Red Hat Streams for Apache Kafka 2.7

KRaft モードの RHEL での Streams for Apache Kafka の使用

Red Hat Enterprise Linux での Streams for Apache Kafka 2.7 のデプロイメントの設定および管理

Last Updated: 2024-05-29

Red Hat Streams for Apache Kafka 2.7 KRaft モードの RHEL での Streams for Apache Kafka の使用

Red Hat Enterprise Linux での Streams for Apache Kafka 2.7 のデプロイメントの設定および管理

法律上の通知

概要

Streams for Apache Kafka でデプロイされた Operator および Kafka コンポーネントを設定し、大規模なメッセージングネットワークを構築します。

目次

はじめに 5

RED HAT ドキュメントへのフィードバック (英語のみ) 6

第1章 STREAMS FOR APACHE KAFKA の概要 7
1.1. KAFKA BRIDGE を使用した KAFKA クラスターへの接続 7
1.2. このドキュメントの表記慣例 8

第2章 FIPS サポート 9
2.1. FIPS モードを有効にして STREAMS FOR APACHE KAFKA のインストール 9

第3章 スタートガイド 11
3.1. インストール環境 11
3.2. STREAMS FOR APACHE KAFKA のダウンロード 12
3.3. KAFKA のインストール 12
3.4. KRAFT モードでの KAFKA クラスターの実行 13
3.5. STREAMS FOR APACHE KAFKA サービスの停止 16
3.6. KAFKA ブローカーの正常なローリング再起動の実行 16

第4章 KRAFT モードへの移行 19

第5章 STREAMS FOR APACHE KAFKA の設定 24
5.1. 標準の KAFKA 設定プロパティーの使用 24
5.2. 環境変数から設定値の読み込み 24
5.3. KAFKA の設定 25

第6章 KAFKA へのアクセスのセキュア化 29
6.1. リスナーの設定 29
6.2. TLS 暗号化 30
6.3. 認証 31
6.4. 認可 41
6.5. OAUTH 2.0 トークンベース認証の使用 44
6.6. OAUTH 2.0 トークンベース承認の使用 72
6.7. OPA ポリシーベースの承認の使用 76

第7章 トピックの作成および管理 79
7.1. パーティションおよびレプリカ 79
7.2. メッセージの保持 79
7.3. トピックの自動作成 80
7.4. トピックの削除 80
7.5. トピックの設定 80
7.6. 内部トピック 81
7.7. トピックの作成 82
7.8. トピックの一覧表示および説明 83
7.9. トピック設定の変更 83
7.10. トピックの削除 84

第8章 KAFKA CONNECT での STREAMS FOR APACHE KAFKA の使用 86
8.1. スタンドアロンモードでの KAFKA CONNECT の使用 86
8.2. 分散モードでの KAFKA CONNECT の使用 87
8.3. コネクターの管理 89

第9章 MIRRORMAKER 2 での STREAMS FOR APACHE KAFKA の使用 95
9.1. ACTIVE/ACTIVE または ACTIVE/PASSIVEモードの設定 95

9.2. MIRRORMAKER 2 コネクターの設定 96
9.3. コネクタープロデューサーおよびコンシューマーの設定 103
9.4. タスクの最大数を指定 104
9.5. ACL ルールの同期 104
9.6. MIRRORMAKER 2 を専用モードで実行する 104
9.7. (非推奨) レガシーモードでの MIRRORMAKER 2 の使用 108

第10章 KAFKA コンポーネントのログの設定 110
10.1. KAFKA ログプロパティーの設定 110
10.2. KAFKA ブローカーロガーのログレベルの動的な変更 110
10.3. KAFKA CONNECT と MIRRORMAKER 2 のログレベルを動的に変更する 112

第11章 KAFKA STATIC QUOTA プラグインを使用したブローカーへの制限の設定 115

第12章 ブローカーの追加または削除によるクラスターのスケーリング 117

第13章 CRUISE CONTROL を使用したクラスターのリバランス 118
13.1. CRUISE CONTROL のコンポーネントと機能 119
13.2. CRUISE CONTROL のダウンロード 120
13.3. CRUISE CONTROL METRICS REPORTER のデプロイ 120
13.4. CRUISE CONTROL の設定および起動 121
13.5. 最適化ゴールの概要 123
13.6. 最適化プロポーザルの概要 127
13.7. リバランスパフォーマンスチューニングの概要 131
13.8. CRUISE CONTROL の設定 135
13.9. 最適化プロポーザルの生成 137
13.10. 最適化プロポーザルの承認 141
13.11. アクティブなクラスターリバランスの停止 142

第14章 CRUISE CONTROL を使用したトピックレプリケーション係数の変更 144

第15章 パーティション再割り当てツールの使用 145
15.1. パーティション再割り当てツールの概要 145
15.2. ブローカーの追加後のパーティションの再割り当て 148
15.3. ブローカーの削除前のパーティションの再割り当て 150
15.4. トピックのレプリケーション係数の変更 153

第16章 分散トレースの設定 156
16.1. 手順の概要 156
16.2. トレースオプション 157
16.3. トレースの環境変数 158
16.4. KAFKA CONNECT のトレースの有効化 158
16.5. MIRRORMAKER 2 のトレースを有効にする 159
16.6. MIRRORMAKER のトレースの有効化 160
16.7. KAFKA クライアントのトレースの初期化 161
16.8. KAFKA プロデューサーおよびコンシューマーをトレース用にインストルメント化 162
16.9. KAFKA STREAMS アプリケーションのトレース用のインストルメント化 164
16.10. OPENTELEMETRY でのトレースシステムの指定 165
16.11. OPENTELEMETRY のカスタムスパン名の指定 166

第17章 KAFKA EXPORTER の使用 168
17.1. コンシューマーラグ 168
17.2. KAFKA EXPORTER アラートルールの例 169
17.3. KAFKA EXPORTER メトリクス 169
17.4. KAFKA EXPORTER の実行 170

17.5. GRAFANA での KAFKA EXPORTER メトリクスの表示 172

第18章 STREAMS FOR APACHE KAFKA および KAFKA のアップグレード 174
18.1. アップグレードの前提条件 174
18.2. クライアントをアップグレードするストラテジー 174
18.3. KAFKA クラスターのアップグレード 174
18.4. KAFKA コンポーネントのアップグレード 176

第19章 JMX を使用したクラスターの監視 179
19.1. JMX エージェントの有効化 179
19.2. JMX エージェントの無効化 179
19.3. メトリックの命名規則 180
19.4. トラブルシューティングのための KAFKA JMX メトリックの分析 181

付録A サブスクリプションの使用 188
アカウントへのアクセス 188
サブスクリプションのアクティベート 188
Zip および Tar ファイルのダウンロード 188
DNF を使用したパッケージのインストール 188

はじめに

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。

改善を提案するには、Jira 課題を作成し、変更案についてご説明ください。ご要望に迅速に対応できるよう、できるだけ詳細にご記入ください。

前提条件

- Red Hat カスタマーポータルのアカウントがある。このアカウントを使用すると、Red Hat Jira Software インスタンスにログインできます。
  アカウントをお持ちでない場合は、アカウントを作成するように求められます。

手順

1. 以下の Create issue をクリックします。
2. **Summary** テキストボックスに、問題の簡単な説明を入力します。
3. **Description** テキストボックスに、次の情報を入力します。
   - 問題が見つかったページの URL
   - 問題の詳細情報
     他のフィールドの情報はデフォルト値のままにすることができます。
4. レポーター名を追加します。
5. **Create** をクリックして、Jira 課題をドキュメントチームに送信します。

フィードバックの提供にご協力いただきありがとうございました。

第1章 STREAMS FOR APACHE KAFKA の概要

AMQ Streams は、Apache Kafka プロジェクトをベースとした、スケーラビリティの高い分散型の高性能データストリーミングをサポートします。

主なコンポーネントは以下で構成されます。

Kafka Broker

生成クライアントから消費側のクライアントにレコードを配信するメッセージングブローカー

Kafka Streams API

ストリームプロセッサー アプリケーションを作成するための 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

Streams for Apache Kafka 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 インターフェイスが提供されます。

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

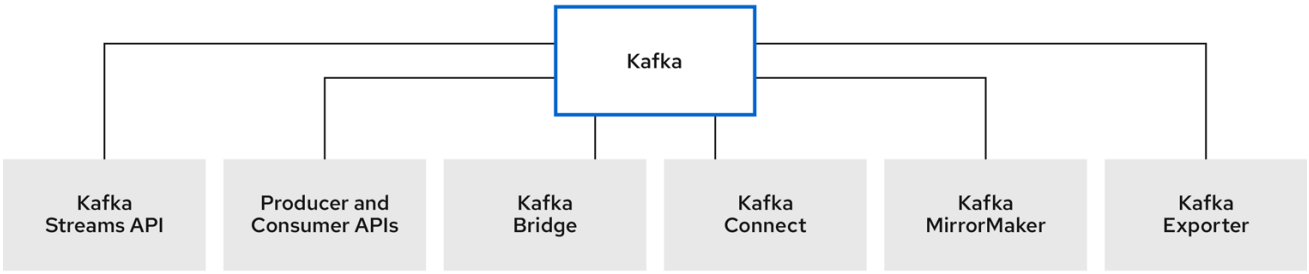
データセンター内またはデータセンター全体の 2 つの Kafka クラスター間でデータをレプリケーションする。

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 ブローカーのクラスターは、これらのすべてのコンポーネントを接続するハブです。

図1.1 Streams for Apache Kafka アーキテクチャー



1.1. KAFKA BRIDGE を使用した KAFKA クラスターへの接続

Streams for Apache Kafka Kafka Bridge API を使用して、コンシューマーを作成および管理し、ネイティブ Kafka プロトコルではなく HTTP を介してレコードを送受信できます。

Kafka Bridge を設定する場合、Kafka クラスターへの HTTP アクセスを設定します。その後、Kafka Bridge を使用して、クラスターからのメッセージを生成および消費したり、REST インターフェイスを介して他の操作を実行することができます。

関連情報

- Kafka Bridge のインストールおよび使用に関する詳細は、Streams for Apache Kafka Kafka Bridge の使用 を参照してください。

1.2. このドキュメントの表記慣例

ユーザー置換値

ユーザーが置き換える値は、**置き換え可能** な値とも呼ばれ、山かっこ (<>) を付けて表示されます。アンダースコア (_) は、複数単語の値に使用されます。値がコードまたはコマンドを参照する場合は **monospace** も使用されます。

たとえば、次のコードは、**<bootstrap_address>** と **<topic_name>** を独自のアドレスとトピック名に置き換える必要があることを示します。

```
bin/kafka-console-consumer.sh --bootstrap-server <broker_host>:<port> --topic <topic_name> --from-beginning
```

第2章 FIPS サポート

FIPS (Federal Information Processing Standards) は、コンピューターセキュリティーおよび相互運用性の標準です。Streams for Apache Kafka で FIPS を使用するには、FIPS 準拠の OpenJDK (Open Java Development Kit)がシステムにインストールされている必要があります。RHEL システムが FIPS 対応である場合、Stream for Apache Kafka の実行時に OpenJDK は自動的に FIPS モードに切り替わります。これにより、Streams for Apache Kafka は、OpenJDK が提供する FIPS 準拠のセキュリティーライブラリーを使用するようになります。

パスワードの最小長

FIPS モードで実行する場合、SCRAM-SHA-512 パスワードは 32 文字以上にする必要があります。32 文字未満のパスワード長を使用するカスタム設定の Kafka クラスターがある場合は、設定を更新する必要があります。32 文字未満のパスワードを持つユーザーがいる場合は、必要な長さのパスワードを再生成する必要があります。

関連情報

- Federal Information Processing Standards (FIPS) の概要

2.1. FIPS モードを有効にして STREAMS FOR APACHE KAFKA のインストール

Streams for Apache Kafka を RHEL にインストールする前に FIPS モードを有効にします。Red Hat は、後で FIPS モードを有効にするのではなく、FIPS モードを有効にして RHEL をインストールすることを推奨します。インストール時に FIPS モードを有効にすると、システムは FIPS で承認されるアルゴリズムと継続的な監視テストですべての鍵を生成するようになります。

RHEL を FIPS モードで実行する場合は、Streams for Apache Kafka 設定が FIPS に準拠していることを確認する必要があります。さらに、Java 実装も FIPS に準拠している必要があります。

注記

FIPS モードで RHEL で Streams for Apache Kafka を実行するには、FIPS 準拠の JDK が必要です。

手順

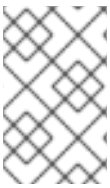
1. RHEL を FIPS モードでインストールします。
   詳細は、RHEL ドキュメント のセキュリティー強化に関する情報を参照してください。
2. Streams for Apache Kafka のインストールに進みます。
3. FIPS 準拠のアルゴリズムとプロトコルを使用するように Streams for Apache Kafka を設定します。
   使用する場合は、次の設定が準拠していることを確認してください。
   - SSL 暗号スイートと TLS バージョンは、JDK フレームワークでサポートされている必要があります。
   - SCRAM-SHA-512 パスワードの長さは少なくとも 32 文字である必要があります。

重要

FIPS 要件の変更に応じて、インストール環境と Streams for Apache Kafka 設定が準拠していることを確認してください。

第3章 スタートガイド

Streams for Apache Kafka は、Kafka コンポーネントのインストールアーティファクトが含まれる ZIP ファイルで配布されます。



注記

Kafka Bridge には個別のインストールファイルがあります。Kafka Bridge のインストールおよび使用に関する詳細は、Streams for Apache Kafka Kafka Bridge の使用 を参照してください。

3.1. インストール環境

Streams for Apache Kafka は Red Hat Enterprise Linux で実行されます。ホスト (ノード) は、物理マシンまたは仮想マシン (VM) にすることができます。Streams for Apache Kafka で提供されるインストールファイルを使用して、Kafka コンポーネントをインストールします。Kafka は、シングルノード環境または複数ノード環境にインストールできます。

シングルノード環境

シングルノード Kafka クラスターは、Kafka コンポーネントのインスタンスを単一のホスト上で実行します。この設定は、実稼働環境には適していません。

マルチノード環境

マルチノード Kafka クラスターは、Kafka コンポーネントのインスタンスを複数のホスト上で実行します。

Kafka と他の Kafka コンポーネント (Kafka Connect など) を別のホストで実行することを推奨します。この方法でコンポーネントを実行すると、各コンポーネントの保守とアップグレードが容易になります。

Kafka クライアントは、**bootstrap.servers** 設定プロパティーを使用して Kafka クラスターへの接続を確立します。たとえば、Kafka Connect を使用している場合、Kafka Connect 設定プロパティーには、Kafka ブローカーが実行されているホストのホスト名とポートを指定する **bootstrap.servers** 値が含まれている必要があります。Kafka クラスターが複数の Kafka ブローカーを持つ複数のホストで実行されている場合は、ブローカーごとにホスト名とポートを指定します。各 Kafka ブローカーは、**node.id** によって識別されます。

3.1.1. 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、Streams for Apache Kafka のパフォーマンスを最適化するために不可欠です。

ブロックストレージが必要です。NFS などのファイルストレージは、Kafka では機能しません。

ブロックストレージには、以下のいずれかのオプションを選択します。

- 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
- ローカルストレージ
- **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 SAN (ストレージエリアネットワーク) ボリューム

3.1.2. ファイルシステム

Kafka は、メッセージの保存にファイルシステムを使用します。Streams for Apache Kafka は、Kafka で一般的に使用される XFS および ext4 ファイルシステムと互換性があります。ファイルシステムを選択して設定するときは、デプロイメントの基盤となるアーキテクチャーと要件を考慮してください。

詳細については、Kafka ドキュメントの Filesystem Selection を参照してください。

3.2. STREAMS FOR APACHE KAFKA のダウンロード

Streams for Apache Kafka の ZIP ファイル配布は、Red Hat の Web サイトからダウンロードできます。Red Hat Streams for Apache Kafka の最新バージョンは、Streams for Apache Kafka software downloads ページ からダウンロードできます。

- Kafka およびその他の Kafka コンポーネントの場合は、**amq-streams-<version>-bin.zip** ファイルをダウンロードします。
- Kafka Bridge の場合は、**amq-streams-<version>-bridge-bin.zip** ファイルをダウンロードします。
  インストール手順は、Streams for Apache Kafka Kafka Bridge の使用 を 参照してください。

3.3. KAFKA のインストール

Streams for Apache Kafka ZIP ファイルを使用して、Red Hat Enterprise Linux に Kafka をインストールします。Kafka は、シングルノード環境または複数ノード環境にインストールできます。この手順では、単一の Kafka インスタンスが単一のホスト (ノード) にインストールされます。

Streams for Apache Kafka インストールファイルには、Kafka Connect、Kafka MirrorMaker 2、Kafka Bridge などの他の Kafka コンポーネントを実行するためのバイナリーが含まれています。単一ノード環境では、Kafka をインストールした同じホストからこれらのコンポーネントを実行できます。ただし、インストールファイルを追加し、他の Kafka コンポーネントは別のホストで実行することを推奨します。

前提条件

- インストールファイル をダウンロードしている。
- Red Hat Enterprise Linux リリースノート の Streams for Apache Kafka 2.7 でサポートされている設定を確認している。
- 管理者 (**root**) ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

ホストに Kafka をインストールします。

1. 新しい **kafka** ユーザーとグループを追加します。

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

2. **amq-streams-<version>-bin.zip** ファイルの内容を抽出して **/opt/kafka** ディレクトリーに移動します。

```
unzip amq-streams-<version>-bin.zip -d /opt
mv /opt/kafka*redhat* /opt/kafka
```

3. **/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。

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

4. Kafka データを格納する **/var/lib/kafka** ディレクトリーを作成し、その所有権を **kafka** ユーザーに設定します。

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

Kafka のデフォルトの設定をシングルノードクラスターとして実行 できるようになりました。

インストールを使用して、Kafka Connect などの他の Kafka コンポーネントを同じホストで実行することもできます。

他のコンポーネントを実行するには、コンポーネント設定の **bootstrap.servers** プロパティーを使用して、Kafka ブローカーに接続するためのホスト名とポートを指定します。

同じホスト上の単一の Kafka ブローカーを指すブートストラップサーバー設定の例

```
bootstrap.servers=localhost:9092
```

ただし、別のホストに Kafka コンポーネントをインストールして実行することを推奨します。

5. (オプション) Kafka コンポーネントを個別のホストにインストールします。

a. インストールファイルを各ホストの **/opt/kafka** ディレクトリーに抽出します。

b. **/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。

c. Kafka ブローカーを実行しているホスト (またはマルチノード環境のホスト) にコンポーネントを接続する **bootstrap.servers** 設定を追加します。

異なるホスト上の Kafka ブローカーを指すブートストラップサーバー設定の例

```
bootstrap.servers=kafka0.<host_ip_address>:9092,kafka1.
<host_ip_address>:9092,kafka2.<host_ip_address>:9092
```

この設定は、Kafka Connect、MirrorMaker 2、および Kafka Bridge に使用できます。

3.4. KRAFT モードでの KAFKA クラスターの実行

KRaft モードで Kafka を設定し、実行します。Kafka は、シングルノードまたはマルチノードの Kafka クラスターとして実行できます。安定性と可用性を確保するために、少なくとも 3 つのブローカーノードと 3 つのコントローラーノードを実行し、ブローカー間でトピックをレプリケーションします。

Kafka ノードは、ブローカー、コントローラー、またはその両方のロールを実行します。

ブローカーのロール

ノードまたはサーバーと呼ばれることもあるブローカーは、メッセージの保存と受け渡しを調整します。

コントローラーのロール

コントローラーはクラスターを調整し、ブローカーとパーティションのステータスを追跡するために使用するメタデータを管理します。

注記

クラスターのメタデータは、内部の **__cluster_metadata** トピックに保存されます。

ブローカーノードとコントローラーノードを組み合わせて使用できますが、これらの機能を分離したい場合があります。組み合わせたロールを実行するブローカーは、より単純なデプロイメントでより便利になります。

クラスターを識別するには、ID を作成します。この ID は、クラスターに追加するノードのログを作成するときに使用されます。

各ノードの設定で以下を指定します。

- ノード ID
- ブローカーのロール
- コントローラーとして機能するノード (または **voters**) のリスト

各コントローラーのノード ID と接続の詳細 (ホスト名とポート) を使用して、**voters** として設定されたコントローラーのリストを指定します。

設定プロパティーファイルを使用して、ロールの設定を含むブローカー設定を適用します。ブローカーの設定は、ロールによって異なります。KRaft は、3 つのブローカー設定プロパティーファイルの例を提供します。

- **/opt/kafka/config/kraft/broker.properties** には、ブローカーロールの設定例があります。
- **/opt/kafka/config/kraft/controller.properties** には、コントローラーロールの設定例があります。
- **/opt/kafka/config/kraft/server.properties** には、統合されたロールの設定例があります。

ブローカー設定は、これらのプロパティーファイルの例に基づくことができます。この手順では、**server.properties** の設定例を使用します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. Kafka クラスターの一意の ID を生成します。
   そのために **kafka-storage** ツールを使用できます。

   ```
   /opt/kafka/bin/kafka-storage.sh random-uuid
   ```

   このコマンドは ID を返します。KRaft モードでクラスター ID が必要です。

2. クラスター内の各ノードの設定プロパティーファイルを作成します。
   ファイルは、Kafka で提供される例に基づくことができます。

a. デュアルロールを **broker**、**controller**、または **broker, controller** として指定します。
たとえば、組み合わせたロールに **process.roles=broker, controller** を指定します。

b. クラスター内の各ノードに **0** から始まる一意の **node.id** を指定します。
たとえば、**node.id=1** です。

c. **<node_id>@<hostname:port>** の形式で **controller.quorum.voters** のリストを指定します。
たとえば、**controller.quorum.voters=1@localhost:9093** です。

d. リスナーを指定します。

- 各リスナーの名前、ホスト名、ポートを設定します。
たとえば、**listeners=PLAINTEXT:localhost:9092,CONTROLLER:localhost:9093** です。

- inter-broker 通信に使用するリスナー名を設定します。
たとえば、**inter.broker.listener.name=PLAINTEXT** です。

- コントローラークォーラムで使用するリスナー名を設定します。
たとえば、**controller.listener.names=CONTROLLER** です。

- Kafka への接続のためにクライアントにアドバタイズされる各リスナーの名前、ホスト名、およびポートを設定します。
たとえば、**advertised.listeners=PLAINTEXT:localhost:9092** です。

3. Kafka クラスターの各ノードにログディレクトリーを設定します。

```
/opt/kafka/bin/kafka-storage.sh format -t <uuid> -c /opt/kafka/config/kraft/server.properties
```

戻り値:

```
Formatting /tmp/kraft-combined-logs
```

<uuid> は、生成したクラスター ID に置き換えます。クラスター内の各ノードで、同じ ID を使用します。

ブローカー用に作成したプロパティーファイルを使用してブローカー設定を適用します。

デフォルトでは、**server.properties** 設定ファイルで指定されたログディレクトリー (**log.dirs**) は **/tmp/kraft-combined-logs** に設定されます。**/tmp** ディレクトリーは通常、システムが再起動するたびにクリアされるため、開発環境のみに適しています。

複数のログディレクトリーを設定するには、コンマ区切りリストを追加できます。

4. 各 Kafka ノードを起動します。

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

5. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

戻り値:

```
process ID kafka.Kafka /opt/kafka/config/kraft/server.properties
```

各ノードのログをチェックして、KRaft クラスターに正常に参加していることを確認します。

```
tail -f /opt/kafka/logs/server.log
```

トピックを作成し、ブローカーからメッセージを送受信できるようになりました。

メッセージを渡すブローカーの場合、クラスター内のブローカー全体でトピックのレプリケーションを使用して、データの耐久性を確保できます。少なくとも 3 のレプリケーション係数と、In-Sync レプリカの最小数がレプリケーション係数より 1 少ない数に設定されるようにトピックを設定します。詳細は、「トピックの作成」 を参照してください。

3.5. STREAMS FOR APACHE KAFKA サービスの停止

スクリプトを実行することで Kafka サービスを停止できます。スクリプトの実行後、Kafka サービスへのすべての接続が終了します。

手順

1. Kafka ノードを停止します。

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

2. Kafka ノードが停止していることを確認します。

```
jcmd | grep kafka
```

3.6. KAFKA ブローカーの正常なローリング再起動の実行

この手順では、マルチノードクラスターでブローカーの正常なローリング再起動を実行する方法を説明します。通常、Kafka クラスター設定プロパティーのアップグレードまたは変更後にローリング再起動が必要です。

注記

一部のブローカー設定では、ブローカーの再起動は必要ありません。詳細は、Apache Kafka ドキュメントの Updating Broker Configs を参照してください。

ブローカーの再起動後に、レプリケーションが不十分なトピックパーティションがないかをチェックして、レプリカパーティションの数が十分にあることを確認します。

可用性を失わずに正常な再起動を実現するには、トピックをレプリケーションしていること、および少なくとも最小数のレプリカ (**min.insync.replicas**) が同期していることを確認してください。**min.insync.replicas** 設定は、書き込みを成功とみなすために書き込みを確認する必要のあるレプリカの最小数を決定します。

マルチノードクラスターの場合に、標準的な方法として、トピックレプリケーション係数を 3 以上に、In-Sync レプリカの最小数をレプリケーション係数よりも 1 少なく設定します。データの持続性のためにプロデューサー設定で **acks=all** を使用している場合は、再起動したブローカーが、次のブローカーを再起動する前にレプリケーションするすべてのパーティションと同期していることを確認します。

すべてのパーティションが同じブローカーにあるため、単一ノードのクラスターは再起動時に利用できなくなります。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Kafka クラスターが想定どおりに動作している。
  レプリケーションが不十分なパーティションや、ブローカーの動作に影響を与えるその他の問題がないかどうかを確認します。この手順では、レプリケーションが不十分なパーティションをチェックする方法について説明します。

手順

各 Kafka ブローカーで以下の手順を実行します。次のステップに進む前に、最初のブローカーの手順を完了してください。コントローラーとしても機能するブローカーでは、最後に手順を実行します。そうしない場合、再起動を複数回行う時にコントローラーを変更する必要があります。

1. Kafka ブローカーを停止します。

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

2. 完了後に再起動を必要とするブローカー設定に変更を加えます。
   詳細は、以下を参照してください。
   - Kafka の設定
   - Kafka ノードのアップグレード
3. Kafka ブローカーを再起動します。

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

4. Kafka が稼働していることを確認します。

   ```
   jcmd | grep kafka
   ```

   戻り値:

   ```
   process ID kafka.Kafka /opt/kafka/config/kraft/server.properties
   ```

   各ノードのログをチェックして、KRaft クラスターに正常に参加していることを確認します。

   ```
   tail -f /opt/kafka/logs/server.log
   ```

5. ブローカで、レプリケーションが不十分なパーティションがゼロになるまで待ちます。コマンドラインから確認するか、メトリクスを使用できます。
   - **--under-replicated-partitions** パラメーターを指定して **kafka-topics.sh** コマンドを使用します。

     ```
     /opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port>  --describe --under-replicated-partitions
     ```

以下に例を示します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --under-replicated-partitions
```

このコマンドは、クラスターでレプリケーションが不十分なパーティションのトピックのリストを表示します。

レプリケーションが不十分なパーティションのトピック

```
Topic: topic3 Partition: 4 Leader: 2 Replicas: 2,3 Isr: 2
Topic: topic3 Partition: 5 Leader: 3 Replicas: 1,2 Isr: 1
Topic: topic1 Partition: 1 Leader: 3 Replicas: 1,3 Isr: 3
# …
```

In-Sync レプリカ (ISR) の数がレプリカの数より少ない場合、レプリケーションが不十分なパーティションが一覧表示されます。リストが返されない場合は、レプリケーションが不十分なパーティションはありません。

- **UnderReplicatedPartitions** メトリックを使用します。

```
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
```

このメトリックからは、レプリカが不十分なパーティションの数がわかります。数がゼロになるまで待機します。

ヒント

トピックにレプリケーションが不十分なパーティションがあると、Kafka Exporter を使用してアラートを作成します。

再起動時のログの確認

ブローカーが起動できない場合は、アプリケーションログで情報を確認します。**/opt/kafka/logs/server.log** アプリケーションログでブローカーのシャットダウンと再起動のステータスを確認することもできます。

第4章 KRAFT モードへの移行

Kafka クラスターでのメタデータ管理に ZooKeeper を使用している場合は、KRaft モードで Kafka を使用するように移行できます。KRaft モードは ZooKeeper を置き換えて分散調整を行い、信頼性、スケーラビリティー、およびスループットを強化します。

移行中に、クラスター管理用の ZooKeeper に代わるコントローラーノードのクォーラムをインストールします。コントローラー設定で KRaft の移行を有効にするには、**zookeeper.metadata.migration.enable** プロパティーを **true** に設定します。コントローラーが起動したら、同設定プロパティーを使用して現在のクラスターブローカーで KRaft の移行を有効にします。移行が完了したら、ブローカーを KRaft の使用に切り替えて、コントローラーを移行モードから外します。

KRaft は複数のディスクを備えた JBOD ストレージをサポートしていないため、移行を開始する前に、環境が KRaft モードで Kafka をサポートできることを確認してください。

前提条件

- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。
- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Kafka 3.7.0 以降で、Streams for Apache Kafka 2.7 以降を使用する必要があります。以前のバージョンの Streams for Apache Kafka を使用している場合は、KRaft モードに移行する前にアップグレードしてください。
- 移行プロセスを確認するためにログが有効になっている。
  クラスター内のコントローラーおよびブローカー上のルートロガーに対して **log4j.properties** で **DEBUG** レベルを設定すると便利です。移行に固有のコントローラーロガーの場合は、**TRACE** を設定します。

  コントローラーロギング設定

  ```
  log4j.rootLogger=DEBUG
  log4j.logger.org.apache.kafka.metadata.migration=TRACE
  ```

手順

1. Kafka クラスターのクラスター ID を取得します。
   これを行うには、**zookeeper-shell** ツールを使用できます。

   ```
   /opt/kafka/bin/zookeeper-shell.sh localhost:2181 get /cluster/id
   ```

   このコマンドはクラスター ID を返します。

2. KRaft コントローラークォーラムをクラスターにインストールします。

   a. **controller.properties** ファイルを使用して、各ホスト上でコントローラーノードを設定します。
   各コントローラーには少なくとも次の設定が必要です。

   - 一意のノード ID
   - 移行有効フラグの **true** への設定

- ZooKeeper の接続の詳細
- コントローラーリスナー
- コントローラー投票者のクォーラム

コントローラー設定の例

```
process.roles=controller
node.id=1

zookeeper.metadata.migration.enable=true
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-
domain.com:2181,zoo3.my-domain.com:2181

listeners=CONTROLLER://0.0.0.0:9090
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT
controller.quorum.voters=1@localhost:9090
```

コントローラークォーラムの形式は、コンマ区切りリストの **<node_id>@<hostname>:<port>** です。

b. 各コントローラーノードのログディレクトリーをセットアップします。

```
/opt/kafka/bin/kafka-storage.sh format -t <uuid> -c
/opt/kafka/config/kraft/controller.properties
```

戻り値:

```
Formatting /tmp/kraft-controller-logs
```

<uuid> は、取得したクラスター ID に置き換えます。クラスター内の各コントローラーノードに同じクラスター ID を使用します。

コントローラー用に設定したプロパティーファイルを使用して、コントローラー設定を適用します。

デフォルトでは、**controller.properties** 設定ファイルで指定されたログディレクトリー (**log.dirs**) は **/tmp/kraft-controller-logs** に設定されます。**/tmp** ディレクトリーは通常、システムが再起動するたびにクリアされるため、開発環境のみに適しています。

複数のログディレクトリーを設定するには、コンマ区切りリストを追加できます。

c. 各コントローラーを起動します。

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

d. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

戻り値:

```
process ID kafka.Kafka /opt/kafka/config/kraft/controller.properties
```

各コントローラーのログをチェックして、KRaft クラスターに正常に参加していることを確認します。

```
tail -f /opt/kafka/logs/controller.log
```

3. 各ブローカーで移行を有効にします。

   a. 実行中の場合は、ホスト上で実行している Kafka ブローカーを停止します。

   ```
   /opt/kafka/bin/kafka-server-stop.sh
   jcmd | grep kafka
   ```

   マルチノードクラスターで Kafka を実行している場合は、「Kafka ブローカーの正常なローリング再起動の実行」 を参照してください。

   b. **server.properties** ファイルを使用して移行を有効にします。
   各ブローカーには少なくとも次の追加設定が必要です。

   - Inter-broker プロトコルバージョンのバージョン 3.5 への設定
   - 移行有効フラグ
   - コントローラーリスナー
   - コントローラー投票者のクォーラム

   ブローカー設定の例

   ```
   broker.id=0
   inter.broker.protocol.version=3.5

   zookeeper.metadata.migration.enable=true
   zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-domain.com:2181

   listeners=CONTROLLER://0.0.0.0:9090
   controller.listener.names=CONTROLLER
   listener.security.protocol.map=CONTROLLER:PLAINTEXT
   controller.quorum.voters=1@localhost:9090
   ```

   ZooKeeper の接続の詳細はすでに存在するはずです。ブローカーのコントローラー設定は、コントローラーの設定と同じです。

   c. 更新したブローカーを再起動します。

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

   移行は自動的に開始しますが、クラスター内のトピックとパーティションの数によっては時間がかかる場合があります。

   d. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

戻り値:

```
process ID kafka.Kafka /opt/kafka/config/kraft/server.properties
```

4. アクティブコントローラーのログをチェックして、移行が完了していることを確認します。

```
/opt/kafka/bin/zookeeper-shell.sh localhost:2181 get /controller
```

Completed migration of metadata from ZooKeeper to KRaft. という **INFO** ログエントリーを探します。

5. 各ブローカーを KRaft モードで実行するように切り替えます。
   a. 前述の手順と同様にブローカーを停止します。
   b. **server.properties** ファイル内のブローカー設定を更新します。
      - **broker.id** を、同じ ID を使用する **node.id** に置き換えます。
      - ブローカーの **broker** KRaft ロールを追加します。
      - inter-broker プロトコルバージョン (**inter.broker.protocol.version**) を削除します。
      - 移行有効フラグ (**zookeeper.metadata.migration.enable**) を削除します。
      - ZooKeeper 設定を削除します。

      KRaft のブローカー設定の例

```
node.id=0
process.roles=broker

listeners=CONTROLLER://0.0.0.0:9090
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT
controller.quorum.voters=1@localhost:9090
```

   c. ブローカー設定で ACLS を使用している場合は、**authorizer.class.name** プロパティーを使用して、オーソライザーを KRaft ベースの標準オーソライザーに更新します。
      ZooKeeper ベースのブローカーは **authorizer.class.name=kafka.security.authorizer.AclAuthorizer** を使用します。

      KRaft ベースのブローカーに移行する場合は、**authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer** を指定します。
   d. 前述の手順と同様にブローカーを再起動します。
6. 各コントローラーを移行モードから切り替えます。
   a. 前述の手順と同様にコントローラーを停止します。
   b. **zookeeper.metadata.migration.enable** プロパティーを **controller.properties** ファイルから削除します。

c. 前述の手順と同様にコントローラーを再起動します。

移行後のコントローラー設定の例

```
process.roles=controller
node.id=1

listeners=CONTROLLER://0.0.0.0:9090
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT
controller.quorum.voters=1@localhost:9090
```

第5章 STREAMS FOR APACHE KAFKA の設定

Kafka 設定プロパティーファイルを使用して、Streams for Apache Kafka を設定します。

プロパティーファイルは Java 形式で、各プロパティーは次の形式で個別の行に記述されます。

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

または **!** で始まる行はコメントとして扱われ、Streams for Apache Kafka コンポーネントによって無視されます。

```
# This is a comment
```

改行/キャリッジリターンの直前で \ を使用して、値を複数の行に分割できます。

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

プロパティーファイルの変更を保存したら、Kafka ノードを再起動する必要があります。マルチノード環境では、クラスター内の各ノードでこのプロセスを繰り返します。

5.1. 標準の KAFKA 設定プロパティーの使用

標準の Kafka 設定プロパティーを使用して Kafka コンポーネントを設定します。

プロパティーは、以下の Kafka コンポーネントの設定を制御および調整するオプションを提供します。

- ブローカー
- トピック
- プロデューサークライアント、コンシューマークライアント、および管理クライアント
- Kafka Connect
- Kafka Streams

ブローカーおよびクライアントパラメーターには、認可、認証、および暗号化を設定するオプションが含まれます。

Kafka 設定プロパティーの詳細と、プロパティーを使用してデプロイメントを調整する方法は、以下のガイドを参照してください。

- Kafka 設定プロパティー
- Kafka 設定のチューニング

5.2. 環境変数から設定値の読み込み

環境変数の設定プロバイダープラグインを使用して、環境変数から設定データを読み込みます。環境変数設定プロバイダーを使用して、環境変数から証明書や JAAS 設定を読み込むことができます。

プロバイダーを使用して、プロデューサーやコンシューマーを含む、すべての Kafka コンポーネントの設定データを読み込むことができます。たとえば、プロバイダーを使用して、Kafka Connect コネクター設定のクレデンシャルを提供します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- 環境変数設定プロバイダーの JAR ファイルがある。
  JAR ファイルは、Streams for Apache Kafka アーカイブ から入手できます。

手順

1. 環境変数設定プロバイダー JAR ファイルを Kafka **libs** ディレクトリーに追加します。
2. Kafka コンポーネントの設定プロパティーファイルで環境変数設定プロバイダーを初期化します。たとえば、Kafka のプロバイダーを初期化するには、設定を **server.properties** ファイルに追加します。

   環境変数の設定プロバイダーを有効にする設定

   ```
   config.providers.env.class=org.apache.kafka.common.config.provider.EnvVarConfigProvider
   ```

3. プロパティーファイルに設定を追加して、環境変数からデータをロードします。

   環境変数からデータをロードするための設定

   ```
   option=${env:<MY_ENV_VAR_NAME>}
   ```

   MY_ENV_VAR_NAME などの大文字または大文字の環境変数の命名規則を使用します。

4. 変更を保存します。
5. Kafka コンポーネントを再起動します。
   マルチノードクラスターでブローカーを再起動する方法は、「Kafka ブローカーの正常なローリング再起動の実行」 を参照してください。

5.3. 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。推奨される設定ファイルの場所は **/opt/kafka/config/kraft/** です。設定ファイルは **kafka** ユーザーが読み取れる必要があります。

Streams for Apache Kafka には、製品のさまざまな基本的な機能と高度な機能を強調する設定ファイルのサンプルが含まれています。これらは、Streams for Apache Kafka インストールディレクトリーの **config/kraft/** にあります。

- (デフォルト) 組み合わせたモードで実行されるノードについては **config/kraft/server.properties**
- ブローカーとして実行されるノードについては **config/kraft/broker.properties**
- コントローラーとして実行されるノードについては **config/kraft/controller.properties**

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。

5.3.1. リスナー

リスナーは、Kafka ブローカーへの接続に使用されます。各 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。リスナーごとに異なる設定が必要なため、別のポートまたはネットワークインターフェイスでリッスンできます。

リスナーを設定するには、Kafka 設定プロパティーファイルの **listeners** プロパティーを編集します。**listeners** プロパティーにコンマ区切りのリストとしてリスナーを追加します。各プロパティーを以下のように設定します。

```
<listener_name>://<hostname>:<port>
```

<hostname> が空の場合、Kafka は **java.net.InetAddress.getCanonicalHostName()** クラスをホスト名として使用します。

複数のリスナーの設定例

```
listeners=internal-1://:9092,internal-2://: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、最初にクラスターノードの 1 つである **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イアントにクラスター内のすべてのブローカーのリストを提供し、クライアントは各ブローカーに個別に接続します。ブローカーのリストは、設定された **listeners** に基づいています。

アドバタイズされたリスナー

任意で、**advertised.listeners** プロパティーを使用して、**listeners** プロパティーに指定されたものとは異なるリスナーアドレスのセットをクライアントに提供できます。これは、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にある場合や、IP アドレスの代わりに外部 DNS 名が使用されている場合に便利です。

advertised.listeners プロパティーは **listeners** プロパティーと同じ方法でフォーマットされます。

アドバタイズされたリスナーの設定例

```
listeners=internal-1://:9092,internal-2://:9093
advertised.listeners=internal-1://my-broker-1.my-domain.com:1234,internal-2://my-broker-1.my-domain.com:1235
```

注記

アドバタイズされたリスナーの名前は、**listeners** プロパティーに記載されているものと一致する必要があります。

inter-broker リスナー

inter-broker リスナー は、Kafka Inter-broker の通信に使用されます。inter-broker 通信は以下に必要です。

- 異なるブローカー間のワークロードの調整
- 異なるブローカーに保存されているパーティション間でのメッセージのレプリケーション

inter-broker リスナーは、任意のポートに割り当てることができます。複数のリスナーが設定されている場合、ブローカー設定の **inter.broker.listener.name** プロパティーで inter-broker リスナーの名前を定義できます。

ここでは、inter-broker リスナーの名前は **REPLICATION** です。

```
listeners=REPLICATION://0.0.0.0:9091
inter.broker.listener.name=REPLICATION
```

コントローラーリスナー

コントローラー設定は、クラスターの調整と、ブローカーおよびパーティションのステータスを追跡するためのメタデータの管理とを行うコントローラーと接続および通信するために使用されます。

デフォルトでは、コントローラーとブローカー間の通信には専用のコントローラーリスナーが使用されます。コントローラーは、パーティションリーダーシップの変更などの管理タスクを調整するため、これらのリスナーが 1 つ以上必要です。

controller.listener.names プロパティーを使用して、コントローラーに使用するリスナーを指定します。**controller.quorum.voters** プロパティーを使用して、コントローラー投票者のクォーラムを指定できます。クォーラムにより、管理タスクのリーダーとフォロワーの構造が有効になります。リーダーは操作をアクティブに管理し、フォロワーはホットスタンバイとして、メモリー内のメタデータの整合性を確保し、フェイルオーバーを容易にします。

```
listeners=CONTROLLER://0.0.0.0:9090
controller.listener.names=CONTROLLER
controller.quorum.voters=1@localhost:9090
```

コントローラー投票者の形式は **<cluster_id>@<hostname>:<port>** です。

5.3.2. 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 が配信する必要がある実際のデータ (レコードの形式) が含まれます。これらのレコードは、ブローカーのアクティビティーの詳細を示すアプリケーションログファイルとは異なることに注意してください。

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。これは、インストール時に作成された **/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、それぞれを別の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
```

5.3.3. ノード ID

ノード ID は、クラスター内の各ノード (ブローカーまたはコントローラー) の一意の識別子です。ノード ID として 0 以上の整数を割り当てることができます。ノード ID は、再起動またはクラッシュ後にノード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されないよう

にすることが重要です。

ノード ID は、Kafka 設定プロパティーファイルで設定されます。

```
node.id=1
```

第6章 KAFKA へのアクセスのセキュア化

クライアントが Kafka ブローカーに対して持つアクセスを管理することで、Kafka クラスターを保護します。Kafka ブローカーとクライアントをセキュリティー保護するための設定オプションを指定する

Kafka ブローカーとクライアント間のセキュアな接続には、次のものが含まれます。

- データ交換の暗号化
- アイデンティティー証明に使用する認証
- ユーザーが実行するアクションを許可または拒否する認可

クライアントに指定された認証および認可メカニズムは、Kafka ブローカーに指定されたものと一致する必要があります。

6.1. 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、どのリスナーがどのセキュリティープロトコルを使用するかを定義します。各リスナー名がセキュリティープロトコルにマッピングされます。サポートされるセキュリティープロトコルは次のとおりです。

PLAINTEXT
: 暗号化または認証を使用しないリスナー。

SSL
: TLS 暗号化を使用し、オプションで TLS クライアント証明書を使用した認証を使用するリスナー。

SASL_PLAINTEXT
: 暗号化なし、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** 設定の場合、

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

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

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

これにより、リスナー **INT1** は暗号化されていない接続および SASL 認証を使用し、リスナー **INT2** は暗号化された接続および SASL 認証を使用し、**REPLICATION** インターフェイスは TLS による暗号化 (TLS クライアント認証が使用される可能性があり) を使用するように設定されます。同じセキュリティープロトコルを複数回使用できます。以下は、有効な設定の例です。

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

このような設定は、すべてのインターフェイスに TLS 暗号化および TLS 認証 (オプション) を使用します。

6.2. 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 による暗号化およびサーバー認証を使用するには、秘密鍵と公開鍵が含まれるキーストアを提供する必要があります。これは通常、Java Keystore (JKS) 形式のファイルを使用して行われます。このファイルのパスは、**ssl.keystore.location** プロパティーに設定されます。**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鍵と自己署名の鍵を使用できます。認証局が署名する鍵を使用することが、常に推奨される方法です。クライアントが接続している Kafka ブローカーのアイデンティティーを検証できるようにするには、証明書に Common Name (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接頭辞 **listener.name.<NameOfTheListener>.** を付けることができます。この場合、リスナーの名前は常に小文字である必要があります。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示しています。

```
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/replication.jks
listener.name.replication.ssl.keystore.password=123456
```

その他の TLS 設定オプション

上記のメインの TLS 設定オプションの他に、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リスト。各暗号スイートは、TLS 接続に使用される認証、暗号化、MAC、および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っています。

ssl.enabled.protocols

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

6.2.1. 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

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. クラスター内のすべての 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たアドレスおよびブートストラップアドレスが必要です。
2. すべてのクラスターノードの Kafka 設定プロパティーファイルを次のように編集します。
   - **listener.security.protocol.map** フィールドを変更して、TLS 暗号化を使用するリスナーに **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持つ JKS キーストアへのパスに設定します。
   - **ssl.keystore.password** オプションを、キーストアの保護に使用したパスワードに設定します。
     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. Kafka ブローカーを (再) 起動します。

6.3. 認証

Kafka クラスターへのクライアント接続を認証するには、次のオプションを使用できます。

TLS クライアント認証

暗号化接続で X.509 証明書を使用する TLS (Transport Layer Security)

Kafka SASL

サポートされる認証メカニズムを使用した Kafka SASL (Simple Authentication and Security Layer)

OAuth 2.0

OAuth 2.0 のトークンベースの認証

SASL 認証は、暗号化されていないプレーン接続と TLS 接続の両方のさまざまな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く認証。
- **SCRAM-SHA-256** および **SCRAM-SHA-512** - Salted Challenge Response Authentication Mechanism (SCRAM) を使用した認証。
- **GSSAPI** - Kerberos サーバーに対する認証。

警告

PLAIN メカニズムは、ネットワークを通じてユーザー名とパスワードを暗号化されていない形式で送信します。TLS による暗号化と組み合わせる場合にのみ使用してください。

6.3.1. TLS クライアント認証の有効化

Kafka ブローカーで TLS クライアント認証を有効にして、すでに TLS 暗号化を使用している Kafka ノードへの接続のセキュリティーを強化します。

ssl.client.auth プロパティーを使用して、次のいずれかの値で TLS 認証を設定します。

- **none** - TLS クライアント認証はオフです (デフォルト)。
- **requested** - TLS クライアント認証は任意です。
- **required** - 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する場合、認証されたプリンシパル名はクライアント証明書内の識別名から派生したものになります。たとえば、**CN=someuser** という識別名の証明書を持つユーザーは、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。このプリンシパル名は、認証されたユーザーまたはエンティティーの一意の識別子を提供します。TLS クライアント認証が使用されておらず、SASL が無効な場合、プリンシパル名はデフォルトで **ANONYMOUS** になります。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- 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される CA (認証局) の公開鍵を含む JKS (Java Keystore) トラストストアを準備します。
2. すべてのクラスターノードの Kafka 設定プロパティーファイルを次のように編集します。
   - **ssl.truststore.location** プロパティーを使用して JKS トラストストアへのパスを指定します。
   - トラストストアがパスワードで保護される場合は、**ssl.truststore.password** プロパティーを使用してパスワードを設定します。
   - **ssl.client.auth** プロパティーを **required** に設定します。

     TLS クライアント認証の設定

```
ssl.truststore.location=/path/to/truststore.jks
ssl.truststore.password=123456
ssl.client.auth=required
```

3. Kafka ブローカーを (再) 起動します。

6.3.2. SASL PLAIN クライアント認証の有効化

Kafka で SASL PLAIN 認証を有効にして、Kafka ノードへの接続のセキュリティーを強化します。

SASL 認証は、**KafkaServer** JAAS コンテキストを使用して、Java Authentication and Authorization Service (JAAS) を通じて有効にします。JAAS 設定は、専用のファイルで定義することも、Kafka 設定で直接定義することもできます。

推奨される専用ファイルの場所は **/opt/Kafka/config/jaas.conf** です。**kafka** ユーザーがファイルを読み取れることを確認してください。すべての Kafka ノードで JAAS 設定ファイルの同期を維持します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. **/opt/kafka/config/jaas.conf** JAAS 設定ファイルを編集または作成して、**PlainLoginModule** を有効にし、許可されるユーザー名とパスワードを指定します。
このファイルがすべての Kafka ブローカーで同じであることを確認します。

   JAAS 設定

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

2. すべてのクラスターノードの Kafka 設定プロパティーファイルを次のように編集します。

   - **listener.security.protocol.map** プロパティーを使用して、特定のリスナーで SASL PLAIN 認証を有効にします。**SASL_PLAINTEXT** または **SASL_SSL** を指定します。
   - **sasl.enabled.mechanisms** プロパティーを **PLAIN** に設定します。

     SASL plain 設定

     ```
     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 (再) 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/kraft/server.properties
```

6.3.3. SASL SCRAM クライアント認証の有効化

Kafka で SASL SCRAM 認証を有効にして、Kafka ノードへの接続のセキュリティーを強化します。

SASL 認証は、**KafkaServer** JAAS コンテキストを使用して、Java Authentication and Authorization Service (JAAS) を通じて有効にします。JAAS 設定は、専用のファイルで定義することも、Kafka 設定で直接定義することもできます。

推奨される専用ファイルの場所は **/opt/Kafka/config/jaas.conf** です。**kafka** ユーザーがファイルを読み取れることを確認してください。すべての Kafka ノードで JAAS 設定ファイルの同期を維持します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. **/opt/kafka/config/jaas.conf** JAAS 設定ファイルを編集または作成して、**ScramLoginModule** を有効にします。
   このファイルがすべての Kafka ブローカーで同じであることを確認します。

   JAAS 設定

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

2. すべてのクラスターノードの Kafka 設定プロパティーファイルを次のように編集します。

   - **listener.security.protocol.map** プロパティーを使用して、特定のリスナーで SASL SCRAM 認証を有効にします。**SASL_PLAINTEXT** または **SASL_SSL** 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 (再) 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/kraft/server.properties
   ```

6.3.4. 複数の SASL メカニズムの有効化

SASL 認証を使用する場合、複数のメカニズムを有効にすることができます。Kafka は複数の SASL メカニズムを同時に使用できます。複数のメカニズムが有効な場合、特定のクライアントが使用するメカニズムを選択できます。

複数のメカニズムを使用するには、各メカニズムに必要な設定をセットアップします。**sasl.mechanism.inter.broker.protocol** プロパティーを使用して、異なる **KafkaServer** JAAS 設定を同じコンテキストに追加し、Kafka 設定内の複数のメカニズムをコンマ区切りのリストとして有効にできます。

複数の SASL メカニズムの JAAS 設定

```
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;
};
```

SASL メカニズムの有効化

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

6.3.5. inter-broker 認証のための SASL の有効化

Kafka ノード間の SASL SCRAM 認証を有効にして、inter-broker 接続のセキュリティーを強化します。Kafka クラスターへのクライアント接続に SASL 認証を使用するだけでなく、inter-broker 認証にも SASL を使用できます。クライアント接続用の SASL とは異なり、inter-broker 通信用に選択できるメカニズムは 1 つだけです。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- SCRAM メカニズムを使用している場合は、Kafka クラスターに SCRAM 認証情報を登録します。
  Kafka クラスター内のすべてのノードに対して **kafka-storage.sh** ツールを使用して、inter-broker SASL SCRAM ユーザーを **__cluster_metadata** トピックに追加します。これにより、Kafka クラスターの実行前に、認証用の認証情報がブートストラップ用に更新されるようになります。

  inter-broker SASL SCRAM ユーザーの登録

  ```
  bin/kafka-storage.sh format \
  --config /opt/kafka/config/kraft/server.properties \
  --cluster-id 1 \
  ```

```
--release-version 3.7 \
--add-scram 'SCRAM-SHA-512=[name=kafka, password=changeit]' \
--ignore formatted
```

手順

1. **sasl.mechanism.inter.broker.protocol** プロパティーを使用して、Kafka 設定で inter-broker SASL メカニズムを指定します。

   Inter-broker SASL メカニズム

   ```
   sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
   ```

2. **username** フィールドと **password** フィールドを使用して、**KafkaServer** JAAS コンテキストでの inter-broker 通信用の ユーザー名とパスワードを指定します。

   Inter-broker JAAS コンテキスト

   ```
   KafkaServer {
       org.apache.kafka.common.security.plain.ScramLoginModule required
       username="kafka"
       password="changeit"
       # ...
   };
   ```

6.3.6. SASL SCRAM ユーザーの追加

この手順では、Kafka で SASL SCRAM を使用して認証用に新しいユーザーを登録するステップの概要を説明します。SASL SCRAM 認証は、クライアント接続のセキュリティーを強化します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- 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。

  ```
  /opt/kafka/kafka-configs.sh \
  --bootstrap-server <broker_host>:<port> \
  --alter \
  --add-config 'SCRAM-SHA-512=[password=<password>]' \
  --entity-type users --entity-name <username>
  ```

  以下に例を示します。

  ```
  /opt/kafka/kafka-configs.sh \
  --bootstrap-server localhost:9092 \
  --alter \
  ```

```
--add-config 'SCRAM-SHA-512=[password=123456]' \
--entity-type users \
--entity-name user1
```

6.3.7. SASL SCRAM ユーザーの削除

この手順では、Kafka で SASL SCRAM を使用して認証用に登録したユーザーを削除するステップの概要を説明します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- 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。

```
/opt/kafka/bin/kafka-configs.sh \
--bootstrap-server <broker_host>:<port> \
--alter \
--delete-config 'SCRAM-SHA-512' \
--entity-type users \
--entity-name <username>
```

以下に例を示します。

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

6.3.8. Kerberos (GSSAPI) 認証の有効化

Streams for Apache Kafka は、Kafka クラスターへのセキュアなシングルサインオンアクセスのために、Kerberos (GSSAPI)認証プロトコルの使用をサポートします。GSSAPI は、Kerberos 機能の API ラッパーで、基盤の実装の変更からアプリケーションを保護します。

Kerberos は、対称暗号化と信頼できるサードパーティーの Kerberos Key Distribution Centre (KDC) 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。

この手順では、Kafka クライアントが Kerberos (GSSAPI)認証を使用して Kafka にアクセスできるように、Streams for Apache Kafka 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、例を用いて以下の設定方法を説明します。

1. サービスプリンシパル

2. Kafka ブローカー (Kerberos ログインを使用するため)
3. プロデューサーおよびコンシューマークライアント (Kerberos 認証を使用して Kafka にアクセスするため)

この手順では、単一のホストでの Kafka インストールの Kerberos セットアップについて、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併せて説明します。

前提条件

Kerberos 認証情報を認証および認可するように Kafka 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

認証用のサービスプリンシパルの追加

Kerberos サーバーから、Kafka ブローカー、Kafka プロデューサーおよびコンシューマ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の形式は **SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-REALM** にする必要があります。

1. Kerberos KDC を使用してサービスプリンシパルと、プリンシパルキーを保存するキータブを作成します。
   Kerberos プリンシパルのドメイン名が大文字であることを確認してください。

   以下に例を示します。

   - **kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
   - **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
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/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
```

Kafka ブローカーサーバー設定して Kerberos ログインを使用

認証に Kerberos Key Distribution Center (KDC) を使用するように **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 Kafka を設定します。

1. **opt/kafka/config/jaas.conf** ファイルを以下の要素で修正します。

```
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. Kafka クラスターの各ブローカーを設定するには、**config/server.properties** ファイルのリスナー設定を変更して、リスナーが SASL/GSSAPI ログインを使用するようにします。
リスナーのセキュリティープロトコルのマップに SASL プロトコルを追加し、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) クライアントとの汎用通信用のセキュアなリスナー (通信用 TLS をサポート)、およびブローカー間通信用のレプリケーションリスナーの 2 つが設定されています。

(2) TLS に対応しているリスナーのプロトコル名は SASL_PLAINTEXT になります。コネクターが TLS に対応していない場合、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 認証における SASL メカニズムは **GSSAPI** になります。

(4) inter-broker 通信の Kerberos 認証。

(5) 認証要求に使用するサービスの名前は、同じ Kerberos 設定を使用している他のサービスと区別するために指定されます。

3. Kafka ブローカーを起動し、JVM パラメーターを使用して 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/kraft/server.properties
```

4. Kafka プロデューサーおよびコンシューマークライアントの設定して Kerberos 認証を使用

認証に Kerberos Key Distribution Center (KDC) を使用するように、**producer1** および **consumer1** 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 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. プロデューサーまたはコンシューマー設定ファイルに Kerberos 設定を追加します。
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 (ユーザー名/パスワード) 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)。

(4) **jaas.conf** で定義されたものと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**、**kinit**、**klist**、**kdestroy**

6.4. 認可

Kafka ブローカーの認可は、authorizer プラグインを使用して実装されます。

このセクションでは、Kafka で提供される **StandardAuthorizer** プラグインを使用する方法を説明します。

または、独自の認可プラグインを使用できます。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 認可 を使用できます。

6.4.1. ACL オーソライザーの有効化

Kafka 設定プロパティーファイルを編集して、ACL オーソライザーを追加します。**authorizer.class.name** プロパティーで完全修飾名を指定して、オーソライザーを有効にします。

オーソライザーの有効化

```
authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

6.4.1.1. ACL ルール

ACL オーソライザーは 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 ルールは次の形式で定義されます。

プリンシパル **P** は、ホスト **H** から <kafka_resource> **R** で <operation> **O** を許可または拒否されます。

たとえば、ユーザー **John** がホスト **127.0.0.1** からのトピック **コメント** を **表示** できるようにルールを設定できます。ホストは、John が接続しているマシンの IP アドレスです。

ほとんどの場合、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 **書き込み** できます。

特定のリソースに 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 設定ファイルで **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。

6.4.1.2. プリンシパル

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 に接続するために使用される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 などの単純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
  例: **User:CN=user1,O=MyCompany,L=Prague,C=CZ**
- **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 プリンシパルの主要部分で、Kerberos を使用して接続する場合の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して、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。

6.4.1.3. ユーザーの認証

認可を使用するには、認証を有効にし、クライアントにより使用される必要があります。そうでないと、すべての接続のプリンシパルは **User:ANONYMOUS** になります。

認証方法の詳細は、「認証」 を参照してください。

6.4.1.4. スーパーユーザー

スーパーユーザーは、ACL ルールに関係なくすべてのアクションを実行できます。

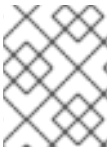
スーパーユーザーは、**super.users** プロパティーを使用して Kafka 設定ファイルで定義されます。

以下に例を示します。

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

6.4.1.5. レプリカブローカーの認証

認可を有効にすると、これは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レプリケーションに使用される inter-broker の接続が含まれます。そのため、認可を有効にする場合は、inter-broker 接続に認証を使用し、ブローカーが使用するユーザーに十分な権限を付与してください。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はユーザー名 **super.users=User:kafka-broker** が含まれている必要があります。



注記

ACL で制御できる Kafka リソースの操作の詳細は、Apache Kafka のドキュメント を参照してください。

6.4.2. ACL ルールの追加

ACL オーソライザーを使用して、アクセス制御リスト (ACL) に基づいて Kafka へのアクセスを制御する場合、**kafka-acls.sh** ユーティリティーを使用して新しい ACL ルールを追加できます。

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、リスト表示、および削除したり、その他の機能を実行したりします。パラメーターには、**--add** など、二重ハイフンの標記が必要です。

前提条件

- ユーザーが作成され、Kafka リソースにアクセスするための適切な権限が付与されています。
- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Kafka ブローカーで 認可が有効 になっている。

手順

- **--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** の **myTopic** からの読み取りを許可します。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal User:user2
```

- **user1** が IP アドレスホスト **127.0.0.1** から **myTopic** を読むためのアクセスを拒否します。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

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

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

6.4.3. ACL ルールの一覧表示

ACL オーソライザーを使用して、アクセス制御リスト (ACL) に基づいて Kafka へのアクセスを制御する場合、**kafka-acls.sh** ユーティリティーを使用して既存の ACL ルールをリスト表示できます。

前提条件

- ACL が追加 されている。

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。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --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
```

6.4.4. ACL ルールの削除

ACL オーソライザーを使用して、アクセス制御リスト (ACL) に基づいて Kafka へのアクセスを制御する場合、**kafka-acls.sh** ユーティリティーを使用して既存の ACL ルールを削除できます。

前提条件

- ACL が追加 されている。

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、**user1** および **user2** の **myTopic** からの読み取りを許可する ACL を削除します。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2

opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal User:user2
```

- **MyConsumerGroup** で **myTopic** のコンシューマーとして **user1** を追加する ACL を削除します。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

- **user1** が IP アドレスホスト **127.0.0.1** から **myTopic** を読むためのアクセスを拒否する ACL を削除します。

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

6.5. OAUTH 2.0 トークンベース認証の使用

Streams for Apache Kafka 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認可を有効にし、中央の認可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。

OAuth 2.0 認証を設定した後に OAuth 2.0 認可 を設定できます。

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 認証は、**simple** または OPA ベースの Kafka 認可と併用することもできます。

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。認可サーバーは、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。

Streams for Apache Kafka 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 認可サーバーと通信し、アクセストークンを取得または検証します。

Streams for Apache Kafka のデプロイメントでは、OAuth 2.0 統合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

Streams for Apache Kafka 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 ライブラリーに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は、**keycloak-core**、**jackson-databind**、および **slf4j-api** などの追加のサードパーティーライブラリーにも依存します。

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が含まれるようにす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。

関連情報

- OAuth 2.0 のサイト

6.5.1. OAuth 2.0 認証メカニズム

Streams for Apache Kafka 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、認可サーバー、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。

クライアントの接続に OAuth 2.0 認証を使用するように Kafka ブローカーリスナーを設定します。必要な場合は、同じ **oauth** リスナーで OAUTHBEARER および PLAIN メカニズムを使用できます。各メカニズムをサポートするプロパティーは、**oauth** リスナー設定で明示的に指定する必要があります。

OAUTHBEARER の概要

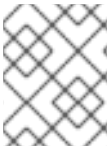
OAUTHBEARER を使用するには、Kafka ブローカーの OAuth 認証リスナー設定で **sasl.enabled.mechanisms** を **OAUTHBEARER** に設定します。詳細な設定は、「OAuth 2.0 Kafka ブローカーの設定」 を参照してください。

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

また、多くの Kafka クライアントツールでは、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。Streams for Apache Kafka では、アプリケーションの開発をサポートするために、アップストリームの Kafka Client Java ライブラリーに **OAuth コールバックハンドラー** が提供されます（ただし、他のライブラリーは対象外）。そのため、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認可サーバーに接続し、アクセストークンを取得する必要があります。

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 Secret (**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



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 を使用するには、**PLAIN** を **sasl.enabled.mechanisms** の値に追加します。

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

PLAIN は、すべての Kafka クライアントツールによって使用される簡単な認証メカニズムです。PLAIN を OAuth 2.0 認証で使用できるようにするために、Streams for Apache Kafka は **OAuth 2.0 over PLAIN** サーバー側のコールバックを提供します。

OAUTHBEARER 認証が使用される場合と同様に、クライアントクレデンシャルは準拠した認可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 Secret (**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

どちらの方法でも、クライアントは Kafka ブローカーにクレデンシャルを渡すために、PLAIN **username** および **password** プロパティーを提供する必要があります。クライアントはこれらのプロパティーを使用してクライアント ID およびシークレット、または、ユーザー名およびアクセストークンを渡します。

クライアント ID およびシークレットは、アクセストークンの取得に使用されます。

アクセストークンは、**password** プロパティーの値として渡されます。**$accessToken:** 接頭辞の有無に関わらずアクセストークンを渡します。

- リスナー設定でトークンエンドポイント (**oauth.token.endpoint.uri**) を設定する場合は、接頭辞が必要です。
- リスナー設定でトークンエンドポイント (**oauth.token.endpoint.uri**) を設定しない場合は、接頭辞は必要ありません。Kafka ブローカーは、パスワードを raw アクセストークンとして解釈します。

アクセストークンとして **password** が設定されている場合、**username** は Kafka ブローカーがアクセストークンから取得するプリンシパル名と同じものを設定する必要があります。**oauth.username.claim**、**oauth.fallback.username.claim**、**oauth.fallback.username.prefix**、および **oauth.userinfo.endpoint.uri** プロパティーを使用すると、リスナーにユーザー名の抽出オプションを指定できます。ユーザー名の抽出プロセスも、認可サーバーによって異なります。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異なります。

注記

OAuth over PLAIN は、(非推奨の) OAuth 2.0 パスワード付与メカニズムを使用したユーザー名とパスワードの受け渡し (パスワード付与) をサポートしていません。

6.5.1.1. プロパティーまたは変数を使用した 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 のプロパティーは、**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できます。
  大文字 (アッパーケース) の環境変数の命名規則を使用できます。

Streams for Apache Kafka 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 認可の設定

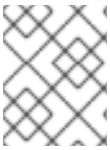
関連情報

- OAuth 2.0 Kafka ブローカーの設定

6.5.2. OAuth 2.0 Kafka ブローカーの設定

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

- 認可サーバーでの 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

認可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。

6.5.2.1. 認可サーバーの 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、認可サーバーで 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

認可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、パブリック認可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。

6.5.2.2. 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 **server.properties** ファイルで Kafka クラスターの OAuth 認証リスナー設定を有効にします。最小限の設定が必要です。また、TLS が inter-broker 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、認可サーバーによるトークン検証用にブローカーを設定できます。

- 高速のローカルトークン検証: 署名付き JWT 形式のアクセストークンと組み合わせた **JWKS** エンドポイント
- **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の設定を示しています。これは、inter-broker 通信がアプリケーションクライアントと同じリスナーを通過することを意味します。

この例では、**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。**LISTENER-NAME** は、リスナーの大文字と小文字を区別しない名前です。ここでは、リスナー **CLIENT** という名前が付けられ、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://<oauth_server_address>" \ (9)
  oauth.jwks.endpoint.uri="https://<oauth_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://<oauth_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 でのクレデンシャル交換に **OAUTHBEARER** メカニズムを有効にします。

