aws

开发人员指南

Amazon Kinesis Data Streams



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

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。

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
Server-Side 加密 6
数据交付 6
限额和限制 8
API 限制 10
KDS 控制层面 API 限制 10
数据层面 API 限制 14
提升配额 15
完成设置 Amazon Kinesis Data Streams 的先决条件 17
注册参加 AWS 账户 17
下载库和工具 17
配置您的开发环境 18
使用 AWS CLI 来执行 Amazon Kinesis Data Streams 操作 19
教程：安装和配置 AWS CLI 适用于 Kinesis Data Streams 19
安装 AWS CLI 19
配置 AWS CLI 20

教程：使用 Kinesis Data Streams 执行基本操作 AWS CLI 21
第 1 步：创建流 21
步骤 2：放置记录 23
步骤 3：获取记录 23
步骤 4：清除 26
入门教程 27
教程：使用 KPL 和 KCL 2.x 处理实时股票数据 27
满足先决条件 28
创建数据流 29
创建 IAM 策略和用户 29
下载并构建代码 34
实现产生器 35
实现消费端 39
（可选）扩展消费端 43
清理 资源 45
教程：使用 KPL 和 KCL 1.x 处理实时股票数据 46
满足先决条件 47
创建数据流 48
创建 IAM 策略和用户 49
下载和构建实现代码 54
实现产生器 55
实现消费端 59
（可选）扩展消费端 62
清理 资源 63
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 65
先决条件 66
第 1 步：设置 AWS CLI 66
步骤 2：创建应用程序 67
教程： AWS Lambda 与 Amazon Kinesis Data Streams 配合使用 83
使用适用于 Amazon Kinesis 的 AWS 流数据解决方案 84
创建和管理 Kinesis 数据流 85
选择正确的流式传入的模式 85
Kinesis Data Streams 中有哪些不同的模式？ 86
On-demand 标准模式的功能和用例 86
On-demand 优势模式的功能和用例 88
预置模式的特征和用例 88

切换模式 89
使用创建直播 AWS 管理控制台 90
使用 API 创建流 90
构建 Kinesis Data Streams 客户端 90
创建流 91
更新流 92
使用控制台 93
使用 API 94
使用 AWS CLI 94
列出流 95
列出分片 96
删除流 99
对流进行重新分片 99
确定重新分片策略 100
拆分分片 101
合并两个分片 102
完成重新分片操作 103
更改数据留存期 105
标记资源 106
查看标签基本知识 106
使用标签跟踪成本 107
了解标签限制 108
使用 Kinesis Data Streams 控制台标记流 108
使用标记直播 AWS CLI 110
使用 Kinesis Data Streams 标记直播 APIs 110
使用标记消费者 AWS CLI 111
使用 Kinesis Data Streams 标记消费者 APIs 111
处理大型记录 112
大型记录的工作原理 112
更新您的流以使用大型记录 113
使用大型记录优化流性能 113
使用大型记录缓解限制 113
使用 Kinesis Data Streams API 处理大型记录 114
AWS 与大型记录兼容的组件 114
支持大型记录的区域 117
使用执行弹性测试 AWS Fault Injection Service 118

预置吞吐量错误 119
迭代器过期异常错误 122
Service-managed 记录分发 124
服务管理记录分发的工作原理 124
何时使用服务管理的记录分发 124
唱片分发策略比较 125
过渡行为 126
重要注意事项 126
配置服务托管分发 127
先决条件 127
使用服务管理的记录分发创建直播 128
在现有直播上启用服务管理的记录分发 128
验证当前记录分发策略 129
在控制台中配置记录分发策略 130
切换回用户管理的分区键 131
安全的迁移模式 131
错误处理 133
IAM 权限 133
将记录写入服务管理的流 134
使用服务托管分发的生产者 API 行为 134
示例 135
错误处理 136
迁移现有生产者 136
最佳实践 137
从服务管理的直播中读取记录 137
GetRecords API 行为 137
SubscribeToShard API 行为 138
使用者注意事项 139
监控服务管理直播的使用情况 139
KPL 和 KCL 的兼容性 139
升级 KPL 以发送空分区密钥 140
向 KP DescribeStreamSummary L 授予许可 140
现有生产者和消费者无需进行任何更改 140
服务管理的流不支持聚合 141
KCL 注意事项 141
Summary 142

整合案例研究 142
AWS Lambda 使用服务管理流触发器 142
Amazon Data Firehose 从服务管理的流中消费 143
适用于 Apache Flink 的亚马逊托管服务 Flink 从服务管理的直播中消费 143
CloudWatch 带有服务管理流的指标流 143
CloudFront 服务管理流的实时日志 144
带有服务管理流的 Amazon EMR 144
带有服务托 EventBridge 管直播的 Amazon Pipes 144
带有 Kinesis 数据流的亚马逊 DynamoDB 145
从服务管理的直播中提取亚马逊 Redshift 流媒体 145
带有服务托管直播的 Amazon Kinesis 代理 145
带有服务管理流的 Amazon EventBridge 事件总线 145
使用服务管理的流式传输表和 Amazon S3 交付 145
将数据写入 Kinesis Data Streams 146
使用 Amazon Kinesis Producer Library（KPL）开发产生器 146
查看 KPL 的作用 147
了解使用 KPL 的优势 148
了解何时不使用 KPL 149
安装 KPL 149
迁移至 KPL 1.x 150
转换为适用于 KPL 的 Amazon Trust Services（ATS）证书 154
KPL 受支持平台 154
KPL 的重要概念 154
将 KPL 与产生器代码集成 157
使用 KPL 写入 Kinesis Data Streams 158
配置 KPL 160
实现消费端取消聚合 161
将 KPL 与 Amazon Data Firehose 搭配使用 164
将 KPL 与 AWS Glue 架构注册表一起使用 164
配置 KPL 代理配置 164
KPL 版本生命周期策略 165
使用 Kinesis Data Streams API 开发制作人 适用于 Java 的 AWS SDK 166
向流添加数据 166
使用 AWS Glue 架构注册表与数据交互 172
使用 Kinesis 代理写入 Amazon Kinesis Data Streams 172
完成 Kinesis 代理的先决条件 173

下载并安装代理 174
配置并启动代理 175
指定代理配置设置 175
监控多个文件目录并写入多个流 178
使用代理预处理数据 179
使用代理 CLI 命令 183
常见问题解答 184
使用其他服务写入 Kinesis Data Streams AWS 185
使用写入 Kinesis Data Streams AWS Amplify 185
使用 Amazon Aurora 写入 Kinesis Data Streams 186
使用亚马逊写入 Kinesis Data Streams CloudFront 186
使用亚马逊 CloudWatch 日志写入 Kinesis Data Streams 186
使用 Connect Customer 写入 Kinesis Data Streams 186
使用写入 Kinesis Data Streams AWS Database Migration Service 187
使用 Amazon DynamoDB 写入 Kinesis Data Streams 187
使用亚马逊写入 Kinesis Data Streams EventBridge 187
使用写入 Kinesis Data Streams AWS IoT Core 187
使用 Amazon Relational Database Service 写入 Kinesis Data Streams 187
使用 Amazon Pinpoint 写入 Kinesis Data Streams 188
使用 Amazon Quantum Ledger Database（Amazon QLDB）写入 Kinesis Data Streams 188
使用第三方集成写入 Kinesis Data Streams 188
Apache Flink 189
Fluentd 189
Debezium 189
甲骨文 GoldenGate 189
Kafka Connect 189
Adobe Experience 189
Striim 189
Kinesis Data Streams 产生器问题排查 190
产生器应用程序的写入速率比预期的慢 190
我收到了未授权的 KMS 主密钥权限错误 192
排查产生器的其他常见问题 192
优化 Kinesis Data Streams 产生器 192
自定义 KPL 重试和速率限制行为 192
将最佳实践应用于 KPL 聚合 193
从 Kinesis Data Streams 读取数据 195

开发具有专用吞吐量的增强扇出型消费端 195
共享吞吐量消费端和增强扇出型消费端之间的区别 197
支持最多 50 个增强型扇出消费者的区域（仅限 On-demand Advantage） 117
管理具有增强扇出功能的消费端 200
在 Kinesis 控制台中使用数据查看器 201
在 Kinesis 控制台中查询您的数据流 202
使用 Kinesis Client Library 202
什么是 Kinesis Client Library？ 202
KCL 主要功能和优势 203
KCL 概念 203
KCL 中的 DynamoDB 元数据表和负载均衡 204
使用 KCL 开发消费端 208
Multi-stream 使用 KCL 进行处理 218
使用 AWS Glue 使用 KCL 的架构注册表 221
KCL 消费端应用程序所必需的 IAM 权限 221
KCL 配置 227
KCL 版本生命周期策略 238
KCL 的单表格式 238
从之前的 KCL 版本迁移 244
先前的 KCL 版本文档 262
通过以下方式培养消费者 适用于 Java 的 AWS SDK 335
使用开发吞吐量共享的消费者 适用于 Java 的 AWS SDK 336
通过以下方式培养更具吸引力的粉丝消费者 适用于 Java 的 AWS SDK 341
使用 AWS Glue 架构注册表与数据交互 343
使用 AWS Lambda开发消费端 344
使用适用于 Apache Flink 的托管服务来开发消费端 344
使用 Amazon Data Firehose 开发消费端 344
使用其他 AWS 服务从 Kinesis Data Streams 读取数据 344
使用 Amazon EMR 从 Kinesis Data Streams 读取数据 345
使用 Amazon EventBridge Pipes 从 Kinesis Data Streams 读取数据 345
使用从 Kinesis Data Streams 读取数据 AWS Glue 345
使用 Amazon Redshift 从 Kinesis Data Streams 读取数据 345
使用第三方集成从 Kinesis Data Streams 中读取 346
Apache Flink 346
Adobe Experience Platform 346
Apache Druid 346

Apache Spark 347
Databricks 347
Kafka Confluent Platform 347
Kinesumer 347
Talend 347
Kinesis Data Streams 消费端问题排查 347
LeaseManagementConfig 构造函数出现编译错误 348
使用 Kinesis Client Library 时会跳过某些 Kinesis Data Streams 记录 349
属于同一分片的记录通过不同的记录处理器同时处理 349
消费端应用程序的读取速率比预期的慢 350
GetRecords 即使流中有数据，也会返回一个空的记录数组 350
分片迭代器意外过期 351
消费端记录处理滞后 351
未授权的 KMS 密钥权限错误 352
DynamoDbException: 更新表达式中提供的文档路径无效，无法更新 352
排查消费端的其他常见问题 352
优化 Kinesis Data Streams 消费端 353
改进低延迟处理 353
使用 Amazon Kinesis AWS Lambda 制作器库处理序列化数据 354
使用重新分片、扩展和并行处理更改分片数量 354
处理重复记录 356
处理启动、关闭和节流 357
流式传输表和 S3 交付 360
数据交付的工作原理 360
送货目的地 361
数据流 361
关键功能 362
要求 363
数据交付概念 363
数据新鲜度 363
直播表的输入格式 363
通用 Amazon S3 存储桶的输入格式 364
架构执行 364
没有回填 364
每次配送都有新桌子 364
Dead-letter 队列 364

内联压缩 365
配送状态 365
Append-only 交货 365
直播表 365
工作原理 366
开始使用 367
管理直播表 369
直播桌的冰山行为 375
S3 一般用途交付 378
工作原理 378
开始使用 381
管理 S3 交付 382
用于 Amazon S3 交付的 S3 输出密钥模板 387
用于数据传输的 IAM 权限 390
生命周期管理权限 390
服务执行角色 391
数据传输的安全性 399
传输中加密 400
静态加密 400
源流加密 401
IAM 访问控制 401
AWS CloudTrail 日志记录 402
监控数据交付 402
CloudWatch 指标 402
CloudWatch 日志 405
AWS CloudTrail 日志记录 406
数据交付的最佳实践 407
吞吐量和数据新鲜度 407
架构管理（Apache Iceberg 上的流式传输表） 408
S3 对象布局（通用亚马逊 S3 存储桶） 408
安全性 408
Dead-letter 队列 408
监控 409
数据传输疑难解答 409
交付停留在 “创建” 状态 409
交付处于失败状态 410

数据新鲜度高 410
失败记录大于零 410
目的地没有显示任何数据 411
配送已暂停 411
CloudWatch 日志中出现权限被拒绝错误 411
我的直播无法配送 412
架构更改后交付停止 412
无法删除直播 412
配送配额和限制 412
资源配额 412
API 限制 413
交付限制 413
更新限制 414
监控 Kinesis Data Streams 415
使用监控 Kinesis Data Streams 服务 CloudWatch 415
Amazon Kinesis Data Streams 维度和指标 416
访问 Kinesis Data Streams 的亚马逊 CloudWatch 指标 430
使用监控 Kinesis Data Streams 代理的生命值 CloudWatch 431
使用监视器 CloudWatch 431
使用记录亚马逊 Kinesis Data Streams API 调用 AWS CloudTrail 432
Kinesis Data Streams 将信息流入 CloudTrail 432
示例：Kinesis Data Firehose 日志文件条目 434
使用监控 KCL CloudWatch 438
指标和命名空间 438
指标级别和维度 438
指标配置 439
指标列表 439
使用以下方式监控 KPL CloudWatch 455
指标、维度和命名空间 456
指标级别和粒度 456
本地访问和 Amazon CloudWatch 上传 457
指标列表 457
安全性 461
Kinesis Data Streams 中的数据保护 461
什么是 Kinesis Data Streams 的服务器端加密？ 462
费用、区域和性能注意事项 463

如何开始使用服务器端加密？ 464
创建和使用用户生成的 KMS 密钥 465
使用用户生成的 KMS 密钥的权限 466
验证 KMS 密钥权限并排查相关问题 468
将 Kinesis Data Streams 与接口 VPC 端点搭配使用 468
使用 IAM 控制对 Kinesis Data Streams 资源的访问 471
策略语法 473
Kinesis Data Streams 的操作 473
Kinesis Data Streams 的 Amazon 资源名称（ARN） 474
Kinesis Data Streams 的示例策略 474
与其他账户共享您的数据流 477
配置一个 AWS Lambda 使用另一个账户从 Kinesis Data Streams 读取的函数 482
使用基于资源的策略共享访问权限 482
流式传输表和 Amazon S3 交付的访问控制 484
使用试运行进行测试 485
支持的 API 486
AWS CloudTrail 日志记录 486
使用 dry run 和 AWS CLI 486
使用 dry run 和 AWS 软件开发工具包 487
合规性验证 488
Kinesis Data Streams 中的韧性 488
Kinesis Data Streams 中的灾难恢复 488
基础设施安全性 489
Kinesis Data Streams 的安全最佳实践 490
实施最低权限访问 490
使用 IAM 角色 490
实现从属资源中的服务器端加密 490
CloudTrail 用于监控 API 调用 490
使用 AWS 软件开发工具包 492
代码示例 494
基本功能 495
了解基本功能 495
操作 499
无服务器示例 556
通过 Kinesis 触发器调用 Lambda 函数 556
通过 Kinesis 触发器报告 Lambda 函数批处理项目失败 566

文档历史记录 581
..... dlxxxv

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。这些应用程序可以使用 Kinesis 客户端库，也可以在亚马逊 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 AWS 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 AWS 大数据解决方案的更多信息，请参阅上的 Big Data AWS。有关 AWS 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

您可以创建 Kinesis Data Streams 应用程序和数据流的有向无环图 (DAGs)。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。将记录放入流中的时间与可以检索的时间（put-to-get 延迟）之间的延迟通常小于 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，控制面板会读取包含正在运行的聚合的 DynamoDB 表以获取报告。 up-to-the-minute

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

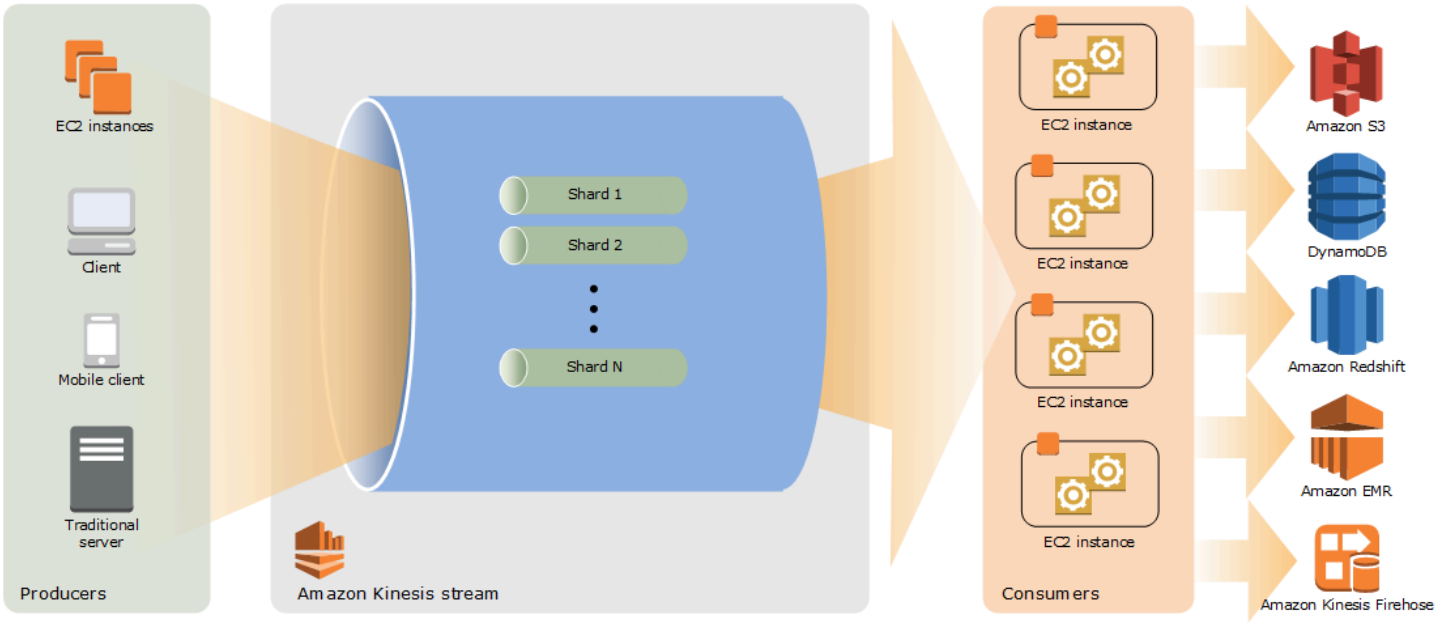
开始使用 Amazon Kinesis Data Streams 之前，有必要了解其架构和术语。

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。消费者（例如在亚马逊 EC2 上运行的自定义应用程序或亚马逊 Data Firehose 交付流）可以使用亚马逊 DynamoDB、亚马逊 Redshift 或 Amazon S3 等 AWS 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 On-demand 标准模式的功能和用例。

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。您可以使用该操作将保留期延长至最多 8760 小时（365 天），使用该IncreaseStreamRetentionPeriod操作可以将保留期缩短至至少 24 小时。DecreaseStreamRetentionPeriod 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。应用程序还可以向各种其他 AWS 服务发送数据。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。您的每个应用程序都必须有一个唯一的名称，该名称仅限于应用程序使用的 AWS 账户和区域。此名称用作亚马逊 DynamoDB 中控制表的名称和亚马逊指标的命名空间。 CloudWatch

Server-Side 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 AWS 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**
>
> 使用服务器端加密会产生 AWS Key Management Service (AWS KMS) 成本。有关更多信息，请参阅 AWS Key Management Service Pricing。

数据交付

数据交付是一种完全托管的功能，无需管理任何基础架构，即可将流数据从 Kinesis 数据流传输到分析或存储目的地。没有连接器、消费类应用程序或计算资源可供配置。Kinesis Data Streams 从您的数据流、缓冲和聚合记录中读取数据，并在几分钟内将其传送到您配置的目的地。数据传输支持两种目标

类型：Apache Iceberg 上的流媒体表和通用的 Amazon S3 存储桶。有关更多信息，请参阅 Amazon Kinesis Data Streams 的流式传输表和 S3 交付。

以 Iceberg 格式传送到直播桌

在不需要 ETL 管道的情况下，将流式数据作为可查询的 Apache Iceberg 表持续体现在 Amazon S3 表上。向流媒体表的交付会将流记录作为Apache Iceberg表在Amazon S3表上持续体现。 Built-in 智能内联压缩可消除小文件扩散，并在不牺牲数据新鲜度的前提下保持查询性能的可预测性。使用任何引擎（包括Apache Spark、Trino、Apache Flink或亚马逊雅典娜）查询或转换数据，直接对流数据进行实时分析。

传输到 Amazon S3

将源格式的流数据传输到 Amazon S3 存储桶，用于备份、存档和下游数据处理。传输到 Amazon S3 将流媒体记录以原始源格式直接传输到通用的 Amazon S3 存储桶。将此功能用于长期数据保留、合规性存档、重放和再处理工作流程，或用作下游批处理管道的过渡层。结合向流媒体表的交付，您可以构建一个完整的湖库架构，其中包含用于分析的实时查询表以及 Amazon S3 中的全保真原始备份。

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	On-demand 模式	预置模式
数据流数	您 AWS 账号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。	并无上限。对于以下情况，默认分片配额 AWS 账户 为每个 20,000 个分片： AWS 区域 • 美国东部（弗吉尼亚北部） • 美国西部（俄勒冈） • 欧洲地区（爱尔兰） 其余所有区域每个 AWS 账户账户的默认分片限额为 1000 或 6000 个分片。您可以通过服务配额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/。 要请求增加分片配额，请使用服务配额控制台或 AWS CLI。有关更多信息，请参阅请求提高限额。

配额	On-demand 模式	预置模式
数据流吞吐量	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） AWS 区域，按需容量模式的数据流最多可扩展到写入吞吐量的 10% 和 20% GB/s 的 GB/s 读取吞吐量。对于其他区域，采用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 个 MB/s 读取吞吐量。如果您需要将这些区域的最多 10 个 GB/s 写入容量和 20 个 GB/s 读取容量增加，请提交支持请求。	并无上限。最大吞吐量取决于为流预置的分片数量。每个分片可支持高达 1 MB/sec 或 1,000 的 records/sec 写入吞吐量或高达 2 MB/sec 或 2,000 的 records/sec 读取吞吐量。如果您需要更多的采集容量，则可以使用 AWS 管理控制台或 API 轻松扩大流中的分片数量。 UpdateShardCount
数据有效负载大小	使用 base64-encoding 之前记录的数据有效载荷的最大大小为 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）。	
GetRecords 事务大小	GetRecords每次调用可以从单个分片中检索最多 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。	
每个分片的数据读取速率	每个分片最多可支持每秒 2 MB 的总数据读取速率。 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。	

配额	On-demand 模式	预置模式
每个数据流的注册消费端数量	使用 Kinesis On-demand Advantage 模式，您最多可以创建 50 个注册用户（增强版 Fan-out）。在 Kinesis On-Demand 标准模式和 Kinesis 配置模式下，您可以为每个数据流创建最多 20 个注册用户（增强 Fan-out 限制）。	
在预置模式和按需模式之间切换	对于 AWS 账户中的每个数据流，您可以在 24 小时内两次在按需和预置容量模式之间切换。	
直播表和 Amazon S3 交付	有关适用于流媒体表和 Amazon S3 交付的配额和限制，请参阅the section called “配送配额和限制”。	不支持。流媒体表和 Amazon S3 交付需要按需容量模式。

API 限制

与大多数 AWS API 一样，Kinesis Data Streams API 操作受速率限制。下面是每个 AWS 账户在每个区域的限制。有关 Kinesis Data Streams API 的更多信息，请参阅 Amazon Kinesis API Reference。

KDS 控制层面 API 限制

下一节介绍 KDS 控制层面 API 限制。利用 KDS 控制面板 API，您可以创建和管理数据流。这些是每个 AWS 账户在每个区域的限制。

控制层面 API 限制

API	API 调用限制	Per Account/Stream	说明
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 `LimitExce`

API	API 调用限制	Per Account/Stream	说明
			ededExcep tion ： • 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	

API	API 调用限制	Per Account/Stream	说明
DisableEnhancedMonitoring	5 TPS	每个流	
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。

API	API 调用限制	Per Account/Stream	说明
RemoveTagsFromStream	5 TPS	每个流	
SplitShard	5 TPS	每个流	仅适用于预置模式
StartStreamEncryption		每个流	您可以在 24 小时的滚动周期内成功应用新的 KMS 密 AWS 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 AWS 账户中的每个数据流，您可以在 24 小时内两次在按需和预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。

API	API 调用限制	Per Account/Stream	说明
UpdateAccountSettings	5 TPS	每个 账户	启用或禁用账户设置，例如 On-demand 优势模式。

数据层面 API 限制

下一部分介绍 KDS 数据平面 API 的限制。利用 KDS 数据平面 API，您可以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 请求过于频繁地提出，则会收到 ProvisionedThrough

API	API 调用限制	负载限制	其他详细信息
			putExceededExcepti on.
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 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。	Kinesis 旨在使用突增容量处理间歇性的大型记录（大小为 1-10MiB）。
SubscribeToShard	每个分片 Subscribe ToShard 每秒可以对每个注册用户进行一次调用。		如果您使用相同的ConsumerARN SubscribeToShard 再次拨打电话，并且 ShardId 在成功通话后的5秒钟内，您将获得. ResourceInUseExcep tion

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。有些请求会自动解决，而另一些请求则提交给 AWS 支持部门。您可以跟踪提交给 AWS 支持部门的配额增加请求的状态。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，请联系客 AWS 服。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

- 注册参加 AWS 账户
- 下载库和工具
- 配置您的开发环境

注册参加 AWS 账户

要开始使用 AWS，你需要一个 AWS 账户. 有关创建的信息 AWS 账户，请参阅《AWS 账户管理 参考指南》 AWS 账户中的 “入门” 。

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 开发制作人 适用于 Java 的 AWS SDK
  - 通过以下方式培养消费者 适用于 Java 的 AWS SDK
  - 创建和管理 Kinesis 数据流
- 适用于 Go 、 Java、 .NET 、 PHP JavaScript 、P ython 和 Ruby 的 AWS 软件开发工具包包括 Kinesis 数据流支持和示例。如果您的版本 适用于 Java 的 AWS SDK 不包含 Kinesis 数据流的示例，您也可以从中GitHub下载这些示例。
- Kinesis Client Library（KCL）提供了一个易于使用的编程模型来处理数据。KCL 可以帮助你快速开始使用 Java、.NET Node.js、Python 和 Ruby 中的 Kinesis 数据流。有关更多信息，请参阅读取流中的数据。
- AWS Command Line Interface 支持 Kinesis Data Streams。 AWS CLI 使您能够通过命令行控制多个 AWS 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 AWS 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 AWS CLI 执行 Amazon Kinesis Data Streams 操作

本节向您展示如何使用 AWS 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：安装和配置 AWS CLI 适用于 Kinesis Data Streams
- 教程：使用 Kinesis Data Streams 执行基本操作 AWS CLI

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证书与 AWS 资源配合使用。

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会不同。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：安装和配置 AWS CLI 适用于 Kinesis Data Streams

安装 AWS CLI

有关如何安装适用于 Windows 以及 AWS CLI 适用于 Linux、OS X 和 Unix 操作系统的详细步骤，请参阅安装 AWS CLI。

使用以下命令列出可用的选项和服务：

```
aws help
```

您将使用 Kinesis Data Streams 服务，因此您可以使用以下命令查看 AWS CLI 与 Kinesis Data Streams 相关的子命令：

```
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。

AWS CLI 现在已成功安装，但尚未配置。这将在下一节展示。

配置 AWS CLI

一般而言，该`aws configure`命令是设置 AWS CLI 安装的最快方法。有关更多信息，请参阅配置 AWS CLI。

教程：使用 Kinesis Data Streams 执行基本操作 AWS CLI

本节介绍如何通过 AWS 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**
>
> 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 AWS 免费套餐的资格。完成本教程后，请删除您的 AWS 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：

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "CREATING",
        "RetentionPeriodHours": 48,
```

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

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": "AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd
+efGN2aHFdkH1rJl4BL9Wyrk+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）。你必须 copy/paste 将分片迭代器放入 get 命令，如下所示。分片迭代器的有效生命周期为 300 秒，这应该足以让分片迭代器进入下一个命令。 copy/paste 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
 AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG4lNR1EMi
+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

如果您使用 Unix-type 命令处理器（例如 bash）运行本教程，则可以使用嵌套命令自动获取分片迭代器，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
```

如果您在支持的系统上运行本教程 PowerShell，则可以使用如下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":"AAAAAAAAAAEDOW3ugseWPE4503kqN1yN1UaodY8unE0sYslMUmC6lX9hlig5+t4RtZM0/
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvPOZvrUIudb8UkH3V
}
```

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，中的 AWS CLI Kinesis Data Streams支持不提供 Base64 解码，因为Base64解码到打印到标准输出的原始二进制内容可能会导致某些平台和终端出现不良行为和潜在的安全问题。如果您使用 Base64 解码器（例如 https://www.base64decode.org/）进行手动解码，dGVzdGRhdGE=您会发现它实际上是。testdata就本教程而言，这已经足够了，因为在实践中， AWS CLI 很少使用来消耗数据。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并不总是返回 stream/shard指定中的所有记录。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，它是从流的末端响应 GetRecords 操作的毫秒数，指示使用者落后当前时间多远。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": "AAAAAAAAAAGCJ5jzQNjmdhO6B/YDIDE56jmZmrmMA/r1WjoHXC/
kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wul/
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： AWS Lambda 与 Amazon Kinesis Data Streams 配合使用
- 使用适用于 Amazon Kinesis 的 AWS 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 AWS 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 AWS 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 AWS 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：安装和配置 AWS CLI 适用于 Kinesis Data Streams。

您必须拥有一个 AWS 帐户和一个网络浏览器才能访问 AWS 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 AWS 管理控制台。有关 AWS 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AWS 安全证书。有关登录您的的详细信息 AWS 账户，请参阅《AWS 登录 用户指南》 AWS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 AWS 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 AWS 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 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 AWS 要求使用细粒度的权限来控制对不同资源的访问权限。 AWS Identity and Access Management (IAM) 允许您在中管理用户和用户权限 AWS。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，数据流处于活动状态，以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，数据流处于活动状态，以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 或 2.x）构建的，则需要 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，KCL 会删除 DynamoDB 表。
`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
   ```

   region

   AWS 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

   AWS 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 (h https://console.aws.amazon.com/iam/ome #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。 AWS

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)：

AWS 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-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）
   - AWS SDK
   - 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getRandomTrade()，该方法每次被调用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
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。可以在 Amazon EC2 实例中运行工作程序并使用自动扩缩组。

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。 AWS 提供了用于将 Kinesis Data Streams AWS 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 AWS 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 AWS 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 [如 Amazon Elastic Compute Cloud（Amazon EC2）] 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 AWS 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：安装和配置 AWS CLI 适用于 Kinesis Data Streams也很有帮助。

您需要一个 AWS 帐户和一个网络浏览器才能访问 AWS 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 AWS 管理控制台。有关 AWS 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AWS 安全证书。有关登录您的的详细信息 AWS 账户，请参阅《AWS 登录 用户指南》 AWS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 AWS SDK 版本。如果您将 Eclipse 用作 IDE，则可改为安装 AWS 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 K GitHub inesis 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 AWS 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 AWS 要求使用细粒度的权限来控制对不同资源的访问权限。 AWS Identity and Access Management (IAM) 允许您在中管理用户和用户权限 AWS。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状态，流中，以及流是否具有消费端。
`SubscribeToShard`，`RegisterStreamConsumer`	Kinesis 数据流	订阅 Kinesis 数据流分片并注册一个消费端。

操作	资源	用途
`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 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
`DeleteItem`	Amazon DynamoDB 表	当使用者对 Kinesis Data Streams 分片执行 split/merge 操作
`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

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

1. 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
```

region

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

AWS 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/ome #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。 AWS
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)：

AWS 服务	操作	进行筛选
Amazon DynamoDB	CreateTable，DeleteItem，DescribeTable，GetItem, PutItem, Scan, UpdateItem	您在步骤 2 中创建的 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: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）
   - AWS SDK
   - 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
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。可以在 Amazon EC2 实例中运行工作程序并使用自动扩缩组。

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。 AWS 提供了用于将 Kinesis Data Streams AWS 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 开发制作人 适用于 Java 的 AWS SDK、使用开发吞吐量共享的消费者 适用于 Java 的 AWS SDK 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。 AWS 使用完此应用程序后，请删除您的 AWS 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AWS 区域。

任务

- 完成练习的先决条件
- 设置 AWS Command Line Interface (AWS 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，请转到注册参加 AWS 账户。

设置 AWS Command Line Interface (AWS 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。 AWS 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 AWS 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《AWS Command Line Interface 用户指南》中的安装 AWS 命令行界面。要检查的版本 AWS CLI，请运行以下命令：
>
> ```
> aws --version
> ```
>
> 本教程中的练习需要以下 AWS CLI 版本或更高版本：

```
aws-cli/1.16.63
```

要设置 AWS CLI

1. 下载并配置 AWS CLI。有关说明，请参阅《AWS Command Line Interface 用户指南》中的以下主题：

   - 安装 AWS Command Line Interface
   - 配置 AWS CLI

2. 在配置文件中为管理员用户添加命名的 AWS CLI 配置文件。执行 AWS 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 AWS 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 AWS 区域的列表，请参阅中的AWS 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 AWS 帐户和之后 AWS CLI，您可以尝试下一个练习，即配置示例应用程序并测试端到端设置。

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 AWS 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。

要创建数据流 (AWS CLI)

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

1. 打开 Amazon S3 控制台，网址为 https://console.aws.amazon.com/s3/。
2. 选择 创建存储桶 。
3. 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。
4. 在配置选项步骤中，让设置保持原样，然后选择下一步。
5. 在设置权限步骤中，让设置保持原样，然后选择下一步。
6. 选择 创建存储桶 。
7. 在 Amazon S3 控制台中，选择 ka-app-code-bucke-***<username>*** bucket，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 AWS 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 AWS Identity and Access Management (IAM) 和 A CloudWatch mazon Logs 资源。使用创建应用程序时 AWS 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 (AWS 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. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
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 ID (*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:*"
            ]
        },
        {
            "Sid": "ListCloudwatchLogStreams",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams"
            ],
            "Resource": [
```

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

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 (AWS CLI)

在本节中，您将使用创建和运行适用 AWS CLI 于 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` AWS 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 (ARN) 中的账户 ID (*`012345678901`*) 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分步说明，请参阅《IAM 用户指南》中的教程：创建和附加您的第一个客户管理型策略。

> **Note**
>
> 要访问其他 AWS 服务，可以使用 适用于 Java 的 AWS 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 AWS 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 **KAReadSourceStreamWriteSinkStream**（您在上一部分中创建的策略）。
   d. 选择KAReadInputStreamWriteOutputStream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。

有关创建角色的分步说明，请参阅《IAM 用户指南》中的在您的中创建 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： AWS Lambda 与 Amazon Kinesis Data Streams 配合使用

在本教程中，您将创建 Lambda 函数来处理 Kinesis 数据流中的事件。在此示例场景中，自定义应用程序将记录写入 Kinesis 数据流。 AWS Lambda 然后轮询此数据流，当它检测到新的数据记录时，调用

您的 Lambda 函数。 AWS Lambda 然后使用您在创建 Lambda 函数时指定的执行角色来执行 Lambda 函数。

有关详细的分步说明，请参阅教程：在 Amazon Kinesis 中使用 AWS Lambda。

> **Note**
>
> 本教程假设您对 Lambda 基本操作和 AWS Lambda 控制台有一定了解。如果您还没有，请按照 Lambda 入门中的说明创建您的 AWS 第一个 Lambd a 函数。

使用适用于 Amazon Kinesis 的 AWS 流数据解决方案

适用于 Amazon Kinesis 的 AWS 流数据解决方案可自动配置必要的 AWS 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 AWS 服务的流数据用例提供了多种选项，包括 Kinesis Data Streams AWS 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 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 和预置容量模式下，分片每秒支持 1 和 1000 条记录 MB/s 的写入，每秒支持 2 条记录的读取。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，您将学习如何设置数据流的容量模式以及如何使用 AWS 管理控制台 或 API 创建流。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 AWS 管理控制台
- 使用 API 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 AWS 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？
- On-demand 标准模式的功能和用例

- On-demand 优势模式的功能和用例
- 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 数据流中，您可以选择 “On-demand 标准” 、“On-demand 优势” 和 “预配置” 作为数据流的模式。

- On-demand 标准 — 采用按需模式的数据流无需容量规划，可自动扩展以处理每分钟千兆字节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。
- On-demand 优势 — 一种账户级模式，可为您提供更多功能，并为按需直播提供更简单的定价结构。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。如果瞬态数据突发导致数据流的规模明显超过其预热容量，您也可以触发缩小规模。在定价方面，不再为每个直播收取固定费用。所有按需流的数据摄取、数据检索和延长保留期的使用量比 On-demand 标准版至少低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。

您可以使用 Kinesis Data Streams `PutRecord` 和 `PutRecords` API 在任何模式下将数据写入数据流。要检索数据，所有三种模式都支持使用 `GetRecords` API 的默认消费者和使用 API 的增强型 Fan-Out (EFO) 消费者。`SubscribeToShard`

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。

On-demand 标准模式的功能和用例

按需模式下的数据流无需容量规划，可自动扩展以处理每分钟千兆字节的写入和读取吞吐量。 On-demand 模式简化了以低延迟摄取和存储大量数据的过程，因为它无需配置和管理服务器、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。

On-demand 模式非常适合满足高度可变和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，由于利用率低，这可能会导致更高的成本。 On-demand 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介于 10 到 40 MB/s 之间 MB/s，则 Kinesis Data Streams 可确保您可以轻松突增至之前峰值吞吐量的两倍，即 80 MB/s。如果同一个数据流承受新的峰值吞吐量达到 50 MB/s，Kinesis 数据流将确保有足够的容量来摄取 100% MB/s 的写入吞吐量。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类应用程序的场景，建议您使用 Kinesis Data Streams 的增强 Fan-Out 功能。增强版 Fan-Out 支持使用 `SubscribeToShard` API 向数据流中添加多达 20 个消费者应用程序，每个消费类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 每个分区的传入流量超过 500 时，它会在 15 分钟内拆分分分分区。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，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，否则数据流会自动适应不均匀的数据分布模式。

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。

On-demand 优势模式的功能和用例

On-demand Advantage 是一种账户级别设置，可解锁更多功能，并为该地区的所有按需直播提供不同的定价结构。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 到 40 MB/s 之间 MB/s，则可以预期它最多能够 80MB/s 在不限制的情况下处理即时吞吐量增长。但是，如果您预测即将发生的事件将在流量达到峰值，则可以将数据流配置为预吞吐量， 200MB/s 以确保数据吞吐量到达时容量可用。 200MB/s 使用热吞吐量不会产生任何额外费用。如果瞬态数据突发导致数据流的扩展超过其预热容量，则可以触发缩小规模。为此，请将流的热吞吐量重新配置为相同或更低的值。流容量减少到请求的数量或在过去一小时内支持峰值数据采集使用量所需的数量，以较高者为准。

On-demand 优势模式的另一个好处是，按需直播可以过渡到更简单的定价结构。启用该模式后，您将不再看到固定的每流费用，您只需要处理数据提取、数据检索和可选的延长保留费用。与 On-demand 标准版中的相应维度相比，每个定价维度也都有很大的折扣。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，借助 On-demand 优势模式，您每次直播最多可以注册 50 名消费者，以使用增强版粉丝分享。启 On-demand 用 Advantage 会要求账户至少 25MiB/s 参与所有按需流的数据采集和 25MiB/s 数据检索。对于满足最低使用要求的账户，Kinesis Data Streams 控制台会检查您的账户的使用模式是否适合使用 On-demand 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 On-demand 用 Advantage 还有至少 24 小时的时间才能禁用该模式。总体而言，如果您的吞吐量持续使用量接近或高于最低承诺，需要大量扇出消费者，或者使用数百个数据流进行操作，那么 On-demand Advantage是使用Kinesis Data Streams进行传输的最佳方式。

预置模式的特征和用例

在预置模式下，在创建数据流后，您可以使用或 API 动态地向上或向下扩展分片容量。 AWS 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

对于您的中的每个数据流 AWS 账户，您可以在 24 小时内两次在按需模式和预置模式之间切换。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。

您可以通过启用账户级别设置从 On-demand 标准模式切换到 On-demand 优势模式。启用后，该账户承诺至少 25MiB/s 在该地区的所有按需流中提取数据和 25MiB/s 使用数据检索。启用后，必须等待至少 24 小时才能禁用 On-demand Advantage，但您可以随时申请更改。如果要从 On-demand Advantage 切换到 On-demand 标准，则必须先移除在按需流上配置的所有热吞吐量。

使用创建直播 AWS 管理控制台

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

使用控制台创建数据流

1. 登录 AWS 管理控制台 并打开 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. 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On-demand或预置容量模式。默认情况下，该On-demand模式处于选中状态。有关更多信息，请参阅 选择正确的流式传入的模式。

   在该On-demand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 API 创建流。

要使用创建直播 AWS CLI

- 有关使用创建流的信息 AWS CLI，请参阅 cre ate-stream 命令。

使用 API 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《AWS 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。

- On-demand:

