Red Hat AMQ 7.6

AMQ Streams 1.4 on OpenShift リリースノート

AMQ Streams on OpenShift Container Platform の使用

Last Updated: 2021-02-15

Red Hat AMQ 7.6 AMQ Streams 1.4 on OpenShift リリースノート

AMQ Streams on OpenShift Container Platform の使用

法律上の通知

概要

本リリースノートには、AMQ Streams 1.4 リリースに含まれる新機能、改良された機能、修正、および問題に関する最新情報が含まれています。

目次

第1章 機能

AMQ Streams バージョン 1.4 は Strimzi 0.17.x をベースにしています。

このリリースで追加され、これまでの AMQ Streams リリースにはなかった機能は次のとおりです。

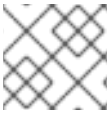
1.1. KAFKA 2.4.0 のサポート

AMQ Streams は Apache Kafka バージョン 2.4.0 に対応するようになりました。

AMQ Streams は Kafka 2.4.0 を使用します。Red Hat によってビルドされた Kafka ディストリビューションのみがサポートされます。

ブローカーおよびクライアントアプリケーションを Kafka 2.4.0 にアップグレードする前に、Cluster Operator を AMQ Streams バージョン 1.4 にアップグレードする必要があります。アップグレードの手順は、「AMQ Streams および Kafka のアップグレード」を参照してください。

詳細は、Kafka 2.3.0 および Kafka 2.4.0 のリリースノートを参照してください。



注記

Kafka 2.3.x は、アップグレードの目的のみで AMQ Streams 1.4 でサポートされます。

サポートされるバージョンの詳細は、カスタマーポータルの「Red Hat AMQ 7 Component Details Page」を参照してください。

Kafka 2.4.0 でのパーティションリバランスプロトコルの変更

Kafka 2.4.0 には、コンシューマーおよび Kafka Streams アプリケーションの **Incremental Cooperative Rebalancing** が追加されました。これは、定義された **リバランスストラテジー** にしたがって **パーティションリバランス** を実装するための、改良されたリバランスプロトコルです。コンシューマーは新しいプロトコルを使用して、リバランス中に割り当てられたパーティションを保持し、クラスターのバランスを保つ必要がある場合に、プロセスの最後でのみパーティションを取り消します。これにより、リバランス中に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を削減します。

Incremental Cooperative Rebalancing を利用するには、コンシューマーおよび Kafka Streams アプリケーションをアップグレードして、**Eager Rebalance Protocol** の代わりに新しいプロトコルを使用する必要があります。

「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」および Apache Kafka ドキュメントの「Notable changes in 2.4.0」を参照してください。

1.1.1. ZooKeeper 3.5.7

Kafka バージョン 2.4.0 では、ZooKeeper の新しいバージョン 3.5.7 が必要です。

手作業で ZooKeeper 3.5.7 にアップグレードする必要は**ありません**。Kafka ブローカーがアップグレードされる ときに、Cluster Operator によって ZooKeeper のアップグレードが実行されます。ただし、この手順の実行中に追加のローリングアップデートが実行されることがあります。

AMQ Streams 1.4 には、ZooKeeper のスケーリングに関する既知の問題が存在します。詳細は、*6章 既知の問題*。

1.2. KAFKACONNECTOR リソース

AMQ Streams は、**KafkaConnector** という新しいカスタムリソースと内部 Operator を使用して、Kafka Connect クラスターでコネクターの Kubernetes ネイティブな管理を提供するようになりました。

KafkaConnector YAML ファイルには、コネクターインスタンスの新規作成または稼働中のコネクターインスタンスの管理を行うために Kubernetes クラスターにデプロイするソースまたはシンクコネクターの設定が記述されます。他の Kafka リソースと同様に、稼働中のコネクターインスタンスは **KafkaConnectors** に定義された設定と一致するように Cluster Operator によって更新されます。

Installation and Example Files の **examples/connector/source-connector.yaml** に **KafkaConnector** リソースの例が含まれるようになりました。YAML ファイルの例をデプロイし、**my-topic** というトピックでメッセージとしてライセンスファイルの各行を Kafka に送信する **FileStreamSourceConnector** を作成します。

