Red Hat AMQ 2021.q2

『Using AMQ Streams on RHEL』

Red Hat Enterprise Linux 上で AMQ Streams 1.7 を使用

Last Updated: 2021-06-17

Red Hat AMQ 2021.q2 『Using AMQ Streams on RHEL』

Red Hat Enterprise Linux 上で AMQ Streams 1.7 を使用

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

概要

本ガイドでは、Red Hat AMQ Streams をインストール、設定、および管理して、大規模なメッセージングネットワークを構築する方法を説明します。

目次

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 プロパティー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。これは大規模な取り組みであるため、これらの変更は今後の複数のリリースで段階的に実施されます。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 AMQ STREAMS の概要

Red Hat AMQ Streams は、Apache ZooKeeper プロジェクトおよび Apache Kafka プロジェクトをベースとする非常にスケーラブルで分散され、高パフォーマンスのデータストリーミングプラットフォームです。

主なコンポーネントは、以下のとおりです。

Kafka Broker

クライアントを消費するためのクライアントの生成からレコードを配信するメッセージングブローカー。
Apache ZooKeeper は Kafka のコア依存関係であり、信頼性の高い分散コーディネーションを実現するためにクラスター調整サービスを提供します。

Kafka Streams API

ストリームプロセッサーアプリケーションを **記述する** API。

プロデューサーおよびコンシューマー API

Kafka ブローカーとの間でメッセージを生成および消費するための Java ベースの API。

Kafka Bridge

AMQ Streams Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターとの対話を可能にする RESTful インターフェースが提供されます。

Kafka Connect

Connector プラグインを使用して Kafka ブローカーと他のシステム間でデータをストリーミングするツールキット。

Kafka MirrorMaker

データセンター内またはデータセンター全体の 2 つの Kafka クラスター間でデータを複製します。

Kafka Exporter

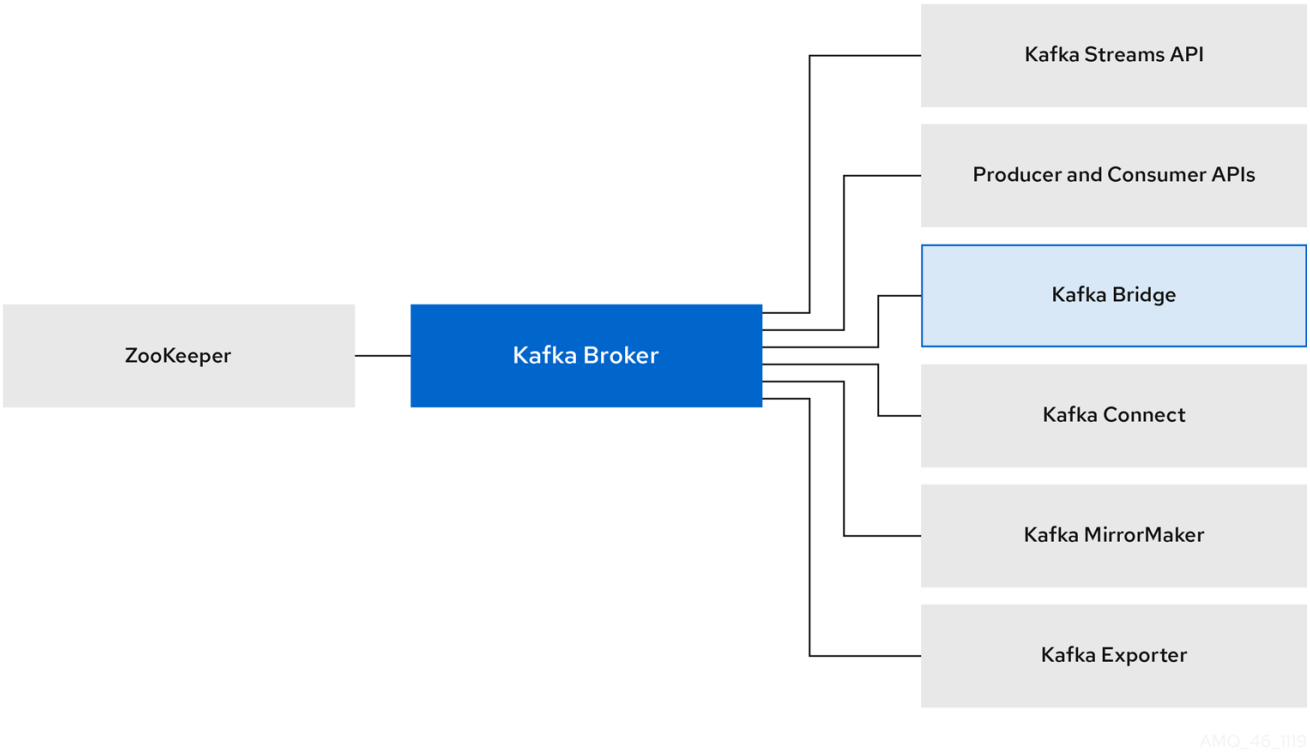
監視用に Kafka メトリクスデータの抽出に使用されるエクスポーター。

Kafka Cruise Control

最適化ゴールと容量制限を基にして、Kafka クラスターをリバランスします。

Kafka ブローカーのクラスターは、これらのコンポーネントをすべて接続するハブです。ブローカーは、設定データを保存およびクラスターの調整に Apache ZooKeeper を使用します。Apache Kafka の実行前に、Apache ZooKeeper クラスターを用意する必要があります。

図1.1 AMQ Streams アーキテクチャー



1.1. KAFKA の機能

Kafka の基盤のデータストリーム処理機能とコンポーネントアーキテクチャーによって以下が提供されます。

- スループットが非常に高く、レイテンシーが低い状態でデータを共有するマイクロサービスおよびその他のアプリケーション。
- メッセージの順序の保証。
- アプリケーションの状態を再構築するためにデータストレージからメッセージを巻き戻し/再生。
- キーバリューログの使用時に古いレコードを削除するメッセージ圧縮。
- クラスター設定での水平スケーラビリティー。
- 耐障害性を制御するデータのレプリケーション。
- 即座にアクセスするために大容量のデータを保持。

1.2. KAFKA のユースケース

Kafka の機能は、以下に適しています。

- イベント駆動型のアーキテクチャー。
- アプリケーションの状態変更をイベントのログとしてキャプチャーするイベントソーシング。
- メッセージのブローカー。

- Web サイトアクティビティーの追跡。
- メトリクスによるオペレーションの監視。
- ログの収集および集計。
- 分散システムのログのコミット。
- アプリケーションがリアルタイムでデータに対応できるようにするストリーム処理。

1.3. サポートされる構成

AMQ Streams をサポートされる構成で実行するには、以下の JVM バージョンの 1 つと、サポートされるオペレーティングシステムの 1 つで実行する必要があります。

表1.1 サポートされる Java 仮想マシンの一覧

Java 仮想マシン	バージョン
OpenJDK	1.8, 11
OracleJDK	1.8, 11
IBM JDK	1.8

表1.2 サポート対象オペレーティングシステムの一覧

オペレーティングシステム	アーキテクチャー	バージョン
Red Hat Enterprise Linux (RHEL)	x86_64	7.x, 8.x

1.4. ドキュメントの規則

置き換え可能なテキスト

本書では、置き換え可能なテキストは、**monospace** フォントのイタリック体、大文字、およびハイフンで記載されています。

たとえば、以下のコードでは **BOOTSTRAP-ADDRESS** および **TOPIC-NAME** を独自のアドレスおよびトピック名に置き換えます。

```
bin/kafka-console-consumer.sh --bootstrap-server BOOTSTRAP-ADDRESS --topic TOPIC-NAME --from-beginning
```

第2章 スタートガイド

2.1. AMQ STREAMS ディストリビューション

AMQ Streams は単一の ZIP ファイルとして配布されます。この ZIP ファイルには、以下の AMQ Streams コンポーネントが含まれます。

- Apache ZooKeeper
- Apache Kafka
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- Kafka Exporter

Kafka Bridge および Cruise Control コンポーネントは、個別の zip アーカイブとして提供されます。

- Kafka Bridge
- Cruise Control

2.2. AMQ STREAMS アーカイブのダウンロード

AMQ Streams のアーカイブされたディストリビューションは、Red Hat の Web サイトからダウンロードできます。以下の手順に従って、ディストリビューションのコピーをダウンロードできます。

手順

- カスタマーポータルから最新バージョンの Red Hat AMQ Streams アーカイブを ダウンロード します。

2.3. AMQ STREAMS のインストール

以下の手順に従って、最新バージョンの AMQ Streams を Red Hat Enterprise Linux にインストールします。

既存のクラスターを AMQ Streams 1.7 にアップグレードする方法は、「AMQ Streams および Kafka のアップグレード」を参照してください。

前提条件

- インストールアーカイブ をダウンロードします。
- レビュー 「サポートされる構成」

手順

1. 新しい **kafka** ユーザーおよびグループを追加します。

```
sudo groupadd kafka
sudo useradd -g kafka kafka
sudo passwd kafka
```

2. **/opt/kafka** ディレクトリーを作成します。

```
sudo mkdir /opt/kafka
```

3. 一時ディレクトリーを作成し、AMQ Streams ZIP ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams_y.y-x.x.x.zip -d /tmp/kafka
```

4. 展開したコンテンツを **/opt/kafka** ディレクトリーに移動し、一時ディレクトリーを削除します。

```
sudo mv /tmp/kafka/kafka_y.y-x.x.x/* /opt/kafka/
rm -r /tmp/kafka
```

5. **/opt/kafka** ディレクトリーの所有権を **kafka** ユーザーに変更します。

```
sudo chown -R kafka:kafka /opt/kafka
```

6. ZooKeeper データを保存し、その所有権を **kafka** ユーザーに設定する **/var/lib/zookeeper** ディレクトリーを作成します。

```
sudo mkdir /var/lib/zookeeper
sudo chown -R kafka:kafka /var/lib/zookeeper
```

7. Kafka データを保存し、その所有権を **kafka** ユーザーに設定する **/var/lib/kafka** ディレクトリーを作成します。

```
sudo mkdir /var/lib/kafka
sudo chown -R kafka:kafka /var/lib/kafka
```

2.4. データストレージに関する留意事項

効率的なデータストレージインフラストラクチャーは、AMQ Streams のパフォーマンスを最適化するために不可欠です。

AMQ Streams にはブロックストレージが必要で、Amazon Elastic Block Store(EBS)などのクラウドベースのブロックストレージソリューションと適切に機能します。ファイルストレージの使用は推奨されません。

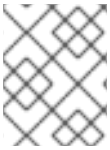
可能な場合はローカルストレージを選択します。ローカルストレージが利用できない場合は、ファイバーチャネルや iSCSI などのプロトコルがアクセスするストレージエリアネットワーク(SAN)を使用できます。

2.4.1. Apache Kafka および ZooKeeper ストレージのサポート

Apache Kafka と ZooKeeper には別々のディスクを使用します。

Kafka は JBOD(Bunch of Disks)ストレージ、複数のディスクまたはボリュームのデータストレージ設定をサポートします。JBOD では、Kafka ブローカーのデータストレージが向上します。また、パフォーマンスを向上することもできます。

ソリッドステートドライブ (SSD) は必須ではありませんが、複数のトピックに対してデータが非同期的に送受信される大規模なクラスターで Kafka のパフォーマンスを向上させることができます。SSD は、高速で低レイテンシーのデータアクセスが必要な ZooKeeper で特に有効です。



注記

Kafka と ZooKeeper の両方にデータレプリケーションが組み込まれているため、複製されたストレージのプロビジョニングは必要ありません。

2.4.2. ファイルシステム

XFS ファイルシステムを使用するようにストレージシステムを設定することが推奨されます。AMQ Streams は ext4 ファイルシステムとも互換性がありますが、最適化するには追加の設定が必要になる場合があります。

関連情報

- XFS の詳細 は、「XFS ファイルシステム」を参照してください。

2.5. 単一ノードの AMQ STREAMS クラスターの実行

この手順では、同じホスト上で実行される単一の Apache ZooKeeper ノードと 1 つの Apache Kafka ノードで構成される基本的な AMQ Streams クラスターを実行する方法を説明します。デフォルトの設定ファイルは ZooKeeper および Kafka に使用されます。

警告

単一ノードの AMQ Streams クラスターは信頼性および高可用性を提供しないため、開発での使用のみに適しています。

前提条件

- AMQ Streams がホストにインストールされていること。

クラスターの実行

1. ZooKeeper 設定ファイル **/opt/kafka/config/zookeeper.properties** を編集します。**dataDir** オプションを **/var/lib/zookeeper/** に設定します。

```
dataDir=/var/lib/zookeeper/
```

2. Kafka 設定ファイル **/opt/kafka/config/server.properties** を編集します。**log.dirs** オプションを **/var/lib/kafka/** に設定します。

```
log.dirs=/var/lib/kafka/
```

3. **kafka** ユーザーに切り替えます。

```
su - kafka
```

4. ZooKeeper を起動します。

```
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

5. ZooKeeper が稼働していることを確認します。

```
jcmd | grep zookeeper
```

以下を返します。

```
number org.apache.zookeeper.server.quorum.QuorumPeerMain
/opt/kafka/config/zookeeper.properties
```

6. Kafka を起動します。

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

7. Kafka が実行されていることを確認します。

```
jcmd | grep kafka
```

以下を返します。

```
number kafka.Kafka /opt/kafka/config/server.properties
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。

2.6. クラスターの使用

この手順では、Kafka コンソールプロデューサーおよびコンシューマークライアントを起動し、これを使用して複数のメッセージを送受信する方法を説明します。

手順 1 つに新しいトピックが自動的に作成されます。トピックの自動作成は、**auto.create.topics.enable** 設定プロパティーを使用して制御されます（デフォルトでは **true** に設定されます）。または、クラスターを使用する前にトピックを設定し、作成することができます。詳細は、「 トピック 」を参照してください。

前提条件

- AMQ Streams がホストにインストールされていること。
- ZooKeeper および Kafka が稼働している必要があります。

手順

1. Kafka コンソールプロデューサーを起動し、メッセージを新しいトピックに送信するように設定します。

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list <bootstrap-address> --topic <topic-name>
```

以下に例を示します。

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-topic
```

2. コンソールに複数のメッセージを入力します。**Enter** キーを押して、個別のメッセージを新しいトピックに送信します。

```
>message 1
>message 2
>message 3
>message 4
```

Kafka が新しいトピックを自動的に作成すると、トピックが存在しないことを示す警告が表示されることがあります。

```
WARN Error while fetching metadata with correlation id 39 :
{4-3-16-topic1=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
```

さらにメッセージを送信した後も警告が表示されるはずです。

3. 新しいターミナルウィンドウで、Kafka コンソールコンシューマーを起動し、新しいトピックの最初からメッセージを読み取るように設定します。

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server <bootstrap-address> --topic <topic-name> --from-beginning
```

以下に例を示します。

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic --from-beginning
```

受信メッセージがコンシューマーコンソールに表示されます。

4. プロデューサーコンソールに切り替え、追加のメッセージを送信します。コンシューマーコンソールに表示されていることを確認します。

5. **Ctrl+C** を押して、Kafka コンソールプロデューサーを停止し、コンシューマーを削除します。

2.7. AMQ STREAMS サービスの停止

スクリプトを実行して、Kafka サービスおよび ZooKeeper サービスを停止できます。Kafka および ZooKeeper サービスへのすべての接続が終了します。

前提条件

- AMQ Streams がホストにインストールされていること。

- ZooKeeper および Kafka が稼働している必要があります。

手順

1. Kafka ブローカーを停止します。

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

2. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

3. ZooKeeper を停止します。

```
su - kafka
/opt/kafka/bin/zookeeper-server-stop.sh
```

2.8. AMQ STREAMS の設定

前提条件

- AMQ Streams がホストにダウンロードされ、インストールされている。

手順

1. テキストエディターで ZooKeeper および Kafka ブローカー設定ファイルを開きます。設定ファイルは以下にあります。

   ZooKeeper

   /opt/kafka/config/zookeeper.properties

   Kafka

   /opt/kafka/config/server.properties

2. 設定オプションを編集します。設定ファイルは Java プロパティー形式になります。すべての設定オプションは、以下の形式で別々の行に指定する必要があります。

   ```
   <option> = <value>
   ```

   # または **!** で始まる行はコメントとして処理され、AMQ Streams コンポーネントによって無視されます。

   ```
   # This is a comment
   ```

   改行やキャリアリッジを返す前に \ を直接使用して、複数の行に分割することができます。

   ```
   sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
       username="bob" \
       password="bobs-password";
   ```

3. 変更を保存します。

4. ZooKeeper または Kafka ブローカーを再起動します。
5. クラスターのすべてのノードでこの手順を繰り返します。

第3章 ZOOKEEPER の設定

Kafka は ZooKeeper を使用して設定データを保存およびクラスターの調整に使用します。複製された ZooKeeper インスタンスのクラスターを実行することが強く推奨されます。

3.1. 基本設定

最も重要な ZooKeeper 設定オプションは次のとおりです。

tickTime

ZooKeeper の基本的な時間単位（ミリ秒単位）。これはハートビートおよびセッションタイムアウトに使用されます。たとえば、セッションタイムアウトの最小数は 2 ティックです。

dataDir

ZooKeeper がトランザクションログとそのスナップショットを保存するディレクトリー。これは、インストール時に作成された **/var/lib/zookeeper/** ディレクトリーに設定する必要があります。

clientPort

クライアントが接続できるポート番号。デフォルトは **2181** です。

config/zookeeper.properties という名前の ZooKeeper 設定ファイルの例は AMQ Streams インストールディレクトリーにあります。ZooKeeper のレイテンシーを最小限に抑えるために、**dataDir** ディレクトリーを別のディスクデバイスに配置することが推奨されます。

ZooKeeper 設定ファイルは **/opt/kafka/config/zookeeper.properties** にあります。設定ファイルの基本例は、以下を参照してください。設定ファイルは、**kafka** ユーザーが読み取り可能でなければなりません。

```
tickTime=2000
dataDir=/var/lib/zookeeper/
clientPort=2181
```

3.2. ZOOKEEPER クラスターの設定

多くの実稼働環境では、複製された ZooKeeper インスタンスのクラスターをデプロイすることが推奨されます。信頼性の高い ZooKeeper サービスでは、ZooKeeper クラスターの安定性および高可用性が重要になります。ZooKeeper クラスターは ensembles とも呼ばれ **ます**。

ZooKeeper クラスターは、通常最適なノード数で構成されます。ZooKeeper では、クラスター内のノードの大半が稼働している必要があります。以下に例を示します。

- 3 つのノードで構成されるクラスターでは、最低でも 2 つのノードが稼働している必要があります。つまり、1 つのノードが停止していることを意味します。
- クラスターでは、5 つのノードで構成されるクラスターでは、3 つ以上のノードが利用可能でなければなりません。これは、2 つのノードが停止していることを意味します。
- 7 つのノードで構成されるクラスターでは、4 つ以上のノードが利用可能である必要があります。つまり、ダウンしている 3 つのノードを許容できることを意味します。

ZooKeeper クラスターでより多くのノードがあると、クラスター全体の回復性および信頼性が向上します。

ZooKeeper は、偶数のノードを使用してクラスターで実行することができます。ただし、追加のノードはクラスターの回復性を増やしません。4 つのノードで構成されるクラスターでは、少なくとも 3 つの

ノードが利用可能必要があり、1つのノードのみを許容できます。そのため、3 つのノードのみを持つクラスターと同じ回復性があります。

理想的には、異なる ZooKeeper ノードは、異なるデータセンターまたはネットワークセグメントに配置する必要があります。ZooKeeper ノードの数を増やすと、クラスター同期で費やされたワークロードが増加します。ほとんどの Kafka ユースケースでは、3、5、または 7 のノードで構成される ZooKeeper クラスターで十分です。

警告

3 つのノードで構成される ZooKeeper クラスターは、1 つの利用できないノードのみを許容できます。つまり、ZooKeeper クラスターの別のノードでメンテナンスを実施している間にクラスターノードがクラッシュすると、ZooKeeper クラスターが利用できないことになります。

複製された ZooKeeper 設定は、スタンドアロン設定でサポートされるすべての設定オプションをサポートします。クラスタリング設定の追加オプションが追加されます。

initLimit

フォロワーがクラスターリーダーに接続および同期できるようにする時間。この時間は多数のティックとして指定されます（詳細は、**timeTick オプション** を使用します）。

syncLimit

フォロワーがリーダーの背後にある時間。この時間は多数のティックとして指定されます（詳細は、**timeTick オプション** を使用します）。

reconfigEnabled

動的再設定 を有効または無効にします。サーバーを ZooKeeper クラスターに追加または削除するには、有効にする必要があります。

standaloneEnabled

1 台のサーバーでのみ ZooKeeper を実行するスタンドアロンモードを有効または無効にします。

上記のオプションに加え、すべての設定ファイルに ZooKeeper クラスターのメンバーである必要があるサーバーのリストが含まれている必要があります。サーバーレコードは **server.id=hostname:port1:port2** の形式で指定する必要があります。

id

ZooKeeper クラスターノードの ID。

hostname

ノードが接続をリッスンするホスト名または IP アドレス。

port1

イントラクラスター通信に使用されるポート番号。

port2

リーダーの選択に使用するポート番号。

以下は、3 つのノードで構成される ZooKeeper クラスターの設定ファイルの例になります。

```
timeTick=2000
```

```
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
reconfigEnabled=true
standaloneEnabled=false

server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
```

注記

ZooKeeper 3.5.7 では、使用する前に 4 文字の単語 コマンドを許可リストに追加する必要があります。詳細は、ZooKeeper ドキュメント を参照してください。

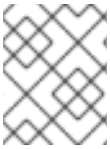
myid files

ZooKeeper クラスターの各ノードには一意の **ID** を割り当てる必要があります。各ノードの **ID** は **myid** ファイルで設定して、**/var/lib/zookeeper/** などの **dataDir** フォルダーに保存する必要があります。**myid** ファイルには、**ID** がテキストとして書き込まれた 1 行のみ含まれている必要があります。**ID** には、1 から 255 までの任意の整数を使用できます。このファイルは、各クラスターノードに手動で作成する必要があります。このファイルを使用して、各 ZooKeeper インスタンスは設定ファイルの対応する **server.** 行の設定を使用してそのリスナーを設定します。また、その他すべての **server.** 行を使用して、他のクラスターメンバーも特定します。

上記の例では、3 つのノードがあるため、それぞれに **1**、**2**、および **3** の値がそれぞれ異なる **myid** があります。

3.3. マルチノードの ZOOKEEPER クラスターの実行

この手順では、ZooKeeper をマルチノードクラスターとして設定および実行する方法を説明します。



注記

ZooKeeper 3.5.7 では、使用する前に 4 文字の単語 コマンドを許可リストに追加する必要があります。詳細は、ZooKeeper ドキュメント を参照してください。

前提条件

- AMQ Streams は、ZooKeeper クラスターノードとして使用されるすべてのホストにインストールされます。

クラスターの実行

1. **/var/lib/zookeeper/** に **myid** ファイルを作成します。最初の ZooKeeper ノードの ID **1**、次の ZooKeeper ノードの **2** を入力します。

   ```
   su - kafka
   echo "<NodeID>" > /var/lib/zookeeper/myid
   ```

   以下に例を示します。

```
su - kafka
echo "1" > /var/lib/zookeeper/myid
```

2. 以下のように ZooKeeper の **/opt/kafka/config/zookeeper.properties** 設定ファイルを編集します。

   - **dataDir** オプションを **/var/lib/zookeeper/** に設定します。
   - **initLimit** および **syncLimit** オプションを設定します。
   - **reconfigEnabled** および **standaloneEnabled** オプションを設定します。
   - すべての ZooKeeper ノードのリストを追加します。一覧には現在のノードも含まれます。

   5 メンバーを持つ ZooKeeper クラスターのノードの設定例

   ```
   timeTick=2000
   dataDir=/var/lib/zookeeper/
   initLimit=5
   syncLimit=2
   reconfigEnabled=true
   standaloneEnabled=false

   server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
   server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
   server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
   server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
   server.5=172.17.0.5:2888:3888:participant;172.17.0.5:2181
   ```

3. デフォルトの設定ファイルで ZooKeeper を起動します。

   ```
   su - kafka
   /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
   ```

4. ZooKeeper が稼働していることを確認します。

   ```
   jcmd | grep zookeeper
   ```

5. クラスターのすべてのノードでこの手順を繰り返します。

6. クラスターのすべてのノードを起動して実行したら、**ncat** ユーティリティーを使用して各ノードに **stat** コマンドを送信して、すべてのノードがクラスターのメンバーであることを確認します。

   ncat stat を使用してノードのステータスを確認します。

   ```
   echo stat | ncat localhost 2181
   ```

   この出力に、ノードが **leader** または **follower** のいずれかであることが確認できます。

   ncat コマンドの出力例

   ```
   ZooKeeper version: 3.4.13-2d71af4dbe22557fda74f9a9b4309b15a7487f03, built on
   06/29/2018 00:39 GMT
   ```

```
Clients:
 /0:0:0:0:0:0:0:1:59726[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 2
Sent: 1
Connections: 1
Outstanding: 0
Zxid: 0x200000000
Mode: follower
Node count: 4
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。

3.4. 認証

デフォルトでは、ZooKeeper は認証形式を使用せず、匿名接続を許可します。ただし、Simple Authentication and Security Layer(SASL)を使用した認証の設定に使用できる Java Authentication and Authorization Service(JAAS)をサポートします。ZooKeeper は、ローカルで保存された認証情報を使用した DIGEST-MD5 SASL メカニズムを使用した認証をサポートします。

3.4.1. SASL での認証

JAAS は別の設定ファイルを使用して設定します。JAAS 設定ファイルを ZooKeeper 設定と同じディレクトリー(/**opt**/**kafka**/**config**/)に配置することが推奨されます。推奨されるファイル名は **zookeeper-jaas.conf** です。複数のノードで ZooKeeper クラスターを使用する場合は、すべてのクラスターノードに JAAS 設定ファイルを作成する必要があります。

JAAS はコンテキストを使用して設定されます。サーバーとクライアントなどの個別の部分は常に個別の **コンテキスト** で設定されます。コンテキストは **設定** オプションで、以下の形式になります。

```
ContextName {
      param1
      param2;
};
```

SASL 認証は、サーバー間通信（ZooKeeper インスタンス間の）とクライアント間の通信（Kafka と ZooKeeper の間）とクライアント間の通信（Kafka と ZooKeeper の間）に対して個別に設定されます。サーバー間の認証は、複数のノードを持つ ZooKeeper クラスターにのみ関連します。

サーバー間の認証

サーバー間の認証の場合、JAAS 設定ファイルには以下の 2 つの部分が含まれます。

- サーバー設定
- クライアント設定

DIGEST-MD5 SASL メカニズムを使用する場合、**QuorumServer** コンテキストを使用して認証サーバーを設定します。暗号化されていない形式でパスワードと接続できるようにするには、すべてのユー

ザー名が含まれている必要があります。2 つ目のコンテキスト **QuorumLearner** は、ZooKeeper に組み込まれているクライアントに設定する必要があります。また、パスワードを暗号化されていない形式で含んでいます。DIGEST-MD5 メカニズムの JAAS 設定ファイルの例を以下に示します。

```
QuorumServer {
      org.apache.zookeeper.server.auth.DigestLoginModule required
      user_zookeeper="123456";
};

QuorumLearner {
      org.apache.zookeeper.server.auth.DigestLoginModule required
      username="zookeeper"
      password="123456";
};
```

JAAS 設定ファイルの他に、以下のオプションを指定して、通常の ZooKeeper 設定ファイルでサーバー間の認証を有効にする必要があります。

```
quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20
```

KAFKA_OPTS 環境変数を使用して、JAAS 設定ファイルを Java プロパティーとして ZooKeeper サーバーに渡します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

サーバー間の認証に関する詳細は、「 ZooKeeper wiki 」を参照してください。

クライアント間の認証

クライアント間の認証は、サーバー間の認証と同じ JAAS ファイルで設定されます。ただし、サーバー間の認証とは異なり、サーバー設定のみが含まれます。設定のクライアント部分は、クライアントで実行する必要があります。認証を使用して ZooKeeper に接続するように Kafka ブローカーを設定する方法は、「 Kafka のインストール 」を参照してください。

Server コンテキストを JAAS 設定ファイルに追加し、クライアント間の認証を設定します。DIGEST-MD5 メカニズムでは、すべてのユーザー名とパスワードを設定します。

```
Server {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   user_super="123456"
   user_kafka="123456"
   user_someoneelse="123456";
};
```

JAAS コンテキストの設定後、以下の行を追加して ZooKeeper 設定ファイルで client-to-server 認証を有効にします。

```
requireClientAuthScheme=sasl
```

```
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

ZooKeeper クラスターの一部であるすべてのサーバーに対して **authProvider.<ID>** プロパティーを追加する必要があります。

KAFKA_OPTS 環境変数を使用して、JAAS 設定ファイルを Java プロパティーとして ZooKeeper サーバーに渡します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

Kafka ブローカーでの ZooKeeper 認証の設定に関する詳細は、「ZooKeeper の認証」 を参照してください。

3.4.2. DIGEST-MD5 を使用したサーバー間の認証の有効化

この手順では、ZooKeeper クラスターのノード間で SASL DIGEST-MD5 メカニズムを使用して認証を有効にする方法を説明します。

前提条件

- AMQ Streams がホストにインストールされていること。
- ZooKeeper クラスターは複数のノードで 設定され ます。

SASL DIGEST-MD5 認証の有効化

1. すべての ZooKeeper ノードで **/opt/kafka/config/zookeeper-jaas.conf** JAAS 設定ファイルを作成または編集し、以下のコンテキストを追加します。

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_<Username>="<Password>";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="<Username>"
    password="<Password>";
};
```

ユーザー名とパスワードは JAAS コンテキストの両方で同じである必要があります。以下に例を示します。

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_zookeeper="123456";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
```

```
        username="zookeeper"
        password="123456";
};
```

2. すべての ZooKeeper ノードで **/opt/kafka/config/zookeeper.properties** ZooKeeper 設定ファイルを編集し、以下のオプションを設定します。

```
quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20
```

3. すべての ZooKeeper ノードを 1 つずつ再起動します。JAAS 設定を ZooKeeper に渡すには、**KAFKA_OPTS** 環境変数を使用します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- ZooKeeper クラスターの実行に関する詳細は、「マルチノードの ZooKeeper クラスターの実行」 を参照してください。

3.4.3. DIGEST-MD5 を使用したクライアント間の認証の有効化

この手順では、ZooKeeper クライアントと ZooKeeper の間で SASL DIGEST-MD5 メカニズムを使用して認証を有効にする方法を説明します。

前提条件

- AMQ Streams がホストにインストールされていること。
- ZooKeeper クラスターが 設定され、実行されている。

SASL DIGEST-MD5 認証の有効化

1. すべての ZooKeeper ノードで **/opt/kafka/config/zookeeper-jaas.conf** JAAS 設定ファイルを作成または編集し、以下のコンテキストを追加します。

```
Server {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   user_super="<SuperUserPassword>"
   user<Username1>_="<Password1>" user<USername2>_="<Password2>";
};
```

super には、管理者の priviledges が自動的に付与します。ファイルには複数のユーザーを含めることができますが、Kafka ブローカーに必要な追加ユーザーのみが 1 つだけです。Kafka ユーザーに推奨される名前は **kafka** です。

以下の例は、クライアント/サーバー認証の **Server** コンテキストを示しています。

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="123456"
    user_kafka="123456";
};
```

2. すべての ZooKeeper ノードで **/opt/kafka/config/zookeeper.properties** ZooKeeper 設定ファイルを編集し、以下のオプションを設定します。

```
requireClientAuthScheme=sasl
authProvider.<IdOfBroker1>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider

authProvider.<IdOfBroker2>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider

authProvider.<IdOfBroker3>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

authProvider.<ID> プロパティーは、ZooKeeper クラスターの一部であるすべてのノードに対して追加する必要があります。3 ノードの ZooKeeper クラスターの設定例は以下のようになります。

```
requireClientAuthScheme=sasl
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

3. すべての ZooKeeper ノードを 1 つずつ再起動します。JAAS 設定を ZooKeeper に渡すには、**KAFKA_OPTS** 環境変数を使用します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-
jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon
/opt/kafka/config/zookeeper.properties
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- ZooKeeper クラスターの実行に関する詳細は、「マルチノードの ZooKeeper クラスターの実行」 を参照してください。

3.5. 承認

ZooKeeper はアクセス制御リスト(ACL)をサポートし、そこに格納されているデータを保護します。Kafka ブローカーは、他の ZooKeeper ユーザーが変更できないように、作成するすべての ZooKeeper レコードの ACL 権限を自動的に設定できます。

Kafka ブローカーでの ZooKeeper ACL の有効化に関する詳細は、「ZooKeeper の承認」 を参照してください。

3.6. TLS

ZooKeeper は暗号化または認証に対して TLS をサポートします。

3.7. 追加の設定オプション

ユースケースに応じて、以下の追加の ZooKeeper 設定オプションを設定できます。

maxClientCnxns

ZooKeeper クラスターの単一のメンバーへの同時クライアント接続の最大数。

autopurge.snapRetainCount

保持される ZooKeeper のインメモリーデータベースのスナップショットの数。デフォルト値は **3** です。

autopurge.purgeInterval

スナップショットを消去する時間間隔。デフォルト値は **0** で、このオプションは無効になります。

使用できる設定オプションはすべて ZooKeeper のドキュメントを参照してください。

3.8. ログ

ZooKeeper はログインフラストラクチャーとして **log4j** を使用します。ロギング設定は、**/opt/kafka/config/** ディレクトリーまたはクラスパスのどちらに配置する必要がある **log4j.properties** 設定ファイルからデフォルトで読み取られます。設定ファイルの場所と名前は、**KAFKA_LOG4J_OPTS** 環境変数を使用して ZooKeeper に渡すことのできる Java プロパティー **log4j.configuration** を使用して変更できます。

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.properties";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

Log4j の設定に関する詳細は、Log4j ドキュメント を参照してください。

第4章 KAFKA の設定

Kafka はプロパティーファイルを使用して静的設定を保存します。設定ファイルの推奨場所は **/opt/kafka/config/server.properties** です。設定ファイルは、**kafka** ユーザーが読み取り可能でなければなりません。

AMQ Streams には、製品のさまざまな基本機能や高度な機能を強調表示する設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/server.properties** にあります。

本章では、最も重要な設定オプションについて説明します。サポートされる Kafka ブローカー設定オプションの完全リストは、付録A *ブローカー設定パラメーター* を参照してください。

4.1. ZOOKEEPER

Kafka ブローカーは、クラスターの一部の設定も保存し、クラスターを調整する必要があります（たとえば、パーティションのリーダーであるノードを決定）。ZooKeeper クラスターの接続の詳細は設定ファイルに保存されます。フィールド **zookeeper.connect** には、zookeeper クラスターのメンバーのホスト名およびポートのコンマ区切りリストが含まれます。

以下に例を示します。

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181
```

Kafka はこれらのアドレスを使用して ZooKeeper クラスターに接続します。この設定では、すべての Kafka **znodes** が ZooKeeper データベースのルートに直接作成されます。そのため、ZooKeeper クラスターは単一の Kafka クラスターにのみ使用できます。単一の ZooKeeper クラスターを使用するように複数の Kafka クラスターを設定するには、Kafka 設定ファイルの ZooKeeper 接続文字列の最後にベース（プレフィックス）パスを指定します。

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181/my-cluster-1
```

4.2. リスナー

Kafka ブローカーは、複数のリスナーを使用するように設定できます。各リスナーは異なるポートまたはネットワークインターフェースをリッスンするために使用され、異なる設定を指定できます。リスナーは、設定ファイルの **listeners** プロパティーで設定されます。**listeners** プロパティーには、各リスナーが **<listenerName>://<hostname>:_<port>_** として設定されたリスナーのリストが含まれます。ホスト名の値が空の場合、Kafka は **java.net.InetAddress.getCanonicalHostName()** をホスト名として使用します。以下の例は、複数のリスナーの設定方法を示しています。

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
```

Kafka クライアントが Kafka クラスターに接続する場合は、最初に **ブートストラップサーバー** に接続します。**ブートストラップサーバー** はクラスターノードの 1 つです。クライアントには、クラスターの一部である他のすべてのブローカーの一覧を提供し、クライアントは個別に接続します。デフォルトでは、**ブートストラップサーバー** は **listeners** フィールドに基づくノードのリストをクライアントに提供します。

アドバタイズされたリスナー

listeners プロパティーに指定されたものとは異なるアドレスのセットをクライアントに付与することが

できます。プロキシーなどの追加のネットワークインフラストラクチャーがクライアントとブローカー間、または IP アドレスの代わりに外部 DNS 名が使用される場合に役立ちます。このブローカーは、advertised.listeners 設定プロパティーでリスナーのアドバタイズされたアドレスを定義できます。このプロパティーは listeners プロパティーと同じフォーマットになります。以下の例は、アドバタイズされたリスナーの設定方法を示しています。

```
listeners=INT1://:9092,INT2://:9093
advertised.listeners=INT1://my-broker-1.my-domain.com:1234,INT2://my-broker-1.my-domain.com:1234:9093
```

注記

リスナーの名前は、**listeners** プロパティーのリスナー名と一致する必要があります。

ブローカー間のリスナー

クラスターにトピックがレプリケートされたら、このようなトピックを担当するブローカーは、これらのトピックでメッセージをレプリケートするために相互に通信する必要があります。複数のリスナーが設定される場合、設定フィールド **inter.broker.listener.name** を使用してブローカー間のレプリケーションに使用されるリスナーの名前を指定します。以下に例を示します。

```
inter.broker.listener.name=REPLICATION
```

4.3. ログのコミット

Apache Kafka は、プロデューサーから受信するすべてのレコードをコミットログに保存します。コミットログには、Kafka が配信する必要のあるレコード形式で実際のデータが含まれます。これらは、ブローカーが実行する内容を記録するアプリケーションのログファイルではありません。

ログディレクトリー

log.dirs プロパティーファイルを使用してログディレクトリーを設定し、1つまたは複数のログディレクトリーにコミットログを保存できます。インストール時に作成した **/var/lib/kafka** ディレクトリーに設定する必要があります。

```
log.dirs=/var/lib/kafka
```

パフォーマンス上の理由から、log.dir を複数のディレクトリーに設定し、各ディレクトリーを異なる物理デバイスに配置して、ディスク I/O のパフォーマンスを向上できます。以下に例を示します。

```
log.dirs=/var/lib/kafka1,/var/lib/kafka2,/var/lib/kafka3
```

4.4. BROKER ID

ブローカー ID は、クラスター内の各ブローカーの一意の識別子です。0 以上の整数をブローカー ID に割り当てることができます。ブローカー ID は、再起動またはクラッシュ後にブローカーを識別するために使用されます。そのため、ID が安定し、時間の経過とともに変更しないことが重要です。ブローカー ID はブローカープロパティーファイルで設定されます。

```
broker.id=1
```

4.5. マルチノードの KAFKA クラスターの実行

この手順では、Kafka をマルチノードクラスターとして設定および実行する方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。
- ZooKeeper クラスターが 設定され、実行されている。

クラスターの実行

AMQ Streams クラスターの Kafka ブローカーごとに以下を行います。

1. 以下のように **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
   - 最初のブローカーの **broker.id** フィールドを **0**、第 2 ブローカーの **1** などに設定します。
   - **zookeeper.connect** オプションで ZooKeeper に接続する詳細を設定します。
   - Kafka リスナーを設定します。
   - コミットログが **logs.dir** ディレクトリーに格納されるディレクトリーを設定します。

   Kafka ブローカーの設定例は次のとおりです。

   ```
   broker.id=0
   zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181
   listeners=REPLICATION://:9091,PLAINTEXT://:9092
   inter.broker.listener.name=REPLICATION
   log.dirs=/var/lib/kafka
   ```

   各 Kafka ブローカーが同じハードウェアで実行されている一般的なインストールでは、**broker.id** 設定プロパティーのみがブローカー設定によって異なります。
2. デフォルトの設定ファイルを使用して Kafka ブローカーを起動します。

   ```
   su - kafka
   /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
   ```
3. Kafka ブローカーが稼働していることを確認します。

   ```
   jcmd | grep Kafka
   ```

ブローカーの検証

クラスターのすべてのノードが稼働したら、**ncat** ユーティリティーを使用して **dump** コマンドを ZooKeeper ノードの 1 つに送信し、すべてのノードが Kafka クラスターのメンバーであることを確認します。このコマンドは、ZooKeeper に登録されているすべての Kafka ブローカーを出力します。

1. ncat stat を使用してノードのステータスを確認します。

   ```
   echo dump | ncat zoo1.my-domain.com 2181
   ```

出力には、先ほど設定および起動したすべての Kafka ブローカーが含まれている必要があります。

3 つのノードで構成される Kafka クラスターの **ncat** コマンドからの出力例：

```
SessionTracker dump:
org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
ephemeral nodes dump:
Sessions with Ephemerals (3):
0x20000015dd00000:
        /brokers/ids/1
0x10000015dc70000:
        /controller
        /brokers/ids/0
0x10000015dc70001:
        /brokers/ids/2
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- ZooKeeper クラスターの実行に関する詳細は、「マルチノードの ZooKeeper クラスターの実行」 を参照してください。
- サポートされる Kafka ブローカー設定オプションの完全リストは、付録A *ブローカー設定パラメーター* を参照してください。

4.6. ZOOKEEPER の認証

デフォルトでは、ZooKeeper と Kafka 間の接続は認証されません。ただし、Kafka および ZooKeeper は、SASL(Simple Authentication and Security Layer)を使用して認証を設定するために使用できる Java Authentication and Authorization Service(JAAS)をサポートします。ZooKeeper は、ローカルで保存された認証情報を使用した DIGEST-MD5 SASL メカニズムを使用した認証をサポートします。

4.6.1. JAAS 設定

ZooKeeper 接続の SASL 認証は JAAS 設定ファイルで設定する必要があります。デフォルトでは、Kafka は ZooKeeper に接続するために **Client** という名前の JAAS コンテキストを使用します。 **Client** コンテキストは **/opt/kafka/config/jass.conf** ファイルで設定する必要があります。以下の例のように、コンテキストは **PLAIN** SASL 認証を有効にする必要があります。

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="kafka"
    password="123456";
};
```

4.6.2. ZooKeeper 認証の有効化

この手順では、ZooKeeper に接続する際に SASL DIGEST-MD5 メカニズムを使用して認証を有効にする方法を説明します。

前提条件

- ZooKeeper でクライアント間の認証が 有効 である。

SASL DIGEST-MD5 認証の有効化

1. すべての Kafka ブローカーノードで **/opt/kafka/config/jaas.conf** JAAS 設定ファイルを作成または編集し、以下のコンテキストを追加します。

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="<Username>"
    password="<Password>";
};
```

ユーザー名とパスワードは ZooKeeper で設定されるものと同じである必要があります。

以下の例は、**Client** コンテキストを示しています。

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="kafka"
    password="123456";
};
```

2. すべての Kafka ブローカーノードを 1 つずつ再起動します。JAAS 設定を Kafka ブローカーに渡すには、**KAFKA_OPTS** 環境変数を使用します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

関連情報

- ZooKeeper でのクライアント/サーバー認証の設定に関する詳細は、「認証」 を参照してください。

4.7. 承認

Kafka ブローカーの承認は、オーソライザープラグインを使用して実装されます。

ここでは、Kafka で提供される **AclAuthorizer** プラグインを使用する方法を説明します。

または、独自の承認プラグインを使用することもできます。たとえば、OAuth 2.0 トークンベースの認証 を使用している場合、OAuth 2.0 承認 を使用できます。

4.7.1. シンプル ACL オーソライザー

AclAuthorizer を含むオーソライザープラグインは、**authorizer.class.name** プロパティーを使用して有効になります。

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
```

選択したオーソライザーには完全修飾名が必要です。**AclAuthorizer** では、完全修飾名は **kafka.security.auth.SimpleAclAuthorizer** です。

4.7.1.1. ACL ルール

AclAuthorizer は、ACL ルールを使用して Kafka ブローカーへのアクセスを管理します。

ACL ルールは、以下の形式で定義されます。

principal **P** is allowed/ denied operation **O** on Kafka resource **R** from host **H**

たとえば、ユーザーが以下を行うようにルールを設定できます。

John は、ホスト 127.0.0.1からトピックの **コメント** を **表示** できます。

host は、John が接続しているマシンの IP アドレスです。

多くの場合、ユーザーはプロデューサーまたはコンシューマーアプリケーションです。

consumer01 は、ホスト **127.0.0.1 から** コンシューマーグループアカウントに **書き込む** ことができる。

ACL ルールが存在しない場合

指定のリソースに ACL ルールが存在しない場合は、すべてのアクションが拒否されます。この動作は、Kafka 設定ファイルの **allow.everyone.if.no.acl.found** プロパティーを **true** に設定すると変更できます。**/opt/kafka/config/server.properties**

4.7.1.2. principals

プリンシパル はユーザーのアイデンティティーを表します。ID の形式は、クライアントが Kafka に接続するための認証メカニズムによって異なります。

- **User:ANONYMOUS** 認証なしで接続する場合。
- **User:<username>** PLAIN や SCRAM などの簡単な認証メカニズムを使用して接続する場合。例： **User:admin** または **User:user1**
- **User:<DistinguishedName>** TLS クライアント認証を使用して接続する場合。たとえば、**User:CN=user1,O=MyCompany,L=Prague,C=CZ** のようになります。
- **User:<Kerberos username>** Kerberos を使用して接続する場合

distinguishedName **は** クライアント証明書からの識別名です。

Kerberos ユーザー名 は、Kerberos プリンシパルの主な部分で、Kerberos を使用した接続時にデフォルトで使用されます。**sasl.kerberos.principal.to.local.rules** プロパティーを使用すると、Kerberos プリンシパルから Kafka プリンシパルを構築する方法を設定できます。

4.7.1.3. ユーザーの認証

承認を使用するには、クライアントが認証を有効にし、使用する必要があります。それ以外の場合は、すべての接続にプリンシパル **User:ANONYMOUS** があります。

認証方法の詳細については、「 Encryption and authentication 」を参照してください。

4.7.1.4. スーパーユーザー

スーパーユーザーを使用すると、ACL ルールに関係なくすべてのアクションを実行することができます。

スーパーユーザーは、**super.users** プロパティーを使用して Kafka 設定ファイルで定義されます。

以下に例を示します。

```
super.users=User:admin,User:operator
```

4.7.1.5. レプリカブローカーの認証

承認を有効にすると、すべてのリスナーおよびすべての接続に適用されます。これには、ブローカー間のデータの複製に使用されるブローカー間接続が含まれます。そのため、承認を有効にする場合は、ブローカー間の接続に認証を使用し、ブローカーが十分な権限を付与していることを確認します。たとえば、ブローカー間の認証で **kafka-broker** ユーザーが使用される場合、スーパーユーザー設定にユーザー名 **super.users=User:kafka-broker** が含まれる必要があります。

4.7.1.6. サポートされるリソース

Kafka ACL は、以下のタイプのリソースに適用できます。

- トピック
- コンシューマーグループ
- クラスター
- TransactionId
- DelegationToken

4.7.1.7. サポートされる演算

AclAuthorizer リソースの操作を承認します。

以下の表の **X** を持つフィールドは、各リソースでサポートされる操作をマークします。

表4.1 リソースでサポートされる操作

	トピック	コンシューマーグループ	クラスター
Read	X	X	
Write	X		
Create			X
Delete	X		
Alter	X		
Describe	X	X	X

	トピック	コンシューマーグループ	クラスター
ClusterAction			X
すべて	X	X	X

4.7.1.8. ACL 管理オプション

ACL ルールは、Kafka ディストリビューションパッケージの一部として提供される **bin/kafka-acls.sh** ユーティリティーを使用して管理されます。

kafka-acls.sh パラメーターオプションを使用して ACL ルールを追加、一覧表示、および削除し、その他の機能を実行します。

パラメーターには、**--add** のように 2 倍の規則が必要です。

オプション	型	説明	デフォルト
add	動作	ACL ルールを追加します。	
remove	動作	ACL ルールを削除します。	
list	動作	ACL ルールの一覧を表示します。	
authorizer	動作	オーソライザーの完全修飾クラス名。	**kafka.security.auth.SimpleAclAuthorizer**
authorizer-properties	設定	初期化のためにオーソライザーに渡されるキー/値のペア。 **AclAuthorizer** の場合、値の例は **zookeeper.connect=zoo1.my-domain.com:2181** です。	
bootstrap-server	リソース	Kafka クラスターに接続するためのホスト/ポートのペア。	両方ではなく、このオプションまたは **authorizer** オプションを使用します。

オプション	型	説明	デフォルト
command-config	リソース	**bootstrap-server** パラメーターとともに使用される Admin Client に渡す設定プロパティーファイル。	
cluster	リソース	クラスターを ACL リソースとして指定します。	
topic	リソース	トピック名を ACL リソースとして指定します。 ワイルドカードとして使用されるアスタリスク (*)は **すべてのトピック** に変換されます。 1 つのコマンドで、複数の **--topic** オプションを指定することができます。	
group	リソース	コンシューマーグループ名を ACL リソースとして指定します。 1 つのコマンドで、複数の **--group** オプションを指定することができます。	
transactional-id	リソース	ACL リソースとしてトランザクション ID を指定します。 トランザクション配信は、プロデューサーによって複数のパーティションに送信されたすべてのメッセージを正常に配信したり、何も指定しないことを意味します。 ワイルドカードとして使用されるアスタリスク (*)は **すべての ID に変換されます。**	

オプション	型	説明	デフォルト
delegation-token	リソース	委譲トークンを ACL リソースとして指定します。 ワイルドカードとして使用されるアスタリスク (*)は **すべてのトークン** に変換されます。	
resource-pattern-type	設定	**add** パラメーターまたは **list** または **remove** パラメーターのリソースパターンのフィルター値を指定します。 **literal** または **prefixed** をリソース名のリソースパターンタイプとして使用します。 **any** または **match** をリソースパターンのフィルターの値、または特定のパターンタイプフィルターとして使用します。	**literal**
allow-principal	principal	許可 ACL ルールに追加されたプリンシパル。 1 つのコマンドで、複数の **--allow-principal** オプションを指定することができます。	
deny-principal	principal	プリンシパルが拒否 ACL ルールに追加される。 1 つのコマンドで、複数の **--deny-principal** オプションを指定することができます。	

オプション	型	説明	デフォルト
principal	principal	プリンシパルの ACL 一覧を返す際に **list** パラメーターで使用されるプリンシパル名。 1 つのコマンドで、複数の **--principal** オプションを指定することができます。	
allow-host	ホスト	**--allow-principal** に一覧表示されているプリンシパルへのアクセスを許可する IP アドレス。 ホスト名または CIDR 範囲はサポートされません。	**--allow-principal** を指定すると、デフォルトは * で「すべてのホスト」を意味します。
deny-host	ホスト	**--deny-principal** に一覧表示されているプリンシパルへのアクセスを拒否する IP アドレス。 ホスト名または CIDR 範囲はサポートされません。	**--deny-principal** を指定すると、デフォルトは * で「すべてのホスト」を意味します。
operation	演算子	操作を許可または拒否します。 1 つのコマンドで、複数の Multiple **--operation** オプションを 1 つのコマンドで指定できます。	すべて
producer	ショートカット	メッセージプロデューサーが必要とするすべての操作を許可または拒否するショートカット（トピック上の WRITE および DESCRIBE、クラスターの CREATE）。	
consumer	ショートカット	メッセージコンシューマーが必要とするすべての操作を許可または拒否するショートカット（トピックの READ および DESCRIBE、コンシューマーグループの READ）。	

オプション	型	説明	デフォルト
idempotent	ショートカット	**--producer** パラメーターと使用する場合にべき等を有効にするショートカットで、メッセージはパーティションに 1 度だけ配信されます。 Idepmotence は、プロデューサーが特定のトランザクション ID に基づいてメッセージを送信できると、自動的に有効になります。	
force	ショートカット	すべてのクエリーを受け入れ、プロンプトを表示しないショートカット。	

4.7.2. 承認の有効化

この手順では、Kafka ブローカーで承認の **AclAuthorizer** プラグインを有効にする方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストにインストール されます。

手順

1. **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集して **AclAuthorizer** を使用します。

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
```

2. (re)Kafka ブローカーを起動します。

関連情報

- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- Kafka クラスターの実行に関する詳細は、「マルチノードの Kafka クラスターの実行」 を参照してください。

4.7.3. ACL ルールの追加

AclAuthorizer アクセス制御リスト(ACL)を使用します。これは、ユーザーがどのユーザーが実行できるかを記述する一連のルールを定義します。

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインの使用時に ACL ルールを追加する方法を説明します。

ルールは **kafka-acls.sh** ユーティリティーを使用して追加され、ZooKeeper に保存されます。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストにインストール されます。
- 承認は Kafka ブローカーで 有効 になります。

手順

1. **--add** オプションを指定して **kafka-acls.sh** を実行します。
例:

   - **MyConsumerGroup** コンシューマーグループを使用して **user1** および **user2** アクセスが **myTopic** から読み取りできるようにします。

     ```
     bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181
     --add --operation Read --topic myTopic --allow-principal User:user1 --allow-principal
     User:user2

     bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181
     --add --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal
     User:user2

     bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181
     --add --operation Read --operation Describe --group MyConsumerGroup --allow-
     principal User:user1 --allow-principal User:user2
     ```

   - IP アドレス **127.0.0.1** から **myTopic** を読み取る **user1** アクセスを拒否します。

     ```
     bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181
     --add --operation Describe --operation Read --topic myTopic --group MyConsumerGroup
     --deny-principal User:user1 --deny-host 127.0.0.1
     ```

   - **MyConsumerGroup** を使用して **myTopic** のコンシューマーとして **user1** を追加します。

     ```
     bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181
     --add --consumer --topic myTopic --group MyConsumerGroup --allow-principal
     User:user1
     ```

関連情報

- すべての **kafka-acls.sh** オプションの一覧は、「シンプル ACL オーソライザー」 を参照してください。

4.7.4. ACL ルールの一覧表示

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインの使用時に既存の ACL ルールの一覧を表示する方法を説明します。

ルールは、**kafka-acls.sh** ユーティリティーを使用して一覧表示されます。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストにインストール されます。
- Kafka ブローカーで承認が 有効になっ ている必要があります。
- ACL が 追加 されました。

手順

- **--list** オプションを指定して **kafka-acls.sh** を実行します。
  以下に例を示します。

```
$ bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --list --topic myTopic

Current ACLs for resource `Topic:myTopic`:

User:user1 has Allow permission for operations: Read from hosts: *
User:user2 has Allow permission for operations: Read from hosts: *
User:user2 has Deny permission for operations: Read from hosts: 127.0.0.1
User:user1 has Allow permission for operations: Describe from hosts: *
User:user2 has Allow permission for operations: Describe from hosts: *
User:user2 has Deny permission for operations: Describe from hosts: 127.0.0.1
```

関連情報

- すべての **kafka-acls.sh** オプションの一覧は、「シンプル ACL オーソライザー」 を参照してください。

4.7.5. ACL ルールの削除

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインの使用時に ACL ルールを削除する方法を説明します。

ルールは **kafka-acls.sh** ユーティリティーを使用して削除されます。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストにインストール されます。
- 承認は Kafka ブローカーで 有効 になります。
- ACL が 追加 されました。

手順

- **--remove** オプションを指定して **kafka-acls.sh** を実行します。
  例:
- **MyConsumerGroup** コンシューマーグループを使用して、**myTopic** から読み込む **user1** および **user2** アクセスを許可する ACL を削除します。

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --remove --operation Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2

bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --remove --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2

bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --remove --operation Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal User:user2
```

- **MyConsumerGroup** を使用して **myTopic** のコンシューマーとして **user1** を追加して ACL を削除します。

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --remove --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

- **user1** アクセスを拒否し、IP アドレスホスト **127.0.0.1** から **myTopic** を読み取るように拒否する ACL を削除します。

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --remove --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

関連情報

- すべての **kafka-acls.sh** オプションの一覧は、「シンプル ACL オーソライザー」 を参照してください。
- 承認の有効化に関する詳細は、「承認の有効化」 を参照してください。

4.8. ZOOKEEPER の承認

Kafka と ZooKeeper 間で認証が有効になっている場合、ZooKeeper アクセス制御リスト(ACL)ルールを使用して、ZooKeeper に保存される Kafka のメタデータへのアクセスを自動的に制御できます。

4.8.1. ACL 設定

ZooKeeper ACL ルールの適用は、**config/server.properties** Kafka 設定ファイルの **zookeeper.set.acl** プロパティーによって制御されます。

このプロパティーはデフォルトでは無効になっており、**true** に設定して有効にします。

```
zookeeper.set.acl=true
```

ACL ルールが有効な場合、**znode** が ZooKeeper で作成されると、変更または削除が可能な Kafka ユーザーのみが ZooKeeper で作成されます。他のすべてのユーザーは読み取り専用アクセスを持ちます。

Kafka は、新規に作成された ZooKeeper **znodes** に対してのみ ACL ルールを設定します。クラスターの初回の起動後に ACL が有効になると、**zookeeper-security-migration.sh** ツールはすべての既存 **znodes** で ACL を設定することができます。

ZooKeeper でのデータの機密性

ZooKeeper に保存されるデータには以下が含まれます。

- トピック名およびその設定
- SASL SCRAM 認証が使用される場合に、ソルトおよびハッシュ化されたユーザー認証情報。

しかし、ZooKeeper は Kafka を使用して送受信されたレコードを保存しません。ZooKeeper に保存されるデータは特定されないことが想定されます。

データが機密であると思われる場合（トピック名には顧客 ID が含まれるなど）、保護に使用できる唯一のオプションはネットワークレベルで ZooKeeper を分離するだけで、Kafka ブローカーへのアクセスを許可します。

4.8.2. 新しい Kafka クラスターの ZooKeeper ACL の有効化

この手順では、新しい Kafka クラスターの Kafka 設定で ZooKeeper ACL を有効にする方法を説明します。この手順は、Kafka クラスターの最初の起動前にのみ使用してください。既に実行しているクラスターで ZooKeeper ACL を有効にする場合は、「既存の Kafka クラスターでの ZooKeeper ACL の有効化」 を参照してください。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。
- ZooKeeper クラスターが 設定され、実行されている。
- ZooKeeper でクライアント間の認証が 有効 になります。
- ZooKeeper 認証は Kafka ブローカーで 有効 になります。
- Kafka ブローカーが起動していません。

手順

1. **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集し、すべてのクラスターノードの **zookeeper.set.acl** フィールドを **true** に設定します。

```
zookeeper.set.acl=true
```

2. Kafka ブローカーを起動します。

4.8.3. 既存の Kafka クラスターでの ZooKeeper ACL の有効化

この手順では、稼働中の Kafka クラスターの Kafka 設定で ZooKeeper ACL を有効にする方法を説明します。**zookeeper-security-migration.sh** ツールを使用して、既存の **znodes** すべてに ZooKeeper ACL を設定します。**zookeeper-security-migration.sh** は AMQ Streams の一部として使用でき、 **bin** ディレクトリーにあります。

前提条件

- Kafka クラスターが 設定され、稼働している。

ZooKeeper ACL の有効化

1. **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集し、すべてのクラスターノードの **zookeeper.set.acl** フィールドを **true** に設定します。

```
zookeeper.set.acl=true
```

2. すべての Kafka ブローカーを 1 つずつ再起動します。

3. **zookeeper-security-migration.sh** ツールを使用して、既存の ZooKeeper **znodes** に ACL を設定します。

```
su - kafka
cd /opt/kafka
KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-security-migration.sh --zookeeper.acl=secure --zookeeper.connect=<ZooKeeperURL>
exit
```

以下に例を示します。

```
su - kafka
cd /opt/kafka
KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-security-migration.sh --zookeeper.acl=secure --zookeeper.connect=zoo1.my-domain.com:2181
exit
```

4.9. 暗号化および認証

AMQ Streams は、リスナー設定の一部として設定された暗号化および認証をサポートします。

4.9.1. リスナーの設定

Kafka ブローカーの暗号化および認証はリスナーごとに設定されます。Kafka リスナーの設定に関する詳細は、「リスナー」 を参照してください。

Kafka ブローカーの各リスナーは、独自のセキュリティープロトコルで設定されます。設定プロパティー **listener.security.protocol.map** は、セキュリティープロトコルを使用するリスナーを定義します。各リスナー名をセキュリティープロトコルにマッピングします。サポートされるセキュリティープロトコルは以下のとおりです。

PLAINTEXT
暗号化または認証のないリスナー。

SSL
TLS による暗号化を使用し、任意で TLS クライアント証明書を使用した認証。

SASL_PLAINTEXT
暗号化がなく、SASL ベースの認証を使用するリスナー。

SASL_SSL
TLS ベースの暗号化および SASL ベースの認証を使用するリスナー。

以下の **listeners** 設定を指定します。

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
```

listener.security.protocol.map は以下の例のようになります。

```
listener.security.protocol.map=INT1:SASL_PLAINTEXT,INT2:SASL_SSL,REPLICATION:SSL
```

これにより、リスナー **INT1** が SASL 認証との暗号化されていない接続を使用するように、リスナー **INT2** は SASL 認証との暗号化された接続を使用するように、および **REPLICATION** インターフェースを TLS 暗号化（TLS クライアント認証など）を使用するように設定します。同じセキュリティープロトコルを複数回使用できます。以下の例は、有効な設定です。

```
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL
```

このような設定では、全インターフェースに TLS 暗号化と TLS 認証を使用します。以下の章では、TLS および SASL の設定方法について説明します。

4.9.2. TLS 暗号化

Kafka は、Kafka クライアントとの通信を暗号化するために TLS をサポートします。

TLS 暗号化およびサーバーの認証を使用するには、秘密鍵と公開鍵を含むキーストアを指定する必要があります。通常、これは Java Keystore(JKS)形式のファイルを使用して行われます。このファイルへのパスは、**ssl.keystore.location** プロパティーで設定されます。**ssl.keystore.password** プロパティーを使用して、キーストアを保護するパスワードを設定する必要があります。以下に例を示します。

```
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456
```

秘密鍵を保護するために、追加のパスワードが使用されることがあります。このようなパスワードは、**ssl.key.password** プロパティーを使用して設定できます。

Kafka は認証局によって署名されたキーと自己署名証明書を使用できます。認証局が署名した鍵の使用は、常に優先される方法である必要があります。クライアントが接続している Kafka ブローカーの ID を検証できるようにするには、証明書に、コモンネーム(CN)または Subject Alternative Names(SAN)としてアドバタイズされたホスト名が常に含まれている必要があります。

異なるリスナーに異なる SSL 設定を使用できます。**ssl.** で始まるすべてのオプションの前に **listener.name.<NameOfTheListener>.** を付けることができます。ここで、リスナーの名前は小文字の場合に常に必要になります。これにより、その特定のリスナーのデフォルトの SSL 設定が上書きされます。以下の例は、異なるリスナーに異なる SSL 設定を使用する方法を表しています。

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL

# Default configuration - will be used for listeners INT1 and INT2
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456

# Different configuration for listener REPLICATION
listener.name.replication.ssl.keystore.location=/path/to/keystore/server-1.jks
listener.name.replication.ssl.keystore.password=123456
```

追加の TLS 設定オプション

上記の主な TLS 設定オプションに加え、Kafka は TLS 設定を調整するための多くのオプションをサポートしています。たとえば、TLS / SSL プロトコルまたは暗号スイートを有効または無効にするには、次のコマンドを実行します。

ssl.cipher.suites

有効な暗号スイートの一覧。各暗号スイートは、TLS 接続に使用される認証、暗号化、MAC、鍵交換アルゴリズムの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートが有効になります。

ssl.enabled.protocols

有効な TLS / SSL プロトコルの一覧。デフォルトは **TLSv1.2,TLSv1.1,TLSv1** です。

サポートされる Kafka ブローカー設定オプションの完全リストは、付録A *ブローカー設定パラメーター* を参照してください。

4.9.3. TLS 暗号化の有効化

この手順では、Kafka ブローカーで暗号化を有効にする方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。

手順

1. クラスターのすべての Kafka ブローカーの TLS 証明書を生成します。証明書には、Common Name または Subject Alternative Name にアドバタイズされ、ブートストラップアドレスが必要です。
2. 以下について、すべてのクラスターノードで **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
   - **listener.security.protocol.map** フィールドを変更し、TLS 暗号化を使用するリスナーの **SSL** プロトコルを指定します。
   - **ssl.keystore.location** オプションを、ブローカー証明書を使用して JKS キーストアへのパスに設定します。
   - キーストアを保護するために使用したパスワードに **ssl.keystore.password** オプションを設定します。
     以下に例を示します。

     ```
     listeners=UNENCRYPTED://:9092,ENCRYPTED://:9093,REPLICATION://:9094
     listener.security.protocol.map=UNENCRYPTED:PLAINTEXT,ENCRYPTED:SSL,REPLICATION:PLAINTEXT
     ssl.keystore.location=/path/to/keystore/server-1.jks
     ssl.keystore.password=123456
     ```
3. (re)Kafka ブローカーを起動します。

関連情報

- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。

- Kafka クラスターの実行に関する詳細は、「マルチノードの Kafka クラスターの実行」 を参照してください。
- クライアントで TLS 暗号化の設定に関する詳細は、以下を参照してください。
  - 付録D *プロデューサー設定パラメーター*
  - 付録C *コンシューマー設定パラメーター*

4.9.4. 認証

認証には、以下を使用できます。

- 暗号化された接続で X.509 証明書を基にした TLS クライアント認証
- サポートされる Kafka SASL(Simple Authentication and Security Layer)メカニズム
- OAuth 2.0 のトークンベースの認証

4.9.4.1. TLS クライアント認証

TLS クライアント認証は、TLS 暗号化を使用している接続でのみ使用できます。TLS クライアント認証を使用するには、パブリックキーを持つトラストストアをブローカーに提供できます。これらのキーは、ブローカーに接続するクライアントを認証するために使用できます。トラストストアは Java Keystore(JKS)形式で提供され、認証局の公開鍵が含まれている必要があります。トラストストアに含まれる認証局のいずれかによって署名された公開鍵および秘密鍵を持つすべてのクライアントは認証されます。トラストストアの場所は、フィールド **ssl.truststore.location** を使用して設定されます。トラストストアがパスワードで保護されている場合は、パスワードは **ssl.truststore.password** プロパティーに設定する必要があります。以下に例を示します。

```
ssl.truststore.location=/path/to/keystore/server-1.jks
ssl.truststore.password=123456
```

トラストストアを設定したら、**ssl.client.auth** プロパティーを使用して TLS クライアント認証を有効にする必要があります。このプロパティーは、3 つの異なる値のいずれかに設定できます。

none
: TLS クライアント認証は無効になっています。（デフォルト値）

requested
: TLS クライアント認証はオプションです。クライアントは TLS クライアント証明書を使用して認証するよう要求されますが、選択することはできません。

required
: クライアントが TLS クライアント証明書を使用して認証する必要があります。

クライアントが TLS クライアント認証を使用して認証される場合、認証されたプリンシパル名は認証されたクライアント証明書からの識別名になります。たとえば、識別名 **CN=someuser** を持つ証明書を持つユーザーは、以下のプリンシパル **CN=someuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown** で認証されます。TLS クライアント認証が使用されず、SASL が無効化されると、プリンシパル名は **ANONYMOUS** になります。

4.9.4.2. SASL 認証

SASL 認証は、Java Authentication and Authorization Service(JAAS)を使用して設定されます。JAAS

は、Kafka と ZooKeeper 間の接続の認証にも使用されます。JAAS は独自の設定ファイルを使用します。このファイルの推奨場所は **/opt/kafka/config/jaas.conf** です。このファイルは、**kafka** ユーザーが読み取り可能でなければなりません。Kafka の実行時に、Java システムプロパティー **java.security.auth.login.config** を使用してこのファイルの場所を指定します。このプロパティーは、ブローカーノードの起動時に Kafka に渡す必要があります。

```
KAFKA_OPTS="-Djava.security.auth.login.config=/path/to/my/jaas.config"; bin/kafka-server-start.sh
```

SASL 認証は、暗号化されていない接続と TLS 接続の両方でサポートされます。SASL はリスナーごとに個別に有効にできます。これを有効にするには、**listener.security.protocol.map** のセキュリティープロトコルは **SASL_PLAINTEXT** または **SASL_SSL** のどちらかでなければなりません。

Kafka の SASL 認証は複数のメカニズムをサポートします。

PLAIN

ユーザー名とパスワードに基づいて認証を実装します。ユーザー名とパスワードは Kafka 設定にローカルに保存されます。

SCRAM-SHA-256 および SCRAM-SHA-512

Salted Challenge Response Authentication Mechanism(SCRAM)を使用した認証を実装します。SCRAM クレデンシャルは ZooKeeper に一元的に保存されます。SCRAM は、ZooKeeper クラスターノードがプライベートネットワークで分離された状態で使用できます。

GSSAPI

Kerberos サーバーに対する認証を実装します。

警告

PLAIN メカニズムは、暗号化されていない形式でネットワーク経由でユーザー名とパスワードを送信します。したがって、TLS による暗号化と組み合わせて使用することはできません。

SASL メカニズムは JAAS 設定ファイルを使用して設定されます。Kafka は **KafkaServer** という名前の JAAS コンテキストを使用します。JAAS で設定後、Kafka 設定で SASL メカニズムを有効にする必要があります。これは、**sasl.enabled.mechanisms** プロパティーを使用して行います。このプロパティーには、有効なメカニズムのコンマ区切りリストが含まれます。

```
sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256,SCRAM-SHA-512
```

ブローカー間の通信に使用されるリスナーが SASL を使用している場合は、**sasl.mechanism.inter.broker.protocol** プロパティーを使用して使用する必要のある SASL メカニズムを指定する必要があります。以下に例を示します。

```
sasl.mechanism.inter.broker.protocol=PLAIN
```

ブローカーの通信に使用されるユーザー名とパスワードは、フィールド **username** および **password** を使用して **KafkaServer** JAAS コンテキストに指定する必要があります。

SASL PLAIN

PLAIN メカニズムを使用するには、接続が許可されるユーザー名とパスワードを JAAS コンテキストで直接指定されます。以下の例は、SASL PLAIN 認証に設定されたコンテキストを示しています。この例では、3 つの異なるユーザーを設定します。

- **admin**
- **user1**
- **user2**

```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   user_admin="123456"
   user_user1="123456"
   user_user2="123456";
};
```

ユーザーデータベースのある JAAS 設定ファイルは、すべての Kafka ブローカーで同期する必要があります。

ブローカー間の認証に SASL PLAIN が使用される場合、**username** および **password** プロパティーは JAAS コンテキストに追加する必要があります。

```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   username="admin"
   password="123456"
   user_admin="123456"
   user_user1="123456"
   user_user2="123456";
};
```

