开发人员指南

AWS Encryption SDK

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

AWS Encryption SDK: 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

那是什么 AWS Encryption SDK？ 1
在开源存储库中开发 2
与加密库和服务的兼容性 2
支持和维护 3
了解更多信息 4
发送反馈 4
概念 5
信封加密 6
数据密钥 7
包装密钥 8
密钥环和主密钥提供程序 9
加密上下文 9
加密的消息 11
算法套件 11
加密材料管理器 11
对称和非对称加密 12
密钥承诺 12
承诺策略 13
数字签名 15
该开发工具包的工作方式 16
如何 AWS Encryption SDK 加密数据 16
如何 AWS Encryption SDK 解密加密的消息 16
支持的算法套件 17
建议：具有密钥派生、签名和密钥承诺的 AES-GCM 17
其他支持的算法套件 18
与之互动 AWS KMS 19
最佳实践 20
配置 SDK 23
选择编程语言 23
选择包装密钥 23
使用多区域 AWS KMS keys 25
选择算法套件 45
限制加密数据密钥 56
创建发现筛选条件 62

需要加密上下文 ... 65
设置承诺策略 ... 72
使用串流数据 ... 73
缓存数据密钥 ... 73
密钥存储 ... 74
密钥商店术语和概念 ... 74
实施最低权限 ... 74
创建密钥库 ... 75
配置密钥存储操作 ... 76
配置您的关键商店操作 ... 77
创建分支密钥 ... 81
轮换您的活动分支密钥 ... 85
密钥环 ... 87
密钥环的工作方式 ... 87
密钥环兼容性 ... 89
对加密密钥环的不同要求 ... 90
兼容的密钥环和主密钥提供程序 ... 90
AWS KMS 钥匙圈 ... 92
AWS KMS 密钥环所需的权限 ... 93
在 AWS KMS 钥匙圈 AWS KMS keys 中识别 ... 94
创建密 AWS KMS 钥环 ... 94
使用 AWS KMS 发现密钥环 ... 108
使用 AWS KMS 区域发现密钥环 ... 115
AWS KMS 分层钥匙圈 ... 123
工作原理 ... 125
先决条件 ... 126
所需的权限 ... 126
选择缓存 ... 127
创建分层密钥环 ... 139
AWS KMS ECDH 钥匙圈 ... 150
AWS KMS ECDH 密钥环所需的权限 ... 151
创建 AWS KMS ECDH 密钥环 ... 151
创建 AWS KMS ECDH 发现密钥环 ... 158
原始 AES 密钥环 ... 163
原始 RSA 密钥环 ... 171
未加工的 ECDH 钥匙圈 ... 179

- 创建原始的 ECDH 密钥环 ... 180
- Multi-keyrings ... 197
- 编程语言 ... 207
- C ... 207
- 安装 ... 208
- 使用 C 开发工具包 ... 209
- 示例 ... 213
- .NET ... 219
- 安装和构建 ... 221
- 调试 ... 221
- 示例 ... 222
- Go ... 229
- 先决条件 ... 230
- 安装 ... 230
- Java ... 231
- 先决条件 ... 231
- 安装 ... 232
- 示例 ... 233
- JavaScript ... 246
- 兼容性 ... 247
- 安装 ... 248
- 模块 ... 249
- 示例 ... 252
- Python ... 260
- 先决条件 ... 260
- 安装 ... 260
- 示例 ... 261
- Rust ... 269
- 先决条件 ... 269
- 安装 ... 270
- 示例 ... 270
- 命令行界面 ... 272
- 安装 CLI ... 274
- 如何使用 CLI ... 277
- 示例 ... 288
- 语法和参数参考 ... 310

版本 322
数据密钥缓存 325
如何使用数据密钥缓存 326
使用数据密钥缓存： Step-by-step 326
数据密钥缓存示例：加密字符串 334
设置缓存安全阈值 350
数据密钥缓存详细信息 351
数据密钥缓存的工作方式 351
创建加密材料缓存 354
创建缓存加密材料管理器 355
在数据密钥缓存条目中包含哪些内容？ 355
加密上下文：如何选择缓存条目 356
我的应用程序是否使用缓存的数据密钥？ 356
数据密钥缓存示例 357
本地缓存结果 358
代码示例 359
CloudFormation 模板 370
的版本 AWS Encryption SDK 386
C 386
C# /.NET 387
命令行界面 (CLI) 388
Java 389
Go 391
JavaScript 391
Python 392
Rust 394
版本详情 394
1.7 之前的版本。x 394
版本 1.7。x 395
版本 2.0。x 397
版本 2.2。x 398
版本 2.3。x 399
迁移你的 AWS Encryption SDK 400
如何迁移和部署 401
阶段 1：将您的应用程序更新到最新版本 1.x 402
阶段 2：将您的应用程序更新到最新版本 403

更新 AWS KMS 主密钥提供程序 ... 403
迁移到严格模式 ... 404
迁移到发现模式 ... 408
更新 AWS KMS 钥匙圈 ... 410
设置您的承诺策略 ... 413
如何设置您的承诺策略 ... 414
对迁移到至最新版本进行故障排除 ... 424
弃用或移除的对象 ... 425
配置冲突：承诺策略和算法套件 ... 425
配置冲突：承诺策略和加密文字 ... 426
密钥承诺验证失败 ... 427
其他加密故障 ... 427
其他加密故障 ... 427
回滚注意事项 ... 427
常见问题 ... 429
和？有何 AWS Encryption SDK 不同 AWS SDKs？ ... 429
与 Amazon S3 加密客户端有何 AWS Encryption SDK 不同？ ... 430
支持哪些加密算法 AWS Encryption SDK，哪一种是默认算法？ ... 430
如何生成初始化向量 (IV) 以及将其存储在何处？ ... 430
如何生成、加密和解密每个数据密钥？ ... 431
如何跟踪用于加密我的数据的数据密钥？ ... 431
如何将加密的数据密钥与其加密数据一起 AWS Encryption SDK 存储？ ... 431
AWS Encryption SDK 消息格式会给我的加密数据增加多少开销？ ... 431
我是否可以使用自己的主密钥提供程序？ ... 432
我是否可以使用多个包装密钥加密数据？ ... 432
我可以使用哪些数据类型进行加密 AWS Encryption SDK？ ... 432
AWS Encryption SDK 加密和解密 input/output (I/O) 流是如何进行的？ ... 432
参考 ... 434
消息格式参考 ... 434
标头结构 ... 435
正文结构 ... 442
脚注结构 ... 446
消息格式示例 ... 447
帧数据（消息格式版本 1） ... 447
帧数据（消息格式版本 2） ... 451
Non-framed 数据（消息格式版本 1） ... 453

正文 AAD 参考 457
算法参考 458
初始化向量参考 462
AWS KMS 分层钥匙圈技术细节 462
文档历史记录 464
最近的更新 464
早期更新 466
..... cdlxviii

那是什么 AWS Encryption SDK？

AWS Encryption SDK 是一个客户端加密库，旨在让每个人都能轻松地使用行业标准和最佳实践对数据进行加密和解密。这样，您就可以专注于应用程序的核心功能，而不是如何以最佳方式加密和解密数据。在 AWS Encryption SDK Apache 2.0 许可证下免费提供。

这些 AWS Encryption SDK 答案将为你解答如下问题：

- 我应该使用哪种加密算法？
- 我应该如何使用该算法或在哪种模式下使用？
- 我如何生成加密密钥？
- 我如何保护加密密钥，以及将其存储在什么位置？
- 我如何使加密的数据具有便携性？
- 我如何确保目标接收者可以读取我的加密数据？
- 我如何确保在写入和读取我的加密数据之间不会修改这些数据？
- 如何使用 AWS KMS 返回的数据密钥？

使用 AWS Encryption SDK，您可以定义主密钥提供程序或密钥环，用于确定使用哪些封装密钥来保护数据。然后，您可以使用提供的简单方法对数据进行加密和解密。 AWS Encryption SDK剩下的 AWS Encryption SDK 就交给了。

如果没有 AWS Encryption SDK，您可能要花更多的精力来构建加密解决方案，而不是花在应用程序的核心功能上。他们通过提供以下内容来 AWS Encryption SDK 回答这些问题。

遵循加密最佳实践的默认实施

默认情况下，会为其加密的每个数据对象 AWS Encryption SDK 生成一个唯一的数据密钥。这遵循在每个加密操作中使用唯一数据密钥的加密最佳实践。

使用安全、经过身份验证的对称密钥算法对您的数据进行 AWS Encryption SDK 加密。有关更多信息，请参阅 the section called “支持的算法套件”。

使用包装密钥保护数据密钥的框架

通过在一个或多个封装密钥下加密数据密钥来 AWS Encryption SDK 保护加密您的数据密钥。通过提供一个使用多个包装密钥加密数据密钥的框架， AWS Encryption SDK 这有助于使您的加密数据具有可移植性。

例如，对来自本地 HSM AWS KMS key 的 AWS KMS 输入和密钥下的数据进行加密。为了避免其中一个密钥不可用或调用方无权使用这两个密钥，您可以使用其中一个包装密钥解密数据。

采用某种格式的消息，它存储加密的数据密钥以及加密的数据

将加密的数据和加密的数据密钥一起 AWS Encryption SDK 存储在使用定义数据格式的加密消息中。这意味着您无需跟踪或保护加密数据的数据密钥，因为这些密钥是为你 AWS Encryption SDK 做的。

的某些语言实现 AWS Encryption SDK 需要 S AWS DK，但 AWS Encryption SDK 不需要 SDK AWS 账户 ，也不依赖于任何 AWS 服务。 AWS 账户 只有当您选择使用来保护您的数据时 AWS KMS keys，才需要一个。

在开源存储库中开发

AWS Encryption SDK 是在上的开源存储库中开发的 GitHub。您可以使用这些存储库查看代码、阅读和提交问题，并且查找特定于您的语言实施的信息。

- AWS Encryption SDK for C — aws-encryption-sdk-c
- AWS Encryption SDK 用于.NE T-`aws-encryption-sdk` 存储库的.NET 目录。
- AWS 加密 CLI — aws-encryption-sdk-cli
- AWS Encryption SDK for Java — aws-encryption-sdk-java
- AWS Encryption SDK for JavaScript — aws-encryption-sdk-javascript
- AWS Encryption SDK for Python — aws-encryption-sdk-python
- AWS Encryption SDK 对于 Rus t — `aws-encryption-sdk` 存储库的 Rust 目录。
- AWS Encryption SDK for G o — `aws-encryption-sdk` 存储库的 Go 目录

与加密库和服务的兼容性

AWS Encryption SDK 有几种编程语言支持。所有语言实施都是可互操作的。您可以使用一种语言实施进行加密，并使用另一种语言实施进行解密。互操作性可能受到语言约束的限制。如果是这样，这些约束将在有关语言实施的主题中进行描述。此外，在加密和解密时，必须使用兼容的密钥环或主密钥和主密钥提供程序。有关更多信息，请参阅 the section called “密钥环兼容性”。

但是， AWS Encryption SDK 无法与其他库互操作。由于每个库以不同的格式返回加密的数据，因此，您无法使用一个库进行加密并使用另一个库进行解密。

DynamoDB 加密客户端和 Amazon S3 客户端加密

AWS Encryption SDK 无法解密由 DynamoDB 加密客户端或 Amazon S3 客户端加密加密的数据。这些库无法解密返回的加密消息。 AWS Encryption SDK

AWS Key Management Service (AWS KMS)

AWS Encryption SDK 可以使用的AWS KMS keys和数据密钥来保护您的数据，包括多区域 KMS 密钥。例如，您可以将配置 AWS Encryption SDK 为在 AWS KMS keys 中的一个或多个下加密您的数据 AWS 账户。但是，您必须使用 AWS Encryption SDK 来解密该数据。

AWS Encryption SDK 无法解密 AWS KMS 加密或操作返回的密文。ReEncrypt同样， AWS KMS 解密操作无法解密返回的加密消息。 AWS Encryption SDK

仅 AWS Encryption SDK 支持对称加密 KMS 密钥。您无法使用非对称 KMS 密钥在 AWS Encryption SDK中进行加密或签名。 AWS Encryption SDK 为对消息进行签名的算法套件生成自己的 ECDSA 签名密钥。

支持和维护

AWS Encryption SDK 使用与 AWS SDK 和工具相同的维护策略，包括其版本控制和生命周期阶段。作为最佳实践，我们建议您使用 AWS Encryption SDK 适用于您的编程语言的最新可用版本，并在新版本发布时进行升级。当版本需要进行重大更改时，例如从 1.7 之前的 AWS Encryption SDK 版本升级。 x 到 2.0 版本。 x 及以后，我们会提供详细的说明来帮助您。

的每种编程语言实现都 AWS Encryption SDK 是在单独的开源 GitHub 存储库中开发的。每个版本的生命周期和支持阶段可能因存储库而异。例如，给定版本的在一种编程语言中 AWS Encryption SDK 可能处于正式发布（完全支持）阶段，但处于另一种编程语言的 end-of-support阶段。我们建议您尽可能使用全面支持的版本，避免使用不再受支持的版本。

要查找您的编程语言 AWS Encryption SDK 版本的生命周期阶段，请查看每个 AWS Encryption SDK 存储库中的`SUPPORT_POLICY.rst`文件。

- AWS Encryption SDK for C — s upport_policy.r
- AWS Encryption SDK 适用于.NET — s upp ort_policy.
- AWS 加密 CLI — s upport_policy.r s
- AWS Encryption SDK for Java — s upport_policy.r
- AWS Encryption SDK for JavaScript — s upport_policy.r
- AWS Encryption SDK for Python — s upport_policy.r

有关更多信息，请参阅的版本 AWS Encryption SDKAWS SDKs 和工具参考指南中的 AWS SDKs 和和和工具维护政策。

了解更多信息

有关 AWS Encryption SDK 和客户端加密的更多信息，请尝试以下来源。

- 有关该开发工具包中使用的术语和概念的帮助，请参阅中的概念 AWS Encryption SDK。
- 有关最佳实践准则，请参阅 的最佳实践 AWS Encryption SDK。
- 有关该开发工具包的工作方式的信息，请参阅该开发工具包的工作方式。
- 有关展示如何在中配置选项的示例 AWS Encryption SDK，请参阅正在配置 AWS Encryption SDK。
- 有关详细的技术信息，请参阅参考。
- 有关的技术规格 AWS Encryption SDK，请参阅中的AWS Encryption SDK 规范 GitHub。
- 有关使用问题的答案 AWS Encryption SDK，请阅读AWS 加密工具讨论论坛并发帖。

有关 AWS Encryption SDK 在不同编程语言中实现的信息。

- C: AWS Encryption SDK for C 请参阅 AWS Encryption SDK C 文档和上的aws-encryption-sdk-c存储库 GitHub。
- C#/.NET：参见AWS Encryption SDK 对于.NET，存储库的aws-encryption-sdk-net`aws-encryption-sdk`目录已打开。 GitHub
- 命令行界面：参见AWS Encryption SDK 命令行界面，阅读 AWS 加密 CLI 的文档，以及上面的aws-encryption-sdk-cli存储库 GitHub。
- Java：参见 AWS Encryption SDK for Java AWS Encryption SDK Javadoc 和上面的aws-encryption-sdk-java存储库。 GitHub

  JavaScript: 请参阅the section called “JavaScript”，aws-encryption-sdk-javascript存储库已打开 GitHub。
- Python：参见 AWS Encryption SDK for Python AWS Encryption SDK Python 文档和上aws-encryption-sdk-python面的存储库 GitHub。

发送反馈

我们欢迎您提供反馈！如果您有任何疑问或意见或者要报告问题，请使用以下资源。

- 如果您在中发现潜在的安全漏洞 AWS Encryption SDK，请通知 AWS 安全部门。不要创建公开 GitHub 问题。
- 要提供相关反馈 AWS Encryption SDK，请在 GitHub 存储库中提交您正在使用的编程语言的问题。
- 要提供有关本文档的反馈，请使用该页面上的反馈链接。您也可以提交问题或为aws-encryption-sdk-docs本文档的开源存储库做出贡献 GitHub。

中的概念 AWS Encryption SDK

本节介绍中使用的概念 AWS Encryption SDK，并提供词汇表和参考资料。它旨在帮助您了解其 AWS Encryption SDK 工作原理以及我们用来描述它的术语。

需要帮助？

- 了解如何 AWS Encryption SDK 使用信封加密来保护您的数据。
- 了解信封加密的要素：保护数据的数据密钥以及保护数据密钥的包装密钥。
- 了解决定您使用哪种包装密钥的密钥环和主密钥提供程序。
- 了解可增强加密过程完整性的加密上下文。这是可选的，但却是我们建议的最佳实践。
- 了解加密方法返回的加密消息。
- 然后，您就可以用自己喜欢的编程语言使用了。 AWS Encryption SDK

主题

- 信封加密
- 数据密钥
- 包装密钥
- 密钥环和主密钥提供程序
- 加密上下文
- 加密的消息
- 算法套件
- 加密材料管理器
- 对称和非对称加密
- 密钥承诺

- 承诺策略
- 数字签名

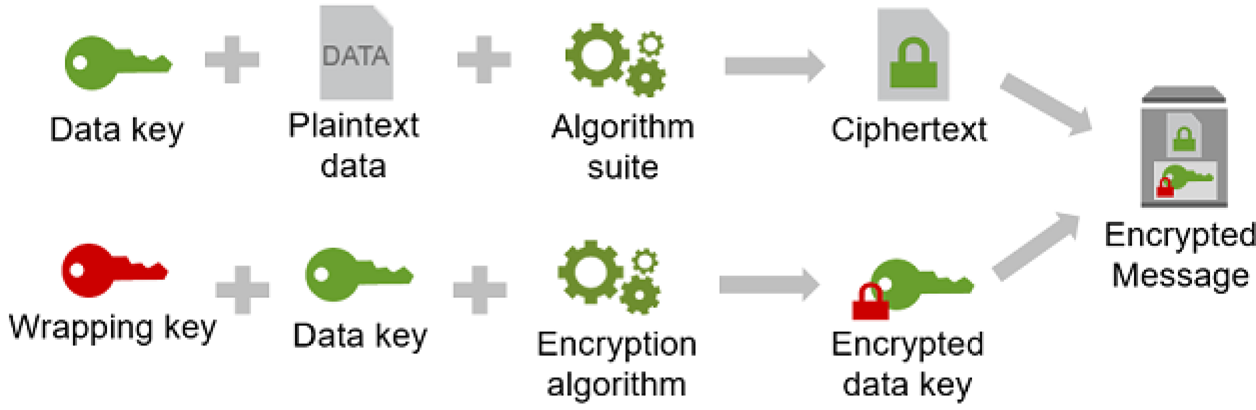
信封加密

加密的数据的安全性部分取决于如何保护可解密该数据的数据密钥。保护数据密钥的一种公认的最佳实践是对其进行加密。为此，您需要另一个加密密钥，称为密钥加密密钥或包装密钥。使用包装密钥加密数据密钥的做法称为信封加密。

保护数据密钥

使用唯一的数据 AWS Encryption SDK 密钥对每条消息进行加密。然后，对您指定的包装密钥下的数据密钥进行加密。将加密的数据密钥与加密的数据一并存储在其返回的加密消息中。

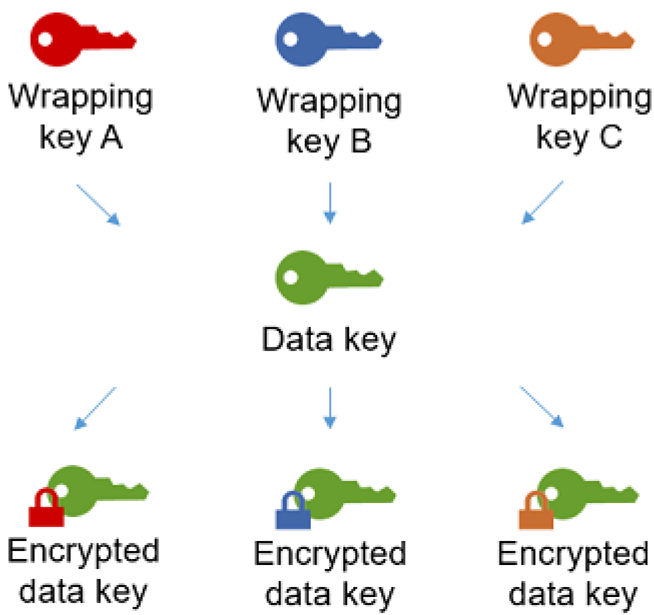
要指定包装密钥，请使用密钥环或主密钥提供程序。



在多个包装密钥下加密相同的数据

您可以在多个包装密钥下加密数据密钥。您可能希望为不同的用户提供不同的包装密钥，或者提供不同类型的包装密钥，或者位于不同的位置。每个包装密钥都加密相同的数据密钥。将所有加密数据密钥和加密数据 AWS Encryption SDK 存储在加密消息中。

要解密数据，您需要提供可以解密任一加密数据密钥的包装密钥。



结合多种算法的优势

要加密您的数据，默认情况下， AWS Encryption SDK 使用具有 AES-GCM 对称加密、密钥派生函数 (HKDF) 和签名的复杂算法套件。要加密数据密钥，您可以指定适合您的包装密钥的对称或非对称加密算法。

通常，与非对称或公有密钥加密 相比，对称密钥加密算法速度更快，生成的密文更小。但公有密钥算法可提供固有的角色分离和更轻松的密钥管理。为了结合每种算法的优势，您可以使用对称密钥加密功能加密数据，然后使用公有密钥加密功能加密数据密钥。

数据密钥

数据密钥 是 AWS Encryption SDK 用于加密数据的加密密钥。每个数据密钥都是一个符合加密密钥要求的字节数组。除非您使用数据密钥缓存，否则会 AWS Encryption SDK 使用唯一的数据密钥来加密每条消息。

您无需指定、生成、实施、扩展、保护或使用数据密钥。当您调用加密和解密操作时， AWS Encryption SDK 会替您完成这些工作。

为了保护您的数据密钥，使用一个或多个密钥 AWS Encryption SDK 加密密钥（称为包装密钥或主密钥）对其进行加密。 AWS Encryption SDK 使用您的纯文本数据密钥加密您的数据后，它会尽快将其从内存中删除。然后，它将加密的数据密钥与加密的数据一并存储在加密操作返回的加密消息中。有关更多信息，请参阅 the section called “该开发工具包的工作方式”。

Tip

在中 AWS Encryption SDK，我们将数据密钥与数据加密密钥区分开来。一些支持的算法套件（包括默认套件）使用密钥派生函数以防止数据密钥达到其加密限制。密钥派生函数将数据密钥作为输入，并返回实际用于加密数据的数据加密密钥。因此，我们通常说数据是“根据”数据密钥加密的，而不是“由”数据密钥加密的。

每个加密的数据密钥都包含元数据，包括对其进行加密的包装密钥的标识符。此元数据使在 AWS Encryption SDK 解密时可以更轻松地识别有效的包装密钥。

包装密钥

包装密钥是一种密钥加密密钥， AWS Encryption SDK 使用该密钥加密用于加密数据的数据密钥。可以使用一个或多个包装密钥加密每个明文数据密钥。在配置密钥环或主密钥提供程序时，您可以决定使用哪些包装密钥来保护您的数据。

Note

包装密钥是指密钥环或主密钥提供程序中的密钥。主密钥通常与您在使用主密钥提供程序时实例化的 `MasterKey` 类相关联。

AWS Encryption SDK 支持多种常用的包装密钥，例如 AWS Key Management Service (AWS KMS) 对称 AWS KMS keys（包括多区域 KMS 密钥）、原始 AES-GCM（高级加密 Standard/Galois 计数器模式）密钥和原始 RSA 密钥。您还可以扩展或实施自己的包装密钥。

在使用信封加密时，您需要保护包装密钥以防止未经授权的访问。您可以通过以下任何方式来执行此操作：

- 使用专用于该用途的 Web 服务，如 AWS Key Management Service (AWS KMS)。
- 使用硬件安全模块 (HSM)，例如，AWS CloudHSM 提供的模块。
- 使用其他密钥管理工具和服务。

如果您没有密钥管理系统，我们建议您使用 AWS KMS。与 AWS Encryption SDK 集成 AWS KMS ，可帮助您保护和使用包装密钥。但是， AWS Encryption SDK 不需要 AWS 或任何 AWS 服务。

密钥环和主密钥提供程序

要指定用于加密和解密的包装密钥，您可以使用密钥环或主密钥提供程序。您可以使用他们提供的密钥环和主密钥提供程序，也可以设计自己的实现。 AWS Encryption SDK AWS Encryption SDK 提供相互兼容的密钥环和主密钥提供程序，但须遵守语言限制。有关更多信息，请参阅 密钥环兼容性。

密钥环 生成、加密和解密数据密钥。定义密钥环时，可以指定用于加密数据密钥的包装密钥。大多数密钥环至少指定一个包装密钥或一项提供和保护包装密钥的服务。您也可以定义不带包装密钥的密钥环，或者使用其他配置选项定义更复杂的密钥环。有关选择和使用 AWS Encryption SDK 定义的钥匙圈的帮助，请参阅密钥环。

以下编程语言支持密钥环：

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK 对于.NET
- 版本 3。 的 x AWS Encryption SDK for Java
- 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
- 版本 1。 x 的 fo r AWS Encryption SDK Rust
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

主密钥提供程序是密钥环的替代方案。主密钥提供程序返回您指定的包装密钥（或主密钥）。每个主密钥与一个主密钥提供程序相关联，但主密钥提供程序通常提供多个主密钥。Java、Python 和 AWS 加密 CLI 支持主密钥提供程序。

您必须指定用于加密的密钥环（或主密钥提供程序）。您可以指定相同的密钥环（或主密钥提供程序）或不同的密钥环进行解密。加密时， AWS Encryption SDK 使用您指定的所有包装密钥来加密数据密钥。解密时， AWS Encryption SDK 仅使用您指定的包装密钥来解密加密的数据密钥。指定用于解密的包装密钥是可选的，但这是 AWS Encryption SDK 最佳实践。

有关指定包装密钥的详细信息，请参阅 选择包装密钥。

加密上下文

为了提高加密操作安全性，请在所有加密数据的请求中包含加密上下文。使用加密上下文是可选的，但这是我们建议遵守的加密最佳实践。

加密上下文 是一组名称值对，其中包含任意非机密经过身份验证的附加数据。加密上下文可以包含您选择的任何数据，但它通常包含用于日志记录和跟踪的数据，例如，有关文件类型、用途或所有权的数据。当您加密数据时，加密上下文以加密方式绑定到加密的数据，以便需要使用相同的加密上下文解密数据。 AWS Encryption SDK 将加密上下文以明文形式包含在其返回的加密的消息标头中。

AWS Encryption SDK 使用的加密上下文包括您指定的加密上下文和加密材料管理器 (CMM) 添加的公钥对。具体来说，每当您使用带签名的加密算法时，CMM 都会向加密上下文（由保留名称、`aws-crypto-public-key` 和表示公有验证密钥的值组成）添加一个名称/值对。加密上下文中的`aws-crypto-public-key`名称由保留 AWS Encryption SDK ，不能在加密上下文中用作任何其他对中的名称。有关详细信息，请参阅“消息格式参考”中的 AAD。

下面的示例加密上下文由请求中指定的两个加密上下文对和 CMM 添加的公有密钥对组成。

```
"Purpose"="Test", "Department"="IT", aws-crypto-public-key=<public key>
```

要解密数据，您可以传入加密的消息。由于 AWS Encryption SDK 可以从加密的邮件标头中提取加密上下文，因此您无需单独提供加密上下文。但是，加密上下文可帮助您确认解密的是正确的加密消息。

- 在 AWS Encryption SDK 命令行界面 (CLI) 中，如果您在 decrypt 命令中提供加密上下文，CLI 将在返回明文数据前验证加密消息的加密上下文中是否存在这些值。
- 在其他编程语言实现中，解密响应包含加密上下文和明文数据。应用程序中的解密函数应始终在返回明文数据前验证解密响应中的加密上下文是否包含加密请求（或子集）中的加密上下文。

> **Note**
>
> 以下版本支持所需的加密上下文 CMM，您可以使用它来要求在所有加密请求中使用加密上下文。
>
> - 版本 3。 的 x AWS Encryption SDK for Java
> - 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
> - 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
> - 版本 1。 x 的 fo r AWS Encryption SDK Rust
> - 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

在选择加密上下文时，请记住它不是机密的。加密上下文以明文形式显示在 AWS Encryption SDK 返回的加密的消息标头中。如果您使用的是 AWS Key Management Service，加密上下文也可能以纯文本形式出现在审计记录和日志中，例如。 AWS CloudTrail

有关在代码中提交和验证加密上下文的示例，请参阅您的首选编程语言示例。

加密的消息

当您使用加密数据时 AWS Encryption SDK，它会返回一条加密的消息。

加密的消息是一种可移植的格式化数据结构，包含加密的数据、数据密钥的加密副本、算法 ID 以及可选的加密上下文和数字签名。 AWS Encryption SDK 中的加密操作返回加密的消息，解密操作将加密的消息作为输入。

将加密的数据及其加密的数据密钥合并在一起可以简化解密操作，您不必将加密的数据密钥独立于它们加密的数据进行存储和管理。

有关加密的消息的技术信息，请参阅加密的消息格式。

算法套件

AWS Encryption SDK 使用算法套件对加密和解密操作返回的加密消息中的数据进行加密和签名。 AWS Encryption SDK 支持一些算法套件。所有支持的套件将高级加密标准 (AES) 作为主要算法，并将其与其他算法和值组合使用。

AWS Encryption SDK 建立了推荐的算法套件，作为所有加密操作的默认算法套件。随着标准和最佳实践的不断改进，默认套件可能会发生变化。您可以在加密数据的请求中或在创建加密材料管理器（CMM）时指定备用算法套件，但除非您的环境需要使用备用套件，否则，最好使用默认套件。当前的默认值是 AES-GCM，具有基于 HMAC 的 extract-and-expand密钥派生函数 (H KDF)、密钥承诺、椭圆曲线数字签名算法 (ECDSA) 签名和 256 位加密密钥。

如果您的应用程序需要高性能，并且加密数据的用户和解密数据的用户同样受到信任，则可以考虑指定不带数字签名的算法套件。但是，我们强烈建议使用包含密钥承诺和密钥派生函数的算法套件。支持没有这些功能的算法套件仅为了保持向后兼容。

加密材料管理器

加密材料管理器（CMM）组装用于加密和解密数据的加密材料。加密材料 包含明文和加密的数据密钥以及可选的消息签名密钥。您永远不会直接与 CMM 交互。加密和解密方法替您进行处理。

您可以使用默认 CMM 或 AWS Encryption SDK 提供的缓存 CMM，也可以编写自定义 CMM。您可以指定 CMM，但这不是必需的。当您指定密钥环或主密钥提供程序时， AWS Encryption SDK 会为您创建默认 CMM。默认 CMM 从您指定的密钥环或主密钥提供程序获取加密或解密材料。这可能涉及调用一个加密服务，如 AWS Key Management Service (AWS KMS)。

由于 CMM 充当 AWS Encryption SDK 与密钥环（或主密钥提供程序）之间的联络人，因此它是自定义和扩展的理想点，例如支持策略实施和缓存。 AWS Encryption SDK 提供了缓存 CMM 以支持数据密钥缓存。

对称和非对称加密

对称加密使用相同的密钥来加密和解密数据。

非对称加密使用数学相关的数据密钥对。密钥对中的一个密钥对数据进行加密；只有密钥对中的另一个密钥可以解密数据。

AWS Encryption SDK 使用信封加密。使用对称数据密钥加密您的数据。使用一个或多个对称或非对称包装密钥加密对称数据密钥。返回一条加密的消息，其中包含加密数据和至少一个数据密钥的加密副本。

加密数据（对称加密）

要加密您的数据， AWS Encryption SDK 使用对称数据密钥和包含对称加密算法的算法套件。要解密数据， AWS Encryption SDK 使用相同的数据密钥和相同的算法套件。

加密数据密钥（对称或非对称加密）

您为加密和解密操作提供的密钥环或主密钥提供程序决定了对称数据密钥的加密和解密方式。您可以选择使用对称加密的密钥环或主密钥提供程序（例如密钥环），也可以选择使用非对称加密的 AWS KMS 密钥环或主密钥提供程序，例如原始的 RSA 密钥环或。`JceMasterKey`

密钥承诺

AWS Encryption SDK 支持密钥承诺（有时称为稳健性），这是一种安全属性，可保证每个密文只能解密为单个纯文本。为此，密钥承诺可保证仅使用加密消息的数据密钥来解密消息。使用密钥承诺进行加密和解密是 AWS Encryption SDK 最佳实践。

大多数现代对称密码（包括 AES）使用单个密钥对明文进行加密，例如 AWS Encryption SDK 用于加密每条明文消息的唯一数据密钥。使用相同的数据密钥解密这些数据会返回与原始数据相同的明文。使用不同的密钥解密通常会失败。但是，有可能使用两个不同的密钥解密加密文字。在极少数情况下，找到一个密钥来将几个字节的加密文字解密成不同但仍然可以理解的明文是可行的。

AWS Encryption SDK 始终使用一个唯一的数据密钥对每条纯文本消息进行加密。可能会使用多个包装密钥（或主密钥）加密该数据密钥，但包装密钥始终加密相同的数据密钥。尽管如此，手动制作的复杂加密的消息实际上可能包含不同的数据密钥，每个数据密钥都由不同的包装密钥加密。例如，如果一个用户对加密的消息进行解密，将返回 0x0（false），而另一个用户解密相同的加密消息则得到 0x1（true）。

为防止出现这种情况， AWS Encryption SDK 支持加密和解密时的密钥承诺。当使用密钥承诺对消息 AWS Encryption SDK 进行加密时，它会以加密方式将生成密文的唯一数据密钥绑定到密钥承诺字符串（非秘密数据密钥标识符）。然后，将密钥承诺字符串存储在加密消息的元数据中。当它使用密钥承诺解密消息时， AWS Encryption SDK 会验证数据密钥是否是该加密消息的唯一密钥。如果数据密钥验证失败，则解密操作将失败。

在版本 1.7.x 中引入了对密钥承诺的支持，可以解密带有密钥承诺的消息，但不会使用密钥承诺进行加密。您可以使用此版本全面部署使用密钥承诺解密加密文字的功能。版本 2.0.x 包括对密钥承诺的全面支持。默认情况下，仅使用密钥承诺进行加密和解密。对于不需要解密由早期版本加密的密文的应用程序来说，这是一种理想的配置。 AWS Encryption SDK

尽管使用密钥承诺进行加密和解密是最佳实践，但我们允许您决定何时使用密钥，并允许您调整采用密钥的进度。从 1.7 版本开始。 x， AWS Encryption SDK 支持用于设置默认算法套件并限制可能使用的算法套件的承诺策略。此策略决定您的数据是否通过密钥承诺进行加密和解密。

密钥承诺会生成稍大（30 多个字节）的加密消息，并且需要更多时间来处理。如果您的应用程序对大小或性能非常敏感，则可以选择退出密钥承诺。但只有在必要时才这样做。

有关迁移到版本 1.7.x 和 2.0.x 的更多信息，包括其密钥承诺功能，请参阅 迁移你的 AWS Encryption SDK。有关密钥承诺的技术信息，请参阅 the section called “算法参考” 和 the section called “消息格式参考”。

承诺策略

承诺策略是一种配置设置，用于确定您的应用程序是否使用密钥承诺进行加密和解密。使用密钥承诺进行加密和解密是 AWS Encryption SDK 最佳实践。

承诺策略有三个值。

Note

您可能需要水平或垂直滚动才能查看整个表。

承诺策略值

值	使用密钥承诺进行加密	不使用密钥承诺进行加密	使用密钥承诺进行解密	不使用密钥承诺进行解密
ForbidEncryptAllowDecrypt	✗	✓	✓	✓
RequireEncryptAllowDecrypt	✓	✗	✓	✓
RequireEncryptRequireDecrypt	✓	✗	✓	✗

1.7 AWS Encryption SDK 版中引入了承诺策略设置。 x。该设置在所有支持的编程语言中都有效。

- `ForbidEncryptAllowDecrypt` 使用或不使用密钥承诺进行解密，但不会使用密钥承诺进行加密。此值在 1.7 版本中引入。 x，旨在让所有运行您的应用程序的主机在遇到使用密钥承诺加密的密文之前准备好使用密钥承诺进行解密。
- `RequireEncryptAllowDecrypt` 始终使用密钥承诺进行加密。可以使用或不使用密钥承诺进行解密。此值在版本 2.0.x 中引入，允许您使用密钥承诺开始加密，但仍然可以不使用密钥承诺解密旧加密文字。
- `RequireEncryptRequireDecrypt` 仅使用密钥承诺进行加密和解密。此值是版本 2.0.x 的默认值。当您确定所有加密文字都使用密钥承诺进行加密时，请使用此值。

承诺策略设置决定了您可以使用哪些算法套件。从 1.7 版本开始。 x， AWS Encryption SDK 支持密钥承诺的算法套件；有签名和不带签名。如果您指定的算法套件与您的承诺策略冲突，则 AWS Encryption SDK 会返回错误。

有关设置承诺策略的帮助，请参阅 设置您的承诺策略。

数字签名

使用经过身份验证的 AWS Encryption SDK 加密算法 AES-GCM 对您的数据进行加密，解密过程无需使用数字签名即可验证加密消息的完整性和真实性。但是，由于 AES-GCM 使用对称密钥，所以能够解密用于解密加密文字的数据密钥的任何人员都可以手动创建新的加密的加密文字，从而造成潜在的安全问题。例如，如果您使用 AWS KMS key 作为包装密钥，则具有`kms:Decrypt`权限的用户无需调用即可创建加密的密文。`kms:Encrypt`

为避免此问题， AWS Encryption SDK 支持在加密消息的末尾添加椭圆曲线数字签名算法 (ECDSA) 签名。使用签名算法套件时，会为每封加密消息 AWS Encryption SDK 生成临时私钥和公钥对。将公钥 AWS Encryption SDK 存储在数据密钥的加密环境中，并丢弃私钥。这样可以确保任何人都无法创建另一个使用公钥进行验证的签名。该算法将公钥绑定到加密的数据密钥作为邮件标头中的其他经过身份验证的数据，从而防止只能解密消息的用户更改公钥或影响签名验证。

签名验证会增加大量的解密性能成本。如果加密数据的用户和解密数据的用户同样受到信任，请考虑使用不包括签名功能的算法套件。

> Note
>
> 如果密钥环或对封装加密材料的访问权限未在加密器和解密器之间划清界限，则数字签名不提供任何加密价值。

AWS KMS 密钥环（包括非对称 RSA AWS KMS 密钥环）可以根据密钥策略和 IAM 策略在加密器和解密器之间进行划分。 AWS KMS

由于其加密性质，以下密钥环无法在加密器和解密器之间进行划分：

- AWS KMS 分层钥匙圈
- AWS KMS ECDH 钥匙圈
- 原始 AES 密钥环
- 原始 RSA 密钥环
- 未加工的 ECDH 钥匙圈

AWS Encryption SDK 工作原理

本节中的工作流程说明了如何 AWS Encryption SDK 加密数据和解密加密的消息。这些工作流使用默认特征描述基本流程。有关定义和使用自定义组件的详细信息，请参阅每种支持的语言实现的 GitHub 存储库。

AWS Encryption SDK 使用信封加密来保护您的数据。每条消息都使用唯一的数据密钥进行加密。然后，使用您指定的包装密钥加密数据密钥。要解密加密的邮件， AWS Encryption SDK 使用您指定的包装密钥来解密至少一个加密的数据密钥。然后其可解密加密文字并返回一条明文消息。

需要有关 AWS Encryption SDK所用术语方面的帮助？ 请参阅the section called “概念”。

如何 AWS Encryption SDK 加密数据

AWS Encryption SDK 提供了加密字符串、字节数组和字节流的方法。有关代码示例，请参阅各个 编程语言 部分的示例主题。

1. 创建指定用于保护您数据的包装密钥的密钥环（或主密钥提供程序）。
2. 将密钥环和明文数据传递给加密方法。建议您传入一个可选的非机密加密上下文。
3. 加密方法要求密钥环提供加密材料。密钥环返回消息的唯一数据加密密钥：一个纯文本数据密钥和一个由每个指定的包装密钥加密的数据密钥的副本。
4. 加密方法使用明文数据密钥加密数据，然后丢弃明文数据密钥。如果您提供加密上下文（ AWS Encryption SDK 最佳实践），加密方法会以加密方式将加密上下文绑定到加密的数据。
5. 加密方法返回一条加密消息，其中包含加密的数据、加密的数据密钥和包括加密上下文（如果有）在内的其他元数据。

如何 AWS Encryption SDK 解密加密的消息

AWS Encryption SDK 提供了解密加密消息并返回纯文本的方法。有关代码示例，请参阅各个 编程语言 部分的示例主题。

解密加密消息的密钥环（或主密钥提供程序）必须与用于加密消息的密钥环兼容。其中一个包装密钥必须能够解密加密消息中的加密数据密钥。有关与密钥环和主密钥提供程序的兼容性的信息，请参阅 the section called “密钥环兼容性”。

1. 使用可解密数据的包装密钥创建密钥环或主密钥提供程序。您可以使用与加密方法相同的密钥环或其他密钥环。
2. 将加密消息和密钥环传递给解密方法。

3. 解密方法要求密钥环或主密钥提供程序解密加密消息中的一个加密数据密钥。它传入加密的消息中的信息，包括加密的数据密钥。
4. 密钥环使用其包装密钥以解密一个加密的数据密钥。如果成功，响应则包含明文数据密钥。如果密钥环或主密钥提供程序指定的包装密钥均无法解密加密数据密钥，则解密调用失败。
5. 解密方法使用明文数据密钥解密数据，丢弃明文数据密钥，然后返回明文数据。

中支持的算法套件 AWS Encryption SDK

算法套件 是一组加密算法和相关的值。加密系统使用算法实现生成密文消息。

该 AWS Encryption SDK 算法套件使用 Galois/Counter 模式下的高级加密标准 (AES) 算法 (GCM)，即 AES-GCM，对原始数据进行加密。 AWS Encryption SDK 支持 256 位、192 位和 128 位加密密钥。初始化向量 (IV) 长度始终为 12 字节。身份验证标签长度始终为 16 字节。

默认情况下， AWS Encryption SDK 使用带有 AES-GCM 的算法套件，该套件具有基于 HMAC 的 extract-and-expand密钥派生功能 (H KDF)、签名和 256 位加密密钥。如果承诺策略需要密钥承诺，则 AWS Encryption SDK 会选择同时支持密钥承诺的算法套件；否则，它会选择具有密钥派生和签名功能但不支持密钥承诺的算法套件。

建议：具有密钥派生、签名和密钥承诺的 AES-GCM

AWS Encryption SDK 建议使用一种算法套件，该套件通过向基于 HMAC 的密钥派生函数 (H extract-and-expand KDF) 提供 256 位的数据加密密钥来获得 AES-GCM 加密密钥。 AWS Encryption SDK 添加了椭圆曲线数字签名算法 (ECDSA) 签名。为了支持密钥承诺，该算法套件还派生了一个密钥承诺字符串（一个非机密数据密钥标识符），该字符串存储在加密消息的元数据中。此密钥承诺字符串同样参照类似于数据加密密钥派生过程通过 HKDF 进行派生。

AWS Encryption SDK 算法套件

加密算法	数据加密密钥长度（位）	密钥派生算法	签名算法	密钥承诺
AES-GCM	256	HKDF 以及 SHA-384	ECDSA 以及 P-384 和 SHA-384	HKDF 以及 SHA-512

HKDF 可以帮助您避免意外重用数据加密密钥，同时降低过度使用数据密钥的风险。

为了签名，该算法套件使用带加密哈希函数算法（SHA-384）的 ECDSA。默认情况下，将使用 ECDSA，即使基础主密钥的策略未指定该算法。消息签名验证消息发件人是否有权加密消息，同时具有不可否认性。如果主密钥的授权策略允许一组用户加密数据，并允许一组不同的用户解密数据，这是特别有用的。

具有密钥承诺的算法套件确保每个加密文字仅解密为一个明文。这些算法套件通过验证用作加密算法输入的数据密钥的身份达到上述目的。加密时，这些算法套件会派生密钥承诺字符串。在解密之前，这些算法套件会验证数据密钥是否与密钥承诺字符串匹配。如果不匹配，Decrypt 调用会失败。

其他支持的算法套件

AWS Encryption SDK 支持以下备用算法套件以实现向后兼容。通常，我们不建议使用这些算法套件。但是，我们认识到签名会严重影响性能，因此我们针对这些情况提供了具有密钥派生的密钥提交套件。对于必须进行更显著的性能权衡的应用程序，我们将继续提供缺少签名、密钥承诺和密钥派生的套件。

不具有密钥承诺的 AES-GCM

不具有密钥承诺的算法套件不会在解密之前验证数据密钥。因此，这些算法套件可能会将单个加密文字解密为不同的明文消息。但是，由于具有密钥承诺的算法套件会生成稍大（+30 字节）的加密消息，并且处理时间更长，因此可能并非各应用程序的最佳选择。

AWS Encryption SDK 支持具有密钥派生、密钥承诺、签名的算法套件，以及具有密钥派生和密钥承诺但不支持签名的算法套件。我们不建议使用不具有密钥承诺的算法套件。如果必须使用，我们建议您使用具有密钥派生和密钥承诺但不支持签名的算法套件。但是，如果您的应用程序性能配置文件支持使用算法套件，最佳实践是使用具有密钥承诺、密钥派生和签名的算法套件。

不具有签名的 AES-GCM

不具有签名的算法套件缺少具有真实性和不可否认性的 ECDSA 签名。如果同等信任加密和解密数据的用户，请仅使用此类套件。

如果使用不具有签名的算法套件，我们建议您选择具有密钥派生和密钥承诺的算法套件。

不具有密钥派生的 AES-GCM

不具有密钥派生的算法套件将数据加密密钥用作 AES-GCM 加密密钥，而非使用密钥派生函数派生唯一的密钥。我们不鼓励使用此套件生成密文，但出于兼容性考虑， AWS Encryption SDK 它们支持它。

有关如何在库中表示和使用这些套件的更多信息，请参阅the section called “算法参考”。

使用 wit AWS Encryption SDK h AWS KMS

要使用 AWS Encryption SDK，您需要为密钥环或主密钥提供程序配置包装密钥。如果您没有密钥基础设施，我们建议使用 AWS Key Management Service (AWS KMS)。中的许多代码示例都 AWS Encryption SDK 需要 AWS KMS key.

要与之交互 AWS KMS， AWS Encryption SDK 需要使用您的首选编程语言的 AWS SDK。 AWS Encryption SDK 客户端库与配合使用 AWS SDKs 以支持存储在中的主密钥 AWS KMS。

准备与 AWS Encryption SDK 一起使用 AWS KMS

1. 创建一个 AWS 账户. 要了解如何操作，请参阅如何创建和激活新的亚马逊 Web Services 账户？在 AWS 知识中心中。
2. 创建对称加密 AWS KMS key。有关帮助信息，请参阅《AWS Key Management Service 开发人员指南》中的创建密钥。

   > **Tip**
   >
   > 要 AWS KMS key 以编程方式使用，您需要的密钥 ID 或 Amazon 资源名称 (ARN)。AWS KMS key要获得有关查找 AWS KMS key ID 和 ARN 的帮助，请参阅《AWS Key Management Service 开发人员指南》中的查找密钥 ID 和 ARN。

3. 生成访问密钥 ID 和安全访问密钥。您可以使用 IAM 用户的访问密钥 ID 和私有访问密钥，也可以使用使用临时安全证书（包括访问密钥 ID、私有访问密钥和会话令牌）创建新会话。 AWS Security Token Service 作为安全最佳实践，我们建议您使用临时证书，而不是与您的 IAM 用户或 AWS （根）用户账户关联的长期证书。

   要创建具有访问密钥的 IAM 用户，请参阅《IAM 用户指南》中的创建 IAM 用户。

   要生成临时安全凭证，请参阅《IAM 用户指南》中的请求临时安全凭证。
4. 使用适用于 Java 的 AWS SDK适用于 JavaScript 的 AWS SDK、AWS SDK for Python (Boto)或适用于 C++ 的 AWS SDK（对于 C）中的说明以及您在步骤 3 中生成的访问密钥 ID 和私有访问密钥来设置您的 AWS 证书。如果您生成了临时凭证，还需要指定会话令牌。

   此过程 AWS SDKs 允许您签署对 AWS 的请求。中与之交互 AWS Encryption SDK 的代码示例 AWS KMS 假设您已完成此步骤。
5. 下载并安装 AWS Encryption SDK。要了解操作方法，请参阅要使用的编程语言的安装说明。

的最佳实践 AWS Encryption SDK

AWS Encryption SDK 旨在让您可以轻松地使用行业标准和最佳实践来保护您的数据。虽然在默认值中为您选择了许多最佳实践，但有些做法是可选的，建议在实际可行时使用。

使用最新版本

开始使用时 AWS Encryption SDK，请使用以您的首选编程语言提供的最新版本。如果您一直在使用 AWS Encryption SDK，请尽快升级到每个最新版本。这样可以确保您使用推荐的配置并利用新的安全属性来保护您的数据。有关支持的版本的详细信息，包括迁移和部署指南，请参阅 支持和维护 和 的版本 AWS Encryption SDK。

如果新版本弃用了您代码中的元素，请尽快将其替换。弃用警告和代码注释中通常会推荐不错的替代方案。

为了简化重大升级并减少出错的可能性，我们偶尔会提供临时或过渡版本。使用这些版本及其随附的文档，确保您可以在不中断生产工作流的情况下升级应用程序。

使用默认值

将最佳实践 AWS Encryption SDK 设计为其默认值。只要可行，就应使用默认值。对于默认值不可行的情况，我们提供替代方案，例如无需签名的算法套件。我们还为高级用户提供自定义机会，例如自定义密钥环、主密钥提供程序和加密材料管理器 () CMMs。请谨慎使用这些高级替代方案，并尽可能让安全工程师验证您的选择。

使用加密上下文

为了提高加密操作安全性，请在所有加密数据的请求中包含具有有意义的值的加密上下文。使用加密上下文是可选的，但这是我们建议遵守的加密最佳实践。加密上下文为 AWS Encryption SDK中的身份验证加密提供额外验证数据（AAD）。尽管这不是密钥，但加密上下文可以帮助您保护加密数据的完整性和真实性。

在中 AWS Encryption SDK，只有在加密时才能指定加密上下文。解密时， AWS Encryption SDK 使用返回的加密消息标头中的加密上下文。 AWS Encryption SDK 在应用程序返回明文数据之前，请验证在解密消息时使用的加密上下文中是否包含加密消息使用的加密上下文。有关详细信息，请参阅您的编程语言的示例。

当您使用命令行界面时，会为您 AWS Encryption SDK 验证加密上下文。

保护包装密钥

AWS Encryption SDK 生成一个唯一的数据密钥来加密每条纯文本消息。然后，使用您提供的包装密钥对数据密钥进行加密。如果您的包装密钥丢失或删除，则您的已加密数据将无法恢复。如果您的密钥不安全，您的数据可能会受到攻击。

使用受安全密钥基础设施保护的包装密钥，例如 AWS Key Management Service（AWS KMS）。使用原始 AES 或原始 RSA 密钥时，请使用符合安全要求的随机源和持久存储。在硬件安全模块 (HSM) 或提供的 HSMs服务（例如）中生成和存储包装密钥是一种最佳实践。 AWS CloudHSM

使用密钥基础设施的授权机制限制包装密钥的访问，只有需要的用户才能访问。实施最佳实践原则，例如最低权限。使用时 AWS KMS keys，请使用实施最佳实践原则的关键策略和 IAM 策略。

指定包装密钥

解密和加密时明确指定包装密钥始终是最佳实践。当你这样做时，只 AWS Encryption SDK 使用你指定的密钥。这种做法可确保仅使用想要的加密密钥。对于 AWS KMS 封装密钥，它还可以防止您无意中使用其他 AWS 账户 或地区的密钥，或者尝试使用您无权使用的密钥进行解密，从而提高性能。

加密时，提供的密钥环和主密钥提供程序要求您指定包装密钥。 AWS Encryption SDK 这些程序仅使用您指定的全部包装密钥。使用原始 AES 密钥环、原始 RSA 密钥环和密钥进行加密和解密时，您还需要指定包装密钥。 JCEMaster

但是，使用密 AWS KMS 钥环和主密钥提供程序进行解密时，您无需指定包装密钥。 AWS Encryption SDK 可以从加密数据密钥的元数据中获取密钥标识符。但是指定包装密钥是我们推荐的最佳实践。

为了在使用 AWS KMS 封装密钥时支持这一最佳实践，我们建议采取以下措施：

- 使用指定包装 AWS KMS 密钥的钥匙圈。加密和解密时，这些密钥环仅使用您指定的指定包装密钥。
- 使用 AWS KMS 主密钥和主密钥提供程序时，请使用版本 1.7 中引入的严格模式构造函数。 的 x 个 AWS Encryption SDK。这些函数创建的提供程序仅使用您指定的包装密钥进行加密和解密。始终使用任何包装密钥进行解密的主密钥提供程序的构造函数在版本 1.7.x 中被弃用，在版本 2.0.x 中被删除。

当指定用于解密的 AWS KMS 包装密钥不切实际时，您可以使用发现提供程序。 AWS Encryption SDK 在 C 语言中并 JavaScript 支持AWS KMS 发现密钥环。在版本 1.7.x 及更高版本中，具有发现模式的主密钥提供程序可用于 Java 和 Python。这些发现提供程序仅用于使用 AWS KMS 包装密钥进行解密，它们明确指示使用加密数据密钥的任何包装密钥。 AWS Encryption SDK

如果您必须使用发现提供程序，请使用其发现筛选条件功能来限制使用的包装密钥。例如，AWS KMS Regional Discovery 密钥环仅使用特定 AWS 区域中的包装密钥。您还可以将 AWS KMS 密钥环和 AWS KMS 主密钥提供程序配置为仅使用包装密钥。 AWS 账户此外，与往常一样，使用密钥策略和 IAM 策略来控制对 AWS KMS 包装密钥的访问。

使用数字签名

最佳实践是使用带签名的算法套件。数字签名可验证邮件发件人是否有权发送消息并保护消息的完整性。默认情况下，所有版本都 AWS Encryption SDK 使用带签名的算法套件。

如果您的安全要求不包括数字签名，则可以选择不带数字签名的算法套件。但是，我们建议使用数字签名，特别是当一组用户加密数据而另一组用户解密该数据时。

使用密钥承诺

最佳实践是使用密钥承诺安全功能。通过验证加密数据的唯一数据密钥的身份，密钥承诺可以防止您解密任何可能生成多条明文消息的加密文字。

从 2.0 版开始，通过密钥承诺 AWS Encryption SDK 提供对加密和解密的全面支持。 x。默认情况下，您的所有消息都使用密钥承诺进行加密和解密。版本 1.7。 x AWS Encryption SDK 可以用密钥承诺解密密文。其旨在帮助早期版本的用户成功部署版本 2.0.x。

对密钥承诺的支持包括新算法套件和新消息格式，该格式生成的加密文字仅比没有密钥承诺的加密文字大 30 字节。该设计最大限度减少了其对性能的影响，因此大多数用户都可以享受密钥承诺带来的好处。如果您的应用程序对大小和性能非常敏感，则可以决定使用承诺策略设置来禁用密钥承诺或允许在没有承诺的情况下解密消息，但只有在 AWS Encryption SDK 必须时才这样做。

限制加密的数据密钥的数量

最佳实践是在您解密的消息中限制加密的数据密钥的数量，尤其是来自不可信来源的消息。使用大量您无法解密的加密数据密钥来解密消息可能会导致延迟时间延长，增加开支，限制您的应用程序和其他共享您账户的应用程序，并可能耗尽密钥基础设施。在没有限制的情况下，加密消息最多可以有 65535（2^16 - 1）个加密数据密钥。有关更多信息，请参阅 限制加密数据密钥。

有关这些最佳实践所 AWS Encryption SDK 依据的安全功能的更多信息，请参阅AWS 安全博客中的改进客户端加密：显式 KeyIds 和密钥承诺。

正在配置 AWS Encryption SDK

AWS Encryption SDK 的设计便于使用。尽管 AWS Encryption SDK 有多个配置选项，但默认值是经过精心选择的，以便对大多数应用程序既实用又安全。但是，您可能需要调整配置以提高性能或在设计中加入自定义功能。

配置实施时，请查看 AWS Encryption SDK 最佳实践并尽可能多地实施。

主题

- 选择编程语言
- 选择包装密钥
- 使用多区域 AWS KMS keys
- 选择算法套件
- 限制加密数据密钥
- 创建发现筛选条件
- 配置所需的加密上下文 CMM
- 设置承诺策略
- 使用串流数据
- 缓存数据密钥

选择编程语言

AWS Encryption SDK 有多种编程语言版本。语言实现旨在实现完全互操作并提供相同的功能，尽管这些功能可能以不同的方式实现。通常，您使用与您的应用程序兼容的库。但是，您可以为特定的实现选择一种编程语言。例如，如果您更喜欢使用钥匙圈，则可以选择 AWS Encryption SDK for C 或。AWS Encryption SDK for JavaScript

选择包装密钥

AWS Encryption SDK 生成一个唯一的对称数据密钥来加密每条消息。除非使用数据密钥缓存，否则无需配置、管理或使用数据密钥。他们是为你 AWS Encryption SDK 做的。

但是，必须选择一个或多个包装密钥来加密每个数据密钥。 AWS Encryption SDK 支持不同大小的 AES 对称密钥和 RSA 非对称密钥。还支持 AWS Key Management Service（AWS KMS）对称加密

AWS KMS keys。您应对包装密钥的安全性和耐用性负责，因此我们建议您在硬件安全模块或密钥基础设施服务（例如）中使用加密密钥 AWS KMS。

要指定用于加密和解密的包装密钥，您可以使用密钥环（C、Java、、.NET 和 Python）或主密钥提供程序（Java JavaScript、Python、Encryption、Encryption AWS CLI）。您可以指定一个包装密钥或多个相同或不同类型的包装密钥。如果您使用多个包装密钥来包装一个数据密钥，则每个包装密钥将加密同一数据密钥的副本。加密的数据密钥（每个包装密钥一个）与加密数据一起存储在 AWS Encryption SDK 返回的加密消息中。要解密数据， AWS Encryption SDK 必须首先使用您的一个包装密钥来解密加密的数据密钥。

要 AWS KMS key 在密钥环或主密钥提供程序中指定，请使用支持的 AWS KMS 密钥标识符。有关密钥的密钥标识符的 AWS KMS 详细信息，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。

- 使用 AWS Encryption SDK for Java、 AWS Encryption SDK for JavaScript AWS Encryption SDK for Python、或加密 CLI 进行 AWS 加密时，您可以使用任何有效的密钥标识符（密钥 ID、密钥 ARN、别名或别名 ARN）作为 KMS 密钥。使用加密时 AWS Encryption SDK for C，您只能使用密钥 ID 或密钥 ARN。

  如果您在加密时为 KMS 密钥指定别名名称或别名 ARN，则 AWS Encryption SDK 会保存当前与该别名关联的密钥 ARN；但不会保存别名。对别名的更改不会影响用于解密数据密钥的 KMS 密钥。
- 在严格模式（指定特定的包装密钥）下解密时，必须使用密钥 ARN 来识别 AWS KMS keys。该要求适用于 AWS Encryption SDK的所有语言实施。

  使用密 AWS KMS 钥环加密时，会将的密钥 ARN AWS Encryption SDK 存储在加密数据密钥的元数据中。 AWS KMS key 在严格模式下解密时，在尝试使用包装密钥解密加密的数据密钥之前，会 AWS Encryption SDK 验证密钥环（或主密钥提供程序）中是否出现相同的密钥 ARN。如果您使用不同的密钥标识符，则即使标识符引用相同的密钥 AWS KMS key，也不会识别或使用。 AWS Encryption SDK

要将原始 AES 密钥或原始 RSA 密钥对指定为密钥环中的包装密钥，必须指定命名空间和名称。在主密钥提供程序中，`Provider ID` 等同于命名空间，`Key ID` 等同于名称。解密时，必须为每个原始包装密钥使用与加密时完全相同的命名空间和名称。如果您使用不同的命名空间或名称，即使密钥材料相同，也 AWS Encryption SDK 不会识别或使用包装密钥。

使用多区域 AWS KMS keys

您可以在中使用 AWS Key Management Service (AWS KMS) 多区域密钥作为包装密钥。 AWS Encryption SDK如果您使用多区域密钥合而为一进行加密 AWS 区域，则可以使用其他密钥中的相关多区域密钥进行解密。 AWS 区域版本 2.3 中引入了对多区域密钥的支持。 AWS Encryption SDK 和版本 3.0 中的 x。 AWS 加密 CLI 中的 x。

AWS KMS 多区域密钥是一组具有相同密钥材料和密钥 ID 的不同 AWS 区域 密钥。 AWS KMS keys 您可以像在不同区域使用相同的密钥一样使用这些相关密钥。多区域密钥支持常见的灾难恢复和备份场景，这些场景要求在一个区域进行加密，并在另一个区域进行解密，而无需进行跨区域调用。 AWS KMS有关多区域密钥的信息，请参阅《AWS Key Management Service 开发人员指南》中的使用多区域密钥。

为了支持多区域密钥， AWS Encryption SDK 包括支持 AWS KMS 多区域的密钥环和主密钥提供程序。每种编程语言中的新 multi-Region-aware符号都支持单区域和多区域密钥。

- 对于单区域密钥，该 multi-Region-aware符号的行为就像单区域密 AWS KMS 钥环和主密钥提供程序一样。该密钥尝试仅使用加密数据的单区域密钥来解密加密文字。
- 对于多区域密钥，该 multi-Region-aware符号尝试使用加密数据的相同多区域密钥或您指定的区域中的相关多区域副本密钥来解密密文。

在使用多个 multi-Region-aware KMS 密钥的密钥环和主密钥提供程序中，您可以指定多个单区域和多区域密钥。但是，您只能从每组相关的多区域副本密钥中指定一个密钥。如果您使用相同的密钥 ID 指定多个密钥标识符，则构造函数调用将失败。

您还可以将多区域密钥与标准、单区域密钥 AWS KMS 环和主密钥提供程序一起使用。但是，您必须在同一区域使用相同的多区域密钥进行加密和解密。单区域密钥环和主密钥提供程序尝试仅使用加密数据的密钥来解密加密文字。

以下示例说明如何使用多区域密钥以及新的密钥 multi-Region-aware环和主密钥提供程序来加密和解密数据。这些示例使用每个`us-east-1`区域中的相关多区域副本密钥加密`us-west-2`区域中的数据并解密该区域中的数据。在运行这些示例之前，请将示例多区域密钥 ARN 替换为您的 AWS 账户中的有效值。

C

要使用多区域密钥进行加密，请使用 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 方法实例化密钥环。指定多区域密钥。

这个简单的示例不包括加密上下文。有关在 C 语言中使用加密上下文的示例，请参阅 加密和解密字符串。

有关完整示例，请参阅上 AWS Encryption SDK for C 存储库中的 kms_multi_region_keys.cpp GitHub。

```
/* Encrypt with a multi-Region KMS key in us-east-1 */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Initialize a multi-Region keyring */
const char *mrk_us_east_1 = "arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder().Build(mrk_us_east_1);

/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
 AWS_CRYPTOSDK_ENCRYPT, mrk_keyring);

aws_cryptosdk_keyring_release(mrk_keyring);

/* Encrypt the data
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
    session, ciphertext, ciphertext_buf_sz, &ciphertext_len, plaintext,
 plaintext_len));

/* Clean up the session */
aws_cryptosdk_session_destroy(session);
```

C# / .NET

要使用美国东部（弗吉尼亚北部）(us-east-1) 区域的多区域密钥进行加密，请使用多区域密钥的密钥标识和指定区域的客户端实例化`CreateAwsKmsMrkKeyringInput`一个对象。 AWS KMS 然后使用 `CreateAwsKmsMrkKeyring()` 方法创建密钥环。

`CreateAwsKmsMrkKeyring()` 方法创建的密钥环只包含一个多区域密钥。要使用多个包装密钥（包括多区域密钥）进行加密，请使用 `CreateAwsKmsMrkMultiKeyring()` 方法。

有关完整的示例，请参阅 AwsKmsMrkKeyringExamplefor .NET 存储库中的 AWS Encryption SDK .cs。 GitHub

```
//Encrypt with a multi-Region KMS key in us-east-1 Region

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();


// Multi-Region keys have a distinctive key ID that begins with 'mrk'
// Specify a multi-Region key in us-east-1
string mrkUSEast1 = "arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

// Create the keyring
// You can specify the Region or get the Region from the key ARN
var createMrkEncryptKeyringInput = new CreateAwsKmsMrkKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USEast1),
    KmsKeyId = mrkUSEast1
};
var mrkEncryptKeyring =
 materialProviders.CreateAwsKmsMrkKeyring(createMrkEncryptKeyringInput);

// Define the encryption context
var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"}
};

// Encrypt your plaintext data.
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = mrkEncryptKeyring,
    EncryptionContext = encryptionContext
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

AWS Encryption CLI

此示例对 us-east-1 区域中的多区域密钥下的 `hello.txt` 文件进行加密。由于此示例指定了带有区域元素的密钥 ARN，所以其中不使用 `--wrapping-keys` 参数的 region 属性。

当包装密钥的密钥 ID 未指定区域时，您可以使用 `--wrapping-keys` 的 region 属性来指定区域，例如 `--wrapping-keys key=$keyID region=us-east-1`。

```
# Encrypt with a multi-Region KMS key in us-east-1 Region

# To run this example, replace the fictitious key ARN with a valid value.
$ mrkUSEast1=arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab

$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$mrkUSEast1 \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .
```

Java

要使用多区域密钥加密，请实例化 `AwsKmsMrkAwareMasterKeyProvider` 并指定多区域密钥。

有关完整的示例，请参阅 AWS Encryption SDK for Java 存储库BasicMultiRegionKeyEncryptionExample.java中的 GitHub。

```
//Encrypt with a multi-Region KMS key in us-east-1 Region

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// Multi-Region keys have a distinctive key ID that begins with 'mrk'
// Specify a multi-Region key in us-east-1
final String mrkUSEast1 = "arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

// Instantiate an AWS KMS master key provider in strict mode for multi-Region keys
// Configure it to encrypt with the multi-Region key in us-east-1
```

```
final AwsKmsMrkAwareMasterKeyProvider kmsMrkProvider =
 AwsKmsMrkAwareMasterKeyProvider
    .builder()
    .buildStrict(mrkUSEast1);

// Create an encryption context
final Map<String, String> encryptionContext = Collections.singletonMap("Purpose",
 "Test");

// Encrypt your plaintext data
final CryptoResult<byte[], AwsKmsMrkAwareMasterKey> encryptResult =
 crypto.encryptData(
    kmsMrkProvider,
    encryptionContext,
    sourcePlaintext);
byte[] ciphertext = encryptResult.getResult();
```

JavaScript Browser

要使用多区域密钥进行加密，请使用 `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` 方法创建密钥环并指定多区域密钥。

有关完整的示例，请参阅上存储库中的 kms_multi_region_simple.ts。 AWS Encryption SDK for JavaScript GitHub

```
/* Encrypt with a multi-Region KMS key in us-east-1 Region */

import {
  buildAwsKmsMrkAwareStrictMultiKeyringBrowser,
  buildClient,
  CommitmentPolicy,
  KMS,
} from '@aws-crypto/client-browser'


/* Instantiate an AWS Encryption SDK client */
const { encrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

declare const credentials: {
  accessKeyId: string
```

```
    secretAccessKey: string
    sessionToken: string
  }

  /* Instantiate an AWS KMS client
   * The AWS Encryption SDK for JavaScript gets the Region from the key ARN
   */
  const clientProvider = (region: string) => new KMS({ region, credentials })


  /* Specify a multi-Region key in us-east-1 */
  const multiRegionUsEastKey =
      'arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

  /* Instantiate the keyring */
  const encryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringBrowser({
      generatorKeyId: multiRegionUsEastKey,
      clientProvider,
    })


  /* Set the encryption context */
  const context = {
      purpose: 'test',
    }

  /* Test data to encrypt */
  const cleartext = new Uint8Array([1, 2, 3, 4, 5])

  /* Encrypt the data */
  const { result } = await encrypt(encryptKeyring, cleartext, {
      encryptionContext: context,
    })
```

JavaScript Node.js

要使用多区域密钥进行加密，请使用 `buildAwsKmsMrkAwareStrictMultiKeyringNode()` 方法创建密钥环并指定多区域密钥。

有关完整的示例，请参阅上存储库中的 kms_multi_region_simple.ts。 AWS Encryption SDK for JavaScript GitHub

```
//Encrypt with a multi-Region KMS key in us-east-1 Region
```

```
import { buildClient } from '@aws-crypto/client-node'

/* Instantiate the AWS Encryption SDK client
const { encrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* Test string to encrypt */
const cleartext = 'asdf'

/* Multi-Region keys have a distinctive key ID that begins with 'mrk'
 * Specify a multi-Region key in us-east-1
 */
const multiRegionUsEastKey =
    'arn:aws:kms:us-east-1:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Create an AWS KMS keyring */
const mrkEncryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringNode({
    generatorKeyId: multiRegionUsEastKey,
  })

/* Specify an encryption context */
const context = {
    purpose: 'test',
  }

/* Create an encryption keyring */
const { result } = await encrypt(mrkEncryptKeyring, cleartext, {
    encryptionContext: context,
  })
```

Python

要使用 AWS KMS 多区域密钥进行加密，请使用`MRKAwareStrictAwsKmsMasterKeyProvider()`方法并指定多区域密钥。

有关完整示例，请参阅上 AWS Encryption SDK for Python 存储库中的 mrk_aware_kms_provider.py GitHub。

```
* Encrypt with a multi-Region KMS key in us-east-1 Region

# Instantiate the client
```

```
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R

# Specify a multi-Region key in us-east-1
mrk_us_east_1 = "arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab"

# Use the multi-Region method to create the master key provider
# in strict mode
strict_mrk_key_provider = MRKAwareStrictAwsKmsMasterKeyProvider(
        key_ids=[mrk_us_east_1]
)

# Set the encryption context
encryption_context = {
        "purpose": "test"
    }

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
        source=source_plaintext,
        encryption_context=encryption_context,
        key_provider=strict_mrk_key_provider
)
```

接下来，将您的加密文字移至 `us-west-2` 区域。您无需重新加密加密文字。

要在区域中以严格模式解密密文，请使用该`us-west-2`区域中相关多区域密钥 multi-Region-aware的密钥 ARN 来实例化符号。`us-west-2`如果您在其他区域（包括加密该密钥的地方）中指定相关多区域密钥的密钥 ARN`us-east-1`，则该 multi-Region-aware符号将为此进行跨区域调用。 AWS KMS key

在严格模式下解密时， multi-Region-aware符号需要密钥 ARN。仅接受每组相关的多区域密钥中的一个密钥 ARN。

在运行这些示例之前，请将示例多区域密钥 ARN 替换为您的有效值。 AWS 账户

C

要在严格模式下使用多区域密钥进行解密，请使用 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 方法实例化密钥环。在本地（us-west-2）区域中指定相关的多区域密钥。

有关完整示例，请参阅上 AWS Encryption SDK for C 存储库中的 kms_multi_region_keys.cpp GitHub。

```
/* Decrypt with a related multi-Region KMS key in us-west-2 Region */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Initialize a multi-Region keyring */
const char *mrk_us_west_2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder().Build(mrk_us_west_2);

/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
 AWS_CRYPTOSDK_ENCRYPT, mrk_keyring);

aws_cryptosdk_session_set_commitment_policy(session,
    COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT);

aws_cryptosdk_keyring_release(mrk_keyring);

/* Decrypt the ciphertext
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
    session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
 ciphertext_len));

/* Clean up the session */
aws_cryptosdk_session_destroy(session);
```

C# / .NET

要在严格模式下使用单个多区域密钥进行解密，请使用与组装输入和创建用于加密的密钥环相同的构造函数和方法。使用相关多区域密钥的密钥 ARN 和美国西部（俄勒冈）(us-west-2) 区域的 AWS KMS 客户机实例化`CreateAwsKmsMrkKeyringInput`对象。然后使用`CreateAwsKmsMrkKeyring()`方法通过一个多区域 KMS 密钥创建多区域密钥环。

有关完整的示例，请参阅 AwsKmsMrkKeyringExamplefor .NET 存储库中的 AWS Encryption SDK .cs。 GitHub

```
// Decrypt with a related multi-Region KMS key in us-west-2 Region

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Specify the key ARN of the multi-Region key in us-west-2
string mrkUSWest2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

// Instantiate the keyring input
// You can specify the Region or get the Region from the key ARN
var createMrkDecryptKeyringInput = new CreateAwsKmsMrkKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    KmsKeyId = mrkUSWest2
};

// Create the multi-Region keyring
var mrkDecryptKeyring =
 materialProviders.CreateAwsKmsMrkKeyring(createMrkDecryptKeyringInput);

// Decrypt the ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = mrkDecryptKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

AWS Encryption CLI

要使用 us-west-2 区域中的相关多区域密钥进行解密，请使用 `--wrapping-keys` 参数的 key 属性来指定其密钥 ARN。

```
# Decrypt with a related multi-Region KMS key in us-west-2 Region

# To run this example, replace the fictitious key ARN with a valid value.
```

```
$ mrkUSWest2=arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab

$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$mrkUSWest2 \
                     --commitment-policy require-encrypt-require-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output .
```

Java

要在严格模式下解密，请实例化 `AwsKmsMrkAwareMasterKeyProvider` 并在本地（us-west-2）区域中指定相关的多区域密钥。

有关完整示例，请参阅上 AWS Encryption SDK for Java GitHub存储库中的 BasicMultiRegionKeyEncryptionExample.java。

```
// Decrypt with a related multi-Region KMS key in us-west-2 Region

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

// Related multi-Region keys have the same key ID. Their key ARNs differs only in
 the Region field.
String mrkUSWest2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

// Use the multi-Region method to create the master key provider
// in strict mode
AwsKmsMrkAwareMasterKeyProvider kmsMrkProvider =
 AwsKmsMrkAwareMasterKeyProvider.builder()
    .buildStrict(mrkUSWest2);

// Decrypt your ciphertext
CryptoResult<byte[], AwsKmsMrkAwareMasterKey> decryptResult = crypto.decryptData(
        kmsMrkProvider,
        ciphertext);
```

```
byte[] decrypted = decryptResult.getResult();
```

JavaScript Browser

要在严格模式下解密，请使用 `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` 方法创建密钥环并在本地（us-west-2）区域中指定相关的多区域密钥。

有关完整的示例，请参阅上存储库中的 [kms_multi_region_simple.ts。](https://github.com/aws/aws-encryption-sdk-javascript) AWS Encryption SDK for JavaScript GitHub

```
/* Decrypt with a related multi-Region KMS key in us-west-2 Region */

import {
  buildAwsKmsMrkAwareStrictMultiKeyringBrowser,
  buildClient,
  CommitmentPolicy,
  KMS,
} from '@aws-crypto/client-browser'


/* Instantiate an AWS Encryption SDK client */
const { decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* Instantiate an AWS KMS client
 * The AWS Encryption SDK for JavaScript gets the Region from the key ARN
 */
const clientProvider = (region: string) => new KMS({ region, credentials })


/* Specify a multi-Region key in us-west-2 */
const multiRegionUsWestKey =
    'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Instantiate the keyring */
const mrkDecryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringBrowser({
```

```
    generatorKeyId: multiRegionUsWestKey,
    clientProvider,
  })


/* Decrypt the data */
const { plaintext, messageHeader } = await decrypt(mrkDecryptKeyring, result)
```

JavaScript Node.js

要在严格模式下解密，请使用 `buildAwsKmsMrkAwareStrictMultiKeyringNode()` 方法创建密钥环并在本地（us-west-2）区域中指定相关的多区域密钥。

有关完整的示例，请参阅上存储库中的 [kms_multi_region_simple.ts。](kms_multi_region_simple.ts) AWS Encryption SDK for JavaScript GitHub

```
/* Decrypt with a related multi-Region KMS key in us-west-2 Region */

import { buildClient } from '@aws-crypto/client-node'

/* Instantiate the client
const { decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* Multi-Region keys have a distinctive key ID that begins with 'mrk'
 * Specify a multi-Region key in us-west-2
 */
const multiRegionUsWestKey =
    'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

/* Create an AWS KMS keyring */
const mrkDecryptKeyring = buildAwsKmsMrkAwareStrictMultiKeyringNode({
    generatorKeyId: multiRegionUsWestKey,
  })

/* Decrypt your ciphertext */
const { plaintext, messageHeader } = await decrypt(decryptKeyring, result)
```

Python

要在严格模式下解密，请使用 `MRKAwareStrictAwsKmsMasterKeyProvider()` 方法创建主密钥提供程序。在本地（us-west-2）区域中指定相关的多区域密钥。

有关完整示例，请参阅上 AWS Encryption SDK for Python 存储库中的 mrk_aware_kms_provider.py GitHub。

```
# Decrypt with a related multi-Region KMS key in us-west-2 Region

# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R

# Related multi-Region keys have the same key ID. Their key ARNs differs only in the
 Region field
mrk_us_west_2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab"

# Use the multi-Region method to create the master key provider
# in strict mode
strict_mrk_key_provider = MRKAwareStrictAwsKmsMasterKeyProvider(
        key_ids=[mrk_us_west_2]
)

# Decrypt your ciphertext
plaintext, _ = client.decrypt(
        source=ciphertext,
        key_provider=strict_mrk_key_provider
)
```

您还可以在发现模式下使用 AWS KMS 多区域密钥进行解密。在发现模式下解密时，不需指定任何 AWS KMS keys。（有关单区域 AWS KMS 发现密钥环的信息，请参阅使用 AWS KMS 发现密钥环。）

如果您使用多区域密钥加密，则发现模式下的 multi-Region-aware符号将尝试使用本地区域中的相关多区域密钥进行解密。如果不存在，则调用失败。在发现模式下， AWS Encryption SDK 不会尝试跨区域调用用于加密的多区域密钥。

> **Note**
>
> 如果您在发现模式下使用 multi-Region-aware符号来加密数据，则加密操作将失败。

以下示例说明如何在发现模式下使用 multi-Region-aware符号进行解密。由于您未指定 AWS KMS key，因此 AWS Encryption SDK 必须从其他来源获取区域。如果可能，请明确指定本地区域。否则，将从 AWS SDK 中为您的编程语言配置的区域 AWS Encryption SDK 获取本地区域。

