



Red Hat AMQ 2021.Q3

AMQ Streams on RHEL の使用

AMQ Streams 1.8 on Red Hat Enterprise Linux 向け

Red Hat AMQ 2021.Q3 AMQ Streams on RHEL の使用

AMQ Streams 1.8 on Red Hat Enterprise Linux 向け

Enter your first name here. Enter your surname here.

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

Enter your email address here.

法律上の通知

Copyright © 2021 | 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

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

目次

多様性を受け入れるオープンソースの強化	9
第1章 AMQ STREAMS の概要	10
1.1. KAFKA の機能	11
1.2. KAFKA のユースケース	11
1.3. サポートされる構成	12
1.4. 本書の表記慣例	12
第2章 スタートガイド	13
2.1. AMQ STREAMS の配布	13
2.2. AMQ STREAMS アーカイブのダウンロード	13
2.3. AMQ STREAMS のインストール	13
2.4. データストレージに関する留意事項	14
2.4.1. Apache Kafka および ZooKeeper ストレージのサポート	14
2.4.2. ファイルシステム	15
2.5. 単一ノードの AMQ STREAMS クラスターの実行	15
2.6. クラスターの使用	16
2.7. AMQ STREAMS サービスの停止	17
2.8. AMQ STREAMS の設定	18
第3章 ZOOKEEPER の設定	20
3.1. 基本設定	20
3.2. ZOOKEEPER クラスターの設定	20
3.3. マルチノードの ZOOKEEPER クラスターの実行	22
3.4. 認証	24
3.4.1. SASL を使用した認証	24
3.4.2. DIGEST-MD5 を使用したサーバー間の認証の有効化	26
3.4.3. 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	27
3.5. 承認	28
3.6. TLS	29
3.7. その他の設定オプション	29
3.8. ロギング	29
第4章 KAFKA の設定	30
4.1. ZOOKEEPER	30
4.2. リスナー	30
4.3. ログのコミット	32
4.4. ブローカー ID	32
4.5. マルチノードの KAFKA クラスターの実行	32
4.6. ZOOKEEPER の認証	34
4.6.1. JAAS 設定	34
4.6.2. ZooKeeper 認証の有効化	34
4.7. 承認	35
4.7.1. シンプルな ACL オーソライザー	35
4.7.1.1. ACL ルール	35
4.7.1.2. プリンシパル	36
4.7.1.3. ユーザーの認証	36
4.7.1.4. スーパーユーザー	36
4.7.1.5. レプリカブローカーの認証	36
4.7.1.6. サポートされるリソース	37
4.7.1.7. サポートされる操作	37
4.7.1.8. ACL 管理オプション	37

4.7.2. 承認の有効化	42
4.7.3. ACL ルールの追加	42
4.7.4. ACL ルールの一覧表示	43
4.7.5. ACL ルールの削除	44
4.8. ZOOKEEPER の承認	45
4.8.1. ACL 設定	45
4.8.2. 新しい Kafka クラスターでの ZooKeeper ACL の有効化	46
4.8.3. 既存の Kafka クラスターでの ZooKeeper ACL の有効化	46
4.9. 暗号化と認証	47
4.9.1. リスナーの設定	47
4.9.2. TLS 暗号化	48
4.9.3. TLS 暗号化の有効化	49
4.9.4. 認証	50
4.9.4.1. TLS クライアント認証	50
4.9.4.2. SASL 認証	50
4.9.5. TLS クライアント認証の有効化	53
4.9.6. SASL PLAIN 認証の有効化	54
4.9.7. SASL SCRAM 認証の有効化	55
4.9.8. SASL SCRAM ユーザーの追加	56
4.9.9. SASL SCRAM ユーザーの削除	57
4.10. OAUTH 2.0 トークンベース認証の使用	57
4.10.1. OAuth 2.0 認証メカニズム	59
4.10.1.1. プロパティまたは変数を使用した OAuth 2.0 の設定	60
4.10.2. OAuth 2.0 Kafka ブローカーの設定	60
4.10.2.1. 承認サーバーの OAuth 2.0 クライアント設定	61
4.10.2.2. Kafka クラスターでの OAuth 2.0 認証設定	61
4.10.2.3. 高速なローカル JWT トークン検証の設定	66
4.10.2.4. OAuth 2.0 インtrospeクシオンエンドポイントの設定	67
4.10.3. Kafka ブローカーの再認証の設定	68
4.10.4. OAuth 2.0 Kafka クライアントの設定	69
4.10.5. OAuth 2.0 のクライアント認証フロー	70
4.10.5.1. クライアント認証フローの例	70
4.10.6. OAuth 2.0 認証の設定	73
4.10.6.1. OAuth 2.0 承認サーバーとしての Red Hat Single Sign-On の設定	73
4.10.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定	74
4.10.6.3. OAuth 2.0 を使用するように Kafka Java クライアントを設定	78
4.11. OAUTH 2.0 トークンベース承認の使用	80
4.11.1. OAuth 2.0 の承認メカニズム	80
4.11.1.1. Kafka ブローカーのカスタムオーソライザー	80
4.11.2. OAuth 2.0 承認サポートの設定	81
4.12. OPA ポリシーベースの承認の使用	83
4.12.1. OPA ポリシーの定義	83
4.12.2. OPA への接続	84
4.12.3. OPA 承認サポートの設定	84
4.13. ログイン	85
4.13.1. Kafka ブローカーロガーのログインレベルの動的な変更	86
ブローカーロガーのリセット	87
第5章 トピック	88
5.1. パーティションおよびレプリカ	88
5.2. メッセージの保持	88
5.3. トピックの自動作成	89
5.4. トピックの削除	89

5.5. トピックの設定	89
5.6. 内部トピック	90
5.7. トピックの作成	91
5.8. トピックの一覧表示および説明	92
5.9. トピック設定の変更	92
5.10. トピックの削除	94
第6章 KAFKA の管理	96
6.1. KAFKA 設定のチューニング	96
6.1.1. Kafka ブローカー設定のチューニング	96
6.1.1.1. 基本的なブローカー設定	96
6.1.1.2. 高可用性のためのトピックの複製	97
6.1.1.3. トランザクションおよびコミットの内部トピック設定	98
6.1.1.4. I/O スレッドの増加によるリクエスト処理スループットの向上	98
6.1.1.5. レイテンシーの高い接続に対する帯域幅の引き上げ	99
6.1.1.6. データ保持ポリシーでのログの管理	99
6.1.1.7. クリーンアップポリシーによるログデータの削除	101
6.1.1.8. ディスク使用率の管理	103
6.1.1.9. 大きなメッセージサイズの処理	104
6.1.1.10. メッセージデータのログフラッシュの制御	106
6.1.1.11. 可用性のためのパーティションリバランス	106
6.1.1.12. クリーンでないリーダーの選出 (unclean leader election)	107
6.1.1.13. 不要なコンシューマーグループリバランスの回避	108
6.1.2. Kafka プロデューサー設定のチューニング	108
6.1.2.1. 基本のプロデューサー設定	108
6.1.2.2. データの持続性	109
6.1.2.3. 順序付き配信	110
6.1.2.4. 信頼性の保証	111
6.1.2.5. スループットおよびレイテンシーの最適化	111
6.1.3. Kafka コンシューマー設定の調整	113
6.1.3.1. 基本的なコンシューマー設定	113
6.1.3.2. コンシューマーグループを使用したデータ消費のスケーリング	114
6.1.3.3. メッセージの順序の保証	114
6.1.3.4. スループットおよびレイテンシーの最適化	115
6.1.3.5. オフセットをコミットする際のデータ損失または重複の回避	116
6.1.3.5.1. トランザクションメッセージの制御	117
6.1.3.6. データ損失を回避するための障害からの復旧	117
6.1.3.7. オフセットポリシーの管理	117
6.1.3.8. リバランスの影響を最小限にする	118
6.2. 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	119
6.3. クラスターのスケーリング	121
6.3.1. Kafka クラスターのスケーリング	121
6.3.1.1. ブローカーのクラスターへの追加	121
6.3.1.2. クラスターからのブローカーの削除	121
6.3.2. パーティションの再割り当て	121
6.3.2.1. 再割り当て JSON ファイル	122
6.3.2.2. 再割り当て JSON ファイルの生成	122
6.3.2.3. 手動による再割り当て JSON ファイルの作成	123
6.3.3. 再割り当てスロットル	123
6.3.4. Kafka クラスターのスケールアップ	124
6.3.5. Kafka クラスターのスケールダウン	125
6.3.6. ZooKeeper クラスターのスケールアップ	126
6.3.7. ZooKeeper クラスターのスケールダウン	127

第7章 JMX を使用したクラスターのモニタリング	129
7.1. JMX 設定オプション	129
7.2. JMX エージェントの無効化	129
7.3. 別のマシンからの JVM への接続	129
7.4. JCONSOLE を使用したモニタリング	130
7.5. 重要な KAFKA ブローカーメトリクス	130
7.5.1. Kafka サーバーメトリクス	131
7.5.2. Kafka ネットワークメトリクス	133
7.5.3. Kafka ログメトリクス	134
7.5.4. Kafka コントローラーメトリクス	135
7.5.5. Yammer メトリクス	135
7.6. プロデューサー MBEAN	136
7.6.1. 一致する MBean kafka.producer:type=producer-metrics,client-id=*	136
7.6.2. 一致する MBean kafka.producer:type=producer-metrics,client-id=*,node-id=*	138
7.6.3. 一致する MBean kafka.producer:type=producer-topic-metrics,client-id=*,topic=*	139
7.7. コンシューマー MBEAN	139
7.7.1. 一致する MBean kafka.consumer:type=consumer-metrics,client-id=*	139
7.7.2. 一致する MBean kafka.consumer:type=consumer-metrics,client-id=*,node-id=*	140
7.7.3. 一致する MBean kafka.consumer:type=consumer-coordinator-metrics,client-id=*	141
7.7.4. 一致する MBean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*	142
7.7.5. 一致する MBean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*	143
7.7.6. 一致する MBean kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*,partition=*	143
7.8. KAFKA CONNECT MBEANS	144
7.8.1. 一致する MBean kafka.connect:type=connect-metrics,client-id=*	144
7.8.2. 一致する MBean kafka.connect:type=connect-metrics,client-id=*,node-id=*	145
7.8.3. 一致する MBean kafka.connect:type=connect-worker-metrics	145
7.8.4. 一致する MBean kafka.connect:type=connect-worker-rebalance-metrics	146
7.8.5. 一致する MBean kafka.connect:type=connector-metrics,connector=*	147
7.8.6. 一致する MBean kafka.connect:type=connector-task-metrics,connector=*,task=*	147
7.8.7. 一致する MBean kafka.connect:type=sink-task-metrics,connector=*,task=*	148
7.8.8. 一致する MBean kafka.connect:type=source-task-metrics,connector=*,task=*	149
7.8.9. 一致する MBean kafka.connect:type=task-error-metrics,connector=*,task=*	150
7.9. KAFKA STREAMS MBEANS	150
7.9.1. 一致する MBean kafka.streams:type=stream-metrics,client-id=*	151
7.9.2. 一致する MBean kafka.streams:type=stream-task-metrics,client-id=*,task-id=*	152
7.9.3. 一致する MBean kafka.streams:type=stream-processor-node-metrics,client-id=*,task-id=*,processor-node-id=*	152
7.9.4. 一致する MBean kafka.streams:type=stream-[store-scope]-metrics,client-id=*,task-id=*,[store-scope]-id=*	153
7.9.5. 一致する MBean kafka.streams:type=stream-record-cache-metrics,client-id=*,task-id=*,record-cache-id=*	155
第8章 KAFKA CONNECT	156
8.1. スタンドアロンモードでの KAFKA CONNECT	156
8.1.1. スタンドアロンモードでの Kafka Connect の設定	156
8.1.2. スタンドアロンモードでの Kafka Connect でのコネクタの設定	157
8.1.3. スタンドアロンモードでの Kafka Connect の実行	157
8.2. 「KAFKA CONNECT IN DISTRIBUTED MODE」	158
8.2.1. 分散モードでの Kafka Connect の設定	158
8.2.2. 分散 Kafka Connect でのコネクタの設定	159
8.2.3. 分散 Kafka Connect の実行	160
8.2.4. コネクタの作成	161

8.2.5. コネクタの削除	162
8.3. コネクタプラグイン	162
8.4. 「ADDING CONNECTOR PLUGINS」	163
第9章 AMQ STREAMS の MIRRORMAKER 2.0 との使用	165
9.1. MIRRORMAKER 2.0 データレプリケーション	165
9.2. クラスターの設定	166
9.2.1. 双方向レプリケーション (active/active)	166
9.2.2. 一方方向レプリケーション (active/passive)	167
9.2.3. トピック設定の同期	167
9.2.4. データの整合性	167
9.2.5. オフセットの追跡	167
9.2.6. コンシューマーグループオフセットの同期	168
9.2.7. 接続性チェック	168
9.3. ACL ルールの同期	169
9.4. MIRRORMAKER 2.0 を使用した KAFKA クラスター間でのデータの同期	169
9.5. レガシーモードでの MIRRORMAKER 2.0 の使用	172
第10章 KAFKA クライアント	174
10.1. KAFKA クライアントを依存関係として MAVEN プロジェクトに追加	174
第11章 KAFKA STREAMS API の概要	176
11.1. KAFKA STREAMS API を依存関係として MAVEN プロジェクトに追加	176
第12章 KAFKA BRIDGE	178
12.1. KAFKA BRIDGE の概要	178
HTTP リクエスト	178
12.1.1. 認証および暗号化	179
12.1.2. Kafka Bridge へのリクエスト	179
12.1.2.1. コンテンツタイプヘッダー	179
12.1.2.2. 埋め込みデータ形式	180
12.1.2.3. メッセージの形式	180
12.1.2.4. Accept ヘッダー	181
12.1.3. Kafka Bridge のログガーの設定	181
12.1.4. Kafka Bridge API リソース	182
12.1.5. Kafka Bridge アーカイブのダウンロード	183
12.1.6. Kafka Bridge プロパティの設定	183
12.1.7. Kafka Bridge のインストール	184
12.2. KAFKA BRIDGE クイックスタート	185
12.2.1. Kafka Bridge のローカルでのデプロイ	185
12.2.2. トピックおよびパーティションへのメッセージの作成	186
12.2.3. Kafka Bridge コンシューマーの作成	187
12.2.4. Kafka Bridge コンシューマーのトピックへのサブスクライブ	188
12.2.5. Kafka Bridge コンシューマーからの最新メッセージの取得	189
12.2.6. ログへのオフセットのコミット	190
12.2.7. パーティションのオフセットのシーク	191
12.2.8. Kafka Bridge コンシューマーの削除	192
第13章 KERBEROS(GSSAPI)認証の使用	193
13.1. KERBEROS(GSSAPI)認証を使用するように AMQ STREAMS を設定	193
認証のサービスプリンシパルの追加	193
Kerberos ログインを使用するように ZooKeeper を設定する	194
Kerberos ログインを使用するように Kafka ブローカーサーバーを設定します。	196
Kafka プロデューサーおよびコンシューマクライアントが Kerberos 認証を使用するように設定する	198

第14章 CRUISE CONTROL によるクラスターのリバランス	200
14.1. CRUISE CONTROL とは	200
14.2. CRUISE CONTROL アーカイブのダウンロード	201
14.3. CRUISE CONTROL METRICS REPORTER のデプロイ	201
14.4. CRUISE CONTROL の設定および起動	203
自動作成されたトピック	204
14.5. 最適化ゴールの概要	205
Cruise Control プロパティファイルのゴールの設定	206
マスター最適化ゴール	206
ハードゴールおよびソフトゴール	206
デフォルトの最適化ゴール	207
ユーザー提供の最適化ゴール	207
14.6. 最適化プロポーザルの概要	208
キャッシュされた最適化プロポーザル	208
最適化プロポーザルの内容	208
14.7. リバランスパフォーマンスチューニングの概要	210
パーティション再割り当てコマンド	210
レプリカの移動ストラテジー	210
リバランスチューニングオプション	211
14.8. CRUISE CONTROL の設定	212
容量の設定	213
Cruise Control Metrics トピックのログクリーンアップポリシー	214
ロギングの設定	214
14.9. 最適化プロポーザルの生成	215
非同期応答	217
14.10. クラスターリバランスの開始	218
14.11. アクティブなクラスターリバランスの停止	218
第15章 分散トレーシング	220
AMQ Streams によるトレーシングのサポート方法	220
手順の概要	221
15.1. OPENTRACING および JAEGER の概要	221
15.2. KAFKA クライアントのトレーシング設定	222
15.2.1. Kafka クライアント用の Jaeger トレーサーの初期化	222
15.2.2. トレーシングのための Kafka プロデューサーおよびコンシューマーのインストルメント化	223
Decorator パターンのカスタムスパン名	224
ビルトインスパン名	225
15.2.3. Kafka Streams アプリケーションのトレーシングのインストルメント化	226
15.3. MIRRORMAKER および KAFKA CONNECT のトレース設定	227
15.3.1. MirrorMaker のトレースの有効化	227
15.3.2. MirrorMaker 2.0 のトレースの有効化	227
15.3.3. Kafka Connect のトレースの有効化	228
15.4. KAFKA BRIDGE のトレースの有効化	229
15.5. トレーシングの環境変数	229
第16章 KAFKA EXPORTER	232
16.1. コンシューマーラグ	232
16.2. KAFKA EXPORTER アラートルールの例	233
16.3. KAFKA EXPORTER メトリクス	233
16.4. KAFKA EXPORTER の実行	234
16.5. GRAFANA での KAFKA EXPORTER メトリクスの表示	236
第17章 AMQ STREAMS および KAFKA のアップグレード	237
17.1. アップグレードの前提条件	237

17.2. アップグレードプロセス	237
17.3. KAFKA バージョン	237
17.4. AMQ STREAMS 1.8 へのアップグレード	238
17.4.1. Kafka ブローカーおよび ZooKeeper のアップグレード	238
17.4.2. Kafka Connect のアップグレード	240
17.5. KAFKA のアップグレード	241
17.5.1. 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード	242
17.5.2. クライアントをアップグレードする戦略	243
17.5.3. クライアントアプリケーションの新しい Kafka バージョンへのアップグレード	244
17.5.4. 新しいメッセージ形式のバージョンを使用するように Kafka ブローカーのアップグレード	245
17.5.5. コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード	246
 付録A ブローカー設定パラメーター	 248
付録B トピック設定パラメーター	286
付録C コンシューマー設定パラメーター	292
付録D プロデューサー設定パラメーター	305
付録E 管理クライアント設定パラメーター	318
付録F KAFKA CONNECT 設定パラメーター	327
付録G KAFKA STREAMS 設定パラメーター	343
付録H サブスクリプションの使用	351
アカウントへのアクセス	351
サブスクリプションのアクティベート	351
Zip および Tar ファイルのダウンロード	351
パッケージ用のシステムの登録	351

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。これは大規模な取り組みであるため、これらの変更は今後の複数のリリースで段階的に実施されます。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#)をご覧ください。

第1章 AMQ STREAMS の概要

Red Hat AMQ Streams は、Apache ZooKeeper および Apache Kafka プロジェクトに基づく、拡張性の高い、分散型の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構成されます。

Kafka Broker

生成側クライアントから消費側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、非常に信頼性の高い分散型の連携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記述するための 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 インターフェースが提供されます。

Kafka Connect

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

Kafka MirrorMaker

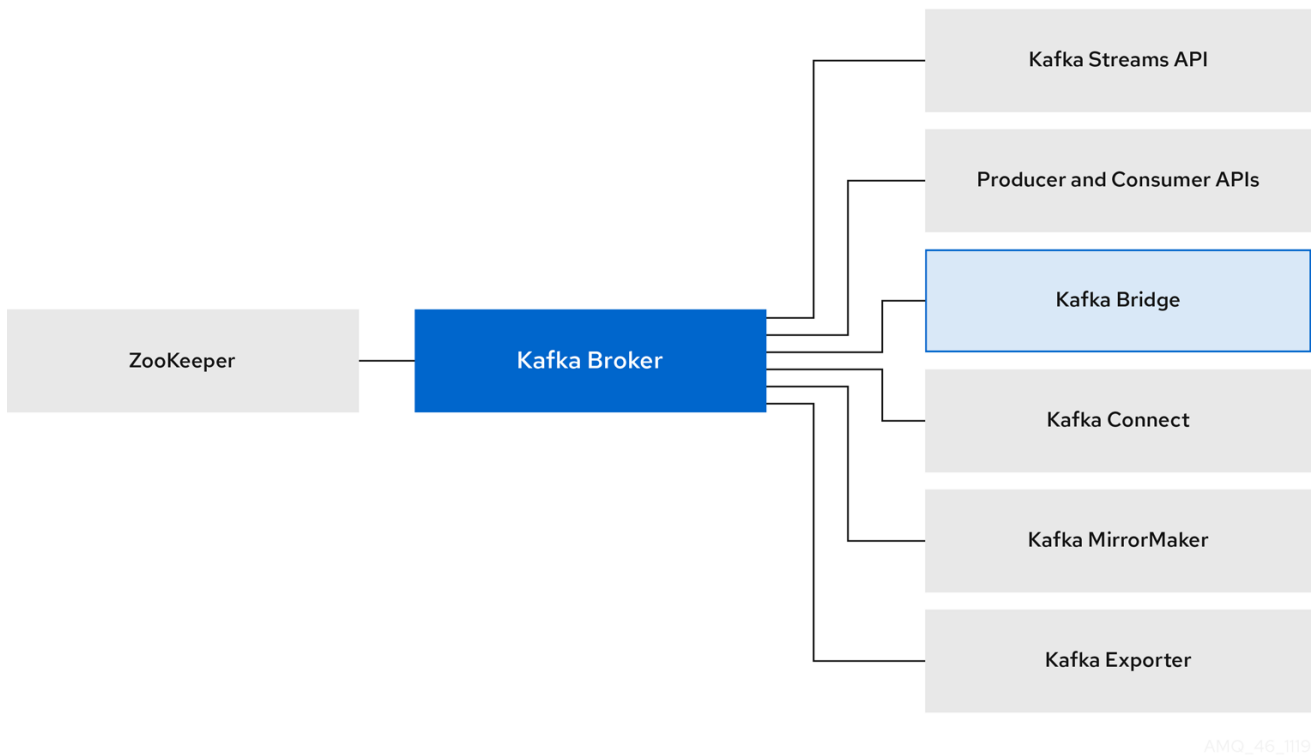
データセンター内または異なるデータセンターにある 2 つの Kafka クラスター間でデータをレプリケートします。

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、これらの全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連携に 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



AMQ_46_1119

1.1. KAFKA の機能

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

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

1.2. KAFKA のユースケース

Kafka の機能は、以下に適しています。

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

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

1.3. サポートされる構成

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

表1.1 サポートされる Java 仮想マシンの一覧

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

表1.2 サポートされるオペレーティングシステムの一覧

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

1.4. 本書の表記慣例

置き換え可能なテキスト

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

たとえば、以下のコードでは、**BOOTSTRAP-ADDRESS** および **TOPIC-NAME** を実際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 の配布

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 のコンポーネントが含まれています。

- 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

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

2.3. AMQ STREAMS のインストール

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

既存のクラスターを AMQ Streams 1.8 にアップグレードする手順は、[「AMQ Streams および Kafka のアップグレード」](#)を参照してください。

前提条件

- [インストールアーカイブ](#) のダウンロード。
- [「サポートされる構成」](#) の確認。

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. ディレクトリー **/opt/kafka** を作成します。

```
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 データを格納するディレクトリ **/var/lib/zookeeper** 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 データを格納するディレクトリ **/var/lib/kafka** 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。

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

可能な場合はローカルストレージを選択します。ローカルストレージが利用できない場合には、Fibre Channel または iSCSI などのプロトコルでアクセスする Storage Area Network (SAN) を使用できません。

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

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

Kafka は、複数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。

2.4.2. ファイルシステム

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

その他のリソース

- XFS の詳細は、[「The XFS File System」](#) を参照してください。

2.5. 単一ノードの AMQ STREAMS クラスターの実行

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



警告

単一ノードの AMQ Streams クラスターは、信頼性と高可用性を提供せず、開発の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 が実行していることを確認します。

```
jcmd | grep zookeeper
```

戻り値:

```
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
```

戻り値:

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

その他のリソース

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

2.6. クラスターの使用

この手順では、Kafka コンソールプロデューサーおよびコンシューマ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ができます。詳細は[「トピック」](#)を参照してください。

前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [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場合があります。

```
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て、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、続いて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
```

2.8. AMQ STREAMS の設定

前提条件

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

手順

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

ZooKeeper

`/opt/kafka/config/zookeeper.properties`

Kafka

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

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

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

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

```
# This is a comment
```

改行/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";
```

3. 変更を保存します。
4. ZooKeeper または Kafka ブローカーを再起動します。

5. クラスターのすべてのノードでこの手順を繰り返します。

第3章 ZOOKEEPER の設定

Kafka は ZooKeeper を使用して設定データを保存し、クラスター内の連携を行います。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。

3.1. 基本設定

最も重要な ZooKeeper 設定オプションは以下のとおりです。

tickTime

ZooKeeper の基本時間単位 (ミリ秒単位)。これはハートビートとセッション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` に配置する必要があります。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
```

3.2. ZOOKEEPER クラスターの設定

ほとんどの実稼働環境では、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、**アンサンブル** とも呼ばれます。

ZooKeeper クラスターは通常、奇数のノードで構成されます。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つのノードで構成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つのノードで構成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つのノードで構成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および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

の回復性を向上させません。4つのノードで構成されるクラスターでは、少なくとも3つのノードが利用可能である必要があり、停止しているノードは1つしか許容できません。そのため、3つのノードでしか構成されないクラスターとまったく同じ回復性しか持ちません。

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つのノードで構成される ZooKeeper クラスターで十分です。



警告

3つのノードで構成される ZooKeeper クラスターでは、利用できないノードは1つしか許容できません。つまり、クラスターノードがクラッシュした時に別のノードでメンテナンスを実施中だった場合、ZooKeeper クラスターが利用できなくなることを意味します。

レプリケートされた ZooKeeper の構成は、スタンドアロン設定でサポートされるすべての設定オプションをサポートします。クラスタリング設定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 [timeTick 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いられる時間。時間はティック数として指定されます (詳細は [timeTick 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つだけのサーバーで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

イントラクラスター通信に使用されるポート番号。

port2

リーダーの選択に使用するポート番号。

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

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

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



注記

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

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/` などの **dataDir** フォルダーに保存する必要があります。**myid** ファイルには、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。

3.3. マルチノードの ZOOKEEPER クラスターの実行

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



注記

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

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

- **dataDir** オプションを `/var/lib/zookeeper/` に設定します。
- **initLimit** および **syncLimit** オプションを設定します。
- **reconfigEnabled** および **standaloneEnabled** オプションを設定します。
- すべての ZooKeeper ノードの一覧を追加します。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

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

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

3. デフォルトの設定ファイルで ZooKeeper を起動します。

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

4. ZooKeeper が稼働していることを確認します。

```
jcmd | grep zookeeper
```

5. クラスターのすべてのノードでこの手順を繰り返します。
6. クラスターのすべてのノードが稼働したら、**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
```

この出力に、ノードが **leader** または **follower** のいずれかである情報が表示されるはずです。

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

```

その他のリソース

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

3.4. 認証

デフォルトでは、ZooKeeper は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で DIGEST-MD5 SASL メカニズムを使用する認証をサポートします。

3.4.1. 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。マルチノードの 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、個別の要素は常に個別のコンテキストで設定されます。コンテキストは **設定 オプション** で、以下の形式となっています。

```

ContextName {
    param1
    param2;
};

```

SASL 認証は、サーバー間通信 (ZooKeeper インスタンス間の通信) とクライアント間の通信 (Kafka と ZooKeeper 間の通信) 用に、個別に設定されます。サーバー間の認証は、マルチノードの 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 つの部分が含まれています。

- 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のパスワードと共に、接続が許可されるすべてのユーザー名

を含める必要があります。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。これにも、暗号化されていない形式のパスワードが含まれます。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については、[Kafka のインストール](#) 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 メカニズム用に、すべてのユーザー名とパスワードが設定されます。

```
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。

3.4.2. DIGEST-MD5 を使用したサーバー間の認証の有効化

この手順では、ZooKeeper クラスターのノード間で SASL DIGEST-MD5 メカニズムを使用し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 クラスターがマルチノードで [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";
};
```

- 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
```

- 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
```

その他のリソース

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

3.4.3. DIGEST-MD5 を使用したクライアント/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 [設定済みで実行されている](#)。

SASL DIGEST-MD5 認証の有効化

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

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


super には、管理者権限が自動的に付与されます。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人だけです。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";
};
```

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

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

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

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

authProvider.<ID> プロパティは、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 ノードで構成される ZooKeeper クラスターの設定例は以下のようになります。

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

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

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

その他のリソース

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

3.5. 承認

ZooKeeper は、そこに格納されているデータを保護するためにアクセス制御リスト (ACL) をサポートします。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 ブローカーでの ZooKeeper ACL の有効化に関する詳細は、[「ZooKeeper の承認」](#)を参照してください。

3.6. TLS

ZooKeeper は、暗号化または認証用に TLS をサポートします。

3.7. 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。

3.8. 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。

第4章 KAFKA の設定

Kafka はプロパティ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記述する設定ファイルの例が含まれています。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を説明します。サポートされる Kafka ブローカー設定オプションの完全リストは、「[付録A ブローカー設定パラメーター](#)」を参照してください。

4.1. ZOOKEEPER

クラスターの連携 (たとえば、どのノードをどのパーティションのリーダーにするかを決定する場合など) に加えて、Kafka ブローカーには、設定の一部を格納するために ZooKeeper が必要になります。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 クラスターに対してのみ使用できます。複数の Kafka クラスターが単一の ZooKeeper クラスターを使用するように設定するには、Kafka 設定ファイルの ZooKeeper 接続文字列の最後にベース (プレフィックス) パスを指定します。

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

4.2. リスナー

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。各リスナーには異なる設定が必要で、これにより別のポートまたはネットワークインターフェースでリッスンすることができます。

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** プロパティにリスナーをコンマ区切りのリストとして追加します。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。

複数リスナーの設定例

```
listeners=internal-1://:9092,internal-2://:9093,replication://:9094
```

Kafka クライアントが Kafka クラスターに接続する場合、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、または IP アドレスではなく外部 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** プロパティと同じように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。

ブローカー間のリスナー

ブローカー間のリスナーは、Kafka ブローカー間の通信に使用されます。ブローカー間の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格納されているパーティション間でのメッセージのレプリケーション
- 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

ブローカー間のリスナーは、希望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** プロパティでブローカー間リスナーの名前を定義できます。

ここでは、ブローカー間リスナーの名前は **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には、**ブローカー間のリスナー** が使用されます。コントローラーは、パーティションリーダーの変更などの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、希望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の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で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、ブローカー間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 [ブローカー間のリスナー](#) が使用されます。

詳細は、「[付録A ブローカー設定パラメーター](#)」を参照してください。

4.3. ログのコミット

Apache Kafka は、プロデューサーから受信するすべてのレコードをコミットログに保存します。コミットログには、Kafka が配信する必要のある実際のデータがレコードとして含まれます。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、それぞれを異なる物理デバイスに配置して、ディスク I/O のパフォーマンスを向上させることができます。以下に例を示します。

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

4.4. 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特定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ことが重要になります。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
```

4.5. マルチノードの KAFKA クラスターの実行

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

前提条件

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

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 **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
 - 最初のブローカーの **broker.id** フィールドを **0** 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
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
```

ブローカーの検証

クラスターのすべてのノードが稼働したら、**ncat** ユーティリティーを使用して **dump** コマンドを ZooKeeper ノードの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。このコマンドは、ZooKeeper に登録されているすべての Kafka ブローカーを出力します。

1. **ncat stat** を使用してノードのステータスを確認します。

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

出力には、ここで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はずです。

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

```
SessionTracker dump:
org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
ephemeral nodes dump:
Sessions with Ephemerals (3):
0x200000015dd00000:
    /brokers/ids/1
0x100000015dc70000:
    /controller
```

```

/brokers/ids/0
0x10000015dc70001:
/brokers/ids/2

```

その他のリソース

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

4.6. ZOOKEEPER の認証

デフォルトでは、ZooKeeper と Kafka 間の接続は認証されません。ただし、Kafka および ZooKeeper は、Simple Authentication and Security Layer (SASL) 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で DIGEST-MD5 SASL メカニズムを使用する認証をサポートします。

4.6.1. JAAS 設定

ZooKeeper 接続の SASL 認証は、JAAS 設定ファイルで設定する必要があります。デフォルトでは、Kafka は ZooKeeper 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/jaas.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";
};

```

4.6.2. ZooKeeper 認証の有効化

この手順では、ZooKeeper に接続する際に SASL DIGEST-MD5 メカニズムを使用し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/jaas.conf` JAAS 設定ファイルを作成または編集し、以下のコンテキストを追加します。

```

Client {
  org.apache.kafka.common.security.plain.PlainLoginModule required

```

```
username="<Username>"
password="<Password>";
};
```

ユーザー名とパスワードは ZooKeeper で設定され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
```

その他のリソース

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

4.7. 承認

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

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。

4.7.1. シンプルな ACL オーソライザー

AclAuthorizer を含むオーソライザープラグインは、**authorizer.class.name** プロパティにより有効になります。

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

選択されたオーソライザーの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。

4.7.1.1. ACL ルール

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

ACL ルールは以下の形式で定義されます。

プリンシパル **P** は、ホスト **H** から Kafka リソース **R** に対する操作 **O** が許可/禁止される。

たとえば、以下のようにルールを設定します。

ユーザー John は、ホスト 127.0.0.1 からトピック コメント を表示することができます。

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 に書き込むことができます。

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。

4.7.1.2. 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す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ときに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。

4.7.1.3. 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、「[暗号化と認証](#)」を参照してください。

4.7.1.4. スーパーユーザー

スーパーユーザーは、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
```

4.7.1.5. レプリカブローカーの認証

承認を有効にすると、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ブローカー間の接続が含まれます。そのため、承認が有効になっている場合は、ブローカー間の接続に認証を使用し、ブローカーが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。

4.7.1.6. サポートされるリソース

Kafka ACL は、以下のようなタイプのリソースに適用できます。

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

4.7.1.7. サポートされる操作

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4.1 リソースに対してサポートされる操作

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

4.7.1.8. ACL 管理オプション

ACL ルールは、Kafka ディストリビューションパッケージの一部として提供される **bin/kafka-acls.sh** ユーティリティ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。	
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.SimpleAclAuthorizer
authorizer-properties	設定	初期化のためにオーソライザーに渡されるキー/値のペア。 AclAuthorizer では、サンプル値は zookeeper.connect=zoo1.my-domain.com:2181 です。	
bootstrap-server	リソース	Kafka クラスターに接続するためのホスト/ポートのペア。	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。	

オプション	型	説明	デフォルト
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>	
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>委譲トークンを 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>	

オプション	型	説明	デフォルト
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
allow-principal	プリンシパル	<p>許可 ACL ルールに追加されるプリンシパル。</p> <p>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。</p>	
deny-principal	プリンシパル	<p>拒否 ACL ルールに追加されるプリンシパル。</p> <p>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。</p>	
principal	プリンシパル	<p>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。</p> <p>1つのコマンドに複数の --principal オプションを指定できます。</p>	
allow-host	ホスト	<p>--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。</p> <p>ホスト名または CIDR 範囲はサポートされていません。</p>	--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。

オプション	型	説明	デフォルト
deny-host	ホスト	<p>--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。</p> <p>ホスト名または CIDR 範囲はサポートされていません。</p>	--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	<p>操作を許可または拒否します。</p> <p>1つのコマンドに複数の --operation オプションを指定できます。</p>	All
producer	ショートカット	メッセージプロデューサーが必要とするすべての操作を許可または拒否するショートカット (トピックでの WRITE および DESCRIBE、クラスターでの CREATE)。	
consumer	ショートカット	メッセージコンシューマーが必要とするすべての操作を許可または拒否するショートカット (トピックでの READ および DESCRIBE、コンシューマーグループでの 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>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ようにプロデューサーが承認されている場合、冪等性は自動的に有効になります。</p>	

オプション	型	説明	デフォルト
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。	

4.7.2. 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

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

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

2. Kafka ブローカーを (再) 起動します。

その他のリソース

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

4.7.3. ACL ルールの追加

AclAuthorizer 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

- 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](#)。
- 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。

以下に例を示します。

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

```
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
```

- IP アドレスホスト **127.0.0.1** からの **myTopic** への **user1** の読み取りアクセスを拒否しま

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

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

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

その他のリソース

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

4.7.4. ACL ルールの一覧表示

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインを使用する場合に、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

- 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](#)。
- 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
```

その他のリソース

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

4.7.5. ACL ルールの削除

この手順では、Kafka ブローカーで **AclAuthorizer** プラグインを使用する場合に、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

- 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。
- 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。
以下に例を示します。
- **MyConsumerGroup** コンシューマーグループを使用して **myTopic** からの **user1** および **user2** の読み取りアクセスを許可する ACL を削除します。

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

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



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

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

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

- IP アドレスホスト **127.0.0.1** からの **myTopic** への **user1** の読み取り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
```

その他のリソース

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

4.8. ZOOKEEPER の承認

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

4.8.1. ACL 設定

ZooKeeper ACL ルールの適用は、**config/server.properties** Kafka 設定ファイルの **zookeeper.set.acl** プロパティ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付与されません。

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に、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る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仮定されます。

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 が含まれる場合など)、データ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。

4.8.2. 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

- AMQ Streams が、Kafka ブローカーとして使用されるすべてのホストに [インストールされている](#)。
- ZooKeeper クラスターが [設定済みで実行されている](#)。
- 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. `/opt/kafka/config/server.properties` Kafka 設定ファイルを編集して、すべてのクラスターノードの `zookeeper.set.acl` フィールドを `true` に設定します。

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

2. Kafka ブローカーを起動します。

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

この手順では、実行中の Kafka クラスターの Kafka 設定で ZooKeeper ACL を有効にする方法を説明します。`zookeeper-security-migration.sh` ツールを使用して、すべての既存の `znodes` に ZooKeeper 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 クラスターが [設定済みで実行されている](#)。

ZooKeeper ACL の有効化

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

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

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

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

以下に例を示します。

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

4.9. 暗号化と認証

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

4.9.1. リスナーの設定

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

Kafka ブローカーの各リスナーは、独自のセキュリティープロトコルで設定されます。設定プロパティー **listener.security.protocol.map** は、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、オプションとして TLS クライアント証明書による認証を使用するリスナー。

SASL_PLAINTEXT

暗号化は使用しないが、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** インターフェースは 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。以下も、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ースに TLS による暗号化および 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。

4.9.2. TLS 暗号化

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

TLS 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。このファイルへのパスは **ssl.keystore.location** プロパティで設定されます。**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、追加のパスワードを使用することがあります。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 では、認証局が署名した鍵に加えて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、常に推奨の方法です。クライアントが接続している Kafka ブローカーのアイデンティティーを確認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を常に含める必要があります。

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 設定オプションに加え、Kafka は TLS 設定を調整するための多くのオプションをサポートします。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場合を以下に示します。

ssl.cipher.suites

有効な暗号スイートの一覧それぞれの暗号スイートは、TLS 接続に使用される認証、暗号化、MAC、およびキー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。

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

4.9.3. TLS 暗号化の有効化

この手順では、Kafka ブローカーで暗号化を有効にする方法を説明します。

前提条件

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

手順

1. クラスター内のすべての Kafka ブローカーの TLS 証明書を作成します。証明書には、Common Name または Subject Alternative Name の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。

その他のリソース

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

- [付録C コンシューマー設定パラメーター](#)

4.9.4. 認証

認証には、以下を使用できます。

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

4.9.4.1. TLS クライアント認証

TLS クライアント認証は、すでに 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無効です (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て認証するよう求められますが、それに従わない選択が可能です。

required

クライアントは TLS クライアント証明書を使用して認証することが要求されます。

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、識別名が **CN=someuser** 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。

4.9.4.2. SASL 認証

SASL 認証は、Java Authentication and Authorization Service (JAAS) を使用して設定されます。JAAS は、Kafka と ZooKeeper との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できる必要があります。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 は各リスナーに対して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** にする必要があります。

Kafka の SASL 認証は、さまざまなメカニズムをサポートします。

PLAIN

ユーザー名とパスワードを基に認証を実装します。ユーザー名およびパスワードは、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 クレデンシャルは、ZooKeeper に一元的に保存されます。SCRAM は、ZooKeeper クラスターノードがプライベートネットワークで分離された状態で実行され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。したがって、TLS による暗号化との組み合わせでのみ使用する必要があります。

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。これには、**sasl.enabled.mechanisms** プロパティーを使用します。このプロパティーには、有効なメカニズムのコンマ区切りリストが含まれます。

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

ブローカー間の通信に使用されるリスナーが SASL を使用している場合、**sasl.mechanism.inter.broker.protocol** プロパティー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
```

ブローカー間の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 人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た状態を維持する必要があります。

SASL PLAIN がブローカー間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 つのメカニズムで構成されます。これらのメカニズムは、使用されるハッシュアルゴリズム (SHA-256 とより強力な SHA-512) が違うだけです。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 つのみをブローカー間の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することができます。たとえば、以下のコマンドを実行して、パスワード 123456 で user1 を追加します。

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

ユーザークレデンシャルを削除するには、以下のコマンドを使用します。

```
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
```

マルチ 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。

4.9.5. TLS クライアント認証の有効化

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

前提条件

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

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。

その他のリソース

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

4.9.6. SASL PLAIN 認証の有効化

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

前提条件

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

手順

1. `/opt/kafka/config/jaas.conf` JAAS 設定ファイルを編集するか、または作成します。このファイルには、すべてのユーザーとユーザー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ようにします。
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に **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. JAAS 設定を Kafka ブローカーに渡す `KAFKA_OPTS` 環境変数を使用して、Kafka ブローカーを(再)起動します。

```

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

```

その他のリソース

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

4.9.7. SASL SCRAM 認証の有効化

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

前提条件

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

手順

1. `/opt/kafka/config/jaas.conf` JAAS 設定ファイルを編集するか、または作成します。**KafkaServer** コンテキストの **ScramLoginModule** を有効にします。このファイルがすべての Kafka ブローカーで同じであるようにします。
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に **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. JAAS 設定を Kafka ブローカーに渡す `KAFKA_OPTS` 環境変数を使用して、Kafka ブローカーを(再)起動します。

```
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
```

その他のリソース

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

4.9.8. SASL SCRAM ユーザーの追加

この手順では、SASL SCRAM を使用する認証に新規ユーザーを追加する方法を説明します。

前提条件

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

手順

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

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

以下に例を示します。

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

その他のリソース

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

4.9.9. SASL SCRAM ユーザーの削除

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

前提条件

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

手順

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

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

以下に例を示します。

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

その他のリソース

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

4.10. OAUTH 2.0 トークンベース認証の使用

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

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

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



注記

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

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

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

AMQ Streams のコンテキストでは以下が行われます。

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

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

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

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

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

kafka-oauth-client

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

kafka-oauth-common

kafka-oauth-client ライブラリー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。依存関係のライブラリーは、今後のバージョンで変更される可能性があります。

OAuth コールバックハンドラー は Kafka Client Java ライブラリーに提供されるので、Java クライアン

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

その他のリソース

- [OAuth 2.0 のサイト](#)

4.10.1. OAuth 2.0 認証メカニズム

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

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

必要な場合は、同じ OAuth 認証リスナー設定で OAUTHBEARER と PLAIN を一緒に有効にできます。

OAUTHBEARER の概要

Kafka は OAUTHBEARER 認証メカニズムをサポートしますが、明示的に設定する必要があります。多くの Kafka クライアントツールでは、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。

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ートークンの形式を取ります。コールバックを使用して、以下の 3 つの方法のいずれかでトークンの提供を設定できます。

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

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

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



注記

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

PLAIN の概要

PLAIN は、すべての Kafka クライアントツール (kafkacat などの開発者ツールを含む) によってサポー

トされる簡易認証メカニズムです。PLAIN を OAuth 2.0 認証と共に使用できるようにするため、AMQ Streams on RHEL にはサーバー側のコールバックが含まれています。PLAIN の AMQ Streams 実装は、**OAuth 2.0 over PLAIN**と呼ばれます。

OAuth 2.0 over PLAIN では、クライアントクレデンシャルは ZooKeeper に保存されません。代わりに、OAUTHBEARER 認証が使用される場合と同様に、クライアント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 およびシークレット (OAuth 2.0 クライアントクレデンシャルメカニズムを使用)
- 設定時に手動で取得された有効期限の長いアクセストークン

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

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

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

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

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

4.10.1.1. プロパティ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 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/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** ファイルに **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 プロパティを省略することができます。
大文字で始める、または大文字の環境変数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](#)

4.10.2. 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。

4.10.2.1. 承認サーバーの OAuth 2.0 クライアント設定

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

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



注記

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

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

Kafka クラスターで OAuth 2.0 認証を使用するには、Kafka **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 がブローカー間の通信に使用される 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の検証用にブローカーを設定できます。

- 高速なローカルトークン検証: 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が適用される最小の設定を示しています。これは、ブローカー間の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、リスナーの名前で大文字と小文字の区別はありません。ここではリスナー名を **CLIENT** としているので、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 ⓯

```

- ❶ SASL でのクレデンシャル変換用に **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用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** が 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.*** プロパティの一部の場合には小文字で指定します。
- ❺ ブローカー間の通信の **OAUTHBEARER** メカニズムを指定します。
- ❻ ブローカー間の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ブローカー間の通信の認証設定を設定します。**oauth.client.id**、**oauth.client.secret**、および **auth.token.endpoint.uri** プロパティはブローカー間の設定に関連します。
- ❾ 有効な発行者の 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**)。
- ❿ JWKS エンドポイント URL (例: **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs**)。
- ⓫ 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。本番環境では常に HTTPSs を使用してください (例: **https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token**)。
- 15 ブローカー間の通信に OAuth 2.0 認証を有効にします (この場合にのみ必要)。
- 16 (任意設定) トークンの有効期限が切れるとセッションの終了を強制し、**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 がブローカー間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 リスナーの設定

```
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 ブローカー間の通信とクライアントアプリケーションには、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ンネルを通じて SASL を使用するように設定します。実稼働環境では、クライアントは暗号化されたチャンネル (**SASL_SSL**) を使用できます。
- 3 **ssl**. プロパティは TLS 設定を定義します。
- 4 乱数ジェネレーターの実装。これが設定されていない場合、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

勿口、ノノオル下但は|||||、ノハ 吐切首の小入下口の快吐で強叩シヨ。

- リスナーのキーストアへのパス。
- リスナーのトラストストアへのパス。
- (ブローカー間の接続に使用される) 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。
- OAuth 2.0 の **CLIENT** リスナーを設定します。承認サーバーとの接続は、セキュアな HTTPS 接続を使用する必要があります。

以下の例は、SASL を介したクレデンシャル交換の PLAIN 認証メカニズムを使用した OAuth 2.0 認証の最小設定を示しています。高速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 ❶
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT ❷
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN ❸
sasl.mechanism.inter.broker.protocol=OAUTHBEARER ❹
inter.broker.listener.name=CLIENT ❺
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOauthValidatorCallbackHandler ❻
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ ❼
  oauth.valid.issuer.uri="http://AUTH_SERVER/auth/realms/REALM" \ ❽
  oauth.jwks.endpoint.uri="https://AUTH_SERVER/auth/realms/REALM/protocol/openid-connect/certs" \ ❾
  oauth.username.claim="preferred_username" \ ❿
  oauth.client.id="kafka-broker" \ ⓫
  oauth.client.secret="kafka-secret" \ ⓬
  oauth.token.endpoint.uri="https://AUTH-SERVER-ADDRESS/token" ; ⓭
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler ⓮
listener.name.client.plain.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.plain.JaasServerOauthOverPlainValidatorCallbackHandler ⓯
listener.name.client.plain.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \ ⓰
  oauth.valid.issuer.uri="https://AUTH-SERVER-ADDRESS" \ ⓱
  oauth.jwks.endpoint.uri="https://AUTH-SERVER-ADDRESS/jwks" \ ⓲
  oauth.username.claim="preferred_username" \ ⓳
  oauth.token.endpoint.uri="http://AUTH_SERVER/auth/realms/REALM/protocol/openid-connect/token" ; ⓴
connections.max.reauth.ms=3600000 ⓵
```

- 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、ブローカー間の通信にも使用されます。

- 2

暗号化されていないチャンネルで SASL を使用するように例の **CLIENT** リスナーを設定します。実稼働環境では、TCP 接続層での盗聴を避けるために、クライアントは暗号化チャンネル

- 3 SASL を介した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。ブローカー間の通信に必要であるため、**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。

PLAIN 認証は、すべてのプラットフォームのすべてのクライアントでサポートされます。Kafka クライアントは **PLAIN** メカニズムを有効にし、**username** および **password** を設定する必要があります。**PLAIN** を使用すると、OAuth アクセストークンを使用するか、OAuth **clientId** および **secret** (クライアントクレデンシャル) を使用して認証できます。動作は、**oauth.token.endpoint.uri** が指定されているかどうかによって追加で制御されます。

oauth.token.endpoint.uri を指定し、クライアントが文字列 **\$accessToken:** で始まるように **password** を設定した場合、サーバーはパスワードをアクセストークンとして解釈し、**username** をアカウントのユーザー名として解釈します。それ以外の場合は、**username** は **clientId** として解釈され、**password** はクライアント名のアクセストークンの取得に使用するためにブローカーが使用するクライアント **secret** として解釈されます。

oauth.token.endpoint.uri が指定されていない場合、**password** は常にアクセストークンとして解釈され、**username** は常にアカウントユーザー名として解釈されます。これは、トークンから抽出されたプリンシパル ID と一致する必要があります。これは、クライアントが常に独自でアクセストークンを取得する必要があり、**clientId** および **secret** を使用できないため、「no-client-credentials」モードと呼ばれます。

- 4 ブローカー間の通信の **OAUTHBEARER** 認証メカニズムを指定します。
- 5 ブローカー間の通信のリスナー (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ブローカー間の通信の認証設定を設定します。**oauth.client.id**、**oauth.client.secret**、および **oauth.token.endpoint.uri** プロパティはブローカー間の設定に関連します。
- 8 有効な発行者の 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 ブローカーのシークレット (すべてのブローカーで同じ)。
- 13 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。実稼働環境の場合は、常に HTTPS を使用してください (例: <https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token>)。
- 14 ブローカー間の通信に 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 有効な発行者の 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。本番環境では常に HTTPSs を使用してください (例: `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token`)。

3 項に示すように `clientId` および `secret` を `username` および `password` として渡すことでクライアントの認証を可能にする、PLAIN メカニズムの追加設定。指定のない場合、アクセストークンを `password` パラメーターとして渡すことでのみ、クライアントは PLAIN 経由で認証できます。

- 21 (任意設定) トークンの有効期限が切れるとセッションの終了を強制し、[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。

4.10.2.3. 高速なローカル JWT トークン検証の設定

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

ローカルチェックでは、トークンに対して以下が確認されます。

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

承認サーバーによって発行されなかったすべてのトークンが拒否されるよう、リスナーの設定時に **有効な発行者の 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 の承認サーバーによって公開される **JWK エンドポイントの 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 有効な発行者の 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。キーの更新はエクスポネンシャルバックオフ (exponential backoff) のルールに従い、`oauth.jwks.refresh.seconds` に到達するまで、一時停止を増やして失敗した更新の再試行を行います。デフォルト値は 1 です。
- ⑤ JWK 証明書が期限切れになる前に有効とみなされる期間。デフォルトは **360** 秒 です。デフォルトよりも長い時間を指定する場合は、無効になった証明書へのアクセスが許可されるリスクを考慮してください。
- ⑥ トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。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。

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

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

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

イントロスペクションエンドポイントのプロパティー例

```
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の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。

4.10.3. Kafka ブローカーの再認証の設定

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

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

リスナーごとにセッションの再認証を指定できます。以下に例を示します。

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

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

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

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 による方法を使用する PLAIN の場合:

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

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

その他のリソース

- [「OAuth 2.0 Kafka ブローカーの設定」](#)
- [「Kafka ブローカーの OAuth 2.0 サポートの設定」](#)
- [KIP-368](#)

4.10.4. OAuth 2.0 Kafka クライアントの設定

Kafka クライアントは以下のいずれかで設定されます。

- 有効なアクセストークンを取得するために承認サーバーとの認証に必要なクレデンシャル (クライアント ID およびシークレット)
- 承認サーバー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。承認サーバーとの認証に使用されるクレデンシャルは、ブローカーに送信 **されません**。クライアントによるアクセストークンの取得後、承認サーバーと通信する必要はありません。

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



注記

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

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

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

4.10.5. OAuth 2.0 のクライアント認証フロー

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

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

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

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

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

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

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

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



注記

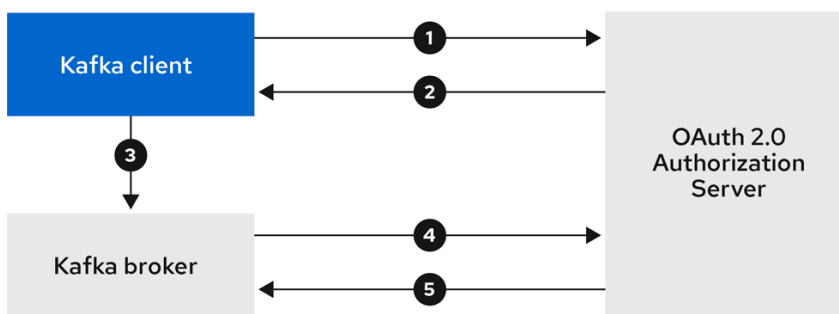
承認サーバーは不透明なアクセストークンの使用のみを許可する可能性があり、この場合はローカルトークンの検証は不可能です。

4.10.5.1. クライアント認証フローの例

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

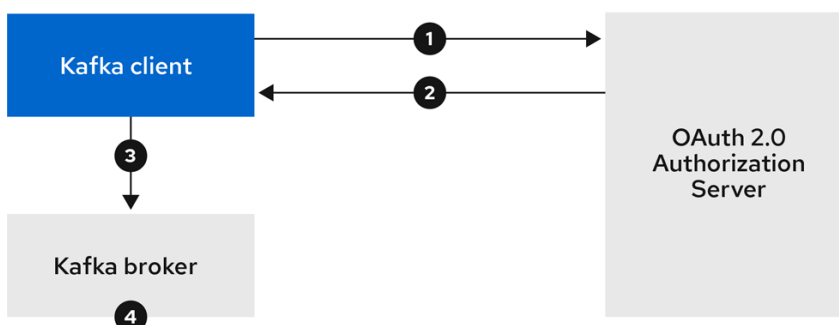
- クライアントではクライアント ID とシークレットが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントではクライアント ID およびシークレットが使用され、ブローカーによって高速のローカルトークン検証が実行される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって高速のローカル検証が実行される場合。

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



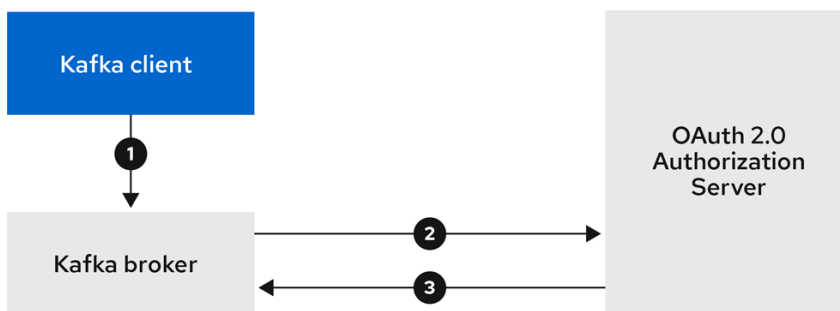
1. Kafka クライアントは承認サーバーからアクセストークンを要求します。これにはクライアント ID とシークレットを使用し、任意で更新トークンも使用します。
2. 承認サーバーによって新しいアクセストークンが生成されます。
3. Kafka クライアントは **SASL OAUTHBEARER** メカニズムを使用してアクセストークンを渡し、Kafka ブローカーの認証を行います。
4. Kafka ブローカーは、独自のクライアント ID およびシークレットを使用して、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。
5. トークンが有効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

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



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

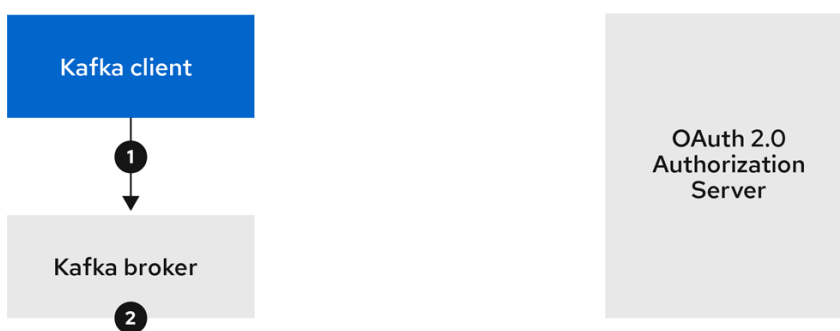
クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合



AMQ_46_1019

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

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



AMQ_46_1019

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



警告

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

4.10.6. OAuth 2.0 認証の設定

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

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

1. [AMQ Streams クラスターおよび Kafka クライアントの OAuth 2.0 承認サーバーを設定します。](#)
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します。](#)
3. [OAuth 2.0 を使用するように Java ベースの Kafka クライアントを更新します。](#)

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

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

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

AMQ Streams で OAuth 2.0 を使用するには、認証レルムを作成および管理できる承認サーバーのデプロイメントが必要です。



注記

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

作業を開始する前の注意事項

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

インストールおよび管理の手順は、以下を参照してください。

- [Server Installation and Configuration Guide](#)
- [Server Administration Guide](#)

前提条件

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

Red Hat Single Sign-On デプロイメントに関する条件:

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

手順

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

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

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

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

選択した承認サーバーおよび実装している承認のタイプをサポートするプロパティを使用して、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](#)
- [暗号化と認証](#)

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

手順

1. **server.properties** ファイルで Kafka ブローカーリスナー設定を設定します。
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接続設定を定義します。
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することができます。

```

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

```

このチェックは、クライアントによる承認サーバーへの接続が信頼できることを確認します。実稼働以外の環境で検証をオフにすることもできます。

4. 選択した認証フローに従って、追加のプロパティを設定します。

```

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://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

```

- 1 承認サーバーへの OAuth 2.0 トークンエンドポイント URL。本番環境では常に HTTPs を使用してください。KeycloakRBACAuthorizer を使用する場合、またはブローカー間の通信に 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 (任意設定) **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 (任意設定): トークンエンドポイントに渡される **scope** パラメーター。スコープは、ブローカー間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** および **secret** を使用した OAuth 2.0 over PLAIN クライアント認証のクライアントの名前でも使用されます。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 (任意設定) **Audience のチェック**。承認サーバーによって **aud** (オーディエンス) クレームが提供され、オーディエンスチェックを強制する場合は、**oauth.check.audience** を **true** に設定します。オーディエンスチェックによって、トークンの目的の受信者が特定されます。そのため、Kafka ブローカーによって、**aud** クレームに **clientId** のないトークンは拒否されます。デフォルトは **false** です。
 - 5 (任意設定) 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、ブローカー間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** および **secret** を使用した OAuth 2.0 over PLAIN クライアント認証のクライアントの名前でも使用されます。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 有効な発行者の 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 (常に必須です)。
 - 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一の、Kafka ブローカーの設定されたシークレット。ブローカーが承認サーバーに対して認証する必要がある場合は、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 (任意設定) Kafka ブローカーの有効期間の長い更新トークン。
 - 10 (任意設定) Kafka ブローカーの有効期間の長いアクセストークン。
5. OAuth 2.0 認証の適用方法や、使用される承認サーバーのタイプに応じて、さらに設定を追加します。

```

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

