

EF Core 3.0 中包含的中断性变更

2019/10/25 • [Edit Online](#)

以下 API 和行为更改有可能使现有应用程序在升级到 3.0.0 时中断。我们将仅影响数据库提供程序的更改记录在[提供程序更改](#)下。

总结

重大更改	影响
不再在客户端上计算 LINQ 查询	高
EF Core 3.0 面向 .NET Standard 2.1, 而不是 .NET Standard 2.0	高
EF Core 命令行工具 dotnet ef 不再是 .NET Core SDK 的一部分	高
DetectChanges 遵循存储生成的键值	高
FromSql、ExecuteSql 和 ExecuteSqlAsync 已重命名	高
查询类型与实体类型合并	高
Entity Framework Core 不再是 ASP.NET Core 共享框架的一部分	中等
默认情况下, 现在会立即发生级联删除	中等
单个查询中现在开始预先加载相关实体	中等
DeleteBehavior.Restrict 具有更简洁的语义	中等
从属类型关系的配置 API 已更改	中等
每个属性使用独立的内存中整数键生成	中等
无跟踪查询不再执行标识解析	中等
元数据 API 更改	中等
特定于提供程序的元数据 API 更改	中等
UseRowNumberForPaging 已删除	中等
FromSql 方法在与存储过程配合使用时, 无法进行组合	中等
只能在查询根上指定 FromSql 方法	低

重大更改	影响
在调试级别记录查询执行已还原	低
不再在实体实例上设置临时键值	低
与主体共享表的依赖实体现为可选项	低
与并发标记列共享表的所有实体都必须将其映射到属性	低
对于所有派生的类型而言, 从未映射的类型继承的属性现在会映射到一个列中	低
外键属性约定不再匹配与主体属性相同的名称	低
现在, 如果在 TransactionScope 完成前不再使用数据库连接, 则该连接会关闭	低
默认情况下使用支持字段	低
如果找到多个兼容的支持字段, 则引发	低
“仅字段”属性名应与字段名匹配	低
AddDbContext/AddDbContextPool 不再调用 AddLogging 和 AddMemoryCache	低
DbContext.Entry 现在执行本地 DetectChanges	低
默认情况下, 字符串和字节数组键不是客户端生成的	低
ILoggerFactory 现在是一个在一定范围内有效的服务	低
延迟加载代理不再假定导航属性已完全加载	低
默认情况下, 现在过度创建内部服务提供程序是一个错误	低
使用单个字符串调用 HasOne/HasMany 的新行为	低
多个异步方法的返回类型已从 Task 更改为 ValueTask	低
关系式: TypeMapping 注释现在只是 TypeMapping	低
派生类型上的 ToTable 会引发异常	低
EF Core 不再发送 pragma 来执行 SQLite FK	低
Microsoft.EntityFrameworkCore.Sqlite 现在依赖于 SQLitePCLRaw.bundle_e_sqlite3	低
GUID 值现在以文本形式存储在 SQLite 上	低
Char 值现在以文本形式存储在 SQLite 上	低

重大更改	影响
现在使用固定区域性的日历生成迁移 ID	低
已从 IDbContextOptionsExtension 中删除扩展信息/元数据	低
已重命名 LogQueryPossibleExceptionWithAggregateOperator	低
阐明 API 的外键约束名称	低
IRelationalDatabaseCreator.HasTables/HasTablesAsync 已公开	低
Microsoft.EntityFrameworkCore.Design 现在是 DevelopmentDependency 包	低
SQLitePCL.raw 已更新为版本 2.0.0	低
NetTopologySuite 已更新为版本 2.0.0	低
使用 Microsoft.Data.SqlClient 而不是 System.Data.SqlClient	低
必须配置多个不明确的自引用关系	低
DbFunction.Schema 为 null 或者空字符串将其配置为位于模型的默认架构中	低

不再在客户端上计算 LINQ 查询

[跟踪问题 #14935](#) 另请参阅[问题 #12795](#)

旧行为

在 3.0 之前，当 EF Core 无法将查询中的表达式转换为 SQL 或参数时，它会在客户端上自动计算表达式的值。默认情况下，客户端对潜在的昂贵表达式的计算仅触发警告。

新行为

从 3.0 开始，EF Core 仅允许在客户端上计算顶级投影中的表达式（查询中的最后一个 `Select()` 调用）。当查询的任何其他部分中的表达式无法转换为 SQL 或参数时，将引发异常。

为什么

自动的客户端查询计算允许执行许多查询，即使它们的重要组成部分无法转换。此行为可能导致意外且具有潜在破坏性的行为，这些行为可能仅在生产中变得明显。例如，`Where()` 调用中无法转换的条件可能导致表中的所有行从数据库服务器传输且筛选器应用于客户端。如果在开发中表中只包含几行，则不容易检测到这种情况，但是当应用程序转入生产环节时，由于表中可能包含数百万行，这种情况会非常严重。在开发过程中，客户端求值警告也很容易被忽视。

除此之外，自动客户端计算可能会导致问题，其中改进特定表达式的查询转换会导致版本之间发生意外中断性变更。

缓解措施

如果无法完全转换查询，则以可转换的形式重写查询，或使用 `AsEnumerable()`、`ToList()` 或类似信息将数据显式返回客户端，然后可以进一步使用 LINQ 到对象处理。

EF Core 3.0 面向 .NET Standard 2.1，而不是 .NET Standard 2.0

[跟踪问题 #15498](#)

旧行为

在 3.0 之前, EF Core 面向 .NET Standard 2.0, 并在支持 .NET Standard 2.0 的所有平台上运行, 包括 .NET Framework。

新行为

从 3.0 开始, EF Core 面向 .NET Standard 2.1, 并且在支持 .NET Standard 2.1 的所有平台上运行。这~~不包括~~ .NET Framework。

为什么

这是 .NET 技术中战略决策的一部分, 旨在将重点放在 .NET Core 和其他新式 .NET 平台, 例如 Xamarin。

缓解措施

请考虑迁移到新式 .NET 平台。如果无法做到这一点, 请继续使用 EF Core 2.1 或 EF Core 2.2, 这两者都支持 .NET Framework。

Entity Framework Core 不再是 ASP.NET Core 共享框架的一部分

[跟踪问题公告 #325](#)

旧行为

在 ASP.NET Core 3.0 之前, 当向 `Microsoft.AspNetCore.App` 或 `Microsoft.AspNetCore.All` 添加包引用时, 它将包括 EF Core 和一些 EF Core 数据提供程序(如 SQL Server 提供程序)。

新行为

从 3.0 开始, ASP.NET Core 共享框架不包括 EF Core 或任何 EF Core 数据提供程序。

为什么

在此更改之前, 获取 EF Core 需要不同的步骤, 具体取决于应用程序是否是面向 ASP.NET Core 和 SQL Server。此外, 升级 ASP.NET Core 会强制升级 EF Core 和 SQL Server 提供程序, 这并不总是可取的。