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

(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。暗号化されていない接続 (TLS なし) には **SASL_PLAINTEXT** が使用されますが、TCP 接続層での盗聴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** プロパティーでは大文字、**listener.name** プロパティー (**listener.name.client**) では小文字、そして **listener.name.client.*** プロパティーの一部である場合は小文字で指定されます。

(5) inter-broker 通信の **OAUTHBEARER** メカニズムを指定します。

(6) inter-broker 通信のリスナーを指定します。仕様は、有効な設定のために必要です。

(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 inter-broker 通信の認証設定を行います。**oauth.client.id**、**oauth.client.secret**、および **auth.token.endpoint.uri** プロパティーは inter-broker 設定に関連するものです。

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受け入れられます。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。値は、使用される認証フローと認可サーバーによって異なります。必要に応じて、**'user.info'.'user.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からユーザー名を取得できます。

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** として認可サーバーに登録されたクライアントです。

13 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)。

14 認可サーバーへの OAuth 2.0 トークンエンドポイント URL。実稼働環境の場合は、常に **https://** urls を使用してください。例: **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**

15 inter-broker 通信の OAuth2.0 認証を有効にします (inter-broker 通信の OAuth2.0 認証にのみ必要)。

16 (オプション): トークンの期限が切れるとセッションが強制的に期限切れとなり、また、Kafka の再認証メカニズム が有効になります。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よりも短い場合、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 inter-broker の通信に使用される 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.secure.random.implementation=SHA1PRNG 4
listener.name.replication.ssl.endpoint.identification.algorithm=HTTPS 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 \ 9
  oauth.valid.issuer.uri="https://<oauth_server_address>" \
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \
  oauth.username.claim="preferred_username" ;
```

1. inter-broker 通信とクライアントアプリケーションには、個別の設定が必要です。
2. **REPLICATION** リスナーが TLS を使用し、**CLIENT** リスナーが暗号化されていないチャネルで SASL を使用するように設定します。実稼働環境では、クライアントは暗号化されたチャンネル (**SASL_SSL**) を使用できます。
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 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。
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. (inter-broker 接続に使用される) TLS 接続の確立時に **REPLICATION** リスナーのクライアントが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。
9. OAuth 2.0 の **CLIENT** リスナーを設定します。認可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 でのクレデンシャル交換に 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。高速な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://<oauth_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://<oauth_server_address>" \ 17
  oauth.jwks.endpoint.uri="https://<oauth_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. 接続するクライアントアプリケーション用のリスナー (この例では **CLIENT**) を設定します。システム **hostname** はアドバタイズされたホスト名として使用されます。このホスト名は、再接続するためにクライアントが解決する必要があります。これは唯一の設定済みリスナーであるため、inter-broker 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 を使用するように例の **CLIENT** リスナーを設定します。実稼働環境では、TCP 接続層での盗聴を避けるために、クライアントは暗号化チャンネル (**SASL_SSL**) を使用する必要があります。
3. SASL でのクレデンシャル交換の **PLAIN** 認証メカニズムおよび **OAUTHBEARER** を有効にします。inter-broker 通信に必要なため、OAUTHBEARER も指定されます。Kafka クライアントは、接続に使用するメカニズムを選択できます。
4. inter-broker 通信の **OAUTHBEARER** 認証メカニズムを指定します。
5. inter-broker 通信のリスナー (この例では **CLIENT**) を指定します。有効な設定のために必要です。
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 inter-broker 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 inter-broker 設定に関連するものです。
8. 有効な発行者 URI。この発行者からのアクセストークンのみが受け入れられます。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、ユーザーの識別に使用される **principal** です。値は、使用される認証フローと認可サーバーによって異なります。必要に応じて、**'user.info'.'user.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からユーザー名を取得できます。
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** として認可サーバーに登録されたクライアントです。
12. Kafka ブローカーの秘密 (すべてのブローカーで同じ)。
13. 認可サーバーへの OAuth 2.0 トークンエンドポイント URL。実稼働環境の場合は、常に **https://** urls を使用してください。例: **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**
14. inter-broker 通信に 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 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 OAuth 2.0 over PLAIN を有効にする任意のプロパティーです。

17 有効な発行者 URI。この発行者からのアクセストークンのみが受け入れられます。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、ユーザーの識別に使用される **principal** です。値は、使用される認証フローと認可サーバーによって異なります。必要に応じて、**'user.info'.'user.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からユーザー名を取得できます。

20 認可サーバーへの OAuth 2.0 トークンエンドポイント URL。PLAIN メカニズムの追加設定。これが指定されている場合、クライアントは **$accessToken:** 接頭辞を使用してアクセストークンを **password** として渡すことで、PLAIN 経由で認証できます。

実稼働環境の場合は、常に **https://** urls を使用してください。例: **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**

21 (オプション): トークンの期限が切れるとセッションが強制的に期限切れとなり、また、Kafka の再認証メカニズム が有効になります。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よりも短い場合、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。

6.5.2.3. 高速なローカル JWT トークン検証の設定

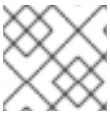
高速なローカル JWT トークンの検証では、JWT トークンの署名がローカルでチェックされます。

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 (期限切れでない) かどうかを確認します。
- 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。

認可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、認可サーバーの通信は必要はありません。OAuth 2.0 認可サーバーによって公開される **JWK 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

認可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ポイントできます。

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://<oauth_server_address>" \ 1
oauth.jwks.endpoint.uri="https://<oauth_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受け入れられます。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。キーの更新は指数バックオフのルールに従い、**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を再試行します。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。値は、使用される認証フローと認可サーバーによって異なります。必要に応じて、**'user.info'.'user.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からユーザー名を取得できます。
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。

6.5.2.4. 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 イントロスペクションベースの検証を設定するには、高速ローカル JWT トークン検証用に指定された JWK 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、認可サーバーに応じて **client ID** および **client secret** 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例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.introspection.endpoint.uri="https://<oauth_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 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。値は、使用される認証フローと認可サーバーによって異なります。必要に応じて、**'user.info'.'user.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からユーザー名を取得できます。

6.5.3. 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 **session re-authentication** 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接続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。これは、**server.properties** ファイルで有効にできます。SASL メカニズムとして OAUTHBEARER または PLAIN を有効にした TLS リスナーに **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** のトークン検証と共に使用できます。

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の接続に再認証されます。

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 とシークレット 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 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、**keycloak** 認可を使用したり、カスタム authorizer をインストールして、認可を設定する場合に考慮できます。

関連情報

- 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: Allow SASL Connections to Periodically Re-Authenticate

6.5.4. OAuth 2.0 Kafka クライアントの設定

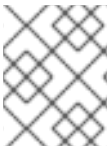
Kafka クライアントは以下のいずれかで設定されます。

- 認可サーバーから有効なアクセストークンを取得するため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)。
- 認可サーバーから提供されたツールを使用して取得された、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。アクセストークンを取得するために認可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、認可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、認可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、認可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、クライアントは認可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
- ユーザー名とパスワード、およびクライアント ID と (オプションで) シークレット

6.5.5. OAuth 2.0 クライアント認証フロー

OAuth 2.0 認証フローは、基礎となる Kafka クライアントおよび Kafka ブローカー設定によって異なります。フローは、使用する認可サーバーによってもサポートされる必要があります。

Kafka ブローカーリスナー設定は、クライアントがアクセストークンを使用して認証する方法を決定します。クライアントはクライアント ID およびシークレットを渡してアクセストークンをリクエストできます。

リスナーが PLAIN 認証を使用するように設定されている場合、クライアントはクライアント ID およびシークレット、または、ユーザー名およびアクセストークンで認証できます。これらの値は PLAIN メカニズムの **username** および **password** プロパティーとして渡されます。

リスナー設定は、以下のトークン検証オプションをサポートします。

- 認可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を使用できます。認可サーバーは、トークンで署名を検証するため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します。
- 認可サーバーが提供するトークンイントロスペクションエンドポイントへの呼び出しを使用することができます。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認可サーバーに渡します。Kafka ブローカーは応答を確認して、トークンが有効かどうかを確認します。

注記

認可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。

Kafka クライアントクレデンシャルは、以下のタイプの認証に対して設定することもできます。

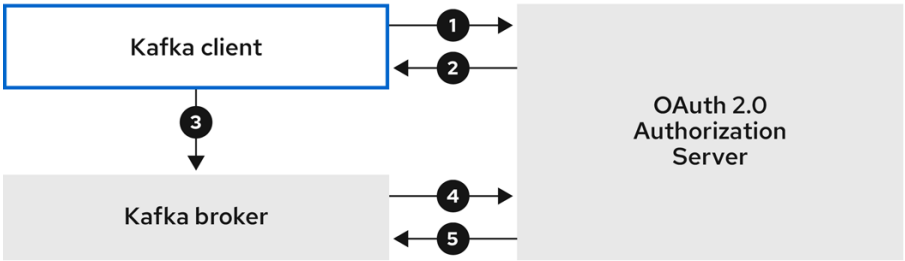
- 以前に生成された有効期間の長いアクセストークンを使用した直接ローカルアクセス
- 新しいアクセストークンを発行するには、認可サーバーに連絡します (クライアント ID とシークレット、または更新トークン、またはユーザー名とパスワードを使用)。

6.5.5.1. SASL OAUTHBEARER メカニズムを使用したクライアント認証フローの例

SASL OAUTHBEARER メカニズムを使用して、Kafka 認証に以下の通信フローを使用できます。

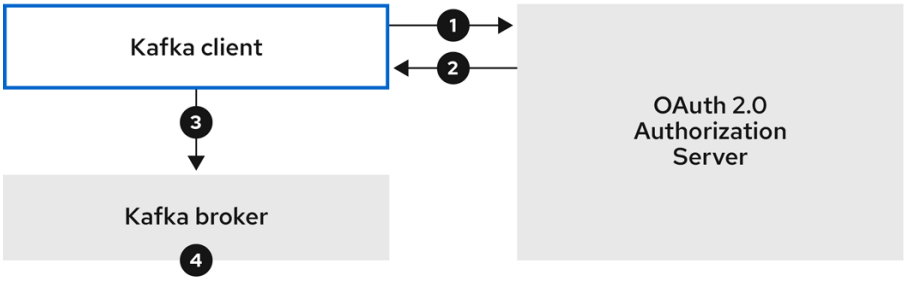
- クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を認可サーバーに委任する場合
- クライアントがクライアント ID およびシークレットを使用し、ブローカーが高速のローカルトークン検証を実行する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが検証を認可サーバーに委任する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが高速のローカル検証を実行する場合

クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を認可サーバーに委任する場合



1. Kafka クライアントは、クライアント ID およびシークレットを使用して認可サーバーからアクセストークンを要求し、必要に応じて更新トークンを要求します。または、クライアントはユーザー名とパスワードを使用して認証することもできます。
2. 認可サーバーは新しいアクセストークンを生成します。
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すことで Kafka ブローカーで認証されます。
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、認可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すことで、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、Kafka クライアントセッションが確立されます。

クライアントがクライアント ID およびシークレットを使用し、ブローカーが高速のローカルトークン検証を実行する場合



1. Kafka クライアントは、クライアント ID およびシークレットを使用し、オプションで更新トークンを使用して、トークンエンドポイントから認可サーバーで認証します。または、クライアントはユーザー名とパスワードを使用して認証することもできます。

2. 認可サーバーは新しいアクセストークンを生成します。

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すことで Kafka ブローカーで認証されます。

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。

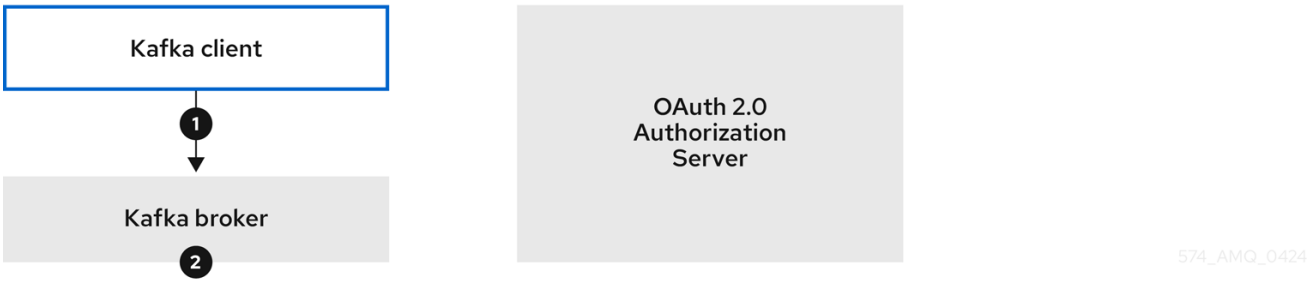
クライアントが有効期限の長いアクセストークンを使用し、ブローカーが検証を認可サーバーに委任する場合

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すために Kafka ブローカーで認証します。

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、認可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、Kafka クライアントセッションが確立されます。

クライアントが有効期限の長いアクセストークンを使用し、ブローカーが高速のローカル検証を実行する場合



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すために Kafka ブローカーで認証します。

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

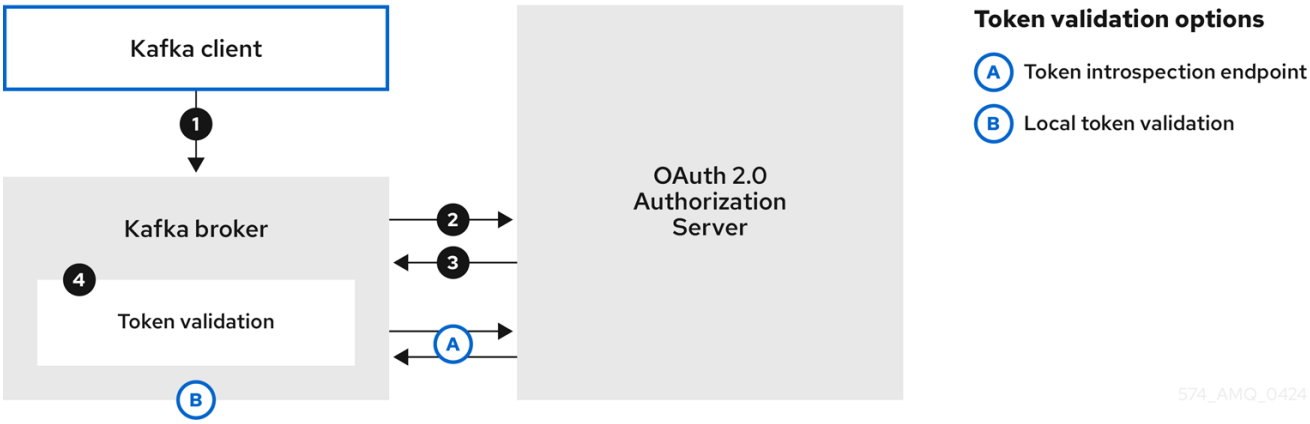
トークンが取り消された場合に認可サーバーとのチェックが行われないため、高速のローカル JWT トークン署名の検証は有効期限の短いトークンにのみ適しています。トークンの有効期限はトークンに書き込まれますが、失効はいつでも発生する可能性があるため、認可サーバーと通信せずに対応することはできません。発行されたトークンはすべて期限切れになるまで有効とみなされます。

6.5.5.2. SASL PLAIN メカニズムを使用したクライアント認証フローの例

OAuth PLAIN メカニズムを使用して、Kafka 認証に以下の通信フローを使用できます。

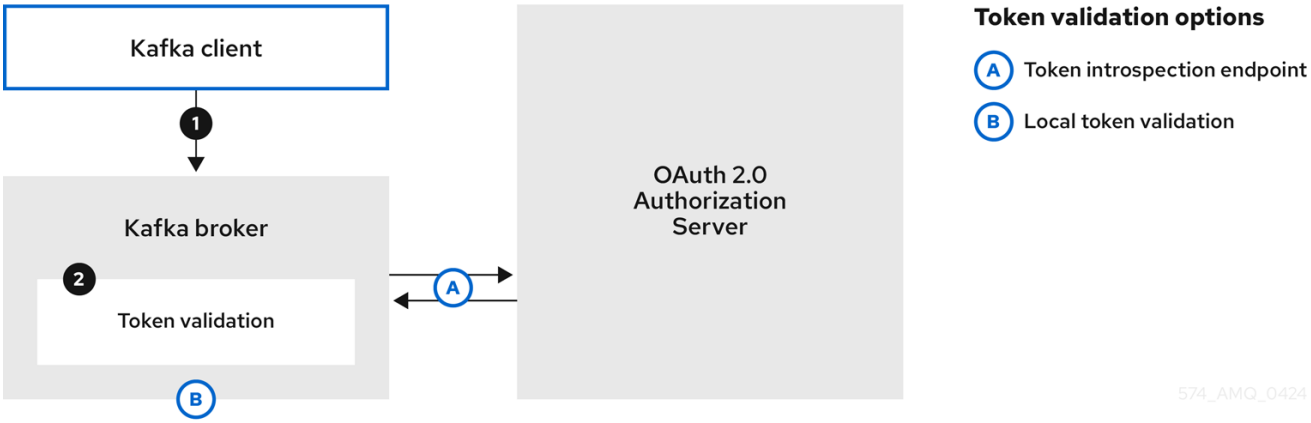
- クライアントがクライアント ID およびシークレットを使用し、ブローカーがクライアントのアクセストークンを取得する場合
- クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合

クライアントがクライアント ID およびシークレットを使用し、ブローカーがクライアントのアクセストークンを取得する場合



1. Kafka クライアントは、**clientId** をユーザー名として、**secret** をパスワードとして渡します。
2. Kafka ブローカーは、トークンエンドポイントを使用して **clientId** および **secret** を認可サーバーに渡します。
3. 認可サーバーは、新しいアクセストークンまたはエラー (クライアントクレデンシャルが有効でない場合) を返します。
4. Kafka ブローカーは、以下のいずれかの方法でトークンを検証します。
    a. トークンイントロスペクションエンドポイントが指定されている場合、Kafka ブローカーは認可サーバーでエンドポイントを呼び出すことで、アクセストークンを検証します。トークンの検証に成功した場合には、セッションが確立されます。
    b. ローカルトークンのイントロスペクションが使用される場合、要求は認可サーバーに対して行われません。Kafka ブローカーは、JWT トークン署名チェックを使用して、アクセストークンをローカルで検証します。

クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合



1. Kafka クライアントはユーザー名とパスワードを渡します。パスワードは、クライアントを実行する前に手動で取得および設定されたアクセストークンの値を提供します。

2. Kafka ブローカーリスナーが認証のトークンエンドポイントで設定されているかどうかに応じて、**$accessToken:** 文字列の接頭辞の有無にかかわらず、パスワードは渡されます。

   a. トークンエンドポイントが設定されている場合、パスワードの前に **$accessToken:** を付け、password パラメーターにクライアントシークレットではなくアクセストークンが含まれていることをブローカーに知らせる必要があります。Kafka ブローカーは、ユーザー名をアカウントのユーザー名として解釈します。

   b. トークンエンドポイントが Kafka ブローカーリスナーで設定されていない場合 (**no-client-credentials mode** を強制)、パスワードは接頭辞なしでアクセストークンを提供する必要があります。Kafka ブローカーは、ユーザー名をアカウントのユーザー名として解釈します。このモードでは、クライアントはクライアント ID およびシークレットを使用せず、**password** パラメーターは常に raw アクセストークンとして解釈されます。

3. Kafka ブローカーは、以下のいずれかの方法でトークンを検証します。

   a. トークンイントロスペクションエンドポイントが指定されている場合、Kafka ブローカーは認可サーバーでエンドポイントを呼び出すことで、アクセストークンを検証します。トークンの検証に成功した場合には、セッションが確立されます。

   b. ローカルトークンイントロスペクションが使用されている場合には、認可サーバーへの要求はありません。Kafka ブローカーは、JWT トークン署名チェックを使用して、アクセストークンをローカルで検証します。

6.5.6. OAuth 2.0 認証の設定

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

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

1. Streams for Apache Kafka クラスターおよび Kafka クライアントの OAuth 2.0 承認サーバーを設定します。

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。

6.5.6.1. OAuth 2.0 認可サーバーとしての 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、Streams for Apache Kafka と統合するための設定方法を説明します。

認可サーバーは、一元的な認証および認可の他、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。

Streams for Apache Kafka で OAuth 2.0 を使用するには、Red Hat Single Sign-On のデプロイメントを使用して認証レルムを作成および管理します。

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。

作業を開始する前に

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
- サーバー管理ガイド

前提条件

- Streams for Apache Kafka および Kafka が稼働している。

Red Hat Single Sign-On デプロイメントの場合:

- Red Hat Single Sign-On でサポートされる設定 を確認している。

手順

1. Red Hat Single Sign-On をインストールします。
   ZIP ファイルから、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、Streams for Apache Kafka 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、Streams for Apache Kafka 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

認可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。

6.5.6.2. 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認可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イスで 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。

選択した認可サーバーをサポートするプロパティーと、実装している認可のタイプを使用して、Kafka ブローカーを設定します。

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に関する詳細は、以下を参照してください。

- リスナー
- OAuth 2.0 認証メカニズム

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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- OAuth 2.0 の認可サーバーがデプロイされている。

手順

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

```
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.Ja
asClientOauthLoginCallbackHandler
```

2. **listener.name.client.oauthbearer.sasl.jaas.config** の一部としてブローカーの接続設定を行います。
   この例では、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://<oauth_server_address>/auth/realms/<realm_name>" \

oauth.jwks.endpoint.uri="https://<oauth_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認可サーバーへのトークン検証の委任

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbear
er.OAuthBearerLoginModule required \

oauth.introspection.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/protocol/openid-connect/introspection" \
  # ...
```

3. 必要な場合は、認可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** などの技術がコンテナーの外部でセキュアなチャネルを設定するために使用される場合を除き、実稼働環境で必要になります。

   a. セキュアな認可サーバーに接続するためのカスタムトラストストアを指定します。認可サーバーへのアクセスには SSL が常に必要になります。
      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. 証明書の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オフにできます。

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

このチェックは、認可サーバーへのクライアント接続が認証されるようにします。実稼働以外の環境で検証をオフにすることもできます。

4. 選択した認証フローに従っ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://<oauth_server_address>/auth/realms/<realm_name>/protocol/openid-connect/token" \ 1
  oauth.custom.claim.check="@.custom == 'custom-value'" \ 2
  oauth.scope="<scope>" \ 3
  oauth.check.audience="true" \ 4
  oauth.audience="<audience>" \ 5
  oauth.valid.issuer.uri="https://https://<oauth_server_address>/auth/<realm_name>" \ 6
  oauth.client.id="kafka-broker" \ 7
  oauth.client.secret="kafka-broker-secret" \ 8
  oauth.connect.timeout.seconds=60 \ 9
  oauth.read.timeout.seconds=60 \ 10
  oauth.http.retries=2 \ 11
  oauth.http.retry.pause.millis=300 \ 12
  oauth.groups.claim="$.groups" \ 13
  oauth.groups.claim.delimiter="," \ 14
  oauth.include.accept.header="false" ; 15
```

1 認可サーバーへの OAuth 2.0 トークンエンドポイント URL。実稼働環境の場合は、常に **https://** urls を使用してください。**KeycloakAuthorizer** を使用する場合、または inter-broker 通信に OAuth 2.0 が有効なリスナーが使用されている場合に必要です。

2 (オプション) **カスタムクレームチェック**。検証時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は、カスタムチェックが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) **scope** パラメーターがトークンエンドポイントに渡されます。**スコープ**

4 (オプション) **オーディエンスチェック**。認可サーバーが **aud** (オーディエンス) クレームを提供していて、オーディエンスチェックを実施する場合は、**ouath.check.audience** を

5 (オプション) トークンエンドポイントに渡される **audience** パラメーター。**オーディエンス** は、inter-broker 認証用にアクセストークンを取得する場合に使用されます。また、**clientId** と **secret** を使用した PLAIN クライアント認証の上にある OAuth 2.0 のクライアント名にも使われています。これは、認可サーバーに応じて、トークンの取得機能とトークンの内容のみに影響します。リスナーによるトークン検証ルールには影響しません。

6 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受け入れられます。(常に必要です)

7 すべてのブローカーで同一の、Kafka ブローカーの設定されたクライアント ID。これは、**kafka-broker** として認可サーバーに登録されたクライアントです。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 **KeycloakAuthorizer** が使用される場合に必要です。

8 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。ブローカーが認証サーバーに対して認証する必要がある場合は、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のいずれかを指定する必要があります。

9 (オプション) 認可サーバーへの接続時のタイムアウト (秒単位)。デフォルト値は 60 です。

10 (オプション): 認可サーバーへの接続時の読み取りタイムアウト (秒単位)。デフォルト値は 60 です。

11 認可サーバーへの失敗した HTTP リクエストを再試行する最大回数。デフォルト値は 0 で、再試行は実行されないことを意味します。このオプションを効果的に使用するには、**oauth.connect.timeout.seconds** オプションと **oauth.read.timeout.seconds** オプションのタイムアウト時間を短縮することを検討してください。ただし、再試行により現在のワーカースレッドが他のリクエストで利用できなくなる可能性があり、リクエストが多すぎると停止する場合、Kafka ブローカーが応答しなくなる可能性があることに注意してください。

12 認可サーバーへの失敗した HTTP リクエストの再試行を行うまでの待機時間。デフォルトでは、この時間はゼロに設定されており、一時停止は適用されません。これは、リクエストの失敗の原因となる問題の多くは、リクエストごとのネットワークの不具合やプロキシーの問題であり、すぐに解決できるためです。ただし、認可サーバーに負荷がかかっている場合、または高トラフィックが発生している場合は、このオプションを 100 ミリ秒以上の値に設定して、サーバーの負荷を軽減し、再試行が成功する可能性を高めることができます。

13 JWT トークンまたはイントロスペクションエンドポイントの応答からグループ情報を抽出するために使用される JsonPath クエリー。デフォルトでは設定されません。これは、カスタム承認者がユーザーグループに基づいて認可を決定するために使用できます。

14 1 つのコンマ区切りの文字列として返されるときにグループ情報を解析するために使用される区切り文字。デフォルト値は ,(コンマ) です。

15 (オプション) **oauth.include.accept.header** を **false** に設定して、リクエストから **Accept** ヘッダーを削除します。ヘッダーを含めることで認可サーバーとの通信時に問題が発生する場合は、この設定を使用できます。

5. OAuth 2.0 認証の適用方法、および使用されている認証サーバーのタイプに応じて、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://<oauth_server_address>/auth/realms/<realm_name>/protocol/openid-connect/userinfo" ; 5
```

1. 認可サーバーが **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. 認可サーバーは、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。必要に応じて、**'client.info'.'client.id'** のような JsonPath 式を使用して、トークン内のネストされた JSON 属性からフォールバックユーザー名を取得できます。

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わからず、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ンドポイントの応答に識別可能な情報が含まれないように、認可サーバーが設定または実装されることがあります。ユーザー ID を取得するには、**userinfo** エンドポイントの URI をフォールバックとして設定します。**oauth.fallback.username.claim**、**oauth.fallback.username.claim**、および **oauth.fallback.username.prefix** 設定が **userinfo** エンドポイントの応答に適用されます。

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。

6.5.6.3. OAuth 2.0 を使用するための Kafka Java クライアントの設定

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 API とコンシューマー API を設定します。コールバックプラグインをクライアントの **pom.xml** ファイルに追加してから、OAuth 2.0 用にクライアントを設定します。

クライアント設定で次を指定します。

- SASL (Simple Authentication and Security Layer) セキュリティープロトコル:
  - TLS 暗号化接続を介した認証用の **SASL_SSL**
  - 暗号化されていない接続を介した認証用の **SASL_PLAINTEXT**
    プロダクションには **SASL_SSL** を使用し、ローカル開発には **SASL_PLAINTEXT** のみを使用してください。**SASL_SSL** を使用する場合は、追加の **ssl.truststore** 設定が必要です。OAuth 2.0 認可サーバーへのセキュアな接続 (**https://**) には、トラストストア設定が必要です。OAuth 2.0 認可サーバーを確認するには、認可サーバーの CA 証明書をクライアント設定のトラストストアに追加します。トラストストアは、PEM または PKCS #12 形式で設定できます。
- Kafka SASL メカニズム:
  - ベアラートークンを使用したクレデンシャル交換用の **OAUTHBEARER**
  - クライアントクレデンシャル (clientId + secret) またはアクセストークンを渡す **PLAIN**
- SASL メカニズムを実装する JAAS (Java Authentication and Authorization Service) モジュール:
  - **org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule** は OAUTHBEARER メカニズムを実装します。
  - **org.apache.kafka.common.security.plain.PlainLoginModule** は plain メカニズムを実装します。

  OAuthbearer メカニズムを使用できるようにするには、**io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler** のカスタムクラスをコールバックハンドラーとして追加する必要もあります。**JaasClientOauthLoginCallbackHandler** は、クライアントのログイン中にアクセストークンの認可サーバーへの OAuth コールバックを処理します。これにより、トークンの自動更新が可能になり、ユーザーの介入なしで継続的な認証が保証されます。さらに、OAuth 2.0 パスワード付与方式を使用してクライアントのログイン認証情報を処理します。
- 以下の認証方法をサポートする SASL 認証プロパティー:
  - OAuth 2.0 クライアントクレデンシャル
  - OAuth 2.0 パスワード付与 (非推奨)
  - アクセストークン
  - トークンの更新

  SASL 認証プロパティーを JAAS 設定として追加します (**sasl.jaas.config** および **sasl.login.callback.handler.class**)。認証プロパティーを設定する方法は、OAuth 2.0 認可サーバーへのアクセスに使用している認証方法によって異なります。この手順では、プロパティーはプロパティーファイルで指定されてから、クライアント設定にロードされます。

注記

認証プロパティーを環境変数または Java システムプロパティーとして指定することもできます。Java システムプロパティーの場合は、**setProperty** を使用して設定し、**-D** オプションを使用してコマンドラインで渡すことができます。

前提条件

- Streams for Apache Kafka および Kafka が稼働している。
- OAuth 2.0 認可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。
- Kafka ブローカーが OAuth 2.0 に対して設定されている。

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>0.15.0.redhat-00006</version>
</dependency>
```

2. プロパティーファイルで以下の設定を指定して、クライアントプロパティーを設定します。
   - セキュリティープロトコル
   - SASL メカニズム
   - 使用されているメソッドに応じた JAAS モジュールと認証プロパティー
     たとえば、以下を **client.properties** ファイルに追加できます。

     クライアントクレデンシャルメカニズムのプロパティー

```
security.protocol=SASL_SSL (1)
sasl.mechanism=OAUTHBEARER (2)
ssl.truststore.location=/tmp/truststore.p12 (3)
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \ (4)
  oauth.client.id="<client_id>" \ (5)
  oauth.client.secret="<client_secret>" \ (6)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ (7)
  oauth.ssl.truststore.password="$STOREPASS" \ (8)
  oauth.ssl.truststore.type="PKCS12" \ (9)
  oauth.scope="<scope>" \ (10)
  oauth.audience="<audience>" ; (11)
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackH
Handler
```

1. TLS 暗号化接続用の **SASL_SSL** セキュリティープロトコル。ローカル開発のみでは、暗号化されていない接続で **SASL_PLAINTEXT** を使用します。
2. **OAUTHBEARER** または **PLAIN** として指定された SASL メカニズム。
3. Kafka クラスターへのセキュアなアクセスのためのトラストストア設定。
4. 認可サーバーのトークンエンドポイントの URI です。
5. クライアント ID。認可サーバーで **クライアント** を作成するときに使用される名前です。
6. 認可サーバーで **クライアント** 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。
7. この場所には、認可サーバーの公開鍵証明書 (**truststore.p12**) が含まれています。
8. トラストストアにアクセスするためのパスワード。
9. トラストストアのタイプ。
10. (オプション): 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**。認可サーバーでは、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。
11. (オプション) 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 **audience**。認可サーバーでは、クライアントによるオーディエンスの指定が必要になることがあります。

パスワード付与メカニズムのプロパティー

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.password.grant.username="<username>" \ 3
  oauth.password.grant.password="<password>" \ 4
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.scope="<scope>" \
  oauth.audience="<audience>" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
```

1. クライアント ID。認可サーバーで **クライアント** を作成するときに使用される名前です。
2. (オプション) 認可サーバーで **クライアント** 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。

3 パスワード付与認証のユーザー名。OAuth パスワード付与設定 (ユーザー名とパスワード) は、OAuth 2.0 パスワード付与メソッドを使用します。パスワード付与を使用

4 パスワード付与認証のパスワード。

注記

SASL PLAIN は、OAuth 2.0 パスワード付与メソッドを使用したユーザー名とパスワードの受け渡し (パスワード付与) をサポートしていません。

アクセストークンのプロパティー

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.access.token="<access_token>" \ 1
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbacl
Handler
```

1 Kafka クライアントの有効期間が長いアクセストークン。

トークンのプロパティーを更新する

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.refresh.token="<refresh_token>" \ 3
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbacl
Handler
```

1 クライアント ID。認可サーバーで **クライアント** を作成するときに使用される名前です。

2 (オプション) 認可サーバーで **クライアント** 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。

3 Kafka クライアントの有効期間が長い更新トークン。

3. OAUTH 2.0 認証のクライアントプロパティーを Java クライアントコードに入力します。

クライアントプロパティーの入力を示す例

```
Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties", StandardCharsets.UTF_8)) {
  props.load(reader);
}
```

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。

6.6. 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 を使用して認可ルールを設定し、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。認可は、そのユーザーのアクセスレベルを決定します。