```

```
# ...
oauth.check.issuer=false \ ❶
oauth.fallback.username.claim="CLIENT-ID" \ ❷
oauth.fallback.username.prefix="CLIENT-ACCOUNT" \ ❸
oauth.valid.token.type="bearer" \ ❹
oauth.userinfo.endpoint.uri="https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/userinfo" ; ❺
```

- ❶ 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。このような場合には、**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。
- ❷ 承認サーバーは、通常ユーザーとクライアントの両方を識別する単一の属性を提供しない場合があります。クライアントが独自の名前で認証される場合、サーバーによって **クライアント ID** が提供されることがあります。更新トークンまたはアクセストークンを取得するために、ユーザー名およびパスワードを使用してユーザーが認証される場合、サーバーによってクライアント ID の他に **ユーザー名** が提供されることがあります。プライマリーユーザー ID 属性が使用できない場合は、このフォールバックオプションで、使用するユーザー名クレーム (属性) を指定します。
- ❸ **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。
- ❹ (**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す。
- ❺ (**oauth.introspection.endpoint.uri** を使用する場合のみ該当): イン트로スペクションエンドポイントの応答に識別可能な情報が含まれないように、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。

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

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

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

前提条件

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

手順

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

```
<dependency>
  <groupId>io.strimzi</groupId>
  <artifactId>kafka-oauth-client</artifactId>
  <version>0.8.1.redhat-00004</version>
</dependency>
```

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

```
System.setProperty(ClientConfig.OAUTH_TOKEN_ENDPOINT_URI, "https://AUTH-
SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token"); 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
```

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

3. Kafka クライアント設定の TLS 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");
props.put("saslmecanism", "PLAIN");
```

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

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

4.11. OAUTH 2.0 トークンベース承認の使用

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

その他のリソース

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

4.11.1. OAuth 2.0 の承認メカニズム

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

4.11.1.1. Kafka ブローカーのカスタムオーソライザー

AMQ Streams では、Red Hat Single Sign-On の **オーソライザー (KeycloakRBACAuthorizer)** が提供されます。Red Hat Single Sign-On によって提供される Authorization Services で Red Hat Single Sign-

On REST エンドポイントを使用できるようにするには、Kafka ブローカーでカスタムオーソライザーを設定します。

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

4.11.2. OAuth 2.0 承認サポートの設定

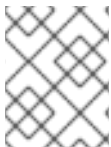
この手順では、Red Hat Single Sign-On の Authorization Services を使用して、OAuth 2.0 承認を使用するように Kafka ブローカーを設定する方法を説明します。

作業を始める前に

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

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

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



注記

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

前提条件

- AMQ Streams は、[トークンベースの認証](#) に Red Hat Single Sign-On と OAuth 2.0 を使用するように設定されている必要があります。承認を設定するときに、同じ Red Hat Single Sign-On サーバーエンドポイントを使用する必要があります。
- [Red Hat Single Sign-On 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する必要があります。

手順

1. Red Hat Single Sign-On の Admin Console にアクセスするか、Red Hat Single Sign-On の Admin CLI を使用して、OAuth 2.0 認証の設定時に作成した Kafka ブローカークライアントの Authorization Services を有効にします。
2. 承認サービスを使用して、クライアントのリソース、承認スコープ、ポリシー、およびパーミッションを定義します。
3. ロールとグループをユーザーとクライアントに割り当てて、パーミッションをユーザーとクライアントにバインドします。
4. Red Hat Single Sign-On 承認を使用するように Kafka ブローカーを設定します。
以下を Kafka **server.properties** 設定ファイルに追加し、Kafka 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 Service 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
```

- ❶ Red Hat Single Sign-On への OAuth 2.0 トークンエンドポイントの URL。本番環境では常に HTTPSs を使用してください
- ❷ 承認サービスが有効になっている Red Hat Single Sign-On の OAuth 2.0 クライアント定義のクライアント ID。通常、**kafka** が ID として使用されます。

6. (任意設定) 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" ❶
```

- ❶ 特定の Kafka クラスターの名前。名前はミッションのターゲットに使用され、同じ Red Hat Single Sign-On レalm内で複数のクラスターを管理できます。デフォルト値は **kafka-cluster** です。

7. (任意設定) 簡易承認に委譲されます。
以下に例を示します。

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

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

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

```
strimzi.authorization.ssl.truststore.location=<path-to-truststore> ❶  
strimzi.authorization.ssl.truststore.password=<my-truststore-password> ❷  
strimzi.authorization.ssl.truststore.type=JKS ❸  
strimzi.authorization.ssl.secure.random.implementation=SHA1PRNG ❹  
strimzi.authorization.ssl.endpoint.identification.algorithm=HTTPS ❺
```

- ❶ 証明書が含まれるトラストストアへのパス。
- ❷ トラストストアのパスワード。
- ❸ 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 `JKS` が使用されます。

ノが使用されます。

- 4 乱数ジェネレーターの実装。これが設定されていない場合、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. (任意設定) 承認サーバーからの付与 (Grants) の更新を設定します。付与の更新ジョブは、アクティブなトークンを列挙し、それぞれに対する最新の付与を要求することで機能します。以下に例を示します。

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

- 1 承認サーバーからの付与の一覧を更新する頻度を指定します (デフォルトでは1分ごと)。デバッグの目的で付与の更新をオフにするには、**"0"** に設定します。
- 2 付与の更新ジョブによって使用されるスレッドプールのサイズ (並列処理のレベル) を指定します。デフォルト値は **"5"** です。

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

4.12. OPA ポリシーベースの承認の使用

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

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



注記

Red Hat は OPA サーバーをサポートしません。

その他のリソース

- [Open Policy Agent の Web サイト](#)

4.12.1. OPA ポリシーの定義

OPA と AMQ Streams を統合する前に、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ます。

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、以下の項目を指定します。

- 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**

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

この例では、トピックへの書き込みが許可されるには、プロデューサークライアントはポリシーを満たす必要があります。

4.12.2. OPA への接続

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

このプラグインは、ポリシーに対してチェックされる JSON 形式のクライアント要求 (ユーザープリンシパル、操作、およびリソース) の詳細を提供します。詳細には、クライアントの一意のアイデンティティが含まれます。これは、たとえば、TLS 認証を使用する場合はクライアント証明書から取得した識別名です。

OPA は、データを使用してプラグインに応答 (**true** または **false** のいずれか) を提供し、リクエストを許可または拒否します。

4.12.3. OPA 承認サポートの設定

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

作業を始める前に

特定のユーザーに必要なアクセス、または制限するアクセスについて検討してください。**ユーザー** および **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 サーバーが利用可能である必要があります。
- [Kafka 用の 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に、クライアント要求の承認に必要な 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 ❶
opa.authorizer.allow.on.error=false ❷
opa.authorizer.cache.initial.capacity=50000 ❸
opa.authorizer.cache.maximum.size=50000 ❹
opa.authorizer.cache.expire.after.seconds=600000 ❺
super.users=User:alice;User:bob ❻
```

- ❶ (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の URL。この例では、ポリシーは **allow** という名前です。
- ❷ 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、クライアントのアクセスをデフォルトで許可または拒否するかどうかを指定するフラグ。
- ❸ 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対してプラグインが OPA ポリシーエンジンにクエリーを行う必要がないように、キャッシュが使用されます。
- ❹ ローカルキャッシュの最大容量 (バイト単位)。
- ❺ OPA ポリシーエンジンからリロードすることでローカルキャッシュが更新される時間 (ミリ秒単位)。
- ❻ スーパーユーザーとして扱われるユーザープリンシパルのリスト。このリストのユーザープリンシパルは、Open Policy Agent ポリシーをクエリーしなくても常に許可されます。

認証および承認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。

4.13. 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。

4.13.1. 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、各ブローカーの複数の **ブローカーロガー** によって提供されます。ブローカーを再起動することなく、ブローカーロガーのロギングレベルを動的に変更できます。ログで返される詳細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 BOOTSTRAP-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
```

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

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

3. 1つ以上のブローカーロガーのロ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 BOOTSTRAP-ADDRESS --alter --add-config "LOGGER-ONE=NEW-LEVEL,LOGGER-TWO=NEW-LEVEL" --entity-type broker-loggers --entity-name BROKER-ID
```

たとえば、ブローカー **0** の場合:

```
/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
```

その他のリソース

- Apache Kafka ドキュメントの「[Updating Broker Configs](#)」

第5章 トピック

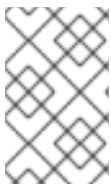
Kafka のメッセージは、常にトピックとの間で送受信されます。本章では、Kafka トピックを設定および管理する方法を説明します。

5.1. パーティションおよびレプリカ

Kafka のメッセージは、常にトピック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、トピックを容易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、クラスター内の異なるブローカーに保存される1つまたは複数のレプリカがあります。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。ある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するのと、コンシューマーがメッセージを消費するのに使用されます。他のレプリカはフォロワーレプリカです。フォロワーはリーダーをレプリケートします。

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。

5.2. 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、時間あるいは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、以下のようにメッセージを保存するように定義できます。

- 7 日間
- パーティションに 1 GB のメッセージが保存されるまで。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 1 GB の制限に達するまで。いずれか最初に達した方の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、新しい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、古い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。

また、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。

圧縮された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多少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。

5.3. トピックの自動作成

プロデューサーまたはコンシューマーが存在してい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と、Kafka はデフォルトでそのトピックを自動的に作成します。この動作は、デフォルトで **true** に設定されている **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
```

5.4. 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
```

5.5. トピックの設定

自動作成されたトピックは、ブローカー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に関する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

保存したメッセージに使用す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

書き込みが正常であると考えられるために、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

ログセグメントを保持する最大期間 (ミリ秒単位)。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 日間) 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。

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

自動作成される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 ブローカー設定で指定できます。

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。

サポートされるすべての Kafka ブローカー設定オプションの一覧は、「[付録A ブローカー設定パラメーター](#)」を参照してください。

5.6. 内部トピック

内部トピックは、Kafka ブローカーおよびクライアントによって内部で作成され、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、プレフィックス **offsets.topic** および **transaction.state.log** で始まる専用の Kafka ブローカー設定オプションを使用して設定できます。最も重要な設定オプションは以下のとおりです。

offsets.topic.replication.factor

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

`offsets.topic.num.partitions`

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

`transaction.state.log.replication.factor`

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

`transaction.state.log.num.partitions`

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

`transaction.state.log.min.isr`

`__transaction_state` トピックへの書き込み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。

5.7. トピックの作成

kafka-topics.sh ツールを使用してトピックを管理できます。**kafka-topics.sh** 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** オプション: 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。

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

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

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

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

mytopic というトピックを記述するコマンドの例

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

その他のリソース

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

5.8. トピックの一覧表示および説明

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、トピックを記述します。

- **--bootstrap-server** オプション: 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。

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

mytopic というトピックを記述するコマンドの例

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

記述コマンドは、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の一覧を表示します。また、すべてのトピック設定オプションも表示されます。

その他のリソース

- トピック設定についての詳細は、「[トピックの設定](#)」を参照してください。
- トピックの作成に関する詳細は、「[トピックの作成](#)」を参照してください。

5.9. トピック設定の変更

kafka-configs.sh ツールを使用して、トピック設定を変更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: 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。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --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

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name mytopic --describe
```

2. **kafka-configs.sh** ツールを使用して、設定を変更します。

- **--bootstrap-server** オプション: 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。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --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

```
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: 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。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --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

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

その他のリソース

- トピック設定についての詳細は、「[トピックの設定](#)」を参照してください。
- トピックの作成に関する詳細は、「[トピックの作成](#)」を参照してください。
- サポートされるすべてのトピック設定オプションの一覧は、「[付録B トピック設定パラメーター](#)」を参照してください。

5.10. トピックの削除

kafka-topics.sh ツールを使用してトピックを管理できます。**kafka-topics.sh** 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** オプション: 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。

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

mytopic というトピックを作成するコマンドの例

```
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。

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

すべてのトピックを一覧表示するコマンドの例

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

その他のリソース

- トピックの作成に関する詳細は、「[トピックの作成](#)」を参照してください。

第6章 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。設定を追加および調整して、AMQ Streams のパフォーマンスに対応できます。たとえば、スループットと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。

6.1. KAFKA 設定のチューニング

設定プロパティを使用して、Kafka ブローカー、プロデューサー、およびコンシューマーのパフォーマンスを最適化します。

最小セットの設定プロパティが必要ですが、プロパティを追加または調整して、プロデューサーとコンシューマーが Kafka ブローカーと対話する方法を変更できます。たとえば、クライアントがリアルタイムでデータに応答できるように、メッセージのレイテンシーおよびスループットをチューニングできます。

メトリックを分析して初期設定を行う場所を判断することから始め、必要な設定になるまで段階的に変更を加え、さらにメトリクスの比較を行うことができます。

その他のリソース

- [Apache Kafka ドキュメント](#)

6.1.1. Kafka ブローカー設定のチューニング

設定プロパティを使用して、Kafka ブローカーのパフォーマンスを最適化します。AMQ Streams によって直接管理されるプロパティを除き、標準の Kafka ブローカー設定オプションを使用できます。

6.1.1.1. 基本的なブローカー設定

基本設定には、ブローカーを特定し、セキュアなアクセスを提供するために以下のプロパティが含まれます。

- **broker.id** は、Kafka ブローカーの ID です。
- **log.dirs** は、ログデータのディレクトリーです。
- **zookeeper.connect** は、Kafka を ZooKeeper に接続する設定です。
- **listener** は Kafka クラスターをクライアントに公開します。
- **authorization** メカニズムはユーザーによって実行されたアクションを許可または拒否します。
- **authentication** メカニズムは Kafka へのアクセスを必要とするユーザーのアイデンティティを証明します。

基本的な設定オプションの詳細は「[Kafka の設定](#)」を参照してください。

通常のブローカー設定には、トピック、スレッド、およびログに関連するプロパティの設定も含まれます。

基本的なブローカープロパティ

```
# ...  
num.partitions=1
```

```

default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...

```

6.1.1.2. 高可用性のためのトピックの複製

基本的なトピックプロパティは、トピックのデフォルト数のパーティションおよびレプリケーション係数を設定します。これは、トピックが自動的に作成される場合を含め、これらのプロパティを明示的に設定せずに作成されたトピックに適用されます。

```

# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
# ...

```

auto.create.topics.enable プロパティはデフォルトで有効になっており、プロデューサーおよびコンシューマーが必要な時にトピックが自動的に作成されます。トピックの自動作成を使用している場合は、**num.partitions** を使用してトピックのデフォルトのパーティション数を設定できます。ただし、このプロパティは無効になっているため、明示的にトピック作成でより多くの制御がトピックで提供されるようになりました。

高可用性環境の場合は、トピックに対してレプリケーション係数を 3 以上に引き上げ、必要な同期レプリカの最小数をレプリケーション係数より 1 少なく設定することをお勧めします。

データの持続性については、トピック設定で **min.insync.replicas** と、プロデューサー設定の **acks=all** を使用してメッセージ配信の謝辞を行う必要があります。

replica.fetch.max.bytes を使用して、リーダーパーティションを複製する各フォロワーが取得したメッセージの最大サイズをバイト単位で設定します。この値は、平均のメッセージサイズおよびスループットに応じて変更します。読み取り/書き込みバッファに必要なメモリー割り当ての合計を考慮する際に、利用可能なメモリーも、すべてのフォロワーで乗算したレプリケートされたメッセージの最大サイズに対応する必要があります。すべてのメッセージをレプリケートできるように、サイズは **message.max.bytes** よりも大きくなければなりません。

delete.topic.enable プロパティはデフォルトで有効になっており、トピックを削除できます。実稼働環境では、誤ってトピックが削除され、データが失われるのを防ぐために、このプロパティを無効にする必要があります。ただし、トピックを一時的に有効にして、トピックを削除してから再度無効にできます。

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

6.1.1.3. トランザクションおよびコミットの内部トピック設定

[トランザクションを使用](#)してプロデューサーからのパーティションへのアトミック書き込みを有効にする場合、トランザクションの状態は内部 **__transaction_state** トピックに保存されます。デフォルトでは、ブローカーはレプリケーション係数が3で設定され、このトピックでは少なくとも2つの同期レプリカが設定されます。つまり、Kafka クラスターには少なくとも3つのブローカーが必要になります。

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
# ...
```

同様に、コンシューマーの状態を保存する内部 **__consumer_offsets** トピックには、パーティションおよびレプリケーション係数の数のデフォルト設定があります。

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

実稼働ではこれらの設定を下げないでください。実稼働環境で設定を大きくすることができます。例外として、単一ブローカーのテスト環境の設定を下げる必要がある場合があります。

6.1.1.4. I/O スレッドの増加によるリクエスト処理スループットの向上

ネットワークスレッドは、クライアントアプリケーションからのリクエストの生成や取得など、Kafka クラスターへのリクエストを処理します。生成リクエストはリクエストキューに配置されます。応答は応答キューに配置されます。

ネットワークスレッドの数は、レプリケーション係数と、Kafka クラスターと、対話するクライアントプロデューサーおよびコンシューマーからのアクティビティのレベルを反映する必要があります。リクエストが多い場合は、スレッドがアイドル状態である時間を使用してスレッドの数を増やし、スレッドを追加するタイミングを決定できます。

輻輳を軽減し、要求トラフィックを規制するには、ネットワークスレッドがブロックされる前に、要求キューで許可されるリクエスト数を制限できます。

I/O スレッドはリクエストキューからリクエストを選択して処理します。スレッド数を増やすとスループットが向上しますが、CPUのコアの数およびディスク帯域幅により、実用的な上限が決まります。最低でも、I/O スレッドの数はストレージボリュームの数と同じでなければなりません。

```
# ...
num.network.threads=3 ❶
queued.max.requests=500 ❷
num.io.threads=8 ❸
num.recovery.threads.per.data.dir=1 ❹
# ...
```

- 1 Kafka クラスターのネットワークスレッドの数。
- 2 リクエストキューで許可されるリクエストの数。
- 3 Kafka ブローカーの I/O スレッドの数。
- 4 起動時のログの読み込みおよびシャットダウン時のフラッシュに使用されるスレッドの数。

すべてのブローカーのスレッドプールへの設定の更新は、クラスターレベルで動的に発生する可能性があります。これらの更新は、現在のサイズの半分から現在のサイズの 2 倍までに制限されます。



注記

Kafka ブローカーメトリクスは、必要なスレッドの数を計算するのに役立ちます。たとえば、ネットワークスレッドがアイドル状態である平均時間のメトリクス (**kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent**) は、使用されたリソースの割合を示します。0% のアイドル時間がある場合、すべてのリソースが使用中であり、スレッドの追加は有益になります。

ディスクの数によりスレッドが遅くなり、制限される場合は、ネットワーク要求のバッファのサイズを増やしてスループットを向上させることができます。

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

また、Kafka が受信可能な最大バイト数も増やします。

```
# ...
socket.request.max.bytes=104857600
# ...
```

6.1.1.5. レイテンシーの高い接続に対する帯域幅の引き上げ

Kafka はデータをバッチ処理して、データセンター間の接続など、Kafka からクライアントへのレイテンシーの高い接続で妥当なスループットを実現します。ただし、レイテンシーの高さが問題である場合、メッセージを送受信するためのバッファのサイズを増やすことができます。

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

帯域幅遅延積の計算を使用して、バッファの最適なサイズを見積もることができます。これは、リンクの最大帯域幅 (バイト/秒) にラウンドトリップ遅延 (秒) を掛けて、最大スループットを維持するために必要なバッファの大きさを見積もります。

6.1.1.6. データ保持ポリシーでのログの管理

Kafka はログを使用してメッセージデータを保存します。ログは、さまざまなインデックスに関連付けられた一連のセグメントです。新しいメッセージは**アクティブ**なセグメントに書き込まれ、その後変更されません。セグメントは、コンシューマーからのフェッチ要求に対応するときに読み取られます。定

期的に、アクティブセグメントが**ロール**されて読み取り専用になり、それを置き換えるために新しいアクティブセグメントが作成されます。一度にアクティブにできるセグメントは1つだけです。古いセグメントは、削除対象となるまで保持されます。

ブローカーレベルでの設定では、ログセグメントの最大サイズをバイト単位で設定し、アクティブなセグメントがロールされるまでの時間をミリ秒単位で設定します。

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

この設定は、**segment.bytes** および **segment.ms** を使用して、トピックレベルで上書きできます。これらの値を下げるまたは上げる必要があるかどうかは、セグメント削除のポリシーによって異なります。サイズが大きいほど、アクティブセグメントに含まれるメッセージが多くなり、ロールされる頻度が少なくなります。セグメントも削除の対象となる頻度が少なくなります。

時間ベースまたはサイズベースのログの保持およびクリーンアップポリシーを設定して、ログを管理しやすくすることができます。要件によっては、ログ保持の設定を使用して古いセグメントを削除できます。ログ保持ポリシーが使用される場合、保持制限に達すると、アクティブではないログセグメントが削除されます。古いセグメントを削除すると、ディスク領域が超過しないように、ログに必要なストレージ領域がバインドされます。

期間ベースのログの保持には、時間、分、およびミリ秒に基づいて保持期間を設定します。保持期間は、メッセージがセグメントに追加された時間に基づいています。

ミリ秒設定は分設定よりも優先され、分設定は時間設定よりも優先されます。分とミリ秒の設定はデフォルトで null ですが、3つのオプションにより、保持するデータを実質的に制御できます。動的に更新できるのは3つのプロパティの1つだけであるため、ミリ秒設定を優先する必要があります。

```
# ...
log.retention.ms=1680000
# ...
```

log.retention.ms が -1 に設定されている場合、ログ保持に適用される時間制限はありません。そのため、すべてのログが保持されます。ディスクの使用状況は常に監視する必要がありますが、-1 の設定は、ディスクがいっぱいになると問題が発生する可能性があり、修正が難しいため、一般的にはお勧めしません。

サイズベースのログの保持には、最大ログサイズ (ログのすべてのセグメント) をバイト単位で設定します。

```
# ...
log.retention.bytes=1073741824
# ...
```

つまり、通常、ログが定常状態に達すると、およそ **log.retention.bytes / log.segment.bytes** の数のセグメントを持ちます。最大ログサイズに達すると、古いセグメントが削除されます。

最大ログサイズの使用に関する潜在的な問題は、メッセージがセグメントに追加された時刻が考慮されていないことです。クリーンアップポリシーに時間ベースおよびサイズベースのログ保持を使用して、必要なバランスをとることができます。どちらのしきい値に最初に到達しても、クリーンアップがトリガーされます。

セグメントファイルがシステムから削除される前に時間遅延を追加する場合は、ブローカーレベルのすべてのトピックには **log.segment.delete.delay.ms**、トピック設定の特定のトピックには **file.delete.delay.ms** を使用して遅延を追加できます。

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

6.1.1.7. クリーンアップポリシーによるログデータの削除

古いログデータを削除する方法は、**ログクリーナー**設定によって決定されます。

ログクリーナーは、ブローカーに対してデフォルトで有効になっています。

```
# ...
log.cleaner.enable=true
# ...
```

クリーンアップポリシーは、トピックまたはブローカーレベルで設定できます。ブローカーレベルの設定は、ポリシーが設定されていないトピックのデフォルトです。

ログの削除、ログの圧縮、その両方を行うためにポリシーを設定できます。

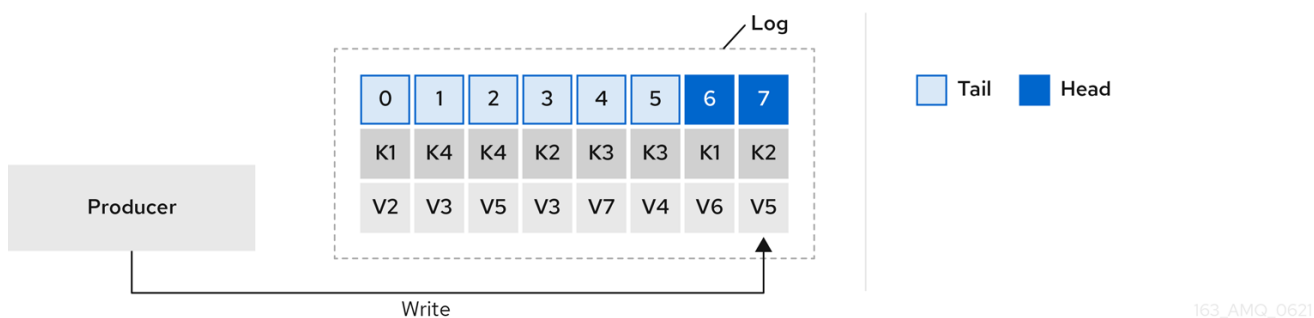
```
# ...
log.cleanup.policy=compact,delete
# ...
```

delete ポリシーは、データ保持ポリシーを使用したログの管理に対応します。データを永久に保持する必要がない場合に適しています。**compact** ポリシーは、各メッセージキーの最新メッセージを保持することを保証します。ログコンパクションは、メッセージ値の変更が可能で、最新の更新を保持する場合に適しています。

ログを削除するようにクリーンアップポリシーが設定されている場合、ログの保持制限に基づいて古いセグメントが削除されます。それ以外の場合、ログクリーナーが有効になっておらず、ログの保持制限がないと、ログは増え続けます。

ログコンパクションにクリーンアップポリシーが設定されている場合、ログの**先頭**は標準の Kafka ログとして機能し、新しいメッセージへの書き込みが順番に追加されます。ログクリーナーが動作する圧縮ログの**末尾**で、同じキーを持つ別のレコードがログの後半で発生した場合、レコードは削除されます。null 値を持つメッセージも削除されます。キーを使用していない場合、関連するメッセージを識別するためにキーが必要になるため、コンパクションを使用することはできません。Kafka は、各キーの最新のメッセージが保持されることを保証しますが、圧縮されたログ全体に重複が含まれないことを保証するものではありません。

図6.1 コンパクション前のオフセットの位置によるキー値の書き込みを示すログ

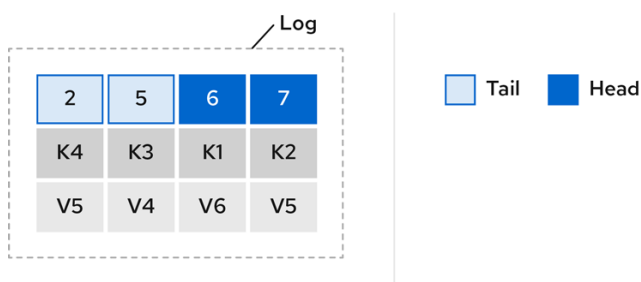


163_AMQ_0621

鍵を使用してメッセージを特定することで、Kafka のコンパクションは特定のメッセージキーの最新メッセージ (オフセットが最大) を維持し、最終的に同じキーを持つ以前のメッセージを破棄します。つまり、最新状態のメッセージは常に利用可能であり、その特定のメッセージの古いレコードは、ログクリーナーの実行時に最終的に削除されます。メッセージを以前の状態に復元できます。

周囲のレコードが削除されても、レコードは元のオフセットを保持します。その結果、末尾は連続しないオフセットを持つ可能性があります。末尾で使用できなくなったオフセットを消費すると、次に高いオフセットを持つレコードが見つかります。

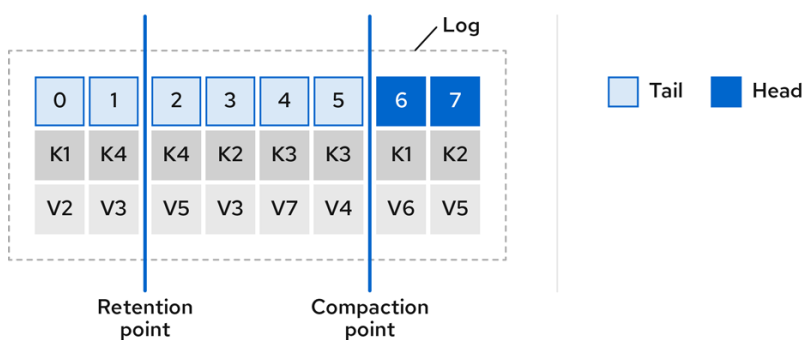
図6.2 コンパクション後のログ



163_AMQ_0621

圧縮ポリシーのみを選択すると、ログが任意に大きくなる可能性があります。この場合、ログの圧縮および削除を行うためにポリシーを設定します。コンパクションおよび削除を選択した場合、まずログデータが圧縮され、ログの先頭にあるキーでレコードが削除されます。その後、ログ保持しきい値より前のデータは削除されます。

図6.3 ログ保持ポイントおよびコンパクションポイント



163_AMQ_0621

ログのクリーンアップがチェックされる頻度をミリ秒単位で設定します。

```
# ...
log.retention.check.interval.ms=300000
# ...
```

ログ保持設定に関連して、ログ保持チェックの間隔を調整します。保持サイズが小さいほど、より頻繁なチェックが必要になる場合があります。

クリーンアップの頻度は、ディスクスペースを管理するのに十分な頻度である必要がありますが、トピックのパフォーマンスに影響を与えるほど頻度を上げてはなりません。

クリーニングするログがない場合にクリーナーをスタンバイにする時間をミリ秒単位で設定することもできます。

```
# ...
log.cleaner.backoff.ms=15000
# ...
```

古いログデータの削除を選択した場合、パージする前に削除されたデータを保持する期間をミリ秒単位で設定できます。

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

削除されたデータの保持期間は、データが完全に削除される前に、データが削除されたことに気付く時間を確保します。

特定のキーに関連するすべてのメッセージを削除するために、プロデューサーは廃棄 (**tombstone**) メッセージを送信できます。廃棄 (**tombstone**) には null 値があり、値が削除されることを示すマーカーとして機能します。コンパクション後に廃棄 (**tombstone**) のみが保持されます。これは、コンシューマーがメッセージが削除されたことを認識するのに十分な期間である必要があります。古いメッセージが削除され、値がないと、**tombstone** キーもパーティションから削除されます。

6.1.1.8. ディスク使用率の管理

ログクリーンアップに関する他の設定には数多くありますが、特に重要なのはメモリー割り当てです。

重複排除 (deduplication) プロパティは、すべてのログクリーナーレッド全体でクリーンアップの合計メモリーを指定します。バッファ負荷係数で使用されるメモリーの割合の上限を設定できます。

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

各ログエントリは正確に 24 バイトを使用するため、バッファが1回の実行で処理できるログエントリ数を計算し、それに応じて設定を調整できます。

可能であれば、ログのクリーニング時間を短縮する場合は、ログクリーナーレッドの数を増やすことを検討してください。

```
# ...
log.cleaner.threads=8
# ...
```

ディスク帯域幅の使用率が100%で問題が発生している場合は、読み書き操作の合計が、操作を実行するディスクの機能に基づいて指定された値の2倍未満になるように、ログクリーナーのI/Oを調整できます。

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

6.1.1.9. 大きなメッセージサイズの処理

メッセージのデフォルトのバッチサイズは1MBで、ほとんどのユースケースで最大のスループットを得るのに最適です。Kafkaは、十分なディスク容量があれば、スループットを下げてもより大きなバッチに対応できます。

大きなメッセージサイズは、以下の4つの方法で処理されます。

1. **プロデューサー側のメッセージ圧縮** は、圧縮メッセージをログに書き込みます。
2. 参照ベースのメッセージングは、メッセージの値で他のシステムに格納されているデータへの参照のみを送信します。
3. インラインメッセージングは、メッセージを同じキーを使用するチャンクに分割し、Kafka Streamsなどのストリームプロセッサを使用して出力に組み合わせられます。
4. より大きなメッセージサイズを処理するように構築されたブローカーおよびプロデューサー/コンシューマクライアントアプリケーション。

リファレンススペースのメッセージングおよびメッセージ圧縮オプションが推奨されます。これはほとんどの状況に対応します。これらのオプションのいずれかを使用する場合は、パフォーマンスの問題が発生しないように注意する必要があります。

プロデューサー側の圧縮

プロデューサー設定では、Gzipなどの**compression.type**を指定します。これはプロデューサーによって生成されたデータのバッチに適用されます。ブローカー設定**compression.type=producer**を使用すると、ブローカーはプロデューサーが使用する圧縮を保持します。プロデューサーとトピックの圧縮が一致しない場合は常に、ブローカーはバッチをログに追加する前に再度圧縮する必要があります。これはブローカーのパフォーマンスに影響を与えます。

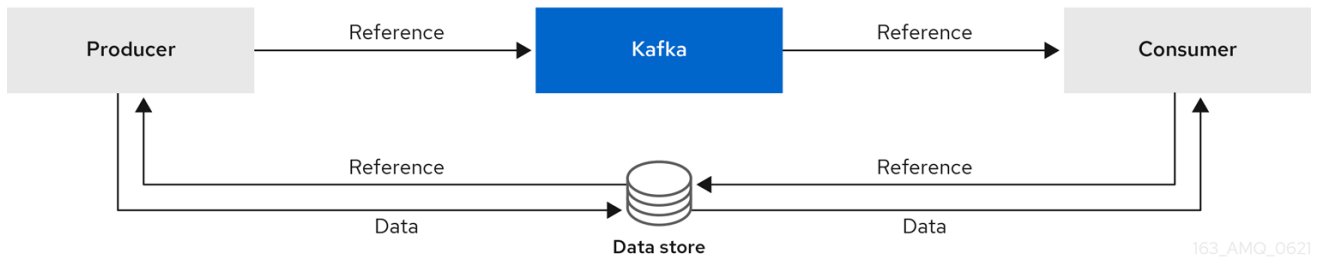
圧縮はまた、プロデューサーに追加の処理オーバーヘッドを追加し、コンシューマーに解凍オーバーヘッドを追加しますが、バッチにより多くのデータが含まれるため、メッセージデータが適切に圧縮される場合、スループットに役立つことがよくあります。

プロデューサー側の圧縮とバッチサイズの微調整を組み合わせ、最適なスループットを促進します。メトリクスを使用すると、必要な平均バッチサイズの測定に役立ちます。

参照ベースのメッセージング

参照ベースのメッセージングは、メッセージの大きさがわからない場合のデータ複製に役立ちます。この設定が機能するには、外部データストアは高速で永続性があり、高可用性である必要があります。データはデータストアに書き込まれ、データへの参照が返されます。プロデューサーは、Kafkaへの参照が含まれるメッセージを送信します。コンシューマーはメッセージから参照を取得し、これを使用してデータストアからデータを取得します。

図6.4 参照ベースのメッセージングフロー



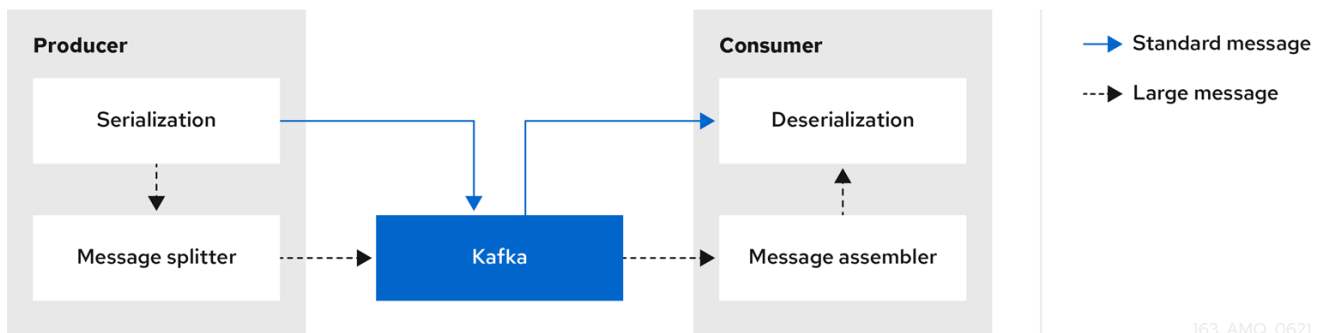
メッセージを渡すにはより多くの通信が必要なため、エンドツーエンドのレイテンシーが増加します。このアプローチのもう1つの重大な欠点は、Kafka メッセージがクリーンアップされたときに、外部システムのデータが自動的にクリーンアップされないことです。ハイブリッドアプローチは、大きなメッセージのみをデータストアに送信し、標準サイズのメッセージを直接処理することです。

インラインメッセージング

インラインメッセージングは複雑ですが、参照ベースのメッセージングのように外部システムに依存するオーバーヘッドはありません。

メッセージが大きすぎる場合、生成するクライアントアプリケーションは、データをシリアル化してからチャンクにする必要があります。その後、プロデューサーはKafkaの **ByteArraySerializer** または同様のものを使用して、各チャンクを再度シリアル化してから送信します。コンシューマーはメッセージを追跡し、完全なメッセージが得られるまでチャンクをバッファリングします。消費側のクライアントアプリケーションは、デシリアライズの前にアSEMBルされたチャンクを受け取ります。完全なメッセージは、チャンクになったメッセージの各セットの最初または最後のチャンクのオフセットに従って、消費する残りのアプリケーションに配信されます。リバランス中の重複を避けるために、完全なメッセージの正常な配信がオフセットメタデータと照合されます。

図6.5 インラインメッセージングフロー



インラインメッセージングは、特に一連の大きなメッセージを並行して処理する場合に必要なバッファリングのために、コンシューマー側でパフォーマンスのオーバーヘッドが発生します。大きなメッセージのチャンクはインターリーブされる可能性があるため、バッファ内の別の大きなメッセージのチャンクが不完全な場合、メッセージのすべてのチャンクが消費されたときにコミットできるとは限りません。このため、バッファリングは通常、メッセージチャンクを永続化するか、コミットロジックを実装することでサポートされます。

より大きなメッセージを処理するための設定

より大きなメッセージを回避できない場合、およびメッセージフローの任意の時点でブロックを回避するために、メッセージ制限を増やすことができます。これには、トピックレベルで **message.max.bytes** を設定し、個別のトピックに最大レコードバッチサイズを設定します。ブローカーレベルで **message.max.bytes** を設定すると、すべてのトピックに大きなメッセージが許可されます。

ブローカーは、**message.max.bytes** に設定された制限を超えるすべてのメッセージを拒否します。プロデューサー (**max.request.size**) およびコンシューマー (**message.max.bytes**) のバッファサイズは、より大きなメッセージに対応できなければなりません。

6.1.1.10. メッセージデータのログフラッシュの制御

ログフラッシュプロパティは、キャッシュされたメッセージデータのディスクへの定期的な書き込みを制御します。スケジューラーは、ログキャッシュのチェック頻度をミリ秒単位で指定します。

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

メッセージがメモリに保持される最大時間と、ディスクに書き込む前にログに記録されるメッセージの最大数に基づいて、フラッシュの頻度を制御できます。

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

フラッシュ間の待機時間には、チェックを行う時間と、フラッシュが実行される前の指定された間隔が含まれます。フラッシュの頻度を増やすと、スループットに影響を及ぼす可能性があります。

一般に、明示的なフラッシュしきい値を設定せず、オペレーティングシステムにデフォルト設定を使用してバックグラウンドフラッシュを実行させることをお勧めします。パーティションレプリケーションは、障害が発生したブローカーが同期レプリカから回復できるため、単一のディスクへの書き込みよりも優れたデータ耐久性を提供します。

アプリケーションフラッシュ管理を使用している場合、より高速なディスクを使用していると、フラッシュしきい値を低く設定するのが適切であることがあります。

6.1.1.11. 可用性のためのパーティションリバランス

フォールトトレランスのために、パーティションはブローカー間で複製できます。指定したパーティションでは、1つのブローカーがリーダーに選出され、すべての生成リクエストを処理します (ログへの書き込み)。他のブローカーのパーティションフォロワーは、リーダーに障害が発生した場合のデータの信頼性のために、パーティションリーダーのパーティションデータを複製します。

通常、フォロワーはクライアントを提供するわけではありません。**broker.rack** では、Kafka クラスターが複数のデータセンターにまたがる場合に、コンシューマーは最も近いレプリカからメッセージを消費できます。フォロワーは、パーティションリーダーからのメッセージを複製して、リーダーに障害が発生した場合に回復できるようにするためにのみ動作します。リカバリーには、同期のフォロワーが必要です。フォロワーは、フェッチ要求をリーダーに送信することで同期を維持します。リーダーは、メッセージを順番にフォロワーに返します。フォロワーは、リーダーで最後にコミットされたメッセージに追いついた場合に、同期していると見なされます。リーダーは、フォロワーによってリクエストされた最後のオフセットを確認してこれをチェックします。[クリーンでないリーダーの選出 \(unclean leader election\)](#) が許可されない限り、非同期のフォロワーは通常、現在のリーダーが失敗した場合にリーダーとしての資格がありません。

フォロワーが同期していないと見なされるまでのラグタイムを調整できます。

```
# ...
replica.lag.time.max.ms=30000
# ...
```


ラグタイムは、メッセージをすべての同期レプリカにレプリケートする時間と、プロデューサーが確認応答を待機する必要がある時間に上限を設定します。フォロワーがフェッチリクエストの作成に失敗し、指定されたラグタイム内に最新のメッセージに追いつくと、同期レプリカから削除されます。失敗したレプリカをより早く検出するためにラグタイムを短縮することができますが、そうすることで、不必要に同期しなくなるフォロワーの数が増えます。適切なラグタイムの値は、ネットワークレイテンシーとブローカーのディスク帯域幅の両方に依存します。

リーダーパーティションが利用できなくなると、同期レプリカの1つが新しいリーダーとして選択されます。パーティションにあるレプリカの一覧の最初のブローカーは、**優先リーダー**と呼ばれます。デフォルトでは、Kafka はリーダー分散の定期的なチェックに基づいて自動パーティションリーダーリバランスに対して有効になっています。つまり、Kafka は優先リーダーが**現在のリーダー**であるかどうかを確認します。リバランスにより、リーダーがブローカー間で均等に分散され、ブローカーがオーバーロードされないようにします。

AMQ Streams には **Cruise Control** を使用して、クラスター全体で負荷を分散するブローカーへのレプリカ割り当てを判別できます。その計算では、リーダーとフォロワーで発生するさまざまな負荷が考慮されています。リーダーが失敗すると、残りのブローカーが追加のパーティションをリードするという余分な作業が発生するため、Kafka クラスターのバランスに影響を与えます。

Cruise Control によって検出される割り当てがバランスを取るには、パーティションのリーダーが優先リーダーである必要があります。Kafka は、優先リーダーが使用されていることを自動的に確認し (可能な場合)、必要に応じて現在のリーダーを変更します。これにより、クラスターは CruiseControl によって検出されたバランスの取れた状態に保たれます。

リバランスチェックの頻度 (秒単位) と、リバランスがトリガーされる前にブローカーに許可される非バランスの最大率を制御できます。

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

ブローカーのリーダーの非バランスの割合は、ブローカーが現在のリーダーであるパーティションの現在の数と、ブローカーが優先リーダーであるパーティションの数との比率です。優先リーダーが同期状態にあることを前提として、割合をゼロにして、優先リーダーが常に選択されるようにすることができます。

リバランスのチェックでさらに制御が必要な場合は、自動リバランスを無効にすることができます。次に、**kafka-leader-election.sh** コマンドラインツールを使用して、リバランスをトリガーするタイミングを選択できます。



注記

AMQ Streams で提供される Grafana ダッシュボードでは、複製の数が最低数未満のパーティションや、アクティブなリーダーを持たないパーティションのメトリクスが表示されます。

6.1.1.12. クリーンでないリーダーの選出 (unclean leader election)

同期レプリカへのリーダーの選出は、データの損失がないことを保証するため、クリーンであると見なされます。これは、デフォルトで行われます。しかし、リーダーに選出する同期レプリカがない場合はどうなるのでしょうか。おそらく、ISR (同期レプリカ) には、リーダーのディスクが停止したときにのみリーダーが含まれていました。同期レプリカの最小数が設定されておらず、ハードドライブに取り返

しのつかない障害が発生したときにパーティションリーダーと同期しているフォロワーがない場合、データはすでに失われています。それだけでなく、同期しているフォロワーがいないため、**新しいリーダーを選出することはできません。**

Kafka がリーダーの失敗を処理する方法を設定できます。

```
# ...
unclean.leader.election.enable=false
# ...
```

クリーンでないリーダーの選択はデフォルトでは無効になっており、同期されていないレプリカはリーダーになれません。クリーンリーダーの選出では、古いリーダーが失われたときに ISR に他のブローカーがない場合に Kafka はそのリーダーがオンラインに戻るまで待機してから、メッセージの読み書きが行われます。クリーンでないリーダーの選出は、同期していないレプリカがリーダーになる可能性があることを意味しますが、メッセージが失われるリスクがあります。どちらを選択するかは、要件が可用性と耐久性のどちらを優先するかによって異なります。

トピックレベルで特定のトピックのデフォルト設定を上書きできます。データ損失のリスクを許容できない場合は、デフォルト設定のままにします。

6.1.1.13. 不要なコンシューマーグループリバランスの回避

新しいコンシューマーグループに参加するコンシューマーの場合、ブローカーへの不要なリバランスを回避するために遅延を追加できます。

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

この遅延は、コーディネーターがメンバーの参加を待つ期間です。遅延が長いほど、すべてのメンバーが時間内に参加し、リバランスを回避できる可能性が高くなります。ただし、遅延により、期間が終了するまでグループは消費できなくなります。

6.1.2. Kafka プロデューサー設定のチューニング

特定のユースケースに合わせて調整されたオプションのプロパティとともに、基本的なプロデューサー設定を使用します。

設定を調整してスループットを最大化すると、レイテンシーが増加する可能性があり、その逆も同様です。必要なバランスを取得するために、プロデューサー設定を実験して調整する必要があります。

6.1.2.1. 基本のプロデューサー設定

接続およびシリアライザープロパティはすべてのプロデューサーに必要です。通常、追跡用のクライアント ID を追加し、プロデューサーで圧縮してリクエストのバッチサイズを減らすことが推奨されます。

基本的なプロデューサー設定には以下が含まれます。

- パーティション内のメッセージの順序は保証されません。
- ブローカーに到達するメッセージの完了通知は持続性を保証しません。

基本的なプロデューサー設定プロパティ

■


```
# ...
bootstrap.servers=localhost:9092 ❶
key.serializer=org.apache.kafka.common.serialization.StringSerializer ❷
value.serializer=org.apache.kafka.common.serialization.StringSerializer ❸
client.id=my-client ❹
compression.type=gzip ❺
# ...
```

- ❶ (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して Kafka クラスターに接続するようプロデューサーを指示します。プロデューサーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して2つまたは3つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。
- ❷ (必須) メッセージがブローカーに送信される前に、各メッセージの鍵をバイトに変換するシリアライザー。
- ❸ (必須) メッセージがブローカーに送信される前に、各メッセージの値をバイトに変換するシリアライザー。
- ❹ (任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで 사용됩니다。
- ❺ (任意) メッセージを圧縮するコーデック。これは、送信され、圧縮された形式で格納された後、コンシューマーへの到達時に圧縮解除される可能性があります。圧縮はスループットを改善し、ストレージの負荷を減らすのに役立ちますが、圧縮や圧縮解除のコストが異常に高い低レイテンシーのアプリケーションには不適切である場合があります。

6.1.2.2. データの持続性

メッセージ配信の完了通知を使用して、データの持続性を適用し、メッセージが失われる可能性を最小限に抑えることができます。

```
# ...
acks=all ❶
# ...
```

- ❶ **acks=all** と指定すると、パーティションリーダーは、メッセージリクエストが正常に受信されたことを確認する前に、特定数のフォロワーに対してメッセージをレプリケートすることを強制されます。**acks=all** の追加のチェックにより、プロデューサーがメッセージを送信してから完了通知を受信するまでのレイテンシーが増加します。

完了通知がプロデューサーに送信される前にメッセージをログに追加する必要があるブローカーの数は、トピックの **min.insync.replicas** 設定によって決定されます。最初に、トピックレプリケーション係数を3にし、他のブローカーの In-Sync レプリカを2にするのが一般的です。この設定では、単一のブローカーが利用できない場合でもプロデューサーは影響を受けません。2番目のブローカーが利用できなくなると、プロデューサーは完了通知を受信せず、それ以上のメッセージを生成できなくなります。

acks=all をサポートするトピック設定

```
# ...
min.insync.replicas=2 ❶
```

...

- 1 2 In-Sync レプリカを使用します。デフォルトは 1 です。



注記

システムに障害が発生すると、バッファの未送信データが失われる可能性があります。

6.1.2.3. 順序付き配信

メッセージは 1 度だけ配信されるため、べき等プロデューサーは重複を回避します。障害発生時でも配信の順序が維持されるように、ID とシーケンス番号がメッセージに割り当てられます。データの一貫性を維持するために **acks=all** を使用している場合は、順序付き配信にべき等を有効にするのは妥当です。

べき等を使った順序付き配信

```
# ...
enable.idempotence=true 1
max.in.flight.requests.per.connection=5 2
acks=all 3
retries=2147483647 4
# ...
```

- 1 **true** を設定してべき等プロデューサーを有効にします。
- 2 べき等配信では、インフライトリクエストの数が 1 を越えることがありますがメッセージの順序は維持されます。デフォルトのインフライトリクエストの数は 5 です。
- 3 **acks** を **all** に設定します。
- 4 失敗したメッセージリクエストを再送信する試行回数を設定します。

パフォーマンスコストが原因で **acks=all** およびべき等を使用しない場合は、インフライト (完了確認されない) リクエストの数を 1 に設定して、順序を保持します。そうしないと、**Message-A** が失敗し、**Message-B** がブローカーに書き込まれた後にのみ成功する可能性があります。

べき等を使用しない順序付け配信

```
# ...
enable.idempotence=false 1
max.in.flight.requests.per.connection=1 2
retries=2147483647
# ...
```

- 1 **false** を設定して、べき等プロデューサーを無効にします。
- 2 インフライトリクエストの数を確実に 1 に設定します。

6.1.2.4. 信頼性の保証

べき等は、1つのパーティションへの書き込みを1回だけ行う場合に便利です。トランザクションをべき等と使用すると、複数のパーティション全体で1度だけ書き込みを行うことができます。

トランザクションは、同じトランザクション ID を使用するメッセージが1度作成され、すべてがそれぞれのログに書き込まれるか、何も書き込まれないかのどちらかになることを保証します。

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID ❶
transaction.timeout.ms=900000 ❷
# ...
```

- ❶ 一意のトランザクション ID を指定します。
- ❷ タイムアウトエラーが返されるまでのトランザクションの最大許容時間 (ミリ秒単位) を設定します。デフォルトは **900000** (15 分) です。

トランザクションの保証を維持するには、**transactional.id** の選択が重要になります。トランザクション ID は、一意なトピックパーティションセットに使用する必要があります。たとえば、トピックパーティション名からトランザクション ID への外部マッピングを使用したり、競合を回避する関数を使用してトピックパーティション名からトランザクション ID を算出したりすると、これを実現できます。

6.1.2.5. スループットおよびレイテンシーの最適化

通常、システムの要件は、指定のレイテンシー内であるメッセージの割合に対して、特定のスループットのターゲットを達成することです。たとえば、95 % のメッセージが 2 秒以内に完了確認される、1 秒あたり 500,000 個のメッセージをターゲットとします。

プロデューサーのメッセージングセマンティック (メッセージの順序付けと持続性) は、アプリケーションの要件によって定義される可能性があります。たとえば、アプリケーションによって提供される重要なプロパティや保証に反することなく、**acks=0** または **acks=1** を使用するオプションがない可能性があります。

ブローカーの再起動は、パーセンタイルの高いの統計に大きく影響します。たとえば、長期間では、99% のレイテンシーはブローカーの再起動に関する動作によるものです。これは、ベンチマークを設計したり、本番環境のパフォーマンスで得られた数字を使ってベンチマークを行い、そのパフォーマンスの数字を比較したりする場合に検討する価値があります。

目的に応じて、Kafka はスループットとレイテンシーのプロデューサーパフォーマンスを調整するために多くの設定パラメーターと設定方法を提供します。

メッセージのバッチ処理 (**linger.ms** および **batch.size**)

メッセージのバッチ処理では、同じブローカー宛のメッセージをより多く送信するために、メッセージの送信を遅らせ、単一の生成リクエストでバッチ処理できるようにします。バッチ処理では、スループットを増やすためにレイテンシーを長くして妥協します。時間ベースのバッチ処理は **linger.ms** を使用して設定され、サイズベースのバッチ処理は **batch.size** を使用して設定されます。

圧縮処理 (**compression.type**)

メッセージ圧縮処理により、プロデューサー (メッセージの圧縮に費やされた CPU 時間) のレイテ

ンシーが追加されますが、リクエスト (および場合によってはディスクの書き込み) を小さくするため、スループットが増加します。圧縮に価値があるかどうか、および使用に最適な圧縮は、送信されるメッセージによって異なります。圧縮処理は **KafkaProducer.send()** を呼び出すスレッドで発生するため、アプリケーションでこの方法のレイテンシーが問題になる場合は、より多くのスレッドを使用するよう検討してください。

パイプライン処理 (**max.in.flight.requests.per.connection**)

パイプライン処理は、以前のリクエストへの応答を受け取る前により多くのリクエストを送信します。通常、パイプライン処理を増やすと、バッチ処理の悪化などの別の問題がスループットに悪影響を与え始めるしきい値まではスループットが増加します。

レイテンシーの短縮

アプリケーションが **KafkaProducer.send()** を呼び出す場合、メッセージには以下が行われます。

- インターセプターによる処理。
- シリアライズ。
- パーティションへの割り当て。
- 圧縮処理。
- パーティションごとのキューでメッセージのバッチに追加。

ここで、**send()** メソッドが返されます。そのため、**send()** がブロックされる時間は、以下によって決定されます。

- インターセプター、シリアライザー、およびパーティションヤーで費やされた時間。
- 使用される圧縮アルゴリズム。
- 圧縮に使用するバッファの待機に費やされた時間。

バッチは、以下のいずれかが行われるまでキューに残ります。

- バッチが満杯になる (**batch.size** による)。
- **linger.ms** によって導入された遅延が経過。
- 送信者は他のパーティションのメッセージバッチを同じブローカーに送信しようとし、このバッチの追加も可能。
- プロデューサーがフラッシュまたは閉じられる。

バッチ処理とバッファの設定を参照して、レイテンシーをブロックする **send()** の影響を軽減します。

```
# ...
linger.ms=100 ①
batch.size=16384 ②
buffer.memory=33554432 ③
# ...
```

- ① **linger** プロパティは、メッセージの大きなバッチが累積され、リクエストで送信されるように、ミリ秒単位の遅延を追加します。デフォルトは **0** です。

- ② 最大 **batch.size** (バイト単位) が使用された場合、その最大値に達したとき、またはメッセージが **linger.ms** よりも長い期間キューに置かれたとき (いずれか早く発生した方) にリクエストが送信さ
- ③ バッファサイズは、少なくともバッチサイズと同じ大きさである必要があり、バッファ、圧縮、およびインフライトリクエストに対応できる必要があります

スループットの増加

メッセージの配信および送信リクエストの完了までの最大待機時間を調整して、メッセージリクエストのスループットを向上します。

また、カスタムパーティションを作成してデフォルトを置き換えることで、メッセージを指定のパーティションに転送することもできます。

```
# ...
delivery.timeout.ms=120000 ①
partitioner.class=my-custom-partitioner ②
# ...
```

- ① 送信リクエストの完了まで待機する最大時間 (ミリ秒単位)。この値を **MAX_LONG** に設定すると、Kafka に回数無制限の再試行を委譲できます。デフォルトは **120000** (2 分) です。
- ② カスタムパーティショナーのクラス名を指定します。

6.1.3. Kafka コンシューマー設定の調整

特定のユースケースに合わせて調整されたオプションのプロパティとともに、基本的なコンシューマー設定を使用します。

コンシューマーを調整する場合、最も重要なことは、取得するデータ量に効率的に対処できるようにすることです。プロデューサーのチューニングと同様に、コンシューマーが想定どおりに動作するまで、段階的に変更を加える必要があります。

6.1.3.1. 基本的なコンシューマー設定

接続およびデシリアライザープロパティはすべてのコンシューマーに必要です。通常、追跡用にクライアント ID を追加することが推奨されます。

コンシューマー設定では、後続の設定に関係なく、以下を行います。

- メッセージをスキップまたは再読み取りするようオフセットを変更しない限り、コンシューマーはメッセージを指定のオフセットから取得し、順番に消費します。
- オフセットはクラスターの別のブローカーに送信される可能性があるため、オフセットを Kafka にコミットした場合でも、ブローカーはコンシューマーが応答を処理したかどうかを認識しません。

基本的なコンシューマー設定プロパティ

```
# ...
bootstrap.servers=localhost:9092 ①
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer ②
```

```
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer ❸
client.id=my-client ❹
group.id=my-group-id ❺
# ...
```

- ❶ (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して、コンシューマーが Kafka クラスターに接続するよう指示します。コンシューマーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して2つまたは3つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。ロードバランサーサービスを使用して Kafka クラスターを公開する場合、可用性はロードバランサーによって処理されるため、サービスのアドレスのみが必要になります。
- ❷ (必須) Kafka ブローカーから取得されたバイトをメッセージキーに変換するデシリアライザー。
- ❸ (必須) Kafka ブローカーから取得されたバイトをメッセージ値に変換するデシリアライザー。
- ❹ (任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで使われます。ID は、時間クォータの処理に基づいてコンシューマーにスロットリングを適用するために使用することもできます。
- ❺ (条件) コンシューマーがコンシューマーグループに参加するには、グループ ID が **必要** です。

6.1.3.2. コンシューマーグループを使用したデータ消費のスケーリング

コンシューマーグループは、特定のトピックから1つまたは複数のプロデューサーによって生成される、典型的な大量のデータストリームを共有します。コンシューマーは **group.id** プロパティを使用してグループ化され、メッセージをメンバー全体に分散できます。グループ内のコンシューマーの1つがリーダーを選択し、パーティションをグループのコンシューマーにどのように割り当てるかを決定します。各パーティションは1つのコンシューマーにのみ割り当てることができます。

コンシューマーの数がパーティションよりも少ない場合、同じ **group.id** を持つコンシューマーインスタンスを追加して、データの消費をスケーリングできます。コンシューマーをグループに追加して、パーティションの数より多くしても、スループットは改善されませんが、コンシューマーが機能しなくなったときに予備のコンシューマーを使用できます。より少ないコンシューマーでスループットの目標を達成できれば、リソースを節約できます。

同じコンシューマーグループのコンシューマーは、オフセットコミットとハートビートを同じブローカーに送信します。グループのコンシューマーの数が多いほど、ブローカーのリクエスト負荷が高くなります。