通过此更改, 通过所有提供程序、支持的 .NET 实现和应用程序类型获取 EF Core 的体验都是一致的。开发人员现在还可以准确控制何时升级 EF Core 和 EF Core 数据提供程序。

缓解措施

若要在 ASP.NET Core 3.0 应用程序或任何其他受支持的应用程序中使用 EF Core, 请显式添加对应用程序将使用的 EF Core 数据库提供程序的包引用。

EF Core 命令行工具 `dotnet ef` 不再是 .NET Core SDK 的一部分

[跟踪问题 #14016](#)

旧行为

.NET Core SDK 3.0 以前的版本包含 `dotnet ef` 工具, 可以随时从任何项目的命令行使用, 无需额外的步骤。

新行为

从 3.0 版开始, .NET SDK 不再包含 `dotnet ef` 工具, 因此, 在使用它之前, 必须将其明确安装为本地或全局工具。

为什么

此更改允许我们在 NuGet 上将 `dotnet ef` 作为常规 .NET CLI 工具分发和更新, 这与 EF Core 3.0 也始终作为 NuGet 包分发的事实一致。

缓解措施

为了能够管理迁移或构架 `DbContext`，请安装 `dotnet-ef` 作为全局工具：

```
$ dotnet tool install --global dotnet-ef
```

使用[工具清单文件](#)恢复声明为工具依赖项的项目依赖项时，还可以将其作为本地工具获取。

FromSql、ExecuteSql 和 ExecuteSqlAsync 已重命名

[跟踪问题 #10996](#)

旧行为

在 EF Core 3.0 之前，这些方法名称是重载的，它们使用普通字符串或应内插到 SQL 和参数中的字符串。

新行为

自 EF Core 3.0 起，可使用 `FromSqlRaw`、`ExecuteSqlRaw` 和 `ExecuteSqlRawAsync` 创建一个参数化的查询，其中参数是从查询字符串中单独传递的。例如：

```
context.Products.FromSqlRaw(
    "SELECT * FROM Products WHERE Name = {0}",
    product.Name);
```

使用 `FromSqlInterpolated`、`ExecuteSqlInterpolated` 和 `ExecuteSqlInterpolatedAsync` 创建一个参数化的查询，其中参数作为内插查询字符串的一部分进行传递。例如：

```
context.Products.FromSqlInterpolated(
    $"SELECT * FROM Products WHERE Name = {product.Name}");
```

请注意，上述两个查询都将生成 SQL 参数相同的同一参数化的 SQL。

为什么

此类方法重载使得在意图调用内插字符串方法时很容易意外调用原始字符串方法，反之亦然。这会导致查询中的本该参数化的结果没有参数化。

缓解措施

切换到使用新的方法名称。

FromSql 方法在与存储过程配合使用时，无法进行组合

[跟踪问题 #15392](#)

旧行为

在 EF Core 3.0 之前，`FromSql` 方法已尝试检测是否可对传入的 SQL 进行组合。当 SQL 像存储过程那样不可组合时，该方法进行客户端评估。以下查询在服务器上运行存储过程并在客户端执行 `FirstOrDefault`。

```
context.Products.FromSqlRaw("[dbo].[Ten Most Expensive Products]").FirstOrDefault();
```

新行为

从 EF Core 3.0 开始，EF Core 将不再尝试分析 SQL。因此，如果在 `FromSqlRaw/FromSqlInterpolated` 之后组合，则 EF Core 会通过引发子查询来组合 SQL。因此，如果将存储过程用于组合，则出现无效 SQL 语法的异常。

为什么

EF Core 3.0 不支持自动客户端评估，因为容易出错，如[此处](#)所述。

缓解措施

如果在 `FromSqlRaw/FromSqlInterpolated` 中使用存储过程，你了解无法对其进行组合，因此可以紧随 `FromSql` 方法调用添加 `AsEnumerable/AsAsyncEnumerable`，以避免在服务器端上进行任何组合。

```
context.Products.FromSqlRaw("[dbo].[Ten Most Expensive Products]").AsEnumerable().FirstOrDefault();
```

只能在查询根上指定 `FromSql` 方法

[跟踪问题 #15704](#)

旧行为

在 EF Core 3.0 之前，可以在查询中的任意位置指定 `FromSql` 方法。

新行为

从 EF Core 3.0 开始，只能在查询根上(即，直接在 `DbSet<>` 上)指定新的 `FromSqlRaw` 和 `FromSqlInterpolated` 方法(替换 `FromSql`)。尝试在其他任何位置指定这些方法将导致编译错误。

为什么

在除 `DbSet` 之外的任意位置指定 `FromSql` 没有附加含义或附加值，并且可能在某些情况下存在多义性。

缓解措施

应移动 `FromSql` 调用以使其直接位于它们所应用的 `DbSet` 上。

无跟踪查询不再执行标识解析

[跟踪问题 #13518](#)

旧行为

在 EF Core 3.0 之前，将对具有给定类型和 ID 的实体的每个匹配项使用同一个实体实例。这与跟踪查询的行为匹配。例如，以下查询：

```
var results = context.Products.Include(e => e.Category).AsNoTracking().ToList();
```

会为与给定类别关联的每个 `Product` 返回相同的 `Category` 实例。

新行为

从 EF Core 3.0 开始，当在返回的图中的不同位置遇到具有给定类型和 ID 的实体时，将创建不同的实体实例。例如，上面的查询现在将为每个 `Product` 返回新的 `Category` 实例，即使两个产品与同一类别关联。

为什么

标识解析(即确定实体与先前遇到的实体具有相同的类型和 ID)会增加额外的性能和内存开销。这通常会运行计数器，原因是最初使用无跟踪查询。此外，尽管标识解析有时会很有用，但如果要对实体进行序列化并将其发送到客户端(这对于无跟踪查询很常见)，则不需要这样做。

缓解措施

如果需要标识解析，请使用跟踪查询。

在调试级别记录查询执行已还原

[跟踪问题 #14523](#)

