



Red Hat AMQ Streams 2.2

RHEL での AMQ Streams の使用

Red Hat Enterprise Linux での AMQ Streams 2.2 のデプロイメントの設定および管理

Red Hat AMQ Streams 2.2 RHEL での AMQ Streams の使用

Red Hat Enterprise Linux での AMQ Streams 2.2 のデプロイメントの設定および管理

Enter your first name here. Enter your surname here.

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

Enter your email address here.

法律上の通知

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

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

AMQ Streams 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

多様性を受け入れるオープンソースの強化	8
第1章 AMQ STREAMS の概要	9
1.1. KAFKA BRIDGE を使用した KAFKA クラスターへの接続	10
1.2. サポートされる設定	10
1.3. 本書の表記慣例	10
第2章 スタートガイド	12
2.1. AMQ STREAMS のディストリビューション	12
2.2. AMQ STREAMS アーカイブのダウンロード	12
2.3. AMQ STREAMS のインストール	12
2.4. データストレージに関する留意事項	13
2.4.1. ファイルシステム	13
2.4.2. Apache Kafka および ZooKeeper ストレージ	14
2.5. 単一ノードの KAFKA クラスターの実行	14
2.6. トピックからのメッセージの送受信	15
2.7. AMQ STREAMS サービスの停止	16
第3章 マルチノード環境の実行	18
3.1. マルチノードの ZOOKEEPER クラスターの実行	18
3.2. マルチノードの KAFKA クラスターの実行	19
3.3. KAFKA ブローカーの正常なローリング再起動の実行	21
第4章 KRAFT モードでの KAFKA の実行 (開発プレビュー)	25
4.1. KRAFT モードの KAFKA で AMQ STREAMS を使用する	25
4.2. KRAFT モードでの KAFKA クラスターの実行	26
第5章 RHEL デプロイメントでの AMQ STREAMS の設定	28
5.1. 標準の KAFKA 設定プロパティの使用	28
5.2. 環境変数から設定値の読み込み	29
5.3. ZOOKEEPER の設定	29
5.3.1. 基本設定	30
5.3.2. ZooKeeper クラスター設定	30
5.3.3. 認証	32
5.3.3.1. SASL を使用した認証	32
5.3.3.2. DIGEST-MD5 を使用したサーバー間の認証の有効化	34
5.3.3.3. DIGEST-MD5 を使用したクライアント/サーバー間の認証の有効化	35
5.3.4. 承認	36
5.3.5. TLS	36
5.3.6. その他の設定オプション	36
5.3.7. ログイン	37
5.4. KAFKA の設定	37
5.4.1. ZooKeeper	37
5.4.2. リスナー	37
5.4.3. ログのコミット	39
5.4.4. ブローカー ID	39
5.4.5. ZooKeeper の認証	40
5.4.5.1. JAAS 設定	40
5.4.5.2. ZooKeeper 認証の有効化	40
5.4.6. 承認	41
5.4.6.1. シンプルな ACL オーソライザー	41
5.4.6.1.1. ACL ルール	41

5.4.6.1.2. プリンシパル	41
5.4.6.1.3. ユーザーの認証	42
5.4.6.1.4. スーパーユーザー	42
5.4.6.1.5. レプリカブローカーの認証	42
5.4.6.1.6. サポートされるリソース	42
5.4.6.1.7. サポートされる操作	43
5.4.6.1.8. ACL 管理オプション	43
5.4.6.2. 承認の有効化	47
5.4.6.3. ACL ルールの追加	48
5.4.6.4. ACL ルールの一覧表示	49
5.4.6.5. ACL ルールの削除	49
5.4.7. ZooKeeper の承認	50
5.4.7.1. ACL 設定	50
5.4.7.2. 新しい Kafka クラスターでの ZooKeeper ACL の有効化	51
5.4.7.3. 既存の Kafka クラスターでの ZooKeeper ACL の有効化	51
5.4.8. 暗号化と認証	52
5.4.8.1. リスナーの設定	52
5.4.8.2. TLS 暗号化	53
5.4.8.3. TLS 暗号化の有効化	54
5.4.8.4. 認証	55
5.4.8.4.1. TLS クライアント認証	55
5.4.8.4.2. SASL 認証	55
5.4.8.5. TLS クライアント認証の有効化	58
5.4.8.6. SASL PLAIN 認証の有効化	59
5.4.8.7. SASL SCRAM 認証の有効化	60
5.4.8.8. SASL SCRAM ユーザーの追加	61
5.4.8.9. SASL SCRAM ユーザーの削除	61
5.4.9. OAuth 2.0 トークンベース認証の使用	61
5.4.9.1. OAuth 2.0 認証メカニズム	63
5.4.9.1.1. プロパティまたは変数を使用した OAuth 2.0 の設定	64
5.4.9.2. OAuth 2.0 Kafka ブローカーの設定	65
5.4.9.2.1. 承認サーバーの OAuth 2.0 クライアント設定	65
5.4.9.2.2. Kafka クラスターでの OAuth 2.0 認証設定	65
5.4.9.2.3. 高速なローカル JWT トークン検証の設定	70
5.4.9.2.4. OAuth 2.0 インtrospeクシオンエンドポイントの設定	71
5.4.9.3. Kafka ブローカーの再認証の設定	72
5.4.9.4. OAuth 2.0 Kafka クライアントの設定	73
5.4.9.5. OAuth 2.0 クライアント認証フロー	74
5.4.9.5.1. 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	74
5.4.9.5.2. 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	76
5.4.9.6. OAuth 2.0 認証の設定	78
5.4.9.6.1. OAuth 2.0 承認サーバーとしての Red Hat Single Sign-On の設定	78
5.4.9.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定	80
5.4.9.6.3. OAuth 2.0 を使用するための Kafka Java クライアントの設定	84
5.4.10. OAuth 2.0 トークンベース承認の使用	85
5.4.10.1. OAuth 2.0 の承認メカニズム	86
5.4.10.1.1. Kafka ブローカーのカスタムオーソライザー	86
5.4.10.2. OAuth 2.0 承認サポートの設定	86
5.4.11. OPA ポリシーベースの承認の使用	89
5.4.11.1. OPA ポリシーの定義	89
5.4.11.2. OPA への接続	90
5.4.11.3. OPA 承認サポートの設定	90
5.4.12. ログイン	91

5.4.12.1. Kafka ブローカーロガーのログingleベルの動的な変更	91
ブローカーロガーのリセット	92
第6章 トピックの作成および管理	94
6.1. パーティションおよびレプリカ	94
6.2. メッセージの保持	94
6.3. トピックの自動作成	95
6.4. トピックの削除	95
6.5. トピックの設定	95
6.6. 内部トピック	96
6.7. トピックの作成	97
6.8. トピックの一覧表示および説明	98
6.9. トピック設定の変更	98
6.10. トピックの削除	100
第7章 KAFKA CONNECT での AMQ STREAMS の使用	101
7.1. スタンドアロンモードでの KAFKA CONNECT	101
7.1.1. スタンドアロンモードでの Kafka Connect の設定	101
7.1.2. スタンドアロンモードでの Kafka Connect でのコネクタの設定	102
7.1.3. スタンドアロンモードでの Kafka Connect の実行	102
7.2. 分散モードでの KAFKA CONNECT	103
7.2.1. 分散モードでの Kafka Connect の設定	103
7.2.2. 分散 Kafka Connect でのコネクタの設定	104
7.2.3. 分散 Kafka Connect の実行	105
7.2.4. コネクタの作成	106
7.2.5. コネクタの削除	106
7.3. コネクタプラグイン	107
7.4. コネクタプラグインの追加	107
第8章 AMQ STREAMS の MIRRORMAKER 2.0 との使用	109
8.1. MIRRORMAKER 2.0 データレプリケーション	109
8.2. クラスター設定	110
8.2.1. 双方向レプリケーション (active/active)	110
8.2.2. 一方方向レプリケーション (active/passive)	111
8.2.3. トピック設定の同期	111
8.2.4. データの整合性	111
8.2.5. オフセットの追跡	111
8.2.6. コンシューマーグループオフセットの同期	112
8.2.7. 接続性チェック	112
8.3. コネクタ設定	113
8.4. コネクタプロデューサーおよびコンシューマーの設定	116
8.5. タスクの最大数を指定	117
8.6. ACL ルールの同期	118
8.7. MIRRORMAKER 2.0 を専用モードで実行する	118
8.8. レガシーモードでの MIRRORMAKER 2.0 の使用	121
第9章 大量のメッセージ処理	124
9.1. 大量メッセージ用の KAFKA CONNECT の設定	125
9.2. 大量のメッセージ用に MIRRORMAKER 2.0 を設定する	126
第10章 KAFKA の管理	127
10.1. 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	127
10.2. KAFKA クラスターのスケーリング	128
10.2.1. Kafka クラスターへのブローカーの追加および削除	128

10.2.2. パーティションの再割り当て	129
10.2.2.1. 再割り当て JSON ファイル	129
10.2.2.2. 再割り当て JSON ファイルの生成	130
10.2.2.3. 手動による再割り当て JSON ファイルの作成	131
10.2.3. 再割り当てスロットル	131
10.2.4. Kafka クラスターのスケールアップ	131
10.2.5. Kafka クラスターのスケールダウン	132
10.2.6. ZooKeeper クラスターのスケールアップ	134
10.2.7. ZooKeeper クラスターのスケールダウン	135
第11章 KAFKA クライアントの追加	136
11.1. MAVEN プロジェクトへの依存関係としての KAFKA クライアントの追加	136
第12章 KAFKA ストリーム API の追加	138
12.1. MAVEN プロジェクトへの依存関係としての KAFKA STREAMS API の追加	138
第13章 KERBEROS (GSSAPI) 認証の使用	140
13.1. KERBEROS (GSSAPI) 認証を使用するための AMQ STREAMS の設定	140
認証用のサービスプリンシパルの追加	140
Kerberos ログインを使用するための ZooKeeper の設定	141
Kerberos ログインを使用するための Kafka ブローカーサーバーの設定	143
Kerberos 認証を使用するための Kafka プロデューサーおよびコンシューマクライアントの設定	144
第14章 CRUISE CONTROL を使用したクラスターのリバランス	147
14.1. CRUISE CONTROL とは	148
14.2. CRUISE CONTROL アーカイブのダウンロード	148
14.3. CRUISE CONTROL METRICS REPORTER のデプロイ	148
14.4. CRUISE CONTROL の設定および起動	150
自動作成されたトピック	151
14.5. 最適化ゴールの概要	152
14.5.1. 優先度のゴールの順序	152
14.5.2. Cruise Control プロパティファイルのゴール設定	153
14.5.3. ハードおよびソフト最適化目標	153
14.5.4. メイン最適化ゴール	154
14.5.5. デフォルトの最適化ゴール	154
14.5.6. ユーザー提供の最適化ゴール	155
14.6. 最適化プロポーザルの概要	155
14.6.1. エンドポイントのリバランス	156
14.6.2. 最適化プロポーザルの承認または拒否	156
14.6.3. 最適化プロポーザルサマリーのプロパティ	158
14.6.4. キャッシュされた最適化プロポーザル	160
14.7. リバランスパフォーマンスチューニングの概要	160
パーティション再割り当てコマンド	160
レプリカの移動ストラテジー	160
リバランスチューニングオプション	161
14.8. CRUISE CONTROL の設定	164
容量の設定	164
Cruise Control Metrics トピックのログクリーンアップポリシー	165
ロギングの設定	166
14.9. 最適化プロポーザルの生成	166
非同期応答	169
14.10. 最適化プロポーザルの承認	170
14.11. アクティブなクラスターリバランスの停止	171

第15章 分散トレースの設定	173
AMQ Streams によるトレーシングのサポート方法	173
手順の概要	174
15.1. OPENTRACING および JAEGER の概要	174
15.2. KAFKA クライアントのトレーシング設定	175
15.2.1. Kafka クライアント用の Jaeger トレーサーの初期化	175
15.2.2. 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	176
Decorator パターンのカスタムスパン名	177
ビルトインスパン名	178
15.2.3. 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	179
15.3. MIRRORMAKER および KAFKA CONNECT のトレース設定	180
15.3.1. MirrorMaker のトレースの有効化	180
15.3.2. MirrorMaker 2.0 のトレースの有効化	180
15.3.3. Kafka Connect のトレースの有効化	181
15.4. KAFKA BRIDGE のトレースの有効化	182
15.5. トレースの環境変数	182
第16章 KAFKA EXPORTER の使用	185
16.1. コンシューマーラグ	185
16.2. KAFKA EXPORTER アラートルールの例	186
16.3. KAFKA EXPORTER メトリクス	186
16.4. KAFKA EXPORTER の実行	187
16.5. GRAFANA での KAFKA EXPORTER メトリクスの表示	189
第17章 AMQ STREAMS および KAFKA のアップグレード	191
17.1. アップグレードの前提条件	191
17.2. KAFKA バージョン	191
17.3. KAFKA ブローカーおよび ZOOKEEPER のアップグレード	192
17.4. KAFKA CONNECT のアップグレード	195
17.5. コンシューマーおよび KAFKA STREAMS アプリケーションの COOPERATIVE REBALANCING へのアップグレード	196
第18章 JMX を使用したクラスターの監視	198
18.1. JMX 設定オプション	198
18.2. JMX エージェントの無効化	198
18.3. 別のマシンからの JVM への接続	198
18.4. JCONSOLE を使用した監視	199
18.5. 重要な KAFKA ブローカーメトリクス	199
18.5.1. Kafka サーバーメトリクス	200
18.5.2. Kafka ネットワークメトリクス	202
18.5.3. Kafka ログメトリクス	203
18.5.4. Kafka コントローラーメトリクス	204
18.5.5. Yammer メトリクス	204
18.6. プロデューサー MBEAN	205
プロデューサーメトリクス	205
ブローカー接続に関するプロデューサーメトリクス	208
トピックに送信されたメッセージに関するプロデューサーメトリクス	209
18.7. コンシューマー MBEAN	210
コンシューマーメトリクス	210
ブローカー接続に関するコンシューマーメトリクス	212
コンシューマーグループメトリクス	213
コンシューマーフェッチメトリクス	215
トピックレベルでのコンシューマーフェッチメトリクス	215
パーティションレベルでのコンシューマーフェッチメトリクス	216

18.8. KAFKA CONNECT MBEAN	216
Kafka Connect メトリクス	217
ブローカーコネクションに関する Kafka Connect メトリクス	219
ワーカーに関する Kafka Connect メトリクス	220
リバランスに関する Kafka Connect メトリクス	220
コネクタに関する Kafka Connect メトリクス	221
コネクタタスクに関する Kafka Connect メトリクス	221
シンクコネクタに関する Kafka Connect メトリクス	222
ソースコネクタに関する Kafka Connect メトリクス	223
コネクタエラーに関する Kafka Connect メトリクス	224
18.9. KAFKA STREAMS MBEAN	225
クライアントの Kafka Streams メトリクス	225
タスクの Kafka Streams メトリクス	226
プロセッサノードの Kafka Streams メトリクス	227
ステートストアの Kafka Streams メトリクス	228
レコードキャッシュの Kafka Streams メトリクス	230
付録A サブスクリプションの使用	231
アカウントへのアクセス	231
サブスクリプションのアクティベート	231
Zip および Tar ファイルのダウンロード	231
DNF を使用したパッケージのインストール	231

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#) をご覧ください。

第1章 AMQ STREAMS の概要

Red Hat AMQ Streams は、Apache ZooKeeper および Apache Kafka プロジェクトをベースとした、スケーラビリティが高く、分散され、高パフォーマンスのデータストリーミングプラットフォームです。

主なコンポーネントは以下で設定されます。

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。

Apache ZooKeeper は Kafka のコア依存関係で、高信頼性の分散調整のためのクラスター調整サービスを提供します。

Kafka Streams API

ストリームプロセッサ 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

AMQ Streams Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターとの対話を可能にする RESTful インターフェ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

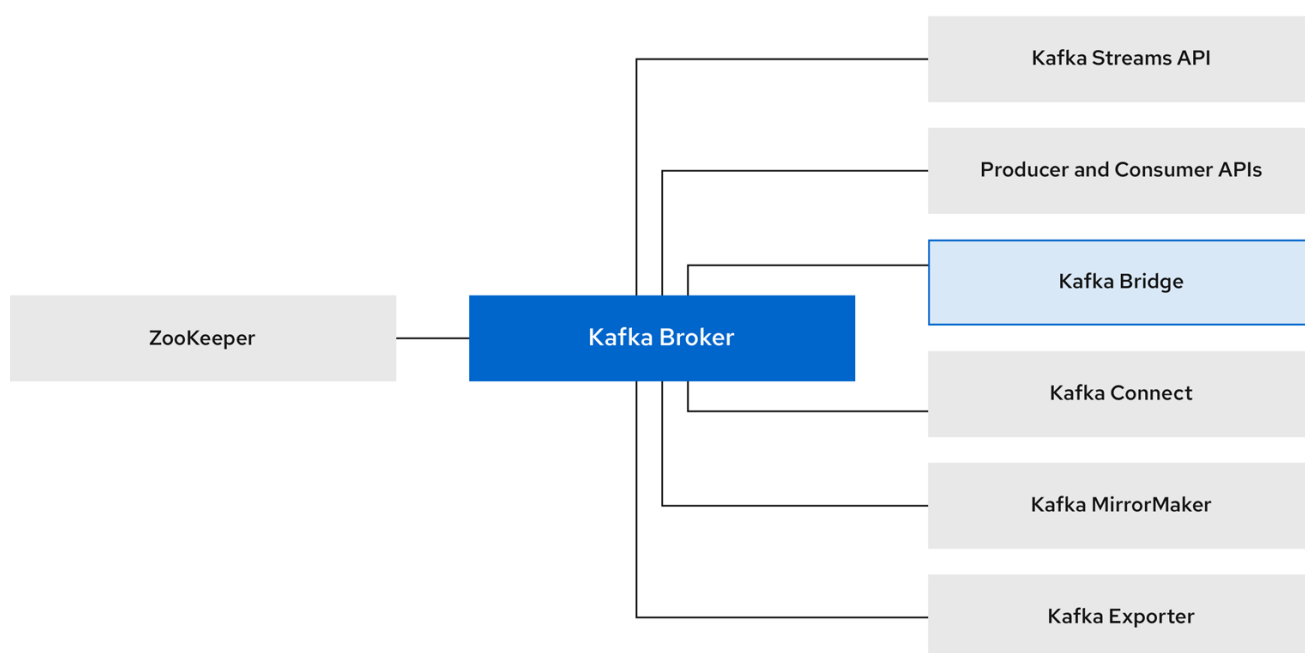
データセンター内またはデータセンター全体の 2 つの Kafka クラスター間でデータをレプリケートします。

Kafka Exporter

監視用に Kafka メトリクスデータの抽出に使用されるエクスポーター。

Kafka 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。ブローカーは、設定データの保存やクラスターの調整に Apache ZooKeeper を使用します。Apache Kafka の実行前に、Apache ZooKeeper クラスターを用意する必要があります。

図1.1 AMQ Streams のアーキテクチャー



222_Streams_0322

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

AMQ Streams 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、[AMQ Streams Kafka Bridge の使用](#) を参照してください。

1.2. サポートされる設定

AMQ Streams をサポートされる設定で実行するには、サポートされるオペレーティングシステムのサポートされる JVM バージョンで AMQ Streams を実行する必要があります。

詳細は、[サポートされている設定](#) を参照してください。

1.3. 本書の表記慣例

ユーザーが置き換えた値

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 **斜体** で表示されます。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、以下のコードでは、**<bootstrap_address>** および **<topic_name>** を独自のアドレスおよびトピック名に置き換えます。

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

第2章 スタートガイド

2.1. AMQ STREAMS のディストリビューション

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

- Apache ZooKeeper
- Apache Kafka
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- Kafka Bridge
- [Kafka Exporter](#)

2.2. AMQ STREAMS アーカイブのダウンロード

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

手順

- [AMQ Streams software downloads](#) ページから、最新バージョンの Red Hat AMQStreams アーカイブをダウンロードします。

2.3. AMQ STREAMS のインストール

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

既存のクラスターを AMQ Streams 2.2 にアップグレードする手順は、[AMQ Streams and Kafka upgrades](#) を参照してください。

前提条件

- [インストールアーカイブ](#) をダウンロードしている。
- [サポートされる設定](#) を確認している。

手順

1. 新しい **kafka** ユーザーおよびグループを追加します。

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

2. **/opt/kafka** ディレクトリーの作成


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

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

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

- 展開した内容を **/opt/kafka** ディレクトリに移動して、一時ディレクトリを削除します。

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

- /opt/kafka** ディレクトリの所有権を **kafka** ユーザーに変更します。

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

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

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

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

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

2.4. データストレージに関する留意事項

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

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。

2.4.1. 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。AMQ Streams 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。

2.4.2. Apache Kafka および ZooKeeper ストレージ

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

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

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



注記

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

関連情報

- [XFS ファイルシステム](#)

2.5. 単一ノードの KAFKA クラスターの実行

この手順では、単一の Apache ZooKeeper ノードと単一の Apache Kafka ノード (両方とも同じホストで実行されている) で設定される基本的な AMQ Streams クラスターを実行する方法を説明します。デフォルトの設定ファイルは ZooKeeper および Kafka に使用されます。



警告

単一ノードの AMQ Streams クラスターは、信頼性および高可用性を提供しないため、開発目的にのみ適しています。

前提条件

- AMQ Streams がホストにインストールされている。

クラスターの実行

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

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

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

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

3. **kafka** ユーザーに切り替えます。

```
su - kafka
```

4. ZooKeeper を起動します。

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

5. ZooKeeper が実行されていることを確認します。

```
jcmd | grep zookeeper
```

以下を返します。

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

6. Kafka を起動します。

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

7. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

以下を返します。

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

2.6. トピックからのメッセージの送受信

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

新しいトピックは、ステップ1で自動的に作成されます。[トピックの自動作成](#)

は、**auto.create.topics.enable** 設定プロパティを使用して制御されます (デフォルトでは **true** に設定されます)。または、クラスターを使用する前にトピックを設定および作成することもできます。詳細は、[トピック](#) を参照してください。

前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [ZooKeeper](#) および [Kafka](#) が稼働している。

手順

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

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

以下に例を示します。

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

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

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

Kafka が新しいトピックを自動的に作成すると、トピックが存在しないという警告が表示される可能性があります。

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

この警告は、さらにメッセージを送信すると再度表示されなくなります。

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

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

以下に例を示します。

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

受信メッセージがコンシューマーコンソールに表示されます。

4. プロデューサーコンソールに切り替え、追加のメッセージを送信します。それらがコンシューマーコンソールに表示されていることを確認します。
5. **Ctrl+C** を押して Kafka コンソールプロデューサーを停止し、コンシューマーを停止します。

2.7. AMQ STREAMS サービスの停止

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

前提条件

- AMQ Streams がホストにインストールされている。
- ZooKeeper および Kafka が稼働している。

手順

1. Kafka ブローカーを停止します。

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

2. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

3. ZooKeeper を停止します。

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

第3章 マルチノード環境の実行

マルチノード環境は、クラスターとして動作する多数のノードで設定されます。レプリケートされた ZooKeeper ノードのクラスターとブローカーノードのクラスターを作成し、ブローカー全体でトピックを複製することができます。

マルチノード環境は、安定性と可用性を提供します。

3.1. マルチノードの ZOOKEEPER クラスターの実行

ZooKeeper をマルチノードクラスターとして設定し、実行します。

前提条件

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

クラスターの実行

1. `/var/lib/zookeeper/` に **myid** ファイルを作成します。最初の ZooKeeper ノードに ID **1** を、2 番目の ZooKeeper ノードに **2** を、それぞれ入力します。

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

以下に例を示します。

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

2. ZooKeeper 設定ファイル `/opt/kafka/config/zookeeper.properties` を編集します。
 - **dataDir** オプションを `/var/lib/zookeeper/` に設定します。
 - **initLimit** および **syncLimit** オプションを設定します。
 - **reconfigEnabled** および **standaloneEnabled** オプションを設定します。
 - すべての ZooKeeper ノードの一覧を追加します。この一覧には、現在のノードも含まれている必要があります。

5つのメンバーを持つ ZooKeeper クラスターのノードの設定例

```
tickTime=2000
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
reconfigEnabled=true
standaloneEnabled=false
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,REPLICATION:PLAINTEXT

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

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

3. デフォルトの設定ファイルで ZooKeeper を起動します。

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

4. ZooKeeper が稼働していることを確認します。

```
jcmd | grep zookeeper
```

以下を返します。

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

5. クラスターのすべてのノードでこの手順を繰り返します。
6. **ncat** ユーティリティを使用して、**stat** コマンドを各ノードに送信して、すべてのノードがクラスターのメンバーであることを確認します。

ncat stat を使用してノードのステータスを確認します。

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

stat のような 4 文字の単語コマンドを使用するには、**zookeeper.properties** で **4lw.commands.whitelist=*** を指定する必要があります。

この出力で、ノードが **leader** または **follower** のいずれかであることがわかります。

ncat コマンドの出力例

```
ZooKeeper version: 3.4.13-2d71af4dbe22557fda74f9a9b4309b15a7487f03, built on
06/29/2018 00:39 GMT
Clients:
/0:0:0:0:0:0:1:59726[0](queued=0,recved=1,sent=0)

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

3.2. マルチノードの KAFKA クラスターの実行

Kafka をマルチノードクラスターとして設定し、実行します。

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- ZooKeeper クラスターが [設定され、稼働している](#)。

クラスターの実行

AMQ Streams クラスターの各 Kafka ブローカーに対して以下を行います。

1. 以下のように、Kafka 設定ファイル `/opt/kafka/config/server.properties` を編集します。
 - 最初のブローカーの **broker.id** フィールドを **0** に、2 番目のブローカーを **1** に、それぞれ設定します。
 - **zookeeper.connect** オプションで ZooKeeper への接続の詳細を設定します。
 - Kafka リスナーを設定します。
 - **logs.dir** のディレクトリーに、コミットログが保存されるディレクトリーを設定します。以下は、Kafka ブローカーの設定例です。

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

各 Kafka ブローカーが同じハードウェアで実行されている通常のインストールでは、**broker.id** 設定プロパティーのみがブローカー設定ごとに異なります。

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/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
```

以下を返します。

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

4. **ncat** ユーティリティを使用して **dump** コマンドを ZooKeeper ノードのいずれかに送信して、すべてのノードが Kafka クラスターのメンバーであることを確認します。

ncat dump を使用して、ZooKeeper に登録されているすべての Kafka ブローカーを確認します。

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

dump のような 4 文字の単語コマンドを使用するには、**zookeeper.properties** で **4lw.commands.whitelist=*** を指定する必要があります。

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

3つのノードで設定される Kafka クラスターの ncat コマンドの出力例

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

3.3. 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、複製数が足りていないトピックパーティションがないかをチェックして、複製パーティションの数が十分にあることを確認します。

トピックを複製し、少なくとも1つのレプリカが同期していることを確認する場合は、可用性を失うことなく、正常な再起動のみを実行できます。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、同期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複製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

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。
- ZooKeeper クラスターが [設定され、稼働している](#)。
- Kafka クラスターが想定どおりに動作している。
複製不足のパーティションや、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、複製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。アクティブなコントローラーの最後のブローカーで手順を実行します。それ以外の場合、アクティブなコントローラーは、再起動を複数回行う時に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 ブローカーおよび ZooKeeper 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/server.properties
```

4. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

以下を返します。

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

5. **ncat** ユーティリティを使用して **dump** コマンドを ZooKeeper ノードのいずれかに送信して、すべてのノードが Kafka クラスターのメンバーであることを確認します。

ncat dump を使用して、ZooKeeper に登録されているすべての Kafka ブローカーを確認します。

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

dump のような 4 文字の単語コマンドを使用するには、**zookeeper.properties** で **4lw.commands.whitelist=*** を指定する必要があります。

出力には、起動した Kafka ブローカーが含まれている必要があります。

3つのノードで設定される Kafka クラスターの ncat コマンドの出力例

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

6. ブローカーで、複製不足の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、メトリックを使用できます。

- **--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 <bootstrap_address> --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、クラスターで複製数が足りていないトピックを表示します。

パーティションの複製数が足りていないトピック

```
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
# ...
```

ISR(同期レプリカ) の数がレプリカの数より少ない場合、複製が足りないパーティションが一覧表示されます。一覧が返されない場合は、複製の数が足りないパーティションはありません。

- **UnderReplicatedPartitions** メトリクスを使用します。

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

このメトリックで、レプリカの数が多いパーティションの数わかります。数がゼロになるまで待機します。

ヒント

トピックに複製数が不足している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。

ブローカーのシャットダウンに成功したログ

```
# ...
[2022-06-08 14:32:29,885] INFO Terminating process due to signal SIGTERM
(org.apache.kafka.common.utils.LoggingSignalHandler)
[2022-06-08 14:32:29,886] INFO [KafkaServer id=0] shutting down (kafka.server.KafkaServer)
[2022-06-08 14:32:29,887] INFO [KafkaServer id=0] Starting controlled shutdown
(kafka.server.KafkaServer)
```

```
[2022-06-08 14:32:29,896] INFO [KafkaServer id=0] Controlled shutdown request returned  
successfully after 6ms (kafka.server.KafkaServer)  
# ...
```

ブローカーの再起動に成功したログ

```
# ...  
[2022-06-08 14:39:35,245] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)  
# ...
```

関連情報

- [「重要な Kafka ブローカーメトリクス」](#)
- [「ロギング」](#)
- [Kafka 設定のチューニング](#)

第4章 KRAFT モードでの KAFKA の実行 (開発プレビュー)

AMQ Streams を KRaft (Kafka Raft メタデータ) モードで実行すると、Kafka クラスターは、ZooKeeper ではなく、コントローラーの内部クォーラムによって管理されます。

Apache Kafka は、ZooKeeper の必要性を段階的に取り除く過程にあります。KRaft モードを試すことができるようになりました。ZooKeeper を使用せずに KRaft モードで Kafka クラスターをデプロイできます。

注意

KRaft モードは実験的なものであり、開発とテスト **のみ** を目的としているため、運用環境では有効にしないでください。

現在、AMQ Streams の KRaft モードには、次の主要な制限があります。

- ZooKeeper を使用する Kafka クラスターから KRaft クラスターへの移動、またはその逆の移動はサポートされていません。
- Apache Kafka バージョンのアップグレードとダウングレードはサポートされていません。
- SCRAM-SHA-512 認証はサポートされていません。
- 複数のディスクを使用する JBOD ストレージはサポートされていません。
- 多くの設定オプションはまだ開発中です。