SASL SCRAM

Kafka の SCRAM 認証は **SCRAM-SHA-256** と **SCRAM-SHA-512** の 2 つのメカニズムで構成されます。これらのメカニズムは、使用されるハッシュアルゴリズムでのみ異なります。SHA-256 と強力な SHA-512。SCRAM 認証を有効にするには、JAAS 設定ファイルに以下の設定ファイルが含まれている必要があります。

```
KafkaServer {
   org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

Kafka 設定ファイルで SASL 認証を有効にする場合、両方の SCRAM メカニズムを一覧表示できます。ただし、ブローカー間通信には選択できるのは 1 つのみです。以下に例を示します。

```
sasl.enabled.mechanisms=SCRAM-SHA-256,SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
```

SCRAM メカニズムのユーザークレデンシャルは ZooKeeper に保存されます。**kafka-configs.sh** ツールは、これらを管理するために使用できます。たとえば、以下のコマンドを実行して、パスワード 123456 で user1 を追加します。

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'SCRAM-SHA-256=
[password=123456],SCRAM-SHA-512=[password=123456]' --entity-type users --entity-name user1
```

ユーザークレデンシャルを削除するには、以下を使用します。

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config 'SCRAM-SHA-512' --
entity-type users --entity-name user1
```

SASL GSSAPI

Kerberos を使用した認証に使用される SASL メカニズムは **GSSAPI** と呼ばれます。Kerberos SASL 認証を設定するには、以下の設定を JAAS 設定ファイルに追加します。

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};
```

Kerberos プリンシパルのドメイン名は常に大文字にする必要があります。

JAAS 設定の他に、Kafka 設定の **sasl.kerberos.service.name** プロパティーで Kerberos サービス名を指定する必要があります。

```
sasl.enabled.mechanisms=GSSAPI
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka
```

複数の SASL メカニズム

Kafka は複数の SASL メカニズムを同時に使用できます。異なる JAAS 設定はすべて同じコンテキストに追加できます。

```
KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    user_admin="123456"
    user_user1="123456"
    user_user2="123456";

    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";

    org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

複数のメカニズムを有効にすると、クライアントが使用するメカニズムを選択することができます。

4.9.5. TLS クライアント認証の有効化

この手順では、Kafka ブローカーで TLS クライアント認証を有効にする方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。
- TLS 暗号化が 有効になっている。

手順

1. ユーザー証明書の署名に使用する認証局の公開鍵が含まれる JKS トラストストアを準備します。
2. 以下について、すべてのクラスターノードで **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
   - ユーザー証明書の認証局が含まれる JKS トラストストアへのパスに **ssl.truststore.location** オプションを設定します。
   - トラストストアを保護するために使用したパスワードに **ssl.truststore.password** オプションを設定します。
   - **ssl.client.auth** オプションを **required** に設定します。
     以下に例を示します。

     ```
     ssl.truststore.location=/path/to/truststore.jks
     ssl.truststore.password=123456
     ssl.client.auth=required
     ```
3. (re)Kafka ブローカーを起動します。

関連情報

- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- Kafka クラスターの実行に関する詳細は、「マルチノードの Kafka クラスターの実行」 を参照してください。
- クライアントで TLS 暗号化の設定に関する詳細は、以下を参照してください。
  - 付録D *プロデューサー設定パラメーター*
  - 付録C *コンシューマー設定パラメーター*

4.9.6. SASL PLAIN 認証の有効化

この手順では、Kafka ブローカーで SASL PLAIN 認証を有効にする方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。

手順

1. **/opt/kafka/config/jaas.conf** JAAS 設定ファイルを編集または作成します。このファイルには、ユーザーとパスワードをすべて含める必要があります。このファイルはすべての Kafka ブローカーで同じであることを確認します。
以下に例を示します。

```
KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    user_admin="123456"
    user_user1="123456"
    user_user2="123456";
};
```

2. 以下について、すべてのクラスターノードで **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。

   - **listener.security.protocol.map** フィールドを変更し、SASL PLAIN 認証を使用するリスナーの **SASL_PLAINTEXT** または **SASL_SSL** プロトコルを指定します。

   - **sasl.enabled.mechanisms** オプションを **PLAIN** に設定します。
   以下に例を示します。

```
listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINTEXT,REPLICATION:PLAINTEXT
sasl.enabled.mechanisms=PLAIN
```

3. （）KAFKA_OPTS 環境変数を使用して Kafka ブローカーを起動し、JAAS 設定を Kafka ブローカーに渡します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

関連情報

- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- Kafka クラスターの実行に関する詳細は、「マルチノードの Kafka クラスターの実行」 を参照してください。
- クライアントで SASL PLAIN 認証の設定に関する詳細は、以下を参照してください。
  - 付録D *プロデューサー設定パラメーター*
  - 付録C *コンシューマー設定パラメーター*

4.9.7. SASL SCRAM 認証の有効化

この手順では、Kafka ブローカーで SASL SCRAM 認証を有効にする方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。

手順

1. **/opt/kafka/config/jaas.conf** JAAS 設定ファイルを編集または作成します。**KafkaServer** コンテキストの **ScramLoginModule** を有効にします。このファイルはすべての Kafka ブローカーで同じであることを確認します。
以下に例を示します。

```
KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

2. 以下について、すべてのクラスターノードで **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。

   - **listener.security.protocol.map** フィールドを変更し、SASL SCRAM 認証を使用するリスナーの **SASL_PLAINTEXT** または **SASL_SSL** プロトコルを指定します。
   - **sasl.enabled.mechanisms** オプションを **SCRAM-SHA-256** または **SCRAM-SHA-512** に設定します。
   以下に例を示します。

```
listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINTEXT,REPLICATION:PLAINTEXT
sasl.enabled.mechanisms=SCRAM-SHA-512
```

3. （）KAFKA_OPTS 環境変数を使用して Kafka ブローカーを起動し、JAAS 設定を Kafka ブローカーに渡します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

関連情報

- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- Kafka クラスターの実行に関する詳細は、「マルチノードの Kafka クラスターの実行」 を参照してください。
- SASL SCRAM ユーザーの追加に関する詳細は、「SASL SCRAM ユーザーの追加」 を参照してください。
- SASL SCRAM ユーザーの削除に関する詳細は、「SASL SCRAM ユーザーの削除」 を参照してください。
- クライアントで SASL SCRAM 認証の設定に関する詳細は、以下を参照してください。
  - 付録D *プロデューサー設定パラメーター*
  - 付録C *コンシューマー設定パラメーター*

4.9.8. SASL SCRAM ユーザーの追加

この手順では、SASL SCRAM を使用して認証用の新規ユーザーを追加する方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。
- SASL SCRAM 認証が 有効になっている。

手順

- **kafka-configs.sh** ツールを使用して、新しい SASL SCRAM ユーザーを追加します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --alter --add-config 'SCRAM-SHA-512=[password=<Password>]' --entity-type users --entity-name <Username>
```

以下に例を示します。

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'SCRAM-SHA-512=[password=123456]' --entity-type users --entity-name user1
```

関連情報

- クライアントで SASL SCRAM 認証の設定に関する詳細は、以下を参照してください。
  - 付録D *プロデューサー設定パラメーター*
  - 付録C *コンシューマー設定パラメーター*

4.9.9. SASL SCRAM ユーザーの削除

この手順では、SASL SCRAM 認証を使用する際にユーザーを削除する方法を説明します。

前提条件

- AMQ Streams は、Kafka ブローカーとして使用されるすべてのホストに インストール されます。
- SASL SCRAM 認証が 有効になっている。

手順

- **kafka-configs.sh** ツールを使用して SASL SCRAM ユーザーを削除します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name <Username>
```

以下に例を示します。

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name user1
```

関連情報

- クライアントで SASL SCRAM 認証の設定に関する詳細は、以下を参照してください。

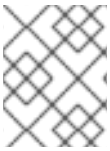
- 付録D *プロデューサー設定パラメーター*
- 付録C *コンシューマー設定パラメーター*

4.10. OAUTH 2.0 トークンベース認証の使用

AMQ Streams は、**OAUTHBEARER** および **PLAIN** メカニズムを使用して OAuth 2.0 認証の使用をサポートします。

OAuth 2.0 は、アプリケーション間で標準的なトークンベースの認証および承認を有効にし、中央の承認サーバーを使用してリソースに制限されたアクセス権限を付与するトークンを発行します。

Kafka ブローカーおよびクライアントの両方が OAuth 2.0 を使用するように設定する必要があります。OAuth 2.0 認証を設定した後に OAuth 2.0 承認 を設定できます。



注記

OAuth 2.0 認証は、使用する承認サーバーに関係なく ACL ベースの Kafka 承認 と併用できます。

OAuth 2.0 認証を使用すると、アプリケーションクライアントはアカウントのクレデンシャルを公開せずにアプリケーションサーバー (**リソースサーバー** と呼ばれる) のリソースにアクセスできます。

アプリケーションクライアントは、アクセストークンを認証の手段として渡します。アプリケーションサーバーはこれを使用して、付与するアクセス権限のレベルを決定することもできます。承認サーバーは、アクセスの付与とアクセスに関する問い合わせを処理します。

AMQ Streams のコンテキストでは以下が行われます。

- Kafka ブローカーは OAuth 2.0 リソースサーバーとして動作します。
- Kafka クライアントは OAuth 2.0 アプリケーションクライアントとして動作します。

Kafka クライアントは Kafka ブローカーに対して認証を行います。ブローカーおよびクライアントは、必要に応じて OAuth 2.0 承認サーバーと通信し、アクセストークンを取得または検証します。

AMQ Streams のデプロイメントでは、OAuth 2.0 インテグレーションは以下を提供します。

- Kafka ブローカーのサーバー側 OAuth 2.0 サポート。
- Kafka MirrorMaker、Kafka Connect、および Kafka Bridge のクライアント側 OAuth 2.0 サポート。

AMQ Streams on RHEL には、2 つの OAuth 2.0 ライブラリーが含まれています。

kafka-oauth-client

io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler という名前のカスタムログインコールバックハンドラークラスを提供します。**OAUTHBEARER** 認証メカニズムを処理するには、Apache Kafka が提供する **OAuthBearerLoginModule** でログインコールバックハンドラーを使用します。

kafka-oauth-common

kafka-oauth-client ライブラリーで必要な機能の一部を提供するヘルパーライブラリーです。

提供されるクライアントライブラリーには、**keycloak-core**、**jackson-databind**、**slf4j-api** などの追加のサードパーティーライブラリーの依存関係も含まれます。

Maven プロジェクトを使用してクライアントをパッケージ化し、すべての依存関係ライブラリーを含めることが推奨されます。依存関係ライブラリーは、今後のバージョンで変更される可能性があります。

OAuth コールバックハンドラー は Kafka Client Java ライブラリー用に提供されるため、Java クライアント用に独自のコールバックハンドラーを作成する必要はありません。アプリケーションクライアントはコールバックハンドラーを使用してアクセストークンを提供できます。Go などの他言語で書かれたクライアントは、カスタムコードを使用して承認サーバーに接続し、アクセストークンを取得する必要があります。

関連情報

- OAuth 2.0 のサイト

4.10.1. OAuth 2.0 認証メカニズム

AMQ Streams は、OAuth 2.0 認証で OAUTHBEARER および PLAIN メカニズムをサポートします。どちらのメカニズムも、Kafka クライアントが Kafka ブローカーで認証されたセッションを確立できるようにします。クライアント、承認サーバー、および Kafka ブローカー間の認証フローは、メカニズムごとに異なります。

可能な限り、OAUTHBEARER を使用するようにクライアントを設定することが推奨されます。OAUTHBEARER では、クライアントクレデンシャルは Kafka ブローカーと**共有されることがない**ため、PLAIN よりも高レベルのセキュリティーが提供されます。OAUTHBEARER をサポートしない Kafka クライアントの場合のみ、PLAIN の使用を検討してください。

必要であれば、同じ OAuth 認証リスナー設定で OAUTHBEARER と PLAIN を両方有効にすることができます。

OAUTHBEARER の概要

Kafka は OAUTHBEARER 認証メカニズムをサポートしますが、明示的に設定する必要があります。多くの Kafka クライアントツールは、プロトコルレベルで OAUTHBEARER の基本サポートを提供するライブラリーを使用します。

OAUTHBEARER を使用する場合、クライアントはクレデンシャルを交換するために Kafka ブローカーでセッションを開始します。ここで、クレデンシャルはコールバックハンドラーによって提供されるベアラートークンの形式を取ります。コールバックを使用して、以下の 3 つの方法のいずれかでトークンの提供を設定できます。

- クライアント ID およびシークレット（**OAuth 2.0 クライアントクレデンシャルメカニズム**を使用）
- 設定時に手動で取得された有効期限の長いアクセストークン
- 設定時に手動で取得された有効期限の長い更新トークン

OAUTHBEARER を使用するには、Kafka ブローカーの OAuth 認証リスナー設定で **sasl.enabled.mechanisms** を **OAUTHBEARER** に設定する必要があります。詳細な設定は、「OAuth 2.0 Kafka ブローカーの設定」 を参照してください。

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
```

注記

OAUTHBEARER 認証は、プロトコルレベルで OAUTHBEARER メカニズムをサポートする Kafka クライアントでのみ使用できます。

PLAIN の概要

PLAIN は、kafkacat などの開発者ツールを含む、すべての Kafka クライアントツールでサポートされる簡易認証メカニズムです。PLAIN を OAuth 2.0 認証と使用できるように、RHEL の AMQ Streams にはサーバー側のコールバックが含まれています。PLAIN の AMQ Streams 実装は **OAuth 2.0 over PLAIN** と呼ばれます。

OAuth 2.0 の PLAIN では、クライアントクレデンシャルは ZooKeeper に保存されません。代わりに、OAUTHBEARER 認証が使用される場合と同様に、準拠した承認サーバーの背後で一元的に処理されます。

OAuth 2.0 over PLAIN コールバックを併用する場合、以下のいずれかの方法を使用して Kafka クライアントは Kafka ブローカーで認証されます。

- クライアント ID およびシークレット（**OAuth 2.0 クライアントクレデンシャルメカニズム**を使用）
- 設定時に手動で取得された有効期限の長いアクセストークン

PLAIN 認証を使用し、**username** および **password** を提供するように、クライアントを有効にする必要があります。パスワードの最初に **$accessToken:** が付けられ、その後にアクセストークンの値が続く場合は、Kafka ブローカーはパスワードをアクセストークンとして解釈します。それ以外の場合は、Kafka ブローカーは **username** をクライアント ID として解釈し、**password** をクライアントシークレットとして解釈します。

password がアクセストークンとして設定されている場合、**username** は Kafka ブローカーによってアクセストークンから取得されるプリンシパル名と同じになるように設定される必要があります。このプロセスは、**userNameClaim**、**fallbackUserNameClaim**、**fallbackUsernamePrefix**、または **userInfoEndpointUri** を使用してユーザー名の抽出を設定する方法によって異なります。また、承認サーバーによっても異なり、特にクライアント ID をアカウント名にマッピングする方法によります。

Kafka ブローカーの OAuth 認証リスナー設定で PLAIN を有効にできます。これを行うには、**PLAIN** を **sasl.enabled.mechanisms** の値に追加します。

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN
```

詳細な設定は、「OAuth 2.0 Kafka ブローカーの設定」 を参照してください。

4.10.1.1. プロパティーまたは変数での OAuth 2.0 の設定

OAuth 2.0 設定は、Java Authentication and Authorization Service(JAAS)プロパティーまたは環境変数を使用して設定できます。

- JAAS プロパティーは **server.properties** 設定ファイルで設定され、**listener.name.LISTENER-NAME.oauthbearer.sasl.jaas.config** プロパティーのキーと値のペアとして渡されます。
- 環境変数を使用する場合は、**server.properties** ファイルで **listener.name.LISTENER-NAME.oauthbearer.sasl.jaas.config** プロパティーを指定する必要がありますが、他の JAAS プロパティーを省略することができます。
  大文字または大文字の環境変数の命名規則を使用できます。

AMQ Streams OAuth 2.0 ライブラリーは、以下で始まるプロパティーを使用します。

- **oauth.** 認証の設定
- **strimzi.** OAuth 2.0 承認の設定

4.10.2. OAuth 2.0 Kafka ブローカーの設定

OAuth 2.0 認証用の Kafka ブローカー設定には、以下が関係します。

- 承認サーバーでの OAuth 2.0 クライアントの作成
- Kafka クラスターでの OAuth 2.0 認証の設定

注記

承認サーバーに関連する Kafka ブローカーおよび Kafka クライアントはどちらも OAuth 2.0 クライアントと見なされます。

4.10.2.1. 承認サーバーの OAuth 2.0 クライアント設定

セッションの開始中に受信されたトークンを検証するように Kafka ブローカーを設定するには、承認サーバーで OAuth 2.0 の **クライアント** 定義を作成し、以下のクライアントクレデンシャルが有効な状態で **機密情報** として設定することが推奨されます。

- **kafka-broker** のクライアント ID（例）
- 認証メカニズムとしてのクライアント ID およびシークレット

注記

承認サーバーのパブリックでないイントロスペクションエンドポイントを使用する場合のみ、クライアント ID およびシークレットを使用する必要があります。高速のローカル JWT トークンの検証と同様に、パブリック承認サーバーのエンドポイントを使用する場合は、通常クレデンシャルは必要ありません。

4.10.2.2. Kafka クラスターでの OAuth 2.0 認証設定

Kafka クラスターで OAuth 2.0 認証を使用するには、Kafka クラスターの OAuth 認証リスナー設定を Kafka **server.properties** ファイルで有効にします。最小の設定が必要です。また、TLS がブローカー間通信に使用される TLS リスナーを設定することもできます。

以下の方法のいずれかを使用して、承認サーバーによってトークン検証のブローカーを設定できます。

- 高速なローカルトークン検証： **JWKS** エンドポイントが署名済み JWT 形式のアクセストークンと組み合わせて使用
- **イントロスペクションエンドポイント**

OAUTHBEARER または PLAIN 認証（またはその両方）を設定できます。

以下の例は、**グローバル** リスナー設定を適用する最小設定を示しています。つまり、ブローカー間通信はアプリケーションクライアントと同じリスナーを通過します。

この例は、**sasl.enabled.mechanisms** ではなく **listener.name.LISTENER-NAME.sasl.enabled.mechanisms** を指定する、特定のリスナーの OAuth 2.0 設定も示しています。**tFineER-NAME** は、リスナーの大文字と小文字を区別しません。ここでは、リスナーの名前を **CLIENT** とし、プロパティー名は **listener.name.client.sasl.enabled.mechanisms** になります。

この例では、OAUTHBEARER 認証を使用します。

例： JWKS エンドポイントを使用した OAuth 2.0 認証の最小リスナー設定

```
sasl.enabled.mechanisms=OAUTHBEARER (1)
listeners=CLIENT://0.0.0.0:9092 (2)
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT (3)
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER (4)
sasl.mechanism.inter.broker.protocol=OAUTHBEARER (5)
inter.broker.listener.name=CLIENT (6)
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasSe
verOauthValidatorCallbackHandler (7)
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAut
hBearerLoginModule required \ (8)
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ (9)
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ (10)
  oauth.username.claim="preferred_username"  \ (11)
  oauth.client.id="kafka-broker" \ (12)
  oauth.client.secret="kafka-secret" \ (13)
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/token" ; (14)
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClien
OauthLoginCallbackHandler (15)
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000 (16)
```

(1) SASL を介した認証情報交換の **OAUTHBEARER** メカニズムを有効にします。

(2) クライアントアプリケーションが接続するリスナーを設定します。システム **hostname** は、再接続のためにクライアントが解決する必要があるアドバタイズされたホスト名として使用されます。この例では、リスナーの名前は **CLIENT** になります。

(3) リスナーのチャネルプロトコルを指定します。**SASL_SSL** は TLS 用です。**SASL_PLAINTEXT** 暗号化されていない接続に使用されますが（TLS なし）、TCP 接続層での盗難や傍受のリスクがあります。

(4) CLIENT リスナーの **OAUTHBEARER** メカニズムを指定します。クライアント名(**CLIENT**)は通常、**listeners** プロパティーで大文字で指定します。**listener.name** プロパティーの場合は小文字(**listener.name.client**)と、**listener.name.client.*** プロパティーの一部が含まれる場合に小文字が指定されます。

(5) ブローカー間の通信に **OAUTHBEARER** メカニズムを指定します。

(6) ブローカー間の通信のリスナーを指定します。設定を有効にするには、仕様が必要です。

(7) クライアントリスナーで OAuth 2.0 認証を設定します。

(8) クライアントおよびブローカー間の通信の認証設定を設定します。**oauth.client.id**、**oauth.client.secret**、および **auth.token.endpoint.uri** プロパティーは、ブローカー間の設定に関連します。

(9) 有効な発行者 URI。この発行者が発行するアクセストークンのみが許可されます。例：**https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME**

(10) JWKS エンドポイント URL。例：**https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs**

(11) トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。この値は、使用される認証フローと承認サーバーによって異なります。

12 すべてのブローカーで同じ Kafka ブローカーのクライアント ID。これは、 **kafka-brokerとして承認サーバーに登録されているクライアント** です。

13 すべてのブローカーで同じ Kafka ブローカーのシークレット。

14 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTP を使用してください。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**

15 ブローカー間の通信に OAuth 2.0 認証を有効にする（および唯一の必須）場合。

16 （任意設定）: トークンの期限が切れるとセッションの有効期限を強制し、Kafka の再認証メカニズム もアクティブにします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

以下の例は、TLS がブローカー間通信に使用される TLS リスナーの最小設定を示しています。

例： OAuth 2.0 認証の TLS リスナー設定

```
listeners=REPLICATION://kafka:9091,CLIENT://kafka:9092 1
listener.security.protocol.map=REPLICATION:SSL,CLIENT:SASL_PLAINTEXT 2
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
inter.broker.listener.name=REPLICATION
listener.name.replication.ssl.keystore.password=KEYSTORE-PASSWORD 3
listener.name.replication.ssl.truststore.password=TRUSTSTORE-PASSWORD
listener.name.replication.ssl.keystore.type=JKS
listener.name.replication.ssl.truststore.type=JKS
listener.name.replication.ssl.endpoint.identification.algorithm=HTTPS 4
listener.name.replication.ssl.secure.random.implementation=SHA1PRNG 5
listener.name.replication.ssl.keystore.location=PATH-TO-KEYSTORE 6
listener.name.replication.ssl.truststore.location=PATH-TO-TRUSTSTORE 7
listener.name.replication.ssl.client.auth=required 8
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasSe
verOauthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAut
hBearerLoginModule required \
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \
  oauth.username.claim="preferred_username" ; 9
```

1 ブローカー間の通信とクライアントアプリケーションには、別の設定が必要です。

2 TLS を使用するように **REPLICATION** リスナーと、暗号化されていないチャネルで SASL を使用するように **CLIENT** リスナーを設定します。クライアントは、実稼働環境で暗号化チャンネル (**SASL_SSL**)を使用することができます。

3 **ssl.** プロパティーは TLS 設定を定義します。

4 乱数ジェネレーターの実装。設定されていない場合は、Java プラットフォーム SDK のデフォルトが使用されます。

5 ホスト名の検証。空の文字列に設定すると、ホスト名の検証はオフになります。設定されていない場合は、デフォルト値は HTTPS で、サーバー証明書のホスト名の検証を強制します。

場合は、デフォルト値は true です。サーバー証明書のホスト名の検証を強制します。

6 リスナーのキーストアへのパス。

7 リスナーのトラストストアへのパス。

8 **REPLICATION** リスナーのクライアントが TLS 接続の確立時にクライアント証明書で認証する必要があることを指定します（ブローカー間接続に使用されます）。

9 OAuth 2.0 の **CLIENT** リスナーを設定します。承認サーバーとの接続は、セキュアな HTTPS 接続を使用する必要があります。

以下の例は、SASL を介した認証情報交換の PLAIN 認証メカニズムを使用した OAuth 2.0 認証の最小設定を示しています。高速のローカルトークン検証が使用されます。

例： PLAIN 認証の最小リスナー設定

```
listeners=CLIENT://0.0.0.0:9092 1
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT 2
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN 3
sasl.mechanism.inter.broker.protocol=OAUTHBEARER 4
inter.broker.listener.name=CLIENT 5
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasSe
verOauthValidatorCallbackHandler 6
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAut
hBearerLoginModule required \ 7
  oauth.valid.issuer.uri="http://AUTH_SERVER/auth/realms/REALM" \ 8
  oauth.jwks.endpoint.uri="https://AUTH_SERVER/auth/realms/REALM/protocol/openid-
connect/certs" \ 9
  oauth.username.claim="preferred_username"  \ 10
  oauth.client.id="kafka-broker" \ 11
  oauth.client.secret="kafka-secret" \ 12
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/token" ; 13
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClien
OauthLoginCallbackHandler 14
listener.name.client.plain.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.plain.JaasServ
rOauthOverPlainValidatorCallbackHandler 15
listener.name.client.plain.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \ 16
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ 17
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ 18
  oauth.username.claim="preferred_username"  \ 19
  oauth.token.endpoint.uri="http://AUTH_SERVER/auth/realms/REALM/protocol/openid-
connect/token" ; 20
connections.max.reauth.ms=3600000 21
```

1 クライアントアプリケーションが接続するリスナー（この例では **CLIENT** ）を設定します。システム **hostname** は、再接続のためにクライアントが解決する必要があるアドバタイズされたホスト名として使用されます。これは唯一設定されたリスナーであるため、ブローカー間の通信にも使用されます。

2

暗号化されていないチャネルで SASL を使用するように **CLIENT** リスナーの例を設定します。実稼働環境では、TCP 接続層で盗難や傍受のために保護するために、クライアントは暗号化チャン

3 SASL を介した認証情報交換の **PLAIN** 認証メカニズムを有効にします。ブローカー間通信には必要であるため、**OAUTHBEARER** も指定されます。Kafka クライアントは、使用するメカニズムを選択できます。

4 ブローカー間の通信に **OAUTHBEARER** 認証メカニズムを指定します。

5 ブローカー間の通信にリスナー（この例では **CLIENT** ）を指定します。設定が有効であるには必要。

6 OAUTHBEARER メカニズムのサーバーコールバックハンドラーを設定します。

7 **OAUTHBEARER** メカニズムを使用して、クライアントおよびブローカー間の通信の認証設定を設定します。**oauth.client.id**、**oauth.client.secret**、および **oauth.token.endpoint.uri** プロパティーは、ブローカー間の設定に関連します。

8 有効な発行者 URI。この発行者からアクセストークンのみが許可されます。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME**

9 JWKS エンドポイント URL。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs**

10 トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーを識別する **プリンシパル** です。この値は、使用される認証フローと承認サーバーによって異なります。

11 すべてのブローカーで同じ Kafka ブローカーのクライアント ID。これは、 **kafka-brokerとして承認サーバーに登録されているクライアント** です。

12 Kafka ブローカーのシークレット（すべてのブローカーで同じ）。

13 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTPS を使用します。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**

14 ブローカー間の通信の OAuth 2.0 認証を有効にします。

15 **PLAIN** 認証のサーバーコールバックハンドラーを設定します。

16 **PLAIN** 認証を使用してクライアント通信の認証設定を設定します。

oauth.token.endpoint.uri OAuth 2.0 **クライアントクレデンシャルメカニズムを使用して OAuth 2.0 を PLAIN で有効にする必須のプロパティーです。**

17 有効な発行者 URI。この発行者からアクセストークンのみが許可されます。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME**

18 JWKS エンドポイント URL。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs**

19 トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーを識別する **プリンシパル** です。この値は、使用される認証フローと承認サーバーによって異なります。

20 （必須）承認サーバーへの OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTP を

リスナーは、**OAuth 2.0 クライアントクレデンシャルメカニズムを使用してクライアントの代わりにアクセストークンを取得します。**リスナーは **username** および **password** パラメーターを **clientId** および **secret** として扱います。

21 （任意設定）: トークンの期限が切れるとセッションの有効期限を強制し、Kafka の再認証メカニズム もアクティブにします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

4.10.2.3. 高速なローカル JWT トークン検証の設定

高速なローカル JWT トークンの検証では、JWTトークンの署名がローカルでチェックされます。

ローカルチェックでは、トークンに対して以下が確認されます。

- アクセストークンに **Bearer** の (**typ**) 要求値が含まれ、トークンがタイプに準拠することを確認します。
- 有効であるか (期限切れでない) を確認します。
- トークンに **validIssuerURI** と一致する発行元があることを確認します。

承認サーバーによって発行されなかったすべてのトークンが拒否されるよう、リスナーの設定時に **有効な発行者 URI** を指定します。

高速のローカル JWT トークン検証の実行中に、承認サーバーの通信は必要はありません。OAuth 2.0 承認サーバーによって公開される **JWKs エンドポイント URI** を指定して、高速のローカル JWT トークン検証をアクティベートします。エンドポイントには、署名済み JWT トークンの検証に使用される公開鍵が含まれます。これらは、Kafka クライアントによってクレデンシャルとして送信されます。

注記

承認サーバーとの通信はすべて HTTPS を使用して実行する必要があります。

TLS リスナーでは、証明書 **トラストストア** を設定し、トラストストアファイルを示すことができます。

高速なローカル JWT トークン検証のプロパティーの例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ 1
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ 2
  oauth.jwks.refresh.seconds="300" \ 3
  oauth.jwks.refresh.min.pause.seconds="1" \ 4
  oauth.jwks.expiry.seconds="360" \ 5
  oauth.username.claim="preferred_username" \ 6
  oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \ 7
  oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \ 8
  oauth.ssl.truststore.type="PKCS12" ; 9
```

1. 有効な発行者 URI。この発行者が発行するアクセストークンのみが許可されます。例：**https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME**
2. JWKS エンドポイント URL。例：**https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs**
3. エンドポイントの更新の間隔（デフォルトは 300）。
4. JWKS 公開鍵の更新が連続して試行される間隔の最小一時停止時間 (秒単位)。不明な署名キーが検出されると、JWKS キーの更新は、最後に更新を試みてから少なくとも指定された期間は一時停止し、通常の定期スケジュール以外でスケジュールされます。キーの更新はエクスポネンシャルバックオフ(exponential backoff)のルールに従い、**oauth.jwks.refresh.seconds** に到達するまで、一時停止を増やして失敗した更新の再試行を行います。デフォルト値は 1 です。
5. JWK 証明書が期限切れになる前に有効とみなされる期間。デフォルトは **360** 秒 です。デフォルトよりも長い時間を指定する場合は、無効になった証明書へのアクセスが許可されるリスクを考慮してください。
6. トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。この値は、使用される認証フローと承認サーバーによって異なります。
7. TLS 設定で使用されるトラストストアの場所。
8. トラストストアにアクセスするためのパスワード。
9. PKCS #12 形式のトラストストアタイプ。

4.10.2.4. OAuth 2.0 イントロスペクションエンドポイントの設定

OAuth 2.0 のイントロスペクションエンドポイントを使用したトークンの検証では、受信したアクセストークンは不透明として対処されます。Kafka ブローカーは、アクセストークンをイントロスペクションエンドポイントに送信します。このエンドポイントは、検証に必要なトークン情報を応答として返します。ここで重要なのは、特定のアクセストークンが有効である場合は最新情報を返すことで、トークンの有効期限に関する情報も返します。

OAuth 2.0 のイントロスペクションベースの検証を設定するには、高速のローカル JWT トークン検証に指定された JWKs エンドポイント URI では **なくイントロスペクションエンドポイント** URI を指定します。通常、イントロスペクションエンドポイントは保護されているため、承認サーバーに応じて **クライアント ID とクライアントシークレット** を指定する必要があります。

イントロスペクションエンドポイントのプロパティーの例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.introspection.endpoint.uri="https://AUTH-SERVER-ADDRESS/introspection" \ 1
  oauth.client.id="kafka-broker" \ 2
  oauth.client.secret="kafka-broker-secret" \ 3
  oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \ 4
  oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \ 5
  oauth.ssl.truststore.type="PKCS12" \ 6
  oauth.username.claim="preferred_username" ; 7
```

1. OAuth 2.0 のイントロスペクションエンドポイント URI。例： **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token/introspect**
2. Kafka ブローカーのクライアント ID。
3. Kafka ブローカーのシークレット。
4. TLS 設定で使用されるトラストストアの場所。
5. トラストストアにアクセスするためのパスワード。
6. PKCS #12 形式のトラストストアタイプ。
7. トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。**oauth.username.claim** の値は、使用される承認サーバーによって異なります。

4.10.3. Kafka ブローカーの再認証の設定

Kafka クライアントと Kafka ブローカー間の OAuth 2.0 **セッションに Kafka セッション再認証** を使用するように OAuth リスナーを設定できます。このメカニズムは、定義された期間後に、クライアントとブローカー間の認証されたセッションを期限切れにします。セッションの有効期限が切れると、クライアントは既存のコネクションを破棄せずに再使用して、新しいセッションを即座に開始します。

セッションの再認証はデフォルトで無効になっています。これを **server.properties** ファイルで有効にできます。OAUTHBEARER または PLAIN が SASL メカニズムとして有効になっている TLS リスナーの **connections.max.reauth.ms** プロパティーを設定します。

リスナーごとにセッションの再認証を指定できます。以下に例を示します。

```
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

セッションの再認証は、クライアントによって使用される Kafka クライアントライブラリーによってサポートされる必要があります。

セッションの再認証は、**高速ローカル JWT** または**イントロスペクションエンドポイント**のトークン検証と使用できます。

クライアントの再認証

ブローカーの認証されたセッションが期限切れになると、クライアントは接続を切断せずに新しい有効なアクセストークンをブローカーに送信し、既存のセッションを再認証する必要があります。

トークンの検証に成功すると、既存の接続を使用して新しいクライアントセッションが開始されます。クライアントが再認証に失敗した場合、さらにメッセージを送受信しようとすると、ブローカーは接続を閉じます。ブローカーで再認証メカニズムが有効になっていると、Kafka クライアントライブラリー 2.2 以降を使用する Java クライアントが自動的に再認証されます。

更新トークンが使用される場合、セッションの再認証は更新トークンにも適用されます。セッションが期限切れになると、クライアントは更新トークンを使用してアクセストークンを更新します。その後、クライアントは新しいアクセストークンを使用して既存接続で再認証を行います。

OAUTHBEARER および PLAIN のセッションの有効期限

セッションの再認証が設定されている場合、OAUTHBEARER と PLAIN 認証ではセッションの有効期限は異なります。

クライアント ID およびシークレットによる方法を使用する OAUTHBEARER およびPLAIN の場合：

- ブローカーの認証されたセッションは、設定された **connections.max.reauth.ms** で期限切れになります。
- アクセストークンが設定期間前に期限切れになると、セッションは設定期間前に期限切れになります。

有効期間の **長いアクセストークンによる** 方法を使用する PLAIN の場合：

- ブローカーの認証されたセッションは、設定された **connections.max.reauth.ms** で期限切れになります。
- アクセストークンが設定期間前に期限切れになると、再認証に失敗します。セッションの再認証は試行されますが、PLAIN にはトークンを更新するメカニズムがありません。

connections.max.reauth.ms が設定されて **いない** 場合は、再認証しなくても、OAUTHBEARER および PLAIN クライアントはブローカーへの接続を無期限に維持します。認証されたセッションは、アクセストークンの期限が切れても終了しません。ただし、これは **keycloak** 承認を使用したり、カスタムオーソライザーをインストールしたりして、承認を設定する場合に考慮されます。

その他のリソース

- 「OAuth 2.0 Kafka ブローカーの設定」
- 「Kafka ブローカーの OAuth 2.0 サポートの設定」
- KIP-368

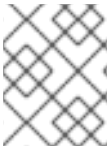
4.10.4. OAuth 2.0 Kafka クライアントの設定

Kafka クライアントは以下のいずれかで設定されます。

- 有効なアクセストークンを取得するために承認サーバーでの認証に必要なクレデンシャル（クライアント ID およびシークレット）
- 承認サーバーによって提供されるツールを使用して取得される、有効期限の長い有効な **アクセストークン** または **更新トークン**。

アクセストークンは、Kafka ブローカーに送信される唯一の情報です。承認サーバーでの認証に使用されるクレデンシャルはブローカー **に** 送信されません。クライアントによるアクセストークンの取得後、承認サーバーと通信する必要はありません。

クライアント ID とシークレットを使用した認証が最も簡単です。有効期間の長いアクセストークンまたは更新トークンを使用すると、承認サーバーツールに追加の依存関係があるため、より複雑になります。



注記

有効期間が長いアクセストークンを使用している場合は、承認サーバーでクライアントを設定し、トークンの最大有効期間を長くする必要があります。

Kafka クライアントが直接アクセストークンで設定されていない場合、クライアントは承認サーバーと通信して Kafka セッションの開始中にアクセストークンのクレデンシャルを交換します。Kafka クライアントは以下のいずれかを交換します。

- クライアント ID およびシークレット
- クライアント ID、更新トークン、および (任意の) シークレット

4.10.5. OAuth 2.0 のクライアント認証フロー

ここでは、Kafka セッションの開始時における Kafka クライアント、Kafka ブローカー、および承認ブローカー間の通信フローを説明および可視化します。フローは、クライアントとサーバーの設定によって異なります。

Kafka クライアントがアクセストークンをクレデンシャルとして Kafka ブローカーに送信する場合、トークンを検証する必要があります。

使用する承認サーバーや利用可能な設定オプションによっては、以下の使用が適している場合があります。

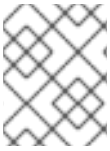
- 承認サーバーと通信しない、JWT の署名確認およびローカルトークンのイントロスペクションをベースとした高速なローカルトークン検証。
- 承認サーバーによって提供される OAuth 2.0 のイントロスペクションエンドポイント。

高速のローカルトークン検証を使用するには、トークンでの署名検証に使用される公開証明書のある JWKS エンドポイントを提供する承認サーバーが必要になります。

この他に、承認サーバーで OAuth 2.0 のイントロスペクションエンドポイントを使用することもできます。新しい Kafka ブローカー接続が確立されるたびに、ブローカーはクライアントから受け取ったアクセストークンを承認サーバーに渡し、応答を確認してトークンが有効であるかどうかを確認します。

Kafka クライアントのクレデンシャルは以下に対して設定することもできます。

- 以前に生成された有効期間の長いアクセストークンを使用した直接ローカルアクセス。
- 新しいアクセストークンの発行についての承認サーバーとの通信。



注記

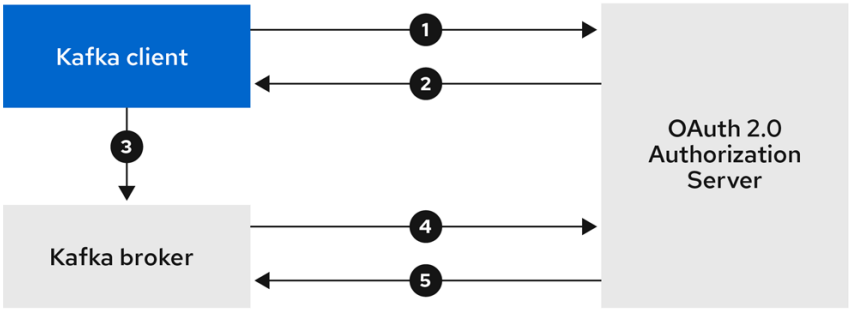
承認サーバーは不透明なアクセストークンの使用のみを許可する可能性があり、この場合はローカルトークンの検証は不可能です。

4.10.5.1. クライアント認証フローの例

Kafka クライアントおよびブローカーが以下に設定されている場合の、Kafka セッション認証中のコミュニケーションフローを確認できます。

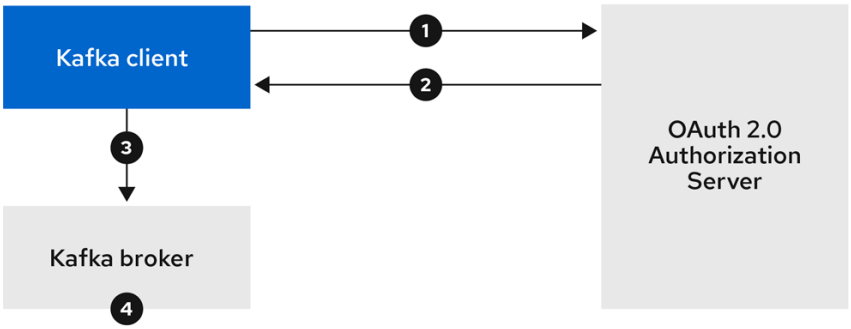
- クライアントではクライアント ID とシークレットが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントではクライアント ID およびシークレットが使用され、ブローカーによって高速のローカルトークン検証が実行される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって高速のローカル検証が実行される場合。

クライアントではクライアント ID とシークレットが使用され、ブローカーによって検証が承認サーバーに委譲される場合



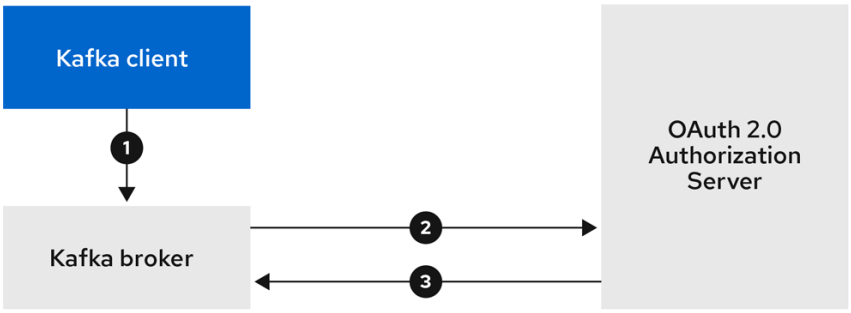
1. Kafka クライアントは承認サーバーからアクセストークンを要求します。これにはクライアント ID とシークレットを使用し、任意で更新トークンも使用します。
2. 承認サーバーによって新しいアクセストークンが生成されます。
3. Kafka クライアントは **SASL OAUTHBEARER** メカニズムを使用してアクセストークンを渡し、Kafka ブローカーの認証を行います。
4. Kafka ブローカーは、独自のクライアント ID およびシークレットを使用して、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。
5. トークンが有効な場合は、Kafka クライアントセッションが確立されます。

クライアントではクライアント ID およびシークレットが使用され、ブローカーによって高速のローカルトークン検証が実行される場合



1. Kafka クライアントは、トークンエンドポイントから承認サーバーの認証を行います。これにはクライアント ID とシークレットが使用され、任意で更新トークンも使用されます。
2. 承認サーバーによって新しいアクセストークンが生成されます。
3. Kafka クライアントは **SASL OAUTHBEARER** メカニズムを使用してアクセストークンを渡し、Kafka ブローカーの認証を行います。
4. Kafka ブローカーは、JWT トークン署名チェックおよびローカルトークンイントロスペクションを使用して、ローカルでアクセストークンを検証します。

クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合



1. Kafka クライアントは、**SASL OAUTHBEARER** メカニズムを使用して有効期限の長いアクセストークンを渡し、Kafka ブローカーの認証を行います。
2. Kafka ブローカーは、独自のクライアント ID およびシークレットを使用して、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。
3. トークンが有効な場合は、Kafka クライアントセッションが確立されます。

クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって高速のローカル検証が実行される場合

1. Kafka クライアントは、**SASL OAUTHBEARER** メカニズムを使用して有効期限の長いアクセストークンを渡し、Kafka ブローカーの認証を行います。
2. Kafka ブローカーは、JWT トークン署名チェックおよびローカルトークンイントロスペクションを使用して、ローカルでアクセストークンを検証します。

警告

トークンが取り消された場合に承認サーバーとのチェックが行われないため、高速のローカル JWT トークン署名の検証は有効期限の短いトークンにのみ適しています。トークンの有効期限はトークンに書き込まれますが、失効はいつでも発生する可能性があるため、承認サーバーと通信せずに対応することはできません。発行されたトークンはすべて期限切れになるまで有効とみなされます。

4.10.6. OAuth 2.0 認証の設定

OAuth 2.0 は、Kafka クライアントと AMQ Streams コンポーネントとの対話に使用されます。

AMQ Streams に OAuth 2.0 を使用するには、以下を行う必要があります。

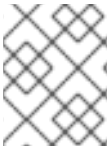
1. AMQ Streams クラスターおよび Kafka クライアントの OAuth 2.0 承認サーバーの設定
2. OAuth 2.0 を使用するよう設定された Kafka ブローカーリスナーで Kafka クラスターをデプロイまたは更新します。
3. OAuth 2.0 を使用するように Java ベースの Kafka クライアントを更新します 。

4.10.6.1. OAuth 2.0 承認サーバーとしての Red Hat Single Sign-On の設定

この手順では、Red Hat Single Sign-On を承認サーバーとしてデプロイし、AMQ Streams と統合するための設定方法を説明します。

承認サーバーは、一元的な認証および承認の他、ユーザー、クライアント、およびパーミッションの一元管理を実現します。Red Hat Single Sign-On にはレルムの概念があります。**レルム** はユーザー、クライアント、パーミッション、およびその他の設定の個別のセットを表します。デフォルトの **マスターレルム** を使用できますが、新しいレルムを作成することもできます。各レルムは独自の OAuth 2.0 エンドポイントを公開します。そのため、アプリケーションクライアントとアプリケーションサーバーはすべて同じレルムを使用する必要があります。

AMQ Streams で OAuth 2.0 を使用するには、認証レルムの作成および管理を可能にする承認サーバーのデプロイメントが必要になります。



注記

Red Hat Single Sign-On がすでにデプロイされている場合は、デプロイメントの手順を省略して、現在のデプロイメントを使用できます。

作業を開始する前の注意事項

Red Hat Single Sign-On を使用するための知識が必要です。

インストールおよび管理の手順は、以下を参照してください。

- 『サーバーインストールおよび設定ガイド』
- Server Administration Guide

前提条件

- AMQ Streams および Kafka が稼働している必要があります。

Red Hat Single Sign-On デプロイメントに関する条件:

- 「Red Hat Single Sign-On でサポートされる構成」 を確認しておく必要があります。

手順

1. Red Hat Single Sign-On をインストールします。
   ZIP ファイルまたは RPM を使用してインストールできます。
2. Red Hat Single Sign-On の Admin Console にログインし、AMQ Streams の OAuth 2.0 ポリシーを作成します。
   ログインの詳細は、Red Hat Single Sign-On のデプロイ時に提供されます。

3. レルムを作成し、有効にします。
   既存のマスターレルムを使用できます。

4. 必要に応じて、レルムのセッションおよびトークンのタイムアウトを調整します。

5. **kafka-broker** というクライアントを作成します。

6. **Settings** タブで以下を設定します。

   - **Access Type** を **Confidential** に設定します。
   - **Standard Flow Enabled** を **OFF** に設定し、このクライアントからの Web ログインを無効にします。
   - **Service Accounts Enabled** を **ON** に設定し、このクライアントが独自の名前で認証できるようにします。

7. 続行する前に **Save** クリックします。

8. **Credentials** タブにある、AMQ Streams の Kafka クラスター設定で使用するシークレットを書き留めておきます。

9. Kafka ブローカーに接続するすべてのアプリケーションクライアントに対して、このクライアント作成手順を繰り返し行います。
   新しいクライアントごとに定義を作成します。

   設定では、名前をクライアント ID として使用します。

次のステップ

承認サーバーのデプロイおよび設定後に、Kafka ブローカーが OAuth 2.0 を使用するように 設定します。

4.10.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定

この手順では、ブローカーリスナーが承認サーバーを使用して OAuth 2.0 認証を使用するように、Kafka ブローカーを設定する方法について説明します。

TLS リスナーを設定して、暗号化されたインターフェースで OAuth 2.0 を使用することが推奨されます。プレーンリスナーは推奨されません。

選択した承認サーバーをサポートするプロパティーを使用して、Kafka ブローカーを設定します。また、実装する承認のタイプを使用します。

作業を開始する前の注意事項

Kafka ブローカーリスナーの設定および認証の詳細は、以下を参照してください。

- リスナー
- 暗号化および認証

リスナー設定で使用されるプロパティーの説明は、以下を参照してください。

- OAuth 2.0 Kafka ブローカーの設定

前提条件

- AMQ Streams および Kafka が稼働している必要があります。
- OAuth 2.0 の承認サーバーがデプロイされている必要があります。

手順

1. **server.properties** ファイルで Kafka ブローカーリスナーを設定します。
   たとえば、OAUTHBEARER メカニズムを使用します。

```
sasl.enabled.mechanisms=OAUTHBEARER
listeners=CLIENT://0.0.0.0:9092
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
inter.broker.listener.name=CLIENT
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server
JaasServerOauthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbear
er.OAuthBearerLoginModule required ;
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.J
asClientOauthLoginCallbackHandler
```

2. **listener.name.client.oauthbearer.sasl.jaas.config** の一部としてブローカー接続を設定します。
   以下の例は、接続設定オプションを示しています。

   例 1: JWKS エンドポイント設定を使用したローカルトークンの検証

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbear
er.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME" \
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-
NAME/protocol/openid-connect/certs" \
  oauth.jwks.refresh.seconds="300" \
  oauth.jwks.refresh.min.pause.seconds="1" \
  oauth.jwks.expiry.seconds="360" \
  oauth.username.claim="preferred_username" \
  oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \
  oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \
  oauth.ssl.truststore.type="PKCS12" ;
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

   例 2: OAuth 2.0 イントロスペクションエンドポイントを使用したトークン検証の承認サーバーへの委譲

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbear
er.OAuthBearerLoginModule required \
  oauth.introspection.endpoint.uri="https://AUTH-SERVER-
ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/introspection" \
  # ...
```

3. 必要な場合は、承認サーバーへのアクセスを設定します。
   この手順は、通常、**サービスメッシュ** などの技術を使用してコンテナー外でセキュアなチャネルを設定する場合を除き、実稼働環境に必要です。

a. セキュアな承認サーバーに接続するためのカスタムトラストストアを提供します。承認サーバーへのアクセスには常に SSL が必要です。
プロパティーを設定してトラストストアを設定します。

以下に例を示します。

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  # ...
  oauth.client.id="kafka-broker" \
  oauth.client.secret="kafka-broker-secret" \
  oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \
  oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \
  oauth.ssl.truststore.type="PKCS12" ;
```

b. 証明書ホスト名がアクセス URL ホスト名と一致しない場合は、証明書のホスト名の検証を無効にできます。

```
oauth.ssl.endpoint.identification.algorithm=""
```

このチェックは、クライアントによる承認サーバーへの接続が認証されるようにします。非実稼働環境で検証をオフにする場合があります。

4. 選択した認証フローに応じて追加のプロパティーを設定します。

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  # ...
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token" \ 1
  oauth.custom.claim.check="@.custom == 'custom-value'" 2
  oauth.check.audience="true" 3
  oauth.valid.issuer.uri="https://https://AUTH-SERVER-ADDRESS/auth/REALM-NAME" \ 4
  oauth.client.id="kafka-broker" \ 5
  oauth.client.secret="kafka-broker-secret" \ 6
  oauth.refresh.token="REFRESH-TOKEN-FOR-KAFKA-BROKERS" \ 7
  oauth.access.token="ACCESS-TOKEN-FOR-KAFKA-BROKERS" ; 8
```

1 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTP を使用してください。**KeycloakRBACAuthorizer** を使用する場合や、ブローカー間の通信に OAuth 2.0 対応のリスナーが使用される場合に必要です。

2 （オプション） **カスタム要求チェック**。検証中に追加のカスタムルールを JWT アクセストークンに適用する JsonPath フィルタークエリー。アクセストークンに必要なデータが含まれていないと拒否されます。イントロスペクションエンドポイントメソッドを使用する場合 、 カスタムチェックはイントロスペクションエンドポイントの応答 JSON に適用されます。

3 （オプション） **オーディエンスチェック**。承認サーバーによって **aud** （オーディエンス）クレームが提供され、オーディエンスチェックを強制する場合は、**ouath.check.audience** を **true** に設定します。オーディエンスチェックによって、トークンの目的の受信者が特定されます。その結果、Kafka ブローカーは **aud** クレームに **clientId** のないトークンを拒否します。デフォルトは **false** です。

4 有効な発行者 URI。この発行者が発行するアクセストークンのみが許可されます。（常に必須です。）

5 すべてのブローカーで同じ Kafka ブローカーの設定済みのクライアント ID。これは、**kafka-broker**として承認サーバーに登録されているクライアント です。イントロスペクションエンドポイントがトークンの検証に使用される場合、または **KeycloakRBACAuthorizer** が使用される場合に必要です。

6 すべてのブローカーで同じ Kafka ブローカーに設定されたシークレット。ブローカーが承認サーバーに対して認証する必要がある場合、クライアントシークレット、アクセストークン、または更新トークンを指定する必要があります。

7 （任意設定）: Kafka ブローカーの有効期限の長い更新トークン。

8 （任意設定）: Kafka ブローカーの有効期限の長いアクセストークン。

5. OAuth 2.0 認証の適用方法や、使用されている承認サーバーのタイプに応じて、追加の設定を追加します。

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  # ...
  oauth.check.issuer=false \ 1
  oauth.fallback.username.claim="CLIENT-ID" \ 2
  oauth.fallback.username.prefix="CLIENT-ACCOUNT" \ 3
  oauth.valid.token.type="bearer" \ 4
  oauth.userinfo.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/userinfo" ; 5
```

1 承認サーバーが **iss** クレームを提供しない場合は、発行者チェックを行うことができません。このような場合、**oauth.check.issuer** を **false** に設定し、**oauth.valid.issuer.uri** を指定しないようにします。デフォルトは **true** です。

2 承認サーバーは、通常ユーザーとクライアントの両方を識別する単一の属性を提供しない場合があります。クライアントが独自の名前で認証される場合、サーバーによって **クライアント ID** が提供されることがあります。更新トークンまたはアクセストークンを取得するために、ユーザー名およびパスワードを使用してユーザーが認証される場合、サーバーによってクライアント ID の他に **ユーザー名** が提供されることがあります。プライマリーユーザー ID 属性が使用できない場合は、このフォールバックオプションで、使用するユーザー名クレーム (属性) を指定します。

3 **oauth.fallback.username.claim** が適用される場合、ユーザー名クレームの値とフォールバックユーザー名クレームの値が競合しないようにする必要もあることがあります。**producer** というクライアントが存在し、**producer** という通常ユーザーも存在する場合について考えてみましょう。この 2 つを区別するには、このプロパティーを使用してクライアントのユーザー ID に接頭辞を追加します。

4 （ **oauth.introspection.endpoint.uri**を使用する場合のみ該当）: 使用している認証サーバーによっては、イントロスペクションエンドポイントによって **トークンタイプ** 属性が返されるかどうかは分からず、異なる値が含まれることがあります。イントロスペクションエンドポイントからの応答に含まれなければならない有効なトークンタイプ値を指定できます。

5 （ **oauth.introspection.endpoint.uri**を使用する場合のみ該当）: イントロスペクションエ

次のステップ

- OAuth 2.0 を使用するように Kafka クライアントを設定 します。

4.10.6.3. OAuth 2.0 を使用するよう Kafka Java クライアントを設定

この手順では、Kafka ブローカーとの対話に OAuth 2.0 を使用するように Kafka プロデューサーおよびコンシューマー API を設定する方法を説明します。

クライアントコールバックプラグインを **pom.xml** ファイルに追加し、システムプロパティーを設定します。

前提条件

- AMQ Streams および Kafka が稼働している必要があります。
- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている必要があります。
- Kafka ブローカーが OAuth 2.0 に対して設定されている必要があります。

手順

1. OAuth 2.0 サポートのあるクライアントライブラリーを Kafka クライアントの **pom.xml** ファイルに追加します。

```
<dependency>
 <groupId>io.strimzi</groupId>
 <artifactId>kafka-oauth-client</artifactId>
 <version>0.7.1.redhat-00003</version>
</dependency>
```

2. コールバックのシステムプロパティーを設定します。
以下に例を示します。

```
System.setProperty(ClientConfig.OAUTH_TOKEN_ENDPOINT_URI, “https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token”); (1)
System.setProperty(ClientConfig.OAUTH_CLIENT_ID, "CLIENT-NAME"); (2)
System.setProperty(ClientConfig.OAUTH_CLIENT_SECRET, "CLIENT_SECRET"); (3)
System.setProperty(ClientConfig.OAUTH_SCOPE, "SCOPE-VALUE") (4)
```

(1) 承認サーバーのトークンエンドポイントの URI です。

(2) クライアント ID。承認サーバーで **client** を作成するときに使用される名前です。

(3) 承認サーバーで **client** を作成するときに作成されるクライアントシークレット。

(4) (任意設定): トークンエンドポイントからトークンを要求するための **scope**。認証サーバーでは、クライアントによるスコープの指定が必要になることがあります。

3. Kafka クライアント設定の TLS 暗号化接続で **OAUTHBEARER** OR **PLAIN** メカニズムを有効にします。
以下に例を示します。

Kafka クライアントの OAUTHBEARER の有効化

```
props.put("sasl.jaas.config",
"org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required;");
props.put("security.protocol", "SASL_SSL");
props.put("sasl.mechanism", "OAUTHBEARER");
props.put("sasl.login.callback.handler.class",
"io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler");
```

Kafka クライアントの PLAIN の有効化

```
props.put("sasl.jaas.config", "org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"$CLIENT_ID_OR_ACCOUNT_NAME\"
password=\"$SECRET_OR_ACCESS_TOKEN\" ;");
props.put("security.protocol", "SASL_SSL"); (1)
props.put("sasl.mechanism", "PLAIN");
```

1. この例では、TLS 接続で **SASL_SSL** を使用します。ローカル開発のみでは、暗号化されていない接続では **SASL_PLAINTEXT** を使用します。

4. Kafka クライアントが Kafka ブローカーにアクセスできることを確認します。

4.11. OAUTH 2.0 トークンベース承認の使用

トークンベースの認証に OAuth 2.0 と Red Hat Single Sign-On を使用している場合、Red Hat Single Sign-On を使用して承認ルールを設定し、Kafka ブローカーへのクライアントのアクセスを制限することもできます。認証はユーザーのアイデンティティーを確立します。承認は、そのユーザーのアクセスレベルを決定します。

AMQ Streams は、Red Hat Single Sign-On の Authorization Services による OAuth 2.0 トークンベースの承認をサポートします。これにより、セキュリティーポリシーとパーミッションの一元的な管理が可能になります。

Red Hat Single Sign-On で定義されたセキュリティーポリシーおよびパーミッションは、Kafka ブローカーのリソースへのアクセスを付与するために使用されます。ユーザーとクライアントは、Kafka ブローカーで特定のアクションを実行するためのアクセスを許可するポリシーに対して照合されます。

Kafka では、デフォルトですべてのユーザーがブローカーに完全アクセスできます。また、アクセス制御リスト (ACL) を基にして承認を設定するために **AclAuthorizer** プラグインが提供されます。

ZooKeeper には、 **ユーザー名** を基にしてリソースへのアクセスを付与または拒否する ACL ルールが保存されます。ただし、Red Hat Single Sign-On を使用した OAuth 2.0 トークンベースの承認では、より柔軟にアクセス制御を Kafka ブローカーに実装できます。さらに、Kafka ブローカーで OAuth 2.0 の承認および ACL が使用されるように設定することができます。

その他のリソース

- OAuth 2.0 トークンベース認証の使用
- Kafka の承認
- Red Hat Single Sign-On のドキュメント

4.11.1. OAuth 2.0 の承認メカニズム

AMQ Streams の OAuth 2.0 での承認では、Red Hat Single Sign-On サーバーの Authorization Services REST エンドポイントを使用して、Red Hat Single Sign-On を使用するトークンベースの認証が拡張されます。これは、定義されたセキュリティーポリシーを特定のユーザーに適用し、そのユーザーの異なるリソースに付与されたパーミッションの一覧を提供します。ポリシーはロールとグループを使用して、パーミッションをユーザーと照合します。OAuth 2.0 の承認では、Red Hat Single Sign-On の Authorization Services から受信した、ユーザーに付与された権限のリストを基にして、権限がローカルで強制されます。

4.11.1.1. Kafka ブローカーのカスタムオーソライザー

AMQ Streams では、Red Hat Single Sign-On の **オーソライザー** (**KeycloakRBACAuthorizer**) が提供されます。Red Hat Single Sign-On によって提供される Authorization Services で Red Hat Single Sign-On REST エンドポイントを使用できるようにするには、Kafka ブローカーでカスタムオーソライザーを設定します。

オーソライザーは必要に応じて付与された権限のリストを承認サーバーから取得し、ローカルで Kafka ブローカーに承認を強制するため、クライアントの要求ごとに迅速な承認決定が行われます。

4.11.2. OAuth 2.0 承認サポートの設定

この手順では、Red Hat Single Sign-On の Authorization Services を使用して、OAuth 2.0 承認を使用するように Kafka ブローカーを設定する方法を説明します。

作業を始める前に

特定のユーザーに必要なアクセス、または制限するアクセスについて検討してください。Red Hat Single Sign-On では、Red Hat Single Sign-On の **グループ**、**ロール**、**クライアント**、および **ユーザー** の組み合わせを使用して、アクセスを設定できます。

通常、グループは組織の部門または地理的な場所を基にしてユーザーを照合するために使用されます。また、ロールは職務を基にしてユーザーを照合するために使用されます。

Red Hat Single Sign-On を使用すると、ユーザーおよびグループを LDAP で保存できますが、クライアントおよびロールは LDAP で保存できません。ユーザーデータへのアクセスとストレージを考慮して、承認ポリシーの設定方法を選択する必要がある場合があります。

注記

スーパーユーザー は、Kafka ブローカーに実装された承認にかかわらず、常に制限なく Kafka ブローカーにアクセスできます。

前提条件

- AMQ Streams は、トークンベースの認証 に Red Hat Single Sign-On と OAuth 2.0 を使用するように設定されている必要があります。承認を設定するときに、同じ Red Hat Single Sign-On サーバーエンドポイントを使用する必要があります。
- Red Hat Single Sign-On の ドキュメント で説明されているように、Red Hat Single Sign-On の Authorization Services のポリシーおよびパーミッションを管理する方法を理解する必要があります。

手順

1. Red Hat Single Sign-On の Admin Console にアクセスするか、Red Hat Single Sign-On の Admin CLI を使用して、OAuth 2.0 認証の設定時に作成した Kafka ブローカークライアントの Authorization Services を有効にします。

2. 承認サービスを使用して、クライアントのリソース、承認スコープ、ポリシー、およびパーミッションを定義します。

3. ロールとグループをユーザーとクライアントに割り当てて、パーミッションをユーザーとクライアントにバインドします。

4. Red Hat Single Sign-On 承認を使用するように Kafka ブローカーを設定します。
以下を Kafka **server.properties** 設定ファイルに追加し、Kafka のオーソライザーをインストールします。

```
authorizer.class.name=io.strimzi.kafka.oauth.server.authorizer.KeycloakRBACAuthorizer
principal.builder.class=io.strimzi.kafka.oauth.server.authorizer.JwtKafkaPrincipalBuilder
```

5. Kafka ブローカーの設定を追加して、承認サーバーと Authorization Services にアクセスします。
ここでは、追加のプロパティーとして **server.properties** に追加された設定の例を紹介しますが、大文字または大文字の命名規則を使用して環境変数として定義することもできます。

```
strimzi.authorization.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token" (1)
strimzi.authorization.client.id="kafka" (2)
```

(1) Red Hat Single Sign-On への OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTP を使用してください。

(2) 承認サービスが有効になっている Red Hat Single Sign-On の OAuth 2.0 クライアント定義のクライアント ID。通常、**kafka** が ID として使用されます。

6. （任意）特定の Kafka クラスターの設定を追加します。
以下に例を示します。

```
strimzi.authorization.kafka.cluster.name="kafka-cluster" (1)
```

(1) 特定の Kafka クラスターの名前。名前はターゲットパーミッションに使用され、同じ Red Hat Single Sign-On レルム内で複数のクラスターを管理できます。デフォルト値は **kafka-cluster** です。

7. （任意設定）: 簡易承認に委譲します。
以下に例を示します。

```
strimzi.authorization.delegate.to.kafka.acl="false" (1)
```

(1) Red Hat Single Sign-On の Authorization Services のポリシーによってアクセスが拒否されている場合は、Kafka **AclAuthorizer** に承認を委譲します。デフォルトは **false** です。

8. （任意設定）: 承認サーバーへの TLS 接続の設定を追加します。
以下に例を示します。

```
strimzi.authorization.ssl.truststore.location=<path-to-truststore> 1
strimzi.authorization.ssl.truststore.password=<my-truststore-password> 2
strimzi.authorization.ssl.truststore.type=JKS 3
strimzi.authorization.ssl.secure.random.implementation=SHA1PRNG 4
strimzi.authorization.ssl.endpoint.identification.algorithm=HTTPS 5
```

1. 証明書が含まれるトラストストアへのパス。
2. トラストストアのパスワード。
3. トラストストアタイプ。設定されていない場合は、デフォルトの Java キーストアタイプが使用されます。
4. 乱数ジェネレーターの実装。設定されていない場合は、Java プラットフォーム SDK のデフォルトが使用されます。
5. ホスト名の検証。空の文字列に設定すると、ホスト名の検証はオフになります。設定されていない場合、デフォルト値は **HTTPS** で、サーバー証明書のホスト名の検証を強制します。

9. （任意設定）: 承認サーバーから付与の更新を設定します。付与(Grants)更新ジョブは、アクティブなトークンを列挙し、それぞれの最新の付与を要求することで機能します。
以下に例を示します。

```
strimzi.authorization.grants.refresh.period.seconds="120" 1
strimzi.authorization.grants.refresh.pool.size="10" 2
```

1. 承認サーバーからの付与リストを更新する頻度を指定します（デフォルトでは 1 分ごと）。デバッグの目的で更新をオフにするには、**"0"** に設定します。
2. 付与更新ジョブによって使用されるスレッドプールのサイズ（並列処理レベル）を指定します。デフォルト値は **"5"** です。

10. クライアントまたは特定のロールを持つユーザーとして Kafka ブローカーにアクセスして、設定したパーミッションを検証し、必要なアクセス権限があり、付与されるべきでないアクセス権限がないことを確認します。

4.12. OPA ポリシーベースの承認の使用

Open Policy Agent (OPA) は、オープンソースのポリシーエンジンです。OPA と AMQ Streams を統合して、Kafka ブローカーでのクライアント操作を許可するポリシーベースの承認メカニズムとして機能します。

クライアントからリクエストが実行されると、OPA は Kafka アクセスに定義されたポリシーに対してリクエストを評価し、リクエストを許可または拒否します。

注記

Red Hat は OPA サーバーをサポートしません。

関連情報

- Open Policy Agent の Web サイト

4.12.1. OPA ポリシーの定義

OPA と AMQ Streams を統合する前に、ポリシーを定義する方法を指定して、きめ細かいアクセス制御を提供することを検討してください。

Kafka クラスター、コンシューマーグループ、およびトピックのアクセス制御を定義できます。たとえば、プロデューサークライアントから特定のブローカートピックへの書き込みアクセスを許可する承認ポリシーを定義できます。

このポリシーでは、以下を指定できます。

- プロデューサークライアントに関連する**ユーザープリンシパル** と **ホストアドレス**
- クライアントで許可される**操作**
- **リソースタイプ** (**topic**)およびポリシーが適用される **リソース名**

決定と拒否の決定はポリシーに書き込まれます。また、提供されるリクエストおよびクライアント ID データに基づいて応答が提供されます。

この例では、プロデューサークライアントはトピックへの書き込みを許可するポリシーを満たす必要があります。

4.12.2. OPA への接続

Kafka が OPA ポリシーエンジンにアクセスしてアクセス制御ポリシーをクエリーできるようにするには、Kafka **server.properties** ファイルでカスタム OPA オーソライザー(**kafka-authorizer-opa-VERSION.jar**)を設定します。

クライアントがリクエストが実行されると、OPA ポリシーエンジンは、指定の URL アドレスと REST エンドポイントを使用してプラグインによってクエリーされます。これは定義されたポリシーの名前である必要があります。

このプラグインは、ポリシーに対して確認する JSON 形式で、クライアントリクエスト（ユーザープリンシパル、操作、およびリソース）の詳細を提供します。詳細には、クライアントの一意のアイデンティティーが含まれます。たとえば、TLS 認証が使用される場合にクライアント証明書から識別名を取得します。

opa はデータを使用して応答を提供します（ **true** または **false** - プラグインにプラグインを設定して要求を許可または拒否します）。

4.12.3. OPA 承認サポートの設定

この手順では、OPA 承認を使用するように Kafka ブローカーを設定する方法を説明します。

作業を始める前に

特定のユーザーに必要なアクセス、または制限するアクセスについて検討してください。**ユーザー** と Kafka **リソース** の組み合わせを使用して、OPA ポリシーを定義できます。

OPA を設定して、LDAP データソースからユーザー情報を読み込むことができます。

注記

スーパーユーザー は、Kafka ブローカーに実装された承認にかかわらず、常に制限なく Kafka ブローカーにアクセスできます。

前提条件

- OPA サーバーは、接続に使用できる必要があります。
- Kafka の op オーソライザープラグイン

手順

1. Kafka ブローカーで操作を実行するために、クライアントリクエストを承認するために必要な OPA ポリシーを作成します。
「 OPA ポリシーの定義 」を参照してください。

   Kafka ブローカーが OPA を使用するように設定します。

2. Kafka の OPA オーソライザープラグイン をインストールします。
「 OPA への接続」を参照して ください。

   プラグインファイルが Kafka クラスパスに含まれていることを確認してください。

3. 以下を Kafka **server.properties** 設定ファイルに追加し、OPA プラグインを有効にします。

```
authorizer.class.name: com.bisnode.kafka.authorization.OpaAuthorizer
```

4. OPA ポリシーエンジンおよびポリシーにアクセスするには、Kafka ブローカーの **server.properties** に設定を追加します。
以下に例を示します。

```
opa.authorizer.url=https://OPA-ADDRESS/allow (1)
opa.authorizer.allow.on.error=false (2)
opa.authorizer.cache.initial.capacity=50000 (3)
opa.authorizer.cache.maximum.size=50000 (4)
opa.authorizer.cache.expire.after.seconds=600000 (5)
super.users=User:alice;User:bob (6)
```

(1) （必須）オーソライザープラグインがクエリーするポリシーの OAuth 2.0 トークンエンドポイント URL。この例では、ポリシーは **allow** という名前です。

(2) オーソライザープラグインが OPA ポリシーエンジンで接続できない場合に、デフォルトでクライアントを許可または拒否されるかどうかを指定するフラグ。

(3) ローカルキャッシュの初期容量（バイト単位）。このキャッシュを使用して、プラグインがすべてのリクエストに対して OPA ポリシーエンジンにクエリーされないようにします。

(4) ローカルキャッシュの最大容量（バイト単位）。

(5) ローカルキャッシュが OPA ポリシーエンジンからリロードすることで、ローカルキャッシュを更新する時間（ミリ秒単位）。

6 スーパーユーザーとして扱われるユーザープリンシパルのリスト。このリストのユーザープリンシパルは、Open Policy Agent ポリシーをクエリーしなくても常に許可されます。

認証および承認のオプションの詳細については、Open Policy Agent の Web サイト を参照してください。

5. 正しい承認を持つクライアントを使用して Kafka ブローカーにアクセスして、設定されたパーミッションを確認します。

4.13. ログ

Kafka ブローカーは、Log4j をロギングインフラストラクチャーとして使用します。デフォルトでは、ロギング設定は **log4j.properties** 設定ファイルから読み取られ、**/opt/kafka/config/** ディレクトリーまたはクラスパスのいずれかになります。設定ファイルの場所と名前は、Java プロパティー **log4j.configuration** を使用して変更できます。これは、**KAFKA_LOG4J_OPTS** 環境変数を使用して Kafka に渡すことができます。

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.config";
/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.properties
```

