開發人員指南

AWS 資料庫加密 SDK

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

AWS 資料庫加密 SDK: 開發人員指南

Copyright © 2025 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 資料庫加密 SDK？ ... 1
在開放原始碼儲存庫中開發 ... 2
支援和維護 ... 3
傳送意見回饋 ... 3
概念 ... 3
封套加密 ... 4
資料金鑰 ... 5
包裝金鑰 ... 6
Keyrings ... 6
密碼編譯動作 ... 7
資料描述 ... 8
加密內容 ... 8
密碼編譯資料管理員 ... 8
對稱和非對稱加密 ... 9
金鑰承諾 ... 9
數位簽章 ... 10
運作方式 ... 11
加密和簽署 ... 11
解密和驗證 ... 12
支援的演算法套件 ... 13
預設演算法套件 ... 15
不含 ECDSA 數位簽章的 AES-GCM ... 16
與 互動 AWS KMS ... 18
設定軟體開發套件 ... 20
選取程式設計語言 ... 20
選取包裝金鑰 ... 20
建立探索篩選條件 ... 21
使用多租戶資料庫 ... 23
建立簽章的信標 ... 23
金鑰存放區 ... 30
金鑰存放區術語和概念 ... 30
實作最低權限的許可 ... 31
建立金鑰存放區 ... 31
設定金鑰存放區動作 ... 32

設定您的金鑰存放區動作 ... 33
建立分支金鑰 ... 36
輪換作用中的分支金鑰 ... 39
Keyrings ... 42
keyring 如何運作 ... 43
AWS KMS 鑰匙圈 ... 43
AWS KMS keyring 的必要許可 ... 44
在 AWS KMS keyring AWS KMS keys 中識別 ... 45
建立 AWS KMS keyring ... 46
使用多區域 AWS KMS keys ... 48
使用 AWS KMS 探索 keyring ... 50
使用 AWS KMS 區域探索 keyring ... 52
AWS KMS 階層式 keyring ... 54
運作方式 ... 56
先決條件 ... 57
所需的許可 ... 58
選擇快取 ... 58
建立階層式 keyring ... 67
使用階層式 keyring 進行可搜尋加密 ... 73
AWS KMS ECDH 鑰匙圈 ... 76
AWS KMS ECDH keyrings 的必要許可 ... 77
建立 AWS KMS ECDH keyring ... 78
建立 AWS KMS ECDH 探索 keyring ... 81
原始 AES keyring ... 84
原始 RSA keyring ... 86
原始 ECDH 鑰匙圈 ... 89
建立原始 ECDH keyring ... 90
多重 keyring ... 98
可搜尋加密 ... 102
信標是否適合我的資料集？ ... 103
可搜尋的加密案例 ... 104
信標 ... 106
標準信標 ... 107
複合信標 ... 108
規劃信標 ... 108
多租戶資料庫的考量事項 ... 110

選擇信標類型 ... 110
選擇信標長度 ... 115
選擇信標名稱 ... 120
設定信標 ... 121
設定標準信標 ... 121
設定複合信標 ... 130
範例組態 ... 139
使用信標 ... 144
查詢信標 ... 146
多租戶資料庫的可搜尋加密 ... 147
查詢多租戶資料庫中的信標 ... 150
Amazon DynamoDB ... 152
用戶端加密和伺服器端加密 ... 153
哪些欄位已加密並簽署？ ... 154
加密屬性值 ... 155
簽署項目 ... 156
DynamoDB 中的可搜尋加密 ... 156
使用信標設定次要索引 ... 156
測試信標輸出 ... 158
更新您的資料模型 ... 164
新增 ENCRYPT_AND_SIGN、 SIGN_ONLY和 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性 ... 165
移除現有屬性 ... 166
將現有ENCRYPT_AND_SIGN屬性變更為 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT ... 166
將現有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為 ENCRYPT_AND_SIGN ... 167
新增DO_NOTHING屬性 ... 167
將現有SIGN_ONLY屬性變更為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT ... 168
將現有SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為 SIGN_ONLY ... 169
程式設計語言 ... 169
Java ... 169
.NET ... 201
Rust ... 215
傳統 ... 220
AWS DynamoDB 版本的資料庫加密 SDK 支援 ... 221

運作方式 221
概念 224
密碼編譯資料提供者 228
程式設計語言 255
變更您的資料模型 280
故障診斷 284
DynamoDB Encryption Client 重新命名 288
參考資料 290
材質描述格式 290
AWS KMS 階層式 keyring 技術詳細資訊 293
文件歷史紀錄 295
..... ccxcvii

什麼是 AWS 資料庫加密 SDK？

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

AWS Database Encryption SDK 是一組軟體程式庫，可讓您在資料庫設計中包含用戶端加密。 AWS Database Encryption SDK 提供記錄層級加密解決方案。您可以指定加密的欄位，以及簽章中包含的欄位，以確保資料的真實性。加密傳輸中和靜態的敏感資料有助於確保您的純文字資料不可供任何第三方使用，包括 AWS。 AWS Database Encryption SDK 是根據 Apache 2.0 授權免費提供。

本開發人員指南提供 AWS 資料庫加密 SDK 的概念概觀，包括其架構的簡介、如何保護資料的詳細資訊、與伺服器端加密的差異，以及為您的應用程式選擇關鍵元件以協助您入門的指引。

AWS Database Encryption SDK 支援具有屬性層級加密的 Amazon DynamoDB。

AWS Database Encryption SDK 具有下列優點：

專為資料庫應用程式而設計

您不需要是密碼編譯專家，即可使用 AWS 資料庫加密 SDK。實作包括專為使用現有應用程式而設計的協助程式方法。

在您建立和設定必要的元件之後，加密用戶端會在您將記錄新增至資料庫時，以透明方式加密和簽署您的記錄，並在擷取記錄時驗證和解密記錄。

包含安全加密和簽署

AWS Database Encryption SDK 包含安全實作，這些實作會使用唯一的資料加密金鑰來加密每個記錄中的欄位值，然後簽署該記錄以保護它免受未經授權的變更，例如新增或刪除欄位，或交換加密的值。

使用來自任何來源的密碼編譯資料

AWS Database Encryption SDK 使用 keyring 來產生、加密和解密唯一資料加密金鑰，以保護記錄。Keyrings 會決定加密該資料金鑰的包裝金鑰。

您可以使用來自任何來源的包裝金鑰，包括密碼編譯服務，例如 AWS Key Management Service(AWS KMS) 或 AWS CloudHSM。 AWS Database Encryption SDK 不需要 AWS 帳戶 或任何 AWS 服務。

支援密碼編譯資料快取

AWS KMS 階層式 keyring 是一種密碼編譯資料快取解決方案，透過使用保留在 Amazon DynamoDB 資料表中的 AWS KMS 受保護分支金鑰，以及加密和解密操作中使用的本機快取分支金鑰資料來減少 AWS KMS 呼叫次數。它可讓您在對稱加密 KMS 金鑰下保護密碼編譯資料，而不必 AWS KMS 在每次加密或解密記錄時呼叫 。 AWS KMS 階層式 keyring 是需要將呼叫降至最低的應用程式的最佳選擇 AWS KMS。

可搜尋加密

您可以設計資料庫來搜尋加密的記錄，而無需解密整個資料庫。根據您的威脅模型和查詢需求，您可以使用可搜尋加密，在加密的資料庫上執行完全相符的搜尋或更自訂的複雜查詢。

支援多租戶資料庫結構描述

AWS Database Encryption SDK 可讓您透過隔離每個租戶與不同的加密材料，以保護存放在資料庫的資料與共用結構描述。如果您有多個使用者在資料庫中執行加密操作，請使用其中一個 AWS KMS keyring 為每位使用者提供可在其密碼編譯操作中使用的不同金鑰。如需詳細資訊，請參閱使用多租戶資料庫。

支援無縫結構描述更新

當您設定 AWS 資料庫加密 SDK 時，您會提供密碼編譯動作，告訴用戶端要加密和簽署哪些欄位、要簽署哪些欄位（但不要加密），以及要忽略哪些欄位。使用 AWS Database Encryption SDK 來保護您的記錄之後，您仍然可以變更資料模型。您可以在單一部署中更新您的密碼編譯動作，例如新增或移除加密的欄位。

在開放原始碼儲存庫中開發

AWS Database Encryption SDK 是在 GitHub 上的開放原始碼儲存庫中開發。您可以使用這些儲存庫來檢視程式碼、讀取和提交問題，以及尋找實作特有的資訊。

適用於 DynamoDB 的 AWS 資料庫加密 SDK

- GitHub 上的 aws-database-encryption-sdk-dynamodb 儲存庫支援適用於 Java、.NET 和 Rust 中 DynamoDB 的 AWS 資料庫加密 SDK 最新版本。

  DynamoDB 的 AWS Database Encryption SDK 是 Dafny 的產品，這是一種驗證意識的語言，您可以在其中撰寫規格、實作規格的程式碼，以及測試規格的證明。結果是程式庫，可在確保功能正確性的架構中實作適用於 DynamoDB 的 AWS Database Encryption SDK 功能。

支援和維護

AWS Database Encryption SDK 使用與 AWS SDK 和工具相同的維護政策，包括其版本控制和生命週期階段。最佳實務是，建議您使用 AWS 資料庫加密 SDK 的最新版本進行資料庫實作，並在新版本發佈時升級。

如需詳細資訊，請參閱《 AWS SDKs AWS 和工具參考指南》中的SDKs和工具維護政策。

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 資料庫加密 SDK 中發現潛在的安全性漏洞，請通知 AWS 安全性。請勿建立公有 GitHub 問題。

若要提供有關此文件的意見回饋，請使用任何頁面上的意見回饋連結。

AWS 資料庫加密 SDK 概念

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

本主題說明 AWS 資料庫加密 SDK 中使用的概念和術語。

若要了解 AWS 資料庫加密 SDK 的元件如何互動，請參閱 AWS 資料庫加密 SDK 的運作方式。

若要進一步了解 AWS 資料庫加密 SDK，請參閱下列主題。

- 了解 AWS Database Encryption SDK 如何使用信封加密來保護您的資料。
- 了解信封加密的元素：保護記錄的資料金鑰，以及保護資料金鑰的包裝金鑰。
- 了解決定您使用哪些包裝金鑰的 keyring。
- 了解為您的加密程序新增完整性的加密內容。
- 了解加密方法新增至記錄的材質描述。
- 了解密碼編譯動作，這些動作會告訴 AWS 資料庫加密 SDK 要加密和簽署哪些欄位。

主題

- 封套加密
- 資料金鑰
- 包裝金鑰
- Keyrings
- 密碼編譯動作
- 資料描述
- 加密內容
- 密碼編譯資料管理員
- 對稱和非對稱加密
- 金鑰承諾
- 數位簽章

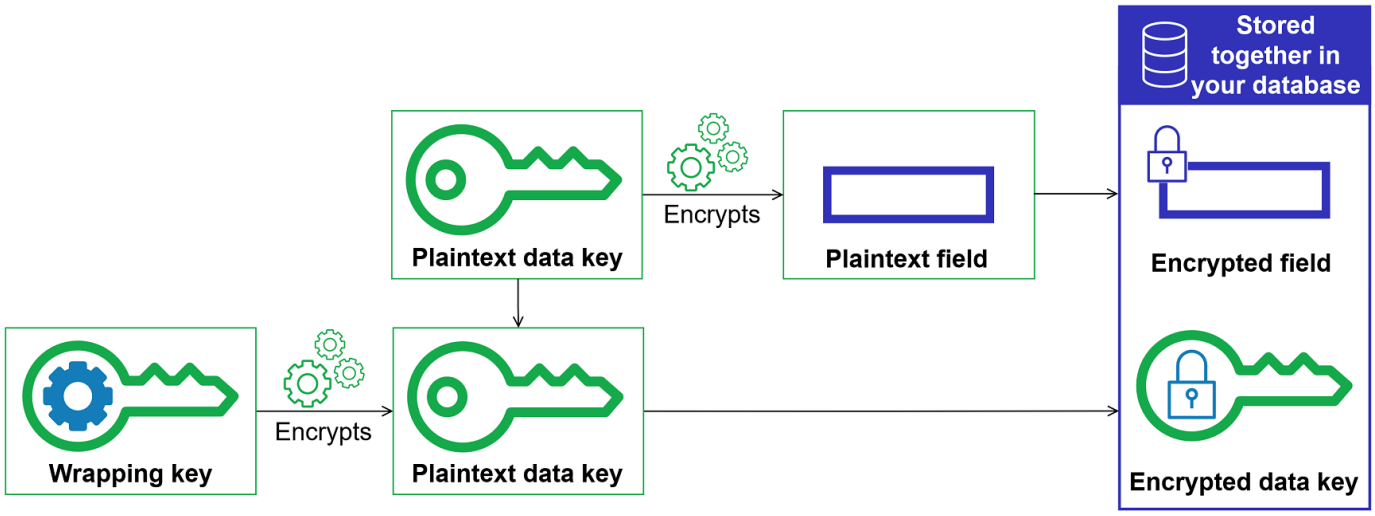
封套加密

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。若要這樣做，您需要另一個加密金鑰，稱為金鑰加密金鑰或包裝金鑰。使用包裝金鑰加密資料金鑰的做法稱為信封加密。

保護資料金鑰

AWS Database Encryption SDK 會使用唯一的資料金鑰加密每個欄位。然後，它會在您指定的包裝金鑰下加密每個資料金鑰。它會將加密的資料金鑰存放在材料描述中。

若要指定包裝金鑰，請使用 keyring。



在多個包裝金鑰下加密相同的資料

您可以使用多個包裝金鑰來加密資料金鑰。您可能想要為不同的使用者提供不同的包裝金鑰，或包裝不同類型的金鑰，或在不同的位置。每個包裝金鑰都會加密相同的資料金鑰。 AWS Database Encryption SDK 會將所有加密的資料金鑰與材料描述中的加密欄位一起存放。

若要解密資料，您需要提供至少一個可解密加密資料金鑰的包裝金鑰。

結合多種演算法的優勢

為了加密您的資料， AWS 資料庫加密 SDK 預設會使用具有 AES-GCM 對稱加密的演算法套件、HMAC 型金鑰衍生函數 (HKDF) 和 ECDSA 簽署。若要加密資料金鑰，您可以指定適合您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。但公有金鑰演算法提供固有的角色區隔。若要結合每個 的優勢，您可以使用公有金鑰加密來加密資料金鑰。

我們建議您盡可能使用其中一個 AWS KMS keyring。當您使用 AWS KMS keyring 時，您可以選擇透過指定非對稱 RSA AWS KMS key 做為包裝金鑰來結合多個演算法的強度。您也可以使用對稱加密 KMS 金鑰。

資料金鑰

資料金鑰是一種加密金鑰， AWS 資料庫加密 SDK 用來加密在密碼編譯動作`ENCRYPT_AND_SIGN`中標記的記錄中的欄位。每個資料金鑰是符合密碼編譯金鑰需求的位元組陣列。 AWS Database Encryption SDK 使用唯一的資料金鑰來加密每個屬性。

您不需要指定、產生、實作、延伸、保護或使用資料金鑰。當您呼叫加密和解密操作時， AWS 資料庫加密 SDK 會為您執行。

為了保護您的資料金鑰， AWS 資料庫加密 SDK 會使用一或多個稱為包裝金鑰的金鑰加密金鑰來加密它們。在 AWS 資料庫加密 SDK 使用您的純文字資料金鑰加密您的資料後，它會盡快將其從記憶體中移除。然後將加密的資料金鑰存放在材料描述中。如需詳細資訊，請參閱 AWS 資料庫加密 SDK 的運作方式。

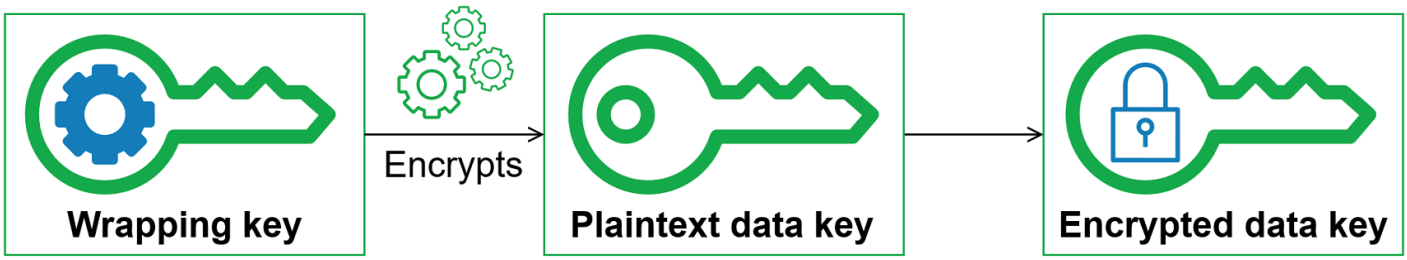
> **Tip**
>
> 在 AWS Database Encryption SDK 中，我們將資料金鑰與資料加密金鑰區分開來。最佳實務是，所有支援的演算法套件都使用金鑰衍生函數。金鑰衍生函數會採用資料金鑰做為輸入，

並傳回實際用來加密記錄的資料加密金鑰。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，包括加密它的包裝金鑰的識別符。此中繼資料可讓 AWS 資料庫加密 SDK 在解密時識別有效的包裝金鑰。

包裝金鑰

包裝金鑰是金鑰加密金鑰， AWS 資料庫加密 SDK 用來加密加密記錄的資料金鑰。每個資料金鑰都可以在一或多個包裝金鑰下加密。當您設定 keyring 時，您可以決定使用哪些包裝金鑰來保護資料。



AWS Database Encryption SDK 支援數個常用的包裝金鑰，例如 AWS Key Management Service(AWS KMS) 對稱加密 KMS 金鑰 （包括多區域 AWS KMS 金鑰) 和非對稱 RSA 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。我們建議您盡可能使用 KMS 金鑰。若要決定應使用哪個包裝金鑰，請參閱選取包裝金鑰。

當您使用信封加密時，您需要保護包裝金鑰免於未經授權的存取。您可以透過下列任何方式執行此操作：

- 使用專為此目的設計的 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，建議您使用 AWS KMS。 AWS Database Encryption SDK 與 整合 AWS KMS ，可協助您保護和使用您的包裝金鑰。

Keyrings

若要指定用於加密和解密的包裝金鑰，請使用 keyring。您可以使用 AWS Database Encryption SDK 提供的 keyring 或設計您自己的實作。

Keying 會產生、加密和解密資料金鑰。它也會產生 MAC 金鑰，用來計算簽章中的雜湊型訊息驗證碼 (HMACs)。當您定義 keyring 時，您可以指定用來加密資料金鑰的包裝金鑰。大多數 keyring 會指定至少一個包裝金鑰，或提供和保護包裝金鑰的服務。加密時， AWS 資料庫加密 SDK 會使用 keyring 中指定的所有包裝金鑰來加密資料金鑰。如需選擇和使用 AWS Database Encryption SDK 定義的 keyring 的說明，請參閱使用 keyring。

密碼編譯動作

密碼編譯動作會告知加密程式要在記錄中的每個欄位執行哪些動作。

密碼編譯動作值可以是下列其中一項：

- 加密和簽署 – 加密 欄位。在簽章中包含加密的欄位。
- 僅限簽署 – 在簽章中包含 欄位。
- 在加密內容中簽署並包含 - 在簽章和加密內容中包含 欄位。

  根據預設，分割區和排序索引鍵是加密內容中包含的唯一屬性。您可以考慮定義其他欄位，`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`以便AWS KMS 階層式 keyring 的分支金鑰 ID 供應商可以識別從加密內容解密所需的分支金鑰。如需詳細資訊，請參閱分支金鑰 ID 供應商。

  > **Note**
  >
  > 若要使用`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀者`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

- 什麼都不做 – 請勿加密或包含簽章中的 欄位。

對於任何可以存放敏感資料的欄位，請使用加密和簽署。對於主索引鍵值 （例如 DynamoDB 資料表中的分割區索引鍵和排序索引鍵），請使用僅簽署或簽署，並包含在加密內容中。如果您指定任何 Sign 並包含在加密內容屬性中，則分割區和排序屬性也必須是 Sign 並包含在加密內容中。您不需要為材料描述指定密碼編譯動作。 AWS Database Encryption SDK 會自動在儲存材料描述的 欄位上簽署。

請謹慎選擇您的密碼編譯動作。如有疑問，請使用加密並簽署。使用 AWS Database Encryption SDK 保護您的記錄後，您無法將現有的 `ENCRYPT_AND_SIGN`、 `SIGN_ONLY`或 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 欄位變更為 `DO_NOTHING`，或變更指派給現有`DO_NOTHING`欄位的密碼編譯動作。不過，您仍然可以對資料模型進行其他變更。例如，您可以在單一部署中新增或移除加密的欄位。

資料描述

材料描述做為加密記錄的標頭。當您使用 AWS 資料庫加密 SDK 加密和簽署欄位時，加密程式會在組合密碼編譯材料時記錄材料描述，並將材料描述存放在加密程式新增至記錄的新欄位 (`aws_dbe_head`) 中。

材料描述是一種可攜式格式化資料結構，其中包含資料金鑰的加密副本和其他資訊，例如加密演算法、加密內容，以及加密和簽署指示。加密程式會在資料描述組合加密和簽署的密碼編譯資料時記錄資料描述。稍後，當它需要組合密碼編譯材料來驗證和解密欄位時，它會使用材料描述作為其指南。

將加密的資料金鑰與加密欄位一起存放，可簡化解密操作，讓您不必獨立於加密的資料存放和管理加密的資料金鑰。

如需材料描述的技術資訊，請參閱 材質描述格式。

加密內容

為了提高密碼編譯操作的安全性， AWS 資料庫加密 SDK 會在所有加密和簽署記錄的請求中包含加密內容。

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。 AWS Database Encryption SDK 包含資料庫的邏輯名稱，以及加密內容中的主索引鍵值（例如 DynamoDB 資料表中的分割區索引鍵和排序索引鍵）。當您加密並簽署欄位時，加密內容會以加密方式繫結至加密記錄，因此解密欄位時需要相同的加密內容。

如果您使用 AWS KMS keyring， AWS 資料庫加密 SDK 也會使用加密內容，在 keyring 進行的呼叫中提供其他已驗證的資料 (AAD) AWS KMS。

每當您使用預設演算法套件時，密碼編譯資料管理員 (CMM) 會將名稱值對新增至加密內容，其中包含預留名稱 `aws-crypto-public-key`和代表公有驗證金鑰的值。公有驗證金鑰會存放在材料描述中。

密碼編譯資料管理員

密碼編譯資料管理員 (CMM) 會組合用於加密、解密和簽署資料的密碼編譯資料。每當您使用預設演算法套件時，密碼編譯材料包含純文字和加密的資料金鑰、對稱簽署金鑰和非對稱簽署金鑰。您永遠不會直接與 CMM 互動。加密和解密方法會為您代勞。

由於 CMM 充當 AWS 資料庫加密 SDK 與 keyring 之間的聯絡點，因此它是自訂和延伸的理想點，例如支援政策強制執行。您可以明確指定 CMM，但並非必要。當您指定 keyring 時， AWS 資料庫加密

SDK 會為您建立預設 CMM。預設 CMM 會從您指定的 keyring 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)。

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。

非對稱加密使用數學上相關的資料金鑰對。對中的一個金鑰會加密資料；只有對中的另一個金鑰可以解密資料。

AWS Database Encryption SDK 使用信封加密。它使用對稱資料金鑰加密您的資料。它會使用一或多個對稱或非對稱包裝金鑰來加密對稱資料金鑰。它會將資料描述新增至記錄，其中包含至少一個資料金鑰的加密複本。

加密您的資料（對稱加密）

為了加密您的資料， AWS 資料庫加密 SDK 使用對稱資料金鑰和包含對稱加密演算法的演算法套件。若要解密資料， AWS 資料庫加密 SDK 會使用相同的資料金鑰和相同的演算法套件。

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操作的 keyring 會決定對稱資料金鑰的加密和解密方式。您可以選擇使用對稱加密的 keyring，例如具有對稱加密 KMS 金鑰的 AWS KMS keyring，或使用非對稱加密的 keyring，例如 AWS KMS 具有非對稱 RSA KMS 金鑰的 keyring。

金鑰承諾

AWS Database Encryption SDK 支援金鑰承諾（有時稱為穩健性)，這是一種安全屬性，可確保每個加密文字只能解密為單一純文字。若要這樣做，金鑰承諾可確保只會使用加密您記錄的資料金鑰來解密它。 AWS Database Encryption SDK 包含所有加密和解密操作的金鑰承諾。

大多數現代對稱密碼（包括 AES) 會在單一私密金鑰下加密純文字，例如 AWS 資料庫加密 SDK 用來加密`ENCRYPT_AND_SIGN`記錄中標記的每個純文字欄位的唯一資料金鑰。使用相同的資料金鑰解密此記錄會傳回與原始相同的純文字。使用不同金鑰進行解密通常會失敗。雖然很困難，但在技術上可以在兩個不同的金鑰下解密加密文字。在極少數情況下，您可以找到能夠部分解密加密文字為不同但仍然無法理解的純文字的金鑰。

AWS Database Encryption SDK 一律會在一個唯一的資料金鑰下加密每個屬性。它可能會在多個包裝金鑰下加密該資料金鑰，但包裝金鑰一律會加密相同的資料金鑰。不過，複雜的手動製作加密記錄實際

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，如果一個使用者解密加密的記錄，則傳回 0x0 (false)，而另一個解密相同加密記錄的使用者則得到 0x1 (true)。

為了防止這種情況， AWS 資料庫加密 SDK 會在加密和解密時包含金鑰承諾。加密方法以密碼編譯方式將產生加密文字的唯一資料金鑰繫結至金鑰承諾，這是使用資料金鑰衍生，根據資料描述計算的雜湊型訊息驗證碼 (HMAC)。然後，它會將金鑰承諾存放在材料描述中。當它使用金鑰承諾解密記錄時， AWS 資料庫加密 SDK 會驗證資料金鑰是該加密記錄的唯一金鑰。如果資料金鑰驗證失敗，解密操作會失敗。

數位簽章

AWS Database Encryption SDK 會使用已驗證的加密演算法 AES-GCM 加密您的資料，而解密程序會驗證加密訊息的完整性和真實性，而無需使用數位簽章。但由於 AES-GCM 使用對稱金鑰，任何可以解密用於解密加密文字的資料金鑰的人，也可以手動建立新的加密加密加密文字，造成潛在的安全問題。例如，如果您使用 AWS KMS key 做為包裝金鑰，具有 `kms:Decrypt` 許可的使用者可以建立加密的密碼文字，而無需呼叫 `kms:Encrypt`。

為了避免此問題，預設演算法套件會將橢圓曲線數位簽章演算法 (ECDSA) 簽章新增至加密的記錄。預設演算法套件`ENCRYPT_AND_SIGN`會使用已驗證的加密演算法 AES-GCM 來加密記錄中標示的欄位。然後，它會在記錄標示為 `ENCRYPT_AND_SIGN`、 和 的欄位上，計算雜湊型訊息驗證碼 (HMACs) `SIGN_ONLY`和非對稱 ECDSA 簽章`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。解密程序會使用簽章來驗證授權使用者是否已加密記錄。

使用預設演算法套件時， AWS 資料庫加密 SDK 會為每個加密的記錄產生臨時私有金鑰和公有金鑰對。 AWS Database Encryption SDK 會將公有金鑰存放在材料描述中，並捨棄私有金鑰。這可確保沒有人可以建立另一個使用公有金鑰驗證的簽章。演算法會將公有金鑰繫結至加密的資料金鑰，作為材料描述中的其他已驗證資料，防止只能解密欄位的使用者更改公有金鑰或影響簽章驗證。

AWS 資料庫加密 SDK 一律包含 HMAC 驗證。ECDSA 數位簽章預設為啟用，但並非必要。如果加密資料的使用者和解密資料的使用者同樣受信任，您可以考慮使用不包含數位簽章的演算法套件來改善效能。如需選取替代演算法套件的詳細資訊，請參閱選擇演算法套件。

> **Note**
>
> 如果 keyring 沒有在加密程式和解密程式之間劃定，數位簽章不會提供密碼編譯值。

AWS KMS keyring，包括非對稱 RSA AWS KMS keyring，可以根據 AWS KMS 金鑰政策和 IAM 政策，在加密程式和解密程式之間劃定。

由於其密碼編譯性質，下列 keyring 無法在加密程式和解密程式之間描述：

- AWS KMS 階層式 keyring
- AWS KMS ECDH keyring
- 原始 AES keyring
- 原始 RSA keyring
- 原始 ECDH keyring

AWS 資料庫加密 SDK 的運作方式

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

AWS Database Encryption SDK 提供用戶端加密程式庫，專為保護您存放在資料庫中的資料而設計。程式庫包括您可以擴展或以原狀使用的安全實作。如需定義和使用自訂元件的詳細資訊，請參閱資料庫實作的 GitHub 儲存庫。

本節中的工作流程說明 AWS Database Encryption SDK 如何加密和簽署和解密，並驗證資料庫中的資料。這些工作流程使用抽象元素和預設功能來描述基本程序。如需有關 AWS 資料庫加密 SDK 如何搭配資料庫實作運作的詳細資訊，請參閱資料庫的「什麼已加密」主題。

AWS Database Encryption SDK 使用信封加密來保護您的資料。每個記錄都會以唯一的資料金鑰加密。資料金鑰用於衍生密碼編譯動作`ENCRYPT_AND_SIGN`中標記的每個欄位的唯一資料加密金鑰。然後，您指定的包裝金鑰會加密資料金鑰的副本。若要解密加密的記錄， AWS 資料庫加密 SDK 會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，它可以解密加密文字並傳回純文字項目。

如需 AWS 資料庫加密 SDK 中使用的術語的詳細資訊，請參閱 AWS 資料庫加密 SDK 概念。

加密和簽署

AWS 資料庫加密 SDK 的核心是記錄加密程式，可加密、簽署、驗證和解密資料庫中的記錄。它會取得有關記錄的資訊，以及要加密和簽署哪些欄位的指示。它會從您指定的包裝金鑰中設定的密碼編譯資料管理員取得加密資料，以及如何使用這些資料的指示。

下列逐步解說說明 AWS Database Encryption SDK 如何加密和簽署您的資料項目。

1. 密碼編譯資料管理員為 AWS 資料庫加密 SDK 提供唯一的資料加密金鑰：一個純文字資料金鑰、由指定包裝金鑰加密的資料金鑰複本，以及 MAC 金鑰。

> **Note**
>
> 您可以在多個包裝金鑰下加密資料金鑰。每個包裝金鑰都會加密資料金鑰的個別複本。AWS Database Encryption SDK 會將所有加密的資料金鑰存放在材料描述中。 AWS Database Encryption SDK 會將新欄位 (`aws_dbe_head`) 新增至存放材料描述的記錄。系統會針對每個資料金鑰的加密複本衍生 MAC 金鑰。MAC 金鑰不會存放在材料描述中。反之，解密方法會使用包裝金鑰再次衍生 MAC 金鑰。

2. 加密方法會加密您指定的密碼編譯動作`ENCRYPT_AND_SIGN`中標記為 的每個欄位。
3. 加密方法`commitKey`從資料金鑰衍生 ，並使用它來產生金鑰承諾值，然後捨棄資料金鑰。
4. 加密方法會將物料描述新增至記錄。材料描述包含加密的資料金鑰，以及有關加密記錄的其他資訊。如需材料描述中包含的資訊完整清單，請參閱材料描述格式。
5. 加密方法使用步驟 1 中傳回的 MAC 金鑰，透過材料描述、加密內容和密碼編譯動作`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`中標記為 `ENCRYPT_AND_SIGN`、`SIGN_ONLY`或 的每個欄位的正式化來計算雜湊型訊息驗證碼 (HMAC) 值。HMAC 值會儲存在加密方法新增至記錄的新欄位 (`aws_dbe_foot`) 中。
6. 加密方法計算 ECDSA 簽章，透過材料描述、加密內容和每個標記為 `ENCRYPT_AND_SIGN`、`SIGN_ONLY`或 欄位的正式化，`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`並將 ECDSA 簽章存放在 `aws_dbe_foot` 欄位中。

> **Note**
>
> ECDSA 簽章預設為啟用，但並非必要。

7. 加密方法會將加密和簽章的記錄存放在您的資料庫中

解密和驗證

1. 密碼編譯資料管理員 (CMM) 提供解密方法，其中包含儲存在資料描述中的解密資料，包括純文字資料金鑰和相關聯的 MAC 金鑰。
   - CMM 會使用指定 keyring 中的包裝金鑰解密加密的資料金鑰，並傳回純文字資料金鑰。
2. 解密方法會比較並驗證材料描述中的鍵承諾值。

3. 解密方法會驗證簽章欄位中的簽章。

   它會SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT從您定義的允許未驗證欄位清單中識別哪些欄位標記為 ENCRYPT_AND_SIGNSIGN_ONLY、 或 。解密方法使用步驟 1 中傳回的 MAC 金鑰來重新計算和比較標示為 ENCRYPT_AND_SIGN、 SIGN_ONLY或 之欄位的 HMAC 值SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。然後，它會使用存放在加密內容中的公有金鑰來驗證 ECDSA 簽章。

4. 解密方法使用純文字資料金鑰來解密每個標示為 的值ENCRYPT_AND_SIGN。然後 AWS ，資料庫加密 SDK 會捨棄純文字資料金鑰。

5. 解密方法會傳回純文字記錄。

AWS Database Encryption SDK 中支援的演算法套件

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

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。

AWS Database Encryption SDK 使用演算法套件來加密和簽署資料庫中的欄位。所有支援的演算法套件都使用進階加密標準 (AES) 演算法搭配稱為 AES-GCM 的 Galois/計數器模式 (GCM) 來加密原始資料。 AWS Database Encryption SDK 支援 256 位元加密金鑰。驗證標籤的長度一律是 16 個位元組。

AWS 資料庫加密 SDK 演算法套件

演算法	加密演算法	資料金鑰長度 (以位元為單位)	金鑰衍生演算法	對稱簽章演算法	非對稱簽章演算法	金鑰承諾
預設	AES-GCM	256	HKDF 搭配 SHA-512	HMAC-SHA-384	ECDSA，P-384 和 SHA-384 式	HKDF 搭配 SHA-512
不含 ECDSA 數	AES-GCM	256	HKDF 搭配 SHA-512	HMAC-SHA-384	無	HKDF 搭配 SHA-512

演算法	加密演算法	資料金鑰長度 (以位元為單位)	金鑰衍生演算法	對稱簽章演算法	非對稱簽章演算法	金鑰承諾
位簽章的 AES-GCM						

加密演算法

使用的加密演算法名稱和模式。 AWS 資料庫加密 SDK 中的演算法套件使用進階加密標準 (AES) 演算法搭配 Galois/計數器模式 (GCM)。

資料金鑰長度

位元的資料金鑰長度。 AWS Database Encryption SDK 支援 256 位元資料金鑰。資料金鑰會用作 HMAC extract-and-expand金鑰衍生函數 (HKDF) 的輸入。HKDF 的輸出做為加密演算法中的資料加密金鑰。

金鑰衍生演算法

用於衍生資料加密金鑰的 HMAC 式擷取和擴展金鑰衍生函數 (HKDF)。 AWS Database Encryption SDK 使用 RFC 5869 中定義的 HKDF。

- 使用的雜湊函數是 SHA-512
- 對於擷取步驟：
  - 不使用 salt。根據 RFC， salt 會設定為字串零。
  - 輸入鍵控資料是 keyring 的資料鍵控。
- 對於擴展步驟：
  - 輸入虛擬亂數金鑰是來自擷取步驟的輸出。
  - 金鑰標籤是以大端位元組順序表示`DERIVEKEY`字串的 UTF-8-encoded位元組。
  - 輸入資訊是演算法 ID 和金鑰標籤 （依該順序） 的串連。
  - 輸出鍵控材料的長度為資料鍵控長度。此輸出將當做加密演算法中的資料加密金鑰。

對稱簽章演算法

用於產生對稱簽章的雜湊型訊息驗證碼 (HMAC) 演算法。所有支援的演算法套件都包含 HMAC 驗證。

AWS Database Encryption SDK 會序列化材料描述和所有標示為 `ENCRYPT_AND_SIGN`、`SIGN_ONLY`或 的欄位`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。然後，它會使用 HMAC 搭配密碼編譯雜湊函數演算法 (SHA-384) 來簽署正式化。

對稱 HMAC 簽章會儲存在 AWS 資料庫加密 SDK 新增至記錄的新欄位 (`aws_dbe_foot`) 中。

非對稱簽章演算法

用來產生非對稱數位簽章的簽章演算法。

AWS Database Encryption SDK 會序列化材料描述和所有標示為 `ENCRYPT_AND_SIGN`、`SIGN_ONLY`或 的欄位`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。然後，它會使用橢圓曲線數位簽章演算法 (ECDSA) 搭配下列詳細資訊來簽署正式化：

- 使用的橢圓曲線是 P-384，如數位簽章標準 (DSS) (FIPS PUB 186-4) 中所定義。
- 使用的雜湊函數是 SHA-384。

非對稱 ECDSA 簽章會與 `aws_dbe_foot` 欄位中的對稱 HMAC 簽章一起存放。

預設包含 ECDSA 數位簽章，但並非必要。

金鑰承諾

用於衍生遞交金鑰的 HMAC extract-and-expand金鑰衍生函數 (HKDF)。

- 使用的雜湊函數是 SHA-512
- 對於擷取步驟：
  - 不使用 salt。根據 RFC， salt 會設定為字串零。
  - 輸入鍵控材料是 keyring 的資料鍵控。
- 對於擴展步驟：
  - 輸入虛擬亂數金鑰是來自擷取步驟的輸出。
  - 輸入資訊是以大端位元組順序排列的`COMMITKEY`字串 UTF-8-encoded位元組。
  - 輸出鍵控材料的長度為 256 位元。此輸出用作遞交金鑰。

遞交金鑰會計算記錄承諾，即資料描述上不同的 256 位元雜湊型訊息驗證碼 (HMAC) 雜湊。如需將金鑰承諾新增至演算法套件的技術說明，請參閱 Cryptology ePrint Archive 中的金鑰承諾 AEADs。

預設演算法套件

根據預設， AWS 資料庫加密 SDK 會使用演算法套件搭配 AES-GCM、HMAC extract-and-expand金鑰衍生函數 (HKDF)、HMAC 驗證、ECDSA 數位簽章、金鑰承諾和 256 位元加密金鑰。

預設演算法套件包含 HMAC 驗證（對稱簽章）和 ECDSA 數位簽章（非對稱簽章）。這些簽章會存放在 AWS 資料庫加密 SDK 新增至記錄的新欄位 (`aws_dbe_foot`) 中。當授權政策允許一組使用者加密資料和一組不同的使用者解密資料時，ECDSA 數位簽章特別有用。

預設演算法套件也會衍生金鑰承諾 – 將資料金鑰與記錄連結的 HMAC 雜湊。金鑰承諾值是從材料描述和遞交金鑰計算的 HMAC。然後，金鑰承諾值會存放在材料描述中。金鑰承諾可確保每個加密文字只解密為一個純文字。他們透過驗證用作加密演算法輸入的資料金鑰來執行此操作。加密時，演算法套件會衍生金鑰承諾 HMAC。在解密之前，他們會驗證資料金鑰是否產生相同的金鑰承諾 HMAC。如果沒有，解密呼叫會失敗。

不含 ECDSA 數位簽章的 AES-GCM

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，某些信任模型將由沒有 ECDSA 數位簽章的演算法套件所滿足。只有在加密資料的使用者和解密資料的使用者同樣受信任時，才使用此套件。

所有 AWS 資料庫加密 SDK 演算法套件都包含 HMAC 驗證（對稱簽章）。唯一的差別是，沒有 ECDSA 數位簽章的 AES-GCM 演算法套件缺少非對稱簽章，可提供多一層的真實性和非否認性。

例如，如果您的 keyring、、 `wrappingKeyA``wrappingKeyB`和 中有多個包裝金鑰`wrappingKeyC`，而且您使用 解密記錄`wrappingKeyA`，HMAC 對稱簽章會驗證記錄是否由具有 存取權的使用者加密`wrappingKeyA`。如果您使用預設演算法套件，HMACs 會提供與 相同的驗證`wrappingKeyA`，並另外使用 ECDSA 數位簽章，以確保記錄是由具有 加密許可的使用者加密`wrappingKeyA`。

若要選取不含數位簽章的 AES-GCM 演算法套件，請在加密組態中包含下列程式碼片段。

Java

以下程式碼片段指定不含 ECDSA 數位簽章的 AES-GCM 演算法套件。如需詳細資訊，請參閱the section called “加密組態”。

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

C# / .NET

以下程式碼片段指定不含 ECDSA 數位簽章的 AES-GCM 演算法套件。如需詳細資訊，請參閱the section called “加密組態”。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

以下程式碼片段指定不含 ECDSA 數位簽章的 AES-GCM 演算法套件。如需詳細資訊，請參閱the section called “加密組態”。

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

搭配 AWS 資料庫加密 SDK 使用 AWS KMS

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

若要使用 AWS 資料庫加密 SDK，您需要設定 keyring 並指定一或多個包裝金鑰。如果您沒有金鑰基礎架構，建議您使用 AWS Key Management Service (AWS KMS)。

AWS Database Encryption SDK 支援兩種類型的 AWS KMS keyring。傳統 AWS KMS keyring 使用 AWS KMS keys 來產生、加密和解密資料金鑰。您可以使用對稱加密 (`SYMMETRIC_DEFAULT`) 或非對稱 RSA KMS 金鑰。由於 AWS Database Encryption SDK 會使用唯一的資料金鑰來加密和簽署每個記錄， AWS KMS 因此 keyring 必須 AWS KMS 針對每個加密和解密操作呼叫 。對於需要將呼叫次數降至最低的應用程式 AWS KMS， AWS 資料庫加密 SDK 也支援AWS KMS 階層式 keyring。階層式 keyring 是一種密碼編譯材料快取解決方案，透過使用保留在 Amazon DynamoDB 資料表中的 AWS KMS 受保護分支金鑰，以及加密和解密操作中使用的本機快取分支金鑰材料來減少 AWS KMS 呼叫次數。建議您盡可能使用 AWS KMS keyring。

若要與 互動 AWS KMS， AWS 資料庫加密 SDK 需要 的 AWS KMS 模組 適用於 Java 的 AWS SDK。

準備搭配 使用 AWS Database Encryption SDK AWS KMS

1. 建立 AWS 帳戶。若要了解如何，請參閱 AWS 知識中心中的如何建立和啟用新的 Amazon Web Services 帳戶？。
2. 建立對稱加密 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 AWS KMS key 程式設計方式使用 ，您需要 的 Amazon Resource Name (ARN) AWS KMS key。如需尋找 之 ARN 的協助 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 ID 和安全存取金鑰。您可以使用 IAM 使用者的存取金鑰 ID 和私密存取金鑰，也可以使用 AWS Security Token Service 來建立新的工作階段，其中包含存取金鑰 ID、私密存取金鑰

和工作階段權杖的臨時安全登入資料。作為安全最佳實務，我們建議您使用臨時憑證，而不是與您的 IAM 使用者或 AWS （根） 使用者帳戶相關聯的長期憑證。

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》中的請求臨時安全登入資料。

4. 使用 中的指示適用於 Java 的 AWS SDK，以及您在步驟 3 中產生的存取金鑰 ID 和私密存取金鑰來設定 AWS 您的登入資料。如果您產生臨時登入資料，您也需要指定工作階段權杖。

   此程序允許 AWS SDKs AWS 為您簽署 請求。與 互動的 AWS 資料庫加密 SDK 中的程式碼範例 AWS KMS 假設您已完成此步驟。

設定 AWS 資料庫加密 SDK

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍提供有關 DynamoDB 加密用戶端的資訊。

AWS Database Encryption SDK 的設計易於使用。雖然 AWS 資料庫加密 SDK 有數個組態選項，但系統會謹慎選擇預設值，以對大多數應用程式實用且安全。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。

主題

- 選取程式設計語言
- 選取包裝金鑰
- 建立探索篩選條件
- 使用多租戶資料庫
- 建立簽章的信標

選取程式設計語言

適用於 DynamoDB 的 AWS Database Encryption SDK 提供多種程式設計語言。語言實作設計為可完全互通，並提供相同的功能，但可能以不同的方式實作。一般而言，您會使用與您的應用程式相容的程式庫。

選取包裝金鑰

AWS Database Encryption SDK 會產生唯一的對稱資料金鑰來加密每個欄位。您不需要設定、管理或使用資料金鑰。 AWS Database Encryption SDK 會為您執行。

不過，您必須選取一或多個包裝金鑰來加密每個資料金鑰。 AWS Database Encryption SDK 支援 AWS Key Management Service(AWS KMS) 對稱加密 KMS 金鑰和非對稱 RSA KMS 金鑰。它還支援不同大小的 AES 對稱金鑰和 RSA 非對稱金鑰。您有責任保護包裝金鑰的安全性和耐久性，因此我們建議您在硬體安全模組或金鑰基礎設施服務中使用加密金鑰，例如 AWS KMS。

若要指定用於加密和解密的包裝金鑰，請使用 keyring。根據您使用的 keyring 類型，您可以指定一個包裝金鑰或相同或不同類型的多個包裝金鑰。如果您使用多個包裝金鑰來包裝資料金鑰，則每個包裝金

鑰都會加密相同資料金鑰的副本。加密的資料金鑰（每個包裝金鑰一個）會存放在與加密欄位一起存放的資料描述中。若要解密資料，AWS 資料庫加密 SDK 必須先使用其中一個包裝金鑰來解密加密的資料金鑰。

我們建議您盡可能使用其中一個 AWS KMS keyring。 AWS Database Encryption SDK 提供 AWS KMS keyring 和AWS KMS 階層 keyring，可減少對 進行的呼叫數量 AWS KMS。若要在 keyring AWS KMS key 中指定 ，請使用支援的 AWS KMS 金鑰識別符。如果您使用 AWS KMS 階層式 keyring，則必須指定金鑰 ARN。如需金鑰識別符的詳細資訊 AWS KMS ，請參閱《 AWS Key Management Service 開發人員指南》中的金鑰識別符。

- 當您使用 AWS KMS keyring 加密時，您可以為對稱加密 KMS 金鑰指定任何有效的金鑰識別符（金鑰 ARN、別名名稱、別名 ARN 或金鑰 ID)。如果您使用非對稱 RSA KMS 金鑰，則必須指定金鑰 ARN。

  如果您在加密時為 KMS 金鑰指定別名名稱或別名 ARN， AWS 資料庫加密 SDK 會儲存目前與該別名相關聯的金鑰 ARN；不會儲存別名。別名的變更不會影響用來解密資料金鑰的 KMS 金鑰。
- 根據預設， AWS KMS keyring 會以嚴格模式（指定特定 KMS 金鑰的位置）解密記錄。您必須使用金鑰 ARN 來識別 AWS KMS keys 以進行解密。

  當您使用 AWS KMS keyring 加密時， AWS 資料庫加密 SDK 會將 的金鑰 ARN 存放在具有加密資料金鑰的資料描述 AWS KMS key 中。在嚴格模式下解密時， AWS 資料庫加密 SDK 會驗證相同的金鑰 ARN 是否出現在 keyring 中，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符，即使識別符參考相同的金鑰 AWS KMS key， AWS 資料庫加密 SDK 也不會識別或使用 。
- 在探索模式中解密時，您不會指定任何包裝金鑰。首先， AWS 資料庫加密 SDK 會嘗試使用存放在材料描述中的金鑰 ARN 解密記錄。如果無法運作， AWS 資料庫加密 SDK AWS KMS 會要求 使用加密記錄的 KMS 金鑰來解密記錄，無論誰擁有或可存取該 KMS 金鑰。

若要將原始 AES 金鑰或原始 RSA 金鑰對指定為 keyring 中的包裝金鑰，您必須指定命名空間和名稱。解密時，您必須為每個原始包裝金鑰使用與加密時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，即使金鑰材料相同， AWS 資料庫加密 SDK 也不會識別或使用包裝金鑰。

建立探索篩選條件

解密使用 KMS 金鑰加密的資料時，最佳實務是在嚴格模式下解密，也就是限制僅用於您指定的包裝金鑰。不過，如有必要，您也可以在探索模式中解密，其中不指定任何包裝金鑰。在此模式中， AWS KMS 可以使用加密它的 KMS 金鑰來解密加密的資料金鑰，無論誰擁有或有權存取該 KMS 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條件，這會限制可用於指定 AWS 帳戶 和 分割區中的 KMS 金鑰。探索篩選條件是選用的，但這是最佳實務。

使用下表來判斷探索篩選條件的分割區值。

區域	分區
AWS 區域	`aws`
中國區域	`aws-cn`
AWS GovCloud (US) Regions	`aws-us-gov`

下列範例示範如何建立探索篩選條件。使用程式碼之前，請將範例值取代為 AWS 帳戶 和 分割區的有效值。

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
```

C# / .NET

```
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;
```

使用多租戶資料庫

使用 AWS 資料庫加密 SDK，您可以使用不同的加密資料隔離每個租用戶，為具有共用結構描述的資料庫設定用戶端加密。考慮多租戶資料庫時，請花一些時間檢閱您的安全需求，以及多租戶如何影響這些需求。例如，使用多租戶資料庫可能會影響您結合 AWS 資料庫加密 SDK 與另一個伺服器端加密解決方案的能力。

如果您有多個使用者在資料庫中執行加密操作，您可以使用其中一個 AWS KMS keyring 為每位使用者提供可在其密碼編譯操作中使用的不同金鑰。管理多租戶用戶端加密解決方案的資料金鑰可能很複雜。我們建議您盡可能依租戶組織您的資料。如果租用戶是由主索引鍵值（例如 Amazon DynamoDB 資料表中的分割區索引鍵）識別，則管理索引鍵會更容易。

您可以使用 AWS KMS keyring，以不同的 AWS KMS keyring 和 隔離每個租用戶 AWS KMS keys。根據每個租用戶進行的呼叫量 AWS KMS ，您可能想要使用 AWS KMS 階層式 keyring 將呼叫降至最低 AWS KMS。AWS KMS 階層式 keyring 是一種密碼編譯資料快取解決方案，可透過使用保留在 Amazon DynamoDB 資料表中的 AWS KMS 受保護分支金鑰，以及加密和解密操作中使用的本機快取分支金鑰資料來減少 AWS KMS 呼叫次數。您必須使用 AWS KMS 階層式 keyring 在您的資料庫中實作可搜尋的加密。

建立簽章的信標

AWS Database Encryption SDK 使用標準信標和複合信標來提供可搜尋的加密解決方案，可讓您搜尋加密的記錄，而不會解密查詢的整個資料庫。不過， AWS 資料庫加密 SDK 也支援簽署的信標，這些信標可以完全從純文字簽署欄位設定。簽章信標是一種複合信標，可對 `SIGN_ONLY` 和 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 欄位編製索引並執行複雜的查詢。

例如，如果您有多租用戶資料庫，您可能想要建立簽章的信標，可讓您查詢資料庫是否有特定租用戶金鑰加密的記錄。如需詳細資訊，請參閱查詢多租戶資料庫中的信標。

您必須使用 AWS KMS 階層式 keyring 來建立簽章的信標。

若要設定已簽章的信標，請提供下列值。

Java

複合信標組態

下列範例會在已簽章的信標組態中於本機定義已簽章的組件清單。

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
```

```
    .name("compoundBeaconName")
    .split(".")
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

信標版本定義

下列範例會在信標版本中全域定義已簽章的組件清單。如需定義信標版本的詳細資訊，請參閱使用信標。

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

請參閱完整的程式碼範例：BeaconConfig.cs

簽章信標組態

下列範例會在已簽章的信標組態中於本機定義已簽章的組件清單。

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Signed = signedPartList,
```

```
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

信標版本定義

下列範例會在信標版本中全域定義已簽章的組件清單。如需定義信標版本的詳細資訊，請參閱使用信標。

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

您可以在本機或全域定義的清單中定義已簽署的組件。我們建議您盡可能在信標版本中的全域清單中定義已簽署的組件。透過全域定義已簽章的組件，您可以定義每個組件一次，然後在多個複合信標組態中重複使用這些組件。如果您只打算使用已簽章的部分一次，您可以在已簽章信標組態的本機清單中定義它。您可以在建構函數清單中參考本機和全域組件。

如果您全域定義已簽章的組件清單，則必須提供建構組件清單，以識別已簽章的信標可以組合信標組態中欄位的所有可能方式。

> **Note**
>
> 若要全域定義已簽章的組件清單，您必須使用 AWS 資料庫加密 SDK 的 3.2 版或更新版本。在全域定義任何新組件之前，將新版本部署到所有讀者。

您無法更新現有的信標組態，以全域定義已簽章的組件清單。

信標名稱

您在查詢信標時所使用的名稱。

簽章的信標名稱不能與未加密欄位的名稱相同。兩個信標不能有相同的信標名稱。

分割字元

用來分隔組成已簽章信標之部分的字元。

分割字元無法出現在已簽署信標建構來源之任何欄位的純文字值中。

已簽章的組件清單

識別已簽章信標中包含的已簽章欄位。

每個部分都必須包含名稱、來源和字首。來源是組件識別的 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 欄位。來源必須是參照巢狀欄位值的欄位名稱或索引。如果您的組件名稱識別來源，您可以省略來源， AWS 資料庫加密 SDK 會自動使用該名稱做為來源。我們建議您盡可能將來源指定為組件名稱。字首可以是任何字串，但必須是唯一的。已簽章的信標中沒有兩個已簽章的部分可以具有相同的字首。建議使用短值來區分部分與複合信標提供的其他部分。

我們建議您盡可能全域定義已簽署的組件。如果您只打算在一個複合信標中使用已簽署的部分，您可以考慮在本機定義。本機定義的部分不能具有與全域定義的部分相同的字首或名稱。

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
    SignedPart signedPartExample = SignedPart.builder()
        .name("signedFieldName")
        .prefix("S-")
        .build();
    signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
```

```
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

建構器清單（選用）

識別建構函式，這些建構函式定義了簽署的組件可由簽署的信標組合的不同方式。

如果您未指定建構函數清單， AWS 資料庫加密 SDK 會將簽章的信標與下列預設建構函數組合。

- 所有已簽章組件依其新增至已簽章組件清單的順序進行
- 需要所有組件

建構函式

每個建構函式都是建構函式組件的排序清單，可定義已簽章信標的組合方式。建構函數部分會依新增至清單的順序聯結在一起，每個部分以指定的分割字元分隔。

每個建構函數組件都會命名一個已簽章的組件，並定義該組件在建構函數中是必要還是選用。例如，如果您想要在 `Field1`、 `Field1.Field2`和 上查詢簽章的信標`Field1.Field2.Field3`，請將 `Field2`和 標記為`Field3`選用，並建立一個建構函數。

每個建構函數必須至少有一個必要的部分。我們建議您讓每個建構函數的第一部分成為必要項目，以便在查詢中使用`BEGINS_WITH`運算子。

如果記錄中存在所有必要的組件，建構函數就會成功。當您撰寫新記錄時，簽章的信標會使用建構函數清單來判斷信標是否可以從提供的值組合。它會嘗試依建構函數新增至建構函數清單的順序組合信標，並使用第一個成功的建構函數。如果沒有建構函數成功，則不會將信標寫入記錄。

所有讀者和寫入者都應指定相同的建構函數順序，以確保其查詢結果正確。

使用以下程序來指定您自己的建構函數清單。

1. 為每個已簽章的組件建立建構組件，以定義是否需要該組件。

   建構函數部分名稱必須是已簽章欄位的名稱。

   下列範例示範如何為一個已簽章欄位建立建構函數部分。

   Java