4.1. KRAFT モードの KAFKA で AMQ STREAMS を使用する

Kafka を KRaft モードで使用する場合、クラスターの調整やメタデータの保存に ZooKeeper を使用する必要はありません。Kafka は、コントローラーとして機能するブローカーを使用して、クラスター自体を調整します。Kafka は、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されるメタデータも保存します。

クラスターを識別するには、ID を作成します。ID は、クラスターに追加するブローカーのログを作成するときに使用されます。

各ブローカーノードの設定で、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ブローカー (または **投票者**) のリスト

ブローカーは、ブローカー、コントローラー、またはその両方のロールを果たします。

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、追跡メタデータを管理します。

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る使用できますが、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、より単純な展開でより効率的になります。

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。

4.2. 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。単一ノードまたは複数ノードの Kafka クラスターを使用している場合は、Kraft モードで Kafka を実行できます。安定性と可用性のために、最低 3 つのブローカーノードとコントローラーノードを実行します。

ブローカーがコントローラーにもなれるように、ブローカーのロールを設定します。設定プロパティ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。

これらのプロパティファイルのサンプルでブローカー設定をベースできます。この手順では、**server.properties** の設定例を使用します。

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。

手順

1. **kafka-storage** ツールを使用して Kafka クラスターの ID を生成します。

```
/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. クラスターの各ブローカーに設定プロパティファイルを作成します。
Kafka で提供される例でファイルをベースできます。
 - a. ロールを **broker**, **controller** または **broker, controller** として指定します
たとえば、**process.roles=broker, controller**
 - 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。
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
```

以下を返します。

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

<uuid> を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ファイルを使用してブローカー設定を適用します。

server.properties 設定ファイルに指定されたデフォルトのログディレクトリーの場所は **/tmp/kraft-combined-logs** です。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 ブローカーを起動します。

```
/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
```

以下を返します。

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

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、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。

第5章 RHEL デプロイメントでの AMQ STREAMS の設定

Kafka および ZooKeeper プロパティファイルを使用して AMQ Streams を設定します。

ZooKeeper

/kafka/config/zookeeper.properties

Kafka

/kafka/config/server.properties

プロパティファイルは Java 形式であり、各プロパティは以下の形式で個別の行にあります。

<option> = <value>

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

This is a comment

改行/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することができます。

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

変更をプロパティファイルに保存したら、Kafka ブローカーまたは ZooKeeper 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プロセスを繰り返す必要があります。

5.1. 標準の KAFKA 設定プロパティの使用

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

プロパティは、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
- クライアント (プロデューサーとコンシューマー)
- 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、承認、認証、および暗号化を設定するオプションが含まれます。



注記

AMQ Streams on OpenShift では、一部の設定プロパティは AMQ Streams によって完全に管理されており、変更できません。

Kafka 設定プロパティの詳細と、プロパティ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](#)
- [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 コネクタ設定のクレデンシャルを提供します。

前提条件

- AMQ Streams がホストにダウンロードされ、インストールされている。
- 環境変数設定プロバイダーの JAR ファイル
JAR ファイルは [AMQ Streams 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
config.providers.env.class=io.strimzi.kafka.EnvVarConfigProvider
```

3. プロパティ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. ZOOKEEPER の設定

Kafka は ZooKeeper を使用して設定データを保存し、クラスターの連携を行います。レプリケートされた ZooKeeper インスタンスのクラスターを実行することが強く推奨されます。

5.3.1. 基本設定

最も重要な ZooKeeper 設定オプションは次のとおりです。

tickTime

ZooKeeper の基本時間単位 (ミリ秒)。ハートビートとセッションのタイムアウトに使用されます。たとえば、最小セッションタイムアウトは 2 ティックになります。

dataDir

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

clientPort

クライアントが接続できるポート番号。デフォルトは **2181** です。

`config/zookeeper.properties` という名前の ZooKeeper 設定ファイルのサンプルは、AMQ Streams のインストールディレクトリーに置かれます。ZooKeeper でレイテンシーを最小限に抑えるために、別のディスクデバイスに **dataDir** ディレクトリーを配置することが推奨されます。

ZooKeeper 設定ファイルは `/opt/kafka/config/zookeeper.properties` に置く必要があります。設定ファイルの基本的な例は以下で確認できます。設定ファイルは **kafka** ユーザーが読み取りできる必要があります。

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

5.3.2. ZooKeeper クラスター設定

多くの実稼働環境では、レプリケートされた ZooKeeper インスタンスのクラスターをデプロイすることが推奨されます。安定した高可用性 ZooKeeper クラスターの実行は、信頼できる ZooKeeper サービスにとって重要です。ZooKeeper クラスターは **ensembles** とも呼ばれます。

ZooKeeper クラスターは通常、奇数のノードで設定されます。ZooKeeper では、クラスター内のほとんどのノードが稼働している必要があります。以下に例を示します。

- 3つのノードで設定されるクラスターでは、少なくとも2つのノードが稼働している必要があります。これは、1つのノードが停止していることを許容できることを意味します。
- 5つのノードで設定されるクラスターでは、最低でも3つのノードが利用可能である必要があります。これは、2つのノードが停止していることを許容できることを意味します。
- 7つのノードで設定されるクラスターでは、最低でも4つのノードが利用可能である必要があります。これは、3つのノードが停止していることを許容できることを意味します。

ZooKeeper クラスターにより多くのノードがあると、クラスター全体の回復性と信頼性が向上します。

ZooKeeper は、偶数のノードを持つクラスターで実行できます。ただし、追加のノードは、クラスターの回復性は向上しません。4つのノードで設定されるクラスターでは、少なくとも3つのノードが利用可能で、1つのノードがダウンしているノードのみを許容する必要があります。そのため、3つのノードのみを持つクラスターと全く同じ回復性があります。

理想的には、異なる ZooKeeper ノードを異なるデータセンターまたはネットワークセグメントに置く必要があります。ZooKeeper ノードの数を増やすと、クラスターの同期に費やされたワークロードが増えます。ほとんどの Kafka のユースケースでは、3、5、または7つのノードで設定される ZooKeeper

クラスターでは十分です。



警告

3つのノードで設定される ZooKeeper クラスターは、利用できないノードを1つだけ許容できます。つまり、クラスターノードがクラッシュし、別のノードでメンテナンスを実施している場合、ZooKeeper クラスターが利用できなくなります。

レプリケートされた ZooKeeper 設定は、スタンドアロン設定でサポートされるすべての設定オプションをサポートします。クラスターリング設定にさらにオプションが追加されます。

initLimit

フォロワーがクラスターリーダーに接続して同期できるようにする時間。時間はティック数として指定されます (詳細は [tickTime オプション](#) を参照してください)。

syncLimit

フォロワーがリーダーの背後にあることのできる時間。時間はティック数として指定されます (詳細は [tickTime オプション](#) を参照してください)。

reconfigEnabled

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

standaloneEnabled

ZooKeeper が1つのサーバーでのみ実行されるスタンドアロンモードを有効または無効にします。

上記のオプションの他に、すべての設定ファイルに ZooKeeper クラスターのメンバーである必要があるサーバーの一覧が含まれている必要があります。サーバーレコードは **server.id=hostname:port1:port2** の形式で指定する必要があります。ここで、

id

ZooKeeper クラスターノードの ID。

hostname

ノードが接続をリッスンするホスト名または IP アドレス。

port1

クラスター内通信に使用されるポート番号。

port2

リーダーエレクトションに使用されるポート番号。

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

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

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

ヒント

4 文字のコマンドを使用するには、**zookeeper.properties** で **4lw.commands.whitelist=*** を指定します。

myid ファイル

ZooKeeper クラスターの各ノードには、一意の **ID** を割り当てる必要があります。各ノードの **ID** は **myid** ファイルで設定し、**/var/lib/zookeeper/** のように **dataDir** フォルダーに保存する必要があります。**myid** ファイルには、テキストとして **ID** が記述された単一行のみが含まれている必要があります。**ID** には、1 から 255 までの任意の整数を指定することができます。このファイルは、各クラスターノードに手動で作成する必要があります。このファイルを使用すると、各 ZooKeeper インスタンスは設定ファイルの対応する **server.** 行の設定を使用して、そのリスナーを設定します。また、他の **server.** 行すべてを使用して、他のクラスターメンバーを特定します。

上記の例では、3 つのノードがあるので、各ノードは値がそれぞれ **1**、**2**、および **3** の異なる **myid** を持ちます。

5.3.3. 認証

デフォルトでは、ZooKeeper はどのような認証も使用せず、匿名の接続を許可します。ただし、Simple Authentication and Security Layer(SASL) を使用した認証の設定に使用できる Java Authentication and Authorization Service (JAAS) をサポートします。ZooKeeper は、ローカルに保存されたクレデンシャルと DIGEST-MD5 SASL メカニズムを使用した認証をサポートします。

5.3.3.1. SASL を使用した認証

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

JAAS はコンテキストを使用して設定されます。サーバーとクライアントなどの個別の部分は、常に別のコンテキストで設定されます。コンテキストは **設定 オプション**で、以下の形式となっています。

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

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

サーバー間の認証

サーバー間の認証では、JAAS 設定ファイルには 2 つの部分が含まれます。

- サーバー設定
- クライアント設定

DIGEST-MD5 SASL メカニズムを使用する場合、認証サーバーの設定に **QuorumServer** コンテキストが使用されます。暗号化されていない形式で、接続できるすべてのユーザー名とパスワードが含まれている必要があります。2つ目のコンテキスト **QuorumLearner** は、ZooKeeper に組み込まれるクライアント用に設定する必要があります。また、暗号化されていない形式のパスワードも含まれます。DIGEST-MD5 メカニズムの JAAS 設定ファイルの例は、以下を参照してください。

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

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

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

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

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

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

サーバー間の認証の詳細は、[ZooKeeper Wiki](#) を参照してください。

クライアント/サーバー間の認証

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

JAAS 設定ファイルにサーバーコンテキストを追加して、クライアント/サーバー間の認証を設定します。DIGEST-MD5 メカニズムの場合は、すべてのユーザー名とパスワードを設定します。

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

JAAS コンテキストの設定後、以下の行を追加して ZooKeeper 設定ファイルでクライアント/サーバー間の認証を有効にします。

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

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

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

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

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

5.3.3.2. DIGEST-MD5 を使用したサーバー間の認証の有効化

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

前提条件

- AMQ Streams がホストにインストールされている。
- ZooKeeper クラスターが複数のノードで [設定](#) されている。

SASL DIGEST-MD5 認証の有効化

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

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

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

ユーザー名とパスワードは両方の JAAS コンテキストで同一である必要があります。以下に例を示します。

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

QuorumLearner {
```

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

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

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

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

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

5.3.3.3. DIGEST-MD5 を使用したクライアント/サーバー間の認証の有効化

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

前提条件

- AMQ Streams がホストにインストールされている。
- ZooKeeper クラスターが [設定され、稼働している](#)。

SASL DIGEST-MD5 認証の有効化

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

```
Server {
  org.apache.zookeeper.server.auth.DigestLoginModule required
  user_super="<SuperUserPassword>"
  user<Username1>_="<Password1>" user<Username2>_="<Password2>";
};
```

super は自動的に管理者特権を持たせます。このファイルには複数のユーザーを含めることができますが、Kafka ブローカーが必要とする追加ユーザーは1つだけです。Kafka ユーザーに推奨される名前は **kafka** です。

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

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



```
user_kafka="123456";
};
```

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

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

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

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

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

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

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

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

5.3.4. 承認

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

Kafka ブローカーで ZooKeeper ACL を有効にする方法は、[「ZooKeeper の承認」](#) を参照してください。

5.3.5. TLS

ZooKeeper は、暗号化または認証用に TLS をサポートします。

5.3.6. その他の設定オプション

ユースケースに基づいて、以下の追加の ZooKeeper 設定オプションを設定できます。

maxClientCnxns

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

autopurge.snapRetainCount

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

autopurge.purgeInterval

スナップショットをパージするための時間間隔 (時間単位)。デフォルト値は **0** で、このオプションは無効になります。

利用可能なすべての設定オプションは、[ZooKeeper のドキュメント](#) を参照してください。

5.3.7. ロギング

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

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

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

5.4. 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/server.properties** です。設定ファイルは **kafka** ユーザーが読み取りできる必要があります。

AMQ Streams 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。AMQ Streams インストールディレクトリーの **config/server.properties** を参照してください。

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。

5.4.1. ZooKeeper

Kafka ブローカーは、設定の一部を保存し、クラスターを調整するために (たとえば、どのノードがどのパーティションのリーダーであるかを決定するために) Zoo Keeper を必要とします。ZooKeeper クラスターの接続の詳細は、設定ファイルに保存されます。**zookeeper.connect** フィールドには、zookeeper クラスターのメンバーのホスト名およびポートのコンマ区切りリストが含まれます。

以下に例を示します。

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

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

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

5.4.2. 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スンできます。

リスナーを設定するには、設定ファイル (`/opt/kafka/config/server.properties`) の **listeners** プロパティを編集します。**listeners** プロパティにコンマ区切りのリストとしてリスナーを追加します。各プロパティを以下のように設定します。

```
<listenerName>://<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一覧を提供し、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** プロパティを使用して、**listeners** プロパティ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 プロパティは **listeners** プロパティ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複製
- パーティションリーダーシップの変更など、コントローラーからの管理タスクの処理

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、**inter.broker.listener.name** プロパティで 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
```

コントロールプレーンリスナー

デフォルトでは、コントローラーと他のブローカー間の通信は [inter-broker リスナー](#) を使用します。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、管理タスクを調整します。

コントローラー接続用に専用の **コントロールプレーンリスナー** を有効にすることができます。コントロールプレーンリスナーは、任意のポートに割り当てることができます。

コントロールプレーンリスナーを有効にするには、リスナー名で **control.plane.listener.name** プロパティを設定します。

```
listeners=CONTROLLER://0.0.0.0:9090,REPLICATION://0.0.0.0:9091
...
control.plane.listener.name=CONTROLLER
```

コントロールプレーンリスナーを有効にすると、コントローラーの通信がブローカー間のデータレプリケーションによって遅延しないため、クラスターのパフォーマンスが向上する可能性があります。データレプリケーションは、inter-broker のリスナーを介して続行されます。

control.plane.listener が設定されていない場合、コントローラー接続には [inter-broker のリスナー](#) が使用されます。

5.4.3. ログのコミット

Apache Kafka は、プロデューサーから受信するすべてのレコードをコミットログに保存します。コミットログには、Kafka が配信する必要がある実際のデータ (レコードの形式) が含まれます。これらは、ブローカーの動作を記録するアプリケーションログファイルではありません。

ログディレクトリー

log.dirs プロパティ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
```

5.4.4. ブローカー ID

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

```
broker.id=1
```

5.4.5. ZooKeeper の認証

デフォルトでは、ZooKeeper と Kafka 間の接続は認証されません。ただし、Kafka および ZooKeeper は、SASL (Simple Authentication and Security Layer) を使用して認証を設定するのに使用できる Java Authentication and Authorization Service (JAAS) をサポートします。ZooKeeper は、ローカルに保存されたクレデンシャルと DIGEST-MD5 SASL メカニズムを使用した認証をサポートします。

5.4.5.1. JAAS 設定

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

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

5.4.5.2. ZooKeeper 認証の有効化

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

前提条件

- ZooKeeper でクライアント/サーバー間の認証が [有効である](#)。

SASL DIGEST-MD5 認証の有効化

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

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

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

Client コンテキストの例を以下に示します。

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

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

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

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。

関連情報

- [認証](#)

5.4.6. 承認

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

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

または、独自の承認プラグインを使用できます。たとえば、[OAuth 2.0 トークンベースの認証](#) を使用している場合、[OAuth 2.0 承認](#) を使用できます。

5.4.6.1. シンプルな ACL オーソライザー

AclAuthorizer を含む authorizer プラグインは **authorizer.class.name** プロパティを使用して有効にします。

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

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

5.4.6.1.1. ACL ルール

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

ACL ルールは以下の形式で定義されます。

プリンシパル **P** は、ホスト **H** から Kafka リソース **R** で操作 **O** を許可または拒否されます。

たとえば、以下のようにルールを設定できます。

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

ACL ルールが存在しない場合

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 **/opt/kafka/config/server.properties** で **allow.everyone.if.no.acl.found** プロパティを **true** に設定すると変更できます。

5.4.6.1.2. 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。

5.4.6.1.3. ユーザーの認証

承認を使用するには、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、[暗号化および認証](#) を参照してください。

5.4.6.1.4. 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
```

5.4.6.1.5. レプリカブローカーの認証

承認を有効にすると、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、承認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。

5.4.6.1.6. サポートされるリソース

Kafka ACL は、以下のタイプのリソースに適用できます。

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

- DelegationToken

5.4.6.1.7. サポートされる操作

AclAuthorizer はリソースでの操作を承認します。

以下の表で **X** の付いたフィールドは、各リソースでサポートされる操作を表します。

表5.1 リソースでサポートされる操作

	トピック	コンシューマーグループ	Cluster
Read	X	X	
Write	X		
Create			X
Delete	X		
Alter	X		
Describe	X	X	X
ClusterAction			X
すべて	X	X	X

5.4.6.1.8. ACL 管理オプション

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

kafka-acls.sh パラメーターオプションを使用して、ACL ルールを追加、一覧表示、および削除したり、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。

オプション	型	説明	デフォルト
add	アクション	ACL ルールを追加します。	
remove	アクション	ACL ルールを削除します。	
list	アクション	ACL ルールを一覧表示します。	

オプション	型	説明	デフォルト
authorizer	アクション	オーソライザーの完全修飾クラス名。	kafka.security.auth.S impleAclAuthorizer
authorizer- properties	設定	<p>初期化のためにオーソライザーに渡されるキー/値のペア。</p> <p>AclAuthorizer では、サンプル値は zookeeper.connect=zoo1.my-domain.com:2181 です。</p>	
bootstrap-server	リソース	Kafka クラスターに接続するためのホスト/ポートのペア。	このオプションまたは authorizer オプションを使用します (両方ではなく)。
command-config	リソース	管理クライアントに渡す設定プロパティファイル。これは bootstrap-server パラメーターと共に使用されます。	
cluster	リソース	クラスターを ACL リソースとして指定します。	
topic	リソース	<p>トピック名を ACL リソースとして指定します。</p> <p>ワイルドカードとして使用されるアスタリスク (*) は、すべてのトピック に解釈されます。</p> <p>1つのコマンドに複数の --topic オプションを指定できます。</p>	
group	リソース	<p>コンシューマーグループ名を ACL リソースとして指定します。</p> <p>1つのコマンドに複数の --group オプションを指定できます。</p>	

オプション	型	説明	デフォルト
transactional-id	リソース	<p>トランザクション ID を ACL リソースとして指定します。</p> <p>トランザクション配信は、プロデューサーによって複数のパーティションに送信されたすべてのメッセージが正常に配信されるか、いずれも配信されない必要があることを意味します。</p> <p>ワイルドカードとして使用されるアスタリスク (*) は、すべての ID に解釈されます。</p>	
delegation-token	リソース	<p>委任トークンを ACL リソースとして指定します。</p> <p>ワイルドカードとして使用されるアスタリスク (*) は、すべてのトークン に解釈されます。</p>	
resource-pattern-type	設定	<p>add パラメーターのリソースパターンのタイプ、または list または remove パラメーターのリソースパターンのフィルター値を指定します。</p> <p>literal または prefixed をリソース名のリソースパターンタイプとして使用します。</p> <p>any または match を、リソースパターンのフィルター値または特定のパターンタイプフィルターとして使用します。</p>	literal

オプション	型	説明	デフォルト
allow-principal	プリンシパル	allow ACL ルールに追加されるプリンシパル。 1つのコマンドに複数の --allow-principal オプションを指定できます。	
deny-principal	プリンシパル	拒否 ACL ルールに追加されるプリンシパル。 1つのコマンドに複数の --deny-principal オプションを指定できます。	
principal	プリンシパル	プリンシパルの ACL の一覧を返すために list パラメーターと共に使用されるプリンシパル名。 1つのコマンドに複数の --principal オプションを指定できます。	
allow-host	Host	--allow-principal に記載されているプリンシパルへのアクセスを許可する IP アドレス。 ホスト名または CIDR 範囲はサポートされていません。	--allow-principal が指定されている場合、デフォルトは * ですべてのホストを意味します。
deny-host	Host	--deny-principal に記載されているプリンシパルへのアクセスを拒否する IP アドレス。 ホスト名または CIDR 範囲はサポートされていません。	--deny-principal が指定されている場合、デフォルトは * ですべてのホストを意味します。
operation	操作	操作を許可または拒否します。 1つのコマンドに複数の --operation オプションを指定できます。	すべて

オプション	型	説明	デフォルト
producer	ショートカット	メッセージプロデューサーが必要とするすべての操作を許可または拒否するためのショートカット (トピックでは WRITE と DESCRIBE、クラスターでは CREATE)。	
consumer	ショートカット	メッセージコンシューマーが必要とするすべての操作を許可または拒否するためのショートカット (トピックについては READ と DESCRIBE、コンシューマーグループについては READ)。	
idempotent	ショートカット	<p>--producer パラメーターとの併用時に冪等性を有効にするショートカット。これにより、メッセージがパーティションに1度だけ配信されるようになります。</p> <p>プロデューサーが特定のトランザクション ID に基づいてメッセージを送信することを許可されている場合、Idempotence は自動的に有効になります。</p>	
force	ショートカット	すべてのクエリーを受け入れ、プロンプトは表示されないショートカット。	

5.4.6.2. 承認の有効化

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

前提条件

- ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。

手順

1. **AclAuthorizer** を使用するように、Kafka 設定ファイル `/opt/kafka/config/server.properties` を編集します。

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

2. Kafka ブローカーを (再) 起動します。

5.4.6.3. ACL ルールの追加

AclAuthorizer は、ユーザーが実行できる/できない操作を記述するルールのセットを定義するアクセス制御リスト (ACL) を使用します。

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインを使用する場合に、ACL ルールを追加する方法を説明します。

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

前提条件

- ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- Kafka ブローカーで承認が [有効](#) である。

手順

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

例:

- **MyConsumerGroup** コンシューマーグループを使用して、**user1** および **user2** の **myTopic** からの読み取りを許可します。

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

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

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

- **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。

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

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

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

関連情報

- [kafka-acls.sh オプション](#)

5.4.6.4. ACL ルールの一覧表示

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインを使用する場合に、既存の ACL ルールを一覧表示する方法を説明します。

ルールは、**kafka-acls.sh** ユーティリティを使用してリストされます。

前提条件

- ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- Kafka ブローカーで承認が [有効](#) である。
- 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。

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

Current ACLs for resource `Topic:myTopic`:

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

関連情報

- [kafka-acls.sh オプション](#)

5.4.6.5. ACL ルールの削除

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインを使用する場合に、ACL ルールを削除する方法を説明します。

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

前提条件

- ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- Kafka ブローカーで承認が [有効](#) である。
- 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。

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

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

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

- **MyConsumerGroup** 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

```
bin/kafka-acls.sh --authorizer-properties zookeeper.connect=zoo1.my-domain.com:2181 --
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。

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

関連情報

- [kafka-acls.sh オプション](#)
- [承認の有効化](#)

5.4.7. ZooKeeper の承認

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

5.4.7.1. ACL 設定

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

プロパティはデフォルトで無効になっていて、**true** に設定することにより有効になります。

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

ACL ルールが有効になっている場合、ZooKeeper で **znode** が作成されると、作成した Kafka ユーザーのみが変更または削除できます。その他のすべてのユーザーには読み取り専用アクセスがあります。

Kafka は、新しく作成された ZooKeeper **znodes** に対してのみ ACL ルールを設定します。ACL がクラスターの最初の起動後にのみ有効である場合、**zookeeper-security-migration.sh** ツールは既存のすべての **znodes** に ACL を設定できます。

ZooKeeper のデータの機密性

ZooKeeper に保存されるデータには以下が含まれます。

- トピック名およびその設定
- SASL SCRAM 認証が使用される場合のソルトおよびハッシュ化されたユーザークレデンシャル

しかし、ZooKeeper は Kafka を使用して送受信されたレコードを保存しません。ZooKeeper に保存されるデータは機密ではないと想定されます。

データが機密として考慮される場合 (たとえば、トピック名にカスタマー ID が含まれるなど)、保護に使用できる唯一のオプションは、ネットワークレベルで ZooKeeper を分離し、Kafka ブローカーにのみアクセスを許可することです。

5.4.7.2. 新しい Kafka クラスターでの ZooKeeper ACL の有効化

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

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- ZooKeeper クラスターが [設定され、稼働している](#)。
- ZooKeeper でクライアント/サーバー間の認証が [有効である](#)。
- Kafka ブローカーで ZooKeeper の認証が [有効である](#)。
- Kafka ブローカーがまだ起動していない。

手順

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

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

2. Kafka ブローカーを起動します。

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

この手順では、稼働している Kafka クラスターの Kafka 設定で ZooKeeper ACL を有効にする方法を説明します。**zookeeper-security-migration.sh** ツールを使用して、既存のすべての **znodes** に ZooKeeper の ACL を設定します。**zookeeper-security-migration.sh** は AMQ Streams の一部として利

用でき、**bin** ディレクトリーにあります。

前提条件

- Kafka クラスタが [設定され、稼働している](#)。

ZooKeeper ACL の有効化

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

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

2. すべての Kafka ブローカーを1つずつ再起動します。
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。
3. **zookeeper-security-migration.sh** ツールを使用して、既存のすべての **znodes** ノードに ACL を設定します。

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

以下に例を示します。

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

5.4.8. 暗号化と認証

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

5.4.8.1. 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ます) を使用するように設定されます。同じセキュリティ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 認証を使用します。以下の章では、TLS および SASL の設定方法について詳しく説明します。

5.4.8.2. 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** プロパティに設定されます。**ssl.keystore.password** プロパティを使用して、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 **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/server-1.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一覧。各暗号スイートは、TLS 接続に使用される認証、暗号化、MAC、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。

5.4.8.3. TLS 暗号化の有効化

この手順では、Kafka ブローカーで暗号化を有効にする方法を説明します。

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。

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. 以下のように、すべてのクラスターノードの **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
 - **listener.security.protocol.map** フィールドを変更して、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。

5.4.8.4. 認証

認証には、以下を使用できます。

- 暗号化接続の X.509 証明書に基づく TLS クライアント認証
- サポートされる Kafka SASL (Simple Authentication and Security Layer) メカニズム
- [OAuth 2.0 のトークンベースの認証](#)

5.4.8.4.1. TLS クライアント認証

TLS クライアント認証は、TLS 暗号化を使用している接続でのみ使用できます。TLS クライアント認証を使用するには、公開鍵のあるトラストストアをブローカーに提供できます。これらのキーは、ブローカーに接続するクライアントを認証するために使用できます。トラストストアは Java Keystore (JKS) 形式で提供され、認証局の公開鍵が含まれている必要があります。トラストストアに含まれる認証局のいずれかによって署名された公開鍵および秘密鍵を持つクライアントはすべて認証されます。トラストストアの場所は、フィールド **ssl.truststore.location** を使用して設定されます。トラストストアがパスワードで保護される場合、**ssl.truststore.password** プロパティで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.truststore.location=/path/to/keystore/server-1.jks
ssl.truststore.password=123456
```

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

none

TLS クライアント認証はオフになっています。(デフォルト値)

requested

TLS クライアント認証は任意です。クライアントは TLS クライアント証明書を使用した認証を要求されますが、選択することはできません。

required

クライアントは 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認証済みクライアント証明書からの識別名になります。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。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 **ANONYMOUS** になります。

5.4.8.4.2. SASL 認証

SASL 認証は、Java Authentication and Authorization Service (JAAS) を使用して設定されます。JAAS は、Kafka と ZooKeeper 間の接続の認証にも使用されます。JAAS は独自の設定ファイルを使用します。このファイルに推奨される場所は **/opt/kafka/config/jaas.conf** です。ファイルは **kafka** ユーザーが読み取りする必要があります。Kafka を実行中の場合、このファイルの場所は Java システムプロパティ **java.security.auth.login.config** を使用して指定されます。このプロパティは、ブローカーノードの起動時に Kafka に渡す必要があります。

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

SASL 認証は、暗号化されていないプレーンの接続と TLS 接続の両方を介してサポートされます。SASL はリスナーごとに個別に有効にできます。これを有効にするには、**listener.security.protocol.map** のセキュリティープロトコルを **SASL_PLAINTEXT** または **SASL_SSL** のいずれかにする必要があります。

Kafka の SASL 認証は、いくつかの異なるメカニズムをサポートします。

PLAIN

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

SCRAM-SHA-256 および SCRAM-SHA-512

Salted Challenge Response Authentication Mechanism (SCRAM) を使用して認証を実装します。SCRAM 認証情報は、ZooKeeper に一元的に保存されます。SCRAM は、ZooKeeper クラスターノードがプライベートネットワークで分離された状態で実行されている場合に使用できます。

GSSAPI

Kerberos サーバーに対して認証を実装します。



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。そのため、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。

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

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

inter-broker 通信に使用されるリスナーが SASL を使用している場合、**sasl.mechanism.inter.broker.protocol** プロパティを使用して使用する SASL メカニズムを指定する必要があります。以下に例を示します。

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

inter-broker 通信に使用されるユーザー名およびパスワードは、フィールド **username** および **password** を使用して **KafkaServer** JAAS コンテキストで指定する必要があります。

SASL プレーン

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

- **admin**

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

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

ユーザーデータベースを持つ JAAS 設定ファイルは、すべての Kafka ブローカーで同期して維持する必要があります。

SASL PLAIN が inter-broker の認証にも使用される場合、**username** および **password** プロパティを JAAS コンテキストに含める必要があります。

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

SASL SCRAM

Kafka の SCRAM 認証は、**SCRAM-SHA-256** および **SCRAM-SHA-512** の2つのメカニズムで設定されます。これらのメカニズムは、使用されるハッシュアルゴリズム (SHA-256 とより強力な SHA-512) のみが異なります。SCRAM 認証を有効にするには、JAAS 設定ファイルに以下の設定を含める必要があります。

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

Kafka 設定ファイルで SASL 認証を有効にすると、両方の SCRAM メカニズムが一覧表示されます。ただし、それらの1つのみを inter-broker 通信に選択できます。以下に例を示します。

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

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

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

ユーザー認証情報を削除するには、以下を使用します。

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

SASL GSSAPI

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

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

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

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

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

複数の SASL メカニズム

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

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

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

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

複数のメカニズムを有効にすると、クライアントは使用するメカニズムを選択できます。

5.4.8.5. TLS クライアント認証の有効化

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

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。
- TLS 暗号化が [有効になっている](#)。

手順

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

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

3. Kafka ブローカーを (再) 起動します。

5.4.8.6. SASL PLAIN 認証の有効化

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

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。

手順

1. `/opt/kafka/config/jaas.conf` JAAS 設定ファイルを編集するか、作成します。このファイルには、すべてのユーザーとそのパスワードが含まれている必要があります。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。
以下に例を示します。