在运行这些示例之前，请将示例账户 ID 和多区域密钥 ARN 替换为您的有效值。 AWS 账户

C

要使用多区域密钥在发现模式下解密，请使用 `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` 方法构建密钥环，使用 `Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()` 方法构建发现筛选条件。要指定本地区域，请定义 `ClientConfiguration` 并在 AWS KMS 客户端中指定。

有关完整示例，请参阅上 AWS Encryption SDK for C 存储库中的 kms_multi_region_keys.cpp GitHub。

```
/* Decrypt in discovery mode with a multi-Region KMS key */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct a discovery filter for the account and partition. The
 *  filter is optional, but it's a best practice that we recommend.
 */
const char *account_id = "111122223333";
const char *partition = "aws";
const std::shared_ptr<Aws::Cryptosdk::KmsKeyring::DiscoveryFilter> discovery_filter
 =

 Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Buil

/* Create an AWS KMS client in the desired region. */
const char *region = "us-west-2";

Aws::Client::ClientConfiguration client_config;
client_config.region = region;
const std::shared_ptr<Aws::KMS::KMSClient> kms_client =
    Aws::MakeShared<Aws::KMS::KMSClient>("AWS_SAMPLE_CODE", client_config);

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()
        .WithKmsClient(kms_client)
        .BuildDiscovery(region, discovery_filter);
```

```
/* Create a session; release the keyring */
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(aws_default_allocator(),
 AWS_CRYPTOSDK_DECRYPT, mrk_keyring);

aws_cryptosdk_keyring_release(mrk_keyring);
commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
/* Decrypt the ciphertext
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
    session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
 ciphertext_len));

/* Clean up the session */
aws_cryptosdk_session_destroy(session);
```

C# / .NET

要在 for .NET 中创建 multi-Region-aware发现密钥环，请 AWS Encryption SDK 实例化一个`CreateAwsKmsMrkDiscoveryKeyringInput`对象，该对象使用特定的 AWS KMS 客户端 AWS 区域，以及一个可选的发现过滤器，将 KMS 密钥限制在特定的 AWS 分区和帐户。然后通过输入对象调用 `CreateAwsKmsMrkDiscoveryKeyring()` 方法。有关完整的示例，请参阅 AwsKmsMrkDiscoveryKeyringExamplefor .NET 存储库中的 AWS Encryption SDK .cs。 GitHub

要为多个密钥环创建 multi-Region-aware发现密钥环 AWS 区域，请使用`CreateAwsKmsMrkDiscoveryMultiKeyring()`方法创建多密钥环，或者使用`CreateAwsKmsMrkDiscoveryKeyring()`创建多个 multi-Region-aware发现密钥环，然后使用该`CreateMultiKeyring()`方法将它们组合成一个多密钥环。

有关示例，请参见 AwsKmsMrkDiscoveryMultiKeyringExample.cs。

```
// Decrypt in discovery mode with a multi-Region KMS key

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

List<string> account = new List<string> { "111122223333" };
```

```
// Instantiate the discovery filter
DiscoveryFilter mrkDiscoveryFilter = new DiscoveryFilter()
{
    AccountIds = account,
    Partition = "aws"
}

// Create the keyring
var createMrkDiscoveryKeyringInput = new CreateAwsKmsMrkDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    DiscoveryFilter = mrkDiscoveryFilter
};
var mrkDiscoveryKeyring =
 materialProviders.CreateAwsKmsMrkDiscoveryKeyring(createMrkDiscoveryKeyringInput);


// Decrypt the ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = mrkDiscoveryKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

AWS Encryption CLI

要在发现模式下解密，请使用 `--wrapping-keys` 参数的 discovery 属性。discovery-account 和 discovery-partition 属性创建了一个发现筛选条件，该筛选条件是可选的，但建议使用。

要指定区域，此命令包括 `--wrapping-keys` 参数的 region 属性。

```
# Decrypt in discovery mode with a multi-Region KMS key

$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-account=111122223333 \
                                     discovery-partition=aws \
                                     region=us-west-2 \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
```

```
                      --output .
```

Java

要指定本地区域，请使用 `builder().withDiscoveryMrkRegion` 参数。否则， AWS Encryption SDK 将从 [适用于 Java 的 AWS SDK](#) 中配置的区域获取本地区域。

有关完整示例，请参阅上 AWS Encryption SDK for Java GitHub存储库中的 [DiscoveryMultiRegionDecryptionExample.java。](#)

```
// Decrypt in discovery mode with a multi-Region KMS key

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

DiscoveryFilter discoveryFilter = new DiscoveryFilter("aws", 111122223333);

AwsKmsMrkAwareMasterKeyProvider mrkDiscoveryProvider =
 AwsKmsMrkAwareMasterKeyProvider
    .builder()
    .withDiscoveryMrkRegion(Region.US_WEST_2)
    .buildDiscovery(discoveryFilter);

// Decrypt your ciphertext
final CryptoResult<byte[], AwsKmsMrkAwareMasterKey> decryptResult = crypto
    .decryptData(mrkDiscoveryProvider, ciphertext);
```

JavaScript Browser

要使用对称多区域密钥在发现模式下解密，请使用 `AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()` 方法。

有关完整的示例，请参阅上存储库中的 [kms_multi_region_discovery.ts。](#) AWS Encryption SDK for JavaScript GitHub

```
/* Decrypt in discovery mode with a multi-Region KMS key */

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser,
  buildClient,
  CommitmentPolicy,
```

```
  KMS,
} from '@aws-crypto/client-browser'


/* Instantiate an AWS Encryption SDK client */
const { decrypt } = buildClient()

declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* Instantiate the KMS client with an explicit Region */
const client = new KMS({ region: 'us-west-2', credentials })


/* Create a discovery filter */
const discoveryFilter = { partition: 'aws', accountIDs: ['111122223333'] }


/* Create an AWS KMS discovery keyring */
const mrkDiscoveryKeyring = new AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser({
    client,
    discoveryFilter,
  })


/* Decrypt the data */
const { plaintext, messageHeader } = await decrypt(mrkDiscoveryKeyring, ciphertext)
```

JavaScript Node.js

要使用对称多区域密钥在发现模式下解密，请使用 `AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()` 方法。

有关完整的示例，请参阅上存储库中的 kms_multi_region_discovery.ts。 AWS Encryption SDK for JavaScript GitHub

```
/* Decrypt in discovery mode with a multi-Region KMS key */

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringNode,
```

```
  buildClient,
  CommitmentPolicy,
  KMS,
} from '@aws-crypto/client-node'

/* Instantiate the Encryption SDK client
const { decrypt } = buildClient()

/* Instantiate the KMS client with an explicit Region */
const client = new KMS({ region: 'us-west-2' })

/* Create a discovery filter */
const discoveryFilter = { partition: 'aws', accountIDs: ['111122223333'] }

/* Create an AWS KMS discovery keyring */
const mrkDiscoveryKeyring = new AwsKmsMrkAwareSymmetricDiscoveryKeyringNode({
    client,
    discoveryFilter,
  })

/* Decrypt your ciphertext */
const { plaintext, messageHeader } = await decrypt(mrkDiscoveryKeyring, result)
```

Python

要使用多区域密钥在发现模式下解密，请使用 `MRKAwareDiscoveryAwsKmsMasterKeyProvider()` 方法。

有关完整示例，请参阅上 AWS Encryption SDK for Python 存储库中的 mrk_aware_kms_provider.py GitHub。

```
# Decrypt in discovery mode with a multi-Region KMS key

# Instantiate the client
client = aws_encryption_sdk.EncryptionSDKClient()

# Create the discovery filter and specify the region
decrypt_kwargs = dict(
        discovery_filter=DiscoveryFilter(account_ids="111122223333",
 partition="aws"),
        discovery_region="us-west-2",
    )
```

```
# Use the multi-Region method to create the master key provider
# in discovery mode
mrk_discovery_key_provider =
 MRKAwareDiscoveryAwsKmsMasterKeyProvider(**decrypt_kwargs)

# Decrypt your ciphertext
plaintext, _ = client.decrypt(
        source=ciphertext,
        key_provider=mrk_discovery_key_provider
)
```

选择算法套件

AWS Encryption SDK 支持多种对称和非对称加密算法，用于在您指定的包装密钥下对数据密钥进行加密。但是，当它使用这些数据密钥来加密您的数据时， AWS Encryption SDK 默认使用推荐的算法套件，该套件使用带有密钥派生、数字签名和密钥承诺的 AES-GCM 算法。尽管默认算法套件可能适用于大多数应用程序，但您可以选择备用算法套件。例如，没有数字签名的算法套件可以满足某些信任模型的需求。有关 AWS Encryption SDK 支持的算法套件的信息，请参阅 中支持的算法套件 AWS Encryption SDK。

以下示例向您显示在加密时如何选择备用算法套件。这些示例选择了推荐的 AES-GCM 算法套件，该套件具有密钥派生和密钥承诺，但没有数字签名。使用不包含数字签名的算法套件进行加密时，请在解密时使用仅限未签名的解密模式。这种模式在流式传输解密时最有用，如果遇到签名的加密文字，则会失败。

C

要在中指定备用算法套件 AWS Encryption SDK for C，必须明确创建 CMM。然后将 `aws_cryptosdk_default_cmm_set_alg_id` 与 CMM 和选定的算法套件一起使用。

```
/* Specify an algorithm suite without signing */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* To set an alternate algorithm suite, create an cryptographic
```

```
   materials manager (CMM) explicitly
 */
struct aws_cryptosdk_cmm *cmm =
 aws_cryptosdk_default_cmm_new(aws_default_allocator(), kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

/* Specify the algorithm suite for the CMM */
aws_cryptosdk_default_cmm_set_alg_id(cmm, ALG_AES256_GCM_HKDF_SHA512_COMMIT_KEY);

/* Construct the session with the CMM,
   then release the CMM reference
 */
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm_2(alloc,
 AWS_CRYPTOSDK_ENCRYPT, cmm);
aws_cryptosdk_cmm_release(cmm);

/* Encrypt the data
   Use aws_cryptosdk_session_process_full with non-streaming data
 */
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(
                          session,
                          ciphertext,
                          ciphertext_buf_sz,
                          &ciphertext_len,
                          plaintext,
                          plaintext_len)) {
    aws_cryptosdk_session_destroy(session);
    return AWS_OP_ERR;
}
```

解密未经数字签名加密的数据时，请使用 AWS_CRYPTOSDK_DECRYPT_UNSIGNED。如果遇到签名的加密文字，这会导致解密失败。

```
/* Decrypt unsigned streaming data */

/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* Create a session for decrypting with the AWS KMS keyring
```

```
   Then release the keyring reference
 */
struct aws_cryptosdk_session *session =

 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT_UNSIGNED,
 kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

if (!session) {
    return AWS_OP_ERR;
}

/* Limit encrypted data keys */
aws_cryptosdk_session_set_max_encrypted_data_keys(session, 1);

/* Decrypt
   Use aws_cryptosdk_session_process_full with non-streaming data
 */
    if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(
                              session,
                              plaintext,
                              plaintext_buf_sz,
                              &plaintext_len,
                              ciphertext,
                              ciphertext_len)) {
    aws_cryptosdk_session_destroy(session);
    return AWS_OP_ERR;
}
```

C# / .NET

要在中 AWS Encryption SDK 为.NET 指定替代算法套件，请指定EncryptInput对象的`AlgorithmSuiteId`属性。f AWS Encryption SDK or .NET 包含可用于标识首选算法套件的常量。

f AWS Encryption SDK or .NET 没有在流式解密时检测签名密文的方法，因为此库不支持流式传输数据。

```
// Specify an algorithm suite without signing

// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =
```

```
    AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Create the keyring
var keyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(keyringInput);

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    AlgorithmSuiteId = AlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

AWS Encryption CLI

加密 `hello.txt` 文件时，此示例使用 `--algorithm` 参数来指定不带数字签名的算法套件。

```
# Specify an algorithm suite without signing

# To run this example, replace the fictitious key ARN with a valid value.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --algorithm AES_256_GCM_HKDF_SHA512_COMMIT_KEY \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --commitment-policy require-encrypt-require-decrypt \
                     --output hello.txt.encrypted \
                     --decode
```

解密时，此示例使用 `--decrypt-unsigned` 参数。建议使用此参数来确保您正在解密未签名的加密文字，尤其是使用 CLI，该工具始终流式传输输入和输出。

```
# Decrypt unsigned streaming data
```

```
# To run this example, replace the fictitious key ARN with a valid value.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --decrypt-unsigned \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --max-encrypted-data-keys 1 \
                     --commitment-policy require-encrypt-require-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

Java

要指定备用算法套件，请使用 `AwsCrypto.builder().withEncryptionAlgorithm()` 方法。此示例指定了不带数字签名的备用算法套件。

```
// Specify an algorithm suite without signing

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
    .build();

String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a master key provider in strict mode
KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Create an encryption context to identify this ciphertext
        Map<String, String> encryptionContext = Collections.singletonMap("Example",
 "FileStreaming");

// Encrypt your plaintext data
CryptoResult<byte[], KmsMasterKey> encryptResult = crypto.encryptData(
    masterKeyProvider,
    sourcePlaintext,
    encryptionContext);
byte[] ciphertext = encryptResult.getResult();
```

在流式传输数据进行解密时，请使用 createUnsignedMessageDecryptingStream() 方法确保您正在解密的所有加密文字均未签名。

```
// Decrypt unsigned streaming data

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
  .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
  .withMaxEncryptedDataKeys(1)
  .build();

// Create a master key provider in strict mode
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
  .buildStrict(awsKmsKey);

// Decrypt the encrypted message
FileInputStream in = new FileInputStream(srcFile + ".encrypted");
CryptoInputStream<KmsMasterKey> decryptingStream =
 crypto.createUnsignedMessageDecryptingStream(masterKeyProvider, in);

// Return the plaintext data
// Write the plaintext data to disk
FileOutputStream out = new FileOutputStream(srcFile + ".decrypted");
IOUtils.copy(decryptingStream, out);
decryptingStream.close();
```

JavaScript Browser

要指定备用算法套件，请使用带有 AlgorithmSuiteIdentifier 枚举值的 suiteId 参数。

```
// Specify an algorithm suite without signing

// Instantiate the client
const { encrypt } = buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Specify a KMS key
const generatorKeyId = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a keyring with the KMS key
const keyring = new KmsKeyringBrowser({ generatorKeyId })
```

```
// Encrypt your plaintext data
const { result } = await encrypt(keyring, cleartext, { suiteId:
 AlgorithmSuiteIdentifier.ALG_AES256_GCM_IV12_TAG16_HKDF_SHA512_COMMIT_KEY,
 encryptionContext: context, })
```

解密时，请使用标准 `decrypt` 方法。 AWS Encryption SDK for JavaScript 在浏览器中没有 `decrypt-unsigned` 模式，因为浏览器不支持串流。

```
// Decrypt unsigned streaming data

// Instantiate the client
const { decrypt } = buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Create a keyring with the same KMS key used to encrypt
const generatorKeyId = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
const keyring = new KmsKeyringBrowser({ generatorKeyId })

// Decrypt the encrypted message
const { plaintext, messageHeader } = await decrypt(keyring, ciphertextMessage)
```

JavaScript Node.js

要指定备用算法套件，请使用带有 `AlgorithmSuiteIdentifier` 枚举值的 `suiteId` 参数。

```
// Specify an algorithm suite without signing

// Instantiate the client
const { encrypt } = buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Specify a KMS key
const generatorKeyId = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Create a keyring with the KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })

// Encrypt your plaintext data
const { result } = await encrypt(keyring, cleartext, { suiteId:
 AlgorithmSuiteIdentifier.ALG_AES256_GCM_IV12_TAG16_HKDF_SHA512_COMMIT_KEY,
 encryptionContext: context, })
```

解密未经数字签名的加密数据时，请使用 decryptUnsignedMessage Stream。如果遇到签名的加密文字，此方法会失败。

```
// Decrypt unsigned streaming data

// Instantiate the client
const { decryptUnsignedMessageStream } =
 buildClient( CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT )

// Create a keyring with the same KMS key used to encrypt
const generatorKeyId = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
const keyring = new KmsKeyringNode({ generatorKeyId })

// Decrypt the encrypted message
const outputStream =
 createReadStream(filename) .pipe(decryptUnsignedMessageStream(keyring))
```

Python

要指定备用加密算法，请使用带有 `Algorithm` 枚举值的 `algorithm` 参数。

```
# Specify an algorithm suite without signing

# Instantiate a client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R
                                               max_encrypted_data_keys=1)

# Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
        key_ids=[aws_kms_key]
)

# Encrypt the plaintext using an alternate algorithm suite
ciphertext, encrypted_message_header = client.encrypt(
    algorithm=Algorithm.AES_256_GCM_HKDF_SHA512_COMMIT_KEY, source=source_plaintext,
 key_provider=kms_key_provider
)
```

解密未经数字签名加密的消息时，请使用 decrypt-unsigned 串流模式，尤其是在流式传输的同时解密时。

```
# Decrypt unsigned streaming data

# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R
                                                max_encrypted_data_keys=1)

# Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
        key_ids=[aws_kms_key]
)

# Decrypt with decrypt-unsigned
with open(ciphertext_filename, "rb") as ciphertext, open(cycled_plaintext_filename,
 "wb") as plaintext:
    with client.stream(mode="decrypt-unsigned",
                       source=ciphertext,
                       key_provider=master_key_provider) as decryptor:
        for chunk in decryptor:
            plaintext.write(chunk)

# Verify that the encryption context
assert all(
   pair in decryptor.header.encryption_context.items() for pair in
 encryptor.header.encryption_context.items()
)
return ciphertext_filename, cycled_plaintext_filename
```

Rust

要在 for Rust 中 AWS Encryption SDK 指定备用算法套件，请在加密请求中指定该algorithm_suite_id属性。

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
```

```
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw AES keyring
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// Encrypt your plaintext data
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(raw_aes_keyring.clone())
    .encryption_context(encryption_context.clone())
    .algorithm_suite_id(AlgAes256GcmHkdfSha512CommitKey)
    .send()
    .await?;
```

Go

```
import (
    "context"
```

```
 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "AES_256_012"

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  key,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
```

```
    panic(err)
}

// Encrypt your plaintext data
algorithmSuiteId := mpltypes.ESDKAlgorithmSuiteIdAlgAes256GcmHkdfSha512CommitKey
res, err := encryptionClient.Encrypt(context.Background(), esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           aesKeyring,
    AlgorithmSuiteId:  &algorithmSuiteId,
})
if err != nil {
    panic(err)
}
```

限制加密数据密钥

您可以在加密消息中限制加密数据密钥的数量。此最佳实践功能可以帮助您在加密时检测配置错误的密钥环，或者在解密时检测恶意加密文字。这样还可以防止对您的关键基础设施进行不必要、昂贵、可能详尽的调用。当您解密来自不可信来源的消息时，限制加密数据密钥最有用。

尽管大多数加密消息在加密中使用的每个包装密钥都有一个加密数据密钥，但加密的消息最多可以包含 65535 个加密数据密钥。恶意攻击者可能会使用成千上万个加密数据密钥构造加密消息，但这些密钥都无法解密。因此， AWS Encryption SDK 会尝试解密每个加密的数据密钥，直到用尽消息中的加密数据密钥。

要限制加密数据密钥，请使用 `MaxEncryptedDataKeys` 参数。从 AWS Encryption SDK版本 1.9.x 和 2.2.x 开始，此参数可用于所有支持的编程语言。这是可选的，并且在加密和解密时有效。以下示例对使用三个不同的包装密钥加密的数据进行解密。将 `MaxEncryptedDataKeys` 值设置为 3。

C

```
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Construct an AWS KMS keyring */
struct aws_cryptosdk_keyring *kms_keyring =
      Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn1, { key_arn2, key_arn3 });

/* Create a session */
```

```
struct aws_cryptosdk_session *session =
    aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT,
 kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

/* Limit encrypted data keys */
aws_cryptosdk_session_set_max_encrypted_data_keys(session, 3);

/* Decrypt */
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(session,
    plaintext_output,
    plaintext_buf_sz_output,
    &plaintext_len_output,
    ciphertext_input,
    ciphertext_len_input,
    &ciphertext_consumed_output);
assert(aws_cryptosdk_session_is_done(session));
assert(ciphertext_consumed == ciphertext_len);
```

C# / .NET

要限制.NET 中的加密数据密钥，请 AWS Encryption SDK 为.NET 实例化客户端，并将其可选`MaxEncryptedDataKeys`参数设置为所需的值。 AWS Encryption SDK 然后，在配置的 AWS Encryption SDK 实例上调用 `Decrypt()` 方法。

```
// Decrypt with limited data keys

// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    MaxEncryptedDataKeys = 3
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

// Create the keyring
string keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
```

```
var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var decryptKeyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

// Decrypt the ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = decryptKeyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

AWS Encryption CLI

```
# Decrypt with limited encrypted data keys

$ aws-encryption-cli --decrypt \
    --input hello.txt.encrypted \
    --wrapping-keys key=$key_arn1 key=$key_arn2 key=$key_arn3 \
    --buffer \
    --max-encrypted-data-keys 3 \
    --encryption-context purpose=test \
    --metadata-output ~/metadata \
    --output .
```

Java

```
// Construct a client with limited encrypted data keys
final AwsCrypto crypto = AwsCrypto.builder()
  .withMaxEncryptedDataKeys(3)
  .build();

// Create an AWS KMS master key provider
final KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
      .buildStrict(keyArn1, keyArn2, keyArn3);

// Decrypt
final CryptoResult<byte[], KmsMasterKey> decryptResult =
 crypto.decryptData(keyProvider, ciphertext)
```

JavaScript Browser

```
// Construct a client with limited encrypted data keys
const { encrypt, decrypt } = buildClient({ maxEncryptedDataKeys: 3 })

declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}
const clientProvider = getClient(KMS, {
  credentials: { accessKeyId, secretAccessKey, sessionToken }
})

// Create an AWS KMS keyring
const keyring = new KmsKeyringBrowser({
  clientProvider,
  keyIds: [keyArn1, keyArn2, keyArn3],
})

// Decrypt
const { plaintext, messageHeader } = await decrypt(keyring, ciphertext)
```

JavaScript Node.js

```
// Construct a client with limited encrypted data keys
const { encrypt, decrypt } = buildClient({ maxEncryptedDataKeys: 3 })

// Create an AWS KMS keyring
const keyring = new KmsKeyringBrowser({
  keyIds: [keyArn1, keyArn2, keyArn3],
})

// Decrypt
const { plaintext, messageHeader } = await decrypt(keyring, ciphertext)
```

Python

```
# Instantiate a client with limited encrypted data keys
client = aws_encryption_sdk.EncryptionSDKClient(max_encrypted_data_keys=3)

# Create an AWS KMS master key provider
master_key_provider = aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(
```

```
    key_ids=[key_arn1, key_arn2, key_arn3])

# Decrypt
plaintext, header = client.decrypt(source=ciphertext,
 key_provider=master_key_provider)
```

Rust

```
// Instantiate the AWS Encryption SDK client with limited encrypted data keys
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .max_encrypted_data_keys(max_encrypted_data_keys)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

//  Generate `max_encrypted_data_keys` raw AES keyrings to use with your keyring
let mut raw_aes_keyrings: Vec<KeyringRef> = vec![];

assert!(max_encrypted_data_keys > 0, "max_encrypted_data_keys MUST be greater than
 0");

let mut i = 0;
while i < max_encrypted_data_keys {
    let aes_key_bytes = generate_aes_key_bytes();

    let raw_aes_keyring = mpl
        .create_raw_aes_keyring()
        .key_name(key_name)
        .key_namespace(key_namespace)
        .wrapping_key(aes_key_bytes)
        .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
        .send()
        .await?;

    raw_aes_keyrings.push(raw_aes_keyring);
    i += 1;
```

```
}

// Create a Multi Keyring with `max_encrypted_data_keys` AES Keyrings
let generator_keyring = raw_aes_keyrings.remove(0);

let multi_keyring = mpl
    .create_multi_keyring()
    .generator(generator_keyring)
    .child_keyrings(raw_aes_keyrings)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client with limited encrypted data keys
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{
    MaxEncryptedDataKeys: &maxEncryptedDataKeys,
})
if err != nil {
    panic(err)
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
```

```
}

//  Generate `maxEncryptedDataKeys` raw AES keyrings to use with your keyring
rawAESKeyrings := make([]mpltypes.IKeyring, 0, maxEncryptedDataKeys)
var i int64 = 0
for i < maxEncryptedDataKeys {
    key, err := generate256KeyBytesAES()
 if err != nil {
  panic(err)
 }
    aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
  KeyName:      keyName,
  KeyNamespace: keyNamespace,
  WrappingKey:  key,
  WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
 }
 aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
    if err != nil {
  panic(err)
 }
    rawAESKeyrings = append(rawAESKeyrings, aesKeyring)
    i++
}

// Create a Multi Keyring with `max_encrypted_data_keys` AES Keyrings
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
    Generator:     rawAESKeyrings[0],
    ChildKeyrings: rawAESKeyrings[1:],
}
multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
if err != nil {
    panic(err)
}
```

创建发现筛选条件

解密使用 KMS 密钥加密的数据时，最佳实践是在严格模式下解密，也就是说，将包装密钥仅限于您指定的密钥。但是，如有必要，您也可以在发现模式下解密，在这种模式下，您无需指定任何包装密钥。

在此模式下， AWS KMS 无论谁拥有或有权访问该 KMS 密钥，都可以使用加密数据密钥的 KMS 密钥对其进行解密。

如果您必须在发现模式下解密，我们建议您始终使用发现过滤器，该过滤器将可用于指定和分区中的KMS 密钥限制在指定 AWS 账户 和分区中的密钥。发现筛选条件是可选的，但这是最佳实践。

使用下表确定发现筛选条件的分区值。

Region	分区
AWS 区域	aws
中国区域	aws-cn
AWS GovCloud (US) Regions	aws-us-gov

本节中的示例向您显示如何创建发现筛选条件。在使用代码之前，请将示例值替换为 AWS 账户 和分区的有效值。

C

有关完整的示例，请参阅 AWS Encryption SDK for C中的 kms_discovery.cpp。

```
/* Create a discovery filter for an AWS account and partition */

const char *account_id = "111122223333";
const char *partition = "aws";
const std::shared_ptr<Aws::Cryptosdk::KmsKeyring::DiscoveryFilter> discovery_filter
 =

 Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Buil
```

C# / .NET

有关完整示例，请参阅 DiscoveryFilterExample.net 中的 AWS Encryption SDK .cs。

```
// Create a discovery filter for an AWS account and partition

List<string> account = new List<string> { "111122223333" };

DiscoveryFilter exampleDiscoveryFilter = new DiscoveryFilter()
```

```
{
    AccountIds = account,
    Partition = "aws"
}
```

AWS Encryption CLI

```
# Decrypt in discovery mode with a discovery filter

$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-account=111122223333 \
                                     discovery-partition=aws \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output .
```

Java

有关完整示例，请参阅中的 DiscoveryDecryptionExample.java。 AWS Encryption SDK for Java

```
// Create a discovery filter for an AWS account and partition

DiscoveryFilter discoveryFilter = new DiscoveryFilter("aws", 111122223333);
```

JavaScript (Node and Browser)

有关完整的示例，请参阅 AWS Encryption SDK for JavaScript中的 kms_filtered_discovery.ts（Node.js）和 kms_multi_region_discovery.ts（浏览器）。

```
/* Create a discovery filter for an AWS account and partition */
const discoveryFilter = {
  accountIDs: ['111122223333'],
  partition: 'aws',
}
```

Python

有关完整的示例，请参阅 AWS Encryption SDK for Python中的 discovery_kms_provider.py。

```
# Create the discovery filter and specify the region
decrypt_kwargs = dict(
        discovery_filter=DiscoveryFilter(account_ids="111122223333",
 partition="aws"),
        discovery_region="us-west-2",
    )
```

Rust

```
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![111122223333.to_string()])
    .partition("aws".to_string())
    .build()?;
```

Go

```
import (
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
)

discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{111122223333},
    Partition:  "aws",
}
```

配置所需的加密上下文 CMM

您可以使用所需的加密上下文 CMM 在加密操作中要求加密上下文。加密上下文是一组非机密键值对。加密上下文以加密方式绑定到加密的数据，以便需要使用相同的加密上下文解密字段。使用必需的加密上下文 CMM 时，可以指定一个或多个必需的加密上下文密钥（必需的密钥），这些密钥必须包含在所有加密和解密调用中。

> **Note**
>
> 只有以下版本支持所需的加密上下文 CMM：
>
> - 版本 3。 的 x AWS Encryption SDK for Java
> - 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK

- 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

如果您使用所需的加密上下文 CMM 加密数据，则只能使用这些支持的版本之一对其进行解密。

加密时， AWS Encryption SDK 验证所有必需的加密上下文密钥是否包含在您指定的加密上下文中。对您指定的加密上下文进行 AWS Encryption SDK 签名。只有非所需密钥的键值对才会序列化并以明文格式存储在加密操作返回的加密消息的标头中。

解密时，必须提供包含代表所需密钥的所有键值对的加密上下文。 AWS Encryption SDK 使用此加密上下文和存储在加密邮件标头中的键值对来重建您在加密操作中指定的原始加密上下文。如果 AWS Encryption SDK 无法重建原始加密上下文，则解密操作失败。如果您提供的键值对包含值不正确的所需密钥，则无法解密加密消息。您提供的键值对必须与加密时指定的相同。

⚠ Important

请仔细考虑您在加密上下文中为所需密钥选择的值。您必须能够在解密时再次提供相同密钥及其对应值。如果您无法重现所需密钥，则无法解密加密消息。

以下示例使用所需的加密上下文 CMM 初始化密 AWS KMS 钥环。

C# / .NET

```
var encryptionContext = new Dictionary<string, string>()
{
    {"encryption", "context"},
    {"is not", "secret"},
    {"but adds", "useful metadata"},
    {"that can help you", "be confident that"},
    {"the data you are handling", "is what you think it is"}
};

// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```

```
// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = kmsKey
};

// Create the keyring
var kmsKeyring = mpl.CreateAwsKmsKeyring(createKeyringInput);

var createCMMInput = new CreateRequiredEncryptionContextCMMInput
{
    UnderlyingCMM = mpl.CreateDefaultCryptographicMaterialsManager(new
 CreateDefaultCryptographicMaterialsManagerInput{Keyring = kmsKeyring}),
    // If you pass in a keyring but no underlying cmm, it will result in a failure
 because only cmm is supported.
    RequiredEncryptionContextKeys = new List<string>(encryptionContext.Keys)
};

// Create the required encryption context CMM
var requiredEcCMM = mpl.CreateRequiredEncryptionContextCMM(createCMMInput);
```

Java

```
// Instantiate the AWS Encryption SDK
final AwsCrypto crypto = AwsCrypto.builder()
        .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
        .build();

// Create your encryption context
final Map<String, String> encryptionContext = new HashMap<>();
encryptionContext.put("encryption", "context");
encryptionContext.put("is not", "secret");
encryptionContext.put("but adds", "useful metadata");
encryptionContext.put("that can help you", "be confident that");
encryptionContext.put("the data you are handling", "is what you think it is");

// Create a list of required encryption contexts
final List<String> requiredEncryptionContextKeys = Arrays.asList("encryption",
 "context");

// Create the keyring
```

```
final MaterialProviders materialProviders = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsKeyringInput keyringInput = CreateAwsKmsKeyringInput.builder()
        .kmsKeyId(keyArn)
        .kmsClient(KmsClient.create())
        .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(keyringInput);

// Create the required encryption context CMM
ICryptographicMaterialsManager cmm =
    materialProviders.CreateDefaultCryptographicMaterialsManager(
        CreateDefaultCryptographicMaterialsManagerInput.builder()
            .keyring(kmsKeyring)
            .build()
    );
ICryptographicMaterialsManager requiredCMM =
    materialProviders.CreateRequiredEncryptionContextCMM(
        CreateRequiredEncryptionContextCMMInput.builder()
            .requiredEncryptionContextKeys(requiredEncryptionContextKeys)
            .underlyingCMM(cmm)
            .build()
        );
```

Python

要将 AWS Encryption SDK for Python 与所需的加密上下文 CMM 一起使用，还必须使用材料提供者库 (MPL)。

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Create your encryption context
encryption_context: Dict[str, str] = {
    "key1": "value1",
    "key2": "value2",
    "requiredKey1": "requiredValue1",
    "requiredKey2": "requiredValue2"
}

# Create a list of required encryption context keys
```

```
required_encryption_context_keys: List[str] = ["requiredKey1", "requiredKey2"]

# Instantiate the material providers library
mpl: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
 keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    kms_key_id=kms_key_id,
    kms_client=boto3.client('kms', region_name="us-west-2")
)
kms_keyring: IKeyring = mpl.create_aws_kms_keyring(keyring_input)

# Create the required encryption context CMM
underlying_cmm: ICryptographicMaterialsManager = \
    mpl.create_default_cryptographic_materials_manager(
        CreateDefaultCryptographicMaterialsManagerInput(
            keyring=kms_keyring
        )
    )

required_ec_cmm: ICryptographicMaterialsManager = \
    mpl.create_required_encryption_context_cmm(
        CreateRequiredEncryptionContextCMMInput(
            required_encryption_context_keys=required_encryption_context_keys,
            underlying_cmm=underlying_cmm,
        )
    )
```

Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Create your encryption context
let encryption_context = HashMap::from([
```

```
    ("key1".to_string(), "value1".to_string()),
    ("key2".to_string(), "value2".to_string()),
    ("requiredKey1".to_string(), "requiredValue1".to_string()),
    ("requiredKey2".to_string(), "requiredValue2".to_string()),
]);

// Create a list of required encryption context keys
let required_encryption_context_keys: Vec<String> = vec![
    "requiredKey1".to_string(),
    "requiredKey2".to_string(),
];

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create the AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

kms_multi_keyring: IKeyring = mat_prov.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)

// Create the required encryption context CMM
let underlying_cmm = mpl
    .create_default_cryptographic_materials_manager()
    .keyring(kms_keyring)
    .send()
    .await?;

let required_ec_cmm = mpl
    .create_required_encryption_context_cmm()
    .underlying_cmm(underlying_cmm.clone())
    .required_encryption_context_keys(required_encryption_context_keys)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = defaultKmsKeyRegion
})

// Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Create a list of required encryption context keys
requiredEncryptionContextKeys := []string{}
```

```
requiredEncryptionContextKeys = append(requiredEncryptionContextKeys,
 "requiredKey1", "requiredKey2")

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create the AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  utils.GetDefaultKMSKeyId(),
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Create the required encryption context CMM
underlyingCMM, err :=
 matProv.CreateDefaultCryptographicMaterialsManager(context.Background(),
 mpltypes.CreateDefaultCryptographicMaterialsManagerInput{Keyring: awsKmsKeyring})
if err != nil {
    panic(err)
}
requiredEncryptionContextInput := mpltypes.CreateRequiredEncryptionContextCMMInput{
    UnderlyingCMM: underlyingCMM,
    RequiredEncryptionContextKeys: requiredEncryptionContextKeys,
}
requiredEC, err := matProv.CreateRequiredEncryptionContextCMM(context.Background(),
 requiredEncryptionContextInput)
if err != nil {
    panic(err)
}
```

设置承诺策略

承诺策略是一种配置设置，用于确定您的应用程序是否使用密钥承诺进行加密和解密。使用密钥承诺进行加密和解密是 AWS Encryption SDK 最佳实践。

设置和调整承诺策略是从 AWS Encryption SDK 版本 1.7.x 及更早版本迁移到版本 2.0.x 及更高版本的关键步骤。迁移主题中详细解释了这一进展。

AWS Encryption SDK 最新版本（从版本 2.0.x 开始）中的默认承诺策略值 `RequireEncryptRequireDecrypt` 非常适合大多数情况。但是，如果您需要解密未经密钥承诺而加密的加密文字，则可能需要将承诺策略更改为 `RequireEncryptAllowDecrypt`。有关如何使用每种编程语言设置承诺策略的示例，请参阅 设置您的承诺策略。

使用串流数据

在流式传输数据进行解密时，请注意，在完整性检查完成后、在验证数字签名之前， AWS Encryption SDK 返回的是解密后的纯文本。为确保在签名通过验证之前不返回或使用明文，我们建议您缓冲流式传输的明文，直到整个解密过程完成。

只有当您流式传输加密文字进行解密时，并且只有当您使用包含数字签名的算法套件（例如默认算法套件）时，才会出现这个问题。

为了简化缓冲，某些 AWS Encryption SDK 语言实现（例如 Node.js AWS Encryption SDK for JavaScript 中）在解密方法中包含缓冲功能。始终流式传输输入和输出的 AWS Encryption CLI 在版本 1.9.x 和 2.2.x 中引入了 `--buffer` 参数。在其他语言实现中，您可以使用现有的缓冲功能。（.N AWS Encryption SDK ET 版不支持流式传输。）

如果您使用的是没有数字签名的算法套件，请务必在每种语言实现中使用 `decrypt-unsigned` 功能。此功能可以解密加密文字，但如果遇到签名的加密文字，则会失败。有关更多信息，请参阅 选择算法套件。

缓存数据密钥

通常，不鼓励重复使用数据密钥，但 AWS Encryption SDK 提供了数据密钥缓存选项，可限制对数据密钥的重复使用。数据密钥缓存可以提高某些应用程序的性能并减少对密钥基础设施的调用。在生产环境中使用数据密钥缓存之前，请调整安全阈值并进行测试，以确保重复使用数据密钥的好处大于缺点。

密钥存储在 AWS Encryption SDK

在中 AWS Encryption SDK，密钥存储是一个 Amazon DynamoDB 表，该表保留了分层密钥环使用的 AWS KMS 分层数据。密钥存储有助于减少使用分层密钥环执行加密操作所需的调用次数。 AWS KMS

密钥库保留并管理分支密钥，分层密钥环使用这些密钥来执行信封加密和保护数据加密密钥。密钥库存储活动分支密钥和分支密钥的所有先前版本。活动分支密钥为最新分支密钥版本。分层密钥环对每个加密请求使用唯一的数据加密密钥，并使用从活动分支密钥派生的唯一包装密钥对每个数据加密密钥进行加密。分层密钥环依赖在活动分支密钥及其派生包装密钥之间建立的层次结构。

密钥商店术语和概念

Key store (密钥存储)

用于保存分层数据（例如分支密钥和信标密钥）的 DynamoDB 表。

根密钥

一种对称加密 KMS 密钥，用于生成和保护密钥库中的分支密钥和信标密钥。

分支密钥

一种数据密钥，可重复用于派生用于信封加密的唯一包装密钥。您可以在一个密钥库中创建多个分支密钥，但是每个分支密钥一次只能有一个有效的分支密钥版本。活动分支密钥为最新分支密钥版本。

分支密钥是 AWS KMS keys 通过使用 k ms: GenerateDataKeyWithoutPlaintext 操作派生的。

包装密钥

一种唯一的数据密钥，用于加密操作中使用的数据加密密钥。

包装密钥源自分支密钥。有关密钥派生过程的更多信息，请参阅AWS KMS 分层密钥环技术细节。

数据加密密钥

用于加密操作的数据密钥。分层密钥环对每个加密请求使用唯一的数据加密密钥。

实施最低权限

使用密钥库和 AWS KMS 分层密钥环时，我们建议您通过定义以下角色来遵循最低权限原则：

密钥库管理员

密钥库管理员负责创建和管理密钥库及其持久保存和保护的分支密钥。密钥存储管理员应该是唯一对用作您的密钥存储的 Amazon DynamoDB 表具有写入权限的用户。他们应该是唯一有权访问特权管理员操作（例如CreateKey和）的用户VersionKey。只有在静态配置密钥库操作时，才能执行这些操作。

CreateKey是一项特权操作，可以将新的 KMS 密钥 ARN 添加到您的密钥库许可名单。此 KMS 密钥可以创建新的活动分支密钥。我们建议限制对此操作的访问权限，因为一旦将 KMS 密钥添加到分支密钥存储中，便无法将其删除。

密钥库用户

在大多数用例中，密钥库用户在加密、解密、签名和验证数据时仅通过分层密钥环与密钥库进行交互。因此，他们只需要对用作您的密钥存储库的 Amazon DynamoDB 表具有读取权限。密钥库用户只需要访问使加密操作成为可能的使用操作，例如GetActiveBranchKeyGetBranchKeyVersion、和GetBeaconKey。他们不需要权限即可创建或管理他们使用的分支密钥。

当密钥存储操作处于静态配置状态时，或者将密钥存储操作配置为用于发现时，您可以执行使用操作。将密钥存储操作配置为用于发现时，您无法执行管理员操作（CreateKey和VersionKey）。

如果您的分支密钥存储管理员在您的分支密钥存储中列入了多个 KMS 密钥，我们建议您的密钥存储用户配置其密钥存储操作以进行发现，以便他们的分层密钥环可以使用多个 KMS 密钥。

创建密钥库

在创建分支密钥或使用分AWS KMS 层密钥环之前，必须先创建密钥存储，即管理和保护分支密钥的 Amazon DynamoDB 表。

Important

请勿删除保留分支密钥的 DynamoDB 表。如果删除此表，则将无法解密使用分层密钥环加密的任何数据。

按照 Amazon DynamoDB 开发者指南中的创建表过程进行操作，使用以下必需的字符串值作为分区键和排序键。

	分区键	排序键
基表	`branch-key-id`	`type`

逻辑密钥库名称

在命名用作密钥存储的 DynamoDB 表时，请务必仔细考虑在配置密钥存储操作时要指定的逻辑密钥存储名称。逻辑密钥库名称充当密钥库的标识符，在第一个用户最初定义后无法更改。在密钥存储操作中，必须始终指定相同的逻辑密钥存储名称。

DynamoDB 表名称和逻辑密钥存储名称之间必须存在 one-to-one映射。为简化 DynamoDB 还原操作，逻辑密钥存储名称以加密方式绑定到表中存储的所有数据。虽然逻辑密钥存储名称可能与您的 DynamoDB 表名称不同，但我们强烈建议将您的 DynamoDB 表名称指定为逻辑密钥存储名称。如果从备份中恢复 DynamoDB 表后您的表名称发生变化，则可以将逻辑密钥存储名称映射到新的 DynamoDB 表名称，以确保分层密钥环仍然可以访问您的密钥存储。

请勿在逻辑密钥存储库名称中包含机密或敏感信息。在 AWS KMS CloudTrail 事件中，逻辑密钥存储库名称以纯文本形式显示为。`tablename`

后续步骤

1. the section called “配置密钥存储操作”
2. the section called “创建分支密钥”
3. 创建 AWS KMS 分层密钥环

配置密钥存储操作

密钥存储操作决定了您的用户可以执行哪些操作，以及他们的 AWS KMS 分层密钥环如何使用您的密钥存储中允许列出的 KMS 密钥。 AWS Encryption SDK 支持以下密钥存储操作配置。

静态

> 当您静态配置密钥存储时，密钥存储只能使用与您在配置密钥存储操作时提供的 KMS 密钥 ARN 关联的 KMS 密钥。`kmsConfiguration`如果在创建、版本控制或获取分支密钥时遇到不同的 KMS 密钥 ARN，则会引发异常。

您可以在中指定多区域 KMS 密钥`kmsConfiguration`，但该密钥的整个 ARN（包括区域）都保留在从 KMS 密钥派生的分支密钥中。您不能在其他区域指定密钥，必须提供完全相同的多区域密钥才能使值匹配。

静态配置密钥存储操作时，可以执行使用操作 (`GetActiveBranchKey`、`GetBranchKeyVersion`、`GetBeaconKey`) 和管理操作 (`CreateKey`和`VersionKey`)。 `CreateKey`是一项特权操作，可以将新的 KMS 密钥 ARN 添加到您的密钥库许可名单。此 KMS 密钥可以创建新的活动分支密钥。我们建议限制对此操作的访问权限，因为将 KMS 密钥添加到密钥存储库后，便无法将其删除。

Discovery

当您配置密钥库操作以进行发现时，密钥库可以使用密钥库中列入许可名单的任何 AWS KMS key ARN。但是，如果遇到多区域 KMS 密钥，并且该密钥的 ARN 中的区域与正在使用的客户端的区域不匹配，则会引发异常。 AWS KMS

在配置密钥库以供发现时，您无法执行管理操作，例如`CreateKey`和`VersionKey`。您只能执行启用加密、解密、签名和验证操作的使用操作。有关更多信息，请参阅 the section called “实施最低权限”。

配置您的关键商店操作

在配置密钥存储操作之前，请确保满足以下先决条件。

- 确定您需要执行哪些操作。有关更多信息，请参阅 the section called “实施最低权限”。
- 选择一个逻辑密钥存储名称

  DynamoDB 表名称和逻辑密钥存储名称之间必须存在 one-to-one映射。逻辑密钥存储名称以加密方式绑定到表中存储的所有数据，以简化 DynamoDB 还原操作，在第一个用户最初定义后无法对其进行更改。在密钥存储操作中，必须始终指定相同的逻辑密钥存储名称。有关更多信息，请参阅 logical key store name。

静态配置

以下示例以静态方式配置密钥存储操作。您必须指定用作密钥存储的 DynamoDB 表的名称、密钥存储的逻辑名称以及标识对称加密 KMS 密钥的 KMS 密钥 ARN。

Note

请仔细考虑您在静态配置密钥存储服务时指定的 KMS 密钥 ARN。该`CreateKey`操作将 KMS 密钥 ARN 添加到您的分支密钥存储许可名单中。将 KMS 密钥添加到分支密钥存储库后，便无法将其删除。

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                  KeyStoreConfig.builder()
                          .ddbClient(DynamoDbClient.create())
                          .ddbTableName(keyStoreName)
                          .logicalKeyStoreName(logicalKeyStoreName)
                          .kmsClient(KmsClient.create())
                          .kmsConfiguration(KMSConfiguration.builder()
                                  .kmsKeyArn(kmsKeyArn)
                                  .build())
                          .build()).build();
```

C# / .NET

```
var kmsConfig = new KMSConfiguration { KmsKeyArn = kmsKeyArn };
  var keystoreConfig = new KeyStoreConfig
  {
      KmsClient = new AmazonKeyManagementServiceClient(),
      KmsConfiguration = kmsConfig,
      DdbTableName = keyStoreName,
      DdbClient = new AmazonDynamoDBClient(),
      LogicalKeyStoreName = logicalKeyStoreName
  };
  var keystore = new KeyStore(keystoreConfig);
```

Python

```
keystore: KeyStore = KeyStore(
    config=KeyStoreConfig(
        ddb_client=ddb_client,
        ddb_table_name=key_store_name,
        logical_key_store_name=logical_key_store_name,
        kms_client=kms_client,
```

```
        kms_configuration=KMSConfigurationKmsKeyArn(
            value=kms_key_id
        ),
    )
)
```

Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let key_store_config = KeyStoreConfig::builder()
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .ddb_client(aws_sdk_dynamodb::Client::new(&sdk_config))
    .ddb_table_name(key_store_name)
    .logical_key_store_name(logical_key_store_name)
    .kms_configuration(KmsConfiguration::KmsKeyArn(kms_key_arn.to_string()))
    .build()?;

let keystore = keystore_client::Client::from_conf(key_store_config)?;
```

Go

```
import (
 keystore "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygenerated"
 keystoretypes "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygeneratedtypes"
)

kmsConfig := keystoretypes.KMSConfigurationMemberkmsKeyArn{
    Value: kmsKeyArn,
}
keyStore, err := keystore.NewClient(keystoretypes.KeyStoreConfig{
    DdbTableName:         keyStoreTableName,
    KmsConfiguration:     &kmsConfig,
    LogicalKeyStoreName: logicalKeyStoreName,
    DdbClient:            ddbClient,
    KmsClient:            kmsClient,
})
if err != nil {
    panic(err)
}
```

发现配置

以下示例配置了用于发现的密钥存储操作。您必须指定用作密钥存储的 DynamoDB 表的名称和逻辑密钥存储名称。

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                 KeyStoreConfig.builder()
                         .ddbClient(DynamoDbClient.create())
                         .ddbTableName(keyStoreName)
                         .logicalKeyStoreName(logicalKeyStoreName)
                         .kmsClient(KmsClient.create())
                         .kmsConfiguration(KMSConfiguration.builder()
                                 .discovery(Discovery.builder().build())
                                 .build())
                         .build()).build();
```

C# / .NET

```
var keystoreConfig = new KeyStoreConfig
 {
     KmsClient = new AmazonKeyManagementServiceClient(),
     KmsConfiguration = new KMSConfiguration {Discovery = new Discovery()},
     DdbTableName = keyStoreName,
     DdbClient = new AmazonDynamoDBClient(),
     LogicalKeyStoreName = logicalKeyStoreName
 };
 var keystore = new KeyStore(keystoreConfig);
```

Python

```
keystore: KeyStore = KeyStore(
    config=KeyStoreConfig(
        ddb_client=ddb_client,
        ddb_table_name=key_store_name,
        logical_key_store_name=logical_key_store_name,
        kms_client=kms_client,
        kms_configuration=KMSConfigurationDiscovery(
            value=Discovery()
        ),
    )
```

```
)
```

Rust

```
let key_store_config = KeyStoreConfig::builder()
        .kms_client(kms_client)
        .ddb_client(ddb_client)
        .ddb_table_name(key_store_name)
        .logical_key_store_name(logical_key_store_name)

 .kms_configuration(KmsConfiguration::Discovery(Discovery::builder().build()?))
        .build()?;
```

Go

```
import (
 keystore "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygenerated"
 keystoretypes "github.com/aws/aws-cryptographic-material-providers-library/mpl/
awscryptographykeystoresmithygeneratedtypes"
)

kmsConfig := keystoretypes.KMSConfigurationMemberdiscovery{}
keyStore, err := keystore.NewClient(keystoretypes.KeyStoreConfig{
    DdbTableName:         keyStoreName,
    KmsConfiguration:     &kmsConfig,
    LogicalKeyStoreName: logicalKeyStoreName,
    DdbClient:            ddbClient,
    KmsClient:            kmsClient,
})
if err != nil {
    panic(err)
}
```

创建有效的分支密钥

分支密钥是派生自分支密钥的数据密钥 AWS KMS key ， AWS KMS 分层密钥环使用该密钥来减少调用的次数。 AWS KMS活动分支密钥为最新分支密钥版本。分层密钥环为每个加密请求生成唯一的数据密钥，并使用从活动分支密钥派生的唯一包装密钥对每个数据密钥进行加密。

要创建新的活动分支密钥，必须静态配置密钥存储操作。 CreateKey是一项特权操作，用于将密钥库操作配置中指定的 KMS 密钥 ARN 添加到密钥库许可名单中。然后，使用 KMS 密钥生成新的活动分支密钥。我们建议限制对此操作的访问权限，因为将 KMS 密钥添加到密钥存储库后，便无法将其删除。

您可以将密钥存储库中的一个 KMS 密钥列入许可名单，也可以通过更新您在密钥存储操作配置中指定的 KMS 密钥 ARN 并再次调用来允许列入多个 KMS 密钥。CreateKey如果您将多个 KMS 密钥列入许可名单，则您的密钥存储用户应配置其密钥存储操作以供发现，以便他们可以使用他们有权访问的密钥存储库中的任何允许列表的密钥。有关更多信息，请参阅 the section called “配置密钥存储操作”。

所需的权限

要创建分支密钥，您需要拥有密钥存储操作中指定的 KMS 密钥的 kms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt 权限。

创建分支密钥

以下操作使用您在密钥存储操作配置中指定的 KMS 密钥创建新的活动分支密钥，并将活动分支密钥添加到用作密钥存储的 DynamoDB 表中。

调用 `CreateKey` 时，您可以选择指定以下可选值。

- `branchKeyIdentifier`：定义自定义 `branch-key-id`。

  要创建自定义 `branch-key-id`，还必须加入包含 `encryptionContext` 参数的其他加密上下文。
- `encryptionContext`: 定义一组可选的非秘密密钥值对，用于在 kms: 调用中包含的加密上下文中提供额外的经过身份验证的数据 (AAD)。GenerateDataKeyWithoutPlaintext

  此额外加密上下文带有 `aws-crypto-ec:` 前缀。

Java

```
final Map<String, String> additionalEncryptionContext =
 Collections.singletonMap("Additional Encryption Context for",
          "custom branch key id");

  final String BranchKey = keystore.CreateKey(
          CreateKeyInput.builder()
                  .branchKeyIdentifier(custom-branch-key-id) //OPTIONAL
                  .encryptionContext(additionalEncryptionContext) //OPTIONAL
```

```
                    .build()).branchKeyIdentifier();
```

C# / .NET

```
var additionalEncryptionContext = new Dictionary<string, string>();
  additionalEncryptionContext.Add("Additional Encryption Context for", "custom
 branch key id");

  var branchKeyId = keystore.CreateKey(new CreateKeyInput
  {
      BranchKeyIdentifier = "custom-branch-key-id", // OPTIONAL
      EncryptionContext = additionalEncryptionContext // OPTIONAL
  });
```

Python

```
additional_encryption_context = {"Additional Encryption Context for": "custom branch
 key id"}

branch_key_id: str = keystore.create_key(
    CreateKeyInput(
        branch_key_identifier = "custom-branch-key-id", # OPTIONAL
        encryption_context = additional_encryption_context, # OPTIONAL
    )
)
```

Rust

```
let additional_encryption_context = HashMap::from([
    ("Additional Encryption Context for".to_string(), "custom branch key
 id".to_string())
]);

let branch_key_id = keystore.create_key()
    .branch_key_identifier("custom-branch-key-id") // OPTIONAL
    .encryption_context(additional_encryption_context) // OPTIONAL
    .send()
    .await?
    .branch_key_identifier
    .unwrap();
```

Go

```
encryptionContext := map[string]string{
    "Additional Encryption Context for": "custom branch key id",
}

branchKey, err := keyStore.CreateKey(context.Background(),
 keystoretypes.CreateKeyInput{
    BranchKeyIdentifier: &customBranchKeyId,
    EncryptionContext:   additional_encryption_context,
})
if err != nil {
    return "", err
}
```

首先，CreateKey 操作生成以下值。

- 适用于 branch-key-id 的版本 4 通用唯一标识（UUID）（除非您指定了自定义 branch-key-id）。
- 适用于分支密钥版本的版本 4 UUID
- timestamp 必须采用协调世界时（UTC）ISO 8601 日期和时间格式。

然后，该CreateKey操作GenerateDataKeyWithoutPlaintext使用以下请求调用 kms:。

```
{
    "EncryptionContext": {
       "branch-key-id" : "branch-key-id",
       "type" : "type",
       "create-time" : "timestamp",
       "logical-key-store-name" : "the logical table name for your key store",
       "kms-arn" : the KMS key ARN,
       "hierarchy-version" : "1",
       "aws-crypto-ec:contextKey": "contextValue"
    },
    "KeyId": "the KMS key ARN you specified in your key store actions",
    "NumberOfBytes": "32"
  }
```

接下来，该CreateKey操作调用 km ReEncrypt s:，通过更新加密上下文为分支密钥创建活动记录。

最后，该CreateKey操作调用 ddb: TransactWriteItems 来编写一个新项目，该项目将保留您在步骤 2 中创建的表中的分支密钥。项目具有以下属性。

```
{
        "branch-key-id" : branch-key-id,
        "type" : "branch:ACTIVE",
        "enc" : the branch key returned by the GenerateDataKeyWithoutPlaintext call,
        "version": "branch:version:the branch key version UUID",
        "create-time" : "timestamp",
        "kms-arn" : "the KMS key ARN you specified in Step 1",
        "hierarchy-version" : "1",
        "aws-crypto-ec:contextKey": "contextValue"
 }
```

轮换您的活动分支密钥

每个分支密钥一次仅能有一个活动版本。通常，每个有效的分支密钥版本都用于满足多个请求。但是您可以控制活动分支密钥的重复使用程度，并确定活动分支密钥的轮换频率。

分支密钥不用于加密明文数据密钥。它们用于派生对明文数据密钥进行加密的唯一包装密钥。包装密钥派生过程生成唯一的 32 字节包装密钥，其随机掩码为 28 字节。这意味着，在发生加密损耗之前，分支密钥可以派生出超过 79 万亿或 2^{96} 个唯一的包装密钥。尽管耗尽风险非常低，但由于业务或合同规则或政府法规，您可能需要轮换活动分支密钥。

在您轮换之前，分支密钥的活动版本会一直处于活动状态。以前版本的活动分支密钥不会用于执行加密操作，也不能用于派生新的包装密钥，但仍然可以查询这些密钥并提供包装密钥来解密它们在活动状态下加密的数据密钥。

所需的权限

要轮换分支密钥，您需要密钥存储操作中指定的 KMS 密钥的 kms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt 权限。

轮换有效的分支密钥

使用该VersionKey操作来轮换您的活动分支密钥。轮换活动分支密钥时，系统会创建新的分支密钥代替先前版本。当您轮换活动分支密钥时，branch-key-id 不会改变。在调用 VersionKey 时，必须指定用于标识当前活动分支密钥的 branch-key-id。

Java

```
keystore.VersionKey(
    VersionKeyInput.builder()
        .branchKeyIdentifier("branch-key-id")
        .build()
);
```

C# / .NET

```
keystore.VersionKey(new VersionKeyInput{BranchKeyIdentifier = branchKeyId});
```

Python

```
keystore.version_key(
    VersionKeyInput(
        branch_key_identifier=branch_key_id
    )
)
```

Rust

```
keystore.version_key()
        .branch_key_identifier(branch_key_id)
        .send()
        .await?;
```

Go

```
_, err = keyStore.VersionKey(context.Background(), keystoretypes.VersionKeyInput{
    BranchKeyIdentifier: branchKeyId,
})
if err != nil {
    return err
}
```

密钥环

支持的编程语言实现使用密钥环来执行信封加密。密钥环生成、加密和解密数据密钥。密钥环确定保护每条消息的唯一数据密钥的来源，以及加密该数据密钥的包装密钥。您在加密时指定一个密钥环，并在解密时指定相同或不同的密钥环。您可以使用开发工具包提供的密钥环，也可以编写您自己的兼容自定义密钥环。

您可以单独使用每个密钥环，也可以将多个密钥环合并为一个多重密钥环。虽然大多数密钥环可以生成、加密和解密数据密钥，但您也可以创建只执行一项特定操作的密钥环，例如只生成数据密钥的密钥环，并将此密钥环与其他密钥环结合使用。

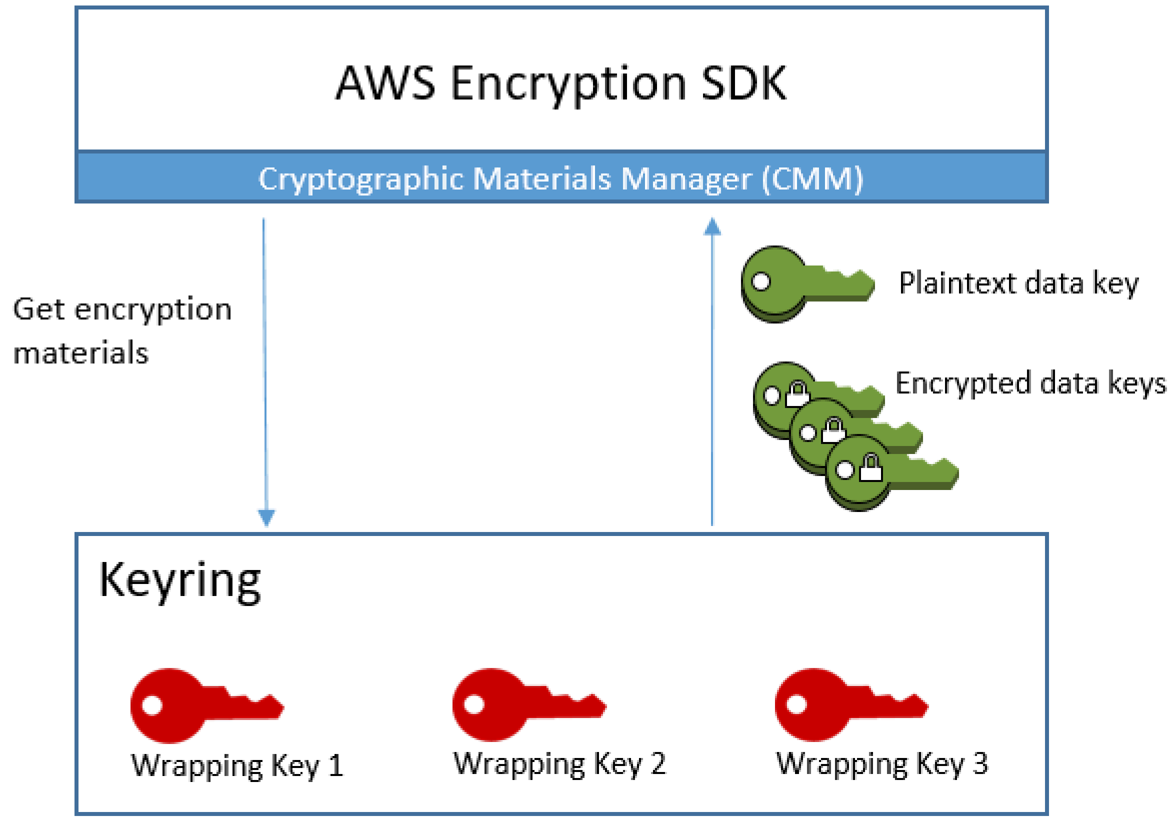
我们建议您使用可保护包装密钥并在安全边界内执行加密操作的密钥环，例如密 AWS KMS 钥环，它使用永不保密 AWS Key Management Service() AWS KMS keys AWS KMS的密钥环。您还可以编写一个使用封装密钥的密钥环，这些密钥存储在硬件安全模块 (HSMs) 中或受其他主密钥服务保护。有关详细信息，请参阅 AWS Encryption SDK Specification 中的 Keyring Interface 主题。

密钥环起着其他编程语言实现中使用的主密钥和主密钥提供者的作用。如果您使用 AWS Encryption SDK 的不同语言实施来加密和解密数据，请务必使用兼容密钥环和主密钥提供程序。有关更多信息，请参阅 密钥环兼容性。

本主题说明如何使用的密钥环功能 AWS Encryption SDK 以及如何选择密钥环。

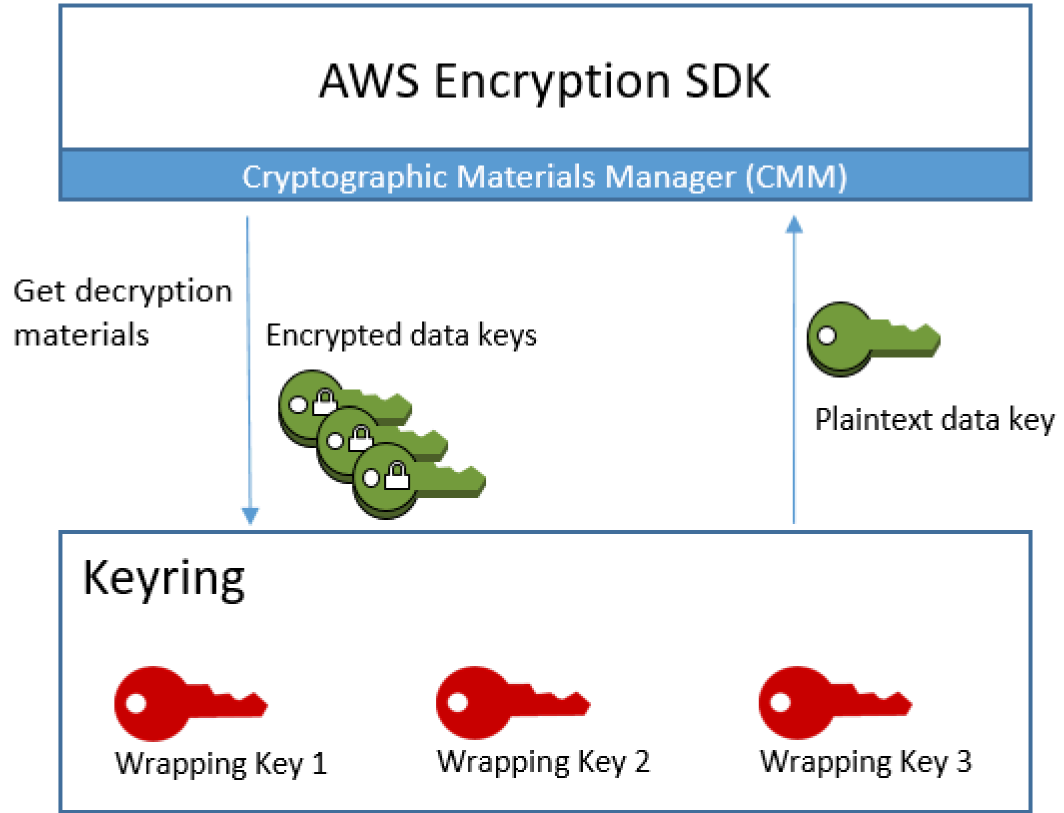
密钥环的工作方式

加密数据时，会 AWS Encryption SDK 要求密钥环提供加密材料。密钥环返回一个明文数据密钥以及由密钥环中的每个包装密钥加密的数据密钥副本。 AWS Encryption SDK 使用明文密钥加密数据，然后销毁纯文本数据密钥。然后， AWS Encryption SDK 返回一条包含加密数据密钥和加密数据的加密消息。



解密数据时，您可以使用加密数据所用的密钥环，也可以使用其他密钥环。要解密数据，解密密钥环必须包含（或有权访问）加密密钥环中的至少一个包装密钥。

将加密的数据密钥从加密的消息 AWS Encryption SDK 传递到密钥环，并要求密钥环解密其中任何一个。密钥环使用其包装密钥以解密一个加密的数据密钥，并返回明文数据密钥。 AWS Encryption SDK 使用明文数据密钥以解密数据。如果密钥环中的所有包装密钥都无法解密任何加密的数据密钥，解密操作将失败。



您可以使用一个密钥环，也可以将相同类型或不同类型的密钥环组合到一个个多重密钥环中。当您加密数据时，多重密钥环返回使用构成该多重密钥环的所有密钥环中的所有包装密钥加密的数据密钥副本。您可以使用包含多重密钥环中任一包装密钥的密钥环解密数据。

密钥环兼容性

尽管不同的语言实现 AWS Encryption SDK 有一些架构差异，但它们完全兼容，但受语言限制的约束。您可以使用一种语言实施加密数据，用其他语言实施进行解密。不过，您必须使用相同或相应的包装密钥加密和解密数据密钥。有关语言限制的信息，请参阅有关每种语言实现的主题，例如 AWS Encryption SDK for JavaScript 主题the section called “兼容性”中的主题。

以下编程语言支持密钥环：

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK 对于.NET
- 版本 3。 的 x AWS Encryption SDK for Java

- 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
- AWS Encryption SDK 对于 Rust
- AWS Encryption SDK for Go

对加密密钥环的不同要求

在除之外的 AWS Encryption SDK 语言实现中 AWS Encryption SDK for C，所有封装在加密密钥环（或多密钥环）或主密钥提供程序中的密钥都必须能够加密数据密钥。如有任何包装密钥无法加密，此加密方法将失败。因此，调用方必须拥有密钥环中所有密钥的所需权限。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败。

唯一的例外是 AWS Encryption SDK for C，加密操作会忽略标准发现密钥环，但是如果您单独或在多密钥环中指定多区域发现密钥环，则会失败。

兼容的密钥环和主密钥提供程序

下表显示了哪些主密钥和主密钥提供程序与它们提供的密钥环兼容。 AWS Encryption SDK 有关语言实施的主题中解释了由于语言约束而导致的任何轻微不兼容情况。

密钥环：	主密钥提供程序：
AWS KMS 钥匙圈	KMSMaster密钥 (Java) KMSMasterKeyProvider (Java) KMSMaster密钥 (Python) KMSMasterKeyProvider（Python） Note AWS Encryption SDK for Python 和 AWS Encryption SDK for Java 不包括等同于AWS KMS 区域发现密钥环的主密钥或主密钥提供程序。
AWS KMS 分层钥匙圈	由以下编程语言和版本支持：

密钥环：	主密钥提供程序：
	• 版本 3。 的 x AWS Encryption SDK for Java • 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK • 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。 • 版本 1。 x 的 fo r AWS Encryption SDK Rust • 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go
AWS KMS ECDH 钥匙圈	由以下编程语言和版本支持： • 版本 3。 的 x AWS Encryption SDK for Java • 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK • 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。 • 版本 1。 x 的 fo r AWS Encryption SDK Rust • 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go
原始 AES 密钥环	与对称加密密钥一起使用时： JceMasterKey(Java) RawMasterKey（Python）
原始 RSA 密钥环	与非对称加密密钥一起使用时： JceMasterKey(Java) RawMasterKey（Python） Note 原始 RSA 密钥环不支持非对称 KMS 密钥。如果要使用非对称 RSA KMS 密钥，请使用版本 4。 .NET 的 AWS Encryption SDK x 及更高版本支持使用对称加密 (`SYMMETRIC_DEFAULT`) 或非对称 RSA 的密 AWS KMS 钥环。 AWS KMS keys

密钥环：	主密钥提供程序：
未加工的 ECDH 钥匙圈	由以下编程语言和版本支持： • 版本 3。 的 x AWS Encryption SDK for Java • 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK • 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。 • 版本 1。 x 的 fo r AWS Encryption SDK Rust • 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

AWS KMS 钥匙圈

密 AWS KMS 钥环AWS KMS keys用于生成、加密和解密数据密钥。 AWS Key Management Service (AWS KMS) 保护您的 KMS 密钥并在 FIPS 边界内执行加密操作。建议您尽可能使用 AWS KMS 密钥环或具有类似安全属性的密钥环。

所有支持密钥环的编程语言实现都支持使用对称加密 KMS AWS KMS 密钥的密钥环。以下编程语言实现还支持使用非对称 RSA KMS AWS KMS 密钥的密钥环：

- 版本 3。 的 x AWS Encryption SDK for Java
- 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
- 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
- 版本 1。 x 的 fo r AWS Encryption SDK Rust
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

如果您尝试将非对称 KMS 密钥纳入任何其他语言实施的加密密钥环中，加密调用将失败。如果将其纳入解密密钥环中，调用将被忽略。

从 2.3 版开始，您可以在密钥 AWS KMS 环或主密钥提供程序中使用 AWS KMS 多区域密钥。 AWS Encryption SDK 和版本 3.0 中的 x。 AWS 加密 CLI 中的 x。有关使用该 multi-Region-aware符号的详细信息和示例，请参阅使用多区域 AWS KMS keys。有关多区域密钥的信息，请参阅《AWS Key Management Service 开发人员指南》中的使用多区域密钥。

Note

中所有提及 KMS 密钥环的 AWS Encryption SDK 内容均指 AWS KMS 密钥环。

AWS KMS 钥匙圈可以包括两种类型的包装钥匙：

- 生成器密钥：生成并加密明文数据密钥。加密数据的密钥环必须有一个生成器密钥。
- 其他密钥：加密生成器密钥生成的纯文本数据密钥。 AWS KMS 钥匙圈可以有零个或多个额外的钥匙。

您使用的必须具有生成器密钥才能对消息进行加密。当 AWS KMS 密钥环只有一个 KMS 密钥时，该密钥用于生成和加密数据密钥。解密时，生成器密钥是可选的，生成器密钥和其他密钥之间的区别将被忽略。

像所有钥匙圈一样， AWS KMS 钥匙圈可以单独使用，也可以与其他相同或不同类型的钥匙圈一起在多钥匙圈中使用。

主题

- AWS KMS 密钥环所需的权限
- 在 AWS KMS 钥匙圈 AWS KMS keys 中识别
- 创建密 AWS KMS 钥环
- 使用 AWS KMS 发现密钥环
- 使用 AWS KMS 区域发现密钥环

AWS KMS 密钥环所需的权限

AWS Encryption SDK 不需要 AWS 账户 ，也不依赖于任何一个 AWS 服务。但是，要使用 AWS KMS 密钥环，您需要对 AWS 账户 密钥环 AWS KMS keys 中的具有以下最低权限。

- 要使用密 AWS KMS 钥环进行加密，您需要生成器密钥的 kms: GenerateDataKey 权限。您需要对密钥环中的所有其他密钥拥有 kms: encrypt 权限。 AWS KMS
- 要使用密钥环进行解密，您需要对密 AWS KMS 钥环中的至少一个密钥具有 kms: Decrypt 权限。 AWS KMS

- 要使用由密钥环组成的多密钥环进行加密，你需要获得生成器密 AWS KMS 钥环中生成器密钥的 kms: GenerateDataKey 权限。您需要对所有其他密钥环中的所有其他密钥具有 kms: encrypt 权限。AWS KMS
- 要使用非对称 RSA AWS KMS 密钥环进行加密，您不需要 kms: GenerateDataKey 或 kms: Encrypt，因为在创建密钥环时必须指定要用于加密的公钥材料。使用此密钥环加密时不会 AWS KMS 发出任何呼叫。要使用非对称 RSA 密 AWS KMS 钥环进行解密，你需要 kms: Decrypt 权限。

有关权限的详细信息 AWS KMS keys，请参阅《AWS Key Management Service 开发人员指南》中的 KMS 密钥访问和权限。

在 AWS KMS 钥匙圈 AWS KMS keys 中识别

一个 AWS KMS 钥匙圈可以包括一个或多 AWS KMS keys个。要在 AWS KMS 密钥环 AWS KMS key 中指定，请使用支持的 AWS KMS 密钥标识符。可用于在密钥环 AWS KMS key 中识别的密钥标识符因操作和语言实现而异。有关 AWS KMS key密钥标识符的详细信息，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。

作为最佳实践，请使用最适合您任务的密钥标识符。

- 在的加密密钥环中 AWS Encryption SDK for C，您可以使用密钥 ARN 或别名 ARN 来识别 KMS 密钥。在所有其他语言实施中，您可以使用密钥 ID、密钥 ARN、别名名称或别名 ARN加密数据。
- 在解密密钥环中，您必须使用密钥 ARN 以标识 AWS KMS keys。该要求适用于 AWS Encryption SDK的所有语言实施。有关更多信息，请参阅 选择包装密钥。
- 在用于加密和解密的密钥环中，您必须使用密钥 ARN 以标识 AWS KMS keys。该要求适用于 AWS Encryption SDK的所有语言实施。

如果您在加密密钥环中为 KMS 密钥指定别名名称或别名 ARN，则加密操作会将当前与该别名关联的密钥 ARN 保存在加密数据密钥的元数据中。它不会保存别名。更改别名不会影响用于解密加密数据密钥的 KMS 密钥。

创建密 AWS KMS 钥环

您可以为每个 AWS KMS 密钥环配置一个 AWS KMS key 或多个 AWS KMS keys 相同或不同的密钥环 AWS 账户 。 AWS 区域 AWS KMS keys 必须是对称加密 KMS 密钥 (SYMMETRIC_DEFAULT) 或非对称 RSA KMS 密钥。您也可以使用对称加密多区域 KMS 密钥。您可以在一个多重密钥环中使用一个或多个 AWS KMS 密钥环。

您可以创建用于加密和解密数据的密 AWS KMS 钥环，也可以创建专门用于加密或解密的 AWS KMS 密钥环。创建用于加密数据的 AWS KMS 密钥环时，必须指定生成器密钥，该密钥用于生成纯文本数据密钥并对其进行加密。 AWS KMS key 数据密钥在数学上与 KMS 密钥无关。然后，如果您愿意，则可以指定用于加密相同纯文本数据密钥的其他 AWS KMS keys 内容。要解密受此密钥环保护的加密字段，您使用的解密密钥环必须至少包含密钥环中 AWS KMS keys 定义的密钥环中的一个，或者不是。AWS KMS keys（没有的 AWS KMS 密钥环 AWS KMS keys 称为AWS KMS 发现密钥环。）

在除之外的 AWS Encryption SDK 语言实现中 AWS Encryption SDK for C，所有封装在加密密钥环或多密钥环中的密钥都必须能够加密数据密钥。如有任何包装密钥无法加密，此加密方法将失败。因此，调用方必须拥有密钥环中所有密钥的所需权限。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败。唯一的例外是 AWS Encryption SDK for C，加密操作会忽略标准发现密钥环，但是如果您单独或在多密钥环中指定多区域发现密钥环，则会失败。

以下示例使用生成器 AWS KMS 密钥和一个附加密钥创建密钥环。生成器密钥和其他密钥均为对称加密 KMS 密钥。这些示例使用密钥 ARNs来识别 KMS 密钥。这是用于加密的 AWS KMS 密钥环的最佳实践，也是用于解密的 AWS KMS 密钥环的要求。有关更多信息，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

C

要 AWS KMS key 在中的加密密钥环中识别 AWS Encryption SDK for C，请指定密钥 ARN 或别名 ARN。在解密密钥环中，您必须使用密钥 ARN。有关更多信息，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

有关完整的示例，请参阅 string.cpp。

```
const char * generator_key = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"

const char * additional_key = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"

struct aws_cryptosdk_keyring *kms_encrypt_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(generator_key,{additional_key});
```

C# / .NET

要在 AWS Encryption SDK 适用于.NET 的中创建包含一个或多个 KMS 密钥的密钥环，请使用`CreateAwsKmsMultiKeyring()`方法。此示例使用两个 AWS KMS 密钥。要指定 KMS 密钥，请仅使用 `Generator` 参数。指定其他 KMS 密钥的 `KmsKeyIds` 参数为可选参数。

此密钥环的输入不接受 AWS KMS 客户端。相反， AWS Encryption SDK 使用由密钥环中的 KMS 密钥表示的每个区域的默认 AWS KMS 客户端。例如，如果由Generator参数值标识的 KMS 密钥位于美国西部（俄勒冈）区域 (us-west-2)，则会为该us-west-2区域 AWS Encryption SDK 创建默认 AWS KMS 客户端。如果需要自定义 AWS KMS 客户端，请使用 CreateAwsKmsKeyring() 方法。

在.NET 中 AWS KMS key 为加密密钥环指定时，可以使用任何有效的密钥标识符：密钥 ID、密钥 AR N、别名或别名 ARN。 AWS Encryption SDK有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用版本 4。 .N AWS Encryption SDK ET 的 x 以及自定义 AWS KMS 客户端CreateAwsKmsKeyring()的方法。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

string generatorKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<string> additionalKeys = new List<string> { "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321" };

// Instantiate the keyring input object
var createEncryptKeyringInput = new CreateAwsKmsMultiKeyringInput
{
    Generator = generatorKey,
    KmsKeyIds = additionalKeys
};

var kmsEncryptKeyring = mpl.CreateAwsKmsMultiKeyring(createEncryptKeyringInput);
```