```
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 AWS 帐户。它还受区域限制。也就是说，两个不同 AWS 账户中的两个直播可以有相同的名称，同一个 AWS 账户但位于两个不同区域中的两个直播可以有相同的名称，但不能有同一个账户和同一个区域中的两个直播同一个名称。

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。更多的碎片也会增加直播的 AWS 收费。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 AWS 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. 在以下位置打开亚马逊 Kinesis 控制台。 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。
4. 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 AWS 账户中的每个数据流，您可以在 24 小时内两次在按需模式和预置模式之间切换。

5. 对于启用了 On-demand Advantage 模式的按需模式的数据流，您可以配置热吞吐量以增加或减少流的摄取容量。

   增加摄取容量

   在 “配置” 选项卡中，选择 “编辑热吞吐量”，然后输入高于流当前容量的目标吞吐量值（英寸 MB/s）。该流会立即为至少指定的热吞吐量配置容量，从而确保它能够在不限制的情况下处理预期的流量。例如，如果您预测即将发生的事件将在 200 左右达到峰值 MB/s，请 MB/s 提前将预热吞吐量设置为 200，无需额外付费。

   减少摄取容量

   如果流的规模已明显超过其热吞吐量值，请将热吞吐量重新配置为该值或更低的值，以触发缩小规模。数据流可扩展到请求的数量或支持过去一小时内观察到的峰值数据采集使用量所需的容量，以较高者为准。这样可以确保直播始终保留足够的容量来处理最近的流量模式。

   使用热吞吐量不会产生任何额外费用。有关更多信息，请参阅 On-demand 优势模式的功能和用例。
6. 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。
7. 要启用数据记录的服务器端加密，请在Server-side加密部分选择编辑。选择 KMS 密钥用作加密的主密钥，或使用由 Kinesis 管理的默认主密钥。 aws/kinesis 如果您为流启用加密并使用自己的 AWS KMS 主密钥，请确保您的生产者和消费者应用程序有权访问您使用的 AWS KMS 主密钥。

要将权限分配给应用程序以访问用户生成的 AWS KMS 密钥，请参阅the section called “使用用户生成的 KMS 密钥的权限”。

8. 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。
9. 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 “使用监控 Kinesis Data Streams 服务 CloudWatch”。

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
- UpdateStreamMode
- UpdateStreamWarmThroughput— 为账户启 On-demand 用 Advantage 后，使用此 API 为按需直播配置热吞吐量。当您使用 UpdateStreamMode API 将流从预置 On-demand 模式切换到模式时，您还可以设置热吞吐量。

使用 AWS CLI

有关使用更新直播的信息 AWS CLI，请参阅 Kinesis CLI 参考文档。

要使用配置热吞吐量 AWS CLI，请使用`update-stream-warm-throughput`命令。例如：

```
aws kinesis update-stream-warm-throughput \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-stream \
    --warm-throughput-mi-bps=200
```

要降低容量，请重新配置`warm-throughput-mi-bps`为相同或更低的值：

```
aws kinesis update-stream-warm-throughput \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-stream \
    --warm-throughput-mi-bps=50
```

数据流可扩展到请求的热吞吐量或支持近期峰值数据采集使用量所需的容量，以较高者为准。

列出流

流的范围限定为与用于实例化 Kinesis Data Streams 客户端的 AWS 凭据关联的 AWS 账户，也限于为该客户端指定的区域。一个 AWS 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。本节中的代码显示如何列出您 AWS 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 API 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。列出或从数据流中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要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参数过滤掉 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是必需的属性，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 和 ListShard API 提供了一种更具可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，DescribeStream配额将在与您 AWS 账户中的所有数据流交互的所有应用程序之间共享。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，操作的扩展性也会更好。
我们建议您迁移所有调用 DescribeStream API 的生产者和消费者，改为调用 ListShard 和 API。 DescribeStreamSummary 为了识别这些生产者和消费者，我们建议使用 Athena 解析 CloudTrail 日志，因为在 API 调用中捕获了 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 DescribeStream API 的 AWS Lambda 和 Amazon Firehose 与 Kinesis 数据流的集成，以改为调用和。DescribeStreamSummary ListShards具体而言，

对于 AWS 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。此类管理应用程序根据亚马逊提供的指标 CloudWatch 或从制作人和消费者那里收集的指标来监控直播的整体性能。相比消费端或创建器，管理应用程序还需要更广泛的 IAM 权限集，因为消费端和创建器一般不需要对用于重新分片的 API 的访问权限。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 指标中获取数据流的一些性能数据。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 使用 MD5 计算分区键中的哈希键。由于您为记录指定了分区键，因此可使用 MD5 计算该记录的哈希键值并将此值记录下来。

您还可记录您的数据记录分配到的分片的 ID。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();
```

利用分片 ID 和哈希键值，您可确定将接收最多流量或最少流量的分片和哈希键。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。哈希键值是在向流添加数据记录时为数据记录指定的分区键的 MD5 哈希。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。这只是一种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 MD5 哈希键值的整个范围。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。您可以使用该IncreaseStreamRetentionPeriod操作或 Kinesis Data Streams 控制台将保留期延长至最多 8760 小时（365 天）。您可以使用该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 AWS 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 AWS 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 AWS CLI
- 使用 Kinesis Data Streams 标记直播 APIs
- 使用标记消费者 AWS 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 AWS 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 AWS 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 AWS 费用进行分类和跟踪。当您对 Kinesis Data Streams 资源应用标签时， AWS 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《AWS 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 AWS。 AWS 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 AWS 管理控制台 并在 /kinesis 上打开 Kinesis 控制台。https://console.aws.amazon.com
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 AWS CLI

您可以使用 AWS 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 Dat 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 AWS CLI

您可以使用 AWS 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 Dat 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

亚马逊 Kinesis Data Streams 支持高达 10 兆字节 () 的记录。MiBs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有效载荷。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有效载荷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，读取的吞吐量限制 MB/s 为 2，支持更大的记录大小，因此保持不变。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。

大型记录的工作原理

亚马逊 Kinesis 数据流接受大小不超过 10 MiB 的记录。您的数据流通过暂时突增超过其持续写入吞吐量来容纳大型记录，然后随着时间的推移恢复到基准速率。这种突发容量会持续补充，因此您的直播可以在处理正常流量的同时处理间歇性的大型记录，而无需进行任何手动容量调整。

要想可视化这种行为，可以将直播的写入容量想象成一个以稳定速度重新填充的容量。当您发送大型唱片（例如 10 MiB 唱片）时，坦克会暂时耗尽。然后，它会立即开始重新填充，这意味着当容量可用时，您可以继续发送较小的记录。

容量的补充速度取决于以下几个因素：

- 大型记录的大小
- 基线记录的大小
- 直播中的整体流量模式
- 你选择的分区键策略

为获得最佳结果，请使用均匀分布的分区键将大型记录分布在流的可用容量中。

在按需模式下，Kinesis 数据流会自动管理容量。您的直播会根据您的流量模式向上和向下扩展其吞吐量，并且可以透明地处理创纪录的突发容量。您无需预置或管理容量即可使用大型记录。有关按需模式如何扩展的更多信息，请参阅On-demand 模式功能和用例。

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 AWS 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

大型唱片专为间歇性使用而设计。为获得最佳效果，请将大型记录保持在总流量的2%以下。由于数据流会暂时超过其持续吞吐量以提供大量记录，因此过于频繁地发送大型记录会减少可用于基准流量的容量。有关使用大型记录优化直播性能的更多信息，请参阅限制和最佳性能的最佳实践。

使用大型记录缓解限制

由于大型记录会暂时消耗突发容量，因此您的数据流可能会限制后续写入，直到容量得到补充。以下步骤有助于减少节流：

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. 使用随机分区键在流的可用容量上分配大型记录。

3. 对于连续的大型记录流，将有效载荷存储在 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 API 处理大型记录

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 API，以处理最多 10 个记录 MiBs。

用于修改记录大小的 API：

- `UpdateMaxRecordSize`：将现有流的最大记录大小限制配置为 10 MiBs。

更新现有 API：

- `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。

所有列出的 API 都保留了对现有流的向后兼容性。有关完整的 API 文档，请参阅 Amazon Kinesis Data Streams 服务 API 参考。

AWS 与大型记录兼容的组件

以下 AWS 组件与大型记录兼容：

组件	说明
AWS SDK	AWS SDK 支持处理大型记录。您可以使用软件开发工具包中的可用方法，将直播的最大记录大小更新为 10 MiB。 AWS 有关更多信息，请参阅将此服务与 AWS SDK 一起使用。
Kinesis Consuer 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 `maxRecordSize` 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 `maxRecordSize` 并使用 KPL。有关更多信息，请参阅使

组件	说明
	用 Amazon Kinesis Producer Library（KPL）开发产生器。
Amazon EMR	搭载 Apache Spark 的 Amazon EMR 支持处理不超过 Kinesis 数据流限制 (10 MiBs) 的大型记录。要使用大型记录支持，请使用 `readStream` 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： • 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。 • Amazon Redshift：Firehose 不支持传输大型记录。 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。对于其他所有目的地，请在概念验证环境中使用大型记录测试行为之后，再扩展到生产使用量。大型记录的处理因目的地而不同。

组件	说明
AWS Lambda	AWS Lambda 最多支持 6 个 MiBs有效载荷。此限制适用于转换为 base-64 编码的 Kinesis 有效载荷，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 进行处理，无需额外配置。对于大于 6 的记录 MiBs，Lambda 使用故障时目的地进行处理。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。 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。
Amazon Redshift	将数据从 Kinesis Data Streams 流式传输到亚马逊 Redshift 时，请检查亚马逊 Redshift 支持的最大记录大小。有关详细信息，请参阅《亚马逊 Redshift 数据库开发人员指南》中的 “从亚马逊 Kinesis 数据流中提取流媒体入门”。超过此限制的记录不予处理。有关未处理记录的日志记录信息的详细信息，请参阅《亚马逊 Redshift 数据库开发者指南》中的 SYS_STREAM_SCAN_ERRORS。
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 MiBs MiB 的记录。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 DataStream API 在适用于 Apache 的亚马逊托管服务 Flink 中使用连接器移动数据。

支持大型记录的区域

此 Amazon Kinesis 数据流功能仅在以下 AWS 地区可用：

AWS 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）

AWS 区域	区域名称
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（西班牙）
cn-northwest-1	中国（宁夏）
eu-central-2	欧洲（苏黎世）
us-gov-east-1	AWS GovCloud (US-East)
us-gov-west-1	AWS GovCloud (US-West)

使用执行弹性测试 AWS Fault Injection Service

AWS Fault Injection Service 是一项完全托管的服务，可帮助您对 AWS 工作负载执行故障注入实验。AWS 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 AWS 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 A CloudWatch pplication 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 AWS 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 AWS 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：`ProvisionedThroughputExceededException`（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。

参数

- 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 AWS FIS API 中，该值是 ISO 8601 格式的字符串。例如，PT1M 代表一分钟。在 AWS 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。 AWS 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",
              "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 AWS Fault Injection Service？

进行基础实验

1. 创建实验模板：使用 AWS 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 AWS 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。

参数

- 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`

Service-managed 唱片分发策略

Amazon Kinesis Data Streams 提供了两种记录分发策略，用于确定记录在流中的分片之间的分布方式：

- User-managed 分区键（默认）— 您为每条记录提供一个分区键，然后 Kinesis Data Streams 使用分区键哈希来确定分片的位置。对于需要为相关记录提供排序保证的工作负载，这是传统行为。
- Service-managed 记录分发 — Kinesis Data Streams 使用服务托管算法自动在分片之间平均分配记录。您无需提供分区键。这是为不需要记录排序的无状态工作负载而设计的。

服务管理记录分发的工作原理

当您在流上启用服务管理的记录分发策略时，Kinesis Data Streams 将接管跨分片分发记录的责任。该服务使用内部算法来均匀分布记录并避免出现热分片。这消除了客户生成的分区密钥的常见问题，例如流量分布不均匀。它还可以防止旋转热键节流。

服务管理记录分发的主要特征：

- Stream-level 设置-记录分发策略是在流级别配置的，适用于写入该流的所有记录。
- On-demand 仅限直播 —— Service-managed 记录分发适用于按需直播（On-Demand 标准版和 On-Demand 优选版）。已配置的流不支持该功能。
- 没有排序保证 — 由于该服务使用内部算法在分片之间分配记录，因此具有相同业务实体的记录不一定到达同一个分片。请勿将此模式用于依赖于排序的工作负载。
- 忽略分区键-启用服务管理模式后，将忽略PutRecord或 PutRecordsAPI 调用中提供的任何分区键值。该服务在内部处理分发。
- 无需额外费用 — 无需额外费用即可分发 Service-managed 唱片。适用标准的 Kinesis Data Streams 定价。

何时使用服务管理的记录分发

对于不需要对记录进行排序的无状态流式处理工作负载，请使用服务管理的记录分发：

- 日志聚合
- 指标收集
- 物联网遥测

- 点击流分析

对于需要订购保证的有状态工作负载，继续使用客户管理的分区键：

- 数据库变更数据捕获 (CDC)，其中必须按顺序处理同一行的更新
- 需要账户级订购的金融交易处理
- Session-based 分析，其中来自用户会话的所有事件必须一起处理
- 业务逻辑依赖于按生成顺序处理相关记录的任何应用程序

唱片分发策略比较

下表比较了 Kinesis Data Streams 中可用的两种记录分发策略。

记录分发策略对比

特征	User-managed 分区键	Service-managed 记录分发
需要分区密钥	是 — 您必须为每条记录提供分区密钥	否 — 该服务会自动分发记录
记录排序	具有相同分区键的记录按顺序传送到同一个分片	没有订购保证 — 分发记录是为了获得最佳吞吐量
分片分配	取决于您的分区密钥策略 — 密钥分布不当可能会导致分片过热	均匀的分发由服务算法管理 — 消除了热分片
支持的直播模式	On-Demand 标准版、 On-Demand 优选版和预配置版	On-Demand 仅限标准版和 On-Demand 优选版
适用于	需要订购的有状态工作负载（CDC、金融交易、会话分析）	无状态工作负载（日志、指标、遥测、点击流）
需要更改生产者代码	N/A （现有行为）	请参考该部分 带有服务管理流的 KPL 和 KCL

过渡行为

您可以随时在服务管理的记录分发和用户管理的记录分发之间切换。模式更改会立即生效，不会导致已写入的记录停机或数据丢失。生产者和消费者应用程序可以继续运行，但以下情况除外：

- 切换到用户管理的发行版后，省略分区键的生产者会失败，必须更新才能提供分区键。
- 在您切换到服务管理分发并且创建者开始省略分区键之后，假设分区键始终存在的使用者在读取没有分区键的记录时可能会失败。在启用之前，请验证您的使用者是否处理了缺少的分区密钥AUTO。有关更多信息，请参阅重要注意事项和中的 AWS SDK 注意事项带有服务管理流的 KPL 和 KCL。

过渡期间：

- 流中已有的记录会根据之前的策略保留其原始分片分配。
- 模式更改后收到的新记录将根据新策略进行分发。
- 无论这些记录是如何分发的，消费者都会继续按顺序读取分片中的记录。

有关启用或禁用服务管理记录分发的说明，请参阅配置服务管理的记录分发。

切换策略会改变以后在分片上放置记录的方式，这会影响每个分区键的排序：

- User-managed 到服务管理（USER_PARTITION_KEY → AUTO）— 先前共享分区键的记录被路由到同一个分片并相对于彼此进行排序。切换后，无论分区键如何，新记录都会在分片之间均匀分布，因此不再保证共享分区键的记录会落在同一个分片上或按顺序处理。
- Service-managed 到用户管理（AUTO → USER_PARTITION_KEY）— 切换后，新记录将再次按分区键哈希进行路由，因此具有相同分区键的记录会恢复在同一个分片上。生产者必须在切换后提供分区密钥；PutRecord省略分区密钥的PutRecords调用将失败。在流处于 AUTO 模式时写入的记录会保留其原始分片位置，并且不会重新分配，因此单个分区键的记录可以在切换前 (AUTO) 和切换后的分片之间拆分。

重要注意事项

- 空分区键的 KPL 升级 — 想要发送没有分区键的记录的客户应升级到最新的 KPL 版本。对于继续提供分区密钥的客户，无需更改 KPL 或 KCL。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。
- AWS SDK 注意事项 — 如果您直接使用 AWS SDK（而不是 KPL 或 KCL）向服务管理的直播制作或使用服务，请在启用之前查看以下内容：AUTO

  - P@@ ro ducter — 升级到最新的 SDK 版本以在写入服务管理的流时省略分区键。在最新的 SDK 上，PartitionKey是可选的，因此您可以PutRecords不使用它PutRecord进行调用。如果您继续提供分区键，则无需更改 SDK，服务会接受该值并在分片放置时忽略该值。在启用AUT0并开始在现有使用者从中读取的流上发送空分区键之前，请更新这些使用者以处理不存在的分区键PartitionKey。
  - 使用者-在服务管理的流上，GetRecords对于没有缺失（空）值的记录，则SubscribeToShard返回PartitionKey为缺失（空）值。任何读取PartitionKey并假设它始终存在的消费者在读取此类记录时都可能在运行时失败——无论SDK版本如何，这都适用。升级到最新的 SDK 版本以安全地读取这些记录，并更新您的代码以处理缺失的值。对于 AWS 适用于 Go 的 S AWS DK、适用于 Kotlin 的 AWS SDK 和适用于 Rust 的 SDK，升级还会更改PartitionKey为可选类型，因此在升级时，未处理的缺失值会显示为编译时错误。不读书的消费者PartitionKey不受影响。
- 不支持聚合-任何 KPL 版本的服务管理流均不支持 KPL 聚合。如果要使用服务管理的记录分发，请考虑禁用聚合。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。
- 不能在直播中混音 — 记录分发策略适用于整个视频流。不能在同一个直播中同时使用服务管理的分区键和用户管理的分区键。对于同时需要有序和无序数据的工作负载，请创建单独的流。
- 没有新的服务限制 — Service-managed 记录分发不会引入任何新的限制。现有的 Kinesis Data Streams 限制继续适用，包括每条记录上限 1 MB、写入每分片每秒 1,000 条记录以及每次请求 500 条记录 PutRecords 。

配置服务管理的记录分发

在创建新直播或更新现有点播流的配置时，您可以启用服务管理的记录分发。记录分发策略是一种直播级设置，可立即生效。

先决条件

在配置服务管理记录分发之前，请确保满足以下条件：

- 您的直播使用按需容量模式（On-Demand 标准或 On-Demand 优势）。 Service-managed 预配置的直播不支持记录分发。
- 如果您使用 KPL 聚合，请确保 KPL 在写入服务管理的流时不会聚合记录。您可以明确禁用聚合，也可以升级到最新的 KPL 版本并向其授予kinesis:DescribeStreamSummary权限，以便它自动检测流并禁用聚合。任何 KPL 版本的服务管理流都不支持聚合。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。

- 如果您想发送没有分区键（空分区键）的记录，请升级到最新的 KPL 版本。对于继续提供分区密钥的客户，无需更改 KPL 或 KCL。

使用服务管理的记录分发创建直播

要使用服务管理的记录分发创建新流，请在 CreateStreamAPI 调用AUTO中加入设置为的RecordDistributionStrategy参数。如果您未指定此参数，则流默认为USER_PARTITION_KEY（用户管理的分区键）。

Example使用服务管理分发创建直播 (AWS CLI)

```
aws kinesis create-stream \
    --stream-name my-telemetry-stream \
    --stream-mode-details StreamMode=ON_DEMAND \
    --record-distribution-strategy AUTO
```

Example使用服务管理分发创建直播 (适用于 Java 的 AWS SDK)

```
CreateStreamRequest request = CreateStreamRequest.builder()
    .streamName("my-telemetry-stream")
    .streamModeDetails(StreamModeDetails.builder()
        .streamMode(StreamMode.ON_DEMAND)
        .build())
    .recordDistributionStrategy(RecordDistributionStrategy.AUTO)
    .build();

kinesisClient.createStream(request);
```

在现有直播上启用服务管理的记录分发

要在现有点播流上启用服务管理的记录分发，请使用 UpdateStreamRecordDistributionStrategy API。更改会立即生效。在启用之前AUTO，请验证您的使用者是否可以处理缺失（空）的分区密钥，并确保任何 KPL 生产者都不会聚合记录。有关更多信息，请参阅 安全的迁移模式。

Example在现有直播上启用服务管理分发 (AWS CLI)

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream \
    --record-distribution-strategy AUTO
```

Example启用服务托管分发 (适用于 Java 的 AWS SDK)

```
UpdateStreamRecordDistributionStrategyRequest request =
    UpdateStreamRecordDistributionStrategyRequest.builder()
        .streamName("my-telemetry-stream")
        .recordDistributionStrategy(RecordDistributionStrategy.AUTO)
        .build();

kinesisClient.updateStreamRecordDistributionStrategy(request);
```

您也可以按名称代替 ARN 来指定直播：

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-name my-telemetry-stream \
    --record-distribution-strategy AUTO
```

验证当前记录分发策略

要查看直播当前正在使用哪种记录分发策略，请调用 DescribeStreamSummary API。响应包括字RecordDistributionStrategy段。

Example查看记录分发策略 (AWS CLI)

```
aws kinesis describe-stream-summary \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream
```

响应包括以下内容：

```
{
    "StreamDescriptionSummary": {
        "StreamName": "my-telemetry-stream",
        "StreamARN": "arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-
stream",
        "StreamStatus": "ACTIVE",
        "StreamModeDetails": {
            "StreamMode": "ON_DEMAND"
        },
        "RecordDistributionStrategy": "AUTO",
        ...
    }
}
```

在控制台中配置记录分发策略

您只能在 Kinesis Data Streams 控制台中查看和配置使用按需容量模式的直播的记录分发策略。如果您的直播使用预配置容量模式，则控制台会显示一个短划线，表示该设置不适用，并且您无法更改策略。

创建直播时选择记录分发策略

1. 打开 Kinesis Data Streams 控制台https://console.aws.amazon.com/kinesis，网址为。
2. 选择创建数据流。
3. 输入数据流的名称。
4. 在数据流容量下，选择On-demand容量模式。记录分发策略面板显示在容量面板下方。选择 “已配置” 模式时，此面板处于隐藏状态。
5. 在 “记录分发策略” 面板中，选择以下选项之一：
   - 自动 — Kinesis Data Streams 在分片之间平均分配记录，并忽略创建者提供的任何分区密钥。对于不需要分区键排序的无状态工作负载，请选择此选项。
   - 用户集 — 生产者必须提供分区键，Kinesis Data Streams 使用它来确定分片的位置。这是默认值。
6. 选择创建数据流。

查看和编辑现有直播的记录分发策略

1. 打开 Kinesis Data Streams 控制台https://console.aws.amazon.com/kinesis，网址为。
2. 选择要更新的数据流。
3. 选择配置选项卡。当前记录分发策略显示在 “数据流容量” 面板中。
4. 选择 “记录分发策略” 旁边的 “编辑”。
5. 选择 “自动” 或 “用户设置”，然后保存更改。更改会立即生效。

> **Note**
>
> 当您切换策略时，控制台会显示一条警告，提醒您服务管理（自动）分发消除了分区密钥排序保证，并且用户管理（用户集）分发要求生产者提供分区键。

您还可以在控制台的以下位置查看记录分发策略：

- 数据流表中的 “记录分配策略” 列显示已置备流的 “自动”、“用户设置” 或 “虚线”。
- 直播详细信息页面上的数据流摘要面板显示当前的记录分发策略。

切换回用户管理的分区键

要将直播切换回用户管理的分区键，请在RecordDistributionStrategy设置为USER_PARTITION_KEY的情况下调用 UpdateStreamRecordDistributionStrategy API。

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream \
    --record-distribution-strategy USER_PARTITION_KEY
```

更改会立即生效。所有后续调用PutRecord和 PutRecordsAPI 调用都需要分区键，并且该服务使用提供的分区键来确定分片的位置。

> **Important**
>
> 切换到用户管理的分区键可以恢复每个分区键的顺序——具有相同分区键的记录会恢复在同一个分片上。切换之前，请确保您的生产者提供了有意义的分区密钥。如果在您切换到用户管理模式后，生产者仍在发送没有分区键的记录，那么 API 调用就会失败。

安全的迁移模式

在直播上切换记录分发策略会立即生效，并在大约一秒钟内传播到所有基础架构。没有通知、耗尽或暂停机制。在批次中处理记录的活跃消费者会立即看到行为变化。使用以下指南仔细规划迁移。

从用户管理迁移到服务管理（USER_PARTITION_KEY → 自动）

请按照以下步骤安全地在生产流上启用服务管理的记录分发：

1. 验证消费者兼容性-确保所有使用者都能处理GetRecords和SubscribeToShard响应中的null分区键。请先在非生产环境中测试。
2. 处理 KPL 聚合 — 如果有生产者使用 KPL 聚合，请确保 KPL 在写入服务管理的流时不会聚合记录。要么明确禁用聚合，要么升级到最新的 KPL 版本并向其授予kinesis:DescribeStreamSummary权限，使其能够自动检测流并禁用聚合。服务管理的流不支持聚合。请参阅带有服务管理流的 KPL 和 KCL。

3. 通知消费者团队 — 与直播中消费的所有团队进行协调。告知他们分区键将变为null（或忽略），并且不再保证分片间的记录顺序。
4. 在低流量时段切换 — 虽然交换机本身不会导致数据丢失，但在低流量时执行切换可以减少受行为变化影响的飞行中记录数量。
5. 启用服务管理分发-调用 UpdateStreamRecordDistributionStrategy API 将其设置RecordDistributionStrategy为。AUTO
6. 切换后进行监控 — 观看 Amazon 的直播 CloudWatch 指标：IncomingRecords、IncomingBytes（每个分片，以验证均匀分配）、WriteProvisionedThroughputExceeded（应降至零）和GetRecords.IteratorAgeMilliseconds（以验证消费者是否跟上）。

从服务管理迁移到用户管理（自动 → USER_PARTITION_KEY）

请按照以下步骤安全地将制作流切换回用户管理的分区密钥：

1. 验证所有生产者都提供分区密钥 — 切换后，省略该PartitionKey字段的PutRecords调用将失败PutRecord并出现验证错误。切换之前，请确认所有生产者都发送了有意义的分区密钥。
2. 验证使用者逻辑可以处理分区键的更改 — 正在处理null分区键的使用者现在将收到实际值。如果您的使用者逻辑使用分区键进行路由或分组，请验证它是否正确处理了转换。
3. 通知所有消费团队 — 通知团队将恢复按分区键哈希分发记录。具有相同分区键的相关记录将再次发送到同一个分片。
4. 在流量较低的时段切换 — 最大限度地减少受行为变化影响的飞行中记录数量。
5. 切换到用户管理-调用 UpdateStreamRecordDistributionStrategy API 将其设置RecordDistributionStrategy为。USER_PARTITION_KEY
6. 切换后监控 — 注意WriteProvisionedThroughputExceeded（如果分区键分布不均匀，可能会返回热分片）和使用者延迟指标。

迁移注意事项

- 即时生效 — 策略变更将在大约一秒钟内覆盖所有服务基础架构。该服务不会逐步推出变更或使用金丝雀机制。处理一批记录的活跃消费者可能会在批次中观察到行为变化。
- 没有消费者通知-当策略发生变化时，该服务不会向活跃的消费者发送通知。消费者不会被精疲力尽或保持沉默。无论当前的直播模式如何，都要设计您的消费类应用程序，使其能够优雅地处理null分区键。

- 控制台警告 — 当您在模式之间切换时，Kinesis Data Streams 控制台会显示警告横幅，提醒您服务管理模式取消了订购保障，用户管理模式要求制作者提供分区密钥。
- 可双面使用 — 您可以随时切换回来。如果您在迁移后发现问题，请立即将直播切换回先前的模式。
- 首先在非生产环境中进行测试 — 在应用于生产流之前，请务必在具有代表性流量模式的非生产环境中测试迁移。

错误处理

配置记录分发策略时可能会出现以下错误：

配置错误

错误	原因	Status
`InvalidArgumentException`	正在尝试在已配置的流上启用服务管理的记录分发。Service-managed 只有按需直播才支持记录分发。该服务返回一条类似于以下内容的消息：`RecordDistributionStrategy=AUTO is only supported for on-demand streams. Stream [StreamName] is using PROVISIONED mode.`	400
`ResourceNotFoundException`	指定的直播不存在。	400
`LimitExceededException`	并发流配置更改太多。请稍候，然后重试。	400

IAM 权限

要配置记录分发策略，IAM 委托人必须具有以下权限：

- `kinesis:CreateStream`— 使用服务管理的记录分发创建直播
- `kinesis:UpdateStreamRecordDistributionStrategy`— 更改现有直播的记录分发策略
- `kinesis:DescribeStreamSummary`— 查看当前的记录分发策略。更新后的 KPL 也需要此权限，它会调`DescribeStreamSummary`用检测直播是否使用服务管理的记录分发。将其授予任何

写入服务管理流的 KPL 制作者。有关更多信息，请参阅 向 KP DescribeStreamSummary L 授予许可。

将记录写入服务管理的流

当流使用服务管理的记录分发时，Kinesis Data Streams 会自动在分片之间分发记录。您可以使用PutRecord和 PutRecordsAPI 写入记录，而无需提供分区键。已经提供分区键的现有生产者无需修改即可继续工作——该服务会忽略提供的值并使用自己的分配算法。

> **Important**
>
> 如果您的制作者使用启用了聚合的 KPL（现有 KPL 版本中的默认设置），则在写入服务管理的流或升级到最新的 KPL 版本之前，您必须禁用聚合。服务管理的流不支持聚合，并且在取消聚合期间，KCL 使用者可以静默删除写入的聚合记录。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。

使用服务托管分发的生产者 API 行为

在直播上启用服务管理的记录分发后，以下行为适用于PutRecord和 PutRecordsAPI：

- PartitionKey：可选。如果提供，则在分片放置时会忽略该值。该服务使用内部算法分发记录。
- ExplicitHashKey— 如果提供，则在分片放置时会忽略该值，与`PartitionKey`。该服务使用内部算法分发记录。
- SequenceNumberForOrdering（PutRecord仅限）— 不得提供。 `PutRecord`包含此字段的请求会被拒绝`InvalidArgumentException`。订购不适用于服务管理的分发。上不存在此字段`PutRecords`。

> **Note**
>
> 启用服务管理模式后，提供分区键的现有生产者应用程序无需修改即可继续运行。该服务会忽略分区键值，这意味着您可以在生产流上启用此功能，而无需协调团队间的代码部署。
> 如果您使用 KPL 发送没有分区键的记录，请升级到最新的 KPL 版本。如果您直接使用 AWS SDK，则只要继续提供分区键，就无需更改 SDK；要省略分区键，必须升级到最新的 SDK 版本。有关更多信息，请参阅重要注意事项和带有服务管理流的 KPL 和 KCL。

示例

Example写一条没有分区键的记录 (AWS CLI)

```
aws kinesis put-record \
    --stream-name my-telemetry-stream \
    --data "eyJtZXRyaWMiOiAiY3B1X3V0aWxpemF0aW9uIiwgInZhbHVlIjogNzIuNX0="
```

Example写一条没有分区键的记录 (适用于 Java 的 AWS SDK)

```
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-telemetry-stream")
    .data(SdkBytes.fromUtf8String("{\"metric\": \"cpu_utilization\", \"value\":
 72.5}"))
    .build();

PutRecordResponse response = kinesisClient.putRecord(request);
```

Example写一批没有分区键的记录 (适用于 Java 的 AWS SDK)

```
List<PutRecordsRequestEntry> entries = new ArrayList<>();
entries.add(PutRecordsRequestEntry.builder()
    .data(SdkBytes.fromUtf8String("{\"event\": \"page_view\", \"page\": \"/home\"}"))
    .build());
entries.add(PutRecordsRequestEntry.builder()
    .data(SdkBytes.fromUtf8String("{\"event\": \"page_view\", \"page\": \"/products
\"}"))
    .build());

PutRecordsRequest request = PutRecordsRequest.builder()
    .streamName("my-telemetry-stream")
    .records(entries)
    .build();

PutRecordsResponse response = kinesisClient.putRecords(request);
```

Example具有分区键的现有生产者（继续工作）

```
// This producer code continues to work without modification.
// The partition key is ignored by the service when service-managed
// record distribution is enabled.
```

```
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-telemetry-stream")
    .partitionKey("device-12345")
    .data(SdkBytes.fromUtf8String("{\"temp\": 22.1}"))
    .build();

PutRecordResponse response = kinesisClient.putRecord(request);
```

错误处理

以下错误是向服务管理的流写入记录所特有的：

服务管理直播的制作人错误

错误	原因	Status
InvalidArgumentException	SequenceNumberForOrdering 已在PutRecord 请求中提供。写入服务管理的流时，不得包含此字段，因为排序不适用。（PutRecords 没有SequenceNumberForOrdering 字段。）提供的会ExplicitHashKey 被忽略并且不会导致错误。该服务返回一条类似于以下内容的消息： Stream has SERVICE_DEFINED PartitionKey enabled. SequenceNumberForOrdering must not be provided.	400

迁移现有生产者

在直播上启用服务管理的记录分发时，您无需修改现有的制作者。提供分区键的现有创建器继续运行——该服务会忽略分区键值并使用内部算法分发记录。

要随着时间的推移迁移生产者：

1. 在直播中启用服务管理的记录分发（请参阅配置服务管理的记录分发）。
2. 通过监控 CloudWatch 指标（参见和按分片），验证您的直播是否正在接收记录IncomingRecords并IncomingBytes均匀分配记录。
3. （可选）更新您的生产者代码以随着时间的推移删除分区键逻辑。这不是该功能正常运行所必需的，但可以简化您的代码库。

> **Important**
>
> 如果您的制作者使用 KPL 聚合，请注意任何 KPL 版本的服务管理流都不支持聚合。必须先禁用聚合，然后才能启用服务管理的记录分发。在取消聚合期间，KCL 使用者可以静默删除写入服务管理流的聚合记录，而不会出现任何错误。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。

最佳实践

- PutRecords 用于批处理-继续使用 PutRecordsAPI 进行批量写入。无论采用何种记录分发策略，批处理都会减少 API 调用开销并提高吞吐量。
- 移除 SequenceNumberForOrdering-此PutRecord字段被服务管理的流拒绝（PutRecords没有这样的字段）。提供的内容会ExplicitHashKey被忽略而不是拒绝，但您也应该将其删除，因为它没有任何效果。更新您的生产者代码以删除这些字段。
- 监控分片利用率-使用 CloudWatch指标来验证记录是否在分片之间均匀分布。监控WriteProvisionedThroughputExceeded以确认没有出现热分片。

从服务管理的直播中读取记录

从服务管理的流中读取数据的消费者使用与具有用户管理分区键的流相同的 API。主要区别在于该PartitionKey字段在响应中的显示方式。无论分发策略如何，消费者都会继续按顺序读取分片中的记录。

GetRecords API 行为

使用 GetRecordsAPI 从服务管理的流中读取记录时，响应包括每条记录的以下行为：

- PartitionKey— null 如果创建者在写入记录时未提供分区键，则返回。如果生产者提供了分区键（在分片放置时被忽略），则原始值将按原样返回。
- SequenceNumber— 正常返回。无论分配策略如何，序列号都会继续分配给每条记录。
- ApproximateArrivalTimestamp— 正常返回。
- 数据-正常返回。记录数据不受分发策略的影响。

Example GetRecords 服务管理流的响应

```
{
    "Records": [
        {
            "SequenceNumber": "496541083111687359213349271857026644210817126 46",
            "ApproximateArrivalTimestamp": 1719849600.123,
            "Data": "eyJtZXRyaWMiOiAiY3B1X3V0aWxpemF0aW9uIiwgInZhbHVlIjogNzIuNX0=",
            "PartitionKey": null,
            "EncryptionType": "KMS"
        },
        {
            "SequenceNumber": "496541083111687359213349271868917755321928237 57",
            "ApproximateArrivalTimestamp": 1719849600.456,
            "Data": "eyJldmVudCI6ICJwYWdlX3ZpZXciLCAicGFnZSI6ICIvaG9tZSJ9",
            "PartitionKey": "device-12345",
            "EncryptionType": "KMS"
        }
    ],
    "NextShardIterator": "EXAMPLEShardIterator1234567890EXAMPLE",
    "MillisBehindLatest": 0
}
```

> **Note**
>
> 在上面的示例中，第一条记录是在没有分区键的情况下写入的（制作者省略了分区键），PartitionKey也是如此null。第二条记录是用分区键写入的（出于分发目的，该分区键被忽略），因此将返回原始值。

SubscribeToShard API 行为

在 SubscribeToShard API 中使用增强型扇出功能时，其行为与相同。GetRecords中每条记录中的PartitionKey字段SubscribeToShardEvent遵循相同的规则：

- null如果生产者未提供分区键，则返回。
- 如果生产者提供了原始值，则返回原始值（即使在分片放置时忽略了该值）。

使用者注意事项

从服务管理的直播中消费时，请记住以下几点：

- 处理空分区键-您的使用者应用程序应优雅地处理null分区键值。请勿将服务管理流中的分区键用于任何业务逻辑。
- 分片迭代保持不变 — 您继续使用分片迭代器来读取记录。遍历分片、处理重新分片和管理检查点的机制保持不变。
- 没有排序假设 — 不要假设服务管理流中的记录之间存在任何排序关系。来自同一个生产者或商业实体的记录可能出现在不同的分片上。
- 下游服务按原样运行 — AWS Lambda Amazon Data Firehose、适用于 Apache Flink 的亚马逊托管服务以及从 Kinesis Data Streams 使用的 AWS 其他服务无需修改即可继续运行。但是，如果您的下游处理逻辑依赖分区键进行路由或分组，则应查看该逻辑。

