルで設定した MySQL データベースサーバーを起動します。

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

2. 以下のコマンドを実行して MySQL データベースのクレデンシャルを設定します。このコマンドによって MySQL データベースのデプロイメント設定が更新され、ユーザー名とパスワードが追加されます。

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

3. 以下のコマンドを実行して MySQL データベースが稼働していることを検証します。コマンドの実行後、MySQL データベースが稼働し、Pod の準備が整っていることを表す出力が表示されます。

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

4. 新しいターミナルを開き、**inventory** データベースのサンプルにログインします。このコマンドは、MySQL データベースを実行している Pod で MySQL コマンドラインクライアントを開きます。クライアントは以前に設定したユーザー名とパスワードを使用します。

```
$ 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. **inventory** データベースのテーブルを一覧表示します。

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

6. データベースを調べ、それに含まれるデータを確認します。たとえば、**customers** テーブルを表示します。

```
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 | annек@noanswer.org |
+----+-----+-----+-----+
4 rows in set (0.00 sec)
```

2.2. KAFKA CONNECT のデプロイ

MySQL データベースをデプロイした後、AMQ Streams を使用して、Debezium MySQL コネクタプラグインを含む Kafka Connect コンテナイメージを構築します。デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、MySQL コネクタアーティファクトに関する情報をイメージに含める **KafkaConnect** CR。
- MySQL コネクタがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクタを開始します。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクタ定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリから必要なアーティファクトをダウンロードし、イメージに組み込みます

す。新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect Pod のデプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドした後、**KafkaConnector** カスタムリソースを使用してコネクタを開始します。

手順

1. OpenShift クラスターにログインし、**debezium** などのプロジェクトを作成またはオープンします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、以下の例のように **metadata.annotations** および **spec.build** プロパティを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

例2.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.00
  build: ❷
  output: ❸
    type: imagestream ❹
    image: debezium-streams-connect:latest
  plugins: ❺
    - name: debezium-connector-mysql
    artifacts:
      - type: zip ❻
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/1.9.5.Final-redhat-❸<build_number>/debezium-connector-mysql-1.9.5.Final.zip ❼
  bootstrapServers: my-cluster-kafka-bootstrap:9093
```

表2.1 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスターオペレーターが KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。

項目	説明
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output の指定に関する詳細は、 AMQ Streams Build スキーマ参照 のドキュメントを参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクターを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクターのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタープラグインに対して、コネクターと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクターアーカイブは、 .zip ファイル形式で提供されます。JDBC ドライバーファイルは .jar 形式です。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。
7	artifacts.url の値は、コネクターアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。OpenShift クラスターは指定されたサーバーにアクセスする必要があります。

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスターに適用します。

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

AMQ Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスターを起動します。設定に一覧表示されているコネクターアーティファクトはクラスターで利用できます。

- KafkaConnector** リソースを作成して、MySQL コネクターのインスタンスを定義します。たとえば、以下の **KafkaConnector** CR を作成し、**debezium-inventory-connector.yaml** として保存します。

例2.2 Debezium コネクターの KafkaConnector カスタムリソースを定義する mysql-inventory-connector.yaml ファイル

