開發人員指南

AWS Encryption SDK

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Encryption SDK: 開發人員指南

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

什麼是 AWS Encryption SDK？ 1
- 在開源存儲庫中開發 2
- 與加密程式庫和服務的相容性 2
- Support 與維護 3
- 進一步了解 4
- 傳送意見回饋 5
- 概念 5
  - 封套加密 6
  - 資料金鑰 7
  - 包裝鍵 8
  - 金鑰圈和主金鑰提供者 9
  - 加密內容 9
  - 加密的訊息 10
  - 演算法套件 11
  - 密碼編譯資料管理員 11
  - 對稱和非對稱加密 11
  - 主要承諾 12
  - 承諾政策 13
  - 數位簽章 14
- SDK 如何運作 15
  - 如何加AWS Encryption SDK密數據 15
  - 如何解AWS Encryption SDK密加密的郵件 15
- 支援的演算法套件 16
  - 建議：AES-GCM，搭配金鑰衍生、簽署和關鍵承諾 16
  - 其他支援的演算法套件 17

與 AWS KMS 互動 19

最佳實務 20

設定軟體開發套件 23
- 選擇一種編程語言 23
- 選擇包裝鍵 23
- 使用多地區 AWS KMS keys 24
- 選擇演算法套件 44
- 限制加密的資料金鑰 53
- 建立探索篩選器 57

設定承諾產品原則 59
使用串流資料 59
快取資料金鑰 60
使用 keyring 61
keyring 如何運作 61
Keyring 相容性 63
加密金鑰圈的不同需求 63
相容 Keyring 和主金鑰提供者 64
選擇鑰匙圈 65
AWS KMS 鑰匙圈 66
AWS KMS 階層式鑰匙圈 83
AWS KMS ECDH 鑰匙圈 105
原始 AES keyring 109
原始 RSA keyring 112
原始 ECDH 鑰匙圈 117
多重 keyring 122
程式設計語言 127
C 127
安裝 128
使用 C 開發套件 129
範例 133
.NET 139
安裝和建置 141
除錯 141
AWS KMS Keyring 142
必要的加密內容 CMM 145
範例 146
Java 154
先決條件 154
安裝 155
AWS KMS Keyring 156
必要的加密內容 CMM 159
範例 160
JavaScript 173
相容性 174
安裝 176

模組 ... 176
範例 ... 179
Python ... 185
必要條件 ... 185
安裝 ... 185
範例 ... 186
命令列界面 ... 197
安裝 CLI ... 198
如何使用 CLI ... 201
範例 ... 213
語法和參數參考 ... 235
版本 ... 246
資料金鑰快取 ... 249
如何使用資料金鑰快取 ... 250
使用資料金鑰快取：S tep-by-step ... 250
資料金鑰快取範例：加密字串 ... 258
設定快取安全性閾值 ... 274
資料金鑰快取詳細資訊 ... 275
資料金鑰快取的運作方式 ... 276
建立密碼編譯資料快取 ... 278
建立快取密碼編譯資料管理員 ... 279
資料金鑰快取項目中有什麼項目？ ... 280
加密內容：如何選擇快取項目 ... 280
我的應用程式是否使用快取的資料金鑰？ ... 281
資料金鑰快取範例 ... 281
本機快取結果 ... 282
範例程式碼 ... 283
AWS CloudFormation 範本 ... 294
的版本 AWS Encryption SDK ... 310
C ... 310
C＃/. ... 311
命令 CLI 介面 ... 312
Java ... 313
JavaScript ... 315
Python ... 316
版本詳情 ... 317

早於 1.7 的版本。 x 317
版本 1.7。 x 317
版本 2.0。 x 320
版本 2.2. x 321
版本 2.3. x 321
遷移您的AWS Encryption SDK 323
如何遷移和部署 324
階段 1：將您的應用程式更新至最新的 1.x版 325
階段 2：將您的應用程式更新至最新版本 326
更新AWS KMS主要金鑰提供者 326
遷移至嚴謹模 327
移轉至探索模式 331
正在更新AWS KMS鑰匙圈 333
設定承諾產品原則 336
如何設定承諾政策 337
移轉至最新版本的疑難排解 344
已取代或移除的物件 345
組態衝突：承諾政策和演算法套件 345
組態衝突：承諾政策和密文 346
金鑰履約承諾驗證失敗 346
其他加密失敗 346
其他解密失敗 347
轉返考量 347
常見問答集 348
參考資料 352
訊息格式參考 352
標題結構 353
本文結構 360
頁尾結構 364
訊息格式範例 365
框架數據（消息格式第 1 版） 365
框架數據（消息格式第 2 版） 369
非框架數據（消息格式第 1 版） 371
內文 AAD 參考 375
演算法參考 376
初始化向量參考 380

AWS KMS 分層鑰匙圈技術細節 ... 381
文件歷史紀錄 ... 382
最近更新 ... 382
舊版更新 ... 384
... ccclxxxvi

什麼是 AWS Encryption SDK？

AWS Encryption SDK是用戶端加密程式庫，旨在讓每個人輕鬆地使用產業標準和最佳實務來加密和解密資料。這樣一來，您就能夠專注於應用程式的核心功能，而不用擔心如何讓資料獲得最好的加密與解密操作。在 Apache 2.0 授權下，您可免費取得 AWS Encryption SDK。

AWS Encryption SDK能提供類似下列問題的解答：

- 我應該使用哪種加密演算法？
- 這種演算法應該如何使用？或是我該使用哪種模式？
- 我應該如何產生加密金鑰？
- 我應該如何保護加密金鑰？還有這份金鑰該存放在哪裡？
- 我可以如何製作可攜式的加密資料？
- 我應該如何確保預定收件人可以讀取我的加密資料？
- 我應該如何確保已加密資料在寫入和讀取的這段期間不會遭到竄改？
- 如何使用AWS KMS返回的數據鍵？

使用時AWS Encryption SDK，您可以定義主要金鑰提供者 (Java 和 Python) 或金鑰環 (C、C#/.NET 和 JavaScript)，以決定您使用哪些包裝金鑰來保護您的資料。然後，您再使用 AWS Encryption SDK 提供的直覺式方法，進行資料的加密和解密。AWS Encryption SDK會處理其餘的工作。

如果沒有 AWS Encryption SDK，您花在建置解決方案的氣力可能遠超過建置應用程式核心功能所花的心力。AWS Encryption SDK提供了下列優勢，為這些問題提供了解答。

符合加密最佳實務的預設實作

預設情況下，AWS Encryption SDK會為其加密的每個資料物件產生唯一的資料金鑰。這個做法符合使用唯一資料金鑰進行每次加密操作的加密最佳實務要求。

AWS Encryption SDK會使用安全、身分驗證的對稱金鑰演算法，加密處理您的資料。如需詳細資訊，請參閱the section called “支援的演算法套件”。

使用包裝密鑰保護數據鍵的框架

通過在一個或多個包裝密鑰下加密數據來AWS Encryption SDK保護數據密鑰。藉由提供架構來使用多個包裝金鑰來加密資料金鑰，AWS Encryption SDK有助於讓您的加密資料可攜式。

例如，將內部部署 HSM AWS KMS key 中的資料AWS KMS和金鑰下的資料加密。您可以使用任何一個包裝密鑰來解密數據，以防其中一個密鑰不可用或調用者沒有使用這兩個密鑰的權限。

儲存已加密資料金鑰和已加密資料的格式化訊息

AWS Encryption SDK會在採用已定義資料格式的已加密訊息中，存放已加密的資料和已加密的資料金鑰。這表示您不需要追蹤或保護處理資料加密的資料金鑰，因為 AWS Encryption SDK全都幫您完成了。

某些語言實現AWS Encryption SDK需要 AWS SDK，但AWS Encryption SDK不需要一個AWS 帳戶，它不依賴於任何AWS服務。AWS 帳戶只有當您選擇使用來保護您的資料時，AWS KMS keys才需要一個。

在開源存儲庫中開發

AWS Encryption SDK是在開放原始碼儲存庫中開發的 GitHub。您可以使用這些儲存庫來檢視程式碼、讀取和送出問題，以及尋找特定於您語言實作的資訊。

- 適用於 C 的 AWS Encryption SDK — aws-encryption-sdk-c
- AWS Encryption SDK對於 .NET-`aws-encryption-sdk-dafny` 存儲庫的aws-encryption-sdk-net目錄。
- AWS加密 CLI — aws-encryption-sdk-cli
- 適用於 JAVA 的 AWS Encryption SDK — aws-encryption-sdk-java
- 適用於 JavaScript 的 AWS Encryption SDK — aws-encryption-sdk-javascript
- 適用於 Python 的 AWS Encryption SDK — aws-encryption-sdk-python

與加密程式庫和服務的相容性

支援AWS Encryption SDK數種程式設計語言。所有語言實作都是可互通的。您可以使用一種語言實作加密，並使用另一種語言進行解密。互通性可能受到語言限制。如果是這樣，這些限制會在語言實作的主題中加以說明。此外，加密和解密時，您必須使用相容的 Keyring，或主金鑰和主金鑰提供者。如需詳細資訊，請參閱 the section called "Keyring 相容性"。

但是，AWS Encryption SDK 無法與其他程式庫交互操作。由於每個程式庫都會以不同的格式傳回加密資料，因此您無法使用一個程式庫加密，並使用另一個程式庫解密。

加密 DynamoDB 戶端和 Amazon S3 用戶端加密

AWS Encryption SDK無法解密由 DynamoDB 加密用戶端或 Amazon S3 用戶端加密加密的資料。這些庫無法解密AWS Encryption SDK返回的加密消息。

AWS Key Management Service (AWS KMS)

AWS Encryption SDK可以使用AWS KMS keys和資料金鑰來保護您的資料，包括多區域 KMS 金鑰。例如，您可AWS Encryption SDK以設定為加密您AWS KMS keys的AWS 帳戶. 不過，您必須使用 AWS Encryption SDK 來解密該資料。

AWS Encryption SDK無法解密加密或ReEncrypt作業傳回的AWS KMS加密文字。同樣地，AWS KMS解密作業無法解密AWS Encryption SDK傳回的加密訊息。

僅AWS Encryption SDK支援對稱加密 KMS 金鑰。您無法使用非對稱 KMS 金鑰進行加密或登入AWS Encryption SDK。AWS Encryption SDK 會針對簽署訊息的演算法套件產生自己的 ECDSA 簽署金鑰。

如需決定要使用哪個程式庫或服務的說明，請參閱如何在加密服務和工具中選擇加AWS密工具或服務。

Support 與維護

AWS Encryption SDK使用 AWS SDK 和 Tools 使用的相同維護政策，包括其版本控制和生命週期階段。最佳作法是，建議您AWS Encryption SDK針對您的程式設計語言使用的最新可用版本，並在發行新版本時進行升級。當某個版AWS Encryption SDK本需要重大變更時，例如從 1.7 之前的版本升級。x 轉換為 2.0 版本。 x 和更高版本，我們提供詳細的說明來幫助您。

的每個程式設計語言實作都AWS Encryption SDK是在單獨的開放原始碼 GitHub 儲存庫中開發。每個版本的生命週期和支持階段可能因存儲庫而異。例如，給定版本的AWS Encryption SDK可能處於一種程式設計語言中的一般可用性 (完全支援) 階段，但在不同程式設計語言中的 end-of-support階段。我們建議您盡可能使用完整支援的版本，並避免使用不再受支援的版本。

若要尋找您程式設計語言AWS Encryption SDK版本的生命週期階段，請參閱每個AWS Encryption SDK儲存庫中的`SUPPORT_POLICY.rst`檔案。

- 適用於 C 的 AWS Encryption SDK— 政策支持
- AWS Encryption SDK對於. NET-支持政策

- AWS加密 CLI — 支援政策
- 適用於 JAVA 的 AWS Encryption SDK— 政策支持
- 適用於 JavaScript 的 AWS Encryption SDK— 政策支持
- 適用於 Python 的 AWS Encryption SDK— 政策支持

如需詳細資訊，請參閱 AWSSDK 的版本 AWS Encryption SDK 和工具參考指南中的和 AWS SDK 和工具維護原則。

進一步了解

如需更多有關 AWS Encryption SDK和用戶端加密的資訊，請嘗試下面參考來源。

- 如需這個 SDK 中所用名詞和概念的說明資訊，請參閱 AWS Encryption SDK中的概念。
- 如需最佳實務指導方針，請參閱AWS Encryption SDK 最佳實務。
- 如需這項 SDK 運作方式的詳細資訊，請參閱 SDK 如何運作。
- 如需展示如何在中設定選項的範例AWS Encryption SDK，請參閱設定AWS Encryption SDK。
- 如需技術層面的詳細資訊，請參閱 參考資料。
- 如需的技術規格AWS Encryption SDK，請參閱中的AWS Encryption SDK規格 GitHub。
- 有關使用問題的解答AWS Encryption SDK，請在AWS加密工具討論論壇上閱讀並發布。

如需運用不同程式設計語言實作的 AWS Encryption SDK的詳細資訊。

- C：適用於 C 的 AWS Encryption SDK請參閱 AWS Encryption SDK C 文檔和 GitHub. aws-encryption-sdk-c
- C＃/.NET：請參閱AWS Encryption SDK適用於 .NET和上aws-encryption-sdk-net存儲庫的目錄。`aws-encryption-sdk-dafny` GitHub
- 命令列介面：請參閱AWS Encryption SDK 命令列界面、閱讀AWS加密 CLI 的文件以及上的aws-encryption-sdk-cli存放庫 GitHub。
- Java：請參閱適用於 JAVA 的 AWS Encryption SDK上的「AWS Encryption SDKJava」 和「aws-encryption-sdk-java儲存庫」。 GitHub

  JavaScript：請參閱the section called “JavaScript”和上的aws-encryption-sdk-javascript儲存庫 GitHub。

- Python：請參閱適用於 Python 的 AWS Encryption SDK，AWS Encryption SDKPython 文檔和上的aws-encryption-sdk-python存儲庫 GitHub。

傳送意見回饋

我們誠摯歡迎您提供意見回饋。如果您有問題或意見、問題或報告，請使用以下資源。

- 如果您在 AWS Encryption SDK 中發現可能的安全性漏洞，請通知 AWS 安全人員。請勿建立公開 GitHub 問題。
- 若要提供有關的意見反應AWS Encryption SDK，請在 GitHub 儲存庫中針對您使用的程式設計語言提出問題。
- 若要針對此文件提供意見反應，請使用此頁面上的意見反應連結。您也可以針對本文件的開放原始碼儲存庫提出問題或貢獻 GitHub。aws-encryption-sdk-docs

AWS Encryption SDK中的概念

本節介紹 AWS Encryption SDK 中使用的概念，並提供詞彙表和參考。它旨在幫助您了解AWS Encryption SDK工作原理以及我們用來描述它的術語。

需要幫助？

- 瞭解如何AWS Encryption SDK使用信封加密來保護您的資料。
- 瞭解信封加密的元素：保護資料的資料金鑰，以及保護資料金鑰的包裝金鑰。
- 瞭解決定您使用哪些環繞金鑰的金鑰環和主要金鑰提供者。
- 瞭解為您的加密程序增加完整性的加密內容。這是可選的，但這是我們建議的最佳做法。
- 瞭解加密方法傳回的加密訊息。
- 然後，您就可以使用您偏好AWS Encryption SDK的程式設計語言。

主題

- 封套加密
- 資料金鑰
- 包裝鍵
- 金鑰圈和主金鑰提供者

- 加密內容
- 加密的訊息
- 演算法套件
- 密碼編譯資料管理員
- 對稱和非對稱加密
- 主要承諾
- 承諾政策
- 數位簽章

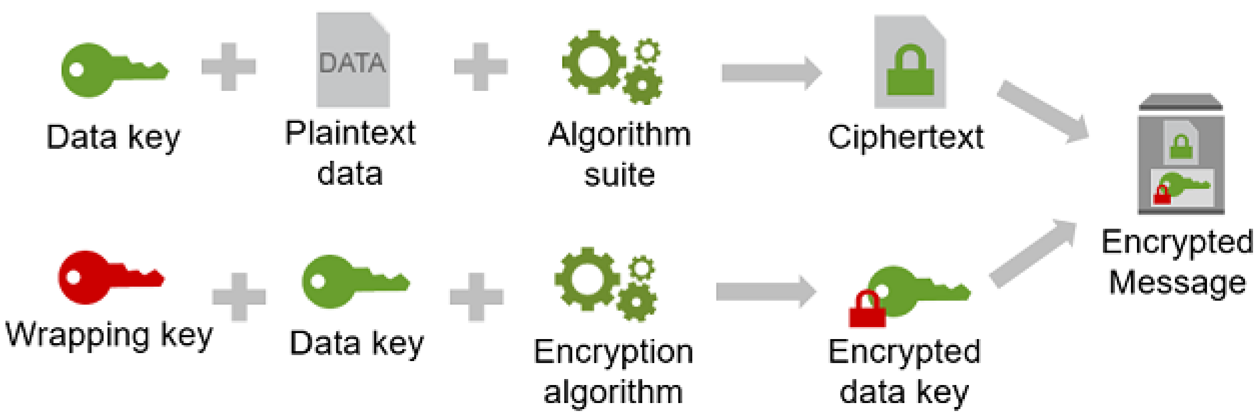
封套加密

加密資料的安全性有一部分取決於保護能夠解密資料的資料金鑰。加密處理金鑰是保護資料金鑰的一種最佳實務。若要這麼做，您需要另一個加密金鑰，稱為金鑰加密金鑰或包裝金鑰。使用包裝金鑰來加密資料金鑰的做法稱為信封加密。

保護資料金鑰

使用唯一的資料金鑰AWS Encryption SDK加密每則訊息。然後它將加密您指定的包裝密鑰下的數據密鑰。它將帶有加密數據的加密數據密鑰存儲在返回的加密消息中。

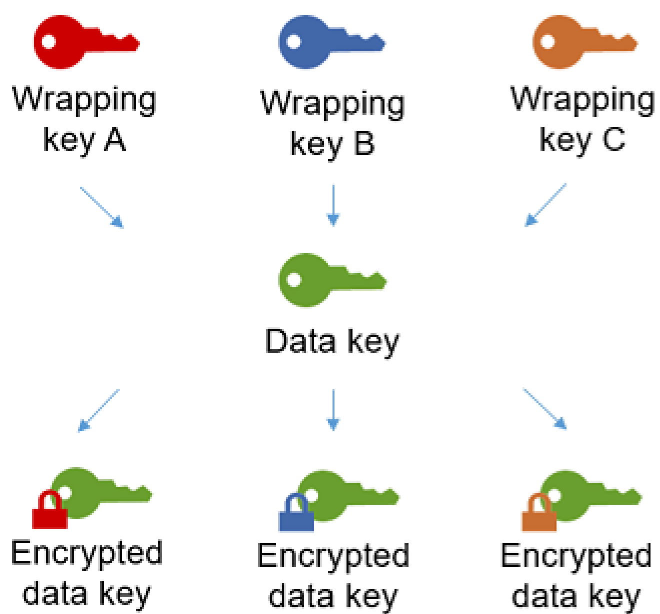
若要指定包裝金鑰，請使用金鑰圈或主要金鑰提供者。



在多個包裝密鑰下加密相同的數據

您可以在多個包裝密鑰下加密數據密鑰。您可能希望為不同的用戶提供不同的包裝鍵，或者包裝不同類型或不同位置的密鑰。每個包裝密鑰都會加密相同的數據密鑰。會將所有加密資料金鑰及加密資料AWS Encryption SDK儲存在加密訊息中。

要解密數據，您需要提供一個包裝密鑰，該密鑰可以解密其中一個加密數據密鑰。



結合多種演算法的優勢

為了加密您的數據，默認情況下，AWS Encryption SDK使用具有 AES-GCM 對稱加密，密鑰派生功能（HKDF）和簽名的複雜算法套件。若要加密資料金鑰，您可以指定適用於包裝金鑰的對稱或非對稱加密演算法。

一般而言，相較於非對稱或公有金鑰加密，對稱金鑰加密演算法速度較快，產生的加密文字較小。但是，公開金鑰演算法本質上就會區隔角色，因此金鑰管理較為方便。為了結合每個優勢，您可以使用對稱密鑰加密對稱密鑰加密數據，然後使用公鑰加密對數據密鑰進行加密。

資料金鑰

資料金鑰是 AWS Encryption SDK 用來加密資料的加密金鑰。每個資料金鑰是符合密碼編譯金鑰需求的位元組陣列。除非您使用資料金鑰快取，否則 AWS Encryption SDK 會使用唯一的資料金鑰來加密每則訊息。

您不需要指定、產生、實作、擴充、保護或使用資料金鑰。AWS Encryption SDK 會在您呼叫加密和解密操作時為您代勞。

為了保護您的資料金鑰，會使用一或多個金鑰加AWS Encryption SDK密金鑰 (稱為包裝金鑰或主金鑰) 來加密這些金鑰。當 AWS Encryption SDK 使用純文字資料金鑰來加密資料後，就會盡快從記憶體中移除它們。然後，將加密的資料金鑰連同加密的資料一起存放在加密操作傳回的已加密訊息中。如需詳細資訊，請參閱 the section called “SDK 如何運作”。

Tip

在 AWS Encryption SDK中，資料金鑰和資料加密金鑰不同。數個支援的演算法套件 (包括預設套件)，使用金鑰衍生函數來防止資料金鑰達到其密碼編譯限制。金鑰衍生函數採用資料金鑰做為輸入，並傳回實際用來加密資料的資料加密金鑰。因此，我們通常會說資料是在資料金鑰「底下」加密，而不是「由」資料金鑰加密。

每個加密的資料金鑰都包含中繼資料，包括加密之環繞金鑰的識別碼。此中繼資料可讓您更輕鬆地在AWS Encryption SDK解密時識別有效的包裝金鑰。

包裝鍵

包裝金鑰是一種金鑰加密金鑰，AWS Encryption SDK用來加密資料的資料金鑰。每個純文本數據密鑰可以在一個或多個包裝密鑰下進行加密。當您設定金鑰圈或主要金鑰提供者時，您可以決定使用哪些包裝金鑰來保護您的資料。

Note

包裝金鑰是指金鑰圈或主要金鑰提供者中的金鑰。主密鑰通常與您使用主密鑰提供者時實例化的`MasterKey`類相關聯。

AWS Encryption SDK支援數種常用的包裝金鑰，例如 AWS Key Management Service (AWS KMS) 對稱金鑰 AWS KMS keys(包括多區域 KMS 金鑰)、原始 AES-GCM (進階加密標準/Galois 計數器模式) 金鑰，以及原始 RSA 金鑰。您也可以擴展或實現自己的包裝鍵。

使用信封加密時，您需要保護包裝密鑰免受未經授權的訪問。您可以透過下列任何一種方式執行此操作：

- 使用專為這個用途所設計的 Web 服務，例如 AWS Key Management Service (AWS KMS)。
- 使用硬體安全模組 (HSM)，例如 AWS CloudHSM 所提供的功能。
- 使用其他金鑰管理工具和服務。

如果您沒有金鑰管理系統，則建議您使用 AWS KMS。與AWS Encryption SDK整合可協AWS KMS助您保護和使用包裝金鑰。但是，AWS Encryption SDK不需要AWS或任何AWS服務。

金鑰圈和主金鑰提供者

若要指定用於加密和解密的包裝金鑰，請使用金鑰環 (C、C# /.NET 和 JavaScript) 或主要金鑰提供者 (Java、Python、CLI)。您可以使用提AWS Encryption SDK供或設計您自己實作的金鑰環和主要金鑰提供者。根據語言限制，AWS Encryption SDK提供彼此相容的金鑰環和主金鑰提供者。如需詳細資訊，請參閱 Keyring 相容性。

Keying 會產生、加密和解密資料金鑰。定義金鑰環時，您可以指定加密資料金鑰的包裝金鑰。大多數金鑰圈至少會指定一個包裝金鑰或提供並保護包裝金鑰的服務。您也可以使用額外的配置選項來定義鑰匙圈，也可以定義更複雜的鑰匙圈。如需選擇和使用AWS Encryption SDK定義之鑰匙圈的說明，請參閱使用 keyring。在 C、C # / .NET 和版本 3 中都支持密鑰環。 JavaScript 的 x 的適用於 JAVA 的 AWS Encryption SDK。

主要金鑰提供者是金鑰圈的替代方案。主金鑰提供者會傳回您指定的包裝金鑰 (或主金鑰)。每個主金鑰都關聯至一個主金鑰提供者，但主金鑰提供者通常可提供多個主金鑰。Java、Python 和AWS加密 CLI 支援主要金鑰提供者。

您必須為加密指定金鑰環 (或主要金鑰提供者)。您可以指定相同的金鑰環 (或主要金鑰提供者) 或不同的金鑰環來進行解密。加密時，AWS Encryption SDK會使用您指定的所有包裝金鑰來加密資料金鑰。解密時，僅AWS Encryption SDK使用您指定的包裝金鑰來解密加密的資料金鑰。指定用於解密的包裝密鑰是可選的，但這是一種AWS Encryption SDK最佳實踐。

如需有關指定環繞鍵的詳細資訊，請參閱選擇包裝鍵。

加密內容

為了改進密碼編譯操作的安全性，請在所有加密資料請求中包含加密內容。使用加密內容是選用的，但卻是建議的密碼編譯最佳實務。

加密內容是一組名稱/值對，其中包含任意非私密的額外驗證資料。加密內容可以包含您選擇的任何資料，但通常包含有利於記錄和追蹤的資料，例如有關檔案類型、用途或擁有權的資料。當您加密資料時，加密內容會以密碼演算法繫結至加密的資料，因此在解密資料時需要相同的加密內容。AWS Encryption SDK 在它傳回的已加密訊息的標頭中，以純文字包含加密內容。

AWS Encryption SDK使用的加密內容包含您指定的加密內容以及加密材料管理員 (CMM) 新增的公開 key pair。具體而言，當您使用加密演算法搭配簽署時，CMM 會將名稱/值對新增到加密內容，其中包含預留名稱 `aws-crypto-public-key` 和代表公有驗證金鑰的值。加密內容中的 `aws-crypto-public-key` 名稱由 AWS Encryption SDK 保留，不能在加密內容中的任何其他配對中做為名稱。如需詳細資訊，請參閱郵件格式參考中的 AAD。

以下範例加密內容包含請求中指定的兩個加密內容對，以及 CMM 新增的公有金鑰對。

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

若要解密資料，您需傳入已加密訊息。由於 AWS Encryption SDK 可以從加密的訊息標頭擷取加密內容，您不需要另外提供加密內容。不過，加密內容可協助您確認您正在解密正確的已加密訊息。

- 在 AWS Encryption SDK 命令列界面 (CLI)，如果您在解密命令中提供加密內容，CLI 在傳回純文字資料之前會驗證值是否存在於已加密訊息的加密內容中。
- 在其他程式設計語言實作中，解密回應包括加密內容和純文字資料。您應用程式中的解密函數在傳回純文字資料之前，應該一律驗證解密回應中的加密內容包含解密請求中的加密內容 (或子集)。

> **Note**
>
> 使用版本 4。 AWS Encryption SDK適用於 .NET 和版本 3 的 x。 x 的 適用於 JAVA 的 AWS Encryption SDK，您可以在具有必要加密內容 CMM 的所有加密請求中要求具有加密內容。

選擇加密內容時，請記住，它不是秘密。加密內容會以純文字顯示在傳回的加密郵件標頭中AWS Encryption SDK。如果您使用 AWS Key Management Service，加密內容也可能出現在稽核記錄和日誌的純文字中，例如 AWS CloudTrail。

如需在程式碼中提交和驗證加密內容的範例，請參閱您慣用程式設計語言的範例。

加密的訊息

使用 AWS Encryption SDK 加密資料時，它會傳回加密的訊息。

加密訊息是一種可攜式格式化的資料結構，其中包含加密的資料以及資料金鑰的加密副本、演算法識別碼，以及選擇性地加密內容和數位簽章。AWS Encryption SDK中的加密操作會傳回已加密訊息，而解密操作會將已加密訊息當做輸入。

結合加密的資料與其加密的資料金鑰可以簡化解密操作，您也不用再將加密的資料金鑰，於其進行加密的資料分開來存放和管理。

如需已加密訊息的相關技術資訊，請參閱加密的訊息格式。

演算法套件

AWS Encryption SDK使用演算法套件來加密和簽署加密訊息中的資料，加密和解密作業會傳回。AWS Encryption SDK 支援數個演算法套件。所有支援的套件都使用進階加密標準 (AES) 做為主要演算法，並將它與其他演算法和值結合。

AWS Encryption SDK會建立建議的演算法套件做為所有加密操作的預設套件。預設套件可能隨著標準和最佳實務改進而變更。您可以在要求加密資料或建立加密材料管理員 (CMM) 時指定替代演算法套件，但除非您的情況需要其他演算法套件，否則最好使用預設值。目前的預設值為 AES-GCM，具有 HMAC 型 extract-and-expand 金鑰衍生函數 (HKDF)、金鑰承諾、橢圓曲線數位簽章演算法 (ECDSA) 簽章，以及 256 位元加密金鑰。

如果您的應用程式需要高效能，而且正在加密資料的使用者和解密資料的使用者都受到同等信任，您可以考慮指定沒有數位簽章的演算法套件。但是，我們強烈建議使用包含金鑰承諾和金鑰衍生函數的演算法套件。沒有這些功能的演算法套件僅支援回溯相容性。

密碼編譯資料管理員

密碼材料管理器（CMM）組合用於加密和解密數據的加密材料。密碼編譯資料包含純文字和加密的資料金鑰，以及選用的訊息簽署金鑰。您永遠不會直接與 CMM 互動。加密和解密方法會為您代勞。

您可以使用預設 CMM 或AWS Encryption SDK提供的快取 CMM，或撰寫自訂 CMM。你可以指定一個 CMM，但它不是必需的。當您指定金鑰環或主要金鑰提供者時，會為您AWS Encryption SDK建立預設 CMM。預設 CMM 會從您指定的金鑰環或主要金鑰提供者取得加密或解密資料。這可能牽涉到呼叫密碼編譯服務，例如 AWS Key Management Service (AWS KMS)。

由於 CMM 充當AWS Encryption SDK和金鑰環 (或主要金鑰提供者) 之間的聯絡，因此它是自訂和延伸 (例如對原則強制執行和快取的支援) 的理想選擇。AWS Encryption SDK 提供快取 CMM 來支援資料金鑰快取。

對稱和非對稱加密

對稱加密使用相同的金鑰來加密和解密資料。

非對稱加密使用與數學相關的資料 key pair。配對中的一個金鑰會加密資料；只有配對中的另一個金鑰可以解密資料。如需詳細資訊，請參閱加密服務和工具指南中的AWS加密演算法。

使AWS Encryption SDK用信封加密。它使用對稱數據密鑰加密您的數據。它使用一個或多個對稱或非對稱包裝密鑰對稱數據密鑰進行加密。它返回一個加密的消息，其中包括加密的數據和數據密鑰的至少一個加密副本。

加密您的資料 (對稱加密)

若要加密資料，AWS Encryption SDK會使用對稱資料金鑰和包含對稱加密演算法的演算法套件。要解密數據，AWS Encryption SDK使用相同的數據密鑰和相同的算法套件。

加密您的資料金鑰 (對稱或非對稱加密)

您提供給加密和解密作業的金鑰環或主金鑰提供者會決定對稱資料金鑰的加密與解密方式。您可以選擇使用對稱加密 (例如金鑰環) 的金鑰環或主要金鑰提供者，或使用非對稱加密的金AWS KMS鑰環或主要金鑰提供者 (例如 RSA 金鑰環或)。`JceMasterKey`

主要承諾

AWS Encryption SDK支持密鑰承諾（有時稱為魯棒性），這是一種保證每個密文只能解密為單個明文的安全屬性。為此，金鑰承諾保證只會使用加密郵件的資料金鑰進行解密。使用關鍵承諾進行加密和解密是AWS Encryption SDK最佳實務。

大多數現代對稱密碼 (包括 AES) 會在單一密碼金鑰下加密純文字，例如AWS Encryption SDK用來加密每個明文訊息的唯一資料金鑰。使用相同的資料金鑰解密此資料會傳回與原始資料完全相同的純文字。使用不同的金鑰解密通常會失敗。但是，可以在兩個不同的密鑰下解密密文。在極少數情況下，找到一個可以將幾個字節的密文解密為不同但仍然可以理解的明文密鑰是可行的。

AWS Encryption SDK總是在一個唯一的數據密鑰下加密每個明文消息。它可能會在多個包裝密鑰（或主密鑰）下加密該數據密鑰，但包裝密鑰始終加密相同的數據密鑰。不過，複雜、手動製作的加密訊息實際上可能包含不同的資料金鑰，每個金鑰都由不同的包裝金鑰加密。例如，如果一個用戶解密加密消息，則返回 0x0（false），而另一個用戶解密相同的加密消息將獲得 0x1（true）。

為了避免發生這種情況，在加密和解密時AWS Encryption SDK支援金鑰承諾。使用金鑰承諾AWS Encryption SDK加密訊息時，會以密碼方式將產生密碼文字的唯一資料金鑰繫結至金鑰承諾字串 (非秘密資料金鑰識別碼)。然後，它會將金鑰承諾字串儲存在加密郵件的中繼資料中。當它解密含有金鑰承諾的訊息時，AWS Encryption SDK會驗證資料金鑰是該加密訊息的唯一金鑰。如果資料金鑰驗證失敗，解密作業就會失敗。

1.7 版中引入了對關鍵承諾的 Support。 x，它可以使用密鑰承諾解密消息，但不會使用密鑰承諾進行加密。您可以使用此版本完全部署使用金鑰承諾來解密密文的功能。版本 2.0。 x 包括對關鍵承諾的全面支持。默認情況下，它僅使用密鑰承諾進行加密和解密。對於不需要解密由舊版的. AWS Encryption SDK

雖然使用關鍵承諾進行加密和解密是最佳做法，但我們可以讓您決定何時使用它，並讓您調整採用它的速度。從 1.7 版開始。 x, AWS Encryption SDK 支援承諾政策，用於設定預設演算法套件，並限制可能使用的演算法套件。此政策決定您的資料是否經過金鑰承諾加密和解密。

金鑰承諾會產生稍大 (+ 30 位元組) 的加密訊息，並且需要更多時間來處理。如果您的應用程式對大小或效能非常敏感，您可以選擇退出主要承諾用量。但是，只有在必須的情況下才這樣做。

如需有關移轉至 1.7 版的詳細資訊。 X 和 2.0。 x，包括其主要承諾產品功能，請參閱遷移您的AWS Encryption SDK。如需關鍵承諾產品的技術資訊，請參閱the section called “演算法參考”和the section called “訊息格式參考”。

承諾政策

承諾產品原則是一種組態設定，可決定您的應用程式是否使用金鑰承諾進行加密和解密。使用關鍵承諾進行加密和解密是AWS Encryption SDK最佳實務。

承諾政策有三個價值觀。

Note

您可能必須水平或垂直捲動才能看到整個表格。

承諾政策價值

值	使用金鑰承諾進行加密	無需金鑰承諾即可加密	使用關鍵承諾進行解密	無需金鑰承諾即可解密
ForbidEncryptAllowDecrypt	✗	✓	✓	✓
RequireEncryptAllowDecrypt	✓	✗	✓	✓
RequireEncryptRequireDecrypt	✓	✗	✓	✗

承諾政策設置在AWS Encryption SDK版本 1.7 中引入。 x. 它在所有支持的編程語言中都有效。

- `ForbidEncryptAllowDecrypt`解密有或沒有密鑰承諾，但它不會用密鑰承諾進行加密。這是 1.7 版中承諾政策的唯一有效值。 x，它用於所有加密和解密操作。它的設計旨在準備所有執行您應用程式的主機，以便在遇到以金鑰承諾加密的密文之前使用金鑰承諾進行解密。
- `RequireEncryptAllowDecrypt`始終使用密鑰承諾進行加密。它可以解密有或沒有密鑰承諾。這個值，在 2.0 版中引入。 x，可讓您開始使用金鑰承諾進行加密，但仍可在沒有金鑰承諾的情況下解密舊密文。
- `RequireEncryptRequireDecrypt`僅使用金鑰承諾進行加密和解密。此值是 2.0 版的預設值。 x. 當您確定所有密文都已使用金鑰承諾加密時，請使用此值。

承諾產品原則設定會決定您可以使用的演算法套件。從 1.7 版開始。 x，AWS Encryption SDK支持密鑰承諾的算法套件; 有簽名和不簽名。如果您指定的演算法套件與承諾原則衝突，就會AWS Encryption SDK傳回錯誤。

如需設定承諾產品原則的說明，請參閱設定承諾產品原則。

數位簽章

若要確保數位訊息在系統間傳送時的完整性，您可以將數位簽章套用至郵件。數位簽章永遠是不對稱的。您可以使用私鑰來創建簽名，並將其附加到原始消息中。您的收件者會使用公開金鑰來確認郵件在您簽署後尚未修改。

使用經過驗證的加AWS Encryption SDK密演算法 AES-GCM 來加密您的資料，而解密程序會在不使用數位簽章的情況下驗證加密訊息的完整性和真實性。但是由於 AES-GCM 使用對稱金鑰，任何可以解密用來解密密文的資料金鑰的人也可以手動建立新的加密密文字，進而造成潛在的安全性考量。例如，如果您使用金AWS KMS鑰做為包裝金鑰，這表示具有 KMS 解密權限的使用者可以在不呼叫 KMS Encrypt 的情況下建立加密的加密文字。

為避免此問題，AWS Encryption SDK支援在加密郵件結尾加入橢圓曲線數位簽章演算法 (ECDSA) 簽章。使用簽署演算法套件時，會為每個加密訊息AWS Encryption SDK產生一個暫時的私密金鑰和公開 key pair。會將公開金鑰AWS Encryption SDK儲存在資料金鑰的加密內容中，並捨棄私密金鑰，而且沒有人可以建立另一個使用公開金鑰進行驗證的簽章。因為演算法會將公開金鑰與加密的資料金鑰做為訊息標頭中的其他驗證資料繫結，因此只能解密郵件的使用者無法更改公開金鑰。

簽名驗證在解密方面會增加顯著的效能成本。如果加密資料的使用者和解密資料的使用者同樣受到信任，請考慮使用不包含簽章的演算法套件。

AWS Encryption SDK 的運作方式

本節中的工作流程說明如何AWS Encryption SDK加密資料和解密加密的訊息。這些工作流程描述了使用默認功能的基本過程。如需有關定義和使用自訂元件的詳細資訊，請參閱每個支援語言實作的 GitHub 儲存庫。

AWS Encryption SDK使用信封加密來保護您的數據。每個消息都在一個唯一的數據密鑰下進行加密。然後，數據密鑰將由您指定的包裝密鑰進行加密。若要解密加密的郵件，會AWS Encryption SDK使用您指定的包裝金鑰來解密至少一個加密的資料金鑰。然後，它可以解密密文並返回一個明文消息。

需要我們在中使用的術語方面的幫助AWS Encryption SDK嗎？ 請參閱the section called “概念”。

如何加AWS Encryption SDK密數據

提AWS Encryption SDK供加密字串、位元組陣列和位元組串流的方法。如需程式碼範例，請參閱每個程式設計語言章節中的「範例」主題。

1. 建立金鑰環 (或主要金鑰提供者)，以指定保護資料的環繞金鑰。
2. 將金鑰環和純文字資料傳遞至加密方法。我們建議您傳入選用的非機密加密內容。
3. 加密方法會詢問金鑰圈是否有加密材料。金鑰圈會傳回訊息的唯一資料加密金鑰：一個純文字資料金鑰，以及由每個指定的包裝金鑰加密的該資料金鑰的一個複本。
4. 這項加密方法會使用純文字資料金鑰來加密資料，接著再捨棄該純文字資料金鑰。如果您提供加密內容 (AWS Encryption SDK最佳作法)，加密方法會以密碼方式將加密內容繫結至加密的資料。
5. 加密方法會傳回包含加密資料、加密資料金鑰和其他中繼資料 (包括加密內容) 的加密訊息 (如果您使用的話)。

如何解AWS Encryption SDK密加密的郵件

提AWS Encryption SDK供解密加密郵件並傳回純文字的方法。如需程式碼範例，請參閱每個程式設計語言章節中的「範例」主題。

解密加密郵件的金鑰圈 (或主要金鑰提供者) 必須與用來加密郵件的金鑰環相容。其中一個包裝金鑰必須能夠解密加密訊息中的加密資料金鑰。如需有關金鑰環和主要金鑰提供者之相容性的資訊，請參閱the section called “Keyring 相容性”。

1. 建立金鑰環或主金鑰提供者，其中包含可以解密資料的金鑰環或主金鑰提供者。您可以使用提供給加密方法的相同金鑰圈，也可以使用不同的金鑰圈。

2. 將加密的消息和密鑰環傳遞給解密方法。
3. 解密方法會要求金鑰環或主要金鑰提供者解密加密訊息中的其中一個加密資料金鑰。它會從加密的訊息傳入資訊，包括加密的資料金鑰。
4. keyring 使用其包裝金鑰來解密其中一個加密的資料金鑰。如果成功，則回應會包含純文字資料金鑰。如果金鑰環或主要金鑰提供者所指定的包裝金鑰都無法解密加密的資料金鑰，則解密呼叫會失敗。
5. 解密方法會使用純文字資料金鑰來解密資料、捨棄純文字資料金鑰，然後傳回純文字資料。

AWS Encryption SDK中的支援演算法套件

演算法套件是加密演算法與相關數值的集合。密碼編譯系統使用演算法實作來產生加密文字訊息。

AWS Encryption SDK演算法套件會使用 Galois/Counter Mode (GCM) 中的進階加密標準 (AES) 演算法 (簡稱為 AES-GCM) 來加密原始資料。所以此AWS Encryption SDK支援 256 位元、192 位元和 128 位元的加密金鑰。初始向量 (IV) 的長度一律是 12 個位元組。驗證標籤的長度一律是 16 個位元組。

在預設情況下，AWS Encryption SDK使用具有 AES-GCM 的算法套件，並且具有基於 HMAC 的 extract-and-expand鍵派生函數（香港發展基金）、簽署和 256 位元加密金鑰。如果承諾需要關鍵承諾，AWS Encryption SDK選擇同時支持密鑰承諾的算法套件；否則，它會選擇具有密鑰派生和簽名的算法套件，但不是關鍵承諾。

建議：AES-GCM，搭配金鑰衍生、簽署和關鍵承諾

所以此AWS Encryption SDK推薦一個算法套件，該算法套件通過向基於 HMAC 的extract-and-expand 密鑰派生函數（HKDF）。所以此AWS Encryption SDK會添加橢圓曲線數位簽章演算法 (ECDSA) 簽章。支援關鍵承諾，這個算法套件也派生一個密鑰承諾字符串— 一個非機密的資料金鑰標識符，存放在中繼資料的中繼資料。此密鑰承諾字符串也通過 HKDF 派生，使用類似於派生數據加密密鑰的過程。

AWS Encryption SDK演算法套件

加密演算法	資料加密金鑰長度 (以位元為單位)	金鑰衍生演算法	簽章演算法	A. 主要承諾
AES-GCM	256	HKDF，SHA-384 式	ECDSA，P-384 和 SHA-384 式	HKDF，SHA-512 式

HKDF 可協助您避免意外重複使用資料加密金鑰，並降低資料金鑰過度使用的風險。

對於簽署，這個演算法套件會搭配加密哈希函數演算法 (SHA-384) 來使用 ECDSA。預設會使用 ECDSA，即使基礎主金鑰政策未指定使用。簽署訊息驗證郵件發件人是否有權加密郵件並提供不可否認性。當主金鑰的授權政策允許一組使用者進行資料加密，並允許另外一組使用者進行資料解密時，這種做法特別有用。

具有關鍵承諾的算法套件可確保每個密文只解密為一個明文。他們通過驗證用作加密算法輸入的數據密鑰的身份來完成此操作。加密時，這些算法套件派生一個密鑰承諾字符串。在解密之前，他們會驗證數據密鑰是否與密鑰承諾字符串匹配。如果沒有，解密呼叫就會失敗。

其他支援的演算法套件

AWS Encryption SDK 支援以下替代演算法套件，以提供回溯相容性。一般而言，我們不建議這些演算法套件。但是，我們認識到簽名可能會嚴重影響性能，因此我們為這些情況提供了一個帶有密鑰派生的密鑰提交套件。對於必須做出更重要的性能權衡的應用程序，我們將繼續提供缺乏簽名、關鍵承諾和密鑰派生的套件。

AES-GCM，不搭配金鑰承諾

沒有密鑰承諾的算法套件不會在解密之前驗證數據密鑰。因此，這些算法套件可能會將單個密文解密為不同的明文消息。但是，由於具有關鍵承諾的算法套件會產生稍大（+30 字節）的加密消息並且需要更長的時間來處理，但它們可能不是每個應用程序的最佳選擇。

所以此AWS Encryption SDK支持具有密鑰派生、密鑰承諾、簽名以及具有密鑰派生和密鑰承諾但不簽名的算法套件。我們不建議您使用沒有金鑰承諾的演算法套件。如果必須，我們建議使用具有密鑰派生和密鑰承諾的算法套件，但不要簽名。但是，如果您的應用程序性能配置文件支持使用算法套件，則使用具有密鑰承諾、密鑰派生和簽名的算法套件是最佳做法。

AES-GCM，不簽署

不搭配簽署的演算法套件會缺少提供真偽和不可否認性的 ECDSA 簽章。僅當負責資料加密和資料解密的用户之間彼此信任時，才能使用這些套件。

在沒有簽名的情況下使用算法套件時，我們建議您選擇具有密鑰派生和密鑰承諾的算法套件。

AES-GCM，不搭配金鑰衍生

不搭配金鑰衍生的演算法套件會使用資料加密金鑰做為 AES-GCM 加密金鑰，而不使用金鑰衍生函數來衍生唯一金鑰。我們不鼓勵使用這個套件來產生加密文字，但AWS Encryption SDK出於兼容性原因支持它。

如需這些套件在程式庫中如何表示與使用的詳細資訊，請參閱the section called “演算法參考”。

搭配 AWS KMS 使用 AWS Encryption SDK

若要使用AWS Encryption SDK，您需要使用包裝金鑰來設定金鑰環或主要金鑰提供者。如果您沒有金鑰基礎架構，建議您使用 AWS Key Management Service (AWS KMS)。中的許多程式碼範例都AWS Encryption SDK需要 AWS KMS key.

若要與互動AWS KMS，AWS Encryption SDK需要您慣用程式設計語言的AWS SDK。用AWS Encryption SDK戶端程式庫與AWS SDK 搭配使用，以支援儲存在中的主金鑰AWS KMS。

準備使用 AWS KMS 搭配 AWS Encryption SDK

1. 建立 AWS 帳戶。如需說明，請參閱如何建立和啟用新的 Amazon Web Services 帳戶？ 在AWS知識中心。
2. 建立對稱密存取AWS KMS key。如需說明，請參閱AWS Key Management Service開發人員指南中的建立金鑰。

> **Tip**
>
> 若要以程式設計方式使用 AWS KMS key，您需要 AWS KMS key 的金鑰 ID 或 Amazon Resource Name (ARN)。如需尋找 ID 或 ARN 的說明AWS KMS key，請參閱AWS Key Management Service開發人員指南中的尋找金鑰 ID 和 ARN。

3. 產生存取金鑰。存取金鑰。您可以使用 IAM 使用者的存取金鑰 ID 和秘密存取金鑰，也可以使用建立包含存AWS Security Token Service取金鑰 ID、秘密存取金鑰和工作階段 Token 的臨時安全登入資料的新工作階段。基於安全最佳實務，建議您使用臨時憑證來存取。建議您使用AWS臨時憑證。

   若要使用存取金鑰建立 IAM 使用者，請參閱 IAM 使用者指南中的建立 IAM 使用者。

   若要產生臨時安全登入資料，請參閱 IAM 使用者指南中的要求臨時安全登入資料。
4. 使用、AWS SDK for Python (Boto)或 AWS SDK for C++(C) 中的指示 AWS SDK for JavaAWS SDK for JavaScript，以及您在步驟 3 中產生的存取金鑰 ID 和秘密存取金鑰來設定您的AWS認證。如果您產生了臨時認證，則還需要指定會話令牌。

   此程序可允許 AWS 軟體開發套件為您簽署向 AWS 提出的請求。AWS Encryption SDK中與 AWS KMS 互動的程式碼範例假設您已完成此步驟。
5. 下載並安裝 AWS Encryption SDK。若要了解做法，請參閱您想使用之程式設計語言的安裝指示。

AWS Encryption SDK 最佳實務

所以此AWS Encryption SDK旨在讓您輕鬆地使用產業標準和最佳實務來保護資料。雖然在預設值中為您選取了許多最佳作法，但有些作法是選擇性的，但建議您隨時可行。

使用最新版本

當你開始使用AWS Encryption SDK，使用您偏好的最新版本程式設計語言。如果您一直在使用AWS Encryption SDK，請盡快升級至每個最新版本。這可確保您使用建議的組態，並利用新的安全性屬性來保護您的資料。如需有關支援版本的詳細資訊，包括移轉和部署指南，請參閱Support與維護和的版本 AWS Encryption SDK。

如果新版本棄用程式碼中的元素，請盡快取代它們。棄用警告和代碼註釋通常會推薦一個很好的替代方案。

為了讓重大升級更容易且不容易發生錯誤，我們偶爾會提供暫時或過渡版本。使用這些版本及其隨附的文件，以確保您可以在不中斷生產工作流程的情況下升級應用程式。

使用預設值

所以此AWS Encryption SDK將最佳實踐設計為其預設值。只要有可能，使用它們。對於默認不切實際的情況，我們提供替代方案，例如沒有簽名的算法套件。我們還為高級用戶提供自定義的機會，例如自定義密鑰環，主密鑰提供商和加密材料管理器（CMM）。謹慎使用這些高級替代方案，並盡可能通過安全工程師驗證您的選擇。

使用加密內容

為了改進密碼編譯操作的安全性，請包含加密內容在加密數據的所有請求中都有一個有意義的值。使用加密內容是選用的，但卻是建議的密碼編譯最佳實務。加密內容提供額外的驗證資料 (AAD)，包括AWS Encryption SDK。儘管它不是秘密，但是加密內容可以幫助您保護完整性和真實性您的加密數據。

在 中AWS Encryption SDK，您只能在加密時指定加密內容。解密時，AWS Encryption SDK在加密訊息的標頭中使用加密內容AWS Encryption SDK傳回：在您的應用程式傳回純文字資料之前，請確認您用來加密訊息的加密內容包含在解密訊息所用的加密內容中。如需詳細資訊，請參閱程式設計語言中的範例。

當您使用命令列界面時，AWS Encryption SDK驗證您的加密內容。

保護您的包裝鑰匙

所以此AWS Encryption SDK產生唯一的資料金鑰來加密每則純文字訊息。然後，它使用包裝您提供的密鑰來加密數據密鑰。如果您的包裝金鑰遺失或刪除，您的加密資料將無法復原。如果您的密鑰不安全，則您的數據可能容易受到攻擊。

使用包裝受安全金鑰基礎架構保護的金鑰，例如AWS Key Management Service(AWS KMS。當您使用原始 AES 或原始 RSA 金鑰時，請使用符合您安全要求的隨機來源。產生並儲存包裝金鑰，並在硬體安全模組 (HSM) 中產生並儲存包裝金鑰，例如AWS CloudHSM，是最佳實務。

使用金鑰基礎結構的授權機制，將包裝金鑰的存取限制為只有需要金鑰的使用者。落實最佳實務原則，例如最低權限。當您使用AWS KMS keys，使用實務的金鑰政策和 IAM 政策最佳實務原則。

指定您的包裝鍵

這始終是最佳做法指定您的包裝鍵在解密和加密時明確。當你這樣做，AWS Encryption SDK只會使用您指定的金鑰。這種做法可確保您只使用您想要的加密金鑰。適用於AWS KMS包裝密鑰，它還可以通過防止您無意中使用其他密鑰來提高性能AWS 帳戶或區域，或嘗試使用您沒有權限使用的密鑰進行解密。

加密時，密鑰環和主密鑰提供者AWS Encryption SDK耗材需要您指定包裝鍵。它們使用所有且僅使用您指定的包裝鍵。使用原始 AES 金鑰環、原始 RSA 金鑰圈和 JCE 進行加密和解密時，您還需要指定包裝金鑰MasterKeys。

但是，當使用解密時AWS KMS金鑰圈和主要金鑰提供者，您不需要指定包裝金鑰。所以此AWS Encryption SDK可以從加密資料金鑰的中繼資料中取得金鑰識別碼。但是，我們建議指定包裝索引鍵是我們建議的最佳實務。

若要在使用時支援此最佳作法AWS KMS包裝金鑰，我們建議以下內容：

- 使用AWS KMS指定包裝金鑰的鑰匙圈。加密和解密時，這些金鑰環只會使用您指定的指定環繞金鑰。
- 當您使用AWS KMS主密鑰和主密鑰提供程序，使用中引入的嚴格模式構造函數1.7 版本.x的AWS Encryption SDK。他們創建僅使用您指定的包裝密鑰進行加密和解密的提供程序。在 1.7 版中，一律以任何包裝金鑰進行解密的主金鑰提供者的建構函式會被棄用。x並在 2.0 版本中刪除。x。

指定時AWS KMS包裝密鑰進行解密是不切實際的，您可以使用發現提供程序。所以此AWS Encryption SDK在 C 和 JavaScript 支持AWS KMS探索 keyring。具有探索模式的主要金鑰提供者可在 1.7 版中使用 Java 和 Python。x和更高版本. 這些探索提供者，僅用於解密AWS KMS包裝密鑰，明確指示AWS Encryption SDK使用任何加密資料金鑰的包裝金鑰。

如果您必須使用探索提供者，請使用其探索篩選器功能來限制他們使用的包裝鍵。例如，AWS KMS區域探索 Keyring只使用特定的包裝鍵AWS 區域。您也可以設定AWS KMSkeyring 和AWS KMS 主金鑰提供者僅使用包裝金鑰尤其是AWS 帳戶。此外，與往常一樣，使用金鑰政策和 IAM 政策來控制您的存取AWS KMS包裝鍵。

使用數位簽章

使用帶有簽名的算法套件是最佳實踐。數位簽章驗證郵件寄件者已獲授權傳送郵件並保護郵件的完整性。所有版本AWS Encryption SDK默認情況下使用帶簽名的算法套件。

如果您的安全性需求不包含數位簽章，您可以選取不含數位簽章的演算法套件。不過，我們建議您使用數位簽章，尤其是當一群使用者加密資料，而另一組使用者將資料解密時。

使用金鑰承諾

使用關鍵承諾安全性功能是最佳作法。通過驗證唯一的身分資料金鑰加密了您的數據，金鑰承諾防止您解密可能導致多個純文本消息的任何密文。

所以此AWS Encryption SDK提供完整的加密和解密支援，並從中開始提供關鍵承諾2.0 版本.x。根據預設，您的所有郵件都會透過金鑰承諾進行加密和解密。1.7 版本.x的AWS Encryption SDK可以使用密鑰承諾解密密文。它旨在幫助舊版的用戶部署 2.0 版本。x成功地。

主要承諾的 Support 包括新的演算法套件和一個新的訊息格式產生的密文只比密文大 30 個字節，沒有密鑰承諾。該設計將其對性能的影響降到最低，因此大多數用戶可以享受關鍵承諾的好處。如果您的應用程式對大小和效能非常敏感，您可能會決定使用承諾政策設定以停用金鑰承諾產品或允許AWS Encryption SDK在沒有承諾的情況下解密郵件，但只有在必須的情況下才這樣做。

限制加密資料金鑰的數量

這是最佳做法限制加密資料金鑰的數量在您解密的郵件中，尤其是來自不受信任來源的郵件。使用無法解密的大量加密資料金鑰來解密訊息可能會造成延長延遲、耗盡費用、限制您的應用程式以及其他共用您帳戶的資訊，並可能耗盡您的金鑰基礎結構。沒有限制，加密訊息最多可以有 65,535 (2^16-1) 的加密資料金鑰。如需詳細資訊，請參閱 限制加密的資料金鑰。

如需有關的詳細資訊AWS Encryption SDK根據這些最佳實務的安全性能，請參閱改進了用戶端加密：explicit KeyIds 和金鑰承諾中的AWS安全性部落格。

設定AWS Encryption SDK

設計AWS Encryption SDK為易於使用。雖然AWS Encryption SDK有數個組態選項，但預設值是經過仔細選擇，以便在大多數應用程式中實用且安全。不過，您可能需要調整組態以改善效能，或在設計中包含自訂功能。

設定實作時，請檢閱最AWS Encryption SDK佳做法並儘可能多地實作。

主題

- 選擇一種編程語言
- 選擇包裝鍵
- 使用多地區 AWS KMS keys
- 選擇演算法套件
- 限制加密的資料金鑰
- 建立探索篩選器
- 設定承諾產品原則
- 使用串流資料
- 快取資料金鑰

選擇一種編程語言

提供多種程式設計語言版本AWS Encryption SDK。語言實現被設計為完全可互操作，並提供相同的功能，儘管它們可能以不同的方式實現。一般而言，您會使用與應用程式相容的程式庫。不過，您可以為特定實作選取程式設計語言。例如，如果您偏好使用金鑰圈，您可以選擇適用於 C 的 AWS Encryption SDK或。適用於 JavaScript 的 AWS Encryption SDK

選擇包裝鍵

會AWS Encryption SDK產生唯一的對稱資料金鑰來加密每則訊息。除非您使用資料金鑰快取，否則您不需要設定、管理或使用資料金鑰。AWS Encryption SDK它為你做。

但是，您必須選取一或多個包裝金鑰來加密每個資料金鑰。AWS Encryption SDK支援不同大小的 AES 對稱金鑰和 RSA 非對稱金鑰。它還支持 AWS Key Management Service（AWS KMS）對稱加密AWS KMS keys。您必須為包裝金鑰的安全性和耐久性負責，因此我們建議您在硬體安全性模組或金鑰基礎架構服務 (例如) 中使用加密金鑰AWS KMS。

若要指定用於加密和解密的包裝金鑰，請使用金鑰環 (C 和 JavaScript) 或主要金鑰提供者 (Java、Python、AWS加密 CLI)。您可以指定相同或不同類型的一個包裝鍵或多個包裝鍵。如果您使用多個包裝金鑰來包裝資料金鑰，每個包裝金鑰都會加密相同資料金鑰的副本。加密的數據密鑰（每個包裝密鑰一個）與加密數據一起存儲在AWS Encryption SDK返回的加密消息中。若要解密資料，AWS Encryption SDK必須先使用其中一個包裝金鑰來解密加密的資料金鑰。

若要AWS KMS key在金鑰環或主要金鑰提供者中指定，請使用支援的AWS KMS金鑰識別碼。如需金鑰的金鑰識別碼的詳細資訊，請參閱AWS Key Management Service開發人員指南中的金鑰識別碼。AWS KMS

- 使用適用於 JAVA 的 AWS Encryption SDK、、或AWS加密 CLI 進行加密時 適用於 JavaScript 的 AWS Encryption SDK適用於 Python 的 AWS Encryption SDK，您可以針對 KMS 金鑰使用任何有效的金鑰識別碼 (金鑰識別碼、金鑰 ARN、別名或別名 ARN)。使用加密時適用於 C 的 AWS Encryption SDK，您只能使用金鑰識別碼或金鑰 ARN。

