aws

デベロッパーガイド

Amazon Kinesis Data Streams

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Kinesis Data Streams: デベロッパーガイド

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスは、Amazon が所有しない製品またはサービスに関連付けて使用することはできず、お客様の混乱を招く可能性がある方法、または Amazon の評判を毀損する、もしくは信用を損なう方法で使用することもできません。Amazon が所有しないその他すべての商標は、それぞれの所有者の資産です。これらの所有者は、必ずしも Amazon と提携している、関連がある、または後援を受けているとは限りません。

Table of Contents

Amazon Kinesis Data Streams とは 1
Kinesis Data Streams でどのようなことができますか? 1
Kinesis Data Streams を使用する利点 2
関連サービス 3
用語と概念 4
Kinesis Data Streams の概要レベルのアーキテクチャを確認する 4
Kinesis Data Streams の用語に精通する 5
Kinesis Data Streams 5
データレコード 5
容量モード 5
保持期間 5
プロデューサー 6
コンシューマー 6
Amazon Kinesis Data Streams アプリケーション 6
シャード 6
パーティションキー 7
シーケンス番号 7
Kinesis Client Library 7
アプリケーション名 7
サーバー側の暗号化 8
クォータと制限 9
API の制限 12
KDS コントロールプレーン API の制限 12
KDS データプレーンの API の制限 17
クォータの引き上げ 19
Amazon Kinesis Data Streams をセットアップするための前提条件を完了する 21
AWS にサインアップする 21
ライブラリとツールをダウンロードする 22
開発環境を設定する 22
AWS CLI を使用して Amazon Kinesis Data Streams オペレーションを実行する 23
チュートリアル: AWS CLI for Kinesis Data Streams をインストールして設定する 23
AWS CLI のインストール 23
AWS CLI を設定する 25
チュートリアル: AWS CLI を使用した基本的な Kinesis Data Streams の操作を実行する 25

ステップ 1: ストリームを作成する 25
ステップ 2: レコードを入力する 27
ステップ 3: レコードを取得する 28
ステップ 4: クリーンアップする 31
入門チュートリアル 32
チュートリアル: KPL と KCL 2.x を使用して株式データをリアルタイム処理する 32
前提条件を満たす 33
データストリームを作成する 34
IAM ユーザーとポリシーを作成する 35
コードをダウンロードしてビルドする 40
プロデューサーを実装する 41
コンシューマーを実装する 46
(オプション) コンシューマーを拡張する 50
リソースをクリーンアップする 52
チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する 53
前提条件を満たす 54
データストリームを作成する 55
IAM ユーザーとポリシーを作成する 57
実装コードをダウンロードおよびビルドする 62
プロデューサーを実装する 63
コンシューマーを実装する 68
(オプション) コンシューマーを拡張する 72
リソースをクリーンアップする 73
チュートリアル: Amazon 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 75
前提条件 76
ステップ 1: アカウントを設定する 76
ステップ 2: をセットアップする AWS CLI 80
ステップ 3: アプリケーションを作成する 81
チュートリアル: AWS Lambda を Amazon Kinesis Data Streams で使用する 99
Amazon Kinesis AWS のストリーミングデータソリューションを使用する 99
Kinesis Data Streams を作成して管理する 101
適切なストリームモードを選ぶ 102
Kinesis Data Streams の各種モードについて 102
オンデマンド標準モードの機能とユースケース 103
オンデマンドアドバンテージモードの機能とユースケース 104

プロビジョンドモード機能とユースケース 105
モードの切り替え 106
を使用してストリームを作成する AWS マネジメントコンソール 107
API を使用してストリームを作成する 108
Kinesis Data Streams クライアントの構築 108
ストリームを作成する 109
ストリームを更新する 110
コンソールを使用する 111
API を使用する 112
を使用する AWS CLI 112
ストリームの一覧表示 112
シャードを一覧表示する 114
ストリームを削除する 117
ストリームをリシャーディングする 117
リシャーディング戦略を決定する 118
シャードを分割する 119
2 つのシャードを結合する 121
リシャーディングアクションを完了する 122
データ保持期間を変更する 125
リソースのタグ付け 126
タグの基本を確認する 126
タグ付けを使用したコストを追跡する 127
タグの制限を理解する 128
Kinesis Data Streams コンソールを使用してストリームにタグ付けする 128
を使用してストリームにタグを付ける AWS CLI 130
Kinesis Data Streams API を使用したストリームにタグを付ける 131
を使用してコンシューマーにタグを付ける AWS CLI 131
Kinesis Data Streams API を使用してコンシューマーにタグ付けする 132
大きなレコードの処理 132
大きなレコードを使用するようにストリームを更新する 133
大きなレコードでストリームのパフォーマンスを最適化する 133
大きなレコードによるスロットリングを軽減する 133
Kinesis Data Streams API を使用して大きなレコードを処理する 134
AWS 大きなレコードと互換性のあるコンポーネント 134
大きなレコードがサポートされるリージョン 138
でレジリエンステストを実行する AWS Fault Injection Service 140

プロビジョンドスループット例外エラー 141
期限切れイテレーター例外エラー 144
データを Kinesis Data Streams に送信する 146
Amazon Kinesis Producer Library (KPL) を使用してプロデューサーを開発する 147
KPL のロールを確認する 148
KPL を使用するメリットを実現する 148
KPL を使用しないタイミングを理解する 149
KPL をインストールする 150
KPL 1.x への移行 150
KPL の Amazon Trust Services (ATS) 証明書への移行 155
KPL でサポートされるプラットフォーム 155
KPL の主要な概念 156
KPL をプロデューサーコードと統合する 158
KPL を使用して Kinesis Data Stream に書き込む 160
KPL を設定する 162
コンシューマーの集約解除を実装する 163
Amazon Data Firehose で KPL を使用する 166
AWS Glue スキーマレジストリで KPL を使用する 167
KPL プロキシ設定を設定する 167
KPL バージョンライフサイクルポリシー 168
Kinesis Data Streams API と AWS SDK for Java を使用してプロデューサーを開発する 169
ストリームにデータを追加する 170
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 176
Kinesis エージェントを使用して Amazon Kinesis Data Streams に書き込む 177
Kinesis Agent の前提条件を完了する 177
エージェントをダウンロードしてインストールする 178
エージェントを設定して開始する 179
エージェント設定を指定する 180
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む 183
エージェントを使用してデータを事前処理する 184
エージェント CLI コマンドを使用する 188
よくある質問 189
その他の AWS サービスを使用した Kinesis Data Streams への書き込み 190
AWS Amplify を使用して Kinesis Data Stream に書き込む 191
Amazon Aurora を使用して Kinesis Data Stream に書き込む 191
Amazon CloudFront を使用して Kinesis Data Stream に書き込む 192

Amazon CloudWatch Logs を使用して Kinesis Data Streams に書き込む 192
Amazon Connect を使用して Kinesis Data Streams に書き込む 192
AWS Database Migration Service を使用して Kinesis Data Stream に書き込む 193
Amazon DynamoDB を使用して Kinesis Data Streams に書き込む 193
Amazon EventBridge を使用して Kinesis Data Streams に書き込む 193
AWS IoT Core を使用して Kinesis Data Stream に書き込む 193
Amazon Relational Database Service を使用して Kinesis Data Streams に書き込む 194
Amazon Pinpoint を使用して Kinesis Data Streams に書き込む 194
Amazon Quantum Ledger Database (Amazon QLDB) を使用して Kinesis Data Streams に書き込む 194
サードパーティーの統合を使用して Kinesis Data Streams に書き込む 195
Apache Flink 195
Fluentd 195
Debezium 195
Oracle GoldenGate 196
Kafka Connect 196
Adobe Experience 196
Striim 196
Kinesis Data Streams プロデューサーのトラブルシューティング 196
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 197
不正な KMS マスターキーアクセス許可エラーが表示される 199
プロデューサーのその他の一般的な問題のトラブルシューティング 199
Kinesis Data Streams プロデューサーを最適化する 199
KPL の再試行とレート制限の動作をカスタマイズする 200
KPL 集約にベストプラクティスを適用する 201
Kinesis Data Streams からデータを読み取る 202
専用スループットを備えた拡張ファンアウトを開発する 202
共有スループッ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違い 204
最大 50 の拡張ファンアウトコンシューマーでサポートされているリージョン (オンデマンド Advantage のみ) 138
..... 205
拡張ファンアウトコンシューマーを管理する 207
Kinesis コンソールでデータビューワーを使用する 209
Kinesis コンソールでデータストリームをクエリする 210
Kinesis Client Library を使用する 210
Kinesis Client Library (KCL) とは何ですか? 210

KCL の主な機能と利点 211
KCL の概念 212
KCL における DynamoDB メタデータテーブルと負荷分散 213
KCL を使用してコンシューマーを開発する 217
KCL を使用したマルチストリーム処理 229
KCL で AWS Glue スキーマレジストリを使用する 232
KCL コンシューマーアプリケーションに必要な IAM アクセス許可 233
KCL の設定 239
KCL バージョンライフサイクルポリシー 253
旧バージョンの KCL からの移行 254
以前の KCL バージョンのドキュメント 269
AWS SDK for Java を使用してコンシューマーを開発する 352
AWS SDK for Java を使用して共有スループットコンシューマーを開発する 353
AWS SDK for Java 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 358
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 361
AWS Lambdaを使用してコンシューマーを開発する 362
Managed Service for Apache Flink を使用してコンシューマーを開発する 362
Amazon Data Firehose を使用してコンシューマーを開発する 362
Kinesis Data Streams からデータを読み取るためにその他の AWS サービスを使用する 363
Amazon EMR を使用して Kinesis Data Streams からデータを読み取る 363
Amazon EventBridge Pipes を使用して Kinesis Data Streams からデータを読み取る 363
AWS Glue を使用して Kinesis Data Streams からデータを読み取る 364
Amazon Redshift を使用して Kinesis Data Streams からデータを読み取る 364
サードパーティーの統合を使用して Kinesis Data Streams から読み取る 364
Apache Flink 365
Adobe Experience Platform 365
Apache Druid 365
Apache Spark 365
Databricks 366
Kafka Confluent Platform 366
Kinesumer 366
Talend 366
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 366
LeaseManagementConfig コンストラクタでのコンパイルエラー 367
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 368

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 368
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 369
ストリームにデータがある場合でも、GetRecords が空のレコード配列を返す 370
シャードイテレーターが予期せずに終了する 371
コンシューマーレコードの処理が遅れる 371
承認されていない KMS キーの権限エラー 372
DynamoDbException: 更新式で指定されたドキュメントパスが更新に対して無効です 372
コンシューマーのその他の一般的な問題のトラブルシューティング 373
Kinesis Data Streams コンシューマーを最適化する 373
低レイテンシー処理を改善する 374
Amazon Kinesis プロデューサーライブラリ AWS Lambda で を使用してシリアル化されたデータを処理する 375
シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。 375
重複レコードを処理する 377
起動、シャットダウン、スロットリングを処理する 379
Kinesis データストリームを監視する 382
CloudWatch による Kinesis Data Streams サービスを監視する 382
Amazon Kinesis Data Streams のディメンションとメトリクス 383
Kinesis Data Streams の Amazon CloudWatch メトリクスにアクセスする 398
CloudWatch で Kinesis Data Streams エージェントのヘルスを監視する 398
CloudWatch を使用して監視する 399
AWS CloudTrail を使用した Amazon Kinesis Data Streams API コールのログを記録する 400
CloudTrail の Kinesis Data Streams の情報 400
例: Kinesis Data Streams ログファイルエントリ 402
CloudWatch で KCL を監視する 406
メトリクスと名前空間 406
メトリクスレベルとディメンション 406
メトリクスの設定 407
メトリクスの一覧 408
CloudWatch による KPL のモニタリング 425
メトリクス、ディメンション、および名前空間 426
メトリクスレベルと詳細度 426
ローカルアクセスと Amazon CloudWatch のアップロード 427
メトリクスの一覧 428

セキュリティ 432
Kinesis Data Streams のデータ保護 433
Kinesis Data Streams 用のサーバー側の暗号化とは? 433
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 435
サーバー側の暗号化の使用開始方法 436
ユーザー生成の KMS キーを作成して使用する 437
ユーザー生成の KMS キーを使用するための許可 437
KMS キー許可の検証とトラブルシューティングを行う 440
インターフェイス VPC エンドポイントと Kinesis Data Streams を使用する 440
IAM を使用した Kinesis Data Streams リソースへのアクセスの制御 444
ポリシー構文 445
Kinesis Data Streams のアクション 446
Kinesis Data Streams の Amazon リソースネーム (ARN) 446
Kinesis Data Streams のポリシーの例 447
別のアカウントとのデータストリームを共有する 450
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 456
リソースベースのポリシーを使用したアクセスを共有する 456
Kinesis Data Streams のコンプライアンス検証 458
Kinesis Data Streams の耐障害性 459
Kinesis Data Streams の災害対策 460
Kinesis Data Streams のインフラストラクチャセキュリティ 461
Kinesis Data Streams のセキュリティのベストプラクティス 461
最小特権アクセスの実装 462
IAM ロールの使用 462
依存リソースでのサーバー側の暗号化を実装する 462
CloudTrail を使用して API コールをモニタリングする 462
AWS SDKs の使用 464
コードの例 466
基本 467
基本を学ぶ 467
アクション 471
サーバーレスサンプル 530
Kinesis トリガーから Lambda 関数を呼び出す 530
Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート 541
ドキュメント履歴 555

..... dlviii

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、および 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を構成しています。

AWS ビッグデータソリューションの詳細については、AWS でのビッグデータ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口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取り込まれてから取得されるまでの遅延 (入力から取得までの遅延) 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。集計実行中の DynamoDB テーブルは、その後、最新レポート用に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

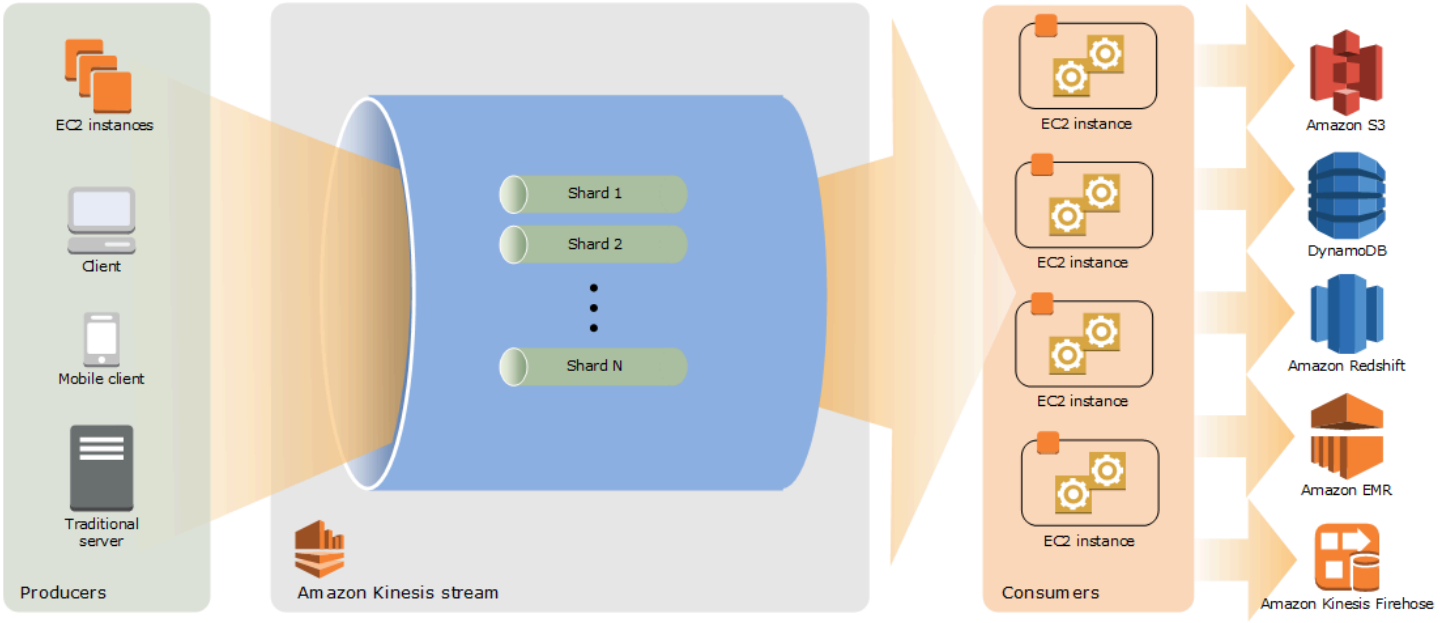
Amazon Kinesis Data Streams の使用を開始する前に、そのアーキテクチャと用語について説明します。

トピック

- Kinesis Data Streams の概要レベルのアーキテクチャを確認する
- Kinesis Data Streams の用語に精通する

Kinesis Data Streams の概要レベルのアーキテクチャを確認する

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 上で実行され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します。



Kinesis Data Streams の用語に精通する

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、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。詳細については、適切なストリームモードを選ぶ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、オンデマンド標準モードの機能とユースケース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、プロビジョンドモード機能とユースケース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。保持期間は、IncreaseStreamRetentionPeriod 操作を使用して最大 8,760時間 (365 日) まで延長し、DecreaseStreamRetentionPeriod 操作を使用して最小 24 時間まで短縮することができます。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 からデータを読み取る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、さまざまな他の AWS 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データ消費に関連するメタ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 3 つのテーブルを作成します。詳細については、Kinesis Client Library を使用する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に限

定される一意の名前が必要です。この名前は、Amazon DynamoDB では制御テーブルと 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 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) の料金が発生します。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

次の表では、Amazon Kinesis Data Streams のストリームおよびシャードのクォータと制限について説明します。

クォータ	オンデマンドモード	プロビジョンドモード
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、以下 AWS アカウント についてあたり 20,000 シャードです AWS リージョン。 • 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) その他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 1,000 または 6,000 シャードです。お使いのアカウントのシャードクォータと

クォータ	オンデマンドモード	プロビジョンドモード
		使用状況は、Service Quotas コンソール (https://console.aws.amazon.com/servicequotas/) で確認できます。 シャードクォータの引き上げをリクエストするには、Service Quotas コンソールまたは AWS CLI を使用します。詳細については、「Requesting a quota increase」(クォータ引き上げのリクエスト) を参照してください。

クォータ	オンデマンドモード	プロビジョンドモード
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。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) AWS リージョンでは、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 10 GB/秒の書き込みと 20 GB/秒の読み取りスループットまでスケールアップします。その他のリージョンでは、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、書き込みスループット 200 MB/秒、読み込みスループット 400 MB/秒までスケールアップします。これらのリージョンで書き込み容量 10 GB/秒、読み込み容量 20 GB/秒までの引き上げが必要な場合は、サポートチケットを送信してください。	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。取り込み容量を増やす必要がある場合は、 AWS マネジメントコンソール 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 10 MiB です。Kinesis は、バーストキャパシティを使用して、1 ~ 10 MiB の断続的な大きなレコードを処理できるように設計されています。	

クォータ	オンデマンドモード	プロビジョンドモード
GetRecords トランザクションのサイズ	GetRecords では、1 つのシャードから最大 10 MB のデータを取得でき、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、GetRecords を介して最大 2 MB/秒の合計データ読み取りレートをサポートします。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	Kinesis オンデマンドアドバンテージモードでは、最大 50 の登録済みコンシューマーを作成できます (拡張ファンアウト)。Kinesis オンデマンド標準モードと Kinesis プロビジョンドモードでは、データストリームごとに最大 20 の登録済みコンシューマー (拡張ファンアウト制限) を作成できます。	
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があります。リージョンごと、 AWS アカウントごとに以下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。 • 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

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
			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	
DescribeAccountSettings	5 TPS	アカウントあたり	
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	
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	

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
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。
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。
RemoveTagsFromStream	5 TPS	ストリームあたり	

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
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。
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、「UpdateShardCount」を参照してください。

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
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。
UpdateStreamWarmThroughput	5 TPS	アカウントあたり	設定できる最大のウォームスループットは、アカウントおよびリージョンにおけるオンデマンドモードのデータストリームスループットの上限となります。
UpdateAccountSettings	5 TPS	アカウントあたり	オンデマンドアドバンテージモードなどのアカウント設定を有効または無効にします。

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 リクエストがあまりに頻繁に行われた場合、ProvisionedThroughputExceededException が発生します。
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	Kinesis は、バーストキャパシティを使用して、1 ~ 10 MiB

API	API コールの制限	ペイロードの制限	その他の詳細
		ポートし、1 秒あたり 10 MiB の最大データ書き込みをサポートします。	の断続的な大きなレコードを処理できるように設計されています。
PutRecords		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 10 MiB、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 10 MiB です。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。	Kinesis は、バーストキャパシティを使用して、1 ~ 10 MiB の断続的な大きなレコードを処理できるように設計されています。
SubscribeToShard	シャードごと、登録済みコンシューマーごとに 1 秒あたり SubscribeToShard を 1 回呼び出すことができます。		呼び出しが成功してから 5 秒以内に同じ ConsumerARN と ShardId で SubscribeToShard を再度呼び出すと、ResourceInUseException が発生します。

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、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。Service Quotas 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 をセットアップするための前提条件を完了する

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環境を設定してください。

タスク

- AWS にサインアップする
- 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

AWS にサインアップする

Amazon Web Services (AWS) にサインアップすると、AWS アカウントが、Kinesis Data Streams を含む、AWS 内のすべてのサービスに自動的にサインアップされます。請求されるのは、使用したサービスの料金のみです。

AWS アカウントを既にお持ちの場合は、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、次に説明する手順にしたがってアカウントを作成してください。

AWS にサインアップして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、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

   AWS アカウント にサインアップすると、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。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

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 を使用してプロデューサーを開発する
  - AWS SDK for Java を使用してコンシューマーを開発する
  - Kinesis Data Streams を作成して管理する
- Go、Java、JavaScript、.NET、PHP、Python、および Ruby 向けの AWS SDK には、Kinesis Data Streams のサポートとサンプルが含まれています。AWS SDK for Java のお使いのバージョンに Kinesis Data Streamsのサンプルが含まれていない場合は、GitHub からダウンロードできます。
- Kinesis Client Library (KCL) には、データ処理用の使いやすいプログラミングモデルが用意されています。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 を使用すると、複数の AWS サービスをコマンドラインから制御したり、スクリプトで自動化したりできます。

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。

AWS CLI を使用して Amazon Kinesis Data Streams オペレーションを実行する

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。

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。

トピック

- チュートリアル: AWS CLI for Kinesis Data Streams をインストールして設定する
- チュートリアル: AWS CLI を使用した基本的な Kinesis Data Streams の操作を実行する

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。

チュートリアル: AWS CLI for Kinesis Data Streams をインストールして設定する

AWS CLI のインストール

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。

チュートリアル: AWS CLI を使用した基本的な Kinesis Data Streams の操作を実行する

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、使用する準備がまだ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。実行が成功すると、出力は次の例のようになります。

```
{
    "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ります。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ります。

次の get-records コマンドの例で以下の操作を行います。

```
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。

```
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。

```
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。

```
{
  "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ることを意味します。返されるレコード

は、ストリーム内のすべてのレコードを現在表すとは限りません。これは正常であり、本番コードは適切な間隔でストリームをポーリングしてレコードがないか調べます。このポーリング速度は、特定のアプリケーション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のレコードでは、データがゴミのように見えることがわかります。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。これはこのチュートリアルのために十分です。実際には、AWS CLI がデータの消費に使用されることはまれです。多くの場合、前述のように (describe-stream と list-streams)、ストリームの状態をモニタリングし、情報を取得するために使用します。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、常に指定されたストリーム/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 これは本稼働アプリケーションでの通常の状況ですが、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。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。

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

ステップ 4: クリーンアップする

ストリームを削除してリソースを解放し、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。clean-up コマンドは次のとおりです。

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

成功すると、出力は発生しません。削除の進行状況を確認するには、describe-stream を使用します。

```
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が表示されます。

```
{
    "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 のチュートリアルの開始方法

Amazon Kinesis Data Streams は、Kinesis データストリームからデータを取り込み、消費するためのさまざまなソリューションを提供します。このセクションのチュートリアルは、Amazon Kinesis Data Streams の概念と機能を理解し、ニーズを満たすソリューションを特定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 を使用して株式データをリアルタイム処理する
- チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する
- チュートリアル: Amazon 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
- Amazon Kinesis AWS のストリーミングデータソリューションを使用する

チュートリアル: KPL と KCL 2.x を使用して株式データをリアルタイム処理する

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基本的な 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**
>
> ストリームを作成すると、Kinesis Data Streams は AWS 無料利用枠の対象外であるため、Kinesis Data Streams の使用に対して アカウントにわずかな料金が発生します。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、リソースをクリーンアップ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

- 前提条件を満たす
- データストリームを作成する
- IAM ユーザーとポリシーを作成する
- コードをダウンロードしてビルドする
- プロデューサーを実装する
- コンシューマーを実装する
- (オプション) コンシューマーを拡張する
- リソースをクリーンアップする

前提条件を満たす

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 アカウントを作成して使用する

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、ガイドチュートリアル: AWS CLI for Kinesis Data Streams をインストールして設定するの手順を完了しておくと役立ちます。

にアクセスするには、 AWS アカウントとウェブブラウザが必要です AWS マネジメントコンソール。

コンソールアクセスの場合は、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 マネジメントコンソール にサインインします。プログラムによるアクセスや長期的な認証情報の代替方法など、 AWS セキュリティ認証情報の詳細については、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。へのサインインの詳細については AWS アカウント、AWS サインイン ユーザーガイドの「 へのサインイン方法 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を満たす

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。

最新バージョンの AWS SDK for Java が必要です。

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。

次の手順

データストリームを作成する

データストリームを作成する

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. にサインイン AWS マネジメントコンソール 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。

次の手順

IAM ユーザーとポリシーを作成する

IAM ユーザーとポリシーを作成する

のセキュリティのベストプラクティス AWS により、きめ細かなアクセス許可を使用してさまざまなリソースへのアクセスを制御できます。 AWS Identity and Access Management (IAM) を使用すると、 でユーザーとユーザーのアクセス許可を管理できます AWS。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

アクション	[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
   ```

   リージョン

   AWS リージョンコード。例: `us-west-2`。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 アカウント設定に示されているアカウント ID。

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 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 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 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。

JSON

```
{
    "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-2:111122223333:stream/StockTradeStream"
            ]
        },
        {
            "Sid": "Stmt234",
            "Effect": "Allow",
            "Action": [
```

```
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": [
                "arn:aws:kinesis:us-west-2:111122223333:stream/StockTradeStream/
*"
            ]
        },
        {
            "Sid": "Stmt456",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:111122223333: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. [Attach existing policies directly (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。

次の手順

コードをダウンロードしてビルドする

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 “プロデューサーを実装する”) に進みます。

次の手順

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、株式ティッカー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 にデータを書き込む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、再試行を使用するなど、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 にデータを書き込むを参照してください。

プロデューサーを実行するには

1. IAM ユーザーとポリシーを作成する 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。

次の手順

コンシューマーを実装する

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、KCL 1.x および 2.x の情報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. reportStats メソッドを実装します。出力形式は必要に応じて変更できます。

```
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.
****************************************************************
```

次の手順

(オプション) コンシューマーを拡張する

(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.
************************************************************
```

次の手順

リソースをクリーンアップする

リソースをクリーンアップする

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。

概要

ほぼリアルタイムで大量のデータを処理するために、複雑な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、基本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。Amazon EC2 インスタンスでワーカーを実行し、Amazon EC2 Auto Scaling グループを使用できます。

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

ストリームが継続的に処理されると、その出力を他の送信先に送信できます。 は、Kinesis Data Streams を他の AWS サービスやサードパーティーのツールと統合するためのコネクタ AWS を提供します。

チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する

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

ストリームを作成すると、Kinesis Data Streams は AWS 無料利用枠の対象外であるため、Kinesis Data Streams の使用に対して アカウントにわずかな料金が発生します。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、リソースをクリーンアップ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

- 前提条件を満たす
- データストリームを作成する
- IAM ユーザーとポリシーを作成する
- 実装コードをダウンロードおよびビルドする
- プロデューサーを実装する
- コンシューマーを実装する
- (オプション) コンシューマーを拡張する
- リソースをクリーンアップする

前提条件を満たす

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

Amazon Web Services アカウントを作成して使用する

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、チュートリアル: AWS CLI for Kinesis Data Streams をインストールして設定するを完了していると役立ちます。

にアクセスするには、 AWS アカウントとウェブブラウザが必要です AWS マネジメントコンソール。

コンソールアクセスの場合は、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 マネジメントコンソール にサインインします。プログラムによるアクセスや長期的な認証情報の代替方法など、 AWS セキュリティ認証情報の詳細については、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。へのサインインの詳細については AWS アカウント、AWS サインイン ユーザーガイドの「 へのサインイン方法 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を満たす

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

データストリームを作成する

データストリームを作成する

チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する 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. にサインイン AWS マネジメントコンソール 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、スループットキャパシティの単位です。[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、Kinesis Data Streams を作成して管理するを参照してください。

次の手順

IAM ユーザーとポリシーを作成する

IAM ユーザーとポリシーを作成する

のセキュリティのベストプラクティス AWS により、きめ細かなアクセス許可を使用してさまざまなリソースへのアクセスを制御できます。 AWS Identity and Access Management (IAM) を使用すると、 でユーザーとユーザーのアクセス許可を管理できます AWS。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

アクション	[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
   ```

   リージョン

   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 アカウント設定に示されているアカウント ID。

   名前

   データストリームを作成する 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 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 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

JSON

```
{
    "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-2:111122223333:stream/StockTradeStream"
            ]
        },
        {
            "Sid": "Stmt234",
            "Effect": "Allow",
            "Action": [
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": [
                "arn:aws:kinesis:us-west-2:111122223333:stream/StockTradeStream/
*"
            ]
        },
        {
            "Sid": "Stmt456",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:111122223333: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. [Attach existing policies directly (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

実装コードをダウンロードおよびビルドする

実装コードをダウンロードおよびビルドする

スケルトンコードは the section called “チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する” 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 “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。

次の手順

プロデューサーを実装する

チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する 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、ストリームにデータを追加する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、単純な再試行を使用するなど、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、ストリームにデータを追加する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。

次の手順

コンシューマーを実装する

コンシューマーを実装する

チュートリアル: KPL と KCL 1.x を使用して株式データをリアルタイム処理する 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 コンシューマーを開発する 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。
- `RecordProcessorFactory` 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 コンシューマーを開発する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いう用語は、ストリーム内のポイントを、これまで消費および処理されたデータレコードまで記録することを意味します。アプリケーションがクラッシュすると、ストリームはストリームの先頭からではなく、その時点から読み取られます。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な拡張方法です。

次の手順

[(オプション) コンシューマーを拡張する]

(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 を使用して株式データをリアルタイム処理する 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.
 ****************************************************************
```

次の手順

リソースをクリーンアップする

リソースをクリーンアップする

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。

概要

ほぼリアルタイムで大量のデータを処理するために、複雑な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、基本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。Amazon EC2 インスタンスでワーカーを実行し、Amazon EC2 Auto Scaling グループを使用できます。

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

ストリームが継続的に処理されると、その出力を他の送信先に送信できます。 は、Kinesis Data Streams を他の AWS サービスやサードパーティーのツールと統合するためのコネクタ AWS を提供します。

次の手順

- 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 を使用してプロデューサーを開発する、AWS SDK for Java を

使用して共有スループットコンシューマーを開発する、および Kinesis Data Streams を作成して管理する を参照してください。

- Kinesis Client Library の詳細については、「KCL 1.x コンシューマーを開発する」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、Amazon Kinesis Data Streams コンシューマーを最適化するを参照してください。

チュートリアル: Amazon 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。

Amazon Managed Service for Apache Flink を使用すると、Java、または Scala のいずれかを使用してストリーミングデータを処理および分析できます。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アプリケーションを作成すると、アカウントは AWS 無料利用枠の対象とならないため、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。このアプリケーションが終了したら、 AWS リソースを削除して料金の発生を停止します。

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
- AWS アカウントをセットアップし、管理者ユーザーを作成する
- AWS Command Line Interface (AWS CLI) のセットアップ
- Apache Flink アプリケーション用 Managed Service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、AWS アカウントをセットアップし、管理者ユーザーを作成するに進みます。

AWS アカウントをセットアップし、管理者ユーザーを作成する

Amazon Managed Service for Apache Flink を初めて使用する前に、以下のタスクを完了してください：

1. にサインアップする AWS

2. IAM ユーザーの作成

にサインアップする AWS

Amazon Web Services (AWS) にサインアップすると、Amazon Managed Service for Apache Flink を含む AWSのすべてのサービスに AWS アカウントが自動的にサインアップされます。請求されるのは、使用したサービスの料金のみです。

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。

すでに AWS 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、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

   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。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

次のタスクで必要になるため、 AWS アカウント ID を書き留めます。

IAM ユーザーの作成

Amazon Managed Service for Apache Flink AWSなどの のサービスでは、アクセス時に認証情報を指定する必要があります。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 マネジメントコンソール では、パスワードを入力する必要があります。

AWS Command Line Interface (AWS CLI) または API にアクセスするための AWS アカウントのアクセスキーを作成できます。ただし、 AWS アカウントの認証情報 AWS を使用して にアクセスするこ

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。

にサインアップしたが AWS、自分で IAM ユーザーを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. にサインイン AWS マネジメントコンソール し、https://console.aws.amazon.com/iam/ で 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. [Next: Permissions] (次のステップ: 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 (次へ: レビュー)] を選択します。
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 マネジメントコンソール。
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、ハイフンのない AWS アカウント ID です。たとえば、 AWS アカウント ID が 1234-5678-9012 の場合、*aws_account_number* を に置き換えます**123456789012**。アカウント番号を検索する方法については、IAM ユーザーガイドの「AWS アカウント ID とそのエイリアス」を参照してください。
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 に AWS アカウント ID を含めない場合は、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、 AWS アカウント ID の使用に戻ります。

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 の開始方法
- IAM ユーザーガイド

次のステップ

AWS Command Line Interface (AWS CLI) のセットアップ

AWS Command Line Interface (AWS CLI) のセットアップ

このステップでは、Amazon Managed Service for Apache Flink で使用する をダウンロードして設定 AWS CLI します。

> **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**
>
> が既に AWS CLI インストールされている場合は、アップグレードして最新の機能を取得する必要がある場合があります。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イスのインストール」を参照してください。のバージョンを確認するには AWS CLI、次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では、次の AWS CLI 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 設定ファイルに管理者ユーザーの名前付きプロファイルを追加します。 AWS CLI コマンドを実行するときは、このプロファイルを使用します。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
   ```

   使用可能な AWS リージョンのリストについては、AWS 『』の「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 をセットアップしたら AWS CLI、次の演習を試すことができます。この演習では、サンプルアプリケーションを設定し、end-to-endのセットアップをテストします。

次のステップ

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

Apache Flink アプリケーション用 Managed Service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、Managed Service for Apache Flink アプリケーションを作成します。

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

この演習でAmazon 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) を作成する必要があります。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。ファイルには、Amazon Managed Service for Apache Flink 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] を選択します。
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] を選択します。
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] を選択します。
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