```
# ...
group.id=my-group-id ❶
# ...
```

- ❶ グループ ID を使用してコンシューマーグループにコンシューマーを追加します。

6.1.3.3. メッセージの順序の保証

Kafka ブローカーは、トピック、パーティション、およびオフセット位置のリストからメッセージを送信するようブローカーに要求するコンシューマーからフェッチリクエストを受け取ります。

コンシューマーは、ブローカーにコミットされたのと同じ順序でメッセージを単一のパーティションで

監視します。つまり、Kafka は単一パーティションのメッセージ **のみ** 順序付けを保証します。逆に、コンシューマーが複数のパーティションからメッセージを消費している場合、コンシューマーによって監視される異なるパーティションのメッセージの順序は、必ずしも送信順序を反映しません。

1つのトピックからメッセージを厳格に順序付ける場合は、コンシューマーごとに1つのパーティションを使用します。

6.1.3.4. スループットおよびレイテンシーの最適化

クライアントアプリケーションが **KafkaConsumer.poll()** を呼び出すときに返されるメッセージの数を制御します。

fetch.max.wait.ms および **fetch.min.bytes** プロパティを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最小量を増やします。時間ベースのバッチ処理は **fetch.max.wait.ms** を使用して設定され、サイズベースのバッチ処理は **fetch.min.bytes** を使用して設定されます。

コンシューマーまたはブローカーの CPU 使用率が高い場合、コンシューマーからのリクエストが多すぎる可能性があります。リクエストの数を減らし、メッセージがより大きなバッチで配信されるように、**fetch.max.wait.ms** および **fetch.min.bytes** プロパティを調整します。より高い値に調整することでスループットが改善されますが、レイテンシーのコストが発生します。生成されるデータ量が少ない場合、より高い値に調整することもできます。

たとえば、**fetch.max.wait.ms** を 500ms に設定し、**fetch.min.bytes** を 16384 バイトに設定した場合、Kafka がコンシューマーからフェッチリクエストを受信すると、いずれかのしきい値に最初に到達した時点で応答されます。

逆に、**fetch.max.wait.ms** および **fetch.min.bytes** プロパティを低く設定すると、エンドツーエンドのレイテンシーを改善できます。

```
# ...
fetch.max.wait.ms=500 ①
fetch.min.bytes=16384 ②
# ...
```

- ① ブローカーがフェッチリクエストを完了するまで待機する最大時間 (ミリ秒単位)。デフォルトは **500** ミリ秒です。
- ② 最小バッチサイズ (バイト単位) が使用された場合、その最小値に達したとき、またはメッセージが **fetch.max.wait.ms** よりも長い期間キューに置かれたとき (いずれか早く発生した方) にリクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

フェッチリクエストサイズの増加によるレイテンシーの短縮

fetch.max.bytes および **max.partition.fetch.bytes** プロパティを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最大量を増やします。

fetch.max.bytes プロパティは、一度にブローカーから取得されるデータ量の上限をバイト単位で設定します。

max.partition.fetch.bytes は、各パーティションに返されるデータ量の上限をバイト単位で設定します。これは、常に **max.message.bytes** のブローカーまたはトピック設定に設定されたバイト数よりも大きくする必要があります。

クライアントが消費できるメモリの最大量は、以下のように概算されます。

NUMBER-OF-BROKERS * `fetch.max.bytes` and **NUMBER-OF-PARTITIONS** *
`max.partition.fetch.bytes`

メモリー使用量がこれに対応できる場合は、これら2つのプロパティの値を増やすことができます。各リクエストでより多くのデータを許可すると、フェッチリクエストが少なくなるため、レイテンシーが向上されます。

```
# ...
fetch.max.bytes=52428800 ❶
max.partition.fetch.bytes=1048576 ❷
# ...
```

❶ フェッチリクエストに対して返されるデータの最大量 (バイト単位)。

❷ 各パーティションに対して返されるデータの最大量 (バイト単位)。

6.1.3.5. オフセットをコミットする際のデータ損失または重複の回避

Kafka の **自動コミットメカニズム** により、コンシューマーはメッセージのオフセットを自動的にコミットできます。有効にすると、コンシューマーはブローカーをポーリングして受信したオフセットを 5000ms 間隔でコミットします。

自動コミットのメカニズムは便利ですが、データ損失と重複のリスクが発生します。コンシューマーが多くのメッセージを取得および変換し、自動コミットの実行時にコンシューマーバッファに処理されたメッセージがある状態でシステムがクラッシュすると、そのデータは失われます。メッセージの処理後、自動コミットの実行前にシステムがクラッシュした場合、リバランス後に別のコンシューマーインスタンスでデータが複製されます。

ブローカーへの次のポーリングの前またはコンシューマーが閉じられる前に、すべてのメッセージが処理された場合は、自動コミットによるデータの損失を回避できます。

データ損失や重複の可能性を最小限にするには、**`enable.auto.commit`** を **`false`** に設定し、クライアントアプリケーションを開発して、オフセットのコミットをさらに制御します。または、**`auto.commit.interval.ms`** を使用して、コミットの間隔を減らすことができます。

```
# ...
enable.auto.commit=false ❶
# ...
```

❶ 自動コミットを `false` に設定すると、オフセットのコミットの制御が強化されます。

`enable.auto.commit` を **`false`** に設定すると、すべての処理が実行され、メッセージが消費された後にオフセットをコミットできます。たとえば、Kafka **`commitSync`** および **`commitAsync`** コミット API を呼び出すようにアプリケーションを設定できます。

`commitSync` API は、ポーリングから返されるメッセージバッチのオフセットをコミットします。バッチのメッセージすべての処理が完了したら API を呼び出します。**`commitSync`** API を使用する場合、アプリケーションはバッチの最後のオフセットがコミットされるまで新しいメッセージをポーリングしません。これがスループットに悪影響する場合は、コミットする頻度を減らすか、**`commitAsync`** API を使用できます。**`commitAsync`** API はブローカーがコミットリクエストに応答するまで待機しませんが、リバランス時にさらに重複が発生するリスクがあります。一般的なアプローチとして、両方のコミット API をアプリケーションで組み合わせ、コンシューマーをシャットダウンまたはリバランスの直前に **`commitSync`** API を使用し、最終コミットが正常に実行されるようにします。

6.1.3.5.1. トランザクションメッセージの制御

プロデューサー側でトランザクション ID を使用し、べき等 (**enable.idempotence=true**) を有効にして、1 回のみの配信の保証を検討してください。コンシューマー側で、**isolation.level** プロパティを使用して、コンシューマーによってトランザクションメッセージが読み取られる方法を制御できます。

isolation.level プロパティに有効な値は 2 つあります。

- **read_committed**
- **read_uncommitted** (デフォルト)

コミットされたトランザクションメッセージのみがコンシューマーによって読み取られるようにするには、**read_committed** を使用します。ただし、これによりトランザクションの結果を記録するトランザクションマーカー (**committed** または **aborted**) がブローカーによって書き込まれるまで、コンシューマーはメッセージを返すことができないため、エンドツーエンドのレイテンシーが長くなります。

```
# ...
enable.auto.commit=false
isolation.level=read_committed ❶
# ...
```

- ❶ コミットされたメッセージのみがコンシューマーによって読み取られるように、**read_committed** に設定します。

6.1.3.6. データ損失を回避するための障害からの復旧

session.timeout.ms および **heartbeat.interval.ms** プロパティを使用して、コンシューマーグループ内のコンシューマー障害をチェックし、復旧するのにかかる時間を設定します。

session.timeout.ms プロパティは、コンシューマーグループのコンシューマーが非アクティブであるとみなされ、そのグループのアクティブなコンシューマー間でリバランスがトリガーされる前に、ブローカーと通信できない最大時間をミリ秒単位で指定します。グループのリバランス時に、パーティションはグループのメンバーに再割り当てされます。

heartbeat.interval.ms プロパティは、コンシューマーがアクティブで接続されていることを示す、コンシューマーグループコーディネーターへのハートビートチェックの間隔をミリ秒単位で指定します。通常、ハートビートの間隔はセッションタイムアウトの間隔の 3 分の 2 にする必要があります。

session.timeout.ms プロパティの値を低く設定すると、失敗するコンシューマーが早期に発見され、リバランスがより迅速に実行されます。ただし、タイムアウトの値を低くしすぎて、ブローカーがハートビートを時間内に受信できず、不必要なリバランスがトリガーされることがないように気を付けてください。

ハートビートの間隔が短くなると、誤ってリバランスを行う可能性が低くなりますが、ハートビートを頻繁に行うとブローカーリソースのオーバーヘッドが増えます。

6.1.3.7. オフセットポリシーの管理

auto.offset.reset プロパティを使用して、オフセットをすべてコミットしなかった場合やコミットされたオフセットが有効でないまたは削除された場合の、コンシューマーの動作を制御します。

コンシューマーアプリケーションを初めてデプロイし、既存のトピックからメッセージを読み取る場合について考えてみましょう。**group.id** が初めて使用されるため、**__consumer_offsets** トピックには、このアプリケーションのオフセット情報は含まれません。新しいアプリケーションは、ログの始めから

すべての既存メッセージの処理を開始するか、新しいメッセージのみ処理を開始できます。デフォルトのリセット値は、パーティションの最後から開始する **latest** で、一部のメッセージは見逃されることを意味します。データの損失を回避し、処理量を増やすには、**auto.offset.reset** を **earliest** に設定し、パーティションの最初から開始します。

また、ブローカーに設定されたオフセットの保持期間 (**offsets.retention.minutes**) が終了したときにメッセージが失われないようにするため、**earliest** オプションを使用することも検討してください。コンシューマーグループまたはスタンドアロンコンシューマーが非アクティブで、保持期間中にオフセットをコミットしない場合、以前にコミットされたオフセットは **__consumer_offsets** から削除されます。

```
# ...
heartbeat.interval.ms=3000 ❶
session.timeout.ms=10000 ❷
auto.offset.reset=earliest ❸
# ...
```

- ❶ 予想されるリバランスに応じて、ハートビートの間隔を短くして調整します。
- ❷ タイムアウトの期限が切れる前に Kafka ブローカーによってハートビートが受信されなかった場合、コンシューマーはコンシューマーグループから削除され、リバランスが開始されます。ブローカー設定に **group.min.session.timeout.ms** および **group.max.session.timeout.ms** がある場合は、セッションタイムアウト値はこの範囲内である必要があります。
- ❸ パーティションの最初に戻り、オフセットがコミットされなかった場合にデータの損失が発生しないようにするには、**earliest** に設定します。

1つのフェッチリクエストで返されるデータ量が大きい場合、コンシューマーが処理する前にタイムアウトが発生することがあります。この場合は、**max.partition.fetch.bytes** の値を低くするか、**session.timeout.ms** の値を高くします。

6.1.3.8. リバランスの影響を最小限にする

グループのアクティブなコンシューマー間で行うパーティションのリバランスは、以下にかかる時間です。

- コンシューマーによるオフセットのコミット
- 作成される新しいコンシューマーグループ
- グループリーダーによるグループメンバーへのパーティションの割り当て。
- 割り当てを受け取り、取得を開始するグループのコンシューマー

明らかに、このプロセスは特にコンシューマーグループクラスターのローリング再起動時に繰り返し発生するサービスのダウンタイムを増やします。

このような場合、**静的メンバーシップ** の概念を使用してリバランスの数を減らすことができます。リバランスによって、コンシューマーグループメンバー全体でトピックパーティションが割り当てられます。静的メンバーシップは永続性を使用し、セッションタイムアウト後の再起動時にコンシューマーインスタンスが認識されるようにします。

コンシューマーグループコーディネーターは、**group.instance.id** プロパティを使用して指定される一意の ID を使用して新しいコンシューマーインスタンスを特定できます。再起動時には、コンシューマーには新しいメンバー ID が割り当てられますが、静的メンバーとして、同じインスタンス ID を使用

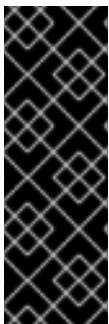
し、同じトピックパーティションの割り当てが行われます。

コンシューマーアプリケーションが最低でも **max.poll.interval.ms** ミリ秒毎にポーリングへの呼び出しを行わない場合、コンシューマーは失敗したと見なされ、リバランスが発生します。アプリケーションがポーリングから返されたすべてのレコードを時間内に処理できない場合は、**max.poll.interval.ms** プロパティを使用して、コンシューマーからの新規メッセージのポーリングの間隔をミリ秒単位で指定して、リバランスの発生を防ぎます。または、**max.poll.records** プロパティを使用して、コンシューマーバッファから返されるレコードの数の上限を設定し、アプリケーションが **max.poll.interval.ms** 内でより少ないレコードを処理できるようにします。

```
# ...
group.instance.id=_UNIQUE-ID_ ❶
max.poll.interval.ms=300000 ❷
max.poll.records=500 ❸
# ...
```

- ❶ 一意のインスタンス ID により、新しいコンシューマーインスタンスに同じトピックパーティションが割り当てられます。
- ❷ コンシューマーがメッセージの処理を継続していることを確認する間隔を設定します。
- ❸ コンシューマーから返される処理済のレコードの数を設定します。

6.2. 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



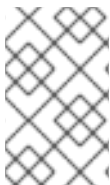
重要

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の機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあります。Red Hat は、本番環境でのテクノロジープレビュー機能の実装は推奨しません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、[「テクノロジ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

- AMQ Streams が、Kafka ブローカーとして使用されるすべてのホストに [インストールされている](#)。
- ZooKeeper クラスタが [設定済みで実行されている](#)。

手順

1. **/opt/kafka/config/server.properties** Kafka 設定ファイルを編集します。
プラグインプロパティーは、この設定例のとおりです。

Kafka Static Quota プラグインの設定例

```
# ...
client.quota.callback.class=io.strimzi.kafka.quotas.StaticQuotaCallback 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1MBps 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1MBps 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
```

その他のリソース

- [Kafka ブローカー設定のチューニング](#)

6.3. クラスターのスケーリング

6.3.1. Kafka クラスターのスケーリング

6.3.1.1. ブローカーのクラスターへの追加

トピックのスループットを向上させる主な方法は、そのトピックのパーティション数を増やすことです。これにより、パーティションによってそのトピックの負荷がクラスター内のブローカー間で共有されるためです。ブローカーがすべて一部のリソース（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するとスループットが向上されません。代わりに、ブローカーをクラスターに追加する必要があります。

追加のブローカーをクラスターに追加する場合、AMQ Streams ではパーティションは自動的に割り当てられません。既存のブローカーから新しいブローカーに移動するパーティションを決定する必要があります。

すべてのブローカー間でパーティションが再分散されたら、各ブローカーはリソース使用率が低くなるはずです。

6.3.1.2. クラスターからのブローカーの削除

クラスターからブローカーを削除する前に、そのブローカーにパーティションが割り当てられていないことを確認する必要があります。廃止されるブローカーの各パーティションを引き継ぐ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、そのパーティションを停止できます。

6.3.2. パーティションの再割り当て

kafka-reassign-partitions.sh ユーティリティは、パーティションを別のブローカーに再割り当てするために使用されます。

これには、以下の 3 つのモードがあります。

--generate

トピックとブローカーのセットを取り、**再割り当て JSON ファイル**を生成します。これにより、トピックのパーティションがブローカーに割り当てられます。**再割り当て JSON ファイルを生成する簡単な方法ですが**、トピック全体で動作するため、使用方法は常に適切ではありません。

--execute

再割り当て JSON ファイルを取り、クラスターのパーティションおよびブローカーに適用します。パーティションを取得するブローカーは、パーティションリーダーのフォロワーになります。特定のパーティションについて、新規ブローカーが ISR をキャッチおよび結合すると、古いブローカーがフォロワーになり、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。

クラスター内で 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、それが完了するまで待機してから、別の再割り

当てを実行して最初の再影響を元に戻す必要があります。**kafka-reassign-partitions.sh** によって、元に戻すための再割り当て JSON が出力の一部として生成されます。大規模な再割り当ては、進行中の再割り当てを停止する必要がある場合に備えて、複数の小さな再割り当てに分割するようにしてください。

6.3.2.1. 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" プロパティは任意で、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。

6.3.2.2. 再割り当て JSON ファイルの生成

特定のトピックセットのすべてのパーティションを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。

```
bin/kafka-reassign-partitions.sh --zookeeper <ZooKeeper> --topics-to-move-json-file <TopicsFile> --broker-list <BrokerList> --generate
```

<TopicsFile> は、移動するトピックを一覧表示する JSON ファイルです。これには、以下の構造があります。

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

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

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

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

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-to-move-json-file topics-to-be-moved.json --broker-list 4,7 --generate
```

topics-to-be-moved.json にはコンテンツが含まれます。

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

6.3.2.3. 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。

6.3.3. 再割り当てスロットル

パーティションの再割り当ては、ブローカー間で大量のデータを移動する必要があるため、処理が遅くなる可能性があります。これによるクライアントへの悪影響を回避するため、再割り当てをスロットルで調整できます。スロットルを使用すると、再割り当てにかかる時間が長くなることがあります。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ためにスループットが低下する可能性があります。

6.3.4. 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に関する再割り当て JSON ファイル。

手順

1. クラスターの他のブローカーと同じ設定を使用して新規ブローカーの設定ファイルを作成します。 **broker.id** は、他のブローカーによって使用されていない数字である必要があります。
2. 前の手順で作成した設定ファイルを **kafka-server-start.sh** スクリプトに引数として渡す新しい Kafka ブローカーを起動します。

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

3. Kafka ブローカーが稼働していることを確認します。

```
jcmd | grep Kafka
```

4. 新しいブローカーごとに上記の手順を繰り返します。
5. **kafka-reassign-partitions.sh** コマンドラインツールを使用して、パーティションの再割り当てを実行します。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** とブローカー間のスロットル率 (バイト/秒単位) を渡すこともできます。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 5000000 --execute
```

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

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

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 10000000 --execute
```


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

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --verify
```

以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --verify
```

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

6.3.5. Kafka クラスターのスケールダウン

その他のリソース

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

前提条件

- 既存の Kafka クラスター。
- ブローカーの削除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に関する再割り当て 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。

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --execute
```

レプリケーションをスロットルで調整する場合、**--throttle** とブローカー間のスロットル率 (バイト/秒単位) を渡すこともできます。以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 5000000 --execute
```

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

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

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --throttle 10000000 --execute
```

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

```
kafka-reassign-partitions.sh --zookeeper <ZooKeeperHostAndPort> --reassignment-json-file <ReassignmentJsonFile> --verify
```

以下に例を示します。

```
kafka-reassign-partitions.sh --zookeeper zookeeper1:2181 --reassignment-json-file reassignment.json --verify
```

4. **--verify** コマンドによって、移動した各パーティションが正常に完了したことが報告されると、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。
5. すべてのパーティションの再割り当てが終了すると、削除されるブローカーはクラスター内のいずれのパーティションにも対応しないはずです。これを検証するには、ブローカーの **log.dirs** 設定パラメーターで指定されている各ディレクトリーをチェックします。ブローカーのログディレクトリーのいずれかに拡張正規表現 **[a-zA-Z0-9.-]+\.[a-z0-9]+-delete\$** に一致しないディレクトリーが含まれる場合、ブローカーにはライブパーティションがあるため、停止しないでください。
これを確認するには、以下のコマンドを実行します。

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

上記のコマンドによって出力が生成される場合、ブローカーにはライブパーティションがあります。この場合、再割り当てが終了していないか、再割り当て JSON ファイルが適切ではありません。

6. ブローカーにライブパーティションがないことが確認できたら、停止できます。

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

7. Kafka ブローカーが停止していることを確認します。

```
jcmd | grep kafka
```

6.3.6. ZooKeeper クラスターのスケールアップ

この手順では、サーバー（ノード）を ZooKeeper クラスターに追加する方法を説明します。ZooKeeper の動的再設定機能は、スケールアッププロセスで安定した ZooKeeper クラスターを維持します。

前提条件

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

手順

各 ZooKeeper サーバーに対して、1つずつ以下の手順を実行します。

1. 「[マルチノードの ZooKeeper クラスターの実行](#)」の説明どおりにサーバーを 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 アドレスおよび設定アクセスポートを書き留めます。
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以下のコマンドを実行します（アクセスがある場合、ZooKeeper ノードまたはローカルマシンのいずれかになります）。

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

4. シェルセッションで以下の行を入力し、新しいサーバーを投票メンバーとしてクォーラムに追加します。

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

以下に例を示します。

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

<positive-id> は新しいサーバー ID **4** に置き換えます。

2つのポートの場合、**<port1> 2888** は ZooKeeper サーバー間の通信用で、**<port2> 3888** はリーダー選択用です。

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対して、ステップ 1-4 を繰り返します。

その他のリソース

- [「ZooKeeper クラスターのスケールダウン」](#)

6.3.7. 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時に安定した ZooKeeper クラスターを維持します。

前提条件

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

手順

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

1. スケールダウン後も保持されるサーバー（例：サーバー 1）で **zookeeper-shell** にログインします。



注記

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

2. サーバー（例：サーバー 5）を削除します。

```
reconfig -remove 5
```

3. 削除したサーバーを非アクティブ化します。
4. ステップ 1-3 を繰り返し、クラスターのサイズを縮小します。

その他のリソース

- [「ZooKeeper クラスターのスケールアップ」](#)
- ZooKeeper [ドキュメントのサーバーの削除](#)

第7章 JMX を使用したクラスターのモニタリング

ZooKeeper、Kafka ブローカー、Kafka Connect、および Kafka [クライアントはすべて](#)、[Java Management Extensions \(JMX\)](#)を使用して管理情報を公開します。ほとんどの管理情報は、Kafka クラスターの状態および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です。その他の Java アプリケーションと同様に、Kafka は管理 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) レベルで動作します。管理情報の取得、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、同じマシン上のツールのみがあり、JVM と同じユーザーで接続できるツールのみです。



注記

ZooKeeper の管理情報は、ここに記載されています。JConsole で ZooKeeper メトリクスを表示できます。[詳細は、「JConsole を使用したモニタリング」](#)を参照してください。

7.1. 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 で提供されるスクリプト (`bin/kafka-server-start.sh` および `bin/connect-distributed.sh` など) は、`KAFKA_JMX_OPTS` 環境変数を使用してこれらのシステムプロパティを設定します。JMX を設定するシステムプロパティは、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複数の異なる方法で JVM を起動します。

7.2. 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と、ローカルの JMX ツールが JVM に接続されないようにすることができます (例: コンプライアンスの理由)。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。

7.3. 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 に接続することができます。これは、JMX ツールで、認証なしでどこからでも接続できるため、これは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 接続がセキュアになるように、認証と SSL を使用することが推奨されます。[これに必要なシステムプロパティの詳細は、JMX のドキュメントを参照してください。](#)

7.4. JCONSOLE を使用したモニタリング

JConsole ツールは、Java Development Kit(JDK)が同梱されています。JConsole を使用して、ローカルまたはリモートの JVM に接続し、Java アプリケーションから管理情報を検出し、表示することができます。JConsole を使用してローカルの JVM に接続する場合は、AMQ Streams の異なるコンポーネントに対応する JVM プロセスの名前。

表7.1 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 が配布される。	org.apache.kafka.connect.cli.ConnectDistributed
Kafka プロデューサー、コンシューマー、または Streams アプリケーション	アプリケーションの main メソッドが含まれるクラスの名前。

JConsole を使用してリモート JVM に接続する場合は、適切なホスト名と JMX ポートを使用します。

他の多くのツールおよびモニタリング製品は、JMX を使用してメトリクスを取得し、それらのメトリクスに基づいてモニタリングおよびアラートを提供するために使用できます。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。

7.5. 重要な KAFKA ブローカーメトリクス

Kafka では、Kafka クラスターのブローカーのパフォーマンスを監視するための MBean が多数提供されます。これらは、クラスター全体ではなく、個別のブローカーに適用されます。

以下の表は、サーバー、ネットワーク、ロギング、およびコントローラーメトリクスに整理されたこれらのブローカーレベルの MBean の選択を示しています。

7.5.1. Kafka サーバーメトリクス

以下の表は、Kafka サーバーの情報をレポートするメトリクスの選択を示しています。

表7.2 Kafka サーバーのメトリクス

メトリクス	MBean	説明	予想される値
1秒あたりのメッセージ	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec	ブローカーによって個別のメッセージが消費されるレート。	クラスター内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。	クラスター内の他のブローカーとほぼ同じです。
レプリケーションバイト (1秒あたりのレプリケーションバイト数)	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec	他のブローカーから送信されるデータがフォロワーブローカーによって消費されるレート。	該当せず
1秒あたりのバイト数	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec	データがコンシューマーによってブローカーから取得および読み取りされるレート。	該当せず
1秒あたりのレプリケーションバイト数	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec	データがブローカーから別のブローカーに送信されるレート。このメトリクスは、パーティショングループのブローカーがリーダーかどうかの監視に役立ちます。	該当せず
複製の数が最低数未満のパーティション	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	フォロワーレプリカで完全に複製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	ISR(In-Sync Replica)最小のパーティション数。ISR 数は、リーダーで最新の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べると、約が必要です。

メトリクス	MBean	説明	予想される値
リーダー数	kafka.server:type=ReplicaManager,name=LeaderCount	このブローカーがリーダーであるレプリカ数。	クラスター内の他のブローカーとほぼ同じです。
ISR 縮小/秒	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	ブローカーの ISR の数が減少する速度	ゼロ
ISR の 1 秒あたりの拡張	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	ブローカーの ISR の数が増加するレート。	ゼロ
最大ラグ	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	リーダーレプリカからメッセージを受信し、フォロワーレプリカによってメッセージを受信されるまでの最大ラグ。	生成要求の最大バッチサイズと比例します。
プロデューサーのペリメーションにおけるリクエスト	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Produce	プロデューサーPurgatoryの送信要求の数。	該当せず
フェッチ法のリクエスト	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Fetch	フェッチ法のフェッチリクエストの数。	該当せず
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。	値が小さいと、ブローカーのワークロードが高いことを示します。
要求（要求（スロットリングの超過）	kafka.server:type=Request	スロットリングから除外される要求の数。	該当せず
ZooKeeper 要求レイテンシー（ミリ秒単位）	kafka.server:type=ZooKeeperClientMetrics,name=ZooKeeperRequestLatencyMs	ブローカーからの ZooKeeper リクエストのレイテンシー（ミリ秒単位）。	該当せず
ZooKeeper セッションの状態	kafka.server:type=SessionExpireListener,name=SessionState	ZooKeeper への接続のステータス。	接続済み

7.5.2. Kafka ネットワークメトリクス

以下の表は、要求に関する情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	予想される値
1秒あたりのリクエスト	kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce FetchConsumer FetchFollower}	要求タイプ1秒あたりのリクエストの合計数。 Produce 、 FetchConsumer 、および FetchFollower リクエストタイプにはそれぞれ独自の MBean があります。	該当せず
リクエストバイト（要求サイズ（バイト単位）	kafka.network:type=RequestMetrics,name=RequestBytes,request={[-.lw]+}	MBean 名の request プロパティによって識別されるリクエストタイプに対して行われるリクエストのサイズ（バイト単位）。利用可能なリクエストタイプに対する個別の MBean は、 RequestBytes ノードの下に一覧表示されます。	該当せず
一時メモリーサイズ（バイト単位）	kafka.network:type=RequestMetrics,name=TemporaryMemoryBytes,request={Produce Fetch}	メッセージ形式の変換およびメッセージの圧縮解除に使用される一時的なメモリー量。	該当せず
メッセージ変換時間	kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce Fetch}	メッセージ形式の変換に費やされた時間（ミリ秒単位）。	該当せず
リクエスト時間の合計（ミリ秒単位）	kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce FetchConsumer FetchFollower}	リクエストの処理に費やされた合計時間（ミリ秒単位）。	該当せず
要求キュー時間（ミリ秒単位）	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce FetchConsumer FetchFollower}	リクエストが現在 request プロパティに指定されたリクエストタイプのキューに費やす時間（ミリ秒単位）。	該当せず

メトリクス	MBean	説明	予想される値
ローカル時間 (leader ローカル処理時間) (ミリ秒単位)	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce FetchConsumer FetchFollower}	リーダーが要求を処理するのにかかる時間 (ミリ秒単位)。	該当せず
リモート時間 (leader リモート処理時間) (ミリ秒単位)	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce FetchConsumer FetchFollower}	リクエストがフォロワーを待つ時間 (ミリ秒単位)。利用可能なリクエストタイプに対する個別の MBean は、 RemoteTimeMs ノードの下に一覧表示されます。	該当せず
応答キュー時間 (ミリ秒単位)	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce FetchConsumer FetchFollower}	リクエストが応答キューで待機する期間 (ミリ秒単位)。	該当せず
応答送信時間 (ミリ秒単位)	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)。	該当せず
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のパーセンテージ。	ゼロと1の間。

7.5.3. Kafka ログメトリクス

以下の表は、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	予想される値
Log flush rate and time (ミリ秒単位)	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs	ログデータがディスクに書き込まれる速度 (ミリ秒単位)。	該当せず

メトリクス	MBean	説明	予想される値
オフラインログディレクトリーの数	kafka.log:type=LogManager,name=OfflineLogDirectoryCount	オフラインログディレクトリーの数（ハードウェア障害などの場合）。	ゼロ

7.5.4. Kafka コントローラーメトリクス

以下の表は、クラスターのコントローラーに関する情報を報告するメトリクスの選択を示しています。

メトリクス	MBean	説明	予想される値
アクティブなコントローラー数	kafka.controller:type=KafkaController,name=ActiveControllerCount	コントローラーとして指定されるブローカーの数。	1つ目は、ブローカーがクラスターのコントローラーであることを示します。
リーダーの選択レートおよび時間（ミリ秒単位）	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	新しいリーダーレプリカが選出される速度。	ゼロ

7.5.5. Yammer メトリクス

時間のレートまたは単位を表すメトリクスは、Yammer メトリクスとして提供されます。Yammer メトリクスを使用する MBean のクラス名の前に **com.yammer.metrics** が付けられます。

Yammer レートメトリクスには、要求を監視するための以下の属性があります。

- Count
- EventType（バイト単位）
- FifteenMinuteRate
- RateUnit（秒数）
- MeanRate
- OneMinuteRate
- FiveMinuteRate

Yammer 時間メトリクスには、要求を監視するための以下の属性があります。

- Max
- Min
- Mean
- StdDev

- 75/95/98/99/99.9th Percentile

7.6. プロデューサー MBEAN

以下の MBean は Kafka Streams アプリケーションや、ソースコネクターを持つ Kafka Connect を含む Kafka プロデューサーアプリケーションに存在します。

7.6.1. 一致する MBean `kafka.producer:type=producer-metrics,client-id=*`

これらはプロデューサーレベルでのメトリクスです。

属性	説明
batch-size-avg	リクエストごとのパーティションごとに送信される平均バイト数。
batch-size-max	リクエストごとにパーティションごとに送信される最大バイト数。
batch-split-rate	1秒あたりのバッチの分割の平均数
batch-split-total	バッチ分割の合計数。
buffer-available-bytes	使用されていないバッファメモリの合計量（未割り当てまたは空きリスト内）。
buffer-total-bytes	クライアントが使用可能なバッファメモリの最大量（現在使用中かどうか）。
bufferpool-wait-time	アペンダーが領域の割り当てを待つ時間（分数）。
compression-rate-avg	圧縮されていないサイズの圧縮サイズにおける圧縮サイズの平均比率として定義される、レコードバッチの平均圧縮レート。
connection-close-rate	ウィンドウ内の1秒あたりの接続が閉じられます。
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秒あたりに確立された新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットから読み取るバイト/秒単位。
io-ratio	I/O スレッドが費やした I/O スレッドの一部。

属性	説明
io-time-ns-avg	選択呼び出しごとの I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	ソケットの読み取りと書き込みの待機に費やされた I/O スレッドの平均長（ナノ秒単位）。
metadata-age	使用されている現在のプロデューサーメタデータの経過時間（秒単位）。
network-io-rate	1秒あたりの全接続におけるネットワーク操作の平均数（読み取りまたは書き込み）。
outgoing-byte-rate	全サーバーに1秒あたりの送信バイト数の平均。
produce-throttle-time-avg	リクエストがブローカーによってスロットリングされた平均時間（ミリ秒単位）。
produce-throttle-time-max	リクエストがブローカーによってスロットリングされた最大時間（ミリ秒単位）。
record-error-rate	エラーの原因となったレコードの平均1秒あたりのレコード送信数。
record-error-total	エラーの原因となったレコードの合計数。
record-queue-time-avg	送信バッファに費やされた ms レコードバッチの平均時間。
record-queue-time-max	送信バッファに費やされた ms レコードバッチの最大時間。
record-retry-rate	再試行レコードの1秒あたりの平均送信数。
record-retry-total	再試行レコードの送信の合計数。
record-send-rate	1秒あたり送信される平均レコード数。
record-send-total	送信されるレコードの合計数。
record-size-avg	平均レコードのサイズ。
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

属性	説明
request-latency-avg	平均要求のレイテンシー（ミリ秒単位）。
request-latency-max	要求の最大レイテンシー（ミリ秒単位）。
request-rate	1秒あたり送信される要求の平均数。
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	ウィンドウ内で送信されたリクエストの最大サイズ。
requests-in-flight	応答の待機中、現在のインフライトリクエストの数。
response-rate	1秒あたりの受信された応答。
select-rate	I/O レイヤーが1秒あたりの新しい I/O を確認する回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続。
waiting-threads	バッファメモリーがレコードをキューに入れるまで待機するユーザースレッドの数。

7.6.2. 一致する MBean `kafka.producer:type=producer-metrics,client-id=*,node-id=*`

これらは、各ブローカーへの接続に関するプロデューサーレベルでメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される平均の応答数。
outgoing-byte-rate	ノードに対して1秒あたり送信される送信バイト数の平均。
request-latency-avg	ノードの平均要求のレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大要求レイテンシー（ミリ秒単位）。
request-rate	ノードに対して1秒あたりに送信される要求の平均数。
request-size-avg	ノードのウィンドウにあるすべての要求の平均サイズ。

属性	説明
request-size-max	ノードのウィンドウで送信される要求の最大サイズ。
response-rate	ノードに対して1秒あたりの受信された応答。

7.6.3. 一致する MBean `kafka.producer:type=producer-topic-metrics,client-id=*,topic=*`

これらは、プロデューサーがメッセージを送信するトピックに関するトピックレベルのメトリクスです。

属性	説明
byte-rate	トピックに対して1秒あたり送信される平均バイト数。
byte-total	トピックに送信されたバイト数。
compression-rate	圧縮されていないサイズの圧縮サイズにおける圧縮サイズの平均比率として、トピックのレコードバッチバッチの平均比率。
record-error-rate	トピックに対してエラーが生じたレコードごとの平均が1秒あたりの送信数の平均。
record-error-total	トピックのエラーとなったレコードの送信合計数。
record-retry-rate	トピックに送信する再試行レコードの平均回数。
record-retry-total	トピックに対して再試行レコードが送信する再試行レコードの合計数。
record-send-rate	トピックに対して1秒あたりに送信される平均レコード数。
record-send-total	トピックに送信されたレコードの合計数。

7.7. コンシューマー MBEAN

以下の MBean は Kafka Streams アプリケーションや、シンクコネクタを持つ Kafka Connect を含む Kafka コンシューマーアプリケーションに存在します。

7.7.1. 一致する MBean `kafka.consumer:type=consumer-metrics,client-id=*`

これらはコンシューマーレベルでのメトリクスです。

属性	説明
connection-close-rate	ウィンドウ内の1秒あたりの接続が閉じられます。
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秒あたりに確立された新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットから読み取るバイト/秒単位。
io-ratio	I/O スレッドが費やした I/O スレッドの一部。
io-time-ns-avg	選択呼び出しごとの I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	ソケットの読み取りと書き込みの待機に費やされた I/O スレッドの平均長（ナノ秒単位）。
network-io-rate	1秒あたりの全接続におけるネットワーク操作の平均数（読み取りまたは書き込み）。
outgoing-byte-rate	全サーバーに1秒あたりの送信バイト数の平均。
request-rate	1秒あたり送信される要求の平均数。
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	ウィンドウ内で送信されたリクエストの最大サイズ。
response-rate	1秒あたりの受信された応答。
select-rate	I/O レイヤーが1秒あたりの新しい I/O を確認する回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続。

7.7.2. 一致する MBean `kafka.consumer:type=consumer-metrics,client-id=*,node-id=*`

これらは、各ブローカーへの接続に関するコンシューマーレベルでのメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される平均の応答数。
outgoing-byte-rate	ノードに対して1秒あたり送信される送信バイト数の平均。
request-latency-avg	ノードの平均要求のレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大要求レイテンシー（ミリ秒単位）。
request-rate	ノードに対して1秒あたりに送信される要求の平均数。
request-size-avg	ノードのウィンドウにあるすべての要求の平均サイズ。
request-size-max	ノードのウィンドウで送信される要求の最大サイズ。
response-rate	ノードに対して1秒あたりの受信された応答。

7.7.3. 一致する MBean `kafka.consumer:type=consumer-coordinator-metrics,client-id=*`

これらは、コンシューマーグループに関するコンシューマーレベルでのメトリクスです。

属性	説明
assigned-partitions	このコンシューマーに現在割り当てられているパーティションの数。
commit-latency-avg	コミットリクエストにかかった平均時間。
commit-latency-max	コミットリクエストにかかった最大時間。
commit-rate	1秒あたりのコミット呼び出しの数。
heartbeat-rate	1秒あたりのハートビートの平均数。
heartbeat-response-time-max	ハートビートリクエストへの応答を受信するためにかかった最大時間。
join-rate	1秒あたりに参加するグループ数。
join-time-avg	グループ再参加にかかった平均時間。
join-time-max	グループ再参加にかかった最大時間。

属性	説明
last-heartbeat-seconds-ago	最後のコントローラーのハートビートからの秒数。
sync-rate	1秒あたりのグループ同期数。
sync-time-avg	グループ同期の平均時間。
sync-time-max	グループ同期にかかった最大時間。

7.7.4. 一致する MBean `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*`

これらは、コンシューマーのフェッチ元に関するコンシューマーレベルでのメトリクスです。

属性	説明
bytes-consumed-rate	1秒あたり消費される平均バイト数。
bytes-consumed-total	消費されるバイト数。
fetch-latency-avg	フェッチリクエストにかかった平均時間。
fetch-latency-max	フェッチリクエストにかかった最大時間。
fetch-rate	1秒あたりのフェッチリクエストの数。
fetch-size-avg	リクエストごとにフェッチされる平均バイト数。
fetch-size-max	リクエストごとにフェッチされる最大バイト数。
fetch-throttle-time-avg	平均スロットル時間（ミリ秒単位）。
fetch-throttle-time-max	最大スロットル時間（ミリ秒単位）。
fetch-total	フェッチリクエストの合計数。
records-consumed-rate	1秒あたり消費される平均レコード数。
records-consumed-total	消費されるレコードの合計数。
records-lag-max	このウィンドウのパーティションに対するレコード数の最大ラグ。
records-lead-min	このウィンドウのパーティションに対するレコード数の最小リード。

属性	説明
records-per-request-avg	各リクエストの平均レコード数。

7.7.5. 一致する MBean `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*`

これらは、コンシューマーのフェッチ元に関するトピックレベルでのメトリクスです。

属性	説明
bytes-consumed-rate	トピックに対して、1秒あたり消費される平均バイト数。
bytes-consumed-total	トピックに対して消費される合計バイト数。
fetch-size-avg	トピックのリクエストごとにフェッチされる平均バイト数。
fetch-size-max	トピックのリクエストごとにフェッチされる最大バイト数。
records-consumed-rate	トピックに対して1秒あたり消費される平均レコード数。
records-consumed-total	トピックに使用されるレコードの合計数。
records-per-request-avg	トピックの各リクエストの平均レコード数。

7.7.6. 一致する MBean `kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*,topic=*,partition=*`

これらは、コンシューマーのフェッチ元に関するパーティションレベルでのメトリクスです。

属性	説明
preferred-read-replica	パーティションに対する現在の読み取りレプリカ。リーダーから読み込む場合は -1。
record-lag	パーティションの最新ラグ。
records-lag-avg	パーティションの平均ラグ。
records-lag-max	パーティションの最大ラグ。
record-lead	パーティションが最新リードします。

属性	説明
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。

7.8. KAFKA CONNECT MBEANS



注記

Kafka Connect には、ドキュメント化されたコネクターに加えて、シンクコネクターのプロデューサーの MBean と、シンクコネクターのコンシューマー MBean が含まれます。

7.8.1. 一致する MBean `kafka.connect:type=connect-metrics,client-id=*`

これらは接続レベルでのメトリクスです。

属性	説明
connection-close-rate	ウィンドウ内の1秒あたりの接続が閉じられます。
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秒あたりに確立された新しい接続。
failed-authentication-rate	認証に失敗した接続。
incoming-byte-rate	すべてのソケットから読み取るバイト/秒単位。
io-ratio	I/O スレッドが費やした I/O スレッドの一部。
io-time-ns-avg	選択呼び出しごとの I/O の平均時間（ナノ秒単位）。
io-wait-ratio	I/O スレッドが待機に費やした時間の割合。
io-wait-time-ns-avg	ソケットの読み取りと書き込みの待機に費やされた I/O スレッドの平均長（ナノ秒単位）。
network-io-rate	1秒あたりの全接続におけるネットワーク操作の平均数（読み取りまたは書き込み）。
outgoing-byte-rate	全サーバーに1秒あたりの送信バイト数の平均。
request-rate	1秒あたり送信される要求の平均数。

属性	説明
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	ウィンドウ内で送信されたリクエストの最大サイズ。
response-rate	1秒あたりの受信された応答。
select-rate	I/O レイヤーが1秒あたりの新しい I/O を確認する回数。
successful-authentication-rate	SASL または SSL を使用して正常に認証された接続。

7.8.2. 一致する MBean `kafka.connect:type=connect-metrics,client-id=*,node-id=*`

これらは、各ブローカーへの接続に関する接続レベルでメトリクスです。

属性	説明
incoming-byte-rate	ノードに対して1秒あたり受信される平均の応答数。
outgoing-byte-rate	ノードに対して1秒あたり送信される送信バイト数の平均。
request-latency-avg	ノードの平均要求のレイテンシー（ミリ秒単位）。
request-latency-max	ノードの最大要求レイテンシー（ミリ秒単位）。
request-rate	ノードに対して1秒あたりに送信される要求の平均数。
request-size-avg	ノードのウィンドウにあるすべての要求の平均サイズ。
request-size-max	ノードのウィンドウで送信される要求の最大サイズ。
response-rate	ノードに対して1秒あたりの受信された応答。

7.8.3. 一致する MBean `kafka.connect:type=connect-worker-metrics`

これらは接続レベルでのメトリクスです。

属性	説明
connector-count	このワーカーで実行されるコネクタの数。
connector-startup-attempts-total	このワーカー試行したコネクタの起動の合計数。
connector-startup-failure-percentage	このワーカーのコネクタの平均のパーセンテージは失敗して開始します。
connector-startup-failure-total	失敗したコネクタの合計数。
connector-startup-success-percentage	このワーカーのコネクタの平均のパーセンテージは成功します。
connector-startup-success-total	成功したコネクタの合計数。
task-count	このワーカーで実行されるタスクの数。
task-startup-attempts-total	このワーカーが試行したタスク起動の合計数。
task-startup-failure-percentage	このワーカーのタスクの平均のパーセンテージは失敗して開始します。
task-startup-failure-total	失敗したタスクの合計数。
task-startup-success-percentage	このワーカーのタスクの平均のパーセンテージは成功します。
task-startup-success-total	成功したタスクの合計数。

7.8.4. 一致する MBean `kafka.connect:type=connect-worker-rebalance-metrics`

属性	説明
completed-rebalances-total	このワーカーによって完了したリバランスの合計数。
connect-protocol	このクラスターで使用される Connect プロトコル。
Epoch	このワーカーのエポックまたは生成番号。
leader-name	グループリーダーの名前。
rebalance-avg-time-ms	このワーカーがリバランスを行うまで費やす平均時間（ミリ秒単位）。

属性	説明
rebalance-max-time-ms	このワーカーがリバランスを行うまで費やす最大時間（ミリ秒単位）。
リバランス	このワーカーが現時点でリバランスされているかどうか。
time-since-last-rebalance-ms	このワーカーが最新のリバランスを完了してからの時間（ミリ秒単位）。

7.8.5. 一致する MBean `kafka.connect:type=connector-metrics,connector=*`

属性	説明
connector-class	コネクタークラスの名前。
connector-type	コネクターのタイプ。'source' または 'sink' のいずれか。
connector-version	コネクターによって報告されるコネクタークラスのバージョン。
status	コネクターのステータス。「unassigned」、「running」、「paused」、「failed」、または「destroyed」のいずれかです。

7.8.6. 一致する MBean `kafka.connect:type=connector-task-metrics,connector=*,task=*`

属性	説明
batch-size-avg	コネクターによって処理されるバッチの平均サイズ。
batch-size-max	コネクターによって処理されるバッチの最大サイズ。
offset-commit-avg-time-ms	このタスクがオフセットをコミットする平均時間（ミリ秒単位）。
offset-commit-failure-percentage	このタスクのオフセットコミットの平均率（失敗した場合）。
offset-commit-max-time-ms	このタスクがオフセットをコミットする最大時間（ミリ秒単位）。

属性	説明
offset-commit-success-percentage	このタスクのオフセットコミットの平均のパーセンテージ。
pause-ratio	このタスクが pause 状態で費やした時間。
running-ratio	このタスクが running 状態で費やした時間。
status	コネクタタスクの状態。「unassigned」、「running」、「paused」、「failed」、または「destroyed」のいずれかです。

7.8.7. 一致する MBean `kafka.connect:type=sink-task-metrics,connector=*,task=*`

属性	説明
offset-commit-completion-rate	オフセットコミットの完了の秒単位の平均。
offset-commit-completion-total	オフセットコミットの完了の合計数。
offset-commit-seq-no	オフセットコミットの現在のシーケンス番号。
offset-commit-skip-rate	受信されたオフセットコミットの完了の秒ごとの平均およびスキップ/無視。
offset-commit-skip-total	受け取ったオフセットコミットの完了の合計数。短縮/無視します。
partition-count	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割り当てられるトピックパーティションの数。
put-batch-avg-time-ms	シンクの記録のバッチを配置するためにこのタスクによる平均時間。
put-batch-max-time-ms	シンクの記録のバッチを配置するこのタスクによる最大時間。
sink-record-active-count	Kafka から読み取られたが、シンクタスクによって完全にコミット/フラッシュ/承認されていないレコードの数。
sink-record-active-count-avg	Kafka から読み取られたが、シンクタスクによって完全にコミット/フラッシュ/確認されていないレコードの平均数。

属性	説明
sink-record-active-count-max	Kafka から読み取られたが、シンクタスクによって完全にコミット/フラッシュ/承認されていないレコードの最大数。
sink-record-lag-max	シンクタスクがトピックパーティションのコンシューマーの位置の背後にあるレコード数の最大ラグ。
sink-record-read-rate	このワーカーで名前付きシンクコネクタに属するこのタスクの Kafka から読み取るレコードの平均（1秒あたりのレコード数）。これは、変換を適用する前に行われます。
sink-record-read-total	タスクが最後に再起動されたため、このワーカーの名前付きシンクコネクタに属するこのタスクによって Kafka から読み取られるレコードの合計数。
sink-record-send-rate	このワーカーで名前付きシンクコネクタに属するこのタスクに、変換および送信/スループットからのレコード出力の平均/スループット。変換の適用後は、変換でフィルターされたレコードはすべて除外されます。
sink-record-send-total	タスクが最後に再起動されたため、このワーカーの名前付きシンクコネクタに属するこのタスクに、変換および送信/スループットからのレコード出力の合計数。

7.8.8. 一致する MBean `kafka.connect:type=source-task-metrics,connector=*,task=*`

属性	説明
poll-batch-avg-time-ms	ソースレコードのバッチをポーリングするこのタスクの平均時間（ミリ秒単位）。
poll-batch-max-time-ms	ソースレコードのバッチをポーリングするこのタスクによる最大時間（ミリ秒単位）。
source-record-active-count	このタスクによって生成されたが Kafka に完全に書き込まれていないレコードの数。
source-record-active-count-avg	このタスクによって生成されたが、Kafka に完全に書き込まれていない平均レコード数。

属性	説明
source-record-active-count-max	このタスクによって生成されたが Kafka に完全に書き込まれていない最大レコード数。
source-record-poll-rate	このワーカーで名前付きソースコネクタに属するこのタスクにより、生成/ポーリング（変換前）の平均レコードの1秒あたりの平均。
source-record-poll-total	このワーカーで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/ポーリングされたレコードの合計数（変換前）。
source-record-write-rate	変換からのレコード出力の平均/秒あたりのレコード出力数。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 に書き込まれます。変換の適用後には、変換でフィルターされたレコードはすべて除外されます。
source-record-write-total	タスクが最後に再起動されたため、このワーカーの名前付きソースコネクタに属するこのタスクの変換および Kafka へのレコード出力数。

7.8.9. 一致する MBean `kafka.connect:type=task-error-metrics,connector=*,task=*`

属性	説明
deadletterqueue-produce-failures	デッドレターキューへの書き込みに失敗した回数。
deadletterqueue-produce-requests	デッドレターキューに書き込みを試行する回数。
last-error-timestamp	このタスクがエラーに最後に発生した場合のエポックのタイムスタンプ。
total-errors-logged	ログに記録されたエラーの数。
total-record-errors	このタスクのレコード処理エラーの数。
total-record-failures	このタスクでのレコード処理の失敗回数。
total-records-skipped	エラーによりスキップされたレコード数。
total-retries	再試行操作の数。

7.9. KAFKA STREAMS MBEANS



注記

Streams アプリケーションには、ここに記載されているものに加えて、プロデューサーとコンシューマーの **MBean** が含まれます。

7.9.1. 一致する MBean `kafka.streams:type=stream-metrics,client-id=*`

これらのメトリクスは、**metrics.recording.level** 設定パラメーターが **info** または **debug** の場合に収集されます。

属性	説明
commit-latency-avg	このスレッドで実行中の全タスクで、コミットする平均実行時間（ミリ秒単位）。
commit-latency-max	このスレッドで実行中の全タスクにコミットする最大時間（ミリ秒単位）。
commit-rate	1秒あたりのコミットの平均数。
commit-total	すべてのタスクにわたるコミット呼び出しの合計数。
poll-latency-avg	このスレッドの全実行中の全タスクで、ポーリングの平均実行時間（ミリ秒単位）。
poll-latency-max	このスレッドで実行中の全タスクに対して、ポーリングに対する最大時間（ミリ秒単位）。
poll-rate	1秒あたりのポーリングの平均数。
poll-total	すべてのタスクにわたるポーリング呼び出しの合計数。
process-latency-avg	このスレッドの全実行中の全タスクに、処理に関する平均実行時間（ミリ秒単位）。
process-latency-max	このスレッドで実行中の全タスクを処理する最大実行時間（ミリ秒単位）。
process-rate	1秒あたりのプロセス呼び出しの平均数。
process-total	全タスクにわたるプロセス呼び出しの合計数。
punctuate-latency-avg	このスレッドの全稼働中のタスクにわたる、句読点の平均実行時間（ミリ秒単位）。
punctuate-latency-max	このスレッドで実行中の全タスクに対して、句読点を区切る最大時間（ミリ秒単位）。

属性	説明
punctuate-rate	1秒あたりの平均句数。
punctuate-total	すべてのタスクで句読した呼び出しの合計数。
skipped-records-rate	1秒あたりのスキップレコードの平均数
skipped-records-total	スキップされたレコードの合計数。
task-closed-rate	1秒あたりに閉じられた平均タスク数
task-closed-total	終了したタスクの合計数。
task-created-rate	1秒あたりの新しく作成されたタスクの平均数
task-created-total	作成されたタスクの合計数

7.9.2. 一致する MBean `kafka.streams:type=stream-task-metrics,client-id=*,task-id=*`

タスクメトリクス。

これらのメトリクスは、`metrics.recording.level` 設定パラメーターが **debug** の場合に収集されます。

属性	説明
commit-latency-avg	このタスクの ns の平均コミット時間。
commit-latency-max	このタスクの ns の最大コミット時間。
commit-rate	1秒あたりのコミット呼び出しの平均数。
commit-total	コミット呼び出しの合計数。

7.9.3. 一致する MBean `kafka.streams:type=stream-processor-node-metrics,client-id=*,task-id=*,processor-node-id=*`

プロセッサノードメトリクス。

これらのメトリクスは、`metrics.recording.level` 設定パラメーターが **debug** の場合に収集されます。

属性	説明
create-latency-avg	ns の平均作成実行時間。
create-latency-max	ns にある最大作成実行時間。

属性	説明
create-rate	1秒あたりの create 操作の平均数
create-total	作成操作の合計数です。
destroy-latency-avg	ns の平均破棄実行時間。
destroy-latency-max	ns にある最大破棄実行時間。
destroy-rate	1秒あたりの破棄操作の平均数。
destroy-total	呼び出された破棄操作の合計数。
forward-rate	ダウストリームから転送されるレコードの平均レート（ソースノード毎秒のみ）。
forward-total	ダウストリームを転送するレコードの合計数（ソースノードのみ）。
process-latency-avg	ns の平均プロセス実行時間。
process-latency-max	ns にあるプロセスの最大実行時間。
process-rate	1秒あたりのプロセス操作の平均数。
process-total	呼び出されたプロセス操作の合計数。
punctuate-latency-avg	ns の平均句読点
punctuate-latency-max	ns の実行時間の上限です。
punctuate-rate	1秒あたりの句読点操作の平均数
punctuate-total	呼び出されるバイトされた操作の合計数。

7.9.4. 一致する MBean `kafka.streams:type=stream-[store-scope]-metrics,client-id=*,task-id=*,[store-scope]-id=*`

ステートストアメトリクス。

これらのメトリクスは、`metrics.recording.level` 設定パラメーターが **debug** の場合に収集されます。

属性	説明
all-latency-avg	ns のすべての操作実行時間の平均。

属性	説明
all-latency-max	ns のすべての操作実行時間。
all-rate	このストアのすべての操作レートの平均。
all-total	このストアに対するすべてのオペレーションコールの合計数。
delete-latency-avg	ns の平均削除実行時間。
delete-latency-max	ns にある最大削除実行時間。
delete-rate	このストアの平均削除レート。
delete-total	このストアに対する削除呼び出しの合計数。
flush-latency-avg	ns の平均フラッシュ実行時間。
flush-latency-max	ns にある最大フラッシュ実行時間。
flush-rate	このストアの平均フラッシュレート。
flush-total	このストアのフラッシュ呼び出しの合計数。
get-latency-avg	ns の平均 get execution time
get-latency-max	ns の最大 get 実行時間。
get-rate	このストアの平均取得レート。
get-total	このストアの取得コールの合計数。
put-all-latency-avg	ns の平均 put-all 実行時間。
put-all-latency-max	ns にある最大 put-all 実行時間。
put-all-rate	このストアの平均 put-all レート。
put-all-total	このストアに対する put-all 呼び出しの合計数。
put-if-absent-latency-avg	ns の平均 put-if-absent 実行時間。
put-if-absent-latency-max	ns にある最大 put-if-absent 実行時間。
put-if-absent-rate	このストアの平均 put-if-absent レート。

属性	説明
put-if-absent-total	このストアに対する put-if-absent 呼び出しの合計数。
put-latency-avg	平均が実行時間を ns にする。
put-latency-max	実行時間が ns に配置される最大時間。
put-rate	このストアの平均配置レート。
put-total	このストアに対する実行合計数。
range-latency-avg	ns の平均範囲実行時間。
range-latency-max	ns にある最大範囲実行時間。
range-rate	このストアの平均範囲レート。
range-total	このストアの範囲呼び出しの合計数。
restore-latency-avg	ns の平均復元時間。
restore-latency-max	ns にある最大復元実行時間。
restore-rate	このストアの平均復元レート。
restore-total	このストアの復元呼び出しの合計数。

7.9.5. 一致する MBean `kafka.streams:type=stream-record-cache-metrics,client-id=*,task-id=*,record-cache-id=*`

キャッシュメトリクスを記録します。

これらのメトリクスは、`metrics.recording.level` 設定パラメーターが **debug** の場合に収集されます。

属性	説明
hitRatio-avg	キャッシュ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の比率として定義される平均キャッシュヒット率。
hitRatio-max	キャッシュの最大ヒット比率。
hitRatio-min	最小キャッシュヒットの比率。

第8章 KAFKA CONNECT

Kafka Connect は、Apache Kafka と外部システムとの間でデータをストリーミングするためのツールです。スケーラビリティ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 クラスター外部のデータベース、ストレージ、メッセージングシステムと統合するために使用されます。

Kafka Connect は、異なるタイプの外部システムの接続を実装するコネクタプラグインを使用します。シンクおよびソースには、2 種類のコネクタプラグインがあります。シンクコネクタは、データを Kafka から外部システムに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取られたユーザー定義の設定を持つ単一ノードで実行されます。

分散モード

Distributed モードでは、Kafka Connect は1つまたは複数のワーカーノードで実行され、ワークロードはそれら間に分散されます。HTTP REST インターフェースを使用して、コネクタおよびその設定を管理します。

8.1. 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 は単一のノードで単一のプロセスとして実行されます。プロパティファイルを使用してスタンドアロンモードで設定を管理します。

8.1.1. スタンドアロンモードでの Kafka Connect の設定

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

bootstrap.servers

Kafka へのブートストラップ接続として使用される Kafka ブローカーアドレスのリスト。例：
kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092

key.converter

メッセージキーの Kafka 形式への変換に使用されるクラス。例：
org.apache.kafka.connect.json.JsonConverter

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。例：
org.apache.kafka.connect.json.JsonConverter

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にあります。サポートされる Kafka Connect 設定オプションの完全リストは、[kafka-connect-configuration-parameters-str] を参照してください。

コネクタプラグインは、ブートストラップアドレスを使用して Kafka ブローカーへのクライアント接続を開きます。これらの接続を設定するには、**producer.** または **consumer.** で始まる標準の Kafka プロデューサーおよびコンシューマー設定オプションを使用します。

Kafka プロデューサーおよびコンシューマーの設定に関する詳細は、以下を参照してください。

- [付録D プロデューサー設定パラメーター](#)
- [付録C コンシューマー設定パラメーター](#)

8.1.2. 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

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

connector.class

コネクタプラグインのクラス。例：**org.apache.kafka.connect.file.FileStreamSinkConnector**

tasks.max

指定のコネクタが使用できるタスクの最大数。タスクを使用すると、コネクタが並行して作業を実行できるようにします。コネクタによって指定の数よりも少ないタスクが作成される可能性があります。

key.converter

メッセージキーの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されたデフォルト値が上書きされます。例：

org.apache.kafka.connect.json.JsonConverter

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されたデフォルト値が上書きされます。例：

org.apache.kafka.connect.json.JsonConverter

さらに、シンクコネクタに以下のオプションのいずれかを設定する必要があります。

topics

入力として使用されるトピックのコンマ区切りリスト。

topics.regex

入力として使用するトピックの Java 正規表現。

その他のオプションは、使用できる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** を参照してください。

8.1.3. 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

- AMQ Streams クラスタがインストールされ、実行している。

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 ブローカーを示すように設定します。以下に例を示します。

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
```

2. 設定ファイルで 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. Kafka Connect が稼働していることを確認します。

```
jcmd | grep ConnectStandalone
```

その他のリソース

- AMQ Streams のインストールに関する詳細は、[「AMQ Streams のインストール」](#) を参照してください。
- AMQ Streams の設定に関する詳細は、[「AMQ Streams の設定」](#) を参照してください。
- サポートされる Kafka Connect 設定オプションの完全リストは、[付録F Kafka Connect 設定パラメーター](#)を参照してください。

8.2. 「KAFKA CONNECT IN DISTRIBUTED MODE」

Distributed モードでは、Kafka Connect は1つまたは複数のワーカーノードで実行され、ワークロードはそれら間に分散されます。HTTP REST インターフェースを使用して、コネクタプラグインおよびその設定を管理します。

8.2.1. 分散モードでの Kafka Connect の設定

分散モードで Kafka Connect を設定するには、**config/connect-distributed.properties** 設定ファイルを編集します。以下のオプションは最も重要なオプションです。

bootstrap.servers

Kafka へのブートストラップ接続として使用される Kafka ブローカーアドレスのリスト。例：
kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092

key.converter

メッセージキーの Kafka 形式への変換に使用されるクラス。例：
org.apache.kafka.connect.json.JsonConverter

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。例：
org.apache.kafka.connect.json.JsonConverter

group.id

分散 Kafka Connect クラスターの名前。これは一意で、別のコンシューマーグループ ID と競合しないようにしてください。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

