aws

開発者ガイド

Amazon Kinesis Data Streams

Amazon Kinesis Data Streams: 開発者ガイド

Table of Contents

Amazon Kinesis Data Streams とは

データレコードの大量のストリームをリアルタイムで収集し、処理するには、Amazon Kinesis Data Streams を使用します。Kinesis Data Streams アプリケーションと呼ばれるデータ処理アプリケーションを作成できます。一般的な Kinesis Data Streams アプリケーションは、データストリームのデータをデータレコードとして読み取ります。これらのアプリケーションは Kinesis Client Library を使用できます。また Amazon EC2 インスタンスで実行できます。処理されたレコードは、ダッシュボードに送信してアラートの生成や、料金設定と広告戦略の動的変更に使用できるほか、他のさまざまな AWS のサービスにデータを送信できます。Kinesis Data Streams の機能と料金については、Amazon Kinesis Data Streamsを参照してください。

Kinesis Data Streams は、Firehose 、Kinesis Video Streams、Managed Service for Apache Flink とともに Kinesis ストリーミングデータプラットフォームの一部です。 https://docs.aws.amazon.com/kinesisanalytics/latest/dev/

AWS ビッグデータソリューションの詳細については、「 のビッグデータ AWS」を参照してください。 AWS ストリーミングデータソリューションの詳細については、ストリーミングデータとは?を参照してください。

トピック

- Kinesis Data Streams でどのようなことができますか?
- Kinesis Data Streams を使用する利点
- 関連 サービス

Kinesis Data Streams でどのようなことができますか?

Amazon Kinesis Data Streams を使用して、高速かつ継続的にデータの取り込みと集約を行うことができます。使用されるデータには、IT インフラストラクチャのログデータ、アプリケーションのログ、ソーシャルメディア、マーケットデータフィード、ウェブのクリックストリームデータなどの種類があります。データの取り込みと処理の応答時間はリアルタイムであるため、処理は一般的に軽量です。

以下に示しているのは、Kinesis Data Streams の一般的なシナリオです。

ログとデータフィードの取り込みと処理の高速化

プロデューサーからストリームにデータを直接プッシュさせることができます。たとえば、システムとアプリケーションのログをプッシュすると、数秒で処理可能になります。これにより、フロントエンドサーバーやアプリケーションサーバーに障害が発生した場合に、ログデータが失われることを防止できます。Kinesis Data Streams では、取り込み用にデータを送信する前にサーバーでデータがバッチ処理されないように、データフィードの取り込みが加速されます。

リアルタイムのメトリクスとレポート作成

Kinesis Data Streams に取り込んだデータを使用して、リアルタイムのデータ分析とレポート作成を簡単に行うことができます。たとえば、データ処理アプリケーションは、バッチデータを受け取るまで待つのではなく、データのストリーミング中にシステムおよびアプリケーションのログに関するメトリクスやレポート作成を操作できます。

リアルタイムデータ分析

これにより、並行処理の能力がリアルタイムデータの価値と同時に得られます。例えば、ウェブサイトのクリックストリームをリアルタイムで処理し、さらに並行して実行される複数の異なる Kinesis Data Streams アプリケーションを使用して、サイトの使いやすさの関与を分析します。

複雑なストリーム処理

Kinesis Data Streams アプリケーションとデータストリームの Directed Acyclic Graphs (DAG) を作成できます。通常、ここでは複数の Kinesis Data Streams アプリケーションから別のストリームにデータを出力し、別の Kinesis Data Streams アプリケーションによって下流処理が行われるようにします。

Kinesis Data Streams を使用する利点

Kinesis Data Streams は、さまざまなデータストリーミングの問題解決に使用できますが、一般的にデータのリアルタイム集計にも使用できます。集計データはその後でデータウェアハウスや MapReduce クラスターに読み込むことができます。

データは Kinesis Data Streams に取り込まれるため、耐久性と伸縮性が確保されます。レコードがストリームに入力されてから取得 (put-to-get 遅延) できるまでの遅延は、通常 1 秒未満です。つまり、Kinesis Data Streams アプリケーションにデータが追加されると同時にストリームのデータを利用し始めることができます。Kinesis Data Streams はマネージドサービスであるため、データ取り込みパイプラインの作成と実行にかかわる運用負荷が軽くなります。MapReduce 型のストリーミングアプリケーションを作成することができます。Kinesis Data Streams は伸縮性に優れており、スト

リームをスケールアップまたはスケールダウンできるため、有効期限が切れる前にデータレコードがなくなることはありません。

複数の Kinesis Data Streams アプリケーションを使用して、ストリームからデータを消費できるため、アーカイブや処理のような複数のアクションを同時に独立して実行できます。たとえば、2 つのアプリケーションが、同じストリームからデータを読み取ることができます。最初のアプリケーションは、集計を実行して計算し、Amazon DynamoDB テーブルを更新します。2 番目のアプリケーションは、データを圧縮して Amazon Simple Storage Service (Amazon S3) などのデータストアにアーカイブします。集計が実行中の DynamoDB テーブルは、 up-to-the-minute レポート用のダッシュボードによって読み取られます。

Kinesis Client Library を使用すると、耐障害性を維持しながらストリームのデータを利用でき、Kinesis Data Streams アプリケーションのスケーリングも可能になります。

関連 サービス

Amazon EMR クラスターを使用して Kinesis Data Streams を直接読み取って処理する方法については、Kinesis コネクタを参照してください。

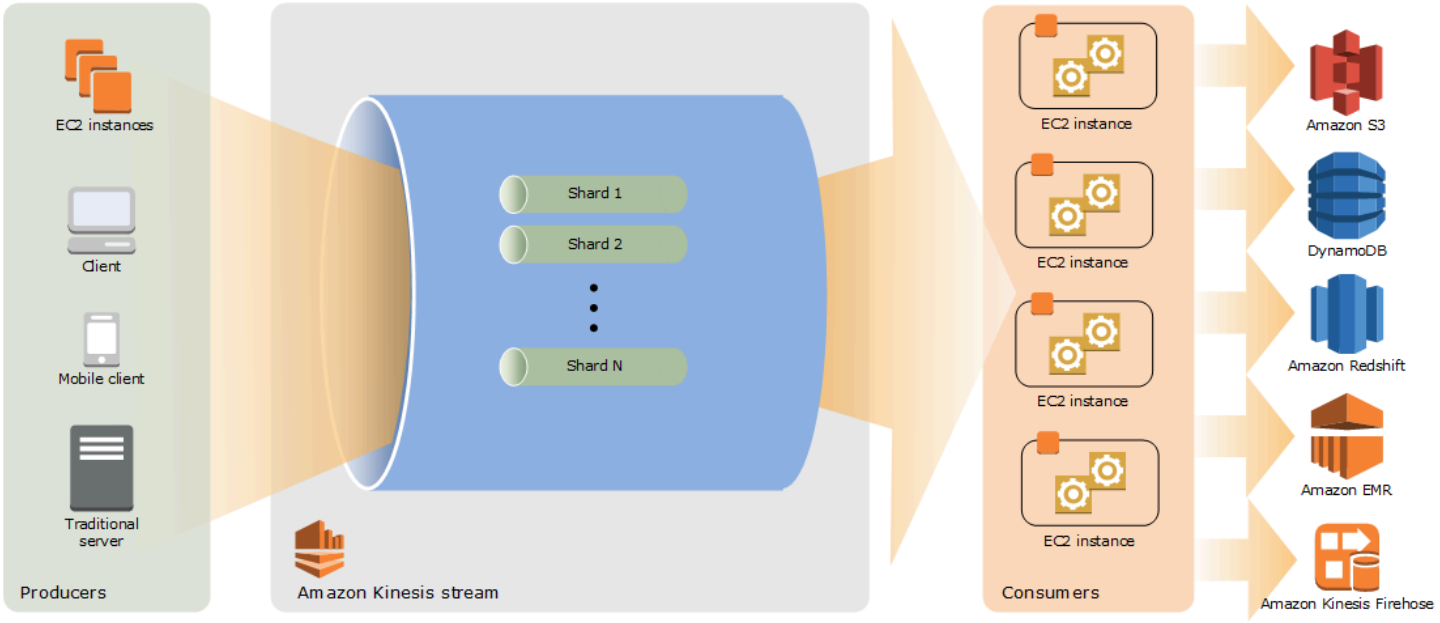
Amazon Kinesis Data Streams の用語と概念

Amazon Kinesis Data Streams を使用し始めると、アーキテクチャと用語を理解していることが強みになります。

トピック

Kinesis Data Streams のアーキテクチャの概要

以下の図に、Kinesis Data Streams のアーキテクチャの概要を示します。プロデューサーは継続的にデータを Kinesis Data Streams にプッシュし、コンシューマーはリアルタイムでデータを処理します。コンシューマー (Amazon EC2 で実行されているカスタムアプリケーションや Amazon Data Firehose 配信ストリームなど) は、Amazon DynamoDB、Amazon Redshift、Amazon S3 などの AWS サービスを使用して結果を保存できます。



Kinesis Data Streams の用語集

Kinesis Data Streams

Kinesis data stream は、シャードのセットです。各シャードにはデータレコードのシーケンスがあります。各データレコードには、Kinesis Data Streams によってシーケンス番号が割り当てられます。

データレコード

Kinesis data stream によって保存されるデータの単位は、データレコードです。データレコードは、シーケンス番号、パーティションキー、データ BLOB (変更不可のバイトシーケンス) で構成されます。Kinesis Data Streams ではいずれの方法でも BLOB のデータが検査、解釈、または変更されることはありません。データ BLOB は最大 1 MB です。

容量モード

データストリーム容量モードは、容量の管理方法と、データストリームの使用に対する課金方法を決定します。現在、Kinesis Data Streams では、データストリームのオンデマンドモードとプロビジョンドモードのいずれかを選択できます。詳細については、「データストリーム容量モードの選択」を参照してください。

オンデマンドモードでは、Kinesis Data Streams は必要なスループットを提供するために、シャードを自動的に管理します。使用した実際のスループットに対してのみ課金されます。Kinesis Data Streams は、ワークロードのスループットニーズが上昇または低下したときに自動的に対応します。詳細については、オンデマンドモードを参照してください。

プロビジョンドモードでは、データストリームのシャードカウントを指定する必要があります。データストリームの総容量は、シャードの容量の合計です。必要に応じて、データストリームのシャードの数を増減することができ、シャードカウントに対して時間料金が発生します。詳細については、プロビジョニングモードを参照してください。

保持期間

保持期間は、データレコードがストリームに追加された後にデータレコードにアクセスできる時間の長さです。ストリームの保持期間は、デフォルトで作成後 24 時間に設定されます。IncreaseStreamRetentionPeriod オペレーションを使用して保持期間を最大 8,760 時間 (365 日) まで延長し、 DecreaseStreamRetentionPeriodオペレーションを使用して保持期間を最小 24 時間に

短縮できます。24 時間を超える保持期間が設定されたストリームには追加料金が適用されます。詳細については、「Amazon Kinesis Data Streams の料金」を参照してください。

プロデューサー

プロデューサーは、Amazon Kinesis Data Streams にレコードを配置します。たとえば、ストリームにログデータを送信するウェブサーバーはプロデューサーです。

コンシューマー

コンシューマーは、Amazon Kinesis Data Streams からレコードを取得して処理します。これらのコンシューマーは Amazon Kinesis Data Streams アプリケーション と呼ばれます。

Amazon Kinesis Data Streams アプリケーション

Amazon Kinesis Data Streams application はストリームのコンシューマーで、一般的に EC2 インスタンスのフリートで実行されます。

開発可能なコンシューマーには、共有ファンアウトコンシューマーと拡張ファンアウトコンシューマーの 2 種類あります。両者間の相違点を確認する方法、各種類のコンシューマーを作成する方法については、Amazon Kinesis Data Streams からのデータの読み取りを参照してください。

Kinesis Data Streams アプリケーションの出力を別のストリームの入力にすることで、リアルタイムにデータを処理する複雑なトポロジを作成できます。アプリケーションは、他のさまざまな AWS のサービスにデータを送信することもできます。複数のアプリケーションが 1 つのストリームを使用して、各アプリケーションが同時にかつ独立してストリームからデータを消費できます。

シャード

シャードは、ストリーム内の一意に識別されたデータレコードのシーケンスです。ストリームは複数のシャードで構成され、各シャードが容量の 1 単位になります。各シャードは、読み取りで最大 5 トランザクション/秒、最大合計データ読み取り速度 2 MB/秒、書き込みで最大 1,000 レコード/秒、最大合計データ書き込み速度 1 MB/秒をサポートできます (パーティションキーを含む)。ストリームのデータ容量は、ストリームに指定したシャードの数によって決まります。ストリームの総容量はシャードの容量の合計です。

データ転送速度が増加した場合、ストリームに割り当てられたシャードカウントを増やしたり、減らしたりできます。詳細については、ストリームをリシャーディングするを参照してください。

パーティションキー

パーティションキーは、ストリーム内のデータをシャード単位でグループ化するために使用されます。Kinesis Data Streams では、ストリームに属するデータレコードを複数のシャードに配分します。この際、各データレコードに関連付けられたパーティションキーを使用して、配分先のシャードを決定します。パーティションキーは Unicode 文字列で、各キーの最大長は 256 文字に制限されています。MD5 ハッシュ関数を使用して、パーティションキーを 128 ビットの整数値にマッピングし、関連付けられたデータレコードをシャードにマッピングします。後者のマッピングにはシャードのハッシュキー範囲を使用します。アプリケーションは、ストリームにデータを配置するときに、パーティションキーを指定する必要があります。

シーケンス番号

各データレコードには、シャード内のパーティションキーごとに一意のシーケンス番号があります。`client.putRecords` または `client.putRecord` を使用してストリームに書き込むと、Kinesis Data Streams によってシーケンス番号が割り当てられます。同じパーティションキーのシーケンス番号は、通常徐々に増加されます。書き込みリクエスト間の期間が長くなるほど、シーケンス番号は大きくなります。

> **Note**
>
> シーケンス番号は、同じストリーム内の一連のデータのインデックスとして使用することはできません。一連のデータを論理的に区別するには、パーティションキーを使用するか、データセットごとに個別のストリームを作成します。

Kinesis Client Library

Kinesis Client Library をアプリケーションにコンパイルすることで、耐障害性を維持しながらストリームからデータを消費できます。Kinesis Client Library により、シャードごとにその実行と処理用のレコードプロセッサが確保されます。また、ストリームからのデータの読み取りが簡素化されます。Kinesis Client Library は、Amazon DynamoDB テーブルを使用してコントロールデータを保存します。また、データを処理するアプリケーションごとに 1 つのテーブルを作成します。

Kinesis Client Library には 2 種類のメジャーバージョンがあります。使用するバージョンは、作成するコンシューマーの種類によって異なります。詳細については、Amazon Kinesis Data Streams からのデータの読み取りを参照してください。

アプリケーション名

Amazon Kinesis Data Streams アプリケーションの名前によって、アプリケーションが識別されます。各アプリケーションには、アプリケーションが使用する AWS アカウントとリージョンを対象とする一意の名前が必要です。この名前は、Amazon DynamoDB のコントロールテーブルと Amazon CloudWatch メトリクスの 名前空間の名前として使用されます。

サーバー側の暗号化

Amazon Kinesis Data Streams は、プロデューサーがストリームに入力するときに、機密データを自動的に暗号化できます。Kinesis Data Streams は暗号化に AWS KMS マスターキーを使用します。詳細については、Amazon Kinesis Data Streams のデータ保護を参照してください。

> **Note**
>
> 暗号化されたストリームに対して読み書きを行うために、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にはマスターキーへのアクセス許可が必要です。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にアクセス許可を付与する方法については、the section called “ユーザー生成の KMS マスターキーを使用するための許可”を参照してください。

> **Note**
>
> サーバー側の暗号化を使用すると、 AWS Key Management Service （ AWS KMS) のコストが発生します。詳細については、AWS Key Management Service の料金を参照してください。

クォータと制限

Amazon Kinesis Data Streams には、次のストリームクォータ、シャードクォータ、および制限があります。

クォータ	オンデマンドモード	プロビジョニングモード
データストリームの数	AWS アカウント内のストリーム数には上限クォータはありません。オンデマンドキャパシティモードでは、デフォルトで最大 50 のデータストリームを作成できます。このクォータの引き上げが必要な場合は、サポートチケットを発行してください。	プロビジョニングモードでは、アカウント内のストリームの数にクォータ上限はありません。
シャード数	上限はありません。シャードの数は、取り込まれたデータの量と、必要なスループットのレベルに応じて異なります。Kinesis Data Streams は、データ量とトラフィックの変化に対応して、シャードの数を自動的にスケールします。	上限はありません。デフォルトのシャードクォータは、米国東部 (バージニア北部)、米国西部 (オレゴン)、欧州 (アイルランド) の各 AWS リージョンで、 AWS アカウントごとに 500 シャードです。 そのほかのすべてのリージョンでは、デフォルトのシャードクォータは AWS アカウントあたり 200 シャードです。shards-per-data ストリームクォータの引き上げをリクエストするには、「クォータの引き上げのリクエスト」を参照してください。
データストリームのスループット	デフォルトで、オンデマンドキャパシティモードで作成された新しいデータストリーム	上限はありません。最大スループットは、ストリーム用にプロビジョニングされた

クォータ	オンデマンドモード	プロビジョニングモード
	では、書き込みスループットが 4 MB/秒、読み取りスループットが 8 MB/秒になっています。オンデマンドキャパシティモードのデータストリームは、トラフィックの増加に合わせて最大 200 MB/秒の書き込みスループット、および最大 400 MB/秒の読み取りスループットまでスケールされます。書き込みキャパシティを 2 GB/秒、読み取りキャパシティを 4 GB/秒に増加させる必要がある場合は、サポートチケットを送信してください。	シャードの数に応じて異なります。各シャードは、最大 1 MB/秒もしくは 1,000 レコード/秒の書き込みスループット、または最大 2 MB/秒もしくは 2,000 レコード/秒の読み取りスループットをサポートできます。より多くの取り込み容量が必要な場合は、AWS Management Console または UpdateShardCount API を使用して、ストリーム内のシャードの数を簡単にスケールアップできます。
データペイロードのサイズ	`base64-encoding` 前のレコードのデータペイロードサイズは、最大で 1 MB です。	
`GetRecords` トランザクションのサイズ	GetRecords は、単一のシャードから呼び出しごとに最大 10 MB のデータを取得し、呼び出しごとに最大 10,000 レコードを取得できます。`GetRecords` への各呼び出しは、1 つの読み込みトランザクションとしてカウントされます。各シャードは 1 秒あたり最大 5 件のトランザクションをサポートできます。各読み込みトランザクションは最大 10,000 レコードを提供でき、トランザクションあたり 10 MB のクォータ上限があります。	
シャードあたりのデータ読み取りレート	各シャードは、 を介して 1 秒あたり最大 2 MB の合計データ読み取りレートをサポートできますGetRecords。`GetRecords` への呼び出しで 10 MB が返される場合、次の 5 秒以内に行われたそれ以降の呼び出しでは、例外がスローされます。	
登録されたコンシューマーのデータストリームあたりの数	登録されたコンシューマーは、データストリームごとに最大 20 個 (拡張ファンアウトの上限) 作成することができます。	

クォータ	オンデマンドモード	プロビジョニングモード
プロビジョニングモードとオンデマンドモードの切り替え	AWS アカウントのデータストリームごとに、オンデマンドキャパシティモードとプロビジョニングキャパシティモードを 24 時間以内に 2 回切り替えることができます。	

API の制限

ほとんどの AWS APIsと同様に、Kinesis Data Streams API オペレーションはレート制限されています。リージョンごとに AWS アカウントごとに次の制限が適用されます。Kinesis Data Streams API の詳細については、Amazon Kinesis Streams API リファレンスを参照してください。

KDS コントロールプレーン API の制限

次のセクションでは、KDS コントロールプレーン API の制限を示します。KDS コントロールプレーン API を使用すると、データストリームを作成および管理できます。これらの制限は、リージョンごと、 AWS アカウントごとに適用されます。

コントロールプレーン API の制限

API	API コールの制限	アカウントあたり/ストリームあたり	説明
AddTagsToStream	1 秒あたり 5 件のトランザクション (TPS)	アカウントあたり	データストリームあたり 50 個のタグ
CreateStream	5 TPS	アカウントあたり	アカウントに存在できるストリームの数にクォータ上限はありません。CreateStream リクエストを行うときに以下のいずれかを実行しようとすると、LimitExceededException が発生します。

API	API コールの制限	アカウントあたり/ストリームあたり	説明
			• 特定の時点で、5 つを超える CREATING 状態のストリームを持つ。 • アカウントに許可されている数よりも多くのシャードを作成する。
DecreaseStreamRetentionPeriod	5 TPS	ストリームあたり	データストリームの保持期間の最小値は 24 時間です。
DeleteResourcePolicy	5 TPS	アカウントあたり	この制限の引き上げが必要な場合は、サポートチケットを発行してください。
DeleteStream	5 TPS	アカウントあたり	
DeregisterStreamConsumer	5 TPS	ストリームあたり	
DescribeLimits	1 TPS	アカウントあたり	
DescribeStream	10 TPS	アカウントあたり	
DescribeStreamConsumer	20 TPS	ストリームあたり	
DescribeStreamSummary	20 TPS	アカウントあたり	

API	API コールの制限	アカウントあたり/ストリームあたり	説明
DisableEnhancedMonitoring	5 TPS	ストリームあたり	
EnableEnhancedMonitoring	5 TPS	ストリームあたり	
GetResourcePolicy	5 TPS	アカウントあたり	この制限の引き上げが必要な場合は、サポートチケットを発行してください。
IncreaseStreamRetentionPeriod	5 TPS	ストリームあたり	ストリームの保持期間の最大値は 8760 時間 (365 日) です。
ListShards	1,000 TPS	ストリームあたり	
ListStreamConsumers	5 TPS	ストリームあたり	
ListStreams	5 TPS	アカウントあたり	
ListTagsForStream	5 TPS	ストリームあたり	
MergeShards	5 TPS	ストリームあたり	プロビジョニングされたもののみに適用されます。
PutResourcePolicy	5 TPS	アカウントあたり	この制限の引き上げが必要な場合は、サポートチケットを発行してください。

API	API コールの制限	アカウントあたり/ストリームあたり	説明
RegisterStreamConsumer	5 TPS	ストリームあたり	コンシューマーはデータストリームごとに最大 20 登録できます。所定のコンシューマーを登録できるのは、一度に 1 つのデータストリームだけです。同時に作成できるコンシューマーは 5 つだけです。つまり、5 つを超える CREATING ステータスのコンシューマーを同時に所有することはできません。CREATING ステータスのコンシューマーが 5 つ存在する間の 6 番目のコンシューマーの登録
RemoveTagsFromStream	5 TPS	ストリームあたり	
SplitShard	5 TPS	ストリームあたり	プロビジョニングされたもののみに適用されます
StartStreamEncryption		ストリームあたり	サーバー側の暗号化に新しい AWS KMS キーを 24 時間で 25 回正常に適用できます。

API	API コールの制限	アカウントあたり/ストリームあたり	説明
StopStreamEncryption		ストリームあたり	24 時間のローリング期間あたり、サーバー側の暗号化を 25 回無効にすることができます。
UpdateShardCount		ストリームあたり	プロビジョニングされたもののみに適用されます。シャード数のデフォルト上限は 10,000 です。この API には追加の制限があります。詳細については、「」を参照してくださいUpdateShardCount。
UpdateStreamMode		ストリームあたり	AWS アカウントのデータストリームごとに、オンデマンドキャパシティモードとプロビジョニングキャパシティモードを 24 時間以内に 2 回切り替えることができます。

KDS データプレーンの API の制限

次のセクションでは、KDS データプレーン API の制限について説明します。KDS データプレーン API を使用すると、データストリームを使用してデータレコードをリアルタイムで収集および処理できます。これらの制限は、データストリーム内のシャードごとに適用されます。

データプレーンの API の制限

API	API コールの制限	ペイロードの制限	その他の詳細
GetRecords	5 TPS	呼び出しごとに返されるレコードの最大数は 10,000 です。GetRecords が返すことができるデータの最大サイズは、10 MB です。	コールがこの量のデータを返す場合、次の 5 秒以内に行われる後続のコールは ProvisionedThroughputExceededException をスローします。ストリームでプロビジョニングされたスループットが不十分である場合、次の 1 秒以内に行われる後続のコールは ProvisionedThroughputExceededException をスローします。
GetShardIterator	5 TPS		シャードイテレーターはリクエスタに返されてから 5 分後に有効期限が切れます。 GetShardIterator リクエストが頻繁に行われる場合は、 を受け取ります

API	API コールの制限	ペイロードの制限	その他の詳細
			ProvisionedThroughputExceededException。
PutRecord	1,000 TPS	各シャードは、1 秒あたり 1,000 レコードまでの書き込みをサポートし、1 秒あたり 1 MB の最大データ書き込みをサポートします。	
PutRecords		各 PutRecords リクエストは、最大 500 個のレコードをサポートできます。リクエストに含まれる各レコードは 1 MB、リクエスト全体の上限はパーティションキーを含めて最大 5 MB。各シャードは、1 秒あたり 1,000 レコードまでの書き込みをサポートし、1 秒あたり 1 MB の最大データ書き込みをサポートします。	

API	API コールの制限	ペイロードの制限	その他の詳細
SubscribeToShard	シャード SubscribeToShard ごとに、登録されたコンシューマーごとに 1 秒あたり 1 回の呼び出しを行うことができます。		同じ ConsumerARN で SubscribeToShard を呼び出し、成功してから 5 秒ShardId 以内に再度呼び出すと、 が返されますResourceInUseException。

クォータの引き上げ

クォータが調整可能な場合は、Service Quotas を使用してクォータの引き上げを要求できます。一部のリクエストは自動的に解決され、その他のリクエストは AWS サポートに送信されます。 AWS サポートに送信されたクォータ引き上げリクエストのステータスを追跡できます。サービスクォータを引き上げるリクエストは、優先サポートを受けません。緊急のリクエストがある場合は、 AWS サポートにお問い合わせください。詳細については、What Is Service Quotas?を参照してください。

サービスクォータの引き上げをリクエストするには、Requesting a Quota Increaseで説明している手順に従います。

Amazon Kinesis Data Streams のセットアップ

Amazon Kinesis Data Streams を初めて使用する場合は、事前に以下のタスクをすべて実行してください。

タスク

- にサインアップ AWS
- ライブラリとツールをダウンロードする
- 開発環境を設定する

にサインアップ AWS

Amazon Web Services (AWS) にサインアップすると、 AWS アカウントは Kinesis Data Streams を含むのすべてのサービスに自動的にサインアップされます。 AWS料金は、使用するサービスの料金のみが請求されます。

AWS 既にアカウントをお持ちの場合は、次のタスクに進んでください。 AWS アカウントをお持ちでない場合は、以下の手順に従ってアカウントを作成してください。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   にサインアップすると AWS アカウント、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティ上のベストプラクティスとして、ユーザーに管理アクセスを割り当て、root ユーザーアクセスを必要とするタスクを実行するときは root ユーザーのみを使用してください。

ライブラリとツールをダウンロードする

以下のライブラリとツールは Kinesis Data Streams での作業に役立ちます。

- Amazon Kinesis API リファレンスは、Kinesis Data Streams でサポートされている基本的なオペレーションのセットです。Java コードを使用した基本的なオペレーションの実行の詳細については、次を参照してください。
  - Amazon Kinesis Data Streams API と AWS SDK for Java を使用したプロデューサーの開発
  - AWS SDK for Java を使用した共有スループットでのカスタムコンシューマーの開発
  - ストリームの作成と管理
- Go、Java、.NET、Node.js JavaScript、PHP、Python、Ruby の AWS SDK には Kinesis Data Streams サポートとサンプルが含まれています。ご使用のバージョンに Kinesis Data Streams AWS SDK for Java のサンプルが含まれていない場合は、GitHubからダウンロードすることもできます。
- Kinesis クライアントライブラリ (KCL) は、 easy-to-use データを処理するためのプログラミングモデルを提供します。KCL では、Kinesis Data Streams を Java、Node.js、.NET、Python、Ruby ですぐに使い始めることができます。詳細については、ストリームからのデータの読み取りを参照してください。
- AWS Command Line Interface は、Kinesis Data Streams をサポートしています。 AWS CLI これにより、 AWS コマンドラインから複数のサービスを制御し、スクリプトを使用してそれらを自動化できます。

開発環境を設定する

KCL を使用するには、Java 開発環境が以下の要件を満たしている必要があります。

- Java 1.7 (Java SE 7 JDK) 以降。最新の Java ソフトウェアは、Oracle ウェブサイトのJava SE ダウンロードからダウンロードできます。
- Apache Commons パッケージ (コード、HTTP クライアント、ログ記録)。
- Jackson JSON プロセッサ

AWS SDK for Java では、サードパーティーフォルダに Apache Commons と Jackson が含まれています。ただし、SDK for Java は Java 1.6 で動作しますが、Kinesis Client Library には Java 1.7 が必要です。

Amazon Kinesis Data Streams の開始方法

このセクションの情報は、Amazon Kinesis Data Streams の使用を開始する上で役立ちます。Kinesis Data Streams を初めて利用する場合は、Amazon Kinesis Data Streams の用語と概念で説明されている概念と用語について理解することから始めてください。

このセクションでは、AWS Command Line Interface を使用して基本的な Amazon Kinesis Data Streams オペレーションを実行する方法を示します。Kinesis Data Streams データフローの基本原則と Kinesis data stream でのデータの入力や取得に必要なステップについて説明します。

トピック

- AWS CLI のインストールと設定
- AWS CLI を使用した基本的な Kinesis Data Stream オペレーションの実行

CLI アクセスには、アクセスキー ID とシークレットアクセスキーが必要です。長期のアクセスキーの代わりに一時的な認証情報をできるだけ使用します。一時的な認証情報には、アクセスキー ID、シークレットアクセスキー、および認証情報の失効を示すセキュリティトークンが含まれています。詳細については、IAM ユーザーガイドの「AWS リソースを使用した一時的なセキュリティ認証情報の使用」を参照してください。

IAM およびセキュリティキーの詳細なセットアップ手順については、IAM ユーザーを作成するを参照してください。

このセクションで説明する特定のコマンドは、特定の値が実行のたびに異なる場合を除き、そのまま使用します。また、この例では米国西部 (オレゴン) リージョンを使用していますが、このセクションの手順は Kinesis Data Streams がサポートされているリージョンのいずれにおいても機能します。

AWS CLI のインストールと設定

AWS CLI をインストールする

Windows 用と、Linux、OS X、および UNIX オペレーティングシステム用の AWS CLI をインストールする方法の詳しいステップについては、AWS CLI のインストールを参照してください。

使用可能なオプションとサービスのリストを表示するには、次のコマンドを使用します。

```
aws help
```

使用する Kinesis Data Streams サービスでは、次のコマンドを実行して、Kinesis Data Streams に関連する AWS CLI サブコマンドを確認できます。

```
aws kinesis help
```

このコマンドの出力には、使用できる Kinesis Data Streams コマンドが含まれます。

```
AVAILABLE COMMANDS

       o add-tags-to-stream

       o create-stream

       o delete-stream

       o describe-stream

       o get-records

       o get-shard-iterator

       o help

       o list-streams

       o list-tags-for-stream

       o merge-shards

       o put-record

       o put-records

       o remove-tags-from-stream

       o split-shard

       o wait
```

このコマンドリストは、Amazon Kinesis Service API リファレンスで説明されている Kinesis Data Streams API に対応しています。たとえば、`create-stream` コマンドは `CreateStream` API アクションに対応します。

これで AWS CLI は正常にインストールされましたが、まだ設定されていません。これについては、次のセクションで説明します。

AWS CLI を設定する

一般的な使用の場合、`aws configure` コマンドが、AWS CLI のインストールをセットアップするための最も簡単な方法です。詳細については、AWS CLI の設定を参照してください。

AWS CLI を使用した基本的な Kinesis Data Stream オペレーションの実行

このセクションでは、AWS CLI を使用した、コマンドラインからの Kinesis data stream の基本的な使用方法について説明します。Amazon Kinesis Data Streams の用語と概念で説明されている概念を理解している必要があります。

> **Note**
>
> Kinesis Data Streams は AWS の無料利用枠の対象外であるため、ストリームの作成後は、Kinesis Data Streams の使用に対してアカウントに少額の料金が発生します。このチュートリアルを終了したら、AWS リソースを削除して料金が発生しないようにしてください。詳細については、ステップ 4: クリーンアップするを参照してください。

トピック

- ステップ 1: ストリームを作成する
- ステップ 2: レコードを入力する
- ステップ 3: レコードを取得する
- ステップ 4: クリーンアップする

ステップ 1: ストリームを作成する

最初のステップは、ストリームを作成し、正常に作成されたことを確認することです。次のコマンドを使用して、Fooという名前のストリームを作成します。

```
aws kinesis create-stream --stream-name Foo
```

次に、次のコマンドを実行して、ストリーム作成の進行状況を確認します。

```
aws kinesis describe-stream-summary --stream-name Foo
```

次の例のような出力が得られます。

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "CREATING",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

この例では、ストリームのステータスは CREATING であり、使用する準備が完全には整っていないことを意味します。しばらくしてからもう一度調べると、次の例のような出力が表示されます。

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
```

```
        ],
        "EncryptionType": "NONE",
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

この出力には、このチュートリアルで気にする必要がない情報も含まれています。ここで重要な項目は "StreamStatus": "ACTIVE" であり、ストリームを使用する準備ができたことと、リクエストした単一のシャードに関する情報が示されています。また、次に示すように、list-streams コマンドを使用して新しいストリームの存在を確認することもできます。

```
aws kinesis list-streams
```

出力:

```
{
    "StreamNames": [
        "Foo"
    ]
}
```

ステップ 2: レコードを入力する

アクティブなストリームができたら、データを入力できます。このチュートリアルでは、最もシンプルなコマンド put-record を使用して、"testdata" というテキストを含む単一のデータレコードをストリームに入力します。

```
aws kinesis put-record --stream-name Foo --partition-key 123 --data testdata
```

このコマンドが成功すると、出力は次の例のようになります。

```
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber": "49546986683135544286507457936321625675700192471156785154"
}
```

これで、ストリームにデータを追加できました。次にストリームからデータを取得する方法を説明します。

ステップ 3: レコードを取得する

GetShardIterator

ストリームからデータを取得するには、対象となるシャードのシャードイテレーターを取得する必要があります。シャードイテレーターは、コンシューマー (ここでは get-record コマンド) が読み取るストリームとシャードの位置を表します。次のように get-shard-iterator コマンドを使用します。

```
aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-iterator-type
 TRIM_HORIZON --stream-name Foo
```

aws kinesis コマンドの背後には Kinesis Data Streams API があります。示されているパラメータに関心がある場合は、GetShardIterator API のリファレンスのトピックを参照してください。実行に成功すると、出力は次の例のようになります (出力全体を表示するには、水平にスクロールします)。

```
{
    "ShardIterator": "AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd
+efGN2aHFdkH1rJl4BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="
}
```

ランダムに見える長い文字列がシャードイテレーターです (お客様のシャードイテレーターはこれとは異なります)。このシャードイテレーターをコピーして、次に示す get コマンドに貼り付ける必要があります。シャードイテレーターの有効期間は 300 秒です。これは、シャードイテレーターをコピーして次のコマンドに貼り付けるのに十分な時間です。次のコマンドに貼り付ける前に、シャードイテレーターから改行を削除する必要があることに注意してください。シャードイテレーターが有効ではないことを示すエラーメッセージが表示された場合は、もう一度 get-shard-iterator コマンドを実行します。

GetRecords

get-records コマンドはストリームからデータを取得し、Kinesis Data Streams API での GetRecords の呼び出しに解決します。シャードイテレーターは、データレコードの逐次読み取りを開始する、シャード内の位置を指定します。イテレーターが指定するシャードの位置にレコードがない場合、GetRecords は空のリストを返します。シャード内のレコードが含まれる位置に到達するために、複数の呼び出しが必要になる場合があることに注意してください。

get-records コマンドの例を次に示します (出力全体を表示するには、水平にスクロールします)。

```
aws kinesis get-records --shard-iterator
 AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG4lNR1EMi
+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

bash など Unix タイプのコマンドプロセッサからこのチュートリアルを実行する場合は、次のように入れ子にしたコマンドを使用して、シャードイテレーターの取得を自動化できます (横方向にスクロールしてコマンド全体を表示)。

```
SHARD_ITERATOR=$(aws kinesis get-shard-iterator --shard-id shardId-000000000000 --
shard-iterator-type TRIM_HORIZON --stream-name Foo --query 'ShardIterator')

aws kinesis get-records --shard-iterator $SHARD_ITERATOR
```

PowerShell をサポートするシステムからこのチュートリアルを実行する場合、次のようなコマンドを使用してシャードのイテレータの取得を自動化できます (横方向にスクロールしてコマンド全体を表示)。

```
aws kinesis get-records --shard-iterator ((aws kinesis get-shard-iterator --shard-id
 shardId-000000000000 --shard-iterator-type TRIM_HORIZON --stream-name Foo).split('"')
[4])
```

get-records コマンドが正常に終了すると、次の例のように、シャードイテレーターを取得するときに指定したシャードに対応するストリーム内のレコードがリクエストされます (出力全体を表示するには、水平にスクロールします)。

```
{
  "Records":[ {
    "Data":"dGVzdGRhdGE=",
    "PartitionKey":"123”,
    "ApproximateArrivalTimestamp": 1.441215410867E9,
    "SequenceNumber":"49544985256907370027570885864065577703022652638596431874"
  } ],
  "MillisBehindLatest":24000,

 "NextShardIterator":"AAAAAAAAAAEDOW3ugseWPE4503kqN1yN1UaodY8unE0sYslMUmC6lX9hlig5+t4RtZM0/
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvPOZvrUIudb8UkH3V
```

```
}
```

上記で get-records をリクエストとして説明しましたが、これは、ストリーム内にレコードが存在する場合でもゼロ件以上のレコードが返される可能性があり、返されたレコードはストリーム内に現存するすべてのレコードを示していない可能性があることを意味します。これは完全に正常で、本稼働用のコードではストリームに対し、適切な間隔でレコードに対するポーリングを行います (このポーリング速度は、個々のアプリケーションの設計要件によって異なります)。

チュートリアルのこのパートでレコードについて最初に気づくのは、データがゴミのように見えることです。これは私たちが送ったクリアテキスト testdata ではありません。これは、バイナリデータを送信できるように、put-record では Base64 エンコーディングを使用しているためです。ただし、AWS CLI での Kinesis Data Streams のサポートでは、Base64 デコーディングを提供していません。これは、Base64 デコーディングされた raw バイナリコンテンツを stdout に出力すると、特定のプラットフォームやターミナルで、意図しない動作やセキュリティ上の問題が発生する可能性があるためです。Base64 デコーダ (https://www.base64decode.org/ など) を使用して手動で dGVzdGRhdGE= をデコードすると、これが実際に testdata であることを確認できます。このチュートリアルではこれで問題ありません。なぜなら、実際には、AWS CLI を使用してデータを利用することはまれであり、通常は前に示したようにストリームの状態をモニタリングしたり、情報を取得したりするために使用されるからです (describe-stream および list-streams)。後のチュートリアルでは、Kinesis Client Library (KCL) を使用して、本稼働品質のコンシューマーアプリケーションを構築する方法を示し、Base64 の処理も検討します。KCL の詳細については、KCL を使用したスループット共有カスタムコンシューマーの開発を参照してください。

get-records によって、常にストリーム/シャード内のすべてのレコードが返されるわけではありません。このような場合は、最後の結果から NextShardIterator を使用して、次のレコードのセットを取得します。したがって、大量のデータがストリームに入力されていた場合 (本稼働アプリケーションでの通常の状況)、毎回 get-records を使用してデータのポーリングを継続できます。ただし、300 秒のシャードイテレーターの有効期間内に次のシャードイテレーターを使用して get-records を呼び出した場合、エラーメッセージが表示され、get-shard-iterator コマンドを使用して最新のシャードイテレーターを取得する必要があります。

この出力には、MillisBehindLatest も含まれています。これは、ストリームの末尾から GetRecords オペレーションのレスポンスまでの時間 (ミリ秒) であり、コンシューマーの時間の現在の時刻からの遅れを示します。値ゼロはレコード処理が追いついて、現在処理する新しいレコードは存在しないことを示します。このチュートリアルの場合は、作業を進めるのに時間をかけていると、この数値がかなり大きくなる可能性があります。これは問題ではなく、デフォルトでは、データレコードはストリームに 24 時間留まり、取得されるのを待ちます。この期間は保持期間と呼ばれ、365 日 まで設定可能です。

get-records が成功したときの結果は、現在ストリームにこれ以上レコードが見つからない場合でも常に NextShardIterator です。これは、プロデューサーがどの時点でもストリームにレコードを入力している可能性があることを前提としたポーリングモデルです。独自のポーリングルーチンを記述することもできますが、開発中のコンシューマーアプリケーションで、前に説明した KCL を使用している場合、このポーリングによって処理が実行されます。

レコードをプルする対象のストリームまたはシャードにそれ以上レコードがなくなるまで get-records を呼び出すと、次の例のような空のレコードの出力が表示されます (出力全体を表示するには、水平にスクロールします)。

```
{
    "Records": [],
    "NextShardIterator": "AAAAAAAAAAGCJ5jzQNjmdhO6B/YDIDE56jmZmrmMA/r1WjoHXC/
kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wul/
EhyNeSs5DYXLSSC5XCapmCAYGFjYER69QSdQjxMmBPE/hiybFDi5qtkT6/PsZNz6kFoqtDk="
}
```

ステップ 4: クリーンアップする

最後に、ストリームを削除してリソースを解放し、前に説明したように、アカウントに対する意図しない料金が発生することを回避できます。ストリームを作成したが、使用する予定がない場合は、必ず実際にこれを行ってください。ストリームでデータを入力および取得したかどうかにかかわらず、ストリームごとに料金が発生するためです。クリーンアップコマンドはシンプルです。

```
aws kinesis delete-stream --stream-name Foo
```

成功しても出力はないため、describe-stream を使用して削除の進行状況を確認できます。

```
aws kinesis describe-stream-summary --stream-name Foo
```

delete コマンドの直後にこのコマンドを実行する場合、次の例のような出力部分が表示される可能性があります。

```
{
    "StreamDescriptionSummary": {
        "StreamName": "samplestream",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/samplestream",
        "StreamStatus": "ACTIVE",
```

ストリームが完全に削除されると、describe-stream はnot foundエラーを返します。

```
A client error (ResourceNotFoundException) occurred when calling the
 DescribeStreamSummary operation:
Stream Foo under account 123456789012 not found.
```

Amazon Kinesis Data Streams のチュートリアル例

このセクションのチュートリアル例は、Amazon Kinesis Data Streams の概念と機能を理解する上で役立つように設計されています。

トピック

- チュートリアル: KPL と KCL 2.x を使用した株式データのリアルタイム処理
- チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理
- チュートリアル: Flink アプリケーション向けの Managed Service for Apache Flink を使用してリアルタイムの株式データを分析する
- チュートリアル: AWS Lambda を Amazon Kinesis Data Streams で使用する
- AWS Streaming Data Solution for Amazon Kinesis

チュートリアル: KPL と KCL 2.x を使用した株式データのリアルタイム処理

このチュートリアルのシナリオでは、株式取引をデータストリームに取り込み、ストリーム上で計算を実行するシンプルな Amazon Kinesis Data Streams アプリケーションを記述する必要があります。レコードのストリームを Kinesis Data Streams に送信し、ほぼリアルタイムでレコードを消費および処理するアプリケーションを実装する方法を説明します。

> **Important**
>
> Kinesis Data Streams は AWS の無料利用枠の対象外であるため、ストリームの作成後は、Kinesis Data Streams の使用に対してアカウントに少額の料金が発生します。コンシューマーアプリケーションが起動すると、Amazon DynamoDB の使用に伴う料金がわずかに発生します。コンシューマーアプリケーションでは、処理状態を追跡する際に DynamoDB を使用します。このアプリケーションを終了したら、AWS リソースを削除して料金が発生しないようにしてください。詳細については、ステップ 7: 終了するを参照してください。

このコードでは、実際の株式市場データにアクセスする代わりに、株式取引のストリームをシミュレートします。シミュレーションには、2015 年 2 月時点における時価総額上位 25 社の株式に関する実際の市場データを基にしたランダム株式取引ジェネレーターが使用されています。リアルタイムの株式取引のストリームにアクセスできたとしたら、そのときに必要としている有益な統計を入手し

たいと考えるかもしれません。たとえば、スライディングウィンドウ分析を実行して、過去 5 分間に購入された最も人気のある株式を調べたいと思われるかもしれません。または、大規模な売り注文 (膨大な株式が含まれる売り注文) が発生したときに通知を受けたいと思われるかもしれません。このシリーズのコードを拡張して、このような機能を使用することもできます。

このチュートリアルにある手順をデスクトップやノートパソコンで実行し、同じマシンまたは定義された要件を満たす任意のプラットフォームで、プロデューサーおよびコンシューマーのコードのいずれも実行できます。

この例では、米国西部 (オレゴン) リージョンが使用されていますが、Kinesis Data Streams がサポートされる AWS リージョンであれば、いずれのリージョンでも動作します。

タスク

- 前提条件
- ステップ 1: データストリームの作成
- ステップ 2: IAM ポリシーとユーザーの作成
- ステップ 3: コードをダウンロードしてビルドする
- ステップ 4: プロデューサーを実装する
- ステップ 5: コンシューマーを実装する
- ステップ 6: (オプション) コンシューマーを拡張する
- ステップ 7: 終了する

前提条件

このチュートリアルを完了するには、以下の要件を満たす必要があります。

Amazon Web Services アカウント

開始する前に、Amazon Kinesis Data Streams の用語と概念で説明されている概念、特にストリーム、シャード、プロデューサー、コンシューマーについて理解しておきます。また、ガイドAWS CLI のインストールと設定の手順を完了しておくと役立ちます。

AWS Management Console にアクセスするときに、AWS アカウントとウェブブラウザが必要になります。

コンソールにアクセスするには、IAM ユーザー名とパスワードを使用して、IAM サインインページからAWS Management Consoleにサインインします。プログラム的なアクセスや、長期的な認

証情報に代わる手段を含めた AWS セキュリティ認証情報については、「IAM ユーザーガイド」の「AWS セキュリティ認証情報」を参照してください。AWS アカウントへのサインインに関する詳細については、「AWS サインイン User Guide」の「How to sign in to AWS」を参照してください。

IAM とセキュリティキーの設定手順の詳細については、IAM ユーザーを作成するを参照してください。

システムソフトウェア要件

アプリケーションの実行に使用するシステムには、Java 7 以降がインストールされている必要があります。最新の Java Development Kit (JDK) をダウンロードおよびインストールするには、Oracle 社の Java SE インストールサイトを参照してください。

Eclipse などの Java IDE を使用している場合は、ソースコードを開いて、編集、ビルド、実行できます。

最新バージョンの AWS SDK for Java が必要です。Eclipse を IDE として使用している場合は、AWS Toolkit for Eclipse を代わりにインストールできます。

コンシューマーアプリケーションには、Kinesis Client Library (KCL) バージョン 2.2.9 以上が必要です。これは、https://github.com/awslabs/amazon-kinesis-client/tree/master の GitHub から入手できます。

次のステップ

ステップ 1: データストリームの作成

ステップ 1: データストリームの作成

最初に、このチュートリアルの後の手順で使用するデータストリームを作成する必要があります。

ストリームを作成するには

1. AWS Management Consoleにサインインして、Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. ナビゲーションペインで、[データストリーム] を選択します。
3. ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
4. [Kinesis ストリームの作成] を選択します。
5. ストリームの名前 (**`StockTradeStream`** など) を入力します。

6. シャードカウントは**1**と入力しますが、[必要なシャードカウントの予想] は折りたたんだままにします。
7. [Kinesis ストリームの作成] を選択します。

[Kinesis streams] リストページで、作成中のストリームのステータスは CREATING になります。ストリームを使用する準備ができると、ステータスは ACTIVE に変わります。

ストリームの名前を選択すると、表示されるページの [Details (詳細)] タブにデータストリーム設定の概要が表示されます。[モニタリング] セクションには、ストリームのモニタリング情報が表示されます。

次のステップ

ステップ 2: IAM ポリシーとユーザーの作成

ステップ 2: IAM ポリシーとユーザーの作成

AWS では、セキュリティのベストプラクティスとして、許可を細分化して、各リソースへのアクセスを制御することが勧められます。AWS Identity and Access Management(IAM) を使用することで、AWS のユーザーとユーザー許可を管理できます。IAM ポリシーは、許可されるアクションとそのアクションが適用されるリソースを明示的にリストアップします。

一般的に、Kinesis Data Streams プロデューサーおよびコンシューマーには、次の最小許可が必要になります。

プロデューサー

アクション	リソース	目的
`DescribeStream` , `DescribeStreamSummary` , `DescribeStreamConsumer`	Kinesis Data Streams	レコードを読み取る前に、コンシューマーは、データスト、アクティブであること、シャードを含んでいることを確
`SubscribeToShard` , `RegisterStreamConsumer`	Kinesis Data Streams	コンシューマーをシャードにサブスクライブして登録しま
`PutRecord` , `PutRecords`	Kinesis Data Streams	Kinesis Data Streams にレコードを書き込みます。

コンシューマー

[Actions] (アクション)	[Resource] (リソース)	目的
`DescribeStream`	Kinesis Data Streams	レコードを読み取る前に、コンシューマーは、データスト、アクティブであること、シャードを含んでいることを確
`GetRecords` , `GetShardIterator`	Kinesis Data Streams	シャードからレコードを読み取ります。
`CreateTable` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB テーブル	Kinesis クライアントライブラリ (KCL) (バージョン 1.x ま ンシューマーが開発されている場合は、アプリケーション ときに DynamoDB テーブルにアクセスする必要があります
`DeleteItem`	Amazon DynamoDB テーブル	コンシューマーが Kinesis Data Streams シャードで分割と を実行する場合。
`PutMetricData`	Amazon CloudWatch Logs	また、KCL は、アプリケーションをモニタリングするのに CloudWatch にアップロードします。

このチュートリアルでは、上記のすべての許可を付与する 1 つの IAM ポリシーを作成します。本番環境では、プロデューサー用とコンシューマー用の 2 つのポリシーを作成してもかまいません。

IAM ポリシーを作成するには

1. 上記の手順で作成した新しいデータストリームの Amazon リソースネーム (ARN) を見つけます。この ARN は、 [ストリーム ARN] として [詳細]タブの上部に表示されます。ARN 形式 は次のとおりです。

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   AWS リージョンコード (`us-west-2` など)。詳細については、リージョンとアベイラビリティーゾーンの概念を参照してください。

アカウント

AWS アカウント ID (アカウント設定を参照してください)。

名前

上記のステップで作成したデータストリームの名前 (StockTradeStream)。

2. コンシューマーによって使用される (および最初のコンシューマーインスタンスによって作成される) DynamoDB テーブルの ARN を決定します。次のような形式になります。

```
arn:aws:dynamodb:region:account:table/name
```

リージョンとアカウント ID は、このチュートリアルで使用しているデータストリームの ARN の値と同じですが、name は、コンシューマーアプリケーションによって作成および使用される DynamoDB テーブルの名前です。KCL では、アプリケーション名をテーブル名として使用します。このステップでは、DynamoDB テーブル名に StockTradesProcessor を使用します。これは、このチュートリアルの後の手順で使用するアプリケーション名であるためです。

3. IAM コンソールのポリシー (https://console.aws.amazon.com/iam/home#policies) で、[ポリシーの作成] を選択します。IAM ポリシーを初めて扱う場合には、[今すぐ始める]、[ポリシーの作成] を選択します。
4. [ポリシージェネレーター] の横の [選択] を選択します。
5. AWS のサービスとして [Amazon Kinesis]を選択します。
6. 許可されるアクションとして、DescribeStream、GetShardIterator、GetRecords、PutRecord、および PutRecords を選択します。
7. このチュートリアルで使用するデータストリームの ARN を入力します。
8. 以下の各項目について、[ステートメントを追加] を使用します。

AWS のサービス	アクション	ARN
Amazon DynamoDB	CreateTable , DeleteItem , DescribeTable , GetItem, PutItem, Scan, UpdateItem	この手順のステップ 2 で作成した DynamoDB テーブルの ARN。

AWS のサービス	アクション	ARN
Amazon CloudWatch	PutMetricData	*

ARN を指定するときに使用されるアスタリスク (*) は必要ありません。PutMetricData アクションが呼び出される特定のリソースが CloudWatch に存在しない場合などがこれに該当します。

9. [Next Step (次のステップ)] をクリックします。
10. [ポリシー名] を StockTradeStreamPolicy に変更し、コードを確認して、[ポリシーの作成] を選択します。

最終的なポリシードキュメントは以下のようになります。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
      "Action": [
        "kinesis:DescribeStream",
        "kinesis:PutRecord",
        "kinesis:PutRecords",
        "kinesis:GetShardIterator",
        "kinesis:GetRecords",
        "kinesis:ListShards",
        "kinesis:DescribeStreamSummary",
        "kinesis:RegisterStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
      ]
    },
    {
      "Sid": "Stmt234",
      "Effect": "Allow",
      "Action": [
        "kinesis:SubscribeToShard",
        "kinesis:DescribeStreamConsumer"
      ],
```

```
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
      ]
    },
    {
      "Sid": "Stmt456",
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
      ]
    },
    {
      "Sid": "Stmt789",
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

IAM ユーザーを作成するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [Users] (ユーザー) ページで、[Add user] (ユーザーを追加) を選択します。
3. [User name] に、StockTradeStreamUser と入力します。
4. [アクセスの種類] で、[プログラムによるアクセス] を選択し、[次の手順: アクセス許可] を選択します。
5. [既存のポリシーを直接アタッチする] を選択します。
6. 上記の手順で作成したポリシーを名前 (StockTradeStreamPolicy) で検索します。ポリシー名の左にあるボックスを選択し、[次の手順: 確認] を選択します。
7. 詳細と概要を確認し、[ユーザーの作成] を選択します。
8. [アクセスキー ID] をコピーし、プライベート用に保存します。[シークレットアクセスキー] で [表示] を選択し、このキーもプライベートに保存します。

9. アクセスキーとシークレットキーを自分しかアクセスできない安全な場所にあるローカルファイルに貼り付けます。このアプリケーションでは、アクセス権限を厳しく制限した `~/.aws/credentials` という名前のファイルを作成します。ファイル形式は次のようになります。

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

IAM ポリシーをユーザーにアタッチするには

1. IAM コンソールで、[ポリシー] を開いて [ポリシーアクション] を選択します。
2. [StockTradeStreamPolicy] および [アタッチ] を選択します。
3. [StockTradeStreamUser] および [ポリシーのアタッチ] を選択します。

次のステップ

ステップ 3: コードをダウンロードしてビルドする

ステップ 3: コードをダウンロードしてビルドする

このトピックでは、データストリームへのサンプル株式取引の取り込み (プロデューサー) とこのデータの処理 (コンシューマー) のサンプル実装コードを示します。

コードをダウンロードしてビルドするには

1. https://github.com/aws-samples/amazon-kinesis-learning GitHub リポジトリからコンピュータにソースコードをダウンロードします。
2. 提供されたディレクトリ構造に従って、IDE でソースコードを使用してプロジェクトを作成します。
3. プロジェクトに次のライブラリを追加します。

   - Amazon Kinesis クライアントライブラリ (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient
   - Apache Commons Lang

- Apache Commons Logging
- Guava (Java 用の Google コアライブラリ)
- Jackson Annotations
- Jackson Core
- Jackson Databind
- Jackson Dataformat: CBOR
- Joda Time

4. IDE によっては、プロジェクトが自動的にビルドされる場合があります。自動的にビルドされない場合は、IDE に適切なステップを使用してプロジェクトをビルドします。

上記のステップが正常に完了したら、次のセクション (the section called “ステップ 4: プロデューサーを実装する”) に進みます。

次のステップ

ステップ 4: プロデューサーを実装する

このチュートリアルでは、株式市場取引をモニタリングする実際のシナリオを使用しています。以下の原理によって、このシナリオをプロデューサーおよびサポートコード構造にマッピングできます。

ソースコードを参照し、以下の情報を確認します。

StockTrade クラス

株式取引は、StockTrade クラスのインスタンスによって個別に表されます。このインスタンスには、ティッカーシンボル、株価、株数、取引のタイプ (買いまたは売り)、取引を一意に識別する ID などの属性が含まれます。このクラスは、既に実装されています。

ストリームレコード

ストリームとは、一連のレコードのことです。レコードとは、JSON 形式による連続する `StockTrade` インスタンスの 1 つを表しています。例:

```
{
  "tickerSymbol": "AMZN",
```

```
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator クラス

StockTradeGenerator には、呼び出されるたびにランダムに生成された新しい株式取引を返す、getRandomTrade() という名前のメソッドが含まれています。このクラスは、既に実装されています。

StockTradesWriter クラス

プロデューサーの main メソッドである StockTradesWriter は、継続的にランダム取引を取得し、以下のタスクを実行してその取引を Kinesis Data Streams に送信します。

1. ストリーム名とリージョン名を入力として読み取ります。
2. KinesisAsyncClientBuilder を使用してリージョン、認証情報、クライアント構成を設定します。
3. ストリームが存在し、アクティブであることを確認します (そうでない場合は、エラーで終了します)。
4. 連続ループで、StockTradeGenerator.getRandomTrade() メソッドに続き sendStockTrade メソッドを呼び出して、100 ミリ秒ごとに取引をストリームに送信します。

sendStockTrade クラスの StockTradesWriter メソッドには次のコードがあります。

```
private static void sendStockTrade(StockTrade trade, KinesisAsyncClient
 kinesisClient,
            String streamName) {
        byte[] bytes = trade.toJsonAsBytes();
        // The bytes could be null if there is an issue with the JSON serialization
 by the Jackson JSON library.
        if (bytes == null) {
            LOG.warn("Could not get JSON bytes for stock trade");
            return;
        }

        LOG.info("Putting trade: " + trade.toString());
```

```
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol
as the partition key, explained in the Supplemental Information section below.
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(bytes))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }
```

次のコードの詳細を参照してください。

- PutRecord API はバイト配列を想定するため、その取引を JSON 形式に変換する必要があります。この操作は、次の 1 行のコードによって行われます。

```
byte[] bytes = trade.toJsonAsBytes();
```

- 取引を送信する前に、新しい PutRecordRequest インスタンス (この場合は request) を作成する必要があります。各 request には、ストリーム名、パーティションキー、データ BLOB が必要です。

```
PutPutRecordRequest request = PutRecordRequest.builder()
    .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol as the
 partition key, explained in the Supplemental Information section below.
    .streamName(streamName)
    .data(SdkBytes.fromByteArray(bytes))
    .build();
```

この例では、株式チケットをパーティションキーとして使用することで、レコードを特定のシャードにマッピングしています。実際には、レコードがストリーム全体に均等に分散するように、シャード 1 つあたりに数百個または数千個のパーティションキーを用意する必要があり

ます。ストリームにデータを追加する方法の詳細については、Amazon Kinesis Data Streams へのデータの書き込みを参照してください。

次に、request をクライアントに送信できます (put オペレーション)。

```
   kinesisClient.putRecord(request).get();
```

- エラーチェックとログ記録は、いつでも追加して損はありません。次のコードによって、エラー状態を記録します。

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

put オペレーションの前後に try/catch ブロックを追加します。

```
try {
  kinesisClient.putRecord(request).get();
} catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
} catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again
 next cycle.", e);
}
```

これは、ネットワークエラーや、ストリームがスループット制限を超えて抑制されたことが原因で、Kinesis Data Streams の put オペレーションが失敗することがあるためです。データが失われることがないように、単純な再試行として使用するなど、put オペレーションの再試行ポリシーを慎重に検討することをお勧めします。

- ステータスのログ記録は有益ですが、オプションです。

```
LOG.info("Putting trade: " + trade.toString());
```

ここに示されているプロデューサーでは、Kinesis Data Streams API のシングルレコード機能 PutRecord が使用されています。実際には、個々のプロデューサーで大量のレコードが生成される場合があります。その場合、PutRecords のマルチレコード機能を使用して、レコードのバッチを一度に送信する方が効率的です。詳細については、Amazon Kinesis Data Streams へのデータの書き込みを参照してください。

プロデューサーを実行するには

1. ステップ 2: IAM ポリシーとユーザーの作成 で取得したアクセスキーとシークレットキーのペアがファイル ~/.aws/credentials に保存されていることを確認します。
2. 次の引数を指定して StockTradeWriter クラスを実行します。

```
StockTradeStream us-west-2
```

us-west-2 以外のリージョンにストリームを作成した場合は、代わりにそのリージョンをここで指定する必要があります。

次のような出力が表示されます。

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Kinesis Data Streams によって株式取引が取り込まれます。

次のステップ

ステップ 5: コンシューマーを実装する

ステップ 5: コンシューマーを実装する

このチュートリアルのコンシューマーアプリケーションは、データストリームの株式取引を継続的に処理します。その後、1 分ごとに売買されている最も人気のある株式を出力します。このアプリケーションは、Kinesis Client Library (KCL) 上に構築されており、コンシューマーアプリケーションに共通する面倒な作業の多くを行います。詳細については、Kinesis Client Library の使用を参照してください。

ソースコードを参照し、次の情報を確認してください。

StockTradesProcessor クラス

事前に用意されているコンシューマーのメインクラスで、次のタスクを実行します。

- 引数として渡されたアプリケーション、データストリーム、リージョン名を読み取ります。
- リージョン名で KinesisAsyncClient インスタンスを作成します。
- ShardRecordProcessor のインスタンスとして機能し、StockTradeRecordProcessor インスタンスによって実装される、StockTradeRecordProcessorFactory インスタンスを作成します。
- KinesisAsyncClient、StreamName、ApplicationName、および StockTradeRecordProcessorFactory インスタンスを使用して ConfigsBuilder インスタンスを作成します。これは、デフォルト値ですべての設定を作成するのに役立ちます。
- ConfigsBuilder インスタンスを使用して KCL スケジューラー (以前は KCL バージョン 1.x では KCL ワーカー) を作成します。
- このスケジューラーは、(このコンシューマーインスタンスに割り当てられた) 各シャードに新しいスレッドを作成します。これにより、継続的にデータストリームからレコードが読み取られます。次に、StockTradeRecordProcessor インスタンスを呼び出して、受信したレコードのバッチを処理します。

StockTradeRecordProcessor クラス

StockTradeRecordProcessor インスタンスを実装したら、次は initialize、processRecords、leaseLost、shardEnded、shutdownRequested の 5 つの必須メソッドを実装します。

KCL は initialize および shutdownRequested メソッドを使用して、レコードの受信を開始できるタイミングと、レコードの受信を停止するタイミングをそれぞれレコードプロセッサに通知し、アプリケーション固有の設定および終了タスクを実行できるようにします。leaseLostおよび shardEnded は、リースが失われたとき、または処理がシャードの終わりに達したときの動作ロジックを実装するために使用します。この例では、これらのイベントを示すメッセージをログに記録するだけです。

これらのメソッドのコードを示しています。主な処理は processRecords メソッドで行われ、そこでは各レコードの processRecord が使用されます。後者のメソッドは、ほとんどの場合、空のスケルトンコードとして提供されます。次のステップでは、これを実装する方法について説明します。詳細については、次のステップを参照してください。

また、processRecord のサポートメソッドである reportStats および resetStats の実装にも注目してください。これらのメソッドは、元のソースコードでは空になっています。

processRecords メソッドは既に実装されており、次のステップを実行します。

- 渡されたレコードごとに processRecord を呼び出します。
- 最後のレポートから 1 分間以上経過した場合は、reportStats() を呼び出して最新の統計を出力し、次の間隔に新しいレコードのみ含まれるように resetStats() を呼び出して統計を消去します。
- 次のレポート時間を設定します。
- 最後のチェックポイントから 1 分間以上経過した場合は、checkpoint() を呼び出します。
- 次のチェックポイント時間を設定します。

このメソッドでは、60 秒間間隔でレポートおよびチェックポイント時間が設定されています。チェックポイントの詳細については、Kinesis Client Library の使用を参照してください。

StockStats クラス

このクラスでは、データを保持し、最も人気のある株式の経時的な統計を示すことができます。このコードは、事前に用意されており、次のメソッドが含まれています。

- addStockTrade(StockTrade): 指定された StockTrade を実行中の統計に取り込みます。
- toString(): 特定の形式の文字列として統計を返します。

このクラスは、各株式の合計取引数と最大取引数を継続的にカウントすることで、最も人気のある株式を追跡します。これらの数は、株式取引を受け取る度に更新されます。

次のステップに示されているコードを StockTradeRecordProcessor クラスのメソッドに追加します。

コンシューマーを実装するには

1. processRecord メソッドを実装するには、サイズの正しい StockTrade オブジェクトを開始し、それにレコードデータを追加します。また、問題が発生した場合に警告がログに記録されるようにします。

```
byte[] arr = new byte[record.data().remaining()];
record.data().get(arr);
StockTrade trade = StockTrade.fromJsonAsBytes(arr);
    if (trade == null) {
        log.warn("Skipping record. Unable to parse record into StockTrade.
 Partition Key: " + record.partitionKey());
        return;
        }
stockStats.addStockTrade(trade);
```

2. 簡単な reportStats メソッドを実装します。出力形式は必要に応じて自由に変更できます。

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute
 ******\n" +
stockStats + "\n" +
"****************************************************************\n");
```

3. 新しい resetStats インスタンスを作成する stockStats メソッドを実装します。

```
stockStats = new StockStats();
```

4. ShardRecordProcessor インターフェイスに必要な以下のメソッドを実装します。

```
@Override
public void leaseLost(LeaseLostInput leaseLostInput) {
    log.info("Lost lease, so terminating.");
```

```
}

@Override
public void shardEnded(ShardEndedInput shardEndedInput) {
    try {
        log.info("Reached shard end checkpointing.");
        shardEndedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        log.error("Exception while checkpointing at shard end. Giving up.", e);
    }
}

@Override
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
    log.info("Scheduler is shutting down, checkpointing.");
    checkpoint(shutdownRequestedInput.checkpointer());
}

private void checkpoint(RecordProcessorCheckpointer checkpointer) {
    log.info("Checkpointing shard " + kinesisShardId);
    try {
        checkpointer.checkpoint();
    } catch (ShutdownException se) {
        // Ignore checkpoint if the processor instance has been shutdown (fail
 over).
        log.info("Caught shutdown exception, skipping checkpoint.", se);
    } catch (ThrottlingException e) {
        // Skip checkpoint when throttled. In practice, consider a backoff and
 retry policy.
        log.error("Caught throttling exception, skipping checkpoint.", e);
    } catch (InvalidStateException e) {
        // This indicates an issue with the DynamoDB table (check for table,
 provisioned IOPS).
        log.error("Cannot save checkpoint to the DynamoDB table used by the Amazon
 Kinesis Client Library.", e);
    }
}
```

コンシューマーを実行するには

1. で記述したプロデューサーを実行し、シミュレートした株式取引レコードをストリームに取り込みます。
2. 前のステップ (IAM ユーザーを作成したとき) で取得したアクセスキーとシークレットキーのペアがファイル ~/.aws/credentials に保存されていることを確認します。
3. 次の引数を指定して StockTradesProcessor クラスを実行します。

   ```
   StockTradesProcessor StockTradeStream us-west-2
   ```

   us-west-2 以外のリージョンにストリームを作成した場合は、代わりにそのリージョンをここで指定する必要があります。

1 分後、次のような出力が表示されます。その後、1 分間ごとに出力が更新されます。

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
****************************************************************
```