```
ConstructorPart field1ConstructorPart = ConstructorPart.builder()
        .name("Field1")
        .required(true)
```

```
        .build();
```

C# / .NET

```
var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
 = true };
```

2. 使用您在步驟 1 中建立的建構函式部分，為每種可能的方式建立已簽章的信標組合建構函式。

   例如，如果您想要在 `Field1.Field2.Field3`和 上查詢 `Field4.Field2.Field3`，則必須建立兩個建構函數。 `Field1`和 `Field4`可能需要兩者，因為它們是在兩個不同的建構函數中定義。

   Java

```
// Create a list for Field1.Field2.Field3 queries
List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
field123ConstructorPartList.add(field1ConstructorPart);
field123ConstructorPartList.add(field2ConstructorPart);
field123ConstructorPartList.add(field3ConstructorPart);
Constructor field123Constructor = Constructor.builder()
        .parts(field123ConstructorPartList)
        .build();
// Create a list for Field4.Field2.Field1 queries
List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
field421ConstructorPartList.add(field4ConstructorPart);
field421ConstructorPartList.add(field2ConstructorPart);
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

   C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
{
```

```
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

3. 建立建構函數清單，其中包含您在步驟 2 中建立的所有建構函數。

   Java

   ```
   List<Constructor> constructorList = new ArrayList<>();
   constructorList.add(field123Constructor)
   constructorList.add(field421Constructor)
   ```

   C# / .NET

   ```
   var constructorList = new List<Constructor>
   {
       field123Constructor,
       field421Constructor
   };
   ```

4. 當您建立簽章的信標`constructorList`時，請指定 。

AWS 資料庫加密 SDK 中的金鑰存放區

在 AWS 資料庫加密 SDK 中，金鑰存放區是 Amazon DynamoDB 資料表，可保留AWS KMS 階層 keyring 所使用的階層資料。金鑰存放區有助於減少使用階層式 keyring AWS KMS 執行密碼編譯操作所需的呼叫次數。

金鑰存放區會維護和管理階層式 keyring 用來執行信封加密和保護資料加密金鑰的分支金鑰。金鑰存放區會存放作用中的分支金鑰和所有舊版的分支金鑰。作用中分支金鑰是最新的分支金鑰版本。階層式 keyring 會針對每個加密請求使用唯一的資料加密金鑰，並使用衍生自作用中分支金鑰的唯一包裝金鑰來加密每個資料加密金鑰。階層式 keyring 取決於作用中分支金鑰與其衍生包裝金鑰之間建立的階層。

金鑰存放區術語和概念

Key store (金鑰存放區)

DynamoDB 資料表可保留階層式資料，例如分支金鑰和信標金鑰。

根金鑰

對稱加密 KMS 金鑰，可產生和保護金鑰存放區中的分支金鑰和信標金鑰。

分支金鑰

重複使用的資料金鑰，以衍生信封加密的唯一包裝金鑰。您可以在一個金鑰存放區中建立多個分支金鑰，但每個分支金鑰一次只能有一個作用中的分支金鑰版本。作用中分支金鑰是最新的分支金鑰版本。

分支金鑰衍生自 AWS KMS keys 使用 kms:GenerateDataKeyWithoutPlaintext 操作。

包裝金鑰

唯一資料金鑰，用於加密加密操作中使用的資料加密金鑰。

包裝金鑰衍生自分支金鑰。如需金鑰衍生程序的詳細資訊，請參閱AWS KMS 階層式 keyring 技術詳細資訊。

資料加密金鑰

用於加密操作的資料金鑰。階層式 keyring 會針對每個加密請求使用唯一的資料加密金鑰。

指標索引鍵

用來產生可搜尋加密信標的資料金鑰。如需詳細資訊，請參閱可搜尋加密。

實作最低權限的許可

使用金鑰存放區和 AWS KMS 階層式 keyring 時，建議您定義下列角色，以遵循最低權限原則：

金鑰存放區管理員

金鑰存放區管理員負責建立和管理金鑰存放區，以及其持續存在和保護的分支金鑰。金鑰存放區管理員應該是唯一具有 Amazon DynamoDB 資料表寫入許可的使用者，該資料表做為您的金鑰存放區。他們應該是唯一有權存取特殊權限管理員操作的使用者，例如 `CreateKey`和 `VersionKey`。您只能在靜態設定金鑰存放區動作時執行這些操作。

`CreateKey` 是一種特殊權限操作，可將新的 KMS 金鑰 ARN 新增至您的金鑰存放區允許清單。此 KMS 金鑰可以建立新的作用中分支金鑰。我們建議您限制對此操作的存取，因為一旦 KMS 金鑰新增至分支金鑰存放區，就無法刪除它。

金鑰存放區使用者

在大多數使用案例中，金鑰存放區使用者只會在加密、解密、簽署和驗證資料時，透過階層式 keyring 與金鑰存放區互動。因此，他們只需要做為您金鑰存放區的 Amazon DynamoDB 資料表的讀取許可。金鑰存放區使用者只需要存取使密碼編譯操作成為可能的使用操作，例如 `GetActiveBranchKey`、 `GetBranchKeyVersion`和 `GetBeaconKey`。他們不需要許可來建立或管理他們使用的分支金鑰。

您可以在靜態設定金鑰存放區動作，或設定探索時執行用量操作。當您的金鑰存放區動作設定為探索時，無法執行管理員操作 (`CreateKey` 和 `VersionKey`)。

如果您的分支金鑰存放區管理員允許列出分支金鑰存放區中的多個 KMS 金鑰，建議您的金鑰存放區使用者設定其金鑰存放區動作以進行探索，以便其階層式 keyring 可以使用多個 KMS 金鑰。

建立金鑰存放區

在建立分支金鑰或使用AWS KMS 階層式 keyring 之前，您必須建立金鑰存放區，這是管理和保護分支金鑰的 Amazon DynamoDB 資料表。

> **Important**
>
> 請勿刪除持續分支金鑰的 DynamoDB 資料表。如果您刪除此資料表，您將無法解密使用階層式 keyring 加密的任何資料。

請遵循 Amazon DynamoDB 開發人員指南中的建立資料表程序，使用下列分割區索引鍵和排序索引鍵所需的字串值。

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

邏輯金鑰存放區名稱

命名做為金鑰存放區的 DynamoDB 資料表時，請務必仔細考慮您在設定金鑰存放區動作時指定的邏輯金鑰存放區名稱。邏輯金鑰存放區名稱可做為您金鑰存放區的識別符，而且在第一個使用者最初定義之後，就無法變更。您必須一律在金鑰存放區動作中指定相同的邏輯金鑰存放區名稱。

DynamoDB 資料表名稱和邏輯金鑰存放區名稱之間必須有one-to-one的映射。邏輯金鑰存放區名稱以密碼編譯方式繫結至存放在資料表中的所有資料，以簡化 DynamoDB 還原操作。雖然邏輯金鑰存放區名稱可以與您的 DynamoDB 資料表名稱不同，但我們強烈建議將您的 DynamoDB 資料表名稱指定為邏輯金鑰存放區名稱。如果您的資料表名稱在從備份還原 DynamoDB 資料表之後變更，則邏輯金鑰存放區名稱可以映射到新的 DynamoDB 資料表名稱，以確保階層式 keyring 仍可存取您的金鑰存放區。

請勿在邏輯金鑰存放區名稱中包含機密或敏感資訊。邏輯金鑰存放區名稱會以純文字形式顯示在 AWS KMS CloudTrail 事件中，做為 `tablename`。

後續步驟

1. the section called “設定金鑰存放區動作”
2. the section called “建立分支金鑰”
3. 建立 AWS KMS 階層式 keyring

設定金鑰存放區動作

金鑰存放區動作會決定使用者可執行的操作，以及其 AWS KMS 階層式 keyring 如何使用金鑰存放區中允許列出的 KMS 金鑰。 AWS Database Encryption SDK 支援下列金鑰存放區動作組態。

靜態

當您靜態設定金鑰存放區時，金鑰存放區只能使用與您在設定金鑰存放區動作`kmsConfiguration`時在 中提供的 KMS 金鑰 ARN 相關聯的 KMS 金鑰。如果在建立、版本控制或取得分支金鑰時遇到不同的 KMS 金鑰 ARN，則會擲回例外狀況。

您可以在 中指定多區域 KMS 金鑰`kmsConfiguration`，但金鑰的整個 ARN，包括區域，會保留在衍生自 KMS 金鑰的分支金鑰中。您無法在不同的區域中指定金鑰，您必須提供完全相同的多區域金鑰，值才能相符。

當您靜態設定金鑰存放區動作時，您可以執行用量操作 (`GetActiveBranchKey`、`GetBranchKeyVersion`、`GetBeaconKey`) 和管理操作 (`CreateKey` 和 `VersionKey`)。 `CreateKey` 是一種特殊權限操作，可將新的 KMS 金鑰 ARN 新增至您的金鑰存放區允許清單。此 KMS 金鑰可以建立新的作用中分支金鑰。我們建議您限制對此操作的存取，因為一旦 KMS 金鑰新增至金鑰存放區，就無法刪除。

探索

當您為探索設定金鑰存放區動作時，金鑰存放區可以使用金鑰存放區中允許列出的任何 AWS KMS key ARN。不過，當遇到多區域 KMS 金鑰，且金鑰 ARN 中的區域與正在使用的 AWS KMS 用戶端區域不相符時，就會擲出例外狀況。

當您設定金鑰存放區進行探索時，無法執行管理操作，例如 `CreateKey`和 `VersionKey`。您只能執行啟用加密、解密、簽署和驗證操作的用量操作。如需詳細資訊，請參閱the section called “實作最低權限的許可”。

設定您的金鑰存放區動作

設定金鑰存放區動作之前，請確定符合下列先決條件。

- 決定您需要執行的操作。如需詳細資訊，請參閱the section called “實作最低權限的許可”。
- 選擇邏輯金鑰存放區名稱

  DynamoDB 資料表名稱和邏輯金鑰存放區名稱之間必須有one-to-one的映射。邏輯金鑰存放區名稱以密碼編譯方式繫結至資料表中存放的所有資料，以簡化 DynamoDB 還原操作，在第一個使用者最初定義後就無法變更。您必須一律在金鑰存放區動作中指定相同的邏輯金鑰存放區名稱。如需詳細資訊，請參閱logical key store name。

靜態組態

下列範例靜態設定金鑰存放區動作。您必須指定做為金鑰存放區的 DynamoDB 資料表名稱、金鑰存放區的邏輯名稱，以及識別對稱加密 KMS 金鑰的 KMS 金鑰 ARN。

> **Note**
>
> 在靜態設定金鑰存放區服務時，請仔細考慮您指定的 KMS 金鑰 ARN。`CreateKey` 操作會將 KMS 金鑰 ARN 新增至分支金鑰存放區允許清單。將 KMS 金鑰新增至分支金鑰存放區後，就無法刪除。

Java

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

C# / .NET

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

Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
```

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

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

探索組態

下列範例會設定探索的金鑰存放區動作。您必須指定做為金鑰存放區的 DynamoDB 資料表名稱，以及邏輯金鑰存放區名稱。

Java

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

C# / .NET

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

Rust

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

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

建立作用中分支金鑰

分支金鑰是從 AWS KMS 衍生的資料金鑰 AWS KMS key ，階層式 keyring 會使用它來減少對 進行的呼叫數量 AWS KMS。作用中分支金鑰是最新的分支金鑰版本。階層式 keyring 會為每個加密請求產生唯一的資料金鑰，並使用衍生自作用中分支金鑰的唯一包裝金鑰來加密每個資料金鑰。

若要建立新的作用中分支金鑰，您必須靜態設定金鑰存放區動作。 `CreateKey` 是一種特殊權限操作，可將金鑰存放區動作組態中指定的 KMS 金鑰 ARN 新增至金鑰存放區允許清單。然後，KMS 金鑰會用來產生新的作用中分支金鑰。我們建議您限制對此操作的存取，因為一旦 KMS 金鑰新增至金鑰存放區，就無法刪除它。

我們建議您透過應用程式控制平面中的 KeyStore Admin 界面使用 `CreateKey`操作。此方法符合金鑰管理的最佳實務。

請勿在資料平面中建立分支金鑰。此實務可能會導致：

- 對 進行不必要的呼叫 AWS KMS
- 在高並行環境中對 AWS KMS 的多個並行呼叫
- 對後端 DynamoDB 資料表的多個 TransactWriteItems 呼叫。

`CreateKey` 操作會在`TransactWriteItems`呼叫中包含條件檢查，以防止覆寫現有的分支金鑰。不過，在資料平面中建立金鑰仍可能導致資源使用效率低下和潛在的效能問題。

您可以在金鑰存放區中允許列出一個 KMS 金鑰，也可以更新您在金鑰存放區動作組態中指定的 KMS 金鑰 ARN 並`CreateKey`再次呼叫，以允許列出多個 KMS 金鑰。如果您允許列出多個 KMS 金鑰，您的金鑰存放區使用者應該為探索設定其金鑰存放區動作，以便他們可以使用他們有權存取的金鑰存放區中任何允許列出的金鑰。如需詳細資訊，請參閱the section called “設定金鑰存放區動作”。

所需的 許可

若要建立分支金鑰，您需要kms:GenerateDataKeyWithoutPlaintext 和 kms:ReEncrypt 許可。

建立分支金鑰

下列操作會使用您在金鑰存放區動作組態中指定的 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做為金鑰存放區的 DynamoDB 資料表。

呼叫 時`CreateKey`，您可以選擇指定下列選用值。

- `branchKeyIdentifier`：定義自訂 `branch-key-id`。

  若要建立自訂 `branch-key-id`，您還必須在 `encryptionContext` 參數中包含其他加密內容。
- `encryptionContext`：定義一組選用的非秘密金鑰/值對，在 kms:GenerateDataKeyWithoutPlaintext 呼叫中包含的加密內容中提供額外的已驗證資料 (AAD)。

  此額外的加密內容會以 `aws-crypto-ec:`字首顯示。

Java

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

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

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

C# / .NET

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

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

```
        EncryptionContext = additionalEncryptionContext // OPTIONAL
    });
```

Rust

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

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

首先，`CreateKey`操作會產生下列值。

- 第 4 版的通用唯一識別碼 (UUID) `branch-key-id`（除非您指定了自訂 `branch-key-id`)。
- 分支金鑰版本的第 4 版 UUID
- ISO 8601 日期和時間格式`timestamp`的 ，以國際標準時間 (UTC) 為單位。

然後，`CreateKey`操作會使用下列請求呼叫 kms:GenerateDataKeyWithoutPlaintext。

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

> **Note**
>
> `CreateKey` 操作會建立作用中分支金鑰和信標金鑰，即使您尚未將資料庫設定為可搜尋加密。這兩個金鑰都存放在您的金鑰存放區中。如需詳細資訊，請參閱使用階層式 keyring 進行可搜尋加密。

接著，`CreateKey`操作會呼叫 kms:ReEncrypt，透過更新加密內容來建立分支金鑰的作用中記錄。

最後，`CreateKey`操作會呼叫 ddb：TransactWriteItems 來寫入新項目，該項目將保留您在步驟 2 中建立的資料表中的分支金鑰。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"
 }
```

輪換作用中的分支金鑰

每個分支金鑰一次只能有一個作用中版本。一般而言，每個作用中分支金鑰版本都會用來滿足多個請求。但是，您可以控制重複使用作用中分支金鑰的程度，並判斷作用中分支金鑰的輪換頻率。

分支金鑰不會用來加密純文字資料金鑰。它們用於衍生加密純文字資料金鑰的唯一包裝金鑰。包裝金鑰衍生程序會產生唯一的 32 位元組包裝金鑰，具有 28 個位元組的隨機性。這表示在密碼編譯耗用發生之前，分支金鑰可以衍生超過 79 個八進制或 2^{96} 個唯一的包裝金鑰。雖然耗盡風險非常低，但由於業務或合約規則或政府法規，您可能需要輪換作用中的分支金鑰。

分支金鑰的作用中版本會保持作用中狀態，直到您將其輪換為止。舊版的作用中分支金鑰不會用於執行加密操作，也無法用於衍生新的包裝金鑰，但仍然可以查詢它們並提供包裝金鑰來解密它們在作用中加密的資料金鑰。

> **Warning**
>
> 在測試環境中刪除分支索引鍵是無法復原的。您無法復原已刪除的分支金鑰。當您在測試環境中刪除並重新建立具有相同 ID 的分支金鑰時，可能會發生下列問題：
>
> - 先前測試執行的資料可能會保留在快取中
> - 有些測試主機或執行緒可能會使用已刪除的分支金鑰來加密資料
> - 使用已刪除分支加密的資料無法解密
>
> 若要防止整合測試中的加密失敗：
>
> - 在建立新的分支金鑰之前，請重設階層 keyring 參考或
> - 為每個測試使用唯一的分支金鑰 IDs

所需的許可

若要輪換分支金鑰，您需要kms:GenerateDataKeyWithoutPlaintext 和 kms:ReEncrypt 許可。

輪換作用中的分支金鑰

使用 `VersionKey`操作來輪換作用中的分支金鑰。當您輪換作用中分支金鑰時，會建立新的分支金鑰以取代先前的版本。當您輪換作用中分支金鑰時， `branch-key-id` 不會變更。當您呼叫 時`branch-key-id`，必須指定可識別目前作用中分支金鑰的 `VersionKey`。

Java

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

C# / .NET

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

Rust

```
keystore.version_key()
```

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

Keyrings

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

AWS Database Encryption SDK 使用 keyring 來執行信封加密。Keyring 會產生、加密及解密資料金鑰。Keyrings 會決定保護每個加密記錄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 keyring 來保護您的包裝金鑰，並在安全界限內執行密碼編譯操作，例如 AWS KMS keyring，其使用永遠 AWS KMS keys 不會讓 AWS Key Management Service(AWS KMS) 未加密。您也可以編寫使用封裝金鑰的 keyring，這些金鑰存放在您的硬體安全模組 (HSMs) 中或受到其他主金鑰服務保護。

您的 keyring 會決定保護資料金鑰的包裝金鑰，最終也決定您的資料。使用對您的任務最安全的包裝金鑰。盡可能使用受到硬體安全模組 (HSM) 或金鑰管理基礎設施保護的包裝金鑰，例如 AWS Key Management Service(AWS KMS) 中的 KMS 金鑰或 中的加密金鑰AWS CloudHSM。

AWS Database Encryption SDK 提供數個 keyring 和 keyring 組態，您可以建立自己的自訂 keyring。您也可以建立包含相同或不同類型之一或多個 keyring 的多 keyring。

主題

- keyring 如何運作
- AWS KMS 鑰匙圈
- AWS KMS 階層式 keyring
- AWS KMS ECDH 鑰匙圈
- 原始 AES keyring
- 原始 RSA keyring
- 原始 ECDH 鑰匙圈
- 多重 keyring

keyring 如何運作

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

當您在資料庫中加密和簽署欄位時， AWS 資料庫加密 SDK 會要求 keyring 提供加密資料。Keyring 會傳回純文字資料金鑰、由 keyring 中每個包裝金鑰加密的資料金鑰複本，以及與資料金鑰相關聯的 MAC 金鑰。 AWS Database Encryption SDK 使用純文字金鑰來加密資料，然後盡快從記憶體中移除純文字資料金鑰。然後， AWS 資料庫加密 SDK 新增了包含加密資料金鑰和其他資訊的材料描述，例如加密和簽署指示。 AWS Database Encryption SDK 使用 MAC 金鑰，透過材料描述的正式化和所有標記 `ENCRYPT_AND_SIGN`或 的欄位來計算雜湊型訊息驗證碼 (HMACs)`SIGN_ONLY`。

當您解密資料時，您可以使用您用來加密資料的相同 keyring，或不同的 keyring。若要解密資料，解密 keyring 必須能夠存取加密 keyring 中的至少一個包裝金鑰。

AWS Database Encryption SDK 會將加密的資料金鑰從材料描述傳遞至 keyring，並要求 keyring 解密其中任何一個。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 Database Encryption SDK 使用純文字資料金鑰來解密資料。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，多重金鑰環會傳回由組成多重金鑰的所有金鑰環中所有包裝金鑰加密的資料金鑰複本，以及與資料金鑰相關聯的 MAC 金鑰。您可以使用 keyring 和多 keyring 中的任何一個包裝金鑰來解密資料。

AWS KMS 鑰匙圈

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

AWS KMS Keyring 使用對稱加密或非對稱 RSA AWS KMS keys來產生、加密和解密資料金鑰。 AWS Key Management Service (AWS KMS) 會保護您的 KMS 金鑰，並在 FIPS 邊界內執行密碼編譯操作。我們建議您盡可能使用具有類似安全屬性的 AWS KMS keyring 或 keyring。

您也可以在 AWS KMS keyring 中使用對稱多區域 KMS 金鑰。如需使用多區域的詳細資訊和範例 AWS KMS keys，請參閱使用多區域 AWS KMS keys。如需多區域金鑰的相關資訊，請參閱《 AWS Key Management Service 開發人員指南》中的使用多區域金鑰。

AWS KMS keyring 可以包含兩種類型的包裝金鑰：

- 產生器金鑰：產生純文字資料金鑰並將其加密。加密資料的 keyring 必須有一個產生器金鑰。
- 其他金鑰：加密產生器金鑰產生的純文字資料金鑰。 AWS KMS keyrings 可以有零個或多個其他金鑰。

您必須擁有產生器金鑰才能加密記錄。當 AWS KMS keyring 只有一個 AWS KMS 金鑰時，該金鑰會用來產生和加密資料金鑰。

與所有 keyring 一樣， AWS KMS keyring 可以獨立使用，也可以與相同或不同類型的其他 keyring ???搭配使用。

主題

- AWS KMS keyring 的必要許可
- 在 AWS KMS keyring AWS KMS keys 中識別
- 建立 AWS KMS keyring
- 使用多區域 AWS KMS keys
- 使用 AWS KMS 探索 keyring
- 使用 AWS KMS 區域探索 keyring

AWS KMS keyring 的必要許可

AWS Database Encryption SDK 不需要 ， AWS 帳戶 也不依賴任何 AWS 服務。不過，若要使用 AWS KMS keyring，您需要在 keyring AWS KMS keys 中的 上具有 AWS 帳戶 和下列最低許可。

- 若要使用 AWS KMS keyring 加密，您需要產生器金鑰上的 kms:GenerateDataKey 許可。您需要 AWS KMS keyring 中所有其他金鑰的 kms:Encrypt 許可。
- 若要使用 AWS KMS keyring 解密，您需要 AWS KMS 在 keyring 中至少一個金鑰上的 kms:Decrypt 許可。
- 若要使用由 AWS KMS keyring 組成的多金鑰環加密，您需要產生器 keyring 中產生器金鑰的 kms:GenerateDataKey 許可。您需要所有其他 AWS KMS keyring 中所有其他金鑰的 kms:Encrypt 許可。
- 若要使用非對稱 RSA AWS KMS keyring 加密，您不需要 kms:GenerateDataKey 或 kms:Encrypt，因為您必須在建立 keyring 時指定要用於加密的公有金鑰材料。使用此 keyring 加密時不會進行任何 AWS KMS 呼叫。若要使用非對稱 RSA AWS KMS keyring 解密，您需要 kms:Decrypt 許可。

如需 許可的詳細資訊 AWS KMS keys，請參閱《 AWS Key Management Service 開發人員指南》中的身分驗證和存取控制。

在 AWS KMS keyring AWS KMS keys 中識別

AWS KMS Keyring 可以包含一或多個 AWS KMS keys。若要在 AWS KMS keyring 中指定 AWS KMS key ，請使用支援的 AWS KMS 金鑰識別符。您可以使用金鑰識別符來識別 keyring AWS KMS key 中的 ，會因 操作和語言實作而有所不同。如需 金鑰識別符的詳細資訊 AWS KMS key，請參閱《 AWS Key Management Service 開發人員指南》中的金鑰識別符。

最佳實務是使用最適用於您任務的特定金鑰識別符。

- 若要使用 AWS KMS keyring 加密，您可以使用金鑰 ID、金鑰 ARN、別名名稱或別名 ARN 來加密資料。

> **Note**
>
> 如果您在加密 keyring 中為 KMS 金鑰指定別名名稱或別名 ARN，加密操作會將目前與別名相關聯的金鑰 ARN 儲存在加密資料金鑰的中繼資料中。它不會儲存別名。別名的變更不會影響用來解密加密資料金鑰的 KMS 金鑰。

- 若要使用 AWS KMS keyring 解密，您必須使用金鑰 ARN 來識別 AWS KMS keys。如需詳細資訊，請參閱 選取包裝金鑰。
- 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。

解密時， AWS 資料庫加密 SDK 會搜尋 AWS KMS keyring AWS KMS key ，尋找可解密其中一個加密資料金鑰的 。具體而言， AWS 資料庫加密 SDK 會對材料描述中的每個加密資料金鑰使用下列模式。

- AWS Database Encryption SDK 會從材料描述的中繼資料取得 AWS KMS key 加密資料金鑰的 金鑰 ARN。
- AWS Database Encryption SDK 會搜尋 AWS KMS key 具有相符金鑰 ARN 的 解密 keyring。
- 如果在 keyring 中找到 AWS KMS key 具有相符金鑰 ARN 的 ， AWS 資料庫加密 SDK AWS KMS 會要求使用 KMS 金鑰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)。

建立 AWS KMS keyring

您可以使用相同 AWS KMS key 或不同 AWS 帳戶 和 AWS KMS keys 中的單一或多個 來設定每個 AWS KMS keyring AWS 區域。 AWS KMS key 必須是對稱加密金鑰 (SYMMETRIC_DEFAULT) 或非對稱 RSA KMS 金鑰。您也可以使用對稱加密多區域 KMS 金鑰。您可以在多 AWS KMS 鍵環中使用一或多個鍵環。

您可以建立加密和解密資料的 AWS KMS keyring，也可以建立專門用於加密或解密的 AWS KMS keyring。當您建立 AWS KMS keyring 來加密資料時，必須指定產生器金鑰，這是用來產生純文字資料金鑰並進行加密 AWS KMS key 的 。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 AWS KMS keys 來加密相同的純文字資料金鑰。若要解密受此 keyring 保護的加密欄位，您使用的解密 keyring 必須包含至少一個在 keyring 中 AWS KMS keys 定義的 或 no AWS KMS keys。（沒有 的 AWS KMS keyring AWS KMS keys 稱為AWS KMS 探索 keyring。）

加密 keyring 或 multi-keyring 中的所有包裝金鑰都必須能夠加密資料金鑰。如果任何包裝金鑰無法加密，加密方法會失敗。因此，呼叫者必須擁有 keyring 中所有金鑰的必要許可。如果您使用探索 keyring 來加密資料，無論是單獨加密或在多 keyring 中加密，加密操作會失敗。

下列範例使用 `CreateAwsKmsMrkMultiKeyring`方法建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。`CreateAwsKmsMrkMultiKeyring` 方法會自動建立 AWS KMS 用戶端，並確保 keyring 將正確處理單一區域和多區域金鑰。這些範例使用金鑰 ARNs 來識別 KMS 金鑰。如需詳細資訊，請參閱在 AWS KMS keyring AWS KMS keys 中識別

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = kmsKeyArn
};
```

```
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
let mat_prov = client::Client::from_conf(provider_config)?;
let kms_keyring = mat_prov
    .create_aws_kms_mrk_multi_keyring()
    .generator(kms_key_id)
    .send()
    .await?;
```

下列範例使用 `CreateAwsKmsRsaKeyring`方法建立具有非對稱 RSA KMS 金鑰的 AWS KMS keyring。若要建立非對稱 RSA AWS KMS keyring，請提供下列值。

- `kmsClient`：建立新的 AWS KMS 用戶端
- `kmsKeyID`：識別非對稱 RSA KMS 金鑰的金鑰 ARN
- `publicKey`：UTF-8 編碼 PEM 檔案的 ByteBuffer，代表您傳遞給 之金鑰的公有金鑰 `kmsKeyID`
- `encryptionAlgorithm`：加密演算法必須為 `RSAES_OAEP_SHA_256`或 `RSAES_OAEP_SHA_1`

Java

```
 final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
    CreateAwsKmsRsaKeyringInput.builder()
        .kmsClient(KmsClient.create())
        .kmsKeyId(rsaKMSKeyArn)
        .publicKey(publicKey)
        .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
        .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
```

```
var createAwsKmsRsaKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = rsaKMSKeyArn,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(rsa_kms_key_arn)
    .public_key(public_key)

 .encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .send()
    .await?;
```

使用多區域 AWS KMS keys

您可以使用多區域 AWS KMS keys 做為 AWS 資料庫加密 SDK 中的包裝金鑰。如果您使用多區域金鑰進行加密 AWS 區域，則可以使用不同 中的相關多區域金鑰進行解密 AWS 區域。

多區域 KMS 金鑰是一組不同 AWS KMS keys 中的 AWS 區域 ，具有相同的金鑰材料和金鑰 ID。您可以使用這些相關金鑰，就像它們在不同區域中是相同的金鑰一樣。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加密，並在不同區域中解密，而無需進行跨區域呼叫 AWS KMS。如需多區域金鑰的相關資訊，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 資料庫加密 SDK AWS KMS multi-Region-aware keyring。此`CreateAwsKmsMrkMultiKeyring`方法同時支援單一區域和多區域金鑰。

- 對於單一區域金鑰，multi-Region-aware符號的行為與單一區域 AWS KMS keyring 相同。它只會嘗試使用加密資料的單一區域金鑰來解密加密文字。為了簡化您的 AWS KMS keyring 體驗，我們建議您每當您使用對稱加密 KMS 金鑰時，使用 `CreateAwsKmsMrkMultiKeyring`方法。
- 對於多區域索引鍵，multi-Region-aware符號會嘗試使用與加密資料相同的多區域索引鍵，或在您指定的區域中使用相關的多區域索引鍵來解密加密文字。

在需要多個 KMS 金鑰的multi-Region-aware keyring 中，您可以指定多個單一區域和多區域金鑰。不過，您只能從一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 指定多個金鑰識別符，建構器呼叫會失敗。

下列範例使用多區域 KMS 金鑰建立 AWS KMS keyring。這些範例指定多區域金鑰做為產生器金鑰，而單一區域金鑰做為子金鑰。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
    CreateAwsKmsMrkMultiKeyringInput.builder()
            .generator(multiRegionKeyArn)
            .kmsKeyIds(Collections.singletonList(kmsKeyArn))
            .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = multiRegionKeyArn,
    KmsKeyIds = new List<String> { kmsKeyArn }
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

```
let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(multiRegion_key_arn)
    .kms_key_ids(vec![key_arn.to_string()])
    .send()
    .await?;
```

當您使用多區域 AWS KMS keyring 時，您可以在嚴格的模式中解密加密文字或探索模式。若要以嚴格模式解密加密文字，請在您要解密加密文字的區域中，使用相關多區域金鑰的金鑰 ARN 來執行個體化多區域multi-Region-aware符號。如果您在不同的區域（例如，記錄加密的區域）中指定相關多區域金鑰的金鑰 ARN，multi-Region-aware符號會為此發出跨區域呼叫 AWS KMS key。

在嚴格模式下解密時，multi-Region-aware符號需要金鑰 ARN。它只接受一組相關多區域金鑰中的一個金鑰 ARN。

您也可以使用 AWS KMS 多區域金鑰在探索模式中解密。在探索模式中解密時，您不會指定任何 AWS KMS keys。（如需單一區域 AWS KMS 探索 keyring 的相關資訊，請參閱 使用 AWS KMS 探索 keyring。）

如果您使用多區域金鑰加密，則探索模式中的multi-Region-aware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，呼叫會失敗。在探索模式中， AWS 資料庫加密 SDK 不會嘗試對用於加密的多區域金鑰進行跨區域呼叫。

使用 AWS KMS 探索 keyring

解密時，最佳實務是指定 AWS 資料庫加密 SDK 可以使用的包裝金鑰。若要遵循此最佳實務，請使用 AWS KMS 解密 keyring，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 keyring，也就是未指定任何包裝金鑰的 AWS KMS keyring。

AWS Database Encryption SDK 為 AWS KMS 多區域金鑰提供標準 AWS KMS 探索 keyring 和探索 keyring。如需有關搭配 AWS 資料庫加密 SDK 使用多區域金鑰的資訊，請參閱 使用多區域 AWS KMS keys。

因為它未指定任何包裝金鑰，探索 keyring 無法加密資料。如果您使用探索 keyring 來加密資料，無論是單獨使用或在多金鑰集中，加密操作會失敗。

解密時，探索 keyring 可讓 AWS Database Encryption SDK 使用加密的資料金鑰 AWS KMS 來要求解密 AWS KMS key 任何加密的資料金鑰，無論誰擁有或有權存取該金鑰 AWS KMS key。只有在呼叫者擁有 的`kms:Decrypt`許可時，呼叫才會成功 AWS KMS key。

> **Important**
>
> 如果您在解密的多金鑰集中包含 AWS KMS 探索 keyring，則探索 keyring 會覆寫多金鑰集中其他 keyring 指定的所有 KMS 金鑰限制。 ???多鍵控的行為與其限制最少的鍵控一樣。如果您使用探索 keyring 來加密資料，無論是單獨使用或在多 keyring 中，加密操作會失敗

AWS Database Encryption SDK 提供 AWS KMS 探索 keyring，方便使用。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 – 探索 AWS KMS keyring 可以使用用於加密材料描述中資料金鑰的任何 AWS KMS key ，只要發起人具有使用該金鑰解密 AWS KMS key 的許可。這可能不是發起 AWS KMS key 人打算使用的 。例如，其中一個加密的資料金鑰可能已在安全性較低的情況下加密 AWS KMS key ，任何人都可以使用。
- 延遲和效能 – 探索 AWS KMS keyring 可能比其他 keyring 更慢，因為 AWS Database Encryption SDK 會嘗試解密所有加密的資料金鑰，包括 AWS KMS keys 其他 AWS 帳戶 和 區域中加密的資料金鑰，而且 AWS KMS keys 發起人無權使用 進行解密。

如果您使用探索 keyring，我們建議您使用探索篩選條件來限制 KMS 金鑰，這些金鑰可用於指定 AWS 帳戶 和 分割區中的金鑰。如需尋找帳戶 ID 和分割區的說明，請參閱 中的您的 AWS 帳戶 識別符和 ARN 格式AWS 一般參考。

下列程式碼範例會使用 AWS KMS 探索篩選條件來執行個體化探索 keyring，將 AWS 資料庫加密 SDK 可以使用的 KMS 金鑰限制在`aws`分割區和`111122223333`範例帳戶中的金鑰。

使用此程式碼之前，請將範例 AWS 帳戶 和分割區值取代為 AWS 帳戶 和分割區的有效值。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 AWS 區域，請使用`aws`分割區值。

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);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

使用 AWS KMS 區域探索 keyring

AWS KMS 區域探索 keyring 是未指定 KMS 金鑰 ARNs keyring。相反地，它允許 AWS 資料庫加密 SDK 僅使用 KMS 金鑰進行解密 AWS 區域。

使用 AWS KMS 區域探索 keyring 解密時， AWS 資料庫加密 SDK 會解密在指定 AWS KMS key 中 下加密的任何加密資料金鑰 AWS 區域。若要成功，發起人必須在加密資料金鑰 AWS 區域 的指定 AWS KMS keys 中，擁有至少一個 的`kms:Decrypt`許可。

與其他探索 keyring 一樣，區域探索 keyring 不會影響加密。只有在解密加密的欄位時才有效。如果您在用於加密和解密的多金鑰集中使用區域探索 keyring，則只有在解密時才有效。如果您使用多區域探索 keyring 來加密資料，無論是單獨加密或在多 keyring 中加密，加密操作會失敗。

> **Important**
>
> 如果您在解密多金鑰集中包含 AWS KMS 區域探索 keyring，區域探索 keyring 會覆寫多金鑰集中其他 keyring 指定的所有 KMS 金鑰限制。 ???多鍵控的行為與其限制最少的鍵控一樣。探索 AWS KMS keyring 本身或多 keyring 使用時，不會影響加密。

AWS Database Encryption SDK 中的區域探索 keyring 嘗試僅使用指定區域中的 KMS 金鑰進行解密。當您使用探索 keyring 時，您可以在 AWS KMS 用戶端上設定 區域。這些 AWS 資料庫加密 SDK 實作不會依區域篩選 KMS 金鑰，但 AWS KMS 會針對指定區域以外的 KMS 金鑰失敗解密請求。

如果您使用探索 keyring，我們建議您使用探索篩選條件，將解密中使用的 KMS 金鑰限制為指定 AWS 帳戶 和 分割區中的金鑰。

例如，下列程式碼會使用探索篩選條件建立 AWS KMS 區域探索 keyring。此 keyring 將 AWS 資料庫加密 SDK 限制在美國西部 （ 奧勒岡 ） 區域 (us-west-2) 的帳戶 111122223333 中的 KMS 金鑰。

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);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter,
    Regions = us-west-2
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .regions(us-west-2)
    .send()
    .await?;
```

AWS KMS 階層式 keyring

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

> **Note**
>
> 自 2023 年 7 月 24 日起，不支援在開發人員預覽期間建立的分支金鑰。建立新的分支金鑰，以繼續使用您在開發人員預覽期間建立的金鑰存放區。

使用 AWS KMS 階層式 keyring，您可以在對稱加密 KMS 金鑰下保護密碼編譯資料，而無需 AWS KMS 在每次加密或解密記錄時呼叫 。對於需要將對 的呼叫降至最低的應用程式 AWS KMS，以及可以重複使用某些密碼編譯材料而不違反其安全要求的應用程式，這是很好的選擇。

階層式 keyring 是一種密碼編譯資料快取解決方案，透過使用保留在 Amazon DynamoDB 資料表中的 AWS KMS 受保護分支金鑰，以及加密和解密操作中使用的本機快取分支金鑰資料來減少 AWS KMS 呼叫次數。DynamoDB 資料表做為金鑰存放區，可管理和保護分支金鑰。它會存放作用中的分支金鑰和所有舊版的分支金鑰。作用中分支金鑰是最新的分支金鑰版本。階層式 keyring 會針對每個加密請求使用唯一的資料加密金鑰，並使用衍生自作用中分支金鑰的唯一包裝金鑰來加密每個資料加密金鑰。階層式 keyring 取決於作用中分支索引鍵與其衍生包裝索引鍵之間建立的階層。

階層式 keyring 通常使用每個分支金鑰版本來滿足多個請求。但是，您可以控制重複使用作用中分支金鑰的程度，並判斷作用中分支金鑰的輪換頻率。分支金鑰的作用中版本會保持作用中，直到您將其輪換為止。舊版的作用中分支金鑰不會用於執行加密操作，但仍可以查詢和用於解密操作。

當您執行個體化階層式 keyring 時，它會建立本機快取。您可以指定快取限制，定義分支金鑰資料在本機快取內儲存的時間上限，以免過期並從快取中移出。階層式 keyring 會 AWS KMS 呼叫一次 來解密分支金鑰，並在操作中第一次指定 `branch-key-id` 時組合分支金鑰材料。然後，分支金鑰材料會存放在本機快取中，並重複使用於指定快取限制過期`branch-key-id`之前的所有加密和解密操作。在本機快取中存放分支金鑰材料可減少 AWS KMS 呼叫。例如，請考慮 15 分鐘的快取限制。如果您在該快取限制內執行 10，000 個加密操作，則傳統 AWS KMS keyring 將需要進行 10，000 次 AWS KMS 呼叫，以滿足 10，000 個加密操作。如果您有一個作用中的 `branch-key-id`，階層式 keyring 只需要呼叫一次 AWS KMS ，以滿足 10，000 個加密操作。

本機快取會將加密資料與解密資料分開。加密資料是從作用中分支金鑰組合而成，並重複使用於所有加密操作，直到快取限制過期為止。解密資料是從加密欄位中繼資料中識別的分支金鑰 ID 和版本組合而成，並且會重複使用於與分支金鑰 ID 和版本相關的所有解密操作，直到快取限制過期為止。本機快取一次可以存放相同分支金鑰的多個版本。當本機快取設定為使用 時branch key ID supplier，它也可以一次儲存來自多個作用中分支金鑰的分支金鑰材料。

> **Note**
>
> AWS 資料庫加密 SDK 中所有提到的階層式 keyring 都參考 AWS KMS 階層式 keyring。

主題

- 運作方式
- 先決條件
- 所需的許可
- 選擇快取
- 建立階層式 keyring
- 使用階層式 keyring 進行可搜尋加密

運作方式

下列逐步解說說明階層式 keyring 如何組合加密和解密材料，以及 keyring 對加密和解密操作進行的不同呼叫。如需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 keyring 技術詳細資訊。

加密和簽署

下列逐步解說說明階層式 keyring 如何組合加密資料並衍生唯一的包裝金鑰。

1. 加密方法會向階層式 keyring 詢問加密資料。Keyring 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金鑰材料來產生包裝金鑰。如果有有效的分支金鑰材料， keyring 會繼續進行步驟 4。
2. 如果沒有有效的分支金鑰材料，階層式 keyring 會查詢作用中分支金鑰的金鑰存放區。
   a. 金鑰存放區會呼叫 AWS KMS 來解密作用中分支金鑰，並傳回純文字作用中分支金鑰。識別作用中分支金鑰的資料會序列化，以在解密呼叫中提供其他已驗證的資料 (AAD) AWS KMS。
   b. 金鑰存放區會傳回純文字分支金鑰，以及可識別該分支金鑰的資料，例如分支金鑰版本。
3. 階層式 keyring 會組合分支金鑰材料（純文字分支金鑰和分支金鑰版本），並將複本存放在本機快取中。
4. 階層式 keyring 從純文字分支金鑰和 16 位元組隨機鹽中衍生唯一的包裝金鑰。它使用衍生的包裝金鑰來加密純文字資料金鑰的副本。

加密方法使用加密資料來加密和簽署記錄。如需如何在 AWS 資料庫加密 SDK 中加密和簽署記錄的詳細資訊，請參閱加密和簽署。

解密和驗證

下列逐步解說說明階層式 keyring 如何組合解密資料並解密加密的資料金鑰。

1. 解密方法會從加密記錄的資料描述欄位中識別加密的資料金鑰，並將其傳遞至階層式 keyring。
2. 階層式 keyring 會將識別加密資料金鑰的資料還原序列化，包括分支金鑰版本、16 位元組 salt，以及描述資料金鑰如何加密的其他資訊。

   如需詳細資訊，請參閱AWS KMS 階層式 keyring 技術詳細資訊。
3. 階層式 keyring 會檢查本機快取中是否有與步驟 2 中識別的分支金鑰版本相符的有效分支金鑰材料。如果有有效的分支金鑰材料， keyring 會繼續進行步驟 6。
4. 如果沒有有效的分支金鑰材料，階層式 keyring 會查詢符合步驟 2 中識別分支金鑰版本的分支金鑰存放區。

   a. 金鑰存放區會呼叫 AWS KMS 來解密分支金鑰，並傳回純文字作用中分支金鑰。識別作用中分支金鑰的資料會序列化，以在解密呼叫中提供其他已驗證的資料 (AAD) AWS KMS。

   b. 金鑰存放區會傳回純文字分支金鑰，以及可識別該分支金鑰的資料，例如分支金鑰版本。
5. 階層式 keyring 會組合分支金鑰材料（純文字分支金鑰和分支金鑰版本），並將複本存放在本機快取中。
6. 階層式 keyring 使用步驟 2 中識別的組合分支金鑰材料和 16 位元組的鹽，來重現加密資料金鑰的唯一包裝金鑰。
7. 階層式 keyring 使用重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字資料金鑰來解密和驗證記錄。如需如何在 AWS 資料庫加密 SDK 中解密和驗證記錄的詳細資訊，請參閱解密和驗證。

先決條件

在您建立和使用階層式 keyring 之前，請確定符合下列先決條件。

- 您或您的金鑰存放區管理員已建立金鑰存放區，並建立至少一個作用中的分支金鑰。
- 您已設定金鑰存放區動作。

> **Note**
>
> 如何設定金鑰存放區動作會決定您可以執行的操作，以及階層式 keyring 可以使用的 KMS 金鑰。如需詳細資訊，請參閱金鑰存放區動作。

- 您擁有存取和使用金鑰存放區和分支金鑰所需的 AWS KMS 許可。如需詳細資訊，請參閱the section called “所需的許可”。
- 您已檢閱支援的快取類型，並設定最符合您需求的快取類型。如需詳細資訊，請參閱 the section called “選擇快取”

所需的許可

AWS Database Encryption SDK 不需要 AWS 帳戶 ，也不依賴任何 AWS 服務。不過，若要使用階層式 keyring，您需要 AWS 帳戶 和下列有關金鑰存放區中對稱加密的最低許可 AWS KMS key(s)。

- 若要使用階層式 keyring 加密和解密資料，您需要 kms:Decrypt。
- 若要建立和輪換分支金鑰，您需要 kms:GenerateDataKeyWithoutPlaintext 和 kms:ReEncrypt。

如需控制對分支金鑰和金鑰存放區之存取的詳細資訊，請參閱the section called “實作最低權限的許可”。

選擇快取

階層式 keyring 可減少對 進行的呼叫數量， AWS KMS 方法是在本機快取用於加密和解密操作的分支金鑰材料。在建立階層 keyring 之前，您需要決定要使用的快取類型。您可以使用預設快取或自訂快取，以最符合您的需求。

階層式 keyring 支援下列快取類型：

- the section called “預設快取”
- the section called “MultiThreaded快取”
- the section called “StormTracking 快取”
- the section called “共用快取”

預設快取

對於大多數使用者，預設快取滿足其執行緒需求。預設快取旨在支援大量多執行緒環境。當分支金鑰材料項目過期時，預設快取 AWS KMS 會提前 10 秒通知一個執行緒，以阻止多個執行緒呼叫分支金鑰材料項目。這可確保只有一個執行緒將請求傳送至 AWS KMS 以重新整理快取。

預設和 StormTracking 快取支援相同的執行緒模型，但您只需指定使用預設快取的進入容量。如需更精細的快取自訂，請使用 the section called “StormTracking 快取”。

除非您想要自訂可在本機快取中存放的分支金鑰材料項目數目，否則建立階層式 keyring 時不需要指定快取類型。如果您未指定快取類型，階層式 keyring 會使用預設快取類型，並將輸入容量設定為 1000。

若要自訂預設快取，請指定下列值：

- 進入容量：限制可在本機快取中存放的分支金鑰材料項目數量。

Java

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

C# / .NET

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

Rust

```
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);
```

MultiThreaded快取

MultiThreaded快取可在多執行緒環境中安全使用，但它不提供任何功能來最小化 AWS KMS 或 Amazon DynamoDB 呼叫。因此，當分支金鑰材料項目過期時，所有執行緒都會同時收到通知。這可能會導致多個 AWS KMS 呼叫重新整理快取。

若要使用MultiThreaded快取，請指定下列值：

- 輸入容量：限制分支金鑰材料項目的數量，這些項目可以存放在本機快取中。
- 項目修剪尾端大小：定義達到項目容量時要修剪的項目數量。

Java

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

C# / .NET

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

Rust

```
CacheType::MultiThreaded(
            MultiThreadedCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .build()?)
```

StormTracking 快取

StormTracking 快取旨在支援大量多執行緒環境。當分支金鑰材料項目過期時，StormTracking 快取 AWS KMS 會通知一個執行緒分支金鑰材料項目會事先過期，以防止多個執行緒呼叫。這可確保只有一個執行緒將請求傳送至 AWS KMS 以重新整理快取。

若要使用 StormTracking 快取，請指定下列值：

- 進入容量：限制可在本機快取中存放的分支金鑰材料項目數量。

  預設值：1000 個項目
- 項目修剪尾端大小：定義一次要修剪的分支金鑰材料項目數量。

  預設值：1 個項目
- 寬限期：定義過期前嘗試重新整理分支金鑰材料的秒數。

  預設值：10 秒
- Grace 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
- Fan out：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
- 飛行中存留時間 (TTL)：定義直到嘗試重新整理分支金鑰材料逾時的秒數。每當快取傳回 以`NoSuchEntry`回應 時`GetCacheEntry`，該分支金鑰會被視為正在傳輸中，直到使用`PutCache`項目寫入相同的金鑰為止。

  預設值：10 秒
- 休眠：定義`fanOut`超過 時執行緒應休眠的秒數。

  預設值：20 毫秒

Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
```

```
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

C# / .NET

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

Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

共用快取

根據預設，階層式 keyring 會在您每次執行個體化 keyring 時建立新的本機快取。不過，共用快取可讓您跨多個階層 keyring 共用快取，有助於節省記憶體。共用快取不會為每個您執行個體化的階層式 keyring 建立新的密碼編譯材料快取，而是在記憶體中只儲存一個快取，所有參考它的階層式 keyring

都可以使用。共用快取可避免在 keyring 之間重複密碼編譯資料，有助於最佳化記憶體用量。相反地，階層式 keyring 可以存取相同的基礎快取，減少整體記憶體佔用空間。

建立共用快取時，您仍然定義快取類型。您可以指定 the section called “預設快取”、 the section called “MultiThreaded快取”或 the section called “StormTracking 快取”做為快取類型，或取代任何相容的自訂快取。

資料分割

多個階層式 keyring 可以使用單一共用快取。當您使用共用快取建立階層式 keyring 時，您可以定義選用的分割區 ID。分割區 ID 會區分要寫入快取的階層式 keyring。如果兩個階層 keyring 參考相同的分割區 ID、 logical key store name和分支金鑰 ID，則兩個 keyring 將在快取中共用相同的快取項目。如果您建立兩個具有相同共用快取的階層式 keyring，但分割區 IDs 不同，則每個 keyring 只會存取共用快取內其自有指定分割區的快取項目。分割區在共用快取中充當邏輯分割，允許每個階層式 keyring 在自己的指定分割區上獨立運作，而不會干擾存放在另一個分割區中的資料。

如果您想要重複使用或共用分割區中的快取項目，您必須定義自己的分割區 ID。當您將分割區 ID 傳遞至階層式 keyring 時，keyring 可以重複使用已存在於共用快取中的快取項目，而不必再次擷取並重新授權分支金鑰材料。如果您未指定分割區 ID，則每次您執行個體化階層式 keyring 時，系統會自動將唯一的分割區 ID 指派給 keyring。

下列程序示範如何使用預設快取類型建立共用快取，並將其傳遞至階層式 keyring。

1. 使用材質提供者程式庫 `CryptographicMaterialsCache`(MPL) 建立 (CMC)。 https://github.com/aws/aws-cryptographic-material-providers-library

   Java

   ```
   // Instantiate the MPL
   final MaterialProviders matProv =
       MaterialProviders.builder()
           .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
           .build();

   // Create a CacheType object for the Default cache
   final CacheType cache =
       CacheType.builder()
           .Default(DefaultCache.builder().entryCapacity(100).build())
           .build();
   ```

```
// Create a CMC using the default cache
final CreateCryptographicMaterialsCacheInput cryptographicMaterialsCacheInput =
    CreateCryptographicMaterialsCacheInput.builder()
        .cache(cache)
        .build();

final ICryptographicMaterialsCache sharedCryptographicMaterialsCache =
    matProv.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

C# / .NET