JavaScript Browser

在中 AWS KMS key 为加密密钥环指定时 AWS Encryption SDK for JavaScript，您可以使用任何有效的密钥标识符：密钥 ID、密钥 AR N、别名或别名 ARN。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

有关完整的示例，请参阅中存储库中的 kms_simple.ts。 AWS Encryption SDK for JavaScript GitHub

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const clientProvider = getClient(KMS, { credentials })
const generatorKeyId = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
const additionalKey = 'alias/exampleAlias'

const keyring = new KmsKeyringBrowser({
  clientProvider,
  generatorKeyId,
  keyIds: [additionalKey]
})
```

JavaScript Node.js

在中 AWS KMS key 为加密密钥环指定时 AWS Encryption SDK for JavaScript，您可以使用任何有效的密钥标识符：密钥 ID、密钥 AR N、别名或别名 ARN。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

有关完整的示例，请参阅中存储库中的 kms_simple.ts。 AWS Encryption SDK for JavaScript GitHub

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'
```

```
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const generatorKeyId = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

const additionalKey = 'alias/exampleAlias'

const keyring = new KmsKeyringNode({
  generatorKeyId,
  keyIds: [additionalKey]
})
```

Java

要使用一个或多个 AWS KMS 密钥创建密钥环，请使用`CreateAwsKmsMultiKeyring()`方法。此示例使用两个 KMS 密钥。要指定 KMS 密钥，请仅使用 `generator` 参数。指定其他 KMS 密钥的 `kmsKeyIds` 参数为可选参数。

此密钥环的输入不接受 AWS KMS 客户端。相反， AWS Encryption SDK 使用由密钥环中的 KMS 密钥表示的每个区域的默认 AWS KMS 客户端。例如，如果由`Generator`参数值标识的 KMS 密钥位于美国西部（俄勒冈）区域 (`us-west-2`)，则会为该`us-west-2`区域 AWS Encryption SDK 创建默认 AWS KMS 客户端。如果需要自定义 AWS KMS 客户端，请使用 `CreateAwsKmsKeyring()` 方法。

在中 AWS KMS key 为加密密钥环指定时 AWS Encryption SDK for Java，您可以使用任何有效的密钥标识符：密钥 ID、密钥 AR N、别名或别名 ARN。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

有关完整示例，请参阅中 AWS Encryption SDK for Java 存储库中的 BasicEncryptionKeyringExample GitHub.java。

```
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder().build();
final MaterialProviders materialProviders = MaterialProviders.builder()
            .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
            .build();

String generatorKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
```

```
List<String> additionalKey = Collections.singletonList("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");
// Create the keyring
final CreateAwsKmsMultiKeyringInput keyringInput =
 CreateAwsKmsMultiKeyringInput.builder()
        .generator(generatorKey)
        .kmsKeyIds(additionalKey)
        .build();
final IKeyring kmsKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);
```

Python

要使用一个或多个 AWS KMS 密钥创建密钥环，请使用`create_aws_kms_multi_keyring()`方法。此示例使用两个 KMS 密钥。要指定 KMS 密钥，请仅使用 `generator` 参数。指定其他 KMS 密钥的 `kms_key_ids` 参数为可选参数。

此密钥环的输入不接受 AWS KMS 客户端。相反， AWS Encryption SDK 使用由密钥环中的 KMS 密钥表示的每个区域的默认 AWS KMS 客户端。例如，如果由`generator`参数值标识的 KMS 密钥位于美国西部（俄勒冈）区域 (`us-west-2`)，则会为该`us-west-2`区域 AWS Encryption SDK 创建默认 AWS KMS 客户端。如果需要自定义 AWS KMS 客户端，请使用 `create_aws_kms_keyring()` 方法。

在中 AWS KMS key 为加密密钥环指定时 AWS Encryption SDK for Python，您可以使用任何有效的密钥标识符：密钥 ID、密钥 AR N、别名或别名 ARN。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`有关完整示例，请参阅中 AWS Encryption SDK for Python 存储库中的 aws_kms_multi_keyring_example.py GitHub。

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
```

```
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
kms_multi_keyring_input: CreateAwsKmsMultiKeyringInput =
 CreateAwsKmsMultiKeyringInput(
    generator="arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    kms_key_ids="arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"
)

kms_multi_keyring: IKeyring = mat_prov.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)
```

Rust

要使用一个或多个 AWS KMS 密钥创建密钥环，请使用`create_aws_kms_multi_keyring()`方法。此示例使用两个 KMS 密钥。要指定 KMS 密钥，请仅使用 `generator` 参数。指定其他 KMS 密钥的 `kms_key_ids` 参数为可选参数。

此密钥环的输入不接受 AWS KMS 客户端。相反， AWS Encryption SDK 使用由密钥环中的 KMS 密钥表示的每个区域的默认 AWS KMS 客户端。例如，如果由`generator`参数值标识的 KMS 密钥位于美国西部（俄勒冈）区域 (`us-west-2`)，则会为该`us-west-2`区域 AWS Encryption SDK 创建默认 AWS KMS 客户端。如果需要自定义 AWS KMS 客户端，请使用 `create_aws_kms_keyring()` 方法。

在 for Rust 中 AWS KMS key 为加密密钥环指定时，可以使用任何有效的密钥标识符：密钥 ID、密钥 ARN、别名或别名 ARN。 AWS Encryption SDK有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`有关完整的示例，请参阅上存储库 Rust 目录中的 aws_kms_keyring_example. rs。 aws-encryption-sdk GitHub

```
// Instantiate the AWS Encryption SDK client
```

```
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create the AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

kms_multi_keyring: IKeyring = mpl.create_aws_kms_multi_keyring(
    input=kms_multi_keyring_input
)
```

Go

要使用一个或多个 AWS KMS 密钥创建密钥环，请使用`create_aws_kms_multi_keyring()`方法。此示例使用两个 KMS 密钥。要指定 KMS 密钥，请仅使用 `generator` 参数。指定其他 KMS 密钥的 `kms_key_ids` 参数为可选参数。

此密钥环的输入不接受 AWS KMS 客户端。相反， AWS Encryption SDK 使用由密钥环中的 KMS 密钥表示的每个区域的默认 AWS KMS 客户端。例如，如果由`generator`参数值标识的 KMS 密钥位于美国西部（俄勒冈）区域 (`us-west-2`)，则会为该`us-west-2`区域 AWS

Encryption SDK 创建默认 AWS KMS 客户端。如果需要自定义 AWS KMS 客户端，请使用 create_aws_kms_keyring() 方法。

在 for Go 中 AWS KMS key 为加密密钥环指定时，您可以使用任何有效的密钥标识符：密钥 ID、密钥 ARN、别名或别名 ARN。 AWS Encryption SDK有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。REQUIRE_ENCRYPT_REQUIRE_DECRYPT

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
```

```
// Create the AWS KMS keyring
awsKmsMultiKeyringInput := mpltypes.CreateAwsKmsMultiKeyringInput{
    Generator: "&arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
    KmsKeyIds: []string{"arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"},
}
awsKmsMultiKeyring, err := matProv.CreateAwsKmsMultiKeyring(context.Background(),
 awsKmsMultiKeyringInput)
```

AWS Encryption SDK 还支持使用非对称 RSA KMS AWS KMS 密钥的密钥环。非对称 RSA AWS KMS 密钥环只能包含一个密钥对。

要使用非对称 RSA AWS KMS 密钥环进行加密，您不需要 kms: GenerateDataKey 或 kms: E ncrypt，因为在创建密钥环时必须指定要用于加密的公钥材料。使用此密钥环进行加密时不会调用任何 AWS KMS 。要使用非对称 RSA 密 AWS KMS 钥环进行解密，你需要 kms: Decrypt 权限。

> Note
>
> 要创建使用非对称 RSA KMS 密钥的密钥 AWS KMS 环，必须使用以下编程语言实现之一：
>
> - 版本 3。 的 x AWS Encryption SDK for Java
> - 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
> - 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
> - 版本 1。 x 的 fo r AWS Encryption SDK Rust
> - 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

以下示例使用该`CreateAwsKmsRsaKeyring`方法创建带有非对称 RSA KMS AWS KMS 密钥的密钥环。要创建非对称 RSA AWS KMS 密钥环，请提供以下值。

- `kmsClient`: 创建新 AWS KMS 客户端
- `kmsKeyID`: 用于识别您的非对称 RSA KMS 密钥的密钥 ARN
- `publicKey`: a 来 ByteBuffer 自 UTF-8 编码的 PEM 文件，该文件代表你传递给的密钥的公钥 `kmsKeyID`

- `encryptionAlgorithm`: 加密算法必须是`RSAES_OAEP_SHA_256`或 `RSAES_OAEP_SHA_1`

C# / .NET

要创建非对称 RSA AWS KMS 密钥环，您必须提供来自非对称 RSA KMS 密钥的公钥和私钥 ARN。公有密钥必须采用 PEM 编码。以下示例使用非对称 RSA AWS KMS 密钥对创建密钥环。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

var publicKey = new MemoryStream(Encoding.UTF8.GetBytes(AWS KMS RSA public key));

// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = AWS KMS RSA private key ARN,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};

// Create the keyring
var kmsRsaKeyring = mpl.CreateAwsKmsRsaKeyring(createKeyringInput);
```

Java

要创建非对称 RSA AWS KMS 密钥环，您必须提供来自非对称 RSA KMS 密钥的公钥和私钥 ARN。公有密钥必须采用 PEM 编码。以下示例使用非对称 RSA AWS KMS 密钥对创建密钥环。

```
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder()
        // Specify algorithmSuite without asymmetric signing here
        //
        // ALG_AES_128_GCM_IV12_TAG16_NO_KDF("0x0014"),
        // ALG_AES_192_GCM_IV12_TAG16_NO_KDF("0x0046"),
        // ALG_AES_256_GCM_IV12_TAG16_NO_KDF("0x0078"),
        // ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256("0x0114"),
        // ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256("0x0146"),
        // ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256("0x0178")

 .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256)
```

```
        .build();

final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

// Create a KMS RSA keyring.
//    This keyring takes in:
//     - kmsClient
//     - kmsKeyId: Must be an ARN representing an asymmetric RSA KMS key
//     - publicKey: A ByteBuffer of a UTF-8 encoded PEM file representing the public
//                  key for the key passed into kmsKeyId
//     - encryptionAlgorithm: Must be either RSAES_OAEP_SHA_256 or RSAES_OAEP_SHA_1
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
        CreateAwsKmsRsaKeyringInput.builder()
                .kmsClient(KmsClient.create())
                .kmsKeyId(rsaKeyArn)
                .publicKey(publicKey)
                .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
                .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Python

要创建非对称 RSA AWS KMS 密钥环，您必须提供来自非对称 RSA KMS 密钥的公钥和私钥 ARN。公有密钥必须采用 PEM 编码。以下示例使用非对称 RSA AWS KMS 密钥对创建密钥环。

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers library
```

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS keyring
keyring_input: CreateAwsKmsRsaKeyringInput = CreateAwsKmsRsaKeyringInput(
    public_key="public_key",
    kms_key_id="kms_key_id",
    encryption_algorithm="RSAES_OAEP_SHA_256",
    kms_client=kms_client
)

kms_rsa_keyring: IKeyring = mat_prov.create_aws_kms_rsa_keyring(
    input=keyring_input
)
```

Rust

要创建非对称 RSA AWS KMS 密钥环，您必须提供来自非对称 RSA KMS 密钥的公钥和私钥 ARN。公有密钥必须采用 PEM 编码。以下示例使用非对称 RSA AWS KMS 密钥对创建密钥环。

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

```
// Create the AWS KMS keyring
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(kms_key_id)
    .public_key(aws_smithy_types::Blob::new(public_key))
 
 .encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(kms_client)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})
```

```
// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create the AWS KMS keyring
awsKmsRSAKeyringInput := mpltypes.CreateAwsKmsRsaKeyringInput{
    KmsClient:           kmsClient,
    KmsKeyId:            kmsKeyID,
    PublicKey:           kmsPublicKey,
    EncryptionAlgorithm: kmstypes.EncryptionAlgorithmSpecRsaesOaepSha256,
}
awsKmsRSAKeyring, err := matProv.CreateAwsKmsRsaKeyring(context.Background(),
 awsKmsRSAKeyringInput)
if err != nil {
    panic(err)
}
```

使用 AWS KMS 发现密钥环

解密时，最佳做法是指定 AWS Encryption SDK 可以使用的包装密钥。要遵循此最佳实践，请使用 AWS KMS 解密密钥环，将 AWS KMS 封装密钥限制在您指定的密钥范围内。但是，您也可以创建AWS KMS 发现密钥环，即不指定任何包装 AWS KMS 密钥的密钥环。

为 AWS KMS 多区域密钥 AWS Encryption SDK 提供了标准 AWS KMS 发现密钥环和发现密钥环。有关多区域密钥与 AWS Encryption SDK共用的信息，请参阅 使用多区域 AWS KMS keys。

由于 Discovery 密钥环未指定任何包装密钥，因此 Discovery 密钥无法加密数据。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败。唯一的例外是 AWS Encryption SDK for

C，加密操作会忽略标准发现密钥环，但是如果您单独或在多密钥环中指定多区域发现密钥环，则会失败。

解密时，发现密钥环允许使用加密后的密钥要求 AWS KMS 解密任何加密的数据密钥，无论谁拥有或有权访问 AWS KMS key 该密钥。 AWS Encryption SDK AWS KMS key只有在调用方拥有 AWS KMS key的 `kms:Decrypt` 权限时，调用才会成功。

> ⚠ Important
>
> 如果您在解密多密钥环中包含 AWS KMS 发现密钥环，则发现密钥环将覆盖多密钥环中其他密钥环中指定的所有 KMS 密钥限制。多重密钥环的行为类似于限制最少的密钥环。单独使用或在多重密钥环中使用时， AWS KMS Discovery 密钥环对加密无效。

为方便起见， AWS Encryption SDK 提供了 AWS KMS 发现钥匙圈。不过，出于以下原因，建议尽可能使用更受限制的密钥环。

- 真实性 — AWS KMS 发现密钥环可以使用任何 AWS KMS key 用于加密加密消息中数据密钥的密钥，这样调用者就可以使用该密钥 AWS KMS key 进行解密。这可能不是呼叫 AWS KMS key 者打算使用的。例如，其中一个加密的数据密钥可能是在任何人都无法 AWS KMS key 使用的安全性下加密的。
- 延迟和性能 — AWS KMS 发现密钥环可能比其他密钥环慢得多，因为他们 AWS Encryption SDK 会尝试解密所有加密的数据密钥，包括其他 AWS 账户 和区域 AWS KMS keys 中加密的数据密钥，而调用者无权使用这些密钥进行解密。 AWS KMS keys

如果您使用发现密钥环，我们建议您使用发现过滤器将可用的 KMS 密钥限制为指定 AWS 账户 和分区中的密钥。 AWS Encryption SDK版本 1.7.x 及更高版本支持发现筛选条件。如需帮助查找您的账户 ID 和分区，请参阅中的您的 AWS 账户 标识符和 ARN 格式。AWS 一般参考

以下代码使用发现过滤器实例化 AWS KMS 发现密钥环，该过滤器将 AWS Encryption SDK 可用的 KMS 密钥限制为`aws`分区和 111122223333 示例账户中的密钥。

在使用此代码之前，请将示例 AWS 账户 和分区值替换为 AWS 账户 和分区的有效值。如果您的 KMS 密钥位于中国区域，请使用 `aws-cn` 分区值。如果您的 KMS 密钥位于 AWS GovCloud (US) Regions，请使用 `aws-us-gov` 分区值。对于所有其他 AWS 区域，请使用 `aws` 分区值。

C

有关完整的示例，请参阅：kms_discovery.cpp。

```
std::shared_ptr<KmsKeyring::> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .Build());

struct aws_cryptosdk_keyring *kms_discovery_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()
       .BuildDiscovery(discovery_filter));
```

C# / .NET

以下示例使用适用于 .NET 的 AWS Encryption SDK 版本 4.x。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// In a discovery keyring, you specify an AWS KMS client and a discovery filter,
// but not a AWS KMS key
var kmsDiscoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = account,
        Partition = "aws"
    }
};

var kmsDiscoveryKeyring =
 mpl.CreateAwsKmsDiscoveryKeyring(kmsDiscoveryKeyringInput);
```

JavaScript Browser

在中 JavaScript，必须明确指定发现属性。

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
```

```
  KmsKeyringBrowser,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const clientProvider = getClient(KMS, { credentials })

const discovery = true
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

JavaScript Node.js

在中 JavaScript，必须明确指定发现属性。

以下示例使用`buildClient`函数来指定[默认的承诺策略](#)`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。您也可以使用`buildClient`来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 [the section called “限制加密数据密钥”](#)。

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const discovery = true

const keyring = new KmsKeyringNode({
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})
```

Java

```
// Create discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Python

```
# Instantiate the AWS Encryption SDK
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Create a boto3 client for AWS KMS
kms_client = boto3.client('kms', region_name=aws_region)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS discovery keyring
discovery_keyring_input: CreateAwsKmsDiscoveryKeyringInput =
 CreateAwsKmsDiscoveryKeyringInput(
    kms_client=kms_client,
```

```
    discovery_filter=DiscoveryFilter(
        account_ids=[aws_account_id],
        partition="aws"
    )
)

discovery_keyring: IKeyring = mat_prov.create_aws_kms_discovery_keyring(
    input=discovery_keyring_input
)
```

Rust

```
// Instantiate the AWS Encryption SDK
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create a AWS KMS client.
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);


// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![aws_account_id.to_string()])
    .partition("aws".to_string())
    .build()?;

// Create the AWS KMS discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_discovery_keyring()
    .kms_client(kms_client.clone())
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
```

```
    panic(err)
}

// Create discovery filter
discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{kmsKeyAccountID},
    Partition:  "aws",
}
awsKmsDiscoveryKeyringInput := mpltypes.CreateAwsKmsDiscoveryKeyringInput{
    KmsClient:       kmsClient,
    DiscoveryFilter: &discoveryFilter,
}
awsKmsDiscoveryKeyring, err :=
 matProv.CreateAwsKmsDiscoveryKeyring(context.Background(),
 awsKmsDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

使用 AWS KMS 区域发现密钥环

AWS KMS 区域发现密钥环是一种不指定 KMS 密钥 ARNs 的密钥环。相反，它允许仅使用 KMS 密钥进行解密。 AWS Encryption SDK AWS 区域

使用 AWS KMS 区域发现密钥环解密时，会 AWS Encryption SDK 解密在指定项下加密的所有加密数据密钥。 AWS KMS key AWS 区域要成功，调用者必须拥有对指定数据密钥 AWS KMS keys 中至少一个加密数据密钥 AWS 区域 的`kms:Decrypt`权限。

与其他 Discovery 密钥环相同，Regional Discovery 密钥环对加密无效。该密钥环仅在解密加密消息时适用。如果您在用于加密和解密的多重密钥环中使用 Regional Discovery 密钥环，则该密钥环仅在解密时有效。如果您单独或在多重密钥环中使用多区域 Discovery 密钥环加密数据，加密操作将失败。

> **Important**
>
> 如果您在解密多密钥环中包含 AWS KMS 区域发现密钥环，则区域发现密钥环将覆盖多密钥环中其他密钥环中指定的所有 KMS 密钥限制。多重密钥环的行为类似于限制最少的密钥环。单独使用或在多重密钥环中使用时， AWS KMS Discovery 密钥环对加密无效。

AWS Encryption SDK for C 尝试中的区域发现密钥环仅使用指定区域中的 KMS 密钥进行解密。在 AWS Encryption SDK for JavaScript 和中 AWS Encryption SDK 为.NET 使用发现密钥环时，需要在 AWS KMS 客户端上配置区域。这些 AWS Encryption SDK 实现不会按区域过滤 KMS 密钥，但对指定区域之外的 KMS 密钥的解密请求 AWS KMS 会失败。

如果您使用发现密钥环，我们建议您使用发现过滤器将解密中使用的 KMS 密钥限制为指定 AWS 账户和分区中的密钥。 AWS Encryption SDK版本 1.7.x 及更高版本支持发现筛选条件。

例如，以下代码使用发现过滤器创建 AWS KMS 区域发现密钥环。此密钥环仅限 AWS Encryption SDK 于美国西部（俄勒冈）地区 (us-west-2) 账户 111122223333 中的 KMS 密钥。

C

要在可正常使用的示例中查看此密钥环和 `create_kms_client` 方法，请参阅 kms_discovery.cpp。

```
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .Build());

struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()

 .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery(discovery_filter))
```

C# / .NET

f AWS Encryption SDK or .NET 没有专用的区域发现密钥环。但是，您可以使用多种方法将解密时使用的 KMS 密钥限制在特定区域。

限制发现密钥环中区域的最有效方法是使用发现密钥环，即使您仅使用单区域密钥对数据进行了加密。 multi-Region-aware当遇到单区域密钥时，密 multi-Region-aware钥环不使用任何多区域功能。

`CreateAwsKmsMrkDiscoveryKeyring()` 方法返回的密钥环会在调用 AWS KMS之前按区域筛选 KMS 密钥。 AWS KMS 只有当加密的数据密钥由对象中的`Region`参数指定的区域中的 KMS 密钥加密时，它才会向发送解密请求。`CreateAwsKmsMrkDiscoveryKeyringInput`

以下示例使用适用于 .NET 的 AWS Encryption SDK 版本 4.x。

```
// Instantiate the AWS Encryption SDK and material providers
```

```
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// Create the discovery filter
var filter = DiscoveryFilter = new DiscoveryFilter
{
    AccountIds = account,
    Partition = "aws"
};

var regionalDiscoveryKeyringInput = new CreateAwsKmsMrkDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    Region = RegionEndpoint.USWest2,
    DiscoveryFilter = filter
};

var kmsRegionalDiscoveryKeyring =
 mpl.CreateAwsKmsMrkDiscoveryKeyring(regionalDiscoveryKeyringInput);
```

您还可以 AWS 区域 通过在 AWS KMS 客户端实例中指定区域来将 KMS 密钥限制为特定的密钥 (AmazonKeyManagementServiceClient)。但是，与使用 multi-Region-aware发现密钥环相比，这种配置效率较低，而且成本可能更高。for .NET 不是在调用之前按区域筛选 KMS 密钥 AWS KMS，而是调 AWS KMS 用每个加密的数据密钥（直到它解密一个），并依靠它 AWS KMS 来将其使用的 KMS 密钥限制在指定区域。 AWS Encryption SDK

以下示例使用适用于 .NET 的 AWS Encryption SDK 版本 4.x。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

List<string> account = new List<string> { "111122223333" };

// Create the discovery filter,
// but not a AWS KMS key
var createRegionalDiscoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(RegionEndpoint.USWest2),
    DiscoveryFilter = new DiscoveryFilter()
    {
```

```
        AccountIds = account,
        Partition = "aws"
    }
};

var kmsRegionalDiscoveryKeyring =
 mlp.CreateAwsKmsDiscoveryKeyring(createRegionalDiscoveryKeyringInput);
```

JavaScript Browser

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})
```

JavaScript Node.js

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

要在工作示例中查看此密钥环和limitRegions函数，请参阅 km s_regional_discovery.ts。

```
import {
  KmsKeyringNode,
```

```
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333'], partition: 'aws' }
})
```

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .regions("us-west-2")
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Python

```
# Instantiate the AWS Encryption SDK
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Create a boto3 client for AWS KMS
kms_client = boto3.client('kms', region_name=aws_region)

# Optional: Create an encryption context
```

```
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS regional discovery keyring
regional_discovery_keyring_input: CreateAwsKmsMrkDiscoveryKeyringInput = \
    CreateAwsKmsMrkDiscoveryKeyringInput(
        kms_client=kms_client,
        region=mrk_replica_decrypt_region,
        discovery_filter=DiscoveryFilter(
            account_ids=[111122223333],
            partition="aws"
        )
)

    regional_discovery_keyring: IKeyring =
 mat_prov.create_aws_kms_mrk_discovery_keyring(
        input=regional_discovery_keyring_input
    )
```

Rust

```
// Instantiate the AWS Encryption SDK
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
```

```
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS client
let decrypt_kms_config = aws_sdk_kms::config::Builder::from(&sdk_config)
    .region(Region::new(mrk_replica_decrypt_region.clone()))
    .build();
let decrypt_kms_client = aws_sdk_kms::Client::from_conf(decrypt_kms_config);


// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .account_ids(vec![aws_account_id.to_string()])
    .partition("aws".to_string())
    .build()?;

// Create the regional discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_mrk_discovery_keyring()
    .kms_client(decrypt_kms_client)
    .region(mrk_replica_decrypt_region)
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
```

```
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Create discovery filter
discoveryFilter := mpltypes.DiscoveryFilter{
    AccountIds: []string{awsAccountID},
    Partition:  "aws",
}

// Create the regional discovery keyring
awsKmsMrkDiscoveryInput := mpltypes.CreateAwsKmsMrkDiscoveryKeyringInput{
    KmsClient:       kmsClient,
    Region:          alternateRegionMrkKeyRegion,
    DiscoveryFilter: &discoveryFilter,
}
awsKmsMrkDiscoveryKeyring, err :=
 matProv.CreateAwsKmsMrkDiscoveryKeyring(context.Background(),
 awsKmsMrkDiscoveryInput)
if err != nil {
```

```
    panic(err)
}
```

AWS Encryption SDK for JavaScript 还导出了 Node.js 和浏览器的`excludeRegions`函数。此函数会创建一个 AWS KMS 区域发现密钥环，该密钥环省略 AWS KMS keys 了特定区域。以下示例创建了一个 AWS KMS 区域发现密钥环，除了美国东部（弗吉尼亚北部）(us-east-1) AWS 区域 之外，该密钥环可在账户 111122223333 中使用 AWS KMS keys 。

AWS Encryption SDK for C 没有类似的方法，但您可以通过创建自定义方法来实现。ClientSupplier

该示例显示了 Node.js 的代码。

```
const discovery = true
const clientProvider = excludeRegions(['us-east-1'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

AWS KMS 分层钥匙圈

使用 AWS KMS 分层密钥环，您可以在对称加密 KMS 密钥下保护您的加密材料，而无需在 AWS KMS 每次加密或解密数据时都调用。对于需要最大限度地减少调用的应用程序以及可以在不违反其安全要求的情况下重复使用某些加密材料的应用程序来说，这是一个不错的选择。 AWS KMS

分层密钥环是一种加密材料缓存解决方案，它使用 AWS KMS 保存在Amazon DynamoDB表中的受保护分支密钥，然后在本地缓存用于加密和解密操作的分支密钥材料，从而减少 AWS KMS 调用次数。DynamoDB 表用作管理和保护分支密钥的密钥存储。其存储活动分支密钥和分支密钥的所有先前版本。活动分支密钥为最新分支密钥版本。分层密钥环使用唯一的数据密钥来加密每条消息，并对每个加密请求的每个数据加密密钥进行加密，并使用从活动分支密钥派生的唯一包装密钥对每个数据加密密钥进行加密。分层密钥环依赖在活动分支密钥及其派生包装密钥之间建立的层次结构。

分层密钥环通常使用各分支密钥版本满足多个请求。但是您可以控制活动分支密钥的重复使用程度，并确定活动分支密钥的轮换频率。在您轮换之前，分支密钥的活动版本会一直处于活动状态。活动分支密钥的先前版本不会用于执行加密操作，但仍可查询并用于解密操作。

当您实例化分层密钥环时，分层密钥环会创建本地缓存。您可以指定缓存限制，该限制定义了分支密钥材料在过期并从缓存中移出之前存储在本地缓存中的最长时间。首次在操作中指定 a `branch-key-id`

时，分层密钥环会 AWS KMS 调用解密分支密钥并组装分支密钥材料。然后，分支密钥材料存储在本地缓存中，并重复用于所有指定 branch-key-id 的加密和解密操作，直至缓存限制到期。将分支密钥材料存储在本地缓存中可以减少 AWS KMS 调用。例如，假设缓存限制为 15 分钟。如果您在该缓存限制内执行 10,000 次加密操作，则传统 AWS KMS 密钥环需要进行 10,000 次 AWS KMS 调用才能满足 10,000 次加密操作。如果您有一个处于活动状态branch-key-id，则分层密钥环只需要进行一次 AWS KMS 调用即可满足 10,000 个加密操作。

本地缓存将加密材料与解密材料分开。加密材料由活动分支密钥组合而成，并在缓存限制到期之前重复用于所有加密操作。解密材料是根据在加密字段的元数据中标识的分支密钥 ID 和版本汇编而成的，在缓存限制到期之前，它们可以重复用于与分支密钥 ID 和版本相关的所有解密操作。本地缓存可以一次存储同一个分支密钥的多个版本。将本地缓存配置为使用时branch key ID supplier，它还可以同时存储来自多个活动分支密钥的分支密钥材料。

中所有提及分层密钥环的 AWS Encryption SDK 内容均指 AWS KMS 分层密钥环。

编程语言兼容性

以下编程语言和版本支持分层密钥环：

- 版本 3。 的 x AWS Encryption SDK for Java
- 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
- 版本 4。 的 x AWS Encryption SDK for Python，当与可选的 MPL 依赖项一起使用时。
- 版本 1。 x 的 fo r AWS Encryption SDK Rust
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

主题

- 工作原理
- 先决条件
- 所需的权限
- 选择缓存
- 创建分层密钥环

工作原理

以下演练描述了分层密钥环如何汇编加密和解密材料，以及密钥环对加密和解密操作的不同调用。有关包装密钥派生和明文数据密钥加密过程的技术详细信息，请参阅 AWS KMS 分层密钥环技术详细信息。

加密并签名

以下演练描述了分层密钥环如何汇编加密材料并派生出唯一的包装密钥。

1. 加密方法要求分层密钥环提供加密材料。密钥环生成明文数据密钥，然后检查本地缓存中是否存在有效分支材料可供生成包装密钥。如果存在有效的分支密钥材料，则密钥环将进入步骤 4。
2. 如果没有有效的分支密钥材料，则分层密钥环会在密钥库中查询活动分支密钥。
    a. 密钥库调用 AWS KMS 解密活动分支密钥并返回纯文本活动分支密钥。标识活动分支密钥的数据会进行序列化，以便在解密调用 AWS KMS时提供额外验证数据。
    b. 密钥库返回纯文本分支密钥和标识该密钥的数据，例如分支密钥版本。
3. 分层密钥环汇编分支密钥材料（明文分支密钥和分支密钥版本），并将其副本存储在本地缓存中。
4. 分层密钥环从明文分支密钥和一个 16 字节的随机加密盐中派生出唯一的包装密钥。其使用派生包装密钥加密明文数据密钥的副本。

此加密方法使用加密材料加密数据。有关更多信息，请参阅如何 AWS Encryption SDK 加密数据。

解密并验证

以下演练描述了分层密钥环如何组装解密材料并解密加密数据密钥。

1. 该解密方法标识来自加密消息的加密数据密钥，并将其传递给分层密钥环。
2. 分层密钥环反序列化标识加密数据密钥的数据，包括分支密钥版本、16 字节的加密盐以及其他描述数据密钥加密方式的信息。

    有关更多信息，请参阅 AWS KMS 分层钥匙圈技术细节。
3. 分层密钥环会检查本地缓存中是否存在与步骤 2 标识的分支密钥版本相匹配的有效分支密钥材料。如果存在有效分支密钥材料，则密钥环将进入步骤 6。
4. 如果没有有效的分支密钥材料，则分层密钥环会在密钥库中查询与步骤 2 中确定的分支密钥版本相匹配的分支密钥。

a. 密钥库调用 AWS KMS 解密分支密钥并返回纯文本活动分支密钥。标识活动分支密钥的数据会进行序列化，以便在解密调用 AWS KMS时提供额外验证数据。

b. 密钥库返回纯文本分支密钥和标识该密钥的数据，例如分支密钥版本。

5. 分层密钥环汇编分支密钥材料（明文分支密钥和分支密钥版本），并将其副本存储在本地缓存中。
6. 分层密钥环使用汇编的分支密钥材料和步骤 2 标识的 16 字节加密盐重现加密数据密钥的唯一包装密钥。
7. 分层密钥环使用重现的包装密钥解密数据密钥并返回明文数据密钥。

该解密方法使用解密材料和明文数据密钥解密加密消息。有关更多信息，请参阅如何 AWS Encryption SDK 解密加密邮件。

先决条件

在创建和使用分层密钥环之前，请确保满足以下先决条件。

- 您或您的密钥库管理员已创建密钥库并创建了至少一个有效的分支密钥。
- 您已经配置了密钥存储操作。

> Note
>
> 如何配置密钥存储操作决定了您可以执行的操作以及分层密钥环可以使用哪些 KMS 密钥。有关更多信息，请参阅密钥存储操作。

- 您拥有访问和使用密钥库和分支密钥所需的 AWS KMS 权限。有关更多信息，请参阅 the section called “所需的权限”。
- 您已经查看了支持的缓存类型并配置了最适合您需求的缓存类型。有关更多信息，请参阅 the section called “选择缓存”。

所需的权限

AWS Encryption SDK 不需要 AWS 账户 ，也不依赖于任何一个 AWS 服务。但是，要使用分层密钥环，您需要对 AWS 账户 密钥库中的对称加密 AWS KMS key具有以下最低权限。

- 要使用分层密钥环加密和解密数据，你需要 kms: Decrypt。
- 要创建和轮换分支密钥，你需要 k ms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt。

有关控制对分支密钥和密钥库的访问权限的更多信息，请参阅the section called “实施最低权限”。

选择缓存

分层密钥环 AWS KMS 通过在本地缓存加密和解密操作中使用的分支密钥材料来减少调用的次数。在创建分层密钥环之前，您需要决定要使用的缓存类型。您可以使用默认缓存或自定义缓存以最适合您的需求。

分层密钥环支持以下缓存类型：

- the section called “默认缓存”
- the section called “MultiThreaded 缓存”
- the section called “StormTracking 缓存”
- the section called “共享缓存”

> **Important**
>
> 所有支持的缓存类型都旨在支持多线程环境。
> 但是，与一起使用时 AWS Encryption SDK for Python，分层密钥环不支持多线程环境。
> 有关更多信息，请参阅 aws-cryptographic-material-providers-Library 存储库中的 Pyt hon README.rst 文件。 GitHub

默认缓存

对于大多数用户而言，默认缓存可满足其线程要求。默认缓存用于支持超多线程环境。当分支密钥材料条目过期时，默认缓存会 AWS KMS 提前 10 秒通知一个线程分支密钥材料条目将过期，从而防止多个线程调用。这样可以确保只有一个线程向发送刷新缓存的请求。 AWS KMS

Default 和 StormTracking 缓存支持相同的线程模型，但您只需要指定入口容量即可使用 Default 缓存。要进行更精细的缓存自定义，请使用。the section called “StormTracking 缓存”

除非要自定义可以存储在本地缓存中的分支密钥材料条目的数量，否则在创建分层密钥环时无需指定缓存类型。如果未指定缓存类型，则分层密钥环使用默认缓存类型并将条目容量设置为 1000。

要自定义默认缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。

Java

```
.cache(CacheType.builder()
        .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

C# / .NET

```
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

Python

```
default_cache = CacheTypeDefault(
    value=DefaultCache(
        entry_capacity=100
    )
)
```

Rust

```
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);
```

Go

```
cache := mpltypes.CacheTypeMemberDefault{
  Value: mpltypes.DefaultCache{
   EntryCapacity: 100,
  },
 }
```

MultiThreaded 缓存

MultiThreaded 缓存可在多线程环境中安全使用，但它不提供任何可最大限度减少 AWS KMS 或 Amazon DynamoDB 调用的功能。因此，当分支密钥材料条目到期时，所有线程均将同时收到通知。这可能会导致多次 AWS KMS 调用刷新缓存。

要使用 MultiThreaded 缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。
- 条目修剪尾部大小：定义在达到条目容量时要修剪的条目数量。

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())
```

C# / .NET

```
CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

Python

```
multithreaded_cache = CacheTypeMultiThreaded(
    value=MultiThreadedCache(
        entry_capacity=100,
        entry_pruning_tail_size=1
    )
)
```

Rust

```
CacheType::MultiThreaded(
            MultiThreadedCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .build()?)
```

Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberMultiThreaded{
  Value: mpltypes.MultiThreadedCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
  },
 }
```

StormTracking 缓存

StormTracking 缓存旨在支持大量多线程环境。当分支密钥材料条目过期时， StormTracking 缓存会提前通知一个线程分支密钥材料条目即将过期，从而防止多个线程调用 AWS KMS 。这样可以确保只有一个线程向发送刷新缓存的请求。 AWS KMS

要使用 StormTracking 缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。

  默认值：1000 个条目
- 条目修剪尾部大小：定义一次要修剪的分支密钥材料条目的数量。

  默认值：1 个条目
- 宽限期：定义在到期前尝试刷新分支密钥材料的秒数。

  默认值：10 秒
- 宽限间隔：定义两次尝试刷新分支密钥材料间隔的秒数。

  默认值：1 秒

- 扇出：定义可以同时尝试刷新分支密钥材料的次数。

  默认值：20 次尝试
- 传输中生存时间（TTL）：定义在分支密钥材料刷新尝试超时之前的秒数。每当缓存为响应 GetCacheEntry 而返回 NoSuchEntry 时，分支密钥均视为传输中，直至相同密钥与 PutCache 条目一起写入。

  默认值：10 秒
- 睡眠：定义超过时线程应休眠的毫秒数。fanOut

  默认值：20 毫秒

Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

C# / .NET

```
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 10,
        SleepMilli = 20
    }
};
```

Python

```
storm_tracking_cache = CacheTypeStormTracking(
    value=StormTrackingCache(
        entry_capacity=100,
        entry_pruning_tail_size=1,
        fan_out=20,
        grace_interval=1,
        grace_period=10,
        in_flight_ttl=10,
        sleep_milli=20
    )
)
```

Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberStormTracking{
  Value: mpltypes.StormTrackingCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
   GraceInterval:        1,
   GracePeriod:          10,
   FanOut:               20,
   InFlightTTL:          10,
   SleepMilli:           20,
  },
 }
```

共享缓存

默认情况下，每次实例化密钥环时，分层密钥环都会创建一个新的本地缓存。但是，共享缓存允许您在多个分层密钥环之间共享缓存，从而有助于节省内存。共享缓存不是为您实例化的每个分层密钥环创建新的加密材料缓存，而是在内存中只存储一个缓存，供所有引用它的分层密钥环使用。共享缓存可避免在密钥环之间重复加密材料，从而帮助优化内存使用率。相反，分层密钥环可以访问相同的底层缓存，从而减少总体内存占用。

创建共享缓存时，仍需要定义缓存类型。您可以指定the section called “默认缓存”the section called “MultiThreaded 缓存”、或the section called “StormTracking 缓存”作为缓存类型，也可以替换任何兼容的自定义缓存。

分区

多个分层密钥环可以使用单个共享缓存。使用共享缓存创建分层密钥环时，可以定义可选的分区 ID。分区 ID 可区分哪个分层密钥环正在写入缓存。如果两个分层密钥环引用相同的分区 ID 和分支密钥 IDlogical key store name，则两个密钥环将在缓存中共享相同的缓存条目。如果您创建了两个具有相同共享缓存但分区 ID 不同的分层密钥环，则每个密钥环只能从共享缓存中自己指定的分区访问缓存条目。分区充当共享缓存中的逻辑分区，允许每个分层密钥环在自己的指定分区上独立运行，而不会干扰存储在另一个分区中的数据。

如果您打算重复使用或共享分区中的缓存条目，则必须定义自己的分区 ID。当您将分区 ID 传递给分层密钥环时，密钥环可以重复使用共享缓存中已存在的缓存条目，而不必再次检索和重新授权分支密钥材料。如果您未指定分区 ID，则每次实例化分层密钥环时，都会自动为密钥环分配一个唯一的分区 ID。

以下过程演示如何创建默认缓存类型的共享缓存并将其传递给分层密钥环。

1. 使用材料提供者库 `CryptographicMaterialsCache` (MPL) 创建 (CMC)。

   Java

   ```
   // Instantiate the MPL
   final MaterialProviders matProv =
       MaterialProviders.builder()
           .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
           .build();

   // Create a CacheType object for the Default cache
   final CacheType cache =
       CacheType.builder()
   ```

```
        .Default(DefaultCache.builder().entryCapacity(100).build())
        .build();

// Create a CMC using the default cache
final CreateCryptographicMaterialsCacheInput cryptographicMaterialsCacheInput =
    CreateCryptographicMaterialsCacheInput.builder()
        .cache(cache)
        .build();

final ICryptographicMaterialsCache sharedCryptographicMaterialsCache =
    matProv.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

C# / .NET

```
// Instantiate the MPL
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create a CacheType object for the Default cache
var cache = new CacheType { Default = new DefaultCache{EntryCapacity = 100} };

// Create a CMC using the default cache
var cryptographicMaterialsCacheInput = new
 CreateCryptographicMaterialsCacheInput {Cache = cache};

var sharedCryptographicMaterialsCache =
 materialProviders.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

Python

```
# Instantiate the MPL
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create a CacheType object for the default cache
cache: CacheType = CacheTypeDefault(
    value=DefaultCache(
        entry_capacity=100,
    )
)

# Create a CMC using the default cache
cryptographic_materials_cache_input = CreateCryptographicMaterialsCacheInput(
```

```
    cache=cache,
)

shared_cryptographic_materials_cache =
 mat_prov.create_cryptographic_materials_cache(
    cryptographic_materials_cache_input
)
```

Rust

```
// Instantiate the MPL
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create a CacheType object for the default cache
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);

// Create a CMC using the default cache
let shared_cryptographic_materials_cache: CryptographicMaterialsCacheRef = mpl.
    create_cryptographic_materials_cache()
    .cache(cache)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
)

// Instantiate the MPL
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
```

```
}

// Create a CacheType object for the default cache
cache := mpltypes.CacheTypeMemberDefault{
    Value: mpltypes.DefaultCache{
        EntryCapacity: 100,
    },
}

// Create a CMC using the default cache
cmcCacheInput := mpltypes.CreateCryptographicMaterialsCacheInput{
    Cache: &cache,
}
sharedCryptographicMaterialsCache, err :=
 matProv.CreateCryptographicMaterialsCache(context.Background(), cmcCacheInput)
if err != nil {
    panic(err)
}
```

2. 为共享缓存创建CacheType对象。

   将sharedCryptographicMaterialsCache您在步骤 1 中创建的传递给新CacheType对象。

   Java

```
// Create a CacheType object for the sharedCryptographicMaterialsCache
final CacheType sharedCache =
    CacheType.builder()
        .Shared(sharedCryptographicMaterialsCache)
        .build();
```

   C# / .NET

```
// Create a CacheType object for the sharedCryptographicMaterialsCache
var sharedCache = new CacheType { Shared = sharedCryptographicMaterialsCache };
```

   Python

```
# Create a CacheType object for the shared_cryptographic_materials_cache
shared_cache: CacheType = CacheTypeShared(
    value=shared_cryptographic_materials_cache
)
```

Rust

```
// Create a CacheType object for the shared_cryptographic_materials_cache
let shared_cache: CacheType =
 CacheType::Shared(shared_cryptographic_materials_cache);
```

Go

```
// Create a CacheType object for the shared_cryptographic_materials_cache
shared_cache :=
 mpltypes.CacheTypeMemberShared{sharedCryptographicMaterialsCache}
```

3. 将步骤 2 中的`sharedCache`对象传递到分层密钥环。

   使用共享缓存创建分层密钥环时，可以选择定义一个`partitionID`以在多个分层密钥环之间共享缓存条目。如果您未指定分区 ID，则分层密钥环会自动为密钥环分配一个唯一的分区 ID。

> **Note**
>
> 如果您创建两个或更多引用相同分区 ID 和分支密钥 ID 的密钥环，则您的分层密钥环将在共享缓存中共享相同的缓存条目。logical key store name如果您不希望多个密钥环共享相同的缓存条目，则必须为每个分层密钥环使用唯一的分区 ID。

以下示例创建了一个分层密钥环branch key ID supplier，其缓存限制为 600 秒。有关以下分层密钥环配置中定义的值的更多信息，请参阅the section called “创建分层密钥环”。

Java

```
// Create the Hierarchical keyring
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(sharedCache)
        .partitionID(partitionID)
        .build();
```

```
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
// Create the Hierarchical keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   Cache = sharedCache,
   TtlSeconds = 600,
   PartitionId = partitionID
};
var keyring =
 materialProviders.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Python

```
# Create the Hierarchical keyring
keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=shared_cache,
    partition_id=partition_id
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
```

```
    // pass it to different Hierarchical Keyrings, it will still point to the
 same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

Go

```
// Create the Hierarchical keyring
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:     keyStore1,
    BranchKeyId: &branchKeyId,
    TtlSeconds:  600,
    Cache:       &shared_cache,
    PartitionId: &partitionId,
}
keyring, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

创建分层密钥环

要创建分层密钥环，必须提供以下值：

- 密钥库名称

  您或您的密钥库管理员创建的用作密钥存储的 DynamoDB 表的名称。

-

  缓存限制生存时间（TTL）

  本地缓存中的分支密钥材料条目在过期之前可使用的时长（以秒为单位）。缓存限制 TTL 决定了客户端调 AWS KMS 用授权使用分支密钥的频率。该值必须大于零。缓存限制 TTL 到期后，该条目将永远不会被提供，并将从本地缓存中逐出。

- 分支密钥标识符

您可以静态配置用于标识密钥库中单个活动分支密钥的，也可以提供分支密钥 ID 供应商。`branch-key-id`

分支密钥 ID 提供者使用存储在加密上下文中的字段来确定解密记录需要哪个分支密钥。

对于每个租户都有自己的分支密钥的多租户数据库，我们强烈建议使用分支密钥 ID 供应商。您可以使用分支密钥 ID 供应商为分支密钥 ID 创建友好名称，以便轻松识别特定租户的正确分支密钥 ID。例如，易记名称使您可以将分支密钥引用为 `tenant1` 而非 `b3f61619-4d35-48ad-a275-050f87e15122`。

对于解密操作，您可以静态配置单个分层密钥环以限制对单个租户进行解密，也可以使用分支密钥 ID 提供程序确定哪个租户负责解密记录。

- （可选）缓存

  如果要自定义缓存类型或可存储在本地缓存中分支密钥材料条目的数量，请在初始化密钥环时指定缓存类型和条目容量。

  分层密钥环支持以下缓存类型：默认、 MultiThreaded StormTracking、和共享。有关演示如何定义每种缓存类型的更多信息和示例，请参阅the section called “选择缓存”。

  如果未指定缓存，则分层密钥环会自动使用默认缓存类型并将条目容量设置为 1000。

- （可选）分区 ID

  如果指定the section called “共享缓存”，则可以选择定义分区 ID。分区 ID 可区分哪个分层密钥环正在写入缓存。如果您打算重复使用或共享分区中的缓存条目，则必须定义自己的分区 ID。您可以为分区 ID 指定任何字符串。如果您未指定分区 ID，则会在创建密钥环时自动为密钥环分配一个唯一的分区 ID。

  有关更多信息，请参阅 Partitions。

  > **Note**
  >
  > 如果您创建两个或更多引用相同分区 ID 和分支密钥 ID 的密钥环，则您的分层密钥环将在共享缓存中共享相同的缓存条目。logical key store name如果您不希望多个密钥环共享相同的缓存条目，则必须为每个分层密钥环使用唯一的分区 ID。

- （可选）授权令牌列表

如果您通过授权控制对分层密钥环中 KMS 密钥的访问权限，则必须在初始化密钥环时提供所有必要的授权令牌。

使用静态分支密钥 ID 创建分层密钥环

以下示例演示如何创建具有静态分支密钥 ID、缓存限制 TTL 为 600 秒的分层密钥环。the section called "默认缓存"

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyId = branch-key-id,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
```

```
    key_store=keystore,
    branch_key_id=branch_key_id,
    ttl_seconds=600
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
        .create_aws_kms_hierarchical_keyring()
        .key_store(key_store.clone())
        .branch_key_id(branch_key_id)
        .ttl_seconds(600)
        .send()
        .await?;
```

Go

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:    keyStore,
    BranchKeyId: &branchKeyID,
    TtlSeconds:  600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

使用分支密钥 ID 供应商创建分层密钥环

以下过程演示如何使用分支密钥 ID 提供者创建分层密钥环。

1. 创建分支密钥 ID 供应商

   以下示例定义了一个分支密钥 ID 提供者，该供应商在加密或解密时使用加密上下文为每个租户选择分支密钥 ID。有关每种语言的有效实现，请参阅：

   - Java：ExampleBranchKeyIdSupplier.java
   - C# /.NET: ExampleBranchKeySupplier.cs
   - Python：branch_key_id_supplier_example.py
   - Rust：e xam ple_branch_key_id_supplier.rs
   - Go：b ran chkeysupplier.go

   Java

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
public class ExampleBranchKeyIdSupplier implements IBranchKeyIdSupplier {
    private static String branchKeyIdForTenantA;
    private static String branchKeyIdForTenantB;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenantA = tenant1Id;
        this.branchKeyIdForTenantB = tenant2Id;
    }

    @Override
    public GetBranchKeyIdOutput GetBranchKeyId(GetBranchKeyIdInput input) {
        Map<String, String> encryptionContext = input.encryptionContext();
        if (!encryptionContext.containsKey("tenant")) {
            throw new IllegalArgumentException(
                "EncryptionContext invalid, does not contain expected tenant key
 value pair.");
        }

        String tenantKeyId = encryptionContext.get("tenant");
        String branchKeyId;
        if (tenantKeyId.equals("TenantA")) {
            branchKeyId = branchKeyIdForTenantA;
```

```
        } else if (tenantKeyId.equals("TenantB")) {
            branchKeyId = branchKeyIdForTenantB;
        } else {
            throw new IllegalArgumentException("Item does not contain valid
 tenant ID");
        }

        return GetBranchKeyIdOutput.builder().branchKeyId(branchKeyId).build();
    }
}

// Create the branch key ID supplier
final IBranchKeyIdSupplier branchKeyIdSupplier = new ExampleBranchKeyIdSupplier(
    branch-key-ID-tenantA, branch-key-ID-tenantB);
```

C# / .NET

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
public class ExampleBranchKeySupplier : BranchKeyIdSupplierBase {
    private string branchKeyTenantA;
    private string branchKeyTenantB;

    public ExampleBranchKeySupplier(string branchKeyTenantA, string
 branchKeyTenantB) {
        this.branchKeyTenantA = branchKeyTenantA;
        this.branchKeyTenantB = branchKeyTenantB;
    }

    // The encryption context is used to determine the Branch Key ID.
    protected override GetBranchKeyIdOutput _GetBranchKeyId(GetBranchKeyIdInput
 input) {
        Dictionary<string, string> encryptionContext = input.EncryptionContext;
        if (!encryptionContext.ContainsKey("tenant")) {
            throw new Exception("EncryptionContext invalid, does not contain
 expected tenant key value pair.");
        }

        string tenant = encryptionContext["tenant"];
        if (tenant.Equals("TenantA")) {
            return new GetBranchKeyIdOutput { BranchKeyId = branchKeyTenantA };
        }
        if (tenant.Equals("TenantB")) {
```

```
            return new GetBranchKeyIdOutput { BranchKeyId = branchKeyTenantB };
        }
        throw new Exception("Item does not have a valid tenantID.");
    }
}

// Create the branch key ID supplier
var branchKeyIdSupplier = new ExampleBranchKeySupplier(
    branch-key-ID-tenantA, branch-key-ID-tenantB);
```

Python

```
# Define a branch key ID supplier that uses the encryption context to
# select a branch key ID for each tenant.
class ExampleBranchKeyIdSupplier(IBranchKeyIdSupplier):
    branch_key_id_for_tenant_A: str
    branch_key_id_for_tenant_B: str

    def __init__(self, tenant_1_id, tenant_2_id):
        self.branch_key_id_for_tenant_A = tenant_1_id
        self.branch_key_id_for_tenant_B = tenant_2_id

    def get_branch_key_id(
        self, param: GetBranchKeyIdInput
    ) -> GetBranchKeyIdOutput:
        encryption_context = param.encryption_context
        if "tenant" not in encryption_context:
            raise ValueError("EncryptionContext invalid, does not contain
 expected tenant key value pair.")

        tenant_key_id = encryption_context.get("tenant")
        if tenant_key_id == "TenantA":
            branch_key_id = self.branch_key_id_for_tenant_A
        elif tenant_key_id == "TenantB":
            branch_key_id = self.branch_key_id_for_tenant_B
        else:
            raise ValueError(f"Item does not contain valid tenant ID:
 {tenant_key_id=}")

        return GetBranchKeyIdOutput(branch_key_id=branch_key_id)

# Create the branch key ID supplier
branch_key_id_supplier: IBranchKeyIdSupplier = ExampleBranchKeyIdSupplier(
```

```
        tenant_1_id=branch_key_id_a,
        tenant_2_id=branch_key_id_b,
)
```

Rust

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant_a: String,
    branch_key_id_for_tenant_b: String,
}

impl ExampleBranchKeyIdSupplier {
    pub fn new(tenant_a_id: &str, tenant_b_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant_a: tenant_a_id.to_string(),
            branch_key_id_for_tenant_b: tenant_b_id.to_string(),
        }
    }
}

// The encryption context is used to determine the Branch Key ID.
impl BranchKeyIdSupplier for ExampleBranchKeyIdSupplier {
    fn get_branch_key_id(&self, input: GetBranchKeyIdInput) ->
 Result<GetBranchKeyIdOutput, Error> {
        let encryption_context: HashMap<String, String> =
 input.encryption_context.unwrap();
        if !encryption_context.contains_key("tenant") {
            return Err(Error::AwsCryptographicMaterialProvidersException {
                message: "EncryptionContext invalid, does not contain expected
 tenant key value pair.".to_string(),
            });
        }

        let tenant_key_id: &str = encryption_context["tenant"].as_str();
        if tenant_key_id == "TenantA" {
            Ok(GetBranchKeyIdOutput::builder()
                .branch_key_id(self.branch_key_id_for_tenant_a.clone())
                .build()
                .unwrap())
        } else if tenant_key_id == "TenantB" {
            Ok(GetBranchKeyIdOutput::builder()
```

```
                .branch_key_id(self.branch_key_id_for_tenant_b.clone())
                .build()
                .unwrap())
        } else {
            Err(Error::AwsCryptographicMaterialProvidersException {
                message: "Item does not contain valid tenant ID.".to_string(),
            })
        }
    }
}

// Create the branch key ID supplier
let branch_key_id_supplier = ExampleBranchKeyIdSupplier::new(
    &branch_key_id_a,
    &branch_key_id_b,
);
```

Go

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
type branchKeySupplier struct {
    branchKeyA string
    branchKeyB string
}

// The encryption context is used to determine the Branch Key ID.
func (b *branchKeySupplier) GetBranchKeyId(input mpltypes.GetBranchKeyIdInput)
 (*mpltypes.GetBranchKeyIdOutput, error) {
    ec := input.EncryptionContext
    if value, exists := ec["tenant"]; !exists || value == "" {
        return nil, fmt.Errorf("EncryptionContext invalid, does not contain
 expected tenant key value pair.")
    }

    branchKeyIdentifier := ec["tenant"]
    if branchKeyIdentifier == "TenantA" {
        return &mpltypes.GetBranchKeyIdOutput{BranchKeyId: b.branchKeyA}, nil
    } else if branchKeyIdentifier == "TenantB" {
        return &mpltypes.GetBranchKeyIdOutput{BranchKeyId: b.branchKeyB}, nil
    } else {
        return &mpltypes.GetBranchKeyIdOutput{}, fmt.Errorf("unknown branch key
 identifier")
```

```
    }
}

// Create the branch key ID supplier
keySupplier := branchKeySupplier{branchKeyA: branchKeyA, branchKeyB: branchKeyB}
```

2. 创建分层密钥环

以下示例使用步骤 1 中创建的分支密钥 ID 供应商初始化分层密钥环，缓存限制 TLL 为 600 秒，最大缓存大小为 1000。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 100 }
   }
};
```

```
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=CacheTypeDefault(
        value=DefaultCache(
            entry_capacity=100
        )
    ),
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store.clone())
    .branch_key_id_supplier(branch_key_id_supplier)
    .ttl_seconds(600)
    .send()
    .await?;
```

Go

```
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:            keyStore,
    BranchKeyIdSupplier: &keySupplier,
```

```
    TtlSeconds:          600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

AWS KMS ECDH 钥匙圈

AWS KMS ECDH 密钥环使用非对称密钥协议AWS KMS keys来派生双方共享的对称包装密钥。首先，密钥环使用 Elliptic Curve Diffie-Hellman (ECDH) 密钥协议算法，从发送者的 KMS 密钥对中的私钥和接收者的公钥中派生出共享密钥。然后，密钥环使用共享密钥来派生用于保护您的数据加密密钥的共享包装密钥。 AWS Encryption SDK 使用 (`KDF_CTR_HMAC_SHA384`) 派生共享包装密钥的密钥派生函数符合 NIST 关于密钥派生的建议。

密钥派生函数返回 64 字节的密钥材料。为确保双方使用正确的密钥材料，使用前 32 个字节作为承诺密钥， AWS Encryption SDK 使用最后 32 字节作为共享封装密钥。解密时，如果密钥环无法复制存储在邮件标题密文中的相同承诺密钥和共享包装密钥，则操作将失败。例如，如果您使用配置有 Alice 私钥和 Bob 公钥的密钥环对数据进行加密，则使用 Bob 的私钥和 Alice 的公钥配置的密钥环将复制相同的承诺密钥和共享包装密钥，并能够解密数据。如果 Bob 的公钥不是来自 KMS 密钥对，那么 Bob 可以创建一个 Ra w ECDH 密钥环来解密数据。

AWS KMS ECDH 密钥环使用 AES-GCM 使用对称密钥对数据进行加密。然后使用 AES-GCM 使用派生的共享包装密钥对数据密钥进行信封加密。每个 AWS KMS ECDH 密钥环只能有一个共享的包装密钥，但您可以在多密钥环中单独或与其他密钥环一起包含多个 AWS KMS ECDH 密钥环。

编程语言兼容性

AWS KMS ECDH 密钥环在加密材料提供程序库 (MPL) 的 1.5.0 版本中引入，并由以下编程语言和版本支持：

- 版本 3。 的 x AWS Encryption SDK for Java
- 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
- 版本 4。 的 x AWS Encryption SDK for Python，与可选的 MPL 依赖项一起使用时。
- 版本 1。 x 的 fo r AWS Encryption SDK Rust
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

主题

- AWS KMS ECDH 密钥环所需的权限
- 创建 AWS KMS ECDH 密钥环
- 创建 AWS KMS ECDH 发现密钥环

AWS KMS ECDH 密钥环所需的权限

AWS Encryption SDK 不需要 AWS 帐户，也不依赖任何 AWS 服务。但是，要使用 AWS KMS ECDH 密钥环，您需要一个 AWS 帐户以及对密钥环 AWS KMS keys 中的以下最低权限。权限因您使用的密钥协议架构而异。

- 要使用密`KmsPrivateKeyToStaticPublicKey`钥协议架构加密和解密数据，您需要在发送方的非对称 KMS 密钥对DeriveSharedSecret上使用 kms: GetPublicKey 和 km s:。如果您在实例化密钥环时直接提供发送者的 DER 编码公钥，则只需要对发送者的非对称 KMS 密钥对DeriveSharedSecret具有 kms: 权限。
- 要使用密`KmsPublicKeyDiscovery`钥协议架构解密数据，您需要对指定的非对称 KMS 密钥对具有 kms: DeriveSharedSecret 和 kms: GetPublicKey 权限。

创建 AWS KMS ECDH 密钥环

要创建用于加密和解密数据的 AWS KMS ECDH 密钥环，必须使用密钥协议架构。`KmsPrivateKeyToStaticPublicKey`要使用密钥协议架构初始化 AWS KMS ECDH `KmsPrivateKeyToStaticPublicKey` 密钥环，请提供以下值：

- 发件人 AWS KMS key 身份证

  必须标识值为的非对称 NIST 推荐的椭圆曲线 (ECC) KMS 密钥对。`KeyUsage` `KEY_AGREEMENT`发送者的私钥用于派生共享密钥。
- （可选）发件人的公钥

  必须是 DER 编码的 X.509 公钥，也称为 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定义。

  该 AWS KMS GetPublicKey操作以所需的 DER 编码格式返回非对称 KMS 密钥对的公钥。

  要减少密钥环 AWS KMS 拨打的次数，您可以直接提供发件人的公钥。如果没有为发件人的公钥提供任何值，则密钥环会调用 AWS KMS 以检索发送者的公钥。

- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 SubjectPublicKeyInfo (SPKI)，如 RFC 5280 中所定义。

  该 AWS KMS GetPublicKey操作以所需的 DER 编码格式返回非对称 KMS 密钥对的公钥。

- 曲线规格

  标识指定密钥对中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

- （可选）授权令牌列表

  如果您通过授权控制对 AWS KMS ECDH 密钥环中 KMS 密钥的访问权限，则在初始化密钥环时必须提供所有必要的授权令牌。

C# / .NET

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选SenderPublicKey参数提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。发件人和收件人的密钥对都在ECC_NIST_P256弯曲中。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Must be DER-encoded X.509 public keys
var BobPublicKey = new MemoryStream(new byte[] { });
var AlicePublicKey = new MemoryStream(new byte[] { });

// Create the AWS KMS ECDH static keyring
var staticConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPrivateKeyToStaticPublicKey = new KmsPrivateKeyToStaticPublicKeyInput
    {
        SenderKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        SenderPublicKey = BobPublicKey,
        RecipientPublicKey = AlicePublicKey
    }
};
```

```
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = staticConfiguration
};

var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选senderPublicKey参数提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。发件人和收件人的密钥对都在ECC_NIST_P256弯曲中。

```
// Retrieve public keys
// Must be DER-encoded X.509 public keys
ByteBuffer BobPublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab");
        ByteBuffer AlicePublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the AWS KMS ECDH static keyring
        final CreateAwsKmsEcdhKeyringInput senderKeyringInput =
          CreateAwsKmsEcdhKeyringInput.builder()
            .kmsClient(KmsClient.create())
            .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
            .KeyAgreementScheme(
              KmsEcdhStaticConfigurations.builder()
                .KmsPrivateKeyToStaticPublicKey(
                  KmsPrivateKeyToStaticPublicKeyInput.builder()
                    .senderKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
                    .senderPublicKey(BobPublicKey)
                    .recipientPublicKey(AlicePublicKey)
                    .build()).build()).build();
```

Python

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选senderPublicKey参数提供发送者的公钥。如果您不提供发件人的公钥，则

密钥环会调用 AWS KMS 以检索发件人的公钥。发件人和收件人的密钥对都在ECC_NIST_P256弯曲中。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey,
    KmsPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Retrieve public keys
# Must be DER-encoded X.509 public keys
bob_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
alice_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321")

# Create the AWS KMS ECDH static keyring
sender_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey(
        KmsPrivateKeyToStaticPublicKeyInput(
            sender_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            sender_public_key = bob_public_key,
            recipient_public_key = alice_public_key,

        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(sender_keyring_input)
```

Rust

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选sender_public_key参数提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
```

```
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
```

```
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Retrieve public keys
// Must be DER-encoded X.509 keys
publicKeySender, err := utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameSender)
if err != nil {
    panic(err)
}
publicKeyRecipient, err :=
 utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create KmsPrivateKeyToStaticPublicKeyInput
kmsEcdhStaticConfigurationInput := mpltypes.KmsPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey:  publicKeyRecipient,
    SenderKmsIdentifier: arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    SenderPublicKey:     publicKeySender,
}
kmsEcdhStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPrivateKeyToStaticPublicKey{
    Value: kmsEcdhStaticConfigurationInput,
```

```
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create AWS KMS ECDH keyring
awsKmsEcdhKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhKeyring, err := matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

创建 AWS KMS ECDH 发现密钥环

解密时，最佳做法是指定 AWS Encryption SDK 可以使用的密钥。要遵循此最佳实践，请使用带有密钥协议架构的 AWS KMS ECDH `KmsPrivateKeyToStaticPublicKey` 密钥环。但是，您也可以创建 AWS KMS ECDH 发现密钥环，即 AWS KMS ECDH 密钥环，该密钥环可以解密任何消息，其中指定 KMS 密钥对的公钥与存储在消息密文中的收件人的公钥相匹配。

> **Important**
>
> 使用密`KmsPublicKeyDiscovery`钥协议架构解密消息时，无论谁拥有所有公钥，都将接受所有公钥。

要使用密钥协议架构初始化 AWS KMS ECDH `KmsPublicKeyDiscovery` 密钥环，请提供以下值：

- 收件人的 AWS KMS key 身份证

  必须标识值为的非对称 NIST 推荐的椭圆曲线 (ECC) KMS 密钥对。`KeyUsage` `KEY_AGREEMENT`

- 曲线规格

标识收件人的 KMS 密钥对中的椭圆曲线规范。

有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

- （可选）授权令牌列表

  如果您通过授权控制对 AWS KMS ECDH 密钥环中 KMS 密钥的访问权限，则在初始化密钥环时必须提供所有必要的授权令牌。

C# / .NET

以下示例创建了一个 AWS KMS ECDH 发现密钥环，曲线上有 KMS 密钥对。ECC_NIST_P256您必须对指定的 KMS 密钥对拥有 kms: GetPublicKey 和 kms: DeriveSharedSecret 权限。此密钥环可以解密任何消息，其中指定 KMS 密钥对的公钥与存储在消息密文中的收件人的公钥相匹配。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create the AWS KMS ECDH discovery keyring
var discoveryConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPublicKeyDiscovery = new KmsPublicKeyDiscoveryInput
    {
        RecipientKmsIdentifier = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
    }

};
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = discoveryConfiguration
};
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

以下示例创建了一个 AWS KMS ECDH 发现密钥环，曲线上有 KMS 密钥对。ECC_NIST_P256您必须对指定的 KMS 密钥对拥有 kms: GetPublicKey 和 kms: DeriveSharedSecret 权限。此密钥环可以解密任何消息，其中指定 KMS 密钥对的公钥与存储在消息密文中的收件人的公钥相匹配。

```
// Create the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

Python

以下示例创建了一个 AWS KMS ECDH 发现密钥环，曲线上有 KMS 密钥对。ECC_NIST_P256您必须对指定的 KMS 密钥对拥有 kms: GetPublicKey 和 kms: DeriveSharedSecret 权限。此密钥环可以解密任何消息，其中指定 KMS 密钥对的公钥与存储在消息密文中的收件人的公钥相匹配。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery,
    KmsPublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create the AWS KMS ECDH discovery keyring
create_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery(
        KmsPublicKeyDiscoveryInput(
            recipient_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321",
        )
```

```
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(create_keyring_input)
```

Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
```

```
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
```

```
    "the data you are handling": "is what you think it is",
}

// Create KmsPublicKeyDiscoveryInput
kmsEcdhDiscoveryStaticConfigurationInput := mpltypes.KmsPublicKeyDiscoveryInput{
    RecipientKmsIdentifier: eccRecipientKeyArn,
}
kmsEcdhDiscoveryStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPublicKeyDiscovery{
    Value: kmsEcdhDiscoveryStaticConfigurationInput,
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create AWS KMS ECDH discovery keyring
awsKmsEcdhDiscoveryKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhDiscoveryStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhDiscoveryKeyring, err :=
 matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

原始 AES 密钥环

AWS Encryption SDK 允许您使用您提供的 AES 对称密钥作为包装密钥来保护您的数据密钥。您需要生成、存储和保护密钥材料，最好是在硬件安全模块（HSM）或密钥管理系统中操作。如果您需要提供包装密钥并在本地或离线加密数据密钥，则请使用原始 AES 密钥环。

原始 AES 密钥环使用 AES-GCM 算法以及您指定为字节数组的包装密钥对数据进行加密。每个原始 AES 密钥环中只能指定一个包装密钥，但每个多重密钥环中可以包含多个原始 AES 密钥环，该等密钥环可单独纳入或与其他密钥环一同纳入。

当 AES 加密密钥与 AES 加密密钥一起使用 AWS Encryption SDK for Python 时，Raw AES 密钥环等同于 AWS Encryption SDK for Java 和中的RawMasterKey类并与该类互操作。JceMasterKey您可以使用一种实现加密数据，用使用相同包装密钥的任何其他实现进行解密。有关更多信息，请参阅 密钥环兼容性。

密钥命名空间和名称

为标识密钥环中的 AES 密钥，原始 AES 密钥环使用您提供的密钥命名空间和密钥名称。这些值不是机密的。其以明文形式出现在加密操作返回的加密消息的标头中。我们建议在 HSM 或密钥管理系统中使用密钥命名空间与用于标识该系统中 AES 密钥的密钥名称。

> **Note**
>
> 密钥命名空间和密钥名称等同于 `JceMasterKey` 和 `RawMasterKey` 中提供程序 ID（或提供程序）和密钥 ID字段。
> .NET AWS Encryption SDK 的 AWS Encryption SDK for C 和保留 KMS `aws-kms` 密钥的密钥命名空间值。请勿在包含这些库的原始 AES 密钥环或原始 RSA 密钥环中使用此命名空间值。

如果您通过构造不同的密钥环加密和解密给定消息，命名空间和名称值则至关重要。如果解密密钥环中的密钥命名空间和密钥名称与加密密钥环中的密钥命名空间和密钥名称不完全匹配、大小写不一致，即使密钥材料字节数相同，也不会使用解密密钥环。

例如，您可以使用密钥命名空间 `HSM_01` 和密钥名称 `AES_256_012` 定义原始 AES 密钥环。然后使用该密钥环加密部分数据。要解密这些数据，请使用相同的密钥命名空间、密钥名称和密钥材料构造原始 AES 密钥环。

以下示例说明了如何创建原始 AES 密钥。`AESWrappingKey` 变量代表您提供的密钥材料。

C

要在中实例化 Raw AES 密钥环， AWS Encryption SDK for C请使用。`aws_cryptosdk_raw_aes_keyring_new()`有关完整示例，请参阅 raw_aes_keyring.c。

```
struct aws_allocator *alloc = aws_default_allocator();

AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_name, "AES_256_012");

struct aws_cryptosdk_keyring *raw_aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
```

```
        alloc, wrapping_key_namespace, wrapping_key_name, aes_wrapping_key,
 wrapping_key_len);
```

C# / .NET

要在 AWS Encryption SDK 中为.NET 创建原始 AES 密钥环，请使用`materialProviders.CreateRawAesKeyring()`方法。有关完整的示例，请参阅 Raw AESKeyring Example.cs。

以下示例使用版本 4。 x 及更高版本 AWS Encryption SDK 适用于.NET。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

// This example uses the key generator in Bouncy Castle to generate the key
 material.
// In production, use key material from a secure source.
var aesWrappingKey = new
 MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());

// Create the keyring that determines how your data keys are protected.
var createKeyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = aesWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var keyring = materialProviders.CreateRawAesKeyring(createKeyringInput);
```

JavaScript Browser

浏览器 AWS Encryption SDK for JavaScript 中的从 API 中获取其加密原语。WebCrypto在构造密钥环之前，必须使用`RawAesKeyringWebCrypto.importCryptoKey()`将原始密钥材料导入 WebCrypto 后端。这样可以确保即使对的所有调用都是异步的，密钥环也是完整 WebCrypto 的。

然后，要实例化原始 AES 密钥环，请使用 RawAesKeyringWebCrypto() 方法。您必须根据密钥材料的长度指定 AES 包装算法（“包装套件”）。有关完整的示例，请参阅 aes_simple.ts（浏览器）。JavaScript

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
  RawAesWrappingSuiteIdentifier,
  RawAesKeyringWebCrypto,
  synchronousRandomValues,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const keyNamespace = 'HSM_01'
const keyName = 'AES_256_012'

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

/* Import the plaintext AES key into the WebCrypto backend. */
const aesWrappingKey = await RawAesKeyringWebCrypto.importCryptoKey(
  rawAesKey,
  wrappingSuite
)

const rawAesKeyring = new RawAesKeyringWebCrypto({
  keyName,
  keyNamespace,
  wrappingSuite,
  aesWrappingKey
})
```

JavaScript Node.js

要在 for Node.js 中实例化 Raw AES 密钥环，请创建该 AWS Encryption SDK for JavaScript 类的实例。 RawAesKeyringNode您必须根据密钥材料的长度指定 AES 包装算法（“包装套件”）。有关完整的示例，请参阅 aes_simple.ts (Node.js)。JavaScript

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
  RawAesKeyringNode,
  buildClient,
  CommitmentPolicy,
  RawAesWrappingSuiteIdentifier,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const keyName = 'AES_256_012'
const keyNamespace = 'HSM_01'

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

const rawAesKeyring = new RawAesKeyringNode({
  keyName,
  keyNamespace,
  aesWrappingKey,
  wrappingSuite,
})
```

Java

要在中实例化 Raw AES 密钥环， AWS Encryption SDK for Java请使用。matProv.CreateRawAesKeyring()

```
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
```

```
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

Python

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。REQUIRE_ENCRYPT_REQUIRE_DECRYPT有关完整示例，请参阅中 AWS Encryption SDK for Python 存储库中的 raw_aes_keyring_example.py GitHub。

```
# Instantiate the AWS Encryption SDK client
client = aws_encryption_sdk.EncryptionSDKClient(
    commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "AES_256_012"

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create Raw AES keyring
keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)
```

```
raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=keyring_input
)
```

Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "AES_256_012";

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw AES keyring
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Go

```
import (
```

```
    mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
    client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
    esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
//Instantiate the AWS Encryption SDK client.
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
// Define the key namespace and key name
var keyNamespace = "A managed aes keys"
var keyName = "My 256-bit AES wrapping key"

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}
// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  aesWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
    panic(err)
}
```

原始 RSA 密钥环

原始 RSA 密钥环使用您提供的 RSA 公有密钥和私有密钥为本地内存中的数据密钥执行非对称加密和解密。您需要生成、存储和保护私有密钥，最好是在硬件安全模块（HSM）或密钥管理系统中操作。加密功能对 RSA 公有密钥下的数据密钥进行加密。解密功能使用私有密钥对数据密钥进行解密。您可以从几种 RSA 填充模式中进行选择。

加密和解密的原始 RSA 密钥环必须包含一个非对称公有密钥和私有密钥对。但是，您可以使用仅具有公有密钥的原始 RSA 密钥环加密数据，并使用仅具有私有密钥的原始 RSA 密钥环解密数据。您可以在多重密钥环中包含任何原始 RSA 密钥环。如果您为原始 RSA 密钥环配置公有密钥和私有密钥，请确保其属于同一个密钥对。的某些语言实现 AWS Encryption SDK 不会使用来自不同对的密钥构建 Raw RSA 密钥环。其他语言实施则需要您验证密钥是否来自同一个密钥对。

Raw RSA 密钥环与 RSA 非对称加密密钥一起使用 AWS Encryption SDK for Python 时，等同于 AWS Encryption SDK for Java 和RawMasterKey中的密钥并与之互操作。JceMasterKey您可以使用一种实现加密数据，用使用相同包装密钥的任何其他实现进行解密。有关更多信息，请参阅 密钥环兼容性。

> **Note**
>
> 原始 RSA 密钥环不支持非对称 KMS 密钥。如果要使用非对称 RSA KMS 密钥，则以下编程语言支持使用非对称 RSA 的 AWS KMS 密钥环： AWS KMS keys
>
> - 版本 3。 的 x AWS Encryption SDK for Java
> - 版本 4。 .NET 的 x 及更高版本AWS Encryption SDK
> - 版本 4。 的 x AWS Encryption SDK for Python，与可选的加密材料提供程序库 (MPL) 依赖项一起使用时。
> - 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go
>
> 如果您使用包含 RSA KMS 密钥公钥的 Raw RSA 密钥环对数据进行加密，则两者都 AWS Encryption SDK 无法 AWS KMS 对其进行解密。您无法将 AWS KMS 非对称 KMS 密钥的私钥导出到原始 RSA 密钥环中。 AWS KMS 解密操作无法解密返回的加密消息。 AWS Encryption SDK

在中构建 Raw RSA 密钥环时 AWS Encryption SDK for C，请务必提供包含每个密钥的 PEM 文件的内容，以空结尾的 C 字符串，而不是路径或文件名。在 JavaScript 中构造原始 RSA 密钥环时，请注意与其他语言实施的潜在不兼容性问题。

命名空间和名称

为标识密钥环中的 RSA 密钥材料，原始 AES 密钥环使用您提供的命名空间和密钥名称。这些值不是机密的。其以明文形式出现在加密操作返回的加密消息的标头中。我们建议在 HSM 或密钥管理系统中使用密钥命名空间与用于标识 RSA 密钥对（或其私有密钥）的密钥名称。

> **Note**
>
> 密钥命名空间和密钥名称等同于 `JceMasterKey` 和 `RawMasterKey` 中提供程序 ID（或提供程序）和密钥 ID字段。
>
> 为 KMS `aws-kms` 密钥 AWS Encryption SDK for C 保留密钥命名空间值。请勿在原始 AES 密钥环或原始 RSA 密钥环中将其与 AWS Encryption SDK for C共用。

如果您通过构造不同的密钥环加密和解密给定消息，命名空间和名称值则至关重要。如果解密密钥环中的密钥命名空间和密钥名称与加密密钥环中的密钥命名空间和密钥名称不完全匹配、大小写不一致，即使密钥来自相同的密钥对，也不会使用解密密钥环。

无论密钥环中包含 RSA 公有密钥、RSA 私有密钥还是密钥对中的两个密钥，加密和解密密钥环中密钥材料的密钥命名空间和密钥名称必须相同。例如，假设您使用包含密钥命名空间 `HSM_01` 和密钥名称 `RSA_2048_06` 的 RSA 公有密钥的原始 RSA 密钥环加密数据。要解密数据，请使用私有密钥（或密钥对）、相同的密钥命名空间和名称构造原始 RSA 密钥环。

填充模式

您必须为用于加密和解密的原始 RSA 密钥环指定填充模式，或者使用为您指定填充模式的语言实施功能。

AWS Encryption SDK 支持以下填充模式，受每种语言的限制。我们建议使用 OAEP 填充模式，尤其是带有 SHA-256 和 MGF1 SHA-256 填充的 OAEP。仅支持PKCS1填充模式是为了向后兼容。

