Red Hat Streams for Apache Kafka 2.7

Kafka クライアントアプリケーションの開発

AMQ Streams を使用して Kafka とやりとりするクライアントアプリケーションを開発する

Last Updated: 2024-05-29

Red Hat Streams for Apache Kafka 2.7 Kafka クライアントアプリケーションの開発

AMQ Streams を使用して Kafka とやりとりするクライアントアプリケーションを開発する

法律上の通知

概要

Kafka ブローカーを介してメッセージを送受信できるクライアントアプリケーションを開発します。クライアントとブローカーの間のセキュアなアクセスを設定します。

目次

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 開発中のクライアントの概要

メッセージの生成、メッセージの消費、またはその両方を実行できる、Streams for Apache Kafka インストール用の Kafka クライアントアプリケーションを開発します。OpenShift 上の Streams for Apache Kafka または Streams for Apache Kafka on RHEL で使用するクライアントアプリケーションを開発できます。

メッセージは、オプションのキー、メッセージデータを含む値、さらにヘッダーと関連メタデータで設定されます。キーはメッセージの件名、またはメッセージのプロパティーを識別します。メッセージのグループを送信時と同じ順序で処理する必要がある場合は、同じキーを使用する必要があります。

メッセージはバッチで配信されます。メッセージには、メッセージのタイムスタンプやオフセット位置など、クライアントによるフィルタリングやルーティングに役立つ詳細を提供するヘッダーとメタデータが含まれています。

Kafka は、クライアントアプリケーションを開発するためのクライアント API を提供します。Kafka プロデューサー API とコンシューマー API は、クライアントアプリケーションで Kafka クラスターとやりとりする主な手段です。API はメッセージのフローを制御します。プロデューサー API は Kafka トピックにメッセージを送信し、コンシューマー API はトピックからメッセージを読み取ります。

Streams for Apache Kafka は、Java で記述されたクライアントをサポートします。クライアントをどのように開発するかは、特定のユースケースによって異なります。データの耐久性が優先される場合や、高スループットが優先される場合があります。これらの要求は、クライアントとブローカーの設定を通じて満たすことができます。ただし、すべてのクライアントは、特定の Kafka クラスター内のすべてのブローカーに接続できる必要があります。

1.1. HTTP クライアントのサポート

クライアントで Kafka プロデューサー API とコンシューマー API を使用する代わりに、Streams for Apache Kafka Kafka Bridge をセットアップして使用できます。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。これは、Kafka プロトコルを解釈する必要があるクライアントアプリケーションを必要とせずに、Strimzi への Web API 接続の利点を提供します。Kafka は TCP 経由でバイナリープロトコルを使用します。

詳細は、Streams for Apache Kafka Kafka Bridge の使用 を 参照してください。

1.2. プロデューサーとコンシューマーの調整

さらに設定プロパティーを追加して、Kafka クライアントのパフォーマンスを最適化できます。これは、クライアントとブローカーの設定がどのように実行されるかを分析する時間があるときに行うとよいでしょう。

詳細は、Kafka 設定のチューニング を参照してください。

1.3. クライアントとのやりとりを監視する

分散トレースにより、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。Kafka コンシューマーおよびプロデューサークライアントアプリケーションでトレースを有効にすることができます。

詳細は、次のガイドの分散トレースに関するドキュメントを参照してください。

- OpenShift での Apache Kafka のデプロイおよびアップグレード
- KRaft モードの RHEL での Streams for Apache Kafka の使用

- ZooKeeper で RHEL での Streams for Apache Kafka の使用

注記

クライアントアプリケーションという用語を使用する場合、特に、Kafka プロデューサーとコンシューマーを使用して Kafka クラスターとの間でメッセージを送受信するアプリケーションを指します。ここでは、独自の使用例や機能を持つ Kafka Connect や Kafka Streams などの他の Kafka コンポーネントは言及しません。

第2章 クライアント開発の前提条件

Streams for Apache Kafka で使用するクライアントを開発するには、以下の前提条件を満たす必要があります。

- Red Hat アカウントを持っている。
- Streams for Apache Kafka で Kafka クラスターが実行されている。
- Kafka ブローカーは、セキュアなクライアント接続のためにリスナーを使用して設定されている。
- クラスター用のトピックが作成されている。
- クライアントを開発およびテストするための IDE がある。
- JDK 11 以降がインストールされている。

第3章 MAVEN プロジェクトにクライアントの依存関係を追加する

Java ベースの Kafka クライアントを開発している場合は、Kafka ストリームを含む Kafka クライアントの Red Hat 依存関係を Maven プロジェクトの **pom.xml** ファイルに追加できます。Streams for Apache Kafka では、Red Hat によって構築されたクライアントライブラリーのみがサポートされます。

次のアーティファクトを依存関係として追加できます。

kafka-clients

Kafka の **Producer**、**Consumer**、および **AdminClient** API が含まれています。

- **Producer** API を使用すると、アプリケーションは Kafka ブローカーにデータを送信できます。
- **Consumer** API を使用すると、アプリケーションは Kafka ブローカーからのデータを消費できるようになります。
- **AdminClient** API は、トピック、ブローカー、その他のコンポーネントを含む、Kafka クラスターを管理するための機能を提供します。

kafka-streams

KafkaStreams API が含まれています。
Kafka Streams を使用すると、アプリケーションは 1 つ以上の入力ストリームからデータを受信できます。この API を使用すると、データのストリームに対してマッピング、フィルタリング、結合などの一連のリアルタイム操作を実行できます。Kafka Streams を使用して、結果を 1 つ以上の出力ストリームに書き込むことができます。これは、Red Hat Maven リポジトリーで利用可能な **kafka-streams** JAR パッケージの一部です。

3.1. KAFKA クライアントの依存関係を MAVEN プロジェクトに追加する

Kafka クライアントの Red Hat 依存関係を Maven プロジェクトに追加します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリーを Maven プロジェクトの **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<repositories>
  <repository>
    <id>redhat-maven</id>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
```

2. **kafka-clients** を **<dependency>** として Maven プロジェクトの **pom.xml** ファイルに追加します。

```
<dependencies>
  <dependency>
```

```
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>3.7.0.redhat-00004</version>
    </dependency>
</dependencies>
```

3. Maven プロジェクトをビルドして、Kafka クライアントの依存関係をプロジェクトに追加します。

3.2. KAFKA STREAMS 依存関係を MAVEN プロジェクトに追加する

Kafka Streams の Red Hat 依存関係を Maven プロジェクトに追加します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリーを Maven プロジェクトの **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<repositories>
    <repository>
        <id>redhat-maven</id>
        <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
</repositories>
```

2. **kafka-streams** を **<dependency>** として Maven プロジェクトの **pom.xml** ファイルに追加します。

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>3.7.0.redhat-00004</version>
    </dependency>
</dependencies>
```

3. Maven プロジェクトをビルドして、Kafka Streams の依存関係をプロジェクトに追加します。

3.3. OAUTH 2.0 依存関係を MAVEN プロジェクトに追加する

OAuth 2.0 の Red Hat 依存関係を Maven プロジェクトに追加します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリーを Maven プロジェクトの **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<repositories>
    <repository>
        <id>redhat-maven</id>
        <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
</repositories>
```

2. **kafka-oauth-client** を **<dependency>** として Maven プロジェクトの **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>
```

3. Maven プロジェクトをビルドして、OAuth 2.0 の依存関係をプロジェクトに追加します。

第4章 KAFKA クラスターに接続するためのクライアントアプリケーションの設定

Kafka クラスターに接続するには、ブローカーを識別して接続を有効にする最小限のプロパティーのセットを使用してクライアントアプリケーションを設定する必要があります。さらに、メッセージを Kafka で使用されるバイト配列形式に変換したり、その形式からメッセージを変換したりするためのシリアライザー/デシリアライザーメカニズムを追加する必要があります。コンシューマークライアントを開発する場合は、最初の接続を Kafka クラスターに追加することから始めます。これは、使用可能なすべてのブローカーを検出するために使用されます。接続を確立したら、Kafka トピックからのメッセージの消費、および Kafka トピックへのメッセージの生成を開始できます。

必須ではありませんが、ログとメトリックの収集でクライアントを識別できるように、一意のクライアント ID を使用することを推奨します。

プロパティーファイルでプロパティーを設定できます。プロパティーファイルを使用すると、コードを再コンパイルせずに設定を変更できます。

たとえば、次のコードを使用して Java クライアントにプロパティーをロードできます。

設定プロパティーをクライアントにロードする

```
Properties props = new Properties();
try (InputStream propStream = Files.newInputStream(Paths.get(filename))) {
    props.load(propStream);
}
```

設定オブジェクトのコードにプロパティーを直接追加することもできます。たとえば、Java クライアントアプリケーションに **setProperty()** メソッドを使用できます。プロパティーを直接追加することは、設定するプロパティーの数が少ない場合に便利なオプションです。

4.1. 基本的なプロデューサークライアントの設定

プロデューサークライアントを開発するときは、次のように設定します。

- Kafka クラスターへの接続
- メッセージキーを Kafka ブローカーのバイトに変換するシリアライザー
- メッセージ値を Kafka ブローカーのバイトに変換するシリアライザー

圧縮されたメッセージを送信および保存する場合は、圧縮タイプを追加することもできます。

基本的なプロデューサークライアント設定プロパティー

