

Amazon Kinesis Data Streams



Amazon Kinesis Data Streams: 开发人员指南

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Amazon Web Services 文档中描述的 Amazon Web Services 服务或功能可能因区域而异。要查看适用于中国区域的差异，请参阅 [中国的 Amazon Web Services 服务入门 \(PDF\)](#)。

Table of Contents

| | |
|---|----|
| 什么是 Amazon Kinesis Data Streams ? | 1 |
| 我能用 Kinesis Data Streams 做什么 ? | 1 |
| 使用 Kinesis Data Streams 的优点 | 2 |
| 相关服务 | 2 |
| 术语和概念 | 3 |
| 查看 Kinesis Data Streams 的高级架构 | 3 |
| 熟悉 Kinesis Data Streams 的术语 | 3 |
| Kinesis 数据流 | 3 |
| 数据记录 | 4 |
| 容量模式 | 4 |
| 保留周期 | 4 |
| Producer | 4 |
| 消费端 | 4 |
| Amazon Kinesis Data Streams 应用程序 | 4 |
| 分片 | 5 |
| 分区键 | 5 |
| 序列号 | 5 |
| Kinesis Client Library | 6 |
| 应用程序名称 | 6 |
| 服务器端加密 | 6 |
| 限额和限制 | 7 |
| API 限制 | 9 |
| KDS 控制层面 API 限制 | 9 |
| 数据层面 API 限制 | 12 |
| 提升配额 | 14 |
| 完成设置 Amazon Kinesis Data Streams 的先决条件 | 15 |
| 注册 Amazon | 15 |
| 下载库和工具 | 15 |
| 配置您的开发环境 | 16 |
| 使用 Amazon CLI 执行 Amazon Kinesis Data Streams 操作 | 17 |
| 教程 : 为 Kinesis Data Streams 安装和配置 Amazon CLI | 17 |
| 安装 Amazon CLI | 17 |
| 配置 Amazon CLI | 18 |
| 教程 : 使用 Amazon CLI 执行基本 Kinesis Data Streams 操作 | 19 |

| | |
|---|----|
| 第 1 步：创建流 | 19 |
| 步骤 2：放置记录 | 21 |
| 步骤 3：获取记录 | 21 |
| 步骤 4：清除 | 24 |
| 入门教程 | 25 |
| 教程：使用 KPL 和 KCL 2.x 处理实时股票数据 | 25 |
| 满足先决条件 | 26 |
| 创建数据流 | 27 |
| 创建 IAM 策略和用户 | 27 |
| 下载并构建代码 | 32 |
| 实现产生器 | 33 |
| 实现消费端 | 37 |
| (可选) 扩展消费端 | 41 |
| 清理 资源 | 43 |
| 教程：使用 KPL 和 KCL 1.x 处理实时股票数据 | 44 |
| 满足先决条件 | 45 |
| 创建数据流 | 46 |
| 创建 IAM 策略和用户 | 47 |
| 下载和构建实现代码 | 52 |
| 实现产生器 | 53 |
| 实现消费端 | 57 |
| (可选) 扩展消费端 | 60 |
| 清理 资源 | 62 |
| 教程：使用适用于 Apache Flink 的亚马逊托管服务分析实时股票数据 | 63 |
| 先决条件 | 64 |
| 步骤 1：设置 账户 | 64 |
| 第 2 步：设置 Amazon CLI | 67 |
| 步骤 3：创建应用程序 | 69 |
| 教程：将 Amazon Lambda 与 Amazon Kinesis Data Streams 搭配使用 | 85 |
| 使用适用于 Amazon Kinesis 的Amazon流数据解决方案 | 85 |
| 创建和管理 Kinesis 数据流 | 87 |
| 选择正确的流式传入的模式 | 87 |
| Kinesis Data Streams 中有哪些不同的模式？ | 88 |
| 按需标准模式的特征和用例 | 88 |
| 按需优势模式的特征和用例 | 89 |
| 预置模式的特征和用例 | 90 |

| | |
|---|-----|
| 切换模式 | 91 |
| 使用创建直播 Amazon Web Services 管理控制台 | 91 |
| 使用 APIs 创建流 | 92 |
| 构建 Kinesis Data Streams 客户端 | 92 |
| 创建流 | 93 |
| 更新流 | 94 |
| 使用控制台 | 94 |
| 使用 API | 95 |
| 使用 Amazon CLI | 96 |
| 列出流 | 96 |
| 列出分片 | 97 |
| 删除流 | 100 |
| 对流进行重新分片 | 100 |
| 确定重新分片策略 | 101 |
| 拆分分片 | 102 |
| 合并两个分片 | 103 |
| 完成重新分片操作 | 104 |
| 更改数据留存期 | 106 |
| 标记资源 | 107 |
| 查看标签基本知识 | 107 |
| 使用标签跟踪成本 | 108 |
| 了解标签限制 | 109 |
| 使用 Kinesis Data Streams 控制台标记流 | 109 |
| 使用标记直播 Amazon CLI | 111 |
| 使用 Kinesis Data Streams 标记直播 APIs | 111 |
| 使用标记消费者 Amazon CLI | 112 |
| 使用 Kinesis Data Streams 标记消费者 APIs | 112 |
| 处理大型记录 | 113 |
| 更新您的流以使用大型记录 | 113 |
| 使用大型记录优化流性能 | 114 |
| 使用大型记录缓解限制 | 114 |
| 使用 Kinesis Data Streams 处理大型记录 APIs | 114 |
| Amazon 与大型唱片兼容的组件 | 114 |
| 支持大型记录的区域 | 117 |
| 使用执行弹性测试 Amazon Fault Injection Service | 119 |
| 预置吞吐量错误 | 120 |

| | |
|---|-----|
| 迭代器过期异常错误 | 122 |
| 将数据写入 Kinesis Data Streams | 124 |
| 使用 Amazon Kinesis Producer Library (KPL) 开发产生器 | 124 |
| 查看 KPL 的作用 | 125 |
| 了解使用 KPL 的优势 | 126 |
| 了解何时不使用 KPL | 127 |
| 安装 KPL | 127 |
| 迁移至 KPL 1.x | 128 |
| 转换为适用于 KPL 的 Amazon Trust Services (ATS) 证书 | 132 |
| KPL 受支持平台 | 132 |
| KPL 的重要概念 | 132 |
| 将 KPL 与产生器代码集成 | 135 |
| 使用 KPL 写入 Kinesis Data Streams | 136 |
| 配置 KPL | 138 |
| 实现消费端取消聚合 | 139 |
| 将 KPL 与 Amazon Data Firehose 搭配使用 | 142 |
| 将 KPL 与 Amazon Glue 架构注册表搭配使用 | 142 |
| 配置 KPL 代理配置 | 142 |
| KPL 版本生命周期策略 | 143 |
| 使用 Kinesis Data Streams API 和 适用于 Java 的 Amazon SDK 开发产生器 | 144 |
| 向流添加数据 | 144 |
| 使用 Amazon Glue 架构注册表与数据交互 | 150 |
| 使用 Kinesis 代理写入 Amazon Kinesis Data Streams | 151 |
| 完成 Kinesis 代理的先决条件 | 151 |
| 下载并安装代理 | 152 |
| 配置并启动代理 | 153 |
| 指定代理配置设置 | 154 |
| 监控多个文件目录并写入多个流 | 156 |
| 使用代理预处理数据 | 157 |
| 使用代理 CLI 命令 | 161 |
| 常见问题 | 162 |
| 使用其他 Amazon 服务写入 Kinesis Data Streams | 163 |
| 使用 Amazon Amplify 写入 Kinesis Data Streams | 163 |
| 使用 Amazon Aurora 写入 Kinesis Data Streams | 164 |
| 使用 Amazon CloudFront 写入 Kinesis Data Streams | 164 |
| 使用 Amazon CloudWatch Logs 写入 Kinesis Data Streams | 164 |

| | |
|--|-----|
| 使用 Amazon Connect 写入 Kinesis Data Streams | 164 |
| 使用 Amazon Database Migration Service 写入 Kinesis Data Streams | 165 |
| 使用 Amazon DynamoDB 写入 Kinesis Data Streams | 165 |
| 使用 Amazon EventBridge 写入 Kinesis Data Streams | 165 |
| 使用 Amazon IoT Core 写入 Kinesis Data Streams | 165 |
| 使用 Amazon Relational Database Service 写入 Kinesis Data Streams | 165 |
| 使用 Amazon Pinpoint 写入 Kinesis Data Streams | 166 |
| 使用 Amazon Quantum Ledger Database (Amazon QLDB) 写入 Kinesis Data Streams | 166 |
| 使用第三方集成写入 Kinesis Data Streams | 166 |
| Apache Flink | 167 |
| Fluentd | 167 |
| Debezium | 167 |
| Oracle GoldenGate | 167 |
| Kafka Connect | 167 |
| Adobe Experience | 167 |
| Striim | 167 |
| Kinesis Data Streams 产生器问题排查 | 168 |
| 产生器应用程序的写入速率比预期的慢 | 168 |
| 我收到了未授权的 KMS 主密钥权限错误 | 170 |
| 排查产生器的其他常见问题 | 170 |
| 优化 Kinesis Data Streams 产生器 | 170 |
| 自定义 KPL 重试和速率限制行为 | 170 |
| 将最佳实践应用于 KPL 聚合 | 171 |
| 从 Kinesis Data Streams 读取数据 | 173 |
| 开发具有专用吞吐量的增强扇出型消费端 | 173 |
| 共享吞吐量消费端和增强扇出型消费端之间的区别 | 175 |
| 支持最多 50 个增强型扇出消费者的区域 (仅限按需优势) | 117 |
| 管理具有增强扇出功能的消费端 | 177 |
| 在 Kinesis 控制台中使用数据查看器 | 178 |
| 在 Kinesis 控制台中查询您的数据流 | 179 |
| 使用 Kinesis Client Library | 180 |
| 什么是 Kinesis Client Library ? | 180 |
| KCL 主要功能和优势 | 180 |
| KCL 概念 | 181 |
| KCL 中的 DynamoDB 元数据表和负载均衡 | 182 |
| 使用 KCL 开发消费端 | 185 |

| | |
|--|-----|
| 使用 KCL 进行多流处理 | 195 |
| 在 KC Amazon Glue L 中使用架构注册表 | 198 |
| KCL 消费端应用程序所必需的 IAM 权限 | 198 |
| KCL 配置 | 204 |
| KCL 版本生命周期策略 | 215 |
| 从之前的 KCL 版本迁移 | 215 |
| 先前的 KCL 版本文档 | 228 |
| 使用适用于 Java 的 Amazon SDK 开发消费端 | 301 |
| 使用适用于 Java 的 Amazon SDK 开发具有共享吞吐量的消费端 | 302 |
| 使用适用于 Java 的 Amazon SDK 开发具有增强扇出功能的消费端 | 307 |
| 使用 Amazon Glue 架构注册表与数据交互 | 309 |
| 使用 Amazon Lambda 开发消费端 | 310 |
| 使用适用于 Apache Flink 的托管服务来开发消费端 | 310 |
| 使用 Amazon Data Firehose 开发消费端 | 310 |
| 使用其他 Amazon 服务从 Kinesis Data Streams 读取数据 | 310 |
| 使用 Amazon EMR 从 Kinesis Data Streams 读取数据 | 311 |
| 使用 Amazon EventBridge Pipes 从 Kinesis Data Streams 读取数据 | 311 |
| 使用 Amazon Glue 从 Kinesis Data Streams 读取数据 | 311 |
| 使用 Amazon Redshift 从 Kinesis Data Streams 读取数据 | 311 |
| 使用第三方集成从 Kinesis Data Streams 中读取 | 312 |
| Apache Flink | 312 |
| Adobe Experience Platform | 312 |
| Apache Druid | 312 |
| Apache Spark | 313 |
| Databricks | 313 |
| Kafka Confluent Platform | 313 |
| Kinesumer | 313 |
| Talend | 313 |
| Kinesis Data Streams 消费端问题排查 | 313 |
| LeaseManagementConfig 构造函数编译错误 | 314 |
| 使用 Kinesis Client Library 时会跳过某些 Kinesis Data Streams 记录 | 315 |
| 属于同一分片的记录通过不同的记录处理器同时处理 | 315 |
| 消费端应用程序的读取速率比预期的慢 | 316 |
| 即使流中有数据，GetRecords 仍然返回空记录阵列 | 316 |
| 分片迭代器意外过期 | 317 |
| 消费端记录处理滞后 | 317 |

| | |
|--|-----|
| 未授权的 KMS 密钥权限错误 | 318 |
| DynamodBException : 在更新表达式中提供的文档路径对于更新无效 | 318 |
| 排查消费端的其他常见问题 | 318 |
| 优化 Kinesis Data Streams 消费端 | 319 |
| 改进低延迟处理 | 319 |
| 使用 Amazon Kinesis Amazon Lambda 制作器库处理序列化数据 | 320 |
| 使用重新分片、扩展和并行处理更改分片数量 | 320 |
| 处理重复记录 | 322 |
| 处理启动、关闭和节流 | 324 |
| 监控 Kinesis Data Streams | 326 |
| 使用 CloudWatch 监控 Kinesis Data Streams 服务 | 326 |
| Amazon Kinesis Data Streams 维度和指标 | 327 |
| 访问 Kinesis Data Streams 的 Amazon CloudWatch 指标 | 341 |
| 使用 CloudWatch 监控 Kinesis Data Streams 代理运行状况 | 342 |
| 使用 CloudWatch 进行监控 | 342 |
| 使用 Amazon CloudTrail 记录 Amazon Kinesis Data Streams API 调用 | 343 |
| CloudTrail 中的 Kinesis Data Streams 信息 | 344 |
| 示例 : Kinesis Data Firehose 日志文件条目 | 345 |
| 使用 CloudWatch 监控 KCL | 349 |
| 指标和命名空间 | 349 |
| 指标级别和维度 | 349 |
| 指标配置 | 350 |
| 指标的列表 | 351 |
| 使用 CloudWatch 监控 KPL | 366 |
| 指标、维度和命名空间 | 367 |
| 指标级别和粒度 | 367 |
| 本地访问和 Amazon CloudWatch 上传 | 368 |
| 指标的列表 | 368 |
| 安全性 | 372 |
| Kinesis Data Streams 中的数据保护 | 372 |
| 什么是 Kinesis Data Streams 的服务器端加密 ? | 373 |
| 费用、区域和性能注意事项 | 374 |
| 如何开始使用服务器端加密 ? | 375 |
| 创建和使用用户生成的 KMS 密钥 | 376 |
| 使用用户生成的 KMS 密钥的权限 | 377 |
| 验证 KMS 密钥权限并排查相关问题 | 379 |

| | |
|---|------|
| 将 Kinesis Data Streams 与接口 VPC 端点搭配使用 | 379 |
| 使用 IAM 控制对 Kinesis Data Streams 资源的访问 | 382 |
| 策略语法 | 383 |
| Kinesis Data Streams 的操作 | 384 |
| Kinesis Data Streams 的亚马逊资源名称 (ARNs) | 385 |
| Kinesis Data Streams 的示例策略 | 385 |
| 与其他账户共享您的数据流 | 388 |
| 将Amazon Lambda函数配置为使用另一个账户从 Kinesis Data Streams 读取 | 393 |
| 使用基于资源的策略共享访问权限 | 393 |
| Kinesis Data Streams 的合规性验证 | 395 |
| Kinesis Data Streams 中的韧性 | 396 |
| Kinesis Data Streams 中的灾难恢复 | 396 |
| Kinesis Data Streams 中的基础设施安全 | 397 |
| Kinesis Data Streams 的安全最佳实践 | 397 |
| 实施最低权限访问 | 398 |
| 使用 IAM 角色 | 398 |
| 实现从属资源中的服务器端加密 | 398 |
| CloudTrail 用于监控 API 调用 | 398 |
| 与 Amazon SDKs | 399 |
| 代码示例 | 400 |
| 基本功能 | 401 |
| 了解基本功能 | 401 |
| 操作 | 405 |
| 无服务器示例 | 462 |
| 通过 Kinesis 触发器调用 Lambda 函数 | 462 |
| 通过 Kinesis 触发器报告 Lambda 函数批处理项目失败 | 472 |
| 文档历史记录 | 487 |
| | cdxc |

什么是 Amazon Kinesis Data Streams？

可以使用 Amazon Kinesis Data Streams 实时收集和处理大型数据记录流。可以创建称为 Kinesis Data Streams 应用程序的数据处理应用程序。典型 Kinesis Data Streams 应用程序会将数据流中的数据作为数据记录读取。这些应用程序可使用 Kinesis Client Library，并且可在 Amazon EC2 实例上运行。可以将处理后的记录发送到控制面板，使用这些记录生成警报，动态更改定价和广告战略，或将数据发送给其他各个 Amazon 服务。有关 Kinesis Data Streams 功能和定价的信息，请参阅 [Amazon Kinesis Data Streams](#)。

Kinesis Data Streams 是 Kinesis 流数据平台的一部分，此外还有 [Firehose](#)、[Kinesis Video Streams](#) 和[适用于 Apache Flink 的托管服务](#)。

有关 Amazon 大数据解决方案的更多信息，请参阅 [Amazon 上的大数据](#)。有关 Amazon 流数据解决方案的更多信息，请参阅[什么是流数据？](#)。

主题

- [我能用 Kinesis Data Streams 做什么？](#)
- [使用 Kinesis Data Streams 的优点](#)
- [相关服务](#)

我能用 Kinesis Data Streams 做什么？

您可以使用 Kinesis Data Streams 实现快速而持续的数据引入和聚合。使用的数据类型可以包括 IT 基础设施日志数据、应用程序日志、社交媒体、市场数据源和 Web 点击流数据。由于数据引入和处理的响应时间是实时的，因此处理通常是轻量级的。

以下是使用 Kinesis Data Streams 的典型场景：

加速的日志和数据源引入和处理

您可以让创建者直接将数据推入流。例如，推送系统和应用程序日志，它们可在几秒内就绪，以用于处理。这可以防止因前端或应用程序服务器故障而造成日志数据丢失。Kinesis Data Streams 可加快数据源接收速度，因为在提交数据以进行引入之前，数据不会在服务器上进行批处理。

实时指标和报告

您可以使用收集到 Kinesis Data Streams 中的数据进行实时的简单数据分析和报告。例如，您的数据处理应用程序可以处理系统和应用程序日志的指标和报告，因为数据将流入而不是等待接收批量数据。

实时数据分析

这可将并行处理的强大功能与实时数据的价值相结合。例如，实时处理网站点击流，然后使用多个并行运行的不同的 Kinesis Data Streams 应用程序来分析站点可用性参与度。

复杂流处理

可以创建 Kinesis Data Streams 应用程序和数据流的有向无环图 (DAG)。这通常会涉及将数据从多个 Kinesis Data Streams 应用程序放入其他流，以供其他 Kinesis Data Streams 应用程序进行下游处理。

使用 Kinesis Data Streams 的优点

虽然可使用 Kinesis Data Streams 解决各种流数据问题，但其常见用途是实时聚合数据，然后将聚合数据加载到数据仓库或 map-reduce 集群。

将数据放入 Kinesis 数据流，以确保持久性和弹性。将记录放入流的时间与可检索记录的时间之间的延迟 (put-to-get 延迟) 通常不到 1 秒。换言之，在添加数据之后，Kinesis Data Streams 应用程序几乎立即可以开始使用流中的数据。Kinesis Data Streams 的托管服务方面可减轻您创建和运行数据引入管道的操作负担。可以创建流式 map-reduce 类型应用程序。利用 Kinesis Data Streams 的弹性，可以扩大或缩小流，以确保数据记录绝不会在过期前丢失。

多个 Kinesis Data Streams 应用程序可以使用流中的数据，以便多个操作（如归档和处理）可以并发且独立地进行。例如，两个应用程序可读取同一流中的数据。第一个应用程序计算正在运行的聚合并更新 Amazon DynamoDB 表，第二个应用程序压缩数据并将数据归档至数据存储，例如 Amazon Simple Storage Service (Amazon S3)。然后，控制面板将读取包含正在运行的聚合的 DynamoDB 表，以获取实时报告（分钟级）。

Kinesis Client Library 支持容错使用流中的数据，并提供针对 Kinesis Data Streams 应用程序的扩展支持。

相关服务

有关使用 Amazon EMR 集群直接读取和处理 Kinesis Data Streams 的信息，请参阅 [Kinesis Connector](#)。

Amazon Kinesis Data Streams 术语和概念

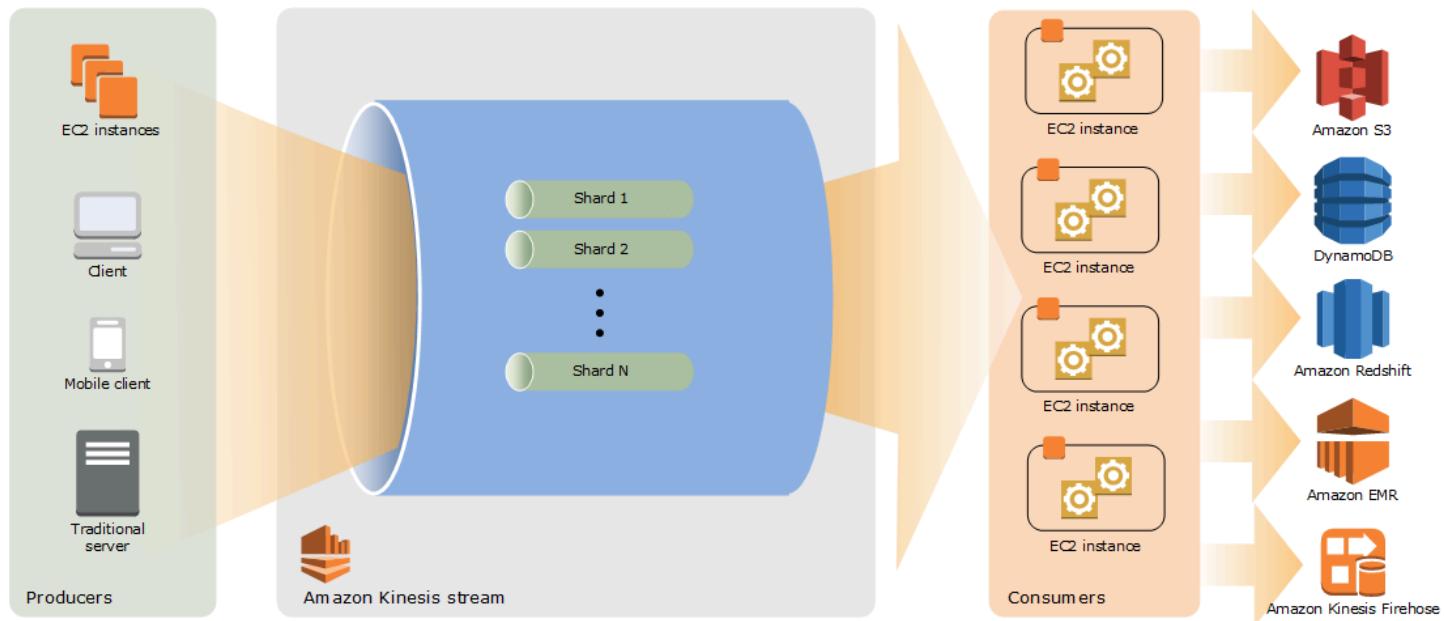
开始使用 Amazon Kinesis Data Streams 之前，有必要了解其架构和术语。

主题

- [查看 Kinesis Data Streams 的高级架构](#)
- [熟悉 Kinesis Data Streams 的术语](#)

查看 Kinesis Data Streams 的高级架构

下图展示了 Kinesis Data Streams 的大致架构。创建器会持续将数据推送到 Kinesis Data Streams，消费端会实时处理数据。消费端（例如在 Amazon EC2 上运行的自定义应用程序或 Amazon Data Firehose 传输流）可以使用 Amazon DynamoDB、Amazon Redshift 或 Amazon S3 等 Amazon 服务存储结果。



熟悉 Kinesis Data Streams 的术语

Kinesis 数据流

Kinesis 数据流是一组分片。每个分片都有一系列数据记录。每个数据记录都有一个由 Kinesis Data Streams 分配的序列号。

数据记录

数据记录是存储在 [Kinesis 数据流](#) 中的数据单位。数据记录由 [序列号](#)、[分区键](#) 和数据 Blob (不可变的字节序列) 组成。Kinesis Data Streams 不会以任何方式检查、解释或更改 Blob 中的数据。数据 Blob 最大可为 1MB。

容量模式

数据流容量模式决定如何管理容量以及如何对数据流的使用收费。目前，在 Kinesis Data Streams 中，您可以为数据流选择按需模式和预置模式。有关更多信息，请参阅 [选择正确的流式传入的模式](#)。

若采用按需模式，Kinesis Data Streams 会自动管理分片来提供必要的吞吐量。您只需为实际使用的吞吐量付费，Kinesis Data Streams 会在工作负载增加或减少时自动适应吞吐量需求。有关更多信息，请参阅 [按需标准模式的特征和用例](#)。

若采用预置模式，您必须为数据流指定分片数。数据流的总容量是其分片容量的总和。您可以根据需要增加或减少数据流中的分片数，并且按小时费率支付分片数的费用。有关更多信息，请参阅 [预置模式的特征和用例](#)。

保留周期

保留期 是数据记录在添加到流中后可供访问的时间长度。在创建之后，流的保留期设置为默认值 24 小时。您可以使用 [IncreaseStreamRetentionPeriod](#) 操作将保留期增加到最高 8760 小时 (365 天)，使用 [DecreaseStreamRetentionPeriod](#) 操作将保留期减少到最短 24 小时。对于保留期设置为 24 小时以上的流，将收取额外费用。有关更多信息，请参阅 [Amazon Kinesis Data Streams 定价](#)。

Producer

创建器会将记录存入 Amazon Kinesis Data Streams。例如，发送日志数据到流的 Web 服务器是创建器。

消费端

消费端会从 Amazon Kinesis Data Streams 获取记录并加以处理。这些消费端称为 [Amazon Kinesis Data Streams 应用程序](#)。

Amazon Kinesis Data Streams 应用程序

Amazon Kinesis Data Streams 应用程序是通常在 EC2 实例集上运行的流的消费端。

可以开发的消费端有两种：共享扇出功能消费端和增强型扇出功能消费端。要了解它们之间的区别，以及了解如何创建每种消费端，请参阅[从 Amazon Kinesis Data Streams 读取数据](#)。

Kinesis Data Streams 应用程序的输出可能是另一个流的输入，这使您能够创建实时处理数据的复杂拓扑。应用程序也可将数据发送到各种其他 Amazon 服务。一个流可以有多个应用程序，每个应用程序可同时单独使用流中的数据。

分片

分片 是流中数据记录的唯一标识序列。一个流由一个或多个分片组成，每个分片提供一个固定的容量单位。每个分片最多可以支持每秒 5 个事务的读取，最大总数据读取速率为每秒 2MB；每个分片最多可以支持每秒 1000 条记录的写入，最大总数据写入速率为每秒 1MB（包括分区键）。流的数据容量是您为流指定的分片数的函数。流的总容量是其分片容量的总和。

如果数据速率增加，您可以增加或减少分配给流的分片数量。有关更多信息，请参阅[对流进行重新分片](#)。

分区键

分区键用于在流中按分片对数据进行分组。Kinesis Data Streams 将属于一个流的数据记录分割到多个分片中。它使用与每个数据记录关联的分区键确定指定数据记录属于哪个分片。分区键是 Unicode 字符串，每个键的最大长度限制为 256 个字符。MD5 哈希函数用于将分区键映射到 128 位整数值，并使用分片的哈希键范围将关联的数据记录映射到分片。当应用程序将数据放入流中时，它必须指定一个分区键。

序列号

每条数据记录都有一个序列号，该序列号对其分片内的每个分区键都是唯一的。在您使用 `client.putRecords` 或 `client.putRecord` 写入流之后，Kinesis Data Streams 将分配序列号。同一分区键的序列号通常会随时间推移增加。写入请求之间的时间段越长，序列号越大。

Note

序列号不能用作相同流中的数据集的索引。为了在逻辑上分隔数据集，请使用分区键或者为每个数据集创建单独的流。

Kinesis Client Library

Kinesis Client Library 将编译成应用程序，从而支持以容错方式使用流中的数据。Kinesis Client Library 确保每个分片有一个用于运行和处理它的记录处理器。库还可以简化流中的数据读取。Kinesis Client Library 使用 Amazon DynamoDB 表来存储与数据使用相关的元数据。它会为每个正在处理数据的应用程序创建三个表。有关更多信息，请参阅 [使用 Kinesis Client Library](#)。

应用程序名称

Amazon Kinesis Data Streams 应用程序的名称用于标识应用程序。每个应用程序必须具有一个唯一名称，此名称的范围限定于应用程序使用的 Amazon 账户和区域。此名称用作 Amazon DynamoDB 中的控制表名称和 Amazon CloudWatch 指标的命名空间。

服务器端加密

当创建器将敏感数据输入流时，Amazon Kinesis Data Streams 可以自动加密这些数据。Kinesis Data Streams 使用 [Amazon KMS](#) 主密钥进行加密。有关更多信息，请参阅 [Amazon Kinesis Data Streams 中的数据保护](#)。

Note

要读取或写入加密的流、创建者和消费端应用程序必须有权访问主密钥。有关向创建者应用程序和消费端应用程序授予权限的信息，请参阅[the section called “使用用户生成的 KMS 密钥的权限”](#)。

Note

使用服务器端加密将产生 Amazon Key Management Service (Amazon KMS) 费用。有关更多信息，请参阅 [Amazon Key Management Service Pricing](#)。

限额和限制

下表显示了 Amazon Kinesis Data Streams 的流和分片限额及限制。

| 配额 | 按需模式 | 预置模式 |
|------|---|--|
| 数据流数 | 您的 Amazon 账户内的直播数量没有上限。默认情况下，您可以使用按需容量模式创建最多 50 个数据流。如需增加此限额，请提交 支持工单 。 | 账户中采用预置模式的流数量没有限额上限。 |
| 分片数量 | 并无上限。分片数量取决于摄取的数据量和所需的吞吐量级别。Kinesis Data Streams 会根据数据量和流量的变化自动扩展分片数量。 | <p>并无上限。以下各项的默认分片配额为每个 20,000 Amazon Web Services 账户 个分片：</p> <p>Amazon Web Services 区域</p> <ul style="list-style-type: none">美国东部 (弗吉尼亚北部)美国西部 (俄勒冈)欧洲地区 (爱尔兰) <p>其余所有区域每个 Amazon Web Services 账户账户的默认分片限额为 1000 或 6000 个分片。您可以通过 Service Quotas 控制台查看账户的分片配额和使用率，网址为https://console.aws.amazon.com/servicequotas/。</p> <p>要申请增加分片配额，请使用 Service Quotas 控制台或 Amazon CLI。有关更多信息，请参阅请求提高限额。</p> |

| 配额 | 按需模式 | 预置模式 |
|-----------------|--|---|
| 数据流吞吐量 | <p>默认情况下，使用按需容量模式创建的新数据流的写入吞吐量为 4 MB/s，读取吞吐 MB/s 量为 8。在美国东部（弗吉尼亚北部）、美国西部（俄勒冈）和欧洲（爱尔兰）Amazon Web Services 区域，按需容量模式的数据流最多可扩展到 10% GB/s 的写入吞吐量和 20% 的 GB/s 读取吞吐量。对于其他区域，按需容量模式的数据流最多可扩展到 200 MB/s 的写入吞吐量和 400 的 MB/s 读取吞吐量。如果您需要将这些区域的 GB/s 写入容量增加到 10 和 20 个 GB/s 读取容量，请提交支持请求。</p> | <p>并无上限。最大吞吐量取决于为流预置的分片数量。每个分片最多可支持 1 MB/sec 或 1,000 个 records/sec 写入吞吐量或最多 2 MB/sec 或 2,000 个 records/sec 读取吞吐量。如果您需要更多载入容量，则可以使用 Amazon Web Services 管理控制台 或 API 轻松扩大流中的分片数量。UpdateShardCount</p> |
| 数据有效负载大小 | <p>使用 base64-encoding 之前记录的数据有效负载的最大大小为 10 MiB。Kinesis 旨在使用突增容量处理间歇性的大型记录（大小为 1-10MiB）。</p> | |
| GetRecords 事务大小 | <p>GetRecords每次调用可以从单个分片中检索多达 10 MB 的数据，每次调用最多可检索 10,000 条记录。对 GetRecords 的每次调用都算作一个读取事务。每个分片每秒可支持多达 5 个读取事务。每个读取事务可提供多达 10000 个记录，每个事务的配额上限为 10 MB。</p> | |
| 每个分片的数据读取速率 | <p>每个分片最多可支持每秒 2 MB 的最大总数据读取速率。GetRecords如果对 GetRecords 的调用返回 10 MB，则在接下来的 5 秒内发出的后续调用将会引发异常。</p> | |

| 配额 | 按需模式 | 预置模式 |
|----------------|---|------|
| 每个数据流的注册消费端数量 | 使用 Kinesis 按需优势模式，您最多可以创建 50 个注册用户（增强型扇出）。使用 Kinesis 按需标准模式和 Kinesis 预配置模式，您可以为每个数据流创建最多 20 个注册使用者（增强扇出限制）。 | |
| 在预置模式和按需模式之间切换 | 对于 Amazon 账户中的每个数据流，您可以在 24 小时内两次在按需和预配置容量模式之间切换。 | |

API 限制

与大多数操作一样 Amazon APIs，Kinesis Data Streams API 操作受到速率限制。下面是每个 Amazon 账户在每个区域的限制。有关 Kinesis Data APIs Streams 的更多信息，请参阅[亚马逊 Kinesis API 参考](#)。

KDS 控制层面 API 限制

下一节介绍了 KDS 控制平面的 APIs 限制。KDS 控制平面 APIs 允许您创建和管理数据流。这些是每个 Amazon 账户在每个区域的限制。

控制层面 API 限制

| API | API 调用限制 | 每个账户/流 | 描述 |
|-----------------|----------------|--------|--|
| AddTagsToStream | 每秒 5 个事务 (TPS) | 每个账户 | 每个数据流 50 个标签 |
| CreateStream | 5 TPS | 每个账户 | 在账户中可以拥有的流数量没有配额上限。当您尝试执行以下操作之一时，如果发出 CreateStream 请求，则会收到 LimitExceededException： |

| API | API 调用限制 | 每个账户/流 | 描述 |
|-------------------------------|----------|--------|--|
| | | | <ul style="list-style-type: none"> 在任何时间点，CREATING 状态中有五个以上的流。 创建的分片数量超过了为您的账户授权的数量。 |
| DecreaseStreamRetentionPeriod | 5 TPS | 每个流 | 数据流的保留期的最小值为 24 小时。 |
| DeleteResourcePolicy | 5 TPS | 每个账户 | 如需增加此限额，请提交 支持工单 。 |
| DeleteStream | 5 TPS | 每个账户 | |
| DeregisterStreamConsumer | 5 TPS | 每个流 | |
| DescribeAccountSettings | 5 TPS | 每个账户 | |
| DescribeLimits | 1 TPS | 每个账户 | |
| DescribeStream | 10 TPS | 每个账户 | |
| DescribeStreamConsumer | 20 TPS | 每个流 | |
| DescribeStreamSummary | 20 TPS | 每个账户 | |
| DisableEnhancedMonitoring | 5 TPS | 每个流 | |

| API | API 调用限制 | 每个账户/流 | 描述 |
|-------------------------------|----------|--------|---|
| EnableEnhancedMonitoring | 5 TPS | 每个流 | |
| GetResourcePolicy | 5 TPS | 每个账户 | 如需增加此限额，请提交 支持工单 。 |
| IncreaseStreamRetentionPeriod | 5 TPS | 每个流 | 流保留期的最大值为 8760 小时 (365 天)。 |
| ListShards | 1000 TPS | 每个流 | |
| ListStreamConsumers | 5 TPS | 每个流 | |
| ListStreams | 5 TPS | 每个账户 | |
| ListTagsForStream | 5 TPS | 每个流 | |
| MergeShards | 5 TPS | 每个流 | 仅适用于预置模式。 |
| PutResourcePolicy | 5 TPS | 每个账户 | 如需增加此限额，请提交 支持工单 。 |
| RegisterStreamConsumer | 5 TPS | 每个流 | 您最多可以为每个数据流注册 20 个消费端。每次只能在一个数据流中注册给定的消费端。只能同时创建 5 个消费端。换言之，同时处于 CREATING 状态的消费端不能超过 5 个。 |
| RemoveTagsFromStream | 5 TPS | 每个流 | |
| SplitShard | 5 TPS | 每个流 | 仅适用于预置模式 |

| API | API 调用限制 | 每个账户/流 | 描述 |
|----------------------------|----------|--------|---|
| StartStreamEncryption | | 每个流 | 您可以在 24 小时内成功应用新的 Amazon KMS 密钥进行服务器端加密 25 次。 |
| StopStreamEncryption | | 每个流 | 在连续 24 小时期间，您可以成功禁用服务器端加密 25 次。 |
| UpdateShardCount | | 每个流 | 仅适用于预置模式。分片数量的默认限制为 10000 个。此 API 还存在其他限制。有关更多信息，请参阅 UpdateShardCount 。 |
| UpdateStreamMode | | 每个流 | 对于 Amazon 账户中的每个数据流，您可以在 24 小时内两次在按需和预配置容量模式之间切换。 |
| UpdateStreamWarmThroughput | 5 TPS | 每个 账户 | 相应账户和区域可配置的最大热吞吐量为按需模式的数据流吞吐量限制。 |
| UpdateAccountSettings | 5 TPS | 每个 账户 | 启用或禁用按需优势模式等账户设置。 |

数据层面 API 限制

下一节介绍了 KDS 数据平面的 APIs 限制。KDS 数据平面 APIs 允许您使用数据流实时收集和处理数据记录。下面是您的数据流中的每分片限制。

数据层面 API 限制

| API | API 调用限制 | 负载限制 | 其他详细信息 |
|------------------|----------|--|--|
| GetRecords | 5 TPS | 每次调用可返回的最大记录数为 10000 个。GetRecords 可返回的数据的最大大小为 10MB。 | 如果某个调用返回此数据量，则在接下来的 5 秒内执行后续调用时，会引发 ProvisionedThroughputExceededException。如果流上的预置吞吐量不足，则在接下来的 1 秒内进行后续调用时，会引发 ProvisionedThroughputExceededException。 |
| GetShardIterator | 5 TPS | | 分片迭代器在其返回给请求者的 5 分钟后过期。如果 GetShardIterator 请求过于频繁，您会收到 ProvisionedThroughputExceededException。 |
| PutRecord | 1000 TPS | 每个分片最多可以支持每秒写入 1000 条记录，最大数据写入总数为每秒 10 MiB。 | Kinesis 旨在使用突增容量处理间歇性的大型记录（大小为 1-10MiB）。 |
| PutRecords | | 每个 PutRecords 请求最多可支持 500 条记录。请求中的每一个记录最大可以为 1 | Kinesis 旨在使用突增容量处理间歇性的大型记录（大小为 1-10MiB）。 |

| API | API 调用限制 | 负载限制 | 其他详细信息 |
|------------------|--|---|---|
| | | MiB，整个请求的上限为 10 MiB，包括分区键。每个分片最多可以支持每秒写入 1000 条记录，最大数据写入总数为每秒 1 MB。 | |
| SubscribeToShard | 每个分片的每个注册用户 SubscribeToShard 每秒可以拨打一个电话。 | | 如果您 SubscribeToShard 再次使用相同的 consumerArn 拨打电话，ShardId 并且在成功通话后的 5 秒钟内，您将获得 ResourceInUseException |

提升配额

如果配额是可调整的，则可以使用服务配额来请求增加配额。有些请求会自动得到解决，而另一些则提交给 Support Amazon。您可以跟踪提交给 Support 的配额增加请求的 Amazon 状态。提高服务配额的请求没有得到优先支持。如果您有紧急请求，请联系 Support Amazon。有关更多信息，请参阅 [什么是服务配额？](#)。

要请求增加服务配额，请按照[请求增加配额](#)中概述的步骤操作。

完成设置 Amazon Kinesis Data Streams 的先决条件

首次使用 Amazon Kinesis Data Streams 之前，请完成以下任务以设置环境。

任务

- [注册 Amazon](#)
- [下载库和工具](#)
- [配置您的开发环境](#)

注册 Amazon

当您注册亚马逊云科技 (Amazon) 时，您的 Amazon 账户会自动注册 Amazon 中的所有服务，包括 Kinesis Data Streams。您只需为使用的服务付费。

如果您已有 Amazon 账户，请跳到下一个任务。如果您还没有 Amazon 账户，请按照以下步骤创建。

如何注册Amazon账户

1. 打开 <https://portal.aws.amazon.com/billing/signup>。
2. 按照屏幕上的说明操作。

在注册时，将接到电话或收到短信，要求使用电话键盘输入一个验证码。

当您注册 Amazon Web Services 账户时，系统将会创建一个 Amazon Web Services 账户根用户。根用户有权访问该账户中的所有 Amazon Web Services 服务和资源。作为最佳安全实践，请为用户分配管理访问权限，并且只使用根用户来执行[需要根用户访问权限的任务](#)。

下载库和工具

以下库和工具可帮助您使用 Kinesis Data Streams：

- [Amazon Kinesis API Reference](#) 是 Kinesis Data Streams 支持的基本操作集。有关使用 Java 代码执行基本操作的更多信息，请参阅以下主题：
 - [使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon SDK 开发产生器](#)
 - [使用适用于 Java 的 Amazon SDK 开发消费端](#)
 - [创建和管理 Kinesis 数据流](#)

- 适用于 [Go](#)、[Java](#)、[JavaScript](#)、[.NET](#)、[PHP](#)、[Python](#) 和 [Ruby](#) 的 Amazon 开发工具包，包括 Kinesis Data Streams 支持和示例。如果 适用于 Java 的 Amazon SDK 版本不包含 Kinesis Data Streams 示例，您也可以从 [GitHub](#) 进行下载。
- Kinesis Client Library (KCL) 提供了一个易于使用的编程模型来处理数据。KCL 可帮助您在 Java、Node.js、.NET、Python 和 Ruby 中快速上手使用 Kinesis Data Streams。有关更多信息，请参阅[读取流中的数据](#)。
- [Amazon Command Line Interface](#) 支持 Kinesis Data Streams。Amazon CLI 让您可以从命令行管理多个 Amazon 服务，并通过脚本自动执行这些服务。

配置您的开发环境

要使用 KCL，请确保您的 Java 开发环境符合以下要求：

- Java 1.7 (Java SE 7 JDK) 或更高版本。您可以从 Oracle 网站上的 [Java SE 下载](#) 页面下载最新的 Java 软件。
- Apache Commons 程序包（代码、HTTP 客户端和日志记录）
- Jackson JSON 处理器

请注意，适用于 Java 的 Amazon SDK 将 Apache Commons 和 Jackson 包含在第三方文件夹中。不过，适用于 Java 的开发工具包可与 Java 1.6 搭配使用，Kinesis Client Library 则需要 Java 1.7。

使用 Amazon CLI 执行 Amazon Kinesis Data Streams 操作

本节向您展示如何使用 Amazon Command Line Interface 执行基本的 Amazon Kinesis Data Streams 操作。您将了解 Kinesis Data Streams 基本的数据流原则，以及在 Kinesis 数据流中存入和取用数据的必要步骤。

如果是首次使用 Kinesis Data Streams，请先熟悉 [Amazon Kinesis Data Streams 术语和概念](#) 中介绍的概念和术语。

主题

- [教程：为 Kinesis Data Streams 安装和配置 Amazon CLI](#)
- [教程：使用 Amazon CLI 执行基本 Kinesis Data Streams 操作](#)

要进行 CLI 访问，您需要访问密钥 ID 和秘密访问密钥。如果可能，请使用临时凭证代替长期访问密钥。临时凭证包括访问密钥 ID、秘密访问密钥，以及一个指示凭证何时到期的安全令牌。有关更多信息，请参阅《IAM 用户指南》中的[将临时凭证用于 Amazon 资源](#)。

您可以在[创建 IAM 用户](#)中找到 IAM 和安全密钥设置的详细分步说明。

在本节中，讨论到的特定命令将一字不差地提供，但其特定值在每次运行时必然不同。此外，示例使用的是美国西部（俄勒冈州）区域，但本节中的步骤在任何[支持 Kinesis Data Streams 的区域](#)中都有效。

教程：为 Kinesis Data Streams 安装和配置 Amazon CLI

安装 Amazon CLI

有关如何安装适用于 Windows、Linux、OS X 和 Unix 操作系统的 Amazon CLI 的详细步骤，请参阅[安装 Amazon CLI](#)。

使用以下命令列出可用的选项和服务：

```
aws help
```

您将要使用 Kinesis Data Streams 服务，因此您可以使用以下命令审查与 Kinesis Data Streams 相关的 Amazon CLI 子命令：

```
aws kinesis help
```

此命令将生成包含可用 Kinesis Data Streams 命令的输出：

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

此命令列表对应着 [Amazon Kinesis Service API Reference](#) 中的 Kinesis Data Streams API。例如，`create-stream` 命令与 `CreateStream` API 操作对应。

Amazon CLI 现已成功安装，但未配置。这将在下一节展示。

配置 Amazon CLI

对于一般用途，`aws configure` 命令是设置 Amazon CLI 安装的最快方法。有关更多信息，请参阅[配置 Amazon CLI](#)。

教程 : 使用 Amazon CLI 执行基本 Kinesis Data Streams 操作

本节介绍如何通过 Amazon CLI 从命令行对 Kinesis 数据流执行基本操作。确保您熟悉[Amazon Kinesis Data Streams 术语和概念](#)中讨论的概念。

 Note

在创建流后，将象征性地向您的账户收取 Kinesis Data Streams 使用费，因为 Kinesis Data Streams 没有获得 Amazon Free Tier 的资格。当您完成本教程时，请删除 Amazon 资源以停止产生费用。有关更多信息，请参阅 [步骤 4：清除](#)。

主题

- 第 1 步：创建流
- 步骤 2：放置记录
- 步骤 3：获取记录
- 步骤 4：清除

第1步：创建流

您的第一步是创建一个流并验证它是否已创建成功。使用以下命令创建一个名为“Foo”的流：

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

接下来，发出以下命令以检查流的创建进度：

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

您应获得类似于以下示例的输出：

```
"StreamCreationTimestamp": 1572297168.0,  
"EnhancedMonitoring": [  
    {  
        "ShardLevelMetrics": []  
    }  
],  
"EncryptionType": "NONE",  
"OpenShardCount": 3,  
"ConsumerCount": 0  
}  
}
```

在此示例中，流的状态为 CREATING，这表示它还未完全做好使用准备。在几分钟后再次检查，您应看到类似于以下示例的输出：

```
{  
    "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  
    }  
}
```

此输出包含您在本教程中无需关注的信息。目前，您需要重点关注的是 "StreamStatus"： "ACTIVE"（告知您流已做好使用准备）和有关您请求的单个分片的信息。您还可以通过使用 `list-streams` 命令验证您的新流是否存在，如下所示：

```
aws kinesis list-streams
```

输出：

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

步骤 2：放置记录

既然您已经拥有活动的流，您便已做好放置一些数据的准备。在本教程中，您将使用最简单的命令 `put-record`，该命令会将一个包含文本“testdata”的数据记录放入流中：

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

如果成功，此命令将生成类似于以下示例的输出：

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

恭喜，您刚刚已将数据添加到流！接下来您将了解如何从流中获取数据。

步骤 3：获取记录

GetShardIterator

您必须先为您感兴趣的分片获取分片迭代器，然后才能从流中获取数据。分片迭代器表示消费端（在本例中为 `get-record` 命令）要从中读取数据的流和分片的位置。您将使用 `get-shard-iterator` 命令，如下所示：

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

请记住，`aws kinesis` 命令后面有一个 Kinesis Data Streams API，因此如果您对显示的任何参数感兴趣，都可以在 [GetShardIterator](#) API 参考主题中加以了解。成功执行后，此命令将生成类似于以下示例的输出：

```
{
```

```
"ShardIterator": "AAAAAAAHSyw1jv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG41NR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWR060TZRKnW9gd+efGN2aHFdkH1rJ14BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="}
```

看起来像随机字符的长字符串就是分片迭代器（您的字符串将与此不同）。您必须将分片迭代器复制/粘贴到接下来显示的 get 命令中。分片迭代器的有效生命周期为 300 秒，应该足以让您将分片迭代器复制/粘贴到下一个命令中。在将分片迭代器粘贴到写一个命令之前，您必须从中删除所有换行符。如果您收到分片迭代器不再有效的错误消息，请再次运行 get-shard-iterator 命令。

GetRecords

get-records 命令从流中获取数据，然后解析为对 Kinesis Data Streams API 中的 [GetRecords](#) 的调用。分片迭代器指定了分片中的一个位置，您希望从该位置开始按顺序读取数据记录。如果迭代器指向的分片中的部分没有可用的记录，GetRecords 将返回空白列表。可能需要进行多次调用才能到达分片中包含记录的部分。

在 get-records 命令的以下示例中：

```
aws kinesis get-records --shard-iterator  
AAAAAAAHSyw1jv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG41NR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWR060TZRKnW9gd+efGN2aHFdkH1rJ14BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

如果您是从 Unix 类型的命令处理器（如 bash）运行本教程，则可以使用嵌套命令自动执行分片迭代器的获取，如下所示：

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

如果您从支持 PowerShell 的系统运行此教程，则可以使用如下所示的命令自动获取分片迭代器：

```
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 命令的成功结果将从您在获取分片迭代器时指定的分片的流中请求记录，如以下示例所示：

```
{  
  "Records": [  
    {  
      "Data": "dGVzdGRhdGE=",  
      "PartitionKey": "123",  
      "ApproximateArrivalTimestamp": 1.441215410867E9,  
      "SequenceNumber": "49544985256907370027570885864065577703022652638596431874"  
    } ],  
    "MillisBehindLatest": 24000,  
  
    "NextShardIterator": "AAAAAAAAED0W3ugseWPE4503kqN1yN1UaodY8unE0sYs1MUmC61X9hlig5+t4RtZM0/  
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr  
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvP0ZvrUIudb8UkH3V  
}  
}
```

请注意，`get-records` 在上面被描述为请求，这意味着即使您的流中有记录，您可能也会收到零个或零个以上的记录。返回的任何记录都不能表示流中当前的所有记录。这是正常的，生产代码将以适当的时间间隔轮询流中的记录。此轮询速度将因您的特定应用程序设计要求而异。

在本教程的这部分中，您会注意到记录中的数据似乎是垃圾，不是我们发送的明文 `testdata`。这归因于 `put-record` 使用 Base64 编码支持您发送二进制数据的方式。但是，Amazon CLI 中的 Kinesis Data Streams 支持未提供 Base64 解码，因为对输出到 `stdout` 的原始二进制内容的 Base64 解码在某些平台和终端上可能会导致非预期的行为和潜在的安全问题。如果您使用 Base64 解码程序（例如，<https://www.base64decode.org/>）对 `dGVzdGRhdGE=` 进行手动解码，您将看到它实际上是 `testdata`。这对本教程来说已足够，在实践中，Amazon CLI 很少用于使用数据。更多时候，它用于监控流的状态并获取信息，如上方所示（`describe-stream` 和 `list-streams`）。有关 KCL 的更多信息，请参阅[使用 KCL 开发具有共享吞吐量的自定义消费端](#)。

`get-records` 并非总是会返回在流/分片中指定的所有记录。当出现这种情况时，请使用最后一个结果中的 `NextShardIterator` 获取下一组记录。如果更多数据会被放入流中（这是生产应用程序中的一般情况），您每次都可以使用 `get-records` 持续轮询数据。但是，如果您在 300 秒的分片迭代器生命周期内未使用下一个分片迭代器调用 `get-records`，则会收到一条错误消息，并且必须使用 `get-shard-iterator` 命令来获取新的分片迭代器。

此输出中还提供了 `MillisBehindLatest`，这是从流的末端响应 [GetRecords](#) 操作的毫秒数，指示消费端落后当前时间多远。零值指示正进行记录处理，此时没有新的记录要处理。在本教程中，如果您一边阅读教程一边操作，则可能会看到这个数值非常大。数据记录默认会在流中保留 24 小时，待您检索。此时间范围称为保留期，可以配置为最多 365 天。

成功执行 `get-records` 的结果总会返回一个 `NextShardIterator`，即使目前流中没有更多记录。这是一个假定创建器在任何给定时间内正在将更多记录放入流中的轮询模型。虽然您可编写自己的轮询例程，但如果您使用之前提到的 KCL 开发消费端应用程序，则系统将会为您执行此轮询。

如果您调用 `get-records`，直到您正在提取的流和分片中没有更多记录，您将看到带有空白记录的输出，类似于以下示例：

```
{  
  "Records": [],  
  "NextShardIterator": "AAAAAAAAAGCJ5jzQNjmdh06B/YDIDE56jmZmrMA/r1WjoHXC/  
kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wu1/  
EhyNeSs5DYXLSSC5XCapmCAYGFjYER69QSdQjxMmBPE/hiybFDi5qtkT6/PsZNz6kFoqtDk="  
}
```

步骤 4：清除

请删除流以释放资源并避免账户产生意外费用。每当您创建了不会使用的流时，请执行此操作，因为费用是按流量计算的，无论您是否使用流放置和获取数据，都会产生费用。清除命令如下所示：

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

成功运行命令会导致没有输出。使用 `describe-stream` 检查删除进度：

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

如果您在执行删除命令后立即执行此命令，您会看到类似于以下示例的输出：

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

在流完全删除后，`describe-stream` 将生成“未找到”错误：

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

Amazon Kinesis Data Streams 入门教程

Amazon Kinesis Data Streams 可提供多种不同解决方案，以便从 Kinesis 数据流中摄取和使用数据。本节中的教程旨在进一步帮助您了解 Amazon Kinesis Data Streams 的概念和功能，并确定能满足您的需求的解决方案。

主题

- [教程：使用 KPL 和 KCL 2.x 处理实时股票数据](#)
- [教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#)
- [教程：使用适用于 Apache Flink 的亚马逊托管服务分析实时股票数据](#)
- [教程：将 Amazon Lambda 与 Amazon Kinesis Data Streams 搭配使用](#)
- [使用适用于 Amazon Kinesis 的Amazon流数据解决方案](#)

教程：使用 KPL 和 KCL 2.x 处理实时股票数据

本教程的场景涉及将股票交易引入数据流中并编写对流执行计算的基础 Amazon Kinesis Data Streams 应用程序。您将了解如何将记录流发送到 Kinesis Data Streams 并实现近乎实时地使用和处理记录的应用程序。

Important

创建直播后，您的账户会因使用 Kinesis Data Streams 而产生象征性的费用，因为 Kinesis Data Streams 不符合Amazon免费套餐的资格。在消费端应用程序启动后，也会象征性收取 Amazon DynamoDB 使用费用。消费端应用程序使用 DynamoDB 跟踪处理状态。在使用完此应用程序后，请删除 Amazon 资源以停止产生费用。有关更多信息，请参阅 [清理 资源](#)。

代码不访问实际股票市场数据，而是模拟股票交易流。它通过使用随机股票交易生成器（将截至 2015 年 2 月市值排名前 25 位的股票的实际市场数据作为起始点）来执行此操作。如果您有权访问实时的股票交易流，则可能有兴趣从该流派生有用且及时的统计数据。例如，您可能希望执行滑动窗口分析，从而确定前 5 分钟内购买的最热门股票。或者，您可能希望在销售订单过大（即具有过多股份）时收到通知。可以扩展此系列代码以提供此类功能。

您可以在台式计算机或笔记本电脑上演练本教程中的步骤，然后在同一台计算机或支持已定义要求的任何平台上同时运行创建器和消费端代码。

显示的示例使用的是美国西部（俄勒冈州）区域，但它们适用于[支持 Kinesis Data Streams 的任何 Amazon 区域](#)。

任务

- [满足先决条件](#)
- [创建数据流](#)
- [创建 IAM 策略和用户](#)
- [下载并构建代码](#)
- [实现产生器](#)
- [实现消费端](#)
- [（可选）扩展消费端](#)
- [清理 资源](#)

满足先决条件

您必须满足以下要求才能完成本教程：

创建和使用亚马逊云科技账户

在开始之前，请确保熟悉[Amazon Kinesis Data Streams 术语和概念](#) 中讨论的概念，特别是流、分片、产生器和消费端。完成以下指南中的步骤也很有帮助：[教程：为 Kinesis Data Streams 安装和配置 Amazon CLI](#)。

您必须拥有一个Amazon帐户和一个网络浏览器才能访问Amazon Web Services 管理控制台。

要访问控制台，请使用您的 IAM 用户名和密码从 IAM 登录页面登录[Amazon Web Services 管理控制台](#)。有关Amazon安全证书的信息，包括编程访问权限和长期证书的替代方案，请参阅 IAM 用户指南中的[Amazon安全证书](#)。有关登录您的的详细信息Amazon Web Services 账户，请参阅《Amazon 登录用户指南》Amazon中的[如何登录](#)。

有关 IAM 和安全密钥设置说明的更多信息，请参阅[创建 IAM 用户](#)。

满足系统软件要求

用于运行应用程序的系统必须已安装 Java 7 或更高版本。要下载和安装最新 Java 开发工具包 (JDK)，请转到[Oracle 的 Java SE 安装站点](#)。

您需要最新的[适用于 Java 的 Amazon SDK](#) 版本。

消费者应用程序需要 Kinesis 客户端库 (KCL) 2.2.9 或更高版本，您可以从 [/tree/master](#) 上获取该版本。GitHub <https://github.com/awslabs/amazon-kinesis-client>

后续步骤

创建数据流

创建数据流

首先，您必须创建将在本教程的后续步骤中使用的数据流。

创建流

1. [登录Amazon Web Services 管理控制台并在 /kinesis 上打开 Kinesis 控制台。<https://console.aws.amazon.com>](#)
2. 在导航窗格中，选择 数据流。
3. 在导航栏中，展开区域选择器并选择一个区域。
4. 选择 Create Kinesis stream (创建 Kinesis 流)。
5. 输入数据流的名称（例如，**StockTradeStream**）。
6. 在分片数量中输入 **1**，但保留估计您需要的分片数量为折叠状态。
7. 选择 Create Kinesis stream (创建 Kinesis 流)。

在 Kinesis 流列表页面上，流状态在创建流的过程中显示为 CREATING。当流可以使用时，状态会更改为主动 (ACTIVE)。

如果您选择流的名称，在显示的页面中，Details (详细信息) 选项卡会显示数据流的配置摘要。Monitoring (监控) 部分显示流的监控信息。

后续步骤

创建 IAM 策略和用户

创建 IAM 策略和用户

的安全最佳实践Amazon要求使用细粒度的权限来控制对不同资源的访问权限。Amazon Identity and Access Management(IAM) 允许您在中管理用户和用户权限Amazon。[IAM policy](#) 明确列出了允许的操作以及这些操作适用于的资源。

下面是 Kinesis Data Streams 创建器和消费端通常需要的最低权限。

Producer

| 操作 | 资源 | 用途 |
|--|-------------|---|
| DescribeStream , DescribeStreamSummary , DescribeStreamConsumer | Kinesis 数据流 | 在尝试读取记录前，消费端会检查数据流是否存在，数据流及分片是否包含在数据流中。 |
| SubscribeToShard , RegisterStreamConsumer | Kinesis 数据流 | 订阅消费端并将其注册到分片。 |
| PutRecord , PutRecords | Kinesis 数据流 | 将记录写入 Kinesis Data Streams。 |

消费端

| 操作 | 资源 | 目的 |
|---|----------------------|--|
| DescribeStream | Kinesis 数据流 | 在尝试读取记录前，消费端会检查数据流是否存在，数据流及分片是否包含在数据流中。 |
| GetRecords , GetShardIterator | Kinesis 数据流 | 从分片读取记录。 |
| CreateTable , DescribeTable , GetItem, PutItem, Scan, UpdateItem | Amazon DynamoDB 表 | 如果消费端是使用 Kinesis Client Library (KCL) (版本 1.x)，要 DynamoDB 表的权限才能跟踪应用程序的处理状态。 |
| DeleteItem | Amazon DynamoDB 表 | 当使用者对 Kinesis Data Streams 分片执行 split/merge 操作时。 |
| PutMetricData | 亚马逊 CloudWatch 日志 | KCL 还会将指标上传到 CloudWatch，这对于监控应用程序具有帮助。 |

对于此教程，您将创建授予上述所有权限的单个 IAM 策略。在生产中，您可能需要创建两个策略，一个针对创建器，另一个针对消费端。

创建 IAM policy

1. 找到您在上一步中创建的新数据流的 Amazon 资源名称 (ARN)。您可以在详细信息选项卡顶部找到作为流 ARN 列出的此 ARN。ARN 格式如下所示：

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

区域

Amazon 地区代码；例如，us-west-2。有关更多信息，请参阅[区域和可用区域概念](#)。

账户

Amazon 账户 ID，如[账户设置](#)中所示。

name

您在上一步中创建的数据流的名称，即 StockTradeStream。

2. 确定要由消费端使用（并要由第一个消费端实例创建）的 DynamoDB 表的 ARN。它必须采用以下格式：

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

区域和账户 ID 与您在本教程中使用的数据流 ARN 中的值相同，但 name 是消费端应用程序创建并使用的 DynamoDB 表的名称。KCL 使用应用程序名称作为表名称。在此步骤中，将 StockTradesProcessor 用作 DynamoDB 表名称，因为这是本教程后续步骤中使用的应用程序名称。

3. 在 IAM 控制台的策略 (<https://console.aws.amazon.com/iam/home#policies>) 中，选择创建策略。如果这是您首次使用 IAM policy，请依次选择开始使用、创建策略。
4. 在 Policy Generator 旁，选择 Select。
5. 选择 Amazon Kinesis 作为服务。Amazon
6. 选择 DescribeStream、GetShardIterator、GetRecords、PutRecord 和 PutRecords 作为允许的操作。
7. 输入您在本教程中使用的数据流的 ARN。
8. 对以下各项使用 Add Statement (添加语句)：

| Amazon服务 | 操作 | 进行筛选 |
|-------------------|---|--|
| Amazon DynamoDB | CreateTable , DeleteItem , DescribeTable , GetItem, PutItem, Scan, UpdateItem | 您在此过程的步骤 2 中 创建的 DynamoDB 表 的 ARN。 |
| Amazon CloudWatch | PutMetricData | * |

在不需要指定 ARN 时使用的星号 (*)。在这种情况下，这是因为没有用于调用 PutMetricData 操作 CloudWatch 的特定资源。

9. 选择下一步。
10. 将 Policy Name (策略名称) 更改为 StockTradeStreamPolicy，审阅代码，然后选择 Create Policy (创建策略)。

生成的策略文档应该如下所示：

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-east-1:123456789012:stream/StockTradeStream"
      ]
    }
  ]
}
```

```
        "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": [
        "*"
    ]
}
]
```

若要创建 IAM 用户

1. 使用 <https://console.aws.amazon.com/iam/> 打开 IAM 控制台。

2. 在 Users (用户) 页面上，选择 Add user (添加用户)。
3. 对于 User name，键入 StockTradeStreamUser。
4. 对于 Access type (访问类型)，选择 Programmatic access (编程访问)，然后选择 Next: Permissions (下一步: 权限)。
5. 选择直接附上现有策略。
6. 按名称搜索您在前面的过程中创建的策略 (StockTradeStreamPolicy)。选中策略名称左侧的框，然后选择 Next: Review (下一步: 审核)。
7. 查看详细信息和摘要，然后选择 Create user (创建用户)。
8. 复制 Access key ID (访问密钥 ID)，并将其私下保存。在 Secret access key (私有访问密钥) 下面选择 Show (显示)，然后也将该密钥私下保存。
9. 将访问密钥和私有密钥粘贴到一个只有您可以访问的位于安全位置的本地文件中。对于此应用程序，请创建名为 `~/.aws/credentials` (具有严格权限) 的文件。该文件应采用以下格式：

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

将 IAM policy 附加到用户

1. 在 IAM 控制台中打开策略，然后选择策略操作。
2. 选择 StockTradeStreamPolicy 和 Attach (附加)。
3. 选择 StockTradeStreamUser 和 Attach Policy (附加策略)。

后续步骤

下载并构建代码

下载并构建代码

本主题提供了引入数据流 (创建器) 和处理此数据 (消费端) 的样本股票交易的示例实施代码。

下载并构建代码

1. 将源代码从 <https://github.com/aws-samples/amazon-kinesis-learning> GitHub repo 下载到您的计算机。
2. 按照提供的目录结构，使用源代码在您的 IDE 中创建一个项目。

3. 将以下库添加到该项目中：

- Amazon Kinesis Client Library (KCL)
- AmazonSDK
- Apache HttpCore
- Apache HttpClient
- Apache Commons Lang
- Apache Commons Logging
- Guava (适用于 Java 的 Google 核心库)
- Jackson Annotations
- Jackson Core
- Jackson Databind
- Jackson Dataformat : CBOR
- Joda Time

4. 根据您的 IDE，项目可能会自动构建。如果未自动构建项目，请使用适合您的 IDE 的步骤构建项目。

如果已成功完成这些步骤，则可进入下一节 [the section called “实现产生器”](#)。

后续步骤

实现产生器

此教程使用股票市场交易监控的实际场景。以下准则简要说明了此场景如何映射到创建器及其支持的代码结构。

请参阅[源代码](#)并查看以下信息。

StockTrade 班级

单次股票交易由一个 StockTrade 类实例表示。此实例包含一些属性，如股票代号、价格、股份数、交易类型（买入或卖出）以及唯一标识交易的 ID。将为您实现此类。

流记录

流是一个记录序列。记录是 JSON 格式的 StockTrade 实例序列化。例如：

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

StockTradeGenerator 班级

StockTradeGenerator 有一个名为的方法，每次调用getRandomTrade()该方法时都会返回一个新的随机生成的股票交易。将为您实现此类。

StockTradesWriter 班级

创建器的 main 方法 StockTradesWriter 将持续检索随机交易，然后通过执行以下任务将该交易发送到 Kinesis Data Streams：

1. 将数据流名称和区域名称作为输入读取。
2. 使用 KinesisAsyncClientBuilder 来设置区域、凭证和客户端配置。
3. 检查流是否存在且处于活动状态 (如果不是这样，它将退出并显示错误)。
4. 在连续循环中，会依次调用 StockTradeGenerator.getRandomTrade() 方法和 sendStockTrade 方法以便每 100 毫秒将交易发送到流一次。

sendStockTrade 类的 StockTradesWriter 方法具有以下代码：

```
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);
    }
}
```

请参阅以下代码细分：

- PutRecord API 需要一个字节数组，并且您必须将交易转换为 JSON 格式。此行代码将执行该操作：

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

- 您需要先创建新的 PutRecordRequest 实例（此示例中称为请求），然后才能发送交易。每个 request 均需要流名称、分区键和数据 Blob。

```
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();
```

该示例使用股票代码作为将记录映射到特定分片的分区键。实际上，每个分片应具有数百或数千个分区键，以便记录均匀地分布在流中。有关如何将数据添加到流的更多信息，请参阅 [将数据写入 Amazon Kinesis Data Streams](#)。

现在 request 已准备好发送到客户端（put 操作）：

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

- 错误检查和日志记录始终是有用的附加功能。此代码将记录错误条件：

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

在put操作周围添加 try/catch 方块：

```
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);  
}
```

这是因为 Kinesis 数据流 put 操作可能因网络错误或数据流达到其吞吐量限额并受到限制而导致失败。建议您仔细考虑针对 put 操作的重试策略以避免数据丢失，例如使用重试。

- 状态日志记录很有用，但它是可选的：

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

此处显示的创建器使用 Kinesis Data Streams API 单记录功能 PutRecord。实际上，如果单个创建者生成许多记录，则使用 PutRecords 的多记录功能并一次性发送批量记录通常会更有效。有关更多信息，请参阅 [将数据写入 Amazon Kinesis Data Streams](#)。

运行创建器

1. 验证在[创建 IAM 策略和用户](#)中检索到的访问密钥和私有密钥对是否保存到文件`~/.aws/credentials`中。
2. 使用以下参数运行`StockTradeWriter`类：

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

如果您在`us-west-2`之外的区域中创建流，则必须改为在此处指定该区域。

您应该可以看到类似于如下所示的输出内容：

```
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
```

您的股票交易现在正由 Kinesis Data Streams 摄取。

后续步骤

[实现消费端](#)

实现消费端

本教程中的消费端应用程序持续处理您数据流中的股票交易。然后，它输出每分钟买入和卖出最多的股票。该应用程序基于 Kinesis Client Library (KCL) 构建，后者需要完成对消费端应用程序常见的大量繁重工作。有关更多信息，请参阅[KCL 1.x 和 2.x 信息](#)。

请参阅源代码并查看以下信息。

StockTradesProcessor 班级

为您提供的消费端的主类，它将执行以下任务：

- 读取作为参数传递的应用程序名称、数据流名称和区域名称。
- 使用区域名称创建 `KinesisAsyncClient` 实例。
- 创建一个 `StockTradeRecordProcessorFactory` 实例，该实例提供由 `ShardRecordProcessor` 实例实施的 `StockTradeRecordProcessor` 的实例。
- 使用 `KinesisAsyncClient`、`StreamName`、`ApplicationName` 和 `StockTradeRecordProcessorFactory` 实例创建 `ConfigsBuilder` 实例。这对于创建具有默认值的所有配置非常有用。
- 使用 `ConfigsBuilder` 实例创建一个 KCL 计划程序（以前在 KCL 版本 1.x 中称为 KCL 工作线程）。
- 此计划程序为每个分片（已分配给此消费端实例）创建一个线程，以持续循环从数据量读取记录。之后，它调用 `StockTradeRecordProcessor` 实例以处理收到的每批记录。

StockTradeRecordProcessor 班级

`StockTradeRecordProcessor` 实例的实施，该实例反过来将实施五个必需方法：`initialize`、`processRecords`、`leaseLost`、`shardEnded` 和 `shutdownRequested`。

`initialize` 和 `shutdownRequested` 方法由 KCL 使用，旨在让记录处理器分别了解何时应准备好开始接收记录，以及何时应停止接收记录，因此该方法可以执行任何特定于应用程序的设置和终止任务。`leaseLost` 和 `shardEnded` 用于实施当租约丢失或处理达到分片末尾时需执行的操作的任何逻辑。在此示例中，我们只记录指示这些事件的消息。

将为您提供这些方法的代码。`processRecords` 方法中进行的主要处理，该处理反过来对每条记录使用 `processRecord`。后一个方法作为大体为空的框架代码提供给您，以便您在下一步骤中实施，届时将更详细地对其进行说明。

另外要注意的是对 `processRecord` 的支持方法 `reportStats` 和 `resetStats` 的实施，二者在初始源代码中为空。

已为您实施 `processRecords` 方法，并执行了以下步骤：

- 对于传入的每条记录，它会对其调用 `processRecord`。
- 如果自上一次报告以来已过去至少 1 分钟，请调用 `reportStats()`（它将打印出最新统计数据），然后调用 `resetStats()`（它将清除统计数据以便下一个间隔仅包含新记录）。
- 设置下一次报告时间。

- 如果自上一检查点以来已过去至少 1 分钟，请调用 `checkpoint()`。
- 设置下一次检查点操作时间。

此方法使用 60 秒间隔作为报告和检查点操作比率。有关检查点操作的更多信息，请参阅 [Using the Kinesis Client Library](#)。

StockStats 班级

此类提供一段时间内针对最热门股票的数据保留和统计数据跟踪。此代码已提供给您并包含以下方法：

- `addStockTrade(StockTrade)`：将给定的 `StockTrade` 注入正在使用的统计数据。
- `toString()`：以格式化字符串形式返回统计数据。

此类跟踪最热门股票的方式是，保留每只股票的总交易数的连续计数和最大计数。每当股票交易达成时，它都会更新这些计数。

将代码添加到 `StockTradeRecordProcessor` 类的方法，如以下步骤中所示。

实施消费端

1. 通过实例化大小正确的 `processRecord` 对象并将记录数据添加到该对象来实施 `StockTrade` 方法，并在出现问题时记录警告。

```
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. 实现 `reportStats` 方法。根据个人喜好修改输出格式。

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

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

3. 实施 `resetStats` 方法，这将创建新的 `stockStats` 实例。

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

4. 实现 `ShardRecordProcessor` 接口所需的以下方法：

```
@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);
}
}
```

运行消费端

1. 运行您在 中编写的创建者以将模拟股票交易记录引入流中。
2. 验证之前 (在创建 IAM 用户时) 检索到的访问密钥和私有密钥对是否保存到文件 `~/.aws/credentials` 中。
3. 使用以下参数运行 `StockTradesProcessor` 类：

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

请注意，如果您在 `us-west-2` 之外的区域中创建流，则必须改为在此处指定该区域。

1 分钟后，您应看到类似以下内容的输出，并且输出在此后每分钟刷新一次：

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

后续步骤

(可选) 扩展消费端

(可选) 扩展消费端

此可选部分演示如何针对更为复杂的场景扩展消费端代码。

如果要了解每分钟的最大销售订单数，可以修改三个位置的 `StockStats` 类以适应此新的优先级。

扩展消费端

1. 添加新实例变量：

```
// 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. 将以下代码添加到 `addStockTrade`：

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

3. 修改 `toString` 方法以打印其他信息：