  如果您在加密時為 KMS 金鑰指定別名或別名 ARN，則會儲AWS Encryption SDK存目前與該別名關聯的金鑰 ARN，而不會儲存別名。變更別名不會影響用於解密資料金鑰的 KMS 金鑰。
- 在嚴謹模式 (指定特定包裝金鑰) 中解密時，您必須使用金鑰 ARN 來識別。AWS KMS keys此要求適用於 AWS Encryption SDK 的所有語言實作。

  當您使用金AWS KMS鑰圈加密時，會將金鑰 ARN AWS Encryption SDK 儲存在加密資料金鑰的 AWS KMS key中繼資料中。在嚴謹模式下解密時，會先AWS Encryption SDK確認金鑰圈 (或主要金鑰提供者) 中是否出現相同的金鑰 ARN，然後再嘗試使用包裝金鑰來解密加密的資料金鑰。如果您使用不同的金鑰識別碼，即使識別碼參照相同的金鑰AWS KMS key，也無法辨識或使用。AWS Encryption SDK

若要將原始 AES 金鑰或原始 RSA key pair 指定為金鑰環中的環繞金鑰，您必須指定命名空間和名稱。在主要金鑰提供者中`Provider ID`，相當於命名空間，`Key ID`且相等於名稱。解密時，您必須為每個原始包裝金鑰使用與加密時使用完全相同的命名空間和名稱。如果您使用不同的命名空間或名稱，則即使密鑰材料相同，也不AWS Encryption SDK會識別或使用包裝鍵。

使用多地區 AWS KMS keys

您可以在中使用 AWS Key Management Service (AWS KMS) 多區域鍵作為包裝鍵。AWS Encryption SDK如果您使用一個多區域金鑰加密AWS 區域，則可以使用不同的相關多區域金鑰來解密。AWS 區域2.3 版中引入了對多區域鍵的 Support。 和 3.0 版AWS Encryption SDK本的 x。 AWS加密 CLI 的 x。

AWS KMS多區域金鑰是具有相同金鑰材料和金鑰 ID 的AWS KMS keys一組不AWS 區域同金鑰。您可以使用這些相關鍵字，就好像它們在不同區域中是相同的索引鍵一樣。多區域金鑰支援常見的災難復原和備份案例，這些案例需要在一個區域中進行加密，並在不同區域進行解密，而不需要跨區域呼叫。AWS KMS如需有關多區域金鑰的詳細資訊，請參閱AWS Key Management Service開發人員指南中的使用多區域金鑰。

為了支援多區域金鑰，AWS Encryption SDK包括AWS KMS多區域感知金鑰環和主金鑰提供者。每種程式語言中的全新多區域感知符號同時支援單一區域和多區域金鑰。

- 對於單一區域金鑰，多區域感知符號的行為與單一區域金鑰圈和主金鑰提供者一AWS KMS樣。它只會嘗試使用加密資料的單一區域金鑰來解密密文。
- 對於多區域金鑰，多區域感知符號會嘗試使用加密資料的相同多區域金鑰，或使用您指定之區域中的相關多區域金鑰來解密密文。

在採用多個 KMS 金鑰的多區域感知金鑰環和主金鑰提供者中，您可以指定多個單一區域和多區域金鑰。不過，您只能從每組相關的多區域金鑰中指定一個金鑰。如果您使用相同的金鑰 ID 指定多個金鑰識別碼，建構函式呼叫會失敗。

您也可以將多區域金鑰與標準、單一區域金鑰AWS KMS圈和主要金鑰提供者搭配使用。不過，您必須在相同區域中使用相同的多區域金鑰來加密和解密。單一區域金鑰圈和主金鑰提供者只會嘗試使用加密資料的金鑰來解密密文。

下列範例顯示如何使用多區域金鑰以及新的多區域感知金鑰環和主金鑰提供者來加密和解密資料。這些範例會加密「`us-east-1`區域」中的資料，並使用每個區域中的相關多`us-west-2`區域金鑰來解密「區域」中的資料。在執行這些範例之前，請將範例多地區索引鍵 ARN 取代為您的. AWS 帳戶

C

若要使用多區域金鑰加密，請使用方`Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()`法來實體化金鑰環。指定多區域金鑰。

這個簡單的範例不包含加密內容。如需在 C 中使用加密內容的範例，請參閱加密和解密字串。

如需完整範例，請參閱的適用於 C 的 AWS Encryption SDK存放庫中的 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

若要使用美國東部 (維吉尼亞北部) (us-east-1) 區域中的多區域金鑰加密，請使用多區域金鑰的金鑰識別碼和指定區域的用戶端來建立`CreateAwsKmsMrkKeyringInput`物件。AWS KMS然後使用該`CreateAwsKmsMrkKeyring()`方法來創建鑰匙圈。

此方`CreateAwsKmsMrkKeyring()`法會建立只有一個多區域金鑰的金鑰圈。若要使用多個包裝金鑰 (包括多區域金鑰) 加密，請使用方`CreateAwsKmsMrkMultiKeyring()`法。

如需完整範例，請參閱中的 AwsKmsMrkKeyringExample.cs (AWS Encryption SDK適用於 .NET 存放 GitHub庫)。

```
//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

此範例會加密 us-east-1 區域中多區域金鑰下的`hello.txt`檔案。由於此範例會指定含 Region 項目的索引鍵 ARN，因此此範例不會使用該`--wrapping-keys`參數的 region 屬性。

當環繞金鑰的金鑰 ID 未指定 Region 時，您可以使用的區域屬性`--wrapping-keys`來指定區域，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

若要使用多區域金鑰加密，請具現化AwsKmsMrkAwareMasterKeyProvider並指定多區域金鑰。

如需完整範例，請參閱BasicMultiRegionKeyEncryptionExample.java的適用於 JAVA 的 AWS Encryption SDK存放庫中的 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

若要使用多區域金鑰加密，請使用該buildAwsKmsMrkAwareStrictMultiKeyringBrowser()方法建立金鑰環並指定多區域金鑰。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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 ### JavaScript # AWS Encryption SDK 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

若要使用多區域金鑰加密，請使用該`buildAwsKmsMrkAwareStrictMultiKeyringNode()`方法建立金鑰環並指定多區域金鑰。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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

若要使用AWS KMS多區域金鑰加密，請使用方`MRKAwareStrictAwsKmsMasterKeyProvider()`法並指定多區域金鑰。

如需完整範例，請參閱的適用於 Python 的 AWS Encryption SDK存放庫中的 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
)
```

接下來，將您的密文移到「us-west-2地區」。您不需要重新加密密文本。

若要在 [區域] 中以嚴格模式解密密文，請使用 [us-west-2區域] 中相關多區域金鑰的金鑰 ARN 來實體化多區域感知符號。us-west-2如果您在不同的區域 (包括加密的地區) 中指定相關多區域金鑰的金鑰 ARNus-east-1，則多區域感知符號會針對此進行跨區域呼叫。AWS KMS key

在嚴謹模式下解密時，多區域感知符號需要金鑰 ARN。它只接受來自每組相關多區域鍵的一個關鍵 ARN。

在執行這些範例之前，請將範例多地區索引鍵 ARN 取代為您的. AWS 帳戶

C

若要使用多區域金鑰在嚴謹模式中解密，請使用方Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()法來實體化金鑰環。在本機 (US-西 -2) 區域中指定相關的多區域金鑰。

如需完整範例，請參閱的適用於 C 的 AWS Encryption SDK存放庫中的 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

若要在嚴謹模式中使用單一多區域金鑰進行解密，請使用您用來組合輸入並建立金鑰環以進行加密的相同建構函式和方法。使用相關多區域金鑰的金鑰 ARN 和美國西部 (奧勒岡) (US-西部 -2) 區域的AWS KMS用戶端來實例化`CreateAwsKmsMrkKeyringInput`物件。然後使用該`CreateAwsKmsMrkKeyring()`方法建立具有一個多區域 KMS 金鑰的多區域金鑰圈。

如需完整範例，請參閱中的 AwsKmsMrkKeyringExample.cs (AWS Encryption SDK適用於 .NET 存放 GitHub庫)。

```
// 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

若要使用 us-west-2 區域中的相關多區域金鑰進行解密，請使用`--wrapping-keys`參數的金鑰屬性來指定其金鑰 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

若要以嚴格模式解密，請在本機 (us-west-2) 區域中實例化AwsKmsMrkAwareMasterKeyProvider並指定相關的多區域金鑰。

如需完整範例，請參閱的適用於 JAVA 的 AWS Encryption SDK存放庫中的 BasicMultiRegionKeyEncryptionExample.java。 GitHub

```
// 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

若要以嚴格模式解密，請使用buildAwsKmsMrkAwareStrictMultiKeyringBrowser()方法建立金鑰環，並在本機 (us-west-2) 區域中指定相關的多區域金鑰。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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 ### JavaScript # AWS Encryption SDK 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

若要以嚴格模式解密，請使用`buildAwsKmsMrkAwareStrictMultiKeyringNode()`方法建立金鑰環，並在本機 (us-west-2) 區域中指定相關的多區域金鑰。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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-east-1
 */
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

若要在嚴謹模式中解密，請使用此MRKAwareStrictAwsKmsMasterKeyProvider()方法建立主要金鑰提供者。在本機 (US-西 -2) 區域中指定相關的多區域金鑰。

如需完整範例，請參閱的適用於 Python 的 AWS Encryption SDK存放庫中的 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
)
```

您也可以使用AWS KMS多區域金鑰在探索模式下解密。在探索模式下解密時，您不會指定任何AWS KMS keys。(如需有關單一區域AWS KMS探索金鑰圈的資訊，請參閱使用 AWS KMS 探索鑰匙圈。)

如果您使用多區域金鑰加密，則探索模式下的多區域感知符號會嘗試使用本機區域中的相關多區域金鑰來解密。如果不存在，則呼叫失敗。在探索模式中，不AWS Encryption SDK會嘗試跨區域呼叫用於加密的多區域金鑰。

> **Note**
> 如果您在探索模式中使用多區域感知符號來加密資料，則加密作業會失敗。

下列範例顯示如何在探索模式中使用多區域感知符號進行解密。因為您未指定AWS KMS key，因此AWS Encryption SDK必須從不同的來源取得「區域」。如果可能，請明確指定本地區域。否則，AWS Encryption SDK從 AWS SDK 中為您的編程語言配置的區域獲取本地區域。

在執行這些範例之前，請將範例帳戶 ID 和多地區金鑰 ARN 取代為您的. AWS 帳戶

C

若要使用多區域金鑰在探索模式中解密，請使用`Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()`方法建立金鑰環，並使用建置探索篩選器的`Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()`方法。若要指定本機區域，請定義`ClientConfiguration`並在AWS KMS用戶端中指定。

如需完整範例，請參閱的適用於 C 的 AWS Encryption SDK存放庫中的 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

若要在 for .NET 中建立多重區域感知探索金鑰環，請具現化一個物件，該CreateAwsKmsMrkDiscoveryKeyringInput物件需要特定AWS KMS用戶端AWS 區域，以及可將 KMS 金鑰限制在特定分割區和帳戶的選擇性探索篩選器。AWS Encryption SDK AWS然後使用輸入對象調用該CreateAwsKmsMrkDiscoveryKeyring()方法。如需完整範例，請參閱中的 AwsKmsMrkDiscoveryKeyringExample.cs (AWS Encryption SDK適用於 .NET 存放 GitHub 庫)。

若要為多個金鑰環建立多個區域感知探索金鑰圈AWS 區域，請使用該CreateAwsKmsMrkDiscoveryMultiKeyring()方法建立多重金鑰環，或使用建立多個多區域感知探索金鑰環，然後使用該方法CreateAwsKmsMrkDiscoveryKeyring()將它們組合成多重金鑰圈。CreateMultiKeyring()

如需範例，請參閱 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

若要在探索模式下解密，請使用`--wrapping-keys`參數的探索屬性。探索帳戶和探索分割區屬性會建立選用的探索篩選器，但建議使用。

若要指定「區域」(Region)，此指令會包含`--wrapping-keys`參數的區域屬性。

```
# 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

若要指定本機區域，請使用`builder().withDiscoveryMrkRegion`參數。否則，AWS Encryption SDK會從中設定的區域取得本機區域[AWS SDK for Java](https://aws.amazon.com/sdk-for-java/)。

如需完整範例，請參閱的適用於 JAVA 的 AWS Encryption SDK存放庫中的[DiscoveryMultiRegionDecryptionExample.java。](https://github.com) GitHub

```
// 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

若要在探索模式中使用對稱的多區域金鑰進行解密，請使用此方`AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()`法。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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

若要在探索模式中使用對稱的多區域金鑰進行解密，請使用此方`AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()`法。

如需完整範例，請參閱上的儲存庫中的。 適用於 JavaScript 的 AWS Encryption SDK 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

若要使用多區域金鑰在探索模式下解密，請使用此方`MRKAwareDiscoveryAwsKmsMasterKeyProvider()`法。

如需完整範例，請參閱的適用於 Python 的 AWS Encryption SDK存放庫中的 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
)
```

選擇演算法套件

AWS Encryption SDK支援數種對稱和非對稱加密演算法，可在您指定的包裝金鑰下加密您的資料金鑰。但是，當它使用這些資料金鑰來加密您的資料時，AWS Encryption SDK預設會使用建議的演算法套件，該演算法套件使用 AES-GCM 演算法搭配金鑰衍生、數位簽章和金鑰承諾。雖然預設演算法套件可能適用於大多數應用程式，但您可以選擇替代演算法套件。例如，沒有數位簽章的演算法套件會

滿足某些信任模型。如需AWS Encryption SDK支援之演算法套件的相關資訊，請參閱AWS Encryption SDK中的支援演算法套件。

下列範例說明如何在加密時選取替代演算法套件。這些範例選取建議的 AES-GCM 演算法套件，其中包含金鑰衍生和金鑰承諾，但不含數位簽章。使用不包含數位簽章的演算法套件進行加密時，請在解密時使用僅限未簽章的解密模式。此模式在遇到已簽署的加密文字時失敗，在串流解密時最有用。

C

若要在中指定替代演算法套件適用於 C 的 AWS Encryption SDK，您必須明確建立 CMM。然後搭`aws_cryptosdk_default_cmm_set_alg_id`配 CMM 和選取的演算法套件使用。

```
/* 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;
}
```

解密沒有數位簽章的加密資料時，請使用AWS_CRYPTOSDK_DECRYPT_UNSIGNED。如果遇到簽名的密文，這會導致解密失敗。

```
/* 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

若要在 .NET 中指定替代演算法套件，請指定EncryptInput物件的`AlgorithmSuiteId`屬性。AWS Encryption SDKFor .NET 包含常數，您可以用來識別偏好的演算法套件。AWS Encryption SDK

For .NET 在AWS Encryption SDK流解密時沒有檢測簽名密文的方法，因為此庫不支持流式數據。

```
// 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`檔案時，此範例會使用`--algorithm`參數來指定不含數位簽章的演算法套件。

```
# 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
```

解密時，此範例會使用`--decrypt-unsigned`參數。建議您使用此參數來確保解密未簽署的加密文字，尤其是使用 CLI (永遠是串流輸入和輸出)。

```
# 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

若要指定替代演算法套件，請使用此`AwsCrypto.builder().withEncryptionAlgorithm()`方法。此範例指定不含數位簽章的替代演算法套件。

```
// 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();
```

當串流資料進行解密時，請使用此`createUnsignedMessageDecryptingStream()`方法來確保您要解密的所有密文都是未簽署的。

```
// 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

若要指定替代演算法套件，請使用具有`AlgorithmSuiteIdentifier`枚舉值的`suiteId`參數。

```
// 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, })
```

解密時，請使用標準`decrypt`方法。 適用於 JavaScript 的 AWS Encryption SDK在瀏覽器中沒有`decrypt-unsigned`模式，因為瀏覽器不支持流媒體。

```
// 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

若要指定替代演算法套件，請使用具有`AlgorithmSuiteIdentifier`枚舉值的`suiteId`參數。

```
// 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, })
```

解密沒有數位簽章的加密資料時，請使用「 decryptUnsignedMessage串流」。如果遇到簽名密文此方法失敗。

```
// 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

若要指定替代加密演算法，請使用具有Algorithm枚舉值的algorithm參數。

```
# 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
)
```

解密沒有數位簽章的加密郵件時，請使用decrypt-unsigned串流模式，尤其是在串流時進行解密時。

```
# 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
```

限制加密的資料金鑰

您可以限制加密訊息中加密資料金鑰的數量。此最佳作法功能可協助您在加密時偵測設定錯誤的金鑰環，或在解密時偵測惡意密文。它還可以防止對您的關鍵基礎結構進行不必要、昂貴且可能詳盡的呼叫。當您解密來自不受信任來源的郵件時，限制加密的資料金鑰是最有價值的。

雖然大多數加密郵件對於加密中使用的每個包裝金鑰都有一個加密資料金鑰，但加密訊息最多可包含 65,535 個加密資料金鑰。惡意執行者可能會建構含有數千個加密資料金鑰的加密訊息，而這些金鑰都無法解密。因此，AWS Encryption SDK會嘗試解密每個加密的資料金鑰，直到用盡訊息中的加密資料金鑰為止。

若要限制加密的資料金鑰，請使用`MaxEncryptedDataKeys`參數。此參數適用於從 1.9 版開始的所有受支援的程式設計語言。 X 和 2.2. 的 x 的AWS Encryption SDK。它是可選的，在加密和解密時有效。下列範例會解密使用三個不同包裝金鑰加密的資料。該`MaxEncryptedDataKeys`值設定為 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

若要限制 .NET 中的加密資料金鑰，AWS Encryption SDK請為 .NET 實例化用戶端，並將其選用`MaxEncryptedDataKeys`參數設定AWS Encryption SDK為所需的值。然後，在配置的AWS Encryption SDK實例上調用該`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)
```

建立探索篩選器

解密使用 KMS 金鑰加密的資料時，最佳做法是在嚴謹模式下進行解密，也就是說，將使用的包裝金鑰限制為只有您指定的金鑰。不過，如有必要，您也可以在探索模式中解密，而您不需要指定任何包裝金鑰。在此模式中，AWS KMS無論誰擁有或擁有該 KMS 金鑰的存取權，都可以使用加密的 KMS 金鑰來解密加密的資料金鑰。

如果您必須在探索模式中解密，建議您一律使用探索篩選器，這會限制可用於指定AWS 帳戶和分割區中的 KMS 金鑰。探索篩選器是選用的，但這是最佳作法。

使用下表決定探索篩選器的分割區值。

區域	分區
AWS 區域	aws
中國地區	aws-cn
AWS GovCloud (US) Regions	aws-us-gov

本節中的範例說明如何建立探索篩選器。在使用程式碼之前，請先以AWS 帳戶和分割區的有效值取代範例值。

C

如需完整範例，請參閱中的 kms_discovery.cpp 適用於 C 的 AWS Encryption SDK。

```
/* 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

如需完整範例，請參閱 DiscoveryFilterExample.NET 中AWS Encryption SDK的 .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

如需完整範例，請參閱中的 DiscoveryDecryptionExample適用於 JAVA 的 AWS Encryption SDK.java。

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

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

JavaScript (Node and Browser)

如需完整範例，請參閱中的. 適用於 JavaScript 的 AWS Encryption SDK Node.js

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

Python

如需完整範例，請參閱中的 discovery_kms_provider.py 適用於 Python 的 AWS Encryption SDK。

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

設定承諾產品原則

承諾產品原則是一種組態設定，可決定您的應用程式是否使用金鑰承諾進行加密和解密。使用關鍵承諾進行加密和解密是AWS Encryption SDK最佳實務。

設定及調整承諾產品原則是從 1.7 版移轉的重要步驟。 AWS Encryption SDK到 2.0 版本的 x 及更早版本。 x 及更新版本。移轉主題中會詳細說明此進度。

最新版本 (從 2.0 版開始AWS Encryption SDK) 中的預設承諾產品原則值。x),RequireEncryptRequireDecrypt, 是大多數情況下的理想選擇。不過，如果您需要解密在沒有金鑰承諾的情況下加密的加密文字，您可能需要將承諾原則變更為。RequireEncryptAllowDecrypt如需如何在每種程式設計語言中設定承諾產品原則的範例，請參閱設定承諾產品原則。

使用串流資料

當您串流資料進行解密時，請注意在完整性檢查完成後，但在驗證數位簽章之前，AWS Encryption SDK傳回解密的純文字。為了確保您在簽章驗證之前不會傳回或使用純文字，建議您緩衝串流的純文字，直到整個解密程序完成為止。

只有當您要串流加密文字進行解密，並且只有在使用包含數位簽章的演算法套件 (例如預設演算法套件) 時，才會出現此問題。

為了使緩衝更容易，某些AWS Encryption SDK語言實現（例如適用於 JavaScript 的 AWS Encryption SDK在 Node.js 中）包含緩衝功能作為解密方法的一部分。一律串流輸入和輸出的AWS加密 CLI 在 1.9 版中引入了一個`--buffer`參數。 X 和 2.2. x. 在其他語言實作中，您可以使用現有的緩衝功能。AWS Encryption SDK對於 .NET 不支持流式傳輸。）

如果您使用的演算法套件不含數位簽章，請務必在每種語言實作中使用`decrypt-unsigned`此功能。此功能會解密密文，但如果遇到簽署的密文，就會失敗。如需詳細資訊，請參閱 選擇演算法套件。

快取資料金鑰

一般而言，不鼓勵重複使用資料金鑰，但提供資料金鑰快取選項，可提供有限的資料金鑰重複使用。AWS Encryption SDK資料金鑰快取可以改善某些應用程式的效能，並減少對金鑰基礎結構的呼叫。在生產環境中使用資料金鑰快取之前，請先調整安全性閾值，並進行測試，以確保其優點超過重複使用資料金鑰的缺點。

使用 keyring

.NET 的 適用於 C 的 AWS Encryption SDK 適用於 JavaScript 的 AWS Encryption SDK、 適用於 JAVA 的 AWS Encryption SDK、和使用金鑰環來執行信封加密。 AWS Encryption SDK Keyring 會產生、加密及解密資料金鑰。金鑰圈會決定保護每個訊息的唯一資料金鑰來源，以及加密該資料金鑰的包裝金鑰。您可以在加密時指定 keyring，並在解密時指定相同或不同的 keyring。您可以使用 SDK 提供的 keyring，或編寫您自己的相容自訂 keyring。

您可以個別使用每個 keyring 或是結合 keyring 成為多重 keyring。雖然多數 keyring 可以產生、加密及解密資料金鑰，您可能想要建立僅執行一個特定操作的 keyring，例如只會產生資料金鑰的 keyring，並將該 keyring 與其他 keyring 結合使用。

我們建議您使用可保護包裝金鑰的金鑰圈，並在安全邊界內執行密碼編譯作業，例如使用永不離開 AWS Key Management Service()AWS KMS未加密 AWS KMS keys 的金 AWS KMS 鑰圈。您也可以撰寫使用儲存在硬體安全性模組 (HSM) 中或受其他主要金鑰服務保護的包裝金鑰環。如需詳細資訊，請參閱AWS Encryption SDK 規格中的金鑰圈介面主題。

金鑰圈在 適用於 JAVA 的 AWS Encryption SDK、 適用於 Python 的 AWS Encryption SDK和 AWS 加密 CLI 中扮演主要金鑰和主要金鑰提供者的角色。如果您使用的不同語言實作 AWS Encryption SDK 來加密和解密資料，請務必使用相容的金鑰環和主金鑰提供者。如需詳細資訊，請參閱 Keyring 相容性。

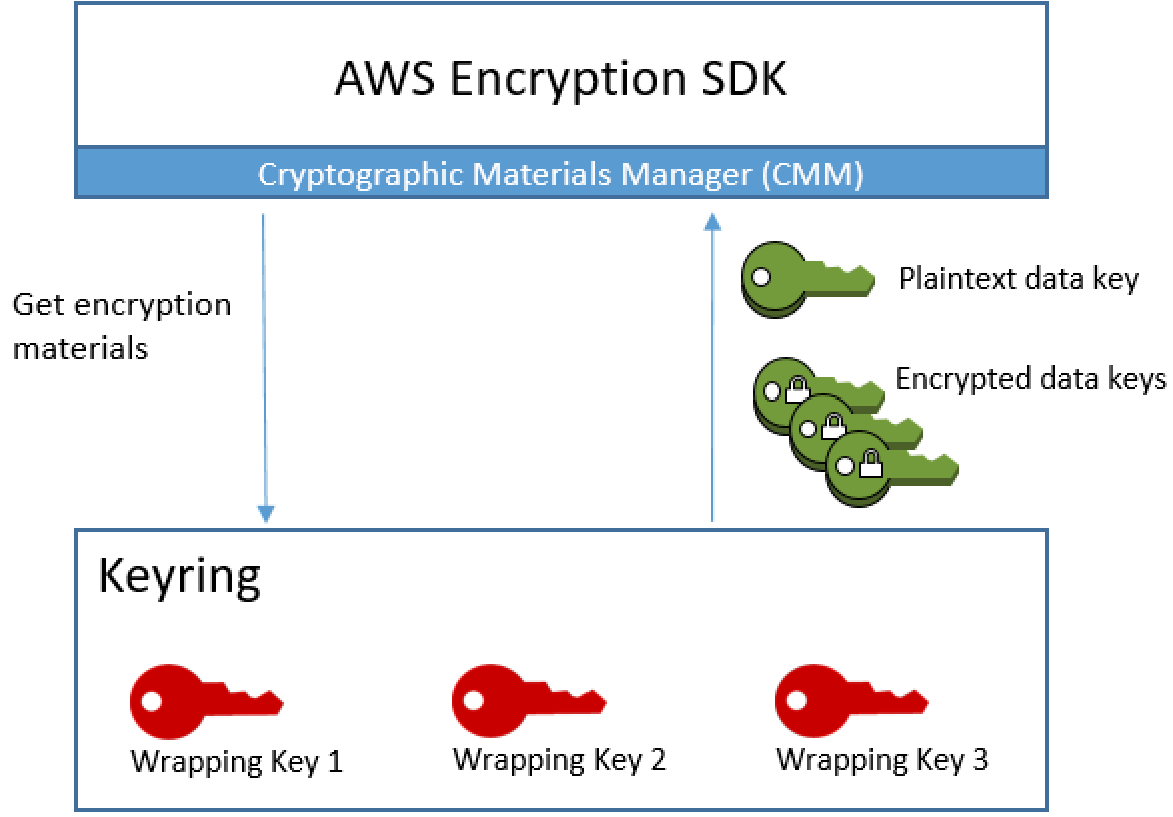
本主題說明如何使用的金鑰圈功能，以 AWS Encryption SDK 及如何選擇金鑰圈。如需建立和使用金鑰環的範例，請參閱 C 和JavaScript主題。

主題

- keyring 如何運作
- Keyring 相容性
- 選擇鑰匙圈

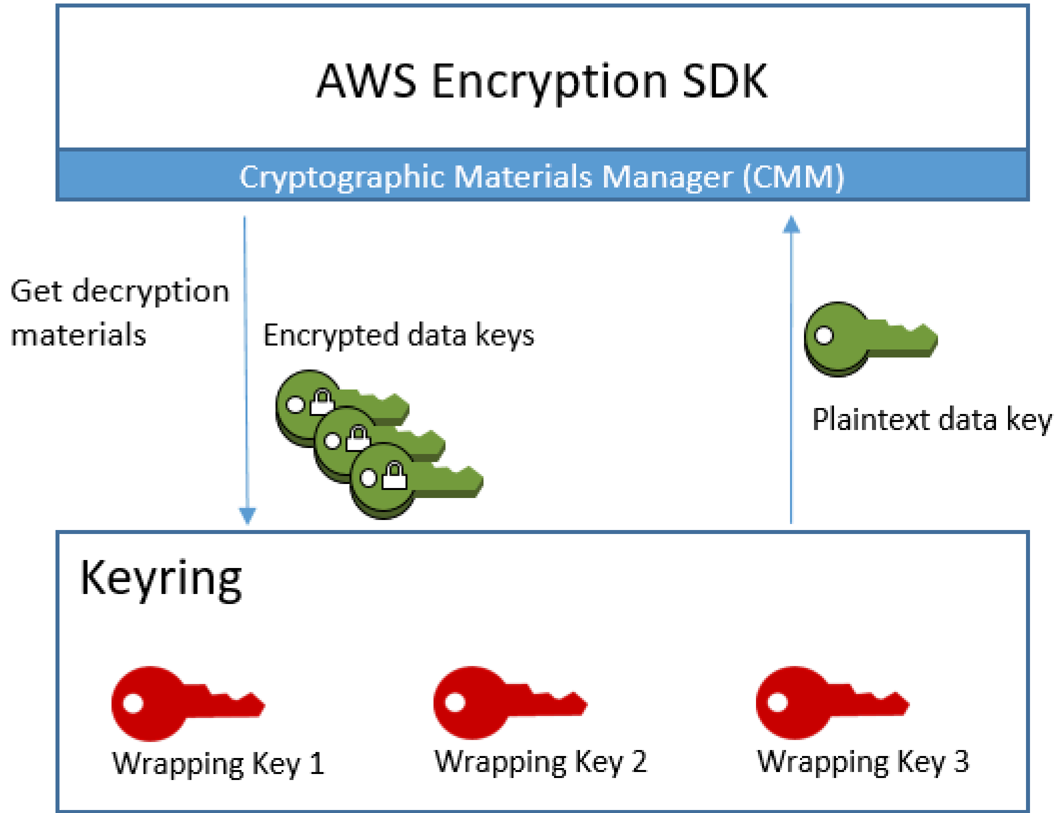
keyring 如何運作

當您加密資料時， AWS Encryption SDK 會詢問金鑰環是否有加密材料。金鑰圈會傳回純文字資料金鑰，以及金鑰圈中每個包裝金鑰所加密的資料金鑰副本。會 AWS Encryption SDK 使用純文字金鑰來加密資料，然後銷毀純文字資料金鑰。然後， AWS Encryption SDK 返回一個加密的消息，其中包括加密的數據密鑰和加密的數據。



解密資料時，您可以使用用來加密資料的相同金鑰環，或使用不同的金鑰環。若要解密資料，解密金鑰環必須在加密金鑰環中包含 (或具有存取權) 至少一個包裝金鑰。

會將加密的資料金鑰從加密的訊息 AWS Encryption SDK 傳遞至金鑰圈，並要求金鑰圈解密其中的任何一個。keyring 使用其包裝金鑰來解密其中一個加密的資料金鑰，並傳回純文字資料金鑰。 AWS Encryption SDK 使用純文字金鑰來解密資料。如果 keyring 中沒有任何包裝金鑰可以解密任何加密的資料金鑰，則解密操作會失敗。



您可以使用單一 keyring，也可以將相同類型或不同類型的 keyring 結合成多重 keyring。當您加密資料時，多重 keyring 會傳回由所有包裝金鑰 (在構成多重 keyring 的所有 keyring 中) 所加密的資料金鑰的副本。您可以使用包含多重金鑰圈中任何一個包裝金鑰的金鑰圈來解密資料。

Keyring 相容性

雖然不同的語言實作 AWS Encryption SDK 有一些架構上的差異，但它們完全相容，但會受到語言限制的限制。您可以使用一種語言實現來加密數據，並使用任何其他語言實現對其進行解密。但是，您必須使用相同或對應的包裝金鑰來加密和解密資料金鑰。如需有關語言條件約束的資訊，請參閱有關每種語言實作的主題，例如 適用於 JavaScript 的 AWS Encryption SDK 主題the section called “相容性”中的。

加密金鑰圈的不同需求

在除了以外的 AWS Encryption SDK 語言實作中 適用於 C 的 AWS Encryption SDK，加密金鑰環 (或多重金鑰環) 或主要金鑰提供者中的所有包裝金鑰都必須能夠加密資料金鑰。如果有任何包裝金鑰無法

加密，則加密方法會失敗。因此，呼叫者必須擁有金鑰環中所有金鑰的必要權限。如果您單獨使用探索金鑰環加密資料，或在多重金鑰環中加密資料，則加密作業會失敗。

例外情況是 適用於 C 的 AWS Encryption SDK，加密作業會忽略標準探索金鑰環，但如果您單獨或在多重金鑰環中指定多區域探索金鑰環，則會失敗。

相容 Keyring 和主金鑰提供者

下表顯示哪些主要金鑰和主要金鑰提供者與提 AWS Encryption SDK 供的金鑰圈相容。任何由於語言限制而導致的輕微不相容，將在語言實作的相關主題中說明。

鑰匙圈:	主要金鑰提供者：
AWS KMS 鑰匙圈	爪哇 MasterKey 爪哇 MasterKeyProvider Python MasterKey Python MasterKeyProvider **Note** 適用於 Python 的 AWS Encryption SDK 和 適用於 JAVA 的 AWS Encryption SDK 不包含相當於AWS KMS 地區探索金鑰圈的主金鑰或主要金鑰提供者。
AWS KMS 分層鑰匙圈	僅適用於版本 4。 AWS Encryption SDK 適用於 .NET 和版本 3 的 x。的 x 的 適用於 JAVA 的 AWS Encryption SDK。
AWS KMS 鑰匙扣	僅適用於版本 3。 的 x 的 適用於 JAVA 的 AWS Encryption SDK。
原始 AES keyring	搭配對稱加密金鑰使用時： JceMasterKey(爪哇) RawMasterKey(Python)
原始 RSA keyring	搭配非對稱加密金鑰使用時： JceMasterKey(爪哇)

鑰匙圈:	主要金鑰提供者：
	RawMasterKey(Python) ⓘ Note 原始 RSA 金鑰圈不支援非對稱 KMS 金鑰。如果您想要使用非對稱的 RSA KMS 金鑰，請參閱第 4 版。 .NET 的 AWS Encryption SDK x 支援使用對稱加密 (SYMMETRIC_DEFAULT) 或非對稱 RSA 的 AWS KMS 金鑰圈。 AWS KMS keys
原始 ECDH 鑰匙圈	僅適用於版本 3。 的 x 的 適用於 JAVA 的 AWS Encryption SDK。

選擇鑰匙圈

您的金鑰圈會決定保護資料金鑰的包裝金鑰，以及最終保護資料的金鑰。使用對您的任務實用的最安全的包裝密鑰。盡可能使用包裝受硬體安全性模組或金鑰管理基礎結構保護的金鑰，例如 AWS Key Management Service(AWS KMS) 中的 KMS 金鑰或加密金鑰AWS CloudHSM。

提 AWS Encryption SDK 供多種編程語言的鑰匙圈和鑰匙圈配置，您可以創建自己的自定義鑰匙圈。您也可以建立包含一或多個相同或不同類型之鑰匙圈的多重金鑰圈。

主題

- AWS KMS 鑰匙圈
- AWS KMS 階層式鑰匙圈
- AWS KMS ECDH 鑰匙圈
- 原始 AES keyring
- 原始 RSA keyring
- 原始 ECDH 鑰匙圈
- 多重 keyring

AWS KMS 鑰匙圈

AWS KMS 金鑰圈使用對稱加密AWS KMS keys來產生、加密和解密資料金鑰。 AWS Key Management Service (AWS KMS) 保護您的 KMS 金鑰，並在 FIPS 界限內執行密碼編譯作業。我們建議您盡可能使用 AWS KMS 金鑰圈或具有類似安全性屬性的鑰匙圈。

您可以在 2.3 版開始的金鑰 AWS KMS 圈或主金鑰提供者中使用 AWS KMS 多區域金鑰。 和 3.0 版 AWS Encryption SDK 本的 x。 AWS 加密 CLI 的 x。如需使用新的多區域感知符號的詳細資訊和範例，請參閱。使用多地區 AWS KMS keys如需有關多區域金鑰的詳細資訊，請參閱AWS Key Management Service 開發人員指南中的使用多區域金鑰。

> Note
>
> 版本 4. AWS Encryption SDK 適用於 .NET 和版本 3 的 x。 x 適用於 JAVA 的 AWS Encryption SDK 是唯一支援使用非對稱 R AWS KMS keys SA 之 AWS KMS 金鑰環的程式設計語言實作。
> 如果您嘗試在任何其他語言實作的加密金鑰環中包含非對稱 KMS 金鑰，則加密呼叫會失敗。如果您將其包含在解密金鑰環中，則會忽略該金鑰環。
> KMS 鑰匙圈中的所有提及都是 AWS Encryption SDK 指鑰匙圈。 AWS KMS

AWS KMS 鑰匙圈可以包含兩種類型的環繞鍵：

- 產生器金鑰：產生純文字資料金鑰並加密。加密資料的金鑰環必須有一個產生器金鑰。
- 其他金鑰：加密產生器金鑰所產生的純文字資料金鑰。 AWS KMS 鑰匙圈可以有零個或多個其他金鑰。

加密時，您使用的 AWS KMS 金鑰環必須有產生器金鑰。解密時，生成器密鑰是可選的，並忽略生成器密鑰和其他密鑰之間的區別。

如果加 AWS KMS 密金鑰圈只有一個 AWS KMS 金鑰，則會使用該金鑰來產生和加密資料金鑰。

與所有鑰匙圈一樣， AWS KMS 鑰匙圈可以單獨使用，也可以與其他相同或不同類型的鑰匙圈一起使用。

主題

- AWS KMS 金鑰圈所需的權限
- 在 AWS KMS 鑰匙圈 AWS KMS keys 中識別

- 建立用於加密的 AWS KMS 金鑰圈
- 建立用於解密的 AWS KMS 金鑰圈
- 使用 AWS KMS 探索鑰匙圈
- 使用 AWS KMS 區域探索金鑰圈

AWS KMS 金鑰圈所需的權限

AWS Encryption SDK 不需要一個 AWS 帳戶 ，它不依賴於任何 AWS 服務. 但是，若要使用 AWS KMS 金鑰圈，您需要在 AWS 帳戶 金鑰圈 AWS KMS keys 中具有以下最低權限。

- 要使用密 AWS KMS 鑰環進行加密，您需要 kms：對生成器密鑰的GenerateDataKey權限。您需要 KMS: 加密金鑰環中所有其他金鑰的權限。 AWS KMS
- 若要使用金 AWS KMS 鑰圈進行解密，您需要 KMS: 在金鑰環中至少有一個金鑰的 Decryp t 權限。 AWS KMS
- 要使用由密鑰環組成的多密 AWS KMS 鑰環進行加密，您需要 kms：對生成器密鑰環中的生成器密鑰的GenerateDataKey權限。您需要 KMS：對所有其他金鑰環中所有其他金鑰的加密權限。 AWS KMS

如需有關權限的詳細資訊 AWS KMS keys，請參閱AWS Key Management Service 開發人員指南中的驗證和存取控制。

在 AWS KMS 鑰匙圈 AWS KMS keys 中識別

一個 AWS KMS 鑰匙圈可以包括一個或多 AWS KMS keys個。若要在金 AWS KMS 鑰環 AWS KMS key 中指定，請使用支援的 AWS KMS 金鑰識別碼。您可以用來識別金鑰圈 AWS KMS key 中的金鑰識別碼會隨作業和語言實作而有所不同。有關的密鑰標識符的詳細信息 AWS KMS key，請參閱AWS Key Management Service 開發人員指南中的密鑰標識符。

最佳做法是使用適合您工作的最具體的金鑰識別碼。

- 在的加密金鑰圈中 適用於 C 的 AWS Encryption SDK，您可以使用金鑰 ARN 或別名 ARN 來識別 KMS 金鑰。在所有其他語言實作中，您可以使用金鑰 ID、金鑰 ARN、別名或別名 ARN 來加密資料。
- 在解密 keyring 中，您必須使用金鑰 ARN 來識別 AWS KMS keys。此要求適用於 AWS Encryption SDK的所有語言實作。如需詳細資訊，請參閱 選擇包裝鍵。
- 在用於加密和解密的 keyring 中，您必須使用金鑰 ARN 來識別 AWS KMS keys。此要求適用於 AWS Encryption SDK的所有語言實作。

如果您在加密金鑰圈中為 KMS 金鑰指定別名或別名 ARN，則加密作業會將目前與別名關聯的金鑰 ARN 儲存在加密資料金鑰的中繼資料中。它不會儲存別名。變更別名不會影響用於解密加密資料金鑰的 KMS 金鑰。

建立用於加密的 AWS KMS 金鑰圈

您可以在相同或不同的 AWS 帳戶 和 AWS 區域中使用單個 AWS KMS key 或多 AWS KMS keys 個配置每個 AWS KMS 鑰匙圈。 AWS KMS keys 必須是對稱加密金鑰 (對稱加密金鑰)。您也可以使用對稱加密多區域 KMS 金鑰。與所有鑰匙圈一樣，您可以在多個鑰匙圈中使用一個或多個 AWS KMS 鑰匙圈。

當您建立金 AWS KMS 鑰環來加密資料時，您必須指定產生器金鑰， AWS KMS key 這是用來產生純文字資料金鑰並加密的產生器金鑰。資料金鑰在數學上與 KMS 金鑰無關。然後，如果您選擇，您可以指定其他加密 AWS KMS keys 相同的純文字資料金鑰。

若要解密受此金鑰圈保護的加密訊息，您使用的金鑰圈必須至少包含金鑰圈中 AWS KMS keys 定義的其中一個，或者否。 AWS KMS keys（沒有的 AWS KMS 鑰匙圈稱 AWS KMS keys 為AWS KMS 探索鑰匙圈。）

在除了以外的 AWS Encryption SDK 語言實作中 適用於 C 的 AWS Encryption SDK，加密金鑰環或多重金鑰環中的所有包裝金鑰都必須能夠加密資料金鑰。如果有任何包裝金鑰無法加密，則加密方法會失敗。因此，呼叫者必須擁有金鑰環中所有金鑰的必要權限。如果您單獨使用探索金鑰環加密資料，或在多重金鑰環中加密資料，則加密作業會失敗。例外情況是 適用於 C 的 AWS Encryption SDK，加密作業會忽略標準探索金鑰環，但如果您單獨或在多重金鑰環中指定多區域探索金鑰環，則會失敗。

以下範例會建立包含一個產生器金鑰和一個其他金鑰的金鑰 AWS KMS 環。這些範例使用金鑰 ARN 來識別 KMS 金鑰。這是用於加密的 AWS KMS 金鑰圈的最佳作法，也是用於解密的 AWS KMS 金鑰圈要求。如需詳細資訊，請參閱 在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

C

若要識別 AWS KMS key 中的加密金鑰環 適用於 C 的 AWS Encryption SDK，請指定金鑰 ARN 或別名 ARN。在解密 Keyring 中，您必須使用金鑰 ARN。如需詳細資訊，請參閱 在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

如需完整範例，請參閱 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

若要在 .NET 中建立含有一或多個 AWS KMS 金鑰的金鑰 AWS KMS 圈， AWS Encryption SDK 請建立多重金鑰圈。.NET 包括一個僅 AWS Encryption SDK 用於密鑰的多鑰匙圈。 AWS KMS

當您在中 AWS KMS key 為 .NET 指定加密金鑰環時，您可以使用任何有效的金鑰識別碼：金鑰識別碼、金鑰 ARN、別名或別名 ARN。 AWS Encryption SDK 如需識別 AWS KMS 金鑰圈 AWS KMS keys 中的說明，請參閱在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

下列範例使用版本 4。 用 AWS Encryption SDK 於 .NET 的 x 來建立含有產生器金 AWS KMS 鑰和其他金鑰的金鑰圈。如需完整範例，請參閱 AwsKmsMultiKeyringExample.cs。

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

string generatorKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<string> additionalKey = new List<string> { "alias/exampleAlias" };

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

var kmsEncryptKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(kmsEncryptKeyringInput);
```

JavaScript Browser

當您在中指定加 AWS KMS key 密金鑰環時 適用於 JavaScript 的 AWS Encryption SDK，您可以使用任何有效的金鑰識別碼：金鑰識別碼、金鑰 ARN、別名或別名 ARN。如需識別 AWS KMS 金鑰圈 AWS KMS keys 中的說明，請參閱在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

如需完整範例，請參閱中的儲存庫中的 kms_simple.ts。 適用於 JavaScript 的 AWS Encryption SDK GitHub

```
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

當您在中指定加 AWS KMS key 密金鑰環時 適用於 JavaScript 的 AWS Encryption SDK，您可以使用任何有效的金鑰識別碼：金鑰識別碼、金鑰 ARN、別名或別名 ARN。如需識別 AWS KMS 金鑰圈 AWS KMS keys 中的說明，請參閱在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

如需完整範例，請參閱中的儲存庫中的 kms_simple.ts。 適用於 JavaScript 的 AWS Encryption SDK GitHub

```
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

若要在中建立含有一或多個 AWS KMS 金鑰的鑰匙 AWS KMS 圈 適用於 JAVA 的 AWS Encryption SDK，請建立多重鑰匙圈。 適用於 JAVA 的 AWS Encryption SDK 包括一個僅用 AWS KMS 於鑰匙的多鑰匙圈。

當您在中指定加 AWS KMS key 密金鑰環時 適用於 JAVA 的 AWS Encryption SDK，您可以使用任何有效的金鑰識別碼：金鑰識別碼、金鑰 ARN、別名或別名 ARN。如需識別 AWS KMS 金鑰圈 AWS KMS keys 中的說明，請參閱在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

如需完整範例，請參閱中的 適用於 JAVA 的 AWS Encryption SDK 存放庫中 GitHub的 BasicEncryptionKeyringExample.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("alias/exampleAlias");

// Create the AWS KMS keyring
final CreateAwsKmsMultiKeyringInput keyringInput =
 CreateAwsKmsMultiKeyringInput.builder()
        .generator(generatorKey)
        .kmsKeyIds(additionalKey)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);
```

建立用於解密的 AWS KMS 金鑰圈

您也可以在解密傳回的加密郵件時指定 AWS KMS 金鑰圈。 AWS Encryption SDK 如果解密金鑰圈指定 AWS KMS keys， AWS Encryption SDK 將僅使用這些包裝金鑰來解密加密訊息中的加密資料金鑰。(您也可以使用未指定任何的AWS KMS 探索金鑰環) AWS KMS keys。

解密時，會在 AWS KMS 金鑰圈中 AWS Encryption SDK 搜尋可以解密其中一個加密資料金鑰的金鑰。 AWS KMS key 具體而言，加密郵件中的每個加密資料金鑰 AWS Encryption SDK 會使用下列模式。

- AWS Encryption SDK 會從加密郵件的中繼資料中取 AWS KMS key 得加密資料金鑰的金鑰 ARN。
- 會在解密金鑰環中 AWS Encryption SDK 搜尋 AWS KMS key 具有相符金鑰 ARN 的。
- 如果在金鑰圈中發現 AWS KMS key 具有相符金鑰 ARN，則 AWS Encryption SDK 會要求使用 KMS 金鑰 AWS KMS 來解密加密的資料金鑰。
- 否則會跳到下一個加密的資料金鑰 (如果有)。

AWS Encryption SDK 永遠不會嘗試解密加密的資料金鑰，除非該加密資料金鑰 AWS KMS key 的金鑰 ARN 包含在解密金鑰圈中。如果解密金鑰圈不包含任何加密任何資料金鑰的 ARN，解密呼叫 AWS KMS keys 就會 AWS Encryption SDK 失敗，而不會呼叫。 AWS KMS

從 1.7 版本開始。 x，解密加密的資料金鑰時，一 AWS Encryption SDK 律會將的金鑰 ARN 傳遞 AWS KMS key 至「 AWS KMS 解密」作業的`KeyId`參數。在解密 AWS KMS key 時識別是 AWS KMS 最佳做法，可確保您使用要使用的包裝金鑰來解密加密的資料金鑰。

當解密金 AWS KMS 鑰圈中至少有一個可以解密加密訊息 AWS KMS key 中的一個加密資料金鑰時，含金鑰圈的解密呼叫會成功。同時，發起人必須具有該 AWS KMS key上的 `kms:Decrypt` 許可。此行為可讓您加密不同帳戶 AWS 區域 和帳戶 AWS KMS keys 中多個下的資料，但提供針對特定帳戶、地區、使用者、群組或角色量身打造的更有限的解密金鑰環。

當您在解密金鑰環 AWS KMS key 中指定時，必須使用其金鑰 ARN。否則，將 AWS KMS key 無法辨識。如需尋找金鑰 ARN 的說明，請參閱AWS Key Management Service 開發人員指南中的尋找金鑰 ID 和 ARN。

> **Note**
>
> 如果您重複使用加密 keyring 進行解密，請確定 keyring 中的 AWS KMS keys 金鑰可依其 ARN 識別。

例如，下列金 AWS KMS 鑰圈只包含加密金鑰圈中使用的其他金鑰。但是，此範例並非依其別名參照其他金鑰`alias/exampleAlias`，而是根據解密呼叫的要求使用其他金鑰的 ARN。

您可以使用此 keyring 來解密已使用產生器金鑰和額外金鑰加密的訊息，前提是您具有使用額外金鑰來解密資料的許可。

C

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

struct aws_cryptosdk_keyring *kms_decrypt_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(additional_key);
```

C# / .NET

由於此解密金鑰環只包含一個 AWS KMS 金鑰，因此範例會將`CreateAwsKmsKeyring()`方法與其`CreateAwsKmsKeyringInput`物件的實體搭配使用。要使用一個 AWS KMS 鑰 AWS KMS 匙

創建鑰匙圈，您可以使用單鍵或多鍵鑰匙圈。如需詳細資訊，請參閱 AWS Encryption SDK為 .NET 加密中的資料。下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 x 來建立用於解密的 AWS KMS 金鑰環。

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

string additionalKey = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321";

// Instantiate a KMS keyring for one AWS KMS key.
var kmsDecryptKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = additionalKey
};

var kmsDecryptKeyring =
 materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);
```

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })
const additionalKey = 'arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321'

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

JavaScript Node.js

```
const additionalKey = 'arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321'

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

Java

由於此解密金鑰環只包含一個 AWS KMS 金鑰，因此範例會將`CreateAwsKmsKeyring()`方法與其`CreateAwsKmsKeyringInput`物件的實體搭配使用。要使用一個 AWS KMS 鑰 AWS KMS 匙創建鑰匙圈，您可以使用單鍵或多鍵鑰匙圈。

```
// 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 additionalKey = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321";

// Create a AwsKmsKeyring
CreateAwsKmsKeyringInput kmsDecryptKeyringInput = CreateAwsKmsKeyringInput.builder()
        .generator(additionalKey)
        .kmsClient(KmsClient.create())
        .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);
```

您還可以使用指定用於解密的生成器密鑰的密鑰 AWS KMS 環，例如以下內容。解密時， AWS Encryption SDK 會忽略產生器金鑰與其他金鑰之間的區別。它可以使用任何指定的 AWS KMS keys 來解密加密的資料金鑰。只有當呼叫者有權使用它 AWS KMS key 來解密資料時，呼叫才 AWS KMS 會成功。

C

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

C# / .NET

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 x。

```
// 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";

// Instantiate a KMS keyring for one AWS KMS key.
var kmsDecryptKeyringInput = new CreateAwsKmsKeyringInput
{
```

```
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = generatorKey
};

var kmsDecryptKeyring =
 materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);
```

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

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

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";

// Create a AwsKmsKeyring
CreateAwsKmsKeyringInput kmsDecryptKeyringInput = CreateAwsKmsKeyringInput.builder()
        .generator(generatorKey)
        .kmsClient(KmsClient.create())
        .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(kmsDecryptKeyringInput);
```

與使用所有指定項目的加密金鑰環不同 AWS KMS keys，您可以使用包含 AWS KMS keys 與加密訊息無關且 AWS KMS keys 呼叫者沒有使用權限的解密金鑰圈來解密加密郵件。如果對 AWS KMS 的解密呼叫失敗，例如當發起人沒有必要的許可時， AWS Encryption SDK 會直接跳到下一個加密的資料金鑰。

使用 AWS KMS 探索鑰匙圈

解密時，最佳做法是指定 AWS Encryption SDK 可以使用的包裝金鑰。若要遵循此最佳作法，請使用 AWS KMS 解密金鑰圈，將 AWS KMS 包裝金鑰限制為您指定的金鑰。不過，您也可以建立AWS KMS 探索金鑰圈，也就是不會指定任何 AWS KMS 環繞金鑰的金鑰圈。

AWS Encryption SDK 提供標準 AWS KMS 探索金鑰圈和 AWS KMS 多區域金鑰的探索鑰匙圈。如需將多區域金鑰搭配使用的資訊 AWS Encryption SDK，請參閱使用多地區 AWS KMS keys。

因為它不會指定任何包裝金鑰，因此探索金鑰環無法加密資料。如果您單獨使用探索金鑰環加密資料，或在多重金鑰環中加密資料，則加密作業會失敗。例外情況是 適用於 C 的 AWS Encryption SDK，加密作業會忽略標準探索金鑰環，但如果您單獨或在多重金鑰環中指定多區域探索金鑰環，則會失敗。

解密時，探索金鑰圈允許 AWS Encryption SDK 要求使用加密的資料金鑰 AWS KMS 來解密任何加密的 AWS KMS key 資料金鑰，無論誰擁有或具有存取權。 AWS KMS key只有當呼叫`kms:Decrypt`者具有. AWS KMS key

> ⚠ Important
>
> 如果您在解密多重金鑰環中包含 AWS KMS 探索金鑰圈，探索金鑰圈會覆寫多重金鑰圈中其他金鑰環指定的所有 KMS 金鑰限制。多重金鑰環的行為與其限制性最小的鑰匙圈一樣。 AWS KMS 探索金鑰圈在單獨使用或多重金鑰環中使用時，對加密沒有影響。

為了方便起見， AWS Encryption SDK 提供 AWS KMS 探索鑰匙圈。不過，基於下列原因，建議您在可能時使用較具限制的 keyring。

- 真實性 — AWS KMS 探索金鑰圈可以使用 AWS KMS key 用來加密加密訊息中的資料金鑰的任何金鑰，只要呼叫者有權使用該金鑰 AWS KMS key 來解密。這可能不是發起人想要使用的 AWS KMS key 。例如，其中一個加密的資料金鑰可能以較不安全的方式加密， AWS KMS key 因此任何人都可以使用。
- 延遲和效能 — AWS KMS 探索金鑰圈的速度可能會比其他金鑰環慢，因為會 AWS Encryption SDK 嘗試解密所有加密的資料金鑰，包括 AWS KMS keys 在其他 AWS 帳戶 和區域中加密的資料金鑰，而且 AWS KMS keys 呼叫者沒有用於解密的權限。

如果您使用探索金鑰圈，建議您使用探索篩選器來限制可用於指定 AWS 帳戶 和磁碟分割中的 KMS 金鑰。1.7 版支援探索篩選器。 x 及更新版本的 AWS Encryption SDK. 如需尋找帳戶 ID 和分割區的協助，請參閱中的您的 AWS 帳戶 識別碼和 ARN 格式AWS 一般參考。

下列程式碼會使用 AWS KMS 探索篩選器來實體化探索金鑰圈，以限制 AWS Encryption SDK 可用於aws磁碟分割中的 KMS 金鑰，以及 111122223333 範例帳戶。

在使用此代碼之前，請將示例 AWS 帳戶 和分區值替換為您的 AWS 帳戶 和分區的有效值。如果您的 KMS 金鑰位於中國區域，請使用aws-cn分割區值。如果您的 KMS 金鑰位於中 AWS GovCloud (US) Regions，請使用aws-us-gov分割區值。對於所有其他項目 AWS 區域，請使用aws分割區值。

C

如需完整範例，請參閱：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

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 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 =
 materialProviders.CreateAwsKmsDiscoveryKeyring(kmsDiscoveryKeyringInput);
```

JavaScript Browser

在中 JavaScript，您必須明確指定探索屬性。

```
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，您必須明確指定探索屬性。

```
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);
```

使用 AWS KMS 區域探索金鑰圈

AWS KMS 區域探索金鑰圈是不會指定 KMS 金鑰 ARN 的金鑰圈。相反地，它允許僅使用 KMS 金鑰 AWS Encryption SDK 進行解密，尤其是 AWS 區域。

使用 AWS KMS 區域探索金鑰圈進行解密時，會將 AWS KMS key 在指定的中加 AWS Encryption SDK 密的任何加密資料金鑰解密。 AWS 區域若要成功，呼叫者必須`kms:Decrypt`擁有至少一個加密資料金鑰 AWS 區域 的指定 AWS KMS keys 中的權限。

與其他探索金鑰環一樣，區域探索金鑰圈對加密沒有影響。它僅在解密加密郵件時才起作用。如果您在用於加密和解密的多重金鑰環中使用區域探索金鑰圈，則只有在解密時才有效。如果您單獨或在多重金鑰環中使用多區域探索金鑰圈來加密資料，則加密作業會失敗。

> **Important**
>
> 如果您在解密多重金鑰圈中包含 AWS KMS 區域探索金鑰圈，則區域探索金[鑰圈會覆寫多重](#)金鑰圈中其他金鑰圈所指定的所有 KMS 金鑰限制。多重金鑰環的行為與其限制性最小的鑰匙圈一樣。 AWS KMS 探索金鑰圈在單獨使用或多重金鑰環中使用時，對加密沒有影響。

適用於 C 的 AWS Encryption SDK 嘗試僅使用指定區域中的 KMS 金鑰進行解密時的區域探索金鑰圈。當您在 適用於 JavaScript 的 AWS Encryption SDK 和 AWS Encryption SDK .NET 中使用探索金鑰圈時，您可以在用 AWS KMS 戶端上設定 [區域]。這些 AWS Encryption SDK 實作不會依區域篩選 KMS 金鑰，但 AWS KMS 會失敗指定區域以外的 KMS 金鑰的解密要求。

如果您使用探索金鑰圈，建議您使用探索篩選器，將用於解密的 KMS 金鑰限制為指定 AWS 帳戶 和分割區中的 KMS 金鑰。1.7 版支援探索篩選器。 x 及更新版本的 AWS Encryption SDK.