Streams for Apache Kafka 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にブローカーへのフルアクセスが許可されており、アクセスコントロールリスト (ACL) に基づいて認可を設定するための **AclAuthorizer** プラグインと **StandardAuthorizer** プラグインも提供されています。これらのプラグインによって管理される ACL ルールは、**username** に基づいてリソースへのアクセスを許可または拒否するために使用され、このようなルールは Kafka クラスター自体の中に保存されます。ただし、Red Hat Single Sign-On を使用した OAuth 2.0 トークンベースの認可では、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 の認可および ACL が使用されるように設定することができます。

関連情報

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

6.6.1. OAuth 2.0 の認可メカニズム

Streams for Apache Kafka 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る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 の認可では、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。

6.6.1.1. Kafka ブローカーのカスタム authorizer

Streams for Apache Kafka では、Red Hat Single Sign-On の **authorizer** (**KeycloakAuthorizer**)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 authorizer を設定します。

authorizer は必要に応じて付与された権限のリストを認可サーバーから取得し、ローカルで Kafka ブローカーに認可を強制するため、クライアントの要求ごとに迅速な認可決定が行われます。

6.6.2. OAuth 2.0 認可サポートの設定

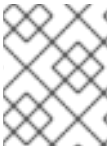
この手順では、Red Hat Single Sign-On の Authorization Services を使用して、OAuth 2.0 認可を使用するように Kafka ブローカーを設定する方法を説明します。

作業を開始する前に

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、認可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

- Streams for Apache Kafka 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する必要があります。認可を設定するときに、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 の説明にあるように、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している必要がある。

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. 認可サービスを使用して、クライアントのリソース、認可スコープ、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 に authorizer をインストールします。

```
authorizer.class.name=io.strimzi.kafka.oauth.server.authorizer.KeycloakAuthorizer
principal.builder.class=io.strimzi.kafka.oauth.server.OAuthKafkaPrincipalBuilder
```

5. Kafka ブローカーの設定を追加して、認可サーバーおよび Authorization Services にアクセスします。
ここでは、**server.properties** への追加プロパティーとして追加される設定例を示しますが、大文字で始める、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の URL。実稼働環境の場合は、常に **https://** urls 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. (オプション) 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。名前はパーミッションをターゲットにするために使用され、同じ Red Hat シングルサインオンレルム内で複数のクラスターを管理できるようにします。デフォルト値は **kafka-cluster** です。

7. (オプション) シンプルな承認に委任します。

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

(1) Red Hat Single Sign-On Authorization Services ポリシーでアクセスが拒否された場合、Kafka **AclAuthorizer** に権限を委任します。デフォルトは **false** です。

8. (オプション) TLS 接続の設定を認可サーバーに追加します。
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. トラストストアのタイプ。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 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. (オプション) 認可サーバーからの付与の更新を設定します。付与更新ジョブは、アクティブなトークンを列挙し、それぞれに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
strimzi.authorization.grants.max.idle.time.seconds="300" 3
strimzi.authorization.grants.gc.period.seconds="300" 4
strimzi.authorization.reuse.grants="false" 5
```

1. 認可サーバーからの付与のリストが更新される頻度を指定します (デフォルトでは 1 分に 1 回)。デバッグの目的で付与の更新をオフにするには、**"0"** に設定します。
2. 付与更新ジョブで使用されるスレッドプールのサイズ (並列度) を指定します。デフォルト値は **"5"** です。
3. キャッシュ内のアイドル許可を削除できるようになるまでの時間 (秒単位)。デフォルト値は 300 です。
4. キャッシュから古い許可を削除するジョブの連続実行間の時間 (秒単位)。デフォルト値は 300 です。
5. 新しいセッションに対して最新の許可を取得するかどうかを制御します。無効にすると、許可が Red Hat Single Sign-On から取得され、ユーザーのためにキャッシュされます。デフォルト値は **true** です。

10. (オプション) 認可サーバーとの通信時のネットワークタイムアウトを設定します。
以下に例を示します。

```
strimzi.authorization.connect.timeout.seconds="60" 1
strimzi.authorization.read.timeout.seconds="60" 2
strimzi.authorization.http.retries="2" 3
```

1. Red Hat Single Sign-On トークンエンドポイントに接続するときの接続タイムアウト (秒単位)。デフォルト値は **60** です。
2. Red Hat Single Sign-On トークンエンドポイントに接続するときの読み取りタイムアウト (秒単位)。デフォルト値は **60** です。

3 認可サーバーへの失敗した HTTP リクエストを (一時停止せずに) 再試行する最大回数。デフォルト値は **0** で、再試行は実行されないことを意味します。このオプションを効果的に

11. (オプション) トークンの検証と認可のために OAuth 2.0 メトリクスを有効にします。

```
oauth.enable.metrics="true" 1
```

1 OAuth メトリクスを有効にするか無効にするかを制御します。デフォルト値は **false** です。

12. (オプション) リクエストから **Accept** ヘッダーを削除します。

```
oauth.include.accept.header="false" 1
```

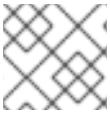
1 ヘッダーを含めることで認可サーバーとの通信時に問題が発生する場合は、**false** に設定します。デフォルト値は **true** です。

13. 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。

6.7. 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 Streams for Apache Kafka 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

6.7.1. OPA ポリシーの定義

OPA と Streams for Apache Kafka を統合する前に、粒度の細かいアクセス制御を提供するポリシーの定義方法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認可ポリシーを定義できます。

このポリシーでは、以下の項目を指定することができます。

- プロデューサークライアントに関連付けられた **ユーザープリンシパル** および **ホストアドレス**
- クライアントに許可される **操作**

- ポリシーが適用される **リソースタイプ** (**topic**) および **リソース名**

許可と拒否の決定がポリシーに書き込まれ、提供された要求とクライアント識別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が許可されるポリシーを満たす必要があります。

6.7.2. 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 authorizer プラグイン (**kafka-authorizer-opa-VERSION.jar**) を設定します。

クライアントがリクエストを行うと、OPA ポリシーエンジンは、指定された 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なければなりません。

プラグインは、ポリシーに対してチェックされる JSON 形式で、クライアント要求の詳細 (ユーザープリンシパル、操作、およびリソース) 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の識別名を取ります。

OPA はデータを使用して、リクエストを許可または拒否するためにプラグインに **true** または **false** のいずれかの応答を提供します。

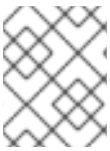
6.7.3. OPA 承認サポートの設定

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。

作業を開始する前に

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** リソースと 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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- 接続には OPA サーバーを利用できる必要がある。
- Kafka の OPA authorizer プラグインがある。

手順

1. Kafka ブローカーで操作を実行するため、クライアントリクエストの承認に必要な OPA ポリシーを記述します。
   OPA ポリシーの定義 を参照してください。

これで、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 authorizer 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. Kafka ブローカーの **server.properties** に設定をさらに追加して、OPA ポリシーエンジンおよびポリシーにアクセスします。
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) authorizer 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) authorizer プラグインが OPA ポリシーエンジンとの接続に失敗した場合に、クライアントがデフォルトで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)。すべてのリクエストについてプラグインに OPA ポリシーエンジンをクエリーする必要がないように、キャッシュが使用されます。

(4) ローカルキャッシュの最大容量 (バイト単位)。

(5) OPA ポリシーエンジンからのリロードによってローカルキャッシュが更新される時間 (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。これにより、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。

認証および認可オプションの詳細は、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と持たないクライアントを使用して、Kafka ブローカーにアクセスして、設定したパーミッションを検証します。

第7章 トピックの作成および管理

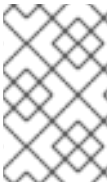
Kafka のメッセージは、常にトピックとの間で送受信されます。この章では、Kafka トピックを作成および管理する方法について説明します。

7.1. パーティションおよびレプリカ

トピックは、常に 1 つ以上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。

各パーティションには 1 つ以上のレプリカを含めることができ、レプリカはクラスター内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、**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。**レプリケーション係数** は、クラスター内で保持するコピーの数を定義します。指定した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、プロデューサーが新しいメッセージを送信し、コンシューマーが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です。フォロワーはリーダーをレプリケーションします。

リーダーが失敗すると、同期しているフォロワーの 1 つが自動的に新しいリーダーになります。各サーバーは、一部のパーティションのリーダーおよび他のパーティションのフォロワーとして機能し、クラスター内で負荷が均等に分散されます。



注記

レプリケーション係数は、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 2 つのフォロワーレプリカが設定されます。

7.2. メッセージの保持

メッセージの保持ポリシーは、Kafka ブローカーにメッセージを保存す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、メッセージの保存に関して以下のように定義できます。

- 7 日間。
- パーティションのメッセージが 1 GB になるまで。制限に達すると、最も古いメッセージが削除されます。
- 7 日間、または 1 GB の制限に達するまで。最初に制限が使用されます。

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。保持ポリシーを超えたメッセージは、新規ログセグメントが作成された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が設定されたログセグメントサイズを超えると作成されます。さらに、ユーザーは定期的に新しいセグメントの作成を要求できます。

Kafka ブローカーはコンパクト化ポリシーをサポートします。

コンパクト化ポリシーのあるトピックでは、ブローカーは常に各キーの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。コンパクト化は定期的に実行されるため、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ても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。

メッセージの保持設定オプションの詳細は、「トピックの設定」 を参照してください。

7.3. トピックの自動作成

デフォルトでは、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受信しようとすると、Kafka は自動的にトピックを作成します。この動作は、デフォルトで **true** に設定された **auto.create.topics.enable** 設定プロパティーによって制御されます。

実稼働環境では、トピックの自動作成を無効にすることを推奨します。無効にするには、Kafka 設定プロパティーファイルで **auto.create.topics.enable** を **false** に設定します。

トピックの自動作成の無効化

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

7.4. トピックの削除

Kafka には、トピックの削除を防止するオプションがあり、**delete.topic.enable property** によって制御されます。デフォルトでは、このプロパティーは **true** に設定されており、トピックを削除できます。

ただし、Kafka 設定プロパティーファイルでこのプロパティーを **false** に設定すると、トピックの削除が無効になります。この場合、トピックを削除しようとすると成功ステータスが返されますが、トピック自体は削除されません。

トピックの削除の無効化

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

7.5. トピックの設定

自動作成されたトピックは、ブローカーの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。トピックの作成後に、トピックの設定を変更することもできます。手動で作成し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

delete または **compact** に保持ポリシーを設定します。**delete** ポリシーは古いレコードを削除します。**compact** ポリシーはログコンパクションを有効にします。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デックを保持) 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ために同期する必要が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ミリ秒数。この値より古いログセグメントは削除されます。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、一番古いログセグメントが削除されます。**-1** の値は無制限を意味します。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ファイルサイズ (バイト単位)。セグメントがそのサイズに達すると、新しいセグメントが起動します。デフォルト値は **1073741824** バイト (1 ギガバイト) です。

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 **message.max.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

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

num.partitions

自動作成される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。

7.6. 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で作成され、使用されます。Kafka にはいくつかの内部トピックがあり、そのうち 2 つはコンシューマーオフセット (**__consumer_offsets**) とトランザクション状態 (**__transaction_state**) を保存するために使用されます。

__consumer_offsets トピックと **__transaction_state** トピックは、接頭辞 **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の数です。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の数です。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です。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です。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が正常であると見なされるために、確認する必要があるレプリカの最小数です。この最小値が満たされない場合、プロデューサーは例外で失敗します。デフォルト値は **2** です。

7.7. 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します。**kafka-topics.sh** は Streams for Apache Kafka 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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

トピックの作成

1. **kafka-topics.sh** ユーティリティーを使用し以下の項目を指定して、トピックを作成します。
   - **--bootstrap-server** における Kafka ブローカーのホストおよびポート。
   - **--create** オプション: 作成される新しいトピック。
   - **--topic** オプション: トピック名。
   - **--partitions** オプション: パーティション数。
   - **--replication-factor** オプション: 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できます。

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

     mytopic というトピックを作成するコマンドの例

     ```
     /opt/kafka/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。

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

mytopic というトピックを記述するコマンドの例

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

7.8. トピックの一覧表示および説明

kafka-topics.sh ツールは、トピックのリスト表示および説明に使用できます。**kafka-topics.sh** は Streams for Apache Kafka 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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

トピックの記述

1. **kafka-topics.sh** ユーティリティーを使用し以下の項目を指定して、トピックを説明します。
   - **--bootstrap-server** における Kafka ブローカーのホストおよびポート。
   - **--describe** オプション: トピックを記述することを指定するために使用します。
   - **--topic** オプション: このオプションでトピック名を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を記述します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --describe --topic <topic_name>
```

mytopic というトピックを記述するコマンドの例

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

このコマンドは、このトピックに属するすべてのパーティションおよびレプリカをリスト表示します。また、すべてのトピック設定オプションも表示されます。

7.9. 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することができます。**kafka-configs.sh** は Streams for Apache Kafka 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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

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** として、**--entity-name** をトピックの名前に設定します。
   - **--describe** オプション: 現在の設定を取得するために使用します。

     ```
     /opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --describe
     ```

     mytopic という名前のトピックの設定を取得するコマンドの例

     ```
     /opt/kafka/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、現在の設定を変更します。
   - **--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** として、**--entity-name** をトピックの名前に設定します。
   - **--alter** オプション: 現在の設定を変更するために使用します。
   - **--add-config** オプション: 追加または変更するオプションを指定します。

     ```
     /opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --alter --add-config <option>=<value>
     ```

     mytopic という名前のトピックの設定を変更するコマンドの例

     ```
     /opt/kafka/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** として、**--entity-name** をトピックの名前に設定します。
   - **--delete-config** オプション: 既存の設定オプションを削除するために使用します。
   - **--remove-config** オプション: 削除するオプションを指定します。

     ```
     /opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_host>:<port> --entity-type topics --entity-name <topic_name> --alter --delete-config <option>
     ```

     mytopic という名前のトピックの設定を変更するコマンドの例

     ```
     /opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --alter --delete-config min.insync.replicas
     ```

7.10. 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** は Streams for Apache Kafka 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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

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** における Kafka ブローカーのホストおよびポート。
   - **--delete** オプション: 既存のトピックを削除することを指定するために使用します。
   - **--topic** オプション: このオプションでトピック名を指定する必要があります。

     ```
     /opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --delete --topic
     <topic_name>
     ```

     mytopic というトピックを作成するコマンドの例

     ```
     /opt/kafka/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たことを確認します。

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --list
   ```

   すべてのトピックをリスト表示するコマンドの例

   ```
   /opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
   ```

第8章 KAFKA CONNECT での STREAMS FOR APACHE KAFKA の使用

Kafka Connect を使用して、Kafka と外部システムの間でデータをストリーミングします。Kafka Connect は、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を提供します。Kafka Connect は通常、Kafka を Kafka クラスター外のデータベース、ストレージシステム、および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 はスタンドアロンモードまたは分散モードで実行されます。

スタンドアロンモード

スタンドアロンモードでは、Kafka Connect は単一ノード上で実行されます。スタンドアロンモードは、開発とテストを目的としています。

分散モード

分散モードでは、Kafka Connect は 1 つまたは複数のワーカーノードで実行され、ワークロードはワーカーノード間で分散されます。分散モードは実稼働向けです。

Kafka Connect は、さまざまな種類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。コネクタープラグインには、シンクとソースの 2 種類があります。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 REST API を使用して、コネクターインスタンスを作成、管理、監視することもできます。

コネクター設定では、ソースコネクターまたはシンクコネクター、読み取りまたは書き込み先の Kafka トピックなどの詳細を指定します。設定を管理する方法は、Kafka Connect をスタンドアロンモードで実行しているか分散モードで実行しているかによって異なります。

- スタンドアロンモードでは、Kafka Connect REST API を通じてコネクター設定を JSON として提供することも、プロパティーファイルを使用して設定を定義することもできます。
- 分散モードでは、Kafka Connect REST API を介してコネクター設定を JSON としてのみ提供できます。

大量のメッセージ処理

設定を調整して、大量のメッセージを処理できます。詳細は、大量のメッセージの処理 を参照してください。

8.1. スタンドアロンモードでの KAFKA CONNECT の使用

Kafka Connect スタンドアロンモードでは、コネクターは Kafka Connect ワーカープロセスと同じノード上で実行され、単一の JVM 内の単一プロセスとして実行されます。これは、ワーカープロセスとコネクターが CPU、メモリー、ディスクなどの同じリソースを共有することを意味します。

8.1.1. スタンドアロンモードでの Kafka Connect の設定

Kafka Connect をスタンドアロンモードで設定するには、**config/connect-standalone.properties** 設定ファイルを編集します。以下のオプションが最も重要です。

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リスト。たとえば、**kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092** です。

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** です。

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。たとえば、**org.apache.kafka.connect.json.JsonConverter** です。

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。

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、標準的な Kafka のプロデューサーとコンシューマーの設定オプションを使用し、**producer.** または **consumer.** 接頭辞を付けます。

8.1.2. スタンドアロンモードでの Kafka Connect の実行

Kafka Connect をスタンドアロンモードで設定して実行します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- プロパティーファイルでコネクター設定を指定している。
  Kafka Connect REST API を使用して コネクターを管理 することもできます。

手順

1. **/opt/kafka/config/connect-standalone.properties** Kafka Connect 設定ファイルを編集し、**bootstrap.server** が Kafka ブローカーを指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. 設定ファイルで Kafka Connect を起動し、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
```

8.2. 分散モードでの KAFKA CONNECT の使用

分散モードでは、Kafka Connect はワーカープロセスのクラスターとして実行され、各ワーカーは個別のノードで実行されます。コネクターはクラスター内の任意のワーカー上で実行できるため、スケーラビリティとフォールトトレランスが向上します。コネクターはワーカーによって管理され、ワーカーは相互に調整して作業を分散し、各コネクターがいつでも単一のノード上で実行されるようにします。

8.2.1. 分散モードでの Kafka Connect の設定

Kafka Connect をスタンドアロンモードで設定するには、**config/connect-distributed.properties** 設定ファイルを編集します。以下のオプションが最も重要です。

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リスト。たとえば、**kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092** です。

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** です。

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。たとえば、**org.apache.kafka.connect.json.JsonConverter** です。

group.id

分散された Kafka Connect クラスターの名前。これは一意でなければならず、他のコンシューマーグループ ID と競合することはできません。デフォルト値は **connect-cluster** です。

config.storage.topic

コネクター設定の保存に使用される 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される Kafka トピック。デフォルト値は **connect-status** です。

Streams for Apache Kafka には、分散モードの Kafka Connect のサンプル設定ファイルが含まれています。Streams for Apache Kafka 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。

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、標準的な Kafka のプロデューサーとコンシューマーの設定オプションを使用し、**producer.** または **consumer.** 接頭辞を付けます。

8.2.2. 分散モードでの Kafka Connect の起動

Kafka Connect を分散モードで設定して実行します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

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 ブローカーを示すようにします。
   - **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 ワーカーノードで **/opt/kafka/config/connect-distributed.properties** 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
```

4. Kafka Connect REST API を使用して コネクターを管理 します。

8.3. コネクターの管理

Kafka Connect REST API は、コネクターを直接作成、更新、削除するためのエンドポイントを提供します。API を使用して、コネクターのステータスを確認したり、ログレベルを変更したりすることもできます。API を通じてコネクターを作成する場合は、API 呼び出しの一部としてコネクターの設定の詳細を指定します。

コネクターをプラグインとして追加および管理することもできます。プラグインは、Kafka Connect API を通じてコネクターを実装するためのクラスを含む JAR ファイルとしてパッケージ化されます。クラスパスでプラグインを指定するか、Kafka Connect のプラグインパスに追加するだけで、起動時にコネクタープラグインが実行されます。

Kafka Connect REST API またはプラグインを使用してコネクターを管理するだけでなく、Kafka Connect をスタンドアロンモードで実行するときにプロパティーファイルを使用してコネクター設定を追加することもできます。これを行うには、Kafka Connect ワーカープロセスの開始時にプロパティーファイルの場所を指定するだけです。プロパティーファイルには、コネクタークラス、ソースおよび宛先トピック、必要な認証またはシリアル化設定など、コネクターの設定の詳細が含まれている必要があります。

8.3.1. Kafka Connect API へのアクセスの制限

Kafka Connect REST API には、アクセスが認証され、ホスト名/IP アドレスおよびポート番号を含むエンドポイント URL を知っている人なら誰でもアクセスできます。Kafka Connect API へのアクセスを信頼できるユーザーのみに制限して、不正なアクションや潜在的なセキュリティーの問題を防ぐことが重要です。

セキュリティーを強化するために、Kafka Connect API の次のプロパティーを設定することを推奨します。

- (Kafka 3.4 以降) **org.apache.kafka.disallowed.login.modules** により、セキュアではないログインモジュールを明確に除外する
- **connector.client.config.override.policy** を **NONE** に設定して、コネクター設定が Kafka Connect 設定と、それが使用するコンシューマーおよびプロデューサーをオーバーライドしないようにします。

8.3.2. コネクターの設定

Kafka Connect REST API またはプロパティーファイルを使用して、コネクターインスタンスを作成、管理、監視します。Kafka Connect をスタンドアロンモードまたは分散モードで使用する場合は、REST API を使用できます。Kafka Connect をスタンドアロンモードで使用する場合は、プロパティーファイルを使用できます。

8.3.2.1. Kafka Connect REST API を使用したコネクターの管理

Kafka Connect REST API を使用する場合、**PUT** または **POST** HTTP リクエストを Kafka Connect REST API に送信し、リクエスト本文でコネクター設定の詳細を指定することで、コネクターを動的に作成できます。

ヒント

PUT コマンドを使用する場合、これはコネクターの起動と更新に使用するコマンドと同じです。

REST インターフェイスはデフォルトでポート 8083 をリッスンし、次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。リクエストボディーは、コネクター設定が含まれる JSON オブジェクトである必要があります。

GET /connectors/<connector_name>
: 特定のコネクターの情報を取得します。

GET /connectors/<connector_name>/config
: 特定のコネクターの設定を取得します。

PUT /connectors/<connector_name>/config
: 特定のコネクターの設定を更新します。

GET /connectors/<connector_name>/status
: 特定のコネクターのステータスを取得します。

GET /connectors/<connector_name>/tasks
: 特定のコネクターのタスクのリストを取得する

GET /connectors/<connector_name>/tasks/<task_id>/status
: 特定のコネクターのタスクのステータスを取得する

PUT /connectors/<connector_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/<connector_name>/stop

コネクターとそのすべてのタスクを停止します。コネクターはメッセージの処理を停止します。コネクター実行の停止は、単に一時停止するよりも長時間停止する場合に適しています。

PUT /connectors/<connector_name>/resume

一時停止されたコネクターを再開します。

POST /connectors/<connector_name>/restart

コネクターに障害が発生した場合に、コネクターを再起動します。

POST /connectors/<connector_name>/tasks/<task_id>/restart

特定のタスクを再起動します。

DELETE /connectors/<connector_name>

コネクターを削除します。

GET /connectors/<connector_name>/topics

特定のコネクターのトピックを取得します。

PUT /connectors/<connector_name>/topics/reset

特定のコネクターのアクティブなトピックのセットを空にします。

GET /connectors/<connector_name>/offsets

コネクターの現在のオフセットを取得します。

DELETE /connectors/<connector_name>/offsets

コネクターのオフセットをリセットします。コネクターは停止状態である必要があります。

PATCH /connectors/<connector_name>/offsets

コネクターのオフセットを調整します (リクエスト内の **offset** プロパティーを使用)。コネクターは停止状態である必要があります。

GET /connector-plugins

サポートされるすべてのコネクタープラグインのリストを取得します。

GET /connector-plugins/<connector_plugin_type>/config

コネクタープラグインの設定を取得します。

PUT /connector-plugins/<connector_type>/config/validate

コネクター設定を検証します。

8.3.2.2. コネクター設定プロパティーの指定

Kafka Connect コネクターを設定するには、ソースコネクターまたはシンクコネクターの設定の詳細を指定する必要があります。これを行うには 2 つの方法があります。Kafka Connect REST API を使用する方法、JSON を使用して設定を提供する方法、またはプロパティーファイルを使用して設定プロパティーを定義する方法です。コネクターの種類ごとに利用できる特定の設定オプションは異なる場合がありますが、どちらの方法でも必要な設定を指定する柔軟な方法が提供されます。

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るコネクターの名前。

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** です。

tasks.max

指定のコネクターが使用できるタスクの最大数。タスクにより、コネクターは並行して作業を実行できます。コネクターは、指定された数よりも少ないタスクを作成す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たデフォルト値がオーバーライドされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** です。

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たデフォルト値がオーバーライドされます。たとえば、**org.apache.kafka.connect.json.JsonConverter** です。

シンクコネクターに対して次のオプションの少なくとも 1 つを設定する必要があります。

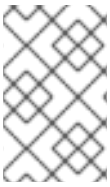
topics

入力として使用されるトピックのコンマ区切りリスト。

topics.regex

入力として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については、Apache Kafka ドキュメント のコネクタープロパティーを参照してください。



注記

Streams for Apache Kafka には、Streams for Apache Kafka インストールディレクトリーに、コネクター設定ファイルの例 **config/connect-file-sink.properties** および **config/connect-file-source.properties** が含まれます。

関連情報

- Kafka Connect REST API OpenAPI ドキュメント

8.3.3. Kafka Connect API を使用したコネクターの作成

Kafka Connect REST API を使用して、Kafka Connect で使用するコネクターを作成します。

前提条件

- Kafka Connect のインストール。

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** に送信してコネクターを作成します。以下の例では、**curl** を使用します。

```
curl -X POST -H "Content-Type: application/json" --data @sink-connector.json
http://connect0.my-domain.com:8083/connectors
```

3. **<KafkaConnectAddress>:8083/connectors** に GET リクエストを送信して、コネクターがデプロイされたことを確認します。以下の例では、**curl** を使用します。

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.4. Kafka Connect API を使用したコネクターの削除

Kafka Connect REST API を使用して、Kafka Connect からコネクターを削除します。

前提条件

- Kafka Connect のインストール。

コネクターの削除

1. **<KafkaConnectAddress>:8083/connectors/<ConnectorName>** に **GET** リクエストを送信して、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** を使用します。

```
curl -X DELETE http://connect0.my-domain.com:8083/connectors/my-connector
```

3. **<KafkaConnectAddress>:8083/connectors** に GET リクエストを送信して、コネクターが削除されたことを確認します。以下の例では、**curl** を使用します。

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.5. コネクタープラグインの追加

Kafka は、コネクター開発の開始点として使用できるサンプルコネクターを提供します。以下のサンプルコネクターは Streams for Apache Kafka 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、そのデータを Kafka トピックに送信します。

どちらのコネクターも **libs/connect-file-<kafka_version>.redhat-<build>.jar** プラグインに含まれています。

Kafka Connect でコネクタープラグインを使用するには、コネクタープラグインをクラスパスに追加するか、Kafka Connect プロパティーファイルでプラグインパスを指定してその場所にプラグインをコピーします。

クラスパスでのサンプルコネクターの指定

```
CLASSPATH=/opt/kafka/libs/connect-file-<kafka_version>.redhat-<build>.jar opt/kafka/bin/connect-distributed.sh
```

プラグインパスの設定