次のステップ

ステップ 6: (オプション) コンシューマーを拡張する

ステップ 6: (オプション) コンシューマーを拡張する

このオプションのセクションでは、さらに複雑なシナリオにも対応できるようにコンシューマーコードを拡張する方法について説明します。

1 分ごとに最大の売り注文を知るには、3 箇所の StockStats クラスを変更し、新しい優先順位を組み込みます。

コンシューマーを拡張するには

1. 新しいインスタンス変数を追加します。

   ```
   // Ticker symbol of the stock that had the largest quantity of shares sold
   ```

```
 private String largestSellOrderStock;
 // Quantity of shares for the largest sell order trade
 private long largestSellOrderQuantity;
```

2. 次のコードを addStockTrade に追加します。

```
if (type == TradeType.SELL) {
     if (largestSellOrderStock == null || trade.getQuantity() >
 largestSellOrderQuantity) {
         largestSellOrderStock = trade.getTickerSymbol();
         largestSellOrderQuantity = trade.getQuantity();
     }
 }
```

3. toString メソッドを変更し、追加情報を出力します。

```
public String toString() {
    return String.format(
        "Most popular stock being bought: %s, %d buys.%n" +
        "Most popular stock being sold: %s, %d sells.%n" +
        "Largest sell order: %d shares of %s.",
        getMostPopularStock(TradeType.BUY),
 getMostPopularStockCount(TradeType.BUY),
        getMostPopularStock(TradeType.SELL),
 getMostPopularStockCount(TradeType.SELL),
        largestSellOrderQuantity, largestSellOrderStock);
}
```

コンシューマーを今すぐ実行すると (プロデューサーも忘れずに実行してください)、次のような出力が表示されます。

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
Largest sell order: 996 shares of BUD.
************************************************************
```

次のステップ

ステップ 7: 終了する

ステップ 7: 終了する

Kinesis Data Streams の使用には料金がかかるため、作業が終わったら、ストリームおよび対応する Amazon DynamoDB テーブルは必ず削除してください。レコードを送信したり取得したりしていなくても、ストリームがアクティブなだけでわずかな料金が発生します。その理由として、アクティブなストリームでは、受信レコードを継続的に "リッスン" し、レコードを取得するようにリクエストすることにリソースが使用されるためです。

ストリームおよびテーブルを削除するには

1. 実行しているプロデューサーおよびコンシューマーをすべてシャットダウンします。
2. Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
3. このアプリケーション用に作成したストリーム (`StockTradeStream`) を選択します 。
4. [ストリームの削除] を選択します。
5. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
6. `StockTradesProcessor` テーブルを削除します。

まとめ

ほぼリアルタイムで大量のデータを処理するために、魔法のコードを記述したり、大規模なインフラストラクチャを開発したりする必要はありません。Kinesis Data Streams を使用すれば、少量のデータを処理するロジックを記述する (`processRecord(Record)` を記述するなど) 場合と同じように簡単にスケールして、大量のストリーミングデータに対応できます。Kinesis Data Streams が代わりに処理してくれるため、処理を拡張する方法を心配しなくて済みます。することと言えば、ストリームレコードを Kinesis Data Streams に送信し、受信した新しい各レコードを処理するロジックを記述するだけです。

このアプリケーションについて考えられる拡張機能は、次のとおりです。

すべてのシャードで集計する

現在は、単一のワーカーが単一のシャードから受け取ったデータレコードの集約に基づく統計が取得されます (複数のワーカーが同時に単一のアプリケーションからシャードを処理することはできません)。拡張するときに複数のシャードがある場合、すべてのシャードで集計しようと考えるかもしれません。そのためには、パイプラインアーキテクチャを用意します。パイプラインアーキテクチャでは、各ワーカーの出力が単一のシャードを持つ別のストリームに供給され、第 1 段階の出力を集計するワーカーによってそのストリームが処理されます。第 1 段階のデータが制限されるため (シャードごとに 1 分間あたり 1 つのサンプル)、シャードごとに処理しやすくなります。

処理の拡張

多数のシャードが含まれるようにストリームを拡張する場合 (多数のプロデューサーがデータを送信している場合)、処理を拡張するには、より多くのワーカーを追加します。複数のワーカーを Amazon EC2 インスタンスで実行し、Auto Scaling グループを使用することができます。

Amazon S3/DynamoDB/Amazon Redshift/Storm へのコネクタを使用する

ストリームは継続的に処理されるため、出力を他の保存先に送信することができます。AWS は、Kinesis DataStreams を他の AWS サービスやサードパーティツールと統合するためのコネクタを提供します。

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理

このチュートリアルのシナリオでは、株式取引をデータストリームに取り込み、ストリーム上で計算を実行するシンプルな Amazon Kinesis Data Streams アプリケーションを記述する必要があります。レコードのストリームを Kinesis Data Streams に送信し、ほぼリアルタイムでレコードを消費および処理するアプリケーションを実装する方法を説明します。

Important

Kinesis Data Streams は AWS の無料利用枠の対象外であるため、ストリームの作成後は、Kinesis Data Streams の使用に対してアカウントに少額の料金が発生します。コンシューマーアプリケーションが起動すると、Amazon DynamoDB の使用に伴う料金がわずかに発生します。コンシューマーアプリケーションでは、処理状態を追跡する際に DynamoDB

を使用します。このアプリケーションを終了したら、AWS リソースを削除して料金が発生しないようにしてください。詳細については、ステップ 7: 終了するを参照してください。

このコードでは、実際の株式市場データにアクセスする代わりに、株式取引のストリームをシミュレートします。シミュレーションには、2015 年 2 月時点における時価総額上位 25 社の株式に関する実際の市場データを基にしたランダム株式取引ジェネレーターが使用されています。リアルタイムの株式取引のストリームにアクセスできたとしたら、そのときに必要としている有益な統計を入手したいと考えるかもしれません。たとえば、スライディングウィンドウ分析を実行して、過去 5 分間に購入された最も人気のある株式を調べたいと思われるかもしれません。または、大規模な売り注文 (膨大な株式が含まれる売り注文) が発生したときに通知を受けたいと思われるかもしれません。このシリーズのコードを拡張して、このような機能を使用することもできます。

このチュートリアルにある手順をデスクトップやノートパソコンで実行し、同じマシンまたは定義された要件を満たす任意のプラットフォーム (例: Amazon Elastic Compute Cloud (Amazon EC2)) で、プロデューサーおよびコンシューマーのコードのいずれも実行できます。

この例では、米国西部 (オレゴン) リージョンが使用されていますが、Kinesis Data Streams がサポートされる AWS リージョンであれば、いずれのリージョンでも動作します。

タスク

- 前提条件
- ステップ 1: データストリームの作成
- ステップ 2: IAM ポリシーとユーザーの作成
- ステップ 3: 実装コードのダウンロードおよびビルド
- ステップ 4: プロデューサーを実装する
- ステップ 5: コンシューマーを実装する
- ステップ 6: (オプション) コンシューマーを拡張する
- ステップ 7: 終了する

前提条件

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理 を作成するための要件を以下に示します。

Amazon Web Services アカウント

開始する前に、Amazon Kinesis Data Streams の用語と概念で説明されている概念、特にストリーム、シャード、プロデューサー、コンシューマーについて理解しておきます。また、AWS CLI のインストールと設定を完了していると役立ちます。

AWS Management Console にアクセスするときに、AWS アカウントとウェブブラウザが必要になります。

コンソールにアクセスするには、IAM ユーザー名とパスワードを使用して、IAM サインインページからAWS Management Consoleにサインインします。プログラム的なアクセスや、長期的な認証情報に代わる手段を含めた AWS セキュリティ認証情報については、「IAM ユーザーガイド」の「AWS セキュリティ認証情報」を参照してください。AWS アカウントへのサインインに関する詳細については、「AWS サインイン User Guide」の「How to sign in to AWS」を参照してください。

IAM とセキュリティキーの設定手順の詳細については、IAM ユーザーを作成するを参照してください。

システムソフトウェア要件

アプリケーションを実行するシステムには、Java 7 以上がインストールされている必要があります。最新の Java Development Kit (JDK) をダウンロードおよびインストールするには、Oracle 社の Java SE インストールサイトを参照してください。

Eclipse などの Java IDE をお持ちの場合は、ソースコードを開いて編集、ビルド、および実行できます。

最新バージョンの AWS SDK for Java が必要です。Eclipse を IDE として使用している場合は、AWS Toolkit for Eclipse を代わりにインストールできます。

コンシューマーアプリケーションには、バージョン 1.2.1 以上の Kinesis クライアントライブラリ (KCL) が必要です。これは、Kinesis クライアントライブラリ (Java) の GitHub から入手することができます。

次のステップ

ステップ 1: データストリームの作成

ステップ 1: データストリームの作成

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理 の最初のステップで、後のステップで使用するストリームを作成します。

ストリームを作成するには

1. AWS Management Consoleにサインインして、Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. ナビゲーションペインで、[データストリーム] を選択します。
3. ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
4. [Kinesis ストリームの作成] を選択します。
5. ストリームの名前 (例: **StockTradeStream**) を入力します。
6. シャードカウントは**1**と入力しますが、[必要なシャードカウントの予想] は折りたたんだままにします。
7. [Kinesis ストリームの作成] を選択します。

[Kinesis streams] リストのページで、作成中のストリームのステータスは CREATING になります。ストリームを使用する準備ができると、ステータスは ACTIVE に変わります。ストリームの名前を選択します。表示されたページの [詳細] タブには、ストリーム設定の概要が示されます。[モニタリング] セクションには、ストリームのモニタリング情報が表示されます。

シャードに関する追加情報

このチュートリアルを除き、初めて Kinesis Data Streams を使用する場合は、より慎重にストリーム作成プロセスを計画する必要がある場合があります。シャードをプロビジョニングするときには、予想される最大需要を考慮する必要があります。このシナリオを例として使用すると、米国の株式市場の取引トラフィックは、昼間 (東部標準時) にピークを迎えます。その時刻をサンプルとして需要の予測を行う必要があります。その後、予想される最大需要に合わせてプロビジョニングするか、需要の変動に応じてストリームを拡大または縮小することができます。

シャードは、スループット容量の単位です。[Kinesis ストリームの作成] ページで、[必要なシャードカウントの予想] を展開します。次のガイドラインに従って、平均レコードサイズ、1 秒間に書き込まれる最大レコード数、コンシューマーアプリケーションの数を入力します。

平均レコードサイズ

計算される平均レコードサイズの予測。この値がわからない場合は、予測される最大レコードサイズを使用します。

書き込まれる最大レコード数

データを提供するエンティティの数と各エンティティで 1 秒間に生成されるおよそのレコード数を考慮に入れます。たとえば、20 台の取引サーバーから株式取引データを取得し、各サーバーで

1 秒間に 250 個の取引が生成される場合、1 秒あたりの合計取引数 (レコード数) は 5,000 になります。

コンシューマーアプリケーションの数

独立してストリームを読み取り、ストリームを固有の方法で処理し、固有の出力を生成するアプリケーションの数。各アプリケーションでは、複数のインスタンスを異なるマシン (つまり、クラスター) で実行することができます。このため、大規模なストリームでも遅延することなく処理できます。

表示された予測シャードカウントが現在のシャード制限を超えた場合は、その数のシャードカウントを含むストリームを作成する前に、制限を引き上げるリクエストの送信が必要な場合があります。シャード制限の引き上げをリクエストするには、Kinesis Data Streams 制限フォームを使用します。ストリームおよびシャードの詳細については、ストリームの作成と管理を参照してください。

次のステップ

ステップ 2: IAM ポリシーとユーザーの作成

ステップ 2: IAM ポリシーとユーザーの作成

AWS では、セキュリティのベストプラクティスとして、許可を細分化して、各リソースへのアクセスを制御することが勧められます。AWS Identity and Access Management(IAM) を使用することで、AWS のユーザーとユーザー許可を管理できます。IAM ポリシーは、許可されるアクションとそのアクションが適用されるリソースを明示的にリストアップします。

一般的に、Kinesis Data Streams プロデューサーおよびコンシューマーには、次の最小許可が必要になります。

プロデューサー

アクション	リソース	目的
`DescribeStream` , `DescribeStreamSummary` , `DescribeStreamConsumer`	Kinesis Data Streams	レコードを書き込む前に、プロデューサーは、ストリーム であること、シャードがストリームに含まれていること、 ンシューマーがあることを確認します。

アクション	リソース	目的
`SubscribeToShard` , `RegisterStreamConsumer`	Kinesis Data Streams	Kinesis Data Stream シャードにサブスクライブし、コンシ。
`PutRecord` , `PutRecords`	Kinesis Data Streams	Amazon Kinesis ストリームにレコードを書き込みます。

コンシューマー

[Actions] (アクション)	[Resource] (リソース)	目的
`DescribeStream`	Kinesis Data Streams	レコードを読み取る前に、コンシューマーは、ストリーム であることを確認し、ストリームにシャードが含まれるこ
`GetRecords` , `GetShardIterator`	Kinesis Data Streams	Kinesis Data Streams シャードからレコードを読み込みま
`CreateTable` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB テーブル	Kinesis クライアントライブラリ (KCL) を使用してコンシ る場合は、アプリケーションの処理状態を追跡するときに の許可が必要です。テーブルは、最初に開始したコンシュ れます。
`DeleteItem`	Amazon DynamoDB テーブル	コンシューマーが Kinesis Data Streams シャードで分割と を実行する場合。
`PutMetricData`	Amazon CloudWatch Logs	また、KCL は、アプリケーションをモニタリングするのに CloudWatch にアップロードします。

このアプリケーションでは、前述のすべての許可を付与する単一の IAM ポリシーを作成します。実際には、プロデューサーとコンシューマーに 1 つずつ、2 つのポリシーを作成することになるかもしれません。

IAM ポリシーを作成するには

1. 新しいストリームの Amazon リソースネーム (ARN) を見つけます。この ARN は、 [ストリーム ARN] として [詳細]タブの上部に表示されます。ARN 形式 は次のとおりです。

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   リージョンコード (us-west-2 など)。詳細については、リージョンとアベイラビリティーゾーンの概念を参照してください。

   アカウント

   AWS アカウント ID (アカウント設定を参照してください)。

   名前

   ステップ 1: データストリームの作成 からのストリームの名前 (StockTradeStream)。

2. コンシューマーによって使用される (最初のコンシューマーインスタンスによって作成された) DynamoDB テーブルの ARN を決定します。次のような形式になります。

   ```
   arn:aws:dynamodb:region:account:table/name
   ```

   リージョンとアカウントは前のステップと同じ場所のものですが、この場合の名前はコンシューマーアプリケーションによって作成および使用されるテーブルの名前となります。コンシューマーによって使用される KCL では、アプリケーション名がテーブル名として使用されます。後で使用されるアプリケーション名である StockTradesProcessor を使用します。

3. IAM コンソールのポリシー (https://console.aws.amazon.com/iam/home#policies) で、[ポリシーの作成] を選択します。IAM ポリシーを初めて扱う場合には、[今すぐ始める]、[ポリシーの作成] を選択します。

4. [ポリシージェネレーター] の横の [選択] を選択します。

5. AWS のサービスとして [Amazon Kinesis]を選択します。

6. 許可されるアクションとして、DescribeStream、GetShardIterator、GetRecords、PutRecord、および PutRecords を選択します。

7. ステップ 1 で作成した ARN を入力します。

8. 以下の各項目について、[ステートメントを追加] を使用します。

AWS のサービス	アクション	ARN
Amazon DynamoDB	`CreateTable` , `DeleteItem` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	ステップ 2 で作成した ARN
Amazon CloudWatch	`PutMetricData`	*

ARN を指定するときに使用されるアスタリスク (*) は必要ありません。`PutMetricData` アクションが呼び出される特定のリソースが CloudWatch に存在しない場合などがこれに該当します。

9. [Next Step (次のステップ)] をクリックします。
10. [ポリシー名] を `StockTradeStreamPolicy` に変更し、コードを確認して、[ポリシーの作成] を選択します。

取得されたポリシードキュメントには、次のような結果が表示されます

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
      "Action": [
        "kinesis:DescribeStream",
        "kinesis:PutRecord",
        "kinesis:PutRecords",
        "kinesis:GetShardIterator",
        "kinesis:GetRecords",
        "kinesis:ListShards",
        "kinesis:DescribeStreamSummary",
        "kinesis:RegisterStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
```

```
      ]
    },
    {
      "Sid": "Stmt234",
      "Effect": "Allow",
      "Action": [
        "kinesis:SubscribeToShard",
        "kinesis:DescribeStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
      ]
    },
    {
      "Sid": "Stmt456",
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
      ]
    },
    {
      "Sid": "Stmt789",
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

IAM ユーザーを作成するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [Users] (ユーザー) ページで、[Add user] (ユーザーを追加) を選択します。
3. [User name] に、`StockTradeStreamUser` と入力します。

4. [アクセスの種類] で、[プログラムによるアクセス] を選択し、[次の手順: アクセス許可] を選択します。
5. [既存のポリシーを直接アタッチする] を選択します。
6. 作成したポリシーの名前で検索します。ポリシー名の左にあるボックスを選択し、[次の手順: 確認] を選択します。
7. 詳細と概要を確認し、[ユーザーの作成] を選択します。
8. [アクセスキー ID] をコピーし、プライベート用に保存します。[シークレットアクセスキー] で [表示] を選択し、このキーもプライベートに保存します。
9. アクセスキーとシークレットキーを自分しかアクセスできない安全な場所にあるローカルファイルに貼り付けます。このアプリケーションでは、アクセス権限を厳しく制限した `~/.aws/credentials` という名前のファイルを作成します。ファイル形式は次のようになります。

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

IAM ポリシーをユーザーにアタッチするには

1. IAM コンソールで、[ポリシー] を開いて [ポリシーアクション] を選択します。
2. [StockTradeStreamPolicy] および [アタッチ] を選択します。
3. [StockTradeStreamUser] および [ポリシーのアタッチ] を選択します。

次のステップ

ステップ 3: 実装コードのダウンロードおよびビルド

ステップ 3: 実装コードのダウンロードおよびビルド

スケルトンコードは the section called “チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理” 用に提供されています。このコードには、株式取引ストリームの取り込み (プロデューサー) およびデータの処理 (コンシューマー) のいずれにも使用できるスタブ実装が含まれています。次の手順は、実装を完了する方法を示しています。

実装コードをダウンロードおよびビルドするには

1. ソースコードをコンピュータにダウンロードします。

2. 提供されたディレクトリ構造に従って、お好みの IDE でソースコードを使用してプロジェクトを作成します。
3. プロジェクトに次のライブラリを追加します。
   - Amazon Kinesis クライアントライブラリ (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient
   - Apache Commons Lang
   - Apache Commons Logging
   - Guava (Java 用の Google コアライブラリ)
   - Jackson Annotations
   - Jackson Core
   - Jackson Databind
   - Jackson Dataformat: CBOR
   - Joda Time
4. IDE によっては、プロジェクトが自動的にビルドされる場合があります。自動的にビルドされない場合は、IDE に適切なステップを使用してプロジェクトをビルドします。

上記のステップが正常に完了したら、次のセクション (the section called “ステップ 4: プロデューサーを実装する”) に進みます。ビルドのいずれかの段階でエラーが発生した場合は、先に進む前に、原因を調査の上、解決してください。

次のステップ

ステップ 4: プロデューサーを実装する

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理 のアプリケーションでは、株式市場取引をモニタリングする実際のシナリオが使用されます。次の原理によって、このシナリオをプロデューサーおよびサポートコード構造にマッピングすることができます。

ソースコードを参照し、次の情報を確認してください。

StockTrade クラス

株式取引は、StockTrade クラスのインスタンスによって個別に表されます。このインスタンスには、ティッカーシンボル、株価、株数、取引のタイプ (買いまたは売り)、取引を一意に識別する ID などの属性が含まれます。このクラスは、既に実装されています。

ストリームレコード

ストリームとは、一連のレコードのことです。レコードとは、JSON 形式による連続する StockTrade インスタンスの 1 つを表しています。例:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator クラス

StockTradeGenerator には、呼び出されるたびにランダムに生成された新しい株式取引を返す、getRandomTrade() と呼ばれるメソッドが含まれています。このクラスは、既に実装されています。

StockTradesWriter クラス

プロデューサーの main メソッドである StockTradesWriter は、継続的にランダム取引を取得し、以下のタスクを実行してそれらを Kinesis Data Streams に送信します。

1. ストリーム名とリージョン名を入力として読み取ります。
2. AmazonKinesisClientBuilder を作成します。
3. クライアントビルダーを使用してリージョン、認証情報、およびクライアント構成を設定します。
4. クライアントビルダーを使用して AmazonKinesis クライアントを構成します。
5. ストリームが存在し、アクティブであることを確認します (そうでない場合は、エラーで終了します)。
6. 連続ループで、StockTradeGenerator.getRandomTrade() メソッドに続き sendStockTrade メソッドを呼び出して、100 ミリ秒ごとに取引をストリームに送信します。

sendStockTrade クラスの StockTradesWriter メソッドには次のコードがあります。

```
private static void sendStockTrade(StockTrade trade, AmazonKinesis kinesisClient,
 String streamName) {
    byte[] bytes = trade.toJsonAsBytes();
    // The bytes could be null if there is an issue with the JSON serialization by
 the Jackson JSON library.
    if (bytes == null) {
        LOG.warn("Could not get JSON bytes for stock trade");
        return;
    }

    LOG.info("Putting trade: " + trade.toString());
    PutRecordRequest putRecord = new PutRecordRequest();
    putRecord.setStreamName(streamName);
    // We use the ticker symbol as the partition key, explained in the Supplemental
 Information section below.
    putRecord.setPartitionKey(trade.getTickerSymbol());
    putRecord.setData(ByteBuffer.wrap(bytes));

    try {
        kinesisClient.putRecord(putRecord);
    } catch (AmazonClientException ex) {
        LOG.warn("Error sending record to Amazon Kinesis.", ex);
    }
}
```

次のコードの詳細を参照してください。

- PutRecord API はバイト配列を想定するため、trade を JSON 形式に変換する必要があります。この操作は、次の 1 行のコードによって行われます。

  ```
  byte[] bytes = trade.toJsonAsBytes();
  ```

- 取引を送信する前に、新しい PutRecordRequest インスタンス (この場合、putRecord と呼ばれる) を作成する必要があります。

  ```
  PutRecordRequest putRecord = new PutRecordRequest();
  ```

  各 PutRecord の呼び出しには、ストリーム名、パーティションキー、およびデータ BLOB が必要です。次のコードによって、putRecord メソッドを使用して、これらのフィールドを setXxxx() オブジェクトに追加します。

```
putRecord.setStreamName(streamName);
putRecord.setPartitionKey(trade.getTickerSymbol());
putRecord.setData(ByteBuffer.wrap(bytes));
```

この例では、株式チケットをパーティションキーとして使用することで、レコードを特定のシャードにマッピングしています。実際には、レコードがストリーム全体に均等に分散するように、シャード 1 つあたりに数百個または数千個のパーティションキーを用意する必要があります。ストリームにデータを追加する方法の詳細については、ストリームへのデータの追加を参照してください。

次に、putRecord をクライアントに送信 (put オペレーション) することができます。

```
kinesisClient.putRecord(putRecord);
```

- エラーチェックとログ記録は、いつでも追加して損はありません。次のコードによって、エラー状態を記録します。

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

put オペレーションの前後に try/catch ブロックを追加します。

```
try {
       kinesisClient.putRecord(putRecord);
} catch (AmazonClientException ex) {
       LOG.warn("Error sending record to Amazon Kinesis.", ex);
}
```

これは、ネットワークエラーや、ストリームがスループット制限を超えて抑制されたことが原因で、Kinesis Data Streams の put オペレーションが失敗することがあるためです。データが失われることがないように、単純な再試行として使用するなど、put オペレーションの再試行ポリシーを慎重に検討することをお勧めします。

- ステータスのログ記録は有益ですが、オプションです。

```
LOG.info("Putting trade: " + trade.toString());
```

ここに示されているプロデューサーでは、Kinesis Data Streams API のシングルレコード機能 PutRecord が使用されています。実際には、個々のプロデューサーで大量のレコードが生成される場合があります。その場合、PutRecords のマルチレコード機能を使用して、レコードのバッチを一度に送信する方が効率的です。詳細については、ストリームへのデータの追加を参照してください。

プロデューサーを実行するには

1. 前のステップ (IAM ユーザーを作成したとき) で取得したアクセスキーとシークレットキーのペアがファイル ~/.aws/credentials に保存されていることを確認します。
2. 次の引数を指定して StockTradeWriter クラスを実行します。

```
StockTradeStream us-west-2
```

us-west-2 以外のリージョンにストリームを作成した場合は、代わりにそのリージョンをここで指定する必要があります。

次のような出力が表示されます。

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Kinesis Data Streams によって株式取引ストリームが取り込まれます。

次のステップ

ステップ 5: コンシューマーを実装する

ステップ 5: コンシューマーを実装する

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理 のコンシューマーアプリケーションでは、で作成した株式取引ストリームを継続的に処理します。その後、1 分ごとに売買されている最も人気のある株式を出力します。このアプリケーションは、Kinesis Client Library (KCL) 上に構築されており、コンシューマーアプリケーションに共通する面倒な作業の多くを行います。詳細については、KCL 1.x コンシューマーの開発を参照してください。

ソースコードを参照し、次の情報を確認してください。

StockTradesProcessor クラス

事前に用意されているコンシューマーのメインクラスで、次のタスクを実行します。

- 引数として渡されたアプリケーション、ストリーム、およびリージョン名を読み取ります。
- `~/.aws/credentials` から認証情報を読み取ります。
- `RecordProcessor` のインスタンスとして機能し、`StockTradeRecordProcessor` インスタンスによって実装される、`RecordProcessorFactory` インスタンスを作成します。
- インスタンスおよび標準設定 (例: ストリーム名、認証情報、アプリケーション名) が指定されたKCL ワーカーを作成します。
- このワーカーは、(このコンシューマーインスタンスに割り当てられた) 各シャードに新しいスレッドを作成します。これにより、継続的に Kinesis Data Streams からレコードが読み取られます。次に、`RecordProcessor` インスタンスを呼び出して、受信したレコードのバッチを処理します。

StockTradeRecordProcessor クラス

`RecordProcessor` インスタンスを実装したら、次に `initialize`、`processRecords`、`shutdown` の 3 つの必須メソッドを実装します。

Kinesis Client Library によって使用される `initialize` および `shutdown` は、名前が示すとおり、レコードの受信がいつ開始し、いつ終了するかをレコードプロセッサに知らせます。これにより、レコードプロセッサは、アプリケーションに固有の設定および終了タスクを行うことができます。これらのコードは事前に用意されています。主な処理は `processRecords` メソッドで行われ、そこでは各レコードの `processRecord` が使用されます。後者のメソッドは、ほとんどの場合、空のスケルトンコードとして提供されます。次のステップでは、これを実装する方法について説明します。詳細は、次のステップを参照してください。

また、`processRecord` のサポートメソッドである `reportStats` および `resetStats` の実装にも注目してください。これらのメソッドは、元のソースコードでは空になっています。

processRecords メソッドは既に実装されており、次のステップを実行します。

- 渡された各レコードについて、レコード上で processRecord を呼び出します。
- 最後のレポートから 1 分間以上経過した場合は、reportStats() を呼び出して最新の統計を出力し、次の間隔に新しいレコードのみ含まれるように resetStats() を呼び出して統計を消去します。
- 次のレポート時間を設定します。
- 最後のチェックポイントから 1 分間以上経過した場合は、checkpoint() を呼び出します。
- 次のチェックポイント時間を設定します。

このメソッドでは、60 秒間間隔でレポートおよびチェックポイント時間が設定されています。チェックポイントの詳細については、コンシューマーに関する追加情報を参照してください。

StockStats クラス

このクラスでは、データを保持し、最も人気のある株式の経時的な統計を示すことができます。このコードは、事前に用意されており、次のメソッドが含まれています。

- addStockTrade(StockTrade): 指定された StockTrade を実行中の統計に取り込みます。
- toString(): 特定の形式の文字列として統計を返します。

このクラスは、各株式の合計取引数と最大取引数を継続的にカウントすることで、最も人気のある株式を追跡します。これらの数は、株式取引を受け取る度に更新されます。

次のステップに示されているコードを StockTradeRecordProcessor クラスのメソッドに追加します。

コンシューマーを実装するには

1. processRecord メソッドを実装するには、サイズの正しい StockTrade オブジェクトを開始し、それにレコードデータを追加します。また、問題が発生した場合に警告がログに記録されるようにします。

```
StockTrade trade = StockTrade.fromJsonAsBytes(record.getData().array());
if (trade == null) {
    LOG.warn("Skipping record. Unable to parse record into StockTrade. Partition
 Key: " + record.getPartitionKey());
    return;
}
stockStats.addStockTrade(trade);
```

2. 簡単な reportStats メソッドを実装します。出力形式は好みに応じて自由に変更することができます。

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute
 ******\n" +
                   stockStats + "\n" +

 "****************************************************************\n");
```

3. 最後に、新しい stockStats インスタンスを作成する resetStats メソッドを実装します。

```
stockStats = new StockStats();
```

コンシューマーを実行するには

1. で記述したプロデューサーを実行し、シミュレートした株式取引レコードをストリームに取り込みます。
2. 前のステップ (IAM ユーザーを作成したとき) で取得したアクセスキーとシークレットキーのペアがファイル ~/.aws/credentials に保存されていることを確認します。
3. 次の引数を指定して StockTradesProcessor クラスを実行します。

```
StockTradesProcessor StockTradeStream us-west-2
```

us-west-2 以外のリージョンにストリームを作成した場合は、代わりにそのリージョンをここで指定する必要があります。

1 分後、次のような出力が表示されます。その後、1 分間ごとに出力が更新されます。

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
****************************************************************
```

コンシューマーに関する追加情報

KCL 1.x コンシューマーの開発などで説明されている Kinesis Client Library のメリットに詳しい方であれば、ここで使用することに疑問を感じるかもしれません。1 つのシャードストリームとそれを

処理する 1 つのコンシューマーインスタンスしか使用しない場合でも、KCL を使用して簡単にコンシューマーを実装することができます。プロデューサーセクションとコンシューマーセクションのコードの実装手順を比較すると、コンシューマーの実装の方が比較的に簡単であることがわかります。これは、KCL で提供されているサービスが大きく関係しています。

このアプリケーションでは、個別のレコードを処理できるレコードプロセッサクラスの実装に焦点を合わせてきました。新しいレコードが使用可能になると、KCL がレコードを取得してレコードプロセッサを呼び出すため、Kinesis Data Streams からレコードを取得する方法を心配しなくて済みます。また、シャードカウントやコンシューマーインスタンス数についても心配しなくて済みます。ストリームがスケールアップされても、複数のシャードやコンシューマーインスタンスを処理するためにアプリケーションを書き直す必要はありません。

チェックポイントとは、ストリームにおける特定のポイントのことで、それまでに消費および処理されたデータレコードが記録されます。このため、アプリケーションがクラッシュしても、ストリームの始めからではなく、そのポイントからストリームが読み取られます。チェックポイントやそのさまざまな設計パターン、およびベストプラクティスは、この章の範囲外です。ただし、本番環境ではこのような問題に直面することがあります。

で学習したように、Kinesis Data Streams API の `put` オペレーションは、パーティションキーを入力として受け取ります。Kinesis Data Streams は、レコードを複数のシャードに分割するメカニズムとしてパーティションキーを使用します (複数のシャードがストリームに含まれる場合)。同じパーティションキーは、常に同じシャードにルーティングされます。このため、同じパーティションキーを持つレコードはそのコンシューマーにのみ送信され、他のコンシューマーに送信されることはないと仮定して、特定のシャードを処理するコンシューマーを設計できます。したがって、コンシューマーのワーカーは、必要なデータが欠落しているかもしれないと心配することなく、同じパーティションキーを持つすべてのレコードを集計できます。

このアプリケーションでは、コンシューマーによるレコードの処理の負荷は高くないため、1 つのシャードを使用して、KCL スレッドと同じスレッドで処理することができます。ただし、実際には、まずシャードの数のスケールアップを検討します。レコードの処理が大変になることが予想される場合は、異なるスレッドに処理を切り替えたり、スレッドプールを使用したりする必要があるかもしれません。このように、その他のスレッドがレコードを並列処理していても、KCL は新しいレコードを迅速に取得できます。一般的に、マルチスレッド設計は簡単ではなく高度な技術が必要になるため、シャードの数を増やすことが最も効果的で簡単な拡張方法です。

次のステップ

ステップ 6: (オプション) コンシューマーを拡張する

ステップ 6: (オプション) コンシューマーを拡張する

チュートリアル: KPL と KCL 1.x を使用した株式データのリアルタイム処理 のアプリケーションは、すでに目的を十分に果たしているかもしれません。このオプションのセクションでは、さらに複雑なシナリオにも対応できるようにコンシューマーコードを拡張する方法について説明します。

1 分ごとに最大の売り注文を知るには、3 箇所の StockStats クラスを変更し、新しい優先順位を組み込みます。

コンシューマーを拡張するには

1. 新しいインスタンス変数を追加します。

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. 次のコードを addStockTrade に追加します。

```
if (type == TradeType.SELL) {
    if (largestSellOrderStock == null || trade.getQuantity() >
largestSellOrderQuantity) {
        largestSellOrderStock = trade.getTickerSymbol();
        largestSellOrderQuantity = trade.getQuantity();
    }
}
```

3. toString メソッドを変更し、追加情報を出力します。

```
public String toString() {
    return String.format(
            "Most popular stock being bought: %s, %d buys.%n" +
            "Most popular stock being sold: %s, %d sells.%n" +
            "Largest sell order: %d shares of %s.",
            getMostPopularStock(TradeType.BUY),
getMostPopularStockCount(TradeType.BUY),
            getMostPopularStock(TradeType.SELL),
getMostPopularStockCount(TradeType.SELL),
            largestSellOrderQuantity, largestSellOrderStock);
}
```

コンシューマーを今すぐ実行すると (プロデューサーも忘れずに実行してください)、次のような出力が表示されます。

```
****** Shard shardId-000000000001 stats for last 1 minute ******
 Most popular stock being bought: WMT, 27 buys.
 Most popular stock being sold: PTR, 14 sells.
 Largest sell order: 996 shares of BUD.
 ************************************************************
```

次のステップ

ステップ 7: 終了する

ステップ 7: 終了する

Kinesis Data Streams の使用には料金がかかるため、作業が終わったら、ストリームおよび対応する Amazon DynamoDB テーブルは必ず削除してください。レコードを送信したり取得したりしていなくても、ストリームがアクティブなだけでわずかな料金が発生します。その理由として、アクティブなストリームでは、受信レコードを継続的に "リッスン" し、レコードを取得するようにリクエストすることにリソースが使用されるためです。

ストリームおよびテーブルを削除するには

1. 実行しているプロデューサーおよびコンシューマーをすべてシャットダウンします。
2. Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
3. このアプリケーション用に作成したストリーム (`StockTradeStream`) を選択します 。
4. [ストリームの削除] を選択します。
5. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
6. `StockTradesProcessor` テーブルを削除します。

まとめ

ほぼリアルタイムで大量のデータを処理するために、魔法のコードを記述したり、大規模なインフラストラクチャを開発したりする必要はありません。Kinesis Data Streams を使用すれば、少量のデータを処理するロジックを記述する (`processRecord(Record)` を記述するなど) 場合と同じように簡単にスケールして、大量のストリーミングデータに対応できます。Kinesis Data Streams が代わりに処理してくれるため、処理を拡張する方法を心配しなくて済みます。することと言えば、ストリー

ムレコードを Kinesis Data Streams に送信し、受信した新しい各レコードを処理するロジックを記述するだけです。

このアプリケーションについて考えられる拡張機能は、次のとおりです。

すべてのシャードで集計する

現在は、単一のワーカーが単一のシャードから受け取ったデータレコードの集約に基づく統計が取得されます (複数のワーカーが同時に単一のアプリケーションからシャードを処理することはできません)。拡張するときに複数のシャードがある場合、すべてのシャードで集計しようと考えるかもしれません。そのためには、パイプラインアーキテクチャを用意します。パイプラインアーキテクチャでは、各ワーカーの出力が単一のシャードを持つ別のストリームに供給され、第 1 段階の出力を集計するワーカーによってそのストリームが処理されます。第 1 段階のデータが制限されるため (シャードごとに 1 分間あたり 1 つのサンプル)、シャードごとに処理しやすくなります。

処理の拡張

多数のシャードが含まれるようにストリームを拡張する場合 (多数のプロデューサーがデータを送信している場合)、処理を拡張するには、より多くのワーカーを追加します。複数のワーカーを Amazon EC2 インスタンスで実行し、Auto Scaling グループを使用することができます。

Amazon S3/DynamoDB/Amazon Redshift/Storm へのコネクタを使用する

ストリームは継続的に処理されるため、出力を他の保存先に送信することができます。AWS は、Kinesis DataStreams を他の AWS サービスやサードパーティツールと統合するためのコネクタを提供します。

次のステップ

- Kinesis Data Streams API オペレーションの使用の詳細については、Amazon Kinesis Data Streams API と AWS SDK for Java を使用したプロデューサーの開発、AWS SDK for Java を使用した共有スループットでのカスタムコンシューマーの開発、および ストリームの作成と管理 を参照してください。
- Kinesis Client Library の詳細については、「KCL 1.x コンシューマーの開発」を参照してください。
- アプリケーションを最適化する方法については、高度なトピックを参照してください。

チュートリアル: Flink アプリケーション向けの Managed Service for Apache Flink を使用してリアルタイムの株式データを分析する

このチュートリアルのシナリオには、株式取引のデータストリームへの取り込みと、ストリームで計算を実行するシンプルな Amazon Managed Service for Apache Flink アプリケーションの記述が含まれます。レコードのストリームを Kinesis Data Streams に送信し、ほぼリアルタイムでレコードを消費および処理するアプリケーションを実装する方法を説明します。

Flink アプリケーション向けの Managed Service for Apache Flink では、Java または Scala を使用してストリーミングデータを処理し、分析することができます。このサービスを使用すると、ストリーミングソースに対して Java または Scala コードを作成して実行し、時系列分析の実行、ダッシュボードへのリアルタイムフィード、メトリクスのリアルタイム作成を行うことができます。

Flink アプリケーションは、Apache Flink に基づくオープンソースライブラリを使用して、Managed Service for Apache Flink で構築できます。Apache Flink は、データストリームを処理するための一般的なフレームワークおよびエンジンです。

> **Important**
>
> 2 つのデータストリームと 1 つのアプリケーションを作成すると、無料利用枠の対象にならないため、アカウントには Kinesis Data Streams と Managed Service for Apache Flink の使用に対してわずかな料金が発生します。 AWS このアプリケーションを使い終わったら、リソースを削除して料金が発生しないようにしてください。 AWS

このコードでは、実際の株式市場データにアクセスする代わりに、株式取引のストリームをシミュレートします。そのために、ランダム株式取引ジェネレーターが使用されます。リアルタイムの株式取引のストリームにアクセスできたとしたら、そのときに必要としている有益な統計を入手したいと考えるかもしれません。たとえば、スライディングウィンドウ分析を実行して、過去 5 分間に購入された最も人気のある株式を調べたいと思われるかもしれません。または、大規模な売り注文 (膨大な株式が含まれる売り注文) が発生したときに通知を受けたいと思われるかもしれません。このシリーズのコードを拡張して、このような機能を使用することもできます。

この例では米国西部 (オレゴン) リージョンが使用されていますが、Managed Service for Apache Flink がサポートされるAWS リージョンであれば、どのリージョンでも動作します。

タスク

- 演習を完了するための前提条件

- ステップ 1: AWS アカウントを設定して管理者ユーザーを作成する
- ステップ 2: AWS Command Line Interface ()AWS CLIをセットアップする
- ステップ 3: Flink アプリケーション向けの Managed Service for Apache Flink を作成して実行する

演習を完了するための前提条件

このガイドの手順を完了するには、以下が必要です。

- Java 開発キット (JDK) バージョン 8。JAVA_HOME 環境変数を、JDK のインストール場所を指すように設定します。
- 開発環境 (Eclipse Java Neon や IntelliJ Idea など) を使用してアプリケーションを開発し、コンパイルすることをお勧めします。
- Git クライアント。Git クライアントをまだインストールしていない場合は、インストールします。
- Apache Maven Compiler Plugin。Maven が作業パスに含まれている必要があります。Apache Maven のインストールをテストするには、次のように入力します。

```
$ mvn -version
```

開始するには、ステップ 1: AWS アカウントを設定して管理者ユーザーを作成するに進みます。

ステップ 1: AWS アカウントを設定して管理者ユーザーを作成する

Flink アプリケーション向けの Amazon Managed Service for Apache Flink を初めて使用する場合は、その前に以下のタスクを実行してください。

1. にサインアップ AWS
2. IAM ユーザーの作成

にサインアップ AWS

Amazon Web Services (AWS) にサインアップすると、 AWS アカウントは Apache Flink 用 Amazon マネージドサービスを含む AWS、のすべてのサービスに自動的にサインアップされます。料金は、使用するサービスの料金のみが請求されます。

Managed Service for Apache Flink では、使用したリソースの料金のみを支払います。 AWS の新規のお客様の場合は、Managed Service for Apache Flink の使用を無料で開始できます。詳細については、AWS 無料利用枠を参照してください。

AWS 既にアカウントをお持ちの場合は、次のタスクに進んでください。 AWS アカウントをお持ちでない場合は、次のステップに従って作成します。

AWS アカウントを作成するには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   にサインアップすると AWS アカウント、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティ上のベストプラクティスとして、管理アクセス権をユーザーに割り当て、root ユーザーアクセスを必要とするタスクの実行には root ユーザーのみを使用してください。

AWS アカウント ID は次のタスクで必要になるため、書き留めておいてください。

IAM ユーザーの作成

Apache Flink 用の Amazon マネージドサービスなどのサービスでは、アクセス時に認証情報を入力する必要があります。 AWSこれにより、サービスのリソースにアクセスする権限の有無が判定されます。 AWS Management Console にはパスワードの入力が必要です。

AWS アカウントが AWS Command Line Interface (AWS CLI) または API にアクセスするためのアクセスキーを作成できます。ただし、 AWS AWS アカウントの認証情報を使用してアクセスすることはお勧めしません。代わりに AWS Identity and Access Management (IAM) を使用することをお勧めします。IAM ユーザーを作成し、管理者アクセス許可を持つ IAM グループにユーザーを追加したら、作成した IAM ユーザーに管理者アクセス許可を付与します。その後、特別な URL とその IAM ユーザーの認証情報を使用して AWS にアクセスできます。

サインアップしたが AWS、自分用の IAM ユーザーをまだ作成していない場合は、IAM コンソールを使用して作成できます。

このガイドの使用開始実習では、管理者権限を持つユーザー (adminuser) が存在すること想定しています。手順に従ってアカウントに adminuser を作成します。

管理者グループを作成する

1. AWS Management Console にサインインし、https://console.aws.amazon.com/iam/ にある IAM コンソールを開きます。
2. ナビゲーションペインで、[Groups] (グループ)、[Create New Group] (新しいグループの作成) の順に選択します。
3. [Group Name] にグループの名前 (例: **Administrators**) を入力し、[Next Step] を選択します。
4. ポリシーのリストで、AdministratorAccessポリシーの横にあるチェックボックスを選択します。[フィルタ] メニューと [検索] ボックスを使用して、ポリシーのリストをフィルタリングできます。
5. [次のステップ]、[グループの作成] の順に選択します。

新しいグループは、[Group Name] の下に表示されます。

自分用の IAM ユーザーを作成するには、管理者グループにユーザーを追加し、パスワードを作成します。

1. ナビゲーションペインで [Users] (ユーザー)、[Add user] (ユーザーの追加) の順に選択します。
2. [ユーザー名] ボックスにユーザー名を入力します。
3. プログラミングによるアクセスとAWS マネジメントコンソールへのアクセスの両方を選択します。
4. [次のステップ: アクセス許可] を選択します。
5. [管理者] グループの横にあるチェックボックスを選択します。続いて、[Next: Review] をクリックします。
6. [ユーザーの作成] を選択します。

新しい IAM ユーザーとしてサインインするには

1. からサインアウトします AWS Management Console。
2. 次の URL 形式を使用してコンソールにサインインします。

   ```
   https://aws_account_number.signin.aws.amazon.com/console/
   ```

aws_account_number はハイフンなしのアカウント ID です。 AWS ***##### AWS ##### ID # 1234-5678-9012 #####aws_account_number ########### 123456789012***アカウント番号を確認する方法については、IAM ユーザーガイドの「アカウント ID とそのエイリアス」を参照してください。 AWS

3. 作成した IAM ユーザー名とパスワードを入力します。サインインすると、ナビゲーションバーに*your_user_name* @ *your_aws_account_id*が表示されます。

> **Note**
>
> サインインページの URL AWS にアカウント ID を含めたくない場合は、アカウントエイリアスを作成できます。

アカウントエイリアスを作成または削除するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. ナビゲーションペインで、ダッシュボード を選択します。
3. IAM ユーザーのサインインリンクを探します。
4. エイリアスを作成するには、[カスタマイズ] を選択します。エイリアスの名前を入力し、[はい、作成する] を選択します。
5. エイリアスを削除するには、カスタマイズ を選択してから、はい、作成する を選択します。サインイン URL はアカウント ID を使用するようになります。 AWS

アカウントエイリアスを作成した後、サインインするには、次の URL を使用します。

`https://your_account_alias.signin.aws.amazon.com/console/`

アカウントの IAM ユーザーのサインインリンクを確認するには、IAM コンソールを開き、ダッシュボードの [IAM users sign-in link] の下を確認します。

IAM の詳細については、以下を参照してください。

- AWS Identity and Access Management (IAM)
- IAM の使用開始
- IAM ユーザーガイド

次のステップ

ステップ 2: AWS Command Line Interface ()AWS CLIをセットアップする

ステップ 2: AWS Command Line Interface ()AWS CLIをセットアップする

このステップでは、をダウンロードして、Flink アプリケーション用 Apache Flink 用 Amazon マネージドサービスで使用するように設定します。 AWS CLI

> **Note**
>
> このガイドの使用開始実習では、操作を実行するために、アカウントの管理者の認証情報 (adminuser) を使用していることが前提となっています。

> **Note**
>
> AWS CLI を既にインストールしている場合、最新の機能を利用するにはアップグレードが必要な場合があります。詳細については、『AWS Command Line Interface ユーザーガイド』の「 AWS コマンドラインインターフェースのインストール」を参照してください。のバージョンを確認するには AWS CLI、以下のコマンドを実行します。
>
> ```
> aws --version
> ```
>
> このチュートリアルの演習には、 AWS CLI 以下のバージョン以降が必要です。
>
> ```
> aws-cli/1.16.63
> ```

をセットアップするには AWS CLI

1. AWS CLIをダウンロードして設定します。手順については、「AWS Command Line Interface ユーザーガイド」の次のトピックを参照してください。
   - AWS Command Line Interfaceのインストール
   - AWS CLIの設定
2. AWS CLI 管理者ユーザーの名前付きプロファイルを設定ファイルに追加します。このプロファイルは、 AWS CLI コマンドを実行するときに使用します。名前付きプロファイルの詳細につい

ては、AWS Command Line Interface ユーザーガイドの「名前付きプロファイル」を参照してください。

```
[profile adminuser]
aws_access_key_id = adminuser access key ID
aws_secret_access_key = adminuser secret access key
region = aws-region
```

AWS 使用可能なリージョンのリストについては、の「AWS リージョンとエンドポイント」を参照してください。Amazon Web Services 全般のリファレンス

3. コマンドプロンプトで以下のヘルプコマンドを入力して、セットアップを確認します。

```
aws help
```

AWS アカウントとを設定したら AWS CLI、次の実習 (サンプルアプリケーションを設定し、 end-to-end セットアップをテストする) に挑戦できます。

次のステップ

ステップ 3: Flink アプリケーション向けの Managed Service for Apache Flink を作成して実行する

ステップ 3: Flink アプリケーション向けの Managed Service for Apache Flink を作成して実行する

この演習では、データストリームをソースおよびシンクとして使用して、Flink アプリケーション向けの Managed Service for Apache Flink を作成します。

このセクションには、以下のステップが含まれています。

- 2 つの Amazon Kinesis Data Streams を作成する
- 入力ストリームへのサンプルレコードの書き込み
- Apache Flink Streaming Java Code のダウンロードと検証
- アプリケーションコードのコンパイル
- Apache Flink Streaming Java Code のアップロードしてください
- Managed Service for Apache Flink アプリケーションを作成して実行する

2 つの Amazon Kinesis Data Streams を作成する

この演習の Flink アプリケーション向けの Managed Service for Apache Flink を作成する前に、2 つの Kinesis データストリーム (`ExampleInputStream` と `ExampleOutputStream`) を作成してください。アプリケーションでは、これらのストリームを使用してアプリケーションの送信元と送信先のストリームを選択します。

これらのストリームは Amazon Kinesis コンソールまたは次の AWS CLI コマンドを使用して作成できます。コンソールを使用した手順については、データストリームの作成および更新を参照してください。

データストリームを作成するには (AWS CLI)

1. 最初のストリーム (`ExampleInputStream`) を作成するには、次の Amazon Kinesis `create-stream` AWS CLI コマンドを使用します。

```
$ aws kinesis create-stream \
--stream-name ExampleInputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

2. アプリケーションが出力の書き込みに使用する 2 つめのストリームを作成するには、ストリーム名を `ExampleOutputStream` に変更して同じコマンドを実行します。

```
$ aws kinesis create-stream \
--stream-name ExampleOutputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

入力ストリームへのサンプルレコードの書き込み

このセクションでは、Python スクリプトを使用して、アプリケーションが処理するサンプルレコードをストリームに書き込みます。

> **Note**
>
> このセクションでは AWS SDK for Python (Boto) が必要です。

1. 次の内容で、stock.py という名前のファイルを作成します。

```
import datetime
import json
import random
import boto3

STREAM_NAME = "ExampleInputStream"


def get_data():
    return {
        "EVENT_TIME": datetime.datetime.now().isoformat(),
        "TICKER": random.choice(["AAPL", "AMZN", "MSFT", "INTC", "TBV"]),
        "PRICE": round(random.random() * 100, 2),
    }


def generate(stream_name, kinesis_client):
    while True:
        data = get_data()
        print(data)
        kinesis_client.put_record(
            StreamName=stream_name, Data=json.dumps(data),
 PartitionKey="partitionkey"
        )


if __name__ == "__main__":
    generate(STREAM_NAME, boto3.client("kinesis"))
```

2. このチュートリアルの後半では、アプリケーションにデータを送信する stock.py スクリプトを実行します。

```
$ python stock.py
```

Apache Flink Streaming Java Code のダウンロードと検証

この例の Java アプリケーションコードは、 GitHubから入手できます。アプリケーションコードをダウンロードするには、次の操作を行います。

1. 次のコマンドを使用してリモートリポジトリのクローンを作成します。

```
git clone https://github.com/aws-samples/amazon-kinesis-data-analytics-java-examples.git
```

2. GettingStarted ディレクトリに移動します。

アプリケーションコードは CustomSinkStreamingJob.java ファイルと CloudWatchLogSink.java ファイルに含まれています。アプリケーションコードに関して、以下の点に注意してください。

- アプリケーションは Kinesis ソースを使用して、ソースストリームから読み取りを行います。次のスニペットでは、Kinesis シンクが作成されます。

```
return env.addSource(new FlinkKinesisConsumer<>(inputStreamName,
                new SimpleStringSchema(), inputProperties));
```

アプリケーションコードのコンパイル

このセクションでは、Apache Maven コンパイラを使用してアプリケーション用の Java コードを作成します。Apache Maven と Java 開発キット (JDK) をインストールする方法については、演習を完了するための前提条件を参照してください。

Java アプリケーションには、次のコンポーネントが必要です。

- プロジェクトオブジェクトモデル (pom.xml) ファイル。このファイルには、Flink アプリケーション向けの Amazon Managed Service for Apache Flink のライブラリなど、アプリケーションの設定と依存関係に関する情報が含まれています。
- アプリケーションのロジックを含む main メソッド。

> **Note**
>
> 次のアプリケーション用の Kinesis コネクタを使用するには、コネクタのソースコードをダウンロードして、Apache Flink ドキュメントで説明されているように構築する必要があります。

アプリケーションコードを作成してコンパイルするには

1. Java/Maven アプリケーションを開発環境で作成します。アプリケーションを作成する方法については、開発環境のドキュメントを参照してください。

   - 最初の Java プロジェクトの作成 (Eclipse Java Neon)
   - 最初の Java アプリケーションの作成、実行、およびパッケージング (IntelliJ Idea)

2. `StreamingJob.java` という名前のファイルに対して次のコードを使用します。

```
package com.amazonaws.services.kinesisanalytics;

import com.amazonaws.services.kinesisanalytics.runtime.KinesisAnalyticsRuntime;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisConsumer;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisProducer;
import
 org.apache.flink.streaming.connectors.kinesis.config.ConsumerConfigConstants;

import java.io.IOException;
import java.util.Map;
import java.util.Properties;

public class StreamingJob {

    private static final String region = "us-east-1";
    private static final String inputStreamName = "ExampleInputStream";
    private static final String outputStreamName = "ExampleOutputStream";

    private static DataStream<String>
 createSourceFromStaticConfig(StreamExecutionEnvironment env) {
        Properties inputProperties = new Properties();
        inputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);

 inputProperties.setProperty(ConsumerConfigConstants.STREAM_INITIAL_POSITION,
 "LATEST");

        return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
 SimpleStringSchema(), inputProperties));
    }
```

```
   private static DataStream<String>
createSourceFromApplicationProperties(StreamExecutionEnvironment env)
           throws IOException {
       Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
       return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(),
               applicationProperties.get("ConsumerConfigProperties")));
   }

   private static FlinkKinesisProducer<String> createSinkFromStaticConfig() {
       Properties outputProperties = new Properties();
       outputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
       outputProperties.setProperty("AggregationEnabled", "false");

       FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(), outputProperties);
       sink.setDefaultStream(outputStreamName);
       sink.setDefaultPartition("0");
       return sink;
   }

   private static FlinkKinesisProducer<String>
createSinkFromApplicationProperties() throws IOException {
       Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
       FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(),
               applicationProperties.get("ProducerConfigProperties"));

       sink.setDefaultStream(outputStreamName);
       sink.setDefaultPartition("0");
       return sink;
   }

   public static void main(String[] args) throws Exception {
       // set up the streaming execution environment
       final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

       /*
        * if you would like to use runtime configuration properties, uncomment the
        * lines below
```

```
         * DataStream<String> input = createSourceFromApplicationProperties(env);
         */

        DataStream<String> input = createSourceFromStaticConfig(env);

        /*
         * if you would like to use runtime configuration properties, uncomment the
         * lines below
         * input.addSink(createSinkFromApplicationProperties())
         */

        input.addSink(createSinkFromStaticConfig());

        env.execute("Flink Streaming Java API Skeleton");
    }
}
```

前述のコード例については、以下の点に注意してください。

- このファイルには、アプリケーションの機能を定義する main メソッドが含まれています。
- アプリケーションでは、ソースおよびシンクコネクタを作成し、StreamExecutionEnvironment オブジェクトを使用して外部リソースにアクセスします。
- アプリケーションでは、静的プロパティを使用してソースおよびシンクコネクタを作成します。動的なアプリケーションプロパティを使用するには、createSourceFromApplicationProperties および createSinkFromApplicationProperties メソッドを使用してコネクタを作成します。これらのメソッドは、アプリケーションのプロパティを読み取ってコネクタを設定します。

3. アプリケーションコードを使用するには、コードをコンパイルして JAR ファイルにパッケージ化します。コードのコンパイルとパッケージ化には次の 2 通りの方法があります。

   - Maven コマンドラインツールを使用します。pom.xml ファイルが格納されているディレクトリで次のコマンドを実行して JAR ファイルを作成します。

     ```
     mvn package
     ```

   - 開発環境を使用します。詳細については、開発環境のドキュメントを参照してください。

パッケージは JAR ファイルとしてアップロードすることも、圧縮して ZIP ファイルとしてアップロードすることもできします。を使用してアプリケーションを作成する場合は AWS CLI、コードコンテンツタイプ (JAR または ZIP) を指定します。

4. コンパイル中にエラーが発生した場合は、JAVA_HOME 環境変数が正しく設定されていることを確認します。

アプリケーションのコンパイルに成功すると、次のファイルが作成されます。

```
target/java-getting-started-1.0.jar
```

Apache Flink Streaming Java Code のアップロードしてください

このセクションでは、Amazon Simple Storage Service (Amazon S3) バケットを作成し、アプリケーションコードをアップロードします。

アプリケーションコードをアップロードするには

1. Amazon S3 コンソール (https://console.aws.amazon.com/s3/) を開きます。
2. [バケットを作成] を選択します。
3. [Bucket name (バケット名)] フィールドに**ka-app-code-*<username>***と入力します。バケット名にユーザー名などのサフィックスを追加して、グローバルに一意にします。[次へ] をクリックします。
4. 設定オプションのステップでは、設定をそのままにし、[次へ] を選択します。
5. アクセス許可の設定のステップでは、設定をそのままにし、[次へ] を選択します。
6. [バケットを作成] を選択します。
7. Amazon S3 コンソールで ka-app-code- <username>バケットを選択し、[アップロード] を選択します。
8. ファイルの選択のステップで、[ファイルを追加] を選択します。前のステップで作成した java-getting-started-1.0.jar ファイルに移動します。[次へ] をクリックします。
9. アクセス許可の設定のステップでは、設定をそのままにします。[次へ] をクリックします。
10. プロパティの設定のステップでは、設定をそのままにします。[アップロード] を選択します。

アプリケーションコードが Amazon S3 バケットに保存され、アプリケーションからアクセスできるようになります。

Managed Service for Apache Flink アプリケーションを作成して実行する

Flink アプリケーション向けの Managed Service for Apache Flink は、コンソールまたは AWS CLIのいずれかを使用して作成し、実行することができます。

> **Note**
>
> コンソールを使用してアプリケーションを作成すると、 AWS Identity and Access Management (IAM) リソースと Amazon CloudWatch Logs リソースが自動的に作成されます。を使用してアプリケーションを作成する場合 AWS CLI、これらのリソースは個別に作成します。

トピック

アプリケーションの作成と実行 (コンソール)

以下の手順を実行し、コンソールを使用してアプリケーションを作成、設定、更新、および実行します。

アプリケーションの作成

1. Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. Amazon Kinesis ダッシュボードで、[分析アプリケーションを作成する] を選択します。
3. [Kinesis Analytics - アプリケーションの作成] ページで、次のようにアプリケーションの詳細を指定します。
   - [アプリケーション名] には**MyApplication**と入力します。
   - [Description (説明)] に**My java test app**と入力します。
   - [ランタイム] には、[Apache Flink 1.6] を選択します。
4. [アクセス許可] には、[IAM ロールの作成 / 更新**kinesis-analytics-MyApplication-us-west-2**] を選択します。
5. [Create application] を選択します。

> **Note**
>
> コンソールを使用して Flink アプリケーション向けの Managed Service for Apache Flink を作成するときは、アプリケーション用の IAM ロールとポリシーを作成するオプションがあります。アプリケーションではこのロールとポリシーを使用して、依存リソースにアクセスします。これらの IAM リソースは、次のようにアプリケーション名とリージョンを使用して命名されます。
>
> - ポリシー: `kinesis-analytics-service-MyApplication-us-west-2`
> - ロール: `kinesis-analytics-MyApplication-us-west-2`

IAM ポリシーの編集

IAM ポリシーを編集し、Kinesis Data Streamsにアクセスするための許可を追加します。

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. [ポリシー] を選択します。前のセクションでコンソールによって作成された **kinesis-analytics-service-MyApplication-us-west-2** ポリシーを選択します。
3. [概要] ページで、[ポリシーの編集] を選択します。[JSON] タブを選択します。
4. 次のポリシー例で強調表示されているセクションをポリシーに追加します。サンプルのアカウント ID (*012345678901*) を自分のアカウント ID に置き換えます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadCode",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": [
                "arn:aws:s3:::ka-app-code-username/java-getting-started-1.0.jar"
            ]
        },
        {
            "Sid": "ListCloudwatchLogGroups",
            "Effect": "Allow",
```

```
            "Action": [
                "logs:DescribeLogGroups"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:*"
            ]
        },
        {
            "Sid": "ListCloudwatchLogStreams",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-
analytics/MyApplication:log-stream:*"
            ]
        },
        {
            "Sid": "PutCloudwatchLogs",
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-
analytics/MyApplication:log-stream:kinesis-analytics-log-stream"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
        }
    ]
```

```
}
```

アプリケーションの設定

1. MyApplicationページで [設定] を選択します。
2. [Configure application] ページで、[Code location] を次のように指定します。
   - [Amazon S3 バケット] で、**ka-app-code-*<username>***と入力します。
   - [Amazon S3 オブジェクトへのパス] で、**java-getting-started-1.0.jar**と入力します。
3. [Access to application resources] の [Access permissions] では、[Create / update IAM role**kinesis-analytics-MyApplication-us-west-2**] を選択します。
4. [Properties] の [Group ID] には、**ProducerConfigProperties**と入力します。
5. 次のアプリケーションのプロパティと値を入力します。

キー	値
flink.inputstream.initpos	**LATEST**
aws:region	**us-west-2**
AggregationEnabled	**false**

6. [Monitoring] の [Monitoring metrics level] が [Application] に設定されていることを確認します。
7. CloudWatch ロギングするには、「有効化」チェックボックスを選択します。
8. [更新] を選択します。

> **Note**
>
> CloudWatch ロギングを有効にすると、Apache Flink 用マネージドサービスによって自動的にロググループとログストリームが作成されます。これらのリソースの名前は次のとおりです。
>
> - ロググループ: `/aws/kinesis-analytics/MyApplication`
> - ログストリーム: `kinesis-analytics-log-stream`

アプリケーションを実行する

1. MyApplicationページで [実行] を選択します。アクションを確認します。
2. アプリケーションが実行されたら、ページを更新します。コンソールには [Application graph] が示されます。

アプリケーションの停止

MyApplicationページで [Stop] を選択します。アクションを確認します。

アプリケーションの更新

コンソールを使用して、アプリケーションのプロパティ、モニタリング設定、アプリケーション JAR の場所またはファイル名などのアプリケーション設定を更新できます。アプリケーションコードを更新する必要がある場合は、アプリケーション JAR を Amazon S3 バケットから再ロードすることもできます。

MyApplicationページで [設定] を選択します。アプリケーションの設定を更新し、[更新] を選択します。

アプリケーションの作成と実行 (AWS CLI)

このセクションでは、を使用して Apache Flink 用管理サービスアプリケーションを作成して実行します。 AWS CLI Apache Flink アプリケーション用マネージドサービスは、`kinesisanalyticsv2` AWS CLI コマンドを使用して Apache Flink アプリケーション用マネージドサービスを作成し、操作します。

アクセス許可ポリシーを作成する

まず、2 つのステートメントを含むアクセス許可ポリシーを作成します。1 つは、ソースストリームの `read` アクションに対するアクセス許可を付与し、もう 1 つはシンクストリームの `write` アクションに対するアクセス許可を付与します。次に、IAM ロール (次のセクションで作成) にポリシーをアタッチします。そのため、 Managed Service for Apache Flinkがこのロールを引き受けると、ソースストリームからの読み取りとシンクストリームへの書き込みを行うために必要なアクセス許可がサービスに付与されます。

次のコードを使用して `KAReadSourceStreamWriteSinkStream` アクセス許可ポリシーを作成します。*`username`* を Amazon S3 バケットの作成に使用したユーザー名に置き換え、アプリケーションコードを保存します。Amazon リソースネーム (ARN) のアカウント ID (*`012345678901`*) を自分のアカウント ID に置き換えます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": ["arn:aws:s3:::ka-app-code-username",
                "arn:aws:s3:::ka-app-code-username/*"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
        }
    ]
}
```

step-by-step アクセス権限ポリシーの作成手順については、IAM ユーザーガイドの「チュートリアル:初めてのカスタマー管理ポリシーを作成してアタッチする」を参照してください。

> **Note**
>
> AWS 他のサービスにアクセスするには、を使用できます。 AWS SDK for Java Managed Service for Apache Flink は、アプリケーションに関連付けられているサービス実行 IAM ロールに、SDK が必要とする認証情報を自動的に設定します。追加の手順は必要ありません。

IAM ロールを作成します。

このセクションでは、Flink アプリケーション向けの Managed Service for Apache Flink がソースストリームを読み取り、シンクストリームに書き込むために引き受けることができる IAM ロールを作成します。

Managed Service for Apache Flink が、許可のないままストリームにアクセスすることはできません。IAM ロールを介してこれらの許可を付与します。各 IAM ロールには、2 つのポリシーがアタッチされます。信頼ポリシーは、ロールを引き受けるための許可を Managed Service for Apache Flink 付与し、許可ポリシーは、ロールを引き受けた後に Managed Service for Apache Flink が実行できる事柄を決定します。

前のセクションで作成したアクセス許可ポリシーをこのロールにアタッチします。

IAM ロールを作成するには

1. IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. ナビゲーションペインで [Roles (ロール)]、[Create Role (ロールの作成)] の順に選択します。
3. [信頼されるエンティティの種類を選択] で、[AWS のサービス] を選択します。[このロールを使用するサービスを選択] で、[Kinesis Analytics] を選択します。[ユースケースの選択] で、[Kinesis Analytics] を選択します。

   [次のステップ: アクセス許可] を選択します。
4. [アクセス権限ポリシーをアタッチする] ページで、[Next: Review] (次: 確認) を選択します。ロールを作成した後に、アクセス許可ポリシーをアタッチします。
5. [Create role (ロールの作成)] ページで、ロールの名前に **`KA-stream-rw-role`** を入力します。[ロールを作成] を選択します。

   これで、`KA-stream-rw-role` と呼ばれる新しい IAM ロールが作成されます。次に、ロールの信頼ポリシーとアクセス許可ポリシーを更新します。
6. アクセス許可ポリシーをロールにアタッチします。

> **Note**
>
> この演習では、Managed Service for Apache Flink が、Kinesis データストリーム (ソース) からのデータの読み取りと、別の Kinesis データストリームへの出力の書き込みの両方を実行するためにこのロールを引き受けます。このため、前のステップで作成したポリシー、the section called “アクセス許可ポリシーを作成する” をアタッチします。

a. [概要] ページで、[アクセス許可] タブを選択します。

b. [Attach Policies (ポリシーのアタッチ)] を選択します。

c. 検索ボックスに**KAReadSourceStreamWriteSinkStream**(前のセクションで作成したポリシー) と入力します。

d. KA ReadInputStreamWriteOutputStream ポリシーを選択し、[ポリシーをアタッチ] を選択します。

これで、アプリケーションがリソースにアクセスするために使用するサービスの実行ロールが作成されました。新しいロールの ARN を書き留めておきます。

step-by-step ロールの作成方法については、『IAM ユーザーガイド』の「IAM ロールの作成 (コンソール)」を参照してください。

Apache Flink アプリケーション用 Managed Serviceの作成

1. 次の JSON コードを `create_request.json` という名前のファイルに保存します。サンプルロールの ARN を、前に作成したロールの ARN に置き換えます。バケット ARN のサフィックス (*username*) を、前のセクションで選択したサフィックスに置き換えます。サービス実行ロールのサンプルのアカウント ID (*012345678901*) を、自分のアカウント ID に置き換えます。

```
{
    "ApplicationName": "test",
    "ApplicationDescription": "my java test app",
    "RuntimeEnvironment": "FLINK-1_6",
    "ServiceExecutionRole": "arn:aws:iam::012345678901:role/KA-stream-rw-role",
    "ApplicationConfiguration": {
        "ApplicationCodeConfiguration": {
            "CodeContent": {
                "S3ContentLocation": {
                    "BucketARN": "arn:aws:s3:::ka-app-code-username",
                    "FileKey": "java-getting-started-1.0.jar"
                }
            },
            "CodeContentType": "ZIPFILE"
        },
        "EnvironmentProperties":  {
         "PropertyGroups": [
            {
               "PropertyGroupId": "ProducerConfigProperties",
```