コンソールまたは AWS CLIのいずれかを使用してManaged Service for Apache Flink を作成し、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ときは AWS CLI、これらのリソースを個別に作成します。

トピック

- アプリケーションを作成して実行する (コンソール)
- アプリケーションを作成して実行する (AWS CLI)

アプリケーションを作成して実行する (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. [アプリケーションを作成] を選択します。

> **Note**
>
> コンソールを使用して Amazon Managed Service for Apache Flink を作成する場合は、IAM ロールとポリシーをアプリケーションが自動的に作成するオプションを選択できます。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 ポリシーを編集する

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。

   JSON

```
{
    "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"
        }
    ]
}
```

アプリケーションを設定する

1. [MyApplication] ページで、[Congirue] 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 logging] では、[Enable] チェックボックスをオンにします。

8. [更新] を選択します。

> **Note**
>
> CloudWatch ログ記録の有効化を選択すると、Managed Service for Apache Flink がユーザーに代わってロググループとログストリームを作成します。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、[Run] 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。

アプリケーションを停止する

[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、[Congirue] 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。

アプリケーションを作成して実行する (AWS CLI)

このセクションでは、 を使用して Managed Service for Apache Flink アプリケーション AWS CLI を作成して実行します。Managed Service for Apache Flink は、 `kinesisanalyticsv2` AWS CLI コマンドを使用して Managed Service for Apache Flink アプリケーションを作成して操作します。

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。

JSON

```
{
    "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"
        }
    ]
}
```

許可ポリシーを作成する詳しい手順については、IAM ユーザーガイドのチュートリアル: はじめてのカスタマーマネージドポリシーの作成とアタッチを参照してください。

> **Note**
>
> 他の AWS 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、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想定できる IAM ロールを作成します。

Apache Flink 用 Managed Service は、許可なしにはストリームにアクセスできません。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。

   [Next: Permissions] (次のステップ: 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。[ロールの作成] を選択してください。

   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. [KAReadInputStreamWriteOutputStream] ポリシーを選択し、[ポリシーのアタッチ] 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。

ロールを作成する手順については、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 Managed Service for Apache Flink メトリクスをチェックして、アプリケーションが機能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。この例では、カスタムアプリケーションは Kinesis データストリームにレコードを書き込みます。AWS 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。AWS Lambda は、Lambda 関数の作成時に指定した実行ロールを引き受けて 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 AWS 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。

Amazon Kinesis AWS のストリーミングデータソリューションを使用する

Amazon Kinesis AWS 用ストリーミングデータソリューションは、ストリーミングデータを簡単にキャプチャ、保存、処理、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、Amazon API Gateway AWS Lambda、Amazon Managed Service for Apache Flink など、複数の AWS サービスを使用するストリーミングデータのユースケースを解決するための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 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

Kinesis Data Streams を作成して管理する

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。

このセクションでは、ストリームのキャパシティモードを設定する方法と、 AWS マネジメントコンソール または APIs を使用してストリームを作成する方法について説明します。その後、ストリームに対して追加のアクションを実行できます。

トピック

- 適切なストリームモードを選ぶ
- を使用してストリームを作成する AWS マネジメントコンソール
- API を使用してストリームを作成する
- ストリームを更新する
- ストリームの一覧表示
- シャードを一覧表示する
- ストリームを削除する
- ストリームをリシャーディングする
- データ保持期間を変更する
- Amazon Kinesis Data Streams リソースにタグを付ける
- 大きなレコードの処理
- でレジリエンステストを実行する AWS Fault Injection Service

適切なストリームモードを選ぶ

以下のトピックでは、アプリケーションに適したモードを選択する方法と、必要に応じてモードを切り替える方法について説明します。

トピック

- Kinesis Data Streams の各種モードについて
- オンデマンド標準モードの機能とユースケース
- オンデマンドアドバンテージモードの機能とユースケース
- プロビジョンドモード機能とユースケース
- モードの切り替え

Kinesis Data Streams の各種モードについて

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モードとしてオンデマンド標準、オンデマンドアドバンテージ、プロビジョンドを選択できます。

- オンデマンド標準 - 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スループットおよび読み込み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ルされます。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。
- オンデマンドアドバンテージ - オンデマンドストリームに対してより多くの機能を利用でき、かつ単純化された料金体系を提供するアカウントレベルのモードです。このモードでは、ストリームの書き込みスループットキャパシティを、任意のタイミングで事前にウォームアップできます。料金については、ストリーム単位の固定料金がなくなり、すべてのオンデマンドストリームにおけるデータインジェスト、データ取得、および拡張保持の使用量に対する料金が、オンデマンド標準の料金と比べて少なくとも 60% 低くなります。
- 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 の `PutRecord` および `PutRecords` API を使用して、任意の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、3 つのモードはすべ

て、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。

オンデマンド標準モードの機能とユースケース

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スループットおよび読み込み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。

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。

オンデマンドアドバンテージモードの機能とユースケース

オンデマンドアドバンテージは、リージョン内のすべてのオンデマンドストリームに追加機能を提供し、異なる料金体系を適用するアカウントレベルの設定です。このモードでは、オンデマンドス

トリームは引き続き既存の機能を保持し、実際のデータ使用量に基づいて自動的に容量をスケールします。ストリームの書き込みスループットキャパシティを事前にウォームアップしたい場合は、ウォームスループットを設定できます。たとえば、データストリームの書き込みスループットが 10 MB/秒〜40 MB/秒の範囲であれば、スロットリングなしで最大 80 MB/秒の瞬間的なスループット増加に対応できます。一方で、今後 200 MB/秒程度のピークが予測される場合は、ウォームスループットを 200 MB/秒に設定しておくことで、データ流入時に必要な容量を確保できます。ウォームスループットを使用しても追加料金は発生しません。

オンデマンドアドバンテージモードのもう一つの利点は、オンデマンドストリームがよりシンプルな料金体系へ移行することです。このモードを有効にすると、アカウントにはストリーム単位の固定料金が発生しなくなり、データインジェスト、データ取得、およびオプションの長期保持料金のみが課金対象となります。また、それぞれの料金項目は、オンデマンド標準と比べて大幅に割引された価格が適用されます。詳細については、「Amazon Kinesis Data Streams の料金表」を参照してください。

このモードでは、拡張ファンアウトによるデータ取得も標準のデータ取得と比べて追加料金は発生しません。さらに、オンデマンドアドバンテージモードでは、ストリームごとに最大 50 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。オンデマンドアドバンテージを有効化すると、アカウントはオンデマンドストリーム全体で、少なくとも 25 MiB/秒のデータインジェストと 25 MiB/秒のデータ取得を利用することが前提となります。最小利用要件を満たしているアカウントについて、Kinesis Data Streams コンソールでは、アカウントの利用パターンがオンデマンドアドバンテージモードに適しているかどうかを確認できる機能が用意されています。

アカウントのデータ使用量が要件を下回る場合は、その不足分について料金が発生しますが、同じ割引率が適用されます。オンデマンドアドバンテージを有効化した場合、無効化できるのは 24 時間経過後となる最低利用期間があります。総合的に見て、オンデマンドアドバンテージは、最低コミットメントに近い、あるいはそれを上回るスループットを継続的に使用する場合、多数の拡張ファンアウトコンシューマーが必要な場合、または数百単位のデータストリームを運用する場合に、Kinesis Data Streams を利用するための最適なモードです。

プロビジョンドモード機能とユースケース

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 マネジメントコンソール 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 = ceiling(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。

モードの切り替え

内のデータストリームごとに AWS アカウント、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。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。

アカウントレベルの設定を有効化することで、オンデマンド標準モードからオンデマンドアドバンテージモードに切り替えることができます。この設定を有効化すると、アカウントはリージョン内のすべてのオンデマンドストリームに対して、データインジェスト 25 MiB/秒、データ取得 25 MiB/秒の最低使用量をコミットすることになります。一度オンデマンドアドバンテージを有効化すると、無効化できるようになるまで最低 24 時間待つ必要がありますが、切り替えリクエスト自体はいつでも行えます。オンデマンドアドバンテージからオンデマンドスタンダードに戻したい場合は、まずオンデマンドストリームに設定されているウォームスループットをすべて削除する必要があります。

を使用してストリームを作成する AWS マネジメントコンソール

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 マネジメントコンソール 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、適切なストリームモードを選ぶ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 を使用してストリームを作成する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 を使用してストリームを作成する

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、コンソールまたはプログラムでストリームを作成できます。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、ストリームの名前を指定します。プロビジョンドモードを使用する場合は、使用するデータストリームのシャード数を指定する必要があります。

- 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。名前は、アプリケーションで使用される 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 つのストリームを持つことはできません。

ストリームのスループットはシャードの数の関数です。プロビジョニングされたスループットを向上させるには、より多くのシャードが必要です。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、適切なストリームモードを選ぶ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" );
}
```

ストリームを更新する

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。

コンソールを使用する

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、「適切なストリームモードを選ぶ」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 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 “CloudWatch による Kinesis Data Streams サービスを監視する”」を参照してください。

API を使用する

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

を使用する 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。

ストリームの一覧表示

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 アカウントを使用して多数のストリームを 1 度にアクティブにできます。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 を使用してストリームを作成する メソッドを使用して、ストリームの状態を確認できます。

シャードを一覧表示する

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のに推奨される方法は、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。この例で使用しているメインオペレーションと、このオペレーションで設定できるすべてのパラメータの詳細については、ListShardsを参照してください。

```
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 パラメーターを使用して API の応答をフィルターで除外できます。一度に 1 つのフィルターしか指定できません。

ListShards API を呼び出すときに 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、機能的には ListShards API の ExclusiveStartShardId

パラメーターと同じです。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。したがって、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作成するにつれてアクションのスケーリングが向上します。
DescribeStream 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 DescribeStreamSummary および ListShard API 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 および KCL のユーザーエージェントが API コールで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
```

API を呼び出す AWS Lambda および Amazon Firehose と Kinesis Data Streams の統合DescribeStream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。

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、ストリームのシャードを組み直すことができます。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 CloudWatch が提供するメトリクス、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

- リシャーディング戦略を決定する
- シャードを分割する
- 2 つのシャードを結合する
- リシャーディングアクションを完了する

リシャーディング戦略を決定する

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 が発行する 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。

シャードを分割する

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、シャードの状態を考慮する 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。

リシャーディングアクションを完了する

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、シャードの状態を考慮する

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、シャードの状態を考慮する

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。アプリケーションは、データがストリーム内のシャードを継続的に流れることを前提とする必要があります。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。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、データ保持期間を変更する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、データ保持期間を変更する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、リシャーディング後に残った親シャードには、リシャーディング前にストリームに追加されてまだ読み取られていないデータがそのまま格納されている可能性があります。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

データ保持期間を変更する

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 オペレーションで更新できます。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) まで増やすことができます。また、DecreaseStreamRetentionPeriod オペレーションまたは Kinesis Data Streams コンソールを使用して、保持期間を最短 24 時間まで減らすことができます。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。そのため、DecreaseStreamRetentionPeriod オペレーションを呼び出すときは、細心の注意が必要です。

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データ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 リソースにタグを付ける

Amazon Kinesis Data Streams で作成したストリームおよび拡張ファンアウトコンシューマーに、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。タグを使用することは、 AWS リソースを管理し、請求データを含むデータを整理するためのシンプルで強力な方法です。

内容

- タグの基本を確認する
- タグ付けを使用したコストを追跡する
- タグの制限を理解する
- Kinesis Data Streams コンソールを使用してストリームにタグ付けする
- を使用してストリームにタグを付ける AWS CLI
- Kinesis Data Streams API を使用したストリームにタグを付ける
- を使用してコンシューマーにタグを付ける AWS CLI
- Kinesis Data Streams API を使用してコンシューマーにタグ付けする

タグの基本を確認する

タグ付けできる Kinesis Data Streams リソースには、データストリームと拡張ファンアウトコンシューマーが含まれます。Kinesis Data Streams コンソール AWS CLI、または Kinesis Data Streams API を使用して、次のタスクを完了します。

- タグの付いたリソースを作成する
- リソースにタグを追加する
- リソースのタグを一覧表示する
- リソースからタグを削除する

> **Note**
>
> Kinesis Data Streams コンソールでは、拡張ファンアウトコンシューマーにタグを適用することはできません。コンシューマーにタグを適用するには、 AWS CLI または Kinesis Data Streams API を使用します。

タグを使用して リソースを分類できます。例えば、目的、所有者、環境などに基づいてリソースを分類できます。タグごとにキーと値を定義するため、特定のニーズに合わせて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リソース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

> **Important**
>
> - ストリームの作成中にタグを追加するには、そのストリームに対して `kinesis:CreateStream` と `kinesis:AddTagsToStream` の権限を付与する必要があります。ストリーム作成時に `kinesis:TagResource` 権限を使用してタグ付けすることはできません。
> - コンシューマー登録中にタグを追加するには、`kinesis:TagResource` と`kinesis:RegisterStreamConsumer` の権限を付与する必要があります。

タグ付けを使用したコストを追跡する

タグを使用して、 AWS コストを分類および追跡できます。Kinesis Data Streams リソース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量とコストが含まれます。コストセンター、アプリケーション名、所有者などの自社のカテゴリ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。

タグの制限を理解する

タグには以下の制限があります。

基本制限

- 1 つのリソースに付けることができるタグの最大数は 50 です。
- タグキーと値は大文字と小文字が区別されます。
- 削除されたリソース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持つタグを追加すると、新しいタグによって既存のキーと値のペアが上書きされます。
- このプレフィックスはユーザーに代わってaws:このプレフィックスで始まるタグ AWS AWS を作成するため、 でタグキーを開始することはできませんが、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てストリームにタグ付けする

Kinesis Data Streams コンソールを使用して、ストリームにタグを追加、更新、一覧表示、および削除できます。

ストリームのタグを表示するには

1. にサインイン AWS マネジメントコンソール し、https://console.aws.amazon.com/kinesis で Kinesis コンソールを開きます。

2. 左のナビゲーションペインの [データストリーム] を選択します。
3. [データストリーム] ページで、タグ付けするストリームを選択します。
4. [ストリームの詳細] ページで、[設定] タブを選択します。
5. [タグ] セクションで、ストリームに適用されているタグを表示します。

タグを使用してデータストリームを作成するには

1. Kinesis Data Streams コンソールを開きます。
2. 左のナビゲーションペインの [データストリーム] 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. [データストリームの作成] ページで、データストリームの名前を入力します。
5. [データストリームの容量] には、[オンデマンド] または [プロビジョンド] のキャパシティモードを選択します。

   キャパシティモードの詳細については、適切なストリームモードを選ぶ を参照してください。
6. [タグ] セクションで、次の操作を行います。

   a. [新しいタグを追加] をクリックします。
   b. [キー] フィールドでタグを入力し、オプションとして [値] フィールドで値を指定します。

      エラーが表示された場合は、指定したタグキーまたは値がタグ制限を満たしていません。詳細については、「タグの制限を理解する」を参照してください。
7. [データストリームの作成] を選択します。

ストリームにタグを追加または更新するには

1. Kinesis Data Streams コンソールを開きます。
2. 左のナビゲーションペインの [データストリーム] を選択します。
3. [データストリーム] ページで、タグを追加または更新するストリームを選択します。
4. [ストリームの詳細] ページで、[設定] タブを選択します。
5. [タグ] セクションで、[タグを管理] を選択します。
6. [タグ] で、次のいずれかを実行します。

   - 新しいタグを追加するには、[新しいタグを追加] を選択し、タグの [キー] と [値] を入力します。このステップを必要な回数繰り返します。

ストリームあたり追加できるタグの最大数は 50 です。

- 既存のタグを更新するには、そのタグの [キー] の [値] フィールドに新しいタグ値を入力します。

エラーが表示された場合は、指定したタグキーまたは値がタグ制限を満たしていません。詳細については、「タグの制限を理解する」を参照してください。

7. [Save changes] (変更の保存) 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。
2. 左のナビゲーションペインの [データストリーム] を選択します。
3. [データストリーム] ページで、タグを削除するストリームを選択します。
4. [ストリームの詳細] ページで、[設定] タブを選択します。
5. [タグ] セクションで、[タグを管理] を選択します。
6. 削除するタグの [キー] と [値] のペアを検出します。次に、[削除] を選択します。
7. [Save changes] (変更の保存) をクリックします。

を使用してストリームにタグを付ける AWS CLI

AWS CLIを使用して、ストリームにタグの追加、一覧表示、および削除できます 例については、次のドキュメントを参照してください。

create-stream

タグ付きのストリームを作成します。

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にタグを付ける

Kinesis Data Streams API を使用して、ストリームにタグを追加、一覧表示、および削除できます。例については、次のドキュメントを参照してください。

CreateStream

タグ付きのストリームを作成します。

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。

を使用してコンシューマーにタグを付ける AWS CLI

AWS CLIを使用して、コンシューマーにタグを追加、一覧表示、および削除できます。例については、次のドキュメントを参照してください。

register-stream-consumer

タグを使用して Kinesis データストリームにコンシューマーを登録します。

tag-resource

指定された Kinesis リソースのタグを追加または更新します。

list-tags-for-resource

指定された Kinesis リソースのタグを一覧表示します。

untag-resource

指定された Kinesis リソースからタグを削除します。

Kinesis Data Streams API を使用してコンシューマーにタグ付けする

Kinesis Data Streams API を使用してコンシューマーにタグを追加、一覧表示、および削除できます。例については、次のドキュメントを参照してください。

RegisterStreamConsumer

タグを使用して Kinesis データストリームにコンシューマーを登録します。

TagResource

指定された Kinesis リソースのタグを追加または更新します。

ListTagsForResource

指定された Kinesis リソースのタグを一覧表示します。

UntagResource

指定された Kinesis リソースからタグを削除します。

大きなレコードの処理

Amazon Kinesis Data Streams は、最大 10 メビバイト (MiB) のレコードをサポートします。この機能は、デフォルトの 1 MiB のレコードサイズ制限を超える断続的なデータペイロードを処理する場合に推奨されます。既存および新規に作成されたストリームのデフォルトの最大レコードサイズは 1 MiB に設定されています。

この機能は、断続的に大きなデータペイロードを処理する必要がある IoT アプリケーション、変更データキャプチャ (CDC) パイプライン、機械学習ワークフローに有用です。ストリームで大きなレコードの使用を開始するには、ストリームの最大レコードサイズ制限を更新します。

> **Important**
>
> 1 シャードあたりのスループット制限 (書き込み 1 MB/秒、読み取り 2 MB/秒) は、大きなレコードサイズをサポートしても変更されません。Kinesis Data Streams は、1 MiB 以下のレコードをベースとしたトラフィックに加えて、断続的に発生する大きなレコードを処理できるように設計されています。継続的に大量の大きなレコードを取り込む用途向けには設計されていません。

大きなレコードを使用するようにストリームを更新する

Kinesis Data Streams で大きなレコードを処理するには、次の手順を実行します。

1. Kinesis Data Streams コンソールに移動します。
2. ストリームを選択し、[設定] タブを開きます。
3. [最大レコードサイズ] の横にある [編集] をクリックします。
4. 最大レコードサイズ (最大 10 MiB) を設定します。
5. 変更内容を保存します。

この設定は、この Kinesis データストリームの最大レコードサイズのみを調整します。この上限を引き上げる前に、すべての下流アプリケーションが大きなレコードを処理できることを確認してください。

AWS CLI を使用してこの設定を更新することもできます。

```
aws kinesis update-max-record-size \ --stream-arn  \
        --max-record-size-in-ki-b 5000
```

大きなレコードでストリームのパフォーマンスを最適化する

大きなレコードは、全体のトラフィックの 2% 未満に維持することをお勧めします。1 つのストリーム内で、各シャードのスループットキャパシティは毎秒 1 MiB です。Kinesis Data Streams は、大きなレコードに対応するため、一時的に最大 10 MiB までバーストできますが、平均では毎秒 1 MiB に抑えられます。この大きなレコードをサポートするための容量は、ストリーム内で継続的に補充されます。補充の速度は、大きなレコードのサイズとベースラインレコードのサイズによって異なります。最良の結果を得るには、均一に分散されたパーティションキーを使用します。Kinesis のオンデマンドスケーリングの仕組みについて詳しくは、「オンデマンド標準モードの機能とユースケース」を参照してください。

大きなレコードによるスロットリングを軽減する

スロットリングを軽減するには

1. プロデューサーアプリケーションに指数バックオフを伴う再試行ロジックを実装します。
2. 利用可能なシャード全体に大きなレコードを分散させるために、ランダム化されたパーティションキーを使用します。

3. 継続的に大きなレコードをストリーミングする場合は、ペイロードを Amazon S3 に保存し、メタデータ参照のみをストリームに送信します。詳細については、Processing large records with Amazon Kinesis Data Streams を参照してください。

Kinesis Data Streams API を使用して大きなレコードを処理する

大きなレコードのサポートでは、新しい API が 1 つ追加され、既存の 2 つのコントロールプレーン API が更新され、最大 10 MiB までのレコードを処理できるようになりました。

レコードサイズを変更するための API:

- `UpdateMaxRecordSize`: 既存のストリームに対して最大 10 MiB までのレコードサイズ上限を設定します。

既存 API の更新点:

- `CreateStream`: ストリーム作成時にレコードサイズ上限を設定できるオプションパラメータ `MaxRecordSizeInKiB` が追加されました。
- `DescribeStreamSummary`: 現在のストリーム設定を表示するための `MaxRecordSizeInKiB` フィールドが返されます。

ここに挙げたすべての API は、既存のストリームとの下位互換性を維持しています。API ドキュメントの詳細については、Amazon Kinesis Data Streams Service API リファレンスを参照してください。

AWS 大きなレコードと互換性のあるコンポーネント

次の AWS コンポーネントは、大きなレコードと互換性があります。

コンポーネント	説明
AWS SDK	AWS SDK は、大規模なレコードの処理をサポートしています。 AWS SDKs で使用可能なメソッドを使用して、ストリームの最大レコードサイズを最大 10 MiB に更新できます。詳細については、AWS 「 SDK でこのサービスを使用する」を参照してください。

コンポーネント	説明
Kinesis Consumer Library (KCL)	バージョン 2.x 以降、KCL は大きなレコードの処理をサポートしています。大きなレコードのサポートを使用するには、ストリームの `maxRecordSize` を更新し、KCL を使用します。詳細については、Use Kinesis Client Library を参照してください。
Kinesis Producer Library (KPL)	バージョン 1.0.5 以降、KPL は大きなレコードの処理をサポートしています。大きなレコードのサポートを使用するには、ストリームの `maxRecordSize` を更新し、KPL を使用します。詳細については、Develop producers using the Amazon Kinesis Producer Library (KPL) を参照してください。
Amazon EMR	Apache Spark を使用した Amazon EMR は、Kinesis Data Streams の上限 (10 MiB) までの大きなレコードの処理をサポートしています。大きなレコードのサポートを使用するには、`readStream` 関数を使用します。詳細については、Amazon EMR and Amazon Kinesis integration を参照してください。

コンポーネント	説明
Amazon Data Firehose	Kinesis Data Streams と併用する場合、Amazon Data Firehose の大きなレコードの動作は配信先によって異なります。 • Amazon S3: 追加設定なしで大きなレコードの配信がサポートされます。大きなレコードの配信は、データ形式変換を使用する場合には、Firehose でサポートされますが、動的パーティショニングを使用する場合、Firehose ではサポートされません。 • Lambda: Firehose が下流で Lambda 関数をトリガーする場合、大きなレコードの使用は推奨されません。これにより、断続的な障害が発生する可能性があります。 • HTTP: Firehose では大きなレコードの配信はサポートされていません。 • Snowflake: Firehose では大きなレコードの配信はサポートされていません。 • Amazon Redshift: Firehose では大きなレコードの配信はサポートされていません。 大きなレコードを Snowflake または Redshift に配信する必要がある場合は、まず Amazon S3 にデータを配信します。その後、抽出、変換、ロード (ETL) プロセスを使用してデータをロードします。その他の配信先については、本番環境にスケールする前に概念実証 (PoC) 環境で大きなレコードの動作を確認してください。大きなレコードの処理方法は配信先によって異なります。

コンポーネント	説明
AWS Lambda	AWS Lambda は最大 6 MiBs のペイロードをサポートします。この上限には、base64 エンコードに変換された Kinesis ペイロードと、イベントソースマッピング (ESM) に関連付けられたメタデータが含まれます。6 MiB 未満のレコードは、追加設定なしで ESM を使用して処理されます。6 MiB を超えるレコードは、Lambda が失敗時の送信先を使用して処理します。処理上限を超えるレコードを扱うには、ESM を使用して失敗時の送信先を設定する必要があります。失敗時の送信先に送信される各イベントは、失敗した呼び出しに関するメタデータを含む JSON ドキュメントです。 レコードサイズにかかわらず、ESM 内で失敗時の送信先を作成することをお勧めします。これにより、いかなるレコードも破棄されないことが保証されます。詳細については、「失敗した呼び出しの送信先の設定」を参照してください。
Amazon Redshift	Amazon Redshift は、Kinesis Data Streams からデータをストリーミングする場合、1 MiB 未満のレコードサイズのみをサポートしています。この上限を超えるレコードは処理されません。処理されなかったレコードは `sys_stream_scan_errors` としてログに記録されます。詳細については、SYS_STREAM_SCAN_ERRORS を参照してください。

コンポーネント	説明
Flink connector for Kinesis Data Streams	Kinesis Data Streams からデータを取り込む方法には、Kinesis ソースコネクタ と Kinesis シンクコネクタ の 2 つのアプローチがあります。ソースコネクタは 1 MiB 未満から最大 10 MiB までのレコードの処理をサポートしています。1 MiB を超えるレコードにはシンクコネクタを使用しないでください。詳細については、Use connectors to move data in Amazon Managed Service for Apache Flink with the DataStream API を参照してください。

大きなレコードがサポートされるリージョン

この Amazon Kinesis Data Streams 機能は、次の AWS リージョンでのみ使用できます。

AWS リージョン	リージョン名
eu-north-1	欧州 (ストックホルム)
me-south-1	中東 (バーレーン)
ap-south-1	アジアパシフィック (ムンバイ)
eu-west-3	欧州 (パリ)
ap-southeast-3	アジアパシフィック (ジャカルタ)
us-east-2	米国東部 (オハイオ)
af-south-1	アフリカ (ケープタウン)
eu-west-1	欧州 (アイルランド)
me-central-1	中東 (アラブ首長国連邦)
eu-central-1	欧州 (フランクフルト)

AWS リージョン	リージョン名
sa-east-1	南米 (サンパウロ)
ap-east-1	アジアパシフィック (香港)
ap-south-2	アジアパシフィック (ハイデラバード)
us-east-1	米国東部 (バージニア北部)
ap-northeast-2	アジアパシフィック (ソウル)
ap-northeast-3	アジアパシフィック (大阪)
eu-west-2	欧州 (ロンドン)
ap-southeast-4	アジアパシフィック (メルボルン)
ap-northeast-1	アジアパシフィック (東京)
us-west-2	米国西部 (オレゴン)
us-west-1	米国西部 (北カリフォルニア)
ap-southeast-1	アジアパシフィック (シンガポール)
ap-southeast-2	アジアパシフィック (シドニー)
il-central-1	イスラエル (テルアビブ)
ca-central-1	カナダ (中部)
ca-west-1	カナダ西部 (カルガリー)
eu-south-2	欧州 (スペイン)
cn-northwest-1	中国 (寧夏)
eu-central-2	欧州 (チューリッヒ)
us-gov-east-1	AWS GovCloud (米国東部)

AWS リージョン	リージョン名
us-gov-west-1	AWS GovCloud (米国西部)

でレジリエンステストを実行する AWS Fault Injection Service

AWS Fault Injection Service は、 AWS ワークロードでフォールトインジェクション実験を実行するのに役立つフルマネージドサービスです。Amazon Kinesis Data Streams と AWS FIS の統合により、制御された環境で一般的な Amazon Kinesis Data Streams API エラーに対するアプリケーションの耐障害性をテストできます。この機能により、障害が発生する前にエラー処理や再試行ロジックを検証し、システムをモニタリングすることができます。詳細については、「とは AWS Fault Injection Service」を参照してください。

アクション

- API 内部エラー: ターゲットの IAM ロールからのリクエストに内部エラーを挿入します。特定のレスポンスは、各サービスと API によって異なります。アクション `aws:fis:inject-api-internal-error` により、`InternalFailure` エラー (HTTP 500) が発生します。
- API スロットルエラー: ターゲットの IAM ロールからのリクエストに内部エラーを挿入します。特定のレスポンスは、各サービスと API によって異なります。アクション `aws:fis:inject-api-throttle-error` により、`ThrottlingException` エラー (HTTP 400) が発生します。
- API 使用不可エラー: ターゲットの IAM ロールからのリクエストに内部エラーを挿入します。特定のレスポンスは、各サービスと API によって異なります。アクション `aws:fis:inject-api-unavailable-error` により、`ServiceUnavailable` エラー (HTTP 503) が発生します。
- API プロビジョンドスループット例外: ターゲットの IAM ロールからのリクエストに内部エラーを挿入します。特定のレスポンスは、各サービスと API によって異なります。アクション `aws:kinesis:inject-api-provisioned-throughput-exception` により、`ProvisionedThroughputExceededException` エラー (HTTP 400) が発生します。
- API 期限切れイテレーター例外: ターゲットの IAM ロールからのリクエストに内部エラーを挿入します。特定のレスポンスは、各サービスと API によって異なります。アクション `aws:kinesis:inject-api-expired-iterator-exception` により、`ExpiredIteratorException` エラー (HTTP 400) が発生します。

詳細については、Amazon Kinesis Data Streams actions を参照してください。

考慮事項

- 上記のアクションは、Amazon Kinesis Data Streams のプロビジョンド型およびオンデマンド型の両方のオファリングで使用できます。
- 選択した期間に基づいて実験が完了すると、ストリーミングが再開されます。実行中の実験は、完了する前に停止することもできます。または、Amazon CloudWatch Application Insights でアプリケーションの状態を定義するアラームに基づいて実験を停止するための停止条件を定義することもできます。
- 最大 280 個のストリームをテストできます。

リージョンのサポートについて詳しくは、AWS Fault Injection Service endpoints and quotas を参照してください。

プロビジョンドスループット例外エラー

Kinesis ストリームに対するリクエストレートが 1 つ以上のシャードのスループット制限を超えた場合、プロビジョンドスループット超過例外エラー (HTTP 400) が発生します。各シャードには読み取りおよび書き込みの容量制限があり、その制限を超えるとこの例外が発生します。この例外が発生する主なシナリオには、データインジェストまたは消費の急増、処理中のデータ量に対して不十分なシャード容量、またはパーティションキーの不均等な分散などがあります。

例外の処理に関する推奨事項

- 指数バックオフと再試行メカニズムを実装します。
- スループットの増加に対応できるように、シャード数を増やします。
- パーティションキーが適切に分散されていることを確認します。
- ストリームメトリクスをモニタリングします。

さらに、Kinesis のオンデマンドキャパシティモードを使用すると、ワークロードが自動的に調整され、この例外の発生を最小限に抑えることができます。詳細については、「What is AWS Fault Injection Service?」を参照してください。

> **Note**
>
> 不適切な分散による問題は、オンデマンドモードの自動スケーリング機能の対象外です。

基本的な実験を実行するには

1. ベースラインメトリクスを使用する: テストを実施する前に、通常時のスループットパターンを記録します。
2. 実験を作成する: aws:kinesis:inject-api-provisioned-throughput-exception アクションを使用します。
3. 強度を設定する: まずはリクエストスロットリングを 25% に設定します。
4. レスポンスをモニタリングする: 指数バックオフを使用して再試行ロジックを検証します。
5. スケーリングを検証する: 自動スケーリングによってアクティベーションがトリガーされることを確認します。
6. アラームを確認する: CloudWatch アラームが想定どおりに動作していることを確認します。

アプリケーションは、適切なバックオフ戦略を実装
し、WriteProvisionedThroughputExceeded メトリクスおよび
ReadProvisionedThroughputExceeded メトリクスをモニタリングし、必要に応じてシャード
スケーリングをトリガーする必要があります。

アクションの詳細

- リソースタイプ: IAM ロール ARN
- ターゲットオペレーション: PutRecord、PutRecords、GetRecords
- エラーコード: ProvisionedThroughputExceededException (HTTP 400)
- 説明: リクエストレートがシャードの容量制限を超えるシナリオをシミュレートし、アプリケーションのスロットリングおよびスケーリング応答をテストします。

パラメータ

- IAM ロール ARN: アプリケーションが Kinesis Data Streams オペレーションに使用するロール。
- オペレーション: ターゲットオペレーション: PutRecord、PutRecords、GetRecords。
- リソースリスト: 特定のストリーム名またはシャード識別子。
- 期間: 実験期間。1 分から 12 時間です。 AWS FIS API では、値は ISO 8601 形式の文字列です。例えば、PT1M は 1 分を表します。 AWS FIS コンソールで、秒数、分数、または時間数を入力します。
- 強度: スロットリングするリクエストの割合。

必要なアクセス許可

- kinesis:InjectApiError

実験テンプレートの例

次の例は、指定されたタグを持つ最大 5 つの Kinesis Data ストリームに対するプロビジョニングされたスループット例外を示しています。 は、ランダムに影響するストリーム AWS FIS を選択します。5 分後に障害が除去されます。

```
{
    "description": "Kinesis stream experiment",
    "targets": {
        "KinesisStreams-Target-1": {
            "resourceType": "aws:kinesis:stream",
            "resourceTags": {
                   "tag-key": "tag-value"
            },
            "selectionMode": "COUNT(5)"
        }
    },
    "actions": {
         "kinesis": {
              "actionId": "aws:kinesis:stream-provisioned-throughput-exception",
              "description": "my-stream",
              "parameters": {
                   "duration": "PT5M",
                   "percentage": "100",
                   "service": "kinesis"
              },
              "targets": {
                    "KinesisStreams": "KinesisStreams-Target-1"
              }
         }
   },
   "stopConditions": [
         {
              "source": "none"
         }
   ],
   "roleArn": "arn:aws:iam::111122223333:role/role-name",
   "tags": {},
   "experimentOptions": {
```