```
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);
}
```

如果您现在运行消费端（请记住同时运行创建器），则应看到类似于以下内容的输出：

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

后续步骤

清理 资源

清理 资源

由于您需要付费使用 Kinesis 数据流，请确保在使用完后删除流和相应的 Amazon DynamoDB 表。即使您不发送和获取记录，活动流也会产生象征性的费用。这是因为活动流将持续“侦听”传入记录和获取记录的请求，这将耗用资源。

删除流和表

1. 关闭您可能仍在运行的任何产生器和消费端。
2. 在 [/kinesis 上打开 Kinesis 控制台](#)。<https://console.aws.amazon.com/kinesis>
3. 选择为此应用程序创建的流 (StockTradeStream)。
4. 选择 Delete Stream (删除流)。
5. 打开 DynamoDB 控制台，网址为。<https://console.aws.amazon.com/dynamodb/>
6. 删除 StockTradesProcessor 表。

Summary

近乎实时处理大量数据不需要编写任何复杂代码或开发大型基础设施。这就像编写逻辑来处理少量数据（如编写 `processRecord(Record)`）一样简单，但要使用 Kinesis Data Streams 进行扩展，才能让该逻辑处理大量流数据。您无需担心处理的扩展方式，因为 Kinesis Data Streams 将为您完成这一工作。您只需将流记录发送到 Kinesis Data Streams 并编写用于处理收到的每条新记录的逻辑。

以下是针对此应用程序的一些可能的改进。

跨所有分片进行聚合

当前，通过聚合单个分片中单个工作线程收到的数据记录来获得统计数据。（一个分片不能同时由单个应用程序中的多个工作线程处理。）当然，当您扩展并具有多个分片时，可能希望跨所有分片聚合。可通过部署管道架构完成此操作。在该架构中，每个工作线程的输出都注入具有单个分片的另一个流，分片由聚合第一个阶段输出的工作线程处理。由于来自第一个阶段的数据是有限的（每分片每分钟一个示例），因此一个分片即可轻松处理它。

缩放处理

当流进行扩展以包含多个分片（因为多个创建器正在发送数据）时，扩展处理的方式是添加更多工作程序。您可以在 Amazon EC2 实例中运行工作程序并使用 Auto Scaling 群组。

使用连接器连接亚马逊 S3/ DynamoDB/Amazon Redshift/Storm

在连续处理流时，其输出可以发送到其他目的地。Amazon 提供了 [用于将 Kinesis Data Streams Amazon 与其他服务和第三方工具集成的连接器](#)。

教程：使用 KPL 和 KCL 1.x 处理实时股票数据

本教程的场景涉及将股票交易引入数据流中并编写对流执行计算的简单 Amazon Kinesis Data Streams 应用程序。您将了解如何将记录流发送到 Kinesis Data Streams 并实现近乎实时地使用和处理记录的应用程序。

Important

创建直播后，您的账户会因使用 Kinesis Data Streams 而产生象征性的费用，因为 Kinesis Data Streams 不符合 Amazon 免费套餐的资格。在消费端应用程序启动后，也会象征性收取 Amazon DynamoDB 使用费用。消费端应用程序使用 DynamoDB 跟踪处理状态。在使用完此应用程序后，请删除 Amazon 资源以停止产生费用。有关更多信息，请参阅 [清理 资源](#)。

代码不访问实际股票市场数据，而是模拟股票交易流。它通过使用随机股票交易生成器（将截至 2015 年 2 月市值排名前 25 位的股票的实际市场数据作为起始点）来执行此操作。如果您有权访问实时的股票交易流，则可能有兴趣从该流派生有用且及时的统计数据。例如，您可能希望执行滑动窗口分析，从而确定前 5 分钟内购买的最热门股票。或者，您可能希望在销售订单过大（即具有过多股份）时收到通知。可以扩展此系列代码以提供此类功能。

您可以在台式机或笔记本电脑上完成本教程中的步骤，并在同一台计算机或任何支持已定义要求的平台（例如亚马逊弹性计算云 (Amazon EC2)）上运行生成器和使用者代码。

显示的示例使用的是美国西部（俄勒冈州）区域，但它们适用于[支持 Kinesis Data Streams 的任何 Amazon 区域](#)。

任务

- [满足先决条件](#)
- [创建数据流](#)
- [创建 IAM 策略和用户](#)
- [下载和构建实现代码](#)
- [实现产生器](#)
- [实现消费端](#)
- [（可选）扩展消费端](#)
- [清理 资源](#)

满足先决条件

以下是完成[教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#)的要求。

创建和使用亚马逊云科技账户

在开始之前，请确保熟悉[Amazon Kinesis Data Streams 术语和概念](#)中讨论的概念，特别是流、分片、创建者和消费端。参阅[教程：为 Kinesis Data Streams 安装和配置 Amazon CLI](#)也很有帮助。

您需要一个Amazon帐户和一个网络浏览器才能访问Amazon Web Services 管理控制台。

要访问控制台，请使用您的 IAM 用户名和密码从 IAM 登录页面登录[Amazon Web Services 管理控制台](#)。有关Amazon安全证书的信息，包括编程访问权限和长期证书的替代方案，请参阅 IAM 用户指南中的[Amazon安全证书](#)。有关登录您的的详细信息Amazon Web Services 账户，请参阅《Amazon 登录用户指南》Amazon中的[如何登录](#)。

有关 IAM 和安全密钥设置说明的更多信息，请参阅[创建 IAM 用户](#)。

满足系统软件要求

用于运行应用程序的系统必须已安装 Java 7 或更高版本。要下载和安装最新 Java 开发工具包 (JDK)，请转到[Oracle 的 Java SE 安装站点](#)。

如果您具有 Java IDE（如[Eclipse](#)），则可打开源代码，然后编辑、构建并运行它。

您需要最新的 [适用于 Java 的 Amazon SDK](#) 版本。如果您将 Eclipse 用作 IDE，则可改为安装 [Amazon Toolkit for Eclipse](#)。

消费者应用程序需要 Kinesis 客户端库 (KCL) 1.2.1 或更高版本，您可以从 [GitHub Kinesis 客户端库 \(Java\)](#) 中获取该版本。

后续步骤

[创建数据流](#)

创建数据流

在 [教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#) 的第一步中，创建后续步骤中将用到的流。

创建流

1. [登录Amazon Web Services 管理控制台并在 /kinesis 上打开 Kinesis 控制台。](#) <https://console.aws.amazon.com/kinesis/>
2. 在导航窗格中，选择 数据流。
3. 在导航栏中，展开区域选择器并选择一个区域。
4. 选择 Create Kinesis stream (创建 Kinesis 流)。
5. 输入流的名称（例如，**StockTradeStream**）。
6. 在分片数量中输入 **1**，但保留估计您需要的分片数量为折叠状态。
7. 选择 Create Kinesis stream (创建 Kinesis 流)。

在 Kinesis 流列表页面上，流状态在创建流的过程中为 CREATING。当流可以使用时，状态会更改为 ACTIVE。选择流的名称。在显示的页面中，Details (详细信息) 选项卡会显示您的流配置摘要。Monitoring (监控) 部分显示流的监控信息。

有关分片的其他信息

在本教程之外开始使用 Kinesis Data Streams 时，可能需要更仔细地计划流创建过程。在配置分片时，您应规划预计最大需求。以此方案为例，美国股票市场某一天（东部时间）的交易流量峰值以及需求估计值应该从这一天的时间中采样。随后，您可以选择配置最大预计需求，或扩大或缩小流以响应需求波动。

分片 是吞吐容量的单位。在创建 Kinesis 流页面中，展开估计您需要的分片数量。根据以下准则输入平均记录大小、每秒写入的最大记录数以及使用应用程序数量：

平均记录大小

您的记录的计算平均大小的估计值。如果您不知道此值，请使用估计的最大记录大小作为此值。

最大写入记录数

考虑提供数据的实体的数量以及每个实体每秒生成的记录的大约数量。例如，如果要从 20 台交易服务器获取股票交易数据，并且每台服务器每秒生成 250 次交易，则每秒的交易（记录）总数为 5000。

使用的应用程序数

应用程序的数量，这些应用程序单独从流进行读取以采用不同的方式处理流并生成不同的输出。每个应用程序可具有在不同计算机上运行（即在群集中运行）的多个实例，以便能跟进大容量流。

如果显示的估计分片数量超过当前分片数量限制，则可能需要先提交提高限制的请求，然后才能创建具有此分片数量的流。要请求增大分片限制，请使用 [Kinesis Data Streams 限制表单](#)。有关流和分片的更多信息，请参阅 [创建和管理 Kinesis 数据流](#)。

后续步骤

[创建 IAM 策略和用户](#)

创建 IAM 策略和用户

的安全最佳实践Amazon要求使用细粒度的权限来控制对不同资源的访问权限。Amazon Identity and Access Management(IAM) 允许您在中管理用户和用户权限Amazon。[IAM policy](#) 明确列出了允许的操作以及这些操作适用于的资源。

下面是 Kinesis Data Streams 创建器和消费端通常需要的最低权限。

Producer

| 操作 | 资源 | 用途 |
|--|-------------|---|
| DescribeStream , DescribeStreamSummary , DescribeStreamConsumer | Kinesis 数据流 | 在尝试写入记录之前，创建者会检查流是否存在且处于活动中，以及流是否具有消费端。 |

| 操作 | 资源 | 用途 |
|--|-------------|-----------------------------|
| SubscribeToShard , RegisterStreamConsumer | Kinesis 数据流 | 订阅 Kinesis 数据流分片并注册一个消费端。 |
| PutRecord , PutRecords | Kinesis 数据流 | 将记录写入 Kinesis Data Streams。 |

消费端

| 操作 | 资源 | 目的 |
|---|----------------------|--|
| DescribeStream | Kinesis 数据流 | 在尝试读取记录前，消费端需检查流是否存在并处于活动状态。在流中。 |
| GetRecords , GetShardIterator | Kinesis 数据流 | 从 Kinesis Data Streams 分片读取记录。 |
| CreateTable , DescribeTable , GetItem, PutItem, Scan, UpdateItem | Amazon DynamoDB 表 | 如果消费端是使用 Kinesis Client Library (KCL) 开发的，则需拥有此权限才能跟踪应用程序的处理状态。第一个消费端已开始创建表。 |
| DeleteItem | Amazon DynamoDB 表 | 当使用者对 Kinesis Data Streams 分片执行 split/merge 操作时。 |
| PutMetricData | 亚马逊 CloudWatch 日志 | KCL 还会将指标上传到 CloudWatch，这对于监控应用程序非常有用。 |

对于此应用程序，请创建授予上述所有权限的单个 IAM policy。实际上，您可能需要考虑创建两个策略，一个策略适用于创建器，另一个策略适用于消费端。

创建 IAM policy

- 找到新流的 Amazon 资源名称 (ARN)。您可以在详细信息选项卡顶部找到作为流 ARN 列出的此 ARN。ARN 格式如下所示：

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

区域

区域代码，例如 us-west-2。有关更多信息，请参阅[区域和可用区域概念](#)。

账户

Amazon账户 ID，如[账户设置](#)中所示。

name

[创建数据流](#) 中的流名称，即 StockTradeStream。

2. 确定要由消费端使用（并由第一个消费端实例创建）的 DynamoDB 表的 ARN。它必须采用以下格式：

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

区域和账户来自上一步骤中的相同位置，但这一次，名称为消费端应用程序创建和使用的表的名称。消费端所使用的 KCL 将应用程序名称用作表名称。使用 StockTradesProcessor，它是稍后使用的应用程序名称。

3. 在 IAM 控制台的策略 (h <https://console.aws.amazon.com/iam/home#policies>) 中，选择创建策略。如果这是您首次使用 IAM policy，请依次选择开始使用、创建策略。
4. 在 Policy Generator 旁，选择 Select。
5. 选择 Amazon Kinesis 作为服务。Amazon
6. 选择 DescribeStream、GetShardIterator、GetRecords、PutRecord 和 PutRecords 作为允许的操作。
7. 输入您在步骤 1 中创建的 ARN。
8. 对以下各项使用 Add Statement (添加语句)：

| Amazon服务 | 操作 | 进行筛选 |
|-----------------|--|--------------------|
| Amazon DynamoDB | CreateTable , DeleteItem , DescribeTable , | 您在步骤 2 中创建的 ARN |

| Amazon服务 | 操作 | 进行筛选 |
|-------------------|---------------------------------------|------|
| | GetItem, PutItem, Scan, UpdateItem | |
| Amazon CloudWatch | PutMetricData | * |

在不需要指定 ARN 时使用的星号 (*)。在这种情况下，这是因为没有用于调用 PutMetricData 操作 CloudWatch 的特定资源。

9. 选择下一步。
10. 将 Policy Name (策略名称) 更改为 StockTradeStreamPolicy，审阅代码，然后选择 Create Policy (创建策略)。

生成的策略文档应类似于以下内容：

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: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"  
            ]  
        }  
    ]  
}
```

```
        "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": [
            "*"
        ]
    }
]
```

若要创建 IAM 用户

1. 使用 <https://console.aws.amazon.com/iam/> 打开 IAM 控制台。
2. 在 Users (用户) 页面上，选择 Add user (添加用户)。
3. 对于 User name，键入 StockTradeStreamUser。
4. 对于 Access type (访问类型)，选择 Programmatic access (编程访问)，然后选择 Next: Permissions (下一步: 权限)。

5. 选择直接附上现有策略。
6. 按名称搜索您创建的策略。选中策略名称左侧的框，然后选择 Next: Review (下一步: 审核)。
7. 查看详细信息和摘要，然后选择 Create user (创建用户)。
8. 复制 Access key ID (访问密钥 ID)，并将其私下保存。在 Secret access key (私有访问密钥) 下面选择 Show (显示)，然后也将该密钥私下保存。
9. 将访问密钥和私有密钥粘贴到一个只有您可以访问的位于安全位置的本地文件中。对于此应用程序，请创建名为 `~/.aws/credentials` (具有严格权限) 的文件。该文件应采用以下格式：

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

将 IAM policy 附加到用户

1. 在 IAM 控制台中打开策略，然后选择策略操作。
2. 选择 StockTradeStreamPolicy 和 Attach (附加)。
3. 选择 StockTradeStreamUser 和 Attach Policy (附加策略)。

后续步骤

下载和构建实现代码

下载和构建实现代码

已提供 [the section called “教程：使用 KPL 和 KCL 1.x 处理实时股票数据”](#) 的框架代码。它包含用于股票交易流引入 (创建器) 和数据处理 (消费端) 的存根实施。以下过程演示如何完成实现。

下载和构建实施代码

1. 将源代码下载到计算机上。
2. 按照提供的目录结构，使用源代码在您喜爱的 IDE 中创建一个项目。
3. 将以下库添加到该项目中：
 - Amazon Kinesis Client Library (KCL)
 - AmazonSDK
 - Apache HttpCore

- Apache HttpClient
- Apache Commons Lang
- Apache Commons Logging
- Guava (适用于 Java 的 Google 核心库)
- Jackson Annotations
- Jackson Core
- Jackson Databind
- Jackson Dataformat : CBOR
- Joda Time

4. 根据您的 IDE，项目可能会自动构建。如果未自动构建项目，请使用适合您的 IDE 的步骤构建项目。

如果已成功完成这些步骤，则可进入下一节 [the section called “实现产生器”](#)。如果您的构建在任何阶段出错，请调查并纠正错误，然后再继续。

后续步骤

实现产生器

[教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#) 中的应用程序使用实际股票市场交易监控场景。以下准则简要说明了此场景如何映射到创建器和支持的代码结构。

请参阅源代码并查看以下信息。

StockTrade 班级

单次股票交易由一个 StockTrade 类实例表示。此实例包含一些属性，如股票代号、价格、股份数、交易类型（买入或卖出）以及唯一标识交易的 ID。将为您实现此类。

流记录

流是一个记录序列。记录是 JSON 格式的 StockTrade 实例序列化。例如：

```
{  
  "tickerSymbol": "AMZN",  
  "tradeType": "BUY",  
  "price": 395.87,
```

```
    "quantity": 16,  
    "id": 3567129045  
}
```

StockTradeGenerator 班级

StockTradeGenerator 包含一个名为 `getRandomTrade()` 的方法，当调用此方法时，它将返回一个随机生成的新股票交易。将为您实现此类。

StockTradesWriter 班级

创建器的 `main` 方法 StockTradesWriter 将持续检索随机交易，然后通过执行以下任务将该交易发送到 Kinesis Data Streams：

1. 将流名称和区域名称作为输入读取。
2. 创建一个 `AmazonKinesisClientBuilder`。
3. 使用客户端生成器来设置区域、凭证和客户端配置。
4. 使用客户端生成器构建一个 `AmazonKinesis` 客户端。
5. 检查流是否存在且处于活动状态 (如果不是这样，它将退出并显示错误)。
6. 在连续循环中，会依次调用 `StockTradeGenerator.getRandomTrade()` 方法和 `sendStockTrade` 方法以便每 100 毫秒将交易发送到流一次。

`sendStockTrade` 类的 StockTradesWriter 方法具有以下代码：

```
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);
}
```

请参阅以下代码细分：

- PutRecord API 需要一个字节数组，并且您必须将 trade 转换为 JSON 格式。此行代码将执行该操作：

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

- 您需要先创建新的 PutRecordRequest 实例（此示例中称为 putRecord），然后才能发送交易：

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

每个 PutRecord 调用均需要流名称、分区键和数据 Blob。以下代码使用 putRecord 对象的 setXXXX() 方法来填充该对象中的这些字段：

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

该示例使用股票行情自动收录器作为将记录映射到特定分片的分区键。实际上，每个分片应具有数百或数千个分区键，以便记录均匀地分布在流中。有关如何将数据添加到流的更多信息，请参阅 [向流添加数据](#)。

现在 putRecord 已准备好发送到客户端（put 操作）：

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

- 错误检查和日志记录始终是有用的附加功能。此代码将记录错误条件：

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

在put操作周围添加 try/catch 方块：

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

这是因为 Kinesis Data Streams put 操作可能因网络错误或数据流达到其吞吐量限额并受到限制而导致失败。建议您仔细考虑针对 put 操作的重试策略，以避免数据丢失，例如使用重试。

- 状态日志记录很有用，但它是可选的：

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

此处显示的创建器使用 Kinesis Data Streams API 单记录功能 PutRecord。实际上，如果单个创建者生成许多记录，则使用 PutRecords 的多记录功能并一次性发送批量记录通常会更有效。有关更多信息，请参阅 [向流添加数据](#)。

运行创建器

1. 验证之前（在创建 IAM 用户时）检索到的访问密钥和私有密钥对是否保存到文件 `~/.aws/credentials` 中。
2. 使用以下参数运行 `StockTradeWriter` 类：

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

如果您在 `us-west-2` 之外的区域中创建流，则必须改为在此处指定该区域。

您应该可以看到类似于如下所示的输出内容：

```
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
```

您的股票交易流现在正由 Kinesis Data Streams 摄取。

后续步骤

[实现消费端](#)

实现消费端

[教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#) 中的消费端应用程序持续处理您在 中创建的股票交易流。然后，它输出每分钟买入和卖出最多的股票。该应用程序基于 Kinesis Client Library (KCL) 构建，后者需要完成对消费端应用程序常见的大量繁重工作。有关更多信息，请参阅 [开发 KCL 1.x 消费端](#)。

请参阅源代码并查看以下信息。

StockTradesProcessor 班级

为您提供的消费端的主类，它将执行以下任务：

- 读取作为参数传递的应用程序名称、流名称和区域名称。
- 从 `~/.aws/credentials` 读取凭证。
- 创建一个 `RecordProcessorFactory` 实例，该实例提供由 `RecordProcessor` 实例实施的 `StockTradeRecordProcessor` 的实例。
- 利用 `RecordProcessorFactory` 实例和标准配置（包括流名称、凭证和应用程序名称）创建 KCL 工作程序。
- 此工作程序为每个分片（已分配给此消费端实例）创建一个线程，以持续循环读取 Kinesis Data Streams 中的记录。之后，它调用 `RecordProcessor` 实例以处理收到的每批记录。

StockTradeRecordProcessor 班级

`RecordProcessor` 实例的实施，该实例反过来将实施三个必需方法：`initialize`、`processRecords` 和 `shutdown`。

正如其名称所示，`initialize` 和 `shutdown` 由 Kinesis Client Library 使用，旨在让记录处理器了解何时应准备好开始接收记录以及何时应停止接收记录，因此该方法可以执行任何特定于应用程

序的设置和终止任务。将为您提供这些方法的代码。`processRecords` 方法中进行的主要处理，该处理反过来对每条记录使用 `processRecord`。后一个方法主要作为空框架代码提供给您，以便您在下一步骤中实施，届时将进一步对其进行说明。

另外要注意的是 `processRecord` 的支持方法 `reportStats` 和 `resetStats` 的实施，二者在初始源代码中为空。

已为您实施 `processRecords` 方法，并执行了以下步骤：

- 对于传入的每条记录，对其调用 `processRecord`。
- 如果自上一次报告以来已过去至少 1 分钟，请调用 `reportStats()`（它将打印出最新统计数据），然后调用 `resetStats()`（它将清除统计数据以便下一个间隔仅包含新记录）。
- 设置下一次报告时间。
- 如果自上一检查点以来已过去至少 1 分钟，请调用 `checkpoint()`。
- 设置下一次检查点操作时间。

此方法使用 60 秒间隔作为报告和检查点操作比率。有关检查点操作的更多信息，请参阅 [有关消费端的其他信息](#)。

StockStats 班级

此类提供一段时间内针对最热门股票的数据保留和统计数据跟踪。此代码已提供给您并包含以下方法：

- `addStockTrade(StockTrade)`：将给定的 `StockTrade` 注入正在使用的统计数据。
- `toString()`：以格式化字符串形式返回统计数据。

此类跟踪最热门股票的方式是，保留每只股票的总交易数的连续计数和最大计数。每当股票交易达成时，它都会更新这些计数。

将代码添加到 `StockTradeRecordProcessor` 类的方法，如以下步骤中所示。

实施消费端

1. 通过实例化大小正确的 `processRecord` 对象并将记录数据添加到该对象来实施 `StockTrade` 方法，并在出现问题时记录警告。

```
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. 实施简单的 `reportStats` 方法。可随时将输出格式修改为您的首选格式。

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

3. 最后，实施 `resetStats` 方法，这将创建新的 `stockStats` 实例。

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

运行消费端

1. 运行您在 中编写的创建者以将模拟股票交易记录引入流中。
2. 验证之前（在创建 IAM 用户时）检索到的访问密钥和私有密钥对是否保存到文件 `~/.aws/credentials` 中。
3. 使用以下参数运行 `StockTradesProcessor` 类：

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

请注意，如果您在 `us-west-2` 之外的区域中创建流，则必须改为在此处指定该区域。

1 分钟后，您应看到类似以下内容的输出，并且输出在此后每分钟刷新一次：

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

有关消费端的其他信息

如果熟悉 Kinesis Client Library 的好处（在 [开发 KCL 1.x 消费端](#) 中和其他位置已讨论），您可能想知道为何应在此处使用它。虽然您只使用单个分片流和单个消费端实例来处理它，但使用 KCL 实施消费

端仍会更轻松。将创建器部分中的代码实施步骤与消费端部分中的进行对比，您会发现实施消费端相对来说容易一些。这主要是因为 KCL 提供的服务。

在此应用程序中，您专注于实施可处理单条记录的记录处理器类。您无需担心如何从 Kinesis Data Streams 提取记录；当有可用的新记录时，KCL 就会提取这些记录并调用记录处理器。此外，不必担心分片和消费端实例的数量。如果已扩展流，则不必重写应用程序以处理多个分片或多个消费端实例。

检查点一词是指记录流中的点，其中直到迄今为止已经使用和处理的数据记录。如果应用程序崩溃，则从该点读取流，而不是从流的开头进行读取。检查点操作主题、各种设计模式及其最佳实践不在本章讨论范围之内。但是，生产环境中可能会涉及上述内容。

正如您在 中了解到的，Kinesis Data Streams API 中的 `put` 操作将采用分区键作为输入。Kinesis Data Streams 使用分区键作为跨多个分片拆分记录的机制（当流中有多个分片时）。相同的分区键将始终路由到同一个分片。这使得能够基于以下假设来设计用于处理特定分片的消费端：具有相同分区键的记录只会发送给该消费端，具有相同分区键的任何记录都不会在任何其他消费端处结束。因此，消费端的工作程序可聚合具有相同分区键的所有记录而不用担心丢失所需的数据。

在此应用程序中，消费端对记录的处理并不集中，因此您可以使用一个分片并在与 KCL 线程相同的线程中执行处理。但在实际应用中，请先考虑增加分片数量。在某些情况下，您可能需要将处理切换到其他线程或需要使用线程池（如果您的记录处理应是集中的）。这样一来，KCL 可以更快地提取新记录，而其他线程可并行处理记录。多线程设计并不是无关紧要的，应使用先进技术来实现，因此增加分片计数通常是最有效的纵向扩展方法。

后续步骤

(可选) 扩展消费端

(可选) 扩展消费端

[教程：使用 KPL 和 KCL 1.x 处理实时股票数据](#) 中的应用程序可能已足以达到您的目的。此可选部分演示如何针对更为复杂的场景扩展消费端代码。

如果要了解每分钟的最大销售订单数，可以修改三个位置的 `StockStats` 类以适应此新的优先级。

扩展消费端

1. 添加新实例变量：

```
// 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. 将以下代码添加到 addStockTrade :

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

3. 修改 `toString` 方法以打印其他信息 :

```
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);
}
```

如果您现在运行消费端（请记住同时运行创建器），则应看到类似于以下内容的输出：

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

后续步骤

清理 资源

清理 资源

由于您需要付费使用 Kinesis 数据流，请确保在使用完后删除流和相应的 Amazon DynamoDB 表。即使您不发送和获取记录，活动流也会产生象征性的费用。这是因为活动流将持续“侦听”传入记录和获取记录的请求，这将耗用资源。

删除流和表

1. 关闭您可能仍在运行的任何创建者和消费端。
2. 在 [/kinesis 上打开 Kinesis 控制台。](https://console.aws.amazon.com/kinesis) <https://console.aws.amazon.com/kinesis>
3. 选择为此应用程序创建的流 (StockTradeStream)。
4. 选择 Delete Stream (删除流)。
5. 打开 DynamoDB 控制台，网址为。<https://console.aws.amazon.com/dynamodb/>
6. 删除 StockTradesProcessor 表。

Summary

近乎实时处理大量数据不需要编写任何复杂代码或开发大型基础设施。这就像编写逻辑来处理少量数据（如编写 `processRecord(Record)`）一样简单，但要使用 Kinesis Data Streams 进行扩展，才能让该逻辑处理大量流数据。您无需担心处理的扩展方式，因为 Kinesis Data Streams 将为您完成这一工作。您只需将流记录发送到 Kinesis Data Streams 并编写用于处理收到的每条新记录的逻辑。

以下是针对此应用程序的一些可能的改进。

跨所有分片进行聚合

当前，通过聚合单个分片中单个工作线程收到的数据记录来获得统计数据。（一个分片不能同时由单个应用程序中的多个工作线程处理。）当然，当您扩展并具有多个分片时，可能希望跨所有分片聚合。可通过部署管道架构完成此操作。在该架构中，每个工作线程的输出都注入具有单个分片的另一个流，分片由聚合第一个阶段输出的工作线程处理。由于来自第一个阶段的数据是有限的（每分片每分钟一个示例），因此一个分片即可轻松处理它。

缩放处理

当流进行扩展以包含多个分片（因为多个创建器正在发送数据）时，扩展处理的方式是添加更多工作程序。您可以在 Amazon EC2 实例中运行工作程序并使用 Auto Scaling 群组。

使用连接器连接亚马逊 S3/ DynamoDB/Amazon Redshift/Storm

在连续处理流时，其输出可以发送到其他目的地。Amazon提供了[用于将 Kinesis Data Streams 与 Amazon 与其他服务和第三方工具集成的连接器](#)。

后续步骤

- 有关使用 Kinesis Data Streams API 操作的更多信息，请参阅 [使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon SDK 开发产生器](#)、[使用适用于 Java 的 Amazon SDK 开发具有共享吞吐量的消费端](#) 和 [创建和管理 Kinesis 数据流](#)。
- 有关 Kinesis Client Library 的更多信息，请参阅 [开发 KCL 1.x 消费端](#)。
- 有关如何优化应用程序的更多信息，请参阅 [优化 Amazon Kinesis Data Streams 消费端](#)。

教程：使用适用于 Apache Flink 的亚马逊托管服务分析实时股票数据

本教程的场景涉及将股票交易引入数据流中并编写对流执行计算的简单的[适用于 Apache Flink 的亚马逊托管服务](#)应用程序。您将了解如何将记录流发送到 Kinesis Data Streams 并实现近乎实时地使用和处理记录的应用程序。

借助适用于 Apache Flink 的亚马逊托管服务，您可以使用 Java 或 Scala 来处理和分析流数据。该服务能让您根据流式处理源编写并运行 Java 或 Scala 代码，以执行时间序列分析、为实时控制面板提供信息和创建实时指标。

您可以在适用于 Apache Flink 的托管服务中使用基于 [Apache Flink](#) 的开源库构建 Flink 应用程序。Apache Flink 是处理数据流的常用框架和引擎。

Important

在您创建两个数据流和一个应用程序后，您的账户会产生名义上的 Kinesis Data Streams 和 Apache Flink 托管服务使用费，因为它们不符合免费套餐的资格。Amazon 使用完此应用程序后，请删除您的Amazon资源以停止产生费用。

代码不访问实际股票市场数据，而是模拟股票交易流。它通过使用随机股票交易生成器来实现这一点。如果您有权访问实时的股票交易流，则可能有兴趣从该流派生有用且及时的统计数据。例如，您可能

希望执行滑动窗口分析，从而确定前 5 分钟内购买的最热门股票。或者，您可能希望在销售订单过大（即具有过多股份）时收到通知。可以扩展此系列代码以提供此类功能。

显示的示例使用美国西部（俄勒冈州）区域，但它们适用于[支持适用于 Apache Flink 的托管服务的任何 Amazon 区域](#)。

任务

- [完成练习的先决条件](#)
- [设置Amazon账户并创建管理员用户](#)
- [设置 Amazon Command Line Interface \(Amazon CLI\)](#)
- [创建并运行适用于 Apache Flink 的托管服务应用程序](#)

完成练习的先决条件

要完成本指南中的步骤，您必须满足以下条件：

- [Java 开发工具包 \(JDK\)](#) 版本 8。设置 JAVA_HOME 环境变量，使其指向您的 JDK 安装位置。
- 我们建议您使用开发环境（如 [Eclipse Java Neon](#) 或 [IntelliJ Idea](#)）来开发和编译您的应用程序。
- [Git 客户端](#)。如果尚未安装 Git 客户端，请安装它。
- [Apache Maven 编译器插件](#)。Maven 必须位于您的有效路径中。要测试您的 Apache Maven 安装，请输入以下内容：

```
$ mvn -version
```

要开始，请转到[设置Amazon账户并创建管理员用户](#)。

设置Amazon账户并创建管理员用户

首次使用适用于 Apache Flink 的亚马逊托管服务之前，请完成以下任务：

1. [报名参加 Amazon](#)
2. [创建 IAM 用户](#)

报名参加 Amazon

当您注册亚马逊 Web Services (Amazon) 时，您的Amazon账户将自动注册所有服务Amazon，包括适用于 Apache Flink 的亚马逊托管服务。您只需为使用的服务付费。

借助适用于 Apache Flink 的托管服务，您仅需为实际使用的资源付费。如果您是 Amazon 新客户，还可以免费试用适用于 Apache Flink 的托管服务。有关更多信息，请参阅 [Amazon 免费套餐](#)。

如果您已经有一个Amazon帐户，请跳到下一个任务。如果您没有 Amazon 账户，请执行这些步骤创建一个账户。

创建Amazon账户

1. 打开[https://portal.aws.amazon.com/billing/注册。](https://portal.aws.amazon.com/billing/)
2. 按照屏幕上的说明操作。

在注册时，将接到电话或收到短信，要求使用电话键盘输入一个验证码。

当您注册时Amazon Web Services 账户，就会创建Amazon Web Services 账户根用户一个。根用户有权访问该账户中的所有 Amazon Web Services 服务 和资源。作为最佳安全实践，请为用户分配管理访问权限，并且只使用根用户来执行[需要根用户访问权限的任务](#)。

记下您的Amazon账户 ID，因为下个任务需要使用它。

创建 IAM 用户

中的服务Amazon，例如适用于 Apache Flink 的亚马逊托管服务，要求您在访问时提供凭证。这样，服务才能确定您是否有权访问该服务所拥有的资源。Amazon Web Services 管理控制台要求您输入密码。

您可以为Amazon账户创建访问密钥以访问 Amazon Command Line Interface (Amazon CLI) 或 API。但是，我们不建议您Amazon使用Amazon账户凭证进行访问。相反，我们建议您使用 Amazon Identity and Access Management (IAM)。创建 IAM 用户，将该用户添加到具有管理权限的 IAM 组，然后向您创建的 IAM 用户授予管理权限。您随后便可以使用一个特殊的 URL 和该 IAM 用户的凭证访问 Amazon。

如果您已注册Amazon，但尚未为自己创建 IAM 用户，则可以使用 IAM 控制台创建一个。

本指南中的入门练习假定您拥有具有管理员权限的用户 (adminuser)。请按照以下过程在您的账户中创建 adminuser。

为管理员创建组

1. 登录Amazon Web Services 管理控制台并打开 IAM 控制台，网址为 <https://console.aws.amazon.com/iam/>。
2. 在导航窗格中，选择 Groups (组)，然后选择 Create New Group (创建新组)。
3. 对于组名，输入组的名称，例如 **Administrators**，然后选择 下一步。
4. 在策略列表中，选中 AdministratorAccess 策略旁边的复选框。您可以使用 Filter (筛选) 菜单和 Search (搜索) 框来筛选策略列表。
5. 选择 Next Step (下一步)，然后选择 Create Group (创建组)。

您的新组列在 Group Name 下方。

要为您自己创建 IAM 用户，请将用户添加到管理员组，并创建密码

1. 在导航窗格中，选择用户，然后选择添加用户。
2. 在 User name(用户名) 框中，输入一个用户名。
3. 选择编程访问和 Amazon 管理控制台访问。
4. 选择下一步: 权限。
5. 选中 Administrators 组旁的复选框。然后选择 Next: Review。
6. 选择创建用户。

以新 IAM 用户身份登录

1. 退出Amazon Web Services 管理控制台。
2. 使用下面的 URL 格式登录控制台：

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

*aws_account_number*这是您的Amazon账户 ID，不含任何连字符。例如，如果您的Amazon账户编号为 1234-5678-9012，请替换为。*aws_account_number* **123456789012**有关如何查找您的账号的信息，请参阅 IAM 用户指南中的[您的Amazon账户 ID 及其别名](#)。

3. 输入您刚创建的 IAM 用户名和密码。登录后，导航栏会显示 *your_user_name* @ *your_aws_account_id*。

Note

如果您不希望登录页面的 URL 包含您的Amazon账户 ID，则可以创建账户别名。

创建或删除账户别名

1. 使用 <https://console.aws.amazon.com/iam/> 打开 IAM 控制台。
2. 在导航窗格上，选择 Dashboard。
3. 查找 IAM 用户登录链接。
4. 要创建别名，请选择 Customize (自定义)。输入要用于别名的名称，然后选择 Yes, Create (是，创建)。
5. 要删除别名，请选择 Customize，然后选择 Yes, Delete。登录网址将恢复为使用您的Amazon账户 ID。

要在创建账户别名后登录，请使用以下 URL：

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

要为您的账户验证 IAM 用户的登录链接，请打开 IAM 控制台并在控制面板的 IAM 用户登录链接下进行检查。

有关 IAM 的更多信息，请参阅以下文档：

- [Amazon Identity and Access Management\(IAM\)](#)
- [IAM 入门](#)
- [IAM 用户指南](#)

后续步骤

[设置 Amazon Command Line Interface \(Amazon CLI\)](#)

设置 Amazon Command Line Interface (Amazon CLI)

在此步骤中，您将下载并配置为与适用于 Apache Flink 的亚马逊托管服务一起使用。Amazon CLI

Note

本指南中的入门练习假定您使用账户中的管理员凭证 (adminuser) 来执行这些操作。

Note

如果您已经Amazon CLI安装了，则可能需要升级才能获得最新功能。有关更多信息，请参阅 [《Amazon Command Line Interface用户指南》中的安装Amazon命令行界面](#)。要检查的版本Amazon CLI，请运行以下命令：

```
aws --version
```

本教程中的练习需要以下Amazon CLI版本或更高版本：

```
aws-cli/1.16.63
```

要设置 Amazon CLI

1. 下载并配置 Amazon CLI。有关说明，请参阅 [《Amazon Command Line Interface用户指南》中的以下主题](#)：
 - [安装 Amazon Command Line Interface](#)
 - [配置 Amazon CLI](#)
2. 在配置文件中为管理员用户添加命名的Amazon CLI配置文件。执行Amazon CLI命令时使用此配置文件。有关命名配置文件的更多信息，请参阅 [Amazon Command Line Interface 用户指南中的命名配置文件](#)。

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

有关可用Amazon区域的列表，请参阅中的[Amazon区域和终端节点Amazon Web Services 一般参考](#)。

3. 在命令提示符处输入以下帮助命令来验证设置：

```
aws help
```

设置Amazon帐户和之后Amazon CLI，您可以尝试下一个练习，即配置示例应用程序并测试 end-to-end设置。

后续步骤

[创建并运行适用于 Apache Flink 的托管服务应用程序](#)

创建并运行适用于 Apache Flink 的托管服务应用程序

在本练习中，您将创建面向应用程序的适用于 Apache Flink 的托管服务，并将数据流作为源和接收器。

本节包含以下步骤：

- [创建两个 Amazon Kinesis 数据流](#)
- [将示例记录写入输入流](#)
- [下载并检查 Apache Flink 流式处理 Java 代码](#)
- [编译应用程序代码](#)
- [上传 Apache Flink 流式处理 Java 代码](#)
- [创建并运行适用于 Apache Flink 的托管服务](#)

创建两个 Amazon Kinesis 数据流

在为本练习创建适用于 Apache Flink 的亚马逊托管服务之前，请创建两个 Kinesis 数据流（`ExampleInputStream` 和 `ExampleOutputStream`）。您的应用程序将这些数据流用于应用程序源和目标流。

可以使用 Amazon Kinesis 控制台或以下 Amazon CLI 命令创建这些流。有关控制台说明，请参阅[创建和更新数据流](#)。

创建数据流 (Amazon CLI)

1. 要创建第一个直播 (`ExampleInputStream`)，请使用以下 Amazon Kinesis 命令`create-stream`Amazon CLI。

```
$ aws kinesis create-stream \
```

```
--stream-name ExampleInputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

2. 要创建应用程序用来写入输出的第二个流，请运行同一命令（将流名称更改为 ExampleOutputStream）。

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

将示例记录写入输入流

在本节中，您使用 Python 脚本将示例记录写入流，以供应用程序处理。

 Note

此部分需要 [Amazon SDK for Python \(Boto\)](#)。

1. 使用以下内容创建名为 stock.py 的文件：

```
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. 在本教程的后面部分，您运行 `stock.py` 脚本，以将数据发送到应用程序。

```
$ python stock.py
```

下载并检查 Apache Flink 流式处理 Java 代码

此示例的 Java 应用程序代码可从中获得 GitHub。要下载应用程序代码，请执行以下操作：

1. 使用以下命令克隆远程存储库：

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

2. 导航到 `GettingStarted` 目录。

应用程序代码位于 `CustomSinkStreamingJob.java` 和 `CloudWatchLogSink.java` 文件中。请注意有关应用程序代码的以下信息：

- 应用程序使用 Kinesis 源从源流中进行读取。以下代码段创建 Kinesis 接收器：

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

编译应用程序代码

在本节中，您使用 Apache Maven 编译器创建应用程序的 Java 代码。有关安装 Apache Maven 和 Java 开发工具包 (JDK) 的信息，请参阅[完成练习的先决条件](#)。

您的 Java 应用程序需要以下组件：

- 一个[项目对象模型 \(pom.xml\)](#) 文件。此文件包含有关应用程序的配置和从属项的信息，包括适用于 Apache Flink 的亚马逊托管服务库。
- 它是一种 main 方法，其中包含应用程序的逻辑。

 Note

要将 Kinesis 连接器用于以下应用程序，您必须下载连接器源代码并构建该连接器，如[Apache Flink 文档](#)中所述。

创建并编译应用程序代码

1. 在您的开发环境中创建 Java/Maven 应用程序。有关创建应用程序的信息，请参阅有关开发环境的文档：
 - [创建您的第一个 Java 项目 \(Eclipse Java Neon\)](#)
 - [创建、运行和打包您的第一个 Java 应用程序 \(IntelliJ Idea\)](#)
2. 将以下代码用于名为 StreamingJob.java 的文件。

```
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");
    }
}
```

请注意以下有关上述代码示例的信息：

- 此文件包含 `main` 方法，它定义应用程序的功能。
- 您的应用程序使用 `StreamExecutionEnvironment` 对象创建源和接收连接器以访问外部资源。

- 该应用程序将使用静态属性创建源和接收连接器。要使用动态应用程序属性，请使用 `createSourceFromApplicationProperties` 和 `createSinkFromApplicationProperties` 方法以创建连接器。这些方法读取应用程序的属性来配置连接器。

3. 要使用您的应用程序代码，您将其编译和打包成 JAR 文件。您可以通过两种方式之一编译和打包您的代码：

- 使用命令行 Maven 工具。在包含 `pom.xml` 文件的目录中通过运行以下命令创建您的 JAR 文件：

`mvn package`

- 设置开发环境。有关详细信息，请参阅您的开发环境文档。

您可以作为 JAR 文件上传您的包，也可以将包压缩为 ZIP 文件并上传。如果使用创建应用程序 Amazon CLI，则需要指定代码内容类型 (JAR 或 ZIP)。

4. 如果编译时出错，请验证 `JAVA_HOME` 环境变量设置正确。

如果应用程序成功编译，则创建以下文件：

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

上传 Apache Flink 流式处理 Java 代码

在本节中，您创建 Amazon Simple Storage Service (Amazon S3) 存储桶并上传应用程序代码。

上传应用程序代码

- 打开 Amazon S3 控制台，网址为 <https://console.aws.amazon.com/s3/>。
- 选择 创建存储桶。
- 在 存储桶名称 字段中输入 `ka-app-code-<username>`。将后缀 (如您的用户名) 添加到存储桶名称，以使其具有全局唯一性。选择 下一步。
- 在配置选项步骤中，让设置保持原样，然后选择下一步。
- 在设置权限步骤中，让设置保持原样，然后选择下一步。
- 选择 创建存储桶。
- 在 Amazon S3 控制台中，选择 `ka-app-code-<username>` 存储桶，然后选择上传。

8. 在选择文件步骤中，选择添加文件。导航到您在上一步中创建的 `java-getting-started-1.0.jar` 文件。选择 下一步。
9. 在设置权限步骤中，让设置保持原样。选择 下一步。
10. 在设置属性步骤中，让设置保持原样。选择上传。

您的应用程序代码现在存储在 Amazon S3 存储桶中，应用程序可以在其中访问代码。

创建并运行适用于 Apache Flink 的托管服务

您可以使用控制台或 Amazon CLI 创建和运行适用于 Apache Flink 的托管服务的应用程序。

Note

当您使用控制台创建应用程序时，系统会为您创建您的 Amazon Identity and Access Management (IAM) 和 Amazon CloudWatch Logs 资源。使用创建应用程序时 Amazon CLI，可以单独创建这些资源。

主题

- [创建并运行应用程序（控制台）](#)
- [创建并运行应用程序（Amazon CLI）](#)

创建并运行应用程序（控制台）

按照以下步骤，使用控制台创建、配置、更新和运行应用程序。

创建应用程序

1. [在 /kinesis 上打开 Kinesis 控制台。https://console.aws.amazon.com](#)
2. 在 Amazon Kinesis 控制面板上，选择创建分析应用程序。
3. 在 Kinesis Analytics – 创建应用程序页面上，提供应用程序详细信息，如下所示：
 - 对于 应用程序名称，输入 **MyApplication**。
 - 对于 描述，输入 **My java test app**。
 - 对于 Runtime (运行时)，请选择 Apache Flink 1.6。
4. 对于访问权限，请选择 创建/更新 IAM 角色 **kinesis-analytics-MyApplication-us-west-2**。

5. 选择创建应用程序。

Note

在使用控制台创建适用于 Apache Flink 的亚马逊托管服务应用程序时，您可以选择为应用程序创建 IAM 角色和策略。您的应用程序使用此角色和策略访问其从属资源。这些 IAM 资源是使用您的应用程序名称和区域命名的，如下所示：

- 策略：kinesis-analytics-service-*MyApplication-us-west-2*
- 角色：kinesis-analytics-*MyApplication-us-west-2*

编辑 IAM 策略

编辑 IAM policy 以添加访问 Kinesis 数据流的权限。

1. 使用 <https://console.aws.amazon.com/iam/> 打开 IAM 控制台。
2. 选择策略。选择控制台在上一部分中为您创建的 **kinesis-analytics-service-MyApplication-us-west-2** 策略。
3. 在 摘要 页面上，选择 编辑策略。选择 JSON 选项卡。
4. 将以下策略示例中突出显示的部分添加到策略中。将示例账户 IDs (**012345678901**) 替换为您的账户 ID。

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

```
        "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/  
ExampleOutputStream"  
    }  
}  
}
```

配置应用程序

1. 在 **MyApplication** 页面上，选择配置。
2. 在 **配置应用程序** 页面上，提供 代码位置：
 - 对于 Amazon S3 存储桶，请输入 **ka-app-code-<username>**。
 - 在 Amazon S3 对象的路径中，输入 **java-getting-started-1.0.jar**。
3. 在 **对应用程序的访问权限** 下，对于 **访问权限**，选择 **创建/更新 IAM 角色 kinesis-analytics-MyApplication-us-west-2**。
4. 在 **Properties (属性)** 下，对于 **Group ID (组 ID)**，输入 **ProducerConfigProperties**。
5. 输入以下应用程序属性和值：

| 键 | 值 |
|----------------------------------|------------------|
| flink.inputstream.initpos | LATEST |
| aws:region | us-west-2 |
| AggregationEnabled | false |

6. 在 **监控** 下，确保 **监控指标级别** 设置为 **应用程序**。
7. 要进行 CloudWatch 日志记录，请选中“**启用**”复选框。
8. 选择更新。

Note

当您选择启用 CloudWatch 日志记录时，适用于 Apache Flink 的托管服务会为您创建日志组和日志流。这些资源的名称如下所示：

- 日志组：`/aws/kinesis-analytics/MyApplication`

- 日志流 : kinesis-analytics-log-stream

运行应用程序

1. 在MyApplication页面上，选择“运行”。确认该操作。
2. 当应用程序正在运行时，请刷新页面。控制台将显示 Application graph (应用程序图表)。

停止应用程序

在MyApplication页面上，选择“停止”。确认该操作。

更新应用程序

使用控制台，您可以更新应用程序设置，例如应用程序属性、监控设置，或应用程序 JAR 文件的位置和文件名。如果您需要更新应用程序代码，您还可以从 Amazon S3 存储桶重新加载应用程序 JAR。

在MyApplication页面上，选择配置。更新应用程序设置，然后选择更新。

创建并运行应用程序 (Amazon CLI)

在本节中，您将使用创建和运行适用于 Amazon CLI 的 Apache Flink 的托管服务应用程序。适用于 Apache Flink 的托管服务使用该 `kinesisanalyticsv2` Amazon CLI 命令为 Apache Flink 应用程序创建托管服务并与之交互。

创建权限策略

首先，使用两个语句创建权限策略：一个语句授予对源流执行 `read` 操作的权限，另一个语句授予对接收器流执行 `write` 操作的权限。然后，将策略附加到 IAM 角色（下一部分中将创建此角色）。因此，在适用于 Apache Flink 的托管服务代入该角色时，服务具有必要的权限从源流进行读取和写入接收器流。

使用以下代码创建 `KAReadSourceStreamWriteSinkStream` 权限策略。将 `username` 替换为您用于创建 Amazon S3 存储桶来存储应用程序代码的用户名。将 Amazon 资源名称 (ARNs) (`012345678901`) 中的账户 ID 替换为您的账户 ID。

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"
  }
]
```

有关创建权限策略的 step-by-step 说明，请参阅 IAM 用户指南中的[教程：创建并附加您的第一个客户托管策略](#)。

 Note

要访问其他 Amazon 服务，可以使用适用于 Java 的 Amazon SDK。Managed Service for Apache Flink 会自动将软件开发工具包所需的证书设置为与您的应用程序关联的服务执行 IAM 角色的证书。无需执行其他步骤。

创建 IAM 角色

在本节中，您将创建一个 IAM 角色，适用于 Apache Flink 的托管服务可以代入此角色来读取源流和写入接收器流。

权限不足时，适用于 Apache Flink 的托管服务无法访问您的串流。您通过 IAM 角色授予这些权限。每个 IAM 角色附加了两种策略。此信任策略授予适用于 Apache Flink 的托管服务代入该角色的权限，权限策略确定适用于 Apache Flink 的托管服务代入这个角色后可以执行的操作。

您将在上一部分中创建的权限策略附加到此角色。

创建 IAM 角色

1. 使用 <https://console.aws.amazon.com/iam/> 打开 IAM 控制台。
2. 在导航窗格中，选择 角色 和 创建角色。
3. 在 选择受信任实体的类型 下，选择 Amazon 服务。在 选择将使用此角色的服务 下，选择 Kinesis。在选择您的使用案例下，选择 Kinesis Analytics。

选择下一步: 权限。

4. 在 附加权限策略 页面上，选择 下一步: 审核。在创建角色后，您可以附加权限策略。
5. 在 创建角色 页面上，输入**KA-stream-rw-role**作为角色名称。选择 创建角色。

现在，您已经创建了一个名为 KA-stream-rw-role 的新 IAM 角色。接下来，您更新角色的信任和权限策略。

6. 将权限策略附加到角色。

Note

对于本练习，适用于 Apache Flink 的托管服务代入此角色，以便同时从 Kinesis 数据流（源）读取数据和将输出写入另一个 Kinesis 数据流。因此，您附加在上一步（[the section called “创建权限策略”](#)）中创建的策略。

- a. 在 摘要 页上，选择 权限 选项卡。
- b. 选择附加策略。
- c. 在搜索框中，输入 **KAResourceStreamWriteSinkStream**（您在上一部分中创建的策略）。
- d. 选择**KAResourceStreamWriteSinkStream**策略，然后选择附加策略。

现在，您已经创建了应用程序用来访问资源的服务执行角色。记下新角色的 ARN。

有关创建角色的 step-by-step 说明，请参阅 [IAM 用户指南中的创建 IAM 角色（控制台）](#)。

创建适用于 Apache Flink 的托管服务应用程序

1. 将以下 JSON 代码保存到名为 `create_request.json` 的文件中。将示例角色 ARN 替换为您之前创建的角色的 ARN。将存储桶 ARN 后缀 (`username`) 替换为在前一部分中选择的后缀。将服务执行角色中的示例账户 ID (`012345678901`) 替换为您的账户 ID。

```
{  
    "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. 使用上述请求执行 [CreateApplication](#) 操作来创建应用程序：

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

应用程序现已创建。您在下一步中启动应用程序。

启动应用程序

在本节中，您使用 [StartApplication](#) 操作来启动应用程序。

启动应用程序

1. 将以下 JSON 代码保存到名为 start_request.json 的文件中。

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

2. 使用上述请求执行 [StartApplication](#) 操作来启动应用程序：

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

应用程序正在运行。您可以在亚马逊 CloudWatch 控制台上查看托管服务的 Apache Flink 指标，以验证应用程序是否正常运行。

停止应用程序

在本节中，您使用 [StopApplication](#) 操作来停止应用程序。

停止应用程序

1. 将以下 JSON 代码保存到名为 `stop_request.json` 的文件中。

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

2. 使用下面的请求执行 [StopApplication](#) 操作来停止应用程序：

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

应用程序现已停止。

教程：将 Amazon Lambda 与 Amazon Kinesis Data Streams 搭配使用

在本教程中，您将创建 Lambda 函数来处理 Kinesis 数据流中的事件。在此示例用例中，自定义应用程序会将记录写入 Kinesis 数据流。然后 Amazon Lambda 会轮询此数据流，如果检测到新的数据记录，则调用 Lambda 函数。之后 Amazon Lambda 通过代入您在创建 Lambda 函数时指定的执行角色来执行该 Lambda 函数。

有关详细的分步说明，请参阅[教程：将 Amazon Lambda 与 Amazon Kinesis 结合使用](#)。

Note

本教程假设您对 Lambda 基本操作和 Amazon Lambda 控制台有一定了解。如果您还没有了解，请按照[Amazon Lambda 入门](#)中的说明创建您的第一个 Lambda 函数。

使用适用于 Amazon Kinesis 的Amazon流数据解决方案

适用于 Amazon Kinesis 的Amazon流数据解决方案可自动配置必要的Amazon服务，以便轻松捕获、存储、处理和交付流数据。该解决方案为解决使用多种Amazon服务的流数据用例提供了多种选项，包括 Kinesis Data Streams Amazon Lambda、Amazon API Gateway 和适用于 Apache Flink 的亚马逊托管服务。

每个解决方案都包含以下组件：

- 用于部署完整示例的 Amazon CloudFormation 软件包。
- 用于显示应用程序指标的 CloudWatch 仪表板。
- CloudWatch 有关最相关的应用程序指标的警报。
- 所有必要的 IAM 角色和策略。

解决方案见此处：[适用于 Amazon Kinesis 的流式处理数据解决方案](#)

创建和管理 Kinesis 数据流

Amazon Kinesis Data Streams 实时吸收大量数据、持久存储数据并使这些数据可供使用。Kinesis Data Streams 存储的数据单位是数据记录。数据流 表示一组数据记录。数据流中的数据记录将分发到分片中。

分片具有流中的一系列数据记录。它是 Kinesis 数据流的基本吞吐量单位。在按需 MB/s 和预配置容量模式下，分片支持每秒 1 和 1000 条写入记录以及每秒 2 MB/s 条读取记录。分片限制确保性能可预测，使设计和操作高度可靠的数据流式传输工作流程变得更加容易。

在本节中，您将学习如何为直播设置容量模式，以及如何使用Amazon Web Services 管理控制台或创建直播 APIs。然后，您可以对流执行其他操作。

主题

- [选择正确的流式传入的模式](#)
- [使用创建直播 Amazon Web Services 管理控制台](#)
- [使用 APIs 创建流](#)
- [更新流](#)
- [列出流](#)
- [列出分片](#)
- [删除流](#)
- [对流进行重新分片](#)
- [更改数据留存期](#)
- [标记 Amazon Kinesis Data Streams 资源](#)
- [处理大型记录](#)
- [使用执行弹性测试 Amazon Fault Injection Service](#)

选择正确的流式传入的模式

以下主题说明了如何为应用程序选择最佳模式，以及如何根据需要切换模式。

主题

- [Kinesis Data Streams 中有哪些不同的模式？](#)
- [按需标准模式的特征和用例](#)

- [按需优势模式的特征和用例](#)
- [预置模式的特征和用例](#)
- [切换模式](#)

Kinesis Data Streams 中有哪些不同的模式？

模式决定如何管理数据流的容量以及如何对数据流的使用收费。在 Amazon Kinesis Data Streams 中，您可以为数据流选择按需标准、按需优势和预置模式。

- **按需标准**：按需模式的数据流无需容量规划，并且可以自动扩展以处理每分钟数 GB 的写入和读取吞吐量。若采用按需模式，Kinesis Data Streams 会自动管理分片来提供必要的吞吐量。
- **按需优势**：这是一种账户级模式，可提供更多功能，并简化按需流的定价体系。在此模式下，您可以在任何时间主动预热某个流的写入吞吐能力。对于定价，不再采用固定的流级计费模式，而且所有按需流的数据摄取、数据检索和延长留存使用量比按需标准流至少低 60%。
- **预置**：对于处于预置模式的数据流，您必须指定数据流的分片数量。数据流的总容量是其分片容量的总和。您可以根据需要增加或减少数据流中的分片数。

你可以使用 Kinesis Data PutRecord Streams PutRecords APIs 和在任何模式下将数据写入数据流。为了检索数据，所有三种模式都支持使用 GetRecords API 的默认消费端和使用 SubscribeToShard API 的增强型扇出 (EFO) 消费端。

按需模式和预置模式均支持 Kinesis Data Streams 的所有功能，包括保留模式、加密、监控指标等。Kinesis Data Streams 在按需和预置容量模式下都具有较高持久性和可用性。

按需标准模式的特征和用例

按需模式的数据流无需容量规划，并且可以自动扩展以处理每分钟数 GB 的写入和读取吞吐量。按需模式以低延迟的方式简化了摄取和存储大量数据的过程，因为它无需预置和管理服务器、存储或吞吐量。您每天可以摄取数十亿条记录，而不会产生任何运营开销。

按需模式非常适合用于应对高度可变且不可预测的应用程序流量的需求。您不再需要为峰值容量预置这些工作负载，因为峰值容量可能会因为利用率低而导致成本更高。按需模式适用于流量模式不可预测且高度可变的工作负载。

在按需容量模式下，您将按数据流中写入和读取的数据，为每 GB 数据付费。您无需指定预期应用程序执行的读写吞吐量。Kinesis Data Streams 会随着工作负载的增加或减少立即进行调整。有关更多信息，请参阅 [Amazon Kinesis Data Streams 定价](#)。

按需模式下的数据流最多可容纳过去 30 天内观察到的峰值写入吞吐量的两倍。当数据流的写入吞吐量达到新的峰值时，Kinesis Data Streams 会自动扩展数据流的容量。例如，如果您的数据流的写入吞吐量在写入吞吐量的 10 MB/s % 到 40MB/s, then Kinesis Data Streams ensures that you can easily burst to double your previous peak throughput, or 80 MB/s. If the same data stream sustains a new peak throughput of 50 MB/s, Kinesis Data Streams ensures that there is enough capacity to ingest 100 MB/s% 之间。但是，如果您的流量在 15 分钟内增加到前一个峰值的两倍以上，则可能会产生写入节流。您需要重试这些受限的请求。

按需模式下的数据流聚合读取容量与写入吞吐量成比例增加。这有助于确保消费端应用程序始终有足够的读取吞吐量来实时处理传入的数据。与使用 GetRecords API 读取数据相比，您获得的写入吞吐量至少是其两倍。我们建议您使用带有 GetRecord API 的消费端应用程序，这样当应用程序需要从停机时间中恢复时，有足够的空间来赶上。对于需要添加多个消费端应用程序的场景，建议您使用 Kinesis Data Streams 的增强型扇出功能。增强型扇出功能支持使用 SubscribeToShard API 将最多 20 个消费端应用程序添加到数据流中，每个消费端应用程序都有专用的吞吐量。

处理读写吞吐量异常

在按需模式（与预置容量模式相同）下，必须为每条记录指定一个分区键，才能将数据写入数据流。Kinesis Data Streams 使用分区键在分片之间分配数据。Kinesis Data Streams 监控每个分片的流量。当 KB/s 每个分片的传入流量超过 500 时，它会在 15 分钟内拆分分该分片。父分片的哈希键值在子分片之间均匀地重新分配。

如果您的传入流量超过之前峰值的两倍，即使您的数据在分片上均匀分配，您也可能会在约 15 分钟内遇到读取或写入异常的问题。我们建议您重试所有此类请求，以便将所有记录正确存储在 Kinesis Data Streams 中。

如果您使用的分区键会导致数据分配不均匀，并且分配给特定分片的记录超出了其限制，则可能会遇到读写异常的问题。在按需模式下，数据流会自动适应以处理不均匀的数据分配模式，除非单个分区键超过分片的 1 MB/s 吞吐量和每秒 1000 条记录的限制。

在按需模式下，Kinesis Data Streams 在检测到流量增加时会均匀地拆分分片。但是，它不会检测和隔离将更高比例的传入流量传送到特定分片的哈希键。如果您使用的分区键非常不均匀，则可能会继续收到写入异常的提示。对于此类使用案例，我们建议您使用支持精细分片拆分的预置容量模式。

按需优势模式的特征和用例

按需优势模式是一种账户级设置，可解锁更多功能，并为区域内的所有按需流提供不同的定价体系。在此模式下，按需流仍保留其功能，并根据数据实际使用量继续自动扩缩容量。如果想主动预热某个流的写入吞吐能力，可配置热吞吐量。例如，如果您的数据流的写入吞吐量介于 10 MB/s, you can

expect it to handle up to 80MB/s of instant throughput increases without throttling. However, if you forecast an upcoming event to peak around 200MB/s of traffic, you can configure the stream with a warm throughput of 200MB/s 到 40 MB/s 之间，则可以确保在数据吞吐量到达时容量可用。使用热吞吐量不会产生额外费用。

按需优势模式的另一个好处是按需流可以过渡至更简单的定价体系。启用该模式后，账户将不再产生固定的按流计费，您只需处理数据摄取、数据检索和可选的延长留存费用。与按需标准模式相比，每个定价维度都有很大的折扣。有关更多信息，请参阅 [Amazon Kinesis Data Streams 定价](#)。

在这种模式下，增强扇出型数据检索也不会产生相较于标准数据检索的价格溢价。此外，在按需优势模式下，您可以为每个直播注册多达 50 个消费者，以使用增强的扇出功能。启用 Ondemand Advantage 会使账户在所有按需流中至少有 25MiB/s of data ingest and 25MiB/s% 的数据检索。对于满足最低使用要求的账户，Kinesis Data Streams 控制台会检查账户的使用模式是否适合使用按需优势模式。

如果账户的数据使用量低于要求，则需要对差额付费，但折扣率仍将保持不变。启用按需优势模式之后，至少还需要 24 小时才能禁用此模式。总体而言，如果持续使用的吞吐量接近或超过最低承诺值、需要大量扇出型消费端或使用数百个数据流，则按需优势模式即为使用 Kinesis Data Streams 进行流式传输的最佳方案。

预置模式的特征和用例

在预配置模式下，在创建数据流之后，您可以使用或 API 动态地向上或向下扩展分片容量。Amazon Web Services 管理控制台 [UpdateShardCount](#) 您可以在 Kinesis Data Streams 创建器或消费端应用程序在向流写入数据或从中读取数据时进行更新。

预置模式适用于容量需求易于预测的可预测流量。如果您想精细控制分片间数据的分配方式，则可以使用预置模式。

若采用预置模式，您必须为数据流指定分片数。要确定预置模式下数据流的大小，您需要以下输入值：

- 写入流的数据记录的平均大小，以 KB 为单位，四舍五入为 1 KB (average_data_size_in_KB)。
- 每秒写入流和从流读取的数据记录数 (records_per_second)。
- 消费端数量，即并发且独立使用流中数据的 Kinesis Data Streams 应用程序的数量 (number_of_consumers)。
- 以 KB 为单位的传入写入带宽 (incoming_write_bandwidth_in_KB)，等于 average_data_size_in_KB 乘以 records_per_second。
- 以 KB 为单位的传出读取带宽 (outgoing_read_bandwidth_in_KB)，等于 incoming_write_bandwidth_in_KB 乘以 number_of_consumers。

可使用以下公式中的输入值来计算流所需的分片数量 (number_of_shards)。

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

如果您未将数据流配置为处理峰值吞吐量，则在预置模式下仍可能遇到读写吞吐量异常的问题。在这种情况下，您必须手动扩展数据流以适应数据流量。

如果您使用的分区键会导致数据分配不均匀，并且分配给特定分片的记录超出了其限制，则可能也会遇到读写异常的问题。要在预置模式下解决此问题，请确定此类分片并手动拆分，以更好地适应流量。有关更多信息，请参阅[对流进行重新分片](#)。

切换模式

对于中的每个数据流Amazon Web Services 账户，您可以在 24 小时内在按需模式和预配置模式之间切换两次。切换模式不会对使用该数据流的应用程序造成任何中断。您可以继续读写该数据流。当您在各模式之间切换（从按需切换到预置，或反之）时，流的状态将设置为正在更新。必须等待数据流状态变为活动，然后才能再次修改其属性。

当您从预置容量模式切换到按需容量模式时，数据流最初会保留转换之前的所有分片数量，从此时起，Kinesis Data Streams 将监控您的数据流量，并根据写入吞吐量扩展此按需数据流的分片数量。当您从按需模式切换到预置模式时，数据流最初也会保留转换之前的所有分片数量，但是从此时起，您将负责监控和调整此数据流的分片数量，以正确适应写入吞吐量。

可以通过启用账户级设置，从按需标准模式切换到按需优势模式。启用后，该账户承诺在该地区所有按需流中至少 25MiB/s of data ingest and 25MiB/s% 的数据检索使用量。启用后，至少须等待 24 小时才能禁用按需优势模式，但您可以随时申请更改。如果要从按需优势模式切换到按需标准模式，则必须先移除任何配置了按需流的热吞吐量。

使用创建直播 Amazon Web Services 管理控制台

可以使用 Kinesis Data Streams 控制台、Kinesis Data Streams API 或 Amazon Command Line Interface (Amazon CLI) 创建流。

使用控制台创建数据流

1. [登录Amazon Web Services 管理控制台并在 /kinesis 上打开 Kinesis 控制台。<https://console.aws.amazon.com/kinesis/>](https://console.aws.amazon.com/kinesis/)
2. 在导航栏中，展开区域选择器并选择一个区域。

3. 选择创建数据流。
4. 在创建 Kinesis 流页面上，输入数据流的名称，然后选择按需或预置容量模式。默认情况下，系统将选择按需模式。有关更多信息，请参阅 [选择正确的流式传入的模式](#)。

在按需模式下，您可以选择创建 Kinesis 流来创建数据流。在预置模式下，您必须指定所需的分片数量，然后选择创建 Kinesis 流。

在 Kinesis stream (Kinesis 流)页面上，当流处于创建中时，流的 Status (状态) 为 Creating (正在创建)。当流可以使用时，Status (状态) 会更改为 Active (有效)。

5. 选择流的名称。Stream Details (流详细信息) 页面显示了流配置摘要以及监控信息。

使用 Kinesis Data Streams API 创建流

- 有关使用 Kinesis Data Streams API 创建流的信息，请参阅 [使用 APIs 创建流](#)。

要使用创建直播 Amazon CLI

- 有关使用创建直播的信息，请参阅 `create-stream` 命令。Amazon CLI

使用 APIs 创建流

请使用以下步骤创建 Kinesis 数据流。

构建 Kinesis Data Streams 客户端

必须先构建客户端对象，然后才能处理 Kinesis 数据流。以下 Java 代码实例化一个客户端生成器，并使用它来设置区域、凭据和客户端配置。然后，它会构建一个客户端对象。

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

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

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

有关更多信息，请参阅《Amazon Web Services 一般参考》中的 [Kinesis Data Streams Regions and Endpoints](#)。

创建流

现在您已创建 Kinesis Data Streams 客户端，接下来可使用控制台或以编程方式来创建流。要以编程方式创建流，请实例化 `CreateStreamRequest` 对象并指定流名称。若要使用预置模式，请为数据流指定分片数量。

- 按需：

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

- 预置：

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

流名称用于标识流。该名称的作用域仅限于应用程序使用的Amazon帐户。它还受区域限制。也就是说，两个不同Amazon账户中的两个直播可以有相同的名称，同一个Amazon账户中但位于两个不同区域的两个直播可以有相同的名称，但不能有两个直播在同一个账户和同一个区域中。

流的吞吐量是分片数量的函数。要获得更高的预置吞吐量，您需要更多的分片。分片越多还会增加对直播Amazon收取的费用。有关计算适合您的应用程序的分片数量的更多信息，请参阅[选择正确的流式传入的模式](#)。

配置 `createStreamRequest` 对象后，应通过对客户端调用 `createStream` 方法来创建流。在调用 `createStream` 之后，应等待流达到 ACTIVE 状态，然后再对流执行任何操作。要查看流的状态，请调用 `describeStream` 方法。但是，如果流不存在，`describeStream` 将引发异常。因此，请将 `describeStream` 调用包括在 `try/catch` 块中。

```
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" );
}
```

更新流

可以使用 Kinesis Data Streams 控制台、Kinesis Data Streams API 或 Amazon CLI 更新流详细信息。

 Note

您可以为现有流或最近刚创建的流启用服务器端加密。

使用控制台

使用控制台更新数据流

1. 打开亚马逊 Kinesis 控制台，网址为。<https://console.aws.amazon.com/kinesis/>
2. 在导航栏中，展开区域选择器并选择一个区域。
3. 在列表中选择流的名称。Stream Details (流详细信息) 页面显示流配置摘要和监控信息。

- 要在数据流的按需容量模式和预置容量模式之间切换，请在配置选项卡中选择编辑容量模式。有关更多信息，请参阅 [选择正确的流式传入的模式](#)。

 **Important**

对于您Amazon账户中的每个数据流，您可以在 24 小时内在按需模式和预配置模式之间切换两次。

- 对于处于预置模式下的数据流，要编辑分片数量，请在配置选项卡中选择编辑预置分片，然后输入新的分片数。
- 要对数据记录启用服务器端加密，请选择服务器端加密部分中的编辑。选择要用作加密主密钥的 KMS 密钥，或者使用由 Kinesis 管理的默认主密钥 aws/kinesis。如果您为直播启用加密并使用自己的Amazon KMS主密钥，请确保您的制作者和使用者应用程序可以访问您使用的Amazon KMS主密钥。要将权限分配给应用程序以访问用户生成的 Amazon KMS 密钥，请参阅[the section called “使用用户生成的 KMS 密钥的权限”](#)。
- 要编辑数据保留期，请选择 Data retention period 部分中的 Edit，然后输入新的数据保留期。
- 如果您已在自己账户上启用自定义指标，请在分片级别指标部分中选择编辑，然后指定流的指标。有关更多信息，请参阅 [the section called “使用 CloudWatch 监控 Kinesis Data Streams 服务”](#)。

使用 API

要使用 API 更新流详细信息，请参阅以下方法：

- [AddTagsToStream](#)
- [DecreaseStreamRetentionPeriod](#)
- [DisableEnhancedMonitoring](#)
- [EnableEnhancedMonitoring](#)
- [IncreaseStreamRetentionPeriod](#)
- [RemoveTagsFromStream](#)
- [StartStreamEncryption](#)
- [StopStreamEncryption](#)
- [UpdateShardCount](#)

使用 Amazon CLI

有关使用更新直播的信息Amazon CLI，请参阅 [Kinesis CLI](#) 参考。

列出流

流的范围限定为与用于实例化 Kinesis Data Streams 客户端的Amazon凭据关联的Amazon账户以及为该客户端指定的区域。一个 Amazon 账户可以同时具有多个活动流。您可在 Kinesis Data Streams 控制台中列出流，也可以编程方式列出流。本节中的代码显示了如何列出您Amazon账户的所有直播。

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

此代码示例首先创建 `ListStreamsRequest` 的一个新实例，然后调用其 `setLimit` 方法来指定对 `listStreams` 的每次调用应返回最多 20 个流。如果您没有为 `setLimit` 指定值，则 Kinesis Data Streams 将返回的流数量小于或等于账户中的流数量。此代码之后将 `listStreamsRequest` 传递到客户端的 `listStreams` 方法。返回值 `listStreams` 存储在 `ListStreamsResult` 对象中。此代码对此对象调用 `getStreamNames` 方法，并将返回的流名称存储在 `streamNames` 列表中。请注意，Kinesis Data Streams 返回的流数量可能少于指定限制值所指定的数量，即使账户和区域中的流数量多于此数量也是如此。要确保检索所有流，请使用下一代码示例中描述的 `getHasMoreStreams` 方法。

```
while (listStreamsResult.getHasMoreStreams())
{
    if (streamNames.size() > 0) {
        listStreamsRequest.setExclusiveStartStreamName(streamNames.get(streamNames.size() - 1));
    }
    listStreamsResult = client.listStreams(listStreamsRequest);
    streamNames.addAll(listStreamsResult.getStreamNames());
}
```

此代码对 `getHasMoreStreams` 调用 `listStreamsRequest` 方法，以检查是否有超出在对 `listStreams` 的初始调用中返回的流数量的可用流。如果有，则此代码将使用在对 `setExclusiveStartStreamName` 的上一调用中返回的最后一个流的名称调用 `listStreams` 方法。`setExclusiveStartStreamName` 方法将导致对 `listStreams` 的下一调用在该流之后开始。

该调用返回的一组流名称之后将添加到 `streamNames` 列表。此过程将继续，直到所有流名称已收集到列表中。

`listStreams` 返回的流可能处于下列状态之一：

- CREATING
- ACTIVE
- UPDATING
- DELETING

您可使用 `describeStream` 方法查看流的状态（如上一节 [使用 APIs 创建流](#) 中所示）。

列出分片

一个数据流可以有一个或多个分片。从数据流中列出或检索分片的推荐方法是使用 API。[ListShards](#) 以下示例说明如何获取数据流中的分片列表。有关本示例中使用的主操作的完整说明以及您可以为该操作设置的所有参数，请参阅[ListShards](#)。

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

import java.util.concurrent.TimeUnit;

public class ShardSample {

    public static void main(String[] args) {

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

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

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

```
        System.out.println(e.getMessage());
    }
}
}
```

要运行上一个代码示例，您可以使用类似于下文的 POM 文件。

```
<?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>
```

通过 `ListShards` API，您可以使用 [ShardFilter](#) 参数筛选出 API 的响应。一次只能指定一个筛选条件。

如果您在调用 ListShards API 时使用 ShardFilter 参数，则 Type 为必填属性，必须指定。如果指定 AT_TRIM_HORIZON、FROM_TRIM_HORIZON 或 AT_LATEST 类型，则无需指定 ShardId 或 Timestamp 可选属性。

如果指定 AFTER_SHARD_ID 类型，则还必须提供可选 ShardId 属性的值。该 ShardId 属性的功能与 ListShards API 的 ExclusiveStartShardId 参数相同。指定 ShardId 属性后，响应将包括多个分片，其中开头的分片 ID 紧随您提供的 ShardId。

如果指定 AT_TIMESTAMP 或 FROM_TIMESTAMP_ID 类型，则还必须提供可选 Timestamp 属性的值。如果指定 AT_TIMESTAMP 类型，则返回在提供的时间戳上打开的所有分片。如果指定 FROM_TIMESTAMP 类型，则返回从提供的时间戳开始到 TIP 的所有分片。

Important

DescribeStreamSummary 并 ListShard APIs 提供一种更具扩展性的方式来检索有关您的数据流的信息。更具体地说，DescribeStream API 的配额可能会导致限制。有关更多信息，请参阅 [限额和限制](#)。另请注意，与您 Amazon 账户中所有数据流交互的所有应用程序均共享 DescribeStream 配额。另一方面，ListShards API 的配额特定于单个数据流。因此，使用 ListShards API 不仅可以获得更高的 TPS，而且随着您创建更多数据流，操作可以更好地扩展。

我们建议您迁移所有调用 DescribeStream API 的生产者和使用者，改为调用 DescribeStreamSummary 和 ListShard APIs。为了识别这些生产者和使用者，我们建议使用 Athena 来 CloudTrail 解析日志，因为在 API 调用中捕获 KPL 和 KCL 的用户代理。

```
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
```

我们还建议重新配置调用 API 的 Amazon Lambda 和 Amazon Firehose 与 Kinesis Data Streams 的集成，以便集成改为调用 DescribeStream 和 DescribeStreamSummary。

ListShards具体而言，对于 Amazon Lambda，您必须更新您的事件源映射。对于 Amazon Firehose，必须更新相应的 IAM 权限，使其包含 ListShards IAM 权限。

删除流

您可使用 Kinesis Data Streams 控制台或以编程方式删除流。要以编程方式删除流，请使用 DeleteStreamRequest，如以下代码所示。

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

删除流之前，请关闭流上运行的任何应用程序。如果应用程序尝试对已删除的流进行操作，则将收到 ResourceNotFound 异常。此外，如果您随后创建名称与之前的流相同的新流，并且在之前的流上运行的应用程序仍在运行，则这些应用程序可能会尝试与新流交互（就像新流是之前的流一样），这将产生无法预测的结果。