```
client.id = my-producer-id (1)
bootstrap.servers = my-cluster-kafka-bootstrap:9092 (2)
key.serializer = org.apache.kafka.common.serialization.StringSerializer (3)
value.serializer = org.apache.kafka.common.serialization.StringSerializer (4)
```

1. クライアントの論理名。
2. クライアントが Kafka クラスターへの初期接続を確立できるようにするためのブートストラップアドレス。

3 Kafka ブローカーに送信される前にメッセージキーをバイトに変換するシリアライザー。

4 Kafka ブローカーに送信される前にメッセージ値をバイトに変換するシリアライザー。

プロデューサークライアント設定をコードに直接追加する

```
Properties props = new Properties();
props.setProperty(ProducerConfig.CLIENT_ID_CONFIG, "my-producer-id");
props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "my-cluster-kafka-
bootstrap:9092");
props.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
KafkaProducer<String, String> producer = new KafkaProducer<>(properties);
```

KafkaProducer は、送信するメッセージの文字列キーと値のタイプを指定します。使用されるシリアライザーは、Kafka に送信する前に、指定された型のキーと値をバイトに変換できる必要があります。

4.2. 基本的なコンシューマークライアントの設定

コンシューマークライアントを開発する場合は、次のように設定します。

- Kafka クラスターへの接続
- Kafka ブローカーから取得したバイトを、クライアントアプリケーションが理解できるメッセージキーに変換するデシリアライザー
- Kafka ブローカーから取得したバイトを、クライアントアプリケーションが理解できるメッセージ値に変換するデシリアライザー

通常、コンシューマーグループ ID も追加して、コンシューマーをコンシューマーグループに関連付けます。コンシューマーグループは、1 つ以上のトピックからの大きなデータストリームの処理を並列コンシューマーに分散するための論理エンティティーです。コンシューマーは **group.id** でグループ化され、メッセージをメンバー全体に分散できます。特定のコンシューマーグループでは、各トピックパーティションが単一のコンシューマーによって読み取られます。1 人のコンシューマーが多くのパーティションを処理できます。並列処理を最大限に高めるには、パーティションごとに 1 つのコンシューマーを作成します。パーティションより多くのコンシューマーが存在する場合、一部のコンシューマーはアイドル状態のままになり、障害が発生した場合に引き継げるように準備が整っています。

基本的なコンシューマークライアント設定プロパティー

```
client.id = my-consumer-id 1
group.id = my-group-id 2
bootstrap.servers = my-cluster-kafka-bootstrap:9092 3
key.deserializer = org.apache.kafka.common.serialization.StringDeserializer 4
value.deserializer = org.apache.kafka.common.serialization.StringDeserializer 5
```

1 クライアントの論理名。

2 コンシューマーが特定のコンシューマーグループに参加できるようにするためのグループ ID。

3

クライアントが Kafka クラスターへの初期接続を確立できるようにするためのブートストラップアドレス。

4 Kafka ブローカーから取得したバイトをメッセージキーに変換するデシリアライザー。

5 Kafka ブローカーから取得したバイトをメッセージ値に変換するデシリアライザー。

コンシューマークライアント設定をコードに直接追加する

```
Properties props = new Properties();
props.setProperty(ConsumerConfig.CLIENT_ID_CONFIG, "my-consumer-id");
props.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "my-group-id");
props.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "my-cluster-kafka-
bootstrap:9092");
props.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
props.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

KafkaConsumer は、受信するメッセージの文字列キーと値のタイプを指定します。使用されるシリアライザーは、Kafka から受信したバイトを指定された型に変換できる必要があります。

注記

各コンシューマーグループには一意の **group.id** が必要です。同じ **group.id** を持つコンシューマーを再起動すると、停止する前に中断したところからメッセージの消費が再開されます。

第5章 セキュアな接続の設定

Kafka クラスターとクライアントアプリケーション間の接続を保護すると、クラスターとクライアント間の通信の機密性、整合性、信頼性が確保されます。

セキュアな接続を実現するには、認証、暗号化、および承認に関連する設定を導入できます。

認証

認証メカニズムを使用して、クライアントアプリケーションの ID を検証します。

暗号化

SSL/TLS 暗号化を使用して、クライアントとブローカーの間で転送されるデータの暗号化を有効にします。

承認

クライアントアプリケーションの認証された ID に基づいて、Kafka ブローカーで許可されるクライアントアクセスと操作を制御します。

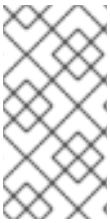
認可は認証なしでは使用できません。認証が有効になっていない場合は、クライアントの ID を判断できないため、認可ルールを強制することができません。これは、認可ルールが定義されていても、認証がなければ適用されないことを意味します。

Streams for Apache Kafka では、リスナーは Kafka ブローカーとクライアントの間のネットワーク接続を設定するために使用されます。リスナー設定オプションは、ブローカーが受信クライアント接続をリッスンする方法、およびセキュアなアクセスを管理する方法を決定します。必要な正確な設定は、選択した認証、暗号化、および承認メカニズムによって異なります。

Kafka ブローカーとクライアントアプリケーションを設定して、セキュリティー機能を有効にします。Kafka クラスターへのクライアント接続を保護するための一般的な概要は次のとおりです。

1. Kafka クラスターを含む Streams for Apache Kafka コンポーネントをインストールします。
2. TLS の場合、ブローカーとクライアントアプリケーションごとに TLS 証明書を生成します。
3. セキュアな接続のためにブローカー設定でリスナーを設定します。
4. クライアントアプリケーションをセキュアな接続用に設定します。

Kafka ブローカーとのセキュアで認証された接続を確立するために使用しているメカニズムに従ってクライアントアプリケーションを設定します。Kafka ブローカーによって使用される認証、暗号化、および承認は、接続するクライアントアプリケーションによって使用されるものと一致する必要があります。クライアントアプリケーションとブローカーは、セキュアな通信を行うためにセキュリティープロトコルと設定について合意する必要があります。たとえば、Kafka クライアントと Kafka ブローカーは同じ TLS バージョンと暗号スイートを使用する必要があります。



注記

クライアントとブローカーの間のセキュリティー設定が一致しないと、接続障害が発生したり、セキュリティー上の脆弱性が発生したりする可能性があります。ブローカーとクライアントアプリケーションの両方を慎重に設定およびテストして、それらが適切に保護され、セキュアに通信できることを確認することが重要です。

5.1. セキュアなアクセスのためのブローカーのセットアップ

セキュアなアクセスのためにクライアントアプリケーションを設定するには、まず、使用するセキュリティーメカニズムをサポートするように Kafka クラスター内のブローカーをセットアップする必要があ

ります。セキュアな接続を有効にするには、セキュリティーメカニズムに適切な設定を使用してリスナーを作成します。

5.1.1. RHEL 上で実行されている Kafka クラスターへのセキュアな接続の確立

RHEL で Streams for Apache Kafka を使用する場合、Kafka クラスターへのクライアント接続を保護するための一般的な概要は次のとおりです。

1. Kafka クラスターを含む Streams for Apache Kafka コンポーネントを RHEL サーバーにインストールします。
2. TLS の場合、Kafka クラスター内のすべてのブローカーの TLS 証明書を生成します。
3. ブローカー設定プロパティーファイルでリスナーを設定します。
   - Kafka クラスターリスナーの認証 (TLS や SASL SCRAM-SHA-512 など) を設定します。
   - Kafka クラスター上のすべての有効なリスナーに対する承認 (**simple** 承認など) を設定します。
4. TLS の場合、クライアントアプリケーションごとに TLS 証明書を生成します。
5. **config.properties** ファイルを作成して、クライアントアプリケーションで使用される接続の詳細と認証情報を指定します。
6. Kafka クライアントアプリケーションを起動し、Kafka クラスターに接続します。
   - **config.properties** ファイルに定義されたプロパティーを使用して、Kafka ブローカーに接続します。
7. クライアントが Kafka クラスターに正常に接続し、メッセージをセキュアに消費および生成できることを確認します。

関連情報

ブローカーをセットアップする方法の詳細は、次のガイドを参照してください。

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

5.1.2. RHEL での Kafka クラスターのセキュアなリスナーの設定

設定プロパティーファイルを使用して、Kafka でリスナーを設定します。Kafka ブローカーのセキュアな接続を設定するには、TLS、SASL、およびその他のセキュリティー関連の設定に関連するプロパティーをこのファイルに設定します。