Log4j の設定に関する詳細は、Log4j マニュアル を参照してください。

4.13.1. Kafka ブローカーロガーのロギングレベルを動的に変更

Kafka ブローカーロギングは、各ブローカーの複数の **ブローカーロガー** によって提供されます。ブローカーを再起動することなく、ブローカーロガーのロギングレベルを動的に変更できます。ログで返される詳細レベルを増やすには、**INFO** から **DEBUG** に変更します。たとえば、Kafka クラスターでパフォーマンスの問題を調査するのに便利です。

ブローカーロガーは、デフォルトのロギングレベルに動的にリセットすることもできます。

前提条件

- AMQ Streams がホストにインストールされていること。
- ZooKeeper および Kafka が稼働している必要があります。

手順

1. **kafka** ユーザーに切り替えます。

```
su - kafka
```

2. **kafka-configs.sh** ツールを使用して、ブローカーのブローカーロガーすべてを一覧表示します。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server BOOTSTRAP-ADDRESS --describe --
entity-type broker-loggers --entity-name BROKER-ID
```

たとえば、ブローカー **0** の場合は以下のようになります。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type broker-loggers --entity-name 0
```

これは、各ロガーのロギングレベル **TRACE**、**DEBUG**、**INFO**、**WARN**、**ERROR**、または **FATAL** を返します。以下に例を示します。

```
#...
kafka.controller.ControllerChannelManager=INFO sensitive=false synonyms={}
kafka.log.TimeIndex=INFO sensitive=false synonyms={}
```

3. 1つ以上のブローカーロガーのログレベルを変更します。**--alter** および **--add-config** オプションを使用して、各ロガーとそのレベルを二重引用符で区切って指定します。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server BOOTSTRAP-ADDRESS --alter --add-config "LOGGER-ONE=NEW-LEVEL,LOGGER-TWO=NEW-LEVEL" --entity-type broker-loggers --entity-name BROKER-ID
```

たとえば、ブローカー **0** の場合は以下のようになります。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config "kafka.controller.ControllerChannelManager=WARN,kafka.log.TimeIndex=WARN" --entity-type broker-loggers --entity-name 0
```

成功すると、以下のようになります。

```
Completed updating config for broker: 0.
```

ブローカーロガーのリセット

kafka-configs.sh ツールを使用して、1つ以上のブローカーロガーをデフォルトのロギングレベルにリセットできます。**--alter** および **--delete-config** オプションを使用して、各ブローカーロガーを二重引用符で区切って指定します。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config "LOGGER-ONE,LOGGER-TWO" --entity-type broker-loggers --entity-name BROKER-ID
```

関連情報

- Apache Kafka ドキュメントの「 Broker Config 」の更新。

第5章 トピック

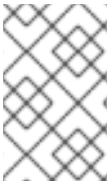
Kafka のメッセージは常にトピックに送信または受信されます。本章では、Kafka トピックを設定し、管理する方法を説明します。

5.1. パーティションおよびレプリカ

Kafka のメッセージは常にトピックに送信または受信されます。トピックは常に、1つまたは複数のパーティションに分割されます。パーティションはシャードとして機能します。つまり、プロデューサーによって送信されたすべてのメッセージは常に単一のパーティションにのみ書き込まれます。メッセージを異なるパーティションにシャーディングにより、トピックは水平的にスケーリングしやすくなります。

各パーティションにはレプリカを1つ以上設定できます。レプリカはクラスターの異なるブローカーに保存されます。トピックの作成時には、**レプリケーション係数** を使用してレプリカ数を設定できます。**レプリケーションファクターは、クラスター内で保持されるコピー数を定義** します。指定のパーティションのレプリカの1つがリーダーとして選択されます。リーダーレプリカはプロデューサーによって新しいメッセージを送信し、コンシューマーによってメッセージを消費するために使用されます。他のレプリカはフォロワーレプリカになります。フォロワーはリーダーを複製します。

リーダーに障害が発生した場合、フォロワーのいずれかが新しいリーダーに自動的に実行されます。各サーバーは、その一部のパーティションのリーダーとして機能し、他のコンポーネントのフォロワーであるため、負荷もクラスター内で均等に分散されます。



注記

レプリケーションファクターは、リーダーとフォロワーを含むレプリカ数を決定します。たとえば、レプリケーション係数を **3** に設定すると、1つのリーダーとフォロワーレプリカが 2 つあります。

5.2. メッセージの保持

メッセージ保持ポリシーでは、メッセージが Kafka ブローカーに保存される期間を定義します。これは、時間、パーティションサイズ、またはその両方に基づいて定義できます。

たとえば、メッセージを保持されるように定義できます。

- 7 日間
- ペーションが 1 GB のメッセージがあるまで。制限に達すると、最も古いメッセージが削除されます。
- 7 日の場合、または 1 GB の上限に達するまで。最初に使用する制限。

警告

Kafka ブローカーはメッセージをログセグメントに保存します。保持ポリシーを超過するメッセージは、新規ログコレクターが作成される場合にのみ削除されます。新しいログセグメントは、以前のログセグメントサイズが設定済みのログセグメントサイズを超える際に作成されます。また、ユーザーは定期的に新しいセグメントを作成するよう要求することもできます。

さらに、Kafka ブローカーはコンパクトなポリシーをサポートします。

compacted ポリシーのあるトピックでは、ブローカーは常にキーごとに最後のメッセージのみを保持します。同じキーを持つ古いメッセージはパーティションから削除されます。圧縮処理は定期的に実行されるアクションなので、同じキーを持つ新しいメッセージがパーティションに送信されるとすぐには実行されません。代わりに、古いメッセージが削除されるまで時間がかかる場合があります。

メッセージ保持設定オプションの詳細については、「トピックの設定」 を参照してください。

5.3. トピックの自動作成

プロデューサーまたはコンシューマーが存在しないトピックからメッセージを送信または受信しようとすると、Kafka はデフォルトでそのトピックを自動的に作成します。この動作は、**auto.create.topics.enable** 設定プロパティーで制御され、デフォルトで **true** に設定されます。

これを無効にするには、Kafka ブローカー設定ファイルで **auto.create.topics.enable** を **false** に設定します。

```
auto.create.topics.enable=false
```

5.4. トピックの削除

Kafka では、トピックの削除を無効にできます。これは、デフォルトで **true** に設定されます（トピックの削除が可能）、**delete.topic.enable** プロパティーで設定されます。このプロパティーを **false** に設定すると、トピックの削除とトピックの削除の試行はすべて成功されますが、トピックは削除されません。

```
delete.topic.enable=false
```

5.5. トピックの設定

自動作成されたトピックは、ブローカープロパティーファイルで指定できるデフォルトのトピック設定を使用します。ただし、トピックを手動で作成する場合は、作成時に設定を指定できます。また、トピックの作成後に変更することもできます。手動で作成されたトピックの主なトピック設定オプションは次のとおりです。

cleanup.policy

保持ポリシーを **delete** または **compact** に設定します。**delete** ポリシーは古いレコードを削除します。**compact** ポリシーにより、ログコンパクションが有効になります。デフォルト値は **delete** です。ログコンパクションの詳細は、Kafka の Web サイト を参照してください。

compression.type

保存されたメッセージに使用される圧縮を指定します。有効な値は、**gzip**、**snappy**、**lz4**、**uncompressed**（圧縮なし）および **producer**（プロデューサーによって使用される圧縮コードが含まれます）です。デフォルト値は **producer** です。

max.message.bytes

Kafka ブローカーによって許可されるメッセージのバッチの最大サイズ（バイト単位）。デフォルト値は **1000012** です。

min.insync.replicas

書き込みが成功したとみなされるには、同期する必要のあるレプリカの最小数。デフォルト値は **1** です。

retention.ms

ログセグメントが保持される最大期間（ミリ秒単位）。この値よりも古いログセグメントが削除されます。デフォルト値は **604800000**（7 日）です。

retention.bytes

パーティションが保持する最大バイト数。パーティションサイズがこの制限を超えると、最も古いログセグメントが削除されます。**-1** の値は制限なしを意味します。デフォルト値は **-1** です。

segment.bytes

単一のコミットログセグメントファイルの最大サイズ（バイト単位）。セグメントのサイズが到達すると、新しいセグメントが開始されます。デフォルト値は **1073741824** バイト(1 gibibyte)です。

サポートされるすべてのトピック設定オプションの一覧は、付録B *トピック設定パラメーター* を参照してください。

自動作成されたトピックのデフォルトは、同様のオプションを使用して Kafka ブローカー設定に指定できます。

log.cleanup.policy

上記の **cleanup.policy** を参照してください。

compression.type

上記の **compression.type** を参照してください。

message.max.bytes

上記の **max.message.bytes** を参照してください。

min.insync.replicas

上記の **min.insync.replicas** を参照してください。

log.retention.ms

上記の **retention.ms** を参照してください。

log.retention.bytes

上記の **retention.bytes** を参照してください。

log.segment.bytes

上記の **segment.bytes** を参照してください。

default.replication.factor

自動的に作成されるトピックのデフォルトのレプリケーション係数。デフォルト値は **1** です。

num.partitions

自動作成されたトピックのデフォルトパーティション数。デフォルト値は **1** です。

サポートされるすべての Kafka ブローカー設定オプションのリストは、付録A *ブローカー設定パラメーター* を参照してください。

5.6. 内部トピック

内部トピックは、Kafka ブローカーおよびクライアントによって内部によって作成および使用されます。Kafka には複数の内部トピックがあります。これらはコンシューマーオフセット (**__consumer_offsets**)またはトランザクションの状態(**__transaction_state**)を保存するために使用されます。これらのトピックは、プレフィックス **offsets.topic.** および **transaction.state.log.** で始まる専用の Kafka ブローカー設定オプションを使用して設定できます。最も重要な設定オプションは、以下のとおりです。

offsets.topic.replication.factor

__consumer_offsets トピックのレプリカ数。デフォルト値は **3** です。

offsets.topic.num.partitions

__consumer_offsets トピックのパーティション数。デフォルト値は **50** です。

transaction.state.log.replication.factor

__transaction_state トピックのレプリカ数。デフォルト値は **3** です。

transaction.state.log.num.partitions

__transaction_state トピックのパーティション数。デフォルト値は **50** です。

transaction.state.log.min.isr

__transaction_state トピックへの書き込みの確認に使用する必要があるレプリカの最小数。この最小値が満たされない場合、プロデューサーは例外によって失敗します。デフォルト値は **2** です。

5.7. トピックの作成

kafka-topics.sh ツールを使用してトピックを管理できます。**kafka-topics.sh** AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。

トピックの作成

1. **kafka-topics.sh** ユーティリティーを使用してトピックを作成し、以下を指定します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポート。
   - **--create** オプションに作成される新しいトピック。
   - **--topic** オプションのトピック名。
   - **--partitions** オプション内のパーティション数。
   - **--replication-factor** オプションのトピックレプリケーション係数。
     また、**--config** オプションを使用して、デフォルトのトピック設定オプションの一部を上書きすることもできます。このオプションは複数回使用して、異なるオプションを上書きすることができます。

```
bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --create --topic <TopicName>
--partitions <NumberOfPartitions> --replication-factor <ReplicationFactor> --config
<Option1>=<Value1> --config <Option2>=<Value2>
```

という名前のトピックを作成するコマンドの例 mytopic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic mytopic --partitions
50 --replication-factor 3 --config cleanup.policy=compact --config min.insync.replicas=2
```

2. **kafka-topics.sh** を使用してトピックが存在することを確認します。

```
bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --describe --topic <TopicName>
```

という名前のトピックを記述するコマンドの例 mytopic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic mytopic
```

関連情報

- トピック設定の詳細は、「トピックの設定」 を参照してください。
- サポートされるすべてのトピック設定オプションの一覧は、付録B *トピック設定パラメーター* を参照してください。

5.8. トピックの一覧表示と説明

kafka-topics.sh ツールを使用してトピックを一覧表示および説明できます。**kafka-topics.sh** AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。
- トピック **mytopic** が存在する。

トピックの記述

1. **kafka-topics.sh** ユーティリティーを使用してトピックを記述し、以下を指定します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポート。
   - **--describe** オプションを使用して、トピックを記述するように指定します。
   - トピック名は **--topic** オプションで指定する必要があります。
   - **--topic** オプションを省略すると、利用可能なすべてのトピックについて説明します。

```
bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --describe --topic
<TopicName>
```

という名前のトピックを記述するコマンドの例 mytopic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic mytopic
```

describe コマンドは、このトピックに属するパーティションおよびレプリカの一覧を表示します。また、トピック設定オプションも表示されます。

関連情報

- トピック設定の詳細は、「トピックの設定」 を参照してください。
- トピックの作成に関する詳細は、「トピックの作成」 を参照してください。

5.9. トピック設定の変更

kafka-configs.sh ツールを使用してトピック設定を変更できます。**kafka-configs.sh** AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。
- トピック **mytopic** が存在する。

トピック設定の変更

1. **kafka-configs.sh** ツールを使用して、現在の設定を取得します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポートを指定します。
   - **--entity-type** を **topic** に、**--entity-name** をトピックの名前に設定します。
   - 現在の設定を取得するには、**--describe** オプションを使用します。

     ```
     bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name <TopicName> --describe
     ```

     名前指定されたトピックの設定を取得するコマンドの例 mytopic

     ```
     bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --describe
     ```

2. **kafka-configs.sh** ツールを使用して設定を変更します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポートを指定します。
   - **--entity-type** を **topic** に、**--entity-name** をトピックの名前に設定します。
   - 現在の設定を変更するには、**--alter** オプションを使用します。
   - オプション **--add-config** に追加または変更するオプションを指定します。

     ```
     bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name <TopicName> --alter --add-config <Option>=<Value>
     ```

     named という名前のトピックの設定を変更するコマンドの例 mytopic

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name
mytopic --alter --add-config min.insync.replicas=1
```

3. **kafka-configs.sh** ツールを使用して、既存の設定オプションを削除します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポートを指定します。
   - **--entity-type** を **topic** に、**--entity-name** をトピックの名前に設定します。
   - 既存の設定オプションを削除するには、**--delete-config** オプションを使用します。
   - オプション **--remove-config** で削除するオプションを指定します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-
name <TopicName> --alter --delete-config <Option>
```

named という名前のトピックの設定を変更するコマンドの例 mytopic

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name
mytopic --alter --delete-config min.insync.replicas
```

関連情報

- トピック設定の詳細は、「トピックの設定」 を参照してください。
- トピックの作成に関する詳細は、「トピックの作成」 を参照してください。
- サポートされるすべてのトピック設定オプションの一覧は、付録B *トピック設定パラメーター* を参照してください。

5.10. トピックの削除

kafka-topics.sh ツールを使用してトピックを管理できます。**kafka-topics.sh** AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。
- トピック **mytopic** が存在する。

トピックの削除

1. **kafka-topics.sh** ユーティリティーを使用してトピックを削除します。
   - **--bootstrap-server** オプションで、Kafka ブローカーのホストおよびポート。
   - **--delete** オプションを使用して、既存のトピックを削除するように指定します。
   - トピック名は **--topic** オプションで指定する必要があります。

```
bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --delete --topic <TopicName>
```

という名前のトピックを作成するコマンドの例 mytopic

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic mytopic
```

2. **kafka-topics.sh** を使用してトピックが削除されていることを確認します。

```
bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --list
```

すべてのトピックを一覧表示するコマンドの例

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

関連情報

- トピックの作成に関する詳細は、「トピックの作成」 を参照してください。

第6章 クライアント設定のチューニング

設定プロパティーを使用して、Kafka プロデューサーおよびコンシューマーのパフォーマンスを最適化します。

最小セットの設定プロパティーが必要ですが、プロパティーを追加または調整して、プロデューサーとコンシューマーが Kafka と対話する方法を変更できます。たとえば、プロデューサーの場合は、クライアントがリアルタイムでデータに応答できるように、メッセージのレイテンシーおよびスループットをチューニングできます。また、設定を変更して、より強力にメッセージの持続性を保証することもできます。

クライアントメトリックを分析して初期設定を行う場所を判断することから始め、必要な設定になるまで段階的に変更を加え、さらに比較を行うことができます。

6.1. KAFKA プロデューサー設定のチューニング

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なプロデューサー設定を使用します。

設定を調整してスループットを最大化すると、レイテンシーが増加する可能性があり、その逆も同様です。必要なバランスを取得するために、プロデューサー設定を実験して調整する必要があります。

6.1.1. 基本のプロデューサー設定

接続およびシリアライザープロパティーはすべてのプロデューサーに必要です。通常、追跡用のクライアント ID を追加し、プロデューサーで圧縮してリクエストのバッチサイズを減らすことが推奨されます。

基本的なプロデューサー設定には以下が含まれます。

- パーティション内のメッセージの順序は保証されません。
- ブローカーに到達するメッセージの完了通知は持続性を保証しません。

```
# ...
bootstrap.servers=localhost:9092 1
key.serializer=org.apache.kafka.common.serialization.StringSerializer 2
value.serializer=org.apache.kafka.common.serialization.StringSerializer 3
client.id=my-client 4
compression.type=gzip 5
# ...
```

1. (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して Kafka クラスターに接続するようプロデューサーを指示します。プロデューサーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して 2 つまたは 3 つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。
2. (必須) メッセージがブローカーに送信される前に、各メッセージの鍵をバイトに変換するシリアライザー。
3. (必須) メッセージがブローカーに送信される前に、各メッセージの値をバイトに変換するシリアライザー。

4 (任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで使用されます。

5 (任意) メッセージを圧縮するコーデック。これは、送信され、圧縮された形式で格納された後、コンシューマーへの到達時に圧縮解除される可能性があります。圧縮はスループットを改善し、ストレージの負荷を減らすのに役立ちますが、圧縮や圧縮解除のコストが異常に高い低レイテンシーのアプリケーションには不適切である場合があります。

6.1.2. データの持続性

メッセージ配信の完了通知を使用して、データの持続性を適用し、メッセージが失われる可能性を最小限に抑えることができます。

```
# ...
acks=all 1
# ...
```

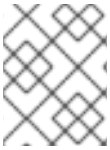
1 **acks=all** と指定すると、パーティションリーダーは、メッセージリクエストが正常に受信されたことを確認する前に、特定数のフォロワーに対してメッセージをレプリケートすることを強制されます。**acks=all** の追加のチェックにより、プルデューサーがメッセージを送信してから完了通知を受信するまでのレイテンシーが増加します。

完了通知がプロデューサーに送信される前にメッセージをログに追加する必要のあるブローカーの数は、トピックの **min.insync.replicas** 設定によって決定されます。最初に、トピックレプリケーション係数を 3 にし、他のブローカーの In-Sync レプリカを 2 にするのが一般的です。この設定では、単一のブローカーが利用できない場合でもプロデューサーは影響を受けません。2 番目のブローカーが利用できなくなると、プロデューサーは完了通知を受信せず、それ以上のメッセージを生成できなくなります。

acks=all をサポートするトピック設定

```
# ...
min.insync.replicas=2 1
# ...
```

1 **2** In-Sync レプリカを使用します。デフォルトは **1** です。



注記

システムに障害が発生すると、バッファーの未送信データが失われる可能性があります。

6.1.3. 順序付き配信

メッセージは 1 度だけ配信されるため、べき等プロデューサーは重複を回避します。障害発生時でも配信の順序が維持されるように、ID とシーケンス番号がメッセージに割り当てられます。データの一貫性を維持するために **acks=all** を使用している場合は、順序付き配信にべき等を有効にするのは妥当です。

べき等を使った順序付き配信

```
# ...
enable.idempotence=true (1)
max.in.flight.requests.per.connection=5 (2)
acks=all (3)
retries=2147483647 (4)
# ...
```

(1) **true** を設定してべき等プロデューサーを有効にします。

(2) べき等配信では、インフライトリクエストの数が 1 を越えることがありますがメッセージの順序は維持されます。デフォルトのインフライトリクエストの数は 5 です。

(3) **acks** を **all** に設定します。

(4) 失敗したメッセージリクエストを再送信する試行回数を設定します。

パフォーマンスコストが原因で **acks=all** およびべき等を使用しない場合は、インフライト (完了確認されない) リクエストの数を 1 に設定して、順序を保持します。そうしないと、**Message-A** が失敗し、**Message-B** がブローカーに書き込まれた後にのみ成功する可能性があります。

べき等を使用しない順序付け配信

```
# ...
enable.idempotence=false (1)
max.in.flight.requests.per.connection=1 (2)
retries=2147483647
# ...
```

(1) **false** を設定して、べき等プロデューサーを無効にします。

(2) インフライトリクエストの数を確実に **1** に設定します。

6.1.4. 信頼性の保証

べき等は、1 つのパーティションへの書き込みを 1 回だけ行う場合に便利です。トランザクションをべき等と使用すると、複数のパーティション全体で 1 度だけ書き込みを行うことができます。

トランザクションは、同じトランザクション ID を使用するメッセージが 1 度作成され、**すべて**がそれぞれのログに書き込まれるか、**何も**書き込まれないかのどちらかになることを保証します。

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

(1) 一意のトランザクション ID を指定します。

(2) タイムアウトエラーが返されるまでのトランザクションの最大許容時間 (ミリ秒単位) を設定しま

す。デフォルトは **900000** (15 分) です。

トランザクションの保証を維持するには、**transactional.id** の選択が重要になります。トランザクション ID は、一意なトピックパーティションセットに使用する必要があります。たとえば、トピックパーティション名からトランザクション ID への外部マッピングを使用したり、競合を回避する関数を使用してトピックパーティション名からトランザクション IDを算出したりすると、これを実現できます。

6.1.5. スループットおよびレイテンシーの最適化

通常、システムの要件は、指定のレイテンシー内であるメッセージの割合に対して、特定のスループットのターゲットを達成することです。たとえば、95 % のメッセージが 2 秒以内に完了確認される、1 秒あたり 500,000 個のメッセージをターゲットとします。

プロデューサーのメッセージングセマンティック (メッセージの順序付けと持続性) は、アプリケーションの要件によって定義される可能性があります。たとえば、アプリケーションによって提供される重要なプロパティーや保証に反することなく、**acks=0** または **acks=1** を使用するオプションがない可能性があります。

ブローカーの再起動は、パーセンタイルの高いの統計に大きく影響します。たとえば、長期間では、99% のレイテンシーはブローカーの再起動に関する動作によるものです。これは、ベンチマークを設計したり、本番環境のパフォーマンスで得られた数字を使ってベンチマークを行い、そのパフォーマンスの数字を比較したりする場合に検討する価値があります。

目的に応じて、Kafka はスループットとレイテンシーのプロデューサーパフォーマンスを調整するために多くの設定パラメーターと設定方法を提供します。

メッセージのバッチ処理 (linger.ms および batch.size)

メッセージのバッチ処理では、同じブローカー宛のメッセージをより多く送信するために、メッセージの送信を遅らせ、単一の生成リクエストでバッチ処理できるようにします。バッチ処理では、スループットを増やすためにレイテンシーを長くして妥協します。時間ベースのバッチ処理は **linger.ms** を使用して設定され、サイズベースのバッチ処理は **batch.size** を使用して設定されます。

圧縮処理 (compression.type)

メッセージ圧縮処理により、プロデューサー (メッセージの圧縮に費やされた CPU 時間) のレイテンシーが追加されますが、リクエスト (および場合によってはディスクの書き込み) を小さくするため、スループットが増加します。圧縮に価値があるかどうか、および使用に最適な圧縮は、送信されるメッセージによって異なります。圧縮処理は **KafkaProducer.send()** を呼び出すスレッドで発生するため、アプリケーションでこの方法のレイテンシーが問題になる場合は、より多くのスレッドを使用するよう検討してください。

パイプライン処理 (max.in.flight.requests.per.connection)

パイプライン処理は、以前のリクエストへの応答を受け取る前により多くのリクエストを送信します。通常、パイプライン処理を増やすと、バッチ処理の悪化などの別の問題がスループットに悪影響を与え始めるしきい値まではスループットが増加します。

レイテンシーの短縮

アプリケーションが **KafkaProducer.send()** を呼び出す場合、メッセージには以下が行われます。

- インターセプターによる処理。
- シリアライズ。
- パーティションへの割り当て。
- 圧縮処理。

- パーティションごとのキューでメッセージのバッチに追加。

ここで、**send()** メソッドが返されます。そのため、**send()** がブロックされる時間は、以下によって決定されます。

- インターセプター、シリアライザー、およびパーティションヤーで費やされた時間。
- 使用される圧縮アルゴリズム。
- 圧縮に使用するバッファーの待機に費やされた時間。

バッチは、以下のいずれかが行われるまでキューに残ります。

- バッチが満杯になる (**batch.size** による)。
- **linger.ms** によって導入された遅延が経過。
- 送信者は他のパーティションのメッセージバッチを同じブローカーに送信しようとし、このバッチの追加も可能。
- プロデューサーがフラッシュまたは閉じられる。

バッチ処理とバッファーの設定を参照して、レイテンシーをブロックする **send()** の影響を軽減します。

```
# ...
linger.ms=100 1
batch.size=16384 2
buffer.memory=33554432 3
# ...
```

1 **linger** プロパティーは、メッセージの大きなバッチが累積され、リクエストで送信されるように、ミリ秒単位の遅延を追加します。デフォルトは **0'.** です。

2 最大 **batch.size** (バイト単位) が使用された場合、その最大値に達したとき、またはメッセージが **linger.ms** よりも長い期間キューに置かれたとき (いずれか早く発生した方) にリクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

3 バッファーサイズは、少なくともバッチサイズと同じ大きさである必要があり、バッファー、圧縮、およびインフライトリクエストに対応できる必要があります

スループットの増加

メッセージの配信および送信リクエストの完了までの最大待機時間を調整して、メッセージリクエストのスループットを向上します。

また、カスタムパーティションを作成してデフォルトを置き換えることで、メッセージを指定のパーティションに転送することもできます。

```
# ...
delivery.timeout.ms=120000 1
partitioner.class=my-custom-partitioner 2

# ...
```

1 送信リクエストの完了まで待機する最大時間 (ミリ秒単位)。この値を **MAX_LONG** に設定すると、Kafka に回数無制限の再試行を委譲できます。デフォルトは **120000** (2 分) です。

2 カスタムパーティショナーのクラス名を指定します。

6.2. KAFKA コンシューマー設定の調整

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なコンシューマー設定を使用します。

コンシューマーを調整する場合、最も重要なことは、取得するデータ量に効率的に対処できるようにすることです。プロデューサーのチューニングと同様に、コンシューマーが想定どおりに動作するまで、段階的に変更を加える必要があります。

6.2.1. 基本的なコンシューマー設定

接続およびデシリアライザープロパティーはすべてのコンシューマーに必要です。通常、追跡用にクライアント ID を追加することが推奨されます。

コンシューマー設定では、後続の設定に関係なく、以下を行います。

- メッセージをスキップまたは再読み取りするようオフセットを変更しない限り、コンシューマーはメッセージを指定のオフセットから取得し、順番に消費します。
- オフセットはクラスターの別のブローカーに送信される可能性があるため、オフセットを Kafka にコミットした場合でも、ブローカーはコンシューマーが応答を処理したかどうかを認識しません。

```
# ...
bootstrap.servers=localhost:9092 1
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer 2
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer 3
client.id=my-client 4
group.id=my-group-id 5
# ...
```

1 (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して、コンシューマーが Kafka クラスターに接続するよう指示しますコンシューマーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して 2 つまたは 3 つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。ロードバランサーサービスを使用して Kafka クラスターを公開する場合、可用性はロードバランサーによって処理されるため、サービスのアドレスのみが必要になります。

2 (必須) Kafka ブローカーから取得されたバイトをメッセージキーに変換するデシリアライザー。

3 (必須) Kafka ブローカーから取得されたバイトをメッセージ値に変換するデシリアライザー。

4 (任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで使用されます。ID は、時間クォータの処理に基づいてコンシューマーにスロットリングを適用するために使用することもできます。

5 (条件) コンシューマーがコンシューマーグループに参加するには、グループ ID が **必要** です。

コンシューマーグループは、特定のトピックから複数のプロデューサーによって生成される、典型的に大量のデータストリームを共有するのに使用します。コンシューマーは **group.id** でグループ化され、メッセージをメンバー全体に分散できます。

6.2.2. コンシューマーグループを使用したデータ消費のスケーリング

コンシューマーグループは、特定のトピックから1つまたは複数のプロデューサーによって生成される、典型的な大量のデータストリームを共有します。**group.id** プロパティーが同じコンシューマーは同じグループになります。グループ内のコンシューマーの1つがリーダーを選択し、パーティションをグループのコンシューマーにどのように割り当てるかを決定します。各パーティションは1つのコンシューマーにのみ割り当てることができます。

コンシューマーの数がパーティションよりも少ない場合、同じ **group.id** を持つコンシューマーインスタンスを追加して、データの消費をスケーリングできます。コンシューマーをグループに追加して、パーティションの数より多くしても、スループットは改善されませんが、コンシューマーが機能しなくなったときに予備のコンシューマーを使用できます。より少ないコンシューマーでスループットの目標を達成できれば、リソースを節約できます。

同じコンシューマーグループのコンシューマーは、オフセットコミットとハートビートを同じブローカーに送信します。グループのコンシューマーの数が多いほど、ブローカーのリクエスト負荷が高くなります。

```
# ...
group.id=my-group-id 1
# ...
```

1 グループ ID を使用してコンシューマーグループにコンシューマーを追加します。

6.2.3. メッセージの順序の保証

Kafka ブローカーは、トピック、パーティション、およびオフセット位置のリストからメッセージを送信するようブローカーに要求するコンシューマーからフェッチリクエストを受け取ります。

コンシューマーは、ブローカーにコミットされたのと同じ順序でメッセージを単一のパーティションで監視します。つまり、Kafka は単一パーティションのメッセージ ***のみ*** 順序付けを保証します。逆に、コンシューマーが複数のパーティションからメッセージを消費している場合、コンシューマーによって監視される異なるパーティションのメッセージの順序は、必ずしも送信順序を反映しません。

1つのトピックからメッセージを厳格に順序付ける場合は、コンシューマーごとに1つのパーティションを使用します。

6.2.4. スループットおよびレイテンシーの最適化

クライアントアプリケーションが **KafkaConsumer.poll()** を呼び出すときに返されるメッセージの数を制御します。

fetch.max.wait.ms および **fetch.min.bytes** プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最小量を増やします。時間ベースのバッチ処理は **fetch.max.wait.ms** を使用して設定され、サイズベースのバッチ処理は **fetch.min.bytes** を使用して設定されます。

コンシューマーまたはブローカーの CPU 使用率が高い場合、コンシューマーからのリクエストが多すぎる可能性があります。リクエストの数を減らし、メッセージがより大きなバッチで配信されるように、**fetch.max.wait.ms** および **fetch.min.bytes** プロパティーを調整します。より高い値に調整するこ

とでスループットが改善されますが、レイテンシーのコストが発生します。生成されるデータ量が少ない場合、より高い値に調整することもできます。

たとえば、**fetch.max.wait.ms** を 500ms に設定し、**fetch.min.bytes** を 16384 バイトに設定した場合、Kafka がコンシューマーからフェッチリクエストを受信すると、いずれかのしきい値に最初に到達した時点で応答されます。

逆に、**fetch.max.wait.ms** および **fetch.min.bytes** プロパティーを低く設定すると、エンドツーエンドのレイテンシーを改善できます。

```
# ...
fetch.max.wait.ms=500 1
fetch.min.bytes=16384 2
# ...
```

1 ブローカーがフェッチリクエストを完了するまで待機する最大時間 (ミリ秒単位)。デフォルトは **500** ミリ秒です。

2 最小バッチサイズ (バイト単位) が使用された場合、その最小値に達したとき、またはメッセージが **fetch.max.wait.ms** よりも長い期間キューに置かれたとき (いずれか早く発生した方) にリクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

フェッチリクエストサイズの増加によるレイテンシーの短縮

fetch.max.bytes および **max.partition.fetch.bytes** プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最大量を増やします。

fetch.max.bytes プロパティーは、一度にブローカーから取得されるデータ量の上限をバイト単位で設定します。

max.partition.fetch.bytes は、各パーティションに返されるデータ量の上限をバイト単位で設定します。これは、常に **max.message.bytes** のブローカーまたはトピック設定に設定されたバイト数よりも大きくする必要があります。

クライアントが消費できるメモリーの最大量は、以下のように概算されます。

> **NUMBER-OF-BROKERS** * fetch.max.bytes and **NUMBER-OF-PARTITIONS** * max.partition.fetch.bytes

メモリー使用量がこれに対応できる場合は、これら 2 つのプロパティーの値を増やすことができます。各リクエストでより多くのデータを許可すると、フェッチリクエストが少なくなるため、レイテンシーが向上されます。

```
# ...
fetch.max.bytes=52428800 1
max.partition.fetch.bytes=1048576 2
# ...
```

1 フェッチリクエストに対して返されるデータの最大量 (バイト単位)。

2 各パーティションに対して返されるデータの最大量 (バイト単位)。

6.2.5. オフセットをコミットする際のデータ損失または重複の回避

Kafka の **自動コミットメカニズム** により、コンシューマーはメッセージのオフセットを自動的にコミットできます。有効にすると、コンシューマーはブローカーをポーリングして受信したオフセットを 5000ms 間隔でコミットします。

自動コミットのメカニズムは便利ですが、データ損失と重複のリスクが発生します。コンシューマーが多くのメッセージを取得および変換し、自動コミットの実行時にコンシューマーバッファーに処理されたメッセージがある状態でシステムがクラッシュすると、そのデータは失われます。メッセージの処理後、自動コミットの実行前にシステムがクラッシュした場合、リバランス後に別のコンシューマーインスタンスでデータが複製されます。

ブローカーへの次のポーリングの前またはコンシューマーが閉じられる前に、すべてのメッセージが処理された場合は、自動コミットによるデータの損失を回避できます。

データ損失や重複の可能性を最小限にするには、**enable.auto.commit** を **false** に設定し、クライアントアプリケーションを開発して、オフセットのコミットをさらに制御します。または、**auto.commit.interval.ms** を使用して、コミットの間隔を減らすことができます。

```
# ...
enable.auto.commit=false (1)
# ...
```

1 自動コミットを false に設定すると、オフセットのコミットの制御が強化されます。

enable.auto.commit を **false** に設定すると、**すべて** の処理が実行され、メッセージが消費された後にオフセットをコミットできます。たとえば、Kafka **commitSync** および **commitAsync** コミット API を呼び出すようにアプリケーションを設定できます。

commitSync API は、ポーリングから返されるメッセージバッチのオフセットをコミットします。バッチのメッセージすべての処理が完了したら API を呼び出します。**commitSync** API を使用する場合、アプリケーションはバッチの最後のオフセットがコミットされるまで新しいメッセージをポーリングしません。これがスループットに悪影響する場合は、コミットする頻度を減らすか、**commitAsync** API を使用できます。**commitAsync** API はブローカーがコミットリクエストに応答するまで待機しませんが、リバランス時にさらに重複が発生するリスクがあります。一般的なアプローチとして、両方のコミット API をアプリケーションで組み合わせ、コンシューマーをシャットダウンまたはリバランスの直前に **commitSync** API を使用し、最終コミットが正常に実行されるようにします。

6.2.5.1. トランザクションメッセージの制御

プロデューサー側でトランザクション ID を使用し、べき等 (**enable.idempotence=true**) を有効にして、1 回のみの配信の保証を検討してください。コンシューマー側で、**isolation.level** プロパティーを使用して、コンシューマーによってトランザクションメッセージが読み取られる方法を制御できます。

isolation.level プロパティーに有効な値は 2 つあります。

- **read_committed**
- **read_uncommitted** (デフォルト)

コミットされたトランザクションメッセージのみがコンシューマーによって読み取られるようにするには、**read_committed** を使用します。ただし、これによりトランザクションの結果を記録するトランザクションマーカー (**committed** または **aborted**) がブローカーによって書き込まれるまで、コンシューマーはメッセージを返すことができないため、エンドツーエンドのレイテンシーが長くなります。

```
# ...
enable.auto.commit=false
```

```
isolation.level=read_committed (1)
# ...
```

(1) コミットされたメッセージのみがコンシューマーによって読み取られるように、**read_committed** に設定します。

6.2.6. データ損失を回避するための障害からの復旧

session.timeout.ms および **heartbeat.interval.ms** プロパティーを使用して、コンシューマーグループ内のコンシューマー障害をチェックし、復旧するのにかかる時間を設定します。

session.timeout.ms プロパティーは、コンシューマーグループのコンシュマーが非アクティブであるとみなされ、そのグループのアクティブなコンシューマー間で**リバランス**がトリガーされる前に、ブローカーと通信できない最大時間をミリ秒単位で指定します。グループのリバランス時に、パーティションはグループのメンバーに再割り当てされます。

heartbeat.interval.ms プロパティーは、コンシューマーがアクティブで接続されていることを示す、コンシューマーグループコーディネーターへの**ハートビート**チェックの間隔をミリ秒単位で指定します。通常、ハートビートの間隔はセッションタイムアウトの間隔の 3 分の 2 にする必要があります。

session.timeout.ms プロパティーの値を低く設定すると、失敗するコンシューマーが早期に発見され、リバランスがより迅速に実行されます。ただし、タイムアウトの値を低くしすぎて、ブローカーがハートビートを時間内に受信できず、不必要なリバランスがトリガーされることがないように気を付けてください。

ハートビートの間隔が短くなると、誤ってリバランスを行う可能性が低くなりますが、ハートビートを頻繁に行うとブローカーリソースのオーバーヘッドが増えます。

6.2.7. オフセットポリシーの管理

auto.offset.reset プロパティーを使用して、オフセットをすべてコミットしなかった場合やコミットされたオフセットが有効でないまたは削除された場合の、コンシューマーの動作を制御します。

コンシューマーアプリケーションを初めてデプロイし、既存のトピックからメッセージを読み取る場合について考えてみましょう。**group.id** が初めて使用されるため、**__consumer_offsets** トピックには、このアプリケーションのオフセット情報は含まれません。新しいアプリケーションは、ログの始めからすべての既存メッセージの処理を開始するか、新しいメッセージのみ処理を開始できます。デフォルトのリセット値は、パーティションの最後から開始する **latest** で、一部のメッセージは見逃されることを意味します。データの損失を回避し、処理量を増やすには、**auto.offset.reset** を **earliest** に設定し、パーティションの最初から開始します。

また、ブローカーに設定されたオフセットの保持期間 (**offsets.retention.minutes**) が終了したときにメッセージが失われないようにするため、**earliest** オプションを使用することも検討してください。コンシューマーグループまたはスタンドアロンコンシューマーが非アクティブで、保持期間中にオフセットをコミットしない場合、以前にコミットされたオフセットは **__consumer_offsets** から削除されます。

```
# ...
heartbeat.interval.ms=3000 (1)
session.timeout.ms=10000 (2)
auto.offset.reset=earliest (3)
# ...
```

(1) 予想されるリバランスに応じて、ハートビートの間隔を短くして調整します。

2 タイムアウトの期限が切れる前に Kafka ブローカーによってハートビートが受信されなかった場合、コンシューマーはコンシューマーグループから削除され、リバランスが開始されます。ブロー

3 パーティションの最初に戻り、オフセットがコミットされなかった場合にデータの損失が発生しないようにするには、**earliest** に設定します。

1 つのフェッチリクエストで返されるデータ量が大きい場合、コンシューマーが処理する前にタイムアウトが発生することがあります。この場合は、**max.partition.fetch.bytes** の値を低くするか、**session.timeout.ms** の値を高くします。

6.2.8. リバランスの影響を最小限にする

グループのアクティブなコンシューマー間で行うパーティションのリバランスは、以下にかかる時間です。

- コンシューマーによるオフセットのコミット
- 作成される新しいコンシューマーグループ
- グループリーダーによるグループメンバーへのパーティションの割り当て。
- 割り当てを受け取り、取得を開始するグループのコンシューマー

明らかに、このプロセスは特にコンシューマーグループクラスターのローリング再起動時に繰り返し発生するサービスのダウンタイムを増やします。

このような場合、**静的メンバーシップ** の概念を使用してリバランスの数を減らすことができます。リバランスによって、コンシューマーグループメンバー全体でトピックパーティションが割り当てられます。静的メンバーシップは永続性を使用し、セッションタイムアウト後の再起動時にコンシューマーインスタンスが認識されるようにします。

コンシューマーグループコーディネーターは、**group.instance.id** プロパティーを使用して指定される一意の ID を使用して新しいコンシューマーインスタンスを特定できます。再起動時には、コンシューマーには新しいメンバー ID が割り当てられますが、静的メンバーとして、同じインスタンス ID を使用し、同じトピックパーティションの割り当てが行われます。

コンシューマーアプリケーションが最低でも **max.poll.interval.ms** ミリ秒毎にポーリングへの呼び出しを行わない場合、コンシューマーは失敗したと見なされ、リバランスが発生します。アプリケーションがポーリングから返されたすべてレコードを時間内に処理できない場合は、**max.poll.interval.ms** プロパティーを使用して、コンシューマーからの新規メッセージのポーリングの間隔をミリ秒単位で指定して、リバランスの発生を防ぎます。または、**max.poll.records** プロパティーを使用して、コンシューマーバッファーから返されるレコードの数の上限を設定し、アプリケーションが **max.poll.interval.ms** 内でより少ないレコードを処理できるようにします。

```
# ...
group.instance.id=UNIQUE-ID 1
max.poll.interval.ms=300000 2
max.poll.records=500 3
# ...
```

1 一意のインスタンス ID により、新しいコンシューマーインスタンスに同じトピックパーティションが割り当てられます。

2 コンシューマーがメッセージの処理を継続していることを確認する間隔を設定します。

3 コンシューマーから返される処理済のレコードの数を設定します。

第7章 クラスターのスケーリング

7.1. KAFKA クラスターのスケーリング

7.1.1. ブローカーのクラスターへの追加

トピックのスループットを向上させる主な方法は、そのトピックのパーティション数を増やすことです。これにより、パーティションによってそのトピックの負荷がクラスター内のブローカー間で共有されるためです。ブローカーがすべてリソース（通常は I/O）によって制約される場合、パーティションを増やすとスループットは向上しません。代わりに、ブローカーをクラスターに追加する必要があります。

追加のブローカーをクラスターに追加する場合、AMQ Streams ではパーティションは自動的に割り当てられません。既存のブローカーから新しいブローカーに移動するパーティションを決定する必要があります。

すべてのブローカー間でパーティションが再分散されたら、各ブローカーにリソースの使用率が低くなるはずです。

7.1.2. クラスターからのブローカーの削除

クラスターからブローカーを削除する前に、そのブローカーにパーティションが割り当てられていないことを確認する必要があります。の使用を停止するブローカーの各パーティションに対応する残りのブローカーを決める必要があります。ブローカーに割り当てられたパーティションがない場合は、これを停止できます。

7.2. パーティションの再割り当て

kafka-reassign-partitions.sh ユーティリティーは、パーティションを異なるブローカーに再割り当てするために使用されます。

これには、以下の 3 つのモードがあります。

--generate

トピックとブローカーのセットを取り、**再割り当て JSON ファイル** を生成します。これにより、トピックのパーティションがブローカーに割り当てられます。**再割り当て JSON ファイル** を生成する簡単な方法ですが、トピック全体で動作するため、使用は常に適しているとは限りません。

--execute

再割り当て JSON ファイル を取り、クラスターのパーティションおよびブローカーに適用します。パーティションを取得するブローカーは、パーティションリーダーのフォロワーになります。特定のパーティションでは、新しいブローカーが ISR を検出し、加わると、古いブローカーがフォロワーではなくなり、そのレプリカが削除されます。

--verify

--verify は、**--execute** ステップと同じ **再割り当て JSON ファイル** を使用して、ファイル内のすべてのパーティションが目的のブローカーに移動されたかどうかを確認します。再割り当てが完了すると、有効な スロットル も削除されます。スロットルを削除しないと、再割り当てが完了した後もクラスターは影響を受け続けます。

クラスターでは、1 度に 1 つの再割り当てのみを実行でき、実行中の再割り当てをキャンセルすることはできません。再割り当てをキャンセルする必要がある場合は、実行して別の再割り当てを実行し、最初の再割り当ての結果を元に戻す必要があります。**kafka-reassign-partitions.sh** によって、元に戻す

ための再割り当て JSON が出力の一部として生成されます。大規模な再割り当ては、進行中の再割り当てを停止する必要がある場合に備えて、複数の小さな再割り当てに分割するようにしてください。

7.2.1. 再割り当て JSON ファイル

再割り当て JSON ファイル には特定の構造があります。

```
{
  "version": 1,
  "partitions": [
    <PartitionObjects>
  ]
}
```

ここで <PartitionObjects> は、以下のようなコンマ区切りのオブジェクトリストになります。

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ],
  "log_dirs": [<LogDirs>]
}
```

"log_dirs" プロパティーはオプションです。パーティションを特定のログディレクトリーに移動します。

以下は、トピック **topic-a** およびパーティション **4** をブローカー **2**、**4**、および **7** に割り当て、トピック **topic-b** およびパーティション **2** をブローカー **1**、**5**、および **7** に割り当てる、再割り当て JSON ファイルの例になります。

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7]
    },
    {
      "topic": "topic-b",
      "partition": 2,
      "replicas": [1,5,7]
    }
  ]
}
```

JSON に含まれていないパーティションは変更されません。

7.2.2. 再割り当て JSON ファイルの生成

指定されたトピックセットのすべてのパーティションを指定のブローカーセットに割り当てる最も簡単な方法は、**kafka-reassign-partitions.sh --generate**, コマンドを使用して再割り当て JSON ファイルを生成します。

```
bin/kafka-reassign-partitions.sh --zookeeper <ZooKeeper> --topics-to-move-json-file <TopicsFile> --broker-list <BrokerList> --generate
```

<TopicsFile> は、移動するトピックを一覧表示する JSON ファイルです。これには、以下の構造が含まれます。

```
{
  "version": 1,
  "topics": [
    <TopicObjects>
  ]
}
```

ここで <TopicObjects> は、以下のようなコンマ区切りのオブジェクトリストになります。

```
{
  "topic": <TopicName>
}
```

たとえば、**topic-a** および **topic-b** のすべてのパーティションをブローカー **4** に移動する場合、 **7**

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-be-moved.json --broker-list 4,7 --generate
```

ここで、**topics-to-be-moved.json** の内容があります。

```
{
  "version": 1,
  "topics": [
    { "topic": "topic-a"},
    { "topic": "topic-b"}
  ]
}
```

7.2.3. 手動による再割り当て JSON ファイルの作成

特定のパーティションを移動したい場合は、再割り当て JSON ファイルを手動で作成できます。

7.3. 再割り当てスロットル

パーティションを再割り当てすると、ブローカーの間で大量のデータを移動する必要があるため、処理が遅くなる可能性があります。クライアントに悪影響を与えることを回避するため、再割り当てを **スロットル** で調整することが可能です。スロットルを使用すると、再割り当てに時間がかかることを意味します。スロットルが低すぎると、新たに割り当てられたブローカーは公開されるレコードに遅れずに対応することはできず、再割り当ては永久に完了しません。スロットルが高すぎると、クライアントに影響します。たとえば、プロデューサーの場合は、承認待ちが通常のレイテンシーよりも大きくなる可能性があります。コンシューマーの場合は、ポーリング間のレイテンシーが大きいことが原因でスループットが低下する可能性があります。

7.4. KAFKA クラスターのスケールアップ

この手順では、Kafka クラスターでブローカーの数を増やす方法を説明します。

前提条件

- 既存の Kafka クラスター。
- AMQ ブローカーが インストールされ た新しいマシン。
- 拡大されたクラスターでパーティションをブローカーに **再割り当てする方法の再割り当て JSON ファイル**。

手順

1. クラスターの他のブローカーと同じ設定を使用して、新しいブローカーの設定ファイルを作成します。ただし、**broker.id** は他のブローカーによって使用されていない数字である必要があります。
2. 前のステップで作成した設定ファイルを引数として **kafka-server-start.sh** スクリプトに渡す新しい Kafka ブローカーを起動します。

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. Kafka ブローカーが稼働していることを確認します。

```
jcmd | grep Kafka
```

4. 新しいブローカーごとに上記の手順を繰り返します。
5. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、パーティションの再割り当てを実行します。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** とブローカー間のスロットル率 (バイト/秒単位) を渡すこともできます。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 5000000 --execute
```

このコマンドは、2 つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備えて、これをファイルに保存する必要があります。2 つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡した目的の再割り当てです。

6. 再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドラインに別のスロットル率を指定して実行します。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 10000000 --execute
```

7. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したかどうかを定期的に確認します。これは先ほどの手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --verify
```

以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --verify
```

8. **--verify** コマンドによって、移動した各パーティションが正常に完了したことが報告されると、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。

7.5. KAFKA クラスターのスケールダウン

その他のリソース

この手順では、Kafka クラスターでブローカーの数を減らす方法を説明します。

前提条件

- 既存の Kafka クラスター。
- ブローカーが削除された後にクラスター内のブローカーにパーティションを **再割り当てする方法の再割り当て JSON ファイル**。

手順

1. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、パーティションの再割り当てを実行します。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** とブローカー間のスロットル率 (バイト/秒単位) を渡すこともできます。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 5000000 --execute
```

このコマンドは、2 つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備えて、これをファイルに保存する必要があります。2 つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡した目的の再割り当てです。

2. 再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドラインに別のスロットル率を指定して実行します。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 10000000 --execute
```

3. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したかどうかを定期的に確認します。これは先ほどの手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --verify
```

以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --verify
```

4. **--verify** コマンドによって、移動した各パーティションが正常に完了したことが報告されると、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。

5. すべてのパーティションの再割り当てが終了すると、削除されるブローカーはクラスター内のいずれのパーティションにも対応しないはずです。これは、ブローカーの **log.dirs** 設定パラメーターに指定された各ディレクトリーを確認して検証できます。ブローカーのいずれかのログディレクトリーに、拡張正規表現 **[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$** に一致しないディレクトリーが含まれる場合、ブローカーにライブパーティションがあるため、停止してはなりません。
これを確認するには、以下のコマンドを実行します。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

上記のコマンドによって出力が生成される場合、ブローカーにはライブパーティションがあります。この場合、再割り当てが終了していないか、再割り当て JSON ファイルが適切ではありません。

6. ブローカーにライブパーティションがないことが確認できたら、停止できます。

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

7. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

7.6. ZOOKEEPER クラスターのスケールアップ

この手順では、サーバー（ノード）を ZooKeeper クラスターに追加する方法を説明します。
ZooKeeper の 動的再設定 機能は、スケールアッププロセス中に ZooKeeper クラスターの安定性を維持します。

前提条件

- 動的再設定は、ZooKeeper 設定ファイル(**reconfigEnabled=true**)で有効化されます。
- ZooKeeper 認証が有効になり、認証メカニズムを使用して新しいサーバーにアクセスできます。

手順

各 ZooKeeper サーバーに対して、1 つずつ以下の手順を実行します。

1. 「マルチノードの ZooKeeper クラスターの実行」 の説明に従い、サーバーを ZooKeeper クラスターに追加し、ZooKeeper を起動します。
2. IP アドレスと、新しいサーバーのアクセスポートを設定することに注意してください。
3. サーバーの **zookeeper-shell** セッションを開始します。クラスターにアクセスできるマシンから以下のコマンドを実行します（これは、ZooKeeper ノードまたはローカルマシンのいずれかにアクセスできる場合があります）。

```
su - kafka
/opt/kafka/bin/zookeeper-shell.sh <ip-address>:<zk-port>
```

4. ZooKeeper ノードが稼働しているシェルセッションで以下の行を入力し、新しいサーバーを投票メンバーとしてクォーラムに追加します。

```
reconfig -add server.<positive-id> = <address1>:<port1>:<port2>[:role];[<client-port-address>:]<client-port>
```

以下に例を示します。

```
reconfig -add server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
```

ここで、**<positive-id>** は新しいサーバー ID **4** です。

2 つのポートでは、**<port1> 2888** は ZooKeeper サーバー間の通信に、 **<port2>** 3888 **は** リーダー選択に使用されます。

新しい設定は ZooKeeper クラスターの他のサーバーに伝播されます。新しいサーバーはクォーラムの完全メンバーになります。

5. 追加する他のサーバーで、ステップ 1-4 を繰り返します。

関連情報

- 「ZooKeeper クラスターのスケールダウン」

7.7. ZOOKEEPER クラスターのスケールダウン

この手順では、ZooKeeper クラスターからサーバー（ノード）を削除する方法を説明します。
ZooKeeper の 動的再設定 機能は、スケールダウンプロセス中に ZooKeeper クラスターの安定性を維持します。

前提条件

- 動的再設定は、ZooKeeper 設定ファイル(**reconfigEnabled=true**)で有効化されます。
- ZooKeeper 認証が有効になり、認証メカニズムを使用して新しいサーバーにアクセスできます。

手順

各 ZooKeeper サーバーに対して、1 つずつ以下の手順を実行します。

1. スケールダウン後に **保持** されるサーバー（例： サーバー 1）で **zookeeper-shell** にログインします。

注記

ZooKeeper クラスターに設定された認証メカニズムを使用して、サーバーにアクセスします。

2. サーバーを削除します（例： server 5）。

```
reconfig -remove 5
```

3. 削除したサーバーを無効化します。
4. 手順 1-3 を繰り返し、クラスターサイズを縮小します。

関連情報

- 「ZooKeeper クラスターのスケールアップ」
- ZooKeeper ドキュメントのサーバーの削除

第8章 JMX を使用したクラスターの監視

ZooKeeper、Kafka ブローカー、Kafka Connect、および Kafka クライアントはすべて Java Management Extensions (JMX)を使用して管理情報を公開します。ほとんどの管理情報は、Kafka クラスターの条件およびパフォーマンスの監視に役立つメトリクスの形式で使用されます。他の Java アプリケーションと同様に、Kafka は管理対象 Bean または MBean を使用してこの管理情報を提供します。

JMX は、JVM（Java 仮想マシン）レベルで機能します。管理情報を取得するには、外部ツールは ZooKeeper、Kafka ブローカーなどを実行している JVM に接続できます。デフォルトでは、同じマシン上でツールのみが接続でき、JVM と同じユーザーとしてのみ接続可能です。

注記

ZooKeeper の管理情報はこちらに記載されています。JConsole で ZooKeeper メトリクスを表示できます。詳細は、「JConsole を使用した監視」を参照して ください。

8.1. JMX 設定オプション

JVM システムプロパティーを使用して JMX を設定します。AMQ Streams（**bin/kafka-server-start.sh** および **bin/connect-distributed.sh** など）で提供されるスクリプトは、**KAFKA_JMX_OPTS** 環境変数を使用してこれらのシステムプロパティーを設定します。通常、Kafka プロデューサー、コンシューマー、およびストリームアプリケーションは通常、異なる方法で JVM を起動することも、JMX を設定するシステムプロパティーは同じです。

8.2. JMX エージェントの無効化

AMQ Streams コンポーネントの JMX エージェントを無効にすることによって、ローカル JMX ツールが JVM に接続できない（コンプライアンス上の理由など）を防ぐことができます。以下の手順では、Kafka ブローカーの JMX エージェントを無効にする方法を説明します。

手順

1. **KAFKA_JMX_OPTS** 環境変数を使用して、**com.sun.management.jmxremote** を **false** に設定します。

```
export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=false
bin/kafka-server-start.sh
```

2. JVM を起動します。

8.3. 別のマシンからの JVM への接続

JMX エージェントがリッスンするポートを設定すると、別のマシンから JVM に接続できます。これは、JMX ツールをどこからも認証なしで接続できるため、安全ではないためです。

手順

1. **KAFKA_JMX_OPTS** 環境変数を使用して **-Dcom.sun.management.jmxremote.port=<port>** を設定します。**<port>** には、Kafka ブローカーが JMX 接続をリッスンするポートの名前を入力します。

```
export KAFKA_JMX_OPTS="-Dcom.sun.management.jmxremote=true
 -Dcom.sun.management.jmxremote.port=<port>
```