KafkaConnector の例

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  Config:
    file: "/opt/kafka/LICENSE"
    topic: my-topic
    # ...
```

1.2.1. KafkaConnectors の有効化

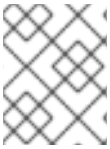
これまでのバージョンの AMQ Streams との互換性を維持するため、**KafkaConnectors** はデフォルトで無効になっています。今後の AMQ Streams リリースでは、これがコネクターを作成および管理するデフォルトの方法になる可能性があります。

AMQ Streams 1.4 Kafka Connect クラスターの **KafkaConnectors** を有効にするには、**strimzi.io/use-connector-resources** アノテーションを **KafkaConnect** リソースに追加します。以下に例を示します。

KafkaConnectors が有効になっている Kafka Connect クラスターの例

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
```

KafkaConnectors が有効になっている場合、Kafka Connect REST API に直接手作業で追加された変更は Cluster Operator によって元に戻されます。



注記

Kafka Connect REST API (8083 番ポート上) は、失敗したタスクを再起動するために必要です。

「コネクターの作成および管理」、「**KafkaConnector** リソースを Kafka Connect にデプロイ」、および「**KafkaConnector** リソースの有効化」を参照してください。

1.3. KAFKA リスナー証明書

以下のタイプのリスナーに、独自のサーバー証明書と秘密鍵を提供できるようになりました。

- TLS リスナー
- TLS 暗号化が有効になっている外部リスナー

これらユーザー提供の証明書は **Kafka リスナー証明書** と呼ばれます。

組織のプライベート認証局 (CA) またはパブリック CA を使用して、独自の Kafka リスナー証明書を生成および署名できます。

リスナーの設定

リスナーの **configuration.brokerCertChainAndKey** プロパティーで Kafka リスナー証明書を設定します。以下に例を示します。

```
# ...
listeners:
  plain: {}
  external:
    type: loadbalancer
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
    tls: true
    authentication:
      type: tls
# ...
```

「Kafka リスナー証明書」および「独自のKafka リスナー証明書の指定」を参照してください。

1.4. OAUTH 2.0 認証

OAuth 2.0 認証のサポートは、テクノロジープレビューから AMQ Streams の一般利用可能なコンポーネントに移行されます。

AMQ Streams は、**SASL OAUTHBEARER** 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 のトークンベースの認証を使用すると、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。クライアントは、認証手段としてアクセストークンを提示します。アプリケーションサーバーはこのアクセストークンを使用して、付与されたアクセス権限のレベルに関する情報も検索できます。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 ブローカーはリソースサーバーとして動作する。
- Kafka クライアントはリソースクライアントとして動作する。

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。

Red Hat Single Sign-On の統合

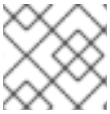
Red Hat Single Sign-On を承認サーバーとしてデプロイし、AMQ Streams と統合するために設定することができます。

Red Hat Red Hat Single Sign-On を使用して、以下を実行できます。

- Kafka ブローカーの認証の設定
- クライアントの設定および承認
- ユーザーおよびロールの設定
- アクセスの取得およびトークンの更新

「OAuth 2.0 トークンベース認証の使用」を参照してください。

1.5. DEBEZIUM FOR CHANGE DATA CAPTURE の統合



注記

Debezium for Change Data Capture は OpenShift 4.x でのみサポートされます。

Dabezium for Change Data Capture は、データベースを監視し、変更イベントストリームを作成する分散プラットフォームです。Debezium は Apache Karaf に構築され、AMQ Streams とデプロイおよび統合できます。AMQ Streams のデプロイ後に、Kafka Connect で Debezium をコネクター設定としてデプロイします。Debezium によって、データベーステーブルの行レベルの変更がキャプチャーされ、対応する変更イベントが AMQ Streams on OpenShift に渡されます。アプリケーションは **変更イベントストリーム** を読み取りでき、変更イベントが発生した順にアクセスできます。

Debezium には、以下を含む複数の用途があります。

- データレプリケーション。
- キャッシュの更新およびインデックスの検索。

- モノリシックアプリケーションの簡素化。
- データ統合。
- ストリーミングクエリーの有効化。

Debezium は、以下の共通データベースのコネクター (Kafka Connect をベースとする) を提供します。

- MySQL
- PostgreSQL
- SQL Server
- MongoDB

AMQ Streams で Debezium をデプロイするための詳細は、「製品ドキュメント」を参照してください。

第2章 改良された機能

このリリースで改良された機能は次のとおりです。

2.1. KAFKA 2.4.0 で改良された機能

Kafka 2.4.0 に導入された改良機能の概要は『Kafka 2.4.0 Release Notes』を参照してください。

2.2. KAFKA BRIDGE による分散トレースのサポート

AMQ Streams on OpenShift の Kafka Bridge コンポーネントで、Jaeger を使用した分散トレースがサポートされるようになりました。