```
        "accountTargeting": "single-account",
        "emptyTargetResolutionMode": "fail"
    }
}
```

実験ロールのアクセス許可の例

次のアクセス許可により、リクエストの 50% に影響する特定のストリームで aws:kinesis:stream-provisioned-throughput-exception および aws:kinesis:stream-expired-iterator-exception のアクションを実行できます。

期限切れイテレーター例外エラー

期限切れイテレーター例外エラー (HTTP 400) は、シャードイテレーターが有効期限切れとなり、GetRecords の呼び出し時にストリームレコードの取得に使用できなくなった場合に発生します。これは、長時間実行されるデータ処理タスク、ネットワークの問題、またはアプリケーションのダウンタイムによって読み取り処理の間に遅延が発生した場合に起こります。

> **Note**
>
> シャードイテレーターは、発行されてから 5 分間有効です。

例外の処理に関する推奨事項

- シャードイテレーターが期限切れになる前に更新します。
- 新しいイテレーターを取得するためのエラー処理を組み込みます。
- シャードイテレーターの有効期限を自動的に管理する Kinesis Client Library (KCL) を使用します。

詳細については、「 とは」を参照してください AWS Fault Injection Service。

基本的な実験を実行するには

1. 実験テンプレートを作成する: AWS FIS コンソールを使用します。
2. アクションを選択する: aws:kinesis:inject-api-expired-iterator-exception アクションを使用します。
3. ターゲットを設定する: IAM ロールと Kinesis Data Streams オペレーションを指定します。
4. 期間を設定する: 初回テストでは 5〜10 分から開始します。

5. 停止条件の追加: の停止条件。 AWS FIS
6. 実験を実行する: アプリケーションの動作をモニタリングします。

アクションの詳細

- リソースタイプ: IAM ロール ARN
- ターゲットオペレーション: `GetRecords`
- エラーコード: `ExpiredIteratorException`(HTTP 400)
- 説明: 指定されたイテレーターが許可されている最大の有効期間を超えた場合に発生し、レコード処理が遅すぎる、またはチェックポイント処理ロジックが失敗するシナリオをシミュレートします。

パラメータ

- IAM ロール ARN: アプリケーションが Kinesis Data Streams オペレーションに使用するロール。
- オペレーション: ターゲットオペレーション: `GetRecords`
- リソースリスト: 特定のストリーム名または ARN。
- 期間: 実験期間。これは設定可能です。
- 強度: スロットリングするリクエストの割合。

必要なアクセス許可

- `kinesis:InjectApiError`

Amazon Kinesis Data Streams にデータを書き込む

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 (KPL) 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AWS CLI を使用して Amazon Kinesis Data Streams オペレーションを実行する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、データ保持期間を変更する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。

トピック

- Amazon Kinesis Producer Library (KPL) を使用してプロデューサーを開発する
- Amazon Kinesis Data Streams API と AWS SDK for Java を使用してプロデューサーを開発する
- Kinesis エージェントを使用して Amazon Kinesis Data Streams に書き込む
- その他の AWS サービスを使用した Kinesis Data Streams への書き込み
- サードパーティーの統合を使用して Kinesis Data Streams に書き込む
- Amazon Kinesis Data Streams プロデューサーのトラブルシューティング
- Kinesis Data Streams プロデューサーを最適化する

Amazon Kinesis Producer Library (KPL) を使用してプロデューサーを開発する

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。Amazon 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 CloudWatch で KPL をモニタリングできます。詳細については、Amazon CloudWatch を使用した Kinesis Client Library を監視するを参照してください。

トピック

- KPL のロールを確認する
- KPL を使用するメリットを実現する
- KPL を使用しないタイミングを理解する
- KPL をインストールする
- KPL 0.x から KPL 1.x に移行する
- KPL の Amazon Trust Services (ATS) 証明書への移行
- KPL でサポートされるプラットフォーム
- KPL の主要な概念
- KPL をプロデューサーコードと統合する
- KPL を使用して Kinesis Data Stream に書き込む
- Amazon Kinesis Producer Library を設定する
- コンシューマーの集約解除を実装する
- Amazon Data Firehose で KPL を使用する
- AWS Glue スキーマレジストリで KPL を使用する
- KPL プロキシ設定を設定する
- KPL バージョンライフサイクルポリシー

> **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 のロールを確認する

KPL は使いやすく、Kinesis Data Streams への書き込みに役立つ、高度な設定可能ライブラリです。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
- Amazon CloudWatch メトリクスをユーザーに代わって送信し、プロデューサーのパフォーマンスを確認可能にする

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を実現する

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 を使用して Kinesis Data Stream に書き込む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 CloudWatch 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、分析できます。KPL は、スループット、エラー、およびその他のメトリクスをユーザーに代わって CloudWatch に送信し、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、ランタイム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 を使用しないタイミングを理解する

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 での AWS SDK の使用の詳細

については、Amazon Kinesis Data Streams API と AWS SDK for Java を使用してプロデューサーを開発する を参照してください。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、Amazon Kinesis Producer Library を設定する を参照してください。

KPL をインストールする

Amazon では、macOS、Windows、最新の Linux ディストリビューション向けに C++ Amazon 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。

以前のバージョンの Linux ディストリビューションのユーザーは、GitHub のソースとともに提供されるビルド手順で KPL をビルドできます。KPL を GitHub からダウンロードするには、Amazon Kinesis Producer Library を参照してください。

> **Important**
>
> Amazon Kinesis Producer Library (KPL) 0.x は、2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 0.x を使用して KPL アプリケーションを最新の KPL バージョンに移行することを強くお勧めします。最新の KPL バージョンについては、Github の KPL ページを参照してください。KPL 0.x から KPL 1.x への移行については、KPL 0.x から KPL 1.x に移行する を参照してください。

KPL 0.x から KPL 1.x に移行する

このトピックでは、コンシューマーを KPL 0.x から KPL 1.x に移行する方法について、段階的に説明します。KPL 1.x は、従来バージョンとのインターフェイス互換性を維持しつつ、AWS SDK for

Java 2.x のサポートを新たに導入しています。KPL 1.x へ移行する際、データ処理の中核となるロジックを更新する必要はありません。

1. 次の前提条件を満たしていることを確認する

   - Java Development Kit (JDK) 8 以降
   - AWS SDK for Java 2.x
   - 依存関係管理用の Maven または Gradle

2. 依存関係を追加する

   Maven を使用している場合は、以下の依存関係を pom.xml ファイルに追加します。必ず groupId を `com.amazonaws` から `software.amazon.kinesis` に更新し、バージョンを `1.x.x` から最新の KPL バージョンに更新してください。

   ```
   <dependency>
       <groupId>software.amazon.kinesis</groupId>
       <artifactId>amazon-kinesis-producer</artifactId>
       <version>1.x.x</version> <!-- Use the latest version -->
   </dependency>
   ```

   Gradle を使用している場合は、以下を `build.gradle` ファイルに追加します。必ず `1.x.x` を最新の KPCL バージョンに置き換えてください。

   ```
   implementation 'software.amazon.kinesis:amazon-kinesis-producer:1.x.x'
   ```

   最新の KPL バージョンは、Maven Central Repository で確認できます。

3. KPL の import ステートメントを更新する

   KPL 1.x では AWS SDK for Java 2.x が使用され、パッケージ名も旧バージョンとは異なります。以前のバージョンではパッケージ名が `com.amazonaws.services.kinesis` で始まっていましたが、KPL 1.x では `software.amazon.kinesis` に変更されています。

   `com.amazonaws.services.kinesis` の import ステートメントを `software.amazon.kinesis` に置き換えます。次の表に、置き換える必要がある import ステートメントを示します。

Import の置き換え

置き換え前:	次と置き換えます。
import com.amazonaws.services.kinesis.producer.Attempt;	import software.amazon.kinesis.producer.Attempt;
import com.amazonaws.services.kinesis.producer.BinaryToHexConverter;	import software.amazon.kinesis.producer.BinaryToHexConverter;
import com.amazonaws.services.kinesis.producer.CertificateExtractor;	import software.amazon.kinesis.producer.CertificateExtractor;
import com.amazonaws.services.kinesis.producer.Daemon;	import software.amazon.kinesis.producer.Daemon;
import com.amazonaws.services.kinesis.producer.DaemonException;	import software.amazon.kinesis.producer.DaemonException;
import com.amazonaws.services.kinesis.producer.FileAgeManager;	import software.amazon.kinesis.producer.FileAgeManager;
import com.amazonaws.services.kinesis.producer.FutureTimedOutException;	import software.amazon.kinesis.producer.FutureTimedOutException;
import com.amazonaws.services.kinesis.producer.GlueSchemaRegistrySerializerInstance;	import software.amazon.kinesis.producer.GlueSchemaRegistrySerializerInstance;
import com.amazonaws.services.kinesis.producer.HashedFileCopier;	import software.amazon.kinesis.producer.HashedFileCopier;
import com.amazonaws.services.kinesis.producer.IKinesisProducer;	import software.amazon.kinesis.producer.IKinesisProducer;
import com.amazonaws.services.kinesis.producer.IrrecoverableError;	import software.amazon.kinesis.producer.IrrecoverableError;

置き換え前:	次と置き換えます。
import com.amazonaws.services.kinesis.producer.KinesisProducer;	import software.amazon.kinesis.producer.KinesisProducer;
import com.amazonaws.services.kinesis.producer.KinesisProducerConfiguration;	import software.amazon.kinesis.producer.KinesisProducerConfiguration;
import com.amazonaws.services.kinesis.producer.LogInputStreamReader;	import software.amazon.kinesis.producer.LogInputStreamReader;
import com.amazonaws.services.kinesis.producer.Metric;	import software.amazon.kinesis.producer.Metric;
import com.amazonaws.services.kinesis.producer.ProcessFailureBehavior;	import software.amazon.kinesis.producer.ProcessFailureBehavior;
import com.amazonaws.services.kinesis.producer.UnexpectedMessageException;	import software.amazon.kinesis.producer.UnexpectedMessageException;
import com.amazonaws.services.kinesis.producer.UserRecord;	import software.amazon.kinesis.producer.UserRecord;
import com.amazonaws.services.kinesis.producer.UserRecordFailedException;	import software.amazon.kinesis.producer.UserRecordFailedException;
import com.amazonaws.services.kinesis.producer.UserRecordResult;	import software.amazon.kinesis.producer.UserRecordResult;
import com.amazonaws.services.kinesis.producer.protobuf.Messages;	import software.amazon.kinesis.producer.protobuf.Messages;
import com.amazonaws.services.kinesis.producer.protobuf.Config;	import software.amazon.kinesis.producer.protobuf.Config;

4. AWS 認証情報プロバイダークラスの import ステートメントを更新する

   KPL 1.x に移行する際、KPL アプリケーションコード内で使用している import ステートメントのパッケージおよびクラスを AWS SDK for Java 1.x に基づくものから AWS SDK for Java 2.x に基づくものへ更新する必要があります。KPL アプリケーションでよく使用される import ス

テートメントは、認証情報プロバイダークラスにあります。認証情報プロバイダーの変更の一覧は、AWS SDK for Java 2.x 移行ガイドのドキュメントの認証情報プロバイダーの変更を参照してください。以下は、KPL アプリケーションで一般的に必要となる import ステートメントの変更例です。

KPL 0.x の import ステートメント

```
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
```

KPL 1.x の import ステートメント

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
```

AWS SDK for Java 1.x に基づくその他の認証情報プロバイダークラスをインポートしている場合は、AWS SDK for Java 2.x に相当するクラスへ更新する必要があります。AWS SDK for Java 1.x のクラスやパッケージを一切インポートしていない場合は、この手順を無視して構いません。

5. KPL 設定内の認証情報プロバイダー設定を更新する

KPL 1.x の認証情報プロバイダー設定には、AWS SDK for Java 2.x の認証情報プロバイダーが必要です。`KinesisProducerConfiguration` でデフォルトの認証情報プロバイダーを上書きし、AWS SDK for Java 1.x の認証情報プロバイダーを渡している場合は、AWS SDK for Java 2.x の認証情報プロバイダーに更新する必要があります。認証情報プロバイダーの変更の一覧は、AWS SDK for Java 2.x 移行ガイドのドキュメントの認証情報プロバイダーの変更を参照してください。KPL 設定でデフォルトの認証情報プロバイダーを上書きしていない場合は、この手順は無視できます。

たとえば、KPL のデフォルト認証情報プロバイダーを次のコードで上書きしている場合です。

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration();
// SDK v1 default credentials provider
config.setCredentialsProvider(new DefaultAWSCredentialsProviderChain());
```

AWS SDK for Java 2.x の認証情報プロバイダーを使用するためには、これらを次のコードに更新する必要があります。

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration();
// New SDK v2 default credentials provider
```

```
config.setCredentialsProvider(DefaultCredentialsProvider.create());
```

KPL 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、Amazon 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 リージョンに影響があります。

ATS への移行の詳細については、独自の認証機関への AWS の移行の準備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

Amazon 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 iOS 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 パブリックライセンスの下で公開され、GitHub の Amazon Kinesis Producer Library 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、Amazon 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
- 集約
- 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。

集約

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 をプロデューサーコードと統合する

Amazon 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	*	*	*	不可	該当なし
Java	Java	Java SDK を直接使用	該当なし	可	集約を使用する場合、`GetRecords` 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	可	集約を無効にする必要があります。
Java	Java	1.3.x	該当なし	可	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	可	なし

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	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	可	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	可	集約を無効にする必要があります。

KPL を使用して Kinesis Data Stream に書き込む

以下のセクションでは、最も基本的な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。Amazon 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、Future から返される addUserRecord 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、ランタイムがブロックされます。ランタイム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);
}
```

Amazon Kinesis Producer Library を設定する

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では、PascalCase 内の名前を使用してパラメータを指定する必要があります。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方法と値の制限の詳細については、sample configuration properties file on 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を実装する

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。

オプション

- 以前のバージョンの KCL から移行する
- KPL の集約解除のための KCL の拡張を使用する
- GetRecords を直接使用する

以前のバージョンの KCL から移行する

集約を使用している場合でも、既存のチェックポイント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 の拡張を使用する

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 UserRecord を必ず使用するため、UserRecord から Record 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。

GetRecords を直接使用する

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、0 のデフォルト値 startingHashKey (ゼロ) と 2^128 -1 のデフォルト値 endingHashKey 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。

Amazon Data Firehose で KPL を使用する

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場合、Firehose はレコード

の集約を解除してから送信先に配信します。データを変換するように配信ストリームを設定する場合、Firehose はレコードの集約を解除してから AWS Lambda に配信します。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 KPL を使用する

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、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 Data Streams でのみサポートされます。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 プロキシ設定を設定する

インターネットに直接接続できないアプリケーションのために、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ています。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、プロキシサーバーを使用する環境内でデータを収集して AWS に送信するために Kinesis Producer Library (KPL) を使用している場合は、アプリケーションに KPL プロキシ設定が必要になります。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);
```

KPL バージョンライフサイクルポリシー

このトピックでは、Amazon Kinesis Producer Library (KPL) のバージョンライフサイクルポリシーについて説明します。AWS は、新機能や機能強化、バグ修正、セキュリティパッチ、依存関係の更新に対応するため、KPL の新しいバージョンを定期的にリリースしています。最新の機能、セキュリティ更新、基本的な依存関係に対応するため、KPL を常に最新バージョンへ更新しておくことをお勧めします。サポートが終了した KPL バージョンを継続使用することはお勧めしません。

主要な KPL バージョンのライフサイクルは、次の 3 つのフェーズで構成されます。

- 一般提供 (GA) – このフェーズでは、そのメジャーバージョンは完全にサポートされます。AWS は、新機能や Kinesis Data Streams の API 更新への対応、さらにバグ修正やセキュリティ修正を含むマイナーバージョンおよびパッチバージョンを定期的に提供します。
- メンテナンスモード - AWS は重大なバグ修正とセキュリティ問題に対応するためのパッチバージョンのみを提供制限します。Kinesis Data Streams の新機能や API に対する更新は、このメジャーバージョンには行われません。
- サポート終了 – そのメジャーバージョンに対して更新やリリースは一切提供されません。以前に公開されたリリースは引き続き公開パッケージマネージャーから入手でき、コードは GitHub に残ります。サポートが終了したバージョンの使用は、ユーザーの裁量で行われます。最新のメジャーバージョンにアップグレードすることをお勧めします。

メジャーバージョン	現在のフェーズ	リリース日	メンテナンスモード日	サポート終了日
KPL 0.x	メンテナンスモード	2015-06-02	2025-04-17	2026-01-30
KPL 1.x	一般提供	2024-12-15	--	--

Amazon Kinesis Data Streams API と AWS SDK for Java を使用してプロデューサーを開発する

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AWS CLI を使用して Amazon Kinesis Data Streams オペレーションを実行する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 (KPL) を使用してプロデューサーを開発する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、「Kinesis Data Streams を作成して管理する」を参照してください。

トピック

- ストリームにデータを追加する
- 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

ストリームにデータを追加する

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 を使用して複数のレコードを追加する
- PutRecord を使用して単一レコードを追加する

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を取得するを参照してください。

> **Important**
>
> データ保持期間を変更する

PutRecords を使用して複数のレコードを追加する

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 使用時のエラーを処理する

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 を使用して単一レコードを追加する

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 を使用して複数のレコードを追加する 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、SequenceNumberForOrderingのサンプルコードに示しているように、PutRecord の例 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て Amazon Kinesis Data Streams に書き込む

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、ストリーミング処理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 (エージェント設定を指定するを参照)。

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

- Kinesis Agent の前提条件を完了する
- エージェントをダウンロードしてインストールする
- エージェントを設定して開始する
- エージェント設定を指定する
- 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む
- 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 CLI コマンドを使用する
- よくある質問

Kinesis Agent の前提条件を完了する

- 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際に AWS 認証情報を指定します (awsAccessKeyId および awsSecretAccessKey を参照してください)。
- /etc/sysconfig/aws-kinesis-agent を編集して、 リージョンと AWS クセスキーを指定します。
- EC2 インスタンスが他の AWS アカウントにある場合、Kinesis Data Streams サービスへのアクセス権を付与する 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 と assumeRoleExternalId を参照)。前のいずれかの方法を使用して、このロールを引き受ける許可がある、他のアカウントのユーザーの AWS 認証情報を指定します。

- 指定した IAM ロールまたは AWS 認証情報には、Kinesis Data Streams PutRecords オペレーションを実行してエージェントからストリームにデータを送信するための許可が必要です。エージェントの CloudWatch モニタリングを有効にしている場合は、CloudWatch PutMetricData オペレーションを実行する許可も必要になります。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 CloudWatch による Kinesis Data Streams エージェントのヘルスに監視する、およびCloudWatch アクセスコントロールを参照してください。

エージェントをダウンロードしてインストールする

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「Linux インスタンスへの接続」を参照してください。接続できない場合は、「Amazon EC2 ユーザーガイド」の「インスタンスへの接続に関する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
```

GitHubを使用してエージェントを設定する

1. エージェントを awlabs/amazon-kinesis-agent から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"]
```

エージェントを設定して開始する

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。

エージェント設定を指定する

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AWS アカウント間の IAM ロールを使用したアクセスの委任を参照してください。

構成設定	説明
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です。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。
awsSecretAccessKey	AWSデフォルトの認証情報を上書きする シークレットキーです。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	エージェントがメトリクスを CloudWatch に出力できるようにします (true に設定された場合)。 デフォルト: true
cloudwatch.endpoint	CloudWatch のリージョンのエンドポイントです。 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

構成設定	説明
	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
maxBufferAgeMillis	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)
maxBufferSizeBytes	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)
maxBufferSizeRecords	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
minTimeBetweenFilePollsMillis	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

構成設定	説明
multiLineStartPattern	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。
partitionKeyOption	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
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む