```
  -Dcom.sun.management.jmxremote.authenticate=false
  -Dcom.sun.management.jmxremote.ssl=false"
bin/kafka-server-start.sh
```

2. JVM を起動します。

重要

リモート JMX 接続のセキュリティーを確保するために、認証および SSL を設定してください。これには、必要なシステムプロパティーの詳細は、JMX のドキュメント を参照してください。

8.4. JCONSOLE を使用した監視

JConsole ツールには、Java Development Kit(JDK)が同梱されています。JConsole を使用してローカルまたはリモート JVM に接続し、Java アプリケーションから管理情報を検出および表示できます。JConsole を使用してローカル JVM に接続する場合、AMQ Streams の異なるコンポーネントに対応する JVM プロセスの名前。

表8.1 AMQ Streams コンポーネントの JVM プロセス

AMQ Streams コンポーネント	JVM プロセス
ZooKeeper	**org.apache.zookeeper.server.quorum.QuorumPeerMain**
Kafka ブローカー	**kafka.Kafka**
Kafka Connect スタンドアロン	**org.apache.kafka.connect.cli.ConnectStandalone**
Kafka Connect の分散	**org.apache.kafka.connect.cli.ConnectDistributed**
Kafka プロデューサー、コンシューマー、またはストリームアプリケーション	アプリケーションの **main** メソッドが含まれるクラスの名前。

JConsole を使用してリモート JVM に接続する場合は、適切なホスト名と JMX ポートを使用します。

その他の多くのツールおよびモニタリング製品は、JMX を使用してメトリクスを取得し、これらのメトリクスに基づいて監視およびアラートを提供できます。これらのツールに関する詳細は、製品ドキュメントを参照してください。

8.5. 重要な KAFKA ブローカーメトリクス

Kafka では、Kafka クラスターのブローカーのパフォーマンスを監視するための MBean が多数提供されます。これらは、クラスター全体ではなく、個別のブローカーに適用されます。

以下の表は、サーバー、ネットワーク、ロギング、およびコントローラーメトリクスに整理されたこれらのブローカーレベルの MBean の選択を示しています。

8.5.1. Kafka サーバーメトリクス

以下の表は、Kafka サーバーに関する情報を報告するメトリクスの選択を示しています。

表8.2 Kafka サーバーのメトリクス

メトリクス	MBean	説明	想定される値
1秒あたりのメッセージ	**kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec**	ブローカーによって個別のメッセージが消費されるレート。	クラスターの他のブローカーとほぼ同じです。
1秒あたりのバイト数	**kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec**	プロデューサーから送信されたデータがブローカーによって消費されるレート。	クラスターの他のブローカーとほぼ同じです。
1秒あたりのレプリケーションバイト数	**kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec**	他のブローカーから送信されるデータがフォロワーブローカーによって使用されるレート。	該当なし
バイト毎秒バイト数	**kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec**	データがコンシューマーによってブローカーから取得され、読み取られるレート。	該当なし
レプリケーションバイト数毎秒単位です。	**kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec**	データがブローカーから送信されるレート。このメトリクスは、ブローカーがパーティションのグループのリーダーであるかどうかをモニターするのに役立ちます。	該当なし
複製の数が最低数未満であるパーティション	**kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions**	フォロワーレプリカで完全にレプリケートされていないパーティションの数。	ゼロ
最小 ISR パーティション数	**kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount**	最小 In-Sync Replica(ISR)数下のパーティション数。ISR カウントは、リーダーに最新となるレプリカのセットを示します。	ゼロ
パーティションの数	**kafka.server:type=ReplicaManager,name=PartitionCount**	ブローカーのパーティション数。	他のブローカーと比較した場合でも約。

メトリクス	MBean	説明	想定される値
リーダー数	**kafka.server:type=ReplicaManager,name=LeaderCount**	このブローカーがリーダーであるレプリカ数。	クラスターの他のブローカーとほぼ同じです。
ISR 縮小毎秒	**kafka.server:type=ReplicaManager,name=IsrShrinksPerSec**	ブローカーの ISR の数を減らす速度	ゼロ
ISR が毎秒拡張する	**kafka.server:type=ReplicaManager,name=IsrExpandsPerSec**	ブローカーの ISR の数が増える速度。	ゼロ
最大ラグ	**kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica**	メッセージがリーダーレプリカとフォロワーレプリカによって受信された時間の最大ラグ。	生成リクエストの最大バッチサイズに比例。
プロデューサーのパージにおけるリクエスト	**kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Produce**	プロデューサーペースマーターでの送信リクエストの数。	該当なし
取得中の要求	**kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Fetch**	フェッチペーターのフェッチリクエストの数。	該当なし
リクエストハンドラーの平均アイドルパーセント	**kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent**	リクエストハンドラー(IO)スレッドが使用されない時間の割合を示します。	値が小さいほど、ブローカーのワークロードが大きいことを示します。
request(Requests exempt from throttling)	**kafka.server:type=Request**	スロットリングによって実施される要求の数。	該当なし
ZooKeeper リクエストレイテンシー（ミリ秒単位）	**kafka.server:type=ZooKeeperClientMetrics,name=ZooKeeperRequestLatencyMs**	ブローカーからの ZooKeeper リクエストのレイテンシー（ミリ秒単位）。	該当なし
ZooKeeper セッションの状態	**kafka.server:type=SessionExpireListener,name=SessionState**	ZooKeeper へのブローカー接続の状態。	接続されました

8.5.2. Kafka ネットワークメトリクス

以下の表は、要求についての情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	想定される値
1秒あたりのリクエスト	**kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|FetchConsumer|FetchFollower}**	1秒あたりのリクエストタイプに対して行われる要求の合計数。**Produce**、**Fetch Consumer**、および **FetchFollower** リクエストタイプにはそれぞれ独自の MBean があります。	該当なし
要求バイト（バイト単位）	**kafka.network:type=RequestMetrics,name=RequestBytes,request=([-.\w]+)**	MBean 名の **request** プロパティーによって識別されるリクエストタイプに対して行われるリクエストのサイズ（バイト単位）。使用できるすべてのリクエストタイプの MBean は **RequestBytes** ノードに表示されます。	該当なし
一時的なメモリーサイズ（バイト単位）	**kafka.network:type=RequestMetrics,name=TemporaryMemoryBytes,request={Produce|Fetch}**	メッセージ形式の変換およびメッセージの圧縮に使用される一時的なメモリーの量。	該当なし
メッセージの変換時間	**kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce|Fetch}**	メッセージ形式の変換に費やされた時間（ミリ秒単位）。	該当なし
要求合計時間（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|FetchConsumer|FetchFollower}**	要求の処理に費やされた合計時間（ミリ秒単位）。	該当なし
要求キュー時間（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}**	リクエストを現在 **request** プロパティーに指定されたリクエストタイプのキューに費やした時間（ミリ秒単位）。	該当なし

メトリクス	MBean	説明	想定される値
ローカル時間（辞書ローカル処理時間）（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce|FetchConsumer|FetchFollower}**	リーダーが要求を処理するのにかかった時間（ミリ秒単位）。	該当なし
リモート時間（誤ったリモート処理時間）（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce|FetchConsumer|FetchFollower}**	リクエストがフォロワーを待つ期間（ミリ秒単位）。使用できるすべてのリクエストタイプの MBean は **RemoteTimeMs** ノードに表示されます。	該当なし
応答キュー時間（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}**	要求が応答キューを待つ期間（ミリ秒単位）。	該当なし
応答送信時間（ミリ秒単位）	**kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce|FetchConsumer|FetchFollower}**	応答を送信するまでの時間（ミリ秒単位）。	該当なし
ネットワークプロセッサーの平均アイドルパーセント	**kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent**	ネットワークプロセッサーがアイドル状態である時間の平均パーセンテージ。	0 から one

8.5.3. Kafka ログメトリクス

以下の表には、ロギングに関する情報を報告するメトリクスの選択をまとめています。

メトリクス	MBean	説明	予想される値
ログフラッシュレートおよび時間（ミリ秒単位）	**kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs**	ログデータがディスクに書き込まれる速度（ミリ秒単位）。	該当なし

メトリクス	MBean	説明	予想される値
オフラインのログディレクトリー数	**kafka.log:type=LogManager,name=OfflineLogDirectoryCount**	オフラインログディレクトリーの数（例： ハードウェア障害後の場合）	ゼロ

8.5.4. Kafka コントローラーメトリクス

以下の表は、クラスターのコントローラーに関する情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	予想される値
アクティブなコントローラー数	**kafka.controller:type=KafkaController,name=ActiveControllerCount**	コントローラーとして指定されたブローカーの数。	1つ目は、ブローカーがクラスターのコントローラーであることを示します。
リーダーの選択レートと時間（ミリ秒単位）	**kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs**	新規リーダーレプリカが選択されるレート。	ゼロ

8.5.5. Yammer メトリクス

レートまたは時間単位を表すメトリクスは、Yammer メトリクスとして提供されます。Yammer メトリクスを使用する MBean のクラス名の前に **com.yammer.metrics** が付けられます。

Yammer レートメトリクスには、要求を監視するための以下の属性があります。

- Count
- EventType(Bytes)
- FifteenMinuteRate
- RateUnit(Seconds)
- MeanRate
- OneMinuteRate
- FiveMinuteRate

Yammer 時間メトリクスには、要求を監視するための以下の属性があります。

- Max
- Min
- Mean

- StdDev
- 75/95/98/99/99.9th パーセンタイル

8.6. プロデューサー MBEAN

以下の MBean は、Kafka Streams アプリケーションやソースコネクターのある Kafka Connect を含む Kafka プロデューサーアプリケーションに存在します。

8.6.1. mbean matching kafka.producer:type=producer-metrics,client-id=*

これらはプロデューサーレベルでメトリクスです。

属性	説明
batch-size-avg	リクエストごとにパーティションごとに送信される平均バイト数。
batch-size-max	リクエストごとにパーティションごとに送信される最大バイト数。
batch-split-rate	1秒あたりのバッチ分割の平均数。
batch-split-total	バッチ分割の合計数。
buffer-available-bytes	使用されていないバッファーメモリーの合計量（割り当てまたは空き一覧内）。
buffer-total-bytes	クライアントが使用できるバッファーメモリーの最大量（現在使用中のかどうか）。
bufferpool-wait-time	アペンダーが領域の割り当てを待機する期間の分数。
compression-rate-avg	圧縮されていないサイズの圧縮の平均比率として定義されたレコードバッチの平均圧縮レート。
connection-close-rate	ウィンドウで、1秒あたり接続が閉じられます。
connection-count	現在のアクティブな接続の数。
connection-creation-rate	ウィンドウに1秒ごとに確立される新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットをバイト/秒単位で読み取ります。
io-ratio	I/O スレッドに費やした I/O スレッドの割合

属性	説明
io-time-ns-avg	選択した呼び出しごとに I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やされた時間の割合
io-wait-time-ns-avg	読み取りまたは書き込みの待機に費やされた I/O スレッドの平均時間（ナノ秒単位）。
metadata-age	使用されている現在のプロデューサーメタデータの経過時間（秒単位）。
network-io-rate	1 秒あたりのすべての接続に対するネットワーク操作（読み取りまたは書き込み）の平均数。
outgoing-byte-rate	1 秒あたり全サーバーへ送信される送信バイトの平均数
produce-throttle-time-avg	リクエストの平均時間（ミリ秒単位）は、リクエストがブローカーによってスロットルされました。
produce-throttle-time-max	リクエストがブローカーによってスロットリングされた最大時間（ミリ秒単位）。
record-error-rate	レコードの 1 秒あたりの平均は、エラーの原因となった記録を送信します。
record-error-total	レコードの合計数は、エラーの原因となったものです。
record-queue-time-avg	送信バッファーで費やされたミリ秒単位の平均時間（ミリ秒）。
record-queue-time-max	送信バッファーで費やされた最大時間（ミリ秒）
record-retry-rate	再試行記録の平均秒数。
record-retry-total	再試行されたレコードの総数
record-send-rate	1 秒あたり送信される平均のレコード数。
record-send-total	送信されたレコードの総数
record-size-avg	平均レコードサイズ
record-size-max	レコードの最大サイズ。

属性	説明
records-per-request-avg	リクエストごとのレコードの平均数。
request-latency-avg	平均リクエストレイテンシー（ミリ秒）
request-latency-max	最大リクエストレイテンシー（ミリ秒）
request-rate	1秒あたり送信されたリクエストの平均数
request-size-avg	ウィンドウ内のすべての要求の平均サイズ。
request-size-max	ウィンドウに送信されたリクエストの最大サイズ。
requests-in-flight	応答を待機中の現在のインフライトリクエストの数。
response-rate	1秒あたり送信された応答。
select-rate	I/O 層が毎秒実行される新しい I/O 層をチェックする回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続
waiting-threads	バッファーメモリーがレコードをキューに入れるのを待つユーザースレッドの数。

8.6.2. mbean matching kafka.producer:type=producer-metrics,client-id=*,node-id=*

これらのメトリクスは、各ブローカーへの接続に関するプロデューサーレベルでメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される応答の平均数。
outgoing-byte-rate	ノードに対して1秒あたりの送信バイトの平均バイト数。
request-latency-avg	ノードの平均リクエストレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大リクエストレイテンシー（ミリ秒）。
request-rate	ノードに対して1秒あたりの送信されるリクエストの平均数

属性	説明
request-size-avg	ノードのウィンドウでのすべての要求の平均サイズ。
request-size-max	ノードのウィンドウに送信された要求の最大サイズ。
response-rate	ノードに対して1秒ごとに送信された応答。

8.6.3. mbean matching kafka.producer:type=producer-topic-metrics,client-id=*,topic=*

これらは、プロデューサーがメッセージを送信するトピックに関するトピックレベルでメトリクスです。

属性	説明
byte-rate	トピックに対して1秒あたりの送信される平均バイト数。
byte-total	トピックに送信された合計バイト数。
compression-rate	圧縮されていないサイズの圧縮サイズの平均比率として定義されたトピックのレコードバッチの平均圧縮レート。
record-error-rate	毎秒のレコード平均は、トピックのエラーの原因となったレコードを送信します。
record-error-total	トピックのエラーとなるレコードの合計数。
record-retry-rate	トピックに送信する再試行レコードの平均（秒単位）。
record-retry-total	トピックに送信する再試行レコードの合計数。
record-send-rate	トピックに対して1秒あたり送信される平均のレコード数。
record-send-total	トピックに送信されたレコードの合計数。

8.7. コンシューマー MBEAN

以下の MBean は、Kafka Streams アプリケーションおよびシンクコネクターのある Kafka Connect を含む Kafka コンシューマーアプリケーションに存在します。

8.7.1. mbean matching kafka.consumer:type=consumer-metrics,client-id=*

これらはコンシューマーレベルでメトリクスです。

属性	説明
connection-close-rate	ウィンドウで、1秒あたり接続が閉じられます。
connection-count	現在のアクティブな接続の数。
connection-creation-rate	ウィンドウに1秒ごとに確立される新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットをバイト/秒単位で読み取ります。
io-ratio	I/O スレッドに費やした I/O スレッドの割合
io-time-ns-avg	選択した呼び出しごとに I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やされた時間の割合
io-wait-time-ns-avg	読み取りまたは書き込みの待機に費やされた I/O スレッドの平均時間（ナノ秒単位）。
network-io-rate	1秒あたりのすべての接続に対するネットワーク操作（読み取りまたは書き込み）の平均数。
outgoing-byte-rate	1秒あたり全サーバーへ送信される送信バイトの平均数
request-rate	1秒あたり送信されたリクエストの平均数
request-size-avg	ウィンドウ内のすべての要求の平均サイズ。
request-size-max	ウィンドウに送信されたリクエストの最大サイズ。
response-rate	1秒あたり送信された応答。
select-rate	I/O 層が毎秒実行される新しい I/O 層をチェックする回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続

8.7.2. mbean matching kafka.consumer:type=consumer-metrics,client-id=*,node-id=*

これらのメトリクスは、各ブローカーへの接続に関するコンシューマーレベルでメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される応答の平均数。
outgoing-byte-rate	ノードに対して1秒あたりの送信バイトの平均バイト数。
request-latency-avg	ノードの平均リクエストレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大リクエストレイテンシー（ミリ秒）。
request-rate	ノードに対して1秒あたりの送信されるリクエストの平均数
request-size-avg	ノードのウィンドウでのすべての要求の平均サイズ。
request-size-max	ノードのウィンドウに送信された要求の最大サイズ。
response-rate	ノードに対して1秒ごとに送信された応答。

8.7.3. mbean matching kafka.consumer:type=consumer-coordinator-metrics,client-id=*

これらは、コンシューマーグループのコンシューマーレベルでメトリクスです。

属性	説明
assigned-partitions	このコンシューマーに現在割り当てられているパーティションの数。
commit-latency-avg	コミットリクエストの平均時間。
commit-latency-max	コミットリクエストにかかった最大時間。
commit-rate	1秒あたりのコミット呼び出しの数。
heartbeat-rate	1秒あたりのハートビートの平均数
heartbeat-response-time-max	ハートビートリクエストへの応答を受信するのにかかった最大時間。
join-rate	1秒あたりのグループ参加数。
join-time-avg	グループ再参加にかかった平均時間。

属性	説明
join-time-max	グループ再参加にかかった最大時間。
last-heartbeat-seconds-ago	最後のコントローラーハートビートからの秒数。
sync-rate	1秒あたりのグループ同期数。
sync-time-avg	グループ同期の平均時間。
sync-time-max	グループ同期の最大時間。

8.7.4. mbean matching kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*

これらは、コンシューマーのャーマーに関するコンシューマーレベルでメトリクスです。

属性	説明
bytes-consumed-rate	1秒あたり消費される平均のバイト数。
bytes-consumed-total	消費される合計バイト数。
fetch-latency-avg	フェッチリクエストにかかった時間。
fetch-latency-max	フェッチリクエストにかかった最大時間。
fetch-rate	1秒あたりのフェッチリクエスト数
fetch-size-avg	リクエストごとにフェッチされる平均バイト数。
fetch-size-max	リクエストごとにフェッチされる最大バイト数。
fetch-throttle-time-avg	平均スロットル時間（ミリ秒）
fetch-throttle-time-max	最大スロットル時間（ミリ秒）
fetch-total	フェッチリクエストの合計数。
records-consumed-rate	1秒あたり消費される平均のレコード数。
records-consumed-total	消費されるレコードの総数
records-lag-max	このウィンドウで任意のパーティションのレコード数という最大ラグ。

属性	説明
records-lead-min	このウィンドウで、パーティションのレコード数の上限です。
records-per-request-avg	各リクエストにおけるレコードの平均数。

8.7.5. mbean matching kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*

これらは、コンシューマーのャーマーに関するトピックレベルでメトリクスです。

属性	説明
bytes-consumed-rate	トピックに対して1秒あたりの消費される平均バイト数。
bytes-consumed-total	トピックで消費される合計バイト数。
fetch-size-avg	トピックのリクエストごとにフェッチされる平均バイト数。
fetch-size-max	トピックのリクエストごとにフェッチされる最大バイト数。
records-consumed-rate	トピックに対して1秒あたりの消費される平均のレコード数。
records-consumed-total	トピックで消費されるレコードの合計数。
records-per-request-avg	トピックの各リクエストにおけるレコードの平均数。

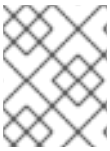
8.7.6. mbean matching kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*,partition=*

これらは、コンシューマーのャーマーに関するパーティションレベルでメトリックです。

属性	説明
preferred-read-replica	パーティションの現在の読み取りレプリカ。リーダーから読み込む場合は -1。
records-lag	パーティションの最新ラグ。
records-lag-avg	パーティションの平均ラグ。

属性	説明
records-lag-max	パーティションの最大ラグ。
records-lead	パーティションの最新のリード。
records-lead-avg	パーティションの平均リード。
records-lead-min	パーティションの最小値リード。

8.8. KAFKA CONNECT MBEAN



注記

Kafka Connect には、ここに記載されているものに加えて、シンクコネクターのソースコネクターおよび コンシューマー MBean の プロデューサー MBean が含まれます。

8.8.1. mbean matching kafka.connect:type=connect-metrics,client-id=*

これらは、接続レベルでメトリクスです。

属性	説明
connection-close-rate	ウィンドウで、1秒あたり接続が閉じられます。
connection-count	現在のアクティブな接続の数。
connection-creation-rate	ウィンドウに1秒ごとに確立される新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットをバイト/秒単位で読み取ります。
io-ratio	I/O スレッドに費やした I/O スレッドの割合
io-time-ns-avg	選択した呼び出しごとに I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やされた時間の割合
io-wait-time-ns-avg	読み取りまたは書き込みの待機に費やされた I/O スレッドの平均時間（ナノ秒単位）。
network-io-rate	1秒あたりのすべての接続に対するネットワーク操作（読み取りまたは書き込み）の平均数。

属性	説明
outgoing-byte-rate	1秒あたり全サーバーへ送信される送信バイトの平均数
request-rate	1秒あたり送信されたリクエストの平均数
request-size-avg	ウィンドウ内のすべての要求の平均サイズ。
request-size-max	ウィンドウに送信されたリクエストの最大サイズ。
response-rate	1秒あたり送信された応答。
select-rate	I/O 層が毎秒実行される新しい I/O 層をチェックする回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続

8.8.2. mbean matching kafka.connect:type=connect-metrics,client-id=*,node-id=*

これらのメトリクスは、各ブローカーへの接続に関する接続レベルでメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される応答の平均数。
outgoing-byte-rate	ノードに対して1秒あたりの送信バイトの平均バイト数。
request-latency-avg	ノードの平均リクエストレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大リクエストレイテンシー（ミリ秒）。
request-rate	ノードに対して1秒あたりの送信されるリクエストの平均数
request-size-avg	ノードのウィンドウでのすべての要求の平均サイズ。
request-size-max	ノードのウィンドウに送信された要求の最大サイズ。
response-rate	ノードに対して1秒ごとに送信された応答。

8.8.3. mbean matching kafka.connect:type=connect-worker-metrics

これらは、接続レベルでメトリクスです。

属性	説明
connector-count	このワーカーで実行されるコネクターの数。
connector-startup-attempts-total	このワーカーが試行したコネクターの起動の合計数。
connector-startup-failure-percentage	このワーカーのコネクターの平均パーセンテージは失敗して開始します。
connector-startup-failure-total	失敗したコネクターの合計数。
connector-startup-success-percentage	このワーカーのコネクターの平均パーセンテージは正常です。
connector-startup-success-total	正常に実行されたコネクターの合計数。
task-count	このワーカーで実行されるタスクの数。
task-startup-attempts-total	このワーカーが試行したタスク起動の合計数。
task-startup-failure-percentage	このワーカーのタスクの平均パーセンテージは、失敗したことを示します。
task-startup-failure-total	失敗したタスクの開始回数。
task-startup-success-percentage	このワーカーのタスクの平均のパーセンテージは正常に開始されます。
task-startup-success-total	成功したタスクの開始回数。

8.8.4. mbean matching kafka.connect:type=connect-worker-rebalance-metrics

属性	説明
completed-rebalances-total	このワーカーによって完了したリバランスの合計数。
connect-protocol	このクラスターによって使用される Connect プロトコル。
epoch	このワーカーのエポックまたは生成数。

属性	説明
leader-name	グループリーダーの名前。
rebalance-avg-time-ms	このワーカーがリバランスに費やされた平均時間（ミリ秒単位）。
rebalance-max-time-ms	このワーカーがリバランスに費やされた最大時間（ミリ秒単位）。
リバランス	このワーカーが現在リバランスされているかどうか。
time-since-last-rebalance-ms	このワーカーが最新のリバランスを完了してからの時間（ミリ秒単位）。

8.8.5. mbean matching kafka.connect:type=connector-metrics,connector=*

属性	説明
connector-class	コネクタークラスの名前。
connector-type	コネクターのタイプ。'source' または 'sink' のいずれか。
connector-version	コネクターによって報告されるコネクタークラスのバージョン。
status	コネクターの状態。'unassigned'、'running'、'paused'、'failed'、または 'destroyed' のいずれか。

8.8.6. mbean matching kafka.connect:type=connector-task-metrics,connector=*,task=*

属性	説明
batch-size-avg	コネクターによって処理されるバッチの平均サイズ。
batch-size-max	コネクターによって処理されるバッチの最大サイズ。
offset-commit-avg-time-ms	このタスクがオフセットをコミットするのにかかった時間（ミリ秒単位）。
offset-commit-failure-percentage	このタスクのオフセットコミットの平均パーセンテージ。失敗した場合。

属性	説明
offset-commit-max-time-ms	このタスクがオフセットをコミットするのにかかった最大時間（ミリ秒単位）。
offset-commit-success-percentage	このタスクのオフセットコミットの平均パーセンテージ。正常に実行された場合。
pause-ratio	このタスクが pause の状態で費やした時間の割合。
running-ratio	このタスクが running 状態で費やした時間の割合。
status	コネクタータスクのステータス。'unassigned'、'running'、'paused'、'failed'、または 'destroyed' のいずれか。

8.8.7. mbean matching kafka.connect:type=sink-task-metrics,connector=*,task=*

属性	説明
offset-commit-completion-rate	正常に完了したオフセットコミットの完了の平均秒数。
offset-commit-completion-total	正常に完了したオフセットコミットの完了の合計数。
offset-commit-seq-no	オフセットコミットの現在のシーケンス番号。
offset-commit-skip-rate	受信時間とスキップされたオフセットコミットの完了の平均毎秒。
offset-commit-skip-total	受信したオフセットコミットの完了の合計数およびスキップ/無視回数。
partition-count	このワーカーの名前付きシンクコネクターに属するこのタスクに割り当てられるトピックパーティションの数。
put-batch-avg-time-ms	シンクレコードのバッチを配置するためにこのタスクによってかかる平均時間。
put-batch-max-time-ms	シンクレコードのバッチを配置するためにこのタスクによってかかる最大時間。
sink-record-active-count	Kafka から読み取られているが、シンクタスクによって完全にコミットされた/フラッシュ/承認されていないレコードの数。

属性	説明
sink-record-active-count-avg	Kafka から読み取られていて、シンクタスクによって完全にコミットされた/フラッシュされた/承認されていないレコードの平均数。
sink-record-active-count-max	Kafka から読み取られていて、シンクタスクによって完全にコミットされた/フラッシュされた/承認されていないレコードの最大数。
sink-record-lag-max	シンクタスクがすべてのトピックパーティションのコンシューマーの位置の背後にあるレコード数の最大ラグ。
sink-record-read-rate	このワーカーで名前付きシンクコネクターに属するこのタスクに対して Kafka から読み取られる平均毎秒（秒単位）。これは変換を適用する前です。
sink-record-read-total	タスクが最後に再起動されたため、このワーカーの名前付きシンクコネクターに属するこのタスクによって Kafka から読み取られるレコードの合計数。
sink-record-send-rate	変換からの 1 秒あたりの平均のレコード数と、このワーカーの名前付きシンクコネクターに属するこのタスクへの送信/スループットの平均。これは変換の適用後に、変換でフィルターされたレコードが除外されます。
sink-record-send-total	変換から出力されたレコードの総数と、タスクが最後に再起動してからこのワーカーの名前付きシンクコネクターに属するこのタスクへの送信/スループット。

8.8.8. mbean matching kafka.connect:type=source-task-metrics,connector=*,task=*

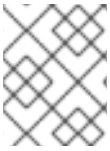
属性	説明
poll-batch-avg-time-ms	ソースレコードのバッチをポーリングするためにこのタスクによってかかる平均時間（ミリ秒単位）。
poll-batch-max-time-ms	ソースレコードのバッチをポーリングするためにこのタスクがかかる最大時間（ミリ秒単位）。
source-record-active-count	このタスクによって生成されたが、Kafka に完全に書き込まれていないレコード数。
source-record-active-count-avg	このタスクによって生成されたものの、Kafka に完全に書き込まれていないレコードの平均数。

属性	説明
source-record-active-count-max	このタスクによって生成されたものの、Kafka に完全に書き込まれていないレコードの最大数。
source-record-poll-rate	このワーカーの名前付きソースコネクターに属するこのタスクによって生成されたレコードの 1 秒あたりのレコードの平均（以前の変換）。
source-record-poll-total	このワーカー内の名前付きソースコネクターに属するこのタスクによって生成されたレコード/ポーリング（変換の前）の合計数。
source-record-write-rate	このワーカーの名前付きソースコネクターに属するこのタスクの変換と Kafka に書き込まれたレコード平均のレコード数。これは変換の適用後に、変換でフィルターされたレコードが除外されます。
source-record-write-total	変換から出力され、タスクが最後に再起動してから、このワーカーの名前付きソースコネクターに属するこのタスクの Kafka に書き込まれるレコード数。

8.8.9. mbean matching kafka.connect:type=task-error-metrics,connector=*,task=*

属性	説明
deadletterqueue-produce-failures	デッドレターキューへの書き込みに失敗した回数。
deadletterqueue-produce-requests	デッドレターキューへの書き込みを試行する回数。
last-error-timestamp	このタスクが最後にエラーが発生した場合のエポックタイムスタンプ。
total-errors-logged	ログに記録されたエラーの数。
total-record-errors	このタスクのレコード処理エラーの数。
total-record-failures	このタスクのレコード処理の失敗回数。
total-records-skipped	エラーによりスキップされたレコードの数。
total-retries	再試行された操作の数。

8.9. KAFKA STREAMS MBEAN



注記

ストリームアプリケーションには、ここに記載されているものに加えて、プロデューサー および コンシューマー MBean が含まれます。

8.9.1. mbean matching kafka.streams:type=stream-metrics,client-id=*

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **info** または **debug** の場合に収集されます。

属性	説明
commit-latency-avg	このスレッドを実行しているすべてのタスクでコミットする平均実行時間（ミリ秒単位）。
commit-latency-max	このスレッドの実行タスクすべてにコミットする最大時間（ミリ秒単位）。
commit-rate	1秒あたりのコミットの平均数。
commit-total	全タスクにわたるコミットコールの合計数。
poll-latency-avg	このスレッドを実行しているすべてのタスクにおけるポーリングの平均実行時間（ミリ秒単位）。
poll-latency-max	このスレッドの実行すべてのタスクにポーリングする最大実行時間（ミリ秒）。
poll-rate	1秒あたりのポーリングの平均数。
poll-total	全タスクにおけるポーリング呼び出しの合計数。
process-latency-avg	このスレッドの実行タスクがすべて実行されるすべての処理にかかる平均実行時間（ミリ秒単位）。
process-latency-max	このスレッドの実行中にすべての実行中のタスクを処理する最大時間（ミリ秒単位）。
process-rate	1秒あたりのプロセス呼び出しの平均数。
process-total	全タスクにわたるプロセス呼び出しの合計数。
punctuate-latency-avg	このスレッドで実行しているすべてのタスクを対象とする平均実行時間（ミリ秒単位）。
punctuate-latency-max	このスレッドで実行しているすべてのタスクに購入する最大時間（ミリ秒単位）。
punctuate-rate	1秒あたりの平均の句数。

属性	説明
punctuate-total	すべてのタスク全体での呼び出しの総数。
skipped-records-rate	1秒あたりのスキップされたレコードの平均数。
skipped-records-total	スキップしたレコードの総数
task-closed-rate	1秒あたり閉じられているタスクの平均数
task-closed-total	終了したタスクの合計数。
task-created-rate	1秒あたりの新規作成されたタスクの平均数。
task-created-total	作成されたタスクの合計数。

8.9.2. mbean matching kafka.streams:type=stream-task-metrics,client-id=*,task-id=*

タスクメトリクス。

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** の場合に収集されます。

属性	説明
commit-latency-avg	このタスクの平均コミット時間(ns)。
commit-latency-max	このタスクの ns での最大コミット時間。
commit-rate	1秒あたりのコミット呼び出しの平均数。
commit-total	コミット呼び出しの総数

8.9.3. mbean matching kafka.streams:type=stream-processor-node-metrics,client-id=*,task-id=*,processor-node-id=*

プロセッサーノードメトリクス。

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** の場合に収集されます。

属性	説明
create-latency-avg	ns の平均作成時間。
create-latency-max	ns での最大作成時間
create-rate	1秒あたりの作成操作の平均数。

属性	説明
create-total	呼び出された作成操作の合計数。
destroy-latency-avg	ns の平均破棄実行時間。
destroy-latency-max	ns での最大破棄実行時間
destroy-rate	1秒あたりの破棄操作の平均数。
destroy-total	呼び出された破棄操作の合計数。
forward-rate	転送されたレコードの平均レートは、1秒あたりのソースノードからのみ転送されます。
forward-total	ソースノードからのみ転送されたレコードの合計数。
process-latency-avg	ns の平均プロセス実行時間。
process-latency-max	ns でのプロセス実行時間
process-rate	1秒あたりのプロセス操作の平均数
process-total	呼び出されたプロセス操作の合計数。
punctuate-latency-avg	ns での平均的な実行時間。
punctuate-latency-max	ns での実行時間の最大値。
punctuate-rate	1秒あたりの遅延操作の平均数。
punctuate-total	呼び出された句操作の合計数。

8.9.4. mbean matching kafka.streams:type=stream-[store-scope]-metrics,client-id=*,task-id=*,[store-scope]-id=*

ステートストアメトリクス。

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** の場合に収集されます。

属性	説明
all-latency-avg	ns でのすべての操作実行時間の平均
all-latency-max	ns でのすべての操作実行時間の最大。

属性	説明
all-rate	このストアの平均操作レート。
all-total	このストアに対する全操作呼び出しの総数
delete-latency-avg	ns の平均削除実行時間。
delete-latency-max	ns での最大削除実行時間。
delete-rate	このストアの平均削除速度。
delete-total	このストアの削除呼び出しの合計数。
flush-latency-avg	ns の平均フラッシュ実行時間。
flush-latency-max	ns での最大フラッシュ実行時間。
flush-rate	このストアの平均フラッシュレート。
flush-total	このストアのフラッシュ呼び出しの合計数。
get-latency-avg	ns の平均取得時間。
get-latency-max	ns での最大実行時間
get-rate	このストアの平均 get レート。
get-total	このストアの get 呼び出しの合計数。
put-all-latency-avg	ns の平均 put-all 実行時間。
put-all-latency-max	ns での put-all 実行最大時間。
put-all-rate	このストアの平均 put-all レート。
put-all-total	このストアの put-all 呼び出しの合計数。
put-if-absent-latency-avg	ns での put-if-absent 実行時間の平均
put-if-absent-latency-max	ns での put-if-absent 実行時間の最大値。
put-if-absent-rate	このストアの平均 put-if-absent レート。
put-if-absent-total	このストアの put-if-absent 呼び出しの合計数。

属性	説明
put-latency-avg	平均実行時間を ns に配置します。
put-latency-max	ns での実行時間の最大化。
put-rate	このストアの平均配置レート。
put-total	このストアに対する put call の合計数。
range-latency-avg	ns の平均範囲実行時間。
range-latency-max	ns での最大範囲実行時間
range-rate	このストアの平均範囲のレート。
range-total	このストアの範囲呼び出しの合計数。
restore-latency-avg	ns の平均復元実行時間。
restore-latency-max	ns での最大復元実行時間
restore-rate	このストアの平均リストアレート。
restore-total	このストアの復元呼び出しの合計数。

8.9.5. mbean matching **kafka.streams:type=stream-record-cache-metrics,client-id=*,task-id=*,record-cache-id=***

キャッシュメトリクスを記録します。

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** の場合に収集されます。

属性	説明
hitRatio-avg	キャッシュの読み取りヒット合計率として定義された平均キャッシュヒット率（合計キャッシュの読み取り要求数）。
hitRatio-max	キャッシュの最大ヒット比率。
hitRatio-min	ミューラムキャッシュのヒット比率。

第9章 KAFKA CONNECT

Kafka Connect は、Apache Kafka と外部システムとの間でデータをストリーミングするためのツールです。スケーラビリティーと信頼性を維持しながら、大量のデータを移動するためのフレームワークが提供されます。通常、Kafka Connect は、Kafka クラスター外部のデータベース、ストレージ、メッセージングシステムと統合するために使用されます。

Kafka Connect は、異なるタイプの外部システムへの接続を実装するコネクタープラグインを使用します。シンクコネクターには、sink および source の 2 つのタイプがあります。シンクコネクターは、Kafka から外部システムにデータをストリーミングします。ソースコネクターは、外部システムから Kafka にデータをストリーミングします。

Kafka Connect はスタンドアロンまたは分散モードで実行できます。

スタンドアロンモード

スタンドアロンモードでは、Kafka Connect はプロパティーファイルから読み取られたユーザー定義の設定を持つ単一ノードで実行されます。

分散モード

Distributed モードでは、Kafka Connect は 1 つまたは複数のワーカーノードで実行され、ワークロードはそれらのワーカーノード間で分散されます。HTTP REST インターフェースを使用して、コネクターとその設定を管理します。

9.1. スタンドアロンモードでの KAFKA CONNECT

スタンドアロンモードでは、Kafka Connect は単一ノードで単一のプロセスとして実行されます。プロパティーファイルを使用してスタンドアロンモードの設定を管理します。

9.1.1. スタンドアロンモードでの Kafka Connect の設定

スタンドアロンモードで Kafka Connect を設定するには、**config/connect-standalone.properties** 設定ファイルを編集します。以下のオプションは最も重要なオプションです。

bootstrap.servers

Kafka へのブートストラップ接続として使用される Kafka ブローカーアドレスの一覧。たとえば、**kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092** のようになります。

key.converter

メッセージキーを Kafka 形式との間で変換するために使用されるクラス。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

offset.storage.file.filename

オフセットデータが保存されるファイルを指定します。

設定ファイルの例は、**config/connect-standalone.properties** のインストールディレクトリーで提供されます。サポートされるすべての Kafka Connect 設定オプションの完全リストは、[kafka-connect-configuration-parameters-str] を参照してください。

コネクタープラグインは、ブートストラップアドレスを使用して Kafka ブローカーへのクライアント接続を開きます。これらの接続を設定するには、**producer.** または **consumer.** で始まる標準の Kafka プロデューサーおよびコンシューマー設定オプションを使用します。

Kafka プロデューサーおよびコンシューマーの設定に関する詳細は、以下を参照してください。

- 付録D *プロデューサー設定パラメーター*
- 付録C *コンシューマー設定パラメーター*

9.1.2. スタンドアロンモードでの Kafka Connect でのコネクターの設定

プロパティーファイルを使用すると、スタンドアロンモードで Kafka Connect のコネクタープラグインを設定できます。ほとんどの設定オプションは、各コネクターに固有のものです。以下のオプションはすべてのコネクターに適用されます。

name
: 現在の Kafka Connect インスタンス内で一意である必要があります。

connector.class
: コネクタープラグインのクラス。たとえば、**org.apache.kafka.connect.file.FileStreamSinkConnector** のようになります。

tasks.max
: 指定されたコネクターが使用可能なタスクの最大数。タスクを使用すると、コネクターが並行して機能できるようになります。コネクターは指定された値よりも少ないタスクが作成される可能性があります。

key.converter
: メッセージキーを Kafka 形式との間で変換するために使用されるクラス。これにより、Kafka Connect 設定によって設定されるデフォルト値が上書きされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

value.converter
: メッセージペイロードの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されるデフォルト値が上書きされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

さらに、シンクコネクターの以下のオプションのいずれかを設定する必要があります。

topics
: 入力に使用されるトピックのコンマ区切りリスト。

topics.regex
: 入力に使用されるトピックの Java 正規表現。

他のすべてのオプションは、利用可能なコネクターのドキュメントを参照してください。

AMQ Streams には、コネクター設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/connect-file-sink.properties** および **config/connect-file-source.properties** を参照してください。

9.1.3. スタンドアロンモードでの Kafka Connect の実行

この手順では、スタンドアロンモードで Kafka Connect を設定して実行する方法を説明します。

前提条件

- AMQ Streams} クラスターがインストールされ、実行されている。

手順

手順

1. **/opt/kafka/config/connect-standalone.properties** Kafka Connect 設定ファイルを編集し、**bootstrap.server** を設定して Kafka ブローカーを示すように設定します。以下に例を示します。

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
```

2. 設定ファイルを使用して Kafka Connect を開始し、1 つ以上のコネクター設定を指定します。

```
su - kafka
/opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties
connector1.properties
[connector2.properties ...]
```

3. Kafka Connect が実行されていることを確認します。

```
jcmd | grep ConnectStandalone
```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- サポートされる Kafka Connect 設定オプションの完全リストは、付録F *Kafka Connect 設定パラメーター* を参照してください。

9.2. 「KAFKA CONNECT IN DISTRIBUTED MODE」

Distributed モードでは、Kafka Connect は 1 つまたは複数のワーカーノードで実行され、ワークロードはそれらのワーカーノード間で分散されます。HTTP REST インターフェースを使用して、コネクタープラグインおよびその設定を管理します。

9.2.1. 分散モードでの Kafka Connect の設定

分散モードで Kafka Connect を設定するには、**config/connect-distributed.properties** 設定ファイルを編集します。以下のオプションは最も重要なオプションです。

bootstrap.servers

Kafka へのブートストラップ接続として使用される Kafka ブローカーアドレスの一覧。たとえば、**kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092** のようになります。

key.converter

メッセージキーを Kafka 形式との間で変換するために使用されるクラス。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

group.id

分散 Kafka Connect クラスターの名前。これは一意で、別のコンシューマーグループ ID と競合しないようにする必要があります。デフォルト値は **connect-cluster** です。

config.storage.topic

コネクター設定を保存するために使用される Kafka トピック。デフォルト値は **connect-configs** です。

offset.storage.topic

オフセットを保存するために使用される Kafka トピック。デフォルト値は **connect-offset** です。

status.storage.topic

ワーカーノードのステータスに使用する Kafka トピック。デフォルト値は **connect-status** です。

AMQ Streams には、分散モードの Kafka Connect の設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/connect-distributed.properties** を参照してください。

サポートされるすべての Kafka Connect 設定オプションの完全リストは、付録F *Kafka Connect 設定パラメーター* を参照してください。

コネクタープラグインは、ブートストラップアドレスを使用して Kafka ブローカーへのクライアント接続を開きます。これらの接続を設定するには、**producer.** または **consumer.** で始まる標準の Kafka プロデューサーおよびコンシューマー設定オプションを使用します。

Kafka プロデューサーおよびコンシューマーの設定に関する詳細は、以下を参照してください。

- 付録D *プロデューサー設定パラメーター*
- 付録C *コンシューマー設定パラメーター*

9.2.2. 分散 Kafka Connect でのコネクターの設定

HTTP REST インターフェース

分散 Kafka Connect のコネクターは HTTP REST インターフェースを使用して設定されます。REST インターフェースは、デフォルトで 8083 番ポートでリッスンします。以下のエンドポイントをサポートします。

GET /connectors

既存のコネクターのリストを返します。

POST /connectors

コネクターを作成します。リクエスト本文はコネクター設定のある JSON オブジェクトである必要があります。

GET /connectors/<name>

特定のコネクターに関する情報を取得します。

GET /connectors/<name>/config

特定のコネクターの設定を取得します。

PUT /connectors/<name>/config

特定のコネクターの設定を更新します。

GET /connectors/<name>/status

特定のコネクターの状態を取得します。

PUT /connectors/<name>/pause

コネクターとそのすべてのタスクを一時停止します。コネクターはメッセージの処理を停止します。

PUT /connectors/<name>/resume

一時停止されたコネクターを再開します。

POST /connectors/<name>/restart

失敗した場合はコネクターを再起動します。

DELETE /connectors/<name>

コネクターを削除します。

GET /connector-plugins

サポートされるコネクタープラグインの一覧を取得します。

コネクターの設定

ほとんどの設定オプションはコネクター固有で、コネクターのドキュメントを参照してください。以下のフィールドはすべてのコネクターで共通しています。

name

コネクターの名前。指定の Kafka Connect インスタンス内で一意である必要があります。

connector.class

コネクタープラグインのクラス。たとえば、**org.apache.kafka.connect.file.FileStreamSinkConnector** のようになります。

tasks.max

このコネクターによって使用されるタスクの最大数。タスクは、作業を並列化するためにコネクターによって使用されます。コンテーターは、指定した量よりも少ないタスクを作成することができます。

key.converter

メッセージキーの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されるデフォルト値が上書きされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

value.converter

メッセージペイロードを Kafka 形式との間で変換するために使用されるクラス。これにより、Kafka Connect 設定によって設定されるデフォルト値が上書きされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** のようになります。

さらに、シンクコネクターに以下のオプションのいずれかを設定する必要があります。

topics

入力に使用されるトピックのコンマ区切りリスト。

topics.regex

入力に使用されるトピックの Java 正規表現。

他のすべてのオプションは、特定のコネクターのドキュメントを参照してください。

AMQ Streams には、コネクター設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/connect-file-sink.properties** および **config/connect-file-source.properties** にあります。

9.2.3. 分散 Kafka Connect の実行

この手順では、分散モードで Kafka Connect を設定して実行する方法を説明します。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。

クラスターの実行

1. すべての Kafka Connect ワーカーノードで **/opt/kafka/config/connect-distributed.properties** Kafka Connect 設定ファイルを編集します。
   - **bootstrap.server** オプションを設定して、Kafka ブローカーを指定します。
   - **group.id** オプションを設定します。
   - **config.storage.topic** オプションを設定します。
   - **offset.storage.topic** オプションを設定します。
   - **status.storage.topic** オプションを設定します。
     以下に例を示します。

     ```
     bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
     group.id=my-group-id
     config.storage.topic=my-group-id-configs
     offset.storage.topic=my-group-id-offsets
     status.storage.topic=my-group-id-status
     ```

2. すべての Kafka Connect ノードで、**/opt/kafka/config/connect-distributed.properties** 設定ファイルを使用して Kafka Connect ワーカーを起動します。

   ```
   su - kafka
   /opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
   ```

3. Kafka Connect が実行されていることを確認します。

   ```
   jcmd | grep ConnectDistributed
   ```

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- サポートされる Kafka Connect 設定オプションの完全リストは、付録F *Kafka Connect 設定パラメーター* を参照してください。

9.2.4. コネクターの作成

この手順では、Kafka Connect REST API を使用して分散モードで Kafka Connect で使用するコネクタープラグインを作成する方法を説明します。

前提条件

- 分散モードで実行される Kafka Connect インストール。

手順

1. コネクター設定で JSON ペイロードを準備します。以下に例を示します。

```
{
  "name": "my-connector",
  "config": {
  "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "topics": "my-topic-1,my-topic-2",
    "file": "/tmp/output-file.txt"
  }
}
```

2. POST リクエストを **<KafkaConnectAddress>:8083/connectors** に送信し、コネクターを作成します。以下の例では、**curl** を使用しています。

```
curl -X POST -H "Content-Type: application/json" --data @sink-connector.json
http://connect0.my-domain.com:8083/connectors
```

3. GET リクエストを **<KafkaConnectAddress>:8083/connectors** に送信し、コネクターがデプロイされていることを確認します。以下の例では、**curl** を使用しています。

```
curl http://connect0.my-domain.com:8083/connectors
```

9.2.5. コネクターの削除

この手順では、Kafka Connect REST API を使用して分散モードの Kafka Connect からコネクタープラグインを削除する方法を説明します。

前提条件

- 分散モードで実行される Kafka Connect インストール。

コネクターの削除

1. **GET** リクエストを **<KafkaConnectAddress>:8083/connectors/<ConnectorName>** に送信し、コネクターが存在することを確認します。以下の例では、**curl** を使用しています。

```
curl http://connect0.my-domain.com:8083/connectors
```

2. コネクターを削除するには、**DELETE** リクエストを **<KafkaConnectAddress>:8083/connectors** に送信します。以下の例では、**curl** を使用しています。

```
curl -X DELETE http://connect0.my-domain.com:8083/connectors/my-connector
```

3. GET リクエストを **<KafkaConnectAddress>:8083/connectors** に送信し、コネクターが削除されていることを確認します。以下の例では、**curl** を使用しています。

```
curl http://connect0.my-domain.com:8083/connectors
```

-

9.3. コネクタープラグイン

以下のコネクタープラグインは AMQ Streams に含まれています。

FileStreamSink Kafka トピックからデータを読み取り、データをファイルに書き込みます。

FileStreamSource ファイルからデータを読み取り、データを Kafka トピックに送信します。

必要に応じて、コネクタープラグインを追加できます。Kafka Connect は起動時に追加のコネクタープラグインを検索し、実行します。kafka Connect がプラグインを検索するパスを定義するには、**plugin.path configuration** オプションを設定します。

```
plugin.path=/opt/kafka/connector-plugins,/opt/connectors
```

plugin.path 設定オプションには、カンマ区切りのパスの一覧を含めることができます。

分散モードで Kafka Connect を実行する場合は、プラグインをすべてのワーカーノードで利用できるようにする必要があります。

9.4. 「ADDING CONNECTOR PLUGINS」

この手順では、追加のコネクタープラグインを追加する方法を説明します。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。

手順

1. **/opt/kafka/connector-plugins** ディレクトリーを作成します。

```
su - kafka
mkdir /opt/kafka/connector-plugins
```

2. **/opt/kafka/config/connect-standalone.properties** または **/opt/kafka/config/connect-distributed.properties** の Kafka Connect 設定ファイルを編集し、**plugin.path** オプションを **/opt/kafka/connector-plugins** に設定します。以下に例を示します。

```
plugin.path=/opt/kafka/connector-plugins
```

3. コネクタープラグインを **/opt/kafka/connector-plugins** にコピーします。

4. Kafka Connect ワーカーを起動または再起動します。

関連情報

- AMQ Streams のインストールに関する詳細は、「AMQ Streams のインストール」 を参照してください。
- AMQ Streams の設定に関する詳細は、「AMQ Streams の設定」 を参照してください。
- Kafka Connect をスタンドアロンモードで実行する方法については、「スタンドアロンモードでの Kafka Connect の実行」 を参照してください。

- 分散モードでの Kafka Connect の実行に関する詳細は、「分散 Kafka Connect の実行」 を参照してください。
- サポートされる Kafka Connect 設定オプションの完全リストは、付録F *Kafka Connect 設定パラメーター* を参照してください。

第10章 AMQ STREAMS の MIRRORMAKER 2.0 との使用

MirrorMaker 2.0 は、データセンター内またはデータセンター全体の 2 台以上の Kafka クラスター間でデータを複製するために使用されます。

クラスター全体のデータレプリケーションでは、以下が必要な状況がサポートされます。

- システム障害時のデータの復旧
- 分析用のデータの集計
- 特定のクラスターへのデータアクセスの制限
- レイテンシーを改善するための特定場所でのデータのプロビジョニング

注記

MirrorMaker 2.0 には、以前のバージョンの MirrorMaker ではサポートされない機能があります。ただし、レガシーモード で使用される MirrorMaker 2.0 を設定できます。

関連情報

- Apache Kafka ドキュメント
- 「MirrorMaker 2.0 のトレースの有効化」

10.1. MIRRORMAKER 2.0 データレプリケーション

MirrorMaker 2.0 はソースの Kafka クラスターからメッセージを消費して、ターゲットの Kafka クラスターに書き込みます。

MirrorMaker 2.0 は以下を使用します。

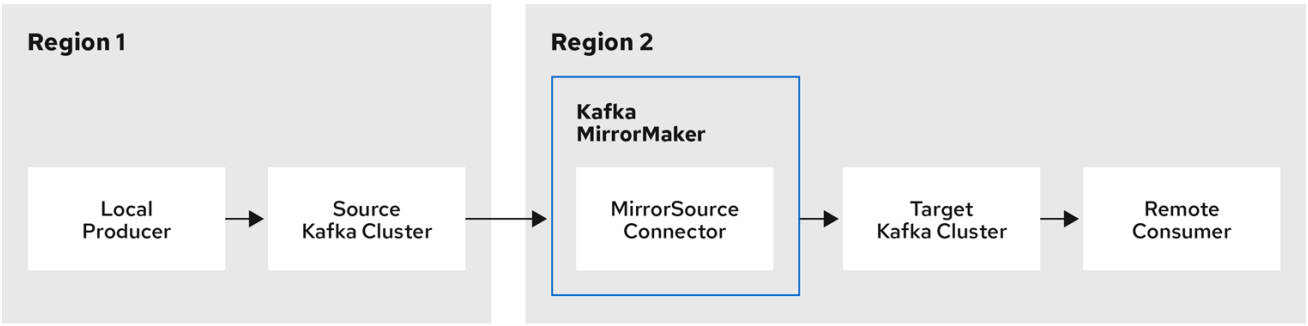
- ソースクラスターからデータを消費するソースクラスターの設定。
- データをターゲットクラスターに出力するターゲットクラスターの設定。

MirrorMaker 2.0 は Kafka Connect フレームワークをベースとし、**コネクター**によってクラスター間のデータ転送が管理されます。MirrorMaker 2.0 の **MirrorSourceConnector** は、ソースクラスターからターゲットクラスターにトピックを複製します。

あるクラスターから別のクラスターにデータを **ミラーリング** するプロセスは非同期です。推奨されるパターンは、ソース Kafka クラスターとともにローカルでメッセージが作成され、ターゲットの Kafka クラスターの近くでリモートで消費されることです。

MirrorMaker 2.0 は、複数のソースクラスターで使用できます。

図10.1 2 つのクラスターにおけるレプリケーション



デフォルトでは、ソースクラスターの新規トピックのチェックは 10 分ごとに行われます。**refresh.topics.interval.seconds** を **KafkaMirrorMaker2** リソースのソースコネクター設定に追加することで、頻度を変更できます。ただし、操作の頻度が増えると、全体的なパフォーマンスに影響する可能性があります。

10.2. クラスターの設定

active/passive または **active/active** クラスター設定で MirrorMaker 2.0 を使用できます。

- **active/active** 設定では、両方のクラスターがアクティブで、同じデータを同時に提供します。これは、地理的に異なる場所で同じデータをローカルで利用可能にする場合に便利です。
- **active/passive** 設定では、アクティブなクラスターからのデータはパッシブなクラスターで複製され、たとえば、システム障害時のデータ復旧などでスタンバイ状態を維持します。

プロデューサーとコンシューマーがアクティブなクラスターのみに接続することを前提とします。

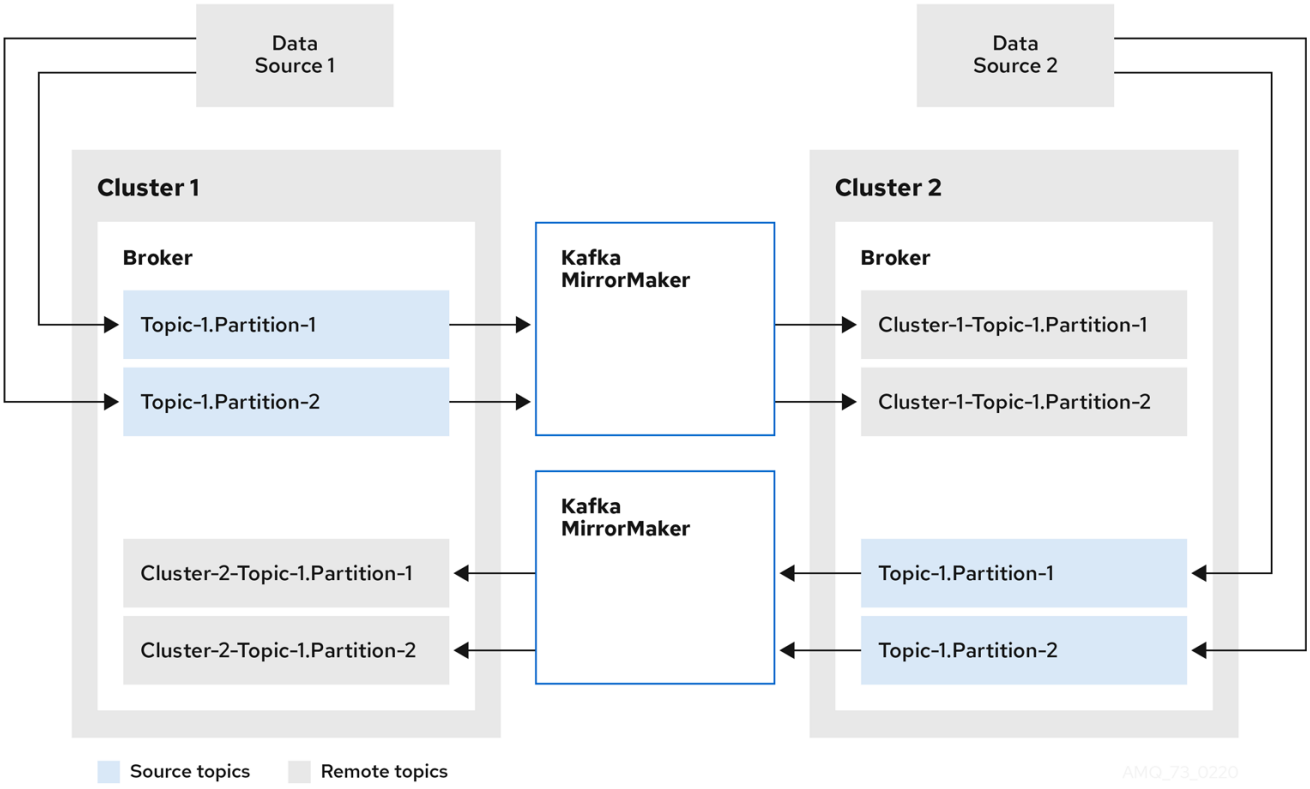
MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

10.2.1. 双方向レプリケーション (active/active)

MirrorMaker 2.0 アーキテクチャーでは、**active/active** クラスター設定で双方向レプリケーションがサポートされます。

各クラスターは、 **source** および **remote** トピックの概念を使用して、別のクラスターのデータを複製します。同じトピックが各クラスターに保存されるため、リモートトピックの名前がソースクラスターを表すように自動的に MirrorMaker 2.0 によって変更されます。元のクラスターの名前の先頭には、トピックの名前が追加されます。

図10.2 トピックの名前変更



ソースクラスターにフラグを付けると、トピックはそのクラスターに複製されません。

remote トピックを介したレプリケーションの概念は、データの集約が必要なアーキテクチャーの設定に役立ちます。コンシューマーは、同じクラスター内でソースおよびリモートトピックにサブスクライブできます。これに個別の集約クラスターは必要ありません。

10.2.2. 一方向レプリケーション (active/passive)

MirrorMaker 2.0 アーキテクチャーでは、**active/passive** クラスター設定で一方向レプリケーションがサポートされます。

active/passiveのクラスター設定を使用してバックアップを作成したり、データを別のクラスターに移行したりできます。この場合、リモートトピックの名前を自動的に変更したくないことがあります。

IdentityReplicationPolicy を **KafkaMirrorMaker2** リソースのソースコネクター設定に追加することで、名前の自動変更をオーバーライドできます。この設定が適用されると、トピックには元の名前が保持されます。

10.2.3. トピック設定の同期

トピック設定は、ソースクラスターとターゲットクラスター間で自動的に同期化されます。設定プロパティーを同期化することで、リバランスの必要性が軽減されます。

10.2.4. データの整合性

MirrorMaker 2.0 は、ソーストピックを監視し、設定変更をリモートトピックに伝播して、不足しているパーティションを確認および作成します。MirrorMaker 2.0 のみがリモートトピックに書き込みできます。

10.2.5. オフセットの追跡

MirrorMaker 2.0 では、**内部トピック**を使用してコンシューマーグループのオフセットを追跡します。

- **オフセット同期** トピックは、複製されたトピックパーティションのソースおよびターゲットオフセットをレコードメタデータからマッピングします。
- **チェックポイント** トピックは、各コンシューマーグループの複製されたトピックパーティションのソースおよびターゲットクラスターで最後にコミットされたオフセットをマッピングします。

チェックポイント トピックのオフセットは、設定によって事前定義された間隔で追跡されます。両方のトピックは、フェイルオーバー時に正しいオフセットの位置からレプリケーションの完全復元を可能にします。

MirrorMaker 2.0 は、**MirrorCheckpointConnector** を使用して、オフセット追跡の **チェックポイント**を生成します。

10.2.6. コンシューマーグループオフセットの同期

__consumer_offsets トピックには、各コンシューマーグループのコミットされたオフセットに関する情報が保存されます。オフセットの同期は、ソースクラスターのコンシューマーグループのコンシューマーオフセットをターゲットクラスターのコンシューマーオフセットに定期的に転送します。

オフセットの同期は、特に active/passive 設定で便利です。アクティブなクラスターがダウンした場合、コンシューマーアプリケーションはパッシブ (スタンバイ) クラスターに切り替え、最後に転送されたオフセットの位置からピックアップできます。

トピックオフセットの同期を使用するには、以下を行います。

- **sync.group.offsets.enabled** を **KafkaMirrorMaker2** リソースのチェックポイントコネクター設定に追加し、プロパティーを **true** に設定して同期を有効にします。同期はデフォルトで無効になっています。
- **IdentityReplicationPolicy** をソースおよびチェックポイントコネクター設定に追加し、ターゲットクラスターのトピックが元の名前を保持するようにします。

トピックオフセットの同期を機能させるため、ターゲットクラスターのコンシューマーグループは、ソースクラスターのグループと同じ ID を使用できません。

同期を有効にすると、ソースクラスターからオフセットの同期が定期的に行われます。**sync.group.offsets.interval.seconds** および **emit.checkpoints.interval.seconds** をチェックポイントコネクター設定に追加すると、頻度を変更できます。これらのプロパティーは、コンシューマーグループのオフセットが同期される頻度 (秒単位) と、オフセットを追跡するためにチェックポイントが生成される頻度を指定します。両方のプロパティーのデフォルトは 60 秒です。**refresh.groups.interval.seconds** プロパティーを使用して、新規コンシューマーグループをチェックする頻度を変更することもできます。デフォルトでは 10 分ごとに実行されます。

同期は時間ベースであるため、コンシューマーによってパッシブクラスターへ切り替えられると、一部のメッセージが重複する可能性があります。

10.2.7. 接続性チェック

ハートビート 内部トピックによって、クラスター間の接続性が確認されます。

ハートビート トピックは、ソースクラスターから複製されます。

ターゲットクラスターは、トピックを使用して以下を確認します。

- クラスター間の接続を管理するコネクターが稼働している。
- ソースクラスターが利用可能である。

MirrorMaker 2.0 は **MirrorHeartbeatConnector** を使用して、これらのチェックを実行する **ハートビート** を生成します。

10.3. ACL ルールの同期

AclAuthorizer が使用されている場合、ブローカーへのアクセスを管理する ACL ルールはリモートトピックにも適用されます。ソーストピックを読み取りできるユーザーは、そのリモートトピックを読み取りできます。

注記

OAuth 2.0 での承認は、このようなリモートトピックへのアクセスをサポートしません。

10.4. MIRRORMAKER 2.0 を使用した KAFKA クラスター間でのデータの同期

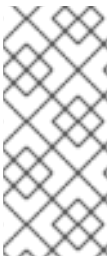
MirrorMaker 2.0 を使用して、設定を介して Kafka クラスター間のデータを同期します。

従来のモードで MirrorMaker 2.0 を実行すると、以前のバージョンの MirrorMaker は継続してサポートされます。

設定では以下を指定する必要があります。

- 各 Kafka クラスター
- TLS 認証を含む各クラスターの接続情報
- レプリケーションのフローおよび方向
  - クラスター対クラスター
  - トピック対トピック
- レプリケーションルール
- コミットされたオフセット追跡間隔

この手順では、プロパティーファイルの設定を作成し、MirrorMaker スクリプトファイルを使用して接続を設定するときにプロパティーを渡すことで、MirrorMaker 2.0 を実装する方法を説明します。



注記

MirrorMaker 2.0 は、Kafka Connect を使用してクラスター間のデータ転送を行います。Kafka は、データレプリケーションの MirrorMaker シンクおよびソースコネクターを提供します。MirrorMaker スクリプトの代わりにコネクターを使用する場合は、Kafka Connect クラスターでコネクターを設定する必要があります。詳細は Apache Kafka のドキュメントを参照してください。

作業を始める前に

設定ファイルのサンプルは、**./config/connect-mirror-maker.properties** にあります。

前提条件

- レプリケートしている各 Kafka クラスターノードのホストに AMQ Streams がインストールされている必要があります。

手順

1. テキストエディターでサンプルファイルを開くか、新しいプロパティーファイルを作成し、そのファイルを編集して、各 Kafka クラスターの接続情報およびレプリケーションフローを追加します。
   以下の例は、**cluster-1** と **cluster-2** の **2 つのクラスター** を接続する設定を示しています。クラスター名は **clusters** プロパティーで設定可能です。

```
clusters=cluster-1,cluster-2 (1)

cluster-1.bootstrap.servers=CLUSTER-NAME-kafka-bootstrap-PROJECT-NAME:443 (2)
cluster-1.security.protocol=SSL (3)
cluster-1.ssl.truststore.password=TRUSTSTORE-NAME
cluster-1.ssl.truststore.location=PATH-TO-TRUSTSTORE/truststore.cluster-1.jks
cluster-1.ssl.keystore.password=KEYSTORE-NAME
cluster-1.ssl.keystore.location=PATH-TO-KEYSTORE/user.cluster-1.p12_

cluster-2.bootstrap.servers=CLUSTER-NAME-kafka-bootstrap-<my-project>:443 (4)
cluster-2.security.protocol=SSL (5)
cluster-2.ssl.truststore.password=TRUSTSTORE-NAME
cluster-2.ssl.truststore.location=PATH-TO-TRUSTSTORE/truststore.cluster-2.jks_
cluster-2.ssl.keystore.password=KEYSTORE-NAME
cluster-2.ssl.keystore.location=PATH-TO-KEYSTORE/user.cluster-2.p12_

cluster-1->cluster-2.enabled=true (6)
cluster-1->cluster-2.topics=.* (7)
cluster-2->cluster-1.enabled=true (8)
cluster-2->cluster-1B->C.topics=.* (9)

replication.policy.separator=- (10)
sync.topic.acls.enabled=false (11)
refresh.topics.interval.seconds=60 (12)
refresh.groups.interval.seconds=60 (13)
```

(1) 各 Kafka クラスターはそのエイリアスで識別されます。

(2) **ブートストラップアドレス** およびポート 443 を使用した **cluster-1** の接続情報。どちらのクラスターもポート 443 を使用して、OpenShift **Routes** を使用して Kafka に接続します。

(3) **ssl.** プロパティーは、**cluster-1** の TLS 設定を定義します。

(4) **cluster-2** の接続情報

(5) **ssl.** プロパティーは、**cluster-2** の TLS 設定を定義します。

6 **cluster-1 クラスターから cluster-2 クラスターへのレプリケーションフローが有効になっている。**

7 **cluster-1 クラスターのすべてのトピックを cluster-2 クラスター** に複製します。

8 **cluster-2 クラスターから cluster-1 クラスターへのレプリケーションフローが有効になっている。**

9 **cluster-2 クラスターから特定のトピックを cluster-1 クラスター** に複製します。

10 リモートトピック名の変更に使用する区切り文字を定義します。

11 有効にすると、同期されたトピックに ACL が適用されます。デフォルトは **false** です。

12 新しいトピックを同期させる間隔。

13 同期する新規コンシューマーグループをチェックする間隔。

2. オプション： 必要な場合は、リモートトピックの名前の自動名前を上書きするポリシーを追加します。その名前の前にソースクラスターの名前を追加する代わりに、トピックが元の名前を保持します。
このオプションの設定は、active/passive バックアップおよびデータ移行に使用されます。

```
replication.policy.class=io.strimzi.kafka.connect.mirror.IdentityReplicationPolicy
```

3. オプション： コンシューマーグループのオフセットを同期する場合は、設定を追加して同期を有効にします。

```
refresh.groups.interval.seconds=60
sync.group.offsets.enabled=true 1
sync.group.offsets.interval.seconds=60 2
emit.checkpoints.interval.seconds=60 3
```

1 コンシューマーグループのオフセットを同期する任意設定。これは、active/passive 設定でのリカバリーに便利です。同期はデフォルトでは有効になっていません。

2 コンシューマーグループオフセットの同期が有効な場合は、同期の頻度を調整できます。

3 オフセット追跡のチェック頻度を調整します。オフセット同期の頻度を変更する場合、これらのチェックの頻度も調整する必要がある場合があります。

4. ターゲットクラスターで ZooKeeper および Kafka を起動します。

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

5. クラスター接続設定と、プロパティーファイルで定義したレプリケーションポリシーで MirrorMaker を起動します。

```
/opt/kafka/bin/connect-mirror-maker.sh /config/connect-mirror-maker.properties
```

MirrorMaker はクラスター間の接続を設定します。

6. 各ターゲットクラスターに対して、トピックが複製されていることを確認します。

```
/bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --list
```

10.5. レガシーモードでの MIRRORMAKER 2.0 の使用

この手順では、MirrorMaker 2.0 をレガシーモードで設定する方法を説明します。レガシーモードは、以前のバージョンの MirrorMaker をサポートします。

MirrorMaker スクリプト **/opt/kafka/bin/kafka-mirror-maker.sh** は、レガシーモードで MirrorMaker 2.0 を実行できます。

前提条件

従来のバージョンの MirrorMaker で現在使用しているプロパティーファイルが必要です。

- **/opt/kafka/config/consumer.properties**
- **/opt/kafka/config/producer.properties**

手順

1. MirrorMaker の **consumer.properties** および **producer.properties** ファイルを編集して、MirrorMaker 2.0 機能をオフにします。
以下に例を示します。

```
replication.policy.class=org.apache.kafka.mirror.LegacyReplicationPolicy (1)

refresh.topics.enabled=false (2)
refresh.groups.enabled=false
emit.checkpoints.enabled=false
emit.heartbeats.enabled=false
sync.topic.configs.enabled=false
sync.topic.acls.enabled=false
```

(1) 従来のバージョンの MirrorMaker をエミュレートします。

(2) 内部 **チェックポイント** および **ハートビートトピック** を含む、MirrorMaker 2.0 の機能が無効になる

2. 以前のバージョンの MirrorMaker で使用されたプロパティーファイルで MirrorMaker を保存し、MirrorMaker を再起動します。

```
su - kafka /opt/kafka/bin/kafka-mirror-maker.sh \
--consumer.config /opt/kafka/config/consumer.properties \
--producer.config /opt/kafka/config/producer.properties \
--num.streams=2
```

consumer プロパティーは、ソースクラスターの設定を提供し、**producer** プロパティーによってターゲットクラスターの設定が提供されます。

MirrorMaker はクラスター間の接続を設定します。

3. ターゲットクラスターで ZooKeeper および Kafka を起動します。

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

4. ターゲットクラスターの場合は、トピックが複製されていることを確認します。

```
/bin/kafka-topics.sh --bootstrap-server <BrokerAddress> --list
```

第11章 KAFKA クライアント

kafka-clients JAR ファイルには、Kafka Producer および Consumer API と Kafka AdminClient API が含まれています。

- プロデューサー API により、アプリケーションはデータを Kafka ブローカーに送信できます。
- Consumer API により、アプリケーションは Kafka ブローカーからデータを取得できます。
- AdminClient API は、トピック、ブローカー、他のコンポーネントなどの Kafka クラスターを管理するための機能を提供します。

11.1. KAFKA クライアントを依存関係として MAVEN プロジェクトに追加

この手順では、AMQ Streams Java クライアントを依存関係として Maven プロジェクトに追加する方法を説明します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリーを **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!-- ... -->

    <repositories>
        <repository>
            <id>redhat-maven</id>
            <url>https://maven.repository.redhat.com/ga/</url>
        </repository>
    </repositories>

    <!-- ... -->

</project>
```

2. **pom.xml** ファイルの **<dependencies>** セクションにクライアントを追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!-- ... -->
```

```
    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.7.0.redhat-00005</version>
        </dependency>
    </dependencies>

    <!-- ... -->
</project>
```

3. Maven プロジェクトをビルドします。

第12章 KAFKA STREAMS API の概要

Kafka Streams API を使用すると、アプリケーションは 1 つ以上の入力ストリームからデータを受信でき、マッピング、フィルタリングや参加などの複雑な操作を実行し、1 つ以上の出力ストリームに結果を記述できます。これは、Red Hat Maven リポジトリーで利用可能な **kafka-streams** JAR パッケージに含まれます。

12.1. KAFKA STREAMS API を依存関係として MAVEN プロジェクトに追加

この手順では、AMQ Streams Java クライアントを依存関係として Maven プロジェクトに追加する方法を説明します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリーを **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!-- ... -->

    <repositories>
        <repository>
            <id>redhat-maven</id>
            <url>https://maven.repository.redhat.com/ga/</url>
        </repository>
    </repositories>

    <!-- ... -->

</project>
```

2. **pom.xml** ファイルの **<dependencies>** セクションに **kafka-streams** を追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!-- ... -->

    <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
```

```
            <version>2.7.0.redhat-00005</version>
        </dependency>
    </dependencies>

    <!-- ... -->
</project>
```

3. Maven プロジェクトをビルドします。

第13章 KAFKA BRIDGE

本章では、Red Hat Enterprise Linux での AMQ Streams Kafka Bridge の概要と、その REST API を使用して AMQ Streams と対話する方法を説明します。ローカル環境で Kafka Bridge を試すには、本章で後述する 「Kafka Bridge クイックスタート」 を参照してください。

関連情報

- リクエストおよび応答の例など、API ドキュメントを確認するには、「 Kafka Bridge API reference 」を参照してください。
- 分散トレーシング向けに Kafka Bridge を設定するには、「Kafka Bridge のトレースの有効化」を参照してください。

13.1. KAFKA BRIDGE の概要

Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターとの対話を可能にする RESTful インターフェースが提供されます。また、クライアントアプリケーションによる Kafka プロトコルの変換は必要なく、Web API コネクションの利点が AMQ Streams に提供されます。

API には**consumers** と **topics**の 2 つの主なリソースがあります。つまり、Kafka クラスターでコンシューマーおよびプロデューサーと対話するためにエンドポイント経由で公開され、アクセスが可能になります。リソースと関係があるのは Kafka ブリッジのみで、Kafka に直接接続されたコンシューマーやプロデューサーとは関係はありません。

HTTP リクエスト
Kafka Bridge は、以下の方法で Kafka クラスターへの HTTP リクエストをサポートします。

- トピックにメッセージを送信する。
- トピックからメッセージを取得する。
- トピックのパーティションリストを取得する。
- コンシューマーを作成および削除する。
- コンシューマーをトピックにサブスクライブし、このようなトピックからメッセージを受信できるようにする。
- コンシューマーがサブスクライブしているトピックの一覧を取得する。
- トピックからコンシューマーのサブスクライブを解除する。
- パーティションをコンシューマーに割り当てる。
- コンシューマーオフセットの一覧をコミットする。
- パーティションで検索して、コンシューマーが最初または最後のオフセットの位置、または指定のオフセットの位置からメッセージを受信できるようにする。

上記の方法で、JSON 応答と HTTP 応答コードのエラー処理を行います。メッセージは JSON またはバイナリー形式で送信できます。

クライアントは、ネイティブの Kafka プロトコルを使用する必要なくメッセージを生成して使用できます。

AMQ Streams のインストールと同様に、Red Hat Enterprise Linux にインストールする Kafka Bridge ファイルをダウンロードします。「Kafka Bridge アーカイブのダウンロード」 を参照してください。

KafkaBridge リソースのホストおよびポートの設定に関する詳細は、「Kafka Bridge プロパティーの設定」 を参照してください。

13.1.1. 認証および暗号化

HTTP クライアントと Kafka Bridge 間の認証および暗号化はまだサポートされていません。そのため、クライアントから Kafka Bridge に送信されるリクエストは以下のようになります。

- 暗号化されず、HTTPS ではなく HTTP を使用する必要がある。
- 認証なしで送信される。

Kafka Bridge と Kafka クラスター間の TLS または SASL ベースの認証 を設定できます。

プロパティーファイルを介して認証に Kafka Bridge を設定し ます。

13.1.2. Kafka Bridge へのリクエスト

データ形式と HTTP ヘッダーを指定し、有効なリクエストが Kafka Bridge に送信されるようにします。

API リクエストおよびレスポンス本文は、常に JSON としてエンコードされます。

13.1.2.1. コンテンツタイプヘッダー

すべてのリクエストに対して **Content-Type** ヘッダーを提出する必要があります。唯一の例外は、**POST** リクエストボディーが空であることです。**Content-Type** ヘッダーを追加すると、リクエストは失敗します。

コンシューマー操作（**/consumers** エンドポイント）およびプロデューサー操作（**/topics** エンドポイント）には異なる **Content-Type** ヘッダーが必要です。

コンシューマー操作の Content-Type ヘッダー

埋め込みデータ形式に関係なく、リクエストボディーにデータ が含まれる場合は、コンシューマー操作の **POST** リクエストに以下の **Content-Type** ヘッダーが含まれている必要があります。

```
Content-Type: application/vnd.kafka.v2+json
```

プロデューサー操作の Content-Type ヘッダー

プロデューサー操作の実行時に、**POST** リクエストは、生成されたメッセージの **埋め込みデータ形式** を指定する **Content-Type** ヘッダーを提供する必要があります。これは、**json** または **binary** のいずれかになります。

表13.1 データ形式の Content-Type ヘッダー

埋め込みデータ形式	Content-Type ヘッダー
JSON	**Content-Type: application/vnd.kafka.json.v2+json**

埋め込みデータ形式	Content-Type ヘッダー
バイナリー	**Content-Type: application/vnd.kafka.binary.v2+json**

次のセクションで説明どおり、埋め込みデータ形式はコンシューマーごとに設定されます。

POST リクエストに空のボディーがある場合は、**Content-Type** を設定**しない**でください。空のボディーを使用して、デフォルト値のコンシューマーを作成できます。

13.1.2.2. 埋め込みデータ形式

埋め込みデータ形式は、Kafka メッセージが Kafka Bridge によりプロデューサーからコンシューマーに HTTP で送信される際の形式です。サポートされる埋め込みデータ形式には、JSON またはバイナリーの 2 つがサポートされます。

/consumers/groupid エンドポイントを使用してコンシューマーを作成する場合、**POST** リクエスト本文で JSON またはバイナリーいずれかの埋め込みデータ形式を指定する必要があります。これは、リクエストボディーの **format** フィールドで指定します。以下に例を示します。

```
{
  "name": "my-consumer",
  "format": "binary", (1)
...
}
```

(1) バイナリー埋め込みデータ形式。

コンシューマーの埋め込みデータ形式が指定されていない場合は、バイナリー形式が設定されます。

コンシューマーの作成時に指定する埋め込みデータ形式は、コンシューマーが消費する Kafka メッセージのデータ形式と一致する必要があります。

バイナリー埋め込みデータ形式を指定する場合は、以降のプロデューサーリクエストで、リクエスト本文にバイナリーデータが Base64 でエンコードされた文字列として含まれる必要があります。たとえば、**POST** リクエストを **/topics/topicname** エンドポイントに送信してメッセージを送信する場合は、**value** を Base64 でエンコードする必要があります。

```
{
  "records": [
    {
      "key": "my-key",
      "value": "ZWR3YXJkdGhldGhyZWVsZWdnZWRjYXQ="
    },
  ]
}
```

プロデューサーリクエストは、埋め込みデータ形式に対応する **Content-Type** ヘッダーも提供する必要があります (例: **Content-Type: application/vnd.kafka.binary.v2+json**)。

13.1.2.3. メッセージの形式

/topics エンドポイントを使用してメッセージを送信する場合は、**records** パラメーターでリクエストボディーにメッセージペイロードを入力します。

records パラメーターには、以下のオプションフィールドを含めることができます。

- メッセージの **key**
- メッセージの **value**
- 宛先の **partition**
- メッセージの **headers**

トピックへの POST リクエストの例

```
curl -X POST \
  http://localhost:8080/topics/my-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
        {
            "key": "my-key",
            "value": "sales-lead-0001"
            "partition": 2
            "headers": [
              {
                "key": "key1",
                "value": "QXBhY2hlIEthZmthIGlzIHRoZSBib21iIQ==" 1
              }
            ]
        },
    ]
}'
```

1 バイナリー形式のヘッダー値。Base64 としてエンコードされます。

13.1.2.4. Accept ヘッダー

コンシューマーを作成したら、以降のすべての GET リクエストには **Accept** ヘッダーが以下のような形式で含まれる必要があります。

```
Accept: application/vnd.kafka.embedded-data-format.v2+json
```

embedded-data-format は、**json** または **binary** のどちらかです。

たとえば、サブスクライブされたコンシューマーのレコードを JSON 埋め込みデータ形式で取得する場合、この Accept ヘッダーが含まれるようにします。

```
Accept: application/vnd.kafka.json.v2+json
```

13.1.3. Kafka Bridge のロガーの設定

AMQ Streams の Kafka ブリッジを使用すると、関連する OpenAPI 仕様によって定義される操作ごとに異なるログレベルを設定できます。

それぞれの操作には、ブリッジが HTTP クライアントから要求を受信する対応の API エンドポイントがあります。各エンドポイントのログレベルを変更すると、送受信 HTTP リクエストに関する詳細なログ情報を生成することができます。

ロガーは **log4j.properties** ファイルで定義されます。このファイルには **healthy** および **ready** エンドポイントの以下のデフォルト設定が含まれています。

```
log4j.logger.http.openapi.operation.healthy=WARN, out
log4j.additivity.http.openapi.operation.healthy=false
log4j.logger.http.openapi.operation.ready=WARN, out
log4j.additivity.http.openapi.operation.ready=false
```

その他すべての操作のログレベルは、デフォルトで **INFO** に設定されます。ロガーは以下のようにフォーマットされます。

```
log4j.logger.http.openapi.operation.<operation-id>
```

ここで、**<operation-id>** は特定の操作の識別子になります。以下は、OpenAPI 仕様で定義されたオペレーションの一覧です。

- **createConsumer**
- **deleteConsumer**
- **subscribe**
- **unsubscribe**
- **poll**
- **assign**
- **commit**
- **send**
- **sendToPartition**
- **seekToBeginning**
- **seekToEnd**
- **seek**
- **healthy**
- **ready**
- **openapi**

13.1.4. Kafka Bridge API リソース

リクエストやレスポンスの例などを含む REST API エンドポイントおよび説明の完全リストは、「Kafka Bridge API reference 」を参照してください。

13.1.5. Kafka Bridge アーカイブのダウンロード

AMQ Streams Kafka Bridge の zip ディストリビューションは、Red Hat の Web サイトからダウンロードできます。

手順

- カスタマーポータルから最新バージョンの Red Hat AMQ Streams Kafka Bridge アーカイブをダウンロードします。

13.1.6. Kafka Bridge プロパティーの設定

この手順では、AMQ Streams Kafka Bridge によって使用される Kafka および HTTP コネクションプロパティーを設定する方法を説明します。

Kafka 関連のプロパティーに適切な接頭辞を使用して、Kafka Bridge を他の Kafka クライアントとして設定します。

- **kafka.** サーバー接続やセキュリティーなどのプロデューサーおよびコンシューマーに適用される一般的な設定。
- **kafka.consumer.** コンシューマーにのみ渡されたコンシューマー固有の設定の場合
- **kafka.producer.** プロデューサー固有の設定の場合は、プロデューサーにのみ渡されます。

HTTP プロパティーは、Kafka クラスターへの HTTP アクセスを有効にする他に、CPRS (Cross-Origin Resource Sharing) により Kafka Bridge のアクセス制御を有効化または定義する機能を提供します。CORS は、複数のオリジンから指定のリソースにブラウザーでアクセスできるようにする HTTP メカニズムです。CORS を設定するには、許可されるリソースオリジンのリストと、それらにアクセスする HTTP メソッドを定義します。リクエストの追加の HTTP ヘッダーには Kafka クラスターへのアクセスが許可されるオリジンが記述 されています。

前提条件

- AMQ Streams がホストにインストールされていること。
- Kafka Bridge インストールアーカイブがダウンロードされます。

手順

1. AMQ Streams Kafka Bridge インストールアーカイブで提供される **application.properties** ファイルを編集します。
   プロパティーファイルを使用して Kafka および HTTP 関連のプロパティーを指定し、分散トレーシングを有効にします。

   a. Kafka コンシューマーおよびプロデューサーに固有のプロパティーなど、標準の Kafka 関連のプロパティーを設定します。
   以下を使用します。

      - **kafka.bootstrap.servers** Kafka クラスターへのホスト/ポート接続を定義します。
      - **kafka.producer.acks** HTTP クライアントに承認を提供する

- **kafka.consumer.auto.offset.reset** Kafka でオフセットのリセット方法を決定するため。
  Kafka プロパティーの設定に関する詳細は Apache Kafka の Web サイト を参照してください。

b. HTTP 関連のプロパティーを設定し、Kafka クラスターへの HTTP アクセスを有効にします。
以下に例を示します。

```
http.enabled=true
http.host=0.0.0.0
http.port=8080 1
http.cors.enabled=true 2
http.cors.allowedOrigins=https://strimzi.io 3
http.cors.allowedMethods=GET,POST,PUT,DELETE,OPTIONS,PATCH 4
```