例如，下列程式碼會使用探索篩選器建立 AWS KMS 區域探索金鑰圈。此金鑰圈限制在美國西部 (奧勒岡) 區域 (美國西部 -2) 帳戶 111122223333 中的 KMS 金鑰。 AWS Encryption SDK

C

若要檢視此 keyring 和 `create_kms_client` 方法的工作實例，請參閱 [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

.NET 沒有專用的 AWS Encryption SDK 區域探索金鑰圈。但是，您可以使用多種技術來限制解密到特定區域時使用的 KMS 金鑰。

限制探索金鑰圈中區域的最有效方法是使用多區域感知探索金鑰圈，即使您只使用單一區域金鑰加密資料也是如此。當它遇到單一區域金鑰時，多區域感知金鑰圈不會使用任何多區域功能。

`CreateAwsKmsMrkDiscoveryKeyring()`方法傳回的金鑰環會在呼叫 AWS KMS之前，依區域篩選 KMS 金鑰。 AWS KMS 只有當加密的資料金鑰透過`CreateAwsKmsMrkDiscoveryKeyringInput`物件中的`Region`參數所指定的區域中的 KMS 金鑰加密時，才會傳送解密要求。

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 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 =
 materialProviders.CreateAwsKmsMrkDiscoveryKeyring(regionalDiscoveryKeyringInput);
```

您也可以在 AWS KMS 用戶端執行個體 ([AmazonKeyManagementServiceClient](https://docs.aws.amazon.com/sdkfornet/v3/apidocs/)) 中指定區域，將 KMS 金鑰限制為特 AWS 區域 定金鑰。不過，與使用多區域感知探索金鑰環相比，此組態效率較低且成本可能更高。而不是在呼叫之前按區域篩選 KMS 金鑰 AWS KMS，而是 AWS Encryption SDK 針 AWS KMS 對每個加密的資料金鑰進行 .NET 呼叫 (直到解密一個金鑰為止)，並依賴 AWS KMS 將其使用的 KMS 金鑰限制在指定區域。

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 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 =
 materialProviders.CreateAwsKmsDiscoveryKeyring(createRegionalDiscoveryKeyringInput);
```

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'], partition: 'aws' }
```

```
})
```

JavaScript Node.js

若要在工作實例中檢視此 keyring 以及 `limitRegions` 和函數，請參閱 kms_regional_discovery.ts。

```
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);
```

適用於 JavaScript 的 AWS Encryption SDK 也會匯出 Node.js 和瀏覽器的`excludeRegions`函式。此功能會建立省略 AWS KMS keys 特定 AWS KMS 區域的區域探索金鑰環。下列範例會建立一個 AWS KMS 區域探索金鑰圈，該金鑰圈可 AWS KMS keys 在帳戶 111122223333 中使用，但美國東部 (us-east-1) AWS 區域 除外。

適用於 C 的 AWS Encryption SDK 沒有類似的方法，但是您可以建立自訂來實作一個方法。ClientSupplier

此範例顯示 Node.js 的程式碼。

```
const discovery = true
const clientProvider = excludeRegions(['us-east-1'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

AWS KMS 階層式鑰匙圈

> **Important**
>
> 階 AWS KMS 層式鑰匙圈僅由第 4 版支援。 AWS Encryption SDK 適用於 .NET 和版本 3 的 x。 的 x 的 適用於 JAVA 的 AWS Encryption SDK。

使用 AWS KMS 階層式金鑰圈，您可以使用對稱式加密 KMS 金鑰保護加密資料，無需 AWS KMS 每次加密或解密資料時都呼叫。對於需要盡量減少呼叫的應用程式，以及可重複使用某些加密資料而不違反其安全性需求的應用程式來說 AWS KMS，這是一個不錯的選擇。

階層式金鑰圈是一種加密材料快取解決方案，可透過使用保存在 Amazon DynamoDB 表格中的 AWS KMS 受保護分支金鑰，然後在本機快取用於加密和解密作業的分支金鑰材料，以減少 AWS KMS 呼叫次數。DynamoDB 表可做為管理和保護分支金鑰的分支金鑰存放區。它存儲活動分支密鑰和分支密鑰的所有以前版本。活動分支密鑰是最新的分支密鑰版本。分層密鑰環使用唯一的數據密鑰來加密每個消息，並使用從活動分支密鑰派生的唯一包裝密鑰對每個數據密鑰進行加密。分層密鑰環取決於活動分支密鑰及其派生包裝密鑰之間建立的層次結構。

分層密鑰環通常使用每個分支密鑰版本來滿足多個請求。但是您可以控制重複使用活動分支密鑰的程度，並確定活動分支鍵的旋轉頻率。分支索引鍵的作用中版本會保持作用中狀態，直到您旋轉它為止。以前版本的 Active 分支密鑰將不會用於執行加密操作，但仍然可以在解密操作中查詢和使用它們。

當您實例化分層密鑰環時，它會創建一個本地緩存。您可以指定一個快取限制，以定義分支索引鍵材料在到期並從快取中逐出之前儲存在本機快取中的時間上限。在作業中第一次指定 a `branch-key-id` 時，階層式金鑰圈會進行一次 AWS KMS 呼叫，以解密分支金鑰並組裝分支金鑰材料。然後，分支金鑰材料會儲存在本機快取中，並重複用於所有指定的加密和解密作業，`branch-key-id`直到快取限制到期為止。在本機快取中儲存分支金鑰材料可減少 AWS KMS 呼叫。例如，假設快取限制為 15 分鐘。如果您在該快取限制內執行 10,000 個加密作業，則傳統 AWS KMS 金鑰圈需要進行 10,000 次 AWS KMS 呼叫，才能滿足 10,000 個加密作業。如果您有一個作用中`branch-key-id`，階層式金鑰圈只需要進行一次 AWS KMS 呼叫即可滿足 10,000 個加密作業。

本機快取由兩個分割區組成，一個用於加密作業，另一個用於解密作業。加密分割區會儲存從使用中分支金鑰組合而來的分支金鑰材料，並在所有加密作業中重複使用，直到快取限制到期為止。解密分割區會儲存為解密作業中識別的其他分支金鑰版本所組合的分支金鑰材料。解密分區可以一次存儲多個活動分支密鑰材料版本。當它設定為在多租戶環境中使用分支金鑰 ID 供應商時，該加密分割區也可以一次儲存多個分支金鑰材料版本。如需詳細資訊，請參閱 在多租戶環境中使用階層式金鑰圈。

> **Note**
>
> 「階層式鑰匙圈」中所有提及的內容， AWS Encryption SDK 請參閱「 AWS KMS 階層式鑰匙圈」。

主題

- 運作方式
- 必要條件
- 建立階層式金鑰圈
- 旋轉您的活動分支密鑰
- 在多租戶環境中使用階層式金鑰圈

運作方式

下列逐步解說說明階層式金鑰環如何組合加密和解密資料，以及金鑰圈針對加密和解密作業所進行的不同呼叫。如需有關包裝金鑰衍生和純文字資料金鑰加密程序的技術詳細資訊，請參閱AWS KMS 階層式金鑰圈技術詳細資料。

加密和簽署

以下逐步解說說明「階層式金鑰圈」如何組合加密材料並衍生唯一的環繞金鑰。

1. 加密方法會詢問加密材料的階層式金鑰圈。金鑰圈會產生純文字資料金鑰，然後檢查本機快取中是否有有效的分支材料來產生環繞金鑰。如果有有效的分支金鑰材料，金鑰圈會進入步驟 5。
2. 如果沒有有效的分支金鑰材料，階層式金鑰圈會查詢分支金鑰存放區以取得使用中的分支金鑰。
    a. 分支密鑰存儲調用 AWS KMS 以解密活動分支密鑰並返回明文活動分支密鑰。識別活動分支密鑰的數據被序列化，以在解密調用中提供額外的身份驗證數據（AAD）。 AWS KMS
    b. 分支密鑰存儲庫返回明文分支密鑰和標識它的數據，例如分支密鑰版本。

3. 階層式金鑰圈會組合分支金鑰材料 (純文字分支金鑰和分支金鑰版本)，並將它們的副本儲存在本機快取中。
4. 分層密鑰環從明文分支密鑰和 16 字節隨機鹽派生一個唯一的包裝密鑰。它使用派生的包裝密鑰來加密純文本數據密鑰的副本。

加密方法使用加密材料對數據進行加密。如需詳細資訊，請參閱如何 AWS Encryption SDK 加密資料。

解密和驗證

以下逐步解說說明「階層式金鑰圈」如何組合解密材料並解密加密的資料金鑰。

1. 解密方法可識別加密訊息中的加密資料金鑰，並將其傳遞至階層式金鑰圈。
2. 階層式金鑰環還原序列化識別加密資料金鑰的資料，包括分支金鑰版本、16 位元組 salt，以及說明資料金鑰加密方式的其他資訊。

   如需詳細資訊，請參閱 AWS KMS 分層鑰匙圈技術細節。
3. 階層式金鑰圈會檢查本機快取中是否有符合步驟 2 中所識別的分支金鑰版本的有效分支金鑰材料。如果有有效的分支金鑰材料，金鑰圈會進入步驟 6。
4. 如果沒有有效的分支金鑰材料，階層式金鑰圈會查詢分支金鑰存放區，找出與步驟 2 中所識別的分支金鑰版本相符的分支金鑰存放區。

   a. 分支密鑰存儲調用 AWS KMS 解密分支密鑰並返回明文活動分支密鑰。識別活動分支密鑰的數據被序列化，以在解密調用中提供額外的身份驗證數據（AAD）。 AWS KMS

   b. 分支密鑰存儲庫返回明文分支密鑰和標識它的數據，例如分支密鑰版本。
5. 階層式金鑰圈會組合分支金鑰材料 (純文字分支金鑰和分支金鑰版本)，並將它們的副本儲存在本機快取中。
6. 階層式金鑰圈使用組裝好的分支金鑰材料和步驟 2 中識別的 16 位元組鹽來重現加密資料金鑰的唯一包裝金鑰。
7. 階層式金鑰圈使用複製的包裝金鑰來解密資料金鑰，並傳回純文字資料金鑰。

解密方法使用解密材料和純文本數據密鑰來解密加密的消息。如需詳細資訊，請參閱如何 AWS Encryption SDK 解密加密的郵件。

必要條件

AWS Encryption SDK 不需要一個 AWS 帳戶 ，它不依賴於任何 AWS 服務. 不過，階層式金鑰圈取決於 AWS KMS 和 Amazon DynamoDB。

若要使用階層式金鑰圈，您需要使用 K MS: 解密權限的對稱 AWS KMS key 加密。您也可以使用對稱加密多區域金鑰。如需有關權限的詳細資訊 AWS KMS keys，請參閱AWS Key Management Service 開發人員指南中的驗證和存取控制。

在建立和使用階層式金鑰圈之前，您必須先建立分支金鑰存放區，並使用第一個使用中的分支金鑰來填入該分支金鑰存放區。

步驟 1：設定新的金鑰存放區服務

金鑰存放區服務提供數種 API 作業，例如`CreateKeyStore`和`CreateKey`，可協助您組合階層式金鑰圈必要條件並管理分支金鑰存放區。

下列範例會建立金鑰存放區服務。您必須指定 DynamoDB 資料表名稱做為分支金鑰存放區的名稱、分支金鑰存放區的邏輯名稱，以及識別可保護分支金鑰之 KMS 金鑰的 KMS 金鑰 ARN。

邏輯金鑰存放區名稱會以密碼方式繫結至儲存在資料表中的所有資料，以簡化 DynamoDB 還原作業。邏輯金鑰存放區名稱可以與 DynamoDB 表名稱相同，但不一定要這樣做。建議您在第一次設定金鑰存放區服務時，將 DynamoDB 表名稱指定為邏輯資料表名稱。您必須永遠指定相同的邏輯資料表名稱。如果您的分支金鑰存放區名稱在從備份還原 DynamoDB 表後變更，邏輯金鑰存放區名稱會對應至您指定的 DynamoDB 表名稱，以確保階層式金鑰圈仍可存取您的分支金鑰存放區。

> **Note**
> 邏輯金鑰存放區名稱會包含在呼叫的所有金鑰存放區服務 API 作業的加密內容中 AWS KMS。加密內容並非秘密，其值 (包括邏輯金鑰存放區名稱) 會出現在記錄檔中的純文字中。 AWS CloudTrail

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);
```

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();
```

步驟 2：CreateKeyStore打電話創建分支密鑰存儲

以下操作會創建將持續存儲並保護您的分支密鑰的分支密鑰存儲區。

C# / .NET

```
var createKeyStoreOutput = keystore.CreateKeyStore(new CreateKeyStoreInput());
```

Java

```
keystore.CreateKeyStore(CreateKeyStoreInput.builder().build());
```

此CreateKeyStore作業會使用您在步驟 1 中指定的資料表名稱和下列必要值建立 DynamoDB 表格。

	分割區索引鍵	排序索引鍵
基本資料表	branch-key-id	type

> **Note**
>
> 您可以手動建立做為分支金鑰存放區的 DynamoDB 表格，而非使用此作業。CreateKeyStore如果您選擇手動建立分支金鑰存放區，則必須為分割區和排序索引鍵指定下列字串值：
>
> - 分區索引鍵: branch-key-id
> - 排序索引鍵：type

步驟 3：打電話創**CreateKey**建一個新的活動分支密鑰

下列作業會使用您在步驟 1 中指定的 KMS 金鑰建立新的作用中分支金鑰，並將作用中分支金鑰新增至您在步驟 2 中建立的 DynamoDB 表格。

撥打電話時CreateKey，您可以選擇指定下列選用值。

- 分支鍵標識符：定義一個自定義branch-key-id。

  若要建立自訂branch-key-id，您還必須在encryptionContext參數中包含額外的加密內容。
- 加密內容：定義一組選用的非秘密金鑰-值配對，可在 kms: call 中包含的加密內容中提供額外的驗證資料 (AAD)。GenerateDataKeyWithoutPlaintext

  此額外的加密內容會以aws-crypto-ec:前置詞顯示。

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
});
```

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();
```

首先，作CreateKey業會產生下列值。

- 版本 4 的通用唯一識別碼 (UUID) branch-key-id (除非您指定了自訂)。branch-key-id
- 分支密鑰版本的版本 4 UUID
- 以 ISO 8601 日期和時間格式表示的 Atimestamp，採用國際標準時間 (UTC) 表示。

然後，作CreateKey業會呼叫 kms：GenerateDataKeyWithoutPlaintext使用下列要求。

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

接下來，CreateKey作業會呼叫 kms: ReEncrypt 透過更新加密內容來建立分支金鑰的使用中記錄。

最後，CreateKey操作調用 ddb: TransactWriteItems 編寫一個新項目，該項目將在步驟 2 中創建的表中保留分支密鑰。項目具有下列屬性。

```
{
    "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"
}
```

建立階層式金鑰圈

若要初始化階層式金鑰圈，您必須提供下列值：

- 分支金鑰存放區名稱

  您建立做為分支金鑰存放區使用的 DynamoDB 表的名稱。

-

  快取限制存留時間 (TTL)

  本機快取中的分支金鑰材料項目在到期前可以使用的時間 (以秒為單位)。該值必須大於零。當快取限制 TTL 到期時，會從本機快取中逐出項目。

- 分支密鑰標識符

  識別分支金鑰存放區中作用中分支金鑰的。`branch-key-id`

> **Note**
> 若要初始化多租戶使用的階層式金鑰環，您必須指定分支金鑰 ID 供應商，而不是指定. `branch-key-id` 如需詳細資訊，請參閱 在多租戶環境中使用階層式金鑰圈。

- (選擇性) 快取

  如果您想要自訂快取類型或可儲存在本機快取中的分支金鑰材料項目數目，請在初始化金鑰環時指定快取類型和項目容量。

  緩存類型定義了線程模型。階層式金鑰環提供三種支援多租戶環境的快取類型：預設值、、MultiThreaded. StormTracking

  如果您未指定快取，階層式金鑰圈會自動使用預設快取類型，並將項目容量設定為 1000。

Default (Recommended)

對於大多數用戶來說，默認緩存滿足他們的線程要求。默認緩存被設計為支持大量多線程環境。當分支金鑰材料項目到期時，預設快取會通知一個執行緒分支金鑰材料項目將提前 10 秒到期，以防止多個執行緒呼叫 AWS KMS 和 Amazon DynamoDB。這確保只有一個線程發送一個請求 AWS KMS 來刷新緩存。

若要使用預設快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。

C # /.

```
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

Java

```
.cache(CacheType.builder()
        .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

預設值和 StormTracking 快取支援相同的執行緒模型，但您只需要指定項目容量，即可使用預設快取來初始化階層式金鑰環。如需更精細的快取自訂項目，請使用 StormTracking 快取。

MultiThreaded

MultiThreaded 快取可在多執行緒環境中安全使用，但不提供任何可減少 AWS KMS 或 Amazon DynamoDB 呼叫的功能。因此，當分支金鑰材料項目到期時，所有執行緒都會同時收到通知。這可能會導致多次 AWS KMS 呼叫以重新整理快取。

若要使用 MultiThreaded 快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。
- 項目修剪尾端大小：定義達到入口容量時要修剪的項目數。

C # /.

```
CacheType multithreadedCache = new CacheType
{
```

```
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())
```

StormTracking

該 StormTracking 緩存被設計為支持大量多線程環境。當分支金鑰材料項目到期時，StormTracking 快取會通知一個執行緒，預先通知一個執行緒分支金鑰材料項目即將到期，以防止多個執行緒呼叫 AWS KMS 和 Amazon DynamoDB。這確保只有一個線程發送一個請求 AWS KMS 來刷新緩存。

若要使用 StormTracking 快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。
- 項目修剪尾部大小：定義一次要修剪的分支關鍵材料項目數。

  預設值：1 個項目
- 寬限期：定義嘗試重新整理分支金鑰材料到期前的秒數。

  預設值：10 秒
- 寬限間隔：定義嘗試重新整理分支金鑰材料之間的秒數。

  預設值：1 秒
- 扇出：定義可同時嘗試重新整理分支金鑰材料的次數。

  預設值：20 次嘗試
- 存留時間 (TTL)：定義嘗試重新整理分支金鑰材料逾時之前的秒數。每當緩存返回`NoSuchEntry`以響應 a 時`GetCacheEntry`，該分支密鑰都會被視為處於飛行狀態，直到用條目寫`PutCache`入相同的密鑰為止。

預設值：20 秒

- 睡眠：定義如果超過執行緒應該睡眠的秒數。`fanOut`

  預設值：20 毫秒

C#/.

```
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 20,
        SleepMilli = 20
    }
};
```

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(20)
        .sleepMilli(20)
        .build())
```

- （可選）授予令牌列表

  如果您使用授權控制階層式金鑰圈中 KMS 金鑰的存取權，則必須在初始化金鑰環時提供所有必要的授權權杖。

下列範例會初始化階層式金鑰圈，其快取限制 TLL 為 600 秒，項目容量為 1000。

C# / .NET

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

// Instantiate the keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
    KeyStore = branchKeyStoreName,
    BranchKeyId = branch-key-id,
    Cache = new CacheType { Default = new DefaultCache{EntryCapacity = 1000} },
    TtlSeconds = 600
};
```

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(1000)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

旋轉您的活動分支密鑰

每個分支鍵一次只能有一個活動版本。分層密鑰環通常使用每個活動分支密鑰版本來滿足多個請求。但是您可以控制重複使用活動分支密鑰的程度，並確定活動分支鍵的旋轉頻率。

分支密鑰不用於加密純文本數據密鑰。它們被用來派生加密明文數據密鑰的唯一包裝密鑰。包裝密鑰派生過程會產生具有 28 個字節隨機性的唯一 32 字節包裝密鑰。這意味著在發生密碼編譯耗損之前，分

支密鑰可以導出超過 79 個八進制或 2^{96} 個唯一包裝密鑰。儘管這種用盡風險非常低，但由於業務或合約規則或政府法規，您可能需要輪換活動中的分支密鑰。

分支索引鍵的作用中版本會保持作用中狀態，直到您旋轉它為止。以前版本的 Active 分支密鑰將不會用於執行加密操作，也無法用於導出新的包裝密鑰。但是仍然可以查詢它們並提供包裝密鑰以解密它們在活動時加密的數據密鑰。

使用金鑰存放區服務VersionKey作業輪換作用中的分支金鑰。當您旋轉作用中的分支索引鍵時，會建立新的分支索引鍵來取代先前的版本。當您旋轉作用中的分支索引鍵時，不branch-key-id會變更。當您呼叫時branch-key-id，您必須指定識別目前作用中分支索引鍵的VersionKey。

C# / .NET

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

Java

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

在多租戶環境中使用階層式金鑰圈

您可以使用在使用中分支金鑰及其衍生包裝金鑰之間建立的金鑰階層來支援多租戶環境，方法是為環境中的每個承租人建立分支金鑰。然後，階層式金鑰圈會使用其不同的分支金鑰來加密指定租用戶的所有資料。這可讓您透過分支金鑰隔離租用戶資料。

每個承租人都有自己的分支索引鍵，由唯一定義branch-key-id。每個版本一次只能有一個branch-key-id作用中版本。

您必須先為每個承租人建立分支金鑰，並建立分支金鑰 ID 供應商，才能初始化多租戶使用的階層式金鑰環。使用分支金鑰 ID 供應商為您建立易記的名稱，branch-key-ids以便更輕鬆地辨識租用戶branch-key-id的正確名稱。例如，易記名稱可讓您參考分支索引鍵，tenant1而不是b3f61619-4d35-48ad-a275-050f87e15122。

針對解密作業，您可以靜態設定單一階層式金鑰環，將解密限制為單一承租人，或使用分支金鑰識別碼供應商來識別負責解密訊息的承租人。

首先，遵循先決條件程序的步驟 1 和步驟 2。然後，使用下列程序為每個承租人建立分支金鑰、建立分支金鑰識別碼供應商，並初始化階層式金鑰圈以供多租戶使用。

步驟 1：為環境中的每個租用戶建立分支金鑰

打電話CreateKey給每個租戶。

下列作業會使用您在建立金鑰存放區服務時指定的 KMS 金鑰建立兩個分支金鑰，並將分支金鑰新增至您建立做為分支金鑰存放區使用的 DynamoDB 表格。相同的 KMS 金鑰必須保護所有分支金鑰。

C# / .NET

```
var branchKeyId1 = keystore.CreateKey(new CreateKeyInput());
var branchKeyId2 = keystore.CreateKey(new CreateKeyInput());
```

Java

```
CreateKeyOutput branchKeyId1 =
 keystore.CreateKey(CreateKeyInput.builder().build());
CreateKeyOutput branchKeyId2 =
 keystore.CreateKey(CreateKeyInput.builder().build());
```

步驟 2：建立分支金鑰 ID 供應商

下列範例會建立分支金鑰 ID 供應商。

C# / .NET

```
var branchKeySupplier =
    new ExampleBranchKeySupplier(branchKeyId1.BranchKeyIdentifier,
 branchKeyId2.BranchKeyIdentifier);
```

Java

```
IBranchKeyIdSupplier branchKeyIdSupplier = new ExampleBranchKeyIdSupplier(
        branchKeyId1.branchKeyIdentifier(), branchKeyId2.branchKeyIdentifier());
```

步驟 3：使用分支密鑰 ID 供應商初始化分層密鑰環

若要初始化階層式金鑰圈，您必須提供下列值：

- 分支金鑰存放區名稱
- 快取限制存留時間 (TTL)
- 分支金鑰 ID 供應商
- (選擇性) 快取

如果您想要自訂快取類型或可儲存在本機快取中的分支金鑰材料項目數目，請在初始化金鑰環時指定快取類型和項目容量。

緩存類型定義了線程模型。階層式金鑰環提供三種支援多租戶環境的快取類型：預設值、、MultiThreaded. StormTracking

如果您未指定快取，階層式金鑰圈會自動使用預設快取類型，並將項目容量設定為 1000。

Default (Recommended)

對於大多數用戶來說，默認緩存滿足他們的線程要求。默認緩存被設計為支持大量多線程環境。當分支金鑰材料項目到期時，預設快取會通知一個執行緒分支金鑰材料項目將提前 10 秒到期，以防止多個執行緒呼叫 AWS KMS 和 Amazon DynamoDB。這確保只有一個線程發送一個請求 AWS KMS 來刷新緩存。

若要使用預設快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。

C # /.

```
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

Java

```
.cache(CacheType.builder()
        .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

預設值和 StormTracking 快取支援相同的執行緒模型，但您只需要指定項目容量，即可使用預設快取來初始化階層式金鑰環。如需更精細的快取自訂項目，請使用 StormTracking 快取。

MultiThreaded

MultiThreaded 快取可在多執行緒環境中安全使用，但不提供任何可減少 AWS KMS 或 Amazon DynamoDB 呼叫的功能。因此，當分支金鑰材料項目到期時，所有執行緒都會同時收到通知。這可能會導致多次 AWS KMS 呼叫以重新整理快取。

若要使用 MultiThreaded 快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。
- 項目修剪尾端大小：定義達到入口容量時要修剪的項目數。

C # /.

```
CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())
```

StormTracking

該 StormTracking 緩存被設計為支持大量多線程環境。當分支金鑰材料項目到期時，StormTracking 快取會通知一個執行緒，預先通知一個執行緒分支金鑰材料項目即將到期，以防止多個執行緒呼叫 AWS KMS 和 Amazon DynamoDB。這確保只有一個線程發送一個請求 AWS KMS 來刷新緩存。

若要使用 StormTracking 快取來初始化階層式金鑰圈，請指定下列值：

- 項目容量：限制可儲存在本機快取中的分支金鑰材料項目數量。
- 項目修剪尾部大小：定義一次要修剪的分支關鍵材料項目數。

  預設值：1 個項目
- 寬限期：定義嘗試重新整理分支金鑰材料到期前的秒數。

  預設值：10 秒
- 寬限間隔：定義嘗試重新整理分支金鑰材料之間的秒數。

  預設值：1 秒
- 扇出：定義可同時嘗試重新整理分支金鑰材料的次數。

  預設值：20 次嘗試
- 存留時間 (TTL)：定義嘗試重新整理分支金鑰材料逾時之前的秒數。每當緩存返回`NoSuchEntry`以響應 a 時`GetCacheEntry`，該分支密鑰都會被視為處於飛行狀態，直到用條目寫`PutCache`入相同的密鑰為止。

  預設值：20 秒
- 睡眠：定義如果超過執行緒應該睡眠的秒數。`fanOut`

  預設值：20 毫秒

C＃/.

```
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 20,
        SleepMilli = 20
    }
};
```

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(20)
        .sleepMilli(20)
        .build())
```

- （可選）授予令牌列表

  如果您使用授權控制階層式金鑰圈中 KMS 金鑰的存取權，則必須在初始化金鑰環時提供所有必要的授權權杖。

下列範例會使用在步驟 2 中建立的分支金鑰 ID 供應商、快取限制 TLL 600 秒，以及 1000 的項目容量，初始化階層式金鑰圈。

C# / .NET

```
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
    KeyStore = keystore,
    BranchKeyIdSupplier = branchKeySupplier,
    Cache = new CacheType { Default = new DefaultCache{EntryCapacity = 1000} },
    TtlSeconds = 600
};
var keyring = mpl.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
```

```
                .Default(DefaultCache.builder()
                        .entryCapacity(100)
                        .build())
                .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

步驟 4：為每個分支鍵創建友好的名稱

下列範例會為在步驟 1 中建立的兩個分支索引鍵建立易記名稱。會 AWS Encryption SDK 使用加密內容將您定義的易記名稱對應至相關聯的名稱`branch-key-id`。

C# / .NET

```
// Create encryption contexts for the two branch keys created in Step 1
var encryptionContextA = new Dictionary<string, string>()
{
    // We will encrypt with branchKeyTenantA
    {"tenant", "TenantA"},
    {"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"}
};
var encryptionContextB = new Dictionary<string, string>()
{
    // We will encrypt with branchKeyTenantB
    {"tenant", "TenantB"},
    {"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 var esdk =  new ESDK(new
 AwsEncryptionSdkConfig());


var encryptInputA = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
```

```
    // Encrypt with branchKeyId1
    EncryptionContext = encryptionContextA
};

var encryptInputB = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    // Encrypt with branchKeyId2
    EncryptionContext = encryptionContextB
};

var encryptOutput = esdk.Encrypt(encryptInputA);
encryptOutput = esdk.Encrypt(encryptInputB);

// Use the encryption contexts to define friendly names for each branch key
public class ExampleBranchKeySupplier : IBranchKeyIdSupplier
{
    private string branchKeyTenantA;
    private string branchKeyTenantB;

    public ExampleBranchKeySupplier(string branchKeyTenantA, string
 branchKeyTenantB)
    {
        this.branchKeyTenantA = branchKeyTenantA;
        this.branchKeyTenantB = branchKeyTenantB;
    }

    public 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"];
        string branchkeyId;

        if (tenant.Equals("TenantA"))
        {
            GetBranchKeyIdOutput output = new GetBranchKeyIdOutput();
```

```
            output.BranchKeyId = branchKeyTenantA;
            return output;
        } else if (tenant.Equals("TenantB"))
        {
            GetBranchKeyIdOutput output = new GetBranchKeyIdOutput();
            output.BranchKeyId = branchKeyTenantB;
            return output;
        }
        else
        {
            throw new Exception("Item does not have a valid tenantID.");
        }
    }
}
```

Java

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

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


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

final CryptoResult<byte[], ?> encryptResultA = crypto.encryptData(keyring,
 plaintext, encryptionContextA);
```

```
final CryptoResult<byte[], ?> encryptResultB = crypto.encryptData(keyring,
 plaintext, encryptionContextB);

// Use the encryption contexts to define friendly names for each branch key
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();
    }
}
```

AWS KMS ECDH 鑰匙圈

Important

AWS KMS ECDH 鑰匙圈僅適用於第 3 版。 的 x 的 適用於 JAVA 的 AWS Encryption SDK。AWS KMS ECDH 鑰匙圈是在材料提供者資料庫的 1.5.0 版中引入的。

AWS KMS ECDH 金鑰圈使用非對稱金鑰協定AWS KMS keys來衍生雙方之間的共用對稱包裝金鑰。首先，金鑰圈會使用橢圓曲線迪菲-赫爾曼 (ECDH) 金鑰合約演算法，從寄件者的 KMS key pair 中的私密金鑰和收件者的公開金鑰衍生共用密碼。然後，金鑰圈會使用共用密碼衍生共用包裝金鑰，以保護您的資料加密金鑰。 AWS Encryption SDK 使用 (`KDF_CTR_HMAC_SHA384`) 衍生共用包裝金鑰的金鑰衍生函式，符合 NIST 針對金鑰衍生的建議。

密鑰派生函數返回 64 個字節的鍵控材料。為了確保雙方使用正確的金鑰資料， AWS Encryption SDK 會使用前 32 個位元組做為承諾金鑰，最後 32 個位元組作為共用包裝金鑰。解密時，如果金鑰圈無法重新產生儲存在郵件標頭加密文字上的相同承諾金鑰和共用包裝金鑰，作業就會失敗。例如，如果您使用以 Alice 私密金鑰和 Bob 公開金鑰設定的金鑰圈來加密資料，則使用 Bob 的私密金鑰和 Alice 公開金鑰設定的金鑰圈將會重新產生相同的承諾金鑰和共用包裝金鑰，並且能夠解密資料。如果 Bob 的公開金鑰不是來自 KMS key pair，Bob 可以建立原始 ECDH 金鑰圈來解密資料。

AWS KMS ECDH 密鑰環使用 AES-GCM 使用對稱密鑰對數據進行加密。然後使用 AES-GCM 使用派生的共享包裝密鑰對數據密鑰進行包絡加密。每個 AWS KMS ECDH 鑰匙圈只能有一個共用包裝金鑰，但您可以將多個 AWS KMS ECDH 鑰匙圈 (單獨或與其他鑰匙圈一起) 包含在多重鑰匙圈中。

主題

- AWS KMS ECDH 金鑰圈所需的權限
- 建立一個 AWS KMS ECDH 金鑰圈
- 建立 AWS KMS ECDH 探索金鑰圈

AWS KMS ECDH 金鑰圈所需的權限

AWS Encryption SDK 不需要 AWS 帳戶，也不依賴於任何 AWS 服務。但是，若要使用 AWS KMS ECDH 金鑰圈，您需要一個 AWS 帳戶以及金鑰圈 AWS KMS keys 中的下列最低權限。權限會根據您使用的金鑰合約結構描述而有所不同。

- 若要使用金`KmsPrivateKeyToStaticPublicKey`鑰合約結構描述加密GetPublicKey和解密資料，您需要寄件者的非對稱 KMS key pair DeriveSharedSecret 上的 kms: 和 kms:。如果您在實體

化金鑰圈時直接提供寄件者的 DER 編碼公開金鑰，則只需要 kms: 寄件者非對稱 KMS key pair 的 DeriveSharedSecret權限。

- 若要使用金KmsPublicKeyDiscovery鑰協定結構描述解密資料，您需要指定的非對稱 KMS key pair 的 kms: DeriveSharedSecret 和 kms: GetPublicKey 權限。

建立一個 AWS KMS ECDH 金鑰圈

若要建立加密和解密資料的 AWS KMS ECDH 金鑰圈，您必須使用金鑰合約結構描述KmsPrivateKeyToStaticPublicKey。若要使用KmsPrivateKeyToStaticPublicKey金鑰合約結構描述初始化 AWS KMS ECDH 金鑰圈，請提供下列值：

- 寄件者 AWS KMS key 識別碼

  必須識別非對稱 NIST 建議的橢圓曲線 (ECC) KMS key pair，其值為。KeyUsage KEY_AGREEMENT發件人的私鑰用於導出共享密鑰。
- (選擇性) 寄件者的公開金鑰

  必須是一個德編碼的 X.509 公開金鑰，也稱為 SubjectPublicKeyInfo (SPKI)，如 RFC 5280 中所定義。

  此 AWS KMS GetPublicKey作業會以必要的 DER 編碼格式傳回非對稱 KMS key pair 的公開金鑰。

  若要減少金鑰圈 AWS KMS 撥打的通話次數，您可以直接提供寄件者的公開金鑰。如果沒有為寄件者的公開金鑰提供值，金鑰圈會呼叫擷 AWS KMS 取寄件者的公開金鑰。
- 收件者的公開金鑰

  您必須提供收件者的 DER 編碼 X.509 公開金鑰，也稱為 SubjectPublicKeyInfo (SPKI)，如 RFC 5280 中所定義。

  此 AWS KMS GetPublicKey作業會以必要的 DER 編碼格式傳回非對稱 KMS key pair 的公開金鑰。
- 曲線規格

  識別指定索引鍵對中的橢圓曲線規格。寄件者和收件者的金鑰配對必須具有相同的曲線規格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512
- （可選）授予令牌列表

  如果您使用授權控制 AWS KMS ECDH 金鑰圈中 KMS 金鑰的存取權，則必須在初始化金鑰環時提供所有必要的授權權杖。

Java

下列範例會使用寄件者的 KMS 金鑰、寄件者的公開金鑰和收件者的公開金鑰建立 AWS KMS ECDH 金鑰圈。此範例使用選用senderPublicKey參數來提供寄件者的公開金鑰。如果您未提供寄件者的公開金鑰，金鑰圈會呼叫 AWS KMS 以擷取寄件者的公開金鑰。寄件者和收件者的金鑰配對都在ECC_NIST_P256曲線上。

```
// 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();
```

建立 AWS KMS ECDH 探索金鑰圈

解密時，最佳做法是指定 AWS Encryption SDK 可以使用的金鑰。若要遵循此最佳做法，請使用 AWS KMS ECDH 金鑰圈搭配KmsPrivateKeyToStaticPublicKey金鑰合約結構描述。不過，您也可以建立 AWS KMS ECDH 探索金鑰圈，也就是 AWS KMS ECDH 金鑰圈，可以將指定 KMS key pair 的公開金鑰與儲存在郵件加密文字中的收件者公開金鑰相符的任何郵件進行解密。

> **Important**
>
> 當您使用金KmsPublicKeyDiscovery鑰合約結構描述解密郵件時，您會接受所有公開金鑰，不論其擁有者為何。

若要使用KmsPublicKeyDiscovery金鑰合約結構描述初始化 AWS KMS ECDH 金鑰圈，請提供下列值：

- 收件人的 AWS KMS key 識別碼

  必須識別非對稱 NIST 建議的橢圓曲線 (ECC) KMS key pair，其值為。KeyUsage KEY_AGREEMENT
- 曲線規格

  識別收件者的 KMS key pair 中的橢圓曲線規格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512
- （可選）授予令牌列表

  如果您使用授權控制 AWS KMS ECDH 金鑰圈中 KMS 金鑰的存取權，則必須在初始化金鑰環時提供所有必要的授權權杖。

Java

下列範例會在曲線上建立具有 KMS key pair 的 AWS KMS ECDH 探索金鑰圈。ECC_NIST_P256您必須擁有指定 KMS key pair 的 kms: GetPublicKey 和 kms: DeriveSharedSecret 權限。此金鑰圈可以解密指定 KMS key pair 的公開金鑰與儲存在郵件加密文字上的收件者公開金鑰相符的任何郵件。

```
// 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();
```

原始 AES keyring

AWS Encryption SDK 可讓您使用您提供的 AES 對稱金鑰做為保護資料金鑰的包裝金鑰。您必須產生、儲存及保護金鑰資料，最好是在硬體安全模組 (HSM) 或金鑰管理系統中。當您需要提供包裝金鑰並在本機或離線加密資料金鑰時，請使用 Raw AES 金鑰圈。

Raw AES 金鑰環會使用 AES-GCM 演算法和您指定為位元組陣列的包裝金鑰來加密資料。您只能在每個 Raw AES 金鑰圈中指定一個環繞金鑰，但您可以在多重金鑰圈中包含多個 Raw AES 鑰匙圈，單獨或與其他鑰匙圈一起包含多個 Raw AES 鑰匙圈。

Raw AES 金鑰圈等同於與 AES 加密金鑰搭配使用 適用於 Python 的 AWS Encryption SDK 時，與中的RawMasterKey類別 適用於 JAVA 的 AWS Encryption SDK 和類別相互操作。JceMasterKey您可以使用一個實作來加密資料，並利用使用相同包裝金鑰的任何其他實作來解密資料。如需詳細資訊，請參閱 Keyring 相容性。

關鍵命名空間和名稱

若要識別金鑰圈中的 AES 金鑰，Raw AES 金鑰圈會使用您提供的金鑰命名空間和金鑰名稱。這些值並非機密。它們會以純文字顯示在加密作業傳回的加密郵件標頭中。我們建議您使用 HSM 或金鑰管理系統的金鑰命名空間，以及識別該系統中 AES 金鑰的金鑰名稱。

> **Note**
>
> 金鑰命名空間和金鑰名稱等同於和中的「提供者 ID」(或「提供者」) 和「金鑰 ID」欄位`RawMasterKey`。`JceMasterKey`
> 適用於 C 的 AWS Encryption SDK 和 AWS Encryption SDK 的 .NET 會保留 KMS 金鑰的`aws-kms`金鑰命名空間值。請勿將此命名空間值用於 Raw AES 金鑰圈或 RSA 金鑰圈與這些程式庫一起使用。

如果您構建不同的密鑰環來加密和解密給定的消息，則命名空間和名稱值非常重要。如果解密金鑰圈中的金鑰命名空間和金鑰名稱與加密金鑰圈中的金鑰命名空間和金鑰名稱不完全相符且區分大小寫，即使金鑰材料位元組相同，也不會使用解密金鑰圈。

例如，您可以使用金鑰命名空間`HSM_01`和金鑰名稱`AES_256_012`來定義 Raw AES 金鑰環。然後，您可以使用該密鑰環來加密某些數據。若要解密該資料，請使用相同的金鑰命名空間、金鑰名稱和金鑰材料建構 Raw AES 金鑰環。

下列範例說明如何建立 Raw AES 金鑰圈。`AESWrappingKey`變數代表您提供的關鍵材料。

C

若要在中實例化 Raw AES 金鑰圈 適用於 C 的 AWS Encryption SDK，請使用. aws_cryptosdk_raw_aes_keyring_new() 如需完整範例，請參閱「鍵盤」。

```
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

若要在中 AWS Encryption SDK 為 .NET 建立原始 AES 金鑰圈，請使用此方materialProviders.CreateRawAesKeyring()法。如需完整範例，請參閱 RAWaes .cs KeyringExample。

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 x。

```
// 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

適用於 JavaScript 的 AWS Encryption SDK 在瀏覽器中從 API 獲取其加密原語。WebCrypto在建構金鑰圈之前，您必須使用RawAesKeyringWebCrypto.importCryptoKey()將原始金鑰材料匯入 WebCrypto 後端。即使所有呼叫都是 WebCrypto 非同步的，這也可以確保金鑰圈完成。

然後，要實例化 Raw AES 密鑰環，請使用該RawAesKeyringWebCrypto()方法。您必須根據密鑰材料的長度指定 AES 包裝算法（「包裝套件）。如需完整範例，請參閱 aes_simple.ts (瀏覽器)。JavaScript

```
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

若要在 Node.js 中實例化原始 AES 金鑰環，適用於 JavaScript 的 AWS Encryption SDK 請建立類別的實體。 RawAesKeyringNode您必須根據金鑰材料的長度指定 AES 包裝演算法（「包裝套件」)。如需完整範例，請參閱 「簡單」(Node.js)。JavaScript

```
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

若要在中實例化 Raw AES 金鑰圈 適用於 JAVA 的 AWS Encryption SDK，請使用. `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);
```

原始 RSA keyring

Raw RSA 金鑰圈會使用您提供的 RSA 公開金鑰和私密金鑰，對本機記憶體中的資料金鑰執行非對稱加密和解密。您必須產生、儲存及保護私密金鑰，最好是在硬體安全模組 (HSM) 或金鑰管理系統中。加密函數會根據 RSA 公開金鑰加密資料金鑰。解密函數會使用私有金鑰解密資料金鑰。您可以從數個 RSA 填補模式中選擇。

加密和解密的原始 RSA keyring，必須包含非對稱公開金鑰和私有金鑰對。不過，您可以使用只有公開金鑰的 RSA 金鑰圈來加密資料，也可以使用只有私密金鑰的 RSA 金鑰圈來解密資料。您可以在多鑰匙圈中包含任何 Raw RSA 鑰匙圈。如果您使用公開金鑰和私密金鑰來設定 Raw RSA 金鑰圈，請確定它們是相同 key pair 的一部分。的某些語言實作不 AWS Encryption SDK 會使用不同配對的金鑰建構 Raw RSA 金鑰圈。其他人依靠您來驗證您的密鑰來自同一個 key pair。

原始 RSA 金鑰圈等同於JceMasterKey中和中的，並與 RSA 非對稱加密金鑰搭配使用 適用於 Python 的 AWS Encryption SDK 時，與RawMasterKey中的相互操作。 適用於 JAVA 的 AWS Encryption

SDK 您可以使用一個實作來加密資料，並利用使用相同包裝金鑰的任何其他實作來解密資料。如需詳細資訊，請參閱 Keyring 相容性。

> **Note**
>
> 原始 RSA 金鑰圈不支援非對稱 KMS 金鑰。如果您想要使用非對稱的 RSA KMS 金鑰，請參閱第 4 版。 AWS Encryption SDK 適用於 .NET 和版本 3 的 x。 使用對稱加密 (SYMMETRIC_DEFAULT) 或非對稱 R SA 的 適用於 JAVA 的 AWS Encryption SDK 支援 AWS KMS 金鑰環的 x 個。 AWS KMS keys
>
> 如果您使用包含 RSA KMS 金鑰的公開金鑰的原始 RSA 金鑰圈來加密資料，則無法解密資料。 AWS Encryption SDK AWS KMS 您無法將 AWS KMS 非對稱 KMS 金鑰的私密金鑰匯出至原始 RSA 金鑰圈。解 AWS KMS 密作業無法解密 AWS Encryption SDK 傳回的加密訊息。

在中建構 RSA 原始金鑰圈時 適用於 C 的 AWS Encryption SDK，請務必提供包含每個金鑰的 PEM 檔案內容做為空結尾的 C 字串，而不是作為路徑或檔案名稱。在中構建 Raw RSA 密鑰環時 JavaScript，請注意與其他語言實現的潛在不兼容性。

命名空間和名稱

若要識別金鑰圈中的 RSA 金鑰材料，Raw RSA 金鑰圈會使用您提供的金鑰命名空間和金鑰名稱。這些值並非機密。它們會以純文字顯示在加密作業傳回的加密郵件標頭中。我們建議您在 HSM 或金鑰管理系統中使用識別 RSA key pair (或其私密金鑰) 的金鑰命名空間和金鑰名稱。

> **Note**
>
> 金鑰命名空間和金鑰名稱等同於和中的「提供者 ID」(或「提供者」) 和「金鑰 ID」欄位`RawMasterKey`。`JceMasterKey`
>
> 會 適用於 C 的 AWS Encryption SDK 保留 KMS 金鑰的`aws-kms`金鑰命名空間值。請勿將其與原始 AES 鑰匙圈或原始 RSA 鑰匙圈一起使用。 適用於 C 的 AWS Encryption SDK

如果您構建不同的密鑰環來加密和解密給定的消息，則命名空間和名稱值非常重要。如果解密金鑰圈中的金鑰命名空間和金鑰名稱與加密金鑰圈中的金鑰命名空間和金鑰名稱不完全相符且區分大小寫，即使金鑰來自相同的金鑰組，也不會使用解密金鑰圈。

無論金鑰圈包含 RSA 公開金鑰、RSA 私密金鑰或 key pair 中的兩個金鑰，加密和解密金鑰環中的金鑰資料的金鑰名稱空間和金鑰資料都必須相同。例如，假設您使用具有金鑰命名空間`HSM_01`和金鑰名稱

之 RSA 公開金鑰的 RSA 金鑰圈來加密資料。RSA_2048_06若要解密該資料，請使用私密金鑰 (或金鑰組) 以及相同的金鑰命名空間和名稱來建構 Raw RSA 金鑰環。

填充模式

您必須為用於加密和解密的 RSA 金鑰環指定填補模式，或使用語言實作的功能來為您指定該密鑰環。

AWS Encryption SDK 支持以下填充模式，受到每種語言的約束。我們建議使用 OAEP 填充模式，特別是使用 SHA-256 和 MGF1 搭配 SHA-256 填充模式。只有向下相容性才支援 PKCS1 填補模式。

- 使用 SHA-1 和 MGF1 配備 SHA-1 填充的「老有增值計劃」
- 使用 SHA-256 和 MGF1 配備 SHA-256 填充的「老有增值計劃」
- 使用 SHA-384 和 MGF1 配備 SHA-384 填充的「老有增值計劃」
- 使用 SHA-512 和 MGF1 配備 SHA-512 填充的「老有增值計劃」
- PKCS1 填充

下列範例說明如何使用 RSA 金鑰組的公開和私密金鑰，以及使用 SHA-256 和 MGF1 (含 SHA-256 填補模式) 建立原始 RSA 金鑰圈。RSAPublicKey和RSAPrivateKey變數代表您提供的關鍵材料。

C

若要在中建立 RSA 金鑰圈 適用於 C 的 AWS Encryption SDK，請使用。aws_cryptosdk_raw_rsa_keyring_new

在中建構 RSA 原始金鑰圈時 適用於 C 的 AWS Encryption SDK，請務必提供包含每個金鑰的 PEM 檔案內容做為空結尾的 C 字串，而不是作為路徑或檔案名稱。如需完整範例，請參閱 Rw _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

若要在 .NET 中實例化原始 RSA 金鑰環， AWS Encryption SDK 請使用方法。materialProviders.CreateRawRsaKeyring()如需完整範例，請參閱勞爾薩 .cs KeyringExample。

下列範例使用版本 4。 AWS Encryption SDK 用於 .NET 的 x。

```
// 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

瀏覽器 適用於 JavaScript 的 AWS Encryption SDK 中的從庫中獲取其加密原語。WebCrypto在建構金鑰圈之前，您必須使用importPublicKey()和/或importPrivateKey()將原始金鑰材料匯入 WebCrypto 後端。即使所有呼叫都是 WebCrypto 非同步的，這也可以確保金鑰圈完成。匯入方法採用的物件包含包裝演算法及其填補模式。

匯入金鑰材料後，請使用該RawRsaKeyringWebCrypto()方法實例化金鑰環。在中構建 Raw RSA 密鑰環時 JavaScript，請注意與其他語言實現的潛在不兼容性。

如需完整範例，請參閱 [rsa_simple.](#) ts (瀏覽器)。JavaScript

```
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

若要在 Node.js 中實例化原始 RSA 金鑰環， 適用於 JavaScript 的 AWS Encryption SDK 請建立類別的新執行個體。`RawRsaKeyringNode`該`wrapKey`參數保存公鑰。`unwrapKey`參數會保留私密金鑰。雖然您可以指定偏好的填充模式，但建`RawRsaKeyringNode`構函式會為您計算預設填補模式。

在中構建原始 RSA 密鑰環時 JavaScript，請注意與其他語言實現的潛在不兼容性。

如需完整範例，請參閱簡單的說明 (Node.js)。 JavaScript

```
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);
```

原始 ECDH 鑰匙圈

> **Important**
>
> 原始 ECDH 鑰匙圈僅適用於第 3 版。 的 x 的 適用於 JAVA 的 AWS Encryption SDK。原始 ECDH 鑰匙圈是在材料提供者資料庫的 1.5.0 版中引入的。

Raw ECDH 金鑰圈會使用您提供的橢圓曲線公開-私密金鑰配對，以衍生雙方之間的共用包裝金鑰。首先，金鑰圈會使用寄件者的私密金鑰、收件者的公開金鑰和橢圓曲線迪菲-赫爾曼 (ECDH) 金鑰協定演算法衍生共用密碼。然後，金鑰圈會使用共用密碼衍生共用包裝金鑰，以保護您的資料加密金鑰。 AWS Encryption SDK 使用 (`KDF_CTR_HMAC_SHA384`) 衍生共用包裝金鑰的金鑰衍生函式，符合 NIST 針對金鑰衍生的建議。

密鑰派生函數返回 64 個字節的密鑰材料。為了確保雙方使用正確的金鑰材料， AWS Encryption SDK 會使用前 32 個位元組做為承諾金鑰，並使用最後 32 個位元組做為共用包裝金鑰。解密時，如果金鑰圈無法重新產生儲存在郵件標頭加密文字上的相同承諾金鑰和共用包裝金鑰，作業就會失敗。例如，如果您使用以 Alice 私密金鑰和 Bob 公開金鑰設定的金鑰圈來加密資料，則使用 Bob 的私密金鑰和 Alice 公開金鑰設定的金鑰圈將會重新產生相同的承諾金鑰和共用包裝金鑰，並且能夠解密資料。如果 Bob 的公開金鑰來自一 AWS KMS key 對，則 Bob 可以建立 AWS KMS ECDH 金鑰圈來解密資料。

原始 ECDH 密鑰環使用 AES-GCM 使用對稱密鑰對數據進行加密。然後使用 AES-GCM 使用派生的共享包裝密鑰對數據密鑰進行包絡加密。每個 Raw ECDH 鑰匙圈只能有一個共用包裝金鑰，但您可以將多個 Raw ECDH 鑰匙圈（單獨或與其他鑰匙圈一起）包含在多重鑰匙圈中。

您必須負責產生、儲存及保護私密金鑰，最好是在硬體安全模組 (HSM) 或金鑰管理系統中。寄件者和收件者的金鑰配對大部分位於相同的橢圓曲線上。 AWS Encryption SDK 支援以下橢圓形切割規格：

- `ECC_NIST_P256`

- `ECC_NIST_P384`
- `ECC_NIST_P512`

建立原始 ECDH 金鑰圈

Raw ECDH 金鑰圈支援三個金鑰合約結構描述：`RawPrivateKeyToStaticPublicKeyEphemeralPrivateKeyToStaticPublicKey`、和。`PublicKeyDiscovery`您選取的金鑰協定結構描述會決定您可以執行哪些密碼編譯作業，以及鍵控材料的組合方式。

主題

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

使用`RawPrivateKeyToStaticPublicKey`金鑰合約結構描述，在金鑰圈中以靜態方式設定寄件者的私密金鑰和收件者的公開金鑰。此金鑰合約結構描述可以加密和解密資料。

若要使用`RawPrivateKeyToStaticPublicKey`金鑰合約結構描述初始化 Raw ECDH 金鑰圈，請提供下列值：

- 寄件者的私密金鑰

  您必須提供寄件者的 PEM 編碼私密金鑰 (PKCS #8 PrivateKeyInfo 結構)，如 RFC 5958 中所定義。
- 收件者的公開金鑰

  您必須提供收件者的 DER 編碼 X.509 公開金鑰，也稱為 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定義。

  您可以指定非對稱金鑰合約 KMS key pair 的公開金鑰，或從外部產生的 key pair 中指定公開金鑰 AWS。
- 曲線規格

  識別指定索引鍵對中的橢圓曲線規格。寄件者和收件者的金鑰配對必須具有相同的曲線規格。

  有效值：`ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

Java

下列 Java 範例會使用RawPrivateKeyToStaticPublicKey金鑰合約結構描述，以靜態方式設定寄件者的私密金鑰和收件者的公開金鑰。兩個鍵對都在ECC_NIST_P256曲線上。

```
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);
}
```

EphemeralPrivateKeyToStaticPublicKey

使用EphemeralPrivateKeyToStaticPublicKey金鑰合約結構描述設定的金鑰圈會在本機建立新的 key pair，並為每個加密呼叫衍生唯一的共用包裝金鑰。

此金鑰合約結構描述只能加密郵件。若要解密使用`EphemeralPrivateKeyToStaticPublicKey`金鑰合約結構描述加密的郵件，您必須使用以相同收件者公開金鑰設定的探索金鑰合約結構描述。若要解密，您可以使用 Raw ECDH 金鑰圈搭配金`PublicKeyDiscovery`鑰合約演算法，或者，如果收件者的公開金鑰來自非對稱金鑰合約 KMS 金鑰組，您可以使用 AWS KMS ECDH 金鑰圈搭配金鑰合約結構描述KmsPublicKeyDiscovery。

若要使用`EphemeralPrivateKeyToStaticPublicKey`金鑰合約結構描述初始化 Raw ECDH 金鑰圈，請提供下列值：

- 收件者的公開金鑰

  您必須提供收件者的 DER 編碼 X.509 公開金鑰，也稱為 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定義。

  您可以指定非對稱金鑰合約 KMS key pair 的公開金鑰，或從外部產生的 key pair 中指定公開金鑰 AWS。
- 曲線規格

  識別指定公開金鑰中的橢圓曲線規格。

  在加密時，密鑰環在指定的曲線上創建一個新的密鑰對，並使用新的私鑰和指定的公鑰來導出共享包裝密鑰。

  有效值：`ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

Java

下列範例會使用`EphemeralPrivateKeyToStaticPublicKey`金鑰合約結構描述建立 Raw ECDH 金鑰圈。在加密時，密鑰環將在指定的`ECC_NIST_P256`曲線上本地創建一個新的密 key pair。

```
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);
}
```

PublicKeyDiscovery

解密時，最佳作法是指定 AWS Encryption SDK 可以使用的包裝金鑰。若要遵循此最佳做法，請使用指定寄件者私密金鑰和收件者公開金鑰的 ECDH 金鑰圈。不過，您也可以建立 Raw ECDH 探索金鑰圈，也就是 Raw ECDH 金鑰圈，可以將指定金鑰的公開金鑰與儲存在郵件加密文字中的收件者公開金鑰相符的任何郵件進行解密。此金鑰合約結構描述只能解密訊息。

> ⚠ Important
>
> 當您使用金`PublicKeyDiscovery`鑰合約結構描述解密郵件時，您會接受所有公開金鑰，不論其擁有者為何。

若要使用`PublicKeyDiscovery`金鑰合約結構描述初始化 Raw ECDH 金鑰圈，請提供下列值：

- 收件者的靜態私密金鑰

  您必須提供收件者的 PEM 編碼私密金鑰 (PKCS #8 PrivateKeyInfo 結構)，如 RFC 5958 中所定義。

- 曲線規格

  識別指定私密金鑰中的橢圓曲線規格。寄件者和收件者的金鑰配對必須具有相同的曲線規格。

  有效值：`ECC_NIST_P256`、`ECC_NIS_P384`、`ECC_NIST_P512`

Java

下列範例會使用`PublicKeyDiscovery`金鑰合約結構描述建立 Raw ECDH 金鑰圈。此金鑰圈可以解密指定私密金鑰的公開金鑰與儲存在郵件加密文字上的收件者公開金鑰相符的任何郵件。

```
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);
}
```

多重 keyring

您可以結合 keyring 成為多重 keyring。多重 keyring 是一種 keyring，其中包含相同或不同類型的一或多個個別 keyring。效果就像是使用系列中的數個 keyring。使用多重 keyring 來加密資料時，其任何 keyring 中的任何包裝金鑰均可以解密該資料。

建立多重 keyring 來加密資料時，您會指定其中一個 keyring 做為產生器 keyring。所有其他 keyring 稱為子 keyring。產生器 keyring 會產生並加密純文字資料金鑰。然後，所有子 keyring 中的所有包裝金鑰會加密相同的純文字資料金鑰。該多重 keyring 會為多重 keyring 中的每個包裝金鑰傳回純文字金鑰和一個加密的資料金鑰。如果您建立沒有產生器 keyring 的多重 keyring，您可以使用它來解密資料，但無法加密。如果產生器金鑰圈是 KMS 金鑰圈，金鑰圈中的產生器金鑰會產生並加密 AWS KMS 純文字金鑰。然後，鑰匙 AWS KMS 圈 AWS KMS keys 中的所有附加密鑰以及多密鑰環中所有子密鑰環中的所有包裝密鑰，對相同的明文密鑰進行加密。

解密時， AWS Encryption SDK 會使用金鑰環嘗試解密其中一個加密的資料金鑰。按照在多重 keyring 中指定的順序呼叫 keyring。只要任何 keyring 中的任何金鑰可以解密已加密的資料金鑰，處理就會停止。

從 1.7 版開始。 x，當加密的資料金鑰在 AWS Key Management Service (AWS KMS) 金鑰環 (或主要金鑰提供者) 下加密時，一 AWS Encryption SDK 律會將的金鑰 ARN 傳遞 AWS KMS key 給 AWS KMS 解密作業的`KeyId`參數。這是 AWS KMS 最佳作法，可確保您使用要使用的包裝金鑰來解密加密的資料金鑰。

若要查看多重 keyring 的工作範例，請參閱：

- C：multi_keyring.cpp
- C #/ MultiKeyringExample.
- JavaScript Node.js: 多個按鍵環.
- JavaScript 瀏覽器：多鍵盤
- 爪哇 MultiKeyringExample

若要建立多重 keyring，請先將子 keyring 執行個體化。在此範例中，我們使用 AWS KMS 鑰匙圈和 Raw AES 鑰匙圈，但您可以將任何支援的鑰匙圈組合在多重鑰匙圈中。

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

```
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

```
// 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);
```

接著，建立多重 keyring，並指定其產生器 keyring (如果有)。在此範例中，我們建立了一個多鑰匙圈，其中 AWS KMS 鑰匙圈是產生器金鑰圈，而 AES 鑰匙圈則是子鑰匙圈。

C

在 C 中的多重 keyring 建構函數中，您只會指定其產生器 keyring。

```
struct aws_cryptosdk_keyring *multi_keyring = aws_cryptosdk_multi_keyring_new(alloc,
 kms_keyring);
```

若要將子 keyring 新增至您的多重 keyring，請使用 `aws_cryptosdk_multi_keyring_add_child` 方法。您需要為您新增的每個子 keyring 呼叫該方法一次。

```
// Add the Raw AES keyring (C only)
aws_cryptosdk_multi_keyring_add_child(multi_keyring, aes_keyring);
```

C# / .NET

.NET `CreateMultiKeyringInput` 建構函式可讓您定義產生器金鑰環和子金鑰環。生成的`CreateMultiKeyringInput`對象是不可變的。

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = kmsKeyring,
    ChildKeyrings = new List<IKeyring>() {aesKeyring}
};

var multiKeyring = materialProviders.CreateMultiKeyring(createMultiKeyringInput);
```

JavaScript Browser

JavaScript 多鑰匙圈是不可變的。 JavaScript 多鑰匙圈構造函數允許您指定生成器密鑰環和多個子密鑰環。

```
const clientProvider = getClient(KMS, { credentials })

const multiKeyring = new MultiKeyringWebCrypto(generator: kmsKeyring, children:
 [aesKeyring]);
```