オフセットの保存に使用される 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 の設定ファイルのサンプルが含まれています。AMQ Streams インストールディレクトリーの **config/connect-distributed.properties** を参照してください。

サポートされる Kafka Connect 設定オプションの完全リストは、[付録F Kafka Connect 設定パラメーター](#)を参照してください。

コネクタプラグインは、ブートストラップアドレスを使用して Kafka ブローカーへのクライアント接続を開きます。これらの接続を設定するには、**producer.** または **consumer.** で始まる標準の Kafka プロデューサーおよびコンシューマー設定オプションを使用します。

Kafka プロデューサーおよびコンシューマーの設定に関する詳細は、以下を参照してください。

- [付録D プロデューサー設定パラメーター](#)
- [付録C コンシューマー設定パラメーター](#)

8.2.2. 分散 Kafka Connect でのコネクタの設定

HTTP REST インターフェース

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

GET /connectors

既存のコネクタ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を持つ JSON オブジェクトである必要があります。

GET /connectors/<name>

特定のコネクタに関する情報を取得します。

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。コネクタはすべての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

コネクタが失敗したら、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

サポートされるコネクタプラグインをすべて取得します。

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です。

name

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

connector.class

コネクタプラグインのクラス。たとえ

ば、**org.apache.kafka.connect.file.FileStreamSinkConnector** のようになります。

tasks.max

このコネクタによって使用されるタスクの最大数。タスクは、作業を並行して行うためにコネクタによって使用されます。Connectors は、指定した数よりも少ないタスクを作成できます。

key.converter

メッセージキーの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されたデフォルト値が上書きされます。例：

org.apache.kafka.connect.json.JsonConverter

value.converter

メッセージペイロードの Kafka 形式への変換に使用されるクラス。これにより、Kafka Connect 設定によって設定されたデフォルト値が上書きされます。例：

org.apache.kafka.connect.json.JsonConverter

さらに、シンクコネクタに以下のオプションのいずれかを設定する必要があります。

topics

入力として使用されるトピックのコンマ区切りリスト。

topics.regex

入力として使用するトピックの Java 正規表現。

その他のオプションは、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 のインストールディレクトリでは、**config/connect-file-sink.properties** および **config/connect-file-source.properties** を参照してください。

8.2.3. 分散 Kafka Connect の実行

この手順では、分散モードで Kafka Connect を設定し、実行する方法を説明します。

前提条件

- AMQ Streams クラスターがインストールされ、実行している。

この手順は、次のように実行します。

ツプスターの実行

1. すべての Kafka Connect ワーカーノードで **/opt/kafka/config/connect-distributed.properties** Kafka Connect 設定ファイルを編集します。

- **bootstrap.server** オプションを設定して、Kafka ブローカーを指定します。
- **group.id** オプションを設定します。
- **config.storage.topic** オプションを設定します。
- **offset.storage.topic** オプションを設定します。
- **status.storage.topic** オプションを設定します。
以下に例を示します。

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
group.id=my-group-id
config.storage.topic=my-group-id-configs
offset.storage.topic=my-group-id-offsets
status.storage.topic=my-group-id-status
```

2. すべての Kafka Connect ノードの **/opt/kafka/config/connect-distributed.properties** 設定ファイルを使用して Kafka Connect ワーカーを起動します。

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

3. Kafka Connect が稼働していることを確認します。

```
jcmd | grep ConnectDistributed
```

その他のリソース

- AMQ Streams のインストールに関する詳細は、[「AMQ Streams のインストール」](#) を参照してください。
- AMQ Streams の設定に関する詳細は、[「AMQ Streams の設定」](#) を参照してください。
- サポートされる Kafka Connect 設定オプションの完全リストは、[付録F Kafka Connect 設定パラメーター](#) を参照してください。

8.2.4. 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 と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

- 分散モードで実行された 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、コネクターを作成します。以下の例では、**curl** を使用しています。

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

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

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

8.2.5. コネクターの削除

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

前提条件

- 分散モードで実行された Kafka Connect インストール。

コネクターの削除

1. GET リクエストを **<KafkaConnectAddress>:8083/connectors/<ConnectorName>** に送信し、コネクターが存在することを確認します。以下の例では、**curl** を使用しています。

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

2. コネクターを削除するには、**DELETE** リクエストを **<KafkaConnectAddress>:8083/connectors** に送信します。以下の例では、**curl** を使用しています。

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

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

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

8.3. コネクタープラグイン

AMQ Streams には、以下のコネクター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込み、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込み、データを Kafka トピックに送信します。

必要に応じて、コネクタ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、パスのコンマ区切りリストを含めることができます。

分散モードで Kafka Connect を実行する場合、プラグインはすべてのワーカーノードで利用可能である必要があります。

8.4. 「ADDING CONNECTOR PLUGINS」

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

- AMQ Streams クラスターがインストールされ、実行している。

手順

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

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

2. **/opt/kafka/config/connect-standalone.properties** または **/opt/kafka/config/connect-distributed.properties** 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。

その他のリソース

- AMQ Streams のインストールに関する詳細は、[「AMQ Streams のインストール」](#) を参照してください。
- AMQ Streams の設定に関する詳細は、[「AMQ Streams の設定」](#) を参照してください。
- スタンドアロンモードで Kafka Connect の実行に関する詳細は、[「スタンドアロンモードでの Kafka Connect の実行」](#) を参照してください。
- 分散モードでの Kafka Connect の実行に関する詳細は、[「分散 Kafka Connect の実行」](#) を参照してください。

- サポートされる Kafka Connect 設定オプションの完全リストは、[付録F Kafka Connect 設定パラメーター](#)を参照してください。

第9章 AMQ STREAMS の MIRRORMAKER 2.0 との使用

MirrorMaker 2.0 は、データセンター内またはデータセンター全体の 2 台以上の Kafka クラスター間でデータを複製するために使用されます。

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

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



注記

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

その他のリソース

- [Apache Kafka ドキュメント](#)
- [「MirrorMaker 2.0 のトレースの有効化」](#)

9.1. 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。

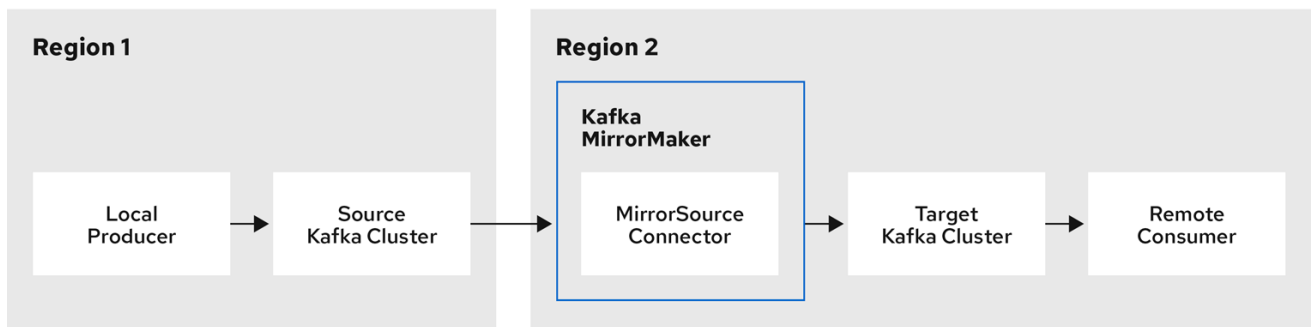
- ソースクラスターからデータを消費するソースクラスターの設定。
- データをターゲットクラスターに出力するターゲットクラスターの設定。

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 の **MirrorSourceConnector** は、ソースクラスターからターゲットクラスターにトピックを複製します。

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

MirrorMaker 2.0 は、複数のソースクラスターで使用できます。

図9.12 つのクラスターにおけるレプリケーション



AMQ_73_0220

デフォルトでは、ソースクラスターの新規トピックのチェックは10分ごとに行われます。**refresh.topics.interval.seconds**をソースコネクター設定に追加することで、頻度を変更できます。ただし、操作の頻度が増えると、全体的なパフォーマンスに影響する可能性があります。

9.2. クラスターの設定

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

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

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

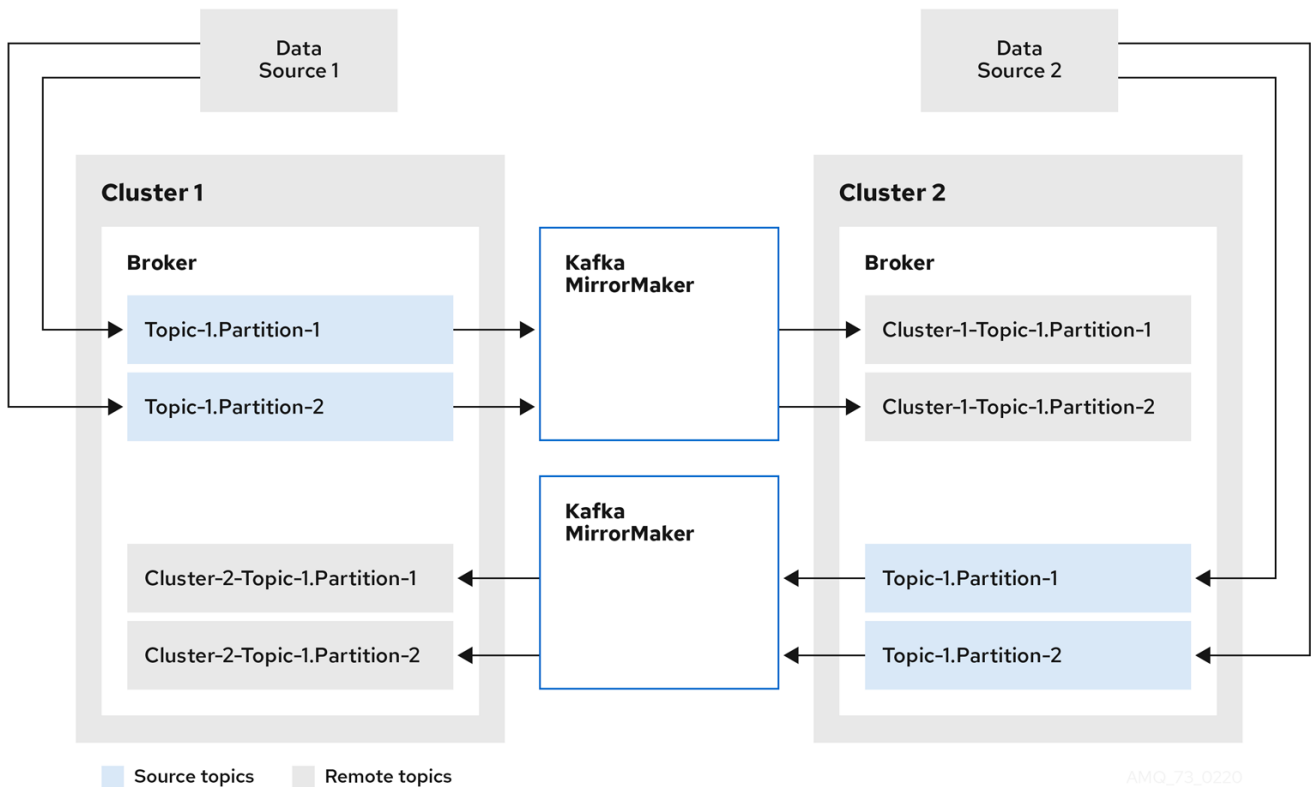
MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

9.2.1. 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9.2 トピックの名前変更



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

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

9.2.2. 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。

9.2.3. 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。

9.2.4. 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。

9.2.5. オフセットの追跡

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

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

チェックポイント トピックのオフセットは、設定によって事前定義された間隔で追跡されます。両方のトピックは、フェイルオーバー時に正しいオフセットの位置からレプリケーションの完全復元を可能にします。

MirrorMaker 2.0 は、**MirrorCheckpointConnector** を使用して、オフセット追跡の **チェックポイント** を生成します。

9.2.6. コンシューマーグループオフセットの同期

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

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

トピックオフセットの同期を使用するには、以下を行います。

- **sync.group.offsets.enabled** をチェックポイントコネクタ設定に追加し、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。
- **IdentityReplicationPolicy** をソースおよびチェックポイントコネクタ設定に追加し、ターゲットクラスターのトピックが元の名前を保持するようにします。

トピックオフセットの同期を機能させるため、ターゲットクラスターのコンシューマーグループは、ソースクラスターのグループと同じ ID を使用できません。

同期を有効にすると、ソースクラスターからオフセットの同期が定期的に行われま

す。**sync.group.offsets.interval.seconds** および **emit.checkpoints.interval.seconds** をチェックポイントコネクタ設定に追加すると、頻度を変更できます。これらのプロパティは、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をチェックする頻度を変更することもできます。デフォルトでは 10 分ごとに実行されます。

同期は時間ベースであるため、コンシューマーによってパッシブクラスターへ切り替えられると、一部のメッセージが重複する可能性があります。

9.2.7. 接続性チェック

ハートビート 内部トピックによって、クラスター間の接続性が確認されます。

ハートビート トピックは、ソースクラスターから複製されます。

ターゲットクラスターは、トピックを使用して以下を確認します。

- クラスター間の接続を管理するコネクタが稼働している。
- ソースクラスターが利用可能である。

MirrorMaker 2.0 は **MirrorHeartbeatConnector** を使用して、これらのチェックを実行する ハートビート を生成します。

9.3. ACL ルールの同期

AclAuthorizer が使用されている場合、ブローカーへのアクセスを管理する ACL ルールはリモートトピックにも適用されます。ソーストピックを読み取りできるユーザーは、そのリモートトピックを読み取ることができます。



注記

OAuth 2.0 での承認は、このようなりモートトピックへのアクセスをサポートしません。

9.4. MIRRORMAKER 2.0 を使用した KAFKA クラスター間でのデータの同期

MirrorMaker 2.0 を使用して、設定を介して Kafka クラスター間のデータを同期します。

従来のモードで [MirrorMaker 2.0](#) を実行すると、以前のバージョンの [MirrorMaker](#) は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に設定を作成し、MirrorMaker スクリプトファイルを使用して接続を設定するときにプロパティを渡すことで、MirrorMaker 2.0 を実装する方法を説明します。



注記

MirrorMaker 2.0 は Kafka Connect を使用して接続を行い、クラスター間のデータを転送します。Kafka は、データレプリケーションに MirrorMaker シンクおよびソースコネクタを提供します。MirrorMaker スクリプトの代わりにコネクタを使用する場合は、コネクタを Kafka Connect クラスターで設定する必要があります。詳細は [Apache Kafka のドキュメント](#) を参照してください。

作業を始める前に

設定例は `./config/connect-mirror-maker.properties` にあります。

前提条件

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

手順

- 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、新しいプロパティを作成してから、ファイルを編集して、各 Kafka クラスターに対して接続情報と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** プロパティを使用して設定できます。

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

```
cluster-1.bootstrap.servers=CLUSTER-NAME-kafka-bootstrap-PROJECT-NAME:443 2
```

```
cluster-1.security.protocol=SSL 3
```

```
cluster-1.ssl.truststore.password=TRUSTSTORE-NAME
```

```
cluster-1.ssl.truststore.location=PATH-TO-TRUSTSTORE/truststore.cluster-1.jks
```

```
cluster-1.ssl.keystore.password=KEYSTORE-NAME
```

```
cluster-1.ssl.keystore.location=PATH-TO-KEYSTORE/user.cluster-1.p12_
```

```
cluster-2.bootstrap.servers=CLUSTER-NAME-kafka-bootstrap-<my-project>:443 4
```

```
cluster-2.security.protocol=SSL 5
```

```
cluster-2.ssl.truststore.password=TRUSTSTORE-NAME
```

```
cluster-2.ssl.truststore.location=PATH-TO-TRUSTSTORE/truststore.cluster-2.jks_
```

```
cluster-2.ssl.keystore.password=KEYSTORE-NAME
```

```
cluster-2.ssl.keystore.location=PATH-TO-KEYSTORE/user.cluster-2.p12_
```

```
cluster-1->cluster-2.enabled=true 6
```

```
cluster-1->cluster-2.topics=. * 7
```

```
cluster-2->cluster-1.enabled=true 8
```

```
cluster-2->cluster-1B->C.topics=. * 9
```

```
replication.policy.separator=- 10
```

```
sync.topic.acls.enabled=false 11
```

```
refresh.topics.interval.seconds=60 12
```

```
refresh.groups.interval.seconds=60 13
```

- 各 Kafka クラスターはそのエイリアスで識別されます。
- ブートストラップアドレスおよびポート 443 を使用する cluster-1 の接続情報。両方のクラスターがポート 443 を使用して、OpenShift ルートを使用して Kafka に接続します。
- ssl. プロパティは、cluster-1 の TLS 設定を定義します。
- cluster-2 の接続情報。
- ssl. プロパティは、cluster-2 の TLS 設定を定義します。
- cluster-1 クラスターから cluster-2 クラスターへのレプリケーションフロー。

- 7 cluster-1 クラスターから cluster-2 クラスターにすべてのトピックを複製します。
 - 8 クラスター-2 クラスターから cluster-1 クラスターへのレプリケーションフロー。
 - 9 cluster-2 クラスターから cluster-1 クラスターに特定のトピックを複製します。
 - 10 リモートトピック名の変更に使用する区切り文字を定義します。
 - 11 有効にすると、同期されたトピックに ACL が適用されます。デフォルトは **false** です。
 - 12 新規トピックが同期するチェックの間隔。
 - 13 新規コンシューマーグループの同期をチェックする間隔。
2. オプション：必要な場合は、リモートトピックの名前の自動変更を上書きするポリシーを追加します。その名前の前にソースクラスターの名前を追加する代わりに、トピックが元の名前を保持します。
- このオプションの設定は、active/passive バックアップおよびデータ移行に使用されます。

```
replication.policy.class=io.strimzi.kafka.connect.mirror.IdentityReplicationPolicy
```

3. オプション：コンシューマーグループオフセットを同期する場合は、設定を追加して同期を有効にし、管理します。

```
refresh.groups.interval.seconds=60
sync.group.offsets.enabled=true ①
sync.group.offsets.interval.seconds=60 ②
emit.checkpoints.interval.seconds=60 ③
```

- ① コンシューマーグループのオフセットを同期する任意設定。これは、active/passive 設定でのリカバリーに便利です。同期はデフォルトでは有効になっていません。
 - ② コンシューマーグループオフセットの同期が有効な場合は、同期の頻度を調整できます。
 - ③ オフセット追跡のチェック頻度を調整します。オフセット同期の頻度を変更する場合、これらのチェックの頻度も調整する必要がある場合があります。
4. ターゲットクラスターで ZooKeeper および Kafka を起動します。

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

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

5. 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。

6. 各ターゲットクラスターについて、トピックがレプリケートされていることを確認します。

■


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

9.5. レガシーモードでの MIRRORMAKER 2.0 の使用

この手順では、レガシーモードで MirrorMaker 2.0 を使用するように設定する方法を説明します。レガシーモードは、以前のバージョンの MirrorMaker をサポートします。

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

前提条件

従来のバージョンの MirrorMaker で現在使用しているプロパティファイルが必要です。

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

手順

1. MirrorMaker **consumer.properties** および **producer.properties** ファイルを編集し、MirrorMaker 2.0 機能をオフにします。
以下に例を示します。

```
replication.policy.class=org.apache.kafka.mirror.LegacyReplicationPolicy 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 以前のバージョンの MirrorMaker をエミュレートします。
- ❷ 内部チェックポイントおよびハートビートピックを含め、MirrorMaker 2.0 機能が無効化されました。

2. 以前のバージョンの MirrorMaker で使用したプロパティファイルで変更を保存し、MirrorMaker を再起動します。

```
su - kafka /opt/kafka/bin/kafka-mirror-maker.sh \
--consumer.config /opt/kafka/config/consumer.properties \
--producer.config /opt/kafka/config/producer.properties \
--num.streams=2
```

consumer プロパティはソースクラスターの設定を指定し、**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の場合、トピックがレプリケートされていることを確認します。

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

第10章 KAFKA クライアント

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

- Producer API により、アプリケーションはデータを Kafka ブローカーに送信できます。
- Consumer API により、アプリケーションは Kafka ブローカーからデータを消費できるようになります。
- AdminClient API は、トピック、ブローカーなどの Kafka クラスターを管理するための機能を提供します。

10.1. KAFKA クライアントを依存関係として MAVEN プロジェクトに追加

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

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

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

```
<?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">

  <!-- ... -->

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

  <!-- ... -->

</project>
```

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

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

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

<dependencies>
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.8.0.redhat-00002</version>
  </dependency>
</dependencies>

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

3. Maven プロジェクトをビルドします。

第11章 KAFKA STREAMS API の概要

Kafka Streams API を使用すると、アプリケーションは1つ以上の入力ストリームからデータを受信したり、マッピングやフィルタリングや結合などの複雑な操作を1つ以上の出力ストリームに書き込みできます。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。

11.1. KAFKA STREAMS API を依存関係として MAVEN プロジェクトに追加

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

前提条件

- 既存の **pom.xml** を持つ Maven プロジェクト。

手順

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

```
<?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">

  <!-- ... -->

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

  <!-- ... -->

</project>
```

2. **pom.xml** ファイルの **<dependencies>** セクションに **kafka-streams** を追加します。

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

  <!-- ... -->

  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-streams</artifactId>
```

```
        <version>2.8.0.redhat-00002</version>
      </dependency>
    </dependencies>

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

3. Maven プロジェクトをビルドします。

第12章 KAFKA BRIDGE

本章では、Red Hat Enterprise Linux 上の AMQ Streams Kafka Bridge の概要を紹介し、その REST API を使用して AMQ Streams と対話する方法を説明します。ローカル環境で Kafka Bridge を試すには、本章で後述する「[Kafka Bridge クイックスタート](#)」を参照してください。

その他のリソース

- リクエストおよび応答の例など、API ドキュメントを確認するには、『[Strimzi Kafka Bridge Documentation](#)』を参照してください。
- 分散トレーシング Kafka Bridge を設定するには、『[Kafka Bridge のトレースの有効化](#)』を参照してください。

12.1. KAFKA BRIDGE の概要

Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターとの対話を可能にする RESTful インターフェースが提供されます。また、クライアントアプリケーションによる Kafka プロトコルの変換は必要なく、Web API コネクションの利点が AMQ Streams に提供されます。

API には主に 2 つの resources-**consumers** と **topics**-- があります。これは、Kafka クラスターでコンシューマーおよびプロデューサーと対話するためにエンドポイント経由で公開され、アクセスが可能になります。リソースと関係があるのは Kafka ブリッジのみで、Kafka に直接接続されたコンシューマーやプロデューサーとは関係はありません。

HTTP リクエスト

Kafka Bridge は、以下の方法で Kafka クラスターへの HTTP リクエストをサポートします。

- トピックにメッセージを送信する。
- トピックからメッセージを取得する。
- トピックのパーティションリストを取得する。
- コンシューマーを作成および削除する。
- コンシューマーをトピックにサブスクライブし、このようなトピックからメッセージを受信できるようにする。
- コンシューマーがサブスクライブしているトピックの一覧を取得する。
- トピックからコンシューマーのサブスクライブを解除する。
- パーティションをコンシューマーに割り当てる。
- コンシューマーオフセットの一覧をコミットする。
- パーティションで検索して、コンシューマーが最初または最後のオフセットの位置、または指定のオフセットの位置からメッセージを受信できるようにする。

上記の方法で、JSON 応答と HTTP 応答コードのエラー処理を行います。メッセージは JSON またはバイナリー形式で送信できます。

クライアントは、ネイティブの Kafka プロトコルを使用する必要なくメッセージを生成して使用できます。

AMQ Streams インストールと同様に、Red Hat Enterprise Linux にインストールする Kafka Bridge ファイルをダウンロードできます。[「Kafka Bridge アーカイブのダウンロード」](#) を参照してください。

KafkaBridge リソースのホストおよびポートの設定に関する詳細は、[「Kafka Bridge プロパティーの設定」](#) を参照してください。

12.1.1. 認証および暗号化

HTTP クライアントと Kafka Bridge 間の認証と暗号化はまだサポートされていません。そのため、クライアントから Kafka Bridge に送信されるリクエストは以下のようになります。

- 暗号化されず、HTTPS ではなく HTTP を使用する必要がある。
- 認証なしで送信される。

Kafka Bridge と Kafka クラスターとの間で [TLS または SASL ベースの認証を設定できます](#)。

[プロパティーファイルを介して認証用に Kafka Bridge を設定します](#)。

12.1.2. Kafka Bridge へのリクエスト

データ形式と HTTP ヘッダーを指定し、有効なリクエストが Kafka Bridge に送信されるようにします。

API リクエストおよびレスポンス本文は、常に JSON としてエンコードされます。

12.1.2.1. コンテンツタイプヘッダー

すべてのリクエストについて **Content-Type** ヘッダーを提出する必要があります。唯一の例外は、**POST** リクエストボディが空で、**Content-Type** ヘッダーを追加するとリクエストが失敗します。

コンシューマー操作（**/consumers** エンドポイント）およびプロデューサー操作（エンドポイント）には異なる **/topics** ヘッダーが必要です。 **Content-Type**

コンシューマー操作の Content-Type ヘッダー

[埋め込みデータ形式に関係なく、リクエストボディにデータが含まれる場合に](#)、コンシューマー操作の **POST** リクエストは以下の **Content-Type** ヘッダーを提供する必要があります。

Content-Type: application/vnd.kafka.v2+json

プロデューサー操作の Content-Type ヘッダー

プロデューサー操作の実行時に、**POST** リクエストは、生成されたメッセージの埋め込みデータ形式を指定する **Content-Type** ヘッダーを提供する必要があります。これは、**json** または **binary** のいずれかになります。

表12.1 データ形式のコンテンツタイプヘッダー

埋め込みデータ形式	Content-Type ヘッダー
JSON	Content-Type: application/vnd.kafka.json.v2+json

埋め込みデータ形式	Content-Type ヘッダー
バイナリー	Content-Type: application/vnd.kafka.binary.v2+json

次のセクションで説明どおり、埋め込みデータ形式はコンシューマーごとに設定されます。

POST リクエストに空のボディがある場合は、**Content-Type** を設定しないでください。空のボディを使用して、デフォルト値のコンシューマーを作成できます。

12.1.2.2. 埋め込みデータ形式

埋め込みデータ形式は、Kafka メッセージがKafka Bridge によりプロデューサーからコンシューマーに HTTP で送信される際の形式です。サポートされる埋め込みデータ形式には、JSON またはバイナリーの 2 つがサポートされます。

`/consumers/groupid` エンドポイントを使用してコンシューマーを作成する場合、**POST** リクエスト本文で JSON またはバイナリーいずれかの埋め込みデータ形式を指定する必要があります。これは、要求ボディの **format** フィールドに指定されます。以下に例を示します。

```
{
  "name": "my-consumer",
  "format": "binary", ❶
  ...
}
```

❶ バイナリー埋め込みデータ形式。

コンシューマーの埋め込みデータ形式が指定されていない場合は、バイナリー形式が設定されます。

コンシューマーの作成時に指定する埋め込みデータ形式は、コンシューマーが消費する Kafka メッセージのデータ形式と一致する必要があります。

バイナリー埋め込みデータ形式を指定する場合は、以降のプロデューサーリクエストで、リクエスト本文にバイナリーデータがBase64 でエンコードされた文字列として含まれる必要があります。たとえば、**POST** 要求を `/topics/topicname` エンドポイントに指定してメッセージを送信する場合は、**value** をBase64 でエンコードする必要があります。

```
{
  "records": [
    {
      "key": "my-key",
      "value": "ZWR3YXJkdGhldGhyZWVsZWdnZWRjYXQ="
    },
  ]
}
```

プロデューサーリクエストは、埋め込みデータ形式に対応する **Content-Type** ヘッダーも提供する必要があります (例: **Content-Type: application/vnd.kafka.binary.v2+json**)。

12.1.2.3. メッセージの形式

`/topics` エンドポイントを使用してメッセージを送信する場合は、**records** パラメーターでリクエストボディにメッセージペイロードを入力します。

records パラメーターには、以下のオプションフィールドを含めることができます。

- メッセージの **key**
- メッセージの **value**
- 宛先の **partition**
- メッセージの **headers**

トピックへの POST リクエストの例

```
curl -X POST \
  http://localhost:8080/topics/my-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
      {
        "key": "my-key",
        "value": "sales-lead-0001"
        "partition": 2
        "headers": [
          {
            "key": "key1",
            "value": "QXBhY2hIEthZmthIGlzIHRoZSBib21iIQ==" ❶
          }
        ]
      },
    ]
  }'
```

❶ バイナリー形式のヘッダー値。Base64 としてエンコードされます。

12.1.2.4. Accept ヘッダー

コンシューマーを作成したら、以降のすべての GET リクエストには **Accept** ヘッダーが以下のような形式で含まれる必要があります。

Accept: `application/vnd.kafka.embedded-data-format.v2+json`

`embedded-data-format` は、**json** または **binary** のいずれかです。

たとえば、サブスクライブされたコンシューマーのレコードを JSON 埋め込みデータ形式で取得する場合、この **Accept** ヘッダーが含まれるようにします。

Accept: `application/vnd.kafka.json.v2+json`

12.1.3. Kafka Bridge のロガーの設定

AMQ Streams Kafka ブリッジでは、関連する OpenAPI 仕様によって定義される操作ごとに異なるログレベルを設定できます。

それぞれの操作には、ブリッジが HTTP クライアントから要求を受信する対応する API エンドポイントがあります。各エンドポイントのログレベルを変更すると、受信および送信 HTTP 要求に関する詳細なログ情報を生成することができます。

ロガーは **log4j.properties** ファイルで定義されます。このファイルには **healthy** および **ready** エンドポイントの以下のデフォルト設定が含まれています。

```
log4j.logger.http.openapi.operation.healthy=WARN, out
log4j.additivity.http.openapi.operation.healthy=false
log4j.logger.http.openapi.operation.ready=WARN, out
log4j.additivity.http.openapi.operation.ready=false
```

その他すべての操作のログレベルは、デフォルトで **INFO** に設定されます。ロガーは以下のようにフォーマットされます。

```
log4j.logger.http.openapi.operation.<operation-id>
```

ここで、**<operation-id>** は特定の操作の識別子です。以下は、OpenAPI 仕様によって定義される操作の一覧です。

- **createConsumer**
- **deleteConsumer**
- **subscribe**
- **unsubscribe**
- **poll**
- **assign**
- **commit**
- **send**
- **sendToPartition**
- **seekToBeginning**
- **seekToEnd**
- **seek**
- **healthy**
- **ready**
- **openapi**

12.1.4. Kafka Bridge API リソース

リクエストやレスポンスの例などを含む REST API エンドポイントおよび説明の完全リストは、「[Kafka Bridge API reference](#)」を参照してください。

12.1.5. Kafka Bridge アーカイブのダウンロード

AMQ Streams Kafka Bridge の zip 形式のディストリビューションは、Red Hat の Web サイトからダウンロードできます。

手順

- [カスタマーポータルから最新バージョンの Red Hat AMQ Streams Kafka Bridge アーカイブ](#)をダウンロードします。

12.1.6. Kafka Bridge プロパティーの設定

この手順では、AMQ Streams Kafka Bridge によって使用される Kafka および HTTP 接続プロパティーを設定する方法を説明します。

Kafka 関連のプロパティーの適切な接頭辞を使用して、他の Kafka クライアントとして Kafka Bridge を設定します。

- **kafka.** サーバー接続やセキュリティなどのプロデューサーとコンシューマーに適用される一般的な設定。
- **kafka.consumer.** コンシューマーのみに渡されるコンシューマー固有の設定。
- **kafka.producer.** プロデューサー固有の設定では、プロデューサーのみに渡されます。

HTTP プロパティーは、Kafka クラスターへの HTTP アクセスを有効にする他に、CPRS (Cross-Origin Resource Sharing) により Kafka Bridge のアクセス制御を有効化または定義する機能を提供します。CORS は、複数のオリジンから指定のリソースにブラウザでアクセスできるようにする HTTP メカニズムです。CORS を設定するには、許可されるリソースオリジンのリストと、それらにアクセスする HTTP メソッドを定義します。リクエストの追加の HTTP ヘッダーには [Kafka クラスターへのアクセスが許可されるオリジンが記述](#) されています。

前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [Kafka Bridge インストールアーカイブ](#)がダウンロードされます。

手順

1. AMQ Streams Kafka Bridge インストールアーカイブで提供される **application.properties** ファイルを編集します。
プロパティーファイルを使用して Kafka および HTTP 関連のプロパティーを指定し、分散トレーシングを有効にします。
 - a. Kafka コンシューマーやプロデューサー固有のプロパティーなど、標準の Kafka 関連のプロパティーを設定します。
以下を使用します。
 - **kafka.bootstrap.servers** Kafka クラスターへのホスト/ポート接続を定義します。
 - **kafka.producer.acks** HTTP クライアントに確認応答を提供する

- **kafka.consumer.auto.offset.reset** Kafka でオフセットのリセットを管理する方法を判断する。
Kafka [プロパティの設定に関する詳細は、Apache Kafka の Web サイト](#)を参照してください。

- b. HTTP 関連のプロパティを設定し、Kafka クラスターへの HTTP アクセスを有効にします。
以下に例を示します。

```
http.enabled=true
http.host=0.0.0.0
http.port=8080 ❶
http.cors.enabled=true ❷
http.cors.allowedOrigins=https://strimzi.io ❸
http.cors.allowedMethods=GET,POST,PUT,DELETE,OPTIONS,PATCH ❹
```

- ❶ 8080 番ポートでKafka Bridge をリッスンするデフォルトのHTTP 設定。
- ❷ CORS を有効にするには **true** に設定します。
- ❸ 許可される CORS オリジンのコンマ区切りリスト。URL または Java 正規表現を使用できます。
- ❹ CORS で許可される HTTP メソッドのコンマ区切りリスト。

- c. 分散トレースを有効または無効にします。

```
bridge.tracing=jaeger
```

プロパティからコードコメントを削除して、分散トレースを有効にします。

その他のリソース

- [15章分散トレーシング](#)
- [「Kafka Bridge のトレースの有効化」](#)

12.1.7. Kafka Bridge のインストール

以下の手順に従って、Red Hat Enterprise Linux に AMQ Streams Kafka Bridge をインストールします。

前提条件

- [AMQ Streams がホストにインストールされている。](#)
- [Kafka Bridge インストールアーカイブがダウンロードされます。](#)
- [Kafka Bridge 設定プロパティが設定されている。](#)

手順

1. まだ設定していない場合は、AMQ Streams Kafka Bridge インストールアーカイブを任意のディレクトリーに展開します。

2. 設定プロパティをパラメーターとして使用して、Kafka Bridge スクリプトを実行します。以下に例を示します。

```
./bin/kafka_bridge_run.sh --config-file=_path_/configfile.properties
```

3. ログインでインストールが正常に行われたことを確認します。

```
HTTP-Kafka Bridge started and listening on port 8080
HTTP-Kafka Bridge bootstrap servers localhost:9092
```

12.2. KAFKA BRIDGE クイックスタート

このクイックスタートを使用して、Red Hat Enterprise Linux で AMQ Streams Kafka Bridge を試すことができます。以下の方法について説明します。

- Kafka Bridge をインストールします。
- Kafka クラスターのトピックおよびパーティションへのメッセージを生成する。
- Kafka Bridge コンシューマーを作成する。
- 基本的なコンシューマー操作を実行する(たとえば、コンシューマーをトピックにサブスクライブする、生成したメッセージを取得するなど)。

このクイックスタートでは、HTTP リクエストはターミナルにコピーおよび貼り付けできる curl コマンドを使用します。

前提条件を確認し、本章に指定されている順序でタスクを行うようにしてください。

データ形式について

このクイックスタートでは、バイナリーではなく JSON 形式でメッセージを生成および消費します。リクエスト例で使用されるデータ形式および HTTP ヘッダーの詳細は、「[認証および暗号化](#)」を参照してください。

クイックスタートの前提条件

- [AMQ Streams](#) がホストにインストールされている。
- [単一ノードの AMQ Streams クラスター](#) が稼働している。
- [Kafka Bridge インストールアーカイブ](#) がダウンロードされます。

12.2.1. Kafka Bridge のローカルでのデプロイ

AMQ Streams Kafka Bridge のインスタンスをホストにデプロイします。インストールアーカイブで提供される **application.properties** ファイルを使用して、デフォルトの設定を適用します。

手順

1. **application.properties** ファイルを開き、デフォルトの **HTTP related settings** が定義されていることを確認します。

```
http.enabled=true
http.host=0.0.0.0
http.port=8080
```

これにより、ポート 8080 でリクエストをリッスンするように Kafka Bridge が設定されます。

2. 設定プロパティをパラメーターとして使用して、Kafka Bridge スクリプトを実行します。

```
./bin/kafka_bridge_run.sh --config-file=<path>/application.properties
```

次のステップ

- トピックおよびパーティションへのメッセージを生成します。

12.2.2. トピックおよびパーティションへのメッセージの作成

`topics` エンドポイントを使用して、JSON 形式のトピックへメッセージを生成します。

以下のように、メッセージの宛先パーティションをリクエスト本文に指定できます。`partitions` エンドポイントは、全メッセージの単一の宛先パーティションをパスパラメーターとして指定する代替方法を提供します。

手順

1. **kafka-topics.sh** ユーティリティーを使用して Kafka トピックを作成します。

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic bridge-quickstart-topic -
-partitions 3 --replication-factor 1 --config retention.ms=7200000 --config
segment.bytes=1073741824
```

3 つのパーティションを指定します。

2. トピックが作成されたことを確認します。

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

3. Kafka Bridge を使用して、作成したトピックに 3 つのメッセージを生成します。

```
curl -X POST \
  http://localhost:8080/topics/bridge-quickstart-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
      {
        "key": "my-key",
        "value": "sales-lead-0001"
      },
      {
        "value": "sales-lead-0002",
        "partition": 2
      },
      {
        "value": "sales-lead-0003"
```

```
}
]
}'
```

- **sales-lead-0001** は、キーのハッシュに基づいてパーティションに送信されます。
 - **sales-lead-0002** は、パーティション2 に直接送信されます。
 - **sales-lead-0003** は、ラウンドロビン方式を使用して **bridge-quickstart-topic** トピックのパーティションに送信されます。
4. リクエストが正常に行われると、Kafka Bridge は **200** (OK) コードと **application/vnd.kafka.v2+json** の **content-type** ヘッダーとともに **offsets** アレイを返します。各メッセージで、**offsets** アレイは以下を記述します。
- メッセージが送信されたパーティション。
 - パーティションの現在のメッセージオフセット。

応答の例

```
#...
{
  "offsets":[
    {
      "partition":0,
      "offset":0
    },
    {
      "partition":2,
      "offset":0
    },
    {
      "partition":0,
      "offset":1
    }
  ]
}
```

次のステップ

トピックおよびパーティションへのメッセージを作成したら、[Kafka Bridge コンシューマーを作成します](#)。

その他のリソース

- API リファレンスドキュメントの [「POST /topics/{topicname}」](#)
- API リファレンスドキュメントの [「POST /topics/{topicname}/partitions/{partitionid}」](#)

12.2.3. Kafka Bridge コンシューマーの作成

Kafka クラスターで何らかのコンシューマー操作を実行するには、まず [consumers エンドポイントを使用してコンシューマーを作成する必要があります](#)。コンシューマーは Kafka Bridge コンシューマーと呼ばれます。

手順

1. **bridge-quickstart-consumer-group** という名前の新しいコンシューマーグループに Kafka Bridge コンシューマーを作成します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group \
-H 'content-type: application/vnd.kafka.v2+json' \
-d '{
  "name": "bridge-quickstart-consumer",
  "auto.offset.reset": "earliest",
  "format": "json",
  "enable.auto.commit": false,
  "fetch.min.bytes": 512,
  "consumer.request.timeout.ms": 30000
}'
```

- コンシューマーには **bridge-quickstart-consumer** という名前を付け、埋め込みデータ形式は **json** として設定します。
- コンシューマーはログへのオフセットに自動でコミットしません。これは、**enable.auto.commit** が **false** に設定されているからです。このクイックスタートでは、オフセットを跡で手作業でコミットします。



注記

リクエスト本文にコンシューマー名を指定しない場合、Kafka Bridge はランダムコンシューマー名を生成します。

リクエストが正常に行われると、Kafka Bridge はレスポンス本文でコンシューマー ID(**instance_id**)とベース URL(**base_uri**)を **200** (OK) コードとともに返します。

応答の例

```
#...
{
  "instance_id": "bridge-quickstart-consumer",
  "base_uri": "http://<bridge-name>-bridge-service:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer"
}
```

2. ベース URL (**base_uri**) をコピーし、このクイックスタートの他のコンシューマー操作で使います。

次のステップ

上記で作成した Kafka Bridge コンシューマーを [トピックにサブスクライブ](#) できます。

その他のリソース

- API リファレンスドキュメントの [「POST /consumers/{groupid}」](#)

12.2.4. Kafka Bridge コンシューマーのトピックへのサブスクライブ

サブスクリプションエンドポイントを使用して、Kafka Bridge コンシューマーを1つ以上のトピックにサブスクライブします。サブスクライブすると、コンシューマーはトピックに生成されたすべてのメッセージの受信を開始します。

手順

- 前述の「トピックおよびパーティションへのメッセージの作成」の手順ですでに作成した **bridge-quickstart-topic** トピックに、コンシューマーをサブスクライブします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-
group/instances/bridge-quickstart-consumer/subscription \
-H 'content-type: application/vnd.kafka.v2+json' \
-d '{
  "topics": [
    "bridge-quickstart-topic"
  ]
}'
```

topics アレイには、（上記のように）単一のトピックまたは複数のトピックを含めることができます。正規表現に一致する複数のトピックにコンシューマーをサブスクライブする場合は、**topics** アレイの代わりに **topic_pattern** 文字列を使用できます。

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

次のステップ

Kafka Bridge コンシューマーをトピックにサブスクライブしたら、[コンシューマーからメッセージを取得](#)できます。

その他のリソース

- API リファレンスドキュメントの「[POST /consumers/{groupid}/instances/{name}/subscription](#)」

12.2.5. Kafka Bridge コンシューマーからの最新メッセージの取得

[records](#) エンドポイントからデータをリクエストして、Kafka Bridge コンシューマーから最新のメッセージを取得します。実稼働環境では、HTTP クライアントはこのエンドポイントを繰り返し（ループで）呼び出すことができます。

手順

1. 「トピックおよびパーティションへのメッセージの生成」の説明に従い、Kafka Bridge コンシューマーに新たなメッセージを生成します。
2. **GET** リクエストを **records** エンドポイントに送信します。

```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-
group/instances/bridge-quickstart-consumer/records \
-H 'accept: application/vnd.kafka.json.v2+json'
```

Kafka Bridge コンシューマーを作成し、サブスクライブすると、最初の GET リクエストによって空の応答が返されます。これは、ポーリング操作はリバランスプロセスをトリガーしてパーティションを割り当てます。

3. 手順2 を繰り返し、Kafka Bridge コンシューマーからメッセージを取得します。
Kafka Bridge は、**200** (OK) コードとともに、トピック名、キー、値、パーティション、および offset with the response body、および offset ボディーに messages ProcessingMode- の配列を返します。メッセージはデフォルトで最新のオフセットから取得されます。

```
HTTP/1.1 200 OK
content-type: application/vnd.kafka.json.v2+json
#...
[
  {
    "topic": "bridge-quickstart-topic",
    "key": "my-key",
    "value": "sales-lead-0001",
    "partition": 0,
    "offset": 0
  },
  {
    "topic": "bridge-quickstart-topic",
    "key": null,
    "value": "sales-lead-0003",
    "partition": 0,
    "offset": 1
  },
  #...
```



注記

空のレスポンスが返される場合は、[「トピックおよびパーティションへのメッセージの生成」](#)の説明に従い、コンシューマーに対して追加のレコードを生成し、メッセージの取得を再試行します。

次のステップ

Kafka Bridge コンシューマーからメッセージを取得したら、[ログへのオフセットをコミット](#)します。

その他のリソース

- API リファレンスドキュメントの [「GET /consumers/{groupid}/instances/{name}/records」](#)

12.2.6. ログへのオフセットのコミット

[オフセットエンドポイント](#)を使用して、Kafka Bridge コンシューマーによって受信されるすべてのメッセージに対して、手動でオフセットをログにコミットします。この操作が必要なのは、前述の [「Kafka Bridge コンシューマーの作成」](#) で作成した Kafka Bridge コンシューマーが **enable.auto.commit** の設定で **false** に指定されているからです。

手順

- bridge-quickstart-consumer** のオフセットをログにコミットします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/offsets
```

リクエスト本文は送信されないため、オフセットはコンシューマーによって受信されたすべて

のレコードに対してコミットされます。この代わりに、リクエスト本文に、オフセットをコミットするトピックおよびパーティションを指定するアレイ ([OffsetCommitSeekList](#)) を含めることができます。

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

次のステップ

オフセットをログにコミットしたら、[オフセットをシーク](#)のエンドポイントを試行します。

その他のリソース

- API リファレンスドキュメントの [/POST /consumers/{groupid}/instances/{name}/offsets/](#)

12.2.7. パーティションのオフセットのシーク

[positions エンドポイントを使用して](#)、Kafka Bridge コンシューマーを設定し、パーティションのメッセージを特定のオフセットから取得し、その後最新のオフセットから取得します。これは Apache Kafka では、シーク操作と呼ばれます。

手順

1. **quickstart-bridge-topic** トピックで、パーティション 0 の特定のオフセットをシークします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions \
-H 'content-type: application/vnd.kafka.v2+json' \
-d '{
  "offsets": [
    {
      "topic": "bridge-quickstart-topic",
      "partition": 0,
      "offset": 2
    }
  ]
}'
```

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

2. **GET** リクエストを **records** エンドポイントに送信します。

```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
-H 'accept: application/vnd.kafka.json.v2+json'
```

Kafka Bridge は、シークしたオフセットからのメッセージを返します。

3. 同じパーティションの最後のオフセットをシークし、デフォルトのメッセージ取得動作を復元します。この時点で、[positions/end](#) エンドポイントを使用します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions/end \
-H 'content-type: application/vnd.kafka.v2+json' \
-d '{
  "partitions": [
```

```
{
  "topic": "bridge-quickstart-topic",
  "partition": 0
}
]
```

リクエストが正常に行われると、Kafka Bridge は別の **204 No Content** コードを返します。



注記

また、[positions/beginning](#) エンドポイントを使用して、1 つ以上のパーティションの最初のオフセットをシークすることもできます。

次のステップ

このクイックスタートでは、AMQ Streams Kafka Bridge を使用して Kafka クラスターの一般的な操作をいくつか実行しました。これで、すでに作成した [Kafka Bridge コンシューマーを削除](#) できます。

その他のリソース

- API リファレンスドキュメントの [「POST /consumers/{groupid}/instances/{name}/positions」](#)
- API リファレンスドキュメントの [「POST /consumers/{groupid}/instances/{name}/positions/beginning」](#)
- API リファレンスドキュメントの [「POST /consumers/{groupid}/instances/{name}/positions/end」](#)

12.2.8. Kafka Bridge コンシューマーの削除

最後に、このクイックスタートを通して使用した Kafka Bridge コンシューマーを削除します。

手順

- **DELETE** リクエストを [instances](#) エンドポイントに送信し、Kafka Bridge コンシューマーを削除します。

```
curl -X DELETE http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer
```

リクエストが正常に行われると、Kafka Bridge は **204 No Content** コードのみを返します。

その他のリソース

- API リファレンスドキュメントの [「DELETE /consumers/{groupid}/instances/{name}」](#)

第13章 KERBEROS(GSSAPI)認証の使用

AMQ Streams では、Kafka クラスターへのシングルサインオンアクセスをセキュアにするために、Kerberos(GSSAPI)認証プロトコルの使用がサポートされます。GSSAPI は Kerberos 機能の API ラッパーで、基礎となる実装の変更からアプリケーションを渡します。

Kerberos は、対称暗号化と信頼できるサードパーティーである 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)認証を使用するように AMQ STREAMS を設定

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 ホストで設定されていることを前提としています。

この手順では、たとえば以下を設定する方法を説明します。

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 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、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とともに、1 台のホストでの ZooKeeper および Kafka インストール1 つに設定された Kerberos を説明します。

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 [サーバーの設定方法](#)や、[ブローカーホストのクライアントの設定手順の詳細](#)は、「[RHEL の設定における Kerberos の例](#)」を参照してください。

Kerberos のデプロイ方法は、お使いのオペレーティングシステムによって異なります。Red Hat は、Red Hat Enterprise Linux に Kerberos を設定する際に Identity Management(IdM)を使用することを推奨します。Oracle または IBM JDK のユーザーは、JCE(Java Cryptography Extension)をインストールする必要があります。

- [Oracle JCE](#)
- [IBM JCE](#)

認証のサービスプリンシパルの追加

Kerberos サーバーから、ZooKeeper、Kafka ブローカー、および Kafka プロデューサー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 でプリンシパルキーを格納するサービスプリンシパルおよびキータブを作成します。
以下に例を示します。

- **zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
ZooKeeper サービスプリンシパルは、Kafka **config/server.properties** ファイルの **zookeeper.connect** 設定と同じホスト名である必要があります。

```
zookeeper.connect=node1.example.redhat.com:2181
```

ホスト名が同じではない場合は、localhost が使用され、認証に失敗します。

2. ホストにディレクトリーを作成し、キータブファイルを追加します。
以下に例を示します。

```
/opt/kafka/krb5/zookeeper-node1.keytab
/opt/kafka/krb5/kafka-node1.keytab
/opt/kafka/krb5/kafka-producer1.keytab
/opt/kafka/krb5/kafka-consumer1.keytab
```

3. **kafka** ユーザーがディレクトリーにアクセスできることを確認します。

```
chown kafka:kafka -R /opt/kafka/krb5
```

Kerberos ログインを使用するように ZooKeeper を設定する

zookeeper 用に事前に作成されたユーザープリンシパルとキータブを使用して、認証に Kerberos Key Distribution Center(KDC)を使用するように ZooKeeper を設定します。

1. **opt/kafka/config/jaas.conf** ファイルを作成または修正して、ZooKeeper のクライアントおよびサーバー操作をサポートします。

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true ①
  storeKey=true ②
  useTicketCache=false ③
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab" ④
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM"; ⑤
};

Server {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
```



```

keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumServer {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
    principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumLearner {
    com.sun.security.auth.module.Krb5LoginModule required debug=true
    useKeyTab=true
    storeKey=true
    keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
    principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

```

- ❶ **true** に設定して、キータブからプリンシパルキーを取得します。
- ❷ プリンシパルキーを保存するには、**true** に設定します。
- ❸ チケットキャッシュから TGT(Ticket Granting Ticket)を取得するには、**true** に設定します。
- ❹ **keyTab** プロパティは、Kerberos KDC からコピーしたキータブファイルの場所を参照します。場所およびファイルは **kafka** ユーザーが読み取りできる必要があります。
- ❺ **principal** プロパティは、**SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-NAME** という形式に続く 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 ログインプリンシパル名の一部としてホスト名を使用するかどうかを指定します。クラスター内のすべてのノードに単一のキータブを使用している場合、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用に各ブローカーホストに個別のキータブおよび完全修飾プリンシパルを生成することが推奨されます。

- 3 Kerberos ネゴシエーションのプリンシパル名からレルム名を取り除くかどうかを制御します。この設定は **false** に設定することを推奨します。
- 4 ZooKeeper サーバーおよびクライアントの SASL 認証メカニズムを有効にします。
- 5 **RequireSasl** プロパティは、マスター選択などのクォラムイベントに SASL 認証が必要であるかどうかを制御します。
- 6 **loginContext** プロパティは、指定されたコンポーネントの認証設定に使用される JAAS 設定でログインコンテキストの名前を識別します。loginContext 名は、**opt/kafka/config/jaas.conf** ファイルの関連するセクションの名前に対応します。
- 7 識別に使用されるプリンシパル名を形成する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るホスト名に自動的に解決されます。

3. JVM パラメーターで ZooKeeper を開始し、Kerberos ログイン設定を指定します。

```
su - kafka
export EXTRA_ARGS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/zookeeper-server-
start.sh -daemon /opt/kafka/config/zookeeper.properties
```

デフォルトのサービス名(**zookeeper**)を使用していない場合は、**-Dzookeeper.sasl.client.username=NAME** パラメーターを使用して名前を追加します。



注記

/etc/krb5.conf の場所を使用している場合は、ZooKeeper、Kafka、または Kafka プロデューサーおよびコンシューマーの起動時に **-Djava.security.krb5.conf=/etc/krb5.conf** を指定する必要はありません。

Kerberos ログインを使用するように Kafka ブローカーサーバーを設定します。

kafka 用に事前に作成されたユーザープリンシパルとキータブを使用して、認証に Kerberos Key Distribution Center(KDC)を使用するように Kafka を設定します。

1. 以下の要素で **opt/kafka/config/jaas.conf** ファイルを変更します。

```
KafkaServer {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/kafka-node1.keytab"
  principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
```



```
keyTab="/opt/kafka/krb5/kafka-node1.keytab"
principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

- リスナーがSASL/GSSAPI ログインを使用できるように、**config/server.properties** ファイルのリスナー設定を変更して、Kafka クラスターの各ブローカーを設定します。
SASL プロトコルをリスナーのセキュリティープロトコルのマッピングに追加し、不要なプロトコルを削除します。

以下に例を示します。

```
# ...
broker.id=0
# ...
listeners=SECURE://:9092,REPLICATION://:9094 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 リスナーが設定された2つのリスナーは、クライアントとの汎用通信（通信用のTLS サポート）とブローカー間の通信を行うレプリケーションリスナーの2つです。
- ❷ TLS 対応のリスナーの場合、プロトコル名はSASL_PLAINTEXTです。TLS が有効ではないコネクタの場合、プロトコル名はSASL_PLAINTEXTになります。SSL が必要ない場合は、**ssl.*** プロパティを削除できます。
- ❸ Kerberos 認証のSASL メカニズムは**GSSAPI**です。
- ❹ ブローカー間通信のKerberos 認証。
- ❺ 認証要求に使用されるサービスの名前が指定され、同じKerberos 設定を使用している可能性がある他のサービスと区別されます。

- JVM パラメーターを使用してKafka ブローカーを起動し、Kerberos ログイン設定を指定します。

```
su - kafka
export KAFKA_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/kafka-server-
start.sh -daemon /opt/kafka/config/server.properties
```

ブローカーおよびZooKeeper クラスターが以前設定され、Kerberos 以外の認証システムで動作している場合、ZooKeeper およびブローカークラスターを起動し、ログで設定エラーの有無を確認できます。

ブローカーおよびZooKeeper インスタンスを起動すると、クラスターはKerberos 認証用に設定されます。

Kafka プロデューサーおよびコンシューマクライアントがKerberos 認証を使用するように設定する

以前 **producer1** および **consumer1** 用に作成されたユーザープリンシパルおよびキータブを使用して、認証に Kerberos Key Distribution Center(KDC)を使用するように Kafka プロデューサーおよびコンシューマクライアントを設定します。

1. Kerberos 設定をプロデューサーまたはコンシューマー設定ファイルに追加します。
以下に例を示します。

/opt/kafka/config/producer.properties

```
# ...
sasl.mechanism=GSSAPI 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）セキュリティー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** に定義されたものと同じプロパティーを使用した JAAS の設定。

/opt/kafka/config/consumer.properties

```
# ...
sasl.mechanism=GSSAPI
security.protocol=SASL_PLAINTEXT
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
    useKeyTab=true \
    useTicketCache=false \
    storeKey=true \
    keyTab="/opt/kafka/krb5/consumer1.keytab" \
    principal="consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

2. クライアントを実行して、Kafka ブローカーからメッセージを送受信できることを確認します。
プロデューサークライアント：

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true";/opt/kafka/bin/kafka-console-producer.sh --producer.config
/opt/kafka/config/producer.properties --topic topic1 --bootstrap-server
```

```
node1.example.redhat.com:9094
```

コンシューマクライアント :

```
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
```

その他のリソース

- Kerberos の man ページ : `krb5.conf(5)`、`kinit(1)`、`klist(1)`、および `kdestroy(1)`
- [RHEL セットアップ上の Kerberos サーバーの例](#)
- [Kerberos チケットを使用して Kafka クラスターと認証するクライアントアプリケーションの例](#)

第14章 CRUISE CONTROL によるクラスターのリバランス



重要

Cruise Control によるクラスターのリバランスはテクノロジープレビューの機能です。テクノロジープレビューの機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあります。Red Hat は、本番環境でのテクノロジープレビュー機能の実装は推奨しません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、[「テクノロジープレビュー機能のサポート範囲」](#)を参照してください。

[Cruise Control](#) を AMQ Streams クラスターにデプロイし、これを使用して Kafka ブローカー全体で負荷を分散できます。

Cruise Control は、クラスターワークロードの監視、事前定義の制約を基にしたクラスターのリバランス、異常の検出および修正などの Kafka の操作を自動化するオープンソースのシステムです。4 つのコンポーネント (Load Monitor、Analyzer、Anomaly Detector、および Executor) および REST API で構成されます。

AMQ Streams および Cruise Control の両方が Red Hat Enterprise Linux にデプロイされると、Cruise Control REST API から Cruise Control 機能にアクセスできます。以下の機能がサポートされます。

- **最適化ゴールと容量制限の設定**
- **`/rebalance` エンドポイントを使用して以下を実行します。**
 - 設定された最適化ゴールまたはリクエストパラメーターとして提供されたユーザー提供ゴールに基づいて、最適化プロポーザルをドライランとして生成します。
 - 最適化プロポーザルを開始して Kafka クラスターをリバランスする
- **`/user_tasks` エンドポイントを使用したアクティブなリバランス操作の進捗の確認**
- **`/stop_proposal_execution` エンドポイントを使用したアクティブなリバランス操作の停止**

異常検出、通知、独自ゴールの作成、トピックレプリケーション係数の変更など、その他の Cruise Control の機能は現在サポートされていません。Web UI コンポーネント (Cruise Control Frontend) はサポートされません。

Red Hat Enterprise Linux 上の Cruise Control for AMQ Streams は、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。詳細は、[「Cruise Control アーカイブのダウンロード」](#)を参照してください。

14.1. CRUISE CONTROL とは

Cruise Control は、効率的な Kafka クラスターを実行する時間と作業を削減し、ブローカー全体でワークロードの均等に分散されます。

通常、クラスターの負荷は時間とともに不均等になります。大量のメッセージトラフィックを処理するパーティションは、使用可能なブローカー全体で不均等に分散される可能性があります。クラスターを再分散するには、管理者はブローカーの負荷を監視し、トラフィックの多いパーティションを容量に余裕のあるブローカーに手作業で再割り当てします。

Cruise Control はこのクラスターのリバランス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に基づいて、リソース使用率のワークロードモデルを構築します。設定可能な最適化ゴールのセットを使用して、Cruise Control に、パーティションの割り当てをより均等にするドライラン最適化プロポーザルを生成するよう指示できます。

ドライラン最適化プロポーザルを確認した後、Cruise Control にそのプロポーザルを基にクラスターのリバランスを開始するよう指示したり、新しいプロポーザルを生成するよう指示できます。

クラスターのリバランス操作が完了すると、ブローカーがより効果的に使用され、Kafka クラスターの負荷がより均等に分散されます。

その他のリソース

- [Cruise Control の Wiki](#)
- [「最適化ゴールの概要」](#)
- [「最適化プロポーザルの概要」](#)
- [容量の設定](#)