我们之所以还原此更改是因为，EF Core 3.0 中的新配置允许应用程序指定任何事件的日志级别。例如，若要将 SQL 日志记录切换为 `Debug`，请在 `OnConfiguring` 或 `AddDbContext` 中显式配置级别：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseSqlServer(connectionString)
        .ConfigureWarnings(c => c.Log((RelationalEventId.CommandExecuting, LogLevel.Debug)));
```

不再在实体实例上设置临时键值

[跟踪问题 #12378](#)

旧行为

在 EF Core 3.0 之前，临时值已分配给所有键属性，这些属性稍后将具有数据库生成的实际值。通常这些临时值是较大负数。

新行为

从 3.0 开始，EF Core 将临时键值存储为实体跟踪信息的一部分，并保持键属性本身不变。

为什么

此更改是为了防止当之前由某个 `DbContext` 实例跟踪的实体移动到另一个 `DbContext` 实例时，临时键值错误地变成永久值。

缓解措施

如果主键是存储生成的并且属于 `Added` 状态的实体，则将主键值分配到外键以在实体之间形成关联的应用可能会依赖于旧行为。可通过以下方式避免：

- 不使用存储生成的密钥。
- 设置导航属性以形成关系，而不是设置外键值。
- 从实体的跟踪信息中获取实际的临时键值。例如，即使 `blog.Id` 本身尚未设置，`context.Entry(blog).Property(e => e.Id).CurrentValue` 也将返回临时值。

DetectChanges 遵循存储生成的键值

[跟踪问题 #14616](#)

旧行为

在 EF Core 3.0 之前，`DetectChanges` 找到的未跟踪实体将在 `Added` 状态中被跟踪，并在调用 `SaveChanges` 时作为新行插入。

新行为

从 EF Core 3.0 开始，如果实体使用生成的键值并设置了某个键值，则将在 `Modified` 状态下跟踪实体。这意味着假定存在实体的行，并且在调用 `SaveChanges` 时将更新该行。如果未设置键值，或者实体类型未使用生成的键，则新实体仍将像先前版本一样被作为 `Added` 跟踪。

为什么

进行此更改是为了在使用存储生成的键时更轻松、更一致地使用断开连接的实体图。

缓解措施

如果将实体类型配置为使用生成的键，但为新实例显式设置了键值，则此更改可能会中断应用程序。解决方案是显式配置键属性，使其不使用生成的值。例如，使用 Fluent API：

```
modelBuilder
    .Entity<Blog>()
    .Property(e => e.Id)
    .ValueGeneratedNever();
```

或使用数据注释：

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public string Id { get; set; }
```

默认情况下，现在会立即发生级联删除

[跟踪问题 #10114](#)

旧行为

在 3.0 之前，直到调用 `SaveChanges` 时，EF Core 才会应用级联操作（删除所需主体时或者在切断与所需主体的关系时删除依赖实体）。

新行为

从 3.0 开始，一旦检测到触发条件，EF Core 就会应用级联操作。例如，调用 `context.Remove()` 来删除主体实体将导致所有跟踪的相关必需依赖项也立即设置为 `Deleted`。

为什么

此更改是为了改善数据绑定和审核方案的体验，在相关体验中，需要了解在调用 `SaveChanges` 之前会删除哪些实体。

缓解措施

可以通过 `context.ChangedTracker` 上的设置还原以前的行为。例如：

```
context.ChangeTracker.CascadeDeleteTiming = CascadeTiming.OnSaveChanges;
context.ChangeTracker.DeleteOrphansTiming = CascadeTiming.OnSaveChanges;
```

单个查询中现在开始预先加载相关实体

[跟踪问题 #18022](#)

旧行为

在 3.0 之前，通过 `Include` 运算符预先加载集合导航会导致在关系数据库上生成多个查询，每个相关实体类型对应一个查询。

新行为

从 3.0 开始，EF Core 会在关系数据库上使用 JOIN 生成单个查询。

为什么

以发出多个查询的方式实现单个 LINQ 查询会导致出现许多问题，包括由于需要执行多次数据库往返而引起的性能不佳问题，以及每个查询都可能会观察到不同的数据库状态的数据不一致问题。

缓解措施

尽管从技术上讲，这不是一项中断性变更，但当单个查询在集合导航中包含大量 `Include` 运算符时，这可能对应用程序性能产生相当大的影响。[请参阅此评论](#)，了解详细信息，以及如何以更有效的方式重写查询。

**

DeleteBehavior.Restrict 具有更简洁的语义

[跟踪问题 #12661](#)

旧行为

3.0 之前, `DeleteBehavior.Restrict` 使用 `Restrict` 语义在数据库中创建外键, 但也以不明显的方式更改了内部修复。

新行为

从 3.0 开始, `DeleteBehavior.Restrict` 确保使用 `Restrict` 语义创建外键--即无级联; 在发生约束冲突时触发 - 但不会同时影响 EF 内部修复。

为什么

进行此更改是为了改进以直观方式使用 `DeleteBehavior` 的体验, 且不产生意外副作用。

缓解措施

可使用 `DeleteBehavior.ClientNoAction` 还原以前的行为。

查询类型与实体类型合并

[跟踪问题 #14194](#)

旧行为

在 EF Core 3.0 之前, [查询类型](#) 是一种查询未以结构化方式定义主键的数据的方法。也就是说, 查询类型用于映射没有键的实体类型(更可能来自视图, 但也可能来自表), 而当有可用的键时则使用常规实体类型(更可能来自表, 但也可能来自视图)。

新行为

现在, 查询类型只是一个没有主键的实体类型。无键实体类型与先前版本中的查询类型具有相同的功能。

为什么

这样做是为了减少对查询类型用途的混淆。具体来说, 它们是无键实体类型, 因此本质上是只读的, 但是不应该仅仅因为实体类型需要是只读的就使用它们。同样, 它们通常映射到视图, 但这只是因为视图通常不定义键。

缓解措施

API 的以下部分现已过时:

- `ModelBuilder.Query<>()` - 需要调用 `ModelBuilder.Entity<>().HasNoKey()` 来将实体类型标记为没有键。这仍然不会按约定配置, 以避免在需要主键但是与约定不匹配时发生配置错误。
- `DbQuery<>` - 应使用 `DbSet<>`。
- `DbContext.Query<>()` - 应使用 `DbContext.Set<>()`。

从属类型关系的配置 API 已更改

[跟踪问题 #12444](#) [跟踪问题 #9148](#) [跟踪问题 #14153](#)

旧行为

EF Core 3.0 之前, 会在 `OwnsOne` 或 `OwnsMany` 调用之后直接执行所拥有关系的配置。

新行为

从 EF Core 3.0 开始, Fluent API 会使用 `WithOwner()` 为所有者配置导航属性。例如:

```
modelBuilder.Entity<Order>.OwnsOne(e => e.Details).WithOwner(e => e.Order);
```

与所有者之间关系相关的配置现会在 `WithOwner()` 之后关联起来, 就像配置其他关系一样。但从属类型本身的配置仍会在 `OwnsOne()/OwnsMany()` 之后关联。例如:

```
modelBuilder.Entity<Order>.OwnsOne(e => e.Details, eb =>
{
    eb.WithOwner()
        .HasForeignKey(e => e.AlternateId)
        .HasConstraintName("FK_OrderDetails");

    eb.ToTable("OrderDetails");
    eb.HasKey(e => e.AlternateId);
    eb.HasIndex(e => e.Id);

    eb.HasOne(e => e.Customer).WithOne();

    eb.HasData(
        new OrderDetails
        {
            AlternateId = 1,
            Id = -1
        });
});
```