以下は、PKCS#12 形式のキーストアとトラストストアを使用して、Kafka ブローカーの **server.properties** 設定ファイルで指定された TLS リスナーの設定例です。

server.properties のリスナー設定の例

```
listeners = listener_1://0.0.0.0:9093, listener_2://0.0.0.0:9094
listener.security.protocol.map = listener_1:SSL, listener_2:PLAINTEXT
ssl.keystore.type = PKCS12
ssl.keystore.location = /path/to/keystore.p12
```

```
ssl.keystore.password = <password>
ssl.truststore.type = PKCS12
ssl.truststore.location = /path/to/truststore.p12
ssl.truststore.password = <password>
ssl.client.auth = required
authorizer.class.name = kafka.security.auth.SimpleAclAuthorizer.
super.users = User:superuser
```

listeners プロパティーでは、各リスナー名、ブローカーがリッスンする IP アドレスとポートを指定します。プロトコルマップは、TLS 暗号化を使用するクライアントに対して SSL プロトコルを使用するように **listener_1** リスナーに指示します。**listener_2** は、TLS 暗号化を使用しないクライアントに PLAINTEXT 接続を提供します。キーストアには、ブローカーの秘密鍵と証明書が含まれています。トラストストアには、クライアントアプリケーションの ID を検証するために使用される信頼された証明書が含まれています。**ssl.client.auth** プロパティーはクライアント認証を強制します。

Kafka クラスターは単純な承認を使用します。オーソライザーは **SimpleAclAuthorizer** に設定されます。すべてのリスナーに対する無制限のアクセスのために、単一のスーパーユーザーが定義されています。Streams for Apache Kafka は、Kafka **SimpleAclAuthorizer** およびカスタムオーソライザープラグインをサポートします。

設定プロパティーの前に **listener.name.<name_of_listener>** を付けると、設定はそのリスナーに固有になります。

これは単なる設定例です。一部の設定オプションはリスナーのタイプに固有です。OAuth 2.0 または Open Policy Agent (OPA) を使用している場合は、特定のリスナーで承認サーバーまたは OPA サーバーへのアクセスを設定する必要もあります。特定の要件と環境に基づいてリスナーを作成できます。

リスナー設定の詳細は、Apache Kafka ドキュメント を参照してください。

ACL を使用してアクセスを微調整する

アクセス制御リスト (ACL) を使用して、Kafka クラスターへのアクセスを微調整できます。アクセス制御リスト (ACL) を作成および管理するには、**kafka-acls.sh** コマンドラインツールを使用します。ACL は、アクセスルールをクライアントアプリケーションに適用します。

次の例では、最初の ACL は、**my-topic** という名前の特定のトピックに対する読み取り権限と説明権限を付与します。**resource.patternType** は **literal** に設定されます。これは、リソース名が正確に一致する必要があることを意味します。

2 番目の ACL は、**my-group** という名前の特定のコンシューマーグループに読み取り権限を付与します。**resource.patternType** は **prefix** に設定されます。これは、リソース名が接頭辞と一致する必要があることを意味します。

ACL 設定の例

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=localhost:2181 --add \
--allow-principal User:my-user --operation Read --operation Describe --topic my-topic --resource-pattern-type literal \
--allow-principal User:my-user --operation Read --group my-group --resource-pattern-type prefixed
```

5.1.3. OpenShift 上で実行している Kafka クラスターへのセキュアな接続の確立

OpenShift で Streams for Apache Kafka を使用する場合、Kafka クラスターへのクライアント接続を保護するための一般的な概要は次のとおりです。

1. Cluster Operator を使用して、OpenShift 環境に Kafka クラスターをデプロイします。**Kafka** カスタムリソースを使用して、クラスターを設定およびインストールし、リスナーを作成します。
    - TLS や SASL SCRAM-SHA-512 などのリスナーの認証を設定します。Cluster Operator は、Kafka ブローカーの ID を検証するためのクラスター CA 証明書を含むシークレットを作成します。
    - **simple** 承認など、有効なすべてのリスナーの承認を設定します。
2. User Operator を使用して、クライアントを表す Kafka ユーザーを作成します。**KafkaUser** カスタムリソースを使用して、ユーザーを設定および作成します。
    - リスナーの認証メカニズムと一致する Kafka ユーザー (クライアント) の認証を設定します。User Operator は、クライアントが Kafka クラスターでの認証に使用するクライアント証明書と秘密鍵を含むシークレットを作成します。
    - リスナーの承認メカニズムと一致する Kafka ユーザー (クライアント) の承認を設定します。承認ルールにより、Kafka クラスターでの特定の操作が許可されます。
3. **config.properties** ファイルを作成して、クライアントアプリケーションがクラスターに接続するために必要な接続の詳細と認証情報を指定します。
4. Kafka クライアントアプリケーションを起動し、Kafka クラスターに接続します。
    - **config.properties** ファイルに定義されたプロパティーを使用して、Kafka ブローカーに接続します。
5. クライアントが Kafka クラスターに正常に接続し、メッセージをセキュアに消費および生成できることを確認します。

関連情報

ブローカーの設定の詳細は、OpenShift での Streams for Apache Kafka の設定 を参照し てください。

5.1.4. OpenShift 上の Kafka クラスターのセキュアなリスナーの設定

Streams for Apache **Kafka** を使用して Kafka カスタムリソースをデプロイする場合は、リスナー設定を Kafka **spec** に追加します。リスナー設定を使用して、Kafka での接続を保護します。Kafka ブローカーのセキュアな接続を設定するには、TLS、SASL、およびその他のセキュリティー関連の設定に関連するプロパティーをリスナーレベルで設定します。

外部リスナーは、OpenShift クラスターの外部から Kafka クラスターへのクライアントアクセスを提供します。Streams for Apache Kafka は、設定に基づいて Kafka クラスターへのアクセスを可能にするリスナーサービスとブートストラップアドレスを作成します。たとえば、次の接続メカニズムを使用する外部リスナーを作成できます。

- ノードポート
- loadbalancers
- OpenShift ルート

Kafka リソースの **nodeport** リスナーの設定例を次に示します。

Kafka リソースのリスナー設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: plaintext
        port: 9092
        type: internal
        tls: false
        configuration:
          useServiceDnsDomain: true
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external
        port: 9094
        type: route
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
      superUsers:
        - CN=superuser
  # ...
```

listeners プロパティーは、**plaintext**、**tls**、および **external** の 3 つのリスナーで設定されます。**external** リスナーのタイプは **nodeport** で、暗号化と認証の両方に TLS を使用します。Cluster Operator を使用して Kafka クラスターを作成すると、CA 証明書が自動的に生成されます。クラスター CA をクライアントアプリケーションのトラストストアに追加して、Kafka ブローカーの ID を確認します。または、ブローカーまたはリスナーレベルで独自の証明書を使用するように Streams for Apache Kafka を設定できます。クライアントアプリケーションが異なるセキュリティー設定を必要とする場合、リスナーレベルでの証明書の使用が必要になる場合があります。リスナーレベルで証明書を使用すると、制御とセキュリティーの層が追加されます。

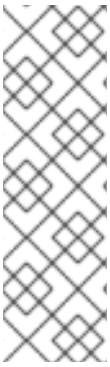
ヒント

設定プロバイダープラグインを使用して、設定データを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にロードします。設定プロバイダープラグインは、シークレットまたは ConfigMap から設定データを読み込みます。たとえば、Strimzi シークレットから証明書を自動的に取得するようにプロバイダーに指示できます。詳細は、OpenShift での実行に関する Streams for Apache Kafka ドキュメント を参照してください。

Kafka クラスターは単純な承認を使用します。承認プロパティーのタイプは **simple** に設定されます。すべてのリスナーに対する無制限のアクセスのために、単一のスーパーユーザーが定義されています。Streams for Apache Kafka は、Kafka **SimpleAclAuthorizer** およびカスタムオーソライザープラグインをサポートします。

これは単なる設定例です。一部の設定オプションはリスナーのタイプに固有です。OAuth 2.0 または Open Policy Agent (OPA) を使用している場合は、特定のリスナーで承認サーバーまたは OPA サーバーへのアクセスを設定する必要もあります。特定の要件と環境に基づいてリスナーを作成できます。

リスナー設定の詳細は、**GenericKafkaListener** スキーマ参照 を参照 してください。



注記

OpenShift 上の Kafka クラスターへのクライアントアクセスに **route** タイプリスナーを使用する場合は、TLS パススルー機能が有効になります。OpenShift ルートは HTTP プロトコルで動作するように設計されていますが、Apache Kafka で使用される Kafka プロトコルなど、他のプロトコルのネットワークトラフィックをプロキシーするために使用することもできます。クライアントはルートへの接続を確立し、ルートは TLS Server Name Indication (SNI) 拡張機能を使用して OpenShift クラスター内で実行されているブローカーにトラフィックを転送し、ターゲットのホスト名を取得します。SNI 拡張により、ルートは各接続のターゲットブローカーを正しく識別できるようになります。

ACL を使用してアクセスを微調整する

アクセス制御リスト (ACL) を使用して、Kafka クラスターへのアクセスを微調整できます。アクセス制御リスト (ACL) を追加するには、**KafkaUser** カスタムリソースを設定します。**KafkaUser** を作成すると、Streams for Apache Kafka が自動的に作成を管理し、ACL を更新します。ACL は、アクセスルールをクライアントアプリケーションに適用します。

次の例では、最初の ACL は、**my-topic** という名前の特定のトピックに対する読み取り権限と説明権限を付与します。**resource.patternType** は **literal** に設定されます。これは、リソース名が正確に一致する必要があることを意味します。