监控服务管理直播的使用情况

使用以下 Amazon CloudWatch 指标来监控服务托管直播的使用情况：

- GetRecords.Records— 每次 GetRecords 呼叫检索的记录数。使用它来验证消费是否正常进行。
- GetRecords.IteratorAgeMilliseconds— 消费者距离最新记录有多远。使用它来检测消费者延迟。
- ReadProvisionedThroughputExceeded— 分片限制的 GetRecords 调用次数。这种情况不应根据分配策略而改变。

带有服务管理流的 KPL 和 KCL

想要将没有分区键（空分区键）的记录发送到服务管理流的客户应升级到最新的 KPL 版本。继续提供分区密钥的客户不需要更改 KCL，但他们必须确保 KPL 在写入服务管理的流时不会聚合记录。

> **Important**
>
> 默认情况下，现有 KPL 版本会汇总记录。服务管理的流不支持聚合，而且 KCL 使用者可以在解聚期间以静默方式删除写入的聚合记录，而不会出现生产者或使用者出现错误。如果您使用 KPL 写入服务管理的流，则要么升级到最新的 KPL 版本并向其授予kinesis:DescribeStreamSummary权限（以便它检测流并停止聚合），要么明确禁用聚

合。有关详细信息，请参阅 向 KP DescribeStreamSummary L 授予许可 和 服务管理的流不支持聚合。

升级 KPL 以发送空分区密钥

要在不向服务管理的流提供分区密钥的情况下发送记录，必须升级到最新的 KPL 版本。更新后的 KPL 可以检测流是否使用服务管理的记录分发 (AUTO)，并相应地处理空分区键。

使用 Kinesis 制作器库的客户只需在 Amazon Kinesis 制作器 GitHub 库中升级到最新版本，即可从此功能中受益。

更新后的 KPL 通过调用 `DescribeStreamSummary` API 来检测直播的记录分发策略。向 KPL 使用的身份授予`kinesis:DescribeStreamSummary`权限。没有它，KPL 就无法检测到服务管理的数据流，而是回退到聚合记录的状态。有关更多信息，请参阅 向 KP DescribeStreamSummary L 授予许可。

向 KP DescribeStreamSummary L 授予许可

更新后的 KPL 会调用 `DescribeStreamSummary` API 来检测直播是否使用服务管理的记录分发 () AUTO。确保您的制作人使用的 IAM 委托人拥有直播`kinesis:DescribeStreamSummary`权限。此权限包含在 KPL 的最低权限策略中；如果您遵循该政策，则无需进行任何更改。

> **Important**
>
> 如果 KPL 无法调用`DescribeStreamSummary`，则无法可靠地检测到直播是否使用服务管理的记录分发。例如，如果未授予`kinesis:DescribeStreamSummary`权限，就会发生这种情况。在这种情况下，KPL 会退回到汇总记录的状态。服务管理的流不支持聚合，因此 KCL 使用者可以在解聚期间以静默方式删除写入的聚合记录，而不会出现生产者或使用者出现错误。在 KPL 制作者写入的直播上启用服务管理记录分发之前，请先授予`kinesis:DescribeStreamSummary`权限。

现有生产者和消费者无需进行任何更改

如果您继续在PutRecord和 PutRecordsAPI 调用中提供分区密钥，并且您的创建器不使用 KPL 聚合，则无需更改 KPL 或 KCL。在流上启用服务管理的记录分发后，您现有的生产者和使用者应用程序无需修改即可继续运行，该服务会忽略您提供的分区键值，并使用内部算法分发记录。

如果您的制作器使用 KPL 聚合（现有 KPL 版本中的默认值），则在写入服务管理的流之前，您必须禁用聚合或升级 KPL。请参阅服务管理的流不支持聚合。

无论您是否发送空分区键，KCL 都不需要更改。KCL 继续像今天一样按顺序从分片中读取记录。

服务管理的流不支持聚合

在任何 KPL 版本中，配置了服务管理记录分发的流都不支持 KPL 聚合。写入时不会拒绝聚合记录。失败以以下两种方式之一浮出水面，具体取决于 KPL 是否将写入视为成功：

- 对消费者进行静默丢弃 — KPL 聚合根据分区键哈希将多个用户记录分组为单个 Kinesis Data Streams 记录，并期望这些记录落在 KPL 预测的分片上。在服务管理的流上，该服务使用自己的算法而不是分区键来放置记录，这会打破 KPL 和 KCL 聚合合约。在去聚合期间，KCL 使用者会默默删除写入的聚合记录，不会出现任何错误。
- 生成器报告写入失败 — 当 KPL 发送的汇总记录未提交到 KPL 预测的分片时，KPL 会重试这些记录大约 30 秒，然后通过回调向您的应用程序报告写入失败。

要避免这两种情况，请禁用聚合，或者升级到最新的 KPL 版本并授予`kinesis:DescribeStreamSummary`权限，使其能够检测服务管理的流并自动禁用聚合。您也可以将其设置`setRecordDistributionStrategyDefault`为，`AUTO`这样 KPL 在发现直播策略之前就不会进行汇总。

> **Note**
>
> 对于按需流，KPL 聚合不提供任何成本优势，因为定价基于吞吐量而不是 PUT 请求的数量。如果要使用服务管理的记录分发，请考虑禁用聚合。

KCL 注意事项

如果您使用 Kinesis 客户端库 (KCL) 从服务管理的直播中进行消费，则无需更改 KCL。KCL 继续像今天一样按顺序处理来自分片的记录，并且不加修改地处理分区键为空的记录。

KCL 本身不需要从服务管理的流中使用任何更改，也不需要升级 KCL。但是，如果生产者将 KPL-aggregated记录写入服务管理的流（例如，因为未禁用聚合），KCL 解聚路径可以静默删除这些记录，因为当服务通过自己的算法放置记录时，聚合合同不成立。确保生产者禁用聚合，或者升级 KPL 并向其授予`kinesis:DescribeStreamSummary`权限，这样聚合记录就不会写入服务管理的流中。

Summary

KPL 和 KCL 兼容性摘要

场景	所需操作	说明
发送分区键为空的记录	升级 KPL	升级到最新的 KPL 版本以支持在没有分区键的情况下发送记录
使用分区键发送记录（现有行为）	无更改	现有的 KPL 按原样运作。该服务会忽略服务管理流上的分区键
KCL 消费者	无更改	无论唱片是如何制作的，KCL 都按原样工作
升级版 KPL 制作人	格兰特 `kinesis:DescribeStreamSummary`	更新后的 KPL 调用`DescribeStreamSummary` 以检测服务管理的流。如果没有此许可，KPL 就会退回到聚合状态。服务管理的流不支持聚合，因此 KCL 使用者可以在取消聚合期间以静默方式删除写入的聚合记录。
服务管理流上的 KPL 聚合	不支持	在任何 KPL 版本中，服务管理的流都不支持聚合。在取消聚合期间，KCL 使用者可以静默删除已写入的聚合记录。禁用聚合或升级 KPL。

依赖整合案例研究

Service-managed 记录分发适用于向亚马逊 Kinesis Data Streams 制作或消费的 AWS 服务。本节介绍当直播使用 AWS 服务管理的记录分发时，公共服务集成的行为以及您需要做的任何更改。

AWS Lambda 使用服务管理流触发器

AWS Lambda 事件源映射无需修改即可与服务管理的流配合使用。Lambda 继续轮询分片并使用批量记录调用您的函数。事件记录中的`partitionKey`字段可能`null`用于写入没有分区键的记录。确保您的 Lambda 函数可以正常处理空分区键。

Amazon Data Firehose 从服务管理的流中消费

从配置了服务管理记录分发的 Kinesis Data Streams 流中获取数据的 Amazon Data Firehose 传输流可以继续运行，无需进行任何更改。Firehose 从分片中读取记录并将其传送到目的地，无论记录如何分布在分片上。无需更改任何配置。

当记录写入没有分区键的服务管理流时，Amazon Data Firehose 会将分区键显示为空字符串 "" ()，null而不是在以下面向客户的案例中：

- 转换 Lambda 函数输入-如果您使用 Lambda 函数转换记录，则对于没有分区键写入的记录，Lambda 输入事件partitionKey中的字段为空字符串。这样可以避免与对字段执行非空验证的常见解析库（例如 Lambda Powertoo AWS ls）存在兼容性问题。partitionKey
- Snowflake 目标元数据-如果您传送到 Snowflake 目标，则分区键元数据列将包含一个空字符串，用于写入没有分区键的记录。

> **Note**
>
> 如果您选择使用服务管理记录分发但不提供分区键，则在这些位置应使用空字符串 ("")，而不是您之前提供的分区键值。使用服务管理的记录分发时，不要依赖分区键值作为业务逻辑。

适用于 Apache Flink 的亚马逊托管服务 Flink 从服务管理的直播中消费

适用于 Apache Flink 的亚马逊托管服务是一个运行您的 Flink 应用程序的托管平台；您选择的 Kinesis 连接器决定了与服务托管流的兼容性。

- Current connector (**flink-connector-aws-kinesis-streams**)-无需修改即可从服务管理的流中消费，包括使用空分区键写入的记录。如果您的应用程序使用分区键作为键控状态或窗口，请注意分区键可能是null；请检查您的逻辑，确保其处理正确性不依赖于分区键值。
- 旧版连接器 (**flink-connector-kinesis**)-不建议用于服务管理的流。传统连接器使用 KPL 进行聚合，服务管理的流不支持聚合。此外，读取分区键的自定义反序列化器必须防范。null如果您使用旧版连接器，请在启用服务管理记录分发之前迁移到当前连接器。

CloudWatch 带有服务管理流的指标流

亚马逊 CloudWatch 指标流可以向 Kinesis Data Streams 流传输指标数据。指标流创建者设置分区键，其使用者读回分区键以进行重复数据删除和路由。在服务管理的流中，分区键仅在分片放置时才会

被忽略，并且仍按提供的方式返回给使用者，因此读回密钥的重复数据删除逻辑继续起作用。无需进行任何更改。请注意，服务管理的记录分发仅适用于按需流；使用预配置流的指标流继续使用用户管理的分区键。

CloudFront 服务管理流的实时日志

亚马逊 CloudFront 可以通过其原生集成将实时日志直接发布到Kinesis Data Streams流。Amazon 为流 CloudFront 生成日志记录；它不使用 KPL，也没有依赖于将分区键与特定分片关联的聚合逻辑。尽管 Amazon CloudFront 生成的记录包含分区键，但该值不用于在服务管理的流上进行路由。在接收 Amazon CloudFront 实时日志的流上启用服务管理记录分发无需进行任何更改，也不会影响日志传输。有关亚马逊 CloudFront实时日志的更多信息，请参阅《亚马逊 CloudFront 开发者指南》中的Real-time 日志。

带有服务管理流的 Amazon EMR

亚马逊 EMR 使用 Spark 连接器对 Kinesis Data Streams 流进行读取和写入。连接器没有依赖于分区键值来读取或写入数据的逻辑。当 Amazon EMR 从服务管理的直播中消费时，无需进行任何更改。

> **Important**
>
> 当 Amazon EMR 向流生成内容时，Spark 连接器会在启用聚合的情况下通过 KPL 进行写入。服务管理的流不支持聚合。使用启用了聚合功能的 Spark 连接器的 Amazon EMR 创建器仍然可以写入服务管理的流，但是 KCL 使用者可以在取消聚合期间以静默方式删除聚合记录，而不会出现任何错误。要安全地从 Amazon EMR 生成到服务托管流，请使用允许您在任务配置中禁用 KPL 聚合或禁用聚合的连接器版本。有关更多信息，请参阅 带有服务管理流的 KPL 和 KCL。

带有服务托 EventBridge 管直播的 Amazon Pipes

Amazon Pip EventBridge es 可以使用 Kinesis Data Streams 流作为目标（Pipes 使用 API 写入记录）和源（Pipes 轮询器读取记录并转发记录）。`PutRecords` EventBridge Pipes 直接使用 AWS SDK，没有 KPL 或 KCL，也没有聚合。

- Service-managed 以直播为目标 — 无需更改即可运行。 EventBridge 管道总是发送一个非空分区键（配置的值或管道执行 ID）。在服务管理的流中，提供的密钥将被忽略以进行路由，记录在分片之间均匀分布。
- Service-managed 流作为源 — Pi EventBridge pes 使用服务管理流中的记录，包括在没有分区键的情况下写入的记录。确保管道中的任何下游目标或扩充步骤都不需要分区键值。

带有 Kinesis 数据流的亚马逊 DynamoDB

您可以将 Amazon DynamoDB 变更数据捕获 (CDC) 记录流式传输到您拥有的 Kinesis Data Streams 流。DynamoDB 始终设置非空分区键（项目密钥的哈希值），并且永远不会从流中读回记录。在服务管理的直播中，DynamoDB 无需更改即可继续运行。

请注意一个行为变化：在用户管理的直播中，单个项目的所有更改记录都映射到同一个分片，因为它们共享一个分区键。在服务管理的流上，路由时会忽略分区键，因此单个项目的更改记录会分布在各个分片中。DynamoDB 不保证根据分片放置情况进行排序。消费者使用该`ApproximateCreationDateTime`字段对记录进行排序，因此这种行为不会影响按时间戳对记录进行排序的消费者的正确性。

从服务管理的直播中提取亚马逊 Redshift 流媒体

Amazon Redshift 流媒体提取允许你在 Kinesis Data Streams 流上创建物化视图。实例化视图架构包括一个分区键列。当您从服务管理的流中提取内容时，未使用分区键写入的记录在实例化视图中显示为空分区键值。无需进行任何更改，并且空分区和非空分区键记录均已成功载入。

带有服务托管直播的 Amazon Kinesis 代理

Amazon Kinesis Agent 是一款仅供制作人使用的应用程序，它使用 API 读取文本文件并将记录写入流。`PutRecords`它始终设置非空分区键（默认为随机值，或记录的 MD5 哈希），并直接使用 AWS SDK 而不使用 KPL 或聚合。在服务管理的流中，提供的分区键会被忽略以进行路由，代理的写入会保持不变。无需进行任何更改。

带有服务管理流的 Amazon EventBridge 事件总线

Amazon EventBridge 事件总线可以将事件作为目标传输到 Kinesis Data Streams 流，并将来自流的事件作为源进行消费。作为生产者，事件总线始终发送非空分区密钥（当未配置 Kinesis 参数时，它会生成一个随机值），并在没有 KPL 或聚合的情况下直接使用 AWS SDK。作为使用者，它没有依赖于分区键值或分片位置的逻辑。在服务管理的流中，提供的分区键会被忽略以进行路由，并且会干净地使用空分区键记录。无需进行任何更改。

使用服务管理的流式传输表和 Amazon S3 交付

您可以将记录从 Kinesis Data Streams 流传输到流式传输表和亚马逊简单存储服务 (Amazon S3) 的交付目的地。在服务管理的流中，不使用分区键写入的记录与包含分区键的记录一起正常传送。无需进行任何更改。

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。你可以使用 适用于 Java 的 AWS SDK 和 Kinesis 制作器库 (KPL) 为 Kinesis Data Streams 构建制作器。

如果是首次使用 Kinesis Data Streams，请先熟悉 什么是 Amazon Kinesis Data Streams？ 和 使用 AWS 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 开发制作人 适用于 Java 的 AWS SDK
- 使用 Kinesis 代理写入 Amazon Kinesis Data Streams
- 使用其他写入 Kinesis Data Streams AWS 务
- 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。

您可以通过 Amazon CloudWatch 监控 KPL。有关更多信息，请参阅 使用亚马逊监控 Kinesis 制作人库 CloudWatch。

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 AWS Glue 架构注册表一起使用
- 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 是一个高度可配置的库 easy-to-use，可帮助您写入 Kinesis 数据流。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，以提供对制作人绩效的可见性

请注意，KPL 与中提供的 Kinesis Data Streams API 不同。AWS SDKsKinesis 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

您可以使用 CloudWatch 亚马逊和 KPL 收集、监控和分析您的 Kinesis Data Streams 制作者。KPL 代表您向 CloudWatch 发送吞吐量、错误和其他指标，并且可以配置为在流、分片或生产者级别进行监控。

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 AWS SDK 中的异步客户端行为相同。

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 AWS 开发工具包。有关将软件开发工具包与 Kinesis AWS Data Streams 配合使用的更多信息，使用 Amazon Kinesis Data Streams API 开发制作人 适用于 Java 的 AWS SDK请参阅。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。

早期的 Linux 发行版的用户可以使用随源代码一起提供的编译说明来构建 KPL。 GitHub要从中下载 KPL GitHub，请参阅 Amazon Kinesis 制作人库。

> **Important**
>
> 亚马逊 Kinesis Producer Library (KPL) 0.x 将于 2026 年 1 月 30 日 end-of-support上线。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

本主题提供了将您的消费者从 KPL 0.x 迁移到 KPL 1.x 的 step-by-step说明。KPL 1.x 引入了对 适用于 Java 的 AWS SDK 2.x 的支持，同时保持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 AWS SDK 2.x
   - 用于依赖项管理的 Maven 或 Gradle

2. 添加依赖关系

   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 AWS SDK 2.x 并使用以开头的更新后的软件包名称`software.amazon.kinesis`，而之前的 KPL 中的软件包名称以开头。`com.amazonaws.services.kinesis`

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:
import com.amazonaws.services.kinesis.producer.Attempt;	import software.amazon.kinesis.producer.Attempt;
import com.amazonaws.services.kinesis.p BinaryToHexConverter;	导入软件.amazon.kinesis.producer。BinaryToHexConverter;
import com.amazonaws.services.kinesis.p CertificateExtractor;	导入软件.amazon.kinesis.producer。CertificateExtractor;
import com.amazonaws.services.kinesis.producer.Daemon;	import software.amazon.kinesis.producer.Daemon;
import com.amazonaws.services.kinesis.p DaemonException;	导入软件.amazon.kinesis.producer。DaemonException;
import com.amazonaws.services.kinesis.p FileAgeManager;	导入软件.amazon.kinesis.producer。FileAgeManager;
import com.amazonaws.services.kinesis.p FutureTimedOutException;	导入软件.amazon.kinesis.producer。FutureTimedOutException;
import com.amazonaws.services.kinesis.p GlueSchemaRegistrySerializerInstance;	导入软件.amazon.kinesis.producer。GlueSchemaRegistrySerializerInstance;
import com.amazonaws.services.kinesis.p HashedFileCopier;	导入软件.amazon.kinesis.producer。HashedFileCopier;
import com.amazonaws.services.kinesis.p IKinesis制片人；	导入软件.amazon.kinesis.producer。 IKinesis制片人；
import com.amazonaws.services.kinesis.p IrrecoverableError;	导入软件.amazon.kinesis.producer。IrrecoverableError;

进行如下替换：	借助:
import com.amazonaws.services.kinesis.p KinesisProducer;	导入软件.amazon.kinesis.producer。 KinesisProducer;
import com.amazonaws.services.kinesis.p KinesisProducerConfiguration;	导入软件.amazon.kinesis.producer。 KinesisProducerConfiguration;
import com.amazonaws.services.kinesis.p LogInputStreamReader;	导入软件.amazon.kinesis.producer。 LogInputStreamReader;
import com.amazonaws.services.kine sis.producer.Metric;	import software.amazon.kinesis.producer.Met ric;
import com.amazonaws.services.kinesis.p ProcessFailureBehavior;	导入软件.amazon.kinesis.producer。 ProcessFailureBehavior;
import com.amazonaws.services.kinesis.p UnexpectedMessageException;	导入软件.amazon.kinesis.producer。 UnexpectedMessageException;
import com.amazonaws.services.kinesis.p UserRecord;	导入软件.amazon.kinesis.producer。 UserRecord;
import com.amazonaws.services.kinesis.p UserRecordFailedException;	导入软件.amazon.kinesis.producer。 UserRecordFailedException;
import com.amazonaws.services.kinesis.p UserRecordResult;	导入软件.amazon.kinesis.producer。 UserRecordResult;
import com.amazonaws.services.kine sis.producer.protobuf.Messages;	import software.amazon.kinesis.producer.pro tobuf.Messages;
import com.amazonaws.services.kine sis.producer.protobuf.Config;	import software.amazon.kinesis.producer.pro tobuf.Config;

4. 更新 AWS 凭证提供程序类的导入语句

   迁移到 KPL 1.x 时，必须将 KPL 应用程序代码中基于 适用于 Java 的 AWS SDK 1.x 的导入包和类更新为基于 2.x 的相应包和类。 适用于 Java 的 AWS SDK KPL 应用程序中通常导入的是凭证

提供程序类。有关凭证提供程序变更的完整列表，请参阅 适用于 Java 的 AWS 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;
```

如果您导入任何其他基于 适用于 Java 的 AWS SDK 1.x 的凭证提供程序，则必须将其更新为 适用于 Java 的 AWS SDK 2.x 的等效凭证提供程序。如果您没有 classes/packages 从 适用于 Java 的 AWS SDK 1.x 中导入任何内容，则可以忽略此步骤。

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 AWS SDK 2.x 凭证提供程序。如果您`KinesisProducerConfiguration`通过覆盖默认凭证提供程序来传递中 适用于 Java 的 AWS SDK 1.x 的凭证提供程序，则必须使用 适用于 Java 的 AWS SDK 2.x 凭据提供程序对其进行更新。有关凭证提供程序变更的完整列表，请参阅 适用于 Java 的 AWS 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 AWS 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 AWS 区域。

有关迁移到 ATS 的信息，请参阅如何为 AWS迁移到自己的证书颁发机构做好准备。

如果您遇到问题，需要技术支持，请通过 AWS 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 接口的源代码根据亚马逊公共许可发布，可在亚马逊 Kinesis Producer Library GitHub 上找到。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。但如果您要检查结果，可使用从 addUserRecord 返回的 Future 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。

属性文件应使用中的名称来指定参数 PascalCase。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，请参阅上的配置属性文件示例 GitHub。

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
- GetRecords 直接使用

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。

GetRecords 直接使用

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 AWS 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 AWS Glue 架构注册表一起使用

您可以将 Kinesis 数据流与 AWS Glue 架构注册表集成。 AWS 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。 AWS Glue Schema Registry 使您能够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 AWS 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，只有使用用 Java 实现的 KPL 生成器的 Kinesis 数据流支持 Kinesis 数据流 AWS Glue 和架构注册表集成。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 “用例：将 Amazon Kinesis Data Streams 与 Glu KPL/KCL e 架构注册表集成” 中的 “使用库与 AWS数据交互” 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 AWS SDK 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 库 (KPL) 在使用代理服务器的环境 AWS 中收集和发送数据，则您的应用程序将需要配置 KPL 代理。KPL 是一个基于 Kinesi AWS s 软件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，请参见 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 生产者库 (KPL) 的版本生命周期政策。 AWS 定期为 KPL 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 KPL 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 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)-在此阶段，将完全支持主要版本。 AWS 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 AWS 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。
- E nd-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
KPL 0.x	维护模式	2015-06-02	2025-04-17	2026-01-30

主要版本	当前阶段	发行日期	维护模式日期	End-of-support 日期
KPL 1.x	正式发布	2024-12-15	--	--

使用 Amazon Kinesis Data Streams API 开发制作人 适用于 Java 的 AWS SDK

你可以使用 Amazon Kinesis Data Streams API 和适用于 Java 的软件开发工具包 AWS 来开发制作者。如果是首次使用 Kinesis Data Streams，请先熟悉 什么是 Amazon Kinesis Data Streams？ 和 使用 AWS 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AWS 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。有关所有可用内容的更多信息 AWS SDKs，请参阅开始使用 Amazon Web Services 进行开发。

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 AWS 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

- 使用添加多条记录 PutRecords
- 使用添加单条记录 PutRecord

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

使用添加多条记录 PutRecords

PutRecords 操作可在一个请求中向 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，直到达到请求限制。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。

使用时处理故障 PutRecords

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": "XzxkYXRhPl8w",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "AbceddeRFfg12asd",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "KFpcd98*7nd1",
     "PartitionKey": "partitionKey3"
        }
    ],
    "StreamName": "myStream"
}
```

Example PutRecords 响应语法

```
{
    "FailedRecordCount”: 1,
    "Records": [
        {
     "SequenceNumber": "21269319989900637946712965403778482371",
     "ShardId": "shardId-000000000001"

        },
        {
     “ErrorCode":”ProvisionedThroughputExceededException”,
     “ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream
 exampleStreamName under account 111111111111."
```

```
        },
        {
     "SequenceNumber": "21269319989999637946712965403778482985",
     "ShardId": "shardId-000000000002"
        }
    ]
}
```

处理不成功的请求可包含在后续 PutRecords 请求中。首先，查看 putRecordsResult 中的 FailedRecordCount 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 失败处理程序

```
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);
}
```

使用添加单条记录 PutRecord

对 PutRecord 的每次调用对一个记录起作用。应首选 PutRecords 中描述的 使用添加多条记录 PutRecords 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 `putRecord` 的结果调用 `getSequenceNumber`。

`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 AWS Glue 架构注册表与数据交互

您可以将 Kinesis 数据流与 AWS Glue 架构注册表集成。 AWS 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。 AWS Glue Schema Registr end-to-end y 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 AWS Glue Schema Registry。设置此集成的方法之一是通过 Java SDK 中 AWS 提供的`PutRecords`和 `PutRecord` Kinesis Data APIs Streams。

有关如何使用 PutRecords PutRecord 和 Kinesis Data Streams 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 APIs 数据流与 Glue 架构注册表集成中的 “使用 Kinesis 数据流与数据交互” 部分。 APIs AWS

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。它还会发布Amazon CloudWatch 指标，以帮助您更好地监控直播过程并对其进行故障排除。

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
- 常见问题解答

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您的 AWS 证书：
  - 当您启动您的 EC2 实例时指定该 IAM 角色。
  - 在配置代理时指定 AWS 凭据（请参阅 awsAccessKeyID 和awsSecretAccess密钥）。
  - 编辑`/etc/sysconfig/aws-kinesis-agent`以指定您的地区和 AWS 访问密钥。
  - 如果您的 EC2 实例位于不同的 AWS 账户中，请创建一个 IAM 角色来提供对 Kinesis Data Streams 服务的访问权限，并在配置代理时指定该角色（参见 A ssumeLearn 和 ID）。assumeRoleExternal使用前面的方法之一来指定另一个账户中有权担任此角色的用户的 AWS 证书。
- 您指定的 IAM 角色或 AWS 证书必须具有执行 Kinesis Data Streams 操作的权限，代理才能将数据发送到您的数据PutRecords流。如果您为代理启用 CloudWatch 监控，则还需要执行该 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 资源的访问使用亚马逊监控 Kinesis Data Streams 代理的运行状况 CloudWatch、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
```

要设置代理，请使用以下方法 GitHub

1. 从 awlab amazon-kinesis-agent s/ 下载代理。
2. 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	说明
assumeRoleARN	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 角色委派跨 AWS 账户访问权限。
assumeRoleExternalId	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。
awsAccessKeyId	AWS 覆盖默认凭证的访问密钥 ID。此设置优先于所有其他凭证提供程序。
awsSecretAccessKey	AWS 覆盖默认凭证的密钥。此设置优先于所有其他凭证提供程序。
cloudwatch.emitMetrics	CloudWatch 如果已设置，则允许代理向其发送指标 (true)。 默认：True
cloudwatch.endpoint	的区域终端节点 CloudWatch。 默认值：monitoring.us-east-1.amazonaws.com
kinesis.endpoint	Kinesis Data Streams 的区域端点。 默认值：kinesis.us-east-1.amazonaws.com

流配置设置如下。

配置设置	说明
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)
maxBufferSizeRecords	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 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

配置设置	说明
multiLine StartPattern	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。
partition KeyOption	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）
truncated RecordTer minator	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，您可以为 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": "CSVTOJSON",
                    "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
```

常见问题解答

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 Agent 的减速速度越来越大 and/or **RecordSendErrors**？

这通常是由于 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，请参见 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。

使用其他写入 Kinesis Data Streams AWS 务

以下 AWS 服务可以直接与 Amazon Kinesis Data Streams 集成，将数据写入 Kinesis 数据流。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

- 使用写入 Kinesis Data Streams AWS Amplify
- 使用 Amazon Aurora 写入 Kinesis Data Streams
- 使用亚马逊写入 Kinesis Data Streams CloudFront
- 使用亚马逊 CloudWatch 日志写入 Kinesis Data Streams
- 使用 Connect Customer 写入 Kinesis Data Streams
- 使用写入 Kinesis Data Streams AWS Database Migration Service
- 使用 Amazon DynamoDB 写入 Kinesis Data Streams
- 使用亚马逊写入 Kinesis Data Streams EventBridge
- 使用写入 Kinesis Data Streams AWS IoT Core
- 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

使用写入 Kinesis Data Streams AWS Amplify

您可以使用 Amazon Kinesis Data Streams 从使用 AWS 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《AWS 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。

使用亚马逊写入 Kinesis Data Streams CloudFront

您可以将 Amazon Kinesis Data Streams CloudFront 与实时日志一起使用，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 数据流使用器，或者使用 Amazon Data Firehose 将日志数据发送到亚马逊 S3、亚马逊 Redshift、亚马逊 OpenSearch 服务或第三方日志处理服务。

有关更多信息，请参阅《Amazon CloudFront 开发者指南》中的Real-time 日志。

使用亚马逊 CloudWatch 日志写入 Kinesis Data Streams

您可以使用 CloudWatch 订阅来访问来自 Amazon CloudWatch Logs 的日志事件的实时提要，并将其传输到 Kinesis 数据流进行处理、分析并加载到其他系统。

有关更多信息，请参阅 Amazon Log CloudWatch s 用户指南中的通过订阅Real-time 处理日志数据。

使用 Connect Customer 写入 Kinesis Data Streams

您可以使用 Kinesis Data Streams 从 Connect 客户实例中实时导出联系记录和代理事件。您还可以启用来自 Connect 客户档案的数据流，以自动接收 Kinesis 数据流中有关创建新资料或更改现有资料的更新。

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。

使用写入 Kinesis Data Streams AWS Database Migration Service

您可以使用将数据迁移 AWS Database Migration Service 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《AWS 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。

使用亚马逊写入 Kinesis Data Streams EventBridge

使用 Kinesis Data Streams，您可以将 API 调用事件 AWS EventBridge 发送到数据流、构建使用者应用程序和处理大量数据。您还可以使用 Kinesis Data Streams 作为 Pipes EventBridge 中的目标，并在可选的筛选和扩充之后从可用来源之一传送记录。

有关更多信息，请参阅亚马逊 EventBridge 用户指南中的向亚马逊 Kinesis 直播发送事件和 EventBridge Pipes。

使用写入 Kinesis Data Streams AWS IoT Core

您可以使用物联网规则操作从 IoT Core 中的 MQT AWS T 消息中实时写入数据。 AWS 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 AWS 服务。

有关更多信息，请参阅《AWS 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
- 甲骨文 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。

甲骨文 GoldenGate

Oracle GoldenGate 是一款软件产品，允许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 GoldenGate Oracle 写入 Kinesis Data Streams 的更多信息，请参阅使用 Oracle 将数据复制到 Kinesis 数据流。 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 是一个完整的内存平台 end-to-end，用于实时收集、筛选、转换、丰富、聚合、分析和交付数据。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，在使用单记录操作 PutRecord 时，吞吐量限制计算会向上舍入到最接近的 KB 值，而多记录操作 PutRecords 对各个调用中记录的累加和进行舍入。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。处理延迟的策略包括以下内容：如果客户可以对记录进行微批处理/缓冲，请在使用单记录操作之前使用 Amazon Kinesis Producer 库（具有高级聚合逻辑）PutRecords、多记录操作或将记录聚合到更大的文件中。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 AWS SDK 和其他 SDKs 包括异步客户端，它们只需很少的代码即可完成此操作。

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，我们建议在移动设备中使用该`PutRecords`操作或Kinesis Recorder。 AWS SDKs有关更多信息，请参阅《 适用于 Android 的 AWS Mobile SDK 入门指南》和《 AWS 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，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 AWS KMS 和 Using IAM Policies with AWS 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 stamp/deadline 组合设置缓冲区优先级。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 time-to-live价值

此策略允许将重试的 KPL 用户记录包含在后续的 Kinesis Data Streams API 调用中，以提高吞吐量并降低复杂性，同时强制执行 Kinesis Data Streams 记录的值。 time-to-live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。无论您的消费者使用的是 KCL 还是 AWS SDK，这都适用。要使用此聚合功能，如果您的使用者是使用 AWS SDK 中提供的 API 编写的，则需要将 KPL 的 Java 部分提取到您的版本中。

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 AWS SDK为 Kinesis Data Streams 构建消费端。您还可以使用其他 AWS 服务来培养消费者 AWS 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 AWS 与其他服务的集成，例如亚马逊 EMR、亚马逊 AWS Glue和 EventBridge亚马逊 Redshift。它还支持第三方集成，包括 Apache Flink、Adobe Experience Platform、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
- 通过以下方式培养消费者 适用于 Java 的 AWS SDK
- 使用开发消费者 AWS 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 AWS 其他服务从 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

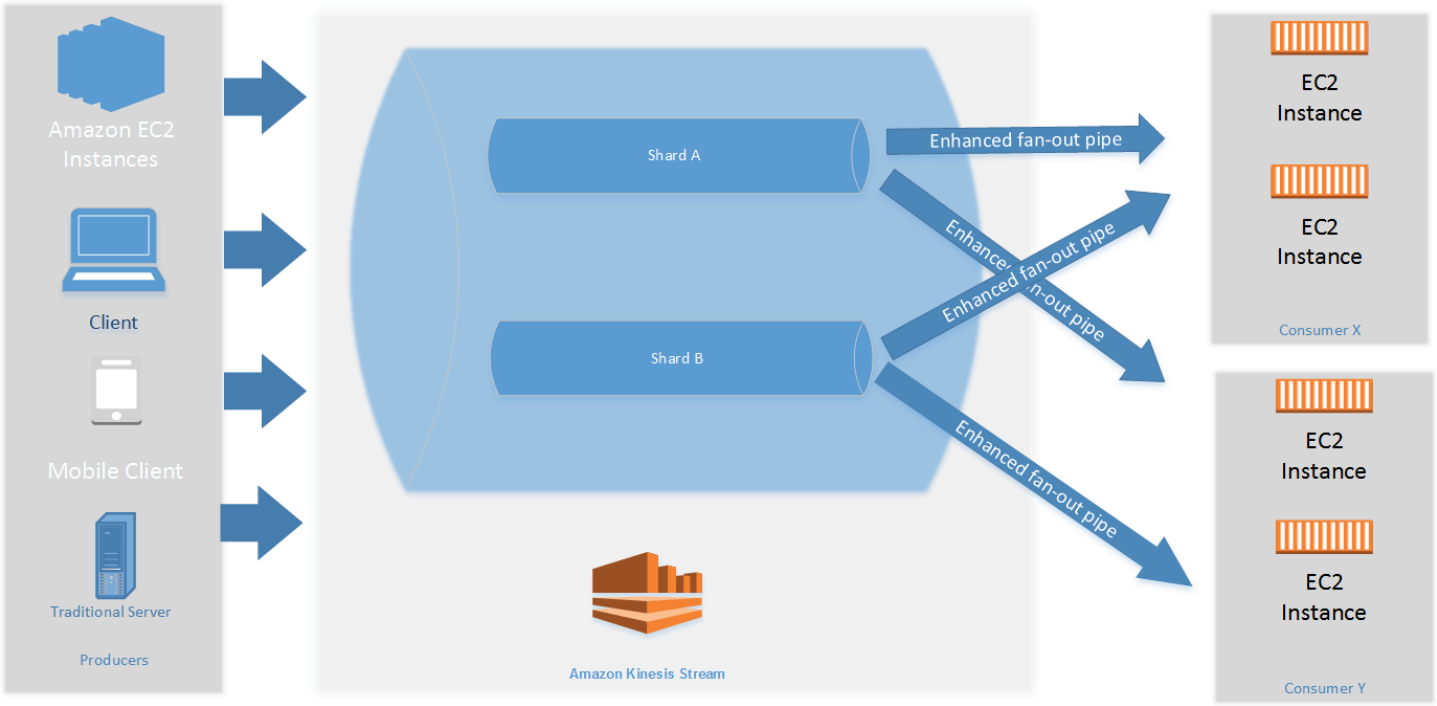
开发具有专用吞吐量的增强扇出型消费端

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 On-demand 优势模式下，您每次直播最多可以注册 50 名消费者，以使用增强版粉丝分享。使用 On-demand 标准和预置直播，您最多可以为每个直播注册 20 个消费者，以使用增强型扇出功能。

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。增强型扇出管道可为每个分片提供多达 2 MB/sec 个数据，与任何其他管道或消费者总数无关。

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 On-demand Advantage）
- 使用增强型粉丝群管理增强型消费者 AWS CLI 或 API

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。消息传播延迟被定义为使用有效载荷调度 API（如 PutRecord 和 PutRecords）发送的有效载荷通过有效载荷消耗 API（如 GetRecords 和 SubscribeToShard）到达消费端应用程序所花费的时间（以毫秒为单位）。

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	固定为每个分片总共 MB/sec 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。他们从分片获得的吞吐量总和不超过 2。 MB/sec	随着消费端注册进行扩展以使用增强型扇出功能。每个注册使用增强型扇出功能的使用者都会获得每个分片自己的读取吞吐量，最多 2 个 MB/sec，独立于其他使用者。
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	使用在 HTTP 上提取模型 GetRecords。	Kinesis 数据流通过使用向您推 HTTP/2 送记录。 Subscribe ToShard

支持最多 50 个增强型扇出消费者的区域（仅限 On-demand Advantage）

在 On-demand 优势模式下最多支持 50 个增强型扇出消费者，仅在以下 AWS 地区可用：

AWS 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）
mx-central-1	墨西哥（中部）
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-east-2	亚太地区（台北）
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）

AWS 区域	区域名称
eu-west-2	欧洲地区（伦敦）
ap-southeast-4	亚太地区（墨尔本）
ap-southeast-5	亚太地区（马来西亚）
ap-southeast-6	亚太地区（新西兰）
ap-southeast-7	亚太地区（泰国）
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（西班牙）
eu-south-1	欧洲地区（米兰）
cn-northwest-1	中国（宁夏）
eu-central-2	欧洲（苏黎世）
us-gov-east-1	AWS GovCloud (US-East)
us-gov-west-1	AWS GovCloud (US-West)

使用增强型粉丝群管理增强型消费者 AWS CLI 或 API

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 AWS CLI 或 Kinesis Data Streams API 来注册、描述、列出和注销在 Kinesis 数据流中使用增强扇出功能的消费者。

使用管理消费者 AWS CLI

您可以使用注册、描述、列出和注销增强型扇出消费者。 AWS 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 API 管理消费端

可使用 Kinesis Data Streams API 注册、描述、列出和取消注册增强扇出型消费端。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 AWS 管理控制台 并打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
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 AWS 管理控制台 并打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
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 Java-based 守护程序提供对 Java 以外语言的 Support MultiLangDaemon。 MultiLangDaemon通过 STDIN 和 STDOUT 与 KCL 应用程序交互。有

关 MultiLangDaemon on 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 AWS 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 AWS 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 还至少进行一次传递。
- 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。
- Multi-language 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 AWS 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处理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 3.x 默认会创建三个 DynamoDB 表来管理元数据：租赁表、工作人员指标表和协调器状态表。从 KCL 3.5 开始，您可以使用单表格式将所有元数据合并到租赁表中。有关更多信息，请参阅 KCL 的单表格式。

> **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。从 KCL 3.5 开始，您可以使用单表格式将这些表合并到租赁表中。