```
// Instantiate the MPL
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create a CacheType object for the Default cache
var cache = new CacheType { Default = new DefaultCache{EntryCapacity = 100} };

// Create a CMC using the default cache
var cryptographicMaterialsCacheInput = new
 CreateCryptographicMaterialsCacheInput {Cache = cache};

var sharedCryptographicMaterialsCache =
 materialProviders.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

Rust

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

// Create a CacheType object for the default cache
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);

// Create a CMC using the default cache
let shared_cryptographic_materials_cache: CryptographicMaterialsCacheRef = mpl.
    create_cryptographic_materials_cache()
    .cache(cache)
    .send()
```

```
    .await?;
```

2. 建立共用快取的`CacheType`物件。

   `sharedCryptographicMaterialsCache` 將您在步驟 1 中建立的 傳遞至新`CacheType`物件。

   Java

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   final CacheType sharedCache =
       CacheType.builder()
           .Shared(sharedCryptographicMaterialsCache)
           .build();
   ```

   C# / .NET

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   var sharedCache = new CacheType { Shared = sharedCryptographicMaterialsCache };
   ```

   Rust

   ```
   // Create a CacheType object for the shared_cryptographic_materials_cache
   let shared_cache: CacheType =
    CacheType::Shared(shared_cryptographic_materials_cache);
   ```

3. 將`sharedCache`物件從步驟 2 傳遞到您的階層 keyring。

   當您使用共用快取建立階層式 keyring 時，您可以選擇定義 `partitionID`，以在多個階層式 keyring 之間共用快取項目。如果您未指定分割區 ID，階層式 keyring 會自動為 keyring 指派唯一的分割區 ID。

   > **Note**
   >
   > 如果您建立兩個或多個參考相同分割區 ID、 和分支金鑰 ID 的 keyringlogical key store name，您的階層 keyring 將共用共用共用快取中的相同快取項目。如果您不希望多個 keyring 共用相同的快取項目，則必須為每個階層 keyring 使用唯一的分割區 ID。

下列範例會使用 建立階層式 keyringbranch key ID supplier，快取限制為 600 秒。如需下列階層式 keyring 組態中定義值的詳細資訊，請參閱 the section called “建立階層式 keyring”。

Java

```
// Create the Hierarchical keyring
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(sharedCache)
        .partitionID(partitionID)
        .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
// Create the Hierarchical keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   Cache = sharedCache,
   TtlSeconds = 600,
   PartitionId = partitionID
};
var keyring =
 materialProviders.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
```

```
    // pass it to different Hierarchical Keyrings, it will still point to the
same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

建立階層式 keyring

若要建立階層式 keyring，您必須提供下列值：

- 金鑰存放區名稱

  您或金鑰存放區管理員建立的 DynamoDB 資料表名稱，以做為您的金鑰存放區。

-

  快取限制存留時間 (TTL)

  分支金鑰材料項目在本機快取內過期前可以使用的秒數。快取限制 TTL 決定用戶端呼叫 AWS KMS 以授權使用分支金鑰的頻率。該值必須大於零。快取限制 TTL 過期後，永遠不會提供項目，並且會從本機快取中移出。

- 分支金鑰識別符

  您可以靜態設定 `branch-key-id` 來識別金鑰存放區中的單一作用中分支金鑰，或提供分支金鑰 ID 供應商。

  分支金鑰 ID 供應商會使用存放在加密內容中的欄位，來判斷解密記錄所需的分支金鑰。根據預設，加密內容中只會包含分割區和排序金鑰。不過，您可以使用`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`密碼編譯動作，在加密內容中包含其他欄位。

  我們強烈建議針對每個租戶都有自己的分支金鑰的多租戶資料庫使用分支金鑰 ID 供應商。您可以使用分支金鑰 ID 供應商為分支金鑰 IDs 建立易記的名稱，以便輕鬆識別特定租用戶的正確分支金鑰 ID。例如，易記名稱可讓您將分支金鑰稱為 `tenant1`，而非 `b3f61619-4d35-48ad-a275-050f87e15122`。

對於解密操作，您可以靜態設定單一階層式 keyring 以限制對單一租用戶的解密，也可以使用分支金鑰 ID 供應商來識別哪些租用戶負責解密記錄。

- （選用）快取

  如果您想要自訂快取類型或可存放在本機快取中的分支金鑰材料項目數量，請在初始化 keyring 時指定快取類型和項目容量。

  階層式 keyring 支援下列快取類型：預設、MultiThreaded、StormTracking和共用。如需示範如何定義每個快取類型的詳細資訊和範例，請參閱 the section called “選擇快取”。

  如果您未指定快取，階層式 keyring 會自動使用預設快取類型，並將輸入容量設定為 1000。

- （選用）分割區 ID

  如果您指定 the section called “共用快取”，您可以選擇定義分割區 ID。分割區 ID 會區分要寫入快取的階層式 keyring。如果您想要重複使用或共用分割區中的快取項目，您必須定義自己的分割區 ID。您可以為分割區 ID 指定任何字串。如果您未指定分割區 ID，建立時會自動將唯一的分割區 ID 指派給 keyring。

  如需詳細資訊，請參閱Partitions。

  > **Note**
  > 如果您建立兩個或多個參考相同分割區 ID、 和分支金鑰 ID 的 keyringlogical key store name，您的階層 keyring 將共用共用共用快取中的相同快取項目。如果您不希望多個 keyring 共用相同的快取項目，則必須為每個階層 keyring 使用唯一的分割區 ID。

- （選用）授予權杖的清單

  如果您使用授權控制對階層式 keyring 中 KMS 金鑰的存取，您必須在初始化 keyring 時提供所有必要的授予權杖。

使用靜態分支金鑰 ID 建立階層式 keyring

下列範例示範如何建立具有靜態分支金鑰 ID、 the section called “預設快取”和快取限制 TTL 600 秒的階層式 keyring。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
```

```
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id(branch_key_id)
    .key_store(branch_key_store_name)
    .ttl_seconds(600)
    .send()
    .await?;
```

使用分支金鑰 ID 供應商建立階層 keyring

下列程序示範如何使用分支金鑰 ID 供應商建立階層 keyring。

1. 建立分支金鑰 ID 供應商

下列範例會為步驟 1 中建立的兩個分支金鑰建立易記的名稱，並呼叫 `CreateDynamoDbEncryptionBranchKeyIdSupplier` 來使用 DynamoDB 用戶端的 AWS Database Encryption SDK 建立分支金鑰 ID 供應商。

Java

```
// Create friendly names for each branch-key-id
class ExampleBranchKeyIdSupplier implements IDynamoDbKeyBranchKeyIdSupplier {
    private static String branchKeyIdForTenant1;
    private static String branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenant1 = tenant1Id;
        this.branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
final DynamoDbEncryption ddbEnc = DynamoDbEncryption.builder()
        .DynamoDbEncryptionConfig(DynamoDbEncryptionConfig.builder().build())
        .build();
final BranchKeyIdSupplier branchKeyIdSupplier =
 ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    CreateDynamoDbEncryptionBranchKeyIdSupplierInput.builder()
            .ddbKeyBranchKeyIdSupplier(new ExampleBranchKeyIdSupplier(branch-
key-ID-tenant1, branch-key-ID-tenant2))
            .build()).branchKeyIdSupplier();
```

C# / .NET

```
// Create friendly names for each branch-key-id
 class ExampleBranchKeyIdSupplier : DynamoDbKeyBranchKeyIdSupplierBase {
    private String _branchKeyIdForTenant1;
    private String _branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this._branchKeyIdForTenant1 = tenant1Id;
        this._branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
var ddbEnc = new DynamoDbEncryption(new DynamoDbEncryptionConfig());
var branchKeyIdSupplier = ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    new CreateDynamoDbEncryptionBranchKeyIdSupplierInput
    {
```

```
        DdbKeyBranchKeyIdSupplier = new ExampleBranchKeyIdSupplier(branch-key-
ID-tenant1, branch-key-ID-tenant2)
    }).BranchKeyIdSupplier;
```

Rust

```
// Create friendly names for each branch_key_id
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant1: String,
    branch_key_id_for_tenant2: String,
}

impl ExampleBranchKeyIdSupplier {
    pub fn new(tenant1_id: &str, tenant2_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant1: tenant1_id.to_string(),
            branch_key_id_for_tenant2: tenant2_id.to_string(),
        }
    }
}

// Create the branch key ID supplier
let dbesdk_config = DynamoDbEncryptionConfig::builder().build()?;
let dbesdk = dbesdk_client::Client::from_conf(dbesdk_config)?;
let supplier = ExampleBranchKeyIdSupplier::new(tenant1_branch_key_id,
 tenant2_branch_key_id);

let branch_key_id_supplier = dbesdk
    .create_dynamo_db_encryption_branch_key_id_supplier()
    .ddb_key_branch_key_id_supplier(supplier)
    .send()
    .await?
    .branch_key_id_supplier
    .unwrap();
```

2. 建立階層式 keyring

下列範例會使用步驟 1 中建立的分支金鑰 ID 供應商初始化階層式 keyring，快取限制 TLL 為 600 秒，快取大小上限為 1000。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
```

```
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 100 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id_supplier(branch_key_id_supplier)
    .key_store(key_store)
    .ttl_seconds(600)
    .send()
    .await?;
```

使用階層式 keyring 進行可搜尋加密

可搜尋加密可讓您在不解密整個資料庫的情況下搜尋加密的記錄。這可透過使用信標為加密欄位的純文字值編製索引來完成。若要實作可搜尋加密，您必須使用階層式 keyring。

金鑰存放區`CreateKey`操作會產生分支金鑰和信標金鑰。分支金鑰用於記錄加密和解密操作。信標金鑰用於產生信標。

分支金鑰和信標金鑰受到 AWS KMS key 您在建立金鑰存放區服務時指定的相同保護。`CreateKey` 操作呼叫 AWS KMS 以產生分支金鑰後，它會再次呼叫 kms:GenerateDataKeyWithoutPlaintext，以使用以下請求產生信標金鑰。

```
{
   "EncryptionContext": {
      "branch-key-id" : "branch-key-id",
      "type" : type,
      "create-time" : "timestamp",
      "logical-key-store-name" : "the logical table name for your key store",
      "kms-arn" : the KMS key ARN,
      "hierarchy-version" : 1
   },
   "KeyId": "the KMS key ARN",
   "NumberOfBytes": "32"
}
```

產生這兩個金鑰後，`CreateKey`操作會呼叫 ddb：TransactWriteItems 來寫入兩個新項目，這些項目將保留分支金鑰和信標金鑰在您的分支金鑰存放區中。

當您設定標準信標時， AWS 資料庫加密 SDK 會查詢信標金鑰的金鑰存放區。然後，它會使用以 HMAC extract-and-expand金鑰衍生函數 (HKDF)，將信標金鑰與標準信標名稱結合，為指定的信標建立 HMAC 金鑰。

與分支金鑰不同，金鑰存放區`branch-key-id`中每個 只有一個信標金鑰版本。信標金鑰永遠不會輪換。

定義您的信標金鑰來源

當您定義標準和複合信標的信標版本時，您必須識別信標金鑰，並定義信標金鑰資料的快取存留時間 (TTL)。信標金鑰資料會存放在與分支金鑰不同的本機快取中。下列程式碼片段示範如何`keySource`為單一租戶資料庫定義 。透過`branch-key-id`與其相關聯的 來識別您的信標金鑰。

Java

```
keySource(BeaconKeySource.builder()
        .single(SingleKeyStore.builder()
                .keyId(branch-key-id)
                .cacheTTL(6000)
                .build())
        .build())
```

C# / .NET

```
KeySource = new BeaconKeySource
{
    Single = new SingleKeyStore
    {
       KeyId = branch-key-id,
       CacheTTL = 6000
    }
}
```

Rust

```
 .key_source(BeaconKeySource::Single(
    SingleKeyStore::builder()
        // `keyId` references a beacon key.
        // For every branch key we create in the keystore,
        // we also create a beacon key.
        // This beacon key is not the same as the branch key,
        // but is created with the same ID as the branch key.
        .key_id(branch_key_id)
        .cache_ttl(6000)
        .build()?,
))
```

定義多租戶資料庫中的信標來源

如果您有多租戶資料庫，您必須在設定 時指定下列值keySource。

- keyFieldName

定義存放與用於為指定租用戶產生信標之信標金鑰branch-key-id相關聯的 的欄位名稱。keyFieldName 可以是任何字串，但對於資料庫中的所有其他欄位而言，它必須是唯一的。當您將新記錄寫入資料庫時，識別用於為該記錄產生任何信標的branch-key-id信標金鑰的 會存放在此欄位中。您必須在信標查詢中包含此欄位，並識別重新計算信標所需的適當信標金鑰材料。如需詳細資訊，請參閱查詢多租戶資料庫中的信標。

- cacheTTL

  在本機信標快取內，信標金鑰材料項目在過期前可以使用的秒數。該值必須大於零。當快取限制 TTL 過期時，會從本機快取中移出項目。

- （選用）快取

  如果您想要自訂快取類型或可存放在本機快取中的分支金鑰材料項目數量，請在初始化 keyring 時指定快取類型和項目容量。

  階層式 keyring 支援下列快取類型：預設、MultiThreaded、StormTracking和共用。如需示範如何定義每個快取類型的詳細資訊和範例，請參閱the section called “選擇快取”。

  如果您未指定快取，階層式 keyring 會自動使用預設快取類型，並將進入容量設定為 1000。

下列範例會使用分支金鑰 ID 供應商建立階層式 keyring，快取限制 TLL 為 600 秒，而進入容量為 1000。

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(1000)
                        .build())
                .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 1000 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
    let mat_prov = client::Client::from_conf(provider_config)?;
    let kms_keyring = mat_prov
        .create_aws_kms_hierarchical_keyring()
        .branch_key_id(branch_key_id)
        .key_store(key_store)
        .ttl_seconds(600)
        .send()
        .await?;
```

AWS KMS ECDH 鑰匙圈

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

> **Important**
>
> AWS KMS ECDH keyring 僅適用於材質提供者程式庫的 1.5.0 版或更新版本。

AWS KMS ECDH keyring 使用非對稱金鑰協議AWS KMS keys，在兩方之間衍生共用對稱包裝金鑰。首先， keyring 使用橢圓曲線 Diffie-Hellman (ECDH) 金鑰協議演算法，從寄件者的 KMS 金鑰對和收件人的公有金鑰中的私有金鑰衍生共用秘密。然後， keyring 會使用共用秘密衍生共用包裝金鑰，以保護資料加密金鑰。 AWS Database Encryption SDK 用來衍生共用包裝金鑰的金鑰衍生函數 (`KDF_CTR_HMAC_SHA384`) 符合金鑰衍生的 NIST 建議。

金鑰衍生函數會傳回 64 個位元組的鍵控材料。為了確保雙方都使用正確的金鑰材料， AWS 資料庫加密 SDK 會使用前 32 個位元組做為承諾金鑰，最後 32 個位元組做為共用包裝金鑰。在解密時，如果 keyring 無法重現存放在加密記錄的材料描述欄位中的相同承諾金鑰和共用包裝金鑰，則操作會失敗。例如，如果您加密的記錄具有以 Alice 的私有金鑰和 Bob 的公有金鑰設定的 keyring，則以 Bob 的私有金鑰和 Alice 的公有金鑰設定的 keyring 將重現相同的承諾金鑰和共用包裝金鑰，並能夠解密記錄。如果 Bob 的公有金鑰不是來自 KMS 金鑰對，則 Bob 可以建立原始 ECDH 金鑰集來解密記錄。

AWS KMS ECDH keyring 使用 AES-GCM 使用對稱金鑰加密記錄。然後，資料金鑰會使用 AES-GCM 使用衍生的共用包裝金鑰進行信封加密。每個 AWS KMS ECDH keyring 只能有一個共用包裝金鑰，但您可以在多 keyring 中單獨包含多個 AWS KMS ECDH keyring 或與其他 keyring 一起包含。

主題

- AWS KMS ECDH keyrings 的必要許可
- 建立 AWS KMS ECDH keyring
- 建立 AWS KMS ECDH 探索 keyring

AWS KMS ECDH keyrings 的必要許可

AWS Database Encryption SDK 不需要 AWS 帳戶，也不依賴任何 AWS 服務。不過，若要使用 AWS KMS ECDH keyring，您需要 AWS 帳戶和下列 keyring AWS KMS keys 中 的最低許可。許可會根據您使用的金鑰協議結構描述而有所不同。

- 若要使用`KmsPrivateKeyToStaticPublicKey`金鑰協議結構描述來加密和解密記錄，您需要寄件者非對稱 KMS 金鑰對上的 kms:GetPublicKey 和 kms:DeriveSharedSecret。如果您在執行個體化 keyring 時直接提供寄件者的 DER 編碼公有金鑰，則只需要寄件者的非對稱 KMS 金鑰對上的 kms:DeriveSharedSecret 許可。
- 若要使用`KmsPublicKeyDiscovery`金鑰協議結構描述解密記錄，您需要指定非對稱 KMS 金鑰對上的 kms:DeriveSharedSecret 和 kms:GetPublicKey 許可。

建立 AWS KMS ECDH keyring

若要建立加密和解密資料的 AWS KMS ECDH keyring，您必須使用`KmsPrivateKeyToStaticPublicKey`金鑰協議結構描述。若要使用`KmsPrivateKeyToStaticPublicKey`金鑰協議結構描述初始化 AWS KMS ECDH keyring，請提供下列值：

- 寄件者的 AWS KMS key ID

  必須識別`KeyUsage`值為 的非對稱 NIST 建議的橢圓曲線 (ECC)KMS 金鑰對`KEY_AGREEMENT`。寄件者的私有金鑰用於衍生共用秘密。

- （選用） 寄件者的公有金鑰

  必須是 DER 編碼的 X.509 公有金鑰，也稱為 `SubjectPublicKeyInfo`(SPKI)，如 RFC 5280 所定義。

  AWS KMS GetPublicKey 操作會以所需的 DER 編碼格式傳回非對稱 KMS 金鑰對的公有金鑰。

  若要減少 keyring 進行的 AWS KMS 呼叫數量，您可以直接提供寄件者的公有金鑰。如果未為寄件者的公有金鑰提供值，則 keyring 會呼叫 AWS KMS 來擷取寄件者的公有金鑰。

- 收件人的公有金鑰

  您必須提供收件人的 DER 編碼 X.509 公有金鑰，也稱為 `SubjectPublicKeyInfo`(SPKI)，如 RFC 5280 所定義。

  AWS KMS GetPublicKey 操作會以所需的 DER 編碼格式傳回非對稱 KMS 金鑰對的公有金鑰。

- 曲線規格

  識別指定金鑰對中的橢圓曲線規格。寄件者和收件人的金鑰對必須具有相同的曲線規格。

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

- （選用） 授予權杖的清單

  如果您使用授權控制對 AWS KMS ECDH keyring 中 KMS 金鑰的存取，您必須在初始化 keyring 時提供所有必要的授予權杖。

C# / .NET

下列範例會使用寄件者的 KMS 金鑰、寄件者的公有金鑰和收件人的公有金鑰，使用 建立 AWS KMS ECDH keyring。此範例使用選用`senderPublicKey`參數來提供寄件者的公有金鑰。如果您未提供寄件者的公有金鑰， keyring 會呼叫 AWS KMS 來擷取寄件者的公有金鑰。寄件者和收件人的金鑰對都在`ECC_NIST_P256`曲線上。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Must be DER-encoded X.509 public keys
var BobPublicKey = new MemoryStream(new byte[] { });
var AlicePublicKey = new MemoryStream(new byte[] { });

// Create the AWS KMS ECDH static keyring
var staticConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPrivateKeyToStaticPublicKey = new KmsPrivateKeyToStaticPublicKeyInput
    {
        SenderKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        SenderPublicKey = BobPublicKey,
        RecipientPublicKey = AlicePublicKey
    }
};

var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = staticConfiguration
};

var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

下列範例會使用寄件者的 KMS 金鑰、寄件者的公有金鑰和收件人的公有金鑰，使用 建立 AWS KMS ECDH keyring。此範例使用選用`senderPublicKey`參數來提供寄件者的公有金鑰。如果您未提供寄件者的公有金鑰， keyring 會呼叫 AWS KMS 來擷取寄件者的公有金鑰。寄件者和收件人的金鑰對都在`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();
```

Rust

下列範例會使用寄件者的 KMS 金鑰、寄件者的公有金鑰和收件人的公有金鑰，使用 建立 AWS KMS ECDH keyring。此範例使用選用`sender_public_key`參數來提供寄件者的公有金鑰。如果您未提供寄件者的公有金鑰， keyring 會呼叫 AWS KMS 來擷取寄件者的公有金鑰。

```
// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
```

```
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

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

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

建立 AWS KMS ECDH 探索 keyring

解密時，最佳實務是指定 AWS 資料庫加密 SDK 可以使用的金鑰。若要遵循此最佳實務，請使用具有`KmsPrivateKeyToStaticPublicKey`金鑰協議結構描述的 AWS KMS ECDH keyring。不過，您也可以建立 AWS KMS ECDH 探索 keyring，也就是可解密指定 KMS 金鑰對之公有金鑰與儲存在加密記錄之材料描述欄位中的收件人公有金鑰之任何記錄的 AWS KMS ECDH keyring。

> **Important**
>
> 當您使用`KmsPublicKeyDiscovery`金鑰協議結構描述解密記錄時，您接受所有公有金鑰，無論誰擁有它。

若要使用`KmsPublicKeyDiscovery`金鑰協議結構描述初始化 AWS KMS ECDH keyring，請提供下列值：

- 收件人的 AWS KMS key ID

  必須識別KeyUsage值為 的非對稱 NIST 建議的橢圓曲線 (ECC)KMS 金鑰對KEY_AGREEMENT。

- 曲線規格

  識別收件人 KMS 金鑰對中的橢圓曲線規格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

- （選用）授予權杖的清單

  如果您使用授權控制對 AWS KMS ECDH keyring 中 KMS 金鑰的存取，您必須在初始化 keyring 時提供所有必要的授予權杖。

C# / .NET

下列範例會在ECC_NIST_P256曲線上建立具有 KMS 金鑰對的 AWS KMS ECDH 探索 keyring。您必須擁有指定 KMS 金鑰對的 kms:GetPublicKey 和 kms:DeriveSharedSecret 許可。此 keyring 可以解密指定 KMS 金鑰對的公有金鑰符合存放在加密記錄之材料描述欄位中的收件人公有金鑰的任何記錄。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create the AWS KMS ECDH discovery keyring
var discoveryConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPublicKeyDiscovery = new KmsPublicKeyDiscoveryInput
    {
        RecipientKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
    }

};
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = discoveryConfiguration
};
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

下列範例會在ECC_NIST_P256曲線上建立具有 KMS 金鑰對的 AWS KMS ECDH 探索 keyring。您必須具有指定 KMS 金鑰對的 kms:GetPublicKey 和 kms:DeriveSharedSecret 許可。此 keyring 可以解密指定 KMS 金鑰對的公有金鑰符合存放在加密記錄之材料描述欄位中的收件人公有金鑰的任何記錄。

```
// 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();
```

Rust

```
// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

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

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
```

```
    .await?;
```

原始 AES keyring

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

AWS Database 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。您需要產生、存放和保護金鑰材料，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，請使用原始 AES 金鑰環。

Raw AES keyring 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。每個原始 AES keyring 中只能指定一個包裝金鑰，但您可以將多個原始 AES keyring 單獨或與其他 keyring 一起包含在多 keyring 中。

金鑰命名空間和名稱

若要識別 keyring 中的 AES 金鑰，原始 AES keyring 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 AWS Database Encryption SDK 新增至記錄的資料描述中。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 中的提供者 ID （或提供者) 和金鑰 ID 欄位`JceMasterKey`。

如果您建構不同的 keyring 來加密和解密指定欄位，命名空間和名稱值至關重要。如果解密 keyring 中的金鑰命名空間和金鑰名稱與加密 keyring 中的金鑰命名空間和金鑰名稱不完全且區分大小寫，即使金鑰材料位元組相同，也不會使用解密 keyring。

例如，您可以定義具有金鑰命名空間`HSM_01`和金鑰名稱 的原始 AES 金鑰環`AES_256_012`。然後，您可以使用該 keyring 來加密一些資料。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來建構原始 AES 金鑰環。

下列範例示範如何建立原始 AES Keyring。`AESWrappingKey` 變數代表您提供的金鑰材料。

Java

```
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);
```

C# / .NET

```
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
var keyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var matProv = new MaterialProviders(new MaterialProvidersConfig());
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
```

```
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

原始 RSA keyring

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

Raw RSA keyring 會使用您提供的 RSA 公有和私有金鑰，對本機記憶體中的資料金鑰執行非對稱加密和解密。您需要產生、存放和保護私有金鑰，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，您可以使用只有公有金鑰的原始 RSA 金鑰環來加密資料，也可以使用只有私有金鑰的原始 RSA 金鑰環來解密資料。您可以在多金鑰集中包含任何原始 RSA 金鑰環。如果您使用公有和私有金鑰設定原始 RSA 金鑰環，請確定它們是相同金鑰對的一部分。

原始 RSA keyring 等同於 ，並在與 RSA 非對稱加密金鑰搭配使用 適用於 JAVA 的 AWS Encryption SDK 時與 中的 JceMasterKey 互通。

> **Note**
>
> Raw RSA keyring 不支援非對稱 KMS 金鑰。若要使用非對稱 RSA KMS 金鑰，請建構 AWS KMS keyring。

命名空間和名稱

若要識別 keyring 中的 RSA 金鑰材料，原始 RSA keyring 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 AWS Database Encryption SDK 新增至記錄的資料描述中。我們建議您使用金鑰命名空間和金鑰名稱，以識別 HSM 或金鑰管理系統中的 RSA 金鑰對 （或其私有金鑰）。

> **Note**
>
> 金鑰命名空間和金鑰名稱等同於 中的提供者 ID （或提供者) 和金鑰 ID 欄位`JceMasterKey`。

如果您建構不同的 keyring 來加密和解密指定的記錄，命名空間和名稱值至關重要。如果解密 keyring 中的金鑰命名空間和金鑰名稱與加密 keyring 中的金鑰命名空間和金鑰名稱不完全且區分大小寫，即使金鑰來自相同的金鑰對，也不會使用解密 keyring。

無論 keyring 包含 RSA 公有金鑰、RDA 私有金鑰，或金鑰對中的兩個金鑰，加密和解密 keyring 中金鑰材料的金鑰命名空間和金鑰名稱都必須相同。例如，假設您使用 RSA 公有金鑰的原始 RSA keyring 來加密資料，該金鑰命名空間`HSM_01`和金鑰名稱為 `RSA_2048_06`。若要解密該資料，請使用私有金鑰 （或金鑰對） 和相同的金鑰命名空間和名稱來建構原始 RSA 金鑰環。

填充模式

您必須為用於加密和解密的原始 RSA 金鑰環指定填充模式，或使用語言實作的功能來為您指定。

AWS Encryption SDK 支援下列填補模式，受限於每種語言的限制。我們建議使用 OAEP 填補模式，特別是使用 SHA-256 的 OAEP 和使用 SHA-256 填補的 MGF1。PKCS1 填補模式僅支援回溯相容性。

- 使用 SHA-1 的 OAEP 和使用 SHA-1 填充的 MGF1 SHA-1
- OAEP 搭配 SHA-256 和 MGF1 搭配 SHA-256 Padding
- 使用 SHA-384 的 OAEP 和使用 SHA-384 Padding 的 MGF1
- OAEP 搭配 SHA-512 和 MGF1 搭配 SHA-512 Padding
- PKCS1 1.5 版填充

下列 Java 範例示範如何使用 RSA 金鑰對的公有和私有金鑰建立原始 RSA 金鑰環，以及使用 SHA-256 建立 OAEP，以及使用 SHA-256 填補模式建立 MGF1。`RSAPublicKey` 和 `RSAPrivateKey`變數代表您提供的金鑰材料。

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);
```

C# / .NET

```
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 keyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
};

// Create the keyring
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name("RSA_2048_06")
    .key_namespace("HSM_01")
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
```

```
    .public_key(RSA_public_key)
    .private_key(RSA_private_key)
    .send()
    .await?;
```

原始 ECDH 鑰匙圈

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

> **Important**
>
> 原始 ECDH keyring 僅適用於 Material Providers Library 的 1.5.0 版。

Raw ECDH keyring 使用您提供的橢圓曲線公有私有金鑰對，在兩方之間衍生共用包裝金鑰。首先，keyring 會使用寄件者的私有金鑰、收件人的公有金鑰和橢圓曲線 Diffie-Hellman (ECDH) 金鑰協議演算法衍生共用秘密。然後， keyring 會使用共用秘密衍生共用包裝金鑰，以保護資料加密金鑰。 AWS Database Encryption SDK 用來衍生共用包裝金鑰的金鑰衍生函數 (`KDF_CTR_HMAC_SHA384`) 符合金鑰衍生的 NIST 建議。

金鑰衍生函數會傳回 64 個位元組的鍵控材料。為了確保雙方都使用正確的金鑰材料， AWS 資料庫加密 SDK 會使用前 32 個位元組做為承諾金鑰，最後 32 個位元組做為共用包裝金鑰。在解密時，如果 keyring 無法重現存放在加密記錄的材料描述欄位中的相同承諾金鑰和共用包裝金鑰，則操作會失敗。例如，如果您加密的記錄具有以 Alice 的私有金鑰和 Bob 的公有金鑰設定的 keyring，則以 Bob 的私有金鑰和 Alice 的公有金鑰設定的 keyring 將重現相同的承諾金鑰和共用包裝金鑰，並能夠解密記錄。如果 Bob 的公有金鑰來自一 AWS KMS key 對，則 Bob 可以建立 AWS KMS ECDH keyring 來解密記錄。

原始 ECDH keyring 使用 AES-GCM 使用對稱金鑰加密記錄。然後，資料金鑰會使用 AES-GCM 使用衍生的共用包裝金鑰進行信封加密。每個原始 ECDH keyring 只能有一個共用包裝金鑰，但您可以在多 keyring 中單獨包含多個原始 ECDH keyring 或與其他 keyring 一起包含。

您負責產生、儲存和保護您的私有金鑰，最好是在硬體安全模組 (HSM) 或金鑰管理系統中。寄件者和收件人的金鑰對大多位於相同的橢圓曲線上。 AWS Database Encryption SDK 支援下列橢圓立方體規格：

- ECC_NIST_P256
- ECC_NIST_P384
- ECC_NIST_P512

建立原始 ECDH keyring

Raw ECDH keyring 支援三個金鑰協議結構描述：RawPrivateKeyToStaticPublicKey、EphemeralPrivateKeyToStaticPublicKey和 PublicKeyDiscovery。您選擇的金鑰協議結構描述會決定您可以執行哪些密碼編譯操作，以及如何組合金鑰材料。

主題

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

使用RawPrivateKeyToStaticPublicKey金鑰協議結構描述，在 keyring 中靜態設定寄件者的私有金鑰和收件人的公有金鑰。此金鑰協議結構描述可以加密和解密記錄。

若要使用RawPrivateKeyToStaticPublicKey金鑰協議結構描述初始化原始 ECDH keyring，請提供下列值：

- 寄件者的私有金鑰

  您必須提供寄件者的 PEM 編碼私有金鑰 (PKCS #8 PrivateKeyInfo 結構），如 RFC 5958 所定義。
- 收件人的公有金鑰

  您必須提供收件人的 DER 編碼 X.509 公有金鑰，也稱為 SubjectPublicKeyInfo(SPKI)，如 RFC 5280 所定義。

  您可以指定非對稱金鑰協議 KMS 金鑰對的公有金鑰，或從外部產生的金鑰對指定公有金鑰 AWS。
- 曲線規格

  識別指定金鑰對中的橢圓曲線規格。寄件者和收件人的金鑰對必須具有相同的曲線規格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var BobPrivateKey = new MemoryStream(new byte[] { });
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH static keyring
     var staticConfiguration = new RawEcdhStaticConfigurations()
     {
      RawPrivateKeyToStaticPublicKey = new RawPrivateKeyToStaticPublicKeyInput
      {
       SenderStaticPrivateKey = BobPrivateKey,
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = staticConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

下列 Java 範例使用`RawPrivateKeyToStaticPublicKey`金鑰協議結構描述來靜態設定寄件者的私有金鑰和收件人的公有金鑰。兩個金鑰對都在`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);
}
```

Rust

下列 Python 範例使用`raw_ecdh_static_configuration`金鑰協議結構描述來靜態設定寄件者的私有金鑰和收件人的公有金鑰。兩個金鑰對必須位於相同的曲線上。

```
// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

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

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
```

```
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

EphemeralPrivateKeyToStaticPublicKey

使用`EphemeralPrivateKeyToStaticPublicKey`金鑰協議結構描述設定的 Keyring 會在本機建立新的金鑰對，並針對每個加密呼叫衍生唯一的共用包裝金鑰。

此金鑰協議結構描述只能加密記錄。若要解密使用`EphemeralPrivateKeyToStaticPublicKey`金鑰協議結構描述加密的記錄，您必須使用以相同收件人的公有金鑰設定的探索金鑰協議結構描述。若要解密，您可以將原始 ECDH 金鑰環與`PublicKeyDiscovery`金鑰協議演算法搭配使用，或者，如果收件人的公有金鑰來自非對稱金鑰協議 KMS 金鑰對，則可以將 AWS KMS ECDH 金鑰環與 KmsPublicKeyDiscovery 金鑰協議結構描述搭配使用。

若要使用`EphemeralPrivateKeyToStaticPublicKey`金鑰協議結構描述初始化原始 ECDH keyring，請提供下列值：

- 收件人的公有金鑰

  您必須提供收件人的 DER 編碼 X.509 公有金鑰，也稱為 `SubjectPublicKeyInfo`(SPKI)，如 RFC 5280 中所定義。

  您可以指定非對稱金鑰協議 KMS 金鑰對的公有金鑰，或從外部產生的金鑰對指定公有金鑰 AWS。
- 曲線規格

  識別指定公有金鑰中的橢圓曲線規格。

  加密時， keyring 會在指定的曲線上建立新的金鑰對，並使用新的私有金鑰和指定的公有金鑰來衍生共用包裝金鑰。

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

C# / .NET

下列範例會使用`EphemeralPrivateKeyToStaticPublicKey`金鑰協議結構描述建立原始 ECDH keyring。加密時， keyring 會在指定`ECC_NIST_P256`曲線上於本機建立新的金鑰對。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH ephemeral keyring
     var ephemeralConfiguration = new RawEcdhStaticConfigurations()
     {
      EphemeralPrivateKeyToStaticPublicKey = new
 EphemeralPrivateKeyToStaticPublicKeyInput
      {
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = ephemeralConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

下列範例會使用`EphemeralPrivateKeyToStaticPublicKey`金鑰協議結構描述建立原始 ECDH keyring。加密時， keyring 會在指定`ECC_NIST_P256`曲線上於本機建立新的金鑰對。

```
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);
}
```

Rust

下列範例會使用`ephemeral_raw_ecdh_static_configuration`金鑰協議結構描述建立原始 ECDH keyring。加密時， keyring 會在指定曲線上於本機建立新的金鑰對。

```
// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

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

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

PublicKeyDiscovery

解密時，最佳實務是指定 AWS 資料庫加密 SDK 可以使用的包裝金鑰。若要遵循此最佳實務，請使用指定寄件者私有金鑰和收件人公有金鑰的 ECDH keyring。不過，您也可以建立原始 ECDH 探索 keyring，也就是原始 ECDH keyring，該 keyring 可以解密指定金鑰的公有金鑰與存放在加密記錄之材料描述欄位中的收件人公有金鑰相符的任何記錄。此金鑰協議結構描述只能解密記錄。

> **Important**
>
> 當您使用`PublicKeyDiscovery`金鑰協議結構描述解密記錄時，您接受所有公有金鑰，無論誰擁有它。

若要使用`PublicKeyDiscovery`金鑰協議結構描述初始化原始 ECDH keyring，請提供下列值：

- 收件人的靜態私有金鑰

  您必須提供收件人的 PEM 編碼私有金鑰 (PKCS #8 PrivateKeyInfo 結構），如 RFC 5958 所定義。
- 曲線規格

  識別指定私有金鑰中的橢圓曲線規格。寄件者和收件人的金鑰對必須具有相同的曲線規格。

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

C# / .NET

下列範例會使用`PublicKeyDiscovery`金鑰協議結構描述建立原始 ECDH keyring。此 keyring 可以解密指定私有金鑰的公有金鑰符合存放在加密記錄之材料描述欄位中的收件人公有金鑰的任何記錄。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePrivateKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH discovery keyring
     var discoveryConfiguration = new RawEcdhStaticConfigurations()
     {
      PublicKeyDiscovery = new PublicKeyDiscoveryInput
      {
       RecipientStaticPrivateKey = AlicePrivateKey
      }
```

```
    };

    var createKeyringInput = new CreateRawEcdhKeyringInput()
    {
     CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
     KeyAgreementScheme = discoveryConfiguration
    };

    var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

下列範例會使用`PublicKeyDiscovery`金鑰協議結構描述建立原始 ECDH keyring。此 keyring 可以解密指定私有金鑰的公有金鑰符合存放在加密記錄之材料描述欄位中的收件人公有金鑰的任何記錄。

```
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);
```

```
}
```

Rust

下列範例會使用`discovery_raw_ecdh_static_configuration`金鑰協議結構描述建立原始 ECDH keyring。此 keyring 可以解密任何訊息，其中指定的私有金鑰的公有金鑰符合存放在訊息加密文字上的收件人公有金鑰。

```
// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

多重 keyring

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

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。如果產生器 keyring 是 KMS keyring， AWS KMS 則 keyring 中的產生器金鑰會產生並加密純文字金鑰。然後， AWS KMS keyring AWS KMS keys 中的所有額外金鑰，以及 multi-keyring 中所有子 keyring 中的所有包裝金鑰，請加密相同的純文字金鑰。

解密時， AWS 資料庫加密 SDK 會使用 keyring 來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。

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 keyring 和 Raw AES keyring，但您可以在多 keyring 中結合任何支援的 keyring。

Java

```
// 1. Create 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);

// 2. Create the AWS KMS keyring.
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
// 1. Create the raw AES keyring.
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createRawAesKeyringInput = new CreateRawAesKeyringInput
{
```

```
    KeyName = "keyName",
    KeyNamespace = "myNamespaces",
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};
var rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
//    We create a MRK multi keyring, as this interface also supports
//    single-region KMS keys,
//    and creates the KMS client for us automatically.
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = keyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
// 1. Create the raw AES keyring
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// 2. Create the AWS KMS keyring
let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(key_arn)
    .send()
    .await?;
```

接著，建立多重 keyring，並指定其產生器 keyring (如果有)。在此範例中，我們會建立多金鑰環，其中 AWS KMS keyring 是產生器 keyring，AES keyring 是子 keyring。

Java

Java `CreateMultiKeyringInput`建構函式可讓您定義產生器 keyring 和子 keyring。產生的`createMultiKeyringInput`物件不可變。

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

C# / .NET

.NET `CreateMultiKeyringInput` 建構函式可讓您定義產生器 keyring 和子 keyring。產生的`CreateMultiKeyringInput`物件不可變。

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = awsKmsMrkMultiKeyring,
    ChildKeyrings = new List<IKeyring> { rawAesKeyring }
};
var multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(aws_kms_mrk_multi_keyring)
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

現在，您可以使用多重 keyring 來加密和解密資料。

可搜尋加密

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

可搜尋加密可讓您在不解密整個資料庫的情況下搜尋加密的記錄。這是使用 信標來完成的，該信標會在寫入欄位的純文字值與實際存放在資料庫中的加密值之間建立映射。 AWS Database Encryption SDK 會將信標存放在新增至記錄的新欄位中。根據您使用的信標類型，您可以對加密的資料執行完全相符的搜尋或更自訂的複雜查詢。

> **Note**
>
> AWS Database Encryption SDK 中的可搜尋加密與學術研究定義的可搜尋對稱加密不同，例如可搜尋對稱加密。

信標是截斷的雜湊型訊息驗證碼 (HMAC) 標籤，可在純文字和欄位的加密值之間建立映射。當您將新值寫入設定為可搜尋加密的加密欄位時， AWS 資料庫加密 SDK 會透過純文字值計算 HMAC。此 HMAC 輸出是該欄位純文字值的一對一 (1：1) 比對。HMAC 輸出會截斷，讓多個不同的純文字值對應至相同的截斷 HMAC 標籤。這些誤報會限制未經授權的使用者識別純文字值的辨別資訊的能力。當您查詢信標時， AWS 資料庫加密 SDK 會自動篩選掉這些誤報，並傳回查詢的純文字結果。

為每個信標產生的偽陽性平均數量取決於截斷後剩餘的信標長度。如需協助判斷實作的適當信標長度，請參閱判斷信標長度。

> **Note**
>
> 可搜尋加密旨在於新的未填入資料庫中實作。在現有資料庫中設定的任何信標只會映射上傳至資料庫的新記錄，信標無法映射現有資料。

主題

- 信標是否適合我的資料集？
- 可搜尋的加密案例

信標是否適合我的資料集？

使用信標對加密的資料執行查詢，可降低與用戶端加密資料庫相關聯的效能成本。當您使用信標時，查詢的效率與資料分佈的公開資訊量之間存在固有權衡。信標不會變更 欄位的加密狀態。當您使用 AWS 資料庫加密 SDK 加密和簽署欄位時，該欄位的純文字值永遠不會公開到資料庫。資料庫會存放 欄位的隨機加密值。

信標會與其計算來源的加密欄位一起存放。這表示即使未經授權的使用者無法檢視加密欄位的純文字值，他們仍可能能夠對信標執行統計分析，以進一步了解資料集的分佈，並在極端情況下識別信標映射到的純文字值。您設定信標的方式可以減輕這些風險。特別是，選擇正確的信標長度可協助您保護資料集的機密性。

安全性與效能

- 信標長度越短，保留的安全性就越多。
- 信標長度越長，保留的效能就越多。

可搜尋加密可能無法為所有資料集提供所需的效能和安全性層級。在設定任何信標之前，請先檢閱您的威脅模型、安全需求和效能需求。

當您判斷可搜尋加密是否適合您的資料集時，請考慮下列資料集唯一性要求。

分佈

信標保留的安全數量取決於資料集的分佈。當您設定加密欄位進行可搜尋加密時， AWS 資料庫加密 SDK 會透過寫入該欄位的純文字值計算 HMAC。針對特定欄位計算的所有信標都是使用相同的金鑰計算，但多租戶資料庫除外，每個租戶使用不同的金鑰。這表示如果多次將相同的純文字值寫入 欄位，則會針對該純文字值的每個執行個體建立相同的 HMAC 標籤。

您應該避免從包含非常常見值的欄位建構信標。例如，請考慮存放伊利諾州每個居民地址的資料庫。如果您從加密`City`欄位建構信標，由於伊利諾州人口中大部分居住在芝加哥，因此透過「芝加哥」計算的信標將過度表示。即使未經授權的使用者只能讀取加密的值和信標值，如果信標保留此分佈，他們仍可能可以識別哪些記錄包含芝加哥居民的資料。若要將分佈的區分資訊量降至最低，您必須充分截斷您的信標。隱藏此不均勻分佈所需的信標長度具有顯著的效能成本，可能不符合應用程式的需求。

您必須仔細分析資料集的分佈，以確定需要截斷多少信標。截斷後剩餘的信標長度與可辨識分佈的統計資訊量直接相關。您可能需要選擇較短的信標長度，以充分減少顯示資料集的辨別資訊量。

在極端情況下，您無法計算分佈不均勻資料集的信標長度，以有效地平衡效能和安全性。例如，您不應該從存放罕見疾病醫療測試結果的欄位建構信標。由於預期NEGATIVE結果在資料集內會明顯更普遍，因此可以透過結果的罕見程度來輕鬆識別POSITIVE結果。當欄位只有兩個可能的值時，隱藏分佈非常困難。如果您使用的信標長度夠短，足以隱藏分佈，所有純文字值都會對應至相同的 HMAC 標籤。如果您使用較長的信標長度，則很明顯哪些信標會映射到純文字POSITIVE值。

關聯性

我們強烈建議您避免從具有相關值的欄位建構不同的信標。從相關欄位建構的信標需要較短的信標長度，以充分地將每個資料集分發給未經授權的使用者的相關資訊量降至最低。您必須仔細分析資料集，包括其熵和相關值的關節分佈，以確定需要截斷多少信標。如果產生的信標長度不符合您的效能需求，則信標可能不適合您的資料集。

例如，您不應該從 `City`和 `ZIPCode` 欄位建構兩個不同的信標，因為郵遞區號可能只會與一個城市相關聯。一般而言，信標產生的誤報會限制未經授權的使用者識別資料集辨別資訊的能力。但是，`City`和 `ZIPCode` 欄位之間的關聯性表示未經授權的使用者可以輕鬆識別哪些結果是誤報，並區分不同的郵遞區號。

您也應該避免從包含相同純文字值的欄位建構信標。例如，您不應該從 `mobilePhone`和 `preferredPhone` 欄位建構信標，因為它們可能具有相同的值。如果您從這兩個欄位建構不同的信標， AWS 資料庫加密 SDK 會在不同金鑰下為每個欄位建立信標。這會導致相同純文字值的兩個不同的 HMAC 標籤。這兩個不同的信標不太可能具有相同的誤報，未經授權的使用者可能可以區分不同的電話號碼。

即使您的資料集包含相關欄位或分佈不均勻，您還是可以使用較短的信標長度來建構信標，以維護資料集的機密性。不過，信標長度不保證資料集中的每個唯一值都會產生許多誤報，以有效地將資料集的辨別資訊量降至最低。Beacon 長度僅估計產生的誤報平均數量。分佈得越不平均，則有效信標長度越低，表示判斷產生的誤報平均數量。

仔細考慮您建構信標的欄位分佈，並考慮需要多少時間來截斷信標長度，以符合您的安全需求。本章中的下列主題假設您的信標是統一分佈的，且不包含相關資料。

可搜尋的加密案例

下列範例示範了簡單的可搜尋加密解決方案。在應用程式中，此範例中使用的範例欄位可能不符合信標的分佈和關聯唯一性建議。您可以在閱讀本章中可搜尋加密概念時，使用此範例做為參考。

請考慮名為 的資料庫Employees，以追蹤公司的員工資料。資料庫中的每個記錄都包含稱為 EmployeeID、LastName、FirstName 和 Address 的欄位。Employees 資料庫中的每個欄位都由主索引鍵 識別EmployeeID。

以下是資料庫中的範例純文字記錄。

```
{
    "EmployeeID": 101,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

如果您在密碼編譯動作ENCRYPT_AND_SIGN中將 LastName和 FirstName 欄位標記為 ，這些欄位中的值會在上傳至資料庫之前在本機加密。上傳的加密資料是完全隨機的，資料庫不會將此資料識別為受保護。它只會偵測典型的資料項目。這表示實際存放在資料庫中的記錄可能如下所示。

```
{
    "PersonID": 101,
    "LastName": "1d76e94a2063578637d51371b363c9682bad926cbd",
    "FirstName": "21d6d54b0aaabc411e9f9b34b6d53aa4ef3b0a35",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

如果您需要在 LastName 欄位中查詢資料庫的完全相符項目，請設定名為 LastName 的標準信標，將寫入LastName欄位的純文字值映射至資料庫中存放的加密值。 LastName

此信標會從 LastName 欄位中的純文字值計算 HMACs。每個 HMAC 輸出都會截斷，使其不再完全符合純文字值。例如， 的完整雜湊和截斷的雜湊Jones可能如下所示。

完成雜湊

2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f

截斷雜湊

b35099d408c833

設定標準信標後，您可以在 LastName 欄位上執行等式搜尋。例如，如果您想要搜尋 Jones，請使用 LastName 信標來執行下列查詢。