对流进行重新分片

Important

您可以使用 API 对直播进行重新分片。[UpdateShardCount](#)否则，您可以按照此处的说明继续执行拆分和合并。

Amazon Kinesis Data Streams 支持重新分片，这使您能够调整流中的分片数量以适应流中数据流的速率变化。重新分片被视为高级操作。如果您不熟悉 Kinesis Data Streams，请在熟悉 Kinesis Data Streams 的所有其他方面之后再回来阅读本主题。

这里有两种类型的重新分片操作：分片拆分和分片合并。在分片拆分中，可将一个分片拆分为两个分片。在分片合并中，可将两个分片组合成一个分片。重新分片始终是成对进行的，也就是说，无法在一次操作中拆分为两个以上的分片，并且无法在一次操作中合并两个以上的分片。重新分片操作涉及的分片或分片对称为父分片。从重新分片操作中生成的分片或分片对称为子分片。

拆分将增加流中分片的数量，从而增加流的数据容量。由于按分片收费，因此拆分将增加流的费用。同样，合并会减少流中的分片数量，从而降低流的数据容量和成本。

重新分片一般由不同于创建者（放置）应用程序和消费端（获取）应用程序的管理应用程序执行。这样的管理应用程序根据Amazon提供的指标 CloudWatch 或从制作者和消费者那里收集的指标来监控直播的整体性能。与使用者或创建者相比，管理应用程序还需要更广泛的 IAM 权限，因为使用者和创建者通常不需要访问 APIs 用于重新分片的用户。有关 Kinesis Data Streams 的 IAM 权限的更多信息，请参阅 [使用 IAM 控制对 Amazon Kinesis Data Streams 资源的访问](#)。

有关重新分片的更多信息，请参阅[如何更改 Kinesis Data Streams 中打开的分片数](#)。

主题

- [确定重新分片策略](#)
- [拆分分片](#)
- [合并两个分片](#)
- [完成重新分片操作](#)

确定重新分片策略

在 Amazon Kinesis Data Streams 中重新分片的目的是，使流能够适应数据流的速率的变化。拆分分片将增加流的容量（和费用）。合并分片将减少流的费用（和容量）。

一种重新分片的方式可能是拆分流中的每个分片，这将使流的容量增加一倍。但是，这提供的容量可能比您实际需要的要多，从而产生不必要的费用。

您还可使用指标确定您的热或冷分片（即，接收的数据多于预期或少于预期的分片）。之后您可选择性地拆分热分片以增加面向这些分片的哈希键的容量。同样，您可合并冷分片以更好地利用其未使用的容量。

您可以从 Kinesis Data Streams 发布的亚马逊 CloudWatch 指标中获取直播的一些性能数据。但是，您也可收集您自己的一些关于流的指标。一种方式是记录由您的数据记录的分区键生成的哈希键值。记住，您在向流添加记录时指定了分区键。

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

Kinesis Data Streams [MD5](#)使用分区键计算哈希键。由于您为记录指定了分区键，因此您可以使用 MD5 计算该记录的哈希键值并将其记录下来。

您也可以记录分配给您的数据记录的分片。IDs 分片 ID 是通过使用 `getShardId` 对象（由 `putRecordResults` 方法返回）和对 `putRecords` 对象（由 `putRecordResult` 方法返回）的 `putRecord` 方法提供的。

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

使用分片 IDs 和哈希键值，您可以确定哪些分片和哈希键接收的流量最多或最少。您之后可使用重新分片操作来增加或减少容量（视这些键的情况而定）。

拆分分片

要在 Amazon Kinesis Data Streams 中拆分分片，需要指定父分片中的哈希键值应如何重新分配给子分片。当您向流添加数据记录时，将基于哈希键值将数据记录分配给分片。哈希键值是您在向流中添加数据记录时为数据记录指定的分区键的[MD5](#)哈希值。具有相同分区键的数据记录也具有相同的哈希键值。

指定分片的可能的哈希键值构成一组有序的连续非负整数。此可能的哈希键值范围是通过以下命令指定的：

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

在拆分分片时，您指定此范围内的一个值。此哈希键值和所有较高的哈希键值将分配到其中一个子分片。所有较小的哈希键值将分配到另一子分片。

以下代码演示在所有子分片之间均匀地重新分配哈希键的分片拆分操作，基本上是将父分片一分为二。这只是一个可能的父分片划分方式。例如，可以拆分分片，以便父分片中下层三分之一的键分配给一个子分片，上层三分之二的键分配给另一子分片。但是，在许多应用程序中，将分片一分为二是一种很有效的方法。

此代码假定 `myStreamName` 包含您的流名称，对象变量 `shard` 包含要拆分的分片。首先实例化一个新的 `splitShardRequest` 对象并设置流名称和分片 ID。

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

确定位于分片中最低值和最高值的中间的哈希键值。这是将包含父分片中上半层哈希键的子分片的起始哈希键值。在 `setNewStartingHashKey` 方法中指定此值。您只需要指定此值。Kinesis Data Streams 会自动将低于此值的哈希键分配给拆分操作所创建的另一子分片。最后一步是对 Kinesis Data Streams 客户端调用 `splitShard` 方法。

```
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);
```

[等待流再次变为活动状态](#)中演示了此过程之后的第一步。

合并两个分片

分片合并操作使用两个指定分片并将它们合并为一个分片。在合并后，单个子分片将收到两个父分片所包含的所有哈希键值的数据。

分片相邻

要合并两个分片，分片必须相邻。如果两个分片的哈希键范围联合构成一个无间断的连续集，则认为两个分片相邻。例如，假设您有两个分片，一个分片的哈希键范围为 276...381，另一个分片的哈希键范围为 382...454。您可以将这两个分片合并为一个分片，其哈希键范围为 276...454。

另举一例，假设您有两个分片，一个分片的哈希键范围为 276...381，另一个分片的哈希键范围为 455...560。您无法合并这两个分片，因为这两个分片之间有一个或多个哈希键位于 382...454 范围的分片。

流中所有OPEN分片的集合（作为一个组）始终跨越整个哈希键值范围。MD5 有关分片状态（例如 CLOSED）的更多信息，请参阅 [重新分片之后考虑数据路由、数据保留和分片状态](#)。

要标识作为合并候选的分片，您应筛选出处于 CLOSED 状态的所有分片。状态为 OPEN（也就是非 CLOSED）的分片的结尾序列号为 null。您可使用以下命令测试此结束序列号：

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

在筛选出已关闭分片后，按每个分片支持的最高哈希键值对剩余分片进行排序。您可使用以下命令检索此值：

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

如果两个分片在这个经过筛选和排序的列表中相邻，则可合并它们。

合并操作的代码

以下代码可合并两个分片。此代码假定 `myStreamName` 包含您的流名称并且对象变量 `shard1` 和 `shard2` 包含要合并的两个相邻分片。

为进行合并操作，首先实例化一个新的 `mergeShardsRequest` 对象。使用 `setStreamName` 方法指定流名称。然后使用 `setShardToMerge` 和 `setAdjacentShardToMerge` 方法指定要合并的两个分片。最后，对 Kinesis Data Streams 客户端调用 `mergeShards` 方法以执行此操作。

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

等待流再次变为活动状态 中演示了此过程之后的第一步。

完成重新分片操作

在 Amazon Kinesis Data Streams 中执行任何类型的重新分片过程之后，在继续常规记录处理之前，需要执行其他流程并注意一些事项。以下各节介绍了这些过程。

主题

- [等待流再次变为活动状态](#)
- [重新分片之后考虑数据路由、数据保留和分片状态](#)

等待流再次变为活动状态

在调用重新分片操作 `splitShard` 或 `mergeShards` 之后，您必须等待流再次变为活动状态。要使用的代码与您在 [创建流](#) 之后等待流变为活动状态时使用的代码相同。代码如下所示：

```
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" );
}
```

重新分片之后考虑数据路由、数据保留和分片状态

Kinesis Data Streams 是一项实时数据流服务。您的应用程序应假定数据不断地在流中的分片内流动。当您重新分片时，流至父分片的数据记录将基于数据记录分区键映射到的哈希键值重新路由至子分片。但是，重新分片前位于父分片中的任何数据记录将仍位于这些父分片中。父分片在重新分片发生后不会消失。它们与重新分片之前包含的数据一起保留。可以使用 Kinesis Data Streams API 中的 [getShardIterator](#) 和 [getRecords](#) 操作或通过 Kinesis Client Library 访问父分片中的数据记录。

Note

数据记录在当前保留期内添加到流中后即可访问。不论在该期间内对流中的分片进行了任何更改，都是如此。有关流的保留期的更多信息，请参阅[更改数据留存期](#)。

在重新分片过程中，父分片将从 OPEN 状态过渡到 CLOSED 状态再过渡到 EXPIRED 状态。

- OPEN：在重新分片操作之前，父分片处于 OPEN 状态，这意味着数据记录可添加到分片中并且可从分片进行检索。
- CLOSED：在重新分片操作之后，父分片将过渡到 CLOSED 状态。这意味着无法再向此分片添加数据记录。原本应该已添加到此分片的数据记录现在将改为添加到子分片。但是，数据记录在有限时间内仍可从此分片进行检索。
- EXPIRED：在流的保留期过期之后，父分片中的所有数据记录将会过期，不再可供访问。此时，父分片自身将过渡到 EXPIRED 状态。用于枚举流中的分片的 `getStreamDescription().getShards` 调用不包括返回的分片列表中的 EXPIRED 分片。有关流的保留期的更多信息，请参阅[更改数据留存期](#)。

在进行重新分片并且流再次处于 ACTIVE 状态之后，您可立即开始读取子分片中的数据。但是，在重新分片后保留的父分片可能仍包含您尚未读取并且已在重新分片前添加到流中的数据。如果您在读取完父分片中的所有数据之前读取子分片中的数据，则可不按数据记录的序列号指定的顺序读取特定哈希键的数据。因此，假定数据顺序很重要，您在重新分片后应始终继续读取父分片中的数据直到读取完。并且只有在这之后才应开始读取子分片中的数据。如果 `getRecordsResult.getNextShardIterator` 返回 `null`，则这指示您已读取完父分片中的所有数据。

更改数据留存期

Amazon Kinesis Data Streams 支持更改数据流的数据记录保留期。Kinesis 数据流是数据记录的有序序列，可用于执行实时写入和读取。因此，数据记录临时存储在您的流的分片中。从添加记录开始，到记录不再可供访问为止的时间段称为保留期。默认情况下，Kinesis 数据流的记录存储时间从 24 小时到 8760 小时（365 天）不等。

您可以通过 Kinesis Data Streams 控制台或使

用[IncreaseStreamRetentionPeriod](#)和[DecreaseStreamRetentionPeriod](#)操作来更新保留期。

利用 Kinesis Data Streams 控制台，您可以同时批量编辑多个数据流的保留期。您可以使用[IncreaseStreamRetentionPeriod](#)操作或 Kinesis Data Streams 控制台将保留期延长至最长 8760 小时（365 天）。您可以使用[DecreaseStreamRetentionPeriod](#)操作或 Kinesis Data Streams 控制台将保留期缩短至至少 24 小时。两个操作的请求语法均包括流名称和保留期（以小时为单位）。最后，您可以通过调用[DescribeStream](#) 操作来检查流的当前保留期。

以下是使用 Amazon CLI 更改保留期的示例：

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

在增加保留期后的数分钟内，Kinesis Data Streams 会开放对处于先前保留期内的记录的访问权限。例如，将保留期从 24 小时更改为 48 小时意味着，在 23 小时 55 分钟之前添加到流中的记录在 24 小时之后仍可用。

在缩短保留期后，Kinesis Data Streams 几乎会立即使比新保留期更早的记录不可供访问。因此，在调用 [DecreaseStreamRetentionPeriod](#) 操作时务必小心谨慎。

设置数据保留期以确保在出现问题时，您的消费端可以在数据过期之前读取数据。您应该仔细考虑所有可能的情况，例如记录处理逻辑出现问题，或者下游依赖关系长时间断开。将保留期视为安全网，可留出更多时间供您的数据消费端恢复。使用保留期 API 操作，您可以主动设置此项，或者积极响应操作事件。

对于保留期设置为 24 小时以上的流，将收取额外费用。有关更多信息，请参阅 [Amazon Kinesis Data Streams 定价](#)。

标记 Amazon Kinesis Data Streams 资源

您可以将自己的元数据以标签的形式分配给您在 Amazon Kinesis Data Streams 中创建的流和增强扇出型消费端。标签是您为流定义的键值对。使用标签是一种管理 Amazon 资源和整理数据（包括账单数据）的简单而强大的方法。

内容

- [查看标签基本知识](#)
- [使用标签跟踪成本](#)
- [了解标签限制](#)
- [使用 Kinesis Data Streams 控制台标记流](#)
- [使用标记直播 Amazon CLI](#)
- [使用 Kinesis Data Streams 标记直播 APIs](#)
- [使用标记消费者 Amazon CLI](#)
- [使用 Kinesis Data Streams 标记消费者 APIs](#)

查看标签基本知识

可标记的 Kinesis Data Streams 资源包括数据流和增强型扇出消费端。你可以使用 Kinesis Data Streams 控制台 Amazon CLI 或 Kinesis Data Streams API 来完成以下任务：

- [使用标签创建资源](#)

- 向资源添加标签
- 列出资源的标签
- 从资源中删除标签

 Note

使用 Kinesis Data Streams 控制台无法将标签应用于增强型扇出消费端。要将标签应用于使用者，请使用 Amazon CLI 或 Kinesis Data Streams API。

您可以使用标签对 资源进行分类。例如，您可以按用途、所有者或环境对资源进行分类。由于您定义每个标签的键和值，因此您可以创建一组自定义类别来满足您的特定需求。例如，您可以定义一组标签来帮助您按拥有者和关联应用程序跟踪资源。以下几个标签示例：

- 项目：项目名称
- 所有者：名称
- 用途：负载测试
- 应用程序：应用程序名称
- 环境：生产

 Important

- 要在创建流时添加标签，必须为流添加 `kinesis:CreateStream` 和 `kinesis:AddTagsToStream` 权限。在创建流时，不能使用 `kinesis:TagResource` 权限来标记流。
- 要在消费端注册期间添加标签，就必须添加 `kinesis:TagResource` 和 `kinesis:RegisterStreamConsumer` 权限。

使用标签跟踪成本

您可以使用标签对 Amazon 费用进行分类和跟踪。当您对 Kinesis Data Streams 资源应用标签时，Amazon 您的成本分配报告包括按标签汇总的使用量和成本。您可以设置代表业务类别（例如成本中心、应用程序名称或所有者）的标签，以便整理多种服务的成本。有关更多信息，请参阅 Amazon Billing 用户指南中的[对自定义账单报告使用成本分配标签](#)。

了解标签限制

以下限制适用于标签：

基本限制

- 每个资源的最大标签数是 50。
- 标签键和值区分大小写。
- 无法更改或编辑已删除的资源的标签。

标签键限制

- 每个标签键必须是唯一的。如果您添加的标签具有已使用的键，则您的新标签将覆盖现有键-值对。
- 标签密钥不能以开头，aws：因为此前缀已保留供使用 Amazon。Amazon 代表您创建以此前缀开头的标签，但您无法对其进行编辑或删除。
- 标签键的长度必须介于 1 和 128 个 Unicode 字符之间。
- 标签键必须包含以下字符：Unicode 字母、数字、空格和以下特殊字符：_ . / = + - @。

标签值限制

- 标签值的长度必须介于 0 和 255 个 Unicode 字符之间。
- 标签值可以为空。另外，它们必须包含以下字符：Unicode 字母、数字、空格和以下任意特殊字符：_ . / = + - @。

使用 Kinesis Data Streams 控制台标记流

您可以使用 Kinesis Data Streams 控制台在流上添加、更新、列出和移除标签。

查看流的标签

1. [登录 Amazon Web Services 管理控制台 并在 /kinesis 上打开 Kinesis 控制台。https://console.aws.amazon.com/kinesis](https://console.aws.amazon.com/kinesis)
2. 在左侧导航窗格中，选择数据流。
3. 在数据流页面上，选择您要标记的流。
4. 在流详细信息页面上，选择配置。

5. 在标签部分，查看应用于流的标签。

创建带有标签的数据流

1. 打开 Kinesis Data Streams 控制台。
2. 在左侧导航窗格中，选择数据流。
3. 选择创建数据流。
4. 在创建数据流页面上，输入数据流的名称。
5. 对于数据流容量，选择按需或预置容量模式。

有关容量模式的更多信息，请参阅[选择正确的流式传入的模式](#)。

6. 请在标签部分执行以下操作：
 - a. 选择添加新标签。
 - b. 在键中输入标签，然后也可以选择在值字段中指定值。

如果出现错误，则您指定的标签键或值不满足标签限制。有关更多信息，请参阅[了解标签限制](#)。

7. 选择创建数据流。

在流中添加或更新标签

1. 打开 Kinesis Data Streams 控制台。
2. 在左侧导航窗格中，选择数据流。
3. 在数据流页面上，选择要向其添加或更新标签的流。
4. 在流详细信息页面上，选择配置。
5. 在标签部分中，选择管理标签。
6. 在标签下，执行以下操作之一：
 - 要添加标签，请选择添加新标签，然后输入标签的键和值数据。将该步骤重复执行所需的次数。

每个流可以添加的最大标签数量为 50 个。

- 要更新现有标签，请在该标签的键的值字段中输入新的标签值。

如果出现错误，则您指定的标签键或值不满足标签限制。有关更多信息，请参阅[了解标签限制](#)。

7. 选择保存更改。

从流中删除标签

1. 打开 Kinesis Data Streams 控制台。
2. 在左侧导航窗格中，选择数据流。
3. 在数据流页面上，选择要从中移除标签的流。
4. 在流详细信息页面上，选择配置。
5. 在标签部分中，选择管理标签。
6. 查找要移除的标签键和值对。然后，选择移除。
7. 选择保存更改。

使用标记直播 Amazon CLI

您可以使用 Amazon CLI 在流上添加、列出和移除标签。有关示例，请参阅以下文档。

[create-stream](#)

创建带有标签的流。

[add-tags-to-stream](#)

为指定的流添加或更新标签。

[list-tags-for-stream](#)

列出指定流的标签。

[remove-tags-from-stream](#)

从指定的流中删除标签。

使用 Kinesis Data Streams 标记直播 APIs

你可以使用 Kinesis Data APIs a Streams 在直播中添加、列出和删除标签。有关示例，请参阅以下文档：

[CreateStream](#)

创建带有标签的流。

[AddTagsToStream](#)

为指定的流添加或更新标签。

[ListTagsForStream](#)

列出指定流的标签。

[RemoveTagsFromStream](#)

从指定的流中删除标签。

使用标记消费者 Amazon CLI

您可以使用 Amazon CLI 在消费端上添加、列出和移除标签。有关示例，请参阅以下文档：

[register-stream-consumer](#)

在带有标签的 Kinesis 数据流中注册消费端。

[tag-resource](#)

为指定的 Kinesis 资源添加或更新标签。

[list-tags-for-resource](#)

列出指定 Kinesis 资源的标签。

[untag-resource](#)

从指定的 Kinesis 资源中移除标签。

使用 Kinesis Data Streams 标记消费者 APIs

您可以使用 Kinesis Data APIs a Streams 在使用者身上添加、列出和删除标签。有关示例，请参阅以下文档：

[RegisterStreamConsumer](#)

在带有标签的 Kinesis 数据流中注册消费端。

[TagResource](#)

为指定的 Kinesis 资源添加或更新标签。

[ListTagsForResource](#)

列出指定 Kinesis 资源的标签。

[UntagResource](#)

从指定的 Kinesis 资源中移除标签。

处理大型记录

Amazon Kinesis Data Streams 支持最大 10 兆MiBs字节 () 的记录。建议使用此功能来处理超过 1 MiB 默认记录大小限制的间歇性数据负载。现有和新建流的默认最大记录大小设置为 1 MiB。

该功能有利于物联网 (IoT) 应用程序、更改数据捕获 (CDC) 管道以及偶尔需要处理较大数据负载的机器学习工作流程。要开始在流中使用大型记录，请更新流的最大记录大小限制。

Important

写入的单个分片吞吐量限制 MB/s 为 1，读取 2 MB/s 的吞吐量限制保持不变，同时支持更大的记录大小。Kinesis Data Streams 旨在容纳间歇性的大型记录，以及小于或等于 1 MiB 的记录基准流量。但并非为适应持续的大容量大型记录摄取而设计。

更新您的流以使用大型记录

使用 Kinesis Data Streams 处理较大记录

1. 导航到 Kinesis Data Streams 控制台。
2. 选择您的流，然后前往配置选项卡。
3. 单击最大记录大小旁边的编辑。
4. 设置最大记录大小（最大值为 10 MiB）。
5. 保存更改。

此设置仅调整该 Kinesis 数据流的最大记录大小。在添加该限制之前，请验证所有下游应用程序是否能处理较大的记录。

您也可以使用 Amazon CLI 更新此设置：

```
aws kinesis update-max-record-size \ --stream-arn \
```

```
--max-record-size-in-ki-b 5000
```

使用大型记录优化流性能

建议将大型记录保持在总流量的 2% 以下。流中每个分片的吞吐能力为每秒 1 MiB。为了容纳大型记录，Kinesis 数据流最多可爆发 10 个 MiBs，而平均每秒 1 MiB。这种支持大型记录的能力会持续注入流中。注入速度取决于大型记录的大小和基准记录的大小。为获得最佳效果，请使用均匀分布的分区键。有关 Kinesis 如何按需扩展的更多信息，请参阅[按需模式功能和用例](#)。

使用大型记录缓解限制

缓解限制

1. 在创建者应用程序中实现带有指数级退避的重试逻辑。
2. 使用随机的分区键在可用分片之间分配大型记录。
3. 对于连续的大型记录流，将有效负载存储在 Amazon S3 中，仅向流发送元数据引用。有关更多信息，请参阅[使用 Amazon Kinesis Data Streams 处理大型记录](#)。

使用 Kinesis Data Streams 处理大型记录 APIs

大型记录支持引入了一个新的 API，并更新了两个现有的控制平面 APIs 以处理最多 10 个记录 MiBs。

用于修改记录大小的 API：

- `UpdateMaxRecordSize`：为现有直播配置最大记录大小限制，最多 10 MiBs。

对现有版本的更新 APIs：

- `CreateStream`：添加可选 `MaxRecordSizeInKiB` 参数，用于在创建流期间设置记录大小限制。
- `DescribeStreamSummary`：返回 `MaxRecordSizeInKiB` 字段显示当前的流配置。

所有 APIs 列出的内容都保持了对现有直播的向后兼容性。有关完整的 API 文档，请参阅[Amazon Kinesis Data Streams 服务 API 参考](#)。

Amazon 与大型唱片兼容的组件

以下 Amazon 组件与大型唱片兼容：

| 组件 | 描述 |
|--------------------------------|--|
| Amazon SDK | Amazon SDK 支持处理大型记录。您可以使用中的可用方法更新直播的最大记录大小 (最多 10 MiB)。Amazon SDKs 有关更多信息，请参阅 将此服务与 Amazon SDK 配合使用 。 |
| Kinesis Consumer Library (KCL) | KCL 从 2.x 版开始支持处理大型记录。要使用大型录制支持，请更新流的 <code>maxRecordSize</code> 并使用 KCL。有关更多信息，请参阅 使用 Kinesis Client Library 。 |
| Kinesis Producer Library (KPL) | KPL 从 1.0.5 版开始支持处理大型记录。要使用大型录制支持，请更新流的 <code>maxRecordSize</code> 并使用 KPL。有关更多信息，请参阅 使用 Amazon Kinesis Producer Library (KPL) 开发产生器 。 |
| Amazon EMR | 带有 Apache Spark 的 Amazon EMR 支持处理不超过 Kinesis Data Streams 限制 (10) 的大型记录。MiBs 要使用大型记录支持，请使用 <code>readStream</code> 函数。有关更多信息，请参阅 Amazon EMR 与 Amazon Kinesis 集成 。 |
| Amazon Data Firehose | 与 Kinesis Data Streams 一起使用时，Amazon Data Firehose 处理大型记录的行为取决于传输目的地： <ul style="list-style-type: none">Amazon S3：支持传输大型记录，无需任何额外配置。当使用数据格式转换时，Firehose 支持传输大型记录。当使用动态分区时，Firehose 不支持传输大型记录。Lambda：当 Firehose 在下游触发 Lambda 函数时，我们不建议将其与大型记录一起使用。这可能会导致间歇性故障。HTTP：Firehose 不支持传输大型记录。Snowflake：Firehose 不支持传输大型记录。 |

| 组件 | 描述 |
|-----------------|--|
| | <ul style="list-style-type: none">Amazon Redshift：Firehose 不支持传输大型记录。 <p>对于需要向 Snowflake 或 Redshift 传输大型记录的应用程序，请先将数据传输至 Amazon S3。然后使用提取、转换、加载 (ETL) 过程来加载数据。对于所有其他目的地，在扩展到生产使用量之前，请在 proof-of-concept 环境中使用大型记录测试行为。大型记录的处理因目的地而不同。</p> |
| Amazon Lambda | <p>Amazon Lambda 支持最多 6 MiBs 个有效载荷。此限制适用于转换为 base-64 编码的 Kinesis 有效负载，以及与事件源映射 (ESM) 有关的元数据。对于小于 6 的记录 MiBs，Lambda 使用 ESM 处理这些记录，无需额外配置。对于大于 6 的记录 MiBs，Lambda 使用失败时目标对其进行处理。必须使用 ESM 配置失败时的目标，才能处理超出 Lambda 处理限制的记录。发送到失败时的目标的各个事件都是一个 JSON 文档，其中包含有关调用失败的元数据。</p> <p>无论记录大小如何，都建议在 ESM 中创建一个失败时的目标。这可以确保不会丢弃任何记录。有关更多信息，请参阅配置调用失败时的目标。</p> |
| Amazon Redshift | <p>在从 Kinesis Data Streams 流式传输数据时，Amazon Redshift 仅支持小于 1 MiB 的记录大小。超过此限制的记录将不予处理。未处理的记录记录为 <code>sys_stream_scan_errors</code>。有关更多信息，请参阅SYS_STREAM_SCAN_ERRORS。</p> |

| 组件 | 描述 |
|----------------------------------|---|
| Kinesis Data Streams 的 Flink 连接器 | 使用来自 Kinesis Data Streams 的数据有两种方法：Kinesis 来源连接器以及 Kinesis 接收器连接器。源连接器支持处理小于 1 MiB 和最多 10 MiB 的记录。MiBs 对于大于 1 MiB 的记录，请勿使用接收器连接器。有关更多信息，请参阅 使用 API 使用连接器在适用于 Apache Flink 的亚马逊托管服务中 DataStream 移动数据 。 |

支持大型记录的区域

此 Amazon Kinesis Data Streams 功能仅在 Amazon 以下区域可用：

| Amazon 区域 | 区域名称 |
|----------------|-------------|
| eu-north-1 | 欧洲地区（斯德哥尔摩） |
| me-south-1 | 中东（巴林） |
| ap-south-1 | 亚太地区（孟买） |
| eu-west-3 | 欧洲地区（巴黎） |
| ap-southeast-3 | 亚太地区（雅加达） |
| us-east-2 | 美国东部（俄亥俄州） |
| af-south-1 | 非洲（开普敦） |
| eu-west-1 | 欧洲地区（爱尔兰） |
| me-central-1 | 中东（阿联酋） |
| eu-central-1 | 欧洲地区（法兰克福） |
| sa-east-1 | 南美洲（圣保罗） |
| ap-east-1 | 亚太地区（香港） |

| Amazon 区域 | 区域名称 |
|----------------|--------------------------|
| ap-south-2 | 亚太地区 (海得拉巴) |
| us-east-1 | 美国东部 (弗吉尼亚州北部) |
| ap-northeast-2 | 亚太地区 (首尔) |
| ap-northeast-3 | 亚太地区 (大阪) |
| eu-west-2 | 欧洲地区 (伦敦) |
| ap-southeast-4 | 亚太地区 (墨尔本) |
| ap-northeast-1 | 亚太地区 (东京) |
| us-west-2 | 美国西部 (俄勒冈州) |
| us-west-1 | 美国西部 (北加利福尼亚) |
| ap-southeast-1 | 亚太地区 (新加坡) |
| ap-southeast-2 | 亚太地区 (悉尼) |
| il-central-1 | 以色列 (特拉维夫) |
| ca-central-1 | 加拿大 (中部) |
| ca-west-1 | 加拿大西部 (卡尔加里) |
| eu-south-2 | 欧洲地区 (西班牙) |
| cn-northwest-1 | 中国 (宁夏) |
| eu-central-2 | 欧洲 (苏黎世) |
| us-gov-east-1 | Amazon GovCloud (美国东部) |
| us-gov-west-1 | Amazon GovCloud (美国西部) |

使用执行弹性测试 Amazon Fault Injection Service

Amazon Fault Injection Service是一项完全托管的服务，可帮助您对Amazon工作负载执行故障注入实验。Amazon FIS与Amazon Kinesis Data Streams的集成使您能够在受控环境中针对常见的Amazon Kinesis Data Streams API错误测试应用程序弹性。此功能支持在遇到故障之前验证错误处理、重试逻辑和监控系统。有关更多信息，请参阅[什么是Amazon Fault Injection Service？](#)。

操作

- API 内部错误：在目标 IAM 角色发出的请求中注入内部错误。具体响应取决于每项服务和 API。`aws:fis:inject-api-internal-error` 操作产生 `InternalFailure` 错误 (HTTP 500)。
- API 限制错误：在目标 IAM 角色发出的请求中注入内部错误。具体响应取决于每项服务和 API。`aws:fis:inject-api-throttle-error` 操作产生 `ThrottlingException` 错误 (HTTP 400)。
- API 不可用错误：在目标 IAM 角色发出的请求中注入内部错误。具体响应取决于每项服务和 API。`aws:fis:inject-api-unavailable-error` 操作产生 `ServiceUnavailable` 错误 (HTTP 503)。
- API 预置吞吐量异常：在目标 IAM 角色发出的请求中注入内部错误。具体响应取决于每项服务和 API。`aws:kinesis:inject-api-provisioned-throughput-exception` 操作产生 `ProvisionedThroughputExceededException` 错误 (HTTP 400)。
- API 迭代器到期异常：在目标 IAM 角色发出的请求中注入内部错误。具体响应取决于每项服务和 API。`aws:kinesis:inject-api-expired-iterator-exception` 操作产生 `ExpiredIteratorException` 错误 (HTTP 400)。

有关更多信息，请参阅[Amazon Kinesis Data Streams 操作](#)。

注意事项

- 可以在Amazon Kinesis Data Streams的预置和按需产品中使用上述操作。
- 根据所选时间完成实验后，流式传输随之恢复。您也可以在实验完成之前停止运行中的实验。或者，您可以根据在Amazon CloudWatch Application Insights中定义应用程序运行状况的警报来定义停止实验的停止条件。
- 最多可以测试280个流。

有关区域支持的更多信息，请参阅[Amazon Fault Injection Service 端点和配额](#)。

预置吞吐量错误

当某个 Kinesis 流的请求速率超出一个或多个分片的吞吐量限制时，就会出现超出预置吞吐量异常错误 (HTTP 400)。每个分片都有特定的读取和写入容量限制，超出这些限制就会触发这种异常。导致此种异常的场景包括：数据摄取或消耗量突然飙升，分片容量不足以应对正在处理的数据量，或者分区键分布不均。

处理异常的建议

- 实现指数退避与重试机制。
- 增加分片数量以适应更高的吞吐量。
- 确保分区键正常分布。
- 监控流指标。

此外，使用 Kinesis 按需容量模式有助于自动调整工作负载，并最大限度地减少这种异常的发生。有关更多信息，请参阅[什么是 Amazon Fault Injection Service？](#)

 Note

分布不当问题不在按需模式的自动扩缩功能范围之内。

进行基础实验

1. 使用基准指标：测试之前记录正常的吞吐量模式。
2. 创建实验：使用 `aws:kinesis:inject-api-provisioned-throughput-exception` 操作。
3. 配置强度：从 25% 的请求限制开始。
4. 监控响应：通过指数退避验证重试逻辑。
5. 验证扩缩：确认自动扩缩放触发了激活。
6. 检查警报：确保 CloudWatch 警报按预期运行。

应用程序应实施适当的退避策略、监控 `WriteProvisionedThroughputExceeded` 和 `ReadProvisionedThroughputExceeded` 指标，并在适当时触发分片扩缩。

操作详细信息

- 资源类型：IAM 角色 ARN

- 目标操作 : PutRecord、PutRecords、GetRecords
- 错误代码 : ProvisionedThroughputExceeded (HTTP 400)
- 描述 : 模拟请求速率超出分片容量限制的场景 , 测试应用程序限制和扩缩响应。

Parameters

- IAM 角色 ARN : 应用程序用于 Kinesis Data Streams 操作的角色。
- 操作 : 目标操作 : PutRecord、PutRecords、GetRecords。
- 资源列表 : 特定的流名称或分片标识符。
- 持续时间 : 实验的持续时间 , 从一分钟到 12 小时不等。在 Amazon FIS API 中 , 该值是 ISO 8601 格式的字符串。例如 , PT1M 代表一分钟。在 Amazon FIS 控制台中 , 您可以输入秒数、分钟数或小时数。
- 强度 : 要施加限制的请求的百分比。

所需的权限

- kinesis:InjectApiError

实验模板示例

以下示例显示了带有指定标签的所有请求 (最多 5 个 Kinesis 数据流) 的预配置吞吐量异常。Amazon FIS 随机选择要影响的直播。5 分钟后故障解决。

```
{  
    "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",  
            "duration": "PT5M",  
            "percentage": 100  
        }  
    }  
}
```

```
        "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"  
    }  
}
```

实验角色权限示例

以下权限支持在特定流上执行 `aws:kinesis:stream-provisioned-throughput-exception` 和 `aws:kinesis:stream-expired-iterator-exception` 操作来影响 50% 的请求。

迭代器过期异常错误

迭代器过期异常错误 (HTTP 400) 是在分片迭代器过期时发生的，在调用 `GetRecords` 时不再用于检索流记录。读取操作之间因为长时间运行数据处理任务、网络问题或应用程序停机而产生延迟时，就会发生这种情况。

Note

分片迭代器在发出后 5 分钟有效。

处理异常的建议

- 在分片迭代器到期之前进行刷新。

- 整合错误处理以获取新的迭代器。
- 利用 Kinesis Client Library (KCL) , 该服务可自动管理分片迭代器的到期时间。

有关更多信息 , 请参阅 [什么是Amazon Fault Injection Service ?](#)

进行基础实验

1. 创建实验模板 : 使用Amazon FIS控制台。
2. 选择操作 : 使用 `aws:kinesis:inject-api-expired-iterator-exception` 操作。
3. 配置目标 : 指定 IAM 角色及 Kinesis Data Streams 操作。
4. 设置持续时间 : 最初测试从 5-10 分钟开始。
5. 添加停止条件 : 的 [停止条件Amazon FIS](#)。
6. 运行实验 : 监控应用程序的行为。

操作详细信息

- 资源类型 : IAM 角色 ARN
- 目标操作 : `GetRecords`
- 错误代码 : `ExpiredIteratorException` (HTTP 400)
- 描述 : 提供的迭代器超出了允许的最大期限 , 模拟了记录处理速度太慢或检查点操作逻辑失败的场景。

Parameters

- IAM 角色 ARN : 应用程序用于 Kinesis Data Streams 操作的角色。
- 操作 : 目标操作 : `GetRecords`
- 资源列表 : 特定的直播名称或 ARNs。
- 持续时间 : 实验的持续时间。此项可配置。
- 强度 : 要施加限制的请求的百分比。

所需的权限

- `kinesis:InjectApiError`

将数据写入 Amazon Kinesis Data Streams

创建器是将数据写入 Amazon Kinesis Data Streams 的应用程序。可以通过将适用于 Java 的 Amazon SDK 和 Kinesis Producer Library (KPL) 结合使用来构建 Kinesis Data Streams 产生器。

如果是首次使用 Kinesis Data Streams , 请先熟悉 [什么是 Amazon Kinesis Data Streams ?](#) 和 [使用 Amazon CLI 执行 Amazon Kinesis Data Streams 操作](#) 中介绍的概念和术语。

Important

Kinesis Data Streams 支持更改数据流的数据记录保留期。有关更多信息 , 请参阅 [更改数据留存期](#)。

要将数据放入流 , 您必须指定流的名称、分区键和要添加到流的数据 Blob。分区键用来确定数据记录将添加到流中的哪个分片。

分片中的所有数据将发送至正在处理分片的同一个工作程序。使用哪个分区键取决于您的应用程序逻辑。通常 , 分区键的数量应比分片的数量多得多。这是因为分区键用来确定如何将数据记录映射到特定分片。如果您有足够的分区键 , 数据可以在流中的分片间均匀分布。

主题

- [使用 Amazon Kinesis Producer Library \(KPL \) 开发产生器](#)
- [使用 Amazon Kinesis Data Streams API 和 适用于 Java 的 Amazon SDK 开发产生器](#)
- [使用 Kinesis 代理写入 Amazon Kinesis Data Streams](#)
- [使用其他 Amazon 服务写入 Kinesis Data Streams](#)
- [使用第三方集成写入 Kinesis Data Streams](#)
- [Amazon Kinesis Data Streams 产生器问题排查](#)
- [优化 Kinesis Data Streams 产生器](#)

使用 Amazon Kinesis Producer Library (KPL) 开发产生器

Amazon Kinesis Data Streams 创建器是指将用户数据记录放入 Kinesis 数据流中 (也称为数据摄取) 的应用程序。Amazon Kinesis Producer Library (KPL) 简化了创建器应用程序的开发 , 从而允许开发人员实现到 Kinesis 数据流的高写入吞吐量。

您可以使用 Amazon CloudWatch 监控 KPL。有关更多信息，请参阅 [使用 Amazon CloudWatch 监控 Kinesis Producer Library](#)。

主题

- [查看 KPL 的作用](#)
- [了解使用 KPL 的优势](#)
- [了解何时不使用 KPL](#)
- [安装 KPL](#)
- [从 KPL 0.x 迁移至 KPL 1.x](#)
- [转换为适用于 KPL 的 Amazon Trust Services \(ATS \) 证书](#)
- [KPL 受支持平台](#)
- [KPL 的重要概念](#)
- [将 KPL 与产生器代码集成](#)
- [使用 KPL 写入 Kinesis Data Streams](#)
- [配置 Amazon Kinesis Producer Library](#)
- [实现消费端取消聚合](#)
- [将 KPL 与 Amazon Data Firehose 搭配使用](#)
- [将 KPL 与 Amazon Glue 架构注册表搭配使用](#)
- [配置 KPL 代理配置](#)
- [KPL 版本生命周期策略](#)

 Note

建议您升级到最新 KPL 版本。KPL 会定期更新至最新版本，包括最新依赖项和安全补丁、错误修复以及向后兼容的新功能。有关更多信息，请参阅 <https://github.com/awslabs/amazon-kinesis-producer/releases/>。

查看 KPL 的作用

KPL 是一个易于使用的且高度可配置的库，可帮助您对 Kinesis 数据流进行写入。它在您的创建器应用程序代码和 Kinesis Data Streams API 操作之间充当中介。KPL 执行以下主要任务：

- 利用可配置的自动重试机制对一个或多个 Kinesis 数据流进行写入

- 收集记录并使用 PutRecords 根据请求将多条记录写入多个分片
- 聚合用户记录以增加负载大小并提高吞吐量
- 与 [Kinesis Client Library \(KCL \)](#) 无缝集成以在消费端上取消聚合批记录
- 代表您提交 Amazon CloudWatch 指标以提供创建器性能的可见性

请注意，KPL 与 [Amazon 开发工具包](#) 中可用的 Kinesis Data Streams API 不同。Kinesis Data Streams API 可帮助您管理 Kinesis Data Streams 的许多方面（包括创建流、重新分片以及放置并获取记录），而 KPL 提供专用于摄取数据的提取层。有关 Kinesis Data Streams API 的信息，请参阅 [Amazon Kinesis API Reference](#)。

了解使用 KPL 的优势

以下列表说明了使用 KPL 开发 Kinesis Data Streams 创建器的部分主要优势。

KPL 可在同步或异步使用案例中使用。除非存在使用同步操作的具体原因，否则建议您使用异步接口的较高性能。有关这两种使用案例和示例代码的更多信息，请参阅 [使用 KPL 写入 Kinesis Data Streams](#)。

性能优势

KPL 可帮助构建高性能创建器。考虑以下情况：您的 Amazon EC2 实例充当代理，从数以百计或数以千计的低功率设备中收集 100 个字节的事件并将记录写入 Kinesis 数据流中。这些 EC2 实例均须将每秒数以千计的事件写入您的数据流。要实现所需的吞吐量，创建器必须实施复杂逻辑（例如，批处理或多线程处理）及重试逻辑并在消费端端取消记录聚合。KPL 为您执行所有此类任务。

消费端端易于使用

对于使用采用 Java 的 KCL 的消费端端开发人员而言，KPL 无需额外工作即可集成。当 KCL 检索包含多个 KPL 用户记录的已聚合 Kinesis Data Streams 记录时，它将自动调用 KPL，以在将单个用户记录返还到用户之前提取此类记录。

对于未使用 KCL 而是直接使用 API 操作 GetRecords 的消费端端开发人员而言，KPL Java 库可用于在将用户记录返还给用户之前提取此类记录。

创建器监控

您可使用 Amazon CloudWatch 和 KPL 收集、监控和分析您的 Kinesis Data Streams 创建器。KPL 代表您向 CloudWatch 发出吞吐量、错误和其他指标，并可配置为在流、分片或创建器级别进行监控。

异步架构

由于 KPL 可在将记录发送到 Kinesis Data Streams 之前对其进行缓冲处理，因此它在继续运行时之前不会强制调用方应用程序阻止和等待确认记录已到达服务器。用于将记录放入 KPL 中的调用应始终立即返回，并且不等待发送记录或接收来自服务器的响应。相反，将创建一个 Future 对象，该对象稍后将接收向 Kinesis Data Streams 发送记录的结果。这与 Amazon 开发工具包中的异步客户端的行为相同。

了解何时不使用 KPL

当 KPL 会导致库（用户可配置）中产生高达 RecordMaxBufferedTime 的额外处理延迟时。RecordMaxBufferedTime 值越大，产生的包装效率和性能就越高。无法容忍此额外延迟的应用程序可能需要直接使用 Amazon 开发工具包。有关使用 Amazon 开发工具包和 Kinesis Data Streams 的更多信息，请参阅 [使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon SDK 开发产生器](#)。有关 RecordMaxBufferedTime 和 KPL 其他用户可配置属性的更多信息，请参阅 [配置 Amazon Kinesis Producer Library](#)。

安装 KPL

Amazon 提供了针对 macOS、Windows 和最新 Linux 发行版（有关支持的平台的详细信息，请参阅下一部分）的 C++ Amazon Kinesis Producer Library（KPL）的预先构建的二进制文件。这些二进制文件打包在 Java .jar 文件中，如果您使用 Maven 安装程序包，将自动调用和使用这些二进制文件。要查找最新版的 KPL 和 KCL，请使用以下 Maven 搜索链接：

- [KPL](#)
- [KCL](#)

Linux 二进制文件已采用 GNU 编译器集合 (GCC) 进行编译并已静态链接到 Linux 上的 libstdc++。这些二进制文件应适用于包含 glibc 2.5 版或更高版本的任何 64 位 Linux 发行版。

早期 Linux 发行版的用户可使用 GitHub 上与源一起提供的构建说明构建 KPL。要从 GitHub 中下载 KPL，请参阅 [Amazon Kinesis Producer Library](#)。

Important

Amazon Kinesis Producer Library (KPL) 0.x 将于 2026 年 1 月 30 日终止支持。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 0.x 的 KPL 应用程序迁移到最新的 KPL 版本。要

查找最新的 KPL 版本，请参阅 [Github 上的 KPL 页面](#)。有关从 KPL 0.x 迁移至 KPL 1.x 的信息，请参阅 [从 KPL 0.x 迁移至 KPL 1.x](#)。

从 KPL 0.x 迁移至 KPL 1.x

本主题提供了将消费端从 KPL 0.x 迁移至 KPL 1.x 的分步说明。KPL 1.x 引入了对适用于 Java 的 Amazon SDK 2.x 的支持，同时保留了与先前版本的接口兼容性。无需更新核心数据处理逻辑即可迁移至 KPL 1.x。

1. 确保有以下先决条件：

- Java Development Kit (JDK) 8 或更高版本
- 适用于 Java 的 Amazon SDK 2.x
- 用于依赖项管理的 Maven 或 Gradle

2. 添加依赖项。

如果您使用的是 Maven，请将以下依赖项添加到您的 pom.xml 文件中。务必将 GroupID 从 com.amazonaws 更新到 software.amazon.kinesis，并将版本 1.x.x 更新到最新的 KPL 版本。

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

如果使用 Gradle，请在 build.gradle 文件中添加以下信息。请务必将 1.x.x 替换为最新 KPL 版本。

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

可以在 [Maven Central 存储库](#) 中查看最新版本的 KPL。

3. 更新 KPL 的导入语句

KPL 1.x 使用适用于 Java 的 Amazon SDK 2.x，更新后的软件包名称以 software.amazon.kinesis 开头，而之前的 KPL 中的软件包名称则以 com.amazonaws.services.kinesis 开头。

将 `com.amazonaws.services.kinesis` 的导入替换为 `software.amazon.kinesis`。下表列出了必须替换的导入。

导入替换

| 进行如下替换： | 借助： |
|---|--|
| <code>import com.amazonaws.services.kinesis.producer.Attempt;</code> | <code>import software.amazon.kinesis.producer.Attempt;</code> |
| <code>import com.amazonaws.services.kinesis.producer.BinaryToHexConverter;</code> | <code>import software.amazon.kinesis.producer.BinaryToHexConverter;</code> |
| <code>import com.amazonaws.services.kinesis.producer.CertificateExtractor;</code> | <code>import software.amazon.kinesis.producer.CertificateExtractor;</code> |
| <code>import com.amazonaws.services.kinesis.producer.Daemon;</code> | <code>import software.amazon.kinesis.producer.Daemon;</code> |
| <code>import com.amazonaws.services.kinesis.producer.DaemonException;</code> | <code>import software.amazon.kinesis.producer.DaemonException;</code> |
| <code>import com.amazonaws.services.kinesis.producer.FileAgeManager;</code> | <code>import software.amazon.kinesis.producer.FileAgeManager;</code> |
| <code>import com.amazonaws.services.kinesis.producer.FutureTimedOutException;</code> | <code>import software.amazon.kinesis.producer.FutureTimedOutException;</code> |
| <code>import com.amazonaws.services.kinesis.producer.GlueSchemaRegistrySerializerInstance;</code> | <code>import software.amazon.kinesis.producer.GlueSchemaRegistrySerializerInstance;</code> |
| <code>import com.amazonaws.services.kinesis.producer.HashedFileCopier;</code> | <code>import software.amazon.kinesis.producer.HashedFileCopier;</code> |
| <code>import com.amazonaws.services.kinesis.producer.IKinesisProducer;</code> | <code>import software.amazon.kinesis.producer.IKinesisProducer;</code> |

| 进行如下替换： | 借助: |
|--|---|
| import com.amazonaws.services.kinesis.producer.IrrecoverableError; | import software.amazon.kinesis.producer.IrrecoverableError; |
| 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. 更新 Amazon 凭证提供程序类的导入语句

迁移至 KPL 1.x 时，必须将 KPL 应用程序代码的导入中基于适用于 Java 的 Amazon SDK 1.x 的包和类更新为基于适用于 Java 的 Amazon SDK 2.x 的相应包和类。KPL 应用程序中通常导入的是凭证提供程序类。有关凭证提供程序更改的完整列表，请参阅适用于 Java 的 Amazon SDK 2.x 迁移指南文档中的[凭证提供程序更改](#)。下面是可能需要在 KPL 应用程序中进行的常见导入更改。

KPL 0.x 中的导入

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

KPL 1.x 中的导入

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

如果导入其他任何基于适用于 Java 的 Amazon SDK 1.x 的凭证提供程序，则必须将其更新为适用于 Java 的 Amazon SDK 2.x 的凭证提供程序。如果未从适用于 Java 的 Amazon SDK 1.x 中导入任何类/包，可忽略此步骤。

5. 更新 KPL 配置中的凭证提供程序配置

KPL 1.x 中的凭证提供程序配置需要适用于 Java 的 Amazon SDK 2.x 凭证提供程序。如果通过覆盖默认凭证提供程序来传递 KinesisProducerConfiguration 中的适用于 Java 的 Amazon SDK 1.x 凭证提供程序，则必须使用适用于 Java 的 Amazon SDK 2.x 凭证提供程序进行更新。有关凭证提供程序更改的完整列表，请参阅适用于 Java 的 Amazon SDK 2.x 迁移指南文档中的[凭证提供程序更改](#)。如果未覆盖 KPL 配置中的默认凭证提供程序，可忽略此步骤。

例如，如果要使用以下代码覆盖 KPL 的默认凭证提供程序：

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

必须使用以下代码进行更新才能使用适用于 Java 的 Amazon SDK 2.x 凭证提供程序：

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration();
// New SDK v2 default credentials provider
config.setCredentialsProvider(DefaultCredentialsProvider.create());
```

转换为适用于 KPL 的 Amazon Trust Services (ATS) 证书

在 2018 年 2 月 9 日上午 9:00 (太平洋标准时间) , Amazon Kinesis Data Streams 安装了 ATS 证书。要能继续使用 Amazon Kinesis Producer Library (KPL) 向 Kinesis Data Streams 写入记录 , 您必须将安装的 KPL 升级到[版本 0.12.6](#) 或更高版本。此更改会影响所有 Amazon 区域。

有关迁移到 ATS 的信息 , 请参阅 [How to Prepare for Amazon's Move to Its Own Certificate Authority](#)。

如果您遇到问题 , 需要技术支持 , 请通过 Amazon Support 中心[创建案例](#)。

KPL 受支持平台

Amazon Kinesis Producer Library (KPL) 采用 C++ 编写 , 作为主用户进程的子进程运行。预编译的 64 位本机二进制文件与 Java 版本捆绑在一起并由 Java 包装程序管理。

无需在以下操作系统上安装任何其他库即可运行 Java 程序包 :

- 内核为 2.6.18 (2006 年 9 月) 及更高版本的 Linux 发行版
- Apple iOS X 10.9 及更高版本
- Windows Server 2008 及更高版本

Important

0.14.0 之前的所有 KPL 版本支持 Windows Server 2008 及更高版本。

KPL 0.14.0 及更高版本将不再支持该 Windows 平台。

请注意 , KPL 仅为 64 位。

源代码

如果 KPL 安装中提供的二进制文件无法满足您的环境 , 则 KPL 的内核将编写为 C++ 模块。C++ 模块和 Java 接口的源代码将在 Amazon 公共许可证下发布并可通过 [Amazon Kinesis Producer Library](#) 上的 GitHub 获得。虽然 KPL 可在安装了最新的符合标准的 C++ 编译器和 JRE 的任何平台上使用 , 但 Amazon 仍未正式支持不在受支持的平台列表中的任何平台。

KPL 的重要概念

以下部分包含了解 Amazon Kinesis Producer Library (KPL) 并从中获益所需的概念和术语。

主题

- [记录](#)
- [批处理](#)
- [聚合](#)
- [集合](#)

记录

在本指南中，我们区分了 KPL 用户记录和 Kinesis Data Streams 记录。当我们使用没有限定词的术语记录时，我们指 KPL 用户记录。当我们提及 Kinesis Data Streams 记录时，我们明确指 Kinesis Data Streams 记录。

KPL 用户记录是对用户有特别含义的数据 Blob。示例包括表示网站上的 UI 事件的 JSON Blob 或 Web 服务器中的日志条目。

Kinesis Data Streams 记录是 Kinesis Data Streams 服务 API 所定义 Record 数据结构的实例。它包含一个分区键、序列号和数据 Blob。

批处理

批处理 指对多个项执行单个操作而不是对每个单独的项重复执行操作。

在此背景下，“项”是一条记录，操作是将项发送到 Kinesis Data Streams。在非批处理情况下，您会将每条记录放置在单独的 Kinesis Data Streams 记录中，并发出一条 HTTP 请求以将其发送到 Kinesis Data Streams。利用批处理，每个 HTTP 请求可携带多条记录而不仅仅是一条记录。

KPL 支持两种批处理：

- 聚合 – 在单条 Kinesis Data Streams 记录中存储多条记录。
- 集合 – 使用 API 操作 PutRecords 将多条 Kinesis Data Streams 记录发送到 Kinesis 数据流中的一个或多个分片。

这两种类型的 KPL 批处理旨在共存并可彼此独立启用或禁用。默认情况下，两种批处理将同时启用。

聚合

聚合指在一条 Kinesis Data Streams 记录中存储多条记录。聚合允许客户增加每个 API 调用发送的记录的数目，这将有效增加创建器吞吐量。

Kinesis Data Streams 分片每秒支持最多 1,000 条 Kinesis Data Streams 记录或 1 MB 吞吐量。每秒 Kinesis Data Streams 记录数限制将绑定具有 1 KB 以下记录的客户。通过记录聚合，客户可以将多条记录合并为一条 Kinesis Data Streams 记录。这使客户能够提高其每分片吞吐量。

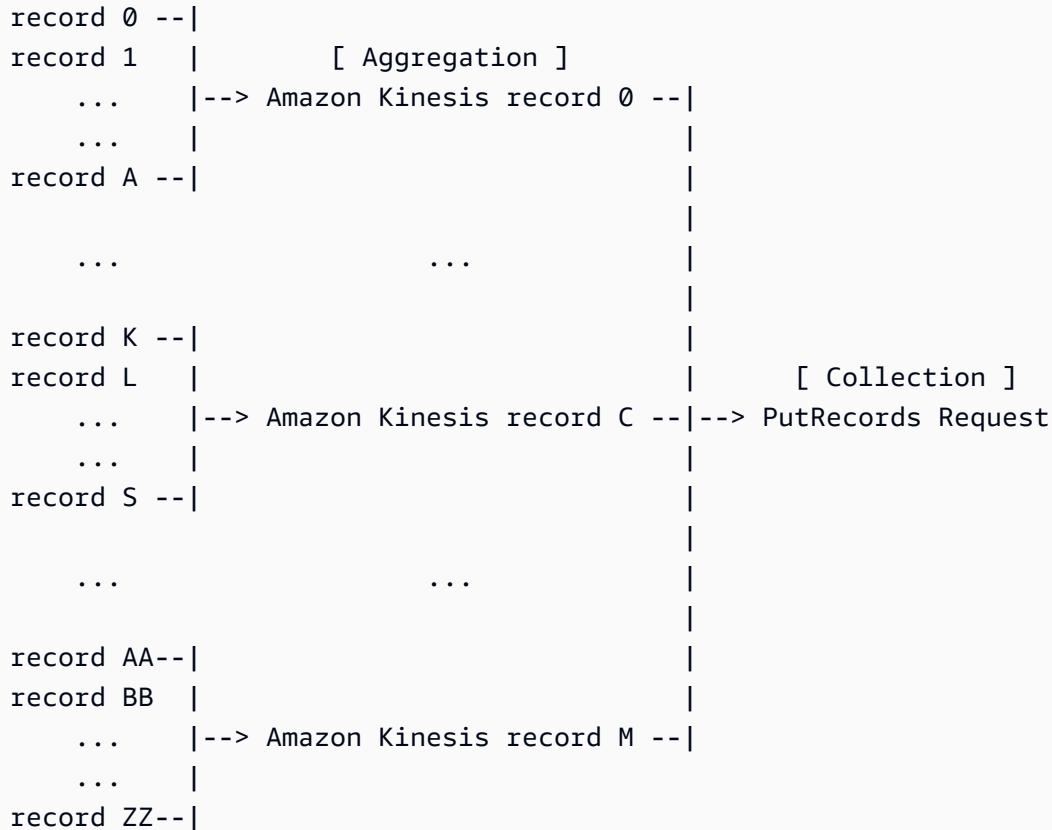
考虑以下情况：区域 us-east-1 中的一个分片当前正在以每秒 1000 条记录的恒速运行，其中每条记录的大小为 512 个字节。利用 KPL 聚合，您可将 1000 条记录打包到 10 条 Kinesis Data Streams 记录中，从而将 RPS 降低到 10（每条记录 50 KB）。

集合

集合指批处理多条 Kinesis Data Streams 记录，并通过调用 API 操作 PutRecords 在单个 HTTP 请求中发送此类记录，而无需在其自己的 HTTP 请求中发送每条 Kinesis Data Streams 记录。

与不使用集合相比，这将增加吞吐量，因为它减少了发出多个单独 HTTP 请求的开销。实际上，PutRecords 本身专用于实现此目的。

集合与聚合的不同之处在于，前者处理的是 Kinesis Data Streams 记录组。正在收集的 Kinesis Data Streams 记录仍可包含来自该用户的多条记录。可通过如下方式来可视化此关系：



将 KPL 与产生器代码集成

Amazon Kinesis Producer Library (KPL) 在单独进程中运行，并使用 IPC 与您的父用户进程通信。此架构有时称为 [微服务](#)，出于两个主要原因选择该架构：

1) 即使 KPL 发生崩溃，也不会妨碍您的用户进程

您的进程可具有与 Kinesis Data Streams 不相关的任务，并且在 KPL 发生崩溃时仍能继续运行。对于您的父用户进程来说，也可重新启动 KPL 并恢复到完全运行状态（此功能位于正式包装程序中）。

例如，将指标发送到 Kinesis Data Streams 的 Web 服务器；即使 Kinesis Data Streams 部分已停止运行，该服务器仍能继续使用页面。由于 KPL 中存在错误，导致整个服务器发生崩溃，从而造成不必要的中断。

2) 可支持任意客户端

始终存在使用官方支持的语言之外的语言的客户。这些客户也应能轻松使用 KPL。

推荐的使用矩阵

以下使用矩阵列出了不同用户的推荐设置，并向您提供了有关是否以及如何使用 KPL 的建议。请记住，如果启用聚合，则还必须使用取消聚合来提取消费端端的记录。

| 创建器端语言 | 消费端端语言 | KCL 版本 | 检查点逻辑 | 您是否可以使用 KPL？ | 警告 |
|----------------|----------------|-------------------|-------|--------------|---------------------------------------|
| 除 Java 之外的任何语言 | * | * | * | 否 | 不适用 |
| Java | Java | 直接使用 Java 软件开发工具包 | 不适用 | 是 | 如果使用聚合，您必须在 GetRecords 调用后使用提供的取消聚合库。 |
| Java | 除 Java 之外的任何语言 | 直接使用软件开发工具包 | 不适用 | 是 | 必须禁用聚合。 |

| 创建器端语言 | 消费端端语言 | KCL 版本 | 检查点逻辑 | 您是否可以使用 KPL？ | 警告 |
|--------|----------------|------------------------------|--------------|--------------|----------------------------|
| Java | Java | 1.3.x | 不适用 | 是 | 必须禁用聚合。 |
| Java | Java | 1.4.x | 调用不带任何参数的检查点 | 是 | 无 |
| Java | Java | 1.4.x | 调用带明确序列号的检查点 | 是 | 禁用聚合或更改代码以使用扩展的序列号进行检查点操作。 |
| Java | 除 Java 之外的任何语言 | 1.3.x + 多语言守护进程 + 特定于语言的包装程序 | 不适用 | 是 | 必须禁用聚合。 |

使用 KPL 写入 Kinesis Data Streams

以下部分说明了正在运行的示例代码，从最基本的产生器到完全异步的代码。

Barebones 产生器代码

以下代码是写入最小工作创建器所需的全部代码。Amazon Kinesis Producer Library (KPL) 用户记录在后台处理。

```
// 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 ...
```

同步响应结果

在上一个示例中，该代码未检查 KPL 用户记录是否已成功。KPL 执行为说明失败原因所需的任何重试。但如果要检查结果，可使用从 Future 返回的 addUserRecord 对象检查结果，如以下示例中所示（显示上一示例是为了提供上下文信息）：

```
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
        }
    }
}
```

异步响应结果

上一个示例对 Future 对象调用 get()，这会阻止运行时。如果您不想阻止运行时，则可使用异步回调，如以下示例所示：

```
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);  
}
```

配置 Amazon Kinesis Producer Library

虽然默认设置应适用于大多数使用案例，但您可能想更改部分默认设置以定制 `KinesisProducer` 的行为来满足您的需求。为此，可将 `KinesisProducerConfiguration` 类的实例传递给 `KinesisProducer` 构造函数，例如：

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

您也可从属性文件中加载配置：

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

您可替换用户进程可访问的任何路径和文件名。此外，您可在通过此方式创建的 `KinesisProducerConfiguration` 实例上调用 `set` 方法来自定义 `config`。

属性文件应使用以帕斯卡拼写法书写的名称来指定参数。这些名称将与 `KinesisProducerConfiguration` 类中的设置方法中使用的名称匹配。例如：

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

有关配置参数使用规则和值限制的更多信息，请参阅 [GitHub 上的示例配置属性文件](#)。

请注意，初始化 `KinesisProducer` 后，更改已使用的 `KinesisProducerConfiguration` 实例不会产生进一步的影响。`KinesisProducer` 当前不支持动态重新配置。

实现消费端取消聚合

从版本 1.4.0 开始，KCL 支持自动取消聚合 KPL 用户记录。在更新 KCL 后，将编译利用 KCL 早期版本编写的消费端应用程序代码，而无需进行任何修改。不过，如果正在创建器端使用 KPL 聚合，则有一个与检查点操作相关的细微之处：已聚合记录中的所有子记录都具有相同的序列号，因此如果您需要区分子记录，则必须利用检查点存储额外数据。此额外数据称作子序列号。

选项

- [从以前的 KCL 版本迁移](#)
- [将 KCL 扩展用于 KPL 取消聚合](#)
- [直接使用 GetRecords](#)

从以前的 KCL 版本迁移

您无需更改现有调用来执行检查点操作与聚合。仍确保您能够成功检索 Kinesis Data Streams 中存储的所有记录。KCL 现在提供两个新检查点操作来支持特定的使用案例，如下所述。

如果您的现有代码先于 KPL 支持之前已为 KCL 写入，并且调用了没有参数的检查点操作，则它等同于对批处理中的上个 KPL 用户记录的序列号进行检查点操作。如果调用带序列号字符串的检查点操作，则它等同于对批处理的指定序列号以及隐式子序列号 0 (零) 进行检查点操作。

调用没有任何参数的新 KCL 检查点 `checkpoint()` 操作，在语义上等同于对批处理中的上个 `Record` 调用的序列号以及隐式子序列号 0 (零) 进行检查点操作。

调用新 KCL 检查点操作 `checkpoint(Record record)`，在语义上等同于对指定 `Record` 的序列号以及隐式子序列号 0 (零) 进行检查点操作。如果 `Record` 调用实际为 `UserRecord`，则对 `UserRecord` 序列号和子序列号进行检查点操作。

调用新 KCL 检查点操作 `checkpoint(String sequenceNumber, long subSequenceNumber)` 会对给定的序列号以及子序列号进行显式检查点操作。

在上述任意情况下，在将检查点存储在 Amazon DynamoDB 检查点表中后，KCL 可正确地恢复检索记录，即使是在应用程序发生崩溃并重新启动时也是如此。如果在序列中包含多条记录，则检索会从具有已进行检查点操作的最新序列号的记录中的下个子序列号记录开始。如果最新检查点包括上一条序列号记录的最新子序列号，则检索会从具有下个序列号的记录开始。

以下部分将讨论必须避免跳过和重复记录的消费端的序列和子序列检查点操作的详细信息。如果在停止并重新启动消费端的记录处理时跳过（或重复）记录并不重要，则可运行您的现有代码而无需进行修改。

将 KCL 扩展用于 KPL 取消聚合

KPL 取消聚合会涉及子序列检查点操作。为了帮助使用子序列检查点操作，已将 `UserRecord` 类添加到 KCL：

```
public class UserRecord extends Record {
    public long getSubSequenceNumber() {
        /* ... */
    }
    @Override
    public int hashCode() {
        /* contract-satisfying implementation */
    }
    @Override
    public boolean equals(Object obj) {
        /* contract-satisfying implementation */
    }
}
```

现在使用的是此类而不是 `Record`。这不会破坏现有代码，因为它是 `Record` 的子类。`UserRecord` 类同时表示实际子记录和标准非聚合记录。非聚合记录可被视为刚好具有一条子记录的聚合记录。

此外，将两个新的操作添加到 `IRecordProcessorCheckpointer`：

```
public void checkpoint(Record record);
public void checkpoint(String sequenceNumber, long subSequenceNumber);
```

要开始使用子序列号检查点操作，您可执行以下转换。更改以下形式代码：

```
checkpointer.checkpoint(record.getSequenceNumber());
```

新的形式代码：

```
checkpointer.checkpoint(record);
```

建议您使用 `checkpoint(Record record)` 形式来进行子序列检查点操作。不过，如果您已以字符串形式存储 `sequenceNumbers` 以用于检查点操作，则您现在也应存储 `subSequenceNumber`，如以下示例所示：

```
String sequenceNumber = record.getSequenceNumber();
long subSequenceNumber = ((UserRecord) record).getSubSequenceNumber(); // ... do other
processing
checkpointer.checkpoint(sequenceNumber, subSequenceNumber);
```

从 `Record` 到 `UserRecord` 的转换始终会成功，因为其实现始终使用 `UserRecord`。除非需要对序列号进行算术运算，否则建议不要采用此方法。

在处理 KPL 用户记录时，KCL 会将子序列号作为每行的额外字段写入 Amazon DynamoDB。在恢复检查点操作时，早期版本的 KCL 使用 `AFTER_SEQUENCE_NUMBER` 提取记录。而具有 KPL 支持的当前 KCL 则使用 `AT_SEQUENCE_NUMBER`。在检索带已进行检查点操作的序列号的记录时，会检查已进行检查点操作的子序列号，而且会根据需要删除子记录（如果上一条子记录为已进行检查点操作的记录，则可能删除所有子记录）。同样，非聚合记录可被视为具有单条子记录的聚合记录，因此相同的算法同时适用于聚合和非聚合记录。

直接使用 GetRecords

您也可以选择不使用 KCL 而是直接调用 API 操作 `GetRecords` 来检索 Kinesis Data Streams 记录。要将已检索到的记录提取到原始 KPL 用户记录中，可在 `UserRecord.java` 中调用下列静态操作之一：

```
public static List<Record> deaggregate(List<Record> records)

public static List<UserRecord> deaggregate(List<UserRecord> records, BigInteger
startingHashKey, BigInteger endingHashKey)
```

第一个操作使用 0 的默认值 `startingHashKey`（零）和 $2^{128} - 1$ 的默认值 `endingHashKey`。

每个操作都会取消将 Kinesis Data Streams 记录的给定列表聚合到 KPL 用户记录的列表中。将从返回的记录列表中删除其显式哈希键或分区键位于 `startingHashKey` (包含) 和 `endingHashKey` (包含) 的范围之外的任何 KPL 用户记录。

将 KPL 与 Amazon Data Firehose 搭配使用

如果使用 Kinesis Producer Library (KPL) 将数据写入 Kinesis 数据流，则可以使用聚合来合并写入该 Kinesis 数据流的记录。如果随后将该数据流用作 Firehose 传输流的来源，Firehose 将在记录传送到目标之前取消聚合。如果您配置传输流来转换数据，Firehose 在将记录传输到 Amazon Lambda 之前，会先取消聚合记录。有关更多信息，请参阅 [Writing to Amazon Firehose Using Kinesis Data Streams](#)。

将 KPL 与 Amazon Glue 架构注册表搭配使用

您可以将 Kinesis 数据流与 Amazon Glue 架构注册表进行集成。Amazon Glue 架构注册表能帮助您集中发现、控制和演变架构，同时确保注册架构持续验证生成的数据。架构定义了数据记录的结构和格式。架构是用于可靠数据发布、使用或存储的版本化规范。通过 Amazon Glue 架构注册表，您能够改善流媒体应用程序中的端到端数据质量和数据治理。有关更多信息，请参阅 [Amazon Glue Schema Registry](#)。设置此集成的方法之一是通过 Java 中的 KPL 和 Kinesis Client Library (KCL) 进行。

Important

目前，只有使用在 Java 中实现的 KPL 产生器的 Kinesis 数据流支持 Kinesis Data Streams 和 Amazon Glue 架构注册表集成。不提供多语言支持。

有关如何使用 KPL 设置 Kinesis Data Streams 与架构注册表集成的详细说明，请参阅 [Use Case: Integrating Amazon Kinesis Data Streams with the Amazon Glue Schema Registry](#) 中的“Interacting with Data Using the KPL/KCL Libraries”部分。

配置 KPL 代理配置

对于无法直接连接到互联网的应用程序，所有 Amazon 开发工具包客户端都支持使用 HTTP 或 HTTPS 代理。在一般企业环境中，所有出站网络流量都必须通过代理服务器。如果您的应用程序使用 Kinesis Producer Library (KPL)，以在使用代理服务器的环境中向 Amazon 收集和发送数据，则您的应用程序将需要配置 KPL 代理。KPL 是基于 Amazon Kinesis 开发工具包构建的高级库。它分为原生进程和包装器。原生进程执行处理和发送记录的所有工作，而包装器则管理原生进程并与之通信。有关更多信息，请参阅 [Implementing Efficient and Reliable Producers with the Amazon Kinesis Producer Library](#)。

包装器在 Java 中编写，而原生进程使用 Kinesis 开发工具包在 C++ 中编写。KPL 0.14.7 及更高版本当前支持 Java 包装器中的代理配置，可以将所有代理配置传递至原生进程。有关更多信息，请参阅<https://github.com/awslabs/amazon-kinesis-producer/releases/tag/v0.14.7>。

您可以使用以下代码向 KPL 应用程序添加代理配置。

```
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 版本生命周期策略

本主题概述了 Amazon Kinesis Producer Library (KPL) 的版本生命周期策略。Amazon 定期为 KPL 版本提供新发行版本，以支持新功能和增强功能、错误修复、安全补丁以及依赖项更新。我们建议您及时了解 KPL 版本，以了解其最新功能、安全更新和底层依赖项。我们不建议继续使用不受支持的 KPL 版本。

主要 KPL 版本的生命周期包括以下三个阶段：

- 正式发布 (GA)：在该阶段，主要版本是得到全面支持的。Amazon 提供常规的次要版本和补丁版本，其中包括支持 Kinesis Data Streams 的新功能或 API 更新，以及错误和安全修复。
- 维护模式：Amazon 将补丁版本限制为仅解决严重错误修复和安全问题。主要版本将不会收到 Kinesis Data Streams 的新功能或 API 更新。
- 终止支持：主要版本将不再收到更新或发行版本。之前发布的版本将继续通过公共程序包管理器提供，并且代码将保留在 GitHub 上。用户可自行决定是否使用已终止支持的版本。建议您升级到最新的主要版本。

| 主要版本 | 当前阶段 | 发行日期 | 维护模式日期 | 终止支持日期 |
|---------|------|------------|------------|-----------------|
| KPL 0.x | 维护模式 | 2015-06-02 | 2025-04-17 | 2026 年 1 月 30 日 |
| KPL 1.x | 正式发布 | 2024-12-15 | -- | -- |

使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon SDK 开发产生器

您可以使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon 开发工具包开发创建器。如果是首次使用 Kinesis Data Streams，请先熟悉 [什么是 Amazon Kinesis Data Streams？](#) 和 [使用 Amazon CLI 执行 Amazon Kinesis Data Streams 操作](#) 中介绍的概念和术语。

这些示例讨论 [Kinesis Data Streams API](#) 并使用[适用于 Java 的 Amazon 开发工具包](#)向流中添加（放置）数据。但是，在大多数使用案例中，您应会更喜欢使用 Kinesis Data Streams KPL 库。有关更多信息，请参阅 [使用 Amazon Kinesis Producer Library \(KPL \) 开发产生器](#)。

本章中的 Java 示例代码演示如何执行基本的 Kinesis Data Streams API 操作，并按照操作类型从逻辑上进行划分。这些示例并非可直接用于生产的代码，因为它们不会检查所有可能的异常，或者不会考虑到所有可能的安全或性能问题。此外，您可使用其他编程语言调用 [Kinesis Data Streams API](#)。有关所有可用 Amazon 开发工具包的更多信息，请参阅[开始使用亚马逊云科技开发](#)。

每个任务都有先决条件；例如，您在创建流之后才能向流中添加数据，而创建流需要先创建一个客户端。有关更多信息，请参阅 [创建和管理 Kinesis 数据流](#)。

主题

- [向流添加数据](#)
- [使用 Amazon Glue 架构注册表与数据交互](#)

向流添加数据

在创建流之后，您可以记录的形式向其中添加数据。记录是一种数据结构，其中包含要处理的数据（采用数据 Blob 形式）。在将数据存储到记录中之后，Kinesis Data Streams 不会以任何形式检查、解释或更改数据。每个记录还有一个关联的序列号和分区键。

Kinesis Data Streams API 中有两种不同操作可向流添加数据：[PutRecords](#) 和 [PutRecord](#)。PutRecords 操作将按 HTTP 请求向您的流发送多个记录，并且单个 PutRecord 操作一次可向您的流发送多个记录（每个记录需要单独的 HTTP 请求）。对于大多数应用程序，您应会更喜欢使用 PutRecords，因为这将使每个数据创建者实现更高的吞吐量。有关每种操作的更多信息，请参阅以下各小节。

主题

- [使用 PutRecords 添加多条记录](#)
- [使用 PutRecord 添加单一记录](#)

请始终记住，在您的源应用程序使用 Kinesis Data Streams API 向流添加数据时，很可能存在一个或多个同时处理离开流的数据的消费端应用程序。有关消费端如何使用 Kinesis Data Streams API 获取数据的信息，请参阅 [从流中获取数据](#)。

Important

[更改数据留存期](#)

使用 PutRecords 添加多条记录

[PutRecords](#) 操作可在单个请求中向 Kinesis Data Streams 发送多条记录。向 Kinesis 数据流发送数据时，创建器可以使用 PutRecords 实现更高的吞吐量。每个 PutRecords 请求最多可以支持 500 条记录。请求中的每一个记录最大可以为 1 MB，整个请求的上限为 5 MB，包括分区键。与下面描述的单个 PutRecord 操作一样，PutRecords 将使用序列号和分区键。但是，PutRecord 参数 SequenceNumberForOrdering 未包含在 PutRecords 调用中。PutRecords 操作将尝试按请求的自然顺序处理所有记录。

每个数据记录都有一个唯一的序列号。此序列号在您调用 `client.putRecords` 向流添加数据记录之后由 Kinesis Data Streams 分配。同一分区键的序列号通常会随时间变化增加；PutRecords 请求之间的时间段越长，序列号变得越大。