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。以下の設定例では、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 ストリームおよび 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できるため、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はありません。

```
{
    "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 でのエージェントの使用の詳細については、「Kinesis エージェントを使用した 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口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口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 コマンドを使用する

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` メトリクスをチェックします。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。高度なエージェント設定については、「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 データストリームへのデータの書き込みのために Amazon 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。関心のある各サービスの情報を確認し、提供されたリファレンスを参照してください。

トピック

- AWS Amplify を使用して Kinesis Data Stream に書き込む
- Amazon Aurora を使用して Kinesis Data Stream に書き込む
- Amazon CloudFront を使用して Kinesis Data Stream に書き込む
- Amazon CloudWatch Logs を使用して Kinesis Data Streams に書き込む
- Amazon Connect を使用して Kinesis Data Streams に書き込む
- AWS Database Migration Service を使用して Kinesis Data Stream に書き込む
- Amazon DynamoDB を使用して Kinesis Data Streams に書き込む
- Amazon EventBridge を使用して Kinesis Data Streams に書き込む
- AWS IoT Core を使用して Kinesis Data Stream に書き込む
- Amazon Relational Database Service を使用して Kinesis Data Streams に書き込む
- Amazon Pinpoint を使用して Kinesis Data Streams に書き込む
- Amazon Quantum Ledger Database (Amazon QLDB) を使用して Kinesis Data Streams に書き込む

AWS Amplify を使用して Kinesis Data Stream に書き込む

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 を使用して Kinesis Data Stream に書き込む

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 を使用して Kinesis Data Stream に書き込む

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。独自の Kinesis データストリームコンシューマーを構築することも、Amazon Data Firehose を使用してログデータを Amazon S3、Amazon Redshift、Amazon OpenSearch Service、またはサードパーティーのログ処理サービスに送信することもできます。

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 を使用して Kinesis Data Streams に書き込む

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、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 を使用して Kinesis Data Streams に書き込む

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 を使用して Kinesis Data Stream に書き込む

AWS Database Migration Service を使用して、Kinesis データストリームにデータを移行できます。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 を使用して Kinesis Data Streams に書き込む

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 テーブルの項目レベルの変更をキャプチャーし、それらを Kinesis Data Streams 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 を使用して Kinesis Data Streams に書き込む

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 を使用して Kinesis Data Stream に書き込む

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 を使用して Kinesis Data Streams に書き込む

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 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 Data 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 を使用して Kinesis Data Streams に書き込む

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 (Amazon QLDB) を使用して Kinesis Data Streams に書き込む

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 Amazon 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。

サードパーティーの統合を使用して Kinesis Data Streams に書き込む

Kinesis Data Streams 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。詳細を確認し、関連ドキュメントのリソースとリンクを検索するオプションを選択します。

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

以下のトピックでは、Amazon Kinesis Data Streams プロデューサーの一般的な問題に対するソリューションを提供します。

- 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
- 不正な KMS マスターキーアクセス許可エラーが表示される
- プロデューサーのその他の一般的な問題のトラブルシューティング

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
- プロデューサーを最適化したい
- flushSync() オペレーションの誤用

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、Kinesis Data Streams を作成して管理する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。

プロデューサーを最適化したい

プロデューサーの最適化を始める前に、次の重要なタスクを完了しておく必要があります。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、Amazon 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。

`flushSync()` オペレーションの誤用

`flushSync()` を誤って使用すると、書き込みパフォーマンスに大きな悪影響を及ぼす可能性があります。`flushSync()` オペレーションは、KPL アプリケーション終了時にバッファされたすべてのレコードが送信されることを保証するための、シャットダウン時専用の操作です。このオペレーションを書き込みのたびに実行するように実装している場合、1 回の書き込みごとに約 500 ミリ秒 の大きな遅延が追加されます。書き込みパフォーマンスの不要な遅延を避けるため、`flushSync()` はアプリケーションのシャットダウン時のみ実行されるように実装されていることを確認してください。

不正な KMS マスターキーアクセス許可エラーが表示される

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、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。

プロデューサーのその他の一般的な問題のトラブルシューティング

- 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 プロデューサーを最適化する

表示される特定の動作に応じて、Amazon Kinesis Data Streams プロデューサーをさらに最適化できます。以下のトピックを確認して、解決策を特定します。

トピック

- KPL の再試行とレート制限の動作をカスタマイズする
- KPL 集約にベストプラクティスを適用する

KPL の再試行とレート制限の動作をカスタマイズする

KPL addUserRecord( ) オペレーションを使用して Amazon 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 集約にベストプラクティスを適用する

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 Amazon 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実装するを参照してください。

Amazon Kinesis Data Streams からデータを読み取る

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、専用スループットを備えた拡張ファンアウトを開発するを参照してください。

Kinesis Data Streams のコンシューマーは、Kinesis Client Library (KCL) または AWS SDK for Java を使用して構築できます。Amazon Managed Service for Apache Flink AWS Lambdaや Amazon Data Firehose AWS などの他のサービスを使用してコンシューマーを開発することもできます。さらに、Kinesis Data Streams は、Amazon EMR、Amazon EventBridge、 AWS Glue、Amazon Redshift などの AWS サービスとの連携をサポートしています。また、Apache Flink、Adobe Experience Platform、Apache Druid、Apache Spark、Databricks、Confluent Platform、Kinesumer、Talend など、サードパーティ製ツールとの連携にも対応しています。

トピック

- 専用スループットを備えた拡張ファンアウトを開発する
- Kinesis コンソールでデータビューワーを使用する
- Kinesis コンソールでデータストリームをクエリする
- Kinesis Client Library を使用する
- AWS SDK for Java を使用してコンシューマーを開発する
- 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 を使用してコンシューマーを開発する
- Amazon Data Firehose を使用してコンシューマーを開発する
- その他の AWS サービスを使用して Kinesis Data Streams からデータを読み取る
- サードパーティーの統合を使用して Kinesis Data Streams から読み取る
- 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 コンシューマーを最適化する

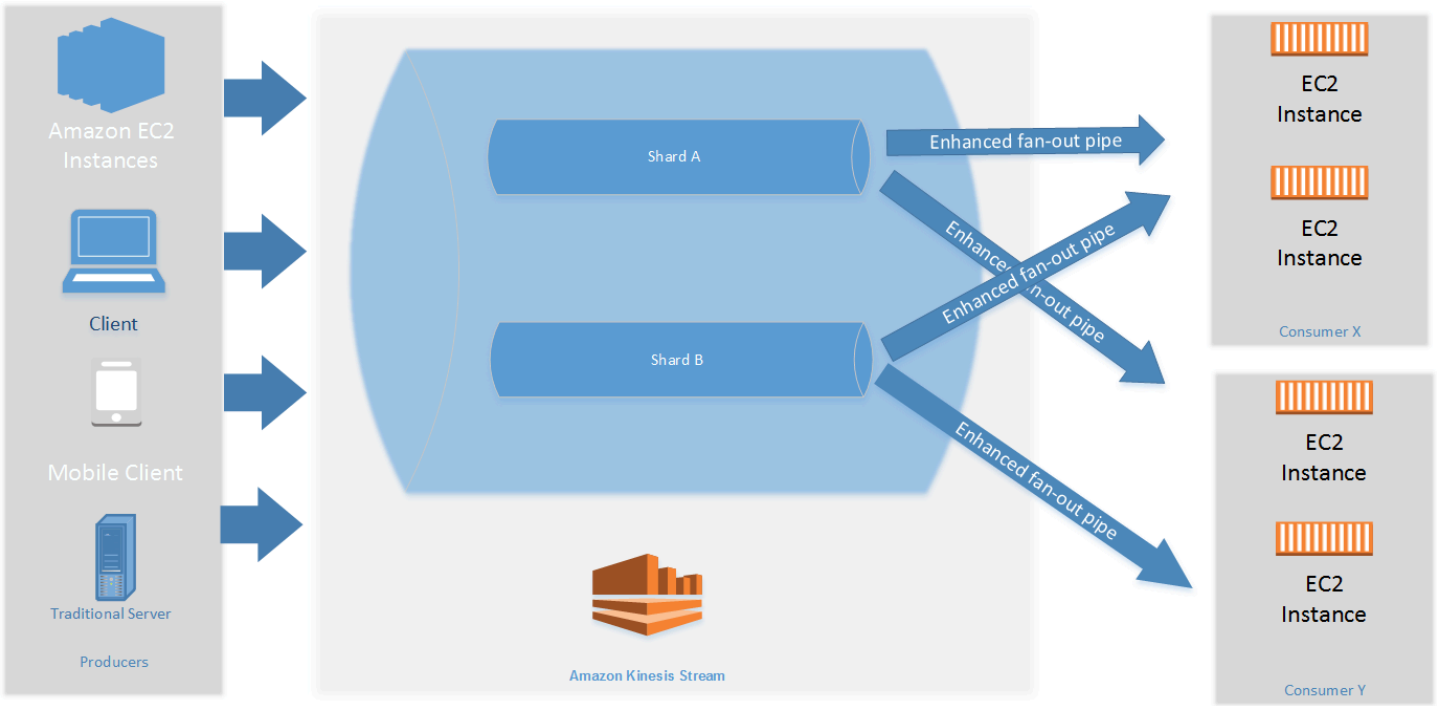
専用スループットを備えた拡張ファンアウトを開発する

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**
>
> オンデマンドアドバンテージモードでは、ストリームごとに最大 50 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。オンデマンドの標準ストリームとプロビジョンドストリームでは、ストリームごとに最大 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

- 共有スループッ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違い
- 最大 50 の拡張ファンアウトコンシューマーでサポートされているリージョン (オンデマンド Advantage のみ)
-
- AWS CLI または APIs 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

共有スループッ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違い

次の表は、デフォルトの共有スループットコンシューマーと、拡張ファンアウトコンシューマーを比較したものです。メッセージ伝達遅延時間は、ペイロードを送信する API (PutRecord や PutRecords) 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 (GetRecords や SubscribeToShard) を経由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った時間 (ミリ秒) として定義されます。

共有スループットコンシューマーと拡張ファンアウトコンシューマーを比較した表

特性	拡張ファンアウトなしの共有スループットコンシューマー	拡張ファンアウトコンシューマー
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	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

特性	拡張ファンアウトなしの共有スループットコンシューマー	拡張ファンアウトコンシューマー
	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。	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 つの場合、この平均は最大約 1,000 ms まで上がります。	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。
Cost	該当しない	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	GetRecords を使用した HTTP 経由のプルモデルです。	Kinesis Data Streams は、SubscribeToShard を使用して、HTTP/2 経由でレコードをプッシュします。

最大 50 の拡張ファンアウトコンシューマーでサポートされているリージョン (オンデマンド Advantage のみ)

オンデマンドアドバンテージモードで最大 50 の拡張ファンアウトコンシューマーのサポートは、次の AWS リージョンでのみ使用できます。

AWS リージョン	リージョン名
eu-north-1	欧州 (ストックホルム)

AWS リージョン	リージョン名
me-south-1	中東 (バーレーン)
ap-south-1	アジアパシフィック (ムンバイ)
eu-west-3	欧州 (パリ)
ap-southeast-3	アジアパシフィック (ジャカルタ)
us-east-2	米国東部 (オハイオ)
af-south-1	アフリカ (ケープタウン)
eu-west-1	欧州 (アイルランド)
me-central-1	中東 (アラブ首長国連邦)
eu-central-1	欧州 (フランクフルト)
sa-east-1	南米 (サンパウロ)
ap-east-1	アジアパシフィック (香港)
ap-south-2	アジアパシフィック (ハイデラバード)
us-east-1	米国東部 (バージニア北部)
ap-northeast-2	アジアパシフィック (ソウル)
ap-northeast-3	アジアパシフィック (大阪)
eu-west-2	欧州 (ロンドン)
ap-southeast-4	アジアパシフィック (メルボルン)
ap-northeast-1	アジアパシフィック (東京)
us-west-2	米国西部 (オレゴン)
us-west-1	米国西部 (北カリフォルニア)

AWS リージョン	リージョン名
ap-southeast-1	アジアパシフィック (シンガポール)
ap-southeast-2	アジアパシフィック (シドニー)
il-central-1	イスラエル (テルアビブ)
ca-central-1	カナダ (中部)
ca-west-1	カナダ西部 (カルガリー)
eu-south-2	欧州 (スペイン)
cn-northwest-1	中国 (寧夏)
eu-central-2	欧州 (チューリッヒ)
us-gov-east-1	AWS GovCloud (米国東部)
us-gov-west-1	AWS GovCloud (米国西部)

AWS CLI または APIs 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、「専用スループットを備えた拡張ファンアウトを開発する」を参照してください。

AWS CLI または Kinesis Data Streams APIs を使用して、Kinesis Data Streams で拡張ファンアウトを使用するコンシューマーを登録、説明、一覧表示、および登録解除できます。

を使用してコンシューマーを管理する AWS CLI

を使用して、拡張ファンアウトコンシューマーを登録、説明、一覧表示、および登録解除できます AWS CLI。例については、次のドキュメントを参照してください。

register-stream-consumer

Kinesis データストリームにコンシューマーを登録します。コンシューマーの登録時にタグを適用できます。

describe-stream-consumer

登録済みコンシューマーの詳細情報を、コンシューマー ARN またはコンシューマー名とストリーム ARN の組み合わせを指定して取得します。

list-stream-consumers

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ように登録されているコンシューマーを一覧表示します。

deregister-stream-consumer

コンシューマー ARN、またはコンシューマー名とストリーム ARN の組み合わせのいずれかを指定して、コンシューマーを登録解除します。

Kinesis Data Streams API を使用してコンシューマーを管理する

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登録、説明、一覧表示、および登録解除できます。例については、次のドキュメントを参照してください。

RegisterStreamConsumer

タグを使用して Kinesis データストリームにコンシューマーを登録します。コンシューマーの登録時にタグを適用できます。

DescribeStreamConsumer

登録済みコンシューマーの詳細情報を、コンシューマー ARN またはコンシューマー名とストリーム ARN の組み合わせを指定して取得します。

ListStreamConsumers

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ように登録されているコンシューマーを一覧表示します。

DeregisterStreamConsumer

コンシューマー ARN、またはコンシューマー名とストリーム ARN の組み合わせのいずれかを指定して、コンシューマーを登録解除します。

コンシューマーのタグ付け

Kinesis Data Streams で作成したストリームおよび拡張ファンアウトコンシューマーに、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。タグを使用して、コンシューマーのコストを分類して追跡できます。属性ベースのアクセス制御 (ABAC) では、タグを使用してコンシューマー関数へのアクセスも制御できます。詳細については、「Amazon Kinesis Data Streams リソースにタグを付ける」を参照してください。

Kinesis コンソールでデータビューワーを使用する

Kinesis マネジメントコンソールのデータビューワーでは、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できます。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. にサインイン AWS マネジメントコンソール し、https://console.aws.amazon.com/kinesis で 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。

[データビューワー] で [レコードを取得] または [次へ] ボタンをクリックすると、[GetRecords] API が呼び出され、1 秒あたり 5 トランザクションの [GetRecords] API 上限に反映されることに注意してください。

Kinesis コンソールでデータストリームをクエリする

Kinesis Data Streams コンソールの Data Analytics タブを使用すると、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、次のステップに従います。

1. にサインイン AWS マネジメントコンソール し、https://console.aws.amazon.com/kinesis で 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、Managed Apache Flink Studio ノートブックを使用してストリーム検査と視覚化を実行できます。Apache Zeppelin を使用して、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」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イスです。ストリームの名前を含むメモを選択します。
5. 既に実行中のストリーム内のデータを出力するためのサンプル SELECT クエリを含む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. タンブリングウィンドウ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] タブで [サンプルクエリを表示] を選択します。クエリをコピーし、データストリームスキーマに合わせて変更してから、Zeppelin ノートの新しい段落で実行します。

Kinesis Client Library を使用する

Kinesis Client Library (KCL) とは何ですか?

Kinesis Client Library (KCL) は、Amazon Kinesis Data Streams からデータを取得して処理する作業を簡素化するために設計された、スタンドアロンの Java ソフトウェアライブラリです。KCL は、分散コンピューティングに伴う複雑な処理を代行するため、開発者はデータ処理のビジネスロジックの

実装に集中できます。KCL は、複数ワーカー間での負荷分散、ワーカー障害への対応、処理済みレコードのチェックポイント管理、ストリーム内のシャード数の変化への対応などの処理を管理します。

KCL は、基盤となるライブラリの新しいバージョンの取り込みや、セキュリティ強化、バグ修正を行うために頻繁に更新されています。既知の問題を回避し、最新の改善点を活用するためにも、常に最新バージョンの KCL を使用することをお勧めします。最新の KCL バージョンについては、KCL の Github を参照してください。

> **Important**
>
> - 既知のバグや問題を回避するために、最新バージョンの KCL を使用することをお勧めします。KCL 2.6.0 以前を使用している場合、ストリーム容量の変化時にシャード処理が停止する可能性があるまれな事象を回避するため、KCL 2.6.1 以降へアップグレードしてください。
> - KCL は Java ライブラリです。Java 以外の言語をサポートするために、MultiLangDaemon と呼ばれる Java ベースのデーモンが提供されています。MultiLangDaemon は、STDIN および STDOUT を介して KCL アプリケーションと連携します。GitHub の MultiLangDaemon の詳細については、Java 以外の言語で KCL を使用してコンシューマーを開発する を参照してください。
> - KCL 3.x で AWS SDK for Java バージョン 2.27.19 ~ 2.27.23 を使用しないでください。これらのバージョンには、KCL が使用する DynamoDB に関連した例外エラーを引き起こす問題が含まれています。この問題を回避するには、 AWS SDK for Java バージョン 2.28.0 以降を使用することをお勧めします。

KCL の主な機能と利点

KCL の主な機能と関連する利点は次のとおりです。

- スケーラビリティ: KCL は、複数のワーカー間で処理負荷を分散することで、アプリケーションを動的にスケールさせることを可能にします。手動または自動スケーリングを使用してアプリケーションをスケールインまたはスケールアウトしても、負荷の再分散について心配する必要はありません。
- 負荷分散: KCL は利用可能なワーカー間で処理負荷を自動的に均等化し、ワーカー間の作業量をバランスよく分散します。

- チェックポイント: KCL は処理済みレコードのチェックポイント管理を行い、アプリケーションが最後に正常に処理した位置から処理を再開できるようにします。
- 耐障害性: KCL には耐障害性メカニズムが組み込まれているため、個々のワーカーが障害を起こした場合でもデータ処理が継続されます。また、KCL は 少なくとも 1 回の配信を保証します。
- ストリームレベルの変更の処理: KCL は、データ量の変化によって発生するシャードの分割や結合に適応します。また、親シャードの処理とチェックポイントが完了するまで子シャードを処理しないようにすることで、レコードの順序が維持されます。
- モニタリング: KCL は Amazon CloudWatch と連携し、コンシューマーレベルのモニタリングを行います。
- 多言語サポート: KCL は Java をネイティブでサポートしており、MultiLangDaemon を通じて複数の Java 以外のプログラミング言語にも対応できます。

KCL の概念

このセクションでは、Kinesis Client Library (KCL) の主要な概念と、その相互関係について説明します。これらの概念は、KCL コンシューマーアプリケーションを開発および管理するうえで基本となるものです。

- KCL コンシューマーアプリケーション - Kinesis Client Library を使用して、Kinesis データストリームからレコードを読み取って処理するように設計されたカスタムビルドアプリケーション。
- ワーカー – KCL コンシューマーアプリケーションは、通常 1 つ以上のワーカーが同時に実行される分散構成で動作します。KCL はワーカー同士を調整し、ストリームからのデータを分散方式で取得できるようにするとともに、複数のワーカー間で処理負荷が均等になるように管理します。
- スケジューラ – KCL ワーカーがデータの処理を開始するために使用する高レベルのクラス。各 KCL ワーカーには 1 つのスケジューラがあります。スケジューラは、Kinesis データストリームからのシャード情報の同期、ワーカー間におけるシャード割り当ての追跡、そしてワーカーに割り当てられたシャードに基づくストリームからのデータ処理といった、さまざまなタスクを初期化し、監督します。スケジューラは、処理対象のストリーム名や AWS 認証情報など、スケジューラの動作に影響するさまざまな設定を行うことができます。また、スケジューラはストリームからレコードプロセッサへのデータレコードの受け渡しを開始する役割も担います。
- レコードプロセッサ - レコードプロセッサは、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受信したデータをどのように処理するかというロジックを定義します。レコードプロセッサ内には、独自のカスタムデータ処理ロジックを実装する必要があります。KCL ワーカーはスケジューラをインスタンス化します。次に、スケジューラは、ワーカーがリースを保持している

各シャードに対して 1 つのレコードプロセッサをインスタンス化します。1 つのワーカーは複数のレコードプロセッサを実行できます。

- リース – ワーカーとシャード間の割り当てを定義します。KCL コンシューマーアプリケーションは、複数のワーカーにデータレコード処理を分散するためにリースを使用します。各シャードは、任意の時点でリースによって 1 つのワーカーにのみ結び付けられ、各ワーカーは同時に 1 つ以上のリースを保持できます。ワーカーが停止または障害によってリースを保持できなくなると、KCL は別のワーカーにそのリースを引き継がせます。リースの詳細については、Github documentation: Lease Lifecycle を参照してください。
- リーステーブル – KCL コンシューマーアプリケーションにおけるすべてのリースを追跡するために使用される、専用の Amazon DynamoDB テーブルです。各 KCL コンシューマーアプリケーションは、独自のリーステーブルを作成します。このリーステーブルは、すべてのワーカー間で状態を保持し、データ処理を調整するために使用されます。詳細については、「KCL における DynamoDB メタデータテーブルと負荷分散」を参照してください。
- チェックポイント — シャード内で最後に正常に処理されたレコードの位置を永続的に保存するプロセス。KCL はチェックポイントを管理し、ワーカーが障害を起こした場合やアプリケーションが再起動した場合でも、最後にチェックポイントされた位置から処理を再開できるようにします。チェックポイントは、リースのメタデータの一部として DynamoDB のリーステーブルに保存されます。これにより、ワーカーは前のワーカーが処理を停止した位置から処理を続行できます。

KCL における DynamoDB メタデータテーブルと負荷分散

KCL は、リースやワーカーの CPU 使用率メトリクスといったメタデータを管理します。KCL はこれらのメタデータを DynamoDB テーブルを使用して追跡します。Amazon Kinesis Data Streams アプリケーションごとに、KCL はメタデータを管理するために、リーステーブル、ワーカーメトリクステーブル、コーディネーター状態テーブルという 3 つの DynamoDB テーブルを作成します。

> **Note**
>
> KCL 3.x では、新たにワーカーメトリクステーブルとコーディネーター状態テーブルの 2 つのメタデータテーブルが追加されました。

> **Important**
>
> KCL アプリケーションが DynamoDB 内でメタデータテーブルを作成し管理できるようにするためには、適切な権限を付与する必要があります。詳細については、「KCL コンシューマーアプリケーションに必要な IAM アクセス許可」を参照してください。
> KCL コンシューマーアプリケーションは、これら 3 つの DynamoDB メタデータテーブルを自動的には削除しません。不要なコストを回避するため、コンシューマーアプリケーションを廃止するときには、KCL コンシューマーアプリケーションによって作成されたこれらの DynamoDB メタデータテーブルを必ず削除してください。

リーステーブル

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スケジューラによってリースされ、処理されているシャードを追跡するために使用される、専用の Amazon DynamoDB テーブルです。各 KCL コンシューマーアプリケーションは、独自のリーステーブルを作成します。KCL はデフォルトで、コンシューマーアプリケーションの名前をリーステーブル名として使用します。設定を使用してカスタムテーブル名を設定できます。また、KCL は効率的なリース検出のため、リーステーブルに leaseOwner をパーティションキーとするグローバルセカンダリインデックスを作成します。グローバルセカンダリインデックスは、ベースとなるリーステーブルの leaseKey 属性をミラーします。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ご使用のアプリケーションのリーステーブルを作成します。

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。

> **Important**
>
> - 各 KCL コンシューマーアプリケーション名は、リーステーブル名の重複を防ぐために一意でなければなりません。
> - 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。

リーステーブルの各行は、コンシューマーアプリケーションのスケジューラによって処理中のシャードを表します。主要なフィールドは以下のとおりです。

- leaseKey: 単一ストリーム処理の場合、これはシャード ID です。KCL を使用したマルチストリーム処理の場合は、`account-id:StreamName:streamCreationTimestamp:ShardId` の形式で構成されます。leaseKey はリーステーブルのパーティションキーです。マルチストリーム処理の詳細については、KCL を使用したマルチストリーム処理 を参照してください。
- checkpoint: シャードの最新チェックポイントのシーケンス番号。
- checkpointSubSequenceNumber: Kinesis Producer Library の集約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レコードを追跡するチェックポイントの拡張です。
- leaseCounter: ワーカーが現在そのリースを積極的に処理しているかどうかを確認するために使用されます。リースの所有権が別のワーカーに移った場合、leaseCounter は増加します。
- leaseOwner: このリースを現在保持しているワーカー。
- ownerSwitchesSinceCheckpoint: 最後のチェックポイント以降、このリースの担当ワーカーが何回変更されたかを示します。
- parentShardId: このシャードの親シャードの ID。親シャードが完全に処理されるまで子シャードの処理が開始されないようにすることで、レコード処理の正しい順序を維持します。
- childShardId: このシャードの分割または結合によって生成された子シャード ID の一覧。リシャーディング処理におけるシャードの系統を追跡し、処理順序を管理するために使用されます。
- startingHashKey: このシャードのハッシュキー範囲の下限。
- endingHashKey: このシャードのハッシュキー範囲の上限。

マルチストリーム処理を KCL で使用している場合、リーステーブルには次の 2 つの追加フィールドが表示されます。詳細については、「KCL を使用したマルチストリーム処理 」を参照してください。

- shardID: シャードの ID。
- streamName: 以下の形式のデータストリームの識別子: `account-id:StreamName:streamCreationTimestamp`。

ワーカーメトリクステーブル

ワーカーメトリクステーブルは、各 KCL アプリケーションに対して固有の Amazon DynamoDB テーブルであり、各ワーカーからの CPU 使用率メトリクスを記録するために使用されます。これらのメトリクスは、KCL が効率的なリース割り当てを行い、ワーカー間でリソース利用を均等化するために利用されます。KCL は、このワーカーメトリクステーブルの名前としてデフォルトで、`KCLApplicationName-WorkerMetricStats` を使用します。

コーディネーター状態テーブル

コーディネーター状態テーブルは、各 KCL アプリケーションに対して固有の Amazon DynamoDB テーブルであり、ワーカーの内部状態情報を保存するために使用されます。たとえば、コーディネーター状態テーブルには、リーダー選出に関するデータや、KCL 2.x から KCL 3.x へのインプレース移行に関連するメタデータが保存されます。KCL は、このコーディネーター状態テーブルの名前としてデフォルトで `KCLApplicationName-CoordinatorState` を使用します。

KCL が作成するメタデータテーブルの DynamoDB キャパシティモード

Kinesis Client Library (KCL) は、リーステーブル、ワーカーメトリクステーブル、コーディネーター状態テーブルといった DynamoDB のメタデータテーブルを、デフォルトでオンデマンドキャパシティモードで作成します。このモードでは、容量計画を行う必要がなく、トラフィックに応じて読み込み容量および書き込み容量が自動的にスケーリングします。これらのメタデータテーブルをより効率的に運用するため、キャパシティモードはオンデマンドのまま使用することを強くお勧めします。

リーステーブルをプロビジョンドキャパシティモードに切り替える場合は、次のベストプラクティスに従ってください。

- 使用パターンを分析する:
  - Amazon CloudWatch メトリクスを使用して、アプリケーションの読み取りおよび書き込みパターン (RCU、WCU) と使用状況をモニタリングします。
  - ピーク時および平均的なスループット要件を把握します。
- 必要なキャパシティを算出する:
  - 分析結果に基づいて、必要な読み込み容量ユニット (RCU) と書き込み容量ユニット (WCU) を見積もります。
  - シャード数、チェックポイント頻度、ワーカー数といった要素も考慮します。
- 自動スケーリングを実装する:
  - DynamoDB 自動スケーリングを使用して、プロビジョンドキャパシティを自動的に調整し、適切な最小キャパシティの下限および最大キャパシティの上限を設定します。
  - DynamoDB 自動スケーリングを使用することで、KCL のメタデータテーブルがキャパシティ制限に達してスロットリングされる事態を防ぐことができます。
- 定期的なモニタリングと最適化:
  - `ThrottledRequests` に関する CloudWatch メトリクスを継続的にモニタリングします。
  - ワークロードの経時変化に応じて、キャパシティを調整します。

KCL コンシューマーアプリケーションのメタデータ用 DynamoDB テーブルで `ProvisionedThroughputExceededException` が発生した場合は、DynamoDB テーブルのプロビジョンドスループットキャパシティを増やす必要があります。コンシューマーアプリケーションを最初に作成した際に特定の読み込み容量ユニット (RCU) および書き込み容量ユニット (WCU) を設定していても、使用量が増えるにつれてそれだけでは不十分になる可能性があります。たとえば、KCL コンシューマーアプリケーションが頻繁にチェックポイントを実行する場合や、多数のシャードを持つストリームを処理する場合は、より多くの容量ユニットが必要になることがあります。DynamoDB のプロビジョンドスループットについての詳細は、Amazon DynamoDB デベロッパーガイドの DynamoDB のスループットキャパシティとテーブルの更新を参照してください。

KCL がワーカーにリースを割り当て、負荷を分散する方法

KCL は、ワーカーを実行しているコンピューティングホストの CPU 使用率メトリクスを継続的に収集してモニタリングし、ワークロードが均等に分散されるようにします。これらの CPU 使用率メトリクスは、DynamoDB のワーカーメトリクステーブルに保存されます。KCL は、一部のワーカーの CPU 使用率が他と比べて高くなっていることを検出すると、負荷の高いワーカーの負荷を下げるために、ワーカー間でリースを再割り当てします。目的は、コンシューマーアプリケーションフリート全体でワークロードをより均等に分散し、特定のワーカーに過度な負荷が集中するのを防ぐことです。KCL がコンシューマーアプリケーションフリート全体で CPU 使用率を分散するため、適切なワーカー数を選択することでコンシューマーアプリケーションフリートのキャパシティを適正化できます。また、自動スケーリングを使用してコンピューティングキャパシティを効率的に管理し、コストを削減することも可能です。

> **Important**
>
> KCL がワーカーから CPU 使用率メトリクスを収集できるのは、特定の前提条件が満たされている場合のみです。詳細については、「前提条件」を参照してください。KCL がワーカーから CPU 使用率メトリクスを収集できない場合、ワーカーごとのスループットに基づいてリースを割り当て、フリート内のワーカー間で負荷を分散する方式にフォールバックします。KCL は、各ワーカーがある時点で受け取っているスループットをモニタリングし、割り当てられたリースから得られる総スループット量が均等になるようにリースを再割り当てします。

KCL を使用してコンシューマーを開発する

Kinesis データストリームのデータを処理するコンシューマー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。

KCL は、複数の言語で使用できます。このトピックでは、Java および Java 以外の言語で KCL コンシューマーを開発する方法について説明します。

- Kinesis Client Library Javadoc リファレンスを表示するには、Amazon Kinesis Client Library Javadoc を参照してください。
- GitHub から Java 用 KCL をダウンロードするには、Amazon Kinesis Client Library for Java を参照してください。
- Apache Maven で Java 用 KCL を見つけるには、KCL Maven Central Repository を参照してください。

トピック

- Java で KCL を使用してコンシューマーを開発する
- Java 以外の言語で KCL を使用してコンシューマーを開発する

Java で KCL を使用してコンシューマーを開発する

前提条件

KCL 3.x の使用を始める前に、以下のものが揃っていることを確認してください。

- Java Development Kit (JDK) 8 以降
- AWS SDK for Java 2.x
- 依存関係管理用の Maven または Gradle

KCL は、ワーカーが実行されているコンピューティングホストから CPU 使用率などの CPU 使用率メトリクスを収集し、ワーカー間でリソース使用量が均等になるように負荷を分散します。KCL がワーカーから CPU 使用率メトリクスを収集できるようにするには、次の前提条件を満たす必要があります。

Amazon Elastic Compute Cloud(Amazon EC2)

- オペレーティングシステムは Linux OS である必要があります。
- EC2 インスタンスで IMDSv2 を有効にする必要があります。

Amazon Elastic Container Service (Amazon ECS) on Amazon EC2

- オペレーティングシステムは Linux OS である必要があります。
- ECS タスクメタデータエンドポイントのバージョン 4 を有効にする必要があります。
- Amazon ECS コンテナエージェントのバージョンは 1.39.0 以降である必要があります。

での Amazon ECS AWS Fargate

- Fargate タスクメタデータエンドポイントのバージョン 4 を有効にする必要があります。Fargate プラットフォームバージョン 1.4.0 以降を使用している場合、これはデフォルトで有効になっています。
- Fargate プラットフォームバージョン 1.4.0 以降

Amazon Elastic Kubernetes Service (Amazon EKS) on Amazon EC2

- オペレーティングシステムは Linux OS である必要があります。

での Amazon EKS AWS Fargate

- Fargate プラットフォーム 1.3.0 以降

> **Important**
>
> KCL がワーカーから CPU 使用率メトリクスを収集できない場合、ワーカーごとのスループットに基づいてリースを割り当て、フリート内のワーカー間で負荷を分散する方式にフォールバックします。詳細については、「KCL がワーカーにリースを割り当て、負荷を分散する方法」を参照してください。

依存関係をインストールして追加する

Maven を使用している場合は、以下の依存関係を pom.xml ファイルに追加します。3.x.x を最新の KCL バージョンに置き換えていることを確認してください。

```
<dependency>
    <groupId>software.amazon.kinesis</groupId>
    <artifactId>amazon-kinesis-client</artifactId>
    <version>3.x.x</version> <!-- Use the latest version -->
</dependency>
```

Gradle を使用している場合は、以下を build.gradle ファイルに追加します。3.x.x を最新の KCL バージョンに置き換えていることを確認してください。

```
implementation 'software.amazon.kinesis:amazon-kinesis-client:3.x.x'
```

最新の KCL バージョンは、Maven Central Repository で確認できます。

コンシューマーを実装する

KCL コンシューマーアプリケーションは、次の主要コンポーネントで構成されます。

主要コンポーネント

- RecordProcessor
- RecordProcessorFactory
- スケジューラー
- メインコンシューマーアプリケーション

RecordProcessor

RecordProcessor は、Kinesis データストリームのレコードを処理するビジネスロジックが実装される中核コンポーネントです。アプリケーションが Kinesis ストリームから受け取ったデータをどのように処理するかを定義します。

主な役割:

- シャードの処理を初期化する
- Kinesis ストリームからのレコードのバッチを処理する
- シャードの処理をシャットダウンする (シャードが分割または結合された場合、またはリースが別のホストに引き継がれる場合など)
- チェックポイントを処理して進行状況を追跡する

以下に実装例を示します。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
```

```
import software.amazon.kinesis.lifecycle.events.*;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public 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;

    @Override
    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);
        }
    }

    @Override
    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())
            );

            // Checkpoint periodically
            processRecordsInput.checkpointer().checkpoint();
        } catch (Throwable t) {
            log.error("Caught throwable while processing records. Aborting.", t);
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    @Override
    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);
        }
    }

    @Override
    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);
        }
    }

    @Override
    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);
        }
    }
}
```

以下に、この例で使用されている各メソッドの詳細な説明を示します。

initialize(InitializationInput initializationInput)

- 目的: レコードを処理するために必要なリソースや状態を設定します。
- 呼び出されるタイミング: KCL がこのレコードプロセッサにシャードを割り当てたときに 1 回だけ呼び出されます。
- キーポイント:

- `initializationInput.shardId()`: このプロセッサが処理するシャードの ID。
- `initializationInput.extendedSequenceNumber()`: 処理を開始するシーケンス番号。

processRecords(ProcessRecordsInput processRecordsInput)

- 目的: 受信レコードを処理し、必要に応じてチェックポイントの進捗を処理します。
- 呼び出されるタイミング: レコードプロセッサがそのシャードのリースを保持している間、繰り返し呼び出されます。
- キーポイント:
  - `processRecordsInput.records()`: 処理するレコードのリスト。
  - `processRecordsInput.checkpointer()`: チェックポイントの進捗の処理に使用されます。
  - KCL の処理が失敗しないよう、レコード処理中に発生する例外を必ず処理してください。
  - このメソッドはべき等である必要があります。予期しないワーカーのクラッシュや再起動により、チェックポイントされていないレコードが再度処理されるなど、同じレコードが複数回処理される可能性があるためです。
  - データ整合性を確保するため、チェックポイントを実行する前に、バッファされたデータを必ずすべてフラッシュしてください。

leaseLost(LeaseLostInput leaseLostInput)

- 目的: このシャードの処理に固有のリソースをクリーンアップします。
- 呼び出されるタイミング: 別のスケジューラがこのシャードのリースを引き継いだときに呼び出されます。
- キーポイント:
  - このメソッドではチェックポイントを実行できません。

shardEnded(ShardEndedInput shardEndedInput)

- 目的: このシャードおよびチェックポイントの処理を完了します。
- 呼び出されるタイミング: シャードが分割または結合され、このシャードのすべてのデータが処理されたことを示すタイミングで呼び出されます。
- キーポイント:

- `shardEndedInput.checkpointer()`: 最終的なチェックポイント処理を実行するために使用します。
- このメソッドでは、処理を完了するためにチェックポイントの実行が必須です。
- ここでフラッシュ処理とチェックポイントを行わないと、シャードが再度オープンされた際に、データ損失や重複処理が発生する可能性があります。

shutdownRequested(ShutdownRequestedInput shutdownRequestedInput)

- 目的: KCL のシャットダウン時にチェックポイントを実行し、リソースをクリーンアップします。
- 呼び出されるタイミング: KCL がシャットダウンする際 (例: アプリケーションが終了するとき) に呼び出されます。
- キーポイント:
  - `shutdownRequestedInput.checkpointer()`: シャットダウン前にチェックポイントを実行するために使用します。
  - アプリケーションが停止する前に進捗が保存されるよう、このメソッド内でチェックポイント処理を実装していることを確認してください。
  - ここでデータのフラッシュとチェックポイントを行わないと、アプリケーションの再起動時にデータ損失やレコードの再処理が発生する可能性があります。

> **Important**
>
> KCL 3.x では、前のワーカーがシャットダウンされる前にチェックポイントを実行することで、リースが別のワーカーに引き継がれてもデータの再処理が最小限で済むようになっています。`shutdownRequested()` メソッドにチェックポイントロジックを実装していない場合、このメリットは得られません。`shutdownRequested()` メソッド内にチェックポイントロジックが実装されていることを確認してください。

RecordProcessorFactory

RecordProcessorFactory は、新しい RecordProcessor インスタンスを作成する役割を担います。KCL は、このファクトリを使用して、アプリケーションが処理する必要のある各シャードに対して新しい RecordProcessor を作成します。

主な役割:

- 必要に応じて新しい RecordProcessor インスタンスを作成する
- 各 RecordProcessor が正しく初期化されていることを保証する

以下に実装例を示します。

```
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleRecordProcessorFactory implements ShardRecordProcessorFactory {
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new SampleRecordProcessor();
    }
}
```

この例では、ファクトリは shardRecordProcessor() が呼び出されるたびに新しい SampleRecordProcessor を作成します。必要に応じて、初期化ロジックを追加する形で拡張できます。

スケジューラー

スケジューラは、KCL アプリケーション内のすべての動作を調整する高レベルのコンポーネントです。データ処理全体のオーケストレーションを担います。

主な役割:

- RecordProcessor のライフサイクルを管理する
- シャードのリース管理を処理する
- チェックポイント処理を調整する
- アプリケーション内の複数ワーカー間でシャード処理の負荷を分散する
- 正常なシャットダウンおよびアプリケーション終了シグナルを処理する

スケジューラは通常、メインアプリケーション内で作成および開始されます。スケジューラの実装例は、この後の「メインコンシューマーアプリケーション」セクションで確認できます。

メインコンシューマーアプリケーション

メインコンシューマーアプリケーションは、すべてのコンポーネントを結び付ける役割を果たします。KCL コンシューマーのセットアップ、必要なクライアントの作成、スケジューラの構成、およびアプリケーションのライフサイクル管理を担当します。

主な役割:

- AWS サービスクライアントの設定 (Kinesis、DynamoDB、CloudWatch)
- KCL アプリケーションを構成する
- スケジューラを作成して起動する
- アプリケーションのシャットダウン処理を行う

以下に実装例を示します。

```
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.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import java.util.UUID;

public class SampleConsumer {
    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    public SampleConsumer(String streamName, Region region) {
        this.streamName = streamName;
        this.region = region;
        this.kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    public void run() {
        DynamoDbAsyncClient dynamoDbAsyncClient =
 DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();
```

```
        ConfigsBuilder configsBuilder = new ConfigsBuilder(
            streamName,
            streamName,
            kinesisClient,
            dynamoDbAsyncClient,
            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();
    }

    public static void main(String[] args) {
        String streamName = "your-stream-name"; // replace with your stream name
        Region region = Region.US_EAST_1; // replace with your region
        new SampleConsumer(streamName, region).run();
    }
}
```

KCL は、デフォルトで専用スループットを持つ拡張ファンアウト (EFO) コンシューマーを作成します。拡張ファンアウトの詳細については、専用スループットを備えた拡張ファンアウトを開発するを参照してください。コンシューマーが 2 未満の場合、または 200 ミリ秒未満の読み取り伝搬遅延を必要としない場合は、共有スループットコンシューマーを使用するために、スケジューラオブジェクトで次の設定を行う必要があります。

```
configsBuilder.retrievalConfig().retrievalSpecificConfig(new PollingConfig(streamName,
 kinesisClient))
```

次のコードは、共有スループットコンシューマーを使用するスケジューラオブジェクトを作成する例です。

インポート:

```
import software.amazon.kinesis.retrieval.polling.PollingConfig;
```

コード:

```
Scheduler scheduler = new Scheduler(
            configsBuilder.checkpointConfig(),
            configsBuilder.coordinatorConfig(),
            configsBuilder.leaseManagementConfig(),
            configsBuilder.lifecycleConfig(),
            configsBuilder.metricsConfig(),
            configsBuilder.processorConfig(),
            configsBuilder.retrievalConfig().retrievalSpecificConfig(new
 PollingConfig(streamName, kinesisClient))
        );/
```

Java 以外の言語で KCL を使用してコンシューマーを開発する

このセクションでは、Python、Node.js、.NET、Ruby などの言語で Kinesis Client Library (KCL) を使用してコンシューマーを実装する方法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、`MultiLangDaemon` と呼ばれる多言語インターフェースを使用して提供されます。このデーモンは Java ベースであり、Java 以外の言語で KCL を使用する場合、バックグラウンドで実行されます。そのため、KCL を Java以外の言語向けにインストールし、コンシューマーアプリケーションを完全に Java 以外の言語で記述している場合でも、`MultiLangDaemon` を使用するために、システムに Java をインストールしておく必要があります。さらに、`MultiLangDaemon` にはデフォルト設定があり、ユースケースによってはカスタマイズが必要になる場合があります (接続先の AWS リージョンなど)。GitHub の `MultiLangDaemon` の詳細については、KCL MultiLangDaemon project を参照してください。

主要な概念はどの言語でも共通ですが、言語ごとに特有の考慮点や実装があります。KCL コンシューマー開発の基本概念については、Java で KCL を使用してコンシューマーを開発するを参照してください。Python、Node.js、.NET、Ruby で KCL コンシューマーを開発する方法の詳細および最新情報については、次の GitHub リポジトリを参照してください。

- Python: amazon-kinesis-client-python

- Node.js: amazon-kinesis-client-nodejs
- .NET: amazon-kinesis-client-net
- Ruby: amazon-kinesis-client-ruby

> **Important**
>
> JDK 8 を使用している場合、以下の Java 以外の言語向け KCL ライブラリのバージョンは使用しないでください。これらのバージョンには、JDK 8 と互換性のない依存関係 (logback) が含まれています。
>
> - KCL Python 3.0.2 および 2.2.0
> - KCL Node.js 2.3.0
> - KCL .NET 3.1.0
> - KCL Ruby 2.2.0
>
> JDK 8 を使用する場合は、問題が発生するこれらのバージョンより前または後にリリースされたバージョンを使用することをお勧めします。

KCL を使用したマルチストリーム処理

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するために必要な KCL 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 KCL 2.3 以降でのみサポートされています。
> - `multilangdaemon` を使用して実行される、Java 以外の言語で記述された KCL コンシューマーでは、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 インターフェイス
  - 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、MultistreamTracker という新しいインターフェイスを実装する必要があります。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。

- streamConfigList が StreamConfig リストに入力します。

```
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 は省略可能です。KCL 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/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L129 を参照してください。StreamIdentifier を作成するには、streamArn および KCL 2.5.0 以降で利用可能な streamCreationEpoch 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。streamArm をサポートしていない KCL v2.3 および v2.4 では、account-id:StreamName:streamCreationTimestamp 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。この形式は廃止され、次のメジャーリリース以降は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 (formerStreamsLeasesDeletionStrategy) も含まれています。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/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java を参照してください。

•

または、複数のストリームを同時に処理する KCL コンシューマーアプリケーションを実装する場合、MultiStreamTracker で ConfigsBuilder を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るようになりました。
- 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。

> **Important**
>
> 既存の 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ように構成されている場合、`leaseKey` (リーステーブルのパーティションキー) 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。これは、マルチストリームをサポートするためには、`leaseKey` の構造が次の形式である必要があるためです: `account-id:StreamName:StreamCreationTimestamp:ShardId`。

KCL で AWS Glue スキーマレジストリを使用する

Kinesis Data Streams 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、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しながら、スキーマを一元的に検出、制御、および進化させることができます。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 Schema Registry」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**
>
> - AWS Glue Kinesis Data Streams のスキーマレジストリ統合は、KCL 2.3 以降でのみサポートされています。
> - AWS Glue Kinesis Data Streams のスキーマレジストリ統合は、 で実行される Java 以外の言語で記述された KCL コンシューマーではサポートされていません`multilangdaemon`。
> - AWS Glue Kinesis Data Streams のスキーマレジストリ統合は、KCL 1.x のどのバージョンでもサポートされていません。

KCL を使用して Kinesis Data Streams と AWS Glue 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。

KCL コンシューマーアプリケーションに必要な IAM アクセス許可

KCL コンシューマーアプリケーションに関連付けられている IAM ロールまたはユーザーに、次の許可を追加する必要があります。

のセキュリティのベストプラクティス AWS により、きめ細かなアクセス許可を使用してさまざまなリソースへのアクセスを制御できます。 AWS Identity and Access Management (IAM) では、 でユーザーとユーザーのアクセス許可を管理できます AWS。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。

次の表は、KCL コンシューマーアプリケーションに一般的に必要となる最小限の IAM アクセス許可を示しています。

KCL コンシューマーアプリケーションの最小限の IAM アクセス許可

サービス	アクション	リソース (ARN)	目的

Amazon Kinesis Data Streams	`DescribeStream`
	
	`DescribeStreamSummary`
	
	`RegisterStreamConsumer`
		KCL アプリケーションがデータを処理する Kinesis データストリーム。
		
		`arn:aws:kinesis:region:account:stream/StreamName`
			レコードを読み取る前に、コンシューマーは、データストリームが存在すること、アクティブであること、シャードを含んでいることを確認します。
			
			コンシューマーをシャードに登録します。

Amazon Kinesis Data Streams	`GetRecords`
	
	`GetShardIterator`
	
	`ListShards`
		KCL アプリケーションがデータを処理する Kinesis データストリーム。
		
		`arn:aws:kinesis:region:acco`
			シャードからレコードを読み取ります。

サービス	アクション	リソース (ARN)	目的
		`unt:stream/StreamName`	

Amazon Kinesis Data Streams	`SubscribeToShard`
	
	`DescribeStreamConsumer`
		KCL アプリケーションがデータを処理する Kinesis データストリーム。このアクションは、拡張ファンアウト (EFO) コンシューマーを使用する場合にのみ追加します。
		
		`arn:aws:kinesis:region:account:stream/StreamName/consumer/*`
			拡張ファンアウト (EFO) コンシューマー用にシャードをサブスクライブします。

Amazon DynamoDB	`CreateTable`
	
	`DescribeTable`
	
	`UpdateTable`
	
	`Scan`
	
	`GetItem`
	
	`PutItem`
	
	`UpdateItem`
	
	`DeleteItem`
		リーステーブル (KCL によって作成された DynamoDB のメタデータテーブル。
		
		`arn:aws:dynamodb:region:account:table/KCLApplicationName`
			これらのアクションは、KCL が DynamoDB で作成されたリーステーブルを管理するために必要です。

サービス	アクション	リソース (ARN)	目的

Amazon DynamoDB	`CreateTable`
	
	`DescribeTable`
	
	`Scan`
	
	`GetItem`
	
	`PutItem`
	
	`UpdateItem`
	
	`DeleteItem`
		KCL によって作成されたワーカーメトリクスとコーディネーター状態テーブル (DynamoDB のメタデータテーブル)。
		
		`arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats`
		
		`arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorState`
			KCL が DynamoDB でワーカーメトリクスとコーディネーター状態メタデータテーブルを管理するには、これらのアクションが必要です。

サービス	アクション	リソース (ARN)	目的
Amazon DynamoDB	`Query`	リーステーブルのグローバルセカンダリインデックス `arn:aws:dynamodb:region:account:table/KCLApplicationName/index/*`	KCL が DynamoDB で作成されたリーステーブルのグローバルセカンダリインデックスを読み取るには、このアクションが必要です。
Amazon CloudWatch	`PutMetricData`	*	アプリケーションをモニタリングするのに便利なメトリクスを CloudWatch にアップロードします。CloudWatch では `PutMetricData` アクションが実行される特定のリソースが存在しないため、アスタリスク (*) が使用されています。

> **Note**
>
> ARNsStreamName」、KCLApplicationName」をそれぞれ独自の AWS リージョン名前、AWS アカウント 数値、Kinesis データストリーム名、および KCL アプリケーション名に置き換えます。KCL 3.x は、DynamoDB にさらに 2 つのメタデータテーブルを作成します。KCL によって作成された DynamoDB メタデータテーブルの詳細については、KCL における DynamoDB メタデータテーブルと負荷分散 を参照してください。KCL が作成するメタデータテーブルの名前を設定でカスタマイズしている場合は、KCL アプリケーション名ではなく、その指定したテーブル名を使用してください。

以下は、KCL コンシューマーアプリケーションのサンプルポリシードキュメントです。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream",
                "kinesis:DescribeStreamSummary",
                "kinesis:RegisterStreamConsumer",
                "kinesis:GetRecords",
                "kinesis:GetShardIterator",
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws:kinesis:us-
east-1:123456789012:stream/STREAM_NAME"
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": "arn:aws:kinesis:us-
east-1:123456789012:stream/STREAM_NAME/consumer/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:DescribeTable",
                "dynamodb:UpdateTable",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:DeleteItem",
                "dynamodb:Scan"
            ],
            "Resource": [
            "arn:aws:dynamodb:us-east-1:123456789012:table/KCL_APPLICATION_NAME"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:DescribeTable",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:DeleteItem",
                "dynamodb:Scan"
            ],
            "Resource": [
            "arn:aws:dynamodb:us-east-1:123456789012:table/KCL_APPLICATION_NAME-
WorkerMetricStats",
    "arn:aws:dynamodb:us-east-1:123456789012:table/KCL_APPLICATION_NAME-
CoordinatorState"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query"
            ],
            "Resource": [
            "arn:aws:dynamodb:us-east-1:123456789012:table/KCL_APPLICATION_NAME/
index/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": "*"
        }
    ]
}
```