- 带有 SHA-1 和 MGF1 SHA-1 填充的 OAEP
- 带有 SHA-256 和 MGF1 SHA-256 填充的 OAEP
- 带有 SHA-384 和 MGF1 SHA-384 填充的 OAEP
- 带有 SHA-512 和 MGF1 SHA-512 填充的 OAEP

- PKCS1 v1.5 填充

以下示例展示了如何使用 RSA 密钥对的公钥和私钥以及使用 SHA-256 和 MGF1 SHA-256 填充模式的 OAEP 创建原始 RSA 密钥环。RSAPublicKey 和 RSAPrivateKey 变量代表您提供的密钥材料。

C

要在中创建 RSA 原始密钥环 AWS Encryption SDK for C，请使用。aws_cryptosdk_raw_rsa_keyring_new

在中构建 Raw RSA 密钥环时 AWS Encryption SDK for C，请务必提供包含每个密钥的 PEM 文件的内容，以空结尾的 C 字符串，而不是路径或文件名。有关完整示例，请参阅 raw_rsa_keyring.c。

```
struct aws_allocator *alloc = aws_default_allocator();

AWS_STATIC_STRING_FROM_LITERAL(key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(key_name, "RSA_2048_06");

struct aws_cryptosdk_keyring *rawRsaKeyring = aws_cryptosdk_raw_rsa_keyring_new(
    alloc,
    key_namespace,
    key_name,
    private_key_from_pem,
    public_key_from_pem,
    AWS_CRYPTOSDK_RSA_OAEP_SHA256_MGF1);
```

C# / .NET

要在 AWS Encryption SDK 适用于.NET 的 Raw RSA 密钥环中实例化，请使用方法。materialProviders.CreateRawRsaKeyring()有关完整的示例，请参阅 Raw RSAKeyring Example.cs。

以下示例使用版本 4。 x 及更高版本 AWS Encryption SDK 适用于.NET。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());

var keyNamespace = "HSM_01";
var keyName = "RSA_2048_06";
```

```
// Get public and private keys from PEM files
var publicKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePublicKey.pem"));
var privateKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePrivateKey.pem"));

// Create the keyring input
var createRawRsaKeyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
};

// Create the keyring
var rawRsaKeyring = materialProviders.CreateRawRsaKeyring(createRawRsaKeyringInput);
```

JavaScript Browser

浏览器 AWS Encryption SDK for JavaScript 中的从库中获取其加密原语。WebCrypto在构造密钥环之前，必须使用`importPublicKey()` and/or `importPrivateKey()`将原始密钥材料导入 WebCrypto 后端。这样可以确保即使对的所有调用都是异步的，密钥环也是完整 WebCrypto 的。导入方法采用的对象包括包装算法及其填充模式。

导入密钥材料后，使用 `RawRsaKeyringWebCrypto()` 方法实例化密钥环。在中构建 Raw RSA 密钥环时 JavaScript，请注意可能与其他语言实现不兼容。

以下示例使用`buildClient`函数来指定默认的承诺策略`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。您也可以使用`buildClient`来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

有关完整的示例，请参阅 rsa_simple .ts（浏览器）。JavaScript

```
import {
  RsaImportableKey,
  RawRsaKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'
```

```
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const privateKey = await RawRsaKeyringWebCrypto.importPrivateKey(
  privateRsaJwKKey
)

const publicKey = await RawRsaKeyringWebCrypto.importPublicKey(
  publicRsaJwKKey
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringWebCrypto({
  keyName,
  keyNamespace,
  publicKey,
  privateKey,
})
```

JavaScript Node.js

要在 AWS Encryption SDK for JavaScript Node.js 中实例化原始 RSA 密钥环，请创建该类的新实例。RawRsaKeyringNodewrapKey 参数用于保存公有密钥。unwrapKey 参数用于保存私有密钥。尽管您可以指定首选填充模式，但 RawRsaKeyringNode 构造函数会为您计算默认填充模式。

在中构造原始 RSA 密钥环时 JavaScript，请注意可能与其他语言实现不兼容。

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

有关完整的示例，请参阅 rsa_simple .ts (Node.js)。JavaScript

```
import {
  RawRsaKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'
```

```
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringNode({ keyName, keyNamespace, rsaPublicKey,
 rsaPrivateKey})
```

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
        .keyName("RSA_2048_06")
        .keyNamespace("HSM_01")
        .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
        .publicKey(RSAPublicKey)
        .privateKey(RSAPrivateKey)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Python

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。REQUIRE_ENCRYPT_REQUIRE_DECRYPT有关完整示例，请参阅中 AWS Encryption SDK for Python 存储库中的 raw_rsa_keyring_example.py GitHub。

```
# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "RSA_2048_06"

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create Raw RSA keyring
keyring_input: CreateRawRsaKeyringInput = CreateRawRsaKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
```

```
    padding_scheme=PaddingScheme.OAEP_SHA256_MGF1,
    public_key=RSAPublicKey,
    private_key=RSAPrivateKey
)

raw_rsa_keyring: IKeyring = mat_prov.create_raw_rsa_keyring(
    input=keyring_input
)
```

Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create an encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "RSA_2048_06";

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create Raw RSA keyring
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(aws_smithy_types::Blob::new(RSAPublicKey))
    .private_key(aws_smithy_types::Blob::new(RSAPrivateKey))
    .send()
    .await?;
```

Go

```
// Instantiate the material providers library
matProv, err :=
 awscryptographymaterialproviderssmithygenerated.NewClient(awscryptographymaterialproviderss

// Create Raw RSA keyring
rsaKeyRingInput :=
 awscryptographymaterialproviderssmithygeneratedtypes.CreateRawRsaKeyringInput{
 KeyName:       "rsa",
 KeyNamespace:  "rsa-keyring",
 PaddingScheme:
 awscryptographymaterialproviderssmithygeneratedtypes.PaddingSchemePkcs1,
 PublicKey:     pem.EncodeToMemory(publicKeyBlock),
 PrivateKey:    pem.EncodeToMemory(privateKeyBlock),
}

rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create an encryption context
encryptionContext := map[string]string{
```

```
    "encryption":                 "context",
    "is not":                     "secret",
    "but adds":                   "useful metadata",
    "that can help you":          "be confident that",
    "the data you are handling": "is what you think it is",
}

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create Raw RSA keyring
rsaKeyRingInput := mpltypes.CreateRawRsaKeyringInput{
    KeyName:       keyName,
    KeyNamespace:  keyNamespace,
    PaddingScheme: mpltypes.PaddingSchemeOaepSha512Mgf1,
    PublicKey:     (RSAPublicKey),
    PrivateKey:    (RSAPrivateKey),
}
rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
if err != nil {
    panic(err)
}
```

未加工的 ECDH 钥匙圈

Raw ECDH 密钥环使用您提供的椭圆曲线公私钥对来派生出双方之间的共享包装密钥。首先，密钥环使用发送者的私钥、收件人的公钥和 Elliptic Curve Diffie-Hellman (ECDH) 密钥协议算法派生出共享密钥。然后，密钥环使用共享密钥来派生用于保护您的数据加密密钥的共享包装密钥。 AWS Encryption SDK 使用 (KDF_CTR_HMAC_SHA384) 派生共享包装密钥的密钥派生函数符合 NIST 关于密钥派生的建议。

密钥派生函数返回 64 字节的密钥材料。为确保双方使用正确的密钥材料， AWS Encryption SDK 使用前 32 字节作为承诺密钥，使用最后 32 字节作为共享包装密钥。解密时，如果密钥环无法复制存储在邮件标题密文中的相同承诺密钥和共享包装密钥，则操作将失败。例如，如果您使用配置有 Alice 私钥和 Bob 公钥的密钥环对数据进行加密，则使用 Bob 的私钥和 Alice 的公钥配置的密钥环将复制相同的承诺密钥和共享包装密钥，并能够解密数据。如果 Bob 的公钥来自一 AWS KMS key 对，那么 Bob 可以创建 AWS KMS ECDH 密钥环来解密数据。

Raw ECDH 密钥环使用 AES-GCM 使用对称密钥对数据进行加密。然后使用 AES-GCM 使用派生的共享包装密钥对数据密钥进行信封加密。每个 Raw ECDH 密钥环只能有一个共享包装密钥，但您可以在多密钥环中单独或与其他密钥环一起包含多个 Raw ECDH 密钥环。

您负责生成、存储和保护您的私钥，最好是在硬件安全模块 (HSM) 或密钥管理系统中。发件人和收件人的密钥对基本上呈相同的椭圆曲线。 AWS Encryption SDK 支持以下椭圆曲线规格：

- `ECC_NIST_P256`
- `ECC_NIST_P384`
- `ECC_NIST_P512`

编程语言兼容性

Raw ECDH 密钥环是在加密材料提供程序库 (MPL) 的 1.5.0 版本中引入的，并由以下编程语言和版本支持：

- 版本 3。 的 x AWS Encryption SDK for Java
- 版本 4。 .NET 的 x 及更高版本 AWS Encryption SDK
- 版本 4。 的 x AWS Encryption SDK for Python，与可选的 MPL 依赖项一起使用时。
- 版本 1。 x 的 fo r AWS Encryption SDK Rust
- 版本 0.1。 x 或更高版本的 fo AWS Encryption SDK r Go

创建原始的 ECDH 密钥环

Raw ECDH 密钥环支持三种密钥协议架构：`RawPrivateKeyToStaticPublicKey`、`EphemeralPrivateKeyToStaticPublicKey`和。`Public`
选择的密钥协议架构决定了您可以执行哪些加密操作以及密钥材料的组装方式。

主题

- RawPrivateKeyToStaticPublicKey

- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

使用`RawPrivateKeyToStaticPublicKey`密钥协议架构在密钥环中静态配置发送者的私钥和收件人的公钥。此密钥协议架构可以加密和解密数据。

要使用密钥协议架构初始化 Raw ECDH `RawPrivateKeyToStaticPublicKey` 密钥环，请提供以下值：

- 发件人的私钥

  您必须提供发件人的 PEM 编码私钥（PKCS #8 PrivateKeyInfo 结构），如 RFC 5958 中所定义。
- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定义。

  您可以指定非对称密钥协议 KMS 密钥对的公钥，也可以指定在外部生成的密钥对中的 AWS公钥。
- 曲线规格

  标识指定密钥对中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：`ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

C# / .NET

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var BobPrivateKey = new MemoryStream(new byte[] { });
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH static keyring
     var staticConfiguration = new RawEcdhStaticConfigurations()
     {
      RawPrivateKeyToStaticPublicKey = new RawPrivateKeyToStaticPublicKeyInput
      {
       SenderStaticPrivateKey = BobPrivateKey,
       RecipientPublicKey = AlicePublicKey
```

```
    }
   };

   var createKeyringInput = new CreateRawEcdhKeyringInput()
   {
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KeyAgreementScheme = staticConfiguration
   };

   var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下 Java 示例使用RawPrivateKeyToStaticPublicKey密钥协议架构静态配置发送者的私钥和收件人的公钥。两个密钥对都在ECC_NIST_P256曲线上。

```
private static void StaticRawKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair senderKeys = GetRawEccKey();
    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH static keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .RawPrivateKeyToStaticPublicKey(
                RawPrivateKeyToStaticPublicKeyInput.builder()
                  // Must be a PEM-encoded private key

 .senderStaticPrivateKey(ByteBuffer.wrap(senderKeys.getPrivate().getEncoded()))
                  // Must be a DER-encoded X.509 public key

 .recipientPublicKey(ByteBuffer.wrap(recipient.getPublic().getEncoded()))
                  .build()
            )
            .build()
```

```
        ).build();

    final IKeyring staticKeyring =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Python

以下 Python 示例使用RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey密钥协议架构静态配置发送者的私钥和接收者的公钥。两个密钥对都在ECC_NIST_P256曲线上。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey,
    RawPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Must be a PEM-encoded private key
bob_private_key = get_private_key_bytes()
# Must be a DER-encoded X.509 public key
alice_public_key = get_public_key_bytes()

# Create the raw ECDH static keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey(
        RawPrivateKeyToStaticPublicKeyInput(
            sender_static_private_key = bob_private_key,
            recipient_public_key = alice_public_key,
        )
    )
)
```

```
keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

以下 Python 示例使用raw_ecdh_static_configuration密钥协议架构静态配置发送者的私钥和接收者的公钥。两个密钥对必须位于同一条曲线上。

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
```

```
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Create keyring input
rawEcdhStaticConfigurationInput := mpltypes.RawPrivateKeyToStaticPublicKeyInput{
    SenderStaticPrivateKey: privateKeySender,
    RecipientPublicKey:     publicKeyRecipient,
}
rawECDHStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberRawPrivateKeyToStaticPublicKey{
    Value: rawEcdhStaticConfigurationInput,
}
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
```

```
    KeyAgreementScheme: rawECDHStaticConfiguration,
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create raw ECDH static keyring
rawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

EphemeralPrivateKeyToStaticPublicKey

使用密钥协议架构配置的`EphemeralPrivateKeyToStaticPublicKey`密钥环在本地创建新的密钥对，并为每个加密调用派生一个唯一的共享包装密钥。

此密钥协议架构只能加密消息。要解密使用密`EphemeralPrivateKeyToStaticPublicKey`钥协议架构加密的消息，必须使用配置有相同收件人公钥的发现密钥协议架构。要解密，您可以使用带有密钥协议算法的原始 ECDH 密钥环，或者，如果接收者的公`PublicKeyDiscovery`钥来自非对称密钥协议 KMS 密钥对，则可以将 AWS KMS ECDH 密钥环与密钥协议架构一起使用。KmsPublicKeyDiscovery

要使用密钥协议架构初始化 Raw ECDH `EphemeralPrivateKeyToStaticPublicKey` 密钥环，请提供以下值：

- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定义。

  您可以指定非对称密钥协议 KMS 密钥对的公钥，也可以指定在外部生成的密钥对中的 AWS公钥。
- 曲线规格

  标识指定公钥中的椭圆曲线规范。

加密时，密钥环会在指定曲线上创建新的密钥对，并使用新的私钥和指定的公钥来派生共享的包装密钥。

有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

以下示例使用密钥协议架构创建一个 Raw ECDH EphemeralPrivateKeyToStaticPublicKey 密钥环。加密后，密钥环将在指定ECC_NIST_P256曲线上本地创建一个新的密钥对。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH ephemeral keyring
     var ephemeralConfiguration = new RawEcdhStaticConfigurations()
     {
      EphemeralPrivateKeyToStaticPublicKey = new
 EphemeralPrivateKeyToStaticPublicKeyInput
      {
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = ephemeralConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下示例使用密钥协议架构创建一个 Raw ECDH EphemeralPrivateKeyToStaticPublicKey 密钥环。加密后，密钥环将在指定ECC_NIST_P256曲线上本地创建一个新的密钥对。

```
private static void EphemeralRawEcdhKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
```

```
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    ByteBuffer recipientPublicKey = getPublicKeyBytes();

    // Create the Raw ECDH ephemeral keyring
    final CreateRawEcdhKeyringInput ephemeralInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .EphemeralPrivateKeyToStaticPublicKey(
              EphemeralPrivateKeyToStaticPublicKeyInput.builder()
                .recipientPublicKey(recipientPublicKey)
                .build()
            )
            .build()
        ).build();

    final IKeyring ephemeralKeyring =
 materialProviders.CreateRawEcdhKeyring(ephemeralInput);
}
```

Python

以下示例使用密钥协议架构创建一个 Raw ECDH `RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey` 密钥环。加密后，密钥环将在指定ECC_NIST_P256曲线上本地创建一个新的密钥对。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey,
    EphemeralPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)
```

```
# Your get_public_key_bytes must return a DER-encoded X.509 public key
recipient_public_key = get_public_key_bytes()

# Create the raw ECDH ephemeral private key keyring
ephemeral_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey(
        EphemeralPrivateKeyToStaticPublicKeyInput(
            recipient_public_key = recipient_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(ephemeral_input)
```

Rust

以下示例使用密钥协议架构创建一个 Raw ECDH `ephemeral_raw_ecdh_static_configuration` 密钥环。加密后，密钥环将在指定曲线上本地创建一个新的密钥对。

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Load public key from UTF-8 encoded PEM files into a DER encoded public key.
let public_key_file_content =
 std::fs::read_to_string(Path::new(EXAMPLE_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content = parse(public_key_file_content)?;
let public_key_recipient_utf8_bytes = parsed_public_key_file_content.contents();

// Create EphemeralPrivateKeyToStaticPublicKeyInput
```

```
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
```

```
// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Load public key from UTF-8 encoded PEM files into a DER encoded public key
publicKeyRecipient, err := LoadPublicKeyFromPEM(eccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create EphemeralPrivateKeyToStaticPublicKeyInput
ephemeralRawEcdhStaticConfigurationInput :=
 mpltypes.EphemeralPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey: publicKeyRecipient,
}
ephemeralRawECDHStaticConfiguration :=
    mpltypes.RawEcdhStaticConfigurationsMemberEphemeralPrivateKeyToStaticPublicKey{
        Value: ephemeralRawEcdhStaticConfigurationInput,
    }

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create raw ECDH ephemeral private key keyring
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: &ephemeralRawECDHStaticConfiguration,
}
ecdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

PublicKeyDiscovery

解密时，最佳做法是指定 AWS Encryption SDK 可以使用的包装密钥。要遵循此最佳实践，请使用同时指定发件人私钥和收件人公钥的 ECDH 密钥环。但是，您也可以创建 Raw ECDH 发现密钥环，即 Raw ECDH 密钥环，它可以解密任何消息，其中指定密钥的公钥与存储在邮件密文中的收件人的公钥相匹配。此密钥协议架构只能解密消息。

> **Important**
>
> 使用密`PublicKeyDiscovery`钥协议架构解密消息时，无论谁拥有所有公钥，都将接受所有公钥。

要使用密钥协议架构初始化 Raw ECDH `PublicKeyDiscovery` 密钥环，请提供以下值：

- 收件人的静态私钥

  您必须提供收件人的 PEM 编码私钥（PKCS #8 PrivateKeyInfo 结构），如 RFC 5958 中所定义。
- 曲线规格

  标识指定私钥中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：`ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

C# / .NET

以下示例使用密钥协议架构创建一个 Raw ECDH `PublicKeyDiscovery` 密钥环。此密钥环可以解密任何消息，其中指定私钥的公钥与存储在消息密文中的收件人的公钥相匹配。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePrivateKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH discovery keyring
     var discoveryConfiguration = new RawEcdhStaticConfigurations()
     {
      PublicKeyDiscovery = new PublicKeyDiscoveryInput
      {
       RecipientStaticPrivateKey = AlicePrivateKey
      }
```

```
    };

    var createKeyringInput = new CreateRawEcdhKeyringInput()
    {
     CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
     KeyAgreementScheme = discoveryConfiguration
    };

    var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下示例使用密钥协议架构创建一个 Raw ECDH `PublicKeyDiscovery` 密钥环。此密钥环可以解密任何消息，其中指定私钥的公钥与存储在消息密文中的收件人的公钥相匹配。

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Python

以下示例使用密钥协议架构创建一个 Raw ECDH RawEcdhStaticConfigurationsPublicKeyDiscovery 密钥环。此密钥环可以解密任何消息，其中指定私钥的公钥与存储在消息密文中的收件人的公钥相匹配。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsPublicKeyDiscovery,
    PublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

# Instantiate the material providers library
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Your get_private_key_bytes must return a PEM-encoded private key
recipient_private_key = get_private_key_bytes()

# Create the raw ECDH discovery keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = RawEcdhStaticConfigurationsPublicKeyDiscovery(
        PublicKeyDiscoveryInput(
            recipient_static_private_key = recipient_private_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

以下示例使用密钥协议架构创建一个 Raw ECDH discovery_raw_ecdh_static_configuration 密钥环。此密钥环可以解密任何消息，其中指定私钥的公钥与存储在消息密文中的收件人的公钥相匹配。

```
// Instantiate the AWS Encryption SDK client and material providers library
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
```

```
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;


// Optional: Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

// Load keys from UTF-8 encoded PEM files.
let mut file = File::open(Path::new(EXAMPLE_ECC_PRIVATE_KEY_FILENAME_RECIPIENT))?;
let mut private_key_recipient_utf8_bytes = Vec::new();
file.read_to_end(&mut private_key_recipient_utf8_bytes)?;

// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
```

```
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Load keys from UTF-8 encoded PEM files.
privateKeyRecipient, err := os.ReadFile(eccPrivateKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create PublicKeyDiscoveryInput
discoveryRawEcdhStaticConfigurationInput := mpltypes.PublicKeyDiscoveryInput{
    RecipientStaticPrivateKey: privateKeyRecipient,
}
```

```
discoveryRawEcdhStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberPublicKeyDiscovery{
    Value: discoveryRawEcdhStaticConfigurationInput,
}

// Create raw ECDH discovery private key keyring
discoveryRawEcdhKeyringInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: discoveryRawEcdhStaticConfiguration,
}

discoveryRawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 discoveryRawEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

Multi-keyrings

您可以将多个密钥环合并为一个多重密钥环。多重密钥环 是由一个或多个相同或不同类型的密钥环组成的密钥环。效果与连续使用多个密钥环类似。在使用多重密钥环加密数据时，其任意密钥环中的任意包装密钥都可以对数据进行解密。

在创建多重密钥环以加密数据时，您可以将一个密钥环指定为生成器密钥环。所有其他密钥环称为子密钥环。生成器密钥环生成并加密明文数据密钥。然后，所有子密钥环中的所有包装密钥都加密相同的明文数据密钥。对于多重密钥环中的每个包装密钥，多重密钥环都返回明文密钥和一个加密的数据密钥。如果生成器密钥环是 KMS 密钥环， AWS KMS 密钥环中的生成器密钥将生成并加密明文密钥。然后，密钥环 AWS KMS keys 中的所有其他密钥，以及多密 AWS KMS 钥环中所有子密钥环中的所有封装密钥，都将加密相同的纯文本密钥。

如果您创建了一个没有生成器密钥环的多密钥环，则可以单独使用它来解密数据，但不能用于加密。或者，要在加密操作中使用不带生成器密钥环的多密钥环，可以在另一个多密钥环中将其指定为子密钥环。没有生成器密钥环的多密钥环不能被指定为另一个多密钥环中的生成器密钥环。

解密时， AWS Encryption SDK 使用密钥环尝试解密其中一个加密的数据密钥。密钥环是按照在多重密钥环中指定的顺序调用的。只要任何密钥环中的任何密钥可以解密加密的数据密钥，处理就会立即停止。

从 1.7 版本开始。 x，当加密的数据密钥在 AWS Key Management Service (AWS KMS) 密钥环（或主密钥提供程序）下加密时， AWS Encryption SDK 始终会将的密钥 ARN 传递给解密KeyId操作 AWS KMS 的参数。 AWS KMS key 这是一种 AWS KMS 最佳实践，可确保您使用要使用的包装密钥解密加密的数据密钥。

要查看多重密钥环的有效示例，请参阅：

- C：multi_keyring.cpp
- C# /.NET: MultiKeyringExample.cs
- JavaScript Node.js：multi_keyring.ts
- JavaScript 浏览器：m ult i_keyring.ts
- Java：MultiKeyringExample.java
- Python：multi_keyring_example.py

要创建多重密钥环，首先要实例化子密钥环。在此示例中，我们使用 AWS KMS 密钥环和 Raw AES 密钥环，但您可以将任何支持的密钥环组合到一个多密钥环中。

C

```
/* Define an AWS KMS keyring. For details, see string.cpp */
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(example_key);

// Define a Raw AES keyring. For details, see raw_aes_keyring.c */
struct aws_cryptosdk_keyring *aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
        alloc, wrapping_key_namespace, wrapping_key_name, wrapping_key,
 AWS_CRYPTOSDK_AES256);
```

C# / .NET

```
// Define an AWS KMS keyring. For details, see AwsKmsKeyringExample.cs.
var kmsKeyring = materialProviders.CreateAwsKmsKeyring(createKmsKeyringInput);

// Define a Raw AES keyring. For details, see RawAESKeyringExample.cs.
var aesKeyring = materialProviders.CreateRawAesKeyring(createAesKeyringInput);
```

JavaScript Browser

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  RawAesKeyringWebCrypto,
  RawAesWrappingSuiteIdentifier,
  MultiKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
  synchronousRandomValues,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const clientProvider = getClient(KMS, { credentials })

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringBrowser({ generatorKeyId: exampleKey })

// Define a Raw AES keyring. For details, see aes_simple.ts.
const aesKeyring = new RawAesKeyringWebCrypto({ keyName, keyNamespace,
 wrappingSuite, masterKey })
```

JavaScript Node.js

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

```
import {
  MultiKeyringNode,
  KmsKeyringNode,
  RawAesKeyringNode,
  RawAesWrappingSuiteIdentifier,
  buildClient,
```

```
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringNode({ generatorKeyId: exampleKey })

// Define a Raw AES keyring. For details, see raw_aes_keyring_node.ts.
const aesKeyring = new RawAesKeyringNode({ keyName, keyNamespace, wrappingSuite,
 unencryptedMasterKey })
```

Java

```
// Define the raw AES keyring.
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateRawAesKeyringInput createRawAesKeyringInput =
 CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// Define the AWS KMS keyring.
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Python

以下示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。REQUIRE_ENCRYPT_REQUIRE_DECRYPT

```
# Create the AWS KMS keyring
```

```
kms_client = boto3.client('kms', region_name="us-west-2")

mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

kms_keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    generator=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    kms_client=kms_client
)

kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
    input=kms_keyring_input
)

# Create Raw AES keyring
key_name_space = "HSM_01"
key_name = "AES_256_012"

raw_aes_keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=raw_aes_keyring_input
)
```

Rust

```
// Instantiate the AWS Encryption SDK client
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create the AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Instantiate the material providers library
```

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

// Create a Raw AES keyring
let key_namespace: &str = "my-key-namespace";
let key_name: &str = "my-aes-key-name";

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
```

```
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  kmsKeyId,
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Create a Raw AES keyring
var keyNamespace = "my-key-namespace"
var keyName = "my-aes-key-name"

aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  AESWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
```

接下来创建多重密钥环并指定其生成器密钥环（如果有）。在此示例中，我们创建了一个多密钥环，其中密钥环是生成器 AWS KMS 密钥环，AES 密钥环是子密钥环。

C

在 C 的多重密钥环构造函数中，您仅指定其生成器密钥环。

```
struct aws_cryptosdk_keyring *multi_keyring = aws_cryptosdk_multi_keyring_new(alloc,
 kms_keyring);
```

要将子密钥环添加到多重密钥环中，请使用 `aws_cryptosdk_multi_keyring_add_child` 方法。您需要为添加的每个子密钥环调用一次该方法。

```
// Add the Raw AES keyring (C only)
aws_cryptosdk_multi_keyring_add_child(multi_keyring, aes_keyring);
```

C# / .NET

.NET `CreateMultiKeyringInput` 构造函数允许您定义生成器密钥环和子密钥环。生成的 `CreateMultiKeyringInput` 对象不可变。

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = kmsKeyring,
    ChildKeyrings = new List<IKeyring>() {aesKeyring}
};

var multiKeyring = materialProviders.CreateMultiKeyring(createMultiKeyringInput);
```

JavaScript Browser

JavaScript 多钥匙圈是不可变的。 JavaScript 多密钥环构造函数允许您指定生成器密钥环和多个子密钥环。

```
const clientProvider = getClient(KMS, { credentials })

const multiKeyring = new MultiKeyringWebCrypto(generator: kmsKeyring, children:
 [aesKeyring]);
```

JavaScript Node.js

JavaScript 多钥匙圈是不可变的。 JavaScript 多密钥环构造函数允许您指定生成器密钥环和多个子密钥环。

```
const multiKeyring = new MultiKeyringNode(generator: kmsKeyring, children:
 [aesKeyring]);
```

Java

Java `CreateMultiKeyringInput` 构造函数允许您定义生成器密钥环和子密钥环。生成的 `createMultiKeyringInput` 对象不可变。

```
final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
        .generator(awsKmsMrkMultiKeyring)
        .childKeyrings(Collections.singletonList(rawAesKeyring))
        .build();
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

Python

```
multi_keyring_input: CreateMultiKeyringInput = CreateMultiKeyringInput(
    generator=kms_keyring,
    child_keyrings=[raw_aes_keyring]
)

multi_keyring: IKeyring = mat_prov.create_multi_keyring(
    input=multi_keyring_input
)
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(kms_keyring.clone())
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

Go

```
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
  Generator:     awsKmsKeyring,
  ChildKeyrings: []mpltypes.IKeyring{rawAESKeyring},
 }
 multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
 if err != nil {
  panic(err)
 }
```

现在，您就可以使用此多重密钥环加密和解密数据了。

AWS Encryption SDK 编程语言

AWS Encryption SDK 适用于以下编程语言。所有语言实施都是可互操作的。您可以使用一种语言实施进行加密，并使用另一种语言实施进行解密。互操作性可能受到语言约束的限制。如果是这样，这些约束将在有关语言实施的主题中进行描述。此外，在加密和解密时，必须使用兼容的密钥环或主密钥和主密钥提供程序。有关详细信息，请参阅the section called "密钥环兼容性"。

主题

- AWS Encryption SDK for C
- AWS Encryption SDK 对于.NET
- AWS Encryption SDK for Go
- AWS Encryption SDK for Java
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK for Python
- AWS Encryption SDK 对于 Rust
- AWS Encryption SDK 命令行界面

AWS Encryption SDK for C

为使用 C 语言编写应用程序的开发人员 AWS Encryption SDK for C 提供了一个客户端加密库。它也是用更高级编程语言实现 AWS Encryption SDK 的基础。

与的所有实现一样 AWS Encryption SDK， AWS Encryption SDK for C 提供了高级数据保护功能。这些功能包括信封加密、其他经过身份验证的数据 (AAD) 以及安全、经过身份验证且对称的密钥算法套件，如具有密钥派生和签名的 256 位 AES-GCM。

的所有特定于语言的实现 AWS Encryption SDK 都是完全可互操作的。例如，您可以使用加密数据，AWS Encryption SDK for C 并使用任何支持的语言实现对其进行解密，包括加密 AWS CLI。

AWS Encryption SDK for C 要求与 AWS Key Management Service (AWS KMS) 适用于 C++ 的 AWS SDK 进行交互。仅当您使用可选的 AWS KMS 密钥环时，才需要使用该工具。但是， AWS Encryption SDK 不需要 AWS KMS 或任何其他 AWS 服务。

了解更多

- 有关使用编程的详细信息 AWS Encryption SDK for C，请参阅 C 示例 GitHub、上aws-encryption-sdk-c 存储库中的示例以及 AWS Encryption SDK for C API 文档。
- 有关如何使用加密数据以便可以将其解密为多个区域的讨论 AWS 区域，请参阅安全博客中的如何使用 C 语言解密多个区域中的密文。 AWS Encryption SDK for C AWS Encryption SDK AWS

主题

- 正在安装 AWS Encryption SDK for C
- 使用 AWS Encryption SDK for C
- AWS Encryption SDK for C 例子

正在安装 AWS Encryption SDK for C

安装最新版本的 AWS Encryption SDK for C。

> **Note**
>
> 2.0.0 AWS Encryption SDK for C 之前的所有版本都处于该阶段。end-of-support
> 您可以安全地从 AWS Encryption SDK for C 版本 2.0.x 及更高版本更新为最新版本，无需更改任何代码或数据。但是，版本 2.0.x 中引入了新的安全功能，不向后兼容。要从 1.7.x 之前的版本更新到 2.0.x 及更高版本，必须先更新到 AWS Encryption SDK for C最新版本 1.x。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

您可以在aws-encryption-sdk-c存储库的 README 文件 AWS Encryption SDK for C中找到有关安装和构建的详细说明。其中包括在 Amazon Linux、Ubuntu、macOS 和 Windows 平台上进行构建的说明。

开始之前，请决定是否要在 AWS Encryption SDK中使用 AWS KMS 密钥环。如果您使用 AWS KMS 钥匙圈，则需要安装。 适用于 C++ 的 AWS SDK需要使用 AWS SDK 才能与 AWS Key Management Service(AWS KMS) 进行交互。使用 AWS KMS 密钥环时， AWS Encryption SDK AWS KMS 用于生成和保护保护数据的加密密钥。

适用于 C++ 的 AWS SDK 如果您使用的是其他密钥环类型，例如原始的 AES 密钥环、原始 RSA 密钥环或不包含密钥环的多密钥环，则无需安装。 AWS KMS 但是，使用原始密钥环类型时，您需要生成并保护自己的原始包装密钥。

如果您在安装时遇到问题，请在 aws-encryption-sdk-c 存储库中提交问题或使用此页面上的任何反馈链接。

使用 AWS Encryption SDK for C

本主题解释了其他编程语言实现中不支持的某些功能。 AWS Encryption SDK for C

本节中的示例说明了如何使用 AWS Encryption SDK for C版本 2.0.x 及更高版本。有关使用早期版本的示例，请在aws-encryption-sdk-c 存储库存储库的版本列表中找到您的版本 GitHub。

有关使用编程的详细信息 AWS Encryption SDK for C，请参阅 C 示例 GitHub、上aws-encryption-sdk-c 存储库中的示例以及 AWS Encryption SDK for C API 文档。

另请参阅：密钥环

主题

- 加密和解密数据的模式
- 引用计数

加密和解密数据的模式

使用时 AWS Encryption SDK for C，将遵循类似于以下的模式：创建密钥环，创建使用密钥环的 CMM，创建使用 CMM（和密钥环）的会话，然后处理会话。

1. 加载错误字符串。

   在 C 或 C++ 代码中调用 aws_cryptosdk_load_error_strings() 方法。该方法加载对调试非常有用的错误信息。

   您只需要调用一次，例如在 main 方法中调用。

   ```
   /* Load error strings for debugging */
   aws_cryptosdk_load_error_strings();
   ```

2. 创建一个密钥环。

   使用要用于加密数据密钥的包装密钥配置密钥环。此示例使用带AWS KMS 钥匙圈的密钥环 AWS KMS key，但您可以使用任何类型的密钥环代替它。

要 AWS KMS key 在中的加密密钥环中识别 AWS Encryption SDK for C，请指定密钥 ARN 或别名 ARN。在解密密钥环中，您必须使用密钥 ARN。有关更多信息，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

```
const char * KEY_ARN = "arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
struct aws_cryptosdk_keyring *kms_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(KEY_ARN);
```

3. 创建会话。

   在中 AWS Encryption SDK for C，您可以使用会话来加密一条纯文本消息或解密一条密文消息，无论其大小如何。会话在整个处理过程中维护消息的状态。

   使用分配器、密钥环和模式（AWS_CRYPTOSDK_ENCRYPT 或 AWS_CRYPTOSDK_DECRYPT）配置会话。如果需要更改会话模式，请使用 aws_cryptosdk_session_reset 方法。

   当您使用密钥环创建会话时， AWS Encryption SDK for C 会自动为您创建默认的加密材料管理器 (CMM)。您无需创建、维护或销毁该对象。

   例如，以下会话使用分配器以及在步骤 1 中定义的密钥环。在加密数据时，模式为 AWS_CRYPTOSDK_ENCRYPT。

```
struct aws_cryptosdk_session * session =
 aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);
```

4. 加密或解密数据。

   要处理会话中的数据，请使用 aws_cryptosdk_session_process 方法。如果输入缓冲区足够大，可以存放整个明文，并且输出缓冲区足够大，可以存放整个加密文字，则可以调用 aws_cryptosdk_session_process_full。不过，如果需要处理串流数据，您可以循环调用 aws_cryptosdk_session_process。有关示例，请参阅 file_streaming.cpp 示例。aws_cryptosdk_session_process_full在 1.9 AWS Encryption SDK 版本中引入。 x 和 2.2。 x。

   将会话配置为加密数据时，明文字段描述输入，密文字段描述输出。plaintext 字段包含要加密的消息，ciphertext 字段接收加密方法返回的加密的消息。

```
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
```

```
                                    ciphertext,
                                    ciphertext_buffer_size,
                                    &ciphertext_length,
                                    plaintext,
                                    plaintext_length)
```

将会话配置为解密数据时，密文字段描述输入，明文字段描述输出。ciphertext 字段包含加密方法返回的加密的消息，plaintext 字段接收解密方法返回的明文消息。

要解密数据，请调用 aws_cryptosdk_session_process_full 方法。

```
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
                                    plaintext,
                                    plaintext_buffer_size,
                                    &plaintext_length,
                                    ciphertext,
                                    ciphertext_length)
```

引用计数

为了防止内存泄漏，在使用完创建的所有对象时，请务必释放对它们的引用。否则，可能会造成内存泄漏。本开发工具包提供了简化该任务的方法。

每次使用一个以下子对象创建父对象时，父对象都会获取并保持对该子对象的引用，如下所示：

- 密钥环，例如，使用密钥环创建会话
- 默认加密材料管理器（CMM），例如，使用默认 CMM 创建会话或自定义 CMM
- 数据密钥缓存，例如，使用密钥环和缓存创建缓存 CMM

除非需要对子对象进行单独的引用，否则，您可以在创建父对象后立即释放对子对象的引用。在销毁父对象时，将释放对子对象的其余引用。该模式确保您只在需要时维护对每个对象的引用，不会因为未释放的引用而造成内存泄漏。

您仅负责释放对您明确创建的子对象的引用。您不负责管理对该开发工具包创建的任何对象的引用。如果该软件开发工具包创建一个对象（例如，aws_cryptosdk_caching_cmm_new_from_keyring 方法添加到会话中的默认 CMM），该软件开发工具包将管理该对象及其引用的创建和销毁过程。

在以下示例中，在使用密钥环创建会话时，会话将获取对密钥环的引用并保持该引用，直到销毁会话为止。如果不需要保持对密钥环的其他引用，您可以在创建会话后立即使用 `aws_cryptosdk_keyring_release` 方法释放密钥环对象。该方法将减少密钥环的引用计数。在调用 `aws_cryptosdk_session_destroy` 以销毁会话时，将释放会话对密钥环的引用。

```
// The session gets a reference to the keyring.
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT, keyring);

// After you create a session with a keyring, release the reference to the keyring
 object.
aws_cryptosdk_keyring_release(keyring);
```

对于更复杂的任务（例如，将密钥环重复用于多个会话或在 CMM 中指定算法套件），您可能需要保持对该对象的单独引用。如果是这样，请不要立即调用 release 方法，而是在不再使用这些对象时，除了销毁会话以外，还要释放引用。

当您使用其他方法（例如缓存 CMM）进行数据密钥缓存时 CMMs，这种引用计数技术也适用。从缓存和密钥环中创建缓存 CMM 时，缓存 CMM 将获取对两个对象的引用。除非您需要使用这些 CMM 执行其他任务，否则，您可以在创建缓存 CMM 后立即释放对缓存和密钥环的单独引用。然后，在使用缓存 CMM 创建会话时，您可以释放对缓存 CMM 的引用。

请注意，您仅负责释放对您明确创建的对象的引用。方法创建的对象（例如，作为缓存 CMM 基础的默认 CMM）是由该方法管理的。

```
/ Create the caching CMM from a cache and a keyring.
struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL, 60,
 AWS_TIMESTAMP_SECS);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);

// Create a session with the caching CMM.
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm_2(allocator,
 AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

// Release your references to the caching CMM.
aws_cryptosdk_cmm_release(caching_cmm);
```

```
// ...

aws_cryptosdk_session_destroy(session);
```

AWS Encryption SDK for C 例子

以下示例向您展示了如何使用 AWS Encryption SDK for C 来加密和解密数据。

本节中的示例说明了如何使用版本 2.0.x 及更高版本的 AWS Encryption SDK for C。有关使用早期版本的示例，请在aws-encryption-sdk-c 存储库存储库的版本列表中找到您的版本 GitHub。

安装和构建时 AWS Encryption SDK for C，这些示例和其他示例的源代码包含在`examples`子目录中，它们会被编译并内置到该`build`目录中。您也可以在上aws-encryption-sdk-c GitHub存储库的示例子目录中找到它们。

主题

- 加密和解密字符串

加密和解密字符串

以下示例向您展示了如何使用 AWS Encryption SDK for C 来加密和解密字符串。

此示例以AWS KMS 密钥环为特色，这是一种使用 AWS KMS key in AWS Key Management Service (AWS KMS) 生成和加密数据密钥的密钥环。该示例包括用 C++ 编写的代码。使用 AWS KMS 钥匙圈 AWS KMS 时 AWS Encryption SDK for C 适用于 C++ 的 AWS SDK 需要调用。如果您使用的是不与之交互的密钥环 AWS KMS，例如原始的 AES 密钥环、未处理的 RSA 密钥环或不包含密钥环的多密钥环，则 AWS KMS 不需要使用。 适用于 C++ 的 AWS SDK

有关创建的帮助 AWS KMS key，请参阅《AWS Key Management Service 开发者指南》中的创建密钥。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

请参阅完整的代码示例：string.cpp

主题

- 加密字符串
- 解密字符串

加密字符串

本示例的第一部分使用带有密钥环的 AWS KMS 密钥环 AWS KMS key 来加密纯文本字符串。

步骤 1：加载错误字符串。

在 C 或 C++ 代码中调用 `aws_cryptosdk_load_error_strings()` 方法。该方法加载对调试非常有用的错误信息。

您只需要调用一次，例如在 `main` 方法中调用。

```
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();
```

步骤 2：构造密钥环。

创建用于加密的 AWS KMS 密钥环。本示例中的密钥环配置为一个 AWS KMS key，但您可以将多个密钥环配置为多个 AWS KMS 密钥环 AWS KMS keys，包括在不同 AWS 区域 和不同的账户 AWS KMS keys 中。

要 AWS KMS key 在中的加密密钥环中识别 AWS Encryption SDK for C，请指定密钥 ARN 或别名 ARN。在解密密钥环中，您必须使用密钥 ARN。有关更多信息，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

在 AWS KMS 钥匙圈 AWS KMS keys 中识别

创建包含多个密钥的密钥环时 AWS KMS keys，可以指定 AWS KMS key 用于生成和加密纯文本数据密钥的，以及用于加密相同纯文本数据密钥的可选附加 AWS KMS keys 数组。在这种情况下，您只需要指定生成器 AWS KMS key。

在运行该代码之前，请将示例密钥 ARN 替换为有效 ARN。

```
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

struct aws_cryptosdk_keyring *kms_keyring =
        Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

步骤 3：创建会话。

使用分配器、模式枚举器和密钥环创建一个会话。

每个会话都需要一种模式：用于加密的 AWS_CRYPTOSDK_ENCRYPT 或用于解密的 AWS_CRYPTOSDK_DECRYPT。要更改现有会话的模式，请使用 aws_cryptosdk_session_reset 方法。

在使用密钥环创建一个会话后，您可以使用该开发工具包提供的方法释放对密钥环的引用。该会话在生命周期内保留对密钥环对象的引用。在销毁该会话时，将释放对密钥环和会话对象的引用。这种引用计数方法有助于防止内存泄漏，并防止在使用对象时将其释放。

```
struct aws_cryptosdk_session *session =
       aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

步骤 4：设置加密上下文。

加密上下文是任意的非机密其他经过身份验证的数据。当您提供有关加密的加密上下文时，会 AWS Encryption SDK 以加密方式将加密上下文绑定到密文，因此解密数据需要相同的加密上下文。使用加密上下文是可选的，但作为一项最佳实践，建议您提供加密上下文。

首先，创建一个包含加密上下文字符串的哈希表。

```
/* Allocate a hash table for the encryption context */
int set_up_enc_ctx(struct aws_allocator *alloc, struct aws_hash_table *my_enc_ctx)

// Create encryption context strings
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key1, "Example");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value1, "String");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key2, "Company");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value2, "MyCryptoCorp");

// Put the key-value pairs in the hash table
aws_hash_table_put(my_enc_ctx, enc_ctx_key1, (void *)enc_ctx_value1, &was_created)
aws_hash_table_put(my_enc_ctx, enc_ctx_key2, (void *)enc_ctx_value2, &was_created)
```

获取会话中加密上下文的可变指针。然后，使用 aws_cryptosdk_enc_ctx_clone 函数将加密上下文复制到会话中。请将副本保留在 my_enc_ctx 中，以便在解密数据后验证该值。

加密上下文是会话的一部分，而不是传递给会话进程函数的参数。这可保证对消息的每个分段使用相同的加密上下文，即使多次调用会话进程函数来加密整个消息也是如此。

```
struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);

aws_cryptosdk_enc_ctx_clone(alloc, session_enc_ctx, my_enc_ctx)
```

步骤 5：加密字符串。

要加密明文字符串，请使用 `aws_cryptosdk_session_process_full` 方法和使用加密模式的会话。此方法在 1.9 AWS Encryption SDK 版本中引入。 x 和 2.2。 x，专为非流式加密和解密而设计。要处理串流数据，请在循环中调用 `aws_cryptosdk_session_process`。

加密时，明文字段为输入字段；密文字段为输出字段。处理完成后，`ciphertext_output` 字段包含加密的消息，其中包括实际密文、加密的数据密钥和加密上下文。您可以对任何支持的编程语言使用来解密此加密消息。 AWS Encryption SDK

```
/* Gets the length of the plaintext that the session processed */
size_t ciphertext_len_output;
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                  ciphertext_output,
                                  ciphertext_buf_sz_output,
                                  &ciphertext_len_output,
                                  plaintext_input,
                                  plaintext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 8;
}
```

步骤6：清理会话。

最后一步将销毁会话，包括对 CMM 和密钥环的引用。

您也可以根据需要重复使用具有相同密钥环和 CMM 的会话来解密字符串，或加密/解密其他消息，而不是销毁会话。要使用会话进行解密，请使用 `aws_cryptosdk_session_reset` 方法将模式更改为 `AWS_CRYPTOSDK_DECRYPT`。

解密字符串

本示例的第二部分将对包含原始字符串密文的加密消息进行解密。

步骤 1：加载错误字符串。

在 C 或 C++ 代码中调用 `aws_cryptosdk_load_error_strings()` 方法。该方法加载对调试非常有用的错误信息。

您只需要调用一次，例如在 `main` 方法中调用。

```
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();
```

步骤 2：构造密钥环。

当你解密数据时 AWS KMS，你会传入加密 API 返回的加密消息。解密 API 不以输入 AWS KMS key 为输入。相反， AWS KMS 使用与加密密文相同的方法 AWS KMS key 来解密密文。但是， AWS Encryption SDK 允许您使用加密和解密来 AWS KMS keys 指定密 AWS KMS 钥环。

在解密时，您可以仅使用要用于解密加密消息 AWS KMS keys 的密钥环来配置密钥环。例如，您可能想创建一个仅包含组织中特定角色使用的密钥环。 AWS KMS key AWS KMS key 除非它出现在解密密钥环中，否则永远不会使用它。 AWS Encryption SDK 如果 SDK 无法使用您提供的密钥环 AWS KMS keys 中的来解密加密的数据密钥，要么是因为密钥环 AWS KMS keys 中没有使用任何数据密钥来加密任何数据密钥，要么是因为调用者无权使用密钥环 AWS KMS keys 中的进行解密，则解密调用将失败。

AWS KMS key 为解密密钥环指定时，必须使用其密钥 ARN。 仅允许 ARNs在加密密钥环中使用 @@ 别名。有关识别 AWS KMS 钥匙圈 AWS KMS keys 中的的帮助，请参阅在 AWS KMS 钥匙圈 AWS KMS keys 中识别。

在此示例中，我们指定了一个密钥环，该密钥环配置为 AWS KMS key 用于加密字符串的密钥环。在运行该代码之前，请将示例密钥 ARN 替换为有效 ARN。

```
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"

struct aws_cryptosdk_keyring *kms_keyring =
        Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

步骤 3：创建会话。

使用分配器和密钥环创建一个会话。要配置解密会话，请使用 `AWS_CRYPTOSDK_DECRYPT` 模式配置会话。

在使用密钥环创建一个会话后，您可以使用该开发工具包提供的方法释放对密钥环的引用。该会话在生命周期内保留对密钥环对象的引用，在销毁该会话时，将会释放会话和密钥环。这种引用计数方法有助于防止内存泄漏，并防止在使用对象时将其释放。

```
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

步骤 4：解密字符串。

要解密字符串，请使用 aws_cryptosdk_session_process_full 方法和配置用于解密的会话。此方法在 AWS Encryption SDK 版本 1.9.x 和 2.2.x 中引入，专为非串流加密和解密而设计。要处理串流数据，请在循环中调用 aws_cryptosdk_session_process。

解密时，密文字段为输入字段，明文字段是输出字段。ciphertext_input 字段包含加密方法返回的加密消息。处理完成后，plaintext_output 字段包含明文（解密后的）字符串。

```
size_t plaintext_len_output;

if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                  plaintext_output,
                                  plaintext_buf_sz_output,
                                  &plaintext_len_output,
                                  ciphertext_input,
                                  ciphertext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 13;
}
```

步骤 5：验证加密上下文。

请确保实际的加密上下文（用于解密消息的上下文）包含您在加密消息时提供的加密上下文。实际加密上下文可能包含额外的键值对，因为加密材料管理器 (CMM) 可以在加密消息前向提供的加密上下文添加键值对。

在中 AWS Encryption SDK for C，解密时无需提供加密上下文，因为加密上下文包含在 SDK 返回的加密消息中。但是，在其返回明文消息前，您的解密函数应验证用于解密消息的加密上下文中包含所提供的加密上下文中的所有键值对。

首先，获取会话中哈希表的只读指针。该哈希表包含用于解密消息的加密上下文。

```
const struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr(session);
```

然后，遍历加密时复制的 `my_enc_ctx` 哈希表中的加密上下文。验证用于解密的 `session_enc_ctx` 哈希表包含用于加密的 `my_enc_ctx` 哈希表中的所有键值对。如果缺少任何密钥，或密钥具有不同的值，则停止处理并输出错误消息。

```
for (struct aws_hash_iter iter = aws_hash_iter_begin(my_enc_ctx); !
aws_hash_iter_done(&iter);
      aws_hash_iter_next(&iter)) {
     struct aws_hash_element *session_enc_ctx_kv_pair;
     aws_hash_table_find(session_enc_ctx, iter.element.key,
 &session_enc_ctx_kv_pair)

    if (!session_enc_ctx_kv_pair ||
        !aws_string_eq(
            (struct aws_string *)iter.element.value, (struct aws_string
 *)session_enc_ctx_kv_pair->value)) {
        fprintf(stderr, "Wrong encryption context!\n");
        abort();
    }
}
```

步骤6：清理会话。

验证加密上下文后，可以销毁会话，或重用会话。如果需要重新配置会话，请使用 `aws_cryptosdk_session_reset` 方法。

```
aws_cryptosdk_session_destroy(session);
```

AWS Encryption SDK 对于.NET

fo AWS Encryption SDK r .NET 是一个客户端加密库，适用于使用 C# 和其他.NET 编程语言编写应用程序的开发人员。它在 Windows、macOS 和 Linux 上受支持。

Note

AWS Encryption SDK 适用于.NET 的 4.0.0 版本与《消息规范》 AWS Encryption SDK 有所不同。因此，由 4.0.0 版加密的消息只能通过.NET 版本 4.0.0 或更高版本进行解密。 AWS Encryption SDK 任何其他编程语言都无法对其进行解密。

for .NET AWS Encryption SDK 的 4.0.1 版根据消息规范写入 AWS Encryption SDK 消息，并且可与其他编程语言实现互操作。默认情况下，版本 4.0.1 可以读取版本 4.0.0 加密的消息。但如果您不想解密由版本 4.0.0 加密的消息，则可以指定 `NetV4_0_0_RetryPolicy` 属性来阻止客户端读取这些消息。有关更多信息，请参阅 aws-encryption-sdk存储库中的 v4.0.1 版本说明。 GitHub

f AWS Encryption SDK or .NET 与其他一些编程语言实现的不同之处 AWS Encryption SDK 在于：

- 不支持数据密钥缓存

  Note

  版本 4。 .NET AWS Encryption SDK 的 x 支持AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

- 不支持流数据
- 来自适用于 .NET 的 AWS Encryption SDK 的无日志记录或堆栈跟踪
- 需要 适用于 .NET 的 AWS SDK

.N AWS Encryption SDK ET 版包括 2.0 版中引入的所有安全功能。 x 及更高版本的其他语言实现 AWS Encryption SDK。但是，如果您使用 for .NET 来解密由 2.0 之前版本加密的数据。 AWS Encryption SDK x 版本的另一种语言实现 AWS Encryption SDK，您可能需要调整承诺政策。有关更多信息，请参阅 如何设置您的承诺策略。

f AWS Encryption SDK or .NET 是 Dafny AWS Encryption SDK 中的产物，这是一种正式的验证语言，你可以用它来编写规范、实现规范的代码以及测试规范。结果为在确保功能正确性的框架中实施 AWS Encryption SDK 功能的库。

了解更多

- 有关显示如何在中配置选项（例如指定备用算法套件 AWS Encryption SDK、限制加密数据密钥和使用 AWS KMS 多区域密钥）的示例，请参阅正在配置 AWS Encryption SDK。

- 有关使用 for .NET AWS Encryption SDK 进行编程的详细信息，请参阅上的 aws-encryption-sdk存储库aws-encryption-sdk-net目录 GitHub。

主题

- AWS Encryption SDK 为.NET 安装的
- AWS Encryption SDK 为.NET 调试
- AWS Encryption SDK 查看.NET 示例

AWS Encryption SDK 为.NET 安装的

.N AWS Encryption SDK ET 版作为AWS.Cryptography.EncryptionSDK软件包提供 NuGet。有关安装和构建.NET 版的 AWS Encryption SDK 详细信息，请参阅存储库中的 README.md 文件。aws-encryption-sdk-net

版本 3.x

版本 3。.NET 版 AWS Encryption SDK 的 x 仅在 Windows 上支持.NET 框架 4.5.2 — 4.8。其在所有支持的操作系统中均支持 .NET Core 3.0+ 和 .NET 5.0 及更高版本。

版本 4.x

版本 4。.NET 的 AWS Encryption SDK x 支持.NET 6.0 和.NET Framework net48 及更高版本。版本 4。x 需要适用于.NET 的 AWS SDK v3。

版本 5.x

版本 5。.NET 的 AWS Encryption SDK x 支持.NET 6.0 和.NET Framework net48 及更高版本。版本 5。x 需要版本 2。材质提供者库 (MPL) 中的 x 和适用于.NET 的 AWS SDK v4。

适用于 .NET 的 SDK 即使你没有使用 AWS Key Management Service (AWS KMS) 键，for .NET 也需要。AWS Encryption SDK 它与 NuGet 软件包一起安装。但是，除非您使用的是 AWS KMS 密钥，AWS Encryption SDK 否则.NET 不需要 AWS 凭据或与任何 AWS 服务的交互。AWS 账户如需有关设置 AWS 账户的帮助，请参阅使用 wit AWS Encryption SDK h AWS KMS。

AWS Encryption SDK 为.NET 调试

fo AWS Encryption SDK r .NET 不生成任何日志。.NET 中的 AWS Encryption SDK 异常会生成异常消息，但不会生成堆栈跟踪。

为了帮助您进行调试，请务必在 适用于 .NET 的 SDK中启用日志记录功能。中的日志和错误消息 适用于 .NET 的 SDK 可以帮助您区分.NET 中出现的 适用于 .NET 的 SDK 错误和.NET 中出现 AWS Encryption SDK 的错误。有关 适用于 .NET 的 SDK 日志记录的帮助，请参阅AWSLogging《适用于 .NET 的 AWS SDK 开发人员指南》。（要查看该主题，请展开 Open to view .NET Framework content 部分。）

AWS Encryption SDK 查看.NET 示例

以下示例显示了您在使用 for .NET AWS Encryption SDK 进行编程时使用的基本编码模式。具体而言，您可以实例化材料提供者库 AWS Encryption SDK 和材料提供者库。然后，在调用每个方法之前，首先实例化定义该方法输入的对象。这与您在 适用于 .NET 的 SDK中使用的编码模式非常相似。

有关显示如何在中配置选项（例如指定备用算法套件 AWS Encryption SDK、限制加密数据密钥和使用 AWS KMS 多区域密钥）的示例，请参阅正在配置 AWS Encryption SDK。

有关使用 for .NET AWS Encryption SDK 进行编程的更多示例，请参阅上aws-encryption-sdk存储库aws-encryption-sdk-net目录中的示例 GitHub。

加密.NET 中的 AWS Encryption SDK 数据

此示例说明了数据加密的基本模式。它使用受一个 AWS KMS 包装密钥保护的数据密钥对一个小文件进行加密。

第 1 步：实例化材料提供者库 AWS Encryption SDK 和材料提供者库。

首先实例化材料提供者库 AWS Encryption SDK 和材料提供者库。您将使用中的方法 AWS Encryption SDK 来加密和解密数据。您将使用材料提供程序库中的方法创建密钥环，密钥环指定哪些密钥保护您的数据。

在版本 3 中，实例化材料提供者库 AWS Encryption SDK 和材质提供者库的方式有所不同。 x 和 4。 .NET AWS Encryption SDK 的 x。两个版本 3 的以下所有步骤都相同。 x 和 4。 .NET AWS Encryption SDK 的 x。

Version 3.x

```
// Instantiate the AWS Encryption SDK and material providers
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders(
```

Version 4.x

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```

步骤 2：为密钥环创建输入对象。

每个密钥环创建方法均有对应的输入对象类。例如，要为 `CreateAwsKmsKeyring()` 方法创建输入对象，请创建 `CreateAwsKmsKeyringInput` 类的实例。

尽管此密钥环的输入未指定生成器密钥，但 `KmsKeyId` 参数指定的单个 KMS 密钥即为生成器密钥。其生成并加密进行数据加密的数据密钥。

此输入对象需要 AWS KMS 客户端来获取 KMS 密钥。 AWS 区域 要创建 AWS KMS 客户端，请在中实例化该`AmazonKeyManagementServiceClient`类。 适用于 .NET 的 SDK调用不带参数的 `AmazonKeyManagementServiceClient()` 构造函数会创建具有默认值的客户端。

在 AWS Encryption SDK 用于.NET 加密的密 AWS KMS 钥环中，您可以使用密钥 ID、密钥 ARN、别名或别名 ARN 来识别 KMS 密钥。在用于解密的密 AWS KMS 钥环中，必须使用密钥 ARN 来识别每个 KMS 密钥。如果您计划重复使用加密密钥环进行解密，请将密钥 ARN 标识符用于所有 KMS 密钥。

```
string keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

步骤 3：创建密钥环。

要创建密钥环，请使用密钥环输入对象调用密钥环方法。此示例使用 `CreateAwsKmsKeyring()` 方法，该方法只需一个 KMS 密钥。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

步骤 4：定义加密上下文。

加密上下文是可选的，但强烈建议在中进行加密操作。 AWS Encryption SDK您可以定义一个或多个非机密键值对。

> **Note**
>
> 使用版本 4。 x AWS Encryption SDK 对于.NET，您可以使用所需的加密上下文 CMM 在所有加密请求中要求使用加密上下文。

```
// Define the encryption context
var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"}
};
```

步骤 5：创建用于加密的输入对象。

在调用 Encrypt() 方法之前，请创建 EncryptInput 类实例。

```
string plaintext = File.ReadAllText("C:\\Documents\\CryptoTest\\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
```

步骤 6：加密明文。

使用Encrypt()的方法，使用您定义 AWS Encryption SDK 的密钥环对纯文本进行加密。

Encrypt() 方法返回的 EncryptOutput 包含用于获取加密消息 (Ciphertext)、加密上下文和算法套件的方法。

```
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

步骤 7：获取加密消息。

for .N AWS Encryption SDK ET 中的`Decrypt()`方法采用`EncryptOutput`实例`Ciphertext`的成员。

`EncryptOutput` 对象的 `Ciphertext` 成员是加密消息，其为便携式对象，其中包括加密数据、加密数据密钥和元数据，包括加密上下文。您可以长时间安全存储加密消息，也可以将其提交给 `Decrypt()` 方法以便恢复明文。

```
var encryptedMessage = encryptOutput.Ciphertext;
```

在 for .NET 中以严格模式解密 AWS Encryption SDK

最佳实践建议您指定用于解密数据的密钥，该选项称为严格模式。仅 AWS Encryption SDK 使用您在密钥环中指定的 KMS 密钥来解密密文。解密密钥环中的密钥必须包含至少一个加密数据的密钥。

此示例说明了使用适用于 .NET 的 AWS Encryption SDK 在严格模式下进行解密的基本模式。

第 1 步：实例化 AWS Encryption SDK 和材料提供者库。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```

步骤 2：为密钥环创建输入对象。

要为密钥环方法指定参数，请创建输入对象。for .NET 中的 AWS Encryption SDK 每个密钥环方法都有一个对应的输入对象。由于此示例使用 `CreateAwsKmsKeyring()` 方法创建密钥环，因此会为输入实例化 `CreateAwsKmsKeyringInput` 类。

在解密密钥环中，您必须使用密钥 ARN 标识 KMS 密钥。

```
string keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
```

```
};
```

步骤 3：创建密钥环。

要创建解密密钥环，此示例使用 `CreateAwsKmsKeyring()` 方法和密钥环输入对象。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

步骤 4：创建用于解密的输入对象。

要为 `Decrypt()` 方法创建输入对象，请实例化 `DecryptInput` 类。

`DecryptInput()` 构造函数的 `Ciphertext` 参数需要 `Encrypt()` 方法返回的 `EncryptOutput` 对象 `Ciphertext` 的成员。`Ciphertext` 属性表示加密消息，其中包括加密数据、加密数据密钥和 AWS Encryption SDK 解密消息所需元数据。

使用版本 4。 x AWS Encryption SDK 对于.NET，您可以使用可选`EncryptionContext`参数在`Decrypt()`方法中指定您的加密上下文。

使用 `EncryptionContext` 参数验证加密时使用的加密上下文是否包含在用于解密加密文字的加密上下文中。如果您使用的是带签名的算法套件（例如默认算法套件），则会将成对 AWS Encryption SDK 添加到加密上下文中，包括数字签名。

```
var encryptedMessage = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

步骤 5：解密加密文字。

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

步骤 6：验证加密上下文 – 版本 3.x

版本 3 `Decrypt()` 的方法。 .NET AWS Encryption SDK 的 x 不采用加密上下文。其会从加密消息的元数据中获取加密上下文值。但是，在返回或使用明文之前，最佳做法是验证用于解密加密文字的加密上下文是否包含您在加密时提供的加密上下文。

验证加密时使用的加密上下文是否包含在用于解密加密文字的加密上下文中。如果您使用的是带签名的算法套件（例如默认算法套件），则会将成对 AWS Encryption SDK 添加到加密上下文中，包括数字签名。

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

在 for .NET 中使用发现密钥环进行解密 AWS Encryption SDK

您可以提供不指定任何 KMS 密钥的密钥环， AWS KMS Discovery 密钥环，而非指定用于解密的 KMS 密钥。发现 AWS Encryption SDK 密钥环允许使用任何加密数据的 KMS 密钥来解密数据，前提是调用者拥有密钥的解密权限。要获得最佳实践，请添加发现过滤器，将可用于特定分区的 KMS 密钥限制为特定 AWS 账户 分区的密钥。

fo AWS Encryption SDK r .NET 提供了一个基本的发现密钥环，它需要一个 AWS KMS 客户端，还有一个发现多密钥环，需要你指定一个或多个密钥环。 AWS 区域客户端和区域均限制可用于解密加密消息的 KMS 密钥。两个密钥环的输入对象均需要建议添加的发现筛选条件。

以下示例说明了使用 AWS KMS Discovery 密钥环和发现筛选条件解密数据的模式。

第 1 步：实例化材料提供者库 AWS Encryption SDK 和材料提供者库。

```
// Instantiate the AWS Encryption SDK and material providers
var esdk =  new ESDK(new AwsEncryptionSdkConfig());
var mpl = new MaterialProviders(new MaterialProvidersConfig());
```

步骤 2：为密钥环创建输入对象。

要为密钥环方法指定参数，请创建输入对象。for .NET 中的 AWS Encryption SDK 每个密钥环方法都有一个对应的输入对象。由于此示例使用 `CreateAwsKmsDiscoveryKeyring()` 方法创建密钥环，因此会为输入实例化 `CreateAwsKmsDiscoveryKeyringInput` 类。

```
List<string> accounts = new List<string> { "111122223333" };

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
    }
};
```

步骤 3：创建密钥环。

要创建解密密钥环，此示例使用 `CreateAwsKmsDiscoveryKeyring()` 方法和密钥环输入对象。

```
var discoveryKeyring =
 materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

步骤 4：创建用于解密的输入对象。

要为 `Decrypt()` 方法创建输入对象，请实例化 `DecryptInput` 类。`Ciphertext` 参数的值为 `Encrypt()` 方法返回的 `EncryptOutput` 对象的 `Ciphertext` 成员。

使用版本 4。 x AWS Encryption SDK 对于.NET，您可以使用可选`EncryptionContext`参数在`Decrypt()`方法中指定您的加密上下文。

使用 `EncryptionContext` 参数验证加密时使用的加密上下文是否包含在用于解密加密文字的加密上下文中。如果您使用的是带签名的算法套件（例如默认算法套件），则会将成对 AWS Encryption SDK 添加到加密上下文中，包括数字签名。

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = discoveryKeyring,
    EncryptionContext = encryptionContext // OPTIONAL

};
```

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

步骤 5：验证加密上下文 – 版本 3.x

版本 3 `Decrypt()` 的方法。 .NET AWS Encryption SDK 的 x 不开启加密上下文`Decrypt()`。其会从加密消息的元数据中获取加密上下文值。但是，在返回或使用明文之前，最佳做法是验证用于解密加密文字的加密上下文是否包含您在加密时提供的加密上下文。

验证加密时使用的加密上下文是否包含在曾用于解密加密文字的加密上下文中。如果您使用的是带签名的算法套件（例如默认算法套件），则会将成对 AWS Encryption SDK 添加到加密上下文中，包括数字签名。

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

AWS Encryption SDK for Go

本主题介绍如何安装和使用 fo AWS Encryption SDK r Go。有关使用 for Go AWS Encryption SDK 进行编程的详细信息，请参阅上 aws-encryption-sdk存储库的 go 目录 GitHub。

f AWS Encryption SDK or Go 与其他一些编程语言实现的不同之处 AWS Encryption SDK 在于：

- 不支持数据密钥缓存。但是，fo AWS Encryption SDK r Go 支持AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。
- 不支持流数据

fo AWS Encryption SDK r Go 包含 2.0 版本中引入的所有安全功能。 x 及更高版本的其他语言实现 AWS Encryption SDK。但是，如果您使用 for AWS Encryption SDK Go 来解密由 2.0 之前版本加密的数据。 x 版本的另一种语言实现 AWS Encryption SDK，您可能需要调整承诺政策。有关更多信息，请参阅 如何设置您的承诺策略。

f AWS Encryption SDK or Go 是 AWS Encryption SDK in Dafny 的产物，这是一种正式的验证语言，你可以用它来编写规范、实现规范的代码以及测试规范。结果为在确保功能正确性的框架中实施 AWS Encryption SDK 功能的库。

了解更多

- 有关显示如何在中配置选项（例如指定备用算法套件 AWS Encryption SDK、限制加密数据密钥和使用 AWS KMS 多区域密钥）的示例，请参阅正在配置 AWS Encryption SDK。
- 有关如何配置和使用 for Go AWS Encryption SDK 的示例，请参阅上 aws-encryption-sdk存储库中的 Go 示例 GitHub。

主题

- 先决条件
- 安装

先决条件

在安装 fo AWS Encryption SDK r Go 之前，请确保满足以下先决条件。

支持的 Go 版本

Go 需要 AWS Encryption SDK 使用 Go 1.23 或更高版本。

有关下载和安装 Go 的更多信息，请参阅 Go 安装。

安装

安装最新版本的 fo AWS Encryption SDK r Go。有关安装和构建 for Go AWS Encryption SDK 的详细信息，请参阅上存储库 go 目录中的 README.md。 aws-encryption-sdk GitHub

安装最新版本

- 安装 fo AWS Encryption SDK r Go

```
go get github.com/aws/aws-encryption-sdk/releases/go/encryption-sdk@latest
```

- 安装加密材料提供程序库 (MPL)

```
go get github.com/aws/aws-cryptographic-material-providers-library/releases/go/mpl
```

AWS Encryption SDK for Java

本主题介绍了如何安装和使用 AWS Encryption SDK for Java。有关使用编程的详细信息 AWS Encryption SDK for Java，请参阅上的aws-encryption-sdk-java存储库 GitHub。有关 API 文档，请参阅适用于 AWS Encryption SDK for Java的 Javadoc。

主题

- 先决条件
- 安装
- AWS Encryption SDK for Java 例子

先决条件

在安装之前 AWS Encryption SDK for Java，请确保满足以下先决条件。

Java 开发环境

您需要使用 Java 8 或更高版本。在 Oracle 网站上，转到 Java SE 下载，然后下载并安装 Java SE Development Kit (JDK)。

如果使用 Oracle JDK，您还必须下载并安装 Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files。

Bouncy Castle

AWS Encryption SDK for Java 需要充气城堡。

- AWS Encryption SDK for Java 1.6.1 及更高版本使用 Bouncy Castle 对加密对象进行序列化和反序列化。您可以使用 Bouncy Castle 或 Bouncy Castle FIPS 以满足该要求。有关安装和配置 Bouncy Castle FIPS 的帮助，请参阅 BC FIPS 文档，尤其是用户指南和安全政策。 PDFs
- 早期版本 AWS Encryption SDK for Java 使用 Bouncy Castle 的 Java 加密 API。仅非 FIPS Bouncy Castle 满足该要求。

如果你没有 Bouncy Castle，请前往下载 Java 版 Bouncy Castle 下载与你的 JDK 对应的提供程序文件。你也可以使用 Apache Mav en 获取标准 Bouncy Castle 提供者（bcprov-ext-jdk15 on）的神器或 Bouncy Castle FIPS 的神器（bc-fips）。

适用于 Java 的 AWS SDK

版本 3。 其中 x AWS Encryption SDK for Java 需要 AWS SDK for Java 2.x，即使你不使用 AWS KMS 钥匙圈。

版本 2。 x 或更早版本 AWS Encryption SDK for Java 不需要 适用于 Java 的 AWS SDK。但是 适用于 Java 的 AWS SDK ，必须使用 AWS Key Management Service(AWS KMS) 作为主密钥提供程序。从 2.4.0 AWS Encryption SDK for Java 版本开始，同时 AWS Encryption SDK for Java 支持 1.x 和 2.x 版本的。 适用于 Java 的 AWS SDK AWS Encryption SDK 适用于 Java 的 AWS SDK 1.x 和 2.x 的代码是可互操作的。例如，您可以使用支持 适用于 Java 的 AWS SDK 1.x 的 AWS Encryption SDK 代码加密数据，然后使用支持的代码对其进行解密 AWS SDK for Java 2.x（反之亦然）。2.4.0 AWS Encryption SDK for Java 之前的版本仅支持 适用于 Java 的 AWS SDK 1.x。有关更新版本的信息 AWS Encryption SDK，请参阅迁移你的 AWS Encryption SDK。

将 AWS Encryption SDK for Java 代码从 1.x 更新到时 AWS SDK for Java 2.x，请将 适用于 Java 的 AWS SDK 1.x 中对`AWSKMS`接口的引用替换为中 适用于 Java 的 AWS SDK 对`KmsClient`接口的引用。 AWS SDK for Java 2.x AWS Encryption SDK for Java 不支持该`KmsAsyncClient`接口。此外，更新代码即可使用 `kmssdkv2` 命名空间中的 AWS KMS相关对象，而不是 `kms` 命名空间。

要安装，请使用 Apache Maven。 适用于 Java 的 AWS SDK

- 要导入整个 适用于 Java 的 AWS SDK以作为依赖项，请在 `pom.xml` 文件中对其进行声明。
- 要在 适用于 Java 的 AWS SDK 1.x 中仅为 AWS KMS 模块创建依赖关系，请按照指定特定模块的说明进行操作，并将设置为。`artifactId` `aws-java-sdk-kms`
- 要在 适用于 Java 的 AWS SDK 2.x 中仅为 AWS KMS 模块创建依赖关系，请按照指定特定模块的说明进行操作。将 `groupId` 设置为 `software.amazon.awssdk`，并将 `artifactId` 设置为 `kms`。

有关更多更改，请参阅《AWS SDK for Java 2.x 开发者指南》中的 适用于 Java 的 AWS SDK 1.x 和 2.x 有什么区别。

《AWS Encryption SDK 开发人员指南》中的 Java 示例使用 AWS SDK for Java 2.x。

安装

安装最新版本的 AWS Encryption SDK for Java。

Note

2.0.0 AWS Encryption SDK for Java 之前的所有版本都处于该阶段。end-of-support

您可以安全地从 AWS Encryption SDK for Java 版本 2.0.x 及更高版本更新为最新版本，无需更改任何代码或数据。但是，版本 2.0.x 中引入了新的安全功能，不向后兼容。要从 1.7.x 之前的版本更新到 2.0.x 及更高版本，必须先更新到 AWS Encryption SDK最新版本 1.x。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

您可以通过以下 AWS Encryption SDK for Java 方式安装。

手动方式

要安装 AWS Encryption SDK for Java，请克隆或下载aws-encryption-sdk-java GitHub存储库。

使用 Apache Maven

可通过 Apache Maven 获得，依赖关系定义如下。 AWS Encryption SDK for Java

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>3.0.0</version>
</dependency>
```

安装完软件开发工具包后，请先查看本指南中的示例 Java 代码，然后打开 J avadoc。 GitHub

AWS Encryption SDK for Java 例子

以下示例向您展示了如何使用 AWS Encryption SDK for Java 来加密和解密数据。这些示例说明了如何使用版本 3。 x 及更高版本的 AWS Encryption SDK for Java. 版本 3。 其中的 x AWS Encryption SDK for Java 需要 AWS SDK for Java 2.x。版本 3。 其中 x AWS Encryption SDK for Java 将主密钥提供程序替换为密钥环。有关使用早期版本的示例，请在上aws-encryption-sdk-java存储库的版本列表中找到您的版本 GitHub。

主题

- 加密和解密字符串
- 加密和解密字节流
- 使用多密钥环加密和解密字节流

加密和解密字符串

以下示例向您展示了如何使用版本 3。 用于加密和解密字符串的 x。 AWS Encryption SDK for Java 在使用字符串之前，请将其转换为字节数组。

此示例使用密AWS KMS 钥环。使用密 AWS KMS 钥环加密时，您可以使用密钥 ID、密钥 ARN、别名或别名 ARN 来识别 KMS 密钥。解密时，必须使用密钥 ARN 来识别 KMS 密钥。

在调用 `encryptData()` 方法时，它返回加密的消息 (`CryptoResult`)，其中包括密文、加密的数据密钥以及加密上下文。在对 `CryptoResult` 对象调用 `getResult` 时，它返回加密的消息的 base-64 编码的字符串版本，您可以将其传递给 `decryptData()` 方法。

同样，当您调用时`decryptData()`，它返回的`CryptoResult`对象包含纯文本消息和一个 AWS KMS key ID。在您的应用程序返回纯文本之前，请验证加密消息中的 AWS KMS key ID 和加密上下文是否符合您的期望。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Collections;
import java.util.Map;

/**
 * Encrypts and then decrypts data using an AWS KMS Keyring.
 *
 * <p>Arguments:
 *
 * <ol>
 *   <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS
 customer master
```

```
 *        key (CMK), see 'Viewing Keys' at
 *        http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html
 * </ol>
 */
public class BasicEncryptionKeyringExample {

  private static final byte[] EXAMPLE_DATA = "Hello
 World".getBytes(StandardCharsets.UTF_8);

  public static void main(final String[] args) {
    final String keyArn = args[0];

    encryptAndDecryptWithKeyring(keyArn);
  }

  public static void encryptAndDecryptWithKeyring(final String keyArn) {
    // 1. Instantiate the SDK
    // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
 commitment policy,
    // which means this client only encrypts using committing algorithm suites and
 enforces
    // that the client will only decrypt encrypted messages that were created with a
 committing
    // algorithm suite.
    // This is the default commitment policy if you build the client with
    // `AwsCrypto.builder().build()`
    // or `AwsCrypto.standard()`.
    final AwsCrypto crypto =
        AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

    // 2. Create the AWS KMS keyring.
    // This example creates a multi keyring, which automatically creates the KMS
 client.
    final MaterialProviders materialProviders =
        MaterialProviders.builder()
            .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
            .build();
    final CreateAwsKmsMultiKeyringInput keyringInput =
        CreateAwsKmsMultiKeyringInput.builder().generator(keyArn).build();
    final IKeyring kmsKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);
```

```
    // 3. Create an encryption context
    // We recommend using an encryption context whenever possible
    // to protect integrity. This sample uses placeholder values.
    // For more information see:
    // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-Integrity-
of-Your-Encrypted-Data-by-Using-AWS-Key-Management
    final Map<String, String> encryptionContext =
        Collections.singletonMap("ExampleContextKey", "ExampleContextValue");

    // 4. Encrypt the data
    final CryptoResult<byte[], ?> encryptResult =
        crypto.encryptData(kmsKeyring, EXAMPLE_DATA, encryptionContext);
    final byte[] ciphertext = encryptResult.getResult();

    // 5. Decrypt the data
    final CryptoResult<byte[], ?> decryptResult =
        crypto.decryptData(
            kmsKeyring,
            ciphertext,
            // Verify that the encryption context in the result contains the
            // encryption context supplied to the encryptData method
            encryptionContext);

    // 6. Verify that the decrypted plaintext matches the original plaintext
    assert Arrays.equals(decryptResult.getResult(), EXAMPLE_DATA);
  }
}
```

加密和解密字节流

以下示例说明如何使用 AWS Encryption SDK 来加密和解密字节流。

此示例使用原始的 AES 密钥环。

加密时，此示例使用 `AwsCrypto.builder() .withEncryptionAlgorithm()` 方法指定不带数字签名的算法套件。解密时，为确保加密文字未签名，此示例使用 `createUnsignedMessageDecryptingStream()` 方法。如果遇到带有数字签名的密文，则该`createUnsignedMessageDecryptingStream()`方法将失败。

如果您使用默认算法套件（包括数字签名）进行加密，请改用 `createDecryptingStream()` 方法，如下一个示例所示。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoAlgorithm;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.AesWrappingAlg;
import software.amazon.cryptography.materialproviders.model.CreateRawAesKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Map;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;


/**
 * <p>
 * Encrypts and then decrypts a file under a random key.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates using a standard Java {@link SecretKey} object as a {@link
 IKeyring} to
 * encrypt and decrypt streaming data.
 */
public class FileStreamingKeyringExample {
    private static String srcFile;
```