14.2. CRUISE CONTROL アーカイブのダウンロード

Red Hat カスタマーポータルから Cruise Control for AMQ Streams on Red Hat Enterprise Linux の zip 形式のディストリビューションを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。

ランタイム時に読み込まれると、Metric Reporter は3つの自動作成されるトピックの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算出します。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- Kafka と ZooKeeper が稼働している必要があります。
- [「Cruise Control アーカイブのダウンロード」](#)。

手順

Kafka クラスターの各ブローカーと1度に1つのブローカー。

1. Kafka ブローカーを停止します。

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

2. Cruise Control Metrics Reporter **.jar** ファイルを Kafka ライブラリーディレクトリーにコピーします。

```
cp /opt/cruise-control/libs/cruise-control-metrics-reporter-y.y.yyy.redhat-0000x.jar
/opt/kafka/libs
```

3. Kafka 設定ファイル(**/opt/kafka/config/server.properties**)で Cruise Control Metrics Reporter を設定します。

- a. **CruiseControlMetricsReporter** クラスを **metric.reporters** 設定オプションに追加します。既存の Metrics 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と値を Kafka 設定ファイルに追加します。

```
cruise.control.metrics.topic.auto.create=true
cruise.control.metrics.topic.num.partitions=1
cruise.control.metrics.topic.replication.factor=1
```

これらのオプションにより、Cruise Control Metrics Reporter が **DELETE** のログクリーンアップポリシーで **CruiseControlMetrics** トピックを作成できます。詳細は、[Cruise Control Metrics の「Auto-created topics and Log cleanup policy」](#)を参照してください。

4. 必要に応じて SSL を設定します。

- a. Kafka 設定ファイル(**/opt/kafka/config/server.properties**)で、関連するクライアント設定プロパティーを設定して Cruise Control Metrics Reporter と Kafka ブローカー間の SSL を設定します。

Metrics Reporter は、すべての標準プロデューサー固有の設定プロパティーに **cruise.control.metrics.reporter** プレフィックスを付けて受け入れます。たとえば、**cruise.control.metrics.reporter.ssl.truststore.password** のようになります。

- b. Cruise Control プロパティーファイル(**/opt/cruise-control/config/cruisecontrol.properties**)で、関連するクライアント設定プロパティーを設定し、Kafka ブローカーと Cruise Control サーバー間の SSL を設定します。

Cruise Control は Kafka から SSL クライアントプロパティーオプションを継承するため、これらのプロパティーをすべての Cruise Control サーバークライアントに使用します。

5. Kafka ブローカーを再起動します。


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

6. 残りのブローカーにステップ1-5 を繰り返します。

14.4. CRUISE CONTROL の設定および起動

Cruise Control によって使用されるプロパティを設定し、**cruise-control-start.sh** スクリプトを使用して Cruise Control サーバーを起動します。サーバーは、Kafka クラスター全体に対して1台のマシンでホストされます。

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

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- [「Cruise Control アーカイブのダウンロード」](#)
- [「Cruise Control 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. 以下の設定例で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 master 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 ❼

# The zookeeper connect of the Kafka cluster
zookeeper.connect=localhost:2181 ❽
```

■

- 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の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)を使用するデフォルトの最適化ゴールのコンマ区切りリスト。数多くのマスター最適化ゴール（5 を参照）はすでにデフォルトの最適化ゴールとして設定されています。必要に応じてゴールを追加または削除できます。詳細は、「[最適化ゴールの概要](#)」を参照してください。
 - 5 FQDN を使用したマスター最適化ゴールのコンマ区切りリスト。最適化プロポーザルの生成に使用されるゴールを、完全に除外するには、一覧から削除します。詳細は、「[最適化ゴールの概要](#)」を参照してください。
 - 6 FQDN を使用したハードゴールのコンマ区切りリスト。マスター最適化ゴールの 7 つはすでにハードゴールとして設定されています。必要に応じてゴールを追加または削除できます。詳細は、「[最適化ゴールの概要](#)」を参照してください。
 - 7 デフォルトの最適化ゴールから生成された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。サーバーは、デフォルトでポート 9092 で起動します。必要に応じて、別のポートを指定します。

```
cd /opt/cruise-control/
./bin/cruise-control-start.sh config/cruisecontrol.properties PORT
```

4. Cruise Control が稼働していることを確認するには、Cruise Control サーバーの **/state** エンドポイントに GET リクエストを送信します。

```
curl 'http://HOST:PORT/kafkacruisecontrol/state'
```

自動作成されたトピック

以下の表は、Cruise Control の起動時に自動作成される 3 つのトピックを表しています。これらのトピックは、Cruise Control が適切に動作するために必要であるため、削除または変更しないでください。

表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。

自動作成されたトピック	作成元	機能
<code>__KafkaCruiseControlPartitionMetricSamples</code>	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。
<code>__KafkaCruiseControlModelTrainingSamples</code>	Cruise Control	クラスターワークロードモデル の作成に使用されるメトリクスサンプルを格納します。

自動作成されたトピックでログコンパクションを無効にするには、[「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て、適切に機能しなくなりました。

その他のリソース

- [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

Kafka クラスターを再分散するため、Cruise Control は [最適化ゴールを使用して最適化プロポーザルを生成します](#)。最適化ゴールは、Kafka クラスター全体のワークロード再分散およびリソース使用の制約です。

Red Hat Enterprise Linux 上の AMQ Streams は、Cruise Control プロジェクトで開発されたすべての最適化ゴールをサポートします。以下に、サポートされるゴールをデフォルトの優先度順に示します。

1. ラックアウェアネス (Rack Awareness)
2. トピックのセットに対するブローカーごとのリーダーレプリカの最小数
3. レプリカの容量
4. 容量: ディスク容量、ネットワークインバウンド容量、ネットワークアウトバウンド容量
5. CPU 容量
6. レプリカの分散
7. 潜在的なネットワーク出力
8. リソース分布: ディスク使用率の分布、ネットワークインバウンド使用率の分布、ネットワークアウトバウンド使用率の分布。
9. リーダーへの単位時間あたりバイト流入量の分布
10. トピックレプリカの分散
11. CPU 使用率の分散
12. リーダーレプリカの分散

13. 優先リーダーの選択
14. Kafka Assigner ディスク使用分布
15. ブローカー内ディスク容量
16. ブローカー内のディスク使用量

各最適化ゴールの詳細は、[Cruise Control Wiki](#) の [Goals](#) を参照してください。

Cruise Control プロパティファイルのゴールの設定

cruise-control/config/ ディレクトリー内の **cruisecontrol.properties** ファイルで、最適化ゴールを設定します。必ず満たさなければならないハード最適化ゴールと、マスターおよびデフォルトの最適化ゴールの設定があります。

任意。ユーザー提供の最適化ゴールは、**/rebalance** エンドポイントへのリクエストのパラメーターとしてランタイム時に設定されます。

最適化ゴールは、ブローカーリソースのあらゆる容量制限の対象となります。

以下のセクションでは、各ゴール設定の詳細を説明します。

マスター最適化ゴール

マスター最適化ゴールはすべてのユーザーが使用できます。マスター最適化ゴールにリストされていないゴールは、Cruise Control 操作で使用できません。

以下のマスター最適化ゴールは、**cruisecontrol.properties** ファイルで優先順に **goals** プロパティに事前設定されています。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; ReplicaDistributionGoal;
PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

簡素化するために、最適化プロポーザルの生成に1つ以上のゴールをそのまま除外する必要がある場合を除き、事前設定されたマスター最適化ゴールを変更しないことが推奨されます。必要な場合、マスター最適化ゴールの優先順位はデフォルトの最適化ゴールの設定で変更できます。

事前設定されたマスター最適化ゴールを変更する必要がある場合は、**goals** プロパティにゴールのリストを優先度が高いものから順に指定します。**cruisecontrol.properties** ファイルに示されるように、完全修飾ドメイン名を使用します。

少なくとも1つのマスターゴールを指定するか、Cruise Control によってクラッシュします。



注記

事前設定されたマスター最適化ゴールを変更する場合、設定済みの **hard.goals** が、設定したマスター最適化ゴールのサブセットになるようにする必要があります。そうでないと、最適化プロポーザルの生成時にエラーが発生します。

ハードゴールおよびソフトゴール

ハードゴールは最適化プロポーザルで必ず満たさなければならないゴールです。ハードゴールとして設定されていないゴールはソフトゴールと呼ばれます。ソフトゴールはベストエフォートのゴールとみなすことができます。最適化プロポーザルで満たす必要はありませんが、最適化の計算に含まれています。

Cruise Control は、すべてのハードゴールを満たし、優先度順にできるだけ多くのソフトゴールを満たす最適化プロポーザルを算出します。すべてのハードゴールを満たさない最適化プロポーザルは Analyzer によって拒否され、ユーザーに送信されません。



注記

たとえば、クラスター全体でトピックのレプリカを均等に分散するソフトゴールがあります（トピックレプリカ分散のゴール）。このソフトゴールを無視すると、設定されたハードゴールがすべて有効になる場合、Cruise Control はこのソフトゴールを無視します。

以下のマスター最適化ゴールは、**hard.goals** プロパティの **cruisecontrol.properties** ファイルのハードゴールとして事前設定されます。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal
```

ハードゴールを変更するには、**hard.goals** プロパティを編集し、完全修飾ドメイン名を使用して目的のゴールを指定します。

ハードゴールの数を増やすと、Cruise Control が有効な最適化プロポーザルを計算して生成する可能性が減ります。

デフォルトの最適化ゴール

Cruise Control はデフォルトの最適化ゴールリストを使用して、キャッシュされた最適化プロポーザルを生成します。詳細は、「[最適化プロポーザルの概要](#)」を参照してください。

[ユーザー提供の最適化ゴール](#)を設定すると、ランタイム時にデフォルトの最適化ゴールを上書きできます。

以下のデフォルトの最適化ゴールは、**default.goals** プロパティで優先順位の降順で **cruisecontrol.properties** ファイルで事前設定されます。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal;
ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal;
NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal;
CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal;
LeaderBytesInDistributionGoal
```

少なくとも1つのデフォルトゴールを指定するか、Cruise Control によってクラッシュします。

デフォルトの最適化ゴールを編集するには、**default.goals** プロパティでゴールのリストを優先度の高いものとして降順に指定します。デフォルトのゴールはマスター最適化ゴールのサブセットである必要があります。完全修飾ドメイン名を使用します。

ユーザー提供の最適化ゴール

ユーザー提供の最適化ゴールは、特定の最適化プロポーザルの設定済みのデフォルトゴールを絞り込みます。必要に応じて、HTTP リクエストのパラメーターとして **/rebalance** エンドポイントに設定できます。詳細は、「[最適化プロポーザルの生成](#)」を参照してください。

ユーザー提供の最適化ゴールは、さまざまな状況の最適化プロポーザルを生成できます。たとえば、ディスクの容量やディスクの使用率を考慮せずに、Kafka クラスター全体でリーダーレプリカの分散を最適化したい場合があります。そのため、リーダーレプリカ分散の単一のゴールが含まれる **/rebalance** エンドポイントにリクエストを送信します。

ユーザー提供の最適化ゴールには以下が必要になります。

- 設定済みの[ハードゴール](#)がすべて含まれるようにする必要があります。そうでないと、エラーが発生します。
- [マスター最適化ゴールのサブセット](#)である必要があります。

最適化プロポーザルの設定済みのハードゴールを無視するには、`skip_hard_goals_check=true` パラメーターをリクエストに追加します。

その他のリソース

- [「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、提案された変更の概要です。これが適用されると、パーティションワークロードがブローカー間でより均等に分散されます。[各最適化プロポーザルは、そのプロポーザルの生成に使用された最適化ゴールのセットが基になっており、ブローカーリソースの設定済みの容量制限が適用されます。](#)

`/rebalance` エンドポイントに POST リクエストを行うと、応答で最適化プロポーザルが返されます。プロポーザルの情報を使用して、プロポーザルに基づいてクラスターのリバランスを開始するかどうかを決定します。この代わりに、最適化ゴールを変更し、別のプロポーザルを生成することができます。

デフォルトでは、最適化プロポーザルは個別に開始する必要のあるドライランとして生成されます。生成できる最適化プロポーザルの数に制限はありません。

キャッシュされた最適化プロポーザル

Cruise Control は、[設定されたデフォルトの最適化ゴールを基にしてキャッシュされた最適化プロポーザルを維持します。](#) キャッシュされた最適化プロポーザルはワークロードモデルから生成され、Kafka クラスターの現在の状況を反映するために 15 分ごとに更新されます。

以下のゴール設定が使用されている場合に、最新のキャッシュされた最適化プロポーザルが返されます。

- デフォルトの最適化ゴール
- 現在のキャッシュされたプロポーザルで合致できるユーザー提供の最適化ゴール

キャッシュされた最適化プロポーザルの更新間隔を変更するには、`cruisecontrol.properties` ファイルの `proposal.expiration.ms` 設定を編集します。更新間隔を短くすると、Cruise Control サーバーの負荷が増えますが、変更が頻繁に行われるクラスターでは、更新間隔を短くするよう考慮してください。

最適化プロポーザルの内容

以下の表は、最適化プロポーザルに含まれるプロパティを表しています。

表14.2 最適化プロポーザルに含まれるプロパティ

プロパティ	説明
-------	----

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す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。クラスターのリバランスの完了にかかる時間が長くなる MB の数が増えます。</p>
n intra-broker replica (y MB) moves	<p>n: クラスターのブローカーのディスク間で転送されるパーティションレプリカの合計数。</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 未満です。</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。値が大きいくほど、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と、同じブローカーのディスク間で移動する方が個別のブローカー間で移動するよりも影響度が低くなります（inter-broker replica movesを参照してください）。</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できます。</p> <p>cruisecontrol.properties ファイル で、topics.excluded.from.partition.movement プロパティで正規表現を指定します。</p> <p>/rebalance エンドポイントへの POST リクエストでは、excluded_topics パラメーターに正規表現を指定します。</p> <p>正規表現に一致するトピックは応答に一覧表示され、クラスターのリバランスから除外されます。</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	<p>n: 最適化プロポーザルの基となるメトリクスウィンドウの数。</p>
n% of the partitions covered	<p>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）。</p>

プロパティ	説明
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を測定します。</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 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>

その他のリソース

- [「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構成されています。プロポーザルを開始すると、Cruise Control サーバーはこれらのコマンドを Kafka クラスターに適用します。

パーティション再割り当てコマンドは、以下のいずれかの操作で構成されます。

- **パーティションの移動:** パーティションレプリカとそのデータを新しい場所に転送します。パーティションの移動は、以下の2つの形式のいずれかになります。
 - **ブローカー間の移動:** パーティションレプリカを、別のブローカーのログディレクトリーに移動します。
 - **ブローカー内の移動:** パーティションレプリカを、同じブローカーの異なるログディレクトリーに移動します。
- **リーダーシップの移動:** パーティションのレプリカのリーダーを切り替えます。

Cruise Control によって、パーティション再割り当てコマンドがバッチで Kafka クラスターに発行されます。リバランス中のクラスターのパフォーマンスは、各バッチに含まれる各タイプの移動数に影響されます。

パーティション再割り当てコマンドを設定するには、「[Rebalance tuning options](#)」を参照してください。

レプリカの移動ストラテジー

クラスターリバランスのパフォーマンスは、パーティション再割り当てコマンドのバッチに適用されるレプリカ移動ストラテジーの影響も受けます。デフォルトでは、Cruise Control は

BaseReplicaMovementStrategy を使用します。これは、生成された順序でコマンドを適用します。ただし、プロポーザルの初期に非常に大きなパーティションの再割り当てがある場合、このストラテジーによって他の再割り当ての適用が遅くなる可能性があります。

Cruise Control は、最適化プロポーザルに適用できる代替のレプリカ移動ストラテジーを3つ提供します。

- **PrioritizeSmallReplicaMovementStrategy**: 再割り当てを昇順で並べ替えます。
- **PrioritizeLargeReplicaMovementStrategy**: 再割り当てを降順で並べ替えます。
- **PostponeUrpReplicaMovementStrategy**: 非同期のレプリカがないパーティションのレプリカの再割り当てを優先します。

これらのストラテジーをシーケンスとして設定できます。最初のストラテジーは、内部ロジックを使用して2つのパーティション再割り当ての比較を試みます。再割り当てが同等である場合は、順番を決定するために再割り当てをシーケンスの次のストラテジーに渡します。

レプリカの移動ストラテジーを設定するには、「Rebalance [tuning options](#)」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ための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、デフォルトの Cruise Control 設定の **cruisecontrol.properties** ファイル
- **/rebalance** エンドポイントへの POST 要求のパラメーター

両メソッドの関連設定の概要は、以下の表で説明されています。

表14.3 リバランスパフォーマンスチューニングの設定

プロパティーおよび要求パラメーターの設定	説明	デフォルト値
num.concurrent.partition.movements.per.broker	各パーティション再割り当てバッチでのブローカー間パーティション移動の最大数。	5
concurrent_partition_movements_per_broker		
num.concurrent.intra.broker.partition.movements	各パーティション再割り当てバッチでのブローカー内パーティション移動の最大数。	2
concurrent_intra_broker_partition_movements		
num.concurrent.leader.movements	各パーティション再割り当てバッチにおけるパーティションリーダー変更の最大数。	1000
concurrent_leader_movements		
default.replication.throttle		null（無制限）

プロパティおよび要求パラメーターの設定	説明	デフォルト値
	パーティションの再割り当てに割り当てられる帯域幅（バイト/秒単位）。	
<code>replication_throttle</code>		
<code>default.replica.movement.strategies</code>	<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。PrioritizeSmallReplicaMovementStrategy、PrioritizeLargeReplicaMovementStrategy、およびPostponeUrpReplicaMovementStrategyの3つのストラテジーがあります。</p> <p>プロパティには、ストラテジークラスの完全修飾名のコンマ区切りリストを使用します（各クラス名の先頭にcom.linkedin.kafka.cruisecontrol.executor.strategy.を追加します）。</p> <p>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>	BaseReplicaMovementStrategy
<code>replica_movement_strategies</code>		

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。

その他のリソース

- 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。ファイルは、以下のいずれかのタイプでプロパティで構成されます。

- 文字列
- 数値
- ブール値

Cruise Control Wiki の [Configurations](#) セクションに記載されているすべてのプロパティ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、破損している場合は、試行された最適化に失敗します。これにより、最適化プロポーザルの生成に最適化が使用されなくなります。

Kafka ブローカーリソースの容量制限は、**cruise-control/config** の3つの **.json** ファイルのいずれかで指定します。

- **capacityJBOD.json**: JBOD Kafka デプロイメント（デフォルトファイル）で使用されます。
- **capacity.json**: 各ブローカーに同じ CPU コアを持つ非JBOD Kafka デプロイメントで使用します。
- **capacityCores.json**: 各ブローカーに CPU コアの数異なる、JBOD 以外の Kafka デプロイメントで使用します。

cruisecontrol.properties の **capacity.config.file** プロパティにファイルを設定します。選択され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**: パーセンテージ(0-100)または多数のコアとしての CPU 使用率
- **NW_IN**: インバウンドネットワークスループット (KB/秒)
- **NW_OUT**: アウトバウンドネットワークスループット (KB/秒)

Cruise Control によって監視されるすべてのブローカーに同じ容量制限を適用するには、ブローカーID **-1** の容量制限を設定します。個別のブローカーに異なる容量制限を設定するには、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「[容量設定ファイルの作成](#)」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（[自動作成されるトピックを参照](#)）には、**COMPACT** ではなく **DELETE** のログクリーンアップポリシーが設定されることが重要です。それ以外の場合は、Cruise Control で必要なレコードが削除されることがあります。

「[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適切に設定されるようになります。

- `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になっているが、Kafka クラスターで有効になっている場合、`__CruiseControlMetrics` トピックは引き続きブローカーによって自動的に作成されます。このような場合には、`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。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
__CruiseControlMetrics --describe
```

2. トピック設定でログクリーンアップポリシーを変更します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
__CruiseControlMetrics --alter --add-config cleanup.policy=delete
```

Cruise Control Metrics Reporter と 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、`__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。

変更を有効にするには、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 は最適化プロポーザルを生成して、指定の最適化ゴールを基にして Kafka クラスターをリバランスします。

dryrun パラメーターが入力され、**false** に設定されていない限り、最適化プロポーザルはドライランとして生成されます。「ドライラン」モードでは、Cruise Control は最適化プロポーザルと予測された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しません。

最適化プロポーザルで返される情報を分析して、プロポーザルを開始するかどうかを判断できます。

以下は、**/rebalance** エンドポイントへの要求のキーパラメーターです。使用できるすべてのパラメーターの詳細は、Cruise Control Wiki の [「REST APIs」](#) を参照してください。

dryrun

type: boolean、default: true

最適化プロポーザルのみを生成するか(**true**)、最適化プロポーザルを生成してクラスターリバランスを行うか、Cruise Control に通知します。**false**

dryrun=true (デフォルト) の場合、**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ができます。これには、最適化プロポーザルの適用前および後の各 Kafka ブローカーの負荷のメトリクスと、before と after 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: list of strings、default: configured **default.goals** list

最適化プロポーザルの準備に使用するユーザー提供の最適化ゴールのリスト。ゴールが指定されない場合、**cruisecontrol.properties** ファイルの設定済みの **default.goals** リストが使用されます。

skip_hard_goals_check

type: boolean、default: **false**

デフォルトでは、Cruise Control はユーザー提供の最適化ゴール (**goals** パラメーター) に設定済みのハードゴール(**hard.goals**)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場合は、リクエストが失敗します。

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、default: **false**

Cruise Control サーバーによって返される応答の型を制御します。指定されない場合、または **false** に設定されている場合、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抽出する場合は、**json=true** を設定します。これにより、jq などのツールやスクリプトやプログラムで解析できる JSON 形式のテキストが返されます。

verbose

type: boolean、default: **false**

Cruise Control サーバーによって返される応答の詳細レベルを制御します。**dryrun=true** と併用することができます。

前提条件

- Kafka および ZooKeeper が稼働している必要があります。
- Cruise Control が稼働している必要があります。

手順

1. コンソールに対してフォーマットされた「dry run」最適化プロポーザルを生成するには、POST 要求を **/rebalance** エンドポイントに送信します。

- 設定した **default.goals** を使用するには、以下を実行します。

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

キャッシュされた最適化プロポーザル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。数分待機した後に、リクエストを再送信します。

- 設定された **default.goals** の代わりにユーザー提供の最適化ゴールを指定するには、**goals** パラメーターにゴールを1つ以上指定します。

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?goals=RackAwareGoal,ReplicaCapacityGoal'
```

指定されたゴールを満たす場合、キャッシュされた最適化プロポーザルは即座に返されます。それ以外の場合、指定の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。算出にかかる時間が長くなります。この挙動を強制するには、**ignore_proposal_cache=true** パラメーターをリクエストに追加します。

- 設定済みのすべてのハードゴールを含まないユーザー提供の最適化ゴールを指定するには、**skip_hard_goal_check=true** パラメーターをリクエストに追加します。

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?goals=RackAwareGoal,ReplicaCapacityGoal,ReplicaDistributionGoal&skip_hard_goal_check=true'
```

2. 応答に含まれる最適化プロポーザルを確認します。プロパティーは、保留中のクラスターリバランス操作を記述します。

このプロポーザルには、提案された最適化の概要、その後の各デフォルト最適化ゴールのサマリー、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。

以下の情報に注意してください。

- **Cluster load after rebalance** の概要要件を満たす場合は、高レベルの概要を使用して、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** ブローカー間のネットワーク全体に移動するデータ量を示します。値が高いほど、リバランス中の Kafka クラスターのパフォーマンスに影響する可能性があります。
- **n intra-broker replica (y MB) moves** ブローカー自体（ディスク）内で移動するデータ量を示します。値が大きいほど、個別のブローカーのパフォーマンスに影響する可能性があります（ただし **n inter-broker replica (y MB) moves** 未満です）。
- リーダーシップの移動の数。これは、リバランス中のクラスターのパフォーマンスに対する影響を与えます。

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で続行されます。最新のキャッシュされた最適化プロポーザルの準備ができない場合や、ユーザー提供の最適化ゴールが **ignore_proposal_cache=true** で指定された場合に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指定されるリクエストに固有の識別子を書き留めておきます。

curl を使用して応答を取得するには、詳細(**-v**)オプションを指定します。

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

以下はヘッダーの例です。

```
* 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)
```

タイムアウト内に最適化プロポーザルが準備状態でない場合は、POST リクエストを再送信できます。これには、ヘッダーの元のリクエストの **User-Task-ID** が含まれます。

```
curl -v -X POST -H 'User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201' 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

次のステップ

「クラスターリバランスの開始」

14.10. クラスターリバランスの開始

最適化プロポーザルに問題がなければ、Cruise Control にクラスターのリバランスを開始して、プロポーザルで説明されているようにパーティションの再割り当てを開始することができます。

最適化プロポーザルの生成とクラスターのリバランスの開始の間には、可能な限り少し時間がかかります。元の最適化プロポーザルを生成以降に時間が経過した場合、クラスターの状態が変更される可能性があります。そのため、開始するクラスターリバランスは確認したものとは異なる可能性があります。不明な場合は、まず新しい最適化プロポーザルを生成します。

一度に使用できるクラスターリバランスは1つのみです。ステータスが「Active」とされています。

前提条件

- Cruise Control から [最適化プロポーザルを生成済み](#) である必要があります。

手順

1. 最近生成された最適化プロポーザルを実行するには、**dryrun=false** パラメーターで **/rebalance** エンドポイントに POST リクエストを送信します。

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?dryrun=false'
```

Cruise Control はクラスターのリバランスを開始し、最適化プロポーザルを返します。

2. 最適化プロポーザルで要約された変更を確認します。変更が想定されていない場合は、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クラスターリバランスタスクをすべて表示するには、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** パラメーターとタスクIDを指定します。

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks?user_task_ids=c459316f-9eb5-482f-9d2d-97b5a4cd294d'
```

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、完了したパーティションの再割り当ては適用済みです。そのため、Kafka クラスターの状態は、リバランス操作の開始前とは異なります。さらなるリバランスが必要な場合は、新しい最適化プロポーザルを生成してください。



注記

中間(停止) 状態の Kafka クラスターのパフォーマンスは、初期状態の場合よりも悪くなる可能性があります。

前提条件

- クラスターリバランスは、「Active」のステータスで表される) 進行中です。

手順

- POST リクエストを **/stop_proposal_execution** エンドポイントに送信します。

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/stop_proposal_execution'
```

その他のリソース

- [「最適化プロポーザルの生成」](#)

第15章 分散トレーシング

分散トレーシングを使用すると、分散システムのアプリケーション間で実行されるトランザクションの進捗を追跡できます。マイクロサービスのアーキテクチャーでは、トレーシングはサービス間のトランザクションの進捗を追跡します。トレースデータは、アプリケーションのパフォーマンスを監視し、ターゲットシステムおよびエンドユーザーアプリケーションの問題を調べるのに役立ちます。

AMQ Streams on Red Hat Enterprise Linux では、トレーシングによってメッセージのエンドツーエンドで、ソースシステムから Kafka、さらに Kafka からターゲットシステムおよびアプリケーションへのメッセージの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

トレーシングのサポートは、以下のクライアントおよびコンポーネントに対して提供されます。

Kafka クライアント :

- Kafka プロデューサーおよびコンシューマー
- Kafka Streams API アプリケーション

Kafka コンポーネント :

- Kafka Connect
- Kafka Bridge
- MirrorMaker
- MirrorMaker 2.0

トレースを有効にするには、4 つのハイレベルなタスクを実行します。

1. Jaeger トレーサーを有効にします。
2. インターセプターを有効にします。
 - Kafka [クライアントの場合は、OpenTracing Apache Kafka Client Instrumentation](#) ライブラリー (AMQ Streams に含まれる) を使用してアプリケーションコードをインストルメント化します。
 - Kafka コンポーネントでは、各コンポーネントに設定プロパティを設定します。
3. [トレーシング環境変数を設定します](#)。
4. クライアントまたはコンポーネントをデプロイします。

インストルメント化されると、クライアントはトレースデータを生成します。たとえば、メッセージを生成したり、ログへのオフセットを書き込む場合などです。

トレースは、サンプリングストラテジーに従いサンプル化され、Jaeger ユーザーインターフェースで可視化されます。



注記

トレーシングはKafka ブローカーではサポートされません。

AMQ Streams 以外のアプリケーションおよびシステムにトレーシングを設定する方法については、本章の対象外となります。この件についての詳細は、[OpenTracing ドキュメント](#)を参照し、「inject and extrac」を検索してください。

手順の概要

AMQ Streams のトレーシングを設定するには、以下の手順を順番に行います。

1. クライアントのトレーシングを設定します。
 - a. [Kafka クライアントのJaeger トレーサーを初期化](#)します。
 - b. [プロデューサーおよびコンシューマーをトレーシング用にインストルメント化](#)します。
 - c. [Kafka Streams アプリケーションをトレーシング用にインストルメント化](#)します。
2. MirrorMaker、MirrorMaker 2.0、およびKafka Connect のトレースを設定します。
 - a. [MirrorMaker のトレースを有効に](#)します。
 - b. [MirrorMaker 2.0 のトレースを有効に](#)します。
 - c. [Kafka Connect のトレースの有効化](#)
3. [Kafka Bridge のトレースの有効化](#)

前提条件

- Jaeger バックエンドコンポーネントがホストオペレーティングシステムにデプロイされます。デプロイメント手順の詳細は、[Jaeger デプロイメントのドキュメント](#)を参照してください。

15.1. OPENTRACING および JAEGER の概要

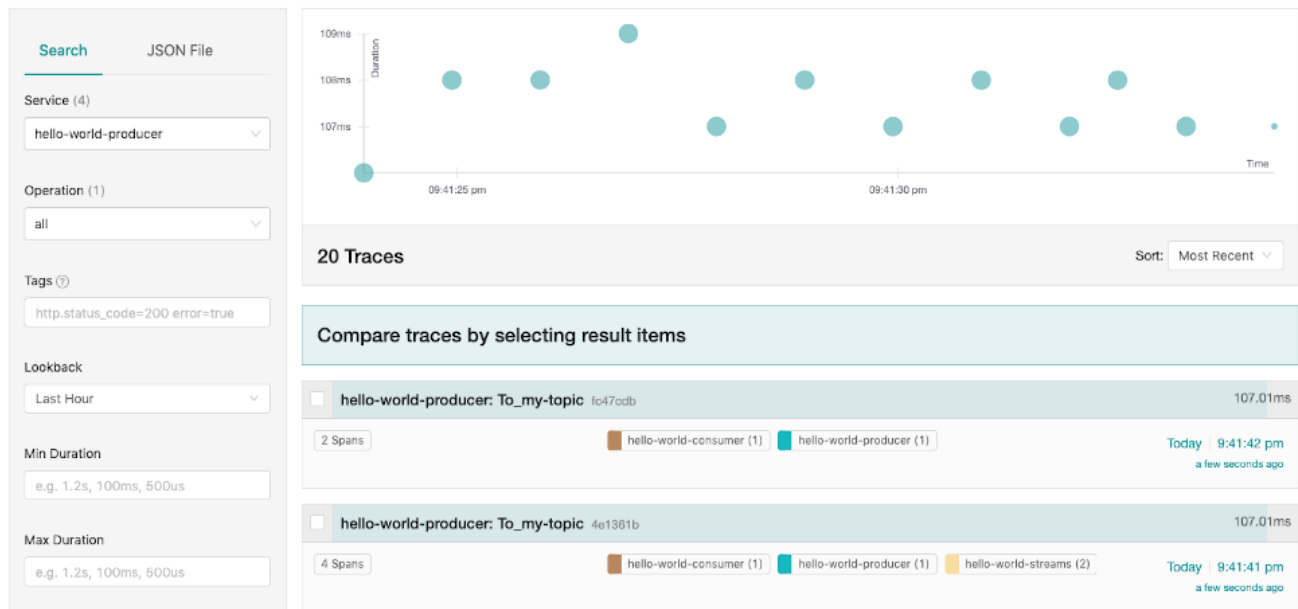
AMQ Streams ではOpenTracing およびJaeger プロジェクトが使用されます。

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 で構成されます。

Jaeger はマイクロサービスベースの分散システムのトレーシングシステムです。

- Jaeger はOpenTracing API を実装し、インストルメント化のクライアントライブラリーを提供します。
- Jaeger ユーザーインターフェースを使用すると、トレースデータをクエリー、フィルター、および分析できます。



その他のリソース

- [OpenTracing](#)
- [Jaeger](#)

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>1.1.0.redhat-00002</version>
</dependency>
```

2. [トレーシング環境変数](#)を使用して Jaeger トレーサーの設定を定義します。
3. 2. で定義した環境変数から、Jaeger トレーサーを作成します。

```
Tracer tracer = Configuration.fromEnv().getTracer();
```



注記

別の Jaeger トレーサーの初期化方法については、[Java OpenTracing ライブラリー](#)のドキュメントを参照してください。

4. Jaeger トレーサーをグローバルトレーサーとして登録します。

```
GlobalTracer.register(tracer);
```

これで、Jaeger トレーサーはクライアントアプリケーションが使用できるように初期化されました。

15.2.2. トレーシングのための Kafka プロデューサーおよびコンシューマーのインストール化

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ール化します。

手順

各プロデューサーおよびコンシューマーアプリケーションのアプリケーションコードで以下を行います。

1. OpenTracing の Maven 依存関係を、プロデューサーまたはコンシューマーの **pom.xml** ファイルに追加します。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-client</artifactId>
  <version>0.1.15.redhat-00001</version>
</dependency>
```

2. Decorator パターンまたは Interceptor のいずれかを使用して、クライアントアプリケーションコードをインストール化します。

- Decorator パターンを使用する場合は以下を行います。

```
// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer:
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer,
  tracer);

// Send:
tracingProducer.send(...);

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer:
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer,
  tracer);
```

```
// Subscribe:
tracingConsumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);

// Retrieve SpanContext from polled record (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

- インターセプターを使用する場合は以下を使用します。

```
// Register the tracer with GlobalTracer:
GlobalTracer.register(tracer);

// Add the TracingProducerInterceptor to the sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send:
producer.send(...);

// Add the TracingConsumerInterceptor to the consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Subscribe:
consumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// Retrieve the SpanContext from a polled message (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

Decorator パターンのカスタムスパン名

スパン は Jaeger の論理作業単位で、操作名、開始時間、および期間が含まれます。

Decorator パターンを使用してプロデューサーおよびコンシューマーの各アプリケーションをインストルメント化する場合、**TracingKafkaProducer** および **TracingKafkaConsumer** オブジェクトの作成時に **BiFunction** オブジェクトを追加の引数として渡すと、カスタムスパン名を定義できます。OpenTracing の Apache Kafka Client Instrumentation ライブラリーには、複数の組み込みスパン名が含まれています。

例: カスタムスパン名を使用した Decorator パターンでのクライアントアプリケーションコードのインストルメント化

-

```
// Create a BiFunction for the KafkaProducer that operates on (String operationName,
// ProducerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ProducerRecord, String> producerSpanNameProvider =
    (operationName, producerRecord) -> "CUSTOM_PRODUCER_NAME";

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>(producer,
    tracer,
    producerSpanNameProvider);

// Spans created by the tracingProducer will now have "CUSTOM_PRODUCER_NAME" as the span
// name.

// Create a BiFunction for the KafkaConsumer that operates on (String operationName,
// ConsumerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ConsumerRecord, String> consumerSpanNameProvider =
    (operationName, consumerRecord) -> operationName.toUpperCase();

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer, passing in the consumerSpanNameProvider
// BiFunction:

TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
    (consumer,
    tracer,
    consumerSpanNameProvider);

// Spans created by the tracingConsumer will have the operation name as the span name, in upper-
// case.
// "receive" -> "RECEIVE"
```

ビルトインスパン名

カスタムスパン名を定義するとき、**ClientSpanNameProvider** クラスで以下の **BiFunctions** を使用できます。**spanNameProvider** の指定がない場合は、**CONSUMER_OPERATION_NAME** および **PRODUCER_OPERATION_NAME** が使用されます。

表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」、プロデューサーには「send」を返します。

BiFunction	説明
CONSUMER_PREFIXED_OPERATION_NAME (String prefix), PRODUCER_PREFIXED_OPERATION_NAME (String prefix)	prefix および operationName の文字列連結を返します。
CONSUMER_TOPIC , PRODUCER_TOPIC	メッセージの送信先または送信元となったトピックの名前を(record.topic()) 形式で返します。
PREFIXED_CONSUMER_TOPIC (String prefix), PREFIXED_PRODUCER_TOPIC (String prefix)	prefix およびトピック名の文字列連結を(record.topic()) 形式で返します。
CONSUMER_OPERATION_NAME_TOPIC , PRODUCER_OPERATION_NAME_TOPIC	操作名およびトピック名を " operationName - record.topic() " 形式で返します。
CONSUMER_PREFIXED_OPERATION_NAME_TOPIC (String prefix), PRODUCER_PREFIXED_OPERATION_NAME_TOPIC (String prefix)	prefix および " operationName - record.topic() " の文字列連結を返します。

15.2.3. Kafka Streams アプリケーションのトレーシングのインストルメント化

サプライヤーインターフェースを使用して、分散トレーシングのために 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 アプリケーションの **pom.xml** ファイルに追加します。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-streams</artifactId>
  <version>0.1.15.redhat-00001</version>
</dependency>
```

2. **TracingKafkaClientSupplier** サプライヤーインターフェースのインスタンスを作成します。

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

3. サプライヤーインターフェースを **KafkaStreams** に提供します。

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

15.3. MIRRORMAKER およびKAFKA CONNECT のトレース設定

ここでは、分散トレーシングに 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の設定パラメーターとして渡すことで、MirrorMaker の分散トレーシングを有効にします。

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 トレーサーを設定および有効にします。
2. **/opt/kafka/config/consumer.properties** ファイルを編集します。
以下の Interceptor 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 Interceptor 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の設定ファイルで MirrorMaker をパラメーターとして起動します。

```
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で Interceptor プロパティを定義して、MirrorMaker 2.0 の分散トレーシングを有効にします。

メッセージは Kafka クラスター間でトレースされます。トレースデータは、MirrorMaker 2.0 コンポーネントを出入りするメッセージを記録します。

手順

1. Jaeger トレーサーを設定および有効にします。
2. MirrorMaker 2.0 設定プロパティファイル **./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 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、メッセージはソースおよび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の同期」](#) の手順に従って MirrorMaker 2.0 を起動します。

その他のリソース

- [9章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 の分散トレーシングを有効にします。

Kafka Connect により生成および消費されるメッセージのみがトレーシングされます。Kafka Connect と外部システム間で送信されるメッセージをトレーシングするには、これらのシステムのコネクターでトレーシングを設定する必要があります。

手順

1. Jaeger トレーサーを設定および有効にします。
2. 関連する Kafka Connect 設定ファイルを編集します。
 - 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す。

その他のリソース

- [「トレーシングの環境変数」](#)
- [「スタンドアロンモードでの Kafka Connect の実行」](#)

- [「分散 Kafka Connect の実行」](#)

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、分散トレーシング用に設定された 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 はメッセージを HTTP クライアントに送信し、HTTP クライアントからメッセージを消費します。
- HTTP クライアントは HTTP リクエストを送信し、Kafka Bridge 経由でメッセージを送受信します。

エンドツーエンドのトレーシングを設定するために、HTTP クライアントでトレーシングを設定する必要があります。

手順

1. Kafka Bridge インストールディレクトリーの **config/application.properties** ファイルを編集します。
以下の行からコードコメントを削除します。

```
bridge.tracing=jaeger
```

2. 設定ファイルを作成します。
3. 設定プロパティをパラメーターとして使用して、**bin/kafka_bridge_run.sh** スクリプトを実行します。

```
cd kafka-bridge-0.xy.x.redhat-0000x  
./bin/kafka_bridge_run.sh --config-file=config/application.properties
```

Kafka Bridge の内部コンシューマーおよびプロデューサーのインターセプターが有効になります。

その他のリソース

- [「Kafka Bridge プロパティの設定」](#)

15.5. トレーシングの環境変数

Kafka クライアントおよびコンポーネント用に Jaeger トレーサーを設定する際に、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 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カンマ区切りの形式リスト。デフォルトは標準の Jaeger 形式です。有効な値は jaeger および b3 です。
JAEGER_REPORTER_LOG_SPANS	不要	レポーターがスパンも記録する必要があるかどうかを示します。
JAEGER_REPORTER_MAX_QUEUE_SIZE	不要	レポーターの最大キューサイズ。
JAEGER_REPORTER_FLUSH_INTERVAL	不要	レポーターのフラッシュ間隔 (ミリ秒単位)。Jaeger レポーターがスパンバッチをフラッシュする頻度を定義します。

プロパティ	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>詳細は、Jaeger 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カンマ区切りリスト。</p> <p>この値に <code>\${envVarName:default}</code> 形式を使用して環境変数を参照することもできます。<code>:default</code> 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

[Kafka Exporter](#) は、Apache Kafka ブローカーおよびクライアントの監視を強化するオープンソースプロジェクトです。

Kafka Exporter は、Kafka クラスターとのデプロイメントを実現するために AMQ Streams で提供され、オフセット、コンシューマーグループ、コンシューマーラグ、およびトピックに関連する Kafka ブローカーから追加のメトリクスデータを抽出します。

一例として、メトリクスデータを使用すると、低速なコンシューマーの識別に役立ちます。

ラグデータは Prometheus メトリクスとして公開され、解析のために Grafana で使用できます。

ビルトイン Kafka メトリクスを監視するために Prometheus および Grafana をすでに使用している場合、Kafka Exporter Prometheus エンドポイントをスクレイプするように Prometheus を設定することもできます。

その他のリソース

Kafka は JMX 経由でメトリクスを公開し、その後に Prometheus メトリクスとしてエクスポートできます。

- [7章JMX を使用したクラスターのモニタリング](#)

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。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

トピックが複製の数が最低数未満であり、ブローカーに十分なパーティションを複製し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。

その他のリソース

アラートルールの設定についての詳細は、Prometheus ドキュメントの「[Configuration](#)」を参照してください。

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
kafka_brokers	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
kafka_topic_partitions	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 は、AMQ Streams のインストールに使用されるダウンロードアーカイブが提供されません。

これを実行して、Grafana ダッシュボードで表示するために Prometheus メトリクスを公開できます。

前提条件

- AMQ Streams がホストにインストールされている。

この手順では、Grafana ユーザーインターフェースにアクセスでき、Prometheus がデータソースとしてデプロイされ、追加されていることを前提とします。

手順

- 適切な設定パラメーター値を使用して Kafka Exporter スクリプトを実行します。

```
./bin/kafka_exporter --kafka.server=<kafka-bootstrap-address>:9092 --kafka.version=2.8.0
--<my-other-parameters>
```

パラメーターには、**--kafka.server** など、二重のハイフン規則が必要です。

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。	.*(all)
topic.filter	メトリクスに含まれるトピックを指定する正規表現。	.*(all)
sasl.<parameter>	ユーザー名とパスワードを使用して SASL/PLAIN 認証を使用して Kafka クラスターで有効にし、接続するパラメーター。	false
tls.<parameter>	TLS 認証を使用して Kafka クラスターへの接続を可能にするパラメーター。オプションの証明書とキー。	false
web.listen-address	メトリクスを公開するポートアドレス。	:9308
web.telemetry-path	公開されるメトリクスのパス。	/metrics
log.level	指定の重大度 (debug、info、warn、error、fatal) 以上でメッセージをログに記録するためのログ設定。	info
log.enable-sarama	Sarama ロギングを有効にするブール値 (Kafka Exporter によって使用される Go クライアントライブラリー)。	false

オプション	説明	デフォルト
legacy.partitions	メトリクスが非アクティブなトピックパーティションやアクティブなパーティションからフェッチできるようにするブール値。非アクティブなパーティションのメトリクスを返す場合は、 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。

2. Kafka Exporter メトリクスを監視するように Prometheus を設定します。
Prometheus の設定に関する詳細は、Prometheus [のドキュメント](#) を参照してください。
3. Grafana を有効にして、Prometheus によって公開される Kafka Exporter メトリクスデータを表示します。
詳細は、「[Grafana での Kafka Exporter メトリクスの変更](#)」を参照してください。

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。

その他のリソース

- [Kafka Exporter のダッシュボードの例](#)
- [Grafana 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。

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必要があります。手順は「[2章 スタートガイド](#)」を参照してください。
- 『[AMQ Streams 1.8 on Red Hat Enterprise Linux リリースノート](#)』に記載されているアップグレードの変更について理解している必要があります。

17.2. アップグレードプロセス

AMQ Streams のアップグレードは2段階のプロセスで行います。ダウンタイムなしでブローカーとクライアントをアップグレードするには、以下の順序でアップグレード手順を必ず完了してください。

1. 最新の AMQ Streams バージョンにアップグレードします。
 - [AMQ Streams 1.8 へのアップグレード](#)
2. すべての Kafka ブローカーおよびクライアントアプリケーションを最新の Kafka バージョンにアップグレードします。
 - [Kafka のアップグレード](#)

17.3. KAFKA バージョン

Kafka のログメッセージ形式バージョンとブローカー間のプロトコルバージョンは、それぞれメッセージに追加されるログ形式バージョンとクラスターで使用される Kafka プロトコルのバージョンを指定します。正しいバージョンが使用されるようにするため、アップグレードプロセスでは、既存の Kafka ブローカーの設定変更と、クライアントアプリケーション(コンシューマーおよびプロデューサー)のコード変更が行われます。

以下の表は、Kafka バージョンの違いを示しています。

Kafka のバージョン	Interbroker プロトコルのバージョン	ログメッセージ形式のバージョン	ZooKeeper のバージョン
2.8.0	2.8	2.8	3.5.9
2.7.0	2.7	2.7	3.5.8

ブローカー間のプロトコルバージョン

Kafka では、ブローカー間の通信に使用されるネットワークプロトコルはブローカー間プロトコル

(Inter-broker protocol) と呼ばれます。Kafka の各バージョンには、互換性のあるバージョンのブローカー間プロトコルがあります。上記の表が示すように、プロトコルのマイナーバージョンは、通常 Kafka のマイナーバージョンと一致するように番号が増加されます。

ブローカー間プロトコルのバージョンは、**Kafka** リソースでクラスター全体に設定されます。これを変更するには、**Kafka.spec.kafka.config** の **inter.broker.protocol.version** プロパティを編集します。

ログメッセージ形式のバージョン

プロデューサーが Kafka ブローカーにメッセージを送信すると、特定の形式を使用してメッセージがエンコードされます。この形式は Kafka のリリース間で変更される可能性があるため、メッセージにはエンコードに使用された形式のバージョンが指定されます。ブローカーがメッセージをログに追加する前に、メッセージを新しい形式バージョンから特定の旧形式バージョンに変換するように、Kafka ブローカーを設定できます。

Kafka には、メッセージ形式のバージョンを設定する 2 通りの方法があります。

- **message.format.version** プロパティはトピックに設定されます。
- **log.message.format.version** プロパティは Kafka ブローカーに設定されます。

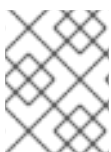
トピックの **message.format.version** のデフォルト値は、Kafka ブローカーに設定される **log.message.format.version** によって定義されます。トピックの **message.format.version** は、トピック設定を編集すると手動で設定できます。

本セクションのアップグレード作業では、メッセージ形式のバージョンが **log.message.format.version** によって定義されることを前提としています。

17.4. AMQ STREAMS 1.8 へのアップグレード

このセクションでは、AMQ Streams 1.8 を使用するようにデプロイメントをアップグレードする手順について説明します。

AMQ Streams によって管理される Kafka クラスターの可用性は、アップグレード操作による影響を受けません。



注記

特定バージョンの AMQ Streams へのアップグレード方法については、そのバージョンをサポートするドキュメントを参照してください。

17.4.1. Kafka ブローカーおよび ZooKeeper のアップグレード

この手順では、ホストマシンで Kafka ブローカーおよび ZooKeeper をアップグレードし、最新バージョンの AMQ Streams を使用する方法を説明します。

前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと、1 度に 1 つずつ以下を実行します。

1. [カスタマーポータルから AMQ Streams アーカイブをダウンロード](#)します。



注記

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、ZooKeeper およびホストで実行されている Kafka ブローカーを停止します。

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

4. 既存のインストールから、**libs**、**bin**、および **docs** ディレクトリーを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin /opt/kafka/docs
```

5. 一時ディレクトリーから **libs**、**bin**、および **docs** ディレクトリーをコピーします。

```
cp -r /tmp/kafka/kafka_y.y-x.x.x/libs /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/docs /opt/kafka/
```

6. 一時ディレクトリーを削除します。

```
rm -r /tmp/kafka
```

7. テキストエディターで、一般的に **/opt/kafka/config/** ディレクトリーに保存されるブローカープロパティファイルを開きます。

8. **inter.broker.protocol.version** および **log.message.format.version** プロパティが現行バージョンに設定されていることを確認します。

```
inter.broker.protocol.version=2.7
log.message.format.version=2.7
```

inter.broker.protocol.version を変更しないと、ブローカーはアップグレード中も相互に通信を継続できます。

プロパティが設定されていない場合は、現行バージョンで追加します。

9. 更新された ZooKeeper および Kafka ブローカーを再起動します。

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

Kafka ブローカーおよび Zookeeper は、最新の Kafka バージョンのバイナリーの使用を開始します。

10. 再起動した Kafka ブローカーが、パーティションレプリカで取得されたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリカが同期し直すようにします。手順は「[トピックの一覧表示および記述](#)」を参照してください。
11. 「[Kafka のアップグレード](#)」の説明に従って、Kafka をアップグレードする手順を実行します。

17.4.2. Kafka Connect のアップグレード

この手順では、ホストマシンで Kafka Connect クラスターをアップグレードする方法を説明します。

Kafka Connect はクライアントアプリケーションで、クライアントのアップグレードに選択したストラテジーに含める必要があります。詳細は、「[クライアントをアップグレードするストラテジー](#)」を参照してください。

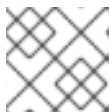
前提条件

- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。
- Kafka Connect は起動していません。

手順

AMQ Streams クラスターの各 Kafka ブローカーと、1 度に 1 つずつ以下を実行します。

1. [カスタマーポータルから AMQ Streams アーカイブをダウンロード](#)します。



注記

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、ホスト上で Kafka ブローカーおよび ZooKeeper を停止します。

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

4. 既存のインストールから、**libs**、**bin**、および **docs** ディレクトリを削除します。

```
rm -rf /opt/kafka/libs /opt/kafka/bin /opt/kafka/docs
```

5. 一時ディレクトリから **libs**、**bin**、および **docs** ディレクトリをコピーします。

```
cp -r /tmp/kafka/kafka_y.y-x.x.x/libs /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/bin /opt/kafka/
cp -r /tmp/kafka/kafka_y.y-x.x.x/docs /opt/kafka/
```

6. 一時ディレクトリを削除します。

```
rm -r /tmp/kafka
```

7. Kafka Connect をスタンドアロンまたは分散モードのいずれかで起動します。

- 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と 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 ...]
```

- 分散モードを開始するには、すべての Kafka Connect ノードで **/opt/kafka/config/connect-distributed.properties** 設定ファイルを使用して Kafka Connect ワーカーを起動します。

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

8. Kafka Connect が稼働していることを確認します。

- スタンドアロンモードでは、以下を行います。

```
jcmd | grep ConnectStandalone
```

- Distributed（分散）：

```
jcmd | grep ConnectDistributed
```

9. Kafka Connect がデータを想定どおりに生成し、消費していることを確認します。

その他のリソース

- [スタンドアロンモードでの Kafka Connect の実行](#)
- [分散 Kafka Connect の実行](#)
- [クライアントをアップグレードするストラテジー](#)

17.5. KAFKA のアップグレード

最新バージョンの AMQ Streams を使用するようにバイナリーをアップグレードした後、ブローカーおよびクライアントをアップグレードして、サポートされる上位バージョンの Kafka を使用することができます。

正しい順序で手順に従うようにしてください。

1. 「新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード」
2. 「クライアントアプリケーションの新しい Kafka バージョンへのアップグレード」
3. 「新しいメッセージ形式のバージョンを使用するように Kafka ブローカーのアップグレード」

Kafka のアップグレードに従い、Kafka コンシューマーをアップグレードして Incremental Cooperative Rebalance プロトコルを使用することができます。

1. [「コンシューマーおよびKafka Streams アプリケーションのCooperative Rebalancing へのアップグレード」](#)

17.5.1. 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード

新しいブローカー間プロトコルバージョンを使用するように、すべての Kafka ブローカーを手動で設定および再起動します。これらの手順を完了すると、新しいブローカー間プロトコルバージョンを使用して Kafka ブローカー間でデータが送信されます。

受信されるメッセージは、それ以前のメッセージ形式のバージョンのメッセージログに追加されます。



警告

この手順の完了後、AMQ Streams のダウングレードを行うことはできません。

前提条件

- ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.8 にアップグレードしている。
- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと、1 度に 1 つずつ以下を実行します。

1. テキストエディターで、更新する Kafka ブローカーのブローカープロパティファイルを開きます。ブローカープロパティファイルは通常 **/opt/kafka/config/** ディレクトリーに保存されます。

2. **inter.broker.protocol.version** を **2.8** に設定します。

```
inter.broker.protocol.version=2.8
```

3. コマンドラインで、変更した Kafka ブローカーを停止します。

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

4. 変更した Kafka ブローカーを再起動します。

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

5. 再起動した Kafka ブローカーが、パーティションレプリカで取得されたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリカが同期し直すようにします。[手順は「トピックの一覧表示および記述」を参照してください。](#)

17.5.2. クライアントをアップグレードするストラテジー

クライアントアプリケーション (Kafka Connect コネクターを含む) のアップグレードに適切な方法は、特定の状況によって異なります。

消費するアプリケーションは、そのアプリケーションが理解するメッセージ形式のメッセージを受信する必要があります。その状態であることを、以下のいずれかの方法で確認できます。

- プロデューサーをアップグレードする前に、トピックのすべてのコンシューマーをアップグレードする。
- ブローカーでメッセージをダウンコンバートする。

ブローカーのダウンコンバートを使用すると、ブローカーに余分な負荷が加わるので、すべてのトピックで長期にわたりダウンコンバートに頼るのは最適な方法ではありません。ブローカーの実行を最適化するには、ブローカーがメッセージを一切ダウンコンバートしないようにしてください。

ブローカーのダウンコンバートは2通りの方法で設定できます。

- トピックレベルの **message.format.version** では単一のトピックが設定されます。
- ブローカーレベルの **log.message.format.version** は、トピックレベルの **message.format.version** が設定されていないトピックのデフォルトです。

新バージョンの形式でトピックにパブリッシュされるメッセージは、コンシューマーによって認識されます。これは、メッセージがコンシューマーに送信されるときでなく、ブローカーがプロデューサーからメッセージを受信するときに、ブローカーがダウンコンバートを実行するからです。

クライアントのアップグレードに使用できるストラテジーは複数あります。

コンシューマーを最初にアップグレード

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. ブローカーレベルの **log.message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。
このストラテジーは分かりやすく、ブローカーのダウンコンバートの発生をすべて防ぎます。ただし、所属組織内のすべてのコンシューマーを整然とアップグレードできることが前提になります。また、コンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。さらにリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加され、以前のコンシューマーバージョンに戻せなくなる場合があります。

トピック単位でコンシューマーを最初にアップグレード

トピックごとに以下を実行します。

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. トピックレベルの **message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。
このストラテジーではブローカーのダウンコンバートがすべて回避され、トピックごとにアップグレードできます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。ここでもリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加される可能性があります。

トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり

トピックごとに以下を実行します。

1. トピックレベルの **message.format.version** を、旧バージョンに変更します(または、デフォルトがブローカーレベルの **log.message.format.version** のトピックを利用します)。
2. コンシューマーおよびプロデューサーとして機能するアプリケーションをすべてアップグレードします。
3. アップグレードしたアプリケーションが正しく機能することを確認します。
4. トピックレベルの **message.format.version** を新バージョンに変更します。
このストラテジーにはブローカーのダウンコンバートが必要ですが、ダウンコンバートは一度に1つのトピック(またはトピックの小さなグループ)のみに必要になるので、ブローカーへの負荷は最小限に抑えられます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションにも通用します。この方法により、新しいメッセージ形式バージョンを使用する前に、アップグレードされたプロデューサーとコンシューマーが正しく機能することが保証されます。

この方法の主な欠点は、多くのトピックやアプリケーションが含まれるクラスターでの管理が複雑になる場合があることです。

クライアントアプリケーションをアップグレードするストラテジーは他にもあります。



注記

複数のストラテジーを適用することもできます。たとえば、最初のいくつかのアプリケーションとトピックに、「トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり」のストラテジーを適用します。これが問題なく適用されたら、より効率的な別のストラテジーの使用を検討できます。

17.5.3. クライアントアプリケーションの新しいKafka バージョンへのアップグレード

この手順では、クライアントアプリケーションを AMQ Streams 1.8. に使用する Kafka バージョンにアップグレードできるアプローチを説明しています。

[この手順は、クライアントをアップグレードするストラテジーについて「ダウンコンバート」のアプローチで最初に「トピックごとのコンシューマー」に基づいています。](#)

クライアントアプリケーションには、プロデューサー、コンシューマー、Kafka Connect、Kafka Streams アプリケーション、および MirrorMaker が含まれます。

前提条件

- [ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.8 にアップグレードしている。](#)
- [新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーを設定している。](#)
- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

トピックごとに以下を実行します。

1. コマンドラインで、**message.format.version** 設定オプションを **2.7** に設定します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
<TopicName> --alter --add-config message.format.version=2.7
```

2. トピックのコンシューマーおよびプロデューサーをすべてアップグレードします。
3. 任意で、Kafka 2.4.0 で追加された Incremental Cooperative Rebalance プロトコルを使用するようにコンシューマーおよび Kafka Streams アプリケーションをアップグレードするには、[「コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード」](#) を参照してください。
4. アップグレードしたアプリケーションが正しく機能することを確認します。
5. トピックの **message.format.version** 設定オプションを **2.8** に変更します。

```
bin/kafka-configs.sh --bootstrap-server <BrokerAddress> --entity-type topics --entity-name
<TopicName> --alter --add-config message.format.version=2.8
```

その他のリソース

- [クライアントをアップグレードする戦略](#)

17.5.4. 新しいメッセージ形式のバージョンを使用するように Kafka ブローカーのアップグレード

クライアントアプリケーションがアップグレードされたら、新しいメッセージ形式のバージョンを使用するように Kafka ブローカーを更新できます。

AMQ Streams 1.8 に必要な Kafka バージョンを使用するように、クライアントアプリケーションをアップグレードしたときにトピック設定を変更しなかった場合、Kafka ブローカーはメッセージを以前のメッセージ形式バージョンに変換して、パフォーマンスが低下する可能性があります。したがって、できるだけ早期に新しいメッセージ形式バージョンを使用するように、すべての Kafka ブローカーを更新することが重要になります。



注記

Kafka ブローカーを1つずつ更新し、再起動します。変更したブローカーを再起動する前に、以前に設定したブローカーを停止して再起動してください。

前提条件

- ZooKeeper バイナリーを更新し、すべての Kafka ブローカーを AMQ Streams 1.8 にアップグレードしている。
- 新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーを設定している。
- **message.format.version** プロパティがトピックレベルで明示的に設定されていないトピックからメッセージを消費するサポート対象のクライアントアプリケーションをアップグレードしている。
- **kafka** ユーザーとして Red Hat Enterprise Linux にログインしている。