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

2. 以下のように、すべてのクラスターノードの `/opt/kafka/config/server.properties` Kafka 設定ファイルを編集します。
 - **listener.security.protocol.map** フィールドを変更して、SASL PLAIN 認証を使用するリスナーの **SASL_PLAINTEXT** または **SASL_SSL** プロトコルを指定します。
 - **sasl.enabled.mechanisms** オプションを **PLAIN** に設定します。
以下に例を示します。

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

3. KAFKA_OPTS 環境変数を使用して Kafka ブローカーを (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/server.properties
```

5.4.8.7. SASL SCRAM 認証の有効化

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

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。

手順

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

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

2. 以下のように、すべてのクラスターノードの **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
 - **listener.security.protocol.map** フィールドを変更して、SASL SCRAM 認証を使用するリスナーの **SASL_PLAINTEXT** または **SASL_SSL** プロトコルを指定します。
 - **sasl.enabled.mechanisms** オプションを **SCRAM-SHA-256** または **SCRAM-SHA-512** に設定します。
以下に例を示します。

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

3. KAFKA_OPTS 環境変数を使用して Kafka ブローカーを (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/server.properties
```


関連情報

- [SASL SCRAM ユーザーの追加](#)
- [SASL SCRAM ユーザーの削除](#)

5.4.8.8. SASL SCRAM ユーザーの追加

この手順では、SASL SCRAM を使用した認証に新しいユーザーを追加する方法を説明します。

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。
- SASL SCRAM 認証が [有効になっている](#)。

手順

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

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

以下に例を示します。

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

5.4.8.9. SASL SCRAM ユーザーの削除

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

前提条件

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams](#) がインストールされている。
- 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_address> --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
```

5.4.9. OAuth 2.0 トークンベース認証の使用

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

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

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



注記

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

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

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

AMQ Streams のコンテキストでは以下が行われます。

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

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

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

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

RHEL での AMQ Streams には OAuth 2.0 ライブラリーが 2 つ含まれています。

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。

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](#)

5.4.9.1. OAuth 2.0 認証メカニズム

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

可能な限り、OAUTHBEARER を使用するようにクライアントを設定することが推奨されます。OAUTHBEARER では、クライアントクレデンシャルは Kafka ブローカーと **共有されることがない**ため、PLAIN よりも高レベルのセキュリティが提供されます。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

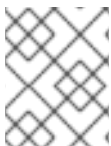
OAUTHBEARER を使用するには、Kafka ブローカーの OAuth 認証リスナー設定で **sasl.enabled.mechanisms** を **OAUTHBEARER** に設定します。詳細な設定は、「[OAuth 2.0 Kafka ブローカーの設定](#)」を参照してください。

```
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。AMQ Streams 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承認サーバーに接続し、アクセストークンを取得する必要があります。

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 認証でできるようにするために、AMQ Streams では **OAuth 2.0 over PLAIN** サーバー側のコールバックが提供されます。

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

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

- クライアント ID および 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、承認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。

5.4.9.1.1. プロパティ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** プロパティを指定する必要がありますが、他の JAAS プロパティを省略できます。
大文字の環境変数の命名規則または大文字の環境変数の命名規則を使用できます。

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

- **oauth.:** 認証の設定

- [strimzi: OAuth 2.0 承認の設定](#)

関連情報

- [OAuth 2.0 Kafka ブローカーの設定](#)

5.4.9.2. 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。

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



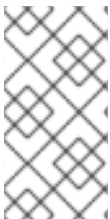
注記

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

5.4.9.2.1. 承認サーバーの OAuth 2.0 クライアント設定

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

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



注記

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

5.4.9.2.2. 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、承認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 ❶
listeners=CLIENT://0.0.0.0:9092 ❷
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT ❸
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER ❹
sasl.mechanism.inter.broker.protocol=OAUTHBEARER ❺
inter.broker.listener.name=CLIENT ❻
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOauthValidatorCallbackHandler ❼
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ ❽
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ ❾
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ ❿
  oauth.username.claim="preferred_username" \ ⓫
  oauth.client.id="kafka-broker" \ ⓬
  oauth.client.secret="kafka-secret" \ ⓭
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/token" ; ⓮
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler ⓯
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000 ⓰
```

- ❶ 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。
- ❷ 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。これは、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。
- ❸ 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。
- ❹ **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** プロパティでは大文字で、**listener.name** プロパティ (**listener.name.client**) では小文字で、**listener.name.client.*** プロパティの一部である場合は小文字で指定されます。
- ❺ inter-broker 通信の **OAUTHBEARER** メカニズムを指定します。
- ❻ 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。
- ❼ クライアントリスナーで OAuth 2.0 認証を設定します。
- ❽ 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 **auth.token.endpoint.uri** プロパティは inter-broker 設定に関連するものです。
- ❾ 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

`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承認サーバーによって異なります。
- 12 すべてのブローカーで同じ Kafka ブローカーのクライアント ID。これは、**kafka-broker** として承認サーバーに登録されたクライアントです。
- 13 すべてのブローカーで同じ Kafka ブローカーのシークレット。
- 14 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。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 (これにのみ必要です)。
- 16 (オプション): トークンの期限が切れるとセッションの期限切れを強制し、**Kafka の再認証メカニズム** も有効にします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

以下の例は、TLS が 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.endpoint.identification.algorithm=HTTPS 4
listener.name.replication.ssl.secure.random.implementation=SHA1PRNG 5
listener.name.replication.ssl.keystore.location=PATH-TO-KEYSTORE 6
listener.name.replication.ssl.truststore.location=PATH-TO-TRUSTSTORE 7
listener.name.replication.ssl.client.auth=required 8
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \
  oauth.username.claim="preferred_username" ; 9
```

- 1 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ネル

- 3 **ssl.** プロパティは 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。承認サーバーとの接続はセキュアな HTTPS 接続を使用する必要があります。

以下の例は、SASL 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.JaasServerOAuthValidatorCallbackHandler 6
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ 7
  oauth.valid.issuer.uri="http://AUTH_SERVER/auth/realms/REALM" \ 8
  oauth.jwks.endpoint.uri="https://AUTH_SERVER/auth/realms/REALM/protocol/openid-connect/certs" \ 9
  oauth.username.claim="preferred_username" \ 10
  oauth.client.id="kafka-broker" \ 11
  oauth.client.secret="kafka-secret" \ 12
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/token" ; 13
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler 14
listener.name.client.plain.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.plain.JaasServerOAuthOverPlainValidatorCallbackHandler 15
listener.name.client.plain.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \ 16
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ 17
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ 18
  oauth.username.claim="preferred_username" \ 19
```



```
oauth.token.endpoint.uri="http://AUTH_SERVER/auth/realms/REALM/protocol/openid-  
connect/token" ; 20  
connections.max.reauth.ms=3600000 21
```

- 1 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。これは、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、ユーザーを識別する **プリンシパル** です。値は、使用される認証フローと承認サーバーによって異なります。
 - 11 すべてのブローカーで同じ Kafka ブローカーのクライアント ID。これは、**kafka-broker** として承認サーバーに登録され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 承認サーバーへの 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、ユーザーを識別する **プリンシパル** です。値は、使用される認証フローと承認サーバーによって異なります。
- 20 承認サーバーへの 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の期限切れを強制し、[Kafka の再認証メカニズム](#) も有効にします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

5.4.9.2.3. 高速なローカル JWT トークン検証の設定

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

ローカルチェックでは、トークンに対して以下が確認されます。

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

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

高速のローカル JWT トークン検証の実行中に、承認サーバーの通信は必要はありません。OAuth 2.0 承認サーバーによって公開される **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

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

TLS リスナーでは、証明書 **トラストストア** を設定し、**トラストストアファイル** をポイントできます。

高速なローカル JWT トークン検証のプロパティの例

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ 1
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ 2
  oauth.jwks.refresh.seconds="300" \ 3
  oauth.jwks.refresh.min.pause.seconds="1" \ 4
```

```

oauth.jwks.expiry.seconds="360" \ 5
oauth.username.claim="preferred_username" \ 6
oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \ 7
oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \ 8
oauth.ssl.truststore.type="PKCS12" ; 9

```

- 1 有効な発行者 URI。この発行者が発行するアクセストークンのみが受け入れられます。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 (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承認サーバーによって異なります。
- 7 TLS 設定で使用す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。

5.4.9.2.4. 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 JWKS エンドポイント URI ではなく、**イントロスペクションエンドポイント URI** を指定します。通常、イントロスペクションエンドポイントは保護されているため、承認サーバーに応じて `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://AUTH-SERVER-ADDRESS/introspection" \ 1
oauth.client.id="kafka-broker" \ 2
oauth.client.secret="kafka-broker-secret" \ 3

```

```

oauth.ssl.truststore.location="PATH-TO-TRUSTSTORE-P12-FILE" \ 4
oauth.ssl.truststore.password="TRUSTSTORE-PASSWORD" \ 5
oauth.ssl.truststore.type="PKCS12" \ 6
oauth.username.claim="preferred_username"; 7

```

- 1 OAuth 2.0 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 設定で使用するトラストストアの場所。
- 5 トラストストアにアクセスするためのパスワード。
- 6 PKCS #12 形式のトラストストアタイプ。
- 7 トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。 **oauth.username.claim** の値は、使用される承認サーバーによって異なります。

5.4.9.3. 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コネクションを破棄せずに再使用して、新しいセッションを即座に開始します。

セッションの再認証はデフォルトで無効になっています。 **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** プロパティを設定します。

リスナーごとにセッションの再認証を指定できます。以下に例を示します。

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

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

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

クライアントの再認証

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

トークンの検証に成功すると、既存の接続を使用して新しいクライアントセッションが開始されます。クライアントが再認証に失敗した場合、さらにメッセージを送受信しようとする、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** 承認を使用したり、カスタムオーソライザーをインストールして、承認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](#)

5.4.9.4. 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。

- 承認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)。
- 承認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承認サーバーでの認証に使用されるクレデンシャルは、ブローカーに送信されません。

クライアントによるアクセストークンの取得後、承認サーバーと通信する必要はありません。

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



注記

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

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

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

5.4.9.5. 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、使用する承認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。

- 承認サーバーと通信しない、JWT の署名確認およびローカルトークンのイントロスペクションをベースとした高速なローカルトークン検証を使用できます。承認サーバーは、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。
- 承認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承認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

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

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ための承認サーバーとの通信 (クライアント ID およびシークレットまたは更新トークンを使用)

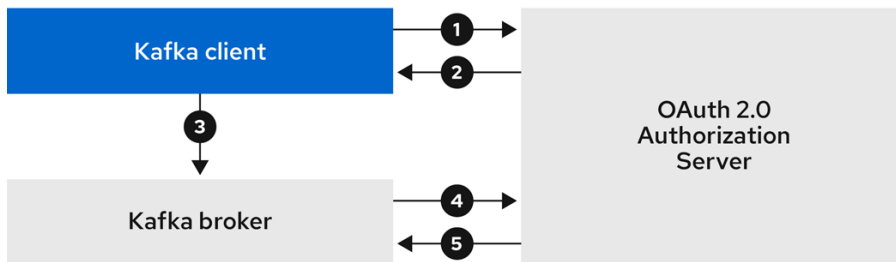
5.4.9.5.1. SASL OAUTHBEARER メカニズムを使用したクライアント認証フローの例

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承認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承認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

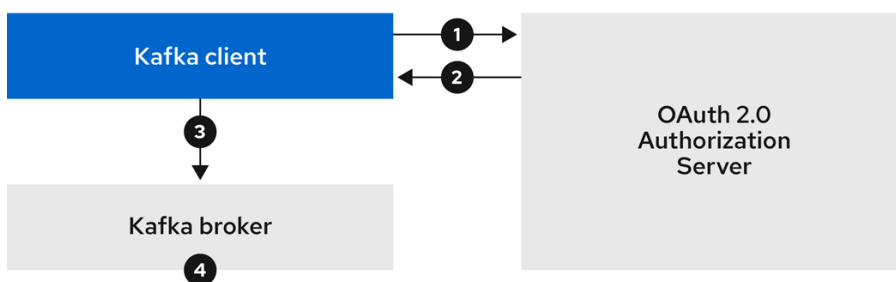
クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を承認サーバーに委任する場合



222_Streams_0322

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承認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。
2. 承認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、承認サーバーでトークンイントロスペクションエンドポイントを呼び出すことで、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

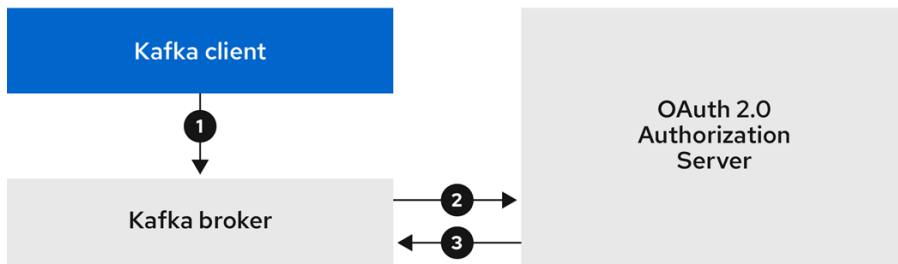


222_Streams_0322

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承認サーバーで認証します。
2. 承認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承認サーバーに委任する場合



222_Streams_0322

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、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。
3. トークンが有効な場合、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



222_Streams_0322

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

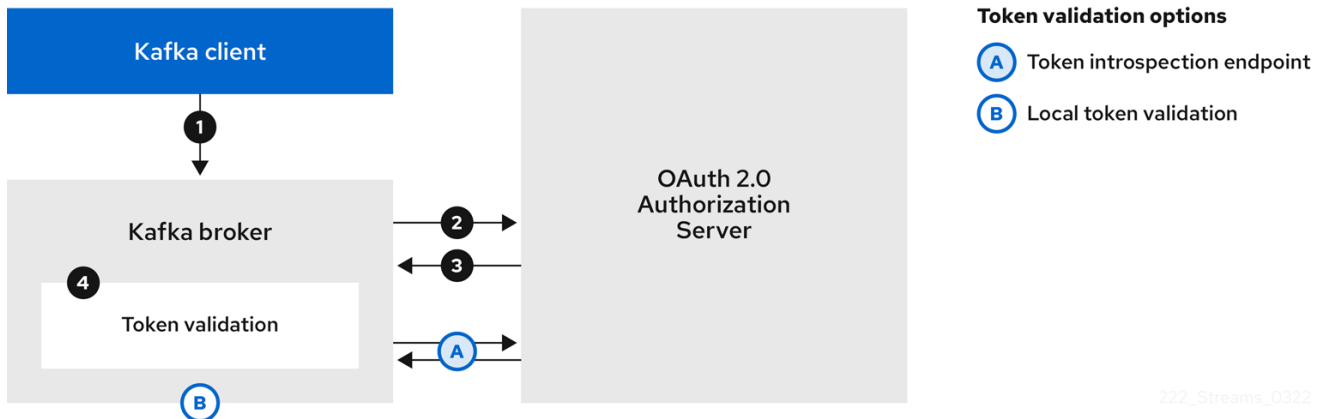
トークンが取り消された場合に承認サーバーとのチェックが行われないため、高速のローカル JWT トークン署名の検証は有効期限の短いトークンにのみ適しています。トークンの有効期限はトークンに書き込まれますが、失効はいつでも発生する可能性があるため、承認サーバーと通信せずに対応することはできません。発行されたトークンはすべて期限切れになるまで有効とみなされます。

5.4.9.5.2. 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。

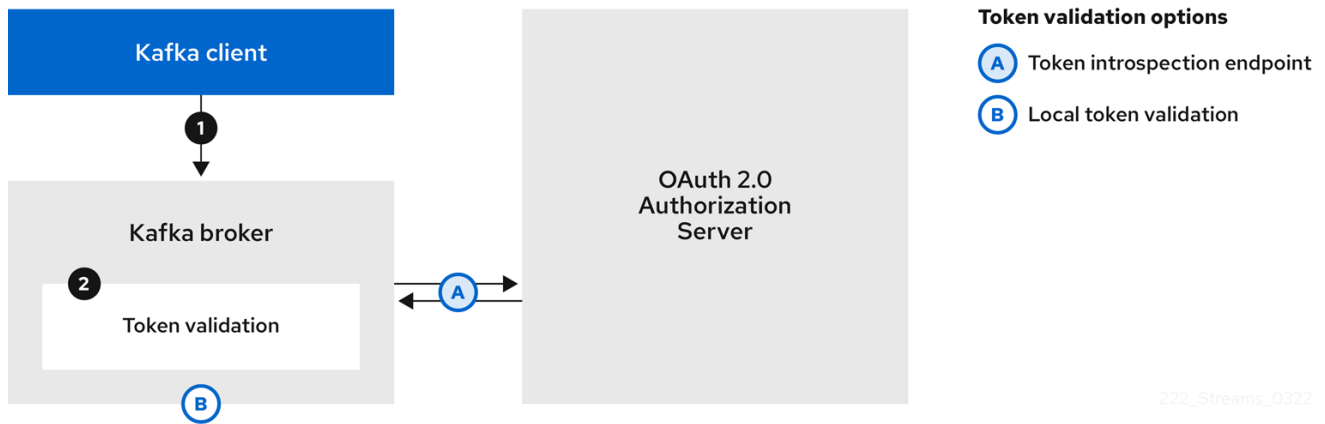
- クライアントがクライアント ID およびシークレットを使用し、ブローカーがクライアントのアクセストークンを取得する場合
- クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合

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** を承認サーバーに渡します。
3. 承認サーバーは、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 ブローカーは承認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、要求は承認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 ブローカーは承認サーバーでエンドポイント呼び出すことで、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、承認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。

5.4.9.6. OAuth 2.0 認証の設定

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

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

1. AMQ Streams クラスターおよび Kafka クライアントの OAuth 2.0 承認サーバーを設定します。
2. OAuth 2.0 を使用するように設定された 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
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。

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

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

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

AMQ Streams で OAuth 2.0 を使用するには、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

- AMQ Streams 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、AMQ Streams の OAuth 2.0 ポリシーを作成します。
ログインの詳細は、Red Hat Single Sign-On のデプロイ時に提供されます。
3. レルムを作成し、有効にします。
既存のマスターレルムを使用できます。
4. 必要に応じて、レルムのセッションおよびトークンのタイムアウトを調整します。
5. **kafka-broker** というクライアントを作成します。
6. **Settings** タブで以下を設定します。
 - **Access Type** を **Confidential** に設定します。

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

7. 続行する前に **Save** クリックします。

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

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

設定では、名前をクライアント ID として使用します。

次のステップ

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

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

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

TLS リスナーを設定して、暗号化されたインターフェイスで 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。

選択した承認サーバーをサポートするプロパティと、実装している承認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するプロパティの説明は、以下を参照してください。

- [OAuth 2.0 Kafka ブローカーの設定](#)

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 の承認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.oauthbearer.OAuthBearerLoginModule required ;
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
```

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

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

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

3. 必要な場合は、承認サーバーへのアクセスを設定します。
この手順は通常、サービスメッシュなどの技術がコンテナ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. セキュアな承認サーバーに接続するためのカスタムトラストストアを指定します。承認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、承認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応じて、追加のプロパティを設定します。

```

listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
# ...
oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token" \ 1
oauth.custom.claim.check="@.custom == 'custom-value'" \ 2
oauth.scope="SCOPE" \ 3
oauth.check.audience="true" \ 4
oauth.audience="AUDIENCE" \ 5
oauth.valid.issuer.uri="https://https://AUTH-SERVER-ADDRESS/auth/REALM-NAME" \ 6
oauth.client.id="kafka-broker" \ 7
oauth.client.secret="kafka-broker-secret" \ 8
oauth.refresh.token="REFRESH-TOKEN-FOR-KAFKA-BROKERS" \ 9
oauth.access.token="ACCESS-TOKEN-FOR-KAFKA-BROKERS" ; 10
oauth.connect.timeout.seconds=60 11
oauth.read.timeout.seconds=60 12
oauth.groups.claim="$groups" 13
oauth.groups.claim.delimiter="," 14

```

- 1 承認サーバーへの 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。KeycloakRBACAuthorizer を使用する場合は、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** を使った 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、承認サーバーに応じて、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。

- 4 (オプション) オーディエンスチェック。オーソリゼーションサーバーが **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したい場合
 - 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** を使った 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、承認サーバーに応じて、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** として承認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 **KeycloakRBACAuthorizer** 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) Kafka ブローカー用の長期間有効な更新トークン。
 - 10 (オプション) Kafka ブローカー用の長期間有効なアクセストークン。
 - 11 (オプション) 承認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。
 - 12 (オプション): 承認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。
 - 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承認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のに使用される区切り文字。デフォルト値は ,(コンマ) です。
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、設定設定を追加します。

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

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

- 2 承認サーバーは、通常ユーザーとクライアントの両方を識別する単一の属性を提供しない場合があります。クライアントが独自の名前で認証される場合、サーバーによって **クライ**
- 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、このプロパティを使用してクライアントのユーザー ID に接頭辞を追加します。
- 4 (**oauth.introspection.endpoint.uri** を使用する場合のみ該当): 使用している認証サーバーによっては、イントロスペクションエンドポイントによって **トークンタイプ** 属性が返されるかどうかは分からず、異なる値が含まれることがあります。イントロスペクションエンドポイントからの応答に含ま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せん。
- 5 (**oauth.introspection.endpoint.uri** を使用する場合のみ該当): インintrospection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、承認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。](#)

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

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

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

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている。
- 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>0.10.0.redhat-00014</version>
</dependency>
```


2. コールバックのシステムプロパティを設定します。
以下に例を示します。

```
System.setProperty(ClientConfig.OAUTH_TOKEN_ENDPOINT_URI, "https://<auth-server-address>/auth/realms/master/protocol/openid-connect/token"); ❶
System.setProperty(ClientConfig.OAUTH_CLIENT_ID, "<client_name>"); ❷
System.setProperty(ClientConfig.OAUTH_CLIENT_SECRET, "<client_secret>"); ❸
System.setProperty(ClientConfig.OAUTH_SCOPE, "<scope_value>"); ❹
System.setProperty(ClientConfig.OAUTH_AUDIENCE, "<audience_value>"); ❺
```

- ❶ 承認サーバーのトークンエンドポイントの URI です。
- ❷ クライアント ID。承認サーバーで **client** を作成するときに使用される名前です。
- ❸ 承認サーバーで **client** を作成するときに作成されるクライアントシークレット。
- ❹ (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**。認証サーバーでは、クライアントによるスコープの指定が必要になることがあります。
- ❺ (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**。認証サーバーでは、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。

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

Kafka クライアントの OAUTHBEARER の有効化

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

Kafka クライアントの PLAIN の有効化

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

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

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

5.4.10. OAuth 2.0 トークンベース承認の使用

トークンベースの認証に OAuth 2.0 と Red Hat Single Sign-On を使用している場合、Red Hat Single Sign-On を使用して承認ルールを設定し、Kafka ブローカーへのクライアントのアクセスを制限するこ

ともできます。認証はユーザーのアイデンティティを確立します。承認は、そのユーザーのアクセスレベルを決定します。

AMQ Streams は、Red Hat Single Sign-On の [認証サービス](#) 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ポリシーおよびパーミッションは、Kafka ブローカーのリソースへのアクセスを付与するために使用されます。ユーザーとクライアントは、Kafka ブローカーで特定のアクションを実行するためのアクセスを許可するポリシーに対して照合されます。

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

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

関連情報

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

5.4.10.1. OAuth 2.0 の承認メカニズム

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

5.4.10.1.1. Kafka ブローカーのカスタムオーソライザー

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

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

5.4.10.2. OAuth 2.0 承認サポートの設定

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

作業を開始する前に

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、承認ポリシーの設定方法を選択する必要がある場合があります。



注記

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

前提条件

- AMQ Streams は、[トークンベースの認証](#) に Red Hat Single Sign-On と OAuth 2.0 を使用するよう設定されている必要がある。承認を設定するときに、同じ Red Hat Single Sign-On サーバーエンドポイントを使用する必要があります。
- [Red Hat Single Sign-On のドキュメント](#) 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. 承認サービスを使用して、クライアントのリソース、承認スコープ、ポリシー、およびパーミッションを定義します。
3. ロールとグループをユーザーとクライアントに割り当てて、パーミッションをユーザーとクライアントにバインドします。
4. Red Hat Single Sign-On 承認を使用するように Kafka ブローカーを設定します。
以下を Kafka **server.properties** 設定ファイルに追加し、Kafka にオーソライザーをインストールします。

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

5. Kafka ブローカーの設定を追加して、承認サーバーおよび 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" ❶  
strimzi.authorization.client.id="kafka" ❷
```

- 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) 単純許可に委任します。
以下に例を示します。

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

- 1 Red Hat Single Sign-On 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 接続の設定を許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) 許可サーバーからの許可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" ❶
strimzi.authorization.grants.refresh.pool.size="10" ❷
```

- ❶ 許可サーバーからの許可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。
- ❷ 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。

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

5.4.11. OPA ポリシーベースの承認の使用

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

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



注記

Red Hat は OPA サーバーをサポートしません。

関連情報

- [Open Policy Agent の Web サイト](#)

5.4.11.1. OPA ポリシーの定義

OPA と AMQ Streams 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承認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。

5.4.11.2. OPA への接続

Kafka が OPA ポリシーエンジンにアクセスしてアクセス制御ポリシーをクエリーできるようにするには、Kafka **server.properties** ファイルでカスタム OPA オーソライザープラグイン (**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。

5.4.11.3. 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

- 接続には OPA サーバーを利用できる必要がある。
- [Kafka の OPA オーソライザープラグイン](#)。

手順

1. Kafka ブローカーで操作を実行するため、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 オーソライザープラグイン](#) をインストールします。
[OPA への接続](#) を参照してください。

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

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

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


4. Kafka ブローカーの **server.properties** にさらに設定を追加して、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) オーソライザープラグインがクエリーするポリシーの OAuth 2.0 トークンエンドポイント URL。この例では、ポリシーは **allow** という名前です。
- 2** オーソライザープラグインが OPA ポリシーエンジン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。

認証および承認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。

5.4.12. ロギング

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

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

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

5.4.12.1. Kafka ブローカーロガーのロギングレベルの動的な変更

Kafka ブローカーロギングは、複数の **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、ブローカーロガーのロギングレベルを動的に変更できます。ログで返される詳細度レベルを上げると (たとえば、**INFO** から **DEBUG** に変更)、Kafka クラスターでパフォー

マンスの問題を調査するのに役立ちます。

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

前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [ZooKeeper](#) および [Kafka](#) が稼働している。

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一覧を表示します。

```
/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** の各ロガーのロギン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ギン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ギングレベルにリセットできます。**--alter** および **--delete-config** オプションを使用して、各ブローカーロガーを二重引用符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](#)

第6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。

6.1. パーティションおよびレプリカ

Kafka のメッセージは、常にトピックとの間で送受信されます。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。メッセージの異なるパーティションへのシャーディングにより、トピックを簡単に水平的にスケーリングできます。

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複製します。

リーダーが失敗すると、フォロワーの1つが新しいリーダーに自動的になります。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。

6.2. 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。
- 解析に 1GB のメッセージが含まれるまで。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 1GB 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。

さらに、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。

圧縮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。圧縮は定期的に行われるため、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。

6.3. トピックの自動作成

プロデューサーまたはコンシューマーが存在しないトピックとの間でメッセージを送受信しようとする、Kafka はデフォルトでそのトピックを自動的に作成します。この動作は、デフォルトで **true** に設定された **auto.create.topics.enable** 設定プロパティによって制御されます。

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

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

6.4. トピックの削除

Kafka では、トピックの削除を無効にすることができます。これは、デフォルトで **true** (つまり、トピックの削除が可能) に設定されている **delete.topic.enable** プロパティで設定されます。このプロパティを **false** に設定すると、トピックの削除はできず、トピックの削除試行はすべて成功を返しますが、トピックは削除されません。

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

6.5. 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** ポリシーはログの圧縮を有効にします。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 **max.message.bytes** を参照してください。

min.insync.replicas

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

log.retention.ms

上記の **retention.ms** を参照してください。

log.retention.bytes

上記の **retention.bytes** を参照してください。

log.segment.bytes

上記の **segment.bytes** を参照してください。

default.replication.factor

自動的に作成されるトピックのデフォルト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。

6.6. 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 には複数の内部トピックがあります。これらはコンシューマーオフセット (**__consumer_offsets**) またはトランザクションの状態 (**__transaction_state**) を格納するために使用されます。これらのトピックは、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` のトピックのレプリカ数デフォルト値は **3** です。

`transaction.state.log.num.partitions`

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

`transaction.state.log.min.isr`

`__transaction_state` トピックへの書き込みが正常であると考えられるために、確認される必要のあるレプリカの最小数。この最小値が満たされない場合、プロデューサーは例外で失敗します。デフォルト値は **2** です。

6.7. 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 AMQ Streams ディストリビューションの一部で、**bin** ディレクトリにあります。

前提条件

- AMQ Streams クラスターがインストールされ、実行されている。

トピックの作成

1. **kafka-topics.sh** ユーティリティーを使用し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
```

関連情報

- [トピックの設定](#)

6.8. トピックの一覧表示および説明

kafka-topics.sh ツールは、トピックの一覧表示および説明に使用できます。**kafka-topics.sh** は AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

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

トピックの記述

1. **kafka-topics.sh** ユーティリティーを使用し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_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
```

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一覧表示します。また、すべてのトピック設定オプションも表示されます。

関連情報

- [トピックの設定](#)
- [トピックの作成](#)

6.9. 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 AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

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

トピック設定の変更

1. **kafka-configs.sh** ツールを使用して、現在の設定を取得します。

- **--bootstrap-server** オプションで Kafka ブローカーのホストおよびポートを指定します。
- **--entity-type** を **topic** 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_address> --entity-type topics  
--entity-name <TopicName> --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のに使用します。
- **--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_address> --entity-type topics  
--entity-name <TopicName> --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のに使用します。
- **--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_address> --entity-type topics  
--entity-name <TopicName> --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
```

関連情報

- [トピックの設定](#)
- [トピックの作成](#)

6.10. 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 AMQ Streams ディストリビューションの一部で、**bin** ディレクトリーにあります。

前提条件

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

トピックの削除

1. **kafka-topics.sh** ユーティリティーを使用してトピックを削除します。
 - **--bootstrap-server** 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のに使用します。
 - **--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_address> --delete --topic <TopicName>
```