このサンプルポリシーを使用する前に、次の項目を確認してください。

- REGION を your AWS リージョン (us-east-1 など) に置き換えます。

- ACCOUNT_ID を AWS アカウント ID に置き換えます。
- STREAM_NAME を自分の Kinesis データストリーム名に置き換えます。
- CONSUMER_NAME を自分のコンシューマー名に置き換えます (KCL を使用する場合、通常はアプリケーション名)。
- KCL_APPLICATION_NAME を自分の KCL アプリケーション名に置き換えます。

KCL の設定

特定の要件に合わせて Kinesis Client Library の機能をカスタマイズできるよう、設定プロパティを指定できます。次の表に、設定プロパティとクラスを示します。

> **Important**
>
> KCL 3.x の負荷分散アルゴリズムは、ワーカーごとのリース数を均等にすることではなく、ワーカー間の CPU 使用率を均等化することを目的としています。`maxLeasesForWorker` を低く設定しすぎると、KCL のワークロードを効果的に分散する能力が制限される可能性があります。`maxLeasesForWorker` 設定を使用する場合は、可能な限り最適な負荷分散が行えるよう、その値を高めに設定することを検討してください。

この表は、KCL の設定プロパティを示しています。

設定プロパティ	設定クラス	説明	デフォルトの 値
`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。	該当しない
`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。	該当しない

設定プロパティ	設定クラス	説明	デフォルトの 値
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。	該当しない
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。	該当しない
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)。シャードの数が多いアプリケーションでは、リースの追跡に必要な DynamoDB の IOPS を減らすために、この値をより大きく設定する場合があります。	10,000 (10 秒)
shardSyncIntervalMillis	LeaseManagementConfig	シャード同期コールの間隔。	60,000 (60 秒)
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。	TRUE

設定プロパティ	設定クラス	説明	デフォルトの 値
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。	FALSE
maxLeasesForWorker	LeaseManagementConfig	1 つのワーカーが受け持つことができるリースの最大数です。この値を低く設定しすぎると、ワーカーがすべてのシャードを処理できず、データ損失を招く可能性があります。また、ワーカー間のリース割り当てが最適にならない原因にもなります。設定する際は、シャード合計数、ワーカー数、ワーカーの処理能力を考慮してください。	無制限
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。	20

設定プロパティ	設定クラス	説明	デフォルトの 値
billingMode	LeaseManagementConfig	DynamoDB に作成されるリーステーブルのキャパシティモードを決定します。選択肢はオンデマンドモード (PAY_PER_REQUEST) とプロビジョンドモードの 2 種類です。容量計画を行う必要がなく、ワークロードに応じて自動的にスケールするため、デフォルト設定であるオンデマンドモードの使用をお勧めします。	PAY_PER_REQUEST (オンデマンドモード)
initialLeaseTableReadCapacity	LeaseManagementConfig	Kinesis Client Library がプロビジョンドキャパシティモードで新しい DynamoDB リーステーブルを作成する必要がある場合に使用される DynamoDB の読み込み容量です。billingMode 設定でデフォルトのオンデマンドキャパシティモードを使用している場合、この設定は無視して構いません。	10

設定プロパティ	設定クラス	説明	デフォルトの 値
initialLeaseTableWriteCapacity	LeaseManagementConfig	Kinesis Client Library で新しい DynamoDB リーステーブルを作成する必要がある場合に使用される DynamoDB 読み込み容量です。billingMode 設定でデフォルトのオンデマンドキャパシティモードを使用している場合、この設定は無視して構いません。	10
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。	InitialPositionInStream.TRIM_HORIZON
reBalanceThresholdPercentage	LeaseManagementConfig	負荷分散アルゴリズムがワーカー間でシャードの再割り当てを検討するタイミングを決定する、パーセンテージ値です。 これは、KCL 3.x で導入された新しい設定です。	10

設定プロパティ	設定クラス	説明	デフォルトの 値
dampening Percentage	LeaseMana gementConfig	1 回のリバランス処理で、過負荷状態のワーカーから移動させる負荷量を抑制するために使用されるパーセンテージ値です。 これは、KCL 3.x で導入された新しい設定です。	60
allowThro ughputOve rshoot	LeaseMana gementConfig	希望するスループット量を超えることになったとしても、過負荷状態のワーカーから追加のリースを取得する必要があるかどうかを決定します。 これは、KCL 3.x で導入された新しい設定です。	TRUE

設定プロパティ	設定クラス	説明	デフォルトの 値
`disableWorkerMetrics`	LeaseManagementConfig	リースの再割り当てや負荷分散を行う際に、KCL がワーカーのリソースメトリクス (CPU 使用率など) を無視するかどうかを指定します。KCL に CPU 使用率に基づく負荷分散を行わせたくない場合は、この値を TRUE に設定してください。 これは、KCL 3.x で導入された新しい設定です。	FALSE
`maxThroughputPerHostKBps`	LeaseManagementConfig	リース割り当て中にワーカーに割り当てる最大スループットの量。 これは、KCL 3.x で導入された新しい設定です。	無制限

設定プロパティ	設定クラス	説明	デフォルトの 値
`isGracefulLeaseHandoffEnabled`	LeaseManagementConfig	ワーカー間でリースを引き継ぐ際の動作を制御します。この値を true に設定すると、KCL はリースを別のワーカーに引き渡す前に、そのシャードの RecordProcessor が処理を完了できる十分な時間を確保し、リースをスムーズに引き継ぐよう試みます。これによりデータの整合性やスムーズな引き継ぎが可能になりますが、リースの引き継ぎに要する時間が長くなる場合があります。 この値を false に設定した場合、RecordProcessor が正常にシャットダウンするのを待たずに、リースは即座に引き渡されます。これによりリースの引き継ぎは高速になりますが、処理が完了しないまま中断されるリスクがあります。	TRUE

設定プロパティ	設定クラス	説明	デフォルトの 値
		注: スムーズなリース引き継ぎ機能の恩恵を受けるためには、RecordProcessor の shutdownRequested() メソッド内にチェックポイント処理を実装しておく必要があります。 これは、KCL 3.x で導入された新しい設定です。	

設定プロパティ	設定クラス	説明	デフォルトの 値
gracefulLeaseHandoffTimeoutMillis	LeaseManagementConfig	現在のシャードの RecordProcessor が適切にシャットダウンするのを待つための最小待機時間 (ミリ秒) を指定します。この時間が経過すると、次の所有者へのリースが強制的に引き継がれます。 processRecords メソッドの処理時間が通常のデフォルト値より長い場合は、この設定値を大きくすることを検討してください。これにより、リースが引き継がれる前に RecordProcessor が処理を完了できる十分な時間を確保できます。 これは、KCL 3.x で導入された新しい設定です。	30,000 (30 秒)
maxRecords	PollingConfig	Kinesis が返すレコードの最大数の設定を許可します。	10,000

設定プロパティ	設定クラス	説明	デフォルトの 値
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。	なし
maxGetRecordsThreadPool	PollingConfig	GetRecords に使用されるスレッドプールのサイズ。	なし
idleTimeBetweenReadsInMillis	PollingConfig	KCL がデータストリームからデータを取得するために GetRecords を呼び出す際、その呼び出し間隔として待機する時間を指定します。単位はミリ秒です。	1,500
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。	FALSE
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。単位はミリ秒です。	10,000 (10 秒)
skipShardSyncAtWorkerInitializationIfLeaseExist	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。	FALSE

設定プロパティ	設定クラス	説明	デフォルトの 値
shardPrioritization	CoordinatorConfig	どのシャードの優先順位付けを使用するか。	NoOpShardPrioritization
ClientVersionConfig	CoordinatorConfig	アプリケーションをどの KCL バージョン互換モードで実行するかを決定します。この設定は、旧バージョンの KCL から移行する場合にのみ使用されます。3.x へ移行する際は、この設定を CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X に設定する必要があります。移行が完了したら、この設定は削除できます。	CLIENT_VERSION_CONFIG_3X
taskBackoffTimeMillis	LifecycleConfig	失敗した KCL タスクを再試行するまでの待機時間。単位はミリ秒です。	500 (0.5 秒)
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。	なし

設定プロパティ	設定クラス	説明	デフォルトの 値
listShardsBackoffTimeInMillis	RetrievalConfig	障害が発生した場合に ListShards を呼び出す間隔 (ミリ秒)。単位はミリ秒です。	1,500 (1.5 秒)
maxListShardsRetryAttempts	RetrievalConfig	失敗とみなすまでの ListShards の再試行の最大回数。	50
metricsBufferTimeMillis	MetricsConfig	メトリクスを CloudWatch に送信する前に、バッファリングして保持する最大時間 (ミリ秒) を指定します。	10,000 (10 秒)
metricsMaxQueueSize	MetricsConfig	メトリクスを CloudWatch に送信する前に、バッファリングして保持できるメトリクスの最大数を指定します。	10,000
metricsLevel	MetricsConfig	有効化して CloudWatch に送信するメトリクスの詳細度を指定します。 指定できる値: NONE、SUMMARY、DETAILED。	MetricsLevel.DETAILED

設定プロパティ	設定クラス	説明	デフォルトの 値
metricsEnabledDimensions	MetricsConfig	CloudWatch メトリクスで許可されるディメンションを制御します。	すべてのディメンション

KCL 3.x で廃止された設定

KCL 3.x では、次の設定プロパティは廃止されています。

この表は、KCL 3.x の廃止された設定プロパティを示しています。

設定プロパティ	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。KCL 3.x ではこの設定は無視され、ワーカーのリソース使用状況に基づいてリースが再割り当てされます。
enablePriorityLeaseAssignment	LeaseManagementConfig	ワーカーが、ターゲットとなるリース数に関係なく、フェイルオーバー時間の 3 倍の期間更新されていない大幅に期限切れのリースや新しいシャードのリースを優先して取得するかどうかを制御します。ただし、最大リース数の制限は引き続き適用されます。KCL 3.x ではこの設定は無視され、期限切れのリースは常に複数のワーカーへ分散されます。

> **Important**
>
> 旧バージョンの KCL から KCL 3.x へ移行する間は、廃止された設定プロパティも引き続き保持しておく必要があります。移行期間中、KCL ワーカーはまず KCL 2.x 互換モードで起動し、アプリケーション内のすべての KCL ワーカーが KCL 3.x を実行できる状態であることを検出すると、KCL 3.x の機能モードへ切り替わります。KCL ワーカーが KCL 2.x 互換モードで動作している間は、これらの廃止された設定が必要になります。

KCL バージョンライフサイクルポリシー

このトピックでは、Amazon Kinesis Client Library (KCL) のバージョンライフサイクルポリシーの概要を説明します。 では、新機能と機能強化、バグ修正、セキュリティパッチ、依存関係の更新をサポートするために、KCL バージョンの新しいリリース AWS を定期的に提供しています。最新の機能、セキュリティ更新、基本的な依存関係に対応するため、KCL を常に最新バージョンへ更新しておくことをお勧めします。サポートが終了した KCL バージョンを継続使用することはお勧めしません。

主要な KCL バージョンのライフサイクルは、次の 3 つのフェーズで構成されます。

- 一般提供 (GA) – このフェーズでは、メジャーバージョンが完全にサポートされています。は、Kinesis Data Streams の新機能や API アップデートのサポート、バグやセキュリティの修正を含む、定期的なマイナーバージョンとパッチバージョンのリリース AWS を提供します。
- メンテナンスモード – パッチバージョンのリリース AWS を制限して、重大なバグ修正とセキュリティの問題にのみ対処します。Kinesis Data Streams の新機能や API に対する更新は、このメジャーバージョンには行われません。
- サポート終了 – そのメジャーバージョンに対して更新やリリースは一切提供されません。以前に公開されたリリースは引き続き公開パッケージマネージャーから入手でき、コードは GitHub に残ります。サポートが終了したバージョンの使用は、ユーザーの裁量で行われます。最新のメジャーバージョンにアップグレードすることをお勧めします。

メジャーバージョン	現在のフェーズ	リリース日	メンテナンスモード日	サポート終了日
KCL 1.x	メンテナンスモード	2013-12-19	2025-04-17	2026-01-30

メジャーバージョン	現在のフェーズ	リリース日	メンテナンスモード日	サポート終了日
KCL 2.x	一般提供	2018-08-02	--	--
KCL 3.x	一般提供	2024-11-06	--	--

旧バージョンの KCL からの移行

このトピックでは、Kinesis Client Library (KCL) の旧バージョンから移行する方法について説明します。

KCL 3.0 の新機能

Kinesis Client Library (KCL) 3.0 では、これまでのバージョンと比べて複数の大幅な強化が行われています。

- コンシューマーアプリケーションのフリート内で、過負荷のワーカーから負荷の低いワーカーへ処理を自動的に再分配することで、アプリケーションのコンピューティングコストを削減します。この新しい負荷分散アルゴリズムにより、ワーカー間の CPU 使用率が均等に分散されるため、ワーカーの過剰プロビジョニングが不要になります。
- リーステーブルに対する読み取りオペレーションを最適化することで、KCL に関連する DynamoDB のコストを削減します。
- 現在のワーカーが処理済みレコードのチェックポイントを完了できるようにすることで、リースが別のワーカーに引き継がれる際のデータ再処理を最小限に抑えます。
- を使用してパフォーマンスとセキュリティ機能 AWS SDK for Java 2.x を向上させ、 AWS SDK for Java 1.x への依存を完全に排除します。

詳細については、KCL 3.0 リリースノートを参照してください。

トピック

- KCL 2.x から KCL 3.x に移行する
- 以前のバージョンにロールバックする
- ロールバック後に KCL 3.x にロールフォワードする
- プロビジョンドキャパシティモードを使用するリーステーブルのベストプラクティス
- KCL 1.x から KCL 3.x への移行

KCL 2.x から KCL 3.x に移行する

このトピックでは、KCL 2.x のコンシューマーを KCL 3.x へ移行するための手順を段階的に説明します。KCL 3.x は、KCL 2.x コンシューマーのインプレース移行をサポートしています。ワーカーをローリング方式で順次移行している間も、Kinesis データストリームからのデータ取得を継続できます。

> **Important**
>
> KCL 3.x は KCL 2.x と同じインターフェイスおよびメソッドを維持しています。したがって、移行中にレコード処理コードを更新する必要はありません。ただし、適切な設定を行い、移行に必要な手順を確認する必要があります。スムーズに移行するために、以下の移行手順に従うことを強くお勧めします。

ステップ 1: 前提条件

KCL 3.x の使用を始める前に、以下のものが揃っていることを確認してください。

- Java Development Kit (JDK) 8 以降
- AWS SDK for Java 2.x
- 依存関係管理用の Maven または Gradle

> **Important**
>
> KCL 3.x で AWS SDK for Java バージョン 2.27.19 ~ 2.27.23 を使用しないでください。これらのバージョンには、KCL が使用する DynamoDB に関連した例外エラーを引き起こす問題が含まれています。この問題を回避するには、 AWS SDK for Java バージョン 2.28.0 以降を使用することをお勧めします。

ステップ 2: 依存関係を追加する

Maven を使用している場合は、以下の依存関係を `pom.xml` ファイルに追加します。3.x.x を最新の KCL バージョンに置き換えていることを確認してください。

```
<dependency>
```

```
    <groupId>software.amazon.kinesis</groupId>
    <artifactId>amazon-kinesis-client</artifactId>
    <version>3.x.x</version> <!-- Use the latest version -->
</dependency>
```

Gradle を使用している場合は、以下を build.gradle ファイルに追加します。3.x.x を最新の KCL バージョンに置き換えていることを確認してください。

```
implementation 'software.amazon.kinesis:amazon-kinesis-client:3.x.x'
```

最新の KCL バージョンは、Maven Central Repository で確認できます。

ステップ 3: 移行関連の設定をセットアップする

KCL 2.x から KCL 3.x に移行するには、次の設定パラメータを設定する必要があります。

- CoordinatorConfig.clientVersionConfig: この設定は、アプリケーションをどの KCL バージョン互換モードで実行するかを決定します。KCL 2.x から 3.x へ移行する際は、この設定を CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X に設定する必要があります。この設定を行うには、スケジューラオブジェクトを作成する際に次の行を追加してください。

```
configsBuilder.coordiantorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONFI
```

以下は、KCL 2.x から 3.x へ移行する際に、CoordinatorConfig.clientVersionConfig を設定する方法の例です。必要に応じて、その他の設定も要件に合わせて調整できます。

```
Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),

 configsBuilder.coordiantorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONF
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
);
```

KCL 2.x と KCL 3.x では負荷分散アルゴリズムが異なるため、コンシューマーアプリケーション内のすべてのワーカーが、同じ時点では同一の負荷分散アルゴリズムを使用していることが重要です。異

なる負荷分散アルゴリズムを使用するワーカーが同時に動作していると、両アルゴリズムが独立して動作するため、負荷分散が最適化されない可能性があります。

この KCL 2.x 互換設定により、KCL 3.x アプリケーションは KCL 2.x と互換性のあるモードで動作し、コンシューマーアプリケーション内のすべてのワーカーが KCL 3.x にアップグレードされるまで、KCL 2.x の負荷分散アルゴリズムを利用できます。移行が完了すると、KCL は自動的に完全な KCL 3.x の機能モードへ切り替わり、稼働中のすべてのワーカーで新しい KCL 3.x の負荷分散アルゴリズムを使用し始めます。

> **Important**
>
> `ConfigsBuilder` を使用せず、設定を行うために `LeaseManagementConfig` オブジェクトを作成している場合は、KCL 3.x 以降では `applicationName` という追加のパラメータを指定する必要があります。詳細については、LeaseManagementConfig コンストラクタでのコンパイルエラーを参照してください。KCL の設定には、`ConfigsBuilder` を使用することをお勧めします。`ConfigsBuilder` は、KCL アプリケーションをより柔軟かつ保守しやすい方法で設定できる手段を提供します。

ステップ 4: shutdownRequested() メソッド実装のベストプラクティスに従う

KCL 3.x では、リース再割り当て時にリースが別のワーカーへ引き継がれる際のデータ再処理を最小限に抑えるため、スムーズなリース引き継ぎと呼ばれる機能が導入されています。これは、リースが引き継がれる前に、処理済みの最新シーケンス番号をリーステーブルにチェックポイントすることで実現されます。スムーズなリース引き継ぎを正しく機能させるためには、`RecordProcessor` クラスの `shutdownRequested` メソッド内で、`checkpointer` オブジェクトを必ず呼び出すようにする必要があります。`shutdownRequested` メソッド内で `checkpointer` オブジェクトを呼び出さない場合は、次の例に示すように実装できます。

> **Important**
>
> - 次の実装例は、スムーズなリース引き継ぎの最小要件です。必要に応じて、チェックポイントに関連する追加のロジックを含めるように拡張できます。非同期処理を実行している場合は、チェックポイントを呼び出す前に、ダウンストリームに配信されたすべてのレコードが処理されていることを確認してください。
> - スムーズなリース引き継ぎによって、リース移行時のデータ再処理が発生する可能性は大幅に低減されますが、その可能性が完全になくなるわけではありません。データの整合性

と一貫性を維持するには、ダウンストリームのコンシューマーアプリケーションをべき等になるように設計します。つまり、重複してレコードを処理する可能性があっても、システム全体に悪影響を及ぼさずに処理できるようにしておく必要があります。

```
/**
 * Invoked when either Scheduler has been requested to gracefully shutdown
 * or lease ownership is being transferred gracefully so the current owner
 * gets one last chance to checkpoint.
 *
 * 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) {
    try {
       // Ensure that all delivered records are processed
       // and has been successfully flushed to the downstream before calling
       // checkpoint
       // If you are performing any asynchronous processing or flushing to
       // downstream, you must wait for its completion before invoking
       // the below checkpoint method.
        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);
    }
}
```

ステップ 5: KCL 3.x でワーカーメトリクスを収集するための前提条件を確認する

KCL 3.x は、ワーカー間で負荷を均等に分散するため、CPU 使用率などの CPU 利用メトリクスを収集します。コンシューマーアプリケーションのワーカーは、Amazon EC2、Amazon ECS、Amazon EKS、または AWS Fargate上で実行できます。KCL 3.x がワーカーから CPU 使用率メトリクスを収集できるのは、以下の前提条件が満たされている場合のみです。

Amazon Elastic Compute Cloud(Amazon EC2)

- オペレーティングシステムは Linux OS である必要があります。

- EC2 インスタンスで IMDSv2 を有効にする必要があります。

Amazon Elastic Container Service (Amazon ECS) on Amazon EC2

- オペレーティングシステムは Linux OS である必要があります。
- ECS タスクメタデータエンドポイントのバージョン 4 を有効にする必要があります。
- Amazon ECS コンテナエージェントのバージョンは 1.39.0 以降である必要があります。

での Amazon ECS AWS Fargate

- Fargate タスクメタデータエンドポイントのバージョン 4 を有効にする必要があります。Fargate プラットフォームバージョン 1.4.0 以降を使用している場合、これはデフォルトで有効になっています。
- Fargate プラットフォームバージョン 1.4.0 以降

Amazon Elastic Kubernetes Service (Amazon EKS) on Amazon EC2

- オペレーティングシステムは Linux OS である必要があります。

での Amazon EKS AWS Fargate

- Fargate プラットフォーム 1.3.0 以降

> **Important**
>
> 前提条件が満たされず、KCL 3.x がワーカーから CPU 使用率メトリクスを収集できない場合、KCL はリースごとのスループット量に基づいて負荷を再分散します。このフォールバックの負荷再分散メカニズムにより、各ワーカーに割り当てられたリースから得られる合計スループット量が、ワーカー間で均等になるように調整されます。詳細については、「KCL がワーカーにリースを割り当て、負荷を分散する方法」を参照してください。

ステップ 6: KCL 3.x 用に IAM アクセス許可を更新する

KCL 3.x コンシューマーアプリケーションに関連付けられている IAM ロールまたは IAM ポリシーに、次の許可を追加する必要があります。これは、KCL アプリケーションで使用中の既存の IAM ポ

リシーを更新する作業を伴います。詳細については、「KCL コンシューマーアプリケーションに必要な IAM アクセス許可」を参照してください。

> **Important**
>
> 既存の KCL アプリケーションでは、KCL 2.x では不要であったため、IAM ポリシーに次の IAM アクションおよびリソースが追加されていない可能性があります。KCL 3.x アプリケーションを実行する前に、これらが追加されていることを必ず確認してください。
>
> - アクション: `UpdateTable`
>   - リソース (ARN): `arn:aws:dynamodb:region:account:table/KCLApplicationName`
> - アクション: `Query`
>   - リソース (ARN): `arn:aws:dynamodb:region:account:table/KCLApplicationName/index/*`
> - アクション: `CreateTable`、`DescribeTable`、`Scan`、`GetItem`、`PutItem`、`UpdateItem`、`DeleteItem`
>   - リソース (ARN): `arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats`、`arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorState`
>
> ARNsKCLApplicationName」をそれぞれ独自のアプリケーション名 AWS リージョン、AWS アカウント 数値、および KCL アプリケーション名に置き換えます。KCL が作成するメタデータテーブルの名前を設定でカスタマイズしている場合は、KCL アプリケーション名ではなく、その指定したテーブル名を使用してください。

ステップ 7: KCL 3.x のコードをワーカーにデプロイする

移行に必要な設定を行い、これまでの移行チェックリストをすべて完了したら、コードをビルドしてワーカーへデプロイできます。

> **Note**
>
> `LeaseManagementConfig` コンストラクタでコンパイルエラーが発生した場合は、トラブルシューティング情報について LeaseManagementConfig コンストラクタでのコンパイルエラーを参照してください。

ステップ 8: 移行を完了する

KCL 3.x のコードをデプロイしている間、KCL は引き続き KCL 2.x のリース割り当てアルゴリズムを使用します。すべてのワーカーに KCL 3.x のコードを正常にデプロイすると、KCL がそれを自動的に検知し、ワーカーのリソース使用状況に基づく新しいリース割り当てアルゴリズムに切り替わります。新しいリース割り当てアルゴリズムの詳細については、KCL がワーカーにリースを割り当て、負荷を分散する方法 を参照してください。

デプロイ中は、CloudWatch に出力される次のメトリクスを使用して移行状況をモニタリングできます。これらのメトリクスは `Migration` オペレーションの下でモニタリングできます。すべてのメトリクスは KCL アプリケーション単位のメトリクスであり、メトリクスレベルは `SUMMARY` に設定されています。`CurrentState:3xWorker` メトリクスの `Sum` 統計値が、KCL アプリケーション内のワーカー合計数と一致した場合、KCL 3.x への移行が正常に完了したことを示します。

> **Important**
>
> すべてのワーカーが新しいリース割り当てアルゴリズムを実行できる状態になってから、KCL がその新しいアルゴリズムに切り替えるまでには、最低でも 10 分かかります。

KCL 移行プロセスの CloudWatch メトリクス

メトリクス	説明
`CurrentState:3xWorker`	KCL 3.x へ正常に移行し、新しいリース割り当てアルゴリズムを実行している KCL ワーカーの数です。このメトリクスの `Sum` 値がワーカーの総数と一致した場合、KCL 3.x への移行が正常に完了したことを示します。 • メトリクスレベル: Summary • 単位: カウント

メトリクス	説明
	• 統計: 最も有用な統計は Sum です。
CurrentState:2xCompatibleWorker	移行プロセス中に KCL 2.x 互換モードで実行されている KCL ワーカーの数です。このメトリクスの値がゼロ以外である場合、移行がまだ進行中であることを示します。 • メトリクスレベル: Summary • 単位: カウント • 統計: 最も有用な統計は Sum です。
Fault	移行プロセス中に発生した例外の数です。これらの例外の多くは一時的なエラーであり、KCL 3.x は自動的にリトライして移行を完了しようとします。Fault メトリクスの値が継続的に記録されている場合は、移行期間のログを確認して追加のトラブルシューティングを行ってください。問題が解決しない場合は、 にお問い合わせください サポート。 • メトリクスレベル: Summary • 単位: カウント • 統計: 最も有用な統計は Sum です。

メトリクス	説明
GsiStatusReady	リーステーブル上のグローバルセカンダリインデックス (GSI) の作成状況です。このメトリクスは、KCL 3.x を実行するための前提条件である、リーステーブル上の GSI が作成されているかどうかを示します。値は 0 または 1 で、1 の場合は GSI が正常に作成されたことを意味します。ロールバック状態の間、このメトリクスは出力されません。ロールフォワード後は、再びこのメトリクスをモニタリングできるようになります。 • メトリクスレベル: Summary • 単位: カウント • 統計: 最も有用な統計は Sum です。
workerMetricsReady	すべてのワーカーからのワーカーメトリクス送信状況です。このメトリクスは、すべてのワーカーが CPU 使用率などのメトリクスを送信しているかどうかを示します。値は 0 または 1 で、1 の場合はすべてのワーカーが正常にメトリクスを送信しており、新しいリース割り当てアルゴリズムの利用準備が整っていることを示します。ロールバック状態の間、このメトリクスは出力されません。ロールフォワード後は、再びこのメトリクスをモニタリングできるようになります。 • メトリクスレベル: Summary • 単位: カウント • 統計: 最も有用な統計は Sum です。

KCL は、移行中に 2.x 互換モードへロールバックできる機能を提供しています。KCL 3.x への移行が完了し、ロールバックが不要になった場合は、CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X

の CoordinatorConfig.clientVersionConfig 設定を削除することをお勧めします。この設定を削除すると、KCL アプリケーションから移行関連のメトリクスが出力されなくなります。

> **Note**
>
> 移行期間中および移行完了後の一定期間は、アプリケーションのパフォーマンスと安定性をモニタリングすることをお勧めします。問題が発生した場合は、KCL 移行ツールを使用して、ワーカーを KCL 2.x 互換の機能にロールバックすることができます。

以前のバージョンにロールバックする

このトピックでは、コンシューマーを前のバージョンにロールバックする手順について説明します。ロールバックが必要な場合は、次の 2 ステップのプロセスを実行します。

1. KCL 移行ツールを実行します。
2. 以前の KCL バージョンコードを再デプロイします (オプション)。

ステップ 1: KCL 移行ツールを実行する

以前の KCL バージョンにロールバックする必要がある場合は、KCL 移行ツールを実行する必要があります。KCL 移行ツールは 2 つの重要なタスクを実行します。

- これにより、DynamoDB のワーカーメトリクステーブルと呼ばれるメタデータテーブルとリーステーブルのグローバルセカンダリインデックスが削除されます。これら 2 つのアーティファクトは KCL 3.x によって作成されますが、以前のバージョンにロールバックするときには必要ありません。
- これにより、すべてのワーカーが KCL 2.x と互換性のあるモードで実行され、以前の KCL バージョンで使用される負荷分散アルゴリズムの使用が開始されます。KCL 3.x の新しい負荷分散アルゴリズムに問題がある場合、この問題はすぐに軽減されます。

> **Important**
>
> DynamoDB のコーディネーター状態テーブルが存在し、移行、ロールバック、ロールフォワードプロセス中に削除されていない必要があります。

> **Note**
>
> コンシューマーアプリケーションのすべてのワーカーが、一度に同じ負荷分散アルゴリズムを使用することが重要です。KCL 移行ツールを使用すると、KCL 3.x コンシューマーアプリケーション内のすべてのワーカーが KCL 2.x 互換モードに切り替えられ、前の KCL バージョンにロールバックしている間、すべてのワーカーが同じ負荷分散アルゴリズムで動作するようにします。

KCL 移行ツールは、KCL GitHub リポジトリのスクリプトディレクトリでダウンロードできます。このスクリプトは、コーディネーター状態テーブルへの書き込み、ワーカーメトリクステーブルの削除、およびリーステーブルの更新を行うための必要な権限を持つ任意のワーカー、または任意のホストから実行できます。スクリプトの実行に必要な IAM アクセス許可については、KCL コンシューマーアプリケーションに必要な IAM アクセス許可 を参照してください。このスクリプトは、KCL アプリケーションごとに 1 回だけ実行する必要があります。KCL 移行ツールは、以下のコマンドで実行できます。

```
python3 ./KclMigrationTool.py --region <region> --mode rollback [--
application_name <applicationName>] [--lease_table_name <leaseTableName>] [--
coordinator_state_table_name <coordinatorStateTableName>] [--worker_metrics_table_name
 <workerMetricsTableName>]
```

パラメータ

- --region: を <region> に置き換えます AWS リージョン。
- --application_name: このパラメータは、DynamoDB メタデータテーブル (リーステーブル、コーディネーター状態テーブル、ワーカーメトリクステーブル) にデフォルト名を使用している場合に必要です。これらのテーブルにカスタム名を指定している場合は、このパラメータを省略できます。<applicationName> を実際の KCL アプリケーションの名前に置き換えます。カスタム名が指定されていない場合、ツールはこの名前を使用してデフォルトのテーブル名を取得します。
- --lease_table_name (オプション): このパラメータは、KCL 設定でリーステーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している場合は、このパラメータを省略できます。leaseTableName をリーステーブルに指定したカスタムテーブル名に置き換えます。
- --coordinator_state_table_name (オプション): このパラメータは、KCL 設定でコーディネーター状態テーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している

場合は、このパラメータを省略できます。<coordinatorStateTableName> を、コーディネーター状態テーブルに指定したカスタムテーブル名に置き換えます。

- --worker_metrics_table_name (オプション): このパラメータは、KCL 設定でワーカーメトリクステーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している場合は、このパラメータを省略できます。<workerMetricsTableName> を、ワーカーメトリクステーブルに指定したカスタムテーブル名に置き換えます。

ステップ 2: 以前の KCL バージョンでコードを再デプロイする (オプション)

ロールバック用に KCL 移行ツールを実行すると、次のいずれかのメッセージが表示されます。

- メッセージ 1: 「ロールバックが完了しました。お使いの KCL アプリケーションは KCL 2.x 互換モードで動作していました。問題が改善されない場合は、以前の KCL バージョンでコードをデプロイして、元のアプリケーションバイナリにロールバックしてください」。
  - 必要なアクション: これは、ワーカーが KCL 2.x 互換モードで実行されていたことを意味します。問題が解決しない場合は、以前の KCL バージョンのコードをワーカーに再デプロイしてください。
- メッセージ 2: 「ロールバックが完了しました。お使いの KCL アプリケーションは KCL 3.x の機能モードで動作していました。問題について 5 分以内に問題が緩和されない場合を除き、以前のアプリケーションバイナリへのロールバックは不要です。問題が解決しない場合は、以前の KCL バージョンでコードをデプロイして、以前のアプリケーションバイナリにロールバックしてください」。
  - 必要なアクション: ワーカーが KCL 3.x モードで実行され、KCL 移行ツールがすべてのワーカーを KCL 2.x 互換モードに切り替えたことを意味します。問題が解決した場合は、以前の KCL バージョンのコードをワーカーに再デプロイする必要はありません。問題が解決しない場合は、以前の KCL バージョンのコードをワーカーに再デプロイしてください。

ロールバック後に KCL 3.x にロールフォワードする

このトピックでは、ロールバック後にコンシューマーを KCL 3.x へロールフォワードする手順について説明します。ロールフォワードが必要な場合は、2 ステップのプロセスを完了する必要があります。

1. KCL 移行ツールを実行します。
2. KCL 3.x を使用してコードをデプロイします。

ステップ 1: KCL 移行ツールを実行する

KCL 移行ツールを実行します。次のコマンドで KCL 移行ツールを実行し、KCL 3.x にロールフォワードします。

```
python3 ./KclMigrationTool.py --region <region> --mode rollforward [--application_name
 <applicationName>] [--coordinator_state_table_name <coordinatorStateTableName>]
```

パラメータ

- --region: を <region> に置き換えます AWS リージョン。
- --application_name: このパラメータは、コーディネーター状態テーブルにデフォルト名を使用している場合に必要です。コーディネーター状態テーブルにカスタム名を指定している場合は、このパラメータを省略できます。<applicationName> を実際の KCL アプリケーションの名前に置き換えます。カスタム名が指定されていない場合、ツールはこの名前を使用してデフォルトのテーブル名を取得します。
- --coordinator_state_table_name (オプション): このパラメータは、KCL 設定でコーディネーター状態テーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している場合は、このパラメータを省略できます。<coordinatorStateTableName> を、コーディネーター状態テーブルに指定したカスタムテーブル名に置き換えます。

移行ツールをロールフォワードモードで実行すると、KCL は KCL 3.x に必要な次の DynamoDB リソースを作成します。

- リーステーブルのグローバルセカンダリインデックス
- ワーカーメトリクステーブル

ステップ 2: KCL 3.x を使用してコードをデプロイする

ロールフォワードの KCL 移行ツールを実行したら、KCL 3.x を使用してコードをワーカーにデプロイします。ステップ 8: 移行を完了する に従って移行を完了します。

プロビジョンドキャパシティモードを使用するリーステーブルのベストプラクティス

KCL アプリケーションのリーステーブルがプロビジョンドキャパシティモードに切り替えられている場合、KCL 3.x はリーステーブル上にグローバルセカンダリインデックスを作成します。これは、プロビジョンド課金モードで作成され、ベースとなるリーステーブルと同じ読み込み容量ユニット (RCU) および書き込み容量ユニット (WCU) が設定されます。グローバルセカンダリインデックス