```
plugin.path=/opt/kafka/connector-plugins,/opt/connectors
```

plugin.path 設定オプションには、コンマ区切りのパスのリストを含めることができます。

必要に応じて、さらにコネクタープラグインを追加できます。Kafka Connect は起動時にコネクタープラグインを検索して実行します。

注記

Kafka Connect を分散モードで実行する場合は、すべてのワーカーノードでプラグインを利用できるようにする必要があります。

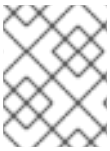
第9章 MIRRORMAKER 2 での STREAMS FOR APACHE KAFKA の使用

MirrorMaker 2 を使用して、データセンター内またはデータセンター間で 2 つ以上のアクティブな Kafka クラスター間でデータをレプリケーションします。

MirrorMaker 2 を設定するには、**config/connect-mirror-maker.properties** 設定ファイルを編集します。必要に応じて、MirrorMaker 2 の分散トレースを有効 にすることができます。

大量のメッセージ処理

設定を調整して、大量のメッセージを処理できます。詳細は、大量のメッセージの処理 を参照してください。



注記

MirrorMaker 2 には、MirrorMaker の以前のバージョンではサポートされていない機能があります。ただし、MirrorMaker 2 をレガシーモードで使用するように設定 できます。

9.1. ACTIVE/ACTIVE または ACTIVE/PASSIVEモードの設定

MirrorMaker 2 は、**active/passive** または **active/active** クラスター設定で使用できます。

アクティブ/アクティブのクラスター設定

アクティブ/アクティブ設定には、双方向でデータをレプリケーションするアクティブなクラスターが 2 つあります。アプリケーションはいずれかのクラスターを使用できます。各クラスターは同じデータを提供できます。これにより、地理的に異なる場所で同じデータを利用できるようにします。コンシューマーグループは両方のクラスターでアクティブであるため、レプリケーションされたトピックのコンシューマーオフセットはソースクラスターに同期されません。

active/passive クラスター設定

active/passive 設定には、passive クラスターにデータをレプリケーションする active クラスターがあります。passive クラスターはスタンバイのままになります。システムに障害が発生した場合に、データ復旧に passive クラスターを使用できます。

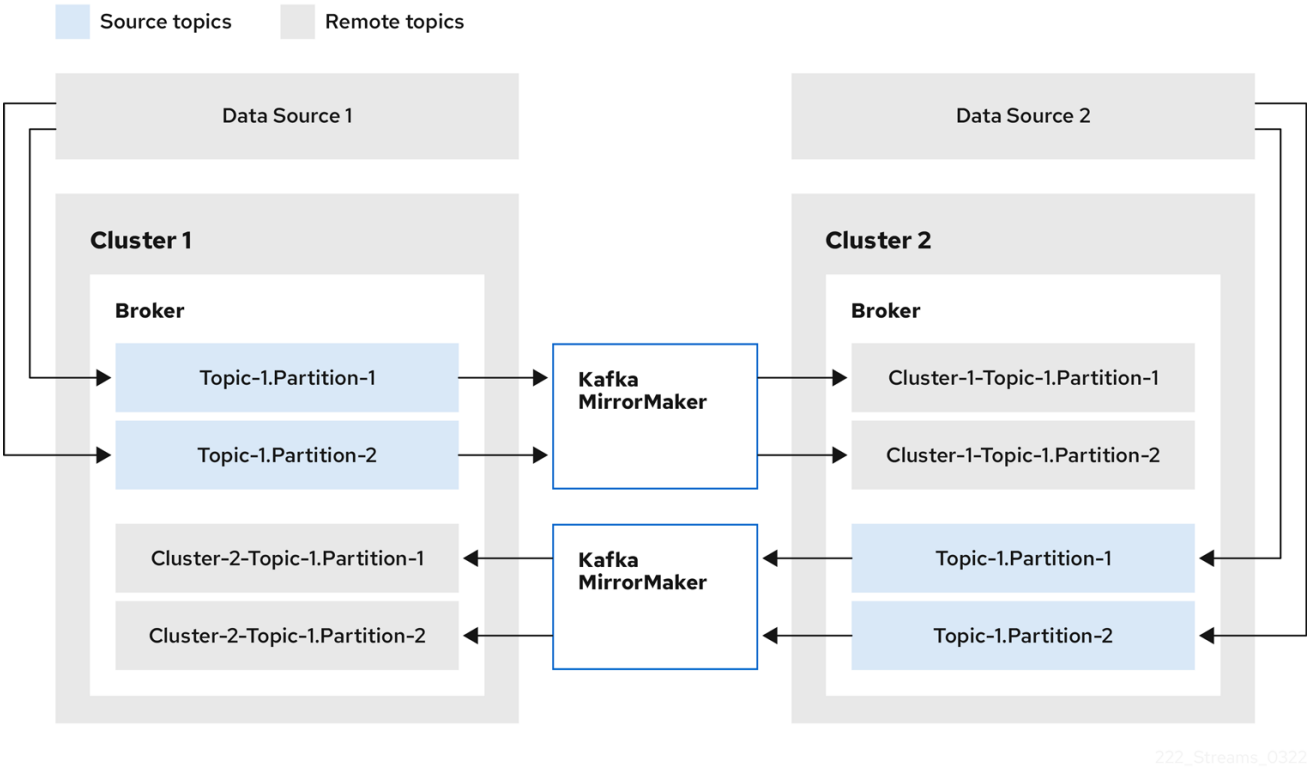
プロデューサーとコンシューマーがアクティブなクラスターのみに接続することを前提とします。MirrorMaker 2 クラスターはターゲットごとに必要です。

9.1.1. 双方向レプリケーション (active/active)

MirrorMaker 2 アーキテクチャーは、**アクティブ/アクティブ** クラスター設定での双方向レプリケーションをサポートします。

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レプリケーションします。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は MirrorMaker 2 によってソースクラスターを表すように自動的に変更されます。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9.1 トピック名の変更



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レプリケーションされません。

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。

9.1.2. 一方向レプリケーション (active/passive)

MirrorMaker 2 アーキテクチャーは、**active/passive** クラスター設定での一方向レプリケーションをサポートします。

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の自動変更は推奨しません。

IdentityReplicationPolicy を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。

9.2. MIRRORMAKER 2 コネクターの設定

Kafka クラスター間のデータの同期を調整する内部コネクターには、MirrorMaker 2 コネクター設定を使用します。

MirrorMaker 2 は次のコネクターで設定されます。

MirrorSourceConnector

ソースコネクターは、トピックをソースクラスターからターゲットクラスターにレプリケーションします。また、ACL をレプリケーションし、**MirrorCheckpointConnector** を実行する必要があります。

MirrorCheckpointConnector

チェックポイントコネクターは定期的にオフセットを追跡します。有効にすると、ソースクラスターとターゲットクラスター間のコンシューマーグループオフセットも同期されます。

MirrorHeartbeatConnector

ハートビートコネクターは、ソースクラスターとターゲットクラスター間の接続を定期的にチェックします。

以下の表は、コネクタープロパティーと、これらを使用するために設定するコネクターについて説明しています。

表9.1 MirrorMaker 2 コネクター設定プロパティー

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
admin.timeout.ms 新規トピックの検出などの管理タスクのタイムアウト。デフォルトは **60000** (1 分) です。	✓	✓	✓
replication.policy.class リモートトピックの命名規則を定義するポリシー。デフォルトは **org.apache.kafka.connect.mirror.DefaultReplicationPolicy** です。	✓	✓	✓
replication.policy.separator ターゲットクラスターのトピックの命名に使用されるセパレーター。デフォルトでは、区切り文字はドット (.) に設定されています。区切り文字の設定は、リモートトピック名を定義する **DefaultReplicationPolicy** レプリケーションポリシークラスにのみ適用されます。トピックは元の名前を保持するため、**IdentityReplicationPolicy** クラスはこのプロパティーを使用しません。	✓	✓	✓
consumer.poll.timeout.ms ソースクラスターをポーリングする際のタイムアウト。デフォルトは **1000** (1 秒) です。	✓	✓	
offset-syncs.topic.location **offset-syncs** トピックの場所。これは、**source** (デフォルト) または **target** クラスターになります。	✓	✓	

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
topic.filter.class レプリケーションするトピックを選択するためのトピックフィルター。デフォルトは **org.apache.kafka.connect.mirror.DefaultTopicFilter** です。	✓	✓	
config.property.filter.class レプリケーションするトピック設定プロパティーを選択するトピックフィルター。デフォルトは **org.apache.kafka.connect.mirror.DefaultConfigPropertyFilter** です。	✓		
config.properties.exclude レプリケーションすべきでないトピック設定プロパティー。コンマ区切りのプロパティー名と正規表現をサポートします。	✓		
offset.lag.max リモートパーティションが同期されるまでの最大許容 (同期外) オフセットラグ。デフォルトは **100** です。	✓		
offset-syncs.topic.replication.factor 内部 **offset-syncs** トピックのレプリケーション係数。デフォルトは **3** です。	✓		
refresh.topics.enabled 新しいトピックおよびパーティションの確認を有効にします。デフォルトは **true** です。	✓		

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
refresh.topics.interval.seconds トピック更新の頻度。デフォルトは **600** (10 分) です。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。頻度は、**refresh.topics.interval.seconds** をソースコネクター設定に追加することで変更できます。	✓		
replication.factor 新しいトピックのレプリケーション係数。デフォルトは **2** です。	✓		
sync.topic.acls.enabled ソースクラスターからの ACL の同期を有効にします。デフォルトは **true** です。詳細は、「ACL ルールの同期」 を参照してください。	✓		
sync.topic.acls.interval.seconds ACL 同期の頻度。デフォルトは **600** (10 分) です。	✓		
sync.topic.configs.enabled ソースクラスターからのトピック設定の同期を有効にします。デフォルトは **true** です。	✓		
sync.topic.configs.interval.seconds トピック設定の同期頻度。デフォルトは **600** (10 分) です。	✓		
checkpoints.topic.replication.factor 内部 **checkpoints** トピックのレプリケーション係数。デフォルトは **3** です。		✓	

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
emit.checkpoints.enabled コンシューマーオフセットをターゲットクラスターに同期できるようにします。デフォルトは **true** です。		✓	
emit.checkpoints.interval.seconds コンシューマーオフセット同期の頻度。デフォルトは **60** (1 分) です。		✓	
group.filter.class レプリケーションするコンシューマーグループを選択するためのグループフィルター。デフォルトは **org.apache.kafka.connect.mirror.DefaultGroupFilter** です。		✓	
refresh.groups.enabled 新規コンシューマーグループの確認を有効にします。デフォルトは **true** です。		✓	
refresh.groups.interval.seconds コンシューマーグループ更新の頻度。デフォルトは **600** (10 分) です。		✓	
sync.group.offsets.enabled ターゲットクラスターの **__consumer_offsets** トピックへのコンシューマーグループオフセットの同期を有効にします。デフォルトは **false** です。		✓	
sync.group.offsets.interval.seconds コンシューマーグループオフセット同期の頻度。デフォルトは **60** (1 分) です。		✓	

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
emit.heartbeats.enabled ターゲットクラスターでの接続性チェックを有効にします。デフォルトは **true** です。			✓
emit.heartbeats.interval.seconds 接続性チェックの頻度。デフォルトは **1** (1 秒) です。			✓
heartbeats.topic.replication.factor 内部 **heartbeats** トピックのレプリケーション係数。デフォルトは **3** です。			✓

9.2.1. コンシューマーグループオフセットの場所の変更トピック

MirrorMaker 2 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。

offset-syncs トピック

offset-syncs トピックは、レプリケーション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。

checkpoints トピック

checkpoints トピックは、各コンシューマーグループでレプリケーション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。

これらは MirrorMaker 2 によって内部的に使用されるため、これらのトピックと直接対話することはありません。

MirrorCheckpointConnector は、オフセット追跡用の **チェックポイント** を発行します。**checkpoints** トピックのオフセットは、設定によって事前に決定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。

offset-syncs トピックの場所は、デフォルトで **source** クラスターです。**offset-syncs.topic.location** コネクター設定を使用して、これを **target** クラスターに変更することができます。トピックが含まれるクラスターへの読み取り/書き込みアクセスが必要です。ターゲットクラスターを **offset-syncs** トピックの場所として使用すると、ソースクラスターへの読み取りアクセス権しかない場合でも、MirrorMaker 2 を使用できるようになります。

9.2.2. 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を passive (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。

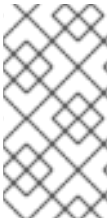
トピックオフセットの同期を使用するには、**sync.group.offsets.enabled** を checkpoint コネクター設定に追加し、プロパティーを **true** に設定して、同期を有効にします。同期はデフォルトで無効になっています。

ソースコネクターで **IdentityReplicationPolicy** を使用する場合は、チェックポイントコネクター設定でも設定する必要があります。これにより、ミラーリングされたコンシューマーオフセットが正しいトピックに適用されます。

コンシューマーオフセットは、ターゲットクラスターでアクティブではないコンシューマーグループに対してのみ同期されます。コンシューマーグループがターゲットクラスターにある場合、Synchronization を実行できず、**UNKNOWN_MEMBER_ID** エラーが返されます。

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。この頻度は、**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ことで変更できます。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のチェック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 passive 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。



注記

Java で作成されたアプリケーションがある場合は、**RemoteClusterUtils.java** ユーティリティーを使用して、アプリケーションを通じてオフセットを同期できます。ユーティリティーは、**checkpoints** トピックからコンシューマーグループのリモートオフセットを取得します。

9.2.3. ハートビートコネクターを使用するタイミングの決定

ハートビートコネクターはハートビートを出力して、ソース Kafka クラスターとターゲット Kafka クラスター間の接続を確認します。内部 **heartbeat** トピックはソースクラスターからレプリケートされます。つまり、ハートビートコネクターがソースクラスターに接続されている必要があります。**heartbeat** トピックはターゲットクラスターに配置されているため、次のことが可能になります。

- データのミラーリング元のすべてのソースクラスターを特定します。
- ミラーリングプロセスの稼働状況と遅延を確認する

これは、プロセスが何らかの理由でスタックしたり停止したりしていないことを確認するのに役立ちます。ハートビートコネクターは、Kafka クラスター間のミラーリングプロセスを監視するための貴重なツールですが、必ずしも使用する必要があるわけではありません。たとえば、デプロイメントのネットワーク遅延が低い場合、またはトピックの数が少ない場合は、ログメッセージやその他の監視ツールを使用してミラーリングプロセスを監視することが推奨されます。ハートビートコネクターを使用しない場合は、MirrorMaker 2 設定からハートビートコネクターを省略してください。

9.2.4. MirrorMaker 2 コネクターの設定の調整

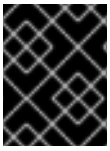
MirrorMaker 2 コネクターが正しく動作することを確認するには、コネクター全体で特定の設定を調整してください。具体的には、次のプロパティーが該当するすべてのコネクターで同じ値であることを確認してください。

- **replication.policy.class**
- **replication.policy.separator**
- **offset-syncs.topic.location**
- **topic.filter.class**

たとえば、**replication.policy.class** の値は、ソース、チェックポイント、およびハートビートコネクターで同じである必要があります。設定が一致していないか欠落していると、データレプリケーションやオフセット同期で問題が発生するため、関連するすべてのコネクターを同じ設定にしておくことが重要です。

9.3. コネクタープロデューサーおよびコンシューマーの設定

MirrorMaker 2 コネクターは、内部プロデューサーとコンシューマーを使用します。必要に応じて、これらのプロデューサーおよびコンシューマーを設定して、デフォルト設定を上書きできます。



重要

プロデューサおよびコンシューマーの設定オプションは MirrorMaker 2 の実装に依存しており、変更される可能性があります。

プロデューサーとコンシューマーの設定は、**すべての** コネクターに適用されます。**config/connect-mirror-maker.properties** ファイルで設定を指定します。

プロパティーファイルを使用して、プロデューサーとコンシューマーのデフォルト設定を次の形式でオーバーライドします。

- **<source_cluster_name>.consumer.<property>**
- **<source_cluster_name>.producer.<property>**
- **<target_cluster_name>.consumer.<property>**
- **<target_cluster_name>.producer.<property>**

次の例は、プロデューサーとコンシューマーを設定する方法を示しています。プロパティーはすべてのコネクターに対して設定されますが、一部の設定プロパティーは特定のコネクターにのみ関連します。

コネクターのプロデューサーとコンシューマーの設定例

```
clusters=cluster-1,cluster-2

# ...
cluster-1.consumer.fetch.max.bytes=52428800
cluster-2.producer.batch.size=327680
cluster-2.producer.linger.ms=100
cluster-2.producer.request.timeout.ms=30000
```

9.4. タスクの最大数を指定

コネクターは、Kafka にデータを出し入れするタスクを作成します。各コネクターは、タスクを実行するワーカー Pod のグループ全体に分散される 1 つ以上のタスクで設定されます。タスクの数を増やすと、多数のパーティションをレプリケーションするとき、または多数のコンシューマーグループのオフセットを同期するときのパフォーマンスの問題に役立ちます。

タスクは並行して実行されます。ワーカーには 1 つ以上のタスクが割り当てられます。1 つのタスクが 1 つのワーカー Pod によって処理されるため、タスクよりも多くのワーカー Pod は必要ありません。ワーカーよりも多くのタスクがある場合、ワーカーは複数のタスクを処理します。

tasksMax プロパティーを使用して、MirrorMaker 設定でコネクタータスクの最大数を指定できます。タスクの最大数を指定しない場合、デフォルト設定のタスク数は 1 つです。

ハートビートコネクターは常に単一のタスクを使用します。

ソースおよびチェックポイントコネクターに対して開始されるタスクの数は、可能なタスクの最大数と **tasks.max** の値の間の低い方の値です。ソースコネクターの場合、可能なタスクの最大数は、ソースクラスターからレプリケーションされるパーティションごとに 1 つです。チェックポイントコネクターの場合、可能なタスクの最大数は、ソースクラスターからレプリケーションされるコンシューマーグループごとに 1 つです。タスクの最大数を設定するときは、プロセスをサポートするパーティションの数とハードウェアリソースを考慮してください。

インフラストラクチャーが処理のオーバーヘッドをサポートしている場合、タスクの数を増やすと、スループットと待機時間が向上する可能性があります。たとえば、タスクを追加すると、多数のパーティションまたはコンシューマーグループがある場合に、ソースクラスターのポーリングにかかる時間が短縮されます。

MirrorMaker コネクターの tasks.max 設定

```
clusters=cluster-1,cluster-2
# ...
tasks.max = 10
```

デフォルトでは、MirrorMaker 2 は 10 分ごとに新しいコンシューマーグループをチェックします。**refresh.groups.interval.seconds** 設定を調整して、頻度を変更できます。低く調整するときは注意してください。より頻繁なチェックは、パフォーマンスに悪影響を及ぼす可能性があります。

9.5. 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。

9.6. MIRRORMAKER 2 を専用モードで実行する

MirrorMaker 2 を使用して、設定を通じて Kafka クラスター間でデータを同期します。この手順では、専用の単一ノード MirrorMaker 2 クラスターを設定して実行する方法を示します。専用クラスターは、Kafka Connect ワーカーノードを使用して、Kafka クラスター間でデータをミラーリングします。

注記

MirrorMaker 2 を分散モードで実行することも可能です。MirrorMaker 2 は、専用モードと分散モードの両方でコネクターとして動作します。専用の MirrorMaker クラスターを実行する場合、コネクターは Kafka Connect クラスターで設定されます。結果として、Kafka Connect クラスターへの直接アクセス、追加のコネクターの実行、および REST API の使用が可能になります。詳細は、Apache Kafka のドキュメント を参照してください。

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
  - クラスターからクラスターへ
  - トピックからトピックへ
- 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に設定を作成し、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によって MirrorMaker 2 を実装する方法について説明します。

ソースクラスターからレプリケーションするトピックおよびコンシューマーグループを指定できます。ソースおよびターゲットクラスターの名前を指定し、レプリケーションするトピックとコンシューマーグループを指定します。

以下の例では、クラスター 1 から 2 のレプリケーションに、トピックとコンシューマーグループが指定されます。

特定のトピックおよびコンシューマーグループをレプリケーションする設定例

```
clusters=cluster-1,cluster-2
cluster-1->cluster-2.topics = topic-1, topic-2
cluster-1->cluster-2.groups = group-1, group-2
```

名前のリストを指定したり、正規表現を使用したりできます。デフォルトでは、これらのプロパティーを設定しないと、すべてのトピックおよびコンシューマーグループがレプリケーションされます。**.*** を正規表現として使用し、すべてのトピックおよびコンシューマーグループをレプリケーションすることもできます。ただし、クラスターに不要な負荷が余分にかかるのを避けるため、必要なトピックとコンシューマーグループのみを指定するようにしてください。

作業を開始する前に

設定プロパティーファイルの例は **./config/connect-mirror-maker.properties** にあります。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. テキストエディターでサンプルプロパティーファイルを開くか、新しいプロパティーファイルを作成し、ファイルを編集して接続情報と各 Kafka クラスターのレプリケーションフローを追加します。
   以下の例は、**cluster-1** および **cluster-2** の 2 つのクラスターを双方向に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できます。

   MirrorMaker 2 の設定例

```
clusters=cluster-1,cluster-2 (1)

cluster-1.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_one>: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-<project_name_two>: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-2->cluster-1.enabled=true (7)
cluster-1->cluster-2.topics=.* (8)
cluster-2->cluster-1.topics=topic-1, topic-2 (9)
cluster-1->cluster-2.groups=.* (10)
cluster-2->cluster-1.groups=group-1, group-2 (11)

replication.policy.separator=- (12)
sync.topic.acls.enabled=false (13)
refresh.topics.interval.seconds=60 (14)
refresh.groups.interval.seconds=60 (15)
```

(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。両方のクラスターはポート 443 を使用し、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** の接続情報です。

(5) **ssl.** プロパティーは、**cluster-2** の TLS 設定を定義します。

(6) **cluster-1** から **cluster-2** へのレプリケーションフローが有効になっています。

(7) **cluster-2** から **cluster-1**. へのレプリケーションフローが有効になっています。

(8) **cluster-1** から **cluster-2** へのすべてのトピックのレプリケーションです。ソースコネクターは指定のトピックをレプリケーションします。チェックポイントコネクターは、指定

されたトピックのオフセットを追跡します。

9 cluster-2 から cluster-1 への特定のトピックのレプリケーション。

10 cluster-1 から cluster-2 へのすべてのコンシューマーグループのレプリケーション。チェックポイントコネクターは、指定されたコンシューマーグループをレプリケーションします。

11 cluster-2 から cluster-1 への特定のコンシューマーグループのレプリケーション。

12 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。

13 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。

14 新しいトピックの同期をチェックする間隔です。

15 新しいコンシューマーグループの同期をチェックする間隔です。

2. オプション: 必要に応じて、リモートトピックの名前の自動変更をオーバーライド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=org.apache.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、管理します。

```
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は、これらのチェックの頻度も調整することを推奨します。

4. ターゲットクラスターで Kafka を起動します。

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

5. プロパティーファイルで定義したクラスター接続設定および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 \
/opt/kafka/config/connect-mirror-maker.properties
```

MirrorMaker はクラスター間の接続を設定します。

6. ターゲットクラスターごとに、トピックがレプリケーションされていることを確認します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --list
```

9.7. (非推奨) レガシーモードでの MIRRORMAKER 2 の使用

この手順では、MirrorMaker 2 をレガシーモードで使用するように設定する方法について説明します。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 を実行できます。

重要

Kafka MirrorMaker 1 (ドキュメントでは単に **MirrorMaker** と呼ばれる) は Apache Kafka 3.0.0 で非推奨となり、Apache Kafka 4.0.0 で削除されます。そのため、Kafka MirrorMaker 1 は Streams for Apache Kafka でも非推奨になりました。Apache Kafka 4.0.0 を導入すると、Kafka MirrorMaker 1 は Streams for Apache Kafka から削除されます。代わりに、**IdentityReplicationPolicy** を備えた MirrorMaker 2 を使用してください。

前提条件

現時点でレガシーバージョンの MirrorMaker と使用しているプロパティーファイルが必要である。

- **/opt/kafka/config/consumer.properties**
- **/opt/kafka/config/producer.properties**

手順

1. MirrorMaker **consumer.properties** ファイルと **producer.properties** ファイルを編集して、MirrorMaker 2 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) MirrorMaker の以前のバージョンをエミュレートします。

(2) MirrorMaker 2 の機能 (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) が無効になります

2. 変更を保存し、以前のバージョンの MirrorMaker で使用していたプロパティーファイルで 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** プロパティーはターゲットクラスターの設定を提供します。

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 Kafka を起動します。

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

4. ターゲットクラスターの場合、トピックがレプリケーションされていることを確認します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_host>:<port> --list
```

第10章 KAFKA コンポーネントのログの設定

Kafka コンポーネントのログレベルを設定プロパティーで直接設定します。Kafka ブローカー、Kafka Connect、および MirrorMaker 2 のブローカーレベルを動的に変更することもできます。

INFO から DEBUG など、ログレベルの詳細を増やすと、Kafka クラスターのトラブルシューティングに役立ちます。ただし、ログが冗長になると、パフォーマンスに悪影響が生じ、問題の診断が難しくなる可能性があります。

10.1. KAFKA ログプロパティーの設定

Kafka コンポーネントは、エラーログに Log4j フレームワークを使用します。デフォルトでは、ログ設定は、次のプロパティーファイルを使用してクラスパスまたは **config** ディレクトリーから読み取られます。

- Kafka の **log4j.properties**
- Kafka Connect および MirrorMaker 2 の **connect-log4j.properties**

これらが明示的に設定されていない場合、ロガーは各ファイルの **log4j.rootLogger** ログレベル設定を継承します。これらのファイルのログレベルを変更できます。他のロガーのログレベルを追加および設定することもできます。

コンポーネントの起動スクリプトで使用される **KAFKA_LOG4J_OPTS** 環境変数を使用して、ログプロパティーファイルの場所と名前を変更できます。

Kafka ノードで使用されるログプロパティーファイルの名前と場所を渡す

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

Kafka Connect で使用されるログプロパティーファイルの名前と場所を渡す

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

MirrorMaker 2 で使用されるログプロパティーファイルの名前と場所を渡す

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

10.2. KAFKA ブローカーロガーのログレベルの動的な変更

Kafka ブローカーのログは、各ブローカー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、実行時にブローカーロガーのログレベルを動的に変更します。

ブローカーロガーをデフォルトのロ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

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Kafka が実行中である。

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のリストを表示します。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --describe --entity-type broker-loggers --entity-name BROKER-ID
```

たとえば、ブローカー **0** の場合:

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

これにより、**TRACE**、**DEBUG**、**INFO**、**WARN**、**ERROR**、または **FATAL** の各ロガーのログレベルが返されます。

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のコンマ区切りリストとして指定します。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_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:

```
/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、以下が返されます。

```
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のコンマ区切りリストとして指定します。

```
/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 Updating Broker Configs

10.3. KAFKA CONNECT と MIRRORMAKER 2 のログレベルを動的に変更する

再起動することなく、実行時に Kafka Connect ワーカーまたは MirrorMaker 2 コネクターのログレベルを動的に変更します。

Kafka Connect API を使用して、ワーカーまたはコネクターロガーのログレベルを一時的に変更します。Kafka Connect API は、ログレベルを取得または変更するための **admin/loggers** エンドポイントを提供します。API を使用してログレベルを変更しても、**connect-log4j.properties** 設定ファイル内のロガー設定は変更されません。必要に応じて、設定ファイル内のログレベルを永続的に変更できます。

注記

MirrorMaker 2 のログレベルは、分散モードまたはスタンドアロンモードの実行時にのみ変更できます。専用の MirrorMaker 2 クラスターには Kafka Connect REST API がないため、ログレベルを変更することはできません。

Kafka Connect API のデフォルトのリスナーはポート 8083 上にあり、この手順で使用されます。**admin.listeners** 設定を使用して、リスナーを変更または追加したり、TLS 認証を有効にしたりできます。

admin エンドポイントのリスナー設定の例

```
admin.listeners=https://localhost:8083
admin.listeners.https.ssl.truststore.location=/path/to/truststore.jks
admin.listeners.https.ssl.truststore.password=123456
admin.listeners.https.ssl.keystore.location=/path/to/keystore.jks
admin.listeners.https.ssl.keystore.password=123456
```

admin エンドポイントを使用可能にしない場合は、設定で空の文字列を指定して無効にすることができます。

admin エンドポイントを無効にするリスナー設定の例

```
admin.listeners=
```

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Kafka が実行中である。
- Kafka Connect または MirrorMaker 2 が実行中である。

手順

1. **kafka** ユーザーに切り替えます。

```
su - kafka
```

2. **connect-log4j.properties** ファイルで設定されているロガーの現在のログレベルを確認します。

```
$ cat /opt/kafka/config/connect-log4j.properties

# ...
log4j.rootLogger=INFO, stdout, connectAppender
# ...
log4j.logger.org.reflections=ERROR
```

curl コマンドを使用して、Kafka Connect API の **admin/loggers** エンドポイントからログレベルを確認します。

```
curl -s http://localhost:8083/admin/loggers/ | jq

{
  "org.reflections": {
    "level": "ERROR"
  },
  "root": {
    "level": "INFO"
  }
}
```

jq は出力を JSON 形式で出力します。このリストには、標準の **org** および **root** レベルのロガーに加えて、ログレベルが変更された特定のロガーが表示されます。

Kafka Connect の **admin.listeners** 設定に TLS (トランスポート層セキュリティー) 認証を設定する場合、ロガーエンドポイントのアドレスは、**https://localhost:8083** など、https としてプロトコルを使用して **admin.listeners** に指定された値になります。

特定のロガーのログレベルを取得することもできます。

```
curl -s
http://localhost:8083/admin/loggers/org.apache.kafka.connect.mirror.MirrorCheckpointConnecto
r | jq

{
  "level": "INFO"
}
```

3. PUT メソッドを使用して、ロガーのログレベルを変更します。

```
curl -Ss -X PUT -H 'Content-Type: application/json' -d '{"level": "TRACE"}'
http://localhost:8083/admin/loggers/root

{
  # ...

  "org.reflections": {
```

```
    "level": "TRACE"
  },
  "org.reflections.Reflections": {
    "level": "TRACE"
  },
  "root": {
    "level": "TRACE"
  }
}
```

root ロガーを変更すると、デフォルトでルートログレベルを使用していたロガーのログレベルも変更されます。

第11章 KAFKA STATIC QUOTA プラグインを使用したブローカーへの制限の設定

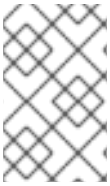
重要

Kafka Static Quota プラグイン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機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、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、テクノロジープレビュー機能のサポート範囲 を参照してください。

Kafka Static Quota プラグインを使用して、Kafka クラスターのブローカーにスループットおよびストレージの制限を設定します。Kafka 設定ファイルにプロパティーを追加して、プラグインを有効にし、制限を設定します。バイトレートのしきい値およびストレージクォータを設定して、ブローカーと対話するクライアントに制限を設けることができます。

プロデューサーおよびコンシューマー帯域幅にバイトレートのしきい値を設定できます。制限の合計は、ブローカーにアクセスするすべてのクライアントに分散されます。たとえば、バイトレートのしきい値として 40 MBps ををプロデューサーに設定できます。2 つのプロデューサーが実行されている場合、それぞれのスループットは 20MBps に制限されます。

ストレージクォータは、Kafka ディスクストレージの制限をソフト制限とハード制限間で調整します。この制限は、利用可能なすべてのディスク容量に適用されます。プロデューサーは、ソフト制限とハード制限の間で徐々に遅くなります。制限により、ディスクの使用量が急激に増加しないようにし、容量を超えないようにします。ディスクがいっぱいになると、修正が難しい問題が発生する可能性があります。ハード制限は、ストレージの上限です。



注記

JBOD ストレージの場合、制限はすべてのディスクに適用されます。ブローカーが 2 つの 1 TB ディスクを使用し、クォータが 1.1 TB の場合は、1 つのディスクにいっぱいになり、別のディスクがほぼ空になることがあります。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。

手順

1. Kafka 設定プロパティーファイルを編集します。
   プラグインプロパティーは、この設定例のとおりです。

   Kafka Static Quota プラグインの設定例

```
# ...
client.quota.callback.class=io.strimzi.kafka.quotas.StaticQuotaCallback (1)
client.quota.callback.static.produce=1000000 (2)
client.quota.callback.static.fetch=1000000 (3)
client.quota.callback.static.storage.soft=400000000000 (4)
```

```
client.quota.callback.static.storage.hard=500000000000 5
client.quota.callback.static.storage.check-interval=5 6
# ...
```

1. Kafka Static Quota プラグインを読み込みます。
2. プロデューサーのバイトレートしきい値を設定します。この例では 1 MBps です。
3. コンシューマーのバイトレートしきい値を設定します。この例では 1 MBps です。
4. ストレージのソフト制限の下限を設定します。この例では 400 GB です。
5. ストレージのハード制限の上限を設定します。この例では 500 GB です。
6. ストレージのチェックの間隔 (秒単位) を設定します。この例では 5 秒です。これを 0 に設定するとチェックを無効にできます。

2. デフォルトの設定ファイルで Kafka ブローカーを起動します。

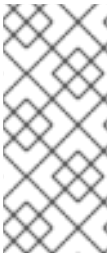
```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/kraft/server.properties
```

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
```

第12章 ブローカーの追加または削除によるクラスターのスケーリング

ブローカーを追加して Kafka クラスターをスケーリングすると、クラスターのパフォーマンスと信頼性が向上します。ブローカーを追加すると、利用可能なリソースが増加し、クラスターがより大きなワークロードを処理し、より多くのメッセージを処理できるようになります。また、より多くのレプリカとバックアップを提供することでフォールトトレランスを向上させることもできます。逆に、十分に活用されていないブローカーを削除すると、リソースの消費が削減され、効率が向上します。中断やデータ損失を避けるために、スケーリングは慎重に行う必要があります。クラスター内のすべてのブローカーにパーティションを再分散することにより、各ブローカーのリソース使用率が削減され、クラスターの全体的なスループットが向上します。



注記

Kafka トピックのスループットを向上させるには、そのトピックのパーティションの数を増やすことができます。これにより、トピックの負荷をクラスター内の異なるブローカー間で共有できるようになります。ただし、すべてのブローカーが特定のリソース (I/O など) によって制約されている場合、パーティションを追加してもスループットは向上しません。この場合、クラスターにブローカーをさらに追加する必要があります。

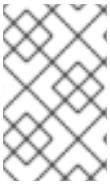
マルチノード Kafka クラスターの実行時にブローカーを追加すると、レプリカとして機能するクラスター内のブローカーの数に影響します。トピックの実際のレプリケーション係数は、**default.replication.factor** および **min.insync.replicas** の設定、および使用可能なブローカーの数によって決まります。たとえば、レプリケーション係数 3 は、トピックの各パーティションが 3 つのブローカー間でレプリケーションされ、ブローカーに障害が発生した場合のフォールトトレランスを確保することを意味します。

レプリカ設定の例

```
default.replication.factor = 3
min.insync.replicas = 2
```

ブローカーを追加または削除しても、Kafka はパーティションを自動的に再割り当てしません。これを行う最良の方法は、Cruise Control を使用することです。クラスターをスケールアップまたはスケールダウンするときに、Cruise Control の **add-brokers** モードと **remove-brokers** モードを使用できます。

- Kafka クラスターをスケールアップした後、**add-brokers** モードを使用して、パーティションレプリカを既存のブローカーから新しく追加したブローカーに移動します。
- Kafka クラスターをスケールダウンする前に、**remove-brokers** モードを使用して、削除されるブローカーからパーティションレプリカを移動します。



注記

ブローカーをスケールダウンする場合、クラスターから削除する特定の Pod を指定することはできません。代わりに、ブローカーの削除プロセスは、最も大きい番号の Pod から開始されます。

第13章 CRUISE CONTROL を使用したクラスターのリバランス

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再分散、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。Cruise Control は Load Monitor、Analyzer、Anomaly Detector、および Executor の主な 4 つのコンポーネントと、クライアントの対話に使用される REST API で設定されます。

Cruise Control を使用して Kafka クラスターを **リバランス** できます。Red Hat Enterprise Linux 上の Cruise Control for Apache Kafka は、個別の zip 形式のディストリビューションとして提供されます。

Streams for Apache Kafka は REST API を使用して、以下の Cruise Control 機能をサポートします。

- 最適化ゴールから最適化プロポーザルを生成します。
- 最適化プロポーザルを基にして Kafka クラスターのリバランスを行います。

最適化ゴール

最適化ゴールは、リバランスから達成する特定のゴールを表します。たとえば、トピックのレプリカをブローカー間でより均等に分散することがゴールになる場合があります。設定から追加するゴールを変更できます。ゴールは、ハードゴールまたはソフトゴールとして定義されます。Cruise Control 展開設定を使用してハード目標を追加できます。また、これらの各カテゴリーに適合するメイン、デフォルト、およびユーザー提供の目標もあります。

- **ハードゴール** は事前設定されており、最適化プロポーザルが正常に実行されるには満たされる必要があります。
- 最適化プロポーザルが正常に実行されるには、**ソフトゴール** を満たす必要はありません。これは、すべてのハードゴールが一致することを意味します。
- **メインゴール** は Cruise Control から継承されます。ハードゴールとして事前設定されているものもあります。メインゴールは、デフォルトで最適化プロポーザルで使用されます。
- **デフォルトのゴール** は、デフォルトでメインゴールと同じです。デフォルトゴールのセットを指定できます。
- **ユーザー提供** のゴールは、特定の最適化プロポーザルを生成するために設定されるデフォルトゴールのサブセットです。

最適化プロポーザル

最適化プロポーザルは、リバランスから達成するゴールで構成されます。最適化プロポーザルを生成して、提案された変更の概要と、リバランス可能な結果を作成します。ゴールは特定の優先順位で評価されます。その後、プロポーザルの承認または拒否を選択できます。プロポーザルを拒否し、調整したゴールセットを使用して再度実行できます。
次の API エンドポイントのいずれかにリクエストを送信することで、最適化の提案を生成して承認できます。

- **/rebalance** エンドポイントで完全なリバランスを実行します。
- **/add_broker** エンドポイントは、Kafka クラスターをスケールアップするときにブローカーを追加した後に再調整します。
- **/remove_broker** エンドポイントを再調整してから、Kafka クラスターをスケールダウンするときにブローカーを削除します。

最適化ゴールは、設定プロパティーファイルで設定します。Streams for Apache Kafka は、Cruise Control のプロパティーファイルの例を提供します。

13.1. CRUISE CONTROL のコンポーネントと機能

Cruise Control は、Load Monitor、Analyzer、Anomaly Detector、Executor の 4 つの主要コンポーネントと、クライアントとの対話用の REST API で設定されています。Streams for Apache Kafka は REST API を使用して、以下の Cruise Control 機能をサポートします。

- 最適化ゴールから最適化プロポーザルを生成します。
- 最適化プロポーザルを基にして Kafka クラスターのリバランスを行います。

最適化ゴール

最適化ゴールは、リバランスから達成する特定のゴールを表します。たとえば、トピックのレプリカをブローカー間でより均等に分散することがゴールになる場合があります。設定から追加するゴールを変更できます。ゴールは、ハードゴールまたはソフトゴールとして定義されます。Cruise Control 展開設定を使用してハード目標を追加できます。また、これらの各カテゴリーに適合するメイン、デフォルト、およびユーザー提供の目標もあります。