此外, 使用固有类型目标调用 `Entity().HasOne()` 或 `Set()` 现将引发异常。

为什么

此更改是为了更清晰的区分从属类型本身的配置和与从属类型的_关系_的配置。这反过来消除了诸如 `HasForeignKey` 之类的方法的模糊性和混淆。

缓解措施

更改从属类型关系的配置以使用新的 API 曲面, 如上例所示。

与主体共享表的依赖实体现为可选项

[跟踪问题 #9005](#)

旧行为

考虑下列模型:

```
public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public OrderDetails Details { get; set; }
}

public class OrderDetails
{
    public int Id { get; set; }
    public string ShippingAddress { get; set; }
}
```

在 EF Core 3.0 之前, 如果 `OrderDetails` 由 `Order` 拥有且显式映射到同一张表, 则在添加新的 `Order` 时, 始终需要 `OrderDetails` 实例。

新行为

自 3.0 起, EF Core 允许添加 `Order` 而不添加 `OrderDetails`, 并将主键之外的所有 `OrderDetails` 属性映射到不为 null 的列中。查询时, 如果其任意所需属性均没有值, 或者它的主键之外没有任何必需属性且所有属性均为 `null`

，则 EF Core 会将 `OrderDetails` 设置为 `null`。

缓解措施

如果你的模型具有依赖于所有可选列的表共享，但指向该共享的导航不应为 `null`，则应修改应用程序，使其处理导航为 `null` 的情况。如果此方法不可行，则应向实体类型添加一个必需属性，或者至少要有一个属性分配有非 `null` 值。

与并发标记列共享表的所有实体都必须将其映射到属性

[跟踪问题 #14154](#)

旧行为

考虑下列模型：

```
public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public byte[] Version { get; set; }
    public OrderDetails Details { get; set; }
}

public class OrderDetails
{
    public int Id { get; set; }
    public string ShippingAddress { get; set; }
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Order>()
        .Property(o => o.Version).IsRowVersion().HasColumnName("Version");
}
```

在 EF Core 3.0 之前，如果 `OrderDetails` 由 `Order` 拥有且显式映射到同一张表，则只更新 `OrderDetails` 时，将不更新客户端上的 `Version` 值且下次更新将失败。

新行为

自 3.0 起，如果新的 `Version` 值拥有 `OrderDetails` 则 EF Core 会将该值传播给 `Order`。否则，会在模型验证期间引发异常。

为什么

进行此更改的目的是避免在仅更新映射到同一张表的其中一个实体时使用过时的并发标记值。

缓解措施

共享表的所有实体都必须包含一个映射到并发标记列的属性。可在影子状态中创建一个：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<OrderDetails>()
        .Property<byte[]>("Version").IsRowVersion().HasColumnName("Version");
}
```

对于所有派生的类型而言，从未映射的类型继承的属性现在会映射到一个列中

[跟踪问题 #13998](#)

旧行为

考虑下列模型：

```
public abstract class EntityBase
{
    public int Id { get; set; }
}

public abstract class OrderBase : EntityBase
{
    public int ShippingAddress { get; set; }
}

public class BulkOrder : OrderBase
{
}

public class Order : OrderBase
{
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<OrderBase>();
    modelBuilder.Entity<EntityBase>();
    modelBuilder.Entity<BulkOrder>();
    modelBuilder.Entity<Order>();
}
```

在 EF Core 3.0 之前，`ShippingAddress` 属性会为 `BulkOrder` 和 `Order` 默认映射到单独的列中。

新行为

自 3.0 起, EF Core 只会为 `ShippingAddress` 创建一个列。

为什么

旧行为不是预期行为。

缓解措施

属性仍可显式映射到所派生的类型上的单独的列中：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<OrderBase>();
    modelBuilder.Entity<EntityBase>();
    modelBuilder.Entity<BulkOrder>()
        .Property(o => o.ShippingAddress).HasColumnName("BulkShippingAddress");
    modelBuilder.Entity<Order>()
        .Property(o => o.ShippingAddress).HasColumnName("ShippingAddress");
}
```

外键属性约定不再匹配与主体属性相同的名称

[跟踪问题 #13274](#)

旧行为

考虑下列模型：

```
public class Customer
{
    public int CustomerId { get; set; }
    public ICollection<Order> Orders { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
}
```

在 EF Core 3.0 之前, `CustomerId` 属性将按约定用于外键。但是, 如果 `Order` 是从属类型, 那么这也会使 `CustomerId` 成为主键, 这通常不是预期结果。

新行为

自 3.0 起, 如果外键的主体属性名称相同, EF Core 不会尝试通过转换来为外键使用属性。与主体属性名称关联的主体类型名称和与主体属性名称模式关联的导航名称仍然相匹配。例如:

```
public class Customer
{
    public int Id { get; set; }
    public ICollection<Order> Orders { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
}
```

```
public class Customer
{
    public int Id { get; set; }
    public ICollection<Order> Orders { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public int BuyerId { get; set; }
    public Customer Buyer { get; set; }
}
```

为什么

此更改是为了避免错误地在从属类型上定义主键属性。

缓解措施

如果该属性将成为外键, 即为主键的一部分, 则显式配置它。

现在, 如果在 `TransactionScope` 完成前不再使用数据库连接, 则该连接会关闭

[跟踪问题 #14218](#)

旧行为

在 EF Core 3.0 之前, 如果上下文打开了 `TransactionScope` 中的连接, 则该连接在 `TransactionScope` 处于活动期间仍保持打开状态。

```
using (new TransactionScope())
{
    using (AdventureWorks context = new AdventureWorks())
    {
        context.ProductCategories.Add(new ProductCategory());
        context.SaveChanges();

        // Old behavior: Connection is still open at this point

        var categories = context.ProductCategories().ToList();
    }
}
```

新行为

自 3.0 起, 一旦不再使用连接, EF Core 就会将其关闭。

为什么

此更改让你能够在同一 `TransactionScope` 中使用多个上下文。新行为也与 EF6 一致。

缓解措施

如果连接需要保持打开状态, 则显式调用 `OpenConnection()` 将确保 EF Core 不永久关闭此连接:

```
using (new TransactionScope())
{
    using (AdventureWorks context = new AdventureWorks())
    {
        context.Database.OpenConnection();
        context.ProductCategories.Add(new ProductCategory());
        context.SaveChanges();

        var categories = context.ProductCategories().ToList();
        context.Database.CloseConnection();
    }
}
```

每个属性使用独立的内存中整数键生成

[跟踪问题 #6872](#)