が作成された後は、DynamoDB コンソールでグローバルセカンダリインデックスの実際の使用状況をモニタリングし、必要に応じて容量ユニットを調整することをお勧めします。KCL によって作成される DynamoDB メタデータテーブルのキャパシティモードを切り替える方法について詳しくは、KCL が作成するメタデータテーブルの DynamoDB キャパシティモード を参照してください。

> **Note**
>
> デフォルトでは、KCL はリーステーブル、ワーカーメトリクステーブル、コーディネーター状態テーブルといったメタデータテーブル、さらにリーステーブル上のグローバルセカンダリインデックスをオンデマンドキャパシティモードで作成します。使用量の変化に基づいて容量を自動的に調整するには、オンデマンドキャパシティモードを使用することをお勧めします。

KCL 1.x から KCL 3.x への移行

このトピックでは、KCL 1.x のコンシューマーを KCL 3.x へ移行するための手順を説明します。KCL 1.x は、KCL 2.x や KCL 3.x とは異なるクラスとインターフェイスを使用します。まずレコードプロセッサ、レコードプロセッサファクトリ、ワーカークラスを KCL 2.x/3.x 互換形式に移行し、KCL 2.x から KCL 3.x への移行手順に従う必要があります。KCL 1.x から KCL 3.x に直接アップグレードできます。

- ステップ 1: レコードプロセッサを移行する

  KCL 1.x から KCL 2.x へのコンシューマーの移行ページのレコードプロセッサを移行するセクションに従います。
- ステップ 2: レコードプロセッサファクトリを移行する

  KCL 1.x から KCL 2.x へのコンシューマーの移行ページのレコードプロセッサファクトリーを移行するセクションに従います。
- ステップ 3: ワーカーを移行する

  KCL 1.x から KCL 2.x へのコンシューマーの移行ページのワーカーを移行するセクションに従います。
- ステップ 4: KCL 1.x 設定を移行する

  KCL 1.x から KCL 2.x へのコンシューマーの移行ページのAmazon Kinesis クライアントを設定するセクションに従います。

- ステップ 5: アイドル時間の削除とクライアント設定の削除を確認する

  KCL 1.x から KCL 2.x へのコンシューマーの移行ページのアイドル時間の削除およびクライアント設定の削除セクションに従います。

- ステップ 6: KCL 2.x から KCL 3.x への移行ガイドの手順に従う

  KCL 2.x から KCL 3.x に移行する ページの手順に従って移行を完了します。以前の KCL バージョンにロールバックするか、ロールバック後に KCL 3.x にロールフォワードする必要がある場合は、以前のバージョンにロールバックする および ロールバック後に KCL 3.x にロールフォワードする を参照してください。

> **Important**
>
> KCL 3.x で AWS SDK for Java バージョン 2.27.19 ~ 2.27.23 を使用しないでください。これらのバージョンには、KCL が使用する DynamoDB に関連した例外エラーを引き起こす問題が含まれています。この問題を回避するには、 AWS SDK for Java バージョン 2.28.0 以降を使用することをお勧めします。

以前の KCL バージョンのドキュメント

次のトピックはアーカイブされました。Kinesis Client Library の現行のドキュメントについては、Kinesis Client Library を使用する を参照してください。

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

廃止されたドキュメント

- KCL 1.x および 2.x の情報

- 共有スループットでカスタムコンシューマーを開発する
- コンシューマーを KCL 1.x から KCL 2.x に移行する

KCL 1.x および 2.x の情報

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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

- KCL について (以前のバージョン)
- KCL の以前のバージョン
- KCL の概念 (以前のバージョン)
- 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
- Schema Registry で KCL AWS Glue を使用する

> **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/releases を参照してください。

KCL について (以前のバージョン)

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 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。例えば、KCL for Python をインストールして、コンシューマーアプリケーションをすべて Python で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectを参照してください。

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 の以前のバージョン

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 コンシューマーを開発するを参照してください。

- KCL 2.x

  詳細については、KCL 2.x コンシューマーを開発する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 を使用した共有スループットでカスタムコンシューマーを開発する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、専用スループットを備えた拡張ファンアウトを開発する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 (以前のバージョン)

- 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のデータストリームの名前や、このデータストリームへのアクセスに必要な AWS 認証情報など、コンシューマーアプリケーションの設定情報を 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、(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 のスケジューラクラスの詳細については、https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/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、ワーカーは 1 つ以上のリースを同時に保持できます (maxLeasesForWorker 変数)。

  > **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、(Java KCL リポジトリにあります)、を参照してください。https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.javaの場合 KCL 1.x およびhttps://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/Lease.java KCL 2.x

- 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
- Kinesis 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`。

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。
- checkpointSubSequenceNumber: Kinesis Producer Library の集約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レコードを追跡するチェックポイントの拡張です。
- 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。
- ownerSwitchesSinceCheckpoint: 最後にチェックポイントが書き込まれてから、このリースのワーカーが何回変更されたかを示します。
- parentShardId: 子シャードの処理を開始する前に、親シャードが完全に処理済みであることを確認するために使用します。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、Kinesis 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。

Kinesis 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。

ShardFilter の詳細については、「https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html」を参照してください。

- 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、GetRecords、SubscribeToShards、および ChildShard を参照してください。
- 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 PeriodicShardSyncManager の詳細については、https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/LeaseManagementConfig.java#L201-L213を参照してください。

  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。

Name	デフォルトの 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

Name	デフォルトの 値	説明
		(ミリ単位)。監査者がストリームのリースの穴を検出すると、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。

新しい CloudWatch メトリクスも発行され、PeriodicShardSyncManager のヘルスをモニタリングします。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。

  ShardFilter の詳細については、「https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html」を参照してください。
- 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 および ChildShard を参照してください。

- 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での PeriodicShardSyncManager の詳細については、https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/KinesisClientLibConfiguration.java#L987-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。

Name	デフォルトの 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

Name	デフォルトの 値	説明
		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。

新しい CloudWatch メトリクスも発行され、PeriodicShardSyncManager のヘルスをモニタリングします。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。

Name	デフォルトの 値	説明
leaseCleanupIntervalMillis	1 分	リースクリーンアップスレッドを実行する間隔。
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

Name	デフォルトの 値	説明
		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 での次の変更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 インターフェイス

  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、MultistreamTracker という新しいインターフェイスを実装する必要があります。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。

  streamConfigList メソッドが StreamConfig リストに入力します。

```
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/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L129 を参照してください。StreamIdentifier を作成するには、streamArn および v2.5.0 以降で利用可能な streamCreationEpoch 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。streamArm をサポートしていない KCL v2.3 および v2.4 では、account-id:StreamName:streamCreationTimestamp 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。この形式は廃止され、次のメジャーリリース以降は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/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.javaを参照してください。

- ConfigsBuilder は、アプリケーション全体のクラスで、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 の構成設定をすべて指定するために使用できます。ConfigsBuilderクラスは MultistreamTracker インターフェイスをサポートするようになりました。ConfigsBuilder は、レコードを消費する 1 つのデータストリームの名前を使用して初期化できます。

```
/**
```

```
     * 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;
    }
```

または、複数のストリームを同時に処理する KCL コンシューマーアプリケーションを実装する場合、`MultiStreamTracker` で ConfigsBuilder を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`。

> **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`。

Schema Registry で KCL AWS Glue を使用する

Kinesis データストリームを AWS Glue Schema Registry 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 Schema Registry の統合は、Java に実装された 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 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。

共有スループットでカスタムコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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。Kinesis Client Library (KCL) または AWS SDK for Javaを使用できます。

トピック

- KCL を使用した共有スループットでカスタムコンシューマーを開発する

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、専用スループットを備えた拡張ファンアウトを開発するを参照してください。

KCL を使用した共有スループットでカスタムコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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。

使用している KCL バージョンの次のトピックから選択します。

トピック

- KCL 1.x コンシューマーを開発する
- KCL 2.x コンシューマーを開発する

KCL 1.x コンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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、KCL について (以前のバージョン)を参照してください。

使用するオプションに応じて、次のトピックから選択します。

内容

- Java での Kinesis クライアントライブラリコンシューマーを開発する
- ode.js で Kinesis Client Library コンシューマーを開発する
- .NET で Kinesis Client Library コンシューマーを開発する
- Python で Kinesis クライアントライブラリコンシューマーを開発する
- Ruby で Kinesis Client Library コンシューマーを開発する

Java での Kinesis クライアントライブラリコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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、AWS Javadoc topic for Class AmazonKinesisClientを参照してください。

GitHub から Java KCL をダウンロードするには、Kinesis Client Library (Java) にアクセスしてください。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ダウンロードするには、GitHub のKCL for Java sample projectページを参照してください。

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 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「Log4j AWS を使用したログ記録」を参照してください。 Log4j

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

- IRecordProcessor メソッドを実装する
- IRecordProcessor インターフェイスのクラスファクトリを実装する
- ワーカーを作成する
- 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 に移行する

IRecordProcessor 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。

IRecordProcessor 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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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。

IRecordProcessor インターフェイスのクラスファクトリを実装する

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();
    }
}
```

ワーカーを作成する

IRecordProcessor 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。

認証情報の設定

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いずれか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。

複数のインスタンスへのワーカー ID を使用する

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 に移行する

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。例えば、次のようになります。

```
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

ode.js で Kinesis Client Library コンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 ライブラリであり、Java 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。そのため、KCL for Node.js をインストールして、コンシューマーアプリケーションをすべて Node.js で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectのページを参照してください。

GitHub から Java KCL をダウンロードするには、Kinesis Client Library (Node.js) にアクセスしてください。

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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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。

認証情報の設定

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いずれか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 Kinesis Client Library コンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 ライブラリであり、Java 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。そのため、KCL for .NET をインストールして、コンシューマーアプリケーションをすべて .NET で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectのページを参照してください。

GitHub から .NET KCL をダウンロードするには、Kinesis Client Library (.NET) にアクセスしてください。.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、GitHub でKCL for .NET sample consumer project のページにアクセスしてください。

.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

- IRecordProcessor クラスのメソッドを実装する

- 設定プロパティを変更する

IRecordProcessor クラスの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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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。

認証情報の設定

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いずれか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 Kinesis クライアントライブラリコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 ライブラリであり、Java 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。そのため、KCL for Python をインストールして、コンシューマーアプリケーションをすべて Python で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectのページを参照してください。

GitHub から Python KCL をダウンロードするには、Kinesis Client Library (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、GitHub でKCL for Python sample projectページにアクセスしてください。

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 クラスのメソッドを実装する
- 設定プロパティを変更する

RecordProcessor クラスの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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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、KCL は、例外発生前に process_records 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。

認証情報の設定

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いずれか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 Kinesis Client Library コンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 ライブラリであり、Java 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。そのため、KCL for .Ruby をインストールして、コンシューマーアプリケーションをすべて Ruby で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectのページを参照してください。

GitHub から Ruby KCL をダウンロードするには、Kinesis Client Library (Ruby) にアクセスしてください。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、GitHub でKCL for Ruby sample projectページにアクセスしてください。

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 コンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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。

使用するオプションに応じて、次のトピックから選択します。

トピック

- Java での Kinesis クライアントライブラリコンシューマーを開発する
- Python で Kinesis クライアントライブラリコンシューマーを開発する
- 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

Java での Kinesis クライアントライブラリコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 Kinesis クライアントライブラリコンシューマーを開発する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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 ライブラリであり、Java 以外の言語のサポートは、MultiLangDaemon と呼ばれる多言語インターフェースを使用して提供されます。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。そのため、KCL for

Python をインストールして、コンシューマーアプリケーションをすべて Python で書く場合でも、MultiLangDaemon を使用するために、Java をシステムにインストールする必要があります。さらに、MultiLangDaemon には、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あります。GitHub の MultiLangDaemon の詳細については、KCL MultiLangDaemon projectのページを参照してください。

GitHub から Python KCL をダウンロードするには、Kinesis Client Library (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、GitHub でKCL for Python sample projectページにアクセスしてください。

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 クラスのメソッドを実装する
- 設定プロパティを変更する

RecordProcessor クラスのメソッドを実装する

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
```

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 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。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する場合は、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。

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

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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、専用スループットを備えた拡張ファンアウトを開発する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

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョ

ン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」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 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);
            }
        }
    }

}
```

コンシューマーを KCL 1.x から KCL 2.x に移行する

> **Important**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub のAmazon Kinesis Client Libraryページを参照してください。最新の KCL バージョンの詳細については、Kinesis Client Library を使用する を参照してください。KCL 1.x から KCL 3.x への移行については、「KCL 1.x から KCL 3.x への移行」を参照してください。

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を移行する
- レコードプロセッサファクトリーを移行する
- ワーカーを移行する
- Amazon Kinesis Client を設定する
- 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を移行する

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を移行する

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();
    }
}
```

ワーカーを移行する

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 を設定する

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。

元のフィールド	新しい設定クラス	説明
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)。
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。

元のフィールド	新しい設定クラス	説明
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。
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。

元のフィールド	新しい設定クラス	説明
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。
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。

元のフィールド	新しい設定クラス	説明
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。
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。

元のフィールド	新しい設定クラス	説明
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() 。

削除されたフィールド	同等の設定
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() 。

AWS SDK for Java を使用してコンシューマーを開発する

Amazon Kinesis Data Streams API を使用してカスタムコンシューマーを開発できます。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。

> **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、Java で KCL を使用してコンシューマーを開発するを参照してください。

トピック

- AWS SDK for Java を使用して共有スループットコンシューマーを開発する
- AWS SDK for Java 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 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

AWS SDK for Java を使用して共有スループットコンシューマーを開発する

共有スループットを利用するカスタム Kinesis Data Streams コンシューマーを開発する方法の 1 つは、AWS SDK for Java を使用して 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。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。

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 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。

トピック

- ストリームからのデータを取得する
- シャードイテレーターを使用する
- GetRecords を使用する
- リシャードに適応する

ストリームからのデータを取得する

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、データ保持期間を変更するを参照してください。

シャードイテレーターを使用する

レコードは、シャード単位でストリームから取得します。シャードごとに、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)。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。詳細については、シャードを一覧表示するを参照してください。

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、ストリームの一覧表示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、シャードを一覧表示する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 を使用する

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。

リシャードに適応する

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 SDK for Java 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、専用スループットを備えた拡張ファンアウトを開発する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。

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、ほぼすべてのタイプのアプリケーションまたはバックエンドサービスのコードを、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 で を使用する」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 を使用してコンシューマーを開発する

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、別の Kinesis ストリーム、Firehose 配信ストリーム、または Lambda 関数に分析結果を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 Data Firehose を使用してコンシューマーを開発する

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、Amazon OpenSearch Service、および Splunk などの送信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、カスタム HTTP エンドポイント、または Datadog、MongoDB、および New Relic などのサポート対象のサードパーティーサービスプロバイダーが所有する HTTP エンドポイント

もサポートします。データを変換し、レコード形式を変換してからデータを送信先に配信するよう 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 Kinesis Data Firehose への書き込み」を参照してください。

その他の AWS サービスを使用して Kinesis Data Streams からデータを読み取る

以下の AWS サービスは Amazon Kinesis Data Streams と直接統合し、Kinesis データストリームからデータを読み込むことができます。関心のある各サービスの情報を確認し、提供されたリファレンスを参照してください。

トピック

- Amazon EMR を使用して Kinesis Data Streams からデータを読み取る
- Amazon EventBridge Pipes を使用して Kinesis Data Streams からデータを読み取る
- AWS Glue を使用して Kinesis Data Streams からデータを読み取る
- Amazon Redshift を使用して Kinesis Data Streams からデータを読み取る

Amazon EMR を使用して Kinesis Data Streams からデータを読み取る

Amazon EMR クラスターは、Hive、Pig、MapReduce、Hadoop Streaming API、Cascading などの Hadoop エコシステムで使い慣れたツールを使用して、Kinesis Streams を直接読み込み、処理することができます。また、実行しているクラスターで Kinesis Data Streams 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 Pipes を使用して Kinesis Data Streams からデータを読み取る

Amazon EventBridge Pipes は、Kinesis Data Streams をソースとしてサポートします。Amazon EventBridge Pipes は、イベントプロデューサーとコンシューマー間でのポイントツーポイント統合の作成に役立ち、オプションとして変換、フィルタリング、および強化ステップを使用できます。EventBridge Pipes は、Kinesis Data Streams からレコードを受け取って、それらのレコードをオプションでフィルタリングまたは強化してから、処理のために利用できる送信先 (Kinesis Data Streams を含む) のいずれかに送信するために使用できます。

詳細については、「Amazon EventBridge リリースガイド」の「Amazon Kinesis ストリームをソースとする場合」を参照してください。

AWS Glue を使用して Kinesis Data Streams からデータを読み取る

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 を使用して Kinesis Data Streams からデータを読み取る

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。

サードパーティーの統合を使用して Kinesis Data Streams から読み取る

Amazon Kinesis Data Streams のデータストリームからのデータは、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読み取ることができます。詳細を確認し、関連ドキュメントのリソースとリンクを検索するオプションを選択します。

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消費に関する詳細については、「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に関する詳細については、「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 のデータの専

用読み取りスループットが得られます。詳しくは、「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。

Spark Streaming を使用した Kinesis データストリームを消費するには、「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 を使用した Kinesis データストリームの消費に関する詳細については、「Connect to Amazon Kinesis」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 Source Connector for Confluent Platform」を参照してください。

Kinesumer

Kinesumer は Go クライアントで、Kinesis データストリームのためのクライアント側の分散型コンシューマーグループクライアントを実装します。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 を使用した Kinesis データストリームの消費に関する詳細については、「Connect Talend to an Amazon Kinesis stream」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

以下のトピックでは、Amazon Kinesis Data Streams コンシューマーの一般的な問題に対するソリューションを提供します。

- LeaseManagementConfig コンストラクタでのコンパイルエラー
- 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
- ストリームにデータがある場合でも、GetRecords が空のレコード配列を返す
- シャードイテレーター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 キーの権限エラー
- DynamoDbException: 更新式で指定されたドキュメントパスが更新に対して無効です
- コンシューマーのその他の一般的な問題のトラブルシューティング

LeaseManagementConfig コンストラクタでのコンパイルエラー

Kinesis Client Library (KCL) バージョン 3.x 以降にアップグレードする際、`LeaseManagementConfig` コンストラクタに関連するコンパイルエラーが発生する場合があります。KCL バージョン 3.x 以降で `ConfigsBuilder` を使用せずに、設定を行うために直接 `LeaseManagementConfig` オブジェクトを作成している場合、KCL アプリケーションコードのコンパイル時に次のエラーメッセージが表示される可能性があります。

```
Cannot resolve constructor 'LeaseManagementConfig(String, DynamoDbAsyncClient,
 KinesisAsyncClient, String)'
```

KCL バージョン 3.x 以降では、tableName パラメータの後に、applicationName (型: String) という追加のパラメータを指定する必要があります。

- 変更前: leaseManagementConfig = new LeaseManagementConfig(tableName, dynamoDBClient, kinesisClient, streamName, workerIdentifier)
- 変更後: leaseManagementConfig = new LeaseManagementConfig(tableName, applicationName, dynamoDBClient, kinesisClient, streamName, workerIdentifier)

LeaseManagementConfig オブジェクトを直接作成するのではなく、KCL 3.x 以降のバージョンでは `ConfigsBuilder` を使用して設定を行うことをお勧めします。`ConfigsBuilder` を使用すると、KCL アプリケーションをより柔軟かつ保守しやすい方法で構成できます。

以下は、ConfigsBuilder を使用して KCL の設定を行う例です。

```
ConfigsBuilder configsBuilder = new ConfigsBuilder(
    streamName,
    applicationName,
    kinesisClient,
    dynamoClient,
    cloudWatchClient,
    UUID.randomUUID().toString(),
    new SampleRecordProcessorFactory()
);

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig()
    .failoverTimeMillis(60000), // this is an example
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
);
```

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、KCL 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します。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、重複レコードを処理する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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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。

ストリームにデータがある場合でも、GetRecords が空のレコード配列を返す

レコードの消費、つまり取得は、プルモデルです。開発者は、バックオフがない連続ループで GetRecords を呼び出す必要があります。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ー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 CloudWatch による Amazon Kinesis Data Streams サービスを監視するを参照してください。次に、Kinesis Client Library (KCL)、`MillisBehindLatest` が出力するカスタム 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の読み取りが各シャードからどのくらい遅れているかを確認します。詳細については、Amazon CloudWatch を使用した Kinesis Client Library を監視する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します。
- 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。processTask オペレーション (RecordProcessor.processRecords.Time、Success、RecordsProcessed など) に関連して KCL が出力する他のカスタム CloudWatch メトリクスを確認することで、この状況を確認できます。
  - 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 キーの権限エラー

このエラーは、AWS KMS 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。

DynamoDbException: 更新式で指定されたドキュメントパスが更新に対して無効です

AWS SDK for Java バージョン 2.27.19 から 2.27.23 で KCL 3.x を使用する場合、次の DynamoDB 例外が発生することがあります。

"software.amazon.awssdk.services.dynamodb.model.DynamoDbException: The document path provided in the update expression is invalid for update (Service: DynamoDb, Status Code: 400, Request ID: xxx)"

このエラーは、KCL 3.x が管理する DynamoDB メタデータテーブルに影響を及ぼす、AWS SDK for Java の既知の問題が原因で発生します。この問題は 2.27.19 で発生するようになり、2.27.23 までのすべてのバージョンで問題が発生します。この問題は AWS SDK for Java バージョン 2.27.24 で解決されました。最適なパフォーマンスと安定性のため、2.28.0 以降のバージョンへアップグレードすることをお勧めします。

コンシューマーのその他の一般的な問題のトラブルシューティング

- 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 ReadProvisionedThroughputExceeded 例外を検出してトラブルシューティング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 コンシューマーを最適化する

表示される特定の動作に基づいて、Amazon Kinesis Data Streams コンシューマーをさらに最適化できます。

以下のトピックを確認して、解決策を特定します。

トピック

- 低レイテンシー処理を改善する
- Amazon Kinesis プロデューサーライブラリ AWS Lambda で を使用してシリアル化されたデータを処理する
- シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。
- 重複レコードを処理する
- 起動、シャットダウン、スロットリングを処理する

低レイテンシー処理を改善する

伝達遅延は、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、エンドツーエンドのレイテンシー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。

Amazon Kinesis プロデューサーライブラリ AWS Lambda で を使用してシリアル化されたデータを処理する

Amazon 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 ではこれらのレコードの集約解除がサポートされていますが、 をストリームのコンシューマー AWS Lambda 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の集約を解除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、Amazon Kinesis Producer Library Deaggregation Modules for AWS Lambda の GitHub から入手できます。このプロジェクトのコンポーネントにより、Java、Node.js AWS Lambda、Python で KPL シリアル化されたデータを処理できます。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。

シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。

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)

Amazon EC2 Auto Scaling を使用して、適切なメトリクスに基づいてインスタンスを自動的にスケーリングできることに注意してください。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。

重複レコードを処理する

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**
>
> AWS SDK を使用する場合は`PutRecord`、「 SDK AWS とツールユーザーガイドSDKs の再試行動作について説明します。

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

これらすべての場合において、shards-to-worker-to-record-processor マッピングは、処理の負荷を分散する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、シャードの数を変更するには、再シャーディング、スケーリング、並列処理を使用します。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、スロットリングを処理する

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。

トピック

- データプロデューサーとデータコンシューマーを起動する
- Amazon Kinesis Data Streams アプリケーションをシャットダウンする
- 読み込みのスロットリング

データプロデューサーとデータコンシューマーを起動する

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。そのため、ストリームから読み込むコンシューマーアプリケーションが常に存在しており、CloudWatch メトリクス GetRecords.IteratorAgeMilliseconds を使用してアプリケーションが着信データに追随していることをモニタリング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。例えば、ストリームがエンドツーエンドで正常に機能していることをテストするために、ストリームで数件の初期レコードを実行する場合があります。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を使用するを参照してください。

Amazon Kinesis Data Streams アプリケーションをシャットダウンする

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 マネジメントコンソール 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。

Kinesis データストリームを監視する

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が付いた Amazon CloudWatch カスタムメトリクスを送信します。
- Kinesis エージェント - Kinesis エージェントは、エージェントが期待どおりに動作している場合、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 - Amazon 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 の IteratorAgeMilliseconds 値が増加し続けるのはなぜですか?

Amazon CloudWatch による Amazon Kinesis Data Streams サービスを監視する

Amazon Kinesis Data Streams は Amazon CloudWatch と統合されているため、Kinesis data streams の CloudWatch メトリクスを収集、表示、および分析できます。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。

ストリームメトリクスと設定する共有レベルのメトリクスは、自動的に収集され、1 分おきに CloudWatch 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。

タイプ	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を明示的に有効にする必要があります。 料金の詳細については、Amazon CloudWatch の料金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のシャードレベルの 2 つのレベルでメトリクスを CloudWatch に送信します。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で、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 CloudWatch ユーザーガイドのCloudWatch 統計を参照してください。

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、1 分ごとにこれらのストリームレベルのメトリクスを CloudWatch 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	このメトリクスは使用できなくなりました。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	ストリームの先頭からの読み取りレコードのミリ秒数は、コンシューマーが現在時刻からどれだけ遅れているかを示します。 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、1 分ごとに次のシャードレベルメトリクスを CloudWatch に送信します。メトリクスディメンションごとに 1 つのCloudWatch メトリクスが作成され、1 か月あたり約 43,200 回の

PutMetricData API コールが行われます。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 CloudWatch カスタムメトリクスの見出しの下のAmazon CloudWatch の料金を参照してください。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警告するように、Maximum 統計の CloudWatch アラームを使用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	レコードがストリームに後れていることを示す Average 統計のCloudWatch アラームを使用する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。Amazon Kinesis Producer Library (KPL) を使用する場合は、PutRecords.Success を使用します。
GetRecords.Success	レコードがストリームの役に立っていないことを示す Average 統計の CloudWatch アラームを使用することをお勧めします。

Kinesis Data Streams の Amazon CloudWatch メトリクスにアクセスする

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. CloudWatch コンソールの https://console.aws.amazon.com/cloudwatch/ を開いてください。
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 [Metrics (メトリクス)] を選択してください。
4. [カテゴリ別の CloudWatch メトリクス] ペインで [Kinesis メトリクス] を選択します。
5. 該当する行をクリックし、指定した [MetricName] と [StreamName] の統計を表示します。

   注: ほとんどのコンソール統計の名前は、読み込みスループットと書き込みスループットを除いて、上記の対応する CloudWatch メトリクス名と一致します。次の統計は 5 分間隔で計算されます。書き込みスループットは `IncomingBytes` CloudWatch メトリクスをモニタリングし、読み込みスループットは `GetRecords.Bytes` をモニタリングします。
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。

AWS CLI を使用してメトリクスにアクセスするには

list-metrics および 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 コマンドと 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 オペレーションと GetMetricStatistics オペレーションを使用します。

Amazon CloudWatch による Kinesis Data Streams エージェントのヘルスに監視する

Kinesis エージェントは、AWSKinesisAgent の名前空間でカスタム CloudWatch メトリクスを発行します。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、「エージェント設定を指定する」を参照してください。

CloudWatch を使用して監視する

Kinesis Data Streams エージェントは、次のメトリクスを CloudWatch に送信します。

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 コールのログを記録する

Amazon Kinesis Data Streams は AWS CloudTrail と統合されています。これは、ユーザー、ロール、または、Kinesis Data Streams の AWS サービスによって実行されたアクションの記録を提供するサービスです。CloudTrail は、Kinesis Data Streams に対するすべての API コールをイベントとしてキャプチャします。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 のイベントなど、Amazon S3 バケットへの CloudTrail イベントの継続的配信を有効にすることができます。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。CloudTrail によって収集された情報を使用して、Kinesis Data Streams に対して実行されたリクエスト、リクエスト実行元の IP アドレス、リクエストの実行者、リクエストの実行時、およびその他の詳細を判断することができます。

設定や有効化の方法など、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。

CloudTrail の Kinesis Data Streams の情報

AWS アカウントを作成すると、そのアカウントに対して CloudTrail が有効になります。サポートされているイベントアクティビティが Kinesis Data Streams で発生すると、そのアクティビティは [イベント履歴] の他の AWS サービスのイベントとともに、CloudTrail イベントに記録されます。最近のイベントは、AWS アカウントで表示、検索、ダウンロードできます。詳細については、CloudTrail イベント履歴でのイベントの表示を参照してください。

inesis Data Streams のイベントなど、AWS アカウントのイベントを継続的に記録する場合は、証跡を作成します。証跡により、ログファイルを CloudTrail で 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、すべての AWS リージョンに証跡が適用されます。証跡は、AWS パーティション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 ログで収集したイベントデータをより詳細に分析し、それに基づいて対応するため、他の AWS サービスを構成できます。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 がサポートするサービスと統合
- CloudTrail 用 Amazon SNS 通知の構成

- 複数のリージョンから CloudTrail ログファイルを受け取るおよび複数のアカウントから CloudTrail ログファイルを受け取る

Kinesis Data Streams は、CloudTrail ログファイルのイベントとして以下のアクションのログ記録をサポートします。

- 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、リクエストの生成者に関する情報が含まれます。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、ルート認証情報と AWS Identity and Access Management (IAM) ユーザー認証情報のどちらを使用して送信されたか。
- 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、CloudTrail userIdentity 要素を参照してください。

例: Kinesis Data Streams ログファイルエントリ

「トレイル」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するように設定できます。CloudTrail のログファイルは、単一か複数のログエントリを含みます。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 呼び出し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。