JavaScript Node.js

JavaScript 多鑰匙圈是不可變的。 JavaScript 多鑰匙圈構造函數允許您指定生成器密鑰環和多個子密鑰環。

```
const multiKeyring = new MultiKeyringNode(generator: kmsKeyring, children:
 [aesKeyring]);
```

Java

Java `CreateMultiKeyringInput` 構造函數允許您定義生成器密鑰環和子密鑰環。生成的`createMultiKeyringInput`對象是不可變的。

```
final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
        .generator(awsKmsMrkMultiKeyring)
        .childKeyrings(Collections.singletonList(rawAesKeyring))
        .build();
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

現在，您可以使用多重 keyring 來加密和解密資料。

AWS Encryption SDK程式設計語言

AWS Encryption SDK 適用於下列程式設計語言。所有語言實作都是可互通的。您可以使用一種語言實作加密，並使用另一種語言進行解密。互通性可能受到語言限制。如果是這樣，這些限制會在語言實作的主題中加以說明。此外，加密和解密時，您必須使用相容的 Keyring，或主金鑰和主金鑰提供者。如需詳細資訊，請參閱the section called “Keyring 相容性”。

主題

- 適用於 C 的 AWS Encryption SDK
- AWS Encryption SDK適用於 .NET
- 適用於 JAVA 的 AWS Encryption SDK
- 適用於 JavaScript 的 AWS Encryption SDK
- 適用於 Python 的 AWS Encryption SDK
- AWS Encryption SDK 命令列界面

適用於 C 的 AWS Encryption SDK

適用於 C 的 AWS Encryption SDK 旨在提供用戶端加密程式庫給以 C 編寫應用程式的開發人員。它也可作為以更高階程式設計語言實作 AWS Encryption SDK的基礎。

與所有 AWS Encryption SDK實作相同，適用於 C 的 AWS Encryption SDK 提供進階的資料保護功能。這些功能包括信封加密、額外的驗證資料 (AAD) 以及安全、已認證的對稱金鑰演算法套件，例如 256 位元 AES-GCM 搭配金鑰衍生和簽署。

AWS Encryption SDK 的所有語言特定實作完全可以互通。例如，您可以使用適用於 C 的 AWS Encryption SDK，然後使用解密資料任何支持的語言實現，包括AWS加密 CLI。

所以此適用於 C 的 AWS Encryption SDK需要AWS SDK for C++來與互動AWS Key Management Service(AWS KMS。如果您使用的是可選AWS KMSKeyring。不過，AWS Encryption SDK 不需要 AWS KMS 或任何其他 AWS 服務。

進一步了解

- 如需相關詳細資訊適用於 C 的 AWS Encryption SDK，請參C 範例，例子中的aws-encryption-sdk-c 資料庫上GitHub，以及適用於 C 的 AWS Encryption SDKAPI 文件。

- 有關如何使用適用於 C 的 AWS Encryption SDK來加密數據，以便您可以將其解密為多個AWS 區域，請參如何在多個區域中使用AWS Encryption SDK適用於 C 的中的AWS安全部落格。

主題

- 安裝 適用於 C 的 AWS Encryption SDK
- 使用 適用於 C 的 AWS Encryption SDK
- 適用於 C 的 AWS Encryption SDK 範例

安裝 適用於 C 的 AWS Encryption SDK

安裝的最新版本適用於 C 的 AWS Encryption SDK。

> **Note**
>
> 所有版本適用於 C 的 AWS Encryption SDK2.0.0 之前的版本end-of-support階段。您可以安全地從 2.0 版更新。x以及更高版本的最新版本適用於 C 的 AWS Encryption SDK沒有任何代碼或數據更改。但是，新安全性功能2.0 版中引入。x不向後兼容。若要從 1.7 之前的版本進行更新。x為 2.0 版x之後，您必須首先更新到最新的 1。x的版本適用於 C 的 AWS Encryption SDK。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

您可以找到有關安裝和構建的詳細說明適用於 C 的 AWS Encryption SDK在自述文件的aws-encryption-sdk-c儲存庫。它包括在亞馬遜 Linux，Ubuntu 的，macOS 和視窗平台上構建的說明。

在開始之前，請決定您是否要使用AWS KMS鑰匙圈在AWS Encryption SDK。如果您使用AWS KMS鑰匙圈，您需要安裝AWS SDK for C++。所以此AWS需要 SDK 才能與之互動AWS Key Management Service(AWS KMS。當您使用AWS KMS鑰匙圈，AWS Encryption SDK使用AWS KMS生成和保護用於保護數據的加密密鑰。

您不需要安裝AWS SDK for C++如果您使用的是其他金鑰圈類型，例如原始 AES 鑰匙圈、原始 RSA 鑰匙圈或不包含AWS KMS鑰匙圈。但是，使用原始金鑰圈類型時，您需要產生並保護自己的原始包裝金鑰。

如需決定要使用哪種金鑰環類型的說明，請參閱the section called “選擇鑰匙圈”。

若您在安裝時發生問題，檔案問題在`aws-encryption-sdk-c`儲存庫或使用此頁面上的任何意見反應連結。

使用 適用於 C 的 AWS Encryption SDK

本主題說明在其他程式設計語言實作中不支援的一些 適用於 C 的 AWS Encryption SDK 功能。

本節中的範例展示如何使用2.0 版。x和後來的適用於 C 的 AWS Encryption SDK。有關使用早期版本的示例，請在版本的清單AWS 加密 sdk-c 存儲庫資 GitHub。

如需相關詳細資訊適用於 C 的 AWS Encryption SDK，請參C 範例，例子中的AWS 加密 sdk-c 存儲庫在 GitHub 上，並且適用於 C 的 AWS Encryption SDKAPI 文件。

另請參閱：使用 keyring

主題

- 加密和解密資料的模式
- 參考計數

加密和解密資料的模式

當您使用 適用於 C 的 AWS Encryption SDK 時，請依照類似這裡的模式：建立 keyring、建立使用 keyring 的 CMM、建立使用 CMM (和 keyring) 的工作階段，然後處理工作階段。

1. 載入錯誤字符串。

   呼叫aws_cryptosdk_load_error_strings()方法，請參考 C 或 C ++ 代碼。它加載對調試非常有用的錯誤信息。

   您只需調用一次，例如在main方法。

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

2. 建立 keyring。

   使用您想用來加密資料金鑰的包裝金鑰來設定 keyring。此範例使用AWS KMSKeyring，具備一個 AWS KMS key，但您可以改用任何類型的 keyring。

   要標識AWS KMS key在加密密鑰環中適用於 C 的 AWS Encryption SDK中，指定金鑰 ARN或者別名 ARN。在解密 Keyring 中，您必須使用金鑰 ARN。如需詳細資訊，請參閱 在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

```
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. 建立工作階段。

   在 適用於 C 的 AWS Encryption SDK 中，您可以使用工作階段來加密單一純文字訊息，或解密單一加密文字訊息，而無論其大小。此工作階段在整個處理過程中會維護訊息的狀態。

   使用分配器、keyring 和模式來設定您的工作階段：`AWS_CRYPTOSDK_ENCRYPT` 或 `AWS_CRYPTOSDK_DECRYPT`。如果您需要變更工作階段的模式，請使用 `aws_cryptosdk_session_reset` 方法。

   當您使用 keyring 建立工作階段時，適用於 C 的 AWS Encryption SDK自動為您建立預設的密碼編譯資料管理員 (CMM)。您不需要建立、維護或銷毀此物件。

   例如，以下工作階段使用在步驟 1 中定義的分配器和 keyring。當您加密資料時，模式為 `AWS_CRYPTOSDK_ENCRYPT`。

```
struct aws_cryptosdk_session * session =
 aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);
```

4. 加密或解密資料。

   若要在工作階段中處理資料，請使用 `aws_cryptosdk_session_process` 方法。如果輸入緩衝區夠大，足以容納整個純文字，並且輸出緩衝區夠大，足以容納整個密文字，則可以呼叫`aws_cryptosdk_session_process_full`。但是，如果您需要處理串流資料，您可以呼叫`aws_cryptosdk_session_process`在一個循環中。如需範例，請參閱 file_streaming.cpp 範例。所以此`aws_cryptosdk_session_process_full`引進於AWS Encryption SDK1.9 版。x和 2.2.x。

   當工作階段設定為加密資料時，純文字欄位描述輸入，加密文字欄位描述輸出。`plaintext` 欄位保留您想要加密的訊息，`ciphertext` 欄位取得加密方法所傳回的加密訊息。

```
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
                                   ciphertext,
                                   ciphertext_buffer_size,
                                   &ciphertext_length,
```

```
                                    plaintext,
                                    plaintext_length)
```

當工作階段設定為解密資料時，加密文字欄位描述輸入，純文字欄位描述輸出。ciphertext 欄位保留加密方法所傳回的已加密訊息，plaintext 欄位取得解密方法傳回的純文字訊息。

若要解密資料，請呼叫 aws_cryptosdk_session_process_full 方法。

```
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
                                    plaintext,
                                    plaintext_buffer_size,
                                    &plaintext_length,
                                    ciphertext,
                                    ciphertext_length)
```

參考計數

為了防止記憶體流失，當您使用完您建立的所有物件參考時，請務必將其釋出。否則會造成記憶體失流。此開發套件提供方法讓您輕鬆這樣做。

每當您使用下列其中一個子物件建立父物件時，父物件會取得並維護對子物件的參考，如下所示：

- keyring，例如，使用 keyring 建立工作階段
- 預設的密碼編譯資料管理員(CMM)，例如，使用預設 CMM 建立工作階段或自訂 CMM
- 資料金鑰快取，例如，使用 keyring 和快取建立快取 CMM

除非您需要對子物件的獨立參考，否則您可以在建立父物件後立即釋出對子物件的參考。在銷毀父物件時，對子物件的其餘參考即會釋出。此模式可確保只在您需要的一段時間內維護每個物件的參考，您不會因為參考未釋放而流失記憶體。

您只需負責釋出您明確建立的子物件參考。您不需負責管理 SDK 為您建立的任何物件的參考。如果 SDK 創建了一個對象（例如默認 CMM）aws_cryptosdk_caching_cmm_new_from_keyring方法，則 SDK 會管理物件及其參考的建立和銷毀。

在下列範例中，當您使用 keyring 建立工作階段，該工作階段會取得 keyring 的參考，並保留該參考，直到工作階段銷毀為止。如果您不需要保有 keyring 的其他參考，您可以使用

`aws_cryptosdk_keyring_release` 方法在建立工作階段後立即釋出 keyring 物件。此方法可遞減 keyring 的參考計數。當您呼叫 `aws_cryptosdk_session_destroy` 來銷毀工作階段時，將會釋出工作階段對 keyring 的參考。

```
// 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);
```

對於更複雜的工作，例如對多個工作階段重複使用 keyring 或在 CMM 中指定演算法套件，您可能需要維護物件的獨立參考。如果是這樣，請勿立即呼叫釋出方法。而是除了銷毀工作階段之外，當您不再使用這些物件時，請釋出您的參考。

當您使用備用 CMM 時，此引用計數技術也可以使用，例如資料金鑰快取。當您從快取和 keyring 建立快取 CMM 時，快取 CMM 會取得對這兩個物件的參考。除非您需要它們進行另一項任務，否則您可以在建立快取後立即釋出對快取和 keyring 的獨立參考。然後，當您使用快取 CMM 建立工作階段時，您可以釋出您對快取 CMM 的參考。

請注意，您只需負責釋出您明確建立的物件參考。方法為您建立的物件，例如快取底層的預設 CMM，是由方法管理。

```
/ 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);
```

適用於 C 的 AWS Encryption SDK 範例

以下範例說明如何使用 適用於 C 的 AWS Encryption SDK 來加密和解密資料。

本節中的範例會示範如何使用 2.0 版。x和後來的適用於 C 的 AWS Encryption SDK。有關使用早期版本的示例，請在版本的清單aws-encryption-sdk-c 儲存庫的儲存庫GitHub。

安裝和建置 適用於 C 的 AWS Encryption SDK，這些原始程式碼和其他範例的原始程式碼會包含在 `examples` 子目錄中，並且將它們編譯和建置到 `build` 目錄。您也可以在例子子目錄的aws-encryption-sdk-c的儲存庫GitHub。

主題

- 加密和解密字串

加密和解密字串

以下範例說明如何使用 適用於 C 的 AWS Encryption SDK 來加密和解密字串。

此示例的特點是AWS KMSKeyring，一種密鑰環類型，它使用AWS KMS key在AWS Key Management Service(AWS KMS)生成和加密數據密鑰。範例包括使用 C ++ 編寫的代碼。所以此適用於 C 的 AWS Encryption SDK需要AWS SDK for C++呼叫AWS KMS當您使用時，AWS KMSkeyring。如果您使用的密鑰環不與AWS KMS，例如原始 AES 密鑰環、原始 RSA 密鑰環或不包含AWS KMSkeyring，AWS SDK for C++不是必要項目。

有關創建AWS KMS key，請參建立金鑰中的AWS Key Management Service開發人員指南。為了幫助識別AWS KMS keys在AWS KMSkeyring，請參在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

請參閱完整的程式碼範例：string.cpp

主題

- 加密字串
- 解密字串

加密字串

此範例中的第一個部分會使用AWS KMS使用一個的 keyringAWS KMS key來加密明文字符串。

步驟 1. 載入錯誤字串。

呼叫`aws_cryptosdk_load_error_strings()`方法，請參考 C 或 C ++ 代碼。它加載對調試非常有用的錯誤信息。

您只需調用一次，例如在`main`方法。

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

步驟 2：建構 keyring。

建立AWS KMS用於加密的 keyring。本示例中的密鑰環配置為一個AWS KMS key，但是您可以配置AWS KMS使用多個的 keyringAWS KMS keys，包括AWS KMS keys在不同的AWS 區域和不同的帳户.

要標識AWS KMS key在加密密鑰環中適用於 C 的 AWS Encryption SDK中，指定金鑰 ARN或者別名 ARN。在解密 Keyring 中，您必須使用金鑰 ARN。如需詳細資訊，請參閱 在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

在 AWS KMS 鑰匙圈 AWS KMS keys 中識別

當您創建一個帶有多個AWS KMS keys中，您可以指定AWS KMS key用來生成和加密純文字資料金鑰，以及額外AWS KMS keys來加密相同純文字資料金鑰。在這種情況下，您只會指定生成器AWS KMS key。

執行此程式碼之前，請將範例金鑰 ARN 換成有效的金鑰 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);
```

步驟 3：建立工作階段。

使用分配器、模式列舉器和 keyring 來建立工作階段。

每個工作階段需要模式：`AWS_CRYPTOSDK_ENCRYPT` 用來加密或 `AWS_CRYPTOSDK_DECRYPT` 用來解密。若要變更現有工作階段的模式，請使用 `aws_cryptosdk_session_reset` 方法。

建立具有 keyring 的工作階段之後，您可以使用 SDK 提供的方法，將您的參考釋出給 keyring。工作階段會在其生命週期期間保留 keyring 物件的參考。當您銷毀工作階段時，會釋出 keyring 和工作階段物件的參考。此參考計數技術有助於避免記憶體流失，並避免在使用物件時釋出物件。

```
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);
```

步驟 4：設定加密內容。

加密內容是一種任意、非私密額外驗證資料。在加密時提供加密內容時，AWS Encryption SDK 會以密碼編譯方式將加密內容繫結至加密文字，使得解密資料會需要相同的加密內容。使用加密內容是選用的，但我們建議使用它作為最佳實務。

先建立包含加密內容字串的雜湊表格。

```
/* 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)
```

取得工作階段中加密內容的可變指標。然後，使用 `aws_cryptosdk_enc_ctx_clone` 函數來將加密內容複製到工作階段。將複本放在 `my_enc_ctx` 中，使得您可以在解密資料之後驗證其值。

加密內容是工作階段的一部分，而非傳遞到工作階段處理函數的參數。這可保證將相同加密內容用於訊息的每個區段，即使呼叫了工作階段處理函數多次來加密整個訊息亦然。

```
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)
```

步驟 5：加密字串。

若要加密純文字字串，請使用 aws_cryptosdk_session_process_full 方法搭配加密模式的工作階段。這種方法，在AWS Encryption SDK1.9 版本。x和 2.2.x，專為非流式加密和解密而設計。要處理流數據，請調用aws_cryptosdk_session_process在一個循環中。

加密時，純文字欄位為輸入欄位；加密文字欄位為輸出欄位。當處理完成時，ciphertext_output 欄位會包含加密的訊息，包括實際加密文字、加密的資料金鑰和加密內容。您可以在任何支援的程式設計語言中使用 AWS Encryption SDK 來將此加密的訊息解密。

```
/* 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;
}
```

步驟 6：清除工作階段。

最後一步會銷毀工作階段，包括對 CMM 和 keyring 的參考。

如果您偏好不要銷毀工作階段，則可以對工作階段重複使用相同的 keyring 和 CMM 來解密字串，或是加密或解密其他訊息。若要將工作階段用於解密，請使用 aws_cryptosdk_session_reset 方法來將模式變更為 AWS_CRYPTOSDK_DECRYPT。

解密字串

此範例的第二個部分會將包含原始字串之加密文字的加密訊息解密。

步驟 1：載入錯誤字串。

呼叫aws_cryptosdk_load_error_strings()方法，請參考 C 或 C ++ 代碼。它加載對調試非常有用的錯誤信息。

您只需調用一次，例如在main方法。

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

步驟 2：建構 keyring。

在 AWS KMS 中解密資料時，您會傳入加密 API 傳回的加密的訊息。所以此解密 API不採用AWS KMS key作為輸入。相反地，AWS KMS 會使用相同的 AWS KMS key 來解密加密所用的加密文字。但是，AWS Encryption SDK可讓您指定AWS KMSkeyringAWS KMS keys的加密和解密。

在解密時，您可以設定讓 keyring 只具有您要用來解密已加密訊息的 AWS KMS keys。例如，您可能想要建立只具有組織中特定角色所使用 AWS KMS key 的 keyring。所以此AWS Encryption SDK將永遠不會使用AWS KMS key除非它出現在解密 Keyring 中。如果 SDK 無法使用您提供的 keyring 中的 AWS KMS keys 來解密加密的資料金鑰，可能是因為 keyring 中的任一個 AWS KMS keys 均未用來加密任何資料金鑰，或因為任何發起人沒有許可可使用 keyring 中的 AWS KMS keys 來解密，而造成解密呼叫失敗。

當您指定AWS KMS key的解密 Keyring，則必須使用其金鑰 ARN。別名 ARN僅允許在加密 keyring 中。為了幫助識別AWS KMS keys在AWS KMSkeyring，請參在 AWS KMS 鑰匙圈 AWS KMS keys 中識別。

在此範例中，我們會指定一個 keyring，其中設定了AWS KMS key用來加密字串。執行此程式碼之前，請將範例金鑰 ARN 換成有效的金鑰 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);
```

步驟 3：建立工作階段。

使用分配器和 keyring 來建立工作階段。若要設定用於解密的工作階段，請使用 AWS_CRYPTOSDK_DECRYPT 模式設定工作階段。

建立具有 keyring 的工作階段之後，您可以使用 SDK 提供的方法，將您的參考釋出給 keyring。工作階段會在其生命週期期間保留對 keyring 物件的參考，當您銷毀工作階段時，工作階段和 keyring 都會釋出。此參考計數技術有助於避免記憶體流失，並避免在使用物件時釋出物件。

```
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);
```

步驟 4：解密字串。

若要解密字串，請使用 `aws_cryptosdk_session_process_full` 方法搭配設定用於解密的工作階段。這種方法，在AWS Encryption SDK1.9 版本。x和 2.2.x，專為非流式加密和解密而設計。要處理流數據，請調用`aws_cryptosdk_session_process`在一個循環中。

解密時，加密文字欄位為輸入欄位，而純文字欄位為輸出欄位。`ciphertext_input` 欄位會保存加密方法傳回的加密的訊息。當處理完成時，`plaintext_output` 欄位會包含純文字 (解密的) 字串。

```
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;
}
```

步驟 5：驗證加密內容。

請確定實際加密內容（用來解密訊息的內容）包含加密訊息時您提供的加密內容。實際加密內容可能包含額外配對，因為密碼編譯資料管理員 (CMM) 可以在加密訊息之前，將配對新增到提供的加密內容。

在 適用於 C 的 AWS Encryption SDK 中，解密時您不需要提供加密內容，因為加密內容包含在 SDK 傳回的加密訊息中。但是，在它傳回純文字訊息之前，您的解密函數應該驗證提供的解密內容中的所有配對會出現在解密訊息所用的加密內容中。

首先，取得工作階段中雜湊表格的唯讀指標。此雜湊表格中包含解密訊息所用的加密內容。

```
const struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr(session);
```

然後循環回應您在加密時複製的 my_enc_ctx 雜湊表格中的加密內容。驗證用來解密的 my_enc_ctx 雜湊表格中的每個配對顯示在解密所用的 session_enc_ctx 雜湊表格中。如果有任何金鑰遺漏，或是該金鑰有不同的值，便停止處理並寫入錯誤訊息。

```
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();
    }
}
```

步驟 6：清除工作階段。

驗證加密內容之後，您可以銷毀工作階段，或是將其重複使用。如果您需要重新配置工作階段，請使用aws_cryptosdk_session_reset方法。

```
aws_cryptosdk_session_destroy(session);
```

AWS Encryption SDK適用於 .NET

For .NET 是AWS Encryption SDK用戶端加密程式庫，適用於使用 C# 和其他 .NET 程式設計語言撰寫應用程式的開發人員。Windows、macOS 和 Linux 都提供支援。

的所有程式設計語言實作AWS Encryption SDK都是完全可互通的。但是，如果您使用第 4 版中所需的加密內容 CMM 來加密資料。 AWS Encryption SDK對於 .NET 的 x，您只能使用版本 4 解密它。AWS Encryption SDK適用於 .NET 或版本 3 的 x。 的 x 的適用於 JAVA 的 AWS Encryption SDK。

> **Note**
>
> AWS Encryption SDK適用於 .NET 的 4.0.0 版本不符合「AWS Encryption SDK訊息規格」。因此，由 4.0.0 版加密的訊息只能透過 .NET 的 4.0.0 版或更新版本解密。AWS Encryption SDK它們不能被任何其他編程語言實現解密。
> 的 4.0.1 版 F AWS Encryption SDK OR .NET 會根據AWS Encryption SDK訊息規格寫入訊息，並且可與其他程式設計語言實作互通。根據預設，版本 4.0.1 可以讀取 4.0.0 版加密的郵件。不過，如果您不想解密以 4.0.0 版加密的郵件，您可以指定`NetV4_0_0_RetryPolicy`屬性以防止用戶端讀取這些訊息。如需詳細資訊，請參閱的存放 aws-encryption-sdk-dafny 庫中的 v4.0.1 版本說明。 GitHub

For .NET 與其他一些程式設計語言實作的不同之處AWS Encryption SDK在AWS Encryption SDK於下列方面：

- 不支援資料金鑰快取

> **Note**
>
> 版本 4. x 的 .NET AWS Encryption SDK 支援AWS KMS階層式金鑰圈，這是一種替代的加密材料快取解決方案。

- 不支援串流資料
- 沒有來自 .NET 的AWS Encryption SDK日誌記錄或堆棧跟踪
- 需要 AWS SDK for .NET

AWS Encryption SDK適用於 .NET 包含 2.0 版中引入的所有安全性功能。 x 及更新版本的其他語言實作AWS Encryption SDK。但是，如果您使用AWS Encryption SDK的是 .NET 來解密 2.0 之前版本加密的資料。 x 版本的另一種語言實作AWS Encryption SDK，您可能需要調整承諾政策。如需詳細資訊，請參閱 如何設定承諾政策。

.NET 是 Dafny AWS Encryption SDK 中的產品，這是一種正式的驗證語言，您可以在其中編寫規格，實現它們的代碼以及AWS Encryption SDK用於測試它們的證明。結果是在框架AWS Encryption SDK中實現的功能庫，以確保功能正確性。

進一步了解

- 如需示範如何在中設定選項的範例AWS Encryption SDK，例如指定替代演算法套件、限制加密的資料金鑰，以及使用AWS KMS多區域金鑰，請參閱設定AWS Encryption SDK。
- 如需有關使用 FOR .NET AWS Encryption SDK 進行程式設計的詳細資訊，請參閱上的 aws-encryption-sdk-dafny存放庫`aws-encryption-sdk-net`目錄 GitHub。

主題

- 安裝AWS Encryption SDK適用於 .NET 的
- AWS Encryption SDK針對. NET 進行偵錯
- AWS KMSAWS Encryption SDK適用於 .NET 的中的鑰匙圈
- 4.x 版中必要的加密內容
- AWS Encryption SDK針對 .NET 範例

安裝AWS Encryption SDK適用於 .NET 的

AWS Encryption SDK適用於 .NET 可作為中的`AWS.Cryptography.EncryptionSDK`套件使用 NuGet。如需有關安裝和建置 .NET 的AWS Encryption SDK詳細資訊，請參閱儲存庫中的 README.md 檔案。`aws-encryption-sdk-net`

版本 3.x

版本 3. 對於. NET AWS Encryption SDK 的 x 支持. NET 框架 4.5.2-4.8 僅在視窗上。它在所有支持的操作系統上支持 .NET 核心 3.0 + 和 .NET 5.0 及更高版本。

版本 4.x

版本 4. AWS Encryption SDK適用於 .NET 的 x 支援 .NET 6.0 及更新版本。

AWS SDK for .NET即使AWS Encryption SDK您沒有使用 AWS Key Management Service (AWS KMS) 鍵，.NET 也需要。它會隨 NuGet 套件一起安裝。不過，除非您使用AWS KMS金鑰，AWS Encryption SDK否則 .NET 不需要AWS 帳戶、AWS認證或與任何AWS服務互動。如需設定AWS帳戶的說明，請參閱搭配 AWS KMS 使用 AWS Encryption SDK。

AWS Encryption SDK針對. NET 進行偵錯

.NET 不會產生任何記錄檔。AWS Encryption SDK.NET 中的AWS Encryption SDK例外會產生例外狀況訊息，但不會產生堆疊追蹤。

若要協助您偵錯，請務必啟用AWS SDK for .NET. 來自的記錄和錯誤訊息AWS SDK for .NET可協助您區分AWS SDK for .NET發生的錯誤與 .NET 中AWS Encryption SDK的錯誤。如需有關AWS SDK for .NET記錄的說明，請參閱AWSLogging開AWS SDK for .NET發人員指南中的。若要查看主題，請展開 [開啟以檢視 .NET 架構內容] 區段。)

AWS KMSAWS Encryption SDK適用於 .NET 的中的鑰匙圈

AWS Encryption SDK適用於 .NET 的基本金AWS KMS鑰圈只會使用一個 KMS 金鑰。他們還需要一個AWS KMS客戶端，這使您有機會為 KMS 密鑰配置客戶端。AWS 區域

要使用一個或多個包裝鑰匙來創建鑰匙AWS KMS圈，請使用多鑰匙圈。For .NET 有一個特殊的多鑰匙圈，它需要一個或多個AWS KMS密鑰，以及一個標準的多鑰匙圈，它需要任何支持類型的一個或多個鑰匙圈。AWS Encryption SDK一些程序員更喜歡使用多鑰匙圈方法來創建他們的所有鑰匙圈，並且用AWS Encryption SDK於 .NET 支持該策略。

For .NET AWS Encryption SDK 為所有典型使用案例提供基本的單一金鑰環和多重金鑰環，包括多區域金鑰。AWS KMS

例如，若要使用一個AWS KMS金鑰建立金鑰AWS KMS圈，您可以使用此方`CreateAwsKmsKeyring()`法。

Version 3.x

下列範例使用版本 3。 用AWS Encryption SDK於 .NET 的 x，為包含指定金鑰的區域建立預設AWS KMS用戶端。

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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
};
```

```
// Create the keyring
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

Version 4.x

下列範例使用版本 4。 用AWS Encryption SDK於 .NET 的 x 為包含指定鍵的區域創建一個AWS KMS客戶端。

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

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

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

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

若要使用一個或多個金鑰建立AWS KMS金鑰圈，請使用方`CreateAwsKmsMultiKeyring()`法。此範例使用兩個AWS KMS金鑰。若要指定一個 KMS 金鑰，請僅使用`Generator`參數。指定其他 KMS 金鑰的`KmsKeyIds`參數是選擇性的。

此金鑰圈的輸入不會接受AWS KMS用戶端。而是針對金鑰圈中以 KMS 金鑰表示的每個區域AWS Encryption SDK使用預設用AWS KMS戶端。例如，如果`Generator`參數值所識別的 KMS 金鑰位於美國西部 (奧勒岡) 區域 (`us-west-2`)，則會為該區`us-west-2`域AWS Encryption SDK建立預設AWS KMS用戶端。如果您需要自定義AWS KMS客戶端，請使用該`CreateAwsKmsKeyring()`方法。

下列範例使用版本 4。 用AWS Encryption SDK於 .NET 的 x 和自訂用AWS KMS戶端的`CreateAwsKmsKeyring()`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 =
 materialProviders.CreateAwsKmsMultiKeyring(createEncryptKeyringInput);
```

版本 4. .NET 的 AWS Encryption SDKx 支援使用對稱加密 (SYMMETRIC_DEFAULT) 或非對稱 RSA KMS AWS KMS 金鑰的金鑰圈。 AWS KMS使用非對稱 RSA KMS 金鑰建立的金鑰圈只能包含一個 key pair。

若要使用非對稱 RSA 金AWS KMS鑰圈加密，您不需要 kms: GenerateDataKey 或 KMS: Encrypt，因為您必須在建立金鑰環時指定要用於加密的公開金鑰材料。使用此金鑰圈加密時，不會進行任何AWS KMS呼叫。若要使用非對稱 RSA AWS KMS 金鑰圈進行解密，您需要 K MS: 解密權限。

若要建立非對稱 RSA 金AWS KMS鑰圈，您必須從非對稱 RSA KMS 金鑰提供公開金鑰和私密金鑰 ARN。公開金鑰必須是 PEM 編碼。下列範例會建立具有非對稱 RSA key pair AWS KMS 圈。

```
// 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);
```

4.x 版中必要的加密內容

使用版本 4。 AWS Encryption SDK對於 .NET，您可以使用必要的加密內容 CMM 來在您的密碼編譯作業中要求加密內容的 x。加密內容是一組非秘密金鑰值配對。加密內容會以密碼編譯方式繫結至加密的資料，因此必須使用相同的加密內容來解密欄位。使用必要的加密內容 CMM 時，您可以指定一或多個必要的加密內容金鑰 (必要金鑰)，這些金鑰必須包含在所有加密和解密呼叫中。

> **Note**
>
> 必要的加密內容 CMM 只能與版本 3 互通。 的 x 的適用於 JAVA 的 AWS Encryption SDK。它不能與任何其他編程語言實現互操作。如果您使用必要的加密內容 CMM 加密資料，則只能使用版本 3 來解密資料。 適用於 JAVA 的 AWS Encryption SDK或版本 4 的 x。 AWS Encryption SDK用於 .NET 的 x。

在加密時，AWS Encryption SDK會驗證所有必要的加密內容金鑰都包含在您指定的加密內容中。會AWS Encryption SDK簽署您指定的加密環境。只有非必要金鑰的索引鍵值配對才會以純文字形式儲存在加密作業傳回的加密訊息標頭中。

解密時，您必須提供一個加密內容，其中包含代表所需金鑰的所有金鑰-值配對。會AWS Encryption SDK使用此加密內容和儲存在加密郵件標頭中的金鑰值配對來重建您在加密作業中指定的原始加密內容。如果AWS Encryption SDK無法重建原始加密內容，則解密作業會失敗。如果您提供的金鑰-值組包含所需金鑰且值不正確，則無法解密加密的訊息。您必須提供與加密時指定的相同鍵值組。

> **Important**
>
> 請仔細考慮您為加密內容中的必要金鑰選擇哪些值。您必須能夠在解密時再次提供相同的金鑰及其對應的值。如果您無法重現所需的金鑰，則無法解密加密的訊息。

下列範例會使用必要的加密內容 CMM 初始化AWS KMS金鑰環。

```
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);
```

如果您使用AWS KMS金鑰環，用AWS Encryption SDK於 .NET 也會使用加密內容，在金鑰環進行的呼叫中提供額外的已驗證資料 (AAD)。AWS KMS

AWS Encryption SDK針對 .NET 範例

下列範例顯示使用 of .NET AWS Encryption SDK 進行程式設計時所使用的基本編碼模式。具體來說，您要具現化AWS Encryption SDK和材料提供者資源庫。然後，在呼叫每個方法之前，您可以具現化一個定義方法輸入的物件。這很像您在中使用的編碼模式AWS SDK for .NET。

如需示範如何在中設定選項的範例AWS Encryption SDK，例如指定替代演算法套件、限制加密的資料金鑰，以及使用AWS KMS多區域金鑰，請參閱設定AWS Encryption SDK。

如需使用 FOR .NET AWS Encryption SDK 進行程式設計的更多範例，請參閱上`aws-encryption-sdk-dafny`存放庫`aws-encryption-sdk-net`目錄中的範例 GitHub。

AWS Encryption SDK為 .NET 加密中的資料

此範例顯示加密資料的基本模式。它使用由一個AWS KMS包裝密鑰保護的數據密鑰對一個小文件進行加密。

步驟 1：實例化AWS Encryption SDK和材料提供者資源庫。

首先實例化AWS Encryption SDK和材料提供者資源庫。您將使用中的方法AWS Encryption SDK來加密和解密資料。您將使用材料提供者資料庫中的方法來建立金鑰環，以指定哪些金鑰保護您的資料。

在版本 3 之間，實例化AWS Encryption SDK和材料提供者資源庫的方式不同。 x 和 4. AWS Encryption SDK用於 .NET 的 x。以下所有步驟對於兩個版本 3 都是相同的。 x 和 4. AWS Encryption SDK用於 .NET 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());
```

步驟 2：為鑰匙圈創建一個輸入對象。

每個建立金鑰環的方法都有對應的輸入物件類別。例如，若要建立方`CreateAwsKmsKeyring()`法的輸入物件，請建立`CreateAwsKmsKeyringInput`類別的實體。

即使此金鑰環的輸入未指定產生器金鑰，`KmsKeyId`參數所指定的單一 KMS 金鑰就是產生器金鑰。它會產生並加密加密資料的資料金鑰。

此輸入物件需要 KMS 金鑰AWS 區域的AWS KMS用戶端。若要建立AWS KMS用戶端，請在中實例化`AmazonKeyManagementServiceClient`類別。AWS SDK for .NET呼叫沒有參數的建`AmazonKeyManagementServiceClient()`構函式會建立具有預設值的用戶端。

在用於為 .NET 加密的金AWS KMS鑰環中AWS Encryption SDK，您可以使用金鑰識別碼、金鑰 ARN、別名或別名 ARN 來識別 KMS 金鑰。在用於解密的金AWS KMS鑰圈中，您必須使用金鑰 ARN 來識別每個 KMS 金鑰。如果您打算重複使用加密金鑰圈進行解密，請為所有 KMS 金鑰使用金鑰 ARN 識別碼。

```
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
};
```

步驟 3：創建鑰匙圈。

若要建立金鑰圈，請使用金鑰圈輸入物件呼叫 keyring 方法。此範例使用的`CreateAwsKmsKeyring()`方法只需要一個 KMS 金鑰。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

步驟 4：定義加密內容。

加密內容是選用的，但強烈建議您在AWS Encryption SDK. 您可以定義一或多個非秘密金鑰值配對。

> **Note**
>
> 使用版本 4。 AWS Encryption SDK對於 .NET，您可以在具有必要加密內容 CMM 的所有加密請求中需要加密內容。

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

```
};
```

第 5 步：創建用於加密的輸入對象。

在呼叫Encrypt()方法之前，請先建立EncryptInput類別的執行個體。

```
string plaintext = File.ReadAllText("C:\\Documents\\CryptoTest\\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
```

第 6 步：加密明文。

使用的Encrypt()方法，使用您定義的金鑰環AWS Encryption SDK來加密純文字。

EncryptOutput該Encrypt() 方法返回的具有獲取加密消息（Ciphertext），加密上下文和算法套件的方法。

```
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

步驟 7：獲取加密的消息。

.NET 中的Decrypt()方法會接受EncryptOutput執行個體的Ciphertext成員。AWS Encryption SDK

EncryptOutput物件的Ciphertext成員是加密訊息，這是一種可攜式物件，其中包含加密資料、加密資料金鑰和中繼資料 (包括加密內容)。您可以長時間安全地儲存加密的郵件，或將其提交給復原純文字的Decrypt()方法。

```
var encryptedMessage = encryptOutput.Ciphertext;
```

在 .NET 中以嚴格模式AWS Encryption SDK解密

最佳作法建議您指定用來解密資料的金鑰，這是一種稱為嚴格模式的選項。只會AWS Encryption SDK使用您在金鑰環中指定的 KMS 金鑰來解密密文。解密金鑰圈中的金鑰必須至少包含一個加密資料的金鑰。

此範例顯示了在嚴謹模式下使用 .NET 進行解密的基本模式。AWS Encryption SDK

步驟 1：具現化AWS Encryption SDK和材料提供者資源庫。

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

步驟 2：為您的鑰匙圈創建輸入對象。

若要指定金鑰圈方法的參數，請建立輸入物件。.NET 中的每個金鑰圈方AWS Encryption SDK法都有對應的輸入物件。因為這個範例會使用`CreateAwsKmsKeyring()`方法來建立 keyring，所以它會實體化輸入的`CreateAwsKmsKeyringInput`類別。

在解密金鑰圈中，您必須使用金鑰 ARN 來識別 KMS 金鑰。

```
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
};
```

步驟 3：創建鑰匙圈。

若要建立解密金鑰環，此範例會使用方`CreateAwsKmsKeyring()`法和金鑰環輸入物件。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

步驟 4：創建用於解密的輸入對象。

若要建立方`Decrypt()`法的輸入物件，請實體化`DecryptInput`類別。

`DecryptInput()`建構函式的`Ciphertext`參數`Ciphertext`會接受`Encrypt()`方法傳回之`EncryptOutput`物件的成員。`Ciphertext`屬性代表加密的郵件，其中包括加密的資料、加密的資料金鑰和解密訊息所AWS Encryption SDK需的中繼資料。

使用版本 4。 x 的 AWS Encryption SDK .NET，您可以使用可選`EncryptionContext`參數在`Decrypt()`方法中指定您的加密內容。

使用`EncryptionContext`參數可驗證加密時使用的加密內容是否包含在用來解密密文的加密內容中。AWS Encryption SDK將配對添加到加密上下文中，包括數字簽名（如果您使用帶簽名的算法套件），例如默認算法套件。

```
var encryptedMessage = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

步驟 5：解密密文。

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

步驟 6：驗證加密內容-版本 3。 x

版本 3 的`Decrypt()`方法。 .NET AWS Encryption SDK 的 x 不會採用加密內容。它會從加密郵件中的中繼資料取得加密內容值。不過，在傳回或使用純文字之前，最佳做法是確認用來解密密文的加密內容是否包含您在加密時提供的加密內容。

確認加密時使用的加密內容包含在用來解密密文的加密內容中。AWS Encryption SDK將配對添加到加密上下文中，包括數字簽名（如果您使用帶簽名的算法套件），例如默認算法套件。

```
// 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");
}
```

使用 .NET 中的探索金鑰圈進行AWS Encryption SDK解密

除了指定用於解密的 KMS 金鑰外，您還可以提供AWS KMS探索金鑰環，此金鑰圈是不會指定任何 KMS 金鑰的金鑰環。探索金鑰圈可讓您使用加AWS Encryption SDK密資料的任何 KMS 金鑰來解密資

料，前提是呼叫者具有金鑰的解密權限。如需最佳做法，請新增探索篩選器，以限制可用於特定磁碟分割AWS 帳戶的 KMS 金鑰。

For .NET 提供了一個基本的探索金鑰圈，該金鑰圈需要AWS KMS用戶端和探索多重金鑰環，需要您指定一或多個。AWS Encryption SDK AWS 區域用戶端和區域都會限制可用來解密加密郵件的 KMS 金鑰。兩個金鑰圈的輸入物件均採用建議的探索篩選器。

下列範例顯示使用AWS KMS探索金鑰圈和探索篩選器解密資料的模式。

步驟 1：實例化AWS Encryption SDK和材料提供者資源庫。

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

步驟 2：為鑰匙圈創建輸入對象。

若要指定金鑰圈方法的參數，請建立輸入物件。.NET 中的每個金鑰圈方AWS Encryption SDK法都有對應的輸入物件。因為這個範例會使用`CreateAwsKmsDiscoveryKeyring()`方法來建立 keyring，所以它會實體化輸入的`CreateAwsKmsDiscoveryKeyringInput`類別。

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

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
    }
};
```

步驟 3：創建鑰匙圈。

若要建立解密金鑰環，此範例會使用方`CreateAwsKmsDiscoveryKeyring()`法和金鑰環輸入物件。

```
var discoveryKeyring =
 materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

步驟 4：創建用於解密的輸入對象。

若要建立方Decrypt()法的輸入物件，請實體化DecryptInput類別。Ciphertext參數的值是方Encrypt()法傳回之EncryptOutput物件的Ciphertext成員。

使用版本 4。 x 的 AWS Encryption SDK .NET，您可以使用可選EncryptionContext參數在Decrypt()方法中指定您的加密內容。

使用此EncryptionContext參數可驗證加密時使用的加密內容是否包含在用來解密密文的加密內容中。AWS Encryption SDK將配對添加到加密上下文中，包括數字簽名（如果您使用帶簽名的算法套件），例如默認算法套件。

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = discoveryKeyring,
    EncryptionContext = encryptionContext // OPTIONAL

};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

步驟 5：驗證加密內容-版本 3。 x

版本 3 的Decrypt()方法。 .NET AWS Encryption SDK 的 x 不會在上接受加密上下文Decrypt()。它會從加密郵件中的中繼資料取得加密內容值。不過，在傳回或使用純文字之前，最佳做法是確認用來解密密文的加密內容是否包含您在加密時提供的加密內容。

確認用於加密的加密內容包含在用來解密密文的加密內容中。AWS Encryption SDK將配對添加到加密上下文中，包括數字簽名（如果您使用帶簽名的算法套件），例如默認算法套件。

```
// 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");
}
```

適用於 JAVA 的 AWS Encryption SDK

本主題說明如何安裝及使用 適用於 JAVA 的 AWS Encryption SDK。如需有關使用程式設計的詳細資訊適用於 JAVA 的 AWS Encryption SDK，請參閱上的aws-encryption-sdk-java存放庫 GitHub。如需 API 文件，請參閱. 適用於 JAVA 的 AWS Encryption SDK

主題

- 先決條件
- 安裝
- AWS KMS鑰匙圈中 適用於 JAVA 的 AWS Encryption SDK
- 3.x 版中必要的加密內容
- 適用於 JAVA 的 AWS Encryption SDK 範例

先決條件

安裝 適用於 JAVA 的 AWS Encryption SDK 之前，請確認您是否具備下列必要項目。

Java 開發環境

您會需要 Java 8 或更新版本。在 Oracle 網站上，移至 Java SE 下載，然後下載並安裝 Java SE 開發套件 (JDK)。

如果您使用 Oracle JDK，您還必須下載並安裝 Java Cryptography Extension (JCE) Unlimited Strength 管轄權政策檔案。

Bouncy Castle

適用於 JAVA 的 AWS Encryption SDK 需要 Bouncy Castle。

- 適用於 JAVA 的 AWS Encryption SDK 版本 1.6.1 和更新版本使用 Bouncy Castle 來序列化和還原序列化密碼編譯物件。您可以使用 Bouncy Castle 或 Bouncy Castle FIPS 來滿足此要求。如需安裝和設定 Bouncy Castle FIPS 的說明，請參閱 BC FIPS 文件，尤其是使用者指南和安全性原則 PDF。
- 適用於 JAVA 的 AWS Encryption SDK 的早期版本使用 Bouncy Castle 的 Java 適用的密碼編譯 API。只有非 FIPS Bouncy Castle 才能滿足此要求。

如果您沒有 Bouncy Castle，請移至 Bouncy Castle 最新版本下載與您 JDK 相對應的提供者檔案。您也可以使用阿帕奇 Maven 獲取標準充氣城堡提供商（bcprov-ext-jdk15 上）或充氣城堡 FIPS（bc-fips）的神器的神器。

AWS SDK for Java

版本 3. x 的適用於 JAVA 的 AWS Encryption SDK需要AWS SDK for Java 2.x，即使您不使用AWS KMS鑰匙圈。

版本 2. x 或更早版本的適用於 JAVA 的 AWS Encryption SDK不需要AWS SDK for Java. 不過，需 AWS SDK for Java要使用 AWS Key Management Service(AWS KMS) 做為主要金鑰提供者。從適用於 JAVA 的 AWS Encryption SDK版本 2.4.0 開始，適用於 JAVA 的 AWS Encryption SDK支援的. AWS SDK for Java AWS Encryption SDKAWS SDK for Java1.x 和 2.x 的代碼是可互操作的。例如，您可以使用支援 AWS SDK for Java 1.x 的AWS Encryption SDK程式碼加密資料，並使用支援的程式碼解密資料 AWS SDK for Java 2.x (反之亦然)。適用於 JAVA 的 AWS Encryption SDK早於 2.4.0 的版本僅支援 AWS SDK for Java 1.x。如需更新您的版本的詳細資訊AWS Encryption SDK，請參閱遷移您的AWS Encryption SDK。

將適用於 JAVA 的 AWS Encryption SDK程式碼從 AWS SDK for Java 1.x 更新為時AWS SDK for Java 2.x，請使用中`AWSKMS`介面的參照來取代 AWS SDK for Java 1.x 中`KmsClient`介面的參照。AWS SDK for Java 2.x適用於 JAVA 的 AWS Encryption SDK不支援`KmsAsyncClient`介面。此外，請更新程式碼以使用命`kmssdkv2`名空間中的AWS KMS相關物件，而非`kms`命名空間。

若要安裝 AWS SDK for Java，請使用 Apache Maven。

- 若要匯入整個 AWS SDK for Java 作為相依性，請在 `pom.xml` 檔案中宣告它。
- 若要僅為 AWS SDK for Java 1.x 中的AWS KMS模組建立相依性，請遵循指定特定模組的指示，並將設定`artifactId`為`aws-java-sdk-kms`。
- 要僅為 AWS SDK for Java 2.x 中的AWS KMS模塊創建依賴關係，請按照指定特定模塊的說明進行操作。`groupId`將設定為`software.amazon.awssdk`和`artifactId`為`kms`。

如需更多變更，請參閱AWS SDK for Java 2.x開發人員指南中的 AWS SDK for Java 1.x 與 2.x 之間有何不同。

AWS Encryption SDK開發人員指南中的 Java 範例使用AWS SDK for Java 2.x.

安裝

安裝最新版本的適用於 JAVA 的 AWS Encryption SDK.

> **Note**
>
> 適用於 JAVA 的 AWS Encryption SDK早於 2.0.0 的所有版本都處於此end-of-support階段。

您可以安全地從 2.0 版更新。 x 及更新版本的最新版本，適用於 JAVA 的 AWS Encryption SDK無需任何代碼或數據更改。但是，2.0 版中引入了新的安全功能。 x 不向後相容。若要從 1.7 之前的版本進行更新。 x 轉換為 2.0 版本。 x 和更新版本，您必須先更新到最新的 1。 x 版本的AWS Encryption SDK. 如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

您可以使用下列方式來安裝 適用於 JAVA 的 AWS Encryption SDK。

手動

若要安裝適用於 JAVA 的 AWS Encryption SDK，請複製或下載存aws-encryption-sdk-java GitHub 放庫。

使用 Apache Maven

適用於 JAVA 的 AWS Encryption SDK 可透過具有下列依存性定義的 Apache Maven 取得。

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>3.0.0</version>
</dependency>
```

安裝 SDK 之後，請參閱本指南中的範例 Java 程式碼以及 Javadoc 開始使用。 GitHub

AWS KMS鑰匙圈中 適用於 JAVA 的 AWS Encryption SDK

版本 3. x 的適用於 JAVA 的 AWS Encryption SDK使用金鑰圈來執行信封加密。在適用於 JAVA 的 AWS Encryption SDK採取只有一個 KMS 密鑰的基本鑰AWS KMS匙圈。他們還需要一個AWS KMS客戶端，這使您有機會為 KMS 密鑰配置客戶端。AWS 區域

要使用一個或多個包裝鑰匙來創建鑰匙AWS KMS圈，請使用多鑰匙圈。適用於 JAVA 的 AWS Encryption SDK具有一個特殊的多鑰匙圈，可以接受一個或多個AWS KMS鑰匙，以及一個標準的多鑰匙圈，可以使用任何支持類型的一個或多個鑰匙圈。一些程序員更喜歡使用多鑰匙圈方法來創建他們的所有鑰匙圈，並適用於 JAVA 的 AWS Encryption SDK支持該策略。

為所有典型使用案例 (包括多區域金鑰) 適用於 JAVA 的 AWS Encryption SDK 提供基本的單一按鍵鑰匙圈和多重金鑰環。AWS KMS

例如，若要使用一個AWS KMS金鑰建立金鑰AWS KMS圈，您可以使用`CreateAwsKmsKeyring()`]方法。

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

// Create the keyring
CreateAwsKmsKeyringInput kmsKeyringInput = CreateAwsKmsKeyringInput.builder()
        .kmsKeyId(keyArn)
        .kmsClient(KmsClient.create())
        .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

若要使用一個或多個金鑰建立AWS KMS金鑰圈，請使用方`CreateAwsKmsMultiKeyring()`法。此範例使用兩個 KMS 金鑰。若要指定一個 KMS 金鑰，請僅使用`generator`參數。指定其他 KMS 金鑰的`msKeyIds`參數是選擇性的。

此金鑰圈的輸入不會接受AWS KMS用戶端。而是針對金鑰圈中以 KMS 金鑰表示的每個區域AWS Encryption SDK使用預設用AWS KMS戶端。例如，如果`Generator`參數值所識別的 KMS 金鑰位於美國西部 (奧勒岡) 區域 (`us-west-2`)，則會為該區`us-west-2`域AWS Encryption SDK建立預設AWS KMS用戶端。如果您需要自定義AWS KMS客戶端，請使用該`CreateAwsKmsKeyring()`方法。

```
// 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 = matProv.CreateAwsKmsMultiKeyring(keyringInput);
```

適用於 JAVA 的 AWS Encryption SDK支援使用對稱加密 (SYMMETRIC_DEFAULT) 或非對稱 RSA KMS 金鑰的AWS KMS金鑰圈。 AWS KMS使用非對稱 RSA KMS 金鑰建立的金鑰圈只能包含一個 key pair。

若要使用非對稱 RSA 金AWS KMS鑰圈加密，您不需要 kms: GenerateDataKey 或 KMS: Encrypt，因為您必須在建立金鑰環時指定要用於加密的公開金鑰材料。使用此金鑰圈加密時，不會進行任何AWS KMS呼叫。若要使用非對稱 RSA AWS KMS 金鑰圈進行解密，您需要 K MS: 解密權限。

若要建立非對稱 RSA 金AWS KMS鑰圈，您必須從非對稱 RSA KMS 金鑰提供公開金鑰和私密金鑰 ARN。公開金鑰必須是 PEM 編碼。下列範例會建立具有非對稱 RSA key pair AWS KMS 圈。

```
// 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);
```

3.x 版中必要的加密內容

使用版本 3。 的 x適用於 JAVA 的 AWS Encryption SDK，您可以使用必要的加密內容 CMM 來在您的密碼編譯作業中要求加密內容。加密內容是一組非秘密金鑰值配對。加密內容會以密碼編譯方式繫結至加密的資料，因此必須使用相同的加密內容來解密欄位。使用必要的加密內容 CMM 時，您可以指定一或多個必要的加密內容金鑰 (必要金鑰)，這些金鑰必須包含在所有加密和解密呼叫中。

> **Note**
>
> 必要的加密內容 CMM 只能與版本 4 互通。 AWS Encryption SDK用於 .NET 的 x。它不能與任何其他編程語言實現互操作。如果您使用必要的加密內容 CMM 加密資料，則只能使用版本 3 來解密資料。 適用於 JAVA 的 AWS Encryption SDK或版本 4 的 x。 AWS Encryption SDK 用於 .NET 的 x。

在加密時，AWS Encryption SDK會驗證所有必要的加密內容金鑰都包含在您指定的加密內容中。會AWS Encryption SDK簽署您指定的加密環境。只有非必要金鑰的索引鍵值配對才會以純文字形式儲存在加密作業傳回的加密訊息標頭中。

解密時，您必須提供一個加密內容，其中包含代表所需金鑰的所有金鑰-值配對。會AWS Encryption SDK使用此加密內容和儲存在加密郵件標頭中的金鑰值配對來重建您在加密作業中指定的原始加密內容。如果AWS Encryption SDK無法重建原始加密內容，則解密作業會失敗。如果您提供的金鑰-值組包含所需金鑰且值不正確，則無法解密加密的郵件。您必須提供與加密時指定的相同鍵值組。

> **Important**
>
> 請仔細考慮您為加密內容中的必要金鑰選擇哪些值。您必須能夠在解密時再次提供相同的金鑰及其對應的值。如果您無法重現所需的金鑰，則無法解密加密的訊息。

下列範例會使用必要的加密內容 CMM 來初始化AWS KMS金鑰環。

```
// 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()
        );
```

適用於 JAVA 的 AWS Encryption SDK 範例

以下範例說明如何使用 適用於 JAVA 的 AWS Encryption SDK 來加密和解密資料。這些範例說明如何使用版本 3。 x 及更新版本的適用於 JAVA 的 AWS Encryption SDK. 版本 3. 的 x 會以金鑰環適用於 JAVA 的 AWS Encryption SDK取代主要金鑰提供者。如需使用舊版的範例，請在上的存放aws-encryption-sdk-java庫的「發行」清單中找到您的版本 GitHub。

主題

- 加密和解密字串
- 加密和解密位元組串流
- 使用多重金鑰環加密和解密位元組串流

加密和解密字串

下列範例說明如何使用版本 3。 x 的適用於 JAVA 的 AWS Encryption SDK來加密和解密字串。使用字串之前，請將其轉換為位元組陣列。

此範例使用AWS KMS金鑰圈。使用金AWS KMS鑰圈加密時，您可以使用金鑰識別碼、金鑰 ARN、別名或別名 ARN 來識別 KMS 金鑰。解密時，您必須使用金鑰 ARN 來識別 KMS 金鑰。

當您呼叫 `encryptData()` 方法時，它會傳回已加密訊息 (`CryptoResult`)，其中包含加密文字、加密的資料金鑰和加密內容。當您在 `CryptoResult` 物件上呼叫 `getResult` 時，它會傳回已加密訊息的 base-64 編碼字串版本，您可以將其傳遞給 `decryptData()` 方法。

同樣地，當您呼叫時`decryptData()`，它傳回的`CryptoResult`物件包含純文字訊息和 ID AWS KMS key。在應用程式傳回純文字之前，請確認加密的訊息中的 AWS KMS key ID 和加密內容如您所預期。

```
// 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);
  }
}
```

加密和解密位元組串流

以下範例說明如何使用 AWS Encryption SDK來加密和解密位元組串流。

此範例使用原始 AES 金鑰圈。

加密時，此範例會使用此AwsCrypto.builder() .withEncryptionAlgorithm()方法來指定沒有數位簽章的演算法套件。解密時，為了確保密文本是無符號的，這個例子使用的方

法。createUnsignedMessageDecryptingStream()該createUnsignedMessageDecryptingStrea
法，如果遇到帶有數字簽名的密文失敗。

如果您使用包含數位簽章的預設演算法套件加密，請改用此createDecryptingStream()方法，如下個範例所示。

```
// 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");
    }
}
```

使用多重金鑰環加密和解密位元組串流

下列範例說明如何AWS Encryption SDK搭配多重金鑰圈使用。使用多重 keyring 來加密資料時，其任何 keyring 中的任何包裝金鑰均可以解密該資料。此範例使用AWS KMS金鑰圈和 Raw RSA 金鑰圈作為子鑰匙圈。

此範例會使用預設演算法套件加密，其中包含數位簽章。串流時，會在完整性檢查之後，但在驗證數位簽章之前AWS Encryption SDK釋放純文字。為了避免在驗證簽章之前使用純文字，此範例會緩衝純文字，並且只有在解密和驗證完成後才會將其寫入磁碟。

```
// 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());

    }
}
```

適用於 JavaScript 的 AWS Encryption SDK

所以此適用於 JavaScript 的 AWS Encryption SDK的目的是要為正在編寫 Web 瀏覽器應用程式的開發人員提供用户端加密程式庫JavaScript或使用 Node.js 編寫的 Web 伺服器應用程式。

與所有 AWS Encryption SDK實作相同，適用於 JavaScript 的 AWS Encryption SDK 提供進階的資料保護功能。這些功能包括信封加密、額外的驗證資料 (AAD) 以及安全、已認證的對稱金鑰演算法套件，例如 256 位元 AES-GCM 搭配金鑰衍生和簽署。

取決於語言限制，AWS Encryption SDK 的所有特定語言實作的設計為可互通。如需有關 JavaScript 的語言限制的詳細資訊，請參閱 the section called "相容性"。

進一步了解

- 如需相關詳細資訊適用於 JavaScript 的 AWS Encryption SDK，請參aws-encryption-sdk-javascript上的儲存庫GitHub。
- 有關編程示例，請參閱the section called "範例"與example-browser和example-node中的模組aws-encryption-sdk-javascript儲存庫。

- 有關使用適用於 JavaScript 的 AWS Encryption SDK加密 Web 應用程序中的數據，請參閱如何在瀏覽器中啟用加密適用於 JavaScript 的 AWS Encryption SDK和 Node.js中的AWS安全部落格。

主題

- 適用於 JavaScript 的 AWS Encryption SDK 的相容性
- 安裝 適用於 JavaScript 的 AWS Encryption SDK
- 適用於 JavaScript 的 AWS Encryption SDK 中的模組
- 適用於 JavaScript 的 AWS Encryption SDK 範例

適用於 JavaScript 的 AWS Encryption SDK 的相容性

適用於 JavaScript 的 AWS Encryption SDK 的設計是為了與 AWS Encryption SDK 的其他語言實作互通。在大多數情況下，您可以使用 適用於 JavaScript 的 AWS Encryption SDK 加密資料，並使用任何其他語言實作 (包括 AWS Encryption SDK 命令列介面) 來解密資料。並且您可以使用適用於 JavaScript 的 AWS Encryption SDK解密已加密的訊息由其他語言實現AWS Encryption SDK。

但是，當您使用 適用於 JavaScript 的 AWS Encryption SDK 時，您需要注意語言 JavaScript 實作和 Web 瀏覽器中的一些相容性問題。

此外，使用不同的語言實作時，請務必設定相容的主密鑰提供程式、主密鑰和鑰環。如需詳細資訊，請參閱 Keyring 相容性。

適用於 JavaScript 的 AWS Encryption SDK 相容性

AWS Encryption SDK 的 JavaScript 實作與其他語言實作在下列方面有所不同：

- 適用於 JavaScript 的 AWS Encryption SDK 的加密操作不會傳回無框架加密文字。但是，適用於 JavaScript 的 AWS Encryption SDK 會將 AWS Encryption SDK 的其他語言實作傳回的框架和無框架加密文字解密。
- 從 Node.js 版本 12.9.0 開始，Node.js 支援以下 RSA 金鑰包裝選項：
  - 具有 SHA1、SHA256、SHA384 或 SHA512 的 OAEP
  - 具有 SHA1 的 OAEP 和具有 SHA1 的 MGF1
  - PKCS1v15
- 在版本 12.9.0 之前，Node.js 僅支援以下 RSA 金鑰包裝選項：
  - 具有 SHA1 的 OAEP 和具有 SHA1 的 MGF1
  - PKCS1v15

瀏覽器相容性

某些 Web 瀏覽器不支援 適用於 JavaScript 的 AWS Encryption SDK 所需的基本密碼編譯操作。如需設定備用程式，您可以彌補部分遺漏的操作。WebCrypto瀏覽器實現的 API。

Web 瀏覽器限制

下列限制為所有 Web 瀏覽器通用：

- 所以此WebCryptoAPI 不支援 PKCS1v15 金鑰包裝。
- 瀏覽器不支援 192 位元金鑰。

必要的密碼編譯操作

適用於 JavaScript 的 AWS Encryption SDK 需要在 Web 瀏覽器中執行以下操作。如果瀏覽器不支援這些操作，則它與 適用於 JavaScript 的 AWS Encryption SDK 相容。

- 瀏覽器必須包含 `crypto.getRandomValues()`，這是一種以密碼編譯方式產生隨機值的方法。如需支援`crypto.getRandomValues()`，請參我可以使用密碼編譯。getRandomValues()?。

必要的備用

適用於 JavaScript 的 AWS Encryption SDK 需要下列程式庫並在 Web 瀏覽器中執行以下操作。如果您支援不符合這些需求的 Web 瀏覽器，則必須設定備用。否則，嘗試使用 適用於 JavaScript 的 AWS Encryption SDK 搭配瀏覽器將會失敗。

- 所以此WebCryptoAPI 會在 Web 應用程式中執行基本密碼編譯操作，但並非可在所有瀏覽器上使用。如需支援 Web 密碼編譯的 Web 瀏覽器版本的相關資訊，請參閱我可以使用 Web 密碼編譯嗎？。
- Safari Web 瀏覽器的現代版本不支援零位元組的 AES-GCM 加密，而這是 AWS Encryption SDK 所需。如果瀏覽器實現WebCryptoAPI，但無法使用 AES-GCM 來加密零位元組，適用於 JavaScript 的 AWS Encryption SDK僅使用回退庫進行零字節加密。它使用WebCryptoAPI 進行所有其他操作。

若要設定任一限制的備用，請將下列陳述式新增至您的程式碼。在 configureFallback 函數中，指定支援遺漏功能的程式庫。下列範例使用 Microsoft ResearchJavaScript密碼編譯庫 (`msrcrypto`)，但是您可以以相容的程式庫取代它。如需完整範例，請參閱 fallback.ts。