Kafka Bridge は、Kafka Bridge と HTTP クライアントとの間でメッセージを送受信する場合や、HTTP クライアントがリクエストを Kafka Bridge REST API に送信しコンシューマーを作成してメッセージを取得する場合などに、トレースを生成します。これらのトレースは Jaeger ユーザーインターフェースで表示できます。

Kafka Bridge のトレースを有効にするには、Jaeger トレースの **KafkaBridge** カスタムリソースを設定します。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

kubectl apply を使用して、Kafka クラスターのリソースを更新します。リソースが更新されると、設定に基づいた Jaeger トレーサーが Kafka Bridge によって初期化されます。

「分散トレーシング」および「MirrorMaker、Kafka Connect、および Kafka Bridge リソースでのトレースの有効化」を参照してください。

2.3. ユーザークォータ

ユーザークォータは、Kafka ブローカーへのアクセスに関し、ユーザーが定義されたアクセスレベルを超えないようにします。**KafkaUser** リソースで 2 種類のユーザークォータを設定できるようになりました。

- バイトしきい値を元にしたネットワーク使用率クオータ。

- CPU 使用率の時間制限を元にした CPU 使用率クォータ。

ユーザークォータを設定するには、**KafkaUser.spec.quotas** リソースの **KafkaUser** プロパティーを編集します。

「Kafka User リソース」、「**KafkaUser** スキーマ参照」、および Apache Kafka ドキュメントの「Quotas」を参照してください。

2.4. PKCS #12 ストレージ

AMQ Streams は **Secrets** を使用して、Kafka クラスターコンポーネントおよびクライアントの秘密鍵および証明書を格納します。Secrets は、Kafka ブローカー間およびブローカーとクライアント間で TLS で暗号化された接続を確立するために使用されます。Secret は相互 TLS 認証にも使用されます。

PKCS #12 は、暗号化オブジェクトをパスワードで保護された単一のファイルに格納するためのアーカイブファイル形式 (**.p12**) を定義します。PKCS #12 を使用して、証明書および鍵を 1 カ所で管理できるようになりました

「PKCS #12 ストレージ」を参照してください。

2.5. KAFKA CONNECT ベースイメージの DOCKERFILE ユーザー

Kafka Connect ベースイメージから Docker イメージを作成するときに Dockerfile に指定された **USER** が変更になりました。

AMQ Streams バージョン	Dockerfile の **USER** 命令の値
1.3	**USER jboss:jboss**
1.4	**USER 1001**

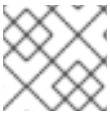
「Kafka Connect ベースイメージからの Docker イメージの作成」を参照してください。

第3章 テクノロジープレビュー

重要

テクノロジープレビューの機能は、Red Hat の実稼働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るため、Red Hat はテクノロジープレビュー機能を実稼働環境に実装することは推奨しません。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。サポート範囲の詳細は、「テクノロジプレビュー機能のサポート範囲」を参照してください。

3.1. OAUTH 2.0 承認



注記

これはテクノロジープレビューの機能です。

トークンベースの認証に OAuth 2.0 を使用している場合、Keycloak を使用して、クライアントの Kafka ブローカーへのアクセスを制限する承認ルールを設定できるようになりました。

OAuth 2.0 トークンベースの承認であるこのテクノロジープレビューは、Red Hat Single Sign-On 7.3 ではサポートされません。この機能を試す場合は、Keycloak 8.0.2 を承認サーバーとする開発環境での使用はテストされています。

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

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

「OAuth 2.0 トークンベース承認の使用」を参照してください。

3.2. SERVICE REGISTRY

注記

これはテクノロジープレビューの機能です。

Service Registry は、データストリーミングのサービススキーマの集中型ストアとして使用できます。Kafka では、Service Registry を使用して **Apache Avro** または JSON スキーマを格納できます。

Service Registry は、REST API および Java REST クライアントを提供し、サーバー側のエンドポイントを介してクライアントアプリケーションからスキーマを登録およびクエリーします。

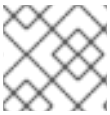
Service Registry を使用すると、クライアントアプリケーションの設定からスキーマ管理のプロセスが分離されます。クライアントコードに URL を指定して、アプリケーションがレジストリーからスキーマを使用できるようにします。