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_address> --list
```

すべてのトピックを一覧表示するコマンドの例

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

関連情報

- [トピックの作成](#)

第7章 KAFKA CONNECT での AMQ STREAMS 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、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プラグインを使用します。コネクタプラグインには、シンクとソースの2つのタイプがあります。シンクコネクタは、Kafka から外部システムにデータをストリーミングします。ソースコネクタは、外部システムから Kafka にデータをストリーミングします。

Kafka Connect はスタンドアロンまたは分散モードで実行できます。

スタンドアロンモード

スタンドアロンモードでは、Kafka Connect はプロパティファイルから読み込んだユーザー定義の設定を持つ単一ノードで実行されます。

分散モード

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。コネクタとその設定は、HTTP REST インターフェイスを使用して管理します。

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、[9章 大量のメッセージ処理](#)を参照してください。

7.1. スタンドアロンモードでの KAFKA CONNECT

スタンドアロンモードでは、Kafka Connect は単一ノードで単一のプロセスとして実行されます。スタンドアロンモードの設定は、プロパティファイルを使用して管理します。

7.1.1. 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。

設定ファイルの例は、**config/connect-standalone.properties** のインストールディレクトリーにあります。

コネクタプラグインは、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。

7.1.2. スタンドアロンモードでの Kafka Connect でのコネクタの設定

プロパティファイルを使用すると、スタンドアロンモードで Kafka Connect のコネクタプラグインを設定できます。ほとんどの設定オプションは、各コネクタに固有のものです。以下のオプションはすべてのコネクタに適用されます。

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の名前。

connector.class

コネクタプラグインのクラス。たとえば、**org.apache.kafka.connect.file.FileStreamSinkConnector** です。

tasks.max

指定のコネクタが使用できるタスクの最大数。タスクにより、コネクタは並行して作業を実行できます。コネクタは、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。

さらに、シンクコネクタには以下のオプションの1つ以上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。

その他のオプションについては、利用可能なコネクタのドキュメントを参照してください。

AMQ Streams には、コネクタ設定ファイルの例が含まれています。AMQ Streams のインストール・ディレクトリーにある **config/connect-file-sink.properties** および **config/connect-file-source.properties** を参照してください。

7.1.3. スタンドアロンモードでの Kafka Connect の実行

この手順では、スタンドアロンモードで Kafka Connect を設定し、実行する方法を説明します。

前提条件

- インストールされ、実行されている AMQ Streams クラスター。

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. KafkaConnect が実行されていることを確認します。

```
jcmod | grep ConnectStandalone
```

7.2. 分散モードでの KAFKA CONNECT

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。コネクタプラグインとその設定は、HTTP REST インターフェイスを使用して管理します。

7.2.1. 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。

AMQ Streams には、分散モードの Kafka Connect の設定ファイル例が含まれています。AMQ Streams のインストールディレクトリーにある **config/connect-distributed.properties** を参照してください。

コネクタプラグインは、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。

7.2.2. 分散 Kafka Connect でのコネクタの設定

HTTP REST インターフェイス

分散 Kafka Connect のコネクタは、HTTP REST インターフェイスを使用して設定されます。REST インターフェイスはデフォルトで 8083 番ポートをリッスンします。以下のエンドポイントをサポートします。

GET /connectors

既存のコネクタのリストを返します。

POST /connectors

コネクタを作成します。リクエストボディは、コネクタ設定が含まれる JSON オブジェクトである必要があります。

GET /connectors/<name>

特定のコネクタの情報を取得します。

GET /connectors/<name>/config

特定のコネクタの設定を取得します。

PUT /connectors/<name>/config

特定のコネクタの設定を更新します。

GET /connectors/<name>/status

特定のコネクタのステータスを取得します。

PUT /connectors/<name>/pause

コネクタとそのすべてのタスクを一時停止します。コネクタはメッセージの処理を停止します。

PUT /connectors/<name>/resume

一時停止されたコネクタを再開します。

POST /connectors/<name>/restart

コネクタに障害が発生した場合に、コネクタを再起動します。

DELETE /connectors/<name>

コネクタを削除します。

GET /connector-plugins

サポートされるすべてのコネクタプラグインのリストを取得します。

コネクタ設定

ほとんどの設定オプションはコネクタ固有で、コネクタのドキュメントに含まれています。以下のフィールドは、すべてのコネクタで共通しています。

name

コネクタの名前。特定の Kafka Connect インスタンス内で一意である必要があります。

connector.class

コネクタプラグインのクラス。たとえば、**org.apache.kafka.connect.file.FileStreamSinkConnector** です。

tasks.max

このコネクタによって使用されるタスクの最大数。タスクは、コネクタが作業を並列処理するために使用します。コネクタは、指定された数よりも少ないタスクを作成する場合があります。

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。

さらに、シンクコネクタには、以下のオプションの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。

その他のオプションについては、特定のコネクタのドキュメントを参照してください。

AMQ Streams には、コネクタ設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/connect-file-sink.properties** および **config/connect-file-source.properties** にあります。

7.2.3. 分散 Kafka Connect の実行

この手順では、Kafka Connect を分散モードで設定および実行する方法を説明します。

前提条件

- インストールされ、実行されている AMQ Streams クラスター。

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. KafkaConnect が実行されていることを確認します。

```
jcmd | grep ConnectDistributed
```

7.2.4. コネクターの作成

この手順では、Kafka Connect REST API を使用して分散モードで Kafka Connect で使用するコネクタプラグインを作成する方法を説明します。

前提条件

- 分散モードで実行する Kafka Connect インストール。

手順

1. コネクタ設定で 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
```

7.2.5. コネクターの削除

この手順では、Kafka Connect REST API を使用して分散モードの Kafka Connect からコネクタプラグインを削除する方法を説明します。

前提条件

- 分散モードで実行する Kafka Connect インストール。

コネクタの削除

1. **<KafkaConnectAddress>:8083/connectors/<ConnectorName>** に **GET** リクエストを送信して、コネクタ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
```

7.3. コネクタプラグイン

AMQ Streams には以下のコネクタプラグインが含まれています。

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。

必要に応じて、さらにコネクタプラグインを追加できます。Kafka Connect は起動時に、追加のコネクタプラグインを検索し、実行します。Kafka Connect がプラグインを検索するパスを定義するには、**plugin.path configuration** オプションを設定します。

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

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

Kafka Connect を分散モードで実行する場合、プラグインはすべてのワーカーノードで利用可能でなければなりません。

7.4. コネクタプラグインの追加

この手順では、コネクタプラグインを追加する方法を説明します。

前提条件

- インストールされ、実行されている AMQ Streams クラスター。

手順

1. **/opt/kafka/connector-plugins** ディレクトリーを作成します。

```
su - kafka  
mkdir /opt/kafka/connector-plugins
```

2. **/opt/kafka/config/connect-standalone.properties** または **/opt/kafka/config/connect-distributed.properties** Kafka Connect 設定ファイルを編集し、**plugin.path** オプションを **/opt/kafka/connector-plugins** に設定します。以下に例を示します。

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

3. コネクタープラグインを **/opt/kafka/connector-plugins** にコピーします。
4. Kafka Connect ワーカーを起動または再起動します。

第8章 AMQ STREAMS の MIRRORMAKER 2.0 との使用

MirrorMaker 2.0 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複製します。

クラスター全体のデータレプリケーションでは、以下が必要な状況がサポートされます。

- システム障害時のデータの復旧
- 分析用のデータの集計
- 特定のクラスターへのデータアクセスの制限
- レイテンシーを改善するための特定場所でのデータのプロビジョニング

MirrorMaker 2.0 を設定するには、**config/connect-mirror-maker.properties** 設定ファイルを編集します。

必要な場合は、[MirrorMaker 2.0 の分散トレースを有効化](#) 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、[9章 大量のメッセージ処理](#) を参照してください。



注記

MirrorMaker 2.0 には、以前のバージョンの MirrorMaker ではサポートされない機能があります。ただし、[MirrorMaker 2.0 をレガシーモードで使用されるように設定](#) できます。

8.1. MIRRORMAKER 2.0 データレプリケーション

MirrorMaker 2.0 はソースの Kafka クラスターからメッセージを消費して、ターゲットの Kafka クラスターに書き込みます。

MirrorMaker 2.0 は以下を使用します。

- ソースクラスターからデータを消費するソースクラスターの設定
- データをターゲットクラスターに出力するターゲットクラスターの設定

MirrorMaker 2.0 は Kafka Connect フレームワークをベースとし、**コネクタ** によってクラスター間のデータ転送が管理されます。

MirrorMaker 2.0 は次のコネクタを使用します。

MirrorSourceConnector

ソースコネクタは、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複製します。

MirrorCheckpointConnector

チェックポイントコネクタ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。

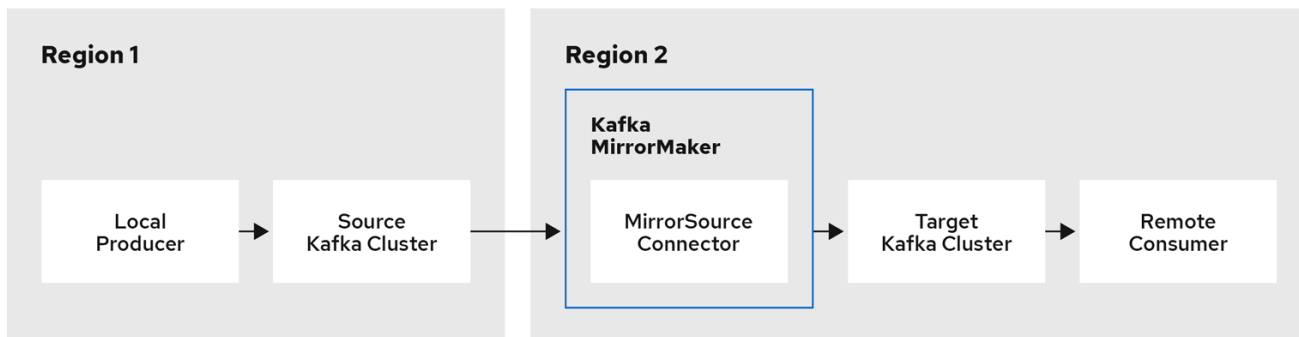
MirrorHeartbeatConnector

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。

あるクラスターから別のクラスターにデータをミラーリングするプロセスは非同期です。推奨されるパターンは、ソース Kafka クラスターとともにローカルでメッセージが作成され、ターゲットの Kafka クラスターの近くでリモートで消費されることです。

MirrorMaker 2.0 は、複数のソースクラスターで使用できます。

図8.12 つのクラスターにおけるレプリケーション



222_Streams_0322

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** をソースコネクタ設定に追加することで変更できます。ただし、操作の頻度が増えると、全体的なパフォーマンスに影響する可能性があります。

8.2. クラスター設定

active/passive または **active/active** クラスター設定で MirrorMaker 2.0 を使用できます。

- **active/active** 設定では、両方のクラスターがアクティブで、同じデータを同時に提供します。これは、地理的に異なる場所で同じデータをローカルで利用可能にする場合に便利です。
- **active/passive** 設定では、アクティブなクラスターからのデータはパッシブなクラスターで複製され、たとえば、システム障害時のデータ復旧などでスタンバイ状態を維持します。

プロデューサーとコンシューマーがアクティブなクラスターのみに接続することを前提とします。

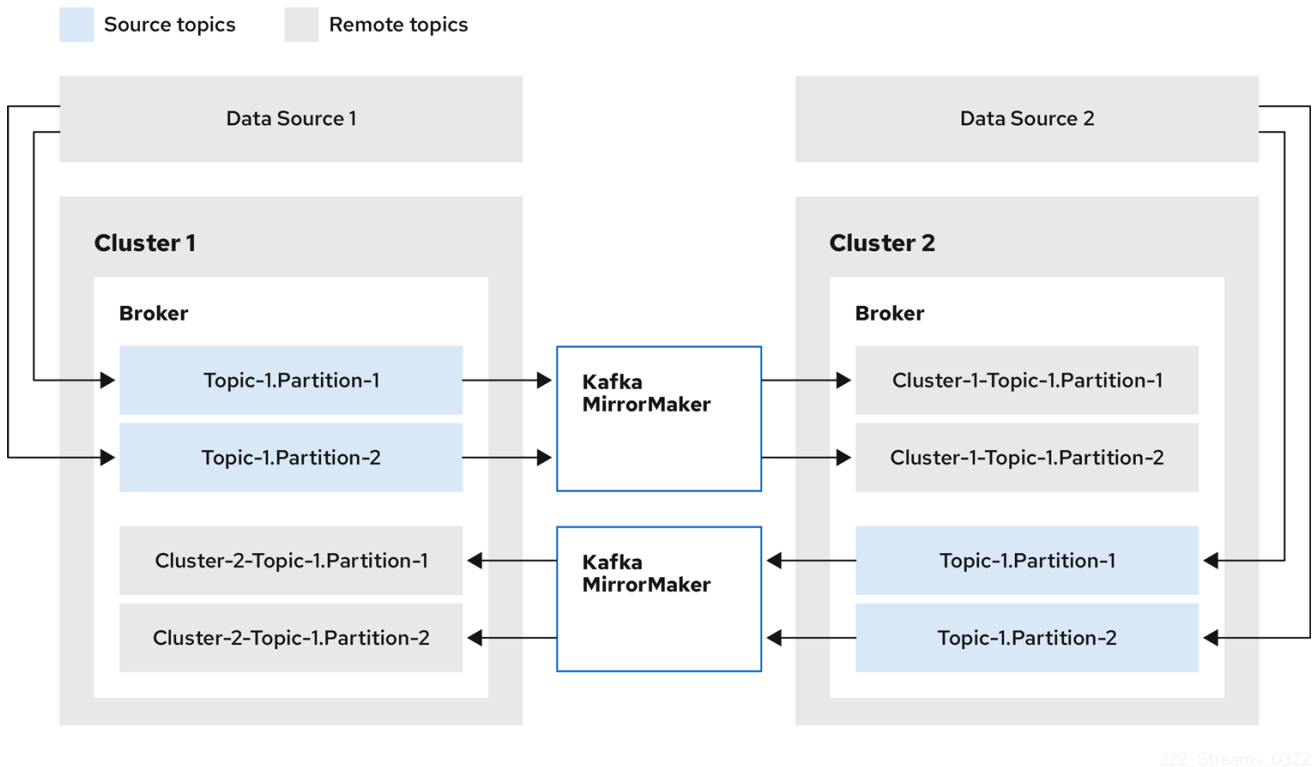
MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

8.2.1. 双方向レプリケーション (active/active)

MirrorMaker 2.0 アーキテクチャーでは、**active/active** クラスター設定で双方向レプリケーションがサポートされます。

各クラスターは、**source** および **remote** トピックの概念を使用して、別のクラスターのデータを複製します。同じトピックが各クラスターに保存されるため、リモートトピックの名前がソースクラスターを表すように自動的に MirrorMaker 2.0 によって変更されます。元のクラスターの名前の先頭には、トピックの名前が追加されます。

図8.2 トピック名の変更



222_Streams_0322

ソースクラスターにフラグを付けると、トピックはそのクラスターに複製されません。

remote トピックを介したレプリケーションの概念は、データの集約が必要なアーキテクチャーの設定に役立ちます。コンシューマーは、同じクラスター内でソースおよびリモートトピックにサブスクライブできます。これに個別の集約クラスターは必要ありません。

8.2.2. 一方向レプリケーション (active/passive)

MirrorMaker 2.0 アーキテクチャーでは、**active/passive** クラスター設定で一方向レプリケーションがサポートされます。

active/passive のクラスター設定を使用してバックアップを作成したり、データを別のクラスターに移行したりできます。この場合、リモートトピックの名前を自動的に変更したくないことがあります。

IdentityReplicationPolicy を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。

8.2.3. トピック設定の同期

トピック設定は、ソースクラスターとターゲットクラスター間で自動的に同期化されます。設定プロパティを同期化することで、リバランスの必要性が軽減されます。

8.2.4. データの整合性

MirrorMaker 2.0 は、ソーストピックを監視し、設定変更をリモートトピックに伝播して、不足しているパーティションを確認および作成します。MirrorMaker 2.0 のみがリモートトピックに書き込みできます。

8.2.5. オフセットの追跡

MirrorMaker 2.0 では、内部トピックを使用してコンシューマーグループのオフセットを追跡します。

- **offset-syncs** トピックは、複製されたトピックパーティションのソースおよびターゲットオフセットをレコードメタデータからマッピングします。
- **チェックポイント** トピックは、各コンシューマーグループで複製されたトピックパーティションのソースおよびターゲットクラスターで、最後にコミットされたオフセットをマッピングします。

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。**チェックポイント** トピックのオフセットは、設定によって事前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しかない場合でも、MirrorMaker 2.0 を使用できます。

8.2.6. コンシューマーグループオフセットの同期

__consumer_offsets トピックには、各コンシューマーグループのコミットされたオフセットに関する情報が保存されます。オフセットの同期は、ソースクラスターのコンシューマーグループのコンシューマーオフセットをターゲットクラスターのコンシューマーオフセットに定期的に転送します。

オフセットの同期は、特に **active/passive** 設定で便利です。アクティブなクラスターがダウンした場合、コンシューマーアプリケーションはパッシブ (スタンバイ) クラスターに切り替え、最後に転送されたオフセットの位置からピックアップできます。

トピックオフセットの同期を使用するには、**sync.group.offsets.enabled** を checkpoint コネクター設定に追加し、プロパティを **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。両方のプロパティのデフォルトは 60 秒です。**refresh.groups.interval.seconds** プロパティを使用して、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パッシブクラスターへ切り替えられると、一部のメッセージが重複する可能性があります。

8.2.7. 接続性チェック

MirrorHeartbeatConnector は **heartbeat** を発行して、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。ターゲットクラスターは、**heartbeat** トピックを使用して次のことを確認します。

- クラスター間の接続を管理するコネクタが稼働しているかどうか
- ソースクラスターが利用可能かどうか

8.3. コネクタ設定

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の Mirrormaker 2.0 コネクタ設定を使用します。

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

表8.1 MirrorMaker 2.0 コネクタ設定プロパティ

プロパティ	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。デフォルトは . (ドット) です。 replication.policy.class が DefaultReplicationPolicy の場合にのみ使用されます。	✓	✓	✓
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	sourceConnector	checkpointConnector	heartbeatConnector
topic.filter.class 複製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 複製するトピック設定プロパティ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 複製すべきでないトピック設定プロパティ。コンマ区切りのプロパティ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。	✓		
refresh.topics.interval.seconds トピック更新の頻度。デフォルトは 600 (10 分) です。	✓		
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。	✓		

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
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。User Operator との互換性はありません。	✓		
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。		✓	
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 複製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。		✓	

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
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。		✓	
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。			✓

8.4. コネクタプロデューサーおよびコンシューマーの設定

MirrorMaker 2.0 コネクタは内部プロデューサーおよび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

プロデューサーとコンシューマーの設定オプションは、MirrorMaker 2.0 の実装に依存し、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
```

8.5. 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トするとき、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ト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ト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 2.0 は新しいコンシューマーグループを 10 分ごとにチェックします。**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。

8.6. 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、このようリモートトピックへのアクセスをサポートしません。

8.7. MIRRORMAKER 2.0 を専用モードで実行する

MirrorMaker 2.0 を使用して、設定を介して Kafka クラスター間のデータを同期します。この手順では、専用の単一ノード MirrorMaker 2.0 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。現在、専用モードの MirrorMaker 2.0 は単一のワーカーノードでのみ動作します。



注記

MirrorMaker 2.0 を分散モードで実行することもできます。分散モードでは、MirrorMaker 2.0 は Kafka Connect クラスターでコネクターとして実行されます。Kafka はデータレプリケーションに MirrorMaker ソースコネクターを提供します。専用の MirrorMaker クラスターを実行する代わりにコネクターを使用する場合は、Kafka Connect クラスターでコネクターを設定する必要があります。詳細は、[Apache Kafka のドキュメント](#) を参照してください。

以前のバージョンの MirrorMaker は、[レガシーモードで MirrorMaker 2.0 を実行](#) することにより、引き続きサポートされます。

設定では以下を指定する必要があります。

- 各 Kafka クラスター
- TLS 認証を含む各クラスターの接続情報
- レプリケーションのフローおよび方向
 - クラスター対クラスター

- トピック対トピック
- レプリケーションルール
- コミットされたオフセット追跡間隔

この手順では、プロパティファイルで設定を作成し、MirrorMaker スクリプトファイルを使用して接続を設定する際にプロパティを渡し、MirrorMaker 2.0 を実装する方法を説明します。

ソースクラスターから複製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、複製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複製する設定例

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

名前の一覧を指定したり、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複製されます。.*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複製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

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

前提条件

- 複製している各 Kafka クラスターノードのホストに AMQ Streams がインストールされている必要がある。

手順

1. テキストエディターでサンプルプロパティファイルを開くか、新しいプロパティファイルを作成し、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** プロパティで設定できます。

MirrorMaker 2.0 の設定例

```
clusters=cluster-1,cluster-2 ❶

cluster-1.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_one>:443 ❷
cluster-1.security.protocol=SSL ❸
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**. プロパティは、**cluster-1** の TLS 設定を定義します。
- 4 **cluster-2** の接続情報。
- 5 **ssl**. プロパティは、**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。ソースコネクタは指定のトピックを複製します。チェックポイントコネクタは、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は、指定されたコンシューマーグループを複製します。
- 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。
- 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。

- 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
```

- 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 ❶
sync.group.offsets.interval.seconds=60 ❷
emit.checkpoints.interval.seconds=60 ❸
```

- ❶ 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。
- ❷ 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。
- ❸ オフセット追跡のチェック頻度を調整します。オフセット同期の頻度を変更する場合、これらのチェックの頻度も調整する必要がある場合があります。

- ターゲットクラスターで ZooKeeper および Kafka を起動します。

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

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

- プロパティ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 /config/connect-mirror-maker.properties
```

MirrorMaker はクラスター間の接続を設定します。

- ターゲットクラスターごとに、トピックが複製されていることを確認します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

8.8. レガシーモードでの MIRRORMAKER 2.0 の使用

この手順では、MirrorMaker 2.0 をレガシーモードで使用する設定方法を説明します。レガシーモードは、以前のバージョンの MirrorMaker をサポートします。

MirrorMaker スクリプト `/opt/kafka/bin/kafka-mirror-maker.sh` は、レガシーモードで MirrorMaker 2.0 を実行できます。



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。その結果、Kafka MirrorMaker 1 は AMQ Streams 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 AMQ Streams から削除されます。代わりに、**IdentityReplicationPolicy** で MirrorMaker 2.0 を使用します。

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ファイルが必要。

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

手順

1. MirrorMaker の **consumer.properties** と **producer.properties** ファイルを編集して、MirrorMaker 2.0 の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 ❶

refresh.topics.enabled=false ❷
refresh.groups.enabled=false
emit.checkpoints.enabled=false
emit.heartbeats.enabled=false
sync.topic.configs.enabled=false
sync.topic.acls.enabled=false
```

- ❶ MirrorMaker の以前のバージョンをエミュレートします。
- ❷ 内部 チェックポイント や ハートビート トピックなど、MirrorMaker 2.0 の機能が無効になりました。

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 プロパティはソースクラスターの設定を提供し、**producer** プロパティ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 ZooKeeper および Kafka を起動します。

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

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

4. ターゲットクラスターごとに、トピックが複製されていることを確認します。

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

第9章 大量のメッセージ処理

AMQ Streams デプロイメントで大量のメッセージを処理する必要がある場合は、設定オプションを使用してスループットとレイテンシーを最適化できます。

Kafka プロデューサーおよびコンシューマー設定は、Kafka ブローカーへのリクエストのサイズおよび頻度を制御するのに役立ちます。設定オプションの詳細は、以下を参照してください。

- [プロデューサーのスループットおよびレイテンシーの最適化](#)
- [スループットおよびレイテンシーに対するコンシューマーの最適化](#)

Kafka Connect ランタイムソースコネクタ (MirrorMaker 2.0 を含む) とシンクコネクタで使用されるプロデューサーとコンシューマーで同じ設定オプションを使用することもできます。

ソースコネクタ

- Kafka Connect ランタイムのプロデューサーは、メッセージを Kafka クラスターに送信します。
- MirrorMaker 2.0 の場合、ソースシステムが Kafka であるため、コンシューマーはソース Kafka クラスターからメッセージを取得します。

シンクコネクタ

- Kafka Connect ランタイムのコンシューマーは、Kafka クラスターからメッセージを取得します。

コンシューマー設定 (**consumer.***) の場合、1回のフェッチ要求でフェッチされるデータの量を増やして、レイテンシーを減らすことができます。**fetch.max.bytes** および **max.partition.fetch.bytes** プロパティを使用して、フェッチ要求のサイズを増やします。**max.poll.records** プロパティを使用して、コンシューマーバッファから返されるメッセージ数の上限を設定することもできます。

プロデューサー設定 (**Producer.***) の場合、1回のプロデューサーリクエストで送信されるメッセージバッチのサイズを大きくすることができます。**batch.size** プロパティを使用してバッチサイズを増やします。バッチサイズを大きくすると、送信する準備ができていない未処理のメッセージの数と、メッセージキュー内のバックログのサイズが減少します。同じパーティションに送信されるメッセージはまとめてバッチ処理されます。バッチサイズに達すると、プロデューサーリクエストがターゲットクラスターに送信されます。バッチサイズを大きくすると、プロデューサーリクエストが遅延し、より多くのメッセージがバッチに追加され、同時にブローカーに送信されます。これにより、多数のメッセージを処理するトピックパーティションが複数ある場合に、スループットが向上します。

プロデューサーが適切なプロデューサーバッチサイズに対して処理するレコードの数とサイズを考慮します。

linger.ms を使用してミリ秒単位の待機時間を追加し、プロデューサーの負荷が減少したときにプロデューサーリクエストを遅らせます。遅延は、最大バッチサイズ未満の場合に、バッチにより多くのレコードをバッチに追加できることを意味します。

Kafka Connect ソースコネクタでは、ターゲット Kafka クラスターへのデータストリーミングパイプラインは以下ようになります。

Kafka Connect ソースコネクタのデータストリーミングパイプライン

外部データソース → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバッファ → ターゲット Kafka トピック

Kafka Connect シンクコネクタの場合、ターゲット外部データソースへのデータストリーミングパイプラインは次のとおりです。

Kafka Connect シンクコネクタのデータストリーミングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) シンクメッセージキュー → コンシューマーバッファ → 外部データソース

MirrorMaker 2.0 の場合、ターゲット Kafka クラスターへのデータミラーリングパイプラインは次のとおりです。

MirrorMaker 2.0 のデータミラーリングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバッファ → ターゲット Kafka トピック

プロデューサーは、バッファ内のメッセージをターゲット Kafka クラスター内のトピックに送信します。これが発生している間、Kafka Connect タスクは引き続きデータソースをポーリングして、ソースメッセージキューにメッセージを追加します。

ソースコネクタのプロデューサーバッファのサイズは、**buffer.memory** プロパティを使用して設定されます。タスクは、バッファがフラッシュされる前に、指定されたタイムアウト期間 (**offset.flush.timeout.ms**) 待機します。これは、送信されたメッセージがブローカーによって確認され、コミットされたデータがオフセットされるのに十分な時間です。ソースタスクは、シャットダウン中を除き、オフセットをコミットする前にプロデューサーがメッセージキューを空にするのを待ちません。

プロデューサーがソースメッセージキュー内のメッセージのスループットについていけない場合、バッファリングは、**max.block.ms** で制限された期間内にバッファに使用可能なスペースができるまでブロックされます。バッファ内に未確認のメッセージがあれば、この期間中に送信されます。これらのメッセージが確認されてフラッシュされるまで、新しいメッセージはバッファに追加されません。

次の設定変更を試して、未処理メッセージの基になるソースメッセージキューを管理可能なサイズに保つことができます。

- **offset.flush.timeout.ms** のデフォルト値 (ミリ秒) を増やす
- 十分な CPU およびメモリーリソースがあることの確認
- 以下を実行して、並行して実行されるタスクの数を増やします。
 - **tasks.max** プロパティを使用して並列実行するタスクの数を増やす
 - タスクを実行するワーカーのノード数を増やす

使用可能な CPU とメモリーリソース、およびワーカーノードの数に応じて、並列実行できるタスクの数を検討してください。必要な効果が得られるまで、設定値を調整し続ける必要がある場合があります。

9.1. 大量メッセージ用の KAFKA CONNECT の設定

Kafka Connect は、ソースの外部データシステムからデータをフェッチし、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケートします。

次の例は、Kafka Connect ソースコネクタの設定を示しています。

大量のメッセージを処理するためのソースコネクタの設定例

```
# ...
producer.batch.size=327680
producer.linger.ms=100
# ...
tasks.max = 2
```

シンクコネクターのコンシューマー設定が追加されます。

大量のメッセージを処理するためのシンクコネクターの設定例

```
# ...
consumer.fetch.max.bytes=52428800
consumer.max.partition.fetch.bytes=1048576
consumer.max.poll.records=500
# ...
tasks.max = 2
```

9.2. 大量のメッセージ用に MIRRORMAKER 2.0 を設定する

MirrorMaker 2.0 は、ソースクラスターからデータをフェッチし、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケートします。

次の例は、MirrorMaker 2.0 の設定を示しています。この設定は、ソースからメッセージをフェッチするコンシューマーと、メッセージをターゲット Kafka クラスターに送信するプロデューサーに関連しています。

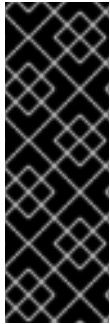
大量のメッセージを処理するための MirrorMaker 2.0 の設定例

```
clusters=cluster-1,cluster-2
# ...
cluster-2.producer.batch.size=327680
cluster-2.producer.linger.ms=100
cluster-1.consumer.fetch.max.bytes=52428800
cluster-1.consumer.max.partition.fetch.bytes=1048576
cluster-1.consumer.max.poll.records=500
# ...
tasks.max = 2
```

第10章 KAFKA の管理

追加の設定プロパティを使用して、AMQ Streams のデプロイメントを維持します。AMQ Streams のパフォーマンスに対応するため、設定を追加および調整できます。たとえば、スループットやデータの信頼性を向上させるために追加の設定を導入できます。

10.1. 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 の実稼働サービスレベルアグリーメント (SLA) でサポートされておらず、機能的に完全でない可能性があります。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 1TB ディスクを使用し、クォータが 1.1TB 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

- Kafka ブローカーとして使用されるすべてのホストに [AMQ Streams がインストールされている](#)。
- ZooKeeper クラスターが [設定され、稼働している](#)。

手順

1. Kafka 設定ファイル `/opt/kafka/config/server.properties` を編集します。プラグインプロパティは、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 ❶
client.quota.callback.static.produce=1000000 ❷
client.quota.callback.static.fetch=1000000 ❸
client.quota.callback.static.storage.soft=400000000000 ❹
client.quota.callback.static.storage.hard=500000000000 ❺
client.quota.callback.static.storage.check-interval=5 ❻
# ...
```