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
```

```

    strimzi.io/cluster: my-connect-cluster
    name: inventory-connector ❶
spec:
  class: io.debezium.connector.mysql.MySqlConnector ❷
  tasksMax: 1 ❸
  config: ❹
    database.hostname: mysql ❺
    database.port: 3306 ❻
    database.user: debezium ❼
    database.password: dbz ❽
    database.server.id: 184054
    database.dbname: mydatabase ❾
    database.server.name: dbserver1 ❿
    database.include.list: inventory ⓫
    database.history.kafka.bootstrap.servers: 'my-cluster-kafka-bootstrap:9092' ⓬
    database.history.kafka.topic: schema-changes.inventory

```

表2.2 コネクター設定の説明

項目	説明
1	Kafka Connect クラスターに登録するコネクターの名前。
2	コネクタークラスの名前。
3	1度に1つのタスクのみが動作する必要があります。MySQL コネクターは MySQL サーバーの binlog を読み取るため、単一のコネクタータスクを使用して、適切な順序とイベント処理を確保します。Kafka Connect サービスは、コネクターを使用して1つ以上のタスクを開始し、作業を完了します。実行中のタスクを Kafka Connect サービスのクラスター全体に自動的に分散します。サービスが停止またはクラッシュした場合、タスクは実行中のサービスに再分散されます。
4	コネクターの設定。
5	データベースホスト。MySQL サーバー (mysql) を実行するコンテナの名前です。
6	データベースインスタンスのポート番号。
7	Debezium がデータベースに接続するユーザーアカウントの名前。
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。サーバー名は、MySQL サーバーまたはサーバーのクラスターの論理識別子です。この名前は、すべての Kafka トピックの接頭辞として使用されます。

項目	説明
11	コネクターが変更イベントをキャプチャーするテーブルの一覧。コネクターは、インベントリーデータベースの変更のみを検出します。
12	コネクターがデータベーススキーマ (イベントの送信先と同じブローカー) の履歴を格納するために使用する Kafka ブローカーとトピックを指定します。再起動後、コネクターは、コネクターが読み取りを再開したときに、binlog 内のポイントに存在していたデータベーススキーマを復元します。

5. 以下のコマンドを実行してコネクターリソースを作成します。

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

以下に例を示します。

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

コネクターは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクター Pod の準備ができると、Debezium が実行されます。

これで、[コネクターが作成されたことを確認](#)し、**inventory** データベースの変更のキャプチャーが開始されたことを確認する準備が整いました。

2.3. コネクターのデプロイメントの確認

コネクターがエラーなしで正常に起動すると、コネクターがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクターが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクターのステータスを確認します。
- コネクターがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクターが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクターは AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクタの名前をクリックします (例: **inventory-connector**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

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

以下に例を示します。

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

このコマンドは、以下の出力のようなステータス情報を返します。

例2.3 KafkaConnector リソースのステータス

```
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:
```



```

inventory_connector
inventory_connector.inventory.addresses
inventory_connector.inventory.customers
inventory_connector.inventory.geom
inventory_connector.inventory.orders
inventory_connector.inventory.products
inventory_connector.inventory.products_on_hand
Events: <none>

```

2. コネクタによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. **KafkaTopics** リストから確認するトピックの名前をクリックします (例: **inventory-connector.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例2.4 KafkaTopic リソースのステータス

```

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
inventory-connector---a96f69b23d6118ff415f772679da623fbbb99421 my-cluster
1 1 True
inventory-connector.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480 my-cluster 1 1 True
inventory-connector.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b my-cluster 1 1 True
inventory-connector.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5 my-cluster 1 1 True
inventory-connector.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d my-cluster 1 1 True
inventory-connector.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef my-cluster 1 1 True
inventory-connector.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. トピックの内容を確認します。

- 端末画面で、以下のコマンドを入力します。

```

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>

```

以下に例を示します。

```

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=inventory_connector.inventory.products_on_hand

```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory_connector.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例2.5 Debezium 変更イベントの内容

```

{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},"optional":false,"name":"inventory_conne
ctor.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":"inventory_connector
.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":"inventory_connector
.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.connecto

```

```
r.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":"transacti
on"},"optional":false,"name":"inventory_connector.inventory.products_on_hand.Envelope"
},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.5.Final-redhat-
00001","connector":"mysql","name":"inventory_connector","ts_ms":1638985247805,"snapsh
ot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":nu
ll,"file":"mysql-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}
```

上記の例では、**payload** 値は、コネクタスナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = **"r"**) イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

これで **Debezium** コネクタが **inventory** データベースからキャプチャーする変更イベントを表示する準備が整いました。

第3章 変更イベントの表示

Debezium MySQL コネクタのデプロイ後に、**inventory** データベースへの変更のキャプチャーが開始されます。

コネクタが開始すると、一連の Apache Kafka トピックにイベントが書き込まれます。各トピックは、MySQL データベース内のテーブルの1つを表します。各トピックの名前は、データベースサーバーの名前 **dbserver1** で始まります。

コネクタは、次の Kafka トピックに書き込みます。

dbserver1

変更がキャプチャーされるテーブルに適用される DDL ステートメントが書き込まれるスキーマ変更トピック。

dbserver1.inventory.products

inventory データベースの **products** テーブルの変更イベントレコードを受け取ります。

dbserver1.inventory.products_on_hand

inventory データベースの **products_on_hand** テーブルの変更イベントレコードを受け取ります。

dbserver1.inventory.customers

inventory データベースの **customers** テーブルの変更イベントレコードを受け取ります。

dbserver1.inventory.orders

inventory データベースの **orders** テーブルの変更イベントレコードを受け取ります。

このチュートリアルの残りの部分では、**dbserver1.inventory.customers** Kafka トピックを調べます。トピックを詳しく見ていくと、さまざまな種類の変更イベントがどのように表されているかがわかり、各イベントをキャプチャーしたコネクタに関する情報が見つかります。