2 番目の ACL は、**my-group** という名前の特定のコンシューマーグループに読み取り権限を付与します。**resource.patternType** は **prefix** に設定されます。これは、リソース名が接頭辞と一致する必要があることを意味します。

KafkaUser リソースの ACL 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Read
          - Describe
      - resource:
          type: group
          name: my-group
```

```
    patternType: prefix
  operations:
    - Read
```

注記

Kafka ユーザーを設定するときに認証オプションとして **tls-external** を指定すると、User Operator によって生成されたクライアント証明書ではなく、独自のクライアント証明書を使用できます。

5.2. セキュアなアクセスのためのクライアントのセットアップ

セキュアな接続をサポートするために Kafka ブローカー上にリスナーを設定したら、次のステップは、これらのリスナーを使用して Kafka クラスターと通信するようにクライアントアプリケーションを設定することです。これには、リスナーに設定されたセキュリティーメカニズムに基づいて、クラスターで認証するための適切なセキュリティー設定を各クライアントに提供することが含まれます。

5.2.1. セキュリティープロトコルの設定

クライアントアプリケーションで使用されるセキュリティープロトコルを Kafka ブローカーリスナーで設定されたプロトコルと一致するように設定します。たとえば、TLS 認証には **SSL** (Secure Sockets Layer) を使用し、TLS 暗号化を使用した SASL (Simple Authentication and Security Layer over SSL) 認証には **SASL_SSL** を使用します。Kafka クラスターへのアクセスに必要な認証メカニズムをサポートするトラストストアとキーストアをクライアント設定に追加します。

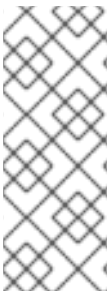
Truststore

トラストストアには、Kafka ブローカーの信頼性を検証するために使用される、信頼された認証局 (CA) のパブリック証明書が含まれています。クライアントがセキュアな Kafka ブローカーに接続するとき、ブローカーの ID を確認する必要がある場合があります。

キーストア

キーストアには、クライアントの秘密鍵とその公開証明書が含まれています。クライアントがブローカーに対して自身を認証したい場合、独自の証明書を提示します。

TLS 認証を使用している場合、Kafka クライアント設定には、Kafka クラスターに接続するためのトラストストアとキーストアが必要です。SASL SCRAM-SHA-512 を使用している場合、認証はデジタル証明書ではなくユーザー名とパスワードの認証情報の交換によって実行されるため、キーストアは必要ありません。SCRAM-SHA-512 はより軽量なメカニズムですが、証明書ベースの認証を使用するほどセキュアではありません。



注記

独自の証明書インフラストラクチャーがあり、サードパーティー CA からの証明書を使用している場合は、クライアントのデフォルトのトラストストアにパブリック CA 証明書がすでに含まれている可能性が高いため、それらをクライアントのトラストストアに追加する必要はありません。サーバーの証明書がデフォルトのトラストストアにすでに含まれているパブリック CA 証明書のいずれかによって署名されている場合、クライアントは自動的にその証明書を信頼します。

config.properties ファイルを作成して、クライアントアプリケーションで使用される認証情報を指定できます。

次の例では、**security.protocol** が **SSL** に設定され、クライアントとブローカーの間で TLS 認証と暗号化が有効になります。

ssl.truststore.location プロパティーと **ssl.truststore.password** プロパティーは、トラストストアの場所とパスワードを指定します。**ssl.keystore.location** プロパティーと **ssl.keystore.password** プロパティーは、キーストアの場所とパスワードを指定します。

PKCS #12 (公開鍵暗号化標準 #12) ファイル形式が使用されます。Base64 でエンコードされた PEM (Privacy Enhanced Mail) 形式を使用することもできます。

TLS 認証のクライアント設定プロパティーの例

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SSL
ssl.truststore.location = /path/to/ca.p12
ssl.truststore.password = truststore-password
ssl.keystore.location = /path/to/user.p12
ssl.keystore.password = keystore-password
client.id = my-client
```

次の例では、**security.protocol** が **SASL_SSL** に設定され、クライアントとブローカーの間で TLS 暗号化を使用した SASL 認証が有効になります。暗号化ではなく認証のみが必要な場合は、**SASL** プロトコルを使用できます。認証用に指定された SASL メカニズムは **SCRAM-SHA-512** です。さまざまな認証メカニズムを使用できます。**sasl.jaas.config** プロパティーは認証情報を指定します。

SCRAM-SHA-512 認証のクライアント設定プロパティーの例

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = SCRAM-SHA-512
sasl.jaas.config = org.apache.kafka.common.security.scram.ScramLoginModule required \
  username = "user" \
  password = "secret";
ssl.truststore.location = path/to/truststore.p12
ssl.truststore.password = truststore_password
ssl.truststore.type = PKCS12
client.id = my-client
```

注記

PEM 形式をサポートしていないアプリケーションの場合は、OpenSSL などのツールを使用して PEM ファイルを PKCS #12 形式に変換できます。

5.2.2. 許可される TLS バージョンと暗号スイートの設定

SSL 設定と暗号スイートを組み込んで、クライアントアプリケーションと Kafka クラスターの間の TLS ベースの通信をさらに保護できます。Kafka ブローカーの設定でサポートされている TLS バージョンと暗号スイートを指定します。クライアントが使用する TLS バージョンと暗号スイートを制限する場合は、クライアントに設定を追加することもできます。クライアントの設定では、ブローカーで有効になっているプロトコルと暗号スイートのみを使用する必要があります。

次の例では、Kafka ブローカーとクライアントアプリケーションの間の通信に **security.protocol** を使用して SSL が有効になっています。暗号スイートはコンマ区切りのリストとして指定します。**ssl.cipher.suites property** はクライアントが使用を許可されている暗号スイートのコンマ区切りのリストです。

Kafka ブローカーの SSL 設定プロパティーの例

```
security.protocol: "SSL"
ssl.enabled.protocols: "TLSv1.3", "TLSv1.2"
ssl.protocol: "TLSv1.3"
ssl.cipher.suites: "TLS_AES_256_GCM_SHA384"
```

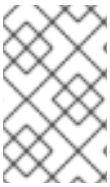
ssl.enabled.protocols プロパティーは、クラスターとそのクライアントの間のセキュアな通信に使用できる TLS バージョンを指定します。この場合は、**TLSv1.3** と **TLSv1.2** の両方が有効になります。**ssl.protocol** プロパティーは、全接続のデフォルトの TLS バージョンを設定します。有効なプロトコルから選択する必要があります。デフォルトでは、クライアントは **TLSv1.3** を使用して通信します。クライアントが TLSv1.2 のみをサポートしている場合でも、サポートされているバージョンを使用してブローカーに接続し、通信できます。同様に、設定がクライアント上にあり、ブローカーが TLSv1.2 のみをサポートする場合、クライアントはサポートされているバージョンを使用します。

Apache Kafka でサポートされる暗号スイートは、使用している Kafka のバージョンと基盤となる環境によって異なります。最高レベルのセキュリティーを提供する、サポートされている最新の暗号スイートを確認してください。

5.2.3. アクセス制御リスト (ACL) の使用

クライアントアプリケーションで ACLS を明示的に設定する必要はありません。ACL は、Kafka ブローカーによってサーバー側で適用されます。クライアントがデータを生成または消費するリクエストをサーバーに送信すると、サーバーは ACL をチェックして、クライアント (ユーザー) がリクエストされた操作を実行する権限を持っているかどうかを判断します。クライアントが承認されていると、リクエストは処理されます。それ以外の場合、リクエストは拒否され、エラーが返されます。ただし、Kafka クラスターとのセキュアな接続を可能にするために、クライアントは引き続き認証され、適切なセキュリティープロトコルを使用する必要があります。

Kafka ブローカーでアクセス制御リスト (ACL) を使用している場合は、制御するトピックおよび操作へのクライアントアクセスを制限するように ACL が適切に設定されていることを確認してください。Open Policy Agent (OPA) ポリシーを使用してアクセスを管理している場合、承認ルールがポリシー内で設定されているため、Kafka ブローカーに対して ACL を指定する必要はありません。OAuth 2.0 にはある程度の柔軟性が備わっています。OAuth 2.0 プロバイダーを使用して ACL を管理できます。または、OAuth 2.0 と Kafka の **simple** 承認を使用して ACL を管理します。



注記

ACL はほとんどのタイプのリクエストに適用され、操作の生成と消費に限定されません。たとえば、ACLS は、トピックの説明などの読み取り操作、新しいトピックの作成などの書き込み操作に適用できます。

5.2.4. トークンベースのアクセスに OAuth 2.0 を使用する

Streams for Apache Kafka での承認に OAuth 2.0 オープン標準を使用して、OAuth 2.0 プロバイダーを通じて承認制御を適用します。OAuth 2.0 は、アプリケーションが他のシステムに保存されているユーザーデータにアクセスするためのセキュアな方法を提供します。承認サーバーは、Kafka クラスターへのアクセスを許可するアクセストークンをクライアントアプリケーションに発行できます。

次の手順では、トークン検証に OAuth 2.0 を設定して使用するための一般的なアプローチを説明します。

1. クライアント ID やシークレットなど、ブローカーとクライアントの認証情報を使用して認可サーバーを設定します。

2. 認可サーバーから OAuth 2.0 認証情報を取得します。

3. OAuth 2.0 認証情報を使用して Kafka ブローカー上のリスナーを設定し、認可サーバーとやりとりします。

4. Oauth 2.0 の依存関係をクライアントライブラリーに追加します。

5. OAuth 2.0 認証情報を使用して Kafka クライアントを設定し、認可サーバーとやりとりします。

6. 実行時にアクセストークンを取得し、OAuth 2.0 プロバイダーでクライアントを認証します。

Kafka ブローカー上で OAuth 2.0 用に設定されたリスナーがある場合は、OAuth 2.0 を使用するようにクライアントアプリケーションをセットアップできます。Kafka クラスターにアクセスするための標準の Kafka クライアント設定に加えて、OAuth 2.0 認証用の特定の設定を含める必要があります。また、使用している承認サーバーが Kafka クラスターおよびクライアントアプリケーションからアクセス可能であることを確認する必要があります。

SASL (Simple Authentication and Security Layer) セキュリティープロトコルとメカニズムを指定します。実稼働環境では、次の設定が推奨されます。

- TLS 暗号化接続用の **SASL_SSL** プロトコル。
- ベアラートークンを使用した認証情報交換のための **OAUTHBEARER** メカニズム

JAAS (Java Authentication and Authorization Service) モジュールは SASL メカニズムを実装します。メカニズムの設定は、使用している認証方法によって異なります。たとえば、認証情報交換を使用して、OAuth 2.0 アクセストークンエンドポイント、アクセストークン、クライアント ID、およびクライアントシークレットを追加します。クライアントは承認サーバーのトークンエンドポイント (URL) に接続して、トークンがまだ有効かどうかを確認します。認証されたアクセスのための認可サーバーの公開鍵証明書を含むトラストストアも必要です。

OAauth 2.0 のクライアント設定プロパティーの例

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = OAUTHBEARER
# ...
sasl.jaas.config = org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required
\
    oauth.token.endpoint.uri = "https://localhost:9443/oauth2/token" \
    oauth.access.token = <access_token> \
    oauth.client.id = "<client_id>" \
    oauth.client.secret = "<client_secret>" \
    oauth.ssl.truststore.location = "/<truststore_location>/oauth-truststore.p12" \
    oauth.ssl.truststore.password = "<truststore_password>" \
    oauth.ssl.truststore.type = "PKCS12" \
```