- **ハードゴール** は事前設定されており、最適化プロポーザルが正常に実行されるには満たされる必要があります。
- 最適化プロポーザルが正常に実行されるには、**ソフトゴール** を満たす必要はありません。これは、すべてのハードゴールが一致することを意味します。
- **メインゴール** は Cruise Control から継承されます。ハードゴールとして事前設定されているものもあります。メインゴールは、デフォルトで最適化プロポーザルで使用されます。
- **デフォルトのゴール** は、デフォルトでメインゴールと同じです。デフォルトゴールのセットを指定できます。
- **ユーザー提供のゴール** は、特定の最適化プロポーザルを生成するために設定されるデフォルトゴールのサブセットです。

最適化プロポーザル

最適化プロポーザルは、リバランスから達成するゴールで構成されます。最適化プロポーザルを生成して、提案された変更の概要と、リバランス可能な結果を作成します。ゴールは特定の優先順位で評価されます。その後、プロポーザルの承認または拒否を選択できます。プロポーザルを拒否し、調整したゴールセットを使用して再度実行できます。
3 つのモードのいずれかで最適化プロポーザルを生成できます。

- **full** はデフォルトのモードで、完全なリバランスを実行します。
- **add-brokers** は、Kafka クラスターをスケールアップするときにブローカーを追加した後に使用するモードです。
- **remove-brokers** は、Kafka クラスターを縮小するときにブローカーを削除する前に使用するモードです。

自己修復、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 の機能は現在サポートされていません。

関連情報

- Cruise Control のドキュメント

13.2. CRUISE CONTROL のダウンロード

Cruise Control の ZIP ファイル配布は、Red Hat Web サイトからダウンロードできます。Red Hat Streams for Apache Kafka の最新バージョンは、Streams for Apache Kafka software downloads ページからダウンロードできます。

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 Streams for Apache Kafka 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 ZIP ファイルの中身を新しいディレクトリーに展開します。

```
unzip amq-streams-<version>-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
```

13.3. CRUISE CONTROL METRICS REPORTER のデプロイ

Cruise Control を起動する前に、提供される 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。Metrics Reporter のファイルは、Streams for Apache Kafka インストールアーティファクトで提供されます。

実行時に読み込まれると、Metrics Reporter は 自動作成された 3 つのトピックの 1 つである **__CruiseControlMetrics** トピックにメトリクスを送信します。Cruise Control はこ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計算します。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。

手順

Kafka クラスターの各ブローカーに対して、以下を 1 つずつ実行します。

1. Kafka ブローカーを停止します。

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

2. Kafka 設定プロパティーファイルを編集して、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 Reporters 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および値を追加します。

```
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、**__CruiseControlMetrics** トピックをログクリーンアップポリシー **DELETE** で作成します。詳細は、自動作成されたトピック および Cruise Control Metrics トピックのログクリーンアップポリシー を参照してください。

3. 必要に応じて SSL を設定します。

a. Kafka 設定プロパティーファイルでは、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、**cruise.control.metrics.reporter** という接頭辞を持つ、すべての標準のプロデューサー固有の設定プロパティーを受け入れます。たとえば、**cruise.control.metrics.reporter.ssl.truststore.password** です。

b. Cruise Control 設定ファイル (**/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 サーバーとの間の 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し、すべての Cruise Control サーバークライアントにこのプロパティーを使用します。

4. Kafka ブローカーを再起動します。

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

マルチノードクラスターでブローカーを再起動する方法は、「Kafka ブローカーの正常なローリング再起動の実行」 を参照してください。

5. 残りのブローカーで手順 1-5 を繰り返します。

13.4. CRUISE CONTROL の設定および起動

Cruise Control が使用するプロパティーを設定し、**kafka-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 クラスター全体の単一のマシンでホストされます。

Cruise Control の起動時に 3 つのトピックが自動作成されます。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

- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。
- Cruise Control をダウンロード した。
- Cruise Control Metrics Reporter をデプロイ した。

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. 以下の設定例のように、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 main 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)

#Set failure detection to true
kafka.broker.failure.detection.enable=true (8)
```

(1) Kafka ブローカーのホストおよびポート番号 (常にポート 9092)。

(2) Kafka メトリックサンプルストアトピックのレプリケーション係数。シングルノードの Kafka 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。多くの主要な最適化ゴール (5 を参照) は、デフォルトの最適化ゴールとしてすでに設定されています。必要に応じて、目標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 を使用した、主な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ゴールが使用されないように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。主な最適化ゴールのうち 7 つは、すでに厳しい目標として設定されています。必要に応じて、目標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 Cruise Control が Kafka API を使用してブローカーの障害を検出できるようにします。

3. Cruise Control サーバーを起動します。デフォルトでは、サーバーはポート 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/
./kafka-cruise-control-start.sh config/cruisecontrol.properties <port_number>
```

4. Cruise Control が実行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 -X GET 'http://<cc_host>:<cc_port>/kafkacruisecontrol/state'
```

自動作成されたトピック

以下の表は、Cruise Control の起動時に自動的に作成される 3 つのトピックを表しています。このトピックは、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13.1 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 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。
__KafkaCruiseControlPartitionMetricSamples	Cruise Control	各パーティションの派生されたメトリックを格納します。これは 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が **無効** になっていることを確認するには、「Cruise Control Metrics Reporter のデプロイ」 の説明に従って Cruise Control Metrics Reporter を設定するようにしてください。ログコンパクションは、Cruise Control が必要とするレコードを削除し、適切に動作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

13.5. 最適化ゴールの概要

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。Cruise Control は Kafka クラスターの再分散を行うために、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。

13.5.1. 優先度によるゴールの順序

Red Hat Enterprise Linux の Streams for Apache Kafka 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. 一連のトピック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. 優先リーダーエレクション
14. Kafka Assigner のディスク使用率の分散
15. ブローカー内のディスク容量
16. ブローカー内のディスク使用率

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。

13.5.2. Cruise Control プロパティーファイルのゴール設定

最適化ゴールの設定は、**cruise-control/config/** ディレクトリー内の **cruisecontrol.properties** ファイルで行います。Cruise Control には、満たなければならない厳しい最適化ゴールのほか、メイン、デフォルト、およびユーザーが指定した最適化ゴールの設定があります。

次の設定では、次のタイプの最適化ゴールを指定できます。

- **メインゴール** – **cruisecontrol.properties** ファイル
- **ハードゴール** – **cruisecontrol.properties** ファイル

- **デフォルトゴール** – **cruisecontrol.properties** ファイル
- **ユーザー提供のゴール** – 実行時パラメーター

オプションで、ユーザー提供 の最適化ゴールは、実行時に **/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。

13.5.3. ハードおよびソフト最適化ゴール

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** ゴールと解釈できます。これらは、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、ユーザーには送信されません。

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。

Cruise Control では、以下の メイン最適化ゴール が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、**cruisecontrol.properties** ファイルの **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。

13.5.4. メイン最適化ゴール

メイン最適化ゴールはすべてのユーザーが使用できます。メイン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。

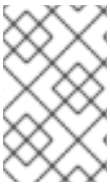
次に示す主な最適化 **goal** は、**cruisecontrol.properties** ファイルの goal プロパティーに優先順位の降順で事前設定されています。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; ReplicaDistributionGoal;
PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

複雑さを軽減するために、最適化の提案を生成するために使用される 1 つ以上のゴールを完全に除外する必要がない限り、事前設定されたメインの最適化ゴールを変更しないことを推奨します。必要な場合、メイン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。

事前設定された主な最適化ゴールを変更するには、目標のリストを **ゴール** プロパティーに優先度の高い順に指定します。**cruisecontrol.properties** ファイルに記載されているように、完全修飾ドメイン名を使用します。

主なゴールを少なくとも 1 つ指定する必要があります。そうしないと、Cruise Control がクラッシュします。



注記

事前設定されたメインの最適化ゴールを変更する場合は、設定された **hard.goals** が、設定したメインの最適化ゴールのサブセットであることを確認する必要があります。そうしないと、最適化プロポーザルの生成時にエラーが発生します。

13.5.5. 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 を設定すると、デフォルトの最適化ゴールを実行時に上書きできます。

次のデフォルトの最適化ゴールは、**cruisecontrol.properties** ファイルの **default.goals** プロパティーに優先順位の降順で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
```

デフォルトのゴールを 1 つ以上指定する必要があります。そうしないと、Cruise Control がクラッシュします。

デフォルトの最適化ゴールを変更するには、**default.goals** プロパティーで目標のリストを優先度の高い順に指定します。デフォルトのゴールは、主要な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。

13.5.6. 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、**/rebalance** エンドポイントへの HTTP リクエストのパラメーターとして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布を最適化したい場合があります。そのため、**/rebalance** エンドポイントに、リーダーレプリカディストリビューションの単一のゴールが含まれる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。そうしないと、エラーが発生します。
- 主な最適化ゴール のサブセットとなる

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

13.6. 最適化プロポーザルの概要

最適化プロポーザル は、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することで、Kafka クラスターの負荷をより均等にするために提案された変更の概要です

各最適化プロポーザルは、それを生成するために使用された 最適化ゴール のセットに基づいており、ブローカーリソースに設定された 容量制限 が適用されます。

すべての最適化プロポーザルは、提案されたリバランスの影響の **見積もり** です。提案は、承認または却下できます。最初に最適化プロポーザルを生成しなければに、クラスターのリバランスは承認できません。

以下のエンドポイントのいずれかを使用して最適化プロポーザルを実行できます。

- **/rebalance**
- **/add_broker**
- **/remove_broker**

13.6.1. エンドポイントのリバランス

最適化プロポーザルを生成するために POST 要求を送信するときに、リバランスエンドポイントを指定します。

/rebalance

/rebalance エンドポイントは、クラスター内のすべてのブローカーにレプリカを移動して完全なリバランスを実行します。

/add_broker

add_broker エンドポイントは、1 つ以上のブローカーを追加することで、Kafka クラスターのスケールアップ後に使用されます。通常、Kafka クラスターをスケールアップした後、新しいブローカーは、新しく作成されたトピックのパーティションのみをホストするために使用されます。新しいトピックが作成されないと、新たに追加されたブローカーは使用されず、既存のブローカーは同じ負荷のままになります。ブローカーをクラスターに追加してすぐに **add_broker** エンドポイントを使用すると、リバランス操作はレプリカを既存のブローカーから新たに追加されたブローカーに移動します。POST 要求で、新しいブローカーを **brokerid** リストとして指定します。

/remove_broker

/remove_broker エンドポイントは、1 つ以上のブローカーを削除して Kafka クラスターをスケールダウンする前に使用されます。Kafka クラスターをスケールダウンすると、レプリカをホストする場合でもブローカーはシャットダウンされます。これにより、レプリケートが不十分なパーティショ

ンとなる可能性があり、一部のパーティションが最小 In-Sync レプリカ (ISR) を下回る可能性があります。この問題を回避するため、**/remove_broker** エンドポイントは、削除予定のブローカーからレプリカを移動します。これらのブローカーがレプリカをホストしなくなった場合は、スケールダウン操作を安全に実行できます。POST 要求で、削除するブローカーを **brokerid** リストとして指定します。

通常、**/rebalance** エンドポイントを使用して、ブローカー間で負荷を分散し、Kafka クラスターをリバランスします。**/add-broker** エンドポイントと **/remove_broker** エンドポイントは、クラスターをスケールアップまたはスケールダウンし、それに応じてレプリカを再調整する場合にのみ使用してください。

結局のところ、リバランスを実行する手順は、3 つの異なるエンドポイント間で同じとなります。唯一の違いは、要求に追加されたブローカーや、要求から削除されるブローカーのリスト表示のみです。

13.6.2. 最適化プロポーザルの承認または拒否

最適化プロポーザルのサマリーは、提案された変更の範囲を示しています。サマリーは、Cruise Control API を介した HTTP リクエストへの応答で返されます。

/rebalance エンドポイントに POST リクエストを行うと、最適化プロポーザルのサマリーがレスポンスで返されます。

最適化プロポーザルの要約を返す方法

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

サマリーを使用して、最適化プロポーザルを承認するか拒否するかを決定します。

最適化プロポーザルの承認

/rebalance エンドポイントに POST リクエストを送信し、**dryrun** パラメーターを **false** (デフォルトは **true**) に設定して、最適化のプロポーザルを承認します。Cruise Control は、プロポーザルを Kafka クラスターに適用し、クラスターのリバランス操作を開始します。

最適化プロポーザルの拒否

最適化プロポーザルを承認しないことを選択した場合は、最適化ゴールの変更 または 任意のリバランスパフォーマンスチューニングオプションの更新 を行い、その後で別のプロポーザルを生成できます。**dryrun** パラメーターなしでリクエストを再送信して、新しい最適化プロポーザルを生成できます。

最適化プロポーザルを使用して、リバランスに必要な動作を評価します。たとえば、要約ではブローカー間およびブローカー内の動きについて記述します。ブローカー間のリバランスは、別々のブローカー間でデータを移動します。JBOD ストレージ設定を使用していると、ブローカー内のリバランスでは同じブローカー上のディスク間でデータが移動します。このような情報は、プロポーザルを承認しない場合でも有用な場合があります。

リバランスの際には Kafka クラスターに追加の負荷がかかるため、最適化プロポーザルを却下したり、承認を遅らせたりする場合があります。

次の例では、プロポーザルは別々のブローカー間のデータのリバランスを提案しています。リバランスには、ブローカー間での 55 個のパーティションレプリカ (合計 12 MB のデータ) の移動が含まれます。パーティションレプリカのブローカー間の移動は、パフォーマンスに大きな影響を与えますが、データ総量はそれほど多くありません。合計データが膨大な場合は、プロポーザルを却下するか、リバランスを承認するタイミングを考慮して Kafka クラスターのパフォーマンスへの影響を制限できます。

リバランスパフォーマンスチューニングオプションは、データ移動の影響を減らすのに有用です。リバランス期間を延長できる場合は、リバランスをより小さなバッチに分割できます。一回のデータ移動が少なくなると、クラスターの負荷も軽減できます。

最適化プロポーザルサマリーの例

```
Optimization has 55 inter-broker replica (12 MB) moves, 0 intra-broker
replica (0 MB) moves and 24 leadership moves with a cluster model of 5
recent windows and 100.000% of the partitions covered.
Excluded Topics: [].
Excluded Brokers For Leadership: [].
Excluded Brokers For Replica Move: [].
Counts: 3 brokers 343 replicas 7 topics.
On-demand Balancedness Score Before (78.012) After (82.912).
Provision Status: RIGHT_SIZED.
```

このプロポーザルでは、24 のパーティションリーダーも別のブローカーに移動します。これによるパフォーマンスへの影響はほとんどありません。

バランススコアは、最適化プロポーザルが承認される前後の Kafka クラスターの全体的なバランスの測定値です。バランススコアは、最適化ゴールに基づいています。すべてのゴールが満たされていると、スコアは 100 になります。達成されないゴールごとにスコアが減少します。バランススコアを比較して、Kafka クラスターのバランスがリバランス後よりも悪いかどうかを確認します。

provision ステータスは、現在のクラスター設定が最適化ゴールをサポートするかどうかを示します。プロビジョニングステータスを確認し、ブローカーを追加または削除する必要があるかどうかを確認します。

表13.2 最適化プロポーザルのプロビジョニングステータス

Status	説明
RIGHT_SIZED	クラスターには、最適化ゴールを満たす適切な数のブローカーがあります。
UNDER_PROVISIONED	クラスターはプロビジョニングされておらず、最適化ゴールに対応するために追加のブローカーが必要になります。
OVER_PROVISIONED	クラスターはオーバープロビジョニングされており、最適化ゴールを満たすためにブローカーの数を減らします。
UNDECIDED	ステータスは関連性がなく、まだ決定されていません。

13.6.3. 最適化プロポーザルサマリーのプロパティー

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13.3 最適化プロポーザルに含まれるプロパティーの概要

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。
n intra-broker replica (y MB) moves	**n**: ディスクとクラスターのブローカーとの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す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は、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** 参照)。
n excluded topics	最適化プロポーザル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たトピックの数。 以下のいずれかの方法で、トピックを除外することができます。 **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。 正規表現と一致するトピックが応答にリスト表示され、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**: 比較的低い。
n recent windows	**n**: 最適化プロポーザルの基になるメトリックウインドウの数。
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、**default.goals** またはユーザー提供ゴールのリストでゴールの位置を示す優先順位が含まれます。**On-demand Balancedness Score** スコアは、違反した各ソフトゴールの **Balancedness Score** の合計を 100 から引いて算出されます。 **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。

13.6.4. 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 デフォルトの最適化ゴール を基にした **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場合に、キャッシュされた最新の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
- 現在キャッシュされているプロポーザルで達成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、Cruise Control デプロイメント設定の **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。

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

13.7. 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行われる方法を制御し、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設定されています。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設定されます。

- **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を行うと、このストラテジーではその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 3 つの代替レプリカ移動ストラテジーを提供します。

- **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の順序です。
- **PostponeUrpReplicaMovementStrategy**: 非同期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、Cruise Control のデフォルト設定および **cruisecontrol.properties** ファイルに設定
- パラメーターとして、**/rebalance** エンドポイントへの POST リクエストに設定

両方の方法に関連する設定を以下の表にまとめています。

表13.4 リバランスパフォーマンスチューニングの設定

Cruise Control プロパティー	KafkaRebalance パラメーター	デフォルト	説明
num.concurrent.partition.movements.per.broker	**concurrent_partition_movements_per_broker**	5	各パーティション再割り当てバッチにおける inter-broker パーティション移動の最大数。
num.concurrent.intra.broker.partition.movements	**concurrent_intra_broker_partition_movements**	2	各パーティション再割り当てバッチにおけるブローカー内パーティション移動の最大数。
num.concurrent.leader.movements	**concurrent_leader_movements**	1000	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。
default.replication.throttle	**replication_throttle**	Null (制限なし)	パーティションの再割り当てに割り当てる帯域幅 (バイト/秒単位)。

Cruise Control プロパティー	KafkaRebalance パラメーター	デフォルト	説明
default.replica.movement.strategies	**replica_movement_strategies**	**BaseReplicaMovementStrategy**	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。3 つのストラテジー **PrioritizeSmallReplicaMovementStrategy**、**PrioritizeLargeReplicaMovementStrategy**、および **PostponeUrpReplicaMovementStrategy** があります。サーバーの設定には、ストラテジークラスの完全修飾名をコンマ区切りの文字列で指定します (各クラス名の先頭に **com.linkedin.kafka.cruisecontrol.executor.strategy.** を追加します)。リバランスパラメーターには、レプリカ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。

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ます。その逆も同様です。

関連情報

- Cruise Control Wiki の Configurations
- Cruise Control Wiki の REST API

13.8. 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。このファイルは、以下のいずれかのタイプのプロパティーで設定されます。

- 文字列
- 数値
- Boolean

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どうかを判断します。1 つ以上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、最適化の試みは失敗します。これにより、最適化を使用して最適化プロポーザルを生成できなくなります。

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に指定します。

- **capacityJBOD.json**: JBOD Kafka デプロイメントでの使用 (デフォルトのファイル)。
- **capacity.json**: 各ブローカーに同じ数の CPU コアがある、JBOD 以外の Kafka デプロイメントでの使用。
- **capacityCores.json**: 各ブローカーによって CPU コアの数が異なる、JBOD 以外の Kafka デプロイメントでの使用。

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) またはコアの数としての CPU 使用率
- **NW_IN**: KB 毎秒単位のインバウンドネットワークスループット
- **NW_OUT**: KB 毎秒単位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各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。個々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を設定することが重要です。設定されていない場合は、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 Metrics Reporter のデプロイ」 で説明されているように、Kafka 設定ファイルに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ます。

- **cruise.control.metrics.topic.auto.create=true**
- **cruise.control.metrics.topic.num.partitions=1**
- **cruise.control.metrics.topic.replication.factor=1**

Cruise Control Metrics Reporter (**cruise.control.metrics.topic.auto.create=false**) でトピックの自動作成が **無効** で、Kafka クラスターで **有効** になっていると、**__CruiseControlMetrics** トピックはブローカーによって自動的に作成されます。この場合は、**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。

```
opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --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。

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --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。

変更を反映するには、Cruise Control サーバーを再起動する必要があります。

13.9. 最適化プロポーザルの生成

/rebalance エンドポイントに POST リクエストを行うと、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。最適化プロポーザルの結果を使用して Kafka クラスターをリバランスできます。

以下のエンドポイントのいずれかを使用して最適化プロポーザルを実行できます。

- **/rebalance**
- **/add_broker**
- **/remove_broker**

使用するエンドポイントは、Kafka クラスターで実行しているすべてのブローカーでリバランスを行うか、Kafka クラスターをスケールダウンした後にリバランスを行うかによって異なります。詳細は、Rebalancing endpoints with broker scaling を参照してください。

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。「ドライランモード」では、Cruise Control は最適化プロポーザルと推定結果を生成しますが、クラスターをリバランスしてプロポーザルを開始することはありません。

最適化プロポーザルで返される情報を分析し、プロポーザルを承認するかどうかを決定できます。

エンドポイントへの要求を行うには、以下のパラメーターを使用します。

dryrun

型: boolean、デフォルト: 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を生成してクラスターのリバランスを実行するか (**false**) を Cruise Control に通知します。

dryrun=true (デフォルト) の場合は、**verbose** パラメーターを渡して Kafka クラスターの状態に関する詳細情報を返すこともできます。これには、最適化プロポーザルの適用前および適用後の各 Kafka ブローカーの負荷のメトリックと、前後の値の違いが含まれます。

excluded_topics

型: 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: 文字列のリスト。デフォルト: 設定済み **default.goals** リスト

最適化プロポーザルを準備するために使用するユーザー提供の最適化ゴールのリスト。goals が指定されていない場合は、**cruisecontrol.properties** ファイルに設定されている **default.goals** リストが使用されます。

skip_hard_goals_check

型: boolean、デフォルト: **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**) 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、リクエストは失敗します。

設定されたすべての **hard.goals** を含まない、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、デフォルト: **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。指定のない場合、または **false** に設定した場合、Cruise Control はコマンドラインでの表示用に書式設定されたテキストを返します。返された情報の要素をプログラムで抽出する場合は、**json=true** を設定します。これにより、**jq** などのツールにパイプ処理できる JSON 形式のテキストを返したり、スクリプトやプログラムで解析することができます。

verbose

型: boolean、デフォルト: **false**

Cruise Control サーバーが返す応答の詳細レベルを制御します。**dryrun=true** と併用できます。

注記

その他のパラメーターも利用可能です。詳細は、Cruise Control Wiki の REST APIs を参照してください。

前提条件

- Kafka が実行中である。
- Cruise Control を設定 した。
- (スケールアップ用のオプション) リバランスに追加するために、ホストに新しいブローカーをインストール した。

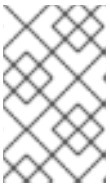
手順

1. **/rebalance**、**/add_broker**、または **/remove_broker** エンドポイントへの POST 要求を使用して最適化プロポーザルを生成します。

   デフォルトのゴールを使用した /rebalance への要求の例

   ```
   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は即座に返されます。



注記

NotEnoughValidWindows が返された場合、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。数分待ってからリクエストを再送信します。

指定されたゴールを使用した /rebalance への要求の例

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
goals=RackAwareGoal,ReplicaCapacityGoal'
```

要求が、指定のゴールを満たしていると、キャッシュされた最適化プロポーザルが即座に返されます。それ以外の場合は、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。計算には時間がかかります。この動作を強制するには、リクエストに **ignore_proposal_cache=true** パラメーターを追加します。

ハードゴールなしで指定されたゴールを使用した /rebalance への要求の例

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?
goals=RackAwareGoal,ReplicaCapacityGoal,ReplicaDistributionGoal&skip_hard_goal_check=tr
ue'
```

指定のブローカーが含まれる /add_broker への要求の例

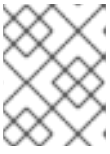
```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?brokerid=3,4'
```

要求には、新規ブローカーの ID のみが含まれます。たとえば、この要求は ID **3** と **4** のブローカーを追加します。レプリカは、リバランス時に既存のブローカーから新しいブローカーに移動します。

指定のブローカーを除外する /remove_broker への要求の例

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?brokerid=3,4'
```

要求には、除外されるブローカーの ID が含まれます。たとえば、この要求は ID が **3** と **4** のブローカーを除外します。レプリカは、リバランス時に他の既存ブローカーから削除されるブローカーから移動されます。



注記

削除されるブローカーがトピックを除外した場合でも、レプリカは移動されます。

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、各デフォルトの最適化ゴールの概要、およびプロポーザルの実行後に予想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。要件を満たす場合には、概要を使用して、提案された変更の影響を評価する必要があります。
- **n inter-broker replica (y MB) moves** はブローカー間でネットワークを介して移動されるデータ量を示します。値が高いほど、リバランス中の Kafka クラスターへの潜在的なパフォーマンスの影響が大きくなります。
- **n intra-broker replica (y MB) moves** は、ブローカー内部で (ディスク間) でどれだけのデータを移動させるかを示します。値が大きいほど、個々のブローカーに対する潜在的なパフォーマンスの影響は大きくなります (ただし **n inter-broker replica (y MB) moves** の影響よりも小さい)。
- リーダーシップ移動の数。これは、リバランス中のクラスターのパフォーマンスにほとんど影響しません。

非同期応答

Cruise Control REST API エンドポイントは、デフォルトでは 10 秒後に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継続されます。最近キャッシュされた最適化プロポーザルが準備状態にない場合や、ユーザー提供の最適化ゴールが **ignore_proposal_cache=true** で指定された場合は、タイムアウトが発生することがあります。

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あるリクエストの一意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 2023 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)
```

タイムアウト時間内に最適化プロポーザルが準備ができなかった場合、POST リクエストを再送信できます。これには、ヘッダーにある元リクエストの **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

「最適化プロポーザルの承認」

13.10. 最適化プロポーザルの承認

最近生成された最適化プロポーザルが適切であれば、Cruise Control にクラスターのリバランスを開始し、パーティションの再割り当てを開始できます。

最適化プロポーザルを生成し、クラスターリバランスを開始するまでの時間をできるだけ短くします。元の最適化プロポーザルの生成後に時間が経過していると、クラスターの状態が変更している場合があります。そのため、開始したクラスターのリバランスが、確認したものとは異なる場合があります。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" の進行中のクラスターリバランスは、一度に 1 つだけになります。

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。

手順

1. **dryrun=false** パラメーターを使用して、POST 要求を **/rebalance**、**/add_broker**、または **/remove_broker** エンドポイントに送信します。
   /add_broker または **/remove_broker** エンドポイントを使用して、ブローカーを含むまたは除外するプロポーザルを生成した場合は、同じエンドポイントを使用して、指定されたブローカーの有無にかかわらずリバランスを実行します。

   /rebalance への要求の例

   ```
   curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?dryrun=false'
   ```

   /add_broker への要求の例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?
   dryrun=false&brokerid=3,4'
   ```

   /remove_broker への要求の例

   ```
   curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?
   dryrun=false&brokerid=3,4'
   ```

   Cruise Control はクラスター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。変更が予想される内容ではない場合は、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。進行中のクラスターリバランスのステータスは "Active" です。
   Cruise Control サーバーで実行される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状態を表示するには、**user-task-ids** パラメーターおよびタスク ID を指定します。

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks?user_task_ids=c459316f-9eb5-482f-9d2d-97b5a4cd294d'
```

(オプション) スケールダウン時のブローカーの削除

リバランスが正常に完了したら、Kafka クラスターをスケールダウンするために除外したブローカーを停止できます。

1. 削除する各ブローカーに、ログ (**log.dirs**) にライブパーティションがないことを確認します。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

ログディレクトリーが正規表現 (**.[a-z0-9]-delete$**) と一致しない場合は、アクティブなパーティションが存在します。アクティブなパーティションが存在する場合は、リバランスが完了していること、または最適化プロポーザルの設定を確認します。プロポーザルを再度実行できます。次のステップに進む前に、アクティブなパーティションが存在しないことを確認します。

2. ブローカーを停止します。

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

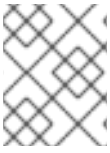
3. ブローカーが停止したことを確認します。

```
jcmd | grep kafka
```

13.11. アクティブなクラスターリバランスの停止

現在進行中の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に、完了したパーティションの再割り当てはすで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、初期状態よりも低下している可能性があります。

前提条件

- クラスターリバランスが進行中である (ステータスは "Active")。

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'
```

関連情報

- 最適化プロポーザルの生成

第14章 CRUISE CONTROL を使用したトピックレプリケーション係数の変更

Cruise Control REST API の **/topic_configuration** エンドポイントにリクエストを送信して、レプリケーション係数を含むトピック設定を変更します。

前提条件

- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。
- Cruise Control を設定 した。
- Cruise Control Metrics Reporter をデプロイ した。

手順

1. Cruise Control サーバーを起動します。デフォルトでは、サーバーはポート 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/
./kafka-cruise-control-start.sh config/cruisecontrol.properties <port_number>
```

2. Cruise Control が実行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 -X GET 'http://<cc_host>:<cc_port>/kafkacruisecontrol/state'
```

3. **--describe** オプションを指定して **bin/kafka-topics.sh** コマンドを実行し、ターゲットトピックの現在のレプリケーション係数を確認します。

```
/opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --topic <topic_name> \
  --describe
```

4. トピックのレプリケーション係数を更新します。

```
curl -X POST 'http://<cc_host>:<cc_port>/kafkacruisecontrol/topic_configuration?topic=
<topic_name>&replication_factor=<new_replication_factor>&dryrun=false'
```

たとえば、**curl -X POST 'localhost:9090/kafkacruisecontrol/topic_configuration?topic=topic1&replication_factor=3&dryrun=false'** とします。

5. 前述の手順と同様に、**--describe** オプションを指定して **bin/kafka-topics.sh** コマンドを実行し、トピックへの変更の結果を確認します。

第15章 パーティション再割り当てツールの使用

Kafka クラスターをスケーリングする場合、ブローカーを追加または削除し、パーティションの分散またはトピックのレプリケーション係数を更新する必要がある場合があります。パーティションとトピックを更新するには、**kafka-reassign-partitions.sh** ツールを使用できます。

kafka-reassign-partitions.sh ツールを使用してトピックのレプリケーション係数を変更できます。このツールを使用して、パーティションを再割り当てし、ブローカー間でパーティションの分散のバランスをとり、パフォーマンスを向上させることもできます。ただし、パーティション再割り当てとクラスターリバランスの自動化 および トピックレプリケーション係数の変更 には、Cruise Control を使用することを推奨します。Cruise Control は、ダウンタイムなしでトピックをあるブローカーから別のブローカーに移動でき、パーティションを再割り当てする最も効率的な方法です。

15.1. パーティション再割り当てツールの概要

パーティション再割り当てツールは、Kafka パーティションとブローカーを管理するための次の機能を提供します。

パーティションレプリカの再配布

ブローカーを追加または削除してクラスターをスケールアップおよびスケールダウンし、負荷の高いブローカーから使用率の低いブローカーに Kafka パーティションを移動します。これを行うには、移動するトピックとパーティションとそれらをどこに移動するかを特定するパーティション再割り当て計画を作成する必要があります。クラスターのリバランスプロセスを自動化 する Cruise Control は、このタイプの操作に推奨されます。

トピックレプリケーション係数の増減のスケーリング

Kafka トピックのレプリケーション係数を増減します。これを行うには、パーティション間の既存のレプリケーション割り当てと、レプリケーション係数の変更を伴う更新された割り当てを識別するパーティション再割り当て計画を作成する必要があります。

優先リーダーの変更

Kafka パーティションの優先リーダーを変更します。これは、現在優先されているリーダーが使用できない場合、またはクラスター内のブローカー間で負荷を再分散したい場合に役立ちます。これを行うには、レプリカの順序を変更して各パーティションの新しい優先リーダーを指定するパーティション再割り当て計画を作成する必要があります。

特定の JBOD ボリュームを使用するようにログディレクトリーを変更する

特定の JBOD ボリュームを使用するように Kafka ブローカーのログディレクトリーを変更します。これは、Kafka データを別のディスクまたはストレージデバイスに移動する場合に便利です。これを行うには、トピックごとに新しいログディレクトリーを指定するパーティション再割り当て計画を作成する必要があります。

15.1.1. パーティション再割り当て計画の生成

パーティション再割り当てツール (**kafka-reassign-partitions.sh**) は、どのパーティションを現在のブローカーから新しいブローカーに移動する必要があるかを指定するパーティション割り当てプランを生成することによって機能します。

計画に満足したら、実行できます。その後、ツールは次の処理を実行します。

- パーティションデータを新しいブローカーに移行する
- Kafka ブローカー上のメタデータを更新して、新しいパーティションの割り当てを反映する
- 新しい割り当てが確実に有効になるように、Kafka ブローカーのローリング再起動をトリガーする

パーティション再割り当てツールには 3 つの異なるモードがあります。

--generate

トピックとブローカーのセットを取得し、**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。これはトピック全体で動作するため、一部のトピックのパーティションを再度割り当てる場合は使用できません。

--execute

再割り当て JSON ファイル を取得し、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。その結果、パーティションを取得したブローカーは、パーティションリーダーのフォロワーになります。新規ブローカーが ISR (同期レプリカ) に参加できたら、古いブローカーは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チェックします。再割り当てが完了すると、**--verify** は有効なトラフィックスロットル (**--throttle**) も削除します。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、完了するまで待ってから別の再割り当てを実行して、最初の再割り当ての効果を元に戻します。**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。

15.1.2. パーティション再割り当て JSON ファイルでのトピックの指定

kafka-reassign-partitions.sh ツールは、再割り当てを行うトピックを指定する再割り当て JSON ファイルを使用します。特定のパーティションを移動させたい場合は、再割り当て JSON ファイルを生成するか、手動でファイルを作成します。