Note

序列号不能用作相同流中的数据集的索引。为了在逻辑上分隔数据集，请使用分区键或者为每个数据集创建单独的流。

`PutRecords` 请求可包含具有不同分区键的记录。请求的应用范围是一个流；每个请求可包含分区键和记录的任何组合，直达到到请求限制。使用许多不同的分区键对具有许多不同分片的流进行的请求一般快于使用少量分区键对少量分片进行的请求。分区键的数量应远大于分片的数量以减少延迟并最大程度提高吞吐量。

PutRecords 示例

以下代码创建 100 个使用连续分区键的数据记录并将其放入名为 `DataStream` 的流中。

```
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` 响应包含响应 `Records` 的数组。响应数组中的每个记录按自然顺序（从请求和响应的顶部到底部）直接与请求数组中的一个记录关联。响应 `Records` 数组包含的记录数量始终与请求数组相同。

处理使用 PutRecords 时的失败问题

默认情况下，请求内的单个记录的失败不会中止对 PutRecords 请求中后续记录的处理。这意味着，响应 Records 数组包含处理成功和不成功的记录。您必须删除处理不成功的记录并在后续调用中包括它们。

成功的记录包括 SequenceNumber 和 ShardID 值，而不成功的记录包含 ErrorCode 和 ErrorMessage 值。ErrorCode 参数反映了错误类型，可能为下列值之一：ProvisionedThroughputExceededException 或 InternalFailure。ErrorMessage 提供有关 ProvisionedThroughputExceededException 异常的更多详细信息，包括账户 ID、流名称和已阻止的记录的分片 ID。以下示例在 PutRecords 请求中有三个记录。第二个记录失败并反映在响应中。

Example PutRecords 请求语法

```
{  
  "Records": [  
    {  
      "Data": "XzxkYXRhP18w",  
      "PartitionKey": "partitionKey1"  
    },  
    {  
      "Data": "AbceddeRFfg12asd",  
      "PartitionKey": "partitionKey1"  
    },  
    {  
      "Data": "KFpcd98*7nd1",  
      "PartitionKey": "partitionKey3"  
    }  
  "StreamName": "myStream"  
}
```

Example PutRecords 响应语法

```
{  
  "FailedRecordCount": 1,  
  "Records": [  
    {  
      "SequenceNumber": "21269319989900637946712965403778482371",  

```

```
  },
  {
    "ErrorCode": "ProvisionedThroughputExceeded",
    "ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream exampleStreamName under account 111111111111."
  },
  {
    "SequenceNumber": "2126931998999637946712965403778482985",
    "ShardId": "shardId-000000000002"
  }
]
```

处理不成功的请求可包含在后续 PutRecords 请求中。首先，查看 FailedRecordCount 中的 putRecordsResult 参数以确认请求中是否存在失败的记录。如果存在，则应将具有 putRecordsEntry (不是 ErrorCode) 的每个 null 添加到后续请求中。有关此类处理程序的示例，请参阅以下代码。

Example PutRecords 故障处理程序

```
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);
}
```

使用 PutRecord 添加单一记录

对 [PutRecord](#) 的每次调用对一个记录起作用。应首选 PutRecords 中描述的 [使用 PutRecords 添加多条记录](#) 操作，除非您的应用程序明确需要每一请求始终发送一条记录，或因某种其他原因无法使用 PutRecords。

每个数据记录都有一个唯一的序列号。此序列号在您调用 `client.putRecord` 向流添加数据记录之后由 Kinesis Data Streams 分配。同一分区键的序列号通常会随时间变化增加；PutRecord 请求之间的时间段越长，序列号变得越大。

在快速连续进行放置时，不保证返回的序列号会递增，因为放置操作的发生对于 Kinesis Data Streams 实际上是同步进行的。要确保相同分区键的序列号严格递增，请使用 `SequenceNumberForOrdering` 参数，如 [PutRecord 示例](#) 代码示例中所示。

无论您是否使用 `SequenceNumberForOrdering`，Kinesis Data Streams 通过 `GetRecords` 调用接收的记录都将按序列号严格进行排序。

Note

序列号不能用作相同流中的数据集的索引。为了在逻辑上分隔数据集，请使用分区键或者为每个数据集创建单独的流。

分区键用于对流中的数据进行分组。数据记录将基于其分区键分配给流中的分片。具体来说，Kinesis Data Streams 使用分区键作为将分区键（和关联数据）映射到特定分片的哈希函数的输入。

作为此哈希机制的结果，具有相同分区键的所有数据记录将映射到流中的同一分片。但是，如果分区键的数量超出分片的数量，则一些分片必定会包含具有不同分区键的记录。从设计的角度看，要确保您的所有分片得到充分利用，分片的数量（由 `setShardCount` 的 `CreateStreamRequest` 方法指定）应远少于唯一分区键的数量，并且流至单一分区键的数据量应远少于分片容量。

PutRecord 示例

以下代码创建跨两个分区键分配的 10 条数据记录，并将它们放入名为 myStreamName 的流中。

```
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();  
}
```

上一个代码示例使用 `setSequenceNumberForOrdering` 来确保每个分区键内的顺序严格递增。要有效使用此参数，请将当前记录（记录 n）的 `SequenceNumberForOrdering` 设置为前一条记录（记录 n-1）的序列号。要获取已添加到流的记录的序列号，请对 `getSequenceNumber` 的结果调用 `putRecord`。

`SequenceNumberForOrdering` 参数可确保相同分区键的序列号严格递增。`SequenceNumberForOrdering` 不提供跨多个分区键的记录排序。

使用 Amazon Glue 架构注册表与数据交互

您可以将 Kinesis 数据流与 Amazon Glue 架构注册表进行集成。Amazon Glue 架构注册表能帮助您集中发现、控制和演变架构，同时确保注册架构持续验证生成的数据。架构定义了数据记录的结构和格式。架构是用于可靠数据发布、使用或存储的版本化规范。通过 Amazon Glue 架构注册表，您能够改善流媒体应用程序中的端到端数据质量和数据治理。有关更多信息，请参阅 [Amazon Glue Schema Registry](#)。使用 Amazon Java 开发工具包中提供的 `PutRecords` 和 `PutRecord Kinesis Data Streams API` 可以设置此集成。

有关如何使用 `PutRecords` 和 `PutRecord Kinesis Data Streams API` 设置 Kinesis Data Streams 与架构注册表的集成的详细说明，请参阅 [Use Case: Integrating Amazon Kinesis Data Streams with the Amazon Glue Schema Registry](#) 中的“Interacting with Data Using the Kinesis Data Streams APIs”一节。

使用 Kinesis 代理写入 Amazon Kinesis Data Streams

Kinesis 代理是独立的 Java 软件应用程序，可提供更轻松的方式来收集数据并将数据发送到 Kinesis Data Streams。此代理持续监控一组文件，并将新数据发送到您的流。代理处理文件轮换、检查点操作并在失败时重试。它以可靠及时的简单方法提供您的所有数据。它还会发出 Amazon CloudWatch 指标，以帮助您更好地监控流处理并排除问题。

默认情况下，会基于换行符 ('\n') 分析每个文件中的记录。但是，也可以将代理配置为分析多行记录（请参阅 [指定代理配置设置](#)）。

您可以在基于 Linux 的服务器环境（如 Web 服务器、日志服务器和数据库服务器）上安装此代理。在安装代理后，通过指定要监控的日志文件和名称来配置代理。在配置好代理之后，代理将持续从这些文件中收集数据并以可靠的方式将数据发送到流。

主题

- [完成 Kinesis 代理的先决条件](#)
- [下载并安装代理](#)
- [配置并启动代理](#)
- [指定代理配置设置](#)
- [监控多个文件目录并写入多个流](#)
- [使用代理预处理数据](#)
- [使用代理 CLI 命令](#)
- [常见问题](#)

完成 Kinesis 代理的先决条件

- 您的操作系统必须是 Amazon Linux AMI 版本 2015.09 或更高版本，或者 Red Hat Enterprise Linux 版本 7 或更高版本。
- 如果您使用 Amazon EC2 运行代理，请启动 EC2 实例。
- 使用以下方法之一管理 Amazon 凭证：
 - 当您启动您的 EC2 实例时指定该 IAM 角色。
 - 当您配置代理时指定 Amazon 凭证（请参阅 [awsAccessKeyId](#) 和 [awsSecretAccessKey](#)）。
 - 编辑 `/etc/sysconfig/aws-kinesis-agent` 以指定区域和 Amazon 访问密钥。
 - 如果 EC2 实例位于其他 Amazon 账户中，请创建一个 IAM 角色来提供对 Kinesis Data Streams 服务的访问，并在配置代理时指定该角色（请参阅 [assumeRoleARN](#) 和 [assumeRoleWithAWSSTDToken](#)）。

[assumeRoleExternalId](#))。使用上述方法之一指定其他账户中有权担任该角色的用户的 Amazon 凭证。

- 您指定的 IAM 角色或 Amazon 凭证必须有权执行 Kinesis Data Streams [PutRecords](#) 操作，代理才能将数据发送到流。如果您为代理启用 CloudWatch 监控，则还需要拥有执行 CloudWatch [PutMetricData](#) 操作的权限。有关更多信息，请参阅 [使用 IAM 控制对 Amazon Kinesis Data Streams 资源的访问](#)、[使用 Amazon CloudWatch 监控 Kinesis Data Streams 代理运行状况](#) 和 [CloudWatch 访问控制](#)。

下载并安装代理

首先，连接到您的实例。有关详细信息，请参阅《Amazon EC2 用户指南》中的[连接到您的实例](#)。如果您在连接时遇到问题，请参阅《Amazon EC2 用户指南》中的[排查 Amazon EC2 Linux 实例的连接问题](#)。

使用 Amazon Linux AMI 设置代理

使用以下命令下载和安装代理：

```
sudo yum install -y aws-kinesis-agent
```

使用 Red Hat Enterprise Linux 设置代理

使用以下命令下载和安装代理：

```
sudo yum install -y https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-latest.amzn2.noarch.rpm
```

使用 GitHub 设置代理

- 从 [awlabs/amazon-kinesis-agent](#) 下载代理。
- 导航到下载目录并运行以下命令来安装代理：

```
sudo ./setup --install
```

在 Docker 容器中设置代理

Kinesis 代理可以在容器中运行，也可以通过 [amazonlinux](#) 容器库运行。使用以下 Dockerfile，然后运行 docker build。

```
FROM amazonlinux

RUN yum install -y aws-kinesis-agent which findutils
COPY agent.json /etc/aws-kinesis/agent.json

CMD ["start-aws-kinesis-agent"]
```

配置并启动代理

配置并启动代理

1. 打开并编辑配置文件（如果使用默认文件访问权限，则以超级用户的身份来执行）：`/etc/aws-kinesis/agent.json`

在此配置文件中，指定代理从中收集数据的文件（"filePattern"）以及代理将数据发送到的流的名称（"kinesisStream"）。请注意，文件名是一种模式，并且代理能够识别文件轮换。每秒内您轮换使用文件或创建新文件的次数不能超过一次。代理使用文件创建时间戳来确定要跟踪并送入您的流中的文件；如果每秒创建新文件或轮换使用文件的次数超过一次，代理将无法正确区分这些文件。

```
{
  "flows": [
    {
      "filePattern": "/tmp/app.log*",
      "kinesisStream": "yourkinesisstream"
    }
  ]
}
```

2. 手动启动代理：

```
sudo service aws-kinesis-agent start
```

- 3.（可选）将代理配置为在系统启动时启动：

```
sudo chkconfig aws-kinesis-agent on
```

现在，代理作为系统服务在后台运行。它会持续监控指定的文件，并将数据发送到指定的流。代理活动记录在 `/var/log/aws-kinesis-agent/aws-kinesis-agent.log` 中。

指定代理配置设置

代理支持两个必需的配置设置，即 `filePattern` 和 `kinesisStream`，以及可用于其他功能的可选配置设置。您可以在 `/etc/aws-kinesis/agent.json` 中指定必需配置和可选配置。

只要您更改配置文件，就必须使用以下命令停止再启动代理：

```
sudo service aws-kinesis-agent stop  
sudo service aws-kinesis-agent start
```

或者，您也可以使用以下命令：

```
sudo service aws-kinesis-agent restart
```

一般的设置配置如下。

| 配置设置 | 描述 |
|-------------------------------------|---|
| <code>assumeRoleARN</code> | 由该用户担任的角色的 ARN。有关更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色委托跨 Amazon 账户的访问权限 。 |
| <code>assumeRoleExternalId</code> | 确定谁可以担任该角色的可选标识符。有关更多信息，请参阅《IAM 用户指南》中的 如何使用外部 ID 。 |
| <code>awsAccessKeyId</code> | 覆盖默认凭证的 Amazon 访问密钥 ID。此设置优先于所有其他凭证提供程序。 |
| <code>awsSecretAccessKey</code> | 覆盖默认凭证的 Amazon 密钥。此设置优先于所有其他凭证提供程序。 |
| <code>cloudwatch.emitMetrics</code> | 如果设置为 <code>true</code> ，则允许代理向 CloudWatch 发出指标。 默认： <code>True</code> |
| <code>cloudwatch.endpoint</code> | CloudWatch 的区域端点。 默认值： <code>monitoring.us-east-1.amazonaws.com</code> |
| <code>kinesis.endpoint</code> | Kinesis Data Streams 的区域端点。 |

| 配置设置 | 描述 |
|------|---------------------------------------|
| | 默认值 : kinesis.us-east-1.amazonaws.com |

流配置设置如下。

| 配置设置 | 描述 |
|--------------------------|--|
| dataProcessingOptions | 在将每个被分析的记录发送到流之前应用于这些记录的处理选项的列表。这些处理选项按指定的顺序执行。有关更多信息，请参阅 使用代理预处理数据 。 |
| kinesisStream | [必需] 流名称。 |
| filePattern | [必需] 目录以及必须与之匹配的文件模式，以便被代理获取。对于与此模式匹配的所有文件，必须向 aws-kinesis-agent-user 授予读取权限。对于包含这些文件的目录，必须向 aws-kinesis-agent-user 授予读取和执行权限。 |
| initialPosition | 开始解析文件的初始位置。有效值为 START_OF_FILE 和 END_OF_FILE。 默认值 : END_OF_FILE |
| maxBufferAgeMillis | 代理在将数据发送到流之前缓冲数据的最长时间（以毫秒计）。 值范围 : 1000 到 900000 (1 秒到 15 分钟) 默认值 : 60,000 (1 分钟) |
| maxBufferSizeBytes | 代理在将数据发送到流之前缓冲的数据的最大大小（以字节计）。 值范围 : 1 到 4194304 (4 MB) 默认值 : 4194304 (4 MB) |
| maxBufferSizeModeRecords | 代理在将数据发送到流之前缓冲数据的最大记录数。 值范围 : 1 到 500 |

| 配置设置 | 描述 |
|-------------------------------|--|
| | 默认值 : 500 |
| minTimeBetweenFilePollsMillis | 代理轮询和分析受监控文件中是否有新数据的时间间隔 (以毫秒计) 。 值范围 : 1 或更大值 默认值 : 100 |
| multiLineStartPattern | 用于标识记录开始的模式。记录由与模式匹配的行以及与模式不匹配的任何以下行组成。有效值为正则表达式。默认情况下，日志文件中的每一个新行都被解析为一条记录。 |
| partitionKeyOption | 生成分区键的方法。有效值为 RANDOM (随机生成的整数) 和 DETERMINISTIC (基于数据计算出来的哈希值) 。 默认值 : RANDOM |
| skipHeaderLines | 代理从受监控文件开头跳过分析的行数。 值范围 : 0 或更大值 默认值 : 0 (零) |
| truncatedRecordTerminator | 代理在记录大小超过 Kinesis Data Streams 记录大小限制时用来截断已解析记录的字符串。 (1000 KB) 默认值 : '\n' (换行符) |

监控多个文件目录并写入多个流

通过指定多个流程配置设置，您可以配置代理以监控多个文件目录并将数据发送到多个流。在以下配置示例中，代理监控两个文件目录，并将数据分别发送到 Kinesis 流和 Firehose 传输流。请注意，您可以在 Kinesis Data Streams 和 Firehose 指定不同的端点，这样您的 Kinesis 流和 Firehose 传输流就不需要位于同一区域。

```
{
  "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"
  }
]
```

有关将代理与 Firehose 结合使用的更多详细信息，请参阅 [Writing to Amazon Data Firehose with Kinesis Agent](#)。

使用代理预处理数据

代理可以预处理从受监控文件分析的记录，然后再将这些记录发送到流。通过将 `dataProcessingOptions` 配置设置添加到您的文件流可以启用此功能。可以添加一个或多个处理选项，这些选项将按指定的顺序执行。

代理支持下面列出的处理选项。由于此代理是开源的，您可以进一步开发和扩展其处理选项。您可以从 [Kinesis 代理](#) 下载代理。

处理选项

SINGLELINE

通过移除换行符和首尾的空格，将多行记录转换为单行记录。

```
{
  "optionName": "SINGLELINE"
}
```

CSVTOJSON

将记录从分隔符分隔的格式转换为 JSON 格式。

```
{
  "optionName": "CSVTOJSON",
  "customFieldNames": [ "field1", "field2", ... ],
  "delimiter": "yourdelimiter"
}
```

}

customFieldNames

[必需] 在每个 JSON 键值对中用作键的字段名称。例如，如果您指定 `["f1", "f2"]`，则记录“v1, v2”将转换为 `{"f1": "v1", "f2": "v2"}`。

delimiter

在记录中用作分隔符的字符串。默认值为逗号 (,)。

LOGTOJSON

将记录从日志格式转换为 JSON 格式。支持的日志格式为 Apache Common Log、Apache Combined Log、Apache Error Log 和 RFC3164 Syslog。

```
{  
    "optionName": "LOGTOJSON",  
    "logFormat": "logformat",  
    "matchPattern": "yourregexpattern",  
    "customFieldNames": [ "field1", "field2", ... ]  
}
```

logFormat

[必需] 日志条目格式。以下是可能的值：

- COMMONAPACHELOG – Apache 通用日志格式。默认情况下每个日志条目都为以下模式：“`%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes}`”。
- COMBINEDAPACHELOG – Apache 组合日志格式。默认情况下每个日志条目都为以下模式：“`%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes} %{referrer} %{agent}`”。
- APACHEERRORLOG – Apache 错误日志格式。默认情况下每个日志条目都为以下模式：“`[%{timestamp}] [%{module}:%{severity}] [pid %{processid}:tid %{threadid}] [client: %{client}] %{message}`”。
- SYSLOG – RFC3164 系统日志格式。默认情况下每个日志条目都为以下模式：“`%{timestamp} %{hostname} %{program}[%{processid}]: %{message}`”。

matchPattern

用于从日志条目中提取值的正则表达式模式。如果您的日志条目不属于其中一种预定义日志格式，则将使用此设置。如果使用此设置，您还必须指定 `customFieldNames`。

customFieldNames

在每个 JSON 键值对中用作键的自定义字段名称。您可以使用此设置定义从 matchPattern 中提取的值的字段名称，或覆盖预定义日志格式的默认字段名称。

Example : LOGTOJSON 配置

下面是一个转换为 JSON 格式的 Apache 通用日志条目的 LOGTOJSON 配置示例：

```
{  
  "optionName": "LOGTOJSON",  
  "logFormat": "COMMONAPACHELOG"  
}
```

转换前：

```
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision  
HTTP/1.1" 200 6291
```

转换后：

```
{"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 : 包含自定义字段的 LOGTOJSON 配置

下面是 LOGTOJSON 配置的另一个示例：

```
{  
  "optionName": "LOGTOJSON",  
  "logFormat": "COMMONAPACHELOG",  
  "customFieldNames": ["f1", "f2", "f3", "f4", "f5", "f6", "f7"]  
}
```

使用此配置设置时，上一个示例中的同一个 Apache 通用日志条目将如下转换为 JSON 格式：

```
{"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 : 转换 Apache 通用日志条目

以下流配置将一个 Apache 通用日志条目转换为 JSON 格式的单行记录：

```
{  
  "flows": [  
    {  
      "filePattern": "/tmp/app.log*",  
      "kinesisStream": "my-stream",  
      "dataProcessingOptions": [  
        {  
          "optionName": "LOGTOJSON",  
          "logFormat": "COMMONAPACHELOG"  
        }  
      ]  
    }  
  ]  
}
```

Example : 转换多行记录

以下流配置分析第一行以“[SEQUENCE=”开头的多行记录。每个记录先转换为一个单行记录。然后，将基于制表分隔符从记录中提取值。提取的值映射到指定的 customFieldNames 值，从而形成 JSON 格式的单行记录。

```
{  
  "flows": [  
    {  
      "filePattern": "/tmp/app.log*",  
      "kinesisStream": "my-stream",  
      "multiLineStartPattern": "\\[SEQUENCE=",  
      "dataProcessingOptions": [  
        {  
          "optionName": "SINGLELINE"  
        },  
        {  
          "optionName": "CSVT0JSON",  
          "customFieldNames": [ "field1", "field2", "field3" ],  
          "delimiter": "\\t"  
        }  
      ]  
    }  
  ]  
}
```

}

Example：具有匹配模式的 LOGTOJSON 配置

下面是一个转换为 JSON 格式的 Apache 通用日志条目的 LOGTOJSON 配置示例，其中省略了最后一个字段 (bytes)：

```
{  
    "optionName": "LOGTOJSON",  
    "logFormat": "COMMONAPACHELOG",  
    "matchPattern": "^([\\d.]+) (\\S+) (\\S+) \\\\([([\\w:/]+\\s[+\\-]\\d{4}))\\\\] \"(.+?)\" (\\d{3})",  
    "customFieldNames": ["host", "ident", "authuser", "datetime", "request",  
    "response"]  
}
```

转换前：

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"  
200
```

转换后：

```
{"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"}
```

使用代理 CLI 命令

系统启动时自动启动代理：

```
sudo chkconfig aws-kinesis-agent on
```

检查代理的状态：

```
sudo service aws-kinesis-agent status
```

停止代理：

```
sudo service aws-kinesis-agent stop
```

从此位置读取代理的日志文件：

```
/var/log/aws-kinesis-agent/aws-kinesis-agent.log
```

卸载代理：

```
sudo yum remove aws-kinesis-agent
```

常见问题

有没有适用于 Windows 的 Kinesis 代理？

适用于 Windows 的 Kinesis 代理与适用于 Linux 平台的 Kinesis 代理是不同的软件。

为什么 Kinesis 代理速度变慢且/或 **RecordSendErrors** 增加？

这通常是由 Kinesis 节流造成的。查看 Kinesis Data Streams 的 `WriteProvisionedThroughputExceeded` 指标或 Firehose 传输流的 `ThrottledRecords` 指标。这些指标从 0 开始的任何增加都表示需要增加流限制。有关更多信息，请参阅 [Kinesis Data Stream limits](#) 和 [Amazon Firehose Delivery Streams](#)。

排除节流后，查看 Kinesis 代理是否配置为跟踪大量小文件。Kinesis 代理跟踪新文件时会有延迟，因此 Kinesis 代理应跟踪少量大文件。尝试将您的日志文件合并为大文件。

为什么我会收到 **java.lang.OutOfMemoryError** 异常？

Kinesis 代理没有足够的内存来处理当前的工作负载。尝试增加 `/usr/bin/start-aws-kinesis-agent` 中的 `JAVA_START_HEAP` 和 `JAVA_MAX_HEAP`，并重新启动代理。

为什么我会收到 **IllegalStateException : connection pool shut down** 异常？

Kinesis 代理没有足够的连接来处理当前的工作负载。尝试在 `/etc/aws-kinesis/agent.json` 的常规代理配置设置中增加 `maxConnections` 和 `maxSendingThreads`。这些字段的默认值是可用运行时系统处理器的 12 倍。有关高级代理配置设置的更多信息，请参阅 [AgentConfiguration.java](#)。

如何调试 Kinesis 代理的其他问题？

可在 `/etc/aws-kinesis/log4j.xml` 中启用 DEBUG 级别日志。

我应该如何配置 Kinesis 代理？

`maxBufferSizeBytes` 越小，Kinesis 代理发送数据的频率就越高。这可能是好事，因为这减少了记录的传输时间，但也增加了每秒对 Kinesis 的请求。

为什么 Kinesis 代理会发送重复记录？

这是由于文件跟踪中的错误配置造成的。确保每个 `fileFlow's filePattern` 只匹配一个文件。如果正在使用的 `logrotate` 模式处于 `copytruncate` 模式，也可能发生这种情况。尝试将模式更改为默认模式或创建模式以避免重复。有关处理重复记录的更多信息，请参阅[处理重复记录](#)。

使用其他 Amazon 服务写入 Kinesis Data Streams

以下 Amazon 服务可以直接与 Amazon Kinesis Data Streams 集成，以便向 Kinesis Data Streams 写入数据。查看您感兴趣的每项服务的信息，并参考所提供的参考资料。

主题

- [使用 Amazon Amplify 写入 Kinesis Data Streams](#)
- [使用 Amazon Aurora 写入 Kinesis Data Streams](#)
- [使用 Amazon CloudFront 写入 Kinesis Data Streams](#)
- [使用 Amazon CloudWatch Logs 写入 Kinesis Data Streams](#)
- [使用 Amazon Connect 写入 Kinesis Data Streams](#)
- [使用 Amazon Database Migration Service 写入 Kinesis Data Streams](#)
- [使用 Amazon DynamoDB 写入 Kinesis Data Streams](#)
- [使用 Amazon EventBridge 写入 Kinesis Data Streams](#)
- [使用 Amazon IoT Core 写入 Kinesis Data Streams](#)
- [使用 Amazon Relational Database Service 写入 Kinesis Data Streams](#)
- [使用 Amazon Pinpoint 写入 Kinesis Data Streams](#)
- [使用 Amazon Quantum Ledger Database \(Amazon QLDB \) 写入 Kinesis Data Streams](#)

使用 Amazon Amplify 写入 Kinesis Data Streams

您可以使用 Amazon Kinesis Data Streams 从使用 Amazon Amplify 构建的移动应用程序中流式传输数据，以进行实时处理。您可以构建实时控制面板，捕获异常并生成警报，推出建议并做出其他实时

业务或运营决策。您还可以将数据发送到其他服务中，如 Amazon Simple Storage Service、Amazon DynamoDB 和 Amazon Redshift。

有关更多信息，请参阅《Amazon Amplify Developer Center》中的 [Using Amazon Kinesis](#)。

使用 Amazon Aurora 写入 Kinesis Data Streams

您可以使用 Amazon Kinesis Data Streams 监控 Amazon Aurora 数据库集群的活动。使用数据库活动流，您的 Aurora 数据库集群能将活动实时推送到 Amazon Kinesis 数据流。然后，您可以为使用这些活动的合规性管理构建应用程序，对其进行审计并生成警报。您还可以使用 Amazon Firehose 存储数据。

有关更多信息，请参阅《Amazon Aurora 开发人员指南》中的[数据库活动流](#)。

使用 Amazon CloudFront 写入 Kinesis Data Streams

您可以将 Amazon Kinesis Data Streams 与 CloudFront 实时日志一起使用，并实时获取有关向分配发出的请求的信息。然后您可以构建自己的 [Kinesis 数据流消费端](#)，或使用 Amazon Data Firehose 将日志数据发送到 Amazon S3、Amazon Redshift、Amazon OpenSearch Service 或第三方日志处理服务。

有关更多信息，请参阅《Amazon CloudFront 开发人员指南》中的[实时日志](#)。

使用 Amazon CloudWatch Logs 写入 Kinesis Data Streams

您可以使用 CloudWatch 订阅来访问 Amazon CloudWatch Logs 中日志事件的实时源，并将其传输到 Kinesis 数据流以进行处理、分析或加载到其他系统中。

有关更多信息，请参阅《Amazon CloudWatch Logs 用户指南》中的[使用订阅实时处理日志数据](#)。

使用 Amazon Connect 写入 Kinesis Data Streams

您可以使用 Kinesis Data Streams 从 Amazon Connect 实例中实时导出联系记录和客服事件。您还可以启用来自 Amazon Connect Customer Profiles 的数据流，以自动接收 Kinesis 数据流中有关创建新个人资料或更改现有个人资料的更新。

然后，您可以构建消费端应用程序来实时处理和分析数据。例如，使用联系记录和客户资料数据，您可以使源系统数据（例如 CRM 和营销自动化工具）保持最新状态。使用客服事件数据，您可以创建显示客服信息和事件的控制面板，并触发特定客服活动的自定义通知。

有关更多信息，请参阅《Amazon Connect Administrator Guide》中的 [data streaming for your instance](#)、[set up real-time export](#) 和 [agent event streams](#)。

使用 Amazon Database Migration Service 写入 Kinesis Data Streams

您可以使用 Amazon Database Migration Service 将数据迁移到 Kinesis 数据流。然后，您可以构建实时处理数据记录的消费端应用程序。您还可以将数据轻松发送到下游的其他服务，如 Amazon Simple Storage Service、Amazon DynamoDB 和 Amazon Redshift。

有关更多信息，请参阅《Amazon Database Migration Service User Guide》中的 [Using Kinesis Data Streams](#)。

使用 Amazon DynamoDB 写入 Kinesis Data Streams

您可以使用 Amazon Kinesis Data Streams 捕获 Amazon DynamoDB 的更改。Kinesis Data Streams 捕获任何 DynamoDB 表中的项目级别修改，并将它们复制到 Kinesis Data Streams。您的消费端应用程序可以访问此流，以实时查看项目级别的更改，并将这些更改传送到下游或根据内容执行操作。

有关更多信息，请参阅《Amazon DynamoDB 开发人员指南》中的 [Kinesis Data Streams 如何与 DynamoDB 结合使用](#)。

使用 Amazon EventBridge 写入 Kinesis Data Streams

使用 Kinesis Data Streams，您可以将 EventBridge 中的 Amazon API 调用[事件](#)发送到流、构建消费端应用程序以及处理大量数据。您还可以在 EventBridge Pipes 中将 Kinesis Data Streams 作为目标，并在筛选和扩充（可选）之后，从可用源之一传送记录。

有关更多信息，请参阅《Amazon EventBridge User Guide》中的 [Send events to an Amazon Kinesis stream](#) 和 [EventBridge Pipes](#)。

使用 Amazon IoT Core 写入 Kinesis Data Streams

您可以使用 Amazon IoT Rule 操作，根据 MQTT 消息在 Amazon IoT Core 中实时写入数据。然后，您可以构建用于处理数据、分析其内容并生成警报的应用程序，然后将其传送给分析应用程序或其他 Amazon 服务。

有关更多信息，请参阅《Amazon IoT Core 开发人员指南》中的 [Kinesis Data Streams](#)。

使用 Amazon Relational Database Service 写入 Kinesis Data Streams

您可以使用 Amazon Kinesis Data Streams 监控 Amazon RDS 实例的活动。使用数据库活动流，Amazon RDS 能将活动实时推送到 Kinesis 数据流。然后，您可以为使用这些活动的合规性管理构建应用程序，对其进行审计并生成警报。您还可以使用 Amazon Data Firehose 存储数据。

有关更多信息，请参阅《Amazon RDS 开发人员指南》中的[数据库活动流](#)。

使用 Amazon Pinpoint 写入 Kinesis Data Streams

您可以将 Amazon Pinpoint 设置为向 Amazon Kinesis Data Streams 发送事件数据。Amazon Pinpoint 可以发送营销活动、旅程以及交易电子邮件和 SMS 消息的事件数据。然后，您可以将数据摄取到分析应用程序中，或者构建自己的消费端应用程序，这些应用程序会根据事件的内容执行操作。

有关更多信息，请参阅《Amazon Pinpoint Developer Guide》中的[Streaming Events](#)。

使用 Amazon Quantum Ledger Database (Amazon QLDB) 写入 Kinesis Data Streams

您可以在 Amazon QLDB 中创建一个流，其捕获提交到您日记账的每个文档修订版本，并将此数据实时传送到 Amazon Kinesis Data Streams。QLDB 流是从分类账的日记账到 Kinesis Data Streams 资源的连续数据流。然后，您可以使用 Kinesis 流平台或 Kinesis Client Library 来使用流、处理数据记录和分析数据内容。QLDB 流通过三种类型的记录将您的数据写入 Kinesis Data Streams：control、block summary 和 revision details。

有关更多信息，请参阅《Amazon QLDB developer Guide》中的[Streams](#)。

使用第三方集成写入 Kinesis Data Streams

您可以使用与 Kinesis Data Streams 集成的以下任一第三方选项，将数据写入 Kinesis Data Streams。选择您想详细了解的选项，然后查找资源和相关文档的链接。

主题

- [Apache Flink](#)
- [Fluentd](#)
- [Debezium](#)
- [Oracle GoldenGate](#)
- [Kafka Connect](#)
- [Adobe Experience](#)
- [Striim](#)

Apache Flink

Apache Flink 是一种开源框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。有关从 Apache Flink 写入 Kinesis Data Streams 的更多信息，请参阅 [Amazon Kinesis Data Streams Connector](#)。

Fluentd

Fluentd 是用于统一日志记录层的开源数据收集器。有关从 Fluentd 写入 Kinesis Data Streams 的更多信息，请参阅 [Stream processing with Kinesis](#)。

Debezium

Debezium 是一种用于更改数据捕获的开源分布式平台。有关从 Debezium 写入 Kinesis Data Streams 的更多信息，请参阅 [Streaming MySQL Data Changes to Amazon Kinesis](#)。

Oracle GoldenGate

Oracle GoldenGate 是一款软件产品，支持您将数据从一个数据库复制、筛选和转换到另一个数据库。有关从 Oracle GoldenGate 写入 Kinesis Data Streams 的更多信息，请参阅 [Data replication to Kinesis Data Stream using Oracle GoldenGate](#)。

Kafka Connect

Kafka Connect 是一种在 Apache Kafka 和其他系统之间以可扩展且可靠的方式流式传输数据的工具。有关将数据从 Apache Kafka 写入 Kinesis Data Streams 的更多信息，请参阅 [Kinesis kafka connector](#)。

Adobe Experience

Adobe Experience Platform 让组织能够集中管理和标准化来自任何系统的客户数据。然后，该平台应用数据科学和机器学习，显著改进丰富的个性化体验的设计和交付。有关将数据从 Adobe Experience Platform 写入 Kinesis Data Streams 的更多信息，请参阅如何创建 [Amazon Kinesis connection](#)。

Striim

Striim 是一个完整的端到端内存平台，用于实时收集、筛选、转换、扩充、聚合、分析和传输数据。有关如何将数据从 Striim 写入 Kinesis Data Streams 的更多信息，请参阅 [Kinesis Writer](#)。

Amazon Kinesis Data Streams 产生器问题排查

以下主题提供了 Amazon Kinesis Data Streams 产生器常见问题的解决方案：

- [产生器应用程序的写入速率比预期的慢](#)
- [我收到了未授权的 KMS 主密钥权限错误](#)
- [排查产生器的其他常见问题](#)

产生器应用程序的写入速率比预期的慢

写入吞吐量低于预期的最常见原因包括：

- [超过服务限制](#)
- [我想优化我的产生器](#)
- [滥用 flushSync\(\) 操作](#)

超过服务限制

要查明是否超过了服务限制，请检查您的创建器是否从服务引发了吞吐量异常，并验证哪些 API 操作受限制。请记住，根据调用会有不同的限制，具体请参阅[限额和限制](#)。例如，除了最广为人知的对读写操作的分片级别限制之外，还存在以下流级别的限制：

- [CreateStream](#)
- [DeleteStream](#)
- [ListStreams](#)
- [GetShardIterator](#)
- [MergeShards](#)
- [DescribeStream](#)
- [DescribeStreamSummary](#)

操作 CreateStream、DeleteStream、ListStreams、GetShardIterator 和 MergeShards 限制为每秒 5 个调用。DescribeStream 操作限制为每秒 10 个调用。DescribeStreamSummary 操作限制为每秒 20 个调用。

如果这些调用不存在问题，请确保您选择了允许在所有分片之间均匀分配 put 操作的分区键，并且没有某个特殊分区键无意中达到了服务限制而其他分区键则未达到限制。这要求您衡量高峰吞吐量并考虑流中的分片数量。有关管理流的详细信息，请参阅[创建和管理 Kinesis 数据流](#)。

Tip

请记住，使用单记录操作 [PutRecord](#) 时，将吞吐量节流计算值向上舍入到最接近的 KB 值；使用多记录操作 [PutRecords](#) 时，对各个调用中记录的累积总和进行舍入。例如，带有 600 个 1.1 KB 大小记录的 PutRecords 请求不会受到限制。

我想优化我的产生器

在您开始优化产生器之前，请完成以下关键任务。首先，根据记录大小和每秒记录数，确定您需要的高峰吞吐量。接下来，排除流容量作为限制因素 ([超过服务限制](#)) 的可能性。如果您已排除了流容量，对于两种常见类型的创建器，请使用以下故障排除提示和优化指南。

大型创建器

大型创建器通常从本地服务器或 Amazon EC2 实例运行。需要由大型创建器提供较高吞吐量的客户通常会关注每记录延迟。处理延迟的策略如下：如果客户可以微量批处理/缓冲记录，则使用 [Amazon Kinesis Producer Library](#) (具有高级聚合逻辑)、多记录操作 [PutRecords](#)，或者先将记录聚合到较大的文件中，再使用单记录操作 [PutRecord](#)。如果无法批处理/缓冲，则使用多个线程来同时写入到 Kinesis Data Streams 服务。适用于 Java 的 Amazon SDK 和其他开发工具包中包括异步客户端，可以用非常少的代码完成此任务。

小型创建器

小型创建器通常是移动应用程序、IoT 设备或 Web 客户端。如果是移动应用程序，我们建议使用 PutRecords 操作或 Amazon Mobile 开发工具包中的 Kinesis 记录器。有关更多信息，请参阅《适用于 Android 的 Amazon Mobile SDK 入门指南》和《Amazon Mobile SDK for iOS 入门指南》。移动应用程序自身必须处理断续连接，并且需要某种类型的批量 put，例如 PutRecords。如果由于某些原因而无法批处理，请参阅上面的大型创建器信息。如果您的创建器是浏览器，则生成的数据量通常非常小。不过，您将 put 操作放在了应用程序的关键路径上，则我们建议不要这样做。

滥用 `flushSync()` 操作

使用 `flushSync()` 不当会严重影响写入性能。`flushSync()` 操作专为关闭场景而设计，用于确保在 KPL 应用程序终止之前发出所有缓冲记录。如果在每次写入操作后都执行此操作，则可能会增加大

量额外的延迟，每次写入延迟约 500 毫秒。请确保仅在应用程序关闭时实现 `flushSync()`，以避免写入性能产生不必要的额外延迟。

我收到了未授权的 KMS 主密钥权限错误

当创建者应用程序在 KMS 主密钥上写入加密流但没有权限时，会发生此错误。要为应用程序分配权限以访问 KMS 密钥，请参阅 [Using Key Policies in Amazon KMS](#) 和 [Using IAM Policies with Amazon KMS](#)。

排查产生器的其他常见问题

- [为什么我的 Kinesis 数据流会返回一个 500 内部服务器错误？](#)
- [我如何排查从 Flink 写入 Kinesis Data Streams 时发生的超时错误？](#)
- [如何排查 Kinesis Data Streams 中的节流错误问题？](#)
- [为什么我的 Kinesis 数据流会节流？](#)
- [如何使用 KPL 将数据记录放入 Kinesis 数据流中？](#)

优化 Kinesis Data Streams 产生器

根据您看到的特定行为，您可以进一步优化 Amazon Kinesis Data Streams 产生器。请查看以下主题以确定解决方案。

主题

- [自定义 KPL 重试和速率限制行为](#)
- [将最佳实践应用于 KPL 聚合](#)

自定义 KPL 重试和速率限制行为

当您使用 KCL `addUserRecord()` 操作添加 Amazon Kinesis Producer Library (KPL) 用户记录时，将利用由 `RecordMaxBufferedTime` 配置参数设置的截止日期为记录提供一个时间戳并将记录添加到缓冲区。此时间戳/截止日期组合将设置缓冲区优先级。记录基于以下标准从缓冲区进行刷新：

- 缓冲区优先级
- 聚合配置
- 集合配置

影响缓冲区行为的聚合和集合配置参数如下所示：

- AggregationMaxCount
- AggregationMaxSize
- CollectionMaxCount
- CollectionMaxSize

然后，通过调用 Kinesis Data Streams API 操作 PutRecords，刷新的记录将作为 Amazon Kinesis Data Streams 记录发送到您的 Kinesis 数据流。PutRecords 操作将请求发送到有时完全失败或部分失败的流。失败的记录将自动添加回 KPL 缓冲区。基于这两个值中的较小值设置新的截止日期：

- 将当前 RecordMaxBufferedTime 配置分为两半
- 记录的生存时间值

借助此策略，重试的 KPL 用户记录可以包含在后续 Kinesis Data Streams API 调用中，从而在强制实施 Kinesis Data Streams 记录的生存时间值时提高吞吐量并降低复杂性。不存在退避算法，这使得该策略成为相对积极的重试策略。由于重试次数过多而造成的垃圾邮件会受到速率限制的阻止，该内容将在下一部分中进行讨论。

速率限制

KPL 包括速率限制功能，该功能可限制从单个创建器发送的每个分片的吞吐量。使用令牌桶算法以及同时适用于 Kinesis Data Streams 记录和字节的单独存储桶来实施速率限制。每当对 Kinesis 数据流进行成功写入时都会将一个令牌（或多个令牌）添加到每个存储桶，直至达到特定阈值。此阈值是可配置的，但默认情况下设定的值将比实际分片限制高出 50%，这将允许单个创建器中的分片饱和。

您可降低此限制来减少因重试次数过多而造成的垃圾邮件。然而，最佳实践是每个创建器主动重试最大吞吐量，通过扩展流的容量并实施相应的分区键策略来处理已确定为过多的任何结果限制。

将最佳实践应用于 KPL 聚合

当结果 Amazon Kinesis Data Streams 记录的序列号架构保持相同时，聚合会形成包含在以 0（零）为开始的聚合 Kinesis Data Streams 记录的 Amazon Kinesis Producer Library（KPL）用户记录的索引；然而，只要您不依赖序列号来唯一标识您的 KPL 用户记录，您的代码可忽略它，因为聚合（属于您的 Kinesis Data Streams 记录中的 KPL 用户记录）和后续取消聚合（属于您的 KPL 用户记录中的 Kinesis Data Streams 记录）将自动为您考虑这一方面。无论消费端是使用 KCL 还是 Amazon 开发工

具包，这一点都适用。要使用此聚合功能，您需要将 KPL 的 Java 部分拉入您的版本中（如果您的消费端是利用 Amazon 开发工具包中提供的 API 编写的）。

如果您打算使用序列号作为 KPL 用户记录的唯一标识符，建议您使用 `Record` 和 `UserRecord` 中提供的遵守合约的 `public int hashCode()` 和 `public boolean equals(Object obj)` 操作来比较您的 KPL 用户记录。此外，如果您想要检查 KPL 用户记录的子序列号，则可将其转换为 `UserRecord` 实例并检索其子序列号。

有关更多信息，请参阅 [实现消费端取消聚合](#)。

从 Amazon Kinesis Data Streams 读取数据

消费端 是一种处理 Kinesis 数据流中所有数据的应用程序。当使用者使用增强的扇出时，它会获得自己的 2 个读取吞吐量 MB/sec 分配，从而允许多个使用者并行读取来自同一流的数据，而不必与其他使用者争夺读取吞吐量。要使用分片的增强型扇出功能，请参阅[开发具有专用吞吐量的增强扇出型消费端](#)。

您可以使用 Kinesis Client Library (KCL) 或 适用于 Java 的 Amazon SDK 为 Kinesis Data Streams 构建消费端。您还可以使用其他Amazon服务来培养消费者Amazon Lambda，例如适用于 Apache Flink 的亚马逊托管服务和 Amazon Data Firehose。Kinesis Data Streams 支持与亚马逊 EMR、EventBridge亚马逊Amazon Glue和亚马逊 Redshift Amazon 等其他服务的集成。它还支持第三方集成，包括 Apache Flink、Adobe 体验平台、Apache Druid、Apache Spark、Databricks、Confluent 平台、Kinesumer 和 Talend。

主题

- [开发具有专用吞吐量的增强扇出型消费端](#)
- [在 Kinesis 控制台中使用数据查看器](#)
- [在 Kinesis 控制台中查询您的数据流](#)
- [使用 Kinesis Client Library](#)
- [使用 适用于 Java 的 Amazon SDK 开发消费端](#)
- [使用以下方法培养消费者 Amazon Lambda](#)
- [使用适用于 Apache Flink 的亚马逊托管服务来开发消费端](#)
- [使用 Amazon Data Firehose 开发消费端](#)
- [使用其他 Amazon 服务从 Kinesis Data Streams 读取数据](#)
- [使用第三方集成从 Kinesis Data Streams 中读取](#)
- [Kinesis Data Streams 消费端问题排查](#)
- [优化 Amazon Kinesis Data Streams 消费端](#)

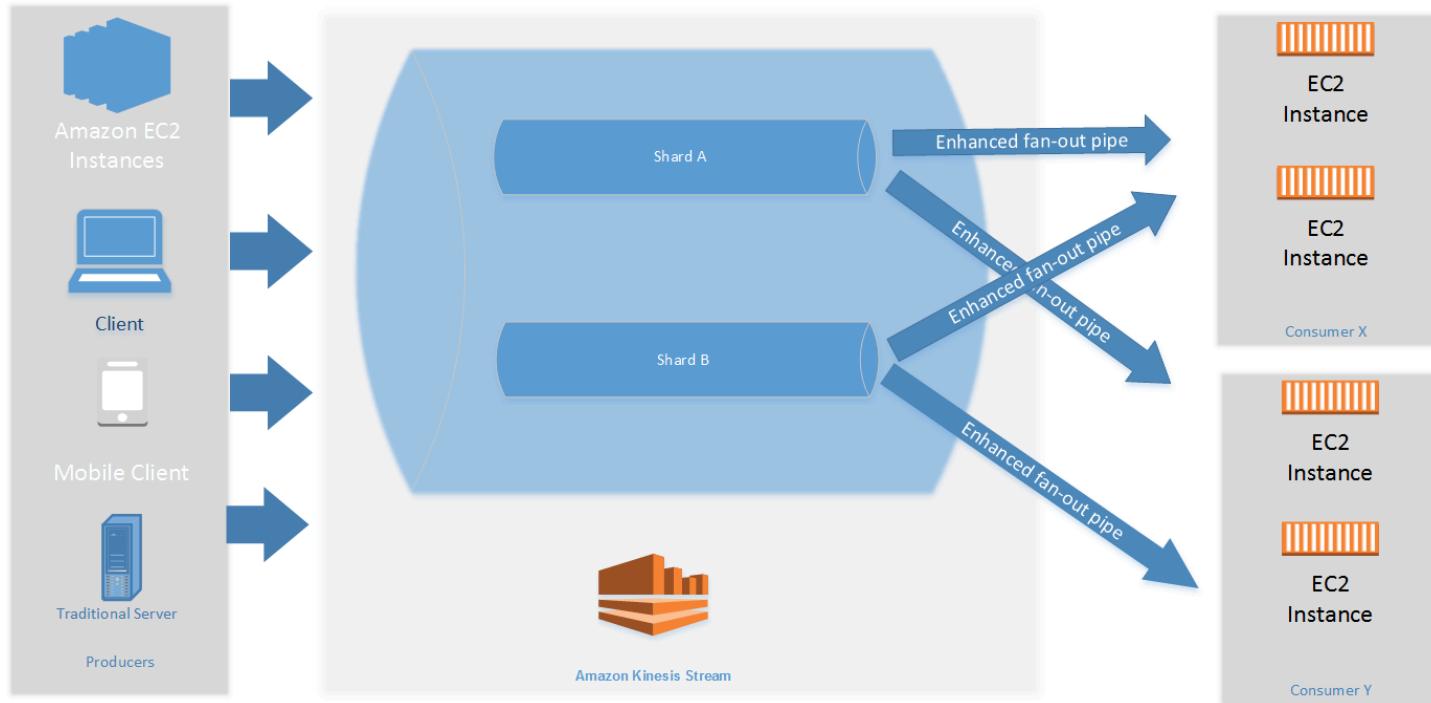
开发具有专用吞吐量的增强扇出型消费端

在 Amazon Kinesis Data Streams 中，可以构建使用增强型扇出功能的消费端。利用此功能，消费端可以从流中接收记录，其数据吞吐量高达每分片 2 MB/秒。此为专用吞吐量，这意味着，使用增强型扇出功能的消费端不必与接收流中数据的其他消费端争夺。Kinesis Data Streams 将流中的数据记录推送到使用增强型扇出功能的消费端。因此，这些消费端无需轮询数据。

⚠ Important

在按需优势模式下，您可以为每个直播注册多达 50 个消费者，以使用增强的扇出功能。使用按需标准和预配置直播，您可以为每个直播注册最多 20 个使用者，以使用增强的扇出功能。

下图显示的是增强型扇出功能架构。如果使用 2.0 版或更高版本的 Amazon Kinesis Client Library (KCL) 构建消费端，则 KCL 会将消费端设置为使用增强型扇出功能接收来自流的所有分片的数据。如果使用 API 构建使用增强型扇出功能的消费端，则可订阅单个分片。



此图显示以下内容：

- 一个具有两个分片的流。
- 使用增强型扇出功能接收流中数据的两个消费端：消费端 X 和消费端 Y。这两个消费端均已订阅流的所有分片和所有记录。如果使用 2.0 版或更高版本的 KCL 构建消费端，则 KCL 将自动为消费端订阅流的所有分片。另一方面，如果使用 API 构建消费端，则可订阅单个分片。
- 表示消费端用于接收流中数据的增强型扇出功能管道的箭头。增强的扇出管道为每个分片提供最多 2 MB/sec 个数据，与任何其他管道或使用者总数无关。

主题

- [共享吞吐量消费端和增强扇出型消费端之间的区别](#)

- [支持最多 50 个增强型扇出消费者的区域（仅限按需优势）](#)
- [使用或管理增强的扇出消费者 Amazon CLI APIs](#)

共享吞吐量消费端和增强扇出型消费端之间的区别

下表将默认共享吞吐量与增强型扇出型消费端进行比较。消息传播延迟定义为使用负载调度（如和）发送的有效负载通过消耗负载（如和）到达使用者应用程序所花费的时间 APIs（如PutRecord和PutRecords），以毫秒为单位。 APIs GetRecords SubscribeToShard

下表比较了共享吞吐量消费端与增强扇出型消费端

| 特性 | 无增强扇出功能的共享吞吐量消费端 | 增强扇出型消费端 |
|--------|--|--|
| 读取吞吐量 | 固定为 MB/sec 每个碎片总共有 2 个。如果有多个消费端正在从同一分片进行读取，则它们将全部共享此吞吐量。它们从分片中接收的吞吐量总和不会超出 2 MB/秒。 | 随着消费端注册进行扩展以使用增强型扇出功能。注册为使用增强型扇出功能的每个消费端均接收其自己的每个分片的读取吞吐量，最多 2MB/秒，独立于其他消费端。 |
| 消息传播延迟 | 平均约 200 毫秒（如果您有一个从流中读取的消费端）。如果您有五个消费端，则这个平均值高达约 1000 毫秒。 | 通常情况下，平均为 70 毫秒，无论您是拥有一个消费端，还是五个消费端。 |
| 成本 | 不适用 | 存在数据检索费用和消费端分片小时费用。有关更多信息，请参阅 Amazon Kinesis Data Streams 定价 。 |
| 记录传输模型 | 使用通过 HTTP 提取模型 GetRecords。 | Kinesis Data Streams 使用通过 HTTP/ SubscribeToShard 2 将记录推送给您。 |

支持最多 50 个增强型扇出消费者的区域（仅限按需优势）

在按需优势模式下，支持最多 50 个增强型扇出使用者仅在以下 Amazon 区域提供：

| Amazon区域 | 区域名称 |
|----------------|------------------|
| eu-north-1 | 欧洲地区 (斯德哥尔摩) |
| me-south-1 | 中东 (巴林) |
| ap-south-1 | 亚太地区 (孟买) |
| eu-west-3 | 欧洲地区 (巴黎) |
| ap-southeast-3 | 亚太地区 (雅加达) |
| us-east-2 | 美国东部 (俄亥俄州) |
| af-south-1 | 非洲 (开普敦) |
| eu-west-1 | 欧洲地区 (爱尔兰) |
| me-central-1 | 中东 (阿联酋) : |
| eu-central-1 | 欧洲地区 (法兰克福) |
| sa-east-1 | 南美洲 (圣保罗) |
| ap-east-1 | 亚太地区 (香港) |
| ap-south-2 | 亚太地区 (海得拉巴) |
| us-east-1 | 美国东部 (弗吉尼亚州北部) |
| ap-northeast-2 | 亚太地区 (首尔) |
| ap-northeast-3 | 亚太地区 (大阪) |
| eu-west-2 | 欧洲地区 (伦敦) |
| ap-southeast-4 | 亚太地区 (墨尔本) |
| ap-northeast-1 | 亚太地区 (东京) |
| us-west-2 | 美国西部 (俄勒冈州) |

| Amazon区域 | 区域名称 |
|----------------|-------------------------|
| us-west-1 | 美国西部 (北加利福尼亚) |
| ap-southeast-1 | 亚太地区 (新加坡) |
| ap-southeast-2 | 亚太地区 (悉尼) |
| il-central-1 | 以色列 (特拉维夫) |
| ca-central-1 | 加拿大 (中部) |
| ca-west-1 | 加拿大西部 (卡尔加里) |
| eu-south-2 | 欧洲 (西班牙) |
| cn-northwest-1 | 中国 (宁夏) |
| eu-central-2 | 欧洲 (苏黎世) |
| us-gov-east-1 | AmazonGovCloud (美国东部) |
| us-gov-west-1 | AmazonGovCloud (美国西部) |

使用或管理增强的扇出消费者 Amazon CLI APIs

在 Amazon Kinesis Data Streams 中使用增强型扇出功能的消费端，可以接收数据流中的记录，其中每分片每秒专用吞吐量高达 2MB 数据。有关更多信息，请参阅 [开发具有专用吞吐量的增强扇出型消费端](#)。

您可以使用Amazon CLI或 Kinesis Data APIs Streams 注册、描述、列出和注销在 Kinesis Data Streams 中使用增强型扇出功能的使用者。

使用管理消费者 Amazon CLI

您可以使用注册、描述、列出和取消注册增强型扇出消费者。Amazon CLI有关示例，请参阅以下文档。

[register-stream-consumer](#)

为 Kinesis 数据流注册消费端。可以在注册消费端时应用标签。

[describe-stream-consumer](#)

通过消费端 ARN 或消费端名称与流 ARN 组合来获取注册消费端的描述。

[list-stream-consumers](#)

列出使用增强扇出功能注册从流中接收数据的消费端。

[deregister-stream-consumer](#)

通过消费端 ARN 或消费端名称与流 ARN 组合来取消注册消费端。

使用 Kinesis Data Streams 管理消费者 APIs

您可以使用 Kinesis Data Streams 注册、描述、列出和取消注册增强型扇出使用者。 APIs 有关示例，请参阅以下文档。

[RegisterStreamConsumer](#)

在带有标签的 Kinesis 数据流中注册消费端。可以在注册消费端时应用标签。

[DescribeStreamConsumer](#)

通过消费端 ARN 或消费端名称与流 ARN 组合来获取注册消费端的描述。

[ListStreamConsumers](#)

列出使用增强扇出功能注册从流中接收数据的消费端。

[DeregisterStreamConsumer](#)

通过消费端 ARN 或消费端名称与流 ARN 组合来取消注册消费端。

标记消费端

您可以将自己的元数据以标签的形式分配给您在 Kinesis Data Streams 中创建的流和增强扇出型消费端。您可以使用标签对消费端成本进行分类和跟踪。还可以使用标签和[基于属性的访问权限控制 \(ABAC\)](#) 来控制对消费端的访问权限。有关更多信息，请参阅[标记 Amazon Kinesis Data Streams 资源](#)。

在 Kinesis 控制台中使用数据查看器

通过 Kinesis 管理控制台中的数据查看器，您无需开发消费端应用程序，即可查看数据流中特定分片内的数据记录。要使用数据查看器，请按照下列步骤操作：

1. [登录Amazon Web Services 管理控制台并在 /kinesis 上打开 Kinesis 控制台。<https://console.aws.amazon.com>](https://console.aws.amazon.com)
2. 选择要使用数据查看器查看其记录的活动数据流，然后选择数据查看器选项卡。
3. 在所选活动数据流的数据查看器选项卡中，选择要查看其记录的分片，选择起始位置，然后单击获取记录。您可以将起始位置设置为以下值之一：
 - 在序列号处：从序列号字段中指定的序列号表示的位置显示记录。
 - 序列号之后：在序列号字段中指定的序列号表示的位置之后显示记录。
 - 在时间戳处：从时间戳字段中指定的时间戳表示的位置显示记录。
 - 裁剪范围：显示分片中最后一条未裁剪记录处的记录，即分片中最早的数据记录。
 - 最新记录：显示分片中最新记录之后的记录，以便您能始终读取分片中的最新数据。

然后，生成的与指定分片 ID 和起始位置相匹配的数据记录将显示在控制台的记录表中。一次最多显示 50 条记录。要查看下一组记录，请单击下一步按钮。

4. 单击任意一条记录，即可在单独的窗口中以原始数据或 JSON 格式查看该记录的负载。

请注意，当您单击 Data Viewer 中的“获取记录”或“下一步”按钮时，这会调用 GetRecordsAPI，这适用于每秒 5 个事务的 GetRecordsAPI 限制。

在 Kinesis 控制台中查询您的数据流

通过 Kinesis Data Streams 控制台的“数据分析”选项卡，可以使用 SQL 查询数据流。要使用此功能，请按照下列步骤操作：

1. [登录Amazon Web Services 管理控制台并在 /kinesis 上打开 Kinesis 控制台。<https://console.aws.amazon.com>](https://console.aws.amazon.com)
2. 选择要使用 SQL 查询的活动数据流，然后选择数据分析选项卡。
3. 在数据分析选项卡中，可以使用托管式 Apache Flink Studio 笔记本执行流检查和可视化。您可以使用 Apache Zeppelin 执行临时 SQL 查询，以检查数据流并在几秒钟内查看结果。在数据分析选项卡中，选择我同意，然后选择创建笔记本，以创建笔记本。
4. 创建笔记本后，选择在 Apache Zeppelin 中打开。这将在新选项卡中打开您的笔记本。笔记本是一个交互式界面，您可以在其中提交 SQL 查询。选择包含您的流的名称的笔记。
5. 您将看到一个笔记，其中包含一个示例 SELECT 查询，用于输出已在运行的流中的数据。您可以由此查看数据流的架构。

6. 要尝试其他查询，例如翻滚或滑动窗口，请在数据分析选项卡中选择查看示例查询。复制查询，对其进行修改以适应您的数据流架构，然后在 Zeppelin 笔记中的新段落中运行它。

使用 Kinesis Client Library

什么是 Kinesis Client Library？

Kinesis Client Library (KCL) 是独立的 Java 软件库，其旨在简化来自 Amazon Kinesis Data Streams 的数据的使用和处理过程。KCL 负责处理与分布式计算相关的多种复杂任务，这样开发人员就可以专注于实现其数据处理业务逻辑。KCL 可管理的活动包括在多个工作程序之间进行负载均衡、响应工作程序故障、已处理记录的检查点操作，以及响应流中分片数量的变化。

KCL 经常更新，以纳入新版底层库、安全改进和错误修复。建议使用最新版本的 KCL，以避免出现已知问题，并从所有最新的改进中受益。要查找最新的 KCL 版本，请访问 [KCL Github](#)。

Important

- 建议使用最新的 KCL 版本，以避免出现已知错误和问题。如果您使用的是 KCL 2.6.0 或更早版本，请升级到 KCL 2.6.1 或更高版本，以避免在流容量变化时出现分片处理受阻，但这种情况比较罕见。
- KCL 属于 Java 库。使用名为的基于 Java 的守护程序提供对 Java 以外其他语言的支持。MultiLangDaemon MultiLangDaemon 通过 STDIN 和 STDOUT 与 KCL 应用程序交互。有关 MultiLangDaemon 的更多信息 GitHub，请参阅 [使用非 Java 语言通过 KCL 开发消费端](#)。
- 不要在 KCL 3.x 中使用 2.27.19 到 2.27.23 适用于 Java 的 Amazon SDK 版本。这些版本出现的问题会导致使用 KCL 的 DynamoDB 时出现相关异常错误。我们建议您使用 2.28.0 或更高适用于 Java 的 Amazon SDK 版本来避免此问题。

KCL 主要功能和优势

以下是 KCL 的主要功能和相关优势：

- 可扩展性：KCL 支持应用程序在多个工作程序之间分配处理负载，以实现动态扩展。您可以手动横向缩减或扩展应用程序，也可以自动扩缩，而不必担心负载的重新分配。
- 负载均衡：KCL 在可用工作程序之间自动平衡处理负载，从而实现跨工作程序的均匀工作分配。
- 检查点：KCL 对已处理记录的检查点操作进行管理，使应用程序能够从上次成功处理的位置恢复处理。

- 容错能力：KCL 有内置容错能力，即使个别工作程序出现故障，也能确保数据处理继续进行。KCL 还提供 at-least-once 送货服务。
- 处理流级别的变化：KCL 可以适应由于数据量变化而可能发生的分片拆分与合并。KCL 通过确保只有在父分片完成处理并进行检查点操作后才处理子分片，从而保持排序状态。
- 监控：KCL 与 Amazon 集成，CloudWatch 用于消费者级监控。
- 多语言支持：KCL 原生支持 Java，并支持多种非 Java 编程语言。MultiLangDaemon

KCL 概念

本节说明了 Kinesis Client Library (KCL) 的核心概念和交互功能。这些概念是开发和管理 KCL 消费端应用程序的基础。

- KCL 消费端应用程序：使用 Kinesis Client Library 自定义构建的应用程序，旨在读取和处理 Kinesis 数据流中的记录。
- 工作程序：KCL 消费端应用程序通常是分布式的，同时运行一个或多个工作程序。KCL 协调工作程序以分布式方式使用来自流的数据，并在多个工作程序之间平衡负载。
- 调度器：KCL 工作程序用于开始处理数据的高级类。每个 KCL 工作程序都有一个调度器。调度器负责初始化和监督各种任务，包括同步 Kinesis 数据流的分片信息、跟踪工作程序中的分片分配以及根据分配给工作程序的分片处理来自流的数据。调度器可以采用多种影响调度器行为的配置，例如待处理流的名称以及 Amazon 凭证。调度器启动数据记录传输，使其从流传输至记录处理器。
- 记录处理器：定义 KCL 消费端应用程序如何处理从数据流中接收的数据的逻辑。必须在记录处理器中实现您自己的自定义数据处理逻辑。KCL 工作程序可实例化调度器。然后，调度器为持有租约的每个分片实例化一个记录处理器。一个工作程序可以运行多个记录处理器。
- 租约：定义工作程序和分片之间的分配。KCL 消费端应用程序使用租约将数据记录分配到多个工作程序中。每个分片在任何给定时间仅通过租约与一个工作程序绑定，而每个工作程序可以同时持有一份或多份租约。当工作程序因停止或故障而不再持有租约时，KCL 会指派其他工作程序承接租约。要了解有关租约的更多信息，请参阅 [Github documentation: Lease Lifecycle](#)。
- 租约表：唯一的 Amazon DynamoDB 表，用于跟踪 KCL 消费端应用程序的所有租约。每个 KCL 消费端应用程序都会创建自己的租约表。租约表用于跨工作程序维持状态，以协调数据的处理。有关更多信息，请参阅 [KCL 中的 DynamoDB 元数据表和负载均衡](#)。
- 检查点：将最后一次成功处理的记录的位置永久存储于分片中的过程。KCL 管理检查点操作，以确保在工作程序故障或应用程序重新启动时，可以从最后一个检查点操作位置恢复处理。检查点作为租约元数据的一部分存储于 DynamoDB 租约表中。这样工作程序就可以从前一个工作程序停止处继续处理。

KCL 中的 DynamoDB 元数据表和负载均衡

KCL 管理来自工作程序的租约和 CPU 利用率指标等元数据。KCL 使用 DynamoDB 表跟踪这些元数据。对于每个 Amazon Kinesis Data Streams 应用程序，KCL 会创建三个 DynamoDB 表来管理元数据：租约表、工作程序指标表和协调器状态表。

Note

KCL 3.x 引入了两个新的元数据表：工作程序指标和协调器状态表。

Important

必须为 KCL 应用程序添加适当的权限，才能在 DynamoDB 中创建和管理元数据表。有关更多信息，请参阅 [KCL 消费端应用程序所必需的 IAM 权限](#)。

KCL 消费端应用程序不会自动移除这三个 DynamoDB 元数据表。在停用消费端应用程序时，务必移除这些由 KCL 消费端应用程序创建的 DynamoDB 元数据表，以避免不必要的成本。

租约表

租约表是唯一的 Amazon DynamoDB 表，用于跟踪由 KCL 消费端应用程序的调度器租赁和处理的分片。每个 KCL 消费端应用程序都会创建自己的租约表。KCL 默认将消费端应用程序的名称用作租约表的名称。可使用配置来设置自定义表名称。KCL 还使用 leaseOwner 的分区键在租约表上创建[全局二级索引](#)，以高效发现租约。全局二级索引镜像了基础租约表中的 leaseKey 属性。如果应用程序启动时 KCL 消费端应用程序的租约表不存在，则其中一个工作程序会为您的应用程序创建租约表。

您可在消费端应用程序运行的同时使用 [Amazon DynamoDB 控制台](#) 查看租约表。

Important

- 每个 KCL 消费端应用程序的名称必须是唯一的，以防止租约表名称出现重复。
- 除开与 Kinesis Data Streams 本身关联的费用，您的账户将被收取与 DynamoDB 表关联的费用。

租约表中的每行表示您消费端应用程序的调度器正在处理的分片。重要字段包括：

- `leaseKey`：对于单流处理，此为分片 ID。使用 KCL 进行多流处理时，其结构为 `account-id:StreamName:streamCreationTimestamp:ShardId`。`leaseKey` 是租约表的分区键。有关多流处理的更多信息，请参阅[使用 KCL 进行多流处理](#)。
- `checkpoint`：分片的最新检查点序号。
- `checkpointSubSequence` 数字：使用 Kinesis Producer 库的聚合功能时，这是对检查点的扩展，用于跟踪 Kinesis 记录中的单个用户记录。
- `leaseCounter`：用于检查工作程序当前是否正在积极处理租约。如果租约所有权转移给其他工作程序，`leaseCounter` 就会增加。
- `leaseOwner`：持有此租约的当前工作程序。
- `ownerSwitchesSince` 检查点：自上次检查点以来，这份租约更换了多少次员工。
- `parentShardId`：此分片的父级 ID。在子分片上开始处理之前，务必确保父分片已完全处理，从而保持正确的记录处理顺序。
- `childShardId`：此分片的拆分或合并 IDs 产生的子分片列表。用于在重新分片操作期间跟踪分片世系并管理处理顺序。
- `startingHashKey`：此分片的哈希键范围的下限。
- `endingHashKey`：此分片的哈希键范围的上限。

如果使用 KCL 进行多流处理，您会在租约表中看到以下两个额外字段。有关更多信息，请参阅[使用 KCL 进行多流处理](#)。

- `shardID`：分片的 ID。
- `streamName`：数据流的标识符采用以下格式：`account-id:StreamName:streamCreationTimestamp`。

工作程序指标表

工作程序指标表是各个 KCL 应用程序唯一的 Amazon DynamoDB 表，用于记录各工作程序的 CPU 利用率指标。KCL 使用这些指标进行高效的租约分配，从而在工作程序之间实现资源利用的平衡。默认情况下，KCL 使用 `KCLApplicationName-WorkerMetricStats` 作为工作程序指标表的名称。

协调器状态表

协调器状态表是各个 KCL 应用程序唯一的 Amazon DynamoDB 表，用于存储工作程序的内部状态信息。例如，协调器状态表存储有关领导选择的数据，或从 KCL 2.x 就地迁移至 KCL 3.x 的相关元数

据。默认情况下，KCL 使用 `KCLApplicationName-CoordinatorState` 作为协调器状态表的名称。

KCL 创建的元数据表的 DynamoDB 容量模式

默认情况下，Kinesis Client Library (KCL) 使用 [按需容量模式](#) 创建 DynamoDB 元数据表，例如租约表、工作程序指标表和协调器状态表。此模式可自动扩缩读取和写入容量以适应流量，而无需进行容量规划。我们强烈建议将容量模式保留为按需模式，以便更有效地操作这些元数据表。

如果决定将租约表切换到 [预置容量模式](#)，请遵循以下最佳实践：

- 分析使用模式：

- 使用 Amazon 指标监控应用程序的读写模式和使用情况 (RCU、WCU)。 CloudWatch
- 了解峰值及平均吞吐量需求。

- 计算所需的容量：

- 根据您的分析估算读取容量单位 (RCUsWCUs) 和写入容量单位 ()。
- 考虑诸如分片数量、检查点频率和工作程序计数之类的因素。

- 实现自动扩缩：

- 使用 [DynamoDB 自动扩缩](#) 自动调整预置容量，并设置适当的最小和最大容量限制。
- DynamoDB 自动扩缩有助于避免 KCL 元数据表达到容量限制和受到限制。

- 定期监控和优化：

- 持续监控的 CloudWatch 指标 `ThrottledRequests`。
- 随着时间推移，按照工作负载的变化调整容量。

如果在 KCL 消费端应用程序的元数据 DynamoDB 表中遇到

`ProvisionedThroughputExceededException`，必须增加 DynamoDB 表的预置吞吐能力。如在首次创建消费端应用程序时设置了某种级别的读取容量单位 (RCU) 和写入容量单位 (WCU)，随着使用量的增长，这种容量可能无法满足需求。例如，如果 KCL 消费端应用程序频繁执行检查点操作或在包含多个分片的流上运行，则可能需要更多的容量单位。有关 DynamoDB 中预置吞吐量的信息，请参阅《Amazon DynamoDB 开发人员指南》中的 [DynamoDB 吞吐能力](#) 和 [更新表](#)。

KCL 如何向工作程序分配租约并平衡负载

KCL 不断收集和监控运行工作程序的计算主机的 CPU 利用率指标，以确保工作负载均匀分配。这些 CPU 利用率指标存储于 DynamoDB 的工作程序指标表中。如果 KCL 检测到某些工作程序的 CPU 利用率高于其他工作程序，它会在工作程序之间重新分配租约，以降低高使用率工作程序的负载。目标是

在消费端应用程序队列中更均匀地平衡工作负载，防止任何单一工作程序过载。由于 KCL 在消费端应用程序队列中分配 CPU 利用率，所以可以通过选择适当数量的工作程序来适当调整消费端应用程序队列容量的大小，或者使用自动扩缩功能来高效管理计算容量以降低成本。

Important

只有在满足某些先决条件的情况下，KCL 才能从工作程序收集 CPU 利用率指标。有关更多信息，请参阅 [先决条件](#)。如果 KCL 无法从工作程序收集 CPU 利用率指标，KCL 将回退到使用每个工作程序的吞吐量来分配租约，并在队列中的工作程序之间平衡负载。KCL 将监控每个工作程序在给定时间收到的吞吐量并重新分配租约，以确保每个工作程序从其被分配的租约中获得类似的总吞吐量水平。

使用 KCL 开发消费端

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的消费端应用程序。

KCL 有多种语言版本。本主题介绍如何使用 Java 和非 Java 语言开发 KCL 消费端。

- 要查看 Kinesis Client Library Javadoc 参考，请参阅 [Amazon Kinesis Client Library Javadoc](#)。
- 要从中下载适用于 Java 的 KCL GitHub，请参阅适用于 Java 的 [Amazon Kinesis 客户端库](#)。
- 要在 Apache Maven 上找到 KCL for Java，请参阅 [KCL Maven Central 存储库](#)。

主题

- [使用 Java 通过 KCL 开发消费端](#)
- [使用非 Java 语言通过 KCL 开发消费端](#)

使用 Java 通过 KCL 开发消费端

先决条件

开始使用 KCL 3.x 之前，请确保已具备以下条件：

- Java Development Kit (JDK) 8 或更高版本
- 适用于 Java 的 Amazon SDK2.x
- 用于依赖项管理的 Maven 或 Gradle

KCL 从运行工作程序的计算主机上收集 CPU 利用率指标（例如 CPU 利用率）来平衡负载，从而在各工作程序之间实现均衡的资源利用率水平。要让 KCL 能够从工作程序收集 CPU 使用率指标，必须满足以下先决条件：

Amazon Elastic Compute Cloud (亚马逊 EC2)

- 操作系统必须是 Linux 操作系统。
- 您必须在您的 EC2 实例[IMDSv2](#)中启用。

亚马逊上的亚马逊弹性容器服务 (Amazon ECS) Container Service EC2

- 操作系统必须是 Linux 操作系统。
- 必须启用 [ECS 任务元数据端点版本 4](#)。
- Amazon ECS 容器代理版本必须为 1.39.0 或更高版本。

Amazon ECS 已开启 Amazon Fargate

- 必须启用 [Fargate 任务元数据端点版本 4](#)。如果使用的是 Fargate 平台版本 1.4.0 或更高版本，则默认启用此功能。
- Fargate (平台版本 1.4.0 或更高版本)。

亚马逊上的亚马逊 Elastic Kubernetes Service (亚马逊 EKS) EC2

- 操作系统必须是 Linux 操作系统。

亚马逊 EKS 开启 Amazon Fargate

- Fargate (平台版本 1.3.0 或更高版本)。

Important

如果 KCL 无法从工作程序收集 CPU 利用率指标，KCL 将回退到使用每个工作程序的吞吐量来分配租约，并在队列中的工作程序之间平衡负载。有关更多信息，请参阅 [KCL 如何向工作程序分配租约并平衡负载](#)。

安装并添加依赖项

如果您使用的是 Maven，请将以下依赖项添加到您的 `pom.xml` 文件中。确保将 `3.x.x` 替换为最新的 KCL 版本。

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

如果使用 Gradle，请在 `build.gradle` 文件中添加以下信息。确保将 `3.x.x` 替换为最新的 KCL 版本。

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

可以在 [Maven Central 存储库](#) 中查看最新版本的 KCL。

实现消费端

KCL 消费端应用程序包含以下关键组件：

关键组件

- [RecordProcessor](#)
- [RecordProcessorFactory](#)
- [调度器](#)
- [主消费端应用程序](#)