関連情報

OAuth 2.0 を使用するようにブローカーを設定する方法の詳細は、次のガイドを参照してください。

- OpenShift での Apache Kafka のデプロイおよびアップグレード
- KRaft モードの RHEL での Streams for Apache Kafka の使用
- ZooKeeper で RHEL での Streams for Apache Kafka の使用

5.2.5. Open Policy Agent (OPA) アクセスポリシーの使用

Streams for Apache Kafka で Open Policy Agent (OPA)ポリシーエージェントを使用して、アクセスポリシーに対して Kafka クラスターに接続するためのリクエストを評価します。Open Policy Agent (OPA) は、認可ポリシーを管理するポリシーエンジンです。ポリシーはアクセス制御を一元化し、クライアントアプリケーションを変更することなく動的に更新できます。たとえば、特定のユーザー (クライアント) のみに特定のトピックへのメッセージの作成と消費を許可するポリシーを作成できます。

Streams for Apache Kafka は、オーソライザーとして Kafka 承認に Open Policy Agent プラグインを使用します。

次の手順では、OPA を設定して使用するための一般的なアプローチを説明します。

1. OPA サーバーのインスタンスをセットアップします。
2. Kafka クラスターへのアクセスを制御する承認ルールを提供するポリシーを定義します。
3. Kafka ブローカーが OPA 承認を受け入れ、OPA サーバーとやりとりするための設定を作成します。
4. Kafka クラスターへの承認されたアクセスのための認証情報を提供するように Kafka クライアントを設定します。

Kafka ブローカー上で OPA 用にリスナーが設定されている場合は、OPA を使用するようにクライアントアプリケーションをセットアップできます。リスナー設定では、OPA サーバーに接続し、クライアントアプリケーションを承認するための URL を指定します。Kafka クラスターにアクセスするための標準の Kafka クライアント設定に加えて、Kafka ブローカーで認証するための認証情報を追加する必要があります。ブローカーは、OPA サーバーにリクエストを送信して認可ポリシーを評価することにより、クライアントがリクエストされた操作を実行するために必要な承認を持っているかどうかを確認します。ポリシーエンジンが認可ポリシーを強制するため、通信を保護するためにトラストストアやキーストアは必要ありません。

OPA 承認のクライアント設定プロパティーの例

```
bootstrap.servers = my-cluster-kafka-bootstrap:9093
security.protocol = SASL_SSL
sasl.mechanism = SCRAM-SHA-512
sasl.jaas.config = org.apache.kafka.common.security.scram.ScramLoginModule required \
  username = "user" \
  password = "secret";
# ...
```

注記

Red Hat は OPA サーバーをサポートしません。

関連情報

OPA を使用するようにブローカーを設定する方法の詳細は、次のガイドを参照してください。

- OpenShift での Apache Kafka のデプロイおよびアップグレード
- KRaft モードの RHEL での Streams for Apache Kafka の使用
- ZooKeeper で RHEL での Streams for Apache Kafka の使用

5.2.6. メッセージのストリーミング時にトランザクションを使用する

ブローカーおよびプロデューサークライアントアプリケーションでトランザクションプロパティーを設定することにより、メッセージが単一のトランザクションで処理されるようにすることができます。トランザクションにより、メッセージのストリーミングに信頼性と一貫性が追加されます。

トランザクションはブローカー上で常に有効になります。次のプロパティーを使用してデフォルト設定を変更できます。

トランザクションの Kafka ブローカー設定プロパティーの例

```
transaction.state.log.replication.factor = 3
transaction.state.log.min.isr = 2
transaction.abort.timed.out.transaction.cleanup.interval.ms = 3600000
```

これは運用環境の一般的な設定であり、内部 **__transaction_state** トピック用に 3 つのレプリカを作成します。**__transaction_state** トピックには、進行中のトランザクションに関する情報が保存されます。トランザクションログには、少なくとも 2 つの同期レプリカが必要です。クリーンアップ間隔は、タイムアウトしたトランザクションをチェックし、対応するトランザクションログをクリーンアップするまでの時間です。

クライアント設定にトランザクションプロパティーを追加するには、プロデューサーとコンシューマーに対して次のプロパティーを設定します。

トランザクションのプロデューサークライアント設定プロパティーの例

```
transactional.id = unique-transactional-id
enable.idempotence = true
max.in.flight.requests.per.connection = 5
acks = all
retries=2147483647
transaction.timeout.ms = 30000
delivery.timeout = 25000
```

トランザクション ID により、Kafka ブローカーはトランザクションを追跡できるようになります。これはプロデューサーの一意の識別子であり、特定のパーティションのセットで使用する必要があります。複数のパーティションセットに対してトランザクションを実行する必要がある場合は、セットごとに異なるトランザクション ID を使用する必要があります。冪等性は、プロデューサーインスタンスが重複したメッセージを作成するのを避けるために有効になっています。冪等性では、メッセージはプロデューサー ID とシーケンス番号を使用して追跡されます。ブローカーはメッセージを受信すると、プロデューサー ID とシーケンス番号をチェックします。同じプロデューサー ID とシーケンス番号を持つメッセージがすでに受信されている場合、ブローカーは重複したメッセージを破棄します。

トランザクションが送信された順序で処理されるように、実行中のリクエストの最大数は 5 に設定されています。パーティションには、メッセージの順序を損なうことなく、最大 5 つの実行中のリクエストを含めることができます。

acks を **all** に設定すると、プロデューサーは、トランザクションが完了したと見なす前に、書き込み先のトピックパーティションのすべての同期レプリカからの確認応答を待機します。これにより、メッセージが Kafka クラスターに永続的に書き込まれ (コミットされ)、ブローカーに障害が発生した場合でもメッセージが失われることがなくなります。

トランザクションタイムアウトは、クライアントがタイムアウトになるまでにトランザクションを完了する必要がある最大時間を指定します。配信タイムアウトは、タイムアウトになるまでにプロデューサーがブローカーによるメッセージ配信の確認応答を待機する最大時間を指定します。メッセージがト

ランザクション期間内に確実に配信されるようにするには、配信タイムアウトをトランザクションタイムアウトよりも小さく設定します。失敗したメッセージ要求の再送信 **試行回数** を指定する場合は、ネットワーク遅延とメッセージスループットを考慮し、一時的な障害も視野に入れます。

トランザクションのコンシューマークライアント設定プロパティーの例

```
group.id = my-group-id
isolation.level = read_committed
enable.auto.commit = false
```

read_committed 分離レベルは、コンシューマーが正常に完了したトランザクションのメッセージのみを読み取ることを指定します。コンシューマーは、進行中のトランザクションまたは失敗したトランザクションの一部であるメッセージを処理しません。これにより、コンシューマーは完全に完了したトランザクションの一部であるメッセージのみを読み取ることが保証されます。