手順

AMQ Streams クラスターの各 Kafka ブローカーと、1 度に1 つずつ以下を実行します。

1. テキストエディターで、更新する Kafka ブローカーのブローカープロパティファイルを開きます。ブローカープロパティファイルは通常 `/opt/kafka/config/` ディレクトリーに保存されます。
2. **log.message.format.version** を **2.8** に設定します。

```
log.message.format.version=2.8
```

3. コマンドラインで、この手順の一部として最近編集および再起動した Kafka ブローカーを停止します。この手順の最初の Kafka ブローカーを変更する場合には、手順 4 に進みます。

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

4. 手順 2 で変更した Kafka ブローカーを再起動します。

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

5. 再起動した Kafka ブローカーが、パーティションレプリカで取得されたことを確認します。**kafka-topics.sh** ツールを使用して、ブローカーに含まれるすべてのレプリカが同期し直すようにします。手順は「[トピックの一覧表示および記述](#)」を参照してください。

17.5.5. コンシューマーおよび Kafka Streams アプリケーションの Cooperative Rebalancing へのアップグレード

Kafka コンシューマーおよび Kafka Streams アプリケーションをアップグレードすることで、パーティションの再分散にデフォルトの **Eager Rebalance** プロトコルではなく **Incremental Cooperative Rebalance** プロトコルを使用できます。この新しいプロトコルが Kafka 2.4.0 に追加されました。

コンシューマーは、パーティションの割り当てを Cooperative Rebalance で保持し、クラスターの分散が必要な場合にプロセスの最後でのみ割り当てを取り消します。これにより、コンシューマーグループまたは Kafka Streams アプリケーションが使用不可能になる状態が削減されます。



注記

Incremental Cooperative Rebalance プロトコルへのアップグレードは任意です。Eager Rebalance プロトコルは引き続きサポートされます。

前提条件

- [「AMQ Streams 1.8 へのアップグレード」](#)
- [「新しいブローカー間プロトコルバージョンを使用するように Kafka ブローカーのアップグレード」](#)
- [「クライアントアプリケーションの新しい Kafka バージョンへのアップグレード」](#)

手順

Incremental Cooperative Rebalance プロトコルを使用するように Kafka コンシューマーをアップグレードするには以下を行います。

1. Kafka クライアント **.jar** ファイルを新バージョンに置き換えます。
2. コンシューマー設定で、**partition.assignment.strategy** に **cooperative-sticky** を追加します。たとえば、**range** ストラテジーが設定されている場合は、設定を **range, cooperative-sticky** に変更します。
3. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。
4. コンシューマー設定から前述の **partition.assignment.strategy** を削除して、グループの各コンシューマーを再設定し、**cooperative-sticky** ストラテジーのみを残します。
5. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。

Incremental Cooperative Rebalance プロトコルを使用するように Kafka Streams アプリケーションをアップグレードするには以下を行います。

1. Kafka Streams の **.jar** ファイルを新バージョンに置き換えます。
2. Kafka Streams の設定で、**upgrade.from** 設定パラメーターをアップグレード前の Kafka バージョンに設定します (例: 2.3)。
3. 各ストリームプロセッサ(ノード) を順次再起動します。
4. **upgrade.from** 設定パラメーターを Kafka Streams 設定から削除します。
5. グループ内の各コンシューマーを順次再起動します。

その他のリソース

- Apache Kafka ドキュメントの [「Notable changes in 2.4.0」](#)。

付録A ブローカー設定パラメーター

advertised.host.name

type: string

Default: null

Importance: high

Dynamic update: read-only

非推奨: **advertised.listeners** または **listeners** が設定されていない場合にのみ使用します。代わりに **advertised.listeners** を使用してください。クライアントが使用する ZooKeeper にパブリッシュするホスト名。IaaS 環境では、ブローカーがバインドするインターフェースとは異なる場合があります。これが設定されていない場合、設定されている場合は **host.name** の値を使用します。それ以外の場合は、`java.net.InetAddress.getCanonicalHostName ()` から返された値を使用します。

advertised.listeners

type: string

Default: null

Importance: high

Dynamic update: per-broker

listeners 設定プロパティーと異なる場合に、クライアントが使用する ZooKeeper に公開するリスナー。IaaS 環境では、ブローカーがバインドするインターフェースとは異なる場合があります。これが設定されていない場合、**listeners** の値が使用されます。**listeners** とは異なり、0.0.0.0 メタアドレスを公開することはできません。また、**listeners** とは異なり、このプロパティーには重複するポートを複製するため、1つのリスナーが別のリスナーのアドレスを公開するように設定できます。これは、外部ロードバランサーが使用される場合に役に立ちます。

advertised.port

type: int

Default: null

Importance: high

Dynamic update: read-only

非推奨: **advertised.listeners** または **listeners** が設定されていない場合にのみ使用します。代わりに **advertised.listeners** を使用してください。クライアントが使用する ZooKeeper にパブリッシュするポート。IaaS 環境では、ブローカーがバインドするポートとは異なる場合があります。これが設定されていない場合、ブローカーがバインドする同じポートを公開します。

auto.create.topics.enable

type: boolean

Default: true

Importance: high

Dynamic update: read-only

サーバー上でトピックの自動作成を有効にします。

auto.leader.rebalance.enable

type: boolean

Default: true

Importance: high

Dynamic update: read-only

自動リーダーの分散を有効にします。バックグラウンドスレッドは、**leader.imbalance.check.interval.seconds** で設定可能な、一定間隔でパーティションリーダーの分散をチェックします。リーダー imbalance が **leader.imbalance.per.broker.percentage** を超える場合は、パーティションに推奨されるリーダーへのリーダーリバランスが発生します。

background.threads**type:** int**Default:** 10**Valid Values:** [1,...]**Importance:** high**Dynamic update:** cluster-wide

さまざまなバックグラウンド処理タスクに使用するスレッドの数。

broker.id**type:** int**デフォルト:** -1**のインポート:** high**Dynamic update:** read-only

このサーバーのブローカーID。設定しないと、一意のブローカーID が生成されます。zookeeper の生成ブローカーID とユーザー設定されたブローカーID の競合を避けるため、生成されるブローカーID は reserved.broker.max.id + 1 から始まります。

compression.type**type:** string**デフォルト:** producer**インポートランス:** high**Dynamic update:** cluster-wide

特定のトピックの最後の圧縮タイプを指定します。この設定では、標準の圧縮コーデック ('gzip', 'snappy', 'lz4', 'zstd') を使用できます。また、圧縮なしと同等の「非圧縮」も受け入れます。「producer」は、プロデューサーによって設定された元の圧縮コードを保持しています。

control.plane.listener.name**type:** string**Default:** null**Importance:** high**Dynamic update:** read-only

コントローラーとブローカー間の通信に使用されるリスナーの名前。ブローカーは control.plane.listener.name を使用して、リスナー一覧にあるエンドポイントを見つけ、コントローラーからの接続をリッスンします。たとえば、ブローカーの設定が: listeners = INTERNAL://192.1.1.8:9092, EXTERNAL://10.1.1.5:9093, CONTROLLER://192.1.1.8:9094 listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL control.listener.name = CONTROLLER:SSL control.plane.listener.name = CONTROLLER On 起動中にブローカーは "192.1.1.8:9094" で起動します。コントローラー側では、zookeeper 経由でブローカーの公開されたエンドポイントを検出すると、control.plane.listener.name を使用してブローカーへの接続を確立するのに使用するエンドポイントを検索します。たとえば、zookeeper にブローカーの公開されたエンドポイント: "endpoints": ["INTERNAL://broker1.example.com:9092", "EXTERNAL://broker1.example.com:9092", "EXTERNAL://bro と、コントローラーの設定が: listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL control.plane.listener.name = CONTROLLER then controller は、セキュリティープロトコル "SSL" とともに "broker1.example.com:9094" を使用してブローカーに接続します。明示的に設定されていない場合、デフォルト値は null になり、コントローラー接続専用のエンドポイントはありません。

delete.topic.enable**type:** boolean**Default:** true**Importance:** high**Dynamic update:** read-only

トピックの削除を有効にします。この設定をオフにしても、管理ツールでトピックを削除しても、影響はありません。

host.name

type: string

デフォルト: ""

Importance: high

Dynamic update: read-only

非推奨: **listeners** が設定されていない場合にのみ使用します。代わりに **listeners** を使用します。ブローカーのホスト名。これが設定されると、このアドレスにのみバインドされます。これが設定されていない場合、すべてのインターフェースにバインドされます。

leader.imbalance.check.interval.seconds

type: long

Default: 300

Importance: high

Dynamic update: read-only

パーティションリバランスチェックを行う頻度がコントローラーによって引き起こされます。

leader.imbalance.per.broker.percentage

type: int

Default: 10

Importance: high

Dynamic update: read-only

ブローカーごとに許可されるリーダー imbalance の比率。ブローカーごとにこの値を上回る場合、コントローラーはリーダーのバランスをトリガーします。値はパーセンテージで指定されます。

listeners

type: string

Default: null

Importance: high

Dynamic update: per-broker

リスナーリスト: リッスンする URI とリスナー名のコンマ区切りリスト。リスナー名がセキュリティープロトコルではない場合、**listener.security.protocol.map** も設定する必要があります。リスナー名とポート番号は一意でなければなりません。すべてのインターフェースにバインドする 0.0.0.0 としてホスト名を指定します。デフォルトインターフェースにバインドする場合は、ホスト名を空のままにします。法リスナー一覧の例: PLAINTEXT://myhost:9092,SSL://:9091 CLIENT://0.0.0.0:9092,REPLICATION://localhost:9093

log.dir

type: string

Default: /tmp/kafka-logs

Importance: high

Dynamic update: read-only

ログデータの保管先のディレクトリー (log.dirs プロパティーの補助グループ)。

log.dirs

type: string

Default: null

Importance: high

Dynamic update: read-only

ログデータが保存されるディレクトリー。設定されていない場合は、log.dir の値が使用されます。

log.flush.interval.messages

type: long

デフォルト : 9223372036854775807

有効な値 : [1,...]

Importance: high

Dynamic update: cluster-wide

メッセージがディスクにフラッシュされる前に、ログパーティションで累積されたメッセージの数。

log.flush.interval.ms

タイプ : long

デフォルト : null

インポートランス : high

Dynamic update: cluster-wide

ディスクにフラッシュされる前に、トピックのメッセージがメモリーに保持される最大時間（ミリ秒単位）。設定されていない場合、log.flush.scheduler.interval.ms の値が使用されます。

log.flush.offset.checkpoint.interval.ms

type: int

Default: 60000 (1 分)

有効値 : [0,...]

Importance: high

Dynamic update: read-only

ログリカバリポイントとして動作する最後のフラッシュの永続レコードを更新する頻度。

log.flush.scheduler.interval.ms

type: long

デフォルト : 9223372036854775807

Importance: high

Dynamic update: read-only

ログフラッシュ担当者がログをディスクにフラッシュする必要があるかどうかをチェックする頻度（ミリ秒単位）。

log.flush.start.offset.checkpoint.interval.ms

type: int

Default: 60000 (1 分)

有効値 : [0,...]

Importance: high

Dynamic update: read-only

ログ開始オフセットの永続レコードを更新する頻度。

log.retention.bytes

タイプ : long

デフォルト : -1

のインポート : high

Dynamic update: cluster-wide

ログを削除する前にログの最大サイズ。

log.retention.hours

type: int

デフォルト : 168

のインポート : high

Dynamic update: read-only

ログファイルを削除するまでにログファイルを保持する時間（時間単位）、log.retention.ms プロパティーのデリム。

log.retention.minutes

type: int

Default: null

Importance: high

Dynamic update: read-only

ログファイルを削除するまでにログファイルを保持する時間（分単位）、log.retention.ms プロパティーへのセカンダリー。設定されていない場合は、log.retention.hours の値が使用されます。

log.retention.ms

タイプ : long

デフォルト : null

インポートランス : high

Dynamic update: cluster-wide

ログファイルを削除する前にログファイルを保持する期間（ミリ秒単位）。設定されていない場合、log.retention.minutes の値が使用されます。-1 に設定すると、時間制限が適用されません。

log.roll.hours

type: int

Default: 168

Valid Values: [1,...]

インポートランス : high

Dynamic update: read-only

新しいログセグメントが展開されるまでの最大時間（時間）、log.roll.ms プロパティーへのセカンダリー。

log.roll.jitter.hours

type: int

デフォルト : 0

有効な値 : [0,...]

インポートランス : high

Dynamic update: read-only

logRollTimeMillis（時間単位）から subtract する最大のジッター。log.roll.jitter.ms プロパティーへのセカンダリー。

log.roll.jitter.ms

タイプ : long

デフォルト : null

インポートランス : high

Dynamic update: cluster-wide

logRollTimeMillis から減算する最大ジッター（ミリ秒単位）。設定されていない場合は、log.roll.jitter.hours の値が使用されます。

log.roll.ms

タイプ : long

デフォルト : null

インポートランス : high

Dynamic update: cluster-wide

新しいログセグメントがロールアウトされるまでの最大時間（ミリ秒単位）。設定されていない場合は、`log.roll.hours` の値が使用されます。

log.segment.bytes

type: int
デフォルト: 1073741824(1 gibibyte)
有効な値: [14,...]
Importance: high
Dynamic update: cluster-wide
 単一ログファイルの最大サイズ。

log.segment.delete.delay.ms

type: long
デフォルト: 60000 (1 分)
有効値: [0,...]
Importance: high
Dynamic update: cluster-wide
 ファイルシステムからファイルを削除するまでの待機時間。

message.max.bytes

type: int
Default: 1048588
Valid Values:[0,...]
Importance: high
Dynamic update: cluster-wide
 Kafka によって許可される最大レコードバッチサイズ（圧縮が有効な場合）。これが増加し、コンシューマーに 0.10.2 よりも古いコンシューマーがある場合、コンシューマーのフェッチサイズも増やす必要があります。これにより、この大規模なレコードバッチを取得できます。最新のメッセージ形式のバージョンでは、レコードは常に効率のためにバッチにグループ化されます。以前のメッセージ形式のバージョンでは、圧縮されていないレコードはバッチにグループ化されず、この制限はその場合における単一レコードにのみ適用されます。これは、トピックレベルの **max.message.bytes** 設定で、トピックごとに設定できます。

min.insync.replicas

type: int
Default: 1
Valid Values:[1,...]
Importance: high
Dynamic update: cluster-wide
 プロデューサーが `ack` を `「all」`（または `「-1」`）に設定する場合、`min.insync.replicas` は書き込みが正常に考慮されるように書き込みを確認する必要があるレプリカの最小数を指定します。この最小値が満たされない場合、プロデューサーは例外を発生させます（`NotEnoughReplicasAfterAppend` のいずれか）。`min.insync.replicas` および `ack` を使用すると、高い持続性の保証を実施することができます。典型的なシナリオは、レプリケーション係数 3 でトピックを作成し、`min.insync.replicas` を 2 に設定し、`「all」` で生成することです。これにより、大多数のレプリカが書き込みを受信しない場合に、プロデューサーが例外を発生させます。

num.io.threads

type: int
Default: 8
Valid Values:[1,...]
Importance: high
Dynamic update: cluster-wide

サーバーが要求の処理に使用するスレッドの数。ディスクI/Oを含む可能性があります。

num.network.threads

type: int

Default: 3

Valid Values: [1,...]

Importance: high

Dynamic update: cluster-wide

ネットワークから要求を受信し、応答をネットワークに送信するためにサーバーが使用するスレッドの数。

num.recovery.threads.per.data.dir

type: int

Default: 1

Valid Values: [1,...]

Importance: high

Dynamic update: cluster-wide

起動時にログリカバリーに使用されるデータディレクトリーごとのスレッド数。シャットダウン時にフラッシュされる。

num.replica.alter.log.dirs.threads

type: int

Default: null

Importance: high

Dynamic update: read-only

ディスクI/Oを含む可能性のあるログディレクトリー間でレプリカを移動できるスレッドの数。

num.replica.fetchers

type: int

デフォルト : 1

のインポート : high

Dynamic update: cluster-wide

ソースブローカーからメッセージをレプリケートするために使用される fetcher スレッドの数。この値を大きくすると、フォロワーブローカーのI/O 並行処理量を増やすことができます。

offset.metadata.max.bytes

type: int

Default: 4096(4 kibibytes)

Importance: high

Dynamic update: read-only

オフセットコミットに関連するメタデータエントリーの最大サイズ。

offsets.commit.required.acks

type: short

デフォルト : -1

障害 : high

Dynamic update: read-only

コミットを許可する前に必要なサックです。通常、デフォルト(-1)は上書きできません。

offsets.commit.timeout.ms

type: int

Default: 5000(5 seconds)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

オフセットコミットは、オフセットトピックのすべてのレプリカがコミットを受信するか、タイムアウトに達するまで遅延します。これは、プロデューサーリクエストのタイムアウトと同様です。

offsets.load.buffer.size

type: int

Default: 5242880

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

オフセットをキャッシュにロードするときにオフセットセグメントから読み取るバッチサイズ (soft-limit、レコードが大きすぎると上書きされます)。

offsets.retention.check.interval.ms

type: long

デフォルト : 600000 (10 分)

有効な値 : [1,...]

Importance: high

Dynamic update: read-only

古いオフセットをチェックする頻度。

offsets.retention.minutes

type: int

デフォルト : 10080

有効値 : [1,...]

インポートランス : high

Dynamic update: read-only

コンシューマーグループがすべてコンシューマー (空になる) を失うと、破棄する前にこの保持期間に対してオフセットが保持されます。スタンドアロンコンシューマー (手動割り当てを使用) の場合、オフセットは最後のコミット時とこの保持期間の後に期限切れになります。

offsets.topic.compression.codec

type: int

デフォルト : 0

のインポート : high

Dynamic update: read-only

オフセットトピックの圧縮コードc - 「アトミック」コミットを達成するために圧縮を使用できません。

offsets.topic.num.partitions

type: int

Default: 50

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

オフセットコミットトピックのパーティション数 (デプロイメント後に変更しないでください)。

offsets.topic.replication.factor

type: short

Default: 3

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

オフセットトピックのレプリケーション係数（可用性を確保するためにより高い設定）。クラスターのサイズがこのレプリケーション係数要件を満たすまで、内部トピックの作成は失敗します。

offsets.topic.segment.bytes

type: int

Default: 104857600(100 mebibytes)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

ログコンパクションとキャッシュの読み込みを迅速化するために、オフセットトピックセグメントバイトを比較的小さくする必要があります。

port

type: int

デフォルト : 9092

インポートランス : high

Dynamic update: read-only

非推奨 : **listeners** が設定されていない場合にのみ使用します。代わりに **listeners** を使用します。ポートを使用して接続をリッスンし、これを受け入れます。

queued.max.requests

type: int

Default: 500

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

ネットワークスレッドをブロックする前にデータプレーンで許可されるキューに入れられた要求の数。

quota.consumer.default

type: long

Default: 9223372036854775807

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

非推奨 : 動的デフォルトクォータが、Zookeeper の<user、<client-id> または<user, client-id> に設定されていない場合にのみ使用されます。clientId/consumer グループによるコンシューマー識別名は、1 秒あたりにこの値を超えるバイトを取得するとスロットルされます。

quota.producer.default

type: long

Default: 9223372036854775807

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

非推奨 : 動的デフォルトクォータが、Zookeeper の<user>、<client-id> または<user, client-id> に設定されていない場合にのみ使用されます。毎秒の値を超えるバイトを生成すると、clientId によるプロデューサーの区別がスロットリングされます。

replica.fetch.min.bytes

type: int

デフォルト : 1

のインポート : high

Dynamic update: read-only

各フェッチ応答に最低バイト数が必要です。十分なバイト数がない場合は、**replica.fetch.wait.max.ms**（ブローカー設定）まで待機します。

replica.fetch.wait.max.ms

type: int

Default: 500

Importance: high

Dynamic update: read-only

フォロワーレプリカで発行される各フェッチリクエストの最大待機時間。低スループットのトピックでISRが大幅に削減されないように、この値は常に **replica.lag.time.max.ms** よりも小さくする必要があります。

replica.high.watermark.checkpoint.interval.ms

type: long

デフォルト : 5000 (5 秒)

インポート : high

Dynamic update: read-only

ハイウォーターマークをディスクに保存する頻度。

replica.lag.time.max.ms

type: long

デフォルト : 30000 (30 秒)

インポート性 : high

Dynamic update: read-only

フォロワーがフェッチリクエストを送信していない場合や、リーダーが少なくともこの時間に対してリーダーのログ終了オフセットに消費されていない場合、リーダーはフォロワーから削除されます。

replica.socket.receive.buffer.bytes

type: int

Default: 65536(64 kibibytes)

修正 : high

Dynamic update: read-only

ネットワーク要求のソケット受信バッファ。

replica.socket.timeout.ms

type: int

デフォルト : 30000 (30 秒)

インポート : high

Dynamic update: read-only

ネットワーク要求のソケットタイムアウト。この値は、少なくとも **replica.fetch.wait.max.ms** である必要があります。

request.timeout.ms

type: int

デフォルト : 30000 (30 秒)

インポート : high

Dynamic update: read-only

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。

socket.receive.buffer.bytes

type: int

Default: 102400(100 kibibytes)

修正 : high

Dynamic update: read-only

ソケットサーバーのソケットの SO_RCVBUF バッファ。値が-1の場合、OS のデフォルトが使用されます。

socket.request.max.bytes

type: int

Default: 104857600(100 mebibytes)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

ソケットリクエストのバイト数。

socket.send.buffer.bytes

type: int

Default: 102400(100 kibibytes)

修正 : high

Dynamic update: read-only

ソケットサーバーのソケットの SO_SNDBUF バッファ。値が-1の場合、OS のデフォルトが使用されます。

transaction.max.timeout.ms

type: int

Default: 900000 (15 分)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

トランザクションに対して許可される最大タイムアウト。クライアントの要求されたトランザクション時間がこの値を超えると、ブローカーは InitProducerIdRequest のエラーを返します。これにより、クライアントがタイムアウトを過剰に防ぎ、トランザクションに含まれるトピックから読み取るコンシューマーを停止することができます。

transaction.state.log.load.buffer.size

type: int

Default: 5242880

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

プロデューサーID とトランザクションをキャッシュにロードする際にトランザクションログセグメントから読み取るバッチサイズ (レコードが大きすぎる場合はsoft-limit)。

transaction.state.log.min.isr

type: int

Default: 2

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

トランザクショントピックの `min.insync.replicas` 設定を上書きします。

transaction.state.log.num.partitions

type: int

Default: 50

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

トランザクショントピックのパーティション数（デプロイメント後に変更しないでください）。

transaction.state.log.replication.factor

type: short

Default: 3

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

トランザクショントピックのレプリケーション係数（可用性を確保するためにより高い設定）。クラスターのサイズがこのレプリケーション係数要件を満たすまで、内部トピックの作成は失敗します。

transaction.state.log.segment.bytes

type: int

Default: 104857600(100 mebibytes)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

ログコンパクションとキャッシュ負荷を迅速化するために、トランザクショントピックセグメントバイトを比較的小さくする必要があります。

transactional.id.expiration.ms

type: int

Default: 604800000(7 days)

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

トランザクションID を期限切れとなる前にトランザクションステータスの更新を受信せずにトランザクションコーディネーターが待機する期間（ミリ秒単位）。この設定は、プロデューサーID の期限切れにも影響します。プロデューサーID は、この時間の後に、指定のプロデューサーID で最後に書き込みが経過した後に期限切れになります。トピックの保持設定が原因でプロデューサーID がプロデューサーID から削除されると、プロデューサーID がまもなく期限切れになる可能性があることに注意してください。

unclean.leader.election.enable

type: boolean

Default: false

Importance: high

Dynamic update: cluster-wide

ISR セットにないレプリカを最後の手段としてリーダーとして選択したくない場合は、データが失われる可能性があります。

zookeeper.connect

type: string

Default: null

Importance: high

Dynamic update: read-only

ホストおよびポートがZooKeeper サーバーのホストとポートである **hostname:port** 形式で ZooKeeper 接続文字列を指定します。ZooKeeper マシンの停止時に他の ZooKeeper ノード経由で接続できるようにするには、**hostname1:port1,hostname2:port2,hostname3:port3** の形式で複数のホストを指定することもできます。サーバーは、ZooKeeper 接続文字列の一部として ZooKeeper chroot パスを持つこともできます。これは、データをグローバル ZooKeeper 名前空間の一部のパスに置いています。たとえば、chroot パスを **/chroot/path** に付与するには、接続文字列を **hostname1:port1,hostname2:port2,hostname3:port3/chroot/path** として指定します。

zookeeper.connection.timeout.ms

type: int

Default: null

Importance: high

Dynamic update: read-only

クライアントが zookeeper への接続を確立するまで待機する最大時間。設定されていない場合は、`zookeeper.session.timeout.ms` の値が使用されます。

zookeeper.max.in.flight.requests

type: int

Default: 10

Valid Values: [1,...]

Importance: high

Dynamic update: read-only

ブロックする前にクライアントが Zookeeper に送信するリクエストの最大数。

zookeeper.session.timeout.ms

type: int

Default: 18000(18 seconds)

Importance: high

Dynamic update: read-only

ZooKeeper セッションのタイムアウト。

zookeeper.set.acl

type: boolean

Default: false

Importance: high

Dynamic update: read-only

セキュアな ACL を使用するようにクライアントを設定します。

broker.id.generation.enable

type: boolean

Default: true

importance: medium

Dynamic update: read-only

サーバーでブローカーIDの自動生成を有効にします。有効にする場合は、`reserved.broker.max.id` に設定した値を確認する必要があります。

broker.rack

type: string

Default: null

Importance: medium

Dynamic update: read-only

ブローカーのラック。これは、フォールトトレランスのラック認識レプリケーション割り当てで使用されます。例: **RACK1**、**us-east-1d**

connections.max.idle.ms

type: long

デフォルト: 600000 (10 分)

Importance: medium

Dynamic update: read-only

アイドル接続のタイムアウト: サーバースocketプロセッサスレッドは、これよりもアイドル状態の接続を切断します。

connections.max.reauth.ms

type: long

デフォルト: 0

インポート割: medium

Dynamic update: read-only

正数（デフォルトは正数ではなく0）に明示的に設定されている場合、設定された値を超えないセッションライフタイムは、認証時にv2.2.0以降のクライアントに通信されます。ブローカーは、セッションライフタイム内で再認証されない接続が切断され、その後再認証以外の目的で使用されます。設定名には、任意でリスナープレフィックスとSASLメカニズム名のプレフィックスをより小文字にすることができます。例:

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

controlled.shutdown.enable

type: boolean

Default: true

importance: medium

Dynamic update: read-only

サーバーの制御されたシャットダウンを有効にします。

controlled.shutdown.max.retries

type: int

Default: 3

importance: medium

Dynamic update: read-only

制御されたシャットダウンは、複数の理由で失敗する可能性があります。これにより、このような障害が発生する際の再試行回数を決定します。

controlled.shutdown.retry.backoff.ms

type: long

Default: 5000(5 seconds)

Importance: medium

Dynamic update: read-only

各再試行の前に、システムは以前の失敗の原因となった状態から回復する時間（コントローラー、レプリカラグなど）が必要になります。この設定は、再試行するまでの待機時間を決定します。

controller.socket.timeout.ms

type: int

デフォルト : 30000 (30 秒)

インポート性 : medium

Dynamic update: read-only

コントローラーからブローカーへのチャネルのソケットタイムアウト。

default.replication.factor

type: int

Default: 1

Importance: medium

Dynamic update: read-only

自動的に作成されたトピックに対するデフォルトのレプリケーション要素。

delegation.token.expiry.time.ms

type: long

デフォルト : 86400000 (1日)

有効値 : [1,...]

Importance: medium

Dynamic update: read-only

トークンの更新が必要となるまでのトークンの有効性時間（秒単位）。デフォルト値は1日です。

delegation.token.master.key

type: password

Default: null

Importance: medium

Dynamic update: read-only

非推奨 : この設定の代わりに使用する delegation.token.secret.key のエイリアス。

delegation.token.max.lifetime.ms

type: long

Default: 604800000(7 days)

Valid Values:[1,...]

Importance: medium

Dynamic update: read-only

トークンには最長有効期間があるため、更新できません。デフォルト値は7日です。

delegation.token.secret.key

type: password

Default: null

Importance: medium

Dynamic update: read-only

委任トークンを生成し、検証するためのシークレットキー。同じキーをすべてのブローカーで設定する必要があります。キーが設定されていないか、または空の文字列に設定されている場合、ブローカーは委譲トークンのサポートを無効にします。

delete.records.purgatory.purge.interval.requests

type: int

Default: 1
Importance: medium
Dynamic update: read-only
削除要求のページ間隔（リクエスト数）。

fetch.max.bytes

type: int
Default: 57671680(55 mebibytes)
Valid Values:[1024,...]
Importance: medium
Dynamic update: read-only
フェッチリクエストに対して返す最大バイト数。1024 以上である必要があります。

fetch.purgatory.purge.interval.requests

type: int
Default: 1000
Importance: medium
Dynamic update: read-only
フェッチリクエストパージの間隔（リクエスト数）。

group.initial.rebalance.delay.ms

type: int
Default: 3000（3 秒）
Importance: medium
Dynamic update: read-only
グループコーディネーターが新しいグループに参加するまで待機してから最初のリバランスを実行する時間。遅延時間が長くなると、リバランスが少なくなりますが、処理を開始するまで時間が長くなります。

group.max.session.timeout.ms

type: int
Default: 1800000（30 分）
Importance: medium
Dynamic update: read-only
登録したコンシューマーに対して許可される最大セッションタイムアウト。タイムアウトが長くなると、障害検出に長い時間が長くなったときに、コンシューマーがハートビートの間にメッセージを処理するのがより長くなります。

group.max.size

type: int
Default: 2147483647
Valid Values:[1,...]
Importance: medium
Dynamic update: read-only
1つのコンシューマーグループに対応できるコンシューマーの最大数。

group.min.session.timeout.ms

type: int
Default: 6000（6 秒）
重要: medium
Dynamic update: read-only

登録されたコンシューマーに対して許可される最小セッションタイムアウト。タイムアウトが短くなると、コンシューマーのハートビートがより頻繁に発生する可能性が高くなり、ブローカーリソースが過剰になる可能性があります。

inter.broker.listener.name

type: string

Default: null

Importance: medium

Dynamic update: read-only

ブローカー間の通信に使用されるリスナーの名前。これが未設定の場合は、リスナー名は `security.inter.broker.protocol` で定義されます。これと `security.inter.broker.protocol` プロパティを同時に設定することはエラーです。

inter.broker.protocol.version

type: string

デフォルト : 2.8-IV1

有効な値 : [0.8.0, 0.8.1, 0.8.2, 0.9.0 0.10.0-IV0, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV1, 2.1-IV1, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.4-IV0, 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1]

Importance: medium:

Dynamic update: read-only

ブローカー間プロトコルを使用するバージョンを指定します。通常、これはすべてのブローカーが新規バージョンにアップグレードされた後に出力されます。一部の有効な値の例としては、0.8.0、0.8.1、0.8.1.1、0.8.2、0.8.2.0、0.8.2.1、0.9.0.0、0.9.0.1 Check ApiVersion などがあります。

log.cleaner.backoff.ms

type: long

デフォルト : 15000 (15 秒)

有効な値 : [0,...]

Importance: medium

Dynamic update: cluster-wide

クリーニングするログがない場合にスリープする時間。

log.cleaner.dedupe.buffer.size

タイプ : long

デフォルト : 134217728

インポート : medium

Dynamic update: cluster-wide

細かい全スレッドでログの重複排除に使用されるメモリーの合計。

log.cleaner.delete.retention.ms

type: long

デフォルト : 86400000 (1 日)

Importance: medium

Dynamic update: cluster-wide

削除レコードの保持期間

log.cleaner.enable

type: boolean

Default: true

importance: medium

Dynamic update: read-only

ログクリーナープロセスを有効にして、サーバーで実行できるようになります。
 cleanup.policy=compact のトピック（内部オフセットトピックを含む）を使用する場合は有効にする必要があります。無効にされている場合、これらのトピックは圧縮されず、継続的に拡張されます。

log.cleaner.io.buffer.load.factor

type: double

デフォルト : 0.9

のインポート : medium

Dynamic update: cluster-wide

ログクリーナーがバッファード係数を減らします。dedupe バッファの割合（パーセント）。
 値が大きいほど一度により多くのログを消去できますが、ハッシュの競合が増える可能性があります。

log.cleaner.io.buffer.size

type: int

Default: 524288

Valid Values:[0,...]

Importance: medium

Dynamic update: cluster-wide

細かい全スレッドでログクリーナー I/O バッファに使用されるメモリーの合計。

log.cleaner.io.max.bytes.per.second

type: double

デフォルト : 1.7976931348623157E308

Importance: medium

Dynamic update: cluster-wide

ログローサーをスロットリングし、読み取りおよび書き込みの合計が平均的にこの値より小さくなるようです。

log.cleaner.max.compaction.lag.ms

type: long

デフォルト : 9223372036854775807

Importance: medium

Dynamic update: cluster-wide

メッセージがログのコンパクトな状態に保つ最大時間。圧縮されるログにのみ適用されます。

log.cleaner.min.cleanable.ratio

type: double

デフォルト : 0.5

インポート : medium

Dynamic update: cluster-wide

クリーニングの対象となるログの合計ログに対するダーティーログの最小比率。また、ログコンパクター.max.cleaner.max.compaction.lag.ms または log.cleaner.min.compaction.lag.ms 設定も指定されると、ログコンパクションに対するログコンパクションの許容量も考慮し（例：ダーティー率のしきい値に達する）場合は、ログに、少なくとも log.cleaner.min.compaction.lag.ms のレコードがあります。または(i)ログに、ほとんどの log.cleaner.max.compaction.lag.ms 期間のダーティー（複合でない）レコードがある場合です。

log.cleaner.min.compaction.lag.ms

タイプ : long

デフォルト : 0

インポートの問題 : medium

Dynamic update: cluster-wide

メッセージがログに記録されない最小時間。圧縮されるログにのみ適用されます。

log.cleaner.threads

type: int

Default: 1

Valid Values: [0,...]

Importance: medium

Dynamic update: cluster-wide

ログ消去に使用するバックグラウンドスレッドの数。

log.cleanup.policy

type: list

Default: delete

Valid Values: [compact, delete]

Importance: medium

Dynamic update: cluster-wide

保持ウィンドウ以外のセグメントのデフォルトのクリーンアップポリシー。有効なポリシーのカンマ区切りの一覧。有効なポリシー : "delete" および "compact"。

log.index.interval.bytes

type: int

Default: 4096(4 kibibytes)

有効な値 : [0,...]

Importance: medium

Dynamic update: cluster-wide

オフセットインデックスにエントリーを追加する間隔。

log.index.size.max.bytes

type: int

Default: 10485760(10 mebibytes)

Valid Values: [4,...]

Importance: medium

Dynamic update: cluster-wide

オフセットインデックスの最大サイズ (バイト単位)。

log.message.format.version

type: string

デフォルト : 2.8-IV1

有効な値 : [0.8.0, 0.8.1, 0.8.2, 0.9.0 0.10.0-IV0, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV1, 2.1-IV1, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.4-IV0, 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1]

Importance: medium:

Dynamic update: read-only

ブローカーがログへのメッセージの追加に使用するメッセージ形式バージョンを指定します。値は有効な ApiVersion である必要があります。いくつかの例 : 0.8.2、0.9.0.0、0.10.0、詳細は ApiVersion を確認してください。特定のメッセージ形式バージョンを設定すると、ディスク上のすべての既存メッセージがすべて、指定のバージョンより小さいか、または等しいことが認定されます。この値を不適切に設定すると、理解のない形式のメッセージを受信するため、以前のバージョンを持つコンシューマーが破損してしまう可能性があります。

log.message.timestamp.difference.max.ms

type: long

デフォルト : 9223372036854775807

Importance: medium

Dynamic update: cluster-wide

ブローカーがメッセージを受信するとメッセージに指定されたタイムスタンプの間の最大差です。log.message.timestamp.type=CreateTime の場合には、タイムスタンプの相違点がこのしきい値を超えると、メッセージは拒否されます。この設定は、log.message.timestamp.type=LogAppendTime の場合は無視されます。最大タイムスタンプの差異はlog.retention.ms より大きくならず、ログローリングで頻繁に行われるのを回避します。

log.message.timestamp.type

type: string

デフォルト : CreateTime

Valid Values:[CreateTime, LogAppendTime]

Importance: medium

Dynamic update: cluster-wide

メッセージのタイムスタンプがメッセージ作成時またはログ追加時間であるかを定義します。値は

CreateTime または **LogAppendTime** のいずれかに指定する必要があります。

log.preallocate

type: boolean

Default: false

Importance: medium

Dynamic update: cluster-wide

新規セグメントの作成時にファイルを事前に割り当てる必要がありますか？Windows で Kafka を使用している場合は、true に設定する必要があります。

log.retention.check.interval.ms

type: long

デフォルト : 300000 (5 分)

有効な値 : [1,...]

Importance: medium

Dynamic update: read-only

ログクリーナーがログの削除の対象となるかどうかを確認する頻度（ミリ秒単位）。

max.connection.creation.rate

type: int

Default: 2147483647

Valid Values:[0,...]

Importance: medium

Dynamic update: cluster-wide

いつでもブローカーで許可される最大コネクション作成速度。リスナーレベルの制限は、設定名とリスナーの接頭辞（例：**listener.name.internal.max.connection.creation.rate**）をプレフィックスして設定できます。ブローカー全体の接続レート制限は、リスナーの制限をアプリケーション要件に応じて設定する必要があります。ブローカー間のリスナー以外のリスナーまたはブローカー制限のいずれかに達した場合、新しい接続はスロットリングされます。ブローカー間リスナーの接続は、リスナーレベルのレート制限に達した場合にのみスロットリングされます。

max.connections

type: int

Default: 2147483647

Valid Values: [0,...]**Importance:** medium**Dynamic update:** cluster-wide

いつでもブローカーで許可される接続の最大数。この制限は、`max.connections.per.ip` を使用して設定したすべてのip 制限に加えて適用されます。リスナーレベルの制限は、設定名にリスナープレフィックス（例：**`listener.name.internal.max.connections`**）をプレフィックスして設定することもできます。リスナーの制限はアプリケーション要件に基づいて設定する必要がありますが、ブローカー全体の制限はブローカーの容量に基づいて設定する必要があります。リスナーまたはブローカーの制限に達すると、新規接続はブロックされます。ブローカー間リスナーの接続は、ブローカー全体の制限に達した場合でも許可されます。この場合は、別のリスナーで最も最近使用された接続が閉じられます。

max.connections.per.ip**type:** int**Default:** 2147483647**Valid Values:** [0,...]**Importance:** medium**Dynamic update:** cluster-wide

各ip アドレスから許可される接続の最大数。`max.connections.per.ip.overrides` プロパティを使用してオーバーライドが設定されている場合には、これを0 に設定できます。制限に達すると、ip アドレスからの新しい接続はドロップされます。

max.connections.per.ip.overrides**type:** string**デフォルト :** ""**Importance:** medium**Dynamic update:** cluster-wide

デフォルトの最大接続数に対して、IP ごとまたはホスト名ごとのオーバーライドのコンマ区切りリスト。指定できる値は `"hostName:100,127.0.0.1:200"` です。

max.incremental.fetch.session.cache.slots**type:** int**Default:** 1000**Valid Values:** [0,...]**Importance:** medium**Dynamic update:** read-only

維持する増分取得セッションの最大数。

num.partitions**type:** int**Default:** 1**Valid Values:** [1,...]**Importance:** medium**Dynamic update:** read-only

トピックごとのデフォルトのログパーティション数。

password.encoder.old.secret**type:** password**Default:** null**Importance:** medium**Dynamic update:** read-only

動的に設定されたパスワードをエンコードするために使用された古いシークレット。これはシーク

レットの更新時にのみ必要です。これが指定されている場合、動的にエンコードされたすべてのパスワードがこの古いシークレットを使用してデコードされ、ブローカーの起動時に `password.encoder.secret` を使用して再度エンコードされます。

`password.encoder.secret`

type: password

Default: null

Importance: medium

Dynamic update: read-only

このブローカーの動的に設定されたパスワードをエンコードするために使用されるシークレット。

`principal.builder.class`

type: class

Default: null

Importance: medium

Dynamic update: per-broker

承認中に使用される `KafkaPrincipalBuilder` インターフェースをビルドするために使用される、`KafkaPrincipalBuilder` インターフェースを実装するクラスの完全修飾名。この設定は、SSL 上のクライアント認証に以前使用されていた非推奨の `PrincipalBuilder` インターフェースもサポートします。プリンシパルビルダーが定義されていない場合、デフォルトの動作は使用中のセキュリティプロトコルに依存します。SSL 認証では、プリンシパルは、指定されたクライアント証明書からの識別名に適用される **`ssl.principal.mapping.rules`** で定義されるルールを使用して派生されます。それ以外の場合は、クライアント認証が必要ない場合、プリンシパル名は `ANONYMOUS` になります。SASL 認証では、GSSAPI が使用されている場合は **`sasl.kerberos.principal.to.local.rules`** と、他のメカニズムの SASL 認証 ID を使用してプリンシパルが派生されます。PLAINTEXT の場合、プリンシパルは `ANONYMOUS` になります。

`producer.purgatory.purge.interval.requests`

type: int

Default: 1000

Importance: medium

Dynamic update: read-only

プロデューサー要求のページ間隔（リクエスト数）。

`queued.max.request.bytes`

type: long

デフォルト: -1

のインポート: medium

Dynamic update: read-only

より多くのリクエストが読み取られなくなるまで許可されるキュー済みバイト数。

`replica.fetch.backoff.ms`

type: int

Default: 1000 (1秒)

有効値: [0,...]

Importance: medium

Dynamic update: read-only

パーティションの取得エラーが発生した際にスリープする時間。

`replica.fetch.max.bytes`

type: int

Default: 1048576(1 mebibyte)

Valid Values: [0,...]

Importance: medium

Dynamic update: read-only

各パーティションの取得を試みるメッセージのバイト数。これは絶対最大値ではありません。フェッチの最初の空でないパーティションの最初のレコードバッチがこの値よりも大きい場合、レコードバッチは引き続き返され、進捗を確実にすることができます。ブローカーが受け入れる最大レコードバッチサイズは、**message.max.bytes**（ブローカー設定）または **max.message.bytes**（トピック設定）で定義されます。

replica.fetch.response.max.bytes

type: int

Default: 10485760(10 mebibytes)

Valid Values: [0,...]

Importance: medium

Dynamic update: read-only

フェッチ応答全体に対して想定される最大バイト数。レコードがバッチでフェッチされ、フェッチの最初の非空でないパーティションにある最初のレコードバッチがこの値よりも大きい場合、レコードバッチは引き続き返され、進捗を確実にできます。したがって、これは絶対最大値ではありません。ブローカーが受け入れる最大レコードバッチサイズは、**message.max.bytes**（ブローカー設定）または **max.message.bytes**（トピック設定）で定義されます。

replica.selector.class

type: string

Default: null

Importance: medium

Dynamic update: read-only

ReplicaSelector を実装する完全修飾クラス名。これは、推奨される読み取りレプリカを見つけるためにブローカーによって使用されます。デフォルトでは、リーダーを返す実装を使用します。

reserved.broker.max.id

type: int

Default: 1000

Valid Values: [0,...]

Importance: medium

Dynamic update: read-only

broker.id に使用できる最大数。

sasl.client.callback.handler.class

type: class

Default: null

Importance: medium

Dynamic update: read-only

AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.enabled.mechanisms

type: list

Default: GSSAPI

Importance: medium

Dynamic update: per-broker

Kafka サーバーで有効にされる SASL メカニズムのリスト。このリストには、セキュリティープロバイダーが利用できるメカニズムが含まれる可能性があります。GSSAPI のみがデフォルトで有効になっています。

sasl.jaas.config

type: password

Default: null

Importance: medium

Dynamic update: per-broker

JAAS 設定ファイルによって使用される形式で、SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式は、[こちらで説明されています](#)。値のフォーマットは

loginModuleClass controlFlag (optionName=optionValue)*; です。ブローカーの場合、設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。例：

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.kinit.cmd

type: string

デフォルト : /usr/bin/kinit

Importance: medium

Dynamic update: per-broker

Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

type: long

デフォルト : 60000

インポートランス : medium

Dynamic update: per-broker

更新試行の間には、ログインスレッドのスリープ時間。

sasl.kerberos.principal.to.local.rules

type: list

デフォルト : DEFAULT

Importance: medium

Dynamic update: per-broker

プリンシパル名から短縮名（通常はオペレーティングシステムのユーザー名）へのマッピングのルールリスト。ルールは順番に評価され、プリンシパル名と一致する最初のルールが短縮名にマップされます。リスト内の後続ルールは無視されます。デフォルトでは、

{username}/{hostname}@{REALM} 形式のプリンシパル名は {username} にマッピングされます。[形式の詳細は、「セキュリティー承認および acls」を参照してください](#)。KafkaPrincipalBuilder の拡張が **principal.builder.class** 設定によって提供された場合、この設定は無視されることに注意してください。

sasl.kerberos.service.name

type: string

Default: null

Importance: medium

Dynamic update: per-broker

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定のいずれかで定義できます。

sasl.kerberos.ticket.renew.jitter

type: double

Default: 0.05

Importance: medium

Dynamic update: per-broker

ランダムなジッターの割合が更新時間に追加されます。

sasl.kerberos.ticket.renew.window.factor

type: double

デフォルト : 0.8

重要 : medium

Dynamic update: per-broker

指定のウィンドウ係数からチケットの有効期限に達するまで、ログインスレッドはスリープ状態になります。

sasl.login.callback.handler.class

type: class

Default: null

Importance: medium

Dynamic update: read-only

AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラー設定の前にリスナープレフィックスと SASL メカニズム名（小文字の）を付ける必要があります。例：

listener.name.sasl_ssl.scram-sha-

256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler。

sasl.login.class

type: class

Default: null

Importance: medium

Dynamic update: read-only

Login インターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。For example,

listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.login.refresh.buffer.seconds

type: short

Default: 300

Importance: medium

Dynamic update: per-broker

認証情報の更新時における認証情報が失効するまでの時間（秒単位）。更新が行われない場合、バッファの秒数より期限切れになり始めたら、更新ができるだけ多くのバッファ時間を維持します。有効な値は 0 から 3600（1 時間）の間です。値が指定されていない場合は、デフォルト値の 300（5 分）が使用されます。認証情報の残存期間が経過すると、この値と

sasl.login.refresh.min.period.seconds はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

type: short

Default: 60

Importance: medium

Dynamic update: per-broker

ログイン更新スレッドがクレデンシャルを更新する前に待機する最低時間（秒単位）。有効な値は 0 から 900（15 分）までの値になります。値が指定されていない場合は、デフォルト値の 60（1

分) が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.buffer.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

type: double

デフォルト: 0.8

重要: medium

Dynamic update: per-broker

ログイン更新スレッドは、認証情報の有効期間と相対的な期間係数に達するまでスリープします。この場合、認証情報の更新を試みます。有効な値は 0.5(50%) と 1.0(100%) です。値が指定されていない場合、デフォルト値の 0.8(80%) が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

type: double

Default: 0.05

Importance: medium

Dynamic update: per-broker

ログイン更新スレッドのスリープ時間に追加されたクレデンシャルの有効期間に対するランダムなジッターの最大数。有効な値は 0 から 0.25(25%) までの値になります。値が指定されていない場合は、デフォルト値の 0.05(5%) が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.mechanism.inter.broker.protocol

type: string

Default: GSSAPI

Importance: medium

Dynamic update: per-broker

ブローカー間の通信に使用される SASL メカニズム。デフォルトは GSSAPI です。

sasl.server.callback.handler.class

type: class

Default: null

Importance: medium

Dynamic update: read-only

AuthenticateCallbackHandler インターフェースを実装する SASL サーバーコールバックハンドラークラスの完全修飾名。サーバーコールバックハンドラーの前に、リスナープレフィックスと SASL メカニズム名を小文字で付加する必要があります。例:

`listener.name.sasl_ssl.plain.sasl.server.callback.handler.class=com.example.CustomPlainCallbackHandler。`

security.inter.broker.protocol

type: string

Default: PLAINTEXT

Importance: medium

Dynamic update: read-only

ブローカー間の通信に使用されるセキュリティープロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。これと `inter.broker.listener.name` プロパティーを同時に設定することはエラーです。

socket.connection.setup.timeout.max.ms

type: long

デフォルト: 30000 (30 秒)

インポート性: medium

Dynamic update: read-only

クライアントがソケット接続を確立するまで待機する最大時間。接続設定のタイムアウトにより、連続する接続の失敗ごとに指数関数的に増加します。接続タイムアウトを回避するために、ランダム化係数 0.2 がタイムアウトに適用されるため、以下の 20% と計算された値の 20% のランダムな範囲が適用されます。

socket.connection.setup.timeout.ms

type: long

Default: 10000 (10 秒)

重要: medium

Dynamic update: read-only

クライアントがソケット接続を確立するのを待機する期間。タイムアウトが経過する前に接続がビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.cipher.suites

type: list

デフォルト: ""

Importance: medium

Dynamic update: per-broker

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティ設定をネゴシエートするために使用される認証、暗号化、MAC およびキー交換アルゴリズムの名前付き組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.client.auth

type: string

Default: none

Valid Values: [required, requested, none]

Importance: medium

Dynamic update: per-broker

クライアント認証を要求するように kafka ブローカーを設定します。以下は一般的な設定です。

- **ssl.client.auth=required** 必要なクライアント認証に設定されているかどうか。
- **ssl.client.auth=requested** これは、クライアント認証は任意となります。必須とは異なり、このオプションがクライアント自体についての認証情報を指定しない場合は、
- **ssl.client.auth=none** これは、クライアント認証が不要なことを意味します。

ssl.enabled.protocols

type: list

Default: TLSv1.2,TLSv1.3

Importance: medium

Dynamic update: per-broker

SSL 接続に対して有効なプロトコル一覧。Java 11 以降、「TLSv1.2」以降で実行する場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーの両方が TLSv1.2 に対応している場合は、クライアントとサーバーは TLSv1.3 を優先します (TLSv1.2 以上をサポートすることを想定します)。多くのケースでは、このデフォルトは問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.key.password

type: password

Default: null

Importance: medium

Dynamic update: per-broker

キーストアファイルまたは 'ssl.keystore.key' で指定された PEM キーの秘密鍵のパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keymanager.algorithm

type: string

デフォルト : SunX509

importance: medium

Dynamic update: per-broker

SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.keystore.certificate.chain

type: password

Default: null

Importance: medium

Dynamic update: per-broker

'ssl.keystore.type' で指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書のリストを持つ PEM 形式のみをサポートします。

ssl.keystore.key

type: password

Default: null

Importance: medium

Dynamic update: per-broker

'ssl.keystore.type' で指定された形式の秘密鍵デフォルトの SSL エンジンファクトリーは、PKCS#8 キーを持つ PEM 形式のみをサポートします。キーが暗号化されている場合は、'ssl.key.password' を使用してキーパスワードを指定する必要があります。

ssl.keystore.location

type: string

Default: null

Importance: medium

Dynamic update: per-broker

キーストアファイルの場所。これはクライアントにはオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password

Default: null

Importance: medium

Dynamic update: per-broker

キーストアファイルのストアパスワード。これはクライアントでは任意で、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアのパスワードは PEM 形式ではサポートされていません。

ssl.keystore.type

type: string

Default: JKS

Importance: medium

Dynamic update: per-broker

キーストアファイルのファイル形式。これはクライアントの場合はオプションになります。

ssl.protocol

type: string

Default: TLSv1.3

Importance: medium

Dynamic update: per-broker

SSLContext の生成に使用される SSL プロトコル。Java 11 以降「TLSv1.2」を使用して実行すると、デフォルトは「TLSv1.3」です。この値は、ほとんどのユースケースで十分です。最近の JVM で許可される値は「TLSv1.2」および「TLSv1.3」です。「TLS」、「TLSv1.1」、「SSL」、「SSLv2」、および「SSLv3」は古い JVM でサポートされる可能性があります、既知のセキュリティ脆弱性が原因で使用は推奨されません。この設定および「ssl.enabled.protocols」のデフォルト値を使用すると、サーバーが「TLSv1.3」をサポートしない場合、クライアントは「TLSv1.2」にダウングレードします。この設定を「TLSv1.2」に設定すると、クライアントは ssl.enabled.protocols の値のいずれかである場合でも、「TLSv1.3」を使用し、サーバーは「TLSv1.3」のみに対応します。

ssl.provider

type: string

Default: null

Importance: medium

Dynamic update: per-broker

SSL 接続に使用されるセキュリティプロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティプロバイダーです。

ssl.trustmanager.algorithm

type: string

デフォルト : PKIX

重要 : medium

Dynamic update: per-broker

SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

ssl.truststore.certificates

type: password

Default: null

Importance: medium

Dynamic update: per-broker

'ssl.truststore.type' で指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書を使用した PEM 形式のみをサポートします。

ssl.truststore.location

type: string

Default: null

Importance: medium

Dynamic update: per-broker

トラストストアファイルの場所。

ssl.truststore.password

type: password

Default: null

Importance: medium

Dynamic update: per-broker

トラストストアファイルのパスワード。パスワードが設定されていない場合は、設定されたトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアのパスワードは PEM 形式ではサポートされていません。

ssl.truststore.type

type: string

Default: JKS

Importance: medium

Dynamic update: per-broker

トラストストアファイルのファイル形式です。

zookeeper.clientCnxnSocket

type: string

Default: null

Importance: medium

Dynamic update: read-only

通常、ZooKeeper への TLS 接続を使用する場合は **org.apache.zookeeper.ClientCnxnSocketNetty** に設定されます。同じ名前の付いた **zookeeper.clientCnxnSocket** システムプロパティーで設定される明示的な値をオーバーライドします。

zookeeper.ssl.client.enable

type: boolean

Default: false

Importance: medium

Dynamic update: read-only

ZooKeeper に接続する際に TLS を使用するようにクライアントを設定します。明示的な値は、**zookeeper.client.secure** システムプロパティーで設定した値よりも優先されます（異なる名前に注意してください）。いずれも設定されていない場合に false に設定されます。true の場合は **zookeeper.clientCnxnSocket** を設定する必要があります（通常は **org.apache.zookeeper.ClientCnxnSocketNetty** です）。他の値を設定するには **zookeeper.ssl.cipher.suites**、**zookeeper.ssl.crl.enable**、**zookeeper.ssl.enabled.protocols**、**zookeeper.ssl.endpoint.identification.algorithm**、**zookeeper.ssl.keystore.location**、**zookeeper.ssl.keystore.password**、**zookeeper.ssl.keystore.type**、**zookeeper.ssl.ocsp.enable**、**zookeeper.ssl.protocol**、**zookeeper.ssl.truststore.location**、**zookeeper.ssl.truststore.password**、**zookeeper.ssl.truststore.type** を設定することができます。

zookeeper.ssl.keystore.location

type: string

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続でクライアント側の証明書を使用する場合のキーストアの場合。 **zookeeper.ssl.keyStore.location** システムプロパティーで設定した明示的な値を上書きします（camelCase に注意してください）。

zookeeper.ssl.keystore.password

type: password

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続でクライアント側の証明書を使用する場合のキーストアパスワード。 **zookeeper.ssl.keyStore.password** システムプロパティーで設定した明示的な値を上書きします（camelCase に注意してください）。ZooKeeper はキーストアパスワードとは異なるキーパス

ワードに対応していないため、キーストアの鍵パスワードはキーストアのパスワードと同じになるように設定してください。それ以外の場合は、Zookeeper への接続は失敗します。

zookeeper.ssl.keystore.type

type: string

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続でクライアント側の証明書を使用する場合のキーストアタイプ。 **zookeeper.ssl.keyStore.type** システムプロパティーで設定した明示的な値を上書きします (camelCase に注意してください)。デフォルト値の **null** は、キーストアのファイル名の拡張子に基づいて型が自動検出されることを意味します。

zookeeper.ssl.truststore.location

type: string

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続を使用する場合のトラストストアの場所。 **zookeeper.ssl.trustStore.location** システムプロパティーで設定した明示的な値を上書きします (camelCase に注意してください)。

zookeeper.ssl.truststore.password

type: password

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続を使用する場合のトラストストアパスワード。 **zookeeper.ssl.trustStore.password** システムプロパティーで設定した明示的な値を上書きします (camelCase に注意してください)。

zookeeper.ssl.truststore.type

type: string

Default: null

Importance: medium

Dynamic update: read-only

ZooKeeper への TLS 接続を使用する場合のトラストストアタイプ。 **zookeeper.ssl.trustStore.type** システムプロパティーで設定した明示的な値を上書きします (camelCase に注意してください)。 **null** のデフォルト値は、トラストストアのファイル名の拡張子に基づいて型が自動検出されることを意味します。

alter.config.policy.class.name

type: class

Default: null

Importance: low

Dynamic update: read-only

検証に使用する変更設定ポリシークラス。クラスは **org.apache.kafka.server.policy.AlterConfigPolicy** インターフェースを実装する必要があります。

alter.log.dirs.replication.quota.window.num

type: int

Default: 11

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

ログディレクトリーのレプリケーションクォータを変更するためにメモリーが保持するサンプルの数。

alter.log.dirs.replication.quota.window.size.seconds

type: int

Default: 1

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

ログディレクトリーのレプリケーションクォータを変更する各サンプルの期間。

authorizer.class.name

type: string

デフォルト: ""

Importance: low

Dynamic update: read-only

ブローカーが承認のためにブローカーによって使用される

sorg.apache.kafka.server.authorizer.Authorizer インターフェースを実装するクラスの完全修飾名。この設定は、以前承認に使用された、非推奨の kafka.security.auth.Authorizer トレイトを実装するオーソライザーもサポートします。

client.quota.callback.class

type: class

Default: null

Importance: low

Dynamic update: read-only

クライアント要求に適用されるクォータ制限を決定するために使用される ClientQuotaCallback インターフェースを実装するクラスの完全修飾名。デフォルトでは、ZooKeeper に保存される <user、client-id>、<user>、または <client-id> クォータが適用されます。特定の要求に対して、セッションのユーザープリンシパルと一致する最も具体的なクォータが適用され、要求の client-id が適用されます。

connection.failed.authentication.delay.ms

type: int

Default: 100

Valid Values: [0,...]