1. 8080 番ポートで Kafka Bridge をリッスンするデフォルトの HTTP 設定。
2. CORS を有効にするには、**true** に設定します。
3. 許可される CORS オリジンのコンマ区切りリスト。URL または Java 正規表現を使用できます。
4. CORS で許可される HTTP メソッドのコンマ区切りリスト。

c. 分散トレーシングを有効または無効にします。

```
bridge.tracing=jaeger
```

プロパティーからコードコメントを削除して、分散トレースを有効にします。

関連情報

- *16章分散トレーシング*
- 「Kafka Bridge のトレースの有効化」

13.1.7. Kafka Bridge のインストール

以下の手順に従って、AMQ Streams Kafka Bridge を Red Hat Enterprise Linux にインストールします。

前提条件

- AMQ Streams がホストにインストールされていること。
- Kafka Bridge インストールアーカイブがダウンロードされます。
- Kafka Bridge 設定プロパティーが設定されます。

手順

1. AMQ Streams Kafka Bridge インストールアーカイブを任意のディレクトリーに展開していない場合は、そのディレクトリーに展開してください。

2. 設定プロパティーをパラメーターとして使用して、Kafka Bridge スクリプトを実行します。以下に例を示します。

```
./bin/kafka_bridge_run.sh --config-file=_path_/configfile.properties
```

3. インストールがログで成功したことを確認します。

```
HTTP-Kafka Bridge started and listening on port 8080
HTTP-Kafka Bridge bootstrap servers localhost:9092
```

13.2. KAFKA BRIDGE クイックスタート

このクイックスタートを使用して、Red Hat Enterprise Linux で AMQ Streams Kafka Bridge を試すことができます。以下の方法について説明します。

- Kafka Bridge のインストール
- Kafka クラスターのトピックおよびパーティションへのメッセージを生成する。
- Kafka Bridge コンシューマーを作成する。
- 基本的なコンシューマー操作を実行する (たとえば、コンシューマーをトピックにサブスクライブする、生成したメッセージを取得するなど)。

このクイックスタートでは、HTTP リクエストはターミナルにコピーおよび貼り付けできる curl コマンドを使用します。

前提条件を確認し、本章に指定されている順序でタスクを行うようにしてください。

データ形式について

このクイックスタートでは、バイナリーではなく JSON 形式でメッセージを生成および消費します。リクエスト例で使用されるデータ形式および HTTP ヘッダーの詳細は、「認証および暗号化」 を参照してください。

クイックスタートの前提条件

- AMQ Streams がホストにインストールされていること。
- AMQ Streams クラスターが稼働している必要があります。
- Kafka Bridge インストールアーカイブがダウンロードされます。

13.2.1. Kafka Bridge のローカルでのデプロイメント

AMQ Streams Kafka Bridge のインスタンスをホストにデプロイします。インストールアーカイブで提供される **application.properties** ファイルを使用して、デフォルトの設定を適用します。

手順

1. **application.properties** ファイルを開き、デフォルトの **HTTP related settings** が定義されていることを確認します。

```
http.enabled=true
http.host=0.0.0.0
http.port=8080
```

これにより、Kafka Bridge がポート 8080 でリクエストをリッスンするように設定されます。

2. 設定プロパティーをパラメーターとして使用して、Kafka Bridge スクリプトを実行します。

```
./bin/kafka_bridge_run.sh --config-file=<path>/application.properties
```

次のステップ

- トピックおよびパーティションへのメッセージを生成 します。

13.2.2. トピックおよびパーティションへのメッセージの作成

topics エンドポイントを使用して、JSON 形式で トピック にメッセージを生成します。

以下のように、リクエスト本文でメッセージの宛先パーティションを指定できます。partitions エンドポイントは、全メッセージの単一の宛先パーティションをパスパラメーターとして指定する代替方法を提供します。

手順

1. **kafka-topics.sh** ユーティリティーを使用して Kafka トピックを作成します。

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic bridge-quickstart-topic --partitions 3 --replication-factor 1 --config retention.ms=7200000 --config segment.bytes=1073741824
```

3 つのパーティションを指定します。

2. トピックが作成されたことを確認します。

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic bridge-quickstart-topic
```

3. Kafka Bridge を使用して、作成したトピックに 3 つのメッセージを生成します。

```
curl -X POST \
  http://localhost:8080/topics/bridge-quickstart-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
        {
            "key": "my-key",
            "value": "sales-lead-0001"
        },
        {
            "value": "sales-lead-0002",
            "partition": 2
        },
        {
            "value": "sales-lead-0003"
```

```
      }
    ]
}'
```

- **sales-lead-0001** は、キーのハッシュに基づいてパーティションに送信されます。
- **sales-lead-0002** は、パーティション 2 に直接送信されます。
- **sales-lead-0003** は、ラウンドロビン方式を使用して **bridge-quickstart-topic** トピックのパーティションに送信されます。

4. リクエストが正常に行われると、Kafka Bridge は **offsets** アレイを **200** (OK)コードと **application/vnd.kafka.v2+json** の **content-type** ヘッダーとともに返します。各メッセージで、**offsets** アレイは以下を記述します。
   - メッセージが送信されたパーティション。
   - パーティションの現在のメッセージオフセット。

   応答の例

```
#...
{
  "offsets":[
    {
      "partition":0,
      "offset":0
    },
    {
      "partition":2,
      "offset":0
    },
    {
      "partition":0,
      "offset":1
    }
  ]
}
```

次のステップ

トピックおよびパーティションへのメッセージを作成したら、Kafka Bridge コンシューマーを作成します。

その他のリソース

- API リファレンスドキュメントの「POST /topics/{topicname} 」
- API リファレンスドキュメントの「POST /topics/{topicname}/partitions/{partitionid} 」

13.2.3. Kafka Bridge コンシューマーの作成

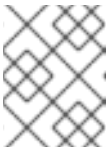
Kafka クラスターで何らかのコンシューマー操作を実行するには、まず consumers エンドポイントを使用して コンシューマー を作成する必要があります。コンシューマーは **Kafka Bridge コンシューマー** と呼ばれます。

手順

1. **bridge-quickstart-consumer-group** という名前の新しいコンシューマーグループに Kafka Bridge コンシューマーを作成します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "name": "bridge-quickstart-consumer",
    "auto.offset.reset": "earliest",
    "format": "json",
    "enable.auto.commit": false,
    "fetch.min.bytes": 512,
    "consumer.request.timeout.ms": 30000
  }'
```

- コンシューマーには **bridge-quickstart-consumer** という名前を付け、埋め込みデータ形式は **json** として設定します。
- コンシューマーはログへのオフセットに自動でコミットしません。これは、**enable.auto.commit** が **false** に設定されているからです。このクイックスタートでは、オフセットを跡で手作業でコミットします。



注記

リクエスト本文にコンシューマー名を指定しない場合、Kafka Bridge はランダムなコンシューマー名を生成します。

リクエストが正常に行われると、Kafka Bridge はレスポンス本文でコンシューマー ID(**instance_id**)とベース URL(**base_uri**)を **200** (OK)コードとともに返します。

応答の例

```
#...
{
  "instance_id": "bridge-quickstart-consumer",
  "base_uri":"http://<bridge-name>-bridge-service:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer"
}
```

2. ベース URL (**base_uri**) をコピーし、このクイックスタートの他のコンシューマー操作で使用します。

次のステップ

上記で作成した Kafka Bridge コンシューマーをトピックにサブスクライブできます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid} 」

13.2.4. Kafka Bridge コンシューマーのトピックへのサブスクライブ

subscription エンドポイントを使用して、Kafka Bridge コンシューマーを 1 つ以上のトピックにサブスクライブします。サブスクライブすると、コンシューマーはトピックに生成されたすべてのメッセージの受信を開始します。

手順

- 前述の「トピックおよびパーティションへのメッセージの作成」の手順ですでに作成した **bridge-quickstart-topic** トピックに、コンシューマーをサブスクライブします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/subscription \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "topics": [
        "bridge-quickstart-topic"
    ]
}'
```

topics アレイには、例のような単一のトピック、または複数のトピックを含めることができます。正規表現に一致する複数のトピックにコンシューマーをサブスクライブする場合は、**topics** アレイの代わりに **topic_pattern** 文字列を使用できます。

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

次のステップ

Kafka Bridge コンシューマーをトピックにサブスクライブしたら、コンシューマーからメッセージを取得できます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/subscription」

13.2.5. Kafka Bridge コンシューマーからの最新メッセージの取得

records エンドポイントからデータをリクエストして、Kafka Bridge コンシューマーから最新メッセージを取得します。実稼働環境では、HTTP クライアントはこのエンドポイントを繰り返し (ループで) 呼び出すことができます。

手順

1. 「トピックおよびパーティションへのメッセージの生成」の説明に従い、Kafka Bridge コンシューマーに新たなメッセージを生成します。
2. **GET** リクエストを **records** エンドポイントに送信します。

```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

Kafka Bridge コンシューマーを作成し、サブスクライブすると、最初の GET リクエストによって空のレスポンスが返されます。これは、ポーリング操作はリバランスプロセスをトリガーしてパーティションを割り当てます。

3. 手順 2 を繰り返し、Kafka Bridge コンシューマーからメッセージを取得します。
Kafka Bridge は、レスポンス本文でメッセージのアレイ（トピック名、キー、値、パーティション、オフセットの記述）を **200** (OK)コードとともに返します。メッセージはデフォルトで最新のオフセットから取得されます。

```
HTTP/1.1 200 OK
content-type: application/vnd.kafka.json.v2+json
#...
[
  {
    "topic":"bridge-quickstart-topic",
    "key":"my-key",
    "value":"sales-lead-0001",
    "partition":0,
    "offset":0
  },
  {
    "topic":"bridge-quickstart-topic",
    "key":null,
    "value":"sales-lead-0003",
    "partition":0,
    "offset":1
  },
#...
```

注記

空のレスポンスが返される場合は、「トピックおよびパーティションへのメッセージの生成」の説明に従い、コンシューマーに対して追加のレコードを生成し、メッセージの取得を再試行します。

次のステップ

Kafka Bridge コンシューマーからメッセージを取得したら、ログへのオフセットをコミットします。

その他のリソース

- API リファレンスドキュメントの「GET /consumers/{groupid}/instances/{name}/records 」

13.2.6. ログへのオフセットのコミット

オフセットエンドポイントを使用して、Kafka Bridge コンシューマーによって受信されるすべてのメッセージのログに手動でオフセットをコミットします。この操作が必要なのは、前述の「Kafka Bridge コンシューマーの作成」で作成した Kafka Bridge コンシューマー が **enable.auto.commit** の設定で **false** に指定されているからです。

手順

- **bridge-quickstart-consumer** のオフセットをログにコミットします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/offsets
```

リクエスト本文は送信されないので、オフセットはコンシューマーによって受信されたすべて

のレコードに対してコミットされます。この代わりに、リクエスト本文に、オフセットをコミットするトピックおよびパーティションを指定するアレイ(OffsetCommitSeekList)を含めることができます。

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

次のステップ

オフセットをログにコミットしたら、オフセットをシークのエンドポイントを試行します。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/offsets 」

13.2.7. パーティションのオフセットのシーク

positions エンドポイントを使用して、Kafka Bridge コンシューマーを設定し、特定のオフセットからパーティションのメッセージを取得するようにし、最新のオフセットから取得します。これは Apache Kafka では、シーク操作と呼ばれます。

手順

1. **quickstart-bridge-topic** トピックで、パーティション 0 の特定のオフセットをシークします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "offsets": [
        {
            "topic": "bridge-quickstart-topic",
            "partition": 0,
            "offset": 2
        }
    ]
}'
```

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

2. **GET** リクエストを **records** エンドポイントに送信します。

```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

Kafka Bridge は、シークしたオフセットからのメッセージを返します。

3. 同じパーティションの最後のオフセットをシークし、デフォルトのメッセージ取得動作を復元します。この時点で、positions/end エンドポイントを使用します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions/end \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "partitions": [
```

```
        {
          "topic": "bridge-quickstart-topic",
          "partition": 0
        }
    ]
}'
```

リクエストが正常に行われると、Kafka Bridge は別の **204 No Content** コードを返します。

注記

positions/beginning エンドポイントを使用して、1 つ以上のパーティションの最初のオフセットをシークすることもできます。

次のステップ

このクイックスタートでは、AMQ Streams Kafka Bridge を使用して Kafka クラスターの一般的な操作をいくつか実行しました。これで、すでに作成した Kafka Bridge コンシューマーを削除 できます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions」
- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions/beginning」
- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions/end」

13.2.8. Kafka Bridge コンシューマーの削除

最後に、このクイックスタートを通して使用した Kafa Bridge コンシューマーを削除します。

手順

- **DELETE** リクエストを instances エンドポイントに送信し、Kafka Bridge コンシューマーを削除します。

```
curl -X DELETE http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer
```

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

その他のリソース

- API リファレンスドキュメントの「DELETE /consumers/{groupid}/instances/{name}」

第14章 KERBEROS(GSSAPI)認証の使用

AMQ Streams は、Kafka クラスターへのシングルサインオンアクセスをセキュアにするために、Kerberos(GSSAPI)認証プロトコルの使用をサポートします。GSSAPI は、基礎となる実装の変更からアプリケーションを調整する、Kerberos 機能の API ラッパーです。

Kerberos は、対称暗号化と信頼できるサードパーティー(Kerberos Key Distribution Centre)を使用してクライアントとサーバーが相互に認証できるようにするネットワーク認証システムです。

14.1. KERBEROS(GSSAPI)認証を使用するように AMQ STREAMS を設定する

この手順では、Kafka クライアントが Kerberos(GSSAPI)認証を使用して Kafka および ZooKeeper にアクセスできるように AMQ Streams を設定する方法を説明します。

この手順では、Kerberos **krb5** リソースサーバーが Red Hat Enterprise Linux ホストに設定されていることを前提としています。

この手順では、設定方法の例を示しています。

1. サービスプリンシパル
2. Kerberos ログインを使用する Kafka ブローカー
3. Kerberos ログインを使用するための ZooKeeper
4. Kerberos 認証を使用して Kafka にアクセスするためのプロデューサーおよびコンシューマークライアント

この手順では、プロデューサーおよびコンシューマークライアントの追加設定とともに、単一のホストで単一の ZooKeeper および Kafka インストールに対して Kerberos セットを説明します。

前提条件

Kerberos クレデンシャルの認証および承認するように Kafka および ZooKeeper を設定するには、以下が必要です。

- Kerberos サーバーへのアクセス
- 各 Kafka ブローカーホストの Kerberos クライアント

Kerberos サーバーを設定し、ブローカーホストにクライアントを 設定する手順についての詳細は、「Kerberos on RHEL のセットアップ」を参照してください。

Kerberos のデプロイ方法は、お使いのオペレーティングシステムによって異なります。Red Hat は、Red Hat Enterprise Linux で Kerberos を設定する際に Identity Management(IdM)を使用することを推奨します。Oracle または IBM JDK を使用する場合は、JCE(Java Cryptography Extension)をインストールする必要があります。

- Oracle JCE
- IBM JCE

認証用のサービスプリンシパルの追加

Kerberos サーバーから、ZooKeeper、Kafka ブローカー、および Kafka プロデューサークライアントのサービスプリンシパル（ユーザー）を作成します。

サービスプリンシパルには SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-REALM の形式にする必要があります。

1. Kerberos KDC でプリンシパルキーを保存するサービスプリンシパルおよびキータブを作成します。
   以下に例を示します。

   - **zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
     ZooKeeper サービスプリンシパルは、Kafka **config/server.properties** ファイルの **zookeeper.connect** 設定と同じホスト名を指定する必要があります。

     ```
     zookeeper.connect=node1.example.redhat.com:2181
     ```

     ホスト名が同じでない場合、**localhost** が使用され、認証に失敗します。

2. ホストにディレクトリーを作成し、キータブファイルを追加します。
   以下に例を示します。

   ```
   /opt/kafka/krb5/zookeeper-node1.keytab
   /opt/kafka/krb5/kafka-node1.keytab
   /opt/kafka/krb5/kafka-producer1.keytab
   /opt/kafka/krb5/kafka-consumer1.keytab
   ```

3. **kafka** ユーザーがディレクトリーにアクセスできることを確認します。

   ```
   chown kafka:kafka -R /opt/kafka/krb5
   ```

Kerberos ログインを使用するように ZooKeeper を設定する

zookeeper に作成しておいたユーザープリンシパルとキータブを使用して認証に Kerberos Key Distribution Center(KDC)を使用するように ZooKeeper を設定します。

1. **opt/kafka/config/jaas.conf** ファイルを作成または変更し、ZooKeeper のクライアントおよびサーバー操作をサポートします。

   ```
   Client {
       com.sun.security.auth.module.Krb5LoginModule required debug=true
       useKeyTab=true 1
       storeKey=true 2
       useTicketCache=false 3
       keyTab="/opt/kafka/krb5/zookeeper-node1.keytab" 4
       principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM"; 5
   };

   Server {
       com.sun.security.auth.module.Krb5LoginModule required debug=true
       useKeyTab=true
       storeKey=true
       useTicketCache=false
   ```

```
    keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
    principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumServer {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
    principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumLearner {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
    principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

1. キータブからプリンシパルキーを取得するには、**true** に設定します。
2. プリンシパルキーを保存するには **true** に設定します。
3. チケットキャッシュから Ticket Granting Ticket(TGT)を取得するには、**true** に設定します。
4. **keyTab** プロパティーは、Kerberos KDC からコピーしたキータブファイルの場所を参照します。場所とファイルは、**kafka** ユーザーが読み取り可能でなければなりません。
5. **principal** プロパティーは、KDC ホストで作成した完全修飾プリンシパル名と一致するように設定されます。形式は **SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-NAME** です。

2. **opt/kafka/config/zookeeper.properties** を編集して、更新された JAAS 設定を使用します。

```
# ...

requireClientAuthScheme=sasl
jaasLoginRenew=3600000 (1)
kerberos.removeHostFromPrincipal=false (2)
kerberos.removeRealmFromPrincipal=false (3)
quorum.auth.enableSasl=true (4)
quorum.auth.learnerRequireSasl=true (5)
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner (6)
quorum.auth.server.loginContext=QuorumServer
quorum.auth.kerberos.servicePrincipal=zookeeper/_HOST (7)
quorum.cnxn.threads.size=20
```

1. チケットの更新間隔に合わせて調整できるログイン更新の頻度をミリ秒単位で制御します。デフォルトは 1 時間です。
2. ホスト名がログインプリンシパル名の一部として使用されるかどうかを指定します。クラスターのすべてのノードで単一のキータブを使用する場合、これは **true** に設定されま

す。ただし、トラブルシューティングを行うために、各ブローカーホストの個別のキータブおよび完全修飾プリンシパルを生成することが推奨されます。

3. Kerberos ネゴシエーションのプリンシパル名からレルム名が削除されるかどうかを制御します。この設定は **false** に設定することが推奨されます。

4. ZooKeeper サーバーおよびクライアントの SASL 認証メカニズムを有効にします。

5. **RequireSasl** プロパティーは、マスターの選択などのクォーラムイベントに SASL 認証が必要であるかどうかを制御します。

6. **loginContext** プロパティーは、指定されたコンポーネントの認証設定に使用される JAAS 設定のログインコンテキストの名前を特定します。loginContext 名は、**opt/kafka/config/jaas.conf** ファイルの関連セクションの名前に対応します。

7. 識別に使用されるプリンシパル名を形成するのに使用される命名規則を制御します。プレースホルダー **_HOST** は、実行時に **server.1** プロパティーによって定義されるホスト名に自動的に解決されます。

3. JVM パラメーターで ZooKeeper を開始し、Kerberos ログイン設定を指定します。

```
su - kafka
export EXTRA_ARGS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/zookeeper-server-
start.sh -daemon /opt/kafka/config/zookeeper.properties
```

デフォルトのサービス名(**zookeeper**)を使用していない場合は、**-Dzookeeper.sasl.client.username=NAME** パラメーターを使用して名前を追加します。

注記

/etc/krb5.conf の場所を使用している場合は、ZooKeeper、Kafka、または Kafka プロデューサーおよびコンシューマーの起動時に **-Djava.security.krb5.conf=/etc/krb5.conf** を指定する必要はありません。

Kerberos ログインを使用するように Kafka ブローカーサーバーを設定する

kafka に作成したユーザープリンシパルとキータブを使用して認証に Kerberos Key Distribution Center(KDC)を使用するように Kafka を設定します。

1. 以下の要素で **opt/kafka/config/jaas.conf** ファイルを変更します。

```
KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/kafka-node1.keytab"
    principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    useTicketCache=false
```

```
    keyTab="/opt/kafka/krb5/kafka-node1.keytab"
    principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

2. リスナーが SASL/GSSAPI ログインを使用できるように、**config/server.properties** ファイルのリスナー設定を変更して、Kafka クラスターの各ブローカーを設定します。
SASL プロトコルをリスナーのセキュリティープロトコルのマップに追加し、不要なプロトコルを削除します。

   以下に例を示します。

```
# ...
broker.id=0
# ...
listeners=SECURE://:9092,REPLICATION://:9094 (1)
inter.broker.listener.name=REPLICATION
# ...
listener.security.protocol.map=SECURE:SASL_PLAINTEXT,REPLICATION:SASL_PLAINTEX
T (2)
# ..
sasl.enabled.mechanisms=GSSAPI (3)
sasl.mechanism.inter.broker.protocol=GSSAPI (4)
sasl.kerberos.service.name=kafka (5)
...
```

   (1) 2 つのリスナーが設定されます。汎用通信用のセキュアなリスナーとクライアントとの通信（通信向けの TLS のサポート）、およびブローカー間通信のレプリケーションリスナー。

   (2) TLS 対応のリスナーの場合、プロトコル名は SASL_PLAINTEXT です。TLS 以外のコネクターの場合、プロトコル名は SASL_PLAINTEXT になります。SSL が必要ない場合は、**ssl.*** プロパティーを削除できます。

   (3) Kerberos 認証用の SASL メカニズムは **GSSAPI** です。

   (4) ブローカー間の通信の Kerberos 認証。

   (5) 認証に使用されるサービス名は、同じ Kerberos 設定を使用する可能性がある他のサービスと区別するために指定されます。

3. JVM パラメーターを使用して、Kafka ブローカーを起動し、Kerberos ログイン設定を指定します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/kafka-server-
start.sh -daemon /opt/kafka/config/server.properties
```

   ブローカーおよび ZooKeeper クラスターが設定されており、Kerberos 以外の認証システムを使用している場合、ZooKeeper およびブローカークラスターを起動し、ログで設定エラーを確認できます。

   ブローカーおよび Zookeeper インスタンスを起動すると、クラスターは Kerberos 認証に対して設定されます。

Kerberos 認証を使用するように Kafka プロデューサークライアントおよびコンシューマークライアントを設定します。

producer1 および **consumer1** 用に作成したユーザープリンシパルおよびキータブを使用して認証に Kerberos Key Distribution Center(KDC)を使用するように Kafka プロデューサーおよびコンシューマークライアントを設定します。

1. Kerberos 設定をプロデューサーまたはコンシューマー設定ファイルに追加します。
以下に例を示します。

/opt/kafka/config/producer.properties

```
# ...
sasl.mechanism=GSSAPI (1)
security.protocol=SASL_PLAINTEXT (2)
sasl.kerberos.service.name=kafka (3)
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ (4)
    useKeyTab=true  \
    useTicketCache=false \
    storeKey=true  \
    keyTab="/opt/kafka/krb5/producer1.keytab" \
    principal="producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

(1) Kerberos(GSSAPI)認証の設定。

(2) Kerberos は SASL プレーンテキスト(username/password)セキュリティープロトコルを使用します。

(3) Kerberos KDC で設定された Kafka のサービスプリンシパル（ユーザー）です。

(4) **jaas.conf** で定義された同じプロパティーを使用した JAAS の設定。

/opt/kafka/config/consumer.properties

```
# ...
sasl.mechanism=GSSAPI
security.protocol=SASL_PLAINTEXT
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true  \
    useTicketCache=false \
    storeKey=true  \
    keyTab="/opt/kafka/krb5/consumer1.keytab" \
    principal="consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

2. クライアントを実行して、Kafka ブローカーからメッセージを送受信できることを確認します。
プロデューサークライアント：

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-producer.sh --producer.config
/opt/kafka/config/producer.properties  --topic topic1 --bootstrap-server
```

```
node1.example.redhat.com:9094
```

コンシューマークライアント：

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-consumer.sh --
consumer.config /opt/kafka/config/consumer.properties  --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

関連情報

- Kerberos の man ページ - krb5.conf(5)、kinit(1)、klist(1)、および kdestroy(1)
- RHEL で設定された Kerberos サーバーの例
- Kerberos チケットを使用して Kafka クラスターで認証するクライアントアプリケーションの例

第15章 CRUISE CONTROL によるクラスターのリバランス

重要

Cruise Control によるクラスターのリバランスはテクノロジープレビューの機能です。テクノロジープレビューの機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあります。Red Hat は、本番環境でのテクノロジープレビュー機能の実装は推奨しません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、「テクノロジプレビュー機能のサポート範囲」を参照してください。

Cruise Control を AMQ Streams クラスターにデプロイし、Kafka ブローカー全体で負荷のリバランスに使用できます。

Cruise Control は、クラスターワークロードの監視、事前定義の制約を基にしたクラスターのリバランス、異常の検出および修正などの Kafka の操作を自動化するオープンソースのシステムです。これは、4 つのコンポーネント（Load Monitor、Analyzer、Anomaly Detector、および Executor）と REST API で構成されます。

AMQ Streams および Cruise Control の両方が Red Hat Enterprise Linux にデプロイされる場合、Cruise Control REST API を使用して Cruise Control 機能にアクセスできます。以下の機能がサポートされます。

- **最適化ゴール** および **容量制限**の設定
- **/rebalance** エンドポイントを使用して以下を行うこと。
  - 設定された **最適化ゴールまたはリクエストパラメーターとして提供された ユーザー提供ゴールに基づいて、最適化 プロポーザル** をドライランとして生成します。
  - Kafka クラスターのリバランスを行う最適化プロポーザルの開始
- **/user_tasks** エンドポイントを使用したアクティブなリバランス操作の進捗の確認
- **/stop_proposal_execution** エンドポイントを使用したアクティブなリバランス操作の停止

異常検出、通知、独自ゴールの作成、トピックレプリケーション係数の変更など、その他の Cruise Control 機能はすべて現時点でサポートされていません。Web UI コンポーネント(Cruise Control Frontend)には対応していません。

Cruise Control for AMQ Streams on Red Hat Enterprise Linux は、個別の zip ディストリビューションとして提供されます。詳細は、「Cruise Control アーカイブのダウンロード」 を参照してください。

15.1. CRUISE CONTROL とは

Cruise Control は、ブローカー全体でより均等に分散され、効率的な Kafka クラスターを実行するための時間および労力を削減します。

通常、クラスターの負荷は時間とともに不均等になります。大量のメッセージトラフィックを処理するパーティションは、使用可能なブローカー全体で不均等に分散される可能性があります。クラスターを再分散するには、管理者はブローカーの負荷を監視し、トラフィックの多いパーティションを容量に余裕のあるブローカーに手作業で再割り当てします。

Cruise Control はこのクラスターのリバランスプロセスを自動化します。CPU、ディスク、およびネットワーク負荷に基づいて、リソース使用 の ワークロードモデルを構築します。設定可能な最適化ゴールのセットを使用すると、Cruise Control に対して、パーティションの割り当てをより均等にする最適化プロポーザルを生成するように指示できます。

ドライラン最適化プロポーザルを確認した後、Cruise Control に対してそのプロポーザルを基にしてクラスターリバランスを開始するか、新しいプロポーザルを生成するように指示できます。

クラスターのリバランス操作が完了すると、ブローカーはより効率的に使用され、Kafka クラスターの負荷はより均等に分散されます。

関連情報

- Cruise Control の Wiki
- 「最適化ゴールの概要」
- 「最適化プロポーザルの概要」
- 容量の設定

15.2. CRUISE CONTROL アーカイブのダウンロード

Red Hat カスタマーポータル から、Red Hat Enterprise Linux 上の AMQ Streams 向けの Cruise Control の zip ディストリビューションを利用できます。

手順

1. Red Hat カスタマーポータルから最新バージョンの **Red Hat AMQ Streams Cruise Control** アーカイブをダウンロードします。
2. **/opt/cruise-control** ディレクトリーを作成します。

```
sudo mkdir /opt/cruise-control
```

3. Cruise Control の ZIP ファイルの内容を新しいディレクトリーに展開します。

```
unzip amq-streams-y.y.y-cruise-control-bin.zip -d /opt/cruise-control
```

4. **/opt/cruise-control** ディレクトリーの所有権を **kafka** ユーザーに変更します。

```
sudo chown -R kafka:kafka /opt/cruise-control
```

15.3. CRUISE CONTROL の METRICS REPORTER のデプロイ

Cruise Control を起動する前に、提供された Cruise Control Metrics Reporter を使用するように Kafka ブローカーを設定する必要があります。

ランタイム時に読み込まれると、Metrics Reporter はメトリクスを **__CruiseControlMetrics** トピックに送信します。1 つは 自動作成されたトピック のいずれかになります。Cruise Control はこれらのメトリクスを使用してワークロードモデルを作成および更新し、最適化プロポーザルを算出します。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- Kafka および ZooKeeper が稼働している必要があります。
- 「Cruise Control アーカイブのダウンロード」.

手順

Kafka クラスターの各ブローカーと 1 つずつ、以下の 1 つに対して以下を行います。

1. Kafka ブローカーを停止します。

```
/opt/kafka/bin/kafka-server-stop.sh
```

2. Cruise Control Metrics Reporter **.jar** ファイルを Kafka ライブラリーディレクトリーにコピーします。

```
cp /opt/cruise-control/libs/cruise-control-metrics-reporter-y.y.yyy.redhat-0000x.jar
/opt/kafka/libs
```

3. Kafka 設定ファイル(**/opt/kafka/config/server.properties**)で、Cruise Control Metrics Reporter を設定します。

   a. **CruiseControlMetricsReporter** クラスを **metric.reporters** 設定オプションに追加します。既存の Metrics Reporter を削除しないでください。

```
metric.reporters=com.linkedin.kafka.cruisecontrol.metricsreporter.CruiseControlMetricsReporter
```

   b. 以下の設定オプションと値を Kafka 設定ファイルに追加します。

```
cruise.control.metrics.topic.auto.create=true
cruise.control.metrics.topic.num.partitions=1
cruise.control.metrics.topic.replication.factor=1
```

   これらのオプションにより、Cruise Control Metrics Reporter は、ログクリーンアップポリシー **DELETE** で **__CruiseControlMetrics** トピックを作成できます。詳細は、「自動作成されたトピック、Cruise Control Metrics トピックの Log cleanup policy 」を参照してください。

4. 必要に応じて SSL を設定します。

   a. Kafka 設定ファイル(**/opt/kafka/config/server.properties**)では、関連するクライアント設定プロパティーを設定して、Cruise Control Metrics Reporter と Kafka ブローカー間の SSL を設定します。
   Metrics Reporter は、すべての標準的なプロデューサー固有の設定プロパティーと **cruise.control.metrics.reporter** プレフィックスを受け入れます。例：**cruise.control.metrics.reporter.ssl.truststore.password**

   b. Cruise Control プロパティーファイル(**/opt/cruise-control/config/cruisecontrol.properties**)では、関連するクライアント設定プロパティーを設定して、Kafka ブローカーと Cruise Control サーバー間で SSL を設定します。
   Cruise Control は Kafka から SSL クライアントプロパティーオプションを継承するため、これらのプロパティーをすべての Cruise Control サーバークライアントに使用します。

5. Kafka ブローカーを再起動します。

```
/opt/kafka/bin/kafka-server-start.sh
```

6. 残りのブローカーにステップ 1-5 を繰り返します。

15.4. CRUISE CONTROL の設定および起動

Cruise Control によって使用されるプロパティーを設定し、**cruise-control-start.sh** スクリプトを使用して Cruise Control サーバーを起動します。サーバーは、Kafka クラスター全体に対して 1 台のマシンでホストされます。

3 つのトピックは、Cruise Control の起動時に自動的に作成されます。詳細は、「自動作成されたトピック」を参照してください。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- 「Cruise Control アーカイブのダウンロード」
- 「Cruise Control の Metrics Reporter のデプロイ」

手順

1. Cruise Control プロパティーファイル(**/opt/cruise-control/config/cruisecontrol.properties**)を編集します。
2. 以下の例に示すプロパティーを設定します。

```
# The Kafka cluster to control.
bootstrap.servers=localhost:9092 1

# The replication factor of Kafka metric sample store topic
sample.store.topic.replication.factor=2 2

# The configuration for the BrokerCapacityConfigFileResolver (supports JBOD, non-JBOD,
and heterogeneous CPU core capacities)
#capacity.config.file=config/capacity.json
#capacity.config.file=config/capacityCores.json
capacity.config.file=config/capacityJBOD.json 3

# The list of goals to optimize the Kafka cluster for with pre-computed proposals
default.goals={List of default optimization goals} 4

# The list of supported goals
goals={list of master optimization goals} 5

# The list of supported hard goals
hard.goals={List of hard goals} 6

# How often should the cached proposal be expired and recalculated if necessary
proposal.expiration.ms=60000 7

# The zookeeper connect of the Kafka cluster
zookeeper.connect=localhost:2181 8
```

-

1. Kafka ブローカーのホストとポート番号（常にポート 9092）。
2. Kafka メトリクスサンプルストアトピックのレプリケーション係数。単一ノードの Kafka および ZooKeeper クラスターで Cruise Control を評価する場合は、このプロパティーを 1 に設定します。実稼働環境で使用する場合は、このプロパティーを 2 以上に設定します。
3. ブローカーリソースの最大容量制限を設定する設定ファイル。Kafka デプロイメント設定に適用されるファイルを使用します。詳細は「容量の 設定」を参照し てください。
4. 完全修飾ドメイン名(FQDN)を使用した、デフォルトの最適化ゴールのコンマ区切りリスト。多くのマスター最適化ゴール（5 を参照）は、デフォルトの最適化ゴールとして設定されています。必要な場合はゴールを追加または削除できます。詳細は、「最適化ゴールの概要」 を参照してください。
5. FQDN を使用するマスター最適化ゴールのコンマ区切りリスト。最適化プロポーザルの生成に使用されるゴールを完全に除外するには、それらをリストから削除します。詳細は、「最適化ゴールの概要」 を参照してください。
6. FQDN を使用したハードゴールのコンマ区切りリスト。マスター最適化ゴールの 7 つがハードゴールとして設定されています。必要な場合はゴールを追加または削除できます。詳細は、「最適化ゴールの概要」 を参照してください。
7. デフォルトの最適化ゴールから生成されるキャッシュされた最適化プロポーザルを更新する間隔（ミリ秒単位）。詳細は、「最適化プロポーザルの概要」 を参照してください。
8. ZooKeeper 接続のホストとポート番号（常にポート 2181）

3. Cruise Control サーバーを起動します。サーバーはデフォルトでポート 9092 で起動します。オプションで別のポートを指定します。

```
cd /opt/cruise-control/
./bin/cruise-control-start.sh config/cruisecontrol.properties PORT
```

4. Cruise Control が稼働していることを確認するには、Cruise Control サーバーの **/state** エンドポイントに GET リクエストを送信します。

```
curl 'http://HOST:PORT/kafkacruisecontrol/state'
```

自動作成されたトピック

以下の表は、Cruise Control の起動時に自動作成される 3 つのトピックを表しています。これらのトピックは、Cruise Control が適切に動作するために必要であるため、削除または変更しないでください。

表15.1 自動作成されたトピック

自動作成されたトピック	作成元	機能
__CruiseControlMetrics	Cruise Control の Metrics Reporter	Metrics Reporter からの raw メトリクスを各 Kafka ブローカーに格納します。

自動作成されたトピック	作成元	機能
__KafkaCruiseControlPartitionMetricSamples	Cruise Control	各パーティションの派生されたメトリクスを格納します。これらは Metric Sample Aggregator によって作成されます。
__KafkaCruiseControlModelTrainingSamples	Cruise Control	クラスターワークロードモデルの作成に使用されるメトリクスサンプルを格納します。

自動作成されたトピックでログコンパクションを **無効** 化するには、「Cruise Control の Metrics Reporter のデプロイ」 の説明に従って Cruise Control Metrics Reporter を設定してください。ログコンパクションにより、Cruise Control が必要とするレコードを削除し、適切に機能しないようにすることができます。

関連情報

- Cruise Control Metrics トピックのログクリーンアップポリシー

15.5. 最適化ゴールの概要

Cruise Control は Kafka クラスターをリバランスするために、最適化ゴールを使用して最適 化プロポーザル を生成します。最適化ゴールは、Kafka クラスター全体のワークロード再分散およびリソース使用の制約です。

AMQ Streams on Red Hat Enterprise Linux は、Cruise Control プロジェクトで開発されたすべての最適化ゴールをサポートします。以下に、サポートされるゴールをデフォルトの優先度順に示します。

1. ラックアウェアネス (Rack Awareness)
2. トピックのセットに対するブローカーごとのリーダーレプリカの最小数
3. レプリカの容量
4. 容量: ディスク容量、ネットワークインバウンド容量、ネットワークアウトバウンド容量
5. CPU 容量
6. レプリカの分散
7. 潜在的なネットワーク出力
8. リソース分布: ディスク使用率の分布、ネットワークインバウンド使用率の分布、ネットワークアウトバウンド使用率の分布。
9. リーダーへの単位時間あたりバイト流入量の分布
10. トピックレプリカの分散
11. CPU 使用率の分散

12. リーダーレプリカの分散
13. 優先リーダーの選択
14. Kafka Assigner ディスク使用の分散
15. ブローカー内のディスクの容量
16. ブローカー内のディスク使用量

各最適化ゴールの詳細は、Cruise Control Wiki の「 Goals 」を参照してください。

Cruise Control プロパティーファイルのゴール設定

cruise-control/config/ ディレクトリーの **cruisecontrol.properties** ファイルで最適化ゴールを設定します。満たさなければならない ハード 最適化ゴールの設定と、マスター および デフォルト の最適化ゴールの設定があります。

オプション： ユーザー提供 の最適化ゴールは、**/rebalance** エンドポイントへのリクエストのパラメーターとして、実行時に設定されます。

最適化ゴールは、ブローカーリソースのあらゆる 容量制限 の対象となります。

以下のセクションでは、各ゴール設定の詳細を説明します。

マスター最適化ゴール

マスター最適化ゴールはすべてのユーザーが使用できます。マスター最適化ゴールにリストされていないゴールは、Cruise Control 操作で使用できません。

以下のマスター最適化ゴールは、**goals** プロパティーの優先度で **cruisecontrol.properties** ファイルに事前設定されています。

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal

最適化プロポーザルの生成には、1 つ以上のゴールを完全に除外する必要がある場合を除き、事前設定されたマスター最適化ゴールを変更しないことが推奨されます。必要な場合、マスター最適化ゴールの優先順位はデフォルトの最適化ゴールの設定で変更できます。

事前設定されたマスター最適化ゴールを変更する必要がある場合は、**goals** プロパティーにゴールのリストを優先度が高いものから順に指定します。**cruisecontrol.properties** ファイルに示されるように、完全修飾ドメイン名を使用します。

少なくとも 1 つのマスターゴールを指定する必要があります。指定しないと、Cruise Control がクラッシュします。

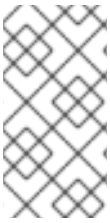
注記

事前設定されたマスター最適化ゴールを変更する場合、設定した **hard.goals** が設定したマスター最適化ゴールのサブセットになるようにする必要があります。そうでないと、最適化プロポーザルの生成時にエラーが発生します。

ハードゴールおよびソフトゴール

ハードゴールは最適化プロポーザルで**必ず**満たさなければならないゴールです。ハードゴールとして設定されていないゴールは**ソフトゴール**と呼ばれます。ソフトゴールはベストエフォートゴールと考えることができ **ます**。最適化プロポーザルで満たす必要はありませんが、最適化の計算に含まれます。

Cruise Control は、すべてのハードゴールを満たし、優先度順にできるだけ多くのソフトゴールを満たす最適化プロポーザルを算出します。すべてのハードゴールを満たさ **ない** 最適化プロポーザルは Analyzer によって拒否され、ユーザーに送信されません。



注記

たとえば、クラスター全体でトピックのレプリカを均等に分散するソフトゴールがあるとします (トピックレプリカ分散のゴール)。このソフトゴールを無視すると、設定されたハードゴールがすべて有効になる場合、Cruise Control はこのソフトゴールを無視します。

以下のマスター最適化ゴールは、**hard.goals** ファイルの **cruisecontrol.properties** ファイルのハードゴールとして事前設定されています。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal
```

ハードゴールを変更するには、**hard.goals** プロパティーを編集し、完全修飾ドメイン名を使用して、希望のゴールを指定します。

ハードゴールの数を増やすと、Cruise Control が有効な最適化プロポーザルを計算して生成する可能性が低くなります。

デフォルトの最適化ゴール
Cruise Control は **デフォルトの最適化ゴール** リストを使用して **キャッシュされた最適化プロポーザル** を生成します。詳細は、「最適化プロポーザルの概要」 を参照してください。

ユーザー提供の最適化ゴールを設定して、実行時にデフォルトの最適化ゴール を上書きできます。

以下のデフォルトの最適化ゴールは、**default.goals** プロパティーの優先度で **cruisecontrol.properties** ファイルに事前設定されています。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal;
ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal;
NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal;
CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal;
LeaderBytesInDistributionGoal
```

少なくとも 1 つのデフォルトのゴールを指定する必要があります。指定しない場合は、Cruise Control がクラッシュします。

デフォルトの最適化ゴールを編集するには、**default.goals** プロパティーでゴールのリストを優先度が高いもので指定します。デフォルトのゴールはマスター最適化ゴールのサブセットである必要があります。完全修飾ドメイン名を使用します。

ユーザー提供の最適化ゴール
ユーザー提供の最適化ゴールは、特定の最適化プロポーザルの設定済みのデフォルトゴールを絞り込みます。必要に応じて、HTTP リクエストのパラメーターとして **/rebalance** エンドポイントへの設定することができます。詳細は、「最適化プロポーザルの生成」 を参照してください。

ユーザー提供の最適化ゴールは、さまざまな状況の最適化プロポーザルを生成できます。たとえば、ディスクの容量やディスクの使用率を考慮せずに、Kafka クラスター全体でリーダーレプリカの分散を最適化したい場合があります。そのため、リーダーレプリカ分散の単一のゴールを含む **/rebalance** エンドポイントにリクエストを送信します。

ユーザー提供の最適化ゴールには以下が必要になります。

- 設定済みのハードゴールがすべて含まれるようにする必要があります。そうでないと、エラーが発生します。
- マスター最適化ゴールのサブセットである必要があります。

最適化プロポーザルの設定済みのハードゴールを無視するには、**skip_hard_goals_check=true** パラメーターをリクエストに追加します。

関連情報

- 「Cruise Control の設定」
- Cruise Control Wiki の「Configurations 」

15.6. 最適化プロポーザルの概要

最適化プロポーザル は提案された変更の概要です。適用されると、パーティションのワークロードをブローカー間でより均等に分散し、よりバランスになった Kafka クラスターを生成します。各最適化プロポーザルは、その プロポーザルの生成に使用された最適化ゴール のセットが基になっており、ブローカーリソースの設定済みの 容量制限 の対象となります。

/rebalance エンドポイントに POST リクエストを送信すると、最適化プロポーザルが応答で返されます。プロポーザルの情報を使用し、プロポーザルを基にしてクラスターのリバランスを開始するかどうかを決定します。または、最適化ゴールを変更し、別のプロポーザルを生成することもできます。

デフォルトでは、最適化プロポーザルは個別に開始する必要 **があるドライラン** として生成されます。生成できる最適化プロポーザルの数に制限はありません。

キャッシュされた最適化プロポーザル

Cruise Control は、設定済みの デフォルト **最適化ゴールを基にしてキャッシュされた最適** 化プロポーザル を維持します。キャッシュされた最適化プロポーザルはワークロードモデルから生成され、Kafka クラスターの現在の状況を反映するために 15 分ごとに更新されます。

以下のゴール設定が使用されると、最新のキャッシュされた最適化プロポーザルが返されます。

- デフォルトの最適化ゴール
- 現在のキャッシュされたプロポーザルによって満たすことができるユーザー提供の最適化ゴール

キャッシュされた最適化プロポーザルの更新間隔を変更するには、**cruisecontrol.properties** ファイルの **proposal.expiration.ms** 設定を編集します。更新間隔を短くすると、Cruise Control サーバーの負荷が増えますが、変更が頻繁に行われるクラスターでは、更新間隔を短くするよう考慮してください。

最適化プロポーザルの内容

以下の表は、最適化プロポーザルに含まれるプロパティーを表しています。

表15.2 最適化プロポーザルに含まれるプロパティー

プロパティー	説明
n inter-broker replica (y MB) moves	**n**: 個別のブローカー間で移動されるパーティションレプリカの数。 **リバランス操作中のパフォーマンスへの影響度**: 比較的高い。 **y MB**: 個別のブローカーに移動される各パーティションレプリカのサイズの合計。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。MB の数が大きくなると、クラスターのリバランスの完了にかかる時間が長くなります。
n intra-broker replica (y MB) moves	**n**: クラスターのブローカーのディスク間で転送されるパーティションレプリカの合計数。 **リバランス操作中のパフォーマンスへの影響** 度： 比較的高いが **inter-broker replica moves** よりも低い。 **y MB**: 同じブローカーのディスク間で移動される各パーティションレプリカのサイズの合計。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。大量のデータを移動する場合、同じブローカーのディスク間で移動する方が個別のブローカー間で移動するよりも影響度が低くなります（**inter-broker replica moves**を参照）。
n excluded topics	最適化プロポーザルでのパーティションレプリカ/リーダーの移動の計算から除外されるトピックの数。 トピックは以下のいずれかの方法で除外できます。 **cruisecontrol.properties** ファイルで、**topics.excluded.from.partition.movement** プロパティーに正規表現を指定します。 **/rebalance** エンドポイントへの POST リクエストで、**excluded_topics** パラメーターに正規表現を指定します。 正規表現に一致するトピックは応答に一覧表示され、クラスターのリバランスから除外されます。
n leadership moves	**n**: リーダーが別のレプリカに切り替えられるパーティション数。ZooKeeper 設定の変更を伴います。 **リバランス操作中のパフォーマンスへの影響度**: 比較的低い。
n recent windows	**n**: 最適化プロポーザルの基になるメトリクスウインドウの数。
n% of the partitions covered	**n%**: 最適化プロポーザルの対象となる Kafka クラスターのパーティションの割合（パーセント）。

プロパティー	説明
On-demand Balancedness Score Before (nn.yyy) After (nn.yyy)	Kafka クラスターの全体的なバランスの測定。 Cruise Control は、複数の要因を基にして **Balancedness Score** を各最適化ゴールに割り当てます。要因には、優先度（**default.goals** またはユーザー提供ゴールのリストのゴールの位置）が含まれます。**On-demand Balancedness Score** は、違反した各ソフトゴールの **Balancedness Score** の合計を 100 から減算して計算されます。 **Before** スコアは、Kafka クラスターの現在の設定を基にします。**After** スコアは、生成された最適化プロポーザルを基にします。

関連情報

- 「最適化ゴールの概要」.
- 「最適化プロポーザルの生成」
- 「クラスターリバランスの開始」

15.7. リバランスパフォーマンスチューニングの概要

クラスターリバランスのパフォーマンスチューニングオプションを調整できます。これらのオプションは、リバランスのパーティションレプリカおよびリーダーシップの移動が実行される方法を制御し、また、リバランス操作に割り当てられた帯域幅も制御します。

パーティション再割り当てコマンド
最適化プロポーザル は、個別のパーティション再割り当てコマンドで構成されています。プロポーザルを開始すると、Cruise Control サーバーはこれらのコマンドを Kafka クラスターに適用します。

パーティション再割り当てコマンドは、以下のいずれかの操作で構成されます。

- **パーティションの移動**: パーティションレプリカとそのデータを新しい場所に転送します。パーティションの移動は、以下の 2 つの形式のいずれかになります。
  - ブローカー間の移動: パーティションレプリカを、別のブローカーのログディレクトリーに移動します。
  - ブローカー内の移動: パーティションレプリカを、同じブローカーの異なるログディレクトリーに移動します。
- **リーダーシップの移動**: パーティションのレプリカのリーダーを切り替えます。

Cruise Control によって、パーティション再割り当てコマンドがバッチで Kafka クラスターに発行されます。リバランス中のクラスターのパフォーマンスは、各バッチに含まれる各タイプの移動数に影響されます。

パーティション再割り当てコマンドを設定するには、「 リバランスチューニングオプション」を参照してください。

レプリカの移動ストラテジー

クラスターリバランスのパフォーマンスは、パーティション再割り当てコマンドのバッチに適用される**レプリカ移動ストラテジー** の影響も受けます。デフォルトでは、Cruise Control は **BaseReplicaMovementStrategy** を使用します。これは、生成された順序でコマンドを適用します。ただし、プロポーザルの初期に非常に大きなパーティションの再割り当てがある場合、このストラテジーによって他の再割り当ての適用が遅くなる可能性があります。

Cruise Control は、最適化プロポーザルに適用できる代替のレプリカ移動ストラテジーを 3 つ提供します。

- **PrioritizeSmallReplicaMovementStrategy**: サイズの昇順で再割り当てを並べ替えます。
- **PrioritizeLargeReplicaMovementStrategy**: サイズの降順で再割り当てを並べ替えます。
- **PostponeUrpReplicaMovementStrategy**: 非同期のレプリカがないパーティションのレプリカの再割り当てを優先します。

これらのストラテジーをシーケンスとして設定できます。最初のストラテジーは、内部ロジックを使用して 2 つのパーティション再割り当ての比較を試みます。再割り当てが同等である場合は、順番を決定するために再割り当てをシーケンスの次のストラテジーに渡します。

レプリカの移動ストラテジーを設定するには、「 リバランスチューニングオプション」を参照し てください。

リバランスチューニングオプション

Cruise Control には、リバランスパラメーターを調整する設定オプションが複数あります。これらのオプションは以下の方法で設定されます。

- プロパティーとして、**cruisecontrol.properties** ファイルのデフォルトの Cruise Control 設定のプロパティー。
- **/rebalance** エンドポイントへの POST リクエストのパラメーター

両方の方法に関連する設定は、以下の表で説明されています。

表15.3 リバランスパフォーマンスチューニングの設定

プロパティーおよびリクエストパラメーターの設定	説明	デフォルト値
num.concurrent.partition.movements.per.broker **concurrent_partition_movements_per_broker**	各パーティション再割り当てバッチでのブローカー間パーティション移動の最大数。	5
num.concurrent.intra.broker.partition.movements **concurrent_intra_broker_partition_movements**	各パーティション再割り当てバッチでのブローカー内パーティション移動の最大数。	2
num.concurrent.leader.movements	各パーティション再割り当てバッチにおけるパーティションリーダー変更の最大数。	1000

プロパティーおよびリクエストパラメーターの設定	説明	デフォルト値
concurrent_leader_movements		
default.replication.throttle **replication_throttle**	パーティション再割り当てに割り当てる帯域幅（バイト/秒単位）	null（制限なし）
default.replica.movement.strategies **replica_movement_strategies**	パーティション再割り当てコマンドが、生成されたプロポーザルに対して実行される順番を決定するために使用されるストラテジー (優先順位順) の一覧。**PrioritizeSmallReplicaMovementStrategy**、**PrioritizeLargeReplicaMovementStrategy**、および **PostponeUrpReplicaMovementStrategy** の 3 つのストラテジーがあります。 このプロパティーでは、ストラテジークラスの完全修飾名のコンマ区切りリストを使用します（各クラス名の先頭に **com.linkedin.kafka.cruisecontrol.executor.strategy.** を追加します）。 パラメーターには、レプリカの移動ストラテジーのクラス名のコンマ区切りリストを使用します。	**BaseReplicaMovementStrategy**

デフォルト設定を変更すると、リバランスの完了までにかかる時間と、リバランス中の Kafka クラスターの負荷に影響します。値を小さくすると負荷は減りますが、かかる時間は長くなり、その逆も同様です。

関連情報

- Cruise Control Wiki の「Configurations」
- Cruise Control Wiki のREST API。

15.8. CRUISE CONTROL の設定

config/cruisecontrol.properties ファイルには、Cruise Control の設定が含まれます。ファイルは、以下のいずれかのタイプのプロパティーで構成されます。

- 文字列

- 数値
- ブール値

Cruise Control Wiki の Configurations セクションに記載されているすべてのプロパティーを指定および設定できます。

容量の設定

Cruise Control は **容量制限** を使用して、特定のリソースベースの最適化ゴールが破損しているか判断します。これらのリソースベースのゴールがハードゴールとして設定され、破損すると、最適化に失敗します。これにより、最適化プロポーザルの生成に最適化が使用されないようにします。

Kafka ブローカーリソースの容量制限は、**cruise-control/config** の以下の 3 つの **.json** ファイルのいずれかで指定します。

- **capacityJBOD.json**: JBOD Kafka デプロイメントで使用します（デフォルトのファイル）。
- **capacity.json**: 各ブローカーが同じ CPU コアの数を持つ JBOD Kafka デプロイメントで使用します。
- **capacityCores.json**: 各ブローカーが CPU コアの数によって異なります。JBOD 以外の Kafka デプロイメントで使用します。

cruisecontrol.properties の **capacity.config.file** プロパティーにファイルを設定します。選択したファイルは、ブローカーの容量解決に使用されます。以下に例を示します。

```
capacity.config.file=config/capacityJBOD.json
```

容量制限は、記述された単位で以下のブローカーリソースに設定できます。

- **DISK**: ディスクストレージ（MB 単位）
- **CPU**: パーセント(0-100)または多数のコアとする CPU 使用率
- **NW_IN**: 1秒あたり KB でのインバウンドネットワークスループット
- **NW_OUT**: 1秒あたり KB のアウトバウンドネットワークスループット

Cruise Control によって監視されるすべてのブローカーに同じ容量制限を適用するには、ブローカー ID **-1** の容量制限を設定します。個別のブローカーに異なる容量制限を設定するには、各ブローカー ID とその容量設定を指定します。

容量制限の設定例

```
{
  "brokerCapacities":[
    {
      "brokerId": "-1",
      "capacity": {
        "DISK": "100000",
        "CPU": "100",
        "NW_IN": "10000",
        "NW_OUT": "10000"
      },
      "doc": "This is the default capacity. Capacity unit used for disk is in MB, cpu is in percentage,
network throughput is in KB."
```

```
    },
    {
      "brokerId": "0",
      "capacity": {
        "DISK": "500000",
        "CPU": "100",
        "NW_IN": "50000",
        "NW_OUT": "50000"
      },
      "doc": "This overrides the capacity for broker 0."
    }
  ]
}
```

詳細は、Cruise Control Wiki の「 Populating the Capacity Configuration File 」を参照してください。

Cruise Control Metrics トピックのログクリーンアップポリシー

自動作成された **__CruiseControlMetrics** トピック（自動作成されたトピックを参照 ）には、**COMPACT** ではなく **DELETE** のログクリーンアップポリシーがあることが重要です。そうしないと、Cruise Control が必要とするレコードが削除される可能性があります。

「Cruise Control の Metrics Reporter のデプロイ」 で説明されているように、Kafka 設定ファイルで以下のオプションを設定すると、**COMPACT** ログクリーンアップポリシーが正しく設定されていることを確認します。

- **cruise.control.metrics.topic.auto.create=true**
- **cruise.control.metrics.topic.num.partitions=1**
- **cruise.control.metrics.topic.replication.factor=1**

トピックの自動作成が Cruise Control Metrics Reporter(**cruise.control.metrics.topic.auto.create=false**)で **無効** になっており、Kafka クラスターで **有効** になっている場合、**__CruiseControlMetrics** トピックは引き続きブローカーによって自動的に作成されます。この場合、**kafka-configs.sh** ツールを使用して、**__CruiseControlMetrics** トピックのログクリーンアップポリシーを **DELETE** に変更する必要があります。

1. **__CruiseControlMetrics** トピックの現在の設定を取得します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name __CruiseControlMetrics --describe
```

2. トピック設定でログクリーンアップポリシーを変更します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name __CruiseControlMetrics --alter --add-config cleanup.policy=delete
```

Cruise Control Metrics Reporter **および** Kafka クラスターの両方でトピックの自動作成が **無効** になっている場合は、**__CruiseControlMetrics** トピックを手動で作成してから、**kafka-configs.sh** ツールを使用して **DELETE** ログクリーンアップポリシーを使用するように設定する必要があります。

詳細は、「トピック設定の変更」 を参照してください。

ロギングの設定

Cruise Control は、すべてのサーバーロギングに **log4j1** を使用します。デフォルト設定を変更するには、**/opt/cruise-control/config/log4j.properties** の **log4j.properties** ファイルを編集します。

変更を反映する前に、Cruise Control サーバーを再起動する必要があります。

15.9. 最適化プロポーザルの生成

/rebalance エンドポイントに POST リクエストを送信すると、Cruise Control は提供された最適化ゴールを基にして、Kafka クラスターをリバランスするために最適化プロポーザルを生成します。

dryrun パラメーターが指定され、**false** に設定されない限り、最適化プロポーザルは **ドライラン** として生成されます。

その後、ドライラン最適化プロポーザルの情報を分析し、開始するかどうかを決定できます。

以下は、**/rebalance** エンドポイントへのリクエストに追加できるキーパラメーターです。使用できるすべてのパラメーターに関する詳細は、Cruise Control Wiki の「 REST APIs 」を参照してください。

dryrun

type: boolean、default: true

最適化プロポーザルのみを生成するか(**true**)、最適化プロポーザルの生成とクラスターリバランス(**false**)が実行されるか（）、Cruise Control に通知します。

excluded_topics

タイプ： regex

最適化プロポーザルの計算から除外するトピックと一致する正規表現。

goals

型： 文字列の一覧、デフォルト： 設定済みの **default.goals** リスト

最適化プロポーザルの準備に使用するユーザー提供の最適化ゴールのリスト。ゴールが指定されていない場合は、**cruisecontrol.properties** ファイルに設定済みの **default.goals** リストが使用されます。

skip_hard_goals_check

型： boolean、default: **false**

デフォルトでは、Cruise Control はユーザー提供の最適化ゴール（ **goals** パラメーター）に設定済みのハードゴール（ **hard.goals**内）がすべて含まれていることを確認します。設定された **hard.goals** のサブセットではないゴールを指定すると、リクエストに失敗します。

設定済みのすべての **hard.goals** が含まれていないユーザー提供の最適化ゴールで最適化プロポーザルを生成する場合は、**skip_hard_goals_check** を **true** に設定します。

json

型： boolean、default: **false**

Cruise Control サーバーによって返される応答のタイプを制御します。指定されていないか、または **false** に設定された場合、Cruise Control はコマンドラインに表示されるようにフォーマットされたテキストを返します。返された情報の要素を抽出する場合は、**json=true** を設定します。これにより、**jq** などのツールやスクリプトやプログラムで解析できる JSON 形式のテキストが返されます。

verbose

型： boolean、default: **false**

Cruise Control サーバーによって返される応答の詳細のレベルを制御します。

前提条件

- Kafka および ZooKeeper が稼働している必要があります。
- Cruise Control が稼働している必要があります。

手順

1. コンソールに対してフォーマットされた最適化プロポーザルを生成するには、POST リクエストを **/rebalance** エンドポイントに送信します。
   - 設定した **default.goals** を使用するには、以下を実行します。

     ```
     curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
     ```

     キャッシュされた最適化プロポーザルがすぐに返されます。

     注記

     NotEnoughValidWindows が返されると、Cruise Control は最適化プロポーザルを生成するために十分なメトリクスデータを記録していません。数分待機した後に要求を再送信します。

   - 設定された **default.goals** の代わりにユーザー定義の最適化ゴールを指定するには、**goals** パラメーターにゴールを 1 つ以上指定します。

     ```
     curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
     goals=RackAwareGoal,ReplicaCapacityGoal'
     ```

     提供されたゴールを満たす場合、キャッシュされた最適化プロポーザルがすぐに返されます。そうでない場合は、提供されたゴールを使用して新しい最適化プロポーザルが生成されます。計算にかかる時間が長くなります。**ignore_proposal_cache=true** パラメーターをリクエストに追加すると、この動作を強制することができます。

   - 設定済みのハードゴールがすべて含まれていないユーザー提供の最適化ゴールを指定するには、**skip_hard_goal_check=true** パラメーターをリクエストに追加します。

     ```
     curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
     goals=RackAwareGoal,ReplicaCapacityGoal,ReplicaDistributionGoal&skip_hard_goal_chec
     k=true'
     ```

2. 応答に含まれる最適化プロポーザルを確認します。プロパティーは、保留中のクラスターリバランス操作を記述します。
   プロポーザルには、提案された最適化の概要が含まれ、その後に各デフォルトの最適化ゴールの概要と、プロポーザルの実行後に想定されるクラスター状態が含まれます。

   以下の情報に特に注意してください。

   - **Cluster load after rebalance** の概要要件を満たす場合、ハイレベルの概要を使用して提案される変更の影響を評価する必要があります。

- **n inter-broker replica (y MB) moves** ブローカー間のネットワーク間で移動するデータの量を示します。値が大きいほど、リバランス中に Kafka クラスターでパフォーマンスが低下する可能性があります。
- **n intra-broker replica (y MB) moves** ブローカー内に移動されるデータ量を指定します（ディスク間の）。値が大きいほど、個別のブローカー（ただし **n inter-broker replica (y MB) moves**未満）でパフォーマンスが低下する可能性があります。
- リーダーシップの移動数。これは、リバランス中のクラスターのパフォーマンスに悪影響を与える可能性があります。

非同期応答
デフォルトでは 10 秒後に Cruise Control REST API エンドポイントがタイムアウトしますが、プロポーザルの生成はサーバー上で続行されます。キャッシュされた最適化プロポーザルが最新の最適化プロポーザルがない場合や、ユーザー提供の最適化ゴールが **ignore_proposal_cache=true** で指定された場合、タイムアウトが発生する可能性があります。

後で最適化プロポーザルを取得できるようにするには、**/rebalance** エンドポイントからの応答のヘッダーに指定されるリクエスト固有の識別子を書き留めておきます。

curl を使用して応答を取得するには、詳細(**-v**)オプションを指定します。

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

ヘッダーの例を以下に示します。

```
* Connected to cruise-control-server (::1) port 9090 (#0)
> POST /kafkacruisecontrol/rebalance HTTP/1.1
> Host: cc-host:9090
> User-Agent: curl/7.70.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Mon, 01 Jun 2020 15:19:26 GMT
< Set-Cookie: JSESSIONID=node01wk6vjzjj12go13m81o7no5p7h9.node0; Path=/
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201
< Content-Type: text/plain;charset=utf-8
< Cruise-Control-Version: 2.0.103.redhat-00002
< Cruise-Control-Commit_Id: 58975c9d5d0a78dd33cd67d4bcb497c9fd42ae7c
< Content-Length: 12368
< Server: Jetty(9.4.26.v20200117-redhat-00001)
```

最適化プロポーザルがタイムアウト内に準備状態にない場合、POST リクエストを再送できます。これには、ヘッダーの元のリクエストの **User-Task-ID** が含まれます。

```
curl -v -X POST -H 'User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201' 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

次のステップ

「クラスターリバランスの開始」

15.10. クラスターリバランスの開始

最適化プロポーザルが適切であれば、Cruise Control に対して、クラスターのリバランスを開始し、プロポーザルにまとめられたようにパーティションの再割り当てを開始するように指示できます。

最適化プロポーザルの生成とクラスターのリバランスの開始間隔は、できるだけ少ない時間のままにします。元の最適化プロポーザルを生成してからしばらくが経過した場合、クラスターの状態が変更する可能性があります。そのため、開始されるクラスターリバランスは確認したものとは異なる可能性があります。不明な場合は、最初に新しい最適化プロポーザルを生成します。

ステータスが「Active」であるクラスターのリバランスは 1 つだけです。

前提条件

- Cruise Control から 最適化プロポーザルを生成済み である必要があります。

手順

1. 最近生成された最適化プロポーザルを実行するには、**dryrun=false** パラメーターを使用して POST リクエストを **/rebalance** エンドポイントに送信します。

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?dryrun=false'
```

   Cruise Control はクラスターのリバランスを開始し、最適化プロポーザルを返します。

2. 最適化プロポーザルで要約された変更を確認します。変更が機能しない場合は、リバランスを停止 できます。

3. **/user_tasks** エンドポイントを使用して、クラスターリバランスの進捗を確認します。クラスターリバランスの進捗には、「Active」のステータスがあります。
   Cruise Control サーバーで実行されたすべてのクラスターリバランスタスクを表示するには、以下を実行します。

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks'

USER TASK ID      CLIENT ADDRESS  START TIME     STATUS  REQUEST URL
c459316f-9eb5-482f-9d2d-97b5a4cd294d  0:0:0:0:0:0:0:1       2020-06-01_16:10:29 UTC
Active      POST /kafkacruisecontrol/rebalance?dryrun=false
445e2fc3-6531-4243-b0a6-36ef7c5059b4  0:0:0:0:0:0:0:1       2020-06-01_14:21:26 UTC
Completed   GET /kafkacruisecontrol/state?json=true
05c37737-16d1-4e33-8e2b-800dee9f1b01  0:0:0:0:0:0:0:1       2020-06-01_14:36:11 UTC
Completed   GET /kafkacruisecontrol/state?json=true
aebae987-985d-4871-8cfb-6134ecd504ab  0:0:0:0:0:0:0:1       2020-06-01_16:10:04 UTC
```

4. 特定のクラスターリバランスタスクのステータスを表示するには、**user-task-ids** パラメーターとタスク ID を指定します。

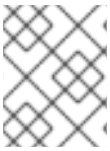
```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks?user_task_ids=c459316f-9eb5-482f-9d2d-97b5a4cd294d'
```

15.11. アクティブなクラスターリバランスの停止

現在進行中であるクラスターリバランスを停止できます。

これにより、現在のパーティション再割り当てのバッチ処理を完了し、リバランスを停止するよう Cruise Control が指示されます。リバランスの停止時、完了したパーティションの再割り当ては既に適

用されています。そのため、Kafka クラスターの状態は、リバランス操作の開始前とは異なります。さらなるリバランスが必要な場合は、新しい最適化プロポーザルを生成してください。



注記

中間 (停止) 状態の Kafka クラスターのパフォーマンスは、初期状態の場合よりも悪くなる可能性があります。

前提条件

- クラスターリバランスは進行中です（「 Active」のステータスで表される）。

手順

- POST リクエストを **/stop_proposal_execution** エンドポイントに送信します。

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/stop_proposal_execution'
```

その他のリソース

- 「最適化プロポーザルの生成」

第16章 分散トレーシング

分散トレーシングを使用すると、分散システムのアプリケーション間で実行されるトランザクションの進捗を追跡できます。マイクロサービスのアーキテクチャーでは、トレーシングはサービス間のトランザクションの進捗を追跡します。トレースデータは、アプリケーションのパフォーマンスを監視し、ターゲットシステムおよびエンドユーザーアプリケーションの問題を調べるのに役立ちます。

AMQ Streams on Red Hat Enterprise Linux では、トレーシングによってメッセージのエンドツーエンドの追跡が容易になります。これは、ソースシステムから Kafka、さらに Kafka からターゲットシステムおよびアプリケーションへの最終的な追跡が容易になります。トレースにより、利用可能な JMX メトリクス が補完されます。

AMQ Streams によるトレーシングのサポート方法

以下のクライアントおよびコンポーネントに対してトレーシングのサポートは提供されます。

Kafka クライアント：

- Kafka プロデューサーおよびコンシューマー
- Kafka Streams API アプリケーション

Kafka コンポーネント：

- Kafka Connect
- Kafka Bridge
- MirrorMaker
- MirrorMaker 2.0

トレーシングを有効にするには、高レベルのタスクを実行します。

1. Jaeger トレーサーを有効にします。
2. インターセプターを有効にします。
   - Kafka クライアントでは、OpenTracing Apache Kafka Client Instrumentation ライブラリー（AMQ Streams に含まれる）を使用してアプリケーションコードを **インストルメント化** します。
   - Kafka コンポーネントでは、各コンポーネントの設定プロパティーを設定します。
3. トレーシング環境変数 を設定します。
4. クライアントまたはコンポーネントをデプロイします。

インストルメント化されると、クライアントはトレースデータを生成します。たとえば、メッセージを作成したり、ログへのオフセットの書き込み時などです。

トレースは、サンプリングストラテジーに従いサンプル化され、Jaeger ユーザーインターフェースで可視化されます。

注記

トレーシングは Kafka ブローカーではサポートされません。

AMQ Streams 以外のアプリケーションおよびシステムにトレーシングを設定する方法については、本章の対象外となります。この件についての詳細は、OpenTracing ドキュメントを参照し、「inject and extrac」を検索してください。

手順の概要

AMQ Streams のトレーシングを設定するには、以下の手順を順番に行います。

1. クライアントのトレーシングを設定します。
   a. Kafka クライアントの Jaeger トレーサーを初期化します。
   b. プロデューサーおよびコンシューマーをトレーシング用にインストルメント化します。
   c. Kafka Streams アプリケーションをトレーシング用にインストルメント化します。
2. MirrorMaker、MirrorMaker 2.0、および Kafka Connect のトレースを設定します。
   a. MirrorMaker のトレースを有効にします。
   b. MirrorMaker 2.0 のトレースを有効にします。
   c. Kafka Connect のトレースを有効にします。
3. Kafka Bridge のトレースを有効にします。

前提条件

- Jaeger バックエンドコンポーネントはホストオペレーティングシステムにデプロイされます。デプロイメント手順の詳細は、Jaeger デプロイメントのドキュメント を参照してください。

16.1. OPENTRACING および JAEGER の概要

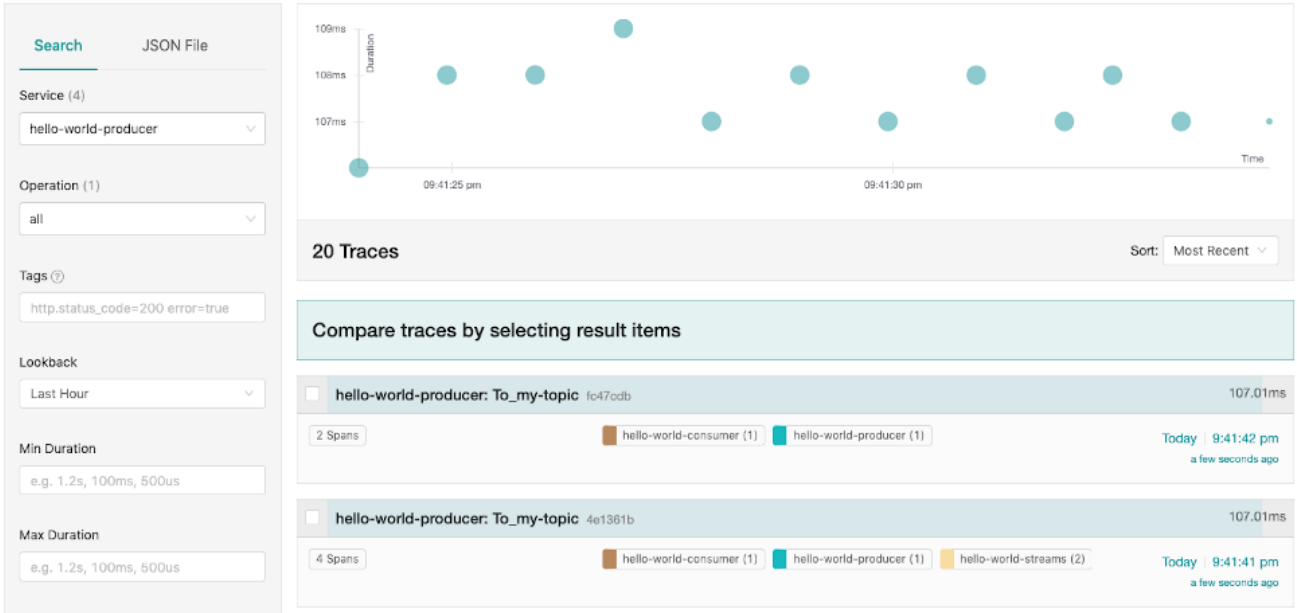
AMQ Streams では OpenTracing および Jaeger プロジェクトが使用されます。

OpenTracing は、トレーシングまたは監視システムに依存しない API 仕様です。

- OpenTracing API は、アプリケーションコードを **インストルメント化** するために使用されます。
- インストルメント化されたアプリケーションは、分散システム全体で個別のトランザクションの **トレース** を生成します。
- トレースは、特定の作業単位を定義する **スパン** で構成されます。

Jaeger はマイクロサービスベースの分散システムのトレーシングシステムです。

- Jaeger は OpenTracing API を実装し、インストルメント化のクライアントライブラリーを提供します。
- Jaeger ユーザーインターフェースを使用すると、トレースデータをクエリー、フィルター、および分析できます。



その他のリソース

- OpenTracing
- Jaeger

16.2. KAFKA クライアントのトレーシング設定

Jaeger トレーサーを初期化し、分散トレーシング用にクライアントアプリケーションをインストルメント化します。

16.2.1. Kafka クライアント用の Jaeger トレーサーの初期化

一連のトレーシング環境変数を使用して、Jaeger トレーサーを設定および初期化します。

手順

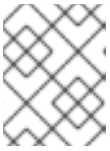
各クライアントアプリケーションで以下を行います。

1. Jaeger の Maven 依存関係をクライアントアプリケーションの **pom.xml** ファイルに追加します。

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-client</artifactId>
    <version>1.1.0.redhat-00002</version>
</dependency>
```

2. トレーシング環境変数を使用して Jaeger トレーサーの設定を定義します。
3. 2. で定義した環境変数から、Jaeger トレーサーを作成します。

```
Tracer tracer = Configuration.fromEnv().getTracer();
```



注記

別の Jaeger トレーサーの初期化方法については、Java OpenTracing ライブラリー のドキュメントを参照してください。

4. Jaeger トレーサーをグローバルトレーサーとして登録します。

```
GlobalTracer.register(tracer);
```

これで、Jaeger トレーサーはクライアントアプリケーションが使用できるように初期化されました。

16.2.2. トレーシングのための Kafka プロデューサーおよびコンシューマーのインストルメント化

Decorator パターンまたは Interceptor を使用して、Java プロデューサーおよびコンシューマーアプリケーションコードをトレーシング用にインストルメント化します。

手順

各プロデューサーおよびコンシューマーアプリケーションのアプリケーションコードで以下を行います。

1. OpenTracing の Maven 依存関係を、プロデューサーまたはコンシューマーの **pom.xml** ファイルに追加します。

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-kafka-client</artifactId>
    <version>0.1.15.redhat-00001</version>
</dependency>
```

2. Decorator パターンまたは Interceptor のいずれかを使用して、クライアントアプリケーションコードをインストルメント化します。

- Decorator パターンを使用する場合は以下を行います。

```
// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer:
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer,
        tracer);

// Send:
tracingProducer.send(...);

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer:
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer,
        tracer);
```

```
// Subscribe:
tracingConsumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);

// Retrieve SpanContext from polled record (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

- インターセプターを使用する場合は以下を使用します。

```
// Register the tracer with GlobalTracer:
GlobalTracer.register(tracer);

// Add the TracingProducerInterceptor to the sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
        TracingProducerInterceptor.class.getName());

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send:
producer.send(...);

// Add the TracingConsumerInterceptor to the consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
        TracingConsumerInterceptor.class.getName());

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Subscribe:
consumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// Retrieve the SpanContext from a polled message (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

Decorator パターンのカスタムスパン名
スパン は Jaeger の論理作業単位で、操作名、開始時間、および期間が含まれます。

Decorator パターンを使用してプロデューサーおよびコンシューマーの各アプリケーションをインストルメント化する場合、**TracingKafkaProducer** および **TracingKafkaConsumer** オブジェクトの作成時に **BiFunction** オブジェクトを追加の引数として渡すと、カスタムスパン名を定義できます。
OpenTracing の Apache Kafka Client Instrumentation ライブラリーには、複数の組み込みスパン名が含まれています。

例: カスタムスパン名を使用した Decorator パターンでのクライアントアプリケーションコードのインストルメント化

```
// Create a BiFunction for the KafkaProducer that operates on (String operationName,
ProducerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ProducerRecord, String> producerSpanNameProvider =
    (operationName, producerRecord) -> "CUSTOM_PRODUCER_NAME";

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>(producer,
        tracer,
        producerSpanNameProvider);

// Spans created by the tracingProducer will now have "CUSTOM_PRODUCER_NAME" as the span
name.

