

Entity Framework Core 3.0 中的新功能

2019/10/11 • [Edit Online](#)

以下列表包括 EF Core 3.0 的主要新功能。

EF Core 3.0 是一个主要版本，还包含多个[重大变更](#)，即可能对现有应用程序产生负面影响的 API 改进。

LINQ 检查

通过 LINQ，可使用所选 .NET 语言编写数据库查询，利用丰富的类型信息来提供 IntelliSense 和编译时类型检查。而 LINQ 还允许编写包含任意表达式（方法调用或操作）的不限数量的复杂查询。如何处理所有的这些组合是 LINQ 提供程序面临的主要挑战。

在 EF Core 3.0 中，我们重构了 LINQ 提供程序，以便能够将更多的查询模式转换为 SQL，可以在更多场景下生成高效的查询，并防止未检测出效率低的查询。新的 LINQ 提供程序是能够在未来版本中提供新的查询功能和性能改进并且无需中断现有应用程序和数据提供程序的基础。

客户端评估受限

最重大的设计变更是必须解决如何处理无法转换为参数或无法转换为 SQL 的 LINQ 表达式。

在前几个版本中，EF Core 明确查询的哪些部分可以转换为 SQL，并在客户端上执行查询的其余部分。在某些情况下，这种客户端执行是可取的，但在其他许多情况下，这可能会导致低效的查询。

例如，如果 EF Core 2.2 无法转换 `Where()` 调用中的谓词，则会执行一个不带筛选器的 SQL 语句，从数据库传输所有行，然后在内存中对其进行筛选：

```
var specialCustomers =
    context.Customers
        .Where(c => c.Name.StartsWith(n) && IsSpecialCustomer(c));
```

如果数据库包含少量行，那么这可能是可以接受的，但如果数据库包含大量行，则可能导致严重的性能问题甚至应用程序故障。

在 EF Core 3.0 中，我们限制客户端评估仅发生在顶级投影（基本上为最后一次调用 `Select()`）上。当 EF Core 3.0 检测到无法在查询中的任何其他位置转换的表达式时，它会引发运行时异常。

若要按照上一个示例的方式在客户端上评估谓词条件，开发人员现在必须将查询评估显式切换为 LINQ to Objects：

```
var specialCustomers =
    context.Customers
        .Where(c => c.Name.StartsWith(n))
        .AsEnumerable() // switches to LINQ to Objects
        .Where(c => IsSpecialCustomer(c));
```

有关这对现有应用程序的影响的详细信息，请参阅[重大变更文档](#)。

每个 LINQ 查询有一个 SQL 语句

3.0 中的另一个重大设计变更为：现在始终为每个 LINQ 查询生成一个 SQL 语句。在前面的版本中，在某些场景下我们通常生成多个 SQL 语句，例如在集合导航属性上转换 `Include()` 调用，以及转换遵循某些包含子查询的模式查询。尽管在某些场景下这种做法很方便，并且对于 `Include()` 而言，这甚至有助于避免通过线路发送冗余数据，但实现较为复杂，这会导致一些效率极为低下的行为（N+1 个查询），并且在一些情况下多个查询返回的数据可能不一致。

与客户端评估类似，若 EF Core 3.0 无法将 LINQ 查询转换为单个 SQL 语句，它将引发运行时异常。但我们已使 EF Core 能够借助联接将用于生成多个查询的许多常见模式转换为单个查询。

Cosmos DB 支持

通过 EF Core 的 Cosmos DB 提供程序，熟悉 EF 编程模型的开发人员可以轻松地将 Azure Cosmos DB 定为应用程序数据库目标。目标是利用 Cosmos DB 的一些优势，如全局分发、“始终开启”可用性、弹性可伸缩性和低延迟，甚至包括 .NET 开发人员可以更轻松地访问它。此提供程序将针对 Cosmos DB 中的 SQL API 启用大部分 EF Core 功能，如自动更改跟踪、LINQ 和值转换。