Importance: low

Dynamic update: read-only

認証の失敗時の接続の終了遅延：認証の失敗時に接続が閉じられる時間（ミリ秒単位）です。これは、接続のタイムアウトを防ぐために connection.max.idle.ms 未満に設定する必要があります。

controller.quota.window.num

type: int

Default: 11

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

コントローラー変更クォータ用にメモリーに保持するサンプル数。

controller.quota.window.size.seconds**type:** int**Default:** 1**Valid Values:** [1,...]**Importance:** low**Dynamic update:** read-only

コントローラー変更クォータの各サンプルの期間。

create.topic.policy.class.name**type:** class**Default:** null**Importance:** low**Dynamic update:** read-only

検証に使用する create topic ポリシークラス。クラスは

org.apache.kafka.server.policy.CreateTopicPolicy インターフェースを実装する必要があります。**delegation.token.expiry.check.interval.ms****type:** long**デフォルト :** 3600000 (1時間)**有効値 :** [1,...]**Importance:** low**Dynamic update:** read-only

期限切れの委任トークンを削除するスキャン間隔。

kafka.metrics.polling.interval.secs**type:** int**Default:** 10**Valid Values:** [1,...]**Importance:** low**Dynamic update:** read-only

kafka.metrics.reporters 実装で利用できるメトリクスポーリング間隔（秒単位）。

kafka.metrics.reporters**type:** list**デフォルト :** ""**Importance:** low**Dynamic update:** read-only

Yammer メトリクスのカスタムレポーターとして使用するクラスの一覧。レポーターは

kafka.metrics.KafkaMetricsReporter トレイトを実装する必要があります。クライアントがカスタムレポーターに JMX オペレーションを公開する場合、カスタムレポーターは**kafka.metrics.KafkaMetricsReporterMBean** トレイトを拡張する MBean トレイトを実装し、登録された MBean 規則に準拠する必要があります。**listener.security.protocol.map****type:** string**Default:**

PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL

Importance: low**Dynamic update:** per-broker

リスナー名とセキュリティープロトコル間のマッピングです。これは、複数のポートまたは IP で使用できるように、同じセキュリティープロトコルに対して定義する必要があります。たとえば、両に SSL が必要な場合でも、内部トラフィックと外部トラフィックを分離することができます。具体

的では、ユーザーは `INTERNAL` と `EXTERNAL` という名前のリスナーを定義することができ、このプロパティは `INTERNAL:SSL,EXTERNAL:SSL` として定義できます。キーと値はコロンで区切られ、マップエントリはコンマで区切られます。各リスナー名はマップで1回のみ表示されます。各リスナーに対して異なるセキュリティ（SSL および SASL）を設定できます。そのためには、通常のプレフィックス（リスナー名が小文字化されます）を設定名に追加します。たとえば、`INTERNAL` リスナーに異なるキーストアを設定するには、`listener.name.internal.ssl.keystore.location` という名前の設定が設定されます。リスナー名の設定が設定されていない場合、設定は汎用設定にフォールバックします（例：`ssl.keystore.location`）。

`log.message.downconversion.enable`

type: boolean

Default: true

Importance: low

Dynamic update: cluster-wide

この設定は、要求を満たすためにメッセージ形式のダウンコンバージョンを有効にするかどうかを制御します。**false** に設定すると、ブローカーは古いメッセージ形式を想定しているコンシューマーにダウンコンバートを実行しません。ブローカーは、このような古いクライアントからのリクエストを消費するために `UNSUPPORTED_VERSION` エラーで応答します。この設定は、レプリケーションがフォロワーに必要となる可能性があるメッセージ形式の変換には適用されません。

`metric.reporters`

type: list

デフォルト: ""

Importance: low

Dynamic update: cluster-wide

メトリクスレポーターとして使用するクラスの一

覧。`org.apache.kafka.common.metrics.MetricsReporter` インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。`JmxReporter` は、JMX 統計を登録するために常に含まれます。

`metrics.num.samples`

type: int

Default: 2

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

メトリクスを計算するために保持されるサンプルの数。

`metrics.recording.level`

type: string

Default: INFO

Importance: low

Dynamic update: read-only

メトリックの最大記録レベル。

`metrics.sample.window.ms`

type: long

デフォルト: 30000 (30 秒)

有効値: [1,...]

Importance: low

Dynamic update: read-only

メトリクスサンプルが計算される期間。

password.encoder.cipher.algorithm

type: string

Default: AES/CBC/PKCS5Padding

Importance: low

Dynamic update: read-only

動的に設定されたパスワードをエンコードするために使用される暗号アルゴリズム。

password.encoder.iterations

type: int

Default: 4096

Valid Values: [1024,...]

インポートランス : low

Dynamic update: read-only

動的に設定されたパスワードをエンコードするために使用される反復数。

password.encoder.key.length

type: int

デフォルト : 128

有効な値 : [8,...]

インポートランス : low

Dynamic update: read-only

動的に設定されたパスワードをエンコードするために使用されるキーの長さ。

password.encoder.keyfactory.algorithm

type: string

Default: null

Importance: low

Dynamic update: read-only

動的に設定されたパスワードをエンコードするために使用される SecretKeyFactory アルゴリズム。
デフォルトは PBKDF2WithHmacSHA512 (ある場合) で、PBKDF2WithHmacSHA1 になります。

quota.window.num

type: int

Default: 11

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

クライアントクォータのメモリーに保持するサンプル数。

quota.window.size.seconds

type: int

Default: 1

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

クライアントクォータの各サンプルの期間。

replication.quota.window.num

type: int

Default: 11

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

レプリケーションクォータのメモリーに保持するサンプル数。

replication.quota.window.size.seconds

type: int

Default: 1

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

レプリケーションクォータの各サンプルの期間。

security.providers

type: string

Default: null

Importance: low

Dynamic update: read-only

セキュリティーアルゴリズムを実装するプロバイダーを返す設定可能な作成者クラスのリスト。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.endpoint.identification.algorithm

type: string

Default: https

Importance: low

Dynamic update: per-broker

サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class

Default: null

Importance: low

Dynamic update: per-broker

種別 **org.apache.kafka.common.security.auth.SslEngineFactory** のクラスで **SSL Engine** オブジェクトを提供します。Default value is **org.apache.kafka.common.security.ssl.DefaultSslEngineFactory**.

ssl.principal.mapping.rules

type: string

デフォルト : DEFAULT

Importance: low

Dynamic update: read-only

識別名からクライアント証明書から短縮名へマッピングするルール of the list. Rules are evaluated in order, and the first rule that matches the principal name is used. Rules that do not match are ignored. By default, the X.500 certificate's distinguished name is used as the principal name. [形式の詳細](#)は、「[セキュリティ承認およびacls](#)」を参照してください。KafkaPrincipalBuilder の拡張が **principal.builder.class** 設定によって提供された場合、この設定は無視されることに注意してください。

ssl.secure.random.implementation

type: string

Default: null

Importance: low

Dynamic update: per-broker

SSL 暗号化操作に使用する SecureRandom PRNG 実装。

transaction.abort.timed.out.transaction.cleanup.interval.ms

type: int

Default: 10000 (10 秒)

有効な値: [1,...]

Importance: low

Dynamic update: read-only

タイムアウトしたトランザクションをロールバックする間隔。

transaction.remove.expired.transaction.cleanup.interval.ms

type: int

Default: 3600000(1 hour)

Valid Values: [1,...]

Importance: low

Dynamic update: read-only

transactional.id.expiration.ms のパスにより有効期限が切れたトランザクションを削除する間隔。

zookeeper.ssl.cipher.suites

type: list

Default: null

Importance: low

Dynamic update: read-only

ZooKeeper TLS ネゴシエーション(csv)で使用される有効な暗号スイートを指定します。 **zookeeper.ssl.ciphersuites** システムプロパティーで設定した明示的な値を上書きします（単一の単語は「暗号スイート」に注意してください）。デフォルト値の **null** は、有効な暗号スイートのリストは、使用されている Java ランタイムによって決定されます。

zookeeper.ssl.crl.enable

type: boolean

Default: false

Importance: low

Dynamic update: read-only

ZooKeeper TLS プロトコルで証明書失効リストを有効にするかどうかを指定します。 **zookeeper.ssl.crl** システムプロパティーで設定した明示的な値を上書きします（より短い名前に注意してください）。

zookeeper.ssl.enabled.protocols

type: list

Default: null

Importance: low

Dynamic update: read-only

ZooKeeper TLS ネゴシエーション(csv)で有効なプロトコルを指定します。 **zookeeper.ssl.enabledProtocols** システムプロパティーで設定した明示的な値を上書きします（camelCase に注意してください）。デフォルト値の **null** は、有効なプロトコルが **zookeeper.ssl.protocol** 設定プロパティーの値であることを示します。

zookeeper.ssl.endpoint.identification.algorithm

type: string

Default: HTTPS

Importance: low

Dynamic update: read-only

ZooKeeper TLS ネゴシエーションプロセスでホスト名の検証を有効にするかどうかを指定します（大文字、小文字の区別なし）。「https」は、ZooKeeper ホスト名の検証が有効で、無効であることを意味します（テスト目的でのみ推奨される設定のみ）。明示的な値は、**zookeeper.ssl.hostnameVerification** システムプロパティーで設定される "true" または "false" 値を上書きします（異なる名前と値）。true は https と false は空白を意味します。

zookeeper.ssl.ocsp.enable

type: boolean

Default: false

Importance: low

Dynamic update: read-only

ZooKeeper TLS プロトコルで Online Certificate Status Protocol を有効にするかどうかを指定します。**zookeeper.ssl.ocsp** システムプロパティーで設定した明示的な値を上書きします（より短い名前に注意してください）。

zookeeper.ssl.protocol

type: string

Default: TLSv1.2

Importance: low

Dynamic update: read-only

ZooKeeper TLS ネゴシエーションで使用されるプロトコルを指定します。明示的な値は、同じ名前指定された **zookeeper.ssl.protocol** システムプロパティーで設定した値よりも優先されます。

zookeeper.sync.time.ms

type: int

デフォルト: 2000 (2 秒)

インポートフロー: low

Dynamic update: read-only

ZK フォロワーが ZK リーダーの背後にある方法。

付録B トピック設定パラメーター

cleanup.policy

type: list

Default: delete

Valid Values: [compact, delete]

Server Default Property: log.cleanup.policy

Importance: medium

「delete」または「compact」のいずれかまたは両方である文字列。この文字列は、古いログセグメントで使用する保持ポリシーを指定します。デフォルトのポリシー（「削除」）は、保持時間またはサイズの制限に達すると、古いセグメントを破棄します。「compact」設定は、トピックに関するログコンパクションを有効にします。

compression.type

type: string

Default: producer

Valid Values: [uncompressed, zstd, lz4, snappy, gzip, producer]

Server Default Property: compression.type

Importance: medium

特定のトピックの最後の圧縮タイプを指定します。この設定では、標準の圧縮コーデック（'gzip'、'snappy'、'lz4'、'zstd'）を使用できます。また、圧縮なしと同等の「非圧縮」も受け入れます。

「producer」は、プロデューサーによって設定された元の圧縮コードを保持しています。

delete.retention.ms

type: long

デフォルト: 86400000 (1日)

有効値: [0,...]

Server Default Property: log.cleaner.delete.retention.ms

のインポート機能: medium

ログコンパクションされたトピック用に tombstone マーカーを削除する時間。この設定により、コンシューマーがオフセット 0 から始まる場合に読み取りを完了し、最終的なステージの有効なスナップショットを取得することができます（スキャンを完了する前に廃棄が収集される可能性があります）。

file.delete.delay.ms

type: long

デフォルト: 60000 (1分)

有効な値: [0,...]

サーバーデフォルトプロパティ: log.segment.delete.delay.ms

インポートance: medium

ファイルシステムからファイルを削除するまでの待機時間。

flush.messages

type: long

デフォルト: 9223372036854775807

有効な値: [0,...]

Server Default Property: log.flush.interval.messages

Importance: medium

この設定により、ログに書き込まれたデータの fsync を強制する間隔を指定できます。たとえば、これが 1 に設定されている場合、すべてのメッセージの後に fsync が実行され、5 つあればメッセージごとに 5 つ後に同期されます。通常は、これを設定せず、永続性を確保するためにレプリケーショ

ンを使用し、オペレーティングシステムのバックグラウンドフラッシュ機能がより効率的であるために推奨されます。この設定は、トピックごとに上書きできます（トピックごとの設定についてのセクションを参照）。

flush.ms

type: long
デフォルト : 9223372036854775807
有効な値 : [0,...]
Server Default Property: log.flush.interval.ms
Importance: medium

この設定により、ログに書き込まれたデータの fsync を強制する時間間隔を指定できます。たとえば、1000 に設定されていた場合は、1000 ミリ秒が経過した後は fsync になります。通常は、これを設定せず、永続性を確保するためにレプリケーションを使用し、オペレーティングシステムのバックグラウンドフラッシュ機能がより効率的であるために推奨されます。

follower.replication.throttled.replicas

type: list
デフォルト : ""
Valid Values: [partitionId]:[brokerId],[partitionId]:[brokerId],...
Server Default Property: follower.replication.throttled.replicas
Importance: medium

フォロワー側でログレプリケーションをスロットルするレプリカの一覧。このリストは、[PartitionId]:[BrokerId],[PartitionId]:[BrokerId]:... 形式でレプリカのセットを記述するか、またはワイルドカード '*' を使用して、本トピックのすべてのレプリカをスロットリングするのに使用する必要があります。

index.interval.bytes

type: int
Default: 4096(4 kibibytes)
Valid values: [0,...]
Server Default Property: log.index.interval.bytes
Importance: medium

この設定は、Kafka がインデックスエントリーをオフセットインデックスに追加する頻度を制御します。デフォルト設定では、4096 バイトごとにメッセージがインデックス化されます。より多くのインデックスを使用すると、ログの正確な位置に読み取りをジャンプできますが、インデックスが大きくなる可能性があります。これを変更する必要はありません。

leader.replication.throttled.replicas

type: list
デフォルト : ""
Valid Values: [partitionId]:[brokerId],[partitionId]:[brokerId],...
Server Default Property: leader.replication.throttled.replicas
Importance: medium

リーダー側でログレプリケーションをスロットリングすべきレプリカの一覧。このリストは、[PartitionId]:[BrokerId],[PartitionId]:[BrokerId]:... 形式でレプリカのセットを記述するか、またはワイルドカード '*' を使用して、本トピックのすべてのレプリカをスロットリングするのに使用する必要があります。

max.compaction.lag.ms

type: long
Default: 9223372036854775807
Valid Values: [1,...]

Server Default Property: log.cleaner.max.compaction.lag.ms

Importance: medium

メッセージがログのコンパクトな状態に保つ最大時間。圧縮されるログにのみ適用されます。

max.message.bytes

type: int

Default: 1048588

Valid Values: [0,...]

Server Default Property: message.max.bytes

Importance: medium

Kafka によって許可される最大レコードバッチサイズ（圧縮が有効な場合）。これが増加し、コンシューマーに 0.10.2 よりも古いコンシューマーがある場合、コンシューマーのフェッチサイズも増やす必要があります。これにより、この大規模なレコードバッチを取得できます。最新のメッセージ形式のバージョンでは、レコードは常に効率のためにバッチにグループ化されます。以前のメッセージ形式バージョンでは、圧縮されていないレコードはバッチにグループ化されず、この制限はその場合に 1 つのレコードにのみ適用されます。

message.format.version

type: string

デフォルト : 2.8-IV1

有効な値 : [0.8.0, 0.8.1, 0.8.2, 0.9.0 0.10.0-IV0, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV0, 2.1-IV1, 2.1-IV1, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.3-IV1, 2.4-IV0 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1]

Server Default Property: log.message.format.version

Importance:

ブローカーがログへのメッセージの追加に使用するメッセージ形式バージョンを指定します。値は有効な ApiVersion である必要があります。いくつかの例 : 0.8.2, 0.9.0.0, 0.10.0、詳細は ApiVersion を確認してください。特定のメッセージ形式バージョンを設定すると、ディスク上のすべての既存メッセージがすべて、指定のバージョンより小さいか、または等しいことが認定されます。この値を不適切に設定すると、理解のない形式のメッセージを受信するため、以前のバージョンを持つコンシューマーが破損してしまう可能性があります。

message.timestamp.difference.max.ms

type: long

デフォルト : 9223372036854775807

有効な値 : [0,...]

Server Default Property: log.message.timestamp.difference.max.ms

Importance: medium

ブローカーがメッセージを受信するとメッセージに指定されたタイムスタンプの間の最大差です。message.timestamp.type=CreateTime の場合、タイムスタンプの相違点がこのしきい値を超えるとメッセージは拒否されます。message.timestamp.type=LogAppendTime の場合、この設定は無視されます。

message.timestamp.type

type: string

デフォルト : CreateTime

Valid Values: [CreateTime, LogAppendTime]

Server Default Property: log.message.timestamp.type

Importance: medium

メッセージのタイムスタンプがメッセージ作成時またはログ追加時間であるかを定義します。値は **CreateTime** または **LogAppendTime** のいずれかに指定する必要があります。

min.cleanable.dirty.ratio

type: double
Default: 0.5
Valid Values: [0,...,1]
Server Default Property: log.cleaner.min.cleanable.ratio
Importance: medium

この設定は、ログコンパクターがログの消去を試行する頻度を制御します（**ログコンパクションが有効であることを前提とします**）。デフォルトでは、ログの50%を超えてログが圧縮されるログを取り除くことはできません。この比率は、重複することで最大領域がログに記録されました（最大50%のログが複製される可能性がある）。比率が大きいほど、より効率的な消去が少なくなります。ログの領域がより多くなります。max.compaction.lag.ms または min.compaction.lag.ms 設定も指定される場合、ログコンパクターは、ダーティーの割合のしきい値が満たされて、ログに、min.compaction.lag.ms のレコードが少なくとも min.compaction.lag.ms のレコードが付きます。または、(i) ログに最大期間の max.compaction.lag.ms 期間のダーティー（複合でない）レコードがある場合です。

min.compaction.lag.ms

type: long
デフォルト : 0
有効値 : [0,...]
サーバーデフォルトプロパティ : log.cleaner.min.compaction.lag.ms
Importance: medium
 メッセージがログに記録されない最小時間。圧縮されるログにのみ適用されます。

min.insync.replicas

type: int
Default: 1
Valid Values: [1,...]
Server Default Property: min.insync.replicas
Importance: medium

プロデューサーがackを「all」（または「-1」）に設定する場合、この設定は、書き込みが正常に考慮されるように書き込みを確認する必要があるレプリカの最小数を指定します。この最小値が満たされない場合、プロデューサーは例外を発生させます（NotEnoughReplicasAfterAppendのいずれか）。**min.insync.replicas**と**acks**を使用すると、より優れた持続性の保証を実施することができます。典型的なシナリオは、レプリケーション係数3のトピックを作成し、**min.insync.replicas**を2に設定し、**acks**が「all」で生成することです。これにより、大多数のレプリカが書き込みを受信しない場合に、プロデューサーが例外を発生させます。

preallocate

type: boolean
Default: false
Server Default Property: log.preallocate
Importance: medium

新規ログセグメントの作成時に、ディスク上のファイルを事前に割り当てる必要がある場合 True。

retention.bytes

type: long
デフォルト : -1
サーバーデフォルトプロパティ : log.retention.bytes
インポートガイドライン : medium

この設定では、パーティションの最大サイズ（ログセグメントで構成される）が拡張でき、「削除」保持ポリシーを使用している場合には、古いログセグメントを破棄して領域を解放することができます。デフォルトでは、時間制限のみに制限はありません。この制限はパーティションレベル

で適用されるため、トピックの保持をバイト単位で計算するためパーティションの数によって乗算されます。

retention.ms

type: long

Default: 604800000(7 days)

有効な値: [-1,...]

Server Default Property: log.retention.ms

Importance: medium

この設定は、「削除」保持ポリシーを使用している場合に、古いログセグメントを破棄して領域を解放するまでの時間を制御します。これは、コンシューマーによるデータの読み取りタイミングに関する SLA を表しています。-1 に設定すると、時間制限が適用されません。

segment.bytes

type: int

デフォルト: 1073741824(1 gibibyte)

有効な値: [14,...]

Server Default Property: log.segment.bytes

Importance: medium

この設定は、ログのセグメントファイルサイズを制御します。保持およびクリーニングは、常にファイルを1つずつ実行します。したがって、セグメントサイズが大きいほどファイルの数が少なくなる一方、保持を制御するよりも粒度が低くなります。

segment.index.bytes

type: int

Default: 10485760(10 mebibytes)

有効な値: [0,...]

サーバーデフォルトプロパティ: log.index.size.max.bytes

のインポート機能: medium

この設定は、オフセットをファイルの位置にマッピングするインデックスのサイズを制御します。このインデックスファイルを事前割り当て、ログロール後にのみ縮小します。通常、この設定を変更する必要はありません。

segment.jitter.ms

type: long

デフォルト: 0

有効な値: [0,...]

サーバーデフォルトプロパティ: log.roll.jitter.ms

インポートランス: medium

セグメントのローリング継承を防ぐために、スケジュールされたセグメントロール時間からランダムな差し引いた最大値。

segment.ms

type: long

Default: 604800000(7 days)

Valid values: [1,...]

Server Default Property: log.roll.ms

Importance: medium

この設定により、保持が古いデータを削除/圧縮できるように、セグメントファイルが一杯でない場合は、Kafka がログのロールを強制的に実行する期間を制御します。

unclean.leader.election.enable

type: boolean

Default: false

サーバーデフォルトプロパティ: `unclean.leader.election.enable`

Importance: medium

ISR セットにないレプリカを最後の手段としてリーダーとして選択したくない場合は、データが失われる可能性があります。

message.downconversion.enable

type: boolean

Default: true

サーバーデフォルトプロパティ: `log.message.downconversion.enable`

Importance: low

この設定は、要求を満たすためにメッセージ形式のダウンコンバージョンを有効にするかどうかを制御します。**false** に設定すると、ブローカーは古いメッセージ形式を想定しているコンシューマーにダウンコンバートを実行しません。ブローカーは、このような古いクライアントからのリクエストを消費するために **UNSUPPORTED_VERSION** エラーで応答します。この設定は、レプリケーションがフォロワーに必要となる可能性があるメッセージ形式の変換には適用されません。

付録C コンシューマー設定パラメーター

key.deserializer

type: class

Importance: high

org.apache.kafka.common.serialization.Deserializer インターフェースを実装するキーのデシリアライザークラス。

value.deserializer

type: class

Importance: high

org.apache.kafka.common.serialization.Deserializer インターフェースを実装する値のデシリアライザークラス。

bootstrap.servers

type: list

デフォルト: ""

Valid Values: non-null string

Importance: high

Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ここで指定するすべてのサーバーの使用を行います。このリストは、サーバーのフルセットを検出するために使用される初期ホストにのみ影響します。このリストは

host1:port1,host2:port2,... の形式にする必要があります。これらのサーバーは、クラスターの全メンバーシップを検出するために初期接続（動的に変更されている可能性がある）にだけ使用されているため、この一覧にはサーバーのフルセットを含める必要はありません（サーバーがダウンした場合の場合もあります）。

fetch.min.bytes

type: int

Default: 1

Valid Values: [0,...]

Importance: high

サーバーがフェッチリクエストに対して返すデータの最小量。十分なデータがない場合には、要求に応答する前に、要求がその大量のデータが累積されるのを待機します。デフォルト設定が1バイトで、データ1バイトが利用可能になり次第、フェッチリクエストに応答が答えるか、またはデータに到達するのを待っていると、フェッチ要求がタイムアウトになります。これを1より大きい値に設定すると、サーバーが大量のデータが累積され、追加のレイテンシーのコストでサーバーのスループットを向上できます。

group.id

type: string

Default: null

Importance: high

このコンシューマーが属するコンシューマーグループを識別する一意の文字列。このプロパティは、**subscribe(topic)** または Kafka ベースのオフセット管理ストラテジーを使用して、コンシューマーがグループ管理機能を使用する場合に必要なになります。

heartbeat.interval.ms

type: int

デフォルト: 3000 (3 秒)

インポートランス: high

Kafka のグループ管理機能を使用する場合に、コンシューマーコーディネーターへのハートビートの

時間が予想されます。ハートビートは、コンシューマーのセッションがアクティブで保ち、新しいコンシューマーに参加したり、グループを退出したりする際のリバランスを容易にするために使用されます。値は **session.timeout.ms** 未満に設定する必要がありますが、通常はこの値の1/3 よりも大きくすることはできません。通常のリバランスの予想される時間を制御するために、低く調整できます。

max.partition.fetch.bytes

type: int

Default: 1048576(1 mebibyte)

Valid Values: [0,...]

Importance: high

サーバーが返すパーティションごとのデータの最大量。レコードはコンシューマーによってバッチで取得されます。フェッチの最初の空でないパーティションの最初のレコードバッチがこの制限よりも大きい場合、コンシューマーが進めるようにバッチが引き続き返されます。ブローカーが受け入れる最大レコードバッチサイズは、**message.max.bytes**（ブローカー設定）または **max.message.bytes**（トピック設定）で定義されます。コンシューマー要求サイズを制限する場合は、**fetch.max.bytes** を参照してください。

session.timeout.ms

type: int

Default: 10000（10 秒）

重要性: high

Kafka のグループ管理機能の使用時にクライアント障害を検出するために使用されるタイムアウト。クライアントは定期的なハートビートを送信し、そのlivenessをブローカーを示します。このセッションタイムアウトの期限が切れる前にブローカーによってハートビートが受信されなかった場合、ブローカーはこのクライアントをグループから削除し、リバランスを開始します。この値は、**group.min.session.timeout.ms** および **group.max.session.timeout.ms** によってブローカー設定で設定される許容範囲である必要があります。

ssl.key.password

type: password

Default: null

Importance: high

キーストアファイルまたは 'ssl.keystore.key' で指定された PEM キーの秘密鍵のパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書のリストを持つ PEM 形式のみをサポートします。

ssl.keystore.key

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の秘密鍵デフォルトの SSL エンジンファクトリーは、PKCS#8 キーを持つ PEM 形式のみをサポートします。キーが暗号化されている場合は、'ssl.key.password' を使用してキーパスワードを指定する必要があります。

ssl.keystore.location

type: string

Default: null

Importance: high

キーストアファイルの場所。これはクライアントにはオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password

Default: null

Importance: high

キーストアファイルのストアパスワード。これはクライアントでは任意で、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアのパスワードは PEM 形式ではサポートされていません。

ssl.truststore.certificates

type: password

Default: null

Importance: high

'ssl.truststore.type' で指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書を使用した PEM 形式のみをサポートします。

ssl.truststore.location

type: string

Default: null

Importance: high

トラストストアファイルの場所。

ssl.truststore.password

type: password

Default: null

Importance: high

トラストストアファイルのパスワード。パスワードが設定されていない場合は、設定されたトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアのパスワードは PEM 形式ではサポートされていません。

allow.auto.create.topics

type: boolean

Default: true

Importance: medium

トピックにサブスクライブしたり、トピックを割り当てる際に、ブローカーの自動トピック作成を許可します。サブスクライブされるトピックは、ブローカーが **auto.create.topics.enable** ブローカー設定を使用できるようにする場合にのみ自動的に作成されます。0.11.0 よりも古いブローカーを使用する場合は、この設定を **false** に設定する必要があります。

auto.offset.reset

type: string

Default: latest

Valid Values: [latest, earliest, none]

Importance: medium

Kafka に初期オフセットがない場合や、現在のオフセットがサーバーにこれ以上存在しない場合（データが削除された場合）：

- **earliest**: オフセットを最も早いオフセットに自動的にリセット
- **latest**: オフセットを最新のオフセットに自動的にリセットする
- **None**: コンシューマーのグループに以前のオフセットが見つからない場合に、コンシューマーに例外をスローします。
- **any else**: コンシューマーに例外をスローします。

client.dns.lookup

type: string

デフォルト: use_all_dns_ips

有効な値: [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]

インポートランス: medium

クライアントがDNS ルックアップを使用する方法を制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスを順番に接続します。接続が切断されると、次の IP が使用されます。すべての IP が1回使用されると、クライアントはホスト名から IP を再度解決します (JVM と OS キャッシュ DNS 名ルックアップの両方)。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズの後に、これは **use_all_dns_ips** と同じように動作します。**default** (非推奨) に設定すると、ルックアップが複数の IP アドレスを返す場合でも、ルックアップによって返される最初の IP アドレスへの接続を試行します。

connections.max.idle.ms

type: long

デフォルト: 540000 (9 分)

Importance: medium

この設定によって指定された期間 (ミリ秒単位) の後にアイドル状態の接続を閉じます。

default.api.timeout.ms

type: int

Default: 60000 (1 分)

有効な値: [0,...]

Importance: medium

クライアント API のタイムアウト (ミリ秒単位) を指定します。この設定は、**timeout** パラメーターを指定しないすべてのクライアント操作のデフォルトタイムアウトとして使用されます。

enable.auto.commit

type: boolean

Default: true

Importance: medium

true の場合、コンシューマーのオフセットはバックグラウンドで定期的にコミットされます。

exclude.internal.topics

type: boolean

Default: true

Importance: medium

サブスクライブしているパターンに一致する内部トピックをサブスクリプションから除外するかどうか。常に内部トピックへの明示的なサブスクライブを行うことができます。

fetch.max.bytes

type: int

Default: 52428800(50 mebibytes)

Valid Values: [0,...]

Importance: medium

サーバーがフェッチリクエストに対して返すデータの最大量。レコードはコンシューマーによってバッチでフェッチされ、フェッチ以外のパーティションの最初のレコードバッチがこの値よりも大きい場合、レコードバッチが引き続き返され、コンシューマーが進捗を確実にすることができません。そのため、これは絶対最大値ではありません。ブローカーが受け入れる最大レコードバッチサイズは、**message.max.bytes**（ブローカー設定）または **max.message.bytes**（トピック設定）で定義されます。コンシューマーは複数のフェッチを並行して実行することに注意してください。

group.instance.id

type: string

Default: null

Importance: medium

エンドユーザーが提供するコンシューマーインスタンスの一意識別子。空でない文字列のみが許可されます。設定されている場合、コンシューマーは静的メンバーとして処理されます。これは、このIDを持つ1つのインスタンスのみが、いつでもコンシューマーグループで許可されます。これは、大規模なセッションタイムアウトと組み合わせることで、一時的な利用停止（プロセスの再起動など）によって生じるグループのリバランスを防ぎます。設定されていない場合、コンシューマーは従来の動作である動的メンバーとしてグループに参加します。

isolation.level

type: string

Default: read_uncommitted

Valid Values: [read_committed, read_uncommitted]

Importance: medium

トランザクションが書き込まれたメッセージの読み取り方法を制御します。**read_committed** に設定された場合、`consumer.poll()` はコミットされたトランザクションメッセージのみを返します。**read_uncommitted**（デフォルト）に設定すると、`consumer.poll()` は中断されたトランザクションメッセージもすべてのメッセージを返します。非トランザクションメッセージは、いずれのモードでも無条件で返されます。

メッセージは常にオフセット順序で返されます。したがって、**read_committed** モードでは、`consumer.poll()` は、最初のオープントランザクションのオフセットよりも1つ小さい最後の安定したオフセット(LSO)までのメッセージのみを返します。特に、継続中のトランザクションに属するメッセージの後に表示されるメッセージは、関連するトランザクションが完了するまでheldになります。その結果、**read_committed** 利用者は、フライトトランザクションがある場合、高基準値を読み取ることができません。

Further, when in `read_committed` the `seekToEnd` method will return the LSO.

max.poll.interval.ms

type: int

Default: 300000 (5 分)

Valid Values: [1,...]

Importance: medium

コンシューマーグループ管理使用時の `poll()` の呼び出し間隔の最大遅延。これにより、より多くのレコードを取得する前にコンシューマーがアイドル状態でいられる期間の上限が設定されます。このタイムアウトの有効期限前に `poll()` が呼び出されなかった場合、コンシューマーは失敗したと見なされ、パーティションを別のメンバーに再割り当てするために、グループはリバランスされます。このタイムアウトに達する null 以外の **group.instance.id** を使用するコンシューマーの場合、

パーティションは即座に再割り当てされません。代わりに、コンシューマーはハートビートとパーティションの送信を停止します。**session.timeout.ms**の有効期限の後に再割り当てされます。これにより、シャットダウンのある静的コンシューマーの動作がミラーリングされます。

max.poll.records

type: int

Default: 500

Valid Values: [1,...]

Importance: medium

poll () に対する単一の呼び出しで返される最大レコード数。**max.poll.records**は、基礎となる動作には影響しません。コンシューマーは各フェッチリクエストからレコードをキャッシュし、各ポーリングからレコードを段階的に返します。

partition.assignment.strategy

type: list

Default: class org.apache.kafka.clients.consumer.RangeAssignor

Valid Values: non-null string

Importance: medium

グループ管理が使用される場合に、クライアントがコンシューマーインスタンス間でパーティション所有権を分散するために使用するサポート対象のパーティション割り当てストラテジーのクラス名またはクラスタイプのリスト。利用可能なオプションは以下の通りです。

- **org.apache.kafka.clients.consumer.RangeAssignor:** デフォルトのアクセサー。これはトピックごとに動作します。
- **org.apache.kafka.clients.consumer.RoundRobinAssignor:** ラウンドロビン方式でパーティションをコンシューマーに割り当てます。
- **org.apache.kafka.clients.consumer.StickyAssignor:** 既存のパーティションの割り当てをできるだけ多く維持しながら、最大限に分散される割り当てを保証します。
- **org.apache.kafka.clients.consumer.CooperativeStickyAssignor:** 同じ StickyAssignor ロジックに従いますが、cooperative Rebalancing が可能になります。
org.apache.kafka.clients.consumer.ConsumerPartitionAssignor インターフェースを実装すると、カスタム割り当てストラテジーをプラグインできます。

receive.buffer.bytes

type: int

Default: 65536(64 kibibytes)

Valid Values: [-1,...]

Importance: medium

データの読み取り時に使用する TCP 受信バッファ (SO_RCVBUF) のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

type: int

デフォルト: 30000 (30 秒)

有効な値: [0,...]

Importance: medium

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。

sasl.client.callback.handler.class**type:** class**Default:** null**Importance:** medium

AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config**type:** password**Default:** null**Importance:** medium

JAAS 設定ファイルによって使用される形式で、SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式は、[こちらで説明されています](#)。値のフォーマットは

loginModuleClass controlFlag (optionName=optionValue)*; です。ブローカーの場合、設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。例：

```
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;
```

sasl.kerberos.service.name**type:** string**Default:** null**Importance:** medium

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定のいずれかで定義できます。

sasl.login.callback.handler.class**type:** class**Default:** null**Importance:** medium

AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラー設定の前にリスナープレフィックスと SASL メカニズム名（小文字の）を付ける必要があります。例：

```
listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler.
```

sasl.login.class**type:** class**Default:** null**Importance:** medium

Login インターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。For example,

```
listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.
```

sasl.mechanism**type:** string**Default:** GSSAPI**Importance:** medium

クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーが利用できるメカニズムになることがあります。GSSAPI がデフォルトのメカニズムです。

security.protocol

type: string

Default: PLAINTEXT

Importance: medium

ブローカーとの通信に使用されるプロトコル。有効な値はPLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

type: int

Default: 131072(128 kibibytes)

Valid Values: [-1,...]

Importance: medium

データ送信時に使用する TCP 送信バッファ(SO_SNDBUF)のサイズ。値が-1 の場合、OS のデフォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ: long

デフォルト: 30000 (30 秒)

インポート: 中

クライアントがソケット接続を確立するまで待機する最大時間。接続設定のタイムアウトにより、連続する接続の失敗ごとに指数関数的に増加します。接続タイムアウトを回避するために、ランダム化係数 0.2 がタイムアウトに適用されるため、以下の 20% と計算された値の 20% のランダムな範囲が適用されます。

socket.connection.setup.timeout.ms

タイプ: long

デフォルト: 10000 (10 秒)

重要: medium

クライアントがソケット接続を確立するのを待機する期間。タイムアウトが経過する前に接続がビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list

Default: TLSv1.2,TLSv1.3

Importance: medium

SSL 接続に対して有効なプロトコル一覧。Java 11 以降、「TLSv1.2」以降で実行する場合、デフォルトは'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーの両方が TLSv1.2 に対応している場合は、クライアントとサーバーは TLSv1.3 を優先します (TLSv1.2 以上をサポートすることを想定します)。多くのケースでは、このデフォルトは問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

type: string

Default: JKS

Importance: medium

キーストアファイルのファイル形式。これはクライアントの場合はオプションになります。

ssl.protocol

type: string

Default: TLSv1.3

importance: medium

SSLContext の生成に使用される SSL プロトコル。Java 11 以降「TLSv1.2」を使用して実行すると、デフォルトは「TLSv1.3」です。この値は、ほとんどのユースケースで十分です。最近の JVM で許

可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされる可能性があります、既知のセキュリティ脆弱性が原因で使用は推奨されません。この設定および 'ssl.enabled.protocols' のデフォルト値を使用すると、サーバーが「TLSv1.3」をサポートしない場合、クライアントは 'TLSv1.2' にダウングレードします。この設定を「TLSv1.2」に設定すると、クライアントは ssl.enabled.protocols の値のいずれかである場合でも、「TLSv1.3」を使用し、サーバーは「TLSv1.3」のみに対応します。

ssl.provider

type: string

Default: null

Importance: medium

SSL 接続に使用されるセキュリティプロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティプロバイダーです。

ssl.truststore.type

type: string

Default: JKS

Importance: medium

トラストストアファイルのファイル形式です。

auto.commit.interval.ms

type: int

Default: 5000(5 seconds)

有効な値: [0,...]

Importance: low

enable.auto.commit が **true** に設定されている場合、コンシューマーオフセットが Kafka に自動で許可される頻度（ミリ秒単位）。

check.crcs

型: ブール値

デフォルト: true

の修正: low

消費したレコードの CRC32 を自動的にチェックします。これにより、メッセージへの伝送時またはディスク上の破損が発生しないようにします。このチェックによりオーバーヘッドが発生する可能性があるため、パフォーマンスが非常に高い場合は無効にすることができます。

client.id

type: string

デフォルト: ""

Importance: low

要求の実行時にサーバーに渡す id 文字列。この目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/ポート以外の要求のソースを追跡できるようにすることです。

client.rack

type: string

デフォルト: ""

Importance: low

このクライアントのラック識別子。このクライアントが物理的に置かれる場所を示す文字列の値を使用できます。ブローカー設定 'broker.rack' に対応します。

fetch.max.wait.ms

type: int

Default: 500

Valid Values: [0,...]

Importance: low

fetch.min.bytes で指定された要件を即座に満たすのに十分なデータがない場合に、サーバーがフェッチリクエストに応答するまでにサーバーがブロックする最大時間。

interceptor.classes

type: list

デフォルト: ""

Valid Values: non-null string

Importance: low

インターセプターとして使用するクラスのリスト。

org.apache.kafka.clients.consumer.ConsumerInterceptor インターフェースを実装すると、コンシューマーによって受信されるレコードを傍受（また変更する可能性があります）できます。デフォルトでは、インターセプターはありません。

metadata.max.age.ms

type: long

デフォルト: 300000 (5 分)

有効な値: [0,...]

Importance: low

パーティションリーダーが変更されておらず、新しいブローカーやパーティションをプロアクティブに検出するようにしていても、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

type: list

デフォルト: ""

Valid Values: non-null string

Importance: low

メトリクスレポーターとして使用するクラスの一覧。

org.apache.kafka.common.metrics.MetricsReporter インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。JmxReporter は、JMX 統計を登録するために常に含まれます。

metrics.num.samples

type: int

Default: 2

Valid Values: [1,...]

Importance: low

メトリクスを計算するために保持されるサンプルの数。

metrics.recording.level

type: string

Default: INFO

Valid Values: [INFO, DEBUG, TRACE]

Importance: low

メトリックの最大記録レベル。

metrics.sample.window.ms

type: long

デフォルト: 30000 (30 秒)

有効値 : [0,...]

Importance: low

メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms

type: long

デフォルト : 1000 (1秒)

有効値 : [0,...]

Importance: low

繰り返し接続に失敗したブローカーに再接続するまで待機する最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは連続した接続障害ごとに指数関数的に増加し、最大接続数までします。バックオフの増加を計算すると、接続のフレッターを回避するために20%のランダムジッターが追加されます。

reconnect.backoff.ms

type: long

デフォルト : 50

有効な値 : [0,...]

インポートランス : low

指定のホストに再接続を試みる前に待機する時間。これにより、密接なループでホストへ繰り返し接続することはできません。このバックオフは、クライアントがブローカーへのすべての接続試行に適用されます。

retry.backoff.ms

type: long

デフォルト : 100

有効な値 : [0,...]

インポートランス : low

特定のトピックパーティションへの失敗した要求を再試行するまでの待機時間。これにより、障害シナリオによっては、1回目のループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

type: string

デフォルト : /usr/bin/kinit

Importance: low

Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ : long

デフォルト : 60000

インポートランス : low

更新試行の間には、ログインスレッドのスリープ時間。

sasl.kerberos.ticket.renew.jitter

type: double

Default: 0.05

Importance: low

ランダムなジッターの割合が更新時間に追加されます。

sasl.kerberos.ticket.renew.window.factor

type: double

デフォルト : 0.8

クォーラム : low

指定のウィンドウ係数からチケットの有効期限に達するまで、ログインスレッドはスリープ状態になります。

sasl.login.refresh.buffer.seconds

type: short

Default: 300

Valid Values:[0,...,3600]

Importance: low

認証情報の更新時における認証情報が失効するまでの時間（秒単位）。更新が行われない場合、バッファの秒数より期限切れになり始めたら、更新ができるだけ多くのバッファ時間を維持します。有効な値は 0 から 3600（1 時間）の間です。値が指定されていない場合は、デフォルト値の 300（5 分）が使用されます。認証情報の残存期間が経過すると、この値と sasl.login.refresh.min.period.seconds はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

type: short

Default: 60

Valid Values:[0,...,900]

Importance: low

ログイン更新スレッドがクレデンシャルを更新する前に待機する最低時間（秒単位）。有効な値は 0 から 900（15 分）までの値になります。値が指定されていない場合は、デフォルト値の 60（1 分）が使用されます。認証情報の残存期間が経過すると、この値と sasl.login.refresh.buffer.seconds はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

type: double

Default: 0.8

Valid Values:[0.5,...,1.0]

Importance: low

ログイン更新スレッドは、認証情報の有効期間と相対的な期間係数に達するまでスリープします。この場合、認証情報の更新を試みます。有効な値は 0.5(50%)と 1.0(100%)です。値が指定されていない場合、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

type: double

Default: 0.05

Valid Values:[0.0,...,0.25]

Importance: low

ログイン更新スレッドのスリープ時間に追加されたクレデンシャルの有効期間に対するランダムなジッターの最大数。有効な値は 0 から 0.25(25%)までの値になります。値が指定されていない場合は、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers

type: string

Default: null

Importance: low

セキュリティーアルゴリズムを実装するプロバイダーを返す設定可能な作成者クラスのリスト。これらのクラスは **org.apache.kafka.common.security.auth.SecurityProviderCreator** インターフェースを実装する必要があります。

ssl.cipher.suites

type: list

Default: null

Importance: low

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC およびキー交換アルゴリズムの名前付き組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.endpoint.identification.algorithm

タイプ: 文字列

デフォルト: https

インポートランス: low

サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class

Default: null

Importance: low

種別 `org.apache.kafka.common.security.auth.SslEngineFactory` のクラスで `SslEngine` オブジェクトを提供します。Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`.

ssl.keymanager.algorithm

type: string

Default: SunX509

Importance: low

SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

type: string

Default: null

Importance: low

SSL 暗号化操作に使用する `SecureRandom` PRNG 実装。

ssl.trustmanager.algorithm

type: string

デフォルト: PKIX

重要性: low

SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

付録D プロデューサー設定パラメーター

key.serializer

type: class

Importance: high

org.apache.kafka.common.serialization.Serializer インターフェースを実装するキーのシリアルライザークラス。

value.serializer

type: class

Importance: high

org.apache.kafka.common.serialization.Serializer インターフェースを実装する値のシリアルライザークラス。

acks

type: string

デフォルト : 1

有効値 : [all, -1, 0, 1]

インポートランス : high

リクエストの完了を検討する前に、プロデューサーにリーダーを受信する必要がある確認の数。これにより、送信されるレコードの持続性を制御します。以下の設定が可能です。

- **acks=0** ゼロに設定すると、プロデューサーはサーバーからの確認応答を待機しません。レコードはソケットバッファに即座に追加され、送信されたとみなされます。この場合、サーバーがレコードを受け取れる保証はなく、**retries** 設定が適用されません（クライアントは一般的に障害を認識しません）。各レコードに対して返されるオフセットは常に **-1** に設定されます。
- **acks=1** つまり、リーダーはレコードをローカルログに書き込みますが、すべてのフォロワーからの完全な確認を待たずに応答します。この場合、レコードの完了直後、フォロワーが複製される前にリーダーは失敗すると、レコードが失われます。
- **acks=all** つまり、リーダーは in-sync レプリカの完全なセットがレコードを認識するのを待機します。これにより、sync レプリカが1 つ以上存続しない限り、レコードが失われなくなります。これは、最も強力な保証です。これは **acks=-1** 設定と同じです。

bootstrap.servers

type: list

デフォルト : ""

Valid Values: non-null string

Importance: high

Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ここで指定するすべてのサーバーの使用を行います。このリストは、サーバーのフルセットを検出するために使用される初期ホストにのみ影響します。このリストは

host1:port1,host2:port2,... の形式にする必要があります。これらのサーバーは、クラスターの全メンバーシップを検出するために初期接続（動的に変更されている可能性がある）にだけ使用されているため、この一覧にはサーバーのフルセットを含める必要はありません（サーバーがダウンした場合の場合もあります）。

buffer.memory

type: long

デフォルト : 33554432

有効な値 : [0,...]

Importance: high

プロデューサーがサーバーに送信されるまで待機するレコードをバッファースするために使用できるメモリーの合計バイト数。レコードがサーバーへ配信されるよりも高速にレコードを送信する場合、プロデューサーは **max.block.ms** のブロックを行い、例外が発生します。

この設定は、プロデューサーが使用する合計メモリーに対応する必要がありますが、プロデューサーが使用するすべてのメモリーはバッファースに使用される訳ではなく、ハードバインドされません。追加のメモリーは圧縮（圧縮が有効になっている場合）やインフライトリクエストを維持するために使用されます。

compression.type

type: string

デフォルト: none

の修正: high

プロデューサーによって生成された全データの圧縮タイプ。デフォルトは none（つまり圧縮なし）です。有効な値は、**none**、**gzip**、**snappy**、**lz4**、または **zstd** です。圧縮はデータの完全なバッチであるため、バッチ処理は圧縮比率にも影響を及ぼします（バッチ処理により、圧縮が向上する）。

retries

type: int

Default: 2147483647

Valid Values: [0,...,2147483647]

Importance: high

ゼロよりも大きい値を設定すると、クライアントは一時的なエラーで送信に失敗するレコードを再送します。この再試行は、クライアントのエラーの受信時にレコードを再送する場合とは異なります。**max.in.flight.requests.per.connection** を 1 に設定せずに再試行を許可すると、レコードの順序が変更される可能性があります。これは、1つのバッチが単一のパーティションに送信され、最初のバッチで再試行され、2番目のバッチのレコードが最初に表示されるためです。また、確認応答の成功前に **delivery.timeout.ms** によってタイムアウトが期限切れになると、再試行回数が使い切られる前にリクエストが失敗することに注意してください。通常は、この設定の設定を解除したままにし、再試行動作を制御する代わりに **delivery.timeout.ms** を使用することが推奨されます。

ssl.key.password

type: password

Default: null

Importance: high

キーストアファイルまたは 'ssl.keystore.key' で指定された PEM キーの秘密鍵のパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書のリストを持つ PEM 形式のみをサポートします。

ssl.keystore.key

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の秘密鍵デフォルトの SSL エンジンファクトリーは、PKCS#8 キーを持つ PEM 形式のみをサポートします。キーが暗号化されている場合は、'ssl.key.password' を使用してキーパスワードを指定する必要があります。

ssl.keystore.location

type: string

Default: null

Importance: high

キーストアファイルの場所。これはクライアントにはオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password

Default: null

Importance: high

キーストアファイルのストアパスワード。これはクライアントでは任意で、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアのパスワードは PEM 形式ではサポートされていません。

ssl.truststore.certificates

type: password

Default: null

Importance: high

'ssl.truststore.type' で指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書を使用した PEM 形式のみをサポートします。

ssl.truststore.location

type: string

Default: null

Importance: high

トラストストアファイルの場所。

ssl.truststore.password

type: password

Default: null

Importance: high

トラストストアファイルのパスワード。パスワードが設定されていない場合は、設定されたトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアのパスワードは PEM 形式ではサポートされていません。

batch.size

type: int

Default: 16384

Valid Values: [0,...]

Importance: medium

プロデューサーは、複数のレコードが同じパーティションに送信されるたびに、バッチレコードを少ないリクエストにまとめよう試行します。これにより、クライアントとサーバーの両方でのパフォーマンスが容易になります。この設定では、デフォルトのバッチサイズをバイト単位で制御します。

このサイズを超えるバッチレコードをバッチ処理しようとしません。

ブローカーに送信されたリクエストには、複数のバッチが含まれます。各パーティションに1つずつ、送信できるデータがあります。

バッチサイズが小さくなると、バッチ処理が少なくなりますが、スループットが減少することがあ

ります（バッチサイズがゼロの場合、バッチサイズは完全に無効になります）。非常に大きなバッチサイズでは、追加のレコードで指定のバッチサイズのバッファを常に割り当てるため、メモリーを少し進む可能性があります。

client.dns.lookup

type: string

デフォルト: use_all_dns_ips

有効な値: [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]

インポートランス: medium

クライアントがDNS ルックアップを使用する方法を制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスを順番に接続します。接続が切断されると、次の IP が使用されます。すべての IP が1回使用されると、クライアントはホスト名から IP を再度解決します（JVM と OS キャッシュ DNS 名ルックアップの両方）。**resolve_canonical_bootstrap_servers_only** に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズの後に、これは **use_all_dns_ips** と同じように動作します。**default**（非推奨）に設定すると、ルックアップが複数の IP アドレスを返す場合でも、ルックアップによって返される最初の IP アドレスへの接続を試行します。

client.id

type: string

デフォルト: ""

Importance: medium

要求の実行時にサーバーに渡す id 文字列。この目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/ポート以外の要求のソースを追跡できるようにすることです。

connections.max.idle.ms

type: long

デフォルト: 540000（9 分）

Importance: medium

この設定によって指定された期間（ミリ秒単位）の後にアイドル状態の接続を閉じます。

delivery.timeout.ms

type: int

Default: 120000（2 分）

有効な値: [0,...]

Importance: medium

send() の呼び出し後に成功または失敗を報告する時間の上限です。これにより、送信前にレコードが遅延する合計時間、ブローカーからの確認応答の待機中に時間、および再試行可能な送信失敗に許可される時間を制限します。プロデューサーは、リカバリー不可能なエラーが発生した場合に設定の前にレコードを送信すること、再試行が使い切られた場合、または以前の配信期限に達したバッチにレコードが追加される可能性があります。この設定の値は、**request.timeout.ms** および **linger.ms** の合計以上である必要があります。

linger.ms

type: long

デフォルト: 0

有効値: [0,...]

Importance: medium

プロデューサーグループは、リクエストの送信間で到達するレコードを1つのバッチ処理リクエストにまとめます。通常、レコードが送信速度よりも早く到達した場合にのみ、負荷がかかります。ただし、状況によっては、クライアントが中程度の負荷でもリクエストの数を減らす必要がある場合があります。この設定は、レコードを即座に送信するのではなく、プロデューサーが指定の遅延を待機して、送信をバッチ処理できるようにすることで、人為的な遅延量を若干追加します。これ

は、TCP の Nagle アルゴリズムに類似したと考えることができます。この設定により、バッチ処理の遅延の上限があります。この設定では、この設定に関係なく、パーティションにレコードが1人ずつ送信されます。ただし、このパーティションに対して累積されたバイト数がこの値より少ない場合は、レコードが表示されるまで指定した時間に対する「残高」を「残します」します。**batch.size**この設定は、デフォルトで0（遅延なし）です。**linger.ms=5**を設定すると、送信される要求の数を減らすことができますが、負荷がない場合に送信されたレコードに最大5ms が追加されます。

max.block.ms

type: long

デフォルト : 60000 (1分)

有効な値 : [0,...]

Importance: medium

この設定では、**KafkaProducer's**

send()、**partitionsFor()**、**initTransactions()**、**sendOffsetsToTransaction()**、**commitTransaction()**、および**abortTransaction()** メソッドがブロックする期間を制御します。**send()** の場合、このタイムアウトはメタデータフェッチとバッファ割り当ての待機中の合計時間をバインドします（ユーザー指定のシリアライザーやパーティションヤーでブロックはこのタイムアウトに対してカウントされません）。**partitionsFor()** の場合、このタイムアウトにより、メタデータの待機に費やされた時間を、利用できない場合にバインドします。トランザクション関連のメソッドは常にブロックしますが、トランザクションコーディネーターを検出できなかったり、タイムアウト内で応答できなかった場合はタイムアウトする場合があります。

max.request.size

type: int

Default: 1048576

有効な値 : [0,...]

Importance: medium

リクエストの最大サイズ（バイト単位）。この設定により、リクエストの送信を防ぐためにプロデューサーが1つのリクエストに送信するレコードバッチの数が制限されます。これは、圧縮されていない最大のレコードバッチサイズの上限です。この場合、レコードバッチサイズでサーバーには独自の上限が設定されている点に注意してください（圧縮が有効になっている場合は圧縮後）。

partitioner.class

type: class

Default: org.apache.kafka.clients.producer.internals.DefaultPartitioner

Importance: medium

org.apache.kafka.clients.producer.Partitioner インターフェースを実装するパーティションクラス。

receive.buffer.bytes

type: int

Default: 32768(32 kibibytes)

Valid Values: [-1,...]

Importance: medium

データの読み取り時に使用する TCP 受信バッファ(SO_RCVBUF)のサイズ。値が-1の場合、OS のデフォルトが使用されます。

request.timeout.ms

type: int

デフォルト : 30000 (30 秒)

有効な値 : [0,...]

Importance: medium

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。これは、不要なプロデューサーの再試行によりメッセージの重複の可能性を低減するために、**replica.lag.time.max.ms**（ブローカーの設定）より大きくする必要があります。

sasl.client.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

type: password

Default: null

Importance: medium

JAAS 設定ファイルによって使用される形式で、SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式は、[こちらで説明されています](#)。値のフォーマットは

loginModuleClass controlFlag (optionName=optionValue)*; です。ブローカーの場合、設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。例：

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string

Default: null

Importance: medium

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定のいずれかで定義できます。

sasl.login.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラー設定の前にリスナープレフィックスと SASL メカニズム名（小文字の）を付ける必要があります。例：

listener.name.sasl_ssl.scram-sha-

256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler。

sasl.login.class

type: class

Default: null

Importance: medium

Login インターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。For example,

listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin。

sasl.mechanism

type: string

Default: GSSAPI

Importance: medium

クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーが利用できるメカニズムになることがあります。GSSAPI がデフォルトのメカニズムです。

security.protocol

type: string

Default: PLAINTEXT

Importance: medium

ブローカーとの通信に使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

type: int

Default: 131072(128 kibibytes)

Valid Values: [-1,...]

Importance: medium

データ送信時に使用する TCP 送信バッファ(SO_SNDBUF)のサイズ。値が-1 の場合、OS のデフォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ: long

デフォルト: 30000 (30 秒)

インポート: 中

クライアントがソケット接続を確立するまで待機する最大時間。接続設定のタイムアウトにより、連続する接続の失敗ごとに指数関数的に増加します。接続タイムアウトを回避するために、ランダム化係数 0.2 がタイムアウトに適用されるため、以下の 20% と計算された値の 20% のランダムな範囲が適用されます。

socket.connection.setup.timeout.ms

タイプ: long

デフォルト: 10000 (10 秒)

重要: medium

クライアントがソケット接続を確立するのを待機する期間。タイムアウトが経過する前に接続がビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list

Default: TLSv1.2,TLSv1.3

Importance: medium

SSL 接続に対して有効なプロトコル一覧。Java 11 以降、「TLSv1.2」以降で実行する場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーの両方が TLSv1.2 に対応している場合は、クライアントとサーバーは TLSv1.3 を優先します (TLSv1.2 以上をサポートすることを想定します)。多くのケースでは、このデフォルトは問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type

type: string

Default: JKS

Importance: medium

キーストアファイルのファイル形式。これはクライアントの場合はオプションになります。

ssl.protocol

type: string**Default:** TLSv1.3**importance:** medium

SSLContext の生成に使用される SSL プロトコル。Java 11 以降「TLSv1.2」を使用して実行すると、デフォルトは「TLSv1.3」です。この値は、ほとんどのユースケースで十分です。最近の JVM で許可される値は「TLSv1.2」および「TLSv1.3」です。「TLS」、「TLSv1.1」、「SSL」、「SSLv2」、および「SSLv3」は古い JVM でサポートされる可能性があります、既知のセキュリティ脆弱性が原因で使用は推奨されません。この設定および「ssl.enabled.protocols」のデフォルト値を使用すると、サーバーが「TLSv1.3」をサポートしない場合、クライアントは「TLSv1.2」にダウングレードします。この設定を「TLSv1.2」に設定すると、クライアントは ssl.enabled.protocols の値のいずれかである場合でも、「TLSv1.3」を使用し、サーバーは「TLSv1.3」のみに対応します。

ssl.provider**type:** string**Default:** null**Importance:** medium

SSL 接続に使用されるセキュリティプロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティプロバイダーです。

ssl.truststore.type**type:** string**Default:** JKS**Importance:** medium

トラストストアファイルのファイル形式です。

enable.idempotence**type:** boolean**Default:** false**Importance:** low

'true' に設定すると、プロデューサーは各メッセージの1つのコピーがストリームに書き込まれるようにします。'false'、ブローカーの失敗によるプロデューサーの再試行回数がある場合、ストリームに再試行されたメッセージの複製が作成される可能性があります。冪等性を有効にするには、**max.in.flight.requests.per.connection** を5以下に設定する必要があります。**retries** は0より大きく、**acks** は「all」である必要があります。これらの値がユーザーによって明示的に設定されていない場合、適切な値が選択されます。互換性のない値が設定されている場合、**ConfigException** はスローされます。

interceptor.classes**type:** list**デフォルト:** ""**Valid Values:** non-null string**Importance:** low

インターセプターとして使用するクラスのリスト。

org.apache.kafka.clients.producer.ProducerInterceptor インターフェースを実装すると、Kafka クラスターにパブリッシュされる前にプロデューサーによって受信されるレコードを傍受（また変更する可能性があります）できます。デフォルトでは、インターセプターはありません。

max.in.flight.requests.per.connection**type:** int**Default:** 5**Valid Values:** [1,...]**Importance:** low

クライアントがブロックする前に1つの接続に送信するリクエストの最大数。この設定が1よりも大きいで送信に失敗する場合は、再試行によるメッセージの再順序のリスクがあります（再試行が有効な場合など）。

metadata.max.age.ms

type: long

デフォルト : 300000 (5 分)

有効な値 : [0,...]

Importance: low

パーティションリーダーが変更されておらず、新しいブローカーやパーティションをプロアクティブに検出するようにしていても、メタデータの更新を強制する期間（ミリ秒単位）。

metadata.max.idle.ms

type: long

デフォルト : 300000 (5 分)

有効値 : [5000,...]

Importance: low

プロデューサーがアイドル状態のトピックのメタデータをキャッシュする期間を制御します。トピックが最後に実行された後の経過時間がメタデータのアイドル期間を超えると、トピックのメタデータは意図されず、次のアクセスでメタデータフェッチリクエストを強制的に実行します。

metric.reporters

type: list

デフォルト : ""

Valid Values: non-null string

Importance: low

メトリクスレポーターとして使用するクラスの一

覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。JmxReporter は、JMX 統計を登録するために常に含まれます。

metrics.num.samples

type: int

Default: 2

Valid Values: [1,...]

Importance: low

メトリクスを計算するために保持されるサンプルの数。

metrics.recording.level

type: string

Default: INFO

Valid Values: [INFO, DEBUG, TRACE]

Importance: low

メトリックの最大記録レベル。

metrics.sample.window.ms

type: long

デフォルト : 30000 (30 秒)

有効値 : [0,...]