チュートリアルには、次のセクションが含まれています。

- [作成 イベントの表示](#)
- [データベースの更新および 更新 イベントの表示](#)
- [データベースのレコードの削除および 削除 イベントの表示](#)
- [Kafka Connect の再起動およびデータベースの変更](#)

3.1. 作成 イベントの表示

dbserver1.inventory.customers トピックを表示すると、MySQL コネクタが **inventory** データベースの **作成** イベントをどのようにキャプチャーしたかが分かります。この場合、**作成** イベントは、データベースに追加された新規顧客をキャプチャーします。

手順

1. 新しいターミナルを開き、**kafka-console-consumer** を使用してトピックの最初から **dbserver1.inventory.customers** トピックを使用します。
このコマンドは、Kafka (**my-cluster-kafka-0**) を実行している Pod で簡単なコンシューマー (**kafka-console-consumer.sh**) を実行します。

```
$ 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
```

コンシューマーは、**customers** テーブルの各行に1つずつ、4つのメッセージ (JSON 形式) を返します。各メッセージには、対応するテーブル行のイベントレコードが含まれます。

各イベントには、**キー** と **値** という2つの JSON ドキュメントがあります。キーは行のプライマリーキーに対応し、値は行の詳細 (行に含まれるフィールド、各フィールドの値、および行で実行された操作のタイプ) を表します。

- 最後のイベントでは、**キー** の詳細を確認します。
最後のイベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

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

このイベントには、**schema** と **payload** の2つの部分があります。**schema** には、ペイロードの内容を記述する Kafka Connect スキーマが含まれています。この場合、ペイロードは **dbserver1.inventory.customers.Key** という名前の構造で、これはオプションではなく、必須フィールドが1つあります (タイプ **int32** の ID)。

payload には、値が **1004** の **id** フィールドが1つあります。

イベントの **key** を確認すると、このイベントは **id** の主キー列の値が **1004** である **inventory.customers** テーブルの行に提供されることが分かります。

- 同じイベントの **値** の詳細を確認します。
イベントの **値** は、行が作成されたことを示し、その行に含まれる内容が記載されています (この場合は挿入された行の **id**、**first_name**、**last_name**、および **email**)。

最後のイベントの **値** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "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": "1.9.5.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
}
}

```

イベントのこの部分ははるかに長くなりますが、イベントのキーと同様に **schema** と **payload** もあります。**schema** には、**dbserver1.inventory.customers.Envelope** (バージョン 1) という名前の Kafka Connect スキーマが含まれており、5つのフィールドを追加できます。

op

操作のタイプを記述する文字列値が含まれる必須フィールド。MySQL コネクタの値は、**c** (作成または挿入)、**u** (更新)、**d** (削除)、および **r** (読み取り、初回のスナップショットでない場合) です。

before

任意のフィールド。存在する場合は、イベント発生前の行の状態が含まれます。この構造は、**dbserver1.inventory.customers.Value** Kafka Connect スキーマによって記述され、**dbserver1** コネクタは **inventory.customers** テーブルのすべての行に使用します。

after

任意のフィールド。存在する場合は、イベント発生後の行の状態が含まれます。この構造は、**before** で使用されるのと同じ **dbserver1.inventory.customers.Value** Kafka Connect スキーマで記述されます。

source

イベントのソースメタデータを記述する構造が含まれる必須のフィールド。MySQL の場合は複数のフィールドが含まれます: コネクタ名、イベントが記録された **binlog** ファイルの名前、**binlog** ファイルでのイベント発生位置、イベント内の行 (複数ある場合)、影響を受けるデータベースおよびテーブルの名前、変更を行った MySQL スレッド ID、このイベントはスナップショットの一部であったかどうか、MySQL サーバー ID (ある場合)、および秒単位のタイムスタンプ。

ts_ms

任意のフィールド。存在する場合は、コネクタがイベントを処理した時間 (Kafka Connect タスクを実行する JVM のシステムクロックを使用) が含まれます。

注記

イベントの JSON 表現は、記述される行よりもはるかに長くなります。これは、すべてのイベントキーと値で Kafka Connect は **ペイロード** を記述する **スキーマ** を提供するためです。今後、この構造が変更される可能性があります。ただし、特に使用する側のアプリケーションが時間とともに進化する場合は、キーと値のスキーマがイベント自体にあると、メッセージを理解するのが非常に容易になります。

Debezium MySQL コネクタは、データベーステーブルの構造に基づいてこれらのスキーマを構築します。DDL ステートメントを使用して MySQL データベースのテーブル定義を変更する場合、コネクタはこれらの DDL ステートメントを読み取り、Kafka Connect スキーマを更新します。これは、イベント発生時にイベントの発生元となったテーブルと全く同じように、各イベントが構造化される唯一の方法です。ただし、単一テーブルのイベントがすべて含まれる Kafka トピックには、テーブル定義の各状態に対応するイベントが含まれる場合があります。

JSON コンバータにはすべてのメッセージのキーおよび値スキーマが含まれるため、非常に詳細なイベントを生成します。

4. イベントの **キー** および **値** スキーマを、**inventory** データベースの状態と比較します。MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