> **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Multi-stream 使用 KCL 进行处理。
- checkpoint：分片的最新检查点序号。
- 检查点SubSequenceNumber：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。
- 业主SwitchesSinceCheckpoint：自上次检查点以来，这份租约更换了多少次员工。
- 父级ShardId：该分片的父级的 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产生的子分片 ID 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。
- end HashKey ing：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 Multi-stream 使用 KCL 进行处理。

- 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：

  - 根据分析估算读取容量单位（RCU）和写入容量单位（WCU）。
  - 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 AWS SDK 2.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（Amazon EC2）

- 操作系统必须是 Linux 操作系统。
- 必须在 EC2 实例中启用 IMDSv2。

Amazon EC2 上的 Amazon Elastic Container Service（Amazon ECS）

- 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 AWS 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）。

Amazon EC2 上的 Amazon Elastic Kubernetes Service（Amazon EKS）

- 操作系统必须是 Linux 操作系统。

亚马逊 EKS 开启 AWS 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（进程）ProcessRecordsInputRecordsInput

- 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租赁）LostInput

- 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。

ShardEndended（ShardEndedInput碎片）EndedInput

- 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（关机）ShutdownRequestedInputRequestedInput

- 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 shard RecordProcessor () 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 AWS 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 会创建具有专用吞吐量的增强型 Fan-out (EFO) 使用器。有关增强版的更多信息 Fan-out，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

本节介绍使用者在 Python、、 Node.js .NET 和 Ruby 中使用 Kinesis 客户端库 (KCL) 的实现。

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。当您使用非 J Java-based ava 语言的 KCL 时，此守护程序将在后台运行。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 on 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: amaz on-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 2.3.0 Node.js
> - 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。

Multi-stream 使用 KCL 进行处理

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**
>
> - Multi-stream 只有 KCL 2.3 或更高版本支持处理。
> - Multi-stream 使用运行的非 Java 语言编写的 KCL 使用者不支持处理。 `multilangdaemon`
> - Multi-stream 任何版本的 KCL 1.x 都不支持处理。

- 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 `streamArm` 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（前StreamsLeasesDeletionStrategy）中旧直播租约的策略。请注意，在消费端应用程序运行时无法更改策略。有关更多信息，请参阅 https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java。
-

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`。

使用 AWS Glue 使用 KCL 的架构注册表

你可以将 Kinesis Data Streams 与架构注册表集成 AWS Glue 。 AWS 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 AWS Glue Schema 注册表，您可以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 AWS 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**
>
> - AWS Glue 只有 KCL 2.3 或更高版本支持 Kinesis Data Streams 的架构注册表集成。
> - AWS 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`
> - AWS 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 AWS 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。 AWS 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 AWS 要求使用细粒度的权限来控制对不同资源的访问权限。 AWS Identity and Access Management (IAM) 允许您在中管理用户和用户权限 AWS。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（ARN）	用途

Amazon Kinesis Data Streams	`DescribeStream`
	
	`DescribeStreamSummary`
		KCL 应用程序从中处理数据的 Kinesis 数据流。
		
		`arn:aws:kinesis:re`
			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。

服务	操作	资源（ARN）	用途
	RegisterStreamConsumer	gion:account:stream/StreamName	将消费端注册到分片。
Amazon Kinesis Data Streams	GetRecords GetShardIterator ListShards	KCL 应用程序从中处理数据的 Kinesis 数据流。 arn:aws:kinesis:region:account:stream/StreamName	从分片读取记录。
Amazon Kinesis Data Streams	SubscribeToShard DescribeStreamConsumer	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。 arn:aws:kinesis:region:account:stream/StreamName/consumer/*	为增强型扇出（EFO）消费端订阅分片。

服务	操作	资源（ARN）	用途

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:dynamodb:region:account:table/KCLApplicationName`
			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（ARN）	用途

Amazon DynamoDB	`CreateTable`
	`DescribeTable`
	`Scan`
	`GetItem`
	`PutItem`
	`UpdateItem`
	`DeleteItem`
	`ConditionCheckItem`
		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-CoordinatorState`
			KCL 需要这些操作才能在 DynamoDB 中管理工作器指标和协调器状态元数据表。该 CoordinatorState 表需要`ConditionCheckItem` 操作来支持单表迁移期间使用的有条件写入。

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（ARN）	用途
亚马逊 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 ARN 中的 “区域”、“账户” 和 “” 分别替换为您自己的 AWS 账户 编号 AWS 区域、Kinesis 数据流名称和 KCL KCLApplicationName 应用程序名称。StreamNameKCL 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。

> **Note**
>
> KCL 3.5 单表迁移完成后，必须手动删除旧版工作器指标和协调器状态表。

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 AWS 区域（例如 us-east-1）。
- 将账号_ID替换为你的 AWS 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
`applicationName`	ConfigsBuilder	此 KCL 应用程序的名称。用作 `tableName` 和 `consumerName` 的默认名称。	不适用
`tableName`	ConfigsBuilder	允许覆盖用于 Amazon DynamoDB 租赁表的表名称。	不适用
`streamName`	ConfigsBuilder	此应用程序从其中处理记录的流的名称。	不适用
`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。	不适用
`failoverTimeMillis`	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。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
shardSync IntervalMillis	LeaseMana gementConfig	分片同步调用之间的时间。	60000（60 秒）
cleanupLe asesUponS hardCompletion	LeaseMana gementConfig	如果设置，只要子租赁已开始处理，即可删除租赁。	TRUE
ignoreUne xpectedCh ildShards	LeaseMana gementConfig	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
maxLeases ForWorker	LeaseMana gementConfig	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
maxLeaseR enewalThreads	LeaseMana gementConfig	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_地平线
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
useAbsoluteReBalanceThreshold	LeaseManagementConfig	确定当负载平衡算法使用 CPU 指标来重新平衡租约时，reBalanceThresholdPercentage 是作为固定百分点 (truefalse) 还是按占机群平均值的相对百分比 () 进行应用。 这是 KCL 3.x 中引入的新配置。	FALSE
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

配置属性	配置类	说明	默认 值
`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
`disableWorkerMetrics`	LeaseManagementConfig	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。 这是 KCL 3.x 中引入的新配置。	FALSE
`maxThroughputPerHostKBps`	LeaseManagementConfig	在租约分配期间分配给工作程序的最大吞吐量。 这是 KCL 3.x 中引入的新配置。	无限制

配置属性	配置类	说明	默认 值
isGracefulLeaseHandoffEnabled	LeaseManagementConfig	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。 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。 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。 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
gracefulLeaseHandoffTimeoutMillis	LeaseManagementConfig	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）。 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。 这是 KCL 3.x 中引入的新配置。	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
taskBackoffTimeMillis	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.DETAILED

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。 AWS 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。我们建议您及时了解 KCL 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 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。 AWS 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 AWS 限制为仅解决关键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。
- 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	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	--	--

KCL 的单表格式

从 KCL 3.5 开始，您可以使用单表格式，将所有 DynamoDB 元数据整合到单个租约表中。默认情况下，KCL 3.x 会为每个应用程序创建三个 DynamoDB 表：租约表、工作线程指标表和协调器状态表。单表格式将这三个表缩减为一个，有助于避免 DynamoDB 账户级别的表限制。

单表格式的工作方式

在单表格式中，KCL 将工作线程指标条目和协调器状态条目与租约条目一起存储在租约表中。每个项目都包含一个 `entityType` 属性，用于区分不同的记录类型。

工作线程指标和协调器状态项目使用与租约表相同的主键结构，但包含不同的 `entityType` 值。通过此属性，KCL 可在表扫描期间识别每个项目的用途。

KCL 的每个组件都会根据 entityType 属性，筛选出其业务逻辑所需的条目。例如，租约分配管理器（LAM）筛选租约和工作线程指标条目以执行租约分配。

配置单表格式

启用单表格式的方式取决于您当前的 KCL 版本：

- 使用 KCL 2.x 时：请按照更新的迁移指南升级到 KCL 3.5。默认情况下，从 2.x 到 3.5 的新迁移使用单表格式。
- 使用 KCL 3.0-3.4 时：您必须执行两阶段部署来迁移到单表格式。请参阅以下配置和迁移步骤。

对于现有的 KCL 3.x 客户，请在 CoordinatorConfig 中设置 migrateAllEntitiesToLeaseTable 配置选项。此选项控制 KCL 是否在租约表中存储所有元数据实体类型。

##的配置值 AllEntitiesToLeaseTable

值	默认	效果
false	是	KCL 为工作线程指标和协调器状态使用单独的表。应用程序代码支持单表格式，但未将其激活。
true	否	KCL 开始将工作线程指标和协调器状态数据写入租约表。在 TableMigrationStatus 进入 DEPLOYED 状态且您已进行了充分的烘焙测试、没有回归问题后，在第 2 阶段部署中设置此值。

从 KCL 3.x 迁移到单表格式需要分两个阶段进行部署：

1. 第 1 阶段：部署更新后的 KCL 3.5 代码，将 migrateAllEntitiesToLeaseTable 设置为 false（默认）。这将安装支持单表格式的新代码，但不会激活迁移。

2. 第 2 阶段：在所有工作线程都运行新代码、TableMigrationStatus 进入 DEPLOYED 状态并且您已验证没有回归问题之后，重新部署并将 migrateAllEntitiesToLeaseTable 设置为 true 以开始迁移。

迁移状态

TableMigrationStateMachine 管理从多表格式向单表格式的过渡。KCL 使用协调器状态表中名为 TableMigration3.5 的单独的协调器状态条目，来跟踪当前的迁移状态。有关状态、过渡和描述的完整列表，请参阅 KCL 单表迁移状态。

单表格式迁移状态

州	说明	过渡条件
INIT	初始状态。所有工作线程都在工作线程指标统计数据中发送最低支持代码。工作线程继续将工作线程指标发送到旧表中，并从旧表和租约表中读取工作线程指标和协调器状态。从功能上讲，INIT 与 DEPLOYED 没有区别。	在烘焙时间内，所有工作线程都会稳定地发送最低支持代码。应用程序已准备好进入第 2 阶段部署。
DEPLOYED	所有工作线程都已使用新代码进行部署（第 1 阶段完成）。应用程序支持单表格式，但未将其激活。	第 2 阶段部署在开始时将 migrateAllEntities ToLeaseTable 设置为 true。
PENDING	所有工作线程都运行新代码。KCL 将工作线程指标表和协调器状态表中的数据迁移到租约表中。	迁移完成后，需要经过 24 小时的默认烘焙时间。
COMPLETE	迁移已完成。KCL 对所有读取和写入仅使用租约表。不再使用旧的工作线程指标表和协调器状态表。	最终状态。不会再继续过渡。

有关每种状态、过渡条件和完整状态机行为的详细信息，请参阅 KCL 单表迁移状态机。

> **Note**
>
> PENDING 和 COMPLETE 状态之间的默认烘焙时间为 24 小时，不过最多可以配置为一周。在此期间，KCL 为所有读取和写入使用租约表，但不删除旧表。确认迁移成功后，您必须手动删除旧的工作线程指标表和协调器状态表。KCL 不会自动删除这些表。

迁移指标

使用 KCL，您可以使用 CloudWatch 指标监控单表迁移的进度和运行状况。使用这些指标来确认工作线程已经采用了新代码，并跟踪迁移在各个状态中的进展情况。在迁移过程中，您还可以检测 DynamoDB 的读取、写入或删除故障。下表按发送指标的 KCL 操作（指标维度）和每个指标的发送时间，对指标进行分组。

无论迁移是否正在进行，当选的领导工作线程都会持续发送以下指标：

领导工作线程始终发送的指标

操作	指标	单位	说明
`TableMigration`	`StatusOrdinal`	无	当前 DynamoDB 迁移状态的序数：`0` = UNKNOWN，1 = INIT，2 = DEPLOYED，3 = PENDING，4 = COMPLETE。
`WorkerMetrics`	`FleetMinSupportCode`	无	队列中所有拥有租约的工作线程的最低支持代码。

只有当迁移正在进行时，当选的领导工作线程才会发送以下指标：

领导工作线程在迁移期间发布的指标

操作	指标	单位	说明
`TableMigration`	`Phase1Worker`	计数	支持使用单个 DynamoDB 表进行操作、但尚未迁移为使用单表的工作线程数量。

操作	指标	单位	说明
TableMigration	Phase2Worker	计数	已迁移到写入单表的工作线程数量。在表迁移完成之前，这些工作线程可能仍会从多个表中读取数据。
TableMigration	PrePhase1Worker	计数	3.5 之前版本的工作线程数量，这些工作线程不支持在单个 DynamoDB 表上进行操作。
TableMigration	WriteFault	计数	DynamoDB 写入失败时为 1，成功时为 0。
TableMigration	DeleteFault	计数	DynamoDB 删除失败时为 1，成功时为 0。
TableMigration	CompletionFault	计数	事务写入 COMPLETE 状态失败时为 1，成功时为 0。
TableMigrationAsyncMove	Success	计数	1在交易成功移动 CoordinatorState 条目时，0在失败时。
TableMigrationAsyncMove	Time	毫秒	异步移动操作的持续时间。
TableMigrationAsyncMove	BatchCount	计数	成功移动的批次数量。

在迁移过程中，所有工作线程都会发送以下指标：

迁移期间所有工作线程发送的指标

操作	指标	单位	说明
TableMigration	ReadFault	计数	未能从 DynamoDB 读取 TableMigrationState 时为 1，读取成功时为 0。
TableMigration	Time	毫秒	状态机运行的持续时间。
TableMigrationInitialize	Success	计数	成功初始化时为 1，失败时为 0。
TableMigrationInitialize	Time	毫秒	初始化操作的持续时间。

使用这些指标来决定何时继续进行迁移。当 StatusOrdinal 持续为 2（DEPLOYED）且 PrePhase1Worker 为 0 时，所有工作线程都支持单表格式，您可以进入表迁移部署的第 2 阶段。在 StatusOrdinal 成为 4（COMPLETE）后，您可以安全地删除旧表。

回滚注意事项

回滚支持取决于当前 TableMigrationStatus：

- 在第 1 阶段（TableMigrationStatus 为 DEPLOYED 或尚未设置）：您可以安全地回滚到以前的版本。新代码在向后兼容模式下运行，并且未向租约表中的非租约条目写入任何数据。
- 在第 2 阶段（TableMigrationStatus 为 DEPLOYED 或 PENDING）：您可以回滚到第 1 阶段。对于工作线程指标和协调器状态，工作线程恢复为使用旧表（多表格式）。迁移已撤消。
- 进入 COMPLETE 状态之后：不支持回滚。KCL 对所有实体都仅使用租约表。即使代码回滚到第 1 阶段，工作线程仍会为所有实体继续使用租约表，并且忽略配置。

> **Warning**
>
> 迁移过程进入 COMPLETE 状态后，应用程序仅在单表模式下运行。migrateAllEntitiesToLeaseTable 配置将被忽略，KCL 不会恢复为使用单表。请确

保在进入 COMPLETE 状态之前有足够长的烘焙时间，因为进入该状态后，即使进行代码回滚也不会切换回多表。

最佳实践

采用单表格式时，请遵循以下最佳实践：

- 第 1 阶段部署后，验证协调器状态条目 `TableMigration3.5` 是否成为 `DEPLOYED` 状态。在进入第 2 阶段之前，请确保没有回归问题。
- 监控 `TableMigration3.5` 协调器状态条目中的 `TableMigrationStatus`，跟踪在 DEPLOYED、PENDING 和 COMPLETE 状态中的进度。在迁移过程变为 COMPLETE 之前，状态作为单独的条目存储在协调器状态表中（而不是租约表中）。
- 请确保在进入 COMPLETE 状态之前有足够长的烘焙时间，因为进入该状态后，即使进行代码回滚也不会切换回多表。应用程序仅在单表模式下运行。
- 迁移过程变为 COMPLETE 之后，请手动删除旧的工作线程指标表和协调器状态表。KCL 不自动删除这些表，只是停止使用。
- 如果您已配置 `CoordinatorConfig.coordinatorStateTableConfig` 或 `LeaseManagementConfig.workerUtilizationAwareAssignmentConfig.workerMetricsTable` 则可以在迁移完成后移除这些配置。在 KCL 3.5 及更高版本中，这些配置已弃用。

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 AWS 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 AWS 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
- 从 KCL 3.5.x 回滚后向前滚动到 KCL 3.5.x
- 使用预置容量模式的租约表的最佳实践
- 从 KCL 3.x 迁移到单表格式
- 从 KCL 1.x 迁移到 KCL 3.x

从 KCL 2.x 迁移至 KCL 3.x

本主题提供了将消费端从 KCL 2.x 迁移至 KCL 3.x 的分步说明。我们建议迁移到 KCL 3.5 或更高版本以使用单表格式。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。

> **Important**
>
> 对于从 KCL 2.x 到 KCL 3.5 或更高版本的新迁移，默认使用单表格式。您的应用程序仅使用所有元数据的租赁表，因此无需单独使用工作器指标和协调器状态表。有关更多信息，请参阅 KCL 的单表格式。

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 AWS 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 AWS 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 AWS 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 迁移到 3.x 时，迁移是分阶段执行的。首先，将此配置设置为`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`并部署到所有工作人员。在创建调度器对象时添加以下行：

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

在此阶段，您的应用程序仍与 KCL 2.x 兼容，并且迁移不会开始，这使您可以安全地在队列中部署 KCL 3.x 库。

在所有工作人员都运行之后CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1，将此配置设置为CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X以开始迁移。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_CONFI
```

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.coordinatorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONF
    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。

> **Note**
>
> 有关 KCL 3.5 与迁移相关的最新配置更改，请参阅 KCL 3.5 配置更新。

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。消费端应用程序的工作程序可运行在 Amazon EC2、Amazon EKS、Amazon EKS 或 AWS Fargate上。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（Amazon EC2）

- 操作系统必须是 Linux 操作系统。
- 必须在 EC2 实例中启用 IMDSv2。

Amazon EC2 上的 Amazon Elastic Container Service（Amazon ECS）

- 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 AWS 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）。

Amazon EC2 上的 Amazon Elastic Kubernetes Service（Amazon EKS）

- 操作系统必须是 Linux 操作系统。

亚马逊 EKS 开启 AWS 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：
>
> - 行动：`UpdateTable`
>   - 资源（ARN）：`arn:aws:dynamodb:region:account:table/KCLApplicationName`
> - 行动：`Query`
>   - 资源（ARN）：`arn:aws:dynamodb:region:account:table/KCLApplicationName/index/*`
> - 操作：`CreateTable`、`DescribeTable`、`Scan`、`GetItem`、`PutItem`、`UpdateItem`、`DeleteIte`
>   - 资源（ARN）：`arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats`，`arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorState`

将 ARN 中的 “区域”、“账户” 和 “KCLApplicationName” 分别替换为您自己的名称 AWS 区域、 AWS 账户 编号和 KCL 应用程序名称。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。所有指标均为每个KCL-application 指标，并设置为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
`CurrentState:3xWorker`	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 `Sum` 计数与

指标	说明
	工作程序总数一致，则表示已成功完成向 KCL 3.x 的迁移。 • 指标级别：汇总 • 单位：计数 • 统计数据：最有用的统计工具是 Sum。
CurrentState:2xCompatibleWorker	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。 • 指标级别：汇总 • 单位：计数 • 统计数据：最有用的统计工具是 Sum。
Fault	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 支持。 • 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	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。 • 指标级别：汇总 • 单位：计数 • 统计数据：最有用的统计工具是 Sum。

指标	说明
workerMetricsReady	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。 • 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 KCL 3.5.x 使用器还原到先前版本的步骤。回滚过程取决于您的应用程序当前所处的迁移阶段。

> **Important**
>
> 从第 2 阶段 () 回滚到第 1 阶段 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) 时，需要使用 KCL 迁移工具。CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1还需要使用该工具才能从第 1 阶段回滚到之前的 KCL 版本 (KCL 2.x)。这适用于您的租赁表中仍包含早期第 2 阶段部署中特定于迁移的条目。如果您的应用程序直接进入第 1 阶段但尚未启动第 2 阶段，则无需运行该工具，即可重新部署之前的代码，从而回滚到之前的 KCL 版本。

从第 1 阶段回滚到之前的 KCL 版本

如果您的应用程序作为从 KCL 2.x 迁移的一部分直接进入第 1 阶段 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1)，但尚未启动第 2 阶段，则可以通过重新部署之前的代码来回滚到之前的 KCL 版本。第 1 阶段向后兼容以前的 KCL 版本，并且不会在租用表中创建任何特定于迁移的条目。无需迁移工具。

要从第 1 阶段回滚，请执行以下操作：

1. 将之前的 KCL 版本的代码重新部署到所有工作人员。

> **Note**
>
> 您无法直接从第 2 阶段回滚到之前的 KCL 版本 (KCL 2.x)。你必须按顺序回滚：
>
> 1. 从第 2 阶段回滚到第 1 阶段（运行该工具，然后重新部署第 1 阶段的代码）。请参阅从第 2 阶段回滚到第 1 阶段。
> 2. 从第 1 阶段回滚到 KCL 2.x（运行该工具，然后重新部署之前的代码）。请参阅从第 1 阶段回滚到 KCL 2.x。
>
> 您可以出于以下两个原因之一重新部署第 1 阶段代码：继续回滚到 KCL 2.x，或者因为单独运行该工具并不能缓解问题，您希望完全返回第 1 阶段。

从第 2 阶段回滚到第 1 阶段

如果您的应用程序处于第 2 阶段 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X)，则必须使用 KCL 迁移工具回滚到第 1 阶段。这是一个包括两个步骤的过程：

1. 运行 KCL Migration Tool。
2. 使用第 1 阶段的配置来重新部署代码（可选）。

> **Note**
>
> KCL 迁移工具仅处理第 2 阶段到第 1 阶段的回滚。要达到第 2 阶段之前的 KCL 版本 (KCL 2.x)，请先使用本节中的步骤回滚从第 1 阶段回滚到 KCL 2.x到第 1 阶段，然后按照rollback-to-v2模式运行该工具并在重新部署之前删除非租赁条目。

步骤 1：运行 KCL 迁移工具

当您需要从第 2 阶段 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) 回滚到第 1 阶段时，请运行 KCL 迁移工具。该工具执行以下任务：

- 它会删除 DynamoDB 租用表上的全局二级索引 (LeaseOwnerToLeaseKeyIndex)。此索引由 KCL 3.5.x 创建，但在回滚到第 1 阶段时不需要此索引。
- 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。如果您在使用 KCL 3.5.x 中的新负载平衡算法时遇到问题，则可以立即缓解问题。

> **Important**
>
> 在迁移、回滚和前滚过程中，不得删除租约表中的协调器状态条目 (Migration3.0)。

> **Note**
>
> 使用者应用程序中的所有工作线程在给定时间均必须使用相同的负载均衡算法。KCL 迁移工具可确保 KCL 3.5.x 消费者应用程序中的所有工作程序都切换到 KCL 2.x 兼容模式，以便在将部署回第 1 阶段的过程中，所有工作程序都运行相同的负载平衡算法。

您可以在 KCL 存储库的脚本目录中下载 KCL GitHub 迁移工具。从任意工作线程或任意具备所需权限（写入和更新租约表的权限）的主机上运行该脚本。您可以参考以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。使用以下命令运行 KCL 迁移工具：

```
python3 ./KclMigrationTool.py --region <region> --mode rollback-to-phase1 [--
application_name <applicationName>] [--lease_table_name <leaseTableName>]
```

参数

- --区域：替换<region>为你的。 AWS 区域
- --application_name：如果您在租赁表中使用默认名称，则必须使用此参数。如果您为租约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。

步骤 2：使用第 1 阶段的配置来重新部署代码（可选）

运行 KCL 迁移工具从第 2 阶段回滚到第 1 阶段后，您将看到以下消息之一：

> **Note**
>
> 如果您回滚以缓解问题并打算继续使用 KCL 3.5.x（稍后无需重新部署即可向前滚动到第 2 阶段），则重新部署第 1 阶段配置是可选的。如果您打算继续回滚到 KCL 2.x，则需要执行此操作，因为该`rollback-to-v2`步骤要求所有工作人员都必须先运行第 1 阶段的配置。

- 消息 1：“回滚已完成。应用程序正在运行第 2 阶段（兼容 2x）功能。请使用第 1 阶段的配置部署 KCL 3.5.x 应用程序，以回滚到第 1 阶段。”
  - 必需的操作：您的工作人员正在第 2 阶段（兼容 2x）模式下运行。使用第 1 阶段配置 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) 将您的 KCL 3.5.x 应用程序重新部署到您的工作人员。
- 消息 2：“回滚已完成。您的 KCL 应用程序在运行第 2 阶段（3x）功能，并已回滚到第 2 阶段（兼容 2x）模式。如果您在短时间内未看到缓解，请使用第 1 阶段配置部署 KCL 3.5.x 应用程序，以回滚到第 1 阶段。”
  - 必需的操作：您的工作人员在第 2 阶段 (3x) 模式下运行，KCL 迁移工具将其恢复到第 2 阶段（兼容 2x）模式。如果问题已解决，则无需重新部署。如果问题仍然存在，请将具有第 1 阶段配置 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) 的 KCL 3.5.x 应用程序重新部署到您的工作人员。

从第 1 阶段回滚到 KCL 2.x

本节介绍当您的应用程序处于第 2 阶段时，如何从第 1 阶段回滚到之前的 KCL 版本 (KCL 2.x)。这是中描述的序列的第二步从第 1 阶段回滚到之前的 KCL 版本：首先从第 2 阶段回滚到第 1 阶段，然后完成本节中的步骤。在所有工作人员都运行第 1 阶段后，在`rollback-to-v2`模式下运行 KCL 迁移工具以清理非租赁条目，然后重新部署之前的代码。

> **Important**
>
> 如果不执行此步骤，则回滚到之前的代码会使应用程序停滞不前，因为之前的代码无法处理租赁表中的非租赁条目。

> **Important**
>
> 在`rollback-to-v2`模式下运行该工具之前，请确认第 2 阶段到第 1 阶段的回滚已完成：队列中的所有工作人员都已完成第 1 阶段的代码回滚部署，`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`并且没有剩下第 2 阶段的工作人员。该工具只有在已经运行时`MigrationState.ClientVersion`才会继续运行`CLIENT_VERSION_2X`，这表示第 2 阶段到第 1 阶段的回滚已经运行。使用您的 CI/CD 系统进行验证。

要从第 1 阶段回滚到之前的 KCL 版本 (KCL 2.x)，请执行以下操作：

1. 在`rollback-to-v2`模式下运行 KCL 迁移工具。您可以从任何工作人员或任何具有写入和更新租赁表所需权限的主机上运行脚本。由于此模式会从租赁表中删除非租赁条目，因此主机必须对租赁表具有删除权限 (`DeleteItem`)。有关所需的 IAM 权限，请参阅KCL 消费端应用程序所必需的 IAM 权限。

   ```
   python3 ./KclMigrationTool.py --region <region> --mode rollback-to-v2 [--
   application_name <applicationName>] [--lease_table_name <leaseTableName>]
   ```

   参数

   - --region：将 `<region>` 替换为您的 AWS 区域。
   - --application_name：如果您在租赁表中使用默认名称，则必须使用此参数。如果您为租约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。
   - （可选）--lease_table_name：在 KCL 配置中为租用表设置了自定义名称后，请使用此参数。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。
2. 该工具扫描租赁表中的非租赁实体，并列出要从租赁表中删除的条目，例如客户端版本迁移条目 (`Migration3.0`) 和工作器指标统计数据条目。查看列出的条目。

3. 当工具提示您时，请准确键入以下确认字符串，以确认第 1 阶段代码回滚已完成：

```
Code rollback to CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1 completed
```

4. 当工具提示您确认删除时，键入yes。该工具将列出的非租赁实体从租赁表中永久删除。
5. 在工具报告 KCL 2.x 回滚的租赁表已清理干净之后，请将之前的 KCL 版本 (KCL 2.x) 的代码重新部署到所有工作程序。

> **Important**
>
> 该工具删除非租赁实体后，如果要向前滚动到第 2 阶段，则需要从第 1 阶段重新迁移。当您准备好再次迁移到 KCL 3.5.x 时，请使用第 1 阶段配置（CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1）进行部署，让第 1 阶段的部署根据您的置信区间进行烘烤，然后使用第 2 阶段的配置（）进行部署（）。CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X这两种部署都不需要任何工具。

从 KCL 3.5.x 回滚后向前滚动到 KCL 3.5.x

本主题说明了在回滚后将使用器回滚到 KCL 3.5.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. 部署第 2 阶段代码。

步骤 1：运行 KCL 迁移工具

使用以下命令运行 KCL 迁移工具，向前滚动到 KCL 3.5.x：

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

参数

- --区域：替换<region>为你的。 AWS 区域
- --application_name：如果您在租赁表中使用默认名称，则必须使用此参数。如果您为租约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。

在前滚模式下运行迁移工具后，KCL 会创建 KCL 3.5.x 所需的以下 DynamoDB 资源：

- 租约表上的全局二级索引

> **Note**
>
> KCL 3.5.x 不会在迁移期间创建单独的工作器指标或协调器状态表。所有元数据都存储在租约表中。

步骤 2：使用第 2 阶段配置部署代码

运行 KCL 迁移工具进行向前滚动后，将使用 KCL 3.5.x 第 2 阶段配置 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`) 的代码部署到您的工作人员。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 3.x 迁移到单表格式

如果您使用所有三个 DynamoDB 元数据表（租赁表、工作器指标表和协调器状态表）运行 KCL 3.x，则可以迁移到单表格式以将所有元数据合并到租赁表中。

先决条件

- 您的应用程序运行包含所有三个元数据表的 KCL 3.x。
- 您的应用程序是稳定的，所有工作人员都很健康。
- 您的应用程序已完成 KCL 3.x 迁移，但尚未进行配置。`CoordinatorConfig.clientVersionConfig`请勿同时执行表迁移和从 2.x 到 3.x 的客户端版本迁移。

要从 KCL 3.x 迁移到单表格式，请遵循两阶段部署流程

1. 第 1 阶段：将项目依赖关系更新到 KCL 3.5 或更高版本。在`migrateAllEntitiesToLeaseTable`设置为`false`（默认）的情况下部署。此部署会安装支持单表格式但不会激活迁移的新代码。
2. 等待所有工作人员的部署完成。监控`TableMigration3.5`协调器状态条目，直到到`TableMigrationStatus`达`DEPLOYED`。确认没有回归并留出足够的烘焙时间。
3. 第 2 阶段：在`migrateAllEntitiesToLeaseTable`设置为的情况下再次部署`true`。此部署开始向单表格式的迁移。
4. 监控`TableMigrationStatus`. 领导者工作人员通过 “已部署”、“待定” 和 “完成” 来推进状态。
5. 状态变为 COMPLETE 后，确认没有回归，并留出额外的烘焙时间以确保无需回滚。
6. 手动删除旧的工作器指标和协调器状态表。KCL 不自动删除这些表，只是停止使用。如果您已配置`CoordinatorConfig.coordinatorStateTableConfig`或`LeaseManagementConfig.workerUt`则可以删除这些配置，因为它们已被弃用。

> **Important**
>
> 迁移到达 “完成” 状态后，请勿回滚应用程序。完成后，应用程序仅在单表模式下运行。即使代码回滚到第 1 阶段也不会切换回多个表，配置会被忽略，应用程序继续仅使用租用表。

有关迁移状态和回滚选项的详细信息，请参阅KCL 的单表格式。

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。必须将记录处理器、记录处理器工厂和工作器类迁移到 KCL 2。 x/3首先是兼容.x 的格式，然后按照 KCL 2.x 到 KCL 3.x 迁移的迁移步骤进行操作。可直接从 KCL 1.x 升级至 KCL 3.x。

> **Important**
>
> 我们建议您迁移到 KCL 3.5 或更高版本，新迁移默认使用单表格式。首先，将您的记录处理器、工厂和工作人员类迁移到 KCL 2。 x/3使用本页上的步骤使用与.x 兼容的格式。然后，按照从 KCL 2.x 到 KCL 3.5 的迁移路径行驶。从 2.x 开始的新迁移会自动启用单表格式。有关更多信息，请参阅 KCL 的单表格式。

- 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：遵循 KCL 2.x 向 KCL 3.x 迁移指南中的分步说明进行操作

  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 版本和从 KCL 3.5.x 回滚后向前滚动到 KCL 3.5.x。

> **Important**
>
> 不要在 KCL 3.x 中使用 2.27.19 到 2.27.23 适用于 Java 的 AWS 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 AWS 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 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。

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 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 AWS Glue 架构注册表

> **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。

请注意，KCL 与 AWS 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。当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。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，这是因为。 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 AWS 区域。有关 MultiLangDaemon on 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 AWS 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，请参阅。https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/coordinator/Scheduler.java

- 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（视最大LeasesForWorker变量的值而定）。

  > **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），https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.java请参阅 KCL 1.x 和 https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/Lease.javaKCL 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。
- 检查点SubSequenceNumber：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。
- 业主SwitchesSinceCheckpoint：自上次写检查点以来，这份租约已经更换了多少次员工。
- par@@ entShardId：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 中预配置吞吐量的信息，Read/Write 请参阅 Amazon DynamoDB 开发者指南中的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，在消费端应用程序的引导启动期间以及每个数据流重新分片事件期间，KCL 会将租约表与通过调用 `ListShards` 或 `DescribeStream` 发现 API 从 Kinesis Data Streams 服务获取的分片信息同步。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.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html。

- 不是所有工作人员都执行 lease/shard 同步以使租用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同 lease/shard步。
- KCL 2.3 使用GetRecords和 SubscribeToShard API 的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。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。

  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
租约 RecoveryAuditorExecutionFrequencyMillis	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

Name	默认 值	说明
		据 leasesRecoveryAuditorInconsistencyConfidenceThreshold 触发分片同步。
租约 RecoveryAuditorInconsistencyConfidenceThreshold	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.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html。

- 不是所有工作人员都执行 lease/shard 同步以使租用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同 lease/shard步。
- KCL 1.14 使用GetRecords和 SubscribeToShard API 的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。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，leasesRecoveryAuditorInconsistencyConfidenceThr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
租约 RecoveryAuditorInconsistencyConfidenceThreshold	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
租赁 CleanupIntervalMillis	1 minute	运行租约清理线程的时间间隔。
已完成 LeaseCleanupIntervalMillis	5 分钟	检查租约是否完成的时间间隔。
垃圾 LeaseCleanupIntervalMillis	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 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 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 streamArm 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。有关更多信息，请参阅 https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java。

- 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 AWS Glue 架构注册表

您可以将 Kinesis 数据流与 AWS Glue 架构注册表集成。 AWS 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。 AWS Glue Schema Registry 允许您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 AWS 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 AWS Glue 和 Schema Registry 集成。 Multi-language 不提供支持。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 Glu KPL/KCL e 架构注册表集成” 中的 “使用库与 AWS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 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 AWS 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 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 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
- 在中开发 Kinesis 客户端库使用者 Node.js
- 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 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 AWS Javadoc 主题。 AmazonKinesisClient

要从中下载 Java KCL GitHub，请前往 K inesis 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 AWS

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 AWS 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 IRecordProcessor 方法
- 为 IRecordProcessor 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 IRecordProcessor 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。

IRecordProcessor 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 IRecordProcessor 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程序的方式。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 AWS 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 实例上运行消费端，则我们建议您使用 IAM 角色启动实例。反映与此 IAM 角色关联的权限的 AWS 凭证可通过实例元数据提供给实例上的应用程序。使用这种方式管理在 EC2 实例上运行的消费端的凭证最安全。

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();
   ```

在中开发 Kinesis 客户端库使用者 Node.js

> **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 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当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。因此，如果您为安装 KCL Node.js 并完全在其中编写消费者应用程序 Node.js，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 AWS 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 Node.js KCL GitHub，请前往 K inesis 客户端库 Node.js ()。

示例代码下载

KCL 有两个代码样本可用： Node.js

- basic-sample

  以下各节用于说明在中构建 KCL 消费者应用程序的基础知识。 Node.js
- 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。

在中实现 KCL 使用者应用程序时，必须完成以下任务： Node.js

任务

- 实现记录处理器

- 修改配置属性

实现记录处理器

使用 KCL for 的最简单使用者 Node.js 必须实现一个recordProcessor函数，该函数反过来又包含函数initializeprocessRecords、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-encoded。

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 AWS 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。 AWS 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。使用这种方式管理在 EC2 实例上运行的消费端应用程序的凭证最安全。

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 default AWS credentials 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 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 .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当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。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 AWS 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.NET KCL GitHub，请访问 K inesis 客户端库 (.NET)。要下载.NET KCL 使用者应用程序的示例代码，请转到上的 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 在实例化记录处理程序时调用此方法，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。

```
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 AWS 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 实例上运行消费端应用程序，则建议您使用 IAM 角色进行配置。反映与此 IAM 角色关联的权限 AWS 凭证可通过实例元数据提供给实例上的应用程序。使用这种方式管理在 EC2 实例上运行的消费端的凭证最安全。

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 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当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。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 AWS 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 K inesis 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

RecordProcess 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-encoded。

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 AWS 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 实例上运行消费端应用程序，则建议您使用 IAM 角色进行配置。反映与此 IAM 角色关联的权限 AWS 凭证可通过实例元数据提供给实例上的应用程序。使用这种方式管理在 EC2 实例上运行的消费端应用程序的凭证最安全。

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 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 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当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。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 AWS 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 K inesis 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 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 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 中的实施示例。如果您想利用增强的扇出功能，请参阅使用增强版的使用者。 Fan-Out

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */


/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
```

```
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.retrieval.polling.PollingConfig;

/**
 * This class will run a simple app that uses the KCL to read data and uses the AWS SDK
 to publish data.
 * Before running this program you must first create a Kinesis stream through the AWS
 console or AWS SDK.
 */
public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    /**
     * Invoke the main method with 2 args: the stream name and (optionally) the region.
```

```
     * Verifies valid inputs and then starts running the app.
     */
    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
The Region may be specified as the second argument.");
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    /**
     * Constructor sets streamName and region. It also creates a KinesisClient object
to send data to Kinesis.
     * This KinesisClient is used to send dummy data so that the consumer has something
to read; it is also used
     * indirectly by the KCL to handle the consumption of the data.
     */
    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {

        /**
         * Sends dummy data to Kinesis. Not relevant to consuming the data with the KCL
         */
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
```

```
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        /**
         * Sets up configuration for the KCL, including DynamoDB and CloudWatch
dependencies. The final argument, a
         * ShardRecordProcessorFactory, is where the logic for record processing lives,
and is located in a private
         * class below.
         */
        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        /**
         * The Scheduler (also called Worker in earlier versions of the KCL) is the
entry point to the KCL. This
         * instance is configured with defaults provided by the ConfigsBuilder.
         */
        Scheduler scheduler = new Scheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                configsBuilder.retrievalConfig().retrievalSpecificConfig(new
PollingConfig(streamName, kinesisClient))
        );

        /**
         * Kickoff the Scheduler. Record processing of the stream of dummy data will
continue indefinitely
         * until an exit is triggered.
         */
        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        /**
```

```
         * Allows termination of app by pressing Enter.
         */
        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        /**
         * Stops sending dummy data.
         */
        log.info("Cancelling producer and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();

        /**
         * Stops consuming data. Finishes processing the current batch of data already
received from Kinesis
         * before shutting down.
         */
        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown.  Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    /**
     * Sends a single record of dummy data to Kinesis.
     */
    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
```

```
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }

    /**
     * The implementation of the ShardRecordProcessor interface is where the heart of
the record processing logic lives.
     * In this example all we do to 'process' is log info about the records.
     */
    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        /**
         * Invoked by the KCL before data records are delivered to the
ShardRecordProcessor instance (via
         * processRecords). In this example we do nothing except some logging.
         *
         * @param initializationInput Provides information related to initialization.
         */
        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
```

```
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Handles record processing logic. The Amazon Kinesis Client Library will
invoke this method to deliver
         * data records to the application. In this example we simply log our records.
         *
         * @param processRecordsInput Provides the records to be processed as well as
information and capabilities
         *                            related to them (e.g. checkpointing).
         */
        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)",
processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /** Called when the lease tied to this record processor has been lost. Once the
lease has been lost,
         * the record processor can no longer checkpoint.
         *
         * @param leaseLostInput Provides access to functions and data related to the
loss of the lease.
         */
        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
```

```
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Called when all data on this shard has been processed. Checkpointing must
occur in the method for record
         * processing to be considered complete; an exception will be thrown otherwise.
         *
         * @param shardEndedInput Provides access to a checkpointer method for
completing processing of the shard.
         */
        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Invoked when Scheduler has been requested to shut down (i.e. we decide to
stop running the app by pressing
         * Enter). Checkpoints and logs the data a final time.
         *
         * @param shutdownRequestedInput Provides access to a checkpointer, allowing a
record processor to checkpoint
         *                               before the shutdown is completed.
         */
        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
```

```
        }
    }

}
```

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 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当您使用 Java 以外的 Java-based KCL 语言时，此守护程序将在后台运行。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 AWS 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 K inesis 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

`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
        a scaling change.
    """
    def __init__(self):
        self._SLEEP_SECONDS = 5
```

```
        self._CHECKPOINT_RETRIES = 5
        self._CHECKPOINT_FREQ_SECONDS = 60
        self._largest_seq = (None, None)
        self._largest_sub_seq = None
        self._last_checkpoint_time = None

    def log(self, message):
        sys.stderr.write(message)

    def initialize(self, initialize_input):
        """
        Called once by a KCLProcess before any calls to process_records

        :param amazon_kclpy.messages.InitializeInput initialize_input: Information
about the lease that this record
            processor has been assigned.
        """
        self._largest_seq = (None, None)
        self._last_checkpoint_time = time.time()

    def checkpoint(self, checkpointer, sequence_number=None, sub_sequence_number=None):
        """
        Checkpoints with retries on retryable exceptions.