トランザクションを使用してメッセージをストリーミングする場合は、**enable.auto.commit** を **false** に設定することが重要です。**true** に設定すると、コンシューマーはトランザクションを考慮せずにオフセットを定期的にコミットします。これは、トランザクションが完全に完了する前にコンシューマーがメッセージをコミットする可能性があることを意味します。**enable.auto.commit** を **false** に設定すると、コンシューマーは、完全に書き込まれ、トランザクションの一部としてトピックにコミットされたメッセージのみを読み取り、コミットします。

第6章 KAFKA クライアントの開発

任意のプログラミング言語で Kafka クライアントを作成し、Streams for Apache Kafka に接続します。

Kafka クラスターとやりとりするには、クライアントアプリケーションがメッセージを生成および消費できる必要があります。基本的な Kafka クライアントアプリケーションを開発して設定するには、少なくとも次のことを行う必要があります。

- Kafka クラスターに接続するための設定をセットアップする
- プロデューサーとコンシューマーを使用してメッセージを送受信する

Kafka クラスターに接続し、プロデューサーとコンシューマーを使用するための基本設定をセットアップすることは、Kafka クライアント開発の最初のステップです。その後、入力、セキュリティー、パフォーマンス、エラー処理、クライアントアプリケーションの機能の改善に拡張できます。

前提条件

以下のプロパティー値を含むクライアントプロパティーファイルが作成されました。

- Kafka クラスターに接続するための基本設定
- 接続を保護するための設定

手順

1. Java、Python、.NET などのプログラミング言語用の Kafka クライアントライブラリーを選択します。Streams for Apache Kafka では、Red Hat によって構築されたクライアントライブラリーのみがサポートされます。現在、Streams for Apache Kafka は Java クライアントライブラリーのみを提供します。
2. パッケージマネージャーを使用するか、ソースからライブラリーをダウンロードして手動でライブラリーをインストールします。
3. Kafka クライアントに必要なクラスと依存関係をコードにインポートします。
4. 作成するクライアントのタイプに応じて、Kafka コンシューマーオブジェクトまたはプロデューサーオブジェクトを作成します。
   クライアントは、Kafka コンシューマー、プロデューサ、Streams プロセッサー、および管理者のいずれかになります。
5. Kafka クラスターに接続するための設定プロパティー (必要に応じてブローカーアドレス、ポート、認証情報など) を指定します。
   ローカルの Kafka デプロイメントの場合は、**localhost:9092** のようなアドレスから始めることができます。ただし、Streams for Apache Kafka によって管理される Kafka クラスターを使用する場合は、**oc** コマンドを使用して、**Kafka** カスタムリソースのステータスからブートストラップアドレスを取得できます。

   ```
   oc get kafka <kafka_cluster_name> -o=jsonpath='{.status.listeners[*].bootstrapServers}{"\n"}'
   ```

   このコマンドは、Kafka クラスター上のクライアント接続のリスナーによって公開されたブートストラップアドレスを取得します。
6. Kafka コンシューマーまたはプロデューサーオブジェクトを使用して、トピックのサブスクライブ、メッセージの生成、または Kafka クラスターからのメッセージの取得を行います。

7. エラー処理に注意してください。これは、Kafka に接続して通信する場合、特に高可用性と操作の容易さが重視される運用システムでは非常に重要です。効果的なエラー処理は、プロトタイプと実稼働レベルのアプリケーションを区別する重要な要素であり、Kafka だけでなく、あらゆる堅牢なソフトウェアシステムにも該当します。

6.1. KAFKA プロデューサーアプリケーションの例

この Java ベースの Kafka プロデューサーアプリケーションは、Kafka トピックへのメッセージを生成する自己完結型アプリケーションの例です。クライアントは Kafka **Producer** API を使用して、いくつかのエラー処理を行いながらメッセージを非同期に送信します。

クライアントは、メッセージ処理用の **Callback** インターフェイスを実装します。

Kafka プロデューサーアプリケーションを実行するには、**Producer** クラスの **main** メソッドを実行します。クライアントは、**randomBytes** メソッドを使用して、メッセージペイロードとしてランダムなバイト配列を生成します。クライアントは、**NUM_MESSAGES** メッセージ (設定例では 50) が送信されるまで、指定された Kafka トピックにメッセージを生成します。プロデューサはスレッドセーフであるため、複数のスレッドが単一のプロデューサインスタンスを使用できます。

Kafka プロデューサーインスタンスはスレッドセーフになるように設計されており、複数のスレッドが 1 つのプロデューサーインスタンスを共有できます。

このサンプルクライアントは、特定のユースケース向けに、より複雑な Kafka プロデューサを構築するための基本基盤を提供します。セキュアな接続の実装 など、追加の機能を組み込むことができます。

前提条件

- 指定された **BOOTSTRAP_SERVERS** で実行されている Kafka ブローカー
- メッセージが生成される **TOPIC_NAME** という名前の Kafka トピック。
- クライアント依存関係

Kafka プロデューサーアプリケーションを実装する前に、プロジェクトに必要な依存関係を含める必要があります。Java ベースの Kafka クライアントの場合は、Kafka クライアント JAR が含まれます。この JAR ファイルには、クライアントの構築と実行に必要な Kafka ライブラリーが含まれています。

Maven プロジェクトの **pom.xml** ファイルに依存関係を追加する方法は、「Kafka クライアントの依存関係を Maven プロジェクトに追加する」 を参照してください。

Configuration

プロデューサー クラスで指定された次の定数を使用して、プロデューサーアプリケーションを設定できます。

BOOTSTRAP_SERVERS
: Kafka ブローカーに接続するためのアドレスおよびポート。

TOPIC_NAME
: メッセージを生成する Kafka トピックの名前。

NUM_MESSAGES
: 停止する前に生成するメッセージの数。

MESSAGE_SIZE_BYTES
: 各メッセージのバイト単位のサイズ。

PROCESSING_DELAY_MS

メッセージ送信間の遅延 (ミリ秒単位)。これにより、メッセージの処理時間をシミュレートでき、テストに役立ちます。

プロデューサーアプリケーションの例

```
import java.util.Properties;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.errors.RetriableException;
import org.apache.kafka.common.serialization.ByteArraySerializer;
import org.apache.kafka.common.serialization.LongSerializer;

public class Producer implements Callback {
    private static final Random RND = new Random(0);
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String TOPIC_NAME = "my-topic";
    private static final long NUM_MESSAGES = 50;
    private static final int MESSAGE_SIZE_BYTES = 100;
    private static final long PROCESSING_DELAY_MS = 1000L;

    protected AtomicLong messageCount = new AtomicLong(0);

    public static void main(String[] args) {
        new Producer().run();
    }

    public void run() {
        System.out.println("Running producer");
        try (var producer = createKafkaProducer()) { 1
            byte[] value = randomBytes(MESSAGE_SIZE_BYTES); 2
            while (messageCount.get() < NUM_MESSAGES) { 3
                sleep(PROCESSING_DELAY_MS); 4
                producer.send(new ProducerRecord<>(TOPIC_NAME, messageCount.get(), value), this);
5
                messageCount.incrementAndGet();
            }
        }
    }

    private KafkaProducer<Long, byte[]> createKafkaProducer() {
        Properties props = new Properties(); 6
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); 7
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "client-" + UUID.randomUUID()); 8
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class); 9
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class);
        return new KafkaProducer<>(props);
    }
```

```
    private void sleep(long ms) { 10
        try {
            TimeUnit.MILLISECONDS.sleep(ms);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    private byte[] randomBytes(int size) { 11
        if (size <= 0) {
            throw new IllegalArgumentException("Record size must be greater than zero");
        }
        byte[] payload = new byte[size];
        for (int i = 0; i < payload.length; ++i) {
            payload[i] = (byte) (RND.nextInt(26) + 65);
        }
        return payload;
    }

    private boolean retriable(Exception e) { 12
        if (e instanceof IllegalArgumentException
            || e instanceof UnsupportedOperationException
            || !(e instanceof RetriableException)) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception e) { 13
        if (e != null) {
            System.err.println(e.getMessage());
            if (!retriable(e)) {
                e.printStackTrace();
                System.exit(1);
            }
        } else {
            System.out.printf("Record sent to %s-%d with offset %d%n",
                metadata.topic(), metadata.partition(), metadata.offset());
        }
    }
}
```

1. クライアントは、**createKafkaProducer** メソッドを使用して Kafka プロデューサを作成します。プロデューサは、Kafka トピックにメッセージを非同期的に送信します。
2. バイト配列は、Kafka トピックに送信される各メッセージのペイロードとして使用されます。
3. 送信されるメッセージの最大数は、**NUM_MESSAGES** 定数値によって決まります。
4. メッセージレートは、送信される各メッセージ間の遅延によって制御されます。
5. プロデューサは、トピック名、メッセージカウント値、およびメッセージ値を渡します。

6 クライアントは、提供された設定を使用して **KafkaProducer** インスタンスを作成します。プロパティーファイルを使用することも、設定を直接追加することもできます。基本設定の詳細について

7 Kafka ブローカーへの接続。

8 ランダムに生成された UUID を使用するプロデューサーの一意のクライアント ID。クライアント ID は必須ではありませんが、リクエストのソースを追跡するのに役立ちます。