```
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 | annек@noanswer.org |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

これは、確認したイベントレコードがデータベースのレコードと一致することを示しています。

3.2. データベースの更新および 更新 イベントの表示

Debezium MySQL コネクタが **inventory** データベースで **作成** イベントをキャプチャーする方法を確認しました。次に、レコードの1つを変更し、コネクタがこれをどのようにキャプチャーするかを見てみましょう。

この手順を完了すると、データベースのコミットで変更した内容の詳細を確認する方法と、変更イベントを比較して、他の変更と関連していつ変更が発生したかを判断する方法について学ぶことができます。

手順

1. MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

```
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. 更新された **customers** テーブルを表示します。

```
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 | annек@noanswer.org   |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

3. **kafka-console-consumer** を実行しているターミナルに切り替え、**新しい** 5 番目のイベントを確認します。

customers テーブルのレコードを変更することで、Debezium MySQL コネクタは新しいイベントを生成しました。新しい JSON ドキュメントが 2 つあるはずです。1 つはイベントの **キー** のドキュメントで、もう 1 つは新しいイベントの **値** のドキュメントです。

更新 イベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key",
    "optional": false,
    "fields": [
```

```

    {
      "field": "id",
      "type": "int32",
      "optional": false
    }
  ],
  "payload": {
    "id": 1004
  }
}

```

このキーは、以前のイベントのキーと同じです。

新しいイベントの値は次のとおりです。**schema** セクションには変更がないため、**payload** セクションのみを表しています (書式を調整して読みやすくしてあります)。

```

{
  "schema": {...},
  "payload": {
    "before": { ❶
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { ❷
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { ❸
      "name": "1.9.5.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", ❹
    "ts_ms": 1486501486308 ❺
  }
}

```

❶ ❶ ❶ **before** フィールドは、データベースのコミット前の行と値の状態を表しています。

❷ ❷ ❷ **after** フィールドは、更新された行の状態を表し、**first_name** の値は **Anne Marie** になっています。

3 3 3 **source** フィールド構造体には以前と同じ値が多数ありますが、**ts_sec** および **pos** フィールドは更新されています (他の状況では **file** が変更されることがあります)。

4 4 4 **op** フィールドの値は **u** になっており、更新によってこの行が変更されたことを示しています。

5 5 5 The **ts_ms** フィールドは、Debezium がこのイベントを処理したときのタイムスタンプを示します。

payload セクションを表示すると、**更新** イベントに関する重要な情報を確認できます。

- **before** と **after** 構造を比較することで、コミットが原因で影響を受けた行で実際に何が変更されたかを判断できます。
- ソース 構造を確認して、MySQL の変更の記録に関する情報を確認できます (トレーサビリティを提供)。
- イベントの **payload** セクションと、同じトピック (または別のトピック) の他のイベントを比較することで、別のイベントと同じ MySQL コミットの前、後、または一部としてイベントが発生したかどうかを判断できます。

3.3. データベースのレコードの削除および 削除 イベントの表示

Debezium MySQL コネクタが **inventory** データベースで **作成** および **更新** イベントをキャプチャーする方法を確認しました。次に、レコードの1つを削除し、コネクタがこれをどのようにキャプチャーするかを見てみましょう。

この手順を完了すると、**削除** イベントの詳細を見つける方法と、Kafka が **ログコンパクション** を使用して、コンシューマーがすべてのイベントを取得できる状態で **削除** イベントの数を減らす方法を説明します。

手順

1. MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

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



注記

上記のコマンドが外部キー制約違反で失敗する場合は、以下のステートメントを使用して、**addresses** テーブルから顧客アドレスの参照を削除する必要があります。

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

2. **kafka-console-consumer** を実行しているターミナルに切り替え、2つの新しいイベントを表示します。
customers テーブルの行を削除することで、Debezium MySQL コネクタは2つの新しいイベントを生成しました。
3. 最初の新規イベントの **キー** および **値** を確認します。

最初の新規イベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

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

この **キー** は、これまで確認した2つのイベントの **キー** と同じです。

最初の新規イベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {...},
  "payload": {
    "before": { ❶
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, ❷
    "source": { ❸
      "name": "1.9.5.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", ❹
    "ts_ms": 1486501558315 ❺
  }
}
```