        :param amazon_kclpy.kcl.Checkpointer checkpointer: the checkpointer provided to
either process_records
            or shutdown
        :param str or None sequence_number: the sequence number to checkpoint at.
        :param int or None sub_sequence_number: the sub sequence number to checkpoint
at.
        """
        for n in range(0, self._CHECKPOINT_RETRIES):
            try:
                checkpointer.checkpoint(sequence_number, sub_sequence_number)
                return
            except kcl.CheckpointError as e:
                if 'ShutdownException' == e.value:
                    #
                    # A ShutdownException indicates that this record processor should
be shutdown. This is due to
                    # some failover event, e.g. another MultiLangDaemon has taken the
lease for this shard.
                    #
                    print('Encountered shutdown exception, skipping checkpoint')
```

```
                    return
                elif 'ThrottlingException' == e.value:
                    #
                    # A ThrottlingException indicates that one of our dependencies is
is over burdened, e.g. too many
                    # dynamo writes. We will sleep temporarily to let it recover.
                    #
                    if self._CHECKPOINT_RETRIES - 1 == n:
                        sys.stderr.write('Failed to checkpoint after {n} attempts,
giving up.\n'.format(n=n))
                        return
                    else:
                        print('Was throttled while checkpointing, will attempt again in
{s} seconds'
                              .format(s=self._SLEEP_SECONDS))
                elif 'InvalidStateException' == e.value:
                    sys.stderr.write('MultiLangDaemon reported an invalid state while
checkpointing.\n')
                else:  # Some other error
                    sys.stderr.write('Encountered an error while checkpointing, error
was {e}.\n'.format(e=e))
            time.sleep(self._SLEEP_SECONDS)

    def process_record(self, data, partition_key, sequence_number,
sub_sequence_number):
        """
        Called for each record that is passed to process_records.

        :param str data: The blob of data that was contained in the record.
        :param str partition_key: The key associated with this recod.
        :param int sequence_number: The sequence number associated with this record.
        :param int sub_sequence_number: the sub sequence number associated with this
record.
        """
        ####################################
        # Insert your processing logic here
        ####################################
        self.log("Record (Partition Key: {pk}, Sequence Number: {seq}, Subsequence
Number: {sseq}, Data Size: {ds}"
                 .format(pk=partition_key, seq=sequence_number,
sseq=sub_sequence_number, ds=len(data)))

    def should_update_sequence(self, sequence_number, sub_sequence_number):
        """
```

```
        Determines whether a new larger sequence number is available

        :param int sequence_number: the sequence number from the current record
        :param int sub_sequence_number: the sub sequence number from the current record
        :return boolean: true if the largest sequence should be updated, false
otherwise
        """
        return self._largest_seq == (None, None) or sequence_number >
self._largest_seq[0] or \
            (sequence_number == self._largest_seq[0] and sub_sequence_number >
self._largest_seq[1])

    def process_records(self, process_records_input):
        """
        Called by a KCLProcess with a list of records to be processed and a
checkpointer which accepts sequence numbers
        from the records to indicate where in the stream to checkpoint.

        :param amazon_kclpy.messages.ProcessRecordsInput process_records_input: the
records, and metadata about the
            records.
        """
        try:
            for record in process_records_input.records:
                data = record.binary_data
                seq = int(record.sequence_number)
                sub_seq = record.sub_sequence_number
                key = record.partition_key
                self.process_record(data, key, seq, sub_seq)
                if self.should_update_sequence(seq, sub_seq):
                    self._largest_seq = (seq, sub_seq)

            #
            # Checkpoints every self._CHECKPOINT_FREQ_SECONDS seconds
            #
            if time.time() - self._last_checkpoint_time >
self._CHECKPOINT_FREQ_SECONDS:
                self.checkpoint(process_records_input.checkpointer,
str(self._largest_seq[0]), self._largest_seq[1])
                self._last_checkpoint_time = time.time()

        except Exception as e:
            self.log("Encountered an exception while processing records. Exception was
{e}\n".format(e=e))
```

```
    def lease_lost(self, lease_lost_input):
        self.log("Lease has been lost")

    def shard_ended(self, shard_ended_input):
        self.log("Shard has ended checkpointing")
        shard_ended_input.checkpointer.checkpoint()

    def shutdown_requested(self, shutdown_requested_input):
        self.log("Shutdown has been requested, checkpointing.")
        shutdown_requested_input.checkpointer.checkpoint()


if __name__ == "__main__":
    kcl_process = kcl.KCLProcess(RecordProcessor())
    kcl_process.run()
```

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 RecordProcessorCheckpointer#checkpoint(String) by default.
#validateSequenceNumberBeforeCheckpointing = true

# The maximum number of active threads for the MultiLangDaemon to permit.
# If a value is provided then a FixedThreadPool is used with the maximum
# active threads set to the provided value. If a non-positive integer or no
# value is provided a CachedThreadPool is used.
#maxActiveThreads = 0
```

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 AWS 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。 AWS 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。使用这种方式管理在 EC2 实例上运行的消费端应用程序的凭证最安全。

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 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 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 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 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 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.kinesis.clientlibrary.interfaces.v2.IRecordProcessor`	`software.amazon.kinesis.processor.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.IShutdownNotificat
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.IRecordProcessorFa` 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
`applicationName`	`ConfigsBuilder`	此 KCL 应用程序的名称。用作 `tableName` 和 `consumerName` 的默认名称。
`tableName`	`ConfigsBuilder`	允许覆盖用于 Amazon DynamoDB 租赁表的表名称。
`streamName`	`ConfigsBuilder`	此应用程序从其中处理记录的流的名称。
`kinesisEndpoint`	`ConfigsBuilder`	此选项已删除。请参阅“客户端配置删除”。
`dynamoDBEndpoint`	`ConfigsBuilder`	此选项已删除。请参阅“客户端配置删除”。
`initialPositionInStreamExtended`	`RetrievalConfig`	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。
`kinesisCredentialsProvider`	`ConfigsBuilder`	此选项已删除。请参阅“客户端配置删除”。
`dynamoDBCredentialsProvider`	`ConfigsBuilder`	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”。
failoverTimeMillis	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
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。
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	如果 Kinesis Client Library 需要创建新的 DynamoDB 租赁表，DynamoDB 将读取使用的 IOP。
initialLeaseTableWriteCapacity	LeaseManagementConfig	如果 Kinesis Client Library 需要创建新的 DynamoDB 租赁表，DynamoDB 将读取使用的 IOP。
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	N/A	此选项已删除。参见 MultiLang 移除。
timeoutInSeconds	N/A	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。
maxGetRecordsThreadPool	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
kinesisEndpoint	使用以下首选端点配置开发工具包 KinesisAsyncClient ：KinesisAsyncClient.builder().endpointOverride(URI.create("https://<kinesis endpoint>")).build() 。
dynamoDBEndpoint	使用以下首选端点配置开发工具包 DynamoDbAsyncClient ：DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://<dynamodb endpoint>")).build() 。
kinesisClientConfig	使用以下所需配置来配置开发工具包 KinesisAsyncClient ：KinesisAsyncClient.builder().overrideConfiguration(<your configuration>).build() 。
dynamoDBClientConfig	使用以下所需配置来配置开发工具包 DynamoDbAsyncClient ：DynamoDbAsyncClient.builder().overrideConfiguration(<your configuration>).build() 。
cloudWatchClientConfig	使用以下所需配置来配置开发工具包 CloudWatchAsyncClient ：CloudWatchAsyncClient.builder().overrideConfiguration(<your configuration>).build() 。
regionName	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 KinesisAsyncClient.builder().region(Region.US_WEST_2).build() 。

通过以下方式培养消费者 适用于 Java 的 AWS SDK

你可以使用 Amazon Kinesis Dat APIs a Streams 开发自定义消费者。本节介绍如何将 Kinesis Data APIs Streams 与配合 适用于 Java 的 AWS SDK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

- 使用开发吞吐量共享的消费者 适用于 Java 的 AWS SDK
- 通过以下方式培养更具吸引力的粉丝消费者 适用于 Java 的 AWS SDK
- 使用 AWS Glue 架构注册表与数据交互

使用开发吞吐量共享的消费者 适用于 Java 的 AWS SDK

开发自定义 Kinesis Data Streams 使用者的方法之一是将 Amazon Kinesis Data APIs Streams 与. 适用于 Java 的 AWS SDK本节介绍如何将 Kinesis Data APIs Streams 与配合 适用于 Java 的 AWS SDK 使用。你可以使用其他不同的编程语言调用 Kinesis Data APIs Streams。有关所有可用内容的更多信息 AWS SDKs，请参阅开始使用 Amazon Web Services 进行开发。

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 APIs Streams 包括`getShardIterator`和`getRecords`方法，您可以调用这些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，你可能更喜欢使用 Kinesis Dat APIs a Streams。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。有关如何在您的 AWS 账户中获取直播的信息，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。

通过以下方式培养更具吸引力的粉丝消费者 适用于 Java 的 AWS SDK

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. 致电RegisterStreamConsumer将您的应用程序注册为使用增强型扇出功能的使用者。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. 要开始监听特定分片，请在调用中将使用者 ARN 传递给。SubscribeToShard然后，Kinesis Data Streams 开始通过 HTTP/2 连接以SubscribeToShardEvent类型的事件形式将记录从该分片推送给你。此连接将保持打开状态长达 5 分钟。如果要在调用返回的记录正常或异常SubscribeToShard完成后继续从分片接收记录，请SubscribeToShard再次调用。future

   > **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者，请致电。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 AWS Glue 架构注册表与数据交互

您可以将 Kinesis 数据流与 AWS Glue 架构注册表集成。 AWS 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。 AWS Glue Schema Registry 使您能够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 AWS Glue Schema Registry。设置此集成的方法之一是通过 Java SDK 中 AWS 提供的 `GetRecords` Kinesis Data Streams API。

有关如何使用 Kinesis 数据流设置 Kinesis 数据流与架构注册表集成的详细说明，请参阅用`GetRecords`例：将 Amazon Kinesis 数据 APIs流与 Glue 架构注册表集成中的 “使用 Kin APIs esis 数据流与数据交互” 部分。 AWS

使用开发消费者 AWS Lambda

您可以使用 AWS Lambda 函数来处理数据流中的记录。 AWS 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 AWS 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AWS 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 AWS 其他服务从 Kinesis Data Streams 读取数据

以下 AWS 服务可以直接与 Amazon Kinesis Data Streams 集成，从 Kinesis 数据流中读取数据。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
- 使用从 Kinesis Data Streams 读取数据 AWS Glue
- 使用 Amazon Redshift 从 Kinesis Data Streams 读取数据

使用 Amazon EMR 从 Kinesis Data Streams 读取数据

亚马逊 EMR 集群可以使用 Hadoop 生态系统中熟悉的工具（例如 Hive、Pig、Hadoop Streaming API 和 Cascading）直接读取和处理 Kinesis 流。 MapReduce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 Pip EventBridge es 支持 Kinesis 数据流作为来源。Amazon Pip EventBridge es 可通过可选 point-to-point的转换、筛选和丰富步骤帮助您在活动制作者和消费者之间创建集成。您可以使用 Pip EventBridge es 接收 Kinesis 数据流中的记录，也可以选择筛选或增强这些记录，然后再将这些记录发送到可用的处理目标之一（包括 Kinesis Data Streams）。

有关更多信息，请参阅亚马逊 EventBridge 发行指南中的亚马逊 Kinesis 直播作为来源。

使用从 Kinesis Data Streams 读取数据 AWS Glue

使用 AWS Glue 流式传输 ETL，您可以创建流式提取、转换和加载 (ETL) 作业，这些任务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《AWS Glue Release Guide》中的 Streaming ETL jobs in AWS 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。它提供了 Java、Scala、Python 和 R 的高级 APIs 版本，以及支持一般执行图的优化引擎。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 构造函数出现编译错误

- 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
- GetRecords 即使流中有数据，也会返回一个空的记录数组
- 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: 更新表达式中提供的文档路径无效，无法更新
- 排查消费端的其他常见问题

LeaseManagementConfig 构造函数出现编译错误

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、dynam DBClient o、KinesisClient、StreamName、workerIdentifier）
- 之后: leaseManagementConfig = new LeaseManagementConfig（表名、应用程序名称、dynamo、KinesisClient、St reamName、workerI DBClient dentifier）

我们建议不要直接创建 LeaseManagementConfig 对象，而是在KCL 3.x及更高版本中使用`ConfigsBuilder`来设置配置。 `ConfigsBuilder`提供了一种更灵活、更易于维护的方式来配置 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,
    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 代码来处理由处理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. 由于多种可能的原因，processRecords调用中的逻辑可能需要比预期更长的时间；逻辑可能占用 CPU 密集、 I/O 阻塞或同步出现瓶颈。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。

GetRecords 即使流中有数据，也会返回一个空的记录数组

使用或获取记录是一种拉取模型。开发人员应GetRecords在没有退缩的情况下连续循环调用。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 使用亚马逊监控亚马逊 Kinesis Data Streams 服务 CloudWatch。接下来，使用 Kinesis 客户端库 (KCL) 发出的自定义 CloudWatch 指标，确定您的使用者应用程序从每个分片中读取的内容落后了多远。MillisBehindLatest有关更多信息，请参阅 使用亚马逊监控 Kinesis 客户端库 CloudWatch。

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。您可以通过查看 KCL 发出的与`processTask`操作相关的其他自定义 CloudWatch 指标（包括`RecordProcessor.processRecords.TimeSuccess`、和）来验证此行为。`RecordsProcessed`
  - 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，您有足够数量的物理资源（内存、CPU 使用率等）。

未授权的 KMS 密钥权限错误

当使用者应用程序在没有 AWS KMS 密钥权限的情况下从加密流中读取数据时，就会发生此错误。要为应用程序分配权限以访问 KMS 密钥，请参阅 Using Key Policies in AWS KMS 和 Using IAM Policies with AWS KMS。

DynamoDbException: 更新表达式中提供的文档路径无效，无法更新

在 2.27.19 至 2.27.23 适用于 Java 的 AWS SDK 版本中使用 KCL 3.x 时，您可能会遇到以下 DynamoDB 异常：

“software.amazon.awssdk.services.dynamodb DynamoDbException: 更新表达式中提供的文档路径无法更新（服务： DynamoDb，状态码：400，请求 ID：xxx）”

之所以出现此错误，是因为中存在影响由 KCL 3.x 管理 适用于 Java 的 AWS SDK 的 DynamoDB 元数据表的已知问题。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 AWS 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 Kinesis Data Streams 中的 ReadProvisionedThroughputExceeded异常并对其进行故障排除？
- 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 AWS 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).withIdleTimeBetweenReadsInMilli
```

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 AWS 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 支持取消聚合这些记录，但在用作流的使用者时，您需要使用 AWS 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。 AWS 该项目中的组件使您能够在 Java AWS 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，如果您的应用程序正在 1 个 EC2 实例上运行，并且正在处理 1 个包含 4 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出发生。

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 EC2 Auto Scaling 用户指南。

当重新分片增加了流中的分片数时，记录处理器的数量的相应增加会增加托管处理器的 EC2 实例上的负载。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 AWS SDKPutRecord，请在AWS SDKs 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，您可以结合使用版本控制和唯一版本 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 应用程序已完成其预定任务时，您应通过终止运行该应用程序的 EC2 实例来予以关闭。您可使用 AWS 管理控制台或 AWS 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。

Amazon Kinesis Data Streams 的流式传输表和 S3 交付

借助 Amazon Kinesis Data Streams，您可以将来自 Kinesis 数据流的流数据传输到两种目标类型：Apache Iceberg 上的流式传输表（亚马逊 S3 表）或通用的 Amazon S3 存储桶。您无需管理任何基础架构，只需几分钟即可开始交付。Amazon Kinesis Data Streams 从您的数据流中读取数据、缓冲和聚合记录，然后将其传送到您配置的目的地。交付是完全托管的 — 无需配置连接器、消费者应用程序或计算资源。

在 On-Demand Advantage 或 On-Demand Standard 容量模式下运行的直播支持这些功能。您可以配置直播中的传送并指定目的地。然后，Amazon Kinesis Data Streams 会自动处理扩展、重试和交付可靠性。交付不会消耗直播的读取吞吐量，也不会对现有使用者产生任何影响。

主题

- 数据交付的工作原理
- 送货目的地
- 数据流
- 关键功能
- 要求
- 数据交付概念
- 直播表
- 亚马逊 S3 一般用途交付
- 用于数据传输的 IAM 权限
- 数据传输的安全性
- 监控数据交付
- 数据交付的最佳实践
- 数据传输疑难解答
- 配送配额和限制

数据交付的工作原理

数据交付通过托管管道将您的 Kinesis 数据流连接到交付目的地：

1. 您可以在模式下将数据发布到 Kinesis 数据流。 On-Demand

2. 您可以调用CreateChannel直播并指定目标（Apache Iceberg 上的流式传输表或通用的 Amazon S3 存储桶）。
3. 交付从流中读取，缓冲记录，并将它们聚合到大小最优的文件中。
4. 交付将在您指定的数据新鲜度窗口内将文件写入您配置的目的地。

送货目的地

数据传输支持两种目标类型：

在 Apache Iceberg 上直播牌桌

流式传输表会持续将您的 Kinesis 数据流传输到存储在 Amazon S3 表中的 Apache Iceberg 表中。当数据到达时，它会自动转换为经过优化的 Apache Parquet 格式，具有智能内联压缩功能，从而消除了小文件问题并降低了下游查询成本。在数据发布到您的直播后的几分钟内，即可通过亚马逊 Athena、亚马逊 EMR、适用于 Apache Flink 的亚马逊托管服务或任何支持 Apache Iceberg 的引擎进行查询。

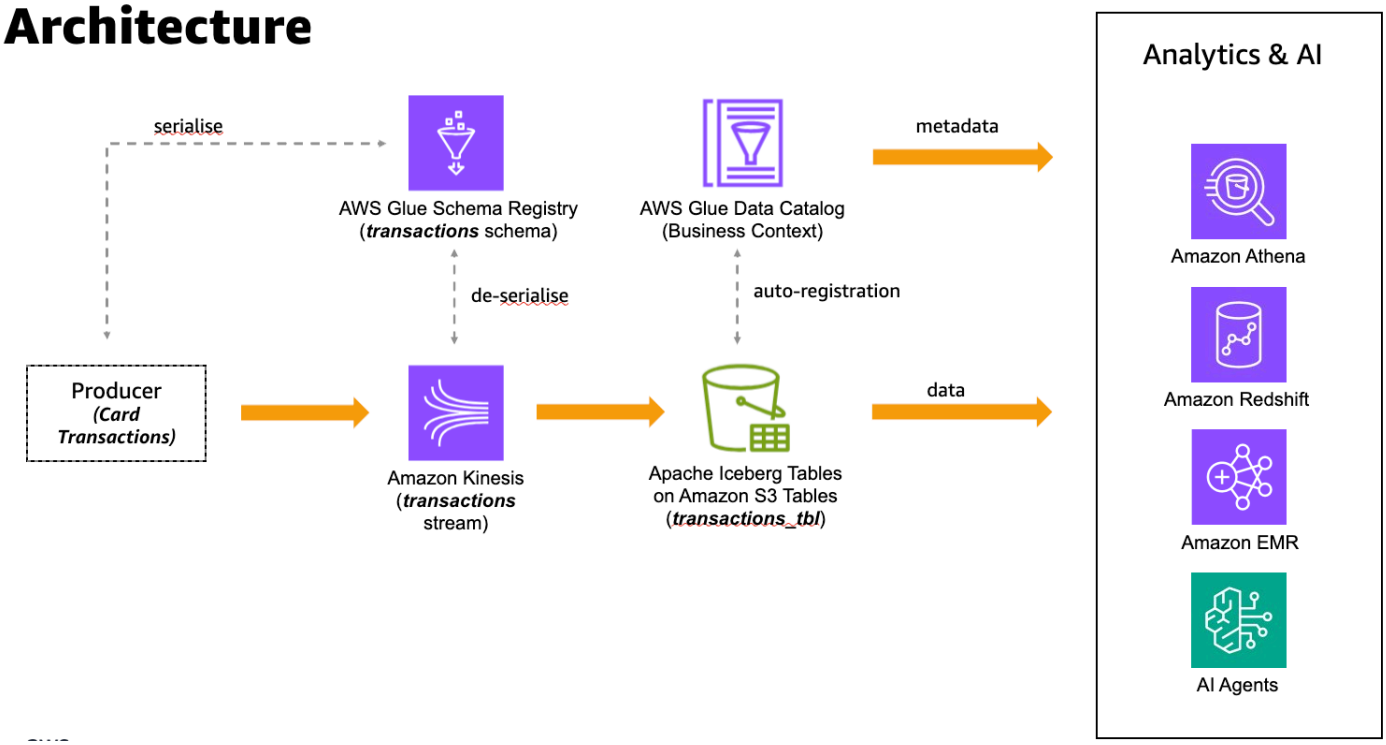
通用亚马逊 S3 存储桶

Amazon S3 交付将来自您的 Kinesis 数据流的流数据直接写入 S3 存储桶。记录以其原始源格式交付，不进行任何转换。使用可配置的压缩和通过输出密钥模板定义的 S3 密钥结构，将多个记录缓冲并批处理成大小最优的对象。这非常适合原始日志存档、事件重播和下游批处理等用例，在这些用例中，您需要持久、低成本地存储流数据，而无需管理交付管道的开销。

数据流

下图显示了传送到 Apache Iceberg 上的流式表的端到端数据流。该图以信用卡交易用例为例。生产者根据架构注册表中的 AWS Glue 架构对记录进行序列化，并将它们写入 Kinesis 数据流。Amazon Kinesis Data Streams 将记录传送到亚马逊 S3 表上的 Apache Iceberg 表。如果您在 S3 表中启用分析集成，则表元数据也会在 AWS Glue 数据目录中注册；默认情况下不会发生这种情况。然后，交付的数据及其元数据可供分析和人工智能引擎使用，例如亚马逊 Athena、Amazon Redshift 和 Amazon EMR。

向通用型 Amazon S3 存储桶的交付遵循类似的流程，但有两个区别。创建者将记录写入 Kinesis 数据流，然后 Amazon Kinesis Data Streams 将其传输到你的 S3 存储桶。记录以其原始源格式交付，无需转换，因此不需要 AWS Glue 架构注册表，也不会在数据目录中注册表元 AWS Glue 数据。Amazon Kinesis Data Streams 将记录缓冲区并批处理成大小最优的对象，并使用您通过输出密钥模板定义的 S3 密钥结构将其写入。然后，交付的对象可用于下游的批处理和分析。



关键功能

数据传输提供以下关键功能：

- Serverless auto-scaling — 根据您的流吞吐量自动扩展，最高可达流的吞吐量容量。没有可供配置的计算资源。
- Exactly-once 按分片传送 — 来自分片的记录仅传送到目的地一次，分片内没有重复或遗漏。
- 近乎实时的交付 — 可配置的数据新鲜度为 5 到 15 分钟（300 到 900 秒）。
- 自动 Parquet 转换 — 对于 Apache Iceberg 上的流媒体表，可将流媒体记录转换为优化的 Apache Parquet 格式，以实现高效的分析查询
- 内联压缩-将记录聚合到大小最优的文件中，以提高分析查询性能。
- 加密-支持客户管理的 AWS KMS 密钥在目标服务器端加密。 AWS 托管式密钥 （`aws/kinesis`别名）不支持目标加密。
- Dead-letter queu e — 无法传送的记录的故障元数据（包括流 ARN、分片 ID、序列号和错误上下文）被写入死信队列。 S3-based
- CloudWatch 指标和日志 — 通过 Amazon CloudWatch 指标监控传输的字节数、记录计数和数据新鲜度。启用 Amazon CloudWatch Logs 的传送日志，以捕获配送批次详情、失败和错误背景以进行故障排除。

- 对@@ 其他使用者没有影响-不消耗增强的扇出插槽或共享吞吐量。

要求

数据传输有以下要求：

- 您的 Kinesis 数据流必须使用 On-Demand 标准或 On-Demand 优势容量模式。
- 您必须创建一个 IAM 服务执行角色来授予写入目标的交付权限。
- 您的目标存储桶或表存储桶必须与您的 Kinesis 数据流位于同一区域。数据传输不支持任何一种目的地类型的跨区域传输。
- 对于 Apache Iceberg 上的直播表，不支持跨账户传输。源流、目标 S3 表存储桶和 AWS Glue 架构注册表必须全部位于相同 AWS 账户 且相同的区域。
- 对于一般用途 Amazon S3 存储桶，仅目标存储桶支持跨账户传输。频道及其源流必须相同 AWS 账户；只有目标存储桶可以位于不同的账户中。
- 要在 Apache Iceberg 上直播表，您必须在 Amazon S3 中配置死信队列。

数据交付概念

使用本主题来查看 Amazon Kinesis 数据流中流式传输表和 Amazon S3 交付的关键概念。

数据新鲜度

数据新鲜度定义了 Kinesis Data Streams 在将传入记录传送到目标之前缓冲这些记录的最长时间（以秒为单位）。您可以在 300 到 900 秒（5 到 15 分钟）之间配置数据新鲜度。默认值为 300 秒。

- 较低的价值 -以较小的批量更快地交付到目的地。
- 更高的值 — 批处理效率更高，在目标位置生成的文件更少、更大。

直播表的输入格式

在 Apache Iceberg 上传输到流媒体表时，数据传输支持以下输入记录格式。两种格式都需要 AWS Glue 架构注册表。

- JSON — 纯的 JSON 记录。您必须提供引用在 AWS Glue 架构注册表中注册的架构的`GSRSchemaARN`（架构注册表 AWS Glue 架构的）。

- GSR_JSON（AWS Glue 架构注册表 JSON）— 每条记录中都嵌入了架构 ID 的 JSON 记录。该架构是从架 AWS Glue 构注册表自动解析的。

通用 Amazon S3 存储桶的输入格式

当传输到通用的 Amazon S3 存储桶时，数据传输支持以下输入记录格式。不需要架构注册表。

- JSON — 纯的 JSON 记录。
- STRIN UTF-8 G — 字符串记录。
- BYTE_ARRAY — 原始二进制记录。

> **Note**
>
> 通用的 Amazon S3 存储桶不支持该GSR_JSON格式。它只能交付到Apache Iceberg上的直播桌。

架构执行

对于 Apache Iceberg 上的流式传输表，每条记录都要根据架构注册表中注册的 AWS Glue 架构进行验证。不符合架构的记录不会传送到您的流媒体表中。故障元数据被写入死信队列以进行故障排除。数据交付不支持架构演变。这包括添加新字段、移除现有字段、重命名字段或更改字段数据类型。

没有回填

您的目的地仅接收在您创建流式表或配置 S3 交付后生成的记录。直播中的现有记录不会被回填。

每次配送都有新桌子

每次直播表交付都会在目的地创建自己的 Iceberg 表。不支持传送到现有表。您在调用CreateChannel时指定表存储桶 ARN、命名空间和表名称。

Dead-letter 队列

死信队列捕获有关无法成功传送的记录的信息。

- 在 Apache Iceberg 上直播表格 — 需要死信队列。您必须为失败的记录指定 S3 存储桶和前缀。

- 通用 Amazon S3 存储桶 — 死信队列是可选的。如果未指定，则默认为带有错误前缀的相同目标存储桶。

死信队列包含记录标识符和错误上下文，而不是完整的记录有效载荷。

内联压缩

数据交付通过将分片中的记录聚合到最佳大小的文件中来执行行内压缩。这减少了目标位置的小文件数量并提高了查询性能。

配送状态

交付在其生命周期中会经历以下状态：

- 正在创建 — 正在配置交付。
- 活跃 — 交付正在交付记录。
- 更新 — 配置更改正在进行中。
- 正在删除 — 正在移除配送。
- 失败 -交付遇到无法恢复的错误。

Append-only 交货

配送仅限追加。不支持更新和更新。流中的每条记录都作为新行写入到目的地。

直播表

流媒体表持续将记录从亚马逊 Kinesis 数据流传输到由 Amazon S3 Tables 支持的 Apache Iceberg 上的流式传输表中。当数据到达时，它会通过内联压缩自动转换为经过优化的Apache Parquet格式，并可通过亚马逊雅典娜、亚马逊EMR和适用于Apache Flink的亚马逊托管服务等引擎进行查询。此功能需要架构注册表中的 AWS Glue 架构和死信队列。

主题

- 直播表交付的工作原理
- 直播表入门
- 管理直播表交付
- 直播桌的冰山行为

直播表交付的工作原理

使用本主题了解流式表传输如何将记录从您的 Amazon Kinesis Data Streams 流发送到由 S3 表存储桶支持的 Apache Iceberg 上的流式表，包括记录转换、目标表创建、分区、数据更新、死信队列和加密。

配送的工作原理

流式传输表按以下步骤将记录发送到 Apache Iceberg 上的流式传输表：

1. 读取 — 交付从 Kinesis 数据流中的所有分片中读取记录。
2. 缓冲区 — 交付缓冲区会记录直到达到数据新鲜度间隔。
3. 验证和转换 — 交付会根据架构注册表中的架构验证每条记录，并将其转换为 Apache Iceberg 表 AWS Glue 架构。验证失败的记录进入死信队列。
4. 压缩并写入 — 交付通过行内压缩将记录转换为经过优化的Apache Parquet文件，并将其写入目标Iceberg表。
5. 提交 — 交付将新文件提交到 Iceberg 表，这样数据就可以查询了。

记录转换

流式表传输需要在架构注册表中注册 AWS Glue 架构。交付使用该架构将传入记录转换为 Apache Iceberg 表架构。它支持以下输入记录格式：

- JSON — 纯的 JSON 记录。您提供的引用`GSRSchemaARN`了在架构注册表中注册的 AWS Glue 架构。
- GSR_JSON — 架构注册表序列化器在每条记录中嵌入架 AWS Glue 构 ID 的 JSON 记录。该架构是从架 AWS Glue 构注册表自动解析的。

有关 AWS Glue 架构注册表到 Iceberg 的完整类型映射和字段处理规则，请参阅。the section called “直播桌的冰山行为”

目标表

当您创建流式表交付时，Amazon Kinesis Data Streams 会在您指定的 S3 表存储桶中创建目标 Iceberg 表。提供以下各项：

- 表存储桶 ARN — 创建表的 S3 表存储桶的 ARN。

- 命名空间 — 表的命名空间。
- 表名 — 要创建的表的名称。每次交付都会创建自己的表格。您无法向现有餐桌配送。
- 分区列 — 用于按小时对表进行分区的`timestamptz`列。有关分区要求，请参见the section called “直播桌的冰山行为”。

数据新鲜度

数据新鲜度定义了传送记录之前的最大缓冲时间。您可以在 300 到 900 秒（5 到 15 分钟）之间配置此值。默认值为 300 秒。值越低，传送速度越快，而值越高，在目标位置生成的文件越少、越大。

Dead-letter 队列

流式传输表需要死信队列。您可以指定一个 Amazon S3 存储桶来接收有关验证失败的记录的信息。您提供存储桶 ARN、预期的存储桶所有者以及可选的错误输出前缀。死信队列包含记录标识符和错误上下文，而不是完整的记录有效载荷。

加密

流式表传输对 Amazon S3 中静态传输的数据进行加密。默认情况下，数据使用 Amazon S3 托管密钥 (SSE-S3) 进行加密。您可以改用客户管理的 AWS KMS 密钥 (SSE-KMS)。

> **Important**
>
> 您不能使用 AWS 托管式密钥（`aws/kinesis`别名）进行目标加密。您必须使用客户管理的 AWS KMS 密钥。此外，如果您的源 Kinesis 数据流使用加密 AWS 托管式密钥，则无法创建交付。有关源流加密要求，请参阅the section called “源流加密”。有关服务执行角色所需的 AWS KMS 权限，请参阅the section called “用于数据传输的 IAM 权限”。

Cross-account 和跨区域交付

流媒体表传输不支持跨账户或跨区域传输。源流、目标 S3 表存储桶和 AWS Glue 架构注册表都必须位于相同 AWS 账户 且相同的区域。

直播表入门

本教程将引导您创建第一个流式表交付，将记录从 Amazon Kinesis 数据流发送到 Apache Iceberg 上的流式传输表，对其进行验证并进行清理。

先决条件

在开始之前，请确保满足以下先决条件：

- 配置为 On-Demand 标准或 On-Demand优势容量模式的 Kinesis 数据流流。
- 一个 IAM 服务执行角色，授予 Kinesis 数据流向您的目的地传送记录的权限。请参阅the section called “用于数据传输的 IAM 权限”。
- 在架构注册表中注册的 AWS Glue 架构。
- 目标 Iceberg 表的 S3 表存储桶。
- 死信队列的 Amazon S3 存储桶。
- 如果使用，则已 AWS CLI 安装和配置 AWS CLI。

第 1 步：创建配送

使用 AWS 管理控制台 或创建流媒体表交付 AWS CLI。有关完整过程和所有配置选项，请参阅the section called “Create”。

第 2 步：验证配送

确认交付已达到 “活动” 状态。有关此步骤，请参阅 the section called “描述”。

第 3 步：监控交付

交付开始后，使用`AWS/Kinesis`命名空间中发布的维度为`ChannelName`和的亚马逊 CloudWatch 指标监控其运行状况`StreamName`。流媒体表交付指标前缀`DeliveryToIceberg`为。有关指标的完整列表，请参阅the section called “监控数据交付”。

步骤 4：清除

为避免持续收费，请在不再需要时删除您创建的资源。活跃的交付会继续处理记录并产生运费，直到您将其删除，任何已经写入目的地的数据都会继续产生 Amazon S3 存储费用。

1. 删除配送。请参阅the section called “删除”。
2. 如果您不再需要服务执行角色（例如，如果任何其他交付都未使用该角色），则将其删除。分离任何客户托管策略，删除任何内联策略，然后使用 IAM `detach-role-policy` `delete-role-policy`、和`delete-role`命令删除该角色。有关详细信息，请参阅 IAM 用户指南中的删除角色或实例配置文件。

3. 如果您不再需要目标表存储桶或死信队列存储桶，请在 Amazon S3 中将其删除，以避免产生存储费用。

管理直播表交付

了解先决条件和 IAM 要求后，您可以使用 AWS 管理控制台 或创建和管理流式表交付 AWS CLI。以下主题描述了每项操作。在 API 和中 AWS CLI，交付由渠道资源表示，因此操作使用渠道术语（例如，`CreateChannel`和`DescribeChannel`）。

主题

- 创建直播桌配送
- 描述直播桌配送
- 更新直播表配送
- 删除直播表配送
- 列出直播桌投放情况

创建直播桌配送

创建一种传输，将记录从 Amazon Kinesis Data Streams 流持续发送到由 S3 表存储桶支持的 Apache Iceberg 上的流式传输表。您可以使用 AWS 管理控制台 或创建配送 AWS CLI。创建交付时，Amazon Kinesis Data Streams 会在您指定的表存储桶中创建目标 Iceberg 表。

> **Note**
>
> 在创建交付之前，请完成中的先决条件the section called “开始使用”，包括中所述的 IAM 服务执行角色the section called “用于数据传输的 IAM 权限”。流式传输表需要架构注册表中的 AWS Glue 架构和死信队列。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择流式传输表，然后选择创建直播表。
3. 步骤 1：源配置。在配送详情下，输入配送名称。可接受的字符包括大写和小写字母、数字、下划线、连字符和句点。对于来源，选择要从中 On-Demand传送的直播。仅显示 On-Demand 直播。选择下一步。

4. 步骤 2：目标配置。在 “录音转换配置” 下，选择一种录音转换格式：

   - JSO@@ N — 根据架构注册表中的 AWS Glue 架构进行验证的 JSON 记录。在 Glue 架构注册表下提供架构 AR N。
   - Glue 架构注册表 JSON — 使用 AWS Glue 架构注册表 JSON 架构转换记录。

   对于 Glue 架构注册表，请指定架构 ARN 或选择浏览以选择一个架构 ARN。

5. 在目标表下，指定 Amazon Kinesis Data Streams 在哪里创建 Iceberg 表：

   - S3 表存储桶 ARN — S3 表存储桶的 ARN。选择 “浏览 S3” 以选择一个。
   - 命名空间 — 表的命名空间（仅限小写字母、数字和下划线）。
   - 表名-要创建的表的名称。
   - 列名 — 架构中用于按小时对表进行分区的日期时间字段的名称。该字段必须是架构中具有日期时间格式的字符串字段。
   - 压缩类型 — 选择 ZSTD、Snappy 或 N one。
   - 数据新鲜度-传送记录之前的最大缓冲时间（以秒为单位）（最小 300，最大 900）。

   选择下一步。

6. 步骤 3：其他配置。配置以下内容，然后选择下一步：

   - 加密配置-默认情况下，数据使用默认 Amazon S3 加密进行加密。要使用客户管理的密钥，请选择自定义加密设置（高级）。
   - 服务访问权限 — 对于 IAM 角色 ARN，选择或创建授予目标访问权限的 IAM 角色。选择创建 IAM 角色来创建一个。
   - 死信队列 — 对于 S3 存储桶 URI，请选择用于存储无法交付的记录的 Amazon S3 存储桶。（可选）设置 S3 存储桶错误输出前缀。
   - 日志传输-（可选）选择 “传送到 Amazon CloudWatch 日志” 以捕获传输诊断。
   - 标签-（可选）添加键值标签以识别和组织您的交付。

7. 步骤 4：查看并创建。查看您的配置，然后选择创建直播表。配置完成后，交付将从 “创建” 转换为 “活动”。

使用 AWS CLI

使用create-channel命令创建流式表传输。此配置需要架构注册表中的 AWS Glue 架构和死信队列：