旧行为

在 EF Core 3.0 之前, 一个共享值生成器用于所有内存中的整数键属性。

新行为

从 EF Core 3.0 开始, 每个整数键属性在使用内存数据库时都会获取其自己的值生成器。此外, 如果删除了数据库, 则会为所有表重置键生成。

为什么

此更改的目的是使内存中的键生成更接近于实际的数据库键生成, 并改进在使用内存中的数据库时隔离测试的功能。

缓解措施

这可能会中断依赖于特定内存中键值的应用程序。请考虑不依赖于特定键值, 或者进行更新以匹配新行为。

默认情况下使用支持字段

[跟踪问题 #12430](#)

旧行为

在 3.0 之前，即使已知属性的支持字段，EF Core 仍将默认使用属性 getter 和 setter 方法读取和写入属性值。例外情况是查询执行，如果已知，将直接设置支持字段。

新行为

从 EF Core 3.0 开始，如果已知属性的支持字段，EF Core 将始终使用支持字段读取和写入该属性。如果应用程序依赖于编码到 getter 或 setter 方法中的其他行为，则可能导致应用程序中断。

为什么

此更改是为了防止 EF Core 在执行涉及实体的数据库操作时默认错误地触发业务逻辑。

缓解措施

可以通过在 `ModelBuilder` 上配置属性访问模式来恢复 3.0 之前的行为。例如：

```
modelBuilder.UsePropertyAccessMode(PropertyAccessMode.PreferFieldDuringConstruction);
```

如果找到多个兼容的支持字段，则引发

[跟踪问题 #12523](#)

旧行为

在 EF Core 3.0 之前，如果多个字段与查找属性的后备字段的规则匹配，则将基于某种优先顺序选择一个字段。这可能导致在不明确的情况下使用错误字段。

新行为

从 EF Core 3.0 开始，如果多个字段与同一属性匹配，则引发异常。

为什么

此更改是为了避免在只有一个字段是正确的情况下无提示地使用另一个字段。

缓解措施

具有不明确的支持字段的属性必须具有显式指定的字段。例如，使用 Fluent API：

```
modelBuilder
    .Entity<Blog>()
    .Property(e => e.Id)
    .HasField("_id");
```

“仅字段”属性名应与字段名匹配

旧行为

在 EF Core 3.0 之前，可通过字符串值指定属性，如果在 .NET 类型上找不到具有该名称的属性，则 EF Core 将尝试使用约定规则将其与字段匹配。

```
private class Blog
{
    private int _id;
    public string Name { get; set; }
}
```

```
modelBuilder
    .Entity<Blog>()
    .Property("Id");
```

新行为

从 EF Core 3.0 开始,“仅字段”属性必须与字段名完全匹配。

```
modelBuilder
    .Entity<Blog>()
    .Property("_id");
```

为什么

进行此更改是为了避免对两个名称相似的属性使用相同的字段,这也使得仅字段属性的匹配规则与映射到 CLR 属性的属性相同。

缓解措施

仅字段属性名必须与它们映射到的字段名相同。在 3.0 版之后的 EF Core 未来版本中,我们计划重新启用显式配置与属性名称不同的字段名称(请参阅问题 [15307](#)):

```
modelBuilder
    .Entity<Blog>()
    .Property("Id")
    .HasField("_id");
```

AddDbContext/AddDbContextPool 不再调用 AddLogging 和 AddMemoryCache

[跟踪问题 #14756](#)

旧行为

在 EF Core 3.0 之前,调用 `AddDbContext` 或 `AddDbContextPool` 的操作也可以通过调用 [AddLogging](#) 和 [AddMemoryCache](#) 在 DI 中注册日志记录和内存缓存服务。

新行为

从 EF Core 3.0 开始, `AddDbContext` 和 `AddDbContextPool` 将无法再在依赖注入 (DI) 中注册这些服务。

为什么

EF Core 3.0 不要求这些服务位于应用程序的 DI 容器中。但是,如果 `ILoggerFactory` 在应用程序的 DI 容器中注册,它仍然会由 EF Core 使用。

缓解措施

如果应用程序需要这些服务,则使用 [AddLogging](#) 或 [AddMemoryCache](#) 将它们显式注册到 DI 容器中。

DbContext.Entry 现在执行本地 DetectChanges

[跟踪问题 #13552](#)

旧行为

在 EF Core 3.0 之前,调用 `DbContext.Entry` 将导致检测到所有被跟踪实体的更改。这确保了 `EntityEntry` 中暴露的状态是最新的。

新行为

从 EF Core 3.0 开始,调用 `DbContext.Entry` 现在只会尝试检测给定实体和与之相关的任何跟踪主体实体的更改。

这意味着可能无法通过调用此方法检测到其他位置的更改，这可能会影响应用程序状态。

请注意，如果 `ChangeTracker.AutoDetectChangesEnabled` 设置为 `false`，则即使是本地更改检测也将被禁用。

导致更改检测的其他方法（例如 `ChangeTracker.Entries` 和 `SaveChanges`）仍然会导致所有被跟踪实体的完整 `DetectChanges`。

为什么

此更改是为了提高使用 `context.Entry` 的默认性能。

缓解措施

在调用 `Entry` 之前显式调用 `ChangeTracker.DetectChanges()` 以确保 3.0 之前的行为。

默认情况下，字符串和字节数组键不是客户端生成的

[跟踪问题 #14617](#)

旧行为

在 EF Core 3.0 之前，可以使用 `string` 和 `byte[]` 键属性，而不需要显式地设置非 null 值。在这种情况下，键值将在客户端上生成为 GUID，并序列化为 `byte[]` 的字节。

新行为

从 EF Core 3.0 开始，将引发异常，指示未设置任何键值。

为什么

之所以进行此更改是因为客户端生成的 `string` / `byte[]` 值通常没有用，并且默认行为使得很难以通用方式推断生成的键值。

缓解措施

如果没有设置其他非 null 值，则可以通过显式指定键属性应使用生成的值来获得 3.0 之前的行为。例如，使用 Fluent API：

```
modelBuilder
    .Entity<Blog>()
    .Property(e => e.Id)
    .ValueGeneratedOnAdd();
```