```
import { configureFallback } from '@aws-crypto/client-browser'
```

```
configureFallback(msrCrypto)
```

安裝 適用於 JavaScript 的 AWS Encryption SDK

適用於 JavaScript 的 AWS Encryption SDK 由相互依存的模組集合組成。模組中的數個只是設計要一起運作的模組集合。部分模組是專為獨立運作而設計。一些模組為所有實作所需；一些模組則僅用於特殊情況。如需中模組的相關資訊AWS Encryption SDK為了 JavaScript，請參閱適用於 JavaScript 的 AWS Encryption SDK 中的模組和`README.md`文件中的每個模塊aws-encryption-sdk-javascript儲存庫於 GitHub。

> **Note**
>
> 所有版本適用於 JavaScript 的 AWS Encryption SDK2.0.0 版以前的資源在end-of-support階段。
> 您可以安全地從 2.0 版更新。x以及更高版本的最新版本適用於 JavaScript 的 AWS Encryption SDK沒有任何代碼或數據更改。但是，新安全性功能2.0 版中引入。x不向後兼容。若要從 1.7 之前的版本進行更新。x至 2.0 版x之後，您必須首先更新到最新的 1。x的版本適用於 JavaScript 的 AWS Encryption SDK。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

若要安裝模組，請使用 npm 套件管理工具。

例如，若要安裝 `client-node` 模組，其中包括在 Node.js 中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式您所需的所有模組，請使用以下命令。

```
npm install @aws-crypto/client-node
```

若要安裝 `client-browser` 模組，其中包括在瀏覽器中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式您所需的所有模組，請使用以下命令。

```
npm install @aws-crypto/client-browser
```

有關如何使用適用於 JavaScript 的 AWS Encryption SDK，請參閱中的範例`example-node`和`example-browser`模組aws-encryption-sdk-javascript儲存庫於 GitHub。

適用於 JavaScript 的 AWS Encryption SDK 中的模組

適用於 JavaScript 的 AWS Encryption SDK 中的模組可讓您輕鬆安裝您的專案所需的程式碼。

的模組JavaScriptNode.js

client-node

包括在 Node.js 中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式您所需的所有模組。

caching-materials-manager-節點

匯出支援的函數資料金鑰快取中的適用於 JavaScript 的 AWS Encryption SDK中的 Node.js。

decrypt-node

匯出會解密並驗證代表資料和資料流的加密訊息的函數。包含在 `client-node` 模組中。

encrypt-node

匯出加密和簽署不同類型資料的函數。包含在 `client-node` 模組中。

example-node

匯出在 Node.js 中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式的工作範例。包括不同類型的 keyring 和不同類型資料的範例。

hkdf-node

匯出基於 HMAC 的密鑰派生函數(香港發展基金) 認為適用於 JavaScript 的 AWS Encryption SDK 在特定的算法套件中使用。Node.js 所以此適用於 JavaScript 的 AWS Encryption SDK中的會使用瀏覽器中的原生 HKDF 函數。WebCryptoAPI。

integration-node

定義測試，以在 Node.js 中驗證 適用於 JavaScript 的 AWS Encryption SDK 是否與 AWS Encryption SDK 的其他語言實作相容。

kms-keyring-node

匯出支援的函數AWS KMSNode.js 中的 keyrings。

raw-aes-keyring-節點

匯出在 Node.js 中支援原始 AES keyring 的函數。

raw-rsa-keyring-節點

匯出在 Node.js 中支援原始 RSA keyring 的函數。

的模組JavaScript瀏覽器

client-browser

包括在瀏覽器中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式您所需的所有模組。

caching-materials-manager瀏覽器

匯出支援的函數資料金鑰快取此功能JavaScript瀏覽器中的。

decrypt-browser

匯出會解密並驗證代表資料和資料流的加密訊息的函數。

encrypt-browser

匯出加密和簽署不同類型資料的函數。

example-browser

在瀏覽器中使用 適用於 JavaScript 的 AWS Encryption SDK 編寫程式的工作範例。包括不同類型的 keyring 和不同類型資料的範例。

integration-browser

定義測試，以在瀏覽器中驗證 適用於 JAVA 的 AWS Encryption SDK 指令碼是否與 AWS Encryption SDK 的其他語言實作相容。

kms-keyring-browser

匯出支援的函數AWS KMSKeyring瀏覽器中的。

raw-aes-keyring瀏覽器

匯出在瀏覽器中支援原始 AES keyring 的函數。

raw-rsa-keyring瀏覽器

匯出在瀏覽器中支援原始 RSA keyring 的函數。

適用於所有實作的模組

cache-material

支援資料金鑰快取功能。提供用於組合隨每個資料金鑰快取的密碼編譯資料的程式碼。

kms-keyring

匯出支援 KMS keyring 的函數。

material-management

實作密碼編譯資料管理員 (CMM)。

raw-keyring

匯出原始 AES 和 RSA keyring 所需的函數。

serialize

匯出 SDK 用來序列化其輸出的函數。

web-crypto-backend

導出使用WebCryptoAPI 中的適用於 JavaScript 的 AWS Encryption SDK瀏覽器中的。

適用於 JavaScript 的 AWS Encryption SDK 範例

以下範例說明如何使用 適用於 JavaScript 的 AWS Encryption SDK 來加密和解密資料。

您可以找到更多使用適用於 JavaScript 的 AWS Encryption SDK中的example-node和example-browser模組中的aws-encryption-sdk-javascript存儲庫GitHub。當您安裝 `client-browser` 或 `client-node` 模組時，不會安裝這些範例模組。

查看完整的程式碼範例：節點：千米簡單。，瀏覽器：千米簡單。

主題

- 使用 加密資料AWS KMSKeyring
- 使用 解密資料AWS KMSKeyring

使用 加密資料AWS KMSKeyring

以下範例說明如何使用 適用於 JavaScript 的 AWS Encryption SDK 來加密和解密短字串或位元組陣列。

此示例的特點是AWS KMSKeyring，一種密鑰環，使用AWS KMS key生成和加密數據密鑰。有關創建AWS KMS key，請參建立金鑰中的AWS Key Management Service開發人員指南。有關識別AWS KMS keys在AWS KMSkeyring，請參在 AWS KMS 鑰匙圈 AWS KMS keys 中識別

步驟 1：建構 keyring。

建立AWS KMS用於加密的金鑰。

當使用AWS KMS密鑰環，則必須指定產生器金鑰，也就是說，一個AWS KMS key，用來產生純文字資料金鑰並加密它。您也可以指定零個或多個額外的金鑰來加密相同的純文字資料金鑰。keyring 會傳回 keyring 中每個 AWS KMS key 金鑰 (包括產生器金鑰) 的純文字資料金鑰，以及該資料金鑰的一個加密副本。若要解密資料，您需要解密任何一個加密的資料金鑰。

若要在 適用於 JavaScript 的 AWS Encryption SDK 中指定加密 keyring 的 AWS KMS keys，您可以使用任何支援的 AWS KMS 金鑰識別符。此範例會使用產生器金鑰 (依別名 ARN 識別)，以及一個額外的金鑰 (依金鑰 ARN 識別)。

> **Note**
>
> 如果您計劃重複使用AWS KMS金鑰環進行解密，您必須使用金鑰 ARN 標識AWS KMS keys在金鑰圈。

執行此程式碼之前，請將範例 AWS KMS key 識別碼以有效的識別碼取代。您必須具有在 keyring 中使用 AWS KMS keys 所需的許可。

JavaScript Browser

首先提供您的登入資料給瀏覽器。所以此適用於 JavaScript 的 AWS Encryption SDK範例會使用Webpack。DefinePlugin，它會以您的實際登入資料替換憑據常數。但是您可以使用任何方法來提供您的登入資料。然後，使用登入資料來建立 AWS KMS 用戶端。

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

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

接下來，指定AWS KMS keys用於產生器金鑰和額外金鑰。然後，創建一個AWS KMS使用keyringAWS KMS客户端和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 })
```

步驟 2：設定加密內容。

加密內容是一種任意、非私密額外驗證資料。在加密時提供加密內容時，AWS Encryption SDK 會以密碼編譯方式將加密內容繫結至加密文字，使得解密資料會需要相同的加密內容。使用加密內容是選用的，但我們建議使用它作為最佳實務。

建立包含加密內容對的簡單物件。每個對組中的索引鍵和值必須是字串。

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'
}
```

步驟 3：加密數據。

若要加密純文字資料，請呼叫 `encrypt` 函數。傳入AWS KMSkeyring、純文字資料和加密內容。

`encrypt` 函數會傳回加密訊息 (`result`)，其中包含加密的資料、加密的資料金鑰和重要的中繼資料，包括加密內容和簽章。

您可以解密此加密郵件通過使用AWS Encryption SDK查看任何受支持的編程語言。

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 解密資料AWS KMSKeyring

您可以使用 適用於 JavaScript 的 AWS Encryption SDK 來解密加密的訊息，並復原原始資料。

在此範例中，我們會解密我們在 the section called "使用 加密資料AWS KMSKeyring" 範例中加密的資料。

步驟 1：建構 keyring。

若要解密資料，請傳入已加密訊息(`result`)，該`encrypt`函數返回。加密的訊息包括加密的資料、加密的資料金鑰和重要的中繼資料，包括加密內容和簽章。

您還必須指定AWS KMSKeyring解密時。您可以使用用來加密資料或不同 keyring 的相同 keyring。若要成功，解密 keyring 中至少有一個 AWS KMS key 可以解密加密訊息中其中一個加密的資料金鑰。由於不會產生任何資料金鑰，您不需要在解密 keyring 中指定產生器金鑰。如果您這麼做，則會以相同方式處理產生器金鑰和額外金鑰。

要指定AWS KMS key中的解密密鑰環適用於 JavaScript 的 AWS Encryption SDK，您必須使用金鑰 ARN。否則，將無法辨識 AWS KMS key。有關識別AWS KMS keys在AWS KMSkeyring，請參在 AWS KMS 鑰匙圈 AWS KMS keys 中識別

> **Note**
>
> 如果您使用相同的 keyring 進行加密和解密，請使用金鑰 ARN 來識別 keyring 中的 AWS KMS keys。

在此範例中，我們會建立只包含加密 keyring 中其中一個 AWS KMS keys 的 keyring。執行此程式碼之前，請將範例金鑰 ARN 換成有效的金鑰 ARN。您必須具有 AWS KMS key 上的 `kms:Decrypt` 許可。

JavaScript Browser

首先提供您的登入資料給瀏覽器。所以此適用於 JavaScript 的 AWS Encryption SDK範例會使用Webpack。DefinePlugin，它會以您的實際登入資料替換憑據常數。但是您可以使用任何方法來提供您的登入資料。然後，使用登入資料來建立 AWS KMS 用戶端。

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

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

接下來，建立AWS KMS使用 keyringAWS KMS客户端。此範例只使用來自加密 keyring 的其中一個 AWS KMS keys。

```
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 })
```

步驟 2：解密資料。

接下來，呼叫 `decrypt` 函數。傳入剛剛創建的解密密鑰環（`keyring`）和已加密訊息認為`encrypt`函數返回 (`result`。AWS Encryption SDK 使用 keyring 來解密其中一個加密的資料金鑰。然後它會使用純文字資料金鑰來解密資料。

如果呼叫成功，plaintext 欄位會包含純文字 (已解密) 資料。messageHeader 欄位包含有關解密程序的中繼資料，包括用來解密資料的加密內容。

JavaScript Browser

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

JavaScript Node.js

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

步驟 3：驗證加密內容。

用來解密資料的加密內容包含在 decrypt 函數傳回的訊息標頭 (messageHeader) 中。在您的應用程式傳回純文字資料之前，請確認您在加密時所提供的加密內容包含在解密時所使用的加密內容中。不相符可能表示資料遭到竄改，或您沒有解密正確的加密文字。

驗證加密內容時，請勿要求完全相符。使用加密演算法搭配簽署時，密碼編譯資料管理員 (CMM) 會在加密訊息之前，將公有簽署金鑰新增至加密內容。但是，您提交的所有加密內容對都應該包含在傳回的加密內容中。

首先，從訊息標頭取得加密內容。然後，確認原始加密內容 (context) 中的每個索引鍵值對符合傳回的加密內容 (encryptionContext) 中的索引鍵值對。

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')
})
```

如果加密內容檢查成功，您可以傳回純文字資料。

適用於 Python 的 AWS Encryption SDK

本主題說明如何安裝及使用 適用於 Python 的 AWS Encryption SDK。如需有關使用程式設計的詳細資訊 適用於 Python 的 AWS Encryption SDK，請參閱上的aws-encryption-sdk-python存放庫 GitHub。如需 API 文件，請參閱閱讀相關文件。

主題

- 必要條件
- 安裝
- 適用於 Python 的 AWS Encryption SDK 範例程式碼

必要條件

在安裝之前 適用於 Python 的 AWS Encryption SDK，請確定您具備下列先決條件。

支援的 Python 版本

3.2.0 及更高版本需要 Python 3.8 或更高 適用於 Python 的 AWS Encryption SDK 版本。

早期版本的 AWS Encryption SDK 支持 Python 2.7 和 Python 3.4 及更高版本，但我們建議您使用最新版本的 AWS Encryption SDK.

若要下載 Python，請參閱 Python 下載。

適用於 Python 的 pip 安裝工具

`pip`包含在 Python 3.6 及更高版本中，儘管您可能需要升級它。如需有關升級或安裝的詳細資訊`pip`，請參閱`pip`說明文件中的安裝。

安裝

安裝最新版本的 適用於 Python 的 AWS Encryption SDK.

Note

3.0.0 適用於 Python 的 AWS Encryption SDK 之前版本的所有版本都處於此end-of-support 階段。

您可以安全地從 2.0 版更新。 x 及更新版本的最新版本， AWS Encryption SDK 無需任何代碼或數據更改。但是，2.0 版中引入了新的安全功能。 x 不向後相容。若要從 1.7 之前的版本進行更新。 x 轉換為 2.0 版本。 x 和更新版本，您必須先更新到最新的 1。 x 版本的 AWS Encryption SDK. 如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

用pip於安裝 適用於 Python 的 AWS Encryption SDK，如下列範例所示。

若要安裝最新版本

```
pip install aws-encryption-sdk
```

如需使用 pip 來安裝及升級套件的詳細資訊，請參閱安裝套件。

適用於 Python 的 AWS Encryption SDK 需要所有平台上的密碼編譯程式庫 (pyca/ 密碼編譯)。所有版本的pip自動在 Windows 上安裝和構建cryptography庫。 pip8.1 及更新版本會自動cryptography在 Linux 上安裝和建置。如果您使用的是舊版，pip且 Linux 環境沒有建置程式cryptography庫所需的工具，則需要安裝它們。如需詳細資訊，請參閱在 Linux 上建置密碼編譯。

版本 1.10.0 和 2.5.0 的引腳在 2.5.0 和 3.3.2 之間的 適用於 Python 的 AWS Encryption SDK 密碼編譯依賴關係。其他版本的 適用於 Python 的 AWS Encryption SDK 安裝最新版本的密碼學。如果您需要的密碼編譯版本低於 3.3.2，建議您使用. 適用於 Python 的 AWS Encryption SDK

如需的最新開發版本 適用於 Python 的 AWS Encryption SDK，請移至中的aws-encryption-sdk-python存放庫 GitHub。

安裝之後 適用於 Python 的 AWS Encryption SDK，請參閱本指南中的 Python 範例程式碼開始使用。

適用於 Python 的 AWS Encryption SDK 範例程式碼

以下範例說明如何使用 適用於 Python 的 AWS Encryption SDK 來加密和解密資料。

本節中的範例展示如何使用2.0 版。x和後來的適用於 Python 的 AWS Encryption SDK。有關使用早期版本的示例，請在版本的清單aws-encryption-sdk-python的儲存庫 GitHub。

主題

- 加密和解密字串
- 加密和解密位元組串流
- 加密和解密具有多個主金鑰提供程序的位元組串流
- 使用資料金鑰快取來加密訊息

加密和解密字串

以下範例說明如何使用 AWS Encryption SDK 來加密和解密字串。此範例使用AWS KMS key在AWS Key Management Service(AWS KMS)作為主金鑰。

加密時，StrictAwsKmsMasterKeyProvider構造函數採用金鑰 ID、金鑰 ARN、別名名稱或別名 ARN。解密時，它需要金鑰 ARN。在這種情況下，因為keyArn參數用於加密和解密，其值必須是密鑰 ARN。如需有關 ID 的資訊AWS KMS鍵，請參閱密鑰標識符中的AWS Key Management Service開發人員指南。

```
# 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 showing basic encryption and decryption of a value already in memory."""
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def cycle_string(key_arn, source_plaintext, botocore_session=None):
    """Encrypts and then decrypts a string under an &KMS; key.

    :param str key_arn: Amazon Resource Name (ARN) of the &KMS; key
    :param bytes source_plaintext: Data to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
```

```
    """
    # Set up an encryption client with an explicit commitment policy. 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_REQU

    # Create an AWS KMS master key provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
        kms_kwargs["botocore_session"] = botocore_session
    master_key_provider =
aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(**kms_kwargs)

    # Encrypt the plaintext source data
    ciphertext, encryptor_header = client.encrypt(source=source_plaintext,
key_provider=master_key_provider)

    # Decrypt the ciphertext
    cycled_plaintext, decrypted_header = client.decrypt(source=ciphertext,
key_provider=master_key_provider)

    # Verify that the "cycled" (encrypted, then decrypted) plaintext is identical to
the source plaintext
    assert cycled_plaintext == source_plaintext

    # Verify that the encryption context used in the decrypt operation includes all key
pairs from
    # the encrypt operation. (The SDK can add pairs, so don't require an exact match.)
    #
    # In production, always use a meaningful encryption context. In this sample, we
omit the
    # encryption context (no key pairs).
    assert all(
        pair in decrypted_header.encryption_context.items() for pair in
encryptor_header.encryption_context.items()
    )
```

加密和解密位元組串流

以下範例說明如何使用 AWS Encryption SDK來加密和解密位元組串流。此範例不使用 AWS。它使用靜態、暫時性主金鑰提供者。

加密時，本示例使用一個不帶[數字簽名](AES_256_GCM_HKDF_SHA512_COMMIT_KEY。當正在加密和解密數據的用户同樣受信任時，此算法套件適用。然後，解密時，該示例使用decrypt-unsigned流模式，如果遇到簽名密文，該模式將失敗。所以此decrypt-unsigned流媒體模式被引入AWS Encryption SDK1.9 版。x和 2.2.x。

```
# 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 showing creation and use of a RawMasterKeyProvider."""
import filecmp
import os

import aws_encryption_sdk
from aws_encryption_sdk.identifiers import Algorithm, CommitmentPolicy,
 EncryptionKeyType, WrappingAlgorithm
from aws_encryption_sdk.internal.crypto.wrapping_keys import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly generates 256-bit keys for each unique key ID."""

    provider_id = "static-random"

    def __init__(self, **kwargs):  # pylint: disable=unused-argument
        """Initialize empty map of keys."""
        self._static_keys = {}

    def _get_raw_key(self, key_id):
        """Returns a static, randomly-generated symmetric key for the specified key
 ID.
        :param str key_id: Key ID
        :returns: Wrapping key that contains the specified static key
        :rtype: :class:`aws_encryption_sdk.internal.crypto.WrappingKey`
```

```
        """
        try:
            static_key = self._static_keys[key_id]
        except KeyError:
            static_key = os.urandom(32)
            self._static_keys[key_id] = static_key
        return WrappingKey(
            wrapping_algorithm=WrappingAlgorithm.AES_256_GCM_IV12_TAG16_NO_PADDING,
            wrapping_key=static_key,
            wrapping_key_type=EncryptionKeyType.SYMMETRIC,
        )


def cycle_file(source_plaintext_filename):
    """Encrypts and then decrypts a file under a custom static master key provider.
    :param str source_plaintext_filename: Filename of file to encrypt
    """
    # 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_REQU

    # Create a static random master key provider
    key_id = os.urandom(8)
    master_key_provider = StaticRandomMasterKeyProvider()
    master_key_provider.add_master_key(key_id)

    ciphertext_filename = source_plaintext_filename + ".encrypted"
    cycled_plaintext_filename = source_plaintext_filename + ".decrypted"

    # Encrypt the plaintext source data
    # We can use an unsigning algorithm suite here under the assumption that the
 contexts that encrypt
    # and decrypt are equally trusted.
    with open(source_plaintext_filename, "rb") as plaintext, open(ciphertext_filename,
 "wb") as ciphertext:
        with client.stream(
            algorithm=Algorithm.AES_256_GCM_HKDF_SHA512_COMMIT_KEY,
            mode="e",
            source=plaintext,
            key_provider=master_key_provider,
        ) as encryptor:
            for chunk in encryptor:
```

```
                ciphertext.write(chunk)

    # Decrypt the ciphertext
    # We can use the recommended "decrypt-unsigned" streaming mode since we encrypted
 with an unsigned algorithm suite.
    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 "cycled" (encrypted, then decrypted) plaintext is identical to
 the source
    # plaintext
    assert filecmp.cmp(source_plaintext_filename, cycled_plaintext_filename)

    # Verify that the encryption context used in the decrypt operation includes all key
 pairs from
    # the encrypt operation
    #
    # In production, always use a meaningful encryption context. In this sample, we
 omit the
    # encryption context (no key pairs).
    assert all(
        pair in decryptor.header.encryption_context.items() for pair in
 encryptor.header.encryption_context.items()
    )
    return ciphertext_filename, cycled_plaintext_filename
```

加密和解密具有多個主金鑰提供程序的位元組串流

以下範例顯示如何使用 AWS Encryption SDK搭配多個主金鑰提供者。使用多個主金鑰提供者可在一個主金鑰提供者無法用於解密時建立備援。此範例使用AWS KMS key和 RSA 金 key pair 做為主金鑰。

此示例使用默認算法套件，其中包含資訊簽章。流式傳輸時，AWS Encryption SDK在完整性檢查後發佈明文，但在驗證數字簽名之前。為了避免在驗證簽名之前使用明文，本示例緩衝明文，並且僅在解密和驗證完成時才將其寫入磁盤。

```
# 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 showing creation of a RawMasterKeyProvider, how to use multiple
master key providers to encrypt, and demonstrating that each master key
provider can then be used independently to decrypt the same encrypted message.
"""
import filecmp
import os

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa

import aws_encryption_sdk
from aws_encryption_sdk.identifiers import CommitmentPolicy, EncryptionKeyType,
 WrappingAlgorithm
from aws_encryption_sdk.internal.crypto.wrapping_keys import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider


class StaticRandomMasterKeyProvider(RawMasterKeyProvider):
    """Randomly generates and provides 4096-bit RSA keys consistently per unique key
 id."""

    provider_id = "static-random"

    def __init__(self, **kwargs):  # pylint: disable=unused-argument
        """Initialize empty map of keys."""
        self._static_keys = {}

    def _get_raw_key(self, key_id):
        """Retrieves a static, randomly generated, RSA key for the specified key id.

        :param str key_id: User-defined ID for the static key
        :returns: Wrapping key that contains the specified static key
        :rtype: :class:`aws_encryption_sdk.internal.crypto.WrappingKey`
        """
        try:
```

```
            static_key = self._static_keys[key_id]
        except KeyError:
            private_key = rsa.generate_private_key(public_exponent=65537,
 key_size=4096, backend=default_backend())
            static_key = private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption(),
            )
            self._static_keys[key_id] = static_key
        return WrappingKey(
            wrapping_algorithm=WrappingAlgorithm.RSA_OAEP_SHA1_MGF1,
            wrapping_key=static_key,
            wrapping_key_type=EncryptionKeyType.PRIVATE,
        )


def cycle_file(key_arn, source_plaintext_filename, botocore_session=None):
    """Encrypts and then decrypts a file using an AWS KMS master key provider and a
 custom static master
    key provider. Both master key providers are used to encrypt the plaintext file, so
 either one alone
    can decrypt it.

    :param str key_arn: Amazon Resource Name (ARN) of the &KMS; key
    (http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html)
    :param str source_plaintext_filename: Filename of file to encrypt
    :param botocore_session: existing botocore session instance
    :type botocore_session: botocore.session.Session
    """
    # "Cycled" means encrypted and then decrypted
    ciphertext_filename = source_plaintext_filename + ".encrypted"
    cycled_kms_plaintext_filename = source_plaintext_filename + ".kms.decrypted"
    cycled_static_plaintext_filename = source_plaintext_filename + ".static.decrypted"

    # 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_REQU

    # Create an AWS KMS master key provider
    kms_kwargs = dict(key_ids=[key_arn])
    if botocore_session is not None:
```

```
        kms_kwargs["botocore_session"] = botocore_session
    kms_master_key_provider =
aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(**kms_kwargs)

    # Create a static master key provider and add a master key to it
    static_key_id = os.urandom(8)
    static_master_key_provider = StaticRandomMasterKeyProvider()
    static_master_key_provider.add_master_key(static_key_id)

    # Add the static master key provider to the AWS KMS master key provider
    #   The resulting master key provider uses AWS KMS master keys to generate (and
encrypt)
    #   data keys and static master keys to create an additional encrypted copy of each
data key.
    kms_master_key_provider.add_master_key_provider(static_master_key_provider)

    # Encrypt plaintext with both AWS KMS and static master keys
    with open(source_plaintext_filename, "rb") as plaintext, open(ciphertext_filename,
"wb") as ciphertext:
        with client.stream(source=plaintext, mode="e",
key_provider=kms_master_key_provider) as encryptor:
            for chunk in encryptor:
                ciphertext.write(chunk)

    # Decrypt the ciphertext with only the AWS KMS master key
    # Buffer the data in memory before writing to disk. This ensures verfication of the
digital signature before returning plaintext.
    with open(ciphertext_filename, "rb") as ciphertext,
open(cycled_kms_plaintext_filename, "wb") as plaintext:
        with client.stream(
            source=ciphertext, mode="d",
key_provider=aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(**kms_kwargs)
        ) as kms_decryptor:
             plaintext.write(kms_decryptor.read())

    # Decrypt the ciphertext with only the static master key
    # Buffer the data in memory before writing to disk to ensure verfication of the
signature before returning plaintext.
    with open(ciphertext_filename, "rb") as ciphertext,
open(cycled_static_plaintext_filename, "wb") as plaintext:
        with client.stream(source=ciphertext, mode="d",
key_provider=static_master_key_provider) as static_decryptor:
             plaintext.write(static_decryptor.read())
```

```
    # Verify that the "cycled" (encrypted, then decrypted) plaintext is identical to
 the source plaintext
    assert filecmp.cmp(source_plaintext_filename, cycled_kms_plaintext_filename)
    assert filecmp.cmp(source_plaintext_filename, cycled_static_plaintext_filename)

    # Verify that the encryption context in the decrypt operation includes all key
 pairs from the
    # encrypt operation.
    #
    # In production, always use a meaningful encryption context. In this sample, we
 omit the
    # encryption context (no key pairs).
    assert all(
        pair in kms_decryptor.header.encryption_context.items() for pair in
 encryptor.header.encryption_context.items()
    )
    assert all(
        pair in static_decryptor.header.encryption_context.items()
        for pair in encryptor.header.encryption_context.items()
    )
    return (ciphertext_filename, cycled_kms_plaintext_filename,
 cycled_static_plaintext_filename)
```

使用資料金鑰快取來加密訊息

下列範例示範如何使用資料金鑰快取中的適用於 Python 的 AWS Encryption SDK。它旨在向您展示如何設定本機快取(LocalCryptoMaterialsCache），其中包含所需容量值和快取密碼編譯資料管理員（緩存 CMM）快取安全性閾值。

這個非常基本的範例建立一個函數來加密固定字串。它允許您指定AWS KMS key、所需的快取大小 (容量) 和存留期上限值。如需更複雜、實際的資料金鑰快取範例，請參閱 資料金鑰快取範例程式碼。

這個範例也使用加密內容做為額外驗證的資料 (選用)。當您解密以加密內容所加密的資料時，請確保您的應用程式在將純文字資料傳回給發起人之前，將會驗證加密內容是否如您所預期。加密內容是任何加密或解密操作的最佳實務元素，但在資料金鑰快取中扮演特殊的角色。如需詳細資訊，請參閱加密內容：如何選擇快取項目。

```
# 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_REQU

    # 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
```

AWS Encryption SDK 命令列界面

所以此AWS Encryption SDK命令列界面 (AWS加密 CLI) 可讓您使用AWS Encryption SDK在命令列和指令碼中以互動方式加密和解密資料。您不需要具備密碼編譯或程式設計的專業知識。

> **Note**
>
> 的版本AWS4.0.0 版以前的加密 CLI 位於end-of-support階段。
> 您可以安全的 2.1 版。x以及更高版本的最新版本AWS加密 CLI，無需任何代碼或數據更改。但是，新的安全性功能在 2.1 版中引入。x不向後兼容。要從 1.7 版。x或更早版本，您必須先更新到最新的 1。x的版本AWS加密 CLI。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。
> 最初發布了新的安全功能AWS加密 CLI 1.7 版。x和 2.0。x。但是，AWS加密版本 1.8.x取代 1.7 版。x和AWSEncryption 2.1.x取代 2.0。x。如需詳細資訊，請參閱相關安全性公告在aws-encryption-sdk-cli儲存庫 GitHub。

就像所有的實現一樣AWS Encryption SDK，該AWS加密 CLI 提供進階資料保護功能。其中包括信封加密、其他驗證資料 (AAD)，以及安全、經驗證的對稱金鑰演算法套件，包含密鑰派生的 256 位元 AES-GCM，關鍵承諾，並簽署。

所以此AWS加密 CLI 是建置在適用於 Python 的 AWS Encryption SDK和在 Linux、macOS 和 Windows 都提供支援。您可以執行命令和指令碼，以便在 Linux 或 macOS 上偏好的殼層、Windows 上的 [命令提示字元] 視窗 (cmd.exe) 以及 PowerShell 任何系統上的控制台。

所有特定於語言的實現AWS Encryption SDK，包括AWS加密 CLI，是可互操作的。例如，您可以使用適用於 JAVA 的 AWS Encryption SDK並用它解密AWS加密 CLI。

本主題介紹AWS加密 CLI 說明如何安裝和使用它，並提供數個範例來協助您開始使用。如需快速入門，請參閱如何加密和解密您的數據AWS加密 CLI在AWS安全部落格。如需詳細資訊，請參閱閱讀文件，並加入我們的發展AWS在中加密 CLIaws-encryption-sdk-cli儲存庫 GitHub。

效能

所以此AWS加密 CLI 是建置在適用於 Python 的 AWS Encryption SDK。每次執行 CLI 時，都會啟動 Python 執行時間的新執行個體。若要改善效能，請盡可能使用單一命令而非一系列的獨立命令。例如，執行會以遞迴方式處理目錄中檔案的一個命令，而不是對每個檔案執行個別命令。

主題

- 安裝命 AWS Encryption SDK 令行界面
- 如何使用加AWS密 CLI
- 示例AWS加密 CLI
- AWS Encryption SDK CLI 語法和參數參考
- 的版本AWS加密 CLI

安裝命 AWS Encryption SDK 令行界面

本主題說明如何安裝 AWS 加密 CLI。有關詳細信息，請參閱上的aws-encryption-sdk-cli存儲庫 GitHub 和閱讀文檔。

主題

- 安裝必要項目
- 安裝和更新加 AWS 密 CLI

安裝必要項目

加 AWS 密 CLI 建置在 適用於 Python 的 AWS Encryption SDK. 若要安裝 AWS 加密 CLI，您需要 Python 和 `pip` Python 套件管理工具。所有支援的平台皆有提供 Python 與 `pip`。

安裝加 AWS 密 CLI 之前，請先安裝下列必要條件，

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。

舊版的 AWS 加密 CLI 支援 Python 2.7 和 3.4 及更新版本，但我們建議您使用最新版本的 AWS 加密 CLI。

大多數 Linux 和 macOS 安裝都包含 Python，但您需要升級到 3.6 或更新版本。我們建議您使用最新版本的 Python。在視窗上，您必須安裝 Python; 默認情況下不會安裝它。若要下載並安裝 Python，請參閱下載。

若要判斷 Python 是否已安裝完畢，請於命令列輸入下列內容。

```
python
```

若要查看 Python 版本，請使用 `-V` (大寫 V) 參數。

```
python -V
```

在視窗上，安裝 Python 之後，將`Python.exe`檔案的路徑新增至路徑環境變數的值。

在預設情況下，Python 會安裝在 `$home` 子目錄的所有使用者目錄或使用者描述檔目錄中 (`%userprofile%` 或 `AppData\Local\Programs\Python`)。若要找出系統中的 `Python.exe` 檔案，請查看下列登錄機碼。您可以使 PowerShell 用搜索註冊表。

```
PS C:\> dir HKLM:\Software\Python\PythonCore\version\InstallPath
# -or-
PS C:\> dir HKCU:\Software\Python\PythonCore\version\InstallPath
```

pip

`pip` 為 Python 套件管理工具。若要安裝加 AWS 密 CLI 及其相依性，您需要 `pip` 8.1 或更新版本。如需安裝或升級的說明`pip`，請參閱`pip`文件中的安裝。

在 Linux 安裝上，8.1 之**`pip`**前的版本無法建置 AWS 加密 CLI 所需的加密程式庫。如果您選擇不更新`pip`版本，則可以單獨安裝構建工具。如需詳細資訊，請參閱在 Linux 上建置密碼編譯。

AWS Command Line Interface

只有在 AWS Command Line Interface (AWS CLI) 搭配 AWS 加密 CLI 使用 AWS KMS keys 時，才需要 AWS Key Management Service (AWS KMS)。如果您使用不同的主要金鑰提供者， AWS CLI 則不需要。

若要 AWS KMS keys 與 AWS 加密 CLI 搭配使用，您需要安裝並設定 AWS CLI. 此組態可讓您用來驗證的認證 AWS KMS 供加 AWS 密 CLI 使用。

安裝和更新加 AWS 密 CLI

安裝最新版本的 AWS 加密 CLI。當您使用安裝pip加 AWS 密 CLI 時，它會自動安裝 CLI 所需的程式庫，包括 Python 密碼編譯程式庫和. 適用於 Python 的 AWS Encryption SDKAWS SDK for Python (Boto3)

> **Note**
>
> 早於 4.0.0 的 AWS 加密 CLI 版本處於此end-of-support階段。
> 您可以從 2.1 版安全地更新。 x 和更新版本的 AWS 加密 CLI 的最新版本，無需任何代碼或數據更改。但是，2.1 版中引入了新的安全功能。 x 不向後相容。要從 1.7 版本更新。 x 或更早版本，您必須先更新到最新的 1。 x 版本的 AWS 加密 CLI。如需詳細資訊，請參閱 遷移您的 AWS Encryption SDK。
> 新的安全功能最初在 AWS 加密 CLI 版本 1.7 中發布。 X 和 2.0。 x. 但是， AWS 加密 CLI 版本 1.8。 x 取代了 1.7 版本。 x 和 AWS 加密 CLI 碼 2.1. x 取代了 2.0。 x. 如需詳細資訊，請參閱的aws-encryption-sdk-cli儲存庫中的相關安全性建議 GitHub。

安裝最新版本的 AWS 加密 CLI

```
pip install aws-encryption-sdk-cli
```

升級至最新版本的 AWS 加密 CLI

```
pip install --upgrade aws-encryption-sdk-cli
```

查找加 AWS 密 CLI 的版本號和 AWS Encryption SDK

```
aws-encryption-cli --version
```

輸出會列出兩個程式庫的版本號碼。

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

升級至最新版本的 AWS 加密 CLI

```
pip install --upgrade aws-encryption-sdk-cli
```

安裝加 AWS 密 CLI 也會安裝最新版本的 AWS SDK for Python (Boto3)(如果尚未安裝)。如果安裝了 Boto3，安裝程序會驗證 Boto3 版本並在需要時更新它。

若要尋找您已安裝的 Boto3 版本

```
pip show boto3
```

要更新到最新版本的肉毒桿菌 3

```
pip install --upgrade boto3
```

若要安裝目前正在開發中的 AWS 加密 CLI 版本，請參閱上的aws-encryption-sdk-cli存放庫 GitHub。

如需使用 `pip` 安裝與升級 Python 套件的詳細資訊，請參閱 pip 文件。

如何使用加AWS密 CLI

本主題說明如何使用AWS加密 CLI 中的參數。如需範例，請參閱示例AWS加密 CLI。如需完整的文件，請參閱閱讀相關文件。這些範例中顯示的語法適用於AWS加密 CLI 2.1 版。 x 及更新版本。

> **Note**
>
> 早於 4.0.0 的AWS加密 CLI 版本處於 e nd-of-support 階段。
> 您可以從 2.1 版安全地更新。 x 和更新版本的AWS加密 CLI 的最新版本，無需任何代碼或數據更改。但是，2.1 版中引入了新的安全功能。 x 不向後相容。要從 1.7 版本更新。 x 或更早版本，您必須先更新到最新的 1。 x 版本的AWS加密 CLI。如需詳細資訊，請參閱 遷移您的 AWS Encryption SDK。

新的安全功能最初在AWS加密 CLI 版本 1.7 中發布。 X 和 2.0。 x. 但是，AWS加密 CLI 版本 1.8。 x 取代了 1.7 版本。 x 和AWS加密指令碼 2.1. x 取代了 2.0。 x. 如需詳細資訊，請參閱的aws-encryption-sdk-cli儲存庫中的相關安全性建議GitHub。

如需示範如何使用限制加密資料金鑰之安全性功能的範例，請參閱限制加密的資料金鑰。

如需展示如何使用AWS KMS多區域金鑰的範例，請參閱使用多地區 AWS KMS keys。

主題

- 如何加密和解密資料
- 如何指定包裝鍵
- 如何提供輸入
- 如何指定輸出位置
- 如何使用加密內容
- 如何指定承諾產品原則
- 如何在組態檔案中存放參數

如何加密和解密資料

加AWS密 CLI 使用的功能，可AWS Encryption SDK輕鬆安全地加密和解密資料。

> **Note**
>
> 該`--master-keys`參數在 1.8 版中已棄用。 AWS加密 CLI 的 x，並在 2.1 版中刪除。 x. 請改用 `--wrapping-keys` 參數。從 2.1 版開始。 x，加密和解密時需要該`--wrapping-keys`參數。如需詳細資訊，請參閱 AWS Encryption SDK CLI 語法和參數參考。

- 當您在AWS加密 CLI 中加密資料時，您可以指定純文字資料和包裝金鑰 (或主要金鑰)，例如 AWS KMS key in AWS Key Management Service (AWS KMS)。如果您使用自訂主要金鑰提供者，您也需要指定提供者。您也可以指定已加密訊息和加密操作相關中繼資料的輸出位置。加密內容是選用的，但建議使用。

  在版本 1.8 中。 x，使用`--commitment-policy`參數時需要`--wrapping-keys`參數；否則無效。從 2.1 版開始。 x，`--commitment-policy`參數是可選的，但建議使用。

```
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
```

加AWS密 CLI 會在唯一的資料金鑰下加密您的資料。然後它加密您指定的包裝鍵下的數據密鑰。它會傳回已加密訊息和操作的相關中繼資料。已加密訊息包含加密的資料 (加密文字) 和資料金鑰的已加密副本。您不需要擔心存放和管理問題，或是遺失資料金鑰。

- 解密資料時，傳入已加密訊息、選用的加密內容，以及純文字輸出和中繼資料的位置。您也可以指定AWS加密 CLI 可用來解密訊息的包裝金鑰，或告訴AWS加密 CLI 它可以使用加密訊息的任何包裝金鑰。

  從版本 1.8 開始。 x，解密時該`--wrapping-keys`參數是可選的，但建議使用。從 2.1 版開始。x，加密和解密時需要該`--wrapping-keys`參數。

  解密時，您可以使用`--wrapping-keys`參數的 key 屬性來指定解密資料的包裝金鑰。在解密時指定AWS KMS包裝金鑰是選擇性的，但最佳作法可防止您使用不想使用的金鑰。如果您使用自訂主要金鑰提供者，則必須指定提供者和包裝金鑰。

  如果您不使用金鑰屬性，則必須將`--wrapping-keys`參數的探索屬性設定為`true`，如此可讓 En AWS cryption CLI 使用加密訊息的任何包裝金鑰來解密。

  最佳作法是使用`--max-encrypted-data-keys`參數來避免解密含有過多加密資料金鑰的格式錯誤訊息。指定預期的加密資料金鑰數目 (加密中使用的每個包裝金鑰各一個) 或合理的最大值 (例如 5)。如需詳細資訊，請參閱 限制加密的資料金鑰。

  只有在處理完所有輸入之後，`--buffer`參數才會傳回純文字，包括驗證數位簽章 (如果有的話)。

  該`--decrypt-unsigned`參數解密文本，並確保消息在解密之前未簽名。如果您使用參數並選取沒有`--algorithm`數位簽章的演算法套件來加密資料，請使用此參數。如果密文已簽署，解密會失敗。

  您可以使用`--decrypt`或`--decrypt-unsigned`進行解密，但不能同時使用兩者。

```
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 使用包裝金鑰來解密加密訊息中的資料金鑰。接著使用資料金鑰來解密您的資料。它會傳回您的純文字資料和操作的相關中繼資料。

如何指定包裝鍵

在加密 CLI 中AWS加密資料時，您需要至少指定一個包裝金鑰 (或主要金鑰)。您可以使用 AWS KMS keys in AWS Key Management Service (AWS KMS)、從自訂主金鑰提供者包裝金鑰，或兩者都使用。自訂主金鑰提供者可以是任何相容的 Python 主金鑰提供者。

要在版本 1.8 中指定包裝鍵。 x 及更高版本，使用`--wrapping-keys`參數 (`-w`)。此參數的值是具有`attribute=value`格式的屬性集合。您使用的屬性取決於主金鑰提供者和命令。

- AWS KMS。 在加密命令中，您必須指定帶有密鑰屬性的`--wrapping-keys`參數。從 2.1 版開始。 x，在解密命令中也需要`--wrapping-keys`參數。解密時，`--wrapping-keys`參數必須具有值為`true` (但不能兩者) 的索引鍵屬性或探索屬性。其他屬性是可選的。
- 自訂主金鑰提供者。您必須在每個指令中指定一個`--wrapping-keys`參數。參數值必須擁有 key 和 provider 屬性。

您可以在同一指令中包括多個`--wrapping-keys`參數和多個關鍵屬性。

包裝關鍵參數屬性

`--wrapping-keys` 參數的值包含下列屬性以及其值。所有加密指令都需要`--master-keys`參數 (或參數)。`--wrapping-keys`從 2.1 版開始。 x，解密時也需要`--wrapping-keys`參數。

如果屬性名稱或值包含空格或特殊字元，請同時用引號括住名稱和值。例如：`--wrapping-keys key=12345 "provider=my cool provider"`。

索引鍵：指定包裝索引鍵

使用 key 屬性來識別包裝鍵。加密時，值可以是主要金鑰提供者可識別的任何金鑰識別碼。

```
--wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
```

在加密命令中，您必須包含至少一個鍵屬性和值。要在多個包裝密鑰下加密數據密鑰，請使用多個密鑰屬性。

```
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，key 的值可以是密鑰 ID，其密鑰 ARN，別名或別名 ARN。例如，此加密命令在 key 屬性的值中使用別名 ARN。有關的密鑰標識符的詳細信息AWS KMS key，請參閱AWS Key Management Service開發人員指南中的密鑰標識符。

```
aws-encryption-cli --encrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:alias/ExampleAlias
```

在使用自訂主金鑰提供者的解密命令中，key 和 provider 屬性是必要的。

```
\\ 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，您可以使用金鑰屬性來指定AWS KMS keys要用於解密的，或使用值為的探索屬性`true`，這可讓 En AWS cryption CLI 使用任何AWS KMS key用來加密訊息的屬性。如果指定AWS KMS key，它必須是用來加密郵件的其中一個包裝金鑰。

指定環繞索引鍵是AWS Encryption SDK最佳作法。它可以確保您使用AWS KMS key要使用的。

在解密命令中，密鑰屬性的值必須是密鑰 ARN。

```
\\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

探索：解密AWS KMS key時使用任何

如果您在AWS KMS keys解密時不需要限制使用，可以使用探索屬性值為。`true`的值`true`允許AWS加密 CLI 使用任何加密訊息AWS KMS key的方式來解密。如果您未指定探查屬性，則探查為`false` (預設值)。探索屬性僅在解密命令中有效，且僅在使用加密郵件時才有效AWS KMS keys。

值為的探索屬性可替代使用索引鍵屬性來指定AWS KMS keys。true解密使用加密的郵件時AWS KMS keys，每個--wrapping-keys參數都必須具有金鑰屬性或探索屬性值為 (但不能同時使用兩者)。true

當探索為 true 時，最佳作法是使用探索分割區和探索帳戶屬性，將使AWS KMS keys用的屬性限制為您指定的屬性。AWS 帳戶在下列範例中，探查屬性允許加AWS密 CLI 使用指定AWS KMS key的任何屬性AWS 帳戶。

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

提供者：指定主要金鑰提供者

provider 屬性識別主金鑰提供者。預設值是 aws-kms，代表 AWS KMS。如果您使用不同的主金鑰提供者，則 provider 屬性為必要。

```
--wrapping-keys key=12345 provider=my_custom_provider
```

如需有關使用自訂 (非AWS KMS) 主要金鑰提供者的詳細資訊，請參閱AWS加密 CLI 存放庫的 README 檔案中的進階組態主題。

區域：指定 AWS 區域

使用區域屬性來指定AWS 區域的AWS KMS key。此屬性僅在加密命令中有效，且僅適用於主金鑰提供者是 AWS KMS 時。

```
--encrypt --wrapping-keys key=alias/primary-key region=us-east-2
```

AWS如果金鑰屬性值包含區域 (例如 ARN)，則加密 CLI 命令會使用在金鑰屬性值中指定的。如果金鑰值指定 aAWS 區域，則會忽略該區域屬性。AWS 區域

region 屬性優先於其他區域規格。如果您不使用區域屬性，En AWS cryption CLI 命令會使用您AWS 區域AWS CLI指定的設定檔 (如果有的話) 中指定的，或您的預設設定檔。

Profile：指定命名設定檔

使用 profile 屬性可指定 AWS CLI 命名描述檔。具名的設定檔可以包含認證和AWS 區域. 此屬性僅適用於主金鑰提供者是 AWS KMS 時。

```
--wrapping-keys key=alias/primary-key profile=admin-1
```

您可以使用 profile 屬性來指定加密和解密命令中的備用登入資料。在加密命令中，只有當金鑰值不包含區域且沒有區域屬性時，Encryp AWS tion CLI 才會AWS 區域在指定的設定檔中使用。在解密命令AWS 區域中，會忽略名稱設定檔中的。

如何指定多個包裝鍵

您可以在每個指令中指定多個環繞金鑰 (或主金鑰)。

如果您指定多個環繞金鑰，第一個包裝金鑰會產生並加密用於加密資料的資料金鑰。其他包裝金鑰會加密相同的資料金鑰。產生的加密訊息包含加密的資料 (「密文」) 和加密資料金鑰的集合，每個包裝金鑰一個加密。任何包裝都可以解密一個加密的數據密鑰，然後解密數據。

有兩種方法可以指定多個包裝鍵：

- 在`--wrapping-keys`參數值中包含多個關鍵屬性。

```
$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
```

- 在同一個命令中加入多個 `--wrapping-keys` 參數。當您指定的屬性值不適用於指令中的所有環繞鍵時，請使用此語法。

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

值為的探索屬性`true`可讓AWS加密 CLI 使用任何AWS KMS key加密郵件的屬性。如果您在同一指令中使用多個`--wrapping-keys`參數，則`discovery=true`在任何`--wrapping-keys`參數中使用會有效地取代其他`--wrapping-keys`參數中關鍵屬性的限制。

例如，在下列命令中，第一個`--wrapping-keys`參數中的金鑰屬性會將AWS加密 CLI 限制為指定的AWS KMS key。不過，第二個`--wrapping-keys`參數中的探索屬性可讓AWS加密 CLI 使用指定帳戶AWS KMS key中的任何來解密郵件。

```
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
```

如何提供輸入

加密 CLI 中的AWS加密作業會將純文字資料做為輸入，並傳回加密的訊息。解密操作採用已加密訊息做為輸入，並傳回純文字資料。

在所有AWS加密 CLI 命令中，都需要`--input`參數 (`-i`) 告訴加AWS密 CLI 在哪裡可以找到輸入。

您可以透過以下任何方式來提供輸入：

- 使用檔案。

```
--input myData.txt
```

- 使用檔案名稱模式。

```
--input testdir/*.xml
```

- 使用目錄或目錄名稱模式。當輸入是目錄時，`--recursive` 參數 (`-r`, `-R`) 為必要。

```
--input testdir --recursive
```

- 將輸入輸送到命令 (stdin)。使用 `-` 參數的 `--input` 值。(`--input` 參數一律為必要)。

```
echo 'Hello World' | aws-encryption-cli --encrypt --input -
```

如何指定輸出位置

該`--output`參數告訴AWS加密 CLI 在何處寫入加密或解密操作的結果。每個AWS加密 CLI 命令都需要此功能。加AWS密 CLI 會為操作中的每個輸入文件創建一個新的輸出文件。

如果輸出檔案已存在，預設情況下，AWS加密 CLI 會列印警告，然後覆寫該檔案。若要防止覆寫，請使用 `--interactive` 參數，這會在覆寫前提示您確認；或者 `--no-overwrite`，如果輸出會造成覆寫則略過輸入。若要隱藏覆寫警告，請使用 `--quiet`。若要從AWS加密 CLI 擷取錯誤和警告，請使用2>&1重新導向運算子將它們寫入輸出串流。

Note

覆寫輸出檔案的命令首先會刪除輸出檔案。如果命令失敗，輸出檔案可能已遭到刪除。

您可以透過幾種方法指定輸出位置。

- 指定檔案名稱。如果指定檔案路徑，路徑中的所有目錄都必須存在，命令才能執行。

```
--output myEncryptedData.txt
```

- 指定目錄。執行命令之前，輸出目錄必須存在。

  如果輸入包含子目錄，命令會在指定的目錄之下重新產生子目錄。

```
--output Test
```

  當輸出位置為目錄 (不含檔案名稱) 時，AWS加密 CLI 會根據輸入檔案名稱加上尾碼建立輸出檔案名稱。加密操作會附加 `.encrypted` 到輸入檔案名稱，而解密操作會附加 `.decrypted`。若要變更尾碼，請使用 `--suffix` 參數。

  例如，如果您加密 `file.txt`，加密命令會建立 `file.txt.encrypted`。如果您解密 `file.txt.encrypted`，解密命令會建立 `file.txt.encrypted.decrypted`。

- 寫入命令列 (stdout)。輸入 `-` 參數的 `--output` 值。您可以使用 `--output -`，將輸出輸送到另一個命令或程式。

```
--output -
```

如何使用加密內容

加AWS密 CLI 可讓您在加密和解密命令中提供加密內容。這不是必要項目，但它是我們建議的密碼編譯最佳實務。

加密內容是一種任意、非私密額外驗證資料。在AWS加密 CLI 中，加密內容由`name=value`配對的集合組成。您可以使用此配對中的任何內容，包括檔案的相關資訊、可協助您在日誌中尋找加密操作的資料，或者您授予或政策要求的資料。

在加密命令中

您在加密命令中指定的加密內容，以及 CMM 新增的任何額外配對，將以密碼編譯的方式繫結至加密的資料。它也會納入命令傳回的已加密訊息中 (以純文字形式)。如果您使用的是AWS KMS key，加密內容也可能會以純文字顯示在稽核記錄和記錄中，例如。AWS CloudTrail

以下範例顯示使用三個 `name=value` 配對的加密內容。

```
--encryption-context purpose=test dept=IT class=confidential
```

在解密命令中

在解密命令中，加密內容可協助您確認您正在解密正確的已加密訊息。

即使加密時有使用加密內容，您也不需要在解密命令中提供加密內容。不過，如果您這麼做，En AWS cryption CLI 會驗證解密命令的加密內容中的每個元素是否符合加密訊息的加密內容中的元素。如果沒有相符元素，解密命令會失敗。

例如，以下命令只有在加密內容包含 `dept=IT` 時，才會解密已加密訊息。

```
aws-encryption-cli --decrypt --encryption-context dept=IT ...
```

加密內容是安全策略的重要部分。不過，在選擇加密內容時，請記住它的值不是秘密。請勿在加密內容中包含任何機密資料。

指定加密內容

- 在 encrypt 命令中，使用 `--encryption-context` 參數搭配一或多個 `name=value` 對組。使用空格來分隔每個對組。

  ```
  --encryption-context name=value [name=value] ...
  ```

- 在 decrypt 命令中，`--encryption-context` 參數值可以包含 `name=value` 對組、`name` 元素 (沒有值)，或兩者的組合。

  ```
  --encryption-context name[=value] [name] [name=value] ...
  ```

如果 `name` 對組中的 `value` 或 `name=value` 包含空格或特殊字元，請用引號括住整個對組。

```
--encryption-context "department=software engineering" "AWS ##=us-west-2"
```

例如，此加密命令包含使用兩個對組 (purpose=test 和 dept=23) 的加密內容。

```
aws-encryption-cli --encrypt --encryption-context purpose=test dept=23 ...
```

這些解密命令可以成功。每個命令中的加密內容是原始加密內容的子集。

```
\\ 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 ...
```

不過，這些解密命令會失敗。已加密訊息的加密內容不包含指定的元素。

```
aws-encryption-cli --decrypt --encryption-context dept=Finance ...
aws-encryption-cli --decrypt --encryption-context scope ...
```

如何指定承諾產品原則

若要設定命令的承諾原則，請使用--commitment-policy參數。此參數在 1.8 版中引入。 x. 它在加密和解密命令中是有效的。您設定的承諾原則僅對其顯示的命令有效。如果您未為命令設定承諾原則，則加AWS密 CLI 會使用預設值。

例如，下列參數值會將履約承諾原則設定為require-encrypt-allow-decrypt，該原則一律會使用金鑰承諾進行加密，但會將加密的加密文字 (含或不使用金鑰承諾使用)。

```
--commitment-policy require-encrypt-allow-decrypt
```

如何在組態檔案中存放參數

您可以將常用的AWS加密 CLI 參數和值儲存在組態檔案中，以節省時間並避免輸入錯誤。

組態檔是一個文字檔，其中包含AWS加密 CLI 命令的參數和值。當您參照AWS加密 CLI 命令中的組態檔時，參照會被組態檔中的參數和值取代。其效果如同您在命令列中輸入檔案內容。組態檔案可使用任何名稱、位於目前使用者可存取的任何目錄中。

下面的示例配置文件,key.conf, 指定了兩個不同AWS KMS keys的區域。

```
--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
```

若要在命令中使用組態檔案，請在檔案名稱前加上 @ 符號 (@)。在PowerShell控制台中，使用反引號字符轉義 at 符號（`` `@ ``）。

此範例命令在加密命令中使用 `key.conf` 檔案。

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
```

組態檔案規則

使用組態檔案的規則如下所示：

- 您可以在每個組態檔案中包含多個參數，它們可用任何順序列出。請在不同的行列出每個參數及其值 (如果有)。
- 使用 `#` 可新增註解到所有行或部分行。
- 您可以將參考加入其他組態檔案。不要使用反引號來逃避標@誌，即使在PowerShell。
- 如果您在組態檔案中使用引號，引號內的文字不能跨越多行。

例如，這是範例 `encrypt.conf` 檔案的內容。

```
# 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
```

您也可以在命令中包含多個組態檔案。此範例命令同時使用 encrypt.conf 和 master-keys.conf 組態檔案。

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
```

下一個:嘗試加AWS密 CLI 範例

示例AWS加密 CLI

使用下列範例來試用AWS在您選用的平台上加密 CLI。如需主金鑰和其他參數的說明，請參閱如何使用加AWS密 CLI。如需快速參考，請參閱 AWS Encryption SDK CLI 語法和參數參考。

> Note
>
> 下列範例會使用AWSCLI 2.1 版加密。x。
> 新的安全功能最初在AWSCLI 1.7 版加密。x和 2.0。x。但是,AWS1.8 版加密 CLI 版。x替換 1.7 版。x和AWSCLI 2.1 加密。x來替換 2.0。x。如需詳細資訊，請參相關的安全建議中的aws-encryption-sdk-cli上的儲存庫GitHub。