```
                "PropertyMap" : {
                     "flink.stream.initpos" : "LATEST",
                     "aws.region" : "us-west-2",
                     "AggregationEnabled" : "false"
                }
            },
            {
                "PropertyGroupId": "ConsumerConfigProperties",
                "PropertyMap" : {
                     "aws.region" : "us-west-2"
                }
            }
         ]
      }
    }
}
```

2. 前述のリクエストを指定して CreateApplication アクションを実行し、アプリケーションを作成します。

```
aws kinesisanalyticsv2 create-application --cli-input-json file://
create_request.json
```

これでアプリケーションが作成されました。次のステップでは、アプリケーションを起動します。

アプリケーションの起動

このセクションでは、StartApplication アクションを使用してアプリケーションを起動します。

アプリケーションを起動するには

1. 次の JSON コードを start_request.json という名前のファイルに保存します。

```
{
    "ApplicationName": "test",
    "RunConfiguration": {
        "ApplicationRestoreConfiguration": {
         "ApplicationRestoreType": "RESTORE_FROM_LATEST_SNAPSHOT"
         }
    }
}
```

2. 前述のリクエストを指定して StartApplication アクションを実行し、アプリケーションを起動します。

```
aws kinesisanalyticsv2 start-application --cli-input-json file://start_request.json
```

アプリケーションが実行されます。Amazon CloudWatch コンソールで Apache Flink 用マネージドサービスのメトリックスを確認して、アプリケーションが動作していることを確認できます。

アプリケーションの停止

このセクションでは、StopApplication アクションを使用してアプリケーションを停止します。

アプリケーションを停止するには

1. 次の JSON コードを stop_request.json という名前のファイルに保存します。

```
{"ApplicationName": "test"
}
```

2. 次のリクエストを指定して StopApplication アクションを実行し、アプリケーションを停止します。

```
aws kinesisanalyticsv2 stop-application --cli-input-json file://stop_request.json
```

アプリケーションが停止します。

チュートリアル: AWS Lambda を Amazon Kinesis Data Streams で使用する

このチュートリアルでは、Kinesis データストリームのイベントを処理する Lambda 関数を作成します。次のシナリオ例では、カスタムアプリケーションによって Kinesis データストリームにレコードを書き込みます。AWS次に、Lambda はこのデータストリームをポーリングし、新しいデータレコードを検出すると、Lambda 関数を呼び出します。AWSLambda は、Lambda 関数の作成時に指定した実行ロールを引き受けることによって Lambda 関数を実行します。

ステップバイステップの手順の詳細については、チュートリアル: Amazon Kinesis で AWS Lambda を使用するを参照してください。

> **Note**
>
> このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての場合は、AWS Lambda の開始方法の手順に従って最初の Lambda 関数を作成してください。

AWS Streaming Data Solution for Amazon Kinesis

AWS Streaming Data Solution for Amazon Kinesis は、ストリーミングデータを簡単に取得、保存、処理、および配信するために必要な AWS サービスを自動的に設定します。このソリューションは、Kinesis Data Streams、AWS Lambda、Amazon API Gateway、および Amazon Managed Service for Apache Flink などの複数の AWS サービスを使用するストリーミングデータユースケースを解決するために、複数のオプションを提供します。

各ソリューションには、以下のコンポーネントとが含まれています。

- 完全な例をデプロイするための AWS CloudFormation パッケージ。
- アプリケーションのメトリクスを表示するための CloudWatch ダッシュボード。
- CloudWatch は、最も関連性の高いアプリケーションメトリクスでアラームを発します。
- すべての必要な IAM ロールとポリシー

ソリューションはこちらでご覧いただけます。Amazon Kinesis 向けストリーミングデータソリューション

ストリームの作成と管理

Amazon Kinesis Data Streams は、大量のデータをリアルタイムで取り込み、そのデータを永続的に保存して、消費できるようにします。Kinesis Data Streams によって保存されるデータの単位は、データレコードです。データストリームは、データレコードのグループを表します。データストリームのデータレコードはシャードに配分されます。

シャードには、ストリーム内のデータレコードのシーケンスです。Kinesis Data Streams の基本スループット単位として機能します。シャードは、オンデマンドモードとプロビジョンド容量モードの両方で、 書き込み 用に 1 MB/秒 と 1,000 レコード/秒、読み取り用に 2 MB/秒をサポートします。シャード制限により、予測可能なパフォーマンスが保証され、信頼性の高いデータストリーミングワークフローの設計と運用が容易になります。

トピック

データストリーム容量モードの選択

トピック

データストリーム容量モードとは

容量モードは、データストリームの容量の管理方法と、データストリームの使用に対する課金方法を決定します。Amazon Kinesis Data Streams では、データストリームのオンデマンドモードとプロビジョンドモードのどちらかを選択できます。

- オンデマンド - オンデマンドモードのデータストリームは、容量計画を必要とせず、1 分あたりの書き込みおよび読み取りスループットのギガバイトを処理するように自動的にスケーリングされます。オンデマンドモードでは、Kinesis Data Streams は必要なスループットを提供するために、シャードを自動的に管理します。
- プロビジョンド - プロビジョンドモードのデータストリームの場合、データストリームのシャードカウントを指定する必要があります。データストリームの総容量は、シャードの容量の合計です。必要に応じて、データストリームのシャードの数を増減することができます。

Kinesis Data Streams `PutRecord` および `PutRecords` API を使用して、オンデマンドとプロビジョンドの両方の容量モードでデータストリームにデータを書き込むことができます。データを取得するために、どちらの容量モードも、`GetRecords` API を使用するデフォルトのコンシューマーと `SubscribeToShard` API を使用する拡張ファンアウト (EFO) のコンシューマーをサポートします。

保持モード、暗号化、モニタリングメトリクスなど、すべての Kinesis Data Streams の機能は、オンデマンドモードとプロビジョンドモードの両方でサポートされています。Kinesis Data Streams は、オンデマンドおよびプロビジョンドの容量モードの両方で、高い耐久性と可用性を提供します。

オンデマンドモード

オンデマンドモードのデータストリームは、容量計画を必要とせず、1 分あたりの 書き込みおよび読み取り スループットのギガバイトを処理するように自動的にスケーリングされます。オンデマンドモードでは、サーバー、ストレージ、またはスループットの プロビジョニングや管理が 不要になるため、低レイテンシーで大量のデータを取り込んで保存することが簡単になります。運用上のオーバーヘッドなしで、1 日あたり数十億ものレコードを取り込むことができます。

オンデマンドモードは、変動性が高く、予測不可能なアプリケーショントラフィックのニーズに対応するのに最適です。これらのワークロードをピーク容量にプロビジョニングする必要がなくなり、使用率が低いため、コストが高くなる可能性があります。オンデマンドモードは、予測不能で変動性の高いトラフィックパターンを持つワークロードに適しています。

オンデマンド容量モードでは、データストリームから読み書きされるデータのギガバイトあたりの料金が発生します。アプリケーションで実行することが予測される読み込みおよび書き込みスループットを指定する必要はありません。Kinesis Data Streams は、ワークロードが増加または減少するときに、即座にワークロードに対応します。詳細については、Amazon Kinesis Data Streams の料金表を参照してください。

Kinesis Data Streams コンソール、API、または CLI コマンドを使用して、オンデマンドモードで新しいデータストリームを 作成できます。

オンデマンドモードのデータストリームは、過去 30 日間に観測されたピーク書き込みスループットの最大 2 倍に対応します。データストリームの書き込みスループットが新しいピークに達すると、Kinesis Data Streams はデータストリームの容量を自動的にスケーリングします。例えば、データストリームの書き込みスループットが 10 MB/秒から 40 MB/秒の間で変化する場合、Kinesis Data Streams を使用すると、以前のピークスループットの 2 倍または 80 MB/秒に簡単にバーストできます。同じデータストリームが 50 MB/秒の新しいピークスループットを維持する場合、Kinesis Data Streams は 100 MB/秒の書き込みスループットを取り込むのに十分な容量を確保します。ただし、15 分以内にトラフィックが以前のピークの 2 倍以上に増加すると、書き込みスロットリングが発生する可能性があります。これらのスロットルされたリクエストは再試行する必要があります。

オンデマンドモードのデータストリームの総読み込み容量は、書き込みスループットに比例して増加します。これにより、コンシューマーアプリケーションは、受信データをリアルタイムで処理するための適切な読み取りスループットを常に確保できます。`GetRecords` API を使用したデータの読み取りと比較して、書き込みスループットは少なくとも 2 倍になります。`GetRecord` API で 1 つのコンシューマーアプリケーションを使用することをお勧めします。これにより、アプリケーションがダウンタイムから回復する必要があるときに追いつくのに十分なスペースが確保されます。複数のコンシューマーアプリケーションを追加する必要があるシナリオでは、Kinesis Data Streams の拡張ファンアウト機能を使用することをお勧めします。拡張ファンアウトは、`SubscribeToShard` API を使用して最大 20 のコンシューマーアプリケーションをデータストリームに追加することをサポートし、各コンシューマーアプリケーションは専用のスループットを備えています。

読み取りおよび書き込みスループットの例外処理

オンデマンド容量モード (プロビジョンド容量モードと同じ) では、データストリームにデータを書き込むために、各レコードでパーティションキーを指定する必要があります。Kinesis Data Streams は、パーティションキーを使用して、シャード間でデータを分散します。Kinesis Data Streams は、各シャードのトラフィックをモニタリングします。着信トラフィックがシャードあたり 500 KB/秒を超えると、15 分以内にシャードが分割されます。親シャードのハッシュキー値は、子シャード間で均等に再配分されます。

着信トラフィックが以前のピークの 2 倍を超えると、データがシャード全体に均等に分散されていても、約 15 分間、読み取りまたは書き込みの例外が発生する可能性があります。すべてのレコードが Kinesis Data Streams に適切に保存されるように、このようなリクエストをすべて再試行することをお勧めします。

不均等なデータ分散につながるパーティションキーを使用し、特定のシャードに割り当てられたレコードがその制限を超えると、読み取りおよび書き込みの例外が発生することがあります。オンデマンドモードでは、単一のパーティションキーがシャードの 1 MB/秒のスループットおよび 1,000 レコード/秒の制限を超えない限り、データストリームは不均等なデータ分散パターンを処理するように自動的に適応します。

オンデマンドモードでは、トラフィックの増加が検出されると、Kinesis Data Streams はシャードを均等に分割します。ただし、特定のシャードへの着信トラフィックの上位部分を駆動しているハッシュキーは検出および分離されません。非常に不均等なパーティションキーを使用している場合は、書き込みの例外を引き続き受け取る可能性があります。このようなユースケースでは、きめ細かくシャードの分割をサポートするプロビジョンド容量モードを使用することをお勧めします。

プロビジョニングモード

プロビジョニングモードでは、データストリームを作成した後、 AWS Management Console または UpdateShardCount API を使用してシャード容量を動的にスケールアップまたはスケールダウンできます。Kinesis Data Streams 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が、ストリームに対してデータを書き込んだり、ストリームからデータを読み取ったりしている間に更新を行うことができます。

プロビジョンドモードは、予測しやすい容量要件を持つ予測可能なトラフィックに適しています。シャード間でのデータの分散方法をきめ細かく制御したい場合は、プロビジョンドモードを使用できます。

プロビジョンドモードでは、データストリームのシャードカウントを指定する必要があります。プロビジョンドモードでデータストリームのサイズを決定するには、以下の入力値が必要です。

- ストリームに書き込まれるデータレコードの KB 単位での平均サイズ (近似の KB 単位 (`average_data_size_in_KB`) まで切り上げられます)。
- 1 秒間にストリームで読み書きされるデータレコードの数 (`records_per_second`) です。
- ストリームから独立して同時にデータを消費する Kinesis Data Streams アプリケーションであるコンシューマーの数 (`number_of_consumers`)。

- KB 単位での受信書き込み帯域幅 (incoming_write_bandwidth_in_KB)。average_data_size_in_KBを records_per_second に乗算した値に等しくなります。
- KB 単位の送信読み取り帯域幅 (outgoing_read_bandwidth_in_KB)。incoming_write_bandwidth_in_KBを number_of_consumers に乗算した値に等しくなります。

ストリームに必要なシャードの数 (number_of_shards) を計算するには、入力値を以下の式にあてはめます。

```
number_of_shards = max(incoming_write_bandwidth_in_KiB/1024,
 outgoing_read_bandwidth_in_KiB/2048)
```

ピークスループットを処理するようにデータストリームを設定しないと、プロビジョンドモードで読み取りおよび書き込みのスループットの例外が発生する可能性があります。この場合、データトラフィックに対応するようにデータストリームを手動でスケーリングする必要があります。

不均等なデータ分散につながるパーティションキーを使用し、あるシャードに割り当てられたレコードがその制限を超えると、読み取りおよび書き込みの例外が発生することがあります。プロビジョンドモードでこの問題を解決するには、このようなシャードを特定し、トラフィックに上手く対応できるように手動で分割します。詳細については、ストリームのリシャーディングを参照してください。

容量モード間の切り替え

データストリームの容量モードをオンデマンドからプロビジョンド、またはプロビジョンドからオンデマンドに切り替えることができます。 AWS アカウントのデータストリームごとに、オンデマンド容量モードとプロビジョンド容量モードを 24 時間で 2 回切り替えることができます。

データストリームの容量モードを切り替えても、このデータストリームを使用するアプリケーションが中断することはありません。このデータストリームへの書き込みとデータストリームからの読み取りを続行できます。オンデマンドからプロビジョンド、またはプロビジョンドからオンデマンドのいずれかで容量モードを切り替えると、ストリームのステータスは更新中に設定されます。プロパティを再度変更するには、データストリームのステータスがアクティブになるのを待つ必要があります。

プロビジョンド容量モードからオンデマンド容量モードに切り替えると、データストリームは最初に移行前のシャードカウントを保持します。この時点から、Kinesis Data Streams はデータトラフィックをモニタリングし、書き込みスループットに応じて、このオンデマンドデータストリームのシャードカウントをスケーリングします。

オンデマンドモードからプロビジョンドモードに切り替えると、データストリームは最初に移行前のシャードカウントを保持しますが、この時点から、書き込みスループットに適切に対応するために、このデータストリームのシャードカウントをモニタリングおよび調整する必要があります。

AWS マネジメントコンソールを使用したストリームの作成

Kinesis Data Streams コンソール、Kinesis Data Streams API、または AWS Command Line Interface (AWS CLI) を使用して、ストリームを作成できます。

コンソールを使用してデータストリームを作成するには

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/kinesis で Kinesis コンソールを開きます。
2. ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
3. [データストリームの作成] を選択します。
4. [Create Kinesis stream] (Kinesis ストリームの作成) ページで、データストリームの名前を入力し、[On-demand] (オンデマンド) または[Provisioned] (プロビジョンド) のどちらかの容量モードを選択します。デフォルトでは、[On-demand] (オンデマンド) モードが選択されます。詳細については、データストリーム容量モードの選択を参照してください。

   [On-demand] (オンデマンド) モードの場合、[Create Kinesis stream] (Kinesis ストリームの作成) を選択して、データストリームを作成することができます。[Provisioned] (プロビジョンド) モードの場合、必要なシャードの数を指定してから、[Create Kinesis stream] (Kinesis ストリームの作成) を選択します。

   ストリームの作成中、[Kinesis ストリーム] ページのストリームのステータスは、Creating になります。ストリームを使用する準備が完了すると、ステータスは Active に変わります。
5. ストリームの名前を選択します。[ストリームの詳細] ページには、ストリーム設定の概要とモニタリング情報が表示されます。

Kinesis Data Streams API を使用してストリームを作成するには

- Kinesis Data Streams API を使用したストリームの作成については、API を使用したストリームの作成を参照してください。

を使用してストリームを作成するには AWS CLI

- を使用してストリームを作成する方法については AWS CLI、create-stream コマンドを参照してください。

API を使用したストリームの作成

次の手順で Kinesis Data Streams を作成します。

Kinesis Data Streams クライアントの構築

Kinesis Data Streams を使用する前に、クライアントオブジェクトを構築する必要があります。次の Java コードは、クライアントビルダーをインスタンス化し、それを使用してリージョン、認証情報、およびクライアント設定を指定します。次に、クライアントオブジェクトを構築します。

```
AmazonKinesisClientBuilder clientBuilder = AmazonKinesisClientBuilder.standard();

clientBuilder.setRegion(regionName);
clientBuilder.setCredentials(credentialsProvider);
clientBuilder.setClientConfiguration(config);

AmazonKinesis client = clientBuilder.build();
```

詳細については、「AWS 全般のリファレンス」で「Kinesis Data Streams のリージョンとエンドポイント」を参照してください。

ストリームを作成する

Kinesis Data Streams クライアントを作成したら、使用するストリームを作成できます。この作業は、Kinesis Data Streams コンソールまたはプログラムから実行できます。プログラムでストリームを作成するには、CreateStreamRequest オブジェクトをインスタンス化し、ストリームの名前と (プロビジョンドモードを使用するなら) ストリームが使用するシャードの数を指定します。

- オンデマンド:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
```

- プロビジョンド:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
createStreamRequest.setShardCount( myStreamSize );
```

ストリーム名はストリームを識別するために使用されます。名前の範囲は、アプリケーションが使用する AWS アカウントに限定されます。また、リージョンにも限定されます。つまり、2 つの異なる AWS アカウントの 2 つのストリームは同じ名前を持つことができ、2 つの異なるリージョンの AWS 2 つのストリームは同じ名前を持つことができますが、同じアカウントと同じリージョンの 2 つのストリームは使用できません。

ストリームのスループットはシャードの数によって決まります。プロビジョンドスループットを高くするほど、必要になるシャードの数は増えます。シャードが増えると、ストリームに AWS 課金されるコストも増加します。アプリケーションに適切なシャードの数の計算の詳細については、データストリーム容量モードの選択を参照してください。

createStreamRequest オブジェクトを設定した後、クライアントの createStream メソッドを呼び出すことで、ストリームを作成します。createStream の呼び出し後、ストリームに対してさらにオペレーションを実行するには、ストリームが ACTIVE 状態になるまで待機します。ストリームの状態を確認するには、describeStream メソッドを呼び出します。ただし、ストリームが存在しない場合、describeStream は例外をスローします。そのために、describeStream の呼び出しは try/catch ブロックで囲みます。

```
client.createStream( createStreamRequest );
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime ) {
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus =
 describeStreamResponse.getStreamDescription().getStreamStatus();
```

```
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
    //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

ストリームの更新

Kinesis Data Streams コンソール、Kinesis Data Streams API、または AWS CLIを使用して、ストリームの詳細を更新できます。

> **Note**
>
> 既存のストリーム、または最近作成したストリームに対して、サーバー側の暗号化を有効にすることができます。

コンソールを使用してデータストリームを更新するには

1. Amazon Kinesis コンソール (https://console.aws.amazon.com/kinesis/) を開きます。
2. ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
3. リストのストリームの名前を選択します。[ストリームの詳細] ページには、ストリーム設定の概要とモニタリング情報が表示されます。
4. データストリームのオンデマンド容量モードとプロビジョンド容量モードを切り替えるには、[Configuratio] (設定) タブの [Edit capacity mode] (容量モードの編集) を選択します。詳細については、「データストリーム容量モードの選択」を参照してください。

> **Important**
>
> AWS アカウントのデータストリームごとに、オンデマンドモードとプロビジョニングモードを 24 時間以内に 2 回切り替えることができます。

5. プロビジョンドモードのデータストリームの場合、シャードの数を編集するには、[Configuration] (設定) タブの [Edit provisioned shards] (プロビジョニングされたシャードの編集) を選択し、新しいシャードカウントを入力します。
6. データレコードのサーバー側の暗号化を有効にするには、[サーバー側の暗号化] セクションの [編集] を選択します。暗号化のマスターキーとして使用する KMS キーを選択するか、 Kinesis によって管理されるデフォルトのマスターキー aws/kinesis を使用します。ストリームの暗号化を有効にし、独自の AWS KMS マスターキーを使用する場合は、プロデューサーアプリケーションとコンシューマーアプリケーションが、使用した AWS KMS マスターキーにアクセスできることを確認してください。ユーザーが生成した AWS KMS キーにアクセスするためのアクセス許可をアプリケーションに割り当てるには、the section called “ユーザー生成の KMS マスターキーを使用するための許可”を参照してください。
7. データ保持期間を編集するには、[Data retention period] セクションの [Edit] を選択し、新しいデータ保持期間を入力します。
8. アカウントでカスタムメトリクスを有効にした場合は、[シャードレベルメトリクスの編集] セクションの [編集] を選択し、ストリームのメトリクスを指定します。詳細については、the section called “による サービスのモニタリング CloudWatch”を参照してください。

API を使用したストリームの更新

API を使用してストリームの詳細を更新するには、次の方法を参照してください。

- AddTagsToStream
- DecreaseStreamRetentionPeriod
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- IncreaseStreamRetentionPeriod
- RemoveTagsFromStream
- StartStreamEncryption
- StopStreamEncryption

- UpdateShardCount

を使用したストリームの更新 AWS CLI

を使用してストリームを更新する方法については AWS CLI、Kinesis CLI リファレンス を参照してください。

ストリームのリスト

前のセクションで説明したように、ストリームの範囲は、Kinesis Data Streams クライアントのインスタンス化に使用される AWS 認証情報に関連付けられた AWS アカウントと、クライアントに指定されたリージョンに限定されます。 AWS アカウントでは、一度に多数のストリームをアクティブにすることができます。ストリームは、Kinesis Data Streams コンソールでリストするか、プログラムによってリストすることができます。このセクションのコードは、 AWS アカウントのすべてのストリームを一覧表示する方法を示しています。

```
ListStreamsRequest listStreamsRequest = new ListStreamsRequest();
listStreamsRequest.setLimit(20);
ListStreamsResult listStreamsResult = client.listStreams(listStreamsRequest);
List<String> streamNames = listStreamsResult.getStreamNames();
```

このコード例では、最初に `ListStreamsRequest` の新しいインスタンスを作成し、その `setLimit` メソッドを呼び出して、最大 20 個のストリームが `listStreams` の呼び出しごとに返されるように指定しています。`setLimit` の値を指定しない場合は、アカウント内のストリーム数以下のストリームが Kinesis Data Streams によって返されます。次に、コードはクライアントの `listStreamsRequest` メソッドに `listStreams` を渡します。`listStreams` の戻り値は `ListStreamsResult` オブジェクトに格納されます。コードはこのオブジェクトの `getStreamNames` メソッドを呼び出して、返されたストリームの名前を `streamNames` リストに格納します。アカウントとリージョンにこの制限で指定したよりも多くのストリームがある場合でも、Kinesis Data Streams によって返されるストリームの数が指定した制限に満たないことがあります。確実にすべてのストリームを取得するには、次のコード例で説明している `getHasMoreStreams` メソッドを使用します。

```
while (listStreamsResult.getHasMoreStreams())
{
    if (streamNames.size() > 0) {
      listStreamsRequest.setExclusiveStartStreamName(streamNames.get(streamNames.size()
 - 1));
```

```
    }
    listStreamsResult = client.listStreams(listStreamsRequest);
    streamNames.addAll(listStreamsResult.getStreamNames());
}
```

このコードは、getHasMoreStreams の listStreamsRequest メソッドを呼び出して、listStreams の最初の呼び出しで返されたストリームの数よりも多いストリームがあるかどうかを確認します。ある場合、コードは setExclusiveStartStreamName メソッドを呼び出して、listStreams の前の呼び出しで返された最後のストリームの名前を指定します。setExclusiveStartStreamName メソッドは listStreams の次の呼び出しをそのストリームの後から開始します。その呼び出しによって返されたストリーム名のグループが streamNames リストに追加されます。すべてのストリームの名前がリストに収集されるまで、この処理を続行します。

listStreams で返されるストリームは以下のいずれかの状態になります。

- CREATING
- ACTIVE
- UPDATING
- DELETING

前のdescribeStreamで示した API を使用したストリームの作成 メソッドを使用して、ストリームの状態を確認できます。

シャードの一覧表示

データストリームは 1 つ以上のシャードを持つことができます。データストリームからシャードを一覧表示 (または取得) するには、2 つの方法があります。

トピック

- ListShards API - 推奨
- DescribeStream API - 非推奨

ListShards API - 推奨

データストリームからシャードを一覧表示または取得するための推奨方法は、 ListShards API を使用することです。次の例では、データストリーム内のシャードを一覧表示する方法を示します。この

例で使用されているメインオペレーションと、オペレーションに設定できるすべてのパラメータの詳細については、「」を参照してくださいListShards。

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ListShardsRequest;
import software.amazon.awssdk.services.kinesis.model.ListShardsResponse;

import java.util.concurrent.TimeUnit;

public class ShardSample {

    public static void main(String[] args) {

        KinesisAsyncClient client = KinesisAsyncClient.builder().build();

        ListShardsRequest request = ListShardsRequest
                .builder().streamName("myFirstStream")
                .build();

        try {
            ListShardsResponse response = client.listShards(request).get(5000,
 TimeUnit.MILLISECONDS);
            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

前のコード例を実行するには、次のような POM ファイルを使用できます。

```
<?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">
    <modelVersion>4.0.0</modelVersion>

    <groupId>kinesis.data.streams.samples</groupId>
    <artifactId>shards</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
```

```
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>kinesis</artifactId>
            <version>2.0.0</version>
        </dependency>
    </dependencies>
</project>
```

ListShards API では、 ShardFilterパラメータを使用して API のレスポンスを除外できます。一度に 1 つのフィルターしか指定できません。

API を呼び出す ListShardsときに ShardFilterパラメータを使用する場合、 Typeは必須プロパティであり、指定する必要があります。AT_TRIM_HORIZON、FROM_TRIM_HORIZON、または AT_LATEST タイプを指定する場合は、ShardId または Timestamp のオプションのプロパティを指定する必要はありません。

AFTER_SHARD_ID タイプを指定する場合は、オプションの ShardId プロパティの値も指定する必要があります。ShardId プロパティは、 ListShards API の ExclusiveStartShardIdパラメータと同じ機能です。ShardId プロパティが指定されている場合、レスポンスには、指定した ShardId の直後に ID が続くシャードで始まるシャードが含まれます。

AT_TIMESTAMP または FROM_TIMESTAMP_ID タイプを指定する場合は、オプションの Timestamp プロパティの値も指定する必要があります。AT_TIMESTAMP タイプを指定する場合は、指定されたタイムスタンプで開いていたすべてのシャードが返されます。FROM_TIMESTAMP タイプを指定する場合は、TIP に指定されたタイムスタンプから始まるすべてのシャードが返されます。

Important

`DescribeStreamSummary` および `ListShard` API は、データストリームに関する情報を取得するための、よりスケーラブルな方法を提供します。具体的には、 DescribeStream API のクォータによってスロットリングが発生する可能性があります。詳細については、「クォータと制限」を参照してください。また、`DescribeStream`クォータは AWS 、アカウント内のすべてのデータストリームとやり取りするすべてのアプリケーション間で共有されることに注意してください。一方、 ListShards API のクォータは、単一のデータストリームに固有です。そのため、 ListShards API を使用すると TPS が高くなるだけでなく、データストリームをさらに作成するにつれてアクションのスケーリングも向上します。
API を呼び出すすべてのプロデューサーとコンシューマーを移行して、代わりに DescribeStream DescribeStreamSummary と ListShard APIsを呼び出すことをお勧めします。これらのプロデューサーとコンシューマーを特定するには、KPL と KCL のユーザーエージェントが API コールにキャプチャされるため、Athena を使用して CloudTrail ログを解析することをお勧めします。

```
SELECT useridentity.sessioncontext.sessionissuer.username,
useridentity.arn,eventname,useragent, count(*) FROM
cloudtrail_logs WHERE Eventname IN ('DescribeStream')  AND
eventtime
    BETWEEN ''
        AND ''
GROUP BY
 useridentity.sessioncontext.sessionissuer.username,useridentity.arn,eventname,useragent
ORDER BY  count(*) DESC LIMIT 100
```

また、`DescribeStream`API を呼び出す AWS Lambda および Amazon Firehose と Kinesis Data Streams の統合は、統合が代わりに `DescribeStreamSummary`および を呼び出すように再設定することをお勧めしますListShards。具体的には、 AWS Lambda の場合は、イベントソースマッピングを更新する必要があります。Amazon Firehose については、対応する IAM 許可を更新して、それらに `ListShards` IAM 許可を含める必要があります。

DescribeStream API - 非推奨

> **Important**
>
> 以下の情報は、 DescribeStream API を介してデータストリームからシャードを取得する現在非推奨の方法について説明します。現在、ListShards API を使用してデータストリームを構成するシャードを取得することを強くお勧めしています。

describeStream メソッドによって返された応答オブジェクトを使用すると、ストリームを構成するシャードについて情報を取得できます。シャードを取得するには、このオブジェクトの getShards メソッドを呼び出します。このメソッドは、1 回の呼び出しでストリームからすべてのシャードを返すとは限りません。以下のコードでは、getHasMoreShards の getStreamDescription メソッドを使用して、返されなかったシャードがあるかどうかを確認しています。ある場合、つまり、このメソッドが true を返した場合は、ループ内で getShards の呼び出しを繰り返して、返されたシャードの新しいバッチをシャードのリストに追加していきます。getHasMoreShards が false を返した場合は、ループが終了します。つまり、すべてのシャードが返されたことになります。getShards は 状態のシャードを返さないことに注意してください。EXPIREDシャードの状態 (EXPIRED 状態など) の詳細については、リシャーディング後のデータのルーティング、データの永続化、シャードの状態を参照してください。

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );
List<Shard> shards = new ArrayList<>();
String exclusiveStartShardId = null;
do {
    describeStreamRequest.setExclusiveStartShardId( exclusiveStartShardId );
    DescribeStreamResult describeStreamResult =
 client.describeStream( describeStreamRequest );
    shards.addAll( describeStreamResult.getStreamDescription().getShards() );
    if (describeStreamResult.getStreamDescription().getHasMoreShards() && shards.size()
 > 0) {
        exclusiveStartShardId = shards.get(shards.size() - 1).getShardId();
    } else {
        exclusiveStartShardId = null;
    }
} while ( exclusiveStartShardId != null );
```

ストリームを削除する

ストリームは Kinesis Data Streams コンソールで削除するか、プログラムによって削除することができます。ストリームをプログラムで削除するには、次のコードに示されているように `DeleteStreamRequest` を使用します。

```
DeleteStreamRequest deleteStreamRequest = new DeleteStreamRequest();
deleteStreamRequest.setStreamName(myStreamName);
client.deleteStream(deleteStreamRequest);
```

ストリームを削除する前に、そのストリーム上で動作しているアプリケーションをすべてシャットダウンします。削除したストリームとアプリケーションがやり取りしようとすると、`ResourceNotFound` 例外を受け取ります。また、削除前のストリームと同じ名前の新しいストリームを作成した場合、前のストリームとやり取りしていたアプリケーションが実行されていると、それらのアプリケーションが前のストリームであるかのように新しいストリームとやり取りしようとして、予期しない動作につながることがあります。

ストリームをリシャーディングする

> **Important**
>
> UpdateShardCount API を使用してストリームを再シャードできます。それ以外の場合は、ここで説明したように分割とマージを実行できます。

Amazon Kinesis Data Streams では、リシャーディングがサポートされています。リシャーディングでは、ストリーム内のシャードの数を調整して、ストリームのデータフロー率の変化に適応させることができます。リシャーディングは高度なオペレーションと見なされます。Kinesis Data Streams を初めて使用する場合は、Kinesis Data Streams の他のあらゆる機能に詳しくなってから、このトピックをお読みください。

リシャーディングには、シャードの分割と結合という 2 種類のオペレーションがあります。シャードの分割では、1 つのシャードを 2 つシャードに分けます。シャードの結合では、2 つシャードを 1 つのシャードに組み合わせます。リシャーディングは、1 回のオペレーションでシャードに分割できる数と 1 回のオペレーションで結合できるシャードの数が 2 個以下に限られるという意味で、常にペアワイズです。リシャーディングオペレーションの対象となるシャードまたはシャードペアは、親シャードと呼ばれます。リシャーディングオペレーションを実行した結果のシャードまたはシャードペアは、子シャードと呼ばれます。

分割によりストリーム内のシャードの数が増え、したがってストリームのデータ容量は増えます。シャード単位で請求されるため、分割によりストリームのコストが増えます。同様に、マージによってストリーム内のシャードの数が減り、それに伴いストリームのデータ容量 (コスト) が減少します。

リシャーディングは、通常、プロデューサー (入力) アプリケーションやコンシューマー (取得) アプリケーションとは別の管理アプリケーションによって実行されます。このような管理アプリケーションは、Amazon が提供するメトリクス、 CloudWatch またはプロデューサーとコンシューマーから収集されたメトリクスに基づいて、ストリームの全体的なパフォーマンスをモニタリングします。管理アプリケーションには、コンシューマーまたはプロデューサーよりも広範な IAM 許可も必要になります。コンシューマーとプロデューサーは通常、リシャーディングに使用される API にアクセスする必要がないためです。Kinesis Data Streams の IAM 許可の詳細については、IAM を使用した Amazon Kinesis Data Streams リソースへのアクセスの制御 を参照してください。

リシャーディングの詳細については、Kinesis Data Streams で開いているシャードの数を変更するにはどうすればよいですか? を参照してください。

トピック

- リシャーディングのための戦略
- シャードの分割
- 2 つのシャードを結合する
- リシャーディング後

リシャーディングのための戦略

Amazon Kinesis Data Streams におけるリシャーディングの目的は、ストリームをデータの流量の変化に適応させることです。シャードを分割すると、ストリームの容量 (およびコスト) が増えます。シャードを結合すると、ストリームのコスト (および容量) が減ります。

リシャーディングへの 1 つのアプローチとして考えられるのは、ストリーム内のすべてのシャードを分割することです。これにより、ストリームの容量は倍増します。ただし、実際に必要になるよりも多くの容量が追加されるため、不要なコストが生じる可能性があります。

メトリクスを使用して、シャードがホットであるかコールドであるか、つまり、想定より過多なデータを受け取っているか、過少なデータを受け取っているかを判断できます。ホットシャードは分割して、それらのハッシュキーに対応した容量を増やすことができます。同様に、コールドシャードは結合して、未使用の容量をより有効に活用できます。

Kinesis Data Streams が公開する Amazon CloudWatch メトリクスからストリームのパフォーマンスデータを取得できます。ただし、ストリームについて独自のメトリックを収集することもできます。1 つのアプローチとして考えられるのは、データレコードのパーティションキーによって生成されたハッシュキー値をログに記録することです。ストリームにレコードを追加するときにパーティションキーを指定していることを思い出してください。

```
putRecordRequest.setPartitionKey( String.format( "myPartitionKey" ) );
```

Kinesis Data Streams では、MD5 を使用してパーティションキーからハッシュキーを計算します。レコードのパーティションキーを指定しているため、MD5 を使用してそのレコードのハッシュキー値を計算し、ログに記録できます。

また、データレコードが割り当てられているシャードの ID をログに記録することもできます。シャード ID は、`getShardId` メソッドによって返される `putRecordResults` オブジェクトおよび `putRecords` メソッドによって返される `putRecordResult` オブジェクトの `putRecord` メソッドを使用することによって利用できます。

```
String shardId = putRecordResult.getShardId();
```

シャード ID とハッシュキー値を使用すると、最も多いまたは少ないトラフィックを受け取っているシャードとハッシュキーを特定できます。その後、リシャーディングによりこれらのハッシュキーに対応した容量を増やすか減らすことができます。

シャードの分割

Amazon Kinesis Data Streams のシャードを分割するには、親シャードのハッシュキー値を子シャードに再配分する方法を指定する必要があります。データレコードをストリームに追加すると、レコードはハッシュキー値に基づいてシャードに割り当てられます。ハッシュキー値は、ストリームに追加するデータレコードに指定するパーティションキーの MD5 ハッシュです。パーティションキーが同じデータレコードはハッシュキー値も同じです。

指定したシャードに使用可能なハッシュキー値は、順序付けられた連続する正の整数で構成されます。ハッシュキーの一連の値は以下の式を使用して導き出します。

```
shard.getHashKeyRange().getStartingHashKey();
shard.getHashKeyRange().getEndingHashKey();
```

シャードを分割するときは、この一連の値を指定します。そのハッシュキー値とそれより上位のすべてのハッシュキー値は、いずれかの子シャードの配分されます。それより下位のすべてのハッシュキー値は、その他の子のシャードに配分されます。

以下のコードでは、子シャード間でハッシュキーを均等に再配分し、親シャードを半分に分割する基本的なシャード分割オペレーションを示します。これは、親シャードを分割する方法の 1 つに過ぎません。たとえば、親シャードの下位 1/3 のキーを 1 つの子シャードに配分し、上位 2/3 のキーをその他の子シャードに配分して、シャードを分割することもできます。ただし、多くアプリケーションに効果的なのは、シャードを半分に分割することです。

以下のコードでは、myStreamName にストリームの名前が格納され、オブジェクト変数 shard に分割するシャードが格納されるとします。新しい splitShardRequest オブジェクトをインスタンス化し、ストリーム名とシャード ID を設定することから始めます。

```
SplitShardRequest splitShardRequest = new SplitShardRequest();
splitShardRequest.setStreamName(myStreamName);
splitShardRequest.setShardToSplit(shard.getShardId());
```

シャード内の最小値と最大値の中間にあるハッシュキー値を決定します。これは、子シャードの開始ハッシュキー値になり、親シャードのハッシュキーの上位半分が含まれます。この値を setNewStartingHashKey メソッドで指定します。この値だけを指定する必要があります。この値より下位のハッシュキーは、分割によって作成されたその他の子シャードに、Kinesis Data Streams によって自動的に配分されます。最後のステップとして、Kinesis Data Streams クライアントで splitShard メソッドを呼び出します。

```
BigInteger startingHashKey = new
 BigInteger(shard.getHashKeyRange().getStartingHashKey());
BigInteger endingHashKey   = new
 BigInteger(shard.getHashKeyRange().getEndingHashKey());
String newStartingHashKey  = startingHashKey.add(endingHashKey).divide(new
 BigInteger("2")).toString();

splitShardRequest.setNewStartingHashKey(newStartingHashKey);
client.splitShard(splitShardRequest);
```

この方法の後の最初の手順は、ストリームが再度アクティブになるまで待機するに示されています。

2 つのシャードを結合する

シャードの結合オペレーションは、指定した 2 つのシャードを取得し、1 つシャードに組み合わせます。結合後、1 つの子シャードは 2 つの親シャードのすべてのハッシュキー値のデータを受け取ります。

シャードの隣接

2 つのシャードを結合するには、シャードが隣接している必要があります。2 つのシャードのハッシュキー範囲が途切れておらず連続している場合、2 つのシャードは隣接していると考えられます。たとえば、2 つのシャードがあり、1 つのハッシュキー範囲が 276〜381、もう 1 つのハッシュキー範囲が 382〜454 であるとします。この 2 つのシャードは 1 つのシャードに結合可能であり、結合した場合のハッシュキー範囲は 276〜454 となります。

別の例として 2 つのシャードがあり、1 つのハッシュキー範囲が 276〜381、もう 1 つのハッシュキー範囲が 455〜560 であるとします。この 2 つのシャードは結合できません。これらの間に 1 つ以上のシャード (ハッシュキー範囲が 382〜454) が介在している可能性があります。

ストリーム内のすべての OPEN シャードのセットは、グループとして、常に MD5 ハッシュキー値の範囲全体にまたがります。CLOSED など、シャード状態の詳細については、リシャーディング後のデータのルーティング、データの永続化、シャードの状態 を参照してください。

結合候補になるシャードを特定するには、CLOSED 状態にあるすべてのシャードを除外する必要があります。OPEN であり、CLOSED ではないシャードには、null の終了シーケンス番号があります。以下のコードを使用してシャードの終了シーケンス番号をテストできます。

```
if( null == shard.getSequenceNumberRange().getEndingSequenceNumber() )
{
  // Shard is OPEN, so it is a possible candidate to be merged.
}
```

CLOSED 状態のシャードを除外した後、各シャードでサポートされている最大ハッシュキー値で、残りのシャードを並べ替えます。以下のコード使用してこの値を取得できます。

```
shard.getHashKeyRange().getEndingHashKey();
```

このフィルタリングして並び替えたリストで 2 つシャードが隣接している場合、それらのシャードは結合できます。

結合オペレーションのコード

以下のコードでは、2 つシャードを結合しています。myStreamName には、ストリームの名前が格納され、オブジェクト変数 shard1 と shard2 には、結合する 2 つの隣接するシャードが格納されるとします。

結合オペレーションの場合、新しい mergeShardsRequest オブジェクトをインスタンス化することから始めます。setStreamName メソッドでストリーム名を指定します。次に、setShardToMerge と setAdjacentShardToMerge のメソッドを使用して、結合する 2 つのシャードを指定します。最後に、Kinesis Data Streams クライアントで mergeShards メソッドを呼び出して、このオペレーションを実行します。

```
MergeShardsRequest mergeShardsRequest = new MergeShardsRequest();
mergeShardsRequest.setStreamName(myStreamName);
mergeShardsRequest.setShardToMerge(shard1.getShardId());
mergeShardsRequest.setAdjacentShardToMerge(shard2.getShardId());
client.mergeShards(mergeShardsRequest);
```

この方法の後の最初の手順は、ストリームが再度アクティブになるまで待機するに示されています。

リシャーディング後

Amazon Kinesis Data Streams でリシャーディングの手順が終了し、通常のレコード処理を再開する前に必要な手順や検討事項があります。以下のセクションでは、これらについて説明します。

トピック

- ストリームが再度アクティブになるまで待機する
- リシャーディング後のデータのルーティング、データの永続化、シャードの状態

ストリームが再度アクティブになるまで待機する

リシャーディングオペレーションとして splitShard または mergeShards のいずれかを呼び出した後、ストリームが再びアクティブになるまで待機する必要があります。使用するコードは、ストリームの作成後にストリームがアクティブになるまで待機する場合のものと同じです。コードは次のとおりです。

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
```

```
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime )
{
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus =
 describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
   //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime )
{
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

リシャーディング後のデータのルーティング、データの永続化、シャードの状態

Kinesis Data Streams はリアルタイムのデータストリーミングサービスです。つまりアプリケーションでは、データがストリーム内のシャードに連続的に流れていることが前提になります。リシャーディングすると、親シャードに流れていたデータレコードは、データレコードのパーティションキーがマッピングされるハッシュキー値に基づいて、子シャードに流れるように再ルーティングされます。ただし、リシャーディング前に親シャードにあったデータレコードはすべて、それらのシャードに残ります。つまり、リシャーディング後に親シャードが失われることはありません。それらのシャードはリシャーディング前に格納されていたデータと共に保持されます。親シャードのデータレコードには、Kinesis Data Streams API の getShardIterator および getRecords オペレーションを使用するか、Kinesis Client Library を介してアクセスできます。

> **Note**
>
> データレコードは、現在の保持期間にストリームを追加した時間からアクセスできます。これは、その期間内のストリームのシャードの変更に関係なく当てはまります。ストリームの保持期間の詳細については、データ保持期間の変更を参照してください。

リシャーディングの過程で、親シャードは OPEN 状態から CLOSED 状態に、さらに EXPIRED 状態へと移行します。

- OPEN: リシャーディングオペレーションに先立って、親シャードは OPEN 状態にあります。つまり、データレコードはシャードに追加したり、シャードから取得したりできます。
- CLOSED: リシャーディングオペレーション後に、親シャードは CLOSED 状態に移行します。つまり、データレコードはシャードに追加されなくなります。このシャードに追加されることになっていたデータレコードは、子シャードに追加されるようになります。ただし、データレコードは引き続き、制限された時間内にシャードから取得できます。
- EXPIRED: ストリーム保持期間の有効期限が切れると、親シャードのすべてのデータレコードが期限切れとなり、アクセスできなくなります。この時点で、シャード自体は EXPIRED 状態に移行します。`getStreamDescription().getShards` を呼び出してストリーム内のシャードを列挙しても、返されるシャードのリストには 状態のシャードは含まれません。EXPIREDストリームの保持期間の詳細については、データ保持期間の変更を参照してください。

リシャーディング後、ストリームが再び ACTIVE 状態になるとすぐに、子シャードからのデータの読み取りを開始できます。ただし、リシャードの後に残っている親シャードには、リシャードの前にストリームに追加された、まだ読み取っていないデータが含まれている場合があります。親シャードからすべてのデータを読み取る前に、子シャードからデータを読み取った場合は、特定のハッシュキーが原因で、読み取ったデータがデータレコードのシーケンス番号に基づいた順序に並ばない可能性があります。したがって、データの順序が重要である場合は、リシャーディング後そのデータを使い切るまで、親シャードからのデータの読み取りを続行する必要があります。子シャードからのデータの読み取りは必ずその後で開始してください。`getRecordsResult.getNextShardIterator` が を返した場合は、親シャード内のすべてのデータを読み取ったということです。`null`Kinesis Client Library を使用してデータを読み取る場合、リシャードの発生後にデータが順番に受信されないことがあります。

データ保持期間の変更

Amazon Kinesis Data Streams は、データストリームのデータレコードの保持期間の変更をサポートしています。Kinesis data stream はデータレコードの順序付けられたシーケンスで、リアルタイムで書き込みと読み取りができることが前提となっています。したがって、データレコードはシャードに一時的に保存されます。レコードが追加されてからアクセスできなくなるまでの期間は、保持期間と呼ばれます。Kinesis data stream は、デフォルトでは 24 時間レコードを保持し、最大値は 8,760 時間 (365 日) です。

保持期間は、Kinesis Data Streams コンソールまたは IncreaseStreamRetentionPeriodおよび DecreaseStreamRetentionPeriodオペレーションを使用して更新できます。Kinesis Data Streams コンソールでは、複数のデータストリームの保持期間を同時に一括編集できます。IncreaseStreamRetentionPeriod オペレーションまたは Kinesis Data Streams コンソールを使用して、保持期間を最大 8760 時間 (365 日) まで延長できます。DecreaseStreamRetentionPeriod オペレーションまたは Kinesis Data Streams コンソールを使用して、保持期間を最低 24 時間に短縮できます。両方のオペレーションに対するリクエスト構文には、時間にストリーム名と保存期間が含まれます。最後に、 DescribeStreamオペレーションを呼び出すことで、ストリームの現在の保持期間を確認できます。

AWS CLIを使用して保持期間を変更する例を以下に示します。

```
aws kinesis increase-stream-retention-period --stream-name retentionPeriodDemo --
retention-period-hours 72
```

Kinesis Data Streams は、数分間延長した保持期間内で古い保持期間のアクセス停止を解除することができます。たとえば、保持期間を 24 時間から 48 時間に変更すると、23 時間 55 分前にストリームに追加されたレコードは、さらに 24 時間後まで使用できます。

Kinesis Data Streams は、保持期間が短縮されると、新しい保持期間よりも古いレコードをほぼ即座にアクセス不能にします。したがって、 DecreaseStreamRetentionPeriodオペレーションを呼び出すときは細心の注意を払ってください。

問題が発生した場合は、期限が切れる前にデータを読めるように保持期間を設定します。レコード処理ロジックの問題または長期間にわたるダウンストリームの依存関係など、あらゆる可能性を慎重に検討してください。データコンシューマーの回復時間に余裕が出るように、保持期間は慎重に設定します。保持期間 API オペレーションでは、この期間を事前に設定したり、オペレーションイベントにリアクティブに対応したりできます。

24 時間を超える保持期間を設定されたストリームに追加料金が適用されます。詳細については、「Amazon Kinesis Data Streams の料金」を参照してください。

Amazon Kinesis Data Streams のストリームのタグ付け

Amazon Kinesis Data Streams で作成したストリームに、独自のメタデータをタグ形式で割り当てることができます。タグは、ストリームに対して定義するキーと値のペアです。タグの使用は、AWS リソースの管理やデータ (請求データなど) の整理を行うシンプルかつ強力な方法です。

目次

タグの基本

Kinesis Data Streams コンソール、AWS CLI、または Kinesis Data Streams API を使用して、以下のタスクを実行します。

- ストリームにタグを追加する
- ストリームのタグを一覧表示する
- ストリームからタグを削除する

タグを使用すると、ストリームを分類できます。たとえば、目的、所有者、環境などに基づいてストリームを分類できます。タグごとにキーと値を定義するため、特定のニーズを満たすためのカテゴリのカスタムセットを作成できます。たとえば、所有者と、関連するアプリケーションに基づいてストリームを追跡するのに役立つタグのセットを定義できます。次にいくつかのタグの例を示します。

- プロジェクト: プロジェクト名
- 所有者: 名前
- 目的: 負荷テスト
- アプリケーション: アプリケーション名

- 環境: 本稼働

タグ付けを使用したコストの追跡

タグを使用して、AWS コストを分類して追跡できます。AWS リソース (ストリームなど) にタグを適用すると、AWS コスト配分レポートに、タグ別に集計された使用状況とコストが表示されます。自社のカテゴリ (たとえばコストセンター、アプリケーション名、所有者) を表すタグを適用すると、複数のサービスにわたってコストを分類することができます。詳細については、AWS Billing ユーザーガイドのコスト配分タグを使用したカスタム請求レポートを参照してください。

タグの制限

タグには次の制限があります。

基本制限

- リソース (ストリーム) あたりのタグの最大数は 50 です。
- タグのキーと値は大文字と小文字が区別されます。
- 削除されたストリームのタグを変更または編集することはできません。

タグキーの制限

- 各タグキーは一意である必要があります。既に使用されているキーを含むタグを追加すると、新しいタグで、既存のキーと値のペアが上書きされます。
- `aws:` は AWS が使用するように予約されているため、このプレフィックスを含むタグキーで開始することはできません。AWSではユーザーの代わりにこのプレフィックスで始まるタグを作成しますが、ユーザーはこれらのタグを編集または削除することはできません。
- タグキーの長さは 1 ~ 128 文字 (Unicode) にする必要があります。
- タグキーは、次の文字で構成する必要があります。Unicode 文字、数字、空白、特殊文字 (`_ . / = + - @`)。

タグ値の制限

- タグ値の長さは 0 ~ 255 文字 (Unicode) にする必要があります。
- タグ値は空白にすることができます。空白にしない場合は、次の文字で構成する必要があります。Unicode 文字、数字、空白、特殊文字 (`_ . / = + - @`)。

Kinesis Data Streams コンソールを使用したストリームのタグ付け

Kinesis Data Streams コンソールを使用してタグの追加、一覧表示、および削除を行うことができます。

ストリームのタグを表示するには

1. Kinesis Data Streams コンソールを開きます。ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
2. [ストリームリスト] ページで、ストリームを選択します。
3. [ストリームの詳細] ページで、[タグ] タブをクリックします。

ストリームにタグを追加するには

1. Kinesis Data Streams コンソールを開きます。ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
2. [ストリームリスト] ページで、ストリームを選択します。
3. [ストリームの詳細] ページで、[タグ] タブをクリックします。
4. [キー] フィールドにタグキーを指定し、オプションとして [値] フィールドにタグ値を指定した後で、[タグの追加] をクリックします。

   [タグの追加] ボタンが有効でない場合は、指定したタグキーまたはタグ値のいずれかがタグの制限を満たしていません。詳細については、タグの制限を参照してください。
5. [タグ] タブのリストに新しいタグを表示するには、更新アイコンをクリックします。

ストリームからタグを削除するには

1. Kinesis Data Streams コンソールを開きます。ナビゲーションバーで、リージョンセレクターを展開し、リージョンを選択します。
2. [Stream List] ページで、ストリームを選択します。
3. [ストリームの詳細] ページで、[タグ] タブをクリックし、タグの [削除] アイコンをクリックします。
4. [タグの削除] ダイアログボックスで、[はい、削除する] をクリックします。

AWS CLI を使用したストリームのタグ付け

AWS CLI を使用してタグの追加、一覧表示、および削除を行うことができます 例については、次のドキュメントを参照してください。

add-tags-to-stream

指定したストリームのタグを追加または更新します。

list-tags-for-stream

指定したストリームのタグを一覧表示します。

remove-tags-from-stream

指定したストリームからタグを削除します。

Kinesis Data Streams API を使用したストリームのタグ付け

Kinesis Data Streams API を使用してタグの追加、一覧表示、および削除を行うことができます。例については、次のドキュメントを参照してください。

AddTagsToStream

指定したストリームのタグを追加または更新します。

ListTagsForStream

指定したストリームのタグを一覧表示します。

RemoveTagsFromStream

指定したストリームからタグを削除します。

Amazon Kinesis Data Streams へのデータの書き込み

プロデューサーは、Amazon Kinesis Data Streams にデータを書き込むアプリケーションです。Kinesis Data Streams のプロデューサーは、AWS SDK for Java および Kinesis Producer Library を使用して構築できます。

Kinesis Data Streams を初めて利用する場合は、Amazon Kinesis Data Streams とはおよびAmazon Kinesis Data Streams の開始方法で説明されている概念と用語について理解することから始めてください。

> **Important**
>
> Kinesis Data Streams は、データストリームのデータレコードの保持期間の変更をサポートしています。詳細については、データ保持期間の変更を参照してください。

ストリームにデータを送信するには、ストリームの名前、パーティションキー、ストリームに追加するデータ BLOB を指定する必要があります。パーティションキーは、データレコードが追加されるストリーム内のシャードを決定するために使用されます。

シャード内のすべてのデータは、そのシャードを処理する同じワーカーに送信されます。使用するパーティションキーはアプリケーションのロジックによって異なります。パーティションキーの数は、通常、シャードカウントよりかなり大きくする必要があります。これは、データレコードを特定のシャードにマッピングする方法を決定するために、パーティションキーが使用されるからです。十分なパーティションキーがある場合、ストリーム内のシャードに均等にデータを分散することができます。

目次

Amazon Kinesis Producer Library を使用したプロデューサーの開発

Amazon Kinesis Data Streams プロデューサーは、ユーザーデータレコードを Kinesis data stream に配置する (データの取り込みとも呼ばれます) アプリケーションです。Kinesis Producer Library (KPL) を使用すると、プロデューサーアプリケーションの開発が簡素化され、デベロッパーは Kinesis Data Streams に対する優れた書き込みスループットを実現できます。

Amazon で KPL をモニタリングできます CloudWatch。詳細については、「Amazon による Kinesis プロデューサーライブラリのモニタリング CloudWatch」を参照してください。

コンテンツ

> **Note**
>
> KPL は、最新バージョンにアップグレードすることが推奨されます。KPL は新しいリリースに伴って定期的に更新されています。これには、最新の依存関係パッチ、セキュリティパッチ、バグ修正、および下位互換性のある新機能が含まれます。詳細については、「https://github.com/awslabs/amazon-kinesis-producer/releases/」を参照してください。

KPL の役割

KPL は easy-to-use、Kinesis データストリームへの書き込みに役立つ、高度に設定可能な ライブラリです。これは、プロデューサーアプリケーションのコードと Kinesis Data Streams API アクション間の仲介として機能します。KPL は次の主要なタスクを実行します。

- 自動的で設定可能な再試行メカニズムにより 1 つ以上の Kinesis Data Streams へ書き込む
- レコードを収集し、`PutRecords` を使用して、リクエストごとに複数シャードへ複数レコードを書き込む
- ユーザーレコードを集約し、ペイロードサイズを増加させ、スループットを改善する
- コンシューマーで Kinesis Client Library (KCL) とシームレスに統合して、バッチ処理されたレコードを集約解除する
- プロデューサーのパフォーマンスを可視化するために、ユーザーに代わって Amazon CloudWatch メトリクスを送信します。

KPL は AWS SDK で使用できる Kinesis Data Streams API とは異なることに注意してください。Kinesis Data Streams API では Kinesis Data Streams の多くの機能 (ストリームの作成、リシャーディング、レコードの入力と取得など) を管理できます。KPL はデータの取り込みに特化した抽象化レイヤーを提供します。Kinesis Data Streams API の詳細については、Amazon Kinesis API リファレンスを参照してください。

KPL を使用するメリット

Kinesis Data Streams プロデューサーの開発に KPL を使用する主な利点を以下に示します。

KPL は、同期または非同期のユースケースで使用できます。同期動作を使用する特別な理由がないかぎり、非同期インターフェイスの優れたパフォーマンスを使用することを推奨します。これら 2 つのユースケースの詳細とコード例については、KPL を使用した Kinesis Data Stream への書き込みを参照してください。

パフォーマンスのメリット

KPL は、高性能のプロデューサーの構築に役立ちます。Amazon EC2 インスタンスをプロキシとして使用し、100 バイトのイベントを数百または数千の低電力デバイスから収集して、レコードを Kinesis Data Streams に書き込む場合を考えてみます。これらの EC2 インスタンスはそれぞれ、毎秒数千イベントをデータストリームに書き込む必要があります。必要なスループットを実現するには、お客様の側で、再試行ロジックとレコード集約解除に加え、バッチ処理やマルチス

レッドなどの複雑なロジックをプロデューサーに実装する必要があります。KPL が、これらのタスクをすべて実行します。

コンシューマー側の使いやすさ

コンシューマー側のデベロッパーが Java で KCL を使用する場合、追加作業なしで KPL が統合されます。KCL で、複数の KPL ユーザーレコードで構成されている集約された Kinesis Data Streams レコードを取得するときは、自動的に KPL が呼び出され、個々のユーザーレコードが抽出され、ユーザーに返されます。

KCL を使用せずに API オペレーション `GetRecords` を直接使用するコンシューマー側のデベロッパーの場合、KPL Java ライブラリを使用して個々のユーザーレコードを抽出して、これらのレコードをユーザーに返すことができます。

プロデューサーのモニタリング

Amazon と KPL を使用して、Kinesis Data Streams プロデューサーを収集、モニタリング CloudWatch 、分析できます。KPL は、 CloudWatch ユーザーに代わってスループット、エラー、およびその他のメトリクスを に出力し、ストリーム、シャード、またはプロデューサーレベルでモニタリングするように設定できます。

非同期アーキテクチャ

KPL は、レコードを Kinesis Data Streams に送信する前にそれらのレコードをバッファ処理する場合があるため、実行を続行する前にレコードがサーバーに到着したことを確認するために、発信者アプリケーションを強制的にブロックし待機させることはしません。レコードを KPL に配置する呼び出しは、必ずすぐに処理が戻り、レコードの送信やサーバーからの応答の受信を待ちません。代わりに、レコードを Kinesis Data Streams に送信した結果を後で受信するための `Future` オブジェクトが作成されます。これは AWS SDK の非同期クライアントと同じ動作です。

KPL の使用が適さない場合

KPL では、ライブラリ内で最大 `RecordMaxBufferedTime` まで追加の処理遅延が生じる場合があります (ユーザーが設定可能)。`RecordMaxBufferedTime` の値が大きいほど、パッキング効率とパフォーマンスが向上します。この追加的な遅延を許容できないアプリケーションは、 AWS SDK を直接使用することが必要になる場合があります。Kinesis Data Streams で AWS SDK を使用する方法の詳細については、「」を参照してくださいAmazon Kinesis Data Streams API と AWS SDK for Java を使用したプロデューサーの開発。`RecordMaxBufferedTime` やその他のユーザー設定可能な KPLのプロパティの詳細については、Kinesis Producer Library の設定を参照してください。

KPL のインストール

Amazon では、macOS、Windows、最新の Linux ディストリビューション向けに C++ Kinesis Producer Library (KPL) のビルド済みバイナリを提供しています (サポートされているプラットフォームの詳細については、次のセクションを参照してください)。これらのバイナリは、Java の .jar ファイルの一部としてパッケージ化されており、Maven を使用してパッケージをインストールする場合、自動的に呼び出され、使用されます。KPL と KCL の最新バージョンを確認するには、次の Maven 検索リンクをご利用ください。

- KPL
- KCL

Linux のバイナリは、GNU コンパイラコレクション (GCC) でコンパイルされ、Linux の libstdc++ に静的にリンクされています。これらのバイナリは、glibc バージョン 2.5 以降を含むすべての 64 ビット Linux ディストリビューションで動作することが推定されています。

古い Linux ディストリビューションのユーザーは、 のソースとともに提供されるビルド手順を使用して KPL を構築できます GitHub。から KPL をダウンロードするには GitHub、「Kinesis Producer Library」を参照してください。

Kinesis Producer Library の Amazon Trust Services (ATS) 証明書への移行

2018 年 2 月 9 日の午前 9:00 (太平洋標準時) に、Amazon Kinesis Data Streams は ATS 証明書をインストールしました。Kinesis Producer Library (KPL) を使用して、Kinesis Data Streams にレコードを継続して書き込むには、KPL のインストールをバージョン 0.12.6 以降にアップグレードする必要があります。この変更はすべての AWS リージョンに影響します。

ATS への移行については、AWS「独自の認証機関への の移動を準備する方法」を参照してください。

問題が発生し、技術サポートが必要な場合は、 AWS サポートセンターでサポートケースを作成してください。

KPL でサポートされるプラットフォーム

Kinesis Producer Library (KPL) は、C++ で書かれており、メインユーザープロセスの子プロセスとして実行されます。プリコンパイルされている 64 ビットのネイティブバイナリは、Java ベースにバンドルされており、Java wrapper によって管理されます。

次のオペレーティングシステムでは、追加ライブラリをインストールすることなく Java のパッケージを実行できます。

- カーネルバージョン 2.6.18 (2006 年 9 月) の Linux ディストリビューション以降
- Apple OS X 10.9 以降
- Windows Server 2008 以降

> **Important**
>
> Windows Server 2008 以降は、バージョン 0.14.0 までのすべての KPL バージョンでサポートされています。
> Windows プラットフォームは、KPL バージョン 0.14.0 以降ではサポートされていません。

KPL は、64 ビット版のみであることに注意してください。

ソースコード

KPL のインストールで提供されるバイナリがお客様の環境に適さない場合は、KPL のコアが C++ のモジュールとして書き込まれます。C++ モジュールと Java インターフェイスのソースコードは Amazon Public License の下でリリースされ、Kinesis Producer Library GitHub の で入手できます。KPL は、最近の規格に準拠した C++ コンパイラと JRE を使用できるすべてのプラットフォームで使用できますが、Amazon では、サポートされるプラットフォームの一覧にないプラットフォームを正式にはサポートしません。

KPL の主要な概念

以下のセクションでは、Kinesis Producer Library (KPL) を理解し、その利点を引き出すために必要な概念と用語について説明します。

トピック

- レコード
- バッチ処理
- 集計
- 収集

レコード

このガイドでは、KPL ユーザーレコードと Kinesis Data Streams レコードを区別します。修飾語を付けずにレコードという用語を使用する場合は、KPL ユーザーレコードを意味します。Kinesis Data Streams レコードを意味するときは、明示的に Kinesis Data Streams レコードと表現します。

KPL ユーザーレコードは、ユーザーにとって特定の意味のあるデータの BLOB です。たとえば、ウェブサイトの UI イベントまたはウェブサーバーのログエントリを表す JSON BLOB がそれに該当します。

Kinesis Data Streams レコードは、Kinesis Data Streams サービス API で定義される`Record` データ構造のインスタンスです。これには、パーティションキー、シーケンス番号、データの BLOB が含まれています。

バッチ処理

バッチ処理は、各項目に対して単一のアクションを繰り返し実行する代わりに、複数の項目に対してそのアクションを実行することを意味します。

ここでは、項目はレコードに対応し、アクションはレコードを Kinesis Data Streams に送信することに対応します。バッチ処理を使用しない場合、各レコードを別々の Kinesis Data Streams レコードに配置し、それぞれを Kinesis Data Streams に送信するたびに HTTP リクエストを実行します。バッチ処理では、各 HTTP リクエストにより、1 つではなく複数のレコードを処理できます。

KPL では、2 種類のバッチ処理がサポートされます。

- 集約 - 複数のレコードを単一の Kinesis Data Streams レコードに保存します。
- 収集 - API オペレーション `PutRecords` を使用して、Kinesis Data Streams 内の 1 つ以上のシャードに複数の Kinesis Data Streams レコードを送信します。

2 種類の KPL バッチ処理は、共存できるように設計されており、互いに独立して有効または無効にできます。デフォルトでは、どちらも有効です。

集計

集約は、複数レコードを 1 つの Kinesis Data Streams レコードに保存することを意味します。集約を使用すると、API コールごとに送信されるレコード数を増やすことができ、効率的にプロデューサーのスループットを高めることができます。

Kinesis Data Streams シャードは、1 秒あたり最大で 1,000 レコードまたは 1 MB のスループットをサポートします。1 秒あたりの Kinesis Data Streams レコードの制限により、お客様のレコードは 1 KB 未満に制限されます。レコードの集約を使用すると、複数のレコードを単一の Kinesis Data Streams レコードに結合できます。そのため、お客様はシャードあたりのスループットを改善することができます。

リージョンが us-east-1 の 1 つのシャードで、1 つが 512 バイトのレコードを 1 秒あたり 1,000 レコードの一定割合で処理する場合を考えます。KPL 集約を使用すると、1,000 レコードを 10 Kinesis Data Streams レコードに詰めることができ、RPS を 10 に減らすことができます (それぞれ 50 KB)。

収集

収集は、各 Kinesis Data Streams レコードをそれぞれの HTTP リクエストで送信するのではなく、複数の Kinesis Data Streams レコードをバッチ処理し、API オペレーション `PutRecords` を呼び出して単一の HTTP リクエストでそれらを送信することを意味します。

これにより、個別の HTTP リクエストを多数実行するオーバーヘッドが減るため、収集を使用しない場合に比べスループットが向上します。実際、`PutRecords` 自体が、この目的のために設計されています。

収集は、Kinesis Data Streams レコードのグループを使用している点で集約と異なります。収集された Kinesis Data Streams レコードには、ユーザーの複数のレコードをさらに含めることができます。この関係は、次のように図示できます。

```
record 0 --|
record 1   |        [ Aggregation ]
    ...    |--> Amazon Kinesis record 0 --|
    ...    |                              |
record A --|                              |
                                           |
    ...                  ...               |
                                           |
record K --|                              |
record L   |                              |      [ Collection ]
    ...    |--> Amazon Kinesis record C --|--> PutRecords Request
    ...    |                              |
record S --|                              |
                                           |
    ...                  ...               |
                                           |
record AA--|                              |
```

```
record BB  |                                |
    ...    |--> Amazon Kinesis record M --|
    ...    |
record ZZ--|
```

KPL とプロデューサーコードの統合

Kinesis Producer Library (KPL) は独立したプロセスで実行され、IPC を使用して親ユーザープロセスと通信します。このアーキテクチャは、マイクロサービスと呼ばれる場合があり、次の 2 つの主な理由からこれが選択されます。

1) KPL がクラッシュしても、ユーザープロセスはクラッシュしません

プロセスには Kinesis Data Streams と無関係なタスクが含まれている場合があり、KPL がクラッシュしてもオペレーションを続行できます。また、親ユーザープロセスが KPL を再起動し、完全に機能する状態に復旧することもできます (この機能は、正式なラッパーに含まれています)。

メトリクスを Kinesis Data Streams に送信するウェブサーバーがその例です。このサーバーは、Kinesis Data Streams 部分が動作を停止してもページの提供を続行できます。そのため、KPL のバグが原因でサーバー全体がクラッシュすると、不要なサービス停止が発生します。

2) 任意のクライアントをサポートできます

正式にサポートされている言語以外の言語を使用するお客様もいます。これらのお客様も KPL を簡単に使用できます。

推奨される使用状況

使用状況の異なるユーザーに推奨される設定を次の表に示します。この表を参考に、KPL を使用できるかどうか、どのように使用できるかを判断できます。集約が有効な場合、コンシューマー側で集約解除を使用してレコードを抽出する必要があることにも注意してください。

プロデューサー側の言語	コンシューマー側の言語	KCL バージョン	チェックポイントロジック	KPL の使用可否	注意
Java 以外	*	*	*	いいえ	該当なし
Java	Java	Java SDK を直接使用	該当なし	はい	集約を使用する場合、GetRecord

プロデューサー側の言語	コンシューマー側の言語	KCL バージョン	チェックポイントロジック	KPL の使用可否	注意
					s を呼び出した後に、提供された集約解除ライブラリを使用する必要があります。
Java	Java 以外	SDK を直接使用	該当なし	はい	集約を無効にする必要があります。
Java	Java	1.3.x	該当なし	はい	集約を無効にする必要があります。
Java	Java	1.4.x	引数なしでチェックポイントを呼び出す	はい	なし
Java	Java	1.4.x	明示的なシーケンス番号を使用してチェックポイントを呼び出す	はい	集約を無効にするかコードを変更し、チェックポイント作成用の拡張されたシーケンス番号を使用します。
Java	Java 以外	1.3.x + 複数言語デーモン + 言語固有のラッパー	該当なし	はい	集約を無効にする必要があります。

KPL を使用した Kinesis Data Stream への書き込み

以下のセクションでは、最もシンプルな最低限のプロデューサーから完全に非同期なコードまで順にサンプルコードを示します。

最低限のプロデューサーコード

次のコードは、最小限の機能するプロデューサーを書くために必要なものがすべて含まれています。Kinesis Producer Library (KPL) ユーザーレコードはバックグラウンドで処理されます。