- ❶ Kafka Static Quota プラグインを読み込みます。
- ❷ 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。
- ❸ 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。
- ❹ 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。
- ❺ 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。
- ❻ 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/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
```

10.2. KAFKA クラスターのスケーリング

Kafka クラスターからブローカーを追加または削除できます。ZooKeeper クラスターからノードを追加または削除することもできます。

ブローカーを追加または削除する場合は、**kafka-reassign-partitions.sh** を使用してパーティションを割り当てることができます。

Cruise Control を使用して、Kafka クラスターのリバランス時にブローカーの数に変更を組み込むこともできます。新しいブローカーをインストールして、リバランスに追加できます。削除する前に除外するブローカーを指定して理バラナスを実行できます。詳細は、[14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。

10.2.1. Kafka クラスターへのブローカーの追加および削除

トピックのスループットを向上させる主な方法は、そのトピックのパーティション数を増やすことです。パーティションによってクラスター内のブローカー間でそのトピックの負荷が共有できます。ブローカーが何らかのリソース (通常は I/O) によって制約されている場合、より多くのパーティションを

使用してもスループットは向上しません。代わりに、クラスターにブローカーを追加する必要があります。

追加のブローカーをクラスターに追加する場合、AMQ Streams ではパーティションは自動的に割り当てられません。既存のブローカーから新しいブローカーに移動するパーティションを決定する必要があります。すべてのブローカー間でパーティションが再分散されたら、各ブローカーのリソース使用率が低下するはずです。

クラスターからブローカーを削除する前に、そのブローカーにパーティションが割り当てられていないことを確認します。使用を停止するブローカーの各パーティションに対応する残りのブローカーを決定する必要があります。ブローカーに割り当てられたパーティションがない場合は、ブローカーを停止することができます。

10.2.2. パーティションの再割り当て

kafka-reassign-partitions.sh は、パーティションを別のブローカーに再割り当てする際に使います。

これには、以下の3つのモードがあります。

--generate

トピックとブローカーのセットを取得し、**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。**再割り当て JSON ファイル** を生成する簡単な方法です。ただし、トピック全体で機能するため、その使用が常に適切であるとは限りません。

--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。パーティションを取得するブローカーは、パーティションリーダーのフォロワーになります。特定のパーティションでは、新規ブローカーが ISR 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確認します。再割り当てが完了すると、有効な **スロットル** も削除されます。スロットルを削除しないと、再割り当てが完了した後もクラスターは影響を受け続けます。

クラスターでは、1度に1つの再割り当てのみを実行でき、実行中の再割り当てをキャンセルすることはできません。再割り当てをキャンセルする必要がある場合は、割り当てが完了するのを待ってから別の再割り当てを実行し、最初の再割り当ての結果を元に戻します。**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。

10.2.2.1. 再割り当て JSON ファイル

再割り当て JSON ファイル には特定の構造があります。

```
{
  "version": 1,
  "partitions": [
    <PartitionObjects>
  ]
}
```

ここで **<PartitionObjects>** は、以下のようなコンマ区切りのオブジェクトリストになります。

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ],
  "log_dirs": [<LogDirs>]
}
```

"log_dirs" プロパティはオプションで、パーティションを特定のログディレクトリーに移動するために使用されます。

以下は、トピック **topic-a** およびパーティション **4** をブローカー **2**、**4** および **7** に割り当て、トピック **topic-b** およびパーティション **2** をブローカー **1**、**5**、および **7** に割り当てる、再割り当て JSON ファイルの例になります。

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7]
    },
    {
      "topic": "topic-b",
      "partition": 2,
      "replicas": [1,5,7]
    }
  ]
}
```

JSON に含まれていないパーティションは変更されません。

10.2.2.2. 再割り当て JSON ファイルの生成

指定のトピックのセットのすべてのパーティションを、指定のブローカーのセットに割り当てる最も簡単な方法は、**kafka-reassign-partitions.sh --generate** コマンドを使用して再割り当て JSON ファイルを生成することです。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server <bootstrap_address> --topics-to-move-json-file <topics_file> --broker-list <broker_list> --generate
```

<topics_file> は、移動するトピックをリストする JSON ファイルです。これには、以下の構造があります。

```
{
  "version": 1,
  "topics": [
    <topic_objects>
  ]
}
```

ここで **<topic_objects>** は、以下のようなコンマ区切りのオブジェクトリストになります。

```
{
  "topic": <TopicName>
}
```

たとえば、**topic-a** および **topic-b** のすべてのパーティションをブローカー **4** および **7** に移動する場合は、以下を実行します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --topics-to-move-json-file
topics-to-be-moved.json --broker-list 4,7 --generate
```

topics-to-be-moved.json のコンテンツがあります。

```
{
  "version": 1,
  "topics": [
    { "topic": "topic-a"},
    { "topic": "topic-b"}
  ]
}
```

10.2.2.3. 手動による再割り当て JSON ファイルの作成

特定のパーティションを移動したい場合は、再割り当て JSON ファイルを手動で作成できます。

10.2.3. 再割り当てスロットル

パーティションの再割り当てには、多くのデータをブローカー間で移動させる必要があるため、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、再割り当てに **スロットル** を使用できます。スロットルを使用すると、再割り当てに時間がかかる可能性があります。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遅れずに対応することはできず、再割り当ては永久に完了しません。スロットルが高すぎると、クライアントに影響します。たとえば、プロデューサーの場合は、承認待ちが通常のレイテンシーよりも大きくなる可能性があります。コンシューマーの場合は、ポーリング間のレイテンシーが大きいことが原因でスループットが低下する可能性があります。

10.2.4. Kafka クラスターのスケールアップ

この手順では、Kafka クラスターでブローカーの数を増やす方法を説明します。

前提条件

- 既存の Kafka クラスター。
- AMQ ブローカーが **インストールされている** 新しいマシン。
- 拡大されたクラスターで、パーティションをブローカーに再割り当てする方法を示す **再割り当て JSON ファイル**。

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/server.properties
```

3. Kafka ブローカーが稼働していることを確認します。

```
jcmd | grep Kafka
```

4. 新しいブローカーごとに上記の手順を繰り返します。

5. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、パーティションの再割り当てを実行します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server <bootstrap_address> --
reassignment-json-file <reassignment_json_file> --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
```

このコマンドは、2つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備えて、これをファイルに保存する必要があります。2つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡したターゲットの再割り当てです。

6. 再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドラインに別のスロットル率を指定して実行します。以下に例を示します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file reassignment.json --throttle 10000000 --execute
```

7. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したかどうかを定期的に確認します。これは先ほどの手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server <bootstrap_address> --
reassignment-json-file <reassignment_json_file> --verify
```

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 --verify
```

8. **--verify** コマンドによって、移動した各パーティションが正常に完了したことが報告されると、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。

10.2.5. Kafka クラスターのスケールダウン

この手順では、Kafka クラスターでブローカーの数を減らす方法を説明します。

前提条件

- 既存の Kafka クラスター。
- ブローカーが削除された後にクラスターのブローカーにパーティションを再割り当てする方法が記述されている **再割り当て JSON ファイル**。

手順

1. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、パーティションの再割り当てを実行します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server <bootstrap_address> --reassignment-json-file <reassignment_json_file> --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
```

このコマンドは、2つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備えて、これをファイルに保存する必要があります。2つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡したターゲットの再割り当てです。

2. 再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドラインに別のスロットル率を指定して実行します。以下に例を示します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-json-file reassignment.json --throttle 10000000 --execute
```

3. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したかどうかを定期的に確認します。これは先ほどの手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
/opt/kafka/bin/kafka-reassign-partitions.sh --bootstrap-server <bootstrap_address> --reassignment-json-file <reassignment_json_file> --verify
```

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 --verify
```

4. **--verify** コマンドによって、移動した各パーティションが正常に完了したことが報告されると、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。
5. 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$`) と一致しない場合には、アクティブなパーティションは引き続き存在します。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ないことを確認します。

6. ブローカーを停止します。

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

7. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

10.2.6. ZooKeeper クラスターのスケールアップ

この手順では、Zoo Keeper クラスターにサーバー (ノード) を追加する方法について説明します。Zoo Keeper の [動的再設定](#) 機能は、スケールアッププロセス中に安定した Zoo Keeper クラスターを維持します。

前提条件

- 動的再設定が ZooKeeper 設定ファイル (`reconfigEnabled=true`) で有効になっている。
- ZooKeeper の認証が有効化され、認証メカニズムを使用して新しいサーバーにアクセスできる。

手順

各 ZooKeeper サーバーに対して、1 つずつ以下の手順を実行します。

1. 「[マルチノードの ZooKeeper クラスターの実行](#)」の説明に従ってサーバーを ZooKeeper クラスターに追加し、ZooKeeper を起動します。
2. 新しいサーバーの IP アドレスと設定されたアクセスポートをメモします。
3. サーバーの `zookeeper-shell` セッションを開始します。クラスターにアクセスできるマシンから次のコマンドを実行します (アクセスできる場合は、Zoo Keeper ノードまたはローカルマシンの 1 つである可能性があります)。

```
su - kafka
/opt/kafka/bin/zookeeper-shell.sh <ip-address>:<zk-port>
```

4. シェルセッションで、Zoo Keeper ノードが実行されている状態で、次の行を入力して、新しいサーバーを投票メンバーとしてクォーラムに追加します。

```
reconfig -add server.<positive-id> = <address1>:<port1>:<port2>[:role];<client-port>
address>:<client-port>
```

以下に例を示します。

```
reconfig -add server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
```

<positive-id> は、新しいサーバー ID 4 です。

2つのポートの <port1> 2888 は ZooKeeper サーバー間の通信用で、<port2> 3888 はリーダーエレクトション用です。

新しい設定は ZooKeeper クラスターの他のサーバーに伝播されます。新しいサーバーはクォーラムの完全メンバーになります。

5. 追加する他のサーバーについて、手順1から4を繰り返します。

10.2.7. ZooKeeper クラスターのスケールダウン

この手順では、ZooKeeper クラスターからサーバー (ノード) を削除する方法を説明します。ZooKeeper の [動的再設定](#) 機能は、スケールダウンプロセス中に安定した ZooKeeper クラスターを維持します。

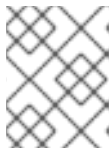
前提条件

- 動的再設定が ZooKeeper 設定ファイル (**reconfigEnabled=true**) で有効になっている。
- ZooKeeper の認証が有効化され、認証メカニズムを使用して新しいサーバーにアクセスできる。

手順

各 ZooKeeper サーバーに対して、1つずつ以下の手順を実行します。

1. スケールダウンの後も **維持される** サーバーのいずれかで、**zookeeper-shell** にログインします (例: サーバー 1)。



注記

ZooKeeper クラスターに設定された認証メカニズムを使用してサーバーにアクセスします。

2. サーバー (サーバー 5 など) を削除します。

```
reconfig -remove 5
```

3. 削除したサーバーを無効にします。
4. ステップ1から3を繰り返し、クラスターサイズを縮小します。

関連情報

- ZooKeeper のドキュメントの [サーバーの削除](#)

第11章 KAFKA クライアントの追加

kafka-clients JAR ファイルには、Kafka Producer および Consumer API と、Kafka AdminClient API が含まれています。

- Producer API は、アプリケーションが Kafka ブローカーにデータを送信できるようにします。
- Consumer API は、アプリケーションが Kafka ブローカーからデータを消費できるようにします。
- AdminClient API は、トピック、ブローカー、およびその他のコンポーネントなどの Kafka クラスターを管理するための機能を提供します。

11.1. MAVEN プロジェクトへの依存関係としての KAFKA クライアントの追加

この手順では、AMQ Streams Java クライアントを Maven プロジェクトに依存関係として追加する方法を説明します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリを **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- ... -->

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

  <!-- ... -->

</project>
```

2. クライアントを **pom.xml** ファイルの **<dependencies>** セクションに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<!-- ... -->

<dependencies>
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.2.3.redhat-00003</version>
  </dependency>
</dependencies>

<!-- ... -->
</project>
```

3. Maven プロジェクトをビルドします。

第12章 KAFKA ストリーム API の追加

Kafka Streams API を使用すると、アプリケーションは1つ以上の入力ストリームからデータを受け取り、マッピング、フィルターリング、または結合などの複雑な操作を実行し、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 リポジトリで利用可能な **kafka-streams** JAR パッケージの一部です。

12.1. MAVEN プロジェクトへの依存関係としての KAFKA STREAMS API の追加

この手順では、AMQ Streams Java クライアントを Maven プロジェクトに依存関係として追加する方法を説明します。

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

1. Red Hat Maven リポジトリを **pom.xml** ファイルの **<repositories>** セクションに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- ... -->

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

  <!-- ... -->

</project>
```

2. **pom.xml** ファイルの **<dependencies>** セクションに **kafka-streams** を追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- ... -->

  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
```

```
        <artifactId>kafka-streams</artifactId>
        <version>3.2.3.redhat-00003</version>
    </dependency>
</dependencies>

<!-- ... -->
</project>
```

3. Maven プロジェクトをビルドします。

第13章 KERBEROS (GSSAPI) 認証の使用

AMQ Streams は、Kafka クラスターへの安全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。

13.1. KERBEROS (GSSAPI) 認証を使用するための AMQ STREAMS の設定

この手順では、Kafka クライアントが Kerberos (GSSAPI) 認証を使用して Kafka および ZooKeeper にアクセスできるように AMQ Streams を設定する方法を説明します。

この手順では、Kerberos `krb5` リソースサーバーが Red Hat Enterprise Linux ホストに設定されていることを前提としています。

この手順では、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. Kerberos ログインを使用する Kafka ブローカー
3. Kerberos ログインを使用するための ZooKeeper
4. Kerberos 認証を使用して Kafka にアクセスするためのプロデューサーおよびコンシューマクライアント

この手順では、単一のホストでの単一の ZooKeeper および Kafka インストールの Kerberos 設定について説明し、プロデューサーおよびコンシューマクライアントの追加設定についても説明します。

前提条件

Kafka および ZooKeeper が Kerberos クレデンシャルを認証および承認するように設定できるようにするには、以下が必要です。

- Kerberos サーバーへのアクセス
- 各 Kafka ブローカーホストの Kerberos クライアント

Kerberos サーバー、およびブローカーホストのクライアントを設定する手順の詳細は、[AMQ Streams on RHEL - Example Kerberos set up configuration](#) を参照してください。

認証用のサービスプリンシパルの追加

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

1. Kerberos KDC 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。
以下に例を示します。
 - `zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM`
 - `kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM`

- **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
ZooKeeper サービスプリンシパルは、Kafka **config/server.properties** ファイルの **zookeeper.connect** 設定と同じホスト名である必要があります。

```
zookeeper.connect=node1.example.redhat.com:2181
```

ホスト名が同じでない場合、**localhost** が使用され、認証に失敗します。

2. ホストにディレクトリーを作成し、キータブファイルを追加します。
以下に例を示します。

```
/opt/kafka/krb5/zookeeper-node1.keytab
/opt/kafka/krb5/kafka-node1.keytab
/opt/kafka/krb5/kafka-producer1.keytab
/opt/kafka/krb5/kafka-consumer1.keytab
```

3. **kafka** ユーザーがディレクトリーにアクセスできることを確認します。

```
chown kafka:kafka -R /opt/kafka/krb5
```

Kerberos ログインを使用するための ZooKeeper の設定

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

1. **opt/kafka/config/jaas.conf** ファイルを作成または変更して、ZooKeeper クライアントおよびサーバー操作をサポートします。

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true ①
  storeKey=true ②
  useTicketCache=false ③
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab" ④
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM"; ⑤
};

Server {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumServer {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

```
QuorumLearner {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

- ❶ **true** に設定し、キータブからプリンシパルキーを取得します。
- ❷ **true** に設定し、プリンシパルキーを保存します。
- ❸ **true** に設定し、チケットキャッシュから Ticket Granting Ticket (TGT) を取得します。
- ❹ **keyTab** プロパティは、Kerberos KDC からコピーされた keytab ファイルの場所を示します。その場所とファイルは、**kafka** ユーザーが読み取りできるものでなければなりません。
- ❺ **principal** プロパティは、KDC ホストで作成された完全修飾プリンシパル名と一致するように設定され、その形式は **SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-NAME** に従います。

2. **opt/kafka/config/zookeeper.properties** を編集して、更新された JAAS 設定を使用します。

```
# ...

requireClientAuthScheme=sasl
jaasLoginRenew=3600000 ❶
kerberos.removeHostFromPrincipal=false ❷
kerberos.removeRealmFromPrincipal=false ❸
quorum.auth.enableSasl=true ❹
quorum.auth.learnerRequireSasl=true ❺
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner ❻
quorum.auth.server.loginContext=QuorumServer
quorum.auth.kerberos.servicePrincipal=zookeeper/_HOST ❼
quorum.cnxn.threads.size=20
```

- ❶ ログイン更新の頻度をミリ秒単位で制御します。これは、チケットの更新間隔に合わせて調整できます。デフォルトは1時間です。
- ❷ ホスト名がログインプリンシパル名の一部として使用されるかどうかを指定します。クラスターのすべてのノードで単一の keytab を使用する場合、これは **true** に設定されます。ただし、トラブルシューティングのために、各ブローカーホストに個別のキータブと完全修飾プリンシパルを生成することが推奨されます。
- ❸ Kerberos ネゴシエーションのプリンシパル名からレルム名を削除するかどうかを制御します。この設定は、**false** にすることをお勧めします。
- ❹ ZooKeeper サーバーおよびクライアントの SASL 認証メカニズムを有効にします。
- ❺ **RequireSasl** プロパティは、マスター選出などのクォーラムイベントに SASL 認証を必要とするかどうかを制御します。

- 6 **loginContext** プロパティは、指定されたコンポーネントの認証設定に使用される JAAS 設定のログインコンテキストの名前を識別します。loginContext 名
- 7 識別に使用されるプリンシパル名を形成するために使用される命名規則を制御します。プレースホルダー **_HOST** は、実行時に **server.1** プロパティによって定義されたホスト名に自動的に解決されます。

3. JVM パラメーターで ZooKeeper を起動し、Kerberos ログイン設定を指定します。

```
su - kafka
export EXTRA_ARGS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/zookeeper-server-
start.sh -daemon /opt/kafka/config/zookeeper.properties
```

デフォルトのサービス名 (**zookeeper**) を使用していない場合は、**-Dzookeeper.sasl.client.username=NAME** パラメーターを使用して名前を追加します。



注記

/etc/krb5.conf を場所として使用している場合は、ZooKeeper、Kafka、Kafka プロデューサーおよびコンシューマーの起動時に **-Djava.security.krb5.conf=/etc/krb5.conf** を指定する必要はありません。

Kerberos ログインを使用するための Kafka ブローカーサーバーの設定

認証に Kerberos Key Distribution Center (KDC) を使用するように **kafka** のため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 ❶
inter.broker.listener.name=REPLICATION
# ...
listener.security.protocol.map=SECURE:SASL_PLAINTEXT,REPLICATION:SASL_PLAINTEXT ❷
# ..
sasl.enabled.mechanisms=GSSAPI ❸
sasl.mechanism.inter.broker.protocol=GSSAPI ❹
sasl.kerberos.service.name=kafka ❺
...
```

- ❶ クライアントとの汎用通信 (通信用の TLS をサポート) と inter-broker 通信用のレプリケーションリスナーの 2 つのリスナーが設定されます。
- ❷ TLS 対応のリスナーの場合、プロトコル名は SASL_PLAINTEXT です。TLS に対応していないコネクタの場合、プロトコル名は SASL_PLAINTEXT です。SSL が必要ない場合は、**ssl.*** プロパティを削除できます。
- ❸ Kerberos 認証のための SASL メカニズムは **GSSAPI** です。
- ❹ inter-broker 通信の Kerberos 認証。
- ❺ 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/server.properties
```

ブローカーおよび ZooKeeper クラスターが以前に設定されていて、Kerberos ベース以外の認証システムで動作している場合は、ZooKeeper およびブローカークラスターを起動し、ログで設定エラーを確認することができます。

ブローカーおよび Zookeeper インスタンスを起動すると、Kerberos 認証用にクラスターが設定されました。

Kerberos 認証を使用するための Kafka プロデューサーおよびコンシューマクライアントの設定
 認証に Kerberos Key Distribution Center (KDC) を使用するように **producer1** および **consumer1** のため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 ❶
security.protocol=SASL_PLAINTEXT ❷
```

```
sasl.kerberos.service.name=kafka ❸
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ ❹
    useKeyTab=true \
    useTicketCache=false \
    storeKey=true \
    keyTab="/opt/kafka/krb5/producer1.keytab" \
    principal="producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

- ❶ Kerberos (GSSAPI) 認証の設定。
- ❷ Kerberos は SASL プレーンテキスト (ユーザー名/パスワード) セキュリティプロトコルを使用します。
- ❸ Kerberos KDC で設定された Kafka のサービスプリンシパル (ユーザー)。
- ❹ **jaas.conf** で定義されたものと同じプロパティを使用した JAAS の設定。

/opt/kafka/config/consumer.properties

```
# ...
sasl.mechanism=GSSAPI
security.protocol=SASL_PLAINTEXT
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true \
    useTicketCache=false \
    storeKey=true \
    keyTab="/opt/kafka/krb5/consumer1.keytab" \
    principal="consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

2. クライアントを実行して、Kafka ブローカーからメッセージを送受信できることを確認します。
プロデューサークライアント:

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-producer.sh --producer.config
/opt/kafka/config/producer.properties --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

コンシューマークライアント:

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-consumer.sh --
consumer.config /opt/kafka/config/consumer.properties --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

関連情報

- Kerberos の man ページ: `krb5.conf(5)`、`kinit(1)`、`klist(1)`、および `kdestroy(1)`
- [AMQ Streams on RHEL - Example Kerberos set up configuration](#)

- [AMQ Streams on RHEL - Example Kafka client with Kerberos authentication](#)

第14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 AMQ Streams の Cruise Control 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。

AMQ Streams 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

最適化の目標は、リバランスから達成する特定の目標を表します。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目標になる場合があります。設定を通じて、含める目標を変更できます。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。調整されたゴールのセットを使用して、プロポーザルを拒否し、再度実行するようプロポーザルを拒否することができます。

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、設定プロパティファイルで設定します。AMQ Streams には、Cruise Control のプロパティファイルのサンプルが含まれています。

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。

14.1. CRUISE CONTROL とは

Cruise Control は、分散された Kafka クラスタを効率的に実行するための時間および労力を削減します。

通常、クラスタの負荷は時間とともに不均等になります。大量のメッセージトラフィックを処理するパーティションは、使用可能なブローカー全体で不均等に分散される可能性があります。クラスタを再分散するには、管理者はブローカーの負荷を監視し、トラフィックの多いパーティションを容量に余裕のあるブローカーに手作業で再割り当てします。

Cruise Control はクラスタのリバランス処理を自動化します。CPU、ディスク、およびネットワーク負荷を基にして、クラスタにおけるリソース使用のワークロードモデルを構築し、パーティションの割り当てをより均等にする、最適化プロポーザル (承認または拒否可能) を生成します。これらのプロポーザルの算出には、設定可能な最適化ゴールが複数使用されます。

最適化プロポーザルを承認すると、Cruise Control はそのプロポーザルを Kafka クラスタに適用します。クラスタのリバランス操作が完了すると、ブローカー Pod はより効率的に使用され、Kafka クラスタはより均等に分散されます。

関連情報

- [Cruise Control の Wiki](#)

14.2. CRUISE CONTROL アーカイブのダウンロード

Red Hat Enterprise Linux 上の AMQ Streams の Cruise Control は zip ディストリビューションとして、[Red Hat カスタマーポータル](#) 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 AMQ Streams Cruise Control アーカイブをダウンロードします。
2. `/opt/cruise-control` ディレクトリーを作成します。

```
sudo mkdir /opt/cruise-control
```

3. Cruise Control ZIP ファイルの内容を新しいディレクトリーに展開します。

```
unzip amq-streams-y.y.y-cruise-control-bin.zip -d /opt/cruise-control
```

4. `/opt/cruise-control` ディレクトリーの所有権を `kafka` ユーザーに変更します。

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

14.3. 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 AMQ Streams インストールアーティファクトで提供されます。

実行時にロードされると、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 はこれらのメトリクスを使用して、ワークロードモデルを作成および更新し、最適化プロポーザルを計算します。

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。
- Kafka と ZooKeeper が実行されている。

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 設定ファイル (**/opt/kafka/config/server.properties**) で、Cruise Control Metrics Reporter を設定します。

- a. **CruiseControlMetricsReporter** クラスを **metric.reporters** 設定オプションに追加します。既存の Metrics 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 設定ファイル (**/opt/kafka/config/server.properties**) では、関連するクライアント設定プロパティを設定することで、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を設定することで、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れらのプロパティを使用します。

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/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。

14.4. CRUISE CONTROL の設定および起動

Cruise Control が使用するプロパティを設定し、**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 アーカイブのダウンロード」](#)
- [「Cruise Control Metrics Reporter のデプロイ」](#)

手順

1. Cruise Control プロパティファイル (**/opt/cruise-control/config/cruisecontrol.properties**) を編集します。
2. 以下の設定例のように、プロパティを設定します。

```
# The Kafka cluster to control.
bootstrap.servers=localhost:9092 ❶

# The replication factor of Kafka metric sample store topic
sample.store.topic.replication.factor=2 ❷

# 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 ❸

# The list of goals to optimize the Kafka cluster for with pre-computed proposals
default.goals={List of default optimization goals} ❹

# The list of supported goals
goals={list of main optimization goals} ❺

# The list of supported hard goals
hard.goals={List of hard goals} ❻
```

```
# How often should the cached proposal be expired and recalculated if necessary
```

```
proposal.expiration.ms=60000 7
```

```
# The zookeeper connect of the Kafka cluster
```

```
zookeeper.connect=localhost:2181 8
```

- 1 Kafka ブローカーのホストおよびポート番号 (常にポート 9092)。
 - 2 Kafka メトリクスサンプルストアトピックのレプリケーション係数。単一ノードの Kafka および ZooKeeper クラスターで Cruise Control を評価する場合は、このプロパティを 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目標 (5 を参照) は、デフォルトの最適化目標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、主な最適化目標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。主な最適化目標のうち 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 ZooKeeper 接続のホストおよびポート番号 (常にポート 2181)。
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 が実行されていることを確認するには、Cruise Control サーバーの **/state** エンドポイントに GET リクエストを送信します。

```
curl 'http://HOST:PORT/kafkacruisecontrol/state'
```

自動作成されたトピック

以下の表は、Cruise Control の起動時に自動的に作成される 3 つのトピックを表しています。これらのトピックは、Cruise Control が適切に動作するために必要であるため、削除または変更しないでください。

表14.1 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クスを格納します。これらは Metric Sample Aggregator によって作成されます。
__KafkaCruiseControlModelTrainingSamples	Cruise Control	クラスターワークロードモデル の作成に使用されるメトリクスサンプルを格納します。

自動作成されたトピックでログコンパクションが **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](#)

14.5. 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 クラスターをリバランスするために、最適化ゴールを使用して [最適化プロポーザル](#) を生成します。

14.5.1. 優先度のゴールの順序

Red Hat Enterprise Linux 上の AMQ Streams は、Cruise Control プロジェクトで開発された最適化の目標をすべてサポートします。以下に、サポートされるゴールをデフォルトの優先度順に示します。

1. ラックアウェアネス (Rack Awareness)
2. トピックのセットに対するブローカーごとのリーダーレプリカの最小数
3. レプリカの容量
4. 容量: ディスク容量、ネットワークインバウンド容量、ネットワークアウトバウンド容量
5. CPU 容量
6. レプリカの分散
7. 潜在的なネットワーク出力

8. リソース分布: ディスク使用率の分布、ネットワークインバウンド使用率の分布、ネットワークアウトバウンド使用率の分布。
9. リーダーへの単位時間あたりバイト流入量の分布
10. トピックレプリカの分散
11. CPU 使用率の分散
12. リーダーレプリカの分散
13. 優先リーダーエレクション
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。

14.5.2. Cruise Control プロパティファイルのゴール設定

最適化目標の設定は、**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、満たす必要のある厳しい最適化目標のほか、メイン、デフォルト、およびユーザー指定の最適化目標の設定があります。

次の設定では、次のタイプの最適化目標を指定できます。

- **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。

14.5.3. ハードおよびソフト最適化目標

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、トピックレプリカ分散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。