有關如何使用限制加密數據密鑰的安全功能的示例，請參閱限制加密的資料金鑰。

有關演示如何使用AWS KMS多區域金鑰，請參使用多地區 AWS KMS keys。

主題

- 加密檔案
- 解密檔案
- 加密目錄中的所有檔案
- 解密目錄中的所有檔案

- 在命令列上加密和解密
- 使用多個主密鑰
- 在指令碼中加密和解密
- 使用資料金鑰快取

加密檔案

此範例使用AWS使用加密 CLI 來加密`hello.txt`檔案，其中包含「Hello World」字串。

在文件上運行加密命令時，AWS加密 CLI 會取得檔案的內容，產生唯一的資料金鑰，加密數據密鑰下的文件內容，然後寫入加密訊息至新檔案。

第一個命 ARN 會將AWS KMS key中的`$keyArn`變數。當使用AWS KMS key，您可以通過使用金鑰 ID、金鑰 ARN、別名名稱或別名 ARN 來識別它。如需相關金鑰標識符的詳細信息，AWS KMS key，請參密鑰標識符中的AWS Key Management Service開發人員指南。

第二個命令會加密檔案內容。此命令會使用 `--encrypt` 參數來指定操作和 `--input` 參數，以指示需要加密的檔案。`--wrapping-keys` 參數加上其必要的 key 屬性，便可通知該命令要使用由金鑰 ARN 顯示的 AWS KMS key。

此命令會使用 `--metadata-output` 參數，指定用於加密操作相關中繼資料的文字檔案。根據最佳實務，此命令會使用 `--encryption-context` 參數來指定加密細節。

此命令還使用`--commitment-policy`參數以明確設定承諾政策。1.8 版中的版本。x，則當您使用`--wrapping-keys`參數。從 2.1 版開始。x，`--commitment-policy`參數為選用參數，但建議使用。

`--output` 參數的值，也就是點 (.)，則會通知此命令要將輸出檔寫入目前的目錄。

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 .
```

當執行成功時，加密命令並不會傳回任何輸出。若要判斷命令是否執行成功，請查看 $? 變數的布林值。當命令執行成功時，$?是`0`（Bash）或`True`(PowerShell。命令失敗時，會使用$?為非零（Bash）或`False`(PowerShell。

Bash

```
$ echo $?
0
```

PowerShell

```
PS C:\> $?
True
```

您也可以使用目錄列出命令，查看加密命令是否建立了新的檔案 `hello.txt.encrypted`。由於這個加密命令不會指定輸出的檔名，AWS加密 CLI 會將輸出寫入與輸入檔同名的檔案，另加`.encrypted`後置詞。若要使用不同的尾碼或隱藏尾碼，請使用 `--suffix` 參數。

`hello.txt.encrypted` 檔案包含了加密的訊息，當中包含 `hello.txt` 檔案的加密文字、資料金鑰的加密副本，以及包括加密細節的額外中繼資料。

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
```

解密檔案

此範例使用AWS加密 CLI 來解密`Hello.txt.encrypted`檔案，這個檔案在上述範例中已加密。

此解密命令會使用 `--decrypt` 參數來指示操作和 `--input` 參數，以確認需要解密的檔案。`--output` 參數的值是一個點，即代表目前的目錄。

所以此`--wrapping-keys`使用的參數鍵屬性指定用來解密已加密訊息的包裝金鑰。在解密命令中使用AWS KMS keys，則關鍵屬性的值必須是金鑰 ARN。所以此`--wrapping-keys`參數在解密命令中參數為必要項目。如果您使用AWS KMS keys，您可以使用鍵來指定的屬性AWS KMS keys用於解密，或者發現屬性的值為`true`(但不是兩者). 如果使用自訂主金鑰提供者，則會使用鍵和供應商屬性是必要屬性。

所以此`--commitment-policy`參數是可選的，從 2.1 版開始。x，但建議使用。使用它明確表示您的意圖，即使您指定了默認值，`require-encrypt-require-decrypt`。

`--encryption-context` 參數在解密命令中屬於選用性，即使加密命令中已有提供加密細節。遇到這種情況時，解密命令會使用加密命令所提供的相同加密細節。在解密之前，AWS加密 CLI 會驗證已加密訊息中的加密細節是否包含`purpose=test`對。如果沒有包含，解密命令執行就會失敗。

`--metadata-output` 參數會指定用於解密操作相關中繼資料的檔案。`--output` 參數的值，也就是點 (.)，則會將輸出檔寫入目前的目錄。

根據最佳實務，請使用`--max-encrypted-data-keys`參數，以避免使用過多的加密數據密鑰對格式錯誤的消息進行解密。指定預期的加密數據密鑰數量（加密中使用的每個包裝密鑰一個）或合理的最大數量（如 5 個）。如需詳細資訊，請參閱 限制加密的資料金鑰。

所以此`--buffer`只有在處理所有輸入後才返回明文，包括驗證數字簽名（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 .
```

當執行成功時，解密命令並不會傳回任何輸出。若要判斷命令是否執行成功，請取得 $? 變數的值。您也可以使用目錄列出命令，查看此命令是否建立了另加尾碼 `.decrypted` 的新檔案。若要查看純文字內容，請使用命令以取得該檔案內容，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
```

加密目錄中的所有檔案

此範例使用AWS加密 CLI，加密目錄中所有檔案的內容。

當命令會影響到多個檔案時，AWS加密 CLI 分別處理每個文件。它會取得檔案內容、從主金鑰取得該檔案的唯一資料金鑰、根據該資料金鑰來加密檔案內容，接著將結果寫入在輸出目錄中的新檔案。因此，您可以獨立解密處理輸出檔。

這份 `TestDir` 目錄清單顯示了我們要加密的純文字檔案。

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
```

第一個命令保存Amazon Resource Name (ARN)的AWS KMS key中的`$keyArn`變數。

第二個命令會加密在 `TestDir` 目錄中之檔案的內容，並將加密細節的檔案寫入 `TestEnc` 目錄。如果該 `TestEnc` 目錄不存在，命令執行就會失敗。由於輸入位置是一個目錄，所以必須使用 `--recursive` 參數。

所以此`--wrapping-keys`參數，及其所需鍵屬性中，指定要使用的包裝鍵。此加密命令會包含加密細節、`dept=IT`。當您在執行加密多個檔案的加密命令中指定某加密細節時，所有檔案都會使用該相同加密細節。

該命令還具有`--metadata-output`參數來告訴AWS加密 CLI 將要寫入加密操作相關中繼資料的位置。所以此AWS加密 CLI 會為每個已加密的檔案編寫中繼資料記錄。

所以此`--commitment-policy parameter`是可選的，從 2.1 版開始。x，但建議使用。如果命令或腳本因無法解密密文而失敗，則顯式承諾策略設置可能有助於您快速檢測問題。

命令完成時，會使用AWS加密 CLI 將加密文件寫入`TestEnc`目錄，但它不會傳回任何輸出。

最後的命令會列出 `TestEnc` 目錄中的檔案。每個純文字內容的輸入檔案，都會有一個加密細節的輸出檔案。由於此命令沒有指定替代尾碼，所以加密命令會在每個輸入檔名後面加上 `.encrypted`。

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
```

解密目錄中的所有檔案

這個範例會解密目錄中的所有檔案。範例一開始是先處理位在 TestEnc 目錄中，先前範例所加密的檔案。

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
```

這個解密命令會解密TestEnc目錄，並將明文文件寫入TestDec目錄。所以此`--wrapping-keys`使用的參數鍵和一個屬性金鑰 ARN值告訴AWS加密 CLIAWS KMS keys來解密這些文件。此命令使用`--interactive`參數來告訴AWS在覆寫同名檔案之前，加密 CLI 會提示您。

此命令也會使用在先前加密檔案時所提供的加密細節。若是要解密多個檔案，會使用AWS加密 CLI 會檢查每個檔案的加密細節。如果對任何文件進行加密上下文檢查失敗，AWS加密 CLI 會拒絕該檔案、編寫警告、在中繼資料中記錄該次失敗，接著繼續檢查其餘檔案。如果AWS由於任何其他原因，加密 CLI 無法解密檔案，則整個解密命令會立刻失敗。

在這個範例中，所有輸入檔中的已加密訊息都會包含 `dept=IT` 加密細節元素。不過，如果要解密的訊息採用不同的加密細節，這時您應該還是可以驗證部分的加密細節。例如，如果某些訊息包含 `dept=finance` 的加密細節，而其他訊息包含的是 `dept=IT`，這時您不用指定該值，就能驗證加密細節是否一直包含 `dept` 名稱。如果您想要設定更多限定，您可以使用個別命令來解密這些檔案。

解密命令不會傳回任何輸出，但您可以使用目錄列出命令，查看其是否建立了另加 `.decrypted` 尾碼的新檔案。若要查看純文字內容，請使用命令以取得該檔案內容。

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
```

在命令列上加密和解密

這些範例會示範如何將輸入輸送到命令 (stdin)，以及將輸出寫入命令列 (stdout)。範例會說明如何在命令中表示 stdin、stdout，以及如何使用內建的 Base64 編碼工具防止 shell 錯誤解譯非 ASCII 字元。

這個範例會將純文字字串輸送到加密命令，並將加密的訊息儲存到變數中。然後，它會將變數中的已加密訊息輸送到解密命令，再由該命令將其輸出寫入到管道 (stdout)。

範例中包含了三種命令：

- 第一個命令保存金鑰 ARN的AWS KMS key中的`$keyArn`變數。

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'
```

- 第二個命令會將 `Hello World` 字串輸送到加密命令，並將執行結果儲存到 `$encrypted` 變數中。

  所以此`--input`和`--output`中的所有參數都是必要參數AWS加密 CLI 命令。若要指示輸入要輸送到命令 (stdin)，- 參數的值應使用連字號 (`--input`)。若要將輸出傳送到命令列 (stdout)，`--output` 參數的值應使用連字號。

  `--encode` 參數會先對輸出進行 Base64 編碼，再將其傳回。這樣可以防止 shell 錯誤解譯已加密訊息中的非 ASCII 字元。

  由於這個命令只是為了概念驗證，所以我們會省略加密細節，並且隱藏中繼資料 (`-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
```

- 第三個命令會將 `$encrypted` 變數中的已加密訊息輸送到解密命令。

這個解密命令會使用 `--input -`，指示輸入會由該管道送入 (stdin)，而且使用 `--output -` 將輸出傳送到該管道 (stdout)。(輸入參數接收的是輸入的位置，而非實際的輸入位元組，因此您不能使用 `$encrypted` 變數做為 `--input` 參數值)。

此範例使用發現的屬性`--wrapping-keys`參數以允許AWS加密 CLI 以使用任何AWS KMS key來解密資料。它沒有指定承諾政策，因此它使用版本 2.1 的默認值。x和更高版本,`require-encrypt-require-decrypt`。

由於輸出是經過加密後再進行編碼，所以解密命令會先使用 `--decode` 參數來解碼經 Base64 編碼處理的輸入，接著再進行解密。您也可以先使用 `--decode` 參數，為經 Base64 編碼處理的解碼，接著再進行加密。

同樣地，這個命令會省略加密細節，並隱藏中繼資料 (-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
```

您也可以運用單一個命令來執行加密和解密操作，完全不用中斷變數。

在上述範例中，`--input` 和 `--output` 參數具有 - 值，而此命令會使用 `--encode` 參數來為輸出進行編碼，並使用 `--decode` 參數來為輸入進行解碼。

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
```

使用多個主密鑰

這個範例會示範如何在運用進行資料的加密及解密時使用多個主金鑰。AWS加密 CLI。

如果您是使用多個主金鑰來加密資料，則其中任何一個主金鑰都可用來為資料進行解密。這個策略可確保您一定可以解密資料，即使其中一個主金鑰發生不可用的情況。如果要將加密數據存儲在多個AWS區域，則通過這個策略，您就能使用位在相同區域中的主金鑰解密資料。

當您使用多個主金鑰來進行加密時，第一個主金鑰會扮演特殊的角色。它會產生將在資料加密時所用到的資料金鑰。其餘的主金鑰則加密處理純文字的資料金鑰。結果產生的加密的訊息，包含了該已加密資料和已加密資料金鑰的集合，每則訊息會對應到個別主金鑰。雖然第一個主金鑰會產生資料金鑰，但其中任何一個主金鑰都可以解密處理任何一個可用來解密處理資料的資料金鑰。

使用三個主金鑰加密

此示例命令使用三個包裝密鑰來加密`Finance.log`文件，三個AWS 區域。

它會將加密的訊息寫入到 `Archive` 目錄。此命令會使用 `--suffix` 參數，且不指定隱藏尾碼的參數值，因此輸入和輸出檔的名稱都是相同的。

此命令會使用 `--wrapping-keys` 參數，並指定三個 key 屬性。您也可以在相同的命令中使用多個 `--wrapping-keys` 參數。

要加密日誌文件，AWS加密 CLI 詢問列表中的第一個包裝密鑰$key1，以產生它用來加密資料的資料金鑰。接著，它會使用其他的每個包裝金鑰，加密相同資料金鑰的純文字副本。在輸出檔案中的已加密訊息，包含了全部三個的已加密資料金鑰。

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
```

這個命令會解密處理已加密的 `Finance.log` 檔案副本，並將其寫入 `Finance.log.clear` 目錄中的 `Finance` 檔案。若要解密已加密在三個範圍內的資料AWS KMS keys，則可以指定相同的三個AWS KMS keys或其中的任何子集。這個範例會指定其中一個AWS KMS keys。

要告訴AWS加密 CLIAWS KMS keys來解密您的資料，請使用鍵的屬性`--wrapping-keys`參數。使用解密時AWS KMS keys的值，鍵屬性必須是金鑰 ARN。

您必須許可才能調用解密 API在AWS KMS keys您指定。如需詳細資訊，請參閱「」的身份驗證與存取控制AWS KMS。

根據最佳實務，這個範例會使用--max-encrypted-data-keys參數，以避免使用過多的加密數據密鑰對格式錯誤的消息進行解密。即使此示例只使用一個包裝密鑰進行解密，但加密消息具有三 (3) 個加密數據密鑰；加密時使用的三個包裝密鑰中的每個密鑰一個。指定預期的加密數據密鑰數量或合理的最大值，如 5。如果指定的最大值小於 3，則命令將失敗。如需詳細資訊，請參閱 限制加密的資料金鑰。

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
```

在指令碼中加密和解密

此範例示範如何使用AWS腳本中的 CLI 加密。您可以編寫只要加密和解密資料的指令碼，或者在資料管理程序中負責加密或解密操作的指令碼。

在這個範例中，腳本會取得包含日誌檔的集合、壓縮這些檔案、加密這些檔案，接著將其複製到 Amazon S3 儲存儲段。這段指令碼會獨立處理個別檔案，所以這些檔案可以單獨地進行解密和展開。

在壓縮和加密檔案時，請務必先完成壓縮，再進行加密。正確完成加密的資料不能進行壓縮。

> **Warning**
>
> 壓縮資料可能包含由惡意人士所操控的秘密和資料，請務必小心。壓縮資料的最終大小，可能會不當透露出與其內容有關的敏感資訊。

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"
                    }
                }
            }
        }
    }
}
```

使用資料金鑰快取

這個範例會在加密處理大量檔案的命令中使用資料金鑰快取。

預設情況下，AWS加密 CLI（以及其他版本的AWS Encryption SDK) 會為其加密的每個檔案產生唯一的資料金鑰。雖然最佳加密實務是為每筆操作使用唯一的資料金鑰，但在某些情況下，仍可接受特定的資料金鑰重複使用。如果您考慮使用資料金鑰快取，請向安全性工程師諮詢實際應用上的安全性需求，並且決定適合您的安全性閾值。

在這個範例中，資料金鑰快取因為減少了向主金鑰提供者提出請求的頻率，使得加密操作速度加快。

在這個範例中的命令會加密處理包含多個子目錄的大型目錄，其中包含總共大約 800 個小型日誌檔。第一個命令會將 AWS KMS key 的 ARN 儲存在 `keyARN` 變數中。第二個命令會加密處理輸入目錄中的所有檔案，並將其寫入封存目錄。這個命令會使用 `--suffix` 參數來指定 `.archive` 尾碼。

`--caching` 參數會啟用資料金鑰快取。負責限制快取中資料金鑰數量的 capacity 屬性則設為 1，因為序列式檔案處理時，一次一律使用一個資料金鑰。負責決定已快取金鑰可以使用多久的 max_age 屬性則設為 10 秒鐘。

選用的 max_messages_encrypted 屬性則設為 10 則訊息，所以在加密處理 10 個以上的檔案時，絕對不會使用單一個資料金鑰。限定每個資料金鑰能夠加密的檔案數目，能在資料金鑰遭洩的難得情況發生時減少受到影響的檔案數量。

若要為作業系統產生的日誌檔執行這個命令，您可能需要具備系統管理員權限 (Linux 的 `sudo`；Windows 的 Run as Administrator (以系統管理員身分執行))。

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
```

為了測試資料金鑰緩存的效果，這個範例會使用測量-命令中的 cmdletPowerShell。若執行此範例時不執行資料金鑰快取功能，完成需時大約 25 秒鐘。這個程序會為目錄中的每個檔案產生新的資料金鑰。

```
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
```

資料金鑰快取能夠加快程序，即使限制每個資料金鑰最多只能處理 10 個檔案。這個命令現在只要不到 12 秒就能完成，因此向主金鑰提供者發出的呼叫次數減少成為原來次數的 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
```

如果刪除 max_messages_encrypted 限制，則所有檔案都會依據相同的資料金鑰進行加密。這項變更會導致重複使用資料金鑰的風險提高，而且程序速度並不會加快。不過，它能將呼叫主金鑰提供者的次數縮減為 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 語法和參數參考

本主題提供語法圖表和概要參數描述，以協助您使用 AWS Encryption SDK 命令列界面 (CLI)。如需包裝鍵和其他參數的說明，請參閱如何使用加AWS密 CLI。如需範例，請參閱示例AWS加密 CLI。如需完整的文件，請參閱閱讀相關文件。

主題

- AWS加密 CLI 語法
- AWS加密 CLI 命令列參數
- 進階參數

AWS加密 CLI 語法

這些AWS加密 CLI 語法圖表顯示您使用AWS加密 CLI 執行的每個工作的語法。它們代表AWS加密 CLI 2.1 版中的建議語法。 x 及更新版本。

新的安全功能最初在AWS加密 CLI 版本 1.7 中發布。 X 和 2.0。 x. 但是，AWS加密 CLI 版本 1.8。 x 會取代 1.7 版。 x 和AWS加密指令碼 2.1. x 取代了 2.0。 x. 如需詳細資訊，請參閱的aws-encryption-sdk-cli儲存庫中的相關安全性建議 GitHub。

> **Note**
>
> 除非在參數描述中註明，否則每個參數或屬性只能在每個命令中使用一次。
> 如果您使用參數不支援的屬性，則AWS加密 CLI 會忽略該不支援的屬性，而不會出現警告或錯誤。

取得說明

若要取得具有參數說明的完整AWS加密 CLI 語法，請使用`--help`或`-h`。

```
aws-encryption-cli (--help | -h)
```

取得版本

若要取得AWS加密 CLI 安裝的版本號碼，請使用`--version`。當您提出問題、回報問題或分享有關使用AWS加密 CLI 的提示時，請務必包含該版本。

```
aws-encryption-cli --version
```

加密資料

下列語法圖表顯示 encrypt 命令使用的參數。

```
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]
```

解密資料

下列語法圖表顯示 decrypt 命令使用的參數。

在 1.8 版中。 x，解密時該`--wrapping-keys`參數是可選的，但建議使用。從 2.1 版開始。 x，加密和解密時需要該`--wrapping-keys`參數。對於AWS KMS keys，您可以使用 key 屬性來指定包裝金鑰 (最佳作法) 或將探索屬性設定為`true`，這不會限制 EnAWS cryption CLI 可以使用的包裝金鑰。

```
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]
```

使用組態檔案

您可以參考包含參數及其值的組態檔案。這相當於在命令中輸入參數和值。如需範例，請參閱 如何在組態檔案中存放參數。

```
aws-encryption-cli @<configuration_file>

# In a PowerShell console, use a backtick to escape the @.
aws-encryption-cli `@<configuration_file>
```

AWS加密 CLI 命令列參數

此清單提供AWS加密 CLI 命令參數的基本說明。如需完整說明，請參閱aws-encryption-sdk-cli文件。

--encrypt (-e)

加密輸入資料。每個命令都必須有一個`--encrypt`、或`--decrypt`、或`--decrypt-unsigned`參數。

--decrypt (-d)

解密輸入資料。每個命令都必須具有`--encrypt--decrypt`、或`--decrypt-unsigned`參數。

-解密-無簽名 [在版本 1.9 中引入。 X 和 2.2. x]

該`--decrypt-unsigned`參數解密文本，並確保消息在解密之前未簽名。如果您使用參數並選取沒有`--algorithm`數位簽章的演算法套件來加密資料，請使用此參數。如果密文已簽署，解密會失敗。

您可以使用`--decrypt`或`--decrypt-unsigned`進行解密，但不能同時使用兩者。

-包裝鍵（-w）[在版本 1.8 中引入。 x]

指定用於加密和解密作業的包裝金鑰 (或主金鑰)。您可以在每個指令中使用多個`--wrapping-keys`參數。

從 2.1 版開始。 x，`--wrapping-keys`參數在加密和解密命令中是必需的。在 1.8 版中。 x，加密命令需要一個`--wrapping-keys`或`--master-keys`參數。在 1.8 版中。 x 解密命令，`--wrapping-keys`參數是可選的，但建議使用。

使用自訂主要金鑰提供者時，加密和解密命令需要金鑰和提供者屬性。使用時AWS KMS keys，加密命令需要密鑰屬性。解密命令需要金鑰屬性或值為`true` (但不能兩者) 的探索屬性。解密時使用金鑰屬性是AWS Encryption SDK最佳作法。如果您要解密一批不熟悉的訊息 (例如 Amazon S3 儲存貯體或 Amazon SQS 佇列中的訊息)，這一點尤為重要。

如需展示如何使用AWS KMS多區域金鑰作為環繞金鑰的範例，請參閱使用多地區 AWS KMS keys。

屬性：`--wrapping-keys` 參數的值包含下列屬性。格式是 `attribute_name=value`。

key

識別作業中使用的包裝金鑰。格式是 key=ID 對組。您可以在每個`--wrapping-keys`參數值中指定多個關鍵屬性。

- 加密命令：所有加密命令都需要密鑰屬性。當您AWS KMS key在加密命令中使用時，金鑰屬性的值可以是金鑰 ID、金鑰 ARN、別名 ARN、別名 ARN。如需AWS KMS金鑰識別碼的說明，請參閱AWS Key Management Service開發人員指南中的金鑰識別碼。
- 解密命令：使用解密時AWS KMS keys，`--wrapping-keys`參數需要具有金鑰 ARN 值的金鑰屬性，或具有值`true` (但不能兩者) 的探索屬性。使用 key 屬性是AWS Encryption SDK最佳做法。使用自訂主要金鑰提供者進行解密時，需要金鑰屬性。

Note

若要在解密命令中指定AWS KMS包裝金鑰，金鑰屬性的值必須是金鑰 ARN。如果您使用金鑰 ID、別名名稱或別名 ARN，AWS加密 CLI 不會識別名稱或別名 ARN。

您可以在每個`--wrapping-keys`參數值中指定多個關鍵屬性。但是，參數中的任何提供者、區域和設定檔屬性都會套用至該`--wrapping-keys`參數值中的所有包裝鍵。若要使用不同的屬性值指定包裝鍵，請在指令中使用多個`--wrapping-keys`參數。

發現

允許加AWS密 CLI 使用任何AWS KMS key來解密訊息。探索值可以是`true`或`false`。預設值為 `false`。探索屬性僅在解密命令中有效，且只有在主要金鑰提供者為時才有效AWS KMS。

使用解密時AWS KMS keys，`--wrapping-keys`參數需要索引鍵屬性或值為`true` (但不能兩者) 的探索屬性。如果您使用 key 屬性，則可以使用值為的探索屬性`false`來明確拒絕探查。

- `False`(預設值) — 未指定探查屬性或其值為時`false`，EnAWS cryption CLI 只會使用`--wrapping-keys`參數金鑰屬性所AWS KMS keys指定的解密訊息。如果您在探索時未指定金鑰屬性`false`，則解密命令會失敗。此值支援加AWS密 CLI 最佳做法。
- `True`— 當探索屬性的值為時`true`，EnAWS cryption CLI 會從加密訊息中的AWS KMS keys中繼資料取得，並使用這些中繼資料AWS KMS keys來解密訊息。值為的探查屬性的`true`行為與 1.8 版之前的AWS加密 CLI 版本相似。 x 在解密時不允許您指定包裝密鑰。但是，您使用 any 的意圖AWS KMS key是明確的。如果您在探查時指定金鑰屬性`true`，則解密指令會失敗。

  該`true`值可能會導致AWS加密 CLIAWS KMS keys 在不同AWS 帳戶的區域中使用，或嘗試使用AWS KMS keys該使用者未獲授權使用。

如果是探索`true`，最佳做法是使用探索分割區和探索帳戶屬性，將使AWS KMS keys用的屬性限制在AWS 帳戶您指定的屬性。

探索帳戶

將AWS KMS keys用於解密的限制在指定的AWS 帳戶。此屬性的唯一有效值是 AWS 帳戶ID。

此屬性是選用的，只有在將探索屬性設AWS KMS keys定為`true`且已指定探索分割區屬性的解密命令中才有效。

每個探索帳戶屬性只需要一個AWS 帳戶 ID，但您可以在相同的`--wrapping-keys`參數中指定多個探索帳戶屬性。指定`--wrapping-keys`參數中指定的所有帳戶都必須位於指定的AWS分割區中。

探索分區

指定探索帳戶屬性中帳戶的AWS分割區。其值必須是AWS分割區，例如`aws``aws-cn`、或`aws-gov-cloud`。如需相關資訊，請參閱 AWS 一般參考.

當您使用探索帳戶屬性時，此屬性是必要的。您只能在每個`--wrapping keys`參數中指定一個探索分割區屬性。若要AWS 帳戶在多個分割區中指定，請使用其他`--wrapping-keys`參數。

provider (提供者)

識別主金鑰提供者。格式是 provider=ID 對組。預設值 aws-kms 代表 AWS KMS。只有當主金鑰提供者不是 AWS KMS 時，此屬性才為必要。

region

識別AWS 區域的AWS KMS key。此屬性僅對AWS KMS keys。僅在 key 識別符未指定區域時才會用到，否則會忽略。使用此屬性時，它會覆寫 AWS CLI 命名設定檔中的預設區域。

profile

識別 AWS CLI 命名描述檔。此屬性僅對AWS KMS keys。只有在命令中的 key 識別符未指定區域，且沒有 region 屬性時，才會使用設定檔中的區域。

--input (-i)

指定加密或解密資料的位置。此為必要參數。這個值可以是檔案或目錄的路徑，或檔案名稱模式。如果您將輸入輸送到命令 (stdin)，請使用 `-`。

如果輸入不存在，命令會順利完成，且不出現錯誤或警告。

--recursive (-r, -R)

在輸入目錄及其子目錄中的檔案上執行操作。當 `--input` 的值是目錄時，此參數為必要。

--decode

解碼 Base64 編碼輸入。

如果您要解密先加密接著編碼的訊息，您必須先解碼訊息，然後才能解密。此參數會為您處理這些工作。

例如，如果您在加密命令中使用 `--encode` 參數，請在對應的解密命令中使用 `--decode` 參數。您也可以使用此參數來解碼 Base64 編碼輸入，接著再進行加密。

--output (-o)

指定輸出的目的地。此為必要參數。這個值可以是檔案名稱、現有目錄，或者 `-`，後者會將輸出寫入命令列 (stdout)。

如果指定的輸出目錄不存在，命令會失敗。如果輸入包含子目錄，則AWS加密 CLI 會在您指定的輸出目錄下重新產生子目錄。

根據預設，加AWS密 CLI 會覆寫具有相同名稱的檔案。若要變更此行為，請使用 `--interactive` 或 `--no-overwrite` 參數。若要隱藏覆寫警告，請使用 `--quiet` 參數。

> **Note**
> 如果覆寫輸出檔案的命令失敗，則會刪除輸出檔案。

--interactive

在覆寫檔案之前出現提示。

--no-overwrite

不要覆寫檔案。相反地，如果輸出檔案存在，則AWS加密 CLI 會略過對應的輸入。

--suffix

為AWS加密 CLI 建立的檔案指定自訂檔案名稱尾碼。若要指示沒有尾碼，請使用參數而不加上值 (`--suffix`)。

在預設情況下，當 `--output` 參數未指定檔案名稱，輸出檔案名稱會具有輸入檔案名稱的相同名稱，再加上尾碼。加密命令的尾碼是 `.encrypted`。解密命令的尾碼是 `.decrypted`。

--encode

套用 Base64 (二進位至文字) 編碼到輸出。編碼可防止殼層主機程式錯誤解譯輸出文字中的非 ASCII 字元。

將加密輸出寫入 stdout (`--output -`) 時，尤其是在 PowerShell 控制台中，即使您要將輸出管道傳送到另一個命令或將其保存在變量中，也可以使用此參數。

--metadata-output

指定密碼編譯操作的相關中繼資料的位置。輸入路徑和檔案名稱。如果目錄不存在，命令會失敗。若要寫入中繼資料至命令列 (stdout)，請使用 `-`。

您不能在相同的命令中寫入命令輸出 (`--output`) 和中繼資料輸出 (`--metadata-output`) 至 stdout。此外，當 `--input` 或 `--output` 的值是目錄 (沒有檔案名稱)，您無法將中繼資料輸出寫入到相同目錄或該目錄的任何子目錄。

如果您指定現有檔案，依預設，AWS加密 CLI 會將新的中繼資料記錄附加至檔案中的任何內容。此功能可讓您建立單一檔案，其中包含所有密碼編譯操作的中繼資料。若要覆寫現有檔案中的內容，請使用 `--overwrite-metadata` 參數。

加AWS密 CLI 會針對命令執行的每個加密或解密作業，傳回 JSON 格式的中繼資料記錄。每個中繼資料記錄都包含輸入和輸出檔案的完整路徑、加密內容、演算法套件和其他有用資訊，供您用來檢視操作並驗證其符合您的安全標準。

--overwrite-metadata

覆寫中繼資料輸出檔案中的內容。在預設情況下，`--metadata-output` 參數會附加中繼資料到檔案中的任何現有內容。

--suppress-metadata (-S)

隱藏加密或解密操作的相關中繼資料。

-承諾政策

指定加密和解密命令的承諾原則。承諾產品原則會判斷您的郵件是否已使用金鑰承諾安全性功能加密和解密。

該`--commitment-policy`參數在 1.8 版中引入。 x. 它在加密和解密命令中是有效的。

在版本 1.8 中。 x，AWS加密 CLI 會針對所有加密和解密作業使用`forbid-encrypt-allow-decrypt`承諾原則。當您在加密或解密命令中使用`--commitment-policy`參數時，需要具有該`forbid-encrypt-allow-decrypt`值的參數。`--wrapping-keys`如果不使用`--wrapping-keys`參數，則`--commitment-policy`參數無效。明確設定承諾產品原則可防止承諾原則在升級至 2.1 版`require-encrypt-require-decrypt`時自動變更為。 x

從 2.1 版開始。 x，支援所有承諾產品原則值。`--commitment-policy`參數是選擇性的，預設值為`require-encrypt-require-decrypt`。

此參數具有以下值：

- forbid-encrypt-allow-decrypt— 無法使用金鑰承諾加密。它可以解密有或沒有密鑰承諾加密的密文。

  在 1.8 版中。 x，這是唯一有效的值。加AWS密 CLI 會針對所有加密和解密作業使用forbid-encrypt-allow-decrypt承諾原則。
- require-encrypt-allow-decrypt— 僅使用關鍵承諾進行加密。無論是否有關鍵承諾，都可以解密。2.1 版會引入此值。 x.
- require-encrypt-require-decrypt(預設值) — 僅使用金鑰承諾進行加密和解密。2.1 版會引入此值。 x. 它是 2.1 版本中的默認值。 x 及更新版本。使用此值時，AWS加密 CLI 將不會解密任何使用AWS Encryption SDK.

如需有關設定承諾產品原則的詳細資訊，請參閱遷移您的AWS Encryption SDK。

--encryption-context (-c)

指定操作的加密內容。此參數非必要，但仍建議使用。

- 在 --encrypt 命令中，輸入一或多個 name=value 對組。使用空格來分隔對組。
- 在--decrypt指令中，輸入name=value配對、沒有值的name元素，或同時輸入兩者。

如果 name 對組中的 value 或 name=value 包含空格或特殊字元，請用引號括住整個對組。例如：--encryption-context "department=software development"。

-緩衝區（-b）[在版本 1.9 中引入。 X 和 2.2. x]

僅在處理完所有輸入之後返回純文本，包括驗證數字簽名（如果存在）。

--max-encrypted-data-keys [在版本 1.9 中引入。 X 和 2.2. x]

指定加密郵件中加密資料金鑰的數目上限。此為選用參數。

有效值為 1 至 65,535。如果忽略此參數，則AWS加密 CLI 不會強制執行任何最大值。加密訊息最多可容納 65,535 (2^16-1) 的加密資料金鑰。

您可以在加密命令中使用此參數，以防止格式錯誤的郵件。您可以在解密命令中使用它來檢測惡意消息，並避免使用無法解密的大量加密數據密鑰解密消息。如需詳細資訊和範例，請參閱限制加密的資料金鑰。

--help (-h)

在命令列印使用方法和語法。

--version

取得AWS加密 CLI 的版本。

-v | -vv | -vvv | -vvvv

顯示詳細資訊、警告和偵錯訊息。輸出中的詳細資訊會隨著參數中的 v 數量而增加。最詳細的設定 (-vvvv) 會從AWS加密 CLI 及其使用的所有元件傳回除錯層級資料。

--quiet (-q)

隱藏警告訊息，例如，當您覆寫輸出檔案時的訊息。

-主鍵 (-m) [已棄用]

Note

--主鍵參數在1.8 中已棄用。 x 並在 2.1 版中刪除。 x. 相反，請使用 -包裝鍵參數。

指定用於加密和解密操作的主金鑰。您可以在每個命令中使用多個主金鑰參數。

在加密命令中 `--master-keys` 參數為必要。只有當您使用自訂 (非AWS KMS) 主要金鑰提供者時，才需要在解密指令中使用此功能。

屬性：`--master-keys` 參數的值包含下列屬性。格式是 `attribute_name=value`。

key

識別作業中使用的包裝金鑰。格式是 key=ID 對組。在所有加密命令中 key 屬性為必要。

當您AWS KMS key在加密命令中使用時，金鑰屬性的值可以是金鑰 ID、金鑰 ARN、別名 ARN、別名 ARN。如需AWS KMS金鑰識別碼的詳細資訊，請參閱AWS Key Management Service開發人員指南中的金鑰識別碼

當主密鑰提供者不是解密命令時，密鑰屬性是必需的AWS KMS。在解密下加密之資料的命令中不允許使用金鑰屬性AWS KMS key。

您可以在每個`--master-keys`參數值中指定多個關鍵屬性。不過，任何 provider、region 和 profile 屬性都會套用至參數值中的所有主金鑰。若要使用不同的屬性值來指定主金鑰，請在命令中使用多個 `--master-keys` 參數。

provider (提供者)

識別主金鑰提供者。格式是 provider=ID 對組。預設值 aws-kms 代表 AWS KMS。只有當主金鑰提供者不是 AWS KMS 時，此屬性才為必要。

region

識別AWS 區域的AWS KMS key。此屬性僅對AWS KMS keys。僅在 key 識別符未指定區域時才會用到，否則會忽略。使用此屬性時，它會覆寫 AWS CLI 命名設定檔中的預設區域。

profile

識別 AWS CLI 命名描述檔。此屬性僅對AWS KMS keys。只有在命令中的 key 識別符未指定區域，且沒有 region 屬性時，才會使用設定檔中的區域。

進階參數

--algorithm

指定替代演算法套件。此參數是選用的，僅在加密命令中有效。

如果您省略此參數，則AWS加密 CLI 會使用 1.8 版中AWS Encryption SDK引入的其中一個預設演算法套件。 x. 這兩種預設演算法都使用 AES-GCM 搭配 HKDF、ECDSA 簽章和 256 位元加密金鑰。一個人使用關鍵承諾; 一個沒有。預設演算法套件的選擇取決於命令的承諾原則。

對於大多數加密操作，建議使用默認算法套件。如需有效值的清單，請參閱 ReName 文件中的`algorithm`參數值。

--frame-length

使用指定的框架長度建立輸出。此參數是選用的，僅在加密命令中有效。

以位元組為單位輸入值。有效值為 0 和 1 — 2 ^ 31-1。0 值表示非框數據。預設值為 4096 (位元組)。

> **Note**
>
> 盡可能使用框架數據。僅AWS Encryption SDK支援舊版使用的非框架資料。的某些語言實作仍然AWS Encryption SDK可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

--max-length

代表從加密訊息讀取的最大框架大小 (或無框架訊息的最大內容長度)，以位元組為單位。此參數是選用的，僅在解密命令中有效。旨在避免您解密非常大型的惡意加密文字。

以位元組為單位輸入值。如果忽略此參數，解密時AWS Encryption SDK不會限制框架大小。

--caching

啟用資料金鑰快取功能，可重複使用資料金鑰，而非為每個輸入檔案產生新的資料金鑰。此參數支援進階案例。使用此功能前，請務必先閱讀資料金鑰快取文件。

`--caching` 參數具有下列屬性。

capacity (必要)

決定快取中的項目數上限。

最小值為 1。沒有最大數值。

max_age (必要)

決定使用快取項目的時間長度 (以秒為單位)，從這些項目新增至快取時間開始。

輸入大於 0 的數值。沒有最大數值。

最大消息 _ 加密（ 可選 ）

決定快取項目可以加密的最大訊息數。

有效值為 1 至 2 ^ 32。預設值為 2^32 (訊息)。

最大字節加密（ 可選 ）

決定快取項目可以加密的最大位元組數。

有效值為 0 和 1 — 2 ^ 63-1。預設值為 2^63-1 (訊息)。值為 0 只允許您在加密空的訊息字串時使用資料金鑰快取。

的版本AWS加密 CLI

我們建議您使用最新版本的AWSCLI。

> **Note**
>
> 的版本AWS4.0.0 版以前的 Encryption 位於end-of-support階段。
> 您可以安全地從 2.1 版更新。x以及更高版本的最新版本AWS加密 CLI，無需任何代碼或數據更改。但是，新的安全性功能2.1 版。x不向後兼容。至 1.7 版。x或更早版本，您必須先更新到最新的 1。x的版本AWSCLI。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

最初發布了新的安全功能AWSEncryption 1.7 版。x及 2.0 及 2.0 通訊x。但是，AWS加密版本 1.8.x取代 1.7 版。x和AWSEncryptionx2.0 通訊x。如需詳細資訊，請參閱相關安全性諮詢在aws-encryption-sdk-cli儲存庫 GitHub。

如需有關重要版本的資訊AWS Encryption SDK，請參閱的版本 AWS Encryption SDK。

我使用哪個版本？

如果您是初次使用AWSCLI，使用最新版本。

若要解密某個版本加密的資料AWS Encryption SDK早於 1.7 版。x，請先遷移至最新版本的AWSCLI。Make所有建議的變更在更新至 2.1 版。x或更高版本。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

進一步了解

- 如需有關移轉至這些新版本的變更和指引的詳細資訊，請參閱遷移您的AWS Encryption SDK。
- 有關新的描述AWS加密 CLI 參數和屬性，請參閱AWS Encryption SDK CLI 語法和參數參考。

下列清單說明對AWS在 1.8 版本中加密 CLI。x及 2.1.x。

1.8 版。x變更的變更AWS加密 CLI

- 棄用的`--master-keys`參數的參數。請改用 `--wrapping-keys` 參數。
- 新增中的新增`--wrapping-keys`(`-w`) 參數的參數。它支持的所有屬性`--master-keys`參數的參數。它還添加了以下可選屬性，這些屬性僅在使用解密時才有效AWS KMS keys。
  - 發現
  - 探索分區
  - 探索帳戶

  對於自訂主金鑰提供者，`--encrypt`和`--decrypt`指令需要`--wrapping-keys`參數或參數`--master-keys`參數（但不是兩者）。另外，一個`--encrypt`使用指令AWS KMS keys需要一個`--wrapping-keys`參數或參數`--master-keys`參數（但不是兩者）。

  在一個`--decrypt`使用指令AWS KMS keys，該`--wrapping-keys`參數是可選的，但建議使用，因為它在 2.1 版本中是必需的。x。如果您使用，您必須擇一指定鍵屬性或發現具有值的屬性`true`（但不能兩者兼而有之）。

- 新增中的新增--commitment-policy參數的參數。唯一有效的值為 forbid-encrypt-allow-decrypt。所以此forbid-encrypt-allow-decrypt承諾政策用於所有加密和解密命令。

  在 1.8 版中。x，當您使用--wrapping-keys參數，一個參數，--commitment-policy參數搭配forbid-encrypt-allow-decrypt值為必要項目。明確設置值可以防止承諾政策從自動變更為require-encrypt-require-decrypt當您升級至 2.1 版。x。

2.1 版。x變更的變更AWS加密 CLI

- 移除中的移除--master-keys參數的參數。請改用 --wrapping-keys 參數。
- 所以此--wrapping-keys所有加密和解密命令都需要參數。您必須擇一指定鍵屬性或一個屬性發現具有值的屬性true（但不能兩者兼而有之）。
- 所以此--commitment-policy參數支援下列值。如需詳細資訊，請參閱 設定承諾產品原則。
  - forbid-encrypt-allow-decrypt
  - require-encrypt-allow-decrypt
  - require-encrypt-require decrypt(預設)
- 所以此--commitment-policy2.1 版中的參數為選用。x。預設值為 require-encrypt-require-decrypt。

1.9 版。x及 2.2.x變更的變更AWS加密 CLI

- 新增中的新增--decrypt-unsigned參數的參數。如需詳細資訊，請參閱 版本 2.2. x。
- 新增中的新增--buffer參數的參數。如需詳細資訊，請參閱 版本 2.2. x。
- 新增中的新增--max-encrypted-data-keys參數的參數。如需詳細資訊，請參閱 限制加密的資料金鑰。

3.0 版。x變更的變更AWSEncryption

- 添加支持AWS KMS多區域金鑰。如需詳細資訊，請參閱使用多地區 AWS KMS keys。

資料金鑰快取

資料金鑰快取會將資料金鑰和相關加密資料全部儲存到快取中。當您加密或解密資料時，會在快取中 AWS Encryption SDK 尋找相符的資料金鑰。如果找到符合的金鑰，它就會使用該快取資料金鑰，而不會產生新的金鑰。資料金鑰快取可以提升效能、降低成本，並且讓您在應用程式規模不斷擴展時維持不超過服務用量。

應用程式在下列條件下能發揮資料金鑰快取優勢：

- 它可以重複使用資料金鑰。
- 它會產生大量的資料金鑰。
- 您的加密操作會異常地變慢速度、成本昂貴、效能受限或過度使用資源。

快取可以減少您對加密服務的使用，例如 AWS Key Management Service (AWS KMS)。如果您達到了AWS KMS requests-per-second極限，緩存可以提供幫助。您的應用程式可以使用快取金鑰來服務部分資料金鑰要求，而非呼叫 AWS KMS。您也可以在 Sup AWS port 中心建立案例，以提高帳戶的限制。)

可 AWS Encryption SDK 協助您建立和管理資料金鑰快取。它提供本機快取和快取加密材料管理員 (快取 CMM)，可與快取互動，並強制執行您設定的安全性閾值。這些元件在整合運作之後，能夠讓您透過重複使用資料金鑰提高產能效率，同時維護系統安全性。

資料金鑰快取是您應該謹慎使用的選用功能。 AWS Encryption SDK 依預設，會為每個加密作業 AWS Encryption SDK 產生新的資料金鑰。這項技術能支援加密操作的最佳實務，而這種做法並不鼓勵過度重複使用資料金鑰。一般而言，資料金鑰快取只會在為了滿足效能目標時才會啟用。接著，請使用資料金鑰快取安全性閾值，確保您是使用最低快取數來達成成本和效能目標。

AWS Encryption SDK 對於 .NET，不支援快取 CMM。版本 3. 適用於 JAVA 的 AWS Encryption SDK 唯一的 x 支援使用舊版主金鑰提供者介面快取 CMM，而不是金鑰圈介面。但是，版本 4。 AWS Encryption SDK 適用於 .NET 和版本 3 的 x。 的 x 適用於 JAVA 的 AWS Encryption SDK 支持AWS KMS 分層密鑰環，一種替代的加密材料緩存解決方案。使用 AWS KMS 階層式金鑰圈加密的內容只能使用 AWS KMS 階層式金鑰圈解密。

如需這些安全性權衡的詳細討論，請參閱 AWS Encryption SDK：如何在 AWS 安全性部落格中判斷資料金鑰快取是否適合您的應用程式。

主題

- 如何使用資料金鑰快取
- 設定快取安全性閾值
- 資料金鑰快取詳細資訊
- 資料金鑰快取範例

如何使用資料金鑰快取

此主題說明如何在應用程式中使用資料金鑰快取。它會逐步引導您完成程序的每個步驟。然後，將步驟結合為簡單範例，在操作中使用資料金鑰快取來加密字串。

本節中的範例說明如何使用 2.0 版。 x 及更新版本的AWS Encryption SDK. 如需使用舊版的範例，請在程式設計語言的 GitHub 儲存庫的 Release 清單中找到您的版本。

如需在 AWS Encryption SDK 中使用資料金鑰快取的完整和經測試範例，請參閱：

- C/C++：caching_cmm.cpp
- 爪哇 SimpleDataKeyCachingExample
- JavaScript 瀏覽器：緩存厘米. TS
- JavaScript Node.js: 緩存 _ 厘米.
- Python：data_key_caching_basic.py

AWS Encryption SDK適用於 .NET 不支援資料金鑰快取。

主題

- 使用資料金鑰快取：S tep-by-step
- 資料金鑰快取範例：加密字串

使用資料金鑰快取：S tep-by-step

這些 step-by-step 指示說明如何建立實作資料金鑰快取所需的元件。

- 建立資料金鑰快取。在這些範例中，我們使用AWS Encryption SDK提供的本機快取。我們將快取限制為 10 個資料金鑰。

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

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

```
// 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)
```

- 建立主要金鑰提供者 (Java 和 Python) 或金鑰環 (C 和 JavaScript)。這些範例使用 AWS Key Management Service (AWS KMS) 主要金鑰提供者或相容的金AWS KMS鑰圈。

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

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

```
// 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

在瀏覽器中，您必須安全地注入您的登入資料。此範例定義會在執行階段解析登入資料的 webpack (kms.webpack.config) 中的登入資料。它會建立來自 AWS KMS 用戶端的 AWS KMS 用戶端提供者執行個體和登入資料。然後，當它創建密鑰環時，它將客戶端提供程序與 AWS KMS key (`generatorKeyId`).

```
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])
```

- 創建緩存加密材料管理器（緩存 CMM）。

將您的快取 CMM 與快取以及主要金鑰提供者或金鑰圈建立關聯。然後，在快取 CMM 上設定快取安全性閾值。

C

在中適用於 C 的 AWS Encryption SDK，您可以從基礎 CMM (例如預設 CMM) 或金鑰環建立快取 CMM。此範例會從 keyring 建立快取 CMM。

建立快取 CMM 之後，您可以釋放金鑰圈和快取的參照。如需詳細資訊，請參閱 the section called “參考計數”。

```
// 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

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. 的 x 適用於 JAVA 的 AWS Encryption SDK 不支援資料金鑰快取，但它確實支援AWS KMS階層式金鑰圈，這是一種替代的加密材料快取解決方案。

```
/*
 * 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
)
```

您只需進行這些操作。然下來，讓 AWS Encryption SDK為您管理快取，或新增您自己的快取管理邏輯。

當您想要在呼叫中使用資料金鑰快取來加密或解密資料時，請指定快取 CMM，而不是主金鑰提供者或其他 CMM。

> **Note**
> 如果您正在加密資料串流或任何未知大小的資料，請務必在請求中指定資料大小。在加密未知大小的資料時，AWS Encryption SDK 不會使用資料金鑰。

C

在 適用於 C 的 AWS Encryption SDK 中，您可以建立具有快取 CMM 的工作階段，然後處理工作階段。

在預設情況下，當訊息大小不明和未限制時，AWS Encryption SDK 不會快取資料金鑰。若要在不知道確切資料大小時允許快取，請使用 `aws_cryptosdk_session_set_message_bound` 方法來設定的訊息大小上限。將限制設定為大於估計的訊息大小。如果實際訊息大小超過限制，則加密操作會失敗。

```
/* 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

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

```
// 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

當您在 Node.js 中使用快取 CMM 適用於 JavaScript 的 AWS Encryption SDK 時，此方`encrypt`法需要純文字的長度。如果您不提供該資料，就不會快取資料金鑰。如果您提供長度，但您提供的純文字資料超過該長度，則加密操作會失敗。如果您不知道純文字的確切長度，例如當您串流資料時，請提供最大的預期值。

```
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
)
```

資料金鑰快取範例：加密字串

這個簡單的程式碼範例在加密字串時使用資料金鑰快取。它將程序中的step-by-step 程式碼結合到您可以執行的測試程式碼中。

此範例會建立一個AWS KMS key本機快取以及一個主要金鑰提供者或金鑰環。然後，它會使用本機快取和主要金鑰提供者或金鑰圈來建立具有適當安全性閾值的快取 CMM。在 Java 和 Python 中，加密要求會指定快取 CMM、要加密的純文字資料以及加密內容。在 C 中，快取 CMM 會在工作階段中指定，並將該工作階段提供給加密請求。

若要執行這些範例，您需要提供. AWS KMS key 請確定您擁有使用 AWS KMS key 的許可，以產生資料金鑰。

如需建立和使用資料金鑰快取的更詳細實際範例，請參閱〈〉資料金鑰快取範例程式碼。

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

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