9 キーおよび値をバイト配列として処理するための適切なシリアライザークラス。

10 指定されたミリ秒数の間、メッセージ送信プロセスに遅延を導入するメソッド。メッセージの送信を担当するスレッドが一時停止中に中断されると、**InterruptedException** エラーがスローされます。

11 Kafka トピックに送信される各メッセージのペイロードとして機能する、特定のサイズのランダムなバイト配列を作成するメソッド。このメソッドはランダムな整数を生成し、ASCII コードの大文字を表す **65** を加算します (65 は **A**、66 は **B** など)。ASCII コードはペイロード配列に 1 バイトとして保存されます。ペイロードサイズがゼロ以下の場合、**IllegalArgumentException** がスローされます。

12 例外の後にメッセージの送信を再試行するかどうかを確認するメソッド。Kafka プロデューサは、接続エラーなどの特定のエラーに対する再試行を自動的に処理します。このメソッドをカスタマイズして、他のエラーを含めることができます。null および指定された例外、または **RetriableException** インターフェイスを実装していない例外の場合は **false** を返します。

13 Kafka ブローカーによってメッセージが確認されたときに呼び出されるメソッド。成功すると、トピック、パーティション、メッセージのオフセット位置の詳細を含むメッセージが出力されます。メッセージの送信時にエラーが発生した場合は、エラーメッセージが出力されます。このメソッドは例外をチェックし、それが致命的なエラーであるか致命的ではないエラーであるかに基づいて適切なアクションを実行します。エラーが致命的ではない場合、メッセージ送信プロセスは続行されます。エラーが致命的である場合、スタックトレースが出力され、プロデューサは終了します。

エラー処理

プロデューサーアプリケーションによって捕捉された致命的な例外:

InterruptedException

一時停止中に現在のスレッドが中断された場合にスローされるエラー。通常、中断はプロデューサーを停止またはシャットダウンするときに発生します。例外は **RuntimeException** として再スローされ、プロデューサーが終了します。

IllegalArgumentException

プロデューサが無効または不適切な引数を受け取ったときにスローされるエラー。たとえば、トピックが欠落している場合、例外がスローされます。

UnsupportedOperationException

操作がサポートされていない場合、またはメソッドが実装されていない場合にスローされるエラー。たとえば、サポートされていないプロデューサー設定を使用しようとしたり、**KafkaProducer** クラスでサポートされていないメソッドを呼び出そうとした場合、例外が出力されます。

プロデューサーアプリケーションによって捕捉された致命的ではない例外:

RetriableException

Kafka クライアントライブラリーによって提供される **RetriableException** インターフェイスを実装する例外に対してスローされるエラー。

致命的ではないエラーの場合、プロデューサーはメッセージの送信を続けます。

注記

デフォルトでは、Kafka は少なくとも 1 回のメッセージ配信セマンティクスで動作するので、特定のシナリオではメッセージが複数回配信され、重複が発生する可能性があります。このリスクを回避するには、Kafka プロデューサーでトランザクションを有効にすること を検討してください。トランザクションは、1 回限りの配信を強化します。さらに、**retries** 設定プロパティーを使用して、プロデューサーがメッセージの送信を中止するまでに再試行する回数を制御できます。この設定は、メッセージ送信エラー時に **retriable** メソッドが **true** を返す回数に影響します。

6.2. KAFKA コンシューマーアプリケーションの例

この Java ベースの Kafka コンシューマーアプリケーションは、Kafka トピックからのメッセージを消費する自己完結型アプリケーションの例です。クライアントは、Kafka **Consumer** API を使用して、指定されたトピックからメッセージを非同期に取得して処理し、いくつかのエラー処理を行います。メッセージが正常に処理された後にオフセットをコミットすることにより、少なくとも 1 回のセマンティクスに従います。

クライアントは、パーティション処理のための **ConsumerRebalanceListener** インターフェイスと、オフセットをコミットするための **OffsetCommitCallback** インターフェイスを実装します。

Kafka コンシューマーアプリケーションを実行するには、**Consumer** クラスの **main** メソッドを実行します。クライアントは、**NUM_MESSAGES** メッセージ (設定例では 50) が消費されるまで、Kafka トピックからのメッセージを消費します。コンシューマーは、複数のスレッドが同時に安全にアクセスできるように設計されていません。

このサンプルクライアントは、特定のユースケース向けに、より複雑な Kafka コンシューマーを構築するための基本基盤を提供します。セキュアな接続の実装 など、追加の機能を組み込むことができます。

前提条件

- 指定された **BOOTSTRAP_SERVERS** で実行されている Kafka ブローカー
- メッセージが消費される **TOPIC_NAME** という名前の Kafka トピック。
- クライアント依存関係

Kafka コンシューマーアプリケーションを実装する前に、プロジェクトに必要な依存関係を含める必要があります。Java ベースの Kafka クライアントの場合は、Kafka クライアント JAR が含まれます。この JAR ファイルには、クライアントの構築と実行に必要な Kafka ライブラリーが含まれています。

Maven プロジェクトの **pom.xml** ファイルに依存関係を追加する方法は、「Kafka クライアントの依存関係を Maven プロジェクトに追加する」 を参照してください。

Configuration

Consumer クラスで指定された次の定数を使用してコンシューマーアプリケーションを設定できます。

BOOTSTRAP_SERVERS
: Kafka ブローカーに接続するためのアドレスおよびポート。

GROUP_ID
: コンシューマーグループの識別子。

POLL_TIMEOUT_MS

各ポーリング中に新しいメッセージを待機する最大時間。

TOPIC_NAME

メッセージを消費する Kafka トピックの名前。

NUM_MESSAGES

停止する前に消費するメッセージの数。

PROCESSING_DELAY_MS

メッセージ送信間の遅延 (ミリ秒単位)。これにより、メッセージの処理時間をシミュレートでき、テストに役立ちます。

コンシューマーアプリケーションの例

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.NoOffsetForPartitionException;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.clients.consumer.OffsetCommitCallback;
import org.apache.kafka.clients.consumer.OffsetOutOfRangeException;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.errors.RebalanceInProgressException;
import org.apache.kafka.common.errors.RetriableException;
import org.apache.kafka.common.serialization.ByteArrayDeserializer;
import org.apache.kafka.common.serialization.LongDeserializer;

import static java.time.Duration.ofMillis;
import static java.util.Collections.singleton;

public class Consumer implements ConsumerRebalanceListener, OffsetCommitCallback {
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String GROUP_ID = "my-group";
    private static final long POLL_TIMEOUT_MS = 1_000L;
    private static final String TOPIC_NAME = "my-topic";
    private static final long NUM_MESSAGES = 50;
    private static final long PROCESSING_DELAY_MS = 1_000L;

    private KafkaConsumer<Long, byte[]> kafkaConsumer;
    protected AtomicLong messageCount = new AtomicLong(0);
    private Map<TopicPartition, OffsetAndMetadata> pendingOffsets = new HashMap<>();

    public static void main(String[] args) {
        new Consumer().run();
    }

    public void run() {
```

```
        System.out.println("Running consumer");
        try (var consumer = createKafkaConsumer()) { 1
            kafkaConsumer = consumer;
            consumer.subscribe(singleton(TOPIC_NAME), this); 2
            System.out.printf("Subscribed to %s%n", TOPIC_NAME);
            while (messageCount.get() < NUM_MESSAGES) { 3
                try {
                    ConsumerRecords<Long, byte[]> records =
consumer.poll(ofMillis(POLL_TIMEOUT_MS)); 4
                    if (!records.isEmpty()) { 5
                        for (ConsumerRecord<Long, byte[]> record : records) {
                            System.out.printf("Record fetched from %s-%d with offset %d%n",
                                record.topic(), record.partition(), record.offset());
                            sleep(PROCESSING_DELAY_MS); 6

                            pendingOffsets.put(new TopicPartition(record.topic(), record.partition()), 7
                                new OffsetAndMetadata(record.offset() + 1, null));
                            if (messageCount.incrementAndGet() == NUM_MESSAGES) {
                                break;
                            }
                        }
                        consumer.commitAsync(pendingOffsets, this); 8
                        pendingOffsets.clear();
                    }
                } catch (OffsetOutOfRangeException | NoOffsetForPartitionException e) { 9
                    System.out.println("Invalid or no offset found, and auto.reset.policy unset, using latest");
                    consumer.seekToEnd(e.partitions());
                    consumer.commitSync();
                } catch (Exception e) {
                    System.err.println(e.getMessage());
                    if (!retriable(e)) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
            }
        }
    }

    private KafkaConsumer<Long, byte[]> createKafkaConsumer() {
        Properties props = new Properties(); 10
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS); 11
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "client-" + UUID.randomUUID()); 12
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID); 13
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class);
14
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ByteArrayDeserializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); 15
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest"); 16
        return new KafkaConsumer<>(props);
    }

    private void sleep(long ms) { 17
```

```
            try {
                TimeUnit.MILLISECONDS.sleep(ms);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        private boolean retriable(Exception e) { 18
            if (e == null) {
                return false;
            } else if (e instanceof IllegalArgumentException
                || e instanceof UnsupportedOperationException
                || !(e instanceof RebalanceInProgressException)
                || !(e instanceof RetriableException)) {
                return false;
            } else {
                return true;
            }
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> partitions) { 19
            System.out.printf("Assigned partitions: %s%n", partitions);
        }