// Create a BiFunction for the KafkaConsumer that operates on (String operationName,
ConsumerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ConsumerRecord, String> consumerSpanNameProvider =
    (operationName, consumerRecord) -> operationName.toUpperCase();

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer, passing in the consumerSpanNameProvider
BiFunction:

TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
(consumer,
        tracer,
        consumerSpanNameProvider);

// Spans created by the tracingConsumer will have the operation name as the span name, in upper-
case.
// "receive" -> "RECEIVE"
```

ビルトインスパン名
カスタムスパン名を定義するとき、**ClientSpanNameProvider** クラスで以下の **BiFunctions** を使用できます。**spanNameProvider** の指定がない場合は、**CONSUMER_OPERATION_NAME** および **PRODUCER_OPERATION_NAME** が使用されます。

表16.1 カスタムスパン名を定義する BiFunctions

BiFunction	説明
CONSUMER_OPERATION_NAME, PRODUCER_OPERATION_NAME	**operationName** をスパン名として返します。コンシューマーには「receive」、プロデューサーには「send」を返します。

BiFunction	説明
CONSUMER_PREFIXED_OPERATION_NAME (String prefix), PRODUCER_PREFIXED_OPERATION_NAME(String prefix)	**prefix** および **operationName** の文字列連結を返します。
CONSUMER_TOPIC, PRODUCER_TOPIC	メッセージの送信先または送信元となったトピックの名前を**(record.topic())** 形式で返します。
PREFIXED_CONSUMER_TOPIC(String prefix), PREFIXED_PRODUCER_TOPIC(String prefix)	**prefix** およびトピック名の文字列連結を **(record.topic())** 形式で返します。
CONSUMER_OPERATION_NAME_TOPIC, PRODUCER_OPERATION_NAME_TOPIC	操作名およびトピック名を **"operationName - record.topic()"** 形式で返します。
CONSUMER_PREFIXED_OPERATION_NAME _TOPIC(String prefix), PRODUCER_PREFIXED_OPERATION_NAME _TOPIC(String prefix)	**prefix** および **"operationName - record.topic()"** の文字列連結を返します。

16.2.3. Kafka Streams アプリケーションのトレーシングのインストルメント化

サプライヤーインターフェースを使用して、分散トレーシング用に Kafka Streams アプリケーションをインストルメント化します。これにより、アプリケーションのインターセプターが有効になります。

手順

各 Kafka Streams アプリケーションで以下を行います。

1. **opentracing-kafka-streams** 依存関係を Kafka Streams アプリケーションの **pom.xml** ファイルに追加します。

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-kafka-streams</artifactId>
    <version>0.1.15.redhat-00001</version>
</dependency>
```

2. **TracingKafkaClientSupplier** サプライヤーインターフェースのインスタンスを作成します。

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

3. サプライヤーインターフェースを **KafkaStreams** に提供します。

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

16.3. MIRRORMAKER および KAFKA CONNECT のトレース設定

本セクションでは、分散トレーシング向けに MirrorMaker、MirrorMaker 2.0、および Kafka Connect を設定する方法を説明します。

コンポーネントごとに Jaeger トレーサーを有効にする必要があります。

16.3.1. MirrorMaker のトレースの有効化

Interceptor プロパティーをコンシューマーおよびプロデューサーの設定パラメーターとして渡すと、MirrorMaker の分散トレースを有効にします。

メッセージはソースクラスターからターゲットクラスターにトレースされます。トレースデータは、MirrorMaker コンポーネントを出入りするメッセージを記録します。

手順

1. Jaeger トレーサーを設定および有効にします。
2. **/opt/kafka/config/consumer.properties** ファイルを編集します。
   以下のインターセプタープロパティーを追加します。

```
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor
```

3. **/opt/kafka/config/producer.properties** ファイルを編集します。
   以下のインターセプタープロパティーを追加します。

```
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
```

4. コンシューマーおよびプロデューサー設定ファイルで、MirrorMaker を起動します。

```
su - kafka
/opt/kafka/bin/kafka-mirror-maker.sh --consumer.config /opt/kafka/config/consumer.properties
--producer.config /opt/kafka/config/producer.properties --num.streams=2
```

16.3.2. MirrorMaker 2.0 のトレースの有効化

MirrorMaker 2.0 プロパティーファイルでインターセプタープロパティーを定義して、MirrorMaker 2.0 の分散トレーシングを有効にします。

メッセージは Kafka クラスター間でトレーシングされます。トレースデータは、MirrorMaker 2.0 コンポーネントを出入りするメッセージを記録します。

手順

1. Jaeger トレーサーを設定および有効にします。
2. MirrorMaker 2.0 設定プロパティーファイル **./config/connect-mirror-maker.properties** を編集し、以下のプロパティーを追加します。

```
header.converter=org.apache.kafka.connect.converters.ByteArrayConverter (1)
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor (2)
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
```

1. Kafka Connect がメッセージヘッダー（トレース ID を含む）を base64 エンコーディングに変換しないようにします。これにより、メッセージがソースクラスターとターゲットクラスターの両方で同じようになります。
2. MirrorMaker 2.0 のインターセプターを有効にします。

3. 「MirrorMaker 2.0 を使用した Kafka クラスター間でのデータの同期」 の手順を使用して、MirrorMaker 2.0 を起動します。

関連情報

- *10章AMQ Streams の MirrorMaker 2.0 との使用*

16.3.3. Kafka Connect のトレースの有効化

設定プロパティーを使用して Kafka Connect の分散トレーシングを有効にします。

Kafka Connect により生成および消費されるメッセージのみがトレーシングされます。Kafka Connect と外部システム間で送信されるメッセージをトレーシングするには、これらのシステムのコネクターでトレーシングを設定する必要があります。

手順

1. Jaeger トレーサーを設定および有効にします。
2. 関連する Kafka Connect 設定ファイルを編集します。
   - Kafka Connect をスタンドアロンモードで実行している場合は、**/opt/kafka/config/connect-standalone.properties** ファイルを編集します。
   - 分散モードで Kafka Connect を実行している場合は、**/opt/kafka/config/connect-distributed.properties** ファイルを編集します。
3. 以下のプロパティーを設定ファイルに追加します。

```
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor
```

4. 設定ファイルを作成します。
5. トレーシング環境変数を設定してから、スタンドアロンまたは分散モードで Kafka Connect を実行します。

Kafka Connect の内部コンシューマーおよびプロデューサーのインターセプターが有効になりました。

関連情報

- 「トレーシングの環境変数」
- 「スタンドアロンモードでの Kafka Connect の実行」
- 「分散 Kafka Connect の実行」

16.4. KAFKA BRIDGE のトレースの有効化

Kafka Bridge 設定ファイルを編集して、Kafka Bridge の分散トレーシングを有効にします。その後、ホストオペレーティングシステムへの分散トレーシング向けに設定された Kafka Bridge インスタンスをデプロイすることができます。

以下の場合にトレースが生成されます。

- Kafka Bridge はメッセージを HTTP クライアントに送信し、HTTP クライアントからメッセージを消費します。
- HTTP クライアントは HTTP リクエストを送信して、Kafka Bridge 経由でメッセージを送受信します。

エンドツーエンドのトレーシングを設定するために、HTTP クライアントでトレーシングを設定する必要があります。

手順

1. Kafka Bridge インストールディレクトリーの **config/application.properties** ファイルを編集します。
   以下の行からコードコメントを削除します。

   ```
   bridge.tracing=jaeger
   ```

2. 設定ファイルを作成します。
3. 設定プロパティーをパラメーターとして使用して、**bin/kafka_bridge_run.sh** スクリプトを実行します。

   ```
   cd kafka-bridge-0.xy.x.redhat-0000x
   ./bin/kafka_bridge_run.sh --config-file=config/application.properties
   ```

   Kafka Bridge の内部コンシューマーおよびプロデューサーのインターセプターが有効になりました。

関連情報

- 「Kafka Bridge プロパティーの設定」

16.5. トレーシングの環境変数

これらの環境変数を使用して、Kafka クライアントおよびコンポーネントに Jaeger トレーサーを設定するときに使用します。

注記

トレーシング環境変数は Jaeger プロジェクトの一部で、変更される場合があります。最新の環境変数については、Jaeger ドキュメント を参照してください。

表16.2 Jaeger トレーサー環境変数

プロパティー	必要性	説明

プロパティー	必要性	説明
JAEGER_SERVICE_NAME	必要	Jaeger トレーサーサービスの名前。
JAEGER_AGENT_HOST	不要	UDP (User Datagram Protocol) を介した **jaeger-agent** との通信のためのホスト名。
JAEGER_AGENT_PORT	不要	UDP を介した **jaeger-agent** との通信に使用されるポート。
JAEGER_ENDPOINT	不要	**traces** エンドポイント。クライアントアプリケーションが **jaeger-agent** を迂回し、**jaeger-collector** に直接接続する場合にのみ、この変数を定義します。
JAEGER_AUTH_TOKEN	不要	エンドポイントに bearer トークンとして送信する認証トークン。
JAEGER_USER	不要	Basic 認証を使用する場合にエンドポイントに送信するユーザー名。
JAEGER_PASSWORD	不要	Basic 認証を使用する場合にエンドポイントに送信するパスワード。
JAEGER_PROPAGATION	不要	トレースコンテキストの伝播に使用するカンマ区切りの形式リスト。デフォルトは標準の Jaeger 形式です。有効な値は **jaeger** および **b3** です。
JAEGER_REPORTER_LOG_SPANS	不要	レポーターがスパンも記録する必要があるかどうかを示します。
JAEGER_REPORTER_MAX_QUEUE_SIZE	不要	レポーターの最大キューサイズ。
JAEGER_REPORTER_FLUSH_INTERVAL	不要	レポーターのフラッシュ間隔 (ミリ秒単位)。Jaeger レポーターがスパンバッチをフラッシュする頻度を定義します。

プロパティー	必要性	説明
JAEGER_SAMPLER_TYPE	不要	クライアントトレースに使用するサンプリングストラテジー。 ● Constant ● Probabilistic ● Rate Limiting ● Remote (デフォルト) すべてのトレースをサンプリングするには、Constant サンプリングストラテジーを使用し、パラメーターを 1 にします。 詳細は、Jaeger ドキュメント を参照してください。
JAEGER_SAMPLER_PARAM	不要	サンプラーのパラメーター (数値)。
JAEGER_SAMPLER_MANAGER_HOST_PORT	不要	リモートサンプリングストラテジーを選択する場合に使用するホスト名およびポート。
JAEGER_TAGS	不要	報告されたすべてのスパンに追加されるトレーサーレベルのタグのカンマ区切りリスト。 この値に **${envVarName:default}** 形式を使用して環境変数を参照することもできます。**:default** は任意の設定で、環境変数が見つからない場合に使用する値を特定します。

第17章 KAFKA EXPORTER

Kafka Exporter は、Apache Kafka ブローカーおよびクライアントの監視を強化するオープンソースプロジェクトです。

Kafka Exporter は、Kafka クラスターとのデプロイメントを実現するために AMQ Streams で提供され、オフセット、コンシューマーグループ、コンシューマーラグ、およびトピックに関連する Kafka ブローカーから追加のメトリクスデータを抽出します。

一例として、メトリクスデータを使用すると、低速なコンシューマーの識別に役立ちます。

ラグデータは Prometheus メトリクスとして公開され、解析のために Grafana で使用できます。

ビルトイン Kafka メトリクスを監視するために Prometheus および Grafana をすでに使用している場合、Kafka Exporter Prometheus エンドポイントをスクレープするように Prometheus を設定することもできます。

関連情報

Kafka は JMX 経由でメトリクスを公開し、Prometheus メトリクスとしてエクスポートすることができます。

- *8章JMX を使用したクラスターの監視*

17.1. コンシューマーラグ

コンシューマーラグは、メッセージの生成と消費の差を示しています。具体的には、指定のコンシューマーグループのコンシューマーラグは、パーティションの最後のメッセージと、そのコンシューマーが現在ピックアップしているメッセージとの時間差を示しています。ラグには、パーティションログの最後を基準とする、コンシューマーオフセットの相対的な位置が反映されます。

この差は、Kafka ブローカートピックパーティションの読み取りと書き込みの場所である、プロデューサーオフセットとコンシューマーオフセットの間の **デルタ** とも呼ばれます。

あるトピックで毎秒 100 個のメッセージがストリーミングされる場合を考えてみましょう。プロデューサーオフセット (トピックパーティションの先頭) と、コンシューマーが読み取った最後のオフセットとの間のラグが 1000 個のメッセージであれば、10 秒の遅延があることを意味します。

コンシューマーラグ監視の重要性

可能な限りリアルタイムのデータの処理に依存するアプリケーションでは、コンシューマーラグを監視して、ラグが過度に大きくならないようにチェックする必要があります。ラグが大きくなるほど、リアルタイム処理の達成から遠ざかります。

たとえば、パージされていない古いデータの大量消費や、予定外のシャットダウンが、コンシューマーラグの原因となることがあります。

コンシューマーラグの削減

通常、ラグを削減するには以下を行います。

- 新規コンシューマーを追加してコンシューマーグループをスケールアップします。
- メッセージがトピックに留まる保持時間を延長します。
- ディスク容量を追加してメッセージバッファーを増強します。

コンシューマーラグを減らす方法は、基礎となるインフラストラクチャーや、AMQ Streams によりサポートされるユースケースによって異なります。たとえば、ラグが生じているコンシューマーの場合、ディスクキャッシュからフェッチリクエストに対応できるブローカーを活用できる可能性は低いでしょう。場合によっては、コンシューマーの状態が改善されるまで、自動的にメッセージをドロップすることが許容されることがあります。

17.2. KAFKA EXPORTER アラートルールの例

Kafka Exporter に固有のサンプルのアラート通知ルールには以下があります。

UnderReplicatedPartition

トピックで複製の数が最低数未満であり、ブローカーがパーティションで複製されないことを警告するアラートです。デフォルトの設定では、トピックに複製の数が最低数未満のパーティションが 1 つ以上ある場合のアラートになります。このアラートは、Kafka インスタンスがダウンしているか Kafka クラスターがオーバーロードの状態であることを示す場合があります。レプリケーションプロセスを再起動するには、Kafka ブローカーの計画的な再起動が必要な場合があります。

TooLargeConsumerGroupLag

特定のトピックパーティションでコンシューマーグループのラグが大きすぎることを警告するアラートです。デフォルト設定は 1000 レコードです。ラグが大きい場合、コンシューマーが遅すぎてプロデューサーの処理に追い付いてない可能性があります。

NoMessageForTooLong

トピックが一定期間にわたりメッセージを受信していないことを警告するアラートです。この期間のデフォルト設定は 10 分です。この遅れは、設定の問題により、プロデューサーがトピックにメッセージを公開できないことが原因である可能性があります。

特定のニーズに合わせてアラートルールを調整できます。

関連情報

アラートルールの設定についての詳細は、Prometheus ドキュメントの「Configuration」を参照してください。

17.3. KAFKA EXPORTER メトリクス

ラグ情報は、Grafana で示す Prometheus メトリクスとして Kafka Exporter によって公開されます。

Kafka Exporter は、ブローカー、トピック、およびコンシューマーグループのメトリクスデータを公開します。

表17.1 ブローカーメトリクスの出力

名前	詳細
kafka_brokers	Kafka クラスターに含まれるブローカーの数

表17.2 トピックメトリクスの出力

名前	詳細
kafka_topic_partitions	トピックのパーティション数

名前	詳細
kafka_topic_partition_current_offset	ブローカーの現在のトピックパーティションオフセット
kafka_topic_partition_oldest_offset	ブローカーの最も古いトピックパーティションオフセット
kafka_topic_partition_in_sync_replica	トピックパーティションの In-Sync レプリカ数
kafka_topic_partition_leader	トピックパーティションのリーダーブローカー ID
kafka_topic_partition_leader_is_preferred	トピックパーティションが優先ブローカーを使用している場合は、**1** が示されます。
kafka_topic_partition_replicas	このトピックパーティションのレプリカ数
kafka_topic_partition_under_replicated_partition	トピックパーティションの複製の数が最低数未満である場合に **1** が示されます。

表17.3 コンシューマーグループメトリクスの出力

名前	詳細
kafka_consumergroup_current_offset	コンシューマーグループの現在のトピックパーティションオフセット
kafka_consumergroup_lag	トピックパーティションのコンシューマーグループの現在のラグ (概算値)

17.4. KAFKA EXPORTER の実行

Kafka Exporter は、AMQ Streams のインストール に使用されるダウンロードアーカイブで提供されます。

このコマンドを実行して、Grafana ダッシュボードで Prometheus メトリクスを公開できます。

前提条件

- AMQ Streams がホストにインストールされていること。

この手順では、Grafana ユーザーインターフェースにアクセスでき、Prometheus がデータソースとして追加されていることを前提とします。

手順

1. 適切な設定値を使用して Kafka Exporter スクリプトを実行します。

```
./bin/kafka_exporter --kafka.server=<kafka-bootstrap-address>:9092 --kafka.version=2.7.0
--<my-other-parameters>
```

パラメーターには、**--kafka.server** のように 2 倍の規則が必要です。

表17.4 Kafka Exporter 設定パラメーター

オプション	説明	デフォルト
kafka.server	Kafka サーバーのホスト/ポストアドレス。	**kafka:9092**
kafka.version	Kafka ブローカーのバージョン。	**1.0.0**
group.filter	メトリクスに含まれるコンシューマーグループを指定する正規表現。	**.*** (all)
topic.filter	メトリクスに含まれるトピックを指定する正規表現。	**.*** (all)
sasl.<parameter>	ユーザー名とパスワードを使用して SASL/PLAIN 認証を使用して Kafka クラスターを有効にし、接続するためのパラメーター。	**false**
tls.<parameter>	任意の証明書およびキーを使用して、TLS 認証を使用して Kafka クラスターへの接続を有効にするパラメーター。	**false**
web.listen-address	メトリクスを公開するポートアドレス。	**:9308**
web.telemetry-path	公開されるメトリクスのパス。	**/metrics**
log.level	指定の重大度 (debug、info、warn、error、fatal) 以上でメッセージをログに記録するためのログ設定。	**info**
log.enable-sarama	Sarama ロギングを有効にするブール値（Kafka Exporter によって使用される Go クライアントライブラリー）。	**false**

プロパティーの情報には **kafka_exporter --help** を使用できます。

2. Kafka Exporter メトリクスを監視するように Prometheus を設定します。
   Prometheus の設定に関する詳細は、Prometheus の ドキュメント を参照してください。

3. Grafana を有効にして、Prometheus によって公開される Kafka Exporter メトリクスデータを表示します。
   詳細は、「 Grafana での Kafka Exporter メトリクスの 表示」を参照してください。

17.5. GRAFANA での KAFKA EXPORTER メトリクスの表示

Kafka Exporter Prometheus メトリクスをデータソースとして使用すると、Grafana チャートのダッシュボードを作成できます。

たとえば、メトリクスから以下の Grafana チャートを作成できます。

- 毎秒のメッセージ (トピックから)
- 毎分のメッセージ (トピックから)
- コンシューマーグループごとのラグ
- 毎分のメッセージ消費 (コンシューマーグループごと)

メトリクスデータが収集されると、Kafka Exporter のチャートにデータが反映されます。

Grafana のチャートを使用して、ラグを分析し、ラグ削減の方法が対象のコンシューマーグループに影響しているかどうかを確認します。たとえば、ラグを減らすように Kafka ブローカーを調整すると、ダッシュボードには **コンシューマーグループごとのラグ** のチャートが下降し **毎分のメッセージ消費** のチャートが上昇する状況が示されます。

関連情報

- Kafka Exporter のダッシュボードの例
- Grafana ドキュメント

第18章 AMQ STREAMS および KAFKA のアップグレード

AMQ Streams は、クラスターのダウンタイムを発生せずにアップグレードできます。AMQ Streams の各バージョンは、Apache Kafka の 1 つ以上のバージョンをサポートします。使用する AMQ Streams バージョンでサポートされれば、より高いバージョンの Kafka にアップグレードできます。より新しいバージョンの AMQ Streams はより新しいバージョンの Kafka をサポートしますが、AMQ Streams を**アップグレードしてから**、サポートされる上位バージョンの Kafka にアップグレードする必要があります。

18.1. アップグレードの前提条件

アップグレードプロセスを開始する前に、以下を確認します。

- AMQ Streams がインストールされている必要があります。手順は *2章スタートガイド* を参照してください。
- 「 AMQ Streams 1.7 on Red Hat Enterprise Linux リリースノート 」に記載されているアップグレードの変更について理解している必要があります。

18.2. アップグレードプロセス

AMQ Streams のアップグレードは 2 段階のプロセスで行います。ダウンタイムなしでブローカーとクライアントをアップグレードするには、以下の順序でアップグレード手順を **必ず** 完了してください。

1. 最新の AMQ Streams バージョンにアップグレードします。
   - AMQ Streams 1.7 へのアップグレード
2. すべての Kafka ブローカーとクライアントアプリケーションを最新の Kafka バージョンにアップグレードします。
   - Kafka のアップグレード

18.3. KAFKA バージョン

Kafka のログメッセージ形式バージョンとブローカー間のプロトコルバージョンは、それぞれメッセージに追加されるログ形式バージョンとクラスターで使用される Kafka プロトコルのバージョンを指定します。正しいバージョンが使用されるようにするため、アップグレードプロセスでは、既存の Kafka ブローカーの設定変更と、クライアントアプリケーション (コンシューマーおよびプロデューサー) のコード変更が行われます。

以下の表は、Kafka バージョンの違いを示しています。

Kafka のバージョン	Interbroker プロトコルのバージョン	ログメッセージ形式のバージョン	ZooKeeper のバージョン
2.6.0	2.6	2.6	3.5.8
2.7.0	2.7	2.7	3.5.8

ブローカー間のプロトコルバージョン

Kafka では、ブローカー間の通信に使用されるネットワークプロトコルは**ブローカー間プロトコル**

(Inter-broker protocol) と呼ばれます。Kafka の各バージョンには、互換性のあるバージョンのブローカー間プロトコルがあります。上記の表が示すように、プロトコルのマイナーバージョンは、通常 Kafka のマイナーバージョンと一致するように番号が増加されます。

ブローカー間プロトコルのバージョンは、**Kafka** リソースでクラスター全体に設定されます。これを変更するには、**Kafka.spec.kafka.config** の **inter.broker.protocol.version** プロパティーを編集します。

ログメッセージ形式のバージョン

プロデューサーが Kafka ブローカーにメッセージを送信すると、特定の形式を使用してメッセージがエンコードされます。この形式は Kafka のリリース間で変更される可能性があるため、メッセージにはエンコードに使用された形式のバージョンが指定されます。ブローカーがメッセージをログに追加する前に、メッセージを新しい形式バージョンから特定の旧形式バージョンに変換するように、Kafka ブローカーを設定できます。

Kafka には、メッセージ形式のバージョンを設定する 2 通りの方法があります。

- **message.format.version** プロパティーはトピックに設定されます。
- **log.message.format.version** プロパティーは Kafka ブローカーに設定されます。

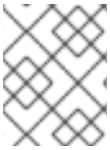
トピックの **message.format.version** のデフォルト値は、Kafka ブローカーに設定される **log.message.format.version** によって定義されます。トピックの **message.format.version** は、トピック設定を編集すると手動で設定できます。

本セクションのアップグレード作業では、メッセージ形式のバージョンが **log.message.format.version** によって定義されることを前提としています。

18.4. AMQ STREAMS 1.7 へのアップグレード

このセクションでは、AMQ Streams 1.7 を使用するようにデプロイメントをアップグレードする手順について説明します。

AMQ Streams によって管理される Kafka クラスターの可用性は、アップグレード操作による影響を受けません。



注記

特定バージョンの AMQ Streams へのアップグレード方法については、そのバージョンをサポートするドキュメントを参照してください。

18.4.1. Kafka ブローカーおよび ZooKeeper のアップグレード

この手順では、ホストマシンで Kafka ブローカーおよび ZooKeeper をアップグレードし、最新バージョンの AMQ Streams を使用する方法について説明します。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと 1 つずつ、以下の 1 つに対して以下を行います。

1. カスタマーポータルから AMQ Streams アーカイブを ダウンロード します。

注記

要求されたら、Red Hat アカウントにログインします。

2. コマンドラインで一時ディレクトリーを作成し、**amq-streams-x.y.z-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-x.y.z-bin.zip -d /tmp/kafka
```

3. 実行している場合は、ZooKeeper およびホストで実行されている Kafka ブローカーを停止します。

```
/opt/kafka/bin/zookeeper-server-stop.sh
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep zookeeper
jcmd | grep kafka
```

4. 既存のインストールから **libs**、**bin**、および **docs** ディレクトリーを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin /opt/kafka/docs
```

5. 一時ディレクトリーから **libs**、**bin**、および **docs** ディレクトリーをコピーします。

```
cp -r /tmp/kafka/kafka_y.y-x.x.x/libs /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/docs /opt/kafka/
```

6. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

7. テキストエディターで、一般的に **/opt/kafka/config/** ディレクトリーに保存されているブローカープロパティーファイルを開きます。

8. **inter.broker.protocol.version** および **log.message.format.version** プロパティーが **現行** バージョンに設定されていることを確認します。

```
inter.broker.protocol.version=2.6
log.message.format.version=2.6
```

inter.broker.protocol.version を変更しないと、ブローカーはアップグレード中も相互に通信を継続できます。

プロパティーが設定されていない場合は、現行バージョンで追加します。

9. 更新された ZooKeeper および Kafka ブローカーを再起動します。

```
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

Kafka ブローカーおよび Zookeeper は、最新の Kafka バージョンのバイナリーの使用を開始します。

10. 再起動した Kafka ブローカーが、以下になっているパーティションレプリカでキャッチされたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリカを同期していることを確認します。手順は「 トピックの一覧表示および説明 」を参照してください。

11. 「Kafka のアップグレード」 の説明通りに、Kafka をアップグレードする手順を実行します。

18.4.2. Kafka Connect のアップグレード

この手順では、ホストマシンで Kafka Connect クラスターをアップグレードする方法を説明します。

Kafka Connect はクライアントアプリケーションで、クライアントをアップグレードするために選択したストラテジーに組み込む必要があります。詳細は、「 クライアントをアップグレードするストラテジー 」を参照してください。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- Kafka Connect は起動していません。

手順

AMQ Streams クラスターの各 Kafka ブローカーと 1 つずつ、以下の 1 つに対して以下を行います。

1. カスタマーポータルから AMQ Streams アーカイブを ダウンロード します。

注記

要求されたら、Red Hat アカウントにログインします。

2. コマンドラインで一時ディレクトリーを作成し、**amq-streams-x.y.z-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-x.y.z-bin.zip -d /tmp/kafka
```

3. 実行している場合は、ホストで実行している Kafka ブローカーおよび ZooKeeper を停止します。

```
/opt/kafka/bin/kafka-server-stop.sh
/opt/kafka/bin/zookeeper-server-stop.sh
```

4. 既存のインストールから **libs**、**bin**、および **docs** ディレクトリーを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin /opt/kafka/docs
```

5. 一時ディレクトリーから **libs**、**bin**、および **docs** ディレクトリーをコピーします。

```
cp -r /tmp/kafka/kafka_y.y-x.x.x/libs /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/docs /opt/kafka/
```

6. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

7. スタンドアロンまたは分散モードで Kafka Connect を起動します。
   - スタンドアロンモードで開始するには、**connect-standalone.sh** スクリプトを実行します。Kafka Connect スタンドアロン設定ファイルおよび Kafka Connect コネクターの設定ファイルを指定します。

     ```
     su - kafka
     /opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties
     connector1.properties
     [connector2.properties ...]
     ```

   - 分散モードで起動するには、すべての Kafka Connect ノードで **/opt/kafka/config/connect-distributed.properties** 設定ファイルを使用して Kafka Connect ワーカーを起動します。

     ```
     su - kafka
     /opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
     ```

8. Kafka Connect が実行されていることを確認します。
   - スタンドアロンモードでは、以下のようになります。

     ```
     jcmd | grep ConnectStandalone
     ```

   - Distributed モードでは以下を行います。

     ```
     jcmd | grep ConnectDistributed
     ```

9. Kafka Connect が想定どおりにデータを生成し、消費していることを確認します。

関連情報

- スタンドアロンモードでの Kafka Connect の実行
- 分散 Kafka Connect の実行
- クライアントをアップグレードするストラテジー

18.5. KAFKA のアップグレード

最新バージョンの AMQ Streams を使用するようにバイナリーをアップグレード した後、ブローカーおよびクライアントをアップグレードして、サポートされる上位バージョンの Kafka を使用できます。

正しい順序で手順を行ってください。

1. 「新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード」
2. 「クライアントアプリケーションの新しい Kafka バージョンへのアップグレード」
3. 「新しいメッセージ形式バージョンを使用するように Kafka ブローカーのアップグレード」

Kafka のアップグレードに従い、必要な場合は Kafka コンシューマーをアップグレードして Incremental Cooperative Rebalance プロトコルを使用できます。

1. 「コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード」

18.5.1. 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード

新しいブローカー間プロトコルバージョンを使用するように、すべての Kafka ブローカーを手動で設定し、再起動します。これらの手順を完了すると、新しいブローカー間プロトコルバージョンを使用して、Kafka ブローカーの間でデータが送信されます。

受け取ったメッセージは、メッセージログには以前のメッセージ形式のバージョンで引き続き追加されます。

警告

この手順の完了後、AMQ Streams のダウングレードは実行できません。

前提条件

- ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.7 にアップグレードしている。
- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと 1 つずつ、以下の 1 つに対して以下を行います。

1. テキストエディターで、更新する Kafka ブローカーのブローカープロパティーファイルを開きます。ブローカーのプロパティーファイルは通常 **/opt/kafka/config/** ディレクトリーに保存されます。
2. **inter.broker.protocol.version** を **2.7** に設定します。

```
inter.broker.protocol.version=2.7
```

3. コマンドラインで、変更した Kafka ブローカーを停止します。

```
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep kafka
```

4. 変更した Kafka ブローカーを再起動します。

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

5. 再起動した Kafka ブローカーが、以下になっているパーティションレプリカでキャッチされたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリ

カを同期していることを確認します。手順は「トピックの一覧表示および説明」を参照してください。

18.5.2. クライアントをアップグレードするストラテジー

クライアントアプリケーション (Kafka Connect コネクターを含む) のアップグレードに適切な方法は、特定の状況によって異なります。

消費するアプリケーションは、そのアプリケーションが理解するメッセージ形式のメッセージを受信する必要があります。その状態であることを、以下のいずれかの方法で確認できます。

- プロデューサーをアップグレードする **前に**、トピックのすべてのコンシューマーをアップグレードする。
- ブローカーでメッセージをダウンコンバートする。

ブローカーのダウンコンバートを使用すると、ブローカーに余分な負荷が加わるので、すべてのトピックで長期にわたりダウンコンバートに頼るのは最適な方法ではありません。ブローカーの実行を最適化するには、ブローカーがメッセージを一切ダウンコンバートしないようにしてください。

ブローカーのダウンコンバートは 2 通りの方法で設定できます。

- トピックレベルの **message.format.version** では単一のとピックが設定されます。
- ブローカーレベルの **log.message.format.version** は、トピックレベルの **message.format.version** が設定されてないトピックのデフォルトです。

新バージョンの形式でトピックにパブリッシュされるメッセージは、コンシューマーによって認識されます。これは、メッセージがコンシューマーに送信されるときでなく、ブローカーがプロデューサーからメッセージを受信するときに、ブローカーがダウンコンバートを実行するからです。

クライアントのアップグレードに使用できるストラテジーは複数あります。

コンシューマーを最初にアップグレード

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. ブローカーレベルの **log.message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。
   このストラテジーは分かりやすく、ブローカーのダウンコンバートの発生をすべて防ぎます。ただし、所属組織内のすべてのコンシューマーを整然とアップグレードできることが前提になります。また、コンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。さらにリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加され、以前のコンシューマーバージョンに戻せなくなる場合があります。

トピック単位でコンシューマーを最初にアップグレード

トピックごとに以下を実行します。

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. トピックレベルの **message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。
   このストラテジーではブローカーのダウンコンバートがすべて回避され、トピックごとに

アップグレードできます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。ここでもリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加される可能性があります。

トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり

トピックごとに以下を実行します。

1. トピックレベルの **message.format.version** を、旧バージョンに変更します (または、デフォルトがブローカーレベルの **log.message.format.version** のトピックを利用します)。
2. コンシューマーおよびプロデューサーとして機能するアプリケーションをすべてアップグレードします。
3. アップグレードしたアプリケーションが正しく機能することを確認します。
4. トピックレベルの **message.format.version** を新バージョンに変更します。
このストラテジーにはブローカーのダウンコンバートが必要ですが、ダウンコンバートは一度に 1 つのトピック (またはトピックの小さなグループ) のみに必要になるので、ブローカーへの負荷は最小限に抑えられます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションにも通用します。この方法により、新しいメッセージ形式バージョンを使用する前に、アップグレードされたプロデューサーとコンシューマーが正しく機能することが保証されます。

   この方法の主な欠点は、多くのトピックやアプリケーションが含まれるクラスターでの管理が複雑になる場合があることです。

クライアントアプリケーションをアップグレードするストラテジーは他にもあります。

注記

複数のストラテジーを適用することもできます。たとえば、最初のいくつかのアプリケーションとトピックに、「トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり」のストラテジーを適用します。これが問題なく適用されたら、より効率的な別のストラテジーの使用を検討できます。

18.5.3. クライアントアプリケーションの新しい Kafka バージョンへのアップグレード

この手順では、AMQ Streams 1.7 に使用される Kafka バージョンにクライアントアプリケーションをアップグレードする方法の 1 つを説明します。

この手順は、「 クライアントをアップグレードするストラテジー」で説明されている「トピック単位でコンシューマーを最初にアップグレードするため の変換」のアプローチを基にしています。

クライアントアプリケーションには、プロデューサー、コンシューマー、Kafka Connect、Kafka Streams アプリケーション、および MirrorMaker が含まれます。

前提条件

- ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.7 にアップグレードしている。
- 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーが設定されている必要があります。

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

トピックごとに以下を実行します。

1. コマンドラインで、**message.format.version** 設定オプションを **2.6** に設定します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
<TopicName> --alter --add-config message.format.version=2.6
```

2. トピックのコンシューマーおよびプロデューサーをすべてアップグレードします。
3. 必要に応じて、コンシューマーおよび Kafka Streams アプリケーションをアップグレードして、Kafka 2.4.0 に追加された **増分 Cooperative Rebalance** プロトコルを使用するようにアップグレードするには、「コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード」 を参照してください。
4. アップグレードしたアプリケーションが正しく機能することを確認します。
5. トピックの **message.format.version** 設定オプションを **2.7** に変更します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
<TopicName> --alter --add-config message.format.version=2.7
```

関連情報

- クライアントをアップグレードするストラテジー

18.5.4. 新しいメッセージ形式バージョンを使用するように Kafka ブローカーのアップグレード

クライアントアプリケーションがアップグレードされると、新しいメッセージ形式バージョンを使用するように Kafka ブローカーを更新できます。

AMQ Streams 1.7 に必要な Kafka バージョンを使用するようにクライアントアプリケーションをアップグレードするときにトピック設定を変更し **ない** と、Kafka ブローカーはメッセージを以前のメッセージ形式バージョンに変換できるようになりました。これにより、パフォーマンスが低下する可能性があります。そのため、すべての Kafka ブローカーを更新して、できるだけ早く新しいメッセージ形式のバージョンを使用するようにすることが重要です。

注記

Kafka ブローカーを 1 つずつ更新および再起動します。変更したブローカーを再起動する前に、以前に設定および再起動したブローカーを停止します。

前提条件

- ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.7 にアップグレードしている。
- 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーが設定されている必要があります。

- **message.format.version** プロパティーがトピックレベルで明示的に設定されて **いない** トピックからメッセージを消費するサポートされるクライアントアプリケーションをアップグレードしました。
- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと 1 つずつ、以下の 1 つに対して以下を行います。

1. テキストエディターで、更新する Kafka ブローカーのブローカープロパティーファイルを開きます。ブローカーのプロパティーファイルは通常 **/opt/kafka/config/** ディレクトリーに保存されます。
2. **log.message.format.version** を **2.7** に設定します。

```
log.message.format.version=2.7
```

3. コマンドラインで、この手順の一部として、最近変更した Kafka ブローカーを停止します。この手順の最初の Kafka ブローカーを変更する場合には、ステップ 4 に移動します。