```
// 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
```

設定快取安全性閾值

實作資料金鑰快取時，您需要設定高速緩存 CMM強制執行。

這類安全性閾值能協助您限制每個已快取資料金鑰可使用的時間長短，以及依據每個資料金鑰可以保護的資料數量。快取項目必須符合所有安全性閾值，CMM 才會傳回已快取的資料金鑰。如果快取項目超過任何一個閾值，該項目就不能用於目前的操作，並將盡快從快取中移出。每個資料金鑰的初次使用 (在快取之前) 不會納入這些閾值中。

根據經驗，最好是使用數量最低、但能滿足成本和效能目標的快取數量。

AWS Encryption SDK 只會快取已使用金鑰衍生函數加密的資料金鑰。此外，它會建立某些閾值的上限值。這些限制能確保，資料金鑰重複使用數量不會超過其加密限制。不過，因為您的純文字資料金鑰已經過快取處理 (預設是記憶體內快取)，所以請嘗試將金鑰儲存時間降到最短。此外，請嘗試限制會在金鑰遭洩時可能受到暴露的資料。

有關設置緩存安全閾值的示例，請參閱AWS Encryption SDK：如何判斷您的應用程式是否適合使用資料金鑰快取中的AWS安全性部落格。

> Note
>
> 快取 CMM 會強制執行下列所有閾值。如果您未指定選用值，則快取 CMM 會使用預設值。
> 若要暫時停用資料金鑰快取，AWS Encryption SDK提供空密碼編譯資料快取（空緩存）。Null 快取會為每個 GET 請求傳回錯過，而且不會回應任何 PUT 請求。建議您使用 null 快取，而不要將快取容量或安全性閾值設定為 0。如需詳細資訊，請參閱 Java 和 Python 中的 null 快取。

最大存留期 (必要)

決定快取項目可以使用的時間長度，從項目加入起開始計時。此值為必填。輸入大於 0 的數值。AWS Encryption SDK 不限制存留期上限值。

所有語言實作AWS Encryption SDK定義最大時間（以秒為單位），除了適用於 JavaScript 的 AWS Encryption SDK，它使用毫秒。

使用可讓應用程式繼續發揮快取優勢的最短間隔。您可以使用像金鑰輪換政策的最大存留期閾值。使用它來限制資料金鑰的重複使用，大幅降低加密資料暴露的風險，以及移出在受到快取時可能造成政策改變的資料金鑰。

最大加密訊息數 (選用)

指定快取資料金鑰可以加密的最大訊息數。此值是選用的。請輸入介於 1 到 2^32 之間的訊息數。預設值為 2^32 則訊息。

每個快取金鑰可以保護之訊息數量的設定值，應該要大至能夠取得重複使用次數值、但又能小至能夠限制當金鑰遭到洩漏時可能受到暴露的訊息數。

最大加密位元組數 (選用)

指定快取資料金鑰可以加密的最大位元組數。此值是選用的。請輸入介於 0 到 2^63 - 1 之間的值。預設值為 2^63 - 1。值為 0 只允許您在加密空的訊息字串時使用資料金鑰快取。

評估此閾值時，將會納入目前請求中的位元組。如果已處理的位元組加上目前位元組超過該閾值，則快取的資料金鑰從會從快取移出，即使該金鑰已用於較小的請求。

資料金鑰快取詳細資訊

大多數應用程式可以使用預設的資料金鑰快取實作，無須撰寫自訂程式碼。本節說明預設實作和一些選項詳細資訊。

主題

- 資料金鑰快取的運作方式
- 建立密碼編譯資料快取
- 建立快取密碼編譯資料管理員
- 資料金鑰快取項目中有什麼項目？
- 加密內容：如何選擇快取項目
- 我的應用程式是否使用快取的資料金鑰？

資料金鑰快取的運作方式

當您在請求中使用資料金鑰快取來加密或解密資料時，AWS Encryption SDK 會先搜尋快取中是否有符合請求的資料金鑰。如果找到有效的相符項目，則使用快取的資料金鑰來加密資料。否則，它會產生一個新的資料金鑰，就像沒有快取一樣。

資料金鑰快取不會用於不明大小的資料，例如串流資料。這使得快取 CMM 能夠正確地強制執行最大字節閾值。若要避免這種行為，請將訊息大小新增至加密請求。

除了快取，資料金鑰快取也會使用快取密碼編譯資料管理員（緩存 CMM）。緩存 CMM 是一個專門的密碼編譯資料管理員 (CMM)，它與高速緩存和底層螢米。(當您指定主金鑰提供者或密鑰環，AWS Encryption SDK會為您創建默認 CMM。 ）快取 CMM 會快取其基礎 CMM 傳回的資料金鑰。快取 CMM 也會強制執行您設定的快取安全性閾值。

為避免從快取選取錯誤的資料金鑰，所有相容的快取 CMM 都要求快取的密碼編譯資料的下列屬性符合資料請求。

- 演算法套件
- 加密內容 (即使為空)
- 分區名稱 (識別快取 CMM 的字串)
- (僅限解密) 加密的資料金鑰

AWS Encryption SDK 只在演算法套件使用金鑰衍生函數時，才會快取資料金鑰。

以下工作流程示範在有和沒有資料金鑰快取的情形下，請求如何處理加密資料。這些範例顯示您建立的快取元件，包括快取和快取 CMM，如何用於處理中。

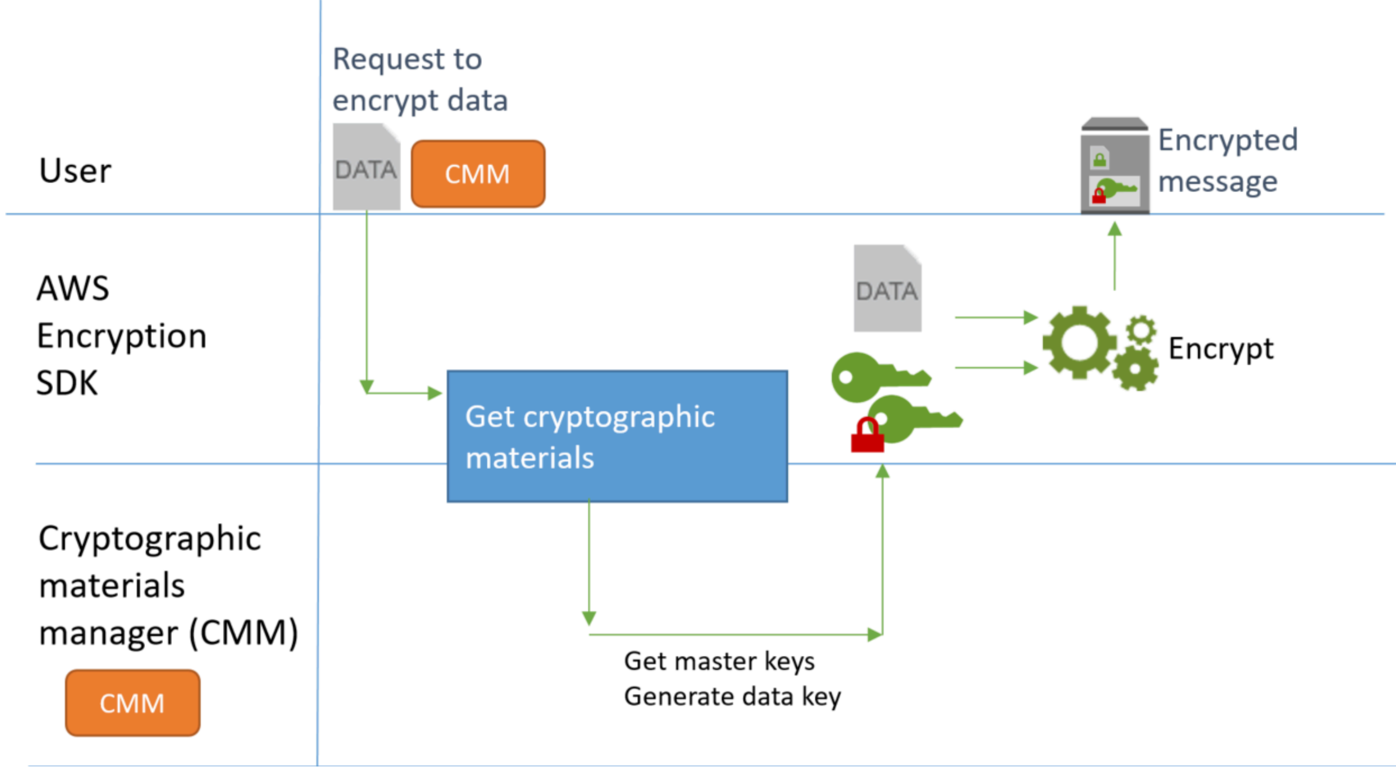
加密資料，不使用快取

若要取得加密資料，但不透過快取：

1. 應用程式要求 AWS Encryption SDK加密資料。

   請求會指定主金鑰提供者或 keyring。所以此AWS Encryption SDK會建立與主金鑰提供者或 keyring 互動的默認 CMM。

2. 所以此AWS Encryption SDK向 CMM 要求加密資料 (取得密碼編譯資料)。
3. CMM 要求其Keyring(C 和JavaScript) 或主金鑰提供者(Java 和 Python) 來取得密碼編譯資料。這可能牽涉到呼叫密碼編譯服務，例如 AWS Key Management Service (AWS KMS)。CMM 會將加密資料傳回給AWS Encryption SDK。
4. AWS Encryption SDK 使用純文字資料金鑰來加密資料。它將加密的資料和加密的資料金鑰存放它傳回給使用者的已加密訊息中。



加密資料，使用快取

若要透過資料金鑰快取來取得加密資料：

1. 應用程式要求 AWS Encryption SDK加密資料。

   請求會指定快取密碼編譯資料管理員 (快取 CMM)與基礎密碼編譯資料管理員 (CMM) 相關聯。指定主金鑰提供者或 keyring 時，AWS Encryption SDK會為您創建默認 CMM。
2. 開發套件向指定的快取 CMM 要求密碼編譯資料。
3. 快取 CMM 從快取中要求取得加密資料。
   a. 如果快取找到相符項目，它會更新存留期並使用相符快取項目的值，傳回快取的加密資料傳回給快取 CMM。

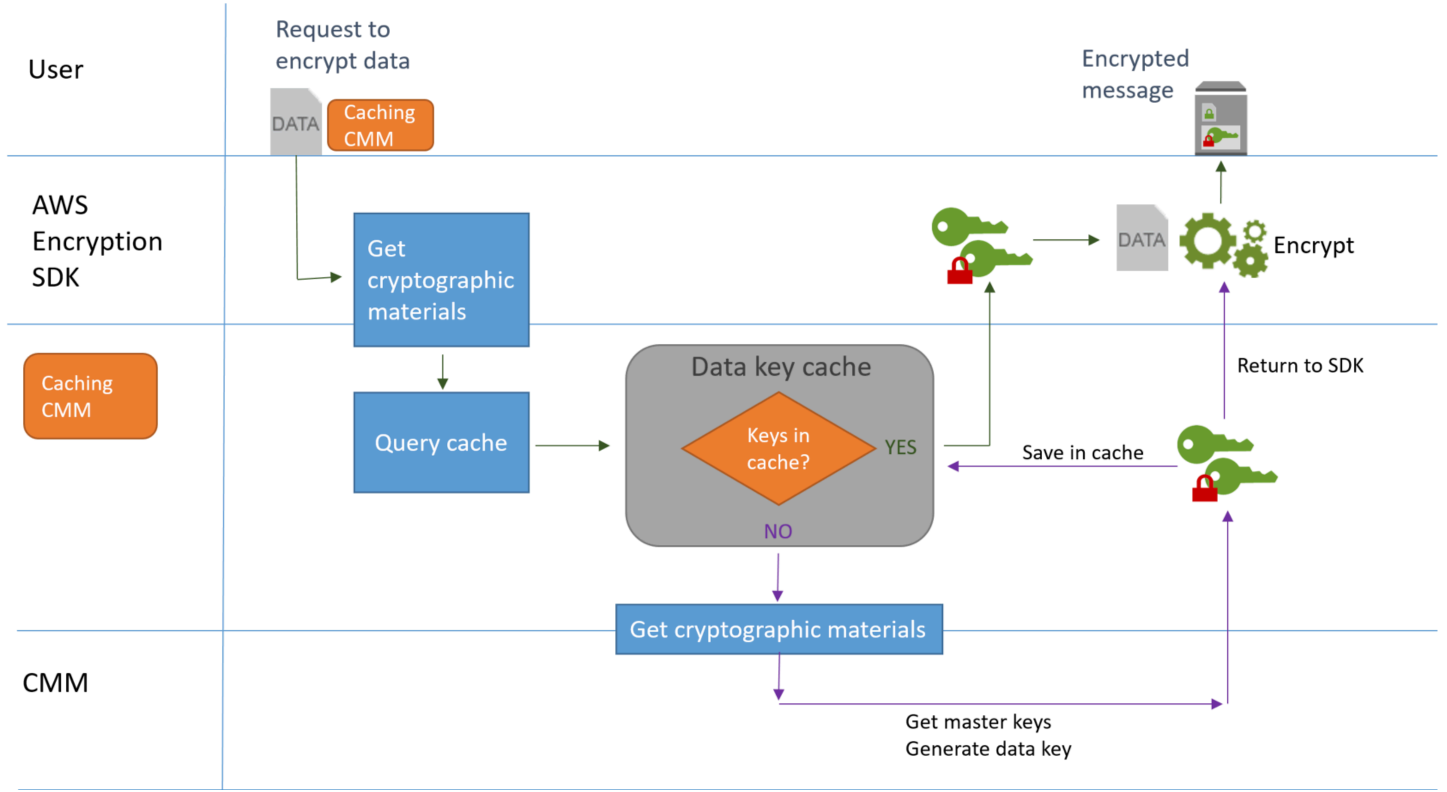
如果緩存條目符合其安全性閾值，快取 CMM 會將其傳回給開發套件。否則，它會通知快取移出項目，並依沒有相符項目的情形繼續進行。

b. 如果快取找不到有效的相符項目，快取 CMM 會要求其基礎 CMM 產生新的資料金鑰。

基礎 CMM 從其 keyring 取得密碼編譯資料 (C 和JavaScript) 或主金鑰提供者 (Java 和 Python)。這可能牽涉到呼叫服務，例如 AWS Key Management Service。基礎 CMM 傳回資料金鑰的純文字和加密副本給快取 CMM。

快取 CMM 會將新的加密資料儲存在快取中。

4. 快取 CMM 會將加密資料傳回給AWS Encryption SDK。
5. AWS Encryption SDK 使用純文字資料金鑰來加密資料。它將加密的資料和加密的資料金鑰存放它傳回給使用者的已加密訊息中。



建立密碼編譯資料快取

AWS Encryption SDK定義用於資料金鑰快取之密碼編譯資料快取的需求。它還提供本機快取，這是一個可配置的、內存中的最久未使用的項目 (LRU) 快取。要創建本地緩存的實例，請使

用`LocalCryptoMaterialsCache`構造函數，getLocalCryptographicMaterialsCache函數 JavaScript，或`aws_cryptosdk_materials_cache_local_newC` 中的構造函數

本地快取包含基本快取管理的邏輯，包括新增、移出和比對快取項目，以及維護快取。您不需要撰寫任何自訂快取管理邏輯。您可以依原狀使用本機快取、對其進行自訂，或取代任何相容的快取。

創建本地緩存時，將其容量，也就是快取可以保留的最大項目數。此設定可協助您以有限的資料金鑰重複使用來設計有效快取。

所以此適用於 JAVA 的 AWS Encryption SDK與適用於 Python 的 AWS Encryption SDK還提供了空密碼編譯資料快取(NullCryptoMaterialsCache。NullCryptoMaterialsCache 會為所有 `GET` 操作傳回錯過，而且不會回應 PUT 操作。您可以在測試中使用 NullCryptoMaterialsCache 或暫時停用包含快取程式碼之應用程式中的快取。

在 中AWS Encryption SDK，每個密碼編譯資料快取都與快取密碼編譯資料管理員（緩存 CMM）。快取 CMM 從快取取取得資料金鑰、將資料金鑰放入快取中，並強制執行安全性閾值你設置的。建立快取 CMM 時，您指定它使用的快取以及用於產生其快取之資料金鑰的基礎 CMM 或主金鑰提供者。

建立快取密碼編譯資料管理員

若要啟用資料金鑰快取，請建立高速緩存和一個快取密碼編譯資料管理員（緩存 CMM）。接著，在您的加密或解密資料請求中，指定快取 CMM，而不是標準的密碼編譯資料管理員 (CMM), 或主金鑰提供者或者Keyring。

有兩種類型的 CMM。兩者都會取得資料金鑰 (以及相關密碼編譯資料)，但使用的方法不同，如下所示：

- CMM 與 keyring (C 或JavaScript) 或主金鑰提供者 (Java 和 Python)。當此開發套件向 CMM 要求取得加密或解密資料時，CMM 會從其 keyring 或主金鑰提供者取得資料。在 Java 和 Python 中，CMM 使用主金鑰來產生、加密或解密資料金鑰。在 C 和 JavaScript 中，keyring 會產生、加密並傳回密碼編譯資料。
- 快取 CMM 與一個快取相關聯，例如本機快取，以及一個基礎 CMM。當開發套件向快取 CMM 要求密碼編譯資料，快取 CMM 會嘗試從快取中取得資料。如果找不到相符項目，快取 CMM 向其基礎 CMM 要求資料。接著，它在將新的密碼編譯資料傳回給發起人之前會快取資料。

緩存 CMM 還強制執行安全性閾值設定的。由於安全性閾值在快取 CMM 中設定並由快取 CMM 強制執行，因此您可以使用任何相容的快取，即使快取不是為敏感材料而設計。

資料金鑰快取項目中有什麼項目？

資料金鑰快取會在快取中存放資料金鑰和相關密碼編譯資料。每個項目都包含下面列出的元素。當您要決定是否使用資料金鑰快取功能，以及要在快取加密資料管理器（快取 CMM）上設定安全性閾值時，這些資訊可能會很有用。

加密請求的快取項目

由於加密操作而加入資料金鑰快取的項目包含下列元素：

- 純文字資料金鑰
- 加密的資料金鑰 (一或多個)
- 加密內容
- 訊息簽署金鑰 (如果使用)
- 演算法套件
- 用於強制執行安全性閾值的中繼資料，包括用量計數器

解密請求的快取項目

由於解密操作而加入資料金鑰快取的項目包含下列元素：

- 純文字資料金鑰
- 簽章驗證金鑰 (如果使用)
- 用於強制執行安全性閾值的中繼資料，包括用量計數器

加密內容：如何選擇快取項目

您可以在加密資料的請求中指定加密內容。不過，加密內容在資料金鑰快取中扮演特殊角色。它可讓您在快取中建立資料金鑰子組，即使資料金鑰來自相同的快取 CMM。

加密內容是一組金鑰/值對，其中包含任意非私密資料。在加密期間，加密內容會以密碼演算法繫結至加密的資料，因此在解密資料時需要相同的加密內容。在 AWS Encryption SDK 中，加密內容與加密的資料和資料金鑰一起存放在已加密訊息中。

使用資料金鑰快取時，您也可以使用快取內容來為您的加密操作選擇特定的快取資料金鑰。加密內容會與會資料金鑰一起儲存在快取項目中 (屬於快取項目 ID 的一部分)。快取的資料金鑰只在其加密內容符

合時才會重複使用。如果您想對加密請求重複使用特定的資料金鑰，請指定相同的加密內容。如果您想避免使用這些資料金鑰，請指定不同的加密內容。

加密內容一律是選用的，但建議使用。如果您不在請求中指定加密內容，則會在快取項目識別符中加入空的加密內容，並符合每個請求。

我的應用程式是否使用快取的資料金鑰？

資料金鑰快取是對某些應用程式和工作負載非常有效的最佳化策略。不過，因為它需要一些風險，請務必判斷它對您的情況如何有效，然後決定優點是否大於風險。

因為資料金鑰快取會重複使用資料金鑰，最明顯的效果就是減少產生新資料金鑰的呼叫次數。實作資料金鑰快取時，AWS Encryption SDK 只會呼叫 AWS KMS`GenerateDataKey` 操作來建立初始資料金鑰，以及快取遺漏時。但是，快取只有在會產生多個具有相同特性 (包括相同加密內容和演算法套件) 的資料金鑰應用程式中，才能明顯地改善效能。

若要判斷您的 AWS Encryption SDK 實作是否實際上使用來自快取的資料金鑰，請嘗試下列技術。

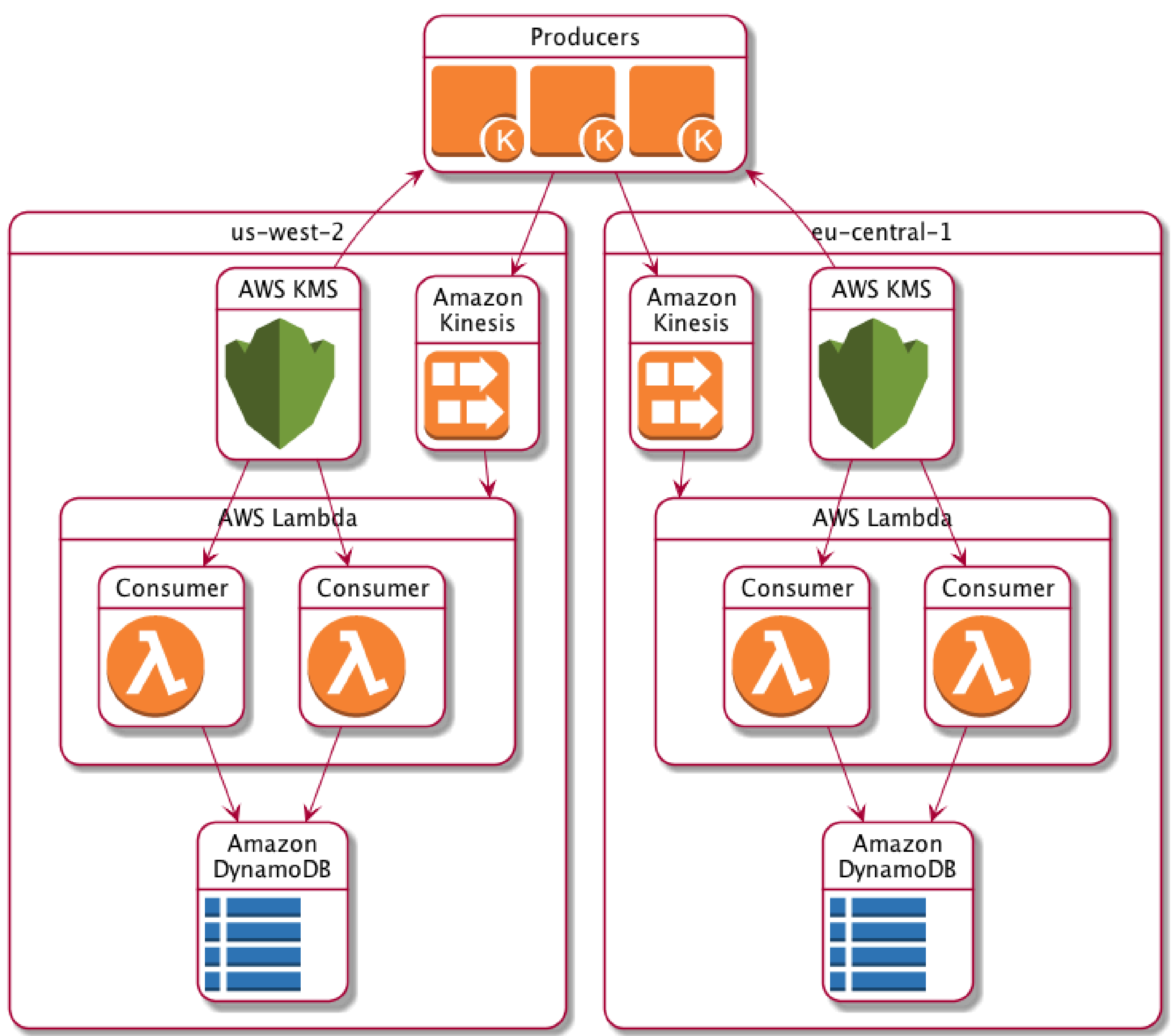
- 在主金鑰基礎結構的記錄中，檢查建立新資料金鑰的呼叫頻率。當資料金鑰快取有效時，建立新金鑰的呼叫次數應該會明顯下降。例如，如果您使用AWS KMS主密鑰提供程序或密鑰環，搜索CloudTrail的 日誌GenerateData金鑰呼叫。
- 比較加密的訊息認為AWS Encryption SDK返回以響應不同的加密請求。例如，如果您使用適用於 JAVA 的 AWS Encryption SDK，比較ParsedCiphertext對象來自不同的加密調用。在 中適用於 JavaScript 的 AWS Encryption SDK中，比較`encryptedDataKeys`屬性MessageHeader。重複使用資料金鑰時，加密訊息中的加密資料金鑰是相同的。

資料金鑰快取範例

此範例使用。資料金鑰快取具有本機快取，來加速應用程式，其中多個裝置產生的資料會在不同區域進行加密與存放。

在此案例中，多個資料製作者產生資料、加密並寫入Kinesis 串流在每個區域。AWS Lambda函數 (消費者) 解密串流並寫入純文字資料到區域的 DynamoDB 表格。資料製作者和消費者使用AWS Encryption SDK和一個AWS KMS主金鑰提供者。為了減少對 KMS 的呼叫，每個製作者和消費者都有自己的本機快取。

您可以在這些範例中找到原始程式碼。Java 和 Python。範例也包含用於定義範例資源的 AWS CloudFormation 範本。



本機快取結果

下表顯示本機快取在此範例中將對 KMS 的呼叫總數 (每秒每個區域) 減少至原始值的 1%。

製作者請求

	每秒每個用戶端的請求數			每個區域的用戶端	每秒每個區域的平均請求數
	產生資料金鑰 (us-west-2)	加密資料金鑰 (eu-central-1)	總數 (每個區域)		

無快取	1	1	1	500	500
本機快取	1 rps/100 次使用	1 rps/100 次使用	1 rps/100 次使用	500	5

消費者請求

	每秒每個用戶端的請求數			每個區域的用戶端	每秒每個區域的平均請求數
	解密資料金鑰	製作者	合計		
無快取	每個製作者 1 rps	500	500	2	1,000
本機快取	每個製作者 1 rps/100 次使用	500	5	2	10

資料金鑰快取範例程式碼

此程式碼範例會使用 Java 和 Python 中的本機快取建立資料金鑰快取的簡單實作。程式碼會建立兩個本機快取執行個體：一個用於正在加密資料的資料生產者，另一個用於解密資料的資料取用者 (AWS Lambda函數)。如需有關在每種語言中實作資料金鑰快取的詳細資 Python，請參閱. AWS Encryption SDK

資料金鑰快取適用於AWS Encryption SDK支援的所有程式設計語言。

如需在 AWS Encryption SDK 中使用資料金鑰快取的完整和經測試範例，請參閱：

- C/C++：caching_cmm.cpp
- 爪哇 SimpleDataKeyCachingExample
- JavaScript 瀏覽器：緩存厘米. TS
- JavaScript Node.js: 緩存 _ 厘米.
- Python：data_key_caching_basic.py

生產者

生產者取得地圖、將其轉換為 JSON、使用對其進行AWS Encryption SDK加密，然後將密文記錄推送至每個地圖中的 Kinesis 串流。AWS 區域

該代碼定義了緩存加密材料管理器（緩存 CMM），並將其與本地緩存和基礎AWS KMS主密鑰提供程序關聯。快取 CMM 會從主金鑰提供者快取資料金鑰 (以及相關的密碼編譯材料)。它也會代表開發套件與快取互動，並強制執行您設定的安全性閾值。

由於對加密方法的呼叫會指定快取 CMM，而不是一般的加密材料管理員 (CMM) 或主要金鑰提供者，因此加密將使用資料金鑰快取。

Java

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

```
/*
 * 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
            )
```

消費者

資料取用者是 Kinesis 事件觸發的AWS Lambda函數。它會解密和還原序列化每個記錄，並將純文字記錄寫入相同區域中的 Amazon DynamoDB 表格。

與生產者代碼一樣，消費者代碼通過在調用解密方法時使用緩存加密材料管理器（緩存 CMM）來啟用數據密鑰緩存。

Java 程式碼會以指定的嚴謹模式建置主金鑰提供者AWS KMS key。解密時不需要嚴格模式，但這是最佳做法。Python 程式碼使用探索模式，可讓AWS Encryption SDK使用任何加密資料金鑰的包裝金鑰來解密。

Java

下列範例使用版本 2。 的 x 的適用於 JAVA 的 AWS Encryption SDK。版本 3. x 的適用於 JAVA 的 AWS Encryption SDK棄用數據密鑰緩存 CMM。使用版本 3。 x，您還可以使用AWS KMS分層密鑰環，這是一種替代的加密材料緩存解決方案。

此代碼創建一個主密鑰提供程序，用於在嚴格模式下解密。只AWS Encryption SDK能使用AWS KMS keys您指定的來解密郵件。

```
/*
 * 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 代碼在發現模式下使用主密鑰提供程序進行解密。它允許AWS Encryption SDK使用加密數據密鑰的任何包裝密鑰來解密它。嚴格模式是最佳作法，您可以在其中指定可用於解密的包裝金鑰。

```
"""
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)
```

資料金鑰快取範例：AWS CloudFormation範本

這一個AWS CloudFormation範本會設定所有必要的AWS資源來重現資料金鑰快取範例。

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 語言實作使用語意版本控制，讓您更輕鬆地識別每個版本中變更的幅度。主要版本號碼的變更，例如 1。 x. x 轉換為 2。 x. x，表示可能需要程式碼變更和規劃部署的重大變更。在新版本中突破更改可能不會影響每個用例，請查看版本說明以查看您是否受到影響。次要版本的變更，例如 x .1。 x 轉換為 x 2. x，始終向後兼容，但可能包含棄用的元素。

請盡可能使用您選擇的程式語言 AWS Encryption SDK 中的最新版本。每個版本的維護和支持策略因編程語言實現而異。如需有關慣用程式設計語言支援版本的詳細資訊，請參閱其GitHub儲存庫中的`SUPPORT_POLICY.rst`檔案。

當升級包含需要特殊配置以避免加密或解密錯誤的新功能時，我們會提供中繼版本以及使用中繼版本的詳細說明。例如，版本 1.7。 x 和 1.8. x 的設計是轉換版本，可協助您從 1.7 之前的版本進行升級。 x 轉換為 2.0 版本。 x 及更新版本。如需詳細資訊，請參閱 遷移您的AWS Encryption SDK。

> Note
>
> 版本號碼中的 x 代表主要和次要版本的任何修補程式。例如，版本 1.7。 x 代表以 1.7 開頭的所有版本，包括 1.7.1 和 1.7.9。
>
> 新的安全功能最初在 AWS 加密 CLI 版本 1.7 中發布。 X 和 2.0。 x. 但是， AWS 加密 CLI 版本 1.8。 x 取代了 1.7 版本。 x 和 AWS 加密 CLI 碼 2.1. x 取代了 2.0。 x. 如需詳細資訊，請參閱上的aws-encryption-sdk-cli儲存庫中的相關安全性建議 GitHub。

下表概述了每種程式設計語言的支援版本之間 AWS Encryption SDK 的主要差異。

C

如需所有變更的詳細描述，請參閱的儲存庫中的變更記錄檔 .md。aws-encryption-sdk-c GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
1.	1.0	初始版本。	終止 Support 階段
	1.7	AWS Encryption SDK 可協助舊版使用者升級	

		至 2.0 版的更新。 x 及更新版本。如需詳細資訊，請參閱版本 1.7。 x.	
2.x	2.0	更新到 AWS Encryption SDK. 如需詳細資訊，請參閱 2.0 版。 x.	一般可用性 (GA)
	2.2	改進了消息解密過程。	
	2.3	添加對 AWS KMS 多區域鍵的支持。	

C # /.

如需所有變更的詳細描述，請參閱的儲存庫中的變更記錄檔 .md。aws-encryption-sdk-net GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
3.x	3.0	初始版本。	一般可用性 (GA) AWS Encryption SDK 適用於 .NET 的第 3.x 版將於 2024 年 5 月 13 日進入維護模式。
4.x	4.0	添加對 AWS KMS 階層式金鑰圈、所需加密上下文 CMM 和非對稱 R AWS KMS SA 金鑰圈的支援。	一般可用性 (GA)

命令 CLI 介面

如需所有變更的詳細描述，請參閱的的版本AWS加密 CLI和儲存庫中的變更記錄檔 .rst。aws-encryption-sdk-cli GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
1.	1.0	初始版本。	終止 Support 階段
	1.7	AWS Encryption SDK 可協助舊版使用者升級至 2.0 版的更新。 x 及更新版本。如需詳細資訊，請參閱版本 1.7。 x.	
2.x	2.0	更新到 AWS Encryption SDK. 如需詳細資訊，請參閱 2.0 版。 x.	終止 Support 階段
	2.1	移除`--discovery` 參數，並將其取代為`--wrapping-keys` 參數的`discovery` 屬性。 AWS 加密 CLI 的 2.1.0 版相當於其他編程語言中的 2.0 版本。	
	2.2	改進了消息解密過程。	
3.x	3.0	添加對 AWS KMS 多區域鍵的支持。	終止 Support 階段
4.x	4.0	加 AWS 密 CLI 不再支持 Python 2 或 Python 3.4。從主要版本 4 開	一般可用性 (GA)

		始。 AWS 加密 CLI 的 x，只支持 Python 3.5 或更高版本。	
	4.1	加 AWS 密 CLI 不再支持 Python 3.5。自 4.1 版本開始。 AWS 加密 CLI 的 x，只支持 Python 3.6 或更高版本。	
	4.2	加 AWS 密 CLI 不再支持 Python 3.6。從 4.2 版本開始。 AWS 加密 CLI 的 x，只支持 Python 3.7 或更高版本。	

Java

如需所有變更的詳細描述，請參閱的存放庫中的變更記錄檔 .rst。aws-encryption-sdk-java GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
1.	1.0	初始版本。	終止 Support 階段
	1.3	添加對加密材料管理器和數據密鑰緩存的支持。轉移到確定性 IV 代。	
	1.6.1	棄用`AwsCrypto.encryptString()` `AwsCrypto.decryptString()` 和取代	

		它們與`AwsCrypto.encryptData()` 。`AwsCrypto.decryptData()`	
	1.7	AWS Encryption SDK 可協助舊版使用者升級至 2.0 版的更新。 x 及更新版本。如需詳細資訊，請參閱版本 1.7。 x.	
2.x	2.0	更新到 AWS Encryption SDK. 如需詳細資訊，請參閱 2.0 版。 x.	一般可用性 (GA) 的 2.x 版 適用於 JAVA 的 AWS Encryption SDK 將於 2024 年進入維護模式。
	2.2	改進了消息解密過程。	
	2.3	添加對 AWS KMS 多區域鍵的支持。	
	2.4	添加對 AWS SDK for Java 2.x.	
3.x	3.0	適用於 JAVA 的 AWS Encryption SDK 與材料提供者資料庫整合。 添加對對稱和非對稱 RSA AWS KMS 密鑰環， AWS KMS 分層密鑰環，Raw AES 密鑰環，Raw RSA 密鑰環，多密鑰環以及所需的加密上下文 CMM 的支持。	一般可用性 (GA)

JavaScript

如需所有變更的詳細描述，請參閱的儲存庫中的變更記錄檔 .md。aws-encryption-sdk-javascript GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
1.	1.0	初始版本。	終止 Support 階段
	1.7	AWS Encryption SDK 可協助舊版使用者升級至 2.0 版的更新。 x 及更新版本。如需詳細資訊，請參閱版本 1.7。 x.	
2.x	2.0	更新到 AWS Encryption SDK. 如需詳細資訊，請參閱 2.0 版。 x.	終止 Support 階段
	2.2	改進了消息解密過程。	
	2.3	添加對 AWS KMS 多區域鍵的支持。	
3.x	3.0	移除節點 10 的 CI 涵蓋範圍。升級相依性不再支援節點 8 和節點 10。	Maintenance (維護) 第 3.x 版的 Support 適用於 JavaScript 的 AWS Encryption SDK 將於 2024 年 1 月 17 日結束。
4.x	4.0	需要的第 3 版 適用於 JavaScript 的 AWS Encryption SDK 才`kms-client` 能	一般可用性 (GA)

使用 AWS KMS 鑰匙圈。

Python

如需所有變更的詳細描述，請參閱的存放庫中的變更記錄檔 .rst。aws-encryption-sdk-python GitHub

主要版本	詳細資訊		SDK 主要版本生命週期階段
1.	1.0	初始版本。	終止 Support 階段
	1.3	添加對加密材料管理器和數據密鑰緩存的支持。轉移到確定性 IV 代。	
	1.7	AWS Encryption SDK 可協助舊版使用者升級至 2.0 版的更新。 x 及更新版本。如需詳細資訊，請參閱版本 1.7。 x.	
2.x	2.0	更新到 AWS Encryption SDK. 如需詳細資訊，請參閱 2.0 版。 x.	終止 Support 階段
	2.2	改進了消息解密過程。	
	2.3	添加對 AWS KMS 多區域鍵的支持。	
3.x	3.0	適用於 Python 的 AWS Encryption SDK 不再支持 Python 2 或 Python 3.4。從主要版本 3 開始。 的 x 適	一般可用性 (GA)

用於 Python 的 AWS Encryption SDK，只支援 Python 3.5 或更新版本。

版本詳情

下列清單說明支援的版本之間的主要差異 AWS Encryption SDK。

主題

- 早於 1.7 的版本。 x
- 版本 1.7。 x
- 版本 2.0。 x
- 版本 2.2. x
- 版本 2.3. x

早於 1.7 的版本。 x

> **Note**
>
> 全部 1. x. x 版本的處 AWS Encryption SDK 於end-of-support相位中。只要可行，請盡快升級到 AWS Encryption SDK 適用於您程式設計語言的最新可用版本。若要從 1.7 之前的 AWS Encryption SDK 版本升級。 x，您必須先升級到 1.7。 x. 如需詳細資訊，請參閱 遷移您的 AWS Encryption SDK。

AWS Encryption SDK 早於 1.7 的版本。 x 提供重要的安全功能，包括使用 Galo/ 計數器模式（AES-GCM）中的高級加密標準算法進行加密，基於 HMAC 的密 extract-and-expand 鑰派生功能（HKDF），簽名和 256 位加密密鑰。不過，這些版本不支援我們建議的最佳做法，包括主要承諾。

版本 1.7。 x

> **Note**
>
> 全部 1. x. x 版本的處 AWS Encryption SDK 於end-of-support相位中。

版本 1.7。 x 旨在協助舊版的使用者升級 AWS Encryption SDK 至 2.0 版。 x 及更新版本。如果您不熟悉 AWS Encryption SDK，可以略過此版本，並以您的程式設計語言開始使用最新的可用版本。

版本 1.7。 x 完全向下相容；它不會引入任何重大變更或變更 AWS Encryption SDK. 它也是向前兼容; 它允許你更新你的代碼，所以它與 2.0 版本兼容。 x. 它包含新功能，但並未完全啟用它們。而且，它需要配置值，以防止您立即採用所有新功能，直到您準備就緒。

版本 1.7。 x 包含下列變更：

AWS KMS 主要金鑰提供者更新 (必要)

版本 1.7。 x 引入了新的建構函式， 適用於 JAVA 的 AWS Encryption SDK 並 適用於 Python 的 AWS Encryption SDK 在嚴格或探索模式下明確建立 AWS KMS 主要金鑰提供者。此版本將類似的更改添加到 AWS Encryption SDK 命令行界面（CLI）。如需詳細資訊，請參閱 更新AWS KMS主要金鑰提供者。

- 在嚴謹模式中， AWS KMS 主金鑰提供者需要包裝金鑰清單，而且只使用您指定的包裝金鑰來加密和解密。這是確保您使用要使用的包裝鍵的 AWS Encryption SDK 最佳做法。
- 在探索模式中， AWS KMS 主金鑰提供者不會採用任何包裝金鑰。您無法使用它們進行加密。解密時，他們可以使用任何包裝密鑰來解密加密的數據密鑰。但是，您可以將用於解密的包裝密鑰限制為特定的密鑰 AWS 帳戶。帳戶篩選是選擇性的，但這是我們建議使用的最佳作法。

建立舊版 AWS KMS 主金鑰提供者的建構函式在 1.7 版中已淘汰。 x 並在 2.0 版中刪除。 x. 這些建構函式會使用您指定的包裝金鑰來實體化加密的主金鑰提供者。但是，他們使用加密它們的包裝密鑰來解密加密的數據密鑰，而不考慮指定的包裝密鑰。用戶可能會無意中使用不打算使用的包裝密鑰來解密消息，包括 AWS KMS keys 在其他 AWS 帳戶 和區域中。

AWS KMS 主密鑰的構造函數沒有更改。加密和解密時， AWS KMS 主金鑰只會使用您指定 AWS KMS key 的金鑰。

AWS KMS 金鑰圈更新 (選擇性)

版本 1.7。 x 在 適用於 C 的 AWS Encryption SDK 和 適用於 JavaScript 的 AWS Encryption SDK 實作中新增了一個篩選器，將AWS KMS 探索金鑰環限制在特 AWS 帳戶定範圍內。這個新帳戶篩選器是選擇性的，但這是我們建議使用的最佳作法。如需詳細資訊，請參閱 正在更新AWS KMS鑰匙圈。

AWS KMS 鑰匙圈的構造函數沒有更改。標準金 AWS KMS 鑰環在嚴謹模式下的行為與主要金鑰提供者相同。 AWS KMS 探索金鑰圈是在探索模式下明確建立的。

傳遞密鑰 ID 進行解 AWS KMS 密

從 1.7 版本開始。 x，解密加密的資料金鑰時， AWS Encryption SDK 永遠會 AWS KMS key 在呼叫「 AWS KMS 解密」作業時指定 a。 AWS Encryption SDK 會從每個加密資料金鑰的 AWS KMS key 中繼資料取得的金鑰 ID 值。此功能不需要更改任何代碼。

若要解密以對稱加密 KMS 金鑰加密的加密文字，不需要指定的金鑰識別碼，但這是最佳作AWS KMS 法。 AWS KMS key就像在金鑰提供者中指定包裝金鑰一樣，此做法可確保 AWS KMS 只使用您想要使用的包裝金鑰來解密。

使用金鑰承諾解密密文

版本 1.7。 x 可以解密使用或不使用金鑰承諾加密的密文。但是，它無法使用密鑰承諾來加密密文本。此屬性可讓您完全部署應用程式，在遇到任何此類加密文字之前，使用金鑰承諾加密加密的應用程式。因為這個版本會解密沒有金鑰承諾的加密郵件，因此您不需要重新加密任何加密文字。

為了實現此行為，版本 1.7。 x 包含新的承諾產品原則組態設定，可決定是否 AWS Encryption SDK 可以使用金鑰承諾加密或解密。在版本 1.7 中。 x 是承諾產品原則的唯一有效值`ForbidEncryptAllowDecrypt`，會用於所有加密和解密作業。此值可防 AWS Encryption SDK 止使用包含金鑰承諾的任一新演算法套件進行加密。它允許在 AWS Encryption SDK 有和沒有密鑰承諾的情況下解密密文。

雖然 1.7 版中只有一個有效的承諾產品原則值。 x，當您使用此版本中引入的新 API 時，我們要求您可以明確設定此值。明確設定值可防止承諾用戶原則在升級至 2.1 版`require-encrypt-require-decrypt`時自動變更為。 x. 相反地，您可以分階段遷移承諾產品原則。

具有關鍵承諾的算法套件

版本 1.7。 x 包含兩個支援關鍵承諾的新演算法套件。其中一個包括簽署；另一個則不包括簽署。像之前支持的算法套件一樣，這兩個新的算法套件都包括使用 AES-GCM 進行加密，256 位加密密鑰和基於 HMAC extract-and-expand 的密鑰派生函數（HKDF）。

但是，用於加密的預設演算法套件不會變更。這些演算法套件已新增至 1.7 版。 x 來準備您的應用程式，以便在 2.0 版中使用它們。 x 及更新版本。

CMM 實作變更

版本 1.7。 x 引入預設密碼材料管理員 (CMM) 介面的變更，以支援金鑰承諾。只有在您撰寫自訂 CMM 時，此變更才會影響您。有關詳細信息，請參閱您的編程語言的 API 文檔或 GitHub 存儲庫。

版本 2.0。 x

版本 2.0。 x 支援中提供的新安全性功能 AWS Encryption SDK，包括指定的包裝金鑰和金鑰承諾。為了支持這些功能，2.0 版本。 x 包含舊版的重大變更 AWS Encryption SDK。您可以透過部署 1.7 版來準備這些變更。 x. 版本 2.0。 x 包含 1.7 版中引入的所有新功能。 x 具有以下添加和更改。

> **Note**
>
> 版本 2. x. 適用於 Python 的 AWS Encryption SDK 適用於 JavaScript 的 AWS Encryption SDK、和 AWS 加密 CLI 的 x 正處於end-of-support階段。
> 如需有關以偏好程式設計語言支援和維護此 AWS Encryption SDK 版本的資訊，請參閱其GitHub儲存庫中的`SUPPORT_POLICY.rst`檔案。

AWS KMS 主要金鑰提供者

在 1.7 版中棄用的原始 AWS KMS 主密鑰提供程序構造函數。 x 在 2.0 版中被刪除。 x. 您必須在嚴謹模式或探索模式下明確建構 AWS KMS 主金鑰提供者。

使用金鑰承諾加密和解密密文

版本 2.0。 x 可以使用或不使用密鑰承諾來加密和解密密碼。其行為是由承諾產品原則設定所決定。默認情況下，它始終使用密鑰承諾進行加密，並且僅解密使用密鑰承諾加密的密文。除非您變更承諾原則，否則 AWS Encryption SDK 將不會解密任何舊版 (包括 1.7 版) 加密的 AWS Encryption SDK密文。 x.

> **Important**
>
> 默認情況下，2.0 版本。 x 不會解密任何在沒有金鑰承諾的情況下加密的密文。如果您的應用程式可能遇到在沒有金鑰承諾的情況下加密的加密文字，請使用. `AllowDecrypt`

在 2.0 版本中。 x，履約承諾原則設定有三個有效值：

- `ForbidEncryptAllowDecrypt`— AWS Encryption SDK 無法使用密鑰承諾進行加密。它可以解密有或沒有密鑰承諾加密的密文。
- `RequireEncryptAllowDecrypt`— AWS Encryption SDK 必須使用密鑰承諾進行加密。它可以解密有或沒有密鑰承諾加密的密文。
- `RequireEncryptRequireDecrypt`(預設值) — AWS Encryption SDK 必須使用金鑰承諾加密。它只解密密文與關鍵承諾.

如果您要從舊版的遷移 AWS Encryption SDK 至 2.0 版。 x，將承諾原則設定為可確保您可以解密應用程式可能遇到的所有現有加密文字的值。您可能會隨著時間的推移調整此設定。

版本 2.2. x

添加對數字簽名的支持並限制加密的數據密鑰。

> **Note**
>
> 版本 2. x. 適用於 Python 的 AWS Encryption SDK 適用於 JavaScript 的 AWS Encryption SDK、和 AWS 加密 CLI 的 x 正處於end-of-support階段。
> 如需有關以偏好程式設計語言支援和維護此 AWS Encryption SDK 版本的資訊，請參閱其GitHub儲存庫中的`SUPPORT_POLICY.rst`檔案。

數位簽章

為了改善解密時對數位簽章的處理， AWS Encryption SDK 包括下列功能：

- 非串流模式 — 僅在處理所有輸入之後傳回純文字，包括驗證數位簽章 (如果存在)。此功能可防止您在驗證數位簽章之前使用純文字。每當您解密使用數位簽章 (預設演算法套件) 加密的資料時，請使用此功能。例如，由於 AWS 加密 CLI 一律以串流模式處理資料，因此在使用`- -buffer`數位簽章解密加密文字時，請使用參數。
- 僅限未簽名的解密模式 — 此功能僅解密未簽署的密文。如果解密在密文中遇到數位簽章，作業就會失敗。使用此功能可避免在驗證簽章之前，無意中處理已簽署郵件的純文字。

限制加密的資料金鑰

您可以限制加密訊息中加密資料金鑰的數量。此功能可協助您在加密時偵測設定錯誤的主要金鑰提供者或金鑰環，或在解密時識別惡意加密文字。

當您解密來自不受信任來源的郵件時，您應該限制加密的資料金鑰。它可以防止對您的金鑰基礎結構進行不必要、昂貴且可能詳盡的呼叫。

版本 2.3. x

添加對 AWS KMS 多區域鍵的支持。如需詳細資訊，請參閱 使用多地區 AWS KMS keys。

Note

加 AWS 密 CLI 支援從 3.0 版開始的多區域金鑰。 x.

版本 2. x. 適用於 Python 的 AWS Encryption SDK 適用於 JavaScript 的 AWS Encryption SDK、和 AWS 加密 CLI 的 x 正處於end-of-support階段。

如需有關以偏好程式設計語言支援和維護此 AWS Encryption SDK 版本的資訊，請參閱其GitHub儲存庫中的`SUPPORT_POLICY.rst`檔案。

遷移您的AWS Encryption SDK

所以此AWS Encryption SDK支持多種可互操作程式設計語言實作，每個都是在開源存儲庫中開發的 GitHub。作為最佳實務，建議您使用最新版本AWS Encryption SDK對於每種語言。

您可以安全地從 2.0 版升級。x或更高版本AWS Encryption SDK至最新版本。然而，2.0.x的版本AWS Encryption SDK引入了重要的新安全功能，其中一些正在突破變化。若要從 1.7 之前的版本進行升級。x至 2.0 版。x之後，您必須先升級到最新的 1。x版本。本節中的主題旨在協助您瞭解變更、為您的應用程式選取正確的版本，以及安全且成功地移轉至AWS Encryption SDK。

如需有關重要版本的資訊AWS Encryption SDK，請參閱的版本 AWS Encryption SDK。

> **Important**
>
> 請勿直接從 1.7 之前的版本升級。x至 2.0 版。x或者更高版本沒有先升級到最新的 1。x版本。如果您直接升級至 2.0 版。x或更高版本並立即啟用所有新功能，AWS Encryption SDK將無法解密在舊版本中加密的密文AWS Encryption SDK。

> **Note**
>
> 最早的版本AWS Encryption SDK.NET 版。x。所有版本AWS Encryption SDK為了 .NET 支持 2.0 中引入的安全最佳實踐。x的AWS Encryption SDK。您可以安全地升級到最新版本，而無需任何代碼或數據更改。
> AWS加密 CLI：閱讀本移轉指南時，請使用 1.7。x遷移指示AWSEncryptionx並使用 2.0。x遷移指示AWS加密 CLI 2.1.x。如需詳細資訊，請參閱 的版本AWS加密 CLI。
> 最初發布了新的安全功能AWS加密 CLI 1.7 版。x和 2.0.x。但是，AWS加密指令行版本 1.8。x取代 1.7 版。x和AWS加密 CLI 2.1.x取代 2.0。x。如需詳細資訊，請參閱相關的安全性諮詢在aws-encryption-sdk-cli儲存庫 GitHub。

新使用者

如果您是初次使用AWS Encryption SDK，安裝最新版本AWS Encryption SDK為您的編程語言。預設值會啟用AWS Encryption SDK，包括使用簽名加密，密鑰派生和主要承諾的。AWS Encryption SDK

最新的使用者

> 建議您儘快從最新版本升級至最新的可用版本。所有 1.x的版本AWS Encryption SDK在end-of-support 階段，正如某些編程語言中的更新版本一樣。有關支持和維護狀態的詳細信息AWS Encryption SDK在您的編程語言中，請參閱Support 與維護。
>
> AWS Encryption SDK2.0 版。x並稍後提供新的安全功能，以協助保護您的資料。但是，AWS Encryption SDK2.0 版。x包括不向後相容的重大變更。為了確保安全的轉換，請從當前版本遷移到最新版本 1 開始。x在你的編程語言。當你最新的 1.x版本已完全部署並成功運行，您可以安全地遷移到 2.0 版本。x和更新版本。這兩步過程對於分佈式應用程序尤其重要。

如需的詳細資訊AWS Encryption SDK基於這些更改的安全功能，請參閱改善用戶端加密：explicit KeyIds 和主要承諾在AWS安全性部落格。

尋求使用的說明適用於 JAVA 的 AWS Encryption SDK與AWS SDK for Java 2.x? 請參閱先決條件。

主題

- 如何遷移和部署AWS Encryption SDK
- 更新AWS KMS主要金鑰提供者
- 正在更新AWS KMS鑰匙圈
- 設定承諾產品原則
- 移轉至最新版本的疑難排解

如何遷移和部署AWS Encryption SDK

從移轉時AWS Encryption SDK1.7 版。x至 2.0 版。x或更高版本，您必須安全地轉換為使用主要承諾。否則，您的應用程序將遇到無法解密的密文。如果您使用AWS KMS主金鑰提供者時，您必須更新為以嚴格模式或探索模式建立主金鑰提供者的新建構函式。

> Note
>
> 本主題是為從舊版移轉的使用者而設計AWS Encryption SDK至 2.0 版。x或更高版本。如果您是新手AWS Encryption SDK，您可以使用預設設定立即開始使用最新的可用版本。

為了避免無法解密您需要閱讀的密文的嚴重情況，我們建議您在多個不同階段進行遷移和部署。在開始下一個階段之前，請確認每個階段都已完成並完全部署。這對於具有多個主機的分佈式應用程序尤為重要。

階段 1：將您的應用程式更新至最新的 1.x版

更新至最新的 1.x適用於您編程語言的版本。在啟動階段 2 之前，請仔細測試、部署您的變更，並確認更新已傳播到所有目的地主機。

> **Important**
>
> 確認您最新的 1.x版本 1.7 版。x或更高版本AWS Encryption SDK。

最新的 1.x的版本AWS Encryption SDK與舊版的向後相容AWS Encryption SDK並向前兼容 2.0 版本。x和更新版本。它們包括 2.0 版中存在的新功能。x，但包含針對此遷移設計的安全預設值。它們允許您升級您的AWS KMS主要金鑰提供者 (如有必要)，並使用演算法套件完全部署，這些演算法套件可以透過金鑰承諾解密文字。

- 取代已停用的元素，包括舊版的建構函式AWS KMS主金鑰提供者。在 中蟒蛇，請務必開啟棄用警告。在最新的 1 中棄用的代碼元素。x版本從 2.0 版本中刪除。x和更新版本。
- 明確設定您的承諾政策`ForbidEncryptAllowDecrypt`。雖然這是最新 1 中唯一有效的值。x版本，當您使用此版本中引入的 API 時，需要此設定。當您遷移到 2.0 版時，它可以防止您的應用程式拒絕在沒有金鑰承諾的情況下加密的加密文字。x和更新版本。如需詳細資訊，請參閱 the section called "設定承諾產品原則"。
- 如果您使用AWS KMS主金鑰提供者，您必須將舊版主金鑰提供者更新為支援的主金鑰提供者嚴格模式和搜尋模式。此更新是必需的適用於 JAVA 的 AWS Encryption SDK、適用於 Python 的 AWS Encryption SDK，以及AWS加密 CLI。如果您在探索模式下使用主要金鑰提供者，建議您實作探索篩選器，以限制特定金鑰的包裝金鑰AWS 帳戶。此更新是可選的，但它是最佳實務我們推薦。如需詳細資訊，請參閱 更新AWS KMS主要金鑰提供者。
- 如果您使用AWS KMS探索鑰匙圈，我們建議您加入探索篩選器，將解密中使用的包裝金鑰限制為特定金鑰AWS 帳戶。此更新是可選的，但它是最佳實務我們推薦。如需詳細資訊，請參閱 正在更新 AWS KMS鑰匙圈。

階段 2：將您的應用程式更新至最新版本

部署最新的 1 之後。x版本成功到所有主機，您可以升級到 2.0 版本。x和更新版本。2.0 版x包括所有早期版本的重大變更AWS Encryption SDK。不過，如果您進行階段 1 中建議的程式碼變更，您可以避免在移轉至最新版本時發生錯誤。

在您更新至最新版本之前，請確認您的承諾產品原則已一致設定為`ForbidEncryptAllowDecrypt`。然後，根據您的數據配置，您可以按照自己的步調遷移到`RequireEncryptAllowDecrypt`然後到默認設置，`RequireEncryptRequireDecrypt`。我們建議使用一系列轉換步驟，如下列模式。

1. 開頭為您的承諾政策設定為`ForbidEncryptAllowDecrypt`。所以此AWS Encryption SDK可以使用密鑰承諾解密消息，但尚未使用密鑰承諾進行加密。

2. 當您就緒後，將您的承諾政策更新至`RequireEncryptAllowDecrypt`。所以此AWS Encryption SDK開始加密您的資料主要承諾。它可以在有和沒有密鑰承諾的情況下解密密文。

   在更新承諾政策之前`RequireEncryptAllowDecrypt`，確認您最新的 1。xversion 會部署到所有主機，包括解密您產生的加密文字的任何應用程式的主機。的版本AWS Encryption SDK1.7 版之前的版本。x無法解密使用金鑰承諾加密的郵件。

   這也是將指標新增至應用程式的好時機，以測量您是否仍在處理密文，而不需要關鍵承諾。這可協助您判斷何時可以安全地將承諾用量原則設定更新為`RequireEncryptRequireDecrypt`。對於某些應用程式 (例如那些在 Amazon SQS 佇列中加密訊息的應用程式)，這可能表示等待時間足夠長，以致所有在舊版本下加密的加密文字都已重新加密或刪除。對於其他應用程式 (例如加密的 S3 物件)，您可能需要下載、重新加密和重新上傳所有物件。

3. 如果您確定沒有任何未使用金鑰承諾的加密郵件，您可以將承諾用戶原則更新至`RequireEncryptRequireDecrypt`。此值可確保您的資料永遠透過金鑰承諾加密和解密。此設定為預設值，因此您不需要明確設定，但我們建議您這麼做。一個明確的設置將援助除錯以及如果您的應用程式在沒有金鑰承諾的情況下遇到加密文字加密時可能需要的任何潛在回復。

更新AWS KMS主要金鑰提供者

要遷移到最新的 1. x 版本的AWS Encryption SDK，然後再到 2.0 版本。 x 或更新版本，您必須使用在嚴謹模式或探索模式下明確建立的主金鑰提供者來取代舊AWS KMS版主金鑰提供者。1.7 版。 x 並在 2.0 版中刪除。 x. 使用適用於 JAVA 的 AWS Encryption SDK、適用於 Python 的 AWS Encryption SDK和AWS加密 CLI 的應用程式和指令碼需要此變更。本節中的範例將說明如何更新程式碼。

Note

在 Python 中，打開棄用警告。這將幫助您識別需要更新的代碼部分。

如果您使用的是AWS KMS主要金鑰 (而非主要金鑰提供者)，您可以略過此步驟。 AWS KMS主金鑰不會被淘汰或移除。它們僅使用您指定的包裝密鑰進行加密和解密。

本節中的範例著重於您需要變更的程式碼元素。如需更新程式碼的完整範例，請參閱您程式設計語言之GitHub儲存庫的「範例」一節。此外，這些範例通常使用關鍵 ARN 來表示AWS KMS keys。當您建立用於加密的主金鑰提供者時，您可以使用任何有效的AWS KMS金鑰識別碼來代表AWS KMS key. 當您建立用於解密的主要金鑰提供者時，您必須使用金鑰 ARN。

進一步了解遷移

對於所有AWS Encryption SDK使用者，請瞭解如何在中設定承諾產品原則the section called “設定承諾產品原則”。

對於適用於 C 的 AWS Encryption SDK和適用於 JavaScript 的 AWS Encryption SDK使用者，請瞭解中金鑰圈的選擇性更新正在更新AWS KMS鑰匙圈。

主題

- 遷移至嚴謹模
- 移轉至探索模式

遷移至嚴謹模

更新到最新的 1 後。 x 版本的AWS Encryption SDK，以嚴格模式中的主要金鑰提供者取代您的舊版主金鑰提供者。在嚴謹模式下，您必須指定加密和解密時要使用的包裝金鑰。僅AWS Encryption SDK使用您指定的環繞鍵。已取代的主金鑰提供者可以使用任何AWS KMS key加密資料金鑰 (包括AWS KMS keys在不同AWS 帳戶和區域) 的資料來解密資料。

1.7AWS Encryption SDK 版中引入了嚴格模式下的主密鑰提供程序。 x. 它們會取代舊式主金鑰提供者，而這些提供者在 1.7 中已淘汰。 x 並在 2.0 中刪除。 x. 在嚴謹模式中使用主要金鑰提供者是AWS Encryption SDK最佳作法。

下列程式碼會在嚴謹模式中建立主金鑰提供者，供您用來加密和解密。

Java

此範例代表使用的應用程式中的程式碼是 1.6.2 版或更早版本的適用於 JAVA 的 AWS Encryption SDK.

此程式碼會使用此`KmsMasterKeyProvider.builder()`方法來實體化使用 1AWS KMS key 做為包裝金鑰的AWS KMS主要金鑰提供者。

```
// 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();
```

此範例代表使用 1.7 版應用程式中的程式碼。 x 或更新版本的適用於 JAVA 的 AWS Encryption SDK. 如需完整範例，請參閱 BasicEncryptionExample.java。

在前面的例子中使用的`Builder.build()`和`Builder.withKeysForEncryption()`方法在 1.7 版中被棄用。 x 並從 2.0 版本中刪除。 x.

若要更新為嚴謹模式主要金鑰提供者，此程式碼會以呼叫新方法來取代對已停用`Builder.buildStrict()`方法的呼叫。這個例子指定一個AWS KMS key作為包裝鍵，但該`Builder.buildStrict()`方法可以採用多個列表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

此範例代表使用 1.4.1 版的應用程式中的程式碼適用於 Python 的 AWS Encryption SDK。此代碼使用`KMSMasterKeyProvider`，在 1.7 版中已棄用。 x 並從 2.0 版本中刪除。 x. 解密時，它會使用任何加密AWS KMS key的資料金鑰，而不考慮AWS KMS keys您指定的。

請注意，未KMSMasterKey被棄用或刪除。加密和解密時，它僅使用AWS KMS key您指定的。

```
# 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]
)
```

此範例代表使用 1.7 版應用程式中的程式碼。 的 x 的適用於 Python 的 AWS Encryption SDK。如需完整範例，請參閱 basic_encryption.py。

若要更新為嚴格模式主要金鑰提供者，此程式碼會以呼叫來KMSMasterKeyProvider()取代對的呼叫StrictAwsKmsMasterKeyProvider()。

```
# 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

此範例顯示如何使用加AWS密 CLI 1.1.7 或更早版本進行加密和解密。

在版本 1.1.7 及更早版本中，加密時，您可以指定一個或多個主要金鑰 (或包裝金鑰)，例如AWS KMS key. 解密時，除非您使用自訂的主要金鑰提供者，否則無法指定任何包裝金鑰。加AWS密 CLI 可以使用任何加密資料金鑰的包裝金鑰。

```
\\ 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 .
```

此範例顯示如何使用加密 CLI 1.7 版進行AWS加密和解密。 x 或更新版本。如需完整範例，請參閱示例AWS加密 CLI。

1.7 版。`--master-keys` x 並在 2.0 版中刪除。 x. 它取代了 by-`-wrapping-keys` 參數，這在加密和解密命令中是必需的。此參數支援嚴格模式和探索模式。嚴格模式是確保您使用想要的包裝鍵的AWS Encryption SDK最佳做法。

若要升級到嚴格模式，請在加密和解密時，使用`--wrapping-keys`參數的 key 屬性來指定包裝金鑰。

```
\\ 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 .
```

移轉至探索模式

1.7 版。 x，對AWS KMS主金鑰提供者使用嚴格模式是AWS Encryption SDK最佳做法，也就是說，在加密和解密時指定包裝金鑰。加密時，您必須一律指定包裝金鑰。但在某些情況下，指定AWS KMS keys用於解密的關鍵 ARN 是不切實際的。例如，如果您在加密AWS KMS keys時使用別名來識別，如果在解密時必須列出關鍵 ARN，就會失去別名的好處。此外，由於探索模式下的主要金鑰提供者的行為類似於原始主要金鑰提供者，因此您可以暫時使用它們做為移轉策略的一部分，然後在嚴謹模式中升級為主要金鑰提供者。

在這種情況下，您可以在探索模式下使用主要金鑰提供者。這些主要金鑰提供者不允許您指定包裝金鑰，因此您無法使用它們進行加密。解密時，他們可以使用任何加密資料金鑰的包裝金鑰。但與傳統主金鑰提供者 (其行為方式相同) 不同，您可以明確地在探索模式下建立這些提供者。在探索模式下使用主要金鑰提供者時，您可以限制可用於特定金鑰的包裝金鑰AWS 帳戶。此探索篩選器是選擇性的，但這是我們建議使用的最佳作法。如需有關AWS分割區和帳戶的資訊，請參閱 AWS 一般參考.

下列範例會以嚴格模式建立AWS KMS主金鑰提供者以供加密，並在探索模式下建立AWS KMS主金鑰提供者以進行解密。探索模式下的主要金鑰提供者使用探索篩選器，將用於解密的包裝金鑰限制為aws分割區和特定範例AWS 帳戶。儘管在這個非常簡單的示例中不需要帳戶過濾器，但是當一個應用程序加密數據並且不同的應用程序解密數據時，這是非常有益的最佳實踐。

Java

此範例代表使用 1.7 版應用程式中的程式碼。 x 或更新版本的適用於 JAVA 的 AWS Encryption SDK. 如需完整範例，請參閱 DiscoveryDecryptionExample.java。

若要在嚴謹模式中實體化主金鑰提供者以進行加密，此範例會使用此`Builder.buildStrict()`方法。要在發現模式中實例化主密鑰提供程序以進行解密，它使用該`Builder.buildDiscovery()`方法。該`Builder.buildDiscovery()`方法採`DiscoveryFilter`用限制指定AWS分區和帳戶AWS KMS keys中的範圍。AWS Encryption SDK

```
// 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

此範例代表使用 1.7 版應用程式中的程式碼。 x 或更新版本的適用於 Python 的 AWS Encryption SDK. 如需完整範例，請參閱 discovery_kms_provider.py。

若要在嚴謹模式中建立主要金鑰提供者以進行加密，此範例使用StrictAwsKmsMasterKeyProvider。若要在探索模式中建立主金鑰提供者以進行解密，它會使DiscoveryAwsKmsMasterKeyProvider用限制在指定AWS分割區和帳戶AWS KMS keys中使用與限制為。DiscoveryFilterAWS Encryption SDK

```
# 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

此範例顯示如何使用加密 CLI 1.7 版進行AWS加密和解密。 x 或更新版本。1.7 版。 x，加密和解密時需要該--wrapping-keys參數。此--wrapping-keys參數支援嚴格模式和探索模式。如需完整範例，請參閱the section called “範例”。

在加密時，這個例子指定了一個包裝密鑰，這是必需的。解密時，它會使用具有值的--wrapping-keys參數discovery屬性，明確選擇探索模式true。

為了將探索模式中AWS Encryption SDK可以使用的包裝索引鍵限制為特定的索引鍵AWS 帳戶，此範例使用--wrapping-keys參數的discovery-partition和discovery-account屬性。只有當屬性設定為時，這些選用discovery屬性才有效true。您必須同時使用discovery-partition和discovery-account屬性；兩者都不是單獨有效的。

```
\\ 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 .
```

正在更新AWS KMS鑰匙圈

所以此AWS KMS鑰匙圈中適用於 C 的 AWS Encryption SDK，該AWS Encryption SDK針對 .NET，以及適用於 JavaScript 的 AWS Encryption SDK支持最佳實務允許您在加密和解密時指定包裝密鑰。如果您建立AWS KMS探索鑰匙圈，您明確地這樣做。

Note

最早的版本AWS Encryption SDK.NET 是 3.0 版。x。所有版本AWS Encryption SDK為了 .NET 支持 2.0 中引入的安全最佳實踐。x的AWS Encryption SDK。您可以安全地升級到最新版本，而無需任何代碼或數據更改。

當您更新到最新的 1.x的版本AWS Encryption SDK，您可以使用探索篩選限制包裝鍵AWS KMS探索鑰匙圈或者AWS KMS區域探索鑰匙圈特別是解密到那些時使用AWS 帳戶。篩選探索金鑰圈是AWS Encryption SDK 最佳實務。

本節中的範例將說明如何將探索篩選器新增至AWS KMS區域探索鑰匙圈。

進一步了解移轉

適用於所有AWS Encryption SDK使用者, 瞭解如何設定承諾原則the section called “設定承諾產品原則”。

對於適用於 JAVA 的 AWS Encryption SDK、適用於 Python 的 AWS Encryption SDK，以及AWS加密 CLI 使用者，瞭解主金鑰提供者的必要更新the section called “更新AWS KMS主要金鑰提供者”。

您的應用程序中可能有如下所示的代碼。本範例會建立AWS KMS區域探索金鑰圈只能在美國西部 (奧勒岡) (us-west-2) 區域中使用包裝金鑰圈。此範例代表AWS Encryption SDK1.7 版。x。但是，它在 1.7 版本中仍然有效。x和更新版本。

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 })
```

從 1.7 版中開始。x，您可以將探索篩選器新增至任何AWS KMS探索鑰匙圈。此探索篩選器會限制AWS KMS keys那AWS Encryption SDK可以用於解密指定分區和帳戶中的那些。在使用此程式碼之前，請視需要變更分割區，並以有效的帳戶 ID 取代範例帳戶 ID。

C

如需完整的範例，請參閱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

如需完整的範例，請參閱已找到的 .ts 檔案。

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

設定承諾產品原則

金鑰承諾可確保您的加密資料永遠解密為相同的純文字。若要提供此安全性屬性，請從 1.7 版開始。 x，AWS Encryption SDK使用具有關鍵承諾的新算法套件。若要判斷您的資料是否已透過金鑰承諾進行加密和解密，請使用承諾產品原則組態設定。使用關鍵承諾對資料進行加密和解密是AWS Encryption SDK最佳做法。

設定承諾產品原則是移轉程序第二個步驟的重要組成部分 — 從最新的 1 移轉。 x 版本的AWS Encryption SDK到 2.0 版本。 x 及更新版本。設定並變更承諾用戶原則之後，請務必先徹底測試您的應用程式，然後再將其部署到生產環境中。如需移轉指引，請參閱如何遷移和部署AWS Encryption SDK。

承諾用戶原則設定在 2.0 版中具有三個有效值。 x 及更新版本。在最新的 1. x 版（從 1.7 版開始。 x），只有`ForbidEncryptAllowDecrypt`有效。

- `ForbidEncryptAllowDecrypt`—AWS Encryption SDK 無法使用密鑰承諾進行加密。它可以解密有或沒有密鑰承諾加密的密文。

  在最新的 1. x 版本，這是唯一有效的值。它可確保您在完全準備好使用金鑰承諾進行解密之前，不會使用金鑰承諾進行加密。明確設定值可防止承諾用戶原則在升級至 2.0 版`require-encrypt-require-decrypt`時自動變更為。 x 或更新版本。相反地，您可以分階段遷移承諾產品原則。
- `RequireEncryptAllowDecrypt`—AWS Encryption SDK 始終使用關鍵承諾進行加密。它可以解密有或沒有密鑰承諾加密的密文。此值會在 2.0 版中加入。 x.
- `RequireEncryptRequireDecrypt`—AWS Encryption SDK 始終使用關鍵承諾進行加密和解密。此值會在 2.0 版中加入。 x. 它是 2.0 版本中的默認值。 x 及更新版本。

在最新的 1. x 版本，唯一有效的承諾產品原則值為ForbidEncryptAllowDecrypt。遷移至 2.0 版之後。 x 或更新版本，您可以在準備就緒時分階段變更承諾產品原則。在您確定沒有任何沒有金鑰承諾的情況下加密的郵件RequireEncryptRequireDecrypt之前，請勿將承諾用量原則更新至。

這些範例說明如何在最新的 1 中設定承諾產品原則。 x 版本和 2.0 版本。 x 及更新版本。該技術取決於您的編程語言。

進一步了解移轉

對於適用於 JAVA 的 AWS Encryption SDK適用於 Python 的 AWS Encryption SDK、和AWS加密 CLI，請瞭解中對主要金鑰提供者所需的變更the section called “更新AWS KMS主要金鑰提供者”。

對於適用於 C 的 AWS Encryption SDK和適用於 JavaScript 的 AWS Encryption SDK，瞭解中金鑰圈的選擇性更新正在更新AWS KMS鑰匙圈。

如何設定承諾政策

您用來設定承諾產品原則的技術會因每種語言實作而有些微差異。這些範例向您展示如何執行此操作。在變更承諾政策之前，請先檢閱中的多階段方法如何遷移和部署。

C

從 1.7 版開始。 x 的適用於 C 的 AWS Encryption SDK，您可以使用aws_cryptosdk_session_set_commitment_policy函數在加密和解密工作階段上設定承諾政策。您設定的承諾用量原則會套用至在該工作階段上呼叫的所有加密和解密作業。

aws_cryptosdk_session_new_from_keyring和aws_cryptosdk_session_new_from_cmm函數在 1.7 版中已淘汰。 x 並在 2.0 版中刪除。 x. 這些函數由aws_cryptosdk_session_new_from_keyring_2返回會話的aws_cryptosdk_session_new_from_cmm_2函數替換。

當您在最新aws_cryptosdk_session_new_from_keyring_2的 1aws_cryptosdk_session_new_from_cmm_2 中使用和時。 x 版本中，您需要使用COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT承諾政策值調用該aws_cryptosdk_session_set_commitment_policy函數。在 2.0 版中。 x 和更高版本，調用此函數是可選的，它需要所有有效的值。2.0 版的預設承諾產品原則。 x 和更高版本是COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT。

如需完整範例，請參閱 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

此require-encrypt-require-decrypt值是 .NET 的所有版本中的預設承諾原則。AWS Encryption SDK您可以將其設定為最佳實務，但這並非必要。不過，如果您使用AWS Encryption SDK for .NET 來解密密碼文字，而該加密文字是由AWS Encryption SDK不含金鑰承諾的其他語言實作所加密，您就必須將承諾原則值變更為REQUIRE_ENCRYPT_ALLOW_DECRYPT或FORBID_ENCRYPT_ALLOW_DECRYPT。否則，嘗試解密文會失敗。

在 .NET 中AWS Encryption SDK，您可以在的執行個體上設定承諾原則AWS Encryption SDK。使用CommitmentPolicy參數實例化一個AwsEncryptionSdkConfig對象，並使用配置對象創建實AWS Encryption SDK例。然後，呼叫已設定AWS Encryption SDK執行個體的Encrypt()和Decrypt()方法。

此範例會將承諾產品原則設定為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

若要在AWS加密 CLI 中設定承諾產品原則，請使用`--commitment-policy`參數。此參數在 1.8 版推出。 x.