14.5.4. 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。

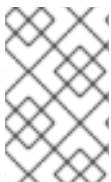
次の主な最適化 **ゴール** は、**cruisecontrol.properties** ファイルの **goal** プロパティに優先順位の降順で事前設定されています。

```
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 つ以上の目標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目標を変更しないことをお勧めします。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目標を変更するには、目標のリストを **ゴール** プロパティ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。

主な目標を少なくとも 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目標を変更する場合は、設定された **hard.goals** が、設定したメインの最適化目標のサブセットであることを確認する必要があります。そうでないと、最適化プロポーザルの生成時にエラーが発生します。

14.5.5. 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目標は、**cruisecontrol.properties** ファイルの **default.goals** プロパティ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目標を変更するには、**default.goals** プロパティで目標のリストを優先度の高い順に指定します。デフォルトの目標は、主要な最適化目標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。

14.5.6. 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。
- [主な最適化目標](#) のサブセットになる

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](#)

14.6. 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、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**

14.6.1. 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場合には、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た後すぐに **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、パーティションが複製されない可能性があり、一部のパーティションが最小 ISR(同期レプリカ) 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行うことができます。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。

リバランスを実行する手順は、実際には3つの異なるエンドポイント間で同じです。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一覧表示のみです。

14.6.2. 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、[最適化目標の変更](#) 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場合、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役立つ場合があります。

リバランス時には 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役立ちます。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。これには、パフォーマンスへの影響が少ない ZooKeeper の設定を変更する必要があります。

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場合、スコアは 100 です。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14.2 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。

14.6.3. 最適化プロポーザルサマリーのプロパティ

以下の表は、最適化プロポーザルに含まれるプロパティを表しています。

表14.3 最適化プロポーザルに含まれるプロパティの概要

プロパティ	説明
n inter-broker replica (y MB) moves	<p>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。</p> <p>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。</p> <p>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。</p> <p>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 の数が多いほど、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。</p>

プロパティ	説明
n intra-broker replica (y MB) moves	<p>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。</p> <p>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。</p> <p>y MB: 同じブローカーのディスク間で移動される各パーティションレプリカのサイズの合計。</p> <p>リバランス操作中のパフォーマンスへの影響度: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。大量のデータを移動する場合、同じブローカーのディスク間で移動する方が個別のブローカー間で移動するよりも影響度が低くなります (inter-broker replica moves 参照)。</p>
n excluded topics	<p>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。</p> <p>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。</p> <p>cruisecontrol.properties ファイル で、topics.excluded.from.partition.movement プロパティに正規表現を指定します。</p> <p>/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。</p> <p>正規表現と一致するトピックが応答に一覧表示され、クラスターのリバランスから除外されます。</p>
n leadership moves	<p>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。ZooKeeper 設定の変更を伴います。</p> <p>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。</p>
n recent windows	n : 最適化プロポーザルの基になるメトリクスウィンドウの数。
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)。
On-demand Balancedness Score Before (nn.yyy) After (nn.yyy)	<p>Kafka クラスターの全体的なバランスの測定。</p> <p>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。</p> <p>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。</p>

14.6.4. 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](#)

14.7. リバランスパフォーマンスチューニングの概要

クラスターリバランスのパフォーマンスチューニングオプションを調整できます。これらのオプションは、リバランスのパーティションレプリカおよびリーダーシップの移動が実行される方法を制御し、また、リバランス操作に割り当てられた帯域幅も制御します。

パーティション再割り当てコマンド

[最適化プロポーザル](#) は、個別のパーティション再割り当てコマンドで設定されています。プロポーザルを開始すると、Cruise Control サーバーはこれらのコマンドを Kafka クラスターに適用します。

パーティション再割り当てコマンドは、以下のいずれかの操作で設定されます。

- **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がある場合、このストラテジーによって他の再割り当ての適用が遅くなる可能性があります。

Cruise Control は、最適化プロポーザルに適用できる 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14.4 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.movement.per.broker	concurrent_partition_movements_per_broker	5	各パーティション再割り当てバッチでの 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でのブローカー内パーティション移動の最大数。

Cruise Control プロパティ	KafkaRebalance パラメーター	デフォルト	説明
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	KafkaRebalance パラメーター	デフォルト	説明
<code>default.replica.movement.strategies</code>	<code>replica_movement_strategies</code>	Base Repl caMo veme ntStr ategy	<p>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</p> <p>PrioritizeSmall ReplicaMove mentStrategy、PrioritizeLargeReplicaMovementStrategy、および PostponeUrp ReplicaMove mentStrategy があります。</p> <p>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。</p>

デフォルト設定を変更すると、リバランスの完了までにかかる時間と、リバランス中の Kafka クラスターの負荷に影響します。値を小さくすると負荷は減りますが、かかる時間は長くなり、その逆も同様です。

関連情報

- Cruise Control Wiki の [Configurations](#)

- Cruise Control Wiki の [REST API](#)

14.8. 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。

- String
- 数値
- ブール値

Cruise Control Wiki の [Configurations](#) セクションに記載されているすべてのプロパティーを指定および設定できます。

容量の設定

Cruise Control は **容量制限** を使用して、特定のリソースベースの最適化ゴールが破損しているか判断します。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** になっている場合、**__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ギングに **log4j1** を使用します。デフォルト設定を変更するには、**/opt/cruise-control/config/log4j.properties** の **log4j.properties** ファイルを編集します。

変更を反映するには、Cruise Control サーバーを再起動する必要があります。

14.9. 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 クラスターですでに実行中のすべてのブローカー De リバランスを行うか、または 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。"dry run mode" 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クスと、前後の値の違いが含まれます。

excluded_topics

type: 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

type: 文字列のリスト。default: 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

type: 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

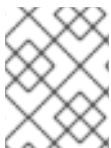
type: boolean、デフォルト: **false**

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** に設定された場合、Cruise Control はコマンドラインで表示するためにフォーマットされたテキストを返します。返された情報の要素を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

type: 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 と ZooKeeper が実行されている。
- Cruise Control が実行されている。
- (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](#) している。

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 が返されると、Cruise Control は最適化プロポーザルを生成するために十分なメトリクスデータを記録していません。数分待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場合、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。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=true'
```

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 2020 15:19:26 GMT
< Set-Cookie: JSESSIONID=node01wk6vjzjj12go13m81o7no5p7h9.node0; Path=/
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201
< Content-Type: text/plain;charset=utf-8
< Cruise-Control-Version: 2.0.103.redhat-00002
< Cruise-Control-Commit_Id: 58975c9d5d0a78dd33cd67d4bcb497c9fd42ae7c
< Content-Length: 12368
< Server: Jetty(9.4.26.v20200117-redhat-00001)
```

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」](#)

14.10. 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た場合、クラスターの状態が変更されている可能性があります。そのため、開始されたクラスターのリバランスは、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 0:0:0:0:0:0:1 2020-06-01_16:10:29 UTC
Active        POST /kafkacruisecontrol/rebalance?dryrun=false
```

```
445e2fc3-6531-4243-b0a6-36ef7c5059b4 0:0:0:0:0:0:1 2020-06-01_14:21:26 UTC
Completed GET /kafkacruisecontrol/state?json=true
05c37737-16d1-4e33-8e2b-800dee9f1b01 0:0:0:0:0:0:1 2020-06-01_14:36:11 UTC
Completed GET /kafkacruisecontrol/state?json=true
aebae987-985d-4871-8cfb-6134ecd504ab 0:0:0:0:0:0:1 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$`) と一致しない場合には、アクティブなパーティションは引き続き存在します。アクティブなパーティションがある場合は、リバランスが完了したか、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
```

14.11. 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、初期状態の場合よりも悪くなる可能性があります。

前提条件

- クラスターリバランスが進行中である (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15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で実行されるトランザクションの進捗を追跡できます。マイクロサービスのアーキテクチャーでは、トレースはサービス間のトランザクションの進捗を追跡します。トレースデータは、アプリケーションのパフォーマンスを監視し、ターゲットシステムおよびエンドユーザーアプリケーションの問題を調べるのに役立ちます。

Red Hat Enterprise Linux での AMQ Streams は、トレーシングにより、ソースシステムから Kafka へ、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。トレースは、利用可能な [JMX メトリクス](#) を補完します。

AMQ Streams によるトレーシングのサポート方法

以下のクライアントおよびコンポーネントに対して、トレースのサポートが提供されます。

Kafka クライアント:

- Kafka プロデューサーおよびコンシューマー
- Kafka Streams API アプリケーション

Kafka コンポーネント:

- Kafka Connect
- Kafka Bridge
- MirrorMaker
- MirrorMaker 2.0



注記

OpenTracing のサポートは非推奨となりました。Jaeger クライアントは廃止され、OpenTracing プロジェクトはアーカイブされました。そのため、将来の Kafka バージョンのサポートを保証することはできません。OpenTelemetry プロジェクトに基づく新しいトレース実装を導入しています。

トレースを有効にするには、ハイレベルタスクを 4 つ実行します。

1. Jaeger トレーサーを有効にします。
2. インターセプターを有効にします。
 - Kafka クライアントの場合、[OpenTracing Apache Kafka Client Instrumentation](#) ライブラリー (AMQ Streams に含まれる) を使用してアプリケーションコードを **インストルメント化** します。
 - Kafka コンポーネントでは、各コンポーネントに設定プロパティを設定します。
3. [トレーシングの環境変数](#) を設定します。
4. クライアントまたはコンポーネントをデプロイします。

インストルメント化されると、クライアントはトレースデータを生成します。たとえば、メッセージの作成時やログへのオフセットの書き込み時などです。

トレースは、サンプリングストラテジーに従いサンプル化され、Jaeger ユーザーインターフェイスで可視化されます。



注記

トレーシングは Kafka ブローカーではサポートされません。

AMQ Streams 以外のアプリケーションおよびシステムにトレーシングを設定する方法については、本章の対象外となります。この件についての詳細は、[OpenTracing のドキュメント](#)を参照し、inject and extrac を検索してください。

手順の概要

AMQ Streams のトレーシングを設定するには、以下の手順を順番に行います。

1. クライアントのトレーシングを設定します。
 - a. [Kafka クライアント用の Jaeger トレーサーの初期化](#)
 - b. [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](#)
 - c. [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](#)
2. MirrorMaker、MirrorMaker 2.0、および Kafka Connect のトレースを設定します。
 - a. [MirrorMaker のトレースの有効化](#)
 - b. [MirrorMaker 2.0 のトレースの有効化](#)
 - c. [Kafka Connect のトレースの有効化](#)
3. [Kafka Bridge のトレースを有効化](#)します。

前提条件

- Jaeger バックエンドコンポーネントが Kubernetes クラスタにデプロイされている。デプロイメント手順は、[Jaeger のドキュメント](#)を参照してください。

15.1. OPENTRACING および JAEGER の概要

AMQ Streams では OpenTracing および Jaeger プロジェクトが使用されます。

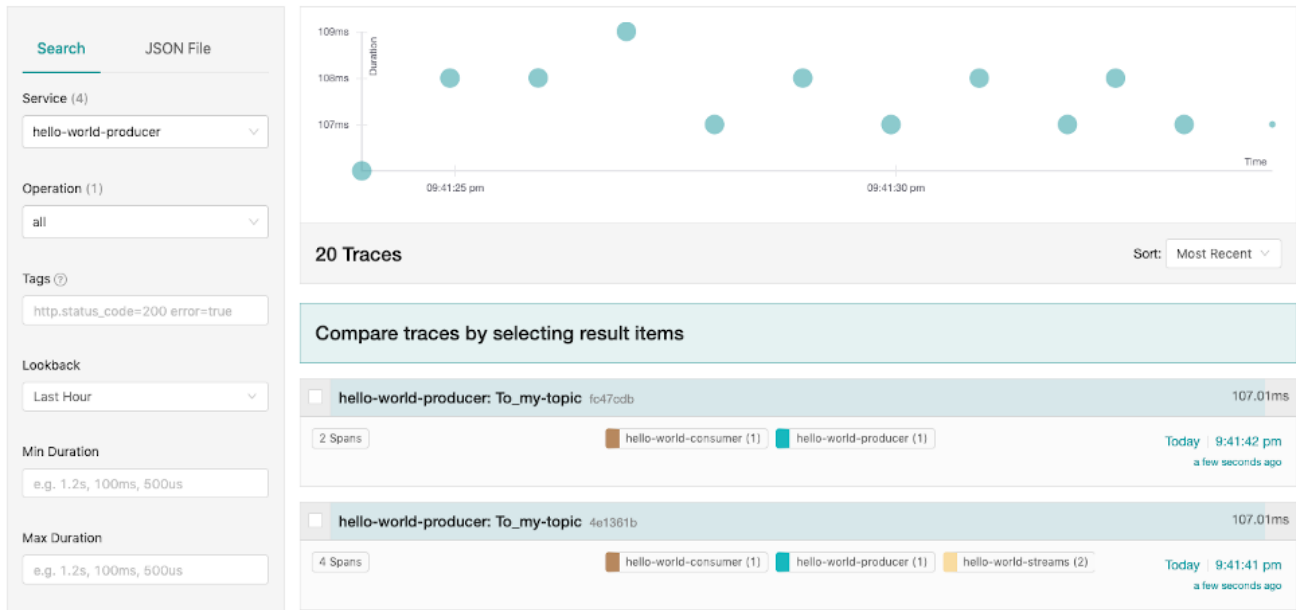
OpenTracing は、トレーシングまたは監視システムに依存しない API 仕様です。

- OpenTracing API は、アプリケーションコードを **インストルメント化** するために使用されます。
- インストルメント化されたアプリケーションは、分散システム全体で個別のトランザクションの **トレース** を生成します。
- トレースは、特定の作業単位を定義する **スパン** で設定されます。

Jaeger はマイクロサービスベースの分散システムのトレーシングシステムです。

- Jaeger は OpenTracing API を実装し、インストルメント化のクライアントライブラリーを提供します。

- Jaeger ユーザーインターフェイスを使用すると、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。



関連情報

- [OpenTracing](#)
- [Jaeger](#)

15.2. KAFKA クライアントのトレーシング設定

Jaeger トレーサーを初期化し、分散トレーシング用にクライアントアプリケーションをインストールメント化します。

15.2.1. Kafka クライアント用の Jaeger トレーサーの初期化

一連の [トレーシング環境変数](#) を使用して、Jaeger トレーサーを設定および初期化します。

手順

各クライアントアプリケーションで以下を行います。

1. Jaeger の Maven 依存関係をクライアントアプリケーションの **pom.xml** ファイルに追加します。

```
<dependency>
  <groupId>io.jaegertracing</groupId>
  <artifactId>jaeger-client</artifactId>
  <version>1.5.0.redhat-00001</version>
</dependency>
```

2. [トレーシング環境変数](#) を使用して Jaeger トレーサーの設定を定義します。
3. 2. で定義した環境変数から、Jaeger トレーサーを作成します。

```
Tracer tracer = Configuration.fromEnv().getTracer();
```



注記

別の Jaeger トレーサーの初期化方法については、[Java OpenTracing ライブラリー](#) のドキュメントを参照してください。

4. Jaeger トレーサーをグローバルトレーサーとして登録します。

```
GlobalTracer.register(tracer);
```

これで、Jaeger トレーサーはクライアントアプリケーションが使用できるように初期化されました。

15.2.2. 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

Decorator パターンまたは Interceptor を使用して、Java プロデューサーおよびコンシューマーアプリケーションコードをトレーシング用にインストルメント化します。

手順

各プロデューサーおよびコンシューマーアプリケーションのアプリケーションコードで以下を行います。

1. OpenTracing の Maven 依存関係を、プロデューサーまたはコンシューマーの **pom.xml** ファイルに追加します。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-client</artifactId>
  <version>0.1.15.redhat-00006</version>
</dependency>
```

2. Decorator パターンまたは Interceptor のいずれかを使用して、クライアントアプリケーションコードをインストルメント化します。

- Decorator パターンを使用する場合は以下を行います。

```
// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer:
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer,
  tracer);

// Send:
tracingProducer.send(...);

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer:
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer,
  tracer);

// Subscribe:
```

```
tracingConsumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);

// Retrieve SpanContext from polled record (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracing);
```

- Interceptor を使用する場合は以下を使用します。

```
// Register the tracer with GlobalTracer:
GlobalTracer.register(tracing);

// Add the TracingProducerInterceptor to the sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingProducerInterceptor.class.getName());

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send:
producer.send(...);

// Add the TracingConsumerInterceptor to the consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingConsumerInterceptor.class.getName());

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Subscribe:
consumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// Retrieve the SpanContext from a polled message (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracing);
```

Decorator パターンのカスタムスパン名

スパン は Jaeger の論理作業単位で、操作名、開始時間、および期間が含まれます。

プロデューサーとコンシューマーのアプリケーションをインストルメントするために Decorator パターンを使用するには、**TracingKafkaProducer** および **TracingKafkaConsumer** オブジェクトを作成する際に、追加の引数として **BiFunction** オブジェクトを渡して、カスタムスパン名を定義します。OpenTracing の Apache Kafka Client Instrumentation ライブラリーには、複数の組み込みスパン名が含まれています。

例: カスタムスパン名を使用した Decorator パターンでのクライアントアプリケーションコードのインストルメント化

```
// Create a BiFunction for the KafkaProducer that operates on (String operationName,
// ProducerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ProducerRecord, String> producerSpanNameProvider =
    (operationName, producerRecord) -> "CUSTOM_PRODUCER_NAME";

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>(producer,
    tracer,
    producerSpanNameProvider);

// Spans created by the tracingProducer will now have "CUSTOM_PRODUCER_NAME" as the span
// name.

// Create a BiFunction for the KafkaConsumer that operates on (String operationName,
// ConsumerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ConsumerRecord, String> consumerSpanNameProvider =
    (operationName, consumerRecord) -> operationName.toUpperCase();

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer, passing in the consumerSpanNameProvider
// BiFunction:

TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
(consumer,
    tracer,
    consumerSpanNameProvider);

// Spans created by the tracingConsumer will have the operation name as the span name, in upper-
// case.
// "receive" -> "RECEIVE"
```

ビルトインスパン名

カスタムスパン名を定義するとき、**ClientSpanNameProvider** クラスで以下の **BiFunctions** を使用できます。**spanNameProvider** を指定しないと、**CONSUMER_OPERATION_NAME** および **PRODUCER_OPERATION_NAME** が使用されます。

表15.1 カスタムスパン名を定義する BiFunctions

BiFunction	説明
CONSUMER_OPERATION_NAME 、 PRODUCER_OPERATION_NAME	operationName をスパン名として返します。コンシューマーには receive、プロデューサーの場合は send を返します。

BiFunction	説明
CONSUMER_PREFIXED_OPERATION_NAME (String prefix), PRODUCER_PREFIXED_OPERATION_NAME (String prefix)	prefix および operationName の文字列連結を返します。
CONSUMER_TOPIC, PRODUCER_TOPIC	メッセージの送信先または送信元となったトピックの名前を (record.topic()) 形式で返します。
PREFIXED_CONSUMER_TOPIC (String prefix), PREFIXED_PRODUCER_TOPIC (String prefix)	prefix およびトピック名の文字列連結を (record.topic()) 形式で返します。
CONSUMER_OPERATION_NAME_TOPIC, PRODUCER_OPERATION_NAME_TOPIC	操作名およびトピック名を " operationName - record.topic() " で返します。
CONSUMER_PREFIXED_OPERATION_NAME_TOPIC (String prefix), PRODUCER_PREFIXED_OPERATION_NAME_TOPIC (String prefix)	prefix および " operationName - record.topic() " の文字列連結を返します。

15.2.3. 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

サプライヤーインターフェイスを使用して、分散トレーシングのために Kafka Streams アプリケーションをインストルメント化します。これにより、アプリケーションのインターセプターが有効になります。

手順

各 Kafka Streams アプリケーションで以下を行います。

1. **opentracing-kafka-streams** 依存関係を、Kafka Streams API アプリケーションの **pom.xml** ファイルに追加します。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-streams</artifactId>
  <version>0.1.15.redhat-00006</version>
</dependency>
```

2. **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ます。

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

3. サプライヤーインターフェイスを **KafkaStreams** に提供します。

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

15.3. MIRRORMAKER および KAFKA CONNECT のトレース設定

本セクションでは、分散トレーシング用に MirrorMaker、MirrorMaker 2.0、および Kafka Connect を設定する方法を説明します。

コンポーネントごとに Jaeger トレーサーを有効にする必要があります。

15.3.1. MirrorMaker のトレースの有効化

Interceptor プロパティ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。

手順

1. Jaeger トレーサーを設定し、有効にします。
2. **/opt/kafka/config/consumer.properties** ファイルを編集します。
以下のインターセプタープロパティを追加します。

```
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor
```

3. **/opt/kafka/config/producer.properties** ファイルを編集します。
以下のインターセプタープロパティを追加します。

```
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
```

4. コンシューマーおよびプロデューサー設定ファイル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 --consumer.config /opt/kafka/config/consumer.properties
--producer.config /opt/kafka/config/producer.properties --num.streams=2
```

15.3.2. MirrorMaker 2.0 のトレースの有効化

MirrorMaker 2.0 プロパティファイルに Interceptor プロパティを定義して、MirrorMaker 2.0 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 2.0 コンポーネントに出入りするメッセージを記録します。

手順

1. Jaeger トレーサーを設定し、有効にします。
2. MirrorMaker 2.0 設定プロパティファイル **./config/connect-mirror-maker.properties** を編集し、以下のプロパティを追加します。


```
header.converter=org.apache.kafka.connect.converters.ByteArrayConverter ❶
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor ❷
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
```

- ❶ 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。
 - ❷ MirrorMaker 2.0 のインターセプターを有効にします。
3. [MirrorMaker 2.0 を使用した Kafka クラスター間でのデータ同期](#) の手順を使用して、MirrorMaker 2.0 を起動します。

関連情報

- [AMQ Streams の MirrorMaker 2.0 との使用](#)

15.3.3. Kafka Connect のトレースの有効化

設定プロパティを使用して 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シングされます。Kafka Connect と外部システム間で送信されるメッセージをトレースするには、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。

手順

1. Jaeger トレーサーを設定し、有効にします。
2. 関連する Kafka Connect 設定ファイルを編集します。
 - 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。
3. 以下のプロパティを設定ファイルに追加します。

```
producer.interceptor.classes=io.opentracing.contrib.kafka.TracingProducerInterceptor
consumer.interceptor.classes=io.opentracing.contrib.kafka.TracingConsumerInterceptor
```

4. 設定ファイルを作成します。
5. トレーシング環境変数を設定してから、スタンドアロンまたは分散モードで Kafka Connect を実行します。

Kafka Connect の内部コンシューマーおよびプロデューサーのインターセプターが有効になりました。

関連情報

- [トレースの環境変数](#)
- [スタンドアロンモードでの Kafka Connect の実行](#)

- [分散 Kafka Connect の実行](#)

15.4. KAFKA BRIDGE のトレースの有効化

Kafka Bridge 設定ファイルを編集して、Kafka Bridge の分散トレーシングを有効にします。その後、分散トレース用に設定された Kafka Bridge インスタンスをホストオペレーティングシステムにデプロイできます。

トレースは、以下の場合に生成されます。

- Kafka Bridge が HTTP クライアントにメッセージを送信し、HTTP クライアントからメッセージを消費する場合
- HTTP クライアントが HTTP リクエストを送信し、Kafka Bridge 経由でメッセージを送受信する場合

エンドツーエンドのトレーシングを設定するために、HTTP クライアントでトレーシングを設定する必要があります。

手順

1. Kafka Bridge インストールディレクトリーの **config/application.properties** ファイルを編集します。
以下の行からコードのコメントを削除します。

```
bridge.tracing=jaeger
```

2. 設定ファイルを作成します。
3. 設定プロパティをパラメーターとして使用し、**bin/kafka_bridge_run.sh** スクリプトを実行します。

```
cd kafka-bridge-0.xy.x.redhat-0000x  
./bin/kafka_bridge_run.sh --config-file=config/application.properties
```

Kafka Bridge の内部コンシューマーおよびプロデューサーのインターセプターが有効になりました。

15.5. トレースの環境変数

Kafka クライアントおよびコンポーネントに Jaeger トレーサーを設定する場合は、これらの環境変数を使用します。



注記

トレーシング環境変数は Jaeger プロジェクトの一部で、変更される場合があります。最新の環境変数については、[Jaeger のドキュメント](#) を参照してください。

表15.2 Jaeger トレーサーの環境変数

プロパティ	必要性	説明
JAEGER_SERVICE_NAME	必要	Jaeger トレーサーサービスの名前。
JAEGER_AGENT_HOST	不要	UDP (User Datagram Protocol) を介した jaeger-agent との通信のためのホスト名。
JAEGER_AGENT_PORT	不要	UDP を介した jaeger-agent との通信に使用されるポート。
JAEGER_ENDPOINT	不要	traces エンドポイント。クライアントアプリケーションが jaeger-agent を迂回し、 jaeger-collector に直接接続する場合にのみ、この変数を定義します。
JAEGER_AUTH_TOKEN	不要	エンドポイントに bearer トークンとして送信する認証トークン。
JAEGER_USER	不要	Basic 認証を使用する場合にエンドポイントに送信するユーザー名。
JAEGER_PASSWORD	不要	Basic 認証を使用する場合にエンドポイントに送信するパスワード。
JAEGER_PROPAGATION	不要	トレースコンテキストの伝播に使用するコンマ区切りの形式リスト。デフォルトは標準の Jaeger 形式です。有効な値は jaeger および b3 です。
JAEGER_REPORTER_LOG_SPANS	不要	レポーターがスパンも記録する必要があるかどうかを示します。
JAEGER_REPORTER_MAX_QUEUE_SIZE	不要	レポーターの最大キューサイズ。
JAEGER_REPORTER_FLUSH_INTERVAL	不要	レポーターのフラッシュ間隔 (ミリ秒単位)。Jaeger レポーターがスパンバッチをフラッシュする頻度を定義します。

プロパティ	必要性	説明
JAEGER_SAMPLER_TYPE	不要	<p>クライアントトレースに使用するサンプリングストラテジー。</p> <ul style="list-style-type: none"> ● Constant ● Probabilistic ● Rate Limiting ● Remote (デフォルト) <p>すべてのトレースをサンプリングするには、Constant サンプリングストラテジーを使用し、パラメーターを1にします。</p> <p>Jaeger アーキテクチャーおよびクライアントサンプリング設定パラメーターの概要は、Jaeger のドキュメント を参照してください。</p>
JAEGER_SAMPLER_PARAM	不要	<p>サンプラーのパラメーター (数値)。</p>
JAEGER_SAMPLER_MANAGER_HOST_PORT	不要	<p>リモートサンプリングストラテジーを選択する場合に使用するホスト名およびポート。</p>
JAEGER_TAGS	不要	<p>報告されたすべてのスパンに追加されるトレーサーレベルのタグのコンマ区切りリスト。</p> <p>また、<code>\${envVarName:default}</code> という形式で環境変数を参照することもできます。:default は任意の設定で、環境変数が見つからない場合に使用する値を特定します。</p>

第16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を実現するために AMQ Streams で提供され、オフセット、コンシューマーグループ、コンシューマーラグ、およびトピックに関連する Kafka ブローカーから追加のメトリクスデータを抽出します。

一例として、メトリクスデータを使用すると、低速なコンシューマーの識別に役立ちます。

ラグデータは Prometheus メトリクスとして公開され、解析のために Grafana で使用できます。

ビルトイン Kafka メトリクスの監視のために Prometheus および Grafana をすでに使用している場合、Kafka Exporter Prometheus エンドポイントをスクレープするように Prometheus を設定することもできます。

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。

16.1. コンシューマーラグ

コンシューマーラグは、メッセージの生成と消費の差を示しています。具体的には、指定のコンシューマーグループのコンシューマーラグは、パーティションの最後のメッセージと、そのコンシューマーが現在ピックアップしているメッセージとの時間差を示しています。ラグには、パーティションログの最後を基準とする、コンシューマーオフセットの相対的な位置が反映されます。

この差は、Kafka ブローカートピックパーティションの読み取りと書き込みの場所である、プロデューサーオフセットとコンシューマーオフセットの間の **デルタ** とも呼ばれます。

あるトピックで毎秒 100 個のメッセージがストリーミングされる場合を考えてみましょう。プロデューサーオフセット (トピックパーティションの先頭) と、コンシューマーが読み取った最後のオフセットとの間のラグが 1000 個のメッセージであれば、10 秒の遅延があることを意味します。

コンシューマーラグ監視の重要性

可能な限りリアルタイムのデータの処理に依存するアプリケーションでは、コンシューマーラグを監視して、ラグが過度に大きくならないようにチェックする必要があります。ラグが大きくなるほど、リアルタイム処理の達成から遠ざかります。

たとえば、パージされていない古いデータの大量消費や、予定外のシャットダウンが、コンシューマーラグの原因となることがあります。

コンシューマーラグの削減

通常、ラグを削減するには以下を行います。

- 新規コンシューマーを追加してコンシューマーグループをスケールアップします。
- メッセージがトピックに留まる保持時間を延長します。
- ディスク容量を追加してメッセージバッファを増強します。

コンシューマーラグを減らす方法は、基礎となるインフラストラクチャーや、AMQ Streams によりサポートされるユースケースによって異なります。たとえば、ラグが生じているコンシューマーの場合、ディスクキャッシュからフェッチリクエストに対応できるブローカーを活用できる可能性は低いでしょう。

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。

16.2. KAFKA EXPORTER アラートルールの例

Kafka Exporter に固有のサンプルのアラート通知ルールには以下があります。

UnderReplicatedPartition

トピックで複製の数が最低数未満であり、ブローカーがパーティションで十分な複製を作成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、トピックに複製の数が最低数未満のパーティションが1つ以上ある場合のアラートになります。このアラートは、Kafka インスタンスがダウンしているか Kafka クラスターがオーバーロードの状態であることを示す場合があります。レプリケーションプロセスを再起動するには、Kafka ブローカーの計画的な再起動が必要な場合があります。

TooLargeConsumerGroupLag

特定のトピックパーティションでコンシューマーグループのラグが大きすぎることを警告するアラートです。デフォルト設定は1000 レコードです。ラグが大きい場合、コンシューマーが遅すぎてプロデューサーの処理に追いついていない可能性があります。

NoMessageForTooLong

トピックが一定期間にわたりメッセージを受信していないことを警告するアラートです。この期間のデフォルト設定は10 分です。この遅れは、設定の問題により、プロデューサーがトピックにメッセージを公開できないことが原因である可能性があります。

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。

16.3. KAFKA EXPORTER メトリクス

ラグ情報は、Grafana で示す Prometheus メトリクスとして Kafka Exporter によって公開されます。

Kafka Exporter は、ブローカー、トピック、およびコンシューマーグループのメトリクスデータを公開します。

表16.1 ブローカーメトリクスの出力

名前	詳細
<code>kafka_brokers</code>	Kafka クラスターに含まれるブローカーの数

表16.2 トピックメトリクスの出力

名前	詳細
<code>kafka_topic_partitions</code>	トピックのパーティション数

名前	詳細
<code>kafka_topic_partition_current_offset</code>	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
<code>kafka_topic_partition_oldest_offset</code>	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
<code>kafka_topic_partition_in_sync_replica</code>	トピックパーティションの In-Sync レプリカ数
<code>kafka_topic_partition_leader</code>	トピックパーティションのリーダーブローカー ID
<code>kafka_topic_partition_leader_is_preferred</code>	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。
<code>kafka_topic_partition_replicas</code>	このトピックパーティションのレプリカ数
<code>kafka_topic_partition_under_replicated_partition</code>	トピックパーティションの複製の数が最低数未満である場合に 1 が示されます。

表16.3 コンシューマーグループメトリクスの出力

名前	詳細
<code>kafka_consumergroup_current_offset</code>	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
<code>kafka_consumergroup_lag</code>	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)

16.4. 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 メトリックを公開します。

Kafka Exporter パッケージをダウンロードしてインストールし、AMQ Streams 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、AMQ Streams サブスクリプションが必要です。

前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [AMQ Streams](#) へのサブスクリプションがあります

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=3.2.3 -  
--<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16.4 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/ポートアドレス。	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	メトリクスを公開するポートアドレス。	:9308
web.telemetry-path	公開されるメトリクスのパス。	/metrics

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 ロギングを有効にするブール値 (Kafka Exporter によって使用される Go クライアントライブラリー)。	false
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クス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クスを返すようにするには、 true に設定します。	false

プロパティの詳細は、**kafka_exporter --help** を使用できます。

4. Kafka Exporter メトリクスを監視するように Prometheus を設定します。
Prometheus の設定に関する詳細は、[Prometheus のドキュメント](#) を参照してください。
5. Grafana を有効にして、Prometheus によって公開される Kafka Exporter メトリクスデータを表示します。
詳細は、[Grafana での Kafka Exporter メトリクスの表示](#) を参照してください。

Kafka Exporter の更新

AMQ Streams 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
```