```
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep kafka
```

4. 設定の手順 2 で変更した Kafka ブローカーを再起動します。

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

5. 再起動した Kafka ブローカーが、以下になっているパーティションレプリカでキャッチされたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリカを同期していることを確認します。手順は「トピックの一覧表示および説明」を参照してください。

18.5.5. コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード

Kafka コンシューマーおよび Kafka Streams アプリケーションをアップグレードすることで、パーティションの再分散にデフォルトの **Eager Rebalance** プロトコルではなく **Incremental Cooperative Rebalance** プロトコルを使用できます。この新しいプロトコルが Kafka 2.4.0 に追加されました。

コンシューマーは、パーティションの割り当てを Cooperative Rebalance で保持し、クラスターの分散が必要な場合にプロセスの最後でのみ割り当てを取り消します。これにより、コンシューマーグループまたは Kafka Streams アプリケーションが使用不可能になる状態が削減されます。

注記

Incremental Cooperative Rebalance プロトコルへのアップグレードは任意です。Eager Rebalance プロトコルは引き続きサポートされます。

前提条件

- 「AMQ Streams 1.7 へのアップグレード」
- 「新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード」

- 「クライアントアプリケーションの新しい Kafka バージョンへのアップグレード」

手順

Incremental Cooperative Rebalance プロトコルを使用するように Kafka コンシューマーをアップグレードするには以下を行います。

1. Kafka クライアント **.jar** ファイルを新バージョンに置き換えます。
2. コンシューマー設定で、**partition.assignment.strategy** に **cooperative-sticky** を追加します。たとえば、**range** ストラテジーが設定されている場合は、設定を **range, cooperative-sticky** に変更します。
3. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。
4. コンシューマー設定から前述の **partition.assignment.strategy** を削除して、グループの各コンシューマーを再設定し、**cooperative-sticky** ストラテジーのみを残します。
5. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。

Incremental Cooperative Rebalance プロトコルを使用するように Kafka Streams アプリケーションをアップグレードするには以下を行います。

1. Kafka Streams の **.jar** ファイルを新バージョンに置き換えます。
2. Kafka Streams の設定で、**upgrade.from** 設定パラメーターをアップグレード前の Kafka バージョンに設定します (例: 2.3)。
3. 各ストリームプロセッサー (ノード) を順次再起動します。
4. **upgrade.from** 設定パラメーターを Kafka Streams 設定から削除します。
5. グループ内の各コンシューマーを順次再起動します。

その他のリソース

- Apache Kafka ドキュメントの「Notable changes in 2.4.0 」

付録A ブローカー設定パラメーター

zookeeper.connect

type: string
Importance: high
Dynamic update: read-only
hostname:port の形式で ZooKeeper 接続文字列を指定します。ここで、ホストとポートは ZooKeeper サーバーのホストとポートになります。ZooKeeper マシンがダウンした場合の他の ZooKeeper ノードへの接続を許可するには、**hostname1:port1,hostname2:port2,hostname3:port3** の形式で複数のホストを指定することもできます。サーバーには ZooKeeper 接続文字列の一部として ZooKeeper chroot パスを持つこともできます。このパスは、データをグローバル ZooKeeper namespace の一部のパスに配置します。たとえば、**/chroot/path** の chroot パスを指定する場合は、接続文字列を **hostname1:port1,hostname2:port2,hostname3:port3/chroot/path** に指定します。

advertised.host.name

type: string
デフォルト： null
Importance: high
Dynamic update: read-only
非推奨： **advertised.listeners** または **listeners** が設定されていない場合にのみ使用されます。代わりに **advertised.listeners** を使用してください。クライアントが使用するために ZooKeeper にパブリッシュするホスト名。IaaS 環境では、ブローカーがバインドするインターフェースとは異なる場合があります。これが設定されていない場合は、**host.name** の値を使用します。そうしないと、java.net.InetAddress.getCanonicalHostName（）から返された値を使用します。

advertised.listeners

type: string
デフォルト： null
Importance:**high**
Dynamic update: high**Dynamic update:** per-broker
クライアントが使用する ZooKeeper に公開するリスナー（**listeners** 設定プロパティーとは異なる場合）。IaaS 環境では、ブローカーがバインドするインターフェースとは異なる場合があります。これを設定しないと、**listeners** の値が使用されます。**listeners** とは異なり、0.0.0.0 meta-address を公開することはできません。また、**listeners** とは異なり、このプロパティーにポートが複製され、1つのリスナーが別のリスナーのアドレスを公開するように設定できます。これは、外部ロードバランサーが使用される場合に役立ちます。

advertised.port

type: int
デフォルト： null
Importance: high
Dynamic update: read-only
非推奨： **advertised.listeners** または **listeners** が設定されていない場合にのみ使用されます。代わりに **advertised.listeners** を使用してください。クライアントが使用するために ZooKeeper にパブリッシュするポート。IaaS 環境では、ブローカーがバインドするポートとは異なる場合があります。これが設定されていない場合、ブローカーがバインドするのと同じポートを公開します。

auto.create.topics.enable

type: boolean
デフォルト： true
Importance: high
Dynamic update: read-only

サーバーでトピックの自動作成を有効にします。

auto.leader.rebalance.enable

type: boolean
デフォルト： true
Importance: high
Dynamic update: read-only
自動リーダーの分散を有効にします。バックグラウンドスレッドは、**leader.imbalance.check.interval.seconds** によって設定可能なパーティションリーダーの分布をチェックします。リーダーのリバランスが **leader.imbalance.per.broker.percentage** を超える場合、パーティションの推奨されるリーダーへのリーダーリバランスがトリガーされます。

background.threads

型： int
デフォルト： 10
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
さまざまなバックグラウンド処理タスクに使用するスレッドの数。

broker.id

型： int
デフォルト： -1
Importance: high
Dynamic update: read-only
このサーバーのブローカー ID。設定されていない場合は、一意のブローカー ID が生成されます。zookeeper 生成されたブローカー ID とユーザー設定ブローカー ID の競合を避けるため、生成されたブローカー ID は reserved.broker.max.id + 1 から開始します。

compression.type

type: string
デフォルト： producer
Importance: high
Dynamic update: cluster-wide
特定のトピックの最終圧縮タイプを指定します。この設定では、標準の圧縮コーデック（'gzip'、'snappy'、'lz4'、'zstd'）を受け入れます。また、圧縮なしでの 'uncompressed' および 'producer' を受け入れます。これは、プロデューサーによって設定された元の圧縮コード c を保持します。

control.plane.listener.name

type: string
デフォルト： null
Importance: high
Dynamic update: read-only
コントローラーとブローカー間の通信に使用されるリスナーの名前。ブローカーは control.plane.listener.name を使用して、リスナー一覧でエンドポイントを特定し、コントローラーからの接続をリッスンします。たとえば、ブローカーの設定が : listeners = INTERNAL://192.1.1.8:9092、EXTERNAL://10.1.1.5:9093、CONTROLLER://192.1.1.8:9094 listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL control.plane.listener.name = CONTROLLER On startup で "SSL" CONTROLLER://192.1.1.8:9098 プロトコルで起動するように起動するようにする場合は、192.SSL プロトコルを起動します。コントローラー側では、zookeeper 経由でブローカーのパブリッシュされたエンドポイントを検出すると、control.plane.listener.name を使用してブローカーへの接続を確立するために使用するエンドポイントを見つけます。たとえば、ZooKeeper でブローカーのパブリッシュエンドポイントが :

listener.security.protocol.map : ["INTERNAL://broker1.example.com:9092","EXTERNAL://broker1.example.com:9093","CONTROLLER とコントローラーの設定が : listener.security.protocol.map = INT.SSL to connect:SSLAINT:SSL INT.SSL to connect to connect, INT明示的に設定されていない場合、デフォルト値は null で、コントローラー接続の専用のエンドポイントはありません。

delete.topic.enable

type: boolean
デフォルト： true
Importance: high
Dynamic update: read-only
トピックの削除を有効にします。この構成がオフの場合は、管理ツールからトピックを削除することによる影響はありません。

host.name

type: string
デフォルト： ""
Importance: high
Dynamic update: read-only
非推奨： **listeners** が設定されていない場合にのみ使用されます。代わりに **listeners** を使用してください。ブローカーのホスト名を指定します。これが設定されている場合、このアドレスにのみバインドされます。これが設定されていない場合、全インターフェースにバインドされます。

leader.imbalance.check.interval.seconds

タイプ： long
デフォルト： 300
Importance: high
Dynamic update: read-only
パーティションリバランスチェックがコントローラーによってトリガーされる頻度。

leader.imbalance.per.broker.percentage

型： int
デフォルト： 10
Importance: high
Dynamic update: read-only
ブローカーごとに許可されるリーダー分散の比率。ブローカーごとにこの値を超えると、コントローラーはリーダーのバランスをトリガーします。値はパーセンテージで指定します。

listeners

type: string
デフォルト： null
Importance:**high**
Dynamic update: high**Dynamic update:** per-broker
リスナー一覧： リッスンする URI のコンマ区切りリストおよびリスナー名。リスナー名がセキュリティープロトコルではない場合、**listener.security.protocol.map** も設定する必要があります。リスナー名とポート番号は一意でなければなりません。ホスト名を 0.0.0.0 として指定して、全インターフェースにバインドします。デフォルトのインターフェースにバインドするには、hostname は空のままにします。法的リスナーの一覧の例： PLAINTEXT://myhost:9092,SSL://:9091 CLIENT://0.0.0.0:9092,REPLICATION://localhost:9093

log.dir

型： 文字列

デフォルト： /tmp/kafka-logs
Importance: high
Dynamic update: read-only
ログデータが保存されるディレクトリー（log.dirs プロパティーの補助グループ）。

log.dirs

type: string
デフォルト： null
Importance: high
Dynamic update: read-only
ログデータが保持されるディレクトリー。設定されていない場合は、log.dir の値が使用されます。

log.flush.interval.messages

タイプ： long
デフォルト： 9223372036854775807
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
メッセージがディスクにフラッシュされる前に、ログパーティションで累積されたメッセージの数。

log.flush.interval.ms

type: long
デフォルト： null
Importance: high
Dynamic update: cluster-wide
ディスクにフラッシュされる前にすべてのトピックのメッセージがメモリーに保持される最大時間（ミリ秒単位）。設定されていない場合は、log.flush.scheduler.interval.ms の値が使用されます。

log.flush.offset.checkpoint.interval.ms

型： int
デフォルト： 60000（1分）
有効な値： [0,...]
Importance: high
Dynamic update: read-only
ログのリカバリーポイントとして動作する最後のフラッシュの永続レコードを更新する頻度。

log.flush.scheduler.interval.ms

タイプ： long
デフォルト： 9223372036854775807
Importance: high
Dynamic update: read-only
ログフラッシュ機能の頻度（ミリ秒単位）。ログがディスクにフラッシュする必要があるかどうかをチェックする時間（ミリ秒単位）。

log.flush.start.offset.checkpoint.interval.ms

型： int
デフォルト： 60000（1分）
有効な値： [0,...]
Importance: high
Dynamic update: read-only
ログ開始オフセットの永続レコードを更新する頻度。

log.retention.bytes

タイプ： long
デフォルト： -1
Importance: high
Dynamic update: cluster-wide
削除前のログの最大サイズ。

log.retention.hours

型： int
デフォルト： 168
Importance: high
Dynamic update: read-only
ログファイルの削除前にログファイルを保持する時間（時間単位）、log.retention.ms プロパティー。

log.retention.minutes

type: int
デフォルト： null
Importance: high
Dynamic update: read-only
log.retention.ms プロパティーのセカンダリーを log.retention.ms プロパティーより前にログファイルを保持する時間（分単位）。設定されていない場合は、log.retention.hours の値が使用されます。

log.retention.ms

type: long
デフォルト： null
Importance: high
Dynamic update: cluster-wide
ログファイルを削除する前に保持する時間（ミリ秒単位）。設定されていない場合には、log.retention.minutes の値が使用されます。-1 に設定すると、時間制限は適用されません。

log.roll.hours

型： int
デフォルト： 168
有効な値： [1,...]
Importance: high
Dynamic update: read-only
新しいログエントリーがロールアウトされる（時間単位）、セカンダリーが log.roll.ms プロパティーにロールアウトされる最大時間。

log.roll.jitter.hours

型： int
デフォルト： 0
有効な値： [0,...]
Importance: high
Dynamic update: read-only
logRollTimeMillis（時間）から log.roll.jitter.ms プロパティーに減算する最大ジター。

log.roll.jitter.ms

type: long
デフォルト： null

Importance: high
Dynamic update: cluster-wide
logRollTimeMillis（ミリ秒単位）から減算する最大ジター。設定されていない場合は、log.roll.jitter.hours の値が使用されます。

log.roll.ms

type: long
デフォルト： null
Importance: high
Dynamic update: cluster-wide
新しいログセグメントがロールアウトされる最大時間（ミリ秒単位）。設定されていない場合は、log.roll.hours の値が使用されます。

log.segment.bytes

型： int
デフォルト： 1073741824(1 gibibyte)
有効な値 [14, ...]
importance: high
Dynamic update: clusterwide
単一ログファイルの最大サイズ。

log.segment.delete.delay.ms

型： long
デフォルト： 60000（1 分）
有効な値： [0,...]
Importance: high
Dynamic update: cluster-wide
ファイルシステムからファイルを削除するまでの待機時間。

message.max.bytes

type: int
Default: 1048588
有効な値： [0,...]
Importance: high
Dynamic update: cluster-wide
Kafka で許可される最大レコードバッチサイズ（圧縮が有効な場合の圧縮後）。これが増加され、0.10.2 よりも古いコンシューマーがある場合、コンシューマーのフェッチサイズも拡張し、この大量に記録できるようにします。最新のメッセージ形式バージョンでは、レコードは常にバッチにグループ化され、効率性を確保します。以前のメッセージ形式のバージョンでは、圧縮されていないレコードはバッチにグループ化されず、この制限はこのような場合の単一レコードにのみ適用されます。これはトピックレベルの **max.message.bytes** 設定でトピックごとに設定できます。

min.insync.replicas

型： int
デフォルト： 1
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
プロデューサーが acks を「all」（または「-1」）に設定すると、min.insync.replicas は書き込みが成功したとみなされる必要のあるレプリカの最小数を指定します。この最小値が満たされない場合、プロデューサーは例外を送出します（NotEnoughReplicas または NotEnoughReplicasAfterAppend）。一緒に使用すると、min.insync.replicas および acks を使用する

ことで、耐久性の高い保証を強制することができます。典型的なシナリオとしては、レプリケーション係数 3 でトピックを作成し、min.insync.replicas を 2 に設定し、acks of「all」で作成します。これにより、多くのレプリカが書き込みを受信しない場合にプロデューサーが例外を発生するようになります。

num.io.threads

型： int
デフォルト： 8
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
サーバーがリクエストの処理に使用するスレッドの数。ディスク I/O を含む可能性があります。

num.network.threads

型： int
デフォルト： 3
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
サーバーがネットワークから要求を受信し、ネットワークへの応答を送信するのに使用するスレッドの数。

num.recovery.threads.per.data.dir

型： int
デフォルト： 1
有効な値 [1,...]
Importance: high
Dynamic update: cluster-wide
シャットダウン時およびフラッシュ時のログ復旧に使用されるデータディレクトリーごとのスレッド数。

num.replica.alter.log.dirs.threads

type: int
デフォルト： null
Importance: high
Dynamic update: read-only
ログディレクトリー間でレプリカを移動できるスレッドの数。ディスク I/O が含まれる場合があります。

num.replica.fetchers

型： int
デフォルト： 1
Importance: high
Dynamic update: cluster-wide
ソースブローカーからメッセージをレプリケートするために使用されるフェッチスレッドの数。この値を増やすと、フォロワーブローカーの I/O 並列処理レベルを増やすことができます。

offset.metadata.max.bytes

タイプ： int
デフォルト： 4096(4 kibibytes)
Importance: high
Dynamic update: read-only

オフセットコミットに関連するメタデータエントリーの最大サイズ。

offsets.commit.required.acks

タイプ： 短い
デフォルト： -1
Importance: high
Dynamic update: read-only
コミットを許可する前に必要なハッキング。通常、デフォルト(-1)は上書きしないでください。

offsets.commit.timeout.ms

型： int
デフォルト： 5000（5 秒）
有効な値 [1,...]
Importance: high
Dynamic update: read-only
オフセットのコミットは、オフセットトピックのすべてのレプリカがコミットを受け取るか、このタイムアウトに達するまで遅延します。これはプロデューサーリクエストのタイムアウトに似ています。

offsets.load.buffer.size

型： int
デフォルト： 5242880
有効な値 [1,...]
Importance: high
Dynamic update: read-only
オフセットをキャッシュにロードする際にオフセットセグメントから読み取りするためのバッチサイズ（レコードが大きすぎるとオーバーライドされます）。

offsets.retention.check.interval.ms

型： long
デフォルト： 600000（10 分）
有効な値： [1,...]
Importance: high
Dynamic update: read-only
古いオフセットをチェックする頻度。

offsets.retention.minutes

型： int
デフォルト： 10080
有効な値 [1,...]
Importance: high
Dynamic update: read-only
コンシューマーグループがそのコンシューマーをすべて失った後（空など）、そのオフセットは、破棄されるまでこの保持期間に保持されます。スタンドアロンコンシューマー（手動割り当てを使用）の場合、オフセットは最後のコミット期間と、この保持期間後に期限切れになります。

offsets.topic.compression.codec

型： int
デフォルト： 0
Importance: high
Dynamic update: read-only
オフセットトピックの圧縮コード c: 圧縮を使用して「atomic」コミットを実現できます。

offsets.topic.num.partitions

型： int
デフォルト： 50
有効な値 [1,...]
Importance: high
Dynamic update: read-only
オフセットコミットトピックのパーティション数（デプロイメント後に変更しないでください）

offsets.topic.replication.factor

タイプ： 短い
デフォルト： 3
有効な値 [1,...]
Importance: high
Dynamic update: read-only
オフセットトピックのレプリケーション係数（可用性を確保するためにより高い値を設定します）。クラスターのサイズがこのレプリケーション係数の要件を満たすまで、内部トピックの作成は失敗します。

offsets.topic.segment.bytes

型： int
デフォルト： 104857600(100 mebibytes)
有効な値 [1,...]
Importance: high
Dynamic update: read-only
ログコンパクションとキャッシュのロードを迅速化するために、オフセットトピックセグメントバイトは比較的小さくする必要があります。

port

型： int
デフォルト： 9092
Importance: high
Dynamic update: read-only
非推奨： **listeners** が設定されていない場合にのみ使用されます。代わりに **listeners** を使用します。ポートを使用して、接続をリッスンし、これを受け入れます。

queued.max.requests

型： int
デフォルト： 500
有効な値 [1,...]
Importance: high
Dynamic update: read-only
ネットワークスレッドをブロックする前に、データプレーンに許可されるキューに置かれたリクエストの数。

quota.consumer.default

タイプ： long
デフォルト： 9223372036854775807
有効な値 [1,...]
Importance: high
Dynamic update: read-only

非推奨： Zookeeper の <user、<client-id> または <user, client-id> に動的のデフォルトクォータが設定されていない場合にのみ使用します。clientId/consumer グループによるコンシューマーの識別名はすべて、この値を 1 秒ごとにフェッチするバイト数を取得するとスロットリングされます。

quota.producer.default

タイプ： long
デフォルト： 9223372036854775807
有効な値 [1,...]
Importance: high
Dynamic update: read-only
非推奨： Zookeeper の <user>、<client-id>、または <user, client-id> に動的のデフォルトクォータが設定されていない場合にのみ使用します。clientId によるプロデューサーの区別は、1 秒ごとにこの値を超えるバイトを生成すると、スロットルされます。

replica.fetch.min.bytes

型： int
デフォルト： 1
Importance: high
Dynamic update: read-only
各フェッチ応答に最低バイトが必要です。十分なバイトがない場合は、**replica.fetch.wait.max.ms**（ブローカー設定）まで待機します。

replica.fetch.wait.max.ms

型： int
デフォルト： 500
Importance: high
Dynamic update: read-only
フォロワーレプリカが発行する各フェッチリクエストの最大待機時間。この値は、スループットが低いトピックで ISR の頻繁に削減することを防ぐために、常に replica.lag.time.max.ms 未満にする必要があります。

replica.high.watermark.checkpoint.interval.ms

タイプ： long
デフォルト： 5000（5 秒）
Importance: high
Dynamic update: read-only
高基準がディスクに保存される頻度。

replica.lag.time.max.ms

型： long
デフォルト： 30000（30 秒）
Importance: high
Dynamic update: read-only
フォロワーがフェッチリクエストを送信していないか、またはこの時間にリーダーのログ終了オフセットに消費されていないと、リーダーはフォロワーからフォロワーを削除します。

replica.socket.receive.buffer.bytes

型： int
デフォルト： 65536(64 kibibytes)
Importance: high
Dynamic update: read-only
ネットワークリクエストのソケット受信バッファー。

replica.socket.timeout.ms

型： int
デフォルト： 30000（30 秒）
Importance: high
Dynamic update: read-only
ネットワーク要求のソケットタイムアウト。この値は、少なくとも replica.fetch.wait.max.ms である必要があります。

request.timeout.ms

型： int
デフォルト： 30000（30 秒）
Importance: high
Dynamic update: read-only
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。

socket.receive.buffer.bytes

型： int
デフォルト： 102400(100 kibibytes)
Importance: high
Dynamic update: read-only
ソケットサーバーソケットの SO_RCVBUF バッファー。値が -1 の場合、OS のデフォルトが使用されます。

socket.request.max.bytes

型： int
デフォルト： 104857600(100 mebibytes)
有効な値 [1,...]
Importance: high
Dynamic update: read-only
ソケットリクエストの最大バイト数。

socket.send.buffer.bytes

型： int
デフォルト： 102400(100 kibibytes)
Importance: high
Dynamic update: read-only
ソケットサーバーソケットの SO_SNDBUF バッファー。値が -1 の場合、OS のデフォルトが使用されます。

transaction.max.timeout.ms

型： int
デフォルト： 900000（15 分）
有効な値： [1,...]
Importance: high
Dynamic update: read-only
トランザクションに対して許可される最大タイムアウト。クライアントが要求したトランザクション時間がこの値を超えると、ブローカーは InitProducerIdRequest でエラーを返します。これにより、クライアントがタイムアウトが大きすぎます。これにより、トランザクションに含まれるトピックからコンシューマーが読み取れることがあります。

transaction.state.log.load.buffer.size

型： int
デフォルト： 5242880
有効な値 [1,...]
Importance: high
Dynamic update: read-only
プロデューサー ID およびトランザクションをキャッシュにロードするときにトランザクションログセグメントから読み取りするためのバッチサイズ（レコードが大きすぎるとオーバーライドされます）。

transaction.state.log.min.isr

型： int
デフォルト： 2
有効な値 [1,...]
Importance: high
Dynamic update: read-only
トランザクショントピックの min.insync.replicas 設定をオーバーライドします。

transaction.state.log.num.partitions

型： int
デフォルト： 50
有効な値 [1,...]
Importance: high
Dynamic update: read-only
トランザクショントピックのパーティション数（デプロイメント後に変更しないでください）

transaction.state.log.replication.factor

タイプ： 短い
デフォルト： 3
有効な値 [1,...]
Importance: high
Dynamic update: read-only
トランザクショントピックのレプリケーション係数（可用性を確保するためにより高い値を設定します）。クラスターのサイズがこのレプリケーション係数の要件を満たすまで、内部トピックの作成は失敗します。

transaction.state.log.segment.bytes

型： int
デフォルト： 104857600(100 mebibytes)
有効な値 [1,...]
Importance: high
Dynamic update: read-only
ログコンパクションとキャッシュのロードを迅速化するために、トランザクショントピックセグメントバイトは比較的小さくなければなりません。

transactional.id.expiration.ms

型： int
デフォルト： 604800000(7 days)
有効な値： [1,...]
Importance: high
Dynamic update: read-only
トランザクションコーディネーターが現在のトランザクションのトランザクションステータスの更

新を受信してからトランザクション ID の期限が切れることなくトランザクションコーディネーターを待つ時間（ミリ秒単位）。この設定は、プロデューサー ID の有効期限にも影響を与えます。指定のプロデューサー ID を持つ最後の書き込みが経過すると、プロデューサー ID が期限切れになります。トピックの保持設定が原因で、プロデューサー ID から最後の書き込みが削除されると、プロデューサー ID が期限切れになることがあります。

unclean.leader.election.enable

type: boolean
デフォルト： false
Importance: high
Dynamic update: cluster-wide
ISR が設定されていないレプリカを、最後の手段としてリーダーとして選択するようにし、データが失われる可能性があるかどうかを示します。

zookeeper.connection.timeout.ms

type: int
デフォルト： null
Importance: high
Dynamic update: read-only
クライアントが zookeeper への接続を確立するまでの最大時間。設定されていない場合は、zookeeper.session.timeout.ms の値が使用されます。

zookeeper.max.in.flight.requests

型： int
デフォルト： 10
有効な値 [1,...]
Importance: high
Dynamic update: read-only
クライアントがブロックされる前に、承認されていないリクエストの最大数を Zookeeper に送信します。

zookeeper.session.timeout.ms

型： int
デフォルト： 18000（18 秒）
Importance: high
Dynamic update: read-only
ZooKeeper セッションのタイムアウト。

zookeeper.set.acl

type: boolean
デフォルト： false
Importance: high
Dynamic update: read-only
セキュアな ACL を使用するようにクライアントを設定します。

broker.id.generation.enable

型： ブール値
デフォルト： true
Importance: medium
Dynamic update: read-only
サーバーでブローカー ID の生成を有効にします。これを有効にすると、reserved.broker.max.id に設定された値を確認する必要があります。

broker.rack

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ブローカーのラック。これは、フォールトトレランスのラック対応レプリケーション割り当てに使用されます。例： **RACK1**、**us-east-1d**

connections.max.idle.ms

タイプ： long
デフォルト： 600000（10 分）
Importance: medium
Dynamic update: read-only
アイドル接続タイムアウト： サーバーソケットプロセッサースレッドは、これ以上アイドル状態の接続を閉じます。

connections.max.reauth.ms

タイプ： long
デフォルト： 0
Importance: medium
Dynamic update: read-only
正数（デフォルトは 0 ではない）に明示的に設定されている場合、設定された値を超えていないセッションライフタイムは、認証時に v2.2.0 またはそれ以降のクライアントに通知されます。ブローカーは、セッション有効期間内で再認証されないこのような接続を切断します。これはその後再認証以外の目的に使用されます。設定名には、小文字でリスナー接頭辞と SASL メカニズム名を付けることができます。For example, listener.name.sasl_ssl.oauthbearer.connections.max.reauth.ms=3600000.

controlled.shutdown.enable

型： ブール値
デフォルト： true
Importance: medium
Dynamic update: read-only
サーバーの制御されたシャットダウンを有効にします。

controlled.shutdown.max.retries

型： int
デフォルト： 3
Importance: medium
Dynamic update: read-only
制御されたシャットダウンは、複数の理由で失敗する可能性があります。これにより、このような障害が発生する回数を決定します。

controlled.shutdown.retry.backoff.ms

タイプ： long
デフォルト： 5000（5 秒）
Importance: medium
Dynamic update: read-only
再試行ごとに、システムは以前の失敗の原因となった状態からのリカバリーに時間がかかります（コントローラーが失敗する、レプリカラグなど）。この設定は、再試行するまで待つ期間を決定します。

controller.socket.timeout.ms

型： int
デフォルト： 30000（30 秒）
Importance: medium
Dynamic update: read-only
controller-to-broker チャネルのソケットタイムアウト。

default.replication.factor

型： int
デフォルト： 1
Importance: medium
Dynamic update: read-only
自動的に作成されるトピックのデフォルトのレプリケーション係数。

delegation.token.expiry.time.ms

タイプ： long
デフォルト： 86400000（1 日）
有効な値 [1,...]
Importance: medium
Dynamic update: read-only
トークンを更新する必要があるまでのトークンの有効期間（ミリ秒単位）。デフォルト値は 1 日です。

delegation.token.master.key

type: password
デフォルト： null
Importance: medium
Dynamic update: read-only
委任トークンを生成して検証するためのマスター/シークレットキー。同じキーはすべてのブローカーで設定する必要があります。キーが設定されていない、または空の文字列に設定されている場合、ブローカーは委任トークンのサポートを無効にします。

delegation.token.max.lifetime.ms

タイプ： long
デフォルト： 604800000(7 days)
有効な値： [1,...]
Importance: medium
Dynamic update: read-only
トークンのライフタイムは最大で、更新できません。デフォルト値は 7 日です。

delete.records.purgatory.purge.interval.requests

型： int
デフォルト： 1
Importance: medium
Dynamic update: read-only
削除レコードのパージ間隔（要求の数）。

fetch.max.bytes

型： int
デフォルト： 57671680(55 mebibytes)
有効な値 [1024, ...]

importance: medium
Dynamic update: read-only
フェッチリクエストに対して返す最大バイト数。1024 以上でなければなりません。

fetch.purgatory.purge.interval.requests

型： int
デフォルト： 1000
Importance: medium
Dynamic update: read-only
フェッチ要求のパージ間隔（リクエスト数）。

group.initial.rebalance.delay.ms

型： int
デフォルト： 3000（3 秒）
Importance: medium
Dynamic update: read-only
グループコーディネーターが新しいグループを参加するまで待機する時間。この期間を超えると最初のリバランスが実行されます。遅延が長くなると、リバランスが少なくなる可能性がありますが、処理が開始されるまでの時間が増加します。

group.max.session.timeout.ms

型： int
デフォルト： 1800000（30 分）
Importance: medium
Dynamic update: read-only
登録済みのコンシューマーに対して許可される最大セッションタイムアウト。タイムアウトが長くなると、障害の検出に伴い、コンシューマーがハートビートの間にメッセージを処理する時間が長くなります。

group.max.size

型： int
デフォルト： 2147483647
有効値： [1,...]
Importance: medium
Dynamic update: read-only
単一コンシューマーグループが対応できるコンシューマーの最大数。

group.min.session.timeout.ms

型： int
デフォルト： 6000（6 秒）
Importance: medium
Dynamic update: read-only
登録済みのコンシューマーに対して許可されるセッションの最小タイムアウト。タイムアウトを短くすると、より頻繁にコンシューマーのハートビートが発生するコストが短縮され、ブローカーリソースが過剰にかかる可能性があります。

inter.broker.listener.name

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only

ブローカー間の通信に使用されるリスナーの名前。これを設定しないと、リスナー名は security.inter.broker.protocol によって定義されます。これと security.inter.broker.protocol プロパティーを同時に設定するエラーです。

inter.broker.protocol.version

型： string
デフォルト： 2.7-IV2
有効な値： [0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV0 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.1-IV0、3IV0-IV0-IV0 2.7-IV0, 2.7-IV1, 2.7-IV2]
Importance: medium
Dynamic update: read-only
ブローカー間プロトコルで使用されるバージョンを指定します。これは通常、すべてのブローカーが新規バージョンにアップグレードされた後に結合されます。一部の有効な値は、0.8.0、0.8.1、0.8.1.1、0.8.2、0.8.2.0、0.8.2.1、0.9.0.0、0.9.0.1 Check ApiVersion（全一覧の場合）です。

log.cleaner.backoff.ms

型： long
デフォルト： 15000（15 秒）
有効な値： [0,...]
Importance: medium
Dynamic update: cluster-wide
消去するログがない場合のスリープ状態の時間。

log.cleaner.dedupe.buffer.size

タイプ： long
デフォルト： 134217728
Importance: medium
Dynamic update: cluster-wide
すべてのクリーナースレッドでのログの重複排除に使用されるメモリーの合計。

log.cleaner.delete.retention.ms

タイプ： long
デフォルト： 86400000（1 日）
Importance: medium
Dynamic update: clusterwide
レコードの削除期間

log.cleaner.enable

型： ブール値
デフォルト： true
Importance: medium
Dynamic update: read-only
ログクリーナープロセスがサーバー上で実行されるようにします。cleanup.policy=compact でトピックを使用する場合は、内部オフセットトピックを含むようにする必要があります。無効化されると、これらのトピックは圧縮されず、常にサイズが増加します。

log.cleaner.io.buffer.load.factor

type:
デフォルト： 0.9
Importance: medium
Dynamic update: clusterwide

ログクリープバッファー負荷係数破棄されたバッファーが満杯になるパーセンテージ。値が大きいほど、より多くのログを一度に消去することが可能になりますが、ハッシュの競合が増えます。

log.cleaner.io.buffer.size

型： int
デフォルト： 524288
有効な値： [0,...]
Importance: medium
Dynamic update: cluster-wide
すべてのクリーナースレッド全体でログクリーナー I/O バッファーに使用されるメモリーの合計。

log.cleaner.io.max.bytes.per.second

type: double
Default: 1.7976931348623157E308
Importance: medium
Dynamic update: clusterwide
ログクリープをスロットリングされ、読み取り/書き込み I/o の合計が平均でこの値よりも小さいようになります。

log.cleaner.max.compaction.lag.ms

タイプ： long
デフォルト： 9223372036854775807
Importance: medium
Dynamic update: clusterwide
メッセージはログ内で圧縮できない最大時間。圧縮されているログにのみ適用可能です。

log.cleaner.min.cleanable.ratio

タイプ：
デフォルト： 0.5
Importance: medium
Dynamic update: clusterwide
クリーニングの対象となるログ合計ログへのダーティーログの最小比率。
log.cleaner.max.compaction.lag.ms または log.cleaner.compaction.lag.ms 設定も指定された場合、ログコンパクションも、ダーティー率のしきい値が満たされた時(i)のように、ログがダーティー率のしきい値に一致し、ログにはダーティー率(uncompacted)が発生します。 または、ログに多数の log.cleaner.max.compaction.lag.ms の期間にダーティー（コンパイルされていない）レコードがある場合。

log.cleaner.min.compaction.lag.ms

タイプ： long
デフォルト： 0
Importance: medium
Dynamic update: cluster-wide
メッセージがログに記録されない最小の時間。圧縮されているログにのみ適用可能です。

log.cleaner.threads

型： int
デフォルト： 1
有効な値： [0,...]
Importance: medium
Dynamic update: clusterwide
ログ消去に使用するバックグラウンドスレッドの数。

log.cleanup.policy

タイプ： list
デフォルト：
有効な値を削除します： [compact, delete]
Importance: medium
Dynamic update: cluster-wide
保持ウィンドウ以外のセグメントのデフォルトのクリーンアップポリシー。有効なポリシーのコンマ区切りリスト。有効なポリシー： "delete" および "compact"

log.index.interval.bytes

タイプ： int
デフォルト： 4096(4 kibibytes)
有効な値 [0,...]
Importance: medium
Dynamic update: cluster-wide
オフセットインデックスにエントリーを追加する間隔。

log.index.size.max.bytes

型： int
デフォルト： 10485760(10 mebibytes)
有効な値 [4,...]
Importance: medium
Dynamic update: cluster-wide
オフセットインデックスの最大サイズ（バイト単位）。

log.message.format.version

型： string
デフォルト： 2.7-IV2
有効な値： [0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV0 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.1-IV0、3IV0-IV0-IV0 2.7-IV0, 2.7-IV1, 2.7-IV2]
Importance: medium
Dynamic update: read-only
ブローカーがログにメッセージを追加するのに使用するメッセージ形式のバージョンを指定します。値は有効な ApiVersion である必要があります。例としては、0.8.2、0.9.0.0、0.10.0 などがあります。詳細は、ApiVersion を参照してください。特定のメッセージ形式のバージョンを設定することで、ディスク上の既存のメッセージがすべて、指定されたバージョンよりも小さくなることが認定されています。この値を誤って設定すると、コンシューマーが認識しない形式のメッセージを受信してしまうため、古いバージョンのコンシューマーが破損することになります。

log.message.timestamp.difference.max.ms

タイプ： long
デフォルト： 9223372036854775807
Importance: medium
Dynamic update: clusterwide
ブローカーがメッセージとメッセージを受け取ると、タイムスタンプとメッセージに指定されたタイムスタンプの間の最大差。log.message.timestamp.type=CreateTime の場合、タイムスタンプの差異がこのしきい値を超えるとメッセージは拒否されます。この設定は、log.message.timestamp.type=LogApp の場合に無視されます。タイムスタンプの差異は、不要なログのローリングを回避するため、log.retention.ms よりも高い値である必要があります。

log.message.timestamp.type

Type: string
デフォルト： CreateTime
有効な値 [CreateTime, LogApp the]
Importance: medium
Dynamic update: cluster-wide
メッセージのタイムスタンプが create time であるか、またはログの追加時間であるかを定義します。値は **CreateTime** または **LogAppendTime** のいずれかでなければなりません。

log.preallocate

type: boolean
デフォルト： false
Importance: medium
Dynamic update: clusterwide
新しいセグメントの作成時にファイルを事前割り当てですか？Windows で Kafka を使用している場合は、true に設定する必要があります。

log.retention.check.interval.ms

型： long
デフォルト： 300000（5 分）
有効な値： [1,...]
Importance: medium
Dynamic update: read-only
ログクリーナーがいずれかのログが削除の対象となるかをチェックする頻度（ミリ秒単位）。

max.connection.creation.rate

型： int
デフォルト： 2147483647
有効値： [0,...]
Importance: medium
Dynamic update: clusterwide
ブローカーで許可される最大接続の作成レート。リスナーレベルの制限は、設定名をリスナー接頭辞のプレフィックスで設定することもできます。たとえば、**listener.name.internal.max.connection.creation.rate**.Broker 全体の接続流量制御は、アプリケーションの要件に基づいてリスナー容量に基づいて設定する必要があります。ブローカー間リスナーを除き、リスナーまたはブローカーの制限のいずれかに達すると、新しい接続がスロットリングされます。ブローカー間リスナー上の接続は、リスナーレベルの流量制御に達した場合にのみスロットリングされます。

max.connections

型： int
デフォルト： 2147483647
有効値： [0,...]
Importance: medium
Dynamic update: clusterwide
いつでもブローカーで許可される接続の最大数。この制限は、max.connections.per.ip を使用して設定された IP ごとの制限の他に適用されます。リスナーレベルの制限は、設定名をリスナー接頭辞のプレフィックスで設定することもできます（例： **listener.name.internal.max.connections** ）。ブローカー全体の制限はブローカーの容量に基づいて設定する必要がありますが、リスナーの制限はアプリケーション要件に基づいて設定する必要があります。リスナーまたはブローカーの制限のいずれかに達すると、新しい接続はブロックされます。ブローカー間のリスナーの接続は、ブローカー全体で制限に達した場合でも許可されます。この場合、別のリスナーで最近使用された接続が切断されます。

max.connections.per.ip

型： int
デフォルト： 2147483647
有効値： [0,...]
Importance: medium
Dynamic update: clusterwide
各 IP アドレスから許可される接続の最大数。max.connections.per.ip.overrides プロパティーを使用して設定されたオーバーライドがある場合は、0 に設定します。制限に達すると、IP アドレスからの新規接続は破棄されます。

max.connections.per.ip.overrides

type: string
デフォルト： ""
Importance: medium
Dynamic update: clusterwide
デフォルトの最大接続数まで、ip ごとまたはホスト名のオーバーライドのコンマ区切りリスト。値の例は "hostName:100,127.0.0.1:200" です。

max.incremental.fetch.session.cache.slots

型： int
デフォルト： 1000
有効な値： [0,...]
Importance: medium
Dynamic update: read-only
メンテナンスする増分フェッチセッションの最大数。

num.partitions

型： int
デフォルト： 1
有効な値 [1,...]
Importance: medium
Dynamic update: read-only
トピックごとのデフォルトのログパーティション数。

password.encoder.old.secret

type: password
デフォルト： null
Importance: medium
Dynamic update: read-only
エンコードに動的に設定されたパスワードに使用される古いシークレット。これは、シークレットが更新されている場合にのみ必要です。これが指定されている場合、動的にエンコードされたすべてのパスワードはこの古いシークレットを使用してデコードされ、ブローカーの起動時に password.encoder.secret を使用して再エンコードされます。

password.encoder.secret

type: password
デフォルト： null
Importance: medium
Dynamic update: read-only
このブローカーの動的に設定されたパスワードのエンコードに使用されるシークレット。

principal.builder.class

型： クラス
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
承認中に使用される KafkaPrincipalBuilder オブジェクトを構築するために使用される KafkaPrincipalBuilder インターフェースを実装するクラスの完全修飾名。この設定は、SSL を介したクライアント認証に使用される非推奨の PrincipalBuilder インターフェースもサポートします。プリンシパルビルダーが定義されていない場合は、デフォルトの動作は使用中のセキュリティープロトコルによって異なります。SSL 認証では、プリンシパルは指定された場合にクライアント証明書から識別名に適用される **ssl.principal.mapping.rules** で定義されるルールを使用して派生されます。クライアント認証が必要ない場合は、プリンシパル名は ANONYMOUS になります。SASL 認証では、GSSAPI が使用されている場合に **sasl.kerberos.principal.to.local.rules** で定義したルールを使用してプリンシパルが派生し、その他のメカニズムの SASL 認証 ID を使用します。
PLAINTEXT の場合、プリンシパルは ANONYMOUS になります。

producer.purgatory.purge.interval.requests

型： int
デフォルト： 1000
Importance: medium
Dynamic update: read-only
プロデューサー要求のパージ間隔（リクエスト数）

queued.max.request.bytes

タイプ： long
デフォルト： -1
Importance: medium
Dynamic update: read-only
要求が読み取られる前に許可されるキューに置かれたバイト数。

replica.fetch.backoff.ms

型： int
デフォルト： 1000（1 秒）
有効な値： [0,...]
Importance: medium
Dynamic update: read-only
パーティションの取得エラーが発生したときのスリープ時間。

replica.fetch.max.bytes

型： int
デフォルト： 1048576(1 mebibyte)
有効な値： [0,...]
Importance: medium
Dynamic update: read-only
各パーティションの取得を試行するメッセージの数。これは絶対最大値ではありません。フェッチの最初の空でないパーティションの最初のレコードバッチがこの値よりも大きい場合、レコードバッチは返され、進捗ができるようにします。ブローカーで許可される最大レコードバッチサイズは、**message.max.bytes** （ブローカー設定）または **max.message.bytes** （トピック設定）で定義されます。

replica.fetch.response.max.bytes

型： int
デフォルト： 10485760(10 mebibytes)
有効な値 [0,...]

Importance: medium
Dynamic update: read-only
フェッチ応答全体で期待される最大バイト。レコードはバッチでフェッチされ、フェッチの最初の空でないパーティションで最初のレコードバッチがこの値よりも大きい場合、レコードバッチは返され、進捗ができるようにします。したがって、これは絶対最大値ではありません。ブローカーで許可される最大レコードバッチサイズは、**message.max.bytes**（ブローカー設定）または **max.message.bytes**（トピック設定）で定義されます。

replica.selector.class

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ReplicaSelector を実装する完全修飾クラス名。これは、優先される読み取りレプリカを見つけるためにブローカーによって使用されます。デフォルトでは、リーダーを返す実装を使用します。

reserved.broker.max.id

型： int
デフォルト： 1000
有効な値： [0,...]
Importance: medium
Dynamic update: read-only
broker.id に使用できる最大数値。

sasl.client.callback.handler.class

Type: class
デフォルト： null
Importance: medium
Dynamic update: read-only
AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.enabled.mechanisms

type: list
デフォルト： GSSAPI
Importance: medium
Dynamic update: ブローカーごとの動的更新。
Kafka サーバーで有効になっている SASL メカニズムのリスト。リストには、セキュリティープロバイダーを利用できるメカニズムが含まれる場合があります。デフォルトでは GSSAPI のみが有効になります。

sasl.jaas.config

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
JAAS 設定ファイルで使用される形式で SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式が ここ に記載されています。値の形式は以下のとおりです。「loginModuleClass controlFlag (optionName=optionValue)*;」.ブローカーの場合、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。例：
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.kinit.cmd

type: string
デフォルト： /usr/bin/kinit
Importance: medium
Dynamic update: medium-broker
Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ： long
デフォルト： 60000
Importance: medium
Dynamic update: ブローカーごとの動的更新。
更新試行間のログインスレッドスリープ時間。

sasl.kerberos.principal.to.local.rules

type: list
デフォルト： DEFAULT
Importance: medium
Dynamic update:broker
プリンシパル名から短縮名（通常はオペレーティングシステムのユーザー名）へのマッピングルールの一覧。ルールは順番に評価され、プリンシパル名に一致する最初のルールが短縮名にマップするために使用されます。リスト内の後続のルールは無視されます。デフォルトでは、{username}/{hostname} keyREALMREALM} 形式のプリンシパル名は {username} にマッピングされます。形式に関する詳細は、「 セキュリティー承認および acls 」を参照してください。KafkaPrincipalBuilder の拡張機能が **principal.builder.class** 設定によって提供される場合は、この設定は無視されます。

sasl.kerberos.service.name

type: string
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定で定義できます。

sasl.kerberos.ticket.renew.jitter

type:
デフォルト： 0.05
Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
更新時間にランダムなジッターが追加されたパーセンテージ。

sasl.kerberos.ticket.renew.window.factor

type:
デフォルト： 0.8
Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
ログインスレッドは、指定のウィンドウ係数からチケットの有効期限に達するまでスリープされます。この期間が経過するとチケットの更新が試行されます。

sasl.login.callback.handler.class

Type: class
デフォルト： null
Importance: medium
Dynamic update: read-only
AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラーの設定には、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。たとえば、listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler です。

sasl.login.class

Type: class
デフォルト： null
Importance: medium
Dynamic update: read-only
ログインインターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナー接頭辞と SASL メカニズム名を付ける必要があります。For example, listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.login.refresh.buffer.seconds

タイプ： 短い
デフォルト： 300
Importance: medium
Dynamic update: ブローカーごとの動的更新。
認証情報を更新するときの認証情報の有効期限（秒単位）。そうでなければ、更新がバッファー秒数よりも期限切れになり、可能な限り多くのバッファー時間を維持するよう更新が行われます。法人の値は 0 から 3600（1 時間）で設定されています。値が指定されていない場合には、デフォルト値の 300（5 分）が使用されます。この値と sasl.login.refresh.min.period.seconds は、合計が認証情報の残りの有効期間を超えた場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

タイプ： 短い
デフォルト： 60
Importance: medium
Dynamic update: ブローカーごとの動的更新。
ログイン更新スレッドで認証情報を更新するまで待つ必要最小限の時間（秒単位）。法人の値は 0 から 900（15 分）で設定されています。値が指定されていない場合には、デフォルト値の 60（1 分）が使用されます。この値と sasl.login.refresh.buffer.seconds は、その合計が認証情報の残りの有効期間を超える場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

type:
デフォルト： 0.8
Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
ログイン更新スレッドは、認証情報の有効期間に比べ、指定されたウィンドウ係数に達するまでスリープ状態になり、クレデンシャルの更新を試みます。法人の値は 0.5(50%)と 1.0(100%)の範囲になります。値が指定されていない場合には、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

type:
デフォルト： 0.05

Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
ログイン更新スレッドのスリープ時間に追加される認証情報の有効期間に対して、ランダムなジッターの最大量。法人の値は 0 から 0.25(25%)で囲まれています。値が指定されていない場合には、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.mechanism.inter.broker.protocol

type: string
デフォルト： GSSAPI
Importance: medium
Dynamic update: ブローカーごとの動的更新。
ブローカー間通信に使用される SASL メカニズム。デフォルトは GSSAPI です。

sasl.server.callback.handler.class

Type: class
デフォルト： null
Importance: medium
Dynamic update: read-only
AuthenticateCallbackHandler インターフェースを実装する SASL サーバーコールバックハンドラークラスの完全修飾名。サーバーコールバックハンドラーの前に、リスナー接頭辞と SASL メカニズム名を付ける必要があります。例：
listener.name.sasl_ssl.plain.sasl.server.callback.handler.class=com.example.CustomPlainCallbackHandler

security.inter.broker.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
Dynamic update: read-only
ブローカー間の通信に使用されるセキュリティープロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。これと Inter.broker.listener.name プロパティーを同時に設定するエラーです。

socket.connection.setup.timeout.max.ms

タイプ： long
デフォルト： 127000（127 秒）
Importance: medium
Dynamic update: read-only
ソケット接続が確立されるまでクライアントが待機する最大時間。接続セットアップのタイムアウトは、この最大値までの連続する接続障害ごとに指数関数的に増加します。接続ストームを回避するために、0.2 のランダム化係数はタイムアウトに適用されます。これにより、計算値の 20% 未満の 20% の範囲がランダムな範囲になります。

socket.connection.setup.timeout.ms

タイプ： long
デフォルト： 10000（10 秒）
Importance: medium
Dynamic update: read-only
クライアントがソケット接続を確立するのを待つ時間。タイムアウトが経過する前にコネクションがビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.cipher.suites

type: list

デフォルト： ""
Importance: medium
Dynamic update: per-broker
暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC、鍵交換アルゴリズムの名前付きの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.client.auth

型： 文字列
デフォルト： none
有効な値： [必須、requested, none]
Importance: medium
Dynamic update: medium-broker
Kafka ブローカーがクライアント認証を要求するように設定します。以下は一般的な設定です。

- **ssl.client.auth=required** 必要なクライアント認証に設定されていると、必要なクライアント認証が必要です。
- **ssl.client.auth=requested** そのため、このオプションが設定されているとクライアント自体の認証情報を提供しないことができない場合があるため、クライアントの認証は任意です。
- **ssl.client.auth=none** これは、クライアント認証が不要であることを意味します。

ssl.enabled.protocols

type: list
デフォルト： TLSv1.2,TLSv1.3
Importance: medium
Dynamic update: medium-broker
SSL 接続に対して有効なプロトコル一覧。Java 11 以降、TLSv1.2' の場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーは、TLSv1.2 へのサポートとフォールバックの両方をサポートする場合は、TLSv1.3 が優先されます（両方が TLSv1.2 をサポートしていることを前提とします）。ほとんどの場合、このデフォルト設定は問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.key.password

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
キーストアファイルまたは 'ssl.keystore.key' に指定された PEM キーのパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keymanager.algorithm

type: string
デフォルト： SunX509
Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.keystore.certificate.chain

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
'ssl.keystore.type' により指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書の一覧で PEM 形式のみをサポートします。

ssl.keystore.key

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
'ssl.keystore.type' により指定された形式の秘密鍵。デフォルトの SSL エンジンファクトリーは、PKCS#8 鍵を持つ PEM 形式のみをサポートします。鍵を暗号化する場合は、'ssl.key.password' を使用して鍵のパスワードを指定する必要があります。

ssl.keystore.location

type: string
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
キーストアファイルの場所。これはクライアント用にオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
キーストアファイルのストアパスワード。これはクライアントでオプションであり、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアパスワードは PEM 形式ではサポートされません。

ssl.keystore.type

type: string
デフォルト： JKS
Importance: medium
Dynamic update: ブローカーごとの動的更新。
キーストアファイルのファイル形式。これはクライアントでオプションです。

ssl.protocol

type: string
デフォルト： TLSv1.3
Importance: **medium**
Dynamic update: ブローカーごとの動的更新。
SSLContext の生成に使用される SSL プロトコル。Java 11 以降、'TLSv1.2' の場合、デフォルトは 'TLSv1.3' です。この値は、ほとんどのユースケースで問題ありません。最新の JVM の許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされていますが、既知のセキュリティー脆弱性により使用は推奨されません。この設定と 'ssl.enabled.protocols' のデフォルト値で、サーバーが 'TLSv1.3' に対応していない場合は、クライア

ントは 'TLSv1.2' にダウングレードします。この設定を 'TLSv1.2' に設定すると、ssl.enabled.protocols の値のいずれかであった場合でも、クライアントは 'TLSv1.3' を使用しません。また、サーバーは 'TLSv1.3' のみをサポートします。

ssl.provider

type: string
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
SSL 接続に使用されるセキュリティープロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティープロバイダーです。

ssl.trustmanager.algorithm

type: string
デフォルト： PKIX
Importance: medium
Dynamic update: ブローカーごとの動的更新。
SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

ssl.truststore.certificates

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
'ssl.truststore.type' によって指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書で PEM 形式のみをサポートします。

ssl.truststore.location

type: string
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
トラストストアファイルの場所。

ssl.truststore.password

type: password
デフォルト： null
Importance: medium
Dynamic update: ブローカーごとの動的更新。
トラストストアファイルのパスワード。パスワードが設定されていないと、設定したトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアパスワードは PEM 形式ではサポートされません。

ssl.truststore.type

type: string
デフォルト： JKS
Importance: medium
Dynamic update: ブローカーごとの動的更新。
トラストストアファイルのファイル形式。

zookeeper.clientCnxnSocket

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
通常、ZooKeeper への TLS 接続を使用する場合は **org.apache.zookeeper.ClientCnxnSocketNetty** に設定されます。同じ名前の **zookeeper.clientCnxnSocket** システムプロパティーで設定した明示的な値をオーバーライドします。

zookeeper.ssl.client.enable

型： ブール値
デフォルト： false
Importance: medium
Dynamic update: read-only
ZooKeeper への接続時に TLS を使用するようにクライアントを設定します。明示的な値は、**zookeeper.client.secure** システムプロパティーで設定した値を上書きします（異なる名前に注意してください）。設定されていない場合はデフォルトで false に設定されます。true の場合、**zookeeper.clientCnxnSocket** を設定する必要があります（通常は **org.apache.zookeeper.ClientCnxnSocketNetty**）。設定する他の値に **zookeeper.ssl.cipher.suites**、**zookeeper.ssl.crl.enable**、**zookeeper.ssl.enabled.protocols** が含まれる場合があります。 **zookeeper.ssl.endpoint.identification.algorithm**, **zookeeper.ssl.keystore.location**, **zookeeper.ssl.keystore.password**, **zookeeper.ssl.keystore.type**, **zookeeper.ssl.ocsp.enable**, **zookeeper.ssl.protocol**, **zookeeper.ssl.truststore.location**, **zookeeper.ssl.truststore.password**, **zookeeper.ssl.truststore.type**

zookeeper.ssl.keystore.location

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続でクライアント側の証明書を使用するときのキーストアの場所。**zookeeper.ssl.keyStore.location** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。

zookeeper.ssl.keystore.password

type: password
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続でクライアント側の証明書を使用する場合のキーストアパスワード。**zookeeper.ssl.keyStore.password** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。ZooKeeper はキーストアパスワードとは異なるキーパスワードをサポートしないため、キーストアのキーパスワードをキーストアパスワードと同じになるように設定してください。設定しないと、ZooKeeper への接続は失敗します。

zookeeper.ssl.keystore.type

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続でクライアント側の証明書を使用する場合のキーストアタイプ。**zookeeper.ssl.keyStore.type** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。**null** のデフォルト値は、キーストアのファイル名拡張子に基づいて

型が自動検出されることを意味します。

zookeeper.ssl.truststore.location

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続を使用する場合のトラストストアの場所。**zookeeper.ssl.trustStore.location** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。

zookeeper.ssl.truststore.password

type: password
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続を使用する場合のトラストストアパスワード。**zookeeper.ssl.trustStore.password** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。

zookeeper.ssl.truststore.type

type: string
デフォルト： null
Importance: medium
Dynamic update: read-only
ZooKeeper への TLS 接続を使用する場合のトラストストアタイプ。**zookeeper.ssl.trustStore.type** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。**null** のデフォルト値は、トラストストアのファイル名の拡張子に基づいてタイプが自動検出されることを意味します。

alter.config.policy.class.name

タイプ： クラス
デフォルト： null
インポート機能： low
Dynamic update: read-only
検証に使用する必要がある変更設定ポリシークラスクラスは **org.apache.kafka.server.policy.AlterConfigPolicy** インターフェースを実装する必要があります。

alter.log.dirs.replication.quota.window.num

型： int
デフォルト： 11
有効な値 [1,...]
Importance: low
Dynamic update: read-only
ログディレクトリーのレプリケーションクォータを変更するためのメモリーに保持するサンプル数。

alter.log.dirs.replication.quota.window.size.seconds

型： int
デフォルト： 1
有効な値 [1,...]

Importance: low
Dynamic update: read-only
ログディレクトリーのレプリケーションクォータを変更するための各サンプルの期間。

authorizer.class.name

type: string
デフォルト： ""
Importance: low
Dynamic update: read-only
承認用にブローカーによって使用される sorg.apache.kafka.server.authorizer.Authorizer インターフェースを実装するクラスの完全修飾名。この設定は、以前に承認に使用された非推奨の kafka.security.auth.Authorizer トレイトを実装するオーソライザーもサポートします。

client.quota.callback.class

タイプ： クラス
デフォルト： null
インポート機能： low
Dynamic update: read-only
クライアント要求に適用されるクォータ制限を決定するために使用される ClientQuotaCallback インターフェースを実装するクラスの完全修飾名。デフォルトでは、ZooKeeper に保存される <user、client-id>、<user> または <client-id> クォータが適用されます。指定されたリクエストでは、セッションのユーザープリンシパルと要求の client-id に一致する最も具体的なクォータが適用されます。

connection.failed.authentication.delay.ms

型： int
デフォルト： 100
有効な値 [0,...]
Importance: low
Dynamic update: read-only
接続を閉じるまでの遅延： これは、認証失敗時に接続を閉じる時間（ミリ秒）です。これは、接続のタイムアウトを防ぐために connection.max.idle.ms 未満に設定する必要があります。

controller.quota.window.num

型： int
デフォルト： 11
有効な値 [1,...]
Importance: low
Dynamic update: read-only
コントローラーの変更クォータのメモリーに保持するサンプル数。

controller.quota.window.size.seconds

型： int
デフォルト： 1
有効な値 [1,...]
Importance: low
Dynamic update: read-only
コントローラー間のクォータに対する各サンプルの時間枠

create.topic.policy.class.name

タイプ： クラス
デフォルト： null

インポート機能： low
Dynamic update: read-only
検証に使用するトピックポリシークラスを作成します。クラスは **org.apache.kafka.server.policy.CreateTopicPolicy** インターフェースを実装する必要があります。

delegation.token.expiry.check.interval.ms

型： long
デフォルト： 3600000（1時間）
有効な値： [1,...]
Importance: low
Dynamic update: read-only
期限切れの委譲トークンを削除する間隔。

kafka.metrics.polling.interval.secs

型： int
デフォルト： 10
有効な値 [1,...]
Importance: low
Dynamic update: read-only
kafka.metrics.reporters 実装で使用できるメトリックポーリング間隔（秒単位）。

kafka.metrics.reporters

タイプ： list
デフォルト： ""
Importance: low
Dynamic update: read-only
Yammer メトリクスカスタムレポーターとして使用するクラスの一覧。レポーターは **kafka.metrics.KafkaMetricsReporter** トレイトを実装する必要があります。クライアントがカスタムレポーターで JMX 操作を公開する場合、カスタムレポーターは **kafka.metrics.KafkaMetricsReporterMBean** トレイトを拡張する MBean トレイトを追加して、登録された MBean が標準の MBean 規則に準拠するようにする必要があります。

listener.security.protocol.map

type: string
Default:
PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
Importance: low
Dynamic update: per-broker
リスナー名とセキュリティープロトコル間のマッピング。これは、同じセキュリティープロトコルが複数のポートまたは IP を使用できるようにするには定義する必要があります。たとえば、両方に SSL が必要な場合であっても、内部および外部トラフィックを分離することができます。実際には、ユーザーは INTERNAL という名前のリスナーおよび EXTERNAL で、このプロパティーを **INTERNAL:SSL,EXTERNAL:SSL** として定義することができます。以下に示すように、キーと値はコロンで区切られ、マップエントリーはコンマで区切ります。各リスナー名はマップ内で1度だけ表示されるはずです。通常はリスナー名（小文字）を設定名に追加することで、リスナーごとに異なるセキュリティー（SSL および SASL）を設定できます。たとえば、INTERNAL リスナーに異なるキーストアを設定するには、**listener.name.internal.ssl.keystore.location** という名前の設定を設定します。リスナー名の設定が設定されていない場合、設定は汎用設定（例：**ssl.keystore.location**）にフォールバックします。

log.message.downconversion.enable

type: boolean
デフォルト： true

Importance: low
Dynamic update: cluster-wide
この設定は、リクエストを消費するために、ダウンコンバートバージョンのメッセージ形式が有効になっているかどうかを制御します。**false** に設定すると、古いメッセージ形式を必要とするコンシューマーに対してブローカーはダウンコンバートを実行しません。このような古いクライアントからの要求を消費するため、ブローカーは **UNSUPPORTED_VERSION** エラーで応答します。この設定は、レプリケーションのフォロワーに必要となる可能性のあるメッセージ形式の変換には適用されません。

metric.reporters

type: list
デフォルト： ""
Importance: low
Dynamic update: clusterwide
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
Dynamic update: read-only
メトリックの計算に維持されるサンプル数。

metrics.recording.level

type: string
デフォルト： INFO
Importance: low
Dynamic update: read-only
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [1,...]
Importance: low
Dynamic update: read-only
メトリクスサンプルが計算される期間。

password.encoder.cipher.algorithm

タイプ： 文字列
デフォルト： AES/CBC/PKCS5Padding
Importance: low
Dynamic update: read-only
動的に設定されたパスワードのエンコードに使用される暗号アルゴリズム。

password.encoder.iterations

型： int
デフォルト： 4096

有効な値 [1024,...]
Importance: low
Dynamic update: read-only
動的に設定されたパスワードに使用する反復数。

password.encoder.key.length

型： int
デフォルト： 128
有効な値： [8,...]
Importance: low
Dynamic update: read-only
動的に設定されたパスワードのエンコードに使用されるキー長。

password.encoder.keyfactory.algorithm

type: string
デフォルト： null
Importance: low
Dynamic update: read-only
動的に設定されたパスワードのエンコードに使用される SecretKeyFactory アルゴリズム。デフォルトは PBKDF2WithHmacSHA512（利用可能な場合で PBKDF2WithHmacSHA1 の場合）です。

quota.window.num

型： int
デフォルト： 11
有効な値 [1,...]
Importance: low
Dynamic update: read-only
クライアントクォータのメモリーに保持するサンプル数。

quota.window.size.seconds

型： int
デフォルト： 1
有効な値 [1,...]
Importance: low
Dynamic update: read-only
クライアントクォータの各サンプルの期間。

replication.quota.window.num

型： int
デフォルト： 11
有効な値 [1,...]
Importance: low
Dynamic update: read-only
レプリケーションクォータのメモリーに保持するサンプル数。

replication.quota.window.size.seconds

型： int
デフォルト： 1
有効な値 [1,...]
Importance: low
Dynamic update: read-only
レプリケーションクォータ用の各サンプルの時間スパン。

security.providers

type: string
デフォルト： null
Importance: low
Dynamic update: read-only
セキュリティーアルゴリズムを実装するプロバイダーを返す、設定可能な作成者クラスの一覧。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.endpoint.identification.algorithm

type: string
デフォルト： https
Importance: low
Dynamic update: per-broker
サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class
デフォルト： null
Importance: low
Dynamic update: per-broker
SSLEngine オブジェクトを提供するために、
org.apache.kafka.common.security.auth.SslEngineFactory タイプのクラス。Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory.

ssl.principal.mapping.rules

型： 文字列
デフォルト： DEFAULT
Importance: low
Dynamic update: read-only
クライアント証明書から短縮名へ識別名からマッピングするルールの一覧。ルールは順番に評価され、プリンシパル名に一致する最初のルールが短縮名にマップするために使用されます。リスト内の後続のルールは無視されます。デフォルトでは、X.500 証明書の識別名はプリンシパルになります。形式に関する詳細は、「 セキュリティー承認および acls 」を参照してください。
KafkaPrincipalBuilder の拡張機能が **principal.builder.class** 設定によって提供される場合は、この設定は無視されます。

ssl.secure.random.implementation

type: string
デフォルト： null
Importance: low
Dynamic update: per-broker
SSL 暗号操作に使用する SecureRandom PRNG 実装。

transaction.abort.timed.out.transaction.cleanup.interval.ms

型： int
デフォルト： 10000（10 秒）
有効な値 [1,...]
Importance: low
Dynamic update: read-only
タイムアウトしたトランザクションをロールバックする間隔。

transaction.remove.expired.transaction.cleanup.interval.ms

型： int
デフォルト： 3600000（1時間）
有効な値： [1,...]
Importance: low
Dynamic update: read-only
transactional.id.expiration.ms を渡すため期限切れのトランザクションを削除する間隔。

zookeeper.ssl.cipher.suites

type: list
デフォルト： null
Importance: low
Dynamic update: read-only
ZooKeeper TLS ネゴシエーション(csv)で使用される有効な暗号スイートを指定します。**zookeeper.ssl.ciphersuites** システムプロパティーで設定した明示的な値を上書きします（"ciphersuites の1つの単語）。**null** のデフォルト値は、有効な暗号スイートのリストが、使用されている Java ランタイムによって決定されます。

zookeeper.ssl.crl.enable

型： ブール値
デフォルト： false
インポート機能： low
Dynamic update: read-only
ZooKeeper TLS プロトコルで証明書失効リストを有効にするかどうかを指定します。**zookeeper.ssl.crl** システムプロパティーで設定した明示的な値を上書きします（短い名前に注意してください）。

zookeeper.ssl.enabled.protocols

type: list
デフォルト： null
Importance: low
Dynamic update: read-only
ZooKeeper TLS ネゴシエーション(csv)で有効なプロトコルを指定します。**zookeeper.ssl.enabledProtocols** システムプロパティーで設定した明示的な値を上書きします（camelCase をメモします）。**null** のデフォルト値は、有効なプロトコルが **zookeeper.ssl.protocol** 設定プロパティーの値になります。

zookeeper.ssl.endpoint.identification.algorithm

type: string
デフォルト： HTTPS
インポート： low
Dynamic update: read-only
ZooKeeper TLS ネゴシエーションプロセスでホスト名検証を有効にするかどうかを指定します（大文字と小文字は区別されない）"https" の場合、ZooKeeper ホスト名の検証が有効になり、明示的な空の値（テスト目的のみ）の使用が推奨されます。明示的な値は、**zookeeper.ssl.hostnameVerification** システムプロパティーで設定した「true」または「false」の値を上書きします（異なる名前と値は https および false は空白を意味します）。

zookeeper.ssl.ocsp.enable

型： ブール値
デフォルト： false
インポート機能： low

Dynamic update: read-only
ZooKeeper TLS プロトコルで Online Certificate Status Protocol を有効にするかどうかを指定します。**zookeeper.ssl.ocsp** システムプロパティーで設定した明示的な値を上書きします（短い名前に注意してください）。

zookeeper.ssl.protocol

型： 文字列
デフォルト： TLSv1.2
インポート： low
Dynamic update: read-only
ZooKeeper TLS ネゴシエーションで使用するプロトコルを指定します。明示的な値は、同じ名前の **zookeeper.ssl.protocol** システムプロパティーで設定されているすべての値を上書きします。

zookeeper.sync.time.ms

型： int
デフォルト： 2000（2 秒）
の**インポート：** low
Dynamic update: read-only
ZK フォロワーが ZK リーダーの背後にある方法。

付録B トピック設定パラメーター

cleanup.policy

type: list
デフォルト： delete
有効な値を削除します： [compact, delete]
Server Default Property: log.cleanup.policy
Importance: medium
"delete" または "compact"（両文字列）のいずれかである文字列。この文字列は、古いログセグメントで使用する保持ポリシーを指定します。デフォルトのポリシー(delete)は、保持期間またはサイズ制限に達した場合に古いセグメントを破棄します。「 comact」設定は、トピックの ログコンパクション が有効になります。

compression.type

型： 文字列
デフォルト： プロデューサー
の有効な値： [uncompressed, zstd, lz4, snappy, gzip, producer]
Server Default Property: compression.type
Importance: medium
特定のトピックの最終圧縮タイプを指定します。この設定では、標準の圧縮コーデック（'gzip'、'snappy'、'lz4'、'zstd'）を受け入れます。また、圧縮なしでの 'uncompressed' および 'producer' を受け入れます。これは、プロデューサーによって設定された元の圧縮コード c を保持します。

delete.retention.ms

Type: long
Default: 86400000(1 day)
有効な値： [0,...]
Server Default Property: log.cleaner.delete.retention.ms
Importance: medium
ログコンパクションされたトピック の削除トゥームストーンのマーカーを保持する時間。この設定は、オフセット 0 から開始して、最終ステージの有効なスナップショットを取得する場合にコンシューマーが読み取りを完了する必要のある時間にバインドし、それらのスキャンの完了前に廃棄します。

file.delete.delay.ms

Type: long
Default: 60000（1 分）
有効な値： [0,...]
Server Default Property: log.segment.delete.delay.ms
Importance: medium
ファイルシステムからファイルを削除するまでの待機時間。

flush.messages

タイプ： long
デフォルト： 9223372036854775807
有効な値 [0, ...]
server Default Property: log.flush.interval.messages
Importance: medium
この設定により、ログに書き込まれたデータの fsync を強制する間隔を指定できます。たとえば、これがすべてのメッセージの後に fsync を指定した場合は、5 つのメッセージ以降は 5 回は fsync になります。通常、これは設定せず、耐久性を確保し、オペレーティングシステムのバックグラウン

ドフラッシュ機能を効率化することが推奨されます。この設定は、トピックごとに上書きできます（トピックごとの 設定についてのセクションを参照してください）。

flush.ms

タイプ： long
デフォルト： 9223372036854775807
有効な値 [0, ...]
サーバーのデフォルトプロパティー： log.flush.interval.ms
Importance: medium
この設定により、ログに書き込まれたデータの fsync を強制する期間を指定できます。たとえば、これが 1000 ミリ秒が経過した後に fsync を 1000 ミリ秒に設定します。通常、これは設定せず、耐久性を確保し、オペレーティングシステムのバックグラウンドフラッシュ機能を効率化することが推奨されます。

follower.replication.throttled.replicas

タイプ： list
デフォルト： ""
有効な値： [partitionId],[brokerId]:[brokerId],[brokerId], ...
サーバーデフォルトプロパティー： **follower**.replication.throttled.replicas
Importance**Importance:** medium
ログレプリケーションがフォロワー側でスロットリングされる必要があるレプリカの一覧。この一覧は、[PartitionId]:[BrokerId],[BrokerId]:[BrokerId]:... の形式でレプリカのセットを記述するか、ワイルドカード '*' を使用して、このトピックのすべてのレプリカにスロットリングできます。

index.interval.bytes

型： int
デフォルト： 4096(4 kibibytes)
有効な値 [0,...]
Server Default Property: log.index.interval.bytes
Importance: medium
この設定は、Kafka がインデックスエントリーをそのオフセットインデックスに追加する頻度を制御します。デフォルト設定では、4096 バイトごとにメッセージをインデックス化します。インデックスが多いと、読み取りはログの正確な位置にジャンプできますが、インデックスが大きくなるようにします。これを変更する必要はありません。

leader.replication.throttled.replicas

タイプ： list
デフォルト： ""
有効な値： [partitionId],[brokerId]:[brokerId],[brokerId], ...
サーバーデフォルトプロパティー： leader.replication.throttled.replicas
Importance: medium
ログレプリケーションがリーダー側でスロットリングされる必要があるレプリカの一覧。この一覧は、[PartitionId]:[BrokerId],[BrokerId]:[BrokerId]:... の形式でレプリカのセットを記述するか、ワイルドカード '*' を使用して、このトピックのすべてのレプリカにスロットリングできます。

max.compaction.lag.ms

Type: long
Default: 9223372036854775807
有効な値 [1,...]
Server Default Property: log.cleaner.max.compaction.lag.ms
importance: medium
メッセージはログ内で圧縮できない最大時間。圧縮されているログにのみ適用可能です。

max.message.bytes

type: int
デフォルト： 1048588
有効な値 [0,...]
Server Default Property: message.max.bytes
Importance: medium
Kafka で許可される最大レコードバッチサイズ（圧縮が有効な場合の圧縮後）。これが増加され、0.10.2 よりも古いコンシューマーがある場合、コンシューマーのフェッチサイズも拡張し、この大量に記録できるようにします。最新のメッセージ形式バージョンでは、レコードは常にバッチにグループ化され、効率性を確保します。以前のメッセージ形式のバージョンでは、圧縮されていないレコードはバッチにグループ化されず、この制限はその場合の単一レコードにのみ適用されます。

message.format.version

型： string
デフォルト： 2.7-IV2
有効な値： [0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV0 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.1-IV0、3IV0-IV0-IV0 2.7-IV0, 2.7-IV1, 2.7-IV2]
Server Default Property: log.message.format.version
Importance: medium
ブローカーがログにメッセージを追加するのに使用するメッセージ形式のバージョンを指定します。値は有効な ApiVersion である必要があります。例としては、0.8.2、0.9.0.0、0.10.0 などがあります。詳細は、ApiVersion を参照してください。特定のメッセージ形式のバージョンを設定することで、ディスク上の既存のメッセージがすべて、指定されたバージョンよりも小さくなることが認定されています。この値を誤って設定すると、コンシューマーが認識しない形式のメッセージを受信してしまうため、古いバージョンのコンシューマーが破損することになります。

message.timestamp.difference.max.ms

タイプ： long
デフォルト： 9223372036854775807
有効な値 [0, ...]
server Default Property: log.message.timestamp.difference.max.ms
Importance: medium
ブローカーがメッセージとメッセージを受け取ると、タイムスタンプとメッセージに指定されたタイムスタンプの間の最大差。message.timestamp.type=CreateTime の場合、タイムスタンプの差異がこのしきい値を超えるとメッセージは拒否されます。この構成は、message.timestamp.type=LogAppluatex の場合に無視されます。

message.timestamp.type

Type: string
デフォルト： CreateTime
有効な値： [CreateTime, LogApp the]
Server Default Property: log.message.timestamp.type
importance: medium
メッセージのタイムスタンプが create time であるか、またはログの追加時間であるかを定義します。値は **CreateTime** または **LogAppendTime** のいずれかでなければなりません。

min.cleanable.dirty.ratio

Type: double
Default: 0.5
Valid Values: [0,...,1]
Server Default Property: log.cleaner.min.cleanable.ratio
Importance: medium

この設定では、ログコンパクターがログの消去を試行する頻度を制御します（ログコンパクションが有効になっていることを前提とします）。デフォルトでは、ログの 50% を超えるログのクリーニングを防ぎます。この比率は、重複してログに最大領域がバインドされます（ログの 50% は最大で 50% になる可能性がありました）。比率が大きいほど、より効率的なクリーニングが行われますが、ログの領域がさらに多くなります。max.compaction.lag.ms 設定または min.compaction.lag.ms 設定も指定された場合、ログコンパクターも、ダーティー率のしきい値が満たされ、ログにはダーティー率のしきい値が満たされ、ログが min.compaction.lagms の期間と即座に圧縮できると見えます。 または、ほとんどの max.compaction.lag.ms の期間にログにダーティー（複雑でない）レコードがある場合。

min.compaction.lag.ms

Type: long
デフォルト： 0
有効な値： [0,...]
Server Default Property: log.cleaner.min.compaction.lag.ms
Importance: medium
メッセージがログに記録されない最小の時間。圧縮されているログにのみ適用可能です。

min.insync.replicas

型： int
デフォルト： 1
有効な値： [1,...]
Server Default Property: min.insync.replicas
Importance: medium
プロデューサーが acks を「all」（または「-1」）に設定すると、この設定で書き込みが成功したとみなされる必要のあるレプリカの最小数を指定します。この最小値が満たされない場合、プロデューサーは例外を送出します（NotEnoughReplicas または NotEnoughReplicasAfterAppend）。一緒に使用すると、**min.insync.replicas** と **acks** を使用すると、耐久性の高い保証を実施することができます。典型的なシナリオは、レプリケーション係数 3 でトピックを作成し、**min.insync.replicas** を 2 に設定し、**acks** の「all」で作成します。これにより、多くのレプリカが書き込みを受信しない場合にプロデューサーが例外を発生するようになります。

preallocate

型： ブール値
デフォルト： false
Server Default Property: log.preallocate
Importance: medium
新しいログセグメントの作成時にディスクにファイルを事前に割り当てた場合には True。

retention.bytes

型： long
デフォルト： -1
Server デフォルトプロパティ： log.retention.bytes
Importance: medium
この設定では、「delete」保持ポリシーを使用している場合に、古いログセグメントを破棄する前にパーティションを拡張できるように、パーティションの最大サイズを制御します。デフォルトでは、サイズ制限は制限されません。この制限はパーティションレベルで実施されるため、パーティションの数で乗算して、トピックの保持（バイト単位）を計算します。

retention.ms

型： long
デフォルト： 604800000(7 days)
有効な値： [-1,...]

Server Default Property: log.retention.ms
Importance: medium
この設定では、「delete」保持ポリシーを使用している場合には、古いログセグメントを破棄する前にログを保持する最大時間を制御します。これは、コンシューマーがデータを読み取れる必要があるかについての SLA を表します。-1 に設定すると、時間制限は適用されません。

segment.bytes

型： int
デフォルト： 1073741824(1 gibibyte)
有効な値 [14, ...]
サーバーのデフォルトプロパティー： log.segment.bytes
Importance: medium
この設定では、ログのセグメントファイルサイズを制御します。保持期間とクリーニングは常にファイルが実行されるため、より大きなセグメントサイズはファイルより少なくなりますが、保持に対する詳細な制御が低くなります。

segment.index.bytes

type: int
デフォルト： 10485760(10 mebibytes)
有効な値 [0,...]
Server Default Property: log.index.size.max.bytes
Importance: medium
この設定では、オフセットをファイルの位置にマップするインデックスのサイズを制御します。このインデックスファイルに事前割り当てを行い、ログのロール後にのみ縮小します。通常、この設定を変更する必要はありません。

segment.jitter.ms

型： long
デフォルト： 0
有効な値： [0,...]
Server Default Property: log.roll.jitter.ms
Importance: medium
スケジュールされたセグメントのゼロ時間からランダムのジッターで引き出された最大のジッターは、セグメントのゲームメンテーションを防ぎます。

segment.ms

型： long
デフォルト： 604800000(7 days)
有効な値： [1,...]
Server Default Property: log.roll.ms
Importance: medium
この設定では、セグメントファイルが満杯になっても、保持が古いデータを削除またはコンパクトにするために、Kafka が強制的にロールバックする期間を制御します。

unclean.leader.election.enable

Type: boolean
デフォルト： false
Server Default Property: unclean.leader.election.enable
Importance: medium
ISR が設定されていないレプリカを、最後の手段としてリーダーとして選択するようにし、データが失われる可能性があるかどうかを示します。

message.downconversion.enable

型： ブール値
デフォルト： true
Server のデフォルト Property: log.message.downconversion.enable
Importance: low
この設定は、リクエストを消費するために、ダウンコンバートバージョンのメッセージ形式が有効になっているかどうかを制御します。**false** に設定すると、古いメッセージ形式を必要とするコンシューマーに対してブローカーはダウンコンバートを実行しません。このような古いクライアントからの要求を消費するため、ブローカーは **UNSUPPORTED_VERSION** エラーで応答します。この設定は、レプリケーションのフォロワーに必要となる可能性のあるメッセージ形式の変換には適用されません。

付録C コンシューマー設定パラメーター

key.deserializer

型： クラス
のインポート： high
org.apache.kafka.common.serialization.Deserializer インターフェースを実装するキーの Deserializer クラス。

value.deserializer

型： クラス
のインポート： high
org.apache.kafka.common.serialization.Deserializer インターフェースを実装する値の Deserializer クラス。

bootstrap.servers

type: list
デフォルト： ""
有効な値： non-null string
Importance: high
Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ブートストラップ用に指定されたすべてのサーバーに関係なく、サーバーの完全なセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2, ...** の形式にする必要があります。これらのサーバーは、クラスターの完全なメンバーシップを検出するために最初の接続に使用されるため（動的に変更される可能性がある）、この一覧にはサーバーの完全セットを含める必要はありません（サーバーがダウンした場合など）。

fetch.min.bytes

型： int
デフォルト： 1
有効な値： [0,...]
Importance: high
フェッチリクエストに対してサーバーが返すデータの最小量。十分なデータが十分にない場合、リクエストはその大量のデータが累積されるまで待機してからリクエストに応答します。デフォルトの 1 バイト設定は、データの 1 バイトが利用可能になり次第フェッチリクエストが即座に応答するか、またはデータに到達するのを待機するフェッチリクエストがタイムアウトすることを意味します。これを 1 よりも大きい値に設定すると、サーバーは大量のデータを累積するのを待機し、追加のレイテンシーのコストでサーバーのスループットを向上できます。

group.id

type: string
デフォルト： null
Importance: high
このコンシューマーが属するコンシューマーグループを識別する一意の文字列。このプロパティーは、コンシューマーが **subscribe(topic)** または Kafka ベースのオフセット管理システムを使用してグループ管理機能を使用する必要がある場合に必要です。

heartbeat.interval.ms

型： int
デフォルト： 3000（3 秒）
のインポート： high
Kafka のグループ管理機能の使用時にハートビートをコンシューマーコーディネーターにかかる合計時間。ハートビートは、コンシューマーのセッションがアクティブになり、新しいコンシューマー

がグループに参加または残す際にリバランスを容易にするために使用されます。値は **session.timeout.ms** よりも低い値を設定する必要がありますが、通常はその値の 1/3 よりも高い値を設定する必要があります。通常のリバランスの予想される時間を制御するように調整することもできます。

max.partition.fetch.bytes

型： int
デフォルト： 1048576(1 mebibyte)
有効な値： [0,...]
Importance: high
サーバーが返すパーティションごとのデータの最大量。レコードはコンシューマーによってバッチで取得されます。フェッチの最初の空でないパーティションで最初のレコードバッチがこの制限よりも大きい場合、バッチが返され、コンシューマーの進捗が保たれるようにします。ブローカーで許可される最大レコードバッチサイズは、**message.max.bytes** （ブローカー設定）または **max.message.bytes** （トピック設定）で定義されます。コンシューマー要求サイズを制限する場合は fetch.max.bytes を参照してください。

session.timeout.ms

型： int
デフォルト： 10000（10 秒）
のイン**ポート：** high
Kafka のグループ管理機能の使用時にクライアントの失敗を検出するために使用されるタイムアウト。クライアントは定期的なハートビートを送信し、その liveness をブローカーに示します。このセッションタイムアウトの期限が切れる前にブローカーによってハートビートが受信されなかった場合、ブローカーはグループからこのクライアントを削除し、リバランスを開始します。この値は、**group.min.session.timeout.ms** および **group.max.session.timeout.ms** によってブローカー設定で設定されるため、許容範囲である必要があることに注意してください。

ssl.key.password

タイプ： password
デフォルト： null
イン**ポート：** high
キーストアファイルまたは 'ssl.keystore.key' に指定された PEM キーのパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

タイプ： password
デフォルト： null
イン**ポート：** high
'ssl.keystore.type' により指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書の一覧で PEM 形式のみをサポートします。

ssl.keystore.key

タイプ： password
デフォルト： null
イン**ポート：** high
'ssl.keystore.type' により指定された形式の秘密鍵。デフォルトの SSL エンジンファクトリーは、PKCS#8 鍵を持つ PEM 形式のみをサポートします。鍵を暗号化する場合は、'ssl.key.password' を使用して鍵のパスワードを指定する必要があります。

ssl.keystore.location

type: string
デフォルト： null
Importance: high
キーストアファイルの場所。これはクライアント用にオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

タイプ： password
デフォルト： null
インポート：** high
キーストアファイルのストアパスワード。これはクライアントでオプションであり、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアパスワードは PEM 形式ではサポートされません。

ssl.truststore.certificates

タイプ： password
デフォルト： null
インポート：** high
'ssl.truststore.type' によって指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書で PEM 形式のみをサポートします。

ssl.truststore.location

type: string
デフォルト： null
Importance: high
トラストストアファイルの場所。

ssl.truststore.password

タイプ： password
デフォルト： null
インポート：** high
トラストストアファイルのパスワード。パスワードが設定されていないと、設定したトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアパスワードは PEM 形式ではサポートされません。

allow.auto.create.topics

型： ブール値
デフォルト： true
インポート：** medium
トピックにサブスクライブしたり、割り当てる際にブローカーでトピックの自動作成を許可します。ブローカーが **auto.create.topics.enable** ブローカー設定を使用できる場合のみ、サブスクライブされるトピックが自動的に作成されます。0.11.0 を経過したブローカーを使用する場合は、この設定を **false** に設定する必要があります。

auto.offset.reset

型： 文字列
デフォルト： 最新の
有効値： [latest, earliest, none]
Importance: medium
Kafka に初期オフセットがない場合や、現在のオフセットがサーバーに多い場合（そのデータが削除されているため）行うこと。

- earliest: オフセットを最速のオフセットに自動的にリセット
- latest: オフセットを最新のオフセットに自動的にリセット
- none: コンシューマーのグループに前のオフセットが見つからない場合は、コンシューマーに例外をスローします。
- 上記以外： コンシューマーに例外をスローします。

client.dns.lookup

type: string
デフォルト**:** use_all_dns_ips
有効な値： [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]
Importance: medium
クライアントが DNS ルックアップをどのように使用するかを制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスに接続します。切断後、次の IP が使用されます。すべての IP を一度使用した後、クライアントはホスト名から IP を再度解決します（JVM と OS キャッシュ DNS 名の検索の両方）。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズ後、これは **use_all_dns_ips** と同じように動作します。**default**（非推奨）に設定すると、ルックアップが複数の IP アドレスを返しても、ルックアップが返した最初の IP アドレスへの接続を試行します。

connections.max.idle.ms

タイプ： long
デフォルト： 540000（9 分）
Importance: medium
この設定によって指定されるミリ秒数の後にアイドル状態の接続を閉じます。

default.api.timeout.ms

型： int
デフォルト： 60000（1 分）
有効な値： [0,...]
Importance: medium
クライアント API のタイムアウト（ミリ秒単位）を指定します。この設定は、**timeout** パラメーターを指定しないすべてのクライアント操作のデフォルトタイムアウトとして使用されます。

enable.auto.commit

型： ブール値
デフォルト： true
インポート**：** medium
true の場合、コンシューマーのオフセットはバックグラウンドで定期的にコミットされます。

exclude.internal.topics

型： ブール値
デフォルト： true
インポート**：** medium
サブスクライブしているパターンに一致する内部トピックがサブスクリプションから除外されるかどうか。常に内部トピックを明示的にサブスクライブできます。

fetch.max.bytes

型： int

デフォルト： 52428800(50 mebibytes)