たとえば、メッセージをシリアライズおよびデシリアライズするスキーマをレジストリーに保存できます。保存後、スキーマを使用するアプリケーションから参照され、アプリケーションが送受信するメッセージがこれらのスキーマと互換性を維持するようにします。

Kafka クライアントアプリケーションは実行時にスキーマを Service Registry からプッシュまたはプルできます。

「Service Registry を使用したスキーマの管理」を参照してください。

3.3. MIRRORMAKER 2.0



注記

これはテクノロジープレビューの機能です。

AMQ Streams で MirrorMaker 2.0 を使用できるようになりました。

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

MirrorMaker 2.0 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 では、クラスターでデータをレプリケートする全く新しい方法が導入されました。MirrorMaker 2.0 の使用を選択した場合、現在、レガシーサポートがないため、リソースを手作業で新しい形式に変換する必要があります。

注記

このテクノロジープレビュー機能では、現在、MirrorMaker 2.0 Operator が照合されるたびにすべてのコネクターが再起動されます。これは機能には影響しませんが、パフォーマンスに影響します。

「AMQ Streams の MirrorMaker 2.0 との使用 」を参照してください。

3.4. OPENSHIFT 4.X の非接続インストール

注記

これはテクノロジープレビューの機能です。

OpenShift クラスターが制限されたネットワークで**非接続クラスター** として使用されている場合、AMQ Streams の **非接続インストール** を実行できます。

非接続インストールでは、必要なイメージを取得し、それらをローカルでコンテナーレジストリーにプッシュします。Operator Lifecycle Manager (OLM) を使用している場合、OperatorHub によって使用されるデフォルトのソースを無効にし、ローカルミラーを作成してローカルソースから AMQ Streams をインストールすることになります。

「ネットワークが制限された環境での Operator Lifecycle Manager の使用」を参照してください。

第4章 非推奨の機能

AMQ Streams 1.4 で非推奨になった機能はありません。

第5章 修正された問題

AMQ Streams 1.4 で修正された問題は、以下の表のとおりです。

課題番号	説明
ENTMQST-1106	**overrides.bootstrap.address** の追加や変更によってローリングアップデートがトリガーされる必要がある。
ENTMQST-1153	Kafka ログおよび例外のスケーリングが適切に動作しない。
ENTMQST-1402	KafkaBridge ステータスの誤った URL
ENTMQST-1411	TLS-sidecar が Kafka コンテナー自体より先に終了できる。
ENTMQST-1481	Kafka Connect 設定で **admin.** 設定接頭辞を使用。
ENTMQST-1530	**[RFE]** 単位を変換する機能の追加。
ENTMQST-1531	Pod の **-Xmx** はメモリー制限ではなくメモリーリクエストから判断される。
ENTMQST-1536	Zookeeper および Kafka のダッシュボードは、メモリーおよび JVM メモリーに誤った式を使用。
ENTMQST-1551	CA の更新が中断されるとクラスターが破損する。
ENTMQST-1563	アラートマネージャー設定ファイルのファイル名が間違っている。
ENTMQST-1652	単一ノードクラスターの **COORDINATOR_NOT_AVAILABLE**
ENTMQST-1717	ZSTD ライブラリーの修正

第6章 既知の問題

このセクションでは、AMQ Streams 1.4 on OpenShift の既知の問題について説明します。

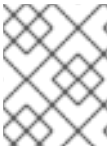
6.1. ZOOKEEPER 3.5.7 のスケールアップまたはダウン

ZooKeeper のスケールアップまたはダウンに関連する既知の問題があります。**ZooKeeper のスケールアップ**とは、サーバーを ZooKeeper クラスターに追加することを言います。**ZooKeeper のスケールダウン**とは、ZooKeeper クラスターからサーバーを削除することを言います。

Kafka 2.4.0 には ZooKeeper 3.5.7 が必要です。

ZooKeeper 3.5.7 サーバーの設定手順は、ZooKeeper 3.4.x サーバーとは大きく異なります。新しい設定手順は 動的再設定 と呼ばれ、ZooKeeper CLI または Admin API を使用してサーバーを追加または削除する必要があります。これにより、スケールアップまたはダウンの処理中に ZooKeeper クラスターの安定性が維持されます。