```
    public static void main(String[] args) throws IOException {
        srcFile = args[0];

        // In this example, we generate a random key. In practice,
        // you would get a key from an existing store
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Create a Raw Aes Keyring using the random key and an AES-GCM encryption
algorithm
        final MaterialProviders materialProviders = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawAesKeyringInput keyringInput =
CreateRawAesKeyringInput.builder()
                .wrappingKey(ByteBuffer.wrap(cryptoKey.getEncoded()))
                .keyNamespace("Example")
                .keyName("RandomKey")
                .wrappingAlg(AesWrappingAlg.ALG_AES128_GCM_IV12_TAG16)
                .build();
        IKeyring keyring = materialProviders.CreateRawAesKeyring(keyringInput);

        // Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        // This example encrypts with an algorithm suite that doesn't include signing
for faster decryption,
        // since this use case assumes that the contexts that encrypt and decrypt are
equally trusted.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)

.withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
                .build();

        // Create an encryption context to identify the ciphertext
```

```
        Map<String, String> context = Collections.singletonMap("Example",
"FileStreaming");

        // Because the file might be too large to load into memory, we stream the data,
instead of
        //loading it all at once.
        FileInputStream in = new FileInputStream(srcFile);
        CryptoInputStream<JceMasterKey> encryptingStream =
crypto.createEncryptingStream(keyring, in, context);

        FileOutputStream out = new FileOutputStream(srcFile + ".encrypted");
        IOUtils.copy(encryptingStream, out);
        encryptingStream.close();
        out.close();

        // Decrypt the file. Verify the encryption context before returning the
plaintext.
        // Since the data was encrypted using an unsigned algorithm suite, use the
recommended
        // createUnsignedMessageDecryptingStream method, which only accepts unsigned
messages.
        in = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream =
crypto.createUnsignedMessageDecryptingStream(keyring, in);
        // Does it contain the expected encryption context?
        if
(!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Exampl
{
            throw new IllegalStateException("Bad encryption context");
        }

        // Write the plaintext data to disk.
        out = new FileOutputStream(srcFile + ".decrypted");
        IOUtils.copy(decryptingStream, out);
        decryptingStream.close();
        out.close();
    }

    /**
     * In practice, this key would be saved in a secure location.
     * For this demo, we generate a new random key for each operation.
     */
    private static SecretKey retrieveEncryptionKey() {
        SecureRandom rnd = new SecureRandom();
```

```
        byte[] rawKey = new byte[16]; // 128 bits
        rnd.nextBytes(rawKey);
        return new SecretKeySpec(rawKey, "AES");
    }
}
```

使用多密钥环加密和解密字节流

以下示例向您展示了如何将与多密钥 AWS Encryption SDK环一起使用。在使用多重密钥环加密数据时，其任意密钥环中的任意包装密钥都可以对数据进行解密。此示例使用AWS KMS 密钥环和 Raw RSA 密钥环作为子密钥环。

此示例使用包含数字签名的默认算法套件进行加密。直播时，会在完整性检查后但在验证数字签名之前 AWS Encryption SDK 发布纯文本。为了避免在签名验证之前使用明文，此示例会缓冲明文，并且仅在解密和验证完成后才将其写入磁盘。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoOutputStream;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateRawRsaKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
import software.amazon.cryptography.materialproviders.model.PaddingScheme;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Collections;
```

```
/**
 * <p>
 * Encrypts a file using both AWS KMS Key and an asymmetric key pair.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS key,
 *   see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/developerguide/
viewing-keys.html
 *
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 * <p>
 * You might use AWS Key Management Service (AWS KMS) for most encryption and
 decryption operations, but
 * still want the option of decrypting your data offline independently of AWS KMS. This
 sample
 * demonstrates one way to do this.
 * <p>
 * The sample encrypts data under both an AWS KMS key and an "escrowed" RSA key pair
 * so that either key alone can decrypt it. You might commonly use the AWS KMS key for
 decryption. However,
 * at any time, you can use the private RSA key to decrypt the ciphertext independent
 of AWS KMS.
 * <p>
 * This sample uses the RawRsaKeyring to generate a RSA public-private key pair
 * and saves the key pair in memory. In practice, you would store the private key in a
 secure offline
 * location, such as an offline HSM, and distribute the public key to your development
 team.
 */
public class EscrowedEncryptKeyringExample {
    private static ByteBuffer publicEscrowKey;
    private static ByteBuffer privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // This sample generates a new random key for each operation.
        // In practice, you would distribute the public key and save the private key in
 secure
        // storage.
        generateEscrowKeyPair();
```

```
        final String kmsArn = args[0];
        final String fileName = args[1];

        standardEncrypt(kmsArn, fileName);
        standardDecrypt(kmsArn, fileName);

        escrowDecrypt(fileName);
    }

    private static void standardEncrypt(final String kmsArn, final String fileName)
 throws Exception {
        // Encrypt with the KMS key and the escrowed public key
        // 1. Instantiate the SDK
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
 commitment policy,
        // which means this client only encrypts using committing algorithm suites and
 enforces
        // that the client will only decrypt encrypted messages that were created with
 a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
                .build();

        // 2. Create the AWS KMS keyring.
        // This example creates a multi keyring, which automatically creates the KMS
 client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMultiKeyringInput keyringInput =
 CreateAwsKmsMultiKeyringInput.builder()
                .generator(kmsArn)
                .build();
        IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

        // 3. Create the Raw Rsa Keyring with Public Key.
        final CreateRawRsaKeyringInput encryptingKeyringInput =
 CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
```

```
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .build();
        IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

        // 4. Create the multi-keyring.
        final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Encrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encrypted");
        final CryptoOutputStream<?> encryptingStream =
crypto.createEncryptingStream(multiKeyring, out);

        IOUtils.copy(in, encryptingStream);
        in.close();
        encryptingStream.close();
    }

    private static void standardDecrypt(final String kmsArn, final String fileName)
throws Exception {
        // Decrypt with the AWS KMS key and the escrow public key.

        // 1. Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        final AwsCrypto crypto = AwsCrypto.builder()
```

```
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
                .build();

        // 2. Create the AWS KMS keyring.
        // This example creates a multi keyring, which automatically creates the KMS
 client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMultiKeyringInput keyringInput =
 CreateAwsKmsMultiKeyringInput.builder()
                .generator(kmsArn)
                .build();
        IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

        // 3. Create the Raw Rsa Keyring with Public Key.
        final CreateRawRsaKeyringInput encryptingKeyringInput =
 CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .build();
        IKeyring rsaPublicKeyring =
 matProv.CreateRawRsaKeyring(encryptingKeyringInput);

        // 4. Create the multi-keyring.
        final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
 code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        // Since we are using a signing algorithm suite, we avoid streaming decryption
 directly to the output file,
        // to ensure that the trailing signature is verified before writing any
 untrusted plaintext to disk.
```

```
        final ByteArrayOutputStream plaintextBuffer = new ByteArrayOutputStream();
        final CryptoOutputStream<?> decryptingStream =
 crypto.createDecryptingStream(multiKeyring, plaintextBuffer);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
        final ByteArrayInputStream plaintextReader = new
 ByteArrayInputStream(plaintextBuffer.toByteArray());
        IOUtils.copy(plaintextReader, out);
        out.close();
    }

    private static void escrowDecrypt(final String fileName) throws Exception {
        // You can decrypt the stream using only the private key.
        // This method does not call AWS KMS.

        // 1. Instantiate the SDK
        final AwsCrypto crypto = AwsCrypto.standard();

        // 2. Create the Raw Rsa Keyring with Private Key.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawRsaKeyringInput encryptingKeyringInput =
 CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .privateKey(privateEscrowKey)
                .build();
        IKeyring escrowPrivateKeyring =
 matProv.CreateRawRsaKeyring(encryptingKeyringInput);


        // 3. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
 code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
        final CryptoOutputStream<?> decryptingStream =
 crypto.createDecryptingStream(escrowPrivateKeyring, out);
        IOUtils.copy(in, decryptingStream);
```

```
        in.close();
        decryptingStream.close();

    }

    private static void generateEscrowKeyPair() throws GeneralSecurityException {
        final KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA");
        kg.initialize(4096); // Escrow keys should be very strong
        final KeyPair keyPair = kg.generateKeyPair();
        publicEscrowKey = RawRsaKeyringExample.getPEMPublicKey(keyPair.getPublic());
        privateEscrowKey = RawRsaKeyringExample.getPEMPrivateKey(keyPair.getPrivate());

    }
}
```

AWS Encryption SDK for JavaScript

旨在 AWS Encryption SDK for JavaScript 为在 Node.js 中编写 Web 浏览器应用程序 JavaScript 或使用编写 Web 服务器应用程序的开发人员提供客户端加密库。

与的所有实现一样 AWS Encryption SDK， AWS Encryption SDK for JavaScript 提供了高级数据保护功能。这些功能包括信封加密、其他经过身份验证的数据 (AAD) 以及安全、经过身份验证且对称的密钥算法套件，如具有密钥派生和签名的 256 位 AES-GCM。

的所有特定于语言的实现 AWS Encryption SDK 都被设计为可互操作，但要遵守语言的限制。有关语言限制的详细信息 JavaScript，请参阅the section called “兼容性”。

了解更多

- 有关使用编程的详细信息 AWS Encryption SDK for JavaScript，请参阅上的aws-encryption-sdk-javascript存储库 GitHub。
- 有关编程示例，请参阅the section called “示例”以及存储库中的示例浏览器和示例节点模块。aws-encryption-sdk-javascript
- 有关使用在 Web 应用程序中加密数据的真实示例，请参阅 AWS 安全博客中的如何使用 AWS Encryption SDK for JavaScript 和 Node.js 在浏览器中启用加密。 AWS Encryption SDK for JavaScript

主题

- 的兼容性 AWS Encryption SDK for JavaScript

- 正在安装 AWS Encryption SDK for JavaScript
- 中的模块 AWS Encryption SDK for JavaScript
- AWS Encryption SDK for JavaScript 例子

的兼容性 AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript 旨在与的其他语言实现实现互操作。 AWS Encryption SDK在大多数情况下，您可以使用加密数据，也可以使用任何其他语言实现（包括AWS Encryption SDK 命令行界面）对其进行解密。 AWS Encryption SDK for JavaScript 而且，您可以使用 AWS Encryption SDK for JavaScript 来解密由的其他语言实现生成的加密消息。 AWS Encryption SDK

但是，在使用时 AWS Encryption SDK for JavaScript，您需要注意 JavaScript 语言实现和 Web 浏览器中的一些兼容性问题。

此外，在使用不同的语言实施时，请务必配置兼容的主密钥提供程序、主密钥和密钥环。有关更多信息，请参阅 密钥环兼容性。

AWS Encryption SDK for JavaScript 兼容性

的 JavaScript 实现与其他语言实现的 AWS Encryption SDK 不同之处在于：

- 的加密操作 AWS Encryption SDK for JavaScript 不会返回非成帧的密文。但是， AWS Encryption SDK for JavaScript 将解密其他语言实现返回的带框和非成帧的密文。 AWS Encryption SDK
- 从 Node.js 12.9.0 版开始，Node.js 支持以下 RSA 密钥包装选项：
  - OAEP 带有 SHA1、 SHA256、 SHA384、或 SHA512
  - OAEP 有 SHA1 和 MGF1 带有 SHA1
  - PKCS1v15
- 在 12.9.0 版之前，Node.js 仅支持以下 RSA 密钥包装选项：
  - OAEP 有 SHA1 和 MGF1 带有 SHA1
  - PKCS1v15

浏览器兼容性

某些 Web 浏览器不支持 AWS Encryption SDK for JavaScript 所需的基本加密操作。您可以通过为浏览器实现的 WebCrypto API 配置备用来弥补一些缺失的操作。

Web 浏览器限制

以下限制是所有 Web 浏览器通用的：

- WebCrypto API 不支持 PKCS1v15 密钥封装。
- 浏览器不支持 192 位密钥。

所需的加密操作

AWS Encryption SDK for JavaScript 需要在 Web 浏览器中执行以下操作。如果浏览器不支持这些操作，则它与 AWS Encryption SDK for JavaScript不兼容。

- 浏览器必须包含 `crypto.getRandomValues()`，这是一种生成加密随机值的方法。有关支持的 Web 浏览器版本的信息`crypto.getRandomValues()`，请参阅我能否使用加密货币。getRandomValues()？。

所需的回退

AWS Encryption SDK for JavaScript 需要在 Web 浏览器中使用以下库和操作。如果您支持的 Web 浏览器不满足这些要求，您必须配置回退。否则，尝试在浏览器中 AWS Encryption SDK for JavaScript 使用将失败。

- 该 WebCrypto API在Web应用程序中执行基本的加密操作，但并非适用于所有浏览器。有关支持 Web 加密的 Web 浏览器版本的信息，请参阅我是否可以使用 Web 加密？。
- 现代版本的 Safari 网络浏览器不支持 AES-GCM 零字节加密，这是必需的。 AWS Encryption SDK 如果浏览器实现了 WebCrypto API，但无法使用 AES-GCM 加密零字节，则仅 AWS Encryption SDK for JavaScript 使用备用库进行零字节加密。它使用 WebCrypto API 进行所有其他操作。

要为这两种限制配置回退，请将以下语句添加到代码中。在 configureFallback 函数中，指定一个支持缺少的功能的库。以下示例使用 Microsoft JavaScript Research 密码学库 (`msrcrypto`)，但你可以将其替换为兼容的库。有关完整的示例，请参阅 fallback.ts。

```
import { configureFallback } from '@aws-crypto/client-browser'
configureFallback(msrCrypto)
```

正在安装 AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript 由一系列相互依存的模块组成。一些模块只是设计为一起工作的模块的集合。一些模块设计为单独工作。一些模块是所有实施所必需的；而一些其他模块仅

在特殊情况下是必需的。有关 for 中模块的信息 JavaScript， AWS Encryption SDK 请参阅，中的模块 AWS Encryption SDK for JavaScript以及aws-encryption-sdk-javascript存储库中每个模块中的`README.md`文件 GitHub。

> **Note**
>
> 2.0.0 AWS Encryption SDK for JavaScript 之前的所有版本都处于该阶段。end-of-support
>
> 您可以安全地从 AWS Encryption SDK for JavaScript 版本 2.0.x 及更高版本更新为最新版本，无需更改任何代码或数据。但是，版本 2.0.x 中引入了新的安全功能，不向后兼容。要从 1.7.x 之前的版本更新到 2.0.x 及更高版本，必须先更新到 AWS Encryption SDK for JavaScript最新版本 1.x。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

要安装这些模块，请使用 npm package manager。

例如，要安装包含在 `client-node` Node.js AWS Encryption SDK for JavaScript 中使用编程所需的所有模块的模块，请使用以下命令。

```
npm install @aws-crypto/client-node
```

要安装该`client-browser`模块（包括需要在浏览器 AWS Encryption SDK for JavaScript 中使用编程的所有模块），请使用以下命令。

```
npm install @aws-crypto/client-browser
```

有关如何使用的工作示例 AWS Encryption SDK for JavaScript，请参阅aws-encryption-sdk-javascript存储库中的`example-node`和`example-browser`模块中的示例 GitHub。

中的模块 AWS Encryption SDK for JavaScript

中的模块可以 AWS Encryption SDK for JavaScript 轻松安装项目所需的代码。

JavaScript Node.js 的模块

client-node

包括在 Node.js 中使用编程所需的所有模块。 AWS Encryption SDK for JavaScript

caching-materials-manager-node

在 Node.js 中导出支持数据密钥缓存功能 AWS Encryption SDK for JavaScript 的函数。

[decrypt-node](#)

导出解密和验证表示数据和数据流的加密消息的函数。包含在 `client-node` 模块中。

[encrypt-node](#)

导出对不同类型的数据进行加密和签名的函数。包含在 `client-node` 模块中。

[example-node](#)

AWS Encryption SDK for JavaScript 在 Node.js 中导出使用编程的工作示例。包括不同类型的密钥环和不同类型的数据的示例。

[hkdf-node](#)

导出基于 HMAC 的密钥派生函数 (HKDF)，Node.js AWS Encryption SDK for JavaScript 中的在特定算法套件中使用该函数。浏览器 AWS Encryption SDK for JavaScript 中使用 WebCrypto API 中的原生 HKDF 函数。

[integration-node](#)

定义测试，以验证 Node.js AWS Encryption SDK for JavaScript 中的是否与的其他语言实现兼容 AWS Encryption SDK。

[kms-keyring-node](#)

在 Node.js 中导出支持 AWS KMS 密钥环的函数。

[raw-aes-keyring-node](#)

导出在 Node.js 中支持原始 AES 密钥环的函数。

[raw-rsa-keyring-node](#)

导出在 Node.js 中支持原始 RSA 密钥环的函数。

JavaScript 浏览器模块

[client-browser](#)

包括你需要在浏览器 AWS Encryption SDK for JavaScript 中使用编程的所有模块。

[caching-materials-manager-browser](#)

在浏览器中导出支持数据密钥缓存功能 JavaScript 的函数。

[decrypt-browser](#)

导出解密和验证表示数据和数据流的加密消息的函数。

encrypt-browser

导出对不同类型的数据进行加密和签名的函数。

example-browser

在浏览器 AWS Encryption SDK for JavaScript 中使用编程的工作示例。包括不同类型的密钥环和不同类型的数据的示例。

integration-browser

定义用于验证浏览器中的 AWS Encryption SDK for Java脚本是否与的其他语言实现兼容的测试 AWS Encryption SDK。

kms-keyring-browser

导出在浏览器中支持 AWS KMS 密钥环的函数。

raw-aes-keyring-browser

导出在浏览器中支持原始 AES 密钥环的函数。

raw-rsa-keyring-browser

导出在浏览器中支持原始 RSA 密钥环的函数。

适用于所有实施的模块

cache-material

支持数据密钥缓存功能。提供代码以组装与每个数据密钥一起缓存的加密材料。

kms-keyring

导出支持 KMS 密钥环的函数。

material-management

实施加密材料管理器 (CMM)。

raw-keyring

导出原始 AES 和 RSA 密钥环所需的函数。

serialize

导出该开发工具包用于序列化其输出的函数。

web-crypto-backend

在浏览器中导出使用该 WebCrypto API AWS Encryption SDK for JavaScript 的函数。

AWS Encryption SDK for JavaScript 例子

以下示例演示了如何使用 AWS Encryption SDK for JavaScript 加密和解密数据。

你可以在上的 e xample-node 和 example- browser 模块中找到更多使用示例。 AWS Encryption SDK for JavaScript aws-encryption-sdk-javascript GitHub在安装 `client-browser` 或 `client-node` 模块时，不会安装这些示例模块。

请参阅完整的代码示例：节点：kms_simple.ts，浏览器：kms_simple.ts

主题

- 使用密钥环加密 AWS KMS 数据
- 使用密钥环解密数据 AWS KMS

使用密钥环加密 AWS KMS 数据

以下示例说明如何使用 AWS Encryption SDK for JavaScript 来加密和解密短字符串或字节数组。

此示例以AWS KMS 密钥环为特色，这是一种使用生成和加密数据密钥 AWS KMS key 的密钥环。有关创建的帮助 AWS KMS key，请参阅《AWS Key Management Service 开发者指南》中的创建密钥。如需帮助识别 AWS KMS 钥匙圈 AWS KMS keys 中的内容，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别

步骤 1：设置承诺政策。

从 1.7 版本开始。 x 中 AWS Encryption SDK for JavaScript，您可以在调用实例化客户端的新`buildClient`函数时设置承诺策略。 AWS Encryption SDK `buildClient` 函数需要表示承诺策略的枚举值。进行加密和解密时，该函数会返回强制执行您的承诺策略的已更新 `encrypt` 和 `decrypt` 函数。

以下示例使用`buildClient`函数来指定默认的承诺策略`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。您也可以使用`buildClient`来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

JavaScript Browser

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

JavaScript Node.js

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

步骤 2：构造密钥环。

创建用于加密的 AWS KMS 密钥环。

使用密 AWS KMS 钥环加密时，必须指定生成器密钥，即用于生成纯文本数据密钥并对其进行加密的生成器密钥。 AWS KMS key 您还可以指定零个或更多附加密钥，以加密相同的明文数据密钥。密钥环返回密钥环中每个 AWS KMS key 密钥的纯文本数据密钥和该数据密钥的一个加密副本，包括生成器密钥。要解密数据，您需要解密任一加密数据密钥。

要在中 AWS KMS keys 为加密密钥环指定 AWS Encryption SDK for JavaScript，您可以使用任何支持的 AWS KMS 密钥标识符。该示例使用一个生成器密钥（由其别名 ARN 标识）和一个附加密钥（由密钥 ARN 标识）。

Note

如果您打算重复使用密 AWS KMS 钥环进行解密，则必须使用密钥 ARNs 来识别密钥环 AWS KMS keys 中的密钥。

在运行此代码之前，请将示例标 AWS KMS key 识符替换为有效的标识符。您必须具有在密钥环中使用 AWS KMS keys所需的权限。

JavaScript Browser

首先，向浏览器提供您的凭证。这些 AWS Encryption SDK for JavaScript 示例使用 webpack。DefinePlugin，它会将凭证常量替换为您的实际证书。但是，您可以使用任何方法以提供凭证。然后，使用凭据创建 AWS KMS 客户端。

```
declare const credentials: {accessKeyId: string, secretAccessKey:string,
 sessionToken:string }

const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

接下来， AWS KMS keys 为生成器密钥和其他密钥指定。然后，使用 AWS KMS 客户端和创建 AWS KMS 密钥环。 AWS KMS keys

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringBrowser({ clientProvider, generatorKeyId, keyIds })
```

JavaScript Node.js

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
```

```
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })
```

步骤 3：设置加密上下文。

加密上下文是任意的非机密其他经过身份验证的数据。当您提供有关加密的加密上下文时，会 AWS Encryption SDK 以加密方式将加密上下文绑定到密文，因此解密数据需要相同的加密上下文。使用加密上下文是可选的，但作为一项最佳实践，建议您提供加密上下文。

创建一个包含加密上下文对的简单对象。每对中的键和值必须是一个字符串。

JavaScript Browser

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

JavaScript Node.js

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

步骤 4：加密数据。

要加密明文数据，请调用 `encrypt` 函数。传入 AWS KMS 密钥环、纯文本数据和加密上下文。

`encrypt` 函数返回加密的消息 (`result`)，其中包含加密的数据、加密的数据密钥和重要元数据，包括加密上下文和签名。

您可以对任何支持的编程语言使用来解密此加密消息。 AWS Encryption SDK

JavaScript Browser

```
const plaintext = new Uint8Array([1, 2, 3, 4, 5])

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

JavaScript Node.js

```
const plaintext = 'asdf'

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

使用密钥环解密数据 AWS KMS

您可以使用 AWS Encryption SDK for JavaScript 来解密加密的邮件并恢复原始数据。

在该示例中，我们解密在the section called “使用密钥环加密 AWS KMS 数据”示例中加密的数据。

步骤 1：设置承诺政策。

从 1.7 版本开始。 x 中 AWS Encryption SDK for JavaScript，您可以在调用实例化客户端的新buildClient函数时设置承诺策略。 AWS Encryption SDK buildClient 函数需要表示承诺策略的枚举值。进行加密和解密时，该函数会返回强制执行您的承诺策略的已更新 encrypt 和 decrypt 函数。

以下示例使用buildClient函数来指定默认的承诺策略REQUIRE_ENCRYPT_REQUIRE_DECRYPT。您也可以使用buildClient来限制加密消息中加密数据密钥的数量。有关更多信息，请参阅 the section called “限制加密数据密钥”。

JavaScript Browser

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

JavaScript Node.js

```
import {
```

```
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-node'

const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)
```

步骤 2：构造密钥环。

要解密数据，请传入 encrypt 函数返回的加密消息 (result)。加密的消息包括加密的数据、加密的数据密钥和重要元数据，包括加密上下文和签名。

在解密时，您还必须指定一个 AWS KMS 密钥环。您可以使用用于加密数据的相同密钥环，或者使用不同的密钥环。要成功，解密密钥环 AWS KMS key 中必须至少有一个能够解密加密消息中的一个加密数据密钥。由于没有生成任何数据密钥，您不需要在解密密钥环中指定生成器密钥。如果这样做，将按相同的方式处理生成器密钥和附加密钥。

要在中 AWS KMS key 为解密密钥环指定一个 AWS Encryption SDK for JavaScript，必须使用密钥 ARN。否则 AWS KMS key ，将无法识别。如需帮助识别 AWS KMS 钥匙圈 AWS KMS keys 中的内容，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别

> **Note**
>
> 如果您使用相同的密钥环进行加密和解密，请使用密钥 ARNs 来识别密钥环中的 AWS KMS keys 密钥。

在此示例中，我们创建了一个仅包含加密密钥环 AWS KMS keys 中的一个的密钥环。在运行该代码之前，请将示例密钥 ARN 替换为有效 ARN。您必须具有 AWS KMS key的 kms:Decrypt 权限。

JavaScript Browser

首先，向浏览器提供您的凭证。这些 AWS Encryption SDK for JavaScript 示例使用 webpack。DefinePlugin，它会将凭证常量替换为您的实际证书。但是，您可以使用任何方法以提供凭证。然后，使用凭据创建 AWS KMS 客户端。

```
declare const credentials: {accessKeyId: string, secretAccessKey:string,
 sessionToken:string }
```

```
const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

接下来，使用 AWS KMS 客户端创建 AWS KMS 密钥环。此示例仅使用加密密钥环 AWS KMS keys 中的一个。

```
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringBrowser({ clientProvider, keyIds })
```

JavaScript Node.js

```
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringNode({ keyIds })
```

步骤 3：解密数据。

接下来，调用 `decrypt` 函数。传入您刚刚创建的解密密钥环 (`keyring`) 和 `encrypt` 函数返回的加密消息 (`result`)。 AWS Encryption SDK 使用密钥环解密其中一个加密的数据密钥。然后，它使用明文数据密钥以解密数据。

如果调用成功，则 `plaintext` 字段包含明文（已解密）数据。`messageHeader` 字段包含有关解密过程的元数据，包括用于解密数据的加密上下文。

JavaScript Browser

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

JavaScript Node.js

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

步骤 4：验证加密上下文。

用于解密数据的加密上下文包含在 `decrypt` 函数返回的消息标头 (`messageHeader`) 中。在应用程序返回明文数据之前，请验证在解密时使用的加密上下文中是否包含在加密时提供的加密上下文。如果不匹配，则可能表明数据被篡改，或者您没有解密正确的密文。

在验证加密上下文时，不需要完全匹配。在将加密算法与签名一起使用时，加密材料管理器 (CMM) 在加密消息之前将公有签名密钥添加到加密上下文中。但是，您提交的所有加密上下文对应包含在返回的加密上下文中。

首先，从消息标头中获取加密上下文。然后，验证原始加密上下文 (`context`) 中的每个键值对与返回的加密上下文 (`encryptionContext`) 中的键值对是否匹配。

JavaScript Browser

```
const { encryptionContext } = messageHeader

Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

JavaScript Node.js

```
const { encryptionContext } = messageHeader

Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

如果加密上下文检查成功，您可以返回明文数据。

AWS Encryption SDK for Python

本主题介绍了如何安装和使用 AWS Encryption SDK for Python。有关使用编程的详细信息 AWS Encryption SDK for Python，请参阅上的aws-encryption-sdk-python存储库 GitHub。有关 API 文档，请参阅阅读文档。

主题

- 先决条件
- 安装
- AWS Encryption SDK for Python 示例代码

先决条件

在安装之前 AWS Encryption SDK for Python，请确保满足以下先决条件。

支持的 Python 版本

3.2.0 及更高版本需要 Python 3.8 或更高 AWS Encryption SDK for Python 版本。

AWS 加密材料提供程序库 (MPL) 是版本 4 中 AWS Encryption SDK for Python 引入的可选依赖项。 x。如果你打算安装 MPL，则必须使用 Python 3.11 或更高版本。

的早期版本 AWS Encryption SDK 支持 Python 2.7 和 Python 3.4 及更高版本，但我们建议您使用最新版本的 AWS Encryption SDK。

要下载 Python，请参阅 Python 下载。

适用于 Python 的 pip 安装工具

`pip` 包含在 Python 3.6 及更高版本中，但您可能需要对其进行升级。有关升级或安装 `pip` 的更多信息，请参阅 `pip` 文档中的 Installation。

安装

安装最新版本的 AWS Encryption SDK for Python。

Note

3.0.0 AWS Encryption SDK for Python 之前的所有版本都处于该阶段。end-of-support
您可以安全地从 AWS Encryption SDK 版本 2.0.x 及更高版本更新为最新版本，无需更改任何代码或数据。但是，版本 2.0.x 中引入了新的安全功能，不向后兼容。要从 1.7.x 之前的版本更新到 2.0.x 及更高版本，必须先更新到 AWS Encryption SDK最新版本 1.x。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

`pip`用于安装 AWS Encryption SDK for Python，如以下示例所示。

安装最新版本

```
pip install "aws-encryption-sdk[MPL]"
```

后`[MPL]`缀安装AWS 加密材料提供程序库 (MPL)。MPL 包含用于加密和解密数据的结构。MPL 是版本 4 中 AWS Encryption SDK for Python 引入的可选依赖项。 x。我们强烈建议安装 MPL。但是，如果您不打算使用 MPL，则可以省略后`[MPL]`缀。

有关使用 pip 安装和升级程序包的更详细信息，请参阅安装程序包。

AWS Encryption SDK for Python 需要所有平台上的密码学库（pyca/密码学）。`pip` 所有版本均会在 Windows 上自动安装和构建 `cryptography` 库。 `pip` 8.1 及更高版本会自动在 Linux 上安装和构建 `cryptography`。如果使用 `pip` 早期版本，并且 Linux 环境没有构建 `cryptography` 库所需的工具，您需要安装这些工具。有关更多信息，请参阅在 Linux 上构建加密。

该版本的1.10.0和2.5.0版本将密码学依赖关系 AWS Encryption SDK for Python 固定在2.5.0和3.3.2之间。其他版本则 AWS Encryption SDK for Python 安装最新版本的密码学。如果您需要比 3.3.2 更高的密码系统版本，我们建议您使用 AWS Encryption SDK for Python的最新主要版本。

有关的最新开发版本 AWS Encryption SDK for Python，请访问中的aws-encryption-sdk-python存储库 GitHub。

安装完成后 AWS Encryption SDK for Python，请先查看本指南中的 Python 示例代码。

AWS Encryption SDK for Python 示例代码

以下示例向您展示了如何使用 AWS Encryption SDK for Python 来加密和解密数据。

本节中的示例说明如何使用版本 4。 其中的 x AWS Encryption SDK for Python 具有可选的加密材料提供程序库依赖项 (`aws-cryptographic-material-providers`)。要查看使用早期版本的示例，或者不使用材料提供者库 (MPL) 的安装，请在aws-encryption-sdk-python存储库的版本列表中找到您的版本。 GitHub

当你使用版本 4 时。 x 在 AWS Encryption SDK for Python MPL 中，它使用密钥环来执行信封加密。AWS Encryption SDK 提供的密钥环与您在先前版本中使用的主密钥提供程序兼容。有关更多信息，请参阅 the section called “密钥环兼容性”。有关从主密钥提供程序迁移到密钥环的示例，请参阅`aws-encryption-sdk-python`存储库中的迁移示例： GitHub

主题

- 加密和解密字符串
- 加密和解密字节流

加密和解密字符串

以下示例说明如何使用 AWS Encryption SDK 来加密和解密字符串。此示例使用带有对称加密 KMS AWS KMS 密钥的密钥环。

此示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`有关更多信息，请参阅 the section called “设置您的承诺策略”。

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example sets up the KMS Keyring

The AWS KMS keyring uses symmetric encryption KMS keys to generate, encrypt and
decrypt data keys. This example creates a KMS Keyring and then encrypts a custom input
 EXAMPLE_DATA
with an encryption context. This example also includes some sanity checks for
 demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

AWS KMS keyrings can be used independently or in a multi-keyring with other keyrings
```

```
of the same or a different type.

"""

import boto3
from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import CreateAwsKmsKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

EXAMPLE_DATA: bytes = b"Hello World"


def encrypt_and_decrypt_with_keyring(
    kms_key_id: str
):
    """Demonstrate an encrypt/decrypt cycle using an AWS KMS keyring.

    Usage: encrypt_and_decrypt_with_keyring(kms_key_id)
    :param kms_key_id: KMS Key identifier for the KMS key you want to use for
 encryption and
    decryption of your data keys.
    :type kms_key_id: string

    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
    # which enforces that this client only encrypts using committing algorithm suites
 and enforces
    # that this client will only decrypt encrypted messages that were created with a
 committing
    # algorithm suite.
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. Create a boto3 client for KMS.
```

```
    kms_client = boto3.client('kms', region_name="us-west-2")

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Create your keyring
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
        kms_key_id=kms_key_id,
        kms_client=kms_client
    )

    kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
        input=keyring_input
    )

    # 5. Encrypt the data with the encryptionContext.
    ciphertext, _ = client.encrypt(
        source=EXAMPLE_DATA,
        keyring=kms_keyring,
        encryption_context=encryption_context
    )

    # 6. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
 code.)
    assert ciphertext != EXAMPLE_DATA, \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 7. Decrypt your encrypted data using the same keyring you used on encrypt.
    plaintext_bytes, _ = client.decrypt(
        source=ciphertext,
        keyring=kms_keyring,
        # Provide the encryption context that was supplied to the encrypt method
```

```
        encryption_context=encryption_context,
    )

    # 8. Demonstrate that the decrypted plaintext is identical to the original
 plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
 code.)
    assert plaintext_bytes == EXAMPLE_DATA, \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
 decryption"
```

加密和解密字节流

以下示例说明如何使用 AWS Encryption SDK 来加密和解密字节流。此示例使用原始的 AES 密钥环。

此示例使用默认承诺策略实例化 AWS Encryption SDK 客户端。REQUIRE_ENCRYPT_REQUIRE_DECRYPT有关更多信息，请参阅 the section called “设置您的承诺策略”。

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example demonstrates file streaming for encryption and decryption.

File streaming is useful when the plaintext or ciphertext file/data is too large to
 load into
memory. Therefore, the AWS Encryption SDK allows users to stream the data, instead of
 loading it
all at once in memory. In this example, we demonstrate file streaming for encryption
 and decryption
using a Raw AES keyring. However, you can use any keyring with streaming.

This example creates a Raw AES Keyring and then encrypts an input stream from the file
`plaintext_filename` with an encryption context to an output (encrypted) file
 `ciphertext_filename`.
It then decrypts the ciphertext from `ciphertext_filename` to a new file
 `decrypted_filename`.
This example also includes some sanity checks for demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.
```

```
See raw_aes_keyring_example.py in the same directory for another raw AES keyring
 example
in the AWS Encryption SDK for Python.
"""
import filecmp
import secrets

from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import AesWrappingAlg,
 CreateRawAesKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_and_decrypt_with_keyring(
    plaintext_filename: str,
    ciphertext_filename: str,
    decrypted_filename: str
):
    """Demonstrate a streaming encrypt/decrypt cycle.

    Usage: encrypt_and_decrypt_with_keyring(plaintext_filename
                                            ciphertext_filename
                                            decrypted_filename)
    :param plaintext_filename: filename of the plaintext data
    :type plaintext_filename: string
    :param ciphertext_filename: filename of the ciphertext data
    :type ciphertext_filename: string
    :param decrypted_filename: filename of the decrypted data
    :type decrypted_filename: string
    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
    # which enforces that this client only encrypts using committing algorithm suites
 and enforces
    # that this client will only decrypt encrypted messages that were created with a
 committing
    # algorithm suite.
```

```
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. The key namespace and key name are defined by you.
    # and are used by the Raw AES keyring to determine
    # whether it should attempt to decrypt an encrypted data key.
    key_name_space = "Some managed raw keys"
    key_name = "My 256-bit AES wrapping key"

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Generate a 256-bit AES key to use with your keyring.
    # In practice, you should get this key from a secure key management system such as
 an HSM.

    # Here, the input to secrets.token_bytes() = 32 bytes = 256 bits
    static_key = secrets.token_bytes(32)

    # 5. Create a Raw AES keyring
    # We choose to use a raw AES keyring, but any keyring can be used with streaming.
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
        key_namespace=key_name_space,
        key_name=key_name,
        wrapping_key=static_key,
        wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
    )

    raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
        input=keyring_input
```

```
    )

    # 6. Encrypt the data stream with the encryptionContext
    with open(plaintext_filename, 'rb') as pt_file, open(ciphertext_filename, 'wb') as
ct_file:
        with client.stream(
            mode='e',
            source=pt_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as encryptor:
            for chunk in encryptor:
                ct_file.write(chunk)

    # 7. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert not filecmp.cmp(plaintext_filename, ciphertext_filename), \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 8. Decrypt your encrypted data stream using the same keyring you used on
encrypt.
    with open(ciphertext_filename, 'rb') as ct_file, open(decrypted_filename, 'wb') as
pt_file:
        with client.stream(
            mode='d',
            source=ct_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as decryptor:
            for chunk in decryptor:
                pt_file.write(chunk)

    # 10. Demonstrate that the decrypted plaintext is identical to the original
plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert filecmp.cmp(plaintext_filename, decrypted_filename), \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
decryption"
```

AWS Encryption SDK 对于 Rust

本主题介绍如何安装和使用 for AWS Encryption SDK Rust。有关使用 for Rust AWS Encryption SDK 进行编程的详细信息，请参阅上 aws-encryption-sdk存储库的 Rust 目录 GitHub。

f AWS Encryption SDK or Rust 与其他一些编程语言实现的不同之处 AWS Encryption SDK 在于：

- 不支持数据密钥缓存。但是，for AWS Encryption SDK Rust 支持AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。
- 不支持流数据

f AWS Encryption SDK or Rust 包含 2.0 版本中引入的所有安全功能。 x 及更高版本的其他语言实现 AWS Encryption SDK。但是，如果你使用 for Rust 来解密由 2.0 之前版本加密的数据。 AWS Encryption SDK x 版本的另一种语言实现 AWS Encryption SDK，您可能需要调整承诺政策。有关更多信息，请参阅 如何设置您的承诺策略。

f AWS Encryption SDK or Rust 是 AWS Encryption SDK in Dafny 的产物，这是一种正式的验证语言，你可以用它来编写规范、实现规范的代码以及测试规范。结果为在确保功能正确性的框架中实施 AWS Encryption SDK 功能的库。

了解更多

- 有关显示如何在中配置选项（例如指定备用算法套件 AWS Encryption SDK、限制加密数据密钥和使用 AWS KMS 多区域密钥）的示例，请参阅正在配置 AWS Encryption SDK。
- 有关如何配置和使用 for Rust AWS Encryption SDK 的示例，请参阅上 aws-encryption-sdk存储库中的 Rust 示例 GitHub。

主题

- 先决条件
- 安装
- AWS Encryption SDK 对于 Rust 的示例代码

先决条件

在安装 f AWS Encryption SDK or Rust 之前，请确保满足以下先决条件。

安装 Rust 和 Cargo

使用 r ustup 安装当前稳定版本的 Rust。

有关下载和安装 rustup 的更多信息，请参阅《货运手册》中的安装程序。

安装

f AWS Encryption SDK or Rust 在 aws-esdkCrates.io 上可以作为箱子使用。有关安装和构建 Rust 版 AWS Encryption SDK 的详细信息，请参阅存储库中的 README.md。 aws-encryption-sdk GitHub

你可以通过以下方式安装 AWS Encryption SDK 适用于 Rust 的。

手动方式

要安装 Rust 版，请克隆或下载aws-encryption-sdk GitHub 存储库。 AWS Encryption SDK

使用 Crates.io

在您的项目目录中运行以下 Cargo 命令：

```
cargo add aws-esdk
```

或者在你的 Cargo.toml 中添加以下一行：

```
aws-esdk = "<version>"
```

AWS Encryption SDK 对于 Rust 的示例代码

以下示例显示了使用 for Rust AWS Encryption SDK 进行编程时使用的基本编码模式。具体而言，您可以实例化材料提供者库 AWS Encryption SDK 和材料提供者库。然后，在调用每个方法之前，先实例化定义该方法输入的对象。

有关展示如何在中配置选项（例如指定备用算法套件和限制加密数据密钥）的示例，请参阅 aws-encryption-sdk存储库中的 Rust 示例 GitHub。 AWS Encryption SDK

在 for Rust 中加密和解密数据 AWS Encryption SDK

此示例显示了加密和解密数据的基本模式。它使用受一个 AWS KMS 包装密钥保护的数据密钥对一个小文件进行加密。

步骤 1：实例化. AWS Encryption SDK

您将使用中的方法 AWS Encryption SDK 来加密和解密数据。

```
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;
```

步骤 2：创建 AWS KMS 客户端。

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);
```

可选：创建您的加密上下文。

```
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);
```

第 3 步：实例化材料提供者库。

您将使用材料提供程序库中的方法创建密钥环，密钥环指定哪些密钥保护您的数据。

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

步骤 4：创建 AWS KMS 密钥环。

要创建密钥环，请使用密钥环输入对象调用密钥环方法。此示例使用`create_aws_kms_keyring()`方法并指定一个 KMS 密钥。

```
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;
```

第 5 步：加密明文。

```
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

let ciphertext = encryption_response
                .ciphertext
                .expect("Unable to unwrap ciphertext from encryption response");
```

第 6 步：使用与加密时相同的密钥环解密您的加密数据。

```
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;

let decrypted_plaintext = decryption_response
                        .plaintext
                        .expect("Unable to unwrap plaintext from decryption
 response");
```

AWS Encryption SDK 命令行界面

AWS Encryption SDK 命令行界面（AWS 加密 CLI）使您能够使用在命令行和脚本中 AWS Encryption SDK 以交互方式加密和解密数据。您不需要具有加密或编程专业知识。

> **Note**
>
> 4.0.0 之前的 AWS 加密 CLI 版本处于该阶段。end-of-support
> 您无需更改任何代码或数据即可安全地从 AWS Encryption CLI 版本 2.1.x 及更高版本更新为最新版本。但是，版本 2.1.x 中引入了新的安全功能，不向后兼容。从 1.7 版本更新。 x 或更

早版本，必须先更新到最新的 1。 AWS 加密 CLI 的 x 版本。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。
新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

与的所有实现一样 AWS Encryption SDK， AWS 加密 CLI 提供高级数据保护功能。这些功能包括信封加密、额外验证数据（AAD）以及经过身份验证的安全对称密钥算法套件，例如，具有密钥派生、密钥承诺和签名的 256 位 AES-GCM。

AWS 加密 CLI 建立在 Linux、macOS AWS Encryption SDK for Python和 Windows 上，并受其支持。你可以在 Linux 或 macOS 的首选 shell 中、Windows 的命令提示符窗口 (cmd.exe) 以及任何系统的 PowerShell 控制台中运行命令和脚本来加密和解密数据。

的所有特定于语言的实现 AWS Encryption SDK，包括加密 AWS CLI，均可互操作。例如，您可以使用加密数据，使用加密 AWS CLI 对其进行解密。AWS Encryption SDK for Java

本主题介绍了 AWS 加密 CLI，解释了如何安装和使用它，并提供了几个示例来帮助您入门。要快速入门，请参阅 AWS 安全博客中的如何使用加密 AWS CLI 加密和解密您的数据。有关更多详细信息，请参阅阅读文档，并加入我们在aws-encryption-sdk-cli存储库中开发 AWS Encryption CLI GitHub。

性能

加 AWS 密 CLI 建立在 AWS Encryption SDK for Python。每次运行 CLI 时，都会启动新的 Python 运行时实例。为了提高性能，请尽可能使用单个命令而不是一系列单独的命令。例如，运行一个递归处理目录中的文件的命令，而不是为每个文件运行单独的命令。

主题

- 安装 AWS Encryption SDK 命令行界面
- 如何使用 AWS 加密 CLI
- AWS 加密 CLI 的示例
- AWS Encryption SDK CLI 语法和参数参考
- AWS 加密 CLI 的版本

安装 AWS Encryption SDK 命令行界面

本主题介绍如何安装 AWS 加密 CLI。有关详细信息，请参阅上的aws-encryption-sdk-cli存储库 GitHub 和 Read the Docs。

主题

- 安装必备组件
- 安装和更新 AWS 加密 CLI

安装必备组件

加 AWS 密 CLI 建立在 AWS Encryption SDK for Python。要安装 AWS 加密 CLI，你需要 Python 和 `pip` Python 包管理工具。可以在所有支持的平台上使用 Python 和 `pip`。

在安装 AWS 加密 CLI 之前，请安装以下必备组件，

Python

AWS 加密 CLI 版本 4.2.0 及更高版本需要 Python 3.8 或更高版本。

早期版本的 AWS 加密 CLI 支持 Python 2.7 和 3.4 及更高版本，但我们建议您使用最新版本的 AWS 加密 CLI。

大多数 Linux 和 macOS 安装中都包含 Python，但您需要升级到 Python 3.6 或更高版本。建议您使用最新的 Python 版本。在 Windows 上，您必须安装 Python；默认未安装。要下载并安装 Python，请参阅 Python downloads。

要确定是否安装了 Python，请在命令行中键入以下内容：

```
python
```

要检查 Python 版本，请使用 `-V`（大写 V）参数。

```
python -V
```

在 Windows 上，在安装 Python 之后，将 `Python.exe` 文件的路径添加到路径环境变量的值中。

默认情况下，Python 安装在所有用户目录中，或者安装在 `$home` 子目录的用户配置文件目录（`%userprofile%` 或 `AppData\Local\Programs\Python`）中。要在您的系统上查找 `Python.exe` 文件的位置，请检查以下注册表项之一。您可以使用 PowerShell 搜索注册表。

```
PS C:\> dir HKLM:\Software\Python\PythonCore\version\InstallPath
# -or-
PS C:\> dir HKCU:\Software\Python\PythonCore\version\InstallPath
```

pip

pip 是 Python 程序包管理器。要安装 AWS 加密 CLI 及其依赖项，你需要 pip 8.1 或更高版本。有关安装或升级 pip 的帮助，请参阅 pip 文档中的 Installation。

在 Linux 安装中，8.1 **pip** 之前的版本无法构建 AWS 加密 CLI 所需的加密库。如果您选择不更新 pip 版本，可以单独安装构建工具。有关更多信息，请参阅在 Linux 上构建加密。

AWS Command Line Interface

只有在 AWS 加密 CLI AWS KMS keys 中使用 in AWS Command Line Interface (AWS CLI) 时，AWS Key Management Service (AWS KMS) 才是必需的。如果您使用的是不同的主密钥提供程序，则 AWS CLI 不是必需的。

要 AWS KMS keys 与 AWS 加密 CLI 配合使用，您需要安装和配置 AWS CLI。该配置使您用于进行身份验证的凭据 AWS KMS 可供 AWS 加密 CLI 使用。

安装和更新 AWS 加密 CLI

安装最新版本的 AWS 加密 CLI。当你使用pip安装 AWS 加密 CLI 时，它会自动安装 CLI 所需的库，包括 Python 加密库和。AWS Encryption SDK for Python适用于 Python (Boto3) 的 AWS SDK

> **Note**
>
> 4.0.0 之前的 AWS 加密 CLI 版本处于该阶段。end-of-support
>
> 您无需更改任何代码或数据即可安全地从 AWS Encryption CLI 版本 2.1.x 及更高版本更新为最新版本。但是，版本 2.1.x 中引入了新的安全功能，不向后兼容。从 1.7 版本更新。 x 或更早版本，必须先更新到最新的 1。 AWS 加密 CLI 的 x 版本。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。
>
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

安装最新版本的 AWS 加密 CLI

```
pip install aws-encryption-sdk-cli
```

升级到最新版本的 AWS 加密 CLI

```
pip install --upgrade aws-encryption-sdk-cli
```

要查找您的 AWS 加密 CLI 的版本号和 AWS Encryption SDK

```
aws-encryption-cli --version
```

输出列出了两个库的版本号。

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

升级到最新版本的 AWS 加密 CLI

```
pip install --upgrade aws-encryption-sdk-cli
```

安装 AWS 加密 CLI 时还会安装最新版本的（如果尚未安装的话）。 适用于 Python (Boto3) 的 AWS SDK如果安装了 Boto3，安装程序会验证 Boto3 版本并在需要时对其进行更新。

查找您安装的 Boto3 版本

```
pip show boto3
```

更新为最新版本的 Boto3

```
pip install --upgrade boto3
```

要安装当前正在开发的 Enc AWS ryption CLI 版本，请查看上的aws-encryption-sdk-cli存储库 GitHub。

有关使用 `pip` 安装和升级 Python 程序包的更多详细信息，请参阅 pip 文档。

如何使用 AWS 加密 CLI

本主题介绍如何使用 AWS 加密 CLI 中的参数。有关示例，请参阅 AWS 加密 CLI 的示例。有关完整文档，请参阅阅读文档。这些示例中显示的语法适用于 AWS 加密 CLI 版本 2.1。 x 及更高版本。

> **Note**
>
> 4.0.0 之前的 AWS 加密 CLI 版本处于该阶段。end-of-support
> 您无需更改任何代码或数据即可安全地从 AWS Encryption CLI 版本 2.1.x 及更高版本更新为最新版本。但是，版本 2.1.x 中引入了新的安全功能，不向后兼容。从 1.7 版本更新。 x 或更早版本，必须先更新到最新的 1。 AWS 加密 CLI 的 x 版本。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

有关展示如何使用限制加密数据密钥的安全功能的示例，请参阅 限制加密数据密钥。

有关展示如何使用 AWS KMS 多区域密钥的示例，请参阅使用多区域 AWS KMS keys。

主题

- 如何加密和解密数据
- 如何指定包装密钥
- 如何提供输入
- 如何指定输出位置
- 如何使用加密上下文
- 如何指定承诺策略
- 如何在配置文件中存储参数

如何加密和解密数据

加 AWS 密 CLI 使用的功能 AWS Encryption SDK ，可以轻松安全地加密和解密数据。

> **Note**
>
> `--master-keys` 参数在 AWS Encryption CLI 版本 1.8.x 中弃用并在版本 2.1.x 中删除。请改用 `--wrapping-keys` 参数。从版本 2.1.x 开始，加密和解密时需要使用 `--wrapping-keys` 参数。有关更多信息，请参阅 AWS Encryption SDK CLI 语法和参数参考。

- 在加密 CLI 中 AWS 加密数据时，需要指定您的纯文本数据和包装密钥（或主密钥），例如 AWS KMS key in AWS Key Management Service (AWS KMS)。如果使用自定义主密钥提供程序，您还需要指定该提供程序。您还需要指定加密的消息以及有关加密操作的元数据的输出位置。加密上下文是可选的，但建议使用。

  在版本 1.8.x 中，使用 `--wrapping-keys` 参数时需要使用 `--commitment-policy` 参数；否则该参数无效。从版本 2.1.x 开始，`--commitment-policy` 参数是可选的，但建议使用。

  ```
  aws-encryption-cli --encrypt --input myPlaintextData \
                     --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                     --output myEncryptedMessage \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --commitment-policy require-encrypt-require-decrypt
  ```

  AWS 加密 CLI 使用唯一的数据密钥对您的数据进行加密。然后，对您指定的包装密钥下的数据密钥进行加密。它返回加密的消息以及有关该操作的元数据。加密的消息包含加密的数据（密文）以及数据密钥的加密副本。您不必担心数据密钥存储、管理或丢失问题。

- 在解密数据时，您将传入加密的消息、可选的加密上下文以及明文输出和元数据位置。您还可以指定 AWS 加密 CLI 可以用来解密邮件的包装密钥，或者告诉 Encryption AWS CLI 它可以使用任何对邮件进行加密的包装密钥。

  从版本 1.8.x 开始，`--wrapping-keys` 参数在解密时是可选的，但建议使用。从版本 2.1.x 开始，加密和解密时需要使用 `--wrapping-keys` 参数。

  解密时，您可以使用 `--wrapping-keys` 参数的 key 属性来指定用于解密数据的包装密钥。在解密时指定 AWS KMS 包装密钥是可选的，但这是一种最佳做法，可以防止您使用本来不打算使用的密钥。如果使用自定义主密钥提供程序，您必须指定该提供程序和包装密钥。

如果您不使用密钥属性，则必须将--wrapping-keys参数的发现属性设置为，这样true，Encryption CLI 就可以使用 AWS 加密邮件的任何包装密钥进行解密。

最佳实践是使用 --max-encrypted-data-keys 参数来避免使用过多的加密数据密钥解密格式错误的消息。指定预期的加密数据密钥数量（加密中使用的每个包装密钥各一个）或合理的最大值（例如 5）。有关更多信息，请参阅 限制加密数据密钥。

只有在处理完所有输入之后，--buffer 参数才会返回明文，包括验证数字签名（如果存在）。

--decrypt-unsigned 参数对加密文字进行解密并确保消息在解密之前未签名。如果您使用 --algorithm 参数并选择了不带数字签名的算法套件来加密数据，请使用此参数。如果加密文字已签名，则解密失败。

您可以使用 --decrypt 或 --decrypt-unsigned 进行解密，但不能同时使用两者。

```
aws-encryption-cli --decrypt --input myEncryptedMessage \
                   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                   --output myPlaintextData \
                   --metadata-output ~/metadata \
                   --max-encrypted-data-keys 1 \
                   --buffer \
                   --encryption-context purpose=test \
                   --commitment-policy require-encrypt-require-decrypt
```

加 AWS 密 CLI 使用包装密钥来解密加密消息中的数据密钥。然后，它使用数据密钥解密数据。它返回明文数据以及有关该操作的元数据。

如何指定包装密钥

在加密 CLI 中 AWS 加密数据时，需要至少指定一个封装密钥（或主密钥）。您可以 AWS KMS keys 在 AWS Key Management Service (AWS KMS) 中使用、包装来自自定义主密钥提供程序的密钥，或者两者兼而有之。自定义主密钥提供程序可以是任何兼容的 Python 主密钥提供程序。

要在版本 1.8.x 及以后的版本中指定包装密钥，请使用 --wrapping-keys 参数 (-w)。该参数的值是具有 attribute=value 格式的属性集合。您使用的属性取决于主密钥提供程序和命令。

- AWS KMS。在加密命令中，您必须指定具有 key 属性的 --wrapping-keys 参数。从版本 2.1.x 开始，解密命令中也需要使用 --wrapping-keys 参数。解密时，--wrapping-keys 参数必须具有值为 true 的 key 属性或 discovery 属性（但不能两者同时使用）。其他属性是可选的。

- 自定义主密钥提供程序。您必须在每个命令中指定 `--wrapping-keys` 参数。该参数值必须具有 key 和 provider 属性。

您可以在同一命令中包含多个 `--wrapping-keys` 参数和多个 key 属性。

包装密钥参数属性

`--wrapping-keys` 参数值包含以下属性及其值。所有加密命令都需要一个 `--wrapping-keys` 参数（或 `--master-keys` 参数）。从版本 2.1.x 开始，解密时也需要 `--wrapping-keys` 参数。

如果属性名称或值包含空格或特殊字符，请将名称和值用引号引起来。例如 `--wrapping-keys key=12345 "provider=my cool provider"`。

密钥：指定包装密钥

使用 key 属性识别包装密钥。加密时，该值可以是主密钥提供程序识别的任何密钥标识符。

```
--wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
```

在加密命令中，您必须至少包含一个 key 属性和值。要在多个包装密钥下加密您的数据密钥，请使用多个 key 属性。

```
aws-encryption-cli --encrypt --wrapping-keys
 key=1234abcd-12ab-34cd-56ef-1234567890ab key=1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d
```

在使用的加密命令中 AWS KMS keys，密钥的值可以是密钥 ID、其密钥 ARN、别名或别名 ARN。例如，该 encrypt 命令在 key 属性值中使用别名 ARN。有关密钥标识符的详细信息 AWS KMS key，请参阅《AWS Key Management Service 开发者指南》中的密钥标识符。

```
aws-encryption-cli --encrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:alias/ExampleAlias
```

在使用自定义主密钥提供程序的 decrypt 命令中，需要使用 key 和 provider 属性。

```
\\ Custom master key provider
aws-encryption-cli --decrypt --wrapping-keys provider='myProvider' key='100101'
```

在使用的解密命令中 AWS KMS，您可以使用密钥属性来指定用于解密的，或者使用值为的发现属性 AWS KMS keys来指定用于加密消息的`true`，允许加密 AWS CLI 使用任何用于加密 AWS KMS key 消息的属性。如果指定 AWS KMS key，则它必须是用于加密消息的包装密钥之一。

指定包装密钥是 AWS Encryption SDK 最佳实践。它可以确保你使用 AWS KMS key 你打算使用的。

在解密命令中，key 属性的值必须是密钥 ARN。

```
\\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

发现：解密 AWS KMS key 时使用 any

如果您在解密时不需要限制使用 AWS KMS keys ，则可以使用值为的 d iscovery 属性。`true`值为`true`允许 AWS 加密 CLI 使用任何加密消息 AWS KMS key 的内容进行解密。如果不指定 discovery 属性，则发现为 `false`（默认值）。发现属性仅在解密命令中有效，并且仅当消息使用加密时才有效。 AWS KMS keys

值为 `true` 的 discovery 属性可以替代使用 key 属性指定 AWS KMS keys。解密使用加密的消息时 AWS KMS keys，每个`--wrapping-keys`参数都必须有一个密钥属性或一个值为的发现属性`true`，但不能两者兼而有之。

当发现属实时，最佳做法是使用发现分区和发现账户属性将使用限制在您 AWS KMS keys 指定的范围内。 AWS 账户 在以下示例中，发现属性允许 AWS 加密 CLI 使用指定 AWS KMS key 中的任何属性 AWS 账户。

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

提供程序：指定主密钥提供程序

provider 属性指定主密钥提供程序。默认值为 `aws-kms`，它表示 AWS KMS。如果使用不同的主密钥提供程序，则需要使用 provider 属性。

```
--wrapping-keys key=12345 provider=my_custom_provider
```

有关使用自定义（非AWS KMS）主密钥提供程序的更多信息，请参阅 AWS Encryption CLI 存储库的 README 文件中的高级配置主题。

区域：指定一个 AWS 区域

使用 re gion 属性来指定 AWS 区域 的 AWS KMS key。该属性仅在 encrypt 命令中有效，并且仅在主密钥提供程序为 AWS KMS时有效。

```
--encrypt --wrapping-keys key=alias/primary-key region=us-east-2
```

AWS 如果加密 CLI 命令包含区域（例如 ARN），则使用密钥属性值中指定的。如果密钥值指定了 AWS 区域，则区域属性将被忽略。 AWS 区域

region 属性优先于指定的其他区域。如果您不使用区域属性， AWS Encryption CLI 命令将使用您的 AWS CLI 命名配置文件（如果有）或默认配置文件中 AWS 区域 指定的属性。

profile：指定命名配置文件

可以使用 profile 属性指定 AWS CLI 命名配置文件。命名配置文件可以包含凭证和 AWS 区域。只有在主密钥提供程序为 AWS KMS时，该属性才有效。

```
--wrapping-keys key=alias/primary-key profile=admin-1
```

您可以使用 profile 属性在 encrypt 和 decrypt 命令中指定备用凭证。在加密命令 AWS 区域 中，只有当 AWS 密钥值不包括区域且没有区域属性时，Encryption CLI 才在命名配置文件中使用。在解密命令中，名称 AWS 区域 中的配置文件将被忽略。

如何指定多个包装密钥

您可以在每个命令中指定多个包装密钥（或主密钥）。

如果指定多个包装密钥，第一个包装密钥将生成并加密用于加密数据的数据密钥。其他包装密钥对相同的数据密钥进行加密。生成的加密消息包含加密的数据（“加密文字”）以及一组加密的数据密钥，每个包装密钥加密一个数据密钥。任何包装密钥可以解密一个加密的数据密钥，然后解密数据。

可以通过两种方法指定多个包装密钥：

- 在 `--wrapping-keys` 参数值中包含多个 key 属性。

```
$key_oregon=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$key_ohio=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

```
--wrapping-keys key=$key_oregon key=$key_ohio
```

- 在同一命令中包含多个 `--wrapping-keys` 参数。如果您指定的属性值不适用于命令中的所有包装密钥，请使用该语法。

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

值为的发现属性`true`允许 AWS Encryption CLI 使用任何加密 AWS KMS key 消息的内容。如果您在同一个命令中使用多个 `--wrapping-keys` 参数，则在任何 `--wrapping-keys` 参数中使用 `discovery=true` 都会有效地覆盖其他 `--wrapping-keys` 参数中 key 属性的限制。

例如，在以下命令中，第一个`--wrapping-keys`参数中的密钥属性将 AWS 加密 CLI 限制为指定的 AWS KMS key。但是，第二个`--wrapping-keys`参数中的发现属性允许 AWS Encryption CLI 使用指定账户 AWS KMS key 中的任意账户来解密邮件。

```
aws-encryption-cli --decrypt \
    --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
    --wrapping-keys discovery=true \
                    discovery-partition=aws \
                    discovery-account=111122223333 \
                    discovery-account=444455556666
```

如何提供输入

加密 CLI 中的 AWS 加密操作将纯文本数据作为输入并返回加密消息。解密操作将加密的消息作为输入，并返回明文数据。

所有 AWS 加密 CLI 命令都需要`--input`参数 (`-i`)，它告诉加 AWS 密 CLI 在哪里可以找到输入。

您可以通过任何以下方法提供输入：

- 使用文件。

```
--input myData.txt
```

- 使用文件名模式。

```
--input testdir/*.xml
```

- 使用目录或目录名称模式。在输入为目录时，需要使用 `--recursive` 参数 (`-r`, `-R`)。

```
--input testdir --recursive
```

- 通过管道将输入发送到命令 (stdin)。请使用 `-` 参数的值 `--input`。（`--input` 参数始终是必需的。）

```
echo 'Hello World' | aws-encryption-cli --encrypt --input -
```

如何指定输出位置

该`--output`参数告诉 AWS 加密 CLI 在哪里写加密或解密操作的结果。每个 AWS 加密 CLI 命令都需要它。 AWS Encryption CLI 为操作中的每个输入文件创建新的输出文件。

如果输出文件已经存在，则默认情况下， AWS 加密 CLI 会打印警告，然后覆盖该文件。要禁止覆盖，请使用 `--interactive` 参数（在覆盖之前提示您确认）或 `--no-overwrite`（在输出导致覆盖时跳过输入）。要禁止显示覆盖警告，请使用 `--quiet`。要从 Encryption C AWS LI 中捕获错误和警告，请使用2>&1重定向运算符将其写入输出流。

> **Note**
>
> 覆盖输出文件的命令先删除输出文件。如果该命令失败，则可能已删除输出文件。

您可以通过多种方法指定输出位置。

- 指定文件名。如果指定文件的路径，在运行该命令之前，路径中的所有目录必须存在。

```
--output myEncryptedData.txt
```

- 指定目录。在运行该命令之前，输出目录必须存在。

  如果输入包含子目录，该命令将在指定的目录中重新生成这些子目录。

```
--output Test
```

  当输出位置为目录（不含文件名）时， AWS Encryption CLI 会根据输入文件名和后缀创建输出文件名。加密操作将 `.encrypted` 附加到输入文件名后面，而解密操作附加 `.decrypted`。要更改后缀，请使用 `--suffix` 参数。

例如，如果加密 `file.txt`，encrypt 命令将创建 `file.txt.encrypted`。如果解密 `file.txt.encrypted`，decrypt 命令将创建 `file.txt.encrypted.decrypted`。

- 写入到命令行 (stdout)。为 - 参数输入值 `--output`。您可以使用 `--output -` 通过管道将输出发送到另一个命令或程序。

```
--output -
```

如何使用加密上下文

加 AWS 密 CLI 允许您在加密和解密命令中提供加密上下文。这不是必需的，但这是我们建议的加密最佳实践。

加密上下文 是一种任意的非机密其他经过身份验证的数据。在 AWS Encryption CLI 中，加密上下文包含一组 `name=value` 对。您可以在对中使用任意内容，包括有关文件的信息、帮助您在日志中查找加密操作的数据或您的授权或策略所需的数据。

在 encrypt 命令中

在 encrypt 命令中指定的加密上下文以及 CMM 添加的任何其他对以加密方式绑定到加密的数据。它还包含（以明文形式）在该命令返回的加密的消息中。如果您使用的是 AWS KMS key，加密上下文也可能以纯文本形式出现在审计记录和日志中，例如。 AWS CloudTrail

以下示例显示具有三个 `name=value` 对的加密上下文。

```
--encryption-context purpose=test dept=IT class=confidential
```

在 decrypt 命令中

在 decrypt 命令中，加密上下文帮助您确认解密的是正确的加密消息。

不需要在 decrypt 命令中提供加密上下文，即使在加密时使用了加密上下文。但是，如果这样做，AWS Encryption CLI 会验证 decrypt 命令的加密上下文中的每个元素是否都与加密邮件的加密上下文中的元素相匹配。如果任何元素不匹配，decrypt 命令将失败。

例如，只有在加密上下文包含 `dept=IT` 时，以下命令才会解密加密的消息。

```
aws-encryption-cli --decrypt --encryption-context dept=IT ...
```

加密上下文是安全策略的重要组成部分。不过，在选择加密上下文时，请记住它的值不是机密的。请不要在加密上下文中包含任何机密数据。

指定加密上下文

- 在 encrypt 命令中，使用具有一个或多个 name=value 对的 --encryption-context 参数。请使用空格分隔每个对。

```
--encryption-context name=value [name=value] ...
```

- 在 decrypt 命令中，--encryption-context 参数值可以包含 name=value 对、name 元素（没有值）或两者的组合。

```
--encryption-context name[=value] [name] [name=value] ...
```

如果 name 对中的 value 或 name=value 包含空格或特殊字符，请将整个对用引号引起来。

```
--encryption-context "department=software engineering" "AWS ##=us-west-2"
```

例如，该 encrypt 命令包含具有两个对（purpose=test 和 dept=23）的加密上下文。

```
aws-encryption-cli --encrypt --encryption-context purpose=test dept=23 ...
```

这些 decrypt 命令将会成功。每个命令中的加密上下文是原始加密上下文的一部分。

```
\\ Any one or both of the encryption context pairs
aws-encryption-cli --decrypt --encryption-context dept=23 ...

\\ Any one or both of the encryption context names
aws-encryption-cli --decrypt --encryption-context purpose ...

\\ Any combination of names and pairs
aws-encryption-cli --decrypt --encryption-context dept purpose=test ...
```

不过，这些 decrypt 命令将会失败。加密的消息中的加密上下文不包含指定的元素。

```
aws-encryption-cli --decrypt --encryption-context dept=Finance ...
aws-encryption-cli --decrypt --encryption-context scope ...
```

如何指定承诺策略

要为命令设置承诺策略，请使用 `--commitment-policy` 参数。此参数在版本 1.8.x 中引入。该参数在加密和解密命令中有效。您设置的承诺策略仅对出现在其中的命令有效。如果您没有为命令设置承诺策略，则 AWS 加密 CLI 将使用默认值。

例如，以下参数值将承诺策略设置为 `require-encrypt-allow-decrypt`，该策略始终使用密钥承诺进行加密，但会解密使用或不使用密钥承诺加密的加密文字。

```
--commitment-policy require-encrypt-allow-decrypt
```

如何在配置文件中存储参数

通过将常用的 Encryption CLI 参数和值保 AWS 存在配置文件中，可以节省时间并避免键入错误。

配置文件是一个文本文件，其中包含 AWS 加密 CLI 命令的参数和值。在 AWS Encryption CLI 命令中引用配置文件时，引用将替换为配置文件中的参数和值。如果在命令行中键入文件内容，效果是相同的。配置文件可以具有任何名称，并且可以位于当前用户可访问的任何目录中。

以下示例配置文件 (`key.conf`) 指定不同区域中的两个 AWS KMS keys 。

```
--wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
--wrapping-keys key=arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

要在命令中使用配置文件，请在文件名前面添加 at 符号 (@)。在 PowerShell 主机中，使用反引号字符对 at 符号 (`@) 进行转义。

以下示例命令在 encrypt 命令中使用 `key.conf` 文件。

Bash

```
$ aws-encryption-cli -e @key.conf -i hello.txt -o testdir
```

PowerShell

```
PS C:\> aws-encryption-cli -e `@key.conf -i .\Hello.txt -o .\TestDir
```

配置文件规则

使用配置文件的规则如下所示：

- 您可以在每个配置文件中包含多个参数，并按任意顺序列出这些参数。请在单独一行中列出每个参数及其值（如果有）。
- 使用 # 为一行的全部或部分内容添加注释。
- 您可以包含对其他配置文件的引用。即使在，也不要使用反引号逃避@标志。 PowerShell
- 如果在配置文件中使用引号，引起来的文本不能跨多个行。

例如，以下是示例 `encrypt.conf` 文件的内容。

```
# Archive Files
--encrypt
--output /archive/logs
--recursive
--interactive
--encryption-context class=unclassified dept=IT
--suffix  # No suffix
--metadata-output ~/metadata
@caching.conf  # Use limited caching
```

也可以在命令中包含多个配置文件。以下示例命令使用 `encrypt.conf` 和 `master-keys.conf` 配置文件。

Bash

```
$  aws-encryption-cli -i /usr/logs @encrypt.conf @master-keys.conf
```

PowerShell

```
PS C:\> aws-encryption-cli -i $home\Test\*.log `@encrypt.conf `@master-keys.conf
```

下一步：试用 AWS Encryption CLI 示例

AWS 加密 CLI 的示例

使用以下示例在您喜欢的平台上试用 AWS 加密 CLI。有关主密钥和其他参数的帮助，请参阅如何使用 AWS 加密 CLI。有关快速参考，请参阅 AWS Encryption SDK CLI 语法和参数参考。

Note

以下示例使用 AWS 加密 CLI 版本 2.1 的语法。 x。

新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

有关展示如何使用限制加密数据密钥的安全功能的示例，请参阅 限制加密数据密钥。

有关如何使用 AWS KMS 多区域密钥的示例，请参阅使用多区域 AWS KMS keys。

主题

- 加密文件
- 解密文件
- 加密目录中的所有文件
- 解密目录中的所有文件
- 在命令行上加密和解密
- 使用多个主密钥
- 在脚本中加密和解密
- 使用数据密钥缓存

加密文件

此示例使用 AWS 加密 CLI 对文件内容进行加密，该`hello.txt`文件包含 “Hello World” 字符串。

当您对文件运行加密命令时， AWS Encryption CLI 会获取文件内容，生成唯一的数据密钥，加密数据密钥下的文件内容，然后将加密的消息写入新文件。

第一个命令将的密钥 ARN 保存在变量 AWS KMS key 中。`$keyArn`使用加密时 AWS KMS key，您可以使用密钥 ID、密钥 ARN、别名或别名 ARN 来识别它。有关密钥标识符的详细信息 AWS KMS key，请参阅《AWS Key Management Service 开发者指南》中的密钥标识符。

第二个命令加密文件内容。该命令使用 `--encrypt` 参数指定操作，并使用 `--input` 参数指示要加密的文件。`--wrapping-keys`参数及其必需的密钥属性告诉命令使用由密钥 ARN AWS KMS key 表示的。

该命令使用 --metadata-output 参数指定一个包含有关加密操作的元数据的文本文件。作为最佳实践，该命令使用 --encryption-context 参数指定一个加密上下文。

此命令还使用 --commitment-policy 参数来明确设置承诺策略。在版本 1.8.x 中，当您使用 --wrapping-keys 参数时，需要使用这个参数。从版本 2.1.x 开始，--commitment-policy 参数是可选的，但建议使用。

--output 参数的值“句点”(.) 指示命令将输出文件写入到当前目录中。

Bash

```
\\ To run this example, replace the fictitious key ARN with a valid value.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --commitment-policy require-encrypt-require-decrypt \
                     --output .
```

PowerShell

```
# To run this example, replace the fictitious key ARN with a valid value.
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --encrypt `
                           --input Hello.txt `
                           --wrapping-keys key=$keyArn `
                           --metadata-output $home\Metadata.txt `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --output .
```

如果 encrypt 命令成功，它不返回任何输出。要确定该命令是否成功，请检查 $? 变量中的布尔值。命令成功后，的值$?为 0 (Bash) 或 True (PowerShell)。命令失败时，的值$?为非零 (Bash) 或 False (PowerShell)。

Bash

```
$ echo $?
0
```

PowerShell

```
PS C:\> $?
True
```

也可以使用目录列表命令查看 encrypt 命令是否创建了新的文件 (`hello.txt.encrypted`)。由于 encrypt 命令没有为输出指定文件名，因此 AWS Encryption CLI 会将输出写入一个与输入文件同名并带有`.encrypted`后缀的文件中。要使用不同的后缀或忽略后缀，请使用 `--suffix` 参数。

`hello.txt.encrypted` 文件包含加密的消息，其中包含 `hello.txt` 文件的密文、数据密钥的加密副本以及额外的元数据（包括加密上下文）。

Bash

```
$  ls
hello.txt  hello.txt.encrypted
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
```

解密文件

此示例使用 AWS 加密 CLI 来解密前面示例中加密`Hello.txt.encrypted`的文件内容。

decrypt 命令使用 `--decrypt` 参数指示操作，并使用 `--input` 参数指定要解密的文件。`--output` 参数的值是一个句点，表示当前的目录。

带有 key 属性的 `--wrapping-keys` 参数指定用于解密加密消息的包装密钥。在使用解密命令中 AWS KMS keys，密钥属性的值必须是密钥 AR N。在解密命令中需要使用 `--wrapping-keys` 参数。如果您正在使用 AWS KMS keys，则可以使用 key 属性来指定 AWS KMS keys 用于解密，也可以使用值为 `true` 的 discovery 属性（但不能两者同时使用）。如果使用自定义主密钥提供程序，则需要使用 key 和 provider 属性。

从版本 2.1.x 开始，`--commitment-policy` 参数是可选的，但建议使用。即使您指定了默认值 `require-encrypt-require-decrypt`，也可以明确使用该参数来明确您的意图。

`--encryption-context` 参数在 decrypt 命令中是可选的，即使在 encrypt 命令中提供了加密上下文。在这种情况下，decrypt 命令使用在 encrypt 命令中提供的相同加密上下文。在解密之前，Encryption CL AWS I 会验证加密消息中的加密上下文是否包含一对。`purpose=test`如果不包含，decrypt 命令将失败。

`--metadata-output` 参数指定一个包含有关解密操作的元数据的文件。`--output` 参数的值“句点”(.) 指示将输出文件写入到当前目录中。

最佳实践是使用 `--max-encrypted-data-keys` 参数来避免使用过多的加密数据密钥解密格式错误的消息。指定预期的加密数据密钥数量（加密中使用的每个包装密钥各一个）或合理的最大值（例如 5）。有关更多信息，请参阅 限制加密数据密钥。

只有在处理完所有输入之后，`--buffer` 才会返回明文，包括验证数字签名（如果存在）。

Bash

```
\\ To run this example, replace the fictitious key ARN with a valid value.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy require-encrypt-require-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output .
```

PowerShell

```
\\ To run this example, replace the fictitious key ARN with a valid value.
```

```
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --decrypt `
                           --input Hello.txt.encrypted `
                           --wrapping-keys key=$keyArn `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
                           --buffer `
                           --output .
```

如果 decrypt 命令成功，它不返回任何输出。要确定该命令是否成功，请获取 $? 变量值。也可以使用目录列表命令查看该命令是否创建了具有 `.decrypted` 后缀的新文件。要查看明文内容，请使用一个命令以获取文件内容，例如 `cat` 或 Get-Content。

Bash

```
$  ls
hello.txt  hello.txt.encrypted  hello.txt.encrypted.decrypted

$  cat hello.txt.encrypted.decrypted
Hello World
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   1:01 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
-a----        9/17/2017   1:08 PM             11 Hello.txt.encrypted.decrypted


PS C:\> Get-Content Hello.txt.encrypted.decrypted
Hello World
```

加密目录中的所有文件

此示例使用 AWS 加密 CLI 对目录中所有文件的内容进行加密。

当一个命令影响多个文件时， AWS 加密 CLI 会单独处理每个文件。它获取文件内容，从主密钥中获取文件的唯一数据密钥，使用该数据密钥加密文件内容，然后将结果写入到输出目录的新文件中。因此，您可以单独解密这些输出文件。

以下 `TestDir` 目录列表显示我们要加密的明文文件。

Bash

```
$  ls testdir
cool-new-thing.py  hello.txt  employees.csv
```

PowerShell

```
PS C:\> dir C:\TestDir

    Directory: C:\TestDir

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/12/2017   3:11 PM           2139 cool-new-thing.py
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:44 PM             46 Employees.csv
```

第一个命令将的 Amazon 资源名称 (ARN) 保存在变量 AWS KMS key 中。`$keyArn`

第二个命令加密 `TestDir` 目录中的文件内容，并将包含加密内容的文件写入到 `TestEnc` 目录中。如果 `TestEnc` 目录不存在，该命令将失败。由于输入位置是一个目录，因此，`--recursive` 参数是必需的。

`--wrapping-keys` 参数及其所需的 key 属性指定要使用的包装密钥。encrypt 命令包含一个加密上下文 (`dept=IT`)。如果在加密多个文件的命令中指定加密上下文，将在所有文件中使用相同的加密上下文。

该命令还有一个`--metadata-output`参数，用于告诉 AWS Encryption CLI 在哪里写入有关加密操作的元数据。加 AWS 密 CLI 为每个加密文件写入一条元数据记录。

从版本 2.1.x 开始，`--commitment-policy parameter` 是可选的，但建议使用。如果命令或脚本因无法解密加密文字而失败，则显式承诺策略设置可以帮助您快速检测问题。

命令完成后， AWS 加密 CLI 会将加密文件写入该`TestEnc`目录，但不会返回任何输出。

最后一个命令列出 `TestEnc` 目录中的文件。每个包含明文内容的输入文件具有一个包含加密内容的输出文件。由于该命令未指定替代后缀，因此，encrypt 命令将 `.encrypted` 附加到每个输入文件名称后面。

Bash

```
# To run this example, replace the fictitious key ARN with a valid master key
 identifier.
$  keyArn=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --encrypt \
                     --input testdir --recursive\
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
                     --metadata-output ~/metadata \
                     --output testenc

$ ls testenc
cool-new-thing.py.encrypted  employees.csv.encrypted  hello.txt.encrypted
```