```
LastName = Jones
```

AWS Database Encryption SDK 會自動篩選掉誤報，並傳回查詢的純文字結果。

信標

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

信標是截斷的雜湊型訊息驗證碼 (HMAC) 標籤，可在寫入欄位的純文字值與實際存放在資料庫中的加密值之間建立映射。信標不會變更 欄位的加密狀態。信標會透過欄位的純文字值計算 HMAC，並將其與加密值一起存放。此 HMAC 輸出是該欄位純文字值的一對一 (1：1) 比對。HMAC 輸出會截斷，讓多個不同的純文字值對應至相同的截斷 HMAC 標籤。這些誤報會限制未經授權的使用者識別純文字值的辨別資訊的能力。

信標只能從密碼編譯動作SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT中標記為 SIGN_ONLY、ENCRYPT_AND_SIGN或 的欄位建構。信標本身不會簽署或加密。您無法以標記為 的欄位建構信標DO_NOTHING。

您設定的信標類型決定您可以執行的查詢類型。支援可搜尋加密的信標有兩種類型。標準信標會執行相等性搜尋。複合信標結合純文字字串和標準信標，以執行複雜的資料庫操作。設定信標之後，您必須先為每個信標設定次要索引，才能搜尋加密的欄位。如需詳細資訊，請參閱使用信標設定次要索引。

主題

- 標準信標
- 複合信標

標準信標

標準信標是在資料庫中實作可搜尋加密的最簡單方法。他們只能對單一加密或虛擬欄位執行平等搜尋。若要了解如何設定標準信標，請參閱設定標準信標。

標準信標建構來源的欄位稱為信標來源。它可識別信標需要映射的資料位置。信標來源可以是加密欄位或虛擬欄位。每個標準信標中的信標來源必須是唯一的。您無法設定具有相同信標來源的兩個信標。

標準信標可用於對加密或虛擬欄位執行平等搜尋。或者，它們可用來建構複合信標，以執行更複雜的資料庫操作。為了協助您組織和管理標準信標， AWS 資料庫加密 SDK 提供下列選用信標樣式，可定義標準信標的預期用途。如需詳細資訊，請參閱定義信標樣式。

您可以建立執行單一加密欄位平等搜尋的標準信標，也可以建立虛擬欄位，建立在多個 `ENCRYPT_AND_SIGN`、 `SIGN_ONLY`和 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 欄位串連上執行平等搜尋的標準信標。

虛擬欄位

虛擬欄位是從一或多個來源欄位建構的概念欄位。建立虛擬欄位不會將新欄位寫入您的記錄。虛擬欄位不會明確存放在您的資料庫中。它用於標準信標組態，提供信標指示，說明如何識別欄位的特定區段，或串連記錄中的多個欄位來執行特定查詢。虛擬欄位至少需要一個加密的欄位。

> **Note**
>
> 下列範例示範您可以使用虛擬欄位執行的轉換和查詢類型。在應用程式中，此範例中使用的範例欄位可能不符合信標的分佈和關聯唯一性建議。

例如，如果您想要對 `FirstName`和 `LastName` 欄位的串連執行平等搜尋，您可以建立下列其中一個虛擬欄位。

- 虛擬`NameTag`欄位，從`FirstName`欄位的第一個字母開始建構，後面接著 `LastName` 欄位，全部小寫。此虛擬欄位可讓您查詢 `NameTag=mjones`。
- 虛擬`LastFirst`欄位，由 `LastName` 欄位建構，後面接著 `FirstName` 欄位。此虛擬欄位可讓您查詢 `LastFirst=JonesMary`。

或者，如果您想要對加密欄位的特定區段執行等式搜尋，請建立虛擬欄位來識別您要查詢的區段。

例如，如果您想要使用 IP 地址的前三個區段來查詢加密`IPAddress`的欄位，請建立下列虛擬欄位。

- 虛擬`IPSegment`欄位，由 建構`Segments('.', 0, 3)`。此虛擬欄位可讓您查詢`IPSegment=192.0.2`。查詢會傳回`IPAddress`值開頭為 "192.0.2" 的所有記錄。

虛擬欄位必須是唯一的。兩個虛擬欄位無法從完全相同的來源欄位建構。

如需設定虛擬欄位和使用它們的信標的說明，請參閱建立虛擬欄位。

複合信標

複合信標會建立索引來改善查詢效能，並可讓您執行更複雜的資料庫操作。您可以使用複合信標來結合文字純文字字串和標準信標，對加密的記錄執行複雜的查詢，例如從單一索引查詢兩種不同的記錄類型，或使用排序索引鍵查詢欄位組合。如需更多複合信標解決方案範例，請參閱選擇信標類型。

複合信標可以從標準信標或標準信標和已簽章欄位的組合建構。它們是從組件清單建構而成。所有複合信標都應包含加密部分清單，以識別信標中包含`ENCRYPT_AND_SIGN`的欄位。每個`ENCRYPT_AND_SIGN`欄位都必須由標準信標識別。更複雜的複合信標也可能包含識別信標中包含純文字`SIGN_ONLY`或`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`欄位的已簽章部分清單，以及識別複合信標組合欄位的所有可能方式的建構器部分清單。

> **Note**
>
> AWS Database Encryption SDK 也支援簽署的信標，這些信標可以完全從純文字`SIGN_ONLY`和`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`欄位設定。簽署的信標是一種複合信標，在簽署但未加密的欄位上編製索引並執行複雜的查詢。如需詳細資訊，請參閱建立簽章的信標。

如需設定複合信標的說明，請參閱設定複合信標。

您設定複合信標的方式決定其可執行的查詢類型。例如，您可以選用一些加密和已簽章的組件，以便在查詢中提供更多彈性。如需複合信標可執行之查詢類型的詳細資訊，請參閱查詢信標。

規劃信標

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

信標旨在於新的未填入資料庫中實作。在現有資料庫中設定的任何信標只會映射寫入資料庫的新記錄。信標是根據欄位的純文字值計算，一旦欄位加密，信標就無法映射現有資料。使用信標寫入新記錄之後，您就無法更新信標的組態。不過，您可以為新增至記錄的新欄位新增新信標。

若要實作可搜尋加密，您必須使用AWS KMS 階層式 keyring 來產生、加密和解密用於保護記錄的資料金鑰。如需詳細資訊，請參閱使用階層式 keyring 進行可搜尋加密。

您必須先檢閱加密需求、資料庫存取模式和威脅模型，以判斷資料庫的最佳解決方案，才能設定可搜尋加密的信標。

您設定的信標類型決定您可以執行的查詢類型。您在標準信標組態中指定的信標長度決定了指定信標產生的預期誤報數量。我們強烈建議您識別和規劃在設定信標之前需要執行的查詢類型。使用信標後，就無法更新組態。

強烈建議您在設定任何信標之前，先檢閱並完成下列任務。

- 判斷信標是否適合您的資料集
- 選擇信標類型
- 選擇信標長度
- 選擇信標名稱

當您為資料庫規劃可搜尋的加密解決方案時，請記住下列指標唯一性要求。

- 每個標準信標都必須具有唯一的信標來源

  多個標準信標無法從相同的加密或虛擬欄位建構。

  不過，單一標準信標可用來建構多個複合信標。
- 避免使用與現有標準信標重疊的來源欄位建立虛擬欄位

  從虛擬欄位建構標準信標，其中包含用於建立另一個標準信標的來源欄位，可以降低兩個信標的安全性。

  如需詳細資訊，請參閱虛擬欄位的安全考量。

多租戶資料庫的考量事項

若要查詢在多租用戶資料庫中設定的信標，您必須包含與在查詢中加密記錄的租用戶branch-key-id相關聯的 儲存欄位。當您定義信標金鑰來源時，可以定義此欄位。若要讓查詢成功，此欄位中的值必須識別重新計算信標所需的適當信標金鑰材料。

設定信標之前，您必須決定計劃如何在查詢branch-key-id中包含 。如需您可以在查詢branch-key-id中包含 之不同方式的詳細資訊，請參閱 查詢多租戶資料庫中的信標。

選擇信標類型

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

使用可搜尋加密，您可以透過將加密欄位中的純文字值與信標映射來搜尋加密的記錄。您設定的信標類型決定您可以執行的查詢類型。

我們強烈建議您識別和規劃在設定信標之前需要執行的查詢類型。設定信標之後，您必須先為每個信標設定次要索引，才能搜尋加密的欄位。如需詳細資訊，請參閱使用信標設定次要索引。

信標會在寫入欄位的純文字值與實際存放在資料庫中的加密值之間建立映射。您無法比較兩個標準信標的值，即使它們包含相同的基礎純文字。兩個標準信標會為相同的純文字值產生兩個不同的 HMAC 標籤。因此，標準信標無法執行下列查詢。

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

只有在您比較複合信標的簽章部分時，才能執行上述查詢，但CONTAINS運算子除外，您可以搭配複合信標使用，以識別組合信標包含的加密或簽章欄位的完整值。當您比較已簽章的組件時，您可以選擇包含加密組件的字首，但不能包含欄位的加密值。如需標準和複合信標可執行之查詢類型的詳細資訊，請參閱查詢信標。

檢閱資料庫存取模式時，請考慮下列可搜尋的加密解決方案。下列範例定義要設定哪些信標以滿足不同的加密和查詢要求。

標準信標

標準信標只能執行等式搜尋。您可以使用標準信標來執行下列查詢。

查詢單一加密欄位

如果您想要識別包含加密欄位特定值的記錄，請建立標準信標。

範例

針對下列範例，請考慮名為 的資料庫UnitInspection，以追蹤生產設施的檢查資料。資料庫中的每個記錄都包含稱為 work_id、inspector_id_last4、 inspection_date和 的欄位unit。完整的檢查器 ID 是介於 0 到 99，999，999 之間的數字。不過，為了確保資料集均勻分佈， inspector_id_last4只會存放檢查器 ID 的最後四位數字。資料庫中的每個欄位都由主索引鍵識別work_id。密碼編譯動作ENCRYPT_AND_SIGN中會標記 inspector_id_last4和 unit 欄位。

以下是UnitInspection資料庫中純文字項目的範例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

查詢記錄中的單一加密欄位

> 如果inspector_id_last4欄位需要加密，但您仍然需要查詢其確切相符項目，請從 inspector_id_last4 欄位建構標準信標。然後，使用標準信標來建立次要索引。您可以使用此次要索引來查詢加密inspector_id_last4欄位。

如需設定標準信標的說明，請參閱設定標準信標。

查詢虛擬欄位

虛擬欄位是從一或多個來源欄位建構的概念欄位。如果您想要對加密欄位的特定區段執行等式搜尋，或對多個欄位的串連執行等式搜尋，請從虛擬欄位建構標準信標。所有虛擬欄位必須至少包含一個加密的來源欄位。

範例

下列範例會建立Employees資料庫的虛擬欄位。以下是Employees資料庫中的範例純文字記錄。

```
{
    "EmployeeID": 101,
    "SSN": 000-00-0000,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

查詢加密欄位的區段

在此範例中， SSN 欄位會加密。

如果您想要使用社會安全號碼的後四位數字查詢SSN欄位，請建立虛擬欄位來識別您計劃查詢的客群。

由 建構的虛擬Last4SSN欄位Suffix(4)可讓您查詢 Last4SSN=0000。使用此虛擬欄位來建構標準信標。然後，使用標準信標來建立次要索引。您可以使用此次要索引來查詢虛擬欄位。此查詢會傳回SSN值結尾為您指定後四碼的所有記錄。

查詢多個欄位的串連

> **Note**
>
> 下列範例示範您可以使用虛擬欄位執行的轉換和查詢類型。在應用程式中，此範例中使用的範例欄位可能不符合信標的分佈和關聯唯一性建議。

如果您想要對 FirstName和 LastName 欄位的串連執行等式搜尋，您可以建立虛擬NameTag欄位，該欄位是從FirstName欄位的第一個字母開始建構，後面接著 LastName 欄位，全部以小寫字母表示。使用此虛擬欄位來建構標準信標。然後，使用標準信標來建立次要索引。您可以使用此次要索引來NameTag=mjones查詢虛擬欄位。

至少必須加密其中一個來源欄位。FirstName 或 LastName可以加密，或兩者都可以加密。任何純文字來源欄位都必須在您的密碼編譯動作SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT中標記為 SIGN_ONLY或 。

如需設定虛擬欄位和使用它們的信標的說明，請參閱建立虛擬欄位。

複合信標

複合信標會從文字純文字字串和標準信標建立索引，以執行複雜的資料庫操作。您可以使用複合信標來執行下列查詢。

查詢單一索引上加密欄位的組合

如果您需要查詢單一索引上加密欄位的組合，請建立複合信標，該信標會結合為每個加密欄位建構的個別標準信標，以形成單一索引。

設定複合信標之後，您可以建立次要索引，指定複合信標做為分割區索引鍵來執行完全相符的查詢，或使用排序索引鍵來執行更複雜的查詢。將複合信標指定為排序索引鍵的次要索引可執行完全相符的查詢和更自訂的複雜查詢。

範例

針對下列範例，請考慮名為 的資料庫`UnitInspection`，以追蹤生產設施的檢查資料。資料庫中的每個記錄都包含稱為 `work_id`、`inspector_id_last4`、 `inspection_date`和 的欄位`unit`。完整的檢查器 ID 是介於 0 到 99，999，999 之間的數字。不過，為了確保資料集均勻分佈， `inspector_id_last4`只會存放檢查器 ID 的最後四位數字。資料庫中的每個欄位都由主索引鍵識別`work_id`。密碼編譯動作`ENCRYPT_AND_SIGN`中會標記 `inspector_id_last4`和 `unit` 欄位。

以下是`UnitInspection`資料庫中純文字項目的範例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

在加密欄位的組合上執行平等搜尋

如果您想要在 上查詢`UnitInspection`資料庫是否完全相符`inspector_id_last4.unit`，請先為 `inspector_id_last4`和 `unit` 欄位建立不同的標準信標。然後，從兩個標準信標建立複合信標。

設定複合信標之後，請建立次要索引，指定複合信標做為分割區索引鍵。使用此次要索引來查詢 上的完全相符項目`inspector_id_last4.unit`。例如，您可以查詢此信標，以尋找檢查器為指定單位執行的檢查清單。

在加密欄位的組合上執行複雜的查詢

如果您想要查詢 inspector_id_last4和 上的UnitInspection資料庫inspector_id_last4.unit，請先為 inspector_id_last4和 unit 欄位建立不同的標準信標。然後，從兩個標準信標建立複合信標。

設定複合信標之後，請建立次要索引，指定複合信標做為排序索引鍵。使用此次要索引來查詢UnitInspection資料庫以特定檢查器開頭的項目，或查詢資料庫以取得特定檢查器檢查之特定單位 ID 範圍內所有單位的清單。您也可以在 上執行完全相符的搜尋inspector_id_last4.unit。

如需設定複合信標的說明，請參閱設定複合信標。

查詢單一索引上加密和純文字欄位的組合

如果您需要查詢單一索引上加密和純文字欄位的組合，請建立複合信標，結合個別標準信標和純文字欄位以形成單一索引。用於建構複合信標的純文字欄位必須在密碼編譯動作SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT中標記 SIGN_ONLY或 。

設定複合信標之後，您可以建立次要索引，指定複合信標做為分割區索引鍵來執行完全相符的查詢，或使用排序索引鍵來執行更複雜的查詢。將複合信標指定為排序索引鍵的次要索引可執行完全相符的查詢和更自訂的複雜查詢。

範例

針對下列範例，請考慮名為 的資料庫UnitInspection，以追蹤生產設施的檢查資料。資料庫中的每個記錄都包含稱為 work_id、inspector_id_last4、 inspection_date和 的欄位unit。完整的檢查器 ID 是介於 0 到 99，999，999 之間的數字。不過，為了確保資料集均勻分佈， inspector_id_last4只會存放檢查器 ID 的最後四位數字。資料庫中的每個欄位都由主索引鍵識別work_id。密碼編譯動作ENCRYPT_AND_SIGN中會標記 inspector_id_last4和 unit 欄位。

以下是UnitInspection資料庫中純文字項目的範例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

在欄位組合上執行等式搜尋

如果您想要查詢UnitInspection資料庫，以了解特定檢查人員在特定日期執行的檢查，請先為 inspector_id_last4 欄位建立標準信標。inspector_id_last4 欄位會在密碼編譯動作ENCRYPT_AND_SIGN中標記。所有加密的組件都需要自己的標準信標。inspection_date 欄位已標記SIGN_ONLY，不需要標準信標。接著，從 inspection_date 欄位和inspector_id_last4標準信標建立複合信標。

設定複合信標之後，請建立次要索引，指定複合信標做為分割區索引鍵。使用此次要索引查詢資料庫，尋找與特定檢查器和檢查日期完全相符的記錄。例如，您可以查詢資料庫，以取得 ID 結尾為的檢查人員在特定日期8744執行的所有檢查清單。

在欄位組合上執行複雜的查詢

如果您想要查詢資料庫是否有在 inspection_date範圍內執行的檢查，或查詢資料庫是否有對 inspector_id_last4或 所inspection_date限制的特定 執行的檢查inspector_id_last4.unit，請先為 inspector_id_last4和 unit 欄位建立不同的標準信標。然後，從純文字inspection_date欄位和兩個標準信標建立複合信標。

設定複合信標之後，請建立次要索引，指定複合信標做為排序索引鍵。使用此次要索引，對特定檢查器在特定日期執行的檢查執行查詢。例如，您可以查詢資料庫，以取得在相同日期檢查的所有單位清單。或者，您可以查詢資料庫，以取得指定檢查日期範圍內在特定單位上執行的所有檢查清單。

如需設定複合信標的說明，請參閱設定複合信標。

選擇信標長度

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

當您將新值寫入設定為可搜尋加密的加密欄位時， AWS 資料庫加密 SDK 會透過純文字值計算 HMAC。此 HMAC 輸出是該欄位純文字值的一對一 (1：1) 比對。HMAC 輸出會截斷，讓多個不同的純文字值對應至相同的截斷 HMAC 標籤。這些碰撞或誤報會限制未經授權的使用者識別純文字值的辨別資訊的能力。

為每個信標產生的偽陽性平均數量取決於截斷後剩餘的信標長度。您只需要在設定標準信標時定義信標長度。複合信標會使用其建構之標準信標的信標長度。

信標不會變更 欄位的加密狀態。不過，當您使用信標時，查詢的效率與資料分佈的公開資訊量之間存在固有權衡。

可搜尋加密的目標是使用信標對加密的資料執行查詢，以降低與用戶端加密資料庫相關的效能成本。信標會與其計算來源的加密欄位一起存放。這表示他們可以顯示有關資料集分佈的辨別資訊。在極端情況下，未經授權的使用者可能可以分析所揭露的分佈相關資訊，並使用它來識別欄位的純文字值。選擇正確的信標長度有助於降低這些風險，並維護分發的機密性。

檢閱您的威脅模型，以判斷您需要的安全層級。例如，擁有資料庫存取權但不應該存取純文字資料的人員越多，您可能想要保護資料集分佈的機密性就越多。為了提高機密性，信標需要產生更多誤報。提高機密性會導致查詢效能降低。

安全性與效能

- 過長的信標長度會產生太少的誤報，並可能顯示有關資料集分佈的辨別資訊。
- 過短的信標長度會產生太多誤報，並提高查詢的效能成本，因為它需要更廣泛的資料庫掃描。

判斷解決方案的適當信標長度時，您必須找到適當保留資料安全性的長度，而不會影響查詢的效能，這絕對必要。信標保留的安全數量取決於資料集的分佈，以及信標建構來源欄位的關聯性。下列主題假設您的信標是統一分佈的，且不包含相關資料。

主題

- 計算信標長度
- 範例

計算信標長度

信標長度以位元定義，是指截斷後保留的 HMAC 標籤位元數。建議的信標長度取決於資料集分佈、相關值的存在，以及您的特定安全和效能需求。如果您的資料集是均勻分佈的，您可以使用下列方程式和程序來協助識別實作的最佳信標長度。這些方程式只會估計信標會產生的平均誤報次數，並不保證資料集中的每個唯一值都會產生特定數量的誤報。

> **Note**
>
> 這些方程式的有效性取決於資料集的分佈。如果您的資料集未統一分佈，請參閱 信標是否適合我的資料集？。
> 一般而言，您的資料集越來自統一分佈，縮短信標長度所需的數量就越多。

1.

估計人口

人口是標準信標建構來源欄位中唯一值的預期數量，不是欄位中存放值的預期總數。例如，請考慮可識別員工會議位置的加密Room欄位。Room 欄位預計總共會存放 100，000 個值，但只有 50 個不同的會議室可供員工預留用於會議。這表示人口為 50，因為只有 50 個可能的唯一值可以存放在 Room 欄位中。

> **Note**
>
> 如果您的標準信標是從虛擬欄位建構，則用於計算信標長度的人口是虛擬欄位建立的唯一組合數目。

估算人口時，請務必考慮資料集的預測增長。使用信標寫入新記錄後，您就無法更新信標長度。檢閱您的威脅模型和任何現有的資料庫解決方案，以建立您預期此欄位在未來五年內存放的唯一值數量的預估值。

您的人口不需要精確。首先，識別目前資料庫中的唯一值數目，或估算您預期在第一年中存放的唯一值數目。接下來，使用下列問題來協助您判斷未來五年內唯一值的預測增長。

- 您是否預期唯一值會乘以 10？
- 您是否預期唯一值會乘以 100？
- 您是否預期唯一值會乘以 1000？

50，000 和 60，000 個唯一值之間的差異並不顯著，它們都會產生相同的建議信標長度。不過，50，000 和 500，000 個唯一值之間的差異將顯著影響建議的信標長度。

請考慮檢閱常見資料類型頻率的公有資料，例如郵遞區號或姓氏。例如，美國有 41，707 個郵遞區號。您使用的人口應與您自己的資料庫成比例。如果資料庫中`ZIPCode`的欄位包含來自美國各地的資料，則即使`ZIPCode`欄位目前沒有 41，707 個唯一值，您也可以將人口定義為 41，707 個。如果資料庫中`ZIPCode`的欄位只包含來自單一狀態的資料，而且只包含來自單一狀態的資料，則您可以將人口定義為該狀態的郵遞區號總數，而不是 41，704。

2. 計算預期碰撞次數的建議範圍

若要判斷指定欄位的適當信標長度，您必須先為預期的碰撞數量識別適當的範圍。預期的碰撞次數代表映射到特定 HMAC 標籤的唯一純文字值的平均預期次數。一個唯一純文字值的預期誤報數量小於預期的碰撞數量。

我們建議預期的碰撞數量大於或等於兩個，且小於人口的平方根。下列方程式只有在您的人口有 16 個或更多唯一值時才有效。

```
2 ≤ number of collisions < √(Population)
```

如果碰撞次數少於兩個，則信標會產生太多誤報。我們建議使用兩個 作為預期碰撞的最小數量，因為它平均表示欄位中的每個唯一值都會映射到另一個唯一值來產生至少一個誤報。

3. 計算信標長度的建議範圍

識別預期碰撞的最小和最大數量後，請使用下列方程式來識別適當的信標長度範圍。

$$\text{number of collisions} = \text{Population} * 2^{-(\text{beacon length})}$$

首先，解決預期碰撞數量等於兩個（建議的最低預期碰撞數量）的信標長度。

$$2 = \text{Population} * 2^{-(\text{beacon length})}$$

然後，解決預期碰撞數量等於人口平方根的信標長度（建議的最大預期碰撞數量）。

$$\sqrt{(\text{Population})} = \text{Population} * 2^{-(\text{beacon length})}$$

我們建議將此方程式產生的輸出四捨五入到較短的信標長度。例如，如果方程式產生 15.6 的信標長度，我們建議將該值四捨五入到 15 位元，而不是四捨五入到 16 位元。

4. 選擇信標長度

這些方程式只會識別您欄位的建議信標長度範圍。我們建議您盡可能使用較短的信標長度來維護資料集的安全性。不過，您實際使用的信標長度取決於您的威脅模型。在檢閱威脅模型時，請考慮您的效能需求，以判斷欄位的最佳信標長度。

使用較短的信標長度會降低查詢效能，而使用較長的信標長度則會降低安全性。一般而言，如果您的資料集分佈不平均，或者如果您從關聯欄位建構不同的信標，則需要使用較短的信標長度，以將資料集分佈的相關資訊量降至最低。

如果您檢閱威脅模型，並決定任何顯示欄位分佈的區別資訊不會對您的整體安全造成威脅，您可以選擇使用比您計算的建議範圍更長的信標長度。例如，如果您將欄位的信標長度建議範圍計算為 9-16 位元，您可以選擇使用 24 位元的信標長度，以避免任何效能損失。

請謹慎選擇您的信標長度。使用信標寫入新記錄後，您就無法更新信標長度。

範例

考慮在密碼編譯動作ENCRYPT_AND_SIGN中將 `unit` 欄位標記為 的資料庫。若要設定 `unit` 欄位的標準信標，我們需要判斷該`unit`欄位的預期誤報數和信標長度。

1. 估計人口

   檢閱我們的威脅模型和目前的資料庫解決方案後，我們預期 `unit` 欄位最終會有 100，000 個唯一值。

   這表示總體 = 100，000。

2. 計算預期碰撞次數的建議範圍。

   在此範例中，預期的碰撞次數應介於 2 到 316 之間。

   ```
   2 ≤ number of collisions < √(Population)
   ```

   a.
   ```
   2 ≤ number of collisions < √(100,000)
   ```

   b.
   ```
   2 ≤ number of collisions < 316
   ```

3. 計算信標長度的建議範圍。

   在此範例中，信標長度應介於 9-16 位元之間。

   ```
   number of collisions = Population * 2^-(beacon length)
   ```

   a. 計算信標長度，其中預期的碰撞數量等於步驟 2 中識別的最小值。

      ```
      2 = 100,000 * 2^-(beacon length)
      ```

      信標長度 = 15.6 或 15 位元

b. 計算預期的碰撞次數等於步驟 2 中識別的最大值的信標長度。

```
316 = 100,000 * 2^-(beacon length)
```

信標長度 = 8.3 或 8 位元

4. 判斷符合您安全和效能需求的信標長度。

對於低於 15 的每個位元，效能成本和安全性會加倍。

- 16 位元
  - 平均而言，每個唯一值都會對應至其他 1.5 個單位。
  - 安全性：具有相同截斷 HMAC 標籤的兩個記錄有 66% 可能具有相同的純文字值。
  - 效能：查詢將擷取您實際請求的每 10 筆記錄的 15 筆記錄。
- 14 位元
  - 平均而言，每個唯一值都會對應至其他 6.1 個單位。
  - 安全：具有相同截斷 HMAC 標籤的兩個記錄有 33% 可能具有相同的純文字值。
  - 效能：查詢將擷取您實際請求之每 10 筆記錄的 30 筆記錄。

選擇信標名稱

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

每個信標都由唯一的信標名稱識別。設定信標後，信標名稱即為您在查詢加密欄位時使用的名稱。信標名稱可與加密欄位或虛擬欄位的名稱相同，但不能與未加密欄位的名稱相同。兩個不同的信標不能有相同的信標名稱。

如需示範如何命名和設定信標的範例，請參閱設定信標。

命名標準信標

命名標準信標時，強烈建議您的信標名稱盡可能解析為信標來源。這表示您標準信標建構來源的加密或虛擬欄位的信標名稱和名稱相同。例如，如果您要為名為 的加密欄位建立標準信標`LastName`，您的信標名稱也應該是 `LastName`。

當您的信標名稱與信標來源相同時，您可以從組態中省略信標來源，資料庫 AWS 加密 SDK 會自動使用信標名稱做為信標來源。

設定信標

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

支援可搜尋加密的信標有兩種類型。標準信標會執行相等性搜尋。它們是在資料庫中實作可搜尋加密的最簡單方法。複合信標結合純文字字串和標準信標，以執行更複雜的查詢。

信標旨在於新的未填入資料庫中實作。在現有資料庫中設定的任何信標只會映射寫入資料庫的新記錄。信標是根據欄位的純文字值計算，一旦欄位加密，信標就無法映射現有資料。使用信標寫入新記錄後，您就無法更新信標的組態。不過，您可以為新增至記錄的新欄位新增新信標。

確定您的存取模式後，設定信標應該是資料庫實作的第二個步驟。然後，在設定所有信標之後，您需要建立AWS KMS 階層式 keyring、定義信標版本、為每個信標設定次要索引、定義密碼編譯動作，以及設定資料庫和 AWS 資料庫加密 SDK 用戶端。如需詳細資訊，請參閱使用信標。

為了更輕鬆地定義信標版本，我們建議您建立標準信標和複合信標的清單。設定每個您建立的信標時，將它們新增至各自的標準或複合信標清單。

主題

- 設定標準信標
- 設定複合信標
- 範例組態

設定標準信標

標準信標是在資料庫中實作可搜尋加密的最簡單方法。他們只能對單一加密或虛擬欄位執行平等搜尋。

組態語法範例

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
```

```
    .name("beaconName")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>();
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "beaconName",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let standard_beacon_list = vec![

 StandardBeacon::builder().name("beacon_name").length(beacon_length_in_bits).build()?,
```

若要設定標準信標，請提供下列值。

指標名稱

您在查詢加密欄位時使用的名稱。

信標名稱可與加密欄位或虛擬欄位的名稱相同，但不能與未加密欄位的名稱相同。我們強烈建議盡可能使用標準信標建構的加密欄位或虛擬欄位的名稱。兩個不同的信標不能有相同的信標名稱。如需判斷實作最佳信標名稱的協助，請參閱選擇信標名稱。

信標長度

截斷後保留的信標雜湊值位元數。

信標長度會決定指定信標產生的平均誤報數。如需詳細資訊並協助判斷實作的適當信標長度，請參閱判斷信標長度。

指標來源（選用）

標準信標的建構來源欄位。

信標來源必須是欄位名稱或參考巢狀欄位值的索引。當您的信標名稱與信標來源相同時，您可以從組態省略信標來源，而 AWS 資料庫加密 SDK 會自動使用信標名稱做為信標來源。

建立虛擬欄位

若要建立虛擬欄位，您必須提供虛擬欄位的名稱和來源欄位清單。您新增來源欄位至虛擬零件清單的順序，會決定它們串連以建置虛擬欄位的順序。下列範例會串連兩個來源欄位，以建立虛擬欄位。

> **Note**
>
> 我們建議您在填入資料庫之前，先驗證您的虛擬欄位是否產生預期結果。如需詳細資訊，請參閱測試信標輸出。

Java

請參閱完整的程式碼範例：VirtualBeaconSearchableEncryptionExample.java

```
List<VirtualPart> virtualPartList = new ArrayList<>();
    virtualPartList.add(sourceField1);
    virtualPartList.add(sourceField2);

VirtualField virtualFieldName = VirtualField.builder()
    .name("virtualFieldName")
    .parts(virtualPartList)
    .build();

List<VirtualField> virtualFieldList = new ArrayList<>();
    virtualFieldList.add(virtualFieldName);
```

C# / .NET

請參閱完整的程式碼範例：VirtualBeaconSearchableEncryptionExample.cs

```
var virtualPartList = new List<VirtualPart> { sourceField1, sourceField2 };

var virtualFieldName = new VirtualField
{
    Name = "virtualFieldName",
    Parts = virtualPartList
};
```

```
var virtualFieldList = new List<VirtualField> { virtualFieldName };
```

Rust

請參閱完整的程式碼範例： virtual_beacon_searchable_encryption.rs

```
let virtual_part_list = vec![source_field_one, source_field_two];

let state_and_has_test_result_field = VirtualField::builder()
    .name("virtual_field_name")
    .parts(virtual_part_list)
    .build()?;

let virtual_field_list = vec![virtual_field_name];
```

若要使用來源欄位的特定區段建立虛擬欄位，您必須先定義轉換，才能將來源欄位新增至虛擬部分清單。

虛擬欄位的安全考量

信標不會變更 欄位的加密狀態。不過，當您使用信標時，查詢的效率與資料分佈的公開資訊量之間存在固有權衡。您設定信標的方式會決定該信標所保留的安全層級。

避免使用與現有標準信標重疊的來源欄位建立虛擬欄位。建立包含已用於建立標準信標之來源欄位的虛擬欄位，可能會降低兩個信標的安全層級。安全性降低的程度取決於其他來源欄位新增的熵層級。熵程度取決於其他來源欄位中唯一值的分佈，以及其他來源欄位對虛擬欄位整體大小的貢獻位元數。

您可以使用人口和信標長度來判斷虛擬欄位的來源欄位是否保留資料集的安全性。人口是欄位中唯一值的預期數量。您的人口不需要精確。如需估算欄位人口的說明，請參閱估算人口。

檢閱虛擬欄位的安全性時，請考慮下列範例。

- Beacon1 由 建構`FieldA`。 `FieldA` 的人口大於 $2^{(\text{Beacon1 長度})}$。
- Beacon2 由 建構`VirtualField`，其由 `FieldA`、`FieldC`、 `FieldB`和 建構`FieldD`。、`FieldBFieldC`、 和 `FieldD`的人口總和大於 2^N

如果下列陳述式為 true，Beacon2 會同時保留 Beacon1 和 Beacon2 的安全性：

```
N ≥ (Beacon1 length)/2
```

以及

```
N ≥ (Beacon2 length)/2
```

定義信標樣式

標準信標可用於對加密或虛擬欄位執行平等搜尋。或者，它們可用來建構複合信標，以執行更複雜的資料庫操作。為了協助您組織和管理標準信標， AWS 資料庫加密 SDK 提供下列選用信標樣式，可定義標準信標的預期用途。

> **Note**
>
> 若要定義信標樣式，您必須使用 3.2 版或更新版本的 AWS 資料庫加密 SDK。將信標樣式新增至信標組態之前，先將新版本部署至所有讀取器。

PartOnly

定義為 的標準信標`PartOnly`只能用來定義複合信標的加密部分。您無法直接查詢`PartOnly`標準信標。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .partOnly(PartOnly.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
```

```
    Style = new BeaconStyle
    {
        PartOnly = new PartOnly()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::PartOnly(PartOnly::builder().build()?))
    .build()?
```

Shared

根據預設，每個標準信標都會產生唯一的 HMAC 金鑰，以進行信標計算。因此，您無法從兩個單獨的標準信標對加密的欄位執行等式搜尋。定義為 的標準信標Shared會使用來自另一個標準信標的 HMAC 金鑰進行計算。

例如，如果您需要將beacon1欄位與beacon2欄位進行比較，請將 beacon2定義為使用來自 的 HMAC 金鑰beacon1進行計算的Shared信標。

> **Note**
>
> 在設定任何Shared信標之前，請考慮您的安全和效能需求。 Shared信標可能會增加可識別的有關資料集分佈的統計資訊量。例如，它們可能會顯示哪些共用欄位包含相同的純文字值。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .shared(Shared.builder().other("beacon1").build())
        .build()
```

```
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        Shared = new Shared { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::Shared(
       Shared::builder().other("beacon1").build()?,
    ))
    .build()?
```

AsSet

根據預設，如果欄位值為集合， AWS 資料庫加密 SDK 會計算集合的單一標準信標。因此，您無法執行 `CONTAINS(a, :value)` `a` 為加密欄位的查詢。定義為 的標準信標會`AsSet`計算集合中每個個別元素的個別標準信標值，並將信標值以集合形式存放在項目中。這可讓 AWS Database Encryption SDK 執行查詢 `CONTAINS(a, :value)`。

若要定義`AsSet`標準信標，集合中的元素必須來自相同的人口，以便它們都可以使用相同的信標長度。如果在計算信標值時發生衝突，信標集的元素可能比純文字集少。

> **Note**
>
> 在設定任何`AsSet`信標之前，請考慮您的安全和效能需求。 `AsSet`信標可能會增加可識別的有關資料集分佈的統計資訊量。例如，它們可能會顯示純文字集的大小。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .asSet(AsSet.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        AsSet = new AsSet()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::AsSet(AsSet::builder().build()?))
    .build()?
```

SharedSet

定義為 的標準信標`SharedSet`結合 `Shared`和 `AsSet`函數，讓您可以對集合和欄位的加密值執行相等性搜尋。這可讓 AWS Database Encryption SDK 執行查詢，`CONTAINS(a, b)`其中 *a* 是加密集，而 *b*是加密欄位。

> **Note**
>
> 在設定任何Shared信標之前，請考慮您的安全和效能需求。 SharedSet信標可能會增加可識別的有關資料集分佈的統計資訊量。例如，它們可能會顯示純文字集的大小，或哪些共用欄位包含相同的純文字值。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .sharedSet(SharedSet.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        SharedSet = new SharedSet { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::SharedSet(
        SharedSet::builder().other("beacon1").build()?,
    ))
```

```
    .build()?
```

設定複合信標

複合信標結合純文字字串和標準信標來執行複雜的資料庫操作，例如從單一索引查詢兩種不同的記錄類型，或查詢欄位與排序索引鍵的組合。複合信標可以從 ENCRYPT_AND_SIGN、 SIGN_ONLY和 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 欄位建構。您必須為複合信標中包含的每個加密欄位建立標準信標。

> **Note**
>
> 我們建議您在填入資料庫之前，先驗證您的複合信標是否產生預期結果。如需詳細資訊，請參閱測試信標輸出。

組態語法範例

Java

複合信標組態

下列範例會在 複合信標組態中於本機定義加密和簽章的組件清單。

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .encrypted(encryptedPartList)
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

指標版本定義

下列範例會在信標版本中全域定義加密和簽章的組件清單。如需定義信標版本的詳細資訊，請參閱使用信標。

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
```

```
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

請參閱完整的程式碼範例：BeaconConfig.cs

複合信標組態

下列範例會在 複合信標組態中於本機定義加密和簽章的組件清單。

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Encrypted = encryptedPartList,
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

指標版本定義

下列範例會在信標版本中全域定義加密和簽章的組件清單。如需定義信標版本的詳細資訊，請參閱使用信標。

```
var beaconVersions = new List<BeaconVersion>
{
```

```
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

請參閱完整的程式碼範例： beacon_config.rs

複合信標組態

下列範例會在 複合信標組態中於本機定義加密和簽章的組件清單。

```
let compound_beacon_list = vec![
    CompoundBeacon::builder()
        .name("compound_beacon_name")
        .split(".")
        .encrypted(encrypted_parts_list)
        .signed(signed_parts_list)
        .constructors(constructor_list)
        .build()?
```

指標版本定義

下列範例會在信標版本中全域定義加密和簽章的組件清單。如需定義信標版本的詳細資訊，請參閱使用信標。

```
let beacon_versions = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
```

```
    .compound_beacons(compound_beacon_list)
    .encrypted_parts(encrypted_parts_list)
    .signed_parts(signed_parts_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_versions];
```

您可以在本機或全域定義的清單中定義加密的組件和已簽章的組件。我們建議您盡可能在信標版本中的全域清單中定義加密和簽署的組件。透過全域定義加密和簽章的組件，您可以定義每個組件一次，然後在多個複合信標組態中重複使用這些組件。如果您只打算使用加密或簽章的 部分一次，您可以在複合信標組態的本機清單中定義它。您可以在建構器清單中參考本機和全域部分。

如果您全域定義加密和簽章的組件清單，則必須提供建構組件清單，以識別複合信標可以組合複合信標組態中欄位的所有可能方式。

> **Note**
>
> 若要全域定義加密和已簽章的組件清單，您必須使用 AWS 資料庫加密 SDK 的 3.2 版或更新版本。先將新版本部署至所有讀者，再全域定義任何新組件。
> 您無法更新現有的信標組態，以定義全域加密和已簽章的組件清單。

若要設定複合信標，請提供下列值。

指標名稱

您在查詢加密欄位時使用的名稱。

信標名稱可與加密欄位或虛擬欄位的名稱相同，但不能與未加密欄位的名稱相同。兩個信標不能具有相同的信標名稱。如需判斷實作最佳信標名稱的協助，請參閱選擇信標名稱。

分割字元

用來分隔組成複合信標之部分的字元。

分割字元不能出現在複合信標建構來源之任何欄位的純文字值中。

加密的組件清單

識別複合信標中包含ENCRYPT_AND_SIGN的欄位。

每個部分都必須包含名稱和字首。部分名稱必須是從加密欄位建構的標準信標名稱。字首可以是任何字串，但必須是唯一的。加密的部分不能具有與已簽章部分相同的字首。我們建議使用短值來區分部分與複合信標提供的其他部分。

我們建議您盡可能全域定義加密的組件。如果您只打算在一個複合信標中使用加密部分，您可以考慮在本機定義加密部分。本機定義的加密部分不能具有與全域定義的加密部分相同的字首或名稱。

Java

```
List<EncryptedPart> encryptedPartList = new ArrayList<>);
EncryptedPart encryptedPartExample = EncryptedPart.builder()
    .name("standardBeaconName")
    .prefix("E-")
    .build();
encryptedPartList.add(encryptedPartExample);
```

C# / .NET

```
var encryptedPartList = new List<EncryptedPart>();
var encryptedPartExample = new EncryptedPart
 {
    Name = "compoundBeaconName",
    Prefix = "E-"
 };
encryptedPartList.Add(encryptedPartExample);
```

Rust

```
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("standard_beacon_name")
        .prefix("E-")
        .build()?
];
```

已簽章的組件清單

識別複合信標中包含的簽署欄位。

> **Note**
>
> 簽章部分是選用的。您可以設定不參考任何已簽章部分的複合信標。

每個部分都必須包含名稱、來源和字首。來源是 部分識別的 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 欄位。來源必須是欄位名稱或參照巢狀欄位值的索引。如果您的部分名稱識別來源，您可以省略來源，而 AWS 資料庫加密 SDK 會自動使用該名稱做為來源。我們建議您盡可能指定來源做為零件名稱。字首可以是任何字串，但必須是唯一的。已簽章的部分不能具有與加密部分相同的字首。我們建議使用短值來區分部分與複合信標提供的其他部分。

我們建議您盡可能全域定義您的已簽署組件。如果您只打算在一個複合信標中使用已簽署的部分，您可以考慮在本機定義。本機定義的已簽章部分不能具有與全域定義的已簽章部分相同的字首或名稱。

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
SignedPart signedPartExample = SignedPart.builder()
    .name("signedFieldName")
    .prefix("S-")
    .build();
signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

Rust

```
let signed_parts_list = vec![
    SignedPart::builder()
```

```
        .name("signed_field_name_1")
        .prefix("S-")
        .build()?,
   SignedPart::builder()
        .name("signed_field_name_2")
        .prefix("SF-")
        .build()?,
];
```

建構器清單

識別建構函式，其定義了加密和簽章的元件可由複合信標組合的不同方式。您可以在建構器清單中參考本機和全域組件。

如果您從全域定義的加密和簽章部分建構複合信標，則必須提供建構器清單。

如果您未使用任何全域定義的加密或簽章部分來建構複合信標，則建構器清單是選用的。如果您未指定建構器清單， AWS 資料庫加密 SDK 會將複合信標與下列預設建構器組合。

- 所有已簽章的組件，依其新增至已簽章的組件清單的順序排列
- 所有加密的組件，其順序會新增至加密的組件清單
- 所有組件都是必要的

建構函式

每個建構函式都是建構函式的排序清單，可定義組合複合信標的一種方式。建構函式部分會依新增至清單的順序聯結在一起，每個部分以指定的分割字元分隔。

每個建構函式部分都會命名加密的組件或已簽章的組件，並定義該組件在建構函式中是必要還是選用。例如，如果您想要在 `Field1`、 `Field1.Field2`和 上查詢複合信標`Field1.Field2.Field3`，請將 `Field2`和 標記為`Field3`選用，並建立一個建構器。

每個建構器必須至少有一個必要的部分。我們建議在每個建構函數中，先做第一部分，以便您可以在查詢中使用`BEGINS_WITH`運算子。

如果記錄中存在所有必要的部分，建構器就會成功。當您撰寫新記錄時，複合信標會使用建構器清單來判斷信標是否可以從提供的值組合。它會嘗試依建構器新增至建構器清單的順序組合信標，並使用第一個成功的建構器。如果沒有建構器成功，則不會將信標寫入記錄。

所有讀取器和寫入器都應指定相同的建構器順序，以確保其查詢結果正確。

使用下列程序來指定您自己的建構器清單。

1. 為每個加密的組件和已簽章的組件建立建構組件，以定義是否需要該組件。

   建構函式部分名稱必須是其代表的標準信標或已簽章欄位的名稱。

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
           .required(true)
           .build();
   ```

   C# / .NET

   ```
   var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
    = true };
   ```

   Rust

   ```
   let field_1_constructor_part = ConstructorPart::builder()
       .name("field_1")
       .required(true)
       .build()?;
   ```

2. 使用您在步驟 1 中建立的建構函式部分，為每個可能的複合信標組合方式建立建構函式。

   例如，如果您想要在 `Field1.Field2.Field3`和 上查詢 `Field4.Field2.Field3`，則必須建立兩個建構函數。 `Field1`和 `Field4`都可能是必要的，因為它們是在兩個不同的建構函數中定義。

   Java

   ```
   // Create a list for Field1.Field2.Field3 queries
   List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
   field123ConstructorPartList.add(field1ConstructorPart);
   field123ConstructorPartList.add(field2ConstructorPart);
   field123ConstructorPartList.add(field3ConstructorPart);
   Constructor field123Constructor = Constructor.builder()
           .parts(field123ConstructorPartList)
           .build();
   // Create a list for Field4.Field2.Field1 queries
   List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
   field421ConstructorPartList.add(field4ConstructorPart);
   field421ConstructorPartList.add(field2ConstructorPart);
   ```

```
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

Rust

```
// Create a list for field1.field2.field3 queries
let field1_field2_field3_constructor = Constructor::builder()
    .parts(vec![
        field1_constructor_part,
        field2_constroctor_part.clone(),
        field3_constructor_part,
    ])
    .build()?;

// Create a list for field4.field2.field1 queries
let field4_field2_field1_constructor = Constructor::builder()
    .parts(vec![
        field4_constructor_part,
        field2_constroctor_part.clone(),
        field1_constructor_part,
    ])
    .build()?;
```

3. 建立建構器清單，其中包含您在步驟 2 中建立的所有建構器。

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

Rust

```
let constructor_list = vec![
    field1_field2_field3_constructor,
    field4_field2_field1_constructor,
];
```

4. 當您建立複合信標`constructorList`時，請指定 。

範例組態

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

下列範例示範如何設定標準和複合信標。下列組態不提供信標長度。如需判斷組態適當信標長度的說明，請參閱選擇信標長度。

若要查看示範如何設定和使用信標的完整程式碼範例，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Java、.NET 和 Rust 可搜尋加密範例。

主題

- 標準信標
- 複合信標

標準信標

如果您想要查詢 `inspector_id_last4` 欄位是否完全相符，請使用下列組態建立標準信標。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>>);
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "inspector_id_last4",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];
```

複合信標

如果您想要查詢 `inspector_id_last4`和 上的`UnitInspection`資料庫`inspector_id_last4.unit`，請使用下列組態建立複合信標。此複合信標只需要加密的組件。

Java

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
List<StandardBeacon> standardBeaconList = new ArrayList<>);
StandardBeacon inspectorBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(inspectorBeacon);

StandardBeacon unitBeacon = StandardBeacon.builder()
    .name("unit")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(unitBeacon);

// 2. Define the encrypted parts.
List<EncryptedPart> encryptedPartList = new ArrayList<>);

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
EncryptedPart encryptedPartInspector = EncryptedPart.builder()
    .name("inspector_id_last4")
    .prefix("I-")
    .build();
encryptedPartList.add(encryptedPartInspector);

EncryptedPart encryptedPartUnit = EncryptedPart.builder()
    .name("unit")
    .prefix("U-")
    .build();
encryptedPartList.add(encryptedPartUnit);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
CompoundBeacon inspectorUnitBeacon = CompoundBeacon.builder()
    .name("inspectorUnitBeacon")
    .split(".")
    .sensitive(encryptedPartList)
```

```
    .build();
```

C# / .NET

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
StandardBeacon inspectorBeacon = new StandardBeacon
 {
   Name = "inspector_id_last4",
   Length = 10
 };
standardBeaconList.Add(inspectorBeacon);
StandardBeacon unitBeacon = new StandardBeacon
 {
    Name = "unit",
    Length = 30
 };
standardBeaconList.Add(unitBeacon);

// 2. Define the encrypted parts.
var last4EncryptedPart = new EncryptedPart

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
var last4EncryptedPart = new EncryptedPart
 {
   Name = "inspector_id_last4",
   Prefix = "I-"
 };
encryptedPartList.Add(last4EncryptedPart);

var unitEncryptedPart = new EncryptedPart
 {
   Name = "unit",
   Prefix = "U-"
 };
encryptedPartList.Add(unitEncryptedPart);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
```

```
var compoundBeaconList = new List<CompoundBeacon>>);
var inspectorCompoundBeacon = new CompoundBeacon
  {
      Name = "inspector_id_last4",
      Split = ".",
      Encrypted = encryptedPartList
  };
compoundBeaconList.Add(inspectorCompoundBeacon);
```

Rust

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];

// 2. Define the encrypted parts.
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("inspector_id_last4")
        .prefix("I-")
        .build()?,
    EncryptedPart::builder().name("unit").prefix("U-").build()?,
];

// 3. Create the compound beacon
// This compound beacon only requires a name, split character,
// and list of encrypted parts
let compound_beacon_list = vec![CompoundBeacon::builder()
    .name("last4UnitCompound")
    .split(".")
    .encrypted(encrypted_parts_list)
    .build()?];
```

使用信標

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

Beacons 可讓您搜尋加密的記錄，而不會解密要查詢的整個資料庫。信標旨在於新的未填入資料庫中實作。在現有資料庫中設定的任何信標只會映射寫入資料庫的新記錄。信標是根據欄位的純文字值計算，一旦欄位加密，信標就無法映射現有資料。使用信標寫入新記錄後，您就無法更新信標的組態。不過，您可以為新增至記錄的新欄位新增新信標。

設定信標之後，您必須先完成下列步驟，才能開始填入資料庫並對信標執行查詢。

1. 建立 AWS KMS 階層式 keyring

   若要使用可搜尋加密，您必須使用AWS KMS 階層式 keyring 來產生、加密和解密用於保護記錄的資料金鑰。

   設定信標之後，請組合階層式 keyring 先決條件並建立階層式 keyring。

   如需為何需要階層式 keyring 的詳細資訊，請參閱使用階層式 keyring 進行可搜尋加密。

2. 定義信標版本

   指定您的 `keyStore`、`keySource`、您設定的所有標準信標清單、您設定的所有複合信標清單、加密組件清單、已簽署組件清單和信標版本。您必須1為信標版本指定 。如需定義 的指引`keySource`，請參閱 定義您的信標金鑰來源。

   下列 Java 範例定義單一租戶資料庫的信標版本。如需定義多租戶資料庫信標版本的協助，請參閱多租戶資料庫的可搜尋加密。

   Java

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartsList)
```

```
        .signedParts(signedPartsList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
```

```
        SingleKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

3. 設定次要索引

   設定信標之後，您必須先設定反映每個信標的次要索引，才能搜尋加密的欄位。如需詳細資訊，請參閱使用信標設定次要索引。

4. 定義您的密碼編譯動作

   用於建構標準信標的所有欄位都必須標記為 ENCRYPT_AND_SIGN。用於建構信標的所有其他欄位都必須標記 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

5. 設定 AWS 資料庫加密 SDK 用戶端

   若要設定 Database AWS Encryption SDK 用戶端來保護 DynamoDB 資料表中的資料表項目，請參閱適用於 DynamoDB 的 Java 用戶端加密程式庫。

查詢信標

您設定的信標類型決定您可以執行的查詢類型。標準信標使用篩選條件表達式來執行等式搜尋。複合信標結合純文字字串和標準信標，以執行複雜的查詢。當您查詢加密的資料時，您會搜尋信標名稱。

您無法比較兩個標準信標的值，即使它們包含相同的基礎純文字。兩個標準信標將產生兩個不同的 HMAC 標籤，用於相同的純文字值。因此，標準信標無法執行下列查詢。

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

複合信標可以執行下列查詢。

- BEGINS_WITH(*a*)，其中 *a*會反映組裝複合信標開頭欄位的整個值。您無法使用運算BEGINS_WITH子來識別以特定子字串開頭的值。不過，您可以使用 BEGINS_WITH(*S_*)，其中 *S_*會反映組裝複合信標開頭的部分字首。
- CONTAINS(*a*)，其中 *a*會反映已組裝複合信標包含之欄位的整個值。您無法使用運算CONTAINS子來識別包含特定子字串或集合內值的記錄。

  例如，您無法執行查詢，CONTAINS(*path*, "*a*"其中 *a*反映集合中的值。
- 您可以比較複合信標的簽章部分。當您比較已簽章的組件時，您可以選擇將加密組件的字首附加到一或多個已簽章的組件，但您無法在任何查詢中包含加密欄位的值。

  例如，您可以在 *signedField1* = *signedField2*或 上比較已簽章的組件和查詢*value* IN (*signedField1*, *signedField2*, ...)。

  您也可以在 上透過查詢來比較已簽章的組件和已加密組件的字首*signedField1.A_* = *signedField2.B_*。
- *field* BETWEEN *a* AND *b*，其中 *a*和 *b*是簽署的組件。您可以選擇將加密部分的字首附加到一或多個已簽章的組件，但您無法在任何查詢中包含加密欄位的值。

您必須在複合信標的查詢中包含每個部分的字首。例如，如果您從兩個欄位 encryptedField 和 建構複合信標 compoundBeacon，signedField則必須在查詢信標時包含為這兩個部分設定的字首。

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

多租戶資料庫的可搜尋加密

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

若要在資料庫中實作可搜尋的加密，您必須使用AWS KMS 階層式 keyring。 AWS KMS 階層式 keyring 會產生、加密和解密用於保護記錄的資料金鑰。它也會建立用來產生信標的信標金鑰。搭配多租戶資料庫使用 AWS KMS 階層式 keyring 時，每個租戶都有不同的分支金鑰和信標金鑰。若要查詢多租戶資料庫中的加密資料，您必須識別用來產生您正在查詢之信標的信標金鑰資料。如需詳細資訊，請參閱the section called “使用階層式 keyring 進行可搜尋加密”。

當您定義多租戶資料庫的信標版本時，請指定您設定的所有標準信標清單、您設定的所有複合信標清單、信標版本和 keySource。您必須將信標金鑰來源定義為 MultiKeyStore，並包含 keyFieldName、本機信標金鑰快取的存留時間，以及本機信標金鑰快取的快取大小上限。

如果您設定了任何已簽章的信標，則必須包含在 中compoundBeaconList。已簽章的信標是一種複合信標類型，可對 SIGN_ONLY 和 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT欄位編製索引並執行複雜的查詢。

Java

```
List<BeaconVersion> beaconVersions = new ArrayList<>();
    beaconVersions.add(
        BeaconVersion.builder()
                .standardBeacons(standardBeaconList)
                .compoundBeacons(compoundBeaconList)
                .version(1) // MUST be 1
                .keyStore(branchKeyStoreName)
                .keySource(BeaconKeySource.builder()
                        .multi(MultiKeyStore.builder()
                                .keyFieldName(keyField)
                                .cacheTTL(6000)
                                .maxCacheSize(10)
                        .build())
                .build())
        .build()
    );
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Multi = new MultiKeyStore
```

```
            {
                KeyId = branch-key-id,
                CacheTTL = 6000,
                MaxCacheSize = 10
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Multi(
        MultiKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .max_cache_size(10)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

keyFieldName

keyFieldName 定義存放與用於為指定租用戶產生信標之信標金鑰branch-key-id相關聯的 的欄位名稱。

當您將新記錄寫入資料庫時，識別用於為該記錄產生任何信標的branch-key-id信標金鑰的 會存放在此欄位中。

根據預設， keyField是概念性欄位，不會明確存放在資料庫中。 AWS Database Encryption SDK 從材料描述中的branch-key-id加密資料金鑰識別 ，並將該值存放在概念中，keyField供

您在複合信標和簽章信標中參考。由於材料描述已簽署，因此概念keyField會被視為已簽署部分。

您也可以將 做為 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 欄位包含在密碼編譯動作keyField中，以明確將 欄位存放在資料庫中。如果您這樣做，keyField每次將記錄寫入資料庫時，都必須手動將 包含在 branch-key-id 中。

查詢多租戶資料庫中的信標

若要查詢信標，您必須在查詢keyField中包含 ，以識別重新計算信標所需的適當信標金鑰材料。您必須指定與用來產生記錄信標的信標金鑰branch-key-id相關聯的 。您無法在分支金鑰 ID 供應商branch-key-id中指定識別租用戶的易記名稱。您可以透過下列方式將 包含在查詢keyField中。

複合信標

無論您是否明確keyField將 存放在記錄中，都可以將 keyField直接納入複合信標中，做為已簽章的部分。keyField 簽署的 部分為必填。

例如，如果您想要建構複合信標compoundBeacon、、兩個欄位的 encryptedField和 signedField，您也必須將 keyField包含為已簽章的部分。這可讓您在 上執行下列查詢compoundBeacon。

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue.K_branch-key-id
```

已簽章的信標

AWS Database Encryption SDK 使用標準和複合信標來提供可搜尋的加密解決方案。這些信標必須至少包含一個加密欄位。不過， AWS 資料庫加密 SDK 也支援簽署的信標，這些信標可以完全從純文字SIGN_ONLY和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT欄位設定。

簽章信標可以從單一部分建構。無論您是否明確keyField將 存放在記錄中，都可以從 建構已簽章的信標，keyField並使用它來建立複合查詢，將已keyField簽章信標上的查詢與其中一個其他信標上的查詢結合。例如，您可以執行下列查詢。

```
keyField = K_branch-key-id AND compoundBeacon =
 E_encryptedFieldValue.S_signedFieldValue
```

如需設定簽章信標的說明，請參閱 建立簽章的信標

直接在 上查詢 keyField

如果您在密碼編譯動作keyField中指定 ，並明確將 欄位存放在記錄中，您可以建立複合查詢，將信標上的查詢與 上的查詢結合在一起keyField。keyField 如果您想要查詢標準信標，您可以選擇直接在 上查詢。例如，您可以執行下列查詢。

```
keyField = branch-key-id AND standardBeacon = S_standardBeaconValue
```

AWS 適用於 DynamoDB 的資料庫加密 SDK

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

適用於 DynamoDB 的 AWS Database Encryption SDK 是一種軟體程式庫，可讓您在 Amazon DynamoDB 設計中包含用戶端加密。DynamoDB 的 AWS Database Encryption SDK 提供屬性層級加密，可讓您指定要加密的項目，以及要包含在簽章中的項目，以確保資料的真實性。加密傳輸中和靜態的敏感資料有助於確保您的純文字資料不可供任何第三方使用，包括 AWS。