以下は `CreateStream`、`DescribeStream`、`ListStreams`、`DeleteStream`、`SplitShard` および `MergeShards` のアクションを示す CloudTrail ログエントリの例です。

```
{
    "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 CloudWatch を使用した Kinesis Client Library を監視する

Amazon Kinesis 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、お客様に代わってカスタム Amazon CloudWatch メトリクスを発行します。CloudWatch console に移動し、[カスタムメトリクス] を選択すると、これらのメトリクスを表示できます。カスタムメトリクスに関する詳細については、Amazon CloudWatch ユーザーガイドのカスタムメトリクスを発行するを参照してください。

KCL によって CloudWatch にアップロードされたメトリクスには、小額の課金が発生します。具体的には、Amazon CloudWatch カスタムメトリクスと Amazon CloudWatch API リクエストの料金が適用されます。詳細については、「Amazon CloudWatch 料金表」をご覧ください。

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

CloudWatch にアップロードされるメトリクスを制御する 2 つのオプションがあります。

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、レポートレベル以下の個別のレベルのメトリクスは CloudWatch に送信されません。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 CloudWatch に送信されます。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、CloudWatch に送信されるディメンションが関連付けられています。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 ユーザーガイドのAmazon CloudWatch の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で、特定のKCL ワーカーが再起動するたびにワーカー ID プロパティに異なる値が使用される場合、新しい WorkerIdentifier ディメンション値を持つ新しいメトリクスのセットが CloudWatch に送信されます。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、KinesisClientLibConfiguration インスタンスを使用して設定でき、このインスタンスは KCL アプリケーションを起動するときにワーカーに渡されます。MultiLangDaemon の場合、metricsLevel プロパティおよび metricsEnabledDimensions プロパティは、MultiLangDaemon KCL アプリケーションを起動するために使用される .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の値は、CloudWatch メトリクスで許可されているディ

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
- ワーカーあたりのメトリクス
- シャードあたりのメトリクス

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範囲内にあるすべての KCL ワーカーにわたって集約されます。

トピック

- LeaseAssignmentManager
- InitializeTask
- ShutdownTask
- ShardSyncTask
- BlockOnParentTask
- PeriodicShardSyncManager
- MultistreamTracker

LeaseAssignmentManager

`LeaseAssignmentManager` オペレーションは、ワーカーへのリースの割り当ておよびリースの再分配を行い、ワーカーリソースの均等な利用を実現する役割を担います。このオペレーションのロジックには、リーステーブルからリース関連のメタデータを読み取り、ワーカーメトリクステーブルからメトリクスを取得し、リースの割り当てを実行する処理が含まれます。

メトリクス	説明
LeaseAndWorkerMetricsLoad.Time	リース割り当てマネージャー (LAM) で、すべてのリースおよびワーカーメトリクスエントリを読み込むのに要した時間を示します。LAM は、KCL 3.x で導入された新しいリース割り当ておよび負荷分散アルゴリズムです。 メトリクスレベル: Detailed 単位: ミリ秒
TotalLeases	現在の KCL アプリケーションのリースの合計数。 メトリクスレベル: Summary 単位: カウント
NumWorkers	KCL アプリケーションのワーカーの合計数。 メトリクスレベル: Summary 単位: カウント
AssignExpiredOrUnassignedLeases.Time	期限切れのリースをメモリ上で割り当てる処理に要した時間。 メトリクスレベル: Detailed 単位: ミリ秒
LeaseSpillover	ワーカーごとのリース数またはスループットの上限に達したために割り当てられなかったリースの数。 メトリクスレベル: Summary 単位: カウント
BalanceWorkerVariance.Time	ワーカー間でリースのメモリ上のバランシングを実行する処理に要した時間。 メトリクスレベル: Detailed

メトリクス	説明
	単位: ミリ秒
NumOfLeasesReassignment	現在の再割り当てイテレーションで実行されたリースの再割り当ての合計数。 メトリクスレベル: Summary 単位: カウント
FailedAssignmentCount	DynamoDB のリーステーブルに対する AssignLease 呼び出しで発生した失敗の数。 メトリクスレベル: Detailed 単位: カウント
ParallelyAssignLeases.Time	DynamoDB のリーステーブルに新しい割り当て内容を書き込む処理に要した時間。 メトリクスレベル: Detailed 単位: ミリ秒
ParallelyAssignLeases.Success	新しい割り当て内容の書き込みが正常に完了した回数。 メトリクスレベル: Detailed 単位: カウント
TotalStaleWorkerMetricsEntry	クリーンアップが必要なワーカーメトリクスエントリの合計数。 メトリクスレベル: Detailed 単位: カウント

メトリクス	説明
StaleWorkerMetrics Cleanup.Time	DynamoDB のワーカーメトリクステーブルからワーカーメトリクスエントリを削除する処理に要した時間。 メトリクスレベル: Detailed 単位: ミリ秒
Time	`LeaseAssignmentManager` 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
Success	`LeaseAssignmentManager` 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
ForceLeaderRelease	リース割り当てマネージャーが 3 回連続で失敗し、リーダーワーカーがリーダーシップを解放していることを示します。 メトリクスレベル: Summary 単位: カウント
NumWorkersWithInvalidEntry	無効と見なされたワーカーメトリクスエントリの数。 メトリクスレベル: Summary 単位: カウント
NumWorkersWithFailingWorkerMetric	ワーカーメトリクスの値の 1 つとして -1 (ワーカーメトリクス値が利用できないことを示す) が含まれているワーカーメトリクスエントリの数。 メトリクスレベル: Summary 単位: カウント

メトリクス	説明
LeaseDeserializationFailureCount	逆シリアル化に失敗したリーステーブル内のリースエントリ。 メトリクスレベル: Summary 単位: カウント

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
Success	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

メトリクス	説明
	単位: カウント
Time	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.Success	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.Success	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

メトリクス	説明
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
Success	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
Time	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.Success	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

メトリクス	説明
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
Success	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
Time	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
Success	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
Time	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 (次に NumStreamsToSync および NumStreamsWithPartialLeases の値が 1 に設定されている場合)、および複数のデータストリームを処理するように KCL が構成されている場合に使用できます。

メトリクス	説明
NumStreamsToSync	部分的なリースを含み、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ているデータストリームの数 (AWS アカウントごと)。 メトリクスレベル: Summary 単位: カウント
NumStreamsWithPartialLeases	部分的なリースを含み、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の数 (AWS アカウントごと)。 メトリクスレベル: Summary 単位: カウント
Success	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
Time	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査し、シャードの同期が必要なデータストリームを特定するために、`PeriodicShardSyncManager` が要する時間 (ミリ秒単位)。 メトリクスレベル: Summary

メトリクス	説明
	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

- WorkerMetricStatsReporter
- LeaseDiscovery
- RenewAllLeases

- TakeLeases

WorkerMetricStatsReporter

WorkerMetricStatReporter オペレーションは、現在のワーカーのメトリクスを定期的にワーカーメトリクステーブルに公開する役割を担います。これらのメトリクスは、LeaseAssignmentManager オペレーションによるリース割り当ての実行に使用されます。

メトリクス	説明
InMemoryMetricStatsReporterFailure	一部のワーカーメトリクスの失敗により、メモリ内ワーカーメトリクス値の取得に失敗した回数。 メトリクスレベル: Summary 単位: カウント
WorkerMetricStatsReporter.Time	WorkerMetricsStats 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
WorkerMetricStatsReporter.Success	WorkerMetricsStats 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

LeaseDiscovery

LeaseDiscovery オペレーションは、LeaseAssignmentManager オペレーションによって現在のワーカーに割り当てられた新しいリースを特定する役割を担います。このオペレーションのロジックには、リーステーブルのグローバルセカンダリインデックスを読み取ることで、現在のワーカーに割り当てられたリースを特定する処理が含まれます。

メトリクス	説明
ListLeaseKeysForWorker.Time	リーステーブルのグローバルセカンダリインデックスを呼び出し、現在のワーカーに割り当てられたリースキーを取得するまでの時間。 メトリクスレベル: Detailed 単位: ミリ秒
FetchNewLeases.Time	リーステーブルからすべての新しいリースを取得するまでに要する時間。 メトリクスレベル: Detailed 単位: ミリ秒
NewLeasesDiscovered	ワーカーに割り当てられた新しいリースの合計数。 メトリクスレベル: Detailed 単位: カウント
Time	`LeaseDiscovery` 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
Success	`LeaseDiscovery` 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
OwnerMismatch	GSI の応答とリーステーブルの整合読み取り結果との間で発生した所有者の不一致の数。 メトリクスレベル: Detailed 単位: カウント

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.Success	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 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
Success	ワーカーに対してリース更新オペレーションが正常に実行された回数。 メトリクスレベル: Summary 単位: カウント
Time	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

メトリクス	説明
	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.Success	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.Success	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	説明
	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
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
Success	`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

メトリクス	説明
Time	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を使用して GetRecords を呼び出すことにより、ストリームからレコードを取得して、レコードプロセッサの processRecords 関数を起動します。

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.Success	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

メトリクス	説明
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
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	`GetRecords` を呼び出したときに受信した ExpiredIteratorException の数。 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

メトリクス	説明
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
Success	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
Time	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 CloudWatch を使用した Kinesis Client Library を監視する

Amazon Kinesis Data Streams の Amazon Kinesis Producer Library (KPL) は、お客様に代わってカスタム Amazon CloudWatch メトリクスを発行します。CloudWatch console に移動し、[カスタムメトリクス] を選択すると、これらのメトリクスを表示できます。カスタムメトリクスに関する詳細については、Amazon CloudWatch ユーザーガイドのカスタムメトリクスを発行するを参照してください。

KPL によって CloudWatch にアップロードされたメトリクスには、小額の課金が発生します。具体的には、Amazon CloudWatch カスタムメトリクスと Amazon CloudWatch API リクエストの料金が適用されます。詳細については、「Amazon CloudWatch 料金表」をご覧ください。ローカルメトリクスの収集では、CloudWatch の課金は発生しません。

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 のアップロード
- 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、より詳細なデータが CloudWatch メトリクスに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

CloudWatch にアップロードされるメトリクスの数を制御する 2 つのオプションがあります。

メトリクスレベル

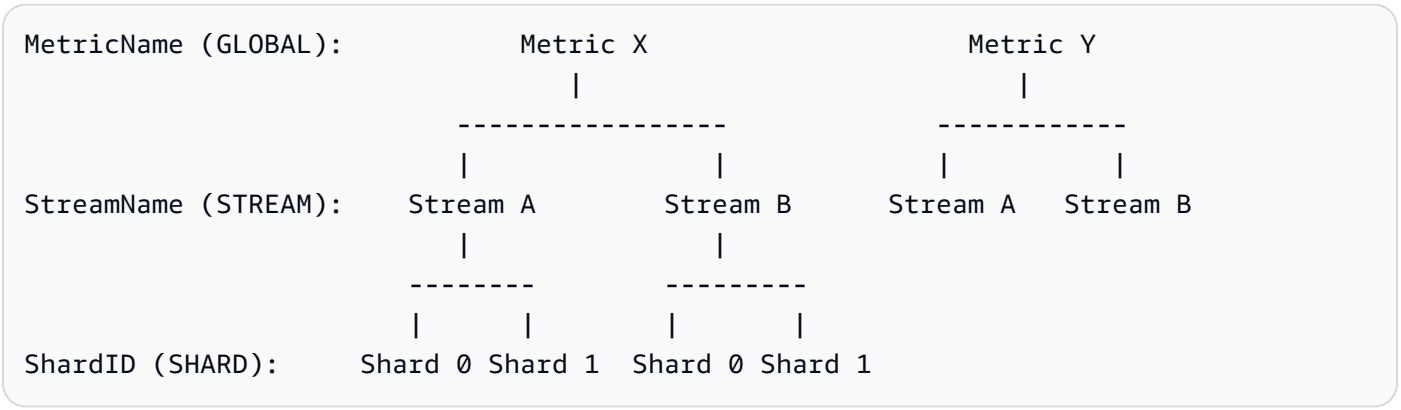
これは、メトリクスの重要性を示すおおよその目安です。すべてのメトリクスにレベルが割り当てられます。レベルを設定すると、それより下のレベルのメトリクスは、CloudWatch に送信されません。このレベルとして、NONE、SUMMARY、DETAILED があります。デフォルト設定は DETAILED であり、すべてのメトリクスが対象です。NONEは、メトリクスが一切ないことを意味し、どのメトリクスもそのレベルに割り当てられません。

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、ある特定のメトリクスについて、それぞれに 2 つのシャードがある 2 つのストリームから、7 つのCloudWatch メトリクスが生成されます。各シャードに 1 つ、各ストリームに 1 つ、全体に 1 つのメトリクスが生成され、これらはどれも同じ統計情報を記述していますが、詳細度のレベルは異なります。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。



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 のアップロード

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 では、CloudWatch の場合と同様に、すべてのメトリクスの合計、平均、最小値、最大値、および個数をローカルで計算します。

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は、CloudWatch へのアップロードに使用することができます。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。そのため、KPL は、メトリクスをローカルで 1 分間のバケットに

集約し、有効なメトリクスごとに 1 分あたり 1 回ずつ統計情報オブジェクトを CloudWatch 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: 数
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: 数
UserRecordsPut	入力に成功した論理ユーザーレコードの数。 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: 数
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

メトリクス	説明
	単位: バイト
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: 数
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: 数

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: 数
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。レコードの有効期限値が大きいと、このメトリクスの出力が大幅に遅延する場合があります。 メトリクスレベル: Detailed 単位: 数
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: 数
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: 数
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: 数

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。お客様は AWS 、最もセキュリティの影響を受けやすい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を活用できます。

セキュリティは、 AWS とお客様の間の責任共有です。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 – AWS 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 AWS 責任があります。 AWS また、 では、安全に使用できるサービスも提供しています。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 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
- IAM を使用した Amazon Kinesis Data Streams リソースへのアクセスの制御
- Amazon Kinesis Data Streams のコンプライアンス検証
- Amazon Kinesis Data Streams の耐障害性
- Kinesis Data Streams のインフラストラクチャセキュリティ
- Kinesis Data Streams のセキュリティのベストプラクティス

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?
- 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の使用開始方法
- ユーザー生成の KMS キーを作成して使用する
- ユーザー生成の KMS キーを使用するための許可
- KMS キー許可の検証とトラブルシューティングを行う
- インターフェイス VPC エンドポイントと Amazon Kinesis Data Streams を使用する

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 の機能であり、指定した AWS KMS カスタマーマスターキー (CMK) を使用して、保管前に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。

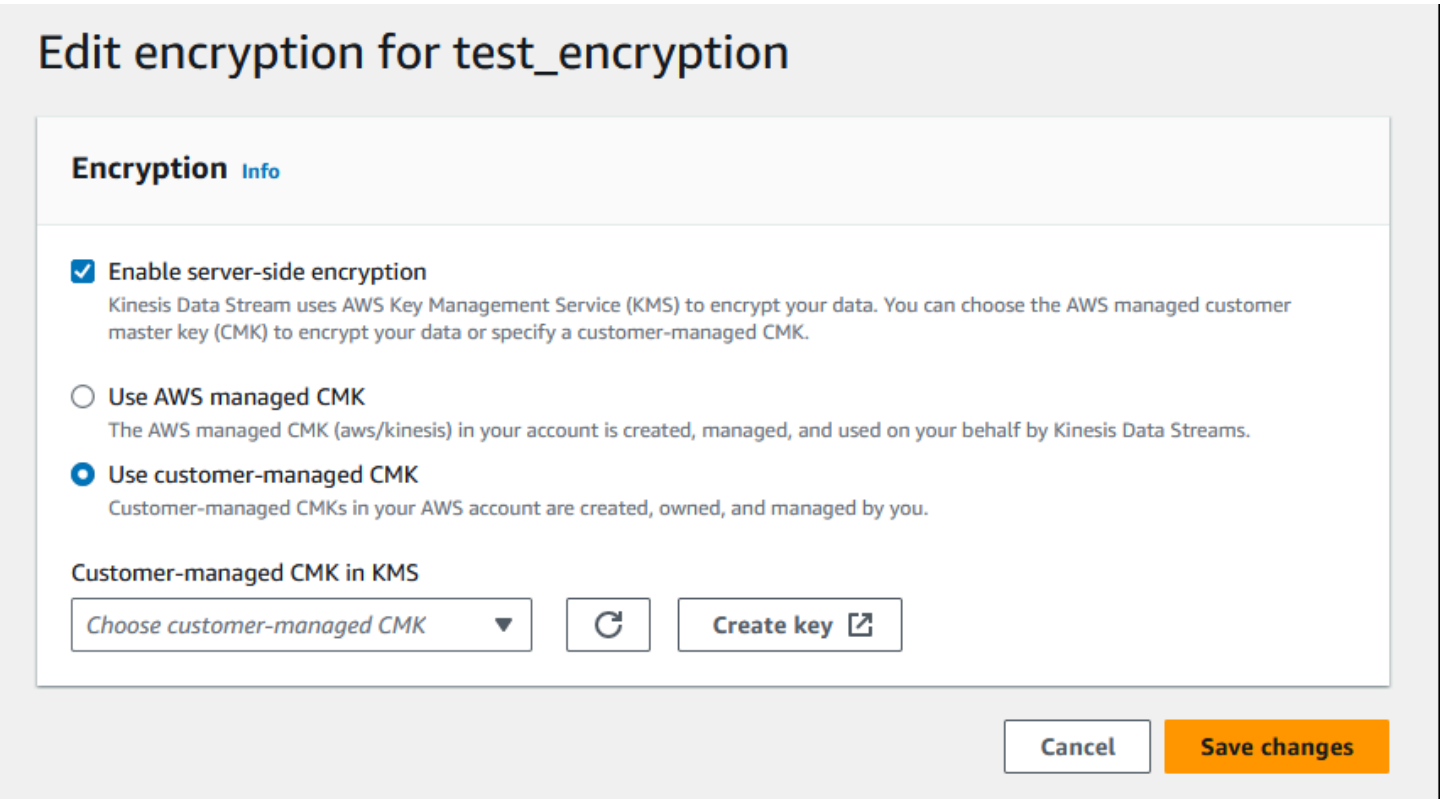
サーバー側の暗号化を使用すると、Kinesis ストリームプロデューサーとコンシューマーがマスターキーや暗号化オペレーションを管理する必要はありません。データは Kinesis Data Streams サービスに出入りするときに自動的に暗号化されるため、保管中のデータは暗号化されます。 AWS KMS は、サーバー側の暗号化機能で使用されるすべてのマスターキーを提供します。 AWS KMS を使用

すると 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、他のプリンシパルへのアクセスを共有するときは、 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は、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スケールされます。各ユーザー認証情報には一意の 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 KMS への一意の API AWS コールが必要であるため、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スケーリングされることに注意してください。IAM ロールを認証に使用すると、各 assume-role-call によって一意のユーザー認証情報が生成され、assume-role-call によって返されたユーザー認証情報をキャッシュして KMS のコストを節約できます。

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 マネジメントコンソール 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 マネジメントコンソール し、Amazon Kinesis Data Streams コンソールを開きます。
2. AWS マネジメントコンソールで 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 マネジメントコンソール、保存を選択します。

ユーザー生成の KMS キーを作成して使用する

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 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 キーを作成する

ユーザー独自の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 キーを使用する

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 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 マネジメントコンソールにある [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 マネジメントコンソールの 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 キーを使用するための許可

ユーザーが生成した KMS 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、「KMS での IAM AWS 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。

JSON

```
{
  "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。

JSON

```
{
  "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 と 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を行う

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 の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、Kinesis データストリームを監視する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 を使用する

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ることを防止できます。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 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、Amazon VPC 内のプライベート IPs を持つ Elastic Network Interface を使用して AWS 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 PrivateLink を利用 AWS しています。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 エンドポイントを使用する
- Kinesis Data Streams の VPCE エンドポイントへのアクセスを制御する
- 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 エンドポイントを使用する

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 用にインターフェイス VPC エンドポイントを作成し、インターフェイス VPC エンドポイントを通じて Amazon VPC リソースから、または Amazon VPC リソースへ流れるトラフィックを開始します。FIPS 対応のインターフェイス VPC エンドポイントは、米国リージョンで使用できます。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。

Amazon Kinesis プロデューサーライブラリ (KPL) と Kinesis コンシューマーライブラリ (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を制御する

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 エンドポイントを介した Kinesis データストリームのアクションへのアクセスのみを許可する IAM ポリシーと組み合わせて使用するときに、指定された VPC エンドポイントへの特定のストリームへのアクセスを制限するために使用します。

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。

  JSON

```
{
   "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、以下を実行できます。

- 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、それらを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 Amazon DynamoDB と Amazon CloudWatch への許可を含める必要があります。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、CloudWatch を使用してユーザーに代わって KCL メトリクスを CloudWatch に送信します。KCL の詳細については、KCL 1.x コンシューマーを開発する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 の開始方法
- 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 CloudWatch の詳細については、Amazon CloudWatch 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」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を共有する
- 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を共有する

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。

JSON

```
{
    "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。

JSON

```
{
    "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。

JSON

```
{
```

```
    "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。

JSON

```
{
    "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を共有する

> **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を有効にする

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のリソースに対してどのような条件下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 Data Streams 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

JSON

```
{
    "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

JSON

```
{
    "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:

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
```

```
        {
            "Sid": "consumerreadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333: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:

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_efo_read_policy_ID",
    "Statement": [
        {
            "Sid": "eforeadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333: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を管理する

の外部では AWS マネジメントコンソール、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 API 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介し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、次のポリシーを注意深く調べて、広範なアクセスが許可されていないことを確かめてください。
  - 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、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を共有する

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 関数とアクセスを共有する」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を共有する

> **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 関数とアクセスを共有する

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 呼び出し権限を持つマネージド IAM ポリシー `AWSLambdaKinesisExecutionRole` 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 Data Streams 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を共有する

- 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 コンストラクターを使用してストリーム ARN を指定します。
   - KCL 2.x の場合: Kinesis Client Library ConfigsBuilder にストリーム ARN または StreamTracker のみを指定できます。StreamTracker には、ライブラリによって生成される 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

アウトなどの登録済みの共有コンシューマーから読み込む場合は、StreamTracker を使用してコンシューマー ARN も指定してください。

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 Data Streams 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を共有する

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、AWS「コンプライアンス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 などの標準への準拠の対象である場合、 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デプロイする手順について説明します AWS。
- 「Architecting for HIPAA Security and Compliance」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このコレクション
- AWS Config – リソース設定が内部プラクティス、業界ガイドライン、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 CSPM – 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、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機能がいくつか用意されています。

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 の 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、ワーカーまたはアプリケーションが失敗したものの、ホスティング EC2 インスタンスがまだ実行されているため、Amazon EC2 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

Amazon EC2 Auto Scaling グループでアプリケーションの Amazon EC2 インスタンスを実行することをお勧めします。これにより、いずれかの EC2 インスタンスに障害が発生した場合、Amazon EC2 Auto Scaling グループは新しいインスタンスを自動的に起動して置き換えます。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手順で保護されています。

AWS 公開された 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、これらは指示ではなく、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 認証情報を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を実装する

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 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ある と統合されています。

CloudTrail によって収集された情報を使用して、Kinesis Data Streams に対して実行されたリクエスト、リクエスト実行元の IP アドレス、リクエストの実行者、リクエストの実行時、およびその他の詳細を判断することができます。

詳細については、「the section called “AWS CloudTrail を使用した Amazon Kinesis Data Streams API コールのログを記録する”」を参照してください。

AWS SDK でのこのサービスの使用

AWS Software Development Kit (SDKsは、多くの一般的なプログラミング言語で使用できます。各 SDK には、デベロッパーが好みの言語でアプリケーションを簡単に構築できるようになる API、コード例、およびドキュメントが提供されています。

SDK ドキュメント	コード例
AWS SDK for C++	AWS SDK for C++ コード例
AWS CLI	AWS CLI コード例
AWS SDK for Go	AWS SDK for Go コード例
AWS SDK for Java	AWS SDK for Java コード例
AWS SDK for JavaScript	AWS SDK for JavaScript コード例
AWS SDK for Kotlin	AWS SDK for Kotlin コード例
AWS SDK for .NET	AWS SDK for .NET コード例
AWS SDK for PHP	AWS SDK for PHP コード例
AWS Tools for PowerShell	AWS Tools for PowerShell コード例
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) コード例
AWS SDK for Ruby	AWS SDK for Ruby コード例
AWS SDK for Rust	AWS SDK for Rust コード例
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP コード例
AWS SDK for Swift	AWS SDK for Swift コード例

可用性の例

必要なものが見つからなかった場合。このページの下側にある [Provide feedback] リンクから、コードの例をリクエストしてください。

AWS SDKsコード例

次のコード例は、 AWS Software Development Kit (SDK) で Kinesis を使用する方法を示しています。

基本 は、重要なオペレーションをサービス内で実行する方法を示すコード例です。

アクションはより大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。アクションは個々のサービス機能を呼び出す方法を示していますが、コンテキスト内のアクションは、関連するシナリオで確認できます。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

コードの例

- AWS SDKs基本的な例
  - AWS SDK を使用した Kinesis の基本について説明します。
  - AWS SDKsアクション
    - AWS SDK または CLI AddTagsToStreamで を使用する
    - AWS SDK または CLI CreateStreamで を使用する
    - AWS SDK または CLI DeleteStreamで を使用する
    - AWS SDK または CLI DeregisterStreamConsumerで を使用する
    - AWS SDK または CLI DescribeStreamで を使用する
    - AWS SDK または CLI GetRecordsで を使用する
    - CLI で GetShardIterator を使用する
    - AWS SDK ListStreamConsumersで を使用する
    - AWS SDK または CLI ListStreamsで を使用する
    - AWS SDK または CLI ListTagsForStreamで を使用する
    - AWS SDK または CLI PutRecordで を使用する
    - AWS SDK または CLI PutRecordsで を使用する
    - AWS SDK または CLI RegisterStreamConsumerで を使用する
- Kinesis のサーバーレス例
  - Kinesis トリガーから Lambda 関数を呼び出す

- Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

AWS SDKs基本的な例

次のコード例は、 AWS SDK を使用した Amazon Kinesis の基本的な使用方法を説明しています。

例

- AWS SDK を使用した Kinesis の基本について説明します。
- AWS SDKsアクション
  - AWS SDK または CLI AddTagsToStreamで を使用する
  - AWS SDK または CLI CreateStreamで を使用する
  - AWS SDK または CLI DeleteStreamで を使用する
  - AWS SDK または CLI DeregisterStreamConsumerで を使用する
  - AWS SDK または CLI DescribeStreamで を使用する
  - AWS SDK または CLI GetRecordsで を使用する
  - CLI で GetShardIterator を使用する
  - AWS SDK ListStreamConsumersで を使用する
  - AWS SDK または CLI ListStreamsで を使用する
  - AWS SDK または CLI ListTagsForStreamで を使用する
  - AWS SDK または CLI PutRecordで を使用する
  - AWS SDK または CLI PutRecordsで を使用する
  - AWS SDK または CLI RegisterStreamConsumerで を使用する

AWS SDK を使用した Kinesis の基本について説明します。

次のコード例は、以下の操作方法を示しています。

- ストリームを作成し、そのストリームにレコードを挿入します。
- シャードイテレーターを作成します。
- レコードを読み取り、リソースをクリーンアップします。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   DATA lo_stream_describe_result TYPE REF TO /aws1/cl_knsdescrstreamoutput.
   DATA lo_stream_description TYPE REF TO /aws1/cl_knsstreamdescription.
   DATA lo_sharditerator TYPE REF TO /aws1/cl_knsgetsharditerator01.
   DATA lo_record_result TYPE REF TO /aws1/cl_knsputrecordoutput.

   "Create stream."
   TRY.
       lo_kns->createstream(
           iv_streamname = iv_stream_name
           iv_shardcount = iv_shard_count ).
       MESSAGE 'Stream created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceeded
exception.' TYPE 'E'.
     CATCH /aws1/cx_knsresourceinuseex.
       MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
   ENDTRY.

   "Wait for stream to becomes active."
   lo_stream_describe_result = lo_kns->describestream( iv_streamname =
iv_stream_name ).
   lo_stream_description = lo_stream_describe_result->get_streamdescription( ).
   WHILE lo_stream_description->get_streamstatus( ) <> 'ACTIVE'.
     IF sy-index = 30.
       EXIT.               "maximum 5 minutes"
     ENDIF.
     WAIT UP TO 10 SECONDS.
```

```
      lo_stream_describe_result = lo_kns->describestream( iv_streamname =
 iv_stream_name ).
      lo_stream_description = lo_stream_describe_result-
>get_streamdescription( ).
    ENDWHILE.

    "Create record."
    TRY.
        lo_record_result = lo_kns->putrecord(
            iv_streamname = iv_stream_name
            iv_data       = iv_data
            iv_partitionkey = iv_partition_key ).
        MESSAGE 'Record created.' TYPE 'I'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knskmsaccessdeniedex.
        MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
 'E'.
      CATCH /aws1/cx_knskmsdisabledex.
        MESSAGE 'KMS key used is disabled.' TYPE 'E'.
      CATCH /aws1/cx_knskmsinvalidstateex.
        MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
 quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
 data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Create a shard iterator in order to read the record."
    TRY.
        lo_sharditerator = lo_kns->getsharditerator(
          iv_shardid = lo_record_result->get_shardid( )
          iv_sharditeratortype = iv_sharditeratortype
          iv_streamname = iv_stream_name ).
        MESSAGE 'Shard iterator created.' TYPE 'I'.
      CATCH /aws1/cx_knsinvalidargumentex.
```

```
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_sgmresourcenotfound.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Read the record."
    TRY.
        oo_result = lo_kns->getrecords(                    " oo_result is
returned for testing purposes. "
            iv_sharditerator   = lo_sharditerator->get_sharditerator( ) ).
        MESSAGE 'Shard iterator created.' TYPE 'I'.
      CATCH /aws1/cx_knsexpirediteratorex.
        MESSAGE 'Iterator expired.' TYPE 'E'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knskmsaccessdeniedex.
        MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
      CATCH /aws1/cx_knskmsdisabledex.
        MESSAGE 'KMS key used is disabled.' TYPE 'E'.
      CATCH /aws1/cx_knskmsinvalidstateex.
        MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Delete stream."
    TRY.
        lo_kns->deletestream(
            iv_streamname = iv_stream_name ).
        MESSAGE 'Stream deleted.' TYPE 'I'.
```

```
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceeded
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourceinuseex.
        MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の以下のトピックを参照してください。
  - CreateStream
  - DeleteStream
  - GetRecords
  - GetShardIterator
  - PutRecord

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDKsアクション

次のコード例は、 AWS SDKs を使用して個々の Kinesis アクションを実行する方法を示しています。それぞれの例には、GitHub へのリンクがあり、そこにはコードの設定と実行に関する説明が記載されています。

以下の例には、最も一般的に使用されるアクションのみ含まれています。詳細な一覧については、「Amazon Kinesis API Reference」(Amazon SES API リファレンス) を参照してください。

例

- AWS SDK または CLI AddTagsToStreamで を使用する
- AWS SDK または CLI CreateStreamで を使用する
- AWS SDK または CLI DeleteStreamで を使用する
- AWS SDK または CLI DeregisterStreamConsumerで を使用する
- AWS SDK または CLI DescribeStreamで を使用する
- AWS SDK または CLI GetRecordsで を使用する

- CLI で GetShardIterator を使用する
- AWS SDK ListStreamConsumersで を使用する
- AWS SDK または CLI ListStreamsで を使用する
- AWS SDK または CLI ListTagsForStreamで を使用する
- AWS SDK または CLI PutRecordで を使用する
- AWS SDK または CLI PutRecordsで を使用する
- AWS SDK または CLI RegisterStreamConsumerで を使用する

AWS SDK または CLI **AddTagsToStream**で を使用する

次のサンプルコードは、AddTagsToStream を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// This example shows how to apply key/value pairs to an Amazon Kinesis
/// stream.
/// </summary>
public class TagStream
{
    public static async Task Main()
    {
        IAmazonKinesis client = new AmazonKinesisClient();

        string streamName = "AmazonKinesisStream";
```

```
            var tags = new Dictionary<string, string>
            {
                { "Project", "Sample Kinesis Project" },
                { "Application", "Sample Kinesis App" },
            };

            var success = await ApplyTagsToStreamAsync(client, streamName, tags);

            if (success)
            {
                Console.WriteLine($"Taggs successfully added to {streamName}.");
            }
            else
            {
                Console.WriteLine("Tags were not added to the stream.");
            }
        }

        /// <summary>
        /// Applies the set of tags to the named Kinesis stream.
        /// </summary>
        /// <param name="client">The initialized Kinesis client.</param>
        /// <param name="streamName">The name of the Kinesis stream to which
        /// the tags will be attached.</param>
        /// <param name="tags">A sictionary containing key/value pairs which
        /// will be used to create the Kinesis tags.</param>
        /// <returns>A Boolean value which represents the success or failure
        /// of AddTagsToStreamAsync.</returns>
        public static async Task<bool> ApplyTagsToStreamAsync(
            IAmazonKinesis client,
            string streamName,
            Dictionary<string, string> tags)
        {
            var request = new AddTagsToStreamRequest
            {
                StreamName = streamName,
                Tags = tags,
            };

            var response = await client.AddTagsToStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「AddTagsToStream」を参照してください。

CLI

AWS CLI

データストリームにタグを追加するには

次の add-tags-to-stream の例では、キー samplekey と値 example を持つタグを指定されたストリームに割り当てます。

```
aws kinesis add-tags-to-stream \
    --stream-name samplestream \
    --tags samplekey=example
```

このコマンドは何も出力しません。

詳細については、「Amazon Kinesis Data Streams Developer Guide」の「Tagging Your Streams」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「AddTagsToStream」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **CreateStream**で を使用する

次のサンプルコードは、CreateStream を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using System;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// This example shows how to create a new Amazon Kinesis stream.
/// </summary>
public class CreateStream
{
    public static async Task Main()
    {
        IAmazonKinesis client = new AmazonKinesisClient();

        string streamName = "AmazonKinesisStream";
        int shardCount = 1;

        var success = await CreateNewStreamAsync(client, streamName,
shardCount);
        if (success)
        {
            Console.WriteLine($"The stream, {streamName} successfully
created.");
        }
    }

    /// <summary>
    /// Creates a new Kinesis stream.
    /// </summary>
    /// <param name="client">An initialized Kinesis client.</param>
    /// <param name="streamName">The name for the new stream.</param>
    /// <param name="shardCount">The number of shards the new stream will
```

```
        /// use. The throughput of the stream is a function of the number of
        /// shards; more shards are required for greater provisioned
        /// throughput.</param>
        /// <returns>A Boolean value indicating whether the stream was created.</
returns>
        public static async Task<bool> CreateNewStreamAsync(IAmazonKinesis
 client, string streamName, int shardCount)
        {
            var request = new CreateStreamRequest
            {
                StreamName = streamName,
                ShardCount = shardCount,
            };

            var response = await client.CreateStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- API の詳細については、AWS SDK for .NET API リファレンスの「CreateStream」を参照してください。

CLI

AWS CLI

データストリームを作成するには

次の `create-stream` の例は、3 つのシャードを含む samplestream という名前のデータストリームを作成します。

```
aws kinesis create-stream \
    --stream-name samplestream \
    --shard-count 3
```

このコマンドは何も出力しません。

詳細については、「Amazon Kinesis Data Streams ディベロッパーガイド」の「API を使用したストリームの作成」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「CreateStream」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.CreateStreamRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class CreateDataStream {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream (for example,
 StockTradeStream).
                """;

        if (args.length != 1) {
```

```
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();
        createStream(kinesisClient, streamName);
        System.out.println("Done");
        kinesisClient.close();
    }

    public static void createStream(KinesisClient kinesisClient, String
 streamName) {
        try {
            CreateStreamRequest streamReq = CreateStreamRequest.builder()
                    .streamName(streamName)
                    .shardCount(1)
                    .build();

            kinesisClient.createStream(streamReq);

        } catch (KinesisException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「CreateStream」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 新しいストリームを作成します。デフォルトでは、このコマンドレットは出力を返さないため、後で使用するために -StreamName パラメータに指定された値を返すために -PassThru スイッチが追加されます。

```
$streamName = New-KINStream -StreamName "mystream" -ShardCount 1 -PassThru
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の CreateStream を参照してください。

Tools for PowerShell V5

例 1: 新しいストリームを作成します。

```
New-KINStream -StreamName "mystream" -ShardCount 1
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の CreateStream を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def create(self, name, wait_until_exists=True):
        """
        Creates a stream.
```

```
        :param name: The name of the stream.
        :param wait_until_exists: When True, waits until the service reports that
                                  the stream exists, then queries for its
 metadata.
        """
        try:
            self.kinesis_client.create_stream(StreamName=name, ShardCount=1)
            self.name = name
            logger.info("Created stream %s.", name)
            if wait_until_exists:
                logger.info("Waiting until exists.")
                self.stream_exists_waiter.wait(StreamName=name)
                self.describe(name)
        except ClientError:
            logger.exception("Couldn't create stream %s.", name)
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「CreateStream」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn make_stream(client: &Client, stream: &str) -> Result<(), Error> {
    client
        .create_stream()
        .stream_name(stream)
        .shard_count(4)
        .send()
        .await?;

    println!("Created stream");
```

```
    Ok(())
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「CreateStream」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       lo_kns->createstream(
           iv_streamname = iv_stream_name
           iv_shardcount = iv_shard_count ).
       MESSAGE 'Stream created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
     CATCH /aws1/cx_knsresourceinuseex.
       MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「CreateStream」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **DeleteStream**で を使用する

次のサンプルコードは、DeleteStream を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using System;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// Shows how to delete an Amazon Kinesis stream.
/// </summary>
public class DeleteStream
{
    public static async Task Main()
    {
        IAmazonKinesis client = new AmazonKinesisClient();
        string streamName = "AmazonKinesisStream";

        var success = await DeleteStreamAsync(client, streamName);

        if (success)
        {
            Console.WriteLine($"Stream, {streamName} successfully deleted.");
        }
        else
```

```
            {
                Console.WriteLine("Stream not deleted.");
            }
        }

        /// <summary>
        /// Deletes a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamName">The name of the string to delete.</param>
        /// <returns>A Boolean value representing the success of the operation.</
returns>
        public static async Task<bool> DeleteStreamAsync(IAmazonKinesis client,
 string streamName)
        {
            // If EnforceConsumerDeletion is true, any consumers
            // of this stream will also be deleted. If it is set
            // to false and this stream has any consumers, the
            // call will fail with a ResourceInUseException.
            var request = new DeleteStreamRequest
            {
                StreamName = streamName,
                EnforceConsumerDeletion = true,
            };

            var response = await client.DeleteStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeleteStream」を参照してください。

CLI

AWS CLI

データストリームを削除するには

次の delete-stream の例は、指定されたデータストリームを削除します。

```
aws kinesis delete-stream \
    --stream-name samplestream
```

このコマンドは何も出力しません。

詳細については、「Amazon Kinesis Data Streams ディベロッパーガイド」の「ストリームを削除する」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「DeleteStream」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.DeleteStreamRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DeleteDataStream {

    public static void main(String[] args) {
        final String usage = """

                Usage:
```

```
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream (for example,
 StockTradeStream)
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();

        deleteStream(kinesisClient, streamName);
        kinesisClient.close();
        System.out.println("Done");
    }

    public static void deleteStream(KinesisClient kinesisClient, String
 streamName) {
        try {
            DeleteStreamRequest delStream = DeleteStreamRequest.builder()
                    .streamName(streamName)
                    .build();

            kinesisClient.deleteStream(delStream);

        } catch (KinesisException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「DeleteStream」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたストリームを削除します。コマンドを実行する前に確認を求められます。確認メッセージを非表示にするには -Force スイッチを使用します。

```
Remove-KINStream -StreamName "mystream"
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の DeleteStream を参照してください。

Tools for PowerShell V5

例 1: 指定されたストリームを削除します。コマンドを実行する前に確認を求められます。確認メッセージを非表示にするには -Force スイッチを使用します。

```
Remove-KINStream -StreamName "mystream"
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の DeleteStream を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
```

```
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def delete(self):
        """
        Deletes a stream.
        """
        try:
            self.kinesis_client.delete_stream(StreamName=self.name)
            self._clear()
            logger.info("Deleted stream %s.", self.name)
        except ClientError:
            logger.exception("Couldn't delete stream %s.", self.name)
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「DeleteStream」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn remove_stream(client: &Client, stream: &str) -> Result<(), Error> {
    client.delete_stream().stream_name(stream).send().await?;

    println!("Deleted stream.");

    Ok(())
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「DeleteStream」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        lo_kns->deletestream(
            iv_streamname = iv_stream_name ).
        MESSAGE 'Stream deleted.' TYPE 'I'.
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourceinuseex.
        MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DeleteStream」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **DeregisterStreamConsumer**で を使用する