```
// KinesisProducer gets credentials automatically like
// DefaultAWSCredentialsProviderChain.
// It also gets region automatically from the EC2 metadata service.
KinesisProducer kinesis = new KinesisProducer();
// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}
// Do other stuff ...
```

結果に対する同期的な応答

前のコード例では、 ユーザーレコードが成功したかどうかをチェックしませんでした。KPL は、失敗に対処するために必要な再試行を実行します。ただし、結果を確認する必要がある場合は、次の例 (分かりやすくするため前の例を使用しています) のように、addUserRecord から返される Future オブジェクトを使用して結果を確認します。

```
KinesisProducer kinesis = new KinesisProducer();

// Put some records and save the Futures
List<Future<UserRecordResult>> putFutures = new
 LinkedList<Future<UserRecordResult>>();
for (int i = 0; i < 100; i++) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    putFutures.add(
        kinesis.addUserRecord("myStream", "myPartitionKey", data));
}
```

```
// Wait for puts to finish and check the results
for (Future<UserRecordResult> f : putFutures) {
    UserRecordResult result = f.get(); // this does block
    if (result.isSuccessful()) {
        System.out.println("Put record into shard " +
                           result.getShardId());
    } else {
        for (Attempt attempt : result.getAttempts()) {
            // Analyze and respond to the failure
        }
    }
}
```

結果に対する非同期的な応答

前の例では、get() オブジェクトに対して Future を呼び出しているため、実行がブロックされます。実行のブロックを避ける必要がある場合には、次の例に示すように非同期コールバックを使用できます。

```
KinesisProducer kinesis = new KinesisProducer();

FutureCallback<UserRecordResult> myCallback = new FutureCallback<UserRecordResult>() {

    @Override public void onFailure(Throwable t) {
        /* Analyze and respond to the failure  */
    };
    @Override public void onSuccess(UserRecordResult result) {
        /* Respond to the success */
    };
};

for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    ListenableFuture<UserRecordResult> f = kinesis.addUserRecord("myStream",
 "myPartitionKey", data);
    // If the Future is complete by the time we call addCallback, the callback will be
 invoked immediately.
    Futures.addCallback(f, myCallback);
}
```

Kinesis Producer Library の設定

デフォルト設定のままで、ほとんどのユースケースに問題なく使用できますが、デフォルト設定の一部を変更することで、ニーズに合わせて KinesisProducer の動作を調整することができます。それには、KinesisProducerConfiguration クラスのインスタンスを KinesisProducer コンストラクタに渡します。たとえば、次のようにします。

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration()
        .setRecordMaxBufferedTime(3000)
        .setMaxConnections(1)
        .setRequestTimeout(60000)
        .setRegion("us-west-1");

final KinesisProducer kinesisProducer = new KinesisProducer(config);
```

プロパティファイルから設定をロードすることもできます。

```
KinesisProducerConfiguration config =
 KinesisProducerConfiguration.fromPropertiesFile("default_config.properties");
```

ユーザープロセスがアクセスできる任意のパスとファイル名に置き換えることができます。さらに、このようにして作成した KinesisProducerConfiguration インスタンスに対して設定メソッドを呼び出して、設定をカスタマイズできます。

プロパティファイルは、 で名前を使用してパラメータを指定する必要があります PascalCase。その名前は、KinesisProducerConfiguration クラスの設定メソッドで使用されるものと一致します。例:

```
RecordMaxBufferedTime = 100
MaxConnections = 4
RequestTimeout = 6000
Region = us-west-1
```

設定パラメータの使用ルールと値の制限の詳細については、「」の「サンプル設定プロパティファイル GitHub」を参照してください。

KinesisProducer の初期化後に、使用した KinesisProducerConfiguration インスタンスを変更しても何の変化もないことに注意してください。現在、KinesisProducer は動的設定をサポートしていません。

コンシューマーの集約解除

KCL は、リリース 1.4.0 から KPL ユーザーレコードの自動集計解除をサポートしています。以前のバージョンの KCL で書かれたコンシューマーアプリケーションのコードは、KCL を更新した後、コードを何も修正せずにコンパイルできます。ただし、プロデューサー側で KPL の集約を使用している場合、チェックポイントが多少関係してきます。集約されたレコード内のすべてのサブレコードは同じシーケンス番号を持っているため、サブレコード間の区別が必要な場合、チェックポイントを使用して追加のデータを保存する必要があります。この追加データは、サブシーケンス番号と呼ばれます。

以前のバージョンの KCL からの移行

集約とともにチェックポイントを作成する既存の呼び出しを変更する必要はありません。Kinesis Data Streams に保存されているすべてのレコードを正しく取得できることが保証されています。以下で説明する特定のユースケースをサポートするために、現在 KCL には、2 つの新しいチェックポイントオペレーションが用意されています。

既存のコードが KPL サポート以前の KCL 用に書かれていて、チェックポイントオペレーションが引数なしで呼び出される場合、そのコードの動作は、バッチ内にある最後の KPL ユーザーレコードのシーケンス番号に対するチェックポイントの作成と同等です。シーケンス番号文字列を使用してチェックポイントオペレーションを呼び出す場合は、暗黙的なサブシーケンス番号 0 (ゼロ) を伴う、バッチの指定されたシーケンス番号に対するチェックポイントの作成と同等です。

引数なしで新しいKCL チェックポイントオペレーション `checkpoint()` を呼び出すことは、暗黙的なサブシーケンス番号 0 (ゼロ) を伴う、バッチ内の最後の `Record` 呼び出しのシーケンス番号に対するチェックポイントの作成と意味的に同等です。

新しい KCL チェックポイントオペレーション `checkpoint(Record record)` を呼び出すことは、暗黙的なサブシーケンス番号 0 (ゼロ) を伴う、指定された `Record` のシーケンス番号に対するチェックポイントの作成と意味的に同等です。`Record` 呼び出しが実際には `UserRecord` である場合、`UserRecord` のシーケンス番号とサブシーケンス番号にチェックポイントが作成されます。

新しい KCL チェックポイントオペレーション `checkpoint(String sequenceNumber, long subSequenceNumber)` を呼び出すと、指定されたシーケンス番号とサブシーケンス番号に明示的にチェックポイントが作成されます。

いずれの場合も、チェックポイントが Amazon DynamoDB チェックポイントテーブルに保存された後は、アプリケーションがクラッシュして再起動した場合、KCL により、レコードの取得が正常に再開されます。さらにレコードがシーケンス内に含まれている場合は、最後にチェックポイントが作

成されたシーケンス番号が付けられているレコード内の次のサブシーケンス番号のレコードから取得が開始されます。前のシーケンス番号のレコードにある最後のサブシーケンス番号が、最新のチェックポイントに含まれている場合、その次のシーケンス番号が付けられているレコードから取得が開始されます。

次のセクションでは、レコードのスキップや重複を避けるために必要な、コンシューマーのシーケンスとサブシーケンスのチェックポイントの詳細について説明します。コンシューマーのレコード処理を停止し再起動するときに、レコードのスキップや重複が重要でない場合は、変更せずに既存のコードを実行してかまいません。

KPL の集約解除のための KCL の拡張

すでに説明したように、KPL の集約解除ではサブシーケンスチェックポイントを使用できます。サブシーケンスチェックポイントを使いやすくするために、UserRecord クラスが KCL に追加されています。

```
public class UserRecord extends Record {
    public long getSubSequenceNumber() {
    /* ... */
    }
    @Override
    public int hashCode() {
    /* contract-satisfying implementation */
    }
    @Override
    public boolean equals(Object obj) {
    /* contract-satisfying implementation */
    }
}
```

このクラスは、現在 Record の代わりに使用されています。これは Record のサブクラスであるため、既存のコードは影響を受けません。UserRecord クラスは、実際のサブレコードと通常の集約されていないレコードの両方を表します。集約されていないレコードは、サブレコードを 1 つだけ含む集約されたレコードと考えることができます。

さらに、2 つの新しいオペレーションが IRecordProcessorCheckpointer に追加されています。

```
public void checkpoint(Record record);
public void checkpoint(String sequenceNumber, long subSequenceNumber);
```

サブシーケンス番号チェックポイントの使用を開始するには、次の変更を行います。次のフォームコードを変更します。

```
checkpointer.checkpoint(record.getSequenceNumber());
```

新しいフォームコードは次のようになります。

```
checkpointer.checkpoint(record);
```

サブシーケンスチェックポイントでは、checkpoint(Record record) フォームを使用することをお勧めします。ただし、チェックポイントの作成で使用する文字列にすでに sequenceNumbers を保存している場合は、次の例に示すように、subSequenceNumber も保存する必要があります。

```
String sequenceNumber = record.getSequenceNumber();
long subSequenceNumber = ((UserRecord) record).getSubSequenceNumber();  // ... do other
 processing
checkpointer.checkpoint(sequenceNumber, subSequenceNumber);
```

この実装では内部で Record を必ず使用するため、UserRecord から UserRecord へのキャストは必ず成功します。シーケンス番号の計算を実行する必要がない場合、この方法はお勧めしません。

KPL ユーザーレコードの処理中に、CL は、サブシーケンス番号を Amazon DynamoDB に各行の追加フィールドとして書き込みます。以前のバージョンの KCL では、チェックポイントを再開するときに AFTER_SEQUENCE_NUMBER を使用してレコードを取得していました。KPL サポートを含む現在の KCL では、代わりに AT_SEQUENCE_NUMBER を使用します。チェックポイントが作成されたシーケンス番号のレコードを取得するとき、チェックポイントが作成されたサブシーケンス番号がチェックされ、サブレコードが必要に応じて削除されます (最後のサブレコードにチェックポイントが作成されている場合、すべてのサブレコードが削除されます)。ここでも、集約されていないレコードは、単一のサブレコードを含む集約されたレコードと考えることができ、集約されたレコードと集約されていないレコードの両方で同じアルゴリズムを使用できます。

を直接使用する GetRecords

KCL の使用を選択せずに、API オペレーション GetRecords を直接呼び出して Kinesis Data Streams レコードを取得することもできます。これらの取得したレコードを元の KPL ユーザーレコードに解凍するには、UserRecord.java にある次の静的なオペレーションの 1 つを呼び出します。

```
public static List<Record> deaggregate(List<Record> records)
```

```
public static List<UserRecord> deaggregate(List<UserRecord> records, BigInteger
 startingHashKey, BigInteger endingHashKey)
```

最初のオペレーションでは、startingHashKey のデフォルト値 0 (ゼロ) と endingHashKey のデフォルト値 2^128 -1 を使用します。

これらの各オペレーションは、Kinesis Data Streams レコードの指定されたリストを KPL ユーザーレコードのリストに集約解除します。KPL ユーザーレコードの明示的なハッシュキーまたはパーティションキーが startingHashKey と endingHashKey の範囲 (境界を含む) 外にある場合、これらのユーザーレコードは、返されるレコードのリストから破棄されます。

Firehose での KPL の使用

Kinesis Producer Library (KPL) を使用して Kinesis データストリームにデータを書き込む場合、集約を使用してその Kinesis データストリームに書き込むレコードを結合できます。その後、そのデータストリームを Firehose 配信ストリームのソースとして使用すると、Firehose はレコードを宛先に配信する前にレコードの集約を解除します。データを変換するように配信ストリームを設定すると、Firehose はレコードを に配信する前にレコードの集約を解除します AWS Lambda。詳細については、「Writing to Amazon Firehose Using Kinesis Data Streams」を参照してください。

AWS Glue スキーマレジストリでの KPL の使用

Kinesis データストリームを AWS Glue スキーマレジストリと統合できます。 AWS Glue スキーマレジストリを使用すると、スキーマを一元的に検出、制御、および進化させながら、生成されたデータが登録されたスキーマによって継続的に検証されるようにできます。スキーマは、データレコードの構造と形式を定義します。スキーマは、信頼性の高いデータの公開、利用、または保存のための仕様をバージョニングしたものです。 AWS Glue スキーマレジストリを使用すると、ストリーミングアプリケーション内の end-to-end データ品質とデータガバナンスを向上させることができます。詳細については、AWS Glue スキーマレジストリを参照してください。この統合を設定する方法の 1 つは、Java で KPL および Kinesis Client Library (KCL) ライブラリを使用することです。

> **Important**
>
> 現在、Kinesis Data Streams と AWS Glue スキーマレジストリ統合は、Java で実装された KPL プロデューサーを使用する Kinesis データストリームでのみサポートされています。多言語サポートは提供されていません。

KPL を使用して Kinesis Data Streams とスキーマレジストリの統合を設定する方法の詳細については、ユースケース: Amazon Kinesis Data Streams と AWS Glue スキーマレジストリの統合の「KPL/KCL ライブラリを使用したデータの操作」セクションを参照してください。

KPL プロキシ設定

インターネットに直接接続できないアプリケーションでは、すべての AWS SDK クライアントが HTTP または HTTPS プロキシの使用をサポートします。一般的なエンタープライズ環境では、すべてのアウトバウンドネットワークトラフィックがプロキシサーバーを経由する必要があります。アプリケーションが Kinesis Producer Library (KPL) を使用してプロキシサーバーを使用する AWS 環境でデータを収集して に送信する場合、アプリケーションには KPL プロキシ設定が必要です。KPL は、 AWS Kinesis SDK 上に構築された高レベルのライブラリです。これは、ネイティブプロセスとラッパーに分割されています。ネイティブプロセスがレコードの処理ジョブと送信ジョブのすべてを実行する一方で、ラッパーはネイティブプロセスの管理と、ネイティブプロセスとの通信を実行します。詳細については、「Implementing Efficient and Reliable Producers with the Amazon Kinesis Producer Library」を参照してください。

ラッパーは Java で記述され、ネイティブプロセスは Kinesis SDK を使用して C++ で記述されます。KPL バージョン 0.14.7 以降では、すべてのプロキシ設定をネイティブプロセスに渡すことができる、Java ラッパー内のプロキシ設定がサポートされるようになりました。詳細については、「https://github.com/awslabs/amazon-kinesis-producer/releases/tag/v0.14.7」を参照してください。

KPL アプリケーションへのプロキシ設定の追加には、以下のコードを使用できます。

```
KinesisProducerConfiguration configuration = new KinesisProducerConfiguration();
// Next 4 lines used to configure proxy
configuration.setProxyHost("10.0.0.0"); // required
configuration.setProxyPort(3128); // default port is set to 443
configuration.setProxyUserName("username"); // no default
configuration.setProxyPassword("password"); // no default

KinesisProducer kinesisProducer = new KinesisProducer(configuration);
```

Amazon Kinesis Data Streams API と AWS SDK for Java を使用したプロデューサーの開発

Amazon Kinesis Data Streams API と AWS SDK for Java を使用したプロデューサーの開発 Kinesis Data Streams を初めて利用する場合は、Amazon Kinesis Data Streams とはおよびAmazon Kinesis Data Streams の開始方法で説明されている概念と用語について理解することから始めてください。

以下の例では、Kinesis Data Streams API について説明し、AWS SDK for Java を使用してストリームにデータを追加 (入力) します。ただし、ほとんどのユースケースでは、Kinesis Data Streams KPL ライブラリを使用します。詳細については、Amazon Kinesis Producer Library を使用したプロデューサーの開発を参照してください。

この章で紹介する Java サンプルコードは、基本的な Kinesis Data Streams API オペレーションを実行する方法を示しており、オペレーションタイプ別に論理的に分割されています。これらのサンプルは、すべての例外を確認しているわけではなく、すべてのセキュリティやパフォーマンスの側面を考慮しているわけでもない点で、本稼働環境に使用できるコードを表すものではありません。また、他のプログラミング言語を使用して Kinesis Data Streams API を呼び出すこともできます。すべての利用可能な AWS SDK の詳細については、Amazon Web Services を使用した開発の開始を参照してください。

各タスクには前提条件があります。たとえば、ストリームを作成するまではストリームにデータを追加できず、ストリームを作成するにはクライアントを作成する必要があります。詳細については、ストリームの作成と管理を参照してください。

トピック

- ストリームへのデータの追加
- AWS Glue スキーマレジストリを使用してデータと相互作用する

ストリームへのデータの追加

ストリームを作成したら、レコードの形式でストリームにデータを追加できます。レコードはデータ BLOB の形式で処理するデータを格納するデータ構造です。データをレコードに保存した後、Kinesis Data Streams ではいずれの方法でもデータが検査、解釈、または変更されることはありません。各レコードにはシーケンス番号とパーティションキーも関連付けられます。

Kinesis Data Streams API には、ストリームにデータを追加するオペレーションとして `PutRecords` と `PutRecord` の 2 つの異なるオペレーションがあります。`PutRecords` オペレーションは HTTP リクエストごとストリームに複数のレコードを送信し、単数形の `PutRecord` オペ

レーションは一度に 1 つずつストリームにレコードを送信します (各レコードについて個別の HTTP リクエストが必要です)。データプロデューサーあたりのスループットが向上するため、ほとんどのアプリケーションでは PutRecords を使用してください。これらの各オペレーションの詳細については、後のそれぞれのサブセクションを参照してください。

トピック

- PutRecords を使用した複数のレコードの追加
- PutRecord を使用した単一レコードの追加

ソースアプリケーションは Kinesis Data Streams API を使用してストリームにデータを追加するため、1 つ以上のコンシューマーアプリケーションが同時にストリームからデータを取得して処理する可能性があることを常に念頭に置いてください。コンシューマーが Kinesis Data Streams API を使用してデータを取得する方法の詳細については、ストリームからのデータの取得を参照してください。

> **Important**
>
> データ保持期間の変更

PutRecords を使用した複数のレコードの追加

PutRecords オペレーションは、1 つのリクエストで Kinesis Data Streams に複数のレコードを送信します。PutRecords を使用することによって、プロデューサーは Kinesis Data Streams にデータを送信するときに高スループットを実現できます。各PutRecords リクエストは、最大 500 レコードをサポートできます。リクエストに含まれる各レコードは 1 MB、リクエスト全体の上限はパーティションキーを含めて最大 5 MB。後で説明する単一の PutRecord オペレーションと同様に、PutRecords はシーケンス番号とパーティションキーを使用します。ただし、PutRecord の SequenceNumberForOrdering パラメータは、PutRecords の呼び出しには含まれません。PutRecords オペレーションでは、リクエストの自然な順序ですべてのレコードを処理するよう試みます。

各データレコードには一意のシーケンス番号があります。シーケンス番号は、client.putRecords を呼び出してストリームにデータレコードを追加した後に、Kinesis Data Streams によって割り当てられます。同じパーティションキーのシーケンス番号は一般的に、時間の経過とともに大きくなります。PutRecordsリクエスト間の期間が長くなるほど、シーケンス番号は大きくなります。

> **Note**
>
> シーケンス番号は、同じストリーム内の一連のデータのインデックスとして使用することはできません。一連のデータを論理的に区別するには、パーティションキーを使用するか、データセットごとに個別のストリームを作成します。

PutRecords リクエストには、異なるパーティションキーのレコードを含めることができます。リクエストのスコープはストリームです。各リクエストには、リクエストの制限まで、パーティションキーとレコードのあらゆる組み合わせを含めることができます。複数の異なるパーティションキーを使用して、複数の異なるシャードを含むストリームに対して実行されたリクエストは、少数のパーティションキーを使用して少数のシャードに対して実行されたリクエストよりも一般的に高速です。レイテンシーを低減し、スループットを最大化するには、パーティションキーの数をシャードの数よりも大きくする必要があります。

PutRecords の例

次のコードでは、シーケンシャルなパーティションキーを持つ 100 件のデータレコードを作成し、DataStream という名前のストリームに格納しています。

```
        AmazonKinesisClientBuilder clientBuilder =
AmazonKinesisClientBuilder.standard();

        clientBuilder.setRegion(regionName);
        clientBuilder.setCredentials(credentialsProvider);
        clientBuilder.setClientConfiguration(config);

        AmazonKinesis kinesisClient = clientBuilder.build();

        PutRecordsRequest putRecordsRequest  = new PutRecordsRequest();
        putRecordsRequest.setStreamName(streamName);
        List <PutRecordsRequestEntry> putRecordsRequestEntryList  = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            PutRecordsRequestEntry putRecordsRequestEntry  = new
PutRecordsRequestEntry();

putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(i).getBytes()));
            putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d",
i));
            putRecordsRequestEntryList.add(putRecordsRequestEntry);
        }
```

```
        putRecordsRequest.setRecords(putRecordsRequestEntryList);
        PutRecordsResult putRecordsResult  =
  kinesisClient.putRecords(putRecordsRequest);
        System.out.println("Put Result" + putRecordsResult);
```

PutRecords のレスポンスには、レスポンスの Records の配列が含まれます。レスポンス配列の各レコードは、リクエスト配列内のレコードと自然な順序 (リクエストやレスポンスの上から下へ) で直接相互に関連付けられます。レスポンスの Records 配列には、常にリクエスト配列と同じ数のレコードが含まれます。

PutRecords 使用時のエラーの処理

デフォルトでは、リクエスト内の個々のレコードでエラーが発生しても、PutRecords リクエスト内のそれ以降のレコードの処理は停止されません。つまり、レスポンスの Records 配列には、正常に処理されたレコードと、正常に処理されなかったレコードの両方が含まれていることを意味します。正常に処理されなかったレコードを検出し、それ以降の呼び出しに含める必要があります。

正常に処理されたレコードには SequenceNumber 値と ShardID 値が、正常に処理されなかったレコードには ErrorCode 値と ErrorMessage 値が含まれます。ErrorCode パラメータはエラーのタイプを反映し、ProvisionedThroughputExceededException または InternalFailure のいずれかの値になります。ErrorMessageは、ProvisionedThroughputExceededException 例外に関するより詳細な情報として、スロットリングされたレコードのアカウント ID、ストリーム名、シャード ID などを示します。次の例では、PutRecords リクエストに 3 つのレコードがあります。2 番目のレコードは失敗し、レスポンスに反映されます。

Example PutRecords リクエストの構文

```
{
    "Records": [
        {
     "Data": "XzxkYXRhPl8w",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "AbceddeRFfg12asd",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "KFpcd98*7nd1",
     "PartitionKey": "partitionKey3"
```

```
        }
    ],
    "StreamName": "myStream"
}
```

Example PutRecords レスポンスの構文

```
{
    "FailedRecordCount”: 1,
    "Records": [
        {
     "SequenceNumber": "21269319989900637946712965403778482371",
     "ShardId": "shardId-000000000001"

        },
        {
     “ErrorCode":”ProvisionedThroughputExceededException”,
     “ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream
 exampleStreamName under account 111111111111."

        },
        {
     "SequenceNumber": "21269319989999637946712965403778482985",
     "ShardId": "shardId-000000000002"
        }
    ]
}
```

正常に処理されなかったレコードは、以降の PutRecords リクエストに含めることができます。最初に、FailedRecordCount の putRecordsResult パラメータを調べて、リクエスト内にエラーとなったレコードがあるかどうかを確認します。このようなレコードがある場合は、putRecordsEntry が ErrorCode 以外である各 null を、以降のリクエストに追加してください。このタイプのハンドラーの例については、次のコードを参照してください。

Example PutRecords エラーハンドラー

```
PutRecordsRequest putRecordsRequest = new PutRecordsRequest();
putRecordsRequest.setStreamName(myStreamName);
List<PutRecordsRequestEntry> putRecordsRequestEntryList = new ArrayList<>();
for (int j = 0; j < 100; j++) {
    PutRecordsRequestEntry putRecordsRequestEntry = new PutRecordsRequestEntry();
    putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(j).getBytes()));
```

```
    putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d", j));
    putRecordsRequestEntryList.add(putRecordsRequestEntry);
}

putRecordsRequest.setRecords(putRecordsRequestEntryList);
PutRecordsResult putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);

while (putRecordsResult.getFailedRecordCount() > 0) {
    final List<PutRecordsRequestEntry> failedRecordsList = new ArrayList<>();
    final List<PutRecordsResultEntry> putRecordsResultEntryList =
 putRecordsResult.getRecords();
    for (int i = 0; i < putRecordsResultEntryList.size(); i++) {
        final PutRecordsRequestEntry putRecordRequestEntry =
 putRecordsRequestEntryList.get(i);
        final PutRecordsResultEntry putRecordsResultEntry =
 putRecordsResultEntryList.get(i);
        if (putRecordsResultEntry.getErrorCode() != null) {
            failedRecordsList.add(putRecordRequestEntry);
        }
    }
    putRecordsRequestEntryList = failedRecordsList;
    putRecordsRequest.setRecords(putRecordsRequestEntryList);
    putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);
}
```

PutRecord を使用した単一レコードの追加

PutRecord の各呼び出しは、1 つのレコードに対して動作します。アプリケーションで常にリクエストごとに 1 つのレコードを送信する必要がある場合や、PutRecords を使用できないその他の理由がある場合を除いて、PutRecords を使用した複数のレコードの追加で説明している PutRecords オペレーションを使用します。

各データレコードには一意のシーケンス番号があります。シーケンス番号は、client.putRecord を呼び出してストリームにデータレコードを追加した後に、Kinesis Data Streams によって割り当てられます。同じパーティションキーのシーケンス番号は一般的に、時間の経過とともに大きくなります。PutRecordリクエスト間の期間が長くなるほど、シーケンス番号は大きくなります。

入力が立て続けに行われた場合、返されるシーケンス番号は大きくなるとは限りません。入力オペレーションが基本的に Kinesis Data Streams に対して同時に実行されるためです。同じパーティションキーに対して厳密にシーケンス番号が大きくなるようにするには、PutRecord の例のサンプルコードに示しているように、SequenceNumberForOrdering パラメータを使用します。

SequenceNumberForOrdering を使用するかどうかにかかわらず、inesis Data Streams が GetRecords の呼び出しを通じて受け取るレコードは厳密にシーケンス番号順になります。

> **Note**
>
> シーケンス番号は、同じストリーム内の一連のデータのインデックスとして使用することはできません。一連のデータを論理的に区別するには、パーティションキーを使用するか、データセットごとに個別のストリームを作成します。

パーティションキーはストリーム内のデータをグループ化するために使用されます。データレコードはそのパーティションキーに基づいてストリーム内でシャードに割り当てられます。具体的には、Kinesis Data Streams ではパーティションキー (および関連するデータ) を特定のシャードにマッピングするハッシュ関数への入力として、パーティションキーを使用します。

このハッシュメカニズムの結果として、パーティションキーが同じすべてのデータレコードは、ストリーム内で同じシャードにマッピングされます。ただし、パーティションキーの数がシャードの数を超えている場合、一部のシャードにパーティションキーが異なるレコードが格納されることがあります。設計の観点から、すべてのシャードが適切に使用されるようにするには、シャードの数 (setShardCount の CreateStreamRequest メソッドで指定) を一意のパーティションキーの数よりも大幅に少なくする必要があります。また、1 つのパーティションキーへのデータの流量をシャードの容量より大幅に小さくする必要があります。

PutRecord の例

以下のコードでは、2 つのパーティションキーに配分される 10 件のデータレコードを作成し、myStreamName という名前のストリームに格納しています。

```
for (int j = 0; j < 10; j++)
{
  PutRecordRequest putRecordRequest = new PutRecordRequest();
  putRecordRequest.setStreamName( myStreamName );
  putRecordRequest.setData(ByteBuffer.wrap( String.format( "testData-%d",
 j ).getBytes() ));
  putRecordRequest.setPartitionKey( String.format( "partitionKey-%d", j/5 ));
  putRecordRequest.setSequenceNumberForOrdering( sequenceNumberOfPreviousRecord );
  PutRecordResult putRecordResult = client.putRecord( putRecordRequest );
  sequenceNumberOfPreviousRecord = putRecordResult.getSequenceNumber();
}
```

上記のコード例では、setSequenceNumberForOrdering を使用して、各パーティションキー内で順番が厳密に増えるようにしています。このパラメータを効果的に使用するには、現在のレコードの SequenceNumberForOrdering (レコード n) を前のレコード (レコード n-1) のシーケンス番号に設定します。ストリームに追加されたレコードのシーケンス番号を取得するには、getSequenceNumber の結果に対して putRecord を呼び出します。

SequenceNumberForOrdering パラメーターを指定すると、同じパーティションキーのシーケンス番号が厳密に大きくなります。SequenceNumberForOrderingでは、複数のパーティションキーにわたるレコードの順序付けは用意されていません。

AWS Glue スキーマレジストリを使用してデータと相互作用する

Kinesis Data Streams を、AWS Glue スキーマレジストリと統合することができます。AWS Glue スキーマレジストリを使用すると、スキーマを一元的に検出、制御、および進化させながら、生成されたデータが登録されたスキーマによって継続的に検証されるようにできます。スキーマは、データレコードの構造と形式を定義します。スキーマは、信頼性の高いデータの公開、利用、または保存のための仕様をバージョニングしたものです。AWS Glue スキーマレジストリを使用すると、ストリーミングアプリケーション内のエンドツーエンドのデータ品質とデータガバナンスを改善できます。詳細については、AWS Glue スキーマレジストリを参照してください。この統合を設定する方法の 1 つは、AWS Java SDK で利用可能な PutRecords および PutRecord Kinesis Data Streams API を使用することです。

Kinesis Data Streams API を使用して Kinesis Data Streams とスキーマレジストリの統合を設定する方法の詳細については、ユースケース: Amazon Kinesis Data Streams と AWS Glue スキーマレジストリの統合のKinesis Data Streams API を使用したデータの操作セクションを参照してください。

Kinesis エージェントを使用した Amazon Kinesis Data Streams への書き込み

Kinesis エージェントはスタンドアロンの Java ソフトウェアアプリケーションであり、データを収集して Kinesis Data Streams に送信する簡単な方法です。このエージェントは一連のファイルを継続的に監視し、新しいデータをストリームに送信します。エージェントはファイルのローテーション、チェックポイント、失敗時の再試行を処理します。タイムリーで信頼性の高い簡単な方法で、すべてのデータを提供します。また、ストリーミングプロセスのモニタリングとトラブルシューティングに役立つ Amazon CloudWatch メトリクスも出力します。

デフォルトでは、レコードは改行文字 ('\n') に基づいて各ファイルから解析されます。しかし、複数行レコードを解析するよう、エージェントを設定することもできます (エージェントの設定を参照)。

このエージェントは、ウェブサーバー、ログサーバーおよびデータベースサーバーなど、Linux ベースのサーバー環境にインストールできます。エージェントをインストールした後で、モニタリング用のファイルとデータストリームを指定して設定します。エージェントが設定されると、ファイルから永続的にデータを収集して、ストリームに安全にデータを送信します。

トピック

前提条件

- オペレーティングシステムは Amazon Linux AMI バージョン 2015.09 以降、または Red Hat Enterprise Linux バージョン 7 以降でなければなりません。
- Amazon EC2 を使用してエージェントを実行している場合は、EC2 インスタンスを起動します。
- 次のいずれかの方法を使用して AWS 認証情報を管理します。
  - EC2 インスタンスを起動する際に IAM ロールを指定します。
  - エージェントを設定するときに AWS 認証情報を指定します (awsAccessKey「ID とawsSecretAccessキー」を参照)。
  - を編集`/etc/sysconfig/aws-kinesis-agent`して、リージョンと AWS アクセスキーを指定します。
  - EC2 インスタンスが別の AWS アカウントにある場合は、Kinesis Data Streams サービスへのアクセスを提供する IAM ロールを作成し、エージェントを設定するときにそのロールを指定します (assumeRoleARN and assumeRoleExternalId」を参照)。前述の方法のいずれかを使用し

て、このロールを引き受けるアクセス許可を持つ他のアカウントのユーザーの AWS 認証情報を指定します。

- 指定する IAM ロールまたは AWS 認証情報には、エージェントがストリームにデータを送信するために Kinesis Data Streams PutRecordsオペレーションを実行するアクセス許可が必要です。エージェント CloudWatch のモニタリングを有効にする場合は、 オペレーションを実行する CloudWatch PutMetricDataアクセス許可も必要です。詳細については、IAM を使用した Amazon Kinesis Data Streams リソースへのアクセスの制御「」、Amazon による Kinesis Data Streams エージェントの状態のモニタリング CloudWatch「」、およびCloudWatch 「アクセスコントロール」を参照してください。

エージェントのダウンロードとインストール

最初に、インスタンスに接続します。詳細については、「Amazon EC2 ユーザーガイド」の「インスタンスに接続する」を参照してください。 Amazon EC2 接続に問題がある場合は、Amazon EC2 ユーザーガイド」の「インスタンスへの接続のトラブルシューティング」を参照してください。

Amazon Linux AMI を使用してエージェントを設定する

次のコマンドを使用して、エージェントをダウンロードしてインストールします。

```
sudo yum install -y aws-kinesis-agent
```

Red Hat Enterprise Linux を使用してエージェントを設定する

次のコマンドを使用して、エージェントをダウンロードしてインストールします。

```
sudo yum install -y https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-
latest.amzn2.noarch.rpm
```

を使用して エージェントを設定するには GitHub

1. awlabs/amazon-kinesis-agent からエージェントをダウンロードします。
2. ダウンロードしたディレクトリまで移動し、次のコマンドを実行してエージェントをインストールします。

   ```
   sudo ./setup --install
   ```

Docker コンテナにエージェントをセットアップするには

Kinesis Agent は、amazonlinux コンテナベースを使ってコンテナで実行することもできます。次の Docker ファイルを使用し、`docker build` を実行します。

```
FROM amazonlinux

RUN yum install -y aws-kinesis-agent which findutils
COPY agent.json /etc/aws-kinesis/agent.json

CMD ["start-aws-kinesis-agent"]
```

エージェントの設定と開始

エージェントを設定して開始するには

1. (デフォルトのファイルアクセス許可を使用している場合、スーパーユーザーとして) 設定ファイル (`/etc/aws-kinesis/agent.json`) を開き、編集します。

   この設定ファイルで、エージェントがデータを集めるファイル ("`filePattern`") とエージェントがデータを送信するストリーム ("`kinesisStream`") を指定します。ファイル名はパターンで、エージェントはファイルローテーションを確認する点に注意してください。1 秒あたりに一度だけ、ファイルを交替または新しいファイルを作成できます。エージェントはファイル作成タイムスタンプを使用して、どのファイルを追跡してストリームに送信するかを判断します。新規ファイルの作成やファイルの交換を 1 秒あたりに一度以上頻繁に交換すると、エージェントはそれらを適切に区別できません。

   ```
   {
      "flows": [
           {
               "filePattern": "/tmp/app.log*",
               "kinesisStream": "yourkinesisstream"
           }
      ]
   }
   ```

2. エージェントを手動で開始する:

   ```
   sudo service aws-kinesis-agent start
   ```

3. (オプション) システムスタートアップ時にエージェントを開始するように設定します。

```
sudo chkconfig aws-kinesis-agent on
```

エージェントは、システムのサービスとしてバックグラウンドで実行されます。継続的に指定ファイルをモニタリングし、指定されたストリームにデータを送信します。エージェント活動は、/var/log/aws-kinesis-agent/aws-kinesis-agent.log に記録されます。

エージェントの設定

エージェントは、2つの必須設定、filePattern と kinesisStream、さらに追加機能として任意の設定をサポートしています。必須およびオプションの設定を /etc/aws-kinesis/agent.json で指定できます。

設定ファイルを変更した場合は、次のコマンドを使用してエージェントを停止および起動する必要があります。

```
sudo service aws-kinesis-agent stop
sudo service aws-kinesis-agent start
```

または、次のコマンドを使用できます。

```
sudo service aws-kinesis-agent restart
```

全般設定は次のとおりです。

構成設定	説明
assumeRoleARN	ユーザーが引き受けるロールの ARN。詳細については、「IAM ユーザーガイド」の「IAM ロールを使用した AWS アカウント間のアクセスの委任」を参照してください。
assumeRoleExternalId	ロールを引き受けることができるユーザーを決定するオプションの ID。詳細については、IAM ユーザーガイドの外部 ID の使用方法を参照してください。
awsAccessKeyId	AWS デフォルトの認証情報を上書きする アクセスキー ID。この設定は、他のすべての認証情報プロバイダーに優先されます。

構成設定	説明
awsSecretAccessKey	AWS デフォルトの認証情報を上書きする シークレットキー。この設定は、他のすべての認証情報プロバイダーに優先されます。
cloudwatch.emitMetrics	設定されている場合、エージェントがメトリクスを に出力 CloudWatch できるようにします (true)。 デフォルト: true
cloudwatch.endpoint	のリージョンエンドポイント CloudWatch。 デフォルト: monitoring.us-east-1.amazonaws.com
kinesis.endpoint	Kinesis Data Streams のリージョンのエンドポイントです。 デフォルト: kinesis.us-east-1.amazonaws.com

フロー設定は次のとおりです。

構成設定	説明
dataProcessingOptions	ストリームに送信される前に解析された各レコードに適用されるの処理オプションの一覧。処理オプションは指定した順序で実行されます。詳細については、エージェントを使用してデータを事前処理するを参照してください。
kinesisStream	[必須] ストリームの名前。
filePattern	〔必須] エージェントによって取得されるために一致する必要があるディレクトリとファイルパターン。このパターンに一致するすべてのファイルは、読み取り権限を aws-kinesis-agent-user に付与する必要があります。ファイルを含むディレクトリには、読み取りと実行権限を aws-kinesis-agent-user に付与する必要があります。
initialPosition	ファイルの解析が開始される最初の位置。有効な値は、START_OF_FILE および END_OF_FILE です。 デフォルト: END_OF_FILE

構成設定	説明
maxBuffer AgeMillis	エージェントがストリームに送信する前にデータをバッファーする最大時間 (ミリ秒)。 値の範囲: 1,000 ~ 900,000 (1 秒 ~ 15 分) デフォルト: 60,000 (1 分)
maxBuffer SizeBytes	エージェントがストリームに送信する前にデータをバッファーする最大サイズ (バイト)。 値の範囲: 1 ~ 4,194,304 (4 MB) デフォルト: 4,194,304 (4 MB)
maxBuffer SizeRecords	エージェントがストリームに送信する前にデータをバッファーするレコードの最大数。 値の範囲: 1 ~ 500 デフォルト: 500
minTimeBe tweenFile PollsMillis	エージェントが新しいデータのモニタリング対象ファイルをポーリングし、解析する時間間隔 (ミリ秒単位)。 値の範囲: 1 以上 デフォルト: 100
multiLine StartPattern	レコードの開始を識別するパターン。レコードは、パターンに一致する 1 行と、それに続くパターンに一致しない行で構成されます。有効な値は正規表現です。デフォルトでは、ログファイルのそれぞれの改行は 1 つのレコードとして解析されます。
partition KeyOption	パーティションのキーを生成する方法。有効な値は RANDOM (ランダムに生成される整数) と DETERMINISTIC (データから計算されるハッシュ値) です。 デフォルト: RANDOM

構成設定	説明
skipHeaderLines	モニタリング対象ファイルの始めにエージェントが解析をスキップするの行数。 値の範囲: 0 以上 デフォルト: 0 (ゼロ)
truncatedRecordTerminator	レコードのサイズが Kinesis Data Streams レコードの許容サイズを超えたときに解析されたレコードを切り捨てるために、エージェントが使用する文字列。(1,000 KB) デフォルト: '\n' (改行)

複数のファイルディレクトリを監視し、複数のストリームに書き込み

複数のフロー設定を指定することによって、エージェントが複数のファイルディレクトリを監視し、複数のストリームにデータを送信するように設定できます。次の設定例では、エージェントは 2 つのファイルディレクトリをモニタリングし、それぞれ Kinesis ストリームと Firehose 配信ストリームにデータを送信します。Kinesis Data Streams と Firehose に異なるエンドポイントを指定して、Kinesis ストリームと Firehose 配信ストリームが同じリージョンに存在する必要がないようにできます。

```
{
    "cloudwatch.emitMetrics": true,
    "kinesis.endpoint": "https://your/kinesis/endpoint",
    "firehose.endpoint": "https://your/firehose/endpoint",
    "flows": [
        {
            "filePattern": "/tmp/app1.log*",
            "kinesisStream": "yourkinesisstream"
        },
        {
            "filePattern": "/tmp/app2.log*",
            "deliveryStream": "yourfirehosedeliverystream"
        }
    ]
}
```

Firehose で エージェントを使用する方法の詳細については、「Kinesis Agent を使用した Amazon Data Firehose への書き込み」を参照してください。

エージェントを使用してデータを事前処理する

エージェントはストリームにレコードを送信する前に、モニタリング対象ファイルから解析したレコードを事前処理できます。ファイルフローに dataProcessingOptions 設定を追加することで、この機能を有効にできます。1 つ以上の処理オプションを追加でき、また指定されている順序で実行されます。

エージェントは、リストされた次の処理オプションに対応しています。エージェントはオープンソースであるため、処理オプションを開発および拡張できます。Kinesis エージェントからエージェントをダウンロードできます。

処理オプション

SINGLELINE

改行文字、先頭のスペース、末尾のスペースを削除することで、複数行レコードを単一行レコードに変換します。

```
{
    "optionName": "SINGLELINE"
}
```

CSVTOJSON

区切り形式から JSON 形式にレコードを変換します。

```
{
    "optionName": "CSVTOJSON",
    "customFieldNames": [ "field1", "field2", ... ],
    "delimiter": "yourdelimiter"
}
```

customFieldNames

[必須] 各 JSON キー値のペアでキーとして使用されるフィールド名。たとえば、["f1", "f2"] を指定した場合は、レコードv1、v2は {"f1":"v1","f2":"v2"} に変換されます。

delimiter

レコードで区切り記号として使用する文字列。デフォルトはカンマ (,) です。

LOGTOJSON

ログ形式から JSON 形式にレコードを変換します。サポートされているログ形式は、Apache Common Log、Apache Combined Log、Apache Error Log、および RFC3164 Syslog です。

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "logformat",
    "matchPattern": "yourregexpattern",
    "customFieldNames": [ "field1", "field2", … ]
}
```

logFormat

[必須] ログエントリ形式。以下の値を指定できます。

- COMMONAPACHELOG — Apache Common Log 形式。各ログエントリは、デフォルトで次のパターン%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes}になります。
- COMBINEDAPACHELOG — Apache Combined Log 形式。各ログエントリは、デフォルトで次のパターン%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes} %{referrer} %{agent}になります。
- APACHEERRORLOG — Apache Error Log 形式。各ログエントリは、デフォルトで次のパターン[%{timestamp}] [%{module}:%{severity}] [pid %{processid}:tid %{threadid}] [client: %{client}] %{message}になります。
- SYSLOG — FC3164 Syslog 形式。各ログエントリは、デフォルトで次のパターン%{timestamp} %{hostname} %{program}[%{processid}]: %{message}になります。

matchPattern

ログエントリから値を取得するために使用する正規表現パターン。この設定は、ログエントリが定義されたログ形式の一つとして存在していない場合に使用されます。この設定を使用する場合は、customFieldNames を指定する必要があります。

customFieldNames

JSON キー値のペアでキーとして使用されるカスタムフィールド名。matchPattern から抽出した値のフィールド名を定義するために、または事前定義されたログ形式のデフォルトのフィールド名を上書きするために、この設定を使用できます。

Example : LOGTOJSON 設定

JSON形式に変換された Apache Common Log エントリの LOGTOJSON 設定の一つの例を次に示します。

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG"
}
```

変換前:

```
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision
 HTTP/1.1" 200 6291
```

変換後:

```
{"host":"64.242.88.10","ident":null,"authuser":null,"datetime":"07/
Mar/2004:16:10:02 -0800","request":"GET /mailman/listinfo/hsdivision
 HTTP/1.1","response":"200","bytes":"6291"}
```

Example : カスタムフィールドがある LOGTOJSON 設定

こちらは LOGTOJSON 設定の別の例です。

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
    "customFieldNames": ["f1", "f2", "f3", "f4", "f5", "f6", "f7"]
}
```

この設定では、前の例からの同じ Apache Common Log エントリは、次のように JSON 形式に変換されます。

```
{"f1":"64.242.88.10","f2":null,"f3":null,"f4":"07/Mar/2004:16:10:02 -0800","f5":"GET /
mailman/listinfo/hsdivision HTTP/1.1","f6":"200","f7":"6291"}
```

Example : Apache Common Log エントリの変換

次のフロー設定は Apache Common Log エントリを JSON 形式の単一行レコードに変換します。

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "dataProcessingOptions": [
                {
                    "optionName": "LOGTOJSON",
                    "logFormat": "COMMONAPACHELOG"
                }
            ]
        }
    ]
}
```

Example : 複数行レコードの変換

次のフロー設定は、最初の行が[SEQUENCE=で開始している複数行レコードを解析します。各レコードはまず単一行レコードに変換されます。次に、値はタブの区切り記号に基づいたレコードから取得されます。取得された値は指定された customFieldNames 値にマッピングされ、JSON 形式の単一行レコードを形成します。

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "multiLineStartPattern": "\\[SEQUENCE=",
            "dataProcessingOptions": [
                {
                    "optionName": "SINGLELINE"
                },
                {
                    "optionName": "CSVTOJSON",
                    "customFieldNames": [ "field1", "field2", "field3" ],
                    "delimiter": "\\t"
                }
            ]
        }
    ]
}
```

Example : 一致パターンで LOGTOJSON 設定

こちらは、最後のフィールド (バイト) が省略された JSON 形式に変換された Apache Common Log エントリの LOGTOJSON 設定の一例です。

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
    "matchPattern": "^([\\d.]+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.
+?)\" (\\d{3})",
    "customFieldNames": ["host", "ident", "authuser", "datetime", "request",
 "response"]
}
```

変換前:

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
 200
```

変換後:

```
{"host":"123.45.67.89","ident":null,"authuser":null,"datetime":"27/Oct/2000:09:27:09
 -0400","request":"GET /java/javaResources.html HTTP/1.0","response":"200"}
```

エージェント CLI コマンド

システムスタートアップ時のエージェントの自動的開始:

```
sudo chkconfig aws-kinesis-agent on
```

エージェントのステータスの確認:

```
sudo service aws-kinesis-agent status
```

エージェントの停止:

```
sudo service aws-kinesis-agent stop
```

この場所からエージェントのログファイルを読む:

```
/var/log/aws-kinesis-agent/aws-kinesis-agent.log
```

エージェントのアンインストール:

```
sudo yum remove aws-kinesis-agent
```

よくある質問

Windows 用の Kinesis Agent はありますか?

Windows 用 Kinesis Agent は Linux プラットフォーム用 Kinesis Agent とは異なるソフトウェアです。

Kinesis Agent の速度が低下したり、**RecordSendErrors** が増加したりするのはなぜですか?

通常、これは Kinesis のスロットリングが原因です。Kinesis Data Streams の `WriteProvisionedThroughputExceeded`メトリクスまたは Firehose 配信ストリームの `ThrottledRecords`メトリクスを確認します。これらのメトリクスが 0 を超えている場合は、ストリームの上限を引き上げる必要があることを意味します。詳細については、「Kinesis Data Stream limits」と「Amazon Firehose Delivery Streams」を参照してください。

スロットリングが原因ではないことがわかったら、Kinesis Agent が大量の小規模ファイルをテーリングするように設定されているかどうかを確認してください。Kinesis Agent が新しいファイルをテーリングするときには遅延が発生するため、少量の大きなファイルをテーリングするようにします。ログファイルを大きなファイルに統合してみてください。

java.lang.OutOfMemoryError の例外が発生するのはなぜですか?

Kinesis Agent に、現在のワークロードを処理するための十分なメモリがないためです。`/usr/bin/start-aws-kinesis-agent` で `JAVA_START_HEAP` と `JAVA_MAX_HEAP` を増やしてエージェントを再起動してみてください。

IllegalStateException : connection pool shut down の例外が発生するのはなぜですか?

Kinesis エージェントに、現在のワークロードを処理するための十分な接続がないためです。`/etc/aws-kinesis/agent.json` の一般的なエージェント設定で `maxConnections`

と maxSendingThreads を増やしてみてください。これらのフィールドのデフォルト値は、使用可能なランタイムプロセッサの 12 倍です。エージェント設定の詳細については、AgentConfiguration「.java」を参照してください。

Kinesis Agent に関する別の問題をデバッグする方法を教えてください。

DEBUG レベルログは /etc/aws-kinesis/log4j.xml で有効にできます。

Kinesis Agent はどのように設定するとよいですか?

maxBufferSizeBytes の値が小さいほど、Kinesis Agent がデータを送信する頻度が高くなります。そのため、レコードの配信時間が短縮されますが、Kinesis への 1 秒あたりのリクエスト数も増えます。

Kinesis Agent が重複レコードを送信するのはなぜですか?

これはファイルテーリングの設定ミスが原因です。各 fileFlow's filePattern が それぞれ 1 つのファイルのみと一致するようにします。また、使用されている logrotate モードが copytruncate モードになっている場合にも発生することがあります。重複を避けるため、モードをデフォルトか作成モードに変更してみてください。重複レコードの処理に関する詳細は、「Handling Duplicate Records」を参照してください。

その他の AWS サービスを使用した Kinesis Data Streams への書き込み

以下は、Kinesis Data Streams へのデータの書き込みのために Kinesis Data Streams と直接統合できる、その他の AWS サービスのリストです。

トピック

- AWS Amplify
- Amazon Aurora
- Amazon CloudFront
- Amazon CloudWatch Logs
- Amazon Connect
- AWS Database Migration Service
- 「Amazon DynamoDB」
- Amazon EventBridge

- AWS IoT Core
- Amazon Relational Database Service
- Amazon Pinpoint
- Amazon Quantum Ledger Database

AWS Amplify

Amazon Kinesis Data Streams を使用して、AWS Amplify を使用して構築されたモバイルアプリケーションからのデータを、リアルタイムでの処理のために簡単にストリーミングすることができます。その後、リアルタイムダッシュボードの構築、例外のキャプチャとアラートの生成、推奨事項の促進、およびビジネスや運用に関するその他のリアルタイムでの判断を行うことができます。また、Amazon Simple Storage Service、Amazon DynamoDB、Amazon Redshift など他のサービスに容易にデータを送信することもできます。

詳細については、「AWS Amplify Developer Center」で「Using Amazon Kinesis」を参照してください。

Amazon Aurora

Amazon Kinesis Data Streams を使用して Amazon Aurora DB クラスター上のアクティビティをモニタリングできます。Aurora DB クラスターは、データベースアクティビティストリームを使用することで、アクティビティを Amazon Kinesis データストリームにリアルタイムでプッシュします。その後、これらのアクティビティを消費し、監査して、アラートを生成する、コンプライアンス管理用のアプリケーションを構築できます。また、Amazon Amazon Firehose を使用してデータを保存することもできます。

詳細については、「Amazon Aurora デベロッパーガイド」の「データベースアクティビティストリーム」を参照してください。

Amazon CloudFront

CloudFront のリアルタイムログで Amazon Kinesis Data Streams を使用して、ディストリビューションに対して行われたリクエストに関する情報をリアルタイムで取得することができます。その後、独自の Kinesis データストリームコンシューマーを構築したり、Amazon Kinesis Data Firehose を使用して、Amazon Simple Storage Service (Amazon S3)、Amazon Redshift、Amazon OpenSearch Service (OpenSearch Service)、またはサードパーティーのログ処理サービスにログデータを送信したりすることができます。

詳細については、「Amazon CloudFront デベロッパーガイド」の「リアルタイムログ」を参照してください。

Amazon CloudWatch Logs

CloudWatch サブスクリプションを使用して Amazon CloudWatch Logs からのログイベントのリアルタイムフィードにアクセスし、カスタム処理、分析、および他のシステムへのロードを行うために、Amazon Kinesis データストリームに配信することができます。

詳細については、「Amazon CloudWatch Logs ユーザーガイド」の「サブスクリプションを使用したログデータのリアルタイム処理」を参照してください。

Amazon Connect

Kinesis Data Streams を使用して、Amazon Connect インスタンスからコンタクトレコードとエージェントイベントをリアルタイムでエクスポートできます。Amazon Connect Customer Profiles からのデータストリーミングを有効にして、新しいプロファイルの作成や既存のプロファイルへの変更に関する Kinesis データストリームへの更新情報を自動的に受け取ることもできます。

その後、コンシューマーアプリケーションを構築して、データをリアルタイムで処理し、分析することができます。例えば、コンタクトレコードやカスタマープロファイルデータを使用することで、最新情報を用いて CRM やマーケティング自動化ツールなどのソースシステムデータを最新の状態に保つことができます。エージェントイベントデータを使用すれば、エージェント情報やイベントを表示するダッシュボードを作成したり、特定のエージェントアクティビティに関するカスタム通知をトリガーしたりすることができます。

詳細については、「Amazon Connect 管理者ガイド」の「インスタンスのデータストリーミング」、「リアルタイムエクスポートの設定」、および「エージェントのイベントストリーム」を参照してください。

AWS Database Migration Service

AWS Database Migration Service を使用して、Amazon Kinesis データストリームにデータを移行することができます。その後、データレコードをリアルタイムで処理するコンシューマーアプリケーションを構築できます。また、Amazon Simple Storage Service、Amazon DynamoDB、および Amazon Redshift などのその他のダウンストリームサービスに、データを簡単に送信することもできます。

詳細については、「AWS Database Migration Service ユーザーガイド」の「ゲートウェイを使用したデータの取り込み」を参照してください。

「Amazon DynamoDB」

Amazon Kinesis Data Streams を使用して Amazon DynamoDB への変更をキャプチャできます。Kinesis Data Streams は、DynamoDB テーブル内での項目レベルの変更をキャプチャし、それらを Kinesis データストリームにレプリケートします。コンシューマーアプリケーションは、このストリームにアクセスして項目レベルの変更をリアルタイムで確認し、これらの変更をダウンストリームに配信したり、内容に基づいてアクションを実行したりすることができます。

詳細については、「Amazon DynamoDB デベロッパーガイド」の「Kinesis Data Streams の DynamoDB との連携について」を参照してください。

Amazon EventBridge

Kinesis Data Streams を使用することで、EventBridge 内の AWS API コールイベントをストリームに送信し、コンシューマーアプリケーションを構築して、大量のデータを処理することができます。また、Kinesis Data Streams を EventBridge Pipes 内のターゲットとして使用して、オプションのフィルタリングと強化を行った後で、利用可能なソースの 1 つからストリームにレコードを配信することもできます。

詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon Kinesis ストリームにイベントを送信する」と「EventBridge Pipes」を参照してください。

AWS IoT Core

AWS IoT Rule アクションを使用することで、AWS IoT Core の MQTT メッセージからリアルタイムでデータを書き込むことができます。その後、書き込まれたデータを処理し、内容を分析してアラートを生成するとともに、データを分析アプリケーションや他の AWS サービスに配信するアプリケーションを構築できます。

詳細については、「AWS IoT デベロッパーガイド」の「Kinesis Data Streams」を参照してください。

Amazon Relational Database Service

Amazon Kinesis Data Streams を使用して、Amazon RDS インスタンス上のアクティビティをモニタリングできます。Amazon RDS は、データベースアクティビティストリームを使用することで、アクティビティを Amazon Kinesis データストリームにリアルタイムでプッシュします。その後、これらのアクティビティを消費し、監査して、アラートを生成する、コンプライアンス管理用のアプリケーションを構築できます。また、Amazon Amazon Firehose を使用してデータを保存することもできます。

詳細については、「Amazon RDS デベロッパーガイド」の「データベースアクティビティストリーム」を参照してください。

Amazon Pinpoint

Amazon Pinpoint は、Amazon Kinesis Data Streams にイベントデータを送信するように設定することができます。Amazon Pinpoint は、キャンペーン、ジャーニー、トランザクション用の Eメールや SMS メッセージのイベントデータを送信することができます。その後、データを分析アプリケーションに取り込むか、イベントの内容に基づいてアクションを実行する独自のコンシューマーアプリケーションを構築することができます。

詳細については、「Amazon Pinpoint Developer Guide」の「Streaming Events」を参照してください。

Amazon Quantum Ledger Database

ジャーナルにコミットされたすべてのドキュメントリビジョンをキャプチャして、このデータを Amazon Kinesis Data Streams にリアルタイムで配信するストリームを QLDB で作成することができます。QLDB ストリーミングは、台帳のジャーナルから Kinesis データストリームリソースへの連続的なデータのフローです。その後、Kinesis ストリーミングプラットフォーム、または Kinesis Client Library を使用して、ストリームを消費し、データレコードを処理して、データコンテンツを分析することができます。QLDB ストリームは、`control`、`block summary`、および `revision details` の 3 つのタイプで Kinesis Data Streams にデータを書き込みます。

詳細については、「Amazon QLDB Developer Guide」の「Streams」を参照してください。

サードパーティー統合の使用

Kinesis データストリームには、Kinesis Data Streams と統合する以下のサードパーティーオプションのいずれかを使用してデータを書き込むことができます。

トピック

- Apache Flink
- Fluentd
- Debezium
- Oracle GoldenGate
- Kafka Connect

- Adobe Experience
- Striim

Apache Flink

Apache Flink は、制限なしおよび制限付きのデータストリームでのステートフル計算のためのフレームワークかつ分散処理エンジンです。Apache Flink から Kinesis Data Streams への書き込みの詳細については、「Amazon Kinesis Data Streams Connector」を参照してください。

Fluentd

Fluentd は、統合ロギングレイヤーのためのオープンソースデータコレクターです。Fluentd から Kinesis Data Streams への書き込みの詳細については、「Stream Processing with Kinesis」を参照してください。

Debezium

Debezium は、変更データキャプチャのためのオープンソースの分散型プラットフォームです。Debezium から Kinesis Data Streams への書き込みの詳細については、「Streaming MySQL Data Changes to Amazon Kinesis」を参照してください。

Oracle GoldenGate

Oracle GoldenGate は、1 つのデータベースから別のデータベースへのデータのレプリケーション、フィルタリング、および変換を可能にするソフトウェア製品です。Oracle GoldenGate から Kinesis Data Streams への書き込みの詳細については、「Data replication to Kinesis Data Stream using Oracle GoldenGate」を参照してください。

Kafka Connect

Kafka Connect は、Apache Kafka と他のシステムの間でデータをスケーラブルかつ確実にストリーミングするためのツールです。Apache Kafka から Kinesis Data Streams へのデータの書き込みの詳細については、「Kinesis kafka connector」を参照してください。

Adobe Experience

Adobe Experience Platform は、組織があらゆるシステムからの顧客データを一元化して標準化することを可能にします。その後、データサイエンスと機械学習を適用して、充実感のあるパーソナライズされたエクスペリエンスの設計と提供を劇的に向上させます。Adobe Experience Platform から

Kinesis Data Streams へのデータの書き込みの詳細については、Amazon Kinesis connection の作成方法を参照してください。

Striim

Striim は、リアルタイムでのデータの収集、フィルタリング、変換、強化、集約、分析、および配信のための完全なエンドツーエンドのインメモリプラットフォームです。Striim から Kinesis Data Streams にデータを書き込む方法の詳細については、「Kinesis Writer」を参照してください。

Amazon Kinesis Data Streams プロデューサーのトラブルシューティング

以下のセクションでは、Amazon Kinesis Data Streams プロデューサーの操作中に発生する可能性がある一般的な問題に対する解決策を示します。

- プロデューサーアプリケーションの書き込みの速度が予想よりも遅い
- 承認されていない KMS マスターキーの権限エラー
- プロデューサーにとって一般的な問題、質問、トラブルシューティングのアイデア

プロデューサーアプリケーションの書き込みの速度が予想よりも遅い

書き込みのスループットが予想よりも遅くなる最も一般的な理由は次のとおりです。

- サービスの制限を超過している
- プロデューサーの最適化

サービスの制限を超過している

サービスの制限を超過している 呼び出しによって制限が異なることに注意して、クォータと制限 を確認してください。たとえば、書き込みと読み取りのシャードレベルの制限は最もよく知られていますが、以下のようなストリームレベルの制限もあります。

- CreateStream
- DeleteStream
- ListStreams
- GetShardIterator
- MergeShards

- DescribeStream
- DescribeStreamSummary

CreateStream、DeleteStream、ListStreams、 GetShardIterator、MergeShards のオペレーションは、1 秒あたり 5 個の呼び出しに制限されます。DescribeStream オペレーションは、1 秒あたり 10 個の呼び出しに制限されます。DescribeStreamSummary オペレーションは、1 秒あたり 20 個の呼び出しに制限されます。

このような呼び出しが原因でない場合は、選択したパーティションキーを使用してすべてのシャードに put オペレーションを均等に分散できること、どのパーティションキーもサービスの制限に達していないことを確認します。これには、ピークスループットを測定して、ストリームのシャードの数を考慮する必要があります。ストリーム管理の詳細については、ストリームの作成と管理を参照してください。

> **Tip**
>
> シングルレコードオペレーション PutRecord では、スループットスロットリングの計算結果がキロバイト単位に四捨五入されます。マルチレコードオペレーション PutRecords では、各セルのレコードの累計が四捨五入されます。たとえば、PutRecords は 1.1 KB になる 600 レコードのリクエストをスロットリングしません。

プロデューサーの最適化

プロデューサーの最適化を始める前に、完了しておかなければならない重要なタスクがいくつかあります。最初に、レコードのサイズと 1 秒あたりのレコード数で必要となるスループットピークを特定します。次に、制限要素としてのストリーム容量を除外します (サービスの制限を超過している)。ストリーム容量を除外している場合は、以下のプロデューサーの 2 つの一般的なタイプのトラブルシューティングのヒントと最適化のガイドラインを使用します。

ラージプロデューサー

ラージプロデューサーは、通常オンプレミスサーバーまたは Amazon EC2 インスタンスから実行されます。ラージプロデューサーからより高いスループットを必要とするお客様は、通常レコードあたりのレイテンシーに注意を払います。レイテンシーを処理する戦略として、お客様がレコードをマイクロバッチ/バッファできる場合は、Kinesis Producer Library (高度な集約ロジックがある) を使用するか、マルチレコードオペレーション PutRecords を使用するか、レコードをより大きいファイルに集約してからシングルレコードオペレーション PutRecord を使用します。バッチ/バッファを使

用できない場合は、複数のスレッドを使用して Kinesis Data Streams サービスに同時に書き込みます。AWS SDK for Java とその他の SDK には、ごく少数のコードでこれを実行できる非同期クライアントが含まれます。

スモールプロデューサー

スモールプロデューサーは、通常モバイルアプリケーション、IoT デバイス、またはウェブクライアントです。モバイルアプリケーションの場合は、PutRecords オペレーションを使用するか、AWS モバイル SDK の Kinesis レコーダーを使用することをお勧めします。詳細については、AWS Mobile SDK for Android 入門ガイドおよび AWS Mobile SDK for iOS 入門ガイドを参照してください。モバイルアプリケーションは、本来断続的な接続を処理する必要があり、PutRecords のようなバッチ put タイプを必要とします。何らかの理由でバッチを使用できない場合は、上記のラージプロデューサーの情報を参照してください。プロデューサーがブラウザの場合、生成されるデータの量は通常非常に小さなものとなります。ただし、アプリケーションの重要なパスに put オペレーションを配置することはお勧めしません。

承認されていない KMS マスターキーの権限エラー

このエラーは、プロデューサーアプリケーションが KMS マスターキーに対するアクセス許可なしで、暗号化されたストリームに書き込みを行うときに発生します。KMS キーにアクセスする許可をアプリケーションに割り当てるには、KMS でのキーポリシーの使用およびAWS KMS での IAM ポリシーの使用を参照してください。

プロデューサーにとって一般的な問題、質問、トラブルシューティングのアイデア

- Kinesis データストリームで 500 内部サーバーエラーが返されるのはなぜですか?
- Flink から Kinesis Data Streams に書き込むときのタイムアウトエラーのトラブルシューティング方法を教えてください。
- Kinesis Data Streams のスロットリングエラーのトラブルシューティング方法を教えてください。
- Kinesis Data Streams がスロットリングされるのはなぜですか?
- KPL を使用して、データレコードを Kinesis Data Streams に入れるにはどうすればよいですか?

Kinesis Data Streams プロデューサーの高度なトピック

このセクションでは、Amazon Kinesis Data Streams プロデューサーを最適化する方法について説明します。

トピック

KPL の再試行とレート制限

KPL addUserRecord( ) オペレーションを使用して Kinesis Producer Library (KPL) ユーザーレコードを追加すると、レコードはタイムスタンプが付けられて、RecordMaxBufferedTime 設定パラメータで期限が設定されたバッファに追加されます。このタイムスタンプと期限の組み合わせにより、バッファの優先順位が設定されます。レコードは、次の条件に基づいてバッファからフラッシュされます。

- バッファの優先度
- 集約設定
- 収集設定

バッファの動作に影響を与える集約および収集の設定パラメータは次のとおりです。

- AggregationMaxCount
- AggregationMaxSize
- CollectionMaxCount
- CollectionMaxSize

フラッシュされたレコードは、Kinesis Data Streams API オペレーション PutRecords の呼び出しを使用して Amazon Kinesis Data Streams レコードとして Kinesis Data Streams に送信されます。PutRecords オペレーションはストリームにリクエストを送信しますが、すべての失敗または部分的な失敗を示す場合があります。失敗したレコードは、自動的に KPL バッファに戻されます。新しい期限は、次の 2 つの値のうち小さい方に基づいて設定されます。

- 現在の RecordMaxBufferedTime 設定の半分
- レコードの有効期限値

この戦略では、再試行する KPL ユーザーレコードをそれ以降の Kinesis Data Streams API コールに含めることができ、Kinesis Data Streams レコードの有効期限値を適用しながら、スループットを改善し、複雑さを減らすことができます。バックオフアルゴリズムがないため、これは比較的積極的な

再試行戦略です。過剰な再試行による大量送信は、次のセクションで説明するレート制限により防止できます。

レート制限

KPL にはレート制限機能があり、1 つのプロデューサーからの送信されるシャード単位のスループットを制限できます。レート制限は、Kinesis Data Streams のレコードとバイトに別々のバケットを使用するトークンバケットアルゴリズムを使用して実装されています。Kinesis Data Streams への書き込みが成功するたびに、特定のしきい値に達するまで、各バケットに 1 つまたは複数のトークンが追加されます。このしきい値は設定できますが、デフォルトでは実際のシャード制限より 50 パーセント大きく設定され、単一のプロデューサーによるシャードの飽和が許されています。

この制限を小さくすることにより、過剰な再試行による大量送信を抑制できます。ただし、ベストプラクティスは、各プロデューサーについて、最大スループットまで積極的に再試行することと、ストリームの容量を拡大し、適切なパーティションキー戦略を実装することにより、結果的に過剰と判断されたスロットリングを適切に処理することです。

KPL 集約を使用するときの考慮事項

結果として得られた Amazon Kinesis Data Streams レコードのシーケンス番号方式は同じままですが、集約された Kinesis Data Streams レコードに含まれるKinesis Producer Library (KPL) ユーザーレコードのインデックス作成は 0 (ゼロ) から始まります。ただし、シーケンス番号に依存しない方法で KPL ユーザーレコードを一意に識別する限り、集約 (KPL ユーザーレコードの Kinesis Data Streams レコードへの集約) とその後の集約解除 (Kinesis Data Streams レコードの KPL ユーザーレコードへの集約解除) で自動的に考慮されるため、このようにインデックス作成が 0 (ゼロ) から始まることをコード上は無視してかまいません。これは、コンシューマーが KCL と AWS SDK のどちらを使用している場合でも適用されます。AWS SDK で提供される API を使用してコンシューマーが書かれている場合、この集約機能を使用するには、KPL の Java 部分をビルドに組み込む必要があります。

KPL ユーザーレコードの一意な識別子としてシーケンス番号を使用する場合、`Record` および `UserRecord` に提供されている、契約に順守した `public int hashCode()` および `public boolean equals(Object obj)` オペレーションを使用して、KPL ユーザーレコードの比較を有効にすることをお勧めします。さらに、KPL ユーザーレコードのサブシーケンス番号を調べる必要がある場合は、そのユーザーレコードを `UserRecord` インスタンスにキャストして、そのサブシーケンス番号を取得することができます。

詳細については、コンシューマーの集約解除を参照してください。

Amazon Kinesis Data Streams からのデータの読み取り

コンシューマーは、Kinesis Data Streams からのデータを処理するアプリケーションです。コンシューマーで拡張ファンアウトを使用すると、独自の 2 MB/秒の読み取りスループットが割り当てられ、その読み取りスループットを他のコンシューマーと競合することなく、複数のコンシューマーが並行して同じストリームからデータを読み取ることができます。シャードの拡張ファンアウト機能を使用するには、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

デフォルトで、ストリーム内のシャードは、シャードあたり 2 MB/秒の読み取りスループットを提供します。このスループットは、指定されたシャードから読み取りを行うすべてのコンシューマー間で共有されます。つまり、シャードあたり 2 MB/秒のデフォルトのスループットは固定であり、そのシャードから複数のコンシューマーが読み取る場合でも変わりません。このデフォルトのシャードのスループットを使用するには、スループット共有カスタムコンシューマーの開発を参照してください。

以下の表は、拡張ファンアウトへのデフォルトスループットを比較したものです。メッセージ伝播遅延は、ペイロードディスパッチ API (やなど) を使用して送信されたペイロードが、ペイロードを消費する API (PutRecord やなど) を介してコンシューマーアプリケーションに到達するまでにかかる時間 (ミリ秒単位 PutRecords) として定義されます。 GetRecords SubscribeToShard

特性	拡張ファンアウトを使用しない未登録コンシューマー	拡張ファンアウトを使用する登録済みコンシューマー
シャードの読み取りスループット	シャードあたり合計 2 MB/秒に固定されています。同じシャードから読み取るコンシューマーが複数ある場合、それらのすべてがこのスループットを共有します。コンシューマーがシャードから受け取るスループットの合計が 2 MB/秒を超えることはありません。	拡張ファンアウトを使用するコンシューマーが登録されるにつれてスケールされます。拡張ファンアウトを使用するように登録された各コンシューマーは、他のコンシューマーとは関係なく、シャードあたりに受け取る独自の読み取りスループットが最大 2 MB/秒です。
メッセージの伝播遅延	ストリームから読み取るコンシューマーが 1 つの場合は平均約 200 ms です。コンシューマーが 5 つの場	コンシューマーが 1 つまたは 5 つかによって、一般的に平均 70 ms です。

特性	拡張ファンアウトを使用しない未登録コンシューマー	拡張ファンアウトを使用する登録済みコンシューマー
	合、この平均は最大約 1,000 ms まで上がります。	
コスト	該当なし	データ取得コストおよびコンシューマー - シャード時間料金がかかります。詳細については、「Amazon Kinesis Data Streams の料金」を参照してください。
レコードの配信モデル	を使用して HTTP 経由でモデルをプルします。GetRecords	Kinesis Data Streams は、を使用して HTTP/2 経由でレコードをプッシュします。SubscribeToShard

トピック

- Kinesis コンソールでのデータビューワーの使用
- Kinesis コンソールでのデータストリームのクエリ
- 以下を使用してコンシューマーを開発する AWS Lambda
- Amazon Managed Service for Apache Flink を使用したコンシューマーの開発
- Amazon データFirehose を使用するコンシューマーの開発
- Kinesis Client Library の使用
- スループット共有カスタムコンシューマーの開発
- スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発
- コンシューマーを KCL 1.x から KCL 2.x に移行する
- Kinesis Data Streams からデータを読み取るためのその他の AWS サービスの使用
- サードパーティー統合の使用
- Kinesis データストリームコンシューマーのトラブルシューティング
- Amazon Kinesis Data Streams コンシューマー向けの高度なトピック

Kinesis コンソールでのデータビューワーの使用

Kinesis マネジメントコンソールのデータビューワーは、コンシューマーアプリケーションを開発することなく、データストリームの指定されたシャード内にあるデータレコードを表示することを可能にします。データビューワーを使用するには、以下のステップを実行してください。

1. AWS Management Console にサインインし、https://console.aws.amazon.com/kinesis にある Kinesis コンソールを開きます。
2. データビューワーで表示したいレコードがあるアクティブなデータストリームを選択してから、[データビューワー] タブを選択します。
3. 選択したアクティブなデータストリームの [データビューワー] タブで表示したいレコードがあるシャードを選択し、[開始位置] を選択してから、[レコードを取得] をクリックします。開始位置は、以下の値のいずれかに設定できます。

   - [シーケンス番号で]: シーケンス番号フィールドで指定されているシーケンス番号が示す位置からのレコードを表示します。
   - [シーケンス番号の後]: シーケンス番号フィールドで指定されているシーケンス番号が示す位置の直後からのレコードを表示します。
   - [タイムスタンプで]: タイムスタンプフィールドで指定されているタイムスタンプが示す位置からのレコードを表示します。
   - [水平トリム]: シャード内にある最後のトリミングされていないレコード、つまりシャード内で最も古いデータレコードでレコードを表示します。
   - [最新]: シャード内にある最新レコード直後のレコードを表示して、シャード内の最新データが常に読み取られるようにします。

   生成されたデータレコードで、指定されたシャード ID と開始位置に一致するものが、コンソールのレコードテーブルに表示されます。一度に表示できるレコードは、最大 50 件です。次のレコードセットを表示するには、[次へ] ボタンをクリックします。
4. 個々のレコードをクリックして、個別のウィンドウにそのレコードのペイロードを raw データまたは JSON 形式で表示します。

データビューアーで [レコードを取得] または [次へ] ボタンをクリックすると API が呼び出され、これが GetRecordsAPI の 1 秒あたり 5 トランザクションという制限に適用されることに注意してください。GetRecords

Kinesis コンソールでのデータストリームのクエリ

Kinesis データストリームコンソールの [データ分析] タブでは、SQL を使用してデータストリームをクエリできます。この機能を使用するには、以下の手順に従ってください。

1. AWS Management Console にサインインし、https://console.aws.amazon.com/kinesis にある Kinesis コンソールを開きます。
2. SQL でクエリするアクティブなデータストリームを選択し、[データ分析] タブを選択します。
3. データ分析タブでは、マネージド Apache Flink Studio ノートブックを使用してストリームの検査と可視化を行うことができます。Apache Zeppelin を使用すると、アドホック SQL クエリを実行してデータストリームを検査し、結果を数秒で表示できます。[データ分析] タブで [同意します] を選択し、[ノートブックの作成] を選択してノートブックを作成します。
4. ノートブックを作成したら、「Apache Zeppelin で開く」を選択します。これにより、ノートブックが新しいタブで開きます。ノートブックは SQL クエリを送信できるインタラクティブなインターフェースです。ストリームの名前を含むメモを選択します。
5. SELECT実行中のストリームのデータを出力するためのサンプルクエリが記載されたメモが表示されます。これにより、データストリームのスキーマを表示できます。
6. タンブリングやスライディングウィンドウなどの他のクエリを試すには、[データ分析] タブの [サンプルクエリを表示] を選択します。クエリをコピーし、データストリームのスキーマに合うように変更してから、Zeppelin ノートの新しい段落で実行します。

以下を使用してコンシューマーを開発する AWS Lambda

AWS Lambda 関数を使用してデータストリーム内のレコードを処理できます。 AWS Lambda は、サーバーのプロビジョニングや管理を行わずにコードを実行できるコンピューティングサービスです。コードは必要に応じて実行され、1 日あたり数件のリクエストから 1 秒あたり数千件のリクエストまで自動的にスケールされます。支払いは、使用したコンピューティング時間に対する料金のみになります。コードが実行されていないときに料金は発生しません。を使用すると AWS Lambda、事実上あらゆる種類のアプリケーションやバックエンドサービスのコードを、すべて管理なしで実行できます。可用性の高いコンピューティングインフラストラクチャでコードを実行するとともに、コンピューティングリソースのあらゆる管理も実行し、これにはサーバーとオペレーティングシステムのメンテナンス、キャパシティのプロビジョニングと自動スケーリング、およびコードのモニタリングとログ記録が含まれます。詳細については、「Amazon Kinesis AWS Lambda との併用」を参照してください。

トラブルシューティング情報については、「Kinesis Data Streams トリガーが Lambda 関数を呼び出せないのはなぜですか?」を参照してください。

Amazon Managed Service for Apache Flink を使用したコンシューマーの開発

SQL、Java、または Scala を使用した Kinesis ストリーム内のデータの処理と分析には、Amazon Managed Service for Apache Flink アプリケーションを使用することができます。Managed Service for Apache Flink アプリケーションは、リファレンスソースを使用したデータの強化、データの経時的な集約、または機械学習を使用したデータ異常の検出を行うことができます。その後、分析結果を別の Kinesis ストリーム、Firehose 配信ストリーム、または Lambda 関数に書き込むことができます。詳細については、「Managed Service for Apache Flink Developer Guide for SQL Applications」または「Managed Service for Apache Flink Developer Guide for Flink Applications」を参照してください。

Amazon データFirehose を使用するコンシューマーの開発

Firehose を使用して Kinesis ストリームからレコードを読み取り、処理することができます。Firehose は、Amazon S3、Amazon Redshift、アマゾンサービス、Splunk OpenSearch などの宛先にリアルタイムのストリーミングデータを配信するためのフルマネージドサービスです。Firehose は、Datadog、MongoDB、New Relic など、サポートされているサードパーティのサービスプロバイダーが所有するすべてのカスタム HTTP エンドポイントまたは HTTP エンドポイントもサポートします。データを宛先に配信する前に、データレコードを変換し、レコード形式を変換するように Firehose を設定することもできます。詳細については、「Kinesis Data Streams を使用した Firehose への書き込み」を参照してください。

Kinesis Client Library の使用

KDS データストリームからデータを処理できるカスタムコンシューマーアプリケーションを開発する方法の 1 つは、Kinesis Client Library (KCL) を使用することです。

トピック

- Kinesis Client Library (KCL) とは何ですか?
- KCL 使用可能なバージョン
- KCL の概念

- リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡する
- Java コンシューマーアプリケーションの同じ KCL 2.x で複数のデータストリームを処理する
- Kinesis クライアントライブラリと AWS Glue スキーマレジストリの使用

> **Note**
>
> KCL 1.x と KCL 2.x については、どちらも使用シナリオに応じて最新の KCL 1.x バージョンまたは KCL 2.x バージョンにアップグレードすることをお勧めします。KCL 1.x と KCL 2.x は、どちらも新しいリリースに伴って定期的に更新されています。これには、最新の依存関係パッチ、セキュリティパッチ、バグ修正、および下位互換性のある新機能が含まれます。詳細については、https://github.com/awslabs/ amazon-kinesis-client /release を参照してください。

Kinesis Client Library (KCL) とは何ですか?

KCL は、分散コンピューティングに関連する複雑なタスクの多くを処理することで、Kinesis Data Streams からデータを消費および処理するのに役立ちます。これには、複数のコンシューマーアプリケーションインスタンス間での負荷分散、コンシューマーアプリケーションインスタンスの障害に対する応答、処理済みのレコードのチェックポイント作成、リシャーディングへの対応が挙げられます。KCL はこれらのサブタスクをすべて処理するため、カスタムレコード処理ロジックの記述に集中できます。

KCL は AWS SDK で使用できる Kinesis Data Streams API とは異なることに注意してください。Kinesis Data Streams API では、Kinesis Data Streams の多くの機能 (ストリームの作成、リシャーディング、レコードの入力と取得など) を管理できます。KCL は、これらすべてのサブタスクの抽象化レイヤーを提供します。具体的には、コンシューマーアプリケーションのカスタムデータ処理ロジックに集中できます。Kinesis Data Streams API の詳細については、Amazon Kinesis API リファレンスを参照してください。

> **Important**
>
> KCL は Java ライブラリです。Java 以外の言語Support は、と呼ばれる多言語インタフェースを使用して提供されます。 MultiLangDaemonこのデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。たとえば、Python 用 KCL をインストールし、コンシューマアプリケーション全体を Python

で作成する場合でも、の理由により、システムに Java をインストールする必要があります。 MultiLangDaemonさらに MultiLangDaemon 、 AWS 接続先のリージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定がいくつかあります。詳細については GitHub、「KCL MultiLangDaemon プロジェクト」を参照してください。MultiLangDaemon

KCL は、レコード処理ロジックと Kinesis Data Streams の仲介として機能します。KCL は、次のタスクを実行します。

- データストリームに接続する
- データストリーム内のシャードを列挙する
- リースを使用して、ワーカーとシャードの関連付けを調整する
- レコードプロセッサで管理する各シャードのレコードプロセッサをインスタンス化する
- データストリームからデータレコードを取得する
- 対応するレコードプロセッサにレコードを送信する
- 処理されたレコードのチェックポイントを作成する
- ワーカーインスタンス数が変更されたとき、またはデータストリームがリシャード (シャードが分割またはマージされる) ときに、シャードワーカーの関連付け (リース) のバランスをとります。

KCL 使用可能なバージョン

現在、次のいずれかのサポートされているバージョンの KCL を使用して、カスタムコンシューマーアプリケーションを構築できます。

- KCL 1.x

  詳細については、KCL 1.x コンシューマーの開発を参照してください。
- KCL 2.x

  詳細については、KCL 2.x コンシューマーの開発を参照してください。

KCL 1.x または KCL 2.x のいずれかを使用して、共有スループットを使用するコンシューマーアプリケーションを構築できます。詳細については、KCL を使用したスループット共有カスタムコンシューマーの開発を参照してください。

専用スループット (拡張ファンアウトコンシューマー) を使用するコンシューマーアプリケーションを構築するには、KCL 2.x のみを使用できます。詳細については、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

KCL 1.x と KCL 2.x の違い、および KCL 1.x から KCL 2.x に移行する方法については、コンシューマーを KCL 1.x から KCL 2.x に移行するを参照してください。

KCL の概念

- KCL コンシューマーアプリケーション - KCL を使用してカスタムビルドされ、データストリームからレコードを取得して処理するように設計されたアプリケーション。
- コンシューマーアプリケーションインスタンス - KCL コンシューマーアプリケーションは、通常、障害時の調整とデータレコード処理の動的な負荷分散のために、1 つ以上のアプリケーションインスタンスが同時に実行され、分散されます。
- ワーカー - KCL コンシューマーアプリケーションインスタンスがデータの処理を開始するために使用する高レベルクラス。

> **Important**
>
> 各 KCL コンシューマーアプリケーションインスタンスには 1 つのワーカーがあります。

ワーカーは、シャードとリース情報の同期、シャード割り当ての追跡、シャードからのデータの処理など、さまざまなタスクを初期化し、監督します。ワーカーは、この KCL コンシューマーアプリケーションが処理するデータレコードを含むデータストリームの名前や、 AWS このデータストリームにアクセスするために必要な認証情報など、コンシューマーアプリケーションの構成情報を KCL に提供します。ワーカーは、その特定の KCL コンシューマーアプリケーションインスタンスを開始して、データストリームからレコードプロセッサにデータレコードを配信します。

> **Important**
>
> KCL 1.x では、このクラスはワーカーと呼ばれます。詳細については (これらは Java KCL リポジトリです)、https://github.com/awslabs/ amazon-kinesis-client /blob/v1.x/src/main/java/com/amazonaws/services/kinesis/ClientLibrary/lib/worker/Worker.java を参照してください。KCL 2.x では、このクラスはスケジューラと呼ばれます。KCL 2.x のスケジューラの目的は、KCL 1.x のワーカーの目的と同じです。KCL 2.x amazon-kinesis-clientのスケジューラクラスの詳細については、amazon-kinesis-client https://github.com/awslabs/ /

blob/master/ /src/main/java/software/amazon/kinesis/coordinator/Scheduler.java を参照してください。

- リース - ワーカーとシャード間のバインディングを定義するデータ。分散型 KCL コンシューマーアプリケーションは、リースを使用して、複数のワーカー間でデータレコード処理を分割します。いつでも、データレコードの各シャードは、leaseKey によって識別されるリースによって特定のワーカーにバインドされます。

  デフォルトでは、1 人のワーカーは (Worker 変数の値に応じて) 1 つ以上のリースを同時に保有できます。maxLeasesFor

  > **Important**
  >
  > すべてのワーカーは、データストリーム内の利用可能なすべてのシャードについて、利用可能なすべてのリースを保持すると競合します。しかし、一度に各リースを正常に保持できるのは1人のワーカーだけです。

  例えば、4 つのシャードを持つデータストリームを処理しているワーカー A を持つコンシューマーアプリケーションインスタンス A がある場合、ワーカー A はシャード 1、2、3、および 4 へのリースを同時に保持できます。ただし、2 つのコンシューマーアプリケーションインスタンス (ワーカー A とワーカー B を含む A と B) があり、これらのインスタンスが 4 つのシャードを持つデータストリームを処理している場合、ワーカー A とワーカー B はシャード 1 へのリースを同時に保持できません。あるワーカーは、このシャードのデータレコードの処理を停止する準備ができるまで、または失敗するまで、特定のシャードへのリースを保持します。あるワーカーがリースの保留を停止すると、別のワーカーがリースを引き取り、保留します。

  詳細については (これらは Java KCL リポジトリです)、KCL 1.x については https://github.com/awslabs/ amazon-kinesis-client /blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.java、KCL 2.x については https://github.com/awslabs/ /blob/master/ /src/main/java/software/amazon/kinesis/leases/Lease.java を参照してください。amazon-kinesis-client amazon-kinesis-client

- リーステーブル - KCL コンシューマーアプリケーションのワーカーによってリースおよび処理されている KDS データストリーム内のシャードを追跡するために使用される一意の Amazon DynamoDB テーブル。リーステーブルは、KCL コンシューマーアプリケーションの実行中に、データストリームからの最新のシャード情報と (ワーカー内およびすべてのワーカー間で) 同期を

維持する必要があります。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

- レコードプロセッサ - KCL コンシューマーアプリケーションがデータストリームから取得したデータを処理する方法を定義するロジック。実行時に、KCL コンシューマーアプリケーションインスタンスがワーカーをインスタンス化し、このワーカーは、リースを保持するシャードごとに 1 つのレコードプロセッサをインスタンス化します。

リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡する

トピック

- リーステーブルとは何ですか?
- スループット
- KDS データストリームのシャードとリーステーブルの同期方法

リーステーブルとは何ですか?

それぞれの Amazon Kinesis Data Streams アプリケーションに、KCLは、一意のリーステーブル (Amazon DynamoDB テーブルに保存されている) を使用して、KCL コンシューマーアプリケーションのワーカーによってリースおよび処理されている KDS データストリーム内のシャードを追跡します。

> **Important**
>
> KCL は、コンシューマーアプリケーションの名前を使用して、このコンシューマーアプリケーションが使用するリーステーブル名を作成します。したがって、各コンシューマーアプリケーション名は一意である必要があります。

コンシューマーアプリケーションの実行中に、Amazon DynamoDB コンソールを使用してリーステーブルを表示できます。

アプリケーションの起動時にKCL コンシューマーアプリケーションのリーステーブルが存在しない場合は、いずれかのワーカーがこのアプリケーションのリーステーブルを作成します。

> **Important**
>
> アカウントには、Kinesis Data Streams 自体に関連するコストに加えて、DynamoDB テーブルに関連するコストが発生します。

テーブルの各行は、コンシューマーアプリケーションのワーカーによって処理中のシャードを表します。KCL コンシューマーアプリケーションが 1 つのデータストリームのみを処理する場合、リーステーブルのハッシュキー `leaseKey` はシャード ID です。Java コンシューマーアプリケーションの同じ KCL 2.x で複数のデータストリームを処理する であれば、leaseKey の構造は次のようになります: `account-id:StreamName:streamCreationTimestamp:ShardId`。例えば `111111111:multiStreamTest-1:12345:shardId-000000000336` です。

シャード ID に加えて、各行には次のデータが含まれます。

- checkpoint: シャードの最新チェックポイントのシーケンス番号。この値はストリームのすべてのシャードで一意です。
- checkpointSubSequence番号:Kinesis プロデューサーライブラリの集計機能を使用する場合、これは Kinesis レコード内の個々のユーザーレコードを追跡するチェックポイントの拡張機能です。
- leaseCounter: ワーカーのリースが他のワーカーに保持されていることをワーカーが検出できるように、リースのバージョニングに使用されます。
- leaseKey: リースの固有識別子。各リースはデータストリームのシャードに固有であり、一度に 1 つのワーカーで保持されます。
- leaseOwner: このリースを保持しているワーカー。
- ownerSwitchesSinceチェックポイント:前回チェックポイントが作成されてから、このリースによってワーカーが何回変更されたか。
- parentShardId: 子シャードで処理を開始する前に、親シャードが完全に処理されていることを確認するために使用されます。これにより、レコードがストリームに入力されたのと同じ順序で処理されるようになります。
- hashrange: `PeriodicShardSyncManager` で使われて、定期的な同期を実行し、リーステーブルで欠落しているシャードを見つけ、必要に応じてリースを作成します。

> **Note**
>
> このデータは、KCL 1.14 および KCL 2.3 で始まるすべてのシャードのリーステーブルに存在します。`PeriodicShardSyncManager` の詳細およびリースとシャード間の定期的

な同期については、KDS データストリームのシャードとリーステーブルの同期方法 を参照してください。

- childshards: `LeaseCleanupManager` で使われて、子シャードの処理ステータスを確認し、親シャードをリーステーブルから削除できるかどうかを決定します。

> **Note**
>
> このデータは、KCL 1.14 および KCL 2.3 で始まるすべてのシャードのリーステーブルに存在します。

- shardID: シャードの ID。

> **Note**
>
> このデータは、Java コンシューマーアプリケーションの同じ KCL 2.x で複数のデータストリームを処理する である場合にのみリーステーブルに存在します。これは Java 用 KCL 2.x でのみサポートされており、Java の場合は KCL 2.3 以降で始まります。

- stream name 以下の形式のデータストリームの識別子: `account-id:StreamName:streamCreationTimestamp`。

> **Note**
>
> このデータは、Java コンシューマーアプリケーションの同じ KCL 2.x で複数のデータストリームを処理する である場合にのみリーステーブルに存在します。これは Java 用 KCL 2.x でのみサポートされており、Java の場合は KCL 2.3 以降で始まります。

スループット

Amazon Kinesis Data Streams アプリケーションでプロビジョニングされたスループットの例外が発生した場合は、DynamoDB テーブルのプロビジョニングされたスループットを増やす必要があります。KCL がテーブルを作成するときにプロビジョニングされるスループットは、1 秒あたりの読み込み 10 回、1 秒あたりの書き込み 10 回ですが、これがユーザーのアプリケーションで十分でない場合があります。例えば、Amazon Kinesis Data Streams アプリケーションが頻繁にチェックポイントを作成する場合や、多くのシャードで構成されるストリームを処理する場合は、より多くのスループットが必要になる可能性があります。

DynamoDB でプロビジョニングされたスループットについては、Amazon DynamoDB デベロッパーガイドの読み取り/書き込み容量モードおよびテーブルとデータの操作を参照してください。

KDS データストリームのシャードとリーステーブルの同期方法

KCL コンシューマーアプリケーションのワーカーは、リースを使用して特定のデータストリームからシャードを処理します。特定の時点でどのワーカーがどのシャードをリースしているかに関する情報は、リーステーブルに保存されます。リーステーブルは、KCL コンシューマーアプリケーションの実行中に、データストリームからの最新のシャード情報と同期を維持する必要があります。KCL は、コンシューマーアプリケーションのブートストラップ (コンシューマーアプリケーションの初期化時または再起動時)、および処理中のシャードが終了 (リシャーディング) に達するたびに、Kinesis Data Streams サービスから取得したシャード情報とリーステーブルを同期します。つまり、ワーカーまたは KCL コンシューマーアプリケーションは、最初のコンシューマーアプリケーションのブートストラップ中、およびコンシューマーアプリケーションでデータストリームリシャードイベントが発生するたびに、処理中のデータストリームと同期されます。

トピック

- KCL 1.0 - 1.13 と KCL 2.0 - 2.2 での同期
- KCL 2.x での同期、KCL 2.3 以降で始まる
- KCL 1.x での同期、KCL 1.14 以降で始まる

KCL 1.0 - 1.13 と KCL 2.0 - 2.2 での同期

KCL 1.0 - 1.13 および KCL 2.0 - 2.2 では、コンシューマーアプリケーションのブートストラップ、および各データストリームのリシャードイベント中に、KCL は、`ListShards` または `DescribeStream` 検出 API を呼び出して、Kinesis Data Streams サービスから取得したシャード情報とリーステーブルを同期します。上記のすべての KCL バージョンで、KCL コンシューマーアプリケーションの各ワーカーは、コンシューマーアプリケーションのブートストラップ中および各ストリームリシャードイベントでリース/シャード同期プロセスを実行するために、次の手順を完了します。

- 処理中のストリームのデータのすべてのシャードをフェッチします。
- リーステーブルからすべてのシャードリースをフェッチします。
- リーステーブルにリースのないオープンシャードをフィルターで除外します。
- 見つかったすべてのオープンシャードと、開いている親を持たない各オープンシャードについて反復処理します。

- 階層ツリーをその祖先パスを通過して、シャードが子孫であるかどうかを判断します。祖先シャードが処理されている場合 (リーステーブルに祖先シャードのリースエントリが存在する場合)、または祖先シャードを処理する必要がある場合 (例えば、初期位置がTRIM_HORIZONまたはAT_TIMESTAMP)、シャードは子孫と見なされます。
- コンテキスト内のオープンシャードが子孫である場合、KCL は初期位置に基づいてシャードをチェックポイントし、必要に応じて親のリースを作成します。

KCL 2.x での同期、KCL 2.3 以降で始まる

サポートされている最新バージョンの KCL 2.x (KCL 2.3) 以降では、ライブラリで同期プロセスに対する次の変更がサポートされるようになりました。これらのリース/シャード同期の変更により、KCL コンシューマーアプリケーションから Kinesis Data Streams サービスに対して実行される API コールの数が大幅に削減され、KCL コンシューマーアプリケーションのリース管理が最適化されます。

- アプリケーションのブートストラップ中に、リーステーブルが空の場合、KCL はListShard API のフィルタリングオプション (ShardFilter オプションのリクエストパラメータ) を使用して、ShardFilter パラメータで指定された時間に開いているシャードのスナップショットに対してのみリースを取得および作成します。ShardFilter パラメーターを使用すると、ListShards API の応答をフィルターで除外できます。ShardFilter パラメータの唯一の必須プロパティは Type です。KCL は Type フィルタープロパティとその次の有効な値を使用して、新しいリースを必要とする可能性のあるオープンシャードのスナップショットを識別して返します。
  - AT_TRIM_HORIZON - 応答には、TRIM_HORIZON で開いていたすべてのシャードが含まれます。
  - AT_LATEST - 応答には、データストリームの現在開いているシャードのみが含まれます。
  - AT_TIMESTAMP - 応答には、開始タイムスタンプが指定されたタイムスタンプ以下で、終了タイムスタンプが指定されたタイムスタンプ以上であるか、またはまだ開いているすべてのシャードが含まれます。

  ShardFilter は空のリーステーブルのリースを作成して、RetrievalConfig#initialPositionInStreamExtended で指定したシャードのスナップショットのリースを初期化するときに使用されます。

  の詳細については、ShardFilterを参照してください。https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html。

- すべてのワーカーがリース/シャード同期を実行して、データストリーム内の最新のシャードでリーステーブルを最新の状態に保つ代わりに、選択された単一のワーカーリーダーがリース/シャードの同期を実行します。
- KCL 2.3 は、GetRecords および SubscribeToShard APIのリターンパラメータ ChildShards を使用して、閉じたシャードに対して SHARD_END で発生するリース/シャード同期を実行します。これにより、KCL ワーカーは、処理が終了したシャードの子シャードに対してのみリースを作成できます。コンシューマーアプリケーション全体で共有する場合、リース/シャード同期のこの最適化では GetRecords API の ChildShards パラメータを使用します。専用スループット (拡張ファンアウト) コンシューマーアプリケーションの場合、リース/シャード同期のこの最適化では SubscribeToShard API の ChildShards パラメータを使用します。詳細については、、およびを参照してくださいGetRecords。SubscribeToShardsChildShard
- 上記の変更により、KCL の動作は、既存のすべてのシャードについて学習するすべてのワーカーのモデルから、各ワーカーが所有するシャードの子シャードについてのみ学習するワーカーのモデルに移行します。したがって、コンシューマーアプリケーションのブートストラップおよびリシャードイベント中に発生する同期に加えて、KCL は、リーステーブル内の潜在的なホールを特定するために、追加の定期的なシャード/リーススキャンを実行して (つまり、すべての新しいシャードについて学習する)、データストリームの完全なハッシュ範囲が処理されていることを確認し、必要に応じてそれらのリースを作成します。PeriodicShardSyncManagerは定期的なリース/シャードスキャンの実行を担当するコンポーネントです。

  KCL 2.3 の詳細についてはPeriodicShardSyncManager、https://github.com/awslabs/ amazon-kinesis-client /blob/master/ /src/main/java/software/amazon/kinesis/leases/ .java #L201-L213 amazon-kinesis-client を参照してください。LeaseManagementConfig

  KCL 2.3 では、新しい設定オプションを使用して、LeaseManagementConfig で PeriodicShardSyncManager を設定できるようになりました。

名前	デフォルト値	説明
leasesRecoveryAuditorExecutionFrequencyMillis	120000 (2 分)	リーステーブルで部分的なリースをスキャンする監査ジョブの頻度 (ミリ単位)。監査者がストリームのリースの

名前	デフォルト値	説明
		穴を検出すると、leasesRecoveryAuditorInconsistencyConfidenceThreshold に基づいてシャード同期がトリガーされます。
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	リーステーブル内のデータストリームのリースが矛盾しているかどうかを判断するための、定期的な監査ジョブの信頼しきい値。監査者がデータストリームに対して同じ不整合セットを何度も繰り返し検出すると、シャード同期がトリガーされます。

CloudWatch の状態を監視するための新しいメトリクスも出力されるようになりました。PeriodicShardSyncManager詳細については、「PeriodicShardSyncManager」を参照してください。

- HierarchicalShardSyncer への最適化を含めて、シャードの 1 つのレイヤーに対してのみリースを作成します。

KCL 1.x での同期、KCL 1.14 以降で始まる

サポートされている最新バージョンの KCL 1.x (KCL 1.14) 以降では、ライブラリで同期プロセスに対する次の変更がサポートされるようになりました。これらのリース/シャード同期の変更により、KCL コンシューマーアプリケーションから Kinesis Data Streams サービスに対して実行される API コールの数が大幅に削減され、KCL コンシューマーアプリケーションのリース管理が最適化されます。

- アプリケーションのブートストラップ中に、リーステーブルが空の場合、KCL はListShard API のフィルタリングオプション (ShardFilter オプションのリクエストパラメータ) を使用して、ShardFilter パラメータで指定された時間に開いているシャードのスナップショットに対してのみリースを取得および作成します。ShardFilter パラメーターを使用すると、ListShards API の応答をフィルターで除外できます。ShardFilter パラメータの唯一の必須プロパティは Type です。KCL は Type フィルタープロパティとその次の有効な値を使用して、新しいリースを必要とする可能性のあるオープンシャードのスナップショットを識別して返します。
  - AT_TRIM_HORIZON - 応答には、TRIM_HORIZON で開いていたすべてのシャードが含まれます。
  - AT_LATEST - 応答には、データストリームの現在開いているシャードのみが含まれます。
  - AT_TIMESTAMP - 応答には、開始タイムスタンプが指定されたタイムスタンプ以下で、終了タイムスタンプが指定されたタイムスタンプ以上であるか、またはまだ開いているすべてのシャードが含まれます。

  ShardFilter は空のリーステーブルのリースを作成して、KinesisClientLibConfiguration#initialPositionInStreamExtended で指定したシャードのスナップショットのリースを初期化するときに使用されます。

  の詳細については、ShardFilterを参照してください。https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html。
- すべてのワーカーがリース/シャード同期を実行して、データストリーム内の最新のシャードでリーステーブルを最新の状態に保つ代わりに、選択された単一のワーカーリーダーがリース/シャードの同期を実行します。
- KCL 1.14 は、GetRecords および SubscribeToShard APIのリターンパラメータ ChildShards を使用して、閉じたシャードに対して SHARD_END で発生するリース/シャード同期を実行します。これにより、KCL ワーカーは、処理が終了したシャードの子シャードに対してのみリースを作成できます。詳細については、「」GetRecordsと「」を参照してください。ChildShard

- 上記の変更により、KCL の動作は、既存のすべてのシャードについて学習するすべてのワーカーのモデルから、各ワーカーが所有するシャードの子シャードについてのみ学習するワーカーのモデルに移行します。したがって、コンシューマーアプリケーションのブートストラップおよびリシャードイベント中に発生する同期に加えて、KCL は、リーステーブル内の潜在的なホールを特定するために、追加の定期的なシャード/リーススキャンを実行して (つまり、すべての新しいシャードについて学習する)、データストリームの完全なハッシュ範囲が処理されていることを確認し、必要に応じてそれらのリースを作成します。PeriodicShardSyncManagerは定期的なリース/シャードスキャンの実行を担当するコンポーネントです。

  KinesisClientLibConfiguration#shardSyncStrategyType が ShardSyncStrategyType.SHARD_END に設定されると、PeriodicShardSync leasesRecoveryAuditorInconsistencyConfidenceThreshold は、シャード同期を強制するために、リーステーブル内のホールを含む連続スキャンの数のしきい値を決定するために使用されます。KinesisClientLibConfiguration#shardSyncStrategyType が ShardSyncStrategyType.PERIODIC に設定されると、leasesRecoveryAuditorInconsistencyConfidenceThreshold は無視されます。

  KCL 1.14 の詳細については、https://github.com/awslabs/ PeriodicShardSyncManager amazon-kinesis-client /blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/ .java #L987 KinesisClientLibConfiguration-L999 を参照してください。

  KCL 1.14 では、新しい設定オプションを使用して、LeaseManagementConfig で PeriodicShardSyncManager を設定できるようになりました。

名前	デフォルト値	説明
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	リーステーブル内のデータストリームのリースが矛盾しているかどうかを判断するための、定期的な監査ジョブの信頼しきい値。監査者がデータストリームに対して同じ不整合セットを何度も繰り返し検出すると、シャード同期がトリガーされます。

CloudWatch の状態を監視するための新しいメトリクスも出力されるようになりました。`PeriodicShardSyncManager`詳細については、「PeriodicShardSyncManager」を参照してください。

- KCL 1.14 は、遅延リースのクリーンアップもサポートするようになりました。リースは、`SHARD_END` に到達したとき、シャードがデータストリームの保持期間を過ぎて期限切れになったとき、またはリシャーディングオペレーションの結果として閉じられたとき、`LeaseCleanupManager` により非同期的に削除されます。

  新しい設定オプションを使用して、`LeaseCleanupManager` を設定できるようになりました。

名前	デフォルト値	説明
leaseCleanupIntervalミリス	1 分	リースクリーンアップスレッド

名前	デフォルト値	説明
		を実行する間隔。
completedLeaseCleanupIntervalMillis	5 分	リースが完了したかどうかをチェックする間隔。
garbageLeaseCleanupIntervalMillis	30 分	リースがガベージであるかどうかをチェックする間隔 (つまり、データストリームの保持期間を過ぎてトリミング)。

- `KinesisShardSyncer` への最適化を含めて、シャードの 1 つのレイヤーに対してのみリースを作成します。

Java コンシューマーアプリケーションの同じ KCL 2.x で複数のデータストリームを処理する

このセクションでは、複数のデータストリームを同時に処理できる KCL コンシューマーアプリケーションを作成できる KCL 2.x for Java における以下の変更点について説明します。

> **Important**
>
> マルチストリーム処理は、Java 用 KCL 2.x でのみサポートされており、Java の場合は KCL 2.3 以降で始まります。
> KCL 2.x を実装できる他の言語では、マルチストリーム処理はサポートされていません。
> マルチストリーム処理は KCL 1.x のどのバージョンでもサポートされていません。

- MultistreamTracker インターフェース

複数のストリームを同時に処理できるコンシューマーアプリケーションを構築するには、という新しいインターフェースを実装する必要がありますMultistreamTracker。このインターフェースには、KCL コンシューマーアプリケーションによって処理されるデータストリームとその設定のリストを返す streamConfigList メソッドが含まれています。処理中のデータストリームは、コンシューマーアプリケーションのランタイム中に変更できることに注意してください。streamConfigListは、処理するデータストリームの変更について学習するために KCL によって定期的に呼び出されます。

streamConfigListStreamConfigこのメソッドはリストにデータを入力します。

```
package software.amazon.kinesis.common;

import lombok.Data;
import lombok.experimental.Accessors;

@Data
@Accessors(fluent = true)
public class StreamConfig {
    private final StreamIdentifier streamIdentifier;
    private final InitialPositionInStreamExtended initialPositionInStreamExtended;
    private String consumerArn;
}
```

StreamIdentifier および InitialPositionInStreamExtended は必須フィールドですが、consumerArn は省略可能である点に注意してください。KCL 2.x を使用して拡張ファンアウトコンシューマーアプリケーションを実装する場合にのみ、consumerArn を提供する必要があります。

の詳細についてはStreamIdentifier、https://github.com/awslabs/ amazon-kinesis-client / blob/v2.5.8/ /src/main/java/software/amazon/kinesis/common/ amazon-kinesis-client .java #L129 を参照してください。StreamIdentifierを作成するにはStreamIdentifier、v2.5.0 以降で使用できるとからマルチストリームインスタンスを作成することをお勧めします。streamArn streamCreationEpochサポートされていない KCL v2.3 と v2.4 では、streamArmこの形式を使用してマルチストリームインスタンスを作成します。account-id:StreamName:streamCreationTimestampこの形式は次のメジャーリリースで廃止され、サポートされなくなります。

MultistreamTracker には、リーステーブル内の古いストリームのリースを削除するための戦略も含まれます(formerStreamsLeasesDeletionStrategy)。コンシューマーアプリケーションのランタイム中は、ストラテジーを変更できないことに注意してください。詳細については、https://github.com/awslabs/ amazon-kinesis-client /blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/ /src/main/java/software/amazon/kinesis/processor/.java amazon-kinesis-client を参照してください。FormerStreamsLeasesDeletionStrategy

- ConfigsBuilderは、KCL コンシューマーアプリケーションの構築時に使用する KCL 2.x のすべての構成設定を指定するために使用できるアプリケーション全体のクラスです。 ConfigsBuilderクラスがインターフェースをサポートするようになりました。MultistreamTracker ConfigsBuilderどちらも、レコードを消費するデータストリームの名前で初期化できます。

```
/**
    * Constructor to initialize ConfigsBuilder with StreamName
    * @param streamName
    * @param applicationName
    * @param kinesisClient
    * @param dynamoDBClient
    * @param cloudWatchClient
    * @param workerIdentifier
    * @param shardRecordProcessorFactory
    */
   public ConfigsBuilder(@NonNull String streamName, @NonNull String
applicationName,
           @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
dynamoDBClient,
           @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
workerIdentifier,
           @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
       this.appStreamTracker = Either.right(streamName);
       this.applicationName = applicationName;
       this.kinesisClient = kinesisClient;
       this.dynamoDBClient = dynamoDBClient;
       this.cloudWatchClient = cloudWatchClient;
       this.workerIdentifier = workerIdentifier;
       this.shardRecordProcessorFactory = shardRecordProcessorFactory;
   }
```

また、複数のストリームを同時に処理する KCL MultiStreamTracker コンシューマーアプリケーションを実装したい場合は、 ConfigsBuilder で初期化することもできます。

```
* Constructor to initialize ConfigsBuilder with MultiStreamTracker
     * @param multiStreamTracker
     * @param applicationName
     * @param kinesisClient
     * @param dynamoDBClient
     * @param cloudWatchClient
     * @param workerIdentifier
     * @param shardRecordProcessorFactory
     */
    public ConfigsBuilder(@NonNull MultiStreamTracker multiStreamTracker, @NonNull
 String applicationName,
            @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
 dynamoDBClient,
            @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
 workerIdentifier,
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.left(multiStreamTracker);
        this.applicationName = applicationName;
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

- KCL コンシューマーアプリケーションにマルチストリームサポートが実装されているため、アプリケーションのリーステーブルの各行に、このアプリケーションが処理する複数のデータストリームのシャード ID とストリーム名が含まれます。
- KCL コンシューマーアプリケーションのマルチストリームサポートが実装されている場合、leaseKey は次の構造を取ります: `account-id:StreamName:streamCreationTimestamp:ShardId`。例えば `111111111:multiStreamTest-1:12345:shardId-000000000336` です。

> **Important**
>
> KCL コンシューマーアプリケーションが 1 つのデータストリームのみを処理する場合、リーステーブルのハッシュキー leaseKey はシャード ID です。この既存の KCL コンシューマーアプリケーションを複数のデータストリームを処理するように再構成すると、リーステーブルが壊れます。マルチストリームサポートでは LeaseKey 構造体は次のようになっている必要があるためです: `account-id:StreamName:StreamCreationTimestamp:ShardId`。

Kinesis クライアントライブラリと AWS Glue スキーマレジストリの使用

Kinesis データストリームを AWS Glue スキーマレジストリと統合できます。 AWS Glue スキーマレジストリを使用すると、スキーマを一元的に検出、制御、および進化させながら、生成されたデータが登録されたスキーマによって継続的に検証されるようにできます。スキーマは、データレコードの構造と形式を定義します。スキーマは、信頼性の高いデータの公開、利用、または保存のための仕様をバージョニングしたものです。 AWS Glue Schema Registry を使用すると、 end-to-end ストリーミングアプリケーション内のデータ品質とデータガバナンスを向上させることができます。詳細については、AWS Glue スキーマレジストリを参照してください。この統合を設定する方法の 1 つは、Java で KCL を使用することです。

> **Important**
>
> 現在、Kinesis Data Streams と AWS Glue スキーマのレジストリ統合は、Java で実装された KCL 2.3 コンシューマーを使用する Kinesis データストリームでのみサポートされています。多言語サポートは提供されていません。KCL 1.0 コンシューマーはサポートされていません。KCL 2.3 より前の KCL 2.x コンシューマーはサポートされていません。

KCL を使用して KKinesis Data Streams とスキーマレジストリの統合を設定する方法の詳細については、「ユースケース:Amazon Kinesis データストリームと Glue スキーマレジストリの統合」の「KPL/KCL ライブラリを使用したデータの操作」セクションを参照してください。 AWS

スループット共有カスタムコンシューマーの開発

Kinesis Data Streams からデータを受け取る際に専用スループットを必要としない場合で、200 ms 以下の読み取り伝達遅延を必要としない場合は、以下のトピックで説明しているようにコンシューマーアプリケーションを構築できます。

トピック

- KCL を使用したスループット共有カスタムコンシューマーの開発
- AWS SDK for Java を使用した共有スループットでのカスタムコンシューマーの開発

専有スループットで Kinesis data streams からレコードを受信できるコンシューマーの構築の詳細については、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

KCL を使用したスループット共有カスタムコンシューマーの開発

共有スループットでカスタムコンシューマーアプリケーションを開発する方法の 1 つは、Kinesis Client Library (KCL) を使用することです。

トピック

- KCL 1.x コンシューマーの開発
- KCL 2.x コンシューマーの開発

KCL 1.x コンシューマーの開発

Kinesis Client Library (KCL) を使用して、Amazon Kinesis Data Streams のコンシューマーアプリケーションを開発することができます。KCL の詳細については、Kinesis Client Library (KCL) とは何ですか?を参照してください。

内容

- Java での Kinesis クライアントライブラリコンシューマーの開発
- ode.js での Kinesis Client Library コンシューマーの開発
- .NET での Kinesis Client Library コンシューマーの開発
- Python での Kinesis クライアントライブラリコンシューマーの開発
- Ruby での Kinesis Client Library コンシューマーの開発

Java での Kinesis クライアントライブラリコンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、Java について説明します。Javadoc リファレンスを表示するには、クラス AWS の Javadoc トピック AmazonKinesisClientを参照してください。

から Java KCL をダウンロードするには GitHub、Kinesis Client Library (Java) に移動します。Apache Maven で Java KCL を検索するには、KCL 検索結果のページを参照してください。Java KCL コンシューマーアプリケーションのサンプルコードを からダウンロードするには GitHub、 の KCL for Java サンプルプロジェクトページに移動します GitHub。

このサンプルアプリケーションは Apache Commons Logging を使用します。ログ設定は、`configure` ファイルで定義されている静的な `AmazonKinesisApplicationSample.java` メソッドを使用して変更できます。Log4j および AWS Java アプリケーションで Apache Commons ログ記録を使用する方法の詳細については、 AWS SDK for Java デベロッパーガイドの「LogLog4j でのログ記録」を参照してください。

Java で KCL コンシューマーアプリケーションを実装する場合は、次のタスクを完了する必要があります。

タスク

- I RecordProcessor メソッドを実装する
- I RecordProcessor インターフェイスの Class Factory を実装する
- ワーカーの作成
- 設定プロパティを変更する
- レコードプロセッサインターフェイスのバージョン 2 への移行

I RecordProcessor メソッドを実装する

KCL は現在、`IRecordProcessor` インターフェイスの 2 つのバージョンをサポートしています。元のインターフェイスは最初のバージョンの KCL で利用可能です。バージョン 2 は KCL バージョン 1.5.0 から利用可能です。両方のインターフェイスが完全にサポートされています。選択するインターフェイスは、お使いのシナリオの要件によって異なります。相違点をすべて確認するには、ローカルに作成した Javadocs、またはソースコードを参照してください。以下のセクションでは、使い始めの最小限の実装を概説します。

I RecordProcessor バージョン

- オリジナルインターフェイス (バージョン 1)
- 更新されたインターフェイス (バージョン 2)

オリジナルインターフェイス (バージョン 1)

オリジナルな `IRecordProcessor` interface (`package com.amazonaws.services.kinesis.clientlibrary.interfaces`) は、コンシューマーが実装しているべき次のレコードプロセッサメソッドを公開します。このサンプルでは、開始点として使用できる実装を提供しています (`AmazonKinesisApplicationSampleRecordProcessor.java` を参照してください)。

```
public void initialize(String shardId)
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
 checkpointer)
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

initialize

KCL は、レコードプロセッサがインスタンス化されると、`initialize` メソッドを呼び出し、特定のシャード ID をパラメータとして渡します。このレコードプロセッサはこのシャードのみを処理し、通常、その逆も真です (このシャードはこのレコード プロセッサによってのみ処理されます)。ただし、コンシューマーでは、データレコードが複数回処理される可能性に対応する必要があります。Kinesis Data Streams は少なくとも 1 回のセマンティクスを使用しています。これは、シャードから取得されたすべてのデータレコードが、コンシューマーのワーカーによって少なくとも 1 回処理されることを意味します。特定のシャードが複数のワーカーによって処理される可能性がある場合の詳細については、リシャーディング、スケーリング、並列処理を参照してください。

```
public void initialize(String shardId)
```

processRecords

KCL は、`processRecords` メソッドを呼び出し、`initialize(shardId)` メソッドで指定されたシャードのデータレコードのリストを渡します。レコードプロセッサは、コンシューマーのセマンティクスに従って、これらのレコードのデータを処理します。例えば、ワーカーはデータの変換を実行し、その結果を Amazon Simple Storage Service (Amazon S3) バケットに保存する場合があります。

```
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
 checkpointer)
```

データ自体に加えて、レコードにもシーケンス番号とパーティションキーが含まれます。ワーカーはデータを処理するときに、これらの値を使用できます。たとえば、ワーカーは、パーティションのキーの値に基づいて、データを格納する S3 バケットを選択できます。Record クラスは、レコードのデータ、シーケンス番号、およびパーティションキーへのアクセスを提供する次のメソッドを公開します。

```
record.getData()
record.getSequenceNumber()
record.getPartitionKey()
```

サンプルでは、プライベートメソッド processRecordsWithRetries に、ワーカーでレコードのデータ、シーケンス番号、およびパーティションキーにアクセスする方法を示すコードが含まれています。

Kinesis Data Streams では、シャードで既に処理されたレコードを追跡するためにレコードプロセッサが必要です。KCL は、チェックポインタ (IRecordProcessorCheckpointer) を processRecords に渡すことで、この追跡をユーザーに代わって処理します。レコードプロセッサは、このインターフェイスで checkpoint メソッドを呼び出し、シャード内のレコードの処理の進行状況を KCL に知らせます。ワーカーでエラーが発生すると、KCL はこの情報を使用して、処理されたことが分かっている最後のレコードからシャードの処理を再開します。

分割または結合オペレーションの場合、KCL は、元のシャードのプロセッサが checkpoint を呼び出して元のシャードの処理がすべて完了したことを通知するまで、新しいシャードの処理を開始しません。

パラメータを渡さないと、checkpoint への呼び出しは、レコードプロセッサに最後のレコードを渡した時点までのすべてのレコードが処理済みであることを意味すると KCL で見なされます。したがって、レコードプロセッサは、渡されたリストにあるすべてのレコードの処理が完了した場合にのみ、checkpoint を呼び出す必要があります。レコードプロセッサは、checkpoint の各呼び出しで processRecords を呼び出す必要はありません。たとえば、プロセッサは、checkpoint を 3 回呼び出すたびに、processRecords を呼び出すことができます。オプションでレコードの正確なシーケンス番号をパラメータとして checkpoint に指定できます。この場合、KCL は、すべてのレコードがそのレコードまで処理されたと見なします。

このサンプルでは、プライベートメソッド checkpoint で、適切な例外処理と再試行のロジックを使用する IRecordProcessorCheckpointer.checkpoint を呼び出す方法を示しています。

KCL は、processRecords を使用して、データレコードの処理から発生するすべての例外を処理します。例外が processRecords からスローされた場合、KCL は、例外発生前に渡されたデータレコードをスキップします。つまり、これらのレコードは、例外をスローしたレコードプロセッサ、またはコンシューマーの他のレコードプロセッサに再送信されません。

shutdown

KCL は、処理が終了した場合 (シャットダウンの理由は TERMINATE) またはワーカーが応答していない場合 (シャットダウンの理由は ZOMBIE)、shutdown メソッドを呼び出します。

```
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

シャードが分割または結合されたか、ストリームが削除されたため、レコードプロセッサがシャードからこれ以上レコードを受信しない場合は、処理が終了します。

KCL はまた、IRecordProcessorCheckpointer インターフェイスを shutdown に渡します。シャットダウンの理由が TERMINATE である場合、レコードプロセッサはすべてのデータレコードの処理を終了し、このインターフェイスの checkpoint メソッドを呼び出します。

更新されたインターフェイス (バージョン 2)

更新された IRecordProcessor interface (package com.amazonaws.services.kinesis.clientlibrary.interfaces.v2) は、コンシューマーが実装しているべき次のレコードプロセッサメソッドを公開します。

```
void initialize(InitializationInput initializationInput)
void processRecords(ProcessRecordsInput processRecordsInput)
void shutdown(ShutdownInput shutdownInput)
```

コンテナオブジェクトのメソッドの呼び出しで、インターフェイスのオリジナルバージョンのすべての引数にアクセスできます。たとえば、processRecords() でレコードのリストを取得には、processRecordsInput.getRecords() が使用できます。

このインターフェイスのバージョン 2 (KCL 1.5.0 以降) では、オリジナルインターフェースで提供される入力に加えて次の新しい入力が使用できます。

シーケンス番号の開始

InitializationInput オペレーションへ渡される initialize() オブジェクトでは、開始シーケンス番号はレコードプロセッサのインスタンスに配信されるレコードです。このシーケン

ス番号は、同じシャードで処理されたレコードプロセッサインスタンスの最後のチェックポイントです。これは、アプリケーションでこの情報が必要になる場合のために提供されます。

保留チェックポイントシーケンス番号

initialize() オペレーションへ渡される InitializationInputオブジェクトの保留チェックポイントシーケンス番号 (ある場合) とは、前のレコードプロセッサインスタンスが停止する前にコミットできなかったものを示します。

I RecordProcessor インターフェイスの Class Factory を実装する

レコードプロセッサのメソッドを実装するクラスのファクトリも実装する必要があります。コンシューマーは、ワーカーをインスタンス化するときに、このファクトリへの参照を渡します。

サンプルでは、オリジナルのレコードプロセッサインターフェースを使用した、AmazonKinesisApplicationSampleRecordProcessorFactory.java ファイルのファクトリクラスを実装します。クラスファクトリでバージョン 2 レコードプロセッサを作成する場合には、com.amazonaws.services.kinesis.clientlibrary.interfaces.v2 とい名のパッケージを使用してください。

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {
    /**
    * Constructor.
    */
    public SampleRecordProcessorFactory() {
        super();
    }
    /**
    * {@inheritDoc}
    */
    @Override
    public IRecordProcessor createProcessor() {
        return new SampleRecordProcessor();
    }
}
```

ワーカーの作成

I RecordProcessor メソッドを実装するで説明しているように、KCL レコードプロセッサには選択できる 2 バージョンがあり、どちらを選ぶかでワーカーの作成方法に影響します。オリジナルレコードプロセッサインターフェイスは、次のコードストラクチャを使用してワーカーを作成します。

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker(recordProcessorFactory, config);
```

レコード プロセッサインターフェイスのバージョン 2 では、Worker.Builder を使用してワーカを作成でき、どのコンストラクタを使うかや引数の順序を考慮する必要はありません。更新されたレコードプロセッサインターフェイスは、次のコードストラクチャを使用してワーカーを作成します。

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

設定プロパティを変更する

このサンプルでは、設定プロパティのデフォルト値を提供します。ワーカーのこの設定データは KinesisClientLibConfiguration オブジェクトにまとめられています。ワーカーをインスタンス化する呼び出しで、このオブジェクトと IRecordProcessor のクラスファクトリへの参照が渡されます。Java の properties ファイルを使用してこれらのプロパティを独自の値にオーバーライドできます (AmazonKinesisApplicationSample.java を参照してください)。

アプリケーション名

KCL には、複数のアプリケーション間、および同じリージョン内の Amazon DynamoDB テーブル間で一意のアプリケーション名が必要です。次のようにアプリケーション名の設定値を使用します。

- このアプリケーション名と関連付けられたすべてのワーカーは、連係して同じストリームを処理していると見なされます。これらのワーカーは複数のインスタンスに分散している場合もあります。同じアプリケーションコードの追加のインスタンスを実行するときに、アプリケーション名が異なる場合、KCL は 2 番目のインスタンスを、同じストリームで動作するまったく別のアプリケーションと見なします。
- KCL はアプリケーション名を使用して DynamoDB テーブルを作成し、このテーブルを使用してアプリケーションの状態情報 (チェックポイントやワーカーとシャードのマッピングなど) を保存します。各アプリケーションには、それぞれ DynamoDB テーブルがあります。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

認証情報を設定する

デフォルトの AWS 認証情報プロバイダーチェーンの認証情報プロバイダーの 1 つが認証情報を利用できるようにする必要があります。例えば、EC2 インスタンスでコンシューマーを実行している場合は、IAM ロールでインスタンスを起動することをお勧めします。この IAM ロールに関連付けられた許可を反映する AWS 認証情報は、インスタンスメタデータを通じて、インスタンス上のアプリケーションで使用できるようになります。これは、EC2 インスタンスで実行されるコンシューマーの認証情報を管理するための最も安全な方法です。

サンプルアプリケーションは、最初にインスタンスメタデータから IAM 認証情報を取得しようとします。

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

サンプルアプリケーションは、インスタンスメタデータから認証情報を取得できない場合、properties ファイルから認証情報を取得しようとします。

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

インスタンスメタデータの詳細については、「Amazon EC2 ユーザーガイド」の「インスタンスメタデータ」を参照してください。 Amazon EC2

複数のインスタンスへのワーカー ID の使用

サンプルの初期化コードは、次のコードスニペットに示すように、ローカルコンピュータ名にグローバルー意識別子を追加して、ワーカーの ID (workerId) を作成します。このアプローチによって、1 台のコンピュータでコンシューマーアプリケーションの複数のインスタンスを実行するシナリオに対応できます。

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +
 UUID.randomUUID();
```

レコードプロセッサインターフェイスのバージョン 2 への移行

オリジナルインターフェースで使われるコードを移行するためには、上記のステップに加えて、次の手順が必要となります。

1. レコードプロセッサのクラスを変更して、バージョン 2 レコードプロセッサインターフェイスにインポートします。

```
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

2. コンテナオブジェクトで get メソッドを使用するには、入力するリファレンスを変更します。たとえば、shutdown() オペレーションで、checkpointer を shutdownInput.getCheckpointer() に変更します。
3. レコードプロセッサのファクトリークラスを変更して、バージョン 2 レコードプロセッサファクトリーインターフェイスにインポートします。

```
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
```

4. ワーカーのコンストラクチャを変更して、Worker.Builder を使います。例:

```
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

ode.js での Kinesis Client Library コンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、Node.js について説明します。

KCL は Java ライブラリです。Java 以外の言語のサポートは、 と呼ばれる多言語インターフェイスを使用して提供されますMultiLangDaemon。このデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。したがって、Node.js 用の KCL をインストールし、コンシューマーアプリケーション全体を Node.js に書き込む場合、 のためにシステムに Java をインストールする必要があります MultiLangDaemon。さらに、接続先の AWS リージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定が MultiLangDaemon いくつかあります。 MultiLangDaemon の の詳細については GitHub、KCL MultiLangDaemon プロジェクトページを参照してください。

から Node.js KCL をダウンロードするには GitHub、Kinesis Client Library (Node.js) に移動します。

サンプルコードのダウンロード

Node.js の KCL で使用可能な 2 つのサンプルコードがあります。

- 基本サンプル

  Node.js で KCL コンシューマーアプリケーションを構築する方法の基本を説明する次のセクションで使用されます。

- click-stream-sample

  基本サンプルコードを理解したあとの、やや上級で実際のシナリオを使用したサンプル。このサンプルはここでは説明しませんが、詳細を説明した README ファイルがあります。

Node.js で KCL コンシューマーアプリケーションを実装する場合は、次のタスクを完了する必要があります。

タスク

- レコードプロセッサを実装する
- 設定プロパティを変更する

レコードプロセッサを実装する

KCL for Node.js を使用した最もシンプルなコンシューマーは、`recordProcessor` 関数を実装する必要があります。この関数には、`initialize`、`processRecords`、および `shutdown` の各関数が含まれます。このサンプルでは、開始点として使用できる実装を提供しています (`sample_kcl_app.js` を参照してください)。

```
function recordProcessor() {
  // return an object that implements initialize, processRecords and shutdown
 functions.}
```

initialize

レコードプロセッサが起動すると、KCL は `initialize` 関数を呼び出します。このレコードプロセッサは `initializeInput.shardId` として渡されるシャード ID のみを処理し、通常、その逆も真です (このシャードはこのレコードプロセッサによってのみ処理されます)。ただし、コンシューマーでは、データレコードが複数回処理される可能性に対応する必要があります。これは、Kinesis Data Streams は少なくとも 1 回のセマンティクスを使用しているからです。つまり、シャードから取得されたすべてのデータレコードが、コンシューマーのワーカーによって少なくとも 1 回処理されることを意味します。特定のシャードが複数のワーカーによって処理される可能性がある場合の詳細については、リシャーディング、スケーリング、並列処理を参照してください。

```
initialize: function(initializeInput, completeCallback)
```

processRecords

KCL は、この関数を呼び出すために initialize 関数に指定したシャードのデータレコードのリストが含まれている入力を使用します。実装するレコードプロセッサは、コンシューマーのセマンティクスに従って、これらのレコードのデータを処理します。例えば、ワーカーはデータの変換を実行し、その結果を Amazon Simple Storage Service (Amazon S3) バケットに保存する場合があります。

```
processRecords: function(processRecordsInput, completeCallback)
```

データ自体に加えて、レコードにもシーケンス番号とパーティションキーが含まれ、ワーカーはデータを処理するときに、これらを使用できます。たとえば、ワーカーは、パーティションのキーの値に基づいて、データを格納する S3 バケットを選択できます。record ディクショナリは、レコードのデータ、シーケンス番号、およびパーティションキーにアクセスする次のキーと値のペアを公開します。

```
record.data
record.sequenceNumber
record.partitionKey
```

データは Base64 でエンコードされていることに注意してください。

基本サンプルでは、関数 processRecords に、ワーカーでレコードのデータ、シーケンス番号、およびパーティションキーにアクセスする方法を示すコードが含まれています。

Kinesis Data Streams では、シャードで既に処理されたレコードを追跡するためにレコードプロセッサが必要です。KCL は、processRecordsInput.checkpointer として渡した checkpointer オブジェクトを使用して、この追跡を処理します。レコードプロセッサは、checkpointer.checkpoint 関数を呼び出して、シャード内のレコードの処理の進行状況を KCL に知らせます。ワーカーでエラーが発生した場合、シャードの処理を再開するときに、処理されたことが分かっている最後のレコードから再開するように、KCL はこの情報を使用します。

分割または結合オペレーションの場合、KCL は、元のシャードのプロセッサが checkpoint を呼び出して元のシャードの処理がすべて完了したことを通知するまで、新しいシャードの処理を開始しません。

checkpoint 関数にシーケンス番号を渡さないと、checkpoint への呼び出しは、レコードプロセッサに最後のレコードを渡した時点までのすべてのレコードが処理済みであることを意味すると KCL で見なされます。したがって、レコードプロセッサは、渡されたリストにあるすべてのレコードの処理が完了した場合にのみ、checkpoint を呼び出す必要があります。レコードプロセッサは、checkpoint の各呼び出しで processRecords を呼び出す必要はありません。たとえば、プロセッサは checkpoint を 3 回の呼び出しごとに呼び出したり、レコードプロセッサの外部イベント (実装したカスタムの認証または検証サービスなど) で呼び出したりできます。

オプションでレコードの正確なシーケンス番号をパラメータとして checkpoint に指定できます。この場合、KCL は、そのレコードまでのすべてのレコードだけが処理されたと見なします。

基本サンプルアプリケーションでは、checkpointer.checkpoint 関数の最もシンプルな呼び出しを示します。関数のこの時点でコンシューマーに必要な他のチェックポイントロジックを追加できます。

shutdown

KCL は、処理が終了した場合 (shutdownInput.reason は TERMINATE) またはワーカーが応答していない場合 (shutdownInput.reason は ZOMBIE)、shutdown 関数を呼び出します。

```
shutdown: function(shutdownInput, completeCallback)
```

シャードが分割または結合されたか、ストリームが削除されたため、レコードプロセッサがシャードからこれ以上レコードを受信しない場合は、処理が終了します。

また、KCL は、shutdownInput.checkpointer オブジェクトも shutdown に渡します。シャットダウンの理由が TERMINATE である場合、レコードプロセッサがすべてのデータレコードの処理を終了したことを確認し、このインターフェイスの checkpoint 関数を呼び出します。

設定プロパティを変更する

このサンプルでは、設定プロパティのデフォルト値を提供します。これらのプロパティを独自の値にオーバーライドできます (基本サンプルの sample.properties を参照してください)。

アプリケーション名

KCL には、複数のアプリケーション間、および同じリージョン内の Amazon DynamoDB テーブル間で一意のアプリケーションが必要です。次のようにアプリケーション名の設定値を使用します。

- このアプリケーション名と関連付けられたすべてのワーカーは、連係して同じストリームを処理していると見なされます。これらのワーカーは複数のインスタンスに分散している場合もあります。

同じアプリケーションコードの追加のインスタンスを実行するときに、アプリケーション名が異なる場合、KCL は 2 番目のインスタンスを、同じストリームで動作するまったく別のアプリケーションと見なします。

- KCL はアプリケーション名を使用して DynamoDB テーブルを作成し、このテーブルを使用してアプリケーションの状態情報 (チェックポイントやワーカーとシャードのマッピングなど) を保存します。各アプリケーションには、それぞれ DynamoDB テーブルがあります。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

認証情報を設定する

デフォルトの AWS 認証情報プロバイダーチェーンの認証情報プロバイダーの 1 つが認証情報を利用できるようにする必要があります。`AWSCredentialsProvider` プロパティを使用して認証情報プロバイダーを設定できます。`sample.properties` ファイルでは、デフォルトの認証情報プロバイダーチェーンのいずれかの認証情報プロバイダーに対して、ユーザーの認証情報を使用可能にする必要があります。Amazon EC2 インスタンスでコンシューマーを実行している場合は、この IAM ロールに関連付けられたアクセス許可を反映する IAM role. AWS credentials を使用してインスタンスを設定することをお勧めします。この IAM ロールは、インスタンスメタデータを介してインスタンス上のアプリケーションで使用できるようになります。これは、EC2 インスタンスで実行されるコンシューマーアプリケーションの認証情報を管理するための最も安全な方法です。

次の例では、KCL を設定し、`sample_kcl_app.js` で指定されているレコードプロセッサを使用して`kclnodejssample`という Kinesis Data Streams を処理します。

```
# The Node.js executable script
executableName = node sample_kcl_app.js
# The name of an Amazon Kinesis stream to process
streamName = kclnodejssample
# Unique KCL application name
applicationName = kclnodejssample
# Use default AWS credentials provider chain
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain
# Read from the beginning of the stream
initialPositionInStream = TRIM_HORIZON
```

.NET での Kinesis Client Library コンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、.NET について説明します。

KCL は Java ライブラリです。Java 以外の言語のサポートは、 と呼ばれる多言語インターフェイスを使用して提供されますMultiLangDaemon。このデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。したがって、.NET 用の KCL をインストールし、コンシューマーアプリを .NET に完全に記述しても、 のためにシステムに Java がインストールされている必要があります MultiLangDaemon。さらに、接続先の AWS リージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定が MultiLangDaemon いくつかあります。 MultiLangDaemon の の詳細については GitHub、KCL MultiLangDaemon プロジェクトページを参照してください。

.NET KCL を からダウンロードするには GitHub、Kinesis Client Library (.NET) に移動します。.NET KCL コンシューマーアプリケーションのサンプルコードをダウンロードするには、「」の「.NET サンプルコンシューマープロジェクト」ページを参照してください GitHub。

.NET で KCL コンシューマーアプリケーションを実装する場合は、次のタスクを完了する必要があります。

タスク

- I RecordProcessor クラスメソッドを実装する
- 設定プロパティを変更する

I RecordProcessor クラスメソッドを実装する

コンシューマーでは、`IRecordProcessor` の次のメソッドを実装する必要があります。出発点として使用できる実装がサンプルコンシューマーに提供されています (`SampleRecordProcessor` の `SampleConsumer/AmazonKinesisSampleConsumer.cs` クラスを参照してください)。

```
public void Initialize(InitializationInput input)
public void ProcessRecords(ProcessRecordsInput input)
public void Shutdown(ShutdownInput input)
```

Initialize

KCL は、レコードプロセッサがインスタンス化されると、このメソッドを呼び出して input パラメータの特定のシャード ID (input.ShardId) を渡します。このレコードプロセッサはこのシャードのみを処理し、通常、その逆も真です (このシャードはこのレコード プロセッサによってのみ処理されます)。ただし、コンシューマーでは、データレコードが複数回処理される可能性に対応する必要があります。これは、Kinesis Data Streams は少なくとも 1 回のセマンティクスを使用しているからです。つまり、シャードから取得されたすべてのデータレコードが、コンシューマーのワーカーによって少なくとも 1 回処理されることを意味します。特定のシャードが複数のワーカーによって処理される可能性がある場合の詳細については、リシャーディング、スケーリング、並列処理を参照してください。