```
aws kinesis create-channel \
    --channel-name "my-iceberg-delivery" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
    --stream-configuration-list '[
        {
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": {
                "RecordFormatType": "JSON",
                "GSRSchemaARN": "arn:aws:glue:us-east-1:111122223333:schema/my-
registry/my-schema"
            }
        }
    ]' \
    --s3-tables-destination-configuration '{
        "DataFreshnessInSeconds": 300,
        "DeadLetterQueueS3Configuration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-dlq-bucket",
            "ExpectedBucketOwner": "111122223333",
            "ErrorOutputPrefix": "dlq/"
        },
        "S3TablesConfigurationList": [
            {
                "TableBucketARN": "arn:aws:s3tables:us-east-1:111122223333:bucket/amzn-
s3-demo-table-bucket",
                "Namespace": "my_namespace",
                "TableName": "my_table",
                "CompressionType": "ZSTD",
                "PartitionSpec": {
                    "PartitionFields": [
                        {
                            "SourceName": "timestamp",
                            "Transform": "TIME_HOUR"
                        }
                    ]
                }
            }
        ]
    }'
```

该命令返回频道 ARN () ChannelARN。将其与一起使用describe-channel来验证交付是否已达到“活动” 状态。请参阅the section called “描述”。

API 参考 — 参见CreateChannel《亚马逊 Kinesis Data Streams API 参考》。

描述直播桌配送

检索流式传输表交付的配置和当前状态，包括其状态 (ChannelStatus) 和状态原因 (ChannelStatusReason)。使用它来验证交付在创建后是否达到 ACTIVE 状态，或者诊断交付处于 FAILED 状态。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择流式传输表。从配送列表中，选择要查看的配送。
3. 交付详细信息页面显示交付状态、源流、目标表、数据新鲜度、服务访问角色和死信队列配置。使用配置选项卡查看源和目标设置，使用日志和标签选项卡查看日志传输和标记。

使用 AWS CLI

使用describe-channel命令并传递返回的频道 ARN (ChannelARN)：create-channel

```
aws kinesis describe-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

响应包括传送ChannelStatus（创建、活动、更新、删除或失败）、源流配置、目标表配置和数据新鲜度设置。当交付处于活动状态时ChannelStatus，已准备好接收记录。

API 参考 — 参见DescribeChannel《亚马逊 Kinesis Data Streams API 参考》。

更新直播表配送

更新现有的直播表交付。在现有交付中，您只能更改数据新鲜度间隔 (DataFreshnessInSeconds) 和 CloudWatch 日志配置。所有其他属性在创建后都是不可变的。要更改任何其他设置，请删除配送并创建新的配送。

使用 AWS 管理控制台

您可以从其详细信息页面编辑配送。选择要更改的设置的选项卡，然后选择 “编辑”。

1. 打开 Kinesis 控制台https://console.aws.amazon.com/kinesis，选择直播表，然后选择要编辑的分发。
2. 要更改数据新鲜度设置，请在 “配置” 选项卡上选择 “编辑”。更新可编辑的设置，然后保存所做的更改。
3. 要更改日志传送或标记，请在日志和标签选项卡上，选择编辑。更新日志传送或标记，然后保存您的更改。

> **Note**
>
> 您也可以通过选择配送并使用 “操作” 菜单，从列表页面编辑或删除该配送。

使用 AWS CLI

使用`update-channel`命令更改数据新鲜度间隔。以下命令将交付更新为 600 秒的数据新鲜度间隔：

```
aws kinesis update-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111" \
    --s3-tables-destination-configuration '{
        "DataFreshnessInSeconds": 600
    }'
```

应用更改后，交付将转换为 “更新”，然后再转换为 “活动”。

API 参考 — 参见`UpdateChannel`《亚马逊 Kinesis Data Streams API 参考》。

删除直播表配送

当您不再需要直播表交付时，请将其删除。删除分发将停止从源流向目标的传输。已传送到 Iceberg 表的数据不会被删除。

> **Important**
>
> 删除配送是不可逆的。直播中未传送的记录不会被传送。已传送到目的地的数据保持完好无损且可访问。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台https://console.aws.amazon.com/kinesis，选择直播表，然后选择要删除的分发。
2. 在配送详情页面上，选择删除。
3. 在确认对话框中，确认删除。删除完成后，该配送将从列表中删除。

使用 AWS CLI

使用 `delete-channel` 命令：

```
aws kinesis delete-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
```

为避免持续收费，还应删除您为交付创建的 IAM 角色，如果您不再需要它们，还要删除目标表存储桶和死信队列存储桶。有关完整的清理演练，请参阅。the section called “开始使用”

> **Note**
>
> 直播处于活动状态时，无法将其删除。要删除该直播，请先删除与其关联的所有配送。要找到它们，请`list-channels`使用直播过滤器。请参阅the section called “列表”。

API 参考 — 参见`DeleteChannel`《亚马逊 Kinesis Data Streams API 参考》。

列出直播桌投放情况

在您的账户中列出配送，可以选择筛选到附加到特定直播的配送。使用它可以在删除直播之前查找分发，或者审核中配置的分发 AWS 区域。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择 Streaming t ables 以查看交付列表，其中包含名称、状态、源流和目标表等列。

使用 AWS CLI

使用`list-channels`命令列出当前的交付情况 AWS 区域：

```
aws kinesis list-channels
```

要仅列出附加到特定直播的分发，请提供直播过滤器：

```
aws kinesis list-channels \
    --stream-filter StreamARN="arn:aws:kinesis:us-east-1:111122223333:stream/my-stream"
```

该响应每页最多返回 100 个配送。在响应中使用分页令牌来检索其他结果。

API 参考 — 参见`ListChannels`《亚马逊 Kinesis Data Streams API 参考》。

直播桌的冰山行为

使用本主题了解流表如何将 AWS Glue 架构注册表架构中的数据映射到 Apache Iceberg 表，包括类型映射、字段处理、表维护和格式规范。

AWS Glue 架构注册表到 Iceberg 类型的映射

流式传输表将 JSON 架构类型从 AWS Glue 架构注册表映射到 Apache Iceberg 类型，如下表所示。

JSON 架构类型	Iceberg 类型	注意
`string`（纯色）	`string`	默认字符串映射。
`string`带格式 `date-time`	`timestamptz`	带时区的时间戳。
`string`带格式 `date`	`date`	日历日期。
`string`带格式 `time`	`time`	一天中的时间。
`string`带格式 `uuid`	`uuid`	通用唯一标识符。
`string`使用编码`byte`或 `base64`	`binary`	编码为 base64 的二进制数据。
`integer`（32 位或更低）	`int`	最大值或不包含最大值 <= 2^31 的值。

JSON 架构类型	Iceberg 类型	注意
integer（大于 32 位）	long	超过 32 位范围的值。
number 与 multipleOf	decimal	Fixed-point 十进制，精度源自 multipeOf。
number（纯色）	double	IEEE 754 双精度浮点数。
boolean	boolean	对还是错。
object带有命名属性	struct	命名字段映射到结构字段。
object 与 additiona lProperties	map	Key-value 使用字符串键进行映射。
array	list	元素的有序集合。
enum	string	存储为字符串的枚举值。

必填列

JSON 架构required数组中列出的字段变为 Iceberg 表中的必填列（不可为空）。如果记录中缺少必填字段，则该记录将被发送到死信队列。

分区键列

您的表必须包含一timestamptz列，该列可用于按小时进行基于时间的分区（TIME_HOUR转换）。在 AWS Glue Schema Registry 架构中，这是一个date-time格式为的string字段，它映射到 Iceberg timestamptz 类型。

无论分区引用的源列是否出现在架构的数组中，它都会自动标记为必填required列。缺少分区键值的记录将发送到死信队列。

您可以根据表的分区列启用 Amazon S3 表记录过期任务。有关更多信息，请参阅 the section called “记录到期”。

现场处理

流式传输表处理传入记录与 Iceberg 表架构之间的不匹配情况，如下所示：

- 额外字段 — 记录中存在但未在架构中定义的字段将被删除且不会写入 Iceberg 表。
- 缺少可选字段-记录中不存在的可选字段按照 Iceberg 表`null`中的方式写入。
- 缺少必填字段-如果记录中缺少必填字段，则整个记录将发送到死信队列。

嵌套限制

Streaming table 支持复杂类型（结构、映射和列表）的最大嵌套深度为 16 级。超过 16 个嵌套级别的架构将在交付创建时被拒绝。

托管表属性

流媒体表会自动管理 Iceberg 表的属性。这些属性是在创建表时设置的，不应在外部进行修改。Streaming table 使用这些属性来控制写入行为、压缩和元数据管理。

由于该表由 Amazon Kinesis Data Streams 管理，因此您不应更新其架构、修改其表属性或直接写入或删除其数据。您可以使用 AWS 分析服务和任何兼容的 Apache Iceberg 查询引擎来查询该表。有关托管表存储桶的更多信息，请参阅 Amazon S3 用户指南中的使用 AWS 托管表存储桶。

S3 表类数据存储服务维护

向由 S3 表支持的 Apache Iceberg 上的流式传输表时，将自动处理以下维护操作：

- 压缩 — 定期将小型数据文件压缩成较大的文件，以提高查询性能并减少元数据开销。
- 快照过期 — 清理过期的快照以回收元数据存储空间并减小表元数据大小。
- 未引用的文件清理-删除任何快照都不再引用的孤立数据文件以回收存储空间。

记录到期

您可以为 Iceberg 表配置记录到期时间。启用后，Amazon S3 表会在维护操作期间根据表的`timestamptz`分区列自动删除早于指定保留期的记录。有关更多信息，请参阅 Amazon S3 用户指南中的管理 Amazon S3 表记录到期时间。

格式规范

直播表对 Iceberg 表使用以下格式规范：

- Iceberg 格式版本 — v 2
- 冰山规格版本 — 1.9. 0
- 文件格式 — Apache 实木复合地板

亚马逊 S3 一般用途交付

亚马逊 S3 通用交付将来自亚马逊 Kinesis 数据流的流数据直接写入通用亚马逊 S3 存储桶。记录以其原始源格式交付，不进行任何转换，通过可配置的压缩和您定义的 S3 密钥结构分批成大小最佳的对象。这非常适合原始日志存档、事件重放和下游批处理等用例。

主题

- 亚马逊 S3 交付的工作原理
- S3 通用交付入门
- 管理 Amazon S3 一般用途交付
- 用于 Amazon S3 交付的 S3 输出密钥模板

亚马逊 S3 交付的工作原理

使用本主题了解 Amazon S3 交付如何将记录从 Amazon Kinesis 数据流发送到通用亚马逊简单存储服务存储桶，包括存储配置、压缩、存储类别、记录格式和加密。

配送的工作原理

Amazon S3 交付通过以下步骤将记录发送到通用的 Amazon S3 存储桶：

1. 读取 — 交付从 Kinesis 数据流中的所有分片中读取记录。
2. 缓冲区 — 对记录进行缓冲，直到达到数据新鲜度间隔。
3. 验证 — 当记录格式为时`STRING`，每条记录都被验证为有效 UTF-8 字符串；当记录格式为时`JSON`，每条记录都被验证为有效的 JSON 负载。验证失败的记录将发送到死信队列。
4. 压缩和格式化 — 根据配置的输出格式对记录进行格式化，并使用指定的压缩算法进行压缩。
5. 交付 — 使用配置的密钥模板将生成的对象写入目标 Amazon S3 存储桶。
6. Dead-letter 队列 — 验证失败的记录将写入带有错误上下文的死信队列。

存储配置

创建 Amazon S3 交付时，您需要指定以下存储配置：

- bucketARN — 目标 Amazon S3 存储桶的 ARN。
- ExpectedBucketOwner— 目标存储桶的预期所有者的 12 位 AWS 账户 ID。此值为必填值，有助于防止在所有权发生变化时传送到意外存储桶。

- CompressionType— 应用于交付对象的压缩类型。此值为必填值；指定NONE在不压缩的情况下交付对象。请参阅the section called “压缩选项”。
- StorageClass— 已交付对象的 Amazon S3 存储类别（可选）。请参阅the section called “存储类”。
- OutputKeyTemplate— 用于构造 Amazon S3 对象密钥的模板（可选）。请参阅the section called “用于 Amazon S3 交付的 S3 输出密钥模板”。

压缩选项

您必须指定压缩类型才能传输到通用 Amazon S3 存储桶。要在不压缩的情况下交付对象，请选择NONE。选择以下压缩类型之一：

- GZIP — 标准的 gzip 压缩。压缩比和速度的良好平衡。
- ZSTD — Z 标准压缩。更高的压缩比和更快的解压速度。
- 无 — 未应用压缩。记录未经压缩传送。

存储类

您可以选择为交付的对象指定 Amazon S3 存储类别。如果未指定，则默认存储类别为标准。支持的存储类别：

- 标准 — 用于存放经常访问数据的通用存储。
- INTELLIGENT_TIERING — 根据使用模式自动在接入层之间移动数据。
- GLACIER_IR — 低成本存储，用于存储需要立即检索的数据。

有关 Amazon S3 存储类的更多信息，请参阅 A mazon S3 用户指南中的了解和管理存储类别。

输出键模板

您可以配置输出密钥模板来控制已交付文件的 Amazon S3 对象密钥结构。该模板支持日期、时间和交付元数据的动态占位符，用于在目标存储桶中组织数据。有关默认模板、支持的变量和示例，请参阅the section called “用于 Amazon S3 交付的 S3 输出密钥模板”。

数据新鲜度

数据新鲜度定义了传送记录之前的最大缓冲时间。您可以在 300 到 900 秒（5 到 15 分钟）之间配置此值。默认值为 300 秒。值越低，交付速度越快，而值越高生成的对象越少、越大。

Dead-letter 队列

死信队列是可选的，可以传送到通用的 Amazon S3 存储桶。如果您未指定死信队列配置，则传送会默认将错误信息写入带有错误前缀的同一个目标存储桶。

如果您指定死信队列，则需要提供以下内容：

- bucketARN — Amazon S3 存储桶的 ARN，用于接收有关无法传送的记录的信息。
- ExpectedBucketOwner— 死信队列存储桶的预期所有者的 12 位 AWS 账户 ID（指定死信队列时为必填项）。
- ErrorOutputPrefix— 用于在存储桶内组织故障元数据对象的可选前缀。

录制格式

Amazon S3 交付支持以下记录格式以传输到通用亚马逊 S3 存储桶：

- JSON — 纯的 JSON 记录。
- STRIN UTF-8 G — 字符串记录。
- BYTE_ARRAY — 原始二进制记录。

> **Note**
>
> 不支持将GSR_JSON记录格式传输到通用 Amazon S3 存储桶。它仅支持传送到需要 AWS Glue 架构注册表的 Apache Iceberg 上的流媒体表。

Cross-account 交货

亚马逊 S3 交付支持跨账户传输到通用亚马逊 S3 存储桶。源流和目标存储桶必须在同一个存储桶中 AWS 区域。配置目标存储桶策略以授予交付 IAM 角色写入对象的权限。

加密

Amazon S3 交付支持使用客户管理的 AWS KMS 密钥对交付到通用的 Amazon S3 存储桶的对象进行服务器端加密。您可以指定客户管理的 AWS KMS 密钥来加密交付的对象。

> **Important**
>
> 您不能使用aws/kinesis AWS KMS 别名进行目标加密。您必须使用客户管理的 AWS KMS 密钥。

如果您的源 Kinesis 数据流已加密，则需要考虑其他注意事项。不支持使用 AWS 托管式密钥（aws/kinesis别名）加密的流，服务执行角色需要 AWS KMS 权限才能解密源记录。有关源流加密要求，请参阅the section called “源流加密”。有关服务执行角色所需的 AWS KMS 权限，请参阅the section called “用于数据传输的 IAM 权限”。

S3 通用交付入门

本教程指导您创建第一个 Amazon S3 交付，将记录从 Amazon Kinesis 数据流发送到通用的 Amazon S3 存储桶，对其进行验证并进行清理。

先决条件

在开始之前，请确保满足以下先决条件：

- 配置为 On-Demand 标准或 On-Demand优势容量模式的 Kinesis 数据流流。
- 一个 IAM 服务执行角色，授予 Kinesis 数据流向您的目的地传送记录的权限。请参阅the section called “用于数据传输的 IAM 权限”。
- 用于目标的通用 Amazon S3 存储桶。
- 如果使用，则已 AWS CLI 安装和配置 AWS CLI。

第 1 步：创建配送

使用 AWS 管理控制台 或创建 Amazon S3 交付 AWS CLI。有关完整过程和所有配置选项，请参阅the section called “Create”。

第 2 步：验证配送

确认交付已达到 “活动” 状态。有关此步骤，请参阅 the section called “描述”。

第 3 步：监控交付

交付开始后，使用AWS/Kinesis命名空间中发布的维度为ChannelName和的亚马逊 CloudWatch 指标监控其运行状况StreamName。亚马逊 S3 交付指标前缀DeliveryToS3为。有关指标的完整列表，请参阅the section called “监控数据交付”。

步骤 4：清除

为避免持续收费，请在不再需要时删除您创建的资源。活跃的交付会继续处理记录并产生运费，直到您将其删除，任何已经写入目的地的数据都会继续产生 Amazon S3 存储费用。

1. 删除配送。请参阅the section called “删除”。
2. 如果您不再需要服务执行角色（例如，如果任何其他交付都未使用该角色），则将其删除。分离任何客户托管策略，删除任何内联策略，然后使用 IAM detach-role-policy delete-role-policy、和delete-role命令删除该角色。有关详细信息，请参阅 IAM 用户指南中的删除角色或实例配置文件。
3. 如果您不再需要目标存储桶或死信队列存储桶，请在 Amazon S3 中将其删除，以免产生存储费用。

管理 Amazon S3 一般用途交付

了解先决条件和 IAM 要求后，您可以使用 AWS 管理控制台 或创建和管理 Amazon S3 交付 AWS CLI。以下主题描述了每项操作。在 API 和中 AWS CLI，交付由渠道资源表示，因此操作使用渠道术语（例如，CreateChannel和DescribeChannel）。

主题

- 创建亚马逊 S3 配送
- 描述亚马逊 S3 的配送
- 更新亚马逊 S3 配送
- 删除亚马逊 S3 配送
- 列出亚马逊 S3 配送

创建亚马逊 S3 配送

创建一个传输，将记录从 Amazon Kinesis Data Streams 流发送到通用的 Amazon S3 存储桶。您可以使用 AWS 管理控制台 或创建配送 AWS CLI。记录以源格式传送，批处理成大小最优的对象，具有可配置的压缩和由您定义的 S3 密钥结构。

> **Note**
>
> 在创建交付之前，请完成中的先决条件the section called “开始使用”，包括中所述的 IAM 服务执行角色the section called “用于数据传输的 IAM 权限”。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择 S3 通用交付，然后选择创建交付。
3. 步骤 1：源配置。在配送详情下，输入配送名称。对于来源，选择要从中 On-Demand 传送的直播。仅显示 On-Demand 直播。选择下一步。
4. 步骤 2：目标配置。选择一种记录格式（JSON、STRING 或 BYTE_ARRAY），然后指定目标：

   - S3 存储桶 ARN — 目标通用型 Amazon S3 存储桶的 ARN。
   - 预期存储桶所有者-预期存储桶所有者的 12 位 AWS 账户 ID。
   - 压缩类型 -选择 GZIP、ZSTD 或无。
   - 存储类别 — 可以选择一个 Amazon S3 存储类别。
   - 输出密钥模板-可选择定义 Amazon S3 对象密钥结构。有关更多信息，请参阅 the section called “用于 Amazon S3 交付的 S3 输出密钥模板”。
   - 数据新鲜度-传送记录之前的最大缓冲时间（以秒为单位）（最小 300，最大 900）。

   选择下一步。
5. 步骤 3：其他配置。配置加密、服务访问权限（IAM 角色）、可选的死信队列、日志传输和标签。选择下一步。
6. 步骤 4：查看并创建。查看您的配置，然后选择创建配送。配置完成后，交付将从 “创建” 转换为 “活动”。

使用 AWS CLI

使用`create-channel`命令创建 Amazon S3 交付，将记录发送到采用 JSON 格式和 GZIP 压缩的通用亚马逊 S3 存储桶：

```
aws kinesis create-channel \
    --channel-name "my-s3-delivery" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
```

```
    --stream-configuration-list '[
        {
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": {
                "RecordFormatType": "JSON"
            }
        }
    ]' \
    --s3-destination-configuration '{
        "DataFreshnessInSeconds": 300,
        "StorageConfiguration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-destination-bucket",
            "ExpectedBucketOwner": "111122223333",
            "CompressionType": "GZIP",
            "StorageClass": "STANDARD"
        }
    }'
```

该命令返回频道 ARN () ChannelARN。将其与一起使用describe-channel来验证交付是否已达到 “活动” 状态。请参阅the section called “描述”。要自定义 Amazon S3 对象密钥结构，请参阅the section called “用于 Amazon S3 交付的 S3 输出密钥模板”。

API 参考 — 参见CreateChannel《亚马逊 Kinesis Data Streams API 参考》。

描述亚马逊 S3 的配送

检索 Amazon S3 交付的配置和当前状态，包括其状态 (ChannelStatus) 和状态原因 (ChannelStatusReason)。使用它来验证交付在创建后是否达到 ACTIVE 状态，或者诊断交付处于 FAILED 状态。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择 S3 通用交付。从配送列表中，选择要查看的配送。
3. 交付详细信息页面显示交付状态、源流、目标存储桶、压缩、存储类别、输出密钥模板、数据新鲜度、服务访问角色和死信队列配置。使用配置选项卡查看源和目标设置，使用日志和标签选项卡查看日志传输和标记。

使用 AWS CLI

使用describe-channel命令并传递返回的频道 ARN (ChannelARN)：create-channel

```
aws kinesis describe-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

响应包括交付ChannelStatus（创建、活动、更新、删除或失败）、源流配置、目标存储桶配置和数据新鲜度设置。当交付处于活动状态时ChannelStatus，已准备好接收记录。

API 参考 — 参见DescribeChannel《亚马逊 Kinesis Data Streams API 参考》。

更新亚马逊 S3 配送

更新现有的 Amazon S3 配送。在现有交付中，您只能更改数据新鲜度间隔 (DataFreshnessInSeconds) 和 CloudWatch 日志配置。所有其他属性在创建后都是不可变的。要更改任何其他设置，请删除配送并创建新的配送。

使用 AWS 管理控制台

您可以从其详细信息页面编辑配送。选择要更改的设置的选项卡，然后选择 “编辑”。

1. 打开 Kinesis 控制台https://console.aws.amazon.com/kinesis，选择 S3 通用交付，然后选择要编辑的交付。
2. 要更改数据新鲜度设置，请在配置选项卡上选择编辑。更新可编辑的设置，然后保存所做的更改。
3. 要更改日志传送或标记，请在日志和标签选项卡上，选择编辑。更新日志传送或标记，然后保存您的更改。

> **Note**
>
> 您也可以通过选择配送并使用 “操作” 菜单，从列表页面编辑或删除该配送。

使用 AWS CLI

使用update-channel命令更改数据新鲜度间隔。以下命令将交付更新为 600 秒的数据新鲜度间隔：

```
aws kinesis update-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111" \
    --s3-destination-configuration '{
        "DataFreshnessInSeconds": 600
```

```
    }'
```

应用更改后，交付将转换为 “更新”，然后再转换为 “活动”。

API 参考 — 参见`UpdateChannel`《亚马逊 Kinesis Data Streams API 参考》。

删除亚马逊 S3 配送

当您不再需要某个 Amazon S3 配送时，将其删除。删除分发将停止从源流向目标的传输。已传输到 Amazon S3 存储桶的数据不会被删除。

> **Important**
>
> 删除配送是不可逆的。直播中未传送的记录不会被传送。已传送到目的地的数据保持完好无损且可访问。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台https://console.aws.amazon.com/kinesis，选择 S3 通用交付，然后选择要删除的交付。
2. 在配送详情页面上，选择删除。
3. 在确认对话框中，确认删除。删除完成后，该配送将从列表中删除。

使用 AWS CLI

使用 `delete-channel` 命令：

```
aws kinesis delete-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

为避免持续收费，还应删除您为交付创建的 IAM 角色，如果您不再需要它们，还要删除目标存储桶和死信队列存储桶。有关完整的清理演练，请参阅。the section called “开始使用”

> **Note**
>
> 直播处于活动状态时，无法将其删除。要删除该直播，请先删除与其关联的所有配送。要找到它们，请`list-channels`使用直播过滤器。请参阅the section called “列表”。

API 参考 — 参见DeleteChannel《亚马逊 Kinesis Data Streams API 参考》。

列出亚马逊 S3 配送

在您的账户中列出配送，可以选择筛选到附加到特定直播的配送。使用它可以在删除直播之前查找分发，或者审核中配置的分发 AWS 区域。

使用 AWS 管理控制台

1. 打开 Kinesis 控制台，网址为。https://console.aws.amazon.com/kinesis
2. 在导航窗格中，选择 S3 通用交付以查看交付列表，其中包含名称、状态、源流和目标存储桶等列。

使用 AWS CLI

使用list-channels命令列出当前的交付情况 AWS 区域：

```
aws kinesis list-channels
```

要仅列出附加到特定直播的分发，请提供直播过滤器：

```
aws kinesis list-channels \
    --stream-filter StreamARN="arn:aws:kinesis:us-east-1:111122223333:stream/my-stream"
```

该响应每页最多返回 100 个配送。在响应中使用分页令牌来检索其他结果。

API 参考 — 参见ListChannels《亚马逊 Kinesis Data Streams API 参考》。

用于 Amazon S3 交付的 S3 输出密钥模板

向通用的 Amazon S3 存储桶传输时，您可以配置 S3 输出密钥模板来控制已交付文件的对象密钥结构。密钥模板是可选的——如果未指定，Amazon S3 交付将使用默认模板。

Default 模板

如果您未指定自定义密钥模板，Amazon S3 交付将使用以下默认值：

```
kinesis-channel/!{channel-name}/!{channel-id}/!{yyyy}/!{MM}/!{dd}/!{HH}/!{channel-
name}-!{channel-id}-!{yyyy}-!{MM}-!{dd}-!{HH}-!{mm}!{extension}
```

最后一个正斜杠 (/) 之前的部分构成对象键前缀（文件夹结构），最后一段构成对象名称。默认模板按渠道和小时组织交付的对象，并在对象名称中重复相同的值，这样每个对象都是自描述的。Amazon Kinesis Data Streams 会自动为每个对象密钥附加一个唯一的后缀，因此即使模板中没有足够精细的时间分量来区分它们，对象密钥也是唯一的。

模板变量

以下变量可在您的密钥模板中使用：

S3 密钥模板变量

变量	说明	示例值
`!{channel-name}`	交付的名称。	`my-channel`
`!{channel-id}`	交付的唯一标识符。	`abc123def456`
`!{stream-name}`	源 Kinesis 数据流的名称。	`my-stream`
`!{yyyy}`	Four-digit 年 (UTC)。	`2026`
`!{yy}`	Two-digit 年 (UTC)。	`26`
`!{MM}`	Two-digit 月 (UTC)，零填充。	`07`
`!{dd}`	Two-digit 月中某一天 (UTC)，填充为零。	`20`
`!{HH}`	Two-digit 小时（UTC，24 小时），零填充。	`14`
`!{mm}`	Two-digit 分钟 (UTC)，零填充。	`30`
`!{extension}`	从配置的压缩类型（例如，对于 GZIP 或 ZSTD）派生`.zst`的文件扩展名。`.gz`如果存在，它必须出现在模板的末尾。	`.gz`
`!{extension:. literal}`	您指定的文字文件扩展名。必须以句点开头，并且仅包含小写字母和句点。如果存在，它必须出现在模板的末尾。	`!{extension:.json}`

模板规则

密钥模板必须符合以下规则：

- 扩展所有变量后，生成的对象密钥不得超过 Amazon S3 对象密钥限制，即 1,024 个字符。由于 Amazon Kinesis 数据流会附加最多 38 个字符的唯一后缀，因此扩展模板本身不得超过 986 个字符。
- 不得包含路径遍历：没有连续的句点 (`..`)，没有仅限句点的路径段 (`.`)，并且不得以正斜杠 () 开头。`/`
- 不得包含连续的斜杠 (`//`)。
- 每个变量占位符都必须关闭（每个占位符`!{`必须有匹配`}`项），并且必须使用支持的变量名称。
- 允许使用的文字字符是字母数字字符和以下字符:`. ! - _ ' . * ( ) / =` 该=角色可以让你表达 Hive-style 分区，例如`year=!{yyyy}/month=!{MM}/`。
- 模板最多可以包含一个扩展占位符（`!{extension}`或`!{extension:.`*`literal`*`}`），如果存在，它必须出现在模板的末尾。
- 启用压缩（GZIP 或 ZSTD）时，模板必须包含扩展占位符。

> **Note**
>
> 您无需在模板中包含唯一性变量。Amazon Kinesis 数据流会自动为每个交付的对象密钥附加一个唯一的后缀。

有效和无效的示例

关键模板示例

模板	有效	Reason
`!{channel-name}/!{yyyy}/!{MM}/!{dd}/!{HH}/records!{extension}`	是	使用支持的变量并以单个扩展占位符结尾。
`data/stream=!{stream-name}/!{yyyy}-!{MM}-!{dd}/!{HH}!{mm}!{extension:.json}`	是	Hive-style 最后用文字扩展名进行分区。

模板	有效	Reason
`!{channel-name}/!{channel-id}`	是	最小模板。Kinesis 数据流会自动附加一个唯一的后缀。
`/data/!{channel-name}!{extension}`	否	以正斜线开头。
`data/../!{channel-name}!{extension}`	否	包含路径遍历 () `..`。
`data//!{channel-name}!{extension}`	否	包含连续的斜杠 (//)。
`!{channel-name}!{extension}/!{yyyy}`	否	扩展占位符不在模板的末尾。
`data/!{partition-id}!{extension}`	否	`!{partition-id}` 不是支持的变量。
`data/!{channel-name}` （使用 GZIP 或 ZSTD 压缩）	否	压缩已启用，但模板没有扩展占位符。
`data/!{channel-name}/!{yyyy}` （缺少右大括号）	否	未封闭的变量占位符。

用于数据传输的 IAM 权限

使用本主题了解在 Amazon Kinesis Data Streams 中创建和管理交付资源所需的 IAM 权限，以及数据交付在将数据传输到目标时所承担的服务执行角色。

生命周期管理权限

要创建、更新、删除、描述和列出交付资源，调用的 IAM 委托人必须具有以下权限。

处理建议	资源	说明
kinesis:CreateChannel	直播 ARN () arn:aws:kinesis: *region*:*account-id* :stream/*stream-name*	在 Kinesis Data Streams 流上创建分发。
kinesis:AssociateStreamsWithChannel	直播 ARN () arn:aws:kinesis: *region*:*account-id* :stream/*stream-name*	将直播与配送关联。这是一个虚拟操作，必须授予调用主体权限才能创建从指定流中读取数据的传送。除此之外，它还获得授权kinesis:CreateChannel 。
kinesis:UpdateChannel	频道 ARN () arn:aws:kinesis: *region*:*account-id* :channel/ *channel-id*	更新现有的交付配置。
kinesis:DeleteChannel	频道 ARN	删除现有配送。
kinesis:DescribeChannel	频道 ARN	检索配送的详细信息。
kinesis:ListChannels	账户等级 (arn:aws:kinesis: *region*:*account-id* :stream/*)	列出账户中的所有配送。

服务执行角色

数据交付扮演 IAM 服务执行角色，将数据传送到您的目的地。您必须创建此角色并附加信任策略和权限策略。

信任策略

信任策略允许 Kinesis Data Streams 服务担任该角色。包括条件键以防止出现混乱的副手问题。

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "kinesis.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "<account-id>"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:kinesis:<region>:<account-id>:channel/*"
                }
            }
        }
    ]
}
```

Apache Iceberg 上直播表格的权限策略

在 Apache Iceberg 上向流式传输表时，服务执行角色需要以下权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3TablesAccess",
            "Effect": "Allow",
            "Action": [
                "s3tables:GetTable",
                "s3tables:GetTableBucket",
                "s3tables:GetTableMetadataLocation",
                "s3tables:UpdateTableMetadataLocation",
                "s3tables:CreateTable",
                "s3tables:CreateNamespace",
                "s3tables:PutTableData",
                "s3tables:GetTableData",
                "s3tables:TagResource",
                "s3tables:PutTableRecordExpirationConfiguration",
                "s3tables:PutTableEncryption"
            ],
            "Resource": [
```

```
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>",
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>/
table/*"
            ]
        },
        {
            "Sid": "AllowCreateTableWithTag",
            "Effect": "Allow",
            "Action": "s3tables:CreateTable",
            "Resource": [
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>/
table/*",
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/TableName": "<table-name>"
                }
            }
        },
        {
            "Sid": "AllowPutTableDataWithTag",
            "Effect": "Allow",
            "Action": "s3tables:PutTableData",
            "Resource": "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-
name>/table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/TableName": "<table-name>"
                }
            }
        },
        {
            "Sid": "DLQBucketAccess",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<dlq-bucket-name>",
                "arn:aws:s3:::<dlq-bucket-name>/*"
            ],
```

```
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "<dlq-account-id>"
                }
            }
        },
        {
            "Sid": "GlueSchemaRegistryAccess",
            "Effect": "Allow",
            "Action": [
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "arn:aws:glue:<region>:<account-id>:registry/<registry-name>",
                "arn:aws:glue:<region>:<account-id>:schema/<registry-name>/<schema-
name>"
            ]
        },
        {
            "Sid": "KMSForCreateTimeValidation",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForS3TablesEncryption",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3.<region>.amazonaws.com"
                },
                "StringLike": {
                    "kms:EncryptionContext:aws:s3:arn": [
                        "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-
name>/*",
                        "arn:aws:s3:::<dlq-bucket-name>/*"
                    ]
```

```
                }
            }
        },
        {
            "Sid": "KMSForCreateTable",
            "Effect": "Allow",
            "Action": ["kms:DescribeKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3tables.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForDecryptSourceStreamRecords",
            "Effect": "Allow",
            "Action": "kms:Decrypt",
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<stream-key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                },
                "StringEquals": {
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:<region>:<account-id>:stream/<stream-name>"
                }
            }
        }
    ]
}
```

- s3tables:PutTableEncryption只有在使用客户管理的密 AWS KMS 钥加密目标表时才需要。如果没有它，CreateTable则会成功，但是表加密失败，并且永远不会创建表。
- s3tables:TagResource是必需的，因为该服务会对其创建的表进行标记。
- AllowCreateTableWithTag和AllowPutTableDataWithTag语句是可选的。只有当你想要基于标签的访问控制而不是仅根据表 ARN 来确定范围时，才包括它们。它们将s3tables:CreateTable和限制s3tables:PutTableData在带有匹配TableName标签的资源上。如果您按照 ARN 确定范围，请省略它们。
- 只有 G glue:GetSchemaVersion lue 架构注册表是必需的。JSON和GSR_JSON输入格式都需要它。

- 该KMSForCreateTimeValidation声明允许 Amazon Kinesis Data Streams 在您创建交付时验证 AWS KMS 您的密钥。只有当您的目标表或源流使用客户托管 AWS KMS 密钥时，才需要这样做。
- 该KMSForS3TablesEncryption语句授予用于加密已交付的静态表数据的 AWS KMS 操作。只有当您的表存储桶使用客户托管密钥时，才需要这样做。
- 该KMSForCreateTable语句允许 Amazon Kinesis Data Streams 在创建目标表时读取密钥配置。只有当表存储桶使用客户托管密钥时，才需要这样做。
- 该KMSForDecryptSourceStreamRecords语句允许角色解密从源流中读取的记录。只有当你的 Kinesis Data Streams 数据流使用客户管理的密钥加密时，才需要这样做。

通用型 Amazon S3 存储桶的权限策略

在向通用的 Amazon S3 存储桶传送数据时，服务执行角色需要以下权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DeliveryBucketList",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<delivery-bucket-name>",
                "arn:aws:s3:::<delivery-bucket-name>/*"
            ]
        },
        {
            "Sid": "DeliveryBucketWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:CreateMultipartUpload",
                "s3:UploadPart",
                "s3:CompleteMultipartUpload",
                "s3:ListMultipartUploads",
                "s3:ListMultipartUploadParts"
            ],
            "Resource": [
                "arn:aws:s3:::<delivery-bucket-name>/*"
```

```
            ]
        },
        {
            "Sid": "DLQBucketAccess",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<dlq-bucket-name>",
                "arn:aws:s3:::<dlq-bucket-name>/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "<dlq-account-id>"
                }
            }
        },
        {
            "Sid": "KMSForCreateTimeValidation",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForS3BucketEncryption",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3.<region>.amazonaws.com"
                },
                "StringLike": {
                    "kms:EncryptionContext:aws:s3:arn": [
                        "arn:aws:s3:::<delivery-bucket-name>/*",
                        "arn:aws:s3:::<dlq-bucket-name>/*"
```

```
                    ]
                }
            }
        },
        {
            "Sid": "KMSForDecryptSourceStreamRecords",
            "Effect": "Allow",
            "Action": "kms:Decrypt",
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<stream-key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                },
                "StringEquals": {
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:<region>:<account-id>:stream/<stream-name>"
                }
            }
        }
    ]
}
```

- 该KMSForCreateTimeValidation声明允许 Amazon Kinesis Data Streams 在您创建交付时验证 AWS KMS 您的密钥。只有当您的目标存储桶或源流使用客户托管 AWS KMS 密钥时，才需要这样做。
- 该KMSForS3BucketEncryption语句授予 Amazon S3 用于加密已交付的静态对象的 AWS KMS 操作。只有当您的目标存储桶或死信队列存储桶与客户托管密钥 SSE-KMS 一起使用时，才需要使用它。显示的kms:EncryptionContext:aws:s3:arn值使用对象 ARN，该对象在 S3 存储桶密钥被禁用时适用。如果您在目标存储桶或死信队列存储桶上启用 S3 存储桶密钥，Amazon S3 将使用存储桶 ARN 而不是对象 ARN 作为加密上下文，因此您必须将相应值更改为该存储桶 ARN（例如，或）。arn:aws:s3:::*delivery-bucket-name* arn:aws:s3:::*dlq-bucket-name*有关更多信息，请参阅 Amazon S3 用户指南中的配置 S 3 存储桶密钥。
- 该KMSForDecryptSourceStreamRecords语句允许角色解密从源流中读取的记录。只有当你的 Kinesis Data Streams 数据流使用客户管理的密钥加密时，才需要这样做。

> **Important**
>
> 如果您将s3:PutObject资源限制为特定的前缀（例如arn:aws:s3:::my-bucket/data*），则输出密钥模板生成的对象密钥必须以该相同的前缀开头。IAM 资源前缀和输出密钥模板不匹配会导致访问被拒绝错误，并且不会传送任何数据。
> 例如，如果您的策略授予s3:PutObject启用arn:aws:s3:::my-bucket/data*，则您的输出密钥模板必须以开头data/，例如data/!{yyyy}/!{MM}/!{dd}/!{HH}/。为避免这种情况，要么将资源范围限定为整个存储桶 (arn:aws:s3:::my-bucket/*)，要么确保前缀完全匹配。

可选 CloudWatch 日志权限

如果您为交付启用 CloudWatch 日志，请向服务执行角色添加以下权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CloudWatchLogsAccess",
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:<region>:<account-id>:log-group:<log-group-
name>:*"
        }
    ]
}
```