有关详细信息，请参阅 [Cosmos DB 提供程序文档](#)。

C#8.0 支持

EF Core 3.0 利用了 [C#8.0 中的一些新功能](#)：

异步流

异步查询结果现在使用新的标准 `IEnumerable<T>` 接口公开，并且可以通过 `await foreach` 使用。

```
var orders =
    from o in context.Orders
    where o.Status == OrderStatus.Pending
    select o;

await foreach(var o in orders.AsAsyncEnumerable())
{
    Process(o);
}
```

有关详细信息，请参阅 [C# 文档中的异步流](#)。

可为空引用类型

在代码中启用此新功能后，EF Core 将检查引用类型属性的为 Null 性，并将它应用到数据库中的相应列和关系：将按照不可为 Null 的引用类型的属性具有 `[Required]` 数据注释属性来处理它们。

例如，在下面的类中，将把标记为类型 `string?` 的属性配置为可选，而将 `string` 配置为必需：

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? MiddleName { get; set; }
}
```

有关详细信息，请参阅 EF Core 文档中的[处理可为 Null 的引用类型](#)。

截获数据库操作

EF Core 3.0 中的新截获 API 允许提供自定义逻辑，以便在发生低级别数据库操作时作为 EF Core 正常运行的一部分自动调用它们。例如，打开连接、提交事务或执行命令时。

与 EF 6 中的截获功能相似，借助侦听器，你可以在操作发生之前或之后拦截它们。在操作发生前截获它们时，将允许你绕过执行，并从截获逻辑提供备用结果。

例如，若要操作命令文本，可以创建 `IDbCommandInterceptor`：

```
public class HintCommandInterceptor : DbCommandInterceptor
{
    public override InterceptionResult ReaderExecuting(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult result)
    {
        // Manipulate the command text, etc. here...
        command.CommandText += " OPTION (OPTIMIZE FOR UNKNOWN)";
        return result;
    }
}
```

然后将它注册到 `DbContext`：

```
services.AddDbContext(b => b
    .UseSqlServer(connectionString)
    .AddInterceptors(new HintCommandInterceptor()));
```

数据库视图的反向工程

查询类型表示可从数据库读取但无法更新的数据，它已重命名为[无键实体类型](#)。由于它们非常适用于映射多数场景中的数据库视图，当执行数据库视图反向工程时，EF Core 现在将自动创建无键实体类型。

例如，利用 [dotnet ef 命令行工具](#)，可以键入：

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
```

此工具现在将自动为无键的视图和表构架类型：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Names>(entity =>
    {
        entity.HasNoKey();
        entity.ToView("Names");
    });

    modelBuilder.Entity<Things>(entity =>
    {
        entity.HasNoKey();
    });
}
```

与主体共享表的依赖实体现为可选项

自 EF Core 3.0 起，如果 `OrderDetails` 由 `Order` 拥有且显式映射到同一张表中，则它将可能添加 `Order` 而不添加 `OrderDetails`，并且除主键外的所有 `OrderDetails` 属性都将映射到不为 null 的列中。

查询时，如果其任意所需属性均没有值，或者它在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`。

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

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

.NET Core 上的 EF 6.3

这并不是真正的 EF Core 3.0 功能, 但我们认为这对当前客户而言非常重要。

我们知道许多现有应用程序使用以前版本的 EF, 而仅为了利用 .NET Core 将其移植到 EF Core 需要大量工作。为此, 我们已决定将最新版本的 EF 6 移植为在 .NET Core 3.0 上运行。

有关更多详细信息, 请参阅 [EF 6 中的新增功能](#)。

推迟的功能

最初计划为 EF Core 3.0 提供的一些功能已推迟到将来的版本:

- 在迁移部分忽略模型的功能, 跟踪编号为 [#2725](#)。
- 属性包实体, 由两个单独的问题跟踪: 关于共享类型实体的 [#9914](#) 和关于索引属性映射支持的 [#13610](#)。