或使用数据注释：

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public string Id { get; set; }
```

ILoggerFactory 现在是一个在一定范围内有效的服务

[跟踪问题 #14698](#)

旧行为

在 EF Core 3.0 之前，`ILoggerFactory` 被注册为单一实例服务。

新行为

从 EF Core 3.0 开始，`ILoggerFactory` 现已注册为作用域。

为什么

此更改是为了允许记录器与 `DbContext` 实例关联，从而启用其他功能并删除一些反常行为，例如内部服务提供商爆炸式增长的情况。

缓解措施

除非在 EF Core 内部服务提供商上注册和使用自定义服务，否则此更改不应影响应用程序代码。这并不常见。在这些情况下，大多数事情仍然有效，但是需要更改依赖于 `ILoggerFactory` 的任何单一实例服务以便以不同的方式获取 `ILoggerFactory`。

如果遇到此类情况，请在 [EF Core GitHub 问题跟踪程序](#) 上提交一个问题，让我们知道你是如何使用 `ILoggerFactory` 的，以便我们更好地理解今后如何避免这种情况再次发生。

延迟加载代理不再假定导航属性已完全加载

[跟踪问题 #12780](#)

旧行为

在 EF Core 3.0 之前，一旦 `DbContext` 被处置，就无法知道从该上下文获得的实体上的给定导航属性是否已完全加载。相反，如果导航有一个非 null 值，代理将假定加载一个引用导航，如果导航非空，则假定加载集合导航。在这些情况下，尝试延迟加载将是无效的。

新行为

从 EF Core 3.0 开始，代理会跟踪是否加载了导航属性。这意味着如果尝试访问在释放了上下文之后加载的导航属性，其结果始终是无操作，即使加载的导航为空或为 null。相反，即使导航属性是非空集合，尝试访问未加载的导航属性也会引发异常。如果出现这种情况，则表示应用程序代码在无效时间尝试使用延迟加载，应将应用程序更改为不执行此操作。

为什么

此更改是为了在尝试对已释放的 `DbContext` 实例进行延迟加载时使行为保持一致和正确。

缓解措施

更新应用程序代码，以避免尝试对已释放的上下文进行延迟加载，或者将其配置为无操作，如异常消息中所述。

默认情况下，现在过度创建内部服务提供商是一个错误

[跟踪问题 #10236](#)

旧行为

在 EF Core 3.0 之前，对于创建了大量内部服务提供商的应用程序，将会记录一个警告。

新行为

从 EF Core 3.0 开始，现在会考虑此警告，并引发错误和异常。

为什么

此更改是为了通过更明显地暴露这个病态案例来驱动生成更好的应用程序代码。

缓解措施

遇到此错误时，最合适的操作是了解根本原因并停止创建如此多的内部服务提供商。但是，可以通过 `DbContextOptionsBuilder` 上的配置将错误转换回警告(或忽略)。例如：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .ConfigureWarnings(w => w.Log(CoreEventId.ManyServiceProvidersCreatedWarning));
}
```

使用单个字符串调用 HasOne/HasMany 的新行为

[跟踪问题 #9171](#)

旧行为

在 EF Core 3.0 之前, 对通过单个字符串调用 `HasOne` 或 `HasMany` 的代码的解释令人困惑。例如:

```
modelBuilder.Entity<Samurai>().HasOne("Entrance").WithOne();
```

该代码看起来是使用 `Entrance` 导航属性(可能是私有属性)将 `Samurai` 与某些其他实体类型关联起来的。

实际上, 此代码试图创建与某个名为 `Entrance` 的实体类型的关系, 该实体类型没有导航属性。

新行为

从 EF Core 3.0 开始, 上面的代码现执行它以前应执行的操作。

为什么

这一旧行为令人非常困惑, 尤其是在读取配置代码和查找错误时。

缓解措施

这只会中断使用类型名称字符串显式配置关系而无需显式指定导航属性的应用程序。这并不常见。以前的行为可以通过显式传递导航属性名称的 `null` 获得。例如:

```
modelBuilder.Entity<Samurai>().HasOne("Some.Entity.Type.Name", null).WithOne();
```

多个异步方法的返回类型已从 Task 更改为 ValueTask

[跟踪问题 #15184](#)

旧行为

以下异步方法之前返回的是 `Task<T>` :

- `DbContext.FindAsync()`
- `DbSet.FindAsync()`
- `DbContext.AddAsync()`
- `DbSet.AddAsync()`
- `ValueGenerator.NextValueAsync()` (及派生类)

新行为

上述方法现返回一个 `ValueTask<T>`, 其中 `T` 与前述相同。

为什么

此更改会减少在调用这些方法时发生的堆分配数量, 从而提高整体性能。

缓解措施

仅需重新编译只等待上述 API 的应用程序 - 无需更改源。更复杂的用法 (例如, 将返回的 `Task` 传递到 `Task.WhenAny()`) 通常需要通过返回的 `ValueTask<T>` 调用 `AsTask()` 来将其转换为 `Task<T>`。请注意, 这会抵消更改所带来的分配数减少优势。

关系式: `TypeMapping` 注释现在只是 `TypeMapping`

[跟踪问题 #9913](#)

旧行为

类型映射注释的注释名称是“`Relational:TypeMapping`”。

新行为

类型映射注释的注释名称现在是“`TypeMapping`”。

为什么

类型映射现在不仅用于关系数据库提供程序。

缓解措施

这只会中断直接作为注释访问类型映射的应用程序, 这不常见。最合适的修复操作是使用 API 曲面来访问类型映射, 而不是直接使用注释。

派生类型上的 `ToTable` 会引发异常

[跟踪问题 #11811](#)

旧行为

在 EF Core 3.0 之前, 将忽略调用派生类型的 `ToTable()`, 因为只有继承映射策略是 TPH, 这是无效的。

新行为

从 EF Core 3.0 开始, 同时为在以后的版本中添加 TPT 和 TPC 支持做准备, 调用派生类型的 `ToTable()` 现在将引发异常, 以避免将来发生意外的映射更改。

为什么

目前, 将派生类型映射到不同的表是无效的。这种改变避免了在将来当它变为有效时被中断。

缓解措施

删除将派生类型映射到其他表的任何尝试。

用 `HasIndex` 替换 `ForSqlServerHasIndex`

[跟踪问题 #12366](#)

旧行为

在 EF Core 3.0 之前, `ForSqlServerHasIndex().ForSqlServerInclude()` 提供了一种配置与 `INCLUDE` 一起使用的列的方法。

新行为

从 EF Core 3.0 开始, 现在支持在关系级别上对索引使用 `Include`。请使用 `HasIndex().ForSqlServerInclude()`。

为什么

此更改是为了将用于索引的 API 与 `Include` 合并到一个位置, 以供所有数据库提供程序使用。

缓解措施

使用新的 API, 如上所示。

元数据 API 更改

[跟踪问题 #214](#)

新行为

以下属性已转换为扩展方法：

- `IEntityType.QueryFilter` -> `GetQueryFilter()`
- `IEntityType.DefiningQuery` -> `GetDefiningQuery()`
- `IProperty.IsShadowProperty` -> `IsShadowProperty()`
- `IProperty.BeforeSaveBehavior` -> `GetBeforeSaveBehavior()`
- `IProperty.AfterSaveBehavior` -> `GetAfterSaveBehavior()`

为什么

此更改简化了上述接口的实现。

缓解措施

使用新的扩展方法。

特定于提供程序的元数据 API 更改

[跟踪问题 #214](#)

新行为

将展开特定于提供程序的扩展方法：

- `IProperty.Relational().ColumnName` -> `IProperty.GetColumnName()`
- `IEntityType.SqlServer().IsMemoryOptimized` -> `IEntityType.IsMemoryOptimized()`
- `PropertyBuilder.UseSqlServerIdentityColumn()` -> `PropertyBuilder.UseIdentityColumn()`