        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> partitions) { 20
            System.out.printf("Revoked partitions: %s%n", partitions);
            kafkaConsumer.commitSync(pendingOffsets);
            pendingOffsets.clear();
        }

        @Override
        public void onPartitionsLost(Collection<TopicPartition> partitions) { 21
            System.out.printf("Lost partitions: {}", partitions);
        }

        @Override
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> map, Exception e) { 22
            if (e != null) {
                System.err.println("Failed to commit offsets");
                if (!retriable(e)) {
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }
    }
```

1 クライアントは、**createKafkaConsumer** メソッドを使用して Kafka コンシューマーを作成します。

2 コンシューマーは特定のトピックをサブスクライブします。トピックをサブスクライブすると、確認メッセージが出力されます。

3 消費されるメッセージの最大数は、**NUM_MESSAGES** 定数値によって決まります。

4 メッセージを取得するための次のポーリングは、リバランスを避けるために **session.timeout.ms** 以内に呼び出す必要があります。

5 Kafka から取得したバッチメッセージを含む **records** オブジェクトが空でないことを確認する条件。**records** オブジェクトが空の場合、処理する新しいメッセージがないため、プロセスはスキップされます。

6 指定されたミリ秒数の間、メッセージ取得プロセスに遅延を導入するメソッド。

7 コンシューマーは、**pendingOffsets** マップを使用して、コミットする必要がある、消費されたメッセージのオフセットを保存します。

8 メッセージのバッチを処理した後、コンシューマーは **commitAsync** メソッドを使用してオフセットを非同期的にコミットし、少なくとも 1 回のセマンティクスを実装します。

9 メッセージを消費し、自動リセットポリシーが設定されていない場合に、致命的でないエラーと致命的なエラーを処理するための捕捉。致命的ではないエラーの場合、コンシューマーはパーティションの末尾までシークし、利用可能な最新のオフセットから消費を開始します。例外を再試行できない場合は、スタックトレースが出力され、コンシューマーは終了します。

10 クライアントは、提供された設定を使用して **KafkaConsumer** インスタンスを作成します。プロパティーファイルを使用することも、設定を直接追加することもできます。基本設定の詳細については、*4章Kafka クラスターに接続するためのクライアントアプリケーションの設定* を参照してください。

11 Kafka ブローカーへの接続。

12 ランダムに生成された UUID を使用するプロデューサーの一意のクライアント ID。クライアント ID は必須ではありませんが、リクエストのソースを追跡するのに役立ちます。

13 コンシューマーがパーティションへの割り当てを調整するためのグループ ID。

14 キーおよび値をバイト配列として処理するための適切なデシリアライザークラス。

15 自動オフセットコミットを無効にする設定。

16 パーティションに対してコミットされたオフセットが見つからない場合に、コンシューマーが最も古い利用可能なオフセットからメッセージの消費を開始するための設定。

17 メッセージを消費するプロセスに指定されたミリ秒数の遅延を導入するメソッド。メッセージの送信を担当するスレッドが一時停止中に中断されると、**InterruptedException** エラーがスローされます。

18 例外の後にメッセージのコミットを再試行するかどうかを確認するメソッド。Null 例外と指定された例外は再試行されません。また、**RebalanceInProgressException** または **RetriableException** インターフェイスを実装していない例外も再試行されません。このメソッドをカスタマイズして、他のエラーを含めることができます。

19 コンシューマーに割り当てられているパーティションのリストを示すメッセージをコンソールに出力するメソッド。

20 コンシューマーグループのリバランス中にコンシューマーがパーティションの所有権を失いそうなときに呼び出されるメソッド。このメソッドは、コンシューマーから取り消されるパーティションのリストを出力します。保留中のオフセットはすべてコミットされます。

21 コンシューマーグループのリバランス中にコンシューマーがパーティションの所有権を失ったが、保留中のオフセットをコミットできなかった場合に呼び出されるメソッド。このメソッドは、コン

22 コンシューマーがオフセットを Kafka にコミットするときに呼び出されるメソッド。オフセットのコミット時にエラーが発生した場合は、エラーメッセージが出力されます。このメソッドは例外をチェックし、それが致命的なエラーであるか致命的ではないエラーであるかに基づいて適切なアクションを実行します。エラーが致命的ではない場合、オフセットコミットプロセスは続行されます。エラーが致命的である場合、スタックトレースが出力され、コンシューマーは終了します。

エラー処理

コンシューマーアプリケーションによって捕捉された致命的な例外:

InterruptedException

一時停止中に現在のスレッドが中断された場合にスローされるエラー。中断は通常、コンシューマーを停止またはシャットダウンするときに発生します。例外は **RuntimeException** として再スローされ、コンシューマーを終了します。

IllegalArgumentException

コンシューマーが無効または不適切な引数を受け取ったときにスローされるエラー。たとえば、トピックが欠落している場合、例外がスローされます。

UnsupportedOperationException

操作がサポートされていない場合、またはメソッドが実装されていない場合にスローされるエラー。たとえば、サポートされていないコンシューマー設定を使用しようとしたり、**KafkaConsumer** クラスでサポートされていないメソッドを呼び出そうとした場合、例外がスローされます。

コンシューマーアプリケーションによって捕捉された致命的ではない例外:

OffsetOutOfRangeException

通常は、オフセットが対象のパーティションの有効なオフセット範囲外にある場合や、自動リセットポリシーが有効でない場合など、コンシューマーがパーティションに無効なオフセットがないか検索しようとするとエラーが出力されます。回復するには、コンシューマーはパーティションの最後まで検索してオフセットを同期的にコミットします (**commitSync**)。自動リセットポリシーが有効な場合、コンシューマーは設定に応じてパーティションの先頭または末尾を検索します。

NoOffsetForPartitionException

パーティションにコミットされたオフセットがない場合、または要求されたオフセットが無効で、自動リセットポリシーが有効になっていない場合に出力されるエラー。回復するには、コンシューマーはパーティションの最後まで検索してオフセットを同期的にコミットします (**commitSync**)。自動リセットポリシーが有効な場合、コンシューマーは設定に応じてパーティションの先頭または末尾を検索します。

RebalanceInProgressException

コンシューマーグループのリバランス中にパーティションが割り当てられているときにスローされるエラー。コンシューマーがリバランスを実行しているときは、オフセットコミットを完了できません。

RetriableException

Kafka クライアントライブラリーによって提供される **RetriableException** インターフェイスを実装する例外に対してスローされるエラー。

致命的ではないエラーの場合、コンシューマーはメッセージの処理を続行します。

6.3. コンシューマーとの協力的なリバランスの使用

Kafka コンシューマーは、適切なリバランスプロトコルによって決定されるパーティション割り当て戦略を使用します。デフォルトでは、Kafka は **RangeAssignor** プロトコルを採用します。これにより、コンシューマーはリバランス中にパーティション割り当てを放棄することになり、サービスが中断される可能性があります。

効率を向上させ、ダウンタイムを短縮するには、協調的なリバランスアプローチである **CooperativeStickyAssignor** プロトコルに切り替えることができます。デフォルトのプロトコルとは異なり、協調リバランスでは、コンシューマーが連携して作業することができ、リバランス中にパーティションの割り当てを保持し、コンシューマーグループ内のバランスを達成するために必要な場合にのみパーティションを解放します。

手順

1. コンシューマー設定では、**partition.assignment.strategy** プロパティーを使用して、プロトコルとして **CooperativeStickyAssignor** を使用するように切り替えます。たとえば、現在の設定が **partition.assignment.strategy=RangeAssignor, CooperativeStickyAssignor** の場合は、**partition.assignment.strategy=CooperativeStickyAssignor** に更新します。
   コンシューマー設定ファイルを直接変更する代わりに、コンシューマーアプリケーションコードで **props.put** を使用してパーティション割り当て戦略を設定することもできます。

```
# ...
props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
"org.apache.kafka.clients.consumer.CooperativeStickyAssignor");
# ...
```

2. グループ内の各コンシューマーを一度に 1 つずつ再起動し、再起動するたびにグループに再参加できるようにします。

警告

CooperativeStickyAssignor プロトコルに切り替えた後、コンシューマーのリバランス中に **RebalanceInProgressException** が発生し、同じコンシューマーグループ内の複数の Kafka クライアントが予期せず停止する可能性があります。さらに、この問題により、Kafka コンシューマーがリバランス中にパーティションの割り当てを変更していない場合でも、コミットされていないメッセージが重複する可能性があります。自動オフセットコミット (**enable.auto.commit=true**) を使用している場合は、何も変更する必要はありません。オフセットを手動でコミットしており (**enable.auto.commit=false**)、手動コミット中に **RebalanceInProgressException** が発生した場合は、コンシューマーの実装を変更して、次のループで **poll()** を呼び出してコンシューマーのリバランスプロセスを完了します。詳細は、カスタマーポータルの **CooperativeStickyAssignor** の記事を参照してください。

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