> **Note**
>
> AWS Database Encryption SDK 不支援 PartiQL。

在 DynamoDB 中，資料表是項目的集合。每個項目都是屬性的集合。每個屬性都有名稱和數值。適用於 DynamoDB 的 AWS Database Encryption SDK 會加密屬性的值。接著，它會透過屬性計算簽章。您可以指定要加密的屬性值，以及在密碼編譯動作的簽章中包含哪些屬性值。

本章中的主題提供 DynamoDB AWS 資料庫加密 SDK 的概觀，包括加密哪些欄位、用戶端安裝和組態的指引，以及協助您開始使用的 Java 範例。

主題

- 用戶端加密和伺服器端加密
- 哪些欄位已加密並簽署？
- DynamoDB 中的可搜尋加密
- 更新您的資料模型
- AWS 適用於 DynamoDB 的資料庫加密 SDK 可用的程式設計語言
- 舊版 DynamoDB 加密用戶端

用戶端加密和伺服器端加密

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

適用於 DynamoDB 的 AWS Database Encryption SDK 支援用戶端加密，您可以在其中加密資料表資料，再將其傳送至資料庫。不過，DynamoDB 提供伺服器端靜態加密功能，可讓您在資料表保留到磁碟時透明地加密資料表，並在您存取資料表時解密資料表。

您所應選擇的工具，取決於資料的敏感度以及應用程式的安全性需求。您可以使用 DynamoDB 的 AWS 資料庫加密 SDK 和靜態加密。當您將加密和簽署的項目傳送至 DynamoDB 時，DynamoDB 不會將項目識別為受保護。它只會偵測具有二進位屬性值的一般資料表項目。

伺服器端靜態加密

DynamoDB 支援靜態加密，這是一種伺服器端加密功能，DynamoDB 會在資料表保留到磁碟時為您透明加密資料表，並在您存取資料表資料時解密資料表。

當您使用 AWS SDK 與 DynamoDB 互動時，您的資料預設會透過 HTTPS 連線在傳輸中加密、在 DynamoDB 端點解密，然後在存放在 DynamoDB 之前重新加密。

- 預設加密。DynamoDB 會在寫入所有資料表時，以透明方式加密和解密。沒有啟用或停用靜態加密的選項。
- DynamoDB 會建立和管理密碼編譯金鑰。 每個資料表的唯一索引鍵受到AWS KMS key永遠不會讓 AWS Key Management Service(AWS KMS) 未加密的 保護。根據預設，DynamoDB 會使用 DynamoDB 服務帳戶中AWS 擁有的金鑰的 ，但您可以選擇帳戶中的 AWS 受管金鑰或客戶受管金鑰來保護您的部分或全部資料表。
- 所有資料表資料都會在磁碟上加密。 將加密的資料表儲存至磁碟時，DynamoDB 會加密所有資料表資料，包括主索引鍵和本機和全域次要索引。如果您的資料表有排序索引鍵，則某些標示範圍界限的排序索引鍵將會以純文字的格式存放在資料表中繼資料中。
- 與資料表相關的物件也會加密。靜態加密可在 DynamoDB 串流、全域資料表和備份寫入耐用媒體時保護它們。
- 您的項目會在您存取時解密。 當您存取資料表時，DynamoDB 會解密包含目標項目的資料表部分，並將純文字項目傳回給您。

AWS 適用於 DynamoDB 的資料庫加密 SDK

無論是在傳輸中、靜態時，還是從來源儲存至 DynamoDB 時，用戶端加密都可為資料提供端對端的保護。您的純文字資料永遠不會公開給任何第三方，包括 AWS。您可以使用適用於 DynamoDB 的 AWS 資料庫加密 SDK 搭配新的 DynamoDB 資料表，或者您可以將現有的 Amazon DynamoDB 資料表遷移至最新版本的 AWS DynamoDB 資料庫加密 SDK。

- 傳輸中和靜態的資料都受到保護。它永遠不會向任何第三方公開，包括 AWS。
- 您可以簽署您的資料表項目。您可以指示 DynamoDB 的 AWS Database Encryption SDK 計算資料表項目的全部或部分簽章，包括主要金鑰屬性。此簽章可讓您偵測對整體項目的未授權變更，包括新增或刪除屬性，或是交換屬性值。
- 您可以透過選取 keyring 來決定如何保護資料。您的 keyring 會決定保護資料金鑰的包裝金鑰，最終也決定您的資料。使用對您的任務最安全的包裝金鑰。
- 適用於 DynamoDB 的 AWS Database Encryption SDK 不會加密整個資料表。您可以選擇在項目中加密的屬性。適用於 DynamoDB 的 AWS Database Encryption SDK 不會加密整個項目。它不會加密屬性名稱，或是主索引鍵 (分割區索引鍵和排序索引鍵) 屬性的名稱或值。

AWS Encryption SDK

如果您要加密存放在 DynamoDB 中的資料，建議您使用 AWS 資料庫加密 SDK for DynamoDB。

AWS Encryption SDK 是用戶端加密程式庫，可協助您進行一般資料的加密和解密。雖然它可保護任何類型的資料，但在設計上並不是用來處理結構化資料 (例如資料庫記錄)。與適用於 DynamoDB 的 AWS 資料庫加密 SDK 不同， AWS Encryption SDK 無法提供項目層級完整性檢查，而且沒有可識別屬性或防止主要金鑰加密的邏輯。

如果您使用 AWS Encryption SDK 來加密資料表的任何元素，請記住它與 DynamoDB 的 AWS Database Encryption SDK 不相容。您無法用一個程式庫加密，但用另一個程式庫解密。

哪些欄位已加密並簽署？

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

適用於 DynamoDB 的 AWS Database Encryption SDK 是用戶端加密程式庫，專為 Amazon DynamoDB 應用程式而設計。Amazon DynamoDB 將資料存放在資料表中，這是項目的集合。每個項目都是屬性的集合。每個屬性都有名稱和數值。適用於 DynamoDB 的 AWS Database Encryption

SDK 會加密屬性的值。接著，它會透過屬性計算簽章。您可以指定哪些屬性值要加密，以及哪些屬性值要包含在簽章中。

加密可保護屬性值的機密性。簽署可提供所有已簽署屬性的完整性及其彼此的關係，並可提供驗證。它可讓您偵測對整體項目的未授權變更 (包括新增或刪除屬性)，或是替換已加密的值。

在加密項目中，部分資料會保留純文字，包括資料表名稱、所有屬性名稱、未加密的屬性值、主索引鍵（分割區索引鍵和排序索引鍵）屬性的名稱和值，以及屬性類型。請勿在這些欄位中存放敏感性資訊。

如需 DynamoDB AWS 資料庫加密開發套件運作方式的詳細資訊，請參閱AWS 資料庫加密 SDK 的運作方式。

> **Note**
>
> DynamoDB 專用 AWS 資料庫加密 SDK 主題中所有提及的屬性動作都是指密碼編譯動作。

主題

- 加密屬性值
- 簽署項目

加密屬性值

適用於 DynamoDB 的 AWS Database Encryption SDK 會加密您指定屬性的值（但不會加密屬性名稱或類型）。若要決定加密的是哪些屬性值，請使用屬性動作。

例如，這個項目包括 `example` 和 `test` 屬性。

```
'example': 'data',
'test': 'test-value',
...
```

如果您將 `example` 屬性加密，但不加密 `test` 屬性，結果如下所示。已加密的 `example` 屬性值是二進位資料，而不是字串。

```
'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb\x9fY
\x9f\xf3\xc9C\x83\r\xbb\\"),
'test': 'test-value'
```

```
...
```

每個項目的主要索引鍵屬性 - 分割區索引鍵和排序索引鍵 - 必須保持純文字，因為 DynamoDB 會使用它們在資料表中尋找項目。這些屬性應加以簽署，但不要加密。

適用於 DynamoDB 的 AWS Database Encryption SDK 會為您識別主要金鑰屬性，並確保其值已簽章，但不會加密。而且，如果您找出您的主要索引鍵，然後試著將它加密，則用戶端會擲出例外狀況。

用戶端會將材料描述存放在新增至項目的新屬性 (`aws_dbe_head`) 中。材料描述說明項目的加密和簽署方式。用戶端會使用此資訊來驗證並解密項目。存放材料描述的欄位不會加密。

簽署項目

加密指定的屬性值後，DynamoDB 的 AWS Database Encryption SDK 會計算雜湊型訊息驗證碼 (HMACs) 和數位簽章，而不是材料描述、加密內容和屬性動作`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`中標記為 `ENCRYPT_AND_SIGN`、`SIGN_ONLY`或 的每個欄位的正式化。ECDSA 簽章預設為啟用，但並非必要。用戶端會將 HMACs 和簽章存放在新增至項目的新屬性 (`aws_dbe_foot`) 中。

DynamoDB 中的可搜尋加密

若要設定 Amazon DynamoDB 資料表進行可搜尋加密，您必須使用AWS KMS 階層式 keyring 來產生、加密和解密用於保護項目的資料金鑰。您還必須在資料表加密組態`SearchConfig`中包含 。

> **Note**
>
> 如果您使用適用於 DynamoDB 的 Java 用戶端加密程式庫，則必須使用適用於 DynamoDB API 的低階 AWS 資料庫加密開發套件來加密、簽署、驗證和解密資料表項目。DynamoDB 增強型用戶端和較低層級`DynamoDBItemEncryptor`不支援可搜尋加密。

主題

- 使用信標設定次要索引
- 測試信標輸出

使用信標設定次要索引

設定信標之後，您必須先設定反映每個信標的次要索引，才能搜尋加密的屬性。

當您設定標準或複合信標時， AWS 資料庫加密 SDK 會將aws_dbe_b_字首新增至信標名稱，以便伺服器輕鬆識別信標。例如，如果您命名複合信標 compoundBeacon，則完整信標名稱實際上是 aws_dbe_b_compoundBeacon。如果您想要設定包含標準或複合信標的次要索引，您必須在識別信標名稱時包含字aws_dbe_b_首。

分割區和排序索引鍵

您無法加密主要金鑰值。必須簽署您的分割區和排序索引鍵。您的主索引鍵值不能是標準或複合信標。

您的主索引鍵值必須是 SIGN_ONLY，除非您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，否則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

您的主索引鍵值可以是簽章的信標。如果您為每個主索引鍵值設定不同的簽章信標，則必須指定屬性名稱，以識別主索引鍵值為簽章信標名稱。不過， AWS 資料庫加密 SDK 不會將aws_dbe_b_字首新增至已簽章的信標。即使您為主要索引鍵值設定不同的簽章信標，您只需要在設定次要索引時為主要索引鍵值指定屬性名稱。

本機次要索引

本機次要索引的排序索引鍵可以是信標。

如果您指定排序索引鍵的信標，則類型必須為字串。如果您為排序索引鍵指定標準或複合信標，則必須在指定信標名稱時包含字aws_dbe_b_首。如果您指定已簽章的信標，請指定不含任何字首的信標名稱。

全域次要索引

全域次要索引的分割區和排序索引鍵可以是信標。

如果您為分割區或排序索引鍵指定信標，則類型必須為字串。如果您為排序索引鍵指定標準或複合信標，則必須在指定信標名稱時包含字aws_dbe_b_首。如果您指定已簽章的信標，請指定不含任何字首的信標名稱。

屬性投影

投影是指從資料表複製到次要索引的屬性集合。資料表的分割區索引鍵和排序索引鍵一律會投影到索引中；您可以投影其他屬性來支援應用程式的查詢需求。DynamoDB 為屬性投影提供三種不同的選項：KEYS_ONLY、 INCLUDE和 ALL。

如果您使用 INCLUDE 屬性投影在信標上搜尋，則必須指定信標建構來源的所有屬性名稱，以及字aws_dbe_b_首為 的信標名稱。例如，如果您從 、 field1field2和 設

定複合信標 compoundBeaconfield3，您必須在投影field3中指定 field2、、aws_dbe_b_compoundBeacon field1和 。

全域次要索引只能使用投影中明確指定的屬性，但本機次要索引可以使用任何屬性。

測試信標輸出

如果您設定複合信標或使用虛擬欄位建構信標，建議您在填入 DynamoDB 資料表之前，先驗證這些信標是否產生預期的輸出。

AWS Database Encryption SDK 提供 DynamoDbEncryptionTransforms服務，協助您疑難排解虛擬欄位和複合信標輸出。

測試虛擬欄位

下列程式碼片段會建立測試項目、使用 DynamoDB 資料表加密組態定義DynamoDbEncryptionTransforms服務，並示範如何使用 ResolveAttributes來驗證虛擬欄位是否產生預期的輸出。

Java

請參閱完整的程式碼範例：VirtualBeaconSearchableEncryptionExample.java

```
// Create test items
final PutItemRequest itemWithHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithHasTestResult)
    .build();

final PutItemResponse itemWithHasTestResultPutResponse =
 ddb.putItem(itemWithHasTestResultPutRequest);

final PutItemRequest itemWithNoHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithNoHasTestResult)
    .build();

final PutItemResponse itemWithNoHasTestResultPutResponse =
 ddb.putItem(itemWithNoHasTestResultPutRequest);

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
```

```
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(itemWithHasTestResult)
    .Version(1)
    .build();
final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Map<String, String> vf = new HashMap<>();
vf.put("stateAndHasTestResult", "CAt");
assert resolveOutput.VirtualFields().equals(vf);
```

C# / .NET

請參閱完整的程式碼範例：VirtualBeaconSearchableEncryptionExample.cs。

```
 // Create item with hasTestResult=true
var itemWithHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("ABC-123"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = true }
};

// Create item with hasTestResult=false
var itemWithNoHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("DEF-456"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = false }
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
```

```
    Item = itemWithHasTestResult,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Debug.Assert(resolveOutput.VirtualFields.Count == 1);
Debug.Assert(resolveOutput.VirtualFields["stateAndHasTestResult"] == "CAt");
```

Rust

請參閱完整的程式碼範例： virtual_beacon_searchable_encryption.rs。

```
// Create item with hasTestResult=true
let item_with_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("ABC-123".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(true)),
]);

// Create item with hasTestResult=false
let item_with_no_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("DEF-456".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(false)),
]);

// Define the transform service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;
```

```
// Verify the configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item_with_has_test_result.clone())
    .version(1)
    .send()
    .await?;

// Verify that VirtualFields has the expected value
let virtual_fields = resolve_output.virtual_fields.unwrap();
assert_eq!(virtual_fields.len(), 1);
assert_eq!(virtual_fields["stateAndHasTestResult"], "CAt");
```

測試複合信標

下列程式碼片段會建立測試項目、使用 DynamoDB 資料表加密組態定義DynamoDbEncryptionTransforms服務，並示範如何使用 ResolveAttributes來驗證複合信標是否產生預期的輸出。

Java

請參閱完整的程式碼範例：CompoundBeaconSearchableEncryptionExample.java

```
// Create an item with both attributes used in the compound beacon.
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("work_id", AttributeValue.builder().s("9ce39272-8068-4efd-a211-
cd162ad65d4c").build());
item.put("inspection_date", AttributeValue.builder().s("2023-06-13").build());
item.put("inspector_id_last4", AttributeValue.builder().s("5678").build());
item.put("unit", AttributeValue.builder().s("011899988199").build());

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(item)
    .Version(1)
    .build();
```

```
final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that CompoundBeacons has the expected value
Map<String, String> cbs = new HashMap<>();
cbs.put("last4UnitCompound", "L-5678.U-011899988199");
assert resolveOutput.CompoundBeacons().equals(cbs);
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

C# / .NET

請參閱完整的程式碼範例：CompoundBeaconSearchableEncryptionExample.cs

```
// Create an item with both attributes used in the compound beacon
var item = new Dictionary<String, AttributeValue>
{
    ["work_id"] = new AttributeValue("9ce39272-8068-4efd-a211-cd162ad65d4c"),
    ["inspection_date"] = new AttributeValue("2023-06-13"),
    ["inspector_id_last4"] = new AttributeValue("5678"),
    ["unit"] = new AttributeValue("011899988199")
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = item,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);


// Verify that CompoundBeacons has the expected value
Debug.Assert(resolveOutput.CompoundBeacons.Count == 1);
Debug.Assert(resolveOutput.CompoundBeacons["last4UnitCompound"] ==
 "L-5678.U-011899988199");
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
```

```
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

Rust

請參閱完整的程式碼範例： compound_beacon_searchable_encryption.rs

```
// Create an item with both attributes used in the compound beacon
let item = HashMap::from([
    (
        "work_id".to_string(),
        AttributeValue::S("9ce39272-8068-4efd-a211-cd162ad65d4c".to_string()),
    ),
    (
        "inspection_date".to_string(),
        AttributeValue::S("2023-06-13".to_string()),
    ),
    (
        "inspector_id_last4".to_string(),
        AttributeValue::S("5678".to_string()),
    ),
    (
        "unit".to_string(),
        AttributeValue::S("011899988199".to_string()),
    ),
]);

// Define the transforms service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item.clone())
    .version(1)
    .send()
    .await?;

// Verify that CompoundBeacons has the expected value
Dlet compound_beacons = resolve_output.compound_beacons.unwrap();
assert_eq!(compound_beacons.len(), 1);
assert_eq!(
    compound_beacons["last4UnitCompound"],
```

```
    "L-5678.U-011899988199"
);
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

更新您的資料模型

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

當您設定 DynamoDB 的 AWS 資料庫加密 SDK 時，您會提供屬性動作。加密時， AWS Database Encryption SDK 會使用屬性動作來識別要加密和簽署的屬性、要簽署的屬性（但不要加密），以及要忽略的屬性。您也可以定義允許的未簽章屬性，明確告知用戶端哪些屬性會從簽章中排除。在解密時，AWS 資料庫加密 SDK 會使用您定義的允許未簽章屬性，來識別簽章中不包含哪些屬性。屬性動作不會儲存在加密的項目中，資料庫 AWS 加密 SDK 不會自動更新您的屬性動作。

請仔細選擇屬性動作。如有疑問，請使用加密並簽署。使用 AWS Database Encryption SDK 保護您的項目之後，您無法將現有的 `ENCRYPT_AND_SIGN`、 `SIGN_ONLY`或 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 屬性變更為 `DO_NOTHING`。不過，您可以安全地進行下列變更。

- 新增 `ENCRYPT_AND_SIGN`、 `SIGN_ONLY`和 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` 屬性
- 移除現有屬性
- 將現有`ENCRYPT_AND_SIGN`屬性變更為 `SIGN_ONLY`或 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- 將現有`SIGN_ONLY`或`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性變更為 `ENCRYPT_AND_SIGN`
- 新增`DO_NOTHING`屬性
- 將現有`SIGN_ONLY`屬性變更為 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- 將現有`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性變更為 `SIGN_ONLY`

可搜尋加密的考量事項

在更新資料模型之前，請仔細考慮您的更新如何影響您從屬性建構的任何信標。使用信標寫入新記錄之後，您就無法更新信標的組態。您無法更新與您用來建構信標的屬性相關聯的屬性動作。如果您移除現有的屬性及其相關聯的信標，您將無法使用該信標查詢現有的記錄。您可以為新增至記錄的新欄位建立新的信標，但無法更新現有的信標以包含新欄位。

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性的考量

根據預設，分割區和排序索引鍵是加密內容中包含的唯一屬性。您可以考慮定義其他欄位，SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT以便AWS KMS 階層式 keyring 的分支金鑰 ID 供應商可以識別從加密內容解密所需的分支金鑰。如需詳細資訊，請參閱分支金鑰 ID 供應商。如果您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，則分割區和排序屬性也必須是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 若要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀取器SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

新增 ENCRYPT_AND_SIGN、 SIGN_ONLY和 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性

若要新增 ENCRYPT_AND_SIGN、 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性，請在屬性動作中定義新屬性。

您無法移除現有的DO_NOTHING屬性，並將其新增為 ENCRYPT_AND_SIGN、 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請將新屬性新增至註釋的資料類別。如果您未指定新屬性的屬性動作註釋，用戶端預設會加密和簽署新屬性（除非屬性是主索引鍵的一部分）。如果您只想要簽署新屬性，則必須使用 @DynamoDBEncryptionSignOnly或 @DynamoDBEncryptionSignAndIncludeInEncryptionContext註釋新增新屬性。

使用物件模型

如果您手動定義屬性動作，請將新屬性新增至物件模型中的屬性動作SIGN_ONLY，並指定 ENCRYPT_AND_SIGN、 或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT作為屬性動作。

移除現有屬性

如果您決定不再需要 屬性，您可以停止將資料寫入該屬性，也可以正式將其從屬性動作中移除。當您停止將新資料寫入屬性時，屬性仍會出現在屬性動作中。如果您未來需要再次開始使用 屬性，這會很有幫助。正式移除屬性動作中的屬性不會將其從資料集中移除。您的資料集仍會包含包含該屬性的項目。

若要正式移除現有的 ENCRYPT_AND_SIGN、SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、SIGN_ONLY或 DO_NOTHING 屬性，請更新您的屬性動作。

如果您移除DO_NOTHING屬性，則不得將該屬性從允許的未簽章屬性中移除。即使您不再將新值寫入該屬性，用戶端仍需要知道該屬性未簽署，以讀取包含該屬性的現有項目。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請從註釋的資料類別中移除 屬性。

使用物件模型

如果您手動定義屬性動作，請從物件模型中的屬性動作中移除屬性。

將現有ENCRYPT_AND_SIGN屬性變更為 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

若要將現有ENCRYPT_AND_SIGN屬性變更為 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，您必須更新您的屬性動作。部署更新之後，用戶端將能夠驗證和解密寫入屬性的現有值，但只會簽署寫入屬性的新值。

> **Note**
>
> 在將現有ENCRYPT_AND_SIGN屬性變更為 SIGN_ONLY或 之前，請仔細考慮您的安全需求SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。任何可存放敏感資料的屬性都應該加密。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請更新現有的屬性，將 @DynamoDBEncryptionSignOnly或

@DynamoDBEncryptionSignAndIncludeInEncryptionContext註釋包含在註釋的資料類別中。

使用物件模型

如果您手動定義屬性動作，請將與現有屬性相關聯的屬性動作從 更新ENCRYPT_AND_SIGN為 物件模型SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT中的 SIGN_ONLY或 。

將現有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為 ENCRYPT_AND_SIGN

若要將現有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為ENCRYPT_AND_SIGN，您必須更新您的屬性動作。部署更新之後，用戶端將能夠驗證寫入屬性的現有值，並將加密和簽署寫入屬性的新值。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請從現有屬性中移除@DynamoDBEncryptionSignOnly或@DynamoDBEncryptionSignAndIncludeInEncryptionContext註釋。

使用物件模型

如果您手動定義屬性動作，請在物件模型ENCRYPT_AND_SIGN中將與屬性相關聯的屬性動作從SIGN_ONLY或 更新SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT為 。

新增DO_NOTHING屬性

若要降低新增DO_NOTHING屬性時的錯誤風險，建議您在命名DO_NOTHING屬性時指定不同的字首，然後使用該字首來定義允許的未簽署屬性。

您無法從註釋的資料類別中移除現有的 ENCRYPT_AND_SIGN、 SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 屬性，然後將 屬性新增回DO_NOTHING屬性。您只能新增全新的DO_NOTHING屬性。

您新增DO_NOTHING屬性所採取的步驟，取決於您在清單中明確定義允許的未簽章屬性，還是使用字首。

使用允許的未簽署屬性字首

如果您使用 定義屬性動作TableSchema，請使用 @DynamoDBEncryptionDoNothing註釋將新DO_NOTHING屬性新增至註釋的資料類別。如果您手動定義屬性動作，請更新您的屬性動作以包含新的屬性。請務必使用 屬性DO_NOTHING動作明確設定新屬性。您必須在新屬性的名稱中包含相同的不同字首。

使用允許的未簽署屬性清單

1. 將新DO_NOTHING屬性新增至允許的未簽章屬性清單，並部署更新的清單。
2. 從步驟 1 部署變更。

   在變更傳播到需要讀取此資料的所有主機之前，您無法繼續進行步驟 3。
3. 將新DO_NOTHING屬性新增至您的屬性動作。

   a. 如果您使用 定義屬性動作TableSchema，請使用 @DynamoDBEncryptionDoNothing註釋將新DO_NOTHING屬性新增至註釋的資料類別。
   b. 如果您手動定義屬性動作，請更新您的屬性動作以包含新的屬性。請務必使用 屬性動作明確設定新DO_NOTHING屬性。
4. 從步驟 3 部署變更。

將現有SIGN_ONLY屬性變更為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

若要將現有SIGN_ONLY屬性變更為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，您必須更新您的屬性動作。部署更新之後，用戶端將能夠驗證寫入屬性的現有值，並將繼續簽署寫入屬性的新值。寫入屬性的新值將包含在加密內容中。

如果您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請將與屬性相關聯的屬性動作從 更新@DynamoDBEncryptionSignOnly為 @DynamoDBEncryptionSignAndIncludeInEncryptionContext。

使用物件模型

如果您手動定義屬性動作，請在物件模型SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT中將與屬性相關聯的屬性動作從 更新SIGN_ONLY為 。

將現有SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為SIGN_ONLY

若要將現有SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為 SIGN_ONLY，您必須更新您的屬性動作。部署更新之後，用戶端將能夠驗證寫入屬性的現有值，並將繼續簽署寫入屬性的新值。寫入屬性的新值不會包含在加密內容中。

在將現有SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性變更為 之前SIGN_ONLY，請仔細考慮您的更新如何影響分支金鑰 ID 供應商的功能。

使用註釋的資料類別

如果您使用 定義屬性動作TableSchema，請將與屬性相關聯的屬性動作從 更新@DynamoDBEncryptionSignAndIncludeInEncryptionContext為 @DynamoDBEncryptionSignOnly。

使用物件模型

如果您手動定義屬性動作，請在物件模型SIGN_ONLY中將與屬性相關聯的屬性動作從 更新SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT為 。

AWS 適用於 DynamoDB 的資料庫加密 SDK 可用的程式設計語言

適用於 DynamoDB 的 AWS Database Encryption SDK 適用於下列程式設計語言。語言專屬的程式庫有所不同，但是產生的實作都是可互通的。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。

主題

- Java
- .NET
- Rust

Java

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

本主題說明如何安裝和使用適用於 DynamoDB 的 Java 用戶端加密程式庫的 3.x 版。如需使用適用於 DynamoDB 的 AWS Database Encryption SDK 進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Java 範例。

> **Note**
>
> 下列主題著重於 DynamoDB 的 Java 用戶端加密程式庫 3.x 版。
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。 AWS Database Encryption SDK 繼續支援舊版 DynamoDB Encryption Client 版本。

主題

- 先決條件
- 安裝
- 使用適用於 DynamoDB 的 Java 用戶端加密程式庫
- Java 範例
- 設定現有的 DynamoDB 資料表以使用 DynamoDB 的 AWS Database Encryption SDK
- 遷移至 DynamoDB Java 用戶端加密程式庫的 3.x 版

先決條件

在安裝適用於 DynamoDB 的 Java 用戶端加密程式庫的 3.x 版之前，請確定您有下列先決條件。

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。

AWS SDK for Java 2.x

適用於 DynamoDB 的 AWS Database Encryption SDK 需要 的 DynamoDB 增強型用戶端模組 AWS SDK for Java 2.x。您可以安裝整個 SDK 或只安裝這個模組。

如需更新 版本的相關資訊 適用於 Java 的 AWS SDK，請參閱從 版本 1.x 遷移至 2.x。 適用於 Java 的 AWS SDK

可透過 Apache Maven 適用於 Java 的 AWS SDK 取得。您可以宣告整個 適用於 Java 的 AWS SDK或僅dynamodb-enhanced模組的相依性。

適用於 Java 的 AWS SDK 使用 Apache Maven 安裝

- 若要匯入整個 適用於 Java 的 AWS SDK 作為相依性，請在 pom.xml 檔案中宣告它。
- 若要僅為 中的 Amazon DynamoDB 模組建立相依性 適用於 Java 的 AWS SDK，請遵循指定特定模組的指示。將 groupId 設定為 software.amazon.awssdk，將 artifactID設定為 dynamodb-enhanced。

> **Note**
>
> 如果您使用 AWS KMS keyring 或 AWS KMS 階層 keyring，您也需要為 AWS KMS 模組建立相依性。將 groupId 設定為 software.amazon.awssdk，將 artifactID設定為 kms。

安裝

您可以使用下列方式安裝適用於 DynamoDB 的 Java 用戶端加密程式庫 3.x 版。

使用 Apache Maven

Amazon DynamoDB Encryption Client for Java 可透過 Apache Maven 取得，其依存性定義如下。

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

使用 Gradle Kotlin

您可以使用 Gradle 在 Amazon DynamoDB Encryption Client for Java 上宣告相依性，方法是將下列項目新增至 Gradle 專案的相依性區段。

```
implementation("software.amazon.cryptography:aws-database-encryption-sdk-
dynamodb:version-number")
```

手動

若要安裝 DynamoDB 的 Java 用戶端加密程式庫，請複製或下載 aws-database-encryption-sdk-dynamodb GitHub 儲存庫。

安裝 SDK 之後，請開始查看本指南中的範例程式碼，以及 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Java 範例。

使用適用於 DynamoDB 的 Java 用戶端加密程式庫

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

本主題說明 DynamoDB Java 用戶端加密程式庫 3.x 版中的一些函數和協助程式類別。

如需使用 DynamoDB 的 Java 用戶端加密程式庫進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Java 範例。

主題

- 項目加密程式
- DynamoDB AWS 資料庫加密 SDK 中的屬性動作
- DynamoDB AWS 資料庫加密 SDK 中的加密組態
- 使用 AWS 資料庫加密 SDK 更新項目
- 解密已簽章的集合

項目加密程式

DynamoDB AWS 資料庫加密 SDK 的核心是項目加密程式。您可以使用適用於 DynamoDB 的 Java 用戶端加密程式庫 3.x 版，以下列方式加密、簽署、驗證和解密 DynamoDB 資料表項目。

DynamoDB 增強型用戶端

您可以使用 來設定 DynamoDB 增強型用戶端`DynamoDbEncryptionInterceptor`，以自動加密和簽署項目用戶端搭配 DynamoDB `PutItem`請求。使用 DynamoDB 增強型用戶端，您可以使用註釋的資料類別定義屬性動作。我們建議您盡可能使用 DynamoDB 增強型用戶端。

DynamoDB 增強型用戶端不支援可搜尋加密。

> **Note**
>
> AWS Database Encryption SDK 不支援巢狀屬性的註釋。

低階 DynamoDB API

您可以使用 來設定低階 DynamoDB APIDynamoDbEncryptionInterceptor，以自動加密項目，並使用 DynamoDB PutItem請求在用戶端簽署項目。

您必須使用低階 DynamoDB API 才能使用可搜尋的加密。

較低層級 **DynamoDbItemEncryptor**

較低層級會DynamoDbItemEncryptor直接加密和簽署或解密，並驗證您的資料表項目，而無需呼叫 DynamoDB。它不會發出 DynamoDB PutItem或 GetItem 請求。例如，您可以使用較低層級DynamoDbItemEncryptor直接解密和驗證您已擷取的 DynamoDB 項目。

較低層級DynamoDbItemEncryptor不支援可搜尋加密。

DynamoDB AWS 資料庫加密 SDK 中的屬性動作

屬性動作會決定哪些屬性值是加密和簽署的，哪些是僅簽署的，哪些是簽署和包含在加密內容中，哪些是忽略的。

> **Note**
>
> 若要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀取器SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

如果您使用低階 DynamoDB API 或低階 DynamoDbItemEncryptor，則必須手動定義屬性動作。如果您使用 DynamoDB 增強型用戶端，您可以手動定義屬性動作，也可以使用註釋的資料類別來產生 TableSchema。為了簡化組態程序，我們建議您使用註釋的資料類別。使用註釋的資料類別時，您只需要建立物件模型一次。

> **Note**
>
> 定義屬性動作之後，您必須定義要從簽章中排除哪些屬性。為了在未來更容易新增未簽章的屬性，我們建議您選擇不同的字首（例如 ":") 來識別未簽章的屬性。在標示DO_NOTHING為定義 DynamoDB 結構描述和屬性動作的所有屬性的屬性名稱中包含此字首。

使用註釋的資料類別

使用註釋的資料類別，透過 DynamoDB 增強型用戶端 和 指定屬性動作DynamoDbEncryptionInterceptor。DynamoDB 適用的 AWS Database Encryption SDK 使用標準 DynamoDB 屬性註釋，定義屬性類型，以判斷如何保護屬性。除了主要索引鍵 (簽署但不加密) 以外，所有屬性都預設會進行加密和簽署。

> **Note**
>
> 若要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

如需 DynamoDB 增強型用戶端註釋的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 SimpleClass.java。 DynamoDB

根據預設，主要金鑰屬性會經過簽署但未加密 (SIGN_ONLY)，而所有其他屬性則會經過加密和簽署 (ENCRYPT_AND_SIGN)。如果您將任何屬性定義為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。若要指定例外狀況，請使用 DynamoDB Java 用戶端加密程式庫中定義的加密註釋。例如，如果您希望只簽署特定屬性，請使用 @DynamoDbEncryptionSignOnly註釋。如果您想要在加密內容中簽署並包含特定屬性，請使用 @DynamoDbEncryptionSignAndIncludeInEncryptionContext。如果您希望特定屬性既不簽署也不加密 (DO_NOTHING)，請使用 @DynamoDbEncryptionDoNothing註釋。

> **Note**
>
> AWS Database Encryption SDK 不支援巢狀屬性上的註釋。

下列範例顯示用於定義 ENCRYPT_AND_SIGN、 SIGN_ONLY和 DO_NOTHING屬性動作的註釋。如需顯示用於定義 之註釋的範例SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，請參閱 SimpleClass4.java。

```
@DynamoDbBean
public class SimpleClass {

    private String partitionKey;
    private int sortKey;
    private String attribute1;
    private String attribute2;
    private String attribute3;

    @DynamoDbPartitionKey
    @DynamoDbAttribute(value = "partition_key")
    public String getPartitionKey() {
        return this.partitionKey;
    }

    public void setPartitionKey(String partitionKey) {
        this.partitionKey = partitionKey;
    }

    @DynamoDbSortKey
    @DynamoDbAttribute(value = "sort_key")
    public int getSortKey() {
        return this.sortKey;
    }

    public void setSortKey(int sortKey) {
        this.sortKey = sortKey;
    }

    public String getAttribute1() {
        return this.attribute1;
    }

    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    @DynamoDbEncryptionSignOnly
    public String getAttribute2() {
```

```
        return this.attribute2;
    }

    public void setAttribute2(String attribute2) {
        this.attribute2 = attribute2;
    }

    @DynamoDbEncryptionDoNothing
    public String getAttribute3() {
        return this.attribute3;
    }

    @DynamoDbAttribute(value = ":attribute3")
    public void setAttribute3(String attribute3) {
        this.attribute3 = attribute3;
    }

}
```

使用註釋的資料類別來建立 TableSchema，如下列程式碼片段所示。

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

手動定義您的屬性動作

若要手動指定屬性動作，請建立名稱值對代表屬性名稱和指定動作的Map物件。

指定 ENCRYPT_AND_SIGN來加密和簽署 屬性。指定 SIGN_ONLY簽署但不要加密 屬性。指定 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT來簽署 屬性，並將其包含在加密內容中。您無法在未簽署屬性的情況下加密該屬性。指定 DO_NOTHING 忽略 屬性。

分割區和排序屬性必須是 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您將任何屬性定義為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 若要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀取器SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be signed
attributeActionsOnEncrypt.put("partition_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
// The sort attribute must be signed
attributeActionsOnEncrypt.put("sort_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute3",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put(":attribute4", CryptoAction.DO_NOTHING);
```

DynamoDB AWS 資料庫加密 SDK 中的加密組態

使用 AWS 資料庫加密 SDK 時，您必須明確定義 DynamoDB 資料表的加密組態。加密組態中所需的值取決於您是手動或使用註釋的資料類別來定義屬性動作。

下列程式碼片段使用 DynamoDB 增強型用戶端 `TableSchema`和由不同字首定義的允許未簽章屬性來定義 DynamoDB 資料表加密組態。

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
        DynamoDbEnhancedTableEncryptionConfig.builder()
            .logicalTableName(ddbTableName)
            .keyring(kmsKeyring)
            .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
            .schemaOnEncrypt(tableSchema)
            // Optional: only required if you use beacons
            .search(SearchConfig.builder()
                    .writeVersion(1) // MUST be 1
                    .versions(beaconVersions)
                    .build())
            .build());
```

邏輯資料表名稱

DynamoDB 資料表的邏輯資料表名稱。

邏輯資料表名稱以密碼編譯方式繫結至資料表中存放的所有資料，以簡化 DynamoDB 還原操作。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。您

必須一律指定相同的邏輯資料表名稱。若要成功解密，邏輯資料表名稱必須符合加密時指定的名稱。如果您的 DynamoDB 資料表名稱在從備份還原 DynamoDB 資料表之後變更，邏輯資料表名稱可確保解密操作仍可辨識資料表。

允許的未簽署屬性

在您的屬性動作DO_NOTHING中標記的屬性。

允許的未簽章屬性會告知用戶端哪些屬性會從簽章中排除。用戶端假設所有其他屬性都包含在簽章中。然後，在解密記錄時，用戶端會決定需要驗證哪些屬性，以及要從您指定的允許未簽章屬性中忽略哪些屬性。您無法從允許的未簽章屬性中移除屬性。

您可以建立列出所有屬性的陣列，明確定義允許的未簽章DO_NOTHING屬性。您也可以在命名DO_NOTHING屬性時指定不同的字首，並使用字首告訴用戶端哪些屬性未簽署。我們強烈建議指定不同的字首，因為它可簡化未來新增DO_NOTHING屬性的程序。如需詳細資訊，請參閱更新您的資料模型。

如果您未指定所有DO_NOTHING屬性的字首，您可以設定allowedUnsignedAttributes陣列，明確列出用戶端在解密時應預期會取消簽署的所有屬性。只有在絕對必要時，您才應該明確定義允許的未簽署屬性。

搜尋組態（選用）

SearchConfig 定義信標版本。

SearchConfig 必須指定 才能使用可搜尋加密或簽章的信標。

演算法套件（選用）

algorithmSuiteId 定義 AWS 資料庫加密 SDK 使用的演算法套件。

除非您明確指定替代演算法套件，否則 AWS Database Encryption SDK 會使用預設演算法套件。預設演算法套件使用 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，某些信任模型將由沒有數位簽章的演算法套件所滿足。如需 AWS 資料庫加密 SDK 支援的演算法套件相關資訊，請參閱 AWS Database Encryption SDK 中支援的演算法套件。

若要選取不含 ECDSA 數位簽章的 AES-GCM 演算法套件，請在資料表加密組態中包含下列程式碼片段。

```
.algorithmSuiteId(
```

```
DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

使用 AWS 資料庫加密 SDK 更新項目

AWS Database Encryption SDK 不支援 ddb：UpdateItem 用於已加密或簽署的項目。若要更新加密或簽署的項目，您必須使用 ddb：PutItem。當您在`PutItem`請求中指定與現有項目相同的主索引鍵時，新項目會完全取代現有項目。您也可以在更新項目後，使用 CLOBBER 在儲存時清除和取代所有屬性。

解密已簽章的集合

在 AWS 資料庫加密 SDK 的 3.0.0 和 3.1.0 版中，如果您將集合類型屬性定義為 `SIGN_ONLY`，集合的值會依提供的順序進行正式化。DynamoDB 不會保留集合的順序。因此，包含集合的項目的簽章驗證可能會失敗。簽章驗證會在以與提供給 AWS 資料庫加密 SDK 的順序不同的順序傳回集合的值時失敗，即使集合屬性包含相同的值。

> **Note**
> AWS 資料庫加密 SDK 的 3.1.1 版和更新版本可將所有集合類型屬性的值標準化，以便以寫入 DynamoDB 的相同順序讀取這些值。

如果簽章驗證失敗，解密操作會失敗，並傳回下列錯誤訊息。

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model.StructuredEncrypti
onException：沒有相符的收件人標籤。
```

如果您收到上述錯誤訊息，並認為您嘗試解密的項目包含使用 3.0.0 或 3.1.0 版簽署的集合，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb-java 儲存庫的 DecryptWithPermute 目錄，以取得如何成功驗證集合的詳細資訊。

Java 範例

```
我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供
DynamoDB Encryption Client 的相關資訊。
```

下列範例示範如何使用適用於 DynamoDB 的 Java 用戶端加密程式庫來保護應用程式中的資料表項目。您可以在 GitHub 的 aws-database-encryption-sdk-dynamodb 儲存庫的 Java 範例中找到更多範例（並自行貢獻）。

下列範例示範如何在新的未填入 Amazon DynamoDB 資料表中設定 DynamoDB 的 Java 用戶端加密程式庫。如果您想要設定現有的 Amazon DynamoDB 資料表進行用戶端加密，請參閱 將 3.x 版新增至現有資料表。

主題

- 使用 DynamoDB 增強型用戶端
- 使用低階 DynamoDB API
- 使用較低層級的 DynamoDbItemEncryptor

使用 DynamoDB 增強型用戶端

下列範例示範如何使用 DynamoDB 增強型用戶端和 `DynamoDbEncryptionInterceptor`搭配 AWS KMS keyring 來加密 DynamoDB 資料表項目，做為 DynamoDB API 呼叫的一部分。

您可以搭配 DynamoDB 增強型用戶端使用任何支援的 keyring，但我們建議您盡可能使用其中一個 AWS KMS keyring。

> **Note**
>
> DynamoDB 增強型用戶端不支援可搜尋加密。`DynamoDbEncryptionInterceptor` 搭配低階 DynamoDB API 使用 來使用可搜尋加密。

請參閱完整的程式碼範例：EnhancedPutGetExample.java

步驟 1：建立 AWS KMS keyring

下列範例使用 `CreateAwsKmsMrkMultiKeyring`建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此`CreateAwsKmsMrkMultiKeyring`方法可確保 keyring 可正確處理單一區域和多區域金鑰。

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

```
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步驟 2：從註釋的資料類別建立資料表結構描述

下列範例使用註釋的資料類別來建立 `TableSchema`。

此範例假設已使用 SimpleClass.java 定義註釋的資料類別和屬性動作。如需註釋屬性動作的更多指引，請參閱 使用註釋的資料類別。

> **Note**
>
> AWS Database Encryption SDK 不支援巢狀屬性的註釋。

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

步驟 3：定義從簽章中排除哪些屬性

下列範例假設所有`DO_NOTHING`屬性共用不同的字首 `":"`，並使用字首定義允許的未簽署屬性。用戶端假設任何具有「`:`」字首的屬性名稱都會從簽章中排除。如需詳細資訊，請參閱Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步驟 4：建立加密組態

下列範例定義代表 DynamoDB 資料表加密組態的`tableConfigs`映射。

此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

> **Note**
>
> 若要使用可搜尋加密或簽章的信標，您還必須在加密組態`SearchConfig`中包含 。

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

步驟 5：建立 **DynamoDbEncryptionInterceptor**

下列範例會使用步驟 4 `DynamoDbEncryptionInterceptor``tableConfigs`中的 建立新的 。

```
final DynamoDbEncryptionInterceptor interceptor =
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

步驟 6：建立新的 AWS SDK DynamoDB 用戶端

下列範例使用步驟 5 `interceptor`中的 建立新的 AWS SDK DynamoDB 用戶端。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                       .addExecutionInterceptor(interceptor)
                       .build())
        .build();
```

步驟 7：建立 DynamoDB 增強型用戶端並建立資料表

下列範例使用步驟 6 中建立的 AWS SDK DynamoDB 用戶端建立 DynamoDB 增強型用戶端，並使用註釋的資料類型建立資料表。 DynamoDB

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

步驟 8：加密和簽署資料表項目

下列範例使用 DynamoDB 增強型用戶端將項目放入 DynamoDB 資料表。項目在傳送至 DynamoDB 之前，會先加密並簽署用戶端。

```
final SimpleClass item = new SimpleClass();
item.setPartitionKey("EnhancedPutGetExample");
item.setSortKey(0);
item.setAttribute1("encrypt and sign me!");
item.setAttribute2("sign me!");
item.setAttribute3("ignore me!");

table.putItem(item);
```

使用低階 DynamoDB API

下列範例示範如何搭配 AWS KMS keyring 使用低階 DynamoDB API，以自動加密和簽署 DynamoDB `PutItem`請求的用戶端項目。

您可以使用任何支援的 keyring，但我們建議您盡可能使用其中一個 AWS KMS keyring。

請參閱完整的程式碼範例：BasicPutGetExample.java

步驟 1：建立 AWS KMS keyring

下列範例使用 `CreateAwsKmsMrkMultiKeyring`建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此`CreateAwsKmsMrkMultiKeyring`方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步驟 2：設定屬性動作

下列範例定義代表資料表項目範例屬性動作的`attributeActionsOnEncrypt`映射。

> **Note**
>
> 下列範例不會將任何屬性定義為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

步驟 3：定義從簽章中排除哪些屬性

下列範例假設所有DO_NOTHING屬性共用不同的字首 ":"，並使用字首定義允許的未簽署屬性。用戶端假設任何具有「:」字首的屬性名稱都會從簽章中排除。如需詳細資訊，請參閱Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步驟 4：定義 DynamoDB 資料表加密組態

下列範例定義代表此 DynamoDB 資料表加密組態的tableConfigs映射。

此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

> **Note**
>
> 若要使用可搜尋加密或簽章的信標，您還必須在加密組態SearchConfig中包含 。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
```

```
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
tableConfigs.put(ddbTableName, config);
```

步驟 5：建立 **DynamoDbEncryptionInterceptor**

下列範例`DynamoDbEncryptionInterceptor`使用步驟 4 `tableConfigs`中的 建立 。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

步驟 6：建立新的 AWS SDK DynamoDB 用戶端

下列範例使用步驟 5 `interceptor`中的 建立新的 AWS SDK DynamoDB 用戶端。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

步驟 7：加密和簽署 DynamoDB 資料表項目

下列範例定義代表範例資料表項目的`item`映射，並將該項目放入 DynamoDB 資料表中。項目在傳送至 DynamoDB 之前，會先加密並簽署用戶端。

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("partition_key", AttributeValue.builder().s("BasicPutGetExample").build());
item.put("sort_key", AttributeValue.builder().n("0").build());
item.put("attribute1", AttributeValue.builder().s("encrypt and sign me!").build());
item.put("attribute2", AttributeValue.builder().s("sign me!").build());
item.put(":attribute3", AttributeValue.builder().s("ignore me!").build());
```

```
final PutItemRequest putRequest = PutItemRequest.builder()
        .tableName(ddbTableName)
        .item(item)
        .build();

final PutItemResponse putResponse = ddb.putItem(putRequest);
```

使用較低層級的 DynamoDbItemEncryptor

下列範例示範如何`DynamoDbItemEncryptor`搭配 AWS KMS keyring 使用較低層級 ，以直接加密和簽署資料表項目。`DynamoDbItemEncryptor` 不會將項目放在 DynamoDB 資料表中。

您可以搭配 DynamoDB 增強型用戶端使用任何支援的 keyring，但我們建議您盡可能使用其中一個 AWS KMS keyring。

> **Note**
>
> 較低層級`DynamoDbItemEncryptor`不支援可搜尋加密。搭配`DynamoDbEncryptionInterceptor`低階 DynamoDB API 使用 來使用可搜尋的加密。

請參閱完整的程式碼範例：I ItemEncryptDecryptExample.java

步驟 1：建立 AWS KMS keyring

下列範例使用 `CreateAwsKmsMrkMultiKeyring`建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此`CreateAwsKmsMrkMultiKeyring`方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步驟 2：設定屬性動作

下列範例定義代表資料表項目範例屬性動作的`attributeActionsOnEncrypt`映射。

> **Note**
>
> 下列範例不會將任何屬性定義為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

步驟 3：定義從簽章中排除哪些屬性

下列範例假設所有DO_NOTHING屬性共用不同的字首 ":"，並使用字首定義允許的未簽署屬性。用戶端假設任何具有「:」字首的屬性名稱都會從簽章中排除。如需詳細資訊，請參閱Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步驟 4：定義**DynamoDbItemEncryptor**組態

下列範例定義 的組態DynamoDbItemEncryptor。

此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

步驟 5：建立 **DynamoDbItemEncryptor**

下列範例DynamoDbItemEncryptor使用步驟 4 config中的 建立新的 。

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

步驟 6：直接加密和簽署資料表項目

下列範例使用 直接加密和簽署項目DynamoDbItemEncryptor。DynamoDbItemEncryptor 不會將項目放入 DynamoDB 資料表。

```
final Map<String, AttributeValue> originalItem = new HashMap<>();
originalItem.put("partition_key",
 AttributeValue.builder().s("ItemEncryptDecryptExample").build());
originalItem.put("sort_key", AttributeValue.builder().n("0").build());
originalItem.put("attribute1", AttributeValue.builder().s("encrypt and sign
 me!").build());
originalItem.put("attribute2", AttributeValue.builder().s("sign me!").build());
originalItem.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final Map<String, AttributeValue> encryptedItem = itemEncryptor.EncryptItem(
        EncryptItemInput.builder()
                .plaintextItem(originalItem)
                .build()
).encryptedItem();
```

設定現有的 DynamoDB 資料表以使用 DynamoDB 的 AWS Database Encryption SDK

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

使用適用於 DynamoDB 的 Java 用戶端加密程式庫 3.x 版，您可以設定現有的 Amazon DynamoDB 資料表進行用戶端加密。本主題提供在現有已填入的 DynamoDB 資料表中新增 3.x 版時必須採取的三個步驟指引。

先決條件

DynamoDB 的 Java 用戶端加密程式庫 3.x 版需要 提供的 DynamoDB 增強型用戶端 AWS SDK for Java 2.x 。如果您仍然使用 DynamoDBMapper，您必須遷移至 AWS SDK for Java 2.x ，才能使用 DynamoDB 增強型用戶端。

請遵循從 1.x 版遷移至 2.x 版 適用於 Java 的 AWS SDK的指示。

然後，遵循使用 DynamoDB 增強型用戶端 API 開始使用的指示。

設定資料表以使用 DynamoDB 的 Java 用戶端加密程式庫之前，您需要`TableSchema`使用註釋的資料類別產生 ，並建立增強型用戶端。

步驟 1：準備讀取和寫入加密的項目

請完成下列步驟，以準備您的 AWS Database Encryption SDK 用戶端讀取和寫入加密的項目。部署下列變更之後，用戶端將繼續讀取和寫入純文字項目。它不會加密或簽署寫入資料表的任何新項目，但一旦加密項目出現，它就能夠立即解密。這些變更可讓用戶端開始加密新項目。您必須先將下列變更部署到每個讀取器，才能繼續下一個步驟。

1. 定義您的屬性動作

   更新您的註釋資料類別，以包含屬性動作，這些動作定義哪些屬性值會加密和簽署，哪些值只會簽署，哪些則會忽略。

   如需 DynamoDB 增強型用戶端註釋的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 SimpleClass.java。

   根據預設，主要金鑰屬性會經過簽署但未加密 (`SIGN_ONLY`)，而所有其他屬性則會經過加密和簽署 (`ENCRYPT_AND_SIGN`)。若要指定例外狀況，請使用 DynamoDB 的 Java 用戶端加密程式庫中定義的加密註釋。例如，如果您希望特定屬性僅簽署 ，請使用`@DynamoDbEncryptionSignOnly`註釋。如果您想要在加密內容中簽署並包含特定屬性，請使用`@DynamoDbEncryptionSignAndIncludeInEncryptionContext`註釋。如果您希望特定屬性既不簽署也不加密 (`DO_NOTHING`)，請使用 `@DynamoDbEncryptionDoNothing`註釋。

   > **Note**
   >
   > 如果您指定任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性，則分割區和排序屬性也必須是 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如需顯示用於定義 之註釋的範例`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`，請參閱 SimpleClass4.java。

如需註釋範例，請參閱 使用註釋的資料類別。

2. 定義將從簽章中排除哪些屬性

   下列範例假設所有DO_NOTHING屬性共用不同的字首 ":"，並使用字首定義允許的未簽署屬性。用戶端會假設簽章中排除任何具有「:」字首的屬性名稱。如需詳細資訊，請參閱Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

3. 建立 keyring

   下列範例會建立 AWS KMS keyring。 AWS KMS Keyring 使用對稱加密或非對稱 RSA AWS KMS keys 來產生、加密和解密資料金鑰。

   此範例使用 CreateMrkMultiKeyring建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此CreateAwsKmsMrkMultiKeyring方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. 定義 DynamoDB 資料表加密組態

   下列範例定義代表此 DynamoDB 資料表加密組態的tableConfigs映射。

   此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

   您必須指定 FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT做為純文字覆寫。此政策會繼續讀取和寫入純文字項目、讀取加密的項目，以及準備用戶端寫入加密的項目。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
```

```
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

5. 建立 **DynamoDbEncryptionInterceptor**

   下列範例DynamoDbEncryptionInterceptor使用步驟 3 tableConfigs中的 建立 。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