❶ **before** フィールドは、データベースのコミットで削除した行の状態を表しています。

- 2 **after** フィールドは **null** で、行が存在しなくなったことが分かります。
- 3 **source** フィールド構造体には以前と同じ値が多数ありますが、**ts_sec** および **pos** フィールドは更新されています (他の状況では **file** が変更されることがあります)。
- 4 **op** フィールドの値は **d** になっており、この行が削除されたことを示しています。
- 5 The **ts_ms** フィールドは、Debezium がこのイベントを処理するときのタイムスタンプを示します。

よって、このイベントは、行の削除を処理に必要な情報をコンシューマーに提供します。古い値も提供されます。これは、コンシューマーによっては削除を適切に処理するのに古い値が必要になることがあるからです。

4. 2つ目の新規イベントの **キー** および **値** を確認します。
2つ目の新規イベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

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

繰り返しになりますが、この **キー** は、これまで確認した3つのイベントのキーと同じです。

同じイベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

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

Kafka が **ログコンパクション** に設定されている場合、トピックの後半に同じキーを持つメッセージが1つ以上あると、トピックから古いメッセージが削除されます。この最後のイベントには、キーと空の値があるため、**tombstone** (トゥームストーン) イベントと呼ばれます。これは、Kafka が同じキーを持つこれまでのメッセージをすべて削除することを意味します。これまでのメッセージが削除されても、tombstone イベントであるため、コンシューマーは最初からトピックを読み取ることができ、イベントを見逃しません。

3.4. KAFKA CONNECT サービスの再起動

Debezium MySQL コネクタが作成、更新、および削除イベントをキャプチャーする方法を確認しました。次に、稼働していない場合でもどのように変更イベントをキャプチャーするかを見てみましょう。

Kafka Connect サービスは、登録されたコネクタのタスクを自動的に管理します。したがって、オフラインになると、再起動時に稼働していないタスクがすべて開始されます。つまり、Debezium が稼働していない場合でも、変更をデータベースに報告できます。

この手順では、Kafka Connect を停止し、データベースのデータを一部変更した後、Kafka Connect を再起動して変更イベントを確認します。

手順

1. Kafka Connect サービスを停止します。

- a. Kafka Connect サービスのデプロイメント設定を開きます。

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

デプロイメント設定が表示されます。

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

- b. **spec.replicas** の値を **0** に変更します。

- c. デプロイメント設定を保存します。

- d. Kafka Connect サービスが停止したことを確認します。

このコマンドを実行すると、Kafka Connect サービスが完了し、稼働している Pod がないことを確認できます。

```
$ 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. Kafka Connect サービスが停止している間に、MySQL クライアントを実行しているターミナルに切り替え、新しいレコードをデータベースに追加します。

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

3. Kafka Connect サービスを再起動します。

- a. Kafka Connect サービスのデプロイメント設定を開きます。

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

デプロイメント設定が表示されます。

```

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

```

- b. **spec.replicas** の値を **1** に変更します。
- c. デプロイメント設定を保存します。
- d. Kafka Connect サービスが再起動したことを確認します。
このコマンドを実行すると、Kafka Connect サービスは稼働中で、Pod の準備ができてい
ることを確認できます。

```

$ 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. **kafka-console-consumer.sh** を実行しているターミナルに切り替えます。新しいイベントを受
け取ると表示されます。
5. Kafka Connect がオフラインだったときに作成したレコードを確認します (書式を調整して読み
やすくしてあります)。

```

{
  ...
  "payload":{
    "id":1005
  }
}
{
  ...
  "payload":{
    "before":null,
    "after":{
      "id":1005,
      "first_name":"Sarah",
      "last_name":"Thompson",
      "email":"kitt@acme.com"
    },
    "source":{
      "version":"1.9.5.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4章 次のステップ

チュートリアルが完了したら、以下のステップを検討します。

- チュートリアルをさらに試してみる。
MySQL コマンドラインクライアントを使用して、データベーステーブルの行を追加、変更、および削除し、トピックへの影響を確認します。外部キーによって参照される行は削除できないことに注意してください。
- Debezium のデプロイメントを計画する。
Debezium を OpenShift または Red Hat Enterprise Linux にインストールできます。詳細は、以下を参照してください。
 - [Debezium の OpenShift へのインストール](#)
 - [Debezium の RHEL へのインストール](#)

改訂日時: 2022-11-27 05:21:57 +1000