Importance: low

メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms**type:** long**デフォルト :** 1000 (1秒)**有効値 :** [0,...]**Importance:** low

繰り返し接続に失敗したブローカーに再接続するまで待機する最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは連続した接続障害ごとに指数関数的に増加し、最大接続数までします。バックオフの増加を計算すると、接続のフレッターを回避するために20%のランダムジッターが追加されます。

reconnect.backoff.ms**type:** long**デフォルト :** 50**有効な値 :** [0,...]**インポートランス :** low

指定のホストに再接続を試みる前に待機する時間。これにより、密接なループでホストへ繰り返し接続することはできません。このバックオフは、クライアントがブローカーへのすべての接続試行に適用されます。

retry.backoff.ms**type:** long**デフォルト :** 100**有効な値 :** [0,...]**インポートランス :** low

特定のトピックパーティションへの失敗した要求を再試行するまでの待機時間。これにより、障害シナリオによっては、1回目のループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd**type:** string**デフォルト :** /usr/bin/kinit**Importance:** low

Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin**タイプ :** long**デフォルト :** 60000**インポートランス :** low

更新試行の間には、ログインスレッドのスリープ時間。

sasl.kerberos.ticket.renew.jitter**type:** double**Default:** 0.05**Importance:** low

ランダムなジッターの割合が更新時間に追加されます。

sasl.kerberos.ticket.renew.window.factor**type:** double**デフォルト :** 0.8**クォーラム :** low

指定のウィンドウ係数からチケットの有効期限に達するまで、ログインスレッドはスリープ状態になります。

sasl.login.refresh.buffer.seconds**type:** short**Default:** 300**Valid Values:**[0,...,3600]**Importance:** low

認証情報の更新時における認証情報が失効するまでの時間（秒単位）。更新が行われない場合、バッファの秒数より期限切れになり始めたら、更新ができるだけ多くのバッファ時間を維持します。有効な値は0 から3600（1時間）の間です。値が指定されていない場合は、デフォルト値の300（5分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.min.period.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds**type:** short**Default:** 60**Valid Values:**[0,...,900]**Importance:** low

ログイン更新スレッドがクレデンシャルを更新する前に待機する最低時間（秒単位）。有効な値は0 から900（15分）までの値になります。値が指定されていない場合は、デフォルト値の60（1分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.buffer.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor**type:** double**Default:** 0.8**Valid Values:**[0.5,...,1.0]**Importance:** low

ログイン更新スレッドは、認証情報の有効期間と相対的な期間係数に達するまでスリープします。この場合、認証情報の更新を試みます。有効な値は0.5(50%)と1.0(100%)です。値が指定されていない場合、デフォルト値の0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter**type:** double**Default:** 0.05**Valid Values:**[0.0,...,0.25]**Importance:** low

ログイン更新スレッドのスリープ時間に追加されたクレデンシャルの有効期間に対するランダムなジッターの最大数。有効な値は0 から0.25(25%)までの値になります。値が指定されていない場合は、デフォルト値の0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers**type:** string**Default:** null**Importance:** low

セキュリティーアルゴリズムを実装するプロバイダーを返す設定可能な作成者クラスのリスト。これらのクラスは `org.apache.kafka.common.security.auth.SecurityProviderCreator` インターフェースを実装する必要があります。

ssl.cipher.suites

type: list

Default: null

Importance: low

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティ設定をネゴシエートするために使用される認証、暗号化、MAC およびキー交換アルゴリズムの名前付き組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.endpoint.identification.algorithm

タイプ: 文字列

デフォルト: https

インポートランス: low

サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class

Default: null

Importance: low

種別 `org.apache.kafka.common.security.auth.SslEngineFactory` のクラスで `SslEngine` オブジェクトを提供します。Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`.

ssl.keymanager.algorithm

type: string

Default: SunX509

Importance: low

SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

type: string

Default: null

Importance: low

SSL 暗号化操作に使用する `SecureRandom` PRNG 実装。

ssl.trustmanager.algorithm

type: string

デフォルト: PKIX

重要性: low

SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

transaction.timeout.ms

type: int

デフォルト: 60000 (1 分)

インポートランス: low

トランザクションのコーディネーターがプロデューサーからトランザクションステータスの更新を待機する最大時間（ミリ秒単位）。この期間を超えると、継続中のトランザクションがプロアクティブに中断されます。この値はブローカー内の `transaction.max.timeout.ms` 設定よりも大きい場合、リクエストは `InvalidTxnTimeoutException` エラーを出して失敗します。

transactional.id

type: string

Default: null

Valid Values: non-empty string

Importance: low

トランザクション配信に使用する TransactionId。これにより、新しいトランザクションの開始前にクライアントが同じ TransactionId を使用するトランザクションが確実に完了するため、複数のプロデューサーセッションにまたがる信頼性のセマンティクスが有効になります。TransactionId が指定されていない場合、プロデューサーはべき等配信に制限されます。TransactionId が設定される場合、**enable.idempotence** は暗示されます。デフォルトでは TransactionId は設定されておらず、トランザクションは使用できません。デフォルトでは、トランザクションには、実稼働用に推奨される設定が3 つ以上あるブローカーで構成されるクラスターが必要になります。開発の場合は、ブローカー設定 **transaction.state.log.replication.factor** を調整することでこれを変更できます。

付録E 管理クライアント設定パラメーター

bootstrap.servers

type: list

Importance: high

Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ここで指定するすべてのサーバーの使用を行います。このリストは、サーバーのフルセットを検出するために使用される初期ホストにのみ影響します。このリストは

host1:port1,host2:port2,... の形式にする必要があります。これらのサーバーは、クラスターの全メンバーシップを検出するために初期接続（動的に変更されている可能性がある）にだけ使用されているため、この一覧にはサーバーのフルセットを含める必要はありません（サーバーがダウンした場合もあります）。

ssl.key.password

type: password

Default: null

Importance: high

キーストアファイルまたは 'ssl.keystore.key' で指定された PEM キーの秘密鍵のパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書のリストを持つ PEM 形式のみをサポートします。

ssl.keystore.key

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の秘密鍵デフォルトの SSL エンジンファクトリーは、PKCS#8 キーを持つ PEM 形式のみをサポートします。キーが暗号化されている場合は、'ssl.key.password' を使用してキーパスワードを指定する必要があります。

ssl.keystore.location

type: string

Default: null

Importance: high

キーストアファイルの場所。これはクライアントにはオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password

Default: null

Importance: high

キーストアファイルのストアパスワード。これはクライアントでは任意で、'ssl.keystore.location' が設定されている場合のみ必要です。キーストアのパスワードは PEM 形式ではサポートされていません。

ssl.truststore.certificates

type: password

Default: null

Importance: high

'ssl.truststore.type' で指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書を使用した PEM 形式のみをサポートします。

ssl.truststore.location

type: string

Default: null

Importance: high

トラストストアファイルの場所。

ssl.truststore.password

type: password

Default: null

Importance: high

トラストストアファイルのパスワード。パスワードが設定されていない場合は、設定されたトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアのパスワードは PEM 形式ではサポートされていません。

client.dns.lookup

type: string

デフォルト : use_all_dns_ips

有効な値 : [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]

インポートランス : medium

クライアントが DNS ルックアップを使用する方法を制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスを順番に接続します。接続が切断されると、次の IP が使用されます。すべての IP が 1 回使用されると、クライアントはホスト名から IP を再度解決します (JVM と OS キャッシュ DNS 名ルックアップの両方)。

resolve_canonical_bootstrap_servers_only に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズの後に、これは **use_all_dns_ips** と同じように動作します。**default** (非推奨) に設定すると、ルックアップが複数の IP アドレスを返す場合でも、ルックアップによって返される最初の IP アドレスへの接続を試行します。

client.id

type: string

デフォルト : ""

Importance: medium

要求の実行時にサーバーに渡す id 文字列。この目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/ポート以外の要求のソースを追跡できるようにすることです。

connections.max.idle.ms

type: long

デフォルト : 300000 (5 分)

Importance: medium

この設定によって指定された期間 (ミリ秒単位) の後にアイドル状態の接続を閉じます。

default.api.timeout.ms

type: int

Default: 60000 (1 分)

有効な値 : [0,...]

Importance: medium

クライアント API のタイムアウト（ミリ秒単位）を指定します。この設定は、**timeout** パラメーターを指定しないすべてのクライアント操作のデフォルトタイムアウトとして使用されます。

receive.buffer.bytes

type: int

Default: 65536(64 kibibytes)

Valid Values: [-1,...]

Importance: medium

データの読み取り時に使用する TCP 受信バッファ (SO_RCVBUF) のサイズ。値が -1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

type: int

デフォルト : 30000 (30 秒)

有効な値 : [0,...]

Importance: medium

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。

sasl.client.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

type: password

Default: null

Importance: medium

JAAS 設定ファイルによって使用される形式で、SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式は、[こちらで説明されています](#)。値のフォーマットは

loginModuleClass controlFlag (optionName=optionValue)*; です。ブローカーの場合、設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。例：

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string

Default: null

Importance: medium

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定のいずれかで定義できます。

sasl.login.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラ設定の前にリスナープレフィックスと SASL メカニズム名（小文字の）を付ける必要があります。例：

`listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler。`

sasl.login.class

type: class

Default: null

Importance: medium

Login インターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリ
スナープレフィックスおよびSASL メカニズム名が付けられます。For example,
`listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin。`

sasl.mechanism

type: string

Default: GSSAPI

Importance: medium

クライアント接続に使用される SASL メカニズム。これは、セキュリティープロバイダーが利用で
きるメカニズムになることがあります。GSSAPI がデフォルトのメカニズムです。

security.protocol

type: string

Default: PLAINTEXT

Importance: medium

ブローカーとの通信に使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、
SASL_SSL です。

send.buffer.bytes

type: int

Default: 131072(128 kibibytes)

Valid Values: [-1,...]

Importance: medium

データ送信時に使用する TCP 送信バッファ(SO_SNDBUF)のサイズ。値が -1 の場合、OS のデ
フォルトが使用されます。

socket.connection.setup.timeout.max.ms

タイプ: long

デフォルト: 30000 (30 秒)

インポート: 中

クライアントがソケット接続を確立するまで待機する最大時間。接続設定のタイムアウトにより、
連続する接続の失敗ごとに指数関数的に増加します。接続タイムアウトを回避するために、ランダム化係
数 0.2 がタイムアウトに適用されるため、以下の 20% と計算された値の 20% のランダムな範囲が適
用されます。

socket.connection.setup.timeout.ms

タイプ: long

デフォルト: 10000 (10 秒)

重要: medium

クライアントがソケット接続を確立するのを待機する期間。タイムアウトが経過する前に接続がビ
ルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.enabled.protocols

type: list**Default:** TLSv1.2,TLSv1.3**Importance:** medium

SSL 接続に対して有効なプロトコル一覧。Java 11 以降、「TLSv1.2」以降で実行する場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーの両方が TLSv1.2 に対応している場合は、クライアントとサーバーは TLSv1.3 を優先します (TLSv1.2 以上をサポートすることを想定します)。多くのケースでは、このデフォルトは問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type**type:** string**Default:** JKS**Importance:** medium

キーストアファイルのファイル形式。これはクライアントの場合はオプションになります。

ssl.protocol**type:** string**Default:** TLSv1.3**importance:** medium

SSLContext の生成に使用される SSL プロトコル。Java 11 以降「TLSv1.2」を使用して実行すると、デフォルトは「TLSv1.3」です。この値は、ほとんどのユースケースで十分です。最近の JVM で許可される値は 'TLSv1.2' および 'TLSv1.3' です。'TLS'、'TLSv1.1'、'SSL'、'SSLv2'、および 'SSLv3' は古い JVM でサポートされる可能性があります、既知のセキュリティ脆弱性が原因で使用は推奨されません。この設定および 'ssl.enabled.protocols' のデフォルト値を使用すると、サーバーが「TLSv1.3」をサポートしない場合、クライアントは 'TLSv1.2' にダウングレードします。この設定を「TLSv1.2」に設定すると、クライアントは ssl.enabled.protocols の値のいずれかである場合でも、「TLSv1.3」を使用し、サーバーは「TLSv1.3」のみに対応します。

ssl.provider**type:** string**Default:** null**Importance:** medium

SSL 接続に使用されるセキュリティプロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティプロバイダーです。

ssl.truststore.type**type:** string**Default:** JKS**Importance:** medium

トラストストアファイルのファイル形式です。

metadata.max.age.ms**type:** long**デフォルト :** 300000 (5 分)**有効な値 :** [0,...]**Importance:** low

パーティションリーダーが変更されておらず、新しいブローカーやパーティションをプロアクティブに検出するようにしていても、メタデータの更新を強制する期間 (ミリ秒単位)。

metric.reporters

type: list

デフォルト: ""

Importance: low

メトリクスレポーターとして使用するクラスの一

覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。JmxReporter は、JMX 統計を登録するために常に含まれます。

metrics.num.samples

type: int

Default: 2

Valid Values: [1,...]

Importance: low

メトリクスを計算するために保持されるサンプルの数。

metrics.recording.level

type: string

Default: INFO

Valid Values: [INFO, DEBUG, TRACE]

Importance: low

メトリックの最大記録レベル。

metrics.sample.window.ms

type: long

デフォルト: 30000 (30 秒)

有効値: [0,...]

Importance: low

メトリクスサンプルが計算される期間。

reconnect.backoff.max.ms

type: long

デフォルト: 1000 (1秒)

有効値: [0,...]

Importance: low

繰り返し接続に失敗したブローカーに再接続するまで待機する最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは連続した接続障害ごとに指数関数的に増加し、最大接続数までします。バックオフの増加を計算すると、接続のフレッターを回避するために 20% のランダムジッターが追加されます。

reconnect.backoff.ms

type: long

デフォルト: 50

有効な値: [0,...]

インポートランス: low

指定のホストに再接続を試みる前に待機する時間。これにより、密接なループでホストへ繰り返し接続することはできません。このバックオフは、クライアントがブローカーへのすべての接続試行に適用されます。

retries

type: int

Default: 2147483647

Valid Values: [0,...,2147483647]

Importance: low

ゼロよりも大きい値を設定すると、クライアントは一時的なエラーで失敗したリクエストを再送信します。値をゼロまたは **MAX_VALUE** に設定し、対応するタイムアウトパラメーターを使用してクライアントの再試行期間を制御することが推奨されます。

retry.backoff.ms

type: long

デフォルト : 100

有効な値 : [0,...]

インポートランス : low

失敗したリクエストを再試行するまでの待機時間。これにより、障害シナリオによっては、1回目のループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

type: string

デフォルト : /usr/bin/kinit

Importance: low

Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ : long

デフォルト : 60000

インポートランス : low

更新試行の間には、ログインスレッドのスリープ時間。

sasl.kerberos.ticket.renew.jitter

type: double

Default: 0.05

Importance: low

ランダムなジッターの割合が更新時間に追加されます。

sasl.kerberos.ticket.renew.window.factor

type: double

デフォルト : 0.8

クォーラム : low

指定のウィンドウ係数からチケットの有効期限に達するまで、ログインスレッドはスリープ状態になります。

sasl.login.refresh.buffer.seconds

type: short

Default: 300

Valid Values: [0,...,3600]

Importance: low

認証情報の更新時における認証情報が失効するまでの時間（秒単位）。更新が行われない場合、バッファの秒数より期限切れになり始めたら、更新ができるだけ多くのバッファ時間を維持します。有効な値は0から3600（1時間）の間です。値が指定されていない場合は、デフォルト値の300（5分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.min.period.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

type: short

Default: 60

Valid Values: [0,...,900]

Importance: low

ログイン更新スレッドがクレデンシャルを更新する前に待機する最低時間（秒単位）。有効な値は 0 から 900（15 分）までの値になります。値が指定されていない場合は、デフォルト値の 60（1 分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.buffer.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

type: double

Default: 0.8

Valid Values: [0.5,...,1.0]

Importance: low

ログイン更新スレッドは、認証情報の有効期間と相対的な期間係数に達するまでスリープします。この場合、認証情報の更新を試みます。有効な値は 0.5(50%)と 1.0(100%)です。値が指定されていない場合、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

type: double

Default: 0.05

Valid Values: [0.0,...,0.25]

Importance: low

ログイン更新スレッドのスリープ時間に追加されたクレデンシャルの有効期間に対するランダムなジッターの最大数。有効な値は 0 から 0.25(25%)までの値になります。値が指定されていない場合は、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

security.providers

type: string

Default: null

Importance: low

セキュリティーアルゴリズムを実装するプロバイダーを返す設定可能な作成者クラスのリスト。これらのクラスは **`org.apache.kafka.common.security.auth.SecurityProviderCreator`** インターフェースを実装する必要があります。

ssl.cipher.suites

type: list

Default: null

Importance: low

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティー設定をネゴシエートするために使用される認証、暗号化、MAC およびキー交換アルゴリズムの名前付き組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.endpoint.identification.algorithm

タイプ: 文字列

デフォルト: https

インポートランス: low

サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class

Default: null

Importance: low

種別 `org.apache.kafka.common.security.auth.SslEngineFactory` のクラスで `SslEngine` オブジェクトを提供します。Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`.

`ssl.keymanager.algorithm`

type: string

Default: SunX509

Importance: low

SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

`ssl.secure.random.implementation`

type: string

Default: null

Importance: low

SSL 暗号化操作に使用する `SecureRandom` PRNG 実装。

`ssl.trustmanager.algorithm`

type: string

デフォルト : PKIX

重要性 : low

SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

付録F KAFKA CONNECT 設定パラメーター

config.storage.topic

type: string

Importance: high

コネクター設定が保存される Kafka トピックの名前。

group.id

type: string

Importance: high

このワーカーが属する Connect クラスターグループを識別する一意の文字列。

key.converter

type: class

Importance: high

Kafka Connect 形式と Kafka に書き込まれるシリアル化された形式の変換に使用されるコンバータークラス。これにより、Kafka から書き込まれたメッセージのキーの形式を制御します。これはコネクターから独立しているため、コネクターはシリアル化形式と連携できます。一般的なフォーマットの例には、JSON および Avro が含まれます。

offset.storage.topic

type: string

Importance: high

コネクターオフセットが保存される Kafka トピックの名前。

status.storage.topic

type: string

Importance: high

コネクターおよびタスクステータスが保存される Kafka トピックの名前。

value.converter

type: class

Importance: high

Kafka Connect 形式と Kafka に書き込まれるシリアル化された形式の変換に使用されるコンバータークラス。これにより、Kafka によって書き込まれたメッセージや、Kafka から読み取られるメッセージの形式を制御します。これはコネクターから独立しているため、コネクターはシリアル化形式と動作します。一般的なフォーマットの例には、JSON および Avro が含まれます。

bootstrap.servers

type: list

デフォルト: localhost:9092

重要: high

Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ここで指定するすべてのサーバーの使用を行います。このリストは、サーバーのフルセットを検出するために使用される初期ホストにのみ影響します。このリストは

host1:port1,host2:port2,... の形式にする必要があります。これらのサーバーは、クラスターの全メンバーシップを検出するために初期接続（動的に変更されている可能性がある）にだけ使用されているため、この一覧にはサーバーのフルセットを含める必要はありません（サーバーがダウンした場合の場合もあります）。

heartbeat.interval.ms

type: int

デフォルト: 3000 (3 秒)

インポートランス: high

Kafka のグループ管理機能を使用する場合に、グループコーディネーターへのハートビートの時間が予想されます。ハートビートは、ワーカーのセッションがアクティブで維持し、新規メンバーの結合時またはグループから離れる際のリバランスを容易にするために使用されます。値は **session.timeout.ms** 未満に設定する必要がありますが、通常はこの値の 1/3 よりも大きくすることはできません。通常のリバランスの予想される時間を制御するために、低く調整できます。

rebalance.timeout.ms

type: int

デフォルト: 60000 (1 分)

インポート: high

リバランスが開始されると、各ワーカーがグループに参加する最大許容時間。これは基本的に、保留中のデータをフラッシュし、オフセットをコミットするのにすべてのタスクに必要な時間の制限です。タイムアウトを超えると、ワーカーはグループから削除され、オフセットのコミットに失敗します。

session.timeout.ms

type: int

Default: 10000 (10 秒)

重要性: high

ワーカーの失敗の検出に使用されるタイムアウト。ワーカーは定期的なハートビートを送信し、その liveness をブローカーを示します。このセッションタイムアウトの期限が切れる前にブローカーによってハートビートが受信されなかった場合、ブローカーはグループからワーカーを削除し、リバランスを開始します。この値は、**group.min.session.timeout.ms** および **group.max.session.timeout.ms** によってブローカー設定で設定される許容範囲である必要があります。

ssl.key.password

type: password

Default: null

Importance: high

キーストアファイルまたは 'ssl.keystore.key' で指定された PEM キーの秘密鍵のパスワード。これは、双方向認証が設定されている場合のみクライアントに必要です。

ssl.keystore.certificate.chain

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の証明書チェーン。デフォルトの SSL エンジンファクトリーは、X.509 証明書のリストを持つ PEM 形式のみをサポートします。

ssl.keystore.key

type: password

Default: null

Importance: high

'ssl.keystore.type' で指定された形式の秘密鍵デフォルトの SSL エンジンファクトリーは、PKCS#8 キーを持つ PEM 形式のみをサポートします。キーが暗号化されている場合は、'ssl.key.password' を使用してキーパスワードを指定する必要があります。

ssl.keystore.location

type: string

Default: null

Importance: high

キーストアファイルの場所。これはクライアントにはオプションであり、クライアントの双方向認証に使用できます。

ssl.keystore.password

type: password

Default: null

Importance: high

キーストアファイルのストアパスワード。これはクライアントでは任意で、'ssl.keystore.location' が設定されている場合にのみ必要です。キーストアのパスワードは PEM 形式ではサポートされていません。

ssl.truststore.certificates

type: password

Default: null

Importance: high

'ssl.truststore.type' で指定された形式の信頼済み証明書。デフォルトの SSL エンジンファクトリーは、X.509 証明書を使用した PEM 形式のみをサポートします。

ssl.truststore.location

type: string

Default: null

Importance: high

トラストストアファイルの場所。

ssl.truststore.password

type: password

Default: null

Importance: high

トラストストアファイルのパスワード。パスワードが設定されていない場合は、設定されたトラストストアファイルが使用されますが、整合性チェックは無効になります。トラストストアのパスワードは PEM 形式ではサポートされていません。

client.dns.lookup

type: string

デフォルト : use_all_dns_ips

有効な値 : [default, use_all_dns_ips, resolve_canonical_bootstrap_servers_only]

インポートランス : medium

クライアントが DNS ルックアップを使用する方法を制御します。**use_all_dns_ips** に設定すると、正常な接続が確立されるまで、返された各 IP アドレスを順番に接続します。接続が切断されると、次の IP が使用されます。すべての IP が1回使用されると、クライアントはホスト名から IP を再度解決します (JVM と OS キャッシュ DNS 名ルックアップの両方)。

resolve_canonical_bootstrap_servers_only に設定すると、各ブートストラップアドレスを正規名のリストに解決します。ブートストラップフェーズの後に、これは **use_all_dns_ips** と同じように動作します。**default** (非推奨) に設定すると、ルックアップが複数の IP アドレスを返す場合でも、ルックアップによって返される最初の IP アドレスへの接続を試行します。

connections.max.idle.ms

type: long

デフォルト : 540000 (9 分)

Importance: medium

この設定によって指定された期間（ミリ秒単位）の後にアイドル状態の接続を閉じます。

connector.client.config.override.policy

type: string

Default: None

Importance: medium

ConnectorClientConfigOverridePolicy の実装のクラス名またはエイリアス。コネクターによって上書きできるクライアント設定を定義します。デフォルトの実装は **None** です。フレームワークの他の可能なポリシーには、**All** および **Principal** が含まれます。

receive.buffer.bytes

type: int

Default: 32768(32 kibibytes)

Valid values: [0,...]

Importance: medium

データの読み取り時に使用する TCP 受信バッファ(SO_RCVBUF)のサイズ。値が-1 の場合、OS のデフォルトが使用されます。

request.timeout.ms

type: int

デフォルト : 40000 (40 秒)

有効値 : [0,...]

Importance: medium

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。

sasl.client.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL クライアントコールバックハンドラークラスの完全修飾名。

sasl.jaas.config

type: password

Default: null

Importance: medium

JAAS 設定ファイルによって使用される形式で、SASL 接続の JAAS ログインコンテキストパラメーター。JAAS 設定ファイルの形式は、[こちらで説明されています](#)。値のフォーマットは **loginModuleClass controlFlag (optionName=optionValue)*;** です。ブローカーの場合、設定の前にリスナープレフィックスおよびSASL メカニズム名が付けられます。例：

listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;

sasl.kerberos.service.name

type: string

Default: null

Importance: medium

Kafka が実行される Kerberos プリンシパル名。これは、Kafka の JAAS 設定または Kafka の設定のいずれかで定義できます。

sasl.login.callback.handler.class

type: class

Default: null

Importance: medium

AuthenticateCallbackHandler インターフェースを実装する SASL ログインコールバックハンドラークラスの完全修飾名。ブローカーの場合、ログインコールバックハンドラー設定の前にリスナープレフィックスと SASL メカニズム名（小文字の）を付ける必要があります。例：

listener.name.sasl_ssl.scram-sha-

256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler。

sasl.login.class

type: class

Default: null

Importance: medium

Login インターフェースを実装するクラスの完全修飾名。ブローカーの場合、ログイン設定の前にリスナープレフィックスおよび SASL メカニズム名が付けられます。For example,

listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin.

sasl.mechanism

type: string

Default: GSSAPI

Importance: medium

クライアント接続に使用される SASL メカニズム。これは、セキュリティプロバイダーが利用できるメカニズムになることがあります。GSSAPI がデフォルトのメカニズムです。

security.protocol

type: string

Default: PLAINTEXT

Importance: medium

ブローカーとの通信に使用されるプロトコル。有効な値は PLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

send.buffer.bytes

type: int

Default: 131072(128 kibibytes)

Valid Values: [0,...]

Importance: medium

データ送信時に使用する TCP 送信バッファ(SO_SNDBUF)のサイズ。値が-1の場合、OS のデフォルトが使用されます。

ssl.enabled.protocols

type: list

Default: TLSv1.2,TLSv1.3

Importance: medium

SSL 接続に対して有効なプロトコル一覧。Java 11 以降、「TLSv1.2」以降で実行する場合、デフォルトは 'TLSv1.2,TLSv1.3' です。Java 11 のデフォルト値は、クライアントとサーバーの両方が TLSv1.2 に対応している場合は、クライアントとサーバーは TLSv1.3 を優先します (TLSv1.2 以上をサポートすることを想定します)。多くのケースでは、このデフォルトは問題ありません。**ssl.protocol** の設定に関するドキュメントも参照してください。

ssl.keystore.type**type:** string**Default:** JKS**Importance:** medium

キーストアファイルのファイル形式。これはクライアントの場合はオプションになります。

ssl.protocol**type:** string**Default:** TLSv1.3**importance:** medium

SSLContext の生成に使用される SSL プロトコル。Java 11 以降「TLSv1.2」を使用して実行すると、デフォルトは「TLSv1.3」です。この値は、ほとんどのユースケースで十分です。最近の JVM で許可される値は「TLSv1.2」および「TLSv1.3」です。「TLS」、「TLSv1.1」、「SSL」、「SSLv2」、および「SSLv3」は古い JVM でサポートされる可能性があります、既知のセキュリティ脆弱性が原因で使用は推奨されません。この設定および「ssl.enabled.protocols」のデフォルト値を使用すると、サーバーが「TLSv1.3」をサポートしない場合、クライアントは「TLSv1.2」にダウングレードします。この設定を「TLSv1.2」に設定すると、クライアントは ssl.enabled.protocols の値のいずれかである場合でも、「TLSv1.3」を使用し、サーバーは「TLSv1.3」のみに対応します。

ssl.provider**type:** string**Default:** null**Importance:** medium

SSL 接続に使用されるセキュリティプロバイダーの名前。デフォルト値は JVM のデフォルトセキュリティプロバイダーです。

ssl.truststore.type**type:** string**Default:** JKS**Importance:** medium

トラストストアファイルのファイル形式です。

worker.sync.timeout.ms**type:** int**デフォルト :** 3000 (3 秒)**インポート :** 中

ワーカーが他のワーカーと同期しておらず、設定を再同期する必要がある場合は、再参加する前に、中止する前のこの時間を待機してから、この時間を待ちます。

worker.unsync.backoff.ms**type:** int**Default:** 300000 (5 分)**Importance:** medium

ワーカーが他のワーカーと同期し、worker.sync.timeout.ms 内で取得できなかった場合は、再参加する前に Connect クラスターをこの時間のままにします。

access.control.allow.methods**type:** string**デフォルト :** ""**Importance:** low

Access-Control-Allow-Methods ヘッダーを設定して、クロスオリジン要求に対応するメソッドを設定します。Access-Control-Allow-Methods ヘッダーのデフォルト値は、GET、POST、およびHEADのクロスオリジンリクエストを許可します。

access.control.allow.origin

type: string

デフォルト: ""

Importance: low

REST API 要求について Access-Control-Allow-Origin ヘッダーを設定します。クロスオリジンアクセスを有効にするには、これを API へのアクセスが許可されるアプリケーションのドメインに設定します。すべてのドメインからのアクセスを許可するには '*'。デフォルト値は、REST API のドメインからのアクセスのみを許可します。

admin.listeners

type: list

Default: null

Valid Values: org.apache.kafka.connect.runtime.WorkerConfig\$AdminListenersValidator@546a03af

Importance: low

管理 REST API がリッスンするコンマ区切りの URI の一覧。サポートされるプロトコルは HTTP および HTTPS です。空の文字列または空の文字列はこの機能を無効にします。デフォルトの動作では、通常のリスナー（「listeners」プロパティーで指定）を使用します。

client.id

type: string

デフォルト: ""

Importance: low

要求の実行時にサーバーに渡す id 文字列。この目的は、サーバー側の要求ロギングに論理アプリケーション名を含めることで、ip/ポート以外の要求のソースを追跡できるようにすることです。

config.providers

type: list

デフォルト: ""

Importance: low

ConfigProvider クラスの名前のコンマ区切りリスト。指定の順序でロードおよび使用されます。インターフェース **ConfigProvider** を実装すると、外部化されたシークレットなど、コネクタ設定の変数参照を置き換えることができます。

config.storage.replication.factor

type: short

デフォルト: 3

有効値: Kafka クラスター内のブローカーの数よりも大きい番号。-1 はブローカーのデフォルト

規則を使用します: low

設定ストレージトピックの作成時に使用されるレプリケーション係数。

connect.protocol

type: string

デフォルト: sessioned

有効値: [eager, compatible, sessioned]

インポートance: low

Kafka Connect Protocol の互換性モード。

header.converter

type: class**Default:** org.apache.kafka.connect.storage.SimpleHeaderConverter**Importance:** low

Kafka Connect 形式と Kafka に書き込まれるシリアル化された形式の変換に使用される HeaderConverter クラス。これは、Kafka から書き込まれたメッセージのヘッダー値の形式を制御します。これはコネクタとは独立しているため、コネクタはシリアル化形式と動作します。一般的なフォーマットの例には、JSON および Avro が含まれます。デフォルトでは、SimpleHeaderConverter は、ヘッダー値を文字列にシリアル化し、スキーマを推測してデシリアル化するために使用されます。

inter.worker.key.generation.algorithm**タイプ:** 文字列**デフォルト:** HmacSHA256**有効値:** ワーカー JVM でサポートされているキー生成アルゴリズム**インポート性:** 低

内部リクエストキーの生成に使用するアルゴリズム。

inter.worker.key.size**type:** int**Default:** null**Importance:** low

内部のリクエストの署名に使用するキーのサイズ（ビット単位）。null の場合、キー生成アルゴリズムのデフォルトの鍵サイズが使用されます。

inter.worker.key.ttl.ms**type:** int**Default:** 3600000 (1時間)**有効値** [0, ..., 2147483647]**Importance:** low

内部リクエスト検証に使用される生成されたセッションキーの TTL（ミリ秒単位）。

inter.worker.signature.algorithm**type:** string**デフォルト:** HmacSHA256**有効値:** ワーカー JVM**Importance:** low でサポートされている MAC アルゴリズム。

内部要求の署名に使用されるアルゴリズム。

inter.worker.verIFICATION.algorithms**type:** list**デフォルト:** HmacSHA256**有効な値:** 1 つ以上の MAC アルゴリズムのリストで、それぞれがワーカー JVM**インポートをサポートします:** low

内部リクエストの検証に使用可能なアルゴリズムの一覧。

internal.key.converter**type:** class**Default:** org.apache.kafka.connect.json.JsonConverter**Importance:** low

Kafka Connect 形式と Kafka に書き込まれるシリアル化された形式の変換に使用されるコンバータークラス。これにより、Kafka から書き込まれたメッセージのキーの形式を制御します。これはコネクタから独立しているため、コネクタはシリアル化形式と連携できます。一般的なフォー

マットの例には、JSON および Avro が含まれます。この設定は、設定やオフセットなどのフレームワークによって使用される内部ブック管理データに使用される形式を制御するため、ユーザーは通常すべての機能コンバーター実装を使用できます。非推奨。将来のバージョンで削除される予定です。

internal.value.converter

type: class

Default: org.apache.kafka.connect.json.JsonConverter

Importance: low

Kafka Connect 形式と Kafka に書き込まれるシリアル化された形式の変換に使用されるコンバータークラス。これにより、Kafka によって書き込まれたメッセージや、Kafka から読み取られるメッセージの形式を制御します。これはコネクタから独立しているため、コネクタはシリアル化形式と動作します。一般的なフォーマットの例には、JSON および Avro が含まれます。この設定は、設定やオフセットなどのフレームワークによって使用される内部ブック管理データに使用される形式を制御するため、ユーザーは通常すべての機能コンバーター実装を使用できます。非推奨。将来のバージョンで削除される予定です。

listeners

type: list

Default: null

Importance: low

REST API がリッスンするコンマ区切りの URI の一覧。サポートされるプロトコルは HTTP および HTTPS です。すべてのインターフェースにバインドする 0.0.0.0 としてホスト名を指定します。デフォルトインターフェースにバインドする場合は、ホスト名を空のままにします。法人リスナー一覧の例：HTTP://myhost:8083,HTTPS://myhost:8084

metadata.max.age.ms

type: long

デフォルト : 300000 (5 分)

有効な値 : [0,...]

Importance: low

パーティションリーダーが変更されておらず、新しいブローカーやパーティションをプロアクティブに検出するようにしていても、メタデータの更新を強制する期間（ミリ秒単位）。

metric.reporters

type: list

デフォルト : ""

Importance: low

メトリクスレポーターとして使用するクラスの一

覧。**org.apache.kafka.common.metrics.MetricsReporter** インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。JmxReporter は、JMX 統計を登録するために常に含まれます。

metrics.num.samples

type: int

Default: 2

Valid Values: [1,...]

Importance: low

メトリクスを計算するために保持されるサンプルの数。

metrics.recording.level

type: string

Default: INFO
Valid Values: [INFO, DEBUG]
Importance: low
メトリックの最大記録レベル。

metrics.sample.window.ms

type: long
デフォルト : 30000 (30 秒)
有効値 : [0,...]
Importance: low
メトリクスサンプルが計算される期間。

offset.flush.interval.ms

type: long
デフォルト : 60000 (1 分)
インポートランス : low
タスクのオフセットをコミットしようとする間隔。

offset.flush.timeout.ms

type: long
デフォルト : 5000 (5 秒)
インポート性 : low
このプロセスをキャンセルし、将来の試行でコミットするオフセットデータを復元する前に、レコードをオフセットストレージにフラッシュする最大期間（ミリ秒単位）。

offset.storage.partitions

type: int
Default: 25
Valid Values: Positive number, or -1 to use the broker's default
Importance: low
オフセットストレージトピックの作成時に使用されるパーティションの数。

offset.storage.replication.factor

type: short
デフォルト : 3
有効値 : Kafka クラスター内のブローカーの数よりも大きい番号。-1 はブローカーのデフォルト規則を使用します。
Importance: low
オフセットストレージトピックの作成時に使用されるレプリケーション係数。

plugin.path

type: list
Default: null
Importance: low
プラグイン（connectors、コンバーター、変換）が含まれるコンマ(,)で区切ったパスの一覧。このリストは、すべての組み合わせが含まれる最上位のディレクトリーで構成されます。a) ディレクトリーと、プラグインとその依存関係b) は、すぐにプラグインとその依存関係を持つディレクトリーとともに、その依存関係を含むディレクトリーとともに、その依存関係のディレクトリー構造をただちに含む必要があります。ただし、シンボリックリンクは、依存関係またはプラグインを検出します。例： plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors Do not use config provider 変数は、設定プロバイダーを初期化し、変数を置き換えるために使用されます。

reconnect.backoff.max.ms**type:** long**デフォルト :** 1000 (1秒)**有効値 :** [0,...]**Importance:** low

繰り返し接続に失敗したブローカーに再接続するまで待機する最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは連続した接続障害ごとに指数関数的に増加し、最大接続数までします。バックオフの増加を計算すると、接続のフレッターを回避するために20%のランダムジッターが追加されます。

reconnect.backoff.ms**type:** long**デフォルト :** 50**有効な値 :** [0,...]**インポートランス :** low

指定のホストに再接続を試みる前に待機する時間。これにより、密接なループでホストへ繰り返し接続することはできません。このバックオフは、クライアントがブローカーへのすべての接続試行に適用されます。

response.http.headers.config**type:** string**デフォルト :** ""

Valid Values: Comma-separated ヘッダールール。各ヘッダールールは '[action] [header name]: [header value]' の形式であり、ヘッダールールの一部にコンマ

Importance (コンマ) が含まれている場合にオプションで二重引用符で囲まれます。

REST API HTTP 応答ヘッダーのルール。

rest.advertised.host.name**type:** string**Default:** null**Importance:** low

これが設定されている場合、これは他のワーカーに接続するように指定されるホスト名になります。

rest.advertised.listener**type:** string**Default:** null**Importance:** low

他のワーカーに指定されるアドバタイズされたリスナー（HTTP または HTTPS）を設定します。

rest.advertised.port**type:** int**Default:** null**Importance:** low

これが設定されている場合、これは他のワーカーに接続するポートになります。

rest.extension.classes**type:** list**デフォルト :** ""**Importance:** low

ConnectRestExtension クラスのコンマ区切りリストで読み込まれ、指定された順序でロードおよ

び呼び出します。インターフェース **ConnectRestExtension** を実装すると、フィルターなどの Connect の REST API ユーザー定義リソースに挿入することができます。通常、ロギング、セキュリティーなどのカスタム機能の追加に使用されます。

rest.host.name

type: string

Default: null

Importance: low

REST API のホスト名。これが設定されると、このインターフェースにのみバインドされます。

rest.port

type: int

Default: 8083

Importance: low

リッスンする REST API のポート。

retry.backoff.ms

type: long

デフォルト : 100

有効な値 : [0,...]

インポートランス : low

特定のトピックパーティションへの失敗した要求を再試行するまでの待機時間。これにより、障害シナリオによっては、1 回目のループでリクエストを繰り返し送信しないようにします。

sasl.kerberos.kinit.cmd

type: string

デフォルト : /usr/bin/kinit

Importance: low

Kerberos kinit コマンドパス。

sasl.kerberos.min.time.before.relogin

タイプ : long

デフォルト : 60000

インポートランス : low

更新試行の間には、ログインスレッドのスリープ時間。

sasl.kerberos.ticket.renew.jitter

type: double

Default: 0.05

Importance: low

ランダムなジッターの割合が更新時間に追加されます。

sasl.kerberos.ticket.renew.window.factor

type: double

デフォルト : 0.8

クォーラム : low

指定のウィンドウ係数からチケットの有効期限に達するまで、ログインスレッドはスリープ状態になります。

sasl.login.refresh.buffer.seconds

type: short

Default: 300

Valid Values: [0,...,3600]

Importance: low

認証情報の更新時における認証情報が失効するまでの時間（秒単位）。更新が行われない場合、バッファの秒数より期限切れになり始めたら、更新ができるだけ多くのバッファ時間を維持します。有効な値は 0 から 3600（1 時間）の間です。値が指定されていない場合は、デフォルト値の 300（5 分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.min.period.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.min.period.seconds

type: short

Default: 60

Valid Values: [0,...,900]

Importance: low

ログイン更新スレッドがクレデンシャルを更新する前に待機する最低時間（秒単位）。有効な値は 0 から 900（15 分）までの値になります。値が指定されていない場合は、デフォルト値の 60（1 分）が使用されます。認証情報の残存期間が経過すると、この値と `sasl.login.refresh.buffer.seconds` はいずれも無視されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.factor

type: double

Default: 0.8

Valid Values: [0.5,...,1.0]

Importance: low

ログイン更新スレッドは、認証情報の有効期間と相対的な期間係数に達するまでスリープします。この場合、認証情報の更新を試みます。有効な値は 0.5(50%)と 1.0(100%)です。値が指定されていない場合、デフォルト値の 0.8(80%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

sasl.login.refresh.window.jitter

type: double

Default: 0.05

Valid Values: [0.0,...,0.25]

Importance: low

ログイン更新スレッドのスリープ時間に追加されたクレデンシャルの有効期間に対するランダムなジッターの最大数。有効な値は 0 から 0.25(25%)までの値になります。値が指定されていない場合は、デフォルト値の 0.05(5%)が使用されます。現在、OAUTHBEARER にのみ適用されます。

scheduled.rebalance.max.delay.ms

type: int

Default: 300000（5 分）

有効な値： [0,...,2147483647]

Importance: low

再分散してコネクタとタスクをグループに再割り当てする前に、1 つ以上のデパートワーカーを返すまで待機するためにスケジュールされる最大遅延。この期間の間、departed ワーカーのコネクタおよびタスクは未割り当てのままです。

socket.connection.setup.timeout.max.ms

type: long

デフォルト： 30000（30 秒）

有効値： [0,...]

Importance: low

クライアントがソケット接続を確立するまで待機する最大時間。接続設定のタイムアウトにより、連続する接続の失敗ごとに指数関数的に増加します。接続タイムアウトを回避するために、ランダム化係数 0.2 がタイムアウトに適用されるため、以下の 20% と計算された値の 20% のランダムな範囲が適用されます。

socket.connection.setup.timeout.ms

type: long

デフォルト: 10000 (10 秒)

有効な値: [0,...]

Importance: low

クライアントがソケット接続を確立するのを待機する期間。タイムアウトが経過する前に接続がビルドされない場合、クライアントはソケットチャネルを閉じます。

ssl.cipher.suites

type: list

Default: null

Importance: low

暗号化スイートの一覧。これは、TLS または SSL ネットワークプロトコルを使用してネットワーク接続のセキュリティ設定をネゴシエートするために使用される認証、暗号化、MAC およびキー交換アルゴリズムの名前付き組み合わせです。デフォルトでは、利用可能なすべての暗号スイートがサポートされます。

ssl.client.auth

type: string

Default: none

Importance: low

クライアント認証を要求するように kafka ブローカーを設定します。以下は一般的な設定です。

- **ssl.client.auth=required** 必要なクライアント認証に設定されているかどうか。
- **ssl.client.auth=requested** これは、クライアント認証は任意となります。必須とは異なり、このオプションがクライアント自体についての認証情報を指定しない場合は、
- **ssl.client.auth=none** これは、クライアント認証が不要なことを意味します。

ssl.endpoint.identification.algorithm

タイプ: 文字列

デフォルト: https

インポートランス: low

サーバー証明書を使用してサーバーのホスト名を検証するエンドポイント識別アルゴリズム。

ssl.engine.factory.class

type: class

Default: null

Importance: low

種別 `org.apache.kafka.common.security.auth.SslEngineFactory` のクラスで `SslEngine` オブジェクトを提供します。Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`.

ssl.keymanager.algorithm

type: string

Default: SunX509

Importance: low

SSL 接続のキーマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたキーマネージャーファクトリーアルゴリズムです。

ssl.secure.random.implementation

type: string

Default: null

Importance: low

SSL 暗号化操作に使用する SecureRandom PRNG 実装。

ssl.trustmanager.algorithm

type: string

デフォルト : PKIX

重要性 : low

SSL 接続のトラストマネージャーファクトリーによって使用されるアルゴリズム。デフォルト値は、Java 仮想マシンに設定されたトラストマネージャーファクトリーアルゴリズムです。

status.storage.partitions

type: int

Default: 5

Valid Values: Positive number, or -1 to use the broker's default

Importance: low

ステータスストレージトピックの作成時に使用されるパーティションの数。

status.storage.replication.factor

type: short

デフォルト : 3

有効値 : Kafka クラスター内のブローカーの数よりも大きい番号。-1 はブローカーのデフォルト規則を使用します。 **low**

ステータスストレージトピックの作成時に使用されるレプリケーション係数。

task.shutdown.graceful.timeout.ms

type: long

デフォルト : 5000 (5 秒)

インポート性 : low

タスクが正常にシャットダウンされるまで待機する時間。これはタスクごとではなく合計時間です。すべてのタスクがシャットダウンがトリガーされ、順次待機します。

topic.creation.enable

型 : ブール値

デフォルト : true

の修正 : low

ソースコネクターを **topic.creation** プロパティーで設定した場合に、ソースコネクターによって使用されるトピックの自動作成を許可するかどうか。各タスクは admin クライアントを使用してトピックを作成し、Kafka ブローカーに依存してトピックを自動的に作成しません。

topic.tracking.allow.reset

型： ブール値

デフォルト： `true`

の修正： `low`

`true` に設定すると、ユーザー要求はコネクターごとにアクティブなトピックセットをリセットできます。

`topic.tracking.enable`

型： ブール値

デフォルト： `true`

の修正： `low`

ランタイム時のコネクターごとのアクティブなトピックセットの追跡を有効にします。

付録G KAFKA STREAMS 設定パラメーター

application.id

type: string

Importance: high

ストリーム処理アプリケーションのID。Kafka クラスター内で一意である必要があります。これは、デフォルトの client-id プレフィックスである1) として使用されます。2) メンバーシップ管理の group-id、および changelog トピックプレフィックスは3 です。

bootstrap.servers

type: list

Importance: high

Kafka クラスターへの最初の接続を確立するために使用するホストとポートのペアの一覧。クライアントは、ここで指定するすべてのサーバーの使用を行います。このリストは、サーバーのフルセットを検出するために使用される初期ホストにのみ影響します。このリストは **host1:port1,host2:port2,...** の形式にする必要があります。これらのサーバーは、クラスターの全メンバーシップを検出するために初期接続（動的に変更されている可能性がある）にだけ使用されているため、この一覧にはサーバーのフルセットを含める必要はありません（サーバーがダウンした場合の場合もあります）。

replication.factor

type: int

デフォルト : 1

の修正 : high

変更ログトピックおよびストリーム処理アプリケーションによって作成されるトピックの再パーティショントピックのレプリケーション係数。ブローカークラスターがバージョン2.4 以降にある場合は、-1 を設定してブローカーのデフォルトのレプリケーション係数を使用できます。

state.dir

type: string

Default: /tmp/kafka-streams

Importance: high

ステートストアのディレクトリーの場所。このパスは、同じ基礎となるファイルシステムを共有するストリームインスタンスごとに固有でなければなりません。

acceptable.recovery.lag

type: long

Default: 10000

Valid Values: [0,...]

Importance: medium

アクティブなタスクでクライアントがキャッチされる最大ラグ（オフセットの数）。指定のワークロードの1分未満のリカバリー時間に対応します。0 以上でなければなりません。

cache.max.bytes.buffering

type: long

デフォルト : 10485760

有効な値 : [0,...]

Importance: medium

すべてのスレッドでバッファーに使用されるメモリーバイトの最大数。

client.id

type: string

デフォルト: ""

Importance: medium

内部コンシューマー、プロデューサー、および `restore-consumer` のクライアント ID に使用される ID プレフィックス文字列。パターン '`<client.id>-StreamThread-<threadSequenceNumber>-<consumer|producer|restore-consumer>`'。

default.deserialization.exception.handler

type: class

Default: `org.apache.kafka.streams.errors.LogAndFailExceptionHandler`

Importance: medium

org.apache.kafka.streams.errors.DeserializationExceptionHandler インターフェースを実装する例外処理クラス。

default.key.serde

type: class

Default: `org.apache.kafka.common.serialization.Serdes$ByteArraySerde`

Importance: medium

org.apache.kafka.common.serialization.Serde インターフェースを実装するキーのデフォルトのシリアライザー/デシリアライザークラス。windowed serde クラスが使用されている場合は、'`default.windowed.key.serde.inner`' または '`default.windowed.value.serde.inner`' でも

org.apache.kafka.common.serialization.Serde インターフェースを実装する内部クラスを設定する必要があります。

default.production.exception.handler

type: class

Default: `org.apache.kafka.streams.errors.DefaultProductionExceptionHandler`

Importance: medium

org.apache.kafka.streams.errors.ProductionExceptionHandler インターフェースを実装する例外処理クラス。

default.timestamp.extractor

Type: class

デフォルト: `org.apache.kafka.streams.processor.FailOnInvalidTimestamp`

Importance: medium

org.apache.kafka.streams.processor.TimestampExtractor インターフェースを実装するデフォルトのタイムスタンプ抽出。

default.value.serde

type: class

Default: `org.apache.kafka.common.serialization.Serdes$ByteArraySerde`

Importance: medium

org.apache.kafka.common.serialization.Serde インターフェースを実装する値のデフォルトシリアライザー/デシリアライザークラス。windowed serde クラスが使用されている場合は、'`default.windowed.key.serde.inner`' または '`default.windowed.value.serde.inner`' でも

org.apache.kafka.common.serialization.Serde インターフェースを実装する内部クラスを設定する必要があります。

default.windowed.key.serde.inner

type: class

Default: null

Importance: medium

ウィンドウ化されたキーの内部クラスのデフォルトのシリアライザー/デシリアライザー。 **org.apache.kafka.common.serialization.Serde** インターフェースを実装する必要があります。

default.windowed.value.serde.inner

type: class

Default: null

Importance: medium

ウィンドウ化された値の内部クラスのデフォルトのシリアライザー/デシリアライザー。 **org.apache.kafka.common.serialization.Serde** インターフェースを実装する必要があります。

max.task.idle.ms

タイプ: long

デフォルト: 0

のインポート: 中

パーティションバッファーにすべてのパーティションバッファーにレコードが含まれていなければ、ストリームタスクがアイドル状態になる最大時間（ミリ秒単位）。これにより、複数の入力ストリーム間で順序不足のレコード処理の可能性を回避します。

max.warmup.replicas

type: int

Default: 2

Valid Values: [1,...]

Importance: medium

ウォームアップレプリカの最大数（設定された num.standbys 以上の期間で1つのインスタンスがタスクを利用可能にする目的で1度に割り当てることができるウォームスタンバイ）の最大数。別のインスタンスでこのタスクを利用できる状態にしておく一方で、再割り当てされたことになります。高可用性に使用できる追加のブローカートラフィックとクラスター状態のサイズを調整するために使用されます。1以上でなければなりません。

num.standby.replicas

type: int

Default: 0

Importance: medium

各タスクのスタンバイレプリカ数。

num.stream.threads

type: int

Default: 1

Importance: medium

ストリーム処理を実行するスレッドの数。

processing.guarantee

type: string

デフォルト: at_least_once

有効な値: [at_least_once, exactly_once_beta]

Importance: medium

処理により、使用するべき確実性があります。使用できる値は、**at_least_once**（デフォルト）、**exactly_once**（ブローカーバージョン 0.11.0 以降が必要）、および **exactly_once_beta**（ブローカーバージョン 2.5 以降が必要）です。完全に1回実行処理には、デフォルトで、実稼働に推奨

される設定である3つ以上のブローカークラスターが必要です。開発の場合は、ブローカーの設定 **transaction.state.log.replication.factor** および **transaction.state.log.min.isr** を調整することで変更できます。

security.protocol

type: string

Default: PLAINTEXT

Importance: medium

ブローカーとの通信に使用されるプロトコル。有効な値はPLAINTEXT、SSL、SASL_PLAINTEXT、SASL_SSL です。

task.timeout.ms

type: long

デフォルト : 300000 (5 分)

有効な値 : [0,...]

Importance: medium

内部エラーが原因でタスクが停止している可能性があり、エラーが発生するまで再試行する可能性がある最大時間（ミリ秒単位）。0ms のタイムアウトでは、タスクは最初の内部エラーに対してエラーを出しました。0ms を超えるタイムアウトの場合には、エラーが発生する前に、タスクは少なくとも1回再試行します。

topology.optimization

type: string

Default: none

Valid Values: [none, all]

Importance: medium

Kafka Streams に指示する設定は、デフォルトで無効化され、トポロジを最適化する必要があります。

application.server

type: string

デフォルト : ""

Importance: low

このKafkaStreams インスタンスのステートストア検出と対話型クエリーに使用できる、ユーザー定義のエンドポイントを参照するhost:port ペア。

buffered.records.per.partition

type: int

Default: 1000

Importance: low

パーティションごとにバッファーする最大レコード数。

built.in.metrics.version

type: string

Default: latest

Valid Values: [0.10.0-2.4, latest]

Importance: low

使用する組み込みメトリクスのバージョン。

commit.interval.ms

type: long

デフォルト : 30000 (30 秒)

有効値 : [0,...]

Importance: low

プロセッサの場所を保存する頻度 (ミリ秒単位)。(`processing.guarantee` が `exactly_once` に設定された場合、デフォルト値は **100** です。それ以外の場合は、デフォルト値は **30000** です。

connections.max.idle.ms

type: long

デフォルト : 540000 (9 分)

Importance: low

この設定によって指定された期間 (ミリ秒単位) の後にアイドル状態の接続を閉じます。

metadata.max.age.ms

type: long

デフォルト : 300000 (5 分)

有効な値 : [0,...]

Importance: low

パーティションリーダーが変更されておらず、新しいブローカーやパーティションをプロアクティブに検出するようにしていても、メタデータの更新を強制する期間 (ミリ秒単位)。

metric.reporters

type: list

デフォルト : ""

Importance: low

メトリクスレポーターとして使用するクラスの一

覧。 `org.apache.kafka.common.metrics.MetricsReporter` インターフェースを実装すると、新しいメトリクス作成の通知となるクラスにプラグインすることができます。 `JmxReporter` は、JMX 統計を登録するために常に含まれます。

metrics.num.samples

type: int

Default: 2

Valid Values:[1,...]

Importance: low

メトリクスを計算するために保持されるサンプルの数。

metrics.recording.level

type: string

Default: INFO

Valid Values:[INFO, DEBUG, TRACE]

Importance: low

メトリックの最大記録レベル。

metrics.sample.window.ms

type: long

デフォルト : 30000 (30 秒)

有効値 : [0,...]

Importance: low

メトリクスサンプルが計算される期間。

partition.grouper

type: class

Default: org.apache.kafka.streams.processor.DefaultPartitionGrouper

Importance: low

org.apache.kafka.streams.processor.PartitionGrouper インターフェースを実装するパーティション分類クラス。警告：この設定は非推奨となり、3.0.0 リリースで削除される予定です。

poll.ms

タイプ: long

デフォルト: 100

修正: low

入力を待機する期間（ミリ秒単位）。

probing.rebalance.interval.ms

type: long

デフォルト: 600000 (10 分)

有効な値: [60000,...]

Importance: low

ウォームアップを終了し、アクティブになるレプリカに対するリバランスをトリガーするまで待機する最大時間（ミリ秒単位）。割り当てがバランスになるまで、リバランスはトリガーを継続します。1 分以上でなければなりません。

receive.buffer.bytes

type: int

Default: 32768(32 kibibytes)

Valid Values: [-1,...]

Importance: low

データの読み取り時に使用する TCP 受信バッファ(SO_RCVBUF)のサイズ。値が-1 の場合、OS のデフォルトが使用されます。

reconnect.backoff.max.ms

type: long

デフォルト: 1000 (1秒)

有効値: [0,...]

Importance: low

繰り返し接続に失敗したブローカーに再接続するまで待機する最大時間（ミリ秒単位）。指定された場合、ホストごとのバックオフは連続した接続障害ごとに指数関数的に増加し、最大接続数までします。バックオフの増加を計算すると、接続のフレッタを回避するために20%のランダムジッターが追加されます。

reconnect.backoff.ms

type: long

デフォルト: 50

有効な値: [0,...]

インポートランス: low

指定のホストに再接続を試みる前に待機する時間。これにより、密接なループでホストへ繰り返し接続することはできません。このバックオフは、クライアントがブローカーへのすべての接続試行に適用されます。

request.timeout.ms

type: int

デフォルト: 40000 (40秒)

有効値: [0,...]

Importance: low

この設定では、クライアントがリクエストの応答を待つ最大時間を制御します。タイムアウトが経過する前に応答が受信されない場合、クライアントがリクエストを再送するか、再試行した場合はリクエストが失敗します。

retries

type: int

Default: 0

Valid Values: [0,...,2147483647]

Importance: low

ゼロよりも大きい値を設定すると、クライアントは一時的なエラーで失敗したリクエストを再送信します。値をゼロまたは **MAX_VALUE** に設定し、対応するタイムアウトパラメーターを使用してクライアントの再試行期間を制御することが推奨されます。

retry.backoff.ms

type: long

デフォルト : 100

有効な値 : [0,...]

インポートランス : low

特定のトピックパーティションへの失敗した要求を再試行するまでの待機時間。これにより、障害シナリオによっては、1回目のループでリクエストを繰り返し送信しないようにします。

rocksdb.config.setter

type: class

Default: null

Importance: low

org.apache.kafka.streams.state.RocksDBConfigSetter インターフェースを実装する Rocks DB 設定セッタークラスまたはクラス名。

send.buffer.bytes

type: int

Default: 131072(128 kibibytes)

Valid Values: [-1,...]

Importance: low

データ送信時に使用する TCP 送信バッファ(SO_SNDBUF)のサイズ。値が-1の場合、OS のデフォルトが使用されます。

state.cleanup.delay.ms

type: long

デフォルト : 600000 (10 分)

Importance: low

パーティションが移行されたときに状態を削除するまでの待機時間（ミリ秒単位）。少なくとも **state.cleanup.delay.ms** で編集されていない状態ディレクトリーのみが削除されます。

upgrade.from

type: string

Default: null

Valid Values: [null, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1, 2.0, 2.1, 2.2, 2.3]

Importance: low

後方互換性がある方法でのアップグレードを可能にします。これは、[0.10.0, 1.1] から 2.0 以降にアップグレードする場合や、[2.0, 2.3] から 2.4+ にアップグレードする際に必要になります。2.4 から新しいバージョンにアップグレードする場合、この設定を指定する必要はありません。デフォルトは

null です。許可される値は "0.10.0"、"0.10.1"、"0.10.2"、"0.11.0"、""1.0"、"1.1"、"2.0"、"2.1"、"2.2"、"2.3" です（対応する古いバージョンからアップグレードする場合）。

window.size.ms

タイプ : long

デフォルト : null

インポートランス : low

ウィンドウの終了時間を計算するために、デシリアライザーのウィンドウサイズを設定します。

windowstore.changelog.additional.retention.ms

type: long

デフォルト : 86400000 (1 日)

Importance: low

データがログ早期から削除されないように、ウィンドウメンテナンス Ms に追加されました。クロックドリフトを許可します。デフォルトは1 日です。

付録H サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **JBOSS INTEGRATION AND AUTOMATION** カテゴリの **Red Hat AMQ Streams** エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。 **Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

パッケージ用のシステムの登録

Red Hat Enterprise Linux に RPM パッケージをインストールするには、システムが登録されている必要があります。zip ファイルまたは tar ファイルを使用している場合は、この手順は必要ありません。

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。
3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムの端末に一覧表示されたコマンドを使用して、登録を完了します。

詳細は、[「How to Register and Subscribe a System to the Red Hat Customer Portal」](#) を参照してください。

改訂日時： 2021-08-27 00:36:38 +1000