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

Kafka Exporter Prometheus メトリクスをデータソースとして使用すると、Grafana チャートのダッシュボードを作成できます。

たとえば、メトリクスから、以下の Grafana チャートを作成できます。

- 毎秒のメッセージ (トピックから)
- 毎分のメッセージ (トピックから)

- `コンシューマーグループごとのラグ`
- `毎分のメッセージ消費 (コンシューマーグループごと)`

メトリクスデータが収集されると、Kafka Exporter のチャートにデータが反映されます。

Grafana のチャートを使用して、ラグを分析し、ラグ削減の方法が対象のコンシューマーグループに影響しているかどうかを確認します。たとえば、ラグを減らすように Kafka ブローカーを調整すると、ダッシュボードには `コンシューマーグループごとのラグ` のチャートが下降し `毎分のメッセージ消費` のチャートが上昇する状況が示されます。

関連情報

- [Example dashboard for Kafka Exporter](#)
- [Grafana のドキュメント](#)

第17章 AMQ STREAMS および KAFKA のアップグレード

AMQ Streams は、クラスターのダウンタイムを発生せずにアップグレードできます。AMQ Streams の各バージョンは、Apache Kafka の1つ以上のバージョンをサポートします。使用する AMQ Streams バージョンでサポートされれば、より高いバージョンの Kafka にアップグレードできます。より新しいバージョンの AMQ Streams はより新しいバージョンの Kafka をサポートしますが、AMQ Streams をアップグレードしてから、サポートされる上位バージョンの Kafka にアップグレードする必要があります。



注記

特定バージョンの AMQ Streams へのアップグレード方法については、そのバージョンをサポートするドキュメントを参照してください。

17.1. アップグレードの前提条件

アップグレードプロセスを開始する前に、以下を確認します。

- AMQ Streams がインストールされている。手順は、[2章 スタートガイド](#)を参照してください。
- [AMQ Streams 2.2 on Red Hat Enterprise Linux Release Notes](#) で説明されているアップグレードの変更を理解している。

17.2. KAFKA バージョン

Kafka のログメッセージ形式バージョンと inter-broker プロトコルバージョンは、それぞれメッセージに追加されるログ形式バージョンとクラスターで使用される Kafka プロトコルのバージョンを指定します。正しいバージョンが使用されるようにするため、アップグレードプロセスでは、既存の Kafka ブローカーの設定変更と、クライアントアプリケーション (コンシューマーおよびプロデューサー) のコード変更が行われます。

以下の表は、Kafka バージョンの違いを示しています。

表17.1 Kafka バージョンの相違点

Kafka バージョン	Inter-broker プロトコルバージョン	ログメッセージ形式バージョン	ZooKeeper バージョン
3.2.3	3.2	3.2	3.6.3
3.1.0	3.1	3.1	3.6.3

Inter-broker プロトコルバージョン

Kafka では、Inter-broker の通信に使用されるネットワークプロトコルは **Inter-broker プロトコル** と呼ばれます。Kafka の各バージョンには、互換性のあるバージョンの Inter-broker プロトコルがあります。上記の表が示すように、プロトコルのマイナーバージョンは、通常 Kafka のマイナーバージョンと一致するように番号が増加されます。

Inter-broker プロトコルのバージョンは、**Kafka** リソースでクラスター全体に設定されます。これを変更するには、**Kafka.spec.kafka.config** の **inter.broker.protocol.version** プロパティを編集します。

ログメッセージ形式バージョン

プロデューサーが Kafka ブローカーにメッセージを送信すると、特定の形式を使用してメッセージがエンコードされます。この形式は Kafka のリリース間で変更される可能性があるため、メッセージにはエンコードに使用されたメッセージ形式のバージョンが指定されます。

特定のメッセージ形式のバージョンを設定するために使用されるプロパティは以下のとおりです。

- トピック用の **message.format.version** プロパティ
- Kafka ブローカーの **log.message.format.version** プロパティ

Kafka 3.0.0 以降、メッセージ形式のバージョンの値は **inter.broker.protocol.version** と一致すると見なされ、設定する必要はありません。値は、使用される Kafka バージョンを反映します。

Kafka 3.0.0 以降にアップグレードする場合、**inter.broker.protocol.version** を更新する際にこれらの設定を削除できます。それ以外の場合は、アップグレード先の Kafka バージョンに基づいてメッセージ形式のバージョンを設定します。

トピックの **message.format.version** のデフォルト値は、Kafka ブローカーに設定される **log.message.format.version** によって定義されます。トピックの **message.format.version** は、トピック設定を編集すると手動で設定できます。

17.3. KAFKA ブローカーおよび ZOOKEEPER のアップグレード

この手順では、最新バージョンの AMQ Streams を使用するように、ホストマシンで Kafka ブローカーおよび ZooKeeper をアップグレードする方法を説明します。

ファイルを更新してから、すべての Kafka ブローカーを設定して再起動し、新しい inter-broker プロトコルバージョンを使用します。これらの手順の実行後に、新しい inter-broker プロトコルバージョンを使用して Kafka ブローカー間でデータが送信されます。



注記

Kafka 3.0.0 以降、メッセージ形式のバージョン値は **inter.broker.protocol.version** と一致することが想定されており、これを設定する必要はありません。値は、使用される Kafka バージョンを反映します。

受信したメッセージは、以前のメッセージ形式のバージョンでメッセージログに追加されます。

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。

手順

AMQ Streams クラスターの各 Kafka ブローカーに対して、以下を1つずつ行います。

1. [AMQStreams ソフトウェアのダウンロードページ](#)から AMQStreams 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-x.y.z-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-x.y.z-bin.zip -d /tmp/kafka
```

3. 実行中の場合は、ホストで実行している ZooKeeper および Kafka ブローカーを停止します。

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

4. 既存のインストールから **libs** 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_y.y-x.x.x/libs /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
```

6. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

7. **/opt/kafka/config/server.properties** プロパティファイルを編集します。
inter.broker.protocol.version および **log.message.format.version** プロパティを **現在のバージョン**に設定します。

たとえば、Kafka バージョン 3.1.0 から 3.2.3 にアップグレードする場合、現在のバージョンは 3.1 になります。

```
inter.broker.protocol.version=3.1
log.message.format.version=3.1
```

アップグレード元の Kafka バージョン (**3.0**、**3.1** など) の正しいバージョンを使用してください。**inter.broker.protocol.version** を現在の設定のままにしておくと、ブローカーはアップグレード中に相互に通信し続けることができます。

プロパティが設定されていない場合は、現在のバージョンでプロパティを追加します。

Kafka 3.0.0 以降からアップグレードする場合は、**inter.broker.protocol.version** を設定するだけで済みます。

8. 更新された ZooKeeper および Kafka ブローカーを再起動します。

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

Kafka ブローカーおよび Zookeeper は、最新の Kafka バージョンのバイナリーの使用を開始します。

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. 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ます。

kafka-topics.sh ツールを使用して、ブローカーに含まれるすべてのレプリカが同期していることを確認します。手順は、[トピックの一覧表示および説明](#) を参照してください。

次の手順では、新しい inter-broker プロトコルバージョンを使用するように Kafka ブローカーを更新します。

各ブローカーを一度に1つずつ更新します。



警告

次の手順を完了した後は、AMQ ストリームをダウングレードすることはできません。

10. **/opt/kafka/config/server.properties** プロパティファイルで **inter.broker.protocol.version** プロパティを **3.2** に設定します。

```
inter.broker.protocol.version=3.2
```

11. コマンドラインで、変更した Kafka ブローカーを停止します。

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

12. Kafka が実行されていないことを確認します。

```
jcmd | grep kafka
```

13. 変更した Kafka ブローカーを再起動します。

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

14. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

15. Kafka 3.0.0 より前のバージョンからアップグレードする場合は、**/opt/kafka/config/server.properties** プロパティファイルで **log.message.format.version** プロパティを **3.2** に設定します。

```
log.message.format.version=3.2
```

16. コマンドラインで、変更した Kafka ブローカーを停止します。

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

17. Kafka が実行されていないことを確認します。

```
jcmd | grep kafka
```

18. 変更した Kafka ブローカーを再起動します。

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

19. Kafka が稼働していることを確認します。

```
jcmd | grep kafka
```

20. 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ます。

kafka-topics.sh ツールを使用して、ブローカーに含まれるすべてのレプリカが同期していることを確認します。手順は、[トピックの一覧表示および説明](#) を参照してください。

17.4. KAFKA CONNECT のアップグレード

この手順では、ホストマシンで Kafka Connect クラスターをアップグレードする方法を説明します。

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。
- Kafka Connect が起動していない。

手順

AMQ Streams クラスターの各 Kafka ブローカーに対して、以下を1つずつ行います。

1. [AMQStreams ソフトウェアのダウンロードページ](#)から AMQStreams 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-x.y.z-bin.zip** ファイルの内容を展開します。

```
mkdir /tmp/kafka
unzip amq-streams-x.y.z-bin.zip -d /tmp/kafka
```

3. 実行中の場合は、ホストで実行している Kafka ブローカーおよび ZooKeeper を停止します。

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

4. 既存のインストールから **libs** 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_y.y-x.x.x/libs /opt/kafka/  
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
```

6. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

7. スタンドアロンまたは分散モードのいずれかで Kafka Connect を起動します。

- スタンドアロンモードで起動するには、**connect-standalone.sh** スクリプトを実行します。Kafka Connect スタンドアロン設定ファイルおよびご使用中の Kafka Connect コネクターの設定ファイルを指定します。

```
su - kafka  
/opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties  
connector1.properties  
[connector2.properties ...]
```

- 分散モードで起動するには、すべての Kafka Connect ノードで **/opt/kafka/config/connect-distributed.properties** 設定ファイルで Kafka Connect ワーカーを起動します。

```
su - kafka  
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

8. KafkaConnect が実行されていることを確認します。

- スタンドアロンモードの場合:

```
jcmd | grep ConnectStandalone
```

- 分散モードの場合:

```
jcmd | grep ConnectDistributed
```

9. Kafka Connect が想定どおりにデータを生成および消費していることを確認します。

関連情報

- [スタンドアロンモードでの Kafka Connect の実行](#)
- [分散 Kafka Connect の実行](#)

17.5. コンシューマーおよび KAFKA STREAMS アプリケーションの COOPERATIVE REBALANCING へのアップグレード

Kafka のアップグレードに続いて、必要に応じて、Kafka コンシューマーと Kafka Streams アプリケーションをアップグレードして、デフォルトの **eager rebalance** プロトコルの代わりに、パーティションリバランスに **incremental cooperative rebalance** プロトコルを使用できます。この新しいプロトコルが Kafka 2.4.0 に追加されました。

コンシューマーは、パーティションの割り当てを Cooperative Rebalance で保持し、クラスターの分散が必要な場合にプロセスの最後でのみ割り当てを取り消します。これにより、コンシューマーグループまたは Kafka Streams アプリケーションが使用不可能になる状態が削減されます。



注記

Incremental Cooperative Rebalance プロトコルへのアップグレードは任意です。Eager Rebalance プロトコルは引き続きサポートされます。

前提条件

- [Kafka ブローカーをアップグレードしている](#)。

手順

Incremental Cooperative Rebalance プロトコルを使用するように Kafka コンシューマーをアップグレードする方法:

1. Kafka クライアント **.jar** ファイルを新バージョンに置き換える。
2. コンシューマー設定で、**partition.assignment.strategy** に **cooperative-sticky** を追加する。たとえば、**range** ストラテジーが設定されている場合は、設定を **range, cooperative-sticky** に変更する。
3. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待つ。
4. コンシューマー設定から前述の **partition.assignment.strategy** を削除して、グループの各コンシューマーを再設定し、**cooperative-sticky** ストラテジーのみを残す。
5. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待つ。

Incremental Cooperative Rebalance プロトコルを使用するように Kafka Streams アプリケーションをアップグレードするには以下を行います。

1. Kafka Streams の **.jar** ファイルを新バージョンに置き換えます。
2. Kafka Streams の設定で、**upgrade.from** 設定パラメーターをアップグレード前の Kafka バージョンに設定します (例: 2.3)。
3. 各ストリームプロセッサ (ノード) を順次再起動します。
4. **upgrade.from** 設定パラメーターを Kafka Streams 設定から削除します。
5. グループ内の各コンシューマーを順次再起動します。

第18章 JMX を使用したクラスターの監視

Zoo Keeper、Kafka ブローカー、Kafka Connect、および Kafka クライアントはすべて、[Java Management Extensions \(JMX\)](#) を使用して管理情報を公開します。管理情報の多くは、Kafka クラスターの状態やパフォーマンスを監視するのに役立つメトリクスの形式になっています。他の Java アプリケーションと同様に、Kafka は管理対象 Bean または MBean を介してこの管理情報を提供します。

JMX は、JVM (Java 仮想マシン) のレベルで動作します。管理情報を取得するために、外部ツールは ZooKeeper、Kafka ブローカーなどを実行している JVM に接続できます。デフォルトでは、同じマシン上で、JVM と同じユーザーとして実行しているツールのみが接続できます。



注記

ZooKeeper の管理情報は、ここには記載されていません。JConsole で ZooKeeper メトリクスを表示できます。詳細は、[JConsole を使用した監視](#) を参照してください。

18.1. JMX 設定オプション

JVM システムプロパティを使用して JMX を設定します。AMQ Streams とともに提供されるスクリプト (`bin/kafka-server-start.sh`、`bin/connect-distributed.sh` など) では、環境変数 **KAFKA_JMX_OPTS** を使用してこれらのシステムプロパティを設定しています。Kafka プロデューサー、コンシューマー、およびストリームアプリケーションは、通常、異なる方法で JVM を起動しますが、JMX を設定するためのシステムプロパティは同じです。

18.2. JMX エージェントの無効化

AMQ Streams コンポーネントの JMX エージェントを無効にすることで、ローカルの JMX ツールが (たとえば、コンプライアンスの理由などで) JVM に接続しないようにすることができます。以下の手順では、Kafka ブローカーの JMX エージェントを無効にする方法を説明します。

手順

1. **KAFKA_JMX_OPTS** 環境変数を使用して **com.sun.management.jmxremote** を **false** に設定します。

```
export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=false
bin/kafka-server-start.sh
```

2. JVM を起動します。

18.3. 別のマシンからの JVM への接続

JMX エージェントがリッスンするポートを設定すると、別のマシンから JVM に接続できます。これは、JMX ツールがどこからでも認証なしで接続できるため、安全ではありません。

手順

1. **KAFKA_JMX_OPTS** 環境変数を使用して **-Dcom.sun.management.jmxremote.port=<port>** を設定します。**<port>** には、Kafka ブローカーが JMX 接続をリッスンするポートの名前を入力します。

```
export KAFKA_JMX_OPTS="-Dcom.sun.management.jmxremote=true
-Dcom.sun.management.jmxremote.port=<port>
```

```
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
bin/kafka-server-start.sh
```

2. JVM を起動します。



重要

リモート JMX 接続をセキュアにするには、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の詳細については、[JMX のドキュメント](#)を参照してください。

18.4. JCONSOLE を使用した監視

JConsole ツールは Java Development Kit (JDK) とともに配布されます。JConsole を使用して、ローカルまたはリモート JVM に接続し、Java アプリケーションから管理情報を検出および表示できます。JConsole を使用してローカル JVM に接続する場合には、JVM プロセスの名前は AMQ Streams コンポーネントに対応します。

表18.1 AMQ Streams コンポーネントの JVM プロセス

AMQ Streams コンポーネント	JVM プロセス
ZooKeeper	org.apache.zookeeper.server.quorum.QuorumPeerMain
Kafka ブローカー	kafka.Kafka
Kafka Connect スタンドアロン	org.apache.kafka.connect.cli.ConnectStandalone
Kafka Connect distributed	org.apache.kafka.connect.cli.ConnectDistributed
Kafka producer、consumer、または Streams application	アプリケーションの main メソッドが含まれるクラスの名前。

JConsole を使用してリモート JVM に接続する場合は、適切なホスト名と JMX ポートを使用します。

その他の多くのツールおよびモニターリング製品を使用して JMX を使用してメトリクスを取得し、そのメトリクスに基づいてモニターリングおよびアラートを提供できます。これらのツールについては、製品ドキュメントを参照してください。

18.5. 重要な KAFKA ブローカーメトリクス

Kafka は、Kafka クラスターのブローカーのパフォーマンスを監視するための多くの MBean を提供します。これらは、クラスター全体ではなく、個別のブローカーに適用されます。

以下の表は、サーバー、ネットワーク、ロギング、およびコントローラーメトリクスに編成されるこれらのブローカーレベルの MBean の一部です。

18.5.1. Kafka サーバーメトリクス

以下の表は、Kafka サーバーに関する情報を報告するメトリクスの一部です。

表18.2 Kafka サーバーのメトリクス

メトリクス	MBean	説明	想定される値
1秒あたりのメッセージ数	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec	個々のメッセージがブローカーによって消費されるレート。	クラスターの他のブローカーとほぼ同じです。
1秒あたりのバイト数	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec	プロデューサーから送信されたデータがブローカーによって消費されるレート。	クラスターの他のブローカーとほぼ同じです。
1秒あたりのレプリケーションバイト数	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec	他のブローカーから送信されたデータがフォロワーブローカーによって消費されるレート。	該当なし
1秒あたりのバイト数	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec	コンシューマーによってブローカーからデータを取得および読み取るレート。	該当なし
1秒あたりのレプリケーションバイト数	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec	ブローカーから他のブローカーにデータを送信するレート。このメトリクスは、ブローカーがパーティションのグループのリーダーであるかどうかを監視するのに役立ちます。	該当なし
複製の数が最低数未満パーティション	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	フォロワーレプリカに完全にレプリケートされていないパーティションの数。	ゼロ
最小 ISR パーティション数	kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount	最小の In-Sync Replica (ISR) カウント下のパーティションの数。ISR 数は、リーダーと最新の状態にあるレプリカのセットを示します。	ゼロ
パーティションの数	kafka.server:type=ReplicaManager,name=PartitionCount	ブローカーのパーティション数。	他のブローカーと比較してほぼ同じです。

メトリクス	MBean	説明	想定される値
リーダー数	kafka.server:type=ReplicaManager,name=LeaderCount	このブローカーがリーダーであるレプリカの数。	クラスターの他のブローカーとほぼ同じです。
ISR は1秒あたりに縮小します	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	ブローカー内の ISR の数が減少する割合	ゼロ
1秒あたりの ISR 拡張	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	ブローカー内の ISR の数が増大する割合	ゼロ
最大ラグ	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	メッセージがリーダーレプリカとフォロワーレプリカによって受信される時間の間の最大ラグ。	生成リクエストの最大バッチサイズに比例します。
producer purgatory での要求	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Produce	producer purgatory の送信リクエストの数。	該当なし
fetch purgatory での要求	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Fetch	fetch purgatory のフェッチリクエストの数。	該当なし
リクエストハンドラーの平均アイドル率	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent	リクエストハンドラー (IO) スレッドが使用されていない時間の割合を示します。	値が小さいほど、ブローカーのワークロードが高いことを示します。
リクエスト (スロットルを除外されるリクエスト)	kafka.server:type=Request	スロットリングから除外される要求の数。	該当なし
Zoo Keeper リクエストのレイテンシー (ミリ秒)	kafka.server:type=ZooKeeperClientMetrics,name=ZooKeeperRequestLatencyMs	ブローカーからの Zoo Keeper リクエストのレイテンシー (ミリ秒単位)。	該当なし
ZooKeeper セッションの状態	kafka.server:type=SessionExpireListener,name=SessionState	ブローカーの ZooKeeper への接続状態。	接続済み

18.5.2. Kafka ネットワークメトリクス

以下の表は、リクエストに関する情報を報告するメトリクスの一部です。

メトリクス	MBean	説明	想定される値
1秒あたりのリクエスト数	<code>kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce FetchConsumer FetchFollower}</code>	1秒あたりの要求タイプに対して行われるリクエストの合計数。 Produce 、 FetchConsumer 、 FetchFollower 要求タイプにはそれぞれ独自の MBean があります。	該当なし
リクエストバイト (バイト単位のリクエストサイズ)	<code>kafka.network:type=RequestMetrics,name=RequestBytes,request=[-.lw]+)</code>	MBean 名の request プロパティで識別されるリクエストタイプに対して行われたリクエストのサイズ (バイト単位)。 RequestBytes ノードの下には、利用可能なすべてのリクエストタイプの個別の MBean が表示されます。	該当なし
バイト単位の一時的メモリーサイズ	<code>kafka.network:type=RequestMetrics,name=TemporaryMemoryBytes,request={Produce Fetch}</code>	メッセージ形式の変換およびメッセージの展開に使用される一時メモリーの量。	該当なし
メッセージ変換時間	<code>kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce Fetch}</code>	メッセージ形式の変換に費やされた時間 (ミリ秒単位)。	該当なし
ミリ秒単位の合計リクエスト時間	<code>kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce FetchConsumer FetchFollower}</code>	リクエストの処理に費やされた合計時間 (ミリ秒単位)。	該当なし
ミリ秒単位の要求キュー時間	<code>kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce FetchConsumer FetchFollower}</code>	request プロパティで指定されたリクエストタイプに対して、リクエストが現在キューで費やす時間 (ミリ秒単位)。	該当なし

メトリクス	MBean	説明	想定される値
ミリ秒単位の現地時間 (リーダーの現地処理時間)	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce FetchConsumer FetchFollower}	リーダーがリクエストを処理するのにかかる時間 (ミリ秒単位)。	該当なし
ミリ秒単位のリモート時間 (リーダーのリモート処理時間)	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce FetchConsumer FetchFollower}	要求がフォロワーを待機する時間の長さ (ミリ秒単位)。すべての利用可能な要求タイプの個別の MBean が RemoteTimeMs ノードの下に一覧表示されます。	該当なし
ミリ秒単位の応答キュー時間	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce FetchConsumer FetchFollower}	要求が応答キューで待機する時間の長さ (ミリ秒単位)。	該当なし
ミリ秒単位の応答送信時間	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce FetchConsumer FetchFollower}	応答の送信にかかった時間 (ミリ秒単位)。	該当なし
ネットワークプロセッサの平均アイドル率	kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent	ネットワークプロセッサがアイドル状態である時間の平均パーセンテージ。	0 から 1 の間。

18.5.3. Kafka ログメトリクス

次の表は、ロギングに関する情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	想定される値
ログのフラッシュ速度と時間 (ミリ秒)	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs	ログデータがディスクに書き込まれる速度 (ミリ秒単位)。	該当なし

メトリクス	MBean	説明	想定される値
オフラインのログディレクトリー数	kafka.log:type=LogManager,name=OfflineLogDirectoryCount	オフラインログディレクトリーの数 (たとえば、ハードウェア障害後)。	ゼロ

18.5.4. Kafka コントローラーメトリクス

次の表は、クラスターのコントローラーに関する情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	想定される値
アクティブなコントローラーの数	kafka.controller:type=KafkaController,name=ActiveControllerCount	コントローラーとして指定されるブローカーの数。	1つは、ブローカーがクラスターのコントローラーであることを示します。
リーダーエレクション率と時間 (ミリ秒)	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	新しいリーダーレプリカが選出されるレート。	ゼロ

18.5.5. Yammer メトリクス

レートまたは時間の単位を表すメトリクスは、Yammer メトリクスとして提供されます。Yammer メトリクスを使用する MBean のクラス名には、**com.yammer.metrics** という接頭辞がつきます。

Yammer レートメトリクスには、要求を監視する以下の属性があります。

- Count
- EventType (バイト)
- FifteenMinuteRate
- RateUnit (秒)
- MeanRate
- OneMinuteRate
- FiveMinuteRate

Yammer 時間メトリクスには、要求を監視するための以下の属性があります。

- Max
- Min
- Mean

- StdDev
- 75/95/98/99/99.9 パーセンタイル

18.6. プロデューサー MBEAN

MBean は、Kafka Streams アプリケーションやソースコネクタのある Kafka Connect などの Kafka プロデューサーアプリケーションに存在します。

プロデューサーメトリクス

表18.3 `kafka.producer:type=producer-metrics,client-id=*` に一致する MBeans

属性	説明
batch-size-avg	要求ごとにパーティションごとに送信されるバイトの平均数。
batch-size-max	リクエストごとおよびパーティションごとに送信されるバイトの最大数。
batch-split-rate	1秒あたりのバッチスプリットの平均数。
batch-split-total	バッチスプリットの合計数。
buffer-available-bytes	使用されていない (未割り当てまたは空きリストにある) バッファメモリーの合計量。
buffer-total-bytes	クライアントが使用できるバッファメモリーの最大量 (現在使用されているかどうかに関係なく)。
bufferpool-wait-time	アペンダーがスペースの割り当てを待つ時間の割合。
bufferpool-wait-time-ns-total	アペンダーがスペースの割り当てをナノ秒単位で待機する合計時間。
bufferpool-wait-time-total	非推奨: アペンダーがスペースの割り当てをナノ秒で待機する合計時間。bufferpool-wait-time-ns-total を代わりに使用してください。
compression-rate-avg	レコードバッチの平均圧縮レート。圧縮バッチサイズを非圧縮サイズでの平均比率として定義されます。
connection-close-rate	ウィンドウで1秒間に閉じられる接続。
connection-close-total	ウィンドウで切断された合計接続数。
connection-count	アクティブな接続の現在の数。

属性	説明
connection-creation-rate	ウィンドウで1秒間に確立される新しい接続。
connection-creation-total	ウィンドウで確立された合計新規接続数。
failed-authentication-rate	1秒あたりに認証に失敗した接続回数。
failed-authentication-total	認証に失敗した接続合計数。
failed-reauthentication-rate	1秒あたりに再認証に失敗した接続回数。
failed-reauthentication-total	再認証に失敗した接続の合計。
incoming-byte-rate	すべてのソケットから読み取られたバイト/秒。
incoming-byte-total	すべてのソケットから読み取られた合計バイト数。
io-ratio	I/O スレッドが I/O の実行に費やした時間の割合。
io-time-ns-avg	選択した呼び出しごとの I/O の平均時間 (ナノ秒単位)。
io-time-ns-total	I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	読み取りまたは書き込みの準備ができたソケットの待機に費やされた I/O スレッドの平均時間 (ナノ秒単位)。
io-wait-time-ns-total	I/O スレッドが待機した合計時間 (ナノ秒単位)。
io-waittime-total	非推奨: I/O スレッドが待機した合計時間 (ナノ秒単位)。io-wait-time-ns-total を代わりに使用してください。
iotime-total	非推奨: I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。io-time-ns-total を代わりに使用してください。
metadata-age	使用されている現在のプロデューサーメタデータの秒単位の経過時間。
network-io-rate	1秒あたりのすべての接続でのネットワーク操作 (読み取りまたは書き込み) の平均数。

属性	説明
network-io-total	全接続のネットワーク操作 (読み取りまたは書き込み) の合計数。
outgoing-byte-rate	すべてのサーバーに 1 秒間に送信する送信バイトの平均数。
outgoing-byte-total	すべてのサーバーに送信されるバイトの合計数。
produce-throttle-time-avg	リクエストがブローカーによって抑制された平均時間 (ミリ秒)。
produce-throttle-time-max	ブローカーによってリクエストがスロットルされた最大時間 (ミリ秒単位)。
reauthentication-latency-avg	再認証により発生した平均レイテンシー (ミリ秒単位)。
reauthentication-latency-max	再認証により発生した最大レイテンシー (ミリ秒単位)。
record-error-rate	エラーとなったレコード送信の 1 秒あたりの平均数。
record-error-total	エラーとなったレコード送信の合計数。
record-queue-time-avg	送信バッファで費やされたレコードバッチの平均時間 (ミリ秒)。
record-queue-time-max	送信バッファで費やされたレコードバッチの最長時間 (ミリ秒)。
record-retry-rate	再試行されたレコード送信の 1 秒あたりの平均数。
record-retry-total	再試行されたレコード送信の合計数。
record-send-rate	1 秒間に送信するレコードの平均数。
record-send-total	送信されたレコードの総数。
record-size-avg	平均レコードサイズ。
record-size-max	最大レコードサイズ。
records-per-request-avg	リクエストごとの平均レコード数。
request-latency-avg	ミリ秒単位の平均リクエストレイテンシー。

属性	説明
request-latency-max	ミリ秒単位の最大リクエストレイテンシー。
request-rate	1 秒間に送信するリクエストの平均数。
request-size-avg	ウィンドウのすべてのリクエストの平均サイズ。
request-size-max	ウィンドウの送信リクエストの最大サイズ。
request-total	送信された合計要求数。
requests-in-flight	応答を待機するインフライトリクエストの現在の数。
response-rate	1 秒間に受信した応答。
response-total	受信する応答の合計数。
select-rate	I/O レイヤーが実行する新しい I/O をチェックする 1 秒あたりの回数。
select-total	I/O レイヤーが実行する新しい I/O をチェックする合計回数。
successful-authentication-no-reauth-total	再認証をサポートしていない、2.2.0 より前の SASL クライアントによって正常に認証された接続の総数。ゼロ以外のみを指定できます。
successful-authentication-rate	SASL または SSL を使用して正常に認証された 1 秒あたりの接続。
successful-authentication-total	SASL または SSL を使用して正常に認証された合計接続数。
successful-reauthentication-rate	SASL を使用して正常に再認証された 1 秒あたりの接続。
successful-reauthentication-total	SASL を使用して正常に再認証された接続の合計。
waiting-threads	バッファメモリーがレコードをキューに入れるのを待機するブロックされたユーザースレッドの数。

ブローカー接続に関するプロデューサーメトリクス

表18.4 `kafka.producer:type=producer-metrics,client-id=*,node-id=*` に一致する MBeans

属性	説明
incoming-byte-rate	ノードが1秒間に受信した平均バイト数。
incoming-byte-total	ノードに受信するバイトの合計数。
outgoing-byte-rate	ノードが1秒間に送信する送信バイトの平均数。
outgoing-byte-total	ノードに送信される送信バイトの合計数。
request-latency-avg	ノードの平均リクエストレイテンシー (ミリ秒単位)。
request-latency-max	ノードの最大リクエストレイテンシー (ミリ秒単位)。
request-rate	ノードが1秒間に送信するリクエストの平均数。
request-size-avg	ノードのウィンドウにあるすべてのリクエストの平均サイズ。
request-size-max	ノードのウィンドウに送信されるリクエストの最大サイズ。
request-total	ノードに送信される要求の合計数。
response-rate	1秒間に受信されるノードの応答。
response-total	ノードで受信する応答の合計数。