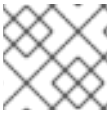
基本的な再割り当て JSON ファイルの構造は次の例に示されており、2 つの Kafka トピックに属する 3 つのパーティションが記述されています。各パーティションは、ブローカー ID によって識別される新しいレプリカのセットに再割り当てされます。プロパティー **version**、**topic**、**partition**、**replicas** はすべて必須です。

パーティションの再割り当ての JSON ファイル構造の例

```
{
  "version": 1, (1)
  "partitions": [ (2)
    {
      "topic": "example-topic-1", (3)
      "partition": 0, (4)
      "replicas": [1, 2, 3] (5)
    },
    {
      "topic": "example-topic-1",
      "partition": 1,
      "replicas": [2, 3, 4]
    },
    {
      "topic": "example-topic-2",
      "partition": 0,
```

```
      "replicas": [3, 4, 5]
    }
  ]
}
```

1. 再割り当て JSON ファイル形式のバージョン。現在、バージョン 1 のみがサポートされているため、これは常に 1 である必要があります。
2. 再割り当てするパーティションを指定する配列。
3. パーティションが属する Kafka トピックの名前。
4. 再割り当てされるパーティションの ID。
5. このパーティションのレプリカとして割り当てる必要があるブローカーの ID の順序付けされた配列。リストの最初のブローカーがリーダーレプリカです。



注記

JSON に含まれていないパーティションは変更されません。

topics 配列を使用してトピックのみを指定すると、パーティション再割り当てツールは、指定されたトピックに属するすべてのパーティションを再割り当てします。

トピックのすべてのパーティションを再割り当てするための再割り当て JSON ファイル構造の例

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

15.1.3. JBOD ボリューム間のパーティションの再割り当て

Kafka クラスターで JBOD ストレージを使用する場合は、特定のボリュームとログディレクトリー (各ボリュームに単一のログディレクトリーがある) との間でパーティションの再割り当てできます。

パーティションを特定のボリュームに再割り当てするには、再割り当て JSON ファイル内の各パーティションの **log_dirs** 値を追加します。各レプリカは特定のログディレクトリーに割り当てる必要があるため、各 **log_dirs** 配列には、**replicas** 配列と同じ数のエントリーが含まれます。**log_dirs** 配列には、ログディレクトリーへの絶対パスまたは特別な値 **any** が含まれます。**any** 値は、Kafka がそのレプリカに対して使用可能な任意のログディレクトリーを選択できることを示します。これは、JBOD ボリューム間でパーティションを再割り当てするときに役立ちます。

ログディレクトリーを含む再割り当て JSON ファイル構造の例

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "example-topic-1",
```

```
      "partition": 0,
      "replicas": [1, 2, 3]
      "log_dirs": ["/var/lib/kafka/data-0/kafka-log1", "any", "/var/lib/kafka/data-1/kafka-log2"]
    },
    {
      "topic": "example-topic-1",
      "partition": 1,
      "replicas": [2, 3, 4]
      "log_dirs": ["any",  "/var/lib/kafka/data-2/kafka-log3", "/var/lib/kafka/data-3/kafka-log4"]
    },
    {
      "topic": "example-topic-2",
      "partition": 0,
      "replicas": [3, 4, 5]
      "log_dirs": ["/var/lib/kafka/data-4/kafka-log5", "any",  "/var/lib/kafka/data-5/kafka-log6"]
    }
  ]
}
```

15.1.4. パーティション再割り当てのスロットル

パーティション再割り当てには、ブローカーの間で大量のデータを転送する必要があるため、処理が遅くなる可能性があります。クライアントへの悪影響を防ぐため、再割り当て処理をススロットルできます。**--throttle** パラメーターを **kafka-reassign-partitions.sh** ツールと共に使用して、再割り当てをスロットルします。ブローカー間のパーティションの移動の最大しきい値をバイト単位で指定します。たとえば **--throttle 5000000** は、パーティションを移動する最大しきい値を 50 MBps に設定します。

スロットリングにより、再割り当ての完了に時間がかかる場合があります。

- 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対応できず、再割り当ては完了しません。
- 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、確認応答を待つ通常のレイテンシーよりも高い可能性があります。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。

15.2. ブローカーの追加後のパーティションの再割り当て

Kafka クラスター内のブローカーの数を増やした後、**kafka-reassign-partitions.sh** ツールによって生成された再割り当てファイルを使用してパーティションを再割り当てします。再割り当てファイルには、拡大された Kafka クラスター内のブローカーにパーティションを再度割り当てる方法を記述する必要があります。ファイルで指定された再割り当てをブローカーに適用し、新しいパーティションの割り当てを確認します。

この手順では、TLS を使用するセキュアなスケーリングプロセスについて説明します。TLS 暗号化と mTLS 認証を使用する Kafka クラスターが必要です。

注記

kafka-reassign-partitions.sh ツールを使用することもできますが、パーティション再割り当てとクラスターの再バランシングを自動化 するには、Cruise Control の使用を推奨します。Cruise Control は、ダウンタイムなしでトピックをあるブローカーから別のブローカーに移動でき、パーティションを再割り当てする最も効率的な方法です。

前提条件

- 既存の Kafka クラスター。
- 追加の 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を指定する JSON ファイルを作成している。
  この手順では、**my-topic** というトピックのすべてのパーティションを再割り当てします。**topic.json** という名前の JSON ファイルはトピックを指定し、**reassignment.json** ファイルの生成に使用されます。

my-topic を指定する JSON ファイルの例

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

手順

1. クラスター内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** には他のブローカで使用されていない番号を指定してください。
2. 前のステップで作成した設定ファイルを**kafka-server-start.sh** スクリプトの引数に渡して、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/kraft/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. まだ行っていない場合は、**kafka-reassign-partitions.sh** ツールを使用して **reassignment.json** という名前の再割り当て JSON ファイルを生成します。

   再割り当て JSON ファイルを生成するコマンドの例

   ```
   /opt/kafka/bin/kafka-reassign-partitions.sh \
     --bootstrap-server localhost:9092 \
     --topics-to-move-json-file topics.json \ 1
     --broker-list 0,1,2,3,4 \ 2
     --generate
   ```

   1 トピックを指定する JSON ファイル。

   2 操作に含める Kafka クラスター内のブローカー ID。これは、ブローカー **4** が追加されたことを前提としています。

現在のレプリカ割り当てと提案されたレプリカ割り当てを示す再割り当て JSON ファイルの例

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,0],"log_dirs":
["any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2,3],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3,4],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":
["any","any","any","any"]}]}
```

後で変更を元に戻す必要がある場合に備えて、このファイルのコピーをローカルに保存します。

6. **--execute** オプションを使用してパーティションの再割り当てを実行します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** と inter-broker 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。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

7. **--verify** オプションを使用して、再割り当てが完了したことを確認します。

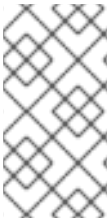
```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --verify
```

--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。

15.3. ブローカーの削除前のパーティションの再割り当て

Kafka クラスター内のブローカーの数を減らす前に、**kafka-reassign-partitions.sh** ツールによって生成された再割り当てファイルを使用してパーティションを再割り当てします。再割り当てファイルでは、Kafka クラスターの残りのブローカーにパーティションを再割り当てする方法を記述する必要があります。ファイルで指定された再割り当てをブローカーに適用し、新しいパーティションの割り当てを確認します。最も番号の大きい Pod のブローカーが最初に削除されます。

この手順では、TLS を使用するセキュアなスケーリングプロセスについて説明します。TLS 暗号化と mTLS 認証を使用する Kafka クラスターが必要です。



注記

kafka-reassign-partitions.sh ツールを使用することもできますが、パーティション再割り当てとクラスターの再バランシングを自動化 するには、Cruise Control の使用を推奨します。Cruise Control は、ダウンタイムなしでトピックをあるブローカーから別のブローカーに移動でき、パーティションを再割り当てする最も効率的な方法です。

前提条件

- 既存の Kafka クラスター。
- 縮小された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を指定する JSON ファイルを作成している。
  この手順では、**my-topic** というトピックのすべてのパーティションを再割り当てします。**topic.json** という名前の JSON ファイルはトピックを指定し、**reassignment.json** ファイルの生成に使用されます。

my-topic を指定する JSON ファイルの例

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

手順

1. まだ行っていない場合は、**kafka-reassign-partitions.sh** ツールを使用して **reassignment.json** という名前の再割り当て JSON ファイルを生成します。

   再割り当て JSON ファイルを生成するコマンドの例

   ```
   /opt/kafka/bin/kafka-reassign-partitions.sh \
     --bootstrap-server localhost:9092 \
     --topics-to-move-json-file topics.json \ 1
     --broker-list 0,1,2,3 \ 2
     --generate
   ```

   1 トピックを指定する JSON ファイル。

   2 操作に含める Kafka クラスター内のブローカー ID。これは、ブローカー **4** が削除されたことを前提としています。

   現在のレプリカ割り当てと提案されたレプリカ割り当てを示す再割り当て JSON ファイルの例

   ```
   Current partition replica assignment
   {"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[3,4,2,0],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":
   ```

```
["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":
["any","any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,0],"log_dirs":
["any","any","any"]}]}
```

後で変更を元に戻す必要がある場合に備えて、このファイルのコピーをローカルに保存します。

2. **--execute** オプションを使用してパーティションの再割り当てを実行します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** と inter-broker 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。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

3. **--verify** オプションを使用して、再割り当てが完了したことを確認します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --verify
```

--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。

4. 削除する各ブローカーに、ログ (**log.dirs**) にライブパーティションがないことを確認します。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

ログディレクトリーが正規表現 (**.[a-z0-9]-delete$**) と一致しない場合は、アクティブなパーティションが存在します。アクティブなパーティションがある場合は、再割り当てが完了したどうか、あるいは、再割り当て JSON ファイルの設定を確認します。再割り当てはもう一度実行できます。次のステップに進む前に、アクティブなパーティションが存在しないことを確認します。

5. ブローカーを停止します。

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

6. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

15.4. トピックのレプリケーション係数の変更

kafka-reassign-partitions.sh ツールを使用して、Kafka クラスター内のトピックのレプリケーション係数を変更します。これは、トピックのレプリカを変更する方法を記述する再割り当てファイルを使用して実行できます。

前提条件

- 既存の Kafka クラスター。
- 操作に含めるトピックを指定するための JSON ファイルを作成している。
  この手順では、**my-topic** というトピックに 4 つのレプリカがあり、それを 3 つに減らしたいと考えています。**topic.json** という名前の JSON ファイルはトピックを指定し、**reassignment.json** ファイルの生成に使用されます。

my-topic を指定する JSON ファイルの例

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

手順

1. まだ行っていない場合は、**kafka-reassign-partitions.sh** ツールを使用して **reassignment.json** という名前の再割り当て JSON ファイルを生成します。

   #### 再割り当て JSON ファイルを生成するコマンドの例

   ```
   /opt/kafka/bin/kafka-reassign-partitions.sh \
     --bootstrap-server localhost:9092 \
     --topics-to-move-json-file topics.json \ 1
     --broker-list 0,1,2,3,4 \ 2
     --generate
   ```

   1 トピックを指定する JSON ファイル。

   2 操作に含める Kafka クラスター内のブローカー ID。

   #### 現在のレプリカ割り当てと提案されたレプリカ割り当てを示す再割り当て JSON ファイルの例

   ```
   Current partition replica assignment
   {"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[3,4,2,0],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":
   ```

```
["any","any","any","any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2,3],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3,4],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":
["any","any","any","any"]}]}
```

後で変更を元に戻す必要がある場合に備えて、このファイルのコピーをローカルに保存します。

2. **reassignment.json** を編集して、各パーティションからレプリカを削除します。
たとえば、**jq** を使用して、トピックの各パーティションのリスト内の最後のレプリカを削除します。

各パーティションの最後のトピックレプリカの削除

```
jq '.partitions[].replicas |= del(.[-1])' reassignment.json > reassignment.json
```

更新されたレプリカを示す再割り当てファイルの例

```
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,4],"log_dirs":
["any","any","any","any"]}]}
```

3. **--execute** オプションを使用してトピックレプリカを変更します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

注記

ブローカーからレプリカを削除する場合、ブローカー間のデータ移動は必要ないため、レプリケーションを調整する必要はありません。レプリカを追加している場合は、スロットルレートを変更することができます。

4. **--verify** オプションを使用して、トピックレプリカへの変更が完了したことを確認します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --verify
```

--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。

5. **--describe** オプションを指定して **bin/kafka-topics.sh** コマンドを実行して、トピックへの変更の結果を確認します。

```
/opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --describe
```

トピックのレプリカ数を削減した結果

```
my-topic  Partition: 0  Leader: 0  Replicas: 0,1,2 Isr: 0,1,2
my-topic  Partition: 1  Leader: 2  Replicas: 1,2,3 Isr: 1,2,3
my-topic  Partition: 2  Leader: 3  Replicas: 2,3,4 Isr: 2,3,4
```

第16章 分散トレースの設定

分散トレースを使用すると、分散システムのアプリケーション間におけ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、トレースが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有用です。

Streams for Apache Kafka では、トレースにより、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、JMX メトリクス で表示できるメトリクスやコンポーネントロガーを補います。

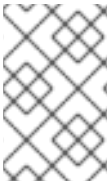
トレースのサポートは、以下の Kafka コンポーネントに組み込まれています。

- Kafka Connect
- MirrorMaker
- MirrorMaker 2
- Streams for Apache Kafka Kafka Bridge

トレースは Kafka ブローカーではサポートされません。

コンポーネントのプロパティーファイルにトレース設定を追加します。

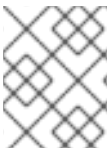
トレースを有効にするには、環境変数を設定し、トレースシステムのライブラリーを Kafka クラスパスに追加します。Jaeger トレースの場合、Jaeger Exporter を使用して OpenTelemetry のトレースアーティファクトを追加できます。



注記

Streams for Apache Kafka は OpenTracing をサポートしなくなりました。これまでに、Jaeger で OpenTracing を使用していた場合は、代わりに OpenTelemetry の使用に移行することを推奨します。

Kafka プロデューサー、コンシューマー、および Kafka Streams API アプリケーションでトレースを有効にするには、アプリケーションコードを **インストルメント化** します。インストルメント化されると、クライアントはメッセージのトレースデータを生成します (メッセージの作成時やログへのオフセットの書き込み時など)。



注記

Streams for Apache Kafka 以外のアプリケーションおよびシステムにトレースを設定する設定は、このコンテンツの範囲外です。

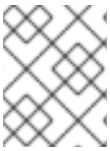
16.1. 手順の概要

Streams for Apache Kafka のトレースを設定するには、以下の手順を順番に行います。

- Kafka Connect、MirrorMaker 2、および MirrorMaker のトレースを設定します。
  - Kafka Connect のトレースの有効化
  - MirrorMaker 2 のトレースを有効にする

- MirrorMaker のトレースの有効化

- クライアントのトレースを設定します。
  - Kafka クライアントの Jaeger トレーサーの初期化
- トレーサーでクライアントをインストルメント化します。
  - プロデューサーおよびコンシューマーをトレース用にインストルメント化
  - Kafka Streams アプリケーションをトレース用にインストルメント化します。



注記

Kafka Bridge のトレースを有効にする方法は、Streams for Apache Kafka Kafka Bridge の使用 を 参照してください。

16.2. トレースオプション

OpenTelemetry を Jaeger トレースシステムとともに使用します。

OpenTelemetry は、トレースまたは監視システムから独立した API 仕様を提供します。

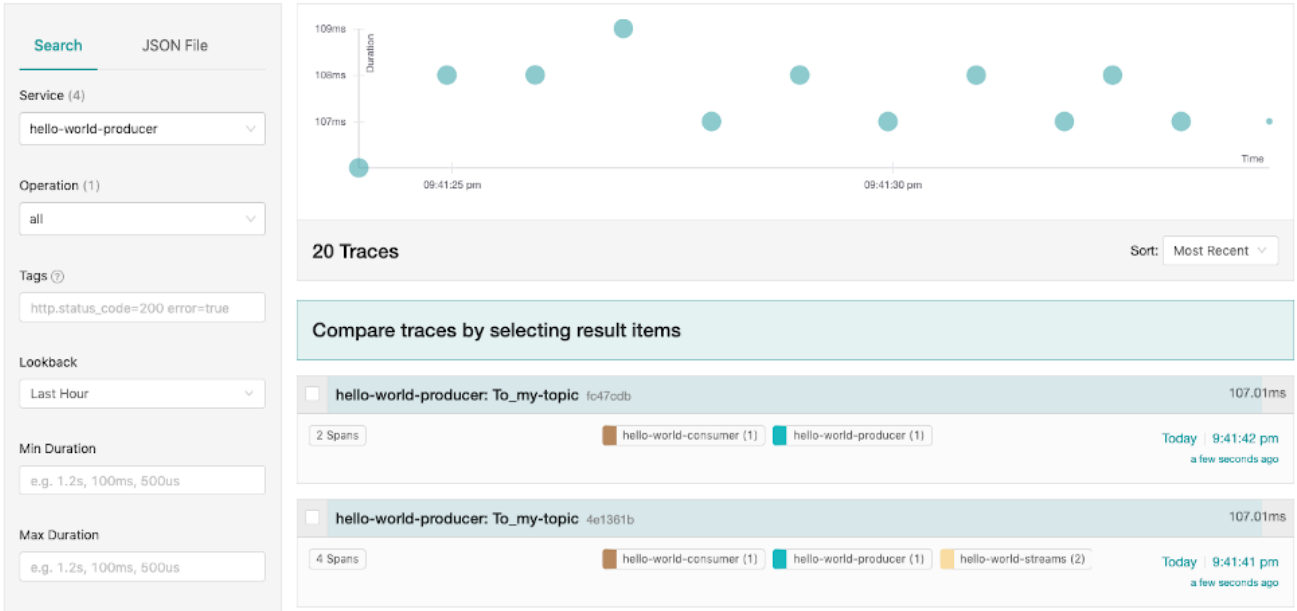
API を使用して、トレース用にアプリケーションコードをインストルメント化します。

- 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リクエストの **トレース** を生成します。
- トレースは、時間軸の中で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スシステムです。

- Jaeger ユーザーインターフェイスを使用すると、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。

簡単なクエリーを表示する Jaeger ユーザーインターフェイス



関連情報

- Jaeger ドキュメント
- OpenTelemetry ドキュメント

16.3. トレースの環境変数

Kafka コンポーネントのトレースを有効にするとき、または Kafka クライアントのトレーサーを初期化するときに、環境変数を使用します。

トレース環境変数は変更する可能性があります。最新情報は、OpenTelemetry のドキュメント を参照してください。

次の表は、トレーサーをセットアップするための主要な環境変数を説明します。

表16.1 OpenTelemetry 環境変数

プロパティー	必要性	説明
OTEL_SERVICE_NAME	必要	OpenTelemetry 向け Jaeger トレースサービスの名前。
OTEL_EXPORTER_JAEGER_ENDPOINT	必要	トレースに使用されるエクスポーター。
OTEL_TRACES_EXPORTER	必要	トレースに使用されるエクスポーター。デフォルトでは **otlp** に設定されています。Jaeger トレースを使用する場合は、この環境変数を **jaeger** として設定する必要があります。別のトレース実装を使用している場合は、使用するエクスポーターを指定します。

16.4. 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 の分散トレースを有効にします。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スされます。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スするには、これらのシステムのコネクターでトレースを設定する必要があります。

OpenTelemetry を使用したトレースを有効にできます。

手順

1. トレースアーティファクトを **opt/kafka/libs** ディレクトリーに追加します。
2. 関連する Kafka Connect 設定ファイルでプロデューサーとコンシューマーのトレースを設定します。
   - スタンドアロンモードで Kafka Connect を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。

次のトレースインターセプタープロパティーを設定ファイルに追加します。

OpenTelemetry のプロパティー

```
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInte
rceptor
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerI
nterceptor
```

トレースを有効にすると、Kafka Connect スクリプトの実行時にトレースを初期化します。

3. 設定ファイルを作成します。
4. トレースの 環境変数 を設定します。
5. 設定ファイルをパラメーター (およびコネクターのプロパティー) として使用して、スタンドアロンまたは分散モードで Kafka Connect を開始します。

スタンドアロンモードでの Kafka Connect の実行

```
su - kafka
/opt/kafka/bin/connect-standalone.sh \
/opt/kafka/config/connect-standalone.properties \
connector1.properties \
[connector2.properties ...]
```

分散モードでの Kafka Connect の起動

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

Kafka Connect の内部コンシューマーとプロデューサーがトレースできるようになりました。

16.5. MIRRORMAKER 2 のトレースを有効にする

MirrorMaker 2 プロパティーファイルで Interceptor プロパティーを定義することにより、MirrorMaker 2 の分散トレースを有効にします。メッセージは Kafka クラスター間でトレースされます。トレースデータは、MirrorMaker 2 コンポーネントに出入りするメッセージを記録します。

OpenTelemetry を使用したトレースを有効にできます。

手順

1. トレースアーティファクトを **opt/kafka/libs** ディレクトリーに追加します。
2. **opt/kafka/config/connect-mirror-maker.properties** ファイルでプロデューサーとコンシューマーのトレースを設定します。
   次のトレースインターセプタープロパティーを設定ファイルに追加します。

OpenTelemetry のプロパティー

```
header.converter=org.apache.kafka.connect.converters.ByteArrayConverter
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInte
rceptor
```

```
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerInterceptor
```

ByteArrayConverter は、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になります。

トレースを有効にすると、Kafka MirrorMaker 2 スクリプトを実行するときにトレースを初期化します。

3. 設定ファイルを作成します。
4. トレースの 環境変数 を設定します。
5. プロデューサーおよびコンシューマー設定ファイルをパラメーターとして使用して MirrorMaker 2 を起動します。

```
su - kafka
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

MirrorMaker 2 の内部コンシューマーとプロデューサのトレースが有効になりました。

16.6. 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設定パラメーターとして渡すことで、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 コンポーネントに出入りするメッセージを記録します。

OpenTelemetry を使用したトレースを有効にできます。

手順

1. トレースアーティファクトを **opt/kafka/libs** ディレクトリーに追加します。
2. **/opt/kafka/config/producer.properties** ファイルでプロデューサートレースを設定します。次のトレースインターセプタープロパティーを追加します。

OpenTelemetry のプロデューサープロパティー

```
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInterceptor
```

3. 設定ファイルを作成します。
4. **/opt/kafka/config/consumer.properties** ファイルでコンシューマートレースを設定します。次のトレースインターセプタープロパティーを追加します。

OpenTelemetry のコンシューマープロパティー

```
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerInterceptor
```

トレースを有効にすると、Kafka MirrorMaker スクリプトの実行時にトレースを初期化します。

5. 設定ファイルを作成します。
6. トレースの 環境変数 を設定します。
7. プロデューサーおよびコンシューマー設定ファイルをパラメーターとして MirrorMaker を起動します。

```
su - kafka
/opt/kafka/bin/kafka-mirror-maker.sh \
--producer.config /opt/kafka/config/producer.properties \
--consumer.config /opt/kafka/config/consumer.properties \
--num.streams=2
```

MirrorMaker の内部コンシューマーとプロデューサーをトレースできるようになりました。

16.7. KAFKA クライアントのトレースの初期化

OpenTelemetry 用のトレーサーを初期化し、分散トレース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。Kafka プロデューサークライアントとコンシューマークライアント、および Kafka Streams API アプリケーションをインストルメント化できます。

一連の トレース環境変数 を使用して、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、トレーサーの依存関係を追加します。

1. クライアントアプリケーションの **pom.xml** ファイルに Maven 依存関係を追加します。

OpenTelemetry の依存関係

```
<dependency>
  <groupId>io.opentelemetry.semconv</groupId>
  <artifactId>opentelemetry-semconv</artifactId>
  <version>1.21.0-alpha</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
  <version>1.34.1</version>
  <exclusions>
    <exclusion>
      <groupId>io.opentelemetry</groupId>
      <artifactId>opentelemetry-exporter-sender-okhttp</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-sender-grpc-managed-channel</artifactId>
  <version>1.34.1</version>
  <scope>runtime</scope>
</dependency>
<dependency>
```

```
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
    <version>1.34.1</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-kafka-clients-2.6</artifactId>
    <version>1.32.0-alpha</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk</artifactId>
    <version>1.34.1</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-sender-jdk</artifactId>
    <version>1.34.1-alpha</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.61.0</version>
</dependency>
```

2. トレース環境変数 を使用して、トレーサーの設定を定義します。
3. 環境変数で初期化されるトレーサーを作成します。

   OpenTelemetry のトレーサーの作成

   ```
   OpenTelemetry ot = GlobalOpenTelemetry.get();
   ```

4. 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);
   ```

5. クライアントをインストルメント化します。
   - 「Kafka プロデューサーおよびコンシューマーをトレース用にインストルメント化」
   - 「Kafka Streams アプリケーションのトレース用のインストルメント化」

16.8. KAFKA プロデューサーおよびコンシューマーをトレース用にインストルメント化

アプリケーションコードを計測して、Kafka プロデューサーとコンシューマーでのトレースを有効にします。デコレーターパターンまたはインターセプターを使用して、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ス用にインストルメント化します。続いて、メッセージが生成されたとき、またはトピックから取得されたときにトレースを記録できます。

OpenTelemetry インストルメント化プロジェクトは、プロデューサーとコンシューマーのインストルメント化をサポートするクラスを提供します。

デコレーターのインストルメント化

デコレーターのインストルメント化では、トレース用に変更したプロデューサーまたはコンシューマーインスタンスを作成します。

インターセプターのインストルメント化

インターセプターのインストルメント化の場合、トレース機能をコンシューマーまたはプロデューサーの設定に追加します。

前提条件

- クライアントのトレースを初期化 している。
  トレース JAR を依存関係としてプロジェクトに追加して、プロデューサーアプリケーションとコンシューマーアプリケーションでインストルメント化を有効にしている。

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、これらの手順を実行します。デコレーターパターンまたはインターセプター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。

- デコレーターパターンを使用するには、変更したプロデューサーまたはコンシューマーインスタンスを作成して、メッセージを送受信します。
  元の **KafkaProducer** または **KafkaConsumer** クラスを渡します。

  OpenTelemetry のデコレーターインストルメント化の例

```
// Producer instance
Producer < String, String > op = new KafkaProducer < > (
    configs,
    new StringSerializer(),
    new StringSerializer()
    );
    Producer < String, String > producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
    configs,
    new StringDeserializer(),
    new StringDeserializer()
    );
    Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

- インターセプターを使用するには、プロデューサーまたはコンシューマーの設定でインターセプタークラスを設定します。
  通常の方法で **KafkaProducer** クラスと **KafkaConsumer** クラスを使用します。**TracingProducerInterceptor** および **TracingConsumerInterceptor** インターセプタークラスは、トレース機能を処理します。

  インターセプターを使用したプロデューサー設定の例

```
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);
```

インターセプターを使用したコンシューマー設定の例

```
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

16.9. KAFKA STREAMS アプリケーションのトレース用のインストルメント化

アプリケーションコードを計測して、Kafka Streams API アプリケーションでのトレースを有効にします。デコレーターパターンまたはインターセプターを使用して、トレース用に Kafka Streams API アプリケーションをインストルメント化します。続いて、メッセージが生成されたとき、またはトピックから取得されたときにトレースを記録できます。

デコレーターのインストルメント化

デコレーターのインストルメント化は、トレース用に変更した Kafka Streams インスタンスを作成します。OpenTelemetry の場合、Kafka Streams のトレースのインストルメント化を提供する **TracingKafkaClientSupplier** カスタムクラスを作成する必要があります。

インターセプターのインストルメント化

インターセプターインストルメント化の場合は、トレース機能を Kafka Streams プロデューサーおよびコンシューマー設定に追加します。

前提条件

- クライアントのトレースを初期化 している。
  トレース JAR を依存関係としてプロジェクトに追加して、Kafka Streams アプリケーションでインストルメント化を有効にしている。
- OpenTelemetry で Kafka Streams をインストルメント化するために、カスタムの **TracingKafkaClientSupplier** を記述している。
- カスタム **TracingKafkaClientSupplier** が Kafka の **DefaultKafkaClientSupplier** を拡張し、プロデューサーとコンシューマーの作成メソッドを上書きして、インスタンスを Telemetry 関連のコードでラップできるようにしている。

カスタム TracingKafkaClientSupplier の例

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {
    @Override
    public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {
        KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getProducer(config));
```

```
    }

    @Override
    public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {
        KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getConsumer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
    }

    @Override
    public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
    }
}
```

手順

Kafka Streams API アプリケーションごとにこの手順を実行します。

- デコレーターパターンを使用するには、**TracingKafkaClientSupplier** サプライヤーインターフェイスのインスタンスを作成し、そのサプライヤーインターフェイスを **KafkaStreams** に提供します。

  デコレーターのインストルメント化の例

  ```
  KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
  KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
  supplier);
  streams.start();
  ```

- インターセプターを使用するには、Kafka Streams プロデューサーおよびコンシューマー設定でインターセプタークラスを設定します。
  TracingProducerInterceptor および **TracingConsumerInterceptor** インターセプタークラスは、トレース機能を処理します。

  インターセプターを使用したプロデューサーとコンシューマーの設定例

  ```
  props.put(StreamsConfig.PRODUCER_PREFIX +
  ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
  TracingProducerInterceptor.class.getName());
  props.put(StreamsConfig.CONSUMER_PREFIX +
  ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
  TracingConsumerInterceptor.class.getName());
  ```

16.10. OPENTELEMETRY でのトレースシステムの指定

デフォルトの Jaeger システムの代わりに、OpenTelemetry でサポートされる他のトレースシステムを指定できます。

OpenTelemetry で別のトレースシステムを使用する場合は、以下の手順を実施します。

1. トレースシステムのライブラリーを Kafka クラスパスに追加します。
2. トレースシステムの名前を追加のエクスポーター環境変数として追加します。

Jaeger を使用しない場合の追加の環境変数

```
OTEL_SERVICE_NAME=my-tracing-service
OTEL_TRACES_EXPORTER=zipkin (1)
OTEL_EXPORTER_ZIPKIN_ENDPOINT=http://localhost:9411/api/v2/spans (2)
```

(1) トレースシステムの名前。この例では、Zipkin を指定します。

(2) スパンをリッスンする特定の選択されたエクスポーターのエンドポイント。この例では、Zipkin エンドポイントが指定されています。

関連情報

- OpenTelemetry エクスポーターの値

16.11. OPENTELEMETRY のカスタムスパン名の指定

トレース **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。スパンには組み込みの名前がありますが、使用する Kafka クライアントインストルメント化で、カスタムスパン名を指定できます。

カスタムスパン名の指定はオプションであり、プロデューサーおよびコンシューマークライアントインストルメント化 または Kafka Streams インストルメント化 でデコレーターパターンを使用する場合にのみ適用されます。

OpenTelemetry でカスタムスパン名を直接指定できません。代わりに、コードをクライアントアプリケーションに追加してスパン名を取得し、追加のタグと属性を抽出します。

属性を抽出するコード例

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor < ProducerRecord < ? , ? >
, Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord) {
        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor < ConsumerRecord < ? ,
? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord) {
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
```

```
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
        Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
        System.setProperty("otel.traces.exporter", "jaeger");
        System.setProperty("otel.service.name", "myapp1");
        KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
            .addProducerAttributesExtractors(new ProducerAttribExtractor())
            .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
            .build();
```

第17章 KAFKA EXPORTER の使用

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のために Streams for Apache Kafka 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 メトリックとして公開され、解析のために Grafana で使用できます。

ビルトイン Kafka メトリクスの監視のために Prometheus および Grafana をすでに使用している場合は、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。

Kafka は JMX 経由でメトリクスを公開し、続いて Prometheus メトリクスとしてエクスポートできます。詳細は、JMX を使用したクラスターの監視 を参照してください。

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、プロセスはリアルタイム処理の目的から遠ざかります。

たとえば、コンシューマーラグは、パージされていない古いデータを大量に消費したり、計画外のシャットダウンが原因である可能性があります。

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やします。

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、Streams for Apache Kafka 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では、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

トピックのレプリケーションが不十分であり、ブローカーが十分なパーティションをレプリケーションしてい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、トピックにレプリケーションが不十分なパーティションが 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。ラグが大きい場合は、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ことを示している可能性があります。

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。

アラートルールは、特定のニーズに合わせて調整できます。

関連情報

アラートルールの設定についての詳細は、Prometheus のドキュメントの 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ック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 トピックメトリックの出力

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レプリケーションが不十分な場合に **1** が示されます。

表17.3 コンシューマーグループメトリックの出力

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 を実行して、Grafana ダッシュボードでのプレゼンテーション用に Prometheus メトリクスを公開します。

Kafka Exporter パッケージをダウンロードしてインストールし、Streams for Apache Kafka で Kafka Exporter を使用します。パッケージをダウンロードしてインストールするには、Streams for Apache Kafka サブスクリプションが必要です。

前提条件

- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- Streams for Apache Kafka のサブスクリプションが ある。

この手順は、Grafana ユーザーインターフェイスへのアクセス権がすでにあり、Prometheus がデプロイされてデータソースとして追加されていることを前提としています。

手順

1. Kafka Exporter パッケージをインストールします。

```
dnf install kafka_exporter
```

2. パッケージがインストールされたことを確認します。

```
dnf info kafka_exporter
```

3. 適切な設定パラメーター値を使用して Kafka Exporter を実行します。

```
kafka_exporter --kafka.server=<kafka_bootstrap_address>:9092 --kafka.version=3.7.0  --<my_other_parameters>
```