RecordProcessor

RecordProcessor 是处理 Kinesis 数据流记录的业务逻辑所在的核心组件。它定义了应用程序如何处理从 Kinesis 流接收的数据。

主要职责：

- 初始化分片的处理
- 处理来自 Kinesis 流的批量记录
- 关闭分片的处理（例如，在分片拆分或合并，或者将租约移交给另一台主机时）
- 处理检查点操作以跟踪进度

以下示例演示如何实施。

```
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);
        }
    }
}
```

下面详细说明了本例中使用的各种方法：

初始化 (初InitializationInput始化输入)

- 目的：为处理记录设置任何必要的资源或状态。
- 调用时间：当 KCL 为该记录处理器分配分片时，调用一次。
- 关键点：
 - `initializationInput.shardId()`：此处理器将要处理的分片的 ID。
 - `initializationInput.extendedSequenceNumber()`：开始处理的序列号。

流程记录 () `ProcessRecordsInput` `processRecordsInput`

- 目的：处理传入的记录，可选择处理检查点进度。
- 调用时间：只要记录处理器持有分片的租约时，就反复调用。
- 关键点：
 - `processRecordsInput.records()`：要处理的记录列表。
 - `processRecordsInput.checkpointer()`：用于进度的检查点操作。
 - 确保在处理过程中处理了所有异常，以防止 KCL 出现故障。
 - 该方法应该具有幂等性，因为在某些情况下，同一条记录可能会处理多次，例如在工作程序意外崩溃或重启之前尚未进行检查点操作的数据。
 - 在进行检查点操作之前，务必刷新任何缓存数据，以确保数据一致性。

LeaseLost () `LeaseLostInput` `leaseLostInput`

- 目的：清理用于处理此分片的所有特定资源。
- 调用时间：当其他调度器接管此分片的租约时。
- 关键点：
 - 该方法不允许进行检查点操作。

ShardEnded () `ShardEndedInput` `shardEndedInput`

- 目的：完成此分片的处理并进行检查点操作。
- 调用时间：当分片拆分或合并时，表示该分片的所有数据都已处理完毕。
- 关键点：
 - `shardEndedInput.checkpointer()`：用于执行最终的检查点操作。
 - 该方法必须进行检查点操作才能完成处理。
 - 若此处未刷新数据和进行检查点操作，可能会导致分片重新打开时出现数据丢失或重复处理。

已请求关机 () ShutdownRequestedInput shutdownRequestedInput

- 目的：在 KCL 关闭时进行检查点操作并清理资源。
- 调用时间：当 KCL 关闭时，例如，在应用程序终止时。
- 关键点：
 - `shutdownRequestedInput.checkpointer()`：用于在关闭前执行检查点操作。
 - 确保在该方法中进行了检查点操作，以便在应用程序停止之前保存进度。
 - 若此处未刷新数据和进行检查点操作，可能会导致在应用程序重新启动时出现数据丢失或重新处理记录。

Important

KCL 3.x 通过在前一个工作程序关闭前进行检查点操作，确保在租约从一个工作程序移交给另一个工作程序时减少数据重复处理。如果未在 `shutdownRequested()` 方法中实现检查点操作逻辑，就无法体验到这一好处。请确保已在 `shutdownRequested()` 方法中实现了检查点操作逻辑。

RecordProcessorFactory

`RecordProcessorFactory` 负责创建新 `RecordProcessor` 实例。KCL 使用此工厂 `RecordProcessor` 为应用程序需要处理的每个分片创建一个新分片。

主要职责：

- 按需创建新 `RecordProcessor` 实例
- 确保每个都已 `RecordProcessor` 正确初始化

以下是一个实施示例：

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

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

}

在此示例中，`SampleRecordProcessor` 每次调用 `shardRecordProcessor()` 时，工厂都会创建一个新的。您可以进行扩展以添加任何必要的初始化逻辑。

调度器

调度器是一个协调 KCL 应用程序所有活动的高级组件。调度器负责数据处理的总体编排。

主要职责：

- 管理生命周期 RecordProcessors
- 处理分片的租约管理
- 协调检查点操作
- 在应用程序的多个工作程序之间平衡分片处理负载
- 处理正常关闭和应用程序终止信号

调度器通常在主应用程序中创建和启动。您可以在下一节“主消费端应用程序”中查看调度器的实现示例。

主消费端应用程序

主消费端应用程序将所有组件联系在一起。它负责设置 KCL 消费端、创建必要的客户端、配置调度器和管理应用程序的生命周期。

主要职责：

- 设置Amazon服务客户端（Kinesis、DynamoDB 等）CloudWatch
- 配置 KCL 应用程序
- 创建并启动调度器
- 处理应用程序关闭

以下是一个实施示例：

```
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();
    }
}
```

默认情况下，KCL 会创建一个具有专用吞吐量的增强型扇出型 (EFO) 消费端。有关增强扇出功能的更多信息，请参阅[开发具有专用吞吐量的增强扇出型消费端](#)。如果消费端少于 2 个，或者不需要低于 200 毫秒的读取传播延迟，则必须在调度器对象中设置以下配置来使用共享吞吐量消费端：

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

以下代码是一个创建使用共享吞吐量消费端的调度器对象的示例：

进口：

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

代码：

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

使用非 Java 语言通过 KCL 开发消费端

本节介绍如何在 Python、Node.js、.NET 和 Ruby 中实现使用 Kinesis Client Library (KCL) 的消费端。

KCL 属于 Java 库。使用名为 MultiLangDaemon 的多语言接口提供对 Java 以外语言的支持。此进程守护程序基于 Java，当您使用 Java 以外语言的 KCL 时，该程序会在后台运行。因此，如果您安装了适用于非 Java 语言的 KCL 并完全在非 Java 语言中编写消费端应用程序，则由于 MultiLangDaemon，您仍需要在您的系统中安装 Java。此外，MultiLangDaemon 存在部分默认设置，您可能需要根据自己的使用案例自定义此类设置（例如所连接到的 AWS 区域）。有关 MultiLangDaemon 的更多信息 GitHub，请参阅 [KCL MultiLangDaemon 项目](#)。

虽然各语言的核心概念一致，但也有一些针对具体语言的注意事项和实现。有关 KCL 消费端开发的核心概念，请参阅 [使用 Java 通过 KCL 开发消费端](#)。有关如何在 Python、Node.js、.NET 和 Ruby 中开发 KCL 使用者的更多详细信息以及最新更新，请参阅以下 GitHub 存储库：

- Python : [amazon-kinesis-client-python](#)
- Node.js : [amazon-kinesis-client-nodejs](#)
- .NET : [amazon-kinesis-client-net](#)
- Ruby : [amazon-kinesis-client-ruby](#)

Important

如果使用的是 JDK 8，请勿使用以下非 Java KCL 库版本。这些版本包含与 JDK 8 不兼容的依赖项（logback）。

- KCL Python 3.0.2 和 2.2.0
- KCL Node.js 2.3.0
- KCL .NET 3.1.0
- KCL Ruby 2.2.0

使用 JDK 8 时，我们建议使用在这些受影响版本之前或之后发布的版本。

使用 KCL 进行多流处理

本节介绍了 KCL 中所需的更改，这些更改让您能够创建可同时处理多个数据流的 KCL 消费端应用程序。

⚠ Important

- 只有 KCL 2.3 或更高版本才支持多流处理功能。
- 使用非 Java 语言编写的通过 multilangdaemon 运行的 KCL 消费端不支持多流处理功能。
- 任何版本的 KCL 1.x 都不支持多流处理功能。

• MultistreamTracker 接口

- 要构建可以同时处理多个流的使用者应用程序，必须实现一个名为的新接口 [MultistreamTracker](#)。此接口包括返回要由 KCL 消费端应用程序处理的数据流及其配置列表的 streamConfigList 方法。请注意，正在处理的数据流可以在消费端应用程序运行时进行更改。KCL 会定期调用 streamConfigList 来了解要处理的数据流的变化。
- streamConfigList 填充 [StreamConfig](#) 列表。

```
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 和 InitialPositionInStreamExtended 是必填字段，consumerArn 是选填字段。只有在使用 KCL 实现增强型扇出消费端应用程序时，才必须提供 consumerArn。
- 有关的更多信息 StreamIdentifier，请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/v2.5.8/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L129>。要创建 StreamIdentifier，我们建议您从 KCL 2.5.0 或更高版本提供的 streamArn 和 streamCreationEpoch 创建一个多流实例。在不支持 streamArn 的 KCL v2.3 和 v2.4 中，请使用格式 account-id:StreamName:streamCreationTimestamp 创建一个多流实例。从下一个主要版本开始，此格式将弃用且不再受支持。

- MultiStreamTracker 还包括删除租赁表中旧直播租约的策略 (formerStreamsLeasesDeletionStrategy)。请注意，在消费端应用程序运行时无法更改策略。欲了解更多信息，请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0.java#L111>。
-

或者，MultiStreamTracker如果要实现同时处理多个流的 KCL 使用者应用程序，也可以使用进行初始化 ConfigsBuilder。

```
* 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;
}
```

- 通过为您的 KCL 消费端应用程序实现多流支持功能，应用程序租约表的每一行现在都包含该应用程序处理的多个数据流的分片 ID 和流名称。

- 在为 KCL 消费端应用程序实现多流支持功能后，leaseKey 采用以下结构：account-id:StreamName:streamCreationTimestamp:ShardId。例如 111111111:multiStreamTest-1:12345:shardId-000000000336。

Important

当现有 KCL 消费端应用程序配置为仅处理一个数据流时，leaseKey（租约表的分区键）就是分片 ID。如果您将此现有 KCL 消费端应用程序重新配置为处理多个数据流，则会破坏租约表，因为 leaseKey 必须采用如下结构才能支持多流功能：account-id:StreamName:StreamCreationTimestamp:ShardId。

在 KC Amazon Glue L 中使用架构注册表

你可以将 Kinesis Data Streams 与架构注册表集成到 Amazon Glue。Amazon Glue 架构注册表允许您集中发现、控制和演变架构，同时确保生成的数据由注册架构持续验证。架构定义了数据记录的结构和格式。架构是用于可靠数据发布、使用或存储的版本化规范。借助 Amazon Glue Schema 注册表，您可以改善流媒体应用程序中的 end-to-end 数据质量和数据治理。有关更多信息，请参阅 [Amazon Glue 架构注册表](#)。设置此集成的方法之一是使用适用于 Java 的 KCL。

Important

- Amazon Glue 只有 KCL 2.3 或更高版本支持 Kinesis Data Streams 的架构注册表集成。
- Amazon Glue 使用运行的非 Java 语言编写的 KCL 使用者不支持 Kinesis Data Streams 的架构注册表集成。multilangdaemon
- Amazon Glue 任何版本的 KCL 1.x 都不支持 Kinesis Data Streams 的架构注册表集成。

有关如何使用 KCL 设置 Kinesis 数据流与 Amazon Glue 架构注册表集成的详细说明，请参阅“[用例：将 Amazon Kinesis 数据流与架构注册表集成](#)”中的“[使用 KPL/KCL 库与数据交互](#)”部分。Amazon Glue

KCL 消费端应用程序所必需的 IAM 权限

必须向与 KCL 消费端应用程序有关的 IAM 角色或用户添加以下权限。

的安全最佳实践Amazon要求使用细粒度的权限来控制对不同资源的访问权限。 Amazon Identity and Access Management(IAM) 允许您在中管理用户和用户权限Amazon。 IAM policy 明确列出了允许的操作以及这些操作适用于的资源。

下表显示了 KCL 消费端应用程序通常需要的最低 IAM 权限：

KCL 消费端应用程序的最低 IAM 权限

| 服务 | 操作 | 资源 (ARNs) | 用途 |
|-----------------------------|------------------------|--|---|
| Amazon Kinesis Data Streams | DescribeStream | KCL 应用程序从中处理数据的 Kinesis 数据流。 | 在尝试读取记录前，消费端会检查数据流是否存在，数据流是否处于活动状态，以及分片是否包含在数据流中。 |
| | DescribeStreamSummary | arn:aws:kinesis:region:account:stream/StreamName | |
| | RegisterStreamConsumer | | 将消费端注册到分片。 |
| Amazon Kinesis Data Streams | GetRecords | KCL 应用程序从中处理数据的 Kinesis 数据流。 | 从分片读取记录。 |
| | GetShardIterator | arn:aws:kinesis:region:account:stream/StreamName | |
| | ListShards | | |
| Amazon Kinesis Data Streams | SubscribeToShard | KCL 应用程序从中处理数据的 Kinesis 数据流。只有在使用增强扇出型 (EFO) 消费端时才添加此操作。 | 为增强型扇出 (EFO) 消费端订阅分片。 |
| | DescribeStreamConsumer | arn:aws:kinesis:region:account:stream/StreamName | |

| 服务 | 操作 | 资源 (ARNs) | 用途 |
|-----------------|---|--|---|
| | | StreamName/ consumer/* | |
| Amazon DynamoDB | CreateTable DescribeTable UpdateTable Scan GetItem PutItem UpdateItem DeleteItem | 租约表 (KCL 创建的 DynamoDB 中的元数 据表) 。 arn:aws:d ynamodb:r egion:acc ount:tabl e/KCLAppl icationName | KCL 需要执行这 些操作才能管理 DynamoDB 中创建的 租约表。 |

| 服务 | 操作 | 资源 (ARNs) | 用途 |
|-----------------|--|--|---|
| Amazon DynamoDB | CreateTable DescribeTable Scan GetItem PutItem UpdateItem DeleteItem | KCL 创建的工作程序指标和协调器状态表 (DynamoDB 中的元数据表)。 arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorStats | KCL 需要这些操作才能管理 DynamoDB 中的工作程序指标和协调器状态元数据表。 |
| Amazon DynamoDB | Query | 租约表上的全局二级索引。 arn:aws:dynamodb:region:account:table/KCLApplicationName/index/* | KCL 需要执行此操作才能读取 DynamoDB 中创建的租约表的全局二级索引。 |

| 服务 | 操作 | 资源 (ARNs) | 用途 |
|-------------------|---------------|-----------|--|
| Amazon CloudWatch | PutMetricData | * | 上传指标对于监控应用程序非常有用。CloudWatch 之所以使用星号(*)，是因为没有用于调用PutMetricData 操作 CloudWatch 的特定资源。 |

Note

将中的“区域”、“账户”、“” 和“KCLApplication名称”分别替换为您自己的Amazon Web Services 账户号码Amazon Web Services 区域、Kinesis 数据流名称和 KCL 应用程序名称。StreamName ARNsKCL 3.x 在 DynamoDB 中又创建了两个元数据表。有关 KCL 创建的 DynamoDB 元数据表的详细信息，请参阅[KCL 中的 DynamoDB 元数据表和负载均衡](#)。如果使用配置来自定义 KCL 创建的元数据表的名称，请使用这些指定的表名称而不是 KCL 应用程序名称。

下面是 KCL 消费端应用程序的策略文档示例。

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": "*"
}
]
```

在使用该策略示例之前，检查以下项：

- 将区域替换为你的Amazon Web Services 区域（例如 us-east-1）。
- 将账号_ID替换为你的Amazon Web Services 账户账号。
- 将 STREAM_NAME 替换为 Kinesis 数据流的名称。
- 使用 KCL 时，请将 CONSUMER_NAME 替换为消费端的名称，这通常是应用程序的名称。
- 将 KCL_APPLICATION_NAME 替换为 KCL 应用程序的名称。

KCL 配置

您可以设置配置属性来自定义 Kinesis Client Library 的功能，以满足具体要求。下表列明了配置属性和类。

⚠ Important

在 KCL 3.x 中，负载均衡算法的目标是在各工作程序之间实现均匀的 CPU 使用率，而不是使各个工作程序的租约数量相等。如果 `maxLeasesForWorker` 设置得太低，可能会限制 KCL 有效平衡工作负载的能力。如果使用 `maxLeasesForWorker` 配置，可考虑增加其值以实现最佳的负载分布。

本表显示了 KCL 的配置属性

| 配置属性 | 配置类 | 说明 | 默认值 |
|-----------------------------------|------------------------------------|--|----------------|
| <code>applicationName</code> | <code>ConfigsBuilder</code> | 此 KCL 应用程序的名称。用作 <code>tableName</code> 和 <code>consumerName</code> 的默认名称。 | 不适用 |
| <code>tableName</code> | <code>ConfigsBuilder</code> | 允许覆盖用于 Amazon DynamoDB 租赁表的表名称。 | 不适用 |
| <code>streamName</code> | <code>ConfigsBuilder</code> | 此应用程序从其中处理记录的流的名称。 | 不适用 |
| <code>workerIdentifier</code> | <code>ConfigsBuilder</code> | 表示应用程序处理器的这种实例化的唯一标识符。此值必须唯一。 | 不适用 |
| <code>failoverTimeInMillis</code> | <code>LeaseManagementConfig</code> | 在您可以将租赁所有者视为已失败之前必须经过的毫秒数。对于拥有大量分片的应用程序，可以将分片数设置为更高的数字，以减少跟踪租约所需的 DynamoDB IOPS 数。 | 10000 (10 秒) |

| 配置属性 | 配置类 | 说明 | 默认值 |
|----------------------------------|-----------------------|--|----------------|
| shardSyncIntervalMillis | LeaseManagementConfig | 分片同步调用之间的时间。 | 60000 (60 秒) |
| cleanupLeasesUponShardCompletion | LeaseManagementConfig | 如果设置，只要子租赁已开始处理，即可删除租赁。 | TRUE |
| ignoreUnexpectedChildShards | LeaseManagementConfig | 如果设置，将忽略具有打开的分片的子分片。这主要适用于 DynamoDB Streams。 | FALSE |
| maxLeasesForWorker | LeaseManagementConfig | 单个工作程序应接受的最大租约数。如果设置太低，则在工作程序无法处理所有分片时可能会导致数据丢失，并造成工作程序之间的租约分配不够理想。在进行配置时，应考虑总分片数、工作程序数和工作程序的处理能力。 | 无限制 |
| maxLeaseRenewalThreads | LeaseManagementConfig | 控制租赁续订线程池的大小。您的应用程序可以容纳的租赁越多，此池应该就越大。 | 20 |

| 配置属性 | 配置类 | 说明 | 默认值 |
|--------------------------------|-----------------------|---|------------------------|
| billingMode | LeaseManagementConfig | 确定 DynamoDB 中创建的租约表的容量模式。容量模式有两个选项：按需模式 (PAY_PER_REQUEST) 和预置模式。我们建议使用默认的按需模式设置，因为这种模式可以自动扩缩以适应工作负载，而无需进行容量规划。 | PAY_PER_REQUEST (按需模式) |
| initialLeaseTableReadCapacity | LeaseManagementConfig | 如果 Kinesis Client Library 需要使用预置的容量模式创建新的 DynamoDB 租约表，DynamoDB 将读取使用的容量。如果在 billingMode 配置中使用默认的按需容量模式，可以忽略上述配置。 | 10 |
| initialLeaseTableWriteCapacity | LeaseManagementConfig | 如果 Kinesis Client Library 需要创建新的 DynamoDB 租约表，DynamoDB 将读取使用的容量。如果在 billingMode 配置中使用默认的按需容量模式，可以忽略上述配置。 | 10 |

| 配置属性 | 配置类 | 说明 | 默认值 |
|---------------------------------|-----------------------|---|------------------------------------|
| initialPositionInStreamExtended | LeaseManagementConfig | 应用程序应在流中开始的初始位置。此值仅在创建初始租赁时使用。 | InitialPositionInStream.trim_HORIZ |
| reBalanceThresholdPercentage | LeaseManagementConfig | 用于确定负载均衡算法何时应考虑在工作程序之间重新分配分片的一个百分比值。 这是 KCL 3.x 中引入的新配置。 | 10 |
| dampeningPercentage | LeaseManagementConfig | 用于抑制单次再平衡操作中将从超载工作程序转移的负载量的一个百分比值。 这是 KCL 3.x 中引入的新配置。 | 60 |
| allowThroughputOvershoot | LeaseManagementConfig | 确定是否仍需要从超载工作程序获得额外的租约，即使这会导致占用的总租约吞吐量超过所需的吞吐量。 这是 KCL 3.x 中引入的新配置。 | TRUE |

| 配置属性 | 配置类 | 说明 | 默认值 |
|--------------------------|-----------------------|--|-------|
| disableWorkerMetrics | LeaseManagementConfig | <p>确定在重新分配租约和负载均衡时，KCL 是否应忽略工作程序的资源指标（例如 CPU 利用率）。如果要阻止 KCL 根据 CPU 利用率进行负载均衡，则将其设置为 TRUE。</p> <p>这是 KCL 3.x 中引入的新配置。</p> | FALSE |
| maxThroughputPerHostKbps | LeaseManagementConfig | <p>在租约分配期间分配给工作程序的最大吞吐量。</p> <p>这是 KCL 3.x 中引入的新配置。</p> | 无限制 |

| 配置属性 | 配置类 | 说明 | 默认值 |
|-------------------------------|-----------------------|--|------|
| isGracefulLeaseHandoffEnabled | LeaseManagementConfig | <p>控制工作程序之间的租约移交行为。设置为 true 时，KCL 将尝试通过在将租约移交给其他工作人员之前让分片 RecordProcessor 有足够的时间完成处理来优雅地转移租约。这有助于确保数据完整性和平稳过渡，但可能会增加移交时间。</p> <p>如果设置为 false，则租约将立即移交，无需等待优雅 RecordProcessor 地关闭。这可以加快移交速度，但可能存在处理不完全的风险。</p> <p>注意：检查点必须在的 shutdownRequested () 方法中实现，RecordProcessor 才能从优雅的租赁移交功能中受益。</p> <p>这是 KCL 3.x 中引入的新配置。</p> | TRUE |

| 配置属性 | 配置类 | 说明 | 默认值 |
|-------------------------------------|-----------------------|--|----------------|
| gracefulLeaveHandoffTimeoutInMillis | LeaseManagementConfig | <p>指定在强制将租约转让给下一个所有者之前，等待当前分片正常关闭的最短时间（RecordProcessor以毫秒为单位）。</p> <p>如果 processRecords方法的典型运行时间比默认值长，可考虑增加此设置。这样可以确保在租赁转让发生之前 RecordProcessor 有足够的时间完成其处理。</p> <p>这是 KCL 3.x 中引入的新配置。</p> | 30000 (30 秒) |
| maxRecords | PollingConfig | 允许设置 Kinesis 返回的最大记录数。 | 10000 |
| retryGetRecordsInSeconds | PollingConfig | 配置 GetRecords 尝试失败之间的延迟。 | 无 |
| maxGetRecordsThreadPool | PollingConfig | 使用的线程池大小 GetRecords。 | 无 |
| idleTimeBetweenReadsInMillis | PollingConfig | 确定 KCL 在两次 GetRecords 调用轮询数据流数据之间等待多长时间。单位为毫秒。 | 1500 |

| 配置属性 | 配置类 | 说明 | 默认值 |
|---|-------------------|--|--------------------------|
| callProcessRecordsEvenForEmptyRecordList | ProcessorConfig | 如果设置，即使 Kinesis 中未提供任何记录，也会调用记录处理器。 | FALSE |
| parentShardPollIntervalMillis | CoordinatorConfig | 记录处理器应轮询多少时间才能查看是否已完成父分片。单位为毫秒。 | 10000 (10 秒) |
| skipShardSyncAtWorkerInitializationIfLeaseExist | CoordinatorConfig | 如果租赁表包含现有租赁，请禁用同步的分片数据。 | FALSE |
| shardPrioritization | CoordinatorConfig | 要使用的分片优先级。 | NoOpShardPrioritization |
| ClientVersionConfig | CoordinatorConfig | 确定应用程序将运行在哪个 KCL 版本兼容模式下。此配置仅适用于从之前的 KCL 版本进行迁移的情况。迁移至 3.x 时，需要将该配置设置为 CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X。完成迁移后，即可删除此配置。 | CLIENT_VERSION_CONFIG_3X |

| 配置属性 | 配置类 | 说明 | 默认值 |
|-------------------------------|-----------------|---|------------------|
| taskBackoffTimeInMillis | LifecycleConfig | 等待重试 KCL 失败任务的时间。单位为毫秒。 | 500 (0.5 秒) |
| logWarningForTaskAfterMillis | LifecycleConfig | 任务尚未完成的情况下在记录警告之前要等待的时长。 | 无 |
| listShardsBackoffTimeInMillis | RetrievalConfig | 发生故障时在调用 ListShards 之间要等待的时间 (以毫秒为单位)。单位为毫秒。 | 1,500 (1.5 秒) |
| maxListShardsRetryAttempts | RetrievalConfig | ListShards 在放弃之前重试的最长时间。 | 50 |
| metricsBufferTimeMillis | MetricsConfig | 指定在发布指标之前缓冲指标的最大持续时间 (以毫秒为单位)。 CloudWatch | 10000 (10 秒) |
| metricsMaxQueueSize | MetricsConfig | 指定发布到之前要缓冲的最大指标数 CloudWatch。 | 10000 |
| metricsLevel | MetricsConfig | 指定要启用和发布的 CloudWatch 指标的粒度级别。 可能的值 : NONE 、 SUMMARY 、 DETAILED 。 | MetricsLevel。 详细 |

| 配置属性 | 配置类 | 说明 | 默认值 |
|--------------------------|---------------|------------------------|------|
| metricsEnabledDimensions | MetricsConfig | 控制 CloudWatch 指标允许的维度。 | 所有维度 |

KCL 3.x 中停用的配置

以下配置属性在 KCL 3.x 中已停用：

下表显示了 KCL 3.x 已停用的配置属性

| 配置属性 | 配置类 | 说明 |
|-------------------------------|-----------------------|--|
| maxLeasesToStealAtOneTime | LeaseManagementConfig | 应用程序一次应尝试窃取的最大租赁数量。KCL 3.x 会忽略此配置，并根据工作程序的资源利用率重新分配租约。 |
| enablePriorityLeaseAssignment | LeaseManagementConfig | 控制工作程序是否应优先考虑已过期的租约（租约未续订的时间达故障转移时间的 3 倍）和新的分片租约，而不管目标租约数如何，但仍需遵守最大租约限制。KCL 3.x 会忽略此配置，并始终将过期租约分布到各个工作程序中。 |

⚠ Important

在从之前的 KCL 版本迁移至 KCL 3.x 时，仍必须拥有停用的配置属性。在迁移过程中，KCL 工作程序首先从 KCL 2.x 兼容模式启动，然后在检测到应用程序的所有 KCL 工作程序做好运行 KCL 3.x 的准备时切换到 KCL 3.x 功能模式。当 KCL 工作程序运行 KCL 2.x 兼容模式时，需要这些停用的配置。

KCL 版本生命周期策略

本主题概述了 Amazon Kinesis 客户端库 (KCL) 的版本生命周期策略。Amazon定期为 KCL 版本提供新版本，以支持新功能和增强功能、错误修复、安全补丁和依赖项更新。我们建议您继续 up-to-date 使用 KCL 版本，以了解最新功能、安全更新和底层依赖关系。我们不建议继续使用不受支持的 KCL 版本。

主要 KCL 版本的生命周期包括以下三个阶段：

- 正式发布 (GA)-在此阶段，将完全支持主要版本。Amazon 提供常规的次要版本和补丁版本，其中包括对 Kinesis Data Streams 的新功能或 API 更新的支持，以及错误和安全修复。
- 维护模式-将补丁版本的发布 Amazon 限制为仅解决关键错误修复和安全问题。主要版本不会收到有关新功能或 Kinesis Data APIs Streams 的更新。
- End-of-support — 主版本将不再接收更新或发布。之前发布的版本将继续通过公共包管理器提供，并且代码将保持不变 GitHub。用户可以自行决定 end-of-support 是否使用已达到的版本。建议您升级到最新的主要版本。

| 主要版本 | 当前阶段 | 发行日期 | 维护模式日期 | End-of-support 日期 |
|---------|------|------------|------------|-------------------|
| KCL 1.x | 维护模式 | 2013-12-19 | 2025-04-17 | 2026-01-30 |
| KCL 2.x | 正式发布 | 2018-08-02 | -- | -- |
| KCL 3.x | 正式发布 | 2024-11-06 | -- | -- |

从之前的 KCL 版本迁移

本主题介绍如何从之前 Kinesis Client Library (KCL) 版本进行迁移。

KCL 3.0 中有何新功能？

与之前的版本相比，Kinesis Client Library (KCL) 3.0 推出了多项主要改进：

- 通过自动将工作从消费端应用程序队列中过度利用的工作程序重新分配给利用不足的工作程序，从而降低消费端应用程序的计算成本。这种新的负载均衡算法确保在各工作程序之间实现均匀的 CPU 利用率分布，并且无需过度配置工作程序。

- 通过优化租约表中的读取操作，降低了与 KCL 相关的 DynamoDB 成本。
- 支持当前工作程序完成对已处理记录的检查点操作，从而最大限度地减少租约重新分配给其他工作程序时对数据的再处理。
- 它 Amazon SDK for Java 2.x 用于改进性能和安全功能，完全消除了对适用于 Java 的 Amazon SDK 1.x 的依赖。

有关更多信息，请参阅 [KCL 3.0 发行说明](#)。

主题

- [从 KCL 2.x 迁移至 KCL 3.x](#)
- [回滚至先前 KCL 版本](#)
- [回滚后前滚到 KCL 3.x](#)
- [使用预置容量模式的租约表的最佳实践](#)
- [从 KCL 1.x 迁移到 KCL 3.x](#)

从 KCL 2.x 迁移至 KCL 3.x

本主题提供了将您的消费者从 KCL 2.x 迁移到 KCL 3.x 的 step-by-step 说明。KCL 3.x 支持 KCL 2.x 消费端进行就地迁移。在滚动迁移工作程序时，可继续使用 Kinesis 数据流中的数据。

Important

KCL 3.x 的接口和方法与 KCL 2.x 保持一致。因此，在迁移期间无需更新记录处理代码。但必须设置正确的配置，并检查迁移所需的步骤。我们强烈建议遵循以下迁移步骤，以获得顺畅的迁移体验。

步骤 1：先决条件

开始使用 KCL 3.x 之前，请确保已具备以下条件：

- Java Development Kit (JDK) 8 或更高版本
- 适用于 Java 的 Amazon SDK 2.x
- 用于依赖项管理的 Maven 或 Gradle

⚠ Important

不要在 KCL 3.x 中使用 2.27.19 到 2.27.23 适用于 Java 的 Amazon SDK 版本。这些版本出现的问题会导致使用 KCL 的 DynamoDB 时出现相关异常错误。我们建议您使用 2.28.0 或更高适用于 Java 的 Amazon SDK 版本来避免此问题。

步骤 2：添加依赖项

如果您使用的是 Maven，请将以下依赖项添加到您的 `pom.xml` 文件中。确保将 `3.x.x` 替换为最新的 KCL 版本。

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

如果使用 Gradle，请在 `build.gradle` 文件中添加以下信息。确保将 `3.x.x` 替换为最新的 KCL 版本。

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

可以在 [Maven Central 存储库](#) 中查看最新版本的 KCL。

步骤 3：设置迁移相关配置

要从 KCL 2.x 迁移至 KCL 3.x，就必须设置以下配置参数：

- `CoordinatorConfig.clientVersionConfig`：此配置决定了应用程序将在哪种 KCL 版本兼容模式下运行。从 KCL 2.x 迁移至 3.x 时，需要将该配置设置为 `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`。要设置此配置，请在创建调度器对象时添加以下行：

```
configsBuilder.coordinatorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X);
```

下面是如何设置 `CoordinatorConfig.clientVersionConfig` 从 KCL 2.x 迁移至 3.x 的示例。您可以按照自己的具体需求，根据需要调整其他配置：

```
Scheduler scheduler = new Scheduler(  
    configsBuilder.checkpointConfig(),  
  
    configsBuilder.coordiantorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONFIG),  
    configsBuilder.leaseManagementConfig(),  
    configsBuilder.lifecycleConfig(),  
    configsBuilder.metricsConfig(),  
    configsBuilder.processorConfig(),  
    configsBuilder.retrievalConfig()  
);
```

重要的是，由于 KCL 2.x 和 3.x 使用的负载均衡算法不同，消费端应用程序中的所有工作程序在给定时间均使用相同的负载均衡算法。如果使用不同的负载均衡算法运行工作程序，可能会导致负载分配不够理想，因为这两种算法是独立运行的。

借助于这种 KCL 2.x 兼容性设置，KCL 3.x 应用程序可以在兼容 KCL 2.x 的模式下运行，并使用 KCL 2.x 的负载均衡算法，直至消费端应用程序中的所有工作程序都升级到 KCL 3.x。迁移完成后，KCL 会自动切换到完整版 KCL 3.x 功能模式，并开始为所有正在运行的工作程序使用新的 KCL 3.x 负载均衡算法。

Important

如果未使用 `ConfigsBuilder`，而是创建 `LeaseManagementConfig` 对象来设置配置，则必须在 KCL 3.x 或更高版本中再添加一个称为 `applicationName` 的参数。有关详细信息，请参阅 [LeaseManagementConfig 构造函数的编译错误](#)。我们建议使用 `ConfigsBuilder` 来设置 KCL 配置。`ConfigsBuilder` 提供了一种更加灵活、更易于维护的方式来配置 KCL 应用程序。

第 4 步：遵循 `shutdownRequested()` 方法实现的最佳实践

KCL 3.x 中推出了一项名为“优雅移交租约”的功能，在租约重新分配过程中将租约移交给其他工作程序时，该功能可最大程度减少对数据的再处理。其实现方法是在租约移交之前，对租约表中最后处理的序列号进行检查点操作。为确保优雅移交租约功能的正常运行，必须保证在 `RecordProcessor` 类的 `shutdownRequested` 方法中调用 `checkpointer` 对象。如果在 `shutdownRequested` 方法中未调用 `checkpointer` 对象，可以按照以下示例所示进行实现。

⚠ Important

- 以下实现示例是优雅移交租约的最低要求。需要时可以进行扩展，以添加与检查点相关的其他逻辑。如果当前正在执行任何异步处理，请确保在调用检查点操作之前已对所有传送到下游的记录进行了处理。
- 虽然优雅移交租约可以显著减小租约转移期间进行数据再处理的可能性，但不能完全消除这种可能性。为保持数据的完整性和一致性，请将下游消费端应用程序设计为具有幂等性。这意味着消费端应用程序应该能够处理潜在的记录重复处理，而不会对整个系统带来不利影响。

```
/**  
 * 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);  
    }  
}
```

步骤 5：检查 KCL 3.x 收集工作程序指标的先决条件

KCL 3.x 收集 CPU 利用率指标（例如工作程序的 CPU 利用率），以期均匀地平衡各工作程序之间的负载。消费者应用程序工作者可以在亚马逊 EC2、亚马逊 ECS、Amazon EKS 或上运行Amazon Fargate。只有在满足以下先决条件时，KCL 3.x 才能从工作程序收集 CPU 利用率指标：

Amazon Elastic Compute Cloud (亚马逊 EC2)

- 操作系统必须是 Linux 操作系统。
- 您必须在您的 EC2 实例[IMDSv2](#)中启用。

亚马逊上的亚马逊弹性容器服务 (Amazon ECS) Container Service EC2

- 操作系统必须是 Linux 操作系统。
- 必须启用 [ECS 任务元数据端点版本 4](#)。
- Amazon ECS 容器代理版本必须为 1.39.0 或更高版本。

Amazon ECS 已开启 Amazon Fargate

- 必须启用 [Fargate 任务元数据端点版本 4](#)。如果使用的是 Fargate 平台版本 1.4.0 或更高版本，则默认启用此功能。
- Fargate (平台版本 1.4.0 或更高版本)。

亚马逊上的亚马逊 Elastic Kubernetes Service (亚马逊 EKS) EC2

- 操作系统必须是 Linux 操作系统。

亚马逊 EKS 开启 Amazon Fargate

- Fargate (平台版本 1.3.0 或更高版本)。

Important

如果 KCL 3.x 因为未满足先决条件而无法从工作程序收集 CPU 利用率指标，它将根据每个租约的吞吐量级别重新平衡负载。这种后备再平衡机制可确保所有工作程序都能从分配给各工作

程序的租约中获得类似级别的总吞吐量。有关更多信息，请参阅 [KCL 如何向工作程序分配租约并平衡负载](#)。

第 6 步：更新 KCL 3.x 的 IAM 权限

必须向与 KCL 3.x 消费端应用程序有关的 IAM 角色或策略添加以下权限。这涉及对 KCL 应用程序使用的现有 IAM 策略进行更新。有关更多信息，请参阅 [KCL 消费端应用程序所必需的 IAM 权限](#)。

Important

您的现有 KCL 应用程序可能没有在 IAM 策略中添加以下 IAM 操作与资源，因为 KCL 2.x 不需要这些操作与资源。在运行 KCL 3.x 应用程序之前，请确保已添加这些操作与资源：

- 动作：UpdateTable
 - 资源 (ARNs): arn:aws:dynamodb:region:account:table/
KCLApplicationName
- 动作：Query
 - 资源 (ARNs): arn:aws:dynamodb:region:account:table/
KCLApplicationName/index/*
- 操作
 - 作：CreateTable、DescribeTable、Scan、GetItem、PutItem、UpdateItem、DeleteItem
 - 资源 (ARNs): arn:aws:dynamodb:region:account:table/
KCLApplicationName-WorkerMetricStats,
arn:aws:dynamodb:region:account:table/KCLApplicationName-
CoordinatorState

将中的“区域”、“账户”和“KCLApplication名称”分别替换为您自己的名称Amazon Web Services 区域、Amazon Web Services 账户号码和 KCL 应用程序名称。ARNs 如果使用配置来自定义 KCL 创建的元数据表的名称，请使用这些指定的表名称而不是 KCL 应用程序名称。

第 7 步：将 KCL 3.x 代码部署到工作程序

在设置了迁移所需的配置并完成之前所有的迁移清单之后，就可以构建代码并将其部署到工作程序中。

Note

如果您看到构造函数出现编译错误，请参阅[LeaseManagementConfig构造函数的编译错误以](#)获取疑难解答信息。[LeaseManagementConfig](#)

步骤 8：完成迁移

在部署 KCL 3.x 代码期间，KCL 将继续使用来自 KCL 2.x 的租约分配算法。成功将 KCL 3.x 代码部署到所有工作程序后，KCL 会自动检测部署情况，并根据工作程序的资源利用率切换到新的租约分配算法。有关新租约分配算法的更多详细信息，请参阅[KCL 如何向工作程序分配租约并平衡负载](#)。

在部署期间，您可以使用向其发送以下指标来 CloudWatch 监控迁移过程。您可以监控 `Migration` 操作下的指标。所有指标均为 `per-KCL-application` 指标，设置为 `SUMMARY` 指标级别。如果 `CurrentState:3xWorker` 指标的 `Sum` 统计数据与 KCL 应用程序的工作程序总数一致，则表示已成功完成向 KCL 3.x 的迁移。

Important

在所有工作程序做好运行新租约分配算法的准备之后，KCL 至少需要 10 分钟才能切换到新算法。

CloudWatch KCL 迁移过程的指标

| 指标 | 说明 |
|--|--|
| <code>CurrentState:3xWorker</code> | 成功迁移至 KCL 3.x 并运行新租约分配算法的 KCL 工作程序数量。如果此指标的 <code>Sum</code> 计数与工作程序总数一致，则表示已成功完成向 KCL 3.x 的迁移。 <ul style="list-style-type: none">指标级别：汇总单位：计数统计数据：最有用的统计工具是 <code>Sum</code>。 |
| <code>CurrentState:2xCompatibleWorker</code> | 迁移过程中在 KCL 2.x 兼容模式下运行的 KCL 工作程序数量。该指标若为非零值，表示迁移仍在进行中。 |

| 指标 | 说明 |
|--------------------|---|
| | <ul style="list-style-type: none">• 指标级别：汇总• 单位：计数• 统计数据：最有用的统计工具是 Sum。 |
| Fault | <p>迁移过程中遇到的异常数量。这些异常大多数是瞬时错误，KCL 3.x 会自动重试以完成迁移。如果发现永久性的 Fault 指标值，请查看迁移期间的日志，以进一步排除故障。如果问题仍然存在，请联系 Amazon Web Services 支持。</p> <ul style="list-style-type: none">• 指标级别：汇总• 单位：计数• 统计数据：最有用的统计工具是 Sum。 |
| GsiStatusReady | <p>租约表上创建全局二级索引 (GSI) 的状态。该指标表示租约表上的 GSI 是否已创建，这是运行 KCL 3.x 的一个先决条件。其值为 0 或 1，其中 1 表示创建成功。在回滚状态下，不会发出该指标。再次向前滚动后，可以继续监控该指标。</p> <ul style="list-style-type: none">• 指标级别：汇总• 单位：计数• 统计数据：最有用的统计工具是 Sum。 |
| workerMetricsReady | <p>所有工作程序指标的发送状态。该指标表示是否所有工作程序都发出诸如 CPU 利用率之类的指标。其值为 0 或 1，其中 1 表示所有工作程序均已成功发出指标，并准备好使用新的租约分配算法。在回滚状态下，不会发出该指标。再次向前滚动后，可以继续监控该指标。</p> <ul style="list-style-type: none">• 指标级别：汇总• 单位：计数• 统计数据：最有用的统计工具是 Sum。 |

KCL 在迁移期间提供回滚至 2.x 兼容模式的能力。成功迁移至 KCL 3.x 后，如果不再需要回滚，我们建议移除 `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X` 的 `CoordinatorConfig.clientVersionConfig` 设置。移除该配置之后，就不会从 KCL 应用程序发出与迁移相关的指标。

Note

我们建议在迁移期间和完成迁移后，监控应用程序的性能和稳定性一段时间。如果发现任何问题，可使用 [KCL 迁移工具](#)回滚工作程序以使用 KCL 2.x 兼容功能。

回滚至先前 KCL 版本

本主题介绍将消费端回滚到先前版本的步骤。需要回滚时，可执行一个两步流程：

1. 运行 [KCL Migration Tool](#)。
2. 重新部署以前的 KCL 版本代码（可选）。

步骤 1：运行 KCL 迁移工具

当需要回滚到先前 KCL 版本时，必须运行 KCL 迁移工具。KCL 迁移工具可完成两项重要任务：

- 它在 DynamoDB 中的租约表上移除一个名为工作线程指标表的元数据表和全局二级索引。这两个构件由 KCL 3.x 创建，但在回滚到先前版本时并不需要。
- 它使所有工作程序在与 KCL 2.x 兼容的模式下运行，并开始使用先前 KCL 版本中使用的负载平衡算法。如果 KCL 3.x 中的新负载均衡算法存在问题，这将立即缓解问题。

Important

DynamoDB 中的协调器状态表必须存在，并且在迁移、回滚和前滚过程中不得删除。

Note

重要的是，使用者应用程序中的所有工作线程在给定时间均使用相同的负载均衡算法。KCL 迁移工具可确保 KCL 3.x 使用者应用程序中的所有工作线程都切换到 KCL 2.x 兼容模式，以便在部署回滚到先前 KCL 版本期间，所有工作线程都运行相同的负载均衡算法。

您可以在 [KCL 存储库的脚本目录中下载 KCL GitHub 迁移工具](#)。可以从任何工作程序或具备以下必需权限的主机上运该脚本：写入协调器状态表、删除工作程序指标表以及更新租约表的权限。可参考 [KCL 消费端应用程序所必需的 IAM 权限](#) 获取运行脚本所需的 IAM 权限。每个 KCL 应用程序只能运行该脚本一次。您可使用以下命令运行 KCL 迁移工具：

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

参数

- **--区域**：替换<region>为你的 Amazon Web Services 区域
- **--application_name**：如果您为 DynamoDB 元数据表（租约表、协调器状态表和工作线程指标表）使用默认名称，则需要此参数。如果您为这些表指定了自定义名称，则可以忽略此参数。将 <applicationName> 替换为实际的 KCL 应用程序名称。如果未提供自定义名称，该工具将使用此名称来派生默认表名称。
- **--lease_table_name**（可选）：如果您在 KCL 配置中为租约表设置了自定义名称，则需要此参数。如果您使用的是默认表名称，则可以忽略此参数。将 leaseTableName 替换为您为租约表指定的自定义表名称。
- **--coordinator_state_table_name**（可选）：如果您在 KCL 配置中为协调器状态表设置了自定义名称，则需要此参数。如果您使用的是默认表名称，则可以忽略此参数。将 <coordinatorStateTableName> 替换为您为协调器状态表指定的自定义表名称。
- **--worker_metrics_table_name**（可选）：如果您在 KCL 配置中为工作线程指标表设置了自定义名称，则需要此参数。如果您使用的是默认表名称，则可以忽略此参数。将 <workerMetricsTableName> 替换为您为工作线程指标表指定的自定义表名称。

步骤 2：使用先前 KCL 版本重新部署代码（可选）

运行 KCL 迁移工具来进行回滚后，您将看到以下消息之一：

- 消息 1：“回滚已完成。您的 KCL 应用程序正在运行 KCL 2.x 兼容模式。如果您看不到回归缓解，请使用先前 KCL 版本部署代码，回滚到先前的应用程序二进制文件。”
- 必需的操作：这意味着您的工作人员正在 KCL 2.x 兼容模式下运行。如果仍有问题，请使用先前 KCL 版本将代码重新部署到工作程序。

- 消息 2：“回滚已完成。您的 KCL 应用程序正在运行 KCL 3.x 功能模式。您无需回滚到以前的应用程序二进制文件，除非在 5 分钟内看不到任何缓解该问题的措施。如果仍有问题，请使用先前 KCL 版本部署代码，以回滚到先前的应用程序二进制文件。”
- 必需的操作：这意味着您的工作人员在 KCL 3.x 模式下运行，KCL 迁移工具已将所有工作人员切换到兼容 KCL 2.x 的模式。如果问题得到解决，则无需使用之前的 KCL 版本重新部署代码。如果仍有问题，请使用先前 KCL 版本将代码重新部署到工作程序。

回滚后前滚到 KCL 3.x

本主题介绍在回滚后将消费端前滚到 KCL 3.x 的步骤。当您需要前滚时，必须完成一个由两步组成的过程：

1. 运行 [KCL Migration Tool](#)。
2. 使用 KCL 3.x 部署代码。

步骤 1：运行 KCL 迁移工具

运行 KCL Migration Tool。具有以下命令的 KCL 迁移工具，用于前滚到 KCL 3.x：

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

参数

- --区域：替换<region>为你的 Amazon Web Services 区域
- --application_name：如果您为协调器状态表使用默认名称，则需要此参数。如果您已为协调器状态表指定了自定义名称，则可以忽略此参数。将 <applicationName> 替换为实际的 KCL 应用程序名称。如果未提供自定义名称，该工具将使用此名称来派生默认表名称。
- --coordinator_state_table_name (可选)：如果您在 KCL 配置中为协调器状态表设置了自定义名称，则需要此参数。如果您使用的是默认表名称，则可以忽略此参数。将 <coordinatorStateTableName> 替换为您为协调器状态表指定的自定义表名称。

在前滚模式下运行迁移工具后，KCL 会创建 KCL 3.x 所需的以下 DynamoDB 资源：

- 租约表上的全局二级索引
- 工作线程指标表

步骤 2：使用 KCL 3.x 部署代码

运行 KCL 迁移工具以进行前滚后，使用 KCL 3.x 将代码部署到工作线程。按照 [步骤 8：完成迁移](#) 完成迁移。

使用预置容量模式的租约表的最佳实践

如果 KCL 应用程序的租约表已切换至预置容量模式，KCL 3.x 会在采用预置计费模式的租约表上创建全局二级索引，租约表的读取容量单位 (RCU) 和写入容量单位 (WCU) 与基础租约表一致。创建全局二级索引时，我们建议在 DynamoDB 控制台中监控全局二级索引的实际使用情况，并根据需要调整容量单位。有关切换由 KCL 创建的 DynamoDB 元数据表容量模式的更详细指南，请参阅[KCL 创建的元数据表的 DynamoDB 容量模式](#)。

Note

默认情况下，KCL 使用按需容量模式在租约表上创建元数据表，例如租约表、工作程序指标表和协调器状态表以及全局二级索引。我们建议您使用按需容量模式，以便根据您的使用量变化自动调整容量。

从 KCL 1.x 迁移到 KCL 3.x

本主题说明如何将消费端从 KCL 1.x 迁移至 KCL 3.x。与 KCL 2.x 和 KCL 3.x 相比，KCL 1.x 使用不同的类和接口。必须先将记录处理器、记录处理器工厂和工作线程类迁移到 KCL 2.x/3.x 兼容格式，然后按照将 KCL 2.x 迁移到 KCL 3.x 的迁移步骤进行操作。可直接从 KCL 1.x 升级至 KCL 3.x。

- 步骤 1：迁移记录处理器

按照[将消费端从 KCL 1.x 迁移至 KCL 2.x](#) 页面中的[迁移记录处理器](#)部分进行操作。

- 步骤 2：迁移记录处理器工厂

按照[将消费端从 KCL 1.x 迁移至 KCL 2.x](#) 页面中的[迁移记录处理器工厂](#)部分进行操作。

- 步骤 3：迁移工作人员

按照[将消费端从 KCL 1.x 迁移至 KCL 2.x](#) 页面中的[迁移工作程序](#)部分进行操作。

- 第 4 步：迁移 KCL 1.x 配置

按照[将消费端从 KCL 1.x 迁移至 KCL 2.x](#) 页面中的[配置 Amazon Kinesis 客户端](#)部分进行操作。

- 第 5 步：检查闲置时间删除和客户端配置移除情况

按照[将消费端从 KCL 1.x 迁移至 KCL 2.x](#) 页面中的[闲置时间删除](#)和[客户端配置移除](#)部分进行操作。

- 第 6 步：按照 KCL 2.x 到 KCL 3.x 迁移 step-by-step 指南中的说明进行操作

遵循[从 KCL 2.x 迁移至 KCL 3.x](#) 页面上的说明完成迁移。如需回滚到之前 KCL 版本，或在回滚后向前滚到 KCL 3.x，请参阅[回滚至先前 KCL 版本](#)和[回滚后前滚到 KCL 3.x](#)。

Important

不要在 KCL 3.x 中使用 2.27.19 到 2.27.23 适用于 Java 的 Amazon SDK 版本。这些版本出现的问题会导致使用 KCL 的 DynamoDB 时出现相关异常错误。我们建议您使用 2.28.0 或更高适用于 Java 的 Amazon SDK 版本来避免此问题。

先前的 KCL 版本文档

以下主题已归档。要查看当前的 Kinesis Client Library 文档，请参阅[使用 Kinesis Client Library](#)。

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问[Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅[使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅[从 KCL 1.x 迁移到 KCL 3.x](#)。

停用的文档

- [KCL 1.x 和 2.x 信息](#)
- [开发具有共享吞吐量的自定义消费端](#)
- [将消费端从 KCL 1.x 迁移到 KCL 2.x](#)

KCL 1.x 和 2.x 信息

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

开发可以处理来自 KDS 数据流的数据的自定义消费端应用程序的一种方法，是使用 Kinesis Client Library (KCL) 。

主题

- [关于 KCL \(先前版本 \)](#)
- [KCL 先前版本](#)
- [KCL 概念 \(先前版本 \)](#)
- [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)
- [使用相同的适用于 Java 的 KCL 2.x 消费端应用程序处理多个数据流](#)
- [将 KCL 与 Amazon Glue 架构注册表一起使用](#)

Note

建议您根据使用场景将 KCL 1.x 和 KCL 2.x 升级到最新的 KCL 1.x 版本或 KCL 2.x 版本。KCL 1.x 和 KCL 2.x 会定期更新至最新版本，包括最新依赖项和安全补丁、错误修复以及向后兼容的新功能。有关更多信息，请参阅 <https://github.com/awslabs/amazon-kinesis-client/releases>。

关于 KCL (先前版本)

KCL 通过处理许多与分布式计算相关的复杂任务，帮助您使用和处理 Kinesis 数据流中的数据。这些任务包括跨多个消费端应用程序实例的负载平衡、对消费端应用程序实例故障的响应、已处理记录的检查点操作以及对重新分片的反应。KCL 负责所有这些子任务，让您可以将精力集中在编写自定义记录处理逻辑上。

KCL 不同于中提供的 Kinesis Data APIs Streams。Amazon SDKs Kinesis Data APIs Streams 可帮助您管理 Kinesis Data Streams 的许多方面，包括创建流、重新分片以及放置和获取记录。KCL 围绕这些子任务提供了一个抽象层，让您可以专注于消费端应用程序的自定义数据处理逻辑工作。有关 Kinesis Data Streams API 的信息，请参阅 [Amazon Kinesis API Reference](#)。

Important

KCL 属于 Java 库，使用名为 MultiLangDaemon 的多语言接口提供对 Java 以外语言的支持。此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后台运行。例如，如果您安装适用于 Python 的 KCL 并完全使用 Python 编写使用者应用程序，则仍然需要在系统上安装 Java，因为。MultiLangDaemon 此外，MultiLangDaemon 还有一些您可能需要根据自己的用例自定义的默认设置，例如它所连接的 Amazon 区域。有关 MultiLangDaemon 的更多信息 GitHub，请参阅 [KCL MultiLangDaemon 项目](#)。

KCL 充当记录处理逻辑和 Kinesis Data Streams 之间的中介。

KCL 先前版本

目前，您可以使用以下任一支持的 KCL 版本来构建自定义消费端应用程序：

- KCL 1.x

有关更多信息，请参阅 [开发 KCL 1.x 消费端](#)。

- KCL 2.x

有关更多信息，请参阅 [开发 KCL 2.x 消费端](#)。

您可以使用 KCL 1.x 或 KCL 2.x 来构建使用共享吞吐量的消费端应用程序。有关更多信息，请参阅 [使用 KCL 开发具有共享吞吐量的自定义消费端](#)。

要构建使用专用吞吐量的消费端应用程序（增强型扇出消费端应用程序），只能使用 KCL 2.x。有关更多信息，请参阅 [开发具有专用吞吐量的增强扇出型消费端](#)。

有关 KCL 1.x 和 KCL 2.x 之间差异的信息，以及如何从 KCL 1.x 迁移到 KCL 2.x 的说明，请参阅 [将消费端从 KCL 1.x 迁移到 KCL 2.x](#)。

KCL 概念（先前版本）

- KCL 消费端应用程序 – 使用 KCL 自定义构建的应用程序，旨在读取和处理数据流中的记录。

- **消费端应用程序实例** – KCL 消费端应用程序通常是分布式应用程序，即一个或多个应用程序实例同时运行，以便协调故障和以动态方式实现数据记录处理负载平衡。
- **工作程序** – KCL 消费端应用程序实例用来开始处理数据的高级类。

 **Important**

每个 KCL 消费端应用程序实例都有一个工作程序。

工作程序负责初始化和监督各种任务，包括同步分片和租约信息、跟踪分片分配以及处理来自分片的数据。工作程序向 KCL 提供使用者应用程序的配置信息，例如此 KCL 使用者应用程序要处理的数据记录的数据流的名称以及访问此数据流所需的 Amazon 凭据。工作程序还会启动特定的 KCL 消费端应用程序实例，将数据记录从数据流传输到记录处理器。

 **Important**

在 KCL 1.x 中，这个类被称为工作程序。有关更多信息（这些是 Java KCL 存储库），请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/Worker.java>。在 KCL 2.x 中，这个类被称为计划程序。KCL 2.x 中计划程序的用途与 KCL 1.x 中工作程序的用途相同。有关 KCL 2.x 中调度器类的更多信息，请参见 <https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/coordinator/Scheduler>

- **租约** – 定义工作程序和分片之间绑定的数据。分布式 KCL 消费端应用程序使用租约在一组工作程序中对数据记录进行分区。在任何指定时间，每个数据记录分片都通过由 leaseKey 变量标识的租约绑定到特定的工作程序。

默认情况下，工作人员可以同时持有一份或多份租约（以 maxLeasesForWorker 变量的值为准）。

 **Important**

每个工作程序都将争相持有数据流中所有可用分片的所有可用租约。但是，无论何时，只有一个工作程序可以成功持有每份租约。

例如，如果您的消费端应用程序实例 A 和工作程序 A 正在处理包含 4 个分片的数据流，则工作程序 A 可以同时持有分片 1、2、3 和 4 的租约。但是，如果您有 A 和 B 两个消费端应用程序实例，以及

工作程序 A 和工作程序 B，并且这些实例正在处理包含 4 个分片的数据流，则工作程序 A 和工作程序 B 不能同时持有分片 1 的租约。一个工作程序持有特定分片的租约，直到该工作程序准备好停止处理该分片的数据记录或该工作程序失败为止。当一个工作程序停止持有租约时，另一个工作程序会接管并持有租约。

有关更多信息（这些是 Java KCL 存储库），请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.java> for KCL 1.x 和 <https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/Lease.java> for KCL 2.x。

- 租约表 – 唯一的 Amazon DynamoDB 表，用于跟踪 KDS 数据流中由 KCL 消费端应用程序的工作程序租赁和处理的分片。在 KCL 消费端应用程序运行时，租约表必须与数据流中的最新分片信息保持同步（在工作程序内部和所有工作程序之间）。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。
- 记录处理器 – 定义 KCL 消费端应用程序如何处理从数据流中获取的数据的逻辑。在运行时，KCL 消费端应用程序实例会实例化一个工作程序，该工作程序会为持有租约的每个分片实例化一个记录处理器。

使用租约表跟踪 KCL 消费端应用程序处理的分片

主题

- [什么是租约表](#)
- [吞吐量](#)
- [租约表如何与 Kinesis 数据流中的分片同步](#)

什么是租约表

对于每个 Amazon Kinesis Data Streams 应用程序，KCL 都使用一份唯一的租约表（存储在 Amazon DynamoDB 表中）来跟踪 KDS 数据流中由 KCL 消费端应用程序的工作程序租赁和处理的分片。

Important

KCL 使用消费端应用程序的名称来创建该消费端应用程序使用的租约表的名称，因此每个消费端应用程序的名称必须是唯一的。

您可在消费端应用程序运行的同时使用 [Amazon DynamoDB 控制台](#) 查看租约表。

如果应用程序启动时 KCL 消费端应用程序的租约表不存在，则其中一个工作程序会为此应用程序创建租约表。

Important

除开与 Kinesis Data Streams 本身关联的费用，您的账户将被收取与 DynamoDB 表关联的费用。

租约表中的每行表示您消费端应用程序正在处理的分片。如果 KCL 消费端应用程序仅处理一个数据流，则租约表的哈希键 leaseKey 就是分片 ID。如果您[使用相同的适用于 Java 的 KCL 2.x 消费端应用程序处理多个数据流](#)，则 leaseKey 的结构如下：account-id:StreamName:streamCreationTimestamp:ShardId。例如 111111111:multiStreamTest-1:12345:shardId-000000000336。

除了分片 ID 以外，每行还包含以下数据：

- checkpoint：分片的最新检查点序号。此值在数据流的所有分片中都是唯一的。
- checkpointSubSequence数字：使用 Kinesis Producer 库的聚合功能时，这是对检查点的扩展，用于跟踪 Kinesis 记录中的单个用户记录。
- leaseCounter：用于租赁版本控制，这样工作程序可以检测其租赁已由其他工作程序获取。
- leaseKey：租赁的唯一标识符。每份租约都是数据流中一个分片所特有的，一份由一个工作程序持有。
- leaseOwner：持有此租赁的工作程序。
- ownerSwitchesSince检查点：自上次写检查点以来，这份租约更换了多少次员工。
- parentShardId：用于确保在子分片上开始处理之前，父分片已得到完全处理。这可以确保记录按照放入流中的相同顺序处理。
- hashrange：PeriodicShardSyncManager 用来运行定期同步，以查找租约表中缺少的分片，并在需要时为分片创建租约。

Note

从 KCL 1.14 和 KCL 2.3 开始的每个分片的租约表中都有此数据。有关租约和分片之间的 PeriodicShardSyncManager 和定期同步的更多信息，请参阅[租约表如何与 Kinesis 数据流中的分片同步](#)。

- childshards : LeaseCleanupManager 用来查看子分片的处理状态并决定是否可以从租约表中删除父分片。

 Note

从 KCL 1.14 和 KCL 2.3 开始的每个分片的租约表中都有此数据。

- shardID : 分片的 ID。

 Note

仅当您使用相同的适用于 Java 的 KCL 2.x 消费端应用程序处理多个数据流时，此数据才会出现在租约表中。只有适用于 Java 的 KCL 2.x 才支持此功能，且从适用于 Java 的 KCL 2.3 及更高版本开始才支持此功能。

- stream name 数据流的标识符采用以下格式 : account-id:StreamName:streamCreationTimestamp。

 Note

仅当您使用相同的适用于 Java 的 KCL 2.x 消费端应用程序处理多个数据流时，此数据才会出现在租约表中。只有适用于 Java 的 KCL 2.x 才支持此功能，且从适用于 Java 的 KCL 2.3 及更高版本开始才支持此功能。

吞吐量

如果您的 Amazon Kinesis Data Streams 收到了预置的吞吐量异常，您应为 DynamoDB 表增加预置的吞吐量。KCL 将创建预置吞吐量为 10 次读取/秒和 10 次写入/秒的表，但这可能无法满足您应用程序的需求。例如，如果您的 Amazon Kinesis Data Streams 执行频繁的检查点操作或对由很多分片组成的流执行操作，您可能需要更多吞吐量。

有关 DynamoDB 表中预置吞吐量的信息，请参阅《Amazon DynamoDB 开发人员指南》中的读/写容量模式和使用表和数据。

租约表如何与 Kinesis 数据流中的分片同步

KCL 消费端应用程序中的工作程序使用租约来处理指定数据流中的分片。哪个工作程序在指定时间租赁哪个分片的相关信息都存储在租约表中。在 KCL 消费端应用程序运行期间，租约表必须与数据流中的最新分片信息保持同步。在消费端应用程序引导启动期间（初始化或重新启动消费端应用程序时），

以及每当正在处理的分片结束时（重新分片），KCL 都会将租约表与从 Kinesis Data Streams 服务获取的分片信息同步。换言之，在消费端应用程序初始引导启动期间，以及每当消费端应用程序遇到数据流重新分片事件时，工作程序或 KCL 消费端应用程序都会与其正在处理的数据流同步。

主题

- [KCL 1.0 - 1.13 和 KCL 2.0 - 2.2 中的同步](#)
- [KCL 2.x 中的同步从 KCL 2.3 及更高版本开始](#)
- [KCL 1.x 中的同步从 KCL 1.14 及更高版本开始](#)

KCL 1.0 - 1.13 和 KCL 2.0 - 2.2 中的同步

在 KCL 1.0-1.13 和 KCL 2.0-2.2 中，在使用者应用程序的引导期间以及每个数据流重新分片事件期间，KCL 通过调用或发现将租用表与从 Kinesis Data Streams 服务获取的分片信息同步。ListShards DescribeStream APIs 在上面列出的所有 KCL 版本中，KCL 使用者应用程序的每个 worker 都要完成以下步骤，以便在使用者应用程序的引导期间和每个流 reshard 事件中执行 lease/shard 同步过程：

- 获取正在处理的流中数据的所有分片
- 从租约表中获取所有分片租约
- 筛选出租约表中没有租约的所有开放分片
- 迭代所有找到的开放分片以及没有开放父分片的所有开放分片：
 - 遍历层次结构树的原级路径，确定该分片是否为后代分片。如果正在处理原级分片（租约表中存在原级分片的租约条目）或者应该处理原级分片（例如初始位置为 TRIM_HORIZON 或 AT_TIMESTAMP），则该分片被视为后代分片
 - 如果上下文中的开放分片是后代分片，KCL 会根据初始位置对分片执行检查点操作，并在需要时为其父分片创建租约

KCL 2.x 中的同步从 KCL 2.3 及更高版本开始

从支持的最新版本 KCL 2.x (KCL 2.3) 及更高版本开始，该库现在支持对同步过程进行以下更改。这些 lease/shard 同步更改显著减少了 KCL 使用者应用程序对 Kinesis Data Streams 服务进行的 API 调用次数，并优化了 KCL 使用者应用程序中的租赁管理。

- 在应用程序的引导启动过程中，如果租约表为空，KCL 将利用 ListShard API 的筛选选项（ShardFilter 可选请求参数），仅针对在 ShardFilter 参数指定时间开放的分片的快照检索和创建租约。ShardFilter 参数可以让您筛选出 ListShards API 的响应。ShardFilter 参数

唯一需要的属性是 Type。KCL 使用 Type 筛选属性及其以下有效值来识别并返回可能需要新租约的开放分片的快照：

- AT_TRIM_HORIZON – 响应包含在 TRIM_HORIZON 时开放的所有分片。
- AT_LATEST – 响应仅包含数据流中当前开放的分片。
- AT_TIMESTAMP – 响应包含起始时间戳小于或等于指定时间戳且结束时间戳大于或等于指定时间戳的所有分片，或仍处于开放状态的所有分片。

`ShardFilter` 用于为空租约表创建租约，以初始化在 `RetrievalConfig#initialPositionInStreamExtended` 中指定的分片快照的租约。

有关 `ShardFilter` 的更多信息，请参阅https://docs.amazonaws.cn/kinesis/latest/APIReference/API_ShardFilter.html。

- 不是所有工作人员都执行 lease/shard 同步以使租赁表与数据流中的最新分片保持同步，而是由一个当选的工作领导者执行租赁/分片同步。
- KCL 2.3 使用 `GetRecords` 和 `ChildShards` 返回参数对已关闭的分片执行 lease/shard 同步，从而允许 KCL 工作程序仅为其完成处理的分片的子分片创建租约。 `SubscribeToShard` APIs `SHARD-END` 为了在整个消费者应用程序中共享，lease/shard 同步的这种优化使用了 `GetRecords` API 的 `ChildShards` 参数。对于专用吞吐量（增强扇出）消费者应用程序，lease/shard 同步的这种优化使用 API 的 `ChildShardsSubscribeToShard` 参数。有关更多信息，请参阅 [GetRecords](#)、[SubscribeToShards](#) 和 [ChildShard](#)。
- 经过上述更改，KCL 的行为正在从所有工作程序学习所有现有分片的模式，转变为工作程序只学习每个工作程序拥有的分片的子分片的模式。因此，除了在使用者应用程序引导和重新分片事件期间发生的同步外，KCL 现在还会执行额外的定期 shard/lease 扫描，以识别租赁表中的任何潜在漏洞（换句话说，了解所有新分片），以确保处理数据流的完整哈希范围，并在需要时为它们创建租约。`PeriodicShardSyncManager` 是负责运行定期 lease/shard 扫描的组件。

有关 KCL 2.3 `PeriodicShardSyncManager` 中的更多信息，请参阅 [https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/LeaseManagementConfig.java #L201-L213](https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/LeaseManagementConfig.java#L201-L213)。

在 KCL 2.3 中，可以使用新的配置选项来配置 `LeaseManagementConfig` 中的 `PeriodicShardSyncManager`：

| Name | 默认值 | 说明 |
|---|-----------------|---|
| leasesRec everyAudit orExecutorExe cutionFrequen cyMillis | 120000 (2 分钟) | 审计程序作业扫描租约表中部分租约的频率 (以毫秒为单位) 。如果审计程序检测到某个流的租约中存在任何漏洞，则会根据 leasesRec everyAudit orIncons istencyCo nfidenceT hreshold 触发分片同步。 |
| leasesRec everyAudit orIncons istencyCo nfidenceT hreshold | 3 | 定期审计程序作业的置信度阈值，用于确定租约表中数据流的租约是否不一致。如果审计程序连续多次发现同一数据流的不一致之处，则会触发分片同步。 |

现在还会发布新的 CloudWatch 指标来监控的 PeriodicShardSyncManager 运行状况。有关更多信息，请参阅 [PeriodicShardSyncManager](#)。

- 包括对 HierarchicalShardSyncer 的优化，可仅为一层分片创建租约。

KCL 1.x 中的同步从 KCL 1.14 及更高版本开始

从支持的最新版本 KCL 1.x (KCL 1.14) 及更高版本开始，该库现在支持对同步过程进行以下更改。这些 lease/shard 同步更改显著减少了 KCL 使用者应用程序对 Kinesis Data Streams 服务进行的 API 调用次数，并优化了 KCL 使用者应用程序中的租赁管理。

- 在应用程序的引导启动过程中，如果租约表为空，KCL 将利用 ListShard API 的筛选选项 (ShardFilter 可选请求参数)，仅针对在 ShardFilter 参数指定时间开放的分片的快照检索和创建租约。ShardFilter 参数可以让您筛选出 ListShards API 的响应。ShardFilter 参数唯一需要的属性是 Type。KCL 使用 Type 筛选属性及其以下有效值来识别并返回可能需要新租约的开放分片的快照：
 - AT_TRIM_HORIZON – 响应包含在 TRIM_HORIZON 时开放的所有分片。
 - AT_LATEST – 响应仅包含数据流中当前开放的分片。
 - AT_TIMESTAMP – 响应包含起始时间戳小于或等于指定时间戳且结束时间戳大于或等于指定时间戳的所有分片，或仍处于开放状态的所有分片。

ShardFilter 用于为空租约表创建租约，以初始化在

KinesisClientLibConfiguration#initialPositionInStreamExtended 中指定的分片快照的租约。

有关 ShardFilter 的更多信息，请参阅https://docs.amazonaws.cn/kinesis/latest/APIReference/API_ShardFilter.html。

- 不是所有工作人员都执行 lease/shard 同步以使租赁表与数据流中的最新分片保持同步，而是由一个当选的工作领导者执行租赁/分片同步。
- KCL 1.14 使用 GetRecords 和的 ChildShards 返回参数对已关闭的 SubscribeToShard APIs 分片执行 lease/shard 同步，从而允许 KCL 工作程序仅为完成处理的分片的子分片创建租约。SHARD_END 有关更多信息，请参阅 [GetRecords](#) 和 [ChildShard](#)。
- 经过上述更改，KCL 的行为正在从所有工作程序学习所有现有分片的模式，转变为工作程序只学习每个工作程序拥有的分片的子分片的模式。因此，除了在使用者应用程序引导和重新分片事件期间发生的同步外，KCL 现在还会执行额外的定期 shard/lease 扫描，以识别租赁表中的任何潜在漏洞（换句话说，了解所有新分片），以确保处理数据流的完整哈希范围，并在需要时为它们创建租约。PeriodicShardSyncManager 是负责运行定期 lease/shard 扫描的组件。

如果 KinesisClientLibConfiguration#shardSyncStrategyType 设置为 ShardSyncStrategyType.SHARD_END，则 PeriodicShardSync leasesRecoveryAuditorInconsistencyConfidenceThreshold 用于确定租约表中包含漏洞的连续扫描次数的阈值，之后将强制执行分片同步。当

KinesisClientLibConfiguration#shardSyncStrategyType 设置为时
ShardSyncStrategyType.PERIODIC , leasesRecoveryAuditorInconsistencyConfidenceThreshold 将被忽略。

有关 KCL 1.14 PeriodicShardSyncManager 中的更多信息，请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/KinesisClientLibConfiguration.java> #L987-L999。

在 KCL 1.14 中，可以使用新的配置选项来配置 LeaseManagementConfig 中的 PeriodicShardSyncManager：

| Name | 默认值 | 说明 |
|---|-----|--|
| leasesRecoveryAuditInconsistencyConfidenceThreshold | 3 | 定期审计程序作业的置信度阈值，用于确定租约表中数据流的租约是否不一致。如果审计程序连续多次发现同一数据流的不一致之处，则会触发分片同步。 |

现在还会发布新的 CloudWatch 指标来监控的 PeriodicShardSyncManager 运行状况。有关更多信息，请参阅 [PeriodicShardSyncManager](#)。

- KCL 1.14 现在还支持延期租约清理。当分片超过数据流的保留期或因重新分片操作而关闭时，LeaseCleanupManager 会在到达 SHARD-END 时异步删除租约。

可以使用新的配置选项来配置 LeaseCleanupManager。

| Name | 默认值 | 说明 |
|----------------------|----------|----------------|
| leaseCleanupInterval | 1 minute | 运行租约清理线程的时间间隔。 |

| Name | 默认值 | 说明 |
|-------------------------------------|------|-------------------------------|
| completedLeaseCleanUpIntervalMillis | 5分钟 | 检查租约是否完成的时间间隔。 |
| garbageLeaseCleanUpIntervalMillis | 30分钟 | 检查租约是否为垃圾租约（即超过数据流的保留期）的时间间隔。 |

- 包括对 `KinesisShardSyncer` 的优化，可仅为一层分片创建租约。

使用相同的适用于 Java 的 KCL 2.x 消费端应用程序处理多个数据流

本节介绍了适用于 Java 的 KCL 2.x 中的以下更改，这些更改让您能够创建可同时处理多个数据流的 KCL 消费端应用程序。

⚠ Important

只有适用于 Java 的 KCL 2.x 才支持多流处理功能，且从适用于 Java 的 KCL 2.3 及更高版本开始才支持此功能。

其他可以实现 KCL 2.x 的语言都不支持多流处理功能。

任何版本的 KCL 1.x 都不支持多流处理功能。

- `MultistreamTracker` 接口

要构建可以同时处理多个流的使用者应用程序，必须实现一个名为的新接口 [MultistreamTracker](#)。此接口包括返回要由 KCL 消费端应用程序处理的数据流及其配置列表的 `streamConfigList` 方法。请注意，正在处理的数据流可以在消费端应用程序运行时进行更改。KCL 会定期调用 `streamConfigList` 来了解要处理的数据流的变化。

该 `streamConfigList` 方法填充 [StreamConfig](#) 列表。

```
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;
}
```

请注意，`StreamIdentifier` 和 `InitialPositionInStreamExtended` 是必填字段，`consumerArn` 是选填字段。只有在使用 KCL 2.x 实现增强型扇出消费端应用程序时，才必须提供 `consumerArn`。

有关的更多信息 `StreamIdentifier`，请参阅 [#L129](https://github.com/awslabs/amazon-kinesis-client/blob/v2.5.8/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java)。要创建 `StreamIdentifier`，我们建议您从 v2.5.0 及更高版本提供的 `streamArn` 和 `streamCreationEpoch` 创建一个多流实例。在不支持 `streamArn` 的 KCL v2.3 和 v2.4 中，请使用格式 `account-id:StreamName:streamCreationTimestamp` 创建一个多流实例。从下一个主要版本开始，此格式将弃用且不再受支持。

`MultistreamTracker` 还包括可删除租约表中旧流租约的策略

`(formerStreamsLeasesDeletionStrategy)`。请注意，在消费端应用程序运行时无法更改策略。欲了解更多信息，请参阅 <https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0.java#L129>

- `ConfigsBuilder` 是一个应用程序范围的类，可用于指定构建 KCL 使用者应用程序时要使用的所有 KCL 2.x 配置设置。`ConfigsBuilder` 类现在支持该 `MultistreamTracker` 接口。你可以用一个数据流的名称进行初始化 `ConfigsBuilder` 以使用来自以下内容的记录：

```
/**
 * 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;
}
```