有効な値 [0,...]
Importance: medium
フェッチリクエストに対してサーバーが返すデータの最大量。レコードはコンシューマーによってバッチで取得されます。フェッチの最初の空でないパーティションにある最初のレコードバッチがこの値よりも大きい場合、レコードバッチが返され、コンシューマーの進捗が保たれるようにします。したがって、これは絶対最大値ではありません。ブローカーで許可される最大レコードバッチサイズは、**message.max.bytes** （ブローカー設定）または **max.message.bytes** （トピック設定）で定義されます。コンシューマーは複数のフェッチを並行して実行することに注意してください。

group.instance.id

type: string
デフォルト： null
Importance: medium
エンドユーザーが提供するコンシューマーインスタンスの一意識別子。空でない文字列のみが許可されます。設定されている場合、コンシューマーは静的メンバーとして扱われます。つまり、このIDを持つ1つのインスタンスのみがコンシューマーグループで常に許可されます。これは、一時的な利用不可の状況（プロセス再起動など）が原因のグループのリバランスを回避するために、大きなセッションタイムアウトと組み合わせて使用できます。設定しないと、コンシューマーはグループを動的メンバーとして追加します。これは従来の動作です。

isolation.level

型： 文字列
デフォルト： read_uncommitted
有効な値： [read_committed, read_uncommitted]
Importance: medium
メッセージをトランザクションに書き込まれる方法を制御します。**read_committed** に設定すると、consumer.poll（）はコミットされたトランザクションメッセージのみを返します。'read_uncommitted'（デフォルト）に設定すると、consumer.poll（）は中断されたトランザクションメッセージであってもすべてのメッセージを返します。非トランザクションメッセージは、どちらのいずれのモードで無条件で返されます。

メッセージは常にオフセット順序で返されます。そのため、**read_committed** モードでは、consumer.poll（）は、最初のオープントランザクションのオフセットよりも少ない最後の安定したオフセット(LSO)までのメッセージのみを返します。特に、継続中のトランザクションに属するメッセージの後に表示されるメッセージは、関連するトランザクションが完了するまで正当化されます。その結果、フライトトランザクションがある場合、**read_committed** コンシューマーは高基準値まで読み取りできなくなります。

> Further, when in `read_committed` the seekToEnd method will return the LSO.

max.poll.interval.ms

型： int
デフォルト： 300000（5 分）
有効な値： [1,...]
Importance: medium
コンシューマーグループ管理を使用する場合の poll（）の呼び出しの最大遅延。これにより、より多くのレコードを取得する前にコンシューマーがアイドル状態でいられる期間の上限が追加されます。このタイムアウトが期限切れになる前に poll（）が呼び出されなかった場合、コンシューマーは失敗したと見なされ、パーティションを別のメンバーに再割り当てするためにリバランスを行います。このタイムアウトに到達した null 以外の **group.instance.id** を使用するコンシューマーの場

合、パーティションは即座に再割り当てされません。代わりに、コンシューマーはハートビートの送信を停止し、**session.timeout.ms** の有効期限後に再割り当てされます。これは、シャットダウンされた静的コンシューマーの動作を反映しています。

max.poll.records

型： int
デフォルト： 500
有効な値 [1,...]
Importance: medium
poll（）への単一呼び出しで返されるレコードの最大数。

partition.assignment.strategy

Type: list
デフォルト： class org.apache.kafka.clients.consumer.RangeAssignor
有効な値： non-null string
Importance: medium
グループ管理の使用時にコンシューマーインスタンス間でパーティション所有権の分散に使用するサポート対象のパーティション割り当てストラテジーの優先順位付け。

以下に指定されるデフォルトクラスの他に、パーティションのラウンドロビン割り当てに 'org.apache.kafka.clients.consumer.RoundRobinAssignor'class を使用できます。

org.apache.kafka.clients.consumer.ConsumerPartitionAssignor インターフェースを実装すると、カスタム割り当て strategy にプラグインすることができます。

receive.buffer.bytes

型： int
デフォルト： 65536(64 kibibytes)
有効な値 [-1,...]
Importance: medium
データの読み取り時に使用する TCP 受信バッファー(SO_RCVBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

型： int
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: medium
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。

sasl.client.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

タイプ： password
デフォルト： null
のイン**ポート：** medium
JAAS 設定ファイルで使用される形式で SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式が ここ に記載されています。値の形式は以下のとおりです。「loginModuleClass controlFlag (optionName=optionValue)*;」.ブローカーの場合、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。例：listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string
デフォルト： null
Importance: medium
Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定で定義できます。

sasl.login.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラーの設定には、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。たとえば、listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler です。

sasl.login.class

型： クラス
デフォルト： null
Importance: medium
ログインインターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナー接頭辞と SASL メカニズム名を付ける必要があります。For example, listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.mechanism

type: string
デフォルト： GSSAPI
Importance: medium
クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーを利用できるメカニズムがある場合があります。GSSAPI はデフォルトのメカニズムです。

security.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
ブローカーと通信するために使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

型： int
デフォルト： 131072(128 kibibytes)
有効な値 [-1,...]

Importance: medium
データ送信時に使用する TCP 送信バッファー(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ： long
デフォルト： 127000（127 秒）
Importance: medium
ソケット接続が確立されるまでクライアントが待機する最大時間。接続セットアップのタイムアウトは、この最大値までの連続する接続障害ごとに指数関数的に増加します。接続ストームを回避するために、0.2 のランダム化係数はタイムアウトに適用されます。これにより、計算値の 20% 未満の 20% の範囲がランダムな範囲になります。

socket.connection.setup.timeout.ms

タイプ： long
デフォルト： 10000（10 秒）
のインポート**中：** medium
クライアントがソケット接続を確立するのを待つ時間。タイムアウトが経過する前にコネクションがビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list
デフォルト： TLSv1.2,TLSv1.3
Importance: medium
SSL 接続に対して有効なプロトコル一覧。Java 11 以降、TLSv1.2' の場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーは、TLSv1.2 へのサポートとフォールバックの両方をサポートする場合は、TLSv1.3 が優先されます（両方が TLSv1.2 をサポートしていることを前提とします）。ほとんどの場合、このデフォルト設定は問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

型： string
デフォルト： JKS
Importance: medium
キーストアファイルのファイル形式。これはクライアントでオプションです。

ssl.protocol

型： 文字列
デフォルト： TLSv1.3
のイン**ポート：** medium
SSLContext の生成に使用される SSL プロトコル。Java 11 以降、'TLSv1.2' の場合、デフォルトは 'TLSv1.3' です。この値は、ほとんどのユースケースで問題ありません。最新の JVM の許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされていますが、既知のセキュリティー脆弱性により使用は推奨されません。この設定と 'ssl.enabled.protocols' のデフォルト値で、サーバーが 'TLSv1.3' に対応していない場合は、クライアントは 'TLSv1.2' にダウングレードします。この設定を 'TLSv1.2' に設定すると、ssl.enabled.protocols の値のいずれかであった場合でも、クライアントは 'TLSv1.3' を使用しません。また、サーバーは 'TLSv1.3' のみをサポートします。

ssl.provider

type: string
デフォルト： null
Importance: medium
SSL 接続に使用されるセキュリティープロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティープロバイダーです。

ssl.truststore.type

型： string
デフォルト： JKS
Importance: medium
トラストストアファイルのファイル形式。

auto.commit.interval.ms

型： int
デフォルト： 5000（5 秒）
有効な値： [0,...]
重要： low
enable.auto.commit が **true** に設定されている場合、コンシューマーオフセットが Kafka に自動コミットされる頻度（ミリ秒単位）。

check.crcs

型： ブール値
デフォルト： true
インポート**：** low
消費されるレコードの CRC32 を自動的に確認します。これにより、メッセージへのオンワイヤやディスク上の破損を防ぎます。このチェックではオーバーヘッドがいくつか追加されるため、パフォーマンスが非常に発生する状況では無効にされる可能性があります。

client.id

型： 文字列
デフォルト： ""
Importance: low
要求の実行時にサーバーに渡す id 文字列。これの目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/port 以外の要求のソースを追跡できます。

client.rack

型： 文字列
デフォルト： ""
Importance: low
このクライアントのラック識別子。この値は、このクライアントが物理的に配置される場所を示す任意の文字列値です。ブローカー設定 'broker.rack' に対応します。

fetch.max.wait.ms

型： int
デフォルト： 500
有効な値： [0,...]
Importance: low
fetch.min.bytes で指定された要件を満たすのに十分なデータがない場合には、サーバーがフェッチリクエストに応答する前にサーバーがブロックする最大時間。

interceptor.classes

type: list
デフォルト： ""
有効な値： non-null string
Importance: low
インターセプターとして使用するクラスの一覧。**org.apache.kafka.clients.consumer.ConsumerInterceptor** インターフェースを実装すると、コンシューマーによって受信されるレコードをインターセプト（およびその他）することができます。デフォルトではインターセプターはありません。

metadata.max.age.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: low
パーティションリーダーを変更して、新しいブローカーまたはパーティションを正常に検出していない場合でも、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

type: list
デフォルト： ""
有効な値： non-null string
Importance: low
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
メトリックの計算に維持されるサンプル数。

metrics.recording.level

タイプ： string
デフォルト： INFO
有効な値： [INFO、DEBUG, TRACE]
重要度： low
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: low
メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms

型： long
デフォルト： 1000（1 秒）
有効な値： [0,...]

重要： low
接続に繰り返し失敗したブローカーに再接続するまで待つ最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは、連続した接続失敗ごとに指数関数的に増加します。この最大値までです。バックオフの増加を算出したら、接続の量を回避するために 20% random jitter が追加されます。

reconnect.backoff.ms

型： long
デフォルト： 50
有効な値： [0,...]
Importance: low
指定のホストに再接続を試みる前に待機する時間のベース時間。これにより、密接ループでホストに繰り返し接続できなくなります。このバックオフは、クライアントがブローカーに試行するすべての接続に適用されます。

retry.backoff.ms

型： long
デフォルト： 100
有効な値： [0,...]
Importance: low
失敗したリクエストを特定のトピックパーティションに再試行するまで待つ期間。これにより、一部の障害シナリオにおいて、密接ループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

型： 文字列
デフォルト： /usr/bin/kinit
重要性： low
Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ： long
デフォルト： 60000
インポート：low
更新試行間のログインスレッドスリープ時間。

sasl.kerberos.ticket.renew.jitter

型：
デフォルト： 0.05
インポート：low
更新時間にランダムなジッターが追加されたパーセンテージ。

sasl.kerberos.ticket.renew.window.factor

型：
デフォルト： 0.8
インポート：low
ログインスレッドは、指定のウィンドウ係数からチケットの有効期限に達するまでスリープされます。この期間が経過するとチケットの更新が試行されます。

sasl.login.refresh.buffer.seconds

タイプ： 短い
デフォルト： 300

有効な値： [0,...,3600]
Importance: low
認証情報を更新するときの認証情報の有効期限（秒単位）。そうでなければ、更新がバッファー秒数よりも期限切れになり、可能な限り多くのバッファー時間を維持するよう更新が行われます。法人の値は 0 から 3600（1 時間）で設定されています。値が指定されていない場合には、デフォルト値の 300（5 分）が使用されます。この値と sasl.login.refresh.min.period.seconds は、合計が認証情報の残りの有効期間を超えた場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

タイプ： 短い
デフォルト： 60
有効な値 [0,...,900]
Importance: low
ログイン更新スレッドで認証情報を更新するまで待つ必要最小限の時間（秒単位）。法人の値は 0 から 900（15 分）で設定されています。値が指定されていない場合には、デフォルト値の 60（1 分）が使用されます。この値と sasl.login.refresh.buffer.seconds は、その合計が認証情報の残りの有効期間を超える場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

型：
デフォルト： 0.8
有効な値 [0.5,...,1.0]
Importance: low
ログイン更新スレッドは、認証情報の有効期間に比べ、指定されたウィンドウ係数に達するまでスリープ状態になり、クレデンシャルの更新を試みます。法人の値は 0.5(50%)と 1.0(100%)の範囲になります。値が指定されていない場合には、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

型：
デフォルト： 0.05
有効な値 [0.0,...,0.25]
Importance: low
ログイン更新スレッドのスリープ時間に追加される認証情報の有効期間に対して、ランダムなジッターの最大量。法人の値は 0 から 0.25(25%)で囲まれています。値が指定されていない場合には、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers

型： 文字列
デフォルト： null
インポート： low
セキュリティーアルゴリズムを実装するプロバイダーを返す、設定可能な作成者クラスの一覧。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.cipher.suites

type: list
デフォルト： null
インポート： low
暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC、鍵交換アルゴリズムの名前付きの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサ

ポートされます。

ssl.endpoint.identification.algorithm

型： 文字列
デフォルト： https
インポート：** low
サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

型： クラス
デフォルト： null
のインポート：** low
SSLEngine オブジェクトを提供するために、
org.apache.kafka.common.security.auth.SslEngineFactory タイプのクラス。Default value is
org.apache.kafka.common.security.ssl.DefaultSslEngineFactory.

ssl.keymanager.algorithm

型： 文字列
デフォルト： SunX509
インポート：** low
SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、
Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

型： 文字列
デフォルト： null
インポート：** low
SSL 暗号操作に使用する SecureRandom PRNG 実装。

ssl.trustmanager.algorithm

タイプ： 文字列
デフォルト： PKIX
インポート：** low
SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値
は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

付録D プロデューサー設定パラメーター

key.serializer

型： クラス
のインポート： high
org.apache.kafka.common.serialization.Serializer インターフェースを実装するキーのシリアライザークラス。

value.serializer

型： クラス
のインポート： high
org.apache.kafka.common.serialization.Serializer インターフェースを実装する値のシリアライザークラス。

acks

型： 文字列
デフォルト： 1
有効な値 [all, -1, 0, 1]
重要性： high
リクエストの完了前に、プロデューサーがリーダーを受信する必要がある確認応答。これにより、送信されるレコードの持続性が制御されます。以下の設定が許可されます。

- **acks=0** ゼロに設定すると、プロデューサーはサーバーから送信をまったく待ちません。レコードは即座にソケットバッファーに追加され、送信済みとみなされます。この場合、サーバーがレコードを受信しないように保証はありません（通常障害を把握しないため）。**retries**各レコードに対して指定されたオフセットは常に **-1** に設定されます。

- **acks=1** これは、リーダーがレコードをローカルログに書き込まれますが、すべてのフォロワーからの完全な確認応答を待たずに応答します。この場合、レコードの確認後、フォロワーでレプリケートされる前にリーダーがコミットに失敗しても、レコードが失われます。

- **acks=all** これは、リーダーが in-Sync レプリカの完全なセットがレコードを認識しているのを待つことを意味します。これにより、1 つ以上の In-sync レプリカが動作している限り、レコードが失われなくなります。これは、最も強固な保証です。これは acks=-1 設定と同じです。

bootstrap.servers

type: list
デフォルト： ""
有効な値： non-null string
Importance: high
Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ブートストラップ用に指定されたすべてのサーバーに関係なく、サーバーの完全なセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2, ...** の形式にする必要があります。これらのサーバーは、クラスターの完全なメンバーシップを検出するために最初の接続に使用されるため（動的に変更される可能性がある）、この一覧にはサーバーの完全セットを含める必要はありません（サーバーがダウンした場合など）。

buffer.memory

タイプ： long
デフォルト： 33554432
有効な値： [0,...]
Importance: high

プロデューサーがサーバーに送信されるまで待機するために使用できるメモリーの合計バイト。レコードがサーバーに配信可能よりも早く送信される場合、プロデューサーは **max.block.ms** のブロックされ、例外が発生します。

この設定は、プロデューサーが使用するメモリーの合計のみに対応する必要がありますが、プロデューサーが使用するすべてのメモリーがバッファーに使用されるわけではありません。一部の追加メモリーは圧縮に使用されます（圧縮が有効になっている場合）。またインフライトリクエストを維持するために使用されます。

compression.type

型： 文字列
デフォルト： none
インポート： high
プロデューサーによって生成されたすべてのデータの圧縮タイプ。デフォルトは none です（圧縮なし）。有効な値は **none**、**gzip**、**snappy**、**lz4**、または **zstd** です。圧縮はデータの完全なバッチであるため、バッチ処理の効率が悪くなるため、圧縮率（バッチ処理がより改善）に影響を及ぼします。

retries

型： int
デフォルト： 2147483647
有効値： [0,...,2147483647]
Importance: high
値をゼロに設定すると、クライアントは、一時的なエラーで失敗したレコードを再送信します。この再試行は、クライアントがエラーの受信時にレコードを再入力する場合とは異なることに注意してください。**max.in.flight.requests.per.connection** を 1 に設定せずに再試行を許可すると、レコードの順序が変更する可能性があります。これは、2 つのバッチで 1 つのバッチに送信され、1 番目に再試行されるため、2 番目のバッチのレコードが最初に表示される可能性があります。確認が成功する前に、**delivery.timeout.ms** 期限切れで設定されるタイムアウトが最初に設定されると、再試行の数が使い切られる前にリクエストも失敗することに注意してください。通常は、この設定を未設定のままにし、代わりに **delivery.timeout.ms** を使用して再試行処理を制御することが推奨されます。

ssl.key.password

タイプ： password
デフォルト： null
インポート： high
キーストアファイルまたは 'ssl.keystore.key' に指定された PEM キーのパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

タイプ： password
デフォルト： null
インポート： high
'ssl.keystore.type' により指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書の一覧で PEM 形式のみをサポートします。

ssl.keystore.key

タイプ： password
デフォルト： null
インポート： high

'ssl.keystore.type' により指定された形式の秘密鍵。デフォルトの SSL エンジンファクトリーは、PKCS#8 鍵を持つ PEM 形式のみをサポートします。鍵を暗号化する場合は、'ssl.key.password' を使用して鍵のパスワードを指定する必要があります。

ssl.keystore.location

type: string
デフォルト： null
Importance: high
キーストアファイルの場所。これはクライアント用にオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

タイプ： password
デフォルト： null
インポート： high
キーストアファイルのストアパスワード。これはクライアントでオプションであり、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアパスワードは PEM 形式ではサポートされません。

ssl.truststore.certificates

タイプ： password
デフォルト： null
インポート： high
'ssl.truststore.type' によって指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書で PEM 形式のみをサポートします。

ssl.truststore.location

type: string
デフォルト： null
Importance: high
トラストストアファイルの場所。

ssl.truststore.password

タイプ： password
デフォルト： null
インポート： high
トラストストアファイルのパスワード。パスワードが設定されていないと、設定したトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアパスワードは PEM 形式ではサポートされません。

batch.size

型： int
デフォルト： 16384
有効な値： [0,...]
Importance: medium
複数のレコードが同じパーティションに送信されるたびに、プロデューサーはレコードをより少ないリクエストにバッチ処理を試みます。これは、クライアントとサーバーの両方でパフォーマンスに役立ちます。この設定では、デフォルトのバッチサイズをバイト単位で制御します。

このサイズを超えるレコードのバッチ試行は行われません。

ブローカーに送信されるリクエストには、複数のバッチが含まれます。パーティションごとに、データの送信が可能となります。

バッチサイズが小さいと共通し、スループットを減らすことができます（バッチサイズがゼロの場合はバッチ処理を完全に無効にします）。非常に大きなバッチサイズは、追加のレコードの分析で指定のバッチサイズのバッファーを常に割り当てているため、ビットをより適切に使用できます。

client.dns.lookup

type: string
デフォルト: use_all_dns_ips
有効な値： [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]
Importance: medium
クライアントが DNS ルックアップをどのように使用するかを制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスに接続します。切断後、次の IP が使用されます。すべての IP を一度使用した後、クライアントはホスト名から IP を再度解決します（JVM と OS キャッシュ DNS 名の検索の両方）。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズ後、これは **use_all_dns_ips** と同じように動作します。**default**（非推奨）に設定すると、ルックアップが複数の IP アドレスを返しても、ルックアップが返した最初の IP アドレスへの接続を試行します。

client.id

型： 文字列
デフォルト： ""
Importance: medium
要求の実行時にサーバーに渡す id 文字列。これの目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/port 以外の要求のソースを追跡できます。

connections.max.idle.ms

タイプ： long
デフォルト： 540000（9 分）
Importance: medium
この設定によって指定されるミリ秒数の後にアイドル状態の接続を閉じます。

delivery.timeout.ms

型： int
デフォルト： 120000（2 分）
有効な値： [0,...]
Importance: medium
send() の呼び出しが返った後、または失敗を報告する時間の上限。これにより、送信前にレコードが遅延する合計時間、ブローカーから待機の確認時間（予想される場合）、および再試行可能な送信失敗を許可する時間を制限します。復旧不可能なエラーが発生した場合、プロデューサーはこの設定よりも前にレコード送信の失敗を報告する可能性があります。再試行が使い切られるか、以前の配信有効期限に達したバッチにレコードが追加されます。この設定の値は、**request.timeout.ms** および **linger.ms** の合計以上である必要があります。

linger.ms

タイプ： long
デフォルト： 0
有効な値： [0,...]
Importance: medium
プロデューサーは、リクエスト間で到達するすべてのレコードを 1 つのバッチリクエストにまとめます。通常、これはレコードの送信速度よりも早く到着する場合のみ発生します。ただし、状況に

よっては、クライアントがモードレートであってもリクエストの数を減らす必要がある場合があります。この設定は、送信をバッチ処理できるようにプロデューサーが待機するレコードを即座に送信するのではなく、少数の人為遅延を追加することで行います。これは TCP の Nagle アルゴリズムに似ています。この設定は、バッチ処理の遅延の上限です。この設定に関係なく **batch.size** worthth のレコードを取得した後、このパーティションに対してこのパーティションに累積されたバイト数より少ない場合は、レコードが表示されるまで待機するために指定した時間に 'linger' になります。この設定は、デフォルトで 0（遅延なし）です。たとえば、**linger.ms=5** を設定すると、送信されるリクエスト数を減らすことができますが、負荷がない場合にレコードに送信されるレイテンシーが最大 5ms に追加されます。

max.block.ms

型： long
デフォルト： 60000（1 分）
有効な値： [0,...]
Importance: medium
この設定では、**KafkaProducer's `send()**、**partitionsFor()**、**initTransactions()**、**sendOffsetsToTransaction()**、**commitTransaction()** および **abortTransaction()** メソッドがブロックされる期間を制御します。**send()** このタイムアウトの場合、メタデータフェッチとバッファーの割り当ての両方を待つ合計時間をバインドします（ユーザーが指定したシリアライザーのブロックまたはパーティションャーはこのタイムアウトに対してカウントされません）。**partitionsFor()** の場合、このタイムアウトによりメタデータが使用できない場合に費やされた時間がバインドされます。トランザクション関連のメソッドは常にブロックされますが、トランザクションコーディネーターが検出されなかったり、タイムアウト内で応答しなかった場合にはタイムアウトする場合があります。

max.request.size

型： int
デフォルト： 1048576
有効値： [0,...]
インポート**中：** メディア
要求の最大サイズ（バイト単位）。この設定により、Huge Page が送信しないように、プロデューサーが単一のリクエストに送信するように、レコードバッチの数を制限します。これは、圧縮されていない最大レコードバッチサイズに上限となります。サーバーにはレコードバッチサイズ（圧縮が有効化されている場合）には独自の上限があり、これとは異なる場合があります。

partitioner.class

型： クラス
デフォルト： org.apache.kafka.clients.producer.internals.DefaultPartitioner
Importance: medium
org.apache.kafka.clients.producer.Partitioner インターフェースを実装する partitioner クラス。

receive.buffer.bytes

型： int
デフォルト： 32768(32 kibibytes)
有効な値 [-1,...]
Importance: medium
データの読み取り時に使用する TCP 受信バッファー(SO_RCVBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

型： int
デフォルト： 30000（30 秒）
有効な値： [0,...]

Importance: medium
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。これは、不要なプロデューサーの再試行回数によりメッセージの重複を減らすために、**replica.lag.time.max.ms** （ブローカー設定）よりも大きくする必要があります。

sasl.client.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

タイプ： password
デフォルト： null
のイン**ポート：** medium
JAAS 設定ファイルで使用される形式で SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式が ここ に記載されています。値の形式は以下のとおりです。「loginModuleClass controlFlag (optionName=optionValue)*;」.ブローカーの場合、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。例：listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string
デフォルト： null
Importance: medium
Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定で定義できます。

sasl.login.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラーの設定には、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。たとえば、listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler です。

sasl.login.class

型： クラス
デフォルト： null
Importance: medium
ログインインターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナー接頭辞と SASL メカニズム名を付ける必要があります。For example, listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.mechanism

type: string
デフォルト： GSSAPI
Importance: medium

クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーを利用できるメカニズムがある場合があります。GSSAPI はデフォルトのメカニズムです。

security.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
ブローカーと通信するために使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

型： int
デフォルト： 131072(128 kibibytes)
有効な値 [-1,...]
Importance: medium
データ送信時に使用する TCP 送信バッファー(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ： long
デフォルト： 127000（127 秒）
Importance: medium
ソケット接続が確立されるまでクライアントが待機する最大時間。接続セットアップのタイムアウトは、この最大値までの連続する接続障害ごとに指数関数的に増加します。接続ストームを回避するために、0.2 のランダム化係数はタイムアウトに適用されます。これにより、計算値の 20% 未満の 20% の範囲がランダムな範囲になります。

socket.connection.setup.timeout.ms

タイプ： long
デフォルト： 10000（10 秒）
のインポート**中：** medium
クライアントがソケット接続を確立するのを待つ時間。タイムアウトが経過する前にコネクションがビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list
デフォルト： TLSv1.2,TLSv1.3
Importance: medium
SSL 接続に対して有効なプロトコル一覧。Java 11 以降、TLSv1.2' の場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーは、TLSv1.2 へのサポートとフォールバックの両方をサポートする場合は、TLSv1.3 が優先されます（両方が TLSv1.2 をサポートしていることを前提とします）。ほとんどの場合、このデフォルト設定は問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

型： string
デフォルト： JKS
Importance: medium
キーストアファイルのファイル形式。これはクライアントでオプションです。

ssl.protocol

型： 文字列
デフォルト： TLSv1.3
のイン**ポート：** medium
SSLContext の生成に使用される SSL プロトコル。Java 11 以降、'TLSv1.2' の場合、デフォルトは 'TLSv1.3' です。この値は、ほとんどのユースケースで問題ありません。最新の JVM の許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされていますが、既知のセキュリティー脆弱性により使用は推奨されません。この設定と 'ssl.enabled.protocols' のデフォルト値で、サーバーが 'TLSv1.3' に対応していない場合は、クライアントは 'TLSv1.2' にダウングレードします。この設定を 'TLSv1.2' に設定すると、ssl.enabled.protocols の値のいずれかであった場合でも、クライアントは 'TLSv1.3' を使用しません。また、サーバーは 'TLSv1.3' のみをサポートします。

ssl.provider

type: string
デフォルト： null
Importance: medium
SSL 接続に使用されるセキュリティープロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティープロバイダーです。

ssl.truststore.type

型： string
デフォルト： JKS
Importance: medium
トラストストアファイルのファイル形式。

enable.idempotence

型： ブール値
デフォルト： false
イン**ポート機能：** low
'true' に設定すると、プロデューサーは各メッセージの 1 つのコピーがストリームに書き込まれるようにします。'false' の場合、ブローカーの失敗によりプロデューサーを再試行する場合、ストリームに再試行されたメッセージの重複が発生する可能性があります。べき等を有効にするには、**max.in.flight.requests.per.connection** は 5 以下でなければなりません。**retries** は 0 よりも大きい必要があり、**acks** は「all」にする必要があります。これらの値がユーザーによって明示的に設定されていない場合は、適切な値が選択されます。互換性のない値が設定されている場合、**ConfigException** が発生します。

interceptor.classes

type: list
デフォルト： ""
有効な値： non-null string
Importance: low
インターセプターとして使用するクラスの一覧。**org.apache.kafka.clients.producer.ProducerInterceptor** インターフェースを実装することで、Kafka クラスターにパブリッシュされる前にプロデューサーによって受信されるレコードをインターセプト（およびその他）することができます。デフォルトではインターセプターはありません。

max.in.flight.requests.per.connection

型： int
デフォルト： 5
有効な値 [1,...]
Importance: low

ブロッキング前にクライアントが単一の接続に送信するリクエストの最大数。この設定が 1 よりも大きいよう設定され、送信に失敗した場合には、再試行によってメッセージの順序が繰り返されるリスク（再試行が有効になっている場合など）が発生することに注意してください。

metadata.max.age.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: low
パーティションリーダーを変更して、新しいブローカーまたはパーティションを正常に検出していない場合でも、メタデータの更新を強制する期間（ミリ秒単位）。

metadata.max.idle.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [5000,...]
Importance: low
プロデューサーがアイドル状態のトピックのメタデータをキャッシュする期間を制御します。トピックが最後にメタデータのアイドル時間よりも経過した時間が経過する場合、トピックのメタデータは記憶され、次にメタデータフェッチリクエストが強制されます。

metric.reporters

type: list
デフォルト： ""
有効な値： non-null string
Importance: low
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
メトリックの計算に維持されるサンプル数。

metrics.recording.level

タイプ： string
デフォルト： INFO
有効な値： [INFO、DEBUG, TRACE]
重要度： low
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: low
メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms

型： long
デフォルト： 1000（1秒）
有効な値： [0,...]
重要： low
接続に繰り返し失敗したブローカーに再接続するまで待つ最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは、連続した接続失敗ごとに指数関数的に増加します。この最大値までです。バックオフの増加を算出したら、接続の量を回避するために 20% random jitter が追加されます。

reconnect.backoff.ms

型： long
デフォルト： 50
有効な値： [0,...]
Importance: low
指定のホストに再接続を試みる前に待機する時間のベース時間。これにより、密接ループでホストに繰り返し接続できなくなります。このバックオフは、クライアントがブローカーに試行するすべての接続に適用されます。

retry.backoff.ms

型： long
デフォルト： 100
有効な値： [0,...]
Importance: low
失敗したリクエストを特定のトピックパーティションに再試行するまで待つ期間。これにより、一部の障害シナリオにおいて、密接ループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

型： 文字列
デフォルト： /usr/bin/kinit
重要性： low
Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ： long
デフォルト： 60000
インポート：low
更新試行間のログインスレッドスリープ時間。

sasl.kerberos.ticket.renew.jitter

型：
デフォルト： 0.05
インポート：low
更新時間にランダムなジッターが追加されたパーセンテージ。

sasl.kerberos.ticket.renew.window.factor

型：
デフォルト： 0.8
インポート：low
ログインスレッドは、指定のウィンドウ係数からチケットの有効期限に達するまでスリープされます。この期間が経過するとチケットの更新が試行されます。

sasl.login.refresh.buffer.seconds

タイプ： 短い
デフォルト： 300
有効な値： [0,...,3600]
Importance: low
認証情報を更新するときの認証情報の有効期限（秒単位）。そうでなければ、更新がバッファー秒数よりも期限切れになり、可能な限り多くのバッファー時間を維持するよう更新が行われます。法人の値は 0 から 3600（1 時間）で設定されています。値が指定されていない場合には、デフォルト値の 300（5 分）が使用されます。この値と sasl.login.refresh.min.period.seconds は、合計が認証情報の残りの有効期間を超えた場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

タイプ： 短い
デフォルト： 60
有効な値 [0,...,900]
Importance: low
ログイン更新スレッドで認証情報を更新するまで待つ必要最小限の時間（秒単位）。法人の値は 0 から 900（15 分）で設定されています。値が指定されていない場合には、デフォルト値の 60（1 分）が使用されます。この値と sasl.login.refresh.buffer.seconds は、その合計が認証情報の残りの有効期間を超える場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

型：
デフォルト： 0.8
有効な値 [0.5,...,1.0]
Importance: low
ログイン更新スレッドは、認証情報の有効期間に比べ、指定されたウィンドウ係数に達するまでスリープ状態になり、クレデンシャルの更新を試みます。法人の値は 0.5(50%)と 1.0(100%)の範囲になります。値が指定されていない場合には、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

型：
デフォルト： 0.05
有効な値 [0.0,...,0.25]
Importance: low
ログイン更新スレッドのスリープ時間に追加される認証情報の有効期間に対して、ランダムなジッターの最大量。法人の値は 0 から 0.25(25%)で囲まれています。値が指定されていない場合には、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers

型： 文字列
デフォルト： null
インポート： low
セキュリティーアルゴリズムを実装するプロバイダーを返す、設定可能な作成者クラスの一覧。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.cipher.suites

type: list
デフォルト： null
インポート： low

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC、鍵交換アルゴリズムの名前付きの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.endpoint.identification.algorithm

型： 文字列
デフォルト： https
インポート： low
サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

型： クラス
デフォルト： null
のインポート： low
SSLEngine オブジェクトを提供するために、
org.apache.kafka.common.security.auth.SslEngineFactory タイプのクラス。Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory.

ssl.keymanager.algorithm

型： 文字列
デフォルト： SunX509
インポート： low
SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

型： 文字列
デフォルト： null
インポート： low
SSL 暗号操作に使用する SecureRandom PRNG 実装。

ssl.trustmanager.algorithm

タイプ： 文字列
デフォルト： PKIX
インポート： low
SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

transaction.timeout.ms

型： int
デフォルト： 60000（1 分）
Importance: low
トランザクションコーディネーターがプロデューサーからトランザクションステータスの更新を待機する最大時間（ミリ秒単位）。この値が継続中のトランザクションをプロアクティブに中止します。この値がブローカーの transaction.max.timeout.ms 設定よりも大きい場合、リクエストは **InvalidTxnTimeoutException** エラーを出して失敗します。

transactional.id

型： 文字列
デフォルト： null

有効な値： non-empty string
Importance: low
トランザクション配信に使用する TransactionalId。これにより、新しいトランザクションを開始する前に、クライアントが同じ TransactionalId を使用するトランザクションを確実に完了させるため、複数のプロデューサーセッションにまたがる信頼性セマンティクスが可能になります。TransactionalId が指定されていない場合、プロデューサーはべき等配信に制限されます。TransactionalId が設定されている場合、**enable.idempotence** は暗黙的に使用されます。デフォルトでは TransactionId は設定されません。そのため、トランザクションは使用できません。デフォルトでは、トランザクションには、実稼働の推奨設定である 3 つ以上のブローカーで構成されるブローカーで構成されるクラスターが必要なことに注意してください。開発の場合は、ブローカー **transaction.state.log.replication.factor** を調整してこれを変更できます。

付録E 管理クライアント設定パラメーター

bootstrap.servers

型： list
のインポート： high
Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ブートストラップ用に指定されたすべてのサーバーに関係なく、サーバーの完全なセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2, ...** の形式にする必要があります。これらのサーバーは、クラスターの完全なメンバーシップを検出するために最初の接続に使用されるため（動的に変更される可能性がある）、この一覧にはサーバーの完全セットを含める必要はありません（サーバーがダウンした場合など）。

ssl.key.password

タイプ： password
デフォルト： null
インポート： high
キーストアファイルまたは 'ssl.keystore.key' に指定された PEM キーのパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

タイプ： password
デフォルト： null
インポート： high
'ssl.keystore.type' により指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書の一覧で PEM 形式のみをサポートします。

ssl.keystore.key

タイプ： password
デフォルト： null
インポート： high
'ssl.keystore.type' により指定された形式の秘密鍵。デフォルトの SSL エンジンファクトリーは、PKCS#8 鍵を持つ PEM 形式のみをサポートします。鍵を暗号化する場合は、'ssl.key.password' を使用して鍵のパスワードを指定する必要があります。

ssl.keystore.location

type: string
デフォルト： null
Importance: high
キーストアファイルの場所。これはクライアント用にオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

タイプ： password
デフォルト： null
インポート： high
キーストアファイルのストアパスワード。これはクライアントでオプションであり、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアパスワードは PEM 形式ではサポートされません。

ssl.truststore.certificates

タイプ： password
デフォルト： null
インポート： high
'ssl.truststore.type' によって指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書で PEM 形式のみをサポートします。

ssl.truststore.location

type: string
デフォルト： null
Importance: high
トラストストアファイルの場所。

ssl.truststore.password

タイプ： password
デフォルト： null
インポート： high
トラストストアファイルのパスワード。パスワードが設定されていないと、設定したトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアパスワードは PEM 形式ではサポートされません。

client.dns.lookup

type: string
デフォルト: use_all_dns_ips
有効な値： [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]
Importance: medium
クライアントが DNS ルックアップをどのように使用するかを制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスに接続します。切断後、次の IP が使用されます。すべての IP を一度使用した後、クライアントはホスト名から IP を再度解決します（JVM と OS キャッシュ DNS 名の検索の両方）。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズ後、これは **use_all_dns_ips** と同じように動作します。**default**（非推奨）に設定すると、ルックアップが複数の IP アドレスを返しても、ルックアップが返した最初の IP アドレスへの接続を試行します。

client.id

型： 文字列
デフォルト： ""
Importance: medium
要求の実行時にサーバーに渡す id 文字列。これの目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/port 以外の要求のソースを追跡できます。

connections.max.idle.ms

タイプ： long
デフォルト： 300000（5 分）
Importance: medium
この設定によって指定されるミリ秒数の後にアイドル状態の接続を閉じます。

default.api.timeout.ms

型： int
デフォルト： 60000（1 分）
有効な値： [0,...]
Importance: medium

クライアント API のタイムアウト（ミリ秒単位）を指定します。この設定は、**timeout** パラメーターを指定しないすべてのクライアント操作のデフォルトタイムアウトとして使用されます。

receive.buffer.bytes

型： int
デフォルト： 65536(64 kibibytes)
有効な値 [-1,...]
Importance: medium
データの読み取り時に使用する TCP 受信バッファー(SO_RCVBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

型： int
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: medium
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。

sasl.client.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

タイプ： password
デフォルト： null
のイン**ポート：** medium
JAAS 設定ファイルで使用される形式で SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式が ここ に記載されています。値の形式は以下のとおりです。「loginModuleClass controlFlag (optionName=optionValue)*;」.ブローカーの場合、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。例：listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string
デフォルト： null
Importance: medium
Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定で定義できます。

sasl.login.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラーの設定には、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。たとえば、

listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler です。

sasl.login.class

型： クラス
デフォルト： null
Importance: medium
ログインインターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナー接頭辞と SASL メカニズム名を付ける必要があります。For example, listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.mechanism

type: string
デフォルト： GSSAPI
Importance: medium
クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーを利用できるメカニズムがある場合があります。GSSAPI はデフォルトのメカニズムです。

security.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
ブローカーと通信するために使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

型： int
デフォルト： 131072(128 kibibytes)
有効な値 [-1,...]
Importance: medium
データ送信時に使用する TCP 送信バッファー(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ： long
デフォルト： 127000（127 秒）
Importance: medium
ソケット接続が確立されるまでクライアントが待機する最大時間。接続セットアップのタイムアウトは、この最大値までの連続する接続障害ごとに指数関数的に増加します。接続ストームを回避するために、0.2 のランダム化係数はタイムアウトに適用されます。これにより、計算値の 20% 未満の 20% の範囲がランダムな範囲になります。

socket.connection.setup.timeout.ms

タイプ： long
デフォルト： 10000（10 秒）
のインポート**中：** medium
クライアントがソケット接続を確立するのを待つ時間。タイムアウトが経過する前にコネクションがビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list
デフォルト： TLSv1.2,TLSv1.3
Importance: medium
SSL 接続に対して有効なプロトコル一覧。Java 11 以降、TLSv1.2' の場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーは、TLSv1.2 へのサポートとフォールバックの両方をサポートする場合は、TLSv1.3 が優先されます（両方が TLSv1.2 をサポートしていることを前提とします）。ほとんどの場合、このデフォルト設定は問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

型： string
デフォルト： JKS
Importance: medium
キーストアファイルのファイル形式。これはクライアントでオプションです。

ssl.protocol

型： 文字列
デフォルト： TLSv1.3
のイン**ポート：** medium
SSLContext の生成に使用される SSL プロトコル。Java 11 以降、'TLSv1.2' の場合、デフォルトは 'TLSv1.3' です。この値は、ほとんどのユースケースで問題ありません。最新の JVM の許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされていますが、既知のセキュリティー脆弱性により使用は推奨されません。この設定と 'ssl.enabled.protocols' のデフォルト値で、サーバーが 'TLSv1.3' に対応していない場合は、クライアントは 'TLSv1.2' にダウングレードします。この設定を 'TLSv1.2' に設定すると、ssl.enabled.protocols の値のいずれかであった場合でも、クライアントは 'TLSv1.3' を使用しません。また、サーバーは 'TLSv1.3' のみをサポートします。

ssl.provider

type: string
デフォルト： null
Importance: medium
SSL 接続に使用されるセキュリティープロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティープロバイダーです。

ssl.truststore.type

型： string
デフォルト： JKS
Importance: medium
トラストストアファイルのファイル形式。

metadata.max.age.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: low
パーティションリーダーを変更して、新しいブローカーまたはパーティションを正常に検出していない場合でも、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

タイプ： list
デフォルト： ""
Importance: low
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
メトリックの計算に維持されるサンプル数。

metrics.recording.level

タイプ： string
デフォルト： INFO
有効な値： [INFO、DEBUG, TRACE]
重要度： low
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: low
メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms

型： long
デフォルト： 1000（1 秒）
有効な値： [0,...]
重要： low
接続に繰り返し失敗したブローカーに再接続するまで待つ最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは、連続した接続失敗ごとに指数関数的に増加します。この最大値までです。バックオフの増加を算出したら、接続の量を回避するために 20% random jitter が追加されます。

reconnect.backoff.ms

型： long
デフォルト： 50
有効な値： [0,...]
Importance: low
指定のホストに再接続を試みる前に待機する時間のベース時間。これにより、密接ループでホストに繰り返し接続できなくなります。このバックオフは、クライアントがブローカーに試行するすべての接続に適用されます。

retries

型： int
デフォルト： 2147483647
有効値： [0,...,2147483647]

Importance: low
値をゼロに設定すると、クライアントは一時的なエラーで失敗したリクエストを再送信します。値をゼロまたは **MAX_VALUE** に設定し、対応する timeout パラメーターを使用してリクエストを再試行する期間を制御することが推奨されます。

retry.backoff.ms

型： long
デフォルト： 100
有効な値： [0,...]
Importance: low
失敗したリクエストを再試行するまで待機する時間。これにより、一部の障害シナリオにおいて、密接ループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

型： 文字列
デフォルト： /usr/bin/kinit
重要性： low
Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ： long
デフォルト： 60000
インポート：low
更新試行間のログインスレッドスリープ時間。

sasl.kerberos.ticket.renew.jitter

型：
デフォルト： 0.05
インポート：low
更新時間にランダムなジッターが追加されたパーセンテージ。

sasl.kerberos.ticket.renew.window.factor

型：
デフォルト： 0.8
インポート：low
ログインスレッドは、指定のウィンドウ係数からチケットの有効期限に達するまでスリープされます。この期間が経過するとチケットの更新が試行されます。

sasl.login.refresh.buffer.seconds

タイプ： 短い
デフォルト： 300
有効な値： [0,...,3600]
Importance: low
認証情報を更新するときの認証情報の有効期限（秒単位）。そうでなければ、更新がバッファー秒数よりも期限切れになり、可能な限り多くのバッファー時間を維持するよう更新が行われます。法人の値は 0 から 3600（1時間）で設定されています。値が指定されていない場合には、デフォルト値の 300（5 分）が使用されます。この値と sasl.login.refresh.min.period.seconds は、合計が認証情報の残りの有効期間を超えた場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

タイプ： 短い

デフォルト： 60
有効な値 [0,...,900]
Importance: low
ログイン更新スレッドで認証情報を更新するまで待つ必要最小限の時間（秒単位）。法人の値は 0 から 900（15 分）で設定されています。値が指定されていない場合には、デフォルト値の 60（1 分）が使用されます。この値と sasl.login.refresh.buffer.seconds は、その合計が認証情報の残りの有効期間を超える場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

型：
デフォルト： 0.8
有効な値 [0.5,...,1.0]
Importance: low
ログイン更新スレッドは、認証情報の有効期間に比べ、指定されたウィンドウ係数に達するまでスリープ状態になり、クレデンシャルの更新を試みます。法人の値は 0.5(50%)と 1.0(100%)の範囲になります。値が指定されていない場合には、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

型：
デフォルト： 0.05
有効な値 [0.0,...,0.25]
Importance: low
ログイン更新スレッドのスリープ時間に追加される認証情報の有効期間に対して、ランダムなジッターの最大量。法人の値は 0 から 0.25(25%)で囲まれています。値が指定されていない場合には、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers

型： 文字列
デフォルト： null
インポート： low
セキュリティーアルゴリズムを実装するプロバイダーを返す、設定可能な作成者クラスの一覧。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.cipher.suites

type: list
デフォルト： null
インポート： low
暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC、鍵交換アルゴリズムの名前付きの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.endpoint.identification.algorithm

型： 文字列
デフォルト： https
インポート： low
サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

型： クラス
デフォルト： null
のインポート： low
SSLEngine オブジェクトを提供するために、org.apache.kafka.common.security.auth.SslEngineFactory タイプのクラス。Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory.

ssl.keymanager.algorithm

型： 文字列
デフォルト： SunX509
インポート： low
SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

型： 文字列
デフォルト： null
インポート： low
SSL 暗号操作に使用する SecureRandom PRNG 実装。

ssl.trustmanager.algorithm

タイプ： 文字列
デフォルト： PKIX
インポート： low
SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

付録F KAFKA CONNECT 設定パラメーター

config.storage.topic

型： 文字列
のインポート： high
コネクター設定が保存される Kafka トピックの名前。

group.id

型： 文字列
のインポート： high
このワーカーが属する Connect クラスターグループを識別する一意の文字列。

key.converter

型： クラス
のインポート： high
Kafka Connect 形式と Kafka に書き込まれるシリアライズされた形式間の変換に使用されるコンバータークラス。これは、Kafka から書き込まれたメッセージのキーの形式を制御します。これは、コネクターがシリアル化形式と操作できるコネクターから独立しているためです。一般的なフォーマットの例には、JSON および Avro が含まれます。

offset.storage.topic

型： 文字列
のインポート： high
コネクターオフセットが保存される Kafka トピックの名前。

status.storage.topic

型： 文字列
のインポート： high
コネクターおよびタスクステータスが保存される Kafka トピックの名前。

value.converter

型： クラス
のインポート： high
Kafka Connect 形式と Kafka に書き込まれるシリアライズされた形式間の変換に使用されるコンバータークラス。これは、Kafka から書き込まれたメッセージの値の形式を制御します。これは、コネクターがシリアル化形式と操作できるコネクターから独立しているためです。一般的なフォーマットの例には、JSON および Avro が含まれます。

bootstrap.servers

type: list
デフォルト： localhost:9092
Importance: high
Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ブートストラップ用に指定されたすべてのサーバーに関係なく、サーバーの完全なセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2, ...** の形式にする必要があります。これらのサーバーは、クラスターの完全なメンバーシップを検出するために最初の接続に使用されるため（動的に変更される可能性がある）、この一覧にはサーバーの完全セットを含める必要はありません（サーバーがダウンした場合など）。

heartbeat.interval.ms

型： int
デフォルト： 3000（3 秒）
のイン**ポート：** high
Kafka のグループ管理機能の使用時に、ハートビートからグループコーディネーターにハートビートの間隔が想定される時間。ハートビートは、ワーカーのセッションがアクティブな状態を維持し、新規メンバーがグループに参加したり離れたりする際にリバランスを容易にするために使用されます。値は **session.timeout.ms** よりも低い値を設定する必要がありますが、通常はその値の 1/3 よりも高い値を設定する必要があります。通常のリバランスの予想される時間を制御するように調整することもできます。

rebalance.timeout.ms

型： int
デフォルト： 60000（1 分）
Importance: high
各ワーカーがグループに参加する最大許容期間を 1 度、リバランスを開始します。これは基本的に、保留中のデータをフラッシュしてオフセットをコミットするために必要な時間の制限です。タイムアウトを超えるとワーカーがグループから削除され、オフセットのコミットに失敗します。

session.timeout.ms

型： int
デフォルト： 10000（10 秒）
のイン**ポート：** high
ワーカーの失敗の検出に使用されるタイムアウト。ワーカーは定期的なハートビートを送信し、その liveness をブローカーに示します。このセッションタイムアウトの期限が切れる前にブローカーによってハートビートが受信されなかった場合、ブローカーはグループからワーカーを削除し、リバランスを開始します。この値は、**group.min.session.timeout.ms** および **group.max.session.timeout.ms** によってブローカー設定で設定されるため、許容範囲である必要があることに注意してください。

ssl.key.password

タイプ： password
デフォルト： null
イン**ポート：** high
キーストアファイルまたは 'ssl.keystore.key' に指定された PEM キーのパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

タイプ： password
デフォルト： null
イン**ポート：** high
'ssl.keystore.type' により指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書の一覧で PEM 形式のみをサポートします。

ssl.keystore.key

タイプ： password
デフォルト： null
イン**ポート：** high
'ssl.keystore.type' により指定された形式の秘密鍵。デフォルトの SSL エンジンファクトリーは、PKCS#8 鍵を持つ PEM 形式のみをサポートします。鍵を暗号化する場合は、'ssl.key.password' を使用して鍵のパスワードを指定する必要があります。

ssl.keystore.location

type: string
デフォルト： null
Importance: high
キーストアファイルの場所。これはクライアント用にオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

タイプ： password
デフォルト： null
インポート： high
キーストアファイルのストアパスワード。これはクライアントでオプションであり、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアパスワードは PEM 形式ではサポートされません。

ssl.truststore.certificates

タイプ： password
デフォルト： null
インポート： high
'ssl.truststore.type' によって指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書で PEM 形式のみをサポートします。

ssl.truststore.location

type: string
デフォルト： null
Importance: high
トラストストアファイルの場所。

ssl.truststore.password

タイプ： password
デフォルト： null
インポート： high
トラストストアファイルのパスワード。パスワードが設定されていないと、設定したトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアパスワードは PEM 形式ではサポートされません。

client.dns.lookup

type: string
デフォルト: use_all_dns_ips
有効な値： [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]
Importance: medium
クライアントが DNS ルックアップをどのように使用するかを制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスに接続します。切断後、次の IP が使用されます。すべての IP を一度使用した後、クライアントはホスト名から IP を再度解決します（JVM と OS キャッシュ DNS 名の検索の両方）。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズ後、これは **use_all_dns_ips** と同じように動作します。**default**（非推奨）に設定すると、ルックアップが複数の IP アドレスを返しても、ルックアップが返した最初の IP アドレスへの接続を試行します。

connections.max.idle.ms

タイプ： long
デフォルト： 540000（9 分）
Importance: medium
この設定によって指定されるミリ秒数の後にアイドル状態の接続を閉じます。

connector.client.config.override.policy

型： 文字列
デフォルト： なし
インポート： medium
ConnectorClientConfigOverridePolicy の実装のクラス名またはエイリアス。コネクターによって上書きできるクライアント設定を定義します。デフォルトの実装は **None** です。フレームワークの他のポリシーには **All** および **Principal** が含まれます。

receive.buffer.bytes

型： int
デフォルト： 32768(32 kibibytes)
有効な値 [0,...]
Importance: medium
データの読み取り時に使用する TCP 受信バッファー(SO_RCVBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

型： int
デフォルト： 40000（40 秒）
有効な値： [0,...]
Importance: medium
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。

sasl.client.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

タイプ： password
デフォルト： null
のインポート： medium
JAAS 設定ファイルで使用される形式で SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式が ここ に記載されています。値の形式は以下のとおりです。「loginModuleClass controlFlag (optionName=optionValue)*;」.ブローカーの場合、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。例：
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string
デフォルト： null
Importance: medium

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定で定義できます。

sasl.login.callback.handler.class

型： クラス
デフォルト： null
Importance: medium
AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラーの設定には、小文字でリスナーのプレフィックスおよび SASL メカニズム名を付ける必要があります。たとえば、listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler です。

sasl.login.class

型： クラス
デフォルト： null
Importance: medium
ログインインターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナー接頭辞と SASL メカニズム名を付ける必要があります。For example, listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.mechanism

type: string
デフォルト： GSSAPI
Importance: medium
クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーを利用できるメカニズムがある場合があります。GSSAPI はデフォルトのメカニズムです。

security.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
ブローカーと通信するために使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

型： int
デフォルト： 131072(128 kibibytes)
有効な値 [0,...]
Importance: medium
データ送信時に使用する TCP 送信バッファー(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

ssl.enabled.protocols

type: list
デフォルト： TLSv1.2,TLSv1.3
Importance: medium
SSL 接続に対して有効なプロトコル一覧。Java 11 以降、TLSv1.2' の場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーは、TLSv1.2 へのサポートとフォールバックの両方をサポートする場合は、TLSv1.3 が優先されます（両方が TLSv1.2 をサポートしていることを前提とします）。ほとんどの場合、このデフォルト設定は問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

型： string
デフォルト： JKS
Importance: medium
キーストアファイルのファイル形式。これはクライアントでオプションです。

ssl.protocol

型： 文字列
デフォルト： TLSv1.3
のイン**ポート：** medium
SSLContext の生成に使用される SSL プロトコル。Java 11 以降、'TLSv1.2' の場合、デフォルトは 'TLSv1.3' です。この値は、ほとんどのユースケースで問題ありません。最新の JVM の許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされていますが、既知のセキュリティー脆弱性により使用は推奨されません。この設定と 'ssl.enabled.protocols' のデフォルト値で、サーバーが 'TLSv1.3' に対応していない場合は、クライアントは 'TLSv1.2' にダウングレードします。この設定を 'TLSv1.2' に設定すると、ssl.enabled.protocols の値のいずれかであった場合でも、クライアントは 'TLSv1.3' を使用しません。また、サーバーは 'TLSv1.3' のみをサポートします。

ssl.provider

type: string
デフォルト： null
Importance: medium
SSL 接続に使用されるセキュリティープロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティープロバイダーです。

ssl.truststore.type

型： string
デフォルト： JKS
Importance: medium
トラストストアファイルのファイル形式。

worker.sync.timeout.ms

型： int
デフォルト： 3000（3 秒）
のイン**ポート：** medium
ワーカーが他のワーカーと同期し、再同期する必要がある場合、グループの追加、参加前のバックオフ期間を待機した後に、この待機時間待機します。

worker.unsync.backoff.ms

型： int
デフォルト： 300000（5 分）
Importance: medium
ワーカーが他のワーカーと同期し、worker.sync.timeout.ms 内でキャッチできない場合、再参加する前にこの期間に Connect クラスターのままにします。

access.control.allow.methods

型： 文字列
デフォルト： ""
Importance: low

Access-Control-Allow-Methods ヘッダーを設定して、クロスオリジン要求でサポートされるメソッドを設定します。Access-Control-Allow-Methods ヘッダーのデフォルト値により、GET、POST、および HEAD のクロスオリジン要求が許可されます。

access.control.allow.origin

型： 文字列
デフォルト： ""
Importance: low
Access-Control-Allow-Origin ヘッダーを REST API 要求用に設定するための値。クロスオリジンアクセスを有効にするには、API へのアクセスが許可されるアプリケーションのドメインに設定する必要があります。あるいは、'*' を設定してすべてのドメインからのアクセスを許可する必要があります。デフォルト値は REST API のドメインからのアクセスのみを許可します。

admin.listeners

type: list
Default: null
Valid Values: org.apache.kafka.connect.runtime.WorkerConfig$AdminListenersValidator@6e3c1e69
Importance: low
Admin REST API がリッスンするカンマ区切りの URI の一覧。サポートされるプロトコルは HTTP および HTTPS です。空の文字列または空の文字列を使用すると、この機能が無効になります。デフォルトの動作では、通常のリスナーを使用します（listeners' プロパティーによって指定されます）。

client.id

型： 文字列
デフォルト： ""
Importance: low
要求の実行時にサーバーに渡す id 文字列。これの目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/port 以外の要求のソースを追跡できます。

config.providers

タイプ： list
デフォルト： ""
Importance: low
指定された順序でロードされ、使用される **ConfigProvider** クラスのコンマ区切りリスト。インターフェース **ConfigProvider** を実装すると、外部化シークレットなど、コネクター設定の変数参照を置き換えることができます。

config.storage.replication.factor

タイプ： 短い
デフォルト： 3
有効な値： Kafka クラスターのブローカー数よりも大きいでない数字、または -1（ブローカーのデフォルト
インポート）: low
設定ストレージトピックの作成時に使用されるレプリケーション係数。

connect.protocol

型： 文字列
デフォルト： sessioned
有効な値： [eager, compatible, sessioned]
Importance: low
Kafka Connect プロトコルの互換モード。

header.converter

type: class
デフォルト： org.apache.kafka.connect.storage.SimpleHeaderConverter
Importance: low
Kafka Connect 形式と Kafka に書き込まれたシリアライズ形式間の変換に使用される HeaderConverter クラス。これにより、Kafka から書き込まれたメッセージのヘッダー値の形式を制御します。これは、コネクターがシリアル化形式と操作できるコネクターから独立しているためです。一般的なフォーマットの例には、JSON および Avro が含まれます。デフォルトでは、SimpleHeaderConverter は、ヘッダー値を文字列にシリアライズし、スキーマを推測してデシリアライズするために使用されます。

inter.worker.key.generation.algorithm

型： 文字列
デフォルト： HmacSHA256
有効な値： ワーカー JVM
インポートでサポートされるキー生成**アルゴリズム：** low
内部要求キーを生成するために使用するアルゴリズム。

inter.worker.key.size

型： int
デフォルト： null
イン**ポート：** low
内部要求の署名に使用するキーのサイズ（ビット単位）。null の場合、キー生成アルゴリズムのデフォルトの鍵サイズが使用されます。

inter.worker.key.ttl.ms

型： int
デフォルト： 3600000（1 時間）
有効な値： [0,...,2147483647]
Importance: low
内部リクエスト検証に使用される生成されたセッションキーの TTL（ミリ秒単位）。

inter.worker.signature.algorithm

型： 文字列
デフォルト： HmacSHA256
有効な値： ワーカー JVM
のインポートにサポートされる任意の MAC**アルゴリズム：** low
内部リクエストの署名に使用するアルゴリズム。

inter.worker.verification.algorithms

type: list
デフォルト： HmacSHA256
有効な値： 1 つ以上の MAC アルゴリズムのリスト。ワーカー JVM
インポートシステムでサポートされる各 MAC アルゴリズムの**リスト：** low
内部要求を検証するための許可されるアルゴリズムのリスト。

internal.key.converter

型： クラス
デフォルト： org.apache.kafka.connect.json.JsonConverter
Importance: low
Kafka Connect 形式と Kafka に書き込まれるシリアライズされた形式間の変換に使用されるコン

バータークラス。これは、Kafka から書き込まれたメッセージのキーの形式を制御します。これは、コネクターがシリアル化形式と操作できるコネクターから独立しているためです。一般的なフォーマットの例には、JSON および Avro が含まれます。この設定は、設定やオフセットなどのフレームワークによって使用される内部ブック管理データに使用される形式を制御します。そのため、ユーザーは通常、正常に機能するコンバーターの実装を使用できます。非推奨。今後のバージョンで削除される予定です。

internal.value.converter

型： クラス
デフォルト： org.apache.kafka.connect.json.JsonConverter
Importance: low
Kafka Connect 形式と Kafka に書き込まれるシリアライズされた形式間の変換に使用されるコンバータークラス。これは、Kafka から書き込まれたメッセージの値の形式を制御します。これは、コネクターがシリアル化形式と操作できるコネクターから独立しているためです。一般的なフォーマットの例には、JSON および Avro が含まれます。この設定は、設定やオフセットなどのフレームワークによって使用される内部ブック管理データに使用される形式を制御します。そのため、ユーザーは通常、正常に機能するコンバーターの実装を使用できます。非推奨。今後のバージョンで削除される予定です。

listeners

type: list
デフォルト： null
インポート： low
REST API がリッスンするカンマ区切りの URI の一覧。サポートされるプロトコルは HTTP および HTTPS です。ホスト名を 0.0.0.0 として指定して、全インターフェースにバインドします。デフォルトのインターフェースにバインドするには、hostname は空のままにします。法的リスナー一覧の例： HTTP://myhost:8083,HTTPS://myhost:8084

metadata.max.age.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: low
パーティションリーダーを変更して、新しいブローカーまたはパーティションを正常に検出していない場合でも、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

タイプ： list
デフォルト： ""
Importance: low
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
メトリックの計算に維持されるサンプル数。

metrics.recording.level

タイプ： string
デフォルト： INFO
有効な値： [INFO, DEBUG]
Importance: low
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: low
メトリクスサンプルが計算される期間。

offset.flush.interval.ms

タイプ： long
デフォルト： 60000（1 分）
Importance: low
タスクのオフセットのコミットを試行する間隔。

offset.flush.timeout.ms

タイプ： long
デフォルト： 5000（5 秒）
のイン**ポート：** low
レコードをオフセットストレージにフラッシュおよびパーティションオフセットデータをコミットするまでの最大期間（ミリ秒単位）。プロセスをキャンセルして、将来の試行でコミットされたオフセットデータを復元します。

offset.storage.partitions

型： int
デフォルト： 25
有効な値： ポジティブ**な** 番号、またはブローカーのデフォルト
インポート製品を使用するよう -1
オフセットストレージトピックの作成時に使用されるパーティションの数。

offset.storage.replication.factor

タイプ： 短い
デフォルト： 3
有効な値： Kafka クラスターのブローカー数よりも大きいでない数字、または -1（ブローカーのデフォルト
インポート）: low
オフセットストレージトピックの作成時に使用されるレプリケーション係数。

plugin.path

type: list
デフォルト： null
イン**ポート：** low
プラグイン（connectors、コンバーター、変換）を含むコンマ(,)で区切られたパスの一覧。このリストは、JAR とプラグインの組み合わせが即座に含まれるトップレベルのディレクトリーで構成される必要があります。1 つのディレクトリーには、プラグインとその依存関係 b）の uber-jars、プラグインとその依存関係の c）ディレクトリーが含まれ、その依存関係のパッケージディレクトリー構

造が含まれます。シンボリックリンクは、依存関係またはプラグインを検出します。例：plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors Do はこのプロパティーの設定プロバイダー変数を使用しません。未加工パスは、設定変更を初期化して変数を置き換える前にワーカーのスキャナーによって使用されます。

reconnect.backoff.max.ms

型： long
デフォルト： 1000（1秒）
有効な値： [0,...]
重要： low
接続に繰り返し失敗したブローカーに再接続するまで待つ最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは、連続した接続失敗ごとに指数関数的に増加します。この最大値までです。バックオフの増加を算出したら、接続の量を回避するために 20% random jitter が追加されます。

reconnect.backoff.ms

型： long
デフォルト： 50
有効な値： [0,...]
Importance: low
指定のホストに再接続を試みる前に待機する時間のベース時間。これにより、密接ループでホストに繰り返し接続できなくなります。このバックオフは、クライアントがブローカーに試行するすべての接続に適用されます。

response.http.headers.config

type: string
Default: ""
Valid Values: Comma-separated header rules: 各ヘッダールールは '[action] [header name]:[header value]' の形式で、任意でヘッダールールの部分にコンマ
Importance（コンマ Importance）が含まれる場合に、任意で二重引用符で囲まれます。
REST API HTTP 応答ヘッダーのルール。

rest.advertised.host.name

型： 文字列
デフォルト： null
インポート： low
これが設定されている場合、これは他のワーカーに接続して接続するホスト名になります。

rest.advertised.listener

型： 文字列
デフォルト： null
インポート： low
他のワーカーが使用するワーカーに指定されるアドバタイズされたリスナー（HTTP または HTTPS）を設定します。

rest.advertised.port

型： int
デフォルト： null
インポート： low
これが設定されている場合、これは他のワーカーに接続して接続するポートになります。

rest.extension.classes

タイプ： list
デフォルト： ""
Importance: low
指定された順序でロードされ、**ConnectRestExtension** クラスのコンマ区切りリスト。インターフェース **ConnectRestExtension** を実装すると、フィルターなどの Connect の REST API ユーザー定義リソースに注入することができます。通常、ロギング、セキュリティーなどのカスタム機能を追加するために使用されます。

rest.host.name

型： 文字列
デフォルト： null
インポート： low
REST API のホスト名。これが設定されている場合、このインターフェースのみにバインドされます。

rest.port

型： int
デフォルト： 8083
インポート： low
リッスンする REST API のポート。

retry.backoff.ms

型： long
デフォルト： 100
有効な値： [0,...]
Importance: low
失敗したリクエストを特定のトピックパーティションに再試行するまで待つ期間。これにより、一部の障害シナリオにおいて、密接ループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

型： 文字列
デフォルト： /usr/bin/kinit
重要性： low
Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ： long
デフォルト： 60000
インポート： low
更新試行間のログインスレッドスリープ時間。

sasl.kerberos.ticket.renew.jitter

型：
デフォルト： 0.05
インポート： low
更新時間にランダムなジッターが追加されたパーセンテージ。

sasl.kerberos.ticket.renew.window.factor

型：
デフォルト： 0.8
インポート： low

ログインスレッドは、指定のウィンドウ係数からチケットの有効期限に達するまでスリープされます。この期間が経過するとチケットの更新が試行されます。

sasl.login.refresh.buffer.seconds

タイプ： 短い
デフォルト： 300
有効な値： [0,...,3600]
Importance: low
認証情報を更新するときの認証情報の有効期限（秒単位）。そうでなければ、更新がバッファー秒数よりも期限切れになり、可能な限り多くのバッファー時間を維持するよう更新が行われます。法人の値は 0 から 3600（1 時間）で設定されています。値が指定されていない場合には、デフォルト値の 300（5 分）が使用されます。この値と sasl.login.refresh.min.period.seconds は、合計が認証情報の残りの有効期間を超えた場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

タイプ： 短い
デフォルト： 60
有効な値 [0,...,900]
Importance: low
ログイン更新スレッドで認証情報を更新するまで待つ必要最小限の時間（秒単位）。法人の値は 0 から 900（15 分）で設定されています。値が指定されていない場合には、デフォルト値の 60（1 分）が使用されます。この値と sasl.login.refresh.buffer.seconds は、その合計が認証情報の残りの有効期間を超える場合に無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

型：
デフォルト： 0.8
有効な値 [0.5,...,1.0]
Importance: low
ログイン更新スレッドは、認証情報の有効期間に比べ、指定されたウィンドウ係数に達するまでスリープ状態になり、クレデンシャルの更新を試みます。法人の値は 0.5(50%)と 1.0(100%)の範囲になります。値が指定されていない場合には、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

型：
デフォルト： 0.05
有効な値 [0.0,...,0.25]
Importance: low
ログイン更新スレッドのスリープ時間に追加される認証情報の有効期間に対して、ランダムなジッターの最大量。法人の値は 0 から 0.25(25%)で囲まれています。値が指定されていない場合には、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

scheduled.rebalance.max.delay.ms

型： int
デフォルト： 300000（5 分）
有効な値 [0,...,2147483647]
Importance: low
コネクターおよびタスクをグループに再割り当てし、1 つ以上の departed ワーカーを返すまで待機するためにスケジュールされる最大遅延。この期間中、部署されたワーカーのコネクターおよびタスクは割り当てられません。

socket.connection.setup.timeout.max.ms

型： long
デフォルト： 127000（127 秒）
有効な値： [0,...]
Importance: low
ソケット接続が確立されるまでクライアントが待機する最大時間。接続セットアップのタイムアウトは、この最大値までの連続する接続障害ごとに指数関数的に増加します。接続ストームを回避するために、0.2 のランダム化係数はタイムアウトに適用されます。これにより、計算値の 20% 未満の 20% の範囲がランダムな範囲になります。

socket.connection.setup.timeout.ms

型： long
デフォルト： 10000（10 秒）
有効な値： [0,...]
インスケーション： low
クライアントがソケット接続を確立するのを待つ時間。タイムアウトが経過する前にコネクションがビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.cipher.suites

type: list
デフォルト： null
インポート：low
暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC、鍵交換アルゴリズムの名前付きの組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.client.auth

型： 文字列
デフォルト： none
インポート：low
Kafka ブローカーがクライアント認証を要求するように設定します。以下は一般的な設定です。

- **ssl.client.auth=required** 必要なクライアント認証に設定されていると、必要なクライアント認証が必要です。
- **ssl.client.auth=requested** そのため、このオプションが設定されているとクライアント自体の認証情報を提供しないことができない場合があるため、クライアントの認証は任意です。
- **ssl.client.auth=none** これは、クライアント認証が不要であることを意味します。

ssl.endpoint.identification.algorithm

型： 文字列
デフォルト： https
インポート：low
サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

型： クラス
デフォルト： null
のインポート：low

SSLEngine オブジェクトを提供するために、
org.apache.kafka.common.security.auth.SslEngineFactory タイプのクラス。Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory.

ssl.keymanager.algorithm

型： 文字列
デフォルト： SunX509
インポート**：** low
SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

型： 文字列
デフォルト： null
インポート**：** low
SSL 暗号操作に使用する SecureRandom PRNG 実装。

ssl.trustmanager.algorithm

タイプ： 文字列
デフォルト： PKIX
インポート**：** low
SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

status.storage.partitions

型： int
デフォルト： 5
有効な値： ポジティブな 番号、またはブローカーのデフォルト
インポート製品を使用するよう -1
ステータスストレージトピックの作成時に使用されるパーティションの数。

status.storage.replication.factor

タイプ： 短い
デフォルト： 3
有効な値： Kafka クラスターのブローカー数よりも大きいでない数字、または -1（ブローカーのデフォルト
インポート）: low
ステータスストレージトピックの作成時に使用されるレプリケーション係数。

task.shutdown.graceful.timeout.ms

タイプ： long
デフォルト： 5000（5 秒）
のインポート**：** low
タスクが正常にシャットダウンするのを待つ時間。これは、タスクごとにではなく、合計時間です。すべてのタスクがシャットダウンがトリガーされてから、順次待機します。

topic.creation.enable

型： ブール値
デフォルト： true
インポート**：** low

ソースコネクターと **topic.creation.** プロパティーが設定されたときに、ソースコネクターによって使用されるトピックの自動作成を許可するかどうか。各タスクは管理クライアントを使用してトピックを作成し、Kafka ブローカーに依存してトピックを自動的に作成します。

topic.tracking.allow.reset

型： ブール値
デフォルト： true
インポート： low
true に設定すると、ユーザーリクエストはコネクターごとにアクティブなトピックのセットをリセットできます。

topic.tracking.enable

型： ブール値
デフォルト： true
インポート： low
ランタイム中のコネクターごとにアクティブなトピックのセットの追跡を有効にします。

付録G KAFKA STREAMS の設定パラメーター

application.id

型： 文字列
のインポート： high
ストリーム処理アプリケーションの識別子。Kafka クラスター内で一意である必要があります。これは、メンバーシップ管理用の group-id のプレフィックスである 1）として使用されます（例： 3）。

bootstrap.servers

型： list
のインポート： high
Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ブートストラップ用に指定されたすべてのサーバーに関係なく、サーバーの完全なセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2, ...** の形式にする必要があります。これらのサーバーは、クラスターの完全なメンバーシップを検出するために最初の接続に使用されるため（動的に変更される可能性がある）、この一覧にはサーバーの完全セットを含める必要はありません（サーバーがダウンした場合など）。

replication.factor

型： int
デフォルト： 1
Importance: high
ログトピックのレプリケーション係数。また、ストリーム処理アプリケーションによって作成されたトピックを再パーティションします。

state.dir

型： 文字列
デフォルト： /tmp/kafka-streams
Importance: high
ステートストアのディレクトリーの場所。このパスは、同じ基礎となるファイルシステムを共有する各ストリームインスタンスに対して一意である必要があります。

acceptable.recovery.lag

タイプ： long
デフォルト： 10000
有効な値： [0,...]
Importance: medium
クライアントに対する最大許容ラグ（取得するオフセットの数）は、アクティブなタスクのキャッチされたと見なされる最大許容ラグ（取得するオフセットの数）です。指定のワークロードの分数に適切に対応します。0 以上でなければなりません。

cache.max.bytes.buffering

型： long
デフォルト： 10485760
有効な値： [0,...]
インポート**中：** medium
すべてのスレッド全体でのバッファーに使用される最大メモリーバイト数。

client.id

型： 文字列
デフォルト： ""
Importance: medium
内部コンシューマー、プロデューサー、および restore-consumer のクライアント ID に使用される ID プレフィックスの文字列（パターン '<client.id>-StreamThread-<threadSequenceNumber>-<consumer|producer|restore-consumer> '）。

default.deserialization.exception.handler

型： クラス
デフォルト： org.apache.kafka.streams.errors.LogAndFailExceptionHandler
Importance: medium
org.apache.kafka.streams.errors.DeserializationExceptionHandler インターフェースを実装するクラスを処理する例外

default.key.serde

Type: class
デフォルト： org.apache.kafka.common.serialization.Serdes$ByteArraySerde
Importance: medium
org.apache.kafka.common.serialization.Serde インターフェースを実装するキーのデフォルトのシリアライザー/デシリアライザークラス。windowed serde クラスが使用される場合、'default.windowed.key.serde.inner' または 'default.windowed.value.serdener' で **org.apache.kafka.common.serialization.Serde** インターフェースを実装する内部サンドクラスを 1 つ設定する必要があります。

default.production.exception.handler

型： クラス
デフォルト： org.apache.kafka.streams.errors.Default ProductionExceptionHandler
Importance: medium
org.apache.kafka.streams.errors.ProductionExceptionHandler インターフェースを実装するクラスを処理する例外

default.timestamp.extractor

Type: class
デフォルト： org.apache.kafka.streams.processor.FailOnInvalidTimestamp
Importance: medium
org.apache.kafka.streams.processor.TimestampExtractor インターフェースを実装するデフォルトのタイムスタンプ抽出クラス。

default.value.serde

Type: class
デフォルト： org.apache.kafka.common.serialization.Serdes$ByteArraySerde
Importance: medium
org.apache.kafka.common.serialization.Serde インターフェースを実装する値のデフォルトのシリアライザー/デシリアライザークラス。windowed serde クラスが使用される場合、'default.windowed.key.serde.inner' または 'default.windowed.value.serdener' で **org.apache.kafka.common.serialization.Serde** インターフェースを実装する内部サンドクラスを 1 つ設定する必要があります。

default.windowed.key.serde.inner

型： クラス
デフォルト： null
Importance: medium

ウィンドウされたキーの内部クラスのデフォルトのシリアライザー / デシリアライザー / デシリアライザー。**org.apache.kafka.common.serialization.Serde** インターフェースを実装する必要があります。

default.windowed.value.serde.inner

型： クラス
デフォルト： null
Importance: medium
ウィンドウ化された値の内部クラスのデフォルトシリアライザー / デシリアライザー。**org.apache.kafka.common.serialization.Serde** インターフェースを実装する必要があります。

max.task.idle.ms

タイプ： long
デフォルト： 0
Importance: medium
複数の入力ストリーム間で out-of-order レコード処理を防ぐために、すべてのパーティションバッファーにレコードが含まれる場合にストリームタスクがアイドル状態のままになります（ミリ秒単位）。

max.warmup.replicas

型： int
デフォルト： 2
有効な値 [1,...]
Importance: medium
ウォームアップレプリカの最大数（設定された num.standbys 以外）は、1 つのインスタンスでタスクを利用できるようにするために、一度に割り当てることができます。ただし、別のインスタンスが再割り当てされる間、このレプリカは 1 つのインスタンスで利用できるようにすることができます。高可用性に使用できるブローカートラフィックとクラスターの状態の量を調整するために使用されます。1 以上でなければなりません。

num.standby.replicas

型： int
デフォルト： 0
Importance: medium
各タスクのスタンバイレプリカ数。

num.stream.threads

型： int
デフォルト： 1
Importance: medium
ストリーム処理を実行するスレッドの数。

processing.guarantee

型： 文字列
デフォルト： at_least_once
有効な値： [at_least_once, exactly_once_beta]
Importance: medium
処理は使用すべきことを保証します。使用できる値は **at_least_once**（デフォルト）、**exactly_once**（ブローカーバージョン 0.11.0 以降が必要）、および **exactly_once_beta**（ブローカーバージョン 2.5 以降が必要）です。1 回だけ処理するには、実稼働に推奨される設定の

3 つ以上のブローカーで構成されるクラスターが必要になります。開発の場合は、ブローカー **transaction.state.log.replication.factor** および **transaction.state.log.min.isr** を調整することで、これを変更することができます。

security.protocol

type: string
デフォルト： PLAINTEXT
Importance: medium
ブローカーと通信するために使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

task.timeout.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: medium
内部エラーが原因でタスクが停止する可能性があり、エラーが発生するまで再試行される可能性がある最大時間（ミリ秒単位）。0ms のタイムアウトの場合は、タスクにより最初の内部エラーのエラーが発生します。0ms を超えるタイムアウトの場合、タスクはエラーが発生する前に少なくとも 1 回リトライされます。

topology.optimization

型： 文字列
デフォルト： none
有効な値： [none, all]
Importance: medium
トポロジーを最適化する必要がある場合は、Kafka Streams に指示する設定（デフォルトで無効）。

application.server

型： 文字列
デフォルト： ""
Importance: low
ユーザー定義のエンドポイントを参照する host:port ペア。これは、ステートストア検出およびこの KafkaStreams インスタンスでの対話的なクエリーに使用できます。

buffered.records.per.partition

型： int
デフォルト： 1000
インポート： low
パーティションごとにバッファーするレコードの最大数。

built.in.metrics.version

型： 文字列
デフォルト： 最新の
有効な値 [0.10.0-2.4, latest]
重要性： low
使用するビルトインメトリクスのバージョン。

commit.interval.ms

型： long
デフォルト： 30000（30 秒）

有効な値： [0,...]
Importance: low
プロセッサーの位置を保存する頻度（ミリ秒単位）。（注記： **processing.guarantee** が **exactly_once** に設定された場合、デフォルト値は **100** です。それ以外の場合は、デフォルト値は **30000** です。

connections.max.idle.ms

タイプ： long
デフォルト： 540000（9 分）
インポート**中：** low
この設定によって指定されるミリ秒数の後にアイドル状態の接続を閉じます。

metadata.max.age.ms

タイプ： long
デフォルト： 300000（5 分）
有効な値： [0,...]
Importance: low
パーティションリーダーを変更して、新しいブローカーまたはパーティションを正常に検出していない場合でも、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

タイプ： list
デフォルト： ""
Importance: low
メトリクスレポーターとして使用するクラスの一覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知を受信するクラスにプラグインすることができます。JmxReporter は JMX 統計を登録するために常に含まれます。

metrics.num.samples

型： int
デフォルト： 2
有効な値 [1,...]
Importance: low
メトリックの計算に維持されるサンプル数。

metrics.recording.level

タイプ： string
デフォルト： INFO
有効な値： [INFO、DEBUG, TRACE]
重要度： low
メトリクスの記録レベル。

metrics.sample.window.ms

型： long
デフォルト： 30000（30 秒）
有効な値： [0,...]
Importance: low
メトリクスサンプルが計算される期間。

partition.grouper

型： クラス
デフォルト： org.apache.kafka.streams.processor.DefaultPartitionGrouper
Importance: low
org.apache.kafka.streams.processor.PartitionGrouper インターフェースを実装するパーティションググループクラス。警告： この設定は非推奨となり、3.0.0 リリースで削除されます。

poll.ms

タイプ： long
デフォルト： 100
インポート： low
入力待ちのブロックする期間（ミリ秒単位）。

probing.rebalance.interval.ms

型： long
デフォルト： 600000（10 分）
有効な値： [60000,...]
Importance: low
ウェイム完了およびアクティブになる準備ができたレプリカをプローブするためにリバランスをトリガーするまで待機する時間（ミリ秒単位）。リバランスの伝播は、割り当てがバランスになるまで引き続きトリガーされます。1 分以上にしてください。

receive.buffer.bytes

型： int
デフォルト： 32768(32 kibibytes)
有効な値 [-1,...]
のインポート： low
データの読み取り時に使用する TCP 受信バッファー(SO_RCVBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

reconnect.backoff.max.ms

型： long
デフォルト： 1000（1 秒）
有効な値： [0,...]
重要： low
接続に繰り返し失敗したブローカーに再接続するまで待つ最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは、連続した接続失敗ごとに指数関数的に増加します。この最大値までです。バックオフの増加を算出したら、接続の量を回避するために 20% random jitter が追加されます。

reconnect.backoff.ms

型： long
デフォルト： 50
有効な値： [0,...]
Importance: low
指定のホストに再接続を試みる前に待機する時間のベース時間。これにより、密接ループでホストに繰り返し接続できなくなります。このバックオフは、クライアントがブローカーに試行するすべての接続に適用されます。

request.timeout.ms

型： int
デフォルト： 40000（40 秒）
有効な値： [0,...]

Importance: low
設定は、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前にレスポンスが受信されなかった場合、再試行が行われるとクライアントはリクエストを再送信します。

retries

型： int
デフォルト： 0
有効な値： [0,...,2147483647]
Importance: low
値をゼロに設定すると、クライアントは一時的なエラーで失敗したリクエストを再送信します。値をゼロまたは **MAX_VALUE** に設定し、対応する timeout パラメーターを使用してリクエストを再試行する期間を制御することが推奨されます。

retry.backoff.ms

型： long
デフォルト： 100
有効な値： [0,...]
Importance: low
失敗したリクエストを特定のトピックパーティションに再試行するまで待つ期間。これにより、一部の障害シナリオにおいて、密接ループでリクエストを繰り返し送信しないようにします。

rocksdb.config.setter

型： クラス
デフォルト： null
のイン**ポート：** low
org.apache.kafka.streams.state.RocksDBConfigSetter インターフェースを実装する Rocks DB 設定セッタークラスまたはクラス名。

send.buffer.bytes

型： int
デフォルト： 131072(128 kibibytes)
有効な値 [-1,...]
Importance: low
データ送信時に使用する TCP 送信バッファー(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

state.cleanup.delay.ms

タイプ： long
デフォルト： 600000（10 分）
Importance: low
パーティションが移行されたときに削除を待機する期間（ミリ秒単位）。少なくとも **state.cleanup.delay.ms** に対して変更されていない状態ディレクトリーのみが削除されます。

upgrade.from

型： 文字列
デフォルト： null
有効な値 [null、0.10.0、0.10.1、0.10.2、0.11.0、1.0、1.1、2.1、2.3、2.3、2.3、2.3、および 2.3]
のインポート中**:** low
後方互換性がある方法でのアップグレードを許可します。これは、[0.10.0、1.1] から 2.0+、または [2.0, 2.3] から 2.4+ にアップグレードする際に必要です。2.4 から新しいバージョンにアップグレードする場合は、この設定を指定する必要はありません。デフォルトは **null** です。許可される値は、

"0.10.0"、"0.10.1"、"0.10.2"、"0.11.0"、"1.0"、"1.1"、"2.2"、"2.2"、"2.3"（対応する古いバージョンからアップグレードする場合）です。

windowstore.changelog.additional.retention.ms

タイプ： long
デフォルト： 86400000（1 日）
Importance: low
画面の maintainMs に追加され、データがログの途中から削除されないようにします。クロックのドリフトを許可します。デフォルトは 1 日です。

付録H サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **JBOSS INTEGRATION AND AUTOMATION**カテゴリーの **Red Hat AMQ Streams**エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

パッケージ用のシステムの登録

Red Hat Enterprise Linux に RPM パッケージをインストールするには、システムを登録する必要があります。zip または tar ファイルを使用している場合は、このステップは必要ありません。

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。
3. OS のバージョンを選択し、次のページに進みます。
4. システムターミナルで listed コマンドを使用して、登録を完了します。

詳細は 「Red Hat カスタマーポータルへのシステム登録およびサブスクライブ方法」を 参照してください。

改訂日時： 2021-06-17 12:41:57 +1000