```
public void Initialize(InitializationInput input)
```

ProcessRecords

KCL は、このメソッドを呼び出し、Initialize メソッドで指定されたシャードの input パラメータ (input.Records) にあるデータレコードのリストを渡します。実装するレコードプロセッサは、コンシューマーのセマンティクスに従って、これらのレコードのデータを処理します。例えば、ワーカーはデータの変換を実行し、その結果を Amazon Simple Storage Service (Amazon S3) バケットに保存する場合があります。

```
public void ProcessRecords(ProcessRecordsInput input)
```

データ自体に加えて、レコードにもシーケンス番号とパーティションキーが含まれます。ワーカーはデータを処理するときに、これらの値を使用できます。たとえば、ワーカーは、パーティションのキーの値に基づいて、データを格納する S3 バケットを選択できます。Record クラスは以下を公開し、レコードのデータ、シーケンス番号、およびパーティションキーのアクセスを可能にします。

```
byte[] Record.Data
string Record.SequenceNumber
string Record.PartitionKey
```

サンプルでは、メソッド ProcessRecordsWithRetries に、ワーカーでレコードのデータ、シーケンス番号、およびパーティションキーにアクセスする方法を示すコードが含まれています。

Kinesis Data Streams では、シャードで既に処理されたレコードを追跡するためにレコードプロセッサが必要です。KCL は、Checkpointer オブジェクトを ProcessRecords に渡すことで、この追跡をユーザーに代わって処理します (input.Checkpointer)。レコードプロセッサは、Checkpointer.Checkpoint メソッドを呼び出して、シャード内のレコード処理の進行状況

を KCL に知らせます。ワーカーでエラーが発生すると、KCL はこの情報を使用して、処理されたことが分かっている最後のレコードからシャードの処理を再開します。

分割または結合オペレーションの場合、KCL は、元のシャードのプロセッサが Checkpointer.Checkpoint を呼び出して元のシャードの処理がすべて完了したことを通知するまで、新しいシャードの処理を開始しません。

パラメータを渡さないと、Checkpointer.Checkpoint への呼び出しは、レコードプロセッサに最後のレコードを渡した時点までのすべてのレコードが処理済みであることを意味すると KCL で見なされます。したがって、レコードプロセッサは、渡されたリストにあるすべてのレコードの処理が完了した場合にのみ、Checkpointer.Checkpoint を呼び出す必要があります。レコードプロセッサは、Checkpointer.Checkpoint の各呼び出しで ProcessRecords を呼び出す必要はありません。たとえば、プロセッサは、3 回または 4 回呼び出すたびに、Checkpointer.Checkpoint を呼び出すことができます。オプションでレコードの正確なシーケンス番号をパラメータとして Checkpointer.Checkpoint に指定できます。この場合、KCL は、レコード処理がそのレコードまで完了したと見なします。

サンプルでは、プライベートメソッド Checkpoint(Checkpointer checkpointer) で、適切な例外処理と再試行のロジックを使用する Checkpointer.Checkpoint メソッドを呼び出す方法を示しています。

KCL for .NET では、例外を処理する方法が他の KCL 言語ライブラリとは異なり、データレコードの処理から発生した例外を扱いません。ユーザーコードからの例外がキャッチされないと、プログラムがクラッシュします。

シャットダウン

KCL は、処理が終了した場合 (シャットダウンの理由は TERMINATE) またはワーカーが応答していない場合 (シャットダウンの input.Reason の値は ZOMBIE)、Shutdown メソッドを呼び出します。

```
public void Shutdown(ShutdownInput input)
```

シャードが分割または結合されたか、ストリームが削除されたため、レコードプロセッサがシャードからこれ以上レコードを受信しない場合は、処理が終了します。

また、KCL は、Checkpointer オブジェクトも shutdown に渡します。シャットダウンの理由が TERMINATE である場合、レコードプロセッサはすべてのデータレコードの処理を終了し、このインターフェイスの checkpoint メソッドを呼び出します。

設定プロパティを変更する

このサンプルコンシューマーでは、設定プロパティのデフォルト値を提供します。これらのプロパティを独自の値にオーバーライドできます (`SampleConsumer/kcl.properties` を参照してください)。

アプリケーション名

KCL には、複数のアプリケーション間、および同じリージョン内の Amazon DynamoDB テーブル間で一意のアプリケーションが必要です。次のようにアプリケーション名の設定値を使用します。

- このアプリケーション名と関連付けられたすべてのワーカーは、連係して同じストリームを処理していると見なされます。これらのワーカーは複数のインスタンスに分散している場合もあります。同じアプリケーションコードの追加のインスタンスを実行するときに、アプリケーション名が異なる場合、KCL は 2 番目のインスタンスを、同じストリームで動作するまったく別のアプリケーションと見なします。
- KCL はアプリケーション名を使用して DynamoDB テーブルを作成し、このテーブルを使用してアプリケーションの状態情報 (チェックポイントやワーカーとシャードのマッピングなど) を保存します。各アプリケーションには、それぞれ DynamoDB テーブルがあります。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

認証情報を設定する

デフォルトの AWS 認証情報プロバイダーチェーンの認証情報プロバイダーの 1 つが認証情報を利用できるようにする必要があります。`AWSCredentialsProvider` プロパティを使用して認証情報プロバイダーを設定できます。sample.properties では、デフォルトの認証情報プロバイダーチェーンのいずれかの認証情報プロバイダーに対して、ユーザーの認証情報を使用可能にする必要があります。EC2 インスタンスでコンシューマーアプリケーションを実行している場合は、IAM ロールでインスタンスを設定することをお勧めします。この IAM ロールに関連付けられた許可を反映する AWS 認証情報は、インスタンスメタデータを通じて、インスタンス上のアプリケーションで使用できるようになります。これは、EC2 インスタンスで実行されるコンシューマーの認証情報を管理するための最も安全な方法です。

サンプルのプロパティファイルでは、 で指定されているレコードプロセッサを使用してwordsというKinesis data stream を処理するように KCL を設定します。

Python での Kinesis クライアントライブラリコンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、Python について説明します。

KCL は Java ライブラリです。Java 以外の言語のサポートは、 と呼ばれる多言語インターフェイスを使用して提供されますMultiLangDaemon。このデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。したがって、Python 用 KCL をインストールし、コンシューマーアプリケーション全体を Python で記述しても、 のためにシステムに Java をインストールする必要があります MultiLangDaemon。さらに、接続先の AWS リージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定が MultiLangDaemon いくつかあります。 MultiLangDaemon の の詳細については GitHub、KCL MultiLangDaemon プロジェクトページを参照してください。

から Python KCL をダウンロードするには GitHub、Kinesis Client Library (Python) に移動します。Python KCL コンシューマーアプリケーションのサンプルコードをダウンロードするには、 の KCL for Python サンプルプロジェクトページに移動します GitHub。

Python で KCL コンシューマーアプリケーションを実装する場合は、次のタスクを完了する必要があります。

タスク

- RecordProcessor クラスメソッドを実装する
- 設定プロパティを変更する

RecordProcessor クラスメソッドを実装する

`RecordProcess` クラスでは、`RecordProcessorBase` を拡張して次のメソッドを実装する必要があります。このサンプルでは、開始点として使用できる実装を提供しています (`sample_kclpy_app.py` を参照してください)。

```
def initialize(self, shard_id)
def process_records(self, records, checkpointer)
def shutdown(self, checkpointer, reason)
```

initialize

KCL は、レコードプロセッサがインスタンス化されると、initialize メソッドを呼び出し、特定のシャード ID をパラメータとして渡します。このレコードプロセッサはこのシャードのみを処理し、通常、その逆も真です (このシャードはこのレコード プロセッサによってのみ処理されます)。ただし、コンシューマーでは、データレコードが複数回処理される可能性に対応する必要があります。これは、Kinesis Data Streams は少なくとも 1 回のセマンティクスを使用しているからです。つまり、シャードから取得されたすべてのデータレコードが、コンシューマーのワーカーによって少なくとも 1 回処理されることを意味します。特定のシャードが複数のワーカーによって処理される可能性がある場合の詳細については、リシャーディング、スケーリング、並列処理を参照してください。

```
def initialize(self, shard_id)
```

process_records

KCL は、このメソッドを呼び出し、initialize メソッドで指定されたシャードのデータレコードのリストを渡します。実装するレコードプロセッサは、コンシューマーのセマンティクスに従って、これらのレコードのデータを処理します。例えば、ワーカーはデータの変換を実行し、その結果を Amazon Simple Storage Service (Amazon S3) バケットに保存する場合があります。

```
def process_records(self, records, checkpointer)
```

データ自体に加えて、レコードにもシーケンス番号とパーティションキーが含まれます。ワーカーはデータを処理するときに、これらの値を使用できます。たとえば、ワーカーは、パーティションのキーの値に基づいて、データを格納する S3 バケットを選択できます。record ディクショナリは、レコードのデータ、シーケンス番号、およびパーティションキーにアクセスする次のキーと値のペアを公開します。

```
record.get('data')
record.get('sequenceNumber')
record.get('partitionKey')
```

データは Base64 でエンコードされていることに注意してください。

サンプルでは、メソッド process_records に、ワーカーでレコードのデータ、シーケンス番号、およびパーティションキーにアクセスする方法を示すコードが含まれています。

Kinesis Data Streams では、シャードで既に処理されたレコードを追跡するためにレコードプロセッサが必要です。KCL は、Checkpointer オブジェクトを process_records に渡すことで、この

追跡をユーザーに代わって処理します。レコードプロセッサは、このオブジェクトの checkpoint メソッドを呼び出して、シャード内のレコードの処理の進行状況を KCL に通知します。ワーカーでエラーが発生すると、KCL はこの情報を使用して、処理されたことが分かっている最後のレコードからシャードの処理を再開します。

分割または結合オペレーションの場合、KCL は、元のシャードのプロセッサが checkpoint を呼び出して元のシャードの処理がすべて完了したことを通知するまで、新しいシャードの処理を開始しません。

パラメータを渡さないと、checkpoint への呼び出しは、レコードプロセッサに最後のレコードを渡した時点までのすべてのレコードが処理済みであることを意味すると KCL で見なされます。したがって、レコードプロセッサは、渡されたリストにあるすべてのレコードの処理が完了した場合にのみ、checkpoint を呼び出す必要があります。レコードプロセッサは、checkpoint の各呼び出しで process_records を呼び出す必要はありません。たとえば、プロセッサは、3 回呼び出すたびに、checkpoint を呼び出すことができます。オプションでレコードの正確なシーケンス番号をパラメータとして checkpoint に指定できます。この場合、KCL は、そのレコードまでのすべてのレコードだけが処理されたと見なします。

サンプルでは、プライベートメソッド checkpoint で、適切な例外処理と再試行のロジックを使用する Checkpointer.checkpoint メソッドを呼び出す方法を示しています。

KCL は、process_records を使用して、データレコードの処理から発生するすべての例外を処理します。例外が process_records からスローされた場合、 は、例外発生前に に渡されたデータレコードをスキップします。つまり、これらのレコードは、例外をスローしたレコードプロセッサ、またはコンシューマーの他のレコードプロセッサに再送信されません。

shutdown

KCL は、処理が終了した場合 (シャットダウンの理由は TERMINATE) またはワーカーが応答していない場合 (シャットダウンの reason は ZOMBIE)、shutdown メソッドを呼び出します。

```
def shutdown(self, checkpointer, reason)
```

シャードが分割または結合されたか、ストリームが削除されたため、レコードプロセッサがシャードからこれ以上レコードを受信しない場合は、処理が終了します。

また、KCL は、Checkpointer オブジェクトも shutdown に渡します。シャットダウンの reason が TERMINATE である場合、レコードプロセッサはすべてのデータレコードの処理を終了し、このインターフェイスの checkpoint メソッドを呼び出します。

設定プロパティを変更する

このサンプルでは、設定プロパティのデフォルト値を提供します。これらのプロパティを独自の値にオーバーライドできます (sample.properties を参照してください)。

アプリケーション名

KCL には、複数のアプリケーション間、および同じリージョン内の Amazon DynamoDB テーブル間で一意のアプリケーションが必要です。次のようにアプリケーション名の設定値を使用します。

- このアプリケーション名と関連付けられたワーカーはすべて、同じストリーム上で連携して処理しているとみなされます。これらのワーカーは複数のインスタンスに分散している場合があります。同じアプリケーションコードの追加のインスタンスを実行するときに、アプリケーション名が異なる場合、KCL は 2 番目のインスタンスを、同じストリームで動作するまったく別のアプリケーションと見なします。
- KCL はアプリケーション名を使用して DynamoDB テーブルを作成し、このテーブルを使用してアプリケーションの状態情報 (チェックポイントやワーカーとシャードのマッピングなど) を保存します。各アプリケーションには、それぞれ DynamoDB テーブルがあります。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

認証情報を設定する

デフォルトの AWS 認証情報プロバイダーチェーンの認証情報プロバイダーの 1 つが認証情報を利用できるようにする必要があります。AWSCredentialsProvider プロパティを使用して認証情報プロバイダーを設定できます。sample.properties では、デフォルトの認証情報プロバイダーチェーンのいずれかの認証情報プロバイダーに対して、ユーザーの認証情報を使用可能にする必要があります。Amazon EC2 インスタンスでコンシューマーアプリケーションを実行している場合は、IAM ロールでインスタンスを設定することをお勧めします。この IAM ロールに関連付けられた許可を反映する AWS 認証情報は、インスタンスメタデータを通じて、インスタンス上のアプリケーションで使用できるようになります。これは、EC2 インスタンスで実行されるコンシューマーアプリケーションの認証情報を管理するための最も安全な方法です。

サンプルのプロパティファイルでは、 で指定されているレコードプロセッサを使用してwordsというKinesis data stream を処理するように KCL を設定します。

Ruby での Kinesis Client Library コンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、Ruby について説明します。

KCL は Java ライブラリです。Java 以外の言語のサポートは、 と呼ばれる多言語インターフェイスを使用して提供されますMultiLangDaemon。このデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。したがって、Ruby 用の KCL をインストールし、コンシューマーアプリを Ruby に完全に記述しても、 のためにシステムに Java をインストールする必要があります MultiLangDaemon。さらに、接続先の AWS リージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定が MultiLangDaemon いくつかあります。 MultiLangDaemon の の詳細については GitHub、KCL MultiLangDaemon プロジェクトページを参照してください。

から Ruby KCL をダウンロードするには GitHub、Kinesis Client Library (Ruby) に移動します。Ruby KCL コンシューマーアプリケーションのサンプルコードをダウンロードするには、「」の「KCL for Ruby sample project」ページを参照してください GitHub。

KCL Ruby サポートライブラリの詳細については、KCL Ruby Gems ドキュメントを参照してください。

KCL 2.x コンシューマーの開発

このトピックでは、バージョン 2.0 の Kinesis Client Library (KCL) を使用する方法について説明します。KCL の詳細については、Kinesis Client Library 1.x を使用したコンシューマーの開発に示されている概要を参照してください。

目次

Java での Kinesis クライアントライブラリコンシューマーの開発

次のコードは、`ProcessorFactory` および `RecordProcessor` の Java のサンプル実装を示しています。拡張ファンアウト機能を活用する方法については、拡張ファンアウトでコンシューマーを使用するを参照してください。

```
/*
```

```
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */


/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
```

```
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.retrieval.polling.PollingConfig;

/**
 * This class will run a simple app that uses the KCL to read data and uses the AWS SDK
 to publish data.
 * Before running this program you must first create a Kinesis stream through the AWS
 console or AWS SDK.
 */
public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    /**
     * Invoke the main method with 2 args: the stream name and (optionally) the region.
     * Verifies valid inputs and then starts running the app.
     */
    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
 The Region may be specified as the second argument.");
```

```
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    /**
     * Constructor sets streamName and region. It also creates a KinesisClient object
to send data to Kinesis.
     * This KinesisClient is used to send dummy data so that the consumer has something
to read; it is also used
     * indirectly by the KCL to handle the consumption of the data.
     */
    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {

        /**
         * Sends dummy data to Kinesis. Not relevant to consuming the data with the KCL
         */
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        /**
         * Sets up configuration for the KCL, including DynamoDB and CloudWatch
dependencies. The final argument, a
```

```
         * ShardRecordProcessorFactory, is where the logic for record processing lives,
and is located in a private
         * class below.
         */
        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        /**
         * The Scheduler (also called Worker in earlier versions of the KCL) is the
entry point to the KCL. This
         * instance is configured with defaults provided by the ConfigsBuilder.
         */
        Scheduler scheduler = new Scheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                configsBuilder.retrievalConfig().retrievalSpecificConfig(new
PollingConfig(streamName, kinesisClient))
        );

        /**
         * Kickoff the Scheduler. Record processing of the stream of dummy data will
continue indefinitely
         * until an exit is triggered.
         */
        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        /**
         * Allows termination of app by pressing Enter.
         */
        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
```

```
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        /**
         * Stops sending dummy data.
         */
        log.info("Cancelling producer and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();

        /**
         * Stops consuming data. Finishes processing the current batch of data already
received from Kinesis
         * before shutting down.
         */
        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown.  Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    /**
     * Sends a single record of dummy data to Kinesis.
     */
    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
```

```
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }

    /**
     * The implementation of the ShardRecordProcessor interface is where the heart of
the record processing logic lives.
     * In this example all we do to 'process' is log info about the records.
     */
    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        /**
         * Invoked by the KCL before data records are delivered to the
ShardRecordProcessor instance (via
         * processRecords). In this example we do nothing except some logging.
         *
         * @param initializationInput Provides information related to initialization.
         */
        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
```

```
        }

        /**
         * Handles record processing logic. The Amazon Kinesis Client Library will
invoke this method to deliver
         * data records to the application. In this example we simply log our records.
         *
         * @param processRecordsInput Provides the records to be processed as well as
information and capabilities
         *                            related to them (e.g. checkpointing).
         */
        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)",
processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /** Called when the lease tied to this record processor has been lost. Once the
lease has been lost,
         * the record processor can no longer checkpoint.
         *
         * @param leaseLostInput Provides access to functions and data related to the
loss of the lease.
         */
        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
```

```
         * Called when all data on this shard has been processed. Checkpointing must
 occur in the method for record
         * processing to be considered complete; an exception will be thrown otherwise.
         *
         * @param shardEndedInput Provides access to a checkpointer method for
 completing processing of the shard.
         */
        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Invoked when Scheduler has been requested to shut down (i.e. we decide to
 stop running the app by pressing
         * Enter). Checkpoints and logs the data a final time.
         *
         * @param shutdownRequestedInput Provides access to a checkpointer, allowing a
 record processor to checkpoint
         *                               before the shutdown is completed.
         */
        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Python での Kinesis クライアントライブラリコンシューマーの開発

Kinesis Data Streams のデータを処理するアプリケーションを構築するには Kinesis Client Library (KCL) を使用できます。Kinesis Client Library は、複数の言語で使用できます。このトピックでは、Python について説明します。

KCL は Java ライブラリです。Java 以外の言語のサポートは、と呼ばれる多言語インターフェイスを使用して提供されます。MultiLangDaemonこのデーモンは Java ベースで、Java 以外の KCL 言語を使用しているときにバックグラウンドで実行されます。したがって、KCL for Python をインストールし、コンシューマーアプリケーションを完全に Python で作成する場合でも、の理由により、システムに Java をインストールする必要があります。 MultiLangDaemonさらに MultiLangDaemon 、AWS接続先のリージョンなど、ユースケースに合わせてカスタマイズする必要があるデフォルト設定がいくつかあります。詳細については GitHub、 MultiLangDaemon KCL MultiLangDaemon プロジェクトページを参照してください。

から Python KCL をダウンロードするには GitHub、Kinesis クライアントライブラリ (Python) にアクセスしてください。Python KCL コンシューマーアプリケーションのサンプルコードをダウンロードするには、の KCL for Python サンプルプロジェクトページにアクセスしてください。 GitHub

Python で KCL コンシューマーアプリケーションを実装する場合は、次のタスクを完了する必要があります。

タスク

- クラスメソッドを実装します。 RecordProcessor
- 設定プロパティを変更する

クラスメソッドを実装します。 RecordProcessor

`RecordProcess` クラスでは、`RecordProcessorBase` クラスを拡張して次のメソッドを実装する必要があります。

```
initialize
process_records
shutdown_requested
```

このサンプルでは、開始点として使用できる実装を提供しています。

```
#!/usr/bin/env python

# Copyright 2014-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Amazon Software License (the "License").
# You may not use this file except in compliance with the License.
# A copy of the License is located at
#
# http://aws.amazon.com/asl/
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.

from __future__ import print_function

import sys
import time

from amazon_kclpy import kcl
from amazon_kclpy.v3 import processor


class RecordProcessor(processor.RecordProcessorBase):
    """
    A RecordProcessor processes data from a shard in a stream. Its methods will be
 called with this pattern:

    * initialize will be called once
    * process_records will be called zero or more times
    * shutdown will be called if this MultiLangDaemon instance loses the lease to this
 shard, or the shard ends due
        a scaling change.
    """
    def __init__(self):
        self._SLEEP_SECONDS = 5
        self._CHECKPOINT_RETRIES = 5
        self._CHECKPOINT_FREQ_SECONDS = 60
        self._largest_seq = (None, None)
        self._largest_sub_seq = None
        self._last_checkpoint_time = None
```

```
    def log(self, message):
        sys.stderr.write(message)

    def initialize(self, initialize_input):
        """
        Called once by a KCLProcess before any calls to process_records

        :param amazon_kclpy.messages.InitializeInput initialize_input: Information
about the lease that this record
            processor has been assigned.
        """
        self._largest_seq = (None, None)
        self._last_checkpoint_time = time.time()

    def checkpoint(self, checkpointer, sequence_number=None, sub_sequence_number=None):
        """
        Checkpoints with retries on retryable exceptions.

        :param amazon_kclpy.kcl.Checkpointer checkpointer: the checkpointer provided to
either process_records
            or shutdown
        :param str or None sequence_number: the sequence number to checkpoint at.
        :param int or None sub_sequence_number: the sub sequence number to checkpoint
at.
        """
        for n in range(0, self._CHECKPOINT_RETRIES):
            try:
                checkpointer.checkpoint(sequence_number, sub_sequence_number)
                return
            except kcl.CheckpointError as e:
                if 'ShutdownException' == e.value:
                    #
                    # A ShutdownException indicates that this record processor should
be shutdown. This is due to
                    # some failover event, e.g. another MultiLangDaemon has taken the
lease for this shard.
                    #
                    print('Encountered shutdown exception, skipping checkpoint')
                    return
                elif 'ThrottlingException' == e.value:
                    #
                    # A ThrottlingException indicates that one of our dependencies is
is over burdened, e.g. too many
                    # dynamo writes. We will sleep temporarily to let it recover.
```

```
                    #
                    if self._CHECKPOINT_RETRIES - 1 == n:
                        sys.stderr.write('Failed to checkpoint after {n} attempts,
giving up.\n'.format(n=n))
                        return
                    else:
                        print('Was throttled while checkpointing, will attempt again in
{s} seconds'
                              .format(s=self._SLEEP_SECONDS))
                elif 'InvalidStateException' == e.value:
                    sys.stderr.write('MultiLangDaemon reported an invalid state while
checkpointing.\n')
                else:  # Some other error
                    sys.stderr.write('Encountered an error while checkpointing, error
was {e}.\n'.format(e=e))
            time.sleep(self._SLEEP_SECONDS)

    def process_record(self, data, partition_key, sequence_number,
sub_sequence_number):
        """
        Called for each record that is passed to process_records.

        :param str data: The blob of data that was contained in the record.
        :param str partition_key: The key associated with this recod.
        :param int sequence_number: The sequence number associated with this record.
        :param int sub_sequence_number: the sub sequence number associated with this
record.
        """
        ####################################
        # Insert your processing logic here
        ####################################
        self.log("Record (Partition Key: {pk}, Sequence Number: {seq}, Subsequence
Number: {sseq}, Data Size: {ds}"
                 .format(pk=partition_key, seq=sequence_number,
sseq=sub_sequence_number, ds=len(data)))

    def should_update_sequence(self, sequence_number, sub_sequence_number):
        """
        Determines whether a new larger sequence number is available

        :param int sequence_number: the sequence number from the current record
        :param int sub_sequence_number: the sub sequence number from the current record
        :return boolean: true if the largest sequence should be updated, false
otherwise
```

```
        """
        return self._largest_seq == (None, None) or sequence_number >
self._largest_seq[0] or \
            (sequence_number == self._largest_seq[0] and sub_sequence_number >
self._largest_seq[1])

    def process_records(self, process_records_input):
        """
        Called by a KCLProcess with a list of records to be processed and a
checkpointer which accepts sequence numbers
        from the records to indicate where in the stream to checkpoint.

        :param amazon_kclpy.messages.ProcessRecordsInput process_records_input: the
records, and metadata about the
            records.
        """
        try:
            for record in process_records_input.records:
                data = record.binary_data
                seq = int(record.sequence_number)
                sub_seq = record.sub_sequence_number
                key = record.partition_key
                self.process_record(data, key, seq, sub_seq)
                if self.should_update_sequence(seq, sub_seq):
                    self._largest_seq = (seq, sub_seq)

            #
            # Checkpoints every self._CHECKPOINT_FREQ_SECONDS seconds
            #
            if time.time() - self._last_checkpoint_time >
self._CHECKPOINT_FREQ_SECONDS:
                self.checkpoint(process_records_input.checkpointer,
str(self._largest_seq[0]), self._largest_seq[1])
                self._last_checkpoint_time = time.time()

        except Exception as e:
            self.log("Encountered an exception while processing records. Exception was
{e}\n".format(e=e))

    def lease_lost(self, lease_lost_input):
        self.log("Lease has been lost")

    def shard_ended(self, shard_ended_input):
        self.log("Shard has ended checkpointing")
```

```
        shard_ended_input.checkpointer.checkpoint()

    def shutdown_requested(self, shutdown_requested_input):
        self.log("Shutdown has been requested, checkpointing.")
        shutdown_requested_input.checkpointer.checkpoint()


if __name__ == "__main__":
    kcl_process = kcl.KCLProcess(RecordProcessor())
    kcl_process.run()
```

設定プロパティを変更する

このサンプルでは、次のスクリプトに示すように、設定プロパティのデフォルト値を提供します。これらのプロパティを独自の値にオーバーライドできます。

```
# The script that abides by the multi-language protocol. This script will
# be executed by the MultiLangDaemon, which will communicate with this script
# over STDIN and STDOUT according to the multi-language protocol.
executableName = sample_kclpy_app.py

# The name of an Amazon Kinesis stream to process.
streamName = words

# Used by the KCL as the name of this application. Will be used as the name
# of an Amazon DynamoDB table which will store the lease and checkpoint
# information for workers with this application name
applicationName = PythonKCLSample

# Users can change the credentials provider the KCL will use to retrieve credentials.
# The DefaultAWSCredentialsProviderChain checks several other providers, which is
# described here:
# http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/auth/
DefaultAWSCredentialsProviderChain.html
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain

# Appended to the user agent of the KCL. Does not impact the functionality of the
# KCL in any other way.
processingLanguage = python/2.7

# Valid options at TRIM_HORIZON or LATEST.
# See http://docs.aws.amazon.com/kinesis/latest/APIReference/
API_GetShardIterator.html#API_GetShardIterator_RequestSyntax
```

```
initialPositionInStream = TRIM_HORIZON

# The following properties are also available for configuring the KCL Worker that is
 created
# by the MultiLangDaemon.

# The KCL defaults to us-east-1
#regionName = us-east-1

# Fail over time in milliseconds. A worker which does not renew it's lease within this
 time interval
# will be regarded as having problems and it's shards will be assigned to other
 workers.
# For applications that have a large number of shards, this msy be set to a higher
 number to reduce
# the number of DynamoDB IOPS required for tracking leases
#failoverTimeMillis = 10000

# A worker id that uniquely identifies this worker among all workers using the same
 applicationName
# If this isn't provided a MultiLangDaemon instance will assign a unique workerId to
 itself.
#workerId =

# Shard sync interval in milliseconds - e.g. wait for this long between shard sync
 tasks.
#shardSyncIntervalMillis = 60000

# Max records to fetch from Kinesis in a single GetRecords call.
#maxRecords = 10000

# Idle time between record reads in milliseconds.
#idleTimeBetweenReadsInMillis = 1000

# Enables applications flush/checkpoint (if they have some data "in progress", but
 don't get new data for while)
#callProcessRecordsEvenForEmptyRecordList = false

# Interval in milliseconds between polling to check for parent shard completion.
# Polling frequently will take up more DynamoDB IOPS (when there are leases for shards
 waiting on
# completion of parent shards).
#parentShardPollIntervalMillis = 10000
```

```
# Cleanup leases upon shards completion (don't wait until they expire in Kinesis).
# Keeping leases takes some tracking/resources (e.g. they need to be renewed,
 assigned), so by default we try
# to delete the ones we don't need any longer.
#cleanupLeasesUponShardCompletion = true

# Backoff time in milliseconds for Amazon Kinesis Client Library tasks (in the event of
 failures).
#taskBackoffTimeMillis = 500

# Buffer metrics for at most this long before publishing to CloudWatch.
#metricsBufferTimeMillis = 10000

# Buffer at most this many metrics before publishing to CloudWatch.
#metricsMaxQueueSize = 10000

# KCL will validate client provided sequence numbers with a call to Amazon Kinesis
 before checkpointing for calls
# to RecordProcessorCheckpointer#checkpoint(String) by default.
#validateSequenceNumberBeforeCheckpointing = true

# The maximum number of active threads for the MultiLangDaemon to permit.
# If a value is provided then a FixedThreadPool is used with the maximum
# active threads set to the provided value. If a non-positive integer or no
# value is provided a CachedThreadPool is used.
#maxActiveThreads = 0
```

Application Name(アプリケーション名)

KCL には、複数のアプリケーション間、および同じリージョン内の Amazon DynamoDB テーブル間で一意のアプリケーションが必要です。次のようにアプリケーション名の設定値を使用します。

- このアプリケーション名と関連付けられたワーカーはすべて、同じストリーム上で連携して処理しているとみなされます。これらのワーカーは複数のインスタンス間に分散している場合があります。同じアプリケーションコードの追加のインスタンスを実行するときに、アプリケーション名が異なる場合、KCL は 2 番目のインスタンスを、同じストリームで動作するまったく別のアプリケーションと見なします。
- KCL はアプリケーション名を使用して DynamoDB テーブルを作成し、このテーブルを使用してアプリケーションの状態情報 (チェックポイントやワーカーとシャードのマッピングなど) を保存します。各アプリケーションには、それぞれ DynamoDB テーブルがあります。詳細については、

「リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡する」を参照してください。

認証情報

デフォルトの認証情報プロバイダチェーンのいずれかの認証情報プロバイダで AWS の認証情報を使用できるようにする必要があります。`AWSCredentialsProvider` プロパティを使用して認証情報プロバイダーを設定できます。Amazon EC2 インスタンスでコンシューマーアプリケーションを実行している場合は、IAM ロールでインスタンスを設定することをお勧めします。この IAM ロールに関連付けられた許可を反映する AWS 認証情報は、インスタンスメタデータを通じて、インスタンス上のアプリケーションで使用できるようになります。これは、EC2 インスタンスで実行されるコンシューマーアプリケーションの認証情報を管理するための最も安全な方法です。

AWS SDK for Java を使用した共有スループットでのカスタムコンシューマーの開発

全体で共有されたるカスタム Kinesis Data Streams コンシューマーを開発する方法の 1 つは、Amazon Kinesis Data Streams API を使用することです。このセクションでは、AWS SDK for Java での Kinesis Data Streams API の使用について説明します。このセクションで紹介する Java サンプルコードは、基本的な KDS API オペレーションを実行する方法を示しており、オペレーションタイプ別に論理的に分割されています。

これらのサンプルコードは、本稼働環境対応のコードではありません。考えられる例外のすべてを確認するものではなく、潜在的なセキュリティまたはパフォーマンス事項も考慮されていません。

また、他のプログラミング言語を使用して Kinesis Data Streams API を呼び出すこともできます。すべての利用可能な AWS SDK の詳細については、Amazon Web Services を使用した開発の開始を参照してください。

> **Important**
>
> 全体で共有されるカスタム Kinesis Data Streams コンシューマーを開発するには、Kinesis Client Library (KCL) を使用することをお勧めします。KCL は、分散コンピューティングに関連する複雑なタスクの多くを処理することで、Kinesis Data Streams からデータを消費および処理するのに役立ちます。詳細については、KCL を使用したスループット共有カスタムコンシューマーの開発を参照してください。

トピック

ストリームからのデータの取得

Kinesis Data Streams API には、データストリームからレコードを取得するために呼び出すことができる `getShardIterator` および `getRecords` メソッドが含まれています。これはプルモデルで、コードはデータストリームのシャードからデータを直接取得します。

> **Important**
>
> KCL によって提供されているレコードプロセッサのサポートを使用して、データストリームからレコードを取得することをお勧めします。これは、データを処理するコードを組み込むプッシュモデルです。KCL は、ストリームからデータレコードを取り出し、アプリケーションコードに配信します。さらに、CL には、フェイルオーバー、リカバリ、負荷分散の機能が用意されています。詳細については、KCL を使用したスループット共有カスタムコンシューマーの開発を参照してください。

ただし、状況によっては Kinesis Data Streams API を使用した方がよい場合があります。例えば、データストリームのモニタリングやデバッグのためのカスタムツールを実装する場合です。

> **Important**
>
> Kinesis Data Streams は、データストリームのデータレコードの保持期間の変更をサポートしています。詳細については、データ保持期間の変更を参照してください。

シャードイテレーターの使用

レコードは、シャード単位でストリームから取得します。シャードごと、およびそのシャードから取得するレコードのバッチごとに、シャードイテレーターを取得する必要があります。シャード

イテレーターは、レコードの取得元になるシャードを指定するために getRecordsRequest オブジェクトで使用します。シャードイテレーターに関連付けられるタイプによって、レコードを取得するシャード内の位置が決まります (詳細については、このセクションの後半を参照してください)。DescribeStream API - 非推奨で説明したように、シャードイテレーターを使用する前にシャードを取得する必要が あります。

最初のシャードイテレーターは、getShardIterator メソッドを使用して取得します。追加のレコードバッチのシャードイテレーターは、getNextShardIterator メソッドによって返された getRecordsResult オブジェクトの getRecords メソッドを使用して取得します。シャードイテレーターの有効時間は 5 分間です。有効時間内にシャードイテレーターを使用すると、新しいシャードイテレーターが取得されます。各シャードイテレーターは、使用後も 5 分間有効です。

最初のシャードイテレーターを取得するには、GetShardIteratorRequest をインスタンス化してから、getShardIterator メソッドに渡します。リクエストを設定するには、ストリームとシャード ID を指定します。AWS アカウントのストリームを取得する方法については、ストリームのリストを参照してください。ストリーム内のシャードを取得する方法については、DescribeStream API - 非推奨を参照してください。

```
String shardIterator;
GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest();
getShardIteratorRequest.setStreamName(myStreamName);
getShardIteratorRequest.setShardId(shard.getShardId());
getShardIteratorRequest.setShardIteratorType("TRIM_HORIZON");

GetShardIteratorResult getShardIteratorResult =
 client.getShardIterator(getShardIteratorRequest);
shardIterator = getShardIteratorResult.getShardIterator();
```

このサンプルコードは、最初のシャードイテレーターを取得するときのイテレータータイプとして TRIM_HORIZON を指定しています。このイテレーター型を指定することで、レコードはまず、シャードに直近に追加されたレコード (tip) からではなく、シャードに最初に追加されたレコードから返されます。以下は、使用可能なイテレータータイプです。

- AT_SEQUENCE_NUMBER
- AFTER_SEQUENCE_NUMBER
- AT_TIMESTAMP
- TRIM_HORIZON
- LATEST

詳細については、ShardIteratorTypeを参照してください。

イテレータータイプによっては、タイプに加えてシーケンス番号を指定する必要があります。以下がその例です。

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

getRecords を使用してレコードを取得したら、レコードの getSequenceNumber メソッドを呼び出すことによって、レコードのシーケンス番号を取得できます。

```
record.getSequenceNumber()
```

さらに、データストリームにレコードを追加するコードは、getSequenceNumber の結果に対して putRecord を呼び出すことで、追加されたレコードのシーケンス番号を取得できます。

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

シーケンス番号は、レコードの順序が厳密に増加することを保証するために使用できます。詳細については、PutRecord の例のサンプルコードを参照してください。

GetRecords の使用

シャードイテレーターを取得したら、GetRecordsRequest オブジェクトをインスタンス化します。リクエストのイテレーターは、setShardIterator メソッドを使用して指定します。

オプションとして、setLimit メソッドを使用することで、取得するレコードの数を設定することもできます。getRecords が返すレコードの数は、常にこの制限以下になります。この制限を指定しない場合、getRecords は取得したレコードの 10 MB を返します。次のサンプルコードは、この制限を 25 レコードに設定します。

レコードが返されない場合、シャードイテレーターが参照するシーケンス番号には、このシャードから現在利用できるデータレコードが存在しないことを意味します。この状況では、アプリケーションが、ストリームのデータソースに対して適切な時間を待機する必要があります。その後、getRecords への以前のコールで返されたシャードイテレーターを使用して、シャードからのデータの取得を再試行します。

getRecordsRequest メソッドに getRecords を渡し、返された値を getRecordsResult オブジェクトとしてキャプチャします。データレコードを取得するには、getRecords オブジェクトに対して getRecordsResult メソッドを呼び出します。

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(25);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

getRecords への別のコールを準備するために、getRecordsResult から次のシャードイテレーターを取得します。

```
shardIterator = getRecordsResult.getNextShardIterator();
```

最良の結果を得るには、getRecords へのコール間で少なくとも 1 秒間 (1,000 ミリ秒) スリープして、getRecords の頻度制限を超えないようにしてください。

```
try {
  Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

一般的に、テストシナリオで単一のレコードを取得している場合でも、getRecords はループで呼び出す必要があります。getRecords への単一のコールは、後続のシーケンス番号ではシャード内に複数のレコードがあるという場合でも、空のレコードリストを返す可能性があります。この状況が発生すると、空のレコードリストとともに返された NextShardIterator が、シャード内の後続のシーケンス番号を参照し、最終的には連続する getRecords コールがレコードを返します。次のサンプルは、ループの使用を表すものです。

例: getRecords

以下のコード例には、このセクションで説明した getRecords のヒント (ループでの呼び出しなど) が反映されています。

```
// Continuously read data records from a shard
List<Record> records;
```

```
while (true) {

  // Create a new getRecordsRequest with an existing shardIterator
  // Set the maximum records to return to 25

  GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
  getRecordsRequest.setShardIterator(shardIterator);
  getRecordsRequest.setLimit(25);

  GetRecordsResult result = client.getRecords(getRecordsRequest);

  // Put the result into record list. The result can be empty.
  records = result.getRecords();

  try {
    Thread.sleep(1000);
  }
  catch (InterruptedException exception) {
    throw new RuntimeException(exception);
  }

  shardIterator = result.getNextShardIterator();
}
```

Kinesis Client Library を使用している場合は、データを返す前に複数回呼び出しが行われる場合があります。この動作は仕様であり、KCL やデータの問題を示すものではありません。

リシャーディングへの適応

`getRecordsResult.getNextShardIterator` が `null` を返す場合、このシャードに関係するシャード分割またはマージが発生したことを示します。このシャードは現在 CLOSED 状態であり、このシャードから使用可能なすべてのデータレコードを読み込んでいます。

このシナリオでは、`getRecordsResult.childShards` を使用して、分割またはマージによって作成された、処理中のシャードの新しい子シャードについて学習することができます。詳細については、ChildShardを参照してください。

分割の場合は、2 つの新しいシャードの両方に、以前処理していたシャードのシャード ID に等しい `parentShardId` があります。`adjacentParentShardId` の値は、これらのシャード両方で `null` になります。

マージの場合は、マージによって作成された単一の新しいシャードに、一方の親シャードの ID に等しい parentShardId と、もう一方の親シャードのシャード ID に等しい adjacentParentShardId があります。アプリケーションはこれらのいずれかのシャードからすべてのデータを読み取り済みです。これは getRecordsResult.getNextShardIterator から null が返されたシャードです。アプリケーションでデータの順序が重要である場合、結合によって作成された子シャードから新しいデータを読み取る前に、その他の親シャードからもすべてのデータを読み取るようにする必要があります。

複数のプロセッサを使用してストリームからデータを取得し (たとえば、シャードごとに 1 つのプロセッサ)、シャードの分割または結合を行う場合、プロセッサの数を増減して、シャードの数の変化に適応させます。

シャードの状態 (CLOSED など) の説明を含むリシャーディングの詳細については、ストリームをリシャーディングするを参照してください。

AWS Glue スキーマレジストリを使用してデータと相互作用する

Kinesis Data Streams を、AWS Glue スキーマレジストリと統合することができます。AWS Glue スキーマレジストリを使用すると、スキーマを一元的に検出、制御、および進化させながら、生成されたデータが登録されたスキーマによって継続的に検証されるようにできます。スキーマは、データレコードの構造と形式を定義します。スキーマは、信頼性の高いデータの公開、利用、または保存のための仕様をバージョニングしたものです。AWS Glue スキーマレジストリを使用すると、ストリーミングアプリケーション内のエンドツーエンドのデータ品質とデータガバナンスを改善できます。詳細については、AWS Glue スキーマレジストリを参照してください。この統合を設定する方法の 1 つは、AWS Java SDK で利用可能な GetRecords Kinesis Data Streams API を使用することです。

GetRecords Kinesis Data Streams API を使用して Kinesis Data Streams とスキーマレジストリの統合を設定する方法の詳細については、ユースケース: Amazon Kinesis Data Streams と AWS Glue スキーマレジストリの統合のKinesis Data Streams API を使用したデータの操作セクションを参照してください。

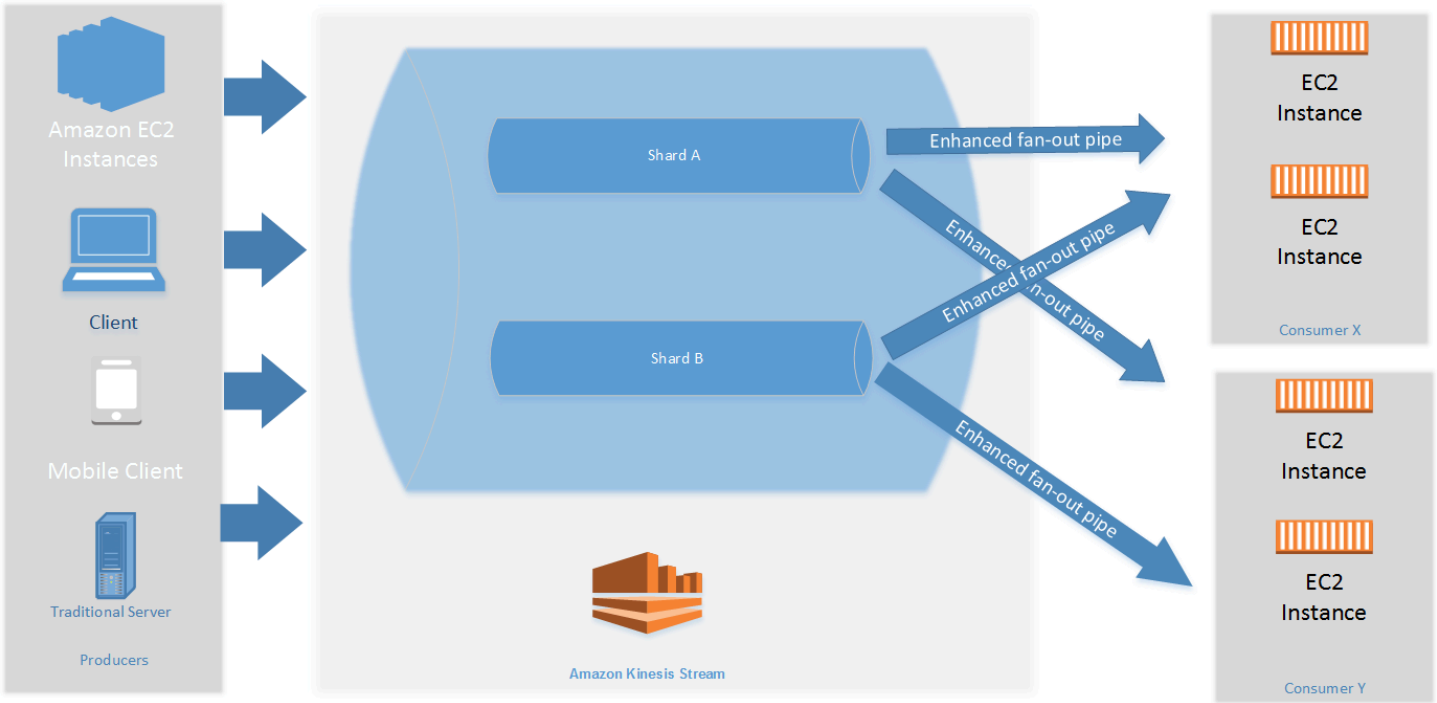
スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発

Amazon Kinesis Data Streams では、拡張ファンアウトと呼ばれる機能を使用するコンシューマーを構築できます。この機能により、コンシューマーは、シャードあたり 1 秒間に最大 2 MB のデータのスループットで、ストリームからレコードを受け取ることができます。このスループットは専用です。つまり、拡張ファンアウトを使用するコンシューマーは、ストリームからデータを受け取る他の

コンシューマーと競合する必要がありません。Kinesis Data Streams は、ストリームのデータレコードを、拡張ファンアウトを使用するコンシューマーに送信します。そのため、これらのコンシューマーはデータをポーリングする必要はありません。

> **Important**
>
> ストリームあたり最大 20 のコンシューマーを登録して、拡張ファンアウトを使用できます。

拡張ファンアウトのアーキテクチャを以下の図に示します。バージョン 2.0 以降の Amazon Kinesis Client Library (KCL) を使用してコンシューマーを構築する場合、KCL は拡張ファンアウトを使用してストリームのすべてのシャードからデータを受け取るように、コンシューマーを設定します。API を使用して、拡張ファンアウトを使用するコンシューマーを構築する場合は、シャードを個別にサブスクライブできます。



図に示す内容は以下のとおりです。

- 2 つのシャードを持つストリーム。
- ストリームからデータを受信するために拡張ファンアウトを使用する 2 つのコンシューマー (コンシューマー X とコンシューマー Y)。2 つのコンシューマーはそれぞれ、ストリームのすべてのシャードとすべてのレコードにサブスクライブされています。バージョン 2.0 以降の KCL を使

用してコンシューマーを構築する場合、KCL は自動的に、ストリームのすべてのシャードにコンシューマーをサブスクライブします。これに対し、API を使用してコンシューマーを構築する場合は、シャードを個別にサブスクライブできます。

- コンシューマーがストリームからデータを受け取るために使用する拡張ファンアウトパイプを表す矢印。拡張されたファンアウトパイプは、シャードあたり最大 2 MB/秒 のデータを送信します。他のパイプやコンシューマーの総数は関係ありません。

トピック

- KCL 2.x を使用して拡張ファンアウトコンシューマーを開発する
- Kinesis Data Streams API を使用して拡張ファンアウトコンシューマーを開発する
- AWS Management Console を使用して拡張ファンアウトコンシューマーを管理する

KCL 2.x を使用して拡張ファンアウトコンシューマーを開発する

Amazon Kinesis Data Streams で拡張ファンアウトを使用するコンシューマーは、シャードあたり 1 秒間に最大 2 MB のデータの専用スループットで、データストリームからレコードを受け取ることができます。このタイプのコンシューマーは、ストリームからデータを受け取っている他のコンシューマーと競合する必要はありません。詳細については、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

拡張ファンアウトを使用してストリームからデータを受け取るアプリケーションを開発するには、バージョン 2.0 以降の Kinesis Client Library (KCL) を使用できます。KCL は、アプリケーションをストリームのすべてのシャードに自動的にサブスクライブし、コンシューマーアプリケーションがシャードあたり 2 MB/秒のスループット値で読み取ることができるようにします。拡張ファンアウトをオンにせずに KCL を使用する場合は、Kinesis Client Library 2.0 を使用したコンシューマーの開発を参照してください。

トピック

- Java で KCL 2.x を使用して拡張ファンアウトコンシューマーを開発する

Java で KCL 2.x を使用して拡張ファンアウトコンシューマーを開発する

拡張ファンアウトを使用してストリームからデータを受け取るアプリケーションを Amazon Kinesis Data Streams で開発するには、バージョン 2.0 以降の Kinesis Client Library (KCL) を使用できます。次のコードは、`ProcessorFactory` および `RecordProcessor` の Java のサンプル実装を示しています。

KinesisClientUtil を使用して KinesisAsyncClient を作成し、KinesisAsyncClient で maxConcurrency を設定することをお勧めします。

> **Important**
>
> すべてのリースと KinesisAsyncClient の追加使用のための十分な高い maxConcurrency を持つよう KinesisAsyncClient を設定しないと、Amazon Kinesis Client で非常に大きなレイテンシーが発生する可能性があります。

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */

/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
 The Region may be specified as the second argument.");
```

```
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        Scheduler scheduler = new Scheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
```

```
                configsBuilder.retrievalConfig()
        );

        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        log.info("Cancelling producer, and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();

        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown. Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
```

```
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }


    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)",
processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
```

```
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Kinesis Data Streams API を使用して拡張ファンアウトコンシューマーを開発する

拡張ファンアウトは Amazon Kinesis Data Streams の機能です。この機能を使用すると、コンシューマーは、シャードあたり 1 秒間に最大 2 MB のデータの専用スループットで、データストリームからレコードを受け取ることができます。拡張ファンアウトを使用するコンシューマーは、ストリームからデータを受け取っている他のコンシューマーと競合する必要はありません。詳細については、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

拡張ファンアウトを inesis Data Streams で使用するコンシューマーを構築するには、API オペレーションを使用します。

Kinesis Data Streams API を使用して拡張ファンアウトでコンシューマーを登録するには

1. 拡張ファンアウトを使用するコンシューマーとしてアプリケーションを登録するには、RegisterStreamConsumer を呼び出します。Kinesis Data Streams は、コンシューマーの Amazon リソースネーム (ARN) を生成し、レスポンスで返します。
2. 特定のシャードのリッスンを開始するには、SubscribeToShard への呼び出しでコンシューマー ARN を渡します。シャードのレコードは Kinesis Data Streams によって、SubscribeToShardEvent イベントの形式で HTTP/2 接続経由で送信されます。接続は最大 5 分間開いたままです。SubscribeToShard への呼び出しによって返される `future` が正常または例外的に完了した後も、引き続きシャードからレコードを受け取る場合は、SubscribeToShard を再度呼び出します。

   > **Note**
   >
   > `SubscribeToShard` API は、現在のシャードの終わりに達すると、現在のシャードの子シャードのリストも返します。

3. 拡張ファンアウトを使用しているコンシューマーの登録を解除するには、DeregisterStreamConsumer を呼び出します。

次のコードは、シャードへのコンシューマーのサブスクライブ、サブスクリプションの定期更新、イベントの処理を行う方法の例です。

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
```

```
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
    import
 software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;

    import java.util.concurrent.CompletableFuture;

    /**
     * See https://github.com/awsdocs/aws-doc-sdk-examples/blob/master/javav2/
example_code/kinesis/src/main/java/com/example/kinesis/KinesisStreamEx.java
     * for complete code and more examples.
     */
    public class SubscribeToShardSimpleImpl {

        private static final String CONSUMER_ARN = "arn:aws:kinesis:us-
east-1:123456789123:stream/foobar/consumer/test-consumer:1525898737";
        private static final String SHARD_ID = "shardId-000000000000";

        public static void main(String[] args) {

            KinesisAsyncClient client = KinesisAsyncClient.create();

            SubscribeToShardRequest request = SubscribeToShardRequest.builder()
                    .consumerARN(CONSUMER_ARN)
                    .shardId(SHARD_ID)
                    .startingPosition(s -> s.type(ShardIteratorType.LATEST)).build();

            // Call SubscribeToShard iteratively to renew the subscription
 periodically.
            while(true) {
                // Wait for the CompletableFuture to complete normally or
 exceptionally.
                callSubscribeToShardWithVisitor(client, request).join();
            }

            // Close the connection before exiting.
            // client.close();
        }


        /**
         * Subscribes to the stream of events by implementing the
 SubscribeToShardResponseHandler.Visitor interface.
         */
```

```
        private static CompletableFuture<Void>
callSubscribeToShardWithVisitor(KinesisAsyncClient client, SubscribeToShardRequest
request) {
            SubscribeToShardResponseHandler.Visitor visitor = new
SubscribeToShardResponseHandler.Visitor() {
                @Override
                public void visit(SubscribeToShardEvent event) {
                    System.out.println("Received subscribe to shard event " + event);
                }
            };
            SubscribeToShardResponseHandler responseHandler =
SubscribeToShardResponseHandler
                    .builder()
                    .onError(t -> System.err.println("Error during stream - " +
t.getMessage()))
                    .subscriber(visitor)
                    .build();
            return client.subscribeToShard(request, responseHandler);
        }
    }
```

event.ContinuationSequenceNumber が null を返す場合、このシャードに関係するシャード分割またはマージが発生したことを示します。このシャードは現在 CLOSED 状態であり、このシャードから使用可能なすべてのデータレコードを読み込んでいます。このシナリオでは、上記の例のように、event.childShards を使用して、分割またはマージによって作成された、処理中のシャードの新しい子シャードについて学習することができます。詳細については、ChildShardを参照してください。

AWS Glue スキーマレジストリを使用してデータと相互作用する

Kinesis Data Streams を、AWS Glue スキーマレジストリと統合することができます。AWS Glue スキーマレジストリを使用すると、スキーマを一元的に検出、制御、および進化させながら、生成されたデータが登録されたスキーマによって継続的に検証されるようにできます。スキーマは、データレコードの構造と形式を定義します。スキーマは、信頼性の高いデータの公開、利用、または保存のための仕様をバージョニングしたものです。AWS Glue スキーマレジストリを使用すると、ストリーミングアプリケーション内のエンドツーエンドのデータ品質とデータガバナンスを改善できます。詳細については、AWS Glue スキーマレジストリを参照してください。この統合を設定する方法の 1 つは、AWS Java SDK で利用可能な GetRecords Kinesis Data Streams API を使用することです。

GetRecords Kinesis Data Streams API を使用して Kinesis Data Streams とスキーマレジストリの統合を設定する方法の詳細については、ユースケース: Amazon Kinesis Data Streams と AWS Glue

スキーマレジストリの統合のKinesis Data Streams API を使用したデータの操作セクションを参照してください。

AWS Management Console を使用して拡張ファンアウトコンシューマーを管理する

Amazon Kinesis Data Streams で拡張ファンアウトを使用するコンシューマーは、シャードあたり 1 秒間に最大 2 MB のデータの専用スループットで、データストリームからレコードを受け取ることができます。詳細については、スループット専有 (拡張ファンアウト) カスタムコンシューマーの開発を参照してください。

特定のストリームで拡張ファンアウトを使用するように登録されているすべてのコンシューマーのリストを表示するには、AWS Management Console を使用します。このようなコンシューマーのそれぞれで、ARN、ステータス、作成日、およびモニタリングメトリクスなどの詳細情報を確認することができます。

拡張ファンアウトやそのステータス、作成日、メトリクスをコンソールで使用するように登録されているコンシューマーを表示するには

1. AWS Management Consoleにサインインして、Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. ナビゲーションペインで、[データストリーム] を選択します。
3. Kinesis Data Streams を選択して、詳細を表示します。
4. ストリームの詳細ページで、[拡張ファンアウト] タブを選択します。
5. コンシューマーを選択して、名前、ステータス、登録日を表示します。

コンシューマーの登録を解除するには

1. Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. ナビゲーションペインで、[データストリーム] を選択します。
3. Kinesis Data Streams を選択して、詳細を表示します。
4. ストリームの詳細ページで、[拡張ファンアウト] タブを選択します。
5. 登録解除する各コンシューマーの名前の左にあるチェックボックスをオンにします。
6. [コンシューマーの登録解除] を選択します。

コンシューマーを KCL 1.x から KCL 2.x に移行する

このトピックでは、Kinesis Client Library (KCL) のバージョン 1.x と 2.x の違いについて説明します。また、コンシューマーを KCL のバージョン 1.x からバージョン 2.x に移行する方法も示します。クライアントを移行すると、最後にチェックポイントが作成された場所からレコードの処理が開始されます。

KCL のバージョン 2.0 では、以下のインターフェイスの変更が導入されています。

KCL インターフェイスの変更

KCL 1.x インターフェイス	KCL 2.0 インターフェイス
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor`	`software.amazon.kinesis.processor.ShardRecordProcessor`
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory`	`software.amazon.kinesis.processor.ShardRecordProcessorFactory`
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware`	`software.amazon.kinesis.processor.ShardRecordProcessor` 内に折りたたみ

トピック

- レコードプロセッサの移行
- レコードプロセッサファクトリーの移行
- ワーカーの移行
- Amazon Kinesis Client の設定
- アイドル時間の削除
- クライアント設定の削除

レコードプロセッサの移行

以下の例は、KCL1.x に実装されたレコードプロセッサを示しています。

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;

public class TestRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        //
        // Process records, and possibly checkpoint
        //
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
                shutdownInput.getCheckpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
```

```
        try {
            checkpointer.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
}
```

レコードプロセッサのクラスを移行するには

1. インターフェイスを com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor および com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificat から software.amazon.kinesis.processor.ShardRecordProcessor に変更します。以下に例を示します。

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// public class TestRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
public class TestRecordProcessor implements ShardRecordProcessor {
```

2. import メソッド initialize とメソッドの processRecords ステートメントを更新します。

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;

//import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
```

3. shutdown メソッドを以下の新しいメソッドに置き換えます。leaseLost、shardEnded、および shutdownRequested。

```
//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shardEnded(...)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shutdownRequested(ShutdownReauestedInput)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
```

```
//            e.printStackTrace();
//        }
//    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
```

以下に示しているのは、レコードプロセッサのクラスの更新されたバージョンです。

```
package com.amazonaws.kcl;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public class TestRecordProcessor implements ShardRecordProcessor {
    @Override
    public void initialize(InitializationInput initializationInput) {

    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {

    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {
```

```
    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

レコードプロセッサファクトリーの移行

レコードプロセッサファクトリーは、リースが取得された際にレコードプロセッサの作成を担当します。以下に示しているのは、KCL 1.x ファクトリーの例です。

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class TestRecordProcessorFactory implements IRecordProcessorFactory {
    @Override
    public IRecordProcessor createProcessor() {
        return new TestRecordProcessor();
```

```
    }
}
```

レコードプロセッサファクトリーを移行するには

1. 実装されているインターフェイスを com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFa から software.amazon.kinesis.processor.ShardRecordProcessorFactory に変更します。以下に例を示します。

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

// public class TestRecordProcessorFactory implements IRecordProcessorFactory {
public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
```

2. createProcessor の戻り署名を変更します。

```
// public IRecordProcessor createProcessor() {
public ShardRecordProcessor shardRecordProcessor() {
```

以下は、2.0 のレコードプロセッサファクトリーの例です。

```
package com.amazonaws.kcl;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new TestRecordProcessor();
    }
}
```

ワーカーの移行

バージョン 2.0 の KCL では、新しいクラス Scheduler によって Worker クラスが置き換えられます。KCL 1.x のワーカーの例を次に示します。

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

ワーカーを移行するには

1. Worker クラスの import ステートメントを Scheduler クラスと ConfigsBuilder クラスのインポートステートメントに変更します。

   ```
   // import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
   import software.amazon.kinesis.coordinator.Scheduler;
   import software.amazon.kinesis.common.ConfigsBuilder;
   ```

2. 次の例に示すように、ConfigsBuilder と Scheduler を作成します。

   KinesisClientUtil を使用して KinesisAsyncClient を作成し、KinesisAsyncClient で maxConcurrency を設定することをお勧めします。

   > **Important**
   >
   > すべてのリースと KinesisAsyncClient の追加使用のための十分な高い maxConcurrency を持つよう KinesisAsyncClient を設定しないと、Amazon Kinesis Client で非常に大きなレイテンシーが発生する可能性があります。

   ```
   import java.util.UUID;

   import software.amazon.awssdk.regions.Region;
   import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
   import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
   import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
   ```

```
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;

...

Region region = Region.AP_NORTHEAST_2;
KinesisAsyncClient kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(region));
DynamoDbAsyncClient dynamoClient =
 DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();

ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, applicationName,
 kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
 SampleRecordProcessorFactory());

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
    );
```

Amazon Kinesis Client の設定

Kinesis Client Library のリリース 2.0 では、クライアントの設定が単一の設定クラス (KinesisClientLibConfiguration) から 6 つの設定クラスに移行されました。次の表で移行を説明します。

設定フィールドとその新しいクラス

元のフィールド	新しい設定クラス	説明
applicationName	ConfigsBuilder	この KCL アプリケーションの名前。tableName および consumerName のデフォルトとして使用されます。
tableName	ConfigsBuilder	Amazon DynamoDB リーステーブルで使用されるテーブル名の上書きを許可します。
streamName	ConfigsBuilder	このアプリケーションがレコードを処理するストリームの名前。
kinesisEndpoint	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
dynamoDBEndpoint	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
initialPositionInStreamExtended	RetrievalConfig	アプリケーションの初期実行から開始し、KCL がレコードの取得を開始するシャード内の場所。
kinesisCredentialsProvider	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
dynamoDBCredentialsProvider	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
cloudWatchCredentialsProvider	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
failoverTimeMillis	LeaseManagementConfig	リース所有者が失敗したとみなすまでの経過時間 (ミリ秒)。

元のフィールド	新しい設定クラス	説明
workerIdentifier	ConfigsBuilder	このアプリケーションプロセッサのインスタンス化を表す一意の識別子。一意である必要があります。
shardSyncIntervalMillis	LeaseManagementConfig	シャード同期コールの間隔。
maxRecords	PollingConfig	Kinesis が返すレコードの最大数の設定を許可します。
idleTimeBetweenReadsInMillis	CoordinatorConfig	このオプションは削除されました。アイドル時間の削除を参照してください。
callProcessRecordsEvenForEmptyRecordList	ProcessorConfig	設定すると、Kinesis から提供されたレコードがない場合でもレコードプロセッサが呼び出されます。
parentShardPollIntervalMillis	CoordinatorConfig	親シャードが完了したかどうかを確認するためにレコードプロセッサがポーリングを行う頻度。
cleanupLeasesUponShardCompletion	LeaseManagementConfig	設定すると、子リースの処理が開始されると即時にリースが削除されます。
ignoreUnexpectedChildShards	LeaseManagementConfig	設定すると、開いているシャードがある子シャードは無視されます。これは、主に DynamoDB Streams 用です。
kinesisClientConfig	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。

元のフィールド	新しい設定クラス	説明
dynamoDBClientConfig	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
cloudWatchClientConfig	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
taskBackoffTimeMillis	LifecycleConfig	失敗したタスクを再試行するまでの待機時間。
metricsBufferTimeMillis	MetricsConfig	CloudWatch メトリックスの発行を制御します。
metricsMaxQueueSize	MetricsConfig	CloudWatch メトリックスの発行を制御します。
metricsLevel	MetricsConfig	CloudWatch メトリックスの発行を制御します。
metricsEnabledDimensions	MetricsConfig	CloudWatch メトリックスの発行を制御します。
validateSequenceNumberBeforeCheckpointing	CheckpointConfig	このオプションは削除されました。チェックポイントシーケンス番号の検証を参照してください。
regionName	ConfigsBuilder	このオプションは削除されました。クライアント設定の削除を参照してください。
maxLeasesForWorker	LeaseManagementConfig	アプリケーションの単一のインスタンスが受け入れるリースの最大数。

元のフィールド	新しい設定クラス	説明
maxLeasesToStealAtOneTime	LeaseManagementConfig	アプリケーションが同時にスティールを試みるリースの最大数。
initialLeaseTableReadCapacity	LeaseManagementConfig	Kinesis Client Library で新しい DynamoDB リーステーブルを作成する場合に使用する DynamoDB 読み取り IOPS。
initialLeaseTableWriteCapacity	LeaseManagementConfig	Kinesis Client Library で新しい DynamoDB リーステーブルを作成する場合に使用する DynamoDB 読み取り IOPS。
initialPositionInStreamExtended	LeaseManagementConfig	アプリケーションが読み取りを開始するストリーム内の初期位置。これは最初のリースの作成時にのみ使用されます。
skipShardSyncAtWorkerInitializationIfLeasesExist	CoordinatorConfig	リーステーブルに既存のリースがある場合、シャードデータの同期を無効にします。TODO: KinesisEco-438
shardPrioritization	CoordinatorConfig	どのシャードの優先順位付けを使用するか。
shutdownGraceMillis	該当なし	このオプションは削除されました。MultiLang の削除を参照してください。
timeoutInSeconds	該当なし	このオプションは削除されました。MultiLang の削除を参照してください。
retryGetRecordsInSeconds	PollingConfig	GetRecords が失敗した場合の試行間隔の遅延時間を設定します。

元のフィールド	新しい設定クラス	説明
maxGetRecordsThreadPool	PollingConfig	GetRecords に使用されるスレッドプールのサイズ。
maxLeaseRenewalThreads	LeaseManagementConfig	リース更新スレッドプールのサイズを制御します。アプリケーションが処理するリースの数が多いほど、このプールも大きくする必要があります。
recordsFetcherFactory	PollingConfig	ストリームから取得するフェッチャーを作成するために使用されるファクトリーの置換を許可します。
logWarningForTaskAfterMillis	LifecycleConfig	タスクが完了していない場合に警告がログに記録されるまでの待機期間。
listShardsBackoffTimeInMillis	RetrievalConfig	障害が発生した場合に ListShards を呼び出す間隔 (ミリ秒)。
maxListShardsRetryAttempts	RetrievalConfig	失敗とみなすまでの ListShards の再試行の最大回数。

アイドル時間の削除

KCL の 1.x バージョンでは、idleTimeBetweenReadsInMillis は 2 つの数量に相当します。

- タスクの送信チェックの間隔。CoordinatorConfig#shardConsumerDispatchPollIntervalMillis を設定することで、タスク間の間隔を設定できるようになりました。
- Kinesis Data Streams から返されるレコードがない場合に休止状態になるまでの時間。バージョン 2.0 では、拡張ファンアウトのレコードはそれぞれのレトリバーからプッシュされます。シャードコンシューマーのアクティビティは、プッシュされたリクエストが到着した場合にのみ発生します。

クライアント設定の削除

バージョン 2.0 では、KCL はクライアントを作成しなくなりました。有効なクライアントの提供はユーザーに任されます。この変更により、クライアントの作成を制御するすべての設定パラメータが削除されました。これらのパラメータが必要な場合は、クライアントを ConfigsBuilder に提供する前にクライアントで設定できます。

削除されたフィールド	同等の設定
kinesisEndpoint	優先エンドポイントを指定した SDK KinesisAsyncClient の設定: KinesisAsyncClient.builder().endpointOverride(URI.create("https://<kinesis endpoint>")).build() 。
dynamoDBEndpoint	優先エンドポイントを指定した SDK DynamoDbAsyncClient の設定: DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://<dynamodb endpoint>")).build() 。
kinesisClientConfig	必要な設定を指定した SDK KinesisAsyncClient の設定: KinesisAsyncClient.builder().overrideConfiguration(<your configuration>).build()
dynamoDBClientConfig	必要な設定を指定した SDK DynamoDbAsyncClient の設定: DynamoDbAsyncClient.builder().overrideConfiguration(<your configuration>).build()
cloudWatchClientConfig	必要な設定を指定した SDK CloudWatchAsyncClient の設定: CloudWatchAsyncClient.builder().overrideConfiguration(<your configuration>).build()
regionName	優先リージョンを指定して SDK を設定します。これは、すべての SDK クライアントで同じです。例えば、KinesisAsyncClient.builder().region(Region.US_WEST_2).build() です。