步驟 2：寫入加密和簽章的項目

在您的DynamoDbEncryptionInterceptor組態中更新純文字政策，以允許用戶端寫入加密和簽署的項目。部署下列變更之後，用戶端會根據您在步驟 1 中設定的屬性動作來加密和簽署新項目。用戶端將能夠讀取純文字項目，以及加密和簽署的項目。

在繼續步驟 3 之前，您必須加密和簽署資料表中的所有現有純文字項目。您可以執行沒有單一指標或查詢，以快速加密現有的純文字項目。使用對您的系統最有意義的程序。例如，您可以使用慢速掃描資料表的非同步程序，並使用您定義的屬性動作和加密組態重寫項目。若要識別資料表中的純文字項目，建議您掃描所有不包含 AWS Database Encryption SDK 在加密和簽署項目時新增至項目的 aws_dbe_head 和 aws_dbe_foot 屬性的項目。

下列範例會從步驟 1 更新資料表加密組態。您必須使用 更新純文字覆寫FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT。此政策會繼續讀取純文字項目，但也會讀取和寫入加密的項目。DynamoDbEncryptionInterceptor 使用更新的 建立新的 tableConfigs。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
```

```
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

步驟 3：僅讀取加密和簽章的項目

在您加密並簽署所有項目之後，請更新DynamoDbEncryptionInterceptor組態中的純文字覆寫，以僅允許用戶端讀取和寫入加密和簽署的項目。部署下列變更之後，用戶端會根據您在步驟 1 中設定的屬性動作來加密和簽署新項目。用戶端只能讀取加密和簽章的項目。

下列範例會從步驟 2 更新資料表加密組態。您可以使用 更新純文字覆寫，FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT或從組態中移除純文字政策。根據預設，用戶端只會讀取和寫入加密和簽署的項目。DynamoDbEncryptionInterceptor 使用更新的 建立新的 tableConfigs。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        // Optional: you can also remove the plaintext policy from your configuration

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

遷移至 DynamoDB Java 用戶端加密程式庫的 3.x 版

> 我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

DynamoDB 的 Java 用戶端加密程式庫 3.x 版是 2.x 程式碼基礎的主要重寫。它包含許多更新，例如新的結構化資料格式、改善的多租戶支援、無縫結構描述變更，以及可搜尋的加密支援。本主題提供如何將程式碼遷移至 3.x 版的指引。

從 1.x 版遷移至 2.x

遷移至 2.x 版後，再遷移至 3.x 版。2.x 版將最近提供者的 符號從 變更為 `MostRecentProvider` `CachingMostRecentProvider`。如果您目前使用適用於 DynamoDB 的 Java 用戶端加密程式庫版本 1.x 搭配 `MostRecentProvider`符號，則必須將程式碼中的符號名稱更新為 `CachingMostRecentProvider`。如需詳細資訊，請參閱更新至最新的提供者。

從 2.x 版遷移至 3.x

下列程序說明如何將程式碼從 2.x 版遷移至 DynamoDB Java 用戶端加密程式庫的 3.x 版。

步驟 1. 準備讀取新格式的項目

請完成下列步驟，以準備您的 AWS Database Encryption SDK 用戶端以讀取新格式的項目。部署下列變更之後，用戶端的行為方式會繼續與 2.x 版相同。您的用戶端會繼續讀取和寫入 2.x 版格式的項目，但這些變更可讓用戶端準備好讀取新格式的項目。

將 更新 適用於 Java 的 AWS SDK 至 2.x 版

DynamoDB 的 Java 用戶端加密程式庫版本 3.x 需要 DynamoDB 增強型用戶端。DynamoDB 增強型用戶端會取代先前版本中使用的 DynamoDBMapper。若要使用增強型用戶端，您必須使用 AWS SDK for Java 2.x。

遵循從 1.x 版遷移至 2.x 版 適用於 Java 的 AWS SDK的指示。

如需需要哪些 AWS SDK for Java 2.x 模組的詳細資訊，請參閱先決條件。

將用戶端設定為讀取舊版加密的項目

下列程序提供以下程式碼範例中所示範步驟的概觀。

1. 建立 keyring。

   Keyrings 和密碼編譯資料管理員會取代 DynamoDB 的 Java 用戶端加密程式庫先前版本中使用的密碼編譯資料提供者。

   > **Important**
   >
   > 您在建立 keyring 時指定的包裝金鑰，必須與 2.x 版中與密碼編譯資料提供者搭配使用的包裝金鑰相同。

2. 在註釋的類別上建立資料表結構描述。

此步驟定義了當您開始以新格式撰寫項目時將使用的屬性動作。

如需有關使用新 DynamoDB 增強型用戶端的指引，請參閱《 適用於 Java 的 AWS SDK 開發人員指南》中的產生 `TableSchema`。

下列範例假設您使用新的屬性動作註釋，從 2.x 版更新了註釋類別。如需註釋屬性動作的更多指引，請參閱 使用註釋的資料類別。

> **Note**
>
> 如果您指定任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性，則分割區和排序屬性也必須是 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如需顯示用於定義 之註釋的範例`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`，請參閱 SimpleClass4.java。

3. 定義要從簽章中排除哪些屬性。
4. 設定 2.x 版模型化類別中設定的屬性動作的明確對應。

   此步驟會定義用來以舊格式寫入項目的屬性動作。

5. 設定`DynamoDBEncryptor`您在適用於 DynamoDB 的 Java 用戶端加密程式庫 2.x 版中使用的 。
6. 設定舊版行為。
7. 建立 `DynamoDbEncryptionInterceptor`。
8. 建立新的 AWS SDK DynamoDB 用戶端。
9. 建立 `DynamoDBEnhancedClient`並使用模型化類別建立資料表。

   如需 DynamoDB 增強型用戶端的詳細資訊，請參閱建立增強型用戶端。

```
public class MigrationExampleStep1 {

    public static void MigrationStep1(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Create a Keyring.
        //    This example creates an AWS KMS Keyring that specifies the
        //    same kmsKeyId previously used in the version 2.x configuration.
        //    It uses the 'CreateMrkMultiKeyring' method to create the
        //    keyring, so that the keyring can correctly handle both single
```

```
        //    region and Multi-Region KMS Keys.
        //    Note that this example uses the AWS SDK for Java v2 KMS client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        // 2. Create a Table Schema over your annotated class.
        //    For guidance on using the new attribute actions
        //    annotations, see SimpleClass.java in the
        //    aws-database-encryption-sdk-dynamodb GitHub repository.
        //    All primary key attributes must be signed but not encrypted
        //    and by default all non-primary key attributes
        //    are encrypted and signed (ENCRYPT_AND_SIGN).
        //    If you want a particular non-primary key attribute to be signed but
        //    not encrypted, use the 'DynamoDbEncryptionSignOnly' annotation.
        //    If you want a particular attribute to be neither signed nor encrypted
        //    (DO_NOTHING), use the 'DynamoDbEncryptionDoNothing' annotation.
        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        // 3. Define which attributes the client should expect to be excluded
        //    from the signature when reading items.
        //    This value represents all unsigned attributes across the entire
        //    dataset.
        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        // 4. Configure an explicit map of the attribute actions configured
        //    in your version 2.x modeled class.
        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        // 5. Configure the DynamoDBEncryptor that you used in version 2.x.
        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
```

```
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 6. Configure the legacy behavior.
        //    Input the DynamoDBEncryptor and attribute actions created in
        //    the previous steps. For Legacy Policy, use
        //    'FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This policy continues to
read
        //    and write items using the old format, but will be able to read
        //    items written in the new format as soon as they appear.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 7. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 8. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 7.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 9. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb client
```

```
        //    created in Step 8, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

步驟 2. 以新格式寫入項目

將步驟 1 的變更部署到所有讀取器之後，請完成下列步驟，以設定您的 AWS Database Encryption SDK 用戶端以新格式寫入項目。部署下列變更後，用戶端將繼續以舊格式讀取項目，並開始以新格式寫入和讀取項目。

下列程序提供以下程式碼範例中所示範步驟的概觀。

1. 繼續設定 keyring、資料表結構描述、舊版屬性動作 和 `allowedUnsignedAttributes`，`DynamoDBEncryptor`如您在步驟 1 中所執行。
2. 更新您的舊版行為，只使用新格式撰寫新項目。
3. 建立 `DynamoDbEncryptionInterceptor`
4. 建立新的 AWS SDK DynamoDB 用戶端。
5. 建立 `DynamoDBEnhancedClient`並使用模型化類別建立資料表。

   如需 DynamoDB 增強型用戶端的詳細資訊，請參閱建立增強型用戶端。

```
public class MigrationExampleStep2 {

    public static void MigrationStep2(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema, legacy
        //    attribute actions, allowedUnsignedAttributes, and
        //    DynamoDBEncryptor as you did in Step 1.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
```

```
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 2. Update your legacy behavior to only write new items using the new
        //    format.
        //    For Legacy Policy, use 'FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This
policy
        //    continues to read items in both formats, but will only write items
        //    using the new format.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
```

```
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb Client
 created
        //    in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

部署步驟 2 變更後，您必須使用新格式重新加密資料表中的所有舊項目，才能繼續執行步驟 3。沒有您可以執行的單一指標或查詢，以快速加密現有的項目。使用對您的系統最有意義的程序。例如，您可以使用慢速掃描資料表的非同步程序，並使用您定義的新屬性動作和加密組態重寫項目。

步驟 3。僅讀取和寫入新格式的項目

使用新格式重新加密資料表中的所有項目之後，您可以從組態中移除舊版行為。請完成下列步驟，將用戶端設定為僅讀取和寫入新格式的項目。

下列程序提供以下程式碼範例中所示範步驟的概觀。

1. 繼續設定 keyring、資料表結構描述，以及`allowedUnsignedAttributes`如您在步驟 1 中所執行。`DynamoDBEncryptor` 從組態中移除舊版屬性動作 和 。
2. 建立 `DynamoDbEncryptionInterceptor`。
3. 建立新的 AWS SDK DynamoDB 用戶端。
4. 建立 `DynamoDBEnhancedClient`並使用模型化類別建立資料表。

如需 DynamoDB 增強型用戶端的詳細資訊，請參閱建立增強型用戶端。

```
public class MigrationExampleStep3 {

    public static void MigrationStep3(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema,
        //    and allowedUnsignedAttributes as you did in Step 1.
        //    Do not include the configurations for the DynamoDBEncryptor or
        //    the legacy attribute actions.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");


        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        //    Do not configure any legacy behavior.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );
```

```
        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK Client
        //    created in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

.NET

本主題說明如何安裝和使用適用於 DynamoDB 的 .NET 用戶端加密程式庫的 3.x 版。如需使用適用於 DynamoDB 的 AWS Database Encryption SDK 進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 .NET 範例。

DynamoDB 的 .NET 用戶端加密程式庫適用於以 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。

DynamoDB AWS 資料庫加密 SDK 的所有程式設計語言實作皆可互通。不過， 適用於 .NET 的 SDK 不支援清單或映射資料類型的空白值。這表示如果您使用適用於 DynamoDB 的 Java 用戶端加密程式庫來撰寫包含清單或映射資料類型空白值的項目，則無法使用適用於 DynamoDB 的 .NET 用戶端加密程式庫解密和讀取該項目。

主題

- 安裝適用於 DynamoDB 的 .NET 用戶端加密程式庫
- 使用 .NET 進行偵錯
- 使用適用於 DynamoDB 的 .NET 用戶端加密程式庫
- .NET 範例
- 設定現有的 DynamoDB 資料表以使用 DynamoDB 的 AWS Database Encryption SDK

安裝適用於 DynamoDB 的 .NET 用戶端加密程式庫

DynamoDB 的 .NET 用戶端加密程式庫可在 NuGet 中做為 AWS.Cryptography.DbEncryptionSDK.DynamoDb 套件使用。如需安裝和建置程式庫的詳細資訊，請參閱 aws-database-encryption-sdk-dynamodb 儲存庫中的 .NET README.md 檔案。 適用於 .NET 的 SDK 即使您未使用 AWS Key Management Service (AWS KMS) 金鑰，DynamoDB 的 .NET 用戶端加密程式庫仍需要 。會與 NuGet 適用於 .NET 的 SDK 套件一起安裝。

DynamoDB 的 .NET 用戶端加密程式庫版本 3.x 支援 .NET 6.0 和 .NET Framework net48 及更新版本。

使用 .NET 進行偵錯

DynamoDB 的 .NET 用戶端加密程式庫不會產生任何日誌。DynamoDB 的 .NET 用戶端加密程式庫中的例外狀況會產生例外狀況訊息，但沒有堆疊追蹤。

為了協助您偵錯，請務必在 中啟用記錄 適用於 .NET 的 SDK。的日誌和錯誤訊息 適用於 .NET 的 SDK 可協助您區分 中產生的錯誤 適用於 .NET 的 SDK 與 DynamoDB 的 .NET 用戶端加密程式庫中的錯誤。如需 適用於 .NET 的 SDK 記錄的說明，請參閱《 適用於 .NET 的 AWS SDK 開發人員指南》中的 AWSLogging。（若要查看主題，請展開開啟以檢視 .NET Framework 內容區段。）

使用適用於 DynamoDB 的 .NET 用戶端加密程式庫

本主題說明 DynamoDB 的 .NET 用戶端加密程式庫 3.x 版中的一些函數和協助程式類別。

如需使用適用於 DynamoDB 的 .NET 用戶端加密程式庫進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 .NET 範例。

主題

- 項目加密程式
- DynamoDB AWS 資料庫加密 SDK 中的屬性動作
- DynamoDB AWS 資料庫加密 SDK 中的加密組態
- 使用 AWS 資料庫加密 SDK 更新項目

項目加密程式

DynamoDB AWS 資料庫加密 SDK 的核心是項目加密程式。您可以使用適用於 DynamoDB 的 .NET 用戶端加密程式庫 3.x 版，以下列方式加密、簽署、驗證和解密 DynamoDB 資料表項目。

適用於 DynamoDB API 的低階 AWS 資料庫加密 SDK

您可以使用資料表加密組態來建構 DynamoDB 用戶端，該用戶端會自動加密和簽署具有 DynamoDB `PutItem`請求的用戶端項目。您可以直接使用此用戶端，也可以建構文件模型或物件持久性模型。

您必須使用 DynamoDB API 的低階 AWS 資料庫加密 SDK，才能使用可搜尋的加密。

較低層級 **`DynamoDbItemEncryptor`**

較低層級會`DynamoDbItemEncryptor`直接加密和簽署或解密，並驗證您的資料表項目，而無需呼叫 DynamoDB。它不會發出 DynamoDB `PutItem`或 `GetItem` 請求。例如，您可以使用較低層級`DynamoDbItemEncryptor`直接解密和驗證您已擷取的 DynamoDB 項目。如果您使用較低層級 `DynamoDbItemEncryptor`，我們建議您使用 適用於 .NET 的 SDK 提供的低階程式設計模型，以與 DynamoDB 通訊。

較低層級`DynamoDbItemEncryptor`不支援可搜尋加密。

DynamoDB AWS 資料庫加密 SDK 中的屬性動作

屬性動作會決定哪些屬性值是加密和簽署的，哪些是僅簽署的，哪些是簽署和包含在加密內容中，哪些是被忽略的。

若要使用 .NET 用戶端指定屬性動作，請使用 物件模型手動定義屬性動作。建立`Dictionary`物件，其中名稱值對代表屬性名稱和指定的動作，以指定您的屬性動作。

指定 `ENCRYPT_AND_SIGN`來加密和簽署 屬性。指定 `SIGN_ONLY`來簽署 屬性，但不加密。指定 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`以簽署 屬性，並將其包含在加密內容中。您無法在未同時簽署屬性的情況下加密屬性。指定 `DO_NOTHING` 忽略 屬性。

分割區和排序屬性必須是 `SIGN_ONLY`或 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如果您將任何屬性定義為 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`，則分割區和排序屬性也必須是 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

> **Note**
>
> 定義屬性動作之後，您必須定義要從簽章中排除哪些屬性。為了在未來更容易新增未簽章的屬性，我們建議您選擇不同的字首 （例如 `":"`) 來識別未簽章的屬性。在標示`DO_NOTHING`為定義 DynamoDB 結構描述和屬性動作的所有屬性的屬性名稱中包含此字首。

下列物件模型示範如何使用 .NET ENCRYPT_AND_SIGN SIGN_ONLY用戶端指定 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、 和 DO_NOTHING 屬性動作。此範例使用字首 ":" 來識別DO_NOTHING屬性。

> **Note**
>
> 若要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT密碼編譯動作，您必須使用 3.3 版或更新版本的 AWS 資料庫加密 SDK。將您的資料模型更新為包含 之前，先將新版本部署至所有讀取器SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The
 partition attribute must be signed
    ["sort_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The sort
 attribute must be signed
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    ["attribute3"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT,
    [":attribute4"] = CryptoAction.DO_NOTHING
};
```

DynamoDB AWS 資料庫加密 SDK 中的加密組態

使用 AWS 資料庫加密 SDK 時，您必須明確定義 DynamoDB 資料表的加密組態。加密組態中所需的值取決於您是手動或使用註釋的資料類別來定義屬性動作。

以下程式碼片段使用 DynamoDB API 的低階 AWS 資料庫加密 SDK 定義 DynamoDB 資料表加密組態，並允許由不同字首定義的未簽章屬性。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
```

```
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: SearchConfig only required if you use beacons
    Search = new SearchConfig
    {
        WriteVersion = 1, // MUST be 1
        Versions = beaconVersions
    }
};
tableConfigs.Add(ddbTableName, config);
```

邏輯資料表名稱

DynamoDB 資料表的邏輯資料表名稱。

邏輯資料表名稱以密碼編譯方式繫結至資料表中存放的所有資料，以簡化 DynamoDB 還原操作。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。您必須一律指定相同的邏輯資料表名稱。若要成功解密，邏輯資料表名稱必須符合加密時指定的名稱。如果您的 DynamoDB 資料表名稱在從備份還原 DynamoDB 資料表之後變更，邏輯資料表名稱可確保解密操作仍可辨識資料表。

允許的未簽署屬性

在您的屬性動作DO_NOTHING中標記的屬性。

允許的未簽章屬性會告知用戶端哪些屬性會從簽章中排除。用戶端假設所有其他屬性都包含在簽章中。然後，在解密記錄時，用戶端會決定需要驗證哪些屬性，以及要從您指定的允許未簽章屬性中忽略哪些屬性。您無法從允許的未簽章屬性中移除屬性。

您可以建立列出所有屬性的陣列，明確定義允許的未簽章DO_NOTHING屬性。您也可以在命名DO_NOTHING屬性時指定不同的字首，並使用字首告訴用戶端哪些屬性未簽署。我們強烈建議指定不同的字首，因為它可簡化未來新增DO_NOTHING屬性的程序。如需詳細資訊，請參閱更新您的資料模型。

如果您未指定所有DO_NOTHING屬性的字首，您可以設定allowedUnsignedAttributes陣列，明確列出用戶端在解密時應預期取消簽署的所有屬性。只有在絕對必要時，您才應該明確定義允許的未簽署屬性。

搜尋組態（選用）

SearchConfig 定義信標版本。

SearchConfig 必須指定 才能使用可搜尋加密或簽章的信標。

演算法套件（選用）

`algorithmSuiteId` 定義 AWS 資料庫加密 SDK 使用的演算法套件。

除非您明確指定替代演算法套件，否則 AWS Database Encryption SDK 會使用預設演算法套件。預設演算法套件使用 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，某些信任模型將由沒有數位簽章的演算法套件所滿足。如需 AWS 資料庫加密 SDK 支援的演算法套件相關資訊，請參閱 AWS Database Encryption SDK 中支援的演算法套件。

若要選取不含 ECDSA 數位簽章的 AES-GCM 演算法套件，請在資料表加密組態中包含下列程式碼片段。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

使用 AWS 資料庫加密 SDK 更新項目

AWS Database Encryption SDK 不支援包含加密或簽章屬性的項目 ddb：UpdateItem。若要更新加密或簽章的屬性，您必須使用 ddb：PutItem。當您在`PutItem`請求中指定與現有項目相同的主索引鍵時，新項目會完全取代現有項目。您也可以在更新項目後，使用 CLOBBER 在儲存時清除和取代所有屬性。

.NET 範例

下列範例示範如何使用適用於 DynamoDB 的 .NET 用戶端加密程式庫來保護應用程式中的資料表項目。若要尋找更多範例（並自行提供），請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 .NET 範例。

下列範例示範如何在新的未填入 Amazon DynamoDB 資料表中設定 DynamoDB 的 .NET 用戶端加密程式庫。如果您想要設定現有的 Amazon DynamoDB 資料表進行用戶端加密，請參閱 將 3.x 版新增至現有資料表。

主題

- 使用適用於 DynamoDB API 的低階 AWS 資料庫加密 SDK
- 使用較低層級 DynamoDbItemEncryptor

使用適用於 DynamoDB API 的低階 AWS 資料庫加密 SDK

下列範例示範如何使用 DynamoDB API 的低階 AWS 資料庫加密 SDK 搭配 AWS KMS keyring，以自動加密 DynamoDB `PutItem`請求的用戶端項目並簽署項目。

您可以使用任何支援的 keyring，但我們建議您盡可能使用其中一個 AWS KMS keyring。

請參閱完整的程式碼範例：BasicPutGetExample.cs

步驟 1：建立 AWS KMS keyring

下列範例使用 `CreateAwsKmsMrkMultiKeyring`建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此`CreateAwsKmsMrkMultiKeyring`方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步驟 2：設定屬性動作

下列範例定義`attributeActionsOnEncrypt`字典，代表資料表項目的範例屬性動作。

> **Note**
>
> 下列範例不會將任何屬性定義為 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如果您指定任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性，則分割區和排序屬性也必須是 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

步驟 3：定義從簽章中排除哪些屬性

下列範例假設所有DO_NOTHING屬性共用不同的字首 ":"，並使用字首定義允許的未簽署屬性。用戶端假設簽章中排除任何具有「:」字首的屬性名稱。如需詳細資訊，請參閱Allowed unsigned attributes。

```
const String unsignAttrPrefix = ":";
```

步驟 4：定義 DynamoDB 資料表加密組態

下列範例定義代表此 DynamoDB 資料表加密組態的tableConfigs映射。

此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

> **Note**
>
> 若要使用可搜尋加密或簽章的信標，您還必須在加密組態SearchConfig中包含 。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
tableConfigs.Add(ddbTableName, config);
```

步驟 5：建立新的 AWS SDK DynamoDB 用戶端

下列範例使用步驟 4 TableEncryptionConfigs中的 建立新的 AWS SDK DynamoDB 用戶端。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

步驟 6：加密和簽署 DynamoDB 資料表項目

下列範例定義一個字典`item`，代表範例資料表項目，並將該項目放入 DynamoDB 資料表中。項目在傳送至 DynamoDB 之前，會先加密並簽署用戶端。

```
var item = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("BasicPutGetExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

PutItemRequest putRequest = new PutItemRequest
{
    TableName = ddbTableName,
    Item = item
};

PutItemResponse putResponse = await ddb.PutItemAsync(putRequest);
```

使用較低層級 `DynamoDbItemEncryptor`

下列範例示範如何使用較低層級 `DynamoDbItemEncryptor`搭配 AWS KMS keyring 來直接加密和簽署資料表項目。`DynamoDbItemEncryptor` 不會將項目放入 DynamoDB 資料表。

您可以搭配 DynamoDB 增強型用戶端使用任何支援的 keyring，但我們建議您盡可能使用其中一個 AWS KMS keyring。

> **Note**
>
> 較低層級`DynamoDbItemEncryptor`不支援可搜尋加密。使用適用於 DynamoDB API 的低階 AWS 資料庫加密 SDK 來使用可搜尋加密。

請參閱完整的程式碼範例：I ItemEncryptDecryptExample.cs

步驟 1：建立 AWS KMS keyring

下列範例使用 `CreateAwsKmsMrkMultiKeyring`建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此`CreateAwsKmsMrkMultiKeyring`方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步驟 2：設定屬性動作

下列範例定義`attributeActionsOnEncrypt`字典，代表資料表項目的範例屬性動作。

> **Note**
>
> 下列範例不會將任何屬性定義為 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如果您指定任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`屬性，則分割區和排序屬性也必須是 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

```
var attributeActionsOnEncrypt = new Dictionary<String, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

步驟 3：定義從簽章中排除哪些屬性

下列範例假設所有`DO_NOTHING`屬性共用不同的字首 `":"`，並使用字首定義允許的未簽署屬性。用戶端假設簽章中排除任何具有「`:`」字首的屬性名稱。如需詳細資訊，請參閱Allowed unsigned attributes。

```
String unsignAttrPrefix = ":";
```

步驟 4：定義DynamoDbItemEncryptor組態

下列範例定義 的組態DynamoDbItemEncryptor。

此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

步驟 5：建立 DynamoDbItemEncryptor

下列範例DynamoDbItemEncryptor使用步驟 4 config中的 建立新的 。

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

步驟 6：直接加密和簽署資料表項目

下列範例使用 直接加密和簽署項目DynamoDbItemEncryptor。DynamoDbItemEncryptor 不會將項目放在 DynamoDB 資料表中。

```
var originalItem = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("ItemEncryptDecryptExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

var encryptedItem = itemEncryptor.EncryptItem(
    new EncryptItemInput { PlaintextItem = originalItem }
).EncryptedItem;
```

設定現有的 DynamoDB 資料表以使用 DynamoDB 的 AWS Database Encryption SDK

使用適用於 DynamoDB 的 .NET 用戶端加密程式庫 3.x 版，您可以設定現有的 Amazon DynamoDB 資料表進行用戶端加密。本主題提供您必須採取的三個步驟指引，以將 3.x 版新增至現有填入的 DynamoDB 資料表。

步驟 1：準備讀取和寫入加密的項目

請完成下列步驟，以準備您的 AWS Database Encryption SDK 用戶端讀取和寫入加密的項目。部署下列變更之後，用戶端將繼續讀取和寫入純文字項目。它不會加密或簽署寫入資料表的任何新項目，但一旦加密項目出現，它就能夠立即解密。這些變更可讓用戶端開始加密新項目。您必須先將下列變更部署到每個讀取器，才能繼續下一個步驟。

1. 定義您的屬性動作

   建立物件模型，以定義哪些屬性值將加密和簽署，哪些將僅簽署，哪些將忽略。

   根據預設，主要金鑰屬性會經過簽署但未加密 (SIGN_ONLY)，而所有其他屬性則會經過加密和簽署 (ENCRYPT_AND_SIGN)。

   指定 ENCRYPT_AND_SIGN來加密和簽署 屬性。指定 SIGN_ONLY 來簽署但不要加密 屬性。指定 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 來簽署和屬性，並將其包含在加密內容中。如果沒有同時簽署屬性，則無法加密該屬性。指定 DO_NOTHING 忽略 屬性。如需詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的屬性動作。

   > **Note**
   >
   > 如果您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT屬性，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

2. 定義將從簽章中排除哪些屬性

   下列範例假設所有DO_NOTHING屬性共用不同的字首 ":"，並使用字首定義允許的未簽署屬性。用戶端會假設簽章中排除任何具有「:」字首的屬性名稱。如需詳細資訊，請參閱Allowed unsigned attributes。

```
const String unsignAttrPrefix = ":";
```

3. 建立 keyring

   下列範例會建立 AWS KMS keyring。 AWS KMS Keyring 使用對稱加密或非對稱 RSA AWS KMS keys 來產生、加密和解密資料金鑰。

   此範例使用 CreateMrkMultiKeyring建立具有對稱加密 KMS 金鑰的 AWS KMS keyring。此CreateAwsKmsMrkMultiKeyring方法可確保 keyring 可正確處理單一區域和多區域金鑰。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. 定義 DynamoDB 資料表加密組態

   下列範例定義代表此 DynamoDB 資料表加密組態的tableConfigs映射。

   此範例指定 DynamoDB 資料表名稱做為邏輯資料表名稱。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。

   您必須指定 FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT做為純文字覆寫。此政策會繼續讀取和寫入純文字項目、讀取加密的項目，以及準備用戶端寫入加密的項目。

   如需資料表加密組態中包含之值的詳細資訊，請參閱DynamoDB AWS 資料庫加密 SDK 中的加密組態。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
```

```
    PlaintextOverride = FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

5. 建立新的 AWS SDK DynamoDB 用戶端

   他遵循範例，使用步驟 4 `TableEncryptionConfigs`中的 建立新的 AWS SDK DynamoDB 用戶端。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

步驟 2：寫入加密和簽章的項目

更新資料表加密組態中的純文字政策，以允許用戶端寫入加密和簽署的項目。部署下列變更之後，用戶端會根據您在步驟 1 中設定的屬性動作來加密和簽署新項目。用戶端將能夠讀取純文字項目，以及加密和簽署的項目。

在繼續步驟 3 之前，您必須加密和簽署資料表中的所有現有純文字項目。您可以執行沒有單一指標或查詢，以快速加密現有的純文字項目。使用對您的系統最有意義的程序。例如，您可以使用慢速掃描資料表的非同步程序，並使用您定義的屬性動作和加密組態重寫項目。若要識別資料表中的純文字項目，建議您掃描所有不包含 AWS Database Encryption SDK 在加密和簽署項目時新增至項目的 `aws_dbe_head` 和 `aws_dbe_foot` 屬性的項目。

下列範例會從步驟 1 更新資料表加密組態。您必須使用 更新純文字覆寫`FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`。此政策會繼續讀取純文字項目，但也會讀取和寫入加密的項目。使用更新的 建立新的 AWS SDK DynamoDB 用戶端`TableEncryptionConfigs`。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
```

```
tableConfigs.Add(ddbTableName, config);
```

步驟 3：僅讀取加密和簽章的項目

在您加密並簽署所有項目之後，請更新資料表加密組態中的純文字覆寫，以僅允許用戶端讀取和寫入加密和簽署的項目。部署下列變更之後，用戶端會根據您在步驟 1 中設定的屬性動作來加密和簽署新項目。用戶端只能讀取加密和簽章的項目。

下列範例會從步驟 2 更新資料表加密組態。您可以使用 更新純文字覆寫，`FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT`或從組態中移除純文字政策。根據預設，用戶端只會讀取和寫入加密和簽署的項目。使用更新的 建立新的 AWS SDK DynamoDB 用戶端`TableEncryptionConfigs`。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: you can also remove the plaintext policy from your configuration
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Rust

本主題說明如何安裝和使用適用於 DynamoDB 的 Rust 用戶端加密程式庫的 1.x 版。如需使用適用於 DynamoDB 的 AWS Database Encryption SDK 進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Rust 範例。

DynamoDB AWS 資料庫加密 SDK 的所有程式設計語言實作皆可互通。

主題

- 先決條件
- 安裝
- 使用適用於 DynamoDB 的 Rust 用戶端加密程式庫

先決條件

在安裝 DynamoDB 的 Rust 用戶端加密程式庫之前，請確定您有下列先決條件。

安裝 Rust 和 Cargo

使用 rustup 安裝目前穩定的 Rust 版本。

如需下載和安裝中斷的詳細資訊，請參閱 Cargo Book 中的安裝程序。

安裝

適用於 DynamoDB 的 Rust 用戶端加密程式庫可在 Crates.io：// 上做為 aws-db-esdk 條板箱使用。如需安裝和建置程式庫的詳細資訊，請參閱 aws-database-encryption-sdk-dynamodb GitHub 儲存庫中的 README.md 檔案。

手動

若要安裝 DynamoDB 的 Rust 用戶端加密程式庫，請複製或下載 aws-database-encryption-sdk-dynamodb GitHub 儲存庫。

若要安裝最新版本

在專案目錄中執行下列 Cargo 命令：

```
cargo add aws-db-esdk
```

或將以下行新增至您的 Cargo.toml：

```
aws-db-esdk = "<version>"
```

使用適用於 DynamoDB 的 Rust 用戶端加密程式庫

本主題說明 DynamoDB 的 Rust 用戶端加密程式庫 1.x 版中的一些函數和協助程式類別。

如需使用 DynamoDB 的 Rust 用戶端加密程式庫進行程式設計的詳細資訊，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 Rust 範例。

主題

- 項目加密程式

- DynamoDB AWS 資料庫加密 SDK 中的屬性動作
- DynamoDB AWS 資料庫加密 SDK 中的加密組態
- 使用 AWS 資料庫加密 SDK 更新項目

項目加密程式

DynamoDB 的 AWS Database Encryption SDK 本身是項目加密程式。您可以使用適用於 DynamoDB 的 Rust 用戶端加密程式庫 1.x 版，以下列方式加密、簽署、驗證和解密 DynamoDB 資料表項目。

適用於 DynamoDB API 的低階 AWS 資料庫加密 SDK

您可以使用資料表加密組態來建構 DynamoDB 用戶端，該用戶端會自動加密和簽署具有 DynamoDB `PutItem`請求的用戶端項目。

您必須使用 DynamoDB API 的低階 AWS 資料庫加密 SDK，才能使用可搜尋的加密。

如需示範如何使用 DynamoDB API 的低階 AWS 資料庫加密 SDK 的範例，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 basic_get_put_example.rs。

較低層級 **`DynamoDbItemEncryptor`**

較低層級會`DynamoDbItemEncryptor`直接加密和簽署或解密，並驗證您的資料表項目，而無需呼叫 DynamoDB。它不會發出 DynamoDB `PutItem`或 `GetItem` 請求。例如，您可以使用較低層級`DynamoDbItemEncryptor`直接解密和驗證您已擷取的 DynamoDB 項目。

較低層級`DynamoDbItemEncryptor`不支援可搜尋的加密。

如需示範如何使用較低層級 的範例`DynamoDbItemEncryptor`，請參閱 GitHub 上 aws-database-encryption-sdk-dynamodb 儲存庫中的 item_encrypt_decrypt.rs。

DynamoDB AWS 資料庫加密 SDK 中的屬性動作

屬性動作會決定哪些屬性值會加密和簽署、哪些是僅簽署、哪些是簽署和包含在加密內容中，以及哪些是忽略的。

若要使用 Rust 用戶端指定屬性動作，請使用 物件模型手動定義屬性動作。建立`HashMap`物件，其中名稱值對代表屬性名稱和指定的動作，以指定您的屬性動作。

指定 `ENCRYPT_AND_SIGN`來加密和簽署 屬性。指定 `SIGN_ONLY` 來簽署但不要加密 屬性。指定 `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`來簽署 屬性，並將其包含在加密內容中。如果沒有同時簽署屬性，則無法加密該屬性。指定 `DO_NOTHING` 忽略 屬性。

分割區和排序屬性必須是 SIGN_ONLY或 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您將任何屬性定義為 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，則分割區和排序屬性也必須是 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 定義屬性動作之後，您必須定義從簽章中排除哪些屬性。為了在未來更容易新增未簽章的屬性，我們建議您選擇不同的字首（例如 ":") 來識別未簽章的屬性。在標示DO_NOTHING為定義 DynamoDB 結構描述和屬性動作的所有屬性的屬性名稱中包含此字首。

下列物件模型示範如何使用 Rust ENCRYPT_AND_SIGN SIGN_ONLY用戶端指定 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、 和 DO_NOTHING 屬性動作。此範例使用字首 ":" 來識別DO_NOTHING屬性。

```
let attribute_actions_on_encrypt = HashMap::from([
    ("partition_key".to_string(), CryptoAction::SignOnly),
    ("sort_key".to_string(), CryptoAction::SignOnly),
    ("attribute1".to_string(), CryptoAction::EncryptAndSign),
    ("attribute2".to_string(), CryptoAction::SignOnly),
    (":attribute3".to_string(), CryptoAction::DoNothing),
]);
```

DynamoDB AWS 資料庫加密 SDK 中的加密組態

當您使用 AWS 資料庫加密 SDK 時，您必須明確定義 DynamoDB 資料表的加密組態。加密組態中所需的值取決於您是手動或使用註釋的資料類別來定義屬性動作。

下列程式碼片段使用 DynamoDB API 的低階 AWS 資料庫加密 SDK 定義 DynamoDB 資料表加密組態，並允許由不同字首定義的未簽章屬性。

```
let table_config = DynamoDbTableEncryptionConfig::builder()
    .logical_table_name(ddb_table_name)
    .partition_key_name("partition_key")
    .sort_key_name("sort_key")
    .attribute_actions_on_encrypt(attribute_actions_on_encrypt)
    .keyring(kms_keyring)
    .allowed_unsigned_attribute_prefix(UNSIGNED_ATTR_PREFIX)
    // Specifying an algorithm suite is optional
    .algorithm_suite_id(
        DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
```

```
    )
    .build()?;

let table_configs = DynamoDbTablesEncryptionConfig::builder()
    .table_encryption_configs(HashMap::from([(ddb_table_name.to_string(),
 table_config)]))
    .build()?;
```

邏輯資料表名稱

DynamoDB 資料表的邏輯資料表名稱。

邏輯資料表名稱以密碼編譯方式繫結至資料表中存放的所有資料，以簡化 DynamoDB 還原操作。當您第一次定義加密組態時，強烈建議指定您的 DynamoDB 資料表名稱做為邏輯資料表名稱。您必須一律指定相同的邏輯資料表名稱。若要成功解密，邏輯資料表名稱必須符合加密時指定的名稱。如果您的 DynamoDB 資料表名稱在從備份還原 DynamoDB 資料表之後變更，邏輯資料表名稱可確保解密操作仍可辨識資料表。

允許的未簽署屬性

在您的屬性動作DO_NOTHING中標記的屬性。

允許的未簽章屬性會告知用戶端哪些屬性會從簽章中排除。用戶端假設所有其他屬性都包含在簽章中。然後，在解密記錄時，用戶端會決定需要驗證哪些屬性，以及要從您指定的允許未簽章屬性中忽略哪些屬性。您無法從允許的未簽章屬性中移除屬性。

您可以建立列出所有屬性的陣列，明確定義允許的未簽章DO_NOTHING屬性。您也可以在命名DO_NOTHING屬性時指定不同的字首，並使用字首告訴用戶端哪些屬性未簽署。我們強烈建議指定不同的字首，因為它可簡化未來新增DO_NOTHING屬性的程序。如需詳細資訊，請參閱更新您的資料模型。

如果您未指定所有DO_NOTHING屬性的字首，您可以設定allowedUnsignedAttributes陣列，明確列出用戶端在解密時遇到這些屬性時應該取消簽署的所有屬性。只有在絕對必要時，您才應該明確定義允許的未簽署屬性。

搜尋組態（選用）

SearchConfig 定義信標版本。

SearchConfig 必須指定 才能使用可搜尋加密或簽章的信標。

演算法套件（選用）

algorithmSuiteId 定義 AWS 資料庫加密 SDK 使用的演算法套件。

除非您明確指定替代演算法套件，否則 AWS Database Encryption SDK 會使用預設演算法套件。預設演算法套件使用 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，某些信任模型將由沒有數位簽章的演算法套件所滿足。如需 AWS 資料庫加密 SDK 支援的演算法套件相關資訊，請參閱 AWS Database Encryption SDK 中支援的演算法套件。

若要選取不含 ECDSA 數位簽章的 AES-GCM 演算法套件，請在資料表加密組態中包含下列程式碼片段。

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

使用 AWS 資料庫加密 SDK 更新項目

AWS Database Encryption SDK 不支援包含加密或簽章屬性的項目 ddb：UpdateItem。若要更新加密或簽章的屬性，您必須使用 ddb：PutItem。當您在`PutItem`請求中指定與現有項目相同的主索引鍵時，新項目會完全取代現有項目。

舊版 DynamoDB 加密用戶端

2023 年 6 月 9 日，我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。 AWS Database Encryption SDK 繼續支援舊版 DynamoDB Encryption Client 版本。如需使用重新命名變更之用戶端加密程式庫不同部分的詳細資訊，請參閱Amazon DynamoDB Encryption Client 重新命名。

若要遷移至 DynamoDB 的 Java 用戶端加密程式庫的最新版本，請參閱 遷移至 3.x 版。

主題

- AWS DynamoDB 版本的資料庫加密 SDK 支援
- DynamoDB 加密用戶端的運作方式
- Amazon DynamoDB 加密用戶端概念
- 密碼編譯資料提供者
- Amazon DynamoDB Encryption Client 可用的程式設計語言
- 變更您的資料模型
- 故障診斷 DynamoDB Encryption Client 應用程式的問題

AWS DynamoDB 版本的資料庫加密 SDK 支援

舊版章節中的主題提供有關適用於 Java 的 DynamoDB 加密用戶端 1.x-2.x 版和適用於 Python 的 DynamoDB 加密用戶端 1.x-3.x 版的資訊。

下表列出支援 Amazon DynamoDB 中用戶端加密的語言和版本。

程式設計語言	版本	SDK 主要版本生命週期階段
Java	1.x 版	End-of-Support階段，自 2022 年 7 月起生效
Java	2.x 版	一般可用性 (GA)
Java	3.x 版	一般可用性 (GA)
Python	1.x 版	End-of-Support階段，自 2022 年 7 月起生效
Python	2.x 版	End-of-Support階段，自 2022 年 7 月起生效
Python	3.x 版	一般可用性 (GA)

DynamoDB 加密用戶端的運作方式

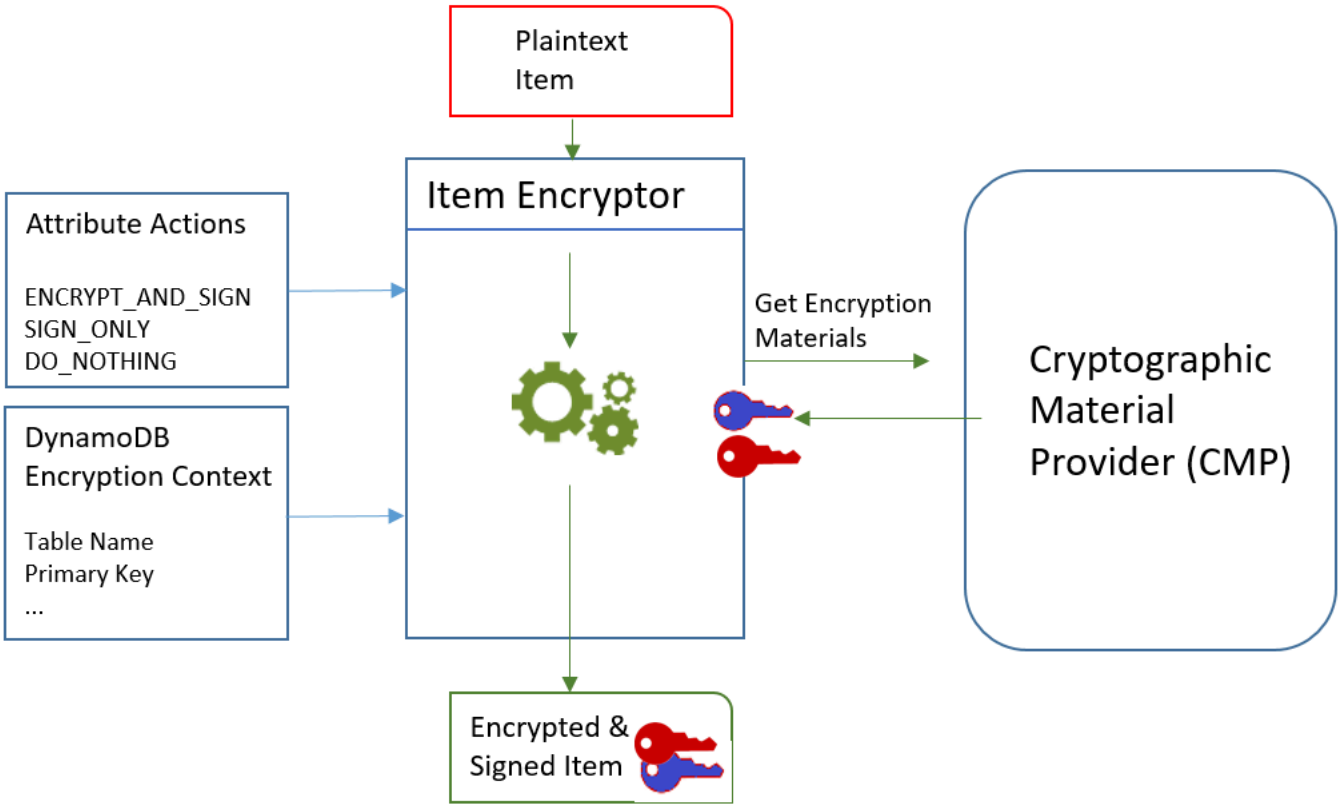
> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

DynamoDB Encryption Client 專為保護您存放在 DynamoDB 中的資料而設計。程式庫包括您可以擴展或以原狀使用的安全實作。再者，大多數元素是由摘要元素表示，所以您可建立和使用相容的自訂元件。

加密並簽署資料表項目

DynamoDB Encryption Client 的核心是一個項目加密程式，可加密、簽署、驗證和解密資料表項目。它會接受資料表項目相關資訊，以及要加密並簽署項目的相關指示。其將從您選取及設定的密碼編譯資料提供者取得加密資料，以及該項資料使用方式的指示。

下圖顯示此程序的高階檢視。



若要加密和簽署資料表項目，DynamoDB Encryption Client 需要：

- 資料表的相關資訊。 它會從您提供的 DynamoDB 加密內容取得資料表的相關資訊。有些協助程式會從 DynamoDB 取得必要資訊，並為您建立 DynamoDB 加密內容。

> **Note**
>
> DynamoDB 加密用戶端中的 DynamoDB 加密內容與 AWS Key Management Service (AWS KMS) 和 中的加密內容無關 AWS Encryption SDK。

- 哪些屬性要加密並簽署。其將從您提供的屬性動作取得這項資訊。
- 加密資料，包括加密金鑰和簽署金鑰。其將從您選取及設定的密碼編譯資料提供者 (CMP) 取得這些資料。

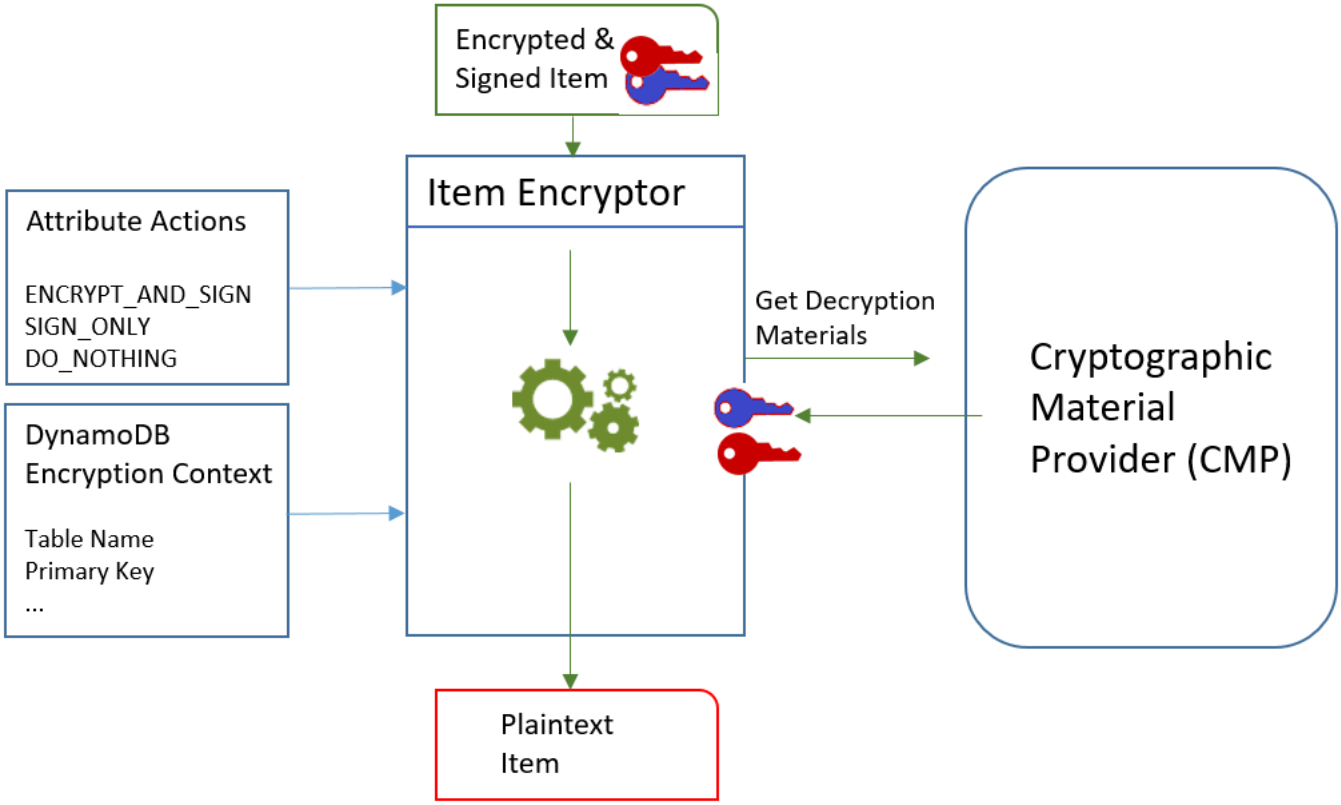
- 加密並簽署項目的指示。CMP 會將可供使用加密資料 (包括加密和簽署演算法) 的指示新增至實際資料描述。

項目加密程式會使用上述所有元素來加密並簽署項目。項目加密程式也會將兩個屬性新增至此項目：包含加密和簽署指示 (實際資料描述) 的資料描述屬性，以及包含簽章的屬性。您可直接與項目加密程式互動，或使用為您與項目加密程式互動的協助程式功能來實作安全的預設行為。

結果是包含已加密並簽署之資料的 DynamoDB 項目。

驗證並解密資料表項目

如下圖所示，這些元件也會一起運作來驗證並解密項目。



若要驗證和解密項目，DynamoDB Encryption Client 需要相同的元件、具有相同組態的元件，或專為解密項目而設計的元件，如下所示：

- DynamoDB 加密內容中資料表的相關資訊。
- 要驗證和解密哪些屬性。其將從屬性動作取得這項資訊。
- 解密資料，包括驗證和加密金鑰，來自您選取和設定的密碼編譯資料提供者 (CMP)。

已加密的項目不包含用來進行加密的 CMP 相關記錄。您必須提供相同的 CMP、具有相同組態的 CMP，或設計用來解密項目的 CMP。

- 如何加密並簽署項目的相關資訊，包括加密和簽署演算法。用戶端會從項目中的資料描述屬性取得這些資訊。

項目加密程式會使用上述所有元素來驗證並解密項目。此外，它不會移除資料描述和簽章屬性。結果是純文字 DynamoDB 項目。

Amazon DynamoDB 加密用戶端概念

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB 加密用戶端 1.x-2.x 版和適用於 Python 的 DynamoDB 加密用戶端 1.x-3.x 版的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

本主題說明 Amazon DynamoDB Encryption Client 中使用的概念和術語。

若要了解 DynamoDB Encryption Client 的元件如何互動，請參閱 DynamoDB 加密用戶端的運作方式。

主題

- 密碼編譯資料提供者 (CMP)
- 項目加密程式
- 屬性動作
- 資料描述
- DynamoDB 加密內容
- 提供者存放區

密碼編譯資料提供者 (CMP)

實作 DynamoDB Encryption Client 時，您的首要任務之一是選取密碼編譯資料提供者 (CMP)（也稱為加密資料提供者)。您的選擇會決定其餘的大部分實作。

密碼編譯資料提供者 (CMP) 會收集、整合及傳回項目加密程式用來加密並簽署資料表項目的密碼編譯資料。CMP 會決定要使用的加密演算法，以及如何產生及保護加密和簽署金鑰。

CMP 會與項目加密程式互動。項目加密程式會請求 CMP 提供加密和解密資料，而 CMP 會將這些資料傳回給項目加密程式。然後，項目加密程式會使用密碼編譯資料將項目加密並簽署，或驗證並解密。

您會在設定用戶端時指定 CMP。您可以建立相容的自訂 CMP，或使用程式庫的其中一個 CMP。大多數 CMP 都適用於多種程式設計語言。

項目加密程式

項目加密器是較低層級的元件，可對 DynamoDB Encryption Client 執行密碼編譯操作。其將請求密碼編譯資料提供者 (CMP) 提供密碼編譯資料，然後使用 CMP 傳回的資料將資料表項目加密並簽署，或驗證並解密。

您可以直接與項目加密程式互動，或使用您的程式庫所提供的協助程式。例如，適用於 Java 的 DynamoDB Encryption Client 包含`AttributeEncryptor`一個協助程式類別，您可以搭配 使用`DynamoDBMapper`，而不是直接與`DynamoDBEncryptor`項目加密程式互動。Python 程式庫包括 `EncryptedTable`、`EncryptedClient` 和 `EncryptedResource` 協助程式類別，這些類別會替您與項目加密程式互動。

屬性動作

屬性動作會告知項目加密程式要對項目的每個屬性執行什麼動作。

屬性動作值可以是下列其中一項：

- 加密和簽署 – 加密屬性值。在項目簽章中包含屬性 (名稱和值)。
- 僅限簽署 – 在項目簽章中包含 屬性。
- 什麼都不做 – 請勿加密或簽署 屬性。

對於可以儲存敏感資料的任何屬性，請使用加密並簽署。針對主索引鍵屬性 (分割區索引鍵和排序索引鍵)，使用僅簽署。資料描述屬性和簽章屬性不會進行簽署或加密。您不需要指定這些屬性的屬性動作。