PowerShell

```
# To run this example, replace the fictitious key ARN with a valid master key
 identifier.
PS C:\> $keyArn = arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

PS C:\> aws-encryption-cli --encrypt `
                           --input .\TestDir --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output .\Metadata\Metadata.txt `
                           --output .\TestEnc
```

```
PS C:\> dir .\TestEnc

    Directory: C:\TestEnc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

解密目录中的所有文件

该示例解密目录中的所有文件。它从 `TestEnc` 目录中在上一个示例中加密的文件开始。

Bash

```
$  ls testenc
cool-new-thing.py.encrypted  hello.txt.encrypted  employees.csv.encrypted
```

PowerShell

```
PS C:\> dir C:\TestEnc

    Directory: C:\TestEnc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

此 decrypt 命令解密目录中的所有文件，并将纯文本文件写入该 TestEnc 目录。 TestDec 带有密钥属性和密钥 ARN 值的`--wrapping-keys`参数告诉加密 AWS CLI 使用哪个 AWS KMS keys 来解密文件。该命令使用`--interactive`参数告诉 AWS 加密 CLI 在覆盖同名文件之前提示您。

该命令还使用在加密文件时提供的加密上下文。解密多个文件时，加密 AWS CLI 会检查每个文件的加密上下文。如果对任何文件的加密上下文检查失败， AWS Encryption CLI 会拒绝该文件，写入警告，在元数据中记录失败，然后继续检查其余文件。如果 AWS 加密 CLI 由于任何其他原因无法解密文件，则整个 decrypt 命令将立即失败。

在该示例中，所有输入文件中的加密消息包含 dept=IT 加密上下文元素。不过，如果解密的消息具有不同的加密上下文，您仍然可以验证加密上下文部分。例如，如果某些消息具有 dept=finance 加密上下文，而其他消息具有 dept=IT 加密上下文，您可以确认加密上下文始终包含未指定值的 dept 名称。如果要了解更具体的信息，您可以在单独的命令中解密这些文件。

decrypt 命令不返回任何输出，但您可以使用目录列表命令查看它是否创建了具有 .decrypted 后缀的新文件。要查看明文内容，请使用一个命令以获取文件内容。

Bash

```
# To run this example, replace the fictitious key ARN with a valid master key
 identifier.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$ aws-encryption-cli --decrypt \
                     --input testenc --recursive \
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 1 \
                     --buffer \
                     --output testdec --interactive

$ ls testdec
cool-new-thing.py.encrypted.decrypted  hello.txt.encrypted.decrypted
 employees.csv.encrypted.decrypted
```

PowerShell

```
# To run this example, replace the fictitious key ARN with a valid master key
 identifier.
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --decrypt `
                           --input C:\TestEnc --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
```

```
                             --buffer `
                             --output C:\TestDec --interactive

PS C:\> dir .\TestDec


    Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        10/8/2017   4:57 PM           2139 cool-new-
thing.py.encrypted.decrypted
-a----        10/8/2017   4:57 PM             46 Employees.csv.encrypted.decrypted
-a----        10/8/2017   4:57 PM             11 Hello.txt.encrypted.decrypted
```

在命令行上加密和解密

以下示例介绍了如何通过管道将输入发送到命令 (stdin)，以及将输出写入到命令行 (stdout)。这些示例说明了如何在命令中表示 stdin 和 stdout，以及如何使用内置的 Base64 编码工具防止 shell 错误地解释非 ASCII 字符。

该示例通过管道将明文字符串发送到 encrypt 命令，并将加密的消息保存在变量中。然后，它通过管道将变量中的加密消息发送到 decrypt 命令，后者将其输出写入到管道中 (stdout)。

该示例包含以下三个命令：

- 第一个命令将的密钥 ARN 保存在变量 AWS KMS key 中。`$keyArn`

  Bash

  ```
  $  keyArn=arn:aws:kms:us-
  west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
  ```

  PowerShell

  ```
  PS C:\> $keyArn = 'arn:aws:kms:us-
  west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
  ```

- 第二个命令通过管道将 `Hello World` 字符串发送到 encrypt 命令，并将结果保存在 `$encrypted` 变量中。

在所有 AWS Encryption CLI 命令中需要使用 --input 和 --output 参数。要指示通过管道将输入发送到命令 (stdin)，请将连字符 (-) 作为 --input 参数值。要将输出发送到命令行 (stdout)，请将连字符作为 --output 参数值。

--encode 参数在返回输出之前对其进行 Base64 编码。这可防止 shell 错误地解释加密消息中的非 ASCII 字符。

由于该命令只是概念验证，因此，我们省略加密上下文并忽略元数据 (-S)。

Bash

```
$ encrypted=$(echo 'Hello World' | aws-encryption-cli --encrypt -S \
                                                      --input - --output - --
encode \
                                                      --wrapping-keys key=
$keyArn )
```

PowerShell

```
PS C:\> $encrypted = 'Hello World' | aws-encryption-cli --encrypt -S `
                                                        --input - --output - --
encode `
                                                        --wrapping-keys key=
$keyArn
```

- 第三个命令通过管道将 $encrypted 变量中的加密消息发送到 decrypt 命令。

  该 decrypt 命令使用 --input - 指示输入来自于管道 (stdin)，并使用 --output - 将输出发送到管道 (stdout)。（input 参数使用输入位置而不是实际输入字节，因此，您不能将 $encrypted 变量作为 --input 参数值。）

  此示例使用--wrapping-keys参数的发现属性允许 AWS Encryption CLI 使用任何属性 AWS KMS key 来解密数据。该示例没有指定承诺策略，因此使用版本 2.1.x 及更高版本的默认值 require-encrypt-require-decrypt。

  由于输出已加密并随后进行编码，因此，decrypt 命令在解密 Base64 编码的输入之前使用 --decode 参数对其进行解码。您也可以在加密 Base64 编码的输入之前使用 --decode 参数对其进行解码。

同样，该命令省略加密上下文并忽略元数据 (-S)。

Bash

```
$  echo $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=true
 --input - --output - --decode --buffer -S
Hello World
```

PowerShell

```
PS C:\> $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=$true
 --input - --output - --decode --buffer -S
Hello World
```

您也可以在单个命令中执行加密和解密操作，而无需使用变量。

正如前面的示例所示，`--input` 和 `--output` 参数具有 `-` 值，命令使用 `--encode` 参数对输出进行编码，并使用 `--decode` 参数对输入进行解码。

Bash

```
$  keyArn=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$  echo 'Hello World' |
          aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
          aws-encryption-cli --decrypt --wrapping-keys discovery=true --input - --
output - --decode -S
Hello World
```

PowerShell

```
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> 'Hello World' |
               aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
               aws-encryption-cli --decrypt --wrapping-keys discovery=$true --input
 - --output - --decode -S
```

```
Hello World
```

使用多个主密钥

此示例说明如何在加密 CLI 中加密和解密数据时使用多个主密钥。 AWS

在使用多个主密钥加密数据时，可以使用任何一个主密钥解密数据。该策略确保您始终可以解密数据，即使某个主密钥不可用。如果您要将加密的数据存储在多个中 AWS 区域，则此策略允许您在同一区域使用主密钥来解密数据。

在使用多个主密钥进行加密时，第一个主密钥起到特殊的作用。它生成用于加密数据的数据密钥。其余主密钥加密明文数据密钥。生成的加密消息包含加密的数据以及加密的数据密钥集合，每个主密钥具有一个加密的数据密钥。虽然数据密钥是第一个主密钥生成的，但任何主密钥都可以解密其中的一个数据密钥，这些数据密钥可用于解密数据。

使用三个主密钥进行加密

该示例命令使用三个包装密钥加密 `Finance.log` 文件，在三个 AWS 区域中各具有一个包装密钥。

它将加密的消息写入到 `Archive` 目录中。该命令使用没有值的 `--suffix` 参数以忽略后缀，因此，输入和输出文件名称是相同的。

该命令使用具有三个 key 属性的 `--wrapping-keys` 参数。您也可以在同一命令中使用多个 `--wrapping-keys` 参数。

要加密日志文件， AWS Encryption CLI 会要求列表中的第一个封装密钥生成用于加密数据的数据密钥。`$key1`然后，分别使用其他包装密钥加密相同数据密钥的明文副本。输出文件中的加密消息包含所有三个加密的数据密钥。

Bash

```
$ key1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ key2=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
$ key3=arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

$ aws-encryption-cli --encrypt --input /logs/finance.log \
                               --output /archive --suffix \
                               --encryption-context class=log \
                               --metadata-output ~/metadata \
```

```
                                --wrapping-keys key=$key1 key=$key2 key=$key3
```

PowerShell

```
PS C:\> $key1 = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
PS C:\> $key2 = 'arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef'
PS C:\> $key3 = 'arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d'

PS C:\> aws-encryption-cli --encrypt --input D:\Logs\Finance.log `
                           --output D:\Archive --suffix `
                           --encryption-context class=log `
                           --metadata-output $home\Metadata.txt `
                           --wrapping-keys key=$key1 key=$key2 key=$key3
```

该命令解密 `Finance.log` 文件的加密副本，并将其写入到 `Finance.log.clear` 目录中的 `Finance` 文件。要解密三以下加密的数据 AWS KMS keys，可以指定相同的三个 AWS KMS keys 或其中的任何子集。该示例仅指定了其中一个 AWS KMS keys。

要告知 AWS 加密 CLI 使用哪个 AWS KMS keys 来解密您的数据，请使用参数的**`--wrapping-keys`**密钥属性。使用解密时 AWS KMS keys，密钥属性的值必须是密钥 AR N。

您必须有权在 AWS KMS keys 您指定的上调用 Decrypt API。有关更多信息，请参阅 AWS KMS的身份验证和访问控制。

作为最佳实践，该示例使用 `--max-encrypted-data-keys` 参数来避免使用过多的加密数据密钥来解密格式错误的消息。尽管该示例仅使用一个包装密钥进行解密，但加密的消息有三（3）个加密的数据密钥；加密时使用的三个包装密钥各有一个。指定预期的加密数据密钥数量或合理的最大值，例如 5。如果指定的最大值小于 3，则命令将失败。有关更多信息，请参阅 限制加密数据密钥。

Bash

```
$ aws-encryption-cli --decrypt --input /archive/finance.log \
                     --wrapping-keys key=$key1 \
                     --output /finance --suffix '.clear' \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 3 \
                     --buffer \
                     --encryption-context class=log
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input D:\Archive\Finance.log `
                           --wrapping-keys key=$key1 `
                           --output D:\Finance --suffix '.clear' `
                           --metadata-output .\Metadata\Metadata.txt `
                           --max-encrypted-data-keys 3 `
                           --buffer `
                           --encryption-context class=log
```

在脚本中加密和解密

此示例说明如何在脚本中使用 AWS 加密 CLI。您可以编写仅加密和解密数据的脚本，或者编写在数据管理过程中加密或解密的脚本。

在该示例中，脚本会获取一组日志文件，压缩并加密这些文件，然后将加密的文件复制到 Amazon S3 存储桶中。该脚本分别处理每个文件，以便单独解密和展开这些文件。

在压缩并加密文件时，请务必在加密之前进行压缩。无法压缩正确加密的数据。

> **Warning**
>
> 在压缩的数据包含可能被恶意攻击者控制的密钥和数据时，要格外小心。压缩的数据的最终大小可能会无意中泄露有关其内容的敏感信息。

Bash

```
# Continue running even if an operation fails.
set +e

dir=$1
encryptionContext=$2
s3bucket=$3
s3folder=$4
masterKeyProvider="aws-kms"
metadataOutput="/tmp/metadata-$(date +%s)"

compress(){
    gzip -qf $1
```

```
}

encrypt(){
    # -e encrypt
    # -i input
    # -o output
    # --metadata-output unique file for metadata
    # -m masterKey read from environment variable
    # -c encryption context read from the second argument.
    # -v be verbose
    aws-encryption-cli -e -i ${1} -o $(dirname ${1}) --metadata-output
 ${metadataOutput} -m key="${masterKey}" provider="${masterKeyProvider}" -c
 "${encryptionContext}" -v
}


s3put (){
    # copy file argument 1 to s3 location passed into the script.
    aws s3 cp ${1} ${s3bucket}/${s3folder}
}

# Validate all required arguments are present.
if [ "${dir}" ] && [ "${encryptionContext}" ] && [ "${s3bucket}" ] &&
 [ "${s3folder}" ] && [ "${masterKey}" ]; then

# Is $dir a valid directory?
test -d "${dir}"
if [ $? -ne 0 ]; then
    echo "Input is not a directory; exiting"
    exit 1
fi

# Iterate over all the files in the directory, except *gz and *encrypted (in case of
 a re-run).
for f in $(find ${dir} -type f \( -name "*" ! -name \*.gz ! -name \*encrypted \) );
 do
    echo "Working on $f"
    compress ${f}
    encrypt ${f}.gz
    rm -f ${f}.gz
    s3put ${f}.gz.encrypted
done;
else
    echo "Arguments: <Directory> <encryption context> <s3://bucketname> <s3 folder>"
```

```
    echo " and ENV var \$masterKey must be set"
    exit 255
fi
```

PowerShell

```
#Requires -Modules AWSPowerShell, Microsoft.PowerShell.Archive
Param
(
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String[]]
    $FilePath,

    [Parameter()]
    [Switch]
    $Recurse,

    [Parameter(Mandatory=$true)]
    [String]
    $wrappingKeyID,

    [Parameter()]
    [String]
    $masterKeyProvider = 'aws-kms',

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $ZipDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $EncryptDirectory,

    [Parameter()]
    [String]
    $EncryptionContext,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
```

```
    $MetadataDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-S3Bucket -BucketName $_})]
    [String]
    $S3Bucket,

    [Parameter()]
    [String]
    $S3BucketFolder
)

BEGIN {}
PROCESS {
    if ($files = dir $FilePath -Recurse:$Recurse)
    {

        # Step 1: Compress
        foreach ($file in $files)
        {
            $fileName = $file.Name
            try
            {
                Microsoft.PowerShell.Archive\Compress-Archive -Path $file.FullName -
DestinationPath $ZipDirectory\$filename.zip
            }
            catch
            {
                Write-Error "Zip failed on $file.FullName"
            }

            # Step 2: Encrypt
            if (-not (Test-Path "$ZipDirectory\$filename.zip"))
            {
                Write-Error "Cannot find zipped file: $ZipDirectory\$filename.zip"
            }
            else
            {
                # 2>&1 captures command output
                $err = (aws-encryption-cli -e -i "$ZipDirectory\$filename.zip" `
                                           -o $EncryptDirectory `
                                           -m key=$wrappingKeyID provider=
$masterKeyProvider `
                                           -c $EncryptionContext `
```

```
                                                    --metadata-output $MetadataDirectory `
                                                    -v) 2>&1

                    # Check error status
                    if ($? -eq $false)
                    {
                        # Write the error
                        $err
                    }
                    elseif (Test-Path "$EncryptDirectory\$fileName.zip.encrypted")
                    {
                        # Step 3: Write to S3 bucket
                        if ($S3BucketFolder)
                        {
                            Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted" -Key "$S3BucketFolder/
$fileName.zip.encrypted"

                        }
                        else
                        {
                            Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted"
                        }
                    }
                }
            }
        }
}
```

使用数据密钥缓存

该示例在加密大量文件的命令中使用数据密钥缓存。

默认情况下， AWS Encryption CLI（以及的其他版本 AWS Encryption SDK）会为其加密的每个文件生成一个唯一的数据密钥。虽然在每个操作中使用唯一的数据密钥是加密最佳实践，但在某些情况下将数据密钥重用有限次数是可以接受的。如果考虑使用数据密钥缓存，请咨询安全工程师以了解您的应用程序的安全要求，并确定适合您的安全阈值。

在该示例中，数据密钥缓存降低发送到主密钥提供程序的请求频率以加快加密操作速度。

该示例中的命令加密一个具有多个子目录的大型目录，其中总共包含大约 800 个小日志文件。第一个命令将 AWS KMS key 的 ARN 保存在 `keyARN` 变量中。第二个命令加密输入目录中的所有文件（以递归方式），并将这些文件写入到存档目录中。该命令使用 `--suffix` 参数指定 `.archive` 后缀。

`--caching` 参数启用数据密钥缓存。capacity 属性（限制缓存中的数据密钥数）设置为 1，因为串行文件处理每次从不使用超过一个数据密钥。max_age 属性（确定可使用缓存的数据密钥的时间长度）设置为 10 秒。

可选的 max_messages_encrypted 属性设置为 10 个消息，因此，从不使用单个数据密钥加密超过 10 个文件。通过限制每个数据密钥加密的文件数，可以减少在极少数情况下数据密钥泄露而受到影响的文件数。

要对操作系统生成的日志文件运行该命令，您可能需要具有管理员权限（Linux 中的 `sudo`；Windows 系统中的以管理员身份运行）。

Bash

```
$  keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$  aws-encryption-cli --encrypt \
                      --input /var/log/httpd --recursive \
                      --output ~/archive --suffix .archive \
                      --wrapping-keys key=$keyArn \
                      --encryption-context class=log \
                      --suppress-metadata \
                      --caching capacity=1 max_age=10 max_messages_encrypted=10
```

PowerShell

```
PS C:\> $keyARN = 'arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --encrypt `
                           --input C:\Windows\Logs --recursive `
                           --output $home\Archive --suffix '.archive' `
                           --wrapping-keys key=$keyARN `
                           --encryption-context class=log `
                           --suppress-metadata `
                           --caching capacity=1 max_age=10
 max_messages_encrypted=10
```

为了测试数据密钥缓存的效果，此示例在中使用了 Measu re-Command cmdlet。 PowerShell如果运行该示例而不使用数据密钥缓存，大约需要 25 秒的时间才能完成。该过程为目录中的每个文件生成新的数据密钥。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata }



Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 25
Milliseconds      : 453
Ticks             : 254531202
TotalDays         : 0.000294596298611111
TotalHours        : 0.00707031116666667
TotalMinutes      : 0.42421867
TotalSeconds      : 25.4531202
TotalMilliseconds : 25453.1202
```

数据密钥缓存可以加快该过程的速度，即使将每个数据密钥限制为最多用于 10 个文件。该命令现在需要不到 12 秒的时间即可完成，并将对主密钥提供程序的调用次数减少到原来的 1/10。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10
 max_messages_encrypted=10}



Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 11
```

```
Milliseconds      : 813
Ticks             : 118132640
TotalDays         : 0.000136727592592593
TotalHours        : 0.00328146222222222
TotalMinutes      : 0.196887733333333
TotalSeconds      : 11.813264
TotalMilliseconds : 11813.264
```

如果消除 max_messages_encrypted 限制，则使用同一数据密钥加密所有文件。该更改增加了重用数据密钥的风险，而不会显著加快该过程的速度。不过，它将对主密钥提供程序的调用次数减少到 1 次。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10}



Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 10
Milliseconds      : 252
Ticks             : 102523367
TotalDays         : 0.000118661304398148
TotalHours        : 0.00284787130555556
TotalMinutes      : 0.170872278333333
TotalSeconds      : 10.2523367
TotalMilliseconds : 10252.3367
```

AWS Encryption SDK CLI 语法和参数参考

本主题提供了语法图表和简要参数描述以帮助您使用 AWS Encryption SDK 命令行界面 (CLI)。有关包装密钥和其他参数的帮助，请参阅 如何使用 AWS 加密 CLI。有关示例，请参阅 AWS 加密 CLI 的示例。有关完整文档，请参阅阅读文档。

主题

- AWS 加密 CLI 语法

- AWS 加密 CLI 命令行参数
- 高级参数

AWS 加密 CLI 语法

这些 AWS 加密 CLI 语法图显示了您使用 AWS 加密 CLI 执行的每项任务的语法。它们代表 AWS 加密 CLI 版本 2.1 中的推荐语法。 x 及更高版本。

新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

> **Note**
>
> 除非在参数描述中注明，否则每个参数或属性只能在每个命令中使用一次。
> 如果您使用参数不支持的属性，Encryption CL AWS I 会忽略该不支持的属性，而不会出现警告或错误。

获取帮助

要获取包含参数描述的完整 AWS 加密 CLI 语法，请使用`--help`或`-h`。

```
aws-encryption-cli (--help | -h)
```

获取版本

要获取 AWS 加密 CLI 安装的版本号，请使用`--version`。在提问、报告问题或分享有关使用 Encryption CLI 的提示时，请务必 AWS 包含该版本。

```
aws-encryption-cli --version
```

加密数据

以下语法图表显示 encrypt 命令使用的参数。

```
aws-encryption-cli --encrypt
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite] [--suffix
 [<suffix>]] [--encode]
```

```
                    --wrapping-keys  [--wrapping-keys] ...
                        key=<keyID> [key=<keyID>] ...
                        [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                    --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                    [--commitment-policy <commitment-policy>]
                    [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                    [--max-encrypted-data-keys <integer>]
                    [--algorithm <algorithm_suite>]
                    [--caching <attributes>]
                    [--frame-length <length>]
                    [-v | -vv | -vvv | -vvvv]
                    [--quiet]
```

解密数据

以下语法图表显示 decrypt 命令使用的参数。

在版本 1.8.x 中，`--wrapping-keys` 参数在解密时是可选的，但建议使用。从版本 2.1.x 开始，加密和解密时需要使用 `--wrapping-keys` 参数。对于 AWS KMS keys，您可以使用 key 属性来指定包装密钥（最佳实践），也可以将 discovery 属性设置为 `true`，这不会限制 AWS Encryption CLI 可以使用的包装密钥。

```
aws-encryption-cli --decrypt (or [--decrypt-unsigned])
                    --input <input> [--recursive] [--decode]
                    --output <output> [--interactive] [--no-overwrite]  [--suffix
 [<suffix>]] [--encode]
                    --wrapping-keys  [--wrapping-keys] ...
                        [key=<keyID>] [key=<keyID>] ...
                        [discovery={true|false}] [discovery-partition=<aws-partition-
name> discovery-account=<aws-account-ID> [discovery-account=<aws-account-ID>] ...]
                        [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                    --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                    [--commitment-policy <commitment-policy>]
                    [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                    [--buffer]
                    [--max-encrypted-data-keys <integer>]
                    [--caching <attributes>]
```

```
                    [--max-length <length>]
                    [-v | -vv | -vvv | -vvvv]
                    [--quiet]
```

使用配置文件

您可以引用包含参数及其值的配置文件。这相当于在命令中键入参数和值。有关示例，请参阅如何在配置文件中存储参数。

```
aws-encryption-cli @<configuration_file>

# In a PowerShell console, use a backtick to escape the @.
aws-encryption-cli `@<configuration_file>
```

AWS 加密 CLI 命令行参数

此列表提供了 AWS 加密 CLI 命令参数的基本描述。有关完整说明，请参阅aws-encryption-sdk-cli文档。

--encrypt (-e)

加密输入数据。每个命令必须具有一个 `--encrypt` 或 `--decrypt` 或 `--decrypt-unsigned` 参数。

--decrypt (-d)

解密输入数据。每个命令必须具有一个 `--encrypt`、`--decrypt` 或 `--decrypt-unsigned` 参数。

--decrypt-unsigned [在版本 1.9.x 和 2.2.x 中引入。]

`--decrypt-unsigned` 参数对加密文字进行解密并确保消息在解密之前未签名。如果您使用 `--algorithm` 参数并选择了不带数字签名的算法套件来加密数据，请使用此参数。如果加密文字已签名，则解密失败。

您可以使用 `--decrypt` 或 `--decrypt-unsigned` 进行解密，但不能同时使用两者。

--wrapping-keys (-w) [在版本 1.8.x 中引入。]

指定在加密和解密操作中使用的包装密钥（或主密钥）。您可以在每个命令中使用多个 `--wrapping-keys` 参数。

从版本 2.1.x 开始，在加密和解密命令中需要使用 `--wrapping-keys` 参数。在版本 1.8.x 中，加密命令需要 `--wrapping-keys` 或 `--master-keys` 参数。在版本 1.8.x 解密命令中，`--wrapping-keys` 参数是可选的，但建议使用。

使用自定义主密钥提供程序时，加密和解密命令需要使用 key 和 provider 属性。使用时 AWS KMS keys，加密命令需要密钥属性。解密命令需要使用 key 属性或值为 `true` 的 discovery 属性（但不能两者同时使用）。解密时使用 key 属性是 AWS Encryption SDK 最佳实践。如果您要解密多批陌生消息，例如 Amazon S3 存储桶或 Amazon SQS 队列中的消息，这一点尤其重要。

有关展示如何使用 AWS KMS 多区域密钥作为包装密钥的示例，请参阅使用多区域 AWS KMS keys。

属性：`--wrapping-keys` 参数值包含以下属性。格式为 `attribute_name=value`。

键

标识操作中使用的包装密钥。格式为 key=ID 对。您可以在每个 `--wrapping-keys` 参数值中指定多个 key 属性。

- 加密命令：所有加密命令都需要使用 key 属性。在加密命令 AWS KMS key 中使用时，密钥属性的值可以是密钥 ID、密钥 ARN、别名或别名 ARN。有关 AWS KMS 密钥标识符的描述，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。
- 解密命令：使用 AWS KMS keys解密时，`--wrapping-keys` 参数需要使用值为密钥 ARN 的 key 属性或值为 `true` 的 discovery 属性（但不能两者同时使用）。使用 key 属性是 AWS Encryption SDK 最佳实践。使用自定义主密钥提供程序解密时，需要使用 key 属性。

> **Note**
> 要在解密命令中指定 AWS KMS 包装密钥，密钥属性的值必须是密钥 ARN。如果您使用密钥 ID、别名或别名 ARN，则加密 AWS CLI 无法识别包装密钥。

您可以在每个 `--wrapping-keys` 参数值中指定多个 key 属性。不过，`--wrapping-keys` 参数中的任何 provider、region 和 profile 属性适用于该参数值中的所有包装密钥。要指定具有不同属性值的包装密钥，请在命令中使用多个 `--wrapping-keys` 参数。

discovery

允许 AWS 加密 CLI 使用任何 AWS KMS key 方法来解密邮件。discovery 值可以是 `true` 或 `false`。默认值为 `false`。discovery 属性仅在加密命令中有效，并且仅在主密钥提供程序为 AWS KMS时有效。

使用解密时 AWS KMS keys，--wrapping-keys参数需要密钥属性或值为true（但不能两者兼而有之）的发现属性。如果您使用 key 属性，则可以使用值为 false 的 discovery 属性来明确拒绝发现。

- False（默认）— 当未指定发现属性或其值为时false，Encryption CLI 仅使用参数的密钥属性 AWS KMS keys 指定的内容来解密消息。 AWS --wrapping-keys如果在 discovery 为 false 时没有指定 key 属性，则解密命令将失败。此值支持加 AWS 密 CLI 最佳实践。
- True— 当发现属性的值为时true， AWS Encryption CLI 会 AWS KMS keys 从加密邮件中的元数据中获取，并使用这些元数据 AWS KMS keys 来解密邮件。值为的发现属性的true行为类似于 1.8 版之前的 AWS 加密 CLI 版本。 x 不允许您在解密时指定包装密钥。但是，您使用 any 的意图 AWS KMS key 是明确的。如果在 discovery 为 true 时指定了 key 属性，则解密命令将失败。

  该true值可能会导致 Encryption CLI AWS KMS keys 在不同的 AWS 账户 区域中使用 AWS KMS keys ，或者尝试使用用户无权使用的 AWS 加密 CLI。

当发现为发现时true，最佳做法是使用发现分区和发现账户属性将使用限制在您 AWS KMS keys 指定的范围内。 AWS 账户

discovery-account

将 AWS KMS keys 用于解密的限制为指定的。 AWS 账户此属性的唯一有效值是 AWS 账户 ID。

此属性是可选的，仅在发现属性设置为且 AWS KMS keys 指定了发现分区属性的解密命令中有效。true

每个发现账户属性只需要一个 AWS 账户 ID，但你可以在同一个参数中指定多个发现账户属性。--wrapping-keys在给定的 --wrapping-keys 参数中指定的所有账户都必须位于指定的 AWS 分区中。

discovery-partition

在 d iscovery-account 属性中为账户指定 AWS 分区。它的值必须是 AWS 分区，例如awsaws-cn、或aws-gov-cloud。有关更多信息，请参阅《AWS 一般参考》 中的 Amazon 资源名称。

当您使用 discovery-account 属性时，需要使用此属性。每个 --wrapping keys 参数中只能指定一个 discovery-partition 属性。要 AWS 账户 在多个分区中指定，请使用其他--wrapping-keys参数。

提供商

指定主密钥提供程序。格式为 provider=ID 对。默认值 aws-kms 表示。 AWS KMS只有当主密钥提供者不要求时，才需要此属性 AWS KMS。

region

标 AWS 区域 识一个 AWS KMS key。此属性仅对有效 AWS KMS keys。只有在密钥标识符未指定区域时，才会使用该属性，否则，将忽略该属性。使用它时，它会覆盖 AWS CLI 中名为 profile 的默认区域。

配置文件

标识已 AWS CLI 命名的配置文件。此属性仅对有效 AWS KMS keys。只有在密钥标识符未指定区域并且在命令中不包含 region 属性时，才会使用配置文件中的区域。

--input (-i)

指定要加密或解密的数据的位置。此参数为必需参数。该值可以是文件或目录的路径，也可以是文件名模式。如果通过管道将输入发送到命令 (stdin)，请使用 `-`。

如果输入不存在，该命令将成功完成，而不会显示错误或警告。

--recursive (-r, -R)

对输入目录及其子目录中的文件执行操作。在 `--input` 值为目录时，需要使用该参数。

--decode

解码 Base64 编码的输入。

如果要解密已加密并随后编码的消息，您必须在解密该消息之前对其进行解码。该参数为您执行此操作。

例如，如果在 encrypt 命令中使用 `--encode` 参数，请在相应的 decrypt 命令中使用 `--decode` 参数。您也可以在加密 Base64 编码的输入之前使用该参数对其进行解码。

--output (-o)

指定输出的目标。此参数为必需参数。该值可以是文件名、现有目录或 `-`，后者将输出写入到命令行 (stdout)。

如果指定的输出目录不存在，该命令将失败。如果输入包含子目录，则 AWS 加密 CLI 会在您指定的输出目录下重现子目录。

默认情况下， AWS 加密 CLI 会覆盖同名文件。要更改该行为，请使用 `--interactive` 或 `--no-overwrite` 参数。要禁止显示覆盖警告，请使用 `--quiet` 参数。

Note

如果覆盖输出文件的命令失败，则会删除输出文件。

--interactive

在覆盖文件之前提示。

--no-overwrite

不覆盖文件。相反，如果输出文件存在，则 AWS 加密 CLI 会跳过相应的输入。

--suffix

为 AWS 加密 CLI 创建的文件指定自定义文件名后缀。要指示没有后缀，请使用没有值的参数 (`--suffix`)。

默认情况下，在 `--output` 参数未指定文件名时，输出文件名与输入文件名相同并加上后缀。encrypt 命令的后缀为 `.encrypted`。decrypt 命令的后缀为 `.decrypted`。

--encode

将 Base64（二进制到文本）编码应用于输出。编码可以防止 shell 主机程序错误地解释输出文本中的非 ASCII 字符。

在将加密输出写入 stdout (`--output -`) 时使用此参数，尤其是在 PowerShell 控制台中，即使您将输出通过管道传输到另一个命令或将其保存在变量中。

--metadata-output

指定有关加密操作的元数据的位置。请输入路径和文件名。如果目录不存在，该命令将失败。要将元数据写入到命令行 (stdout) 中，请使用 `-`。

您无法在同一命令中将命令输出 (`--output`) 和元数据输出 (`--metadata-output`) 写入到 stdout。此外，如果 `--input` 或 `--output` 值为目录（没有文件名），您无法将元数据输出写入到同一目录或该目录的任何子目录中。

如果您指定现有文件，默认情况下，Encrypt AWS ion CLI 会将新的元数据记录附加到文件中的任何内容。通过使用该功能，您可以创建一个包含所有加密操作的元数据的文件。要覆盖现有文件中的内容，请使用 `--overwrite-metadata` 参数。

AWS 加密 CLI 会为该命令执行的每个加密或解密操作返回 JSON 格式的元数据记录。每个元数据记录包含输入和输出文件的完整路径、加密上下文、算法套件以及其他有价值的信息，您可以使用这些信息查看操作并验证它是否符合您的安全标准。

--overwrite-metadata

覆盖元数据输出文件中的内容。默认情况下，`--metadata-output` 参数将元数据附加到文件中的任何现有内容后面。

--suppress-metadata (-S)

禁止显示有关加密或解密操作的元数据。

--commitment-policy

指定加密和解密命令的承诺策略。承诺策略决定您的消息是否使用密钥承诺安全功能进行加密和解密。

`--commitment-policy` 参数在版本 1.8.x 中引入。该参数在加密和解密命令中有效。

在 1.8 版本中。 x， AWS 加密 CLI 对所有加密和解密操作使用`forbid-encrypt-allow-decrypt`承诺策略。当您在加密或解密命令中使用 `--wrapping-keys` 参数时，需要使用具有 `forbid-encrypt-allow-decrypt` 值的 `--commitment-policy` 参数。如果您不使用 `--wrapping-keys` 参数，则 `--commitment-policy` 参数无效。明确设置承诺策略可防止您的承诺策略在升级到版本 2.1.x 时自动更改为 `require-encrypt-require-decrypt`

从版本 2.1.x 开始，支持所有承诺策略值。`--commitment-policy` 参数是可选的，默认值为 `require-encrypt-require-decrypt`。

此参数具有以下值：

- `forbid-encrypt-allow-decrypt` – 无法使用密钥承诺进行加密。可以解密使用或不使用密钥承诺加密的加密文字。

  在版本 1.8.x 中，这是唯一的有效值。加 AWS 密 CLI 对所有加密和解密操作使用`forbid-encrypt-allow-decrypt`承诺策略。
- `require-encrypt-allow-decrypt` – 仅使用密钥承诺进行加密。使用和不使用密钥承诺进行解密。此值在版本 2.1.x 中引入。
- `require-encrypt-require-decrypt`（默认）– 仅使用密钥承诺进行加密和解密。此值在版本 2.1.x 中引入。在版本 2.1.x 和更高版本中，这是默认值。使用此值，Encryption CLI 将不会解密使用早期版本加密的任何密文。 AWS AWS Encryption SDK

有关设置承诺策略的详细信息，请参阅 迁移你的 AWS Encryption SDK。

--encryption-context (-c)

为操作指定加密上下文。该参数不是必需的，但建议使用。

- 在 `--encrypt` 命令中，输入一个或多个 `name=value` 对。请使用空格分隔这些对。
- 在 `--decrypt` 命令中，输入 `name=value` 对和/或没有值的 `name` 元素。

如果 `name` 对中的 `value` 或 `name=value` 包含空格或特殊字符，请将整个对用引号引起来。例如 `--encryption-context "department=software development"`。

--buffer (-b) [在版本 1.9.x 和 2.2.x 中引入。]

仅在处理完所有输入之后返回明文，包括验证数字签名（如果存在）。

--max-encrypted-data-keys [在 1.9 版本中引入。 x 和 2.2。 x]

指定加密消息中加密数据密钥的最大数量。此参数为可选的。

有效值为 1-65535。如果省略此参数，则 AWS 加密 CLI 不会强制执行任何最大值。加密消息最多可以容纳 65535（2^16-1）个加密数据密钥。

您可以在加密命令中使用此参数来防止出现格式错误的消息。您可以在解密命令中使用该参数检测恶意消息，并避免使用大量无法解密的加密数据密钥解密消息。有关详细信息和示例，请参阅限制加密数据密钥。

--help (-h)

在命令行中输出用法和语法。

--version

获取加 AWS 密 CLI 的版本。

-v | -vv | -vvv | -vvvv

显示详细信息、警告和调试消息。输出中的详细信息随参数中的 v 数量而增加。最详细的设置 (-vvvv) 返回来自加密 AWS CLI 及其使用的所有组件的调试级数据。

--quiet (-q)

禁止显示警告消息，例如，在覆盖输出文件时显示的消息。

--master-keys (-m) [已弃用]

Note

--master-keys 参数在版本 1.8.x 中弃用并在版本 2.1.x 中删除。请改用 --wrapping-keys 参数。

指定在加密和解密操作中使用的主密钥。您可以在每个命令中使用多个主密钥参数。

需要在 encrypt 命令中使用 `--master-keys` 参数。只有在使用自定义（非AWS KMS）主密钥提供程序时，才需要在解密命令中使用该参数。

属性：`--master-keys` 参数值包含以下属性。格式为 `attribute_name=value`。

键

标识操作中使用的包装密钥。格式为 key=ID 对。需要在所有 encrypt 命令中使用 key 属性。

在加密命令 AWS KMS key 中使用时，密钥属性的值可以是密钥 ID、密钥 ARN、别名或别名 ARN。有关 AWS KMS 密钥标识符的详细信息，请参阅《AWS Key Management Service 开发者指南》中的密钥标识符。

在主密钥提供程序不是 AWS KMS时，需要在解密命令中使用 key 属性。在解密根据 AWS KMS key加密的数据的命令中，不允许使用 key 属性。

您可以在每个 `--master-keys` 参数值中指定多个 key 属性。不过，任何 provider、region 和 profile 属性适用于该参数值中的所有主密钥。要指定具有不同属性值的主密钥，请在命令中使用多个 `--master-keys` 参数。

提供商

指定主密钥提供程序。格式为 provider=ID 对。默认值 aws-kms 表示。 AWS KMS只有当主密钥提供者不要求时，才需要此属性 AWS KMS。

region

标 AWS 区域 识一个 AWS KMS key。此属性仅对有效 AWS KMS keys。只有在密钥标识符未指定区域时，才会使用该属性，否则，将忽略该属性。使用它时，它会覆盖 AWS CLI 中名为 profile 的默认区域。

配置文件

标识已 AWS CLI 命名的配置文件。此属性仅对有效 AWS KMS keys。只有在密钥标识符未指定区域并且在命令中不包含 region 属性时，才会使用配置文件中的区域。

高级参数

--algorithm

指定备用的算法套件。该参数是可选的，仅在 encrypt 命令中有效。

如果省略此参数，则 AWS 加密 CLI 将使用 1.8 版中 AWS Encryption SDK 引入的默认算法套件之一。 x。两种默认算法都使用带有 HKDF、ECDSA 签名和 256 位加密密钥的 AES-GCM。一种算法使用密钥承诺；一种不使用。默认算法套件的选择由命令的承诺策略决定。

建议将默认算法套件用于大多数加密操作。有关有效值的列表，请参阅 Read the Docs 中 `algorithm` 参数的值。

--frame-length

创建具有指定帧长度的输出。该参数是可选的，仅在 encrypt 命令中有效。

请输入一个值（字节）。有效值为 0 和 1-2^31-1。值 0 表示非帧数据。默认值为 4096（字节）。

> **Note**
>
> 尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

--max-length

指示要从加密的消息中读取的最大帧大小（或非帧消息的最大内容长度），以字节为单位。该参数是可选的，仅在 decrypt 命令中有效。它旨在防止您解密非常大的恶意密文。

请输入一个值（字节）。如果省略此参数，则解密时 AWS Encryption SDK 不会限制帧大小。

--caching

启用数据密钥缓存功能，该功能重用数据密钥，而不是为每个输入文件生成新的数据密钥。该参数支持高级方案。在使用该功能之前，请务必阅读数据密钥缓存文档。

`--caching` 参数具有以下属性。

capacity（必需）

确定缓存中的最大条目数。

最小值为 1。没有最大值。

max_age（必需）

确定使用缓存条目的时间长度（秒），从将条目添加到缓存时算起。

请输入一个大于 0 的值。没有最大值。

max_messages_encrypted（可选）

确定缓存的条目可以加密的最大消息数。

有效值为 1-2^32。默认值为 2^32（消息）。

max_bytes_encrypted（可选）

确定缓存的条目可以加密的最大字节数。

有效值为 0 和 1-2^63-1。默认值为 2^63-1（消息）。在使用值 0 时，您只能在加密空消息字符串时使用数据密钥缓存。

AWS 加密 CLI 的版本

我们建议您使用最新版本的 AWS 加密 CLI。

> **Note**
>
> 4.0.0 之前的 AWS 加密 CLI 版本处于该阶段。end-of-support
> 您无需更改任何代码或数据即可安全地从 AWS Encryption CLI 版本 2.1.x 及更高版本更新为最新版本。但是，版本 2.1.x 中引入了新的安全功能，不向后兼容。从 1.7 版本更新。 x 或更早版本，必须先更新到最新的 1。 AWS 加密 CLI 的 x 版本。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

有关重要版本的信息 AWS Encryption SDK，请参见的版本 AWS Encryption SDK。

我使用哪个版本？

如果您不熟悉加 AWS 密 CLI，请使用最新版本。

解密由 1.7 AWS Encryption SDK 之前版本加密的数据。 x，首先迁移到最新版本的 AWS 加密 CLI。在更新到 2.1.x 或更高版本之前，请执行所有建议的更改。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

了解详情

- 有关更改的详细信息以及迁移到这些新版本的指南，请参阅 迁移你的 AWS Encryption SDK。
- 有关新的 AWS 加密 CLI 参数和属性的描述，请参阅AWS Encryption SDK CLI 语法和参数参考。

以下列表描述了 1.8 版本中对 AWS 加密 CLI 的更改。 x 和 2.1。 x。

版本 1.8。 AWS 加密 CLI 的 x 项更改

- 弃用 `--master-keys` 参数。请改用 `--wrapping-keys` 参数。
- 添加 `--wrapping-keys` (-w) 参数。支持 `--master-keys` 参数的所有属性。还添加了以下可选属性，这些属性仅在使用 AWS KMS keys解密时才有效。
  - discovery
  - discovery-partition
  - discovery-account

  对于自定义主密钥提供程序，`--encrypt` 和 `--decrypt` 命令需要使用 `--wrapping-keys` 参数或 `--master-keys` 参数（但不能两者同时使用）。此外，带的`--encrypt`命令 AWS KMS keys 需要`--wrapping-keys`参数或`--master-keys`参数（但不能两者兼而有之）。

  在带的`--decrypt`命令中 AWS KMS keys，`--wrapping-keys`参数是可选的，但建议使用，因为在 2.1 版本中它是必需的。 x。如果使用该参数，则必须指定 key 属性或值为 `true` 的 discovery 属性（但不能两者同时使用）。
- 添加 `--commitment-policy` 参数。唯一有效值为 `forbid-encrypt-allow-decrypt`。`forbid-encrypt-allow-decrypt` 承诺策略用于所有加密和解密命令。

  在版本 1.8.x 中，当您使用 `--wrapping-keys` 参数时，需要使用值为 `forbid-encrypt-allow-decrypt` 的 `--commitment-policy` 参数。明确设置该值可防止您的承诺策略在升级到版本 2.1.x 时自动更改为 `require-encrypt-require-decrypt`。

版本 2.1。 AWS 加密 CLI 的 x 项更改

- 移除 `--master-keys` 参数。请改用 `--wrapping-keys` 参数。
- 在所有加密和解密命令中需要使用 `--wrapping-keys` 参数。必须指定 key 属性或值为 `true` 的 discovery 属性（但不能两者同时使用）。
- `--commitment-policy` 参数支持以下值。有关更多信息，请参阅 设置您的承诺策略。
  - `forbid-encrypt-allow-decrypt`
  - `require-encrypt-allow-decrypt`
  - `require-encrypt-require decrypt`（默认值）
- 在版本 2.1.x 中，`--commitment-policy` 参数是可选的。默认值为 `require-encrypt-require-decrypt`。

版本 1.9。 x 和 2.2。 AWS 加密 CLI 的 x 项更改

- 添加 `--decrypt-unsigned` 参数。有关更多信息，请参阅 版本 2.2。x。
- 添加 `--buffer` 参数。有关更多信息，请参阅 版本 2.2。x。
- 添加 `--max-encrypted-data-keys` 参数。有关更多信息，请参阅 限制加密数据密钥。

版本 3.0。 AWS 加密 CLI 的 x 项更改

- 增加了对 AWS KMS 多区域密钥的支持。有关详细信息，请参阅使用多区域 AWS KMS keys。

数据密钥缓存

数据密钥缓存 将数据密钥和相关的加密材料存储在缓存中。加密或解密数据时，会在缓存中 AWS Encryption SDK 查找匹配的数据密钥。如果找到匹配项，它就使用缓存的数据密钥，而不是生成新的密钥。数据密钥缓存可以提高性能、降低成本，并且可以帮助您在应用程序扩展时保持在服务限制内。

在以下情况下，您的应用程序可以从数据密钥缓存中受益：

- 应用程序可以重用数据密钥。
- 应用程序生成大量数据密钥。
- 您的加密操作过于缓慢、成本高、受限制或消耗大量资源。

缓存可以减少您对加密服务的使用，例如 AWS Key Management Service (AWS KMS)。如果您已达到AWS KMS requests-per-second极限，缓存可以提供帮助。您的应用程序可以使用缓存的密钥来处理您的某些数据密钥请求，而不必调用 AWS KMS。（您还可以在 AWS Support Center 中创建一个案例以提高您账户的限制。）

可 AWS Encryption SDK 帮助您创建和管理数据密钥缓存。该工具包提供一个本地缓存和缓存加密材料管理器（缓存 CMM），以便与缓存交互并实施您设置的安全阈值。这些组件配合使用可以帮助您从重用数据密钥获得的高效率中受益，同时保持系统的安全性。

数据密钥缓存是的一项可选功能 AWS Encryption SDK ，您应谨慎使用。默认情况下，会为每个加密操作 AWS Encryption SDK 生成一个新的数据密钥。这种方法支持加密最佳实践，该最佳实践不建议过度重用数据密钥。通常，只有在需要满足性能目标时，才应使用数据密钥缓存。此外，还应使用数据密钥缓存安全阈值，以确保您使用满足成本和性能目标所需的最小缓存量。

版本 3。 x AWS Encryption SDK for Java 仅支持带有传统主密钥提供程序接口的缓存 CMM，不支持密钥环接口。但是，版本 4。 .NET 的 AWS Encryption SDK x 及更高版本，版本 3。 的 x AWS Encryption SDK for Java，版本 4。 的 x AWS Encryption SDK for Python，版本 1。 Rust 和 0.1 版本的 x。 AWS Encryption SDK x 或更高版本的 fo AWS Encryption SDK r Go 支持AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。使用 AWS KMS 分层密钥环加密的内容只能使用分层密钥环进行解 AWS KMS 密。

有关这些安全折衷方案的详细讨论，请参阅 AWS 安全博客中的 AWS Encryption SDK: How to Decide if Data Key Caching is Right for Your Application。

主题

- 如何使用数据密钥缓存
- 设置缓存安全阈值
- 数据密钥缓存详细信息
- 数据密钥缓存示例

如何使用数据密钥缓存

本主题介绍如何在您的应用程序中使用数据密钥缓存。它将指导您逐步完成该过程。然后，它将这些步骤合并到一个简单示例中，该示例在操作中使用数据密钥缓存以加密字符串。

本节中的示例说明了如何使用 AWS Encryption SDK版本 2.0.x 及更高版本。有关使用早期版本的示例，请在您的编程语言 GitHub 存储库的版本列表中找到您的版本。

有关在中使用数据密钥缓存的完整且经过测试的示例 AWS Encryption SDK，请参阅：

- C/C++：caching_cmm.cpp
- Java：SimpleDataKeyCachingExample. java
- JavaScript 浏览器：caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python：data_key_caching_basic.py

适用于 .NET 的AWS Encryption SDK 不支持数据密钥缓存。

主题

- 使用数据密钥缓存： Step-by-step
- 数据密钥缓存示例：加密字符串

使用数据密钥缓存： Step-by-step

这些 step-by-step说明向您展示了如何创建实现数据密钥缓存所需的组件。

- 创建数据密钥缓存。在这些示例中，我们使用 AWS Encryption SDK 提供的本地缓存。我们将缓存限制为 10 个数据密钥。

C

```
// Cache capacity (maximum number of entries) is required
size_t cache_capacity = 10;
struct aws_allocator *allocator = aws_default_allocator();

struct aws_cryptosdk_materials_cache *cache =
 aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);
```

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
// Cache capacity (maximum number of entries) is required
int MAX_CACHE_SIZE = 10;

CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(MAX_CACHE_SIZE);
```

JavaScript Browser

```
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

JavaScript Node.js

```
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

Python

```
# Cache capacity (maximum number of entries) is required
MAX_CACHE_SIZE = 10

cache = aws_encryption_sdk.LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
```

- 创建主密钥提供程序（Java 和 Python）或密钥环（C 和 JavaScript）。这些示例使用 AWS Key Management Service (AWS KMS) 主密钥提供程序或兼容的AWS KMS 密钥环。

C

```
// Create an AWS KMS keyring
//   The input is the Amazon Resource Name (ARN)
//   of an AWS KMS key
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);
```

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
// Create an AWS KMS master key provider
//   The input is the Amazon Resource Name (ARN)
//   of an AWS KMS key
MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn);
```

JavaScript Browser

在浏览器中，您必须安全地注入凭证。该示例在 webpack (kms.webpack.config) 中定义凭证，webpack 将在运行时解析凭证。它通过 AWS KMS 客户端和证书创建 AWS KMS 客户端提供程序实例。然后，当它创建密钥环时，它会将客户端提供者与 AWS KMS key (`generatorKeyId`)一起传递给构造函数。

```
const { accessKeyId, secretAccessKey, sessionToken } = credentials

const clientProvider = getClient(KMS, {
    credentials: {
      accessKeyId,
      secretAccessKey,
      sessionToken
    }
  })
```

```
/*  Create an AWS KMS keyring
 *  You must configure the AWS KMS keyring with at least one AWS KMS key
*  The input is the Amazon Resource Name (ARN)
 */ of an AWS KMS key
const keyring = new KmsKeyringBrowser({
    clientProvider,
    generatorKeyId,
    keyIds,
  })
```

JavaScript Node.js

```
/* Create an AWS KMS keyring
 *   The input is the Amazon Resource Name (ARN)
*/   of an AWS KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })
```

Python

```
# Create an AWS KMS master key provider
#  The input is the Amazon Resource Name (ARN)
#  of an AWS KMS key
key_provider =
 aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])
```

- 创建缓存加密材料管理器（缓存 CMM）。

将您的缓存 CMM 与缓存和主密钥提供程序或密钥环相关联。然后，在缓存 CMM 上设置缓存安全阈值。

C

在中 AWS Encryption SDK for C，您可以从底层 CMM（例如默认 CMM）或密钥环创建缓存 CMM。该示例从密钥环中创建缓存 CMM。

在创建缓存 CMM 后，您可以释放对密钥环和缓存的引用。有关更多信息，请参阅 the section called “引用计数”。

```
// Create the caching CMM
//   Set the partition ID to NULL.
//   Set the required maximum age value to 60 seconds.
struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL,
 60, AWS_TIMESTAMP_SECS);

// Add an optional message threshold
//   The cached data key will not be used for more than 10 messages.
aws_status = aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, 10);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);
```

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 的 x AWS Encryption SDK for Java 不支持数据密钥缓存，但它支持AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
/*
 * Security thresholds
 *   Max entry age is required.
 *   Max messages (and max bytes) per entry are optional
 */
int MAX_ENTRY_AGE_SECONDS = 60;
int MAX_ENTRY_MSGS = 10;

//Create a caching CMM
CryptoMaterialsManager cachingCmm =
    CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                                 .withCache(cache)
                                 .withMaxAge(MAX_ENTRY_AGE_SECONDS,
 TimeUnit.SECONDS)
                                 .withMessageUseLimit(MAX_ENTRY_MSGS)
                                 .build();
```

JavaScript Browser

```
/*
```

```
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new WebCryptoCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

JavaScript Node.js

```
/*
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new NodeCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

Python

```
# Security thresholds
#   Max entry age is required.
#   Max messages (and max bytes) per entry are optional
#
MAX_ENTRY_AGE_SECONDS = 60.0
MAX_ENTRY_MESSAGES = 10

# Create a caching CMM
```

```
caching_cmm = CachingCryptoMaterialsManager(
    master_key_provider=key_provider,
    cache=cache,
    max_age=MAX_ENTRY_AGE_SECONDS,
    max_messages_encrypted=MAX_ENTRY_MESSAGES
)
```

这是您需要执行的全部操作。然后，让他们为您 AWS Encryption SDK 管理缓存，或者添加您自己的缓存管理逻辑。

如果要在加密或解密数据的调用中使用数据密钥缓存，请指定您的缓存 CMM 而不是主密钥提供程序或其他 CMM。

> **Note**
>
> 如果要加密数据流或任何未知大小的数据，请务必在请求中指定数据大小。加密大小 AWS Encryption SDK 未知的数据时，不使用数据密钥缓存。

C

在中 AWS Encryption SDK for C，您可以创建与缓存 CMM 的会话，然后处理该会话。

默认情况下，当消息大小未知且不受限制时， AWS Encryption SDK 不缓存数据密钥。要允许在不知道确切数据大小时缓存，请使用 `aws_cryptosdk_session_set_message_bound` 方法设置消息的最大大小。设置大于估计消息大小的边界。如果实际消息大小超出边界，加密操作就会失败。

```
/* Create a session with the caching CMM. Set the session mode to encrypt. */
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 caching_cmm);

/* Set a message bound of 1000 bytes */
aws_status = aws_cryptosdk_session_set_message_bound(session, 1000);

/* Encrypt the message using the session with the caching CMM */
aws_status = aws_cryptosdk_session_process(
             session, output_buffer, output_capacity, &output_produced,
 input_buffer, input_len, &input_consumed);
```

```
/* Release your references to the caching CMM and the session. */
aws_cryptosdk_cmm_release(caching_cmm);
aws_cryptosdk_session_destroy(session);
```

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
// When the call to encryptData specifies a caching CMM,
// the encryption operation uses the data key cache
final AwsCrypto encryptionSdk = AwsCrypto.standard();
return encryptionSdk.encryptData(cachingCmm, plaintext_source).getResult();
```

JavaScript Browser

```
const { result } = await encrypt(cachingCmm, plaintext)
```

JavaScript Node.js

在 for Node.js 中使用缓存 CMM 时，该`encrypt`方法需要纯文本的长度。 AWS Encryption SDK for JavaScript 如果未提供，则不会缓存数据密钥。如果提供了长度，但提供的明文数据超过该长度，加密操作将会失败。如果您不知道明文的确切长度（例如，在流式传输数据时），请提供最大的预期值。

```
const { result } = await encrypt(cachingCmm, plaintext, { plaintextLength:
 plaintext.length })
```

Python

```
# Set up an encryption client
client = aws_encryption_sdk.EncryptionSDKClient()

# When the call to encrypt specifies a caching CMM,
# the encryption operation uses the data key cache
#
encrypted_message, header = client.encrypt(
    source=plaintext_source,
    materials_manager=caching_cmm
)
```

数据密钥缓存示例：加密字符串

在加密字符串时，该简单代码示例使用数据密钥缓存。它将step-by-step 过程中的代码组合成可以运行的测试代码。

该示例为 AWS KMS key创建本地缓存和主密钥提供程序或密钥环。然后，使用本地缓存和主密钥提供程序或密钥环创建一个具有相应安全阈值的缓存 CMM。在 Java 和 Python 中，加密请求指定缓存 CMM、要加密的明文数据以及加密上下文。在 C 中，缓存 CMM 是在会话中指定的，并向加密请求提供会话。

要运行这些示例，您需要提供 AWS KMS key的 Amazon 资源名称（ARN）。确保您有权使用 AWS KMS key 以生成数据密钥。

有关创建和使用数据密钥缓存的更详细真实示例，请参阅 数据密钥缓存示例代码。

C

```
/*
 * 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://aws.amazon.com/apache2.0/
 *
 * 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.
 */

#include <aws/cryptosdk/cache.h>
#include <aws/cryptosdk/cpp/kms_keyring.h>
#include <aws/cryptosdk/session.h>

void encrypt_with_caching(
    uint8_t *ciphertext,     // output will go here (assumes ciphertext_capacity
 bytes already allocated)
    size_t *ciphertext_len,  // length of output will go here
    size_t ciphertext_capacity,
    const char *kms_key_arn,
    int max_entry_age,
```

```
    int cache_capacity) {
    const uint64_t MAX_ENTRY_MSGS = 100;

    struct aws_allocator *allocator = aws_default_allocator();

    // Load error strings for debugging
    aws_cryptosdk_load_error_strings();

    // Create a keyring
    struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);

    // Create a cache
    struct aws_cryptosdk_materials_cache *cache =
 aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);

    // Create a caching CMM
    struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(
        allocator, cache, kms_keyring, NULL, max_entry_age, AWS_TIMESTAMP_SECS);
    if (!caching_cmm) abort();

    if (aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, MAX_ENTRY_MSGS))
 abort();

    // Create a session
    struct aws_cryptosdk_session *session =
        aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 caching_cmm);
    if (!session) abort();

    // Encryption context
    struct aws_hash_table *enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);
    if (!enc_ctx) abort();
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key, "purpose");
    AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value, "test");
    if (aws_hash_table_put(enc_ctx, enc_ctx_key, (void *)enc_ctx_value, NULL))
 abort();

    // Plaintext data to be encrypted
    const char *my_data = "My plaintext data";
    size_t my_data_len  = strlen(my_data);
    if (aws_cryptosdk_session_set_message_size(session, my_data_len)) abort();
```

```
    // When the session uses a caching CMM, the encryption operation uses the data
 key cache
    // specified in the caching CMM.
    size_t bytes_read;
    if (aws_cryptosdk_session_process(
            session,
            ciphertext,
            ciphertext_capacity,
            ciphertext_len,
            (const uint8_t *)my_data,
            my_data_len,
            &bytes_read))
        abort();
    if (!aws_cryptosdk_session_is_done(session) || bytes_read != my_data_len)
 abort();

    aws_cryptosdk_session_destroy(session);
    aws_cryptosdk_cmm_release(caching_cmm);
    aws_cryptosdk_materials_cache_release(cache);
    aws_cryptosdk_keyring_release(kms_keyring);
}
```

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.examples;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoMaterialsManager;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.CryptoMaterialsCache;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import java.nio.charset.StandardCharsets;
```

```
import java.util.Collections;
import java.util.Map;
import java.util.concurrent.TimeUnit;

/**
 * <p>
 * Encrypts a string using an &KMS; key and data key caching
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>KMS Key ARN: To find the Amazon Resource Name of your &KMS; key,
 *     see 'Find the key ID and ARN' at https://docs.aws.amazon.com/kms/latest/
developerguide/find-cmk-id-arn.html
 * <li>Max entry age: Maximum time (in seconds) that a cached entry can be used
 * <li>Cache capacity: Maximum number of entries in the cache
 * </ol>
 */
public class SimpleDataKeyCachingExample {

    /*
     * Security thresholds
     *   Max entry age is required.
     *   Max messages (and max bytes) per data key are optional
     */
    private static final int MAX_ENTRY_MSGS = 100;

    public static byte[] encryptWithCaching(String kmsKeyArn, int maxEntryAge, int
 cacheCapacity) {
        // Plaintext data to be encrypted
        byte[] myData = "My plaintext data".getBytes(StandardCharsets.UTF_8);

        // Encryption context
        // Most encrypted data should have an associated encryption context
        // to protect integrity. This sample uses placeholder values.
        // For more information see:
        // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-
Integrity-of-Your-Encrypted-Data-by-Using-AWS-Key-Management
        final Map<String, String> encryptionContext =
 Collections.singletonMap("purpose", "test");

        // Create a master key provider
        MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder()
```

```
            .buildStrict(kmsKeyArn);

        // Create a cache
        CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(cacheCapacity);

        // Create a caching CMM
        CryptoMaterialsManager cachingCmm =

 CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                .withCache(cache)
                .withMaxAge(maxEntryAge, TimeUnit.SECONDS)
                .withMessageUseLimit(MAX_ENTRY_MSGS)
                .build();

        // When the call to encryptData specifies a caching CMM,
        // the encryption operation uses the data key cache
        final AwsCrypto encryptionSdk = AwsCrypto.standard();
        return encryptionSdk.encryptData(cachingCmm, myData,
 encryptionContext).getResult();
    }
}
```

JavaScript Browser

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

/* This is a simple example of using a caching CMM with a KMS keyring
 * to encrypt and decrypt using the AWS Encryption SDK for Javascript in a browser.
 */

import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
  WebCryptoCachingMaterialsManager,
  getLocalCryptographicMaterialsCache,
} from '@aws-crypto/client-browser'
import { toBase64 } from '@aws-sdk/util-base64-browser'
```

```
/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* This is injected by webpack.
 * The webpack.DefinePlugin or @aws-sdk/karma-credential-loader will replace the
 values when bundling.
 * The credential values are pulled from @aws-sdk/credential-provider-node
 * Use any method you like to get credentials into the browser.
 * See kms.webpack.config
 */
declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* This is done to facilitate testing. */
export async function testCachingCMMExample() {
  /* This example uses an &KMS; keyring. The generator key in a &KMS; keyring
 generates and encrypts the data key.
   * The caller needs kms:GenerateDataKey permission on the &KMS; key in
 generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding additional KMS keys that can decrypt.
   * The caller must have kms:Encrypt permission for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
```

```
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
   */
  const keyIds = [
    'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
  ]

  /* Need a client provider that will inject correct credentials.
   * The credentials here are injected by webpack from your environment bundle is
 created
   * The credential values are pulled using @aws-sdk/credential-provider-node.
   * See kms.webpack.config
   * You should inject your credential into the browser in a secure manner
   * that works with your application.
   */
  const { accessKeyId, secretAccessKey, sessionToken } = credentials

  /* getClient takes a KMS client constructor
   * and optional configuration values.
   * The credentials can be injected here,
   * because browsers do not have a standard credential discovery process the way
 Node.js does.
   */
  const clientProvider = getClient(KMS, {
    credentials: {
      accessKeyId,
      secretAccessKey,
      sessionToken,
    },
  })

  /* You must configure the KMS keyring with your &KMS; keys */
  const keyring = new KmsKeyringBrowser({
    clientProvider,
    generatorKeyId,
    keyIds,
  })

  /* Create a cache to hold the data keys (and related cryptographic material).
   * This example uses the local cache provided by the Encryption SDK.
   * The `capacity` value represents the maximum number of entries
   * that the cache can hold.
   * To make room for an additional entry,
```

```
  * the cache evicts the oldest cached entry.
  * Both encrypt and decrypt requests count independently towards this threshold.
  * Entries that exceed any cache threshold are actively removed from the cache.
  * By default, the SDK checks one item in the cache every 60 seconds (60,000
milliseconds).
  * To change this frequency, pass in a `proactiveFrequency` value
  * as the second parameter. This value is in milliseconds.
  */
 const capacity = 100
 const cache = getLocalCryptographicMaterialsCache(capacity)

 /* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
  * By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
  * use the same partition name for both caching CMMs.
  * If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
  * As a result, sharing elements in the cache MUST be an intentional operation.
  */
 const partition = 'local partition name'

 /* maxAge is the time in milliseconds that an entry will be cached.
  * Elements are actively removed from the cache.
  */
 const maxAge = 1000 * 60

 /* The maximum number of bytes that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxBytesEncrypted = 100

 /* The maximum number of messages that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxMessagesEncrypted = 10

 const cachingCMM = new WebCryptoCachingMaterialsManager({
   backingMaterials: keyring,
   cache,
   partition,
   maxAge,
```

```
    maxBytesEncrypted,
    maxMessagesEncrypted,
  })

  /* Encryption context is a *very* powerful tool for controlling
   * and managing access.
   * When you pass an encryption context to the encrypt function,
   * the encryption context is cryptographically bound to the ciphertext.
   * If you don't pass in the same encryption context when decrypting,
   * the decrypt function fails.
   * The encryption context is ***not*** secret!
   * Encrypted data is opaque.
   * You can use an encryption context to assert things about the encrypted data.
   * The encryption context helps you to determine
   * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
   * For example, if you are are only expecting data from 'us-west-2',
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt. */
  const plainText = new Uint8Array([1, 2, 3, 4, 5])

  /* Encrypt the data.
   * The caching CMM only reuses data keys
   * when it know the length (or an estimate) of the plaintext.
   * However, in the browser,
   * you must provide all of the plaintext to the encrypt function.
   * Therefore, the encrypt function in the browser knows the length of the
 plaintext
   * and does not accept a plaintextLength option.
   */
```

```
  const { result } = await encrypt(cachingCMM, plainText, { encryptionContext })

  /* Log the plain text
   * only for testing and to show that it works.
   */
  console.log('plainText:', plainText)
  document.write('</br>plainText:' + plainText + '</br>')

  /* Log the base64-encoded result
   * so that you can try decrypting it with another AWS Encryption SDK
 implementation.
   */
  const resultBase64 = toBase64(result)
  console.log(resultBase64)
  document.write(resultBase64)

  /* Decrypt the data.
   * NOTE: This decrypt request will not use the data key
   * that was cached during the encrypt operation.
   * Data keys for encrypt and decrypt operations are cached separately.
   */
  const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

  /* Grab the encryption context so you can verify it. */
  const { encryptionContext: decryptedContext } = messageHeader

  /* Verify the encryption context.
   * If you use an algorithm suite with signing,
   * the Encryption SDK adds a name-value pair to the encryption context that
 contains the public key.
   * Because the encryption context might contain additional key-value pairs,
   * do not include a test that requires that all key-value pairs match.
   * Instead, verify that the key-value pairs that you supplied to the `encrypt`
 function are included in the encryption context that the `decrypt` function
 returns.
   */
  Object.entries(encryptionContext).forEach(([key, value]) => {
    if (decryptedContext[key] !== value)
      throw new Error('Encryption Context does not match expected values')
  })

  /* Log the clear message
   * only for testing and to show that it works.
   */
```

```
  document.write('</br>Decrypted:' + plaintext)
  console.log(plaintext)

  /* Return the values to make testing easy. */
  return { plainText, plaintext }
}
```

JavaScript Node.js

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
  NodeCachingMaterialsManager,
  getLocalCryptographicMaterialsCache,
} from '@aws-crypto/client-node'

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

export async function cachingCMMNodeSimpleTest() {
  /* An &KMS; key is required to generate the data key.
   * You need kms:GenerateDataKey permission on the &KMS; key in generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding alternate &KMS; keys that can decrypt.
   * Access to kms:Encrypt is required for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
```

```
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
   */
  const keyIds = [
    'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
  ]

  /* The &KMS; keyring must be configured with the desired &KMS; keys
   * This example passes the keyring to the caching CMM
   * instead of using it directly.
   */
  const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

  /* Create a cache to hold the data keys (and related cryptographic material).
   * This example uses the local cache provided by the Encryption SDK.
   * The `capacity` value represents the maximum number of entries
   * that the cache can hold.
   * To make room for an additional entry,
   * the cache evicts the oldest cached entry.
   * Both encrypt and decrypt requests count independently towards this threshold.
   * Entries that exceed any cache threshold are actively removed from the cache.
   * By default, the SDK checks one item in the cache every 60 seconds (60,000
 milliseconds).
   * To change this frequency, pass in a `proactiveFrequency` value
   * as the second parameter. This value is in milliseconds.
   */
  const capacity = 100
  const cache = getLocalCryptographicMaterialsCache(capacity)

  /* The partition name lets multiple caching CMMs share the same local
 cryptographic cache.
   * By default, the entries for each CMM are cached separately. However, if you
 want these CMMs to share the cache,
   * use the same partition name for both caching CMMs.
   * If you don't supply a partition name, the Encryption SDK generates a random
 name for each caching CMM.
   * As a result, sharing elements in the cache MUST be an intentional operation.
   */
```

```
  const partition = 'local partition name'

  /* maxAge is the time in milliseconds that an entry will be cached.
   * Elements are actively removed from the cache.
   */
  const maxAge = 1000 * 60

  /* The maximum amount of bytes that will be encrypted under a single data key.
   * This value is optional,
   * but you should configure the lowest value possible.
   */
  const maxBytesEncrypted = 100

  /* The maximum number of messages that will be encrypted under a single data key.
   * This value is optional,
   * but you should configure the lowest value possible.
   */
  const maxMessagesEncrypted = 10

  const cachingCMM = new NodeCachingMaterialsManager({
    backingMaterials: keyring,
    cache,
    partition,
    maxAge,
    maxBytesEncrypted,
    maxMessagesEncrypted,
  })

  /* Encryption context is a *very* powerful tool for controlling
   * and managing access.
   * When you pass an encryption context to the encrypt function,
   * the encryption context is cryptographically bound to the ciphertext.
   * If you don't pass in the same encryption context when decrypting,
   * the decrypt function fails.
   * The encryption context is ***not*** secret!
   * Encrypted data is opaque.
   * You can use an encryption context to assert things about the encrypted data.
   * The encryption context helps you to determine
   * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
   * For example, if you are are only expecting data from 'us-west-2',
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
```

```
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt.  A simple string. */
  const cleartext = 'asdf'

  /* Encrypt the data.
   * The caching CMM only reuses data keys
   * when it know the length (or an estimate) of the plaintext.
   * If you do not know the length,
   * because the data is a stream
   * provide an estimate of the largest expected value.
   *
   * If your estimate is smaller than the actual plaintext length
   * the AWS Encryption SDK will throw an exception.
   *
   * If the plaintext is not a stream,
   * the AWS Encryption SDK uses the actual plaintext length
   * instead of any length you provide.
   */
  const { result } = await encrypt(cachingCMM, cleartext, {
    encryptionContext,
    plaintextLength: 4,
  })

  /* Decrypt the data.
   * NOTE: This decrypt request will not use the data key
   * that was cached during the encrypt operation.
   * Data keys for encrypt and decrypt operations are cached separately.
   */
  const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

  /* Grab the encryption context so you can verify it. */
  const { encryptionContext: decryptedContext } = messageHeader
```

```
  /* Verify the encryption context.
   * If you use an algorithm suite with signing,
   * the Encryption SDK adds a name-value pair to the encryption context that
 contains the public key.
   * Because the encryption context might contain additional key-value pairs,
   * do not include a test that requires that all key-value pairs match.
   * Instead, verify that the key-value pairs that you supplied to the `encrypt`
 function are included in the encryption context that the `decrypt` function
 returns.
   */
  Object.entries(encryptionContext).forEach(([key, value]) => {
    if (decryptedContext[key] !== value)
      throw new Error('Encryption Context does not match expected values')
  })

  /* Return the values so the code can be tested. */
  return { plaintext, result, cleartext, messageHeader }
}
```

Python

```
# Copyright 2017 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://aws.amazon.com/apache2.0/
#
# 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.
"""Example of encryption with data key caching."""
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_with_caching(kms_key_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an &KMS; key and data key caching.

    :param str kms_key_arn: Amazon Resource Name (ARN) of the &KMS; key
```

```
    :param float max_age_in_cache: Maximum time in seconds that a cached entry can
be used
    :param int cache_capacity: Maximum number of entries to retain in cache at once
    """
    # Data to be encrypted
    my_data = "My plaintext data"

    # Security thresholds
    #   Max messages (or max bytes per) data key are optional
    MAX_ENTRY_MESSAGES = 100

    # Create an encryption context
    encryption_context = {"purpose": "test"}

    # Set up an encryption client with an explicit commitment policy. Note that if
you do not explicitly choose a
    # commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
    client =
aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R

    # Create a master key provider for the &KMS; key
    key_provider =
aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])

    # Create a local cache
    cache = aws_encryption_sdk.LocalCryptoMaterialsCache(cache_capacity)

    # Create a caching CMM
    caching_cmm = aws_encryption_sdk.CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=max_age_in_cache,
        max_messages_encrypted=MAX_ENTRY_MESSAGES,
    )

    # When the call to encrypt data specifies a caching CMM,
    # the encryption operation uses the data key cache specified
    # in the caching CMM
    encrypted_message, _header = client.encrypt(
        source=my_data, materials_manager=caching_cmm,
encryption_context=encryption_context
    )
```

```
    return encrypted_message
```

设置缓存安全阈值

在实施数据密钥缓存时，您需要配置缓存 CMM 强制执行的安全阈值。

安全阈值有助于限制每个缓存的数据密钥的使用时间长度，以及每个数据密钥保护的数据量。只有在缓存条目符合所有安全阈值时，缓存 CMM 才会返回缓存的数据密钥。如果缓存条目超过任何阈值，则不会在当前操作中使用该条目，并会尽快将其从缓存中逐出。每个数据密钥的第一次使用（在缓存之前）都不计入这些阈值内。

一般来说，请使用满足您的成本和性能目标所需的最低缓存量。

AWS Encryption SDK 仅缓存使用密钥派生函数加密的数据密钥。此外，它还为某些阈值设置了上限。这些限制确保数据密钥的重用次数不会超过其加密限制。不过，由于缓存的是明文数据密钥（默认在内存中），请尽量缩短保存这些密钥的时间。此外，还要尽量限制在密钥泄露时可能会泄露的数据。

有关设置缓存安全阈值的示例，请参AWS Encryption SDK阅 “ AWS 安全博客” 中的 “如何确定数据密钥缓存是否适合您的应用程序”。

> Note
>
> 缓存 CMM 实施所有以下阈值。如果未指定可选的值，则缓存 CMM 使用默认值。
> 要临时禁用数据密钥缓存， AWS Encryption SDK 的 Java 和 Python 实现提供一个空加密材料缓存（空缓存）。空缓存为每个 GET 请求返回未命中，并且不响应 PUT 请求。建议您使用空缓存，而不是将缓存容量或安全阈值设置为 0。有关更多信息，请参阅 Java 和 Python 中的空缓存。

最长使用期限（必需）

确定缓存的条目的使用时长，从添加该条目时算起。该值为必填项。请输入一个大于 0 的值。AWS Encryption SDK 不限制最大年龄值。

的所有语言实现都 AWS Encryption SDK 定义了以秒为单位的最大年龄，但使用毫秒的 AWS Encryption SDK for JavaScript除外。

请尽可能使用最短的时间间隔，但前提是您的应用程序仍能从缓存中受益。您可以像密钥轮换策略一样使用最长使用期限阈值。可以使用该值限制数据密钥重用次数，最大限度减少加密材料泄露，以及逐出在缓存期间策略可能已发生变化的数据密钥。

加密的最大消息数（可选）

指定缓存的数据密钥可以加密的最大消息数。该值为可选项。请输入 1 到 2^32 之间的值。默认值为 2^32 个消息。

将每个缓存的密钥保护的消息数设置得足够大可以从重用中受益，但设置得足够小则可以限制在密钥泄露时可能会泄露的消息数。

加密的最大字节数（可选）

指定缓存的数据密钥可以加密的最大字节数。该值为可选项。请输入 0 到 2^63 - 1 之间的值。默认值为 2^63 - 1。在使用值 0 时，您只能在加密空消息字符串时使用数据密钥缓存。

在评估该阈值时，将包括当前请求中的字节数。如果已处理的字节数加上当前的字节数超过该阈值，即便是可能用于较小的请求，缓存的数据密钥也会从缓存中被逐出。

数据密钥缓存详细信息

大多数应用程序可以使用默认数据密钥缓存实施，而无需编写自定义代码。本节介绍了默认实施以及有关选项的一些详细信息。

主题

- 数据密钥缓存的工作方式
- 创建加密材料缓存
- 创建缓存加密材料管理器
- 在数据密钥缓存条目中包含哪些内容？
- 加密上下文：如何选择缓存条目
- 我的应用程序是否使用缓存的数据密钥？

数据密钥缓存的工作方式

在加密或解密数据的请求中使用数据密钥缓存时， AWS Encryption SDK 先在缓存中搜索与请求匹配的数据密钥。如果找到有效的匹配项，它使用缓存的数据密钥加密数据。否则，它生成新的数据密钥，就像没有缓存一样。

不会在未知大小的数据中使用数据密钥缓存，如流式数据。这样缓存 CMM 就可以正确强制执行最大字节数阈值。为了避免该行为，请将消息大小添加到加密请求中。

除缓存外，数据密钥缓存还使用缓存加密材料管理器（缓存 CMM）。缓存 CMM 是一种专用的加密材料管理器（CMM），该管理器与缓存和基础 CMM 进行交互。（在指定主密钥提供程序或密钥环时，AWS Encryption SDK 将创建一个默认 CMM。）缓存 CMM 缓存其基础 CMM 返回的数据密钥。缓存 CMM 还强制执行您设置的缓存安全阈值。

为了防止从缓存中选择错误的数据密钥，所有兼容的缓存 CMM 都要求缓存的加密材料的以下属性与材料请求相匹配。

- 算法套件
- 加密上下文（甚至为空时）
- 分区名称（用于标识缓存 CMM 的字符串）
- （仅解密）加密数据密钥

> **Note**
>
> 仅当算法套件使用密钥派生函数时，才会 AWS Encryption SDK 缓存数据密钥。

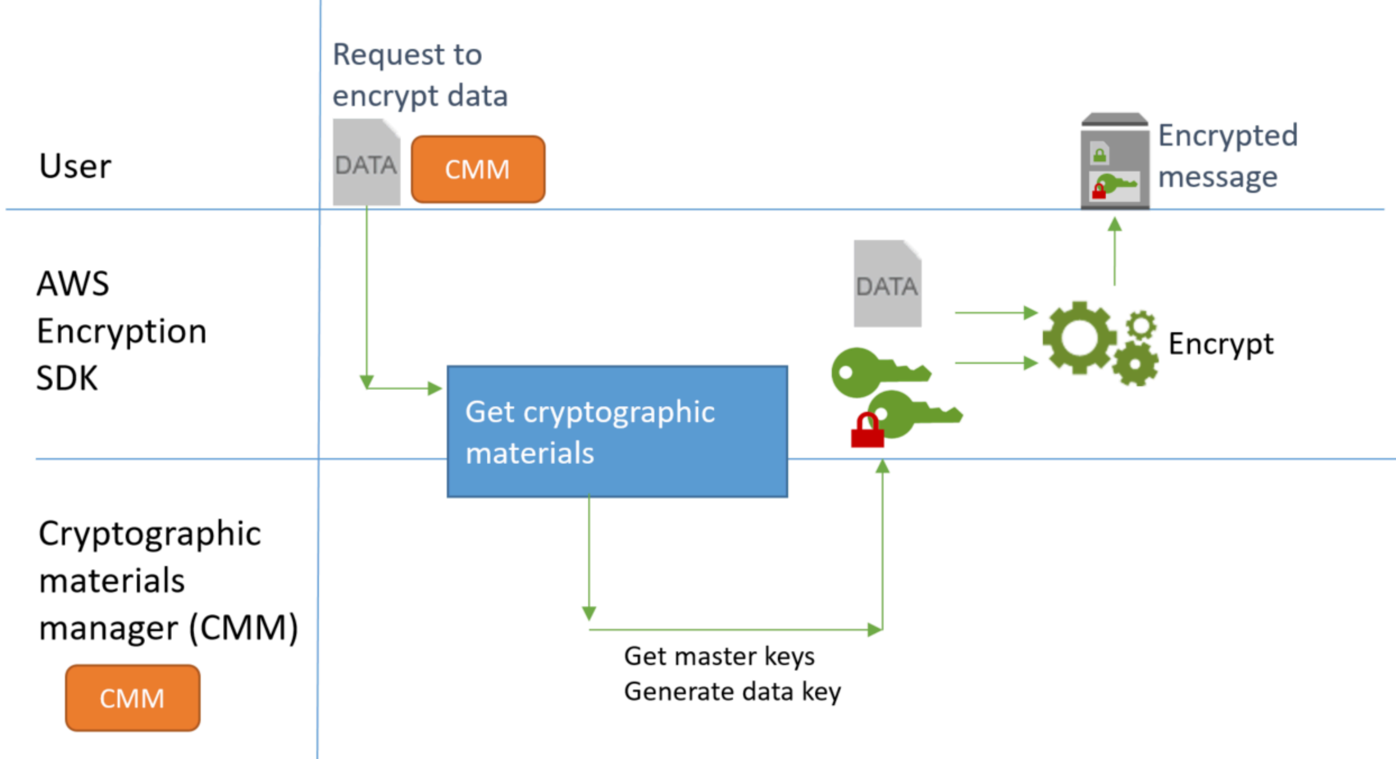
以下工作流介绍了如何在缓存和不缓存数据密钥的情况下处理数据加密请求。这些工作流说明了如何在该过程中使用您创建的缓存组件，包括缓存和缓存 CMM。