为什么

此更改简化了上述扩展方法的实现。

缓解措施

使用新的扩展方法。

EF Core 不再发送 pragma 来执行 SQLite FK

[跟踪问题 #12151](#)

旧行为

在 EF Core 3.0 之前，当打开与 SQLite 的连接时，EF Core 会发送 `PRAGMA foreign_keys = 1`。

新行为

从 EF Core 3.0 开始，当打开到 SQLite 的连接时，EF Core 不再发送 `PRAGMA foreign_keys = 1`。

为什么

之所以进行此更改，是因为 EF Core 默认使用 `SQLitePCLRaw.bundle_e_sqlite3`，这意味着 FK 强制执行操作在默认情况下是打开的，并且不需要在每次打开连接时显式启用。

缓解措施

默认情况下，`SQLitePCLRaw.bundle_e_sqlite3` 中启用了默认用于 EF Core 的外键。对于其他情况，可以通过在连接字符串中指定 `Foreign Keys=True` 来启用外键。

Microsoft.EntityFrameworkCore.Sqlite 现在依赖于 SQLitePCLRaw.bundle_e_sqlite3

旧行为

在 EF Core 3.0 之前, EF Core 使用 `SQLitePCLRaw.bundle_green`。

新行为

从 EF Core 3.0 开始, EF Core 使用 `SQLitePCLRaw.bundle_e_sqlite3`。

为什么

此更改是为了使 iOS 上使用的 SQLite 版本与其他平台一致。

缓解措施

若要在 iOS 上使用本机 SQLite 版本, 请配置 `Microsoft.Data.Sqlite` 以使用其他 `SQLitePCLRaw` 捆绑包。

GUID 值现在以文本形式存储在 SQLite 上

[跟踪问题 #15078](#)

旧行为

GUID 值之前以 BLOB 值形式存储在 SQLite 上。

新行为

Guid 值现在以文本形式存储。

为什么

GUID 的二进制格式不会进行标准化。以文本形式存储值使数据库与其他技术更兼容。

缓解措施

现在通过执行如下的 SQL, 可以将现有数据库转成新的格式。

```
UPDATE MyTable
SET GuidColumn = hex(substr(GuidColumn, 4, 1)) ||
                hex(substr(GuidColumn, 3, 1)) ||
                hex(substr(GuidColumn, 2, 1)) ||
                hex(substr(GuidColumn, 1, 1)) || '-' ||
                hex(substr(GuidColumn, 6, 1)) ||
                hex(substr(GuidColumn, 5, 1)) || '-' ||
                hex(substr(GuidColumn, 8, 1)) ||
                hex(substr(GuidColumn, 7, 1)) || '-' ||
                hex(substr(GuidColumn, 9, 2)) || '-' ||
                hex(substr(GuidColumn, 11, 6))
WHERE typeof(GuidColumn) == 'blob';
```

在 EF Core 中, 还可以通过为这些属性配置值转换器, 继续使用以前的行为模式。

```
modelBuilder
    .Entity<MyEntity>()
    .Property(e => e.GuidProperty)
    .HasConversion(
        g => g.ToByteArray(),
        b => new Guid(b));
```

Microsoft.Data.Sqlite 仍然能够从“BLOB”和“文本”列读取 GUID 值;但是, 由于参数和常量的默认格式已更改, 可能需要针对涉及 GUID 的大多数情况采取措施。

Char 值现在以文本形式存储在 SQLite 上

跟踪问题 #15020

旧行为

Char 值之前以整数值形式存储在 SQLite 上。例如, A 的 char 值存储为整数值 65。

新行为

Char 值现在以文本形式存储。

为什么

以文本形式存储值显得更加自然, 并且使数据库与其他技术更兼容。

缓解措施

现在通过执行如下的 SQL, 可以将现有数据库转成新的格式。

```
UPDATE MyTable
SET CharColumn = char(CharColumn)
WHERE typeof(CharColumn) = 'integer';
```

在 EF Core 中, 还可以通过为这些属性配置值转换器, 继续使用以前的行为模式。

```
modelBuilder
    .Entity<MyEntity>()
    .Property(e => e.CharProperty)
    .HasConversion(
        c => (long)c,
        i => (char)i);
```

Microsoft.Data.Sqlite 也仍然能够读取整数列和文本列的字符值, 因此某些情况可能不需要任何操作。

现在使用固定区域性的日历生成迁移 ID

跟踪问题 #12978

旧行为

以前使用当前区域性的日历无意生成迁移 ID。

新行为

现在始终使用固定区域性的日历(公历)生成迁移 ID。

为什么

更新数据库或解决合并冲突时, 迁移的顺序非常重要。使用固定日历可以避免因团队成员采用不同系统日历而产生的顺序问题。

缓解措施

如果有人使用年份时间大于公历日历的非公历日历(如泰国佛历), 则会受到影响。现有迁移 ID 需要进行更新, 以便新迁移排在现有迁移之后。

在迁移的设计者文件的 Migration 属性中可以找到迁移 ID。

```
[DbContext(typeof(MyDbContext))]  
-[Migration("25620318122820_MyMigration")]  
+[Migration("20190318122820_MyMigration")]  
partial class MyMigration  
{
```

迁移历史记录表还需要更新。

```
UPDATE __EFMigrationsHistory  
SET MigrationId = CONCAT(LEFT(MigrationId, 4) - 543, SUBSTRING(MigrationId, 4, 150))
```

UseRowNumberForPaging 已删除

[跟踪问题 #16400](#)

旧行为

在 EF Core 3.0 之前，`UseRowNumberForPaging` 可用于生成与 SQL Server 2008 兼容的分页 SQL。

新行为

从 EF Core 3.0 开始，EF 将仅生成仅与更高的 SQL Server 版本兼容的分页 SQL。

为什么

我们正在进行此更改，因为 [SQL Server 2008 不再是受支持的产品](#)，并且更新此功能以使用在 EF Core 3.0 中做出的查询更改是一项重要工作。

缓解措施

建议更新到更高的 SQL Server 版本，或者使用更高的兼容性级别，以便支持生成的 SQL。这就是说，如果无法执行此操作，请在[跟踪问题中](#)做出详细注释。我们可能会根据反馈重新考虑这个决定。

已从 IDbContextOptionsExtension 中删除扩展信息/元数据

[跟踪问题 #16119](#)

旧行为

`IDbContextOptionsExtension` 包含用于提供扩展元数据的方法。

新行为

这些方法已迁移到从新 `IDbContextOptionsExtension.Info` 属性返回的新 `DbContextOptionsExtensionInfo` 抽象基类。