請仔細選擇屬性動作。如有疑問，請使用加密並簽署。使用 DynamoDB Encryption Client 來保護資料表項目之後，您就無法在沒有簽章驗證錯誤的風險的情況下變更屬性的動作。如需詳細資訊，請參閱變更您的資料模型。

> **Warning**
>
> 請勿加密主索引鍵的屬性。它們必須保持純文字，以便 DynamoDB 可以在不執行完整資料表掃描的情況下找到項目。

如果 DynamoDB 加密內容識別您的主要金鑰屬性，則如果您嘗試加密它們，用戶端會擲回錯誤。

您用來為每種程式設計語言指定屬性動作的技巧都不同。而且，該技巧可以專屬於您所使用的協助程式類別。

如需詳細資訊，請參閱程式設計語言的文件。

- Python
- Java

資料描述

已加密資料表項目的資料描述包含資料表項目加密和簽署方式的相關資訊 (例如加密演算法)。密碼編譯資料提供者 (CMP) 會在整合可供加密和簽署的密碼編譯資料時記錄資料描述。稍後，當它需要整合密碼編譯資料來驗證和解密項目時，它會使用資料描述作為其指南。

在 DynamoDB Encryption Client 中，材料描述是指三個相關元素：

請求的資料描述

有些密碼編譯資料提供者 (CMP) 可讓您指定進階選項，例如加密演算法。若要指出您的選擇，請將名稱/值對新增至請求中 DynamoDB 加密內容的材料描述屬性，以加密資料表項目。此元素又稱為請求的資料描述。請求的資料描述中的有效值由您所選的 CMP 定義。

> **Note**
>
> 由於資料描述可能覆寫安全的預設值，除非您有充分理由要使用請求的資料描述，否則建議您將其省略。

實際資料描述

密碼編譯資料提供者 (CMP) 傳回的資料描述又稱為實際資料描述。它會描述 CMP 在整合密碼編譯資料時所用的實際值。通常包含請求的資料描述及新增和變更 (如果有)。

資料描述屬性

用戶端會在已加密項目的資料描述屬性中儲存實際資料描述。資料描述屬性名稱為 `amzn-ddb-map-desc`，而其值為實際資料描述。用戶端會使用資料描述屬性中的值來驗證並解密項目。

DynamoDB 加密內容

DynamoDB 加密內容會將資料表和項目的相關資訊提供給密碼編譯資料提供者 (CMP)。在進階實作中，DynamoDB 加密內容可以包含請求的資料描述。

當您加密資料表項目時，DynamoDB 加密內容會以加密方式繫結至加密的屬性值。當您解密時，如果 DynamoDB 加密內容與用來加密的 DynamoDB 加密內容不完全、區分大小寫相符，解密操作會失敗。如果您直接與項目加密器互動，您必須在呼叫加密或解密方法時提供 DynamoDB 加密內容。大多數協助程式會為您建立 DynamoDB 加密內容。

> **Note**
>
> DynamoDB Encryption Client 中的 DynamoDB 加密內容與 AWS Key Management Service (AWS KMS) 和 中的加密內容無關 AWS Encryption SDK。

DynamoDB 加密內容可包含下列欄位。所有欄位和值都是選用的。

- 資料表名稱
- 分割區索引鍵名稱
- 排序索引鍵名稱
- 屬性名稱值組
- 請求的資料描述

提供者存放區

提供者存放區 是一個可傳回密碼編譯資料提供者 (CMP) 的元件。提供者存放區可以建立 CMP，或從另一個來源 (例如另一個提供者存放區) 取得。提供者存放區會在持久性儲存體中儲存其建立的 CMP 版本，而存放的每個 CMP 都是依照請求者的資料名稱與版本號碼進行識別。

DynamoDB Encryption Client 中的最近提供者會從提供者存放區取得其 CMPs，但您可以使用提供者存放區將 CMPs 提供給任何元件。每個最近提供者都與一個提供者存放區相關聯，但提供者存放區可以將 CMP 提供給多部主機上的許多請求者。

提供者存放區會隨需建立新的 CMP 版本，並傳回新的和現有版本。它也會傳回特定資料名稱的最新版本號碼。這使請求者能夠得知提供者存放區何時有可請求的新版 CMP。

DynamoDB Encryption Client 包含 MetaStore，這是使用存放在 DynamoDB 中的金鑰建立包裝 CMPs 並使用內部 DynamoDB Encryption Client 加密的提供者商店。

進一步了解：

- 提供者存放區：Java、Python
- 中繼存放區：Java、Python

密碼編譯資料提供者

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB 加密用戶端 1.x-2.x 版和適用於 Python 的 DynamoDB 加密用戶端 1.x-3.x 版的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

使用 DynamoDB Encryption Client 時，您所做的最重要決策之一是選擇密碼編譯資料提供者 (CMP)。CMP 會整合密碼編譯資料，並將其傳回至項目加密程式。它也會決定加密和簽署金鑰的產生方式、要為每個項目產生新的金鑰資料還是重複使用這些資料，以及所使用的加密和簽署演算法等。

您可以從 DynamoDB Encryption Client 程式庫中提供的實作中選擇 CMP，或建置相容的自訂 CMP。您的 CMP 選擇也可能取決於您所使用的程式設計語言。

本主題說明最常見的 CMP，並提供相關建議以協助您選擇最適用於應用程式的 CMP。

直接 KMS 資料提供者

Direct KMS 材料提供者會在 下保護您的資料表項目AWS KMS key，而這些項目永遠不會讓 AWS Key Management Service(AWS KMS) 處於未加密狀態。您的應用程式不會產生或管理任何密碼編譯資料。由於它使用 AWS KMS key 為每個項目產生唯一的加密和簽署金鑰， AWS KMS 因此每次加密或解密項目時，此提供者都會呼叫 。

如果您使用 AWS KMS ，且每筆交易只有一個 AWS KMS 呼叫對您的應用程式來說很實用，則此供應商是不錯的選擇。

如需詳細資訊，請參閱 直接 KMS 資料提供者。

包裝資料提供者 (包裝 CMP)

包裝材料提供者 （ 包裝 CMP) 可讓您在 DynamoDB 加密用戶端外部產生和管理包裝和簽署金鑰。

包裝 CMP 會為每個項目產生唯一的加密金鑰。接著，它會使用您所提供的包裝 (或取消包裝) 和簽署金鑰。如此，您可決定包裝和簽署金鑰的產生方式，以及要讓每個項目各有唯一的金鑰還是重複使用這些金鑰。包裝 CMP 是 Direct KMS 提供者的安全替代方案，適用於不使用 AWS KMS 且可以安全管理密碼編譯資料的應用程式。

如需詳細資訊，請參閱 包裝資料提供者。

最近提供者

最近提供者是一個密碼編譯資料提供者 (CMP)，旨在與提供者存放區搭配使用。它會從提供者存放區取得 CMP，並取得它從 CMP 傳回的密碼編譯資料。最近提供者通常會使用各個 CMP 因應多次密碼編譯資料請求，但您也可以使用提供者存放區的功能來控制資料重複使用的程度、決定輪換 CMP 的頻率，甚至在不變更最近提供者的情況下變更所使用的 CMP 類型。

您可以將最近提供者與任何相容的提供者存放區搭配使用。DynamoDB Encryption Client 包含 MetaStore，這是傳回包裝 CMPs的提供者存放區。

對於需要盡可能避免呼叫其密碼編譯來源的應用程式，以及可重複使用部分密碼編譯資料而不會違反安全性需求的應用程式，最近提供者將是理想的選擇。例如，它可讓您在 AWS KMS key AWS Key Management Service(AWS KMS) 的 下保護您的密碼編譯資料，而不必 AWS KMS 在每次加密或解密項目時呼叫 。

如需詳細資訊，請參閱 最近提供者。

靜態資料提供者

靜態資料提供者是針對測試、概念驗證示範和舊版相容性而設計的。它不會為每個項目產生任何唯一的密碼編譯資料。它會傳回您所提供的相同加密和簽署金鑰，並直接使用這些金鑰來加密、解密和簽署您的資料表項目。

> **Note**
>
> Java 程式庫中的非對稱靜態提供者不是靜態提供者。其僅提供包裝 CMP 的替代建構函數。此提供者可在生產環境中安全地使用，但只要情況允許，您即應直接使用包裝 CMP。

主題

- 直接 KMS 資料提供者
- 包裝資料提供者
- 最近提供者
- 靜態資料提供者

直接 KMS 資料提供者

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB 加密用戶端 1.x-2.x 版和適用於 Python 的 DynamoDB 加密用戶端 1.x-3.x 版的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

Direct KMS 材料提供者 (Direct KMS 提供者） 可在 下保護您的資料表項目AWS KMS key，而這些項目永遠不會讓 AWS Key Management Service(AWS KMS) 處於未加密狀態。此密碼編譯資料提供者會為每個資料表項目傳回唯一的加密金鑰和簽署金鑰。若要這樣做， AWS KMS 它會在每次加密或解密項目時呼叫 。

如果您以高頻率和大規模處理 DynamoDB 項目，可能會超過 AWS KMS requests-per-second數限制，導致處理延遲。如果您需要超過限制，請在 AWS 支援 中心建立案例。您也可以考慮使用金鑰重複使用有限的密碼編譯材料提供者，例如最近提供者。

若要使用 Direct KMS 提供者，發起人必須擁有 AWS 帳戶、至少一個 AWS KMS key和 許可，才能在 上呼叫 GenerateDataKey 和 Decrypt 操作 AWS KMS key。 AWS KMS key 必須是對稱加密金鑰；DynamoDB Encryption Client 不支援非對稱加密。如果您使用的是 DynamoDB 全域資料表，您可能想要指定AWS KMS 多區域金鑰。如需詳細資訊，請參閱 使用方式。

> **Note**
>
> 當您使用 Direct KMS 提供者時，您的主要金鑰屬性的名稱和值會以純文字顯示在相關 AWS KMS 操作的AWS KMS 加密內容和 AWS CloudTrail 日誌中。不過，DynamoDB Encryption Client 永遠不會公開任何加密屬性值的純文字。

Direct KMS 提供者是 DynamoDB Encryption Client 支援的幾個密碼編譯材料提供者 (CMPs) 之一。如需其他 CMP 的相關資訊，請參閱密碼編譯資料提供者。

如需範例程式碼，請參閱：

- Java：AwsKmsEncryptedItem
- Python：aws-kms-encrypted-table，aws-kms-encrypted-item

主題

- 使用方式
- 運作方式

使用方式

若要建立直接 KMS 提供者，請使用金鑰 ID 參數在您的帳戶中指定對稱加密 KMS 金鑰。金鑰 ID 參數的值可以是 的金鑰 ID、金鑰 ARN、別名名稱或別名 ARN AWS KMS key。如需金鑰識別符的詳細資訊，請參閱《 AWS Key Management Service 開發人員指南》中的金鑰識別符。

Direct KMS 提供者需要對稱加密 KMS 金鑰。無法使用非對稱 KMS 金鑰。不過，您可以在自訂金鑰存放區中使用多區域 KMS 金鑰、具有匯入金鑰材料的 KMS 金鑰，或 KMS 金鑰。您必須擁有 KMS 金鑰上的 kms:GenerateDataKey 和 kms:Decrypt 許可。因此，您必須使用客戶受管金鑰，而非 AWS 受管或 AWS 擁有的 KMS 金鑰。

DynamoDB Encryption Client for Python 會在金鑰 ID 參數值中，決定要 AWS KMS 從 區域呼叫的區域，如果包含一個。否則，如果您在 AWS KMS 用戶端中指定一個 或您在 中設定的 區域，它會使用區域 適用於 Python (Boto3) 的 AWS SDK。如需 Python 中區域選擇的資訊，請參閱適用於 Python (Boto3) 的 AWS SDK API 參考中的組態。

如果您指定的用戶端包含 區域，則適用於 Java 的 DynamoDB 加密用戶端會 AWS KMS 決定 AWS KMS 從用戶端中的 區域呼叫的區域。否則，它會使用您在 中設定的 區域 適用於 Java 的 AWS SDK。如需 中區域選擇的相關資訊 適用於 Java 的 AWS SDK，請參閱《 適用於 Java 的 AWS SDK 開發人員指南》中的AWS 區域 選擇。

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
```

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

下列範例使用金鑰 ARN 來指定 AWS KMS key。如果您的金鑰識別符不包含 AWS 區域，則 DynamoDB Encryption Client 會從設定的 Botocore 工作階段、如果有，或從 Boto 預設值取得區域。

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

如果您使用的是 Amazon DynamoDB 全域資料表，建議您使用 AWS KMS 多區域金鑰加密資料。多區域金鑰 AWS KMS keys 不同 AWS 區域 ，可以互換使用，因為它們具有相同的金鑰 ID 和金鑰材料。如需詳細資訊，請參閱《 AWS Key Management Service 開發人員指南》中的使用多區域金鑰。

> **Note**
>
> 如果您使用的是全域資料表 2017.11.29 版，則必須設定屬性動作，以便不會加密或簽署預留複寫欄位。如需詳細資訊，請參閱 舊版全域資料表的問題。

若要搭配 DynamoDB 加密用戶端使用多區域金鑰，請建立多區域金鑰並將其複寫至應用程式執行所在的區域。然後設定直接 KMS 提供者，在 DynamoDB Encryption Client 呼叫的區域中使用多區域金鑰 AWS KMS。

下列範例會設定 DynamoDB Encryption Client 來加密美國東部（維吉尼亞北部）(us-east-1) 區域中的資料，並使用多區域金鑰在美國西部（奧勒岡）(us-west-2) 區域中解密資料。

Java

在此範例中，DynamoDB Encryption Client AWS KMS 會從 AWS KMS 用戶端中的 區域取得要呼叫的 區域。此`keyArn`值可識別相同區域中的多區域金鑰。

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
```

```
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2

// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

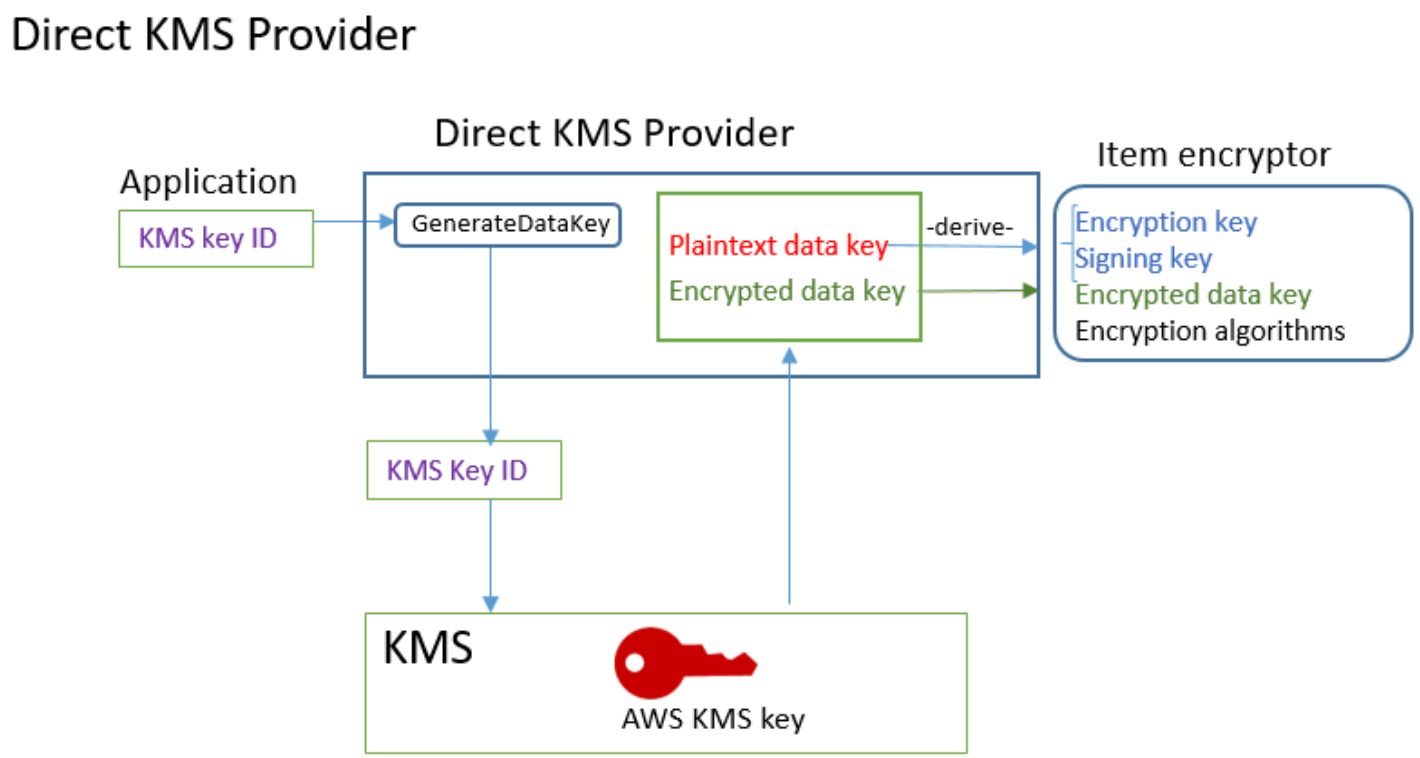
在此範例中，DynamoDB 加密用戶端 AWS KMS 會從金鑰 ARN 中的區域取得要呼叫的區域。

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

運作方式

Direct KMS 提供者會傳回受您指定 保護 AWS KMS key 的加密和簽署金鑰，如下圖所示。



- 若要產生加密資料，Direct KMS 提供者 AWS KMS 會要求 使用您指定的 AWS KMS key 為每個項目產生唯一的資料金鑰。它會從資料金鑰的純文字複本衍生項目的加密和簽署金鑰，然後傳回加密和簽署金鑰，以及在項目資料描述屬性中存放的加密資料金鑰。

  項目加密程式會在使用加密和簽署金鑰後，盡快將其從記憶體中移除。加密的項目中只會儲存用來衍生這些金鑰的資料金鑰加密複本。
- 若要產生解密資料，直接 KMS 提供者 AWS KMS 會要求解密加密的資料金鑰。隨後，它會從純文字資料金鑰衍生驗證和簽署金鑰，並將它們傳回至項目加密程式。

  項目加密程式會驗證該項目，如果驗證成功，則會將加密的值解密。接著，它會盡快從記憶體中移除這些金鑰。

取得加密資料

本節將詳細說明直接 KMS 提供者在接收到來自項目加密程式的加密資料請求時的輸入、輸出和處理情形。

輸入 (從應用程式)

- 的金鑰 ID AWS KMS key。

輸入 (從項目加密程式)

- DynamoDB 加密內容

輸出 (到項目加密程式)

- 加密金鑰 (純文字)
- 簽署金鑰
- 在實際資料描述中：這些值會儲存在用戶端新增至項目的資料描述屬性中。
  - amzn-ddb-env-key：由 加密的 Base64-encoded資料金鑰 AWS KMS key
  - amzn-ddb-env-alg：加密演算法，預設為 AES/256
  - amzn-ddb-sig-alg：簽署演算法，預設為 HmacSHA256/256
  - amzn-ddb-wrap-alg：kms

處理

1. Direct KMS 提供者傳送 AWS KMS 請求，以使用指定的 AWS KMS key 來產生項目的唯一資料金鑰。此操作會傳回以 AWS KMS key加密的純文字金鑰和複本。這項資料又稱為初始金鑰資料。

   此要求包含 AWS KMS 加密內容中的下列純文字值。這些非機密值會以密碼編譯的方式繫結至加密的物件，而在解密時需要相同的加密細節。您可以使用這些值來識別 AWS CloudTrail 日誌 AWS KMS 中的 呼叫。

   - amzn-ddb-env-alg – 預設 AES/256 加密演算法
   - amzn-ddb-sig-alg – 簽署演算法，預設 HmacSHA256/256
   - （選用） aws-kms-table – *#####*
   - （選用） *########* – *#######* （二進位值為 Base64-encoded)
   - （選用） *#######* – *######* （二進位值以 Base64-encoded)

   Direct KMS 提供者會從項目的 DynamoDB AWS KMS 加密內容取得加密內容的值。 DynamoDB 如果 DynamoDB 加密內容不包含值，例如資料表名稱，則會從 AWS KMS 加密內容中省略該名稱/值對。

2. 直接 KMS 提供者會從資料金鑰衍生對稱加密金鑰和簽署金鑰。根據預設，其將使用安全雜湊演算法 (SHA) 256 和 RFC5869 HMAC 式金鑰衍生函數，衍生 256 位元 AES 對稱加密金鑰和 256 位元 HMAC-SHA-256 簽署金鑰。
3. 直接 KMS 提供者會將輸出傳回至項目加密程式。

4. 項目加密程式會採用實際資料描述中指定的演算法，使用加密金鑰為指定的屬性加密，並使用簽署金鑰加以簽署。它會盡快從記憶體中移除這些純文字金鑰。

取得解密資料

本節將詳細說明直接 KMS 提供者在接收到來自項目加密程式的解密資料請求時的輸入、輸出和處理情形。

輸入 (從應用程式)

- 的金鑰 ID AWS KMS key。

  金鑰 ID 的值可以是 的金鑰 ID、金鑰 ARN、別名名稱或別名 ARN AWS KMS key。未包含在金鑰 ID 中的任何值，例如區域，都必須在AWS 具名設定檔中可用。金鑰 ARN 提供所有需要的值 AWS KMS 。

輸入 (從項目加密程式)

- DynamoDB 加密內容的複本，其中包含材料描述屬性的內容。

輸出 (到項目加密程式)

- 加密金鑰 (純文字)
- 簽署金鑰

處理

1. Direct KMS 提供者會從加密項目中的資料描述屬性取得加密的資料金鑰。
2. 它會要求 AWS KMS 使用指定的 AWS KMS key 來解密加密的資料金鑰。此操作會傳回純文字金鑰。

   此要求必須使用先前用來產生和加密資料金鑰的相同 AWS KMS 加密內容。

   - aws-kms-table – *#####*
   - *########* – *#######* （二進位值以 Base64-encoded)
   - （選用） *#######* – *######* （二進位值為 Base64-encoded)
   - amzn-ddb-env-alg – 預設 AES/256 加密演算法
   - amzn-ddb-sig-alg – 簽署演算法，預設 HmacSHA256/256

3. 直接 KMS 提供者會使用安全雜湊演算法 (SHA) 256 和 RFC5869 HMAC 式金鑰衍生函數，從資料金鑰衍生 256 位元 AES 對稱加密金鑰和 256 位元 HMAC-SHA-256 簽署金鑰。
4. 直接 KMS 提供者會將輸出傳回至項目加密程式。
5. 項目加密程式會使用簽署金鑰來驗證項目。如果成功，則會使用對稱加密金鑰將加密的屬性值解密。這些操作會使用實際資料描述中指定的加密和簽署演算法。項目加密程式會盡快從記憶體中移除這些純文字金鑰。

包裝資料提供者

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

包裝材料提供者（包裝 CMP) 可讓您使用來自任何來源的包裝和簽署金鑰搭配 DynamoDB 加密用戶端。包裝 CMP 不依賴於任何 AWS 服務。不過，您必須在用戶端以外產生及管理包裝和簽署金鑰，包括提供正確的金鑰來驗證和解密項目。

包裝 CMP 會為每個項目產生唯一的項目加密金鑰。其將使用您所提供的包裝金鑰來包裝項目加密金鑰，並將包裝的項目加密金鑰儲存至項目的資料描述屬性。因為您提供包裝和簽署金鑰，所以由您決定包裝和簽署金鑰的產生方式，以及要讓每個項目各有唯一的金鑰還是重複使用。

對於可以管理密碼編譯資料的應用程式而言，包裝 CMP 是安全的實作與理想的選擇。

包裝 CMP 是 DynamoDB Encryption Client 支援的幾個密碼編譯材料提供者 (CMPs) 之一。如需其他 CMP 的相關資訊，請參閱密碼編譯資料提供者。

如需範例程式碼，請參閱：

- Java：AsymmetricEncryptedItem
- Python：wrapped-rsa-encrypted-table，wrapped-symmetric-encrypted-table

主題

- 使用方式

- 運作方式

使用方式

若要建立包裝 CMP，請指定包裝金鑰 (加密時需要)、取消包裝金鑰 (解密時需要) 以及簽署金鑰。您必須在加密和解密項目時提供金鑰。

包裝、取消包裝和簽署金鑰可以是對稱金鑰或非對稱金鑰對。

Java

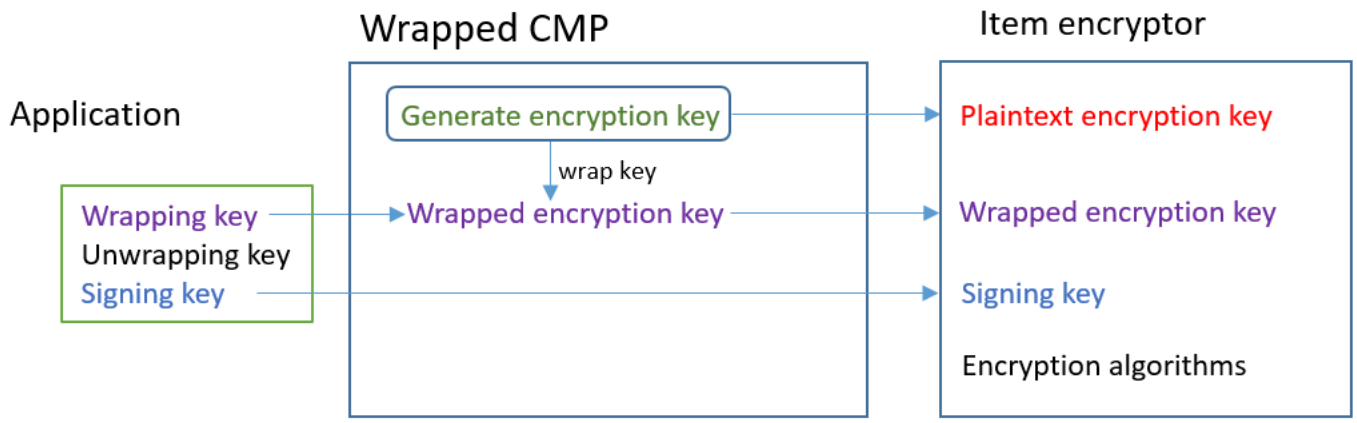
```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

運作方式

包裝 CMP 會為每個項目產生新的項目加密金鑰。如下圖所示，它會使用您所提供的包裝、取消包裝和簽署金鑰。



取得加密資料

本節將詳細說明包裝資料提供者 (包裝 CMP) 在接收到加密資料請求時的輸入、輸出和處理情形。

輸入 (從應用程式)

- 包裝金鑰：進階加密標準 (AES) 對稱金鑰，或 RSA 公有金鑰。如有任何已加密的屬性值，則為必要。否則為選用並予以忽略。
- 取消包裝金鑰：選用並予以忽略。
- 簽署金鑰

輸入 (從項目加密程式)

- DynamoDB 加密內容

輸出 (到項目加密程式)：

- 純文字項目加密金鑰
- 簽署金鑰 (不變)
- 實際資料描述：這些值會儲存在用戶端新增至項目的資料描述屬性中。
  - `amzn-ddb-env-key`：Base64 編碼的包裝項目加密金鑰
  - `amzn-ddb-env-alg`：用來加密項目的加密演算法。預設為 AES-256-CBC。
  - `amzn-ddb-wrap-alg`：包裝 CMP 用來包裝項目加密金鑰的包裝演算法。如果包裝金鑰是 AES 金鑰，則會使用未填補的 `AES-Keywrap` (如 RFC 3394 定義) 來包裝此金鑰。如果包裝金鑰是 RSA 金鑰，則會使用 RSA OAEP (MGF1 填補) 來加密此金鑰。

處理

當您加密項目時，您會傳入包裝金鑰和簽署金鑰。取消包裝金鑰為選用並予以忽略。

1. 包裝 CMP 會為資料表項目產生唯一的對稱項目加密金鑰。
2. 它會使用您指定的包裝金鑰來包裝項目加密金鑰。接著，它會盡快從記憶體中移除此金鑰。
3. 其將傳回純文字項目加密金鑰、您所提供的簽署金鑰，以及包含包裝項目加密金鑰和加密與包裝演算法的實際資料描述。
4. 項目加密程式會使用純文字加密金鑰來加密項目。它會使用您所提供的簽署金鑰來簽署金鑰。接著，它會盡快從記憶體中移除這些純文字金鑰。它會將實際資料描述中的欄位 (包括包裝加密金鑰 (`amzn-ddb-env-key`)) 複製到項目的資料描述屬性。

取得解密資料

本節將詳細說明包裝資料提供者 (包裝 CMP) 在接收到解密資料請求時的輸入、輸出和處理情形。

輸入 (從應用程式)

- 包裝金鑰：選用並予以忽略。
- 取消包裝金鑰：相同的進階加密標準 (AES) 對稱金鑰，或與加密使用的 RSA 公有金鑰對應的 RSA 私有金鑰。如有任何已加密的屬性值，則為必要。否則為選用並予以忽略。
- 簽署金鑰

輸入 (從項目加密程式)

- DynamoDB 加密內容的複本，其中包含材料描述屬性的內容。

輸出 (到項目加密程式)

- 純文字項目加密金鑰
- 簽署金鑰 (不變)

處理

當您解密項目時，您會傳入取消包裝金鑰和簽署金鑰。包裝金鑰為選用並予以忽略。

1. 包裝 CMP 會從項目的資料描述屬性取得包裝項目加密金鑰。

2. 它會使用取消包裝金鑰和演算法來取消包裝項目加密金鑰。
3. 它會將純文字項目加密金鑰、簽署金鑰以及加密和簽署演算法傳回給項目加密程式。
4. 項目加密程式會使用簽署金鑰來驗證項目。如果成功，則會使用項目加密金鑰來將項目解密。接著，它會盡快從記憶體中移除這些純文字金鑰。

最近提供者

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

最近提供者是一個密碼編譯資料提供者 (CMP)，旨在與提供者存放區搭配使用。它會從提供者存放區取得 CMP，並取得它從 CMP 傳回的密碼編譯資料。其通常會使用各個 CMP 因應多次密碼編譯資料請求。但您也可以使用提供者存放區的功能來控制資料的重複使用程度、決定 CMP 的輪換頻率，甚至在不變更「最近提供者」的情況下變更所使用的 CMP 類型。

> **Note**
>
> 與最近提供者的 `MostRecentProvider`符號相關聯的程式碼，可能會在程序的生命週期內將密碼編譯資料存放在記憶體中。它可能會允許呼叫者使用他們不再獲授權使用的金鑰。`MostRecentProvider` 該符號在 DynamoDB Encryption Client 的較舊支援版本中被取代，並從 2.0.0 版中移除。它被 `CachingMostRecentProvider`符號取代。如需詳細資訊，請參閱 最新提供者的更新。

對於需要盡可能避免呼叫提供者存放區與其密碼編譯來源的應用程式，以及可重複使用部分密碼編譯資料而不會違反安全性需求的應用程式，最近提供者將是理想的選擇。例如，它可讓您在 AWS KMS key AWS Key Management Service(AWS KMS) 的 下保護您的密碼編譯資料，而不必 AWS KMS 在每次加密或解密項目時呼叫 。

您選擇的提供者存放區可決定最近提供者使用的 CMP 類型，以及其取得新 CMP 的頻率。您可以使用任何相容提供者存放區搭配最近提供者，包括您所設計的自訂提供者存放區。

DynamoDB 加密用戶端包含 MetaStore，可建立和傳回包裝材料提供者（包裝 CMPs)。MetaStore 會儲存其在內部 DynamoDB 資料表中產生的多個包裝 CMPs 版本，並透過 DynamoDB 加密用戶端的內部執行個體使用用戶端加密來保護它們。

您可以設定 MetaStore 使用任何類型的內部 CMP 來保護資料表中的資料，包括產生受您保護之密碼編譯資料的直接 KMS 提供者 AWS KMS key、使用您提供的包裝和簽署金鑰的包裝 CMP，或您設計的相容自訂 CMP。

如需範例程式碼，請參閱：

- Java：MostRecentEncryptedItem
- Python：most_recent_provider_encrypted_table

主題

- 使用方式
- 運作方式
- 最新提供者的更新

使用方式

若要建立最近提供者，您必須建立及設定提供者存放區，然後建立可使用該提供者存放區的最近提供者。

下列範例示範如何建立使用 MetaStore 的最新提供者，並使用來自 Direct KMS 提供者的密碼編譯資料來保護其內部 DynamoDB 資料表中的版本。這些範例使用 `CachingMostRecentProvider`符號。

每個最近提供者都有一個名稱，可識別其在 MetaStore 資料表中的 CMPs、time-to-live(TTL) 設定，以及決定快取可保留多少項目的快取大小設定。這些範例將快取大小設定為 1000 個項目，TTL 為 60 秒。

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'

// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
```

```
// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms,
 keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)
```

```
# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
    version_ttl=60.0,
    cache_size=1000
)
```

運作方式

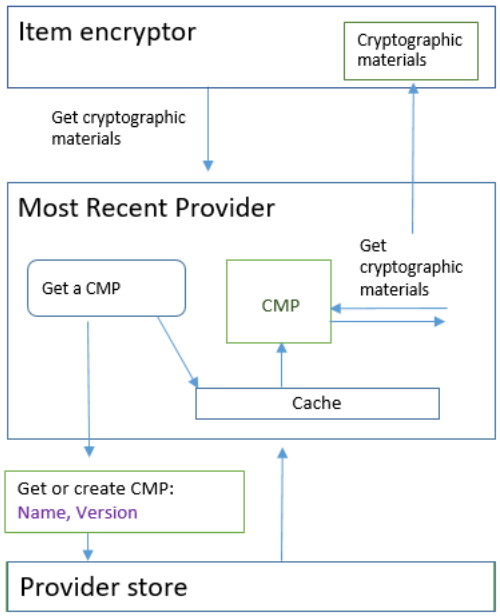
最近提供者會從提供者存放區取得 CMP。然後，它會使用 CMP 來產生密碼編譯資料並將其傳回給項目加密程式。

關於最近提供者

最近的提供者會從提供者存放區取得密碼編譯資料提供者 (CMP)。然後，它會使用 CMP 來產生可傳回的密碼編譯資料。每個最近提供者都會與一個提供者存放區相關聯，但提供者存放區可以將 CMP 提供給多部主機上的多個提供者。

最近提供者可與任何提供者存放區中的任何相容 CMP 搭配使用。它會向 CMP 請求加密或解密資料，並將輸出傳回給項目加密器。並不會執行任何密碼編譯操作。

若要向提供者存取區請求 CMP，最近提供者可提供其資料名稱以及想要使用的現有 CMP 版本。針對加密資料，最近提供者一律會請求最大 (「最近」) 版本。針對解密資料，其將請求用來建立加密資料的 CMP 版本 (如下圖所示)。



最近提供者會將提供者存放區傳回的 CMP 版本儲存於記憶體中的本機最久未使用 (LRU) 快取。此快取可讓最近提供者取得它所需的 CMP，而不需針對每個項目呼叫提供者存放區。您可以隨需清除此快取。

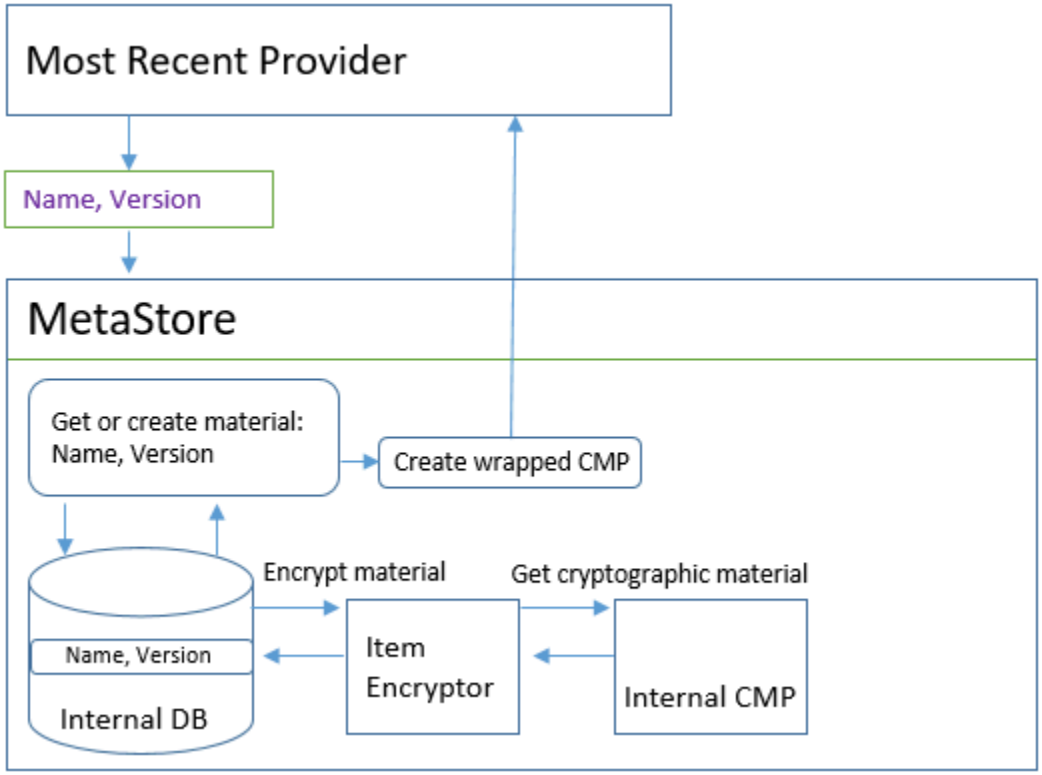
最近的提供者使用可設定的time-to-live值，您可以根據應用程式的特性來調整。

關於中繼存放區

您可以使用最近提供者搭配任何提供者存放區，包括相容的自訂提供者存放區。DynamoDB Encryption Client 包含 MetaStore，這是您可以設定和自訂的安全實作。

中繼存放區是一個提供者存放區，可建立及傳回包裝 CMP 所需的包裝金鑰、取消包裝金鑰和簽署金鑰設定的包裝 CMP。MetaStore 是最近提供者的安全選項，因為包裝 CMPs 一律會為每個項目產生唯一的項目加密金鑰。只有可保護項目加密金鑰的包裝金鑰以及簽署金鑰能重複使用。

下圖說明中繼存放區元件以及其與最近提供者互動的方式。



MetaStore 會產生包裝 CMPs，然後將它們（以加密形式）存放在內部 DynamoDB 資料表中。分割區索引鍵是最新提供者資料的名稱；排序索引鍵是版本編號。資料表中的資料受到內部 DynamoDB 加密用戶端的保護，包括項目加密器和內部密碼編譯資料提供者 (CMP)。

您可在中繼存放區使用任何類型的內部 CMP，包括直接 KMS 提供者、具有您所提供密碼編譯資料的包裝 CMP，或相容的自訂 CMP。如果 MetaStore 中的內部 CMP 是直接 KMS 提供者，則可重複使用的包裝和簽署金鑰會受到 AWS Key Management Service() AWS KMS key 中的 保護AWS KMS。每次 MetaStore 將新的 CMP 版本新增至其內部資料表，或從其內部資料表取得 CMP 版本 AWS KMS 時，都會呼叫 MetaStore。

設定time-to-live值

您可以為您建立的每個最近提供者設定time-to-live(TTL) 值。一般而言，請使用應用程式適用的最低 TTL 值。

最近提供者的 `CachingMostRecentProvider`符號中會變更 TTL 值的使用。

> **Note**
>
> 最近提供者的 `MostRecentProvider`符號已在 DynamoDB Encryption Client 的較舊支援版本中取代，並從 2.0.0 版中移除。它被 `CachingMostRecentProvider`符號取代。建議您盡快更新程式碼。如需詳細資訊，請參閱 最新提供者的更新。

`CachingMostRecentProvider`

會以兩種不同的方式`CachingMostRecentProvider`使用 TTL 值。

- TTL 會決定最近提供者檢查提供者存放區是否有新版本的 CMP 的頻率。如果新版本可用，最近提供者會取代其 CMP 並重新整理其密碼編譯資料。否則，它會繼續使用其目前的 CMP 和密碼編譯資料。
- TTL 會決定快取中 CMPs的使用時間。在使用快取的 CMP 進行加密之前，最近提供者會評估其在快取中的時間。如果 CMP 快取時間超過 TTL，則會從快取中移出 CMP，而最近提供者會從其提供者存放區取得新的最新版本 CMP。

`MostRecentProvider`

在 中`MostRecentProvider`，TTL 會決定最近提供者檢查提供者存放區是否有新版本的 CMP 的頻率。如果新版本可用，最近提供者會取代其 CMP 並重新整理其密碼編譯資料。否則，它會繼續使用其目前的 CMP 和密碼編譯資料。

TTL 不會判斷新 CMP 版本的建立頻率。您可以透過輪換密碼編譯材料來建立新的 CMP 版本。

理想的 TTL 值會隨應用程式及其延遲和可用性目標而有所不同。較低的 TTL 可縮短密碼編譯資料存放在記憶體中的時間，藉此改善您的安全設定檔。此外，較低的 TTL 會更頻繁地重新整理重要資訊。例如，如果您的內部 CMP 是直接 KMS 提供者，它會更頻繁地驗證發起人是否仍獲得使用 的授權 AWS KMS key。

不過，如果 TTL 過短，對提供者商店的頻繁呼叫可能會增加您的成本，並導致提供者商店限流來自您應用程式和其他共用您服務帳戶的應用程式的請求。您可能也受益於將 TTL 與輪換密碼編譯材料的速率進行協調。

在測試期間， 會在不同的工作負載下變更 TTL 和快取大小，直到您找到適用於您應用程式以及安全和效能標準的組態為止。

輪換密碼編譯資料

當最近提供者需要加密資料時，一律會使用其已知的最新版本 CMP。檢查是否有較新版本的頻率，取決於您在設定最近提供者時所設定的time-to-live(TTL) 值。

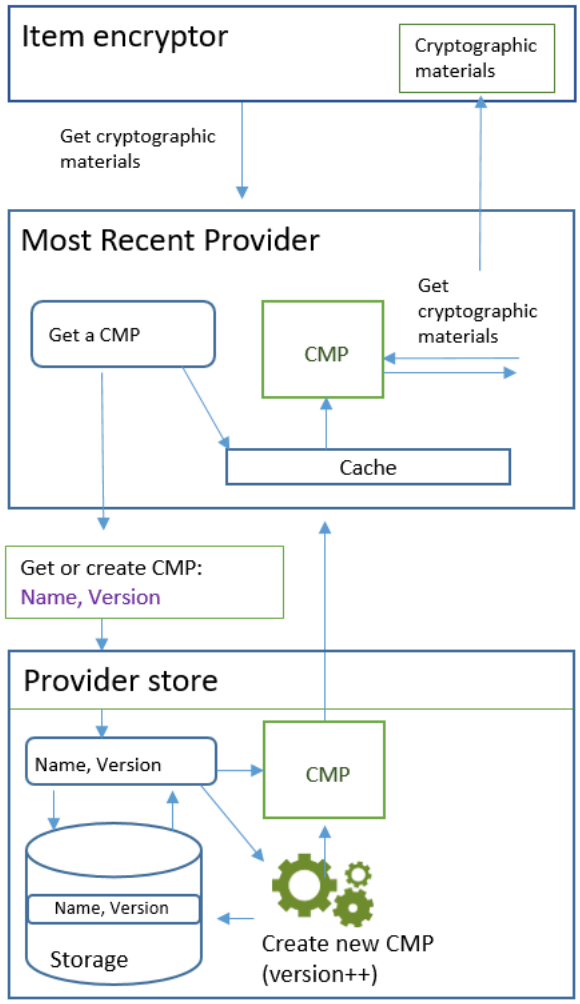
當 TTL 過期時，最近的提供者會檢查提供者存放區是否有較新的 CMP 版本。如果可用，則最新提供者會取得它，並取代其快取中的 CMP。它會使用此 CMP 及其密碼編譯材料，直到發現提供者存放區有較新的版本為止。

若要告知提供者存放區為最近提供者建立新的 CMP 版本，請以最近提供者的資料名稱呼叫提供者存放區的「建立新提供者」操作。提供者存放區會建立新的 CMP，並且在其內部儲存體中使用較大版本號碼儲存已加密的複本。(它也會傳回 CMP，但您可予以捨棄。) 因此，當最近提供者下次查詢提供者存放區以取得其 CMPs的最大版本編號時，就會取得新的更高版本編號，並在後續向存放區提出的請求中使用它來查看是否已建立新的 CMP 版本。

您可以根據時間、處理的項目或屬性數目，或是對您應用程式有意義的任何其他度量，為您的「建立新提供者」呼叫進行排程。

取得加密資料

最近提供者會使用下列程序 (如下圖所示)，取得它傳回給項目加密程式的加密資料。輸出取決於提供者存放區所傳回的 CMP 類型。最近的提供者可以使用任何相容的提供者存放區，包括 DynamoDB 加密用戶端中包含的 MetaStore。



當您使用 `CachingMostRecentProvider`符號建立最近提供者時，您可以指定提供者存放區、最近提供者的名稱，以及time-to-live(TTL) 值。您也可以選擇性地指定快取大小，以決定快取中可存在的加密資料數目上限。

當項目加密程式向最近提供者詢問加密資料時，最近提供者會開始搜尋其快取中的 CMP 最新版本。

- 如果它在其快取中找到最新版本 CMP，且 CMP 未超過 TTL 值，則最新提供者會使用 CMP 產生加密資料。接著，它會將加密資料傳回給項目加密程式。此作業不需要呼叫提供者存放區。
- 如果 CMP 的最新版本不在快取中，或位於快取中但已超過其 TTL 值，則最近提供者會向其提供者存放區請求 CMP。此請求包括最近提供者資料名稱及其所知的最大版本號碼。
  1. 提供者存放區會從其持久性儲存體傳回 CMP。如果提供者存放區是 MetaStore，則會使用最近提供者材料名稱做為分割區索引鍵，並將版本編號做為排序索引鍵，從其內部 DynamoDB 資料表取得加密的包裝 CMP。中繼存放區會使用其內部項目加密程式和內部 CMP，將包裝 CMP 解密。接著，它會將純文字 CMP 傳回給最近提供者。如果內部 CMP 是直接 KMS 提供者，這個步驟就包括對 AWS Key Management Service (AWS KMS) 的呼叫。

2. CMP 會將 `amzn-ddb-meta-id` 欄位新增至實際資料描述。其值為資料名稱與其內部資料表中的 CMP 版本。提供者存放區會將 CMP 傳回給最近提供者。
3. 最近提供者會快取記憶體中的 CMP。
4. 最近提供者會使用 CMP 來產生加密資料。接著，它會將加密資料傳回給項目加密程式。

取得解密資料

當項目加密程式向最近提供者詢問解密資料時，最近提供者會使用下列程序來取得和傳回解密資料。

1. 最近提供者會向提供者存放區詢問用來加密項目的密碼編譯資料版本號碼。其將從項目的資料描述屬性傳入實際資料描述。
2. 提供者存放區會從實際資料描述中的 `amzn-ddb-meta-id` 欄位取得加密 CMP 版本號碼，並將它傳回給最近提供者。
3. 最近提供者會在其快取中搜尋用來加密和簽署項目的 CMP 版本。

- 如果發現 CMP 的相符版本在其快取中，且 CMP 未超過time-to-live(TTL) 值，則最新提供者會使用 CMP 產生解密資料。接著，它會將解密資料傳回給項目加密程式。此操作不需要呼叫提供者存放區或任何其他 CMP。
- 如果 CMP 的相符版本不在快取中，或快取 AWS KMS key 的 超過其 TTL 值，則最近提供者會向其提供者存放區請求 CMP。其會在請求中傳送資料名稱與加密 CMP 版本號碼。
  1. 提供者存放區會使用最近提供者名稱作為分割區索引鍵並使用版本號碼作為排序索引鍵，在其持久性儲存體中搜尋 CMP。
     - 如果其持久性儲存體中沒有此名稱和版本號碼，則提供者存放區會擲出例外狀況。如果提供者存放區用來產生 CMP，則 CMP 應存放在其持久性儲存體中 (除非它遭到故意刪除)。
     - 如果提供者存放區的持久性儲存體中有具備相符名稱和版本號碼的 CMP，則提供者存放區會將指定的 CMP 傳回給最近提供者。

       如果提供者存放區是 MetaStore，則會從其 DynamoDB 資料表取得加密的 CMP。然後，它會先使用其內部 CMP 提供的密碼編譯資料、將已加密的 CMP 解密，再將 CMP 傳回給最近提供者。如果內部 CMP 是直接 KMS 提供者，這個步驟就包括對 AWS Key Management Service (AWS KMS) 的呼叫。
  2. 最近提供者會快取記憶體中的 CMP。
  3. 最近提供者會使用 CMP 來產生解密資料。接著，它會將解密資料傳回給項目加密程式。

最新提供者的更新

最近提供者的 符號已從 變更為 `MostRecentProvider` `CachingMostRecentProvider`。

> **Note**
>
> 代表最近提供者的 `MostRecentProvider`符號在適用於 Java 的 DynamoDB 加密用戶端 1.15 版和適用於 Python 的 DynamoDB 加密用戶端 1.3 版中被取代，並在兩種語言實作中從 DynamoDB 加密用戶端 2.0.0 版中移除。反之，請使用 `CachingMostRecentProvider`。

`CachingMostRecentProvider` 實作下列變更：

- 當記憶體中的密碼編譯材料時間超過設定的time-to-live(TTL) 值時，會`CachingMostRecentProvider`定期從記憶體中移除密碼編譯材料。

  `MostRecentProvider` 可能會在程序的生命週期內將密碼編譯資料存放在記憶體中。因此，最近提供者可能不知道授權變更。在來電者撤銷使用加密金鑰的許可之後，它可能會使用加密金鑰。

  如果您無法更新至此新版本，您可以定期呼叫快取上的 `clear()`方法，以取得類似的效果。此方法會手動排清快取內容，並要求最近提供者請求新的 CMP 和新的密碼編譯資料。
- `CachingMostRecentProvider` 也包含快取大小設定，可讓您進一步控制快取。

若要更新 `CachingMostRecentProvider`，您必須變更程式碼中的符號名稱。在所有其他方面，`CachingMostRecentProvider` 與 完全向後相容`MostRecentProvider`。您不需要重新加密任何資料表項目。

不過， `CachingMostRecentProvider`會產生更多對基礎金鑰基礎設施的呼叫。它會在每個time-to-live(TTL) 間隔至少呼叫提供者存放區一次。具有許多作用中 CMPs的應用程式 （ 由於頻繁輪換） 或具有大型機群的應用程式最有可能對此變更敏感。

在發佈更新後的程式碼之前，請徹底進行測試，以確保更頻繁的呼叫不會損害您的應用程式，或導致提供者所依賴的服務限流，例如 AWS Key Management Service (AWS KMS) 或 Amazon DynamoDB。若要緩解任何效能問題，`CachingMostRecentProvider`請根據您觀察到的效能特性，調整 的快取大小和time-to-live。如需準則，請參閱設定time-to-live值。

靜態資料提供者

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB 加密用戶端 1.x-2.x 版和適用於 Python 的 DynamoDB 加密用戶端 1.x-3.x 版的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

Static Materials Provider (Static CMP) 是非常簡單的密碼編譯材料提供者 (CMP)，旨在測試、proof-of-concept示範和舊版相容性。

若要使用靜態 CMP 加密資料表項目，請提供進階加密標準 (AES) 對稱加密金鑰與簽署金鑰或金鑰對。您必須提供相同的金鑰，才能將已加密的項目解密。靜態 CMP 不會執行任何密碼編譯操作。它反而會以原狀傳遞您提供給項目加密程式的加密金鑰。項目加密程式會直接以加密金鑰加密項目。然後，直接使用簽署金鑰進行簽署。

因為靜態 CMP 不會產生任何獨特的密碼編譯資料，您處理的所有資料表項目都會使用相同加密金鑰進行加密並由相同的簽署金鑰進行簽署。當您使用相同的金鑰加密眾多項目的屬性值，或使用相同金鑰或金鑰對來簽署所有項目時，可能會超出金鑰的密碼編譯限制。

> **Note**
>
> Java 程式庫中的非對稱靜態提供者不是靜態提供者。其僅提供包裝 CMP 的替代建構函數。此可在生產環境中安全地使用，但只要情況允許，您即應直接使用包裝 CMP。

靜態 CMP 是 DynamoDB Encryption Client 支援的幾個密碼編譯材料提供者 (CMPs) 之一。如需其他 CMP 的相關資訊，請參閱密碼編譯資料提供者。

如需範例程式碼，請參閱：

- Java：SymmetricEncryptedItem

主題

- 使用方式
- 運作方式

使用方式

若要建立靜態提供者，請提供加密金鑰或金鑰對和簽署金鑰或金鑰對。您必須提供金鑰資料才能加密和解密資料表項目。

Java

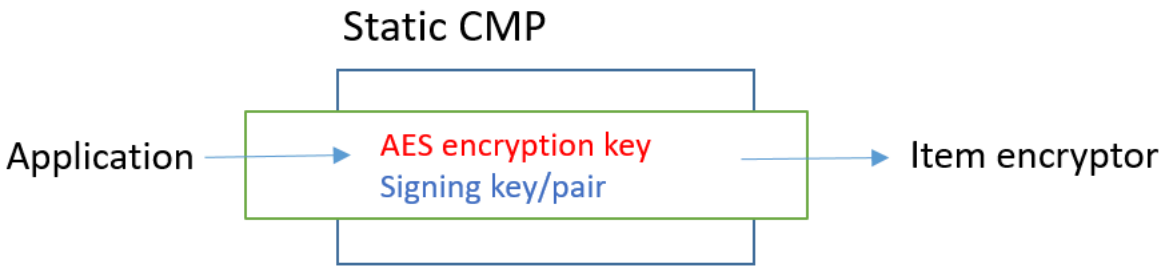
```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