加密数据而不进行缓存

获取加密材料而不缓存：

1. 应用程序要求对数据 AWS Encryption SDK 进行加密。

   该请求指定一个主密钥提供程序或密钥环。 AWS Encryption SDK 创建一个与主密钥提供程序或密钥环交互的默认 CMM。
2. 向 CMM AWS Encryption SDK 索要加密材料（获取加密材料）。
3. CMM 要求其密钥环（C 和 JavaScript）或主密钥提供者（Java 和 Python）提供加密材料。这可能涉及调用加密服务，例如 AWS Key Management Service (AWS KMS)。CMM 将加密材料返回给 AWS Encryption SDK。
4. AWS Encryption SDK 使用纯文本数据密钥对数据进行加密。它在返回给用户的加密消息中存储加密数据和加密数据密钥。



使用缓存加密数据

获取加密材料并且缓存数据密钥：

1. 应用程序要求对数据 AWS Encryption SDK 进行加密。

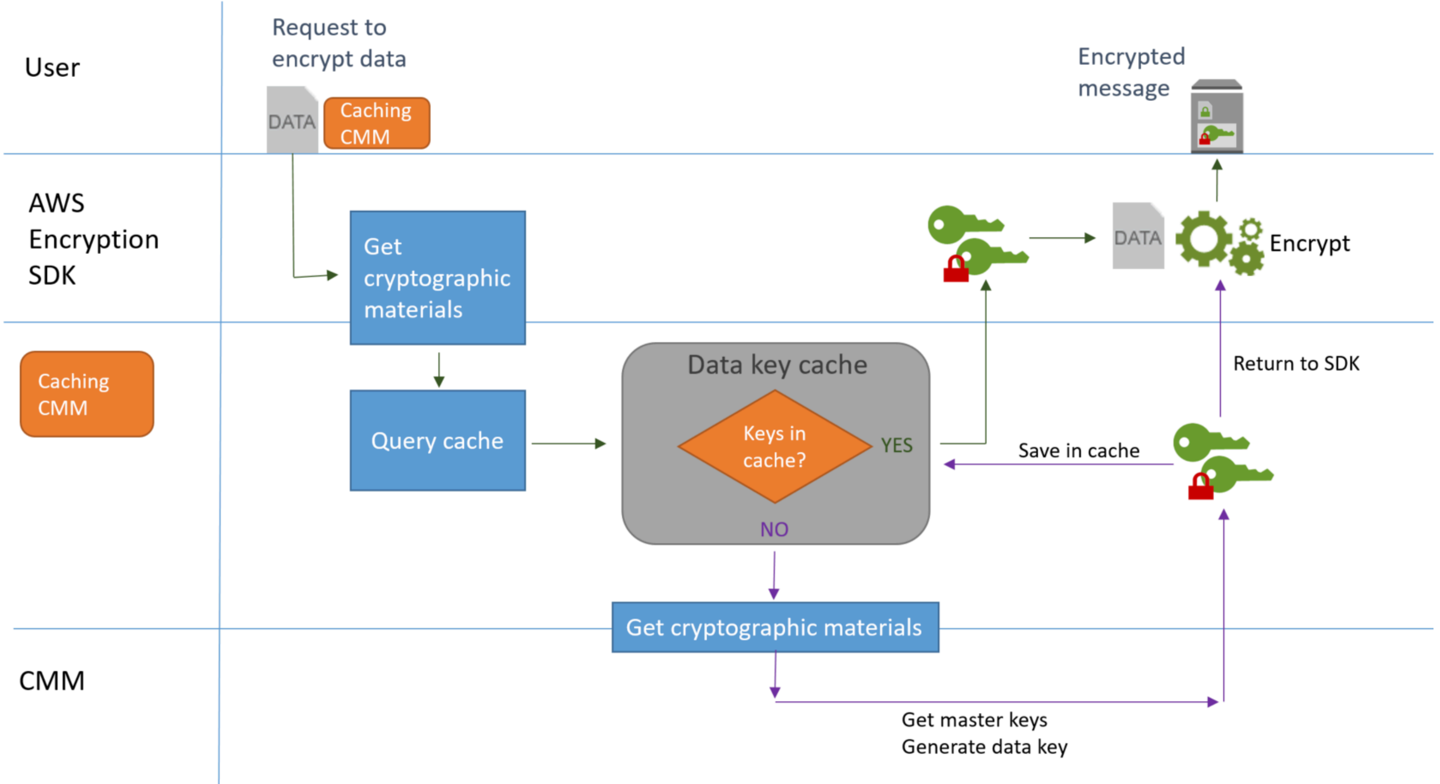
   该请求指定了与底层加密材料管理器（CMM）关联的缓存加密材料管理器（缓存 CMM）。如果您指定主密钥提供程序或密钥环， AWS Encryption SDK 将创建默认的 CMM。
2. 该开发工具包要求指定的缓存 CMM 提供加密材料。
3. 缓存 CMM 从缓存中请求加密材料。
   a. 如果缓存找到匹配项，将更新匹配的缓存条目的期限和使用值，并将缓存的加密材料返回给缓存 CMM。

      如果缓存条目符合其安全阈值，则缓存 CMM 将此条目返回到该开发工具包。否则，它通知缓存逐出该条目并继续操作，就好像没有匹配项一样。
   b. 如果缓存找不到有效的匹配项，缓存 CMM 将请求其基础 CMM 生成新的数据密钥。

      底层 CMM 从其密钥环（C 和 JavaScript）或主密钥提供程序（Java 和 Python）获取加密材料。这可能涉及调用一个服务，如 AWS Key Management Service。基础 CMM 将数据密钥的明文和加密副本返回给缓存 CMM。

缓存 CMM 在缓存中保存新的加密材料。

4. 缓存 CMM 将加密材料返回给 AWS Encryption SDK。
5. AWS Encryption SDK 使用纯文本数据密钥对数据进行加密。它在返回给用户的加密消息中存储加密数据和加密数据密钥。



创建加密材料缓存

AWS Encryption SDK 定义了数据密钥缓存中使用的加密材料缓存的要求。此外，缓存 CMM 还提供本地缓存，即可在内存中配置的最近最少使用（LRU）缓存。要创建本地缓存的实例，请使用 Java 和 Python 中的`LocalCryptoMaterialsCache`构造函数 JavaScript、中的 get LocalCryptographicMaterialsCache 函数或 C 中的`aws_cryptosdk_materials_cache_local_new`构造函数

本地缓存包含基本缓存管理逻辑，包括添加、驱逐和匹配缓存的条目以及维护缓存。您无需编写任何自定义缓存管理逻辑。您可以按原样使用本地缓存，对其进行自定义或替换任何兼容的缓存。

在创建本地缓存时，您可以设置其容量，即，该缓存可以保存的最大条目数。该设置有助于设计具有有限数据密钥重用次数的高效缓存。

AWS Encryption SDK for Java 和 AWS Encryption SDK for Python 还提供了一个空的加密材料缓存 (NullCryptoMaterialsCache)。会 NullCryptoMaterialsCache 返回所有GET操作的失误，并且不对PUT操作做出响应。可以在测试 NullCryptoMaterialsCache 中使用，也可以在包含缓存代码的应用程序中暂时禁用缓存。

在中 AWS Encryption SDK，每个加密材料缓存都与缓存加密材料管理器（缓存 CMM）相关联。缓存 CMM 从缓存中获取数据密钥，将数据密钥放在缓存中，然后强制执行您设置的安全阈值。在创建缓存 CMM 时，您可以指定其使用的缓存以及生成其缓存的数据密钥的基础 CMM 或主密钥提供程序。

创建缓存加密材料管理器

要启用数据密钥缓存，您需要创建缓存和缓存加密材料管理器（缓存 CMM）。然后，在加密或解密数据的请求中指定缓存 CMM，而不是标准加密材料管理器（CMM）或主密钥提供程序或密钥环。

共有两种类型的 CMM。它们均获取数据密钥（和相关的加密材料），但使用不同的方法，如下所示：

- CMM 与密钥环（C 或 JavaScript）或主密钥提供程序（Java 和 Python）相关联。当开发工具包要求 CMM 提供加密或解密材料时，CMM 从其密钥环或主密钥提供程序获取材料。在 Java 和 Python 中，CMM 使用主密钥生成、加密或解密数据密钥。在 C 和中 JavaScript，密钥环生成、加密和返回加密材料。
- 缓存 CMM 与一个缓存（例如本地缓存）和基础 CMM 相关联。在该开发工具包请求缓存 CMM 提供加密材料时，缓存 CMM 尝试从缓存中获取这些材料。如果找不到匹配项，缓存 CMM 将请求其基础 CMM 提供材料。然后，它在向调用方返回新的加密材料之前将其缓存。

缓存 CMM 还强制执行您为每个缓存条目设置的安全阈值。由于安全阈值是在缓存 CMM 中设置并由其强制执行的，所以您可以使用任何兼容的缓存，即使该缓存不是为敏感材料设计的。

在数据密钥缓存条目中包含哪些内容？

数据密钥缓存将数据密钥和相关的加密材料存储在缓存中。每个条目包含下面列出的元素。加密和解密有单独的缓存，因此加密消息不会预热用于解密的缓存条目。在决定是否使用数据密钥缓存功能以及在缓存加密材料管理器（缓存 CMM）上设置安全阈值时，您可能会发现该信息非常有用。

为加密请求缓存的条目

由于加密操作而添加到数据密钥缓存的条目包括以下元素：

- 明文数据密钥
- 加密的数据密钥（一个或多个）

- 加密上下文
- 消息签名密钥（如果使用）
- 算法套件
- 元数据，包括用于实施安全阈值的使用计数器

为解密请求缓存的条目

由于解密操作而添加到数据密钥缓存的条目包括以下元素：

- 明文数据密钥
- 签名验证密钥（如果使用）
- 元数据，包括用于实施安全阈值的使用计数器

加密上下文：如何选择缓存条目

您可以在任何加密数据的请求中指定加密上下文。不过，加密上下文在数据密钥缓存中起到特殊的作用。该上下文允许在您的缓存中创建数据密钥子组，即使数据密钥来自于相同的缓存 CMM。

加密上下文是一组包含任意非机密数据的键值对。在加密期间，加密上下文以加密方式绑定到加密的数据，以便需要使用相同的加密上下文解密数据。在中 AWS Encryption SDK，加密上下文存储在带有加密数据和数据密钥的加密消息中。

在使用数据密钥缓存时，您还可以使用加密上下文为加密操作选择特定的缓存数据密钥。加密上下文与数据密钥一起保存在缓存条目中（它是缓存条目 ID 的一部分）。只有在加密上下文匹配时，才会重用缓存的数据密钥。如果要在加密请求中重用某些数据密钥，请指定相同的加密上下文。如果要避免使用这些数据密钥，请指定不同的加密上下文。

加密上下文始终是可选的，但建议使用。如果在请求中未指定加密上下文，则在缓存条目标识符中包含空加密上下文并与每个请求匹配。

我的应用程序是否使用缓存的数据密钥？

数据密钥缓存是一种优化策略，对于某些应用程序和工作负载是非常有效的。不过，由于它存在一些风险，请务必确定它对您的环境可能有多有效，然后确定收益是否大于风险。

由于数据密钥缓存重复使用数据密钥，因此，最明显的效果是减少了生成新数据密钥的调用次数。当实现数据密钥缓存时，仅在缓存丢失时才 AWS Encryption SDK 调用该 AWS KMS

GenerateDataKey操作来创建初始数据密钥。但是，仅在生成大量具有相同特性（包括相同加密上下文和算法套件）的数据密钥的应用程序中，缓存才会显著提高性能。

要确定您的实现是否实际上 AWS Encryption SDK 是在使用缓存中的数据密钥，请尝试以下技术。

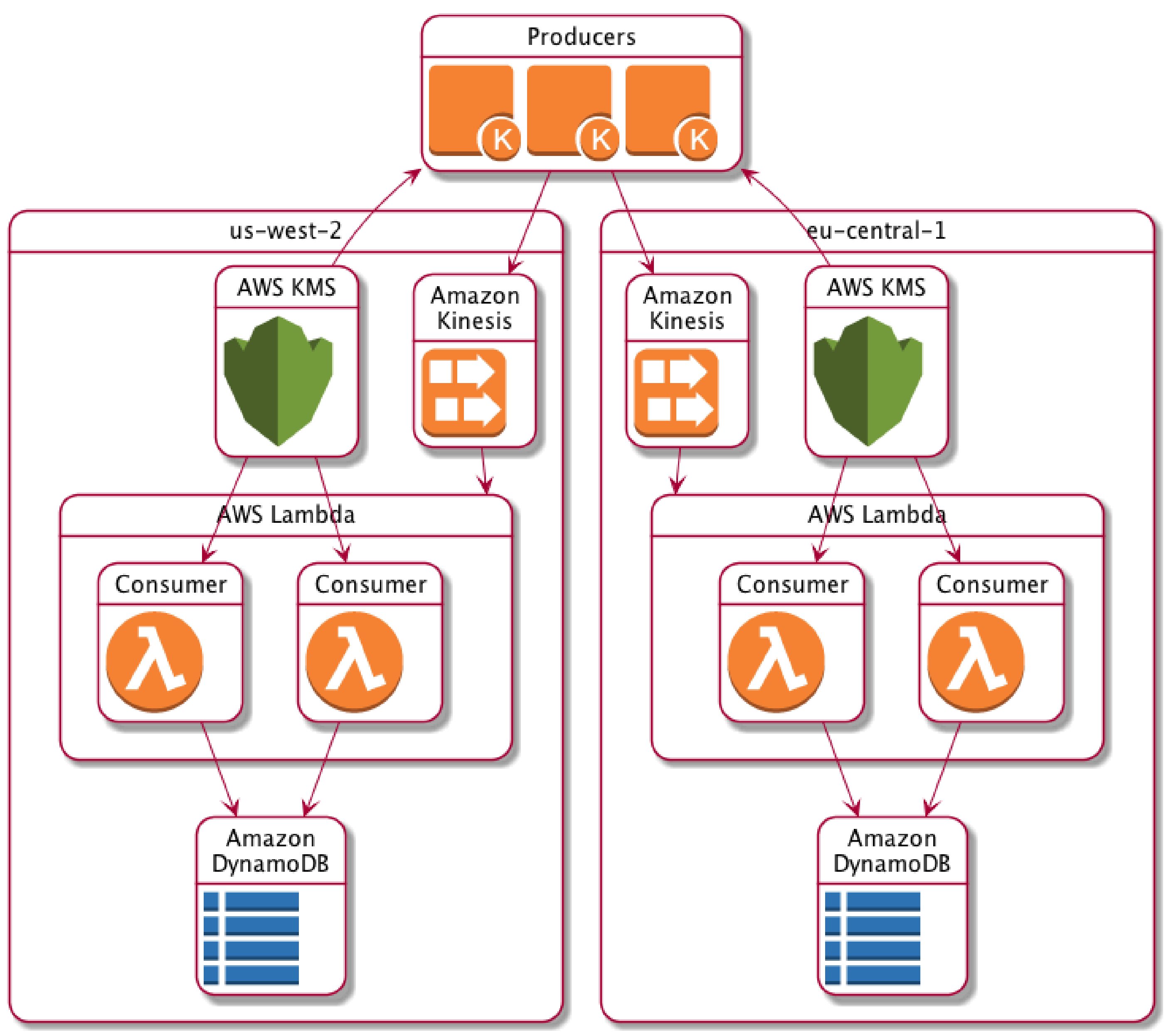
- 在主密钥基础设施的日志中，检查创建新数据密钥的调用频率。在数据密钥缓存有效时，创建新密钥的调用次数应明显减少。例如，如果您使用的是 AWS KMS 主密钥提供程序或密钥环，请在 CloudTrail 日志中搜索GenerateDataKey呼叫。
- 比较 AWS Encryption SDK 为响应不同的加密请求而返回的加密消息。例如，如果您使用的是 AWS Encryption SDK for Java，请比较来自不同加密调用的ParsedCiphertext对象。在中 AWS Encryption SDK for JavaScript，比较encryptedDataKeys属性的内容MessageHeader。在重复使用数据密钥时，加密消息中的加密数据密钥是完全相同的。

数据密钥缓存示例

该示例将数据密钥缓存与本地缓存一起使用以加快应用程序速度，其中，加密多个设备生成的数据并存储在不同的区域中。

在这种情况下，多个数据生成器生成数据，对其进行加密，然后写入到每个区域中的 Kinesis 流。AWS Lambda 函数（消费端）对流进行解密，然后将明文数据写入到区域中的 DynamoDB 表。数据生成器和消费端使用 AWS Encryption SDK 和 AWS KMS 密钥提供程序。要减少对 KMS 的调用，每个生成器和消费端具有自己的本地缓存。

您可以在 Java and Python 中找到这些示例的源代码。该示例还包括一个定义样本资源的 CloudFormation 模板。



本地缓存结果

下表演示了本地缓存将该示例中的总 KMS 调用次数（每个区域每秒）减少到原始值的 1%。

生成器请求

	每个客户端每秒的请求数			每个区域的客户端数	每个区域每秒的平均请求数
	生成数据密钥 (us-west-2)	加密数据密钥 (eu-central-1)	总计（每个区域）		

无缓存	1	1	1	500	500
本地缓存	1 rps/100 次使用	1 rps/100 次使用	1 rps/100 次使用	500	5

使用者请求

	每个客户端每秒的请求数			每个区域的客户端数	每个区域每秒的平均请求数
	解密数据密钥	创建者	Total		
无缓存	每个创建者 1 rps	500	500	2	1000
本地缓存	每个创建者 1 rps/100 次使用	500	5	2	10

数据密钥缓存示例代码

该代码示例在 Java 和 Python 中创建使用本地缓存的简易数据密钥缓存实施。该代码创建了两个本地缓存实例：一个用于加密数据的数据生产者，另一个用于解密数据的数据使用者（AWS Lambda 函数）。有关每种语言的数据密钥缓存实施的详细信息，请参阅适用于 AWS Encryption SDK的 Javadoc 和 Python 文档。

数据密钥缓存适用于 AWS Encryption SDK 支持的所有编程语言。

有关在中使用数据密钥缓存的完整且经过测试的示例 AWS Encryption SDK，请参阅：

- C/C++：caching_cmm.cpp
- Java：SimpleDataKeyCachingExample. java
- JavaScript 浏览器：caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python：data_key_caching_basic.py

Producer

制作者获取地图，将其转换为 JSON，使用对其进行加密，然后将密文记录推送到每个地图中的 Kinesis 流。 AWS Encryption SDK AWS 区域

该代码定义了缓存加密材料管理器（缓存 CMM），并将其与本地缓存和相关 AWS KMS 主密钥提供程序关联。缓存 CMM 缓存来自主密钥提供程序的数据密钥（和相关的加密材料）。它还代表该开发工具包与缓存进行交互，并实施您设置的安全阈值。

由于对加密方法的调用指定缓存 CMM 而非常规缓存加密材料管理器（CMM）或主密钥提供程序，加密将使用数据密钥缓存。

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

```
/*
 * Copyright 2017 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://aws.amazon.com/apache2.0
 *
 * 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.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
```

```
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.json.Jackson;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kms.KmsClient;

/**
 * Pushes data to Kinesis Streams in multiple Regions.
 */
public class MultiRegionRecordPusher {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 300000;
    private static final long MAX_ENTRY_USES = 100;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final String streamName_;
    private final ArrayList<KinesisClient> kinesisClients_;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;

    /**
     * Creates an instance of this object with Kinesis clients for all target
 Regions and a cached
     * key provider containing KMS master keys in all target Regions.
     */
    public MultiRegionRecordPusher(final Region[] regions, final String
 kmsAliasName,
        final String streamName) {
        streamName_ = streamName;
        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();
        kinesisClients_ = new ArrayList<>();

        AwsCredentialsProvider credentialsProvider =
 DefaultCredentialsProvider.builder().build();
```

```
        // Build KmsMasterKey and AmazonKinesisClient objects for each target region
        List<KmsMasterKey> masterKeys = new ArrayList<>();
        for (Region region : regions) {
            kinesisClients_.add(KinesisClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build());

            KmsMasterKey regionMasterKey = KmsMasterKeyProvider.builder()
                .defaultRegion(region)
                .builderSupplier(() ->
KmsClient.builder().credentialsProvider(credentialsProvider))
                .buildStrict(kmsAliasName)
                .getMasterKey(kmsAliasName);

            masterKeys.add(regionMasterKey);
        }

        // Collect KmsMasterKey objects into single provider and add cache
        MasterKeyProvider<?> masterKeyProvider =
MultipleProviderFactory.buildMultiProvider(
            KmsMasterKey.class,
            masterKeys
        );

        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
            .withMasterKeyProvider(masterKeyProvider)
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .withMessageUseLimit(MAX_ENTRY_USES)
            .build();
    }

    /**
     * JSON serializes and encrypts the received record data and pushes it to all
target streams.
     */
    public void putRecord(final Map<Object, Object> data) {
        String partitionKey = UUID.randomUUID().toString();
        Map<String, String> encryptionContext = new HashMap<>();
        encryptionContext.put("stream", streamName_);

        // JSON serialize data
```

```
        String jsonData = Jackson.toJsonString(data);

        // Encrypt data
        CryptoResult<byte[], ?> result = crypto_.encryptData(
            cachingMaterialsManager_,
            jsonData.getBytes(),
            encryptionContext
        );
        byte[] encryptedData = result.getResult();

        // Put records to Kinesis stream in all Regions
        for (KinesisClient regionalKinesisClient : kinesisClients_) {
            regionalKinesisClient.putRecord(builder ->
                builder.streamName(streamName_)
                    .data(SdkBytes.fromByteArray(encryptedData))
                    .partitionKey(partitionKey));
        }
    }
}
```

Python

```
"""
Copyright 2017 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

https://aws.amazon.com/apache-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 json
import uuid

from aws_encryption_sdk import EncryptionSDKClient, StrictAwsKmsMasterKeyProvider,
 CachingCryptoMaterialsManager, LocalCryptoMaterialsCache, CommitmentPolicy
from aws_encryption_sdk.key_providers.kms import KMSMasterKey
```

```
import boto3


class MultiRegionRecordPusher(object):
    """Pushes data to Kinesis Streams in multiple Regions."""
    CACHE_CAPACITY = 100
    MAX_ENTRY_AGE_SECONDS = 300.0
    MAX_ENTRY_MESSAGES_ENCRYPTED = 100

    def __init__(self, regions, kms_alias_name, stream_name):
        self._kinesis_clients = []
        self._stream_name = stream_name

        # Set up EncryptionSDKClient
        _client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

        # Set up KMSMasterKeyProvider with cache
        _key_provider = StrictAwsKmsMasterKeyProvider(kms_alias_name)

        # Add MasterKey and Kinesis client for each Region
        for region in regions:
            self._kinesis_clients.append(boto3.client('kinesis',
 region_name=region))
            regional_master_key = KMSMasterKey(
                client=boto3.client('kms', region_name=region),
                key_id=kms_alias_name
            )
            _key_provider.add_master_key_provider(regional_master_key)

        cache = LocalCryptoMaterialsCache(capacity=self.CACHE_CAPACITY)
        self._materials_manager = CachingCryptoMaterialsManager(
            master_key_provider=_key_provider,
            cache=cache,
            max_age=self.MAX_ENTRY_AGE_SECONDS,
            max_messages_encrypted=self.MAX_ENTRY_MESSAGES_ENCRYPTED
        )

    def put_record(self, record_data):
        """JSON serializes and encrypts the received record data and pushes it to
 all target streams.

        :param dict record_data: Data to write to stream
        """
```

```
        # Kinesis partition key to randomize write load across stream shards
        partition_key = uuid.uuid4().hex

        encryption_context = {'stream': self._stream_name}

        # JSON serialize data
        json_data = json.dumps(record_data)

        # Encrypt data
        encrypted_data, _header = _client.encrypt(
            source=json_data,
            materials_manager=self._materials_manager,
            encryption_context=encryption_context
        )

        # Put records to Kinesis stream in all Regions
        for client in self._kinesis_clients:
            client.put_record(
                StreamName=self._stream_name,
                Data=encrypted_data,
                PartitionKey=partition_key
            )
```

使用者

数据消费端是一个由 Kinesis 事件触发的 AWS Lambda 函数。其解密并反序列化每个记录，并将明文记录写入到同一区域中的 Amazon DynamoDB 表。

与生成器代码一样，消费端代码在对 Decrypt 方法的调用中使用缓存加密材料管理器（缓存 CMM）以启用数据密钥缓存。

Java 代码使用指定的在严格模式下构建主密钥提供程序 AWS KMS key。解密时无需使用严格模式，但这是最佳实践。Python 代码使用发现模式，允许 AWS Encryption SDK 使用加密数据密钥的任何包装密钥对其进行解密。

Java

以下示例使用版本 2。 的 x 个 AWS Encryption SDK for Java。版本 3。 其中 x AWS Encryption SDK for Java 已弃用数据密钥缓存 CMM。使用版本 3。 x，你也可以使用AWS KMS 分层密钥环，这是一种替代的加密材料缓存解决方案。

此代码创建了在严格模式下解密的主密钥提供程序。 AWS Encryption SDK 只能使用 AWS KMS keys 您指定的来解密您的消息。

```
/*
 * Copyright 2017 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://aws.amazon.com/apache2.0
 *
 * 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.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;
import com.amazonaws.util.BinaryUtils;
import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

/**
 * Decrypts all incoming Kinesis records and writes records to DynamoDB.
 */
public class LambdaDecryptAndWrite {
```

```
    private static final long MAX_ENTRY_AGE_MILLISECONDS = 600000;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;
    private final DynamoDbTable<Item> table_;

    /**
     * Because the cache is used only for decryption, the code doesn't set the max 
bytes or max
     * message security thresholds that are enforced only on on data keys used for 
encryption.
     */
    public LambdaDecryptAndWrite() {
        String kmsKeyArn = System.getenv("CMK_ARN");
        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()

.withMasterKeyProvider(KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn))
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .build();

        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

        String tableName = System.getenv("TABLE_NAME");
        DynamoDbEnhancedClient dynamodb = DynamoDbEnhancedClient.builder().build();
        table_ = dynamodb.table(tableName, TableSchema.fromClass(Item.class));
    }

    /**
     * @param event
     * @param context
     */
    public void handleRequest(KinesisEvent event, Context context)
        throws UnsupportedEncodingException {
        for (KinesisEventRecord record : event.getRecords()) {
            ByteBuffer ciphertextBuffer = record.getKinesis().getData();
            byte[] ciphertext = BinaryUtils.copyAllBytesFrom(ciphertextBuffer);

            // Decrypt and unpack record
            CryptoResult<byte[], ?> plaintextResult = 
crypto_.decryptData(cachingMaterialsManager_,
                ciphertext);
```

```
            // Verify the encryption context value
            String streamArn = record.getEventSourceARN();
            String streamName = streamArn.substring(streamArn.indexOf("/") + 1);
            if (!
streamName.equals(plaintextResult.getEncryptionContext().get("stream"))) {
                throw new IllegalStateException("Wrong Encryption Context!");
            }

            // Write record to DynamoDB
            String jsonItem = new String(plaintextResult.getResult(),
 StandardCharsets.UTF_8);
            System.out.println(jsonItem);
            table_.putItem(Item.fromJSON(jsonItem));
        }
    }

    private static class Item {

        static Item fromJSON(String jsonText) {
            // Parse JSON and create new Item
            return new Item();
        }
    }
}
```

Python

此 Python 代码在发现模式下使用主密钥提供程序进行解密。该代码允许 AWS Encryption SDK 使用任何加密数据密钥的包装密钥对其进行解密。最佳实践是使用严格模式，在这种模式下，您可以指定可用于解密的包装密钥。

```
"""
Copyright 2017 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

https://aws.amazon.com/apache-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 base64
import json
import logging
import os

from aws_encryption_sdk import EncryptionSDKClient,
 DiscoveryAwsKmsMasterKeyProvider, CachingCryptoMaterialsManager,
 LocalCryptoMaterialsCache, CommitmentPolicy
import boto3

_LOGGER = logging.getLogger(__name__)
_is_setup = False
CACHE_CAPACITY = 100
MAX_ENTRY_AGE_SECONDS = 600.0

def setup():
    """Sets up clients that should persist across Lambda invocations."""
    global encryption_sdk_client
    encryption_sdk_client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

    global materials_manager
    key_provider = DiscoveryAwsKmsMasterKeyProvider()
    cache = LocalCryptoMaterialsCache(capacity=CACHE_CAPACITY)

    #  Because the cache is used only for decryption, the code doesn't set
    #   the max bytes or max message security thresholds that are enforced
    #   only on on data keys used for encryption.
    materials_manager = CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=MAX_ENTRY_AGE_SECONDS
    )
    global table
    table_name = os.environ.get('TABLE_NAME')
    table = boto3.resource('dynamodb').Table(table_name)
    global _is_setup
    _is_setup = True
```

```
def lambda_handler(event, context):
    """Decrypts all incoming Kinesis records and writes records to DynamoDB."""
    _LOGGER.debug('New event:')
    _LOGGER.debug(event)
    if not _is_setup:
        setup()
    with table.batch_writer() as batch:
        for record in event.get('Records', []):
            # Record data base64-encoded by Kinesis
            ciphertext = base64.b64decode(record['kinesis']['data'])

            # Decrypt and unpack record
            plaintext, header = encryption_sdk_client.decrypt(
                source=ciphertext,
                materials_manager=materials_manager
            )
            item = json.loads(plaintext)

            # Verify the encryption context value
            stream_name = record['eventSourceARN'].split('/', 1)[1]
            if stream_name != header.encryption_context['stream']:
                raise ValueError('Wrong Encryption Context!')

            # Write record to DynamoDB
            batch.put_item(Item=item)
```

数据密钥缓存示例： CloudFormation 模板

此 CloudFormation 模板设置了所有必要的 AWS 资源来重现数据密钥缓存示例。

JSON

```
{
    "Parameters": {
        "SourceCodeBucket": {
            "Type": "String",
            "Description": "S3 bucket containing Lambda source code zip files"
        },
        "PythonLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
```

```
        },
        "PythonLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
        "JavaLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
        },
        "JavaLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
        "KeyAliasSuffix": {
            "Type": "String",
            "Description": "Suffix to use for KMS key Alias (ie: alias/
KeyAliasSuffix)"
        },
        "StreamName": {
            "Type": "String",
            "Description": "Name to use for Kinesis Stream"
        }
    },
    "Resources": {
        "InputStream": {
            "Type": "AWS::Kinesis::Stream",
            "Properties": {
                "Name": {
                    "Ref": "StreamName"
                },
                "ShardCount": 2
            }
        },
        "PythonLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
```

```
                "KeySchema": [
                    {
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "PythonLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "PythonLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
```

```
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
                                        "kinesis:DescribeStream",
                                        "kinesis:ListStreams"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "PythonLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Python consumer",
                "Runtime": "python2.7",
                "MemorySize": 512,
                "Timeout": 90,
                "Role": {
                    "Fn::GetAtt": [
                        "PythonLambdaRole",
                        "Arn"
```

```
                    ]
                },
                "Handler":
 "aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler",
                "Code": {
                    "S3Bucket": {
                        "Ref": "SourceCodeBucket"
                    },
                    "S3Key": {
                        "Ref": "PythonLambdaS3Key"
                    },
                    "S3ObjectVersion": {
                        "Ref": "PythonLambdaObjectVersionId"
                    }
                },
                "Environment": {
                    "Variables": {
                        "TABLE_NAME": {
                            "Ref": "PythonLambdaOutputTable"
                        }
                    }
                }
            }
        },
        "PythonLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
                    "Ref": "PythonLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "JavaLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
```

```
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "JavaLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "JavaLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
```

```
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
                                        "kinesis:DescribeStream",
                                        "kinesis:ListStreams"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "JavaLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Java consumer",
                "Runtime": "java8",
                "MemorySize": 512,
                "Timeout": 90,
```

```
                "Role": {
                    "Fn::GetAtt": [
                        "JavaLambdaRole",
                        "Arn"
                    ]
                },
                "Handler":
 "com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest",
                "Code": {
                    "S3Bucket": {
                        "Ref": "SourceCodeBucket"
                    },
                    "S3Key": {
                        "Ref": "JavaLambdaS3Key"
                    },
                    "S3ObjectVersion": {
                        "Ref": "JavaLambdaObjectVersionId"
                    }
                },
                "Environment": {
                    "Variables": {
                        "TABLE_NAME": {
                            "Ref": "JavaLambdaOutputTable"
                        },
                        "CMK_ARN": {
                            "Fn::GetAtt": [
                                "RegionKinesisCMK",
                                "Arn"
                            ]
                        }
                    }
                }
            }
        },
        "JavaLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
```

```
                    "Ref": "JavaLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "RegionKinesisCMK": {
            "Type": "AWS::KMS::Key",
            "Properties": {
                "Description": "Used to encrypt data passing through Kinesis Stream
 in this region",
                "Enabled": true,
                "KeyPolicy": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "AWS": {
                                    "Fn::Sub": "arn:aws:iam::${AWS::AccountId}:root"
                                }
                            },
                            "Action": [
                                "kms:Encrypt",
                                "kms:GenerateDataKey",
                                "kms:CreateAlias",
                                "kms:DeleteAlias",
                                "kms:DescribeKey",
                                "kms:DisableKey",
                                "kms:EnableKey",
                                "kms:PutKeyPolicy",
                                "kms:ScheduleKeyDeletion",
                                "kms:UpdateAlias",
                                "kms:UpdateKeyDescription"
                            ],
                            "Resource": "*"
                        },
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "AWS": [
                                    {
                                        "Fn::GetAtt": [
                                            "PythonLambdaRole",
                                            "Arn"
```

```
                                        ]
                                    },
                                    {
                                        "Fn::GetAtt": [
                                            "JavaLambdaRole",
                                            "Arn"
                                        ]
                                    }
                                ]
                            },
                            "Action": "kms:Decrypt",
                            "Resource": "*"
                        }
                    ]
                }
            }
        },
        "RegionKinesisCMKAlias": {
            "Type": "AWS::KMS::Alias",
            "Properties": {
                "AliasName": {
                    "Fn::Sub": "alias/${KeyAliasSuffix}"
                },
                "TargetKeyId": {
                    "Ref": "RegionKinesisCMK"
                }
            }
        }
    }
}
```

YAML

```
Parameters:
    SourceCodeBucket:
        Type: String
        Description: S3 bucket containing Lambda source code zip files
    PythonLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    PythonLambdaObjectVersionId:
        Type: String
```

```
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    JavaLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    JavaLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    KeyAliasSuffix:
        Type: String
        Description: 'Suffix to use for KMS CMK Alias (ie: alias/<KeyAliasSuffix>)'
    StreamName:
        Type: String
        Description: Name to use for Kinesis Stream
Resources:
    InputStream:
        Type: AWS::Kinesis::Stream
        Properties:
            Name: !Ref StreamName
            ShardCount: 2
    PythonLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    PythonLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
```

```
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: PythonLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    PythonLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Python consumer
            Runtime: python2.7
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt PythonLambdaRole.Arn
            Handler:
 aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler
            Code:
                S3Bucket: !Ref SourceCodeBucket
```

```
                S3Key: !Ref PythonLambdaS3Key
                S3ObjectVersion: !Ref PythonLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref PythonLambdaOutputTable
    PythonLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref PythonLambdaFunction
            StartingPosition: TRIM_HORIZON
    JavaLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    JavaLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: JavaLambdaAccess
```

```
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    JavaLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Java consumer
            Runtime: java8
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt JavaLambdaRole.Arn
            Handler:
 com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref JavaLambdaS3Key
                S3ObjectVersion: !Ref JavaLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref JavaLambdaOutputTable
                    CMK_ARN: !GetAtt RegionKinesisCMK.Arn
    JavaLambdaSourceMapping:
```

```
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref JavaLambdaFunction
            StartingPosition: TRIM_HORIZON
    RegionKinesisCMK:
        Type: AWS::KMS::Key
        Properties:
            Description: Used to encrypt data passing through Kinesis Stream in this
 region
            Enabled: true
            KeyPolicy:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            AWS: !Sub arn:aws:iam::${AWS::AccountId}:root
                        Action:
                            # Data plane actions
                            - kms:Encrypt
                            - kms:GenerateDataKey
                            # Control plane actions
                            - kms:CreateAlias
                            - kms:DeleteAlias
                            - kms:DescribeKey
                            - kms:DisableKey
                            - kms:EnableKey
                            - kms:PutKeyPolicy
                            - kms:ScheduleKeyDeletion
                            - kms:UpdateAlias
                            - kms:UpdateKeyDescription
                        Resource: '*'
                    -
                        Effect: Allow
                        Principal:
                            AWS:
                                - !GetAtt PythonLambdaRole.Arn
                                - !GetAtt JavaLambdaRole.Arn
                        Action: kms:Decrypt
                        Resource: '*'
```

```
    RegionKinesisCMKAlias:
        Type: AWS::KMS::Alias
        Properties:
            AliasName: !Sub alias/${KeyAliasSuffix}
            TargetKeyId: !Ref RegionKinesisCMK
```

的版本 AWS Encryption SDK

AWS Encryption SDK 语言实现使用语义版本控制，以便您更轻松地识别每个版本中变化的幅度。主要版本编号的更改，例如 1.x.x 更改为 2.x.x，表示一项重大更改，可能需要更改代码和计划部署。新版本中的重大更改可能不会影响每个用例，请查看发行说明以了解您是否受到影响。次要版本的更改，例如 x.1.x 更改为 x.2.x，始终向后兼容，但可能包含已弃用的元素。

只要有可能，请使用所选编程语言 AWS Encryption SDK 中的最新版本的。每个版本的维护和支持策略因编程语言实现而异。有关首选编程语言支持的版本的详细信息，请参阅其GitHub存储库中的SUPPORT_POLICY.rst文件。

当升级包含需要特殊配置以避免加密或解密错误的新功能时，我们会提供中间版本和详细的使用说明。例如，版本 1.7.x 和 1.8.x 被设计为过渡版本，可帮助您从 1.7.x 之前的版本升级到版本 2.0.x 及更高版本。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

> **Note**
>
> 版本号中的 x 表示主要版本和次要版本的任何补丁。例如，版本 1.7.x 表示所有以 1.7 开头的版本，包括 1.7.1 和 1.7.9。
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是，AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅 aws-encryption-sdk-cli 存储库中的相关安全公告。 GitHub

下表概述了每种编程语言支持的版本之间的主要区别。 AWS Encryption SDK

C

有关所有更改的详细说明，请参阅 aws-encryption-sdk-c 存储库CHANGELOG.md中的。 GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
1.x	1.0	初始版本。	End-of-Support 阶段
	1.7	的 AWS Encryption SDK 更新可帮助早期	

		版本的用户升级到 2.0 版。 x 及更高版本。有关更多信息，请参阅版本 1.7。 x。	
2.x	2.0	的更新 AWS Encryption SDK. 有关更多信息，请参阅 2.0 版。 x。	正式发布（GA）
	2.2	消息解密过程的改进。	
	2.3	增加了对 AWS KMS 多区域密钥的支持。	

C# /.NET

有关所有更改的详细说明，请参阅`aws-encryption-sdk-net`存储库CHANGELOG.md中的 GitHub。

主要版本	详细信息		SDK 主要版本生命周期阶段
3.x	3.1.0	初始版本。	End-of-Support AWS Encryption SDK 适用于.NET 的 3.x 版本已进入终止支持；请升级到 4. x。
4.x	4.0	增加了对 AWS KMS 分层密钥环、所需的加密上下文 CMM 和非对称 RSA 密钥环的支持。 AWS KMS	正式发布（GA）
5.x	5.0	更新至版本 2。 材料提供者库 (MPL) 的	正式发布（GA）

		x。取消对适用于.NET v3 的 AWS SDK 的支持，需要使用适用于.NET v4 的 AWS SDK。	

命令行界面 (CLI)

有关所有更改的详细说明，请参阅AWS 加密 CLI 的版本上的 aws-encryption-sdk-cli 存储库CHANGELOG.rst中的。 GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
1.x	1.0	初始版本。	End-of-Support 阶段
	1.7	的 AWS Encryption SDK 更新可帮助早期版本的用户升级到 2.0 版。 x 及更高版本。有关更多信息，请参阅版本 1.7。 x。	
2.x	2.0	的更新 AWS Encryption SDK. 有关更多信息，请参阅 2.0 版。 x。	End-of-Support 阶段
	2.1	移除`--discovery` 参数并将其替换为`--wrapping-keys` 参数的`discovery` 属性。 AWS 加密 CLI 的 2.1.0 版本等同于其他编程语言中的 2.0 版。	

	2.2	消息解密过程的改进。	
3.x	3.0	增加了对 AWS KMS 多区域密钥的支持。	End-of-Support 阶段
4.x	4.0	AWS 加密 CLI 不再支持 Python 2 或 Python 3.4。从主版本4开始。 x 在 AWS 加密 CLI 中，仅支持 Python 3.5 或更高版本。	正式发布（GA）
	4.1	AWS 加密 CLI 不再支持 Python 3.5。从 4.1 版本开始。 x 在 AWS 加密 CLI 中，仅支持 Python 3.6 或更高版本。	
	4.2	AWS 加密 CLI 不再支持 Python 3.6。从版本 4.2 开始。 x 在 AWS 加密 CLI 中，仅支持 Python 3.7 或更高版本。	

Java

有关所有更改的详细说明，请参阅 aws-encryption-sdk-java 存储库CHANGELOG.rst中的。 GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
1.x	1.0	初始版本。	End-of-Support 阶段
	1.3	增加了对加密材料管理器和数据密钥缓存的	

		支持。移至确定性 IV 代。	
	1.6.1	弃用`AwsCrypto.encryptString()` 和并将其替换为`AwsCrypto.decryptString()` 和。`AwsCrypto.encryptData()` `AwsCrypto.decryptData()`	
	1.7	的 AWS Encryption SDK 更新可帮助早期版本的用户升级到 2.0 版。 x 及更高版本。有关更多信息，请参阅版本 1.7。 x。	
2.x	2.0	的更新 AWS Encryption SDK. 有关更多信息，请参阅 2.0 版。 x。	正式发布（GA） 的 2.x 版本 AWS Encryption SDK for Java 将于 2024 年进入维护模式。
	2.2	消息解密过程的改进。	
	2.3	增加了对 AWS KMS 多区域密钥的支持。	
	2.4	添加了对的支持 AWS SDK for Java 2.x。	

3.x	3.0	AWS Encryption SDK for Java 与材料提供者库 (MPL) 集成。 增加了对对称和非对称 RSA AWS KMS 密钥环、 AWS KMS ECDH 密钥环、 AWS KMS 分层密钥环、原始 AES 密钥环、原始 RSA 密钥环、未处理 ECDH 密钥环和所需的加密上下文 CMM 的支持。 Multi-keyrings	正式发布（GA）

Go

有关所有更改的详细说明，请参阅 aws-encr CHANGELOG.mdyption-s dk 存储库的 Go 目录中的。 GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
0.1。 x	0.1.0	初始版本。	正式发布（GA）

JavaScript

有关所有更改的详细描述，请参阅 aws-encryption-sdk-javascript 存储库CHANGELOG.md中的。 GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
1.x	1.0	初始版本。	End-of-Support 阶段

	1.7	的 AWS Encryption SDK 更新可帮助早期版本的用户升级到 2.0 版。 x 及更高版本。有关更多信息，请参阅版本 1.7。 x。	
2.x	2.0	的更新 AWS Encryption SDK. 有关更多信息，请参阅 2.0 版。 x。	End-of-Support 阶段
	2.2	消息解密过程的改进。	
	2.3	增加了对 AWS KMS 多区域密钥的支持。	
3.x	3.0	移除节点 10 的 CI 覆盖范围。升级依赖关系以不再支持节点 8 和节点 10。	Maintenance 对版本 3.x 的支持 AWS Encryption SDK for JavaScript 将于 2024 年 1 月 17 日结束。
4.x	4.0	需要版本 3 `kms-client` 才能使用 AWS KMS 密钥环。 AWS Encryption SDK for JavaScript	正式发布（GA）

Python

有关所有更改的详细描述，请参阅 aws-encryption-sdk-python 存储库CHANGELOG.rst中的。 GitHub

主要版本	详细信息	SDK 主要版本生命周期阶段

1.x	1.0	初始版本。	End-of-Support 阶段
	1.3	增加了对加密材料管理器和数据密钥缓存的支持。移至确定性 IV 代。	
	1.7	的 AWS Encryption SDK 更新可帮助早期版本的用户升级到 2.0 版。 x 及更高版本。有关更多信息，请参阅版本 1.7。 x。	
2.x	2.0	的更新 AWS Encryption SDK. 有关更多信息，请参阅 2.0 版。 x。	End-of-Support 阶段
	2.2	消息解密过程的改进。	
	2.3	增加了对 AWS KMS 多区域密钥的支持。	
3.x	3.0	AWS Encryption SDK for Python 不再支持 Python 2 或 Python 3.4。从主版本3开始。其中 x AWS Encryption SDK for Python，仅支持 Python 3.5 或更高版本。	正式发布（GA）
4.x	4.0	AWS Encryption SDK for Python 与材料提供者库 (MPL) 集成。	正式发布（GA）

Rust

有关所有更改的详细说明，请参阅 aws-encr CHANGELOG.mdyption-s dk 存储库 Rust 目录中的。GitHub

主要版本	详细信息		SDK 主要版本生命周期阶段
1.x	1.0	初始版本。	正式发布（GA）

版本详情

以下列表描述了支持的 AWS Encryption SDK版本之间的主要区别。

主题

- 1.7 之前的版本。x
- 版本 1.7。x
- 版本 2.0。x
- 版本 2.2。x
- 版本 2.3。x

1.7 之前的版本。x

> **Note**
>
> 全部 1. x。 x 版本的 AWS Encryption SDK 处于支持终止阶段。只要可行，请尽快升级到 AWS Encryption SDK 适用于您的编程语言的最新可用版本。从 1.7 之前的 AWS Encryption SDK 版本升级。 x，必须先升级到 1.7。 x。有关更多信息，请参阅 迁移你的 AWS Encryption SDK。

1.7 AWS Encryption SDK 之前的版本。 x 提供重要的安全功能，包括在 Galois/Counter 模式 (AES-GCM) 中使用高级加密标准算法进行加密、 HMAC-based 提取和扩展密钥派生功能 (HKDF)、签名和 256 位加密密钥。但是，这些版本不支持我们推荐的最佳实践，包括密钥承诺。

版本 1.7。x

Note

全部 1. x。 x 版本的 AWS Encryption SDK 处于支持终止阶段。

版本 1.7。 x 旨在帮助早期版本的用户升级 AWS Encryption SDK 到 2.0 版。 x 及更高版本。如果您不熟悉 AWS Encryption SDK，可以跳过此版本，从您的编程语言的最新可用版本开始。

版本 1.7.x 完全向后兼容；未引入任何重大更改或更改 AWS Encryption SDK的行为。该版本还向前兼容；允许您更新代码，以便与版本 2.0.x 兼容。其中包含新功能，但并未完全启用。此外，该版本需要配置值，以防止您在准备就绪之前立即采用所有新功能。

版本 1.7.x 包含以下更改：

AWS KMS 主密钥提供程序更新（必需）

版本 1.7。 x 向 AWS Encryption SDK for Java 和引入了新的构造函数 AWS Encryption SDK for Python ，这些构造函数在严格模式或发现模式下显式创建 AWS KMS 主密钥提供程序。此版本对 AWS Encryption SDK 命令行界面 (CLI) 添加了类似的更改。有关更多信息，请参阅 更新 AWS KMS 主密钥提供程序。

- 在严格模式下， AWS KMS 主密钥提供程序需要包装密钥列表，这些提供程序仅使用您指定的包装密钥进行加密和解密。这是一种 AWS Encryption SDK 最佳实践，可确保您使用的是要使用的包装密钥。
- 在发现模式下， AWS KMS 主密钥提供程序不使用任何包装密钥。您不能使用这些提供程序进行加密。在解密时，这些提供程序可以使用任何包装密钥解密加密的数据密钥。但是，您可以将用于解密的包装密钥限制为特定的 AWS 账户。账户筛选是可选的，但这是我们推荐的最佳实践。

在 1.7 版本中，创建早期版本 AWS KMS 的主密钥提供程序的构造函数已被弃用。 x 并在版本 2.0 中移除。 x。这些构造函数实例化主密钥提供程序，这些提供程序使用您指定的包装密钥进行加密。但是，这些提供程序使用加密数据密钥的包装密钥来解密已加密的数据密钥，而不考虑指定的包装密钥。用户可能会无意中使用他们不打算使用的包装密钥解密邮件，包括在其他 AWS KMS keys 地区 AWS 账户 和区域。

AWS KMS 主密钥的构造函数没有变化。加密和解密时， AWS KMS 主密钥仅使用您指定的密钥。AWS KMS key

AWS KMS 密钥环更新（可选）

版本 1.7。 x 在 AWS Encryption SDK for C 和 AWS Encryption SDK for JavaScript 实现中添加了一个新的过滤器，将AWS KMS 发现密钥环限制为特定 AWS 账户。这个新的账户筛选条件是可选的，但这是我们推荐的最佳实践。有关更多信息，请参阅 更新 AWS KMS 钥匙圈。

AWS KMS 钥匙圈的构造函数没有变化。在严格模式下，标准密 AWS KMS 钥环的行为类似于主密钥提供程序。 AWS KMS 发现密钥环是在发现模式下显式创建的。

将密钥 ID 传递给 AWS KMS 解密

从 1.7 版本开始。 x，解密加密的数据密钥时， AWS Encryption SDK 始终 AWS KMS key 在调用 D AWS KMS ec rypt 操作时指定。 AWS KMS key 从每个加密数据密钥中的元数据中 AWS Encryption SDK 获取的密钥 ID 值。此功能不需要任何代码更改。

解密使用对 AWS KMS key 称加密 KMS 密钥加密的密文不需要指定密钥 ID，但这是最佳实践。AWS KMS与在密钥提供程序中指定包装密钥一样，这种做法可确保 AWS KMS 仅使用您打算使用的包装密钥进行解密。

使用密钥承诺解密加密文字

版本 1.7.x 可以解密使用或不使用密钥承诺加密的加密文字。但是，无法使用密钥承诺加密加密文字。此属性允许您在遇到任何此类加密文字之前，完全部署能够解密使用密钥承诺加密的加密文字的应用程序。由于此版本可解密未经密钥承诺而加密的消息，您无需重新加密任何加密文字。

要实现此行为，请使用版本 1.7。 x 包括新的承诺策略配置设置，该设置决定了是否 AWS Encryption SDK 可以使用密钥承诺进行加密或解密。在版本 1.7.x 中，`ForbidEncryptAllowDecrypt` 是承诺策略的唯一有效值，用于所有加密和解密操作。此值可防止 AWS Encryption SDK 使用包含密钥承诺的新算法套件进行加密。它允许使用和 AWS Encryption SDK 不使用密钥承诺来解密密文。

尽管版本 1.7.x 中只有一个有效的承诺策略值，但我们要求您在使用本版本中引入的新 API 时可以明确设置此值。明确设置该值可防止您的承诺策略在升级到版本 2.1.x 时自动更改为 `require-encrypt-require-decrypt`。相反，您可以分阶段迁移承诺策略。

带有密钥承诺的算法套件

版本 1.7.x 包括两个支持密钥承诺的新算法套件。一个包括签名；另一个不包括。与之前支持的算法套件一样，这两个新的算法套件都包括使用加密 AES-GCM、256 位加密密钥和 HMAC-based 提取和扩展密钥派生函数 (HKDF)。

但是，用于加密的默认算法套件不会更改。这些算法套件已添加至版本 1.7.x，让您的应用程序做好准备，以便在版本 2.0.x 及更高版本中使用这些套件。

CMM 实现更改

版本 1.7.x 引入了对默认加密材料管理器（CMM）界面的更改，以支持密钥承诺。此更改仅在您编写了自定义 CMM 时才会影响到您。有关详细信息，请参阅您的编程语言的 API 文档或 GitHub 存储库。

版本 2.0。x

版本 2.0。 x 支持中提供的新安全功能 AWS Encryption SDK，包括指定的包装密钥和密钥承诺。为支持这些功能，版本 2.0.x 包括对 AWS Encryption SDK早期版本的重大更改。您可以通过部署版本 1.7.x 为这些更改做好准备。版本 2.0.x 包含版本 1.7.x 中引入的所有新功能，有以下补充和更改。

> **Note**
>
> 版本 2。 x。 AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、和 AWS 加密 CLI 中的 x 处于支持终止阶段。
> 有关以您的首选编程语言支持和维护此 AWS Encryption SDK 版本的信息，请参阅其GitHub存储库中的`SUPPORT_POLICY.rst`文件。

AWS KMS 主密钥提供者

1.7 版本中不推荐使用的原始 AWS KMS 主密钥提供程序构造函数。 x 已在 2.0 版本中移除。 x。您必须在严格模式或发现模式下明确构造 AWS KMS 主密钥提供程序。

使用密钥承诺加密和解密加密文字

版本 2.0.x 可以使用或不使用密钥承诺来加密和解密加密文字。其行为由承诺策略的设置决定。默认情况下，始终使用密钥承诺进行加密，并且仅解密使用密钥承诺加密的加密文字。除非您更改承诺策略，否则 AWS Encryption SDK 不会解密由任何 AWS Encryption SDK早期版本（包括版本 1.7.x）加密的加密文字。

> **Important**
>
> 默认情况下，版本 2.0.x 不会解密任何不使用密钥承诺加密的加密文字。如果您的应用程序可能遇到不使用密钥承诺加密的加密文字，请使用 `AllowDecrypt` 设置承诺策略值。

在版本 2.0.x 中，承诺策略设置有三个有效值：

- `ForbidEncryptAllowDecrypt` – AWS Encryption SDK 无法使用密钥承诺进行加密。可以解密使用或不使用密钥承诺加密的加密文字。
- `RequireEncryptAllowDecrypt` – AWS Encryption SDK 必须使用密钥承诺进行加密。可以解密使用或不使用密钥承诺加密的加密文字。
- `RequireEncryptRequireDecrypt`（默认）- AWS Encryption SDK 必须使用密钥承诺进行加密。仅使用密钥承诺解密加密文字。

如果您要从的早期版本迁移 AWS Encryption SDK 到版本 2.0。 x，将承诺策略设置为一个值，该值可确保您可以解密应用程序可能遇到的所有现有密文。随着时间的推移，您可能会调整此设置。

版本 2.2。x

增加了对数字签名和限制加密数据密钥的支持。

> **Note**
>
> 版本 2。 x。 AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、和 AWS 加密 CLI 中的 x 处于支持终止阶段。
> 有关以您的首选编程语言支持和维护此 AWS Encryption SDK 版本的信息，请参阅其GitHub存储库中的`SUPPORT_POLICY.rst`文件。

数字签名

为了改进解密时对数字签名的处理， AWS Encryption SDK 包括以下功能：

- Non-streaming 模式 — 仅在处理完所有输入后才返回纯文本，包括验证数字签名（如果存在）。此功能可防止您在验证数字签名之前使用明文。每当您解密使用数字签名（默认算法套件）加密的数据时，请使用此功能。例如，由于 AWS Encryption CLI 始终以流模式处理数据，因此在使用数字签名解密密文时使用该`- -buffer`参数。
- Unsigned-only 解密模式 — 此功能仅解密未签名的密文。如果解密遇到加密文字中的数字签名，则操作将失败。使用此功能可以避免在验证签名之前无意中处理已签名邮件中的明文。

限制加密数据密钥

您可以在加密消息中限制加密数据密钥的数量。此功能可以帮助您在加密时检测配置错误的主密钥提供程序或密钥环，或者在解密时识别恶意加密文字。

解密来自不可信来源的消息时，应限制加密数据密钥。这样可以防止对您的密钥基础设施进行不必要、昂贵、可能详尽的调用。

版本 2.3。x

增加了对 AWS KMS 多区域密钥的支持。有关更多信息，请参阅 使用多区域 AWS KMS keys。

> **Note**
>
> AWS 加密 CLI 支持从 3.0 版开始的多区域密钥。 x。
>
> 版本 2。 x。 AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、和 AWS 加密 CLI 中的 x 处于支持终止阶段。
>
> 有关以您的首选编程语言支持和维护此 AWS Encryption SDK 版本的信息，请参阅其GitHub存储库中的`SUPPORT_POLICY.rst`文件。

迁移你的 AWS Encryption SDK

AWS Encryption SDK 支持多种可互操作的编程语言实现，每种实现都是在上的开源存储库中开发的。GitHub作为最佳实践，我们建议您对每种语言使用最新版本 AWS Encryption SDK 的。

您可以安全地从 2.0 版本升级。 x 或更高版本 AWS Encryption SDK 到最新版本。但是，2.0。 x 版本 AWS Encryption SDK 引入了重要的新安全功能，其中一些是重大更改。要从 1.7.x 之前的版本更新到版本 2.0.x 及更高版本，必须先更新到最新版本 1.x。本节中的主题旨在帮助您了解更改，为应用程序选择正确的版本，并成功安全迁移到 AWS Encryption SDK最新版本。

有关重要版本的信息 AWS Encryption SDK，请参见的版本 AWS Encryption SDK。

> **Important**
>
> 若未升级到最新版本 1.x，请勿直接从 1.7.x 之前的版本升级到版本 2.0.x 或更高版本。如果您直接升级到 2.0 版。 x 或更高版本并立即启用所有新功能，则 AWS Encryption SDK 无法解密在旧版本下加密的密文。 AWS Encryption SDK

> **Note**
>
> .NET AWS Encryption SDK 的最早版本是 3.0 版。 x。 AWS Encryption SDK 适用于.NET 的所有版本都支持 2.0 中引入的安全最佳实践。 的 x 个 AWS Encryption SDK。无需更改任何代码或数据即可安全升级到最新版本。
>
> AWS 加密 CLI：阅读本迁移指南时，请使用 1.7。 x AWS 加密 CLI 1.8 的迁移说明。 x 然后使用 2.0。 x AWS 加密 CLI 2.1 的迁移说明。 x。有关更多信息，请参阅 AWS 加密 CLI 的版本。
>
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

新用户

如果您不熟悉 AWS Encryption SDK，请安装 AWS Encryption SDK 适用于您的编程语言的最新版本的。默认值启用的所有安全功能 AWS Encryption SDK，包括带签名的加密、密钥派生和密钥承诺。 AWS Encryption SDK

当前用户

> 我们建议您尽快从当前版本升级到最新可用版本。全部 1. AWS Encryption SDK 的 x 版本正end-of-support 处于阶段，某些编程语言的更高版本也是如此。有关您的编程语言中 AWS Encryption SDK 的支持和维护状态的详细信息，请参阅 支持和维护。
>
> AWS Encryption SDK 版本 2.0。 x 及更高版本提供了新的安全功能来帮助保护您的数据。但是，AWS Encryption SDK 版本为 2.0。 x 包括不向后兼容的重大更改。为确保安全过渡，请首先在你的编程语言中从当前版本迁移到最新版本 1.x。如果最新版本 1.x 已全面部署并成功运行，您可以安全迁移到版本 2.0.x 及更高版本。这个两步过程至关重要，对于分布式应用程序尤其如此。

有关这些更改所依据 AWS Encryption SDK 的安全功能的更多信息，请参阅AWS 安全博客中的改进客户端加密：显式 KeyIds 和密钥承诺。

在使用时正在寻求帮助 AWS SDK for Java 2.x？ AWS Encryption SDK for Java 请参阅先决条件。

主题

- 如何迁移和部署 AWS Encryption SDK
- 更新 AWS KMS 主密钥提供程序
- 更新 AWS KMS 钥匙圈
- 设置您的承诺策略
- 对迁移到至最新版本进行故障排除

如何迁移和部署 AWS Encryption SDK

从 1.7 之前的 AWS Encryption SDK 版本迁移时。 x 到 2.0 版。 x 或更高版本，您必须安全地过渡到使用密钥承诺进行加密。否则，您的应用程序将遇到无法解密的加密文字。如果您使用的是 AWS KMS 主密钥提供程序，则必须更新到在严格模式或发现模式下创建主密钥提供程序的新构造函数。

> **Note**
>
> 本主题专为从 AWS Encryption SDK 早期版本迁移到版本 2.0.x 或更高版本的用户而设计。如果您不熟悉 AWS Encryption SDK，则可以在默认设置下立即开始使用最新的可用版本。

为避免出现无法解密需要读取的加密文字的严重情况，我们建议您分多个不同阶段进行迁移和部署。在开始下一阶段之前，请确认各阶段均已结束并全面部署。这对于具有多台主机的分布式应用程序尤其重要。

阶段 1：将您的应用程序更新到最新版本 1.x

更新到适用于您的编程语言的最新版本 1.x。在开始阶段 2 之前，请仔细测试、部署更改并确认更新已传播到所有目标主机。

> **Important**
>
> 验证最新版本 1.x 为 AWS Encryption SDK版本 1.7.x 或更高版本。

最新的 1. x 版本向后兼容的旧版本， AWS Encryption SDK 向前兼容版本 2.0。 AWS Encryption SDK x 及更高版本。最新版本包括版本 2.0.x 的新功能，但也包括为此迁移设计的安全默认值。它们允许您在必要时升级 AWS KMS 主密钥提供程序，并使用可通过密钥承诺解密密文的算法套件进行全面部署。

- 替换弃用元素，包括旧版 AWS KMS 主密钥提供程序的构造函数。在 Python 中，请确保打开弃用警告。最新版本 1.x 弃用的代码元素已从版本 2.0.x 及更高版本中移除。
- 将您的承诺策略明确设置为 `ForbidEncryptAllowDecrypt`。尽管这是最新版本中唯一的有效值 1. x 版本，当您使用本版本中 APIs 引入的版本时，需要此设置。当您迁移到版本 2.0.x 及更高版本时，其可防止您的应用程序拒绝未使用密钥承诺加密的加密文字。有关更多信息，请参阅 the section called “设置您的承诺策略”。
- 如果您使用 AWS KMS 主密钥提供程序，则必须将旧版主密钥提供程序更新为支持严格模式和发现模式的主密钥提供程序。 AWS Encryption SDK for Java、 AWS Encryption SDK for Python和 AWS 加密 CLI 需要进行此更新。如果您在发现模式下使用主密钥提供程序，我们建议您实施发现筛选条件，将使用的包装密钥限制为特定 AWS 账户的密钥。此项更新为可选项，但却是我们建议的最佳实践。有关更多信息，请参阅 更新 AWS KMS 主密钥提供程序。
- 如果您使用 AWS KMS Discovery 密钥环，我们建议您纳入将可用于解密的包装密钥限制为特定 AWS 账户密钥的发现筛选条件。此项更新为可选项，但却是我们建议的最佳实践。有关更多信息，请参阅 更新 AWS KMS 钥匙圈。

阶段 2：将您的应用程序更新到最新版本

所有主机成功部署最新版本 1.x 版本后，您可以升级到版本 2.0.x 及更高版本。版本 2.0。 x 包括对所有早期版本的重大更改 AWS Encryption SDK。但是，如果您按照阶段 1 的建议更改代码，可以避免在迁移到最新版本时出错。

在更新到最新版本之前，请确认您的承诺策略始终设置为 `ForbidEncryptAllowDecrypt`。然后，根据您的数据配置，您可以按照自己的节奏迁移到 `RequireEncryptAllowDecrypt`，然后迁移到默认设置 `RequireEncryptRequireDecrypt`。我们建议采取一系列过渡步骤，例如以下模式。

1. 首先，将您的承诺策略设置为 `ForbidEncryptAllowDecrypt`。 AWS Encryption SDK 可以使用密钥承诺解密消息，但尚未使用密钥承诺进行加密。
2. 准备就绪后，将承诺策略更新为 `RequireEncryptAllowDecrypt`。 AWS Encryption SDK 开始使用密钥承诺对您的数据进行加密。其可使用也可不使用密钥承诺解密加密文字。

   在将您的承诺策略更新为 `RequireEncryptAllowDecrypt` 之前，请确认您的最新版本 1.x 部署到所有主机，包括解密您生成的加密文字的任何应用程序的主机。1.7 AWS Encryption SDK 之前版本的版本。 x 无法解密使用密钥承诺加密的消息。

   这也是向应用程序添加指标的好时机，以衡量您是否仍在未使用密钥承诺处理加密文字。这将帮助您确定何时可以安全地将承诺策略设置更新为 `RequireEncryptRequireDecrypt`。对于某些应用程序，例如对 Amazon SQS 队列中的消息进行加密的应用程序，这可能意味着要等待足够长的时间才能重新加密或删除使用旧版本加密的所有加密文字。对于其他应用程序，例如加密的 S3 对象，您可能需要下载、重新加密和重新上传所有对象。
3. 如果您确定没有任何未使用密钥承诺加密的消息，则可以将承诺策略更新为 `RequireEncryptRequireDecrypt`。此值可确保您的数据始终使用密钥承诺进行加密和解密。此设置为默认设置，因此您无需明确设置，但我们建议您这样做。如果您的应用程序遇到未使用密钥承诺加密的加密文字，明确设置将协助调试和可能需要的任何潜在回滚。

更新 AWS KMS 主密钥提供程序

要迁移到最新版本 1. x 版本的 AWS Encryption SDK，然后是 2.0 版。 x 或更高版本，您必须将传统 AWS KMS 的主密钥提供程序替换为在严格模式或发现模式下显式创建的主密钥提供程序。旧版主密钥提供程序在版本 1.7.x 中弃用并在版本 2.0.x 中移除。使用 AWS Encryption SDK for Java、AWS Encryption SDK for Python 和 AWS Encryption CLI 的应用程序和脚本需要进行此项更改。本节中的示例将演示如何更新代码。

Note

在 Python 中，打开弃用警告。这将帮助您识别代码中需要更新的部分。

如果您使用的是 AWS KMS 主密钥（不是主密钥提供程序），则可以跳过此步骤。 AWS KMS 主密钥不会被弃用或删除。这些密钥仅使用您指定的包装密钥进行加密和解密。

本节中的示例重点介绍需要更改的代码元素。有关更新后的代码的完整示例，请参阅您的编程语言GitHub存储库的 “示例” 部分。此外，这些示例通常使用 key ARNs 来表示 AWS KMS keys。在创建用于加密的主密钥提供程序时，可以使用任何有效的 AWS KMS 密钥标识符来表示。 AWS KMS key 如果创建用于解密的主密钥提供程序，必须使用密钥 ARN。

了解有关迁移的更多信息

对于所有 AWS Encryption SDK 用户，请在中了解如何设置您的承诺政策the section called “设置您的承诺策略”。

对于 AWS Encryption SDK for C 和 AWS Encryption SDK for JavaScript 用户，请在中更新 AWS KMS 钥匙圈了解钥匙圈的可选更新。

主题

- 迁移到严格模式
- 迁移到发现模式

迁移到严格模式

更新到最新版本后 1. x 版本的 AWS Encryption SDK，在严格模式下，用主密钥提供程序替换旧版主密钥提供程序。在严格模式下，必须指定加密和解密时要使用的包装密钥。仅 AWS Encryption SDK 使用您指定的包装密钥。已弃用的主密钥提供程序可以使用任何加密数据密钥的提供程序 AWS KMS key 来解密数据，包括在不同的 AWS KMS keys AWS 账户 地区和区域。

1.7 AWS Encryption SDK 版本中引入了严格模式的主密钥提供程序。 x。这些主密钥提供程序取代旧版主密钥提供程序，其在 1.7.x 中弃用并在 2.0.x 中移除。在严格模式下使用主密钥提供程序是一种 AWS Encryption SDK 最佳实践。

以下代码创建可用于加密和解密的严格模式下的主密钥提供程序。

Java

此示例表示使用版本 AWS Encryption SDK for Java 1.6.2 或更早版本的应用程序中的代码。

此代码使用该`KmsMasterKeyProvider.builder()`方法来实例化使用 AWS KMS 主密钥提供程序 AWS KMS key 作为包装密钥。

```
// Create a master key provider
// Replace the example key ARN with a valid one
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .withKeysForEncryption(awsKmsKey)
    .build();
```

此示例表示使用 AWS Encryption SDK for Java 版本 1.7.x 或更高版本的应用程序中的代码。有关完整的示例，请参阅 BasicEncryptionExample.java。

前面的示例使用的 `Builder.build()` 和 `Builder.withKeysForEncryption()` 方法在版本 1.7.x 中弃用并在版本 2.0.x 中移除。

为了更新为严格模式下的主密钥提供程序，此代码将对弃用方法的调用替换为对新 `Builder.buildStrict()` 方法的调用。此示例指定一个 AWS KMS key 作为包装密钥，但该`Builder.buildStrict()`方法可以采用多个列表 AWS KMS keys。

```
// Create a master key provider in strict mode
// Replace the example key ARN with a valid one from your AWS ##.
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);
```

Python

此示例表示使用 AWS Encryption SDK for Python版本 1.4.1 的应用程序中的代码。此代码使用 `KMSMasterKeyProvider`，其在版本 1.7.x 中弃用并在版本 2.0.x 中移除。解密时，它会使用任何加密数据密钥的密钥 AWS KMS key ，而不考虑 AWS KMS keys 您指定的密钥。

请注意，`KMSMasterKey` 并未弃用或移除。加密和解密时，它仅使用您指定的。 AWS KMS key

```
# Create a master key provider
# Replace the example key ARN with a valid one
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = KMSMasterKeyProvider(
   key_ids=[key_1, key_2]
)
```

此示例表示使用 AWS Encryption SDK for Python版本 1.7.x 的应用程序中的代码。有关完整示例，请参阅 basic_encryption.py。

为了更新为严格模式下的主密钥提供程序，此代码将对 `KMSMasterKeyProvider()` 的调用替换为对 `StrictAwsKmsMasterKeyProvider()` 的调用。

```
# Create a master key provider in strict mode
# Replace the example key ARNs with valid values from your AWS ##
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2]
)
```

AWS Encryption CLI

此示例说明如何使用加密 AWS CLI 版本 1.1.7 或更早版本进行加密和解密。

在版本 1.1.7 及更早版本中，加密时，您可以在指定一个或多个主密钥（或包装密钥），例如 AWS KMS key。解密时，除非您使用的是自定义主密钥提供程序，否则无法指定任何包装密钥。 AWS 加密 CLI 可以使用任何对数据密钥进行加密的包装密钥。

```
\\ Replace the example key ARN with a valid one
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
```

```
                    --input hello.txt \
                    --master-keys key=$keyArn \
                    --metadata-output ~/metadata \
                    --encryption-context purpose=test \
                    --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                    --input hello.txt.encrypted \
                    --encryption-context purpose=test \
                    --metadata-output ~/metadata \
                    --output .
```

此示例说明如何使用加密 AWS CLI 版本 1.7 进行加密和解密。 x 或更高版本。有关完整示例，请参阅 AWS 加密 CLI 的示例。

`--master-keys` 参数在版本 1.7.x 中弃用并在版本 2.0.x 中移除。该参数由加密和解密命令所需的 `--wrapping-keys` 参数取代。该参数支持严格模式和发现模式。严格模式是一种 AWS Encryption SDK 最佳实践，可确保您使用所需的包装密钥。

要升级到严格模式，请在加密和解密时使用 `--wrapping-keys` 参数的密钥属性指定包装密钥。

```
\\ Replace the example key ARN with a valid value
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                    --input hello.txt \
                    --wrapping-keys key=$keyArn \
                    --metadata-output ~/metadata \
                    --encryption-context purpose=test \
                    --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                    --input hello.txt.encrypted \
                    --wrapping-keys key=$keyArn \
                    --encryption-context purpose=test \
                    --metadata-output ~/metadata \
                    --output .
```

迁移到发现模式

从 1.7 版本开始。 x， AWS Encryption SDK 最佳做法是对 AWS KMS 主密钥提供者使用严格模式，也就是说，在加密和解密时指定包装密钥。加密时必须始终指定包装密钥。但是在某些情况下，指定用于解密 ARNs 的 AWS KMS keys 密钥是不切实际的。例如，如果您在加密 AWS KMS keys 时使用别名进行识别，那么如果在解密时必须列出密钥 ARNs ，则会失去别名的好处。此外，由于发现模式下的主密钥提供程序的行为类似于原始主密钥提供程序，因此您可以暂时将其用作迁移策略的一部分，然后升级到严格模式下的主密钥提供程序。

在这种情况下，您可以在发现模式下使用主密钥提供程序。这些主密钥提供程序不允许您指定包装密钥，因此您不能用其进行加密。在解密时，这些提供程序可以任何使用加密数据密钥的包装密钥。但是，与行为方式相同的旧版主密钥提供程序不同，您可以在发现模式下明确创建。在发现模式下使用主密钥提供程序时，您可以将可以使用的包装密钥限制为特定 AWS 账户。此发现筛选条件为可选项，但却是我们建议的最佳实践。有关 AWS 分区和账户的信息，请参阅《AWS 一般参考》 中的 Amazon 资源名称。

以下示例在严格模式下创建用于加密 AWS KMS 的主密钥提供程序和在发现模式下创建用于解密 AWS KMS 的主密钥提供程序。发现模式下的主密钥提供程序使用发现筛选条件将用于解密的包装密钥限制在 aws 分区和特定示例 AWS 账户的密钥。尽管这一简易示例没有必要使用账户筛选条件，但是当一个应用程序加密数据而另一个应用程序解密数据时，这是一种非常有益的最佳实践。

Java

此示例表示使用 AWS Encryption SDK for Java版本 1.7.x 或更高版本的应用程序中的代码。有关完整的示例，请参阅 DiscoveryDecryptionExample.java。

为了实例化用于加密的严格模式下的主密钥提供程序，此示例使用 `Builder.buildStrict()` 方法。为了实例化用于解密的发现模式下的主密钥提供程序，此示例使用 `Builder.buildDiscovery()` 方法。该`Builder.buildDiscovery()`方法采用`DiscoveryFilter`，将限制 AWS Encryption SDK AWS KMS keys 为指定 AWS 分区和帐户。

```
// Create a master key provider in strict mode for encrypting
// Replace the example alias ARN with a valid one from your AWS ##.
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias";

KmsMasterKeyProvider encryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Create a master key provider in discovery mode for decrypting
// Replace the example account IDs with valid values.
```

```
DiscoveryFilter accounts = new DiscoveryFilter("aws", Arrays.asList("111122223333",
 "444455556666"));

KmsMasterKeyProvider decryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildDiscovery(accounts);
```

Python

此示例表示使用 AWS Encryption SDK for Python 版本 1.7.x 或更高版本的应用程序中的代码。有关完整示例，请参阅 discovery_kms_provider.py。

为了在创建用于加密的严格模式下的主密钥提供程序，此示例使用 StrictAwsKmsMasterKeyProvider。要在发现模式下创建用于解密的主密钥提供程序，DiscoveryFilter该提供程序 AWS Encryption SDK 将DiscoveryAwsKmsMasterKeyProvider与限制 AWS KMS keys 在指定的 AWS 分区和帐户中。

```
# Create a master key provider in strict mode
# Replace the example key ARN and alias ARNs with valid values from your AWS ##.
key_1 = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias"
key_2 = "arn:aws:kms:us-
west-2:444455556666:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2]
)

# Create a master key provider in discovery mode for decrypting
# Replace the example account IDs with valid values
accounts = DiscoveryFilter(
    partition="aws",
    account_ids=["111122223333", "444455556666"]
)
aws_kms_master_key_provider = DiscoveryAwsKmsMasterKeyProvider(
        discovery_filter=accounts
)
```

AWS Encryption CLI

此示例说明如何使用加密 AWS CLI 版本 1.7 进行加密和解密。 x 或更高版本。从版本 1.7.x 开始，加密和解密时需要使用 --wrapping-keys 参数。--wrapping-keys 参数支持严格模式和发现模式。有关完整示例，请参阅 the section called “示例”。

加密时，此示例指定包装密钥，这是必需的。解密时，此示例使用值为 `true` 的 `--wrapping-keys` 参数的 `discovery` 属性明确选择发现模式。

为了将 AWS Encryption SDK 可以在发现模式下使用的包装密钥限制为特定的封装密钥 AWS 账户，此示例使用了`--wrapping-keys`参数的`discovery-partition`和`discovery-account`属性。这些可选属性仅在 `discovery` 属性设置为 `true` 时有效。必须同时使用 `discovery-partition` 和 `discovery-account` 属性；单独使用则无效。

```
\\ Replace the example key ARN with a valid value
$ keyAlias=arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyAlias \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
\\ Replace the example account IDs with valid values
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-partition=aws \
                                     discovery-account=111122223333 \
                                     discovery-account=444455556666 \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

更新 AWS KMS 钥匙圈

中的 AWS KMS 密钥环AWS Encryption SDK for C、AWS Encryption SDK 适用于.NET 的密钥圈以及允许您在加密和解密时指定包装密钥来AWS Encryption SDK for JavaScript支持最佳实践。如果您创建了 AWS KMS Discovery 密钥环，需要明确执行此操作。

Note

.NET AWS Encryption SDK 的最早版本是 3.0 版。 x。.NET 的所有版本都支持 2.0 中引入的安全最佳实践。 AWS Encryption SDK 的 x 个 AWS Encryption SDK。无需更改任何代码或数据即可安全升级到最新版本。

当你更新到最新版本时 1. x 版本的 AWS Encryption SDK，您可以使用发现过滤器将发现密钥环或 AWS KMS 区域发现密钥环在解密时使用的包装密钥限制为特定的封装密钥。AWS KMS AWS 账户筛选发现密钥环是 AWS Encryption SDK 最佳做法。

本节中的示例将演示如何向 AWS KMS Regional Discovery 密钥环添加发现筛选条件。

了解有关迁移的更多信息

对于所有 AWS Encryption SDK 用户，请在中了解如何设置您的承诺政策the section called “设置您的承诺策略”。

对于 AWS Encryption SDK for Java AWS Encryption SDK for Python、和 AWS 加密 CLI 用户，请在中了解主密钥提供程序所需的更新the section called “更新 AWS KMS 主密钥提供程序”。

您的应用程序中可能有如下类似代码。此示例创建 AWS KMS Regional Discovery 密钥环，该密钥环仅能在美国西部（俄勒冈州）（us-west-2）区域使用包装密钥。此示例表示 1.7 之前 AWS Encryption SDK 版本中的代码。 x。但其在版本 1.7.x 及更高版本中仍然有效。

C

```
struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()
       .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery());
```

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser({ clientProvider, discovery })
```

JavaScript Node.js

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({ clientProvider, discovery })
```

从 1.7 版本开始。 x，您可以向任何发现密钥环添加 AWS KMS 发现过滤器。此发现过滤器将 AWS KMS keys AWS Encryption SDK 可用于解密的限制为指定分区和帐户中的分区和帐户。在使用此代码之前，如有必要，请更改分区，并将示例帐户 IDs 替换为有效的帐户。

C

有关完整示例，请参阅：kms_discovery.cpp。

```
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .AddAccount("444455556666")
        .Build());

struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()

 .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery(discovery_filter))
```

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

JavaScript Node.js

有关完整示例，请参阅：kms_filtered_discovery.ts。

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

设置您的承诺策略

密钥承诺确保您的加密数据始终解密为相同的明文。要提供此安全属性，请从 1.7 版本开始。 x，AWS Encryption SDK 使用带有密钥承诺的新算法套件。要确定您的数据是否使用密钥承诺进行加密和解密，请使用承诺策略配置设置。使用密钥承诺加密和解密数据是 AWS Encryption SDK 最佳实践。

设置承诺政策是迁移过程第二步的重要组成部分，即从最新版本迁移 1. x 版本是 AWS Encryption SDK to 版本 2.0。 x 及更高版本。设置并更改承诺策略后，请确保在将应用程序部署到生产环境之前对其进行全面测试。有关迁移指南，请参阅 如何迁移和部署 AWS Encryption SDK。

在版本 2.0.x 及更高版本中，承诺策略设置有三个有效值。在最新版本 1.x（从版本 1.7.x 开始），仅 `ForbidEncryptAllowDecrypt` 有效。

- `ForbidEncryptAllowDecrypt`— AWS Encryption SDK 无法使用密钥承诺进行加密。可以解密使用或不使用密钥承诺加密的加密文字。

  在最新版本 1.x 中，这是唯一的有效值。其可确保您在完全准备好使用密钥承诺进行解密之前不会使用密钥承诺进行加密。明确设置该值可防止您的承诺策略在升级到版本 2.0.x 或更高版本时自动更改为 `require-encrypt-require-decrypt`。相反，您可以分阶段迁移承诺策略。
- `RequireEncryptAllowDecrypt`— AWS Encryption SDK 始终使用密钥承诺进行加密。可以解密使用或不使用密钥承诺加密的加密文字。版本 2.0.x 中加入了此值。
- `RequireEncryptRequireDecrypt`— AWS Encryption SDK 始终使用密钥承诺进行加密和解密。版本 2.0.x 中加入了此值。在版本 2.0.x 和更高版本中，这是默认值。

在最新版本 1.x 中，唯一有效的承诺策略值为 `ForbidEncryptAllowDecrypt`。迁移到版本 2.0.x 或更高版本之后，您可以在准备就绪后分阶段更改承诺策略。除非您确定所有消息均使用密钥承诺加密，否则请勿将您的承诺策略更新为 `RequireEncryptRequireDecrypt`。

这些示例演示了如何在最新版本 1.x、版本 2.0.x 及更高版本中设置承诺策略。该方法取决于您的编程语言。

了解有关迁移的更多信息

对于 AWS Encryption SDK for Java AWS Encryption SDK for Python、和 AWS Encryption CLI，请在中了解主密钥提供程序所需的更改the section called “更新 AWS KMS 主密钥提供程序”。

对于 AWS Encryption SDK for C 和 AWS Encryption SDK for JavaScript，请在中了解钥匙圈的可选更新。更新 AWS KMS 钥匙圈

如何设置您的承诺策略

您使用的承诺策略设置方法因语言实施而略有不同。这些示例演示了如何进行操作。在更改承诺策略之前，请查看 如何迁移和部署 中的多阶段方法。

C

从 1.7 版本开始。 其中 x AWS Encryption SDK for C，您可以使用`aws_cryptosdk_session_set_commitment_policy`函数为加密和解密会话设置承诺策略。您设置的承诺策略适用于在该会话中调用的所有加密和解密操作。

`aws_cryptosdk_session_new_from_keyring` 和 `aws_cryptosdk_session_new_from_cmm` 函数在版本 1.7.x 中弃用并在版本 2.0.x 中移除。这些函数由返回会话的 `aws_cryptosdk_session_new_from_keyring_2` 和 `aws_cryptosdk_session_new_from_cmm_2` 函数取代。

在最新版本 1.x 中使用 `aws_cryptosdk_session_new_from_keyring_2` 和 `aws_cryptosdk_session_new_from_cmm_2` 时，需要使用 `COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT` 承诺策略值调用 `aws_cryptosdk_session_set_commitment_policy` 函数。在版本 2.0.x 及更高版本中，调用此函数为可选项，需要所有有效值。版本 2.0.x 及更高版本的默认承诺策略为 `COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。

有关完整的示例，请参阅 string.cpp。

```
/* Load error strings for debugging */
aws_cryptosdk_load_error_strings();

/* Create an AWS KMS keyring */
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
```

```
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* Create an encrypt session with a CommitmentPolicy setting */
struct aws_cryptosdk_session *encrypt_session =
 aws_cryptosdk_session_new_from_keyring_2(
    alloc, AWS_CRYPTOSDK_ENCRYPT, kms_keyring);

aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(encrypt_session,
    COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

...
/* Encrypt your data */

size_t plaintext_consumed_output;
aws_cryptosdk_session_process(encrypt_session,
                              ciphertext_output,
                              ciphertext_buf_sz_output,
                              ciphertext_len_output,
                              plaintext_input,
                              plaintext_len_input,
                              &plaintext_consumed_output)
...