トピックに送信されたメッセージに関するプロデューサーメトリクス

表18.5 `kafka.producer:type=producer-topic-metrics,client-id=*,topic=*` と一致する MBeans

属性	説明
byte-rate	トピックに対して1秒間に送信するバイトの平均数。
byte-total	トピックに対して送信するバイトの合計数。
compression-rate	トピックの記録バッチの平均圧縮レート。圧縮バッチサイズを非圧縮サイズで割った平均比率として定義されます。
record-error-rate	トピックに対してエラーとなったレコード送信の1秒あたりの平均数。

属性	説明
record-error-total	トピックに対してエラーとなったレコード送信の合計数。
record-retry-rate	トピックに対して再試行されたレコード送信の1秒あたりの平均数。
record-retry-total	トピックに対して再試行されたレコード送信の合計数。
record-send-rate	トピックに対して1秒間に送信するレコードの平均数。
record-send-total	トピックに対して送信するレコードの合計数。

18.7. コンシューマー MBEAN

MBean は、Kafka Streams アプリケーションやシンクコネクタのある Kafka Connect などの、Kafka コンシューマーアプリケーションに存在します。

コンシューマーメトリクス

表18.6 `kafka.consumer:type=consumer-metrics,client-id=*` と一致する MBeans

属性	説明
connection-close-rate	ウィンドウで1秒間に閉じられる接続。
connection-close-total	ウィンドウで切断された合計接続数。
connection-count	アクティブな接続の現在の数。
connection-creation-rate	ウィンドウで1秒間に確立される新しい接続。
connection-creation-total	ウィンドウで確立された合計新規接続数。
failed-authentication-rate	1秒あたりに認証に失敗した接続回数。
failed-authentication-total	認証に失敗した接続合計数。
failed-reauthentication-rate	1秒あたりに再認証に失敗した接続回数。
failed-reauthentication-total	再認証に失敗した接続の合計。
incoming-byte-rate	すべてのソケットから読み取られたバイト/秒。
incoming-byte-total	すべてのソケットから読み取られた合計バイト数。

属性	説明
io-ratio	I/O スレッドが I/O の実行に費やした時間の割合。
io-time-ns-avg	選択した呼び出しごとの I/O の平均時間 (ナノ秒単位)。
io-time-ns-total	I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	読み取りまたは書き込みの準備ができたソケットの待機に費やされた I/O スレッドの平均時間 (ナノ秒単位)。
io-wait-time-ns-total	I/O スレッドが待機した合計時間 (ナノ秒単位)。
io-waittime-total	非推奨: I/O スレッドが待機した合計時間 (ナノ秒単位)。io-wait-time-ns-total を代わりに使用してください。
iotime-total	非推奨: I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。io-time-ns-total を代わりに使用してください。
network-io-rate	1秒あたりのすべての接続でのネットワーク操作 (読み取りまたは書き込み) の平均数。
network-io-total	全接続のネットワーク操作 (読み取りまたは書き込み) の合計数。
outgoing-byte-rate	すべてのサーバーに1秒間に送信する送信バイトの平均数。
outgoing-byte-total	すべてのサーバーに送信されるバイトの合計数。
reauthentication-latency-avg	再認証により発生した平均レイテンシー (ミリ秒単位)。
reauthentication-latency-max	再認証により発生した最大レイテンシー (ミリ秒単位)。
request-rate	1秒間に送信するリクエストの平均数。
request-size-avg	ウィンドウのすべてのリクエストの平均サイズ。
request-size-max	ウィンドウの送信リクエストの最大サイズ。

属性	説明
request-total	送信された合計要求数。
response-rate	1 秒間に受信した応答。
response-total	受信する応答の合計数。
select-rate	I/O レイヤーが実行する新しい I/O をチェックする 1 秒あたりの回数。
select-total	I/O レイヤーが実行する新しい I/O をチェックする合計回数。
successful-authentication-no-reauth-total	再認証をサポートしていない、2.2.0 より前の SASL クライアントによって正常に認証された接続の総数。ゼロ以外のみを指定できます。
successful-authentication-rate	SASL または SSL を使用して正常に認証された 1 秒あたりの接続。
successful-authentication-total	SASL または SSL を使用して正常に認証された合計接続数。
successful-reauthentication-rate	SASL を使用して正常に再認証された 1 秒あたりの接続。
successful-reauthentication-total	SASL を使用して正常に再認証された接続の合計。

ブローカー接続に関するコンシューマーメトリクス

表18.7 `kafka.consumer:type=consumer-metrics,client-id=*,node-id=*` と一致する MBeans

属性	説明
incoming-byte-rate	ノードが 1 秒間に受信した平均バイト数。
incoming-byte-total	ノードに受信するバイトの合計数。
outgoing-byte-rate	ノードが 1 秒間に送信する送信バイトの平均数。
outgoing-byte-total	ノードに送信される送信バイトの合計数。
request-latency-avg	ノードの平均リクエストレイテンシー (ミリ秒単位)。
request-latency-max	ノードの最大リクエストレイテンシー (ミリ秒単位)。

属性	説明
request-rate	ノードが1秒間に送信するリクエストの平均数。
request-size-avg	ノードのウィンドウにあるすべてのリクエストの平均サイズ。
request-size-max	ノードのウィンドウに送信されるリクエストの最大サイズ。
request-total	ノードに送信される要求の合計数。
response-rate	1秒間に受信されるノードの応答。
response-total	ノードで受信する応答の合計数。

コンシューマーグループメトリクス

表18.8 `kafka.consumer:type=consumer-coordinator-metrics,client-id=*` と一致する MBeans

属性	説明
assigned-partitions	このコンシューマーに現在割り当てられているパーティションの数。
commit-latency-avg	コミットリクエストにかかる平均時間。
commit-latency-max	コミットリクエストにかかる最大時間。
commit-rate	1秒あたりのコミットコールの数。
commit-total	コミットコールの合計数。
failed-rebalance-rate-per-hour	1時間あたりの失敗したグループリバンスイベントの数。
failed-rebalance-total	失敗したグループリバンスの合計数。
heartbeat-rate	1秒あたりのハートビートの平均数。
heartbeat-response-time-max	ハートビート要求への応答を受信するのにかかる最大時間。
heartbeat-total	ハートビートの合計数。
join-rate	1秒あたりのグループ参加数。

属性	説明
join-time-avg	グループの再参加にかかる平均時間。
join-time-max	グループの再参加にかかる最大時間。
join-total	グループ参加の合計数。
last-heartbeat-seconds-ago	最後のコントローラーハートビートからの秒数。
last-rebalance-seconds-ago	最後のリバランスイベントからの経過時間 (秒単位)。
partitions-assigned-latency-avg	on-partitions-assigned リバランスリスナーのコールバックにかかる平均時間。
partitions-assigned-latency-max	on-partitions-assigned リバランスリスナーのコールバックにかかる最大時間。
partitions-lost-latency-avg	on-partitions-lost リバランスリスナーのコールバックにかかる平均時間。
partitions-lost-latency-max	on-partitions-lost リバランスリスナーコールバックにかかる最大時間。
partitions-revoked-latency-avg	on-partitions-revoked リバランスリスナーコールバックにかかる平均時間。
partitions-revoked-latency-max	on-partitions-revoked リバランスリスナーコールバックにかかる最大時間。
rebalance-latency-avg	グループのリバランスにかかる平均時間。
rebalance-latency-max	グループのリバランスにかかる最大時間。
rebalance-latency-total	これまでのグループのリバランスにかかった合計時間。
rebalance-rate-per-hour	1時間あたりに参加したグループリバランスの数。
rebalance-total	参加したグループリバランスの合計数。
sync-rate	1秒あたりのグループ同期数。
sync-time-avg	グループ同期にかかる平均時間。
sync-time-max	グループ同期にかかる最大時間。

属性	説明
sync-total	グループ同期の合計数。

コンシューマーフェッチメトリクス

表18.9 `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*` と一致する MBeans

属性	説明
bytes-consumed-rate	1秒あたり消費される平均のバイト数。
bytes-consumed-total	消費された総バイト数。
fetch-latency-avg	フェッチリクエストにかかる平均時間。
fetch-latency-max	任意のフェッチリクエストにかかる最大時間。
fetch-rate	1秒あたりのフェッチリクエストの数。
fetch-size-avg	リクエストごとにフェッチされたバイトの平均数。
fetch-size-max	リクエストごとにフェッチされたバイトの最大数。
fetch-throttle-time-avg	ミリ秒単位の平均スロットル時間。
fetch-throttle-time-max	ミリ秒単位の最大スロットル時間。
fetch-total	フェッチリクエストの総数。
records-consumed-rate	1秒あたり消費される平均のレコード数。
records-consumed-total	消費されるレコードの総数。
records-lag-max	このウィンドウの任意のパーティションのレコード数に関する最大ラグ。
records-lead-min	このウィンドウの任意のパーティションのレコード数に関する最小のリード。
records-per-request-avg	各リクエストの平均レコード数。

トピックレベルでのコンシューマーフェッチメトリクス

表18.10 `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*` と一致する MBeans

属性	説明
bytes-consumed-rate	トピックに対して1秒あたり消費される平均のバイト数。
bytes-consumed-total	トピックで消費された総バイト数。
fetch-size-avg	トピックに対してリクエストごとにフェッチされたバイトの平均数。
fetch-size-max	トピックに対してリクエストごとにフェッチされたバイトの最大数。
records-consumed-rate	トピックに対して1秒あたり消費される平均のレコード数。
records-consumed-total	トピックで消費されたレコードの合計数。
records-per-request-avg	トピックの各リクエストの平均レコード数。

パーティションレベルでのコンシューマーフェッチメトリクス

表18.11 `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*,partition=*` に一致する MBeans

属性	説明
preferred-read-replica	パーティションの現在の読み取りレプリカ。リーダーから読み取る場合は -1。
records-lag	パーティションの最新のラグ。
records-lag-avg	パーティションの平均ラグ。
records-lag-max	パーティションの最大ラグ数。
records-lead	パーティションの最新のリード。
records-lead-avg	パーティションの平均リード。
records-lead-min	パーティションの最小リード。

18.8. KAFKA CONNECT MBEAN



注記

Kafka Connect には、ここに記載されているものに加えて、ソースコネクタ用の [プロデューサー](#) MBean とシンクコネクタ用の [コンシューマー](#) MBean が含まれます。

Kafka Connect メトリクス

表18.12 kafka.connect:type=connect-metrics,client-id=* と一致する MBeans

属性	説明
connection-close-rate	ウィンドウで1秒間に閉じられる接続。
connection-close-total	ウィンドウで切断された合計接続数。
connection-count	アクティブな接続の現在の数。
connection-creation-rate	ウィンドウで1秒間に確立される新しい接続。
connection-creation-total	ウィンドウで確立された合計新規接続数。
failed-authentication-rate	1秒あたりに認証に失敗した接続回数。
failed-authentication-total	認証に失敗した接続合計数。
failed-reauthentication-rate	1秒あたりに再認証に失敗した接続回数。
failed-reauthentication-total	再認証に失敗した接続の合計。
incoming-byte-rate	すべてのソケットから読み取られたバイト/秒。
incoming-byte-total	すべてのソケットから読み取られた合計バイト数。
io-ratio	I/O スレッドが I/O の実行に費やした時間の割合。
io-time-ns-avg	選択した呼び出しごとの I/O の平均時間 (ナノ秒単位)。
io-time-ns-total	I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	読み取りまたは書き込みの準備ができたソケットの待機に費やされた I/O スレッドの平均時間 (ナノ秒単位)。
io-wait-time-ns-total	I/O スレッドが待機した合計時間 (ナノ秒単位)。
io-waittime-total	非推奨: I/O スレッドが待機した合計時間 (ナノ秒単位)。io-wait-time-ns-total を代わりに使用してください。

属性	説明
iotime-total	非推奨: I/O スレッドが I/O の実行に費やした合計時間 (ナノ秒単位)。io-time-ns-total を代わりに使用してください。
network-io-rate	1 秒あたりのすべての接続でのネットワーク操作 (読み取りまたは書き込み) の平均数。
network-io-total	全接続のネットワーク操作 (読み取りまたは書き込み) の合計数。
outgoing-byte-rate	すべてのサーバーに 1 秒間に送信する送信バイトの平均数。
outgoing-byte-total	すべてのサーバーに送信されるバイトの合計数。
reauthentication-latency-avg	再認証により発生した平均レイテンシー (ミリ秒単位)。
reauthentication-latency-max	再認証により発生した最大レイテンシー (ミリ秒単位)。
request-rate	1 秒間に送信するリクエストの平均数。
request-size-avg	ウィンドウのすべてのリクエストの平均サイズ。
request-size-max	ウィンドウの送信リクエストの最大サイズ。
request-total	送信された合計要求数。
response-rate	1 秒間に受信した応答。
response-total	受信する応答の合計数。
select-rate	I/O レイヤーが実行する新しい I/O をチェックする 1 秒あたりの回数。
select-total	I/O レイヤーが実行する新しい I/O をチェックする合計回数。
successful-authentication-no-reauth-total	再認証をサポートしていない、2.2.0 より前の SASL クライアントによって正常に認証された接続の総数。ゼロ以外のみを指定できます。

属性	説明
successful-authentication-rate	SASL または SSL を使用して正常に認証された 1 秒あたりの接続。
successful-authentication-total	SASL または SSL を使用して正常に認証された合計接続数。
successful-reauthentication-rate	SASL を使用して正常に再認証された 1 秒あたりの接続。
successful-reauthentication-total	SASL を使用して正常に再認証された接続の合計。

ブローカー接続に関する Kafka Connect メトリクス

表18.13 `kafka.connect:type=connect-metrics,client-id=*,node-id=*` と一致する MBeans

属性	説明
incoming-byte-rate	ノードが 1 秒間に受信した平均バイト数。
incoming-byte-total	ノードに受信するバイトの合計数。
outgoing-byte-rate	ノードが 1 秒間に送信する送信バイトの平均数。
outgoing-byte-total	ノードに送信される送信バイトの合計数。
request-latency-avg	ノードの平均リクエストレイテンシー (ミリ秒単位)。
request-latency-max	ノードの最大リクエストレイテンシー (ミリ秒単位)。
request-rate	ノードが 1 秒間に送信するリクエストの平均数。
request-size-avg	ノードのウィンドウにあるすべてのリクエストの平均サイズ。
request-size-max	ノードのウィンドウに送信されるリクエストの最大サイズ。
request-total	ノードに送信される要求の合計数。
response-rate	1 秒間に受信されるノードの応答。
response-total	ノードで受信する応答の合計数。

ワーカーに関する Kafka Connect メトリクス

表18.14 `kafka.connect.type=connect-worker-metrics` と一致する MBeans

属性	説明
<code>connector-count</code>	このワーカーで実行されるコネクタの数。
<code>connector-startup-attempts-total</code>	このワーカーが試みたコネクタ起動の合計数。
<code>connector-startup-failure-percentage</code>	このワーカーのコネクタ起動のうち、失敗したものの平均割合。
<code>connector-startup-failure-total</code>	失敗したコネクタ起動の総数。
<code>connector-startup-success-percentage</code>	このワーカーのコネクタ起動のうち、成功したものの平均割合。
<code>connector-startup-success-total</code>	成功したコネクタ起動の総数。
<code>task-count</code>	このワーカーで実行されるタスクの数。
<code>task-startup-attempts-total</code>	このワーカーが試みたタスク起動の合計数。
<code>task-startup-failure-percentage</code>	このワーカーのタスク起動のうち、失敗したものの平均割合。
<code>task-startup-failure-total</code>	失敗したタスク開始の合計数。
<code>task-startup-success-percentage</code>	このワーカーのタスク起動のうち、成功したものの平均割合。
<code>task-startup-success-total</code>	成功したタスク開始の合計数。

リバランスに関する Kafka Connect メトリクス

表18.15 `kafka.connect.type=connect-worker-rebalance-metrics` と一致する MBeans

属性	説明
<code>completed-rebalances-total</code>	このワーカーが完了したリバランスの合計数。
<code>connect-protocol</code>	このクラスターによって使用される Connect プロトコル。
<code>epoch</code>	このワーカーのエポックまたは生成番号。
<code>leader-name</code>	グループリーダーの名前。

属性	説明
rebalance-avg-time-ms	このワーカーがリバランスに費やした平均時間 (ミリ秒単位)。
rebalance-max-time-ms	このワーカーがリバランスするために費やした最大時間 (ミリ秒単位)。
rebalancing	このワーカーが現在リバランス中であるかどうか。
time-since-last-rebalance-ms	このワーカーが最新のリバランスを完了してからのミリ秒単位の時間。

コネクタに関する Kafka Connect メトリクス

表18.16 `kafka.connect:type=connector-metrics,connector=*` に一致する MBeans

属性	説明
connector-class	コネクタクラスの名前。
connector-type	コネクタのタイプ。ソースまたはシンクのいずれか。
connector-version	コネクタによって報告されるコネクタクラスのバージョン。
status	コネクタのステータス。unassigned、running、paused、failed、destroyed のいずれか。

コネクタタスクに関する Kafka Connect メトリクス

表18.17 `kafka.connect:type=connector-task-metrics,connector=*,task=*` と一致する MBeans

属性	説明
batch-size-avg	コネクタによって処理されるバッチの平均サイズ。
batch-size-max	コネクタによって処理されるバッチの最大サイズ。
offset-commit-avg-time-ms	このタスクがオフセットをコミットするのに要した平均時間 (ミリ秒単位)。
offset-commit-failure-percentage	このタスクのオフセットコミット試行のうち、失敗したものの平均割合。

属性	説明
offset-commit-max-time-ms	このタスクがオフセットをコミットするのにかかる最大時間 (ミリ秒単位)。
offset-commit-success-percentage	このタスクのオフセットコミット試行のうち、成功したものの平均割合。
pause-ratio	このタスクが pause 状態で費やした時間の割合。
running-ratio	このタスクが running 状態で費やした時間の割合。
status	コネクタタスクのステータス。unassigned、running、paused、failed、destroyed のいずれか。

シンクコネクターに関する Kafka Connect メトリクス

表18.18 `kafka.connect:type=sink-task-metrics,connector=*,task=*` と一致する MBeans

属性	説明
offset-commit-completion-rate	正常に完了したオフセットコミット完了の1秒あたりの平均数。
offset-commit-completion-total	正常に完了したオフセットコミット完了の合計数。
offset-commit-seq-no	オフセットコミットの現在のシーケンス番号。
offset-commit-skip-rate	受信が遅すぎてスキップ/無視されたオフセットコミット完了の1秒あたりの平均数。
offset-commit-skip-total	受信が遅すぎてスキップ/無視されたオフセットコミット完了の合計数。
partition-count	このワーカーの名前付きシンクコネクターに属するこのタスクに割り当てられたトピックパーティションの数。
put-batch-avg-time-ms	このタスクがシンクの記録を一括して行うためにかった平均時間。
put-batch-max-time-ms	このタスクがシンクの記録を一括して行うためにかった最大時間。
sink-record-active-count	Kafka から読み込まれたものの、シンクタスクがまだ完全にコミット、フラッシュ、確認していないレコードの数。

属性	説明
sink-record-active-count-avg	Kafka から読み込まれたものの、シンクタスクが完全にコミット、フラッシュ、確認していないレコードの平均数。
sink-record-active-count-max	Kafka から読み込まれたものの、シンクタスクが完全にコミット、フラッシュ、確認していないレコードの最大数。
sink-record-lag-max	任意のトピックパーティションで、シンクタスクがコンシューマーの位置の背後であるレコード数に関する最大ラグ。
sink-record-read-rate	このワーカーの名前付きシンクコネクタに属するこのタスクで、Kafka から読み取られるレコードの1秒あたりの平均数。これは、変換が適用される前に行われます。
sink-record-read-total	タスクが最後に再起動されてから、このワーカーの名前付きシンクコネクタに属するこのタスクによって Kafka から読み取られたレコードの合計数。
sink-record-send-rate	変換から出力されたレコードの1秒あたりの平均数。このワーカーの名前付きシンクコネクタに属するこのタスクに送信または配置します。これは、変換の適用後で、変換によってフィルターリングされたレコードをすべて除外します。
sink-record-send-total	タスクが最後に再起動されてから、このワーカーの名前付きシンクコネクタに属するこのタスクに送信または配置する、変換から出力されたレコードの合計数。

ソースコネクタに関する Kafka Connect メトリクス

表18.19 `kafka.connect:type=source-task-metrics,connector=*,task=*` と一致する MBeans

属性	説明
poll-batch-avg-time-ms	このタスクがソースレコードのバッチをポーリングするためにかかった平均時間 (ミリ秒単位)。
poll-batch-max-time-ms	このタスクがソースレコードのバッチをポーリングするためにかかった最大時間 (ミリ秒単位)。
source-record-active-count	このタスクによって生成され、まだ完全に Kafka に書き込まれていないレコードの数。

属性	説明
source-record-active-count-avg	このタスクによって生成され、まだ完全に Kafka に書き込まれていないレコードの平均数。
source-record-active-count-max	このタスクによって生成され、まだ完全に Kafka に書き込まれていないレコードの最大数。
source-record-poll-rate	このワーカーの名前付きソースコネクタに属するこのタスクによって生成/ポーリングされた (変換前) レコードの1秒あたりの平均数。
source-record-poll-total	このワーカーの名前付きソースコネクタに属するこのタスクによって生成/ポーリングされた (変換前) レコードの合計数。
source-record-write-rate	変換から出力され、このワーカーの名前付きソースコネクタに属するこのタスクの Kafka に書き込まれたレコードの1秒あたりの平均数。これは、変換の適用後で、変換によってフィルターリングされたレコードをすべて除外します。
source-record-write-total	タスクが最後に再起動されてから、変換から出力され、このワーカーの名前付きソースコネクタに属するこのタスクの Kafka に書き込まれたレコードの数。
transaction-size-avg	タスクがこれまでにコミットしたトランザクション内のレコードの平均数。
transaction-size-max	タスクがコミットした最大トランザクション内のレコード数。
transaction-size-min	タスクがコミットした最小トランザクション内のレコード数。

コネクタエラーに関する Kafka Connect メトリクス

表18.20 `kafka.connect:type=task-error-metrics,connector=*,task=*` と一致する MBeans

属性	説明
deadletterqueue-produce-failures	デッドレターキューへの書き込み失敗数。
deadletterqueue-produce-requests	デッドレターキューへの書き込み試行の数。
last-error-timestamp	このタスクで最後にエラーが発生したときのエポックタイムスタンプ。

属性	説明
total-errors-logged	ログに記録されたエラーの数。
total-record-errors	このタスクでのレコード処理エラーの数。
total-record-failures	このタスクでのレコード処理の失敗数。
total-records-skipped	エラーによりスキップされたレコードの数。
total-retries	再試行された操作の数。

18.9. KAFKA STREAMS MBEAN



注記

Streams アプリケーションには、ここで説明したものに加えて、[プロデューサー](#) と [コンシューマー](#) MBean が含まれます。

クライアントの Kafka Streams メトリクス

これらのメトリクスは、`metrics.recording.level` 設定パラメーターが **info** または **debug** の場合に収集されます。

表18.21 `kafka.streams:type=stream-metrics,client-id=*` と一致する MBeans

属性	説明
commit-latency-avg	このスレッドのすべての実行中のタスクにおけるコミットの平均実行時間 (ミリ秒単位)。
commit-latency-max	このスレッドのすべての実行中のタスクにおけるコミットの最大実行時間 (ミリ秒単位)。
commit-rate	1秒あたりの平均コミット数。
commit-total	コミットコールの合計数。
poll-latency-avg	コンシューマーポーリングの平均実行時間 (ミリ秒単位)。
poll-latency-max	コンシューマーポーリングの最大実行時間 (ミリ秒単位)。
poll-rate	1秒あたりのコンシューマーポーリングコールの平均数。
poll-total	コンシューマーポーリングコールの合計数。

属性	説明
process-latency-avg	処理の平均実行時間 (ミリ秒単位)。
process-latency-max	処理の最大実行時間 (ミリ秒単位)。
process-rate	1 秒あたりの処理レコードの平均数。
process-total	処理レコードの総数。
punctuate-latency-avg	中断の平均実行時間 (ミリ秒単位)。
punctuate-latency-max	中断の最大実行時間 (ミリ秒単位)。
punctuate-rate	1 秒あたりの中断コールの平均数。
punctuate-total	中断コールの合計数。
task-closed-rate	1 秒間に閉じられたタスクの平均数。
task-closed-total	閉じられたタスクの合計数。
task-created-rate	1 秒あたり作成されるタスクの平均数。
task-created-total	作成されたタスクの合計数。

タスクの Kafka Streams メトリクス

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** のときに収集されます。

表18.22 **kafka.streams:type=stream-task-metrics,client-id=*,task-id=*** に一致する MBeans

属性	説明
active-process-ratio	割り当て済みでアクティブなタスクすべての中で、ストリームスレッドがこのタスクの処理に費やした時間の割合。
commit-latency-avg	コミットの平均実行時間 (ナノ秒単位)。
commit-latency-max	コミットの最大実行時間 (ナノ秒単位)。
commit-rate	1 秒あたりのコミットコールの平均数。
commit-total	コミットコールの合計数。
dropped-records-rate	このタスク内でドロップされたレコードの平均数。

属性	説明
dropped-records-total	このタスク内でドロップされたレコードの合計数。
enforced-processing-rate	1秒あたりの平均強制処理数。
enforced-processing-total	強制的に実行された合計処理数。
process-latency-avg	処理の平均実行時間 (ナノ秒単位)。
process-latency-max	処理の最大実行時間 (ナノ秒単位)。
process-rate	このタスクのすべてのソースプロセッサースレッドにおける1秒あたりの処理レコードの平均数。
process-total	このタスクのすべてのソースプロセッサースレッドにおける処理済みレコードの合計数。
record-lateness-avg	レコードで発生した平均遅延 (ストリーム時間 - レコードのタイムスタンプ)。
record-lateness-max	レコードで発生した最大遅延 (ストリーム時間 - レコードのタイムスタンプ)。

プロセッサースレッドの Kafka Streams メトリクス

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** のときに収集されます。

表18.23 **kafka.streams:type=stream-processor-node-metrics,client-id=*,task-id=*,processor-node-id=*** と一致する MBeans

属性	説明
process-rate	1秒あたりのソースプロセッサースレッドで処理されるレコードの平均数。
process-total	1秒あたりのソースプロセッサースレッドで処理されるレコードの合計数。
record-e2e-latency-avg	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの平均エンドツーエンド遅延。
record-e2e-latency-max	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの最大エンドツーエンド遅延。

属性	説明
record-e2e-latency-min	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの最小エンドツーエンド遅延。
suppression-emit-rate	抑制操作ノードからダウンストリームに出力されるレコードの割合。
suppression-emit-total	抑制操作ノードからダウンストリームに出力されるレコードの合計数。

ステートストアの Kafka Streams メトリクス

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** のときに収集されます。

表18.24 kafka.streams:type=stream-[store-scope]-metrics,client-id=*,task-id=*,[store-scope]-id=*
と一致する MBeans

属性	説明
all-latency-avg	すべての操作の平均実行時間 (ns)。
all-latency-max	すべての操作の最大実行時間 (ns)。
all-rate	このストアのすべての操作レートの平均。
delete-latency-avg	平均削除実行時間 (ナノ秒単位)。
delete-latency-max	最大削除実行時間 (ナノ秒単位)。
delete-rate	このストアの平均削除レート。
flush-latency-avg	フラッシュの平均実行時間 (ナノ秒単位)。
flush-latency-max	フラッシュの最大実行時間 (ナノ秒単位)。
flush-rate	このストアの平均フラッシュレート。
get-latency-avg	get の平均実行時間 (ナノ秒単位)。
get-latency-max	get の最大実行時間 (ナノ秒単位)。
get-rate	このストアの平均 get レート。
put-all-latency-avg	put-all の平均実行時間 (ナノ秒単位)。

属性	説明
put-all-latency-max	put-all の最大実行時間 (ナノ秒単位)。
put-all-rate	このストアの平均 put-all レート。
put-if-absent-latency-avg	put-if-absent の平均実行時間 (ナノ秒単位)。
put-if-absent-latency-max	put-if-absent の最大実行時間 (ナノ秒単位)。
put-if-absent-rate	このストアの平均 put-if-absent レート。
put-latency-avg	put の平均実行時間 (ナノ秒単位)。
put-latency-max	put の最大実行時間 (ナノ秒単位)。
put-rate	このストアの平均 put レート。
range-latency-avg	平均範囲実行時間 (ナノ秒単位)。
range-latency-max	最大範囲実行時間 (ナノ秒単位)。
range-rate	このストアの平均範囲のレート。
record-e2e-latency-avg	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの平均エンドツーエンド遅延。
record-e2e-latency-max	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの最大エンドツーエンド遅延。
record-e2e-latency-min	レコードのタイムスタンプをノードによって完全に処理されたときのシステム時刻と比較することによって測定された、レコードの最小エンドツーエンド遅延。
restore-latency-avg	復元の平均実行時間 (ナノ秒単位)。
restore-latency-max	復元の最大実行時間 (ナノ秒単位)。
restore-rate	このストアの平均復元レート。
suppression-buffer-count-avg	サンプリングウィンドウ上でバッファされたレコードの平均数。

属性	説明
suppression-buffer-count-max	サンプリングウィンドウ上でバッファーされた最大レコード数。
suppression-buffer-size-avg	サンプリングウィンドウ上でのバッファーデータの合計サイズ (バイト単位)。
suppression-buffer-size-max	サンプリングウィンドウ上でのバッファーデータの最大サイズ (バイト単位)。

レコードキャッシュの Kafka Streams メトリクス

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **debug** のときに収集されます。

表18.25 **kafka.streams:type=stream-record-cache-metrics,client-id=*,task-id=*,record-cache-id=*** に一致する MBeans

属性	説明
hit-ratio-avg	キャッシュ読み取り要求の合計に対するキャッシュ読み取りヒット率として定義される平均キャッシュヒット率。
hit-ratio-max	最大キャッシュヒット率。
hit-ratio-min	最小キャッシュヒット率。

付録A サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **製品のダウンロード** ページにログインします。
2. **インテグレーションおよび自動化** カテゴリで、**AMQ Streams for Apache Kafka** エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **ダウンロード** 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: 2022-12-03 23:08:10 +1000