次のサンプルコードは、DeregisterStreamConsumer を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// Shows how to deregister a consumer from an Amazon Kinesis stream.
    /// </summary>
    public class DeregisterConsumer
    {
        public static async Task Main(string[] args)
        {
            IAmazonKinesis client = new AmazonKinesisClient();

            string streamARN = "arn:aws:kinesis:us-west-2:000000000000:stream/
AmazonKinesisStream";
            string consumerName = "CONSUMER_NAME";
            string consumerARN = "arn:aws:kinesis:us-west-2:000000000000:stream/
AmazonKinesisStream/consumer/CONSUMER_NAME:000000000000";

            var success = await DeregisterConsumerAsync(client, streamARN,
 consumerARN, consumerName);

            if (success)
            {
                Console.WriteLine($"{consumerName} successfully deregistered.");
            }
            else
            {
                Console.WriteLine($"{consumerName} was not successfully
 deregistered.");
            }
```

```
        }

        /// <summary>
        /// Deregisters a consumer from a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamARN">The ARN of a Kinesis stream.</param>
        /// <param name="consumerARN">The ARN of the consumer.</param>
        /// <param name="consumerName">The name of the consumer.</param>
        /// <returns>A Boolean value representing the success of the operation.</
returns>
        public static async Task<bool> DeregisterConsumerAsync(
            IAmazonKinesis client,
            string streamARN,
            string consumerARN,
            string consumerName)
        {
            var request = new DeregisterStreamConsumerRequest
            {
                StreamARN = streamARN,
                ConsumerARN = consumerARN,
                ConsumerName = consumerName,
            };

            var response = await client.DeregisterStreamConsumerAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeregisterStreamConsumer」を参照してください。

CLI

AWS CLI

データストリームコンシューマーの登録を解除するには

次の `deregister-stream-consumer` 例では、指定されたコンシューマーを指定されたデータストリームから登録解除します。

```
aws kinesis deregister-stream-consumer \
    --stream-arn arn:aws:kinesis:us-west-2:123456789012:stream/samplestream \
    --consumer-name KinesisConsumerApplication
```

このコマンドでは何も出力されません。

詳細については、「Amazon Kinesis Data Streams Developer Guide」の「Developing Consumers with Enhanced Fan-Out Using the Kinesis Data Streams API」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeregisterStreamConsumer」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **DescribeStream**で を使用する

次のサンプルコードは、DescribeStream を使用する方法を説明しています。

CLI

AWS CLI

データストリームを記述するには

次の describe-stream の例は、指定されたデータストリームの詳細を返します。

```
aws kinesis describe-stream \
    --stream-name samplestream
```

出力:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "113427455640312821154458202477256070484"
```

```
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49600871682957036442365024926191073437251060580128653314"
                }
            },
            {
                "ShardId": "shardId-000000000001",
                "HashKeyRange": {
                    "StartingHashKey": "113427455640312821154458202477256070485",
                    "EndingHashKey": "226854911280625642308916404954512140969"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49600871682979337187563555549332609155523708941634633746"
                }
            },
            {
                "ShardId": "shardId-000000000002",
                "HashKeyRange": {
                    "StartingHashKey": "226854911280625642308916404954512140970",
                    "EndingHashKey": "340282366920938463463374607431768211455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49600871683001637932762086172474144873796357303140614178"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/
samplestream",
        "StreamName": "samplestream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1572297168.0
    }
```

```
}
```

詳細については、「Amazon Kinesis Data Streams ディベロッパーガイド」の「ストリームの作成と管理」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「DescribeStream」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたストリームの詳細を返します。

```
Get-KINStream -StreamName "mystream"
```

出力:

```
HasMoreShards        : False
RetentionPeriodHours : 24
Shards               : {}
StreamARN            : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName           : mystream
StreamStatus         : ACTIVE
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の DescribeStream を参照してください。

Tools for PowerShell V5

例 1: 指定されたストリームの詳細を返します。

```
Get-KINStream -StreamName "mystream"
```

出力:

```
HasMoreShards        : False
RetentionPeriodHours : 24
Shards               : {}
StreamARN            : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName           : mystream
```

```
StreamStatus         : ACTIVE
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の DescribeStream を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def describe(self, name):
        """
        Gets metadata about a stream.

        :param name: The name of the stream.
        :return: Metadata about the stream.
        """
        try:
            response = self.kinesis_client.describe_stream(StreamName=name)
            self.name = name
            self.details = response["StreamDescription"]
            logger.info("Got stream %s.", name)
        except ClientError:
```

```
            logger.exception("Couldn't get %s.", name)
            raise
        else:
            return self.details
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「DescribeStream」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn show_stream(client: &Client, stream: &str) -> Result<(), Error> {
    let resp = client.describe_stream().stream_name(stream).send().await?;

    let desc = resp.stream_description.unwrap();

    println!("Stream description:");
    println!("  Name:              {}:", desc.stream_name());
    println!("  Status:            {:?}", desc.stream_status());
    println!("  Open shards:       {:?}", desc.shards.len());
    println!("  Retention (hours): {}", desc.retention_period_hours());
    println!("  Encryption:        {:?}", desc.encryption_type.unwrap());

    Ok(())
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「DescribeStream」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_result = lo_kns->describestream(
            iv_streamname = iv_stream_name ).
        DATA(lt_stream_description) = oo_result->get_streamdescription( ).
        MESSAGE 'Streams retrieved.' TYPE 'I'.
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DescribeStream」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **GetRecords**で を使用する

次のサンプルコードは、GetRecords を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

CLI

AWS CLI

シャードからレコードを取得するには

次の get-records の例は、指定されたシャードイテレーターを使用して Kinesis データストリームのシャードからデータレコードを取得します。

```
aws kinesis get-records \
    --shard-iterator AAAAAAAAAAF7/0mWD7IuHj1yGv/
TKuNgx2ukD5xipCY4cy4gU96orWwZwcSXh3K9tAmGYeOZyLZrvzzeOFVf9iN99hUPw/w/
b0YWYeehfNvnf1DYt5XpDJghLKr3DzgznkTmMymDP3R+3wRKeuEw6/kdxY2yKJH0veaiekaVc4N2VwK/
GvaGP2Hh9Fg7N++q0Adg6fIDQPt4p8RpavDbk+A4sL9SWGE1
```

出力:

```
{
    "Records": [],
    "MillisBehindLatest": 80742000
}
```

詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の AWS 「 SDK for Java で Kinesis Data Streams API を使用してコンシューマーを開発する」を参照してください。 Amazon Kinesis

- API の詳細については、AWS CLI コマンドリファレンスの「GetRecords」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
```

```
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamResponse;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamRequest;
import software.amazon.awssdk.services.kinesis.model.Shard;
import software.amazon.awssdk.services.kinesis.model.GetShardIteratorRequest;
import software.amazon.awssdk.services.kinesis.model.GetShardIteratorResponse;
import software.amazon.awssdk.services.kinesis.model.Record;
import software.amazon.awssdk.services.kinesis.model.GetRecordsRequest;
import software.amazon.awssdk.services.kinesis.model.GetRecordsResponse;
import java.util.ArrayList;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class GetRecords {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream to read from (for
 example, StockTradeStream).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();
```

```
        getStockTrades(kinesisClient, streamName);
        kinesisClient.close();
    }

    public static void getStockTrades(KinesisClient kinesisClient, String
streamName) {
        String shardIterator;
        String lastShardId = null;
        DescribeStreamRequest describeStreamRequest =
DescribeStreamRequest.builder()
                .streamName(streamName)
                .build();

        List<Shard> shards = new ArrayList<>();
        DescribeStreamResponse streamRes;
        do {
            streamRes = kinesisClient.describeStream(describeStreamRequest);
            shards.addAll(streamRes.streamDescription().shards());

            if (shards.size() > 0) {
                lastShardId = shards.get(shards.size() - 1).shardId();
            }
        } while (streamRes.streamDescription().hasMoreShards());

        GetShardIteratorRequest itReq = GetShardIteratorRequest.builder()
                .streamName(streamName)
                .shardIteratorType("TRIM_HORIZON")
                .shardId(lastShardId)
                .build();

        GetShardIteratorResponse shardIteratorResult =
kinesisClient.getShardIterator(itReq);
        shardIterator = shardIteratorResult.shardIterator();

        // Continuously read data records from shard.
        List<Record> records;

        // Create new GetRecordsRequest with existing shardIterator.
        // Set maximum records to return to 1000.
        GetRecordsRequest recordsRequest = GetRecordsRequest.builder()
                .shardIterator(shardIterator)
                .limit(1000)
                .build();
```

```
        GetRecordsResponse result = kinesisClient.getRecords(recordsRequest);

        // Put result into record list. Result may be empty.
        records = result.records();

        // Print records
        for (Record record : records) {
            SdkBytes byteBuffer = record.data();
            System.out.printf("Seq No: %s - %s%n", record.sequenceNumber(), new
 String(byteBuffer.asByteArray()));
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「GetRecords」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: この例では、一連の 1 つ以上のレコードからデータを返して抽出する方法を示しています。Get-KINRecord に供給されたイテレータは、この例で変数 $records にキャプチャされるレコードの開始位置を決定します。その後、$records コレクションにインデックスを付けることで、個々のレコードにアクセスできます。レコード内のデータが UTF-8 でエンコードされたテキストであると仮定すると、最終コマンドは、オブジェクト内の MemoryStream からデータを抽出してテキストとしてコンソールに返す方法を示します。

```
$records
$records = Get-KINRecord -ShardIterator "AAAAAAAAAAGIc....9VnbiRNaP"
```

出力:

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
0                  AAAAAAAAAAERNIq...uDn11HuUs  {Key1, Key2}
```

```
$records.Records[0]
```

出力:

```
ApproximateArrivalTimestamp Data                    PartitionKey SequenceNumber
--------------------------- ----                    ------------ --------------
3/7/2016 5:14:33 PM         System.IO.MemoryStream Key1
 4955986459776...931586
```

```
[Text.Encoding]::UTF8.GetString($records.Records[0].Data.ToArray())
```

出力:

```
test data from string
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の GetRecords を参照してください。

Tools for PowerShell V5

例 1: この例では、一連の 1 つ以上のレコードからデータを返して抽出する方法を示しています。Get-KINRecord に供給されたイテレータは、この例で変数 $records にキャプチャされるレコードの開始位置を決定します。その後、$records コレクションにインデックスを付けることで、個々のレコードにアクセスできます。レコード内のデータが UTF-8 でエンコードされたテキストであると仮定すると、最終コマンドは、オブジェクト内の MemoryStream からデータを抽出してテキストとしてコンソールに返す方法を示します。

```
$records
$records = Get-KINRecord -ShardIterator "AAAAAAAAAAGIc....9VnbiRNaP"
```

出力:

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
0                  AAAAAAAAAAERNIq...uDn11HuUs  {Key1, Key2}
```

```
$records.Records[0]
```

出力:

```
ApproximateArrivalTimestamp Data                    PartitionKey SequenceNumber
```

```
-------------------------- ----                    ------------ --------------
3/7/2016 5:14:33 PM        System.IO.MemoryStream Key1
 4955986459776...931586
```

```
[Text.Encoding]::UTF8.GetString($records.Records[0].Data.ToArray())
```

出力:

```
test data from string
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の GetRecords を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def get_records(self, max_records):
        """
        Gets records from the stream. This function is a generator that first
 gets
```

```
        a shard iterator for the stream, then uses the shard iterator to get
records
        in batches from the stream. The shard iterator can be accessed through
the
        'details' property, which is populated using the 'describe' function of
this class.
        Each batch of records is yielded back to the caller until the specified
        maximum number of records has been retrieved.

        :param max_records: The maximum number of records to retrieve.
        :return: Yields the current batch of retrieved records.
        """
        try:
            response = self.kinesis_client.get_shard_iterator(
                StreamName=self.name,
                ShardId=self.details["Shards"][0]["ShardId"],
                ShardIteratorType="LATEST",
            )
            shard_iter = response["ShardIterator"]
            record_count = 0
            while record_count < max_records:
                response = self.kinesis_client.get_records(
                    ShardIterator=shard_iter, Limit=10
                )
                shard_iter = response["NextShardIterator"]
                records = response["Records"]
                logger.info("Got %s records.", len(records))
                record_count += len(records)
                yield records
        except ClientError:
            logger.exception("Couldn't get records from stream %s.", self.name)
            raise



    def describe(self, name):
        """
        Gets metadata about a stream.

        :param name: The name of the stream.
        :return: Metadata about the stream.
        """
        try:
            response = self.kinesis_client.describe_stream(StreamName=name)
```

```
            self.name = name
            self.details = response["StreamDescription"]
            logger.info("Got stream %s.", name)
        except ClientError:
            logger.exception("Couldn't get %s.", name)
            raise
        else:
            return self.details
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「GetRecords」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_kns->getrecords(             " oo_result is returned for
testing purposes. "
           iv_sharditerator = iv_shard_iterator ).
       DATA(lt_records) = oo_result->get_records( ).
       MESSAGE 'Record retrieved.' TYPE 'I'.
     CATCH /aws1/cx_knsexpirediteratorex.
       MESSAGE 'Iterator expired.' TYPE 'E'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knskmsaccessdeniedex.
       MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
     CATCH /aws1/cx_knskmsdisabledex.
       MESSAGE 'KMS key used is disabled.' TYPE 'E'.
     CATCH /aws1/cx_knskmsinvalidstateex.
       MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
```

```
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「GetRecords」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

CLI で **GetShardIterator** を使用する

次のサンプルコードは、GetShardIterator を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

CLI

AWS CLI

シャードイテレータを取得するには

次の get-shard-iterator の例では、AT_SEQUENCE_NUMBER シャードイテレータタイプを使用してシャードイテレータを生成し、指定されたシーケンス番号で示される位置からデータレコードの読み取りを正確に開始します。

```
aws kinesis get-shard-iterator \
    --stream-name samplestream \
    --shard-id shardId-000000000001 \
    --shard-iterator-type LATEST
```

出力:

```
{
    "ShardIterator": "AAAAAAAAAAFEvJjIYI+3jw/4aqgH9FifJ+n48XWTh/
IFIsbILP6o5eDueD39NXNBfpZ10WL5K6ADXk8w+5H+Qhd9cFA9k268CPXCz/kebq1TGYI7Vy
+lUkA9BuN3xvATxMBGxRY3zYK05gqgvaIRn94O8SqeEqwhigwZxNWxID3Ej7YYYcxQi8Q/fIrCjGAy/
n2r5Z9G864YpWDfN9upNNQAR/iiOWKs"
}
```

詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の AWS 「 SDK for Java で Kinesis Data Streams API を使用してコンシューマーを開発する」を参照してください。 Amazon Kinesis

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetShardIterator」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたシャードと開始位置のシャードイテレータを返します。シャード識別子とシーケンス番号の詳細は、返されたストリームオブジェクトのシャードコレクションを参照することで、Get-KINStream コマンドレットの出力から取得できます。返されたイテレータを Get-KINRecord コマンドレットとともに使用して、シャード内のデータレコードをプルできます。

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

出力:

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の GetShardIterator を参照してください。

Tools for PowerShell V5

例 1: 指定されたシャードと開始位置のシャードイテレータを返します。シャード識別子とシーケンス番号の詳細は、返されたストリームオブジェクトのシャードコレクションを参照することで、Get-KINStream コマンドレットの出力から取得できます。返されたイテレータを Get-KINRecord コマンドレットとともに使用して、シャード内のデータレコードをプルできます。

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

出力:

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の「GetShardIterator」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK **ListStreamConsumers**で を使用する

次の例は、ListStreamConsumers を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
```

```
    using Amazon.Kinesis.Model;

    /// <summary>
    /// List the consumers of an Amazon Kinesis stream.
    /// </summary>
    public class ListConsumers
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();

            string streamARN = "arn:aws:kinesis:us-east-2:000000000000:stream/
AmazonKinesisStream";
            int maxResults = 10;

            var consumers = await ListConsumersAsync(client, streamARN,
 maxResults);

            if (consumers.Count > 0)
            {
                consumers
                    .ForEach(c => Console.WriteLine($"Name: {c.ConsumerName} ARN:
 {c.ConsumerARN}"));
            }
            else
            {
                Console.WriteLine("No consumers found.");
            }
        }

        /// <summary>
        /// Retrieve a list of the consumers for a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamARN">The ARN of the stream for which we want to
        /// retrieve a list of clients.</param>
        /// <param name="maxResults">The maximum number of results to return.</
param>
        /// <returns>A list of Consumer objects.</returns>
        public static async Task<List<Consumer>>
 ListConsumersAsync(IAmazonKinesis client, string streamARN, int maxResults)
        {
            var request = new ListStreamConsumersRequest
            {
```

```
                StreamARN = streamARN,
                MaxResults = maxResults,
            };

            var response = await client.ListStreamConsumersAsync(request);

            return response.Consumers;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の ListStreamConsumersを参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **ListStreams**で を使用する

次のサンプルコードは、ListStreams を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// Retrieves and displays a list of existing Amazon Kinesis streams.
```

```
    /// </summary>
    public class ListStreams
    {
        public static async Task Main(string[] args)
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            var response = await client.ListStreamsAsync(new
 ListStreamsRequest());

            List<string> streamNames = response.StreamNames;

            if (streamNames.Count > 0)
            {
                streamNames
                    .ForEach(s => Console.WriteLine($"Stream name: {s}"));
            }
            else
            {
                Console.WriteLine("No streams were found.");
            }
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListStreams」を参照してください。

CLI

AWS CLI

データストリームを一覧表示するには

次の `list-streams` の例は、現在のアカウントとリージョンのすべてのアクティブなデータストリームを一覧表示します。

```
aws kinesis list-streams
```

出力:

```
{
```

```
    "StreamNames": [
        "samplestream",
        "samplestream1"
    ]
}
```

詳細については、「Amazon Kinesis Data Streams ディベロッパーガイド」の「ストリームのリスト」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「ListStreams」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn show_streams(client: &Client) -> Result<(), Error> {
    let resp = client.list_streams().send().await?;

    println!("Stream names:");

    let streams = resp.stream_names;
    for stream in &streams {
        println!("  {}", stream);
    }

    println!("Found {} stream(s)", streams.len());

    Ok(())
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「ListStreams」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_kns->liststreams(        " oo_result is returned for
testing purposes. "
           "Set Limit to specify that a maximum of streams should be returned."
           iv_limit = iv_limit ).
       DATA(lt_streams) = oo_result->get_streamnames( ).
       MESSAGE 'Streams listed.' TYPE 'I'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「ListStreams」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **ListTagsForStream**で を使用する

次のサンプルコードは、ListTagsForStream を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    using System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// Shows how to list the tags that have been attached to an Amazon Kinesis
    /// stream.
    /// </summary>
    public class ListTags
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            string streamName = "AmazonKinesisStream";

            await ListTagsAsync(client, streamName);
        }

        /// <summary>
        /// List the tags attached to a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamName">The name of the Kinesis stream for which you
        /// wish to display tags.</param>
        public static async Task ListTagsAsync(IAmazonKinesis client, string
streamName)
        {
            var request = new ListTagsForStreamRequest
            {
                StreamName = streamName,
```

```
                Limit = 10,
            };

            var response = await client.ListTagsForStreamAsync(request);
            DisplayTags(response.Tags);

            while (response.HasMoreTags)
            {
                request.ExclusiveStartTagKey = response.Tags[response.Tags.Count
- 1].Key;
                response = await client.ListTagsForStreamAsync(request);
            }
        }

        /// <summary>
        /// Displays the items in a list of Kinesis tags.
        /// </summary>
        /// <param name="tags">A list of the Tag objects to be displayed.</param>
        public static void DisplayTags(List<Tag> tags)
        {
            tags
                .ForEach(t => Console.WriteLine($"Key: {t.Key} Value:
{t.Value}"));
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListTagsForStream」を参照してください。

CLI

AWS CLI

データストリームのタグを一覧表示するには

次の list-tags-for-stream の例では、指定されたデータストリームにアタッチされたタグを一覧表示します。

```
aws kinesis list-tags-for-stream \
    --stream-name samplestream
```

出力:

```
{
    "Tags": [
        {
            "Key": "samplekey",
            "Value": "example"
        }
    ],
    "HasMoreTags": false
}
```

詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の「ストリームのタグ付け」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListTagsForStream」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **PutRecord**で を使用する

次のサンプルコードは、PutRecord を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

CLI

AWS CLI

データストリームにデータレコードを書き込むには

次の put-record の例は、指定されたパーティションキーを使用して、指定されたデータストリームに単一のデータレコードを書き込みます。

```
aws kinesis put-record \
```

```
    --stream-name samplestream \
    --data sampledatarecord \
    --partition-key samplepartitionkey
```

出力:

```
{
    "ShardId": "shardId-000000000009",
    "SequenceNumber": "49600902273357540915989931256901506243878407835297513618",
    "EncryptionType": "KMS"
}
```

詳細については、Amazon Kinesis Data Streams デベロッパーガイド」の「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」を参照してください。 Amazon Kinesis

- API の詳細については、AWS CLI コマンドリファレンスの「PutRecord」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamRequest;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamResponse;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
```

```
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class StockTradesWriter {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream to which records
 are written (for example, StockTradeStream)
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();

        // Ensure that the Kinesis Stream is valid.
        validateStream(kinesisClient, streamName);
        setStockData(kinesisClient, streamName);
        kinesisClient.close();
    }

    public static void setStockData(KinesisClient kinesisClient, String
 streamName) {
        try {
            // Repeatedly send stock trades with a 100 milliseconds wait in
 between.
            StockTradeGenerator stockTradeGenerator = new StockTradeGenerator();

            // Put in 50 Records for this example.
            int index = 50;
            for (int x = 0; x < index; x++) {
```

```
                StockTrade trade = stockTradeGenerator.getRandomTrade();
                sendStockTrade(trade, kinesisClient, streamName);
                Thread.sleep(100);
            }

        } catch (KinesisException | InterruptedException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Done");
    }

    private static void sendStockTrade(StockTrade trade, KinesisClient
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) {
            System.out.println("Could not get JSON bytes for stock trade");
            return;
        }

        System.out.println("Putting trade: " + trade);
        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);
        } catch (KinesisException e) {
            System.err.println(e.getMessage());
        }
    }

    private static void validateStream(KinesisClient kinesisClient, String
streamName) {
```

```
        try {
            DescribeStreamRequest describeStreamRequest =
 DescribeStreamRequest.builder()
                    .streamName(streamName)
                    .build();

            DescribeStreamResponse describeStreamResponse =
 kinesisClient.describeStream(describeStreamRequest);

            if (!
describeStreamResponse.streamDescription().streamStatus().toString().equals("ACTIVE"))
 {
                System.err.println("Stream " + streamName + " is not active.
 Please wait a few moments and try again.");
                System.exit(1);
            }

        } catch (KinesisException e) {
            System.err.println("Error found while describing the stream " +
 streamName);
            System.err.println(e);
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「PutRecord」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: -Text パラメータに指定された文字列を含むレコードを書き込みます。

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

例 2: 指定されたファイルに含まれるデータを含むレコードを書き込みます。ファイルはバイトのシーケンスとして扱われるため、テキストが含まれている場合は、このコマンドレットで使用する前に、必要なエンコードで記述する必要があります。

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
 "Key2"
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の PutRecord を参照してください。

Tools for PowerShell V5

例 1: -Text パラメータに指定された文字列を含むレコードを書き込みます。

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

例 2: 指定されたファイルに含まれるデータを含むレコードを書き込みます。ファイルはバイトのシーケンスとして扱われるため、テキストが含まれている場合は、このコマンドレットで使用する前に、必要なエンコードで記述する必要があります。

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
 "Key2"
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V5) の PutRecord を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
```

```
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def put_record(self, data, partition_key):
        """
        Puts data into the stream. The data is formatted as JSON before it is
passed
        to the stream.

        :param data: The data to put in the stream.
        :param partition_key: The partition key to use for the data.
        :return: Metadata about the record, including its shard ID and sequence
number.
        """
        try:
            response = self.kinesis_client.put_record(
                StreamName=self.name, Data=json.dumps(data),
PartitionKey=partition_key
            )
            logger.info("Put record in stream %s.", self.name)
        except ClientError:
            logger.exception("Couldn't put record in stream %s.", self.name)
            raise
        else:
            return response
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「PutRecord」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn add_record(client: &Client, stream: &str, key: &str, data: &str) ->
 Result<(), Error> {
    let blob = Blob::new(data);

    client
        .put_record()
        .data(blob)
        .partition_key(key)
        .stream_name(stream)
        .send()
        .await?;

    println!("Put data into stream.");

    Ok(())
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「PutRecord」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_kns->putrecord(            " oo_result is returned for
testing purposes. "
           iv_streamname = iv_stream_name
           iv_data       = iv_data
           iv_partitionkey = iv_partition_key ).
       MESSAGE 'Record created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knskmsaccessdeniedex.
       MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
     CATCH /aws1/cx_knskmsdisabledex.
       MESSAGE 'KMS key used is disabled.' TYPE 'E'.
     CATCH /aws1/cx_knskmsinvalidstateex.
       MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
     CATCH /aws1/cx_knskmsnotfoundex.
       MESSAGE 'KMS key used is not found.' TYPE 'E'.
     CATCH /aws1/cx_knskmsoptinrequired.
       MESSAGE 'KMS key option is required.' TYPE 'E'.
     CATCH /aws1/cx_knskmsthrottlingex.
       MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
     CATCH /aws1/cx_knsprovthruputexcdex.
       MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
     CATCH /aws1/cx_knsresourcenotfoundex.
       MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「PutRecord」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **PutRecords**で を使用する

次のサンプルコードは、PutRecords を使用する方法を説明しています。

CLI

AWS CLI

複数のレコードをデータストリームに書き込むには

次の put-records の例では、指定されたパーティションキーを使用してデータレコードを書き込み、1 回の呼び出しで別のパーティションキーを使用して別のデータレコードを書き込みます。

```
aws kinesis put-records \
    --stream-name samplestream \
    --
records Data=blob1,PartitionKey=partitionkey1 Data=blob2,PartitionKey=partitionkey2
```

出力:

```
{
    "FailedRecordCount": 0,
    "Records": [
        {
            "SequenceNumber":
 "49600883331171471519674795588238531498465399900093808706",
            "ShardId": "shardId-000000000004"
        },
        {
            "SequenceNumber":
 "49600902273357540915989931256902715169698037101720764562",
            "ShardId": "shardId-000000000009"
        }
    ],
    "EncryptionType": "KMS"
}
```

詳細については、Amazon Kinesis Data Streams デベロッパーガイド」の「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」を参照してください。 Amazon Kinesis

- API の詳細については、AWS CLI コマンドリファレンスの「PutRecords」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { PutRecordsCommand, KinesisClient } from "@aws-sdk/client-kinesis";

/**
 * Put multiple records into a Kinesis stream.
 * @param {{ streamArn: string }} config
 */
export const main = async ({ streamArn }) => {
  const client = new KinesisClient({});
  try {
    await client.send(
      new PutRecordsCommand({
        StreamARN: streamArn,
        Records: [
          {
            Data: new Uint8Array(),
            /**
             * Determines which shard in the stream the data record is assigned
 to.
             * Partition keys are Unicode strings with a maximum length limit of
 256
             * characters for each key. Amazon Kinesis Data Streams uses the
 partition
             * key as input to a hash function that maps the partition key and
             * associated data to a specific shard.
             */
            PartitionKey: "TEST_KEY",
          },
          {
            Data: new Uint8Array(),
            PartitionKey: "TEST_KEY",
          },
```

```
        ],
      }),
    );
  } catch (caught) {
    if (caught instanceof Error) {
      //
    } else {
      throw caught;
    }
  }
};

// Call function if run directly.
import { fileURLToPath } from "node:url";
import { parseArgs } from "node:util";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const options = {
    streamArn: {
      type: "string",
      description: "The ARN of the stream.",
    },
  };

  const { values } = parseArgs({ options });
  main(values);
}
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「PutRecords」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI **RegisterStreamConsumer**で を使用する

次のサンプルコードは、RegisterStreamConsumer を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// This example shows how to register a consumer to an Amazon Kinesis
    /// stream.
    /// </summary>
    public class RegisterConsumer
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            string consumerName = "NEW_CONSUMER_NAME";
            string streamARN = "arn:aws:kinesis:us-east-2:000000000000:stream/
AmazonKinesisStream";

            var consumer = await RegisterConsumerAsync(client, consumerName,
 streamARN);

            if (consumer is not null)
            {
                Console.WriteLine($"{consumer.ConsumerName}");
            }
        }

        /// <summary>
        /// Registers the consumer to a Kinesis stream.
        /// </summary>
        /// <param name="client">The initialized Kinesis client object.</param>
```

```
        /// <param name="consumerName">A string representing the consumer.</
param>
        /// <param name="streamARN">The ARN of the stream.</param>
        /// <returns>A Consumer object that contains information about the
 consumer.</returns>
        public static async Task<Consumer> RegisterConsumerAsync(IAmazonKinesis
 client, string consumerName, string streamARN)
        {
            var request = new RegisterStreamConsumerRequest
            {
                ConsumerName = consumerName,
                StreamARN = streamARN,
            };

            var response = await client.RegisterStreamConsumerAsync(request);
            return response.Consumer;
        }
    }
```

- API の詳細については、AWS SDK for .NET API リファレンスの「[スRegisterStreamConsumer]」を参照してください。

CLI

AWS CLI

データストリームコンシューマーを登録するには

次の register-stream-consumer の例では、KinesisConsumerApplication というコンシューマーを指定されたデータストリームに登録します。

```
aws kinesis register-stream-consumer \
    --stream-arn arn:aws:kinesis:us-west-2:012345678912:stream/samplestream \
    --consumer-name KinesisConsumerApplication
```

出力:

```
{
    "Consumer": {
```

```
        "ConsumerName": "KinesisConsumerApplication",
        "ConsumerARN": "arn:aws:kinesis:us-west-2: 123456789012:stream/
samplestream/consumer/KinesisConsumerApplication:1572383852",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1572383852.0
    }
}
```

詳細については、「Amazon Kinesis Data Streams ディベロッパーガイド」の「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」を参照してください。

- API の詳細については、AWS CLI API リファレンスの「スRegisterStreamConsumer」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_kns->registerstreamconsumer(       " oo_result is returned
for testing purposes. "
           iv_streamarn = iv_stream_arn
           iv_consumername = iv_consumer_name ).
       MESSAGE 'Stream consumer registered.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_sgmresourcelimitexcd.
       MESSAGE 'You have reached the limit on the number of resources.' TYPE
'E'.
     CATCH /aws1/cx_sgmresourceinuse.
       MESSAGE 'Resource being accessed is in use.' TYPE 'E'.
     CATCH /aws1/cx_sgmresourcenotfound.
       MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「RegisterStreamConsumer」を参照してください。

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Kinesis のサーバーレス例

次のコード例は、 AWS SDKs で Kinesis を使用する方法を示しています。

例

- Kinesis トリガーから Lambda 関数を呼び出す
- Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

Kinesis トリガーから Lambda 関数を呼び出す

次のコード例では、Kinesis ストリームからレコードを受信することによってトリガーされるイベントを受け取る、Lambda 関数の実装方法を示しています。この関数は Kinesis ペイロードを取得し、それを Base64 からデコードして、そのレコードの内容を口グ記録します。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

.NET を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
```

```
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
```

```
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Go を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
```

```
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Java を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
```

```
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

JavaScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
```

```
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
```

```
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

PHP を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
```

```
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Python を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Ruby を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
```

```
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Rust を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
```

```
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

以下のコード例では、Kinesis ストリームからイベントを受け取る Lambda 関数のための、部分的なバッチレスポンスの実装方法を示しています。この関数は、レスポンスとしてバッチアイテムの失敗を報告し、対象のメッセージを後で再試行するよう Lambda に伝えます。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

.NET を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
```

```
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
{record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
item immediately.
                   Lambda will immediately begin to retry processing from this
failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
{ ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
```

```
            string data = Encoding.UTF8.GetString(bytes);
            await Task.CompletedTask; //Placeholder for actual async work
            return data;
        }
    }

    public class StreamsEventResponse
    {
        [JsonPropertyName("batchItemFailures")]
        public IList<BatchItemFailure> BatchItemFailures { get; set; }
        public class BatchItemFailure
        {
            [JsonPropertyName("itemIdentifier")]
            public string ItemIdentifier { get; set; }
        }
    }
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Go を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
```

```
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Java を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Javascript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};
```

```
async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

PHP を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }
```

```
    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Python を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Ruby を使用して Lambda で Kinesis バッチアイテム失敗のレポートをします。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Rust を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
```

```
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();
```

```
    run(service_fn(function_handler)).await
}
```

AWS SDK 開発者ガイドとコード例の完全なリストについては、「」を参照してくださいAWS SDK でのこのサービスの使用。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

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
による耐障害性テストのサポートを追加しました AWS Fault Injection Service。	でレジリエンステストを実行する AWS Fault Injection Service が追加されました。	2025 年 10 月 15 日
最大 10 MiB の大きなレコードサイズのサポートが追加されました。	大きなレコードの処理 が追加されました。	2025 年 10 月 27 日
KPL および KCL のサポート終了日とバージョンライフサイクルポリシー。	Amazon Kinesis Producer Library (KPL) および Amazon Kinesis Client Library (KCL) のサポート終了日とバージョンライフサイクルポリシーに関する情報を追加しました。詳細については、「KPL バージョンライフサイクルポリシー」および「KCL バージョンライフサイクルポリシー」を参照してください。	2025 年 3 月 13 日
アカウント間でのデータストリーム共有のサポートを追加しました。	別のアカウントとのデータストリームを共有する 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。	適切なストリームモードを選ぶ が追加されました。	2021 年 11 月 29 日

変更	説明	変更日
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
拡張 CloudWatch メトリクスの新しいコンテンツ。	Kinesis データストリームを監視する を更新しました。	2016 年 4 月 19 日
拡張 Kinesis エージェントの新しいコンテンツ。	Kinesis エージェントを使用して Amazon Kinesis Data Streams に書き込む を更新しました。	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て Amazon Kinesis Data Streams に書き込むが追加されました。	2015 年 10 月 2 日
リリース 0.10.0 への KPL コンテンツの更新	Amazon Kinesis Producer Library (KPL) を使用してプロデューサーを開発するが追加されました。	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 CloudWatch を使用した Kinesis Client Library を監視するが追加されました。	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 (KPL) を使用してプロデューサーを開発するが追加されました。	2015 年 6 月 02 日
新しい KCL メトリクスのトピック。	Amazon CloudWatch を使用した Kinesis Client Library を監視するが追加されました。	2015 年 5 月 19 日

変更	説明	変更日
KCL .NET のサポート	.NET で Kinesis Client Library コンシューマーを開発するが追加されました。	2015 年 5 月 1 日
KCL Node.js のサポート	ode.js で Kinesis Client Library コンシューマーを開発するが追加されました。	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	the section called “PutRecords を使用して複数のレコードを追加する”に、新しい PutRecords API に関する情報を追加しました。	2014 年 12 月 15 日
タグ指定のサポート	Amazon Kinesis Data Streams リソースにタグを付けるが追加されました。	2014 年 9 月 11 日
新しい CloudWatch 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	Kinesis データストリームを監視する と Amazon CloudWatch による Amazon Kinesis Data Streams サービスを監視する 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

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。