数据传输的安全性

使用本主题了解在 Amazon Kinesis Data Streams 中传输数据的安全功能和最佳实践，包括加密、访问控制和审计日志。

传输中加密

数据传输与其目的地之间的所有通信在传输过程中均使用 TLS 1.2 或更高版本进行加密。这包括通过Apache Iceberg和通用Amazon S3存储桶传输到流媒体表的数据，以及与 AWS Glue 架构注册表和死信队列的通信。

静态加密

流媒体表和亚马逊 S3 交付存储在 Amazon S3 中传输数据。所有交付的数据均在服务器端静态加密。这适用于两种目标类型，因为Apache Iceberg上的流式传输表和通用Amazon S3存储桶都将数据存储在Amazon S3中。您无需采取任何操作即可对交付的数据进行加密。

静态加密选项

您可以为传送的数据选择以下加密选项之一：

- SSE-S3 （默认）— 使用 Amazon S3 托管密钥进行 Server-side加密。如果未指定客户管理密钥，则这是默认设置。无需其他配置。
- SSE-KMS 使用客户管理的密钥 — 使用您拥有和管理的对称客户管理 AWS KMS 密钥进行 Server-side 加密。

使用客户管理的密钥进行静态加密

由于两种目标类型的交付数据都存储在 Amazon S3 中，因此静态加密与 Amazon S3 服务器端加密的工作方式相同，包括 AWS KMS 密钥的使用方式、所需的密钥权限、加密上下文和审计。仅支持对称加密 AWS KMS 密钥。有关 Amazon S3 如何使用 AWS KMS 密钥进行服务器端加密的完整详细信息，请参阅 Amaz on S3 用户指南中的使用带 AWS KMS 密钥的服务器端加密 (SSE-KMS)。

要使用客户管理的密钥加密传送的数据，请在使用创建交付时在`EncryptionConfiguration`参数中指定密钥`CreateChannel`。有关更多信息，请参阅《亚马逊 Kinesis 数据流 API 参考》中的CreateChannel操作。创建交付后，加密配置是不可变的；要对其进行更改，请删除交付并创建新的交付。

有关服务执行角色需要使用客户管理密钥的特定 AWS KMS 权限，请参阅the section called “用于数据传输的 IAM 权限”。

> **Note**
>
> 您必须使用客户管理的 AWS KMS 密钥进行目标加密。不支持 AWS 托管式密钥 （ aws/kinesis别名 ）。也不支持使用加密 AWS 托管式密钥 的源流；请参阅the section called “源流加密”。

监控 AWS KMS 交互

您可以审核客户管理密钥的使用情况 AWS CloudTrail。 AWS KMS 加密和解密交付对象（例如GenerateDataKey和Decrypt ）的加密操作会出现在您的 CloudTrail 事件历史记录中。有关查找 AWS KMS 事件的更多信息，请参阅AWS Key Management Service 开发人员指南中的审计 AWS KMS 密钥使用情况。

源流加密

> **Important**
>
> 如果您的 Kinesis 数据流使用 AWS 托管式密钥 （ aws/kinesis别名 ）加密，则无法创建交付。CreateChannel使用加密的流同步失败。 AWS 托管式密钥

要在加密流上使用流媒体表或 Amazon S3 交付，请使用客户管理的密钥进行流加密。您可以使用更改现有直播的加密密钥StartStreamEncryption。完全支持使用客户管理密钥加密的流。

IAM 访问控制

遵循以下最佳实践，通过数据传输进行 IAM 访问控制：

- 最低权限 -仅授予服务执行角色所需的最低权限。将 Amazon S3 权限范围限定为特定的存储桶和前缀，而不是使用通配符。
- 范围限定到特定资源 -限制交付写入的特定存储桶、表存储桶或表的权限。避免使用 Resource: "*"。
- 防范混乱 — 始终在信任政策中加入aws:SourceArn和aws:SourceAccount条件，以防止其他服务或账户承担执行职责。

AWS CloudTrail 日志记录

所有交付 API 调用均由记录 AWS CloudTrail。这包括`CreateChannel`、`UpdateChannel`、`DeleteChannel`、`DescribeChannel`、和`ListChannels`。您可以使用 AWS CloudTrail 日志来审核谁创建、修改或删除了交付资源，以及这些操作何时发生。

监控数据交付

使用本主题学习如何使用 Amazon CloudWatch 指标、 CloudWatch 日志和监控您的配送 AWS CloudTrail。

CloudWatch 指标

每次交付都会将指标发布到`AWS/Kinesis`命名空间 CloudWatch 中。指标名称以目标类型为前缀：`DeliveryToS3`用于一般用途 Amazon S3 交付和`DeliveryToIceberg`流式传输表传输。使用这些指标来监控配送的运行状况和绩效。

下表列出了一般用途 Amazon S3 交付的指标 (`DeliveryToS3`)。

指标名称	单位	说明
`DeliveryToS3.DataFreshness`	秒	尚未送达目的地的最早记录的年龄。价值上升表明交付落后。
`DeliveryToS3.BytesIn`	字节	从流中读入传输路径的数据量。
`DeliveryToS3.BytesProcessed`	字节	交付处理的数据量。
`DeliveryToS3.BytesOut`	字节	写入目标的数据量。
`DeliveryToS3.RecordCount`	计数	从流中读取的记录总数。
`DeliveryToS3.SuccessfulRecordCount`	计数	成功传送到目的地的记录数。

指标名称	单位	说明
DeliveryToS3.FailedRecordCount	计数	传送失败的记录数。非零值是关键的错误信号。
DeliveryToS3.DeliverySuccess	计数	成功交付操作的数量。
DeliveryToS3.DLQDeliverySuccess	计数	成功写入死信队列的记录数。

下表列出了流式传输表交付的指标 (DeliveryToIceberg)。

指标名称	单位	说明
DeliveryToIceberg.DataFreshness	秒	尚未送到表格的最早记录的年龄。价值上升表明交付落后。
DeliveryToIceberg.BytesIn	字节	从流中读入传输路径的数据量。
DeliveryToIceberg.BytesProcessed	字节	交付处理的数据量。
DeliveryToIceberg.BytesOut	字节	写入表的数据量。
DeliveryToIceberg.TotalRowCount	计数	从流中读取的总行数。
DeliveryToIceberg.SuccessfulRowCount	计数	成功传送到表中的行数。
DeliveryToIceberg.FailedRowCount	计数	传送失败的行数。非零值是关键的错误信号。
DeliveryToIceberg.CommitSuccess	计数	成功提交 Iceberg 的次数。

指标名称	单位	说明
DeliveryToIceberg.DLQDeliverySuccess	计数	成功写入死信队列的记录数。

Dimensions

发布的配送指标采用以下维度：

- ChannelName— 配送名称。
- ChannelId— 交货的唯一标识符。
- StreamName— 源 Kinesis Data Streams 流的名称。

> **Note**
>
> 要在 CloudWatch 警报中定位特定的交付，请指定所有三个维度。要使用GetMetricData和进行浏览或查询ListMetrics，您可以按维度的子集进行筛选，例如ChannelName单独筛选。

建议的警报

下表列出了推荐的配送 CloudWatch 警报。

警报	指标	条件	说明
故障率高	DeliveryToS3.FailedRecordCount	> 0 持续 5 分钟	记录传送失败。调查架构不匹配或目标问题。
数据新鲜度降低	DeliveryToS3.DataFreshness	> 阈值	数据交付落在了后面。调查交付问题，例如缺少服务执行角色的权限或目的地的限制。
未传送任何数据	DeliveryToS3.Succe	= 0 持续 15 分钟	尚未提供任何记录。验证直播是否有数据并且传送处于活动状态。

警报	指标	条件	说明
	ssfulRecordCount		
DLQ 交付失败	DeliveryToS3.DLQDeliverySuccess	大于 0 时 FailedRecordCount 为 0	失败的记录无法写入死信队列。检查 DLQ 存储桶权限。

> **Note**
>
> 上述警报使用了 Amazon S3 交付的DeliveryToS3指标。
> 对于流式表传输，请使用等效的DeliveryToIceberg指标
> （DeliveryToIceberg.FailedRowCountDeliveryToIceberg.DataFreshness、DeliveryTo
> 和DeliveryToIceberg.DLQDeliverySuccess）。

CloudWatch 日志

每次交付都可以将详细日志发布到日 CloudWatch 志，以便进行调试和查看操作。

默认情况下，交付会写入以下日志组和日志流：

- 默认日志组-/aws/kinesis/*channelName*/*channelId*
- 日志流 — DestinationDelivery

通过 CLI 启用 CloudWatch 日志

使用以下命令在创建或更新交付时启用 CloudWatch 日志 AWS CLI：

```
aws kinesis create-channel \
    --channel-name "my-channel" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
    --stream-configuration-list '[
        {
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": { "RecordFormatType": "JSON" }
        }
```

```
    ]' \
    --s3-destination-configuration '{
        "StorageConfiguration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-destination-bucket",
            "ExpectedBucketOwner": "111122223333",
            "CompressionType": "GZIP"
        }
    }' \
    --logging-configuration '{
        "CloudWatchLogs": {
            "Enabled": true
        }
    }'
```

如果在未指定 CloudWatch 日志组的情况下启用 Logs，则交付将使用默认日志组`/aws/kinesis/`*`channelName`*`/`*`channelId`*。如果您指定自定义`LogGroupName`，请确保服务执行角色的“ CloudWatch 日志” 权限仅限于同一个日志组。有关所需的权限，请参阅the section called “用于数据传输的 IAM 权限”。

您还可以使用`update-channel`命令启用或更新现有交付的 CloudWatch 日志，传递相同的`--logging-configuration`值。

所需的权限

要启用 CloudWatch 日志，交付的服务执行角色必须具有以下权限：

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

将这些权限的范围限定为特定的日志组 ARN。the section called “可选 CloudWatch 日志权限”有关完整的策略示例，请参阅。

AWS CloudTrail 日志记录

所有交付 API 调用均由记录 AWS CloudTrail。以下是`CreateChannel`呼叫的示例 AWS CloudTrail 事件：

```
{
    "eventVersion": "1.08",
```

```
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA1EXAMPLE:user-session",
        "arn": "arn:aws:sts::<account-id>:assumed-role/<role-name>/user-session",
        "accountId": "<account-id>"
    },
    "eventTime": "2026-07-20T20:00:00Z",
    "eventSource": "kinesis.amazonaws.com",
    "eventName": "CreateChannel",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.1",
    "userAgent": "aws-cli/2.x",
    "requestParameters": {
        "streamName": "my-stream",
        "channelName": "my-channel"
    },
    "responseElements": {
        "channelARN": "arn:aws:kinesis:us-east-1:<account-id>:channel/<channel-id>"
    },
    "requestID": "a1b2c3d4-example",
    "eventID": "e5f6g7h8-example",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "<account-id>"
}
```

数据交付的最佳实践

以下最佳实践可帮助您充分利用 Amazon Kinesis Data Streams 中的数据交付。有关这些建议中提及的配送指标的完整列表，请参阅the section called “监控数据交付”。

吞吐量和数据新鲜度

调整传送频率并观察传送延迟，以便您的输出文件大小和数据新鲜度与您的工作负载相匹配：

- 根据您的直播吞吐量设置`DataFreshnessInSeconds`参数。较低的值会增加传送频率，但可能会导致输出文件变小。值越高，每个交付周期可以累积更多的数据。
- 根据该`DataFreshness`指标创建 Amazon CloudWatch 警报，以检测交付延迟何时超过可接受的阈值。
- 您可以从同一 Kinesis Data Streams 流创建多个交付。每次交付都是独立运行的，允许您同时将数据传送到多个目的地。

架构管理（Apache Iceberg 上的流式传输表）

由于流式传输表交付绑定到固定架构且不支持架构演变，因此请谨慎管理架构以避免交付失败：

- 创建交付之前，请在 AWS Glue 架构注册表中注册您的架构。交付使用注册的架构来定义目标表结构。
- 交付不支持架构演变。如果您的架构发生变化，则必须删除并使用更新的架构重新创建交付。
- 在将所有记录写入流之前，请使用生产者端架构验证来确保所有记录都符合注册的架构。这样可以最大限度地减少记录传送失败。

S3 对象布局（通用亚马逊 S3 存储桶）

对交付的对象进行布局以提高下游查询性能并控制存储成本：

- 选择与下游使用者的查询模式相匹配的输出密钥模板。例如，如果您的查询经常按时间范围进行筛选，请使用基于日期的前缀。
- 启用 GZIP 或 ZSTD 压缩以降低存储成本并提高下游分析工作负载的读取性能。
- 根据您的数据访问模式选择相应的 Amazon S3 存储类别。对于经常访问的数据，使用 STANDARD，使用 INTELLIGENT_TIERING 让 Amazon S3 在访问层之间自动移动对象，或者使用 GLACIER_IR 来处理仍然需要毫秒检索的很少访问的数据。

安全性

通过最低权限访问、加密和审核来保护您的交付数据和凭证：

- 将您的 IAM 策略范围限定为交付写入的特定存储桶或表。避免使用通配符资源 ARN。
- 在交付使用的 IAM 角色的信任策略中包含`aws:SourceArn`和`aws:SourceAccount`条件密钥。这样可以确保只有您的特定交付人员才能担任该角色，从而防止混乱的副手问题。
- 启用 AWS CloudTrail 日志记录以审核所有交付 API 调用和配置更改。
- 对敏感数据使用客户管理的 AWS KMS 密钥。这使您可以完全控制密钥轮换、访问策略和审计跟踪。

Dead-letter 队列

通过配置和监控死信队列来捕获和诊断无法传送的记录：

- 请务必为配送配置死信队列。无法传送到目的地的记录将发送到死信队列以供日后检查。
- 监控中的DLQDeliverySuccess指标（DeliveryToS3.DLQDeliverySuccess或DeliveryToIceberg.DLQDeliverySuccess）CloudWatch ，以检测何时将记录路由到死信队列。
- 定期检查死信队列条目，以确定交付失败的模式，并在生产者或交付配置级别采取纠正措施。

监控

通过对关键 CloudWatch 指标发出警报并启用交付日志，尽早发现交付问题：

- 针对交付DataFreshness和失败记录指标创建 CloudWatch 警报（DeliveryToS3.FailedRecordCount或DeliveryToIceberg.FailedRowCount），以主动检测交付问题。
- 启用交付 CloudWatch 日志，以捕获详细的交付诊断和错误消息。
- 跟踪已交付字节指标（DeliveryToS3.BytesOut或DeliveryToIceberg.BytesOut），以监控吞吐量并识别数据量的意外下降。

数据传输疑难解答

使用本节来解决数据传输的常见问题。

交付停留在 “创建” 状态

当您创建交付时，它将在配置资源时进入创建状态。配置通常在几分钟内完成。如果交付在 CREATING 中停留很长一段时间或过渡到 FAILED，则可能是配置错误造成的。

DescribeChannel致电查看当前状态和状态原因。常见原因包括：

- IAM 角色 ARN 无效或角色策略中的权限不足。
- 目标 Amazon S3 存储桶不存在或位于其他区域。
- 无法解析架构注册表中的 AWS Glue 架构 ARN。

交付处于失败状态

您无法恢复位ChannelStatus于FAILED（退货方DescribeChannel）的配送。从中读取ChannelStatusReason字段DescribeChannel以确定根本原因。修复潜在问题，删除失败的交付，然后使用更正的配置重新创建。

数据新鲜度高

该DataFreshness指标衡量的是最古老的未交付记录的年龄。较高的值表示交付落后于摄取。常见原因：

- 目标表的分区数过高会增加元数据开销。
- 许多小提交导致的表元数据增长会降低提交吞吐量。
- 低流吞吐量加上低新鲜度设置会导致频繁的小规模交付。

解决方案：对于 Apache Iceberg 上的流媒体表，启用 Amazon S3 表维护（压缩和快照到期）以管理元数据增长。对于低吞吐量流，增加该DataFreshnessInSeconds值以允许在每个交付周期中批处理更多数据。

最严格的数据新鲜度设置需要最低的持续数据流吞吐量，这样每个周期都能积累足够的数据，以实现高效传输和在线压缩。如果您的直播产生的吞吐量低于该吞吐量，请使用更高的DataFreshnessInSeconds值。

失败记录大于零

当失败记录指标为非零时（DeliveryToS3.FailedRecordCount对于 Amazon S3 传输或DeliveryToIceberg.FailedRowCount流表传输），记录将发送到死信队列而不是目的地。

对于 Apache Iceberg 上的直播桌：

- 架构不匹配 — 记录不符合注册的架构。
- 缺少必填字段 — 不可为空的列在记录中没有值。
- 不使用 GSR_JSON 格式的 AWS Glue 架构注册表序列化器，生产者必须使用 AWS Glue 架构注册表生成器库。

对于一般用途的 Amazon S3 存储桶：

- 格式不匹配 — 记录格式与配置的输入格式不匹配。

解决方法：检查死信队列条目以获取详细的错误信息。查看交付 CloudWatch 日志，查看具体的解析或验证错误。修复生产者发送符合要求的记录。

目的地没有显示任何数据

如果交付处于 “活动” 状态，但目的地没有显示任何数据，则最常见的原因是：

- 权限问题 — IAM 角色无法写入目标。检查 CloudWatch 日志中是否`AccessDenied`存在错误。
- 输出密钥前缀不匹配（通用 Amazon S3 存储桶）— 如果您的`s3:PutObject`权限范围限于前缀`arn:aws:s3:::my-bucket/data*`，例如，则输出密钥模板生成的密钥必须以开头。`data/`不匹配会导致每次写入都被拒绝。
- 缺少表权限（流式传输表）— 当您使用客户管理的 AWS KMS 密钥加密目标表时，请验证服务执行角色除其他`s3tables`操作之外还包括`s3tables:PutTableEncryption`在内。没有它，`CreateTable`成功但表加密失败，永远不会创建表，也不会传送任何数据。
- 缺少日志记录权限 — 如果服务执行角色缺少`logs:CreateLogStream`和`logs:PutLogEvents`，则交付失败不会记录在 CloudWatch 日志中，这可能会使权限问题显示为静默故障。当没有日志时，请先验证日志权限。
- 创建后没有新数据——交付不会回填流中的现有数据。只有在交付变为 “激活” 之后写入的记录才会被传送。

配送已暂停

当目的地不可用或不兼容时，配送进入暂停状态。常见原因：

- Apache Iceberg 目标表上的直播表已删除。
- Amazon S3 存储桶拥有者与预期的账户不匹配（所有权不匹配）。
- 在目标表中检测到不兼容的分区列。

暂停配送无法恢复。您必须使用有效的目的地配置创建新的交付。

CloudWatch 日志中出现权限被拒绝错误

`AccessDenied`交付 CloudWatch 日志中的错误表明存在权限问题。常见原因：

- 创建交付后，IAM 角色策略已修改。
- Amazon S3 存储桶策略已更改为拒绝该角色的访问。

- 该角色的信任策略不允许 Kinesis 数据流服务代入该角色。
- 密 AWS KMS 钥策略拒绝对交付角色进行加密或解密访问。

查看并更正相关政策，然后确认配送已恢复。

我的直播无法配送

流媒体表和 Amazon S3 交付要求 Kinesis 数据流处于 On-Demand 标准或 On-Demand 优势容量模式。如果您的直播使用预置模式，则必须先将其切换到 On-Demand 模式，然后才能创建分发。

架构更改后交付停止

交付不支持架构演变。如果您在创建交付后在 S AWS Glue chema Registry 中更新架构，则使用新架构版本生成的记录可能无法通过验证并被路由到死信队列。

解决方法：在生产者处恢复架构更改，或者删除现有交付并使用更新的架构重新创建。

无法删除直播

`ResourceInUseException`当直播中有一个或多个活跃的交付时，`DeleteStream`请求会失败。直播附加交付后，无法将其删除。

解决方法：使用`ListChannels`（使用流过滤器）列出直播中的交付，使用删除每个分`DeleteChannel`发，然后删除该流。

配送配额和限制

下表描述了亚马逊 Kinesis 数据流中流式传输表和 Amazon S3 交付的配额和交付限制。

资源配额

资源	默认	可调整	注意
每个直播的配送量	2	否	一次交付到Apache Iceberg上的流媒体表中，一次交付到通用亚马逊 S3 存储桶。
每次交付的直播次数	1	否	每次交付正好从一个数据流读取。

资源	默认	可调整	注意
每次配送的餐桌数	1	否	每次交付只写入一个目标表。

> **Note**
>
> 一个账户中的交付数量受每个区域每个账户的 On-Demand 直播配额限制，因为每个直播最多可以有两次交付。有关每个账户的直播配额，请参阅。限额和限制

API 限制

限制	默认	注意
API 限制率（每个区域的每个账户）	每秒 5 个事务 (TPS)	适用于控制平面 API 操作。
ListChannels 每页的最大结果数	100	使用分页令牌检索其他结果。

交付限制

以下限制适用于所有交付配置：

- 配送支持仅追加配送。无法在目标位置更新或删除记录。
- 数据传输不支持任何一种目标类型的跨区域交付。Kinesis 数据流和目标必须位于同一区域。
- 直播表（Apache Iceberg）交付不支持跨账户交付。源流、目标 S3 表存储桶和 AWS Glue 架构注册表必须全部位于同一个存储桶中 AWS 账户。
- 对于一般用途的 Amazon S3 交付，只能将目标存储桶放在不同的存储桶中 AWS 账户。该频道及其源流必须在同一个账户中。
- 不对数据进行任何转换。记录按原样从直播传送到目的地。
- 数据交付不支持架构演变。更改架构需要删除并重新创建交付。
- 配送不会回填现有数据。只有在传送达到 ACTIVE 状态后写入直播的记录才会被传送。
- 每次交付都会在目的地创建一个新表。您无法向现有餐桌配送。
- 源 Kinesis 数据流必须处于 On-Demand 标准或 On-Demand 优势容量模式。不支持预置模式。

更新限制

在现有交付中，只能更新以下属性：

- DataFreshnessInSeconds
- LoggingConfiguration

所有其他配置属性在创建后都是不可变的。要更改任何其他设置，必须删除现有交付并使用所需配置创建新交付。

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 CloudWatch 亚马逊发送自定义指标，并对每个直播进行详细监控。
- Kinesis Agent — Kinesis 代理发布自定义 CloudWatch 指标，以帮助评估代理是否按预期运行。
- API 日志记录 – Kinesis Data Streams 使用 AWS 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 价值不断增加？

使用亚马逊监控亚马逊 Kinesis Data Streams 服务 CloudWatch

亚马逊 Kinesis Data Streams 和 CloudWatch Amazon 已集成，因此您可以收集、查看和 CloudWatch 分析 Kinesis 数据流的指标。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。

系统会自动收集您配置的流指标和分片级指标，并将其推送到 CloudWatch 每分钟。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	说明
基本 (流级)	Stream-level 数据每分钟自动发送一次，不收费。

类型	说明
增强（分片级）	Shard-level 数据每分钟发送一次，但需要支付额外费用。要获得此级别的数据，您必须使用 EnableEnhancedMonitoring 操作专门针对该流启用它。 有关定价的信息，请参阅 Amazon CloudWatch 产品页面。

Amazon Kinesis Data Streams 维度和指标

Kinesis Data Streams 将指标发送 CloudWatch 到两个级别：流级别和分片级别（可选）。 Stream-level 指标适用于正常条件下最常见的监控用例。 Shard-level 指标用于特定的监控任务，通常与故障排除有关，使用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

`AWS/Kinesis` 命名空间包括以下流级指标。

Kinesis Data Streams 将这些直播级别的指标发送 CloudWatch 到每分钟。这些指标始终可用。

指标	说明
`GetRecords.Bytes`	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。

指标	说明
	Shard-level 指标名称：OutgoingBytes 尺寸： 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。 Shard-level 指标名称：IteratorAgeMilliseconds 尺寸： 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。 尺寸： StreamName 统计数据：Minimum、Maximum、Average 单位：毫秒

指标	说明
GetRecords.Records	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。 Shard-level 指标名称：OutgoingRecords 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
GetRecords.Success	在指定时段内测量的每个流中的成功 GetRecords 操作数。 尺寸： StreamName 统计数据：Average、Sum、Samples 单位：计数
IncomingBytes	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。 Shard-level 指标名称：IncomingBytes 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节

指标	说明
IncomingRecords	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。 Shard-level 指标名称：IncomingRecords 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
PutRecord.Bytes	在指定时段内使用 PutRecord 操作放入 Kinesis 流的字节数。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节
PutRecord.Latency	在指定时段内测量的每个 PutRecord 操作所用的时间。 尺寸： StreamName 统计数据：Minimum、Maximum、Average 单位：毫秒

指标	说明
PutRecord.Success	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。 尺寸： StreamName 统计数据：Average、Sum、Samples 单位：计数
PutRecords.Bytes	在指定时段内使用 PutRecords 操作放入 Kinesis 流的字节数。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节
PutRecords.Latency	在指定时段内测量的每个 PutRecords 操作所用的时间。 尺寸： StreamName 统计数据：Minimum、Maximum、Average 单位：毫秒
PutRecords.Records	此指标已弃用。使用 PutRecords.SuccessfulRecords 。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

指标	说明
PutRecords.Success	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。 尺寸： StreamName 统计数据：Average、Sum、Samples 单位：计数
PutRecords.TotalRecords	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。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
PutRecords.SuccessfulRecords	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。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
PutRecords.FailedRecords	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。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

指标	说明
PutRecords.ThrottledRecords	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。 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
ReadProvisionedThroughputExceeded	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。 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。 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。 Shard-level 指标名称：ReadProvisionedThroughputExceeded 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
SubscribeToShard.RateExceeded	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。 尺寸： StreamName， ConsumerName

指标	说明
SubscribeToShard.Success	该指标记录 SubscribeToShard订阅是否成功建立。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。 尺寸： StreamName， ConsumerName
SubscribeToShardEvent.Bytes	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。 Shard-level 指标名称：OutgoingBytes 尺寸： StreamName， ConsumerName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节
SubscribeToShardEvent.MillisBehindLatest	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。 尺寸： StreamName， ConsumerName 统计数据：Minimum、Maximum、Average、Samples 单位：毫秒

指标	说明
SubscribeToShardEvent.Records	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。 Shard-level 指标名称：OutgoingRecords 尺寸： StreamName， ConsumerName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
SubscribeToShardEvent.Success	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。 尺寸： StreamName， ConsumerName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

指标	说明
WriteProvisionedThroughputExceeded	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。 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。 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。 Shard-level 指标名称：WriteProvisionedThroughputExceeded 尺寸： StreamName 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

增强的分片级指标

AWS/Kinesis 命名空间包括以下分片级指标。

Kinesis 每分钟向其发送以下分片级别的指标。 CloudWatch 每个指标维度都会创建 1 个 CloudWatch 指标，每月调用大约 43,200 PutMetricData 个 API。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 “亚马逊 CloudWatch自定义指标” 标题下的亚马逊 CloudWatch 定价。按每个月每个指标每个分片收费。

指标	说明
IncomingBytes	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。 Stream-level 指标名称：IncomingBytes

指标	说明
	尺寸： StreamName， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节
IncomingRecords	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。 Stream-level 指标名称：IncomingRecords 尺寸： StreamName， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数
IteratorAgeMilliseconds	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。 Stream-level 指标名称：GetRecords.IteratorAgeMilliseconds 尺寸： StreamName， ShardId 统计数据：Minimum、Maximum、Average、Samples 单位：毫秒

指标	说明
OutgoingBytes	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。 Stream-level 指标名称：GetRecords.Bytes 尺寸： StreamName ， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：字节
OutgoingRecords	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。 Stream-level 指标名称：GetRecords.Records 尺寸： StreamName ， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

指标	说明
ReadProvisionedThroughputExceeded	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。 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。 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。 Stream-level 指标名称：ReadProvisionedThroughputExceeded 尺寸： StreamName， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

指标	说明
WriteProvisionedThroughputExceeded	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。 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。 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。 Stream-level 指标名称：WriteProvisionedThroughputExceeded 尺寸： StreamName， ShardId 统计数据：Minimum、Maximum、Average、Sum、Samples 单位：计数

Amazon Kinesis Data Streams 指标的维度

维度	说明
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。我们建议您在最大值统计数据上使用 CloudWatch 警报，以便在此损失成为风险之前提醒您。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。
PutRecord.Success，PutRecords.Success	我们建议在 Average 统计数据上使用 CloudWatch 警报来指示何时无法进入直播。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	我们建议在 Average 统计数据上使用 CloudWatch 警报来指示数据流中何时出现记录失败。

访问 Kinesis Data Streams 的亚马逊 CloudWatch 指标

您可以使用控制台、命令行或 API 监控 Kinesis Data Streams CloudWatch 的指标。 CloudWatch 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. 打开 CloudWatch 控制台，网址为https://console.aws.amazon.com/cloudwatch/。
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 “按类别划分的CloudWatch 指标” 窗格中，选择 Kinesis 指标。

5. 单击相关行可查看指定MetricName和的统计信息StreamName。

   注意：除读取吞吐量和写入吞吐 CloudWatch 量外，大多数控制台统计信息名称都与前面列出的相应指标名称相匹配。这些统计数据以 5 分钟为间隔计算：写入吞吐量监控`IncomingBytes` CloudWatch 指标，读取吞吐量监视器`GetRecords.Bytes`。

6. （可选）在图表窗格中，选择统计数据和时间段，然后使用这些设置创建 CloudWatch 警报。

要访问指标，请使用 AWS CLI

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。

使用亚马逊监控 Kinesis Data Streams 代理的运行状况 CloudWatch

代理发布命名空间为的自定义 CloudWatch 指标AWS KinesisAgent。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中指定的区域中报告`cloudwatch.endpoint`。 CloudWatch 从多个 Kinesis 代理发布的指标是汇总或合并的。有关代理配置的更多信息，请参阅指定代理配置设置。

使用监视器 CloudWatch

Kinesis Data Streams 代理向发送以下指标 CloudWatch。

指标	说明
`BytesSent`	在指定时段内发送到 Kinesis Data Streams 的字节数。 单位：字节

指标	说明
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

使用记录亚马逊 Kinesis Data Streams API 调用 AWS CloudTrail

Amazon Kinesis Data Streams AWS CloudTrail与一项服务集成，该服务提供用户、角色或 AWS 服务在 Kinesis Data Streams 中采取的操作的记录。 CloudTrail 将 Kinesis Data Streams 的所有 API 调用捕获为事件。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。如果您创建了跟踪，则可以允许将 CloudTrail 事件持续传输到 Amazon S3 存储桶，包括 Kinesis Data Streams 的事件。如果您未配置跟踪，您仍然可以在 CloudTrail 控制台的 “事件历史记录” 中查看最新的事件。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。

要了解更多信息 CloudTrail，包括如何配置和启用它，请参阅AWS CloudTrail 用户指南。

Kinesis Data Streams 将信息流入 CloudTrail

CloudTrail 在您创建 AWS 账户时已在您的账户上启用。当 Kinesis Dat CloudTrail a Streams 中出现支持的事件活动时，该活动会 AWS 与其他服务事件一起记录在事件历史记录中。您可以在自己的 AWS 账户中查看、搜索和下载最近发生的事件。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 AWS 账户中的事件，包括 Kinesis Data Streams 的事件，请创建跟踪。跟踪允许 CloudTrail 将日志文件传输到 Amazon S3 存储桶。默认情况下，当您在控制台中创建跟踪时，该跟踪将应用于所有 AWS 区域。跟踪记录 AWS 分区中所有区域的事件，并将日志文件传送到您指定的

Amazon S3 存储桶。此外，您可以配置其他 AWS 服务，以进一步分析和处理 CloudTrail 日志中收集的事件数据。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
- 配置 Amazon SNS 通知 CloudTrail
- 接收来自多个区域的 CloudTrail 日志文件和接收来自多个账户的 CloudTrail 日志文件

Kinesis Data Streams 支持将以下操作作为事件记录 CloudTrail 在日志文件中：

- 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。身份信息有助于您确定以下内容：

- 请求是使用根证书还是 AWS Identity and Access Management (IAM) 用户凭证发出。
- 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 AWS 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，允许将事件作为日志文件传输到您指定的 Amazon S3 存储桶。 CloudTrail 日志文件包含一个或多个日志条目。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，因此它们不会以任何特定顺序显示。

以下示例显示了一个演示CreateStream、、、DescribeStreamListStreamsDeleteStreamSplitShard、和MergeShards操作的 CloudTrail 日志条目。


```
{
    "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"
        }
    ]
```

```
}
```

使用亚马逊监控 Kinesis 客户端库 CloudWatch

适用于 Amazon K inesis Data Streams 的 Kinesis 客户端库 (KCL) 使用您的 KCL 应用程序的名称作为命名空间，代表您发布自定义的 CloudWatch 亚马逊指标。您可以通过导航到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。

KCL 上传到 CloudWatch 的指标收取象征性费用；具体而言，亚马逊 CloudWatch 自定义指标和亚马逊 CloudWatch API 请求会收取费用。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。当您设置指标报告级别时，个人级别低于报告级别的指标不会发送到 CloudWatch。级别如下：NONE、SUMMARY 和 DETAILED。默认设置为DETAILED；也就是说，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 指标列表。

启用的维度

每个 KCL 指标都有关联的维度，这些维度也会发送到。 CloudWatch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《亚马逊 CloudWatch 用户指南》中 “亚马逊 CloudWatch 概念” 主题中的 “维度” 部分。

启用WorkerIdentifier维度后，如果每次特定 KCL 工作器重新启动时工作人员 ID 属性都使用不同的值，则会将具有新WorkerIdentifier维度值的新指标集发送到。 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以使用实例进行配置，该 KinesisClientLibConfiguration 实例在启动 KCL 应用程序时会传递给 Worker。 MultiLangDaemon 在这种情况下，可以在用于启动 MultiLangDaemon KCL 应用程序的.properties 文件中指定metricsLevel和metricsEnabledDimensions属性。

可向指标级别分配下列三个值之一：NONE、SUMMARY 或 DETAILED。启用的维度值必须是以逗号分隔的字符串，其中包含指标允许使用的维度列表。 CloudWatch KCL 应用程序使用的维度为 Operation、ShardId 和 WorkerIdentifier。

指标列表

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

这些指标汇总了应用程序范围内所有 KCL 工作线程（由 Amazon CloudWatch 命名空间定义）。

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	说明
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。 指标级别：详细 单位：毫秒
TotalLeases	当前 KCL 应用程序的租约总数。 指标级别：汇总 单位：计数
NumWorkers	当前 KCL 应用程序中的工作程序总数。 指标级别：汇总 单位：计数
AssignExpiredOrUnassignedLeases.Time	指在内存中对过期租约进行分配的时间。

指标	说明
	指标级别：详细 单位：毫秒
LeaseSpillover	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。 指标级别：汇总 单位：计数
BalanceWorkerVariance.Time	指在内存中进行工作程序之间租约平衡的时间。 指标级别：详细 单位：毫秒
NumOfLeasesReassignment	指在当前重新分配迭代中重新分配的租约总数。 指标级别：汇总 单位：计数
FailedAssignmentCount	AssignLease 调用 DynamoDB 租用表时失败的次数。 指标级别：详细 单位：计数
ParallelyAssignLeases.Time	指向 DynamoDB 租约表刷新分配的时间。 指标级别：详细 单位：毫秒
ParallelyAssignLeases.Success	指成功刷新分配的次数。 指标级别：详细 单位：计数

指标	说明
TotalStaleWorkerMetricsEntry	指必须清理的工作程序指标条目总数。 指标级别：详细 单位：计数
StaleWorkerMetricsCleanup.Time	指从 DynamoDB 工作程序指标表中删除工作程序指标条目的时间。 指标级别：详细 单位：毫秒
时间	`LeaseAssignmentManager` 操作花费的时间。 指标级别：汇总 单位：毫秒
成功	`LeaseAssignmentManager` 操作已成功完成的次数。 指标级别：汇总 单位：计数
ForceLeaderRelease	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。 指标级别：汇总 单位：计数
NumWorkersWithInvalidEntry	指视为无效的工作程序指标条目的数量。 指标级别：汇总 单位：计数

指标	说明
NumWorkersWithFailingWorkerMetric	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）。 指标级别：汇总 单位：计数
LeaseDeserializationFailureCount	指租约表中未能反序列化的租约条目。 指标级别：汇总 单位：计数

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	说明
KinesisDataFetcher.getIterator.Success	每个 KCL 应用程序的成功 `GetShardIterator` 操作数。 指标级别：详细 单位：计数
KinesisDataFetcher.getIterator.Time	给定 KCL 应用程序的每个 `GetShardIterator` 操作所花费的时间。 指标级别：详细 单位：毫秒
RecordProcessor.initialize.Time	记录处理器的初始化方法所花费的时间。 指标级别：汇总 单位：毫秒
成功	成功的记录处理程序初始化的数目。

指标	说明
	指标级别：汇总 单位：计数
时间	KCL 工作程序初始化记录处理器所花费的时间。 指标级别：汇总 单位：毫秒

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，将调用记录处理程序 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	说明
CreateLease.Success	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。 指标级别：详细 单位：计数
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	说明
	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
时间	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	说明
CreateLease.Success	将新的分片信息添加到 KCL 应用程序 DynamoDB 表的成功尝试次数。 指标级别：详细 单位：计数
CreateLease.Time	在 KCL 应用程序 DynamoDB 表中添加新的分片信息所花费的时间。 指标级别：详细

指标	说明
	单位：毫秒
成功	成功的分片同步操作的数目。 指标级别：汇总 单位：计数
时间	分片同步操作所花费的时间。 指标级别：汇总 单位：毫秒

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	说明
成功	父分片完成的成功检查的数目。 指标级别：汇总 单位：计数
时间	父分片完成所花费的时间。 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。

将 KCL 配置为处理单个数据流（然后将和的值设置 NumStreamsWithPartialLeases 为 1）NumStreamsToSync 以及将 KCL 配置为处理多个数据流时，以下指标可用。

指标	说明
NumStreamsToSync	包含部分租约且必须移交以进行同步的使用者应用程序正在处理的数据流（每个 AWS 账户）的数量。 指标级别：汇总 单位：计数
NumStreamsWithPartialLeases	使用者应用程序正在处理的包含部分租约的数据流（每个 AWS 账户）的数量。 指标级别：汇总 单位：计数
成功	`PeriodicShardSyncManager` 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
时间	`PeriodicShardSyncManager` 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 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	说明
DeletedStreams.Count	在指定时段内删除的数据流数量。 指标级别：汇总 单位：计数
ActiveStreams.Count	正在处理的活动数据流的数量。 指标级别：汇总 单位：计数
StreamsPendingDeletion.Count	依据 `FormerStreamsLeasesDeletionStrategy` 待删除的数据流的数量。 指标级别：汇总 单位：计数

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	说明
InMemoryMetricStatsReporterFailure	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。 指标级别：汇总 单位：计数
WorkerMetricStatsReporter.Time	`WorkerMetricsStats` 操作花费的时间。 指标级别：汇总 单位：毫秒
WorkerMetricStatsReporter.Success	`WorkerMetricsStats` 操作已成功完成的次数。 指标级别：汇总 单位：计数

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	说明
ListLeaseKeysForWorker.Time	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。 指标级别：详细 单位：毫秒
FetchNewLeases.Time	指从租约表中获取所有新租约的时间。 指标级别：详细 单位：毫秒

指标	说明
NewLeasesDiscovered	指分配给工作程序的新租约总数。 指标级别：详细 单位：计数
时间	`LeaseDiscovery` 操作花费的时间。 指标级别：汇总 单位：毫秒
成功	`LeaseDiscovery` 操作已成功完成的次数。 指标级别：汇总 单位：计数
OwnerMismatch	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。 指标级别：详细 单位：计数