或者，`MultiStreamTracker`如果要实现同时处理多个流的 KCL 使用者应用程序，也可以使用进行初始化 `ConfigsBuilder`。

```
* 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;
    }
```

- 通过为您的 KCL 消费端应用程序实现多流支持功能，应用程序租约表的每一行现在都包含该应用程序处理的多个数据流的分片 ID 和流名称。
- 在为 KCL 消费端应用程序实现多流支持功能后，leaseKey 采用以下结构：account-id:StreamName:streamCreationTimestamp:ShardId。例如 111111111:multiStreamTest-1:12345:shardId-000000000336。

 **Important**

当现有 KCL 消费端应用程序配置为仅处理一个数据流时，leaseKey（租约表的哈希键）就是分片 ID。如果您将此现有 KCL 消费端应用程序重新配置为处理多个数据流，则会破坏租约表，因为支持多流处理功能后，leaseKey 必须采用如下结构：account-id:StreamName:StreamCreationTimestamp:ShardId。

将 KCL 与Amazon Glue架构注册表一起使用

您可以将 Kinesis 数据流与Amazon Glue架构注册表集成。Amazon Glue 架构注册表能帮助您集中发现、控制和演变架构，同时确保注册架构持续验证生成的数据。架构定义了数据记录的结构和格式。架构是用于可靠数据发布、使用或存储的版本化规范。Amazon GlueSchema Registry end-to-end 由允许您改善流媒体应用程序中的数据质量和数据治理。有关更多信息，请参阅 [Amazon Glue Schema Registry](#)。设置此集成的方法之一是使用适用于 Java 的 KCL。

 **Important**

目前，只有使用用 Java 实现的 KCL 2.3 使用者的 Kinesis 数据流支持 Kinesis Streams Amazon Glue 和 Schema Registry 集成。不提供多语言支持。不支持 KCL 1.0 消费端。不支持 KCL 2.3 之前的 KCL 2.x 消费端。

有关如何使用 KCL 设置 Kinesis Data Streams 与架构注册表集成的详细说明，请参阅“[用例：将 Amazon Kinesis Data Streams 与 Glue KPL/KCL 架构注册表集成](#)”中的“[使用库与Amazon数据交互](#)”部分。

开发具有共享吞吐量的自定义消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

如果您在从 Kinesis Data Streams 接收数据时不需要专用吞吐量，并且在 200 毫秒下不需要读取传播延迟，则可以构建消费端应用程序，如以下主题所述。您可以使用 Kinesis Client Library (KCL) 或适用于 Java 的 Amazon SDK。

主题

- [使用 KCL 开发具有共享吞吐量的自定义消费端](#)

有关使用专用吞吐量构建可从 Kinesis Data Streams 接收记录的消费端的信息，请参阅 [开发具有专用吞吐量的增强扇出型消费端](#)。

使用 KCL 开发具有共享吞吐量的自定义消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

开发具有共享吞吐量的定制消费端应用程序的方法之一是使用 Kinesis Client Library (KCL)。

根据您正在使用的 KCL 版本，从以下主题进行选择。

主题

- [开发 KCL 1.x 消费端](#)

- [开发 KCL 2.x 消费端](#)

开发 KCL 1.x 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

您可以使用 Kinesis Client Library (KCL) 为 Amazon Kinesis Data Streams 开发消费端应用程序。

有关 KCL 的更多信息，请参阅 [关于 KCL \(先前版本 \)](#)。

根据要使用的选项，从以下主题进行选择。

内容

- [在 Java 中开发 Kinesis Client Library 消费端](#)
- [在 Node.js 中开发 Kinesis Client Library 消费端](#)
- [在 .NET 中开发 Kinesis Client Library 消费端](#)
- [在 Python 中开发 Kinesis Client Library 消费端](#)
- [在 Ruby 中开发 Kinesis Client Library 消费端](#)

在 Java 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 Java。要查看 Javadoc 参考资料，请参阅类的 [AmazonJavadoc 主题](#)。AmazonKinesisClient

要从中下载 Java KCL GitHub，请前往 [Kinesis 客户端库 \(Java\)](#)。要查找 Apache Maven 上的 Java KCL，请转至 [KCL 搜索结果](#) 页。要从中下载 Java KCL 使用者应用程序的示例代码 GitHub，请转到上方的 [KCL for Java 示例项目](#) 页面。GitHub

该示例应用程序使用 [Apache Commons Logging](#)。可以使用 `configure` 文件中定义的静态 `AmazonKinesisApplicationSample.java` 方法更改日志记录配置。有关如何在 Log4j 和 Amazon Java 应用程序中使用 Apache 共享日志记录的更多信息，请参阅《开发人员指南》中的使用 [Log4j 进行日志记录](#)。适用于 Java 的 Amazon SDK

在 Java 中实现 KCL 消费端应用程序时，您必须完成下列任务：

任务

- [实现 IRecord处理器方法](#)
- [为 IRecord处理器接口实现类工厂](#)
- [创建工作线程](#)
- [修改配置属性](#)
- [迁移到版本 2 的记录处理器接口](#)

实现 IRecord处理器方法

KCL 当前支持 `IRecordProcessor` 接口的两个版本：原始接口可与第一个版本的 KCL 一起可用；而版本 2 从 KCL 1.5.0 版才开始可用。这两个接口都完全受支持。您的选择取决于您的特定方案要求。要查看所有区别，请参阅您在本地构建的 Javadocs 或源代码。以下各节概述了开始使用的最低实施要求。

IRecord处理器版本

- [原始接口 \(版本 1 \)](#)
- [更新后的接口 \(版本 2 \)](#)

原始接口 (版本 1)

原始 `IRecordProcessor` 接口 (package `com.amazonaws.services.kinesis.clientlibrary.interfaces`) 公开了下列记

录处理器方法，您的消费端必须实施这些方法。该示例提供了可用作起点的实现（请参阅 [AmazonKinesisApplicationSampleRecordProcessor.java](#)）。

```
public void initialize(String shardId)
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
checkpointer)
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

初始化

KCL 在实例化记录处理器时调用 `initialize` 方法，并将特定分片 ID 作为参数传递。此记录处理器仅处理此分片，并且通常情况下反过来说也成立（此分片仅由此记录处理器处理）。但是，您的消费端应该考虑数据记录可能会经过多次处理的情况。Kinesis Data Streams 具有至少一次语义，即分片中的每个数据记录至少会由消费端中的工作程序处理一次。有关特定分片可能由多个工作程序进行处理的情况的更多信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

```
public void initialize(String shardId)
```

processRecords

KCL 调用 `processRecords` 方法，并传递来自由 `initialize(shardId)` 方法指定的分片的数据记录的列表。记录处理器根据消费端的语义处理这些记录中的数据。例如，工作程序可能对数据执行转换，然后将结果存储在 Amazon Simple Storage Service (Amazon S3) 存储桶中。

```
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
checkpointer)
```

除了数据本身之外，记录还包含一个序号和一个分区键。工作程序可在处理数据时使用这些值。例如，工作线程可选择 S3 存储桶，并在其中根据分区键的值存储数据。`Record` 类公开了以下方法，这些方法提供对记录的数据、序列号和分区键的访问。

```
record.getData()
record.getSequenceNumber()
record.getPartitionKey()
```

在该示例中，私有方法 `processRecordsWithRetries` 具有显示工作程序如何访问记录的数据、序号和分区键的代码。

Kinesis Data Streams 需要记录处理器来跟踪已在分片中处理的记录。KCL 通过将检查指针 (`IRecordProcessorCheckpointer`) 传递到 `processRecords` 来为您执行此跟踪。记录处理器

将对此接口调用 `checkpoint` 方法，以向 KCL 告知记录处理器处理分片中的记录的进度。如果工作程序失败，KCL 将使用此信息在已知的上一个已处理记录处重新启动对分片的处理。

对于拆分或合并操作，在原始分片的处理器调用 `checkpoint` 以指示原始分片上的所有处理操作都已完成之前，KCL 不会开始处理新分片。

如果您未传递参数，KCL 将假定对 `checkpoint` 的调用表示所有记录都已处理，一直处理到传递到记录处理器的最后一个记录。因此，记录处理器只应在已处理传递到它的列表中的所有记录后才调用 `checkpoint`。记录处理器不需要在每次调用 `checkpoint` 时调用 `processRecords`。例如，处理器可以在每第三次调用 `checkpoint` 时调用 `processRecords`。您可以选择性地将某个记录的确切序号指定为 `checkpoint` 的参数。在本例中，KCL 将假定所有记录都已处理，直至处理到该记录。

在该示例中，私有方法 `checkpoint` 展示了如何使用适当的异常处理和重试逻辑调用 `IRecordProcessorCheckpointer.checkpoint`。

KCL 依靠 `processRecords` 来处理由处理数据记录引起的任何异常。如果 `processRecords` 引发了异常，则 KCL 将跳过在异常发生前已传递的数据记录。也就是说，这些记录不会重新发送到引发异常的记录处理器或消费端中的任何其他记录处理器。

shutdown

KCL 在处理结束（关闭原因为 `TERMINATE`）或工作程序不再响应（关闭原因为 `ZOMBIE`）时调用 `shutdown` 方法。

```
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

处理操作在记录处理器不再从分片中接收任何记录时结束，因为分片已被拆分或合并，或者流已删除。

KCL 还会将 `IRecordProcessorCheckpointer` 接口传递到 `shutdown`。如果关闭原因为 `TERMINATE`，则记录处理器应完成处理任何数据记录，然后对此接口调用 `checkpoint` 方法。

更新后的接口（版本 2）

更新后的 `IRecordProcessor` 接口（package `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`）公开了下列记录处理器方法，您的消费端必须实施这些方法：

```
void initialize(InitializationInput initializationInput)
void processRecords(ProcessRecordsInput processRecordsInput)
```

```
void shutdown(ShutdownInput shutdownInput)
```

原始版本的接口中的所有参数可通过容器对象上的 `get` 方法进行访问。例如，要检索 `processRecords()` 中的记录的列表，可使用 `processRecordsInput.getRecords()`。

自此接口的版本 2 (KCL 1.5.0 及更高版本) 起，除了原始接口提供的输入之外，以下新输入也可用：

起始序列号

在传递给 `InitializationInput` 运算的 `initialize()` 对象中，将提供给记录处理器实例的记录的起始序列号。这是由之前处理同一分片的记录处理器实例进行最近一次检查点操作的序列号。此序列号在您的应用程序需要此信息时提供。

待进行检查点操作的序列号

在传递给 `initialize()` 运算的 `InitializationInput` 对象中，在上一个记录处理器实例停止前可能无法提交的待进行检查点操作的序列号 (如果有)。

为 `IRecordProcessor` 接口实现类工厂

您还需要为实现记录处理器方法的类实现一个工厂。当消费端实例化工作程序时，它将传递对此工厂的引用。

以下示例使用原始记录处理器接口在文件

`AmazonKinesisApplicationSampleRecordProcessorFactory.java` 中实现工厂类。如果您希望此工厂类创建版本 2 记录处理器，请使用程序包名称 `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`。

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {  
    /**  
     * Constructor.  
     */  
    public SampleRecordProcessorFactory() {  
        super();  
    }  
    /**  
     * {@inheritDoc}  
     */  
    @Override  
    public IRecordProcessor createProcessor() {  
        return new SampleRecordProcessor();  
    }  
}
```

```
    }  
}
```

创建工作线程

如 [实现 IRecord处理器方法](#) 中所述，有两个版本的 KCL 记录处理器接口可供选择，这将影响您创建工作线程的方式。原始记录处理器接口使用以下代码结构创建工作线程：

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)  
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();  
final Worker worker = new Worker(recordProcessorFactory, config);
```

当使用版本 2 的记录处理器接口时，您可使用 `Worker.Builder` 创建工作线程而无需担心要使用的构造函数以及参数的顺序。更新后的记录处理器接口使用以下代码结构创建工作线程：

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

修改配置属性

该示例提供了配置属性的默认值。工作程序的此配置数据随后将整合到 `KinesisClientLibConfiguration` 对象中。此对象和对 `IRecordProcessor` 的类工厂的引用将传入用于实例化工作程序的调用。您可借助 Java 属性文件（请参阅 `AmazonKinesisApplicationSample.java`）用您自己的值覆盖任何这些属性。

应用程序名称

KCL 需要一个应用程序名称，该名称在您的应用程序中以及同一区域的 Amazon DynamoDB 表中处于唯一状态。KCL 通过以下方法使用应用程序名称配置值：

- 假定所有与此应用程序名称关联的工作程序在同一数据流上合作。这些工作程序可被分配到多个实例上。如果运行同一应用程序代码的其他实例，但使用不同的应用程序名称，则 KCL 会将第二个实例视为在同一数据流运行的完全独立的应用程序。
- KCL 利用应用程序名称创建 DynamoDB 表并使用该表保留应用程序的状态信息（如检查点和工作程序-分片映射）。每个应用程序都有自己的 DynamoDB 表。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

设置凭证

您必须将您的Amazon证书提供给默认凭证提供者链中的一个凭证提供商。例如，如果您在 EC2 实例上运行使用器，我们建议您使用 IAM 角色启动该实例。Amazon 反映与此 IAM 角色关联的权限的证书可通过实例元数据提供给实例上的应用程序。对于在 EC2 实例上运行的使用者，这是管理凭证的最安全的方式。

示例应用程序首先尝试从实例元数据中检索 IAM 凭证：

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

如果示例应用程序无法从实例元数据中获取凭证，它会尝试从属性文件中检索凭证：

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

有关实例元数据的更多信息，请参阅 Amazon EC2 用户指南中的[实例元数据](#)。

将工作线程 ID 用于多个实例

示例初始化代码通过使用本地计算机的名称并附加一个全局唯一的标识符为工作程序创建 ID (workerId)，如以下代码段所示。此方法支持消费端应用程序的多个实例在单台计算机上运行的方案。

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +  
UUID.randomUUID();
```

迁移到版本 2 的记录处理器接口

如果要迁移使用原始接口的代码，则除了上述步骤之外，还需要执行以下步骤：

1. 更改您的记录处理器类以导入版本 2 记录处理器接口：

```
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

2. 更改对输入的引用以在容器对象上使用 get 方法。例如，在 shutdown() 运算中，将“checkpointer”更改为“shutdownInput.getCheckpointer()”。
3. 更改您的记录处理器工厂类以导入版本 2 记录处理器工厂接口：

```
import  
com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
```

4. 更改工作线程的结构以使用 `Worker.Builder`。例如：

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

在 Node.js 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 Node.js。

KCL 是一个 Java 库；对 Java 以外其他语言的支持是使用名为的多语言接口提供的。MultiLangDaemon 此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后台运行。因此，如果您安装适用于 Node.js 的 KCL 并完全使用 Node.js 编写消费者应用程序，则仍然需要在系统上安装 Java，因为 MultiLangDaemon 此外 MultiLangDaemon，您可能需要根据自己的用例自定义一些默认设置，例如它所连接的 Amazon 区域。有关 MultiLangDaemon 的更多信息 GitHub，请访问 [KCL MultiLangDaemon 项目](#) 页面。

要从中下载 Node.js KCL GitHub，请前往 [Kinesis 客户端库 \(Node.js\)](#)。

示例代码下载

Node.js 中有两个代码示例可用于 KCL：

- [basic-sample](#)

在下列节中用于阐释在 Node.js 中构建 KCL 消费端应用程序的基础知识。

- [click-stream-sample](#)

稍微复杂一些，使用了现实世界的情景，适合在您熟悉基本示例代码之后采用。此示例在这里不做讨论，但它有一个包含更多信息的自述文件。

在 Node.js 中实现 KCL 消费端应用程序时，您必须完成下列任务：

任务

- [实现记录处理器](#)
- [修改配置属性](#)

实现记录处理器

使用适用于 Node.js 的 KCL 的最简易的潜在消费端必须实现 `recordProcessor` 函数，该函数反之包含函数 `initialize`、`processRecords` 和 `shutdown`。该示例提供了可用作起点的实现（请参阅 [sample_kcl_app.js](#)）。

```
function recordProcessor() {  
    // return an object that implements initialize, processRecords and shutdown  
    // functions.  
}
```

初始化

KCL 在记录处理器启动时调用 `initialize` 函数。此记录处理器只处理作为 `initializeInput.shardId` 传递的分片 ID，并且通常情况下反过来说也成立（此分片只能由此记录处理器处理）。但是，您的消费端应该考虑数据记录可能会经过多次处理的情况。这是因为 Kinesis Data Streams 具有至少一次语义，即分片中的每个数据记录在您的消费端中由工作程序至少处理一次。有关特定分片可能由多个工作线程处理的情况的更多信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

```
initialize: function(initializeInput, completeCallback)
```

processRecords

KCL 使用包含一个数据记录的列表（这些记录来自在 `initialize` 函数中指定的分片）的输入来调用此函数。您实现的记录处理器根据您的消费端的语义处理这些记录中的数据。例如，工作程序可能对数据执行转换，然后将结果存储在 Amazon Simple Storage Service (Amazon S3) 存储桶中。

```
processRecords: function(processRecordsInput, completeCallback)
```

除了数据本身之外，记录还包含工作程序在处理数据时可使用的序号和分区键。例如，工作线程可选择 S3 存储桶，并在其中根据分区键的值存储数据。record 词典公开了以下键-值对来访问记录的数据、序号和分区键：

```
record.data  
record.sequenceNumber  
record.partitionKey
```

请注意，数据是 Base64 编码的。

在该基本示例中，函数 `processRecords` 具有显示工作程序如何访问记录的数据、序号和分区键的代码。

Kinesis Data Streams 需要记录处理器来跟踪已在分片中处理的记录。KCL 利用作为 `processRecordsInput.checkpointer` 传递的 `checkpointer` 对象执行此跟踪。您的记录处理器将调用 `checkpointer.checkpoint` 函数以向 KCL 告知记录处理器处理分片中的记录的进度。如果工作程序失败，KCL 将在您重新启动分片的处理时使用此信息，以便在上一个已知的已处理记录处继续处理。

对于拆分或合并操作，在原始分片的处理器调用 `checkpoint` 以指示原始分片上的所有处理操作都已完成之前，KCL 不会开始处理新分片。

如果您未将序列号传递到 `checkpoint` 函数，KCL 将假定对 `checkpoint` 的调用表示所有记录都已处理，一直处理到传递到记录处理器的最后一个记录。因此，记录处理器只应在已处理传递到它的列表中的所有记录后才调用 `checkpoint`。记录处理器不需要在每次调用 `checkpoint` 时调用 `processRecords`。例如，处理器可以调用 `checkpoint` 用每三次呼叫或记录处理器外部的某个事件（例如您实现的自定义 verification/validation 服务）。

您可以选择性地将某个记录的确切序号指定为 `checkpoint` 的参数。在本例中，KCL 将假定所有记录都已处理，直至处理到该记录。

基本示例应用程序显示了对 `checkpointer.checkpoint` 函数最简单的调用。您此时可以在该函数中为您的消费端添加您需要的其他检查点逻辑。

`shutdown`

KCL 在处理结束（`shutdownInput.reason` 为 `TERMINATE`）或工作程序不再响应（`shutdownInput.reason` 为 `ZOMBIE`）时调用 `shutdown` 函数。

```
shutdown: function(shutdownInput, completeCallback)
```

处理操作在记录处理器不再从分片中接收任何记录时结束，因为分片已被拆分或合并，或者流已删除。

KCL 还会将 `shutdownInput.checkpointer` 对象传递到 `shutdown`。如果关闭原因为 `TERMINATE`，则应确保记录处理器已完成处理任何数据记录，然后对此接口调用 `checkpoint` 函数。

修改配置属性

该示例提供了配置属性的默认值。您可使用自己的值覆盖任何这些属性（请参阅基本示例中的 `sample.properties`）。

应用程序名称

KCL 需要一个应用程序，该应用程序在您的各个应用程序中以及同一区域的各个 Amazon DynamoDB 表中处于唯一状态。KCL 通过以下方法使用应用程序名称配置值：

- 假定所有与此应用程序名称关联的工作程序在同一数据流上合作。这些工作程序可被分配到多个实例上。如果运行同一应用程序代码的其他实例，但使用不同的应用程序名称，则 KCL 会将第二个实例视为在同一数据流运行的完全独立的应用程序。
- KCL 利用应用程序名称创建 DynamoDB 表并使用该表保留应用程序的状态信息（如检查点和工作程序-分片映射）。每个应用程序都有自己的 DynamoDB 表。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

设置凭证

您必须将您的Amazon证书提供给默认凭证提供者链中的一个凭证提供商。可以使用 `AWSCredentialsProvider` 属性设置凭证提供程序。`sample.properties` 文件必须向 [默认凭证提供程序链](#) 中的凭证提供程序之一提供您的凭证。如果您在 Amazon EC2 实例上运行使用器，我们建议您使用 IAM 角色配置该实例。Amazon 反映与此 IAM 角色关联的权限的证书可通过实例元数据提供给实例上的应用程序。这是管理 EC2 实例上运行的使用者应用程序凭证的最安全的方式。

以下示例配置 KCL 以使用 `sample_kcl_app.js` 中提供的记录处理器，处理名为 `kclnodejssample` 的 Kinesis 数据流。

```
# 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 defaultAmazoncredentials provider chain
```

```
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain
# Read from the beginning of the stream
initialPositionInStream = TRIM_HORIZON
```

在 .NET 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 .NET。

KCL 是一个 Java 库；对 Java 以外其他语言的支持是使用名为的多语言接口提供的。MultiLangDaemon 此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后台运行。因此，如果您安装适用于 .NET 的 KCL 并完全使用 .NET 编写使用者应用程序，则仍然需要在系统上安装 Java，因为。MultiLangDaemon 此外 MultiLangDaemon，您可能需要根据自己的用例自定义一些默认设置，例如它所连接的 Amazon 区域。有关 MultiLangDaemon 的更多信息 GitHub，请访问 [KCL MultiLangDaemon 项目](#) 页面。

要从中下载 .NET KCL GitHub，请访问 [Kinesis 客户端库 \(.NET\)](#)。要下载 .NET KCL 使用者应用程序的示例代码，请转到 [KCL for .NET 使用者项目示例](#) 页面。GitHub

在 .NET 中实现 KCL 消费端应用程序时，您必须完成下列任务：

任务

- [实现 IRecordProcessor 类方法](#)
- [修改配置属性](#)

实现 IRecordProcessor 类方法

消费端必须实现适用于 `IRecordProcessor` 的以下方法。示例消费端提供了可用作起点的实现（请参阅 `SampleRecordProcessor` 中的 `SampleConsumer/AmazonKinesisSampleConsumer.cs` 类）。

```
public void Initialize(InitializationInput input)
public void ProcessRecords(ProcessRecordsInput input)
public void Shutdown(ShutdownInput input)
```

初始化

KCL 在实例化记录处理器时调用此方法，并将特定分片 ID 传入 `input` 参数 (`input.ShardId`)。此记录处理器只处理此分片，并且通常情况下反过来也成立（此分片只能由此记录处理器处理）。但是，您的消费端应该考虑数据记录可能会经过多次处理的情况。这是因为 Kinesis Data Streams 具有至少一次语义，即分片中的每个数据记录在您的消费端中由工作程序至少处理一次。有关特定分片可能由多个工作线程处理的情况的更多信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

```
public void Initialize(InitializationInput input)
```

ProcessRecords

KCL 调用此方法，并将由 `Initialize` 方法指定的分片中的数据记录的列表传入 `input` 参数 (`input.Records`)。您实现的记录处理器根据您的消费端的语义处理这些记录中的数据。例如，工作程序可能对数据执行转换，然后将结果存储在 Amazon Simple Storage Service (Amazon S3) 存储桶中。

```
public void ProcessRecords(ProcessRecordsInput input)
```

除了数据本身之外，记录还包含一个序号和一个分区键。工作程序可在处理数据时使用这些值。例如，工作线程可选择 S3 存储桶，并在其中根据分区键的值存储数据。`Record` 类公开了以下代理来访问记录的数据、序号和分区键：

```
byte[] Record.Data
string Record.SequenceNumber
string Record.PartitionKey
```

在该示例中，方法 `ProcessRecordsWithRetries` 具有显示工作程序如何访问记录的数据、序号和分区键的代码。

Kinesis Data Streams 需要记录处理器来跟踪已在分片中处理的记录。KCL 通过将 `Checkpointer` 对象传递到 `ProcessRecords` (`input.Checkpointer`) 来为您执行此跟踪。记录处理器将调用 `Checkpointer.Checkpoint` 方法以向 KCL 告知记录处理器处理分片中的记录的进度。如果工作程序失败，KCL 将使用此信息在已知的上一个已处理记录处重新启动对分片的处理。

对于拆分或合并操作，在原始分片的处理器调用 `Checkpointer.Checkpoint` 以指示原始分片上的所有处理操作都已完成之前，KCL 不会开始处理新分片。

如果您未传递参数，KCL 将假定对 `Checkpointer.Checkpoint` 的调用表示所有记录都已处理，一直处理到传递到记录处理器的最后一个记录。因此，记录处理器只应在已处理传递到它的列表中的所有记录后才调用 `Checkpointer.Checkpoint`。记录处理器不需要在每次调用 `Checkpointer.Checkpoint` 时调用 `ProcessRecords`。例如，处理器在每第三次或第四次调用时调用 `Checkpointer.Checkpoint`。您可以选择性地将某个记录的确切序号指定为 `Checkpointer.Checkpoint` 的参数。在本例中，KCL 将假定记录都已处理，直至处理到该记录。

在该示例中，私有方法 `Checkpoint(Checkpointer checkpointer)` 展示了如何使用适当的异常处理和重试逻辑调用 `Checkpointer.Checkpoint` 方法。

适用于 .NET 的 KCL 处理异常的方式不同于其他 KCL 语言库，前者不处理因处理数据记录而引起的任何异常。用户代码中未捕获的任何异常都将使程序崩溃。

关闭

KCL 在处理结束（关闭原因为 `TERMINATE`）或工作程序不再响应（关闭 `input.Reason` 值为 `ZOMBIE`）时调用 `Shutdown` 方法。

```
public void Shutdown(ShutdownInput input)
```

处理操作在记录处理器不再从分片中接收任何记录时结束，因为分片已被拆分或合并，或者流已删除。

KCL 还会将 `Checkpointer` 对象传递到 `shutdown`。如果关闭原因为 `TERMINATE`，则记录处理器应完成处理任何数据记录，然后对此接口调用 `checkpoint` 方法。

修改配置属性

示例消费端提供了配置属性的默认值。您可使用自己的值覆盖任何这些属性（请参阅 `SampleConsumer/kcl.properties`）。

应用程序名称

KCL 需要一个应用程序，该应用程序在您的各个应用程序中以及同一区域的各个 Amazon DynamoDB 表中处于唯一状态。KCL 通过以下方法使用应用程序名称配置值：

- 假定所有与此应用程序名称关联的工作程序在同一数据流上合作。这些工作程序可被分配到多个实例上。如果运行同一应用程序代码的其他实例，但使用不同的应用程序名称，则 KCL 会将第二个实例视为在同一数据流运行的完全独立的应用程序。

- KCL 利用应用程序名称创建 DynamoDB 表并使用该表保留应用程序的状态信息（如检查点和工作程序-分片映射）。每个应用程序都有自己的 DynamoDB 表。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

设置凭证

您必须将您的Amazon证书提供给默认凭证提供者链中的一个凭证提供商。可以使用 `AWSCredentialsProvider` 属性设置凭证提供程序。[sample.properties](#) 必须向默认凭证提供程序链中的凭证提供程序之一提供您的凭证。如果您在 EC2 实例上运行使用者应用程序，我们建议您使用 IAM 角色配置该实例。Amazon反映与此 IAM 角色关联的权限的证书可通过实例元数据提供给实例上的应用程序。对于在 EC2 实例上运行的使用者，这是管理凭证的最安全的方式。

该示例的属性文件将配置 KCL 以使用 `AmazonKinesisSampleConsumer.cs` 中提供的记录处理器处理名为“words”的 Kinesis 数据流。

在 Python 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub有关最新 KCL 版本的信息，请参阅[使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅[从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 Python。

KCL 是一个 Java 库；对 Java 以外其他语言的支持是使用名为的多语言接口提供的。MultiLangDaemon此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后台运行。因此，如果您安装适用于 Python 的 KCL 并完全使用 Python 编写消费者应用程序，则仍然需要在系统上安装 Java，因为。MultiLangDaemon此外 MultiLangDaemon，您可能需要根据自己的用例自定义一些默认设置，例如它所连接的Amazon区域。有关 MultiLangDaemon on 的更多信息 GitHub，请访问 [KCL MultiLangDaemon 项目](#) 页面。

要从中下载 Python KCL GitHub，请前往 [Kinesis 客户端库 \(Python\)](#)。要下载 Python KCL 使用者应用程序的示例代码，请转到上的 [KCL for Python 示例项目](#) 页面。GitHub

在 Python 中实现 KCL 消费端应用程序时，您必须完成下列任务：

任务

- [实现 RecordProcessor 类方法](#)
- [修改配置属性](#)

实现 RecordProcessor 类方法

RecordProcessor 类必须扩展 RecordProcessorBase 以实现以下方法。该示例提供了可用作起点的实现（请参阅 `sample_kclpy_app.py`）。

```
def initialize(self, shard_id)
def process_records(self, records, checkpointer)
def shutdown(self, checkpointer, reason)
```

初始化

KCL 在实例化记录处理器时调用 `initialize` 方法，并将特定分片 ID 作为参数传递。此记录处理器只处理此分片，并且通常情况下反过来说也成立（此分片只能由此记录处理器处理）。但是，您的消费端应该考虑数据记录可能会经过多次处理的情况。这是因为 Kinesis Data Streams 具有至少一次语义，即分片中的每个数据记录在您的消费端中由工作程序至少处理一次。有关特定分片可能由多个工作程序进行处理的情况的更多信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

```
def initialize(self, shard_id)
```

process_records

KCL 调用此方法，并传递由 `initialize` 方法指定的分片中的数据记录的列表。您实现的记录处理器根据您的消费端的语义处理这些记录中的数据。例如，工作程序可能对数据执行转换，然后将结果存储在 Amazon Simple Storage Service (Amazon S3) 存储桶中。

```
def process_records(self, records, checkpointer)
```

除了数据本身之外，记录还包含一个序号和一个分区键。工作程序可在处理数据时使用这些值。例如，工作线程可选择 S3 存储桶，并在其中根据分区键的值存储数据。`record` 词典公开了以下键-值对来访问记录的数据、序号和分区键：

```
record.get('data')
record.get('sequenceNumber')
record.get('partitionKey')
```

请注意，数据是 Base64 编码的。

在该示例中，方法 `process_records` 具有显示工作程序如何访问记录的数据、序号和分区键的代码。

Kinesis Data Streams 需要记录处理器来跟踪已在分片中处理的记录。KCL 通过将 `Checkpointer` 对象传递到 `process_records` 来为您执行此跟踪。记录处理器将对此对象调用 `checkpoint` 方法，以向 KCL 告知记录处理器处理分片中的记录的进度。如果工作程序失败，KCL 将使用此信息在已知的上一个已处理记录处重新启动对分片的处理。

对于拆分或合并操作，在原始分片的处理器调用 `checkpoint` 以指示原始分片上的所有处理操作都已完成之前，KCL 不会开始处理新分片。

如果您未传递参数，KCL 将假定对 `checkpoint` 的调用表示所有记录都已处理，一直处理到传递到记录处理器的最后一个记录。因此，记录处理器只应在已处理传递到它的列表中的所有记录后才调用 `checkpoint`。记录处理器不需要在每次调用 `checkpoint` 时调用 `process_records`。例如，处理器可在每第三次调用时调用 `checkpoint`。您可以选择性地将某个记录的确切序号指定为 `checkpoint` 的参数。在本例中，KCL 将假定所有记录都已处理，直至处理到该记录。

在该示例中，私有方法 `checkpoint` 展示了如何使用适当的异常处理和重试逻辑调用 `Checkpointer.checkpoint` 方法。

KCL 依靠 `process_records` 来处理由处理数据记录引起的任何异常。如果 `process_records` 引发了异常，则 KCL 将跳过在异常发生前已传递到 `process_records` 的数据记录。也就是说，这些记录不会重新发送到引发异常的记录处理器或消费端中的任何其他记录处理器。

shutdown

KCL 在处理结束（关闭原因为 `TERMINATE`）或工作程序不再响应（关闭 `reason` 为 `ZOMBIE`）时调用 `shutdown` 方法。

```
def shutdown(self, checkpointer, reason)
```

处理操作在记录处理器不再从分片中接收任何记录时结束，因为分片已被拆分或合并，或者流已删除。

KCL 还会将 `Checkpointer` 对象传递到 `shutdown`。如果关闭 `reason` 是 `TERMINATE`，则记录处理器应完成处理任何数据记录，然后对此接口调用 `checkpoint` 方法。

修改配置属性

该示例提供了配置属性的默认值。您可使用自己的值覆盖任何这些属性（请参阅 `sample.properties`）。

应用程序名称

KCL 需要一个应用程序名称，该名称在您的各个应用程序中以及同一区域的各个 Amazon DynamoDB 表中处于唯一状态。KCL 通过以下方法使用应用程序名称配置值：

- 假定与此应用程序名称关联的所有工作线程在同一个流上一起运行。这些工作线程可分布在多个实例上。如果运行同一应用程序代码的其他实例，但使用不同的应用程序名称，则 KCL 会将第二个实例视为在同一数据流运行的完全独立的应用程序。
- KCL 利用应用程序名称创建 DynamoDB 表并使用该表保留应用程序的状态信息（如检查点和工作程序-分片映射）。每个应用程序都有自己的 DynamoDB 表。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

设置凭证

您必须将您的Amazon证书提供给默认凭证提供者链中的一个凭证提供商。可以使用 `AWSCredentialsProvider` 属性设置凭证提供程序。[sample.properties](#) 必须向默认凭证提供程序链中的凭证提供程序之一提供您的凭证。如果您在 Amazon EC2 实例上运行使用者应用程序，我们建议您使用 IAM 角色配置该实例。Amazon反映与此 IAM 角色关联的权限的证书可通过实例元数据提供给实例上的应用程序。这是管理 EC2实例上运行的使用者应用程序凭证的最安全的方式。

该示例的属性文件将配置 KCL 以使用 `sample_kclpy_app.py` 中提供的记录处理器处理名为“words”的 Kinesis 数据流。

在 Ruby 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 Ruby。

KCL 是一个 Java 库；对 Java 以外其他语言的支持是使用名为的多语言接口提供的。MultiLangDaemon 此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后

台运行。因此，如果您安装适用于 Ruby 的 KCL 并完全使用 Ruby 编写消费者应用程序，则仍然需要在系统上安装 Java，因为 MultiLangDaemon 此外 MultiLangDaemon，您可能需要根据自己的用例自定义一些默认设置，例如它所连接的Amazon区域。有关 MultiLangDaemon on 的更多信息 GitHub，请访问 [KCL MultiLangDaemon 项目](#) 页面。

要从中下载 Ruby KCL GitHub，请前往 [Kinesis 客户端库 \(Ruby\)](#)。要下载 Ruby KCL 使用者应用程序的示例代码，请转到上的 [KCL for Ruby 示例项目](#) 页面。 GitHub

有关 KCL Ruby 支持库的更多信息，请参阅 [KCL Ruby Gems Documentation](#)。

开发 KCL 2.x 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。 GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

本主题将说明如何使用 2.0 版本的 Kinesis Client Library (KCL)。

有关 KCL 的更多信息，请参阅 [Developing Consumers Using the Kinesis Client Library 1.x](#) 中提供的概述。

根据要使用的选项，从以下主题进行选择。

主题

- [在 Java 中开发 Kinesis Client Library 消费端](#)
- [在 Python 中开发 Kinesis Client Library 消费端](#)
- [使用 KCL 2.x 开发具有增强扇出功能的消费端](#)

在 Java 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL

应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问[上的 Amazon Kinesis 客户端库页面](#)。GitHub有关最新 KCL 版本的信息，请参阅[使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅[从 KCL 1.x 迁移到 KCL 3.x](#)。

以下代码显示 ProcessorFactory 和 RecordProcessor 在 Java 中的实现示例。如果要利用增强型扇出功能，请参阅[利用使用增强型扇出功能的用户](#)。

```
/*
 * 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
 * theAmazonSDK to publish data.
 * Before running this program you must first create a Kinesis stream through
 * theAmazonconsole orAmazonSDK.
 */
```

```
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.randomAlphanumeric(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);
    }
}
}

}
```

在 Python 中开发 Kinesis Client Library 消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 Kinesis Client Library (KCL) 构建处理 Kinesis 数据流中数据的应用程序。Kinesis Client Library 提供多种语言版本。本主题将讨论 Python。

KCL 是一个 Java 库；对 Java 以外其他语言的支持是使用名为的多语言接口提供的。MultiLangDaemon 此进程守护程序基于 Java，当您使用 Java 以外的 KCL 语言时，该程序会在后台运行。因此，如果您安装适用于 Python 的 KCL 并完全使用 Python 编写消费者应用程序，则仍然需要在系统上安装 Java，因为。MultiLangDaemon 此外 MultiLangDaemon，您可能需要根据自己的用例自定义一些默认设置，例如它所连接的 Amazon 区域。有关 MultiLangDaemon 的更多信息 GitHub，请访问 [KCL MultiLangDaemon 项目](#) 页面。

要从中下载 Python KCL GitHub，请前往 [Kinesis 客户端库 \(Python\)](#)。要下载 Python KCL 使用者应用程序的示例代码，请转到 [KCL for Python 示例项目](#) GitHub 页面。

在 Python 中实现 KCL 消费端应用程序时，您必须完成下列任务：

任务

- [实现 RecordProcessor 类方法](#)
- [修改配置属性](#)

实现 RecordProcessor 类方法

RecordProcess 类必须扩展 RecordProcessorBase 类以实现以下方法：

```
initialize
process_records
shutdown_requested
```

此示例提供了可用作起点的实现。

```
#!/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
    """

    def initialize(self, shard_id, session_token, shard_info):
        print("Initialize shard %s" % shard_id)

    def process_records(self, records):
        print("Process %d records" % len(records))

    def shutdown(self, reason):
        print("Shutting down shard %s" % reason)
```

```
    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 record.
        :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()
```

修改配置属性

该示例提供了配置属性的默认值，如以下脚本所示。您可使用自己的值覆盖任何这些属性。

```
# 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 RecordProcessorCheckpoint#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
```

应用程序名称

KCL 需要一个应用程序名称，该名称在您的各个应用程序中以及同一区域的各个 Amazon DynamoDB 表中处于唯一状态。KCL 通过以下方法使用应用程序名称配置值：

- 假定与此应用程序名称关联的所有工作线程在同一个流上一起运行。这些工作线程可分布在多个实例中。如果运行同一应用程序代码的其他实例，但使用不同的应用程序名称，则 KCL 会将第二个实例视为在同一数据流运行的完全独立的应用程序。
- KCL 利用应用程序名称创建 DynamoDB 表并使用该表保留应用程序的状态信息（如检查点和工作程序-分片映射）。每个应用程序都有自己的 DynamoDB 表。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

凭据

您必须将您的Amazon证书提供给默认凭证提供者链中的一个凭证提供商。可以使用 `AWSCredentialsProvider` 属性设置凭证提供程序。如果您在 Amazon EC2 实例上运行使用者应用程序，我们建议您使用 IAM 角色配置该实例。Amazon 反映与此 IAM 角色关联的权限的证书可通过实例元数据提供给实例上的应用程序。这是管理 EC2 实例上运行的使用者应用程序凭证的最安全的方式。

使用 KCL 2.x 开发具有增强扇出功能的消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

在 Amazon Kinesis Data Streams 中使用增强型扇出功能的消费端，可以接收数据流中的记录，其中每分片每秒专用吞吐量高达 2MB 数据。此类消费端不必与接收流中数据的其他消费端争夺。有关更多信息，请参阅 [开发具有专用吞吐量的增强扇出型消费端](#)。

可以使用 2.0 版或更高版本的 Kinesis Client Library (KCL) 开发使用增强型扇出功能接收流中数据的应用程序。KCL 会自动为您的应用程序订阅流的所有分片，并确保您的使用者应用程序可以读取，每个分片的吞吐量值为 2 MB/sec。如果要在未开启增强型扇出功能的情况下使用 KCL，请参阅 [Developing Consumers Using the Kinesis Client Library 2.0](#)。

主题

- [使用 KCL 2.x 在 Java 中开发具有增强扇出功能的消费端](#)

使用 KCL 2.x 在 Java 中开发具有增强扇出功能的消费端

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

可以使用 2.0 版或更高版本的 Kinesis Client Library (KCL)，在 Amazon Kinesis Data Streams 中开发使用增强型扇出功能接收流中数据的应用程序。以下代码显示 ProcessorFactory 和 RecordProcessor 在 Java 中的实施示例。

建议您使用 KinesisClientUtil 创建 KinesisAsyncClient，并在 KinesisAsyncClient 中配置 maxConcurrency。

Important

Amazon Kinesis 客户端可能会看到延迟大幅增加，除非您将 KinesisAsyncClient 配置为具有足够高的 maxConcurrency，以允许所有租期以及额外使用 KinesisAsyncClient。

```
/*
 * 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);
        }
    }
}
```

将消费端从 KCL 1.x 迁移到 KCL 2.x

Important

Amazon Kinesis Client Library (KCL) 版本 1.x 和 2.x 已过时。KCL 1.x 将于 2026 年 1 月 30 日 end-of-support 上市。我们强烈建议您在 2026 年 1 月 30 日之前，将使用版本 1.x 的 KCL 应用程序迁移到最新的 KCL 版本。要查找最新的 KCL 版本，请访问上的 [Amazon Kinesis 客户端库页面](#)。GitHub 有关最新 KCL 版本的信息，请参阅 [使用 Kinesis Client Library](#)。有关从 KCL 1.x 迁移到 KCL 3.x 的信息，请参阅 [从 KCL 1.x 迁移到 KCL 3.x](#)。

本主题介绍 Kinesis Client Library (KCL) 版本 1.x 和 2.x 之间的区别。其中还向您展示如何将消费端从 KCL 版本 1.x 迁移到版本 2.x。在迁移您的客户端后，它将从最后一个检查点位置开始处理记录。

KCL 版本 2.0 引入了以下接口更改：

KCL 接口更改

| KCL 1.x 接口 | KCL 2.0 接口 |
|---|--|
| com.amazonaws.services.kine sis.clientlibrary.interface s.v2.IRecordProcessor | software.amazon.kinesis.pro cessor.ShardRecordProcessor |

| KCL 1.x 接口 | KCL 2.0 接口 |
|---|---|
| com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory | software.amazon.kinesis.processor.ShardRecordProcessorFactory |
| com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware | 折叠为 software.amazon.kinesis.processor.ShardRecordProcessor |

主题

- [迁移记录处理器类](#)
- [迁移记录处理器工厂](#)
- [迁移工作线程](#)
- [配置 Amazon Kinesis 客户端](#)
- [闲置时间删除](#)
- [客户端配置删除](#)

迁移记录处理器类

以下示例显示了为 KCL 1.x 实现的记录处理器：

```
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();
    }
}
}
```

迁移记录处理器类

1. 将接口从

com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor
和

com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotification
更改为 software.amazon.kinesis.processor.ShardRecordProcessor，如下所示：

```
// 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. 更新 import 和 initialize 方法的 processRecords 语句。

```
// 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. 使用以下新方法替换 shutdown 方法：leaseLost、shardEnded 和 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();
    }
}
```

下面是记录处理器类的更新版本。

```
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();
    }
}

}
```

迁移记录处理器工厂

记录处理器工厂负责在获得租约时创建记录处理器。下面是 KCL 1.x 工厂的示例。

```
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();
    }
}
```

迁移记录处理器工厂

1. 将已实施的接口从

`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory` 更改为 `software.amazon.kinesis.processor.ShardRecordProcessorFactory`，如下所示。

```
// 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. 更改 `createProcessor` 的返回签名。

```
// public IRecordProcessor createProcessor() {  
public ShardRecordProcessor shardRecordProcessor() {
```

下面是 2.0 中的记录处理器工厂的示例：

```
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();  
    }  
}
```

迁移工作线程

在 KCL 版本 2.0 中，名为 `Scheduler` 的新类取代了 `Worker` 类。下面是 KCL 1.x 工作程序的示例。

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

迁移工作程序

1. 将 `Worker` 类的 `import` 语句更改为 `Scheduler` 和 `ConfigsBuilder` 类的导入语句。

```
// import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;  
import software.amazon.kinesis.coordinator.Scheduler;  
import software.amazon.kinesis.common.ConfigsBuilder;
```

2. 创建 `ConfigsBuilder` 和 `Scheduler`，如以下示例所示。

建议您使用 `KinesisClientUtil` 创建 `KinesisAsyncClient`，并在 `KinesisAsyncClient` 中配置 `maxConcurrency`。

 **Important**

Amazon Kinesis 户端可能会看到延迟大幅增加，除非您将 `KinesisAsyncClient` 配置为具有足够高的 `maxConcurrency`，以允许所有租期以及额外使用 `KinesisAsyncClient`。

```
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()  
);
```

配置 Amazon Kinesis 客户端

随着 Kinesis Client Library 版本 2.0 的发布，客户端的配置已从单个配置类（`KinesisClientLibConfiguration`）变为 6 个配置类。下表描述了迁移。

配置字段及其新类

| 原始字段 | 新配置类 | 说明 |
|--|------------------------------|--|
| <code>applicationName</code> | <code>ConfigsBuilder</code> | 此 KCL 应用程序的名称。用作 <code>tableName</code> 和 <code>consumerName</code> 的默认名称。 |
| <code>tableName</code> | <code>ConfigsBuilder</code> | 允许覆盖用于 Amazon DynamoDB 租赁表的表名称。 |
| <code>streamName</code> | <code>ConfigsBuilder</code> | 此应用程序从其中处理记录的流的名称。 |
| <code>kinesisEndpoint</code> | <code>ConfigsBuilder</code> | 此选项已删除。请参阅“客户端配置删除”。 |
| <code>dynamoDBEndpoint</code> | <code>ConfigsBuilder</code> | 此选项已删除。请参阅“客户端配置删除”。 |
| <code>initialPositionInStreamExtended</code> | <code>RetrievalConfig</code> | 分片中 KCL 开始获取记录的位置，从应用程序的初始运行开始。 |
| <code>kinesisCredentialsProvider</code> | <code>ConfigsBuilder</code> | 此选项已删除。请参阅“客户端配置删除”。 |
| <code>dynamoDBCredentialProvider</code> | <code>ConfigsBuilder</code> | 此选项已删除。请参阅“客户端配置删除”。 |

| 原始字段 | 新配置类 | 说明 |
|--|-----------------------|-------------------------------------|
| cloudWatchCredentialsProvider | ConfigsBuilder | 此选项已删除。请参阅“客户端配置删除”。 |
| failoverTimeInMillis | LeaseManagementConfig | 在您可以将租赁所有者视为已失败之前必须经过的毫秒数。 |
| workerIdentifier | ConfigsBuilder | 表示应用程序处理器的这种实例化的唯一标识符。此值必须唯一。 |
| shardSyncIntervalMillis | LeaseManagementConfig | 分片同步调用之间的时间。 |
| maxRecords | PollingConfig | 允许设置 Kinesis 返回的最大记录数。 |
| idleTimeBetweenReadsInMillis | CoordinatorConfig | 此选项已删除。请参阅“闲置时间删除”。 |
| callProcessRecordsEvenForEmptyRecordList | ProcessorConfig | 如果设置，即使 Kinesis 中未提供任何记录，也会调用记录处理器。 |
| parentShardPollIntervalMillis | CoordinatorConfig | 记录处理器应轮询多少时间才能查看是否已完成父分片。 |
| cleanupLeasesUponShardCompletion | LeaseManagementConfig | 如果设置，只要子租赁已开始处理，即可删除租赁。 |

| 原始字段 | 新配置类 | 说明 |
|---|-----------------------|--|
| ignoreUnexpectedShardConfigs | LeaseManagementConfig | 如果设置，将忽略具有打开的分片的子分片。这主要适用于 DynamoDB Streams。 |
| kinesisClientConfig | ConfigsBuilder | 此选项已删除。请参阅“客户端配置删除”。 |
| dynamoDBClientConfig | ConfigsBuilder | 此选项已删除。请参阅“客户端配置删除”。 |
| cloudWatchClientConfig | ConfigsBuilder | 此选项已删除。请参阅“客户端配置删除”。 |
| taskBackoffTimeMillis | LifecycleConfig | 等待重试失败任务的时间。 |
| metricsBufferTimeMillis | MetricsConfig | 控制 CloudWatch 指标发布。 |
| metricsMaxQueueSize | MetricsConfig | 控制 CloudWatch 指标发布。 |
| metricsLevel | MetricsConfig | 控制 CloudWatch 指标发布。 |
| metricsEnabledDimensions | MetricsConfig | 控制 CloudWatch 指标发布。 |
| validateSequenceNumberBeforeCheckpointing | CheckpointConfig | 此选项已删除。请参阅“检查点序列号验证”。 |

| 原始字段 | 新配置类 | 说明 |
|--|-----------------------|---|
| regionName | ConfigsBuilder | 此选项已删除。请参阅“客户端配置删除”。 |
| maxLeasesForWorker | LeaseManagementConfig | 应用程序的单个实例应接受的最大租赁数量。 |
| maxLeasesToStealAtOneTime | LeaseManagementConfig | 应用程序一次应尝试窃取的最大租赁数量。 |
| initialLeaseTableReadCapacity | LeaseManagementConfig | 在 Kinesis 客户端库需要创建新的 DynamoDB 租用表时使用的 DynamoDB IOPs 读取。 |
| initialLeaseTableWriteCapacity | LeaseManagementConfig | 在 Kinesis 客户端库需要创建新的 DynamoDB 租用表时使用的 DynamoDB IOPs 读取。 |
| initialPositionInStreamExtended | LeaseManagementConfig | 应用程序应在流中开始的初始位置。此值仅在创建初始租赁时使用。 |
| skipShardSyncAtWorkerInitializationIfLeasesExist | CoordinatorConfig | 如果租赁表包含现有租赁，请禁用同步的分片数据。待办事项：KinesisEco-438 |
| shardPrioritization | CoordinatorConfig | 要使用的分片优先级。 |
| shutdownGraceMillis | 不适用 | 此选项已删除。参见“MultiLang 移除”。 |
| timeoutInSeconds | 不适用 | 此选项已删除。参见“MultiLang 移除”。 |

| 原始字段 | 新配置类 | 说明 |
|-------------------------------|-----------------------|---------------------------------------|
| retryGetRecordsInSeconds | PollingConfig | 配置 GetRecords 尝试失败之间的延迟。 |
| maxGetRecordsInThreadsPool | PollingConfig | 使用的线程池大小 GetRecords。 |
| maxLeaseRenewalThreads | LeaseManagementConfig | 控制租赁续订线程池的大小。您的应用程序可以容纳的租赁越多，此池应该就越大。 |
| recordsFetcherFactory | PollingConfig | 允许替换用于创建从流中检索的提取程序的工厂。 |
| logWarningForTaskAfterMillis | LifecycleConfig | 任务尚未完成的情况下在记录警告之前要等待的时长。 |
| listShardsBackoffTimeInMillis | RetrievalConfig | 发生故障时在调用 ListShards 之间要等待的时间（以毫秒为单位）。 |
| maxListShardsRetryAttempts | RetrievalConfig | ListShards 在放弃之前重试的最长时间。 |

闲置时间删除

在 KCL 版本 1.x 中，idleTimeBetweenReadsInMillis 对应了两个数量：

- 任务分派检查之间的时间量。您现在可以通过设置 CoordinatorConfig#shardConsumerDispatchPollIntervalMillis 来在任务之间配置此时间。
- 未从 Kinesis Data Streams 中返回任何记录时的睡眠时间量。在版本 2.0 中，带增强型扇出功能的记录是从其各自的检索器中推送的。分片消费端上的活动仅发生在推送的请求到达时。

客户端配置删除

在版本 2.0 中，KCL 不再创建客户端。这取决于用户提供有效的客户端。进行此更改后，已删除控制客户端创建的所有配置参数。如果需要这些参数，您可以在向 `ConfigsBuilder` 提供客户端之前在客户端上设置它们。

| 已删除字段 | 等效配置 |
|-------------------------------------|--|
| <code>kinesisEndpoint</code> | 使用以下首选端点配置开发工具包 <code>KinesisAsyncClient</code> ： <code>KinesisAsyncClient.builder().endpointOverride(URI.create("https://<kinesis endpoint>")).build()</code> 。 |
| <code>dynamoDBEndpoint</code> | 使用以下首选端点配置开发工具包 <code>DynamoDbAsyncClient</code> ： <code>DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://<dynamodb endpoint>")).build()</code> 。 |
| <code>kinesisClientConfig</code> | 使用以下所需配置来配置开发工具包 <code>KinesisAsyncClient</code> ： <code>KinesisAsyncClient.builder().overrideConfiguration(<your configuration>).build()</code> 。 |
| <code>dynamoDBClientConfig</code> | 使用以下所需配置来配置开发工具包 <code>DynamoDbAsyncClient</code> ： <code>DynamoDbAsyncClient.builder().overrideConfiguration(<your configuration>).build()</code> 。 |
| <code>cloudWatchClientConfig</code> | 使用以下所需配置来配置开发工具包 <code>CloudWatchAsyncClient</code> ： <code>CloudWatchAsyncClient.builder().overrideConfiguration(<your configuration>).build()</code> 。 |
| <code>regionName</code> | 使用首选区域配置开发工具包。这对所有开发工具包客户端均相同。例如 <code>KinesisAsyncClient.builder().region(Region.US_WEST_2).build()</code> 。 |

使用适用于 Java 的 Amazon SDK 开发消费端

可使用 Amazon Kinesis Data Streams API 开发自定义消费端。本部分介绍如何将 Kinesis Data Streams API 与适用于 Java 的 Amazon SDK 配合使用。

⚠ Important

有关开发具有共享吞吐量的自定义 Kinesis Data Streams 消费端的方法，建议使用 Kinesis Client Library (KCL)。KCL 通过处理许多与分布式计算相关的复杂任务，帮助您使用和处理 Kinesis 数据流中的数据。有关更多信息，请参阅 [使用 Java 通过 KCL 开发消费端](#)。

主题

- [使用适用于 Java 的 Amazon SDK 开发具有共享吞吐量的消费端](#)
- [使用适用于 Java 的 Amazon SDK 开发具有增强扇出功能的消费端](#)
- [使用 Amazon Glue 架构注册表与数据交互](#)

使用适用于 Java 的 Amazon SDK 开发具有共享吞吐量的消费端

开发具有共享吞吐量的自定义 Kinesis Data Streams 消费端的方法之一是结合使用 Amazon Kinesis Data Streams API 和适用于 Java 的 Amazon SDK。本部分介绍如何将 Kinesis Data Streams API 与适用于 Java 的 Amazon SDK 配合使用。您可以使用其他不同的编程语言调用 Kinesis Data Streams API。有关所有可用 Amazon 开发工具包的更多信息，请参阅[开始使用亚马逊云科技开发](#)。

本部分的 Java 示例代码演示如何执行基本的 Kinesis Data Streams API 操作，并按照操作类型从逻辑上进行划分。这些示例并非可直接用于生产的代码。它们不会检查所有可能的异常，或者不会考虑到所有可能的安全或性能问题。

主题

- [从流中获取数据](#)
- [使用分片迭代器](#)
- [使用 GetRecords](#)
- [适应重新分片](#)

从流中获取数据

Kinesis Data Streams API 包括 `getShardIterator` 和 `getRecords` 方法，您可以调用此类方法从数据流中检索记录。这是拉取模型，您的代码可以直接从数据流的分片中抽取数据记录。

⚠ Important

我们建议您使用由 KCL 提供的记录处理器支持功能，以从数据流中检索记录。这是推送模型，您可以通过实现代码来处理数据。KCL 将从数据流中获取数据记录并将数据记录传送给您的应用程序代码。此外，KCL 还提供失效转移、恢复和负载均衡功能。有关更多信息，请参阅 [Developing Custom Consumers with Shared Throughput Using KCL](#)。

但是，在某些情况下，您可能会更倾向于使用 Kinesis Data Streams API。例如，在实施自定义工具以监控或调试数据流时。

⚠ Important

Kinesis Data Streams 支持更改数据流的数据记录保留期。有关更多信息，请参阅 [更改数据留存期](#)。

使用分片迭代器

可从流中按分片检索记录。对于每个分片以及您从分片中检索的每批记录，您必须获取分片迭代器。可在 `getRecordsRequest` 对象中使用分片迭代器来指定要从中检索记录的分片。与分片迭代器关联的类型决定了应在分片中检索记录的起点（有关更多信息，请参阅此部分中后面的内容）。您必须先检索分片，然后才能使用分片迭代器。有关更多信息，请参阅 [列出分片](#)。

使用 `getShardIterator` 方法获取初始分片迭代器。使用 `getNextShardIterator` 对象（由 `getRecordsResult` 方法返回）的 `getRecords` 方法为其他记录批次获取分片迭代器。分片迭代器的有效时间为 5 分钟。如果使用有效期内的分片迭代器，则将获得一个新的迭代器。每个分片迭代器在 5 分钟内一直有效，即使使用过也是如此。

要获取初始分片迭代器，请实例化 `GetShardIteratorRequest` 并将其传递给 `getShardIterator` 方法。要配置请求，请指定流和分片 ID。有关如何获取 Amazon 账户中的流的信息，请参阅 [列出流](#)。有关如何获取流中分片的信息，请参阅 [列出分片](#)。

```
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();
```

此示例代码在获取初始分片迭代器时将 TRIM_HORIZON 指定为迭代器类型。此迭代器类型意味着记录应从添加到分片的第一个记录而不是从最近添加的记录（也称为顶端）开始返回。以下是可能的迭代器类型：

- AT_SEQUENCE_NUMBER
- AFTER_SEQUENCE_NUMBER
- AT_TIMESTAMP
- TRIM_HORIZON
- LATEST

有关更多信息，请参阅 [ShardIteratorType](#)。

部分迭代器类型除了需要指定类型之外，还需要指定序列号；例如：

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");  
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

使用 `getRecords` 获取记录之后，可通过调用记录的 `getSequenceNumber` 方法来获取记录的序列号。

```
record.getSequenceNumber()
```

此外，将记录添加到数据流的代码可通过对 `getSequenceNumber` 的结果调用 `putRecord` 获取已添加记录的序列号。

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

您可使用序列号确保记录的顺序严格递增。有关更多信息，请参阅 [PutRecord 示例](#) 中的代码示例。

使用 GetRecords

获取分片迭代器之后，请实例化 `GetRecordsRequest` 对象。使用 `setShardIterator` 方法为请求指定迭代器。

(可选) 您还可使用 `setLimit` 方法设置要检索的记录的数量。`getRecords` 返回的记录数量始终等于或小于此限制。如果您未指定此限制，`getRecords` 将返回已检索记录的 10MB。以下示例代码将此限制设置为 25 个记录。

如果未返回任何记录，则意味着此分片中当前没有分片迭代器引用的序列号对应的可用数据记录。在这种情况下，您的应用程序应等待流的数据来源所需的时间。然后尝试使用对 `getRecords` 的上一调用返回的分片迭代器再次从分片获取数据。

将 `getRecordsRequest` 传递给 `getRecords` 方法并捕获返回的值作为 `getRecordsResult` 对象。要获取数据记录，请对 `getRecords` 对象调用 `getRecordsResult` 方法。

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(25);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

要准备对 `getRecords` 的另一次调用，请通过 `getRecordsResult` 获取下一分片迭代器。

```
shardIterator = getRecordsResult.getNextShardIterator();
```

为获得最佳效果，请在对 `getRecords` 的各次调用之间停止至少 1 秒 (1000 毫秒) 以免超出 `getRecords` 频率限制。

```
try {
    Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

通常，您应循环调用 `getRecords`，甚至当您在测试方案中检索单一记录时也是如此。对 `getRecords` 的单一调用可能返回空的记录列表，即使分片包含更多具有之后的序列号的记录也是如此。出现此情况时，将返回 `NextShardIterator`，同时空记录列表将引用分片中之后的序列号，并且后续的 `getRecords` 调用最终将返回记录。以下示例演示循环的使用。

示例：getRecords

以下代码示例反映了此节中的 getRecords 顶端，包括循环发出调用。

```
// 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();
}
}
```

如果您使用 Kinesis Client Library，则可能在返回数据之前发出多次调用。此行为是设计使然，不代表 KCL 或您的数据存在问题。

适应重新分片

如果 `getRecordsResult.getNextShardIterator` 返回 `null`，则表示发生了涉及此分片的分片拆分或合并。此分片现在处于 CLOSED 状态，并且您已从其中读取了所有可用的数据记录。

在这种情况下，您可以使用 `getRecordsResult.childShards` 来了解正在处理的分片中由拆分或合并创建的新子分片。有关更多信息，请参阅 [ChildShard](#)。

在拆分中，两个新分片的 `parentShardId` 都与您之前处理的分片的分片 ID 相同。这两个分片的 `adjacentParentShardId` 值为 `null`。

在合并中，合并创建的一个新分片的 `parentShardId` 等于父分片之一的分片 ID，并且 `adjacentParentShardId` 等于另一父分片的分片 ID。您的应用程序已读取这些分片之一中的所有数据。这是 `getRecordsResult.getNextShardIterator` 返回 `null` 的分片。如果数据顺序对于您的应用程序很重要，则应确保它在读取合并创建的子分片中的任何新数据之前，还读取另一父分片中的所有数据。

如果您使用多个处理器从流检索数据（假定一个分片一个处理器），并且出现分片拆分或合并时，您应增加或减少处理器数量以适应分片数量的变化。

有关重新分片的更多信息，包括有关分片状态（如 `CLOSED`）的讨论，请参阅 [对流进行重新分片](#)。

使用适用于 Java 的 Amazon SDK 开发具有增强扇出功能的消费端

增强型扇出是一种 Amazon Kinesis Data Streams 功能，支持消费端从数据流中接收记录，其中每分片每秒专用吞吐量高达 2MB 数据。使用增强型扇出功能的消费端不必与接收流中数据的其他消费端争夺。有关更多信息，请参阅 [开发具有专用吞吐量的增强扇出型消费端](#)。

可以使用 API 操作构建在 Kinesis Data Streams 中使用增强型扇出功能的消费端。

使用 Kinesis Data Streams API 注册采用增强型扇出功能的消费端

1. 调用 [RegisterStreamConsumer](#) 将应用程序注册为使用增强型扇出功能的消费端。Kinesis Data Streams 为消费端生成一个 Amazon 资源名称 (ARN) 并在响应中返回此名称。
2. 要开始侦听特定分片，请将调用中的消费端 ARN 传递给 [SubscribeToShard](#)。之后，Kinesis Data Streams 会通过 HTTP/2 连接将分片中的记录以 [SubscribeToShardEvent](#) 类型的事件形式推送给您。此连接将保持打开状态长达 5 分钟。如果要在通过调用 [SubscribeToShard](#) 返回的 `future` 正常或异常完成之后继续接收分片中的记录，请再次调用 [SubscribeToShard](#)。

 Note

到达当前分片的末尾时，[SubscribeToShard](#) API 还会返回当前分片的子分片列表。

3. 要取消注册使用增强型扇出功能的消费端，请调用 [DeregisterStreamConsumer](#)。

以下代码是一个示例，演示如何为消费端订阅分片、定期续订订阅以及处理事件。

```
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);
 * }
 * }
```

如果 `event.ContinuationSequenceNumber` 返回 `null`，则表示发生了涉及此分片的分片拆分或合并。此分片现在处于 `CLOSED` 状态，并且您已从其中读取了所有可用的数据记录。在这种情况下，按照上文示例所述，您可以使用 `event.childShards` 来了解正在处理的分片中由拆分或合并创建的新子分片。有关更多信息，请参阅 [ChildShard](#)。

使用 Amazon Glue 架构注册表与数据交互

您可以将 Kinesis 数据流与 Amazon Glue 架构注册表进行集成。Amazon Glue 架构注册表能帮助您集中发现、控制和演变架构，同时确保注册架构持续验证生成的数据。架构定义了数据记录的结构和格式。架构是用于可靠数据发布、使用或存储的版本化规范。通过 Amazon Glue 架构注册表，您能够改善流媒体应用程序中的端到端数据质量和数据治理。有关更多信息，请参阅 [Amazon Glue Schema Registry](#)。使用 Amazon Java 开发工具包中提供的 `GetRecords` Kinesis Data Streams API 可以设置此集成。

有关如何使用 `GetRecords` Kinesis Data Streams API 设置 Kinesis Data Streams 与 Schema 注册表集成的详细说明，请参阅 [Use Case: Integrating Amazon Kinesis Data Streams with the Amazon Glue Schema Registry](#) 中的“Interacting with Data Using the Kinesis Data Streams APIs”部分。

使用以下方法培养消费者 Amazon Lambda

您可以使用Amazon Lambda函数来处理数据流中的记录。Amazon Lambda是一项计算服务，允许您在不预置或管理服务器的情况下运行代码。它只在需要时才执行您的代码并自动缩放，从每天几个请求到每秒数千个请求。您只需按使用的计算时间付费。代码不运行时不会产生任何费用。使用Amazon Lambda，您几乎可以为任何类型的应用程序或后端服务运行代码，所有这些都无需管理。它在可用性高的计算基础设施上运行您的代码，执行计算资源的所有管理工作，其中包括服务器和操作系统维护、容量预置和自动扩展、代码监控和记录。有关更多信息，请参阅[Amazon Lambda与 Amazon Kinesis 搭配使用](#)。

有关问题排查信息，请参阅[为什么 Kinesis Data Streams 触发器无法调用我的 Lambda 函数？](#)

使用适用于 Apache Flink 的亚马逊托管服务来开发消费端

您可以使用适用于 Apache Flink 的亚马逊托管服务的应用程序，以使用 SQL、Java 或 Scala 来处理和分析 Kinesis 流中的数据。适用于 Apache Flink 应用程序的托管服务可以使用参考源来丰富数据，聚合一段时间内的数据，或者使用机器学习来查找数据异常。然后，您可以将分析结果写入到另一个 Kinesis 流、Firehose 传输流或 Lambda 函数。有关更多信息，请参阅[Managed Service for Apache Flink Developer Guide for SQL Applications](#) 或 [Managed Service for Apache Flink Developer Guide for Flink Applications](#)。

使用 Amazon Data Firehose 开发消费端

您可以使用 Firehose 读取和处理 Kinesis 流中的记录。Firehose 是一项完全托管的服务，用于向亚马逊 S3、亚马逊 Redshift、OpenSearch 亚马逊服务和 Splunk 等目的地提供实时流数据。Firehose 还支持由受支持的第三方服务提供商（包括 Datadog、MongoDB 和 New Relic）拥有的任何一个或多个自定义 HTTP 端点。您还可以配置 Firehose 以转换您的数据记录和记录格式，然后再将您的数据传输到其目标。有关更多信息，请参阅[Writing to Firehose Using Kinesis Data Streams](#)。

使用其他 Amazon 服务从 Kinesis Data Streams 读取数据

以下 Amazon 服务可以直接与 Amazon Kinesis Data Streams 集成，以便从 Kinesis 数据流读取数据。查看您感兴趣的每项服务的信息，并参考所提供的参考资料。

主题

- [使用 Amazon EMR 从 Kinesis Data Streams 读取数据](#)

- [使用 Amazon EventBridge Pipes 从 Kinesis Data Streams 读取数据](#)
- [使用 Amazon Glue 从 Kinesis Data Streams 读取数据](#)
- [使用 Amazon Redshift 从 Kinesis Data Streams 读取数据](#)

使用 Amazon EMR 从 Kinesis Data Streams 读取数据

Amazon EMR 集群可以使用 Hadoop 生态系统中的熟悉工具（如 Hive、Pig、MapReduce、Hadoop Streaming API 和 Cascading）直接读取和处理 Kinesis 流。您还可以将 Kinesis Data Streams 中的实时数据与正在运行的集群中 Amazon S3、Amazon DynamoDB 和 HDFS 上的现有数据进行连接。您可以直接将 Amazon EMR 中的数据加载到 Amazon S3 或 DynamoDB 来进行后处理。

有关更多信息，请参阅《Amazon EMR Release Guide》中的 [Amazon Kinesis](#)。

使用 Amazon EventBridge Pipes 从 Kinesis Data Streams 读取数据

Amazon EventBridge Pipes 支持将 Kinesis 数据流作为源。Amazon EventBridge Pipes 通过可选的转换、筛选和扩充步骤，帮助您在事件产生器和消费端之间创建点对点集成。您可以使用 EventBridge Pipes 接收来自 Kinesis 数据流的记录，也可以选择筛选或扩充这些记录，然后再将它们发送到包括 Kinesis Data Streams 在内的可用目标之一进行处理。

有关更多信息，请参阅《Amazon EventBridge Release Guide》中的 [Amazon Kinesis stream as a source](#)。

使用 Amazon Glue 从 Kinesis Data Streams 读取数据

通过使用 Amazon Glue 流式处理 ETL，您可以创建流式处理的提取、转换、加载（ETL）任务，其持续运行并使用来自 Amazon Kinesis Data Streams 的数据。这些任务会清理并转换数据，然后将结果加载到 Amazon S3 数据湖或 JDBC 数据存储中。

有关更多信息，请参阅《Amazon Glue Release Guide》中的 [Streaming ETL jobs in Amazon Glue](#)。

使用 Amazon Redshift 从 Kinesis Data Streams 读取数据

Amazon Redshift 支持来自 Amazon Kinesis Data Streams 的串流摄取。Amazon Redshift 串流摄取功能以低延迟、高速度的方式将流数据从 Amazon Kinesis Data Streams 摄取到 Amazon Redshift 实体化视图中。使用 Amazon Redshift 流式摄取时，在将流式数据摄取到 Redshift 之前，无需在 Amazon S3 中暂存数据。

有关更多信息，请参阅《Amazon Redshift 数据库开发人员指南》中的[串流摄取](#)。

使用第三方集成从 Kinesis Data Streams 中读取

您可以使用以下与 Kinesis Data Streams 集成的任一第三方选项，从 Amazon Kinesis Data Streams 数据流中读取数据。选择您想详细了解的选项，然后查找资源和相关文档的链接。

主题

- [Apache Flink](#)
- [Adobe Experience Platform](#)
- [Apache Druid](#)
- [Apache Spark](#)
- [Databricks](#)
- [Kafka Confluent Platform](#)
- [Kinesumer](#)
- [Talend](#)

Apache Flink

Apache Flink 是一种开源框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。有关利用 Apache Flink 使用 Kinesis Data Streams 的更多信息，请参阅 [Amazon Kinesis Data Streams Connector](#)。

Adobe Experience Platform

Adobe Experience Platform 让组织能够集中管理和标准化来自任何系统的客户数据。然后，该平台应用数据科学和机器学习，显著改进丰富的个性化体验的设计和交付。有关利用 Adobe Experience Platform 使用 Kinesis 数据流的更多信息，请参阅 [Amazon Kinesis 连接器](#)。

Apache Druid

Druid 是一种具备实时分析功能的高性能数据库，可在负载下大规模提供对流数据和批量数据的亚秒级查询。有关利用 Apache Druid 摄取 Kinesis Data Streams 数据的更多信息，请参阅 [Amazon Kinesis ingestion](#)。

Apache Spark

Apache Spark 是用于大规模数据处理的统一分析引擎。其提供了 Java、Scala、Python 和 R 这几种语言的高级别 API，以及支持常规执行图的优化引擎。您可以使用 Apache Spark 来构建流处理应用程序，这些应用程序使用您的 Kinesis 数据流中的数据。

要通过 Apache Spark Structured Streaming 使用 Kinesis 数据流，请使用 Amazon Kinesis Data Streams [连接器](#)。此连接器支持增强扇出型使用，可为您的应用程序提供高达每分片 2 MB/s 的专用数据读取吞吐量。有关更多信息，请参阅[开发具有专用吞吐量的自定义消费端（增强型扇出功能）](#)。

要通过 Spark Streaming 使用 Kinesis Data Streams，请参阅[Spark Streaming + Kinesis Integration](#)。

Databricks

Databricks 是一种基于云的平台，可为数据工程、数据科学和机器学习提供协作环境。有关通过 Databricks 使用 Kinesis 数据流的更多信息，请参阅[Connect to Amazon Kinesis](#)。

Kafka Confluent Platform

Confluent Platform 是基于 Kafka 构建的平台，提供了其他特性和功能，可帮助企业构建并管理实时数据管道和流式应用程序。有关利用 Confluent Platform 使用 Kinesis 数据流的更多信息，请参阅[Amazon Kinesis Source Connector for Confluent Platform](#)。

Kinesumer

Kinesumer 是一个 Go 客户端，用于为 Kinesis 数据流实现客户端分布式消费端组客户端。有关更多信息，请参阅[Kinesumer Github repository](#)。

Talend

Talend 是一种数据集成和管理软件，支持用户以可扩展且高效的方式收集、转换和连接来自各种来源的数据。有关利用 Talend 使用 Kinesis 数据流的更多信息，请参阅[Connect talend to an Amazon Kinesis stream](#)。

Kinesis Data Streams 消费端问题排查

以下主题提供了 Amazon Kinesis Data Streams 消费端常见问题的解决方案：

- [LeaseManagementConfig 构造函数编译错误](#)

- [使用 Kinesis Client Library 时会跳过某些 Kinesis Data Streams 记录](#)
- [属于同一分片的记录通过不同的记录处理器同时处理](#)
- [消费端应用程序的读取速率比预期的慢](#)
- [即使流中有数据，GetRecords 仍然返回空记录阵列](#)
- [分片迭代器意外过期](#)
- [消费端记录处理滞后](#)
- [未授权的 KMS 密钥权限错误](#)
- [DynamodBException：在更新表达式中提供的文档路径对于更新无效](#)
- [排查消费端的其他常见问题](#)

LeaseManagementConfig 构造函数编译错误

升级到 Kinesis Client Library (KCL) 3.x 或更高版本时，可能会遇到与 LeaseManagementConfig 构造函数相关的编译错误。如果直接创建 LeaseManagementConfig 对象来设置配置，而不是在 KCL 3.x 或更高版本中使用 ConfigsBuilder，则在编译 KCL 应用程序代码时可能会看到以下错误消息。

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

KCL 3.x 或更高版本要求在 tableName 参数之后再添加一个参数，即 applicationName (type: String)。

- 之前：leaseManagementConfig = new LeaseManagementConfig(tableName, dynamoDBClient, kinesisClient, streamName, workerIdentifier)
- 之后：leaseManagementConfig = new LeaseManagementConfig(tableName, applicationName, dynamoDBClient, kinesisClient, streamName, workerIdentifier)

在 KCL 3.x 及更高版本中，我们建议不要直接创建 LeaseManagementConfig 对象，而是使用 ConfigsBuilder 设置配置。ConfigsBuilder 提供了更灵活且更易于维护的方式配置 KCL 应用程序。

下面是使用 ConfigsBuilder 设置 KCL 配置的示例。