パラメーターには、**--kafka.server** など、二重ハイフンの標記が必要です。

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。	**.*** (すべて)
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。	**.*** (すべて)
sasl.<parameter>	ユーザー名とパスワードで SASL/PLAIN 認証を使用して Kafka クラスターを有効にし、接続するパラメーター。	**false**
tls.<parameter>	任意の証明書およびキーで 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	メトリックを公開するポートアドレス。	**:9308**
web.telemetry-path	公開されるメトリックのパス。	**/metrics**

オプション	説明	デフォルト
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**
legacy.partitions	非アクティブなトピックパーティションおよびアクティブなパーティションからメトリックを取得できるようにするブール値。Kafka Exporter が非アクティブなパーティションのメトリックを返すようにするには、**true** に設定します。	**false**

プロパティーの詳細は、**kafka_exporter --help** を使用できます。

4. Kafka Exporter メトリックを監視するように 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。

5. Grafana を有効にして、Prometheus によって公開される Kafka Exporter メトリック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。

Kafka Exporter の更新

Streams for Apache Kafka インストールで最新バージョンの Kafka Exporter を使用します。

更新を確認するには、次を使用します。

```
dnf check-update
```

Kafka Exporter を更新するには、以下を使用します。

```
dnf update kafka_exporter
```

17.5. GRAFANA での KAFKA EXPORTER メトリクスの表示

Kafka Exporter Prometheus メトリック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、メトリックから、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別のラグ
- 毎分のメッセージ消費 (コンシューマーグループ別)

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別のラグ** 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

- Example dashboard for Kafka Exporter
- Grafana のドキュメント

第18章 STREAMS FOR APACHE KAFKA および KAFKA のアップグレード

ダウンタイムなしで Kafka クラスターをアップグレードします。Streams for Apache Kafka 2.7 は、Apache Kafka バージョン 3.7.0 をサポートし、使用します。Kafka 3.6.0 は、Streams for Apache Kafka 2.7 にアップグレードする目的でのみサポートされます。Streams for Apache Kafka の最新バージョンをインストールするときに、サポートされている最新バージョンの Kafka にアップグレードします。

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、Streams for Apache Kafka 2.7 on Red Hat Enterprise Linux Release Notes に記載されているアップグレードの変更点をよく理解していることを確認してください。

18.2. クライアントをアップグレードするストラテジー

Kafka クライアントをアップグレードすると、Kafka の新しいバージョンで導入された機能、修正、改善の恩恵を受けることができます。アップグレードされたクライアントは、他のアップグレードされた Kafka コンポーネントとの互換性を維持します。クライアントのパフォーマンスと安定性も向上する可能性があります。

スムーズな移行を確保するために、Kafka クライアントとブローカーをアップグレードするための最適なアプローチを検討してください。選択するアップグレード戦略は、ブローカーを最初にアップグレードするかクライアントを最初にアップグレードするかによって異なります。Kafka 3.0 以降、ブローカーとクライアントを独立して任意の順序でアップグレードできるようになりました。クライアントまたはブローカーをアップグレードするかどうかは、最初に、アップグレードする必要があるアプリケーションの数や許容できるダウンタイムの量など、いくつかの要因によって決まります。

ブローカーより前にクライアントをアップグレードすると、一部の新機能はブローカーによってまだサポートされていないため、機能しない可能性があります。ただし、ブローカーは、異なるバージョンで実行され、異なるログメッセージバージョンをサポートするプロデューサーとコンシューマーを処理できます。

18.3. KAFKA クラスターのアップグレード

KRaft ベースの Kafka クラスターを、サポートされている新しい Kafka バージョンおよび KRaft メタデータバージョンにアップグレードします。インストールファイルを更新し、すべての Kafka ノードを設定して再起動します。これらの手順の実行後に、新しいメタデータバージョンに従って Kafka ブローカー間でデータが送信されます。

警告

KRaft ベースの Strimzi Kafka クラスターを下位バージョンにダウングレードする場合 (3.7.0 から 3.6.0 への移行など)、Kafka クラスターで使用されるメタデータバージョンが、ダウングレードする Kafka バージョンでサポートされているバージョンであることを確認してください。ダウングレード元の Kafka バージョンのメタデータバージョンは、ダウングレード先のバージョンより高くすることはできません。

前提条件

- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。
- Streams for Apache Kafka が各ホストにインストールされ、設定ファイルが利用可能です。
- インストールファイル をダウンロードしている。

手順

Streams for Apache Kafka クラスターの各 Kafka ノードについて、コントローラーノードとブローカー、および一度に 1 つずつ開始します。

1. Streams for Apache Kafka ソフトウェアダウンロードページ から Streams for Apache Kafka アーカイブをダウンロード します。

注記

プロンプトが表示されたら、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-<version>-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-<version>-bin.zip -d /tmp/kafka
```

3. 実行中の場合は、ホスト上で実行している Kafka ブローカーを停止します。

```
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep kafka
```

マルチノードクラスターで Kafka を実行している場合は、「Kafka ブローカーの正常なローリング再起動の実行」 を参照してください。

4. 既存のインストールから **libs** および **bin** ディレクトリーを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin
```

5. 一時ディレクトリーから **libs** および **bin** ディレクトリーをコピーします。

```
cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/
cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
```

6. 必要に応じて、**config** ディレクトリー内の設定ファイルを更新して、新しい Kafka バージョンの変更を反映します。

7. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

8. 更新した Kafka ノードを再起動します。

ロールを組み合わせたノードの再起動

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

コントローラーノードの再起動

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

broker ロールを持つノードの再起動

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

Kafka ブローカーは、最新 Kafka バージョンのバイナリーの使用を開始します。

マルチノードクラスターでブローカーを再起動する方法は、「Kafka ブローカーの正常なローリング再起動の実行」 を参照してください。

9. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

10. Kafka メタデータのバージョンを更新します。

```
./bin/kafka-features.sh --bootstrap-server <broker_host>:<port> upgrade --metadata 3.7
```

アップグレード先の Kafka バージョンに応じた正しいバージョンを使用してください。

注記

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が同期していることを確認し、再起動した Kafka ブローカーが、フォローしているパーティションレプリカに追いついたことを確認してください。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。

18.4. KAFKA コンポーネントのアップグレード

ホストマシン上の Kafka コンポーネントをアップグレードして、最新バージョンの Streams for Apache Kafka を使用します。Streams for Apache Kafka インストールファイルを使用して、次のコンポーネントをアップグレードできます。

- Kafka Connect
- MirrorMaker
- Kafka Bridge (別の ZIP ファイル)

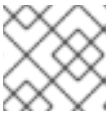
前提条件

- Red Hat Enterprise Linux に **kafka** ユーザーとしてログインしている。
- インストールファイル をダウンロードした。
- Kafka をアップグレード した。
  Kafka コンポーネントが Kafka と同じホストで実行されている場合は、アップグレード時に Kafka を停止して起動することも必要です。

手順

Kafka コンポーネントのインスタンスを実行している各ホストに、以下を行います。

1. Streams for Apache Kafka ソフトウェアダウンロードページ から、Streams for Apache Kafka または Kafka Bridge インストールファイルをダウンロード します。



注記

プロンプトが表示されたら、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-<version>-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-<version>-bin.zip -d /tmp/kafka
```

Kafka Bridge の場合は、**amq-streams-<version>-bridge-bin.zip** ファイルを抽出します。

3. 実行中の場合は、ホストで実行中の Kafka コンポーネントを停止します。

4. 既存のインストールから **libs** および **bin** ディレクトリーを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin
```

5. 一時ディレクトリーから **libs** および **bin** ディレクトリーをコピーします。

```
cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/
cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
```

6. 必要に応じて、**config** ディレクトリー内の設定ファイルを更新して、新しいバージョンの変更を反映します。

7. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

8. 適切なスクリプトとプロパティーファイルを使用して Kafka コンポーネントを起動します。

スタンドアロンモードでの Kafka Connect の起動

```
/opt/kafka/bin/connect-standalone.sh \
/opt/kafka/config/connect-standalone.properties <connector1>.properties
[<connector2>.properties ...]
```

分散モードでの Kafka Connect の開始

```
/opt/kafka/bin/connect-distributed.sh \
/opt/kafka/config/connect-distributed.properties
```

MirrorMaker 2 を専用モードで起動する

```
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

Kafka Bridge の起動

```
su - kafka
./bin/kafka_bridge_run.sh \
--config-file=<path>/application.properties
```

9. Kafka コンポーネントが実行中で、期待どおりにデータを生成または消費していることを確認します。

スタンドアロンモードで Kafka Connect が実行されていることを確認する

```
jcmd | grep ConnectStandalone
```

分散モードの Kafka Connect が実行されていることを確認する

```
jcmd | grep ConnectDistributed
```

MirrorMaker 2 が専用モードで実行されていることを確認する

```
jcmd | grep mirrorMaker
```

ログをチェックして Kafka Bridge が実行されていることを確認する

```
HTTP-Kafka Bridge started and listening on port 8080
HTTP-Kafka Bridge bootstrap servers localhost:9092
```

第19章 JMX を使用したクラスターの監視

メトリックを収集することは、Kafka デプロイメントの健全性とパフォーマンスを理解するために重要です。メトリックを監視することで、問題が重大になる前に積極的に特定し、リソースの割り当てとキャパシティープランニングについて情報に基づいた意思決定を行うことができます。メトリックがないと、Kafka デプロイメントの動作の可視性が制限される可能性があります。これによりトラブルシューティングがより困難になり、時間がかかる可能性があります。メトリックをセットアップすると、長期的には時間とリソースを節約でき、Kafka デプロイメントの信頼性を確保するのに役立ちます。

Kafka コンポーネントは、Java Management Extensions (JMX) を使用して、メトリクスにより管理情報を共有します。これらのメトリクスは、Kafka クラスターのパフォーマンスと全体的な健全性を監視するために重要です。他の多くの Java アプリケーションと同様に、Kafka は管理 Bean (MBean) を採用して、監視ツールやダッシュボードにメトリクスデータを提供します。JMX は JVM レベルで動作し、外部ツールが Kafka コンポーネントに接続して管理情報を取得できるようにします。通常、JVM に接続するには、デフォルトで同じマシンかつ同じユーザー権限でこれらのツールを実行する必要があります。

19.1. JMX エージェントの有効化

JVM システムプロパティーを使用して、Kafka コンポーネントの JMX 監視を有効にします。**KAFKA_JMX_OPTS** 環境変数を使用して、JMX モニタリングを有効にするために必要な JMX システムプロパティーを設定します。Kafka コンポーネントを実行するスクリプトは、これらのプロパティーを使用します。

手順

1. JMX モニタリングを有効にするために、**KAFKA_JMX_OPTS** 環境変数に JMX プロパティーを設定します。

```
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
```

   <port> を、Kafka コンポーネントが JMX 接続をリッスンするポートの名前に置き換えます。

2. **org.apache.kafka.common.metrics.JmxReporter** を **server.properties** ファイルの **metric.reporters** に追加します。

```
metric.reporters=org.apache.kafka.common.metrics.JmxReporter
```

3. ブローカーの場合は **bin/kafka-server-start.sh**、Kafka Connect の場合は **bin/connect-distributed.sh** など、適切なスクリプトを使用して Kafka コンポーネントを起動します。

重要

リモート JMX 接続を保護するために認証と SSL を設定することが推奨されます。これを行うために必要なシステムプロパティーの詳細は、Oracle のドキュメント を参照してください。

19.2. JMX エージェントの無効化

KAFKA_JMX_OPTS 環境変数を更新して、Kafka コンポーネントの JMX モニタリングを無効にします。

手順

1. **KAFKA_JMX_OPTS** 環境変数を設定して、JMX モニタリングを無効にします。

```
export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=false
```

注記

JMX モニタリングを無効にすると、ポート、認証、SSL プロパティーなどの他の JMX プロパティーを指定する必要はありません。

2. Kafka の **server.properties** ファイルで **auto.include.jmx.reporter** を **false** に設定します。

```
auto.include.jmx.reporter=false
```

注記

auto.include.jmx.reporter プロパティーは非推奨になりました。Kafka 4 以降、JMXReporter は、プロパティーファイルの **metric.reporters** 設定に **org.apache.kafka.common.metrics.JmxReporter** が追加されている場合にのみ有効になります。

3. ブローカーの場合は **bin/kafka-server-start.sh**、Kafka Connect の場合は **bin/connect-distributed.sh** など、適切なスクリプトを使用して Kafka コンポーネントを起動します。

19.3. メトリックの命名規則

Kafka JMX メトリックを使用する場合は、特定のメトリックを識別して取得するために使用される命名規則を理解することが重要です。Kafka JMX メトリックは次の形式を使用します。

メトリックの形式

```
<metric_group>:type=<type_name>,name=<metric_name><other_attribute>=<value>
```

- <metric_group> はメトリックグループの名前です。
- <type_name> はメトリックのタイプの名前です。
- <metric_name> は特定のメトリックの名前です。
- <other_attribute> は、ゼロ以上の追加属性を表します。

たとえば、**BytesInPerSec** メトリックは、**kafka.server** グループの **BrokerTopicMetrics** タイプです。

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
```

場合によっては、メトリックにエンティティーの ID が含まれる場合があります。たとえば、特定のクライアントを監視する場合、メトリック形式にはクライアント ID が含まれます。

特定のクライアントのメトリック

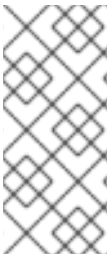
```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>
```

同様に、メトリックを特定のクライアントとトピックにさらに絞り込むことができます。

特定のクライアントとトピックのメトリック

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>,topic=<topic_id>
```

これらの命名規則を理解すると、監視および分析するメトリックを正確に指定できるようになります。



注記

Strimzi インストールで使用可能な JMX メトリックの完全なリストを表示するには、JConsole などのグラフィカルツールを使用できます。JConsole は、Kafka を含む Java アプリケーションを監視および管理できる Java 監視および管理コンソールです。プロセス ID を使用して Kafka コンポーネントを実行している JVM に接続すると、ツールのユーザーインターフェイスでメトリックのリストを表示できます。

19.4. トラブルシューティングのための KAFKA JMX メトリックの分析

JMX は、Kafka ブローカーのパフォーマンスとリソース使用量を監視および管理するために、Kafka ブローカーに関するメトリックを収集する方法を提供します。これらのメトリックを分析することで、高い CPU 使用率、メモリーリーク、スレッド競合、応答時間の遅さなどのブローカーの一般的な問題を診断して解決できます。特定のメトリックにより、これらの問題の根本原因を正確に特定できます。

JMX メトリックは、Kafka クラスターの全体的な健全性とパフォーマンスに関する洞察も提供します。これらは、システムのスループット、遅延、可用性の監視、問題の診断、パフォーマンスの最適化に役立ちます。このセクションでは、一般的な問題の特定に役立つ JMX メトリックの使用について説明し、Kafka クラスターのパフォーマンスについての洞察を提供します。

Prometheus や Grafana などのツールを使用してこれらのメトリックを収集してグラフ化すると、返された情報を視覚化できます。これは、問題の検出やパフォーマンスの最適化に特に役立ちます。時間の経過に伴うメトリクスをグラフ化することは、傾向の特定やリソース消費の予測にも役立ちます。

19.4.1. レプリケーションが不十分なパーティションのチェック

最適なパフォーマンスを得るには、バランスのとれた Kafka クラスターが重要です。バランスの取れたクラスターでは、パーティションとリーダーがすべてのブローカーに均等に分散され、I/O メトリックはこれを反映します。メトリックを使用するだけでなく、**kafka-topics.sh** ツールを使用して、レプリケーションが不十分なパーティションのリストを取得し、問題のあるブローカーを特定することもできます。レプリケーションが不十分なパーティションの数が変動している場合、または多くのブローカーが高いリクエスト遅延を示している場合、これは通常、調査が必要なクラスター内のパフォーマンスの問題を示しています。一方、クラスター内の多くのブローカーによって報告される、レプリケーションが不十分なパーティションの安定した (変化しない) 数は、通常、クラスター内のブローカーの 1 つがオフラインであることを示します。

kafka-topics.sh ツールの **description --under-replicated-partitions** オプションを使用して、クラスター内で現在アンダーレプリケートされているパーティションに関する情報を表示します。これらは、設定されたレプリケーション係数よりも少ないレプリカを持つパーティションです。

出力が空白の場合、Kafka クラスターには不十分にレプリケートされたパーティションはありません。それ以外の場合、出力には、同期していないか、使用できないレプリカが表示されます。

次の例では、各パーティションで 3 つのレプリカのうち 2 つだけが同期しており、ISR からレプリカが欠落しています (同期レプリカ)。

コマンドラインからレプリケーションが不十分なパーティションに関する情報を返す

```
bin/kafka-topics.sh --bootstrap-server :9092 --describe --under-replicated-partitions

Topic: topic-1 Partition: 0 Leader: 4 Replicas: 4,2,3 Isr: 4,3
Topic: topic-1 Partition: 1 Leader: 3 Replicas: 2,3,4 Isr: 3,4
Topic: topic-1 Partition: 2 Leader: 3 Replicas: 3,4,2 Isr: 3,4
```

I/O およびレプリケーションが不十分なパーティションをチェックするためのいくつかのメトリックを次に示します。

レプリケーションが不十分なパーティションをチェックするためのメトリック

```
kafka.server:type=ReplicaManager,name=PartitionCount 1
kafka.server:type=ReplicaManager,name=LeaderCount 2
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec 3
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec 4
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions 5
kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount 6
```

1. クラスター内のすべてのトピックにわたるパーティションの合計数。
2. クラスター内のすべてのトピックにわたるリーダーの合計数。
3. 各ブローカーの 1 秒あたりの受信バイト数の割合。
4. 各ブローカーの 1 秒あたりの送信バイト数。
5. クラスター内のすべてのトピックにわたる、レプリケーションが不十分なパーティションの数。
6. 最小 ISR を下回るパーティションの数。

トピック設定が高可用性向けに設定されており、トピックのレプリケーション係数が少なくとも 3 で、同期レプリカの最小数がレプリケーション係数より 1 少ない場合は、レプリケーションが不十分なパーティションも引き続き使用できます。逆に、最小 ISR を下回るパーティションでは可用性が低下します。これらは、**kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount** メトリクスと、**kafka -topics.sh** ツールの **under-min-isr-partitions** オプションを使用して監視できます。

ヒント

Cruise Control を使用して、Kafka クラスターの監視と再バランスのタスクを自動化し、パーティションの負荷が均等に分散されるようにします。詳細は、*13章Cruise Control を使用したクラスターのリバランス* を参照してください。

19.4.2. Kafka クラスター内のパフォーマンスの問題を特定する

クラスターメトリックのスパイクは、ブローカーの問題を示している可能性があります。これは、多くの場合、ストレージデバイスの低速または障害、または他のプロセスからのコンピューティング制限に関連しています。オペレーティングシステムまたはハードウェアレベルに問題がない場合は、同じ

Kafka トピック内の他のパーティションと比較して、一部のパーティションが不均衡なトラフィックを受信し、Kafka クラスターの負荷が不均衡になっている可能性があります。

Kafka クラスターのパフォーマンスの問題を予測するには、**RequestHandlerAvgIdlePercent** メトリックを監視すると便利です。**RequestHandlerAvgIdlePercent** は、クラスターがどのように動作しているかを総合的に示す優れた指標となります。このメトリックの値は 0 から 1 です。0.7 未満の値は、スレッドが時間の 30% でビジー状態であり、パフォーマンスが低下し始めていることを示します。値が 50% を下回ると、特にクラスターのスケーリングや再バランスが必要な場合に問題が発生する可能性があります。30% では、クラスターはほとんど使用できません。

もう 1 つの便利なメトリクスは **kafka.network:type=Processor,name=IdlePercent** です。これを使用すると、Kafka クラスター内のネットワークプロセッサーがアイドル状態になっている範囲 (パーセンテージとして) を監視できます。このメトリクスは、プロセッサーが過剰に使用されているか、十分に使用されていないかを識別するのに役立ちます。

最適なパフォーマンスを確保するには、**num.io.threads** プロパティーをシステム内のプロセッサー (ハイパースレッドプロセッサーを含む) の数に設定します。クラスターのバランスは取れているが、単一のクライアントが要求パターンを変更して問題を引き起こしている場合は、クラスターの負荷を軽減するか、ブローカーの数を増やしてください。

単一ブローカー上の単一ディスク障害がクラスター全体のパフォーマンスに重大な影響を与える可能性があることに注意することが重要です。プロデューサークライアントは、トピックのパーティションをリードするすべてのブローカーに接続し、それらのパーティションはクラスター全体に均等に分散されているため、ブローカーのパフォーマンスが低下すると、プロデュースリクエストが遅くなり、プロデューサーにバックプレッシャーが発生し、すべてのブローカーへのリクエストが遅くなります。複数の物理ディスクドライブを 1 つの論理ユニットに結合する RAID (Redundant Array of Inexpensive Disks) ストレージ設定は、この問題を防ぐのに役立ちます。

Kafka クラスターのパフォーマンスをチェックするためのいくつかのメトリックを次に示します。

Kafka クラスターのパフォーマンスをチェックするためのメトリック

```
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent (1)
# attributes: OneMinuteRate, FifteenMinuteRate
kafka.server:type=socket-server-metrics,listener=([-.\w]+),networkProcessor=([\d]+) (2)
# attributes: connection-creation-rate
kafka.network:type=RequestChannel,name=RequestQueueSize (3)
kafka.network:type=RequestChannel,name=ResponseQueueSize (4)
kafka.network:type=Processor,name=IdlePercent,networkProcessor=([-.\w]+) (5)
kafka.server:type=KafkaServer,name=TotalDiskReadBytes (6)
kafka.server:type=KafkaServer,name=TotalDiskWriteBytes (7)
```

(1) Kafka ブローカーのスレッドプール内のリクエストハンドラースレッドの平均アイドル率。**OneMinuteRate** 属性と **FifteenMinuteRate** 属性は、それぞれ過去 1 分間と 15 分間のリクエストレートを示します。

(2) Kafka ブローカーの特定のリスナーの特定のネットワークプロセッサー上で新しい接続が作成される速度。**listen** 属性はリスナーの名前を参照し、**networkProcessor** 属性はネットワークプロセッサーの ID を参照します。**connection-creation-rate** 属性は、1 秒あたりの接続数での接続作成速度を示します。

(3) リクエストキューの現在のサイズ。

(4) 応答キューの現在のサイズ。

5 指定された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割合。**networkProcessor** は、監視するネットワークプロセッサーの ID を指定します。

6 Kafka サーバーによってディスクから読み取られた合計バイト数。

7 Kafka サーバーによってディスクに書き込まれた合計バイト数。

19.4.3. Kafka コントローラーのパフォーマンスの問題を特定する

Kafka コントローラーは、ブローカーの登録、パーティションの再割り当て、トピック管理など、クラスターの全体的な状態の管理を担当します。Kafka クラスター内のコントローラーの問題は診断が難しく、多くの場合、Kafka 自体のバグのカテゴリーに分類されます。コントローラーの問題は、ブローカーのメタデータが同期していない、ブローカーが正常に見えてもレプリカがオフラインになっている、トピックの作成などのトピックに対するアクションが正しく行われていないなどとして現れる可能性があります。

コントローラーを監視する方法はそれほど多くありませんが、アクティブなコントローラー数とコントローラーキューサイズを監視できます。これらのメトリックを監視すると、問題がある場合に大まかな指標が得られます。キューサイズの急増が予想されますが、この値が継続的に増加するか、高い値で安定して低下しない場合は、コントローラーがスタックしている可能性があることを示しています。この問題が発生した場合は、コントローラーを別のブローカーに移動できます。その場合、現在コントローラーであるブローカーをシャットダウンする必要があります。

Kafka コントローラーのパフォーマンスをチェックするためのメトリックをいくつか示します。

Kafka コントローラーのパフォーマンスをチェックするためのメトリック

```
kafka.controller:type=KafkaController,name=ActiveControllerCount 1
kafka.controller:type=KafkaController,name=OfflinePartitionsCount 2
kafka.controller:type=ControllerEventManager,name=EventQueueSize 3
```

1 Kafka クラスター内のアクティブなコントローラーの数。値 1 は、アクティブなコントローラーが 1 つだけ存在することを示し、これが望ましい状態です。

2 現在オフラインになっているパーティションの数。この値が継続的に増加するか、高い値が続く場合は、コントローラーに問題がある可能性があります。

3 コントローラー内のイベントキューのサイズ。イベントは、新しいトピックの作成やパーティションの新しいブローカーへの移動など、コントローラーによって実行される必要があるアクションです。値が継続的に増加するか、高い値に留まる場合は、コントローラーがスタックして必要なアクションを実行できない可能性があります。

19.4.4. リクエストの問題の特定

RequestHandlerAvgIdlePercent メトリクスを使用して、リクエストが遅いかどうかを判断できます。さらに、リクエストメトリクスにより、どの特定のリクエストで遅延やその他の問題が発生しているかを特定できます。

Kafka リクエストを効果的に監視するには、カウントと 99 パーセンタイルレイテンシー (テイルレイテンシーとも呼ばれます) という 2 つの主要なメトリックを収集することが重要です。

count メトリクスは、特定の時間間隔で処理されるリクエストの数を表します。これは、Kafka クラスターによって処理されるリクエストの量に関する洞察を提供し、トラフィックの急増または低下を特定するのに役立ちます。

99 パーセンタイルレイテンシーメメトリクスは、リクエストの処理にかかる時間であるリクエストレイテンシーを測定します。これは、リクエストの 99% が処理される期間を表します。ただし、残りの 1% のリクエストの正確な期間に関する情報は提供されません。つまり、99 パーセンタイルの遅延メトリックは、リクエストの 99% が特定の期間内に処理され、残りの 1% にはさらに時間がかかる可能性があることを示しています。ただし、この残り 1% の正確な期間は不明です。99 パーセンタイルの選択は、主にリクエストの大部分に焦点を当て、結果を歪める可能性のある外れ値を除外することを目的としています。

このメトリクスは、大部分のリクエストに関連するパフォーマンスの問題やボトルネックを特定するのに特に役立ちますが、リクエストのごく一部で発生する最大レイテンシーの全体像を示すものではありません。

カウントと 99 パーセンタイルレイテンシーメトリクスの両方を収集して分析することで、Kafka クラスターの全体的なパフォーマンスと健全性、および処理中のリクエストのレイテンシーを把握できます。

Kafka リクエストのパフォーマンスをチェックするためのいくつかのメトリックを次に示します。

リクエストのパフォーマンスをチェックするためのメトリック

```
# requests: EndTxn, Fetch, FetchConsumer, FetchFollower, FindCoordinator, Heartbeat,
InitProducerId,
# JoinGroup, LeaderAndIsr, LeaveGroup, Metadata, Produce, SyncGroup, UpdateMetadata (1)
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=([\w]+) (2)
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=([\w]+) (3)
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=([\w]+) (4)
kafka.network:type=RequestMetrics,name=LocalTimeMs,request=([\w]+) (5)
kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=([\w]+) (6)
kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=([\w]+) (7)
kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=([\w]+) (8)
kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=([\w]+) (9)
# attributes: Count, 99thPercentile (10)
```

(1) リクエストのメトリックを分類するためのリクエストのタイプ。

(2) Kafka ブローカーによってリクエストが処理される 1 秒あたりの速度。

(3) リクエストが処理されるまでにブローカーのリクエストキューで待機するのに費やした時間 (ミリ秒)。

(4) リクエストがブローカーによって受信されてから、レスポンスがクライアントに送り返されるまで、リクエストが完了するまでにかかる合計時間 (ミリ秒単位)。

(5) リクエストがローカルマシン上のブローカーによって処理されるのに費やした時間 (ミリ秒)。

(6) リクエストがクラスター内の他のブローカーによって処理されるのに費やした時間 (ミリ秒)。

(7) リクエストがブローカーによって調整されるのに費やした時間 (ミリ秒単位)。スロットリングは、クライアントがあまりにも多くのリクエストをあまりにも早く送信しているため、速度を落とす必要があるとブローカーが判断した場合に発生します。

8 応答がクライアントに送り返されるまでにブローカーの応答キューで待機するのにかかる時間 (ミリ秒)。

9 応答がブローカーによって生成されてからクライアントに返送されるまでにかかる時間 (ミリ秒)。

10 すべてのリクエストメトリックについて、**Count** 属性と **99thPercentile** 属性は、それぞれ、処理されたリクエストの合計数と、最も遅い 1% のリクエストが完了するまでにかかる時間を示します。

19.4.5. メトリックを使用してクライアントのパフォーマンスをチェックする

クライアントメトリックを分析することで、ブローカーに接続されている Kafka クライアント (プロデューサーとコンシューマー) のパフォーマンスを監視できます。これは、コンシューマーグループから頻繁に除外されるコンシューマー、高いリクエスト失敗率、頻繁な切断など、ブローカーログで強調表示される問題を特定するのに役立ちます。

Kafka クライアントのパフォーマンスをチェックするためのいくつかのメトリックを次に示します。

クライアントリクエストのパフォーマンスをチェックするためのメトリック

```
kafka.consumer:type=consumer-metrics,client-id=([-.\w]+) 1
# attributes: time-between-poll-avg, time-between-poll-max
kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+) 2
# attributes: heartbeat-response-time-max, heartbeat-rate, join-time-max, join-rate, rebalance-rate-per-hour
kafka.producer:type=producer-metrics,client-id=([-.\w]+) 3
# attributes: buffer-available-bytes, bufferpool-wait-time, request-latency-max, requests-in-flight
# attributes: txn-init-time-ns-total, txn-begin-time-ns-total, txn-send-offsets-time-ns-total, txn-commit-time-ns-total, txn-abort-time-ns-total
# attributes: record-error-total, record-queue-time-avg, record-queue-time-max, record-retry-rate, record-retry-total, record-send-rate, record-send-total
```

1 (コンシューマー) ポーリングリクエスト間の平均時間と最大時間。これは、コンシューマーがメッセージフローに追いつくのに十分な頻度でメッセージをポーリングしているかどうかを判断するのに役立ちます。**time-between-poll-avg** 属性と **time-between-poll-max** 属性は、コンシューマーによる連続したポーリング間の平均時間と最大時間をそれぞれミリ秒単位で示します。

2 (コンシューマー) Kafka コンシューマーとブローカーコーディネーターの間の調整プロセスを監視するメトリック。属性は、ハートビート、結合、およびリバランスのプロセスに関連します。

3 (プロデューサー) Kafka プロデューサーのパフォーマンスを監視するメトリック。属性は、バッファーの使用状況、リクエストのレイテンシー、実行中のリクエスト、トランザクション処理、およびレコード処理に関連します。

19.4.6. メトリックを使用してトピックとパーティションのパフォーマンスをチェックする

トピックとパーティションのメトリックは、Kafka クラスターの問題の診断にも役立ちます。また、クライアントメトリクスを収集できない場合に、特定のクライアントに関する問題をデバッグするためにこれらを使用することもできます。

特定のトピックとパーティションのパフォーマンスをチェックするためのいくつかのメトリックを次に示します。

トピックとパーティションのパフォーマンスをチェックするためのメトリック

```
#Topic metrics
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=([-.\w]+) 1
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=([-.\w]+) 2
kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=([-.\w]+) 3
kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=([-.\w]+) 4
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=([-.\w]+) 5
kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=([-.\w]+) 6
kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=([-.\w]+) 7
#Partition metrics
kafka.log:type=Log,name=Size,topic=([-.\w]+),partition=([\d]+)) 8
kafka.log:type=Log,name=NumLogSegments,topic=([-.\w]+),partition=([\d]+)) 9
kafka.log:type=Log,name=LogEndOffset,topic=([-.\w]+),partition=([\d]+)) 10
kafka.log:type=Log,name=LogStartOffset,topic=([-.\w]+),partition=([\d]+)) 11
```

1. 特定のトピックの 1 秒あたりの受信バイト数の割合。
2. 特定のトピックの 1 秒あたりの送信バイト数の割合。
3. 特定のトピックに対する 1 秒あたりの失敗したフェッチリクエストの割合。
4. 特定のトピックについて 1 秒あたりに失敗したプロデュースリクエストの割合。
5. 特定のトピックの 1 秒あたりの受信メッセージ率。
6. 特定のトピックに対する 1 秒あたりのフェッチリクエスト (成功および失敗) の合計速度。
7. 特定のトピックに対する 1 秒あたりのフェッチリクエスト (成功および失敗) の合計速度。
8. 特定のパーティションのログのサイズ (バイト単位)。
9. 特定のパーティション内のログセグメントの数。
10. 特定のパーティションのログ内の最後のメッセージのオフセット。
11. 特定のパーティションのログ内の最初のメッセージのオフセット

関連情報

- 利用可能なメトリックの完全なリストは、Apache Kafka ドキュメント を参照してください。
- Prometheus ドキュメント
- Grafana のドキュメント

付録A サブスクリプションの使用

Streams for Apache Kafka 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. **My Subscriptions** 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、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. **Streams for Apache Kafka** エントリーの場所を **INTEGRATION AND AUTOMATION** カテゴリーで特定します。
3. 必要な Streams for Apache Kafka 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。

DNF を使用したパッケージのインストール

パッケージとすべてのパッケージ依存関係をインストールするには、以下を使用します。

```
dnf install <package_name>
```

ローカルディレクトリーからダウンロード済みのパッケージをインストールするには、以下を使用します。

```
dnf install <path_to_download_package>
```

改訂日時: 2024-04-30