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	说明
RenewLease.Success	工作程序成功续订租约的数目。 指标级别：详细 单位：计数
RenewLease.Time	租约续订操作所花费的时间。 指标级别：详细

指标	说明
	单位：毫秒
CurrentLeases	续订所有租约后由工作程序拥有的分片租约数。 指标级别：汇总 单位：计数
LostLeases	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。 指标级别：汇总 单位：计数
成功	工作程序的成功的租约续订操作次数。 指标级别：汇总 单位：计数
时间	续订工作程序的所有租约所花费的时间。 指标级别：汇总 单位：毫秒

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	说明
ListLeases.Success	成功从 KCL 应用程序 DynamoDB 表中检索所有分片租约的次数。 指标级别：详细 单位：计数
ListLeases.Time	从 KCL 应用程序 DynamoDB 表中检索所有分片租约所花费的时间。

指标	说明
	指标级别：详细 单位：毫秒
TakeLease.Success	工作程序成功从其他 KCL 工作程序中提取分片租约的次数。 指标级别：详细 单位：计数
TakeLease.Time	利用该工作程序提取的租约更新租约表所花费的时间。 指标级别：详细 单位：毫秒
NumWorkers	工作程序总数，由特定工作程序标识。 指标级别：汇总 单位：计数
NeededLeases	当前工作程序为平衡分片处理负载所需的分片租约数。 指标级别：详细 单位：计数
LeasesToTake	工作程序将尝试提取的租约的数目。 指标级别：详细 单位：计数
TakenLeases	工作程序已成功提取的租约的数目。 指标级别：汇总 单位：计数

指标	说明
TotalLeases	KCL 应用程序正在处理的分片的总数。 指标级别：详细 单位：计数
ExpiredLeases	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。 指标级别：汇总 单位：计数
成功	`TakeLeases` 操作已成功完成的次数。 指标级别：汇总 单位：计数
时间	工作程序的 `TakeLeases` 操作所花费的时间。 指标级别：汇总 单位：毫秒

Per-shard 指标

这些指标跨单个记录处理程序聚合一起。

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	说明
KinesisDataFetcher .getRecords.Success	每个 Kinesis 数据流分片的成功 `GetRecords` 操作数。 指标级别：详细

指标	说明
	单位：计数
KinesisDataFetcher.getRecords.Time	Kinesis 数据流分片的每个 `GetRecords` 操作所花费的时间。 指标级别：详细 单位：毫秒
UpdateLease.Success	给定分片的记录处理程序完成的成功检查点的数目。 指标级别：详细 单位：计数
UpdateLease.Time	给定分片的每个检查点操作所花费的时间。 指标级别：详细 单位：毫秒
DataBytesProcessed	每个 `ProcessTask` 调用所处理的记录的总大小（以字节为单位）。 指标级别：汇总 单位：字节
RecordsProcessed	每个 `ProcessTask` 调用所处理的记录的数量。 指标级别：汇总 单位：计数
ExpiredIterator	呼叫时 ExpiredIteratorException 收到的数量`GetRecords` 。 指标级别：汇总 单位：计数

指标	说明
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.processRecords.Time	记录处理器的 `processRecords` 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
时间	处理任务操作所花费的时间。 指标级别：汇总 单位：毫秒

使用亚马逊监控 Kinesis 制作人库 CloudWatch

亚马逊 Kinesis Data Streams 的亚马逊 Kinesis 制作者库 (KPL) 代表你发布自定义的 CloudWatch 亚马逊指标。您可以通过导航到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。

KPL 上传到 CloudWatch 的指标收取象征性费用；具体而言，亚马逊 CloudWatch 自定义指标和亚马逊 CloudWatch API 请求会收取费用。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。如果您想在指标中包含更精细的数据，这很有用。CloudWatch 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已上传到 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。当您设置级别时，级别低于该级别的指标不会发送到 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，对于一个特定的指标，两个各有两个分片的流将生成七个 CloudWatch 指标：每个分片一个，每个流一个，总体一个；所有指标描述的统计数据相同，但粒度级别不同。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：

```
MetricName (GLOBAL):           Metric X                    Metric Y
                                  |                           |
                          -----------------            ------------
                          |               |            |          |
```

```
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 在本地计算每个指标的总和、平均值、最小值、最大值和计数，如所示。 CloudWatch

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。

所有指标均可上传至 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。因此，KPL 会在本地将指标聚合到 1 分钟存储桶中，并根据启用的指标将统计数据对象上传到每分钟 CloudWatch 一次。

指标列表

指标	说明
`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	说明
UserRecordsPending	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。 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。 指标级别：详细 单位：个
UserRecordsPut	已成功放置的逻辑用户记录的计数。 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。 指标级别：汇总 单位：个
UserRecordsDataPut	逻辑用户记录成功放置的字节数。 指标级别：详细 单位：字节
KinesisRecordsPut	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）。 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。 指标级别：汇总 单位：个
KinesisRecordsDataPut	Kinesis Data Streams 记录中的字节数。 指标级别：详细 单位：字节

指标	说明
ErrorsByCode	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。 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。 指标级别：汇总 单位：个
AllErrors	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。 指标级别：汇总 单位：个
RetriesPerRecord	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。 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。 指标级别：详细 单位：个
BufferingTime	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。 指标级别：汇总 单位：毫秒

指标	说明
Request Time	执行 PutRecordsRequests 所花费的时间。 指标级别：详细 单位：毫秒
User Records per Kinesis Record	已聚合到单条 Kinesis Data Streams 记录中的逻辑用户记录的数目。 指标级别：详细 单位：个
Amazon Kinesis Records per PutRecordsRequest	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。 指标级别：详细 单位：个
User Records per PutRecordsRequest	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。 指标级别：详细 单位：个

Amazon Kinesis Data Streams 中的安全性

云安全 AWS 是重中之重。作为 AWS 客户，您可以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 AWS 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 — AWS 负责保护在 AWS 云中运行 AWS 服务的基础架构。 AWS 还为您提供可以安全使用的服务。 Third-party 作为AWS 合规计划的一部分，审计师定期测试和验证我们安全的有效性。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AWS 服务。
- 云端安全-您的责任由您使用的 AWS 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 AWS 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
- 通过试运行测试您的权限并请求输入
- Amazon Kinesis Data Streams 的合规性验证
- Amazon Kinesis Data Streams 中的故障恢复能力
- 亚马逊 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

Server-side 使用 AWS Key Management Service (AWS KMS) 密钥进行加密，通过对 Amazon Kinesis Data Streams 中的静态数据进行加密，您可以轻松满足严格的数据管理要求。

> **Note**
>
> 如果您在 AWS 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？

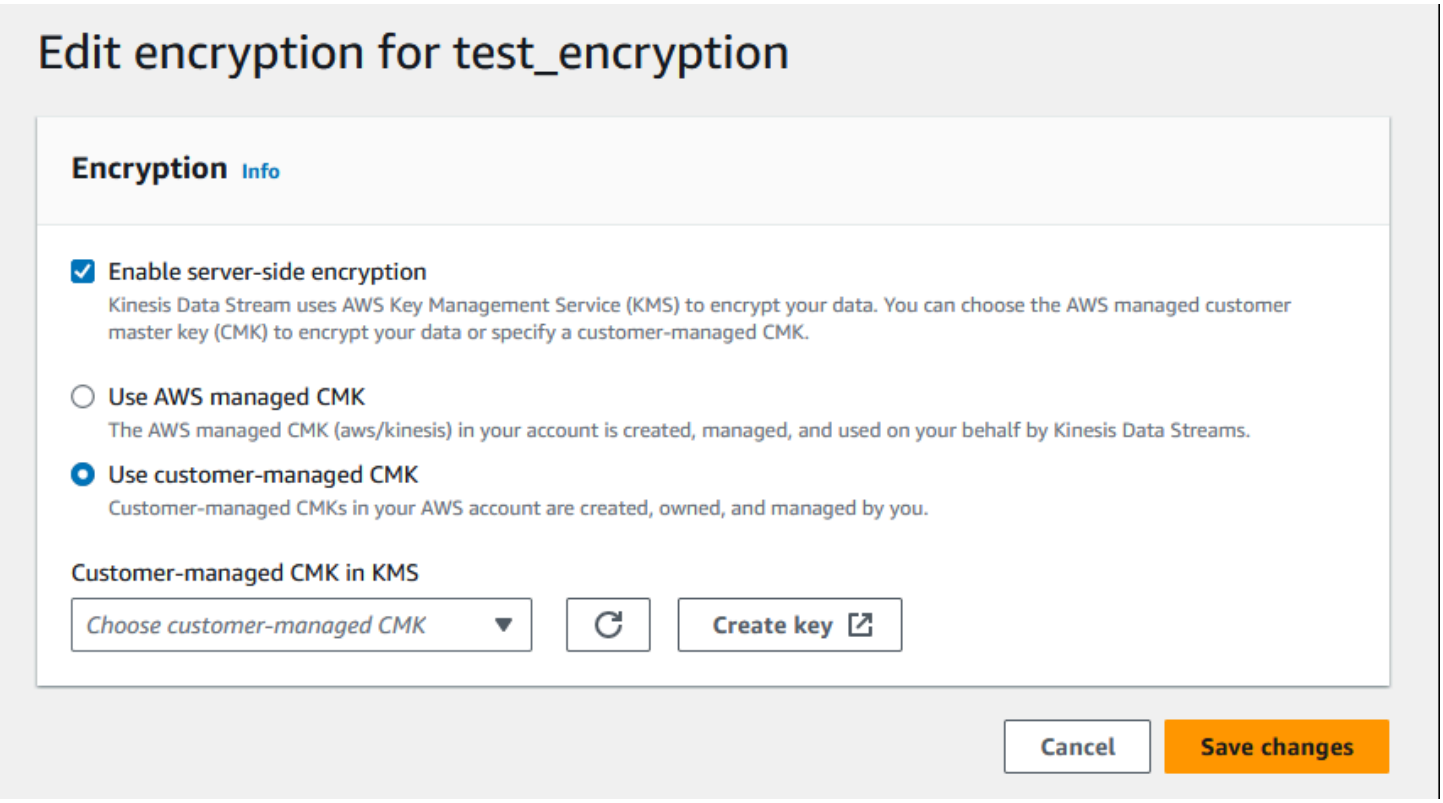
Server-side 加密是 Amazon Kinesis Data Streams 中的 AWS KMS 一项功能，它使用您指定的客户主密钥 (CMK) 在数据处于静止状态之前自动对其进行加密。数据在写入 Kinesis 流存储层之前加密，并在从存储检索到之后进行解密。因此，在 Kinesis Data Streams 服务中对数据进行静态加密。这样，您就可以满足严格的监管要求并增强您数据的安全性。

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。 AWS KMS 提供了服务器端加密功能使用的所有主密钥。 AWS KMS 便于使用 AWS由管理的 Kinesis 的 CMK、 AWS KMS 用户指定的 CMK 或导入到服务中的主密钥。 AWS KMS

> **Note**
>
> Server-side 加密仅在启用加密后才会对传入的数据进行加密。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。

在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。 AWS 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 AWS 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：



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

在应用服务器端加密时，您需要支付 AWS 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 AWS KMS 。在 30 天内，由 Kinesis 直播发起的 AWS 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唯一的 API 调用。 AWS 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

- AWS 由（别名 `aws/kinesis` = ）管理的 Kinesis 用户主密钥是免费的。
- User-generated KMS 密钥需支付 KMS 密钥费用。有关更多信息，请参阅 AWS 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 AWS KMS 进行唯一的 API 调用。当您使用 IAM 角色进行身份验证时，每次 assume-role 调用都将生成唯一的用户凭证，您可能需要缓存 assume-role 调用返回的用户凭证以节省 KMS 费用。

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 AWS 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 AWS KMS 服务呼叫量。生成新数据加密密钥的 API 请求需支付 AWS KMS 使用费用。有关更多信息，请参阅 AWS Key Management Service Pricing: Usage。

按区域的服务器端加密的可用性

目前，Kinesis 流的服务器端加密在 Kinesis Data Streams 支持的所有区域（US-West包括 () ）和中国区域中都可用。 AWS GovCloud 有关 Kinesis Data Stream https://docs.aws.amazon.com/general/latest/gr/ak.htmls 支持的区域的更多信息，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 AWS 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 AWS 管理控制台 并打开 Amazon Kinesis Data Streams 控制台。
2. 在 AWS 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 加密中，选择编辑。
5. 除非要使用用户生成的 KMS 主密钥，否则请确保选择（默认） aws/kinesisKMS 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 AWS 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Server-side 加密中选择禁用 AWS 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《AWS 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 AWS 账户或其他 AWS 账户中使用自定义 KMS 密钥。在 中的 KMS Master Key AWS 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 AWS 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 AWS 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 AWS KMS 权限的示例和更多信息，请参阅 AWS KMS 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 AWS KMS 配合使用。

Kinesis Data Streams 加密上下文

当 Amazon Kinesis Data Streams 代表您 AWS KMS 调用时，它会将加密上下文传递 AWS KMS 给该上下文，该上下文可用作密钥策略和授权中的授权条件。Kinesis Data Streams 在所有调用中都使用流 ARN 作为加密上下文。 AWS KMS

```
"encryptionContext": {
    "aws:kinesis:arn": "arn:aws:kinesis:region:account-id:stream/stream-name"
}
```

您可以使用加密上下文来识别审核记录和日志中您的 KMS 密钥的使用情况。它还以纯文本形式出现在日志中，例如。 AWS CloudTrail

要将您的 KMS 密钥的使用限制为从 Kinesis Data Streams 发出的针对特定流的请求，请使用 `kms:EncryptionContext:aws:kinesis:arn` KMS 密钥策略或 IAM 策略中的条件密钥。

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 AWS 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`调用的成功情况：

- `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 Direct Connect 连接。接口 VPC 终端节点由一项 AWS 技术提供支持 AWS PrivateLink，该技术允许使用弹性网络接口在 AWS 服务之间进行私密通信，并使用您的 Amazon VPC 中的私有 IP。有关更多信息，请参阅亚马逊 Virtual Private Cloud 和接口 VPC 终端节点 (AWS PrivateLink)。

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。为您的 Kinesis Data Streams 创建接口 VPC 终端节点，以启动通过接口 VPC 终端节点流入您的亚马逊 VPC 资源的流量。 FIPS-enabled 接口 VPC 终端节点适用于美国区域。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 AWS 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。

以下是用于访问 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）
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)
- 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

AWS Identity and Access Management (IAM) 使您能够执行以下操作：

- 在您的 AWS 账户下创建用户和群组
- 为您 AWS 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 AWS 资源执行任务的权限
- 允许其他 AWS 账户中的用户共享您的 AWS 资源
- 为您的 AWS 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 AWS 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 AWS 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：

- AWS 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 用户对您 AWS 账户的访问权限。

内容

- 策略语法
- Kinesis Data Streams 的操作
- Kinesis Data Streams 的 Amazon 资源名称（ARN）
- Kinesis Data Streams 的示例策略
- 与其他账户共享您的数据流
- 配置一个 AWS Lambda 使用另一个账户从 Kinesis Data Streams 读取的函数
- 使用基于资源的策略共享访问权限
- 流式传输表和 Amazon S3 交付的访问控制

策略语法

IAM 策略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 策略语句中，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 Amazon 资源名称（ARN）

每个 IAM 策略语句适用于您使用资源的 ARN 指定的资源。

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

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-1:111122223333: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:*:111122223333: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 AWS 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。Resource-based 策略是您附加到资源（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 AWS 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它账户中的 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 AWS 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 AWS 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 AWS API 操作同名。

可以共享的 Kinesis Data Streams 操作：

处理建议	访问级别
DescribeStreamConsumer	使用者
DescribeStreamSummary	数据流
GetRecords	数据流
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: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::111122223333: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_efo_read_policy_ID",
    "Statement": [
        {
            "Sid": "eforeadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamConsumer",
                "kinesis:SubscribeToShard"
            ],
            "Resource": "arn:aws:kinesis:us-east-2:123456789012:stream/
datastreamABC/consumer/consumerDEF:1674696300"
        }
    ]
}
```

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的策略

除此之外 AWS 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

PutResourePolicy 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：
  - Identity-based 附加到关联 AWS 委托人的策略（例如，IAM 角色）
  - Resource-based 关联到关联 AWS 资源的策略（例如， AWS Key Management Service KMS 密钥）
- AWS 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。
- 不支持规范用户 ID。
- 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 AWS 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。

配置一个 AWS Lambda 使用另一个账户从 Kinesis Data Streams 读取的函数

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 与跨账户共享访问权限 AWS Lambda 函数。

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。

与跨账户共享访问权限 AWS Lambda 函数

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 AWS 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AWS Lambda 控制台中，创建一个 AWS 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 或向 Kinesis 客户端库提供直播 StreamTrackerARN。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（如增强版）那里读取 Fan-Out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 AWS 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。

流式传输表和 Amazon S3 交付的访问控制

流式传输表和 Amazon S3 交付使用 Amazon Kinesis Data Streams 假设的 IAM 服务执行角色将数据写入您的目标。请遵循以下访问控制实践：

- 最低权限-仅授予服务执行角色所需的最低权限。将 Amazon S3 权限的范围限定为特定的存储桶和前缀，而不是使用通配符。
- 特定资源的范围-限制对交付写入的特定存储桶、表存储桶或表的权限。避免使用 `Resource: "*"`。

- 混乱的副手预防 — 在服务执行角色信任策略中包含aws:SourceArn和aws:SourceAccount条件，以便只有您的交付资源才能担任该角色。
- 源流加密-如果您的直播使用 AWS 托管式密钥 （ aws/kinesis别名 ）加密，则无法创建分发。改为使用客户管理的密钥进行直播加密。

有关完整的服务执行角色策略，请参阅the section called “用于数据传输的 IAM 权限”。有关加密的详细信息和审核日志，请参阅the section called “数据传输的安全性”。

通过试运行测试您的权限并请求输入

您可以将该DryRun参数与 Kinesis Data Streams 数据平面 API 配合使用，以验证您是否拥有所需的权限以及您的请求参数是否有效。此验证是在您对 Amazon Kinesis Data Streams 资源运行操作之前进行的。当您设置为 DryRun true 时，Kinesis Data Streams 会验证您是否具有访问直播所需的 IAM 权限，检查您的请求参数，并确认目标资源是否存在。它在不运行操作或读取或写入任何数据的情况下执行这些检查。

如果所有检查都通过，API 将返回带有 a 的 HTTP 400DryRunOperationException，确认如果没有该DryRun参数，请求就会成功。如果任何检查失败，API 将返回与实际操作返回的错误相同的错误（例如AccessDeniedException）。除其他外，您可能会收到以下例外情况：

- DryRunOperationException— 如果没有DryRun参数，请求就会成功。
- AccessDeniedException— 调用者没有 API 操作所需的 IAM 权限。
- ResourceNotFoundException— 指定的直播或消费者不存在。
- ValidationException— 一个或多个请求参数无效（例如，缺少必填字段、ARN 格式错误）。
- InvalidArgumentException— 参数值超出范围或不支持（例如，无效 ShardIteratorType，future 时间戳）。

DryRun启用该功能的请求将受到每个流每秒 1 笔交易 (TPS) 的专用限制限制，这与流的正常每分片吞吐量限制不同。该限制在同一个直播中所有支持的试运行 API（PutRecordPutRecords、GetRecords、GetShardIterator、和SubscribeToShard）之间共享。如果您超过此限制，API 将返回ThrottlingException。

使用该DryRun参数不收取任何额外费用。DryRun已启用的请求的计费方式与DryRun禁用的等效请求的计费方式相同。例如，对于PutRecord和PutRecords，将根据预配置模式下的输入有效载荷大小向您收费。直播和分片时长的所有费用将继续适用。有关定价的更多信息，请参阅 Amazon Kinesis Data Streams 定价。

> **Note**
>
> 该DryRun参数用于验证 Kinesis Data Streams API 操作的 IAM 权限。它不验证加密流的 AWS Key Management Service 密钥权限。如果您的直播使用带有客户托管密 AWS KMS 钥的服务器端加密，则必须单独验证您的制作人是否拥有该密钥的kms:GenerateDataKey权限，而您的消费者是否kms:Decrypt有权使用该 AWS KMS 密钥。要验证 AWS KMS 权限，请参阅《 AWS Key Management Service 开发者指南》中的测试您的权限。

支持的 API

以下 API 支持该DryRun参数：

- PutRecord
- PutRecords
- GetRecords
- GetShardIterator
- SubscribeToShard

AWS CloudTrail 日志记录

如果您在中启用数据事件 AWS CloudTrail，则会 AWS CloudTrail 记录使用DryRun参数设置为 true 的数据平面 API 调用。相应 CloudTrail 的事件在请求中包含DryRun参数，如果验证成功，则会DryRunOperationException显示 ErrorCode 字段。

使用 dry run 和 AWS CLI

在 AWS CLI 命令中指定--dry-run标志。

以下示例测试 PutRecord 权限：

```
aws kinesis put-record \
    --stream-name <your-stream-name> \
    --data <your-data-payload> \
    --partition-key <your-partition-key> \
    --dry-run
```

如果请求成功，则响应将包含以下消息：`DryRunOperation validation succeeded while calling PutRecord operation.: Request would have succeeded, but DryRun flag is set.`

使用 dry run 和 AWS 软件开发工具包

Python (Boto3)

```
import boto3
from botocore.exceptions import ClientError

kinesis = boto3.client('kinesis')

try:
    kinesis.put_record(
        StreamName='my-stream',
        Data=b'test-data',
        PartitionKey='my-partition-key',
        DryRun=True
    )
except ClientError as e:
    if e.response['Error']['Code'] == 'DryRunOperationException':
        print('SUCCESS: Request would have succeeded')
    else:
        print(f'FAILURE: {e.response["Error"]["Code"]}')
```

Java (AWS SDK for Java 2.x)

```
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-stream")
    .data(SdkBytes.fromUtf8String("test-data"))
    .partitionKey("my-partition-key")
    .dryRun(true)
    .build();

try {
    kinesisClient.putRecord(request);
} catch (DryRunOperationException e) {
    System.out.println("SUCCESS: " + e.getMessage());
} catch (AccessDeniedException e) {
    System.out.println("FAILURE: " + e.getMessage());
}
```

Amazon Kinesis Data Streams 的合规性验证

我们全新的 AWS 注册体验不是为受监管的工作负载而设计的。如果您正在使用我们的全新 AWS 注册体验，但又想 AWS 用于受监管的工作负载，则可以注册 AWS（高级）或激活适合您 AWS 环境的高级功能。

要了解是否属于特定合规计划的范围，请参阅AWS 服务 “按合规计划划分的范围” ”，然后选择您感兴趣的合规计划。 AWS 服务 有关一般信息，请参阅AWS 合规计划AWS。

您可以使用下载第三方审计报告 AWS Artifact。有关更多信息，请参阅中的 “下载报告” 中的 “ AWS Artifact。

您在使用 AWS 服务 时的合规责任取决于您的数据的敏感性、贵公司的合规目标以及适用的法律和法规。有关您在使用时的合规责任的更多信息 AWS 服务，请参阅AWS 安全文档。

Amazon Kinesis Data Streams 中的故障恢复能力

AWS 全球基础设施是围绕 AWS 区域和可用区构建的。 AWS 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 AWS 区域和可用区的更多信息，请参阅AWS 全球基础设施。

除了 AWS 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
- 托管该应用程序的一个或多个实例的 EC2 实例可能失败

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。

在工作程序或应用程序失败、未检测到失败且应用程序有其他实例正在其他 EC2 实例上运行的情况下，这些实例上的工作程序会处理该失败。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 实例上的负载也会相应地增加。

此处描述的情形假定，尽管工作程序或应用程序已失败，但托管 EC2 实例仍在运行并因此不由自动扩缩组重新启动。

Amazon EC2 实例故障

我们建议您在自动扩缩组中为您的应用程序运行 EC2 实例。由此一来，如果其中一个 EC2 实例失败，自动扩缩组会自动启动新的实例进行代替。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。

亚马逊 Kinesis Data Streams 中的基础设施安全

作为一项托管服务，受 AWS 全球网络安全的保护。有关 AWS 安全服务以及如何 AWS 保护基础设施的信息，请参阅AWS 云安全。要使用基础设施安全的最佳实践来设计您的 AWS 环境，请参阅 S AWS ecurity Pillar Well-Architected Fram ework 中的基础设施保护。

您可以使用 AWS 已发布的 API 调用通过网络进行访问。客户端必须支持以下内容：

- 传输层安全性协议（TLS）。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密 (PFS) 的密码套件，例如 DHE（短暂的）或 ECDHE（椭圆曲线短暂的 Diffie-Hellman）。 Diffie-Hellman大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

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 AWS 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 AWS CloudTrail与一项服务集成，该服务提供用户、角色或 AWS 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 “使用记录亚马逊 Kinesis Data Streams API 调用 AWS CloudTrail”。

将此服务与 AWS SDK

AWS 软件开发套件 (SDK) 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	代码示例
适用于 C++ 的 AWS SDK	适用于 C++ 的 AWS SDK 代码示例
AWS CLI	AWS CLI 代码示例
适用于 Go 的 AWS SDK	适用于 Go 的 AWS SDK 代码示例
适用于 Java 的 AWS SDK	适用于 Java 的 AWS SDK 代码示例
适用于 JavaScript 的 AWS SDK	适用于 JavaScript 的 AWS SDK 代码示例
适用于 Kotlin 的 AWS SDK	适用于 Kotlin 的 AWS SDK 代码示例
适用于 .NET 的 AWS SDK	适用于 .NET 的 AWS SDK 代码示例
适用于 PHP 的 AWS SDK	适用于 PHP 的 AWS SDK 代码示例
AWS Tools for PowerShell	AWS Tools for PowerShell 代码示例
AWS 适用于 Python 的 SDK v4	AWS 适用于 Python 的 SDK v4 代码示例
适用于 Python (Boto3) 的 AWS SDK	适用于 Python (Boto3) 的 AWS SDK 代码示例
适用于 Ruby 的 AWS SDK	适用于 Ruby 的 AWS SDK 代码示例
适用于 Rust 的 AWS SDK	适用于 Rust 的 AWS SDK 代码示例
适用于 SAP ABAP 的 AWS SDK	适用于 SAP ABAP 的 AWS SDK 代码示例
适用于 Swift 的 AWS SDK	适用于 Swift 的 AWS SDK 代码示例

示例可用性

找不到所需的内容？ 通过使用此页面底部的提供反馈链接请求代码示例。

使用 Kinesis 的代码示例 AWS SDKs

以下代码示例展示了如何将 Kinesis 与 AWS 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 AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS SDKs
  - 使用 SDK 学习 Kinesis 的基础知识 AWS
  - 使用 Kinesis 的操作 AWS SDKs
    - AddTagsToStream与 AWS SDK 或 CLI 配合使用
    - CreateStream与 AWS SDK 或 CLI 配合使用
    - DeleteStream与 AWS SDK 或 CLI 配合使用
    - DeregisterStreamConsumer与 AWS SDK 或 CLI 配合使用
    - DescribeStream与 AWS SDK 或 CLI 配合使用
    - GetRecords与 AWS SDK 或 CLI 配合使用
    - 将 GetShardIterator 与 CLI 配合使用
    - 与 AWS SDK ListStreamConsumers 配合使用
    - ListStreams与 AWS SDK 或 CLI 配合使用
    - ListTagsForStream与 AWS SDK 或 CLI 配合使用
    - PutRecord与 AWS SDK 或 CLI 配合使用
    - PutRecords与 AWS SDK 或 CLI 配合使用
    - RegisterStreamConsumer与 AWS 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 AWS SDKs

以下代码示例展示了如何使用 Amazon Kinesis 的基础知识。 AWS SDKs

示例

- 使用 SDK 学习 Kinesis 的基础知识 AWS
- 使用 Kinesis 的操作 AWS SDKs
  - AddTagsToStream与 AWS SDK 或 CLI 配合使用
  - CreateStream与 AWS SDK 或 CLI 配合使用
  - DeleteStream与 AWS SDK 或 CLI 配合使用
  - DeregisterStreamConsumer与 AWS SDK 或 CLI 配合使用
  - DescribeStream与 AWS SDK 或 CLI 配合使用
  - GetRecords与 AWS SDK 或 CLI 配合使用
  - 将 GetShardIterator 与 CLI 配合使用
  - 与 AWS SDK ListStreamConsumers 配合使用
  - ListStreams与 AWS SDK 或 CLI 配合使用
  - ListTagsForStream与 AWS SDK 或 CLI 配合使用
  - PutRecord与 AWS SDK 或 CLI 配合使用
  - PutRecords与 AWS SDK 或 CLI 配合使用
  - RegisterStreamConsumer与 AWS SDK 或 CLI 配合使用

使用 SDK 学习 Kinesis 的基础知识 AWS

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 AWS 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_knsputrecordoutput.

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

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

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

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

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

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

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

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

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

- 有关 API 详细信息，请参阅《AWS SDK for SAP ABAP API Reference》中的以下主题。
  - CreateStream
  - DeleteStream
  - GetRecords
  - GetShardIterator
  - PutRecord

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS SDKs

以下代码示例演示了如何使用执行各个 Kinesis 操作。 AWS 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 AWS SDK 或 CLI 配合使用
- CreateStream与 AWS SDK 或 CLI 配合使用
- DeleteStream与 AWS SDK 或 CLI 配合使用
- DeregisterStreamConsumer与 AWS SDK 或 CLI 配合使用
- DescribeStream与 AWS SDK 或 CLI 配合使用
- GetRecords与 AWS SDK 或 CLI 配合使用
- 将 GetShardIterator 与 CLI 配合使用
- 与 AWS SDK ListStreamConsumers 配合使用
- ListStreams与 AWS SDK 或 CLI 配合使用

- ListTagsForStream与 AWS SDK 或 CLI 配合使用
- PutRecord与 AWS SDK 或 CLI 配合使用
- PutRecords与 AWS SDK 或 CLI 配合使用
- RegisterStreamConsumer与 AWS SDK 或 CLI 配合使用

`AddTagsToStream`与 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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($"Taggs successfully added to {streamName}.");
            }
            else
            {
                Console.WriteLine("Tags were not added to the stream.");
            }
        }

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

            var response = await client.AddTagsToStreamAsync(request);

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

- 有关 API 的详细信息，请参阅 适用于 .NET 的 AWS SDK API 参考AddTagsToStream中的。

CLI

AWS 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AWS CLI 命令参考AddTagsToStream中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考CreateStream中的。

CLI

AWS 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AWS 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 AWS 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 AWS 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 AWS 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 AWS Tools for PowerShell Cmdlet 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 AWS 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CreateStream于 Python 的AWS SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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CreateStream于 Rust 的AWS SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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->createstream(
           iv_streamname = iv_stream_name
           iv_shardcount = iv_shard_count ).
       MESSAGE 'Stream created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
     CATCH /aws1/cx_knsresourceinuseex.
       MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
   ENDTRY.
```

- 有关 API 的详细信息，请参阅适用CreateStream于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考DeleteStream中的。

CLI

AWS 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AWS 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 AWS 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 AWS 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 AWS 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 AWS 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 AWS 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DeleteStream于 Python 的AWS SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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DeleteStream于 Rust 的AWS SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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DeleteStream于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考DeregisterStreamConsumer中的。

CLI

AWS 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AWS CLI 命令参考DeregisterStreamConsumer中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 DescribeStream。

CLI

AWS 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":
 "496008716830016379327620861724741448737963573031406141 78"
                }
            }
        ],
        "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AWS 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 AWS Tools for PowerShell Cmdlet 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 AWS Tools for PowerShell Cmdlet 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 AWS 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DescribeStream于 Python 的AWS SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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DescribeStream于 Rust 的AWS SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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DescribeStream于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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

AWS 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 AAAAAAAAAAF7/0mWD7IuHj1yGv/
TKuNgx2ukD5xipCY4cy4gU96orWwZwcSXh3K9tAmGYeOZyLZrvzzeOFVf9iN99hUPw/w/
b0YWYeehfNvnf1DYt5XpDJghLKr3DzgznkTmMymDP3R+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。 AWS

- 有关 API 的详细信息，请参阅AWS 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 AWS 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:
```

```
                    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 AWS 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 "AAAAAAAAAAGIc....9VnbiRNaP"
```

输出：

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
0                  AAAAAAAAAAERNIq...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 AWS Tools for PowerShell Cmdlet 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 "AAAAAAAAAAGIc....9VnbiRNaP"
```

输出：

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
```

```
0                 AAAAAAAAAAERNIq...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 AWS 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 AWS 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GetRecords于 Python 的AWS 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 AWS 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_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GetRecords于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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

AWS 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/kebq1TGYI7Vy
+lUkA9BuN3xvATxMBGxRY3zYK05gqgvaIRn94O8SqeEqwhigwZxNWxID3Ej7YYYcxQi8Q/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。 AWS

- 有关 API 的详细信息，请参阅AWS 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：

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- 有关 API 的详细信息，请参阅 AWS Tools for PowerShell Cmdlet 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：

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- 有关 API 的详细信息，请参阅 AWS Tools for PowerShell Cmdlet 参考 (V 5) GetShardIterator中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考ListStreamConsumers中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考ListStreams中的。

CLI

AWS 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AWS 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 AWS 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AWS SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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_knslimitexceededex.
       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ListStreams于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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>
    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 AWS SDK API 参考ListTagsForStream中的。

CLI

AWS 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 samplestream
```

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AWS CLI 命令参考ListTagsForStream中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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

AWS 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。 AWS

- 有关 API 的详细信息，请参阅AWS 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 AWS 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 AWS 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 AWS 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 AWS Tools for PowerShell Cmdlet 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 AWS 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PutRecord于 Python 的AWS SDK (Boto3) API 参考。

Rust

适用于 Rust 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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PutRecord于 Rust 的AWS SDK API 参考。

SAP ABAP

适用于 SAP ABAP 的 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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PutRecord于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS SDK 或 CLI 配合使用

以下代码示例演示如何使用 PutRecords。

CLI

AWS 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 开发工具包开发生产者。 AWS

- 有关 API 的详细信息，请参阅AWS 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 AWS 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 AWS SDK API 参考PutRecords中的。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 AWS 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 SDK

> **Note**
>
> 还有更多相关信息 GitHub。在 AWS 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 AWS SDK API 参考[RegisterStreamConsumer](#)中的。

CLI

AWS 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AWS 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 AWS 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RegisterStreamConsumer于 S AP 的AWS SDK ABAP API 参考。

有关 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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。 AWS 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 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}"
```

```
      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 AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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 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.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
```

```
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
```

```
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

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 S AWS DK 开发者指南和代码示例的完整列表，请参阅将此服务与 AWS SDK。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。

变更	说明	日期
增加了对从 KCL 3.x 第 1 阶段回滚到 KCL 2.x 的支持	添加了从以前的 KCL 版本迁移，涵盖了`rollback-to-v2` 模式和`useAbsoluteReBalanceThreshold` 配置。	2026 年 9 月 25 日
添加了服务管理的记录分发	添加了Service-managed记录分发。	2026年9月23日
为增强的扇出消费者添加了区域	增加了 AWS 区域 对开发具有专用吞吐量的增强型扇出消费者的新支持。	2026年9月1日
添加了直播表和 Amazon S3 交付章节	为亚马逊 Kinesis Data Streams 添加了流式传输表和亚马逊 S3 交付。	2026 年 8 月 31 日
为数据平面 API 添加了试运行验证	添加了通过试运行测试您的权限并请求输入。	2026 年 8 月 25 日
添加了温吞吐量缩小行为和更新 CLI/API	更新了使用温吞吐量缩小行为创建和管理 Kinesis 数据流。	2026年7月22日
记录了 KCL 的单表迁移指标	使用迁移指标更新了 KCL 的单表格式。	2026 年 7 月 21 日
添加了 KCL v3.5 单表格式文档	为 KCL 添加了单表格式。	2026 年 6 月 18 日
增加了对最大为 10 MiB 的大型记录的支持	添加了处理大型记录。	2025 年 10 月 27 日
增加了对弹性测试的支持 AWS Fault Injection Service	添加了使用执行弹性测试 AWS Fault Injection Service。	2025 年 10 月 15 日

End-of-support 日期和版本生命周期政策	添加了有关 Amazon Kinesis Producer Library（KPL）和 Amazon Kinesis Client Library（KCL）的终止支持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 版本生命周期政策。	2025 年 3 月 13 日
增加了对跨账户共享数据流的支持	新增与其他账户共享您的数据流。	2023 年 11 月 22 日
增加了对按需和预配置数据流容量模式的支持	新增选择正确的直播模式	2021 年 11 月 29 日
在 Kinesis 客户端库中启用了多流处理	为 Java 消费者应用程序添加了使用相同的 KCL 2.x 处理多个数据流	2020 年 10 月 20 日
添加了两项新的 API 功能，以简化使用 Kinesis 数据流中的数据	增加了两项新的 API 功能，可简化对 Kinesis Data Streams 中数据的使用。参见列出分片。	2020 年 8 月 18 日
添加了有关控制 Kinesis Data Streams 的 VPC 终端节点访问权限的信息	添加了 Kinesis Data Streams 对 VPC 终端节点的访问控制。	2019 年 8 月 23 日
为使用增强扇出功能的消费者准备的新文档	新增开发具有专用吞吐量的增强型扇出消费者	2018 年 8 月 2 日
更新了服务限制摘要	在 “配额和限制” 中更清楚地描述了服务限制。	2018 年 6 月 6 日
服务器端加密的新内容	在 Amazon Kinesis Data Streams 中增加了数据保护。	2017 年 7 月 7 日
增强 CloudWatch 指标的新内容	更新了 Monito r Kinesis Data Streams。	2016 年 4 月 19 日

增强版 Kinesis Agent 的新内容	更新了使用 Kinesis 代理写入亚马逊 Kinesis Data Streams	2016 年 4 月 11 日
使用 Kinesis Agent 的新内容	添加了使用 Kinesis 代理写入亚马逊 Kinesis Data Streams。	2015 年 10 月 2 日
更新 0.10.0 版本的 KPL 内容	添加了使用 Amazon Kinesis 制作人库 (KPL) 开发制作人。	2015 年 7 月 15 日
更新可配置指标的 KCL 指标主题	更新了使用亚马逊监控亚马逊 Kinesis 客户端库。 CloudWatch	2015 年 7 月 9 日
Re-organized content	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1日
KPL 的新话题	添加了使用 Amazon Kinesis 制作人库 (KPL) 开发制作人。	2015年6月2日
新的 KCL 指标主题	添加了使用亚马逊监控亚马逊 Kinesis 客户端库。 CloudWatch	2015 年 5 月 19 日
对 KCL .NET 的支持	添加了在.NET 中开发 Amazon Kinesis 客户端库使用者。	2015 年 5 月 1 日
Support 对 KCL Node.js	在中添加了开发 Amazon Kinesis 客户端库使用者。 Node.js	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
全新 API PutRecords	添加了有关用于添加多条记录的新 PutRecords API 的信息 PutRecords。	2014 年 12 月 15 日

对标签的支持	添加了为你的 Amazon Kinesis Data Streams 资源添加标签。	2014 年 9 月 11 日
新 CloudWatch 指标	在 Amazon Kinesis Data Streams 的维度和指标GetRecords.IteratorAgeMilliseconds 中添加了该指标。	2014 年 9 月 3 日
新的监控章节	在亚马逊上添加了监控 Kinesis 数据流和监控亚马逊 Kinesis Data Streams 服务。CloudWatch	2014 年 7 月 30 日
默认分片限制	更新了配额和限制。默认分片限制已从 5 提高到 10。	2014 年 2 月 25 日
默认分片限制	更新了配额和限制。默认分片限制已从 2 提高到 5。	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。