ZooKeeper 3.5.7 クラスターをスケールアップまたはダウンするには、この既知の問題に記載されている手順を実行する必要があります。



注記

今後の AMQ Streams のリリースでは、ZooKeeper のスケールアップおよびスケールダウンは Cluster Operator によって処理されます。

AMQ Streams 1.4 クラスターでの ZooKeeper 3.5.7 サーバーのスケールアップ

この手順では、以下を前提とします。

- AMQ Streams は **namespace** namespace で実行され、Kafka クラスターの名前は **my-cluster** です。
- 3 つのノードで構成される ZooKeeper クラスターが稼働しています。

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. **Kafka** カスタムリソースの **spec.zookeeper.replicas** プロパティーを編集します。レプリカ数を **4** (**n=4**) に設定します。

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
  # ...
  zookeeper:
    replicas: 4
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
  # ...
```

2. ZooKeeper サーバー (**zookeeper-3**) を通常通り起動し、既存のクォーラムへのリンクを確立します。
これを確認するには、以下を実行します。

```
kubectl exec -n <namespace> -it <my-cluster>-zookeeper-3 -c zookeeper -- bash -c "echo 'srvr' | nc 127.0.0.1 21813 | grep 'Mode:'"
```

コマンドの出力は **Mode: follower** と似たものになるはずです。

注記

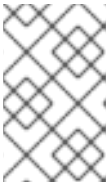
新しい ZooKeeper ノード **zookeeper-x** の名前にあるインデックス番号は、**nc 127.0.0.1 2181x** コマンドにあるクライアントポートの最後の番号と一致します。

3. 元のクラスターを構成するノードの 1 つ (ノード 0、1、または 2) で **zookeeper-shell** セッションを開きます。

```
kubectl exec -n <namespace> -it <my-cluster>-zookeeper-0 -c zookeeper -- ./bin/zookeeper-shell.sh localhost:21810
```

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.4=127.0.0.1:28883:38883:participant;127.0.0.1:21813
```



注記

ZooKeeper クラスター内では、ノードはノード名のようにゼロからではなく、1 からインデックス化されます。そのため、新しい **zookeeper-3** ノードは ZooKeeper 設定内で **server.4** と呼ばれます。

これにより、新規クラスター設定が出力されます。

```
server.1=127.0.0.1:28880:38880:participant;127.0.0.1:21810
server.2=127.0.0.1:28881:38881:participant;127.0.0.1:21811
server.3=127.0.0.1:28882:38882:participant;127.0.0.1:21812
server.4=127.0.0.1:28883:38883:participant;127.0.0.1:21813
version=100000054
```

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. **Kafka** カスタムリソースの **spec.zookeeper.replicas** で、レプリカ数を 1 つ増やします (**n=5**)。

6. ZooKeeper サーバー (**zookeeper-<n-1>**) を通常通り起動し、既存のクォーラムへのリンクを確立します。これを確認するには、以下を実行します。

```
kubectl exec -n <namespace> -it <my-cluster>-zookeeper-<n-1> -c zookeeper -- bash -c "echo 'srvr' | nc 127.0.0.1 2181<n-1> | grep 'Mode:'"
```

コマンドの出力は **Mode: follower** と似たものになるはずです。

7. 元のクラスターを構成するノードの 1 つ (この例ではノード 0 >= i <= n-2) で **zookeeper-shell** セッションを開きます。

```
kubectl exec -n <namespace> -it <my-cluster>-zookeeper-<i> -c zookeeper --
./bin/zookeeper-shell.sh localhost:2181<i>
```

8. シェルセッションで以下の行を入力し、新しい ZooKeeper 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.<n>=127.0.0.1:2888<n-1>:3888<n-1>:participant;127.0.0.1:2181<n-1>
```

9. 追加するサーバーごとに、ステップ 5 から 8 を繰り返します。

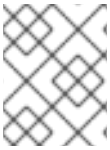
10. 希望するサイズのクラスターがある場合、ZooKeeper クラスターを再度ロールできることを Cluster Operator に通信する必要があります。これには、**Kafka** カスタムリソースで **manual-zk-scaling** アノテーションを **false** に設定します。ZooKeeper レプリカの数を変更すると、Cluster Operator によって自動的に **true** に設定されます。