/* Create a decrypt session with a CommitmentPolicy setting */

struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
struct aws_cryptosdk_session *decrypt_session =
 *aws_cryptosdk_session_new_from_keyring_2(
        alloc, AWS_CRYPTOSDK_DECRYPT, kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(decrypt_session,
        COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

/* Decrypt your ciphertext */
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(decrypt_session,
                              plaintext_output,
                              plaintext_buf_sz_output,
                              plaintext_len_output,
                              ciphertext_input,
                              ciphertext_len_input,
```

```
&ciphertext_consumed_output)
```

C# / .NET

该require-encrypt-require-decrypt值是 AWS Encryption SDK 适用于.NET 的所有版本中的默认承诺策略。您可以将其明确设置为最佳实践，但这并非必需。但是，如果您使用 for .NET 来解密由 AWS Encryption SDK 无密钥承诺的另一种语言实现加密的密文，则需要将承诺策略值更改为或。 AWS Encryption SDK REQUIRE_ENCRYPT_ALLOW_DECRYPT FORBID_ENCRYPT_ALLOW_DECRYPT否则，加密文字解密尝试会失败。

在 f AWS Encryption SDK or .NET 中，您可以为的实例设置承诺策略 AWS Encryption SDK。使用CommitmentPolicy参数实例化AwsEncryptionSdkConfig对象，然后使用配置对象创建实例。 AWS Encryption SDK 然后，调用已配置 AWS Encryption SDK 实例的Encrypt()和Decrypt()方法。

此示例将承诺策略设置为 require-encrypt-allow-decrypt。

```
// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    CommitmentPolicy = CommitmentPolicy.REQUIRE_ENCRYPT_ALLOW_DECRYPT
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

string keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"}encryptionSdk
};

var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

```
var keyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);

// Decrypt your ciphertext
var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = keyring
};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

AWS Encryption CLI

要在 AWS 加密 CLI 中设置承诺策略，请使用`--commitment-policy`参数。此参数在版本 1.8.x 中引入。

在最新版本 1.x 中，当您在 `--encrypt` 或 `--decrypt` 命令中使用 `--wrapping-keys` 参数时，需要有 `forbid-encrypt-allow-decrypt` 值的 `--commitment-policy` 参数。否则 `--commitment-policy` 参数无效。

在版本 2.1.x 及更高版本中，`--commitment-policy` 参数为可选项，默认为 `require-encrypt-require-decrypt` 值，其不会加密或解密任何不使用密钥承诺加密的加密文字。但是，我们建议您在所有加密和解密调用中明确设置承诺策略，以便进行维护和故障排除。

此示例设置了承诺策略。从版本 1.8.x 开始，此示例还使用取代 `--master-keys` 参数的 `--wrapping-keys` 参数。有关更多信息，请参阅 the section called “更新 AWS KMS 主密钥提供程序”。有关完整示例，请参阅 AWS 加密 CLI 的示例。

```
\\ To run this example, replace the fictitious key ARN with a valid value.
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data - no change to algorithm suite used
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
```

```
                      --wrapping-keys key=$keyArn \
                      --commitment-policy forbid-encrypt-allow-decrypt \
                      --metadata-output ~/metadata \
                      --encryption-context purpose=test \
                      --output .

\\ Decrypt your ciphertext - supports key commitment on 1.7 and later
$ aws-encryption-cli --decrypt \
                      --input hello.txt.encrypted \
                      --wrapping-keys key=$keyArn \
                      --commitment-policy forbid-encrypt-allow-decrypt \
                      --encryption-context purpose=test \
                      --metadata-output ~/metadata \
                      --output .
```

Java

从 1.7 版本开始。 其中 x AWS Encryption SDK for Java，您对代表 AWS Encryption SDK 客户端的对象的`AwsCrypto`实例设置了承诺策略。该承诺策略设置适用于在该客户端中调用的所有加密和解密操作。

在最新的 1 版本中，该`AwsCrypto()`构造函数已被弃用。 在 2.0 版本中移除了 AWS Encryption SDK for Java 和的 x 版本。 x。该构造函数由新的 `Builder` 类、`Builder.withCommitmentPolicy()` 方法和 `CommitmentPolicy` 枚举类型取代。

在最新版本 1.x 中，`Builder` 类需要 `Builder.withCommitmentPolicy()` 方法和 `CommitmentPolicy.ForbidEncryptAllowDecrypt` 参数。从版本 2.0.x 开始，`Builder.withCommitmentPolicy()` 方法为可选项；默认值为 `CommitmentPolicy.RequireEncryptRequireDecrypt`。

有关完整的示例，请参阅 SetCommitmentPolicyExample.java。

```
// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.ForbidEncryptAllowDecrypt)
    .build();

// Create a master key provider in strict mode
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
```

```
    .buildStrict(awsKmsKey);

// Encrypt your plaintext data
CryptoResult<byte[], KmsMasterKey> encryptResult = crypto.encryptData(
    masterKeyProvider,
    sourcePlaintext,
    encryptionContext);
byte[] ciphertext = encryptResult.getResult();

// Decrypt your ciphertext
CryptoResult<byte[], KmsMasterKey> decryptResult = crypto.decryptData(
        masterKeyProvider,
        ciphertext);
byte[] decrypted = decryptResult.getResult();
```

JavaScript

从 1.7 版本开始。 x 中 AWS Encryption SDK for JavaScript，您可以在调用实例化客户端的新`buildClient`函数时设置承诺策略。 AWS Encryption SDK `buildClient` 函数需要表示承诺策略的枚举值。进行加密和解密时，该函数会返回强制执行您的承诺策略的已更新 `encrypt` 和 `decrypt` 函数。

在最新版本 1.x 中，`buildClient` 函数需要 `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT` 参数。从版本 2.0.x 开始，承诺策略参数为可选项，默认值为 `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。

Node.js 和浏览器的代码相同，唯一区别在于浏览器需要使用语句设置凭证。

以下示例使用密 AWS KMS 钥环对数据进行加密。新的 `buildClient` 函数将承诺策略设置为 `FORBID_ENCRYPT_ALLOW_DECRYPT`，其为最新版本 1.x 的默认值。`buildClient` 返回的已升级 `encrypt` 和 `decrypt` 函数强制执行您设置的承诺策略。

```
import { buildClient } from '@aws-crypto/client-node'
const { encrypt, decrypt } =
 buildClient(CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT)

// Create an AWS KMS keyring
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })
```

```
// Encrypt your plaintext data
const { ciphertext } = await encrypt(keyring, plaintext, { encryptionContext:
 context })

// Decrypt your ciphertext
const { decrypted, messageHeader } = await decrypt(keyring, ciphertext)
```

Python

从 1.7 版本开始。 其中 x AWS Encryption SDK for Python，您对代表 AWS Encryption SDK 客户端的新对象的`EncryptionSDKClient`实例设置了承诺策略。您设置的承诺策略适用于所有使用该客户端实例的 `encrypt` 和 `decrypt` 调用。

在最新版本 1.x 中，`EncryptionSDKClient` 构造函数需要 `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT` 枚举值。从版本 2.0.x 开始，承诺策略参数为可选项，默认值为 `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。

此示例使用新的 `EncryptionSDKClient` 构造函数并将承诺策略设置为 1.7.x 默认值。构造函数实例化表示 AWS Encryption SDK的客户端。当您在此客户端上调用 `encrypt`、`decrypt` 或 `stream` 方法时，这些方法会强制执行您设置的承诺策略。此示例还使用了`StrictAwsKmsMasterKeyProvider`类的新构造函数，该构造函数指定了 AWS KMS keys 何时加密和解密。

有关完整示例，请参阅 set_commitment.py。

```
# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_AL

// Create a master key provider in strict mode
aws_kms_key = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
aws_kms_strict_master_key_provider = StrictAwsKmsMasterKeyProvider(
        key_ids=[aws_kms_key]
)

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
        source=source_plaintext,
        encryption_context=encryption_context,
        master_key_provider=aws_kms_strict_master_key_provider
)
```

```
# Decrypt your ciphertext
decrypted, decrypt_header = client.decrypt(
        source=ciphertext,
        master_key_provider=aws_kms_strict_master_key_provider
)
```

Rust

该`require-encrypt-require-decrypt`值是 for Rust 所有版本中的 AWS Encryption SDK 默认承诺策略。您可以将其明确设置为最佳实践，但这并非必需。但是，如果您使用 for Rust 来解密由 AWS Encryption SDK 无密钥承诺的另一种语言实现加密的密文，则需要将承诺策略值更改为或。 AWS Encryption SDK `REQUIRE_ENCRYPT_ALLOW_DECRYPT` `FORBID_ENCRYPT_ALLOW_DECRYPT`否则，加密文字解密尝试会失败。

在 f AWS Encryption SDK or Rust 中，您可以为的实例设置承诺策略 AWS Encryption SDK。使用`comitment_policy`参数实例化`AwsEncryptionSdkConfig`对象，然后使用配置对象创建实例。 AWS Encryption SDK 然后，调用已配置 AWS Encryption SDK 实例的`Encrypt()`和`Decrypt()`方法。

此示例将承诺策略设置为 `forbid-encrypt-allow-decrypt`。

```
// Configure the commitment policy on the AWS Encryption SDK instance
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .commitment_policy(ForbidEncryptAllowDecrypt)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

// Create an AWS KMS client
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

// Create your encryption context
let encryption_context = HashMap::from([
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);
```

```
// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create an AWS KMS keyring
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;

// Encrypt your plaintext data
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

// Decrypt your ciphertext
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
```

```
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

// Instantiate the AWS Encryption SDK client
commitPolicyForbidEncryptAllowDecrypt :=
 mpltypes.ESDKCommitmentPolicyForbidEncryptAllowDecrypt
encryptionClient, err :=
 client.NewClient(esdktypes.AwsEncryptionSdkConfig{CommitmentPolicy:
 &commitPolicyForbidEncryptAllowDecrypt})
if err != nil {
    panic(err)
}

// Create an AWS KMS client
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}

// Create an AWS KMS keyring
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  kmsKeyId,
```

```
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}

// Encrypt your plaintext data
res, err := forbidEncryptClient.Encrypt(context.Background(),
 esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}

// Decrypt your ciphertext
decryptOutput, err := forbidEncryptClient.Decrypt(context.Background(),
 esdktypes.DecryptInput{
    Ciphertext:        res.Ciphertext,
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

对迁移到至最新版本进行故障排除

在将应用程序更新到 2.0 版之前。 x 或更高版本 AWS Encryption SDK，更新到最新的 1。 x 版本的，AWS Encryption SDK 然后将其完全部署。这将有助于避免在更新到版本 2.0.x 及更高版本时可能遇到的多数错误。有关详细指南（包括示例），请参阅 迁移你的 AWS Encryption SDK。

> **Important**
>
> 验证最新版本 1.x 为 AWS Encryption SDK版本 1.7.x 或更高版本。

> **Note**
>
> AWS 加密 CLI：本指南中对版本 1.7 的参考。 其中的 x AWS Encryption SDK 适用于版本 1.8。 AWS 加密 CLI 中的 x。本指南中对版本 2.0 的引用。 其中的 x AWS Encryption SDK 适用于 2.1。 AWS 加密 CLI 中的 x。
> 新的安全功能最初是在 AWS 加密 CLI 版本 1.7 中发布的。 x 和 2.0。 x。但是， AWS 加密 CLI 版本为 1.8。 x 取代了 1.7 版。 x 和 AWS 加密 CLI 2.1。 x 取代 2.0。 x。有关详细信息，请参阅aws-encryption-sdk-cli存储库中的相关安全公告 GitHub。

本主题旨在帮助您识别和解决可能遇到的最常见错误。

主题

- 弃用或移除的对象
- 配置冲突：承诺策略和算法套件
- 配置冲突：承诺策略和加密文字
- 密钥承诺验证失败
- 其他加密故障
- 其他加密故障
- 回滚注意事项

弃用或移除的对象

版本 2.0.x 包含几项重大更改，包括移除版本 1.7.x 中弃用的旧版构造函数、方法、函数和类。为避免编译器错误、导入错误、语法错误和未找到符号错误（取决于您的编程语言），请先升级到最新的 1。AWS Encryption SDK 适用于您的编程语言的 x 版本。（此版本必须为版本 1.7.x 或更高版本。）在使用最新版本 1.x 时，您可以在移除原始符号之前开始使用替换元素。

如果需要立即升级到版本 2.0.x 或更高版本，请查阅您的编程语言的更改日志，然后用更改日志建议的符号替换旧版符号。

配置冲突：承诺策略和算法套件

如果您指定的算法套件与您的承诺策略冲突，加密调用会失败并出现配置冲突错误。

为避免此类错误，请勿指定算法套件。默认情况下， AWS Encryption SDK 会选择与您的承诺策略兼容的最安全的算法。但是，如果您必须指定算法套件（例如不具有签名的算法套件），请确保选择与您的承诺策略兼容的算法套件。

承诺策略	兼容算法套件
ForbidEncryptAllowDecrypt	任何不具有密钥承诺的算法套件，例如： AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384 （03 78）（具有签名） AES_256_GCM_IV12_TAG16_HKDF_SHA256 （01 78）（不具有签名）
RequireEncryptAllowDecrypt RequireEncryptRequireDecrypt	任何具有密钥承诺的算法套件，例如： AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384 （05 78）（具有签名） AES_256_GCM_HKDF_SHA512_COMMIT_KEY （04 78）（不具有签名）

如果您在未指定算法套件时遇到此错误，则可能是您的加密材料管理器 （CMM）选择了冲突算法套件。默认 CMM 不会选择冲突算法套件，但自定义 CMM 可能会。如需帮助，请参阅自定义 CMM 相关文档。

配置冲突：承诺策略和加密文字

`RequireEncryptRequireDecrypt`承诺策略不允许 AWS Encryption SDK 解密未使用密钥承诺加密的消息。如果您要求解 AWS Encryption SDK 密未承诺密钥的消息，则会返回配置冲突错误。

为避免此错误，在设置 `RequireEncryptRequireDecrypt` 承诺策略之前，请确保所有未使用密钥承诺加密的加密文字均使用密钥承诺进行解密和重新加密，或者由其他应用程序处理。如果您遇到此错误，可以针对冲突加密文字返回错误，或者将您的承诺策略暂时更改为 `RequireEncryptAllowDecrypt`。

如果您因为在未首先升级到最新版本 1.x（1.7.x 或更高版本）的情况下，从 1.7.x 之前的版本升级到版本 2.0.x 或更高版本遇到此错误，可以考虑回滚到最新版本 1.x，并在升级到版本 2.0.x 或更高版本之前将该版本部署到所有主机。有关帮助信息，请参阅 如何迁移和部署 AWS Encryption SDK。

密钥承诺验证失败

解密使用密钥承诺加密的消息时，可能会收到密钥承诺验证失败的错误消息。这表示解密调用失败，因为加密消息中的数据密钥与消息的唯一数据密钥不同。通过在解密过程中验证数据密钥，密钥承诺可以避免解密可能生成多个明文的消息。

此错误表示 AWS Encryption SDK未返回您尝试解密的加密消息。此错误可能是手动制作的消息或数据损坏的结果。如果您遇到此错误，您的应用程序可能会拒绝该消息并继续，或者停止处理新消息。

其他加密故障

加密故障原因多种多样。您不可使用 AWS KMS Discovery 密钥环或发现模式下的主密钥提供程序加密消息。

请确保指定包含您有权用其加密的包装密钥的密钥环或主密钥提供程序。有关权限的帮助 AWS KMS keys，请参阅《AWS Key Management Service 开发者指南》 AWS KMS key中的查看密钥策略和确定对的访问权限。

其他加密故障

如果加密消息解密尝试失败，则意味着 AWS Encryption SDK 无法（或不会）解密消息中的任何加密数据密钥。

如果您使用的是指定包装密钥的密钥环或主密钥提供程序，则仅 AWS Encryption SDK 使用您指定的包装密钥。请确认您使用的是打算使用且拥有至少一个包装密钥 `kms:Decrypt` 权限的包装密钥。如果您使用的是备用方法 AWS KMS keys，则可以尝试在发现模式下使用AWS KMS 发现密钥环或主密钥提供程序来解密消息。如果操作成功，在返回明文之前，请验证用于解密消息的密钥是否是您信任的密钥。

回滚注意事项

如果您的应用程序无法加密或解密数据，则通常可以通过更新代码符号、密钥环、主密钥提供程序或承诺策略来解决问题。但是，在某些情况下，您可能会决定最好将应用程序回滚到 AWS Encryption SDK 先前版本。

如果必须回滚，请谨慎操作。1.7 AWS Encryption SDK 之前的版本。 x 无法解密使用密钥承诺加密的密文。

- 从最新版本 1.x 回滚到 AWS Encryption SDK 先前版本通常是安全的。您可能需要撤消对代码所做更改才能使用先前版本不支持的符号和对象。
- 在版本 2.0.x 或更高版本中开始使用密钥承诺进行加密后（将您的承诺策略设置为 `RequireEncryptAllowDecrypt`），您可以回滚到版本 1.7.x 而非任何早期版本。1.7 AWS Encryption SDK 之前的版本。 x 无法解密使用密钥承诺加密的密文。

如果您在所有主机均可使用密钥承诺解密之前意外启用了使用密钥承诺进行加密，则最好继续推出而非回滚。如果消息是暂时的，或者可以安全删除，则可以考虑在消息丢失的情况下进行回滚。如果需要回滚，您可以考虑编写解密和重新加密所有消息的工具。

常见问题

常见问题

- 和？有何 AWS Encryption SDK 不同 AWS SDKs？
- 与 Amazon S3 加密客户端有何 AWS Encryption SDK 不同？
- 支持哪些加密算法 AWS Encryption SDK，哪一种是默认算法？
- 如何生成初始化向量 (IV) 以及将其存储在何处？
- 如何生成、加密和解密每个数据密钥？
- 如何跟踪用于加密我的数据的数据密钥？
- 如何将加密的数据密钥与其加密数据一起 AWS Encryption SDK 存储？
- AWS Encryption SDK 消息格式会给我的加密数据增加多少开销？
- 我是否可以使用自己的主密钥提供程序？
- 我是否可以使用多个包装密钥加密数据？
- 我可以使用哪些数据类型进行加密 AWS Encryption SDK？
- AWS Encryption SDK 加密和解密 input/output (I/O) 流是如何进行的？

和？有何 AWS Encryption SDK 不同 AWS SDKs？

AWS SDKs提供了用于与 Amazon Web Services 交互的库 (AWS)，包括 AWS Key Management Service (AWS KMS)。的某些语言实现（例如AWS Encryption SDK 适用于.NET 的）始终要求使用相同的编程语言的 AWS SDK。 AWS Encryption SDK只有当您在密钥环或主密钥提供程序中使用 AWS KMS 密钥时，其他语言实现才需要相应的 AWS SDK。有关详细信息，请参阅 AWS Encryption SDK 编程语言 中有关您的编程语言的主题。

您可以使用 AWS SDKs 进行交互 AWS KMS，包括加密和解密少量数据（使用对称加密密钥最多为 4,096 字节），以及生成用于客户端加密的数据密钥。但是，生成数据密钥时，必须管理整个加密和解密过程，包括使用外部的数据密钥加密数据、安全地丢弃纯文本数据密钥 AWS KMS、存储加密的数据密钥，然后解密数据密钥和解密数据。 AWS Encryption SDK 为您处理此过程。

AWS Encryption SDK 提供了一个使用行业标准和最佳实践对数据进行加密和解密的库。其生成数据密钥，使用您指定的包装密钥对其进行加密，然后返回加密消息，其为包含加密数据和解密所需的加密数据密钥的便携式数据对象。到了解密的时候，你传入加密的消息和至少一个包装密钥（可选），然后 AWS Encryption SDK 返回你的纯文本数据。

您可以在中 AWS KMS keys 用作包装密钥 AWS Encryption SDK，但这不是必需的。您可以使用自己生成的加密密钥以及来自密钥管理器或本地硬件安全模块的加密密钥。 AWS Encryption SDK 即使您没有 AWS 帐户，也可以使用。

与 Amazon S3 加密客户端有何 AWS Encryption SDK 不同？

中的 Amazon S3 加密客户端为您存储在亚马逊简单存储服务 (Amazon S3) Service 中的数据 AWS SDKs 提供加密和解密功能。这些客户端与 Amazon S3 紧密耦合在一起，并且仅适用于在该位置中存储的数据。

为可以存储在任何地方的数据 AWS Encryption SDK 提供加密和解密功能。 AWS Encryption SDK 和 Amazon S3 加密客户端不兼容，因为它们生成数据格式不同的密文。

支持哪些加密算法 AWS Encryption SDK，哪一种是默认算法？

AWS Encryption SDK 使用高级加密标准 (AES) Galois/Counter 模式下的对称算法 (GCM)（称为 AES-GCM）来加密您的数据。其允许您从多种对称和非对称算法中进行选择，以便对加密数据的数据密钥进行加密。

对于 AES-GCM，默认算法套件是 AES-GCM，具有 256 位密钥、密钥派生 (HKDF)、数字签名和密钥承诺。 AWS Encryption SDK 还支持 192 位和 128 位加密密钥以及加密算法，无需数字签名和密钥承诺。

在所有情况下，初始化向量 (IV) 长度为 12 个字节；身份验证标签长度为 16 个字节。默认情况下，SDK 使用数据密钥作为基于 HMAC 的密 extract-and-expand钥派生函数 (HKDF) 的输入来派生 AES-GCM 加密密钥，还会添加椭圆曲线数字签名算法 (ECDSA) 签名。

有关选择要使用的算法的信息，请参阅支持的算法套件。

有关支持的算法的实施详细信息，请参阅算法参考。

如何生成初始化向量 (IV) 以及将其存储在何处？

AWS Encryption SDK 使用确定性方法为每帧构造不同的 IV 值。此过程可 IVs 确保消息中永远不会重复。（在 AWS Encryption SDK for Java 和的 1.3.0 版本之前 AWS Encryption SDK for Python，它们会为每帧 AWS Encryption SDK 随机生成一个唯一的 IV 值。）

IV 存储在 AWS Encryption SDK 返回的加密消息中。有关更多信息，请参阅AWS Encryption SDK 消息格式参考。

如何生成、加密和解密每个数据密钥？

方法取决于您使用的密钥环或主密钥提供程序。

中的 AWS KMS 密钥环和主密钥提供程序 AWS Encryption SDK 使用 AWS KMS GenerateDataKeyAPI 操作生成每个数据密钥，并在其包装密钥下对其进行加密。要加密其他 KMS 密钥下的数据密钥副本，他们使用 AWS KMS Encrypt 操作。要解密数据密钥，他们使用 AWS KMS 解密操作。有关详细信息，请参阅中的 " AWS Encryption SDK 规范" 中的AWS KMS 密钥环。 GitHub

其他密钥环使用各种编程语言的最佳实践方法生成数据密钥并进行加密和解密。有关详细信息，请参阅《规范》的 "框架" 部分中的密钥环或主密钥提供程序 AWS Encryption SDK 规范。 GitHub

如何跟踪用于加密我的数据的数据密钥？

他们为你 AWS Encryption SDK 做这件事。在加密数据时，该开发工具包加密数据密钥，并将加密的密钥与加密的数据一起存储在它返回的加密的消息中。在解密数据时， AWS Encryption SDK 从加密的消息中提取加密的数据密钥，对其进行解密，然后使用该密钥解密数据。

如何将加密的数据密钥与其加密数据一起 AWS Encryption SDK 存储？

中的加密操作会 AWS Encryption SDK 返回一条加密消息，即包含加密数据及其加密数据密钥的单一数据结构。消息格式包含至少两个部分：标头 和正文。消息标头包含加密的数据密钥以及有关消息正文组成方式的消息。消息正文包含加密的数据。如果算法套件包含数字签名，则消息格式包括包含签名的页脚。有关更多信息，请参阅 AWS Encryption SDK 消息格式参考。

AWS Encryption SDK 消息格式会给我的加密数据增加多少开销？

增加的开销金额 AWS Encryption SDK 取决于多个因素，包括以下因素：

- 明文数据的大小
- 使用哪些支持的算法
- 是否提供其他经过身份验证的数据 (AAD) 以及该 AAD 的长度
- 包装密钥或主密钥的数量和类型
- 帧大小（在使用帧数据时）

当您使用默认配置（一个 AWS KMS key 作为包装密钥（或主密钥）、无 AAD、非成帧数据和带签名的加密算法）时，开销约为 600 字节。 AWS Encryption SDK 一般来说，您可以合理地假设 AWS Encryption SDK 增加 1 KB 或更少的开销，不包括提供的 AAD。有关更多信息，请参阅 AWS Encryption SDK 消息格式参考。

我是否可以使用自己的主密钥提供程序？

可以。根据您使用的支持的编程语言，实现细节会有所不同。但是，所有支持的语言都允许您定义自定义加密材料管理器 (CMMs) Ms)、主密钥提供程序、密钥环、主密钥和包装密钥。

我是否可以使用多个包装密钥加密数据？

可以。您可以使用附加包装密钥（或主密钥）加密数据密钥，以便在位于不同区域或无法用于解密时提供冗余。

要使用多个包装密钥加密数据，请创建一个具有多个包装密钥的密钥环或主密钥提供程序。在使用密钥环时，您可以创建一个具有多个包装密钥的密钥环或多重密钥环。

当您使用多个包装密钥加密数据时，会 AWS Encryption SDK 使用一个包装密钥生成纯文本数据密钥。数据密钥是唯一的，在数学上与包装密钥无关。此操作会返回明文数据密钥以及由包装密钥加密的数据密钥的副本。然后是加密方法，即使用其他包装密钥加密数据密钥。生成的加密消息包含加密的数据以及加密的数据密钥，每个包装密钥具有一个加密的数据密钥。

加密后的消息可以使用加密操作中使用的任何一个包装密钥来解密。 AWS Encryption SDK 使用包装密钥来解密加密的数据密钥。然后，它使用明文数据密钥以解密数据。

我可以使用哪些数据类型进行加密 AWS Encryption SDK？

的大多数编程语言实现 AWS Encryption SDK 都可以加密原始字节（字节数组）、 I/O 流（字节流）和字符串。.N AWS Encryption SDK ET 版不支持 I/O 直播。我们为每种支持的编程语言提供了示例代码。

AWS Encryption SDK 加密和解密 input/output (I/O) 流是如何进行的？

AWS Encryption SDK 创建封装底层流的加密或解密流。 I/O 加密或解密流对读取或写入调用执行加密操作。例如，它可以读取基础流上的明文数据，并在返回结果之前对其进行加密。或者，它可以从基础

流中读取密文，并在返回结果之前对其进行解密。我们为每种支持流式传输的支持的编程语言提供了加密和解密流的示例代码。

.N AWS Encryption SDK ET 版不支持 I/O 直播。

AWS Encryption SDK 引用

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

AWS Encryption SDK 使用支持的算法返回包含加密数据和相应加密数据密钥的单个数据结构或消息。以下主题介绍了这些算法和数据结构。可以使用该信息构建一些库，它们可以读取和写入与该开发工具包兼容的密文。

主题

- AWS Encryption SDK 消息格式参考
- AWS Encryption SDK 消息格式示例
- 正文其他经过身份验证的数据 (AAD) 参考 AWS Encryption SDK
- AWS Encryption SDK 算法参考
- AWS Encryption SDK 初始化向量参考
- AWS KMS 分层钥匙圈技术细节

AWS Encryption SDK 消息格式参考

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

中的加密操作 AWS Encryption SDK 返回包含加密数据（密文）和所有加密数据密钥的单个数据结构或加密消息。要了解该数据结构或构建读取和写入该结构的库，您需要了解消息格式。

消息格式包含至少两个部分：标头 和正文。在某些情况下，消息格式包含第三个部分（脚注）。消息格式按网络字节顺序定义有序的字节序列，也称为 big-endian 格式。消息格式从标头开始，然后是正文，最后是脚注（如果有）。

AWS Encryption SDK 支持的算法套件使用两种消息格式版本之一。没有密钥承诺的算法套件使用消息格式版本 1。带有密钥承诺的算法套件使用消息格式版本 2。

主题

- 标头结构
- 正文结构
- 脚注结构

标头结构

消息标头包含加密的数据密钥以及有关消息正文组成方式的消息。下表描述了在消息格式版本 1 和版本 2 中构成标头的字段。字节是按显示的顺序附加的。

不存在值表示该字段在该版本的消息格式中不存在。粗体文本表示每个版本中不同的值。

Note

您可能需要水平或垂直滚动才能查看此表中的所有数据。

标头结构

字段	消息格式版本 1 长度（字节）	消息格式版本 2 长度（字节）
Version	1	1
Type	1	不存在
Algorithm ID	2	2

字段	消息格式版本 1 长度（字节）	消息格式版本 2 长度（字节）
Message ID	16	32
AAD Length	2 如果加密上下文为空，则 2 字节 AAD 长度字段的值为 0。	2 如果加密上下文为空，则 2 字节 AAD 长度字段的值为 0。
AAD	变量。此字段的长度显示在前 2 个字节中（AAD 长度字段）。 如果加密上下文为空，则在标头中不包含 AAD 字段。	变量。此字段的长度显示在前 2 个字节中（AAD 长度字段）。 如果加密上下文为空，则在标头中不包含 AAD 字段。
Encrypted Data Key Count	2	2
Encrypted Data Key(s)	变量。由加密的数据密钥数和每个密钥的长度决定。	变量。由加密的数据密钥数和每个密钥的长度决定。
Content Type	1	1
Reserved	4	不存在
IV Length	1	不存在
Frame Length	4	4
Algorithm Suite Data	不存在	变量。由生成消息的算法决定。
Header Authentication	变量。由生成消息的算法决定。	变量。由生成消息的算法决定。

版本

该消息格式的版本。版本为 1 或 2，在十六进制表示法中编码为字节 `01` 或 `02`

类型

该消息格式的类型。该类型指示结构的种类。唯一支持的类型描述为客户验证的加密数据。其类型值为 128，在十六进制表示法中编码为字节 `80`。

此字段在消息格式版本 2 中不存在。

算法 ID

使用的算法的标识符。这是一个解释为 16 位无符号整数的 2 字节值。有关算法的更多信息，请参阅AWS Encryption SDK 算法参考。

消息 ID

随机生成的值，用于标识消息。消息 ID：

- 唯一地标识加密的消息。
- 将消息标头弱绑定到消息正文。
- 提供一种机制以安全地在多个加密的消息中重用数据密钥。
- 防止在 AWS Encryption SDK中意外重复使用数据密钥或导致密钥失效。

此值在消息格式版本 1 中为 128 位，在版本 2 中为 256 位。

AAD 长度

其他经过身份验证的数据 (AAD) 的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含 AAD 的字节数。

如果加密上下文为空，则 AAD 长度字段的值为 0。

AAD

其他经过身份验证的数据。AAD 是加密上下文的编码，加密上下文是一个由键值对组成的数组，其中每个密钥和值都是一串已编码的 UTF-8 字符。加密上下文将转换为一个字节序列并用于 AAD 值。如果加密上下文为空，则在标头中不包含 AAD 字段。

在使用具有签名的算法时，加密上下文必须包含 `{'aws-crypto-public-key', Qtxt}` 键值对。Qtxt 表示根据 SEC 1 2.0 版压缩并进行 Base64 编码的椭圆曲线点 Q。加密上下文可以包含额外的值，但构造的 AAD 的最大长度为 2^16 - 1 字节。

下表描述了构成 AAD 的字段。 Key-value 对按键按 UTF-8 字符代码升序排序。字节是按显示的顺序附加的。

AAD 结构

字段	长度（字节）
Key-Value Pair Count	2
Key Length	2
Key	变量。等于在前 2 个字节（键长度）中指定的值。
Value Length	2
Value	变量。等于在前 2 个字节（值长度）中指定的值。

Key-Value 配对数

AAD 中的键值对数。这是一个解释为 16 位无符号整数的 2 字节值，它指定 AAD 中的键值对数。AAD 中的最大键值对数为 2^16 - 1 个。

如果没有加密上下文或加密上下文为空，则在 AAD 结构中不包含该字段。

密钥长度

键值对的键长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含键的字节数。

钥匙

键值对的键。它是一个 UTF-8 编码字节序列。

值长度

键值对的值长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含值的字节数。

价值

键值对的值。它是一个 UTF-8 编码字节序列。

加密数据密钥计数

加密的数据密钥数。这是一个解释为 16 位无符号整数的 2 字节值，它指定加密的数据密钥数。每条消息中加密数据密钥的最大数量为 65535（2^16 - 1）。

加密的数据密钥

加密的数据密钥序列。序列长度由加密的数据密钥数和每个密钥的长度决定。该序列包含至少一个加密的数据密钥。

下表描述了组成每个加密的数据密钥的字段。字节是按显示的顺序附加的。

加密的数据密钥结构

字段	长度（字节）
Key Provider ID Length	2
Key Provider ID	变量。等于在前 2 个字节（密钥提供程序 ID 长度）中指定的值。
Key Provider Information Length	2
Key Provider Information	变量。等于在前 2 个字节（密钥提供程序信息长度）中指定的值。
Encrypted Data Key Length	2
Encrypted Data Key	变量。等于在前 2 个字节（加密的数据密钥长度）中指定的值。

密钥提供商 ID 长度

密钥提供程序标识符的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含密钥提供程序 ID 的字节数。

密钥提供商 ID

密钥提供程序标识符。它用于指示加密的数据密钥的提供程序，并且可以进行扩展。

密钥提供者信息长度

密钥提供程序信息的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含密钥提供程序信息的字节数。

密钥提供商信息

密钥提供程序信息。这是由密钥提供程序决定的。

当 AWS KMS 是主密钥提供者或者您正在使用密 AWS KMS 钥环时，此值包含的 Amazon 资源名称 (ARN)。 AWS KMS key

加密数据密钥长度

加密的数据密钥的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含加密的数据密钥的字节数。

加密的数据密钥

加密的数据密钥。这是密钥提供程序加密的数据加密密钥。

内容类型

加密数据的类型（非帧或帧）。

> **Note**
>
> 尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

帧数据分割成一些等长的部分；每个部分是单独加密的。帧内容为类型 2，在十六进制表示法中编码为字节 `02`。

非成帧数据不会被分割；它是一个单一的加密 blob。 Non-framed content 是类型 1，以十六进制表示法编码为字节`01`。

已保留

预留的 4 字节序列。该值必须为 0。它在十六进制表示法中编码为字节 `00 00 00 00`（即，等于 0 的 32 位整数值的 4 字节序列）。

此字段在消息格式版本 2 中不存在。

四、长度

初始化向量 (IV) 的长度。这是一个解释为 8 位无符号整数的 1 字节值，它指定包含 IV 的字节数。该值由生成消息的算法的 IV 字节值决定。

此字段在消息格式版本 2 中不存在，该版本仅支持在消息标头中使用确定性 IV 值的算法套件。

帧长

帧数据的每个帧的长度。这是一个解释为 32 位无符号整数的 4 字节值，该值指定每个帧的字节数。当数据为非帧数据时，也就是说，当 `Content Type` 字段的值为 1 时，该值必须为 0。

> **Note**
>
> 尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

算法套件数据

生成消息的算法所需的补充数据。长度和内容由算法决定。其长度可能是 0。

此字段在消息格式版本 1 中不存在。

标头认证

标头身份验证是由生成消息的算法决定的。标头身份验证是在整个标头上计算的。它包含 IV 和身份验证标签。字节是按显示的顺序附加的。

标头身份验证结构

字段	版本 1.0 中的长度（字节）	版本 2.0 中的长度（字节）
IV	变量。由生成消息的算法的 IV 字节值决定。	N/A
Authentication Tag	变量。由生成消息的算法的身份验证标签字节值决定。	变量。由生成消息的算法的身份验证标签字节值决定。

四

用于计算标头身份验证标签的初始化向量 (IV)。

此字段在消息格式版本 2 的标头中不存在。消息格式版本 2 仅支持在消息标头中使用确定性 IV 值的算法套件。

身份验证标签

标头的身份验证值。它用于对全部标头内容进行身份验证。

正文结构

消息正文包含加密的数据（称为密文）。正文结构取决于内容类型（非帧或帧）。以下几节介绍了每种内容类型的消息正文格式。消息格式版本 1 和 2 中的消息正文结构相同。

主题

- Non-framed 数据
- 帧数据

Non-framed 数据

Non-framed 数据在具有独特的 IV 和正文 AA D 的单个 blob 中进行加密。

尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

下表描述了组成非帧数据的字段。字节是按显示的顺序附加的。

Non-Framed 车身结构

字段	长度（字节）
IV	变量。等于在标头的 IV Length 字节中指定的值。
Encrypted Content Length	8
Encrypted Content	变量。等于在前 8 个字节（加密的内容长度）中指定的值。
Authentication Tag	变量。由使用的算法实施决定。

四

与加密算法一起使用的初始化向量 (IV)。

加密内容长度

加密的内容（或密文）的长度。这是一个解释为 64 位无符号整数的 8 字节值，它指定包含加密的内容的字节数。

从技术上讲，允许的最大值为 2^63 - 1 或 8 艾字节 (8 EiB)。但实际上，由于实施的算法施加的限制，最大值为 2^36 - 32 或 64 吉字节 (64 GiB)。

Note

由于 Java 语言限制，该开发工具包的 Java 实施进一步将该值限制为 2^31 - 1 或 2 吉字节 (2 GiB)。

加密内容

加密算法返回的加密内容（密文）。

身份验证标签

正文的身份验证值。它用于对消息正文进行身份验证。

帧数据

在帧数据中，明文数据拆分为相等长度的部分（称为帧）。使用独一无二的 IV 和主体 AA D 分别 AWS Encryption SDK 加密每帧。

Note

尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

对于每条消息，帧长度（帧中的加密内容的长度）可能是不同的。帧中的最大字节数为 2^32 - 1。消息中的最大帧数为 2^32 - 1。

共有两种类型的帧：常规和最终。每条消息必须包含最终帧或由最终帧组成。

消息中的所有常规帧具有相同的帧长度。最终帧可能具有不同的帧长度。

帧数据中的帧组成部分因加密的内容长度而异。

- 等于帧长度 – 如果加密的内容长度与常规帧的帧长度相同，则消息可能由包含数据的常规帧以及后面的零（0）长度的最终帧组成。或者，消息可能仅由包含数据的最终帧组成。在这种情况下，最终帧的帧长度与常规帧相同。
- 帧长度的倍数 – 如果加密的内容长度是常规帧的帧长度的整数倍数，则消息可能以包含数据的常规帧结尾，后跟零（0）长度的最终帧。或者，消息可能以包含数据的最终帧结尾。在这种情况下，最终帧的帧长度与常规帧相同。
- 不是帧长度的倍数 – 如果加密的内容长度不是常规帧的帧长度的整数倍数，最终帧将包含其余数据。最终帧的帧长度小于常规帧的帧长度。
- 小于帧长度 – 如果加密的内容长度小于常规帧的帧长度，则消息由包含所有数据的最终帧组成。最终帧的帧长度小于常规帧的帧长度。

下表描述了组成帧的字段。字节是按显示的顺序附加的。

帧正文结构 - 常规帧

字段	长度（字节）
Sequence Number	4
IV	变量。等于在标头的 IV Length 字节中指定的值。
Encrypted Content	变量。等于在标头的 Frame Length 中指定的值。
Authentication Tag	变量。由使用的算法（在标头的 Algorithm ID 中指定）决定。

序列号

帧序列号。它是递增的帧计数器编号。这是一个解释为 32 位无符号整数的 4 字节值。

帧数据必须从序列号 1 开始。后续的帧必须按顺序编号，并且必须在前一个帧的基础上增加 1。否则，解密过程将停止并报告错误。

四

帧的初始化向量 (IV)。该开发工具包使用确定性的方法为消息中的每个帧构造不同的 IV。其长度由使用的算法套件指定。

加密内容

加密算法返回的帧加密内容（密文）。

身份验证标签

帧的身份验证值。它用于对整个帧进行身份验证。

帧正文结构 - 最终帧

字段	长度（字节）
Sequence Number End	4
Sequence Number	4
IV	变量。等于在标头的 IV Length 字节中指定的值。
Encrypted Content Length	4
Encrypted Content	变量。等于在前 4 个字节（加密的内容长度）中指定的值。
Authentication Tag	变量。由使用的算法（在标头的 Algorithm ID 中指定）决定。

序列号结尾

最终帧的指示符。该值在十六进制表示法中编码为 4 字节 FF FF FF FF。

序列号

帧序列号。它是递增的帧计数器编号。这是一个解释为 32 位无符号整数的 4 字节值。

帧数据必须从序列号 1 开始。后续的帧必须按顺序编号，并且必须在前一个帧的基础上增加 1。否则，解密过程将停止并报告错误。

四

帧的初始化向量 (IV)。该开发工具包使用确定性的方法为消息中的每个帧构造不同的 IV。IV 长度是由算法套件指定的。

加密内容长度

加密的内容的长度。这是一个解释为 32 位无符号整数的 4 字节值，它指定包含帧加密内容的字节数。

加密内容

加密算法返回的帧加密内容（密文）。

身份验证标签

帧的身份验证值。它用于对整个帧进行身份验证。

脚注结构

在使用具有签名的算法时，消息格式包含脚注。消息脚注包含在消息标头和正文上计算的数字签名。下表描述了组成脚注的字段。字节是按显示的顺序附加的。消息格式版本 1 和 2 中的消息脚注结构相同。

脚注结构

字段	长度（字节）
Signature Length	2
Signature	变量。等于在前 2 个字节（签名长度）中指定的值。

签名长度

签名的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含签名的字节数。

签名

签名。

AWS Encryption SDK 消息格式示例

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

以下主题显示了 AWS Encryption SDK 消息格式的示例。每个示例都显示了以十六进制表示法表示的原始字节，随后是有关这些字节所表示内容的说明。

主题

- 帧数据（消息格式版本 1）
- 帧数据（消息格式版本 2）
- Non-framed 数据（消息格式版本 1）

帧数据（消息格式版本 1）

以下示例显示了消息格式版本 1 中帧数据的消息格式。

```
+--------+
| Header |
+--------+
01                                        Version (1.0)
80                                        Type (128, customer authenticated encrypted
 data)
0378                                      Algorithm ID (see ####)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27       Message ID (random 128-bit value)
008E                                      AAD Length (142)
0004                                      AAD Key-Value Pair Count (4)
0005                                      AAD Key-Value Pair 1, Key Length (5)
30746869 73                               AAD Key-Value Pair 1, Key ("0This")
0002                                      AAD Key-Value Pair 1, Value Length (2)
6973                                      AAD Key-Value Pair 1, Value ("is")
0003                                      AAD Key-Value Pair 2, Key Length (3)
31616E                                    AAD Key-Value Pair 2, Key ("1an")
```

```
000A                                    AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                  AAD Key-Value Pair 2, Value ("encryption")
0008                                    AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                       AAD Key-Value Pair 3, Key ("2context")
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69     AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
416A4173 7569326F 7430364C 4B77715A     AAD Key-Value Pair 4, Value
 ("AjAsui2ot06LKwqZXDJnU/Aqc2vD+0OkpOZ1cc8Tg2qd7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
614C5467 376C7666 5545572F 38362B2F
35773D3D
0002                                    EncryptedDataKeyCount (2)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77     Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83     Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C3F F02C897B
7A12EB19 8BF2D802 0110803B 24003D1F
A5474FBC 392360B5 CB9997E0 6A17DE4C
A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
4707E356 ADA3735A 7C52D778 B3135A47
9F224BF9 E67E87
```

```
0007                                       Encrypted Data Key 2, Key Provider ID Length
 (7)
6177732D 6B6D73                            Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                       Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63        Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                       Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B        Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C36 CD985E12
D218B674 5BBC6102 0110803B 0320E3CD
E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
57DCC69B AAB1294F 21202C01 9A50D323
72EBAAFD E24E3ED8 7168E0FA DB40508F
556FBD58 9E621C
02                                         Content Type (2, framed data)
00000000                                   Reserved
0C                                         IV Length (12)
00000100                                   Frame Length (256)
4ECBD5C0 9899CA65 923D2347                 IV
0B896144 0CA27950 CA571201 4DA58029        Authentication Tag
+------+
| Body |
+------+
00000001                                   Frame 1, Sequence Number (1)
6BD3FE9C ADBCB213 5B89E8F1                 Frame 1, IV
1F6471E0 A51AF310 10FA9EF6 F0C76EDF        Frame 1, Encrypted Content
F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E
FBD9A0C3 C6E3FB59 C125DBF2 89AC7939
BDEE43A8 0F00F49E ACBBD8B2 1C785089
A90DB923 699A1495 C3B31B50 0A48A830
201E3AD9 1EA6DA14 7F6496DB 6BC104A4
DEB7F372 375ECB28 9BF84B6D 2863889F
```

```
CB80A167 9C361C4B 5EC07438 7A4822B4
A7D9D2CC 5150D414 AF75F509 FCE118BD
6D1E798B AEBA4CDB AD009E5F 1A571B77
0041BC78 3E5F2F41 8AF157FD 461E959A
BB732F27 D83DC36D CC9EBC05 00D87803
57F2BB80 066971C2 DEEA062F 4F36255D
E866C042 E1382369 12E9926B BA40E2FC
A820055F FB47E428 41876F14 3B6261D9
5262DB34 59F5D37E 76E46522 E8213640
04EE3CC5 379732B5 F56751FA 8E5F26AD        Frame 1, Authentication Tag
00000002                                    Frame 2, Sequence Number (2)
F1140984 FF25F943 959BE514                  Frame 2, IV
216C7C6A 2234F395 F0D2D9B9 304670BF        Frame 2, Encrypted Content
A1042608 8A8BCB3F B58CF384 D72EC004
A41455B4 9A78BAC9 36E54E68 2709B7BD
A884C1E1 705FF696 E540D297 446A8285
23DFEE28 E74B225A 732F2C0C 27C6BDA2
7597C901 65EF3502 546575D4 6D5EBF22
1FF787AB 2E38FD77 125D129C 43D44B96
778D7CEE 3C36625F FF3A985C 76F7D320
ED70B1F3 79729B47 E7D9B5FC 02FCE9F5
C8760D55 7779520A 81D54F9B EC45219D
95941F7E 5CBAEAC8 CEC13B62 1464757D
AC65B6EF 08262D74 44670624 A3657F7F
2A57F1FD E7060503 AC37E197 2F297A84
DF1172C2 FA63CF54 E6E2B9B6 A86F582B
3B16F868 1BBC5E4D 0B6919B3 08D5ABCF
FECDC4A4 8577F08B 99D766A1 E5545670
A61F0A3B A3E45A84 4D151493 63ECA38F        Frame 2, Authentication Tag
FFFFFFFF                                    Final Frame, Sequence Number End
00000003                                    Final Frame, Sequence Number (3)
35F74F11 25410F01 DD9E04BF                  Final Frame, IV
0000008E                                    Final Frame, Encrypted Content Length (142)
F7A53D37 2F467237 6FBD0B57 D1DFE830        Final Frame, Encrypted Content
B965AD1F A910AA5F 5EFFFFF4 BC7D431C
BA9FA7C4 B25AF82E 64A04E3A A0915526
88859500 7096FABB 3ACAD32A 75CFED0C
4A4E52A3 8E41484D 270B7A0F ED61810C
3A043180 DF25E5C5 3676E449 0986557F
C051AD55 A437F6BC 139E9E55 6199FD60
6ADC017D BA41CDA4 C9F17A83 3823F9EC
B66B6A5A 80FDB433 8A48D6A4 21CB
811234FD 8D589683 51F6F39A 040B3E3B        Final Frame, Authentication Tag
+--------+
```

```
| Footer |
+--------+
0066                                        Signature Length (102)
30640230 085C1D3C 63424E15 B2244448         Signature
639AED00 F7624854 F8CF2203 D7198A28
758B309F 5EFD9D5D 2E07AD0B 467B8317
5208B133 02301DF7 2DFC877A 66838028
3C6A7D5E 4F8B894E 83D98E7C E350F424
7E06808D 0FE79002 E24422B9 98A0D130
A13762FF 844D
```

帧数据（消息格式版本 2）

以下示例显示了消息格式版本 2 中帧数据的消息格式。

```
+--------+
| Header |
+--------+
02                                          Version (2.0)
0578                                        Algorithm ID (see Algorithms reference)
122747eb 21dfe39b 38631c61 7fad7340
cc621a30 32a11cc3 216d0204 fd148459         Message ID (random 256-bit value)
008e                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30546869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616e                                      AAD Key-Value Pair 2, Key ("1an")
000a                                        AAD Key-Value Pair 2, Value Length (10)
656e6372 79707469 6f6e                      AAD Key-Value Pair 2, Value ("encryption")
0008                                        AAD Key-Value Pair 3, Key Length (8)
32636f6e 74657874                           AAD Key-Value Pair 3, Key ("2context")
0007                                        AAD Key-Value Pair 3, Value Length (7)
6578616d 706c65                             AAD Key-Value Pair 3, Value ("example")
0015                                        AAD Key-Value Pair 4, Key Length (21)
6177732d 63727970 746f2d70 75626c69         AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632d6b65 79
0044                                        AAD Key-Value Pair 4, Value Length (68)
41746733 72703845 41345161 36706669         AAD Key-Value Pair 4, Value
 ("QXRnM3JwOEVBNFFhNnBmaTk3MUlTNTk3NHpOMnlZWE5vSmtwRHFPc0dIYkVaVDRqME5OMlFkRStmbTFFVY01WdThnPT0=
```

```
39373149 53353937 347a4e32 7959584e
6f4a6b70 44714f73 47486245 5a54346a
304e4e32 5164452b 666d3155 634d5675
38673d3d
0001                                        Encrypted Data Key Count (1)
0007                                        Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732d 6b6d73                             Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004b                                        Encrypted Data Key 1, Key Provider
 Information Length (75)
61726e3a 6177733a 6b6d733a 75732d77         Encrypted Data Key 1, Key
 Provider Information ("arn:aws:kms:us-west-2:658956600833:key/b3537ef1-
d8dc-4780-9f5a-55776cbb2f7f")
6573742d 323a3635 38393536 36303038
33333a6b 65792f62 33353337 6566312d
64386463 2d343738 302d3966 35612d35
35373736 63626232 663766
00a7                                        Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010100 7840f38c 275e3109 7416c107         Encrypted Data Key 1, Encrypted Data Key
29515057 1964ada3 ef1c21e9 4c8ba0bd
bc9d0fb4 14000000 7e307c06 092a8648
86f70d01 0706a06f 306d0201 00306806
092a8648 86f70d01 0701301e 06096086
48016503 04012e30 11040c39 32d75294
06063803 f8460802 0110803b 2a46bc23
413196d2 903bf1d7 3ed98fc8 a94ac6ed
e00ee216 74ec1349 12777577 7fa052a5
ba62e9e4 f2ac8df6 bcb1758f 2ce0fb21
cc9ee5c9 7203bb
02                                          Content Type (2, framed data)
00001000                                    Frame Length (4096)
05cd035b 29d5499d 4587570b 87502afe         Algorithm Suite Data (key commitment)
634f7b2c c3df2aa9 88a10105 4a2c7687
76cb339f 2536741f 59a1c202 4f2594ab         Authentication Tag
+------+
| Body |
+------+
ffffffff                                    Final Frame, Sequence Number End
00000001                                    Final Frame, Sequence Number (1)
00000000 00000000 00000001                  Final Frame, IV
00000009                                    Final Frame, Encrypted Content Length (9)
fa6e39c6 02927399 3e                        Final Frame, Encrypted Content
```

```
f683a564 405d68db eeb0656c d57c9eb0         Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
0067                                        Signature Length (103)
30650230 2a1647ad 98867925 c1712e8f         Signature
ade70b3f 2a2bc3b8 50eb91ef 56cfdd18
967d91d8 42d92baf 357bba48 f636c7a0
869cade2 023100aa ae12d08f 8a0afe85
e5054803 110c9ed8 11b2e08a c4a052a9
074217ea 3b01b660 534ac921 bf091d12
3657e2b0 9368bd
```

Non-framed 数据（消息格式版本 1）

以下示例显示了非帧数据的消息格式。

> **Note**
>
> 尽可能使用帧数据。仅 AWS Encryption SDK 支持传统使用的非成帧数据。的某些语言实现仍然 AWS Encryption SDK 可以生成非成帧的密文。所有支持的语言实现都可以解密成帧和非帧加密文字。

```
+--------+
| Header |
+--------+
01                                          Version (1.0)
80                                          Type (128, customer authenticated encrypted
 data)
0378                                        Algorithm ID (see ####)
B8929B01 753D4A45 C0217F39 404F70FF         Message ID (random 128-bit value)
008E                                        AAD Length (142)
0004                                        AAD Key-Value Pair Count (4)
0005                                        AAD Key-Value Pair 1, Key Length (5)
30746869 73                                 AAD Key-Value Pair 1, Key ("0This")
0002                                        AAD Key-Value Pair 1, Value Length (2)
6973                                        AAD Key-Value Pair 1, Value ("is")
0003                                        AAD Key-Value Pair 2, Key Length (3)
31616E                                      AAD Key-Value Pair 2, Key ("1an")
000A                                        AAD Key-Value Pair 2, Value Length (10)
```

```
656E6372 79774690 6F6E                     AAD Key-Value Pair 2, Value ("encryption")
0008                                       AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                          AAD Key-Value Pair 3, Key ("2context")
0007                                       AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                            AAD Key-Value Pair 3, Value ("example")
0015                                       AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69        AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                       AAD Key-Value Pair 4, Value Length (68)
41734738 67473949 6E4C5075 3136594B        AAD Key-Value Pair 4, Value
 ("AsG8gG9InLPu16YKlqXTOD+nykG8YqHAhqecj8aXfD2e5B4gtVE73dZkyClA+rAMOQ==")
6C715854 4F442B6E 796B4738 59714841
68716563 6A386158 66443265 35423467
74564537 33645A6B 79436C41 2B72414D
4F513D3D
0002                                       Encrypted Data Key Count (2)
0007                                       Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                            Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                       Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77        Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                       Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83        Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C28 4116449A
0F2A0383 659EF802 0110803B B23A8133
3A33605C 48840656 C38BCB1F 9CCE7369
E9A33EBE 33F46461 0591FECA 947262F3
418E1151 21311A75 E575ECC5 61A286E0
3E2DEBD5 CB005D
```

```
0007                                      Encrypted Data Key 2, Key Provider ID Length
 (7)
6177732D 6B6D73                           Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                      Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63       Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                      Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B       Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040CB2 A820D0CC
76616EF2 A6B30D02 0110803B 8073D0F1
FDD01BD9 B0979082 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F18A46 80E2C43F A34C0E58 11D05114
2A363C2A E11397
01                                        Content Type (1, nonframed data)
00000000                                  Reserved
0C                                        IV Length (12)
00000000                                  Frame Length (0, nonframed data)
734C1BBE 032F7025 84CDA9D0                IV
2C82BB23 4CBF4AAB 8F5C6002 622E886C       Authentication Tag
+------+
| Body |
+------+
D39DD3E5 915E0201 77A4AB11                IV
00000000 0000028E                         Encrypted Content Length (654)
E8B6F955 B5F22FE4 FD890224 4E1D5155       Encrypted Content
5871BA4C 93F78436 1085E4F8 D61ECE28
59455BD8 D76479DF C28D2E0B BDB3D5D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E3F54653 DF205D30 0081D2D8
55FCDA5B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05EA5 0E2F2F40 47A60344
```

```
ECE10AA7 559AF633 9DE2C21B 12AC8087
95FE9C58 C65329D1 377C4CD7 EA103EC1
31E4F48A 9B1CC047 EE5A0719 704211E5
B48A2068 8060DF60 B492A737 21B0DB21
C9B21A10 371E6179 78FAFB0B BAAEC3F4
9D86E334 701E1442 EA5DA288 64485077
54C0C231 AD43571A B9071925 609A4E59
B8178484 7EB73A4F AAE46B26 F5B374B8
12B0000C 8429F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D4C2F
A15D0551 DAEBA4AF 2060D0D5 CB1DA4E6
5E2034DB 4D19E7CD EEA6CF7E 549C86AC
46B2C979 AB84EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 88F5AFE1 98488557
1BABBAE4 BE55325E 4FB7E602 C1C04BEE
F3CB6B86 71666C06 6BF74E1B 0F881F31
B731839B CF711F6A 84CA95F5 958D3B44
E3862DF6 338E02B5 C345CFF8 A31D54F3
6920AA76 0BF8E903 552C5A04 917CCD11
D4E5DF5C 491EE86B 20C33FE1 5D21F0AD
6932E67C C64B3A26 B8988B25 CFA33E2B
63490741 3AB79D60 D8AEFBE9 2F48E25A
978A019C FE49EE0A 0E96BF0D D6074DDB
66DFF333 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2AA3F1 88251874 FDC94F6B
9247EF61 3E7B7E0D 29F3AD89 FA14A29C
76E08E9B 9ADCDF8C C886D4FD A69F6CB4
E24FDE26 3044C856 BF08F051 1ADAD329
C4A46A1E B5AB72FE 096041F1 F3F3571B
2EAFD9CB B9EB8B83 AE05885A 8F2D2793
1E3305D9 0C9E2294 E8AD7E3B 8E4DEC96
6276C5F1 A3B7E51E 422D365D E4C0259C
50715406 822D1682 80B0F2E5 5C94
65B2E942 24BEEA6E A513F918 CCEC1DE3      Authentication Tag
+--------+
| Footer |
+--------+
0067                                     Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D627      Signature
CBE194F1 1CC0F8CF D27B7F8B F50658C0
BE84B355 3CED1721 A0BE2A1B 8E3F449E
1BEB8281 023100B2 0CB323EF 58A4ACE3
1559963B 889F72C3 B15D1700 5FB26E61
```

```
331F3614 BC407CEE B86A66FA CBF74D9E
34CB7E4B 363A38
```

正文其他经过身份验证的数据 (AAD) 参考 AWS Encryption SDK

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

您必须为每个加密操作的AES-GCM 算法提供额外的身份验证数据 (AAD)。这适用于帧和非帧正文数据。有关 AAD 及其在 Galois/Counter 模式 (GCM) 中的使用方式的更多信息，请参阅分组密码操作模式建议： Galois/Counter 模式 (GCM) 和 G MAC。

下表描述了组成正文 AAD 的字段。字节是按显示的顺序附加的。

正文 AAD 结构

字段	长度（字节）
Message ID	16
Body AAD Content	变量。请参阅以下列表中的正文 AAD 内容。
Sequence Number	4
Content Length	8

消息 ID

在消息标头中设置的相同 Message ID 值。

正文 AAD 内容

由所用正文数据类型确定的 UTF-8 编码值。

对于非帧数据，请使用 `AWSKMSEncryptionClient Single Block` 值。

对于帧数据中的常规帧，请使用 `AWSKMSEncryptionClient Frame` 值。

对于帧数据中的最终帧，请使用 `AWSKMSEncryptionClient Final Frame` 值。

序列号

解释为 32 位无符号整数的 4 字节值。

对于帧数据，这是帧序列号。

对于非帧数据，请使用值 1（在十六进制表示法中编码为 4 字节 `00 00 00 01`）。

内容长度

为加密算法提供的明文数据的长度（字节）。这是一个解释为 64 位无符号整数的 8 字节值。

AWS Encryption SDK 算法参考

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

如果您正在构建自己的库，该库可以读取和写入与兼容的密文 AWS Encryption SDK，则需要了解如何 AWS Encryption SDK 实现支持的算法套件来加密原始数据。

AWS Encryption SDK 支持以下算法套件。所有 AES-GCM 算法套件都有一个 12 字节的初始化向量和一个 16 字节的 AES-GCM身份验证标记。默认算法套件因 AWS Encryption SDK 版本和所选密钥承诺策略而异。有关详细信息，请参阅承诺策略和算法套件。

AWS Encryption SDK 算法套件

算法 ID	消息格式版本	加密算法	数据密钥长度（位）	密钥派生算法	签名算法	密钥承诺算法	算法套件数据长度（字节）
`05 78`	0x02	AES-GCM	256	HKDF 与 SHA-512	带有和的 ECDSA	HKDF 与 SHA-512	32（密钥承诺）

算法 ID	消息格式版本	加密算法	数据密钥长度（位）	密钥派生算法	签名算法	密钥承诺算法	算法套件数据长度（字节）
					P-384 SHA-384		
04 78	0x02	AES-GCM	256	HKDF 与 SHA-512	无	HKDF 与 SHA-512	32（密钥承诺）
03 78	0x01	AES-GCM	256	HKDF 与 SHA-384	带有和的 ECDSA P-384 SHA-384	无	N/A
03 46	0x01	AES-GCM	192	HKDF 与 SHA-384	带有和的 ECDSA P-384 SHA-384	无	N/A
02 14	0x01	AES-GCM	128	HKDF 与 SHA-256	带有和的 ECDSA P-256 SHA-256	无	N/A
01 78	0x01	AES-GCM	256	HKDF 与 SHA-256	无	无	N/A
01 46	0x01	AES-GCM	192	HKDF 与 SHA-256	无	无	N/A
01 14	0x01	AES-GCM	128	HKDF 与 SHA-256	无	无	N/A
00 78	0x01	AES-GCM	256	无	无	无	N/A
00 46	0x01	AES-GCM	192	无	无	无	N/A

算法 ID	消息格式版本	加密算法	数据密钥长度（位）	密钥派生算法	签名算法	密钥承诺算法	算法套件数据长度（字节）
`00 14`	0x01	AES-GCM	128	无	无	无	N/A

算法 ID

一个 2 字节十六进制值，用于唯一地标识算法实施。该值存储在加密文字的消息标头中。

消息格式版本

消息格式的版本。带有密钥承诺的算法套件使用消息格式版本 2（0x02）。没有密钥承诺的算法套件使用消息格式版本 1（0x01）。

算法套件数据长度

特定于算法套件的数据长度（以字节为单位）。只有消息格式版本 2（0x02）支持此字段。在消息格式版本 2（0x02）中，此数据出现在消息标头的 `Algorithm suite data` 字段中。支持密钥承诺的算法套件使用 32 字节作为密钥承诺字符串。有关更多信息，请参阅该列表中的密钥承诺算法。

数据密钥长度

数据密钥的长度（以位为单位）。 AWS Encryption SDK 支持 256 位、192 位和 128 位密钥。数据密钥是由密钥环或主密钥生成的。

在某些实现中，此数据密钥用作 HMAC-based提取和扩展密钥派生函数 (HKDF) 的输入。HKDF 的输出用作加密算法中的数据加密密钥。有关更多信息，请参阅该列表中的密钥派生算法。

加密算法

与加密算法一起使用的名称和模式。中的算法套件 AWS Encryption SDK 使用带 Galois/Counter 模式 (GCM) 的高级加密标准 (AES) 加密算法。

密钥承诺算法

用于计算密钥承诺字符串的算法。输出存储在消息标头的 `Algorithm suite data` 字段中，用于验证密钥承诺的数据密钥。

有关向算法套件添加密钥承诺的技术说明，请参阅 Cryptology ePrint Archive 中的 Key Committing AEADs。

密钥派生算法

用于派生 HMAC-based 数据加密密钥的提取和扩展密钥派生函数 (HKDF)。 AWS Encryption SDK 使用 RFC 5869 中定义的 HKDF。

没有密钥承诺的算法套件（算法 ID `01xx` – `03xx`）

- 使用的哈希函数是 SHA-384 或 SHA-256，具体取决于算法套件。
- 对于提取步骤：
  - 不使用加密盐。根据 RFC，加密盐设置为包含零的字符串。字符串长度等于哈希函数输出的长度，哈希函数输出的长度为 48 字节 SHA-384 ，为 32 字节 SHA-256。
  - 输入加密材料是来自密钥环或主密钥提供程序的数据密钥。
- 对于扩展步骤：
  - 输入伪随机密钥是提取步骤的输出。
  - 输入信息是将算法 ID 和消息 ID（按此顺序）串联在一起的结果。
  - 输出加密材料的长度是数据密钥长度。该输出用作加密算法中的数据加密密钥。

带有密钥承诺的算法套件（算法 ID `04xx` 和`05xx`）

- 使用的哈希函数是 SHA-512。
- 对于提取步骤：
  - 加密盐是一个 256 位的加密随机值。在消息格式版本 2（0x02）中，此值存储在 `MessageID` 字段中。
  - 初始加密材料是来自密钥环或主密钥提供程序的数据密钥。
- 对于扩展步骤：
  - 输入伪随机密钥是提取步骤的输出。
  - 密钥标签是按大 UTF-8-encoded 字节顺序排列的`DERIVEKEY`字符串的字节。
  - 输入信息是将算法 ID 和密钥标签（按此顺序）串联在一起的结果。
  - 输出加密材料的长度是数据密钥长度。该输出用作加密算法中的数据加密密钥。

消息格式版本

算法套件中使用的消息格式的版本。有关更多信息，请参阅 消息格式参考。

签名算法

用于在加密文字标头和正文上生成数字签名的签名算法。 AWS Encryption SDK 使用椭圆曲线数字签名算法 (ECDSA)，具体细节如下：

- 使用的椭圆曲线可以是 P-384 或 P-256 曲线，由算法 ID 指定。这些曲线是在数字签名标准 (DSS) (FIPS PUB 186-4) 中定义的。
- 使用的哈希函数是 SHA-384（使用 P-384 曲线）或 SHA-256（使用 P-256 曲线）。

AWS Encryption SDK 初始化向量参考

本页面提供了在您构建与 AWS Encryption SDK兼容的加密库时可供参考的信息。如果您不需要构建自己的兼容加密库，则可能不需要此信息。

要 AWS Encryption SDK 在支持的编程语言之一中使用，请参阅编程语言。

有关定义适当 AWS Encryption SDK 实现要素的规范，请参阅中的AWS Encryption SDK 规范 GitHub。

AWS Encryption SDK 提供所有支持的算法套件所需的初始化向量 (IV)。该开发工具包使用帧序列号构造一个 IV，以便同一消息中的两个帧不能具有相同的 IV。

每个 96 位（12 字节）IV 是通过两个按以下顺序串联的 big-endian 字节数组构造的：

- 64 位：0（保留以供将来使用）
- 32 位：帧序列号。对于标头身份验证标签，该值全部为零。

在引入数据密钥缓存之前， AWS Encryption SDK 始终使用新数据密钥加密每条消息，并随机生成所有 IV。随机生成的 IV 从加密角度上是安全的，因为从不重用数据密钥。在该开发工具包引入数据密钥缓存（有意重用数据密钥）后，我们更改了该开发工具包生成 IV 的方式。

通过使用无法在消息中重复的确定性 IV，可以显著增加可根据单个数据密钥安全执行的调用次数。此外，缓存的数据密钥始终使用具有密钥派生函数的算法套件。使用具有伪随机密钥派生函数的确定性 IV 从数据密钥派生加密密钥，可以在不超过加密边界的情况下加密 2^32 条消息。 AWS Encryption SDK

AWS KMS 分层钥匙圈技术细节

AWS KMS 分层密钥环使用唯一的数据密钥来加密每条消息，并使用源自活动分支密钥的唯一包装密钥对每个数据密钥进行加密。它使用计数器模式下的密钥派生和带有 HMAC SHA-256 的伪随机函数，通过以下输入推导出 32 字节的包装密钥。

- 一个 16 字节的随机加密盐
- 活动分支密钥
- 密钥提供商标识符 “aws-kms-hierarchy” 的UTF-8 编码值

分层密钥环使用派生的包装密钥使用 AES-GCM-256 带有 16 字节身份验证标签和以下输入的纯文本数据密钥副本进行加密。

- 派生的包装密钥用作 AES-GCM 密码密钥
- 数据密钥用作 AES-GCM 消息
- 使用 12 字节的随机初始化向量 (IV) 作为 AES-GCM IV
- 包含以下序列化值的其他额外验证数据（AAD）。

值	长度（字节）	解释为
“aws-kms-hierary”	17	UTF-8 已编码
分支密钥标识符	变量	UTF-8 已编码
分支密钥版本	16	UTF-8 已编码
加密上下文	变量	UTF-8 编码的密钥值对

《AWS Encryption SDK 开发者指南》的文档历史记录

本主题介绍了有关 AWS Encryption SDK 开发人员指南的重要更新。

主题

- 最近的更新
- 早期更新

最近的更新

下表介绍了自 2017 年 11 月起对此文档的一些重要更改。除了此处列出的主要更改以外，我们还会经常更新文档，以改进说明和示例以及处理您发送给我们的反馈意见。要获得有关重要更改的通知，请订阅 RSS 源。

变更	说明	日期
AWS Encryption SDK 适用于.NET 版本 5.x	材料提供者库 (MPL) 2.0 版的更新。取消对适用于.NET v3 的 AWS SDK 的支持，并添加了对适用于.NET v4 的 AWS SDK 的支持。	2025 年 3 月 25 日
正式发布	添加了 AWS KMS ECDH 密钥环和 Ra w EC DH 密钥环的文档。	2024 年 6 月 17 日
AWS Encryption SDK for Java 版本 3.x	AWS Encryption SDK for Java 与材料提供者库集成。添加对密钥环和所需的加密上下文 CMM 的支持。	2023 年 12 月 6 日
AWS Encryption SDK 适用于.NET 版本 4.x	增加了对 AWS KMS 分层密钥环、所需的加密上下文 CMM 和非对称 RSA 密钥环的支持。AWS KMS	2023 年 10 月 12 日

正式发布	引入对.NET AWS Encryption SDK 的支持。	2022 年 5 月 17 日
文档更改	将 “客户主密钥 (CMK)” AWS Key Management Service 一词替换为 “AWS KMS key和 KMS 密钥”。	2021 年 8 月 30 日
正式发布	添加了对的支持 AWS Key Management Service。 (AWS KMS) 多区域密钥。多区域密钥是不同的 AWS KMS 密钥 AWS 区域 ，可以互换使用，因为它们具有相同的密钥 ID 和密钥材料。	2021 年 6 月 8 日
正式发布	添加并更新有关改进的消息解密过程的文档。	2021 年 5 月 11 日
正式发布	为加 AWS 密 CLI 版本 1.8 的正式发布版本添加和更新了文档。 x 来取代 AWS 加密 CLI 版本 1.7。 x 和加 AWS 密 CLI 2.1。 x 来取代 AWS 加密 CLI 2.0。 x。	2020 年 10 月 27 日
正式发布	添加并更新 AWS Encryption SDK 版本 1.7.x 和 2.0.x 的正式发布版本文档，包括最佳实践指南、迁移指南、更新的概念、更新的编程语言主题、更新的算法套件参考、更新的消息格式参考以及新的消息格式示例。	2020 年 9 月 24 日
正式发布	添加并更新了AWS Encryption SDK for JavaScript通用版的文档。	2019 年 10 月 1 日

预览版	添加并更新了AWS Encryption SDK for JavaScript公开测试版的文档。	2019 年 6 月 21 日
正式发布	添加并更新了AWS Encryption SDK for C通用版的文档。	2019 年 5 月 16 日
预览版	添加了AWS Encryption SDK for C预览版的文档。	2019 年 2 月 5 日
新版本	添加 AWS Encryption SDK命令行界面的文档。	2017 年 11 月 20 日

早期更新

下表介绍了 2017 年 11 月之前对《AWS Encryption SDK 开发人员指南》做出的重要更改。

更改	描述	日期
新版本	添加了数据密钥缓存章节以介绍新功能。 添加了说明 SDK 从随机生成变 IVs 为构造确定性的the section called “初始化向量参考” IVs主题。 添加 the section called “概念” 主题来解释概念，包括新的加密材料管理器。	2017 年 7 月 31 日
更新	将消息格式参考文档扩充为 AWS Encryption SDK 引用中的一个新小节。 添加了有关 AWS Encryption SDK 支持的算法套件.	2017 年 3 月 21 日

更改	描述	日期
新版本	除此之外， AWS Encryption SDK 现在还支持Python编程语言Java。	2017 年 3 月 21 日
初始版本	AWS Encryption SDK 和本文档的初始版本。	2016 年 3 月 22 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。