```
ConfigsBuilder configsBuilder = new ConfigsBuilder()  
    .streamName(  
    .applicationName(  
    .leaseManagementConfig(  
        .tableName("my-stream")  
        .applicationName("my-app")  
        .dynamoDBClient(dynamoDBClient)  
        .kinesisClient(kinesisClient)  
        .streamName("my-stream")  
        .workerIdentifier("my-worker")  
    )  
);
```

```
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()  
);
```

使用 Kinesis Client Library 时会跳过某些 Kinesis Data Streams 记录

跳过记录的最常见原因是未处理从 `processRecords` 引发的异常。Kinesis Client Library (KCL) 依靠 `processRecords` 代码来处理由数据记录引起的任何异常。`processRecords` 引发的任何异常都会被 KCL 吸收。为避免因为反复出现的故障无休止地进行重试，KCL 不会重新发送在发生异常时处理的那批记录。然后，KCL 会在不重新启动记录处理器的情况下对下一批数据记录调用 `processRecords`。这有效地导致消费端应用程序观察到跳过的记录。要防止跳过记录，请适当处理 `processRecords` 中的所有异常。

属于同一分片的记录通过不同的记录处理器同时处理

对于任何正在运行的 Kinesis Client Library (KCL) 应用程序，一个分片只有一个所有者。但是，多个记录处理器可以临时处理同一分片。如果工作程序实例网络连接中断，KCL 会假定此无法联系的工作程序不再处理记录，并会在失效转移时间到期后指示其他工作程序来接管其工作。在一段很短的时间内，新的记录处理器和来自无法联系的工作程序的记录处理器可能都会处理来自同一个分片的数据。

根据您的应用程序设置适当的失效转移时间。对于低延迟应用程序，默认值 10 秒可以代表您可以等待的最长时间。但是，在一些情况下，如您预计会有一些连接问题（如跨地理区域打电话）并且连接可能较频繁地中断时，这个数字设置可能就过低了。

您的应用程序应预料到并处理这种情况，尤其是因为网络连接通常会恢复到之前无法访问的工作程序。如果一个记录处理器让另一个记录处理器接手其分片，它必须处理以下两种情况才能顺利执行关闭：

1. 对 `processRecords` 的当前调用完成后，KCL 会对记录处理器调用 `shutdown` 方法，并说明关闭原因为“ZOMBIE”。您的记录处理器应视情况清除所有资源然后退出。
2. 当您尝试从“zombie”工作程序执行检查点操作时，KCL 会引发 `ShutdownException`。收到此异常后，您的代码应完全退出当前方法。

有关更多信息，请参阅 [处理重复记录](#)。

消费端应用程序的读取速率比预期的慢

读取吞吐量低于预期的最常见原因如下：

1. 多个消费端应用程序的总读取量超过每个分片的限制。有关更多信息，请参阅 [限额和限制](#)。在此情况下，您可以增加 Kinesis 数据流中的分片数量。
2. 指定每个调用的 `GetRecords` 最大数量的 [限制](#) 可能配置为较低值。如果您正在使用 KCL，您可能已经使用一个较低的 `maxRecords` 属性值配置了工作程序。一般来说，我们推荐对此属性使用系统默认值。
3. 出于很多可能的原因，您的 `processRecords` 调用内的逻辑花费的时间可能超过预期；该逻辑可能是 CPU 使用率高、I/O 阻止或同步出现瓶颈。要测试是否如此，请对空的记录处理器进行测试运行并比较读取吞吐量。有关如何跟踪传入数据的信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

如果您只有一个消费端应用程序，那么读取速率比放入速率至少高两倍始终是可能的。这是因为每秒最多可以写入 1000 条记录，最大总数据写入速率为每秒 1MB（包括分区键）。每个开放分片每秒最多可读取 5 个事务，最大总数据读取速率为每秒 2 MB。请注意，每次读取（`GetRecords` 调用）都将获取一批记录。`GetRecords` 返回的数据的大小因分片的使用率而异。`GetRecords` 可返回的数据的最大大小为 10MB。如果某个调用返回了该限制，在接下来 5 分钟内进行的后续调用将引发 `ProvisionedThroughputExceededException`。

即使流中有数据，`GetRecords` 仍然返回空记录阵列

使用或获取记录是一种拉取模型。开发人员应该在不会回退的连续循环中调用 [GetRecords](#)。每个 `GetRecords` 调用还将返回一个 `ShardIterator` 值，该值必须在循环的下一个迭代中使用。

`GetRecords` 操作不会阻止。相反，它将立即返回一些相关数据记录或一个空的 `Records` 元素。在两种情况下，将返回空的 `Records` 元素：

1. 目前分片中没有更多数据。

2. ShardIterator 指向的分片部分附近没有数据。

后一种情况很微妙，但却是避免在检索数据时搜寻时间无止境（延迟）的一种必要的设计折衷。因此，流使用应用程序应循环并调用 GetRecords，并且理所当然地处理空记录。

在生产场景中，仅当 NextShardIterator 值为 NULL 时，才应退出连续循环。当 NextShardIterator 为 NULL 时，这意味着当前分片已关闭，ShardIterator 值的指向应越过最后一条记录。如果使用应用程序从不调用 SplitShard 或 MergeShards，分片将保持打开状态，并且对 GetRecords 的调用从不返回为 NextShardIterator 的 NULL 值。

如果您使用 Kinesis Client Library (KCL)，则会自动为您抽取以上使用模式。这包括自动处理一组动态变化的分片。在 KCL 中，开发人员只需提供处理传入记录的逻辑。这是可能的，因为该库会为您连续调用 GetRecords。

分片迭代器意外过期

每个 GetRecords 请求返回新的分片迭代器（作为 NextShardIterator），然后您可以将其用于下一个 GetRecords 请求（作为 ShardIterator）。此分片迭代器一般来说不会在您使用前过期。不过，您可能会发现，由于您超过 5 分钟没有调用 GetRecords，或者您执行了消费端应用程序的重新启动操作，该分片迭代器会过期。

如果分片迭代器在您还没能使用之前很快过期，这可能表示 Kinesis 使用的 DynamoDB 表没有足够的容量存储租约数据。如果您的分片数量很多，则很可能发生这种情况。要解决此问题，请增加分配给分片表的写入容量。有关更多信息，请参阅 [使用租约表跟踪 KCL 消费端应用程序处理的分片](#)。

消费端记录处理滞后

对于大多数使用案例，消费端应用程序从流中读取最新数据。在特定情况下，消费端读取可能会滞后，而您可能并不希望出现这种情况。在确定您的消费端读取滞后多久后，请查看消费端滞后的最常见原因。

从 GetRecords.IteratorAgeMilliseconds 指标开始，该指标跟踪流中所有分片和消费端的读取位置。请注意，如果某个迭代器的寿命超过了保留期的 50%（默认值为 24 小时，最长可配置为 365 天），则存在记录过期造成数据丢失的风险。一种快速的权宜之计是增加保留期。这会在您进一步对问题进行故障排除时防止重要数据丢失。有关更多信息，请参阅 [使用 Amazon CloudWatch 监控 Amazon Kinesis Data Streams 服务](#)。接下来，使用 Kinesis Client Library (KCL) 发出的自定义 CloudWatch 指标 MillisBehindLatest，确定消费端应用程序从每个分片读取的滞后时间长度。有关更多信息，请参阅 [使用 Amazon CloudWatch 监控 Kinesis Client Library](#)。

下面是消费端滞后的最常见原因：

- `GetRecords.IteratorAgeMilliseconds` 或 `MillisBehindLatest` 突然发生大幅提升，通常表明临时性问题，例如下游应用程序的 API 操作失败。如果任何一个指标持续指示此行为，则调查这些突然增长的原因。
- 这些指标的逐步增大表明，由于处理记录的速度不够快，消费端无法与流保持同步。此行为最常见的根本原因是没有足够的物理资源，或者记录处理逻辑没有随着流吞吐量的增大而进行扩展。您可以查看 KCL 发出的与 `processTask` 操作关联的其他自定义 CloudWatch 指标，包括 `RecordProcessor.processRecords.Time`、`Success` 和 `RecordsProcessed`，从而验证此行为。
 - 如果您发现与吞吐量上升相关的 `processRecords.Time` 指标发生增长，则应该分析记录处理逻辑，以确定为什么逻辑没有随吞吐量的增长而扩展。
 - 如果您发现与吞吐量上升无关的 `processRecords.Time` 值发生增长，请检查您是否在关键路径中执行了任何阻塞性调用，这通常会导致记录处理速度下降。替代方法是通过增加分片数来提高并行度。最后，请确认需求高峰期间在底层处理节点上，您有足够的物理资源（内存、CPU 使用率等）。

未授权的 KMS 密钥权限错误

当消费端应用程序从加密流读取但没有 Amazon KMS 密钥的权限时，会发生此错误。要为应用程序分配权限以访问 KMS 密钥，请参阅 [Using Key Policies in Amazon KMS](#) 和 [Using IAM Policies with Amazon KMS](#)。

DynamodBException：在更新表达式中提供的文档路径对于更新无效

在 KCL 3.x 中使用 2.27.19 - 2.27.23 版的适用于 Java 的 Amazon SDK 时，可能会遇到以下 DynamoDB 异常：

“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)”

出现此种错误的原因是，适用于 Java 的 Amazon SDK 中存在的已知问题影响到了 KCL 3.x 管理的 DynamoDB 元数据表。该问题是在 2.27.19 版本中引入的，2.27.23 之前的所有版本都受其影响。这一问题已在 2.27.24 版的适用于 Java 的 Amazon SDK 中得到解决。为获得最佳性能和稳定性，我们建议升级到 2.28.0 或更高版本。

排查消费端的其他常见问题

- [为什么 Kinesis Data Streams 触发器无法调用我的 Lambda 函数？](#)

- [如何检测并排查 Kinesis Data Streams 中的 ReadProvisionedThroughputExceeded 异常？](#)
- [为什么我会遇到 Kinesis Data Streams 延迟高的问题？](#)
- [为什么我的 Kinesis 数据流会返回一个 500 内部服务器错误？](#)
- [如何为 Kinesis Data Streams 排查受阻或停滞的 KCL 应用程序？](#)
- [能否将不同的 Amazon Kinesis Client Library 应用程序与同一 Amazon DynamoDB 表一起使用？](#)

优化 Amazon Kinesis Data Streams 消费端

根据您看到的特定行为，您可以进一步优化 Amazon Kinesis Data Streams 消费端。

请查看以下主题以确定解决方案。

主题

- [改进低延迟处理](#)
- [使用 Amazon Kinesis Amazon Lambda 制作器库处理序列化数据](#)
- [使用重新分片、扩展和并行处理更改分片数量](#)
- [处理重复记录](#)
- [处理启动、关闭和节流](#)

改进低延迟处理

传播延迟定义为从记录写入流的那一刻起到消费者应用程序读取该记录的 end-to-end 延迟。此延迟会因各种因素而变化，但主要受消费端应用程序的轮询间隔影响。

对于大多数应用程序，我们建议针对每个应用程序每秒轮询每个分片一次。通过该操作，您能够具有并行处理流的多个消费端应用程序，而不会达到 Amazon Kinesis Data Streams 每秒 5 次 GetRecords 调用的限制。此外，若要处理大批量的数据，降低您的应用程序中的网络和其他下游延迟时往往更高效。

KCL 的默认值设置为遵循每 1 秒轮询一次的最佳实践。此默认值导致了通常少于 1 秒的平均传播延迟。

Kinesis Data Streams 记录一经写入，即可立即读取。有一些需要利用此延迟并且在流中的数据可用时立即需要使用它的使用案例。您可通过覆盖 KCL 默认设置来更频繁地进行轮询，从而显著降低传播延迟，如以下示例所示。

Java KCL 配置代码：

```
kinesisClientLibConfiguration = new
    KinesisClientLibConfiguration(applicationName,
        streamName,
        credentialsProvider,
        workerId).withInitialPositionInStream(initialPositionInStream).withIdleTimeBetweenReadsInMillis(250)
```

Python 和 Ruby KCL 的属性文件设置：

```
idleTimeBetweenReadsInMillis = 250
```

 Note

由于 Kinesis Data Streams 每个分片具有每秒 5 次 GetRecords 调用的限制，因此将 idleTimeBetweenReadsInMillis 属性设置为少于 200ms 可能导致您的应用程序观察到 ProvisionedThroughputExceededException 异常。如果这类异常过多，则可能导致指数退避，并因此导致处理过程中出现重大的意外延迟。如果您将此属性设置为 200 ms 或更大并且具有多个正在处理的应用程序，则会遇到类似的限制。

使用 Amazon Kinesis Amazon Lambda 制作器库处理序列化数据

[Amazon Kinesis Producer Library](#) (KPL) 将较小的用户格式化记录聚合为较大的记录 (最大为 1 MB)，以更好地利用 Amazon Kinesis Data Streams 吞吐量。虽然 Java 版 KCL 支持解聚这些记录，但在用作流的使用者时，您需要使用 Amazon Lambda 特殊模块来取消聚合记录。您可以从[适用 GitHub 于 Lambda 的 Amazon Kinesis Producer 库解聚模块](#)中获取必要的项目代码和说明。Amazon 该项目中的组件使您能够在 Java Amazon Lambda、Node.js 和 Python 中处理 KPL 序列化数据。这些组件可用作[多语言 KCL 应用程序](#)的一部分。

使用重新分片、扩展和并行处理更改分片数量

利用重新分片，您可以增加或减少流中的分片的数量，以便适应流过流的数据的速率的变化。重新分片通常是由监控分片数据处理指标的管理应用程序执行的。尽管 KCL 本身不启动重新分片操作，但它能够适应由于重新分片而生成的分片的数量的变化。

如[使用租约表跟踪 KCL 消费端应用程序处理的分片](#)中所述，KCL 使用 Amazon DynamoDB 表跟踪流中的分片。当由于重新分片而创建新分片时，KCL 会发现新分片并在该表中填充新行。工作程序将自动发现新的分片并创建处理器以处理来自分片的数据。KCL 还将跨所有可用工作程序和记录处理器分配流中的分片。

KCL 将确保优先处理在重新分片之前已存在于分片中的任何数据。在处理该数据后，新分片中的数据将发送到记录处理器。这样，KCL 便能保留为特定分区键将数据记录添加到流的顺序。

示例：重新分片、扩展和并行处理

以下示例将演示 KCL 如何帮助您处理扩展和重新分片：

- 例如，如果您的应用程序在一个 EC2 实例上运行，并且正在处理一个包含四个分片的 Kinesis 数据流。这 1 个实例包含 1 个 KCL 工作程序和 4 个记录处理器（每个分片有 1 个记录处理器）。这 4 个记录处理器在同一进程内并行运行。
- 接下来，如果您扩展应用程序以使用其他实例，您将有 2 个处理 1 个包含 4 个分片的流的实例。当 KCL 工作程序在第二个实例上启动时，它会与第一个实例进行负载均衡，以便让每个实例现在处理两个分片。
- 如果您随后决定将 4 个分片拆分为 5 个分片。KCL 会再次跨实例协调处理：一个实例处理 3 个分片，另一个实例处理 2 个分片。类似协调在合并分片时发生。

通常，在使用 KCL 时，您应确保实例的数量不超过分片的数量（故障待机除外）。每个分片正好由一个 KCL 工作程序处理，并且正好有一个对应的记录处理器，因此您永远不需要多个实例来处理一个分片。但是，一个工作程序可处理任意数量的分片，因此分片的数量超过实例的数量没有关系。

要扩展您的应用程序中的处理，您应测试以下方法的组合：

- 增加实例大小（因为所有记录处理器都在进程内并行运行）
- 增加实例的数量，最多为开放分片的最大数量（因为分片可以单独处理）
- 增加分片的数量（这会提高并行机制的级别）

请注意，您可使用自动扩缩，从而基于适当的指标自动扩展您的实例。有关更多信息，请参阅 [Amazon Auto Scaling 用户指南](#)。

当重新分片增加流中的分片数量时，记录处理器数量的相应增加会增加托管这些分片的 EC2 实例的负载。如果实例为 Auto Scaling 组的一部分，并且负载增加得足够多，则 Auto Scaling 组会添加更多实例来处理增加的负载。您应在启动时配置用来启动 Amazon Kinesis Data Streams 应用程序的实例，以便让其他工作程序和记录处理器在新实例上立即激活。

有关重新分片的更多信息，请参阅[对流进行重新分片](#)。

处理重复记录

有两个主要原因可能导致多次向您的 Amazon Kinesis Data Streams 应用程序传送记录：创建器重试和消费端重试。您的应用程序必须预计并适当地应对多次处理单个记录的问题。

产生器重试

假设某个创建器已经对 PutRecord 进行调用但仍在从 Amazon Kinesis Data Streams 接收确认前遇到了网络相关的超时。创建器无法确定记录是否已传输到 Kinesis Data Streams。假定每个记录对应用程序都很重要，创建者应该已被写入以重试对相同数据的调用。如果对相同数据的两次 PutRecord 调用已成功提交到 Kinesis Data Streams，则会有两个 Kinesis Data Streams 记录。尽管这两个记录具有相同的数据，但它们各具有唯一的序号。需要严格保证的应用程序应在记录中嵌入一个主键，以便在随后的处理过程中删除重复项。请注意，由创建者重试产生的重复项的数量通常低于由消费端重试产生的重复项的数量。

 Note

如果您使用 Amazon SDKPutRecord，请在AmazonSDKs 和工具用户指南中了解 SDK [重试行为](#)。

消费端重试

消费端（数据处理应用程序）重试在记录处理器重新启动时发生。相同分片的记录处理器在以下情况下重新启动：

1. 工作程序意外终止
2. 已添加或删除工作程序实例
3. 已拆分或合并分片
4. 已部署应用程序

在所有这些情况下，-rec shards-to-worker-to ord-processor 映射会不断更新以进行负载平衡处理。已迁移到其他实例的分片处理器将从上一个检查点开始重新启动处理记录。这导致了重复的记录处理，如以下示例所示。有关负载均衡的更多信息，请参阅[使用重新分片、扩展和并行处理更改分片数量](#)。

示例：导致重新传送记录的消费端重试

在此示例中，您有一个持续从流中读取记录、将记录聚合到本地文件并将文件上传到 Amazon S3 的应用程序。为简便起见，假定只有 1 个分片和 1 个处理该分片的工作程序。考虑以下示例顺序的事件，假定上一个检查点位于记录编号 10000 处：

1. 工作程序从该分片中读取下一批记录，即从 10001 到 20000 的记录。
2. 工作程序随后将这批记录传递到关联的记录处理器。
3. 记录处理器聚合数据、创建 Amazon S3 文件并将文件成功上传到 Amazon S3。
4. 工作程序在新的检查点出现之前意外终止。
5. 应用程序、工作程序和记录处理器重新启动。
6. 工作程序现在开始从上次成功的检查点（在本案例中为 10001）开始读取。

因此，记录 10001-20000 使用了多次。

实现对消费端重试的弹性

尽管记录可被处理多次，但您的应用程序可能会带来副作用，就像记录只能处理一次一样（幂等处理）。有关此问题的解决方案因复杂性和准确性而异。如果最终数据的目标可以很好地处理重复，我们建议依靠最终目标来实现幂等处理。例如，使用 [OpenSearch](#)，您可以使用版本控制和唯一版本控制的组合 IDs 来防止重复处理。

在上一节的示例应用程序中，它持续从流中读取记录、将记录聚合到本地文件并将文件上传到 Amazon S3。如图所示，记录 10001 - 20000 使用了多次，从而生成了具有相同数据的多个 Amazon S3 文件。减少此示例中的重复的一种方法是确保步骤 3 使用了以下方案：

1. 记录处理器对每个 Amazon S3 文件使用了固定数量的记录，如 5000。
2. 文件名使用以下架构：Amazon S3 前缀、分片 ID 和 First-Sequence-Num。在本例中，它可以是类似于 sample-shard000001-10001 的形式。
3. 在上传 Amazon S3 文件后，通过指定 Last-Sequence-Num 进行检查点操作。在本例中，您将在记录编号 15000 处进行检查点操作。

利用此方案，即使记录被处理了多次，生成的 Amazon S3 文件也会具有相同的名称和数据。重试只会导致将相同的数据多次写入到相同的文件。

对于重新分片操作，分片中剩余的记录的数量可能少于您需要的固定数量。在本例中，您的 shutdown() 方法必须将文件刷新到 Amazon S3 并对上一个序列号进行检查点操作。以上方案也与重新分片操作兼容。

处理启动、关闭和节流

以下是要融入到您的 Amazon Kinesis Data Streams 应用程序设计中的部分其他注意事项。

主题

- [启动数据产生器和数据消费端](#)
- [关闭 Amazon Kinesis Data Streams 应用程序](#)
- [读取节流](#)

启动数据产生器和数据消费端

默认情况下，KCL 从流的提示处开始读取记录，并且是最新添加的记录。在此配置中，如果数据生成应用程序在任何接收记录的处理器运行前向流添加记录，那么记录处理器不会在启动后读取这些记录。

要更改记录处理器的行为以便它始终从流的开头处读取数据，请在属性文件中为您的 Amazon Kinesis Data Streams 应用程序设置以下值：

```
initialPositionInStream = TRIM_HORIZON
```

默认情况下，Amazon Kinesis Data Streams 会将所有数据存储 24 小时。它还支持长达 7 天的延长保留和长达 365 天的长期保留。此时间范围称为保留期。将开始位置设置为 TRIM_HORIZON 时，将根据保留期的定义，对流中最早的数据启动记录处理器。即使使用 TRIM_HORIZON 设置，如果记录处理器在超过了保留期很长一段时间之后启动，流中的一些数据将不再可用。因此，您应该始终让使用者应用程序从流中读取数据，并使用该 CloudWatch 指标 `GetRecords.IteratorAgeMilliseconds` 来监控应用程序是否跟上了传入数据的步伐。

在某些情况下，对于记录处理器来说，错过流中的前几个记录没有关系。例如，您可能会在直播中运行一些初始记录，以测试直播是否 end-to-end 按预期运行。在此初步验证之后，您随后会启动您的工作程序并开始将生产数据放入流中。

有关 TRIM_HORIZON 设置的更多信息，请参阅[使用分片迭代器](#)。

关闭 Amazon Kinesis Data Streams 应用程序

当您的 Amazon Kinesis Data Streams 应用程序完成其预期任务后，您应该通过终止其正在运行 EC2 的实例来将其关闭。您可使用 [Amazon Web Services 管理控制台](#) 或 [Amazon CLI](#) 来终止实例。

关闭 Amazon Kinesis Data Streams 后，您应删除 KCL 用于跟踪应用程序状态的 Amazon DynamoDB 表。

读取节流

流的吞吐量在分片级别进行配置。每个分片的读取吞吐量最高为每秒 5 个事务，最大总数据读取速率 为每秒 2 MB。如果某个应用程序（或对相同的流执行操作的一组应用程序）尝试以较快的速率从分片 中获取数据，Kinesis Data Streams 将限制对应的 Get 操作。

在 Amazon Kinesis Data Streams 应用程序中，如果记录处理器处理数据的速度超过限制（例如在失 效转移的情况下），则会出现节流。由于 KCL 管理应用程序与 Kinesis Data Streams 之间的交互，因此 节流异常会在 KCL 代码中出现，而不会在应用程序代码中出现。然而，由于 KCL 会记录此类异常，因此 您可在日志中进行查看。

如果您发现您的应用程序一直受到限制，则应考虑增加流的分片的数量。

监控 Kinesis Data Streams

您可使用以下功能监控 Amazon Kinesis Data Streams 中的数据流：

- [CloudWatch 指标](#) – Kinesis Data Streams 向 Amazon CloudWatch 发送自定义指标，并对每个流进行详细监控。
- [Kinesis 代理](#) – Kinesis 代理会发布自定义 CloudWatch 指标，帮助评测代理是否按预期运行。
- [API 日志记录](#) – Kinesis Data Streams 使用 Amazon CloudTrail 记录 API 调用并将数据存储在 Amazon S3 存储桶中。
- [Kinesis Client Library](#) – Kinesis Client Library (KCL) 提供针对分片、工作程序和 KCL 应用程序的指标。
- [Kinesis Producer Library](#) – Amazon Kinesis Producer Library (KPL) 提供针对分片、工作程序和 KPL 应用程序的指标。

有关常见监控问题、疑虑和问题排查的更多信息，请参阅以下文章：

- [我应该使用哪些指标来监控和排查 Kinesis Data Streams 问题？](#)
- [为什么 Kinesis Data Streams 中的 IteratorAgeMilliseconds 值不断增加？](#)

使用 Amazon CloudWatch 监控 Amazon Kinesis Data Streams 服务

Amazon Kinesis Data Streams 已与 Amazon CloudWatch 集成，方便您收集、查看和分析 Kinesis 数据流的 CloudWatch 指标。例如，要跟踪分片使用情况，您可监控 IncomingBytes 和 OutgoingBytes 指标并将它们与流中分片数量进行比较。

系统每分钟会自动收集一次您配置的流指标和分片级指标并将其推送到 CloudWatch。指标会存档两周。两周后，数据会被丢弃。

下表介绍了适用于 Kinesis Data Streams 的基本型流级别监控和增强型分片级别监控。

| 类型 | 描述 |
|---------|------------------|
| 基本 (流级) | 每分钟自动发送流级数据是免费的。 |

| 类型 | 描述 |
|------------|--|
| 增强 (分片级) | <p>每分钟发送分片级数据需要额外付费。要获取此级别的数据，您必须使用 EnableEnhancedMonitoring 操作专门针对该流启用它。</p> <p>有关定价的更多信息，请参阅 Amazon CloudWatch 产品页面。</p> |

Amazon Kinesis Data Streams 维度和指标

Kinesis Data Streams 在两个级别向 CloudWatch 发送指标；流级别和分片级别（可选）。流级别指标适用于一般条件下的最常见的监控使用案例。分片级别指标用于特定的监控任务，通常是与问题排查相关的任务，通过 [EnableEnhancedMonitoring](#) 操作启用。

有关从 CloudWatch 指标收集的统计数据的说明，请参阅《Amazon CloudWatch 用户指南》中的 [CloudWatch 统计数据](#)。

主题

- [基本的流级指标](#)
- [增强的分片级指标](#)
- [Amazon Kinesis Data Streams 指标的维度](#)
- [推荐的 Amazon Kinesis Data Streams 指标](#)

基本的流级指标

Amazon/Kinesis 命名空间包括以下流级指标。

Kinesis Data Streams 每分钟向 CloudWatch 发送一次流级别指标。这些指标始终可用。

| 指标 | 描述 |
|------------------|---|
| GetRecords.Bytes | <p>在指定时段内测量的从 Kinesis 流中检索的字节数。统计数据 Minimum、Maximum 和 Average 表示指定时段内流的单个 GetRecords 操作中的字节数。</p> <p>分片级别指标名称：OutgoingBytes</p> |

| 指标 | 描述 |
|------------------------------------|---|
| | 维度 : StreamName 统计数据 : Minimum、Maximum、Average、Sum、Samples 单位 : 字节 |
| GetRecords.IteratorAge | 不再使用此指标。使用 .GetRecords.IteratorAgeMilliseconds |
| GetRecords.IteratorAgeMilliseconds | 在指定时段内测量的，对某个 Kinesis 流进行的所有 GetRecords 调用中最后一条记录的存在时间。存在时间是当前时间与最后一条 GetRecords 调用记录写入流的时间之差。Minimum 和 Maximum 统计数据可用于跟踪 Kinesis 消费端应用程序的进度。值为“零”表示正在读取的记录已完全与流匹配。 分片级别指标名称 : IteratorAgeMilliseconds 维度 : StreamName 统计数据 : Minimum、Maximum、Average、Samples 单位 : 毫秒 |
| GetRecords.Latency | 在指定时段内测量的每个 GetRecords 操作所用的时间。 维度 : StreamName 统计数据 : Minimum、Maximum、Average 单位 : 毫秒 |

| 指标 | 描述 |
|--------------------|---|
| GetRecords.Records | <p>在指定时段内测量的从分片中检索的记录数。统计数据 Minimum、Maximum 和 Average 表示指定时段内流的单个 GetRecords 操作中的记录数。</p> <p>分片级别指标名称 : OutgoingRecords</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |
| GetRecords.Success | <p>在指定时段内测量的每个流中的成功 GetRecords 操作数。</p> <p>维度 : StreamName</p> <p>统计数据 : Average、Sum、Samples</p> <p>单位 : 计数</p> |
| IncomingBytes | <p>在指定时段内成功放入 Kinesis 流的字节数。该指标包含来自 PutRecord 和 PutRecords 的字节数。统计数据 Minimum、Maximum 和 Average 表示指定时段内流的单个 put 操作中的字节数。</p> <p>分片级别指标名称 : IncomingBytes</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 字节</p> |

| 指标 | 描述 |
|-------------------|---|
| IncomingRecords | <p>在指定时段内成功放入 Kinesis 流的记录数。该指标包含来自 PutRecord 和 PutRecords 的记录数。统计数据 Minimum、Maximum 和 Average 表示指定时段内流的单个 put 操作中的记录数。</p> <p>分片级别指标名称 : IncomingRecords</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |
| PutRecord.Bytes | <p>在指定时段内使用 PutRecord 操作放入 Kinesis 流的字节数。</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 字节</p> |
| PutRecord.Latency | <p>在指定时段内测量的每个 PutRecord 操作所用的时间。</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average</p> <p>单位 : 毫秒</p> |

| 指标 | 描述 |
|--------------------|---|
| PutRecord.Success | <p>在指定时段内测量的每个 Kinesis 流中的成功 PutRecord 操作数。平均值反映了对流的成功写入的百分比。</p> <p>维度 : StreamName</p> <p>统计数据 : Average、Sum、Samples</p> <p>单位 : 计数</p> |
| PutRecords.Bytes | <p>在指定时段内使用 PutRecords 操作放入 Kinesis 流的字节数。</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 字节</p> |
| PutRecords.Latency | <p>在指定时段内测量的每个 PutRecords 操作所用的时间。</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average</p> <p>单位 : 毫秒</p> |
| PutRecords.Records | <p>此指标已弃用。使用 .PutRecords.SuccessfulRecords</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |

| 指标 | 描述 |
|------------------------------|--|
| PutRecords.Success | <p>在指定时段内测量的，每个 Kinesis 流中至少有一条记录成功的 PutRecords 操作的数量。</p> <p>维度：StreamName</p> <p>统计数据：Average、Sum、Samples</p> <p>单位：计数</p> |
| PutRecords.TotalRecords | <p>在指定时段内测量的，每个 Kinesis 数据流的 PutRecords 操作中发送的记录总数。</p> <p>维度：StreamName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |
| PutRecords.SuccessfulRecords | <p>在指定时段内测量的，每个 Kinesis 数据流的 PutRecords 操作中的成功记录数。</p> <p>维度：StreamName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |
| PutRecords.FailedRecords | <p>在指定时段内测量的，每个 Kinesis 数据流的 PutRecords 操作中因内部故障而遭拒的记录数。偶尔会出现内部故障，遇到之时请重试。</p> <p>维度：StreamName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |

| 指标 | 描述 |
|-----------------------------------|---|
| PutRecords.ThrottledRecords | <p>在指定时段内测量的，每个 Kinesis 数据流的 PutRecords 操作中因节流而遭拒的记录数。</p> <p>维度：StreamName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |
| ReadProvisionedThroughputExceeded | <p>在指定时段内针对流的受限的 GetRecords 调用数。此指标的最常用的统计数据为 Average。</p> <p>当统计数据 Minimum 的值为 1 时，流的所有记录在指定时段内将受限。</p> <p>当统计数据 Maximum 的值为 0 (零) 时，流的任何记录在指定时段内将不受限。</p> <p>分片级别指标名称：ReadProvisionedThroughputExceeded</p> <p>维度：StreamName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |
| SubscribeToShard.RateExceeded | <p>当新订阅尝试失败时，将发出此指标，因为同一消费端已有活动订阅，或者超过了此操作允许的每秒调用数，也将发出此指标。</p> <p>维度：StreamName、ConsumerName</p> |

| 指标 | 描述 |
|--|---|
| SubscribeToShard.Success | <p>此指标记录 SubscribeToShard 订阅是否已成功建立。订阅最多只能有效 5 分钟。因此，该指标至少每 5 分钟发送一次。</p> <p>维度：StreamName、ConsumerName</p> |
| SubscribeToShardEvent.Bytes | <p>在指定时段内测量的从分片中接收的字节数。统计数据 Minimum、Maximum 和 Average 表示指定时段内单个事件中的已发布字节数。</p> <p>分片级别指标名称：OutgoingBytes</p> <p>维度：StreamName、ConsumerName</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：字节</p> |
| SubscribeToShardEvent.MillisBehindLatest | <p>读取记录距离流开头的毫秒数，表示消费端落后当前时间有多远。</p> <p>维度：StreamName、ConsumerName</p> <p>统计数据：Minimum、Maximum、Average、Samples</p> <p>单位：毫秒</p> |

| 指标 | 描述 |
|-------------------------------|--|
| SubscribeToShardEvent.Records | <p>在指定时段内测量的从分片中接收的记录数。统计数据 Minimum、Maximum 和 Average 表示指定时段内单个事件中的记录数。</p> <p>分片级别指标名称 : OutgoingRecords</p> <p>维度 : StreamName、ConsumerName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |
| SubscribeToShardEvent.Success | <p>每次成功发布事件时都会发出此指标。只有在有活动订阅时，才会发出它。</p> <p>维度 : StreamName、ConsumerName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |

| 指标 | 描述 |
|------------------------------------|--|
| WriteProvisionedThroughputExceeded | <p>在指定时段内因流限制而被拒绝的记录数。该指标包含来自 PutRecord 和 PutRecords 操作的限制。此指标的最常用的统计数据为 Average。</p> <p>当统计数据 Minimum 的值不为零时，流的记录在指定时段内将受限。</p> <p>当统计数据 Maximum 的值为 0 (零) 时，流的任何记录在指定时段内将不受限。</p> <p>分片级别指标名称 : WriteProvisionedThroughputExceeded</p> <p>维度 : StreamName</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |

增强的分片级指标

Amazon/Kinesis 命名空间包括以下分片级指标。

Kinesis 每分钟向 CloudWatch 发送以下分片级别指标一次。每个指标维度创建 1 个 CloudWatch 指标，每月进行大约 43200 次 PutMetricData API 调用。默认情况下，这些指标未启用。系统会对 Kinesis 发出的增强型指标收费。有关更多信息，请参阅 Amazon CloudWatch 自定义指标标题下的 [Amazon CloudWatch 定价](#)。按每个月每个指标每个分片收费。

| 指标 | 描述 |
|---------------|--|
| IncomingBytes | <p>在指定时段内成功放置到分片的字节数。该指标包含来自 PutRecord 和 PutRecords 的字节数。统计数据 Minimum、Maximum 和 Average 表示指定时段内分片的单个 put 操作中的字节数。</p> <p>流级别指标名称 : IncomingBytes</p> |

| 指标 | 描述 |
|-------------------------|--|
| | <p>维度 : StreamName、ShardId</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 字节</p> |
| IncomingRecords | <p>在指定时段内成功放置到分片的记录数。该指标包含来自 PutRecord 和 PutRecords 的记录数。统计数据 Minimum、Maximum 和 Average 表示指定时段内分片的单个 put 操作中的记录数。</p> <p>流级别指标名称 : IncomingRecords</p> <p>维度 : StreamName、ShardId</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |
| IteratorAgeMilliseconds | <p>对某个分片进行的所有 GetRecords 调用中最后一条记录的存在时间，是在指定的时间段测量的。存在时间是当前时间与最后一条 GetRecords 调用记录写入流的时间之差。Minimum 和 Maximum 统计数据可用于跟踪 Kinesis 消费端应用程序的进度。值为 0 (零) 表示正在读取的记录已完全与流匹配。</p> <p>流级别指标名称 : GetRecords.IteratorAgeMilliseconds</p> <p>维度 : StreamName、ShardId</p> <p>统计数据 : Minimum、Maximum、Average、Samples</p> <p>单位 : 毫秒</p> |

| 指标 | 描述 |
|-----------------|--|
| OutgoingBytes | <p>在指定时段内测量的从分片中检索的字节数。统计数据 Minimum、Maximum 和 Average 表示单个 GetRecords 操作中返回的字节数或指定时段内分片的单个 SubscribeToShard 事件中发布的字节数。</p> <p>流级别指标名称 : GetRecords.Bytes</p> <p>维度 : StreamName、ShardId</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 字节</p> |
| OutgoingRecords | <p>在指定时段内测量的从分片中检索的记录数。统计数据 Minimum、Maximum 和 Average 表示单个 GetRecords 操作中返回的记录数或指定时段内分片的单个 SubscribeToShard 事件中发布的记录数。</p> <p>流级别指标名称 : GetRecords.Records</p> <p>维度 : StreamName、ShardId</p> <p>统计数据 : Minimum、Maximum、Average、Sum、Samples</p> <p>单位 : 计数</p> |

| 指标 | 描述 |
|-----------------------------------|---|
| ReadProvisionedThroughputExceeded | <p>在指定时段内针对分片的受限的 GetRecords 调用数。此异常计数涵盖了以下限制的所有维度：每秒读取每个分片 5 次或每分片每秒 2 MB。此指标的最常用的统计数据为 Average。</p> <p>当统计数据 Minimum 的值为 1 时，分片的所有记录在指定时段内将受限。</p> <p>当统计数据 Maximum 的值为 0 (零) 时，分片的任何记录在指定时段内将不受限。</p> <p>流级别指标名称：ReadProvisionedThroughputExceeded</p> <p>维度：StreamName、ShardId</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |

| 指标 | 描述 |
|------------------------------------|---|
| WriteProvisionedThroughputExceeded | <p>在指定时段内因分片限制而被拒绝的记录数。此指标包括来自 PutRecord 和 PutRecords 操作的限制，并涵盖以下限制的所有维度：每分片每秒 1000 条记录或每分片每秒 1 MB。此指标的最常用的统计数据为 Average。</p> <p>当统计数据 Minimum 的值不为零时，分片的记录在指定时段内将受限。</p> <p>当统计数据 Maximum 的值为 0 (零) 时，分片的任何记录在指定时段内将不受限。</p> <p>流级别指标名称：WriteProvisionedThroughputExceeded</p> <p>维度：StreamName、ShardId</p> <p>统计数据：Minimum、Maximum、Average、Sum、Samples</p> <p>单位：计数</p> |

Amazon Kinesis Data Streams 指标的维度

| 维度 | 描述 |
|------------|---|
| StreamName | Kinesis 流的名称。所有可用统计数据按 StreamName 进行筛选。 |

推荐的 Amazon Kinesis Data Streams 指标

Kinesis Data Streams 的客户可能会对 Amazon Kinesis Data Streams 的几个指标特别感兴趣。以下列表提供了推荐的指标及其用法。

| 指标 | 使用说明 |
|--|---|
| GetRecords.IteratorAgeMilliseconds | 跟踪流中所有分片和消费端的读取位置。如果某个迭代器的寿命超过了保留期的 50% (默认值为 24 小时, 可配置为最高 7 天), 则存在由于记录过期造成数据丢失的风险。我们建议您针对 Maximum 统计数据使用 CloudWatch 警报, 以便在此问题构成风险之前提醒您。要了解使用此指标的示例情景, 请参阅 消费端记录处理滞后 。 |
| ReadProvisionedThroughputExceeded | 当您的消费端端记录处理滞后时, 有时难以确定瓶颈的位置。使用此指标可确定读取操作是否由于超出了读取吞吐量上限而受到限制。此指标的最常用的统计数据为 Average。 |
| WriteProvisionedThroughputExceeded | 这与 ReadProvisionedThroughputExceeded 指标的用途相同, 但是此指标用于流的创建者 (put) 端。此指标的最常用的统计数据为 Average。 |
| PutRecords.Success, PutRecords.Success | 我们建议针对 Average 统计数据使用 CloudWatch 警报, 以指示将记录放入流失败的时间。根据创建器所使用的对象选择一种或两种 put 类型。如果使用的是 Amazon Kinesis Producer Library (KPL), 请使用 PutRecords.Success。 |
| GetRecords.Success | 我们建议针对 Average 统计数据使用 CloudWatch 警报, 以指示从流中读取记录失败的时间。 |

访问 Kinesis Data Streams 的 Amazon CloudWatch 指标

您可以使用 CloudWatch 控制台、命令行或 CloudWatch API 监控 Kinesis Data Streams 的指标。以下过程介绍如何使用这些不同的方式访问指标。

使用 CloudWatch 控制台访问指标

1. 通过 <https://console.aws.amazon.com/cloudwatch/> 打开 CloudWatch 控制台。
2. 在导航栏中, 选择一个区域。
3. 在导航窗格中, 选择指标。

4. 在 CloudWatch Metrics by Category (按类别显示的 CloudWatch 指标) 窗格中，选择 Kinesis Metrics (Kinesis 指标)。
5. 单击相关行可查看指定的 MetricName 和 StreamName 的统计数据。

注意：大部分控制台统计数据名称与上面列出的对应 CloudWatch 指标名称匹配，但读取吞吐量和写入吞吐量除外。这些统计数据按照 5 分钟的间隔计算：写入吞吐量监控 IncomingBytes CloudWatch 指标，读取吞吐量监控 GetRecords.Bytes。
6. (可选) 在图形窗格中，选择一个统计数据和时间段，然后使用这些设置创建 CloudWatch 警报。

使用 Amazon CLI 访问指标

使用 [list-metrics](#) 和 [get-metric-statistics](#) 命令。

使用 CloudWatch CLI 访问指标

使用 [mon-list-metrics](#) 和 [mon-get-stats](#) 命令。

使用 CloudWatch API 访问指标

使用 [ListMetrics](#) 和 [GetMetricStatistics](#) 操作。

使用 Amazon CloudWatch 监控 Kinesis Data Streams 代理运行状况

代理会发布自定义 CloudWatch 指标以及 AmazonKinesisAgent 命名空间。这些指标可帮助您评测代理是否按指定方式将数据提交到 Kinesis Data Streams、运行是否正常以及在数据创建器上是否使用适当数量的 CPU 和内存资源。记录数和发送的字节数等指标对于了解代理将数据提交到流的速率非常有用。当这些指标低于预期阈值一定的百分比或者降低为零时，可能表明存在配置问题、网络错误或代理运行状况问题。诸如主机上的 CPU 和内存消耗以及代理错误计数器等指标可用于指示数据创建器资源使用情况，并提供对潜在的配置或主机错误的深入分析。最后，代理还会记录服务异常，以帮助调查代理问题。这些指标在代理配置设置 `cloudwatch.endpoint` 指定的区域中报告。从多个 Kinesis 代理发布的 CloudWatch 指标是聚合或合并的。有关代理配置的更多信息，请参阅[指定代理配置设置](#)。

使用 CloudWatch 进行监控

Kinesis Data Streams 代理会向 CloudWatch 发送以下指标：

| 指标 | 描述 |
|--------------------|--|
| BytesSent | 在指定时段内发送到 Kinesis Data Streams 的字节数。 单位：字节 |
| RecordSendAttempts | 在指定的时间范围内对 PutRecords 的一次调用中尝试的记录数（第一次，或者作为重试）。 单位：计数 |
| RecordSendErrors | 在指定时间范围内对 PutRecords 的一次调用中返回故障状态的记录数，包括重试。 单位：计数 |
| ServiceErrors | 在指定时间范围内产生服务错误（限制错误之外的其他错误）的 PutRecords 调用次数。 单位：计数 |

使用 Amazon CloudTrail 记录 Amazon Kinesis Data Streams API 调用

Amazon Kinesis Data Streams 已与 Amazon CloudTrail 集成，后者是一种服务，可在 Kinesis Data Streams 中提供由用户、角色或 Amazon 服务所采取操作的记录。CloudTrail 将 Kinesis Data Streams 的所有 API 调用作为事件捕获。捕获的调用包括来自 Kinesis Data Streams 控制台的调用以及对 Kinesis Data Streams API 操作的代码调用。如果您创建跟踪，则可以将 CloudTrail 事件持续交付到 Amazon S3 存储桶，包括 Kinesis Data Streams 的事件。如果您不配置跟踪，则仍可在 CloudTrail 控制台中的事件历史记录中查看最新事件。使用 CloudTrail 收集的信息，您可以确定向 Kinesis Data Streams 发出了什么请求、发出请求的 IP 地址、何人发出的请求、请求的发出时间以及其他详细信息。

要了解有关 CloudTrail 的更多信息（包括如何对其进行配置和启用），请参阅 [Amazon CloudTrail 用户指南](#)。

CloudTrail 中的 Kinesis Data Streams 信息

在您创建 Amazon 账户时，将在该账户上启用 CloudTrail。当 Kinesis Data Streams 中发生受支持的事件活动时，该活动将记录在 CloudTrail 事件中，并与其他 Amazon 服务事件一同保存在事件历史记录中。您可以在 Amazon 账户中查看、搜索和下载最新事件。有关更多信息，请参阅[使用 CloudTrail 事件历史记录查看事件](#)。

要持续记录 Amazon 账户中的事件（包括 Kinesis Data Streams 的事件），请创建跟踪。通过跟踪记录，CloudTrail 可将日志文件传送至 Amazon S3 存储桶。预设情况下，在控制台中创建跟踪时，此跟踪应用于所有 Amazon 区域。此跟踪记录在 Amazon 分区中记录所有区域中的事件，并将日志文件传送至您指定的 Amazon S3 存储桶。此外，您可以配置其他 Amazon 服务，进一步分析在 CloudTrail 日志中收集的事件数据并采取行动。有关更多信息，请参阅下列内容：

- [创建跟踪概述](#)
- [CloudTrail 支持的服务和集成](#)
- [为 CloudTrail 配置 Amazon SNS 通知](#)
- [从多个区域接收 CloudTrail 日志文件](#)和[从多个账户接收 CloudTrail 日志文件](#)

Kinesis Data Streams 支持将以下操作记录为 CloudTrail 日志文件中的事件：

- [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](#)

每个事件或日志条目都包含有关生成请求的人员信息。身份信息可帮助您确定以下内容：

- 请求是使用根用户凭证还是 Amazon Identity and Access Management (IAM) 用户凭证发出的。
- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 Amazon 服务发出。

有关更多信息，请参阅 [CloudTrail userIdentity 元素](#)。

示例：Kinesis Data Firehose 日志文件条目

跟踪是一种配置，可用于将事件作为日志文件传送到您指定的 Amazon S3 存储桶。CloudTrail 日志文件包含一个或多个日记账条目。一个事件表示来自任何源的一个请求，包括有关所请求的操作、操作的日期和时间、请求参数等方面的信息。CloudTrail 日志文件不是公用 API 调用的有序堆栈跟踪，因此它们不会按任何特定顺序显示。

下面的示例显示了一个 CloudTrail 日志条目，该条目说明了 CreateStream、DescribeStream、ListStreams、DeleteStream、SplitShard 和 MergeShards 操作。

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

使用 Amazon CloudWatch 监控 Kinesis Client Library

适用于 Amazon Kinesis Data Streams 的 [Kinesis Client Library](#) (KCL) 会代表您发布自定义 Amazon CloudWatch 指标，并使用您 KCL 应用程序的名称作为命名空间。您可以通过导航至 [CloudWatch 控制台](#) 并选择自定义指标来查看这些指标。有关自定义指标的更多信息，请参阅《Amazon CloudWatch 用户指南》中的 [发布自定义指标](#)。

对于由 KCL 上传到 CloudWatch 的指标，系统会象征性地收取少许费用；具体而言，将收取 Amazon CloudWatch 自定义指标和 Amazon CloudWatch API 请求费用。有关更多信息，请参阅 [Amazon CloudWatch 定价](#)。

主题

- [指标和命名空间](#)
- [指标级别和维度](#)
- [指标配置](#)
- [指标的列表](#)

指标和命名空间

用于上传指标的命名空间是您在启动 KCL 时指定的应用程序名称。

指标级别和维度

可通过两个选项控制上传到 CloudWatch 的指标：

指标级别

每个指标分配有一个独立的级别。在您设置指标报告级别后，独立级别低于报告级别的指标将不会发送到 CloudWatch。级别如下：NONE、SUMMARY 和 DETAILED。默认设置为 DETAILED；即，所有指标都将发送到 CloudWatch。NONE 报告级别意味着不会发送任何指标。有关将哪些级别分配到哪些指标的信息，请参阅 [指标的列表](#)。

启用的维度

每个 KCL 指标都有会发送到 CloudWatch 的关联维度。在 KCL 2.x 中，如果将 KCL 配置为处理单个数据流，则默认启用所有指标维度（Operation、ShardId 和 WorkerIdentifier）。此外，在 KCL 2.x 中，如果将 KCL 配置为处理单个数据流，则无法禁用 Operation 维度。在 KCL 2.x 中，如果将 KCL 配置为处理多个数据流，则默认启用所有指标维度（Operation、ShardId、StreamId 和 WorkerIdentifier）。此外，在 KCL 2.x 中，如果将 KCL 配置为处理多个数据流，则无法禁用 Operation 和 StreamId 维度。StreamId 维度仅对 per-shard 指标有效。

在 KCL 1.x 中，默认只启用 Operation 和 ShardId 维度，而 WorkerIdentifier 维度会被禁用。在 KCL 1.x 中，无法禁用 Operation 维度。

有关 CloudWatch 指标维度的更多信息，请参阅《Amazon CloudWatch 用户指南》中“Amazon CloudWatch 的概念”主题下的[维度](#)部分。

启用 WorkerIdentifier 维度后，如果特定 KCL 工作程序每次重新启动时使用的是不同的工作程序 ID 属性值，则会将新的指标组和新的 WorkerIdentifier 维度值组发送到 CloudWatch。如果您需要 WorkerIdentifier 维度值在特定的 KCL 工作程序重新启动时保持不变，则必须在每个工作程序初始化时显式指定相同的工作程序 ID 值。请注意，每个有效 KCL 工作程序的工作程序 ID 值在所有 KCL 工作程序中必须是唯一的。

指标配置

指标级别和启用的维度可使用 KinesisClientLibConfiguration 实例进行配置，该实例将在启动 KCL 应用程序时将传递给工作程序。在 MultiLangDaemon 案例中，metricsLevel 和 metricsEnabledDimensions 属性可在用于启动 MultiLangDaemon KCL 应用程序的 .properties 文件中指定。

可向指标级别分配下列三个值之一：NONE、SUMMARY 或 DETAILED。启用的维度值在 CloudWatch 指标允许的维度的列表中必须是以逗号分隔的字符串。KCL 应用程序使用的维度为 Operation、ShardId 和 WorkerIdentifier。

指标的列表

下表列出了按作用域和操作分组的 KCL 指标。

主题

- [Per-KCL-application 指标](#)
- [Per-worker 指标](#)
- [Per-shard 指标](#)

Per-KCL-application 指标

这些指标跨应用程序作用域内的所有 KCL 工作程序聚合一起，就像 Amazon CloudWatch 命名空间所定义的那样。

主题

- [LeaseAssignmentManager](#)
- [InitializeTask](#)
- [ShutdownTask](#)
- [ShardSyncTask](#)
- [BlockOnParentTask](#)
- [PeriodicShardSyncManager](#)
- [MultistreamTracker](#)

LeaseAssignmentManager

LeaseAssignmentManager 操作负责向工作程序分配租约，并重新平衡工作程序之间的租约，以实现工作程序资源的均匀利用。此操作的逻辑包括从租约表中读取租约相关元数据，以及从工作程序指标表中读取指标，并执行租约分配。

| 指标 | 描述 |
|--------------------------------|--|
| LeaseAndWorkerMetricsLoad.Time | 指在租约分配管理器 (LAM) 中加载所有租约和工作程序指标条目所花费的时间，LAM 是 KCL 3.x 中引入的新租约分配和负载均衡算法。 |

| 指标 | 描述 |
|--------------------------------------|---|
| | 单位 : 毫秒 |
| TotalLeases | <p>当前 KCL 应用程序的租约总数。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |
| NumWorkers | <p>当前 KCL 应用程序中的工作程序总数。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |
| AssignExpiredOrUnassignedLeases.Time | <p>指在内存中对过期租约进行分配的时间。</p> <p>指标级别 : 详细</p> <p>单位 : 毫秒</p> |
| LeaseSpillover | <p>指由于达到每个工作程序的最大租约数或最大吞吐量限制而未分配的租约数。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |
| BalanceWorkerVariance.Time | <p>指在内存中进行工作程序之间租约平衡的时间。</p> <p>指标级别 : 详细</p> <p>单位 : 毫秒</p> |
| NumOfLeasesReassignment | <p>指在当前重新分配迭代中重新分配的租约总数。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |

| 指标 | 描述 |
|--------------------------------|---|
| FailedAssignmentCount | <p>指对 DynamoDB 租约表进行 AssignLease 调用失败的次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| ParallellyAssignLeases.Time | <p>指向 DynamoDB 租约表刷新分配的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| ParallellyAssignLeases.Success | <p>指成功刷新分配的次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| TotalStaleWorkerMetricsEntry | <p>指必须清理的工作程序指标条目总数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| StaleWorkerMetricsCleanup.Time | <p>指从 DynamoDB 工作程序指标表中删除工作程序指标条目的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| Time | <p>LeaseAssignmentManager 操作花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |
| 成功 | <p>LeaseAssignmentManager 操作已成功完成的次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |

| 指标 | 描述 |
|-----------------------------------|--|
| ForceLeaderRelease | <p>表示租约分配管理器已连续 3 次失败，且领导工作程序正在卸除领导权。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| NumWorkersWithInvalidEntry | <p>指视为无效的工作程序指标条目的数量。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| NumWorkersWithFailingWorkerMetric | <p>指工作程序指标条目数量，其中一个工作程序指标值为 -1 (表示工作程序指标值不可用)。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| LeaseDeserializationFailureCount | <p>指租约表中未能反序列化的租约条目。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |

InitializeTask

`InitializeTask` 操作负责初始化 KCL 应用程序的记录处理器。此操作的逻辑包括从 Kinesis Data Streams 中获取分片迭代器并初始化记录处理程序。

| 指标 | 描述 |
|--|--|
| KinesisDataFetcher.getIterator.Success | <p>每个 KCL 应用程序的成功 <code>GetShardIterator</code> 操作数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |

| 指标 | 描述 |
|-------------------------------------|---|
| KinesisDataFetcher.getIterator.Time | <p>给定 KCL 应用程序的每个 <code>GetShardIterator</code> 操作所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| RecordProcessor.initialize.Time | <p>记录处理器的初始化方法所花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |
| 成功 | <p>成功的记录处理器初始化的数目。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| Time | <p>KCL 工作程序初始化记录处理器所花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |

ShutdownTask

`ShutdownTask` 操作初始化分片处理的关闭顺序。这可能是因为分片被拆分或合并，或者当分片租赁从工作程序中丢失时。在这两种情况下，将调用记录处理器 `shutdown()` 函数。在分片被拆分或合并的情况下也会发现新的分片，这将创建一个或两个新的分片。

| 指标 | 描述 |
|---------------------|--|
| CreateLease.Success | <p>新的子分片在父分片关闭后成功添加到 KCL 应用程序 DynamoDB 表的次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |

| 指标 | 描述 |
|-------------------------------|--|
| CreateLease.Time | 在 KCL 应用程序 DynamoDB 表中添加新的子分片信息所花费的时间。 指标级别：详细 单位：毫秒 |
| UpdateLease.Success | 记录处理程序关闭期间成功的最终检查点的数目。 指标级别：详细 单位：计数 |
| UpdateLease.Time | 记录处理程序关闭期间检查点操作所花费的时间。 指标级别：详细 单位：毫秒 |
| RecordProcessor.shutdown.Time | 记录处理器的关闭方法所花费的时间。 指标级别：汇总 单位：毫秒 |
| 成功 | 成功的关闭任务的数目。 指标级别：汇总 单位：计数 |
| Time | KCL 工作程序关闭任务所花费的时间。 指标级别：汇总 单位：毫秒 |

ShardSyncTask

ShardSyncTask 操作会发现对 Kinesis 数据流的分片信息所做的更改，因此 KCL 应用程序可处理新的分片。

| 指标 | 描述 |
|---------------------|---|
| CreateLease.Success | <p>将新的分片信息添加到 KCL 应用程序 DynamoDB 表的成功尝试次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| CreateLease.Time | <p>在 KCL 应用程序 DynamoDB 表中添加新的分片信息所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| 成功 | <p>成功的分片同步操作的数目。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| Time | <p>分片同步操作所花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |

BlockOnParentTask

如果一个分片被拆分或与其他分片合并，则会创建新的子分片。BlockOnParentTask 操作确保新分片的记录处理在 KCL 完全处理父分片之前不会开始。

| 指标 | 描述 |
|------|---|
| 成功 | <p>父分片完成的成功检查的数目。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| Time | 父分片完成所花费的时间。 |

| 指标 | 描述 |
|----|----------------------------------|
| | 指标级别 ：汇总 单位 ：毫秒 |
| | |

PeriodicShardSyncManager

`PeriodicShardSyncManager` 负责检查 KCL 消费端应用程序正在处理的数据流、识别具有部分租约的数据流并转交出去进行同步。

当 KCL 配置为处理单个数据流（随后将 `NumStreamsToSync` 和 `NumStreamsWithPartialLeases` 的值设置为 1），且 KCL 也配置为处理多个数据流时，以下指标可用。

| 指标 | 描述 |
|--|--|
| <code>NumStreamsToSync</code> | 消费端应用程序正在处理的数据流（每个 Amazon 账户）的数量，这些数据流包含部分租约并且必须移交以进行同步。 指标级别 ：汇总 单位 ：计数 |
| <code>NumStreamsWithPartialLeases</code> | 消费端应用程序正在处理的数据流（每个 Amazon 账户）的数量，这些数据流包含部分租约。 指标级别 ：汇总 单位 ：计数 |
| 成功 | <code>PeriodicShardSyncManager</code> 能够成功识别消费端应用程序正在处理的数据流中部分租约的次数。 指标级别 ：汇总 单位 ：计数 |
| <code>Time</code> | <code>PeriodicShardSyncManager</code> 检查消费端应用程序正在处理的数据流以确定哪些数据流需要分片同步所花费的时间（以毫秒为单位）。 指标级别 ：汇总 |

| 指标 | 描述 |
|----|---------|
| | 单位 : 毫秒 |

MultistreamTracker

`MultistreamTracker` 接口能够让您构建可以同时处理多个数据流的 KCL 消费端应用程序。

| 指标 | 描述 |
|------------------------------|--|
| DeletedStreams.Count | <p>在指定时段内删除的数据流数量。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |
| ActiveStreams.Count | <p>正在处理的活动数据流的数量。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |
| StreamsPendingDeletion.Count | <p>依据 <code>FormerStreamsLeasesDeletionStrategy</code> 待删除的数据流的数量。</p> <p>指标级别 : 汇总</p> <p>单位 : 计数</p> |

Per-worker 指标

这些指标跨使用 Kinesis 数据流 (例如 Amazon EC2 实例) 中的数据的所有记录处理器进行聚合。

主题

- [WorkerMetricStatsReporter](#)
- [LeaseDiscovery](#)
- [RenewAllLeases](#)
- [TakeLeases](#)

WorkerMetricStatsReporter

`WorkerMetricStatReporter` 操作负责定期向工作程序指标表发布当前工作程序指标。 `LeaseAssignmentManager` 操作使用这些指标来执行租约分配。

| 指标 | 描述 |
|--|--|
| <code>InMemoryMetricStatsReporter.Failure</code> | <p>指由于某些工作程序指标失效而无法捕获内存中工作程序指标值的次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| <code>WorkerMetricStatsReporter.Time</code> | <p><code>WorkerMetricsStats</code> 操作花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |
| <code>WorkerMetricStatsReporter.Success</code> | <p><code>WorkerMetricsStats</code> 操作已成功完成的次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |

LeaseDiscovery

`LeaseDiscovery` 操作负责确定由 `LeaseAssignmentManager` 操作分配给当前工作程序的新租约。此操作的逻辑包括通过读取租约表的全局二级索引来确定分配给当前工作程序的租约。

| 指标 | 描述 |
|--|---|
| <code>ListLeaseKeysForWorker.Time</code> | <p>指在租约表上调用全局二级索引并获取分配给当前工作程序的租约键的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| <code>FetchNewLeases.Time</code> | 指从租约表中获取所有新租约的时间。 |

| 指标 | 描述 |
|---------------------|--|
| | <p>指标级别：详细</p> <p>单位：毫秒</p> |
| NewLeasesDiscovered | <p>指分配给工作程序的新租约总数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| Time | <p>LeaseDiscovery 操作花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |
| 成功 | <p>LeaseDiscovery 操作已成功完成的次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| OwnerMismatch | <p>指 GSI 响应和租约表的一致性读取中出现所有者不匹配的次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |

RenewAllLeases

RenewAllLeases 操作定期续订由特定工作程序实例拥有的分片租约。

| 指标 | 描述 |
|--------------------|---|
| RenewLease.Success | <p>工作程序成功续订租约的数目。</p> <p>指标级别：详细</p> <p>单位：计数</p> |

| 指标 | 描述 |
|-----------------|--|
| RenewLease.Time | <p>租约续订操作所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| CurrentLeases | <p>续订所有租约后由工作程序拥有的分片租约数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| LostLeases | <p>在尝试续订由工作程序拥有的所有租约后丢失的分片租约数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| 成功 | <p>工作程序的成功的租约续订操作次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| Time | <p>续订工作程序的所有租约所花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |

TakeLeases

TakeLeases 操作使所有 KCL 工作程序之间的记录处理达到平衡。如果当前 KCL 工作程序拥有的分片租约少于所需的分片租约，则将从已过载的另一个工作程序中提取分片租约。

| 指标 | 描述 |
|--------------------|--------------------------------------|
| ListLeases.Success | 成功从 KCL 应用程序 DynamoDB 表中检索所有分片租约的次数。 |

| 指标 | 描述 |
|-------------------|--|
| | <p>指标级别：详细</p> <p>单位：计数</p> |
| ListLeases.Time | <p>从 KCL 应用程序 DynamoDB 表中检索所有分片租约所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| TakeLease.Success | <p>工作程序成功从其他 KCL 工作程序中提取分片租约的次数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| TakeLease.Time | <p>利用该工作程序提取的租约更新租约表所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| NumWorkers | <p>工作程序总数，由特定工作程序标识。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| NeededLeases | <p>当前工作程序为平衡分片处理负载所需的分片租约数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| LeasesToTake | <p>工作程序将尝试提取的租约的数目。</p> <p>指标级别：详细</p> <p>单位：计数</p> |

| 指标 | 描述 |
|---------------|--|
| TakenLeases | <p>工作程序已成功提取的租约的数目。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| TotalLeases | <p>KCL 应用程序正在处理的分片的总数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| ExpiredLeases | <p>未由任何工作程序处理的分片的总数，由特定工作程序标识。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| 成功 | <p>TakeLeases 操作已成功完成的次数。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| Time | <p>工作程序的 TakeLeases 操作所花费的时间。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |

Per-shard 指标

这些指标跨单个记录处理程序聚合成一起。

ProcessTask

ProcessTask 操作利用当前迭代器位置调用 [GetRecords](#) 以从流中检索记录并调用记录处理器 `processRecords` 函数。

| 指标 | 描述 |
|---------------------------------------|--|
| KinesisDataFetcher.getRecords.Success | <p>每个 Kinesis 数据流分片的成功 GetRecords 操作数。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| KinesisDataFetcher.getRecords.Time | <p>Kinesis 数据流分片的每个 GetRecords 操作所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| UpdateLease.Success | <p>给定分片的记录处理程序完成的成功检查点的数目。</p> <p>指标级别：详细</p> <p>单位：计数</p> |
| UpdateLease.Time | <p>给定分片的每个检查点操作所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| DataBytesProcessed | <p>每个 ProcessTask 调用所处理的记录的总大小（以字节为单位）。</p> <p>指标级别：汇总</p> <p>单位：字节</p> |
| RecordsProcessed | <p>每个 ProcessTask 调用所处理的记录的数量。</p> <p>指标级别：汇总</p> <p>单位：计数</p> |
| ExpiredIterator | <p>调用 GetRecords 时收到的 ExpiredIteratorException 的数量。</p> <p>指标级别：汇总</p> |

| 指标 | 描述 |
|--------------------------------------|---|
| 单位 : 计数 | |
| MillisBehindLatest | 当前迭代器晚于分片中最新记录 (tip) 的时间。此值为小于或等于响应中最新一条记录的时间和当前时间之间的时差。与比较最新一条响应记录中的时间戳相比，此指标可以更精确地反映分片相对于末端的距离。此值适用于最近一批记录，而不是每条记录中所有时间戳的平均值。 |
| 指标级别 : 汇总 | |
| 单位 : 毫秒 | |
| RecordProcessor.pr ocessRecords.Time | 记录处理器的 <code>processRecords</code> 方法所花费的时间。 |
| 指标级别 : 汇总 | |
| 单位 : 毫秒 | |
| 成功 | 成功处理任务操作的数目。 |
| 指标级别 : 汇总 | |
| 单位 : 计数 | |
| Time | 处理任务操作所花费的时间。 |
| 指标级别 : 汇总 | |
| 单位 : 毫秒 | |

使用 Amazon CloudWatch 监控 Kinesis Producer Library

适用于 Amazon Kinesis Data Streams 的 [Amazon Kinesis Producer Library](#) (KPL) 代表您发布自定义 Amazon CloudWatch 指标。您可以通过导航至 [CloudWatch 控制台](#) 并选择自定义指标来查看这些指标。有关自定义指标的更多信息，请参阅《Amazon CloudWatch 用户指南》中的 [发布自定义指标](#)。

对于由 KPL 上传到 CloudWatch 的指标，系统会象征性地收取少许费用；具体而言，将对 Amazon CloudWatch 自定义指标和 Amazon CloudWatch API 请求收取费用。有关更多信息，请参阅 [Amazon CloudWatch 定价](#)。本地指标收集不会产生 CloudWatch 费用。

主题

- [指标、维度和命名空间](#)
- [指标级别和粒度](#)
- [本地访问和 Amazon CloudWatch 上传](#)
- [指标的列表](#)

指标、维度和命名空间

您可在启动 KPL 时指定应用程序名称，该名称随后在上传指标时将用作命名空间的一部分。这是可选操作；如果未设置应用程序名称，则 KPL 会提供一个默认值。

您也可以配置 KPL 将任意其他维度添加到指标。这在您希望获得 CloudWatch 指标中的更详细数据时很有用。例如，您可将主机名作为维度添加，随后该维度将允许您标识实例集中不均匀的负载分配。所有 KPL 配置设置是不可变的，因此您在初始化 KPL 实例后将无法更改其他维度。

指标级别和粒度

可通过两个选项控制上传到 CloudWatch 的指标的数目：

指标级别

这是对指标重要性的粗略估计。为每个指标分配了一个级别。如果设置了级别，低于该级别的指标将不会发送到 CloudWatch。级别为 NONE、SUMMARY 和 DETAILED。默认设置为 DETAILED；即，所有指标。NONE 表示没有任何指标，因此实际上未为该级别分配任何指标。

粒度

这可控制是否在其他粒度级别发出相同指标。级别为 GLOBAL、STREAM 和 SHARD。默认设置为 SHARD，其中包含粒度最高的指标。

选择 SHARD 后，发出将流名称和分片 ID 作为维度的指标。此外，还发出仅具有流名称维度的相同指标和不带流名称的指标。这意味着，对于特定指标，具有两个分片的两个流均会生成 7 个 CloudWatch 指标：一个针对每个分片的指标、一个针对每个流的指标以及一个针对整体的指标；所有指标都描述处于不同级别粒度的相同统计数据。有关说明，请见下图。

层次结构中的不同粒度级别与系统中的所有指标构成了一个基于指标名称的树：



| | | | | |
|----------------------|----------|----------|----------|----------|
| StreamName (STREAM): | Stream A | Stream B | Stream A | Stream B |
| | | | | |
| ShardID (SHARD): | Shard 0 | Shard 1 | Shard 0 | Shard 1 |

并不会在分片级别提供所有指标；一些指标本来就属于流级别或全局级别。不会在分片级别生成这些指标，即使已启用分片级别指标（上图中的 Metric Y）。

在指定其他维度时，必须提供 `tuple:<DimensionName, DimensionValue, Granularity>` 的值。粒度用于确定自定义维度在层次结构中的插入位置：GLOBAL 表示其他维度将插入到指标名称的后面；STREAM 表示其他维度将插入到流名称的后面；SHARD 表示其他维度将插入到分片 ID 的后面。如果为每个粒度级别提供多个其他维度，则这些维度将按给定顺序插入。

本地访问和 Amazon CloudWatch 上传

将实时本地提供当前 KPL 实例的指标；您可随时查询 KPL 以获得这些指标。KPL 本地计算每个指标的总和、平均值、最小值、最大值和计数，就像在 CloudWatch 中一样。

您可获取从程序启动到当前时间点或在过去 N 秒内（其中 N 为介于 1 和 60 之间的整数）使用滚动窗口累积的统计数据。

所有指标均可上传到 CloudWatch。这在跨多台主机聚合数据、监控和警报时特别有用。此功能在本地不可用。

如前所述，您可选择使用指标级别 和粒度 设置来上传的指标。未上传的指标在本地可用。

不支持单独上传数据点，因为如果流量过高，则每秒会生成数以百万计的上传。为此，KPL 会将指标本地聚合到 1 分钟存储桶中，并每分钟按启用的指标将统计数据对象上传到 CloudWatch 一次。

指标的列表

| 指标 | 描述 |
|---------------------|--|
| UserRecordsReceived | 由放置操作的 KPL 内核接收的逻辑用户记录的计数。在分片级别不可用。 指标级别：详细 单位：个 |

| 指标 | 描述 |
|-----------------------|--|
| UserRecordsPending | <p>当前挂起的用户记录的数目的定期取样。如果记录当前已缓冲并等待发送，或已发送和正在发送到后端服务，则记录处于挂起状态。在分片级别不可用。</p> <p>KPL 提供了专用方法在全局级别检索此指标，以便客户管理其放置速率。</p> <p>指标级别：详细</p> <p>单位：个</p> |
| UserRecordsPut | <p>已成功放置的逻辑用户记录的计数。</p> <p>KPL 会输出零作为失败记录数。这允许用于提供成功率的平均值、用于提供总尝试数的计数以及用于提供失败计数的计数与总和之间的差值。</p> <p>指标级别：汇总</p> <p>单位：个</p> |
| UserRecordsDataPut | <p>逻辑用户记录成功放置的字节数。</p> <p>指标级别：详细</p> <p>单位：字节</p> |
| KinesisRecordsPut | <p>已成功放置的 Kinesis Data Streams 记录的计数（每条 Kinesis Data Streams 记录均可包含多条用户记录）。</p> <p>KPL 会输出零作为失败记录数。这允许用于提供成功率的平均值、用于提供总尝试数的计数以及用于提供失败计数的计数与总和之间的差值。</p> <p>指标级别：汇总</p> <p>单位：个</p> |
| KinesisRecordsDataPut | <p>Kinesis Data Streams 记录中的字节数。</p> <p>指标级别：详细</p> <p>单位：字节</p> |

| 指标 | 描述 |
|------------------|--|
| ErrorsByCode | <p>每种错误代码的计数。这引入了 ErrorCode 和 StreamName 等正常维度以及 ShardId 的其他维度。并非每个错误都可跟踪到分片。无法跟踪的错误仅在流或全局级别发出。此指标捕获有关限制、分片映射更改、内部故障、服务不可用、超时等的信息。</p> <p>每条 Kinesis Data Streams 记录计算一次 Kinesis Data Streams API 错误。Kinesis Data Streams 记录中的多条用户记录不会生成多个计数。</p> <p>指标级别：汇总</p> <p>单位：个</p> |
| AllErrors | <p>这是由与 Errors by Code 相同的错误引发的，但并不区分类型。这在常规监控错误率而不需要手动汇总所有不同类型错误的计数时非常有用。</p> <p>指标级别：汇总</p> <p>单位：个</p> |
| RetriesPerRecord | <p>每个用户记录执行的重试次数。为一次性尝试成功的记录发出零。</p> <p>在用户记录完成时（成功时或不再能够重试时）发出数据。如果记录生存时间是一个较大的值，则此指标可能显著延迟。</p> <p>指标级别：详细</p> <p>单位：个</p> |
| BufferingTime | <p>用户记录到达 KPL 和离开前往后端之间的时间。此信息将基于记录传回给用户，还可用作聚合的统计数据。</p> <p>指标级别：汇总</p> <p>单位：毫秒</p> |

| 指标 | 描述 |
|--|--|
| Request Time | <p>执行 PutRecordsRequests 所花费的时间。</p> <p>指标级别：详细</p> <p>单位：毫秒</p> |
| User Records per Kinesis Record | <p>已聚合到单条 Kinesis Data Streams 记录中的逻辑用户记录的数目。</p> <p>指标级别：详细</p> <p>单位：个</p> |
| Amazon Kinesis Records per PutRecordsRequest | <p>已聚合到单个 PutRecordsRequest 中的 Kinesis Data Streams 记录的数目。在分片级别不可用。</p> <p>指标级别：详细</p> <p>单位：个</p> |
| User Records per PutRecordsRequest | <p>包含在 PutRecordsRequest 中的用户记录的总数。这大致相当于之前两个指标的产品。在分片级别不可用。</p> <p>指标级别：详细</p> <p>单位：个</p> |

Amazon Kinesis Data Streams 中的安全性

云安全Amazon是重中之重。作为Amazon客户，您将受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。

安全是双方共同承担Amazon的责任。[责任共担模式](#)将其描述为云的 安全性和云中的 安全性：

- 云安全 — Amazon 负责保护在Amazon云中运行Amazon服务的基础架构。 Amazon还为您提供可以安全使用的服务。作为 [Amazon 合规性计划](#)的一部分，我们的安全措施的有效性定期由第三方审计员进行测试和验证。要了解适用于 Kinesis Data Streams 的合规性计划，请参阅[按合规性计划提供的范围内 Amazon 服务](#)。
- 云端安全-您的责任由您使用的Amazon服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您组织的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 Kinesis Data Streams 时应用责任共担模式。以下主题说明如何配置 Kinesis Data Streams 以实现您的安全性和合规性目标。您还将学习如何使用其他Amazon服务来帮助您监控和保护您的 Kinesis Data Streams 资源。

主题

- [Amazon Kinesis Data Streams 中的数据保护](#)
- [使用 IAM 控制对 Amazon Kinesis Data Streams 资源的访问](#)
- [Amazon Kinesis Data Streams 的合规性验证](#)
- [Amazon Kinesis Data Streams 中的故障恢复能力](#)
- [Kinesis Data Streams 中的基础设施安全](#)
- [Kinesis Data Streams 的安全最佳实践](#)

Amazon Kinesis Data Streams 中的数据保护

使用 Amazon Key Management Service (Amazon KMS) 密钥进行服务器端加密，可以对 Amazon Kinesis Data Streams 中的静态数据进行加密，从而轻松满足严格的数据管理要求。

Note

如果您在Amazon通过命令行界面或 API 进行访问时需要经过 FIPS 140-2 验证的加密模块，
请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅 [《美国联邦信息处理标准 \(FIPS\) 第 140-2 版》](#)。

主题

- [什么是 Kinesis Data Streams 的服务器端加密？](#)
- [费用、区域和性能注意事项](#)
- [如何开始使用服务器端加密？](#)
- [创建和使用用户生成的 KMS 密钥](#)
- [使用用户生成的 KMS 密钥的权限](#)
- [验证 KMS 密钥权限并排查相关问题](#)
- [将 Amazon Kinesis Data Streams 与接口 VPC 端点搭配使用](#)

什么是 Kinesis Data Streams 的服务器端加密？

服务器端加密是 Amazon Kinesis Data Streams 中的Amazon KMS一项功能，它使用您指定的客户主密钥 (CMK) 在数据静止之前自动对其进行加密。数据在写入 Kinesis 流存储层之前加密，并在从存储检索到之后进行解密。因此，在 Kinesis Data Streams 服务中对数据进行静态加密。这样，您就可以满足严格的监管要求并增强您数据的安全性。

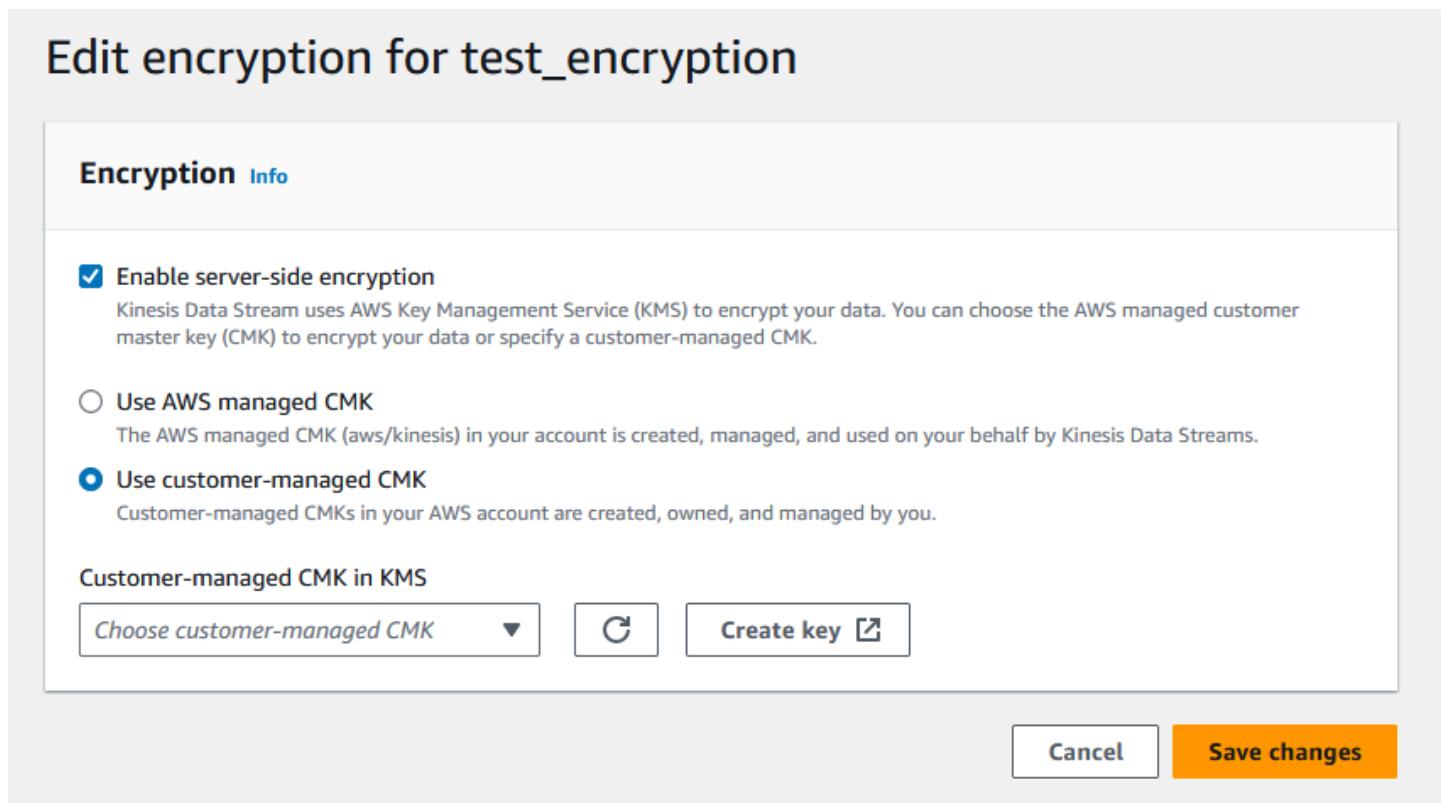
采用服务器端加密时，您的 Kinesis 流创建器和消费端不需要管理主密钥或加密操作。您的数据在进入和离开 Kinesis Data Streams 服务时会自动加密，因此您的静态数据会被加密。Amazon KMS 提供了服务器端加密功能使用的所有主密钥。Amazon KMS便于使用Amazon由管理的 Kinesis 的 CMK、Amazon KMS用户指定的 CMK 或导入到服务中的主密钥。Amazon KMS

Note

服务器端加密仅在启用加密后加密传入数据。启用服务器端加密后，未加密流中预先存在的数据不会加密。

加密您的数据流并与其他委托人共享访问权限时，您必须在密钥策略和外部账户的 IAM 策略中授予权限。Amazon KMS有关更多信息，请参阅 [允许其他账户中的用户使用 KMS 密钥](#)。

如果您已使用Amazon托管 KMS 密钥为数据流启用服务器端加密，并希望通过资源策略共享访问权限，则必须切换到使用客户托管密钥 (CMK) ，如下所示：



Enable server-side encryption

Kinesis Data Stream uses AWS Key Management Service (KMS) to encrypt your data. You can choose the AWS managed customer master key (CMK) to encrypt your data or specify a customer-managed CMK.

Use AWS managed CMK

The AWS managed CMK (aws/kinesis) in your account is created, managed, and used on your behalf by Kinesis Data Streams.

Use customer-managed CMK

Customer-managed CMKs in your AWS account are created, owned, and managed by you.

Customer-managed CMK in KMS

Choose customer-managed CMK ▼

Create key +

Cancel Save changes

此外，您必须允许您的共享主体实体使用 KMS 跨账户共享功能来访问您的 CMK。此外还务必要对共享主体实体的 IAM policy 进行更改。有关更多信息，请参阅[允许其他账户中的用户使用 KMS 密钥](#)。

费用、区域和性能注意事项

当你应用服务器端加密时，你需要支付 Amazon KMS API 使用量和密钥费用。与自定义 KMS 主密钥不同，(Default) aws/kinesis 客户主密钥 (CMK) 是免费提供的。但是，您仍必须支付因您产生的 Amazon Kinesis Data Streams API 使用费。

API 使用费适用于所有 CMK，包括自定义的 CMK。在 Kinesis Data Streams 轮换数据密钥时，大约每五分钟调用一次 Amazon KMS。在 30 天内，由 Kinesis 直播发起的 Amazon KMS API 调用的总费用应少于几美元。此费用会随着您在数据创建者和使用者身上使用的用户凭证数量而变化，因为每个用户凭证都需要对每个用户凭证进行唯一的 API 调用。Amazon KMS 当您使用 IAM 角色进行身份验证时，每个代入角色调用都会产生唯一的用户凭据。为了节约 KMS 成本，您可能想要缓存代入角色调用返回的用户凭据。

下面按资源介绍各项费用：

键

- Amazon由 (别名 aws/kinesis =) 管理的 Kinesis 用户主密钥是免费的。
- 用户生成的 KMS 密钥需要收取 KMS 密钥费用。有关更多信息 , 请参阅 [Amazon Key Management Service Pricing](#)。

API 使用费适用于所有 CMK , 包括自定义的 CMK。在 Kinesis Data Streams 轮换数据密钥时 , 大约每 5 分钟调用一次 KMS。以 30 天为一个月 , 由 Kinesis 数据流启动的 KMS API 调用的总费用应该不到几美元。请注意 , 此费用会随着您在数据创建者和使用者上使用的用户凭证数量而变化 , 因为每个用户凭证都需要对 Amazon KMS 进行唯一的 API 调用。当您使用 IAM 角色进行身份验证时 , 每个角色都 `assume-role-call` 将生成唯一的用户证书 , 您可能需要缓存返回的用户证书 `assume-role-call` 以节省 KMS 成本。

KMS API 使用量

对于每个加密流 , 当从 TIP 读取数据并在读取者和写入者之间使用单个 IAM account/user 访问密钥时 , Kinesis Amazon KMS 服务大约每 5 分钟调用该服务 12 次。不从 TIP 读取数据可能会导致更高的 Amazon KMS 服务呼叫量。生成新数据加密密钥的 API 请求需支付 Amazon KMS 使用费用。有关更多信息 , 请参阅 [Amazon Key Management Service Pricing: Usage](#)。

按区域的服务器端加密的可用性

目前 , Kinesis 数据流的服务器端加密可在所有支持 Kinesis 数据流 Amazon GovCloud 的区域 (包括 (美国西部) 和中国区域。有关 Kinesis Data Streams 支持的区域的更多信息 , <https://docs.aws.amazon.com/general/latest/gr/ak.html> 请参阅)。

性能注意事项

由于应用加密存在服务开销 , 应用服务器端加密的开销会将 `PutRecord`、`PutRecords` 和 `GetRecords` 的典型延迟增加不到 100 微秒。

如何开始使用服务器端加密 ?

开始使用服务器端加密的最简单方法是使用 Amazon Web Services 管理控制台和 Amazon Kinesis KMS 服务密钥。aws/kinesis

下面的过程演示了如何为 Kinesis 流启用服务器端加密。

为 Kinesis 流启用服务器端加密

1. 登录Amazon Web Services 管理控制台并打开 [Amazon Kinesis Data Streams](#) 控制台。
2. 在 Amazon Web Services 管理控制台 中创建或选择 Kinesis 流。
3. 选择 Details (详细信息) 选项卡。
4. 在 Server-side encryption (服务器端加密) 中，选择 edit (编辑)。
5. 除非您希望使用用户生成的 KMS 主密钥，否则应确保选中 (Default) aws/kinesis ((默认) aws/kinesis) KMS 主密钥。这是 Kinesis 服务生成的 KMS 主密钥。选择启用，然后选择保存。

Note

默认的 Kinesis 服务主密钥是免费的，但是，Kinesis 对该Amazon KMS服务进行的 API 调用需要支付 KMS 使用成本。

6. 流结束挂起状态。一旦该流恢复到活动状态并启用加密，对于所有写入该流的传入数据，都将使用您选择的 KMS 主密钥进行加密。
7. 要禁用服务器端加密，请在中的服务器端加密中选择禁用Amazon Web Services 管理控制台，然后选择保存。

创建和使用用户生成的 KMS 密钥

本部分介绍如何创建和使用您自己的 KMS 密钥，而不是使用 Amazon Kinesis 管理的主密钥。

创建用户生成的 KMS 密钥

有关创建您自己的密钥的说明，请参阅《Amazon Key Management Service Developer Guide》中的 [Creating Keys](#)。在您为自己的账户创建密钥后，Kinesis Data Streams 服务会在 KMS 主密钥列表中返回这些密钥。

使用用户生成的 KMS 密钥

向您的消费者、生产者和管理员应用正确的权限后，您可以在自己的Amazon账户或其他Amazon账户中使用自定义 KMS 密钥。在 中的 KMS Master KeyAmazon Web Services 管理控制台 列表内显示您账户中的所有 KMS 主密钥。

要使用位于另一个账户中的自定义 KMS 主密钥，您需要有使用这些密钥的权限。您还必须在 Amazon Web Services 管理控制台的 ARN 输入框中指定 KMS 主密钥的 ARN。

使用用户生成的 KMS 密钥的权限

在将服务器端加密与用户生成的 KMS 密钥配合使用之前，必须配置Amazon KMS密钥策略以允许对流进行加密以及对流记录进行加密和解密。有关Amazon KMS权限的示例和更多信息，请参阅[AmazonKMS API 权限：操作和资源参考](#)。

Note

使用默认服务密钥进行加密不需要应用自定义 IAM 权限。

在您使用用户生成的 KMS 主密钥之前，请确保您的 Kinesis 流创建器和消费端（IAM 主体）是 KMS 主密钥政策中的用户。否则，与流相关的读写操作会失败，这可能最终导致数据丢失、处理延迟或应用程序挂起。您可以使用 IAM policy 来管理 KMS 密钥的权限。有关更多信息，请参阅[将 IAM 策略与 Amazon KMS 配合使用](#)。

产生器权限示例

您的 Kinesis 流创建器必须拥有 kms:GenerateDataKey 权限。

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"  
    }  
  ]  
}
```

```
        }
    ]
}
```