```
kubectl -n <namespace> annotate statefulset <my-cluster>-zookeeper strimzi.io/manual-zk-
scaling=false --overwrite
```

AMQ Streams 1.4 クラスターの ZooKeeper 3.5.7 サーバーのスケールダウン

この手順では、AMQ Streams が **namespace** namespaceで実行され、Kafka クラスターの名前が **my-cluster** であることを仮定します。

ZooKeeper ノードを削除する際、番号が最も大きいサーバーから降順で削除されます。したがって、5 つのノードで構成されるクラスターを 3 つのノードにスケールダウンする場合、**zookeeper-4** および **zookeeper-3** が削除され、**zookeeper-0**、**zookeeper-1**、および **zookeeper-2** が維持されます。



注記

次に進む前に、ZooKeeper ドキュメント の「Removing servers」にある注記をお読みください。

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

1. スケールダウンの後も**維持された**ノードのいずれかで、**zookeeper-shell** にログインします 。

```
kubectl exec -n <namespace> -it <my-cluster>-zookeeper-0 -c zookeeper --
./bin/zookeeper-shell.sh localhost:21810
```

注記

ZooKeeper ノード **zookeeper-x** の名前にあるインデックス番号は、**zookeeper-shell.sh localhost:2181x** コマンドにあるクライアントポートの最後の番号と一致します。

2. **config** コマンドを使用して既存のクラスター設定を出力します。

```
config
```

5 つの ZooKeeper ノードがあるクラスターからスケールダウンする場合、コマンドの出力は以下のようになります。

```
server.1=127.0.0.1:28880:38880:participant;127.0.0.1:21810
server.2=127.0.0.1:28881:38881:participant;127.0.0.1:21811
server.3=127.0.0.1:28882:38882:participant;127.0.0.1:21812
server.4=127.0.0.1:28883:38883:participant;127.0.0.1:21813
server.5=127.0.0.1:28884:38884:participant;127.0.0.1:21814
version=100000057
```

3. 次に、番号が一番大きいサーバーを最初に削除します。この場合は **server.5** になります。

```
reconfig -remove 5
```

これにより、クォーラムの他のメンバーすべてに伝播される新しい設定が出力されます。

```
server.1=127.0.0.1:28880:38880:participant;127.0.0.1:21810
server.2=127.0.0.1:28881:38881:participant;127.0.0.1:21811
server.3=127.0.0.1:28882:38882:participant;127.0.0.1:21812
server.4=127.0.0.1:28883:38883:participant;127.0.0.1:21813
version=200000012
```

4. 伝播が完了したら、**Kafka** リソースの **zookeeper** セクションのレプリカ数を 1 つ減らすことができます。これにより、**zookeeper-4** (**server.5**) がシャットダウンされます。

5. ステップ 1 から 4 を繰り返し、クラスターのサイズを段階的に縮小します。**必ずサーバーを降順に削除してください**。

6. 希望するサイズのクラスターがある場合、ZooKeeper クラスターを再度ロールできることを Cluster Operator に通信する必要があります。これには、**Kafka** カスタムリソースで **manual-zk-scaling** アノテーションを **false** に設定します。ZooKeeper レプリカの数を変更すると、Cluster Operator によって自動的に **true** に設定されます。

```
kubectl -n <namespace> annotate statefulset <my-cluster>-zookeeper strimzi.io/manual-zk-scaling=false --overwrite
```

注記

複数のサーバーを指定して一度に削除することもできます。たとえば、**reconfig -remove 4,5** を入力して一度に最も大きな番号のサーバーを 2 つ削除し、一度に 5 から 3 にスケールダウンすることも可能です。ただし、これは不安定になる可能性があるため、**推奨されません**。

第7章 サポートされる統合製品

AMQ Streams 1.4 は、以下の Red Hat 製品との統合をサポートします。

- OAuth 2.0 認証用の **Red Hat Single Sign-On 7.3**(およびテクノロジープレビューの Keycloak を使用した OAuth 2.0 承認)
- Kafka ブリッジをセキュアにし、追加の API 管理機能を提供する **Red Hat 3scale API Management 2.6**。
- データベースを監視し、イベントストリームを作成する **Red Hat Debezium 1.0 以上**。
- データストリーミングのサービススキーマの集中型ストアとする **Service Registry 2020-04 およびそれ以降 (テクノロジープレビュー)**。

これらの製品によって AMQ Streams デプロイメントに導入可能な機能の詳細は、AMQ Streams 1.4 のドキュメントを参照してください。

第8章 重要なリンク

- Red Hat AMQ 7 でサポートされる構成
- Red Hat AMQ 7 Component Details

改訂日時: 2021-02-10 01:38:35 UTC