为什么

在从版本 2.0 迁移到版本 3.0 的过程中，我们需要多次添加或更改这些方法。将它们拆分成新抽象基类可以更轻松地进行此类更改，而不会破坏现有扩展。

缓解措施

将扩展更新为采用新模式。例如，许多用于 EF Core 源代码中不同种类扩展的 `IDbContextOptionsExtension` 实现。

已重命名 LogQueryPossibleExceptionWithAggregateOperator

[跟踪问题 #10985](#)

更改

`RelationalEventId.LogQueryPossibleExceptionWithAggregateOperator` 已重命名为 `RelationalEventId.LogQueryPossibleExceptionWithAggregateOperatorWarning`。

为什么

将此警告事件的命名方式与所有其他警告事件保持一致。

缓解措施

使用新名称。（注意：事件 ID 号码未更改。）

阐明 API 的外键约束名称

[跟踪问题 #10730](#)

旧行为

在 EF Core 3.0 之前，外键约束名称被简单地称为“名称”。例如：

```
var constraintName = myForeignKey.Name;
```

新行为

从 EF Core 3.0 开始，外键约束名称现在被称为“约束名称”。例如：

```
var constraintName = myForeignKey.ConstraintName;
```

为什么

这样不仅可以此领域的命名方式保持一致，还阐明了这是外键约束的名称，不是定义外键所依据的列或属性名称。

缓解措施

使用新名称。

IRelationalDatabaseCreator.HasTables/HasTablesAsync 已公开

[跟踪问题 #15997](#)

旧行为

在 EF Core 3.0 推出前，这些方法受保护。

新行为

自 EF Core 3.0 起，这些方法是公共的。

为什么

EF 使用这些方法来确定数据库是否已创建但为空。这也适用于在 EF 外部确定是否要应用迁移。

缓解措施

更改任何重写的可访问性。

Microsoft.EntityFrameworkCore.Design 现在是 DevelopmentDependency 包

[跟踪问题 #11506](#)

旧行为

在 EF Core 3.0 推出前，Microsoft.EntityFrameworkCore.Design 是常规 NuGet 包，它的程序集可以由依赖它的项目引用。

新行为

自 EF Core 3.0 起, 它是 DevelopmentDependency 包。这意味着, 依赖项不会过渡流动到其他项目中, 并且你也无法再默认引用它的程序集。

为什么

此包仅用于设计时。已部署的应用程序不得引用它。让它成为 DevelopmentDependency 包强化了此建议。

缓解措施

如果需要引用此包来重写 EF Core 的设计时行为, 可以更新项目中的 PackageReference 项元数据。如果正在通过 Microsoft.EntityFrameworkCore.Tools 过渡引用此包, 必须向此包添加显式 PackageReference, 以更改它的元数据。

```
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.0.0">
  <PrivateAssets>all</PrivateAssets>
  <!-- Remove IncludeAssets to allow compiling against the assembly -->
  <!--<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>-->
</PackageReference>
```

SQLitePCL.raw 已更新为版本 2.0.0

[跟踪问题 #14824](#)

旧行为

Microsoft.EntityFrameworkCore.Sqlite 以前依赖 SQLitePCL.raw 版本 1.1.12。

新行为

我们已将包更新为依赖版本 2.0.0。

为什么

SQLitePCL.raw 版本 2.0.0 定目标到 .NET Standard 2.0。它以前定目标到 .NET Standard 1.1, 这就要求必须将可传递的包用作大型收尾, 才能正常运行。

缓解措施

SQLitePCL.raw 版本 2.0.0 包括一些重大变化。有关详细信息, 请参阅[发行说明](#)。

NetTopologySuite 已更新为版本 2.0.0

[跟踪问题 #14825](#)

旧行为

空间包以前依赖于 NetTopologySuite 的 1.15.1 版。

新行为

我们已将包更新为依赖版本 2.0.0。

为什么

NetTopologySuite 2.0.0 版旨在解决 EF Core 用户遇到的几个可用性问题。

缓解措施

NetTopologySuite 2.0.0 版包括一些重大更改。有关详细信息, 请参阅[发行说明](#)。

使用 Microsoft.Data.SqlClient 而不是 System.Data.SqlClient

[跟踪问题 #15636](#)

旧行为

Microsoft.EntityFrameworkCore.SqlServer 以前依赖 System.Data.SqlClient。

新行为

我们已将包更新为依赖 Microsoft.Data.SqlClient。

为什么

Microsoft.Data.SqlClient 是今后用于 SQL Server 的旗舰版数据访问驱动程序。而 System.Data.SqlClient 不再是开发的重点。一些重要功能(例如 Always Encrypted)仅可在 Microsoft.Data.SqlClient 上使用。

缓解措施

如果你的代码直接依赖于 System.Data.SqlClient, 则必须将其更改为 Microsoft.Data.SqlClient; 由于这两个包与 API 的兼容性都非常高, 因此这只是简单的包和命名空间更改。

必须配置多个不明确的自引用关系

[跟踪问题 #13573](#)

旧行为

具有多个自引用单向导航属性和匹配的 FK 的实体类型被错误配置为单个关系。例如:

```
public class User
{
    public Guid Id { get; set; }
    public User CreatedBy { get; set; }
    public User UpdatedBy { get; set; }
    public Guid CreatedById { get; set; }
    public Guid? UpdatedById { get; set; }
}
```

新行为

现已在模型构建中检测到此场景, 引发了一个指示模型不明确的异常。

为什么

生成的模型是不明确的, 在这种情况下可能会出现错误。

缓解措施

使用关系的完全配置。例如:

```
modelBuilder
    .Entity<User>()
    .HasOne(e => e.CreatedBy)
    .WithMany();

modelBuilder
    .Entity<User>()
    .HasOne(e => e.UpdatedBy)
    .WithMany();
```

DbFunction.Schema 为 null 或者空字符串将其配置为位于模型的默认架构中

[跟踪问题 #12757](#)

旧行为

配置了实为空字符串的架构的 DbFunction 被视为不带架构的内置函数。例如, 下述代码会将 `DatePart` CLR 函数映射到 SqlServer 上的 `DATEPART` 内置函数。

```
[DbFunction("DATEPART", Schema = "")]  
public static int? DatePart(string datePartArg, DateTime? date) => throw new Exception();
```

新行为

所有 DbFunction 映射都被视为映射到用户定义的函数。因此，空字符串值会将函数置于模型的默认架构中。这可能是通过 Fluent API `modelBuilder.HasDefaultSchema()` 显式配置的架构，否则为 `dbo`。

为什么

如果之前的架构为空，可以此将函数视为内置项，但此逻辑仅适用于内置函数不属于任何架构的 `SqlServer`。

缓解措施

手动配置 DbFunction 的转换，以将其映射到内置函数中。

```
modelBuilder  
    .HasDbFunction(typeof(MyContext).GetMethod(nameof(MyContext.DatePart)))  
    .HasTranslation(args => SqlFunctionExpression.Create("DatePart", args, typeof(int?), null));
```