Kinesis Data Streams からデータを読み取るためのその他の AWS サービスの使用

以下は、Kinesis Data Streams データを読み取るために Kinesis Data Streams と直接統合できる、その他の AWS サービスのリストです。

トピック

Amazon EMR の使用

Amazon EMR クラスターは、Hive、Pig、Hadoop ストリーミング API、カスケーディングなどの Hadoop エコシステム内の使い慣れたツールを使用して Amazon Kinesis ストリームを直接読み取って処理できます。 MapReduceまた、実行しているクラスターで Amazon Kinesis のリアルタイムデータを Amazon S3、Amazon DynamoDB、および HDFS の既存データと結合できます。後処理のアクティビティとして、Amazon EMR から Amazon S3 または DynamoDB に直接データをロードできます。

詳細については、「Amazon EMR Release Guide」の「Amazon Kinesis」を参照してください。

Amazon EventBridge パイプの使用

Amazon EventBridge Pipes は Amazon Kinesis Data Streams をソースとしてサポートしています。Amazon EventBridge Pipes では、オプションの変換、フィルタ、エンリッチのステップを使用して、 point-to-point イベントプロデューサーとコンシューマー間のインテグレーションを作成できます。 EventBridge パイプを使用して Kinesis データストリームでレコードを受信し、オプションでこれらのレコードをフィルタリングまたは拡張してから、Kinesis Data Streams などの利用可能な処理先に送信できます。

詳細については、『Amazon EventBridge リリースガイド』の「ソースとしての Amazon Kinesis ストリーム」を参照してください。

AWS Glue を使用する

AWS Glue のストリーミング ETL を使用することで、継続的に実行され、Amazon Kinesis Data Streams からのデータを消費する、ストリーミング抽出、変換、ロード (ETL) ジョブを作成できます。ジョブはデータをクレンジングして変換し、その結果を Amazon S3 データレイクまたは JDBC データストアにロードします。

詳細については、「AWS Glue リリースガイド」の「AWS Glue でのストリーミング ETL ジョブ」を参照してください。

Amazon Redshift の使用

Amazon Redshift は、Amazon Kinesis Data Streams からのストリーミング取り込みをサポートします。Amazon Redshift のストリーミング取り込み機能は、Amazon Kinesis Data Streams からのストリーミングデータの Amazon Redshift マテリアライズドビューへの低レイテンシーかつ高速な取り込みを実現します。Amazon Redshift のストリーミング取り込みにより、Amazon Redshift への取り込み前に Amazon S3 でデータをステージングする必要がなくなります。

詳細については、「Amazon Redshift リリースガイド」の「ストリーミング取り込み」を参照してください。

サードパーティー統合の使用

Amazon Kinesis Data Streams データストリームからデータを読み取るには、Kinesis Data Streams と統合されている以下のサードパーティーオプションのいずれかを使用します。

トピック

- Apache Flink
- Adobe Experience Platform
- Apache Druid
- Apache Spark
- Databricks
- Kafka Confluent Platform
- Kinesumer
- Talend

Apache Flink

Apache Flink は、制限なしおよび制限付きのデータストリームでのステートフル計算のためのフレームワークかつ分散処理エンジンです。Apache Flink を使用した Kinesis データストリームの消費に関する詳細については、「Amazon Kinesis Data Streams Connector」を参照してください。

Adobe Experience Platform

Adobe Experience Platform は、組織があらゆるシステムからの顧客データを一元化して標準化することを可能にします。その後、データサイエンスと機械学習を適用して、充実感のあるパーソナライズされたエクスペリエンスの設計と提供を劇的に向上させます。Adobe Experience Platform を使用した Kinesis データストリームの使用の詳細については、「Amazon Kinesis connector」を参照してください。

Apache Druid

Druid は、ストリーミングとバッチデータに対する 1 秒未満のクエリを、大規模に、かつ負荷がかかった状態で実現する高性能のリアルタイム分析データベースです。Apache Druid を使用した Kinesis データストリームの取り込みの詳細については、「Amazon Kinesis ingestion」を参照してください。

Apache Spark

Apache Spark は、大規模データ処理のための統合分析エンジンです。Java、Scala、Python、および R の高レベルな API と、汎用実行グラフをサポートする最適化されたエンジンを提供します。Apache Spark を使用して、Kinesis データストリーム内のデータを消費するストリーム処理アプリケーションを構築できます。

Apache Spark 構造化ストリーミングを使用して Kinesis データストリームを使用するには、Amazon Kinesis Data Streams コネクタを使用します。このコネクタは、拡張ファンアウトによる消費をサポートします。これにより、アプリケーションはシャードあたり 1 秒あたり最大 2 MB のデータという専用読み取りスループットを提供します。詳細については、「専有スループット (拡張ファンアウト) を使用したカスタムコンシューマーの開発」を参照してください。

Spark Streaming を使用して Kinesis データストリームを使用するには、「Spark Streaming + Kinesis Integration」を参照してください。

Databricks

Databricks は、データエンジニアリング、データサイエンス、および機械学習のための共同環境を提供するクラウドベースのプラットフォームです。Databricks を使用して Kinesis データストリームを使用する方法の詳細については、「Amazon Kinesis に接続する」を参照してください。

Kafka Confluent Platform

Confluent Platform は Kafka 上に構築されており、エンタープライズがリアルタイムのデータパイプラインおよびストリーミングアプリケーションを構築して管理するために役立つ追加機能を提供します。Confluent Platform を使用して Kinesis データストリームを使用する方法の詳細については、「Amazon Kinesis Source Connector for Confluent Platform」を参照してください。

Kinesumer

Kinesumer は、Kinesis データストリーム用のクライアント側の分散コンシューマーグループクライアントを実装する Go クライアントです。詳細については、「Kinesumer Github リポジトリ」を参照してください。

Talend

Talend は、ユーザーがさまざまなソースからのデータをスケーラブルかつ効率的な方法で収集、変換、および接続することを可能にするデータ統合およびデータ管理ソフトウェアです。Talend を使用して Kinesis データストリームを使用する方法の詳細については、「Amazon Kinesis ストリームに talend を接続する」を参照してください。

Kinesis データストリームコンシューマーのトラブルシューティング

以下のセクションでは、Amazon Kinesis Data Streams コンシューマーの操作中に発生する可能性がある一般的な問題に対する解決策を示します。

- Kinesis クライアントライブラリの使用時に一部の Kinesis Data Streams レコードがスキップされる
- 同じシャードに属するレコードが、異なるレコードプロセッサによって同時に処理される
- コンシューマーアプリケーションの読み取りの速度が予想よりも遅い
- GetRecords ストリームにデータがある場合でも、空のレコード配列を返します。

- シャードイテレータが予期せずに終了する
- コンシューマーレコードの処理が遅れる
- 承認されていない KMS マスターキーの権限エラー
- コンシューマーにとって一般的な問題、質問、トラブルシューティングのアイデア

Kinesis クライアントライブラリの使用時に一部の Kinesis Data Streams レコードがスキップされる

レコードがスキップされる最も一般的な原因は、processRecords からスローされる処理されない例外です。Kinesis Client Library (KCL) は、processRecords コードを使用して、データレコードの処理で発生するすべての例外を処理します。processRecords からスローされるすべての例外は、 KCLによって吸収されます。反復的なエラーに対する無限再試行を回避するために、KCL では例外の発生時に処理中であったレコードのバッチを再送信しません。KCL は、レコードプロセッサを再起動することなく、データレコードの次のバッチで processRecords を呼び出します。これにより、事実上、コンシューマーアプリケーションではレコードがスキップされたことになります。レコードのスキップを防止するには、processRecords 内ですべての例外を適切に処理します。

同じシャードに属するレコードが、異なるレコードプロセッサによって同時に処理される

実行されている Kinesis Client Library (KCL) アプリケーションでは、シャードの所有者はひとりだけです。ただし、複数のレコードプロセッサが一時的に同じシャードを処理する場合があります。ネットワーク接続を紛失したワーカーインスタンスの場合、CL はフェイルオーバー時間の期限が切れた後に、到達できないワーカーはレコードを処理していないと仮定し、他のワーカーインスタンスが引き継ぐように指示します。このとき短時間ですが、新しいレコードプロセッサと到達不可能なワーカーのレコードプロセッサが同じシャードのデータを処理する場合があります。

アプリケーションに適したフェイルオーバー時間を設定する必要があります。低レイテンシーアプリケーションの場合、10秒のデフォルトは、待機する最大時間を表している場合があります。ただし、より頻繁に接続が失われる地域で通話を行うなどの接続問題が予想される場合、この数値は低すぎる場合があります。

ネットワーク接続は通常、以前の到達不可能なワーカーに復元されるため、アプリケーションではこのシナリオを予期して処理する必要があります。レコードプロセッサのシャードが別のレコードプロセッサに引き継がれた場合、レコードプロセッサは正常なシャットダウンを実行するために次の 2 つのケースを処理する必要があります。

1. processRecords への現在の呼び出しが完了した後で、KCL はシャットダウンの理由ZOMBIE を使用してレコードプロセッサでシャットダウンメソッドを呼び出します。レコードプロセッサは、すべてのリソースを必要に応じて適切にクリーンアップした後、終了する必要があります。
2. zombieワーカーからチェックポイントを作成しようとすると、KCL は ShutdownException をスローします。この例外を受け取った後、コードは現在のメソッドを正常に終了する必要があります。

詳細については、重複レコードの処理を参照してください。

コンシューマーアプリケーションの読み取りの速度が予想よりも遅い

読み取りのスループットが予想よりも遅くなる最も一般的な理由は次のとおりです。

1. 複数のコンシューマーアプリケーションの読み取りの合計が、シャードごとの制限を超えています。詳細については、クォータと制限を参照してください。この場合、Kinesis データストリームのシャードの数が増えます。
2. 呼び出しごとの GetRecords の最大数を指定する制限が、低い値で設定されている可能性があります。KCL を使用している場合は、ワーカーに設定した maxRecords プロパティの値が低い可能性があります。一般的に、このプロパティにはシステムのデフォルトを使用することをお勧めします。
3. processRecords 呼び出し内のロジックに予想よりも時間がかかる場合があります。これには、ロジックが CPU を大量に消費する、I/O をブロックする、同期のボトルネックになっているなど、多くの理由が考えられます。これに該当するかどうかをテストするには、空のレコードプロセッサをテスト実行し、読み取りスループットを比較します。受信データに遅れずに対応する方法については、リシャーディング、スケーリング、並列処理を参照してください。

コンシューマーアプリケーションが 1 つのみである場合、通常、PUT レートの少なくとも 2 倍高速に読み取りを実行できます。これは、書き込みに対して 1 秒あたり最大 1,000 レコードを書き込むことができ、最大合計データ書き込み速度が 1 秒あたり 1 MB (パーティションキーを含む) になるためです。オープンな各シャードは、読み取りに対して 1 秒あたり最大 5 トランザクションをサポートでき、最大合計データ読み取り速度は 1 秒あたり 2MB です。各読み取り (GetRecords) は、レコードのバッチを取得します。GetRecords によって返されるデータのサイズは、シャードの使用状況によって異なります。GetRecords が返すことができるデータの最大サイズは、10 MB です。呼び出しがその制限を返す場合、次の 5 秒以内に行われるそれ以降の呼び出しは ProvisionedThroughputExceededException をスローします。

GetRecords ストリームにデータがある場合でも、空のレコード配列を返します。

レコードの消費、つまり取得は、プルモデルです。デベロッパーは、バックオフなしで連続ループGetRecordsで を呼び出すことが期待されます。GetRecords のすべての呼び出しは、ShardIterator 値も返します。この値は、ループの次のイテレーションで使用する必要があります。

GetRecords オペレーションはブロックしません。その代わりに、関連データレコードまたは空の Records 要素とともに、直ちに制御を戻します。空の Records 要素は、2 つの条件の下で返されます。

1. 現在シャードにはそれ以上のデータがない。
2. シャードの ShardIterator で指定されたパートの近くにデータがない。

後者の条件は微妙ですが、レコードを取得するときに無限のシーク時間 (レイテンシー) を回避するために必要な設計上のトレードオフです。そのため、ストリームを使用するアプリケーションはループし、GetRecords を呼び出して、当然のこととして空のレコードを処理します。

本稼働シナリオで、連続ループが終了するのは、NextShardIterator の値が NULL である場合のみにする必要があります。NextShardIterator が NULL である場合、現在のシャードが閉じられ、ShardIterator 値は最後のレコードを過ぎたことを示します。コンシューマーアプリケーションが SplitShard または MergeShards を呼び出さない場合、シャードは開いたままになり、GetRecords の呼び出しは NextShardIterator である NULL 値を返しません。

Kinesis Client Library (KCL) を使用する場合、お客様に対しては前述の消費パターンは抽象化されます。これには、動的に変更する一連のシャードの自動処理が含まれます。KCL により、デベロッパーは入力レコードを処理するロジックのみを提供します。ライブラリが自動的に GetRecords の継続的な呼び出しを行うため、これが可能になります。

シャードイテレータが予期せずに終了する

新しいシャードのイテレータは、GetRecords リクエスト (NextShardIterator として) 返されます。これは次の GetRecordsリクエスト (ShardIterator として) 使用します。通常の場合、このシャードイテレーターは使用する前に有効期限が切れることはありません。ただし、5 分以上 GetRecords を呼び出さなかったため、またはコンシューマーアプリケーションの再起動を実行したため、シャードイテレータの有効期限が切れる場合があります。

シャードイテレーターの有効期限がすぐに切れて使用できない場合、これは Kinesis で使用している DynamoDB テーブルの容量不足でリースデータを保存できないことを示している可能性があります。この状況は、多数のシャードがある場合により発生する可能性が高くなります。この問題を解決するには、シャードテーブルに割り当てられた書き込み容量を増やします。詳細については、リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するを参照してください。

コンシューマーレコードの処理が遅れる

ほとんどのユースケースで、コンシューマーアプリケーションはストリームから最新のデータを読み取ります。特定の状況下では、コンシューマーの読み取りが遅れるという好ましくない事態が発生します。コンシューマーの読み取りの遅れ具合を確認したら、遅れの最も一般的な理由を参照してください。

`GetRecords.IteratorAgeMilliseconds` メトリクスを起動して、ストリーム内のすべてのシャードとコンシューマーの読み取り位置を追跡します。イテレータの経過日数が保持期間 (デフォルトで 24 時間、最大で 365 日まで設定可能) の 50% を経過すると失効する場合、レコードの有効期限切れによるデータ損失のリスクがあります。とりあえずの解決策は、保持期間を長くすることです。これにより、問題のトラブルシューティングを行う間に重要なデータが失われるのを防ぎます。詳細については、「Amazon による Amazon Kinesis Data Streams Service のモニタリング CloudWatch」を参照してください。次に、Kinesis Client Library (KCL)、 によって出力されるカスタム CloudWatch メトリクスを使用して、コンシューマーアプリケーションが各シャードから読み取っている距離を特定します`MillisBehindLatest`。詳細については、「Amazon による Kinesis Client Library のモニタリング CloudWatch」を参照してください。

コンシューマーが遅れる最も一般的な理由:

- `GetRecords.IteratorAgeMilliseconds` の突然の上昇または `MillisBehindLatest` は、通常ダウンストリームアプリケーションに対する API オペレーションの障害などの一時的な問題を示します。どちらかのメトリクスが恒常的にこのような動きを示す場合、この急激な上昇を調査する必要があります。
- これらのメトリクスが徐々に上昇する場合は、レコードの処理速度が不十分なためストリームにコンシューマーが追いついていないことを示します。この状況に共通の原因は、物理リソースの不足またはストリームスループットの上昇にレコード処理ロジックが追随できないことです。この動作を確認するには、、`RecordProcessor.processRecords.Time`、 `Success`など、KCL が `processTask`オペレーションに関連付けられた他のカスタム CloudWatchメトリクスを確認します`RecordsProcessed`。

- スループットの増加に伴う processRecords.Time メトリクスの上昇が確認された場合、レコード処理ロジックを分析して、スループットの増加に対応したスケーリングができない理由を調べる必要があります。
- スループットの上昇とは関連性がない processRecords.Time 値の上昇が認められた場合は、重要なパスでブロック呼び出しを行っていないか確認します。これは、レコード処理の低下を招きます。代替策として、シャードの数を増やして並列処理を増やす方法があります。最後に、ピーク需要時に適切な容量の物理リソース (メモリ、CPU 使用率など) が基盤の処理ノードに存在することを確認します。

承認されていない KMS マスターキーの権限エラー

このエラーは、KMS マスターキーのアクセス許可なしで、コンシューマーアプリケーションが暗号化されたストリームから読み取りを行ったときに発生します。KMS キーにアクセスする許可をアプリケーションに割り当てるには、AWS KMS でのキーポリシーの使用およびAWS KMS での IAM ポリシーの使用を参照してください。

コンシューマーにとって一般的な問題、質問、トラブルシューティングのアイデア

- Kinesis Data Streams トリガーが Lambda 関数を呼び出すことができないのはなぜですか?
- Kinesis Data Streams で例外を検出してトラブルシューティング ReadProvisionedThroughputExceededする方法を教えてください。
- Kinesis Data Streams で高レイテンシーの問題が発生するのはなぜですか?
- Kinesis データストリームで 500 内部サーバーエラーが返されるのはなぜですか?
- Kinesis Data Streams 用のブロックまたはスタックされた KCL アプリケーションのトラブルシューティング方法を教えてください。
- 同じ Amazon DynamoDB テーブルで異なる Amazon Kinesis クライアントライブラリアプリケーションを使用できますか ?

Amazon Kinesis Data Streams コンシューマー向けの高度なトピック

Amazon Kinesis Data Streams コンシューマーを最適化する方法を説明します。

コンテンツ

低レイテンシー処理

伝播遅延は、 end-to-end レコードがストリームに書き込まれた瞬間からコンシューマーアプリケーションによって読み取られるまでの待ち時間として定義されます。この遅延はいくつかの要因によって異なりますが、最も大きく影響するのはコンシューマーアプリケーションのポーリング間隔です。

ほとんどのアプリケーションについては、アプリケーションごとに各シャードを 1 秒 1 回ポーリングすることをお勧めします。この設定では、Amazon Kinesis Data Streams の制限 (1 秒あたり 5 回の`GetRecords` 呼び出し) を超えることなく、複数のコンシューマーアプリケーションで同時に 1 つのストリームを処理できます。また、処理するデータバッチが大きいほど、アプリケーション内のネットワークおよびその他ダウンストリームのレイテンシーをより効率的に短縮できる傾向があります。

KCL のデフォルト値は、毎秒のポーリングのベストプラクティスに従うよう設定されています。このデフォルト設定により、平均的な伝達遅延が通常 1 秒未満になります。

Kinesis Data Streams レコードは、書き込まれた後、すぐに読み取り可能になります。ユースケースには、この性能を活用して、ストリームが使用可能になり次第、ストリームからデータを使用することが必要なものもあります。次の例に示されているように、KCL のデフォルト設定を上書きしてポーリングの頻度を高くすると、伝達遅延を大幅に短縮できます。

Java KCL 設定コードを次に示します。

```
kinesisClientLibConfiguration = new
        KinesisClientLibConfiguration(applicationName,
        streamName,
        credentialsProvider,

 workerId).withInitialPositionInStream(initialPositionInStream).withIdleTimeBetweenReadsInMilli
```

Python および Ruby KCL のプロパティファイル設定を次に示します。

```
idleTimeBetweenReadsInMillis = 250
```

> **Note**
>
> Kinesis Data Streams は、GetRecords コールをシャードごとに 1 秒あたり 5 回に制限しているため、idleTimeBetweenReadsInMillis プロパティを 200 ms 未満に設定すると、アプリケーションで ProvisionedThroughputExceededException 例外が発生する可能性があります。この例外の発生回数が多くなりすぎると、エクスポネンシャルバックオフが発生することになり、処理中の予期しない大幅なレイテンシーの原因になります。このプロパティを 200 ms またはそれより少し高く設定した場合も、処理中のアプリケーションが複数あれば、同様のスロットリングが発生します。

Kinesis プロデューサーライブラリでの使用 AWS Lambda

Kinesis Producer Library (KPL) は、小さなユーザーフォーマットレコードを最大 1 MB のレコードに集約して、Amazon Kinesis Data Streams スループットを有効に利用できます。KCL for Java はこれらのレコードの非集計をサポートしていますが、 AWS Lambda ストリームのコンシューマーとして使用する場合は、特別なモジュールを使用してレコードを非集計する必要があります。必要なプロジェクトコードと手順は、Lambda 用 Kinesis GitHub プロデューサーライブラリデアグリゲーションモジュールから入手できます。 AWS このプロジェクトのコンポーネントを使用すると、KPL のシリアル化されたデータを Java、Node.js AWS Lambda、および Python 内で処理できます。これらのコンポーネントは、複数言語の KCL アプリケーションの一部として使用することもできます。

リシャーディング、スケーリング、並列処理

リシャーディングによって、ストリームのデータフロー率の変化に合わせて、ストリーム内のシャードカウントを増減できます。通常、リシャーディングはシャードのデータ処理メトリクスを監視する管理アプリケーションによって実行されます。KCL 自体はリシャーディングオペレーションを開始しませんが、リシャーディングに起因するシャードの数の変化に適応するように設計されています。

リーステーブルを使用して KCL コンシューマーアプリケーションによって処理されたシャードを追跡するで説明したように、KCL は Amazon DynamoDB tテーブルを使用してストリーム内のシャードを追跡します。リシャーディングの結果として新しいシャードが作成されるときに、KCL は新しいシャードを検出し、テーブル内の新しい行に値を入力します。ワーカーは、新しいシャードを自動的に検出して、それらからのデータを処理するためのプロセッサを作成します。また、KCL は、ストリーム内のシャードを、利用可能なすべてのワーカーとレコードプロセッサに分散させます。

KCL は、リシャーディング前にシャードに存在していたすべてのデータが最初に処理されるようにします。このデータが処理されると、新しいシャードからのデータがレコードプロセッサに送信されます。このようにして、KCL は、データレコードが特定のパーティションキーのストリームに追加された順序を保持します。

例: リシャーディング、スケーリング、並列処理

次の例は、KCL を使用してスケーリングとリシャーディングを処理する方法を示しています。

- アプリケーションが 1 つの EC2 インスタンスで実行中であり、4 つのシャードを含む 1 つの Kinesis Data Streams を処理しているとします。この 1 つのインスタンスに 1 つの KCL ワーカーと、4 つのレコードプロセッサ (各シャードに 1 つのレコードプロセッサ) があります。これらの 4 つのレコードプロセッサは、同一のプロセス内で並列実行されます。
- 次に、別のインスタンスを使用するようにアプリケーションをスケールし、4 つのシャードが含まれる 1 つのストリームを 2 つのインスタンスが処理するとします。KCL ワーカーが 2 番目のインスタンスで起動すると、最初のインスタンスとの間で負荷分散が行われ、各インスタンスで 2 つのシャードが処理されるようになります。
- その後、4 つのシャードを 5 つのシャードに分割するとします。KCL は再度インスタンスでの処理を調整します。一方のインスタンスが 3 つのシャードを処理し、もう一方のインスタンスが 2 つのシャードを処理するように調整されます。シャードをマージするときにも、同様の調整が行われます。

通常、KCL を使用する場合、インスタンスの数がシャードの数を超過しないように注意します (障害に対するスタンバイを目的とする場合を除く)。各シャードは厳密に 1 つの KCL ワーカーによって処理され、対応するレコードプロセッサが厳密に 1 つ存在するため、1 つのシャードを処理するために複数のインスタンスが必要になることはありません。ただし、1 つのワーカーで任意の数のシャードを処理できるため、シャードの数がインスタンスの数を超えても問題はありません。

アプリケーションでの処理をスケールアップするには、次のようなアプローチの組み合わせをテストするようにしてください。

- インスタンスのサイズを大きくする (プロセス内ではすべてのレコードプロセッサが並列実行されるため)
- インスタンスの数をオープンシャードの最大数まで増やす (シャードは個別に処理できるため)
- シャードの数を増やす (並列性のレベルを向上させる)

Auto Scaling を使用すると、適切なメトリクスに基づいて自動的にインスタンスを拡張できます。詳細については、Amazon EC2 Auto Scaling ユーザーガイドを参照してください。

リシャーディングによってストリーム内のシャードの数が増加すると、レコードプロセッサの数もそれに合わせて増加するため、これらをホストする EC2 インスタンスの負荷が高くなります。インスタンスが Auto Scaling グループの一部であり、負荷の増加が基準を満たす場合は、Auto Scaling グループがインスタンスを追加して増加した負荷を処理します。新しいインスタンスで追加のワーカーやレコードプロセッサがすぐにアクティブになるように、インスタンスの起動時に Amazon Kinesis Data Streams アプリケーションを起動するように設定してください。

リシャーディングの詳細については、ストリームをリシャーディングするを参照してください。

重複レコードの処理

レコードが複数回 Amazon Kinesis Data Streams アプリケーションに配信される理由は、主にプロデューサーの再試行とコンシューマーの再試行の 2 つになります。アプリケーションは、個々のレコードを複数回処理することを見込んで、適切に処理する必要があります。

プロデューサーの再試行

プロデューサーで `PutRecord` を呼び出してから Amazon Kinesis Data Streams の受信確認を受け取るまでの間に、ネットワーク関連のタイムアウトが発生する場合があります。この場合、プロデューサーはレコードが Kinesis Data Streams に配信されたかどうかを確認できません。各レコードがアプリケーションにとって重要であれば、同じデータを使用して呼び出しを再試行するようにプロデューサーが定義されているはずです。同じデータを使用した `PutRecord` の呼び出しが両方とも Kinesis Data Streams に正常にコミットされると、Kinesis Data Streams レコードは 2 つになります。2 つのレコードには同一のデータがありますが、一意のシーケンス番号も付けられています。厳密な保証を必要とするアプリケーションは、レコード内にプライマリキーを埋め込んで、後ほど処理するときに重複を削除する必要があります。プロデューサーの再試行に起因する重複の数が、コンシューマーの再試行に起因する重複の数より通常は少ないことに注意してください。

> **Note**
>
> SDK を使用する場合は`PutRecord`、 AWS SDK とツールのユーザーガイドで SDK リトライ動作について学んでください。AWS

コンシューマーの再試行

コンシューマー (データ処理アプリケーション) の再試行は、レコードプロセッサが再開するときに発生します。同じシャードのレコードプロセッサは、次の場合に再開します。

1. ワーカーが予期せず終了する
2. ワーカーインスタンスが追加または削除される
3. シャードがマージまたは分割される
4. アプリケーションがデプロイされる

いずれの場合も、 shards-to-worker-to-record-processor マッピングは負荷分散処理のために継続的に更新されます。他のインスタンスに移行されたシャードプロセッサは、最後のチェックポイントからレコードの処理を再開します。これにより、以下の例にあるような重複レコード処理が発生します。負荷分散の詳細については、リシャーディング、スケーリング、並列処理を参照してください。

例: コンシューマーの再試行によるレコードの再配信

この例では、ストリームから継続的にレコードを読み取り、ローカルファイルにレコードを集約し、このファイルを Amazon S3 にアップロードするアプリケーションがあるとします。分かりやすくするため、1 つのシャードと、このシャードを処理する 1 つのワーカーのみがあるとします。最後のチェックポイントがレコード番号 10,000 であると仮定して、次の例の一連のイベントを考えてみます。

1. ワーカーで、シャードから次のレコードのバッチを読み込みます (1,0001 から 20000)。
2. 次に、ワーカーがレコードのバッチを関連付けられたレコードプロセッサに渡します。
3. レコードプロセッサはデータを集約し、Amazon S3 ファイルを作成して、このファイルを Amazon S3 に正常にアップロードします。
4. 新しいチェックポイントが生成される前に、ワーカーが予期せず終了します。
5. アプリケーション、ワーカー、およびレコードプロセッサが再開します。
6. ワーカーは、正常な最後のチェックポイント (この場合は 1,0001) から読み込みを開始しました。

したがって、1,0001 から 20000 のレコードは複数回使用されます。

コンシューマーの再試行に対する弾力性

レコードが複数回処理される可能性はあるものの、アプリケーションでは、レコードが 1 回だけ処理されたかのような付随効果 (冪等処理) を提示する場合があります。この問題に対するソリュー

ションは、複雑性と正確性に応じて異なります。最終的なデータの送信先が重複を適切に処理できる場合は、冪等処理の実行は最終送信先に任せることをお勧めします。例えば、Opensearch では、バージョニングと一意の ID の組み合わせを使用して重複処理を防ぐことができます。

前セクションのアプリケーション例では、ストリームから継続的にレコードを読み取り、レコードをローカルファイルに集約して、ファイルを Amazon S3 にアップロードします。図に示すように、1,0001 から 20000 のレコードが複数回使用されることにより、複数の Amazon S3 ファイルのデータは同じになります。この例からの重複を軽減する方法の 1 つは、ステップ 3 での次のスキーマの使用を確実にすることです。

1. レコードプロセッサは、各 Amazon S3 ファイルに固定のレコード番号 (5000 など) を使用します。
2. ファイル名には、このスキーマ (Amazon S3 プレフィックス、シャード ID、および `First-Sequence-Num`) を使用します。この場合は、`sample-shard000001-10001` のようになります。
3. Amazon S3 ファイルをアップロードした後で、`Last-Sequence-Num` を指定してチェックポイントを作成します。この場合は、レコード番号 15000 にチェックポイントが作成されます。

このスキーマを使用すると、レコードが複数回処理されても、Amazon S3 ファイルには同じ名前と同じデータが保持されます。再試行しても、同じファイルに同じデータが複数回書き込まれるだけになります。

リシャーディング操作の場合は、シャードに残っているレコードの数が必要な一定数よりも少ないことがあります。この場合、`shutdown()` メソッドは Amazon S3 にファイルをフラッシュし、最後のシーケンス番号でチェックポイントを作成する必要があります。上記のスキーマは、リシャーディング操作との互換性もあります。

起動、シャットダウン、スロットリングの処理

ここでは、Amazon Kinesis Data Streams アプリケーションの設計に取り入れる必要がある、追加の考慮事項を示します。

コンテンツ

データプロデューサーとデータコンシューマーの起動

デフォルトでは、KCL はストリームの末尾 (最後に追加されたレコード) からレコードの読み込みを開始します。この設定では、受信側のレコードプロセッサが実行される前に、データプロデューサーアプリケーションがストリームにレコードを追加した場合、レコードプロセッサが起動した後、これらのレコードはレコードプロセッサによって読み込まれません。

レコードプロセッサの動作を変更して、常にストリームの先頭からデータを読み込むには、Amazon Kinesis Data Streams アプリケーションの properties ファイルで次の値を設定します。

```
initialPositionInStream = TRIM_HORIZON
```

デフォルトでは、Amazon Kinesis Data Streams はすべてのデータを 24 時間保存します。また、最大 7 日間の延長保存と最大 365 日間の長期保存もサポートします。この期間は保持期間と呼ばれます。開始位置を TRIM_HORIZON に設定すると、保持期間で定義されているとおりに、ストリーム内の最も古いデータでレコードプロセッサが起動します。TRIM_HORIZON に設定しても、保持期間を上回る時間が経過した後でレコードプロセッサが起動される場合は、ストリーム内のレコードの一部が利用できなくなります。このため、コンシューマーアプリケーションには常にストリームからデータを読み込ませ、 CloudWatch GetRecords.IteratorAgeMilliseconds メトリックを使用してアプリケーションが受信データを処理しているかどうかを監視する必要があります。

シナリオによっては、レコードプロセッサがストリームの最初の数レコードを処理しなくても問題ない場合があります。たとえば、 end-to-end ストリームが期待どおりに機能していることをテストするために、いくつかの初期レコードをストリームに通す場合があります。この初期確認を行った後でワーカーを起動し、ストリームへの本番データの送信を開始します。

TRIM_HORIZON の設定の詳細については、シャードイテレーターの使用を参照してください。

Amazon Kinesis Data Streams アプリケーションのシャットダウン

Amazon Kinesis Data Streams アプリケーションが目的のタスクを完了したら、アプリケーションが実行されている EC2 インスタンスを終了することによって、アプリケーションをシャットダウンする必要があります。インスタンスは、AWS Management Console または AWS CLI を使用して終了することができます。

Amazon Kinesis Data Streams アプリケーションのシャットダウン後に、KCL がアプリケーションの状態を追跡するために使用した Amazon DynamoDB テーブルを削除する必要があります。

読み込みのスロットリング

ストリームのスループットは、シャードレベルでプロビジョニングされます。各シャードは、読み取りに対して 1 秒あたり最大 5 トランザクションのスループットで、最大合計データ読み取り速度は 1 秒あたり 2MB です。アプリケーション (または同じストリームで動作するアプリケーションのグループ) がシャードからデータをより高速に取得しようとすると、Kinesis Data Streams は対応する GET オペレーションを調整します。

Amazon Kinesis Data Streams アプリケーションでは、レコードプロセッサが制限よりも高速にデータを処理している場合 (フェイルオーバーの場合など)、スロットリングが発生します。KCL によってアプリケーションと Kinesis Data Streams とのやり取りが管理されるため、スロットリング例外は、アプリケーションコードではなく KCL コードで発生します。ただし、KCL によってこれらの例外がログに記録されるため、ログで例外を確認できます。

アプリケーションが絶えずスロットリングされていると思われる場合は、ストリームのシャード数を増やすことを検討してください。

Amazon Kinesis Data Streams のモニタリング

次の機能を使用して Amazon Kinesis Data Streams のデータストリームをモニタリングできます。

- CloudWatch メトリクス — Kinesis Data Streams は、各ストリームの詳細モニタリングを含む Amazon CloudWatch カスタムメトリクスを送信します。
- Kinesis Agent — Kinesis Agent は、エージェントが期待どおりに動作しているかどうかを評価するのに役立つカスタム CloudWatch メトリクスを発行します。
- API ログ記録 - Kinesis Data Streams は AWS CloudTrail を使用して API コールをログに記録し、そのデータを Amazon S3 バケットに保存します。
- Kinesis Client Library - Kinesis Client Library (KCL) は、シャード、ワーカー、および KCL アプリケーションごとのメトリクスを提供します。
- Kinesis Producer Library - Kinesis Producer Library (KPL) は、シャード、ワーカー、およびKPL アプリケーションごとのメトリクスを提供します。

一般的なモニタリングの問題、質問、およびトラブルシューティングの詳細については、以下を参照してください。

- Kinesis Data Streams の問題をモニタリングおよびトラブルシューティングするには、どのメトリクスを使用すべきですか?
- Kinesis Data Streams IteratorAgeMilliseconds の値が増え続けるのはなぜですか ?

Amazon による Amazon Kinesis Data Streams Service のモニタリング CloudWatch

Amazon Kinesis Data Streams と Amazon CloudWatch は統合されているため、Kinesis データストリームのメトリクスを収集、表示、分析 CloudWatchできます。たとえば、シャードの使用状況の追跡に、`IncomingBytes` メトリクス `OutgoingBytes` とメトリクスをモニタリングし、ストリーム内のシャードカウントと比較できます。

ストリーム用に設定したメトリクスは自動的に収集され、1 分 CloudWatch ごとに にプッシュされます。2 週間分のメトリクスがアーカイブされ、その期間が経過したデータは破棄されます。

次の表は、Kinesis data streams の基本的なストリームレベルと拡張シャードレベルのモニタリングについて説明しています。

型	説明
ベーシック (ストリームレベル)	ストリーム レベルのデータは、1 分間ごとに自動的に送信されます。料金は発生しません。
拡張 (シャードレベル)	シャードレベルのデータは、1分ごとに送信されます。追加料金が発生します。このレベルのデータを取得するには、 EnableEnhancedMonitoringオペレーションを使用してストリームに対してそのデータを有効にする必要があります。 料金の詳細については、「Amazon CloudWatch 製品ページ」を参照してください。

Amazon Kinesis Data Streams のディメンションとメトリクス

Kinesis Data Streams は、ストリームレベルと、オプション CloudWatch でシャードレベルの 2 つのレベルで にメトリクスを送信します。ストリームレベルのメトリクスは、通常の条件での最も一般的なモニタリングのユースケース用です。シャードレベルのメトリクスは、通常はトラブルシューティングに関連する特定のモニタリングタスク用であり、 EnableEnhancedMonitoringオペレーションを使用して有効になります。

CloudWatch メトリクスから収集された統計の説明については、「Amazon ユーザーガイド」のCloudWatch 「統計」を参照してください。 CloudWatch

トピック

- 基本ストリームレベルメトリクス
- 拡張シャードレベルメトリクス
- Amazon Kinesis Data Streams メトリクスのディメンション
- 推奨される Amazon Kinesis Data Streams メトリクス

基本ストリームレベルメトリクス

AWS/Kinesis 名前空間には、次のストリームレベルメトリクスが含まれます。

Kinesis Data Streams は、これらのストリームレベルのメトリクスを 1 分 CloudWatch ごとに に送信します。これらのメトリクスは常に利用することができます。

メトリクス	説明
GetRecords.Bytes	指定された期間に測定された、Kinesis ストリームから取得したバイト数。Minimum、Maximum、および Average の統計は、指定した期間内のストリームの単一 GetRecords オペレーションでのバイト数です。 シャードレベルメトリクス名: OutgoingBytes ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト
GetRecords.IteratorAge	このメトリクスは廃止されました。GetRecords.IteratorAgeMilliseconds を使用します。
GetRecords.IteratorAgeMilliseconds	Kinesis ストリームに対して行われたすべての GetRecords 呼び出しの最後のレコードの期間 (指定された時間に測定)。期間は、現在の時刻と、GetRecords 呼び出しの最後のレコードがストリームに書き込まれた時刻の差です。Minimum および Maximum 統計は、Kinesis コンシューマーアプリケーションのプロセスを追跡するのに使用できます。値がゼロの場合は、読み取り中のレコードがストリームに完全に追いついていることを示します。 シャードレベルメトリクス名: IteratorAgeMilliseconds ディメンション： StreamName 統計: Minimum、Maximum、Average、Samples 単位: ミリ秒

メトリクス	説明
GetRecords.Latency	指定された期間に測定された GetRecords オペレーションごとにかかった時間。 ディメンション： StreamName 統計: Minimum、Maximum、Average 単位: ミリ秒
GetRecords.Records	指定された期間に測定された、シャードから取得したレコード数。Minimum、Maximum、および Average の統計は、指定した期間内のストリームの単一 GetRecords オペレーションでのレコード数です。 シャードレベルメトリクス名: OutgoingRecords ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
GetRecords.Success	指定された期間に測定された、ストリームごとの成功した GetRecords オペレーションの数。 ディメンション： StreamName 統計: Average、Sum、Samples 単位: カウント

メトリクス	説明
IncomingBytes	指定された期間に、Kinesis ストリームに正常に送信されたバイト数。このメトリクスには、PutRecord　および PutRecords　オペレーションのバイト数も含まれます。Minimum、Maximum、および Average の統計は、指定した期間内のストリームの単一 put オペレーションでのバイト数です。 シャードレベルメトリクス名: IncomingBytes ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト
IncomingRecords	指定された期間に、Kinesis ストリームに正常に送信されたレコードの数。このメトリクスには、PutRecord および PutRecords　オペレーションのレコード数も含まれます。Minimum、Maximum、および Average の統計は、指定した期間内のストリームの単一 put オペレーションでのレコード数です。 シャードレベルメトリクス名: IncomingRecords ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
PutRecord.Bytes	指定された期間に PutRecord　オペレーションを使用して Kinesis ストリームに送信されたバイト数。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト

メトリクス	説明
PutRecord.Latency	指定された期間に測定された PutRecord オペレーションごとにかかった時間。 ディメンション： StreamName 統計: Minimum、Maximum、Average 単位: ミリ秒
PutRecord.Success	指定された期間に測定された、Kinesis ストリームごとの成功した PutRecord オペレーションの数。Average はストリームへの書き込み成功率を反映しています。 ディメンション： StreamName 統計: Average、Sum、Samples 単位: カウント
PutRecords.Bytes	指定された期間に PutRecords オペレーションを使用して Kinesis ストリームに送信されたバイト数。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト
PutRecords.Latency	指定された期間に測定された PutRecords オペレーションごとにかかった時間。 ディメンション： StreamName 統計: Minimum、Maximum、Average 単位: ミリ秒

メトリクス	説明
PutRecords.Records	このメトリクスは廃止されました。PutRecords.SuccessfulRecords を使用します。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
PutRecords.Success	指定された期間に測定された、Kinesis ストリームあたりの最低 1 つのレコードが成功した PutRecords オペレーションの数。 ディメンション： StreamName 統計: Average、Sum、Samples 単位: カウント
PutRecords.TotalRecords	指定された期間に測定された、Kinesis Data Streams ごとに PutRecords オペレーションで送信されたレコードの総数。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
PutRecords.SuccessfulRecords	指定された期間に測定された、Kinesis Data Streams ごとの PutRecords オペレーションの正常なレコード数。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

メトリクス	説明
PutRecords.FailedRecords	指定された期間に測定された、Kinesis Data Streams ごとに PutRecords オペレーションで内部障害のために拒否されたレコードの数。時折内部障害が予想されるため、再試行する必要があります。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
PutRecords.ThrottledRecords	指定された期間に測定された、Kinesis Data Streams ごとに PutRecords オペレーションでスロットリングのために拒否されたレコードの数。 ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

メトリクス	説明
ReadProvisionedThroughputExceeded	指定された期間のストリームで調整された GetRecords 呼び出し回数。このメトリクスで最も一般的に使用される統計は Average です。 Minimum の統計の値が 1 の場合、指定された期間にストリームについてすべてのレコードが調整されました。 Maximum の統計の値が 0 (ゼロ) の場合、指定された期間にストリームについてどのレコードも調整されていません。 シャードレベルメトリクス名: ReadProvisionedThroughputExceeded ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
SubscribeToShard.RateExceeded	このメトリックスは、同じコンシューマーによるアクティブなサブスクリプションがすでにあるため新しいサブスクリプションが失敗したとき、またはこのオペレーションで許可される 1 秒あたりの呼び出し回数を超えた場合に出力されます。 ディメンション: StreamName、 ConsumerName
SubscribeToShard.Success	このメトリクスは、 SubscribeToShardサブスクリプションが正常に確立されたかどうかを記録します。このサブスクリプションの有効期間は最大で 5 分のみです。したがって、このメトリックスは少なくとも 5 分に 1 回発行されます。 ディメンション： StreamName , ConsumerName

メトリクス	説明
SubscribeToShardEvent.Bytes	指定された期間に測定された、シャードから受信したバイト数。Minimum、Maximum、および Average の統計は、指定した期間内の単一イベントで発行されたバイト数です。 シャードレベルメトリクス名: OutgoingBytes ディメンション： StreamName , ConsumerName 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト
SubscribeToShardEvent.MillisBehindLatest	現在の時刻と、 SubscribeToShard イベントの最後のレコードがストリームに書き込まれた日時の差。 ディメンション： StreamName , ConsumerName 統計: Minimum、Maximum、Average、Samples 単位: ミリ秒
SubscribeToShardEvent.Records	指定された期間に測定された、シャードから受信したレコード数。Minimum、Maximum、および Average の統計は、指定した期間内の単一イベント内のレコードです。 シャードレベルメトリクス名: OutgoingRecords ディメンション： StreamName , ConsumerName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

メトリクス	説明
SubscribeToShardEvent.Success	このメトリックスは、イベントが正常に発行されるたびに出力されます。これは、アクティブなサブスクリプションがある場合にのみ出力されます。 ディメンション： StreamName, ConsumerName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
WriteProvisionedThroughputExceeded	指定された期間にストリームのスロットリングにより拒否されたレコードの数。このメトリクスには、PutRecord および PutRecords オペレーションのスロットリングも含まれます。このメトリクスで最も一般的に使用される統計は Average です。 Minimum の統計がゼロ以外の値の場合、指定された期間にストリームについてレコードが調整中でした。 Maximum の統計の値が 0 (ゼロ) の場合、指定された期間のストリームで、どのレコードも調整中ではありませんでした。 シャードレベルメトリクス名: WriteProvisionedThroughputExceeded ディメンション： StreamName 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

拡張シャードレベルメトリクス

AWS/Kinesis 名前空間には、次のシャードレベルメトリクスが含まれます。

Kinesis は、次のシャードレベルのメトリクスを 1 分 CloudWatch ごとに に送信します。各メトリクスディメンションは 1 CloudWatch メトリクスを作成し、1 か月あたり約 43,200 件の

PutMetricData API コールを実行します。デフォルトでは、これらのメトリクスは有効ではありません。Kinesis から発生した拡張メトリクスには、料金がかかります。詳細については、「Amazon Custom Metrics」の見出しの「Amazon の CloudWatch 料金」を参照してください。 CloudWatch 料金は、1 ヶ月あたりのシャードごとに表示されます。

メトリクス	説明
IncomingBytes	指定された期間に、シャードに正常に送信されたバイト数。このメトリクスには、PutRecord および PutRecords オペレーションのバイト数も含まれます。Minimum、Maximum、および Average の統計は、指定した期間内のシャードの単一 put オペレーションでのバイト数です。 ストリームレベルメトリクス名: IncomingBytes ディメンション： StreamName , ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト
IncomingRecords	指定された期間に、シャードに正常に送信されたレコードの数。このメトリクスには、PutRecord および PutRecords オペレーションのレコード数も含まれます。Minimum、Maximum、および Average の統計は、指定した期間内のシャードの単一 put オペレーションでのレコード数です。 ストリームレベルメトリクス名: IncomingRecords ディメンション： StreamName , ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
IteratorAgeMilliseconds	シャードに対して行われたすべての GetRecords 呼び出しの最後のレコードの期間 (指定された時間に測定)。期間は、現在の時刻と、GetRecords 呼び出し

メトリクス	説明
	の最後のレコードがストリームに書き込まれた時刻の差です。Minimum および Maximum 統計は、Kinesis コンシューマーアプリケーションのプロセスを追跡するのに使用できます。値が 0 (ゼロ) の場合は、読み取り中のレコードがストリームに完全に追いついていることを示します。 ストリームレベルメトリクス名: `GetRecords.IteratorAgeMilliseconds` ディメンション： StreamName ， ShardId 統計: Minimum、Maximum、Average、Samples 単位: ミリ秒
`OutgoingBytes`	指定された期間に測定された、シャードから取得したバイト数。Minimum、Maximum、および Average の統計は、指定した期間内のシャードの単一 `GetRecords` オペレーションで返されたバイト数または単一の `SubscribeToShard` イベントで発行されたバイト数です。 ストリームレベルメトリクス名: `GetRecords.Bytes` ディメンション： StreamName ， ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: バイト

メトリクス	説明
OutgoingRecords	指定された期間に測定された、シャードから取得したレコード数。Minimum、Maximum、および Average の統計は、指定した期間内のシャードの単一 GetRecords オペレーションで返されたレコードまたは単一の SubscribeToShard イベントで発行されたレコードです。 ストリームレベルメトリクス名: GetRecords.Records ディメンション： StreamName , ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント
ReadProvisionedThroughputExceeded	指定された期間のシャードで調整された GetRecords 呼び出し回数。この例外カウントは、1 秒あたり 1 つのシャードあたり 5 回の読み込みまたは 1 つのシャードあたり 1 秒あたり 2 MB の制限のすべてのディメンションを含みます。このメトリクスで最も一般的に使用される統計は Average です。 Minimum の統計の値が 1 の場合、指定された期間にシャードについてすべてのレコードが調整されました。 Maximum の統計の値が 0 (ゼロ) の場合、指定された期間にシャードについてどのレコードも調整されていません。 ストリームレベルメトリクス名: ReadProvisionedThroughputExceeded ディメンション： StreamName , ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

メトリクス	説明
WriteProvisionedThroughputExceeded	指定された期間にシャードのスロットリングにより拒否されたレコードの数。このメトリクスには、PutRecord および PutRecords オペレーションのスロットリングが含まれ、さらに、1 つのシャードあたり 1 秒あたり 1,000 レコードまたは 1 つのシャードあたり 1 秒あたり 1 MB の制限のすべてのディメンションが含まれます。このメトリクスで最も一般的に使用される統計は Average です。 Minimum の統計がゼロ以外の値の場合、指定された期間にシャードについてレコードが調整中でした。 Maximum の統計の値が 0 (ゼロ) の場合、指定された期間のシャードで、どのレコードも調整中ではありませんでした。 ストリームレベルメトリクス名: WriteProvisionedThroughputExceeded ディメンション： StreamName , ShardId 統計: Minimum、Maximum、Average、Sum、Samples 単位: カウント

Amazon Kinesis Data Streams メトリクスのディメンション

ディメンション	説明
StreamName	Kinesis ストリームの名前。使用可能なすべての統計が StreamName によってフィルタリングされます。

推奨される Amazon Kinesis Data Streams メトリクス

Amazon Kinesis Data Streams メトリクスのいくつかは、Kinesis Data Streams のお客様に特に重要です。次のリストは推奨メトリクスとご利用方法を提供しています。

メトリクス	使用に関する注意事項
GetRecords.IteratorAgeMilliseconds	ストリームのすべてのシャードとコンシューマーの読み込み場所を追跡します。イテレータの経過日数が保持期間 (デフォルトで 24 時間、最大で 7 日まで設定可能) の 50% を経過すると、レコードの有効期限切れによるデータ損失のリスクがあります。この損失がリスクになる前に、Maximum 統計の CloudWatch アラームを使用して警告することをお勧めします。このメトリクスを使用するシナリオ例は、コンシューマーレコードの処理が遅れるを参照してください。
ReadProvisionedThroughputExceeded	コンシューマー側のレコード処理に後れが生じているときに、ボトルネックがどこにあるかを確認するのは難しい場合があります。このメトリクスを使用して、読み取りスループット制限を超えたために、読み取りが調整されているかを判断してください。このメトリクスで最も一般的に使用される統計は Average です。
WriteProvisionedThroughputExceeded	これは、ReadProvisionedThroughputExceeded メトリクスと同じ目的ですが、ストリームのプロデューサー (PUT) 側用です。このメトリクスで最も一般的に使用される統計は Average です。
PutRecord.Success , PutRecords.Success	Average 統計の CloudWatch アラームを使用して、レコードがストリームに失敗するタイミングを示すことをお勧めします。プロデューサーが使用しているものに応じて、一つまたは両方の種類の PUT 種類を選択します。Kinesis Producer Library (KPL) を使用する場合は、PutRecords.Success を使用します。
GetRecords.Success	Average 統計の CloudWatch アラームを使用して、レコードがストリームから失敗するタイミングを示すことをお勧めします。

Kinesis Data Streams の Amazon CloudWatch メトリクスへのアクセス

CloudWatch コンソール、コマンドライン、または CloudWatch API を使用して、Kinesis Data Streams のメトリクスをモニタリングできます。次の手順は、これらのさまざまなメソッドを使用してメトリクスにアクセスする方法を示しています。

CloudWatch コンソールを使用してメトリクスにアクセスするには

1. https://console.aws.amazon.com/cloudwatch/ で CloudWatch コンソールを開きます。
2. ナビゲーションバーで、リージョンを選択します。
3. ナビゲーションペインで メトリクスを選択します。
4. CloudWatch カテゴリ別のメトリクスペインで、Kinesis Metrics を選択します。
5. 関連する行をクリックして、指定された MetricNameおよび の統計を表示しますStreamName。

   注： ほとんどのコンソール統計名は、読み取りスループット と書き込みスループット を除き、上記の対応する CloudWatch メトリクス名と一致します。 これらの統計は 5 分間隔で計算されます。書き込みスループットはメトリクスをモニタリングし`IncomingBytes` CloudWatch、読み取りスループットは をモニタリングします`GetRecords.Bytes`。
6. (オプション) グラフペインで統計と期間を選択し、これらの設定を使用して CloudWatch アラームを作成します。

を使用してメトリクスにアクセスするには AWS CLI

list-metrics と get-metric-statistics コマンドを使用します。

CloudWatch CLI を使用してメトリクスにアクセスするには

mon-list-metrics および mon-get-stats コマンドを使用します。

CloudWatch API を使用してメトリクスにアクセスするには

ListMetrics および GetMetricStatisticsオペレーションを使用します。

Amazon による Kinesis Data Streams エージェントの状態のモニタリング CloudWatch

エージェントは、 の名前空間を持つカスタム CloudWatch メトリクスを発行しますAWS KinesisAgent。これらのメトリクスを使用して、エージェントがデータを指定されたとおりに

Kinesis Data Streams にデータを送信しており、エージェントが正常であり、データプロデューサーで適切な量の CPU とメモリリソースを消費しているかを評価できます。送信されたレコード数やバイト数などのメトリクスは、エージェントがストリームにデータを送信する速度を知るのに便利です。これらのメトリクスが、ある程度の割合低下するかゼロになることで期待されるしきい値を下回っている場合は、設定の問題、ネットワークエラー、エージェントの状態の問題を示している場合があります。オンホスト CPU やメモリなどの消費量とエージェントエラーカウンターなどのメトリクスは、プロデューサーのリソース使用率を示し、潜在的な構成またはホストのエラーに対する洞察を提供します。最後に、エージェントの問題を調査するのに役立つサービス例外を記録します。これらのメトリクスは、エージェント構成設定 `cloudwatch.endpoint` で指定されたリージョンで報告されます。複数の Kinesis エージェントから発行された CloudWatch メトリクスは、集約または結合されます。エージェント設定の詳細については、「エージェントの設定」を参照してください。

によるモニタリング CloudWatch

Kinesis Data Streams エージェントは、次のメトリクスを に送信します CloudWatch。

メトリクス	説明
`BytesSent`	指定された期間に Kinesis Data Streams に送信されたバイト数。 単位: バイト
`RecordSendAttempts`	指定した期間内の `PutRecords` 呼び出しのレコード数 (初回または再試行の)。 単位: カウント
`RecordSendErrors`	指定した期間内の、`PutRecords` への呼び出しの失敗ステータス (再試行など) のレコード数。 単位: カウント
`ServiceErrors`	指定した期間内の、サービスエラー (スロットリングエラーを除く) となった `PutRecords` への呼び出し数。 単位: カウント

AWS CloudTrailを使用した Amazon Kinesis Data Streams API コールのログ記録

Amazon Kinesis Data Streams は と統合されています。これは AWS CloudTrail、Kinesis Data Streams のユーザー、ロール、または AWS サービスによって実行されたアクションを記録するサービスです。Kinesis Data Streams のすべての API コールをイベントとして CloudTrail キャプチャします。キャプチャされたコールには、Kinesis Data Streams コンソールからのコールと、Kinesis Data Streams API オペレーションへのコードコールが含まれます。証跡を作成する場合は、Kinesis Data Streams の CloudTrail イベントなど、Amazon S3 バケットへのイベントの継続的な配信を有効にすることができます。 Amazon S3 証跡を設定しない場合でも、 CloudTrail コンソールのイベント履歴 で最新のイベントを表示できます。で収集された情報を使用して CloudTrail、Kinesis Data Streams に対するリクエスト、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

の設定と有効化の方法など CloudTrail、 の詳細については、「 AWS CloudTrail ユーザーガイド」を参照してください。

の Kinesis Data Streams 情報 CloudTrail

CloudTrail AWS アカウントを作成すると、 がアカウントで有効になります。Kinesis Data Streams でサポートされているイベントアクティビティが発生すると、そのアクティビティは CloudTrail イベント履歴 の他の AWS サービスイベントとともにイベントに記録されます。 AWS アカウントで最近のイベントを表示、検索、ダウンロードできます。詳細については、「イベント履歴を使用した CloudTrailイベントの表示」を参照してください。

Kinesis Data Streams のイベントなど、 AWS アカウント内のイベントの継続的な記録については、証跡を作成します。証跡により CloudTrail 、 はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成すると、証跡はすべての AWS リージョンに適用されます。証跡は、 AWS パーティション内のすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、 CloudTrail ログで収集されたイベントデータをより詳細に分析し、それに基づいて行動するように、他の AWS サービスを設定できます。詳細については、次を参照してください:

- 証跡の作成のための概要
- CloudTrail サポートされているサービスと統合
- の Amazon SNS 通知の設定 CloudTrail

- 複数のリージョンからの CloudTrail ログファイルの受信と複数のアカウントからの CloudTrail ログファイルの受信

Kinesis Data Streams では、次のアクションをイベントとして CloudTrail ログファイルに記録できます。

- AddTagsToStream
- CreateStream
- DecreaseStreamRetentionPeriod
- DeleteStream
- DeregisterStreamConsumer
- DescribeStream
- DescribeStreamConsumer
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- GetRecords
- GetShardIterator
- IncreaseStreamRetentionPeriod
- ListStreamConsumers
- ListStreams
- ListTagsForStream
- MergeShards
- PutRecord
- PutRecords
- RegisterStreamConsumer
- RemoveTagsFromStream
- SplitShard
- StartStreamEncryption
- StopStreamEncryption
- SubscribeToShard
- UpdateShardCount

- UpdateStreamMode

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます:

- リクエストが root または AWS Identity and Access Management (IAM) ユーザーの認証情報を使用して行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、CloudTrailuserIdentity Element」を参照してください。

例: Kinesis Data Streams ログファイルエントリ

証跡は、指定した Amazon S3 バケットにイベントをログファイルとして配信できるようにする設定です。 CloudTrail ログファイルには 1 つ以上のログエントリが含まれます。イベントは任意ソースからの単一リクエストを表し、リクエストされたアクション、アクションの日時、リクエストパラメータなどの情報を含みます。 CloudTrail ログファイルはパブリック API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

次の例
は、、CreateStream、、DescribeStream、、ListStreamsDeleteStreamSplitShardおよび MergeShardsアクションを示す CloudTrail ログエントリを示しています。

```
{
    "Records": [
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:31Z",
            "eventSource": "kinesis.amazonaws.com",
```

```
            "eventName": "CreateStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardCount": 1,
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "db6c59f8-c757-11e3-bc3b-57923b443c1c",
            "eventID": "b7acfcd0-6ca9-4ee1-a3d7-c4e8d420d99b"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:06Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DescribeStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f0944d86-c757-11e3-b4ae-25654b1d3136",
            "eventID": "0b2f1396-88af-4561-b16f-398f8eaea596"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
```

```
            },
            "eventTime": "2014-04-19T00:15:02Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "ListStreams",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "limit": 10
            },
            "responseElements": null,
            "requestID": "a68541ca-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "22a5fb8f-4e61-4bee-a8ad-3b72046b4c4d"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:07Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DeleteStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f10cd97c-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "607e7217-311a-4a08-a904-ec02944596dd"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
```

```
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:15:03Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "SplitShard",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardToSplit": "shardId-000000000000",
                "streamName": "GoodStream",
                "newStartingHashKey": "11111111"
            },
            "responseElements": null,
            "requestID": "a6e6e9cd-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "dcd2126f-c8d2-4186-b32a-192dd48d7e33"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:56Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "MergeShards",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream",
                "adjacentShardToMerge": "shardId-000000000002",
                "shardToMerge": "shardId-000000000001"
            },
            "responseElements": null,
            "requestID": "e9f9c8eb-c757-11e3-bf1d-6948db3cd570",
            "eventID": "77cf0d06-ce90-42da-9576-71986fec411f"
        }
    ]
```

```
}
```

Amazon による Kinesis Client Library のモニタリング CloudWatch

Amazon KinesisKinesis Data Streams の Kinesis Client Library (KCL) は、KCL アプリケーションの名前を名前空間として使用して、ユーザーに代わってカスタム Amazon CloudWatch メトリクスを発行します。これらのメトリクスを表示するには、CloudWatch コンソールに移動し、カスタムメトリクス を選択します。カスタムメトリクスの詳細については、「Amazon ユーザーガイド」の「カスタムメトリクスの発行」を参照してください。 CloudWatch

KCL CloudWatch によって にアップロードされたメトリクスには少額の料金が発生します。具体的には、Amazon CloudWatch Custom Metrics と Amazon CloudWatch API Requests の料金が適用されます。詳細については、「Amazon CloudWatch の料金」を参照してください。

トピック

- メトリクスと名前空間
- メトリクスレベルとディメンション
- メトリクスの設定
- メトリクスの一覧

メトリクスと名前空間

メトリクスのアップロードに使用される名前空間は、KCL の起動時に指定されたアプリケーション名です。

メトリクスレベルとディメンション

にアップロードするメトリクスを制御するには、次の 2 つのオプションがあります CloudWatch。

メトリクスレベル

すべてのメトリクスに、個別のレベルが割り当てられます。メトリクスのレポートレベルを設定すると、個々のレベルがレポートレベルを下回るメトリクスは に送信されません CloudWatch。このレベルとして、NONE、SUMMARY、DETAILED があります。デフォルト設定はですDETAILED。つまり、すべてのメトリクスが に送信されます CloudWatch。レポートレベル NONE は、メトリクスがまったく送信されないことを意味します。各メトリクスに割り当てられるメトリクスの詳細については、メトリクスの一覧を参照してください。

有効なディメンション

すべての KCL メトリクスには、 にも送信されるディメンションが関連付けられています CloudWatch。KCL 2.x では、単一のデータストリームを処理するように KCL が設定されている場合、すべてのメトリクスディメンション (Operation、ShardId、およびWorkerIdentifier) が、デフォルトで有効になっています。また、KCL 2.x では、単一のデータストリームを処理するように KCL が設定されている場合、Operation ディメンションを無効にすることはできません。KCL 2.x では、KCL が複数のデータストリームを処理するように構成されている場合、すべてのメトリクスディメンション (Operation、ShardId、StreamId、およびWorkerIdentifier) は、デフォルトで有効になっています。また、KCL 2.x では、KCL が複数のデータストリームを処理するように設定されている場合、Operation と StreamId ディメンションを無効にすることはできません。StreamIdディメンションは、シャードごとのメトリクスでのみ使用できます。

KCL 1.x では、デフォルトでは、Operation および ShardId ディメンションのみが有効であり、WorkerIdentifier ディメンションは無効となります。KCL 1.x では、Operation ディメンションを無効にすることはできません。

CloudWatch メトリクスディメンションの詳細については、「Amazon CloudWatch ユーザーガイドhttps://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/cloudwatch_concepts.html#Dimension」の「Amazon CloudWatch 概念」トピックの「ディメンション」セクションを参照してください。

WorkerIdentifier ディメンションが有効になっている場合、特定の KCL ワーカーが再起動するたびにワーカー ID プロパティに別の値が使用されると、新しいWorkerIdentifierディメンション値を持つ新しいメトリクスセットが に送信されます CloudWatch。特定の KCL ワーカーの再起動で、WorkerIdentifier ディメンションの値が同じである必要がある場合、各ワーカーの初期化中に同じワーカー ID 値を明示的に指定する必要があります。アクティブな各 KCL ワーカーのワーカー ID 値は、すべての KCL ワーカー間で一意である必要があります。

メトリクスの設定

メトリクスレベルと有効なディメンションは、KCL アプリケーションの起動時にワーカーに渡される KinesisClientLibConfiguration インスタンスを使用して設定できます。 MultiLangDaemon この場合、 metricsLevel および metricsEnabledDimensionsプロパティは、 MultiLangDaemon KCL アプリケーションの起動に使用される .properties ファイルで指定できます。

メトリクスレベルには、NONE、SUMMARY、または DETAILED の 3 つの値のうち 1 つを割り当てることができます。有効なディメンション値は、 CloudWatch メトリクスに使用できるディメンショ

ンのリストを含むカンマ区切りの文字列である必要があります。KCL アプリケーションによって使用されるディメンションは、`Operation`、`ShardId`、および `WorkerIdentifier` です。

メトリクスの一覧

次の表には、範囲およびオペレーションによってグループ分けされた KCL メトリクスが一覧表示されています。

トピック

KCL アプリケーションあたりのメトリクス

これらのメトリクスは、Amazon CloudWatch 名前空間で定義されているように、アプリケーションのスコープ内のすべての KCL ワーカーにわたって集計されます。

トピック

InitializeTask

`InitializeTask` オペレーションは、KCL アプリケーションのレコードプロセッサを初期化します。このオペレーションのロジックには、Kinesis Data Streams からのシャードイテレーターの取得とレコードプロセッサの初期化が含まれています。

メトリクス	説明
KinesisDataFetcher.getIterator.Success	KCL アプリケーションあたりの `GetShardIterator` オペレーションの成功回数。

メトリクス	説明
	メトリクスレベル: Detailed 単位: カウント
KinesisDataFetcher.getIterator.Time	指定された KCL アプリケーションの `GetShardIterator` オペレーションあたりの所要時間。 メトリクスレベル: Detailed 単位: ミリ秒
RecordProcessor.initialize.Time	レコードプロセッサの初期化メソッドにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒
成功	レコードプロセッサの初期化の成功回数。 メトリクスレベル: Summary 単位: カウント
時間	KCL ワーカーがレコードプロセッサの初期化にかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

ShutdownTask

`ShutdownTask` オペレーションは、シャード処理のシャットダウンシーケンスを開始します。これは、シャードが分割または結合された場合やシャードリースがワーカーから失われた場合に発生する場合があります。どちらの場合も、レコードプロセッサ `shutdown()` 関数が呼び出されます。また、シャードが分割または結合された場合、新しいシャードが 1 つまたは 2 つ作成されるため、新しいシャードが検出されます。

メトリクス	説明
CreateLease.成功	親シャードのシャットダウンの後に、新しい子シャードが KCL アプリケーションの DynamoDB テーブルに正常に追加された回数。 メトリクスレベル: Detailed 単位: カウント
CreateLease.Time	KCL アプリケーションの DynamoDB テーブルに新しい子シャード情報を追加する所要時間。 メトリクスレベル: Detailed 単位: ミリ秒
UpdateLease.成功	レコードプロセッサのシャットダウン中に成功した最終チェックポイントの数。 メトリクスレベル: Detailed 単位: カウント
UpdateLease.Time	レコードプロセッサのシャットダウン中にチェックポイントオペレーションにかかった時間。 メトリクスレベル: Detailed 単位: ミリ秒
RecordProcessor.shutdown.Time	レコードプロセッサのシャットダウンメソッドにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒
成功	シャットダウンタスクの成功回数。 メトリクスレベル: Summary 単位: カウント

メトリクス	説明
時間	KCL ワーカーがシャットダウンタスクにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

ShardSyncTask

`ShardSyncTask` オペレーションは、Kinesis Data Streams のシャード情報に対する変更を検出するため、KCL アプリケーションで新しいシャードを処理できます。

メトリクス	説明
CreateLease.成功	KCL アプリケーションの DynamoDB テーブルへの新しいシャード情報の追加が成功した回数。 メトリクスレベル: Detailed 単位: カウント
CreateLease.Time	KCL アプリケーションの DynamoDB テーブルに新しいシャード情報を追加する所要時間。 メトリクスレベル: Detailed 単位: ミリ秒
成功	シャード同期オペレーションの成功回数。 メトリクスレベル: Summary 単位: カウント
時間	シャード同期オペレーションにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

BlockOnParentTask

シャードが分割または他のシャードと結合された場合、新しい子シャードが作成されます。`BlockOnParentTask` オペレーションは、KCL による親シャードの処理が完了するまで、新しいシャードのレコード処理が開始されないようにします。

メトリクス	説明
成功	親シャードの完了チェックの成功回数。 メトリクスレベル: Summary 単位: カウント
時間	親シャードが完了するまでにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

PeriodicShardSyncManager

`PeriodicShardSyncManager` は、KCL コンシューマーアプリケーションによって処理されているデータストリームを調べ、部分リースを持つデータストリームを特定し、同期のためにそれらを引き渡します。

次のメトリクスは、KCL が単一のデータストリームを処理するように設定されている場合 (NumStreamsToSync と の値が 1 NumStreamsWithPartialLeases に設定されている場合)、および KCL が複数のデータストリームを処理するように設定されている場合に使用できます。