消费端权限示例

您的 Kinesis 流消费端必须拥有 `kms:Decrypt` 权限。

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"
        }
    ]
}
```

适用于 Apache Flink 的亚马逊托管服务，并Amazon Lambda使用角色来消费 Kinesis 直播。确保将 `kms:Decrypt` 权限添加到这些消费端使用的角色。

流管理员权限

Kinesis 流管理员必须有权调用 `kms>List*` 和 `kms:DescribeKey*`。

验证 KMS 密钥权限并排查相关问题

在 Kinesis 直播上启用加密后，我们建议您使用以下亚马逊 CloudWatch 指标监控 `putRecord`、`putRecords` 和 `getRecords` 呼叫的成功：

- `PutRecord.Success`
- `PutRecords.Success`
- `GetRecords.Success`

有关更多信息，请参阅 [监控 Kinesis Data Streams](#)。

将 Amazon Kinesis Data Streams 与接口 VPC 端点搭配使用

您可以使用接口 VPC 端点，以防止 Amazon VPC 和 Kinesis Data Streams 之间的流量离开亚马逊网络。接口 VPC 终端节点不需要互联网网关、NAT 设备、VPN 连接或 Amazon Direct Connect 连接。接口 VPC 终端节点由一项 Amazon 技术提供支持 Amazon PrivateLink，该技术允许使用弹性网络接口在 Amazon 服务之间进行私密通信，并使用您的 Amazon VPC IPs 中的私有接口。有关更多信息，请参阅 [亚马逊 Virtual Private Cloud](#) 和 [接口 VPC 终端节点 \(AmazonPrivateLink\)](#)。

主题

- [使用 Kinesis Data Streams 的接口 VPC 端点](#)
- [控制对 Kinesis Data Streams 的 VPC 端点的访问](#)
- [Kinesis Data Streams 的 VPC 端点策略的可用性](#)

使用 Kinesis Data Streams 的接口 VPC 端点

要开始使用，您不需要更改流、产生器或用户的设置。为 Kinesis Data Streams 创建一个接口 VPC 端点，以便进出您的 Amazon VPC 资源的流量流过接口 VPC 端点。启用 FIPS 的接口 VPC 端点适用于美国区域。有关更多信息，请参阅 [创建接口端点](#)。

亚马逊 Kinesis 生产者库 (KPL) 和 Kinesis 消费者库 (KCL) 使用公共终端节点或私有接口 VPC 终端节点（以使用哪个为准）调用 Amazon 亚马逊和亚马逊 CloudWatch DynamoDB 等服务。例如，如果您的 KCL 应用程序在带有启用了 VPC 端点的 DynamoDB 接口的 VPC 中运行，则 DynamoDB 和您的 KCL 应用程序之间的调用会流经接口 VPC 端点。

控制对 Kinesis Data Streams 的 VPC 端点的访问

借助 VPC 端点策略，您可以控制访问，方式是：将策略附加到 VPC 端点或使用附加到 IAM 用户、组或角色的策略中的额外字段，从而限制只能通过特定 VPC 端点进行访问。当与 IAM 策略共同使用以便仅授予通过指定 VPC 端点访问 Kinesis 数据流操作的权限时，这些策略可用于将对特定流的访问限制到指定 VPC 端点。

以下是在访问 Kinesis 数据流的示例端点策略。

- **VPC 策略示例：只读访问** – 此示例策略可以附加到 VPC 端点。（有关更多信息，请参阅[控制对 Amazon VPC 资源的访问](#)）。它限制仅能通过其附加的 VPC 端点列出或描述 Kinesis 数据流。

```
{  
  "Statement": [  
    {  
      "Sid": "ReadOnly",  
      "Principal": "*",  
      "Action": [  
        "kinesis>List*",  
        "kinesis>Describe*"  
      ],  
      "Effect": "Allow",  
      "Resource": "*"  
    }  
  ]  
}
```

- **VPC 策略示例：限制对特定 Kinesis 数据流的访问** – 此示例策略可以附加到 VPC 端点。它限制通过其附加的 VPC 端点访问特定的数据流。

```
{  
  "Statement": [  
    {  
      "Sid": "AccessToSpecificDataStream",  
      "Principal": "*",  
      "Action": "kinesis:*",  
      "Effect": "Allow",  
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream"  
    }  
  ]  
}
```

}

- **IAM 策略示例**：限制只能通过特定的 VPC 端点访问特定流 – 此示例策略可以附加到 IAM 用户、角色或组。它限制只能通过特定的 VPC 端点访问特定的 Kinesis 数据流。

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 的 VPC 端点策略的可用性

以下区域支持带有策略的 Kinesis Data Streams 接口 VPC 端点：

- 欧洲地区 (巴黎)
- 欧洲地区 (爱尔兰)
- 美国东部 (弗吉尼亚州北部)
- 欧洲地区 (斯德哥尔摩)
- 美国东部 (俄亥俄州)
- 欧洲地区 (法兰克福)
- 南美洲 (圣保罗)
- 欧洲地区 (伦敦)
- 亚太地区 (东京)
- 美国西部 (北加利福尼亚)

- 亚太地区 (新加坡)
- 亚太地区 (悉尼)
- 中国 (北京)
- 中国 (宁夏)
- 亚太地区 (香港)
- 中东 (巴林)
- 中东 (阿联酋) :
- 欧洲地区 (米兰)
- 非洲 (开普敦)
- 亚太地区 (孟买)
- 亚太地区 (首尔)
- 加拿大 (中部)
- 美国西部 (俄勒冈州) (usw2-az4 除外)
- AmazonGovCloud (美国东部)
- AmazonGovCloud (美国西部)
- 亚太地区 (大阪)
- 欧洲 (苏黎世)
- 亚太地区 (海得拉巴)

使用 IAM 控制对 Amazon Kinesis Data Streams 资源的访问

Amazon Identity and Access Management(IAM) 使您能够执行以下操作：

- 在您的Amazon账户下创建用户和群组
- 为您Amazon账户下的每位用户分配唯一的安全证书
- 控制每个用户使用Amazon资源执行任务的权限
- 允许其他Amazon账户中的用户共享您的Amazon资源
- 为您的Amazon账户创建角色并定义可以担任这些角色的用户或服务
- 使用企业的现有身份授予使用Amazon资源执行任务的权限

通过将 IAM 与 Kinesis Data Streams 配合使用，您可以控制组织中的用户能否使用特定的 Kinesis Data Streams API 操作执行任务，以及他们能否使用特定的 Amazon 资源。

如果您使用 Kinesis 客户端库 (KCL) 开发应用程序，则您的策略必须包括亚马逊 DynamoDB 和亚马逊的权限；CloudWatchKCL 使用 DynamoDB 来跟踪应用程序的状态信息，并代表您向发送 KCL 指标。CloudWatch CloudWatch有关 KCL 的更多信息，请参阅[开发 KCL 1.x 消费端](#)。

有关 IAM 的更多信息，请参阅以下文档：

- [Amazon Identity and Access Management\(IAM\)](#)
- [IAM 入门](#)
- [IAM 用户指南](#)

有关 IAM 和 AmazonDynamoDB 的更多信息，请参阅《Amazon DynamoDB 开发人员指南》中的 [使用 IAM 控制对 Amazon DynamoDB 资源的访问](#)。

有关 IAM 和 Amazon 的更多信息 CloudWatch，请参阅亚马逊用户指南中的控制 CloudWatch 用户[访问您的Amazon账户](#)。

內容

- [策略语法](#)
- [Kinesis Data Streams 的操作](#)
- [Kinesis Data Streams 的亚马逊资源名称 \(ARNs\)](#)
- [Kinesis Data Streams 的示例策略](#)
- [与其他账户共享您的数据流](#)
- [将Amazon Lambda函数配置为使用另一个账户从 Kinesis Data Streams 读取](#)
- [使用基于资源的策略共享访问权限](#)

策略语法

IAM policy 是包含一个或多个语句的 JSON 文档。每个语句的结构如下：

```
{  
  "Statement": [  
    "Effect": "effect",  
    "Action": "action",  
    "Resource": "arn",  
    "Condition": {  
      "condition": {
```

```
        "key": "value"  
    }  
}  
]  
]
```

组成语句的各个元素如下：

- Effect：此 effect 可以是 Allow 或 Deny。在默认情况下，IAM 用户没有使用资源和 API 操作的许可，因此，所有请求均会被拒绝。显式允许将覆盖默认规则。显式拒绝将覆盖任何允许。
- Action：action 是对其授予或拒绝权限的特定 API 操作。
- Resource：受操作影响的资源。要在语句中指定资源，您需要使用其 Amazon 资源名称（ARN）。
- 条件：条件是可选的。它们可以用于控制策略生效的时间。

在创建和管理 IAM policy 时，您可能希望使用 [IAM Policy 生成器](#) 和 [IAM Policy Simulator](#)。

Kinesis Data Streams 的操作

在 IAM policy 语句中，您可以从支持 IAM 的任何服务中指定任何 API 操作。

对于 Kinesis Data Streams，请使用以下前缀为 API 操作命名：`kinesis:`。

例如：`kinesis:CreateStream`、`kinesis>ListStreams` 和
`kinesis:DescribeStreamSummary`。

要在单个语句中指定多项操作，请使用逗号将它们隔开，如下所示：

```
"Action": ["kinesis:action1", "kinesis:action2"]
```

您也可以使用通配符指定多项操作。例如，您可以指定名称以单词“Get”开头的所有操作，如下所示：

```
"Action": "kinesis:Get*"
```

要指定所有 Kinesis Data Streams 操作，请使用 * 通配符，如下所示：

```
"Action": "kinesis:/*"
```

有关 Kinesis Data Streams API 操作的完整列表，请参阅 [Amazon Kinesis API Reference](#)。

Kinesis Data Streams 的亚马逊资源名称 (ARNs)

每个 IAM 政策声明都适用于您使用其指定的资源 ARNs。

请对 Kinesis Data Streams 使用以下 ARN 资源格式：

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

例如：

```
"Resource": arn:aws:kinesis:*:111122223333:stream/my-stream
```

Kinesis Data Streams 的示例策略

以下示例策略演示如何控制用户对您的 Kinesis Data Streams 的访问。

Example 1: Allow users to get data from a stream

Example

此策略允许用户或组对特定流执行 `DescribeStreamSummary`、`GetShardIterator` 和 `GetRecords` 操作，对任何流执行 `ListStreams` 操作。此策略可应用于应该能够从特定流获取数据的用户。

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": "*"
    }
  ]
}
```

```
        "Action": [
            "kinesis:ListStreams"
        ],
        "Resource": [
            "*"
        ]
    }
]
```

Example 2: Allow users to add data to any stream in the account

Example

此策略允许用户或组对账户的任一流使用 PutRecord 操作。此策略可应用于应该能够向账户中的所有流添加数据记录的用户。

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

此策略允许用户或组对指定流使用任何 Kinesis Data Streams 操作。此策略可应用于应该对特定流有管理控制权限的用户。

JSON

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "kinesis:*",  
            "Resource": [  
                "arn:aws:kinesis:us-east-1:1112223333:stream/stream1"  
            ]  
        }  
    ]  
}
```

Example 4: Allow any Kinesis Data Streams action on any stream

Example

此策略允许用户或组对账户中的任何流使用任何 Kinesis Data Streams 操作。由于此策略会授予对您的所有流的完全访问权限，您应该将其限制为仅对管理员可用。

JSON

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "kinesis:*",  
            "Resource": [  
                "arn:aws:kinesis:*:1112223333:stream/*"  
            ]  
        }  
    ]  
}
```

与其他账户共享您的数据流

Note

Kinesis Producer Library 目前不支持在写入数据流时指定流 ARN。如果您想写入跨账户数据流，请使用 Amazon SDK。

将基于资源的策略附加到您的数据流，以向其他账户、IAM 用户或 IAM 角色授予访问权限。基于资源的策略是附加到资源（例如数据流）的 JSON 策略文档。这些策略将向指定的主体授予对该资源执行特定操作的权限，并定义这在哪些条件下适用。一个策略可以有多个语句。您必须在基于资源的策略中指定主体。委托人可以包括账户、用户、角色、联合用户或Amazon服务。您可通过 Kinesis Data Streams 控制台、API 或 SDK 配置策略。

请注意，要与已注册的使用者（例如增强型扇出功能）共享访问权限，则数据流 ARN 和使用者 ARN 都需要配置策略。

启用跨账户访问

要启用跨账户访问，您可以将整个账户或其他账户中的 IAM 实体指定为基于资源的策略中的主体。将跨账户主体添加到基于资源的策略只是建立信任关系工作的一半而已。当委托人和资源位于不同的 Amazon 账户中时，您还必须使用基于身份的策略来授予委托人访问资源的权限。但是，如果基于资源的策略向同一个账户中的主体授予访问权限，则不需要额外的基于身份的策略。

有关将基于资源的策略用于跨账户存取的更多信息，请参阅 [IAM 中的跨账户资源访问](#)。

数据流管理员可以使用Amazon Identity and Access Management策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。JSON 策略的 Action 元素描述可用于在策略中允许或拒绝访问的操作。策略操作通常与关联的 Amazon API 操作同名。

可以共享的 Kinesis Data Streams 操作：

| Action | 访问级别 |
|--|------|
| DescribeStreamConsumer | 使用者 |
| DescribeStreamSummary | 数据流 |
| GetRecords | 数据流 |

| Action | 访问级别 |
|----------------------------------|------|
| GetShardIterator | 数据流 |
| ListShards | 数据流 |
| PutRecord | 数据流 |
| PutRecords | 数据流 |
| SubscribeToShard | 使用者 |

以下是使用基于资源的策略向您的数据流或注册使用者授予跨账户存取权限的示例。

要执行跨账户操作，您必须指定用于数据流访问的流 ARN，以及用于注册使用者访问的使用者 ARN。

Kinesis Data Streams 基于资源的策略示例

由于需要执行的操作，共享注册的使用者既涉及数据流策略，也涉及使用者策略。

Note

Principal 的示例有效值如下：

- {"AWS": "123456789012"}
- IAM 用户 - {"AWS": "arn:aws:iam::123456789012:user/user-name"}
- IAM 角色 - {"AWS": ["arn:aws:iam::123456789012:role/role-name"]}
- 多个主体 (可以是账户、用户、角色的组合) - {"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:ListStreams"
      ],
      "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
    }
]
```

```
        "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

数据流策略语句：

JSON

```
{
    "Version": "2012-10-17",
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
        {
            "Sid": "consumerreadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::1112222333:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis>ListShards"
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

使用者策略语句：

JSON

```
{  
    "Version": "2012-10-17",  
    "Id": "__default_efs_read_policy_ID",  
    "Statement": [  
        {  
            "Sid": "eforeadstatement",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::1112222333:role/role-name"  
            },  
            "Action": [  
                "kinesis:DescribeStreamConsumer",  
                "kinesis:SubscribeToShard"  
            ],  
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/  
datastreamABC/consumer/consumerDEF:1674696300"  
        }  
    ]  
}
```

为确保遵守最低权限原则，操作或主体字段不支持通配符（*）。

以编程方式管理数据流策略

除此之外Amazon Web Services 管理控制台，Kinesis Data Streams 还有三个 API 用于管理您的数据流策略：

- [PutResourcePolicy](#)
- [GetResourcePolicy](#)
- [DeleteResourcePolicy](#)

[PutResourcePolicy](#) 用于附加或覆盖数据流或使用者策略。[GetResourcePolicy](#) 用于检查和查看指定数据流或使用者的策略。[DeleteResourcePolicy](#) 用于删除指定数据流或使用者的策略。

策略限制

Kinesis Data Streams 资源策略有以下限制：

- 不支持通配符 (*)，以防通过直接附加到数据流或注册消费端的资源策略授予广泛访问权限。此外，还要仔细检查以下策略，来确认它们不会授予广泛访问权限：
 - 附加到关联Amazon委托人（例如，IAM 角色）的基于身份的策略
 - 附加到关联资源的基于Amazon资源的策略（例如 Amazon Key Management Service KMS 密钥）
- Amazon校长不支持服务校长，以防止副校长可能[感到困惑](#)。
- 不支持联合身份验证主体。
- 不支持规范 IDs 用户。
- 策略大小不能超过 20KB。

共享对加密数据的访问权限

如果您已使用Amazon托管 KMS 密钥为数据流启用服务器端加密，并希望通过资源策略共享访问权限，则必须切换到使用客户托管密钥 (CMK)。有关更多信息，请参阅 [什么是 Kinesis Data Streams 的服务器端加密？](#)。此外，您必须允许您的共享主体实体使用 KMS 跨账户共享功能来访问您的 CMK。此外还务必要对共享主体实体的 IAM policy 进行更改。有关更多信息，请参阅[允许其他账户中的用户使用 KMS 密钥](#)。

将Amazon Lambda函数配置为使用另一个账户从 Kinesis Data Streams 读取

有关如何配置 Lambda 函数以在其他账户中读取 Kinesis Data Streams 中的数据的示例，请参阅 [使用跨账户 Amazon Lambda 功能共享访问权限](#)。

使用基于资源的策略共享访问权限

Note

更新现有基于资源的策略意味着替换现有策略，因此新策略中务必要包含所有必要的信息。

使用跨账户Amazon Lambda功能共享访问权限

Lambda 运算符

1. 前往 [IAM 控制台](#) 创建一个 IAM 角色，该角色将用作您的Amazon Lambda函数的 [Lambda 执行角色](#)。添加具有所需的 Kinesis Data Streams 和 Lambda 调用权限的托管 IAM 策略

`AWSLambdaKinesisExecutionRole`。此策略还将授予对您可能有权访问的所有潜在 Kinesis Data Streams 资源的访问权限。

2. 在[Amazon Lambda 控制台](#)中，创建一个Amazon Lambda函数[来处理 Kinesis Data Streams 数据流中的记录](#)，并在设置执行角色的过程中，选择您在上一步中创建的角色。
3. 向 Kinesis Data Streams 资源所有者提供执行角色，以用来配置资源策略。
4. 完成 Lambda 函数设置。

Kinesis Data Streams 资源所有者

1. 获取将调用该 Lambda 函数的跨账户 Lambda 执行角色。
2. 在 Amazon Kinesis Data Streams 控制台上，选择该数据流。选择数据流共享选项卡，然后选择创建共享策略按钮以启动直观的策略编辑器。要共享数据流中的注册使用者，请选择该使用者，然后选择创建共享策略。您也可以直接编写 JSON 策略。
3. 将跨账户 Lambda 执行角色指定为主体，并指定您要共享访问权限的具体 Kinesis Data Streams 操作。务必要包括 `kinesis:DescribeStream` 操作。有关 Kinesis Data Streams 资源策略示例的更多信息，请参阅[Kinesis Data Streams 基于资源的策略示例](#)。
4. 选择创建策略或使用将策略附加[PutResourcePolicy](#)到您的资源。

与跨账户 KCL 消费端共享访问权限

- 如果您使用的是 KCL 1.x，请务必使用 KCL 1.15.0 或更高版本。
- 如果您使用的是 KCL 2.x，请务必使用 KCL 2.5.3 或更高版本。

KCL 运算符

1. 向资源所有者提供将运行 KCL 应用程序的 IAM 用户或 IAM 角色。
2. 要求资源所有者提供数据流或使用者 ARN。
3. KCL 配置中务必要指定所提供的流 ARN。
 - 对于 KCL 1.x：使用[KinesisClientLibConfiguration](#)构造函数并提供流 ARN。
 - 对于 KCL 2.x：你可以只提供直播 ARN 或 Kinesis 客户端[StreamTracker](#)库。[ConfigsBuilder](#)对于 StreamTracker，请提供库生成的 DynamoDB 租赁表中的流 ARN 和创建 Epoch。如果您想从共享注册消费者（如增强型扇出）那里读取数据，请使用 StreamTracker 并提供消费者 ARN。

Kinesis Data Streams 资源所有者

1. 获取将运行 KCL 应用程序的跨账户 IAM 用户或 IAM 角色。
2. 在 Amazon Kinesis Data Streams 控制台上，选择该数据流。选择数据流共享选项卡，然后选择创建共享策略按钮以启动直观的策略编辑器。要共享数据流中的注册使用者，请选择该使用者，然后选择创建共享策略。您也可以直接编写 JSON 策略。
3. 将跨账户 KCL 应用程序的 IAM 用户或 IAM 角色指定为主体，并指定您要共享访问权限的具体 Kinesis Data Streams 操作。有关 Kinesis Data Streams 资源策略示例的更多信息，请参阅 [Kinesis Data Streams 基于资源的策略示例](#)。
4. 选择创建策略或使用将策略附加[PutResourcePolicy](#)到您的资源。

共享对加密数据的访问权限

如果您已使用Amazon托管 KMS 密钥为数据流启用服务器端加密，并希望通过资源策略共享访问权限，则必须切换到使用客户托管密钥 (CMK)。有关更多信息，请参阅 [什么是 Kinesis Data Streams 的服务器端加密？](#)。此外，您必须允许您的共享主体实体使用 KMS 跨账户共享功能来访问您的 CMK。此外还务必要对共享主体实体的 IAM policy 进行更改。有关更多信息，请参阅 [允许其他账户中的用户使用 KMS 密钥](#)。

Amazon Kinesis Data Streams 的合规性验证

作为Amazon多项合规计划的一部分，第三方审计机构评估亚马逊 Kinesis Data Streams 的安全与合规性。其中包括 SOC、PCI、FedRAMP、HIPAA 及其他。

有关特定合规计划范围内的Amazon服务列表，请参阅[合规性计划范围内的Amazon服务](#)。有关一般信息，请参阅 [Amazon合规性计划](#)。

您可以使用下载第三方审计报告Amazon Artifact。有关更多信息，请参阅在 [Artifact 中Amazon 下载报告](#)。

您在使用 Kinesis Data Streams 时的合规性责任由您的数据的敏感性、您公司的合规性目标以及适用的法律法规决定。如果您在使用 Kinesis Data Streams 时必须符合 HIPAA、PCI 或 FedRAMP Amazon 等标准，请提供资源来帮助：

- [安全与合规性快速入门指南](#) — 这些部署指南讨论了架构注意事项，并提供了在上部署以安全性和合规性为重点的基准环境的步骤。Amazon
- [HIPAA 安全与合规架构白皮书 — 本白皮书](#) 描述了公司如何使用Amazon创建符合 HIPAA 标准的应用程序。

- [Amazon合规资源](#) — 此工作簿和指南集合可能适用于您的行业和所在地
- [Amazon Config](#) — 该Amazon服务可评估您的资源配置在多大程度上符合内部实践、行业指导方针和法规。
- [Amazon Security Hub CSPM](#) — 此Amazon服务可全面了解您的安全状态Amazon，帮助您检查是否符合安全行业标准和最佳实践。

Amazon Kinesis Data Streams 中的故障恢复能力

Amazon全球基础设施是围绕Amazon区域和可用区构建的。 Amazon区域提供多个物理隔离和隔离的可用区，这些可用区通过低延迟、高吞吐量和高度冗余的网络相连。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础架构相比，可用区具有更高的可用性、容错性和可扩展性。

有关Amazon区域和可用区的更多信息，请参阅[Amazon全球基础设施](#)。

除了Amazon全球基础架构外，Kinesis Data Streams 还提供多项功能来帮助支持您的数据弹性和备份需求。

Amazon Kinesis Data Streams 中的灾难恢复

故障可能在您使用 Amazon Kinesis Data Streams 应用程序处理流中的数据时在以下级别发生：

- 记录处理器可能失败
- 工作程序可能失败，或者已实例化工作程序的应用程序的实例可能失败
- 托管一个或多个应用程序实例的实例可能会失败 EC2

记录处理器失败

工作器使用 Java [ExecutorService](#)任务调用记录处理器方法。如果任务失败，工作程序将保留对记录处理器之前在处理的分片的控制。工作程序启动一项新的记录处理器任务来处理该分片。有关更多信息，请参阅[读取节流](#)。

工作线程或应用程序失败

如果工作程序或 Amazon Kinesis Data Streams 应用程序的实例出现故障，您应该检测并处理这类情况。例如，如果 `Worker.run` 方法引发了一项异常，则您应捕获并处理它。

如果应用程序本身失败，您应检测此应用程序并重新启动它。该应用程序启动时，它会实例化一个新工作程序，这会反过来实例化自动获得要处理的分片的新记录处理器。这些分配可能是记录处理器在失败前正在处理的相同分片或这些处理器新接触的分片。

如果工作程序或应用程序出现故障，未检测到故障，并且有其他应用程序实例在其他实例上运行，则这些其他 EC2 实例上的工作程序会处理故障。它们将创建额外的记录处理器来处理不再由失败的工作程序处理的分片。这些其他 EC2 实例的负载会相应增加。

此处描述的场景假设，尽管工作程序或应用程序出现故障，但托管 EC2 实例仍在运行，因此未由 Auto Scaling 组重新启动。

Amazon EC2 实例故障

我们建议您在 Auto Scaling 组中运行应用程序的 EC2 实例。这样，如果其中一个 EC2 实例出现故障，Auto Scaling 组会自动启动一个新实例来替换它。您应该将实例配置为在启动时启动您的 Amazon Kinesis Data Streams 应用程序。

Kinesis Data Streams 中的基础设施安全

作为一项托管服务，Amazon Kinesis Data Streams 受 Amazon [亚马逊网络服务：安全流程概述白皮书](#) 中描述的 [全球网络安全程序](#) 的保护。

您可以使用 Amazon 已发布的 API 调用通过网络访问 Kinesis Data Streams。客户端必须支持传输层安全性协议 (TLS) 1.2 或更高版本。客户端还必须支持具有完全向前保密 (PFS) 的密码套件，例如 Ephemeral Diffie-Hellman (DHE) 或 Elliptic Curve Ephemeral Diffie-Hellman (ECDHE)。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 主体关联的秘密访问密钥来对请求进行签名。或者，您可以使用 [Amazon Security Token Service \(Amazon STS\)](#) 生成临时安全凭证来对请求进行签名。

Kinesis Data Streams 的安全最佳实践

Amazon Kinesis Data Streams 提供了在您开发和实施自己的安全策略时需要考虑的大量安全功能。以下最佳实践是一般指导原则，并不代表完整安全解决方案。由于这些最佳实践可能不适合您的环境或不满足您的环境要求，因此将其视为有用的考虑因素而不是惯例。

实施最低权限访问

在授予权限时，您可以决定谁获得哪些 Kinesis Data Streams 资源的哪些权限。您可以对这些资源启用希望允许的特定操作。因此，您应仅授予执行任务所需的权限。实施最低权限访问对于减小安全风险以及可能由错误或恶意意图造成的影响至关重要。

使用 IAM 角色

创建器和客户端应用程序必须具有有效的凭证来访问 Kinesis Data Streams。您不应将Amazon证书直接存储在客户端应用程序或 Amazon S3 存储桶中。这些是不会自动轮换的长期凭证，如果它们受到损害，可能会对业务产生重大影响。

相反，您应该使用 IAM 角色来管理创建器和客户端应用程序的临时凭证以访问 Kinesis 数据流。在使用角色时，您不必使用长期凭证（如用户名和密码或访问密钥）来访问其他资源。

有关更多信息，请参阅 IAM 用户指南中的以下主题：

- [IAM 角色](#)
- [针对角色的常见情形：用户、应用程序和服务](#)

实现从属资源中的服务器端加密

可以在 Kinesis Data Streams 中加密静态数据和传输中数据。有关更多信息，请参阅 [Amazon Kinesis Data Streams 中的数据保护](#)。

CloudTrail 用于监控 API 调用

Kinesis Data Streams Amazon CloudTrail 与一项服务集成，该服务提供用户、角色或Amazon服务在 Kinesis Data Streams 中采取的操作的记录。

使用收集的信息 CloudTrail，您可以确定向 Kinesis Data Streams 发出的请求、发出请求的 IP 地址、谁发出了请求、何时发出请求以及其他详细信息。

有关更多信息，请参阅 [the section called “使用 Amazon CloudTrail 记录 Amazon Kinesis Data Streams API 调用”](#)。

将此服务与 Amazon SDK 配合使用

Amazon 软件开发套件 (SDKs) 可用于许多流行的编程语言。每个软件开发工具包都提供 API、代码示例和文档，使开发人员能够更轻松地以其首选语言构建应用程序。

SDK 文档

[Amazon CLI](#)

[适用于 Java 的 Amazon SDK](#)

[适用于 JavaScript 的 Amazon SDK](#)

[适用于 .NET 的 Amazon SDK](#)

[适用于 PHP 的 Amazon SDK](#)

[Amazon Tools for PowerShell](#)

[适用于 Python \(Boto3\) 的 Amazon SDK](#)

[适用于 Ruby 的 Amazon SDK](#)

[适用于 SAP ABAP 的 Amazon SDK](#)

使用 Kinesis 的代码示例 Amazon SDKs

以下代码示例展示了如何将 Kinesis 与 Amazon 软件开发套件 (SDK) 配合使用。

基本功能是向您展示如何在服务中执行基本操作的代码示例。

操作是大型程序的代码摘录，必须在上下文中运行。您可以通过操作了解如何调用单个服务函数，还可以通过函数相关场景的上下文查看操作。

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

代码示例

- [使用 Kinesis 的基本示例 Amazon SDKs](#)
 - [使用 SDK 学习 Kinesis 的基础知识 Amazon](#)
 - [使用 Kinesis 的操作 Amazon SDKs](#)
 - [AddTagsToStream与 Amazon SDK 或 CLI 配合使用](#)
 - [CreateStream与 Amazon SDK 或 CLI 配合使用](#)
 - [DeleteStream与 Amazon SDK 或 CLI 配合使用](#)
 - [DeregisterStreamConsumer与 Amazon SDK 或 CLI 配合使用](#)
 - [DescribeStream与 Amazon SDK 或 CLI 配合使用](#)
 - [GetRecords与 Amazon SDK 或 CLI 配合使用](#)
 - [将 GetShardIterator 与 CLI 配合使用](#)
 - [与 Amazon SDK ListStreamConsumers 配合使用](#)
 - [ListStreams与 Amazon SDK 或 CLI 配合使用](#)
 - [ListTagsForStream与 Amazon SDK 或 CLI 配合使用](#)
 - [PutRecord与 Amazon SDK 或 CLI 配合使用](#)
 - [PutRecords与 Amazon SDK 或 CLI 配合使用](#)
 - [RegisterStreamConsumer与 Amazon SDK 或 CLI 配合使用](#)
 - [Kinesis 无服务器示例](#)
 - [通过 Kinesis 触发器调用 Lambda 函数](#)
 - [通过 Kinesis 触发器报告 Lambda 函数批处理项目失败](#)

使用 Kinesis 的基本示例 Amazon SDKs

以下代码示例展示了如何使用 Amazon Kinesis 的基础知识。Amazon SDKs

示例

- [使用 SDK 学习 Kinesis 的基础知识 Amazon](#)
- [使用 Kinesis 的操作 Amazon SDKs](#)
 - [AddTagsToStream与 Amazon SDK 或 CLI 配合使用](#)
 - [CreateStream与 Amazon SDK 或 CLI 配合使用](#)
 - [DeleteStream与 Amazon SDK 或 CLI 配合使用](#)
 - [DeregisterStreamConsumer与 Amazon SDK 或 CLI 配合使用](#)
 - [DescribeStream与 Amazon SDK 或 CLI 配合使用](#)
 - [GetRecords与 Amazon SDK 或 CLI 配合使用](#)
 - [将 GetShardIterator 与 CLI 配合使用](#)
 - [与 Amazon SDK ListStreamConsumers 配合使用](#)
 - [ListStreams与 Amazon SDK 或 CLI 配合使用](#)
 - [ListTagsForStream与 Amazon SDK 或 CLI 配合使用](#)
 - [PutRecord与 Amazon SDK 或 CLI 配合使用](#)
 - [PutRecords与 Amazon SDK 或 CLI 配合使用](#)
 - [RegisterStreamConsumer与 Amazon SDK 或 CLI 配合使用](#)

使用 SDK 学习 Kinesis 的基础知识 Amazon

以下代码示例展示了如何：

- 创建流并在其中放置一条记录。
- 创建分片迭代器。
- 读取记录，然后清理函数。

SAP ABAP

适用于 SAP ABAP 的 SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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_knspputrecordoutput.

"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_knskmsoptionrequired.
    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_knsprovthruputexcex.
    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_knsresourcefoundex.
    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_knsprovthruputexcex.
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_knskmsoptionrequired.
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_knsprovthruputexcex.
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.
```

- 有关 API 详细信息，请参阅《Amazon SDK for SAP ABAP API Reference》中的以下主题。
 - [CreateStream](#)
 - [DeleteStream](#)
 - [GetRecords](#)
 - [GetShardIterator](#)
 - [PutRecord](#)

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

使用 Kinesis 的操作 Amazon SDKs

以下代码示例演示了如何使用执行各个 Kinesis 操作。Amazon SDKs 每个示例都包含一个指向的链接 GitHub，您可以在其中找到有关设置和运行代码的说明。

以下示例仅包括最常用的操作。有关完整列表，请参阅[Amazon Kinesis API 参考](#)。

示例

- [AddTagsToStream 与 Amazon SDK 或 CLI 配合使用](#)
- [CreateStream 与 Amazon SDK 或 CLI 配合使用](#)
- [DeleteStream 与 Amazon SDK 或 CLI 配合使用](#)
- [DeregisterStreamConsumer 与 Amazon SDK 或 CLI 配合使用](#)
- [DescribeStream 与 Amazon SDK 或 CLI 配合使用](#)
- [GetRecords 与 Amazon SDK 或 CLI 配合使用](#)
- [将 GetShardIterator 与 CLI 配合使用](#)
- [与 Amazon SDK ListStreamConsumers 配合使用](#)
- [ListStreams 与 Amazon SDK 或 CLI 配合使用](#)

- [ListTagsForStream与 Amazon SDK 或 CLI 配合使用](#)
- [PutRecord与 Amazon SDK 或 CLI 配合使用](#)
- [PutRecords与 Amazon SDK 或 CLI 配合使用](#)
- [RegisterStreamConsumer与 Amazon SDK 或 CLI 配合使用](#)

AddTagsToStream与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 AddTagsToStream。

.NET

适用于 .NET 的 Amazon SDK

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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($"Tags 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;
}
}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[AddTagsToStream](#)中的。

CLI

Amazon CLI

为数据流添加标签

以下 `add-tags-to-stream` 示例将键为 `samplekey` 且值为 `example` 的标签分配给指定流。

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

此命令不生成任何输出。

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[标记流](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[AddTagsToStream](#)中的。

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

CreateStream与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 `CreateStream`。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- [了解基本功能](#)

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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;
    }
}
```

- 有关 API 的详细信息，请参阅 适用于 .NET 的 Amazon SDK API 参考[CreateStream](#)中的。

CLI

Amazon CLI

创建数据流

以下 `create-stream` 示例创建一个名为 `samplestream` 的数据流，其中包含 3 个分片。

```
aws kinesis create-stream \
  --stream-name samplestream \
  --shard-count 3
```

此命令不生成任何输出。

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[创建流](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[CreateStream](#)中的。

Java

适用于 Java 的 SDK 2.x

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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);
    }
}
}
```

- 有关 API 的详细信息，请参阅 Amazon SDK for Java 2.x API 参考[CreateStream](#)中的。

PowerShell

适用于 PowerShell V4 的工具

示例 1：创建新流。默认情况下，此 cmdlet 不返回任何输出，因此添加-`PassThru` 开关以返回提供给-`StreamName` 参数的值以供日后使用。

```
$streamName = New-KINStream -StreamName "mystream" -ShardCount 1 -PassThru
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShell Cmdlet 参考 (V 4) [CreateStream](#)中的。

适用于 PowerShell V5 的工具

示例 1：创建新流。

```
New-KINStream -StreamName "mystream" -ShardCount 1
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 5) [CreateStream](#) 中的。

Python

适用于 Python 的 SDK (Boto3)

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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
```

- 有关 API 的详细信息，请参阅适用于[CreateStream](#)于 Python 的 Amazon SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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(())
}
```

- 有关 API 的详细信息，请参阅适用于 [CreateStream](#) 于 Rust 的 Amazon SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

TRY.

```
lo_kns->creatostream(
    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.
```

- 有关 API 的详细信息，请参阅适用于 [CreateStream](#) 于 SAP 的 Amazon SDK ABAP API 参考。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

DeleteStream 与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 DeleteStream。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- [了解基本功能](#)

.NET

适用于 .NET 的 Amazon SDK

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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;
    }
}
```

- 有关 API 的详细信息，请参阅 适用于 .NET 的 Amazon SDK API 参考[DeleteStream](#)中的。

CLI

Amazon CLI

删除数据流

以下 `delete-stream` 示例删除指定的数据流。

```
aws kinesis delete-stream \
    --stream-name samplestream
```

此命令不生成任何输出。

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[删除流](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[DeleteStream](#)中的。

Java

适用于 Java 的 SDK 2.x

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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);
    }
}
}
```

- 有关 API 的详细信息，请参阅 Amazon SDK for Java 2.x API 参考[DeleteStream](#)中的。

PowerShell

适用于 PowerShell V4 的工具

示例 1：删除指定的流。在命令执行之前，系统会提示您进行确认。要取消确认提示，请使用 `-Force` 开关。

```
Remove-KINStream -StreamName "mystream"
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShell Cmdlet 参考 (V 4) [DeleteStream](#)中的。

适用于 PowerShell V5 的工具

示例 1：删除指定的流。在命令执行之前，系统会提示您进行确认。要取消确认提示，请使用 -Force 开关。

```
Remove-KINStream -StreamName "mystream"
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShell Cmdlet 参考 (V 5) [DeleteStream](#) 中的。

Python

适用于 Python 的 SDK (Boto3)

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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
```

- 有关 API 的详细信息，请参阅适用于[DeleteStream](#)于 Python 的 Amazon SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

 Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
async fn remove_stream(client: &Client, stream: &str) -> Result<(), Error> {  
    client.delete_stream().stream_name(stream).send().await?  
  
    println!("Deleted stream.");  
  
    Ok(())  
}
```

- 有关 API 的详细信息，请参阅适用于[DeleteStream](#)于 Rust 的 Amazon SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

 Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

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.
```

- 有关 API 的详细信息，请参阅适用于 SAP 的 Amazon SDK ABAP API 参考。

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

DeregisterStreamConsumer与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 DeregisterStreamConsumer。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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;  
}  
}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[DeregisterStreamConsumer](#)中的。

CLI

Amazon CLI

注销数据流消费端

以下 `deregister-stream-consumer` 示例注销指定数据流的指定消费端。

```
aws kinesis deregister-stream-consumer \  
--stream-arn arn:aws:kinesis:us-west-2:123456789012:stream/samplestream \  
--consumer-name KinesisConsumerApplication
```

此命令不生成任何输出。

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[使用 Kinesis Data Streams API 开发具有增强型扇出功能的消费端](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[DeregisterStreamConsumer](#)中的。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

DescribeStream与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 `DescribeStream`。

CLI

Amazon CLI

描述数据流

以下 `describe-stream` 示例返回指定数据流的详细信息。

```
aws kinesis describe-stream \
--stream-name samplestream
```

输出：

```
{
  "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
}
}
```

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[创建和管理流](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[DescribeStream](#)中的。

PowerShell

适用于 PowerShell V4 的工具

示例 1：返回指定流的详细信息。

```
Get-KINStream -StreamName "mystream"
```

输出：

```
HasMoreShards      : False
RetentionPeriodHours : 24
Shards            : {}
StreamARN         : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName        : mystream
```

| | |
|--------------|----------|
| StreamStatus | : ACTIVE |
|--------------|----------|

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 4) [DescribeStream](#) 中的。

适用于 PowerShell V5 的工具

示例 1：返回指定流的详细信息。

```
Get-KINStream -StreamName "mystream"
```

输出：

```
HasMoreShards      : False
RetentionPeriodHours : 24
Shards            : {}
StreamARN         : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName        : mystream
StreamStatus       : ACTIVE
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 5) [DescribeStream](#) 中的。

Python

适用于 Python 的 SDK (Boto3)

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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
```

- 有关 API 的详细信息，请参阅适用于 Python 的 Amazon SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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(())
}
```

- 有关 API 的详细信息，请参阅适用于 [DescribeStream](#) 的 Amazon SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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.
```

- 有关 API 的详细信息，请参阅适用于 [DescribeStream](#) 的 Amazon SDK ABAP API 参考。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

GetRecords 与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 GetRecords。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- [了解基本功能](#)

CLI

Amazon CLI

获取分片中的记录

以下 get-records 示例使用指定分片迭代器从 Kinesis 数据流的分片中获取数据记录。

```
aws kinesis get-records \
  --shard-iterator AAAAAAAAAF7/0mWD7IuHj1yGv/
TKuNgx2ukD5xipCY4cy4gU96orlwZwcSXh3K9tAmGYe0ZyLZrvzze0FVf9iN99hUPw/w/
b0YWYeehfNvnf1DYt5XpDJghLKr3DzgznkTmMyDP3R+3wRKeuEw6/kdxY2yKJH0veaiekaVc4N2VwK/
GvaGP2Hh9Fg7N++q0Adg6fIDQPt4p8RpavDbk+A4sL9SWGE1
```

输出：

```
{
  "Records": [],
  "MillisBehindLatest": 80742000
}
```

有关更多信息，请参阅亚马逊 Kinesis Data Streams 开发者指南中的使用 Kinesis Data Streams [API 和 Java 开发工具包开发消费者](#)。Amazon

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[GetRecords](#)中的。

Java

适用于 Java 的 SDK 2.x

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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:
        """

        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()));
}
}
```

- 有关 API 的详细信息，请参阅 Amazon SDK for Java 2.x API 参考[GetRecords](#)中的。

PowerShell

适用于 PowerShell V4 的工具

示例 1：此示例将展示如何从一条或多条记录中返回和提取数据。提供的迭代器 Get-KINRecord 用于确定要返回的记录的起始位置，在本例中，这些记录被捕获到变量 \$records 中。然后，可以通过对 \$records 集合进行索引来访问每条记录。假设记录中的数据是 UTF-8 编码的文本，则最后一个命令显示如何从对象 MemoryStream 中提取数据并将其作为文本返回到控制台。

```
$records
```

```
$records = Get-KINRecord -ShardIterator "AAAAAAAAAGIC....9VnbiRNaP"
```

输出：

| MillisBehindLatest | NextShardIterator | Records |
|--------------------|----------------------------|--------------|
| 0 | AAAAAAAAAERNIq...uDn11HuUs | {Key1, Key2} |

```
$records.Records[0]
```

输出：

| 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())
```

输出：

```
test data from string
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 4) [GetRecords](#) 中的。

适用于 PowerShell V5 的工具

示例 1：此示例将展示如何从一条或多条记录中返回和提取数据。提供的迭代器 Get-KINRecord 用于确定要返回的记录的起始位置，在本例中，这些记录被捕获到变量 \$records 中。然后，可以通过对 \$records 集合进行索引来访问每条记录。假设记录中的数据是 UTF-8 编码的文本，则最后一个命令显示如何从对象 MemoryStream 中提取数据并将其作为文本返回到控制台。

```
$records
$records = Get-KINRecord -ShardIterator "AAAAAAAAAGIC....9VnbiRNaP"
```

输出：

| MillisBehindLatest | NextShardIterator | Records |
|--------------------|-------------------|---------|
|--------------------|-------------------|---------|

0

AAAAAAAAAERNIq...uDn11HuUs {Key1, Key2}

\$records.Records[0]

输出：

| 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())

输出：

test data from string

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShell Cmdlet 参考 (V 5) [GetRecords](#) 中的。

Python

适用于 Python 的 SDK (Boto3)

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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
```

- 有关 API 的详细信息，请参阅适用于 Python 的 Amazon SDK (Boto3) API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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_knskmsoptionrequired.
MESSAGE 'KMS key option is required.' TYPE 'E'.
CATCH /aws1/cx_knskmsrottlingex.
MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
CATCH /aws1/cx_knsprovthruputexcex.
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.
```

- 有关 API 的详细信息，请参阅适用于 SAP 的 Amazon SDK ABAP API 参考。

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

将 **GetShardIterator** 与 CLI 配合使用

以下代码示例演示如何使用 **GetShardIterator**。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- [了解基本功能](#)

CLI

Amazon CLI

获取分片迭代器

以下 `get-shard-iterator` 示例使用 `AT_SEQUENCE_NUMBER` 分片迭代器类型并生成分片迭代器，以便从指定序列号所表示的位置开始精确读取数据记录。

```
aws kinesis get-shard-iterator \
--stream-name samplestream \
--shard-id shardId-000000000001 \
--shard-iterator-type LATEST
```

输出：

```
{  
    "ShardIterator": "AAAAAAAAAAFEvJjIYI+3jw/4aqgH9FifJ+n48XWTh/  
IFIsbILP6o5eDueD39NXNBfpZ10WL5K6ADXk8w+5H+Qhd9cFA9k268CPXCz/kebq1TGYI7Vv  
+lUkA9BuN3xvATxMBGxRY3zYK05gqqvaiRn9408SqeEqwhigwZxNwxD3Ej7YYYcxQi8Q/fIrCjGAy/  
n2r5Z9G864YpWDfn9upNNQAR/iioWks"  
}
```

有关更多信息，请参阅亚马逊 Kinesis Data Streams 开发者指南中的使用 Kinesis Data Streams [API 和 Java 开发工具包开发消费者](#)。Amazon

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[GetShardIterator](#)中的。

PowerShell

适用于 PowerShell V4 的工具

示例 1：返回指定分片和起始位置的分片迭代器。通过引用返回的流对象的 `Shards` 集合，可以从 `Get-KINStream` cmdlet 的输出中获取分片标识符和序列号的详细信息。返回的迭代器可以与 `Get-KINRecord` cmdlet 一起使用，在分片中提取数据记录。

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -  
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

输出：

```
AAAAAAAAAGIc....9VnbirNaP
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 4) [GetShardIterator](#)中的。

适用于 PowerShell V5 的工具

示例 1：返回指定分片和起始位置的分片迭代器。通过引用返回的流对象的 Shards 集合，可以从 Get-KINStream cmdlet 的输出中获取分片标识符和序列号的详细信息。返回的迭代器可以与 Get-KINRecord cmdlet 一起使用，在分片中提取数据记录。

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -  
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

输出：

```
AAAAAAAAAGIC....9VnbiRNnP
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 5) [GetShardIterator](#) 中的。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

与 Amazon SDK **ListStreamConsumers** 配合使用

以下代码示例演示了如何使用 **ListStreamConsumers**。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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;
}

}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[ListStreamConsumers](#)中的。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

ListStreams 与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 ListStreams。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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.");
        }
    }
}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[ListStreams](#)中的。

CLI

Amazon CLI

列出数据流

以下 `list-streams` 示例列出了当前账户和区域中的所有活动数据流。

```
aws kinesis list-streams
```

输出：

```
{
    "StreamNames": [
        "samplestream",
        "samplestream1"
    ]
}
```

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[列出流](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[ListStreams](#)中的。

Rust

适用于 Rust 的 SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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(())
}
```

- 有关 API 的详细信息，请参阅[适用 ListStreams 于 Rust 的 Amazon SDK API 参考](#)。

SAP ABAP

适用于 SAP ABAP 的 SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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_knslimitexceedededex.  
        MESSAGE 'The request processing has failed because of a limit exceed  
exception.' TYPE 'E'.  
    ENDTRY.
```

- 有关 API 的详细信息，请参阅适用于 SAP 的 Amazon SDK ABAP API 参考。

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

ListTagsForStream 与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 ListTagsForStream。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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>
```

```
/// </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}"));  
}  
}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[ListTagsForStream](#)中的。

CLI

Amazon CLI

列出数据流的标签

以下 `list-tags-for-stream` 示例列出附加到指定数据流的标签：

```
aws kinesis list-tags-for-stream \  
--stream-name sampliestream
```

输出：

```
{  
    "Tags": [  
        {  
            "Key": "samplekey",  
            "Value": "example"  
        }  
    ],  
    "HasMoreTags": false  
}
```

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[标记流](#)。

- 有关 API 的详细信息，请参阅 Amazon CLI 命令参考[ListTagsForStream](#)中的。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

PutRecord与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 PutRecord。

操作示例是大型程序的代码摘录，必须在上下文中运行。在以下代码示例中，您可以查看此操作的上下文：

- [了解基本功能](#)

CLI

Amazon CLI

将记录写入数据流

以下 put-record 示例使用指定的分区键将单个数据记录写入指定的数据流。

```
aws kinesis put-record \
  --stream-name samplestream \
  --data sampledatarecord \
  --partition-key samplepartitionkey
```

输出：

```
{  
  "ShardId": "shardId-000000000009",  
  "SequenceNumber": "49600902273357540915989931256901506243878407835297513618",  
  "EncryptionType": "KMS"  
}
```

有关更多信息，请参阅[《亚马逊 Kinesis Data Streams 开发者指南》](#)中的[使用亚马逊 Kinesis Data Streams API 和 Java 开发工具包开发生产者](#)。Amazon

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[PutRecord](#)中的。

Java

适用于 Java 的 SDK 2.x

 Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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);
    }
}
```

- 有关 API 的详细信息，请参阅 Amazon SDK for Java 2.x API 参考[PutRecord](#)中的。

PowerShell

适用于 PowerShell V4 的工具

- 示例 1：写入一条包含提供给 -Text 参数的字符串的记录。

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

- 示例 2：写入包含指定文件所包含数据的记录。该文件被视为字节序列，因此，如果它包含文本，则在将其与此 cmdlet 搭配使用之前，应使用任何必要的编码进行编写。

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
"Key2"
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShell Cmdlet 参考 (V 4) [PutRecord](#) 中的。

适用于 PowerShell V5 的工具

- 示例 1：写入一条包含提供给 -Text 参数的字符串的记录。

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

- 示例 2：写入包含指定文件所包含数据的记录。该文件被视为字节序列，因此，如果它包含文本，则在将其与此 cmdlet 搭配使用之前，应使用任何必要的编码进行编写。

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
"Key2"
```

- 有关 API 的详细信息，请参阅 Amazon Tools for PowerShellCmdlet 参考 (V 5) [PutRecord](#) 中的。

Python

适用于 Python 的 SDK (Boto3)

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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
```

- 有关 API 的详细信息，请参阅适用于 [PutRecord](#) 的 Amazon SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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(())
}
```

- 有关 API 的详细信息，请参阅适用于 [PutRecord](#) 的 Amazon SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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.
```

- 有关 API 的详细信息，请参阅适用于 SAP 的 Amazon SDK ABAP API 参考。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

PutRecords 与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 PutRecords。

CLI

Amazon CLI

将多条记录写入数据流中

以下 put-records 示例在一次调用中使用指定的分区键写入一条数据记录，并使用不同的分区键写入另一条数据记录。

```
aws kinesis put-records \
  --stream-name samplestream \
  --
  records Data=blob1,PartitionKey=partitionkey1 Data=blob2,PartitionKey=partitionkey2
```

输出：

```
{  
  "FailedRecordCount": 0,  
  "Records": [  
    {  
      "SequenceNumber":  
        "49600883331171471519674795588238531498465399900093808706",  
      "ShardId": "shardId-000000000004"  
    },  
    {  
      "SequenceNumber":  
        "49600902273357540915989931256902715169698037101720764562",  
      "ShardId": "shardId-000000000009"  
    }  
  ],  
  "EncryptionType": "KMS"  
}
```

有关更多信息，请参阅[《亚马逊 Kinesis Data Streams 开发者指南》中的使用亚马逊 Kinesis Data Streams API 和 Java 开发工具包开发者](#)。Amazon

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[PutRecords](#)中的。

JavaScript

适用于 JavaScript (v3) 的软件开发工具包

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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);
}
```

- 有关 API 的详细信息，请参阅适用于 JavaScript 的 Amazon SDK API 参考[PutRecords](#)中的。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

RegisterStreamConsumer与 Amazon SDK 或 CLI 配合使用

以下代码示例演示如何使用 RegisterStreamConsumer。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在 [Amazon 代码示例存储库](#) 中查找完整示例，了解如何进行设置和运行。

```
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;
}
```

- 有关 API 的详细信息，请参阅适用于 .NET 的 Amazon SDK API 参考[RegisterStreamConsumer](#)中的。

CLI

Amazon CLI

注册数据流消费端

以下 `register-stream-consumer` 示例将名为 `KinesisConsumerApplication` 的消费端注册到指定数据流。

```
aws kinesis register-stream-consumer \
--stream-arn arn:aws:kinesis:us-west-2:012345678912:stream/samplestream \
--consumer-name KinesisConsumerApplication
```

输出：

```
{  
    "Consumer": {
```

```
        "ConsumerName": "KinesisConsumerApplication",
        "ConsumerARN": "arn:aws:kinesis:us-west-2: 123456789012:stream/
samplestream/consumer/KinesisConsumerApplication:1572383852",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1572383852.0
    }
}
```

有关更多信息，请参阅《Amazon Kinesis Data Streams 开发人员指南》中的[使用 Kinesis Data Streams API 开发具有增强型扇出功能的消费端](#)。

- 有关 API 的详细信息，请参阅Amazon CLI命令参考[RegisterStreamConsumer](#)中的。

SAP ABAP

适用于 SAP ABAP 的 SDK

Note

还有更多相关信息 GitHub。在[Amazon 代码示例存储库](#)中查找完整示例，了解如何进行设置和运行。

```
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.
```

- 有关 API 的详细信息，请参阅适用于 SAP 的 Amazon SDK ABAP API 参考。

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

Kinesis 无服务器示例

以下代码示例展示了如何将 Kinesis 与配合使用。Amazon SDKs

示例

- [通过 Kinesis 触发器调用 Lambda 函数](#)
- [通过 Kinesis 触发器报告 Lambda 函数批处理项目失败](#)

通过 Kinesis 触发器调用 Lambda 函数

以下代码示例演示了如何实现一个 Lambda 函数，该函数接收通过接收来自 Kinesis 流的记录而触发的事件。该函数检索 Kinesis 有效负载，将 Base64 解码，并记录下记录内容。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

通过 .NET 将 Kinesis 事件与 Lambda 结合使用。

```
// 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

适用于 Go 的 SDK V2

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

使用 Go 将 Kinesis 事件与 Lambda 结合使用。

```
// 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

适用于 Java 的 SDK 2.x

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

使用 Java 将 Kinesis 事件与 Lambda 结合使用。

```
// 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

适用于 JavaScript (v3) 的软件开发工具包

Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

使用 Lambda 消耗 Kinesis 事件。JavaScript

```
// 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;
}
```

使用 Lambda 消耗 Kinesis 事件。TypeScript

```
// 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

适用于 PHP 的 SDK

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

通过 PHP 将 Kinesis 事件与 Lambda 结合使用。

```
// 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

适用于 Python 的 SDK (Boto3)

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

使用 Python 将 Kinesis 事件与 Lambda 结合使用。

```
# 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

适用于 Ruby 的 SDK

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

通过 Ruby 将 Kinesis 事件与 Lambda 结合使用。

```
# 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}"
        end
    end
end
```

```
        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

适用于 Rust 的 SDK

 Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

通过 Rust 将 Kinesis 事件与 Lambda 结合使用。

```
// 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
}
```

有关 S Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

通过 Kinesis 触发器报告 Lambda 函数批处理项目失败

以下代码示例显示如何为接收来自 Kinesis 流的事件的 Lambda 函数实现部分批处理响应。该函数在响应中报告批处理项目失败，并指示 Lambda 稍后重试这些消息。

.NET

适用于 .NET 的 Amazon SDK

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告使用 .NET 进行 Lambda Kinesis 批处理项目失败。

```
// 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.DefaultLambdaJsonSerializer))]

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

适用于 Go 的 SDK V2

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告通过 Go 进行 Lambda Kinesis 批处理项目失败。

```
// 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

适用于 Java 的 SDK 2.x

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

报告使用 Java 进行 Lambda Kinesis 批处理项目失败。

```
// 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

适用于 JavaScript (v3) 的软件开发工具包

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告使用 Javascript 进行 Lambda Kinesis 批处理项目失败。

```
// 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;
}
```

使用 Lambda 报告 Kinesis 批处理项目失败。TypeScript

```
// 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

适用于 PHP 的 SDK

 Note

还有更多相关信息 GitHub。在[无服务器示例](#)存储库中查找完整示例，并了解如何进行设置和运行。

报告通过 PHP 进行 Lambda Kinesis 批处理项目失败。

```
// 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

适用于 Python 的 SDK (Boto3)

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告使用 Python 进行 Lambda Kinesis 批处理项目失败。

```
# 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

适用于 Ruby 的 SDK

Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告通过 Ruby 进行 Lambda Kinesis 批处理项目失败。

```
# 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

适用于 Rust 的 SDK

 Note

还有更多相关信息 GitHub。在[无服务器示例存储库](#)中查找完整示例，并了解如何进行设置和运行。

报告通过 Rust 进行 Lambda Kinesis 批处理项目失败。

```
// 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
}
```

有关 Amazon DK 开发者指南和代码示例的完整列表，请参阅[将此服务与 Amazon SDK 配合使用](#)。本主题还包括有关入门的信息以及有关先前的 SDK 版本的详细信息。

文档历史记录

下表介绍了对 Amazon Kinesis Data Streams 文档所做的重要更改。

| 更改 | 描述 | 更改日期 |
|---|--|------------------|
| 增加了对弹性测试的支持 Amazon Fault Injection Service。 | 增加了 使用执行弹性测试 Amazon Fault Injection Service 。 | 2025 年 10 月 15 日 |
| 添加了对高达 10 MiB 的大型记录的支持。 | 增加了 处理大型记录 。 | 2025 年 10 月 27 日 |
| End-of-support KPL 和 KCL 的日期和版本生命周期政策。 | 添加了有关亚马逊 Kinesis 制作者库 (KPL) 和亚马逊 Kinesis 客户端库 (KCL) 的 end-of-support 日期和版本生命周期策略的信息。有关更多信息，请参阅 KPL 版本生命周期策略 和 KCL 版本生命周期策略 。 | 2025 年 3 月 13 日 |
| 增加了对跨账户共享数据流的支持。 | 增加了 与其他账户共享您的数据流 。 | 2023 年 11 月 22 日 |
| 增加了对按需和预置数据流容量模式的支持。 | 增加了 选择正确的流式传入的模式 。 | 2021 年 11 月 29 日 |
| 新增了服务器端加密的相关内容 | 增加了 Amazon Kinesis Data Streams 中的数据保护 。 | 2017 年 7 月 7 日 |
| 增强 CloudWatch 指标的新内容。 | 已更新 监控 Kinesis Data Streams 。 | 2016 年 4 月 19 日 |
| 新增有关 Kinesis 增强代理的内容。 | 已更新 使用 Kinesis 代理写入 Amazon Kinesis Data Streams 。 | 2016 年 4 月 11 日 |

| 更改 | 描述 | 更改日期 |
|-----------------------|---|------------------|
| 新增有关使用 Kinesis 代理的内容。 | 增加了 使用 Kinesis 代理写入 Amazon Kinesis Data Streams 。 | 2015 年 10 月 2 日 |
| 更新 0.10.0 版的 KPL 内容。 | 增加了 使用 Amazon Kinesis Producer Library (KPL) 开发产生器 。 | 2015 年 7 月 15 日 |
| 更新可配置指标的 KCL 指标主题。 | 增加了 使用 Amazon CloudWatch 监控 Kinesis Client Library 。 | 2015 年 7 月 9 日 |
| 重新组织了内容。 | 为提供更简洁的树视图和更有逻辑的分组，内容主题经过了大幅地重新组织。 | 2015 年 7 月 01 日 |
| “新 KPL 开发人员的指南”主题。 | 增加了 使用 Amazon Kinesis Producer Library (KPL) 开发产生器 。 | 2015 年 6 月 02 日 |
| “新 KCL 指标”主题。 | 增加了 使用 Amazon CloudWatch 监控 Kinesis Client Library 。 | 2015 年 5 月 19 日 |
| 对 KCL .NET 的支持 | 增加了 在 .NET 中开发 Kinesis Client Library 消费端 。 | 2015 年 5 月 1 日 |
| 对 KCL Node.js 的支持 | 增加了 在 Node.js 中开发 Kinesis Client Library 消费端 。 | 2015 年 3 月 26 日 |
| 对 KCL Ruby 的支持 | 添加了指向 KCL Ruby 库的链接。 | 2015 年 1 月 12 日 |
| 新 PutRecords API | 向添加了有关新 PutRecords API 的信息 the section called “使用 PutRecords 添加多条记录” 。 | 2014 年 12 月 15 日 |
| 对标签的支持 | 增加了 标记 Amazon Kinesis Data Streams 资源 。 | 2014 年 9 月 11 日 |
| 新 CloudWatch 指标 | 向 <code>GetRecords.IteratorAgeMilliseconds</code> 添加了指标 Amazon Kinesis Data Streams 维度和指标 。 | 2014 年 9 月 3 日 |

| 更改 | 描述 | 更改日期 |
|----------|--|------------------|
| 新的监控章节 | 添加了 监控 Kinesis Data Streams 和 使用 Amazon CloudWatch 监控 Amazon Kinesis Data Streams 服务 。 | 2014 年 7 月 30 日 |
| 默认分片限制 | 更新了 限额和限制 ：默认分片限制从 5 个提高到 10 个。 | 2014 年 2 月 25 日 |
| 默认分片限制 | 更新了 限额和限制 ：默认分片限制从 2 个提高到 5 个。 | 2014 年 1 月 28 日 |
| API 版本更新 | 对 Kinesis Data Streams API 版本 2013-12-02 进行了更新。 | 2013 年 12 月 12 日 |
| 初始版本 | 发布《Amazon Kinesis Developer Guide》初始版本。 | 2013 年 11 月 14 日 |

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。