Python

```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
)
```

運作方式

靜態提供者會傳遞您提供給項目加密程式的加密和簽署金鑰，直接用來加密和簽署資料表項目。除非您針對每個項目提供不同的金鑰，否則每個項目都會使用相同金鑰。



取得加密資料

本節將詳細說明靜態資料提供者 (靜態 CMP) 在接收到加密資料請求時的輸入、輸出和處理情形。

輸入 (從應用程式)

- 加密金鑰 – 這必須是對稱金鑰，例如進階加密標準 (AES) 金鑰。
- 簽署金鑰 – 可以是對稱金鑰或非對稱金鑰對。

輸入 (從項目加密程式)

- DynamoDB 加密內容

輸出 (到項目加密程式)

- 當作輸入傳遞的加密金鑰。
- 當作輸入傳遞的簽署金鑰。
- 實際資料描述：請求的資料描述 (如果有) 會維持原狀。

取得解密資料

本節將詳細說明靜態資料提供者 (靜態 CMP) 在接收到解密資料請求時的輸入、輸出和處理情形。

雖然它包括取得加密資料及取得解密資料的個別方法，但是行為相同。

輸入 (從應用程式)

- 加密金鑰 – 這必須是對稱金鑰，例如進階加密標準 (AES) 金鑰。
- 簽署金鑰 – 可以是對稱金鑰或非對稱金鑰對。

輸入 (從項目加密程式)

- DynamoDB 加密內容 （未使用）

輸出 (到項目加密程式)

- 當作輸入傳遞的加密金鑰。
- 當作輸入傳遞的簽署金鑰。

Amazon DynamoDB Encryption Client 可用的程式設計語言

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

Amazon DynamoDB Encryption Client 適用於下列程式設計語言。語言專屬的程式庫有所不同，但是產生的實作都是可互通的。例如，您可以使用 Java 用戶端來加密 (和簽署) 項目，以及使用 Python 用戶端將項目解密。

如需詳細資訊，請參閱相關主題。

主題

- 適用於 Java 的 Amazon DynamoDB 加密用戶端
- 適用於 Python 的 DynamoDB 加密用戶端

適用於 Java 的 Amazon DynamoDB 加密用戶端

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption

Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

本主題說明如何安裝和使用 Amazon DynamoDB Encryption Client for Java。如需使用 DynamoDB 加密用戶端進行程式設計的詳細資訊，請參閱 Java 範例、GitHub 上 aws-dynamodb-encryption-java 儲存庫中的範例，以及 DynamoDB 加密用戶端的 Javadoc。

> **Note**
>
> 適用於 Java 的 DynamoDB Encryption Client 版本 1.x.x 自 2022 年 7 月起處於end-of-support 階段。盡快升級至較新的版本。

主題

- 先決條件
- 安裝
- 使用適用於 Java 的 DynamoDB 加密用戶端
- 適用於 Java 的 DynamoDB 加密用戶端的範例程式碼

先決條件

安裝適用於 Java 的 Amazon DynamoDB Encryption Client 之前，請確定您具備下列先決條件。

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。

適用於 Java 的 AWS SDK

適用於 Java 的 AWS SDK 即使應用程式未與 DynamoDB 互動，DynamoDB Encryption Client 也需要 的 DynamoDB 模組。您可以安裝整個 SDK 或只安裝這個模組。如果您使用 Maven，請將 `aws-java-sdk-dynamodb` 新增到 `pom.xml` 檔案。

如需安裝和設定 的詳細資訊 適用於 Java 的 AWS SDK，請參閱 適用於 Java 的 AWS SDK。

安裝

您可以透過下列方式安裝適用於 Java 的 Amazon DynamoDB Encryption Client。

手動

若要安裝適用於 Java 的 Amazon DynamoDB Encryption Client，請複製或下載 aws-dynamodb-encryption-java GitHub 儲存庫。

使用 Apache Maven

Amazon DynamoDB Encryption Client for Java 可透過 Apache Maven 取得，其依存性定義如下。

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

安裝 SDK 之後，請開始查看本指南中的範例程式碼，以及 GitHub 上的 DynamoDB Encryption Client Javadoc。

使用適用於 Java 的 DynamoDB 加密用戶端

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x—2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x—3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

本主題說明在 Java 中 DynamoDB Encryption Client 的某些功能，這些功能在其他程式設計語言實作中可能找不到。

如需使用 DynamoDB 加密用戶端進行程式設計的詳細資訊，請參閱 Java 範例、GitHub `aws-dynamodb-encryption-java repository` 上的 範例，以及 DynamoDB 加密用戶端的 Javadoc。

主題

- 項目加密程式：AttributeEncryptor 和 DynamoDBEncryptor
- 設定儲存行為
- Java 中的屬性動作
- 覆寫表格名稱

項目加密程式：AttributeEncryptor 和 DynamoDBEncryptor

Java 中的 DynamoDB 加密用戶端有兩個項目加密程式：較低層級的 DynamoDBEncryptor 和 AttributeEncryptor。

`AttributeEncryptor` 是協助程式類別，可協助您在 DynamoDB 加密用戶端的 中使用 DynamoDBMapper 適用於 Java 的 AWS SDK 與 。 `DynamoDB Encryptor` DynamoDB 當您搭配 `DynamoDBMapper` 使用 `AttributeEncryptor` 時，它會在您保存項目時，透明地加密和簽署您的項目。當您載入項目時，它也會透明地驗證和解密您的項目。

設定儲存行為

您可以使用 `AttributeEncryptor`和 `DynamoDBMapper`，將資料表項目新增或取代為僅簽署或加密和簽署的屬性。對於這些任務，我們建議您將其設定為使用 `PUT` 儲存行為，如下列範例所示。否則，您將可能無法解密資料。

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

如果您使用預設儲存行為，只會更新資料表項目中建模的屬性，則未建模的屬性不會包含在簽章中，也不會因資料表寫入而變更。因此，在稍後讀取所有屬性時，簽章將不會驗證，因為它不包含未建模的屬性。

您也可以使用 `CLOBBER` 儲存行為。這種行為與 `PUT` 儲存行為完全相同，差別在於它會停用樂觀鎖定並覆寫表格中的項目。

為了防止簽章錯誤，如果 `AttributeEncryptor` 與未設定儲存行為的 搭配使用`DynamoDBMapper`，則 DynamoDB Encryption Client `CLOBBER` 會擲回執行期例外狀況`PUT`。

若要查看範例中使用的此程式碼，請參閱 GitHub 中`aws-dynamodb-encryption-java`儲存庫中的使用 DynamoDBMapper和 AwsKmsEncryptedObject.java 範例。 AwsKmsEncryptedObject.java

Java 中的屬性動作

屬性動作決定哪些屬性值會加密並簽署，哪些屬性值只會簽署，以及哪些屬性值會予忽略。您用來指定屬性動作的方法取決於您使用的是 `DynamoDBMapper` 和 `AttributeEncryptor`，還是較低層層級的 DynamoDBEncryptor。

> **Important**
>
> 使用屬性動作加密表格項目後，從資料模型新增或移除屬性可能會導致簽章驗證錯誤，讓您無法解密資料。如需更詳細的說明，請參閱變更您的資料模型。

DynamoDBMapper 的屬性動作

當您使用 `DynamoDBMapper` 和 `AttributeEncryptor` 時，您可使用註釋來指定屬性動作。DynamoDB Encryption Client 使用標準 DynamoDB 屬性註釋來定義屬性類型，以判斷如何保護屬性。除了主要索引鍵 (簽署但不加密) 以外，所有屬性都預設會進行加密和簽署。

> **Note**
>
> 雖然您可以 (而且應該) 簽署屬性值，但請勿使用 @DynamoDBVersionAttribute 註釋來加密屬性值。否則，使用其值的情況將會造成想不到的影響。

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

若要指定例外狀況，請使用 DynamoDB Encryption Client for Java 中定義的加密註釋。如果您在類別層級指定例外狀況，這些例外狀況就會成為類別的預設值。

```
// Sign only
@DoNotEncrypt
```

```
// Do nothing; not encrypted or signed
@DoNotTouch
```

例如，這些註釋會簽署但不會加密 `PublicationYear` 屬性，且不會加密或簽署 `ISBN` 屬性值。

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

DynamoDBEncryptor 的屬性動作

若要在您直接使用 DynamoDBEncryptor 時指定屬性動作，請建立 `HashMap` 物件，其中的名稱值組代表屬性名稱和指定的動作。

屬性動作的有效值會定義於 `EncryptionFlags` 列舉類型中。您可以同時使用 `ENCRYPT` 與 `SIGN`、單獨使用 `SIGN`，或省略兩者。不過，如果您`ENCRYPT`單獨使用 ，DynamoDB Encryption Client 會擲回錯誤。您無法加密您未簽署的屬性。

```
ENCRYPT
SIGN
```

> **Warning**
>
> 請勿加密主索引鍵的屬性。它們必須保持純文字，以便 DynamoDB 可以在不執行完整資料表掃描的情況下找到項目。

如果您在加密內容中指定主金鑰，然後在任一主金鑰屬性的屬性動作`ENCRYPT`中指定 ，DynamoDB Encryption Client 會擲回例外狀況。

例如，下列 Java 程式碼會建立 `actions` HashMap，該 HashMap 會加密並簽署 `record` 項目中的所有屬性。例外狀況是分割索引鍵和排序索引鍵屬性 (已簽署但未加密)，以及未簽署或加密的 `test` 屬性。

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
```

```
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

然後，當您呼叫 的 encryptRecord 方法時`DynamoDBEncryptor`，請將映射指定為 `attributeFlags` 參數的值。例如，`encryptRecord` 的這項呼叫會使用 `actions` 映射。

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

覆寫表格名稱

在 DynamoDB Encryption Client 中，DynamoDB 資料表的名稱是 DynamoDB 加密內容的元素，會傳遞給加密和解密方法。當您加密或簽署資料表項目時，DynamoDB 加密內容，包括資料表名稱，會以密碼編譯方式繫結至加密文字。如果傳遞至解密方法的 DynamoDB 加密內容與傳遞至加密方法的 DynamoDB 加密內容不相符，解密操作會失敗。

有時候，表格的名稱會發生變更，例如當您備份表格或執行時間點復原時。當您解密或驗證這些項目的簽章時，您必須傳遞用於加密和簽署項目的相同 DynamoDB 加密內容，包括原始資料表名稱。目前的表格名稱是不需要的。

使用 時`DynamoDBEncryptor`，您可以手動組合 DynamoDB 加密內容。不過，如果您使用的是 `DynamoDBMapper`， 會為您`AttributeEncryptor`建立 DynamoDB 加密內容，包括目前

的資料表名稱。若要告訴 AttributeEncryptor 建立具有不同資料表名稱的加密內容，請使用 EncryptionContextOverrideOperator。

例如，下列程式碼會建立密碼編譯材料提供者 (CMP) 和 DynamoDBEncryptor。然後它會呼叫 DynamoDBEncryptor 的 setEncryptionContextOverrideOperator 方法。它使用覆寫一個表格名稱的 overrideEncryptionContextTableName 運算子。以這種方式設定時， AttributeEncryptor會建立包含 的 DynamoDB 加密內容newTableName，以取代 oldTableName。如需完整的範例，請參閱 EncryptionContextOverridesWithDynamoDBMapper.java。

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

當您呼叫 DynamoDBMapper (解密和驗證項目) 的載入方法時，您可以指定原始資料表名稱。

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

您也可以使用 overrideEncryptionContextTableNameUsingMap 運算子，該運算子會覆寫多個表格名稱。

表格名稱覆寫運算子通常用於解密資料和驗證簽章。不過，您可以在加密和簽署時，使用它們將 DynamoDB 加密內容中的資料表名稱設定為不同的值。

如果您使用的是 DynamoDBEncryptor，請勿使用表格名稱覆寫運算子。請改為使用原始表格名稱建立加密內容，並將其提交至解密方法。

適用於 Java 的 DynamoDB 加密用戶端的範例程式碼

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x—2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x—3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

下列範例示範如何使用適用於 Java 的 DynamoDB 加密用戶端來保護應用程式中的 DynamoDB 資料表項目。您可以在 GitHub 上 aws-dynamodb-encryption-java 儲存庫的範例目錄中找到更多範例（並自行提供）。

主題

- 使用 DynamoDBEncryptor
- 使用 DynamoDBMapper

使用 DynamoDBEncryptor

這個範例說明如何使用較低層級的 DynamoDBEncryptor 搭配直接 KMS 提供者。Direct KMS 提供者會在您指定的 AWS KMS key in AWS Key Management Service (AWS KMS) 下產生並保護其密碼編譯資料。

您可以使用任何相容的密碼編譯資料提供者 (CMP) 搭配 `DynamoDBEncryptor`，而且可以使用直接 KMS 提供者搭配 `DynamoDBMapper` 與 AttributeEncryptor。

查看完整的程式碼範例：AwsKmsEncryptedItem.java

步驟 1：建立直接 KMS 提供者

建立具有指定區域的 AWS KMS 用戶端執行個體。然後，使用用戶端執行個體，使用您偏好的 建立 Direct KMS 提供者執行個體 AWS KMS key。

此範例使用 Amazon Resource Name (ARN) 來識別 AWS KMS key，但您可以使用任何有效的金鑰識別符。

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

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

步驟 2：建立項目

這個範例會定義 `record` HashMap，其代表範例資料表項目。

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";
```

```
final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00,
 0x01, 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

步驟 3：建立 DynamoDBEncryptor

使用直接 KMS 提供者建立 `DynamoDBEncryptor` 的執行個體。

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

步驟 4：建立 DynamoDB 加密內容

DynamoDB 加密內容包含資料表結構及其加密和簽署方式的相關資訊。如果您使用 `DynamoDBMapper`，`AttributeEncryptor` 會為您建立加密細節。

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

步驟 5：建立屬性動作物件

屬性動作決定項目的哪些屬性會加密並簽署，哪些屬性只會簽署，以及哪些屬性不會加密或簽署。

在 Java 中，若要指定屬性動作，您可建立屬性名稱和 `EncryptionFlags` 值組的 HashMap。

例如，下列 Java 程式碼會建立 `actions` HashMap，其可加密並簽署 `record` 項目中的所有屬性，但是分割區索引鍵與排序索引鍵屬性 (簽署但不加密) 以及 `test` 屬性 (不簽署或加密) 除外。

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
```

```
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

步驟 6：將項目加密並簽署

若要加密並簽署資料表項目，請在 `encryptRecord` 的執行個體上呼叫 `DynamoDBEncryptor` 方法。指定資料表項目 (`record`)、屬性動作 (`actions`) 及加密細節 (`encryptionContext`)。

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

步驟 7：將項目放入 DynamoDB 資料表

最後，將加密和簽署的項目放入 DynamoDB 資料表。

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

使用 DynamoDBMapper

下列範例示範如何搭配 Direct KMS 提供者使用 DynamoDB 映射器協助程式類別。Direct KMS 提供者會在您指定的 AWS KMS key AWS Key Management Service (AWS KMS) 中產生並保護其密碼編譯資料。

您可以使用任何相容的密碼編譯資料提供者 (CMP) 搭配 `DynamoDBMapper`，而且可以使用直接 KMS 提供者搭配較低層級的 `DynamoDBEncryptor`。

查看完整的程式碼範例：AwsKmsEncryptedObject.java

步驟 1：建立直接 KMS 提供者

建立具有指定區域的 AWS KMS 用戶端執行個體。然後，使用用戶端執行個體，使用您偏好的 建立 Direct KMS 提供者執行個體 AWS KMS key。

此範例使用 Amazon Resource Name (ARN) 來識別 AWS KMS key，但您可以使用任何有效的金鑰識別符。

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

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

步驟 2：建立 DynamoDB 加密程式和 DynamoDBMapper

使用您在上一個步驟中建立的直接 KMS 提供者來建立 DynamoDB 加密程式的執行個體。您需要執行個體化較低層級的 DynamoDB Encryptor，才能使用 DynamoDB Mapper。

接著，建立 DynamoDB 資料庫的執行個體和映射器組態，並使用它們來建立 DynamoDB Mapper 的執行個體。

> **Important**
>
> 使用 `DynamoDBMapper` 加入或編輯已簽署 (或已加密並簽署) 的項目時，請將其設定成使用儲存行為 (例如 PUT) 納入所有屬性，如以下範例所示。否則，您將可能無法解密資料。

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

步驟 3：定義您的 DynamoDB 資料表

接著，定義您的 DynamoDB 資料表。請使用註釋指定屬性動作。此範例會建立代表資料表項目的 DynamoDB 資料表`ExampleTable`、 和 `DataPoJo`類別。

該範例資料表的主索引鍵屬性將經過簽署但未加密。這包括了標註 @DynamoDBHashKey 的 partition_attribute，以及標註 @DynamoDBRangeKey 的 sort_attribute。

凡是標註 @DynamoDBAttribute 的屬性 (例如 some numbers) 都將進行加密並簽署。例外狀況是使用 DynamoDB Encryption Client 定義的 @DoNotEncrypt（僅簽署） 或 @DoNotTouch（不加密或簽署） 加密註釋的屬性。例如，leave me 屬性由於設有 @DoNotTouch 註釋，其將不會進行加密或簽署。

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "example")
  public String getExample() {
    return example;
  }

  public void setExample(String example) {
    this.example = example;
  }
```

```
  @DynamoDBAttribute(attributeName = "some numbers")
  public long getSomeNumbers() {
    return someNumbers;
  }

  public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
  }

  @DynamoDBAttribute(attributeName = "and some binary")
  public byte[] getSomeBinary() {
    return someBinary;
  }

  public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
  }

  @DynamoDBAttribute(attributeName = "leave me")
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe +
 "]";
  }
}
```

步驟 4：加密並儲存資料表項目

現在，當您建立資料表項目並使用 DynamoDB Mapper 儲存該項目時，該項目會在新增至資料表之前自動加密和簽署。

本範例定義的資料表項目名為 `record`。該項目儲存至資料表之前，其屬性將根據 `DataPoJo` 類別中的註釋進行加密與簽署。就本例而言，所有屬性除了 `PartitionAttribute`、`SortAttribute` 和 `LeaveMe` 以外都將加密並簽署。`PartitionAttribute` 和 `SortAttributes` 只會進行簽署，而 `LeaveMe` 屬性則完全未加密或簽署。

若要加密並簽署 `record` 項目，然後將其加入至 `ExampleTable`，請呼叫 `DynamoDBMapper` 類別的 `save` 方法。由於您的 DynamoDB Mapper 已設定為使用PUT儲存行為，因此項目會使用相同的主索引鍵取代任何項目，而不是更新它。這可確保簽章相符，而且您從資料表取得該項目後便能將其解密。

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");

mapper.save(record);
```

適用於 Python 的 DynamoDB 加密用戶端

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

本主題說明如何安裝和使用適用於 Python 的 DynamoDB Encryption Client。您可以在 GitHub 上的 aws-dynamodb-encryption-python 儲存庫中找到程式碼 (包括完整和已測試的範本程式碼)，協助您開始使用。

> **Note**
>
> 適用於 Python 的 DynamoDB Encryption Client 版本 1.x.x 和 2.x.x 自 2022 年 7 月起處於end-of-support階段。盡快升級至較新的版本。

主題

- 先決條件
- 安裝
- 使用適用於 Python 的 DynamoDB 加密用戶端
- DynamoDB Encryption Client for Python 的範例程式碼

先決條件

在安裝適用於 Python 的 Amazon DynamoDB Encryption Client 之前，請確定您有下列先決條件。

支援的 Python 版本

Amazon DynamoDB Encryption Client for Python 3.3.0 版及更新版本需要 Python 3.8 或更新版本。若要下載 Python，請參閱 Python 下載。

適用於 Python 的 Amazon DynamoDB Encryption Client 較早版本支援 Python 2.7 和 Python 3.4 及更新版本，但我們建議您使用最新版本的 DynamoDB Encryption Client。

適用於 Python 的 pip 安裝工具

Python 3.6 和更新版本包含 pip，但您可能想要升級。如需有關升級或安裝 pip 的詳細資訊，請參閱 pip 文件中的安裝。

安裝

使用 pip 安裝適用於 Python 的 Amazon DynamoDB Encryption Client，如下列範例所示。

若要安裝最新版本

```
pip install dynamodb-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。

DynamoDB 加密用戶端在所有平台上都需要密碼編譯程式庫。Windows 上所有版本的 pip 都將安裝並建置密碼編譯程式庫，而 Linux 上的 pip 8.1 和更新版本則會安裝並建置密碼編譯。如果您使用舊版 pip，而且您的 Linux 環境沒有建置密碼編譯程式庫所需的工具，您就需要加以安裝。如需詳細資訊，請參閱在 Linux 上建置密碼編譯。

您可以從 GitHub 上的 aws-dynamodb-encryption-python 儲存庫取得最新的 DynamoDB Encryption Client 開發版本。

安裝 DynamoDB Encryption Client 之後，請開始查看本指南中的範例 Python 程式碼。

使用適用於 Python 的 DynamoDB 加密用戶端

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x—2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x—3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

本主題說明 DynamoDB Encryption Client for Python 的部分功能，這些功能在其他程式設計語言實作中可能找不到。這些功能旨在讓您更輕鬆地以最安全的方式使用 DynamoDB Encryption Client。若非遇到罕見的使用案例，我們都建議您使用這些功能。

如需使用 DynamoDB Encryption Client 進行程式設計的詳細資訊，請參閱本指南中的 Python 範例、GitHub 上 aws-dynamodb-encryption-python 儲存庫中的範例，以及 DynamoDB Encryption Client 的 Python 文件。

主題

- 用戶端協助程式類別
- TableInfo 類別
- Python 中的屬性動作

用戶端協助程式類別

適用於 Python 的 DynamoDB Encryption Client 包含數個用戶端協助程式類別，這些類別會反映 DynamoDB 的 Boto 3 類別。這些協助程式類別旨在讓您更輕鬆地將加密和簽署新增至現有的 DynamoDB 應用程式，並避免最常見的問題，如下所示：

- 藉由將主索引鍵的覆寫動作新增至 AttributeActions 物件，或藉由在您的 AttributeActions 物件明確指示用戶端為主索引鍵加密時擲回例外狀況，以防止您將項目中的主索引鍵加密。如果 AttributeActions 物件中的預設動作為 DO_NOTHING，則用戶端協助程式類別會將該動作用於主索引鍵。否則會使用 SIGN_ONLY。
- 建立 TableInfo 物件，並根據對 DynamoDB 的呼叫填入 DynamoDB 加密內容。 DynamoDB 這有助於確保您的 DynamoDB 加密內容準確，且用戶端可以識別主金鑰。
- 當您寫入 DynamoDB 資料表或從 DynamoDB 資料表讀取時get_item，可透明加密和解密資料表項目的支援方法，例如 put_item和 。只有 update_item 方法不受支援。

您可以使用用戶端協助程式類別，而無須直接與較低層級的項目加密程式互動。只有在需要於項目加密程式中設定進階選項時，才使用這些類別。

用戶端協助程式類別包括：

- EncryptedTable 適用於在 DynamoDB 中使用資料表資源的應用程式，一次處理一個資料表。
- EncryptedResource 適用於在 DynamoDB 中使用 Service Resource 類別進行批次處理的應用程式。
- EncryptedClient 適用於在 DynamoDB 中使用較低層級用戶端的應用程式。

若要使用用戶端協助程式類別，呼叫者必須具有在目標資料表上呼叫 DynamoDB DescribeTable 操作的許可。

TableInfo 類別

TableInfo 類別是代表 DynamoDB 資料表的協助程式類別，其主要索引鍵和次要索引的欄位為完整。它可協助您取得正確而即時的資料表相關資訊。

如果您使用的是用戶端協助程式類別，其將為您建立並使用 TableInfo 物件。否則，您可以明確建立一個物件。如需範例，請參閱「使用項目加密程式」。

當您在TableInfo物件上呼叫 refresh_indexed_attributes方法時，它會呼叫 DynamoDB DescribeTable 操作來填入物件的屬性值。查詢資料表會比進行索引名稱的硬編碼可靠得多。TableInfo 類別也包含 encryption_context_values 屬性，提供 DynamoDB 加密內容所需的值。

若要使用 refresh_indexed_attributes方法，發起人必須具有在目標資料表上呼叫 DynamoDB DescribeTable 操作的許可。

Python 中的屬性動作

屬性動作會告知項目加密程式要對項目的每個屬性執行什麼動作。若要指定 Python 中的屬性動作，請建立具有預設動作和特定屬性之任何例外狀況的 `AttributeActions` 物件。有效值會在 `CryptoAction` 列舉類型中加以定義。

> **Important**
>
> 使用屬性動作加密表格項目後，從資料模型新增或移除屬性可能會導致簽章驗證錯誤，讓您無法解密資料。如需更詳細的說明，請參閱變更您的資料模型。

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

例如，此 `AttributeActions` 物件會建立 `ENCRYPT_AND_SIGN` 作為所有屬性的預設值，並指定 `ISBN` 和 `PublicationYear` 屬性的例外狀況。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

若您使用的是用戶端協助程式類別，您不需要指定主索引鍵屬性的屬性動作。用戶端協助程式類別可防止您將主索引鍵加密。

若未使用用戶端協助程式類別，且預設動作為 `ENCRYPT_AND_SIGN`，則必須指定主索引鍵的動作。建議的主索引鍵動作為 `SIGN_ONLY`。若要加以簡化，請使用會對主索引鍵使用 SIGN_ONLY 的 `set_index_keys` 方法，或是在預設動作為 DO_NOTHING 時使用此動作。

> **Warning**
>
> 請勿加密主索引鍵的屬性。它們必須保持純文字，以便 DynamoDB 可以在不執行完整資料表掃描的情況下找到項目。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

DynamoDB Encryption Client for Python 的範例程式碼

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

下列範例示範如何使用適用於 Python 的 DynamoDB Encryption Client 來保護應用程式中的 DynamoDB 資料。您可以在 GitHub 上 aws-dynamodb-encryption-python 儲存庫的範例目錄中找到更多範例（並自行貢獻）。

主題

- 使用 EncryptedTable 用戶端協助程式類別
- 使用項目加密程式

使用 EncryptedTable 用戶端協助程式類別

下列範例示範如何搭配`EncryptedTable`用戶端協助程式類別使用 Direct KMS Provider。此範例使用與以下使用項目加密程式範例相同的密碼編譯材料提供者。不過，其將使用 `EncryptedTable` 類別，而不是直接與較低層級的項目加密程式互動。

比較這些範例，您就可以看見用戶端協助程式類別為您所做的事情。這包括建立 DynamoDB 加密內容，並確保主要金鑰屬性一律已簽章，但絕不會加密。若要建立加密內容並探索主金鑰，用戶端協助程式類別會呼叫 DynamoDB DescribeTable 操作。若要執行此程式碼，您必須擁有呼叫此作業的權利。

查看完整的程式碼範例：aws_kms_encrypted_table.py

步驟 1：建立資料表

首先，使用資料表名稱建立標準 DynamoDB 資料表的執行個體。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

步驟 2：建立密碼編譯資料提供者

建立您所選密碼編譯資料提供者 (CMP) 的執行個體。

本範例使用直接 KMS 提供者，但您可以使用任何相容的 CMP。若要建立直接 KMS 提供者，請指定 AWS KMS key。此範例使用 的 Amazon Resource Name (ARN) AWS KMS key，但您可以使用任何有效的金鑰識別符。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

步驟 3：建立屬性動作物件

屬性動作會告知項目加密程式要對項目的每個屬性執行什麼動作。這個範例中的 `AttributeActions` 物件會加密並簽署所有項目，但 `test` 屬性 (予以忽略) 除外。

當您使用用戶端協助程式類別時，請勿指定主要索引鍵屬性的屬性動作。`EncryptedTable` 類別會簽署 (但絕不會加密) 主要索引鍵屬性。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

步驟 4：建立已加密的資料表

使用標準資料表、直接 KMS 提供者和屬性動作，來建立已加密的資料表。這個步驟可完成設定。

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

步驟 5：在資料表中放入純文字項目

當您在 上呼叫 `put_item`方法時`encrypted_table`，您的資料表項目會透明加密、簽署，並新增至您的 DynamoDB 資料表。

首先，定義資料表項目。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

然後，在資料表中放入該項目。

```
encrypted_table.put_item(Item=plaintext_item)
```

若要從 DynamoDB 資料表以加密形式取得項目，請呼叫 `table` 物件上的 `get_item`方法。若要取得已解密的項目，請在 `get_item` 物件上呼叫 `encrypted_table` 方法。

使用項目加密程式

此範例說明如何在加密資料表項目時，直接與 DynamoDB Encryption Client 中的項目加密器互動，而不是使用與您項目加密器互動的用戶端協助程式類別。 DynamoDB

當您使用此技術時，您可以手動建立 DynamoDB 加密內容和組態物件 (`CryptoConfig`)。此外，您會在一個呼叫中加密項目，並在另一個呼叫中將其放入 DynamoDB 資料表中。這可讓您自訂`put_item`呼叫，並使用 DynamoDB Encryption Client 來加密和簽署從未傳送至 DynamoDB 的結構化資料。

本範例使用直接 KMS 提供者，但您可以使用任何相容的 CMP。

查看完整的程式碼範例：aws_kms_encrypted_item.py

步驟 1：建立資料表

首先，使用資料表名稱建立標準 DynamoDB 資料表資源的執行個體。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

步驟 2：建立密碼編譯資料提供者

建立您所選密碼編譯資料提供者 (CMP) 的執行個體。

本範例使用直接 KMS 提供者，但您可以使用任何相容的 CMP。若要建立直接 KMS 提供者，請指定 AWS KMS key。此範例使用 的 Amazon Resource Name (ARN) AWS KMS key，但您可以使用任何有效的金鑰識別符。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

步驟 3：使用 TableInfo 協助程式類別

若要從 DynamoDB 取得資料表的相關資訊，請建立 TableInfo 協助程式類別的執行個體。當您直接使用項目加密程式時，您需要建立 `TableInfo` 執行個體及呼叫其方法。用戶端協助程式類別會為您執行此作業。

`refresh_indexed_attributes` 方法`TableInfo`使用 DescribeTable DynamoDB 操作來取得有關資料表的即時準確資訊。這包括其主要索引鍵及其本機和全域輔助索引。呼叫端必須具備呼叫 `DescribeTable` 的許可。

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

步驟 4：建立 DynamoDB 加密內容

DynamoDB 加密內容包含資料表結構及其加密和簽署方式的相關資訊。此範例會明確建立 DynamoDB 加密內容，因為它會與項目加密程式互動。用戶端協助程式類別會為您建立 DynamoDB 加密內容。

若要取得分割區索引鍵和排序索引鍵，您可以使用 TableInfo 協助程式類別的屬性。

```
index_key = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
    partition_key_name=table_info.primary_index.partition,
```

```
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

步驟 5：建立屬性動作物件

屬性動作會告知項目加密程式要對項目的每個屬性執行什麼動作。這個範例中的 `AttributeActions` 物件會加密並簽署所有項目，但主要索引鍵屬性 (簽署但不加密) 和 `test` 屬性 (予以忽略) 除外。

當您直接與項目加密程式互動且預設動作為 `ENCRYPT_AND_SIGN` 時，您必須為主要索引鍵指定替代動作。您可以使用 `set_index_keys` 方法，該方法針對主要索引鍵使用 `SIGN_ONLY`，或使用 `DO_NOTHING` (如果這是預設動作)。

若要指定主索引鍵，此範例會使用 TableInfo 物件中的索引鍵，這會由呼叫 DynamoDB 填入。這項技巧比硬編碼主要索引鍵名稱還要安全。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
actions.set_index_keys(*table_info.protected_index_keys())
```

步驟 6：建立項目的組態

若要設定 DynamoDB 加密用戶端，請使用您剛在資料表項目的 CryptoConfig 組態中建立的物件。用戶端協助程式類別會為您建立 CryptoConfig。

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

步驟 7：將項目加密

此步驟會加密並簽署項目，但不會將其放入 DynamoDB 資料表中。

當您使用用戶端協助程式類別時，您的項目會透明加密和簽署，然後在呼叫協助程式類別`put_item`的方法時新增至 DynamoDB 資料表。當您直接使用項目加密程式時，加密和放置動作都是獨立的。

首先，建立純文字項目。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

然後，將它加密並簽署。`encrypt_python_item` 方法需要 `CryptoConfig` 組態物件。

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

步驟 8：在資料表中放入此項目

此步驟會將加密和簽署的項目放入 DynamoDB 資料表。

```
table.put_item(Item=encrypted_item)
```

若要檢視已加密的項目，請在原始 `get_item` 物件 (而非 `table` 物件) 上呼叫 `encrypted_table` 方法。不需進行驗證或解密，即可從 DynamoDB 資料表取得項目。

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

下圖顯示一部分已加密並簽署資料表項目的範例。

已加密的屬性值為二進位資料。主要索引鍵屬性 (`partition_attribute` 和 `sort_attribute`) 及 `test` 屬性的名稱和值都保持純文字形式。輸出也會顯示包含簽章 (`*amzn-ddb-map-sig*`) 的屬性和資料描述屬性 (`*amzn-ddb-map-desc*`)。

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```


變更您的資料模型

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x—2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x—3.x 的相關資訊。如需詳細資訊，請參閱 AWS DynamoDB 版本支援的資料庫加密 SDK。

每次加密或解密項目時，您需要提供屬性動作，以告知 DynamoDB Encryption Client 要加密和簽署哪些屬性、要簽署哪些屬性（但不要加密），以及要忽略哪些屬性。屬性動作不會儲存在加密的項目中，DynamoDB Encryption Client 不會自動更新您的屬性動作。

> **Important**
>
> DynamoDB Encryption Client 不支援加密現有、未加密的 DynamoDB 資料表資料。

每當您變更資料模型時，也就是，當您新增或移除資料表項目中的屬性時，就可能發生錯誤。如果您指定的屬性動作並未考量項目中的所有屬性，此項目便無法如您預期的方式進行加密和簽署。更重要的是，如果您在加密項目時提供的屬性動作與您在解密項目時提供的屬性動作不同，則簽章驗證可能會失敗。

例如，如果用來加密項目的屬性動作告知要簽署 `test` 屬性，則項目中的簽章會包含 `test` 屬性。但是如果用來解密項目的屬性動作並未考量 `test` 屬性，則驗證會因為用戶端嘗試驗證不包含 `test` 屬性的簽章而失敗。

當多個應用程式讀取和寫入相同的 DynamoDB 項目時，這是一個特別的問題，因為 DynamoDB 加密用戶端必須計算所有應用程式中項目的相同簽章。對於任何分散式應用程式來說，這也是一個問題，因為屬性動作的變更必須傳播到所有主機。即使您的 DynamoDB 資料表是由一個主機在一個程序中存取，建立最佳實務程序也有助於在專案變得更加複雜時防止錯誤。

若要避免使您無法讀取資料表項目的簽章驗證錯誤，請使用下列指導。

- 新增屬性 — 如果新屬性變更您的屬性動作，請在項目中包含新屬性之前，先完全部署屬性動作變更。
- 移除屬性 — 如果您在項目中停止使用屬性，請勿變更屬性動作。
- 變更動作 — 在您使用屬性動作組態來加密資料表項目之後，您無法安全地變更現有屬性的預設動作或動作，而無需重新加密資料表中的每個項目。

簽章驗證錯誤可能非常難以解決，因此最好的方法是避免這些錯誤的發生。

主題

- 新增屬性
- 移除屬性

新增屬性

當您將新屬性新增至資料表項目時，您可能需要變更屬性動作。若要避免簽章驗證錯誤，我們建議您在兩階段的程序中實作此變更。在啟動第二個階段之前，請確認第一個階段已完成。

1. 在所有讀取或寫入資料表的應用程式中變更屬性動作。部署這些變更，並確認更新已傳播到所有目的地主機。
2. 將值寫入資料表項目中的新屬性。

這個兩階段的方法可確保所有應用程式和主機都具有相同的屬性動作，並在遇到新屬性之前計算相同的簽章。即使屬性的動作是不執行任何動作 (不加密或簽署)，這個方法仍很重要，因為某些加密器的預設值是加密和簽署。

下列範例顯示此程序中第一個階段的程式碼。這些範例會新增項目屬性 link，此屬性會存放另一個資料表項目的連結。此連結必須維持純文字的形式，因此此範例會為其指派僅簽署動作。完全部署此變更，然後確認所有應用程式和主機都有新的屬性動作之後，您就可以開始在資料表項目中使用該 link 屬性。

Java DynamoDB Mapper

使用 DynamoDB Mapper 和 AttributeEncryptor 時，主要索引鍵 (簽署但不加密) 以外，所有屬性都預設會進行加密和簽署。若要指定僅簽署動作，請使用 @DoNotEncrypt 註釋。

此範例會針對新 link 屬性使用 @DoNotEncrypt 註釋。

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
```

```
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
}
```

Java DynamoDB encryptor

在較低層級的 DynamoDB 加密程式中，您必須為每個屬性設定動作。此範例使用 switch 陳述式，其中預設值為 `encryptAndSign`，而且會針對分割索引鍵、排序索引鍵和新 `link` 屬性指定例外狀況。在此範例中，如果在使用連結屬性程式碼之前未將其完全部署，則某些應用程式會加密並簽署此連結屬性，但其他應用程式則僅簽署此連結屬性。

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
}
```

Python

在適用於 Python 的 DynamoDB Encryption Client 中，您可以為所有屬性指定預設動作，然後指定例外狀況。

若您使用的是 Python 用戶端協助程式類別，您不需要指定主索引鍵屬性的屬性動作。用戶端協助程式類別可防止您將主索引鍵加密。但是，如果您不使用用戶端協助程式類別，則必須在您的分割區索引鍵和排序索引鍵上設定 SIGN_ONLY 動作。如果您不小心加密了分割區或排序索引鍵，您將無法在沒有進行完整資料表掃描的情況下復原資料。

此範例會指定新 `link` 屬性 (取得 `SIGN_ONLY` 動作) 的例外狀況。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

移除屬性

如果您在已使用 DynamoDB Encryption Client 加密的項目中不再需要 屬性，則可以停止使用 屬性。但是，請勿刪除或變更該屬性的動作。如果您這樣做，然後遇到具有該屬性的項目，則針對該項目計算的簽章將不符合原始簽章，且簽章驗證將會失敗。

雖然您可能想要從程式碼中移除屬性的所有追蹤，但請新增註解，指出該項目已不再使用，而不是將其刪除。即使您執行完整資料表掃描以刪除該屬性的所有執行個體，具有該屬性的加密項目可能會被快取或在組態中的某處處理中。

故障診斷 DynamoDB Encryption Client 應用程式的問題

> **Note**
>
> 我們的用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。下列主題提供適用於 Java 的 DynamoDB Encryption Client 版本 1.x-2.x 和適用於 Python 的 DynamoDB Encryption Client 版本 1.x-3.x 的相關資訊。如需詳細資訊，請參閱AWS 資料庫加密 SDK for DynamoDB 版本支援。

本節說明使用 DynamoDB 加密用戶端時可能遇到的問題，並提供解決這些問題的建議。

若要提供 DynamoDB 加密用戶端的意見回饋，請在 aws-dynamodb-encryption-java 或 aws-dynamodb-encryption-python GitHub 儲存庫中提交問題。

若要提供有關此文件的意見回饋，請使用任何頁面上的意見回饋連結。

主題

- 存取遭拒
- 簽章驗證失敗
- 舊版全域資料表的問題
- 最近提供者的效能不佳

存取遭拒

問題：您的應用程式遭拒，無法存取其所需的資源。

建議：了解必要的許可，並將這些許可新增至您的應用程式執行所在的安全性細節。

詳細資訊

若要執行使用 DynamoDB Encryption Client 程式庫的應用程式，發起人必須具有使用其元件的許可。否則他們會遭拒，無法存取所需的元素。

- DynamoDB Encryption Client 不需要 Amazon Web Services (AWS) 帳戶或依賴任何 AWS 服務。不過，如果您的應用程式使用 AWS，您需要 AWS 帳戶和 具有使用 帳戶許可的使用者。
- DynamoDB 加密用戶端不需要 Amazon DynamoDB。不過，如果使用 用戶端的應用程式建立 DynamoDB 資料表、將項目放入資料表，或從資料表取得項目，呼叫者必須具有在您的 中使用所需 DynamoDB 操作的許可 AWS 帳戶。如需詳細資訊，請參閱《Amazon DynamoDB 開發人員指南》中的存取控制主題。
- 如果您的應用程式在 DynamoDB Encryption Client for Python 中使用用戶端協助程式類別，呼叫者必須具有呼叫 DynamoDB DescribeTable 操作的許可。
- DynamoDB 加密用戶端不需要 AWS Key Management Service (AWS KMS)。不過，如果您的應用程式使用 Direct KMS 材料提供者，或者其使用最近提供者搭配使用的供應商存放區 AWS KMS，則發起人必須具有使用 AWS KMS GenerateDataKey 和 Decrypt 操作的許可。

簽章驗證失敗

問題：項目因為簽章驗證失敗而無法解密。此項目也無法如您所願進行加密和簽署。

建議：確保您提供的屬性動作會考量項目中的所有屬性。將項目解密時，請務必提供與用來解密項目之動作相符的屬性動作。

詳細資訊

您提供的屬性動作會告知 DynamoDB Encryption Client 要加密和簽署的屬性、要簽署的屬性（但不要加密），以及要忽略的屬性。

如果您指定的屬性動作並未考量項目中的所有屬性，此項目便無法如您預期的方式進行加密和簽署。如果您在加密項目時提供的屬性動作與您在解密項目時提供的屬性動作不同，則簽章驗證可能會失敗。這是一個特殊問題，出現於新屬性動作尚未傳播到所有主機的分散式應用程式中。

簽章驗證錯誤很難解決。如需協助防範其發生，請在變更資料模型時採取額外的預防措施。如需詳細資訊，請參閱 變更您的資料模型。

舊版全域資料表的問題

問題：舊版 Amazon DynamoDB 全域資料表中的項目無法解密，因為簽章驗證失敗。

建議：設定屬性動作，讓保留的複寫欄位不會加密或簽署。

詳細資訊

您可以使用 DynamoDB Encryption Client 搭配 DynamoDB 全域資料表。我們建議您使用具有多區域 KMS 金鑰的全域資料表，並將 KMS 金鑰複寫到複寫全域資料表的所有 AWS 區域 位置。

從全域資料表 2019.11.21 版開始，您可以搭配 DynamoDB Encryption Client 使用全域資料表，而不需要任何特殊組態。不過，如果您使用全域資料表 2017.11.29 版，您必須確保保留的複寫欄位未加密或簽署。

如果您使用的是全域資料表 2017.11.29 版，則必須在 `DO_NOTHING` Java 或 Python `@DoNotTouch`中將下列屬性的屬性動作設為 。

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

如果您使用的是任何其他版本的全域資料表，則不需要任何動作。

最近提供者的效能不佳

問題：您的應用程式回應較差，特別是在更新至較新版本的 DynamoDB Encryption Client 之後。

建議：調整time-to-live值和快取大小。

詳細資訊

最新提供者旨在透過允許有限的密碼編譯材料重複使用，改善使用 DynamoDB 加密用戶端的應用程式效能。當您為應用程式設定最新提供者時，您必須平衡改善的效能與快取和重複使用引起的安全問題。

在較新版本的 DynamoDB Encryption Client 中，time-to-live(TTL) 值會決定快取密碼編譯材料提供者 CMPs) 的使用時間。TTL 也會判斷最近提供者檢查 CMP 新版本的頻率。

如果您的 TTL 太長，您的應用程式可能會違反您的業務規則或安全標準。如果您的 TTL 太短，經常呼叫提供者商店可能會導致提供者商店限流來自您應用程式和其他共用您服務帳戶的應用程式的請求。若要解決此問題，請將 TTL 和快取大小調整為符合您延遲和可用性目標的值，並符合您的安全標準。如需詳細資訊，請參閱設定time-to-live值。

Amazon DynamoDB Encryption Client 重新命名

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

2023 年 6 月 9 日，我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。 AWS Database Encryption SDK 與 Amazon DynamoDB 相容。它可以解密和讀取舊版 DynamoDB Encryption Client 加密的項目。如需舊版 DynamoDB Encryption Client 版本的詳細資訊，請參閱 AWS DynamoDB 版本的資料庫加密 SDK 支援。

AWS Database Encryption SDK 提供適用於 DynamoDB 的 Java 用戶端加密程式庫 3.x 版，這是適用於 Java 的 DynamoDB Encryption Client 的主要重寫。它包含許多更新，例如新的結構化資料格式、改善的多租戶支援、無縫結構描述變更，以及可搜尋的加密支援。

若要進一步了解 AWS 資料庫加密 SDK 引進的新功能，請參閱下列主題。

可搜尋加密

您可以設計資料庫來搜尋加密的記錄，而無需解密整個資料庫。根據您的威脅模型和查詢需求，您可以使用可搜尋加密，對加密的記錄執行完全相符的搜尋或更自訂的複雜查詢。

Keyrings

AWS Database Encryption SDK 使用 keyring 來執行信封加密。Keyrings 會產生、加密和解密保護記錄的資料金鑰。 AWS Database Encryption SDK 支援使用對稱加密或非對稱 RSA AWS KMS keys保護資料金鑰的 AWS KMS keyring，以及 AWS KMS 階層式 keyring，可讓您以對稱加密 KMS 金鑰保護密碼編譯資料，而不必 AWS KMS 在每次加密或解密記錄時呼叫 。您也可以使用原始 AES keyring 和原始 RSA keyring 指定自己的金鑰材料。

無縫結構描述變更

當您設定 AWS 資料庫加密 SDK 時，您會提供密碼編譯動作，告訴用戶端要加密和簽署哪些欄位、要簽署哪些欄位（但不要加密），以及要忽略哪些欄位。使用 AWS Database Encryption SDK 來保護您的記錄之後，您仍然可以變更資料模型。您可以在單一部署中更新您的密碼編譯動作，例如新增或移除加密的欄位。

設定現有的 DynamoDB 資料表進行用戶端加密

DynamoDB Encryption Client 的舊版設計用於在新的未填入資料表中實作。使用適用於 DynamoDB 的 AWS Database Encryption SDK，您可以將現有的 Amazon DynamoDB 資料表遷移至適用於 DynamoDB 的 Java 用戶端加密程式庫的 3.x 版。

參考資料

我們的用戶端加密程式庫已重新命名為 AWS 資料庫加密 SDK。此開發人員指南仍會提供 DynamoDB Encryption Client 的相關資訊。

下列主題提供 AWS Database Encryption SDK 的技術詳細資訊。

材質描述格式

材料描述做為加密記錄的標頭。當您使用 AWS 資料庫加密 SDK 加密和簽署欄位時，加密程式會在其組合密碼編譯材料時記錄材料描述，並將材料描述存放在加密程式新增至記錄的新欄位 (`aws_dbe_head`) 中。材料描述是一種可攜式格式化資料結構，其中包含加密的資料金鑰，以及記錄加密和簽署方式的相關資訊。下表說明組成物料描述的值。位元組依顯示順序附加。

Value	位元組長度
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	變數
Encryption Context Length	2
???	變數
Encrypted Data Key Count	1
Encrypted Data Keys	變數
Record Commitment	1

版本

`aws_dbe_head` 此欄位格式的版本。

已啟用簽章

編碼此記錄是否已啟用 ECDSA 數位簽章。

位元組值	意義
0x01	ECDSA 數位簽章已啟用（預設）
0x00	ECDSA 數位簽章已停用

記錄 ID

隨機產生的 256 位元值，可識別記錄。記錄 ID：

- 唯一識別加密的記錄。
- 將材料描述繫結至加密的記錄。

加密圖例

已加密已驗證欄位的序列化描述。Encrypt 圖例用於判斷解密方法應該嘗試解密的欄位。

位元組值	意義
0x65	ENCRYPT_AND_SIGN
0x73	SIGN_ONLY

Encrypt 圖例會依下列方式序列化：

1. 依代表其正規路徑的位元組序列進行文字描述。
2. 對於每個欄位，請依序附加上述指定的其中一個位元組值，以指出該欄位是否應加密。

加密內容長度

加密內容的長度。它會以 2 位元組數值表示，並解譯為 16 位元的無符號整數。長度上限為 65，535 個位元組。

加密內容

一組名稱/值對，其中包含任意、非秘密的額外已驗證資料。

啟用 ECDSA 數位簽章時，加密內容會包含金鑰值對 `{"aws-crypto-footer-ecdsa-key": Qtxt}`。 `Qtxt`代表Q根據 SEC 1 2.0 版壓縮的橢圓曲線點，然後是 base64 編碼。

加密的資料金鑰計數

加密資料金鑰的數量。它是 1 位元組值，解譯為 8 位元未簽署整數，指定加密資料金鑰的數量。每個記錄中的加密資料金鑰數量上限為 255。

加密的資料金鑰

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	位元組長度
Key Provider ID Length	2
Key Provider ID	變數. 等於前 2 個位元組中指定的值 (金鑰提供者 ID 長度)。
Key Provider Information Length	2
Key Provider Information	變數. 等於前 2 個位元組中指定的值 (金鑰提供者資訊長度)。
Encrypted Data Key Length	2
Encrypted Data Key	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。

當您使用 AWS KMS keyring 時，此值包含 的 Amazon Resource Name (ARN) AWS KMS key。

加密的資料金鑰長度

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。

加密的資料金鑰

加密資料金鑰。這是金鑰提供者加密的資料金鑰。

記錄承諾

不同的 256 位元雜湊型訊息驗證碼 (HMAC) 雜湊，使用遞交金鑰計算所有上述材料描述位元組。

AWS KMS 階層式 keyring 技術詳細資訊

AWS KMS 階層式 keyring 使用不適當的資料金鑰來加密每個欄位，並使用衍生自作用中分支金鑰的唯一包裝金鑰來加密每個資料金鑰。它使用計數器模式中的金鑰衍生，搭配 HMAC SHA-256 的虛擬隨機函數，以下列輸入衍生 32 位元組包裝金鑰。

- 16 位元組隨機鹽
- 作用中分支金鑰
- 金鑰提供者識別碼 "aws-kms-hierarchy" 的 UTF-8 編碼值

階層式 keyring 使用衍生的包裝金鑰，使用 AES-GCM-256 搭配 16 位元組身分驗證標籤和下列輸入來加密純文字資料金鑰的副本。

- 衍生的包裝金鑰會用作 AES-GCM 密碼金鑰
- 資料金鑰會用作 AES-GCM 訊息
- 12 位元組隨機初始化向量 (IV) 用作 AES-GCM IV

- 包含下列序列化值的其他已驗證資料 (AAD)。

Value	位元組長度	解譯為
"aws-kms-hierarchy"	17	UTF-8 編碼
分支金鑰識別符	變數	UTF-8 編碼
分支金鑰版本	16	UTF-8 編碼
加密內容	變數	UTF-8 編碼金鑰值對

AWS 資料庫加密 SDK 開發人員指南的文件歷史記錄

下表說明本文件的重大變更。除了下述各項主要變更外，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
新功能	新增 AWS KMS ECDH keyring 和 Raw ECDH keyring 的文件。	2024 年 6 月 17 日
一般可用性 (GA) 版本	介紹支援適用於 DynamoDB 的 .NET 用戶端加密程式庫。	2024 年 1 月 17 日
一般可用性 (GA) 版本	更新 DynamoDB Java 用戶端加密程式庫 3.x 版的 GA 版本文件。 **Warning** 不再支援在開發人員預覽版本期間建立的分支金鑰。	2023 年 7 月 24 日
DynamoDB 加密用戶端的更名	用戶端加密程式庫已重新命名為 AWS Database Encryption SDK。	2023 年 6 月 9 日
預覽版本	新增和更新 DynamoDB Java 用戶端加密程式庫 3.x 版的文件，其中包括新的結構化資料格式、改善的多租戶支援、無縫結構描述變更和可搜尋的加密支援。	2023 年 6 月 9 日
文件變更	將 AWS Key Managemen t Service 術語客戶主金鑰	2021 年 8 月 30 日

	(CMK) 取代為 AWS KMS key和 KMS 金鑰。	
新功能	新增對 AWS Key Management Service (AWS KMS) 多區域金鑰的支援。多區域金鑰是 AWS KMS 不同 中的金鑰 AWS 區域 ，可以互換使用，因為它們具有相同的金鑰 ID 和金鑰材料。	2021 年 6 月 8 日
新範例	新增了使用 DynamoDBMapper 的 Java 範例。	2018 年 9 月 6 日
Python 支援	除 Java 外另新增了對 Python 的支援。	2018 年 5 月 2 日
初始版本	本文件的初始版本。	2018 年 5 月 2 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。