メトリクス	説明
NumStreamsToSync	部分リースを含み、同期のために引き渡す必要があるコンシューマーアプリケーションによって処理されるデータストリームの数 (AWS アカウントあたり)。 メトリクスレベル: Summary 単位: カウント

メトリクス	説明
NumStreamsWithPartialLeases	コンシューマーアプリケーションが処理している部分リースを含むデータストリームの数 (AWS アカウントあたり) 。 メトリクスレベル: Summary 単位: カウント
成功	回数 `PeriodicShardSyncManager` は、コンシューマーアプリケーションが処理しているデータストリーム内の部分リースを正常に識別できました。 メトリクスレベル: Summary 単位: カウント
時間	その時間の量 (ミリ秒) では、シャード同期が必要なデータストリームを特定するために、コンシューマーアプリケーションが処理しているデータストリームを調べます。 メトリクスレベル: Summary 単位: ミリ秒

MultistreamTracker

`MultistreamTracker` インターフェイスを使用すると、複数のデータストリームを同時に処理できる KCL コンシューマーアプリケーションを構築できます。

メトリクス	説明
DeletedStreams.Count	この期間に削除されたデータストリームの数。 メトリクスレベル: Summary 単位: カウント
ActiveStreams.Count	処理されているアクティブなデータストリームの数。

メトリクス	説明
	メトリクスレベル: Summary 単位: カウント
StreamsPendingDeletion.Count	`FormerStreamsLeasesDeletionStrategy` に基づいて削除が保留中のデータストリームの数。 メトリクスレベル: Summary 単位: カウント

ワーカーあたりのメトリクス

これらのメトリクスは、Amazon EC2 インスタンスなど、Kinesis Data Streams のデータを消費するすべてのレコードプロセッサにわたって集約されます。

トピック

- RenewAllLeases
- TakeLeases

RenewAllLeases

`RenewAllLeases` オペレーションは、特定のワーカーインスタンスによって所有されるシャードリースを定期的に更新します。

メトリクス	説明
RenewLease.成功	ワーカーによるリース更新の成功回数。 メトリクスレベル: Detailed 単位: カウント
RenewLease.Time	リース更新オペレーションにかかった時間。 メトリクスレベル: Detailed

メトリクス	説明
	単位: ミリ秒
CurrentLeases	すべてのリースの更新後にワーカーによって所有されているシャードリースの数。 メトリクスレベル: Summary 単位: カウント
LostLeases	ワーカーによって所有されているすべてのリースの更新を試みたときに失われたシャードリースの数。 メトリクスレベル: Summary 単位: カウント
成功	ワーカーのリース更新オペレーションが成功した回数。 メトリクスレベル: Summary 単位: カウント
時間	ワーカーのすべてのリースを更新するのにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

TakeLeases

TakeLeases オペレーションは、すべての KCL ワーカー間でレコード処理の負荷を分散させます。現在のKCL ワーカーのシャードリースが、必要数を下回る場合、過負荷になっている他のワーカーからシャードリースを取得します。

メトリクス	説明
ListLeases.成功	すべてのシャードリースがKCL アプリケーションのDynamoDB テーブルから正常に取得された回数。

メトリクス	説明
	メトリクスレベル: Detailed 単位: カウント
ListLeases.Time	KCL アプリケーションの DynamoDB テーブルからすべてのシャードリースを取得する所要時間。 メトリクスレベル: Detailed 単位: ミリ秒
TakeLease.成功	ワーカーが他のKCL ワーカーからシャードリースを正常に取得した回数。 メトリクスレベル: Detailed 単位: カウント
TakeLease.Time	ワーカーが取得したリースを使用してリーステーブルを更新するのにかかった時間。 メトリクスレベル: Detailed 単位: ミリ秒
NumWorkers	特定のワーカーにより識別されるワーカーの総数。 メトリクスレベル: Summary 単位: カウント
NeededLeases	現在のワーカーがシャード処理の負荷を分散するのに必要なシャードリースの数。 メトリクスレベル: Detailed 単位: カウント

メトリクス	説明
LeasesToTake	ワーカーが取得を試みるリースの数。 メトリクスレベル: Detailed 単位: カウント
TakenLeases	ワーカーが取得に成功したリースの数。 メトリクスレベル: Summary 単位: カウント
TotalLeases	CL アプリケーションが処理しているシャードの総数。 メトリクスレベル: Detailed 単位: カウント
ExpiredLeases	特定のワーカーによって識別されるどのワーカーでも処理されていないシャードの総数。 メトリクスレベル: Summary 単位: カウント
成功	TakeLeases　オペレーションが正常に完了した回数。 メトリクスレベル: Summary 単位: カウント
時間	ワーカーの TakeLeases　オペレーションにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

シャードあたりのメトリクス

これらのメトリクスは、単一のレコードプロセッサについて集約されます。

ProcessTask

ProcessTask オペレーションは、現在のイテレーター位置GetRecordsで を呼び出してストリームからレコードを取得し、レコードプロセッサprocessRecords関数を呼び出します。

メトリクス	説明
KinesisDataFetcher.getRecords.Success	Kinesis data stream シャードあたりの GetRecords オペレーションの成功回数。 メトリクスレベル: Detailed 単位: カウント
KinesisDataFetcher.getRecords.Time	Kinesis data stream シャードの GetRecords オペレーションあたりの所要時間。 メトリクスレベル: Detailed 単位: ミリ秒
UpdateLease.成功	指定されたシャードのレコードプロセッサによってチェックポイントが正常に作成された回数。 メトリクスレベル: Detailed 単位: カウント
UpdateLease.Time	指定されたシャードの各チェックポイントオペレーションにかかった時間。 メトリクスレベル: Detailed 単位: ミリ秒
DataBytesProcessed	ProcessTask の各呼び出しで処理されたレコードのバイト単位の合計サイズ。 メトリクスレベル: Summary 単位: バイト

メトリクス	説明
RecordsProcessed	`ProcessTask` の各呼び出しで処理されたレコード数。 メトリクスレベル: Summary 単位: カウント
ExpiredIterator	を呼び出すときに ExpiredIteratorException 受信した の数`GetRecords` 。 メトリクスレベル: Summary 単位: カウント
MillisBehindLatest	現在のイテレーターがシャード内の最新のレコード (先端) から遅れている時間。この値は、応答の最新レコードと現在時間における時間差と同じかそれ以下です。これは、最新の応答レコードのタイムスタンプを比較するよりも、シャードが先端からどれくらい離れているかを示すより正確な反映です。この値は、各レコードの全タイムスタンプの平均ではなく、レコードの最新バッチに適用されます。 メトリクスレベル: Summary 単位: ミリ秒
RecordProcessor.processRecords.Time	レコードプロセッサの `processRecords` メソッドにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒
成功	プロセスタスクオペレーションの成功回数。 メトリクスレベル: Summary 単位: カウント

メトリクス	説明
時間	プロセスタスクオペレーションにかかった時間。 メトリクスレベル: Summary 単位: ミリ秒

Amazon による Kinesis プロデューサーライブラリのモニタリング CloudWatch

Amazon Kinesis Data Streams の Kinesis プロデューサーライブラリ (KPL) は、ユーザーに代わってカスタム Amazon CloudWatch メトリクスを発行します。 Amazon Kinesis これらのメトリクスを表示するには、CloudWatch コンソールに移動し、カスタムメトリクス を選択します。カスタムメトリクスの詳細については、「Amazon ユーザーガイド」の「カスタムメトリクスの発行」を参照してください。 CloudWatch

KPL CloudWatch によって にアップロードされたメトリクスには少額の料金が発生します。具体的には、Amazon CloudWatch Custom Metrics と Amazon CloudWatch API Requests の料金が適用されます。詳細については、「Amazon CloudWatch の料金」を参照してください。ローカルメトリクスの収集には料金は発生 CloudWatchしません。

トピック

- メトリクス、ディメンション、および名前空間
- メトリクスレベルと詳細度
- ローカルアクセスと Amazon CloudWatch アップロード
- メトリクスの一覧

メトリクス、ディメンション、および名前空間

KPL の起動時にアプリケーション名を指定できます。これは、メトリクスをアップロードする際、名前空間の一部として使用されます。これはオプションであり、アプリケーション名を設定しない場合は、KPL によりデフォルト値が設定されます。

また、メトリクスに任意の追加ディメンションを追加するように KPL を設定できます。これは、CloudWatch メトリクスにきめ細かなデータが必要な場合に便利です。たとえば、ディメンションと

してホスト名を追加でき、これによりフリート全体の均一でない負荷分散を特定できます。すべての KPL 構成設定は変更不可能なため、KPL インスタンスを初期化した後、これらの追加ディメンションを変更することはできません。

メトリクスレベルと詳細度

にアップロードされたメトリクスの数を制御するには、次の 2 つのオプションがあります CloudWatch。

メトリクスレベル

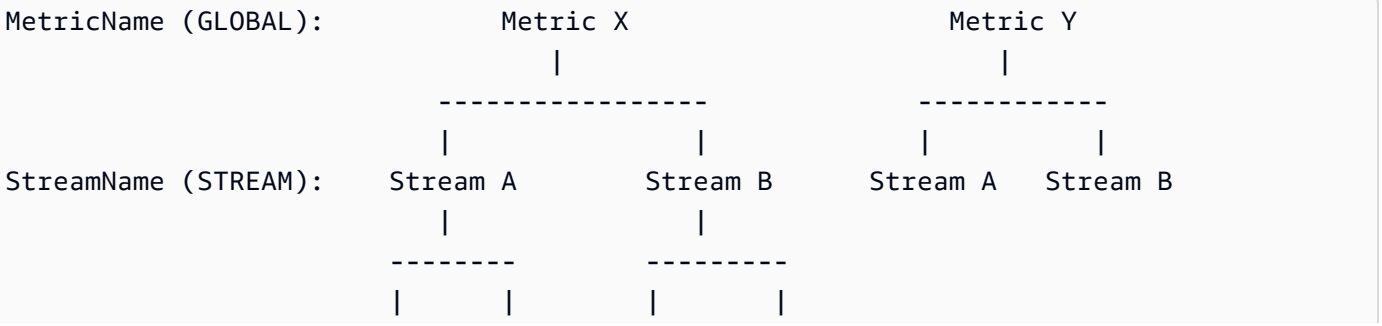
これは、メトリクスの重要性を示すおおよその目安です。すべてのメトリクスにレベルが割り当てられます。レベルを設定すると、レベル 以下のメトリクスは に送信されません CloudWatch。このレベルとして、NONE、SUMMARY、DETAILED があります。デフォルト設定は DETAILED であり、すべてのメトリクスが対象です。NONEは、メトリクスが一切ないことを意味し、どのメトリクスもそのレベルに割り当てられません。

詳細度

これは、追加の詳細度レベルで同じメトリクスが出力されるかどうかを制御します。このレベルとして、GLOBAL、STREAM、SHARD があります。デフォルト設定は SHARD で、最も詳細なメトリクスが含まれます。

SHARD が選択されると、ストリーム名とシャード ID をディメンションとしてメトリクスが出力されます。また、同じメトリクスは、ストリーム名のディメンションのみを使用して出力されるため、そのメトリクスにはストリーム名がありません。つまり、特定のメトリクスについて、それぞれ 2 つのシャードを持つ 2 つのストリームが 7 つの CloudWatch メトリクスを生成します。1 つはシャードごとに、1 つはストリームごとに、もう 1 つは全体に対して生成されます。すべて同じ統計を記述しますが、粒度は異なります。次の図は、これを説明するものです。

異なる詳細度から階層が形成され、システム内のすべてのメトリクスから、メトリクス名をルートとするツリーが構成されます。



```
ShardID (SHARD):     Shard 0 Shard 1  Shard 0 Shard 1
```

すべてのメトリクスをシャードレベルで使用できるわけではありません。一部のメトリクスはストリームレベルまたは本質的にグローバルです。これらは、シャードレベルのメトリクスを有効にしても、シャードレベルで生成されません (前の図の `Metric Y`)。

追加のディメンションを指定すると、`tuple:<DimensionName, DimensionValue, Granularity>` に値を指定する必要があります。詳細度は、カスタムディメンションが階層のどこに挿入されたかを判断するのに使用されます。`GLOBAL`は、追加のディメンションがメトリクス名の後に挿入されたことを意味し、`STREAM` はストリーム名の後に、`SHARD` は ID シャードの後に挿入されたことをそれぞれ意味します。複数の追加ディメンションが詳細度レベルごとに指定された場合、それらは指定された順序で挿入されます。

ローカルアクセスと Amazon CloudWatch アップロード

現在の KPL インスタンスのメトリクスはローカルでリアルタイムに使用できるため、いつでも KPL にクエリを実行してメトリクスを取得できます。KPL は、 のように、すべてのメトリクスの合計、平均、最小、最大、およびカウントをローカルで計算します CloudWatch。

プログラムの開始から現在の時点までの累積として、または過去 N 秒間 (N は 1 から 60 までの整数) のローリングウィンドウを使用して統計情報を取得できます。

すべてのメトリクスを にアップロードできます CloudWatch。これは、複数のホスト、モニタリング、およびアラームの間でデータを集約するのに特に役立ちます。この機能は、ローカルでは使用できません。

前に説明したように、メトリクスレベルと詳細度の設定を使用してどのメトリクスをアップロードするかを選択できます。ローカルでメトリクスをアップロードしたり使用したりすることはできません。

データポイントを個別にアップロードするのは、高トラフィックの場合、毎秒数百万のアップロードが発生するためお勧めしません。このため、KPL はメトリクスをローカルで 1 分のバケットに集約し、有効なメトリクスごとに統計オブジェクトを 1 分あたり CloudWatch 1 回にアップロードします。

メトリクスの一覧

メトリクス	説明
UserRecordsReceived	入力オペレーションで KPL コアにより受信された論理ユーザーレコードの数。シャードレベルでは使用できません。 メトリクスレベル: Detailed 単位: 個
UserRecordsPending	現在保留状態にあるユーザーレコード数の定期的なサンプリング。レコードが現在バッファ処理されていて送信待ちの場合、または、送信済みでバックエンドサービスで処理中の場合、そのレコードは保留状態です。シャードレベルでは使用できません。 KPL が提供する専用のメソッドを使用して、グローバルレベルでこのメトリクスを取得することで、お客様は PUT レートを管理できます。 メトリクスレベル: Detailed 単位: 個
UserRecordsPut	入力に成功した論理ユーザーレコードの数。 このメトリクスの場合、KPL では、失敗したレコードがカウントされません。このため、平均が成功率に、個数が総試行回数に、個数と合計の差が失敗件数にそれぞれ対応します。 メトリクスレベル: Summary 単位: 個
UserRecordsDataPut	入力に成功した論理ユーザーレコードのバイト数。 メトリクスレベル: Detailed 単位: バイト

メトリクス	説明
KinesisRecordsPut	入力に成功した Kinesis Data Streams レコードの数 (各 Kinesis Data Streams レコードには複数のユーザーレコードを含めることができます)。 KPL は、失敗したレコードに対してゼロを出力します。このため、平均が成功率に、個数が総試行回数に、個数と合計の差が失敗件数にそれぞれ対応します。 メトリクスレベル: Summary 単位: 個
KinesisRecordsDataPut	Kinesis Data Streams レコードのバイト数。 メトリクスレベル: Detailed 単位: バイト
ErrorsByCode	各種類のエラーコードの数。これにより、ErrorCode や StreamName などの通常のディメンションに加え、ディメンション ShardId が追加されます。シャードに対して、すべてのエラーを追跡することはできません。追跡できないエラーは、ストリームレベルまたはグローバルレベルでのみ出力されます。このメトリクスは、スロットリング、シャードマッピングの変更、内部エラー、サービス使用不可、タイムアウトなどに関する情報をとらえます。 Kinesis Data Streams API のエラーは、Kinesis Data Streams レコードごとに 1 回カウントされます。Kinesis Data Streams レコード内の複数のユーザーレコードで複数回のカウントが生じることはありません。 メトリクスレベル: Summary 単位: 個

メトリクス	説明
AllErrors	これは、コード別のエラーと同じエラーによってトリガーされますが、エラーの種類は区別されません。異なる種類のすべてのエラーから件数の合計を手計算する必要がなくなるため、これはエラー率の総合的なモニタリングに役立ちます。 メトリクスレベル: Summary 単位: 個
RetriesPerRecord	ユーザーレコードあたりの再試行の実行回数。1 回の試行でレコードが成功した場合は、ゼロが出力されます。 ユーザーレコードが終了すると (成功した場合またはそれ以上再試行されない場合)、直ちにデータが出力されます。レコード time-to-live の値が大きい場合、このメトリクスは大幅に遅れる可能性があります。 メトリクスレベル: Detailed 単位: 個
BufferingTime	ユーザーレコードが KPL に到着してからバックエンドに送信されるまでの時間。この情報は、レコード単位でユーザーに返されますが、集約された統計情報としても使用できます。 メトリクスレベル: Summary 単位: ミリ秒
Request Time	PutRecordsRequests の実行にかかる時間。 メトリクスレベル: Detailed 単位: ミリ秒

メトリクス	説明
User Records per Kinesis Record	単一の Kinesis Data Streams レコードに集約された論理ユーザーレコードの数。 メトリクスレベル: Detailed 単位: 個
Amazon Kinesis Records per PutRecordsRequest	単一の PutRecordsRequest に集約された Kinesis Data Streams レコードの数。シャードレベルでは使用できません。 メトリクスレベル: Detailed 単位: 個
User Records per PutRecordsRequest	PutRecordsRequest に含まれているユーザーレコードの総数。これは、前の 2 つのメトリクスの積にほぼ一致します。シャードレベルでは使用できません。 メトリクスレベル: Detailed 単位: 個

Amazon Kinesis Data Streams のセキュリティ

のクラウドセキュリティが最優先事項 AWS です。 AWS のお客様は、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャの恩恵を受けることができます。

セキュリティは、 AWS とユーザーの間で共有される責任です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ — クラウドで AWS サービスを実行するインフラストラクチャを保護する責任 AWS は AWS にあります。 AWS また、 は、安全に使用できるサービスも提供します。セキュリティの有効性については、AWS コンプライアンスプログラムの一環として、サードパーティー監査人によるテストと検証が定期的に行われています。Kinesis Data Streams に適用されるコンプライアンスプログラムについては、コンプライアンスプログラムによるAWS 対象範囲内のサービスを参照してください。
- クラウドのセキュリティ — お客様の責任は、使用する AWS サービスによって決まります。お客様は、データの機密性、組織の要件、および適用法令と規制などのその他要因に対する責任も担います。

このドキュメントは、Kinesis Data Streams の使用時に責任共有モデルがどのように適用されるかを理解するために役立ちます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するように Kinesis Data Streams を設定する方法を説明します。また、Kinesis Data Streams リソースのモニタリングや保護に役立つ他の AWS のサービスの使用方法についても説明します。

トピック

Amazon Kinesis Data Streams のデータ保護

AWS Key Management Service （ AWS KMS) キーを使用したサーバー側の暗号化により、Amazon Kinesis Data Streams 内の保管中のデータを暗号化することで、厳格なデータ管理要件を簡単に満たすことができます。

> **Note**
>
> コマンドラインインターフェイスまたは API AWS を介して にアクセスするときに FIPS 140-2 検証済みの暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、連邦情報処理規格 (FIPS) 140-2を参照してください。

トピック

Kinesis Data Streams 用のサーバー側の暗号化とは?

サーバー側の暗号化は、Amazon Kinesis Data Streams の機能で、指定した AWS KMS カスタマーマスターキー (CMK) を使用して、保管中のデータを自動的に暗号化します。データは、Kinesis ストリームストレージレイヤーに書き込まれる前に暗号化され、ストレージから取得された後で復号されます。その結果、Kinesis Data Streams サービス内で保管中のデータは暗号化されます。これにより、厳格な規制要件を満たし、データのセキュリティを強化することが可能になります。

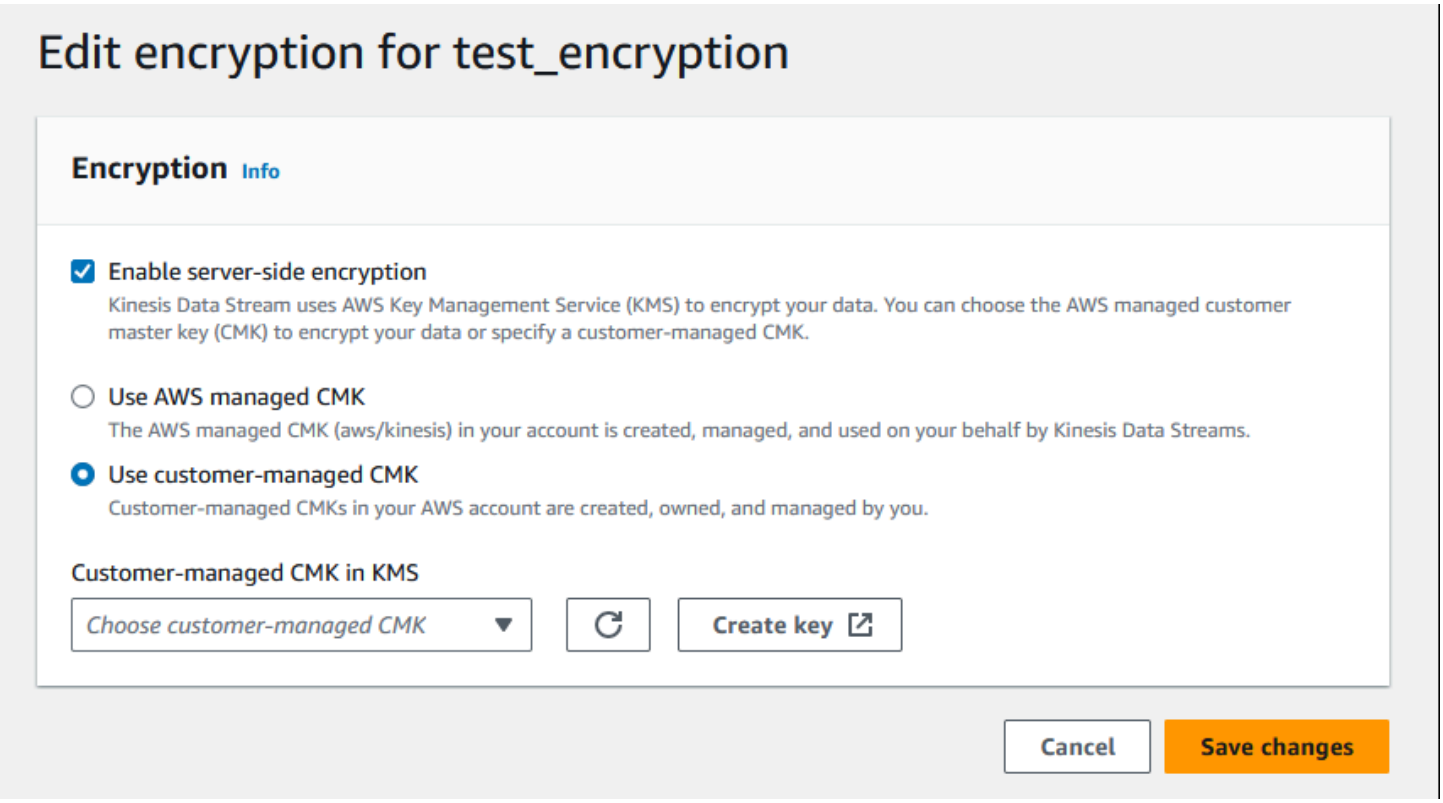
サーバー側の暗号化を使用すると、Kinesis ストリームプロデューサーとコンシューマーがマスターキーや暗号化オペレーションを管理する必要はありません。データは Kinesis Data Streams サービスに出入りするときに自動的に暗号化されるため、保管中のデータは暗号化されます。 AWS KMS は、サーバー側の暗号化機能で使用されるすべてのマスターキーを提供します。 AWS KMS は

AWS、 によって管理される Kinesis 用の CMK、ユーザーが指定した AWS KMS CMK、または AWS KMS サービスにインポートされたマスターキーを簡単に使用できます。

> **Note**
>
> サーバー側の暗号化では、暗号化が有効になって初めて受信データが暗号化されます。暗号化されていないストリーム内に既に存在するデータは、サーバー側の暗号化が有効になった後も暗号化されません。

データストリームを暗号化し、他のプリンシパル (複数可) へのアクセスを共有する場合は、 AWS KMS キーのキーポリシーと外部アカウントの IAM ポリシーの両方でアクセス許可を付与する必要があります。詳細については、「その他のアカウントのユーザーに KMS キーの使用を許可する」を参照してください。

AWS マネージド KMS キーでデータストリームのサーバー側の暗号化を有効にしていて、リソースポリシーを介してアクセスを共有する場合は、次に示すように、カスタマーマネージドキー (CMK) の使用に切り替える必要があります。



さらに、KMS のクロスアカウント共有機能を使用して、共有プリンシパルエンティティが CMK にアクセスできるようにする必要があります。共有プリンシパルエンティティの IAM ポリシーも必ず

変更してください。詳細については、「その他のアカウントのユーザーに KMS キーの使用を許可する」を参照してください。

コスト、リージョン、およびパフォーマンスに関する考慮事項

サーバー側の暗号化を適用すると、 AWS KMS API の使用とキーコストが適用されます。カスタム KMS マスターキーとは異なり、(Default) aws/kinesis カスタマーマスターキー (CMK) は無料で提供されています。ただし、引き続きユーザーに代わって Amazon Kinesis Data Streams によって発生する API の使用料を支払う必要があります。

API の使用料金は、カスタムキーを含めたすべての CMK に適用されます。Kinesis Data Streams は、データキーを回転させている場合、約 5 分ごとに AWS KMS を呼び出します。30 日間の月では、Kinesis ストリームによって開始される AWS KMS API コールの合計コストが数ドル未満である必要があります。このコストは、データプロデューサーとコンシューマーで使用するユーザー認証情報の数に応じて増加します。各ユーザー認証情報には への一意の API コールが必要なためです AWS KMS。認証に IAM ロールを使用すると、各ロールの継承コールは、一意のユーザー認証情報になります。KMS コストを節約するには、assume role コールによって返されるユーザー認証情報をキャッシュしておくとよい場合があります。

以下は、リソース別の料金の説明です。

キー

- (AWS エイリアス = aws/kinesis) によって管理される Kinesis の CMK は無料です。
- ユーザー生成の KMS キーは、KMS キー料金の対象となります。詳細については、AWS Key Management Service の料金を参照してください。

API の使用料金は、カスタムキーを含めたすべての CMK に適用されます。Kinesis Data Streams は、データキーを回転させている場合、約 5 分ごとに KMS を呼び出します。1 か月 (30 日) では、Kinesis Data Streams によって開始された KMS API コールの合計コストは、数ドル未満になるはずです。各ユーザー認証情報には AWS KMS への一意の API コールが必要なため、このコストは、データプロデューサーとコンシューマーで使用するユーザー認証情報の数に応じて増加します。認証に IAM ロールを使用すると、それぞれ assume-role-callに一意のユーザー認証情報が生成され、KMS コストを節約するために から返されたユーザー認証情報 assume-role-call をキャッシュできます。

KMS API の使用

暗号化されたストリームごとに、TIP から読み取り、リーダーとライター間で単一の IAM アカウント/ユーザーアクセスキーを使用する場合、Kinesis サービスは 5分ごとに約 12 回 AWS KMS を呼び出します。TIP から読み取らないと、 AWS KMS サービスへの呼び出しが増える可能性があります。新しいデータ暗号化キーを生成する API リクエストには、 AWS KMS 使用コストがかかります。詳細については、AWS Key Management Service の料金: 使用量を参照してください。

リージョン別のサーバー側暗号化の利用可能性

現在、Kinesis ストリームのサーバー側の暗号化は、 AWS GovCloud （ 米国西部) や中国リージョンなど、Kinesis Data Streams でサポートされているすべてのリージョンで利用できます。Kinesis Data Streams がサポートされているるリージョンについては、https://docs.aws.amazon.com/general/latest/gr/ak.htmlを参照してください。

パフォーマンスに関する考慮事項

暗号化適用のサービスオーバーヘッドにより、サーバー側の暗号化を適用とすると、`PutRecord`、`PutRecords`、および `GetRecords` の標準的なレイテンシーが増加します (100µs 未満)。

サーバー側の暗号化の使用開始方法

サーバー側の暗号化を開始する最も簡単な方法は、 AWS Management Console と Amazon Kinesis KMS サービスキー を使用することです`aws/kinesis`。

次の手順では、Kinesis ストリームに対してサーバー側の暗号化を有効にする方法を示します。

Kinesis ストリームに対してサーバー側の暗号化を有効にするには

1. にサインイン AWS Management Console し、Amazon Kinesis Data Streams コンソール を開きます。
2. AWS Management Consoleで Kinesis ストリームを作成または選択します。
3. [詳細] タブを選択します。
4. [サーバー側の暗号化] で [編集] を選択します。
5. ユーザー生成の KMS マスターキーを使用する場合を除き、KMS マスターキーとして [aws/kinesis] (デフォルト) が選択されていることを確認します。これは、Kinesis サービスによって生成される KMS マスターキーです。[有効] を選択してから、[保存] を選択します。

> **Note**
>
> デフォルトの Kinesis サービスマスターキーは無料ですが、Kinesis が AWS KMS サービスに対して行う API コールには KMS 使用コストが適用されます。

6. ストリームはしばらく [保留中] 状態になります。ストリームの状態が暗号化を有効にしてアクティブ状態に戻った後、そのストリームに書き込まれるすべての着信データは、ユーザーが選択した KMS マスターキーを使用して暗号化されます。
7. サーバー側の暗号化を無効にするには、 のサーバー側の暗号化で無効 を選択し AWS Management Console、保存 を選択します。

ユーザー生成の KMS マスターキーの作成と使用

このセクションでは、Amazon Kinesis によって管理されるマスターキーを使用する代わりに、ユーザー独自の KMS マスターキーを作成して使用する方法について説明します。

ユーザー生成の KMS マスターキーの作成

ユーザー独自のマスターキーを作成する手順については、AWS Key Management Service デベロッパーガイドのキーの作成を参照してください。アカウントのキーを作成すると、inesis Data Streams サービスは KMS マスターキーリストにそれらのキーを返します。

ユーザー生成の KMS マスターキーの使用

コンシューマー、プロデューサー、管理者に正しいアクセス許可が適用されたら、自分の AWS アカウントまたは別の AWS アカウントでカスタム KMS マスターキーを使用できます。アカウント内のすべての KMS マスターキーは、 AWS Management Consoleにある [KMS マスターキー] リストに表示されます。

別のアカウントにあるカスタム KMS マスターキーを使用するには、それらのキーを使用するための許可が必要です。 AWS Management Consoleの ARN 入力ボックスで、KMS マスターキーの ARN を指定する必要もあります。

ユーザー生成の KMS マスターキーを使用するための許可

ユーザー生成の KMS マスターキーでサーバー側の暗号化を使用する前に、ストリームの暗号化とストリームレコードの暗号化と復号を許可するように AWS KMS キーポリシーを設定する必要があり

ます。アクセス AWS KMS 許可の例と詳細については、AWS 「KMS API のアクセス許可: アクションとリソースのリファレンス」を参照してください。

> **Note**
>
> 暗号化のためのデフォルトサービスキーの使用では、カスタム IAM 許可の適用は必要ありません。

ユーザー生成 KMS マスターキーを使用する前に、Kinesis ストリームプロデューサーおよびコンシューマー (IAM プリンシパル) が、KMS マスターキーポリシーでユーザーになっていることを確認します。ユーザーになっていない場合は、ストリームに対する読み取りと書き込みが失敗し、最終的にはデータの損失、処理の遅延、またはアプリケーションのハングにつながる可能性があります。IAM ポリシーを使用して KMS キーの許可を管理できます。詳細については、AWS 「KMS での IAM ポリシーの使用」を参照してください。

プロデューサーのアクセス許可の例

Kinesis ストリームプロデューサーには `kms:GenerateDataKey` 許可が必要です。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:GenerateDataKey"
        ],
        "Resource": "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:PutRecord",
            "kinesis:PutRecords"
        ],
        "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

コンシューマーのアクセス許可の例

Kinesis ストリームコンシューマーには kms:Decrypt 許可が必要です。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:Decrypt"
        ],
        "Resource": "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:GetRecords",
            "kinesis:DescribeStream"
        ],
        "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

Amazon Managed Service for Apache Flink および AWS Lambda は、ロールを使用して Kinesis ストリームを消費します。これらのコンシューマーが使用するロールには、kms:Decrypt 許可を追加するようにしてください。

ストリーム管理者の許可

Kinesis ストリーム管理者には、kms:List* と kms:DescribeKey* を呼び出すための権限が必要です。

KMS キー許可の検証とトラブルシューティング

Kinesis ストリームで暗号化を有効にしたら、次の Amazon CloudWatch メトリクスを使用してputRecord、、putRecords、および getRecords呼び出しの成功をモニタリングすることをお勧めします。

- PutRecord.Success

- PutRecords.Success
- GetRecords.Success

詳細については、Amazon Kinesis Data Streams のモニタリングを参照してください。

インターフェイス VPC エンドポイントと Amazon Kinesis Data Streams の使用

インターフェイス VPC エンドポイントを使用して、Amazon VPC と Kinesis Data Streams 間のトラフィックが Amazon ネットワークから離れないように維持できます。インターフェイス VPC エンドポイントには、インターネットゲートウェイ、NAT デバイス、VPN 接続、または AWS Direct Connect 接続は必要ありません。インターフェイス VPC エンドポイントは PrivateLink、Amazon VPC 内のプライベート IP を備えた Elastic Network Interface を使用して AWS のサービス間のプライベート通信を可能にする AWS テクノロジーである を利用 AWS しています。 IPs 詳細については、Amazon Virtual Private Cloud and Interface VPC Endpoints (AWS PrivateLink) 」を参照してください。

トピック

- Kinesis Data Streams のインターフェイス VPC エンドポイントの使用
- Kinesis Data Streams の VPCE エンドポイントへのアクセス制御
- Kinesis Data Streams の VPC エンドポイントポリシーの可用性

Kinesis Data Streams のインターフェイス VPC エンドポイントの使用

使用を開始するために、ストリーム、プロデューサー、またはコンシューマーの設定を変更する必要はありません。Kinesis Data Streams トラフィックが Amazon VPC リソースとの間でやり取りされるようにするには、インターフェイス VPC エンドポイントを作成するだけです。詳細については、インターフェイスエンドポイントの作成を参照してください。

Kinesis Producer Library (KPL) と Kinesis Consumer Library (KCL) は、パブリックエンドポイントまたはプライベートインターフェイス VPC エンドポイントのいずれかを使用している Amazon CloudWatch や Amazon DynamoDB などの AWS サービスを呼び出します。例えば、KCL アプリケーションが実行されている VPC で DynamoDB インターフェイス VPC エンドポイントが有効になっている場合、DynamoDB と KCL アプリケーション間の呼び出しは、そのインターフェイス VPC エンドポイントを経由して流れます。

Kinesis Data Streams の VPCE エンドポイントへのアクセス制御

VPC エンドポイントポリシーは、VPC エンドポイントにポリシーをアタッチするか、IAM ユーザー、グループ、またはロールにアタッチされたポリシーの追加フィールドを使用して、アクセスが指定された VPC エンドポイント経由のみで行われるように制限することで、アクセスを制御することを可能にします。これらのポリシーは、指定された VPC エンドポイント経由での Kinesis データストリームアクションへのアクセスのみを付与する IAM ポリシーと組み合わせて使用するときに、特定のストリームへのアクセスを指定された VPC エンドポイントに制限するために使用できます。

以下は、Kinesis データストリームにアクセスするためのエンドポイントポリシーの例です。

- VPC ポリシーの例: 読み取り専用アクセス – このサンプルポリシーは、VPC エンドポイントにアタッチできます。詳細については、Amazon VPC のリソースに対するアクセスの制御を参照してください。このポリシーは、アタッチされている VPC エンドポイント経由の Kinesis データストリームの一覧表示と説明のみにアクションを制限します。

```
{
  "Statement": [
    {
      "Sid": "ReadOnly",
      "Principal": "*",
      "Action": [
        "kinesis:List*",
        "kinesis:Describe*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

- VPC ポリシーの例: アクセスを特定の Kinesis データストリームに制限 - このサンプルポリシーは、VPC エンドポイントにアタッチできます。このポリシーは、ポリシーがアタッチされている VPC エンドポイント経由の特定のデータストリームにアクセスを制限します。

```
{
  "Statement": [
    {
      "Sid": "AccessToSpecificDataStream",
```

```
      "Principal": "*",
      "Action": "kinesis:*",
      "Effect": "Allow",
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream"
    }
  ]
}
```

- IAM ポリシーの例: 特定のストリームへのアクセスを特定の VPC エンドポイントからのみに制限 - このサンプルポリシーは、IAM ユーザー、ロール、またはグループにアタッチできます。このポリシーは、指定された Kensis データストリームへのアクセスが、指定された VPC エンドポイント以外からは行われないように制限します。

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AccessFromSpecificEndpoint",
         "Action": "kinesis:*",
         "Effect": "Deny",
         "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream",
         "Condition": { "StringNotEquals" : { "aws:sourceVpce": "vpce-11aa22bb" } }
      }
   ]
}
```

Kinesis Data Streams の VPC エンドポイントポリシーの可用性

ポリシーを使用した Kinesis Data Streams インターフェイス VPC エンドポイントは、次のリージョンでサポートされています。

- 欧州 (パリ)
- 欧州 (アイルランド)
- 米国東部 (バージニア北部)
- 欧州 (ストックホルム)
- 米国東部 (オハイオ)

- 欧州 (フランクフルト)
- 南米 (サンパウロ)
- 欧州 (ロンドン)
- アジアパシフィック (東京)
- 米国西部 (北カリフォルニア)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- 中国 (北京)
- 中国 (寧夏)
- アジアパシフィック (香港)
- 中東 (バーレーン)
- 中東 (アラブ首長国連邦)
- 欧州 (ミラノ)
- アフリカ (ケープタウン)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (ソウル)
- カナダ (中部)
- usw2-az4 を除く米国西部 (オレゴン)
- AWS GovCloud （ 米国東部 ）
- AWS GovCloud （ 米国西部 ）
- アジアパシフィック (大阪)
- 欧州 (チューリッヒ)
- アジアパシフィック (ハイデラバード)

IAM を使用した Amazon Kinesis Data Streams リソースへのアクセスの制御

AWS Identity and Access Management (IAM) では、次のことを実行できます。

- AWS アカウントでユーザーとグループを作成する
- AWS アカウント内の各ユーザーに一意のセキュリティ認証情報を割り当てる

- AWS リソースを使用してタスクを実行するための各ユーザーのアクセス許可を制御する
- 別の AWS アカウントのユーザーに AWS リソースの共有を許可する
- AWS アカウントのロールを作成し、引き受けることができるユーザーまたはサービスを定義する
- エンタープライズの既存の ID を使用して、 AWS リソースを使用してタスクを実行するためのアクセス許可を付与する

Kinesis Data Streams と組み合わせて IAM を使用すると、組織のユーザーが特定の Kinesis Data Streams API アクションを使用してタスクを実行できるかどうか、また、特定の AWS リソースを使用できるかどうかを制御できます。

Kinesis Client Library (KCL) を使用してアプリケーションを開発する場合、ポリシーには Amazon DynamoDB と Amazon のアクセス許可が含まれている必要があります CloudWatch。KCL は DynamoDB を使用してアプリケーションの状態情報を追跡し、ユーザーに代わって KCL メトリクスを CloudWatch CloudWatch に送信します。KCL の詳細については、KCL 1.x コンシューマーの開発を参照してください。

IAM の詳細については、以下を参照してください。

- AWS Identity and Access Management (IAM)
- IAM の使用開始
- IAM ユーザーガイド

IAM と Amazon DynamoDB の詳細については、Amazon DynamoDB デベロッパーガイドのIAM を使用した Amazon DynamoDB リソースへのアクセスのコントロールを参照してください。

IAM と Amazon の詳細については CloudWatch、「Amazon ユーザーガイド」のAWS 「アカウントへのユーザーアクセスの制御 CloudWatch 」を参照してください。

内容

- ポリシー構文
- Kinesis Data Streams のアクション
- Kinesis Data Streams の Amazon リソースネーム (ARN)
- Kinesis Data Streams のポリシーの例
- 別のアカウントとのデータストリームの共有
- 別のアカウントの Kinesis Data Streams から読み取るように AWS Lambda 関数を設定する

- リソースベースのポリシーを使用したアクセスの共有

ポリシー構文

IAM ポリシーは、1 つ、または複数のステートメントで構成される JSON ドキュメントです。各ステートメントは次のように構成されます。

```
{
  "Statement":[{
    "Effect":"effect",
    "Action":"action",
    "Resource":"arn",
    "Condition":{
      "condition":{
        "key":"value"
        }
      }
    }
  ]
}
```

ステートメントはさまざまなエレメントで構成されています。

- [Effect]: effect は、Allow または Deny にすることができます。デフォルトでは、IAM ユーザーはリソースおよび API アクションを使用するアクセス許可がないため、リクエストはすべて拒否されます。明示的な許可はデフォルトに優先します。明示的な拒否はすべての許可に優先します。
- [アクション]: アクション は、アクセス許可を付与または拒否する対象とする、特定の API アクションです。
- [リソース]: アクションによって影響を及ぼされるリソースです。ステートメント内でリソースを指定するには、Amazonリソースネーム(ARN)を使用する必要があります。
- Condition: condition はオプションです。これらは、ポリシーがいつ有効になるかを制御するために使用できます。

IAM のポリシーを作成および管理するときは、IAM Policy Generator と IAM Policy Simulator を使用することもできます。

Kinesis Data Streams のアクション

IAM ポリシーステートメントで、IAM をサポートするすべてのサービスからの任意の API アクションを指定できます。Kinesis Data Streams の場合、API アクションの名前とともに次のプレフィックスを使用します: `kinesis:`。例えば、`kinesis:CreateStream`、`kinesis:ListStreams`、および `kinesis:DescribeStreamSummary` のようになります。

単一のステートメントで複数のアクションを指定するには、次のようにカンマで区切ります。

```
"Action": ["kinesis:action1", "kinesis:action2"]
```

ワイルドカードを使用して複数のアクションを指定することもできます。たとえば、Getという単語で始まる名前のすべてのアクションは、以下のように指定できます。

```
"Action": "kinesis:Get*"
```

すべてのKinesis Data Streams オペレーションを指定するには、次のように * ワイルドカードを使用します。

```
"Action": "kinesis:*"
```

Kinesis Data Streams API アクションの一覧については、Amazon Kinesis API リファレンス を参照してください。

Kinesis Data Streams の Amazon リソースネーム (ARN)

各 IAM ポリシーステートメントは、ARN を使用して指定されたリソースに適用されます。

Kinesis data streams には、以下の ARN リソースフォーマットを使用します。

```
arn:aws:kinesis:region:account-id:stream/stream-name
```

例:

```
"Resource": arn:aws:kinesis:*:111122223333:stream/my-stream
```

Kinesis Data Streams のポリシーの例

次のポリシー例は、Kinesis Data Streams へのユーザーアクセスの制御方法について説明しています。

Example 1: Allow users to get data from a stream

Example

このポリシーは、ユーザーまたはグループが、指定されたストリームで DescribeStreamSummary、GetShardIterator、または GetRecords 操作を実行し、任意のストリームで ListStreams を実行できるようにします。このポリシーは、特定のストリームからデータを取得できるユーザーに適用できます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:Get*",
                "kinesis:DescribeStreamSummary"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Example 2: Allow users to add data to any stream in the account

Example

このポリシーは、ユーザーまたはグループが、アカウント内の任意のストリームで PutRecord 操作を使用できるようにします。このポリシーは、アカウント内のすべてのストリームにデータレコードを追加できるユーザーに適用できます。

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/*"
            ]
        }
    ]
}
```

Example 3: Allow any Kinesis Data Streams action on a specific stream

Example

このポリシーでは、ユーザーまたはグループが、指定したストリームに対して任意の Kinesis Data Streams オペレーションを実行できます。このポリシーは、特定のストリームに対して管理的な制御を行えるユーザーに適用できます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        }
    ]
}
```

Example 4: Allow any Kinesis Data Streams action on any stream

Example

このポリシーでは、ユーザーまたはグループが、アカウントの任意のストリームに対して任意の Kinesis Data Streams オペレーションを実行できます。このポリシーはすべてのストリームへの完全なアクセス権を付与するため、管理者のみに制限する必要があります。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:*:111122223333:stream/*"
            ]
        }
    ]
}
```

別のアカウントとのデータストリームの共有

> **Note**
>
> Kinesis Producer Library は現在、データストリームへの書き込み時にストリーム ARN の指定をサポートしていません。クロスアカウントデータストリームに書き込む場合は、 AWS SDK を使用します。

リソースベースのポリシーをデータストリームにアタッチして、別のアカウント、IAM ユーザー、または IAM ロールへのアクセス権を付与します。リソースベースのポリシーは、データストリームなどのリソースにアタッチする JSON ポリシードキュメントです。これらのポリシーでは、そのリソースに対して特定のアクションを実行する指定されたプリンシパルアクセス許可を付与し、このアクセス許可が適用される条件を定義します。ポリシーには複数のステートメントを含めることができます。リソースベースのポリシーで、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または AWS サービスを含めることができます。ポリシーは、Kinesis Data Streams コンソール、API、SDK で設定できます。

拡張ファンアウトなどの登録済みコンシューマーへのアクセスを共有するには、データストリーム ARN とコンシューマー ARN の両方でポリシーが必要であることに注意してください。

クロスアカウントアクセスの有効化

クロスアカウントアクセスを有効にするには、アカウント全体、または別のアカウントの IAM エンティティをリソースベースのポリシーのプリンシパルとして指定します。リソースベースのポリシー

にクロスアカウントのプリンシパルを追加しても、信頼関係は半分しか確立されない点に注意してください。プリンシパルとリソースが別々の AWS アカウントにある場合は、アイデンティティベースのポリシーを使用して、プリンシパルにリソースへのアクセスを許可する必要があります。ただし、リソースベースのポリシーで、同じアカウントのプリンシパルへのアクセス権が付与されている場合は、ID ベースのポリシーをさらに付与する必要はありません。

クロスアカウントアクセスでリソースベースのポリシーを使用する方法の詳細については、「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

データストリーム管理者は、 AWS Identity and Access Management ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。JSON ポリシーの`Action`要素には、ポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連付けられた AWS API オペレーションと同じです。

共有可能な Kinesis Data Streams のアクション:

アクション	アクセスのレベル
DescribeStreamConsumer	コンシューマー
DescribeStreamSummary	データストリーム
GetRecords	データストリーム
GetShardIterator	データストリーム
ListShards	データストリーム
PutRecord	データストリーム
PutRecords	データストリーム
SubscribeToShard	コンシューマー

リソースベースのポリシーを使用して、データストリームまたは登録済みコンシューマーにクロスアカウントアクセスを許可する例を以下に示します。

クロスアカウントのアクションを実行するには、データストリームにアクセスするためのストリーム ARN と登録済みコンシューマーにアクセスするためのコンシューマー ARN を指定する必要があります。

Kinesis データストリームのリソースベースのポリシーの例

登録済みコンシューマーの共有には、必要なアクションを実行するために、データストリームポリシーとコンシューマーポリシーの両方が必要になります。

> **Note**
>
> 次に示すのは、Principal の有効な値の例です。
>
> - {"AWS": "123456789012"}
> - IAM ユーザー – {"AWS": "arn:aws:iam::123456789012:user/user-name"}
> - IAM ロール – {"AWS":["arn:aws:iam::123456789012:role/role-name"]}
> - 複数のプリンシパル (アカウント、ユーザー、ロールの組み合わせが可能) — {"AWS": ["123456789012", "123456789013", "arn:aws:iam::123456789012:user/user-name"]}

Example 1: Write access to the data stream

Example

```
{
    "Version": "2012-10-17",
    "Id": "__default_write_policy_ID",
    "Statement": [
        {
            "Sid": "writestatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "Account12345"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards",
                "kinesis:PutRecord",
                "kinesis:PutRecords"
```

```
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

Example 2: Read access to the data stream

Example

```
{
    "Version": "2012-10-17",
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
        {
            "Sid": "sharedthroughputreadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "Account12345"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards",
                "kinesis:GetRecords",
                "kinesis:GetShardIterator"
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

Example 3: Share enhanced fan-out read access to a registered consumer

Example

データストリームポリシーステートメント:

```
{
    "Version": "2012-10-17",
```

```
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
        {
            "Sid": "consumerreadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::Account12345:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

コンシューマーポリシーステートメント:

```
{
    "Version": "2012-10-17",
    "Id": "__default_efo_read_policy_ID",
    "Statement": [
        {
            "Sid": "eforeadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::Account12345:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamConsumer",
                "kinesis:SubscribeToShard"
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC/consumer/consumerDEF:1674696300"
        }
    ]
}
```

最小特権の原則を維持するため、アクションやプリンシパルフィールドではワイルドカード (*) はサポートされていません。

データストリームのポリシーのプログラムによる管理

以外では AWS Management Console、Kinesis Data Streams にはデータストリームポリシーを管理するための 3 つの APIS があります。

- PutResourcePolicy
- GetResourcePolicy
- DeleteResourcePolicy

PutResourePolicy を使用して、データストリームまたはコンシューマーのポリシーをアタッチまたは上書きします。GetResourcePolicy を使用して、指定したデータストリームまたはコンシューマーのポリシーを確認し、表示します。DeleteResourcePolicy を使用して、指定したデータストリームまたはコンシューマーのポリシーを削除します。

ポリシー制限

Kinesis Data Streams リソースポリシーには次の制限があります。

- ワイルドカード (*) は、データストリームまたは登録されたコンシューマーに直接アタッチされたリソースポリシーを通じて広範なアクセスが付与されないようにするためにサポートされていません。さらに、以下のポリシーを注意深く調べて、広範なアクセスを許可していないことを確認してください。
  - 関連付けられた AWS プリンシパルにアタッチされたアイデンティティベースのポリシー (IAM ロールなど)
  - 関連付けられたリソースにアタッチされた AWS リソースベースのポリシー (AWS Key Management Service KMS キーなど)
- AWS サービスプリンシパルは、混乱した代理 の可能性を防ぐためにプリンシパルではサポートされていません。
- フェデレーションプリンシパルはサポートされていません。
- 正規ユーザー ID はサポートされていません。
- ポリシーのサイズは 20 KB までです。

暗号化されたデータへのアクセスの共有

AWS マネージド KMS キーでデータストリームのサーバー側の暗号化を有効にしていて、リソースポリシーを介してアクセスを共有する場合は、カスタマーマネージドキー (CMK) の使用に切り替え

る必要があります。詳細については、「Kinesis Data Streams 用のサーバー側の暗号化とは?」を参照してください。さらに、KMS のクロスアカウント共有機能を使用して、共有プリンシパルエンティティが CMK にアクセスできるようにする必要があります。共有プリンシパルエンティティの IAM ポリシーも必ず変更してください。詳細については、「その他のアカウントのユーザーに KMS キーの使用を許可する」を参照してください。

別のアカウントの Kinesis Data Streams から読み取るように AWS Lambda 関数を設定する

別のアカウントの Kinesis Data Streams から読み取るように Lambda 関数を設定する方法の例については、「クロスアカウント AWS Lambda 関数とのアクセスの共有」を参照してください。

リソースベースのポリシーを使用したアクセスの共有

> **Note**
>
> 既存のリソースベースのポリシーを更新すると既存のポリシーが置き換えられるため、新しいポリシーには必要な情報をすべて含めるようにしてください。

クロスアカウント AWS Lambda 関数とのアクセスの共有

Lambda 演算子

1. IAM コンソールに移動して、 AWS Lambda 関数の Lambda 実行ロールとして使用する IAM ロールを作成します。必要な Kinesis Data Streams と Lambda 呼び出し許可`AWSLambdaKinesisExecutionRole`を持つ マネージド IAM ポリシーを追加します。このポリシーは、ユーザーがアクセスする可能性のあるすべての Kinesis Data Streams リソースへのアクセスも付与します。
2. AWS Lambda コンソール で、Kinesis Data Streams データストリームのレコードを処理する AWS Lambda 関数を作成し、実行ロールのセットアップ中に、前のステップで作成したロールを選択します。
3. リソースポリシーを設定するための実行ロールを Kinesis Data Streams リソースの所有者に提供します。
4. Lambda 関数のセットアップを完了します。

Kinesis Data Streams リソースの所有者

1. Lambda 関数を呼び出すクロスアカウントの Lambda 実行ロールを取得します。
2. Amazon Kinesis Data Streams コンソールで、データストリームを選択します。[データストリーム共有] タブを選択し、[共有ポリシーの作成] ボタンをクリックしてビジュアルポリシーエディタを起動します。登録済みのコンシューマーをデータストリーム内で共有するには、コンシューマーを選択し、次に [共有ポリシーの作成] を選択します。JSON ポリシーを直接作成することもできます。
3. クロスアカウントの Lambda 実行ロールをプリンシパルとして指定し、アクセスを共有する正確な Kinesis Data Streams アクションを指定します。必ず `kinesis:DescribeStream` アクションを含めてください。Kinesis Data Streams リソースポリシーの例については、「Kinesis データストリームのリソースベースのポリシーの例」を参照してください。
4. ポリシーの作成を選択するかPutResourcePolicy、 を使用してポリシーをリソースにアタッチします。

クロスアカウントの KCL コンシューマーとのアクセスの共有

- KCL 1.x を使用している場合は、KCL 1.15.0 以降を使用していることを確認してください。
- KCL 2.x を使用している場合は、KCL 2.5.3 以降を使用していることを確認してください。

KCL 演算子

1. KCL アプリケーションを実行する IAM ユーザーまたは IAM ロールをリソース所有者に提供します。
2. リソース所有者にデータストリームまたはコンシューマー ARN を依頼してください。
3. KCL 設定の一部として、提供されたストリーム ARN を指定していることを確認してください。
   - KCL 1.x の場合: KinesisClientLibConfigurationコンストラクタを使用してストリーム ARN を指定します。
   - KCL 2.x の場合: ストリーム ARN または のみを Kinesis Client Library StreamTrackerに提供できますConfigsBuilder。には StreamTracker、ライブラリによって生成された DynamoDB リーステーブルからストリーム ARN と作成エポックを指定します。拡張ファンアウトなどの共有登録済みコンシューマーから読み取る場合は、 を使用し、コンシューマー ARN StreamTracker も指定します。

Kinesis Data Streams リソースの所有者

1. KCL アプリケーションを実行するクロスアカウントの IAM ユーザーまたは IAM ロールを取得します。
2. Amazon Kinesis Data Streams コンソールで、データストリームを選択します。[データストリーム共有] タブを選択し、[共有ポリシーの作成] ボタンをクリックしてビジュアルポリシーエディタを起動します。登録済みのコンシューマーをデータストリーム内で共有するには、コンシューマーを選択し、次に [共有ポリシーの作成] を選択します。JSON ポリシーを直接作成することもできます。
3. クロスアカウントの KCL アプリケーションの IAM ユーザーまたは IAM ロールをプリンシパルとして指定し、アクセスを共有する正確な Kinesis Data Streams アクションを指定します。Kinesis Data Streams リソースポリシーの例については、「Kinesis データストリームのリソースベースのポリシーの例」を参照してください。
4. ポリシーの作成を選択するかPutResourcePolicy、 を使用してポリシーをリソースにアタッチします。

暗号化されたデータへのアクセスの共有

AWS マネージド KMS キーでデータストリームのサーバー側の暗号化を有効にしていて、リソースポリシーを介してアクセスを共有する場合は、カスタマーマネージドキー (CMK) の使用に切り替える必要があります。詳細については、「Kinesis Data Streams 用のサーバー側の暗号化とは?」を参照してください。さらに、KMS のクロスアカウント共有機能を使用して、共有プリンシパルエンティティが CMK にアクセスできるようにする必要があります。共有プリンシパルエンティティの IAM ポリシーも必ず変更してください。詳細については、「その他のアカウントのユーザーに KMS キーの使用を許可する」を参照してください。

Amazon Kinesis Data Streams のコンプライアンス検証

サードパーティーの監査者は、複数の コンプライアンスプログラムの一環として Amazon Kinesis Data Streams のセキュリティと AWS コンプライアンスを評価します。これらのプログラムには、SOC、PCI、FedRAMP、HIPAA などがあります。

特定のコンプライアンスプログラムの対象となる AWS サービスのリストについては、「コンプライアンスAWS プログラムによる対象範囲内のサービス」を参照してください。一般的な情報については、「AWS コンプライアンスプログラム」を参照してください。

を使用して、サードパーティーの監査レポートをダウンロードできます AWS Artifact。詳細については、AWS 「アーティファクト でのレポートのダウンロード」を参照してください。

Kinesis Data Streams を使用する際のお客様のコンプライアンス責任は、お客様のデータの機密性や貴社のコンプライアンス目的、適用可能な法律および規制によって決定されます。Kinesis Data Streams の使用が HIPAA、PCI、FedRAMP などの標準に準拠していることを前提としている場合、は以下に役立つリソース AWS を提供します。

- セキュリティとコンプライアンスのクイックスタートガイド – これらのデプロイガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスに重点を置いたベースライン環境を にデプロイする手順について説明します AWS。
- 「HIPAA セキュリティとコンプライアンスの設計」ホワイトペーパー – このホワイトペーパーでは、企業が AWS を使用して HIPAA 準拠のアプリケーションを作成する方法について説明します。
- AWS コンプライアンスリソース – お客様の業界や地域に適用される可能性のあるワークブックとガイドのコレクション
- AWS Config – リソース設定が社内プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価するこの AWS サービス。
- AWS Security Hub – この AWS サービスは、 内のセキュリティ状態を包括的に把握 AWS し、セキュリティ業界標準とベストプラクティスへの準拠を確認するのに役立ちます。

Amazon Kinesis Data Streams の耐障害性

AWS グローバルインフラストラクチャは、 AWS リージョンとアベイラビリティーゾーンを中心に構築されています。 AWS リージョンは、低レイテンシー、高スループット、および高度に冗長なネットワークで接続された、物理的に分離された複数のアベイラビリティーゾーンを提供します。アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS リージョンとアベイラビリティーゾーンの詳細については、AWS 「 グローバルインフラストラクチャ」を参照してください。

グローバル AWS インフラストラクチャに加えて、Kinesis Data Streams には、データの耐障害性とバックアップのニーズをサポートするのに役立ついくつかの機能が用意されています。

Amazon Kinesis Data Streams の災害対策

Amazon Kinesis Data Streams アプリケーションを使用してストリームからのデータを処理するときに、次のレベルで障害が発生する可能性があります。

- レコードプロセッサで障害が発生する
- ワーカーで障害が発生する、またはワーカーをインスタンス化したアプリケーションのインスタンスで障害が発生する
- アプリケーションのインスタンスを 1 つ、または複数ホストしている EC2 インスタンスで障害が発生する

レコードプロセッサの障害

ワーカーは Java ExecutorServiceタスクを使用してレコードプロセッサメソッドを呼び出します。タスクが失敗した場合でも、ワーカーはレコードプロセッサが処理していたシャードの制御を維持します。ワーカーは、このシャードを処理するための新しいレコードプロセッサタスクを開始します。詳細については、読み込みのスロットリングを参照してください。

ワーカーまたはアプリケーションの障害

ワーカーまたは Amazon Kinesis Data Streams アプリケーションのインスタンスで障害が発生した場合は、状況を検出して処理する必要があります。例えば、`Worker.run` メソッドが例外をスローする場合、この例外を認識して処理する必要があります。

アプリケーション自体に障害が発生した場合は、これを検出し、再起動する必要があります。アプリケーションが起動すると、アプリケーションが新しいワーカーをインスタンス化します。次に、インスタンス化されたワーカーが、処理するシャードに自動的に割り当てられる新しいレコードプロセッサをインスタンス化します。シャードは、障害が発生する前にこれらのレコードプロセッサが処理していたものと同じシャードである場合も、これらのプロセッサにとって新しいシャードである場合もあります。

ワーカーまたはアプリケーションに障害が発生してもその障害が検出されず、このアプリケーションの他のインスタンスが他の EC2 インスタンスで実行されているという場合は、これらの他のインスタンス上のワーカーが障害に対処します。これらのワーカーは、追加のレコードプロセッサを作成することで、障害が発生したワーカーで処理されなくなったシャードを処理します。これに伴って、これらの他の EC2 インスタンスの負荷も増加します。

ここで説明するシナリオでは、ワーカーやアプリケーションに障害が発生した場合でも、ホストしている EC2 インスタンスは実行されているため、Auto Scaling グループによって再起動されないことを前提としています。

Amazon EC2 インスタンスの障害

アプリケーションの EC2 インスタンスを Auto Scaling グループで実行することをお勧めします。このようにすることで、いずれかの EC2 インスタンスに障害が発生した場合でも、Auto Scaling グループによって、自動的にそのインスタンスを置き換える新しいインスタンスが起動されます。起動時に Amazon Kinesis Data Streams アプリケーションを起動するようにインスタンスを設定する必要があります。

Kinesis Data Streams のインフラストラクチャセキュリティ

マネージドサービスである Amazon Kinesis Data Streams は、ホワイトペーパー「Amazon Web Services: セキュリティプロセスの概要」に記載されている AWS グローバルネットワークセキュリティの手順で保護されています。

が AWS 公開した API コールを使用して、ネットワーク経由で Kinesis Data Streams にアクセスします。クライアントは、Transport Layer Security (TLS) 1.2 以降をサポートする必要があります。クライアントは、Ephemeral Diffie-Hellman (DHE) や Elliptic Curve Ephemeral Diffie-Hellman (ECDHE) などの Perfect Forward Secrecy (PFS) を使用する暗号スイートもサポートする必要があります。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。

また、リクエストには、アクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、AWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

Kinesis Data Streams のセキュリティのベストプラクティス

Amazon Kinesis Data Streams には、独自のセキュリティポリシーを策定および実装する際に考慮すべきさまざまなセキュリティ機能が用意されています。以下のベストプラクティスは一般的なガイドラインであり、完全なセキュリティソリューションを説明するものではありません。これらのベストプラクティスはお客様の環境に適切ではないか、十分ではない場合があるため、これらは処方箋ではなく、有用な考慮事項と見なしてください。

最小特権アクセスの実装

許可を付与する場合、どのユーザーにどの Kinesis Data Streams リソースに対する許可を付与するかは、お客様が決定します。これらのリソースで許可したい特定のアクションを有効にするのも、お客様になります。このため、タスクの実行に必要なアクセス許可のみを付与する必要があります。最小特権アクセスの実装は、セキュリティリスクと、エラーや悪意によってもたらされる可能性のある影響の低減における基本になります。

IAM ロールの使用

プロデューサーおよびクライアントアプリケーションは、Kinesis Data Streams にアクセスするための有効な認証情報を持っている必要があります。 AWS 認証情報は、クライアントアプリケーションまたは Amazon S3 バケットに直接保存しないでください。これらは自動的にローテーションされない長期的な認証情報であり、漏洩するとビジネスに大きな影響が及ぶ場合があります。

代わりに、IAM ロールを使用して、Kinesis Data Streams にアクセスするためのプロデューサーおよびクライアントアプリケーションの一時的な認証情報を管理してください。ロールを使用するときは、他のリソースにアクセスするために長期的な認証情報 (ユーザー名とパスワード、またはアクセスキーなど) を使用する必要がありません。

詳細については、「IAM ユーザーガイド」にある下記のトピックを参照してください。

- IAM ロール
- ロールの一般的なシナリオ: ユーザー、アプリケーション、およびサービス

依存リソースでのサーバー側の暗号化の実装

保管中のデータと転送中のデータは Kinesis Data Streams で暗号化できます。詳細については、「Amazon Kinesis Data Streams のデータ保護」を参照してください。

CloudTrail を使用して API コールをモニタリングする

Kinesis Data Streams は AWS CloudTrail、Kinesis Data Streams のユーザー、ロール、または サービスによって実行されたアクションを記録する AWS サービスである と統合されています。

で収集された情報を使用して CloudTrail、Kinesis Data Streams に対するリクエスト、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

詳細については、「the section called “AWS CloudTrailを使用した Amazon Kinesis Data Streams API コールのログ記録”」を参照してください。

ドキュメント履歴

次のテーブルに、Amazon Kinesis Data Streams ドキュメントの重要な変更を示します。

変更	説明	変更日
アカウント間でのデータストリームの共有のサポートが追加されました。	別のアカウントとのデータストリームの共有が追加されました。	2023 年 11 月 22 日
オンデマンドとプロビジョニングのデータストリームキャパシティモードのサポートを追加しました。	データストリーム容量モードの選択が追加されました。	2021 年 11 月 29 日
サーバー側の暗号化の新しいコンテンツ。	Amazon Kinesis Data Streams のデータ保護が追加されました。	2017 年 7 月 7 日
CloudWatch 強化された指標のための新しいコンテンツ。	更新済み Amazon Kinesis Data Streams のモニタリング。	2016 年 4 月 19 日
拡張 Kinesis エージェントの新しいコンテンツ。	更新済み Kinesis エージェントを使用した Amazon Kinesis Data Streams への書き込み。	2016 年 4 月 11 日
Kinesis エージェントを使用するための新しいコンテンツ。	Kinesis エージェントを使用した Amazon Kinesis Data Streams への書き込みが追加されました。	2015 年 10 月 2 日

変更	説明	変更日
リリース 0.10.0 への KPL コンテンツの更新	Amazon Kinesis Producer Library を使用したプロデューサーの開発が追加されました。	2015 年 7 月 15 日
設定可能なメトリクスの KCL メトリクスのトピックの更新。	Amazon による Kinesis Client Library のモニタリング CloudWatchが追加されました。	2015 年 7 月 9 日
コンテンツの再編成。	コンテンツトピックを大幅に再編成し、より簡潔なツリー表示とより論理的なグループ化を行いました。	2015 年 7 月 01 日
新しい KPL 開発者ガイドのトピック。	Amazon Kinesis Producer Library を使用したプロデューサーの開発が追加されました。	2015 年 6 月 02 日
新しい KCL メトリクスのトピック。	Amazon による Kinesis Client Library のモニタリング CloudWatchが追加されました。	2015 年 5 月 19 日
KCL .NET のサポート	.NET での Kinesis Client Library コンシューマーの開発が追加されました。	2015 年 5 月 1 日
KCL Node.js のサポート	ode.js での Kinesis Client Library コンシューマーの開発が追加されました。	2015 年 3 月 26 日
KCL Ruby のサポート	KCL Ruby ライブラリへのリンクを追加しました。	2015 年 1 月 12 日
新しい API PutRecords	新しい PutRecords API the section called “PutRecords を使用した複数のレコードの追加” に関する情報をに追加しました。	2014 年 12 月 15 日
タグ指定のサポート	Amazon Kinesis Data Streams のストリームのタグ付けが追加されました。	2014 年 9 月 11 日

変更	説明	変更日
CloudWatch 新しいメトリクス	`GetRecords.IteratorAgeMilliseconds` メトリックを Amazon Kinesis Data Streams のディメンションとメトリクス に追加しました。	2014 年 9 月 3 日
モニタリングに関する章の新規追加	Amazon Kinesis Data Streams のモニタリング と Amazon による Amazon Kinesis Data Streams Service のモニタリング CloudWatch が追加されました。	2014 年 7 月 30 日
デフォルトのシャード制限	クォータと制限 の更新点: デフォルトのシャード制限が 5 から 10 に増えました。	2014 年 2 月 25 日
デフォルトのシャード制限	クォータと制限 の更新点: デフォルトのシャード制限が 2 から 5 に増えました。	2014 年 1 月 28 日 2014 年 1 月 3 日
API バージョンに合わせた更新	Kinesis Data Streams API の 2013-12-02 バージョンに合わせた更新。	2013年12月12日
初回リリース	Amazon Kinesis デベロッパーガイドの初回リリース。	2013 年 11 月 14 日

AWS 用語集

AWS の最新の用語については、「AWS の用語集リファレンス」の「AWS 用語集」を参照してください。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。