在最新的 1. x 版本，當您在`--encrypt`或`--decrypt`命令中使用`--wrapping-keys`參`--commitment-policy`數時，需要具有該`forbid-encrypt-allow-decrypt`值的參數。否則，`--commitment-policy`參數無效。

在 2.1 版中。 x 及更高版本，該`--commitment-policy`參數是可選的，默認為該`require-encrypt-require-decrypt`值，如果沒有密鑰承諾，它不會加密或解密任何加密的加密文本。但是，我們建議您在所有加密和解密呼叫中明確設定承諾政策，以協助進行維護和解密。

此範例會設定承諾產品原則。它也會使用從`--wrapping-keys` 1.8 版開始取代`--master-keys`參數的參數。 x. 如需詳細資訊，請參閱 the section called “更新AWS KMS主要金鑰提供者”。如需完整範例，請參閱示例AWS加密 CLI。

```
\\ 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

從 1.7 版開始。 x 的適用於 JAVA 的 AWS Encryption SDK，您可以在您的執行個體上設定承諾原則`AwsCrypto`，代表用AWS Encryption SDK戶端的物件。此承諾用戶端原則設定會套用至在該用戶端上呼叫的所有加密和解密作業。

`AwsCrypto()`構造函數在最新的 1 中被棄用。 x 版本的適用於 JAVA 的 AWS Encryption SDK 和已在 2.0 版中移除。 x. 它會由新`Builder`類別、`Builder.withCommitmentPolicy()`方法和`CommitmentPolicy`列舉型別取代。

在最新的 1. x 版本中，`Builder`類需要該`Builder.withCommitmentPolicy()`方法和`CommitmentPolicy.ForbidEncryptAllowDecrypt`參數。從 2.0 版開始。 x，方`Builder.withCommitmentPolicy()`法是可選的；預設值為`CommitmentPolicy.RequireEncryptRequireDecrypt`。

如需完整範例，請參閱 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

從 1.7 版開始。 x 的適用於 JavaScript 的 AWS Encryption SDK，您可以在呼叫具現化AWS Encryption SDK用戶端的新`buildClient`函式時設定承諾原則。`buildClient`函數採用代表承諾原則的列舉值。當您加密`encrypt`和解密時，它會傳回更新的`decrypt`功能，以及強制執行承諾政策的功能。

在最新的 1. x 版本，該`buildClient`函數需要`CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT`參數。從 2.0 版開始。 x，承諾產品原則引數是選擇性的，預設值為`CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`。

Node.js 和瀏覽器的代碼是相同的，除了瀏覽器需要一個語句來設置憑據。

下列範例會使用AWS KMS金鑰環加密資料。新`buildClient`功能會將承諾原則設定為`FORBID_ENCRYPT_ALLOW_DECRYPT`，最新 1 中的預設值。 x 范圍內。升級`encrypt`後的`buildClient`傳回`decrypt`功能會強制執行您設定的承諾產品原則。

```
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

從 1.7 版開始。 x 的適用於 Python 的 AWS Encryption SDK，您可以在執行個體上設定承諾原則EncryptionSDKClient，代表AWS Encryption SDK用戶端的新物件。您設定的承諾用量原則會套用至使用該用戶端執行個體的所有encrypt和decrypt呼叫。

在最新的 1. x 版本中，EncryptionSDKClient構造函數需要CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT枚舉值。從 2.0 版開始。 x，承諾產品原則引數是選擇性的，預設值為CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT。

此範例使用新的EncryptionSDKClient建構函式，並將承諾原則設定為 1.7。 x 預設值。構造函數實例化一個代表AWS Encryption SDK. 當您呼叫此用戶端上的encryptdecrypt、或stream方法時，它們會強制執行您設定的承諾原則。這個範例也會針對StrictAwsKmsMasterKeyProvider類別使用 new 建構函式，這個建構函式會在加密和解密 AWS KMS keys時指定。

如需完整範例，請參閱 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
)
```

移轉至最新版本的疑難排解

在將您的應用程式更新至 2.0 版。x或更高版本AWS Encryption SDK，更新到最新的 1。x的版本AWS Encryption SDK並完全部署它。這將幫助您避免在更新到 2.0 版時可能遇到的大多數錯誤。x和更新版本。如需詳細指引 (包括範例)，請參閱遷移您的AWS Encryption SDK。

> **Important**
>
> 請驗證您最新的 1。x版。x或更高版本AWS Encryption SDK。

> **Note**
>
> AWSCLI (加密)：本指南的參考資料至 1.7 版。x的AWS Encryption SDK適用於 1.8 版。x的AWSCLI 的 Encryption。本指南的參考資料至 2.0 版。x的AWS Encryption SDK適用於 2.1.x的AWSCLI 的 Encryption。
> 最初發布了新的安全功能AWS加密 CLI 1.7 版。x2.0 和 2.0.x。但是，AWS加密版本 1.8.x取代 1.7 版。x和AWS加密x2.0 版。x。如需詳細資訊，請參閱相關安全性諮詢在aws-encryption-sdk-cli儲存庫 GitHub。

本主題旨在協助您辨識並解決您可能遇到的最常見錯誤。

主題

- 已取代或移除的物件

- 組態衝突：承諾政策和演算法套件
- 組態衝突：承諾政策和密文
- 金鑰履約承諾驗證失敗
- 其他加密失敗
- 其他解密失敗
- 轉返考量

已取代或移除的物件

2.0 版。x包含幾項重大變更，包括移除 1.7 版中已淘汰的舊版建構函式、方法、函數和類別。x。若要避免編譯器錯誤、匯入錯誤、語法錯誤和找不到符號錯誤 (視您的程式設計語言而定)，請先升級到最新的 1。x的版本AWS Encryption SDK為您的編程語言。(此版必須是 1.7 版。x或更高版本。) 在使用最新的 1.x版本，您可以在移除原始符號之前開始使用取代元素。

如果您需要升級至 2.0 版。x或稍後立即，諮詢變更用於您的編程語言，並用更改日誌建議的符號替換舊式符號。

組態衝突：承諾政策和演算法套件

如果您指定的算法套件與您的承諾政策，對加密的呼叫失敗組態衝突錯誤。

為了避免這種類型的錯誤，請不要指定演算法套件。根據預設，AWS Encryption SDK選擇與承諾政策相容的最安全演算法。不過，如果您必須指定演算法套件 (例如未簽署的演算法套件)，請務必選擇與承諾產品原則相容的演算法套件。

承諾政策	相容演算法套件
ForbidEncryptAllowDecrypt	任何演算法套件無主要承諾，例如： `AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384` (03 78) (含簽署) `AES_256_GCM_IV12_TAG16_HKDF_SHA256` (01 78) (未簽署)
RequireEncryptAllowDecrypt RequireEncryptRequireDecrypt	任何演算法套件取代為主要承諾，例如： `AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384` (05 78) (含簽署)

承諾政策	相容演算法套件
	AES_256_GCM_HKDF_SHA512_COMMIT_KEY (04 78) (未簽署)

如果您在未指定演算法套件時遇到此錯誤，表示您可能已選擇衝突的演算法套件密碼材料管理(厘米). 默認 CMM 不會選擇衝突的算法套件，但自定義 CMM 可能會選擇。如需說明，請參閱您自訂 CMM 的說明文件。

組態衝突：承諾政策和密文

所以此RequireEncryptRequireDecrypt 承諾政策不允許AWS Encryption SDK解密未加密的郵件關鍵承諾。如果你問AWS Encryption SDK要解密沒有密鑰承諾的消息，它會返回一個組態衝突錯誤。

若要避免此錯誤，請在設定RequireEncryptRequireDecrypt承諾政策，請確保所有加密而不承諾金鑰的加密文本都經過密鑰承諾進行解密和重新加密，或者由不同的應用程序處理。如果您遇到這個錯誤，您可以傳回衝突密文字的錯誤，或是暫時變更承諾原則RequireEncryptAllowDecrypt。

如果因為升級到 2.0 版而遇到此錯誤。x或 1.7 之前的版本更高版本。x沒有先升級到最新的 1。x版（1.7 版。x或更高版本），考慮還原到最新的 1。x在升級到 2.0 版之前，對該版本進行版本並部署到所有主機。x或更高版本。如需協助，請參閱 如何遷移和部署AWS Encryption SDK。

金鑰履約承諾驗證失敗

當您解密使用金鑰承諾加密的郵件時，您可能會收到金鑰履約承諾驗證失敗錯誤訊息。這表示解密呼叫失敗，因為加密郵件與訊息的唯一資料金鑰不相同。通過在解密過程中驗證數據密鑰，關鍵承諾防止您解密可能導致多個純文本的消息。

此錯誤表示您嘗試解密的加密訊息未傳回AWS Encryption SDK。這可能是手動製作的訊息或資料損毀的結果。如果您遇到此錯誤，您的應用程式可以拒絕郵件並繼續，或停止處理新郵件。

其他加密失敗

加密失敗可能會因多種原因而失敗。您無法使用AWS KMS探索鑰匙圈或一個探索模式下的主要金鑰提供者加密郵件。

請務必使用包裝您擁有的金鑰來指定金鑰環或主要金鑰提供者許可用於加密。如需有關權限的說明AWS KMS keys，請參閱檢視金鑰政策和判斷的存取權AWS KMS key在AWS Key Management Service開發人員指南。

其他解密失敗

如果您嘗試解密加密的郵件失敗，則表示AWS Encryption SDK無法（或不會）解密消息中的任何加密數據密鑰。

如果您使用指定包裝金鑰的金鑰環或主要金鑰提供者，AWS Encryption SDK僅使用您指定的包裝鍵。驗證您正在使用您打算並且有包裝鍵`kms:Decrypt`許可至少一個包裝鍵。如果您正在使用AWS KMS keys，作為後備，您可以嘗試使用AWS KMS探索鑰匙圈或一個探索模式下的主要金鑰提供者。如果作業成功，在傳回純文字之前，請確認用來解密訊息的金鑰是否為您信任的金鑰。

轉返考量

如果您的應用程序無法加密或解密數據，通常可以通過更新代碼符號，密鑰環，主密鑰提供程序來解決問題，或承諾政策。但是，在某些案例中，您可能會判斷最好將您的應用程式轉返至先前版本的AWS Encryption SDK。

如果您必須回滾，請謹慎操作。的版本AWS Encryption SDK1.7 版之前的版本。x無法解密使用加密的密文關鍵承諾。

- 從最新的 1 回滾。x版AWS Encryption SDK通常是安全的。您可能必須復原對程式碼所做的變更，才能使用舊版不支援的元件和物件。
- 一旦您開始使用金鑰承諾產品加密 (將承諾產品原則設定為`RequireEncryptAllowDecrypt`) 於 2.0 版。x或更高版，您可以轉返 1.7 版。x，但不適用於任何早期版本。的版本AWS Encryption SDK1.7 版之前的版本。x無法解密使用加密的密文關鍵承諾。

如果您在所有主機都可以使用金鑰承諾進行解密之前不小心啟用了使用金鑰承諾加密，則最好是繼續推出，而不是復原。如果訊息是暫時性的或可以安全地捨棄，則您可能會考慮回復而遺失訊息。如果需要復原，您可以考慮編寫一個解密和重新加密所有郵件的工具。

常見問答集

- AWS Encryption SDK與 AWS 開發套件有何不同？
- 的運作方式AWS Encryption SDK與 Amazon S3 加密用户端有所不同？
- AWS Encryption SDK支援哪些密碼編譯演算法？何者為預設值？
- 初始化向量 (IV) 如何產生？存放在哪裡？
- 每個資料金鑰如何產生、加密及解密？
- 如何追蹤用來加密資料的資料金鑰？
- AWS Encryption SDK如何存放加密的資料金鑰與其加密的資料？
- AWS Encryption SDK的訊息格式會對我的加密資料增加多少負擔？
- 我是否可以使用自己的主金鑰提供者？
- 我是否可以在一個以上的包裝金鑰下加密資料？
- 我可以使用 AWS Encryption SDK加密哪些資料類型？
- AWS Encryption SDK如何加密和解密輸入/輸出 (I/O) 串流？

AWS Encryption SDK與 AWS 開發套件有何不同？

所以此AWS開發套件提供用於與 Amazon Web Services 互動的資料庫 (AWS），包括AWS Key Management Service(AWS KMS。一些語言實現AWS Encryption SDK，例如AWS Encryption SDK.NET，始終需要AWSSDK 使用相同的編程語言。其他語言實現需要相應的AWSSDK 僅當您使用AWS KMS密鑰在密鑰環或主密鑰提供程序中。如需詳細資訊，請參程式設計語言的主題：AWS Encryption SDK程式設計語言。

您可以使用AWS要與之交互的 SDKAWS KMS，包括對少量數據進行加密和解密（使用對稱加密密鑰最多 4,096 字節），以及生成用於客户端加密的數據密鑰。但是，在生成數據密鑰時，您必須管理整個加密和解密過程，包括使用AWS KMS，安全地丟棄明文數據密鑰，存儲加密的數據密鑰，然後解密數據密鑰並解密您的數據。所以此AWS Encryption SDK為您處理此過程。

所以此AWS Encryption SDK提供使用行業標準和最佳實務來加密和解密資料的程式庫。它生成數據密鑰，在您指定的包裝密鑰下對其進行加密，並返回已加密訊息，這是一個便攜式資料對象，其中包含已加密資料和解密所需的加密資料金鑰。當需要解密時，您將傳入加密消息和至少一個包裝密鑰（可選），並且AWS Encryption SDK傳回純文字資料。

您可以將AWS KMS keys作為包裝鍵在AWS Encryption SDK，但並不是必要項目。您可以使用您生成的加密密鑰以及從密鑰管理器或本地硬件安全模塊生成的加密密鑰。您可以使用AWS Encryption SDK即使您沒有AWS帳戶。

的運作方式AWS Encryption SDK與 Amazon S3 加密用户端有所不同？

所以此Amazon S3 加密用户端中的AWS軟體開發套件會為您存放在 Amazon Simple Storage Service (Amazon S3) 中的資料，提供加密和解密。這些用户端與 Amazon S3 緊密結合，僅適用於存放在這裏的資料。

AWS Encryption SDK則對您存放在任何地方的資料提供加密和解密。所以此AWS Encryption SDK 和 Amazon S3 加密用户端會產生不同資料格式的加密文字，因此並不相容。

AWS Encryption SDK支援哪些密碼編譯演算法？何者為預設值？

所以此AWS Encryption SDK使用 Galois/Counter Mode (GCM) 中的進階加密標準 (AES) 對稱演算法 (簡稱為 AES-GCM) 來加密資料。它允許您從多種對稱和非對稱算法中進行選擇，以加密數據加密的數據密鑰。

對於 AES-GCM，默認算法套件是 AES-GCM，具有 256 位密鑰、密鑰派生 (HKDF)、數字簽名，以及關鍵承諾。AWS Encryption SDK也支援 192 位元和 128 位元的加密金鑰和加密演算法，無需數位簽章和金鑰承諾。

在所有情況下，初始化向量 (IV) 的長度一律為 12 個位元組；身份驗證標籤的長度一律為 16 個位元組。預設情況下，軟體開發套件會使用資料金鑰做為輸入到 extract-and-expand 金鑰衍生函數 (HKDF) 來衍生 AES-GCM 加密金鑰，也可以新增 Elliptic Curve 數位簽章演算法 (ECDSA) 簽章。

如需選擇使用哪個演算法的相關資訊，請參閱支援的演算法套件。

如需受支援演算法的詳細資訊，請參閱演算法參考。

初始化向量 (IV) 如何產生？存放在哪裡？

所以此AWS Encryption SDK使用決定性方法來為每個框架建構不同的 IV 值。此過程可確保 IVs 永遠不會在消息中重複。（在 1.3.0 版之前的適用於 JAVA 的 AWS Encryption SDK與適用於 Python 的 AWS Encryption SDK，AWS Encryption SDK為每個框架隨機產生唯一的 IV 值。）

IV 存放在已加密訊息中，AWS Encryption SDK傳回。如需詳細資訊，請參閱 AWS Encryption SDK 訊息格式參考。

每個資料金鑰如何產生、加密及解密？

方法取決於您使用的金鑰環或主金鑰提供者。

所以此AWS KMS密鑰環和主密鑰提供程序AWS Encryption SDK使用AWS KMS GenerateDataKeyAPI 操作來生成每個數據密鑰並在其包裝密鑰下對其進行加密。要在其他 KMS 密鑰下加密數據密鑰的副本，它們使用AWS KMS 加密operation. 要解密數據密鑰，它們使用AWS KMS 解密operation. 如需詳細資訊，請參閱AWS KMSKeyring中的AWS Encryption SDK規格 GitHub。

其他密鑰環使用每種編程語言的最佳實踐方法生成數據密鑰、加密和解密。有關詳細信息，請參閱框架部分的AWS Encryption SDK規格 GitHub。

如何追蹤用來加密資料的資料金鑰？

AWS Encryption SDK會為您妥善處理。當您加密資料時，軟體開發套件會加密資料金鑰並將加密的金鑰與加密的資料一起存放在它傳回的已加密訊息中。當您解密資料時，AWS Encryption SDK 會從加密的訊息中擷取加密的資料金鑰、將其解密，然後使用它來解密資料。

AWS Encryption SDK如何存放加密的資料金鑰與其加密的資料？

AWS Encryption SDK中的加密操作會傳回已加密訊息，這是包含已加密資料及其加密資料金鑰的單一資料結構。訊息格式包含兩個部分：標題與本文。訊息標題會包含加密的資料金鑰，以及說明訊息標題組成方式的資訊。訊息內文包含加密的資料。如果算法套件包含數位簽章，則訊息格式會包含頁腳，其中包含簽章。如需詳細資訊，請參閱 AWS Encryption SDK 訊息格式參考。

AWS Encryption SDK的訊息格式會對我的加密資料增加多少負擔？

AWS Encryption SDK增加的負擔量取決於數個因素，包括下列幾項：

- 純文字資料的大小
- 所使用的支援演算法
- 是否提供額外的驗證資料 (AAD)，以及該 AAD 的長度
- 包裝金鑰或主金鑰的數量和類型
- 框架大小 (使用具框架資料時)

當您使用AWS Encryption SDK的默認配置（一個AWS KMS key做為包裝金鑰（或主金鑰）、沒有 AAD、無框架資料和帶簽章的加密演算法），負擔大約是 600 個位元組。一般而言，您可以合理假設 AWS Encryption SDK增加 1 KB 或更少的負擔，不包含提供的 AAD。如需詳細資訊，請參閱 AWS Encryption SDK 訊息格式參考。

我是否可以使用自己的主金鑰提供者？

是。實作詳細資訊會因您使用的受支援程式設計語言而異。但是，所有支持的語言都允許您定義自定義加密材料管理器 (CMM)、主密鑰提供程序、密鑰環、主密鑰和包裝密鑰。

我是否可以在一個以上的包裝金鑰下加密資料？

是。當金鑰位於不同區域或無法用於解密的情況下，您可以使用額外的包裝金鑰（或主金鑰）來加密資料金鑰。

若要在多個包裝金鑰下加密資料，請建立具有多個包裝金鑰的金鑰環或主金鑰提供者。使用 keyring 時，您可以建立具有多個包裝金鑰的單一 keyring 或多個 keyring。

當您使用多個包裝密鑰加密數據時，AWS Encryption SDK使用一個包裝鍵來生成一個純文本的數據密鑰。數據密鑰是唯一的，在數學上與包裝鍵無關。操作會傳回純文字資料金鑰和由包裝金鑰加密的資料金鑰副本。然後，加密方法會使用其他包裝金鑰來加密資料金鑰。產生的已加密訊息包含已加密資料和每個包裝金鑰的加密資料金鑰。

您可以使用加密操作中所用的任何一個包裝金鑰來解密加密的訊息。所以此AWS Encryption SDK 使用包裝金鑰來解密加密的資料金鑰。接著，使用純文字資料金鑰來解密資料。

我可以使用 AWS Encryption SDK加密哪些資料類型？

大多數程式設計語言實作AWS Encryption SDK可以加密原始位元組 (位元組陣列)、I/O 串流 (位元組串流) 和字串。所以此AWS Encryption SDK不支援 I/O 串流。我們提供每個受支援程式設計語言的範例程式碼。

AWS Encryption SDK如何加密和解密輸入/輸出 (I/O) 串流？

AWS Encryption SDK會建立加密或解密串流，來包裝基礎 I/O 串流。加密或解密串流會在讀取或寫入呼叫上執行密碼編譯操作。例如，它可以讀取基礎串流上的純文字資料，並將其加密再傳回結果。或者，它可以讀取基礎串流的加密文字，並將其解密再傳回結果。我們提供每個用於加密和解密流的範例程式碼支援程式設計語言，支持流式傳輸。

所以此AWS Encryption SDK不支援 I/O 串流。

AWS Encryption SDK 參考

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

會 AWS Encryption SDK 使用支援的演算法傳回包含加密資料和對應加密資料金鑰的單一資料結構或訊息。下列主題說明演算法和資料結構。使用此資訊來建置程式庫，用以讀取和寫入與此開發套件相容的加密文字。

主題

- AWS Encryption SDK 訊息格式參考
- AWS Encryption SDK 訊息格式範例
- AWS Encryption SDK的內文額外的驗證資料 (AAD) 參考
- AWS Encryption SDK 演算法參考
- AWS Encryption SDK 初始化向量參考
- AWS KMS 分層鑰匙圈技術細節

AWS Encryption SDK 訊息格式參考

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

中的加密操作 AWS Encryption SDK 返回包含加密數據（密文）和所有加密數據密鑰的單個數據結構或加密消息。您需要了解訊息格式，才能掌握此資料結構，或建置可加以讀寫的程式庫。

訊息格式包含兩個部分：標題與本文。在某些情況下，訊息格式還會包含第三個部分，也就是頁尾。訊息格式會定義網路位元組順序中的有序序列，又稱為 Big Endian 格式。訊息格式會以標題做為開頭，然後依序接著本文與頁尾 (如果有)。

AWS Encryption SDK 使用兩種訊息格式版本之一所支援的演算法套件。沒有金鑰承諾的演算法套件會使用訊息格式版本 1。具有金鑰承諾的演算法套件會使用訊息格式版本 2。

主題

- 標題結構
- 本文結構
- 頁尾結構

標題結構

訊息標題會包含加密的資料金鑰，以及說明訊息標題組成方式的資訊。下表說明在郵件格式版本 1 和 2 中形成標頭的欄位。位元組依顯示順序附加。

[不存在] 值表示該欄位不存在於該版本的郵件格式中。粗體文字表示每個版本中不同的值。

Note

您可能需要水平或垂直捲動，才能查看此資料表中的所有資料。

標題結構

欄位	訊息格式版本 1 長度 (位元組)	訊息格式版本 2 長度 (位元組)
版本	1	1
Type	1	不存在
演算法 ID	2	2
訊息 ID	16	32

欄位	訊息格式版本 1 長度 (位元組)	訊息格式版本 2 長度 (位元組)
AAD 長度	2 當加密內容為空時，2 位元組 AAD 長度欄位的值為 0。	2 當加密內容為空時，2 位元組 AAD 長度欄位的值為 0。
AAD	變數. 此欄位的長度會顯示在前 2 個位元組 (AAD 長度欄位) 中。 當加密內容為空白時，標題中不會有 AAD 欄位。	變數. 此欄位的長度會顯示在前 2 個位元組 (AAD 長度欄位) 中。 當加密內容為空白時，標題中不會有 AAD 欄位。
加密資料金鑰計數	2	2
加密資料金鑰	變數. 取決於加密資料金鑰的數量與每個加密資料金鑰的長度。	變數. 取決於加密資料金鑰的數量與每個加密資料金鑰的長度。
內容類型	1	1
已預留	4	不存在
IV 長度	1	不存在
框架長度	4	4
演算法套件資料	不存在	變數。取決於用來產生訊息的演算法。
標題驗證	變數. 取決於用來產生訊息的演算法。	變數. 取決於用來產生訊息的演算法。

版本

此本訊息格式的版本。該版本是編碼為字節或十六進制表示法02的 1 01 或 2

Type

此本訊息格式的類型。類型會指出結構的種類。所支援的唯一類型會如客戶驗證的加密資料所述。其類型值為 128，並在十六進位表示法中編碼為 80。

郵件格式版本 2 中不存在此欄位。

演算法 ID

所使用演算法的識別碼。它會以 2 位元組數值表示，並解譯為 16 位元的無符號整數。如需演算法的詳細資訊，請參閱AWS Encryption SDK 演算法參考。

訊息 ID

識別訊息的隨機產生值。訊息 ID：

- 唯一識別加密訊息。
- 以弱式繫結方式，將訊息標題繫結至訊息本文。
- 提供安全的機制來搭配多個加密訊息重複使用資料金鑰。
- 避免在 AWS Encryption SDK中意外重複使用資料金鑰，或用盡所有金鑰。

這個值是 128 位元的訊息格式版本 1，而在版本 2 中則為 256 位元。

AAD 長度

額外的驗證資料 (AAD) 的長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含 AAD 的位元組數量。

當加密內容為空時，「AAD 長度」欄位的值為 0。

AAD

額外的驗證資料。AAD 為加密內容編碼方式，它是金鑰值組的陣列，當中的每個金鑰與值皆為 UTF-8 編碼字元組成的字串。加密內容會轉換為位元組序列，並用於表示 AAD 數值。當加密內容為空白時，標題中不會有 AAD 欄位。

如果使用含有簽章的演算法，加密內容就必須包含金鑰值組 `{'aws-crypto-public-key', Qtxt}`。Qtxt 代表根據 SEC 1 版本 2.0 壓縮，然後以 base64 編碼的橢圓曲線點 Q。加密內容可以包含其他值，但所建構 AAD 的長度上限為 2^16 - 1 位元組。

下表將說明 AAD 的組成欄位。金鑰會依照 UTF-8 字元碼來遞增排序金鑰值組。位元組依顯示順序附加。

AAD 結構

欄位	長度 (位元組)
金鑰值組計數	2
金鑰長度	2
金鑰	變數. 等於前 2 個位元組中指定的值 (金鑰長度)。
值長度	2
Value	變數. 等於前 2 個位元組中指定的值 (數值長度)。

金鑰值組計數

AAD 中的金鑰值組數量。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出 AAD 中的金鑰值組數量。AAD 中的金鑰值組數量上限為 2^16 - 1。

如果沒有加密內容，或加密內容為空白，此欄位就不會出現在 AAD 結構中。

金鑰長度

金鑰值組的金鑰長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含金鑰的位元組數量。

金鑰

金鑰值組的金鑰。它會以 UTF-8 編碼位元組序列表示。

值長度

金鑰值組的值長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含值的位元組數量。

Value

金鑰值組的值。它會以 UTF-8 編碼位元組序列表示。

加密資料金鑰計數

加密資料金鑰的數量。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出加密資料金鑰的數量。每則訊息中的加密資料金鑰數目上限為 65,535 (2^16-1)。

加密資料金鑰

加密資料金鑰的序列。序列長度取決於加密資料金鑰的數量與每個加密資料金鑰的長度。序列會包含至少一個加密資料金鑰。

下表將說明每個加密資料金鑰的組成欄位。位元組依顯示順序附加。

加密資料金鑰結構

欄位	長度 (位元組)
金鑰提供者 ID 長度	2
金鑰提供者 ID	變數. 等於前 2 個位元組中指定的值 (金鑰提供者 ID 長度)。
金鑰提供者資訊長度	2
金鑰提供者資訊	變數. 等於前 2 個位元組中指定的值 (金鑰提供者資訊長度)。
加密資料金鑰長度	2
加密資料金鑰	變數. 等於前 2 個位元組中指定的值 (加密資料金鑰長度)。

金鑰提供者 ID 長度

金鑰提供者識別碼的長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含金鑰提供者 ID 的位元組數量。

金鑰提供者 ID

金鑰提供者識別碼。它會用來指出加密資料金鑰的提供者，以而且可供擴充。

金鑰提供者資訊長度

金鑰提供者資訊的長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含金鑰提供者資訊的位元組數量。

金鑰提供者資訊

金鑰提供者資訊。它會取決於金鑰提供者。

如果 AWS KMS 是主金鑰提供者或您正在使用金 Amazon AWS KMS 環，此值 ARN 包含. AWS KMS key

加密資料金鑰長度

加密資料金鑰的長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含加密資料金鑰的位元組數量。

加密資料金鑰

加密資料金鑰。這是由金鑰提供者所加密的資料加密金鑰。

內容類型

加密資料的類型，無論是非框架或框架。

> **Note**
>
> 盡可能使用框架數據。僅 AWS Encryption SDK 支援舊版使用的非框架資料。的某些語言實作仍然 AWS Encryption SDK 可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

框架數據分為相同長度的部分; 每個部分分別加密。具框架內容為類型 2，並在十六進位表示法中編碼為位元組 `02`。

非框架數據不被分割; 它是一個單一的加密 Blob。無框架內容為類型 1，並在十六進位表示法中編碼為位元組 `01`。

已預留

已保留的 4 位元組序列。此值必須為 0。它會在十六進位表示法中編碼為 `00 00 00 00` (也就是以 4 位元組序列表示且等於 0 的 32 位元整數值)。

郵件格式版本 2 中不存在此欄位。

IV 長度

初始向量 (IV) 的長度。它會以 1 元組數值表示，並解譯為 8 位元的無符號整數，指出包含 IV 的位元組數量。此值取決於產生訊息的演算法 IV 位元組值。

此欄位不存在於郵件格式版本 2 中，此版本 2 只支援在郵件標頭中使用確定性 IV 值的演算法套件。

框架長度

框架數據的每一幀的長度。它是一個 4 字節值，解釋為 32 位無符號整數，用於指定每個幀中的字節數。當資料為非框架時，也就是當`Content Type`欄位的值為 1 時，此值必須為 0。

Note

盡可能使用框架數據。僅 AWS Encryption SDK 支援舊版使用的非框架資料。的某些語言實作仍然 AWS Encryption SDK 可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

演算法套件資料

產生訊息的演算法所需的補充資料。長度和內容由演算法決定。它的長度可能是 0。

郵件格式版本 1 中不存在此欄位。

標題驗證

標題驗證取決於產生訊息的演算法。標題驗證會計算整個標題。當中包含一個 IV 與一個驗證標籤。位元組依顯示順序附加。

標題驗證結構

欄位	1.0 版本的長度 (位元組)	2.0 版本的長度（字節）
IV	變數. 取決於產生訊息的演算法 IV 位元組值。	N/A
驗證標籤	變數. 取決於產生訊息的演算法驗證標籤位元組值。	變數. 取決於產生訊息的演算法驗證標籤位元組值。

IV

用來計算標題驗證標籤的初始向量 (IV)。

此欄位不會出現在郵件格式版本 2 的標頭中。郵件格式版本 2 僅支援在郵件標頭中使用確定性 IV 值的演算法套件。

驗證標籤

標題的驗證值。這個值會用來驗證標題的整體內容。

本文結構

訊息本文會包含加密資料，也就是所謂的加密文字。本文的結構會取決於內容類型 (無框架或具框架)。下面章節將說明每個內容類型的訊息本文格式。郵件內文結構在郵件格式版本 1 和 2 中是相同的。

主題

- 無框架資料
- 具框架資料

無框架資料

無框架資料會於單一 Blob 中使用唯一的 IV 與本文 AAD 進行加密。

Note

盡可能使用框架數據。僅 AWS Encryption SDK 支援舊版使用的非框架資料。的某些語言實作仍然 AWS Encryption SDK 可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

下表將說明無框架資料的組成欄位。位元組依顯示順序附加。

無框架本文結構

欄位	長度 (以位元組為單位)
IV	變數. 等於標題 IV 長度位元組中指定的值。
加密內容長度	8
加密內容	變數. 等於前 8 個位元組中指定的值 (加密內容長度)。
驗證標籤	變數. 取決於使用的演算法實作。

IV

搭配加密演算法使用的初始向量 (IV)。

加密內容長度

加密內容或加密文字的長度。它會以 8 元組數值表示，並解譯為 64 位元的無符號整數，指出包含加密內容的位元組數量。

就技術層面而言，允許的最大值為 2^63 - 1，或 8 Exbibyte (8 EiB)。不過，基於實作演算法所帶來的限制，現實層面的最大值則為 2^36 - 32，也就是 64 Gibibyte (64 GiB)。

Note

基於語言限制，此 SDK 的 Java 實作會將這個值進一步限制在 2^31 - 1 內，也就是 20 Gibibyte (2 GiB)。

加密內容

由加密演算法傳回的加密內容 (加密文字)。

驗證標籤

本文的驗證值。這個值會用來驗證訊息本文。

具框架資料

在具框架資料中，純文字資料被劃分為等長的部分，稱為框架。使用獨 AWS Encryption SDK 特的 IV 和主體 A AD 分別加密每個幀。

Note

盡可能使用框架數據。僅 AWS Encryption SDK 支援舊版使用的非框架資料。的某些語言實作仍然 AWS Encryption SDK 可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

框架長度，即框架中加密內容的長度，可能會因每個訊息而有所不同。框架中的位元組數目上限為 2^32 - 1。訊息中框架的位元組數目上限為 2^32 - 1。

框架包含兩種類型：一般與最終。每個訊息都必須組成為或包含最終框架。

訊息中的所有一般框架都有相同的框架長度。最終框架可以有不同的框架長度。

具框架資料中的框架組成會因加密內容的長度而有所不同。

- 等於影格長度 — 當加密的內容長度與一般影格的影格長度相同時，訊息可能包含包含資料的一般影格，後面接著零 (0) 長度的最後一個影格。或者，訊息的組成可以為僅包含資料的最終框架。在此情況下，最終框架的框架長度與一般框架相同。
- 框架長度的倍數 — 當加密的內容長度是一般影格之影格長度的精確倍數時，訊息可能會以包含資料的一般框架結束，接著是零 (0) 長度的最後一個影格。或者，訊息的結尾可以為包含資料的最終框架。在此情況下，最終框架的框架長度與一般框架相同。
- 不是影格長度的倍數 — 當加密的內容長度不是一般影格的影格長度的精確倍數時，最後一個影格會包含剩餘的資料。最終框架的框架長度小於一般框架的框架長度。
- 小於框架長度 — 當加密的內容長度小於一般框架的框架長度時，訊息會由包含所有資料的最終影格組成。最終框架的框架長度小於一般框架的框架長度。

下表將說明框架的組成欄位。位元組依顯示順序附加。

具框架本文結構、一般框架

欄位	長度 (以位元組為單位)
序號	4
IV	變數. 等於標題 IV 長度位元組中指定的值。
加密內容	變數. 等於標題框架長度中指定的值。
驗證標籤	變數. 取決於使用的演算法，會於標題的演算法 ID 中指定。

序號

框架序號。這個序號為框架的遞增計數器編號。它會以 4 位元組數值表示，並解譯為 32 位元的無符號整數。

具框架資料必須從序號 1 開始編號。後續框架必須依序編號，並比前一個框架多出 1。否則，加密程序就會停止，並回報錯誤。

IV

框架的初始向量 (IV)。SDK 會使用決定性方法，為訊息中的每個框架建構不同的 IV。它的長度會由使用的演算法套件指定。

加密內容

由加密演算法傳回的框架加密內容 (加密文字)。

驗證標籤

框架的驗證值。這個值會用來驗證整個框架。

具框架本文結構、最終框架

欄位	長度 (以位元組為單位)
序號結束	4
序號	4
IV	變數. 等於標題 IV 長度位元組中指定的值。
加密內容長度	4
加密內容	變數. 等於前 4 個位元組中指定的值 (加密內容長度)。
驗證標籤	變數. 取決於使用的演算法，會於標題的演算法 ID 中指定。

序號結束

最終框架的指標。這個值會在十六進位表示法中編碼為 4 位元組的 FF FF FF FF。

序號

框架序號。這個序號為框架的遞增計數器編號。它會以 4 位元組數值表示，並解譯為 32 位元的無符號整數。

具框架資料必須從序號 1 開始編號。後續框架必須依序編號，並比前一個框架多出 1。否則，加密程序就會停止，並回報錯誤。

IV

框架的初始向量 (IV)。SDK 會使用決定性方法，為訊息中的每個框架建構不同的 IV。IV 長度會由演算法套件指定。

加密內容長度

加密內容的長度。它會以 4 元組數值表示，並解譯為 32 位元的無符號整數，指出包含框架加密內容的位元組數量。

加密內容

由加密演算法傳回的框架加密內容 (加密文字)。

驗證標籤

框架的驗證值。這個值會用來驗證整個框架。

頁尾結構

如果使用含有簽章的演算法，訊息格式就會包含頁尾。郵件頁尾包含透過郵件標頭和內文計算的數位簽章。下表將說明頁尾的組成欄位。位元組依顯示順序附加。郵件格式版本 1 和 2 中的郵件頁尾結構相同。

頁尾結構

欄位	長度 (以位元組為單位)
簽章長度	2
簽章	變數. 等於前 2 個位元組中指定的值 (簽章長度)。

簽章長度

簽章的長度。它會以 2 元組數值表示，並解譯為 16 位元的無符號整數，指出包含簽章的位元組數量。

簽章

簽章本身。

AWS Encryption SDK 訊息格式範例

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

下列主題顯示 AWS Encryption SDK 郵件格式的範例。每個範例顯示十六進位表示法的原始位元組，接著說明這些位元組代表什麼。

主題

- 框架數據（消息格式第 1 版）
- 框架數據（消息格式第 2 版）
- 非框架數據（消息格式第 1 版）

框架數據（消息格式第 1 版）

下列範例顯示訊息格式版本 1 中的框架資料的訊息格式。

```
+--------+
| 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
```

框架數據（消息格式第 2 版）

下面的例子顯示了在消息格式版本 2 幀數據的消息格式。

```
+--------+
| 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
```

非框架數據（消息格式第 1 版）

以下範例顯示無框架資料的訊息格式。

> **Note**
>
> 盡可能使用框架數據。僅 AWS Encryption SDK 支援舊版使用的非框架資料。的某些語言實作仍然 AWS Encryption SDK 可以產生非框架加密文字。所有支持的語言實現都可以解密框架和非框加密文本。

```
+--------+
| 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
```

AWS Encryption SDK的內文額外的驗證資料 (AAD) 參考

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

您必須為每個密碼編譯操作，將額外的驗證資料 (AAD) 提供給 AES-GCM 演算法。對於具框架和無框架內文資料都是如此。如需有關 AAD 及其在 Galois/計數器模式 (GCM) 中的用法的詳細資訊，請參閱區塊加密操作模式的建議：Galois/計數器模式 (GCM) 和 GMAC。

下表說明內文 AAD 的組成欄位。位元組依顯示順序附加。

內文 AAD 結構

欄位	長度 (以位元組為單位)
訊息 ID	16
內文 AAD 內容	變數. 請參閱下列清單中的內文 AAD 內容。
序號	4
內容長度	8

訊息 ID

在訊息標頭中設定的相同 訊息 ID 值。

內文 AAD 內容

由所使用內文資料類型決定的 UTF-8 編碼值。

對於無框架資料，請使用值 `AWSKMSEncryptionClient Single Block`。

對於具框架資料中的一般框架，請使用值 `AWSKMSEncryptionClient Frame`。

對於具框架資料中的最終框架，請使用值 `AWSKMSEncryptionClient Final Frame`。

序號

4 位元組值，解譯為 32 位元的無正負號整數。

對於具框架資料，這是框架序號。

對於無框架資料，請使用值 1，在十六進位表示法中編碼為 4 位元組 `00 00 00 01`。

內容長度

提供給演算法進行加密的純文字資料長度，以位元組為單位。它是 8 位元組值，並解譯為 64 位元的無正負號整數。

AWS Encryption SDK 演算法參考

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

如果您正在構建自己的庫，該庫可以讀取和寫入與兼容的密文 AWS Encryption SDK，則需要了解如何 AWS Encryption SDK 實現支持的算法套件來加密原始數據。

AWS Encryption SDK 支援下列演算法套件。所有 AES-GCM 演算法套件都具有 12 位元組初始化向量和 16 位元組 AES-GCM 驗證標記。預設演算法套件會隨 AWS Encryption SDK 版本和選取的金鑰承諾原則而有所不同。如需詳細資訊，請參閱承諾政策和演算法套件。

AWS Encryption SDK 演算法套件

演算法 ID	訊息格式版本	加密演算法	資料金鑰長度 (位元)	金鑰衍生演算法	簽章演算法	關鍵承諾演算法	演算法套件資料長度 (位元組)
05 78	0X02	AES-G 厘米	256	香港文憑基金與 SHA-512	ECDSA，P 84 和 SHA-384 式	香港文憑基金與 SHA-512	32（主要承諾）
04 78	0X02	AES-G 厘米	256	香港文憑基金與 SHA-512	無	香港文憑基金與 SHA-512	32（主要承諾）
03 78	0x01	AES-G 厘米	256	HKDF，SH 384 式	ECDSA，P 84 和 SHA-384 式	無	N/A
03 46	0x01	AES-G 厘米	192	HKDF，SH 384 式	ECDSA，P 84 和 SHA-384 式	無	N/A
02 14	0x01	AES-G 厘米	128	HKDF，搭配 SHA-256	ECDSA，搭配 P-256 和 SHA-256	無	N/A
01 78	0x01	AES-G 厘米	256	HKDF，搭配 SHA-256	無	無	N/A
01 46	0x01	AES-G 厘米	192	HKDF，搭配 SHA-256	無	無	N/A

演算法 ID	訊息格式版本	加密演算法	資料金鑰長度 (位元)	金鑰衍生演算法	簽章演算法	關鍵承諾演算法	演算法套件資料長度 (位元組)
`01 14`	0x01	AES-G 厘米	128	HKDF，搭配 SHA-256	無	無	N/A
`00 78`	0x01	AES-G 厘米	256	無	無	無	N/A
`00 46`	0x01	AES-G 厘米	192	無	無	無	N/A
`00 14`	0x01	AES-G 厘米	128	無	無	無	N/A

演算法 ID

可唯一識別演算法實作的 2 位元組十六進位值。這個值會儲存在加密文字的郵件標頭中。

訊息格式版本

郵件格式的版本。具有金鑰承諾用量的演算法套件會使用訊息格式版本 2 (0x02)。沒有金鑰承諾的演算法套件會使用訊息格式版本 1 (0x01)。

演算法套件資料長度

演算法套件特定資料的位元組長度。只有郵件格式版本 2 (0x02) 才支援此欄位。在郵件格式版本 2 (0x02) 中，此資料會出現在郵件標頭的`Algorithm suite data`欄位中。支援金鑰承諾用量的演算法套件會使用 32 個位元組做為金鑰承諾字串。如需詳細資訊，請參閱此清單中的金鑰承諾演算法。

資料金鑰長度

資料金鑰的長度 (以位元為單位)。 AWS Encryption SDK 支援 256 位元、192 位元和 128 位元金鑰。資料金鑰由金鑰圈或主要金鑰產生。

在某些實現中，此數據密鑰被用作基於 HMAC 的 extract-and-expand 密鑰派生函數（HKDF）的輸入。HKDF 的輸出做為加密演算法中的資料加密金鑰。如需詳細資訊，請參閱此清單中的金鑰衍生演算法。

加密演算法

使用的加密演算法的名稱和模式。算法套件中 AWS Encryption SDK 使用高級加密標準（AES）加密算法與伽羅瓦/計數器模式（GCM）。

關鍵承諾演算法

用來計算金鑰履約承諾字串的演算法。輸出會儲存在訊息標頭的`Algorithm suite data`欄位中，用來驗證金鑰承諾的資料金鑰。

如需將金鑰承諾加入演算法套件的技術說明，請參閱密碼學 ePrint 封存中的金鑰提交 AEAD。

金鑰衍生演算法

基於 HMAC 的 extract-and-expand 密鑰派生功能（HKDF）用於導出數據加密密鑰。RFC 58 AWS Encryption SDK 69 中定義的香港港元化基金的用途。

沒有金鑰承諾的演算法套件 (演算法 ID `01xx` —`03xx`)

- 使用的哈希函數是 SHA-384 或 SHA-256，這取決於算法套件。
- 對於擷取步驟：
  - 不使用 salt。根據 RFC，鹽被設置為一個零字符串。字串長度等於雜湊函數輸出的長度，SHA-384 為 48 個位元組，SHA-256 則為 32 個位元組。
  - 輸入鍵控材料是來自金鑰圈或主要金鑰提供者的資料金鑰。
- 對於擴展步驟：
  - 輸入虛擬亂數金鑰是來自擷取步驟的輸出。
  - 輸入信息是算法 ID 和消息 ID 的串聯（按該順序）。
  - 輸出鍵合材料的長度為資料鍵長度。此輸出將當做加密演算法中的資料加密金鑰。

具有金鑰承諾量的演算法套件 (演算法 ID `04xx` 和`05xx`)

- 使用的哈希函數是 SHA-512。
- 對於擷取步驟：
  - 鹽是 256 位元的密碼編譯隨機值。在郵件格式版本 2 (0x02) 中，此值會儲存在`MessageID`欄位中。
  - 初始金鑰材料是來自金鑰圈或主要金鑰提供者的資料金鑰。

- 對於擴展步驟：
  - 輸入虛擬亂數金鑰是來自擷取步驟的輸出。
  - 索引鍵標籤是以大端位元組順序排列的字DERIVEKEY串 UTF-8 編碼位元組。
  - 輸入信息是算法 ID 和密鑰標籤的串聯（按該順序）。
  - 輸出鍵合材料的長度為資料鍵長度。此輸出將當做加密演算法中的資料加密金鑰。

訊息格式版本

與演算法套件搭配使用的訊息格式版本。如需詳細資訊，請參閱 訊息格式參考。

簽章演算法

用於在密文標頭和內文上生成數字簽名的簽名算法。 AWS Encryption SDK 使用具有下列細節的橢圓曲線數位簽章演算法 (ECDSA)：

- 使用的橢圓曲線為 P-384 或 P-256 曲線 (如演算法 ID 所指定)。這些曲線定義於數位簽章標準 (DSS) (FIPS PUB 186-4) 中。
- 使用的雜湊函數是 SHA-384 (搭配 P-384 曲線) 或 SHA-256 (搭配 P-256 曲線)。

AWS Encryption SDK 初始化向量參考

本頁面上提供的參考可讓您建置自己的並與 AWS Encryption SDK相容的加密儲存庫。如果您不是自己建置相容的加密儲存庫，可能不需要此資訊。

若要在其 AWS Encryption SDK 中一種受支援的程式設計語言中使用，請參閱程式設計語言。

如需定義適當 AWS Encryption SDK 實作之元素的規格，請參閱中的AWS Encryption SDK 規格 GitHub。

會提 AWS Encryption SDK 供所有支援演算法套件所需的初始化向量 (IV)。開發套件使用框架序號來建構 IV，因此同一訊息沒有兩個框架可使用相同的 IV。

每個 96 位元 (12 位元組) IV 由兩個大端序位元組陣列建構得來，並以下列順序串連：

- 64 位元：0 (保留以供日後使用)。
- 32 位元：框架序號。對於標頭驗證標籤，這個值全是零。

在引進資料金鑰快取之前， AWS Encryption SDK 一直使用新的資料金鑰來加密每則訊息，並隨機產生所有 IV。因為資料金鑰從未重複使用，因此隨機產生的 IV 在密碼演算法上是安全的。當開發套件引進資料金鑰快取 (特意重複使用資料金鑰)，我們也變更開發套件產生 IV 的方式。

在訊息中使用無法重複的決定性 IV，會大幅增加可在單一資料金鑰下安全執行的呼叫數量。此外，快取的資料金鑰一律使用演算法套件搭配金鑰衍生函數。使用具有虛擬隨機金鑰衍生函數的確定性 IV，從資料金鑰衍生加密金鑰可讓您在不超過加密界限的情況下 AWS Encryption SDK 加密 2^32 個訊息。

AWS KMS 分層鑰匙圈技術細節

AWS KMS 分層密鑰環使用 unqiue 數據密鑰來加密每個字段，並使用從活動分支密鑰派生的唯一包裝密鑰對每個數據密鑰進行加密。它在計數器模式下使用密鑰派生，並帶有 HMAC SHA-256 的偽隨機函數，以導出具有以下輸入的 32 字節包裝密鑰。

- 一個 16 字節的隨機鹽
- 作用中的分支索引鍵
- 金鑰提供者識別碼 "aws-kms-hierarchy」的 UTF-8 編碼值

階層式金鑰環使用衍生的包裝金鑰，使用具有 16 位元組驗證標籤和下列輸入的 AES-GCM-256 來加密純文字資料金鑰的副本。

- 派生的包裝密鑰被用作 AES-GCM 密鑰
- 資料金鑰會用作 AES-GCM 訊息
- 一個 12 字節的隨機初始化向量（IV）被用作 AES-GCM IV
- 包含下列序列化值的其他驗證資料 (AAD)。

Value	長度 (位元組)	解釋為
"aws-kms-hierarchy"	17	UTF-8 編碼
分支密鑰標識符	變數	UTF-8 編碼
分支密鑰版本	16	UTF-8 編碼
加密內容	變數	UTF-8 編碼的金鑰值配對

AWS Encryption SDK 開發人員指南的文件歷史記錄

此主題描述《AWS Encryption SDK 開發人員指南》的重大更新。

主題

- 最近更新
- 舊版更新

最近更新

下表說明此文件自 2017 年 11 月後的重大變更。除了這裡所列的主要變更外，我們也會經常更新文件以改進說明內容和範例，並且反映您傳送給我們的意見回饋。如要接收重大變更的通知，請訂閱 RSS 摘要。

變更	描述	日期
一般可用性	已新增 AWS KMS ECDH 鑰匙圈和原始 E CDH 鑰匙圈的說明文件。	2024年6月17日
適用於 JAVA 的 AWS Encryption SDK 版本 3.x	適用於 JAVA 的 AWS Encryption SDK 與材料提供者資料庫整合。添加對金鑰環和所需加密內容 CMM 的支援。	2023 年 12 月 6 日
AWS Encryption SDK 對於.NET 版本 4.x	添加對 AWS KMS 階層式金鑰圈、所需加密上下文 CMM 和非對稱 R AWS KMS SA 金鑰圈的支援。	2023 年 10 月 12 日
一般可用性	介紹對 .NET AWS Encryption SDK 的支援。	2022 年 5 月 17 日
文件變更	將 AWS Key Management Service 術語客戶主金鑰 (CMK) 取代為AWS KMS key和 KMS 金鑰。	2021 年 8 月 30 日

一般可用性	增加了對 AWS Key Management Service. (AWS KMS) 多區域鍵。多區域金鑰是不同的 AWS KMS 金鑰，可 AWS 區域 以互換使用，因為它們具有相同的金鑰 ID 和金鑰材料。	2021 年 6 月 8 日
一般可用性	已新增並更新有關改善訊息解密程序的文件。	2021 年 5 月 11 日
一般可用性	新增並更新了 AWS 加密 CLI 1.8 版正式發行版本的文件。x 表示取代 AWS 加密 CLI 版本 1.7。 x 和 AWS 加密 CLI 碼 2.1. x 來取代 AWS 加密 CLI 2.0。 x.	2020 年 10 月 27 日
一般可用性	增加和更新的文檔 1.7 的正式發行 AWS Encryption SDK 版本。 X 和 2.0。 x，包括最佳做法指南、移轉指南、更新的概念、更新的程式設計語言主題、更新的演算法套件參考資料、更新的郵件格式參考，以及新的郵件格式範例。	2020 年 9 月 24 日
一般可用性	新增和更新 適用於 JavaScript 的 AWS Encryption SDK 的正式發行版本文件。	2019 年 10 月 1 日
預覽版本	新增和更新 適用於 JavaScript 的 AWS Encryption SDK 的公開 Beta 版本文件。	2019 年 6 月 21 日
一般可用性	新增和更新 適用於 C 的 AWS Encryption SDK 的正式發行版本文件。	2019 年 5 月 16 日

預覽版本	新增 適用於 C 的 AWS Encryption SDK 預覽版的文件。	2019 年 2 月 5 日
新版本	已新增的命令列介面文件 AWS Encryption SDK。	2017 年 11 月 20 日

舊版更新

下表說明 2017 年 11 月之前對開AWS Encryption SDK 發人員指南進行的重大變更。

變更	描述	日期
新版本	新增說明新功能的資料金鑰快取章節。 新增the section called “初始化向量參考”主題，其中說明開發套件從產生隨機 IV 變更為建構決定性 IV。 新增主the section called “概念”題以說明概念，包括新的密碼編譯材料管理員。	2017 年 7 月 31 日
更新	擴充訊息格式參考文件為加入新的AWS Encryption SDK 參考章節。 已新增關於 AWS Encryption SDK 支援的演算法套件.	2017 年 3 月 21 日
新版本	AWS Encryption SDK 現在支援Python程式設計語言，除了Java.	2017 年 3 月 21 日

變更	描述	日期
初始版本	AWS Encryption SDK 和此文件的初始版本